

Machine Learning Methods

Ying Nian Wu

UCLA Department of Statistics and Data Science

Written with Claude

Figures taken from internet belong to original authors

Bibliographies to be added

[Table of Contents Clickable for Convenient Reading](#)

December 2024

Contents

1	Linear, Piecewise Linear and Logistic Regression	1
1.1	Simplest Linear Regression	2
1.2	Loss Function and Optimization	3
1.2.1	Loss Function	3
1.2.2	Finding the Minimum	4
1.2.3	Vector Representation	4
1.2.4	Geometric Interpretation	5
1.2.5	Regression Towards the Mean	5
1.3	Multiple Linear Regression	6
1.3.1	Data Representation	6
1.3.2	Model Formulation	6
1.3.3	Vector Notation	7
1.3.4	Loss Function and Gradient	7
1.3.5	Geometric Interpretation	7
1.3.6	General Solution Methods	8
1.4	Piecewise Linear Regression and Modern Interpolation Paradigm	9
1.4.1	Piecewise Linear Model	9
1.4.2	Overfitting and Regularization	10
1.4.3	Neural Network Interpretation	12
1.4.4	Implicit Regularization	13
1.4.5	Benefits of Overparameterization	14
1.4.6	Learning as Interpolatable Memorization	14
1.4.7	Double Descent Phenomenon	15
1.4.8	Benign Overfitting	17
1.4.9	Connection to Deep Learning	19
1.4.10	Reflection: Classical versus Interpolation Paradigms	20
1.5	Logistic Regression and Classification	22
1.5.1	Maximum Likelihood Perspective	22
1.5.2	Logistic Regression Model	23
1.5.3	Likelihood and Gradients	24
1.6	Gradient Descent	24
1.6.1	Generic Notation and Taylor Expansion	24
1.6.2	Geometric Interpretation	25
1.6.3	Basic Algorithm	25

1.6.4	Gradient Descent with Momentum	26
2	Multi-Layer Perceptron	29
2.0.1	Foundation	29
2.0.2	Architecture Development	30
2.0.3	Advanced Topics	30
2.0.4	Notable Features	30
2.0.5	Applications	31
2.1	Logistic Regression as Perceptron	31
2.1.1	Notation Comparison	31
2.1.2	Network Architecture	32
2.2	One Hidden Layer	32
2.2.1	One-dimensional Input	32
2.2.2	Two-dimensional Input	33
2.2.3	Maximum Likelihood Estimation	34
2.2.4	Chain Rule Backpropagation	35
2.2.5	Overparameterization and Learning Dynamics	36
2.3	General Multi-layer Perceptron	38
2.4	Backpropagation for General MLP	40
2.4.1	Scalar Intuition	41
2.4.2	Vector and Matrix Form	41
2.4.3	Detailed Component-wise Verification	42
2.5	Stochastic Gradient Descent	44
2.5.1	Mini-batch Structure	44
2.5.2	Gradient Computation	45
2.5.3	Update Rule	45
2.5.4	Stochasticity from Mini-batches	45
2.5.5	The Concept of Epochs	45
2.5.6	Single-Epoch Learning	46
2.5.7	Practical Considerations	46
2.5.8	Advantages of Mini-batch SGD	47
2.5.9	Single vs Multiple Epochs	47
2.6	Adam Optimizer	48
2.6.1	Recall: Momentum	48
2.6.2	Adaptive Learning Rates	48
2.6.3	Geometric Intuition	48
2.6.4	Adam Algorithm	48
2.6.5	Geometric Benefits	49
2.6.6	Benefits in Practice	49
2.7	Parameter Initialization	50
2.7.1	Basic Principles	50
2.7.2	Common Initialization Methods	50
2.7.3	Variance Analysis	51
2.7.4	Practical Guidelines	52
2.7.5	Impact on Training Dynamics	52

2.7.6	Initialization and Optimization Interplay	53
2.8	Multi-class Classification	53
2.8.1	Common Examples	53
2.8.2	Network Architecture	55
2.8.3	From Logit Scores to Probabilities	55
2.8.4	Loss Function Derivation	56
2.8.5	Gradient Derivation	56
2.8.6	Comparison: Binary vs Multi-class	56
2.8.7	Progressive Abstraction in Hidden Layers	57
2.9	Word Embedding	59
2.9.1	Model Structure	60
2.9.2	Interpretation of Embedding Matrix	60
2.9.3	Forward Pass Example	60
2.9.4	Gradient Derivation	61
2.9.5	Implementation Note	61
2.10	The Profound Idea of Embedding	62
2.10.1	From Sparse to Dense Representation	62
2.10.2	Thought Vectors	62
2.10.3	Vector Operations in Neural Networks	63
2.10.4	Properties of Embedding Space	63
2.10.5	Learning Embeddings	64
2.10.6	Impact on Deep Learning	64
2.11	Associative Memory	64
2.11.1	Model Structure	65
2.11.2	Interpretation	65
2.11.3	Associative Memory	66
2.11.4	Gradient Derivation	66
2.11.5	Learning Dynamics	66
2.11.6	Linear Associative Memory	67
2.11.7	Non-linear Associative Memory	68
2.12	Embedding for Recommender Systems	70
2.12.1	Basic Model	70
2.12.2	Learning from Observations	70
2.12.3	Neural Network Interpretation	70
2.12.4	Interpretation of User Embeddings	71
2.12.5	Mathematical Properties	71
2.12.6	Addiction Mechanism	71
2.12.7	Extension to Non-linear Models	72
2.13	Superposition	72
2.13.1	Beyond Individual Components	72
2.13.2	Basis Representation	72
2.13.3	Subspace Decomposition	73
2.13.4	Example: Barack Obama Embedding	73
2.13.5	Mathematical Properties	73
2.13.6	Implications	74

2.13.7	Neural Network Perspective	74
2.14	Normalization	75
2.14.1	RMS Normalization	75
2.14.2	Geometric Interpretation	75
2.14.3	Benefits for Loss Landscape	75
2.14.4	Error Correction Properties	76
2.14.5	Cosine Similarity	76
2.14.6	Application in Neural Networks	77
2.15	Dropout	77
2.15.1	Basic Mechanism	77
2.15.2	Testing Phase Adjustment	78
2.15.3	Advantages and Intuitions	78
2.15.4	Mathematical Analysis	79
2.15.5	Implementation Considerations	80
2.15.6	Fault Tolerance: RMS Norm vs Dropout Comparison	80
3	Convolutional Neural Networks	83
3.1	Neural Networks as Computer Programs	84
3.1.1	Recall: The Neural Language	84
3.1.2	Basic Operations in Neural Language	84
3.1.3	Neural Networks as Computer Programs	84
3.1.4	Programming with Vectors	85
3.1.5	Learning as Program Writing	85
3.1.6	Understanding Neural Programs	86
3.2	Computer Vision	86
3.2.1	Input Image Structure	86
3.2.2	Layers and Representations	87
3.2.3	Convolutional Layer Computation	89
3.2.4	Dimension Considerations	90
3.2.5	Inductive Bias in Convolution	91
3.2.6	Subsampling in Convolutional Layers	93
3.2.7	Fully Connected Layer Computation	93
3.2.8	Channel and Kernel View	94
3.2.9	1×1 Convolution	94
3.3	Backpropagation in CNN	95
3.3.1	Error Signal	95
3.3.2	Backprop through FC Layers	96
3.3.3	Backprop through Convolutional Layers	96
3.3.4	Backprop through Subsampling	96
3.3.5	Parallelization	96
3.3.6	Implementation Structure	97

4	Recurrent Neural Networks	99
4.1	Vector Evolution over Time	100
4.2	Next Word Prediction	101
4.2.1	Forward Computation	101
4.2.2	Meaning of Hidden States	101
4.2.3	Backpropagation Through Time	102
4.2.4	Gradient Vanishing	103
4.3	LSTM Innovation 1: Memory Stream	104
4.3.1	Memory Stream	105
4.3.2	Example: “I love machine learning”	105
4.3.3	Superposition in Memory Stream	106
4.3.4	Detailed Gradient Calculation	107
4.3.5	Memory Organization	109
4.4	LSTM Innovation 2: Multiplicative Gates	109
4.4.1	Key Innovations	109
4.4.2	Memory Update Mechanism	110
4.5	Multi-layer Recurrent Networks	111
4.5.1	Backpropagation Through Layers and Time	112
4.5.2	Fast Generation/Inference	114
4.5.3	Memory Streams	115
4.6	Residual Stream	115
4.6.1	Residual Stream Through Layers	115
4.6.2	Residual Stream in MLPs: A Computational Time Perspective	116
4.6.3	Computational Time Interpretation	118
4.6.4	Contrast with Real Time	118
4.6.5	Gradient Flow	119
4.6.6	Residual Stream as Assembly Line	119
4.6.7	Learning Simplification	119
4.6.8	Parallel with Memory Stream	120
4.6.9	Gradient Flow as Quality Control	120
4.7	Residual Stream as Learned Iterative Algorithm	121
4.7.1	Finite Step Iterative Algorithm	121
4.7.2	Algorithm Without Explicit Objectives	121
4.7.3	Learned Update Rule	122
4.7.4	Finite-Step Design	122
4.7.5	Advantages of Learning the Algorithm	123
4.7.6	Neural Programs with For Loops	123
4.8	Neural Programming Language	124
4.8.1	Neural network as a computer program	124
4.8.2	Data as the Programmer	124
4.8.3	Role of Residual Stream	124
4.8.4	Foundation of Digital Intelligence	125
4.9	Parameter Sharing Across Streams	126
4.9.1	Memory Stream and Residual Stream	126
4.9.2	Rationale for the Difference	126

4.9.3	Adding Recurrence/Residual to CNNs	127
4.9.4	Computational Structure	127
4.9.5	Advantages	128
4.10	Vanilla RNN vs Temporal CNN	128
4.10.1	With or Without Horizontal Recurrent Connections	128
4.10.2	Key Differences	129
4.11	State Space Models	130
4.11.1	Basic Formulation	130
4.11.2	Unrolled Form	130
4.11.3	Unifying Recurrent and Convolutional Views	131
4.11.4	Computational Advantages	131
4.12	Continuous-Time State Space Model	132
4.12.1	Memory Stream Form	132
4.12.2	Zero-Order Hold (ZOH) Discretization	132
4.13	Mamba: Selective State Space Model	133
4.13.1	Key Innovation	133
4.14	Quantum Mechanics as RNN	135
4.14.1	Basic Structure	136
4.14.2	Special Properties	136
4.14.3	Squared-Softmax and Born Rule	137
4.14.4	Norm Conservation in Quantum Measurement	137
4.14.5	Hidden Layer as Fundamental Reality	138
4.14.6	Interface to Classical Reality	138
4.14.7	The Role of the Observer	139
4.14.8	Classical Reality as Rendered Display	139
4.14.9	Philosophical Implications	139
5	Transformer and GPT	141
5.1	Embedding, Thought Vectors and Distributed Representations	142
5.1.1	Superposition Nature	142
5.1.2	Information Extraction	143
5.1.3	Properties	144
5.1.4	Neural Operations	144
5.1.5	Residual Stream: Building Superposition	145
5.1.6	Assembly Line Process	145
5.2	Transformer Residual Stream	145
5.2.1	Dual Retrieval Mechanism	146
5.2.2	Assembly Line Process	146
5.2.3	Two Forms of Retrieval	147
5.2.4	Complementary Nature	148
5.2.5	Attention Mechanism	149
5.2.6	Mixture of Experts	151
5.3	Complete Transformer Architecture	153
5.3.1	Token and Position Embeddings	153
5.3.2	Layer Processing	153

5.3.3	Output Generation	153
5.3.4	Backpropagation and Parallelization	154
5.4	Associative Memory	155
5.4.1	SVD as Memory Structure	155
5.4.2	MLP as Query Generator	156
5.4.3	Memory Cleaning	157
5.4.4	Memory Editing	157
5.4.5	Low-Rank Adaptation (LoRA)	158
5.4.6	Query-Key-Value Projection as Associative Memory	159
5.5	Reflection: A Matrix = A Thousand Rules	160
5.5.1	Matrix as Infinite Association Rules	160
5.5.2	Advantages over Discrete Systems	160
5.5.3	Learnability through Backpropagation	161
5.5.4	Compositional Learning	162
5.5.5	Implications	162
5.5.6	Counter Argument: The Power of Abstract Logic	162
5.6	Architectural Comparison	163
5.6.1	Bottom-up Architecture	163
5.6.2	Context Access	164
5.6.3	Memory Metaphor	164
5.6.4	Inference Process	165
5.6.5	Trade-offs	165
5.7	Original Transformer for Translation	166
5.7.1	Architecture Overview	166
5.7.2	Three Types of Attention	166
5.7.3	Encoder Matrix Implementation	167
5.7.4	Decoder Masked Attention	167
5.8	Transformer Family: Translation to BERT and GPT	168
5.8.1	Architectural Heritage	168
5.8.2	BERT (Bidirectional Encoder Representations from Transformers)	168
5.8.3	GPT (Generative Pre-trained Transformer)	169
5.8.4	Core Distinctions	169
5.9	Original Transformer Parameters	170
5.9.1	Core Architecture Parameters	170
5.9.2	Key Design Choices	170
5.10	GPT-3 175B Architecture	171
5.10.1	Key Parameters	171
5.10.2	Parameter Distribution	171
5.10.3	Computation Flow	171
5.10.4	Design Choices	172
5.11	Scaling Laws	172
5.11.1	Power Law Relationships	172
5.11.2	Optimal Allocation	173
5.11.3	Chinchilla Scaling	174
5.11.4	Implications	174

5.11.5	Example Scales	175
5.12	Two-Stage Training	175
5.12.1	Pre-training Stage	175
5.12.2	Instruction Fine-tuning	175
5.12.3	Training Process	176
5.12.4	Benefits of Two-Stage Approach	177
5.12.5	Example Instructions	177
5.13	Data Curation Pipeline	178
5.13.1	Pre-training Stages	178
5.13.2	Instruction Fine-tuning	178
5.13.3	RLHF Data	179
5.13.4	Quality Progression	179
5.13.5	Continuous Improvement	180
5.14	Reinforcement Learning from Human Feedback	180
5.14.1	Basic Concept	180
5.14.2	Reward Modeling	180
5.14.3	Bradley-Terry Model	182
5.14.4	Policy Gradient Fine-tuning	183
5.14.5	Comparison with Maximum Likelihood	185
5.15	Proximal Policy Optimization (PPO)	186
5.15.1	Importance Sampling Form	186
5.15.2	Motivation for Clipping	186
5.15.3	PPO Clipped Objective	186
5.15.4	Implementation Benefits	187
5.15.5	Understanding PPO	188
5.16	Vision Transformer (ViT)	189
5.16.1	Image to Sequence	190
5.16.2	Patch Embedding	190
5.16.3	Processing Architecture	190
5.16.4	Comparison with CNN	190
5.16.5	Computational Aspects	191
5.16.6	Practical Considerations	191
5.17	CLIP: Contrastive Language–Image Pretraining	192
5.17.1	Dual Encoder Architecture	192
5.17.2	Contrastive Loss	193
5.17.3	Understanding Contrastive Learning	193
5.17.4	Temperature Scaling	193
5.17.5	Training Process	194
5.17.6	Applications	194
6	Diffusion Model	195
6.1	Probability Preliminaries: Counting Population	196
6.1.1	Discrete Random Variables: Population Movement Between States . .	196
6.1.2	Continuous Random Variables: Population Distribution on a Line . .	198
6.2	Noising and Denoising: A Single Step	199

6.2.1	The Forward Noising Process	199
6.2.2	The Backward Denoising Process	199
6.2.3	Reversibility of the Noising Process	201
6.2.4	Why Score Reverses Noising	201
6.2.5	Stochastic Denoising and Deterministic Denoising	202
6.3	Trajectory-Based Data Augmentation	204
6.3.1	Motivation and Challenges	204
6.3.2	Trajectory-Based Approach	204
6.3.3	Learning the Generation Process	205
6.3.4	Generation Process	205
6.3.5	Comparison with Autoregressive Models	205
6.4	Simple Gaussian Trajectory Construction: Noising and Denoising	206
6.4.1	Forward Process Construction	206
6.4.2	Transition Probability	206
6.4.3	Terminal Distribution Analysis	206
6.4.4	Derivation of the Reserve Transition Distribution	207
6.4.5	Uniqueness of Gaussian	208
6.4.6	Necessity of Small Noise Variance	208
6.5	Score-Based Parametrization	209
6.5.1	Single Neural Network Parametrization	209
6.5.2	Learning the diffusion model	210
6.5.3	Generation Process	210
6.5.4	Alternative Loss: Predicting Clean Data	210
6.5.5	Noise Prediction	210
6.5.6	Generation Process with Noise Prediction	211
6.5.7	Scaling	211
6.5.8	UNet Parametrization of the Score Network	211
6.6	Variance Reduction via Trajectory Averaging	213
6.6.1	Multiple Trajectories Perspective	213
6.6.2	Variance-Reduced Loss with Conditional Mean	214
6.6.3	Deriving Alternative Loss via Conditional Mean	214
6.7	Connection to Denoising Auto-Encoder and Vincent Identity	215
6.7.1	Denoising Auto-Encoder	215
6.7.2	Proof of Vincent Identity	215
6.8	Noise Prediction Parameterization	216
6.8.1	Loss Function	216
6.8.2	Training Algorithm	217
6.8.3	Sampling Algorithm	217
6.9	Maximum Likelihood and Kullback-Leibler Divergence	218
6.9.1	General Setting	218
6.9.2	Extension to Trajectories	218
6.9.3	Trajectory Distributions	218
6.9.4	Learning the diffusion model	219
6.9.5	Connection to KL Divergence	220
6.9.6	Trajectory Distribution Factorization	220

6.9.7	KL Divergence Decomposition	220
6.9.8	Local KL Terms	221
6.9.9	Final Objective	221
6.10	Deterministic Sampling: $t - 2$ Reasoning	221
6.11	Continuous Time Analysis	222
6.11.1	Forward Process	222
6.11.2	Stochastic Differential Equation (SDE) Backward	223
6.11.3	Deterministic Ordinary Differential Equation (ODE) Backward	223
6.11.4	Understanding Continuous Time Through Movies	224
6.12	Stochastic Noising and Deterministic Denoising	224
6.12.1	Distribution Preservation	224
6.12.2	Intuitive Understanding	225
6.12.3	Langevin Dynamics for Equilibrium Sampling	225
6.12.4	Non-equilibrium Sampling	226
6.13	General Forward Process with Drift	226
6.13.1	Forward Process Analysis	226
6.13.2	Backward Processes	227
6.14	Random Drift Process	227
6.14.1	Process Comparison	227
6.14.2	Accumulated Variance Analysis	228
6.14.3	Deterministic Equivalence	228
6.14.4	Backward Process	228
6.15	Fokker-Planck Analysis	229
6.15.1	Test Function Perspective	229
6.15.2	SDE Analysis	229
6.15.3	ODE Analysis	230
6.15.4	SDE-ODE Equivalence	230
6.15.5	Random Drift Analysis	230
6.15.6	Extension to Multivariate Case	230
6.16	Flow Matching with Straight Trajectories	231
6.16.1	Design Principle	231
6.16.2	Non-Markovian Trajectory Data	231
6.16.3	Setup	232
6.16.4	Backward Process Analysis	232
6.16.5	Flow Matching Learning	233
6.16.6	Connection to Noise and Score Prediction	233
6.17	Variance Scheduling	234
6.17.1	Forward Process Construction	234
6.17.2	Deriving the Marginal Distribution	234
6.17.3	Training and Sampling	236
6.17.4	Forward Process SDE	236
6.17.5	Backward Processes	237
6.18	Applications of Diffusion Models	238
6.18.1	Text-to-Image Generation	238
6.18.2	Diffusion Transformer	239

7	VAE and GAN	241
7.1	Maximum Likelihood and KL-Divergence	242
7.1.1	Empirical Distribution and Log-likelihood	242
7.1.2	True Model Log-likelihood and Entropy	242
7.1.3	KL Divergence as Log-likelihood Gap	242
7.1.4	Information Geometric Interpretation	243
7.1.5	Implications	243
7.2	Deconvolution Network with Latent Space	243
7.2.1	Structured Latent Representation	243
7.2.2	Deconvolution Network Architecture	244
7.2.3	Training	244
7.2.4	Latent Space Interpolation	245
7.2.5	Applications	246
7.3	Latent Variable Models: From Effect to Cause	246
7.3.1	Data Augmentation with Latent Variables	246
7.3.2	Generative Model Structure	246
7.3.3	Manifold Learning Perspective	247
7.3.4	Historical Connection: Factor Analysis	247
7.4	From Marginal to Joint KL Divergence	247
7.4.1	Log-likelihood and KL Divergence	247
7.4.2	Extension to Complete Data	248
7.4.3	Key Decomposition	248
7.4.4	Two Forms of ELBO	249
7.4.5	Analysis of Gaps	250
7.5	Inference Model	251
7.5.1	From Data Augmentation to Learnable Inference	251
7.5.2	Joint Optimization	251
7.5.3	Evidence Lower Bound with Learnable Inference	252
7.5.4	Interpreting Form 1 of the ELBO	252
7.5.5	Interpreting Form 2 of the ELBO	253
7.5.6	Mode Covering versus Mode Seeking Behavior	254
7.5.7	Connection to EM Algorithm	255
7.6	Variational Autoencoder Implementation	256
7.6.1	Neural Network Parametrization	256
7.6.2	The Reparametrization Trick	257
7.6.3	Computing the ELBO	257
7.6.4	Training Algorithm	257
7.6.5	Practical Considerations	258
7.6.6	Generation and Reconstruction	258
7.7	Comparison with Diffusion Models	258
7.7.1	Latent Variable Structure	258
7.7.2	Key Distinction: Fixed vs Learned Inference	259
7.7.3	Theoretical Guarantees	259
7.7.4	Philosophical Perspective	259
7.8	Generative Adversarial Networks	260

7.8.1	Data Structure	260
7.8.2	Learning the Discriminator	261
7.8.3	Game-Theoretic Perspective	261
7.8.4	Implementation Form	262
7.8.5	Wasserstein GAN	262
7.8.6	Mode Collapse	263
8	Deep Reinforcement Learning	265
8.1	Theoretical Foundations of Sequential Decision Making	266
8.1.1	Basic Setup	267
8.1.2	Key Functions	267
8.1.3	Model-Based vs Model-Free Paradigms	268
8.2	Fundamental Theorems in Reinforcement Learning	268
8.2.1	Policy Gradient Theorem	268
8.2.2	Fundamental Relationships in Value-Based RL	269
8.2.3	Implications	271
8.2.4	Core Algorithm Derivations	271
8.2.5	Advanced Methods	272
8.3	The Game of Go	273
8.3.1	Game Complexity	273
8.3.2	Formal Game Definition	273
8.3.3	Rules and Gameplay	274
8.4	Neural Network Architecture	274
8.4.1	Policy Network	274
8.4.2	Value Network	275
8.5	Training Methodology	275
8.5.1	Supervised Learning of Policy Network	275
8.5.2	Reinforcement Learning of Policy Network	276
8.5.3	Training the Value Network	276
8.6	Progressive Introduction to Monte Carlo Tree Search	276
8.6.1	From Simple Policy to Look-ahead Search	276
8.6.2	Basic Monte Carlo Look-ahead	277
8.6.3	Advantage of Looking Ahead	278
8.6.4	Foundation for Full MCTS	279
8.6.5	Q-value Update on the Whole Branch	279
8.6.6	Policy-Guided Action Selection	282
8.6.7	Full MCTS with Dynamic Tree Growth	284
8.6.8	Complementary Roles of Policy and Value for Search	287
8.6.9	Value Network and Bootstrap Principle	289
8.7	From AlphaGo to AlphaGo Zero	291
8.7.1	Original AlphaGo Architecture	291
8.7.2	The Key Insight	292
8.7.3	The Natural Evolution	292
8.7.4	Birth of AlphaGo Zero	293
8.7.5	Why This Works	293

8.8	Reflections: System 1 and System 2	293
8.8.1	System 1 and System 2 in AlphaGo Zero	293
8.8.2	The Consciousness Parallel	294
8.8.3	Learning as Memorization	295
8.8.4	Primacy of Planning	295
8.8.5	Generalization and Transfer	296
8.9	Deep Q-Learning for Atari Games	296
8.9.1	The Atari Environment	297
8.9.2	Q-Learning Formulation	297
8.9.3	Key Components	297
8.9.4	Training Process	298
8.9.5	Contrast with AlphaGo	298
8.9.6	Practical Considerations	298
8.9.7	Q-Learning and MCTS: Shared Principles	299
8.10	Policy Gradient Methods for Atari Games	300
8.10.1	Core Idea	300
8.10.2	Policy Gradient Theorem	301
8.10.3	REINFORCE Algorithm	301
8.10.4	Variance Reduction	301
8.10.5	Practical Implementation	302
8.10.6	Comparison with Q-Learning	302
8.11	Value-Based versus Policy-Based Methods	302
8.11.1	Fundamental Differences	303
8.11.2	Key Properties	303
8.11.3	Learning Characteristics	304
8.11.4	Implementation Aspects	304
8.11.5	Practical Trade-offs	305
8.11.6	Empirical Results in Atari	305
8.11.7	Motivation for Hybrid Approaches	306
8.12	Actor-Critic Methods for Atari Games	306
8.12.1	Core Architecture	306
8.12.2	Advantage Estimation	307
8.12.3	Implementation for Atari	308
8.12.4	Key Advantages for Atari	308
8.12.5	Practical Considerations	309
8.12.6	Comparison to Other Methods	309
8.12.7	Proximal Policy Optimization (PPO)	310
8.12.8	Actor-Critic Implementation in PPO	311
8.13	Bootstrapping in Dense-Reward Settings	313
8.13.1	Core Bootstrap Concept	313
8.13.2	One-Step Bootstrap	313
8.13.3	Multi-Step Bootstrap	314
8.13.4	Why Bootstrap Works in Dense Rewards	314
8.13.5	Implementation Considerations	315
8.13.6	Success in Practice	316

8.13.7	Dense-Reward vs MCTS Bootstrapping	316
8.14	Temporal Difference Learning	318
8.14.1	The TD Learning Principle	318
8.14.2	Comparison with Other Methods	319
8.14.3	TD Learning Properties	320
8.14.4	Variants and Extensions	320
8.14.5	Connection to Other Concepts	321
8.15	On-Policy versus Off-Policy Learning	321
8.15.1	Fundamental Definitions	321
8.15.2	Mathematical Formulation	322
8.15.3	Algorithm Examples	322
8.15.4	Key Trade-offs	322
8.15.5	Implementation Considerations	323
8.15.6	Unified View	324
8.15.7	Application Examples	324
8.16	Dense versus Sparse Rewards	324
8.16.1	Reward Characteristics	324
8.16.2	Implications for Learning	325
8.16.3	Solution Approaches	326
8.16.4	Architectural Implications	326
8.17	Model-Based vs Model-Free Approaches	326
8.17.1	Model Definition	326
8.17.2	Analysis by Game Type	327
8.17.3	Algorithmic Approaches	327
8.17.4	Hybrid Approaches	328
8.17.5	Trade-offs Summary	329
8.18	Model Predictive Control (MPC)	329
8.18.1	Core Concept	329
8.18.2	Mathematical Formulation	330
8.18.3	Algorithm Structure	330
8.18.4	Key Advantages	331
8.18.5	Comparison to Other Methods	331
8.18.6	Unifying View: MPC and AlphaGo Planning	331
8.19	Planning versus Policy Approaches	333
8.19.1	Fundamental Distinction	333
8.19.2	Computational Properties	334
8.19.3	Information Usage	334
8.19.4	Decision Quality	335
8.19.5	Hybrid Approaches	335
8.19.6	Domain-Specific Considerations	336
8.19.7	Implementation Considerations	337
8.19.8	Future Trends	337
8.20	Relationship Between Planning and Control	338
8.20.1	Core Definitions and Distinctions	338
8.20.2	Mathematical Formulations	338

8.20.3	Key Distinctions	339
8.20.4	Model Predictive Control: A Bridge	340
8.20.5	Comparative Analysis	340
8.20.6	Modern Integration	341
8.20.7	Future Directions	342
8.21	Policy and Value Functions in Planning and Control	342
8.21.1	Fundamental Roles	342
8.21.2	Integration in Planning	343
8.21.3	Integration in Control	343
8.21.4	Hybrid Architectures	344
8.21.5	Learning Mechanisms	345
8.21.6	Implementation Considerations	345
8.21.7	Future Directions	346
8.22	Online versus Offline Reinforcement Learning	346
8.22.1	Fundamental Distinctions	346
8.22.2	Mathematical Formulation	347
8.22.3	Key Challenges	347
8.22.4	Modern Algorithms	348
8.22.5	Implementation Considerations	349
8.22.6	Applications	350
8.22.7	Future Directions	350
8.23	Summary	351
8.23.1	Core Components in Deep RL	351
8.23.2	Algorithm Classification	351
8.23.3	Key Trade-offs	352
8.23.4	Unified Learning Framework	352
8.23.5	Domain-Specific Insights	352
9	Trees and Boosting	355
9.1	Incremental Model Improvement: From Deep Learning to Trees	356
9.1.1	The Principle of Incremental Learning	356
9.1.2	Three Paradigms of Incremental Improvement	356
9.1.3	Geometric Interpretation	356
9.1.4	Common Mathematical Structure	356
9.1.5	The Role of Gradients	357
9.1.6	Looking Ahead	357
9.2	Decision Trees	357
9.2.1	A Motivating Example	357
9.2.2	Decision Rules and Tree Structure	358
9.2.3	The Concept of Purity	358
9.2.4	Splitting Criterion	359
9.2.5	From Classification to Regression	359
9.3	Regression Trees	359
9.3.1	Mathematical Framework	359
9.3.2	Optimization Problem	360

9.3.3	Recursive Binary Splitting	360
9.3.4	Split Selection Algorithm	360
9.3.5	Tree Growing Procedure	361
9.3.6	Statistical Properties	361
9.4	Least Squares Boosting	362
9.4.1	Basic Framework	362
9.4.2	Regularized Optimization	362
9.4.3	Tree Construction with Regularization	362
9.4.4	Extension to Weighted Least Squares	363
9.4.5	Complete Algorithm	363
9.4.6	Connection to XGBoost	363
9.5	XGBoost for Logistic Regression	364
9.5.1	The Logistic Model	364
9.5.2	Loss Function Analysis	364
9.5.3	Adding a New Tree	365
9.5.4	Geometric Interpretation	367
9.5.5	The Golf Analogy	367
9.5.6	Connection to Error Back-propagation	368
9.5.7	Tree Learning as Back-propagation	368
9.5.8	Connection to Iterative Reweighted Least Squares	369
9.6	Surrogate Loss Functions and Incremental Learning	371
9.6.1	The Role of Surrogate Losses	371
9.6.2	Gradient Descent as Surrogate Minimization	371
9.6.3	Boosting and Surrogate Losses	372
9.6.4	XGBoost’s Second-Order Surrogate	372
9.6.5	A Unified View Through Surrogate Functions	373
9.6.6	Design Principles for Surrogate Functions	373
9.6.7	Connection to Earlier Sections	374
9.7	The “Lazy” Nature of Boosting and Implicit Regularization	374
9.7.1	Gradient Flow and Function Space	374
9.7.2	The Principle of Least Action	375
9.7.3	Spectral Bias in Function Learning	375
9.7.4	Early Stopping as Complexity Control	375
9.7.5	Implicit Regularization Through Optimization	376
9.7.6	Comparison with Neural Networks	376
9.7.7	Practical Implications	376
9.8	AdaBoost	377
9.8.1	The Exponential Loss Framework	377
9.8.2	Properties of Exponential Loss	378
9.8.3	Forward Stagewise Additive Modeling	380
9.8.4	Optimal Base Classifier	380
9.8.5	Optimal Weight Coefficient	380
9.8.6	Weight Update Rule	380
9.8.7	The Complete Algorithm	381
9.8.8	Comparison with XGBoost	381

9.9	Random Forests	383
9.9.1	Ensemble Framework	383
9.9.2	Sources of Randomization	383
9.9.3	Tree Construction	384
9.9.4	Statistical Properties	384
9.9.5	Variable Importance Measures	385
9.9.6	Theoretical Results	385
9.9.7	Comparison with Boosting Methods	386
9.9.8	Implementation Considerations	386
10	Support Vector Machine	387
10.1	Primal Problem: Max Margin	388
10.1.1	The Geometric Intuition	388
10.1.2	The Separation Problem	388
10.1.3	Connection to Standard SVM Formulation	389
10.2	From Primal to Dual: MinMax = MaxMin	390
10.2.1	The Lagrangian Formulation	390
10.2.2	The Minimax Problem	390
10.2.3	Equivalence of Max-Min Lagrangian to Primal Problem	390
10.2.4	Saddle Point and Max-Min Equality	391
10.2.5	Game Theoretic Interpretation of Max-Min Equality	393
10.3	Dual Problem: Min Distance	395
10.3.1	Initial Dual Derivation	395
10.3.2	Geometric Interpretation via frontal points	396
10.3.3	The Distance Interpretation	396
10.3.4	The Minimum Distance Problem	397
10.3.5	Projections and Separation	397
10.3.6	Karush-Kuhn-Tucker (KKT) Conditions	398
10.4	Dual Coordinate Ascent	398
10.4.1	Dual Problem with $b = 0$	398
10.4.2	Coordinate-wise Optimization	398
10.4.3	Optimal Update	399
10.4.4	Algorithm	399
10.4.5	Implementation Details	399
10.5	The Kernel Trick	400
10.5.1	Motivation	400
10.5.2	Kernel Function	401
10.5.3	Kernelized Dual Problem	401
10.5.4	Common Kernel Functions	401
10.5.5	Mercer's Theorem	401
10.5.6	Kernelized Coordinate Descent	402
10.5.7	Implementation Considerations	402
10.5.8	Reproducing Kernel Hilbert Space	402
10.5.9	Example: Gaussian RBF Kernel	404
10.6	Soft Margin SVM	404

10.6.1	Motivation	404
10.6.2	Primal Problem	405
10.6.3	Lagrangian	405
10.6.4	KKT Conditions	406
10.6.5	Dual Problem	406
10.6.6	Support Vector Cases	406
10.6.7	Bias Term Computation	407
10.6.8	Model Selection	407
10.7	Sequential Minimal Optimization (SMO)	407
10.7.1	Problem Structure	407
10.7.2	Two-Variable Subproblem	408
10.7.3	Analytical Solution	408
10.7.4	Constraint Handling	408
10.7.5	Algorithm and Convergence	409
10.7.6	Connection to Coordinate Methods	409
10.7.7	Comparison of Incremental Learning Strategies	409
10.8	From Slack Variables to Hinge Loss	411
10.8.1	Three Major Loss Functions	411
10.8.2	Properties	412
10.8.3	Derivatives	412
10.8.4	Statistical Interpretation	413
10.8.5	Practical Considerations	413
10.9	A Unified View of Modern Learning Methods	413
10.9.1	General Framework	413
10.9.2	Hidden Layer Characteristics	414
10.9.3	Feature Construction	414
10.9.4	Learning Paradigms	415
10.9.5	Model Complexity Control	415
10.9.6	Advantages and Trade-offs	415
10.9.7	Practical Considerations	416
10.10	Model Complexity and Regularization	417
10.10.1	Fundamental Principle: Explaining Away Noise	417
10.10.2	Three Views of Complexity	417
10.10.3	Controlling Complexity	418
10.10.4	Unified Understanding	419
10.10.5	Theoretical Guarantees	419
10.10.6	Modern Perspectives	419
10.10.7	Practical Implications	419

Chapter 1

Linear, Piecewise Linear and Logistic Regression

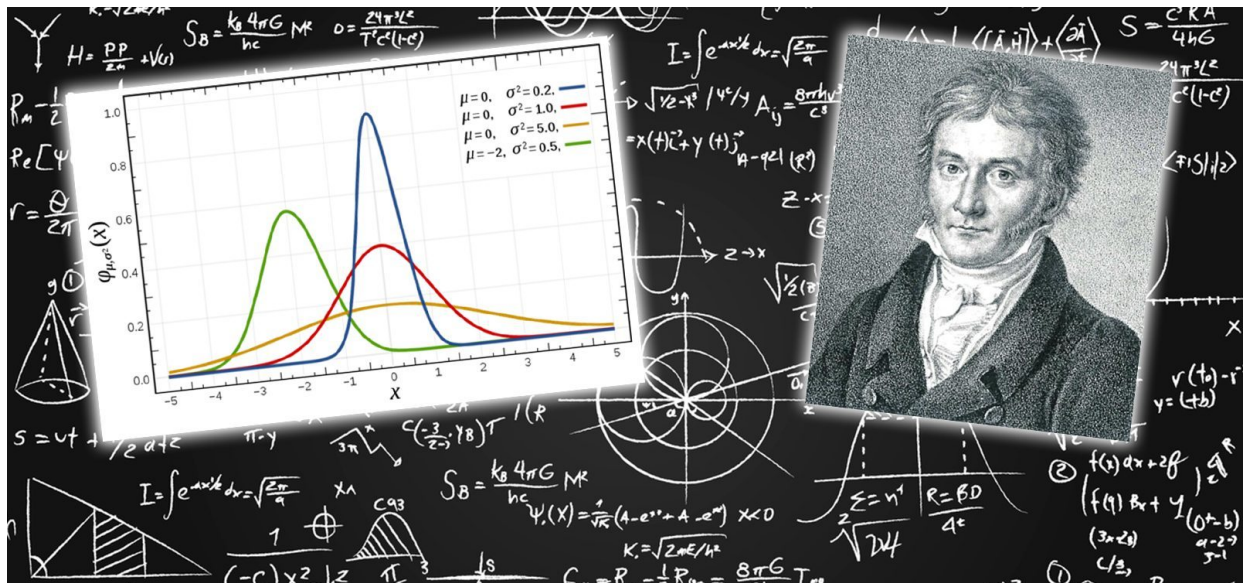


Figure 1.1: Gaussian paradigm: loss function (least squares), probabilistic formulation (Gaussian distribution), theoretical optimality (Gauss-Markov theorem), and empirical verification (planetoid Ceres)

Everything is regression.

— Jan deLeeuw

Founding chair of UCLA Department of Statistics and Data Science

Chapter Overview

This chapter introduces fundamental concepts of machine learning through increasingly sophisticated regression models. We begin with the simplest linear regression using father-son

height data, which serves as a gentle introduction to key ideas of prediction, loss function, and optimization.

We then extend to multiple linear regression, which forms the foundation of modern machine learning. This framework introduces vector representations, matrix operations, and geometric interpretations that will pervade throughout deep learning.

The chapter takes a significant turn with piecewise linear regression, which captures many essential aspects of deep learning:

- ReLU network interpretation of piecewise linear functions
- Overparameterization and interpolation phenomena
- Implicit regularization through gradient descent
- Double descent behavior where more parameters can lead to better generalization
- Balance between memorization and generalization

Finally, we study logistic regression for classification, which introduces:

- Non-linear activation function (sigmoid)
- Probabilistic interpretation of outputs
- Maximum likelihood estimation
- Non-trivial example of gradient descent optimization

This progression builds from simple to complex, revealing how modern deep learning emerges naturally from classical statistical concepts. The piecewise linear model serves as a bridge between traditional statistics and neural networks, while logistic regression introduces key nonlinear elements central to deep learning.

1.1 Simplest Linear Regression

Linear regression begins with training data consisting of paired observations. In our fundamental example, we study the relationship between fathers' and sons' heights, where each pair represents a training example indexed from 1 to n .

i	x_i (Father's Height)	y_i (Son's Height)
1	x_1	y_1
2	x_2	y_2
\vdots	\vdots	\vdots
i	x_i	y_i
\vdots	\vdots	\vdots
n	x_n	y_n

Table 1.1: Structure of height data

The model predicts a son's height using a simple linear formula $s_i = x_i\beta$, where:

- s_i represents the predicted height (linear score)
- x_i is the father's height (input)
- β is our model parameter to be learned

We measure the prediction error (residual) as $e_i = y_i - s_i$, which captures the difference between actual and predicted values.

1.2 Loss Function and Optimization

1.2.1 Loss Function

To evaluate our model's performance, we define a loss function that penalizes prediction errors:

$$L(\beta) = \frac{1}{2} \sum_{i=1}^n e_i^2 = \frac{1}{2} \sum_{i=1}^n (y_i - x_i\beta)^2 \quad (1.1)$$

The factor of $\frac{1}{2}$ simplifies our derivative calculations. When divided by n , this function is known as Mean Squared Error (MSE). The loss function forms a parabola with a unique minimum point $\hat{\beta}$.

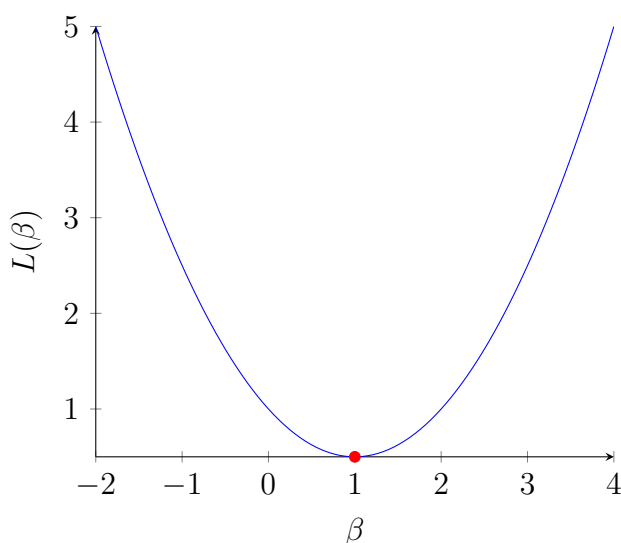


Figure 1.2: Loss function with minimum at $\hat{\beta}$

1.2.2 Finding the Minimum

To find the minimum of our loss function, we calculate its derivative using the chain rule:

$$\frac{dL}{d\beta} = \sum_{i=1}^n \frac{\partial L}{\partial e_i} \cdot \frac{\partial e_i}{\partial s_i} \cdot \frac{\partial s_i}{\partial \beta} \quad (1.2)$$

$$= \sum_{i=1}^n e_i \cdot (-1) \cdot x_i \quad (1.3)$$

$$= - \sum_{i=1}^n x_i e_i \quad (1.4)$$

$$= - \sum_{i=1}^n x_i (y_i - x_i \beta) \quad (1.5)$$

We can find the minimum in two ways:

- Gradient Descent (Iterative Method):
 - Start with an initial guess for β
 - Update using $\beta_{new} = \beta_{old} - \eta L'(\beta_{old})$
 - Use learning rate η (small positive number)
 - Repeat until convergence
- Closed-form Solution (Direct Method):
 - Set $L'(\beta) = 0$ and solve
 - Obtain

$$\hat{\beta} = \frac{\sum_{i=1}^n x_i y_i}{\sum_{i=1}^n x_i^2} \quad (1.6)$$

1.2.3 Vector Representation

We can express our data more compactly using vector notation:

Index	X (Fathers' Heights)	Y (Sons' Heights)
1	x_1	y_1
2	x_2	y_2
\vdots	\vdots	\vdots
n	x_n	y_n
Vector Form	$\mathbf{X} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$	$\mathbf{Y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$

Table 1.2: Vector representation of data

Our model now involves four key vectors:

- Input vector \mathbf{X} containing fathers' heights
- Output vector \mathbf{Y} containing sons' heights
- Prediction vector $\mathbf{X}\hat{\beta}$
- Error vector $\mathbf{e} = \mathbf{Y} - \mathbf{X}\hat{\beta}$

1.2.4 Geometric Interpretation

Geometrically, $\mathbf{X}\hat{\beta}$ represents the projection of \mathbf{Y} onto \mathbf{X} . The error vector \mathbf{e} is perpendicular to \mathbf{X} , and the angle θ indicates the alignment between \mathbf{X} and \mathbf{Y} .

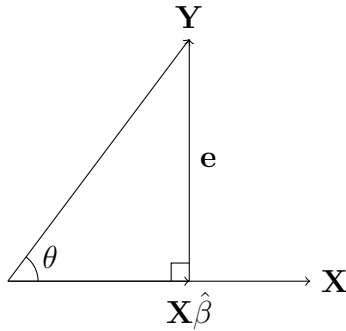


Figure 1.3: Geometric view of linear regression

1.2.5 Regression Towards the Mean

To better understand the regression behavior, we first normalize our data using:

- Mean calculations:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i, \quad \bar{y} = \frac{1}{n} \sum_{i=1}^n y_i \quad (1.7)$$

- Standard deviation calculations:

$$\sigma_x^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2, \quad \sigma_y^2 = \frac{1}{n} \sum_{i=1}^n (y_i - \bar{y})^2 \quad (1.8)$$

- Normalized data points:

$$\tilde{x}_i = \frac{x_i - \bar{x}}{\sigma_x}, \quad \tilde{y}_i = \frac{y_i - \bar{y}}{\sigma_y} \quad (1.9)$$

The normalized model $\tilde{y}_i = \beta \tilde{x}_i + \epsilon_i$ has several important properties:

- The regression coefficient equals the correlation: $\beta = \cos \theta$

- For positive correlation, β is always less than 1
- Predictions move toward the mean: $s_i = \tilde{x}_i\beta$

In modern usage, “regression” has evolved to encompass any prediction of y from x , regardless of whether it exhibits the original “toward mean” behavior, covering a wide range of predictive techniques.

1.3 Multiple Linear Regression

Multiple linear regression extends our analysis to handle multiple input variables. Consider predicting a child’s height using three predictors: father’s height, mother’s height, and child’s gender.

1.3.1 Data Representation

The input variables are:

- x_{i1} : Father’s height
- x_{i2} : Mother’s height
- x_{i3} : Child’s gender (0 for female, 1 for male)

Our target variable is y_i , representing the child’s height. The data structure takes the following form:

Observation	Father’s Height (x_{i1})	Mother’s Height (x_{i2})	Gender (x_{i3})	Child’s Height (y_i)
1	x_{11}	x_{12}	x_{13}	y_1
2	x_{21}	x_{22}	x_{23}	y_2
\vdots	\vdots	\vdots	\vdots	\vdots
n	x_{n1}	x_{n2}	x_{n3}	y_n

Table 1.3: Multiple linear regression data structure

1.3.2 Model Formulation

The basic model extends the simple linear regression by including multiple terms:

$$s_i = \beta_0 + x_{i1}\beta_1 + x_{i2}\beta_2 + x_{i3}\beta_3 \quad (1.10)$$

Here, s_i represents the predicted height for child i , β_0 is the intercept term, and $\beta_1, \beta_2, \beta_3$ are the coefficients for each predictor.

1.3.3 Vector Notation

We can express this model more compactly using vector notation. Initially:

$$\mathbf{x}_i = \begin{pmatrix} x_{i1} \\ x_{i2} \\ x_{i3} \end{pmatrix}, \quad \boldsymbol{\beta} = \begin{pmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{pmatrix} \quad (1.11)$$

With model: $s_i = \beta_0 + \mathbf{x}_i^\top \boldsymbol{\beta}$

For even more concise notation, we use extended vectors:

$$\mathbf{x}_i = \begin{pmatrix} 1 \\ x_{i1} \\ x_{i2} \\ x_{i3} \end{pmatrix}, \quad \boldsymbol{\beta} = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{pmatrix} \quad (1.12)$$

This gives us the simplified model: $s_i = \mathbf{x}_i^\top \boldsymbol{\beta}$

1.3.4 Loss Function and Gradient

The error for each observation is:

$$e_i = y_i - s_i = y_i - \mathbf{x}_i^\top \boldsymbol{\beta} \quad (1.13)$$

Leading to the loss function:

$$L(\boldsymbol{\beta}) = \frac{1}{2} \sum_{i=1}^n e_i^2 = \frac{1}{2} \sum_{i=1}^n (y_i - \mathbf{x}_i^\top \boldsymbol{\beta})^2 \quad (1.14)$$

Using the chain rule, we calculate partial derivatives:

$$\frac{\partial L}{\partial \beta_k} = \sum_{i=1}^n \frac{\partial L}{\partial e_i} \cdot \frac{\partial e_i}{\partial s_i} \cdot \frac{\partial s_i}{\partial \beta_k} = - \sum_{i=1}^n e_i x_{ik} \quad (1.15)$$

Setting these derivatives to zero gives us the normal equations:

$$\sum_{i=1}^n e_i x_{ik} = 0, \quad \text{for } k = 0, 1, 2, 3 \quad (1.16)$$

1.3.5 Geometric Interpretation

The geometric interpretation extends to multiple dimensions with several key vectors:

- \mathbf{X}_j : Column vector of $(x_{ij}, i = 1, \dots, n)$
- \mathbf{Y} : Column vector of $(y_i, i = 1, \dots, n)$
- $\hat{\mathbf{Y}}$: Projection of \mathbf{Y} onto span of \mathbf{X}_j

- \mathbf{e} : Error vector ($e_i, i = 1, \dots, n$)

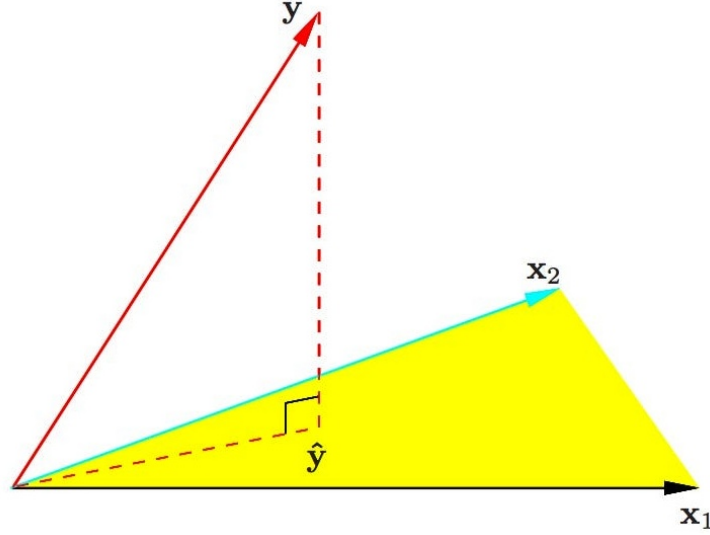


Figure 1.4: Least squares regression as projection

The error vector \mathbf{e} remains perpendicular to all predictor vectors \mathbf{X}_j , extending our simple regression geometry to multiple dimensions.

1.3.6 General Solution Methods

For a general case with p variables, we define:

$$\mathbf{x}_i = \begin{pmatrix} x_{i1} \\ x_{i2} \\ \vdots \\ x_{ip} \end{pmatrix}, \quad \boldsymbol{\beta} = \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_p \end{pmatrix} \quad (1.17)$$

The gradient vector takes the form:

$$L'(\boldsymbol{\beta}) = \begin{pmatrix} \frac{\partial L}{\partial \beta_1} \\ \vdots \\ \frac{\partial L}{\partial \beta_p} \end{pmatrix} = - \sum_{i=1}^n \mathbf{x}_i e_i \quad (1.18)$$

We can solve this optimization problem using two methods:

1. Gradient Descent:

$$\boldsymbol{\beta}^{(t+1)} = \boldsymbol{\beta}^{(t)} + \eta \sum_{i=1}^n \mathbf{x}_i (y_i - \mathbf{x}_i^\top \boldsymbol{\beta}^{(t)}) \quad (1.19)$$

2. Closed-form Solution:

$$\sum_{i=1}^n \mathbf{x}_i (y_i - \mathbf{x}_i^\top \boldsymbol{\beta}) = \mathbf{0} \quad (1.20)$$

$$\sum_{i=1}^n \mathbf{x}_i y_i = \sum_{i=1}^n \mathbf{x}_i \mathbf{x}_i^\top \boldsymbol{\beta} \quad (1.21)$$

$$\hat{\boldsymbol{\beta}} = \left(\sum_{i=1}^n \mathbf{x}_i \mathbf{x}_i^\top \right)^{-1} \sum_{i=1}^n \mathbf{x}_i y_i \quad (1.22)$$

In matrix form, we can write:

$$\mathbf{X} = \begin{pmatrix} \mathbf{x}_1^\top \\ \mathbf{x}_2^\top \\ \vdots \\ \mathbf{x}_n^\top \end{pmatrix} = (\mathbf{X}_1, \dots, \mathbf{X}_j, \dots, \mathbf{X}_p), \quad \mathbf{Y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \quad (1.23)$$

Leading to the normal equations and OLS solution:

$$\mathbf{X}^\top \mathbf{X} \boldsymbol{\beta} = \mathbf{X}^\top \mathbf{Y} \quad (1.24)$$

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{Y} \quad (1.25)$$

For this solution to exist, we require:

- $\mathbf{X}^\top \mathbf{X}$ must be invertible
- Columns of \mathbf{X} must be linearly independent
- No severe multicollinearity among predictors

1.4 Piecewise Linear Regression and Modern Interpolation Paradigm

Non-linear regression extends our modeling framework beyond simple linear relationships. We'll explore this through piecewise linear functions, discussing overfitting, regularization, and connections to neural networks.

1.4.1 Piecewise Linear Model

The basic model takes a one-dimensional raw input x_i^{raw} and transforms it using a set of break points $b_1, \dots, b_j, \dots, b_p$. At each break point, we create new features using the ReLU (rectified linear unit) function:

$$x_{ij} = \max(0, x_i^{raw} - b_j) \quad (1.26)$$

Our prediction then becomes:

$$s_i = \beta_0 + \sum_{j=1}^p x_{ij}\beta_j \quad (1.27)$$

where β_j represents the change in slope at break point b_j . The $\max(0, \cdot)$ function ensures features only become active when x_i^{raw} exceeds the break point. We assume the design points $\{b_j\}_{j=1}^p$ are designed and equally spaced over the range of x :

$$b_j = x_{\min} + j\Delta, \quad \Delta = \frac{x_{\max} - x_{\min}}{p + 1} \quad (1.28)$$

i	x_i^{raw}	x_{i1}	x_{i2}	\dots	x_{ip}	y_i
1	x_1^{raw}	$\max(0, x_1^{raw} - b_1)$	$\max(0, x_1^{raw} - b_2)$	\dots	$\max(0, x_1^{raw} - b_p)$	y_1
2	x_2^{raw}	$\max(0, x_2^{raw} - b_1)$	$\max(0, x_2^{raw} - b_2)$	\dots	$\max(0, x_2^{raw} - b_p)$	y_2
\vdots	\vdots	\vdots	\vdots	\ddots	\vdots	\vdots
n	x_n^{raw}	$\max(0, x_n^{raw} - b_1)$	$\max(0, x_n^{raw} - b_2)$	\dots	$\max(0, x_n^{raw} - b_p)$	y_n

Table 1.4: Data structure for piecewise linear regression

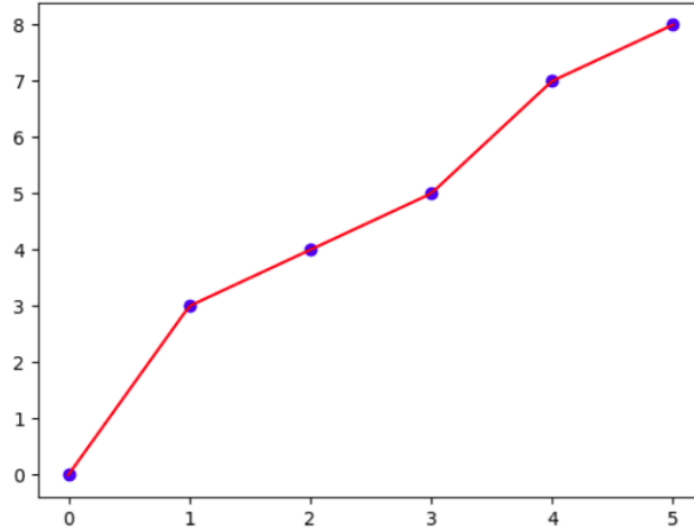


Figure 1.5: Piecewise linear relationship

1.4.2 Overfitting and Regularization

To prevent overfitting, we employ two main regularization strategies:

1. L2 Regularization (Ridge):

- Adds penalty term $\lambda \sum_{j=1}^p \beta_j^2$
- Loss function becomes: $L(\boldsymbol{\beta}) = \frac{1}{2} \sum_{i=1}^n (y_i - s_i)^2 + \lambda \sum_{j=1}^p \beta_j^2$
- Results in smoother curves and shrinks coefficients toward zero

2. L1 Regularization (Lasso):

- Uses penalty term $\lambda \sum_{j=1}^p |\beta_j|$
- Loss function becomes: $L(\boldsymbol{\beta}) = \frac{1}{2} \sum_{i=1}^n (y_i - s_i)^2 + \lambda \sum_{j=1}^p |\beta_j|$
- Creates sparse solutions by selecting fewer break points

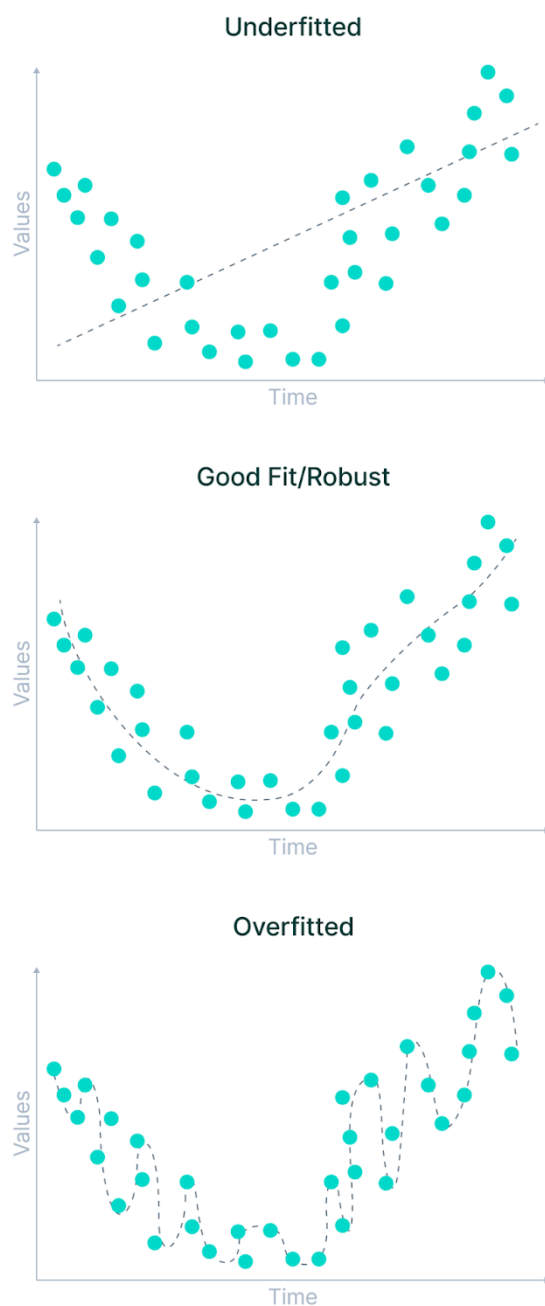


Figure 1.6: Comparison of exact vs. regularized interpolation

1.4.3 Neural Network Interpretation

The piecewise linear model can be interpreted as a simple neural network with three layers:

- Input layer: Contains the raw input x^{raw}
- Hidden layer: Applies ReLU activation $\max(\cdot, 0)$ to create features

- ReLU: Rectified Linear Unit is a commonly used nonlinear transformation in neural network
- Output layer: Computes weighted sum for final prediction

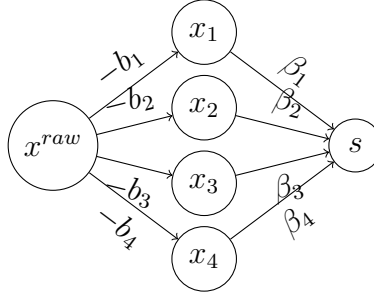


Figure 1.7: Neural network representation

1.4.4 Implicit Regularization

Gradient descent provides natural regularization through its optimization behavior:

- Gradient form: $\nabla L(\beta) = -\sum_{i=1}^n \mathbf{x}_i e_i$
- Update rule: $\beta^{(t+1)} = \beta^{(t)} + \eta \sum_{i=1}^n \mathbf{x}_i e_i$
- Linear combination property: $\hat{\beta} = \sum_{i=1}^n c_i \mathbf{x}_i$

The solution has the minimum L2 norm property among all interpolating solutions. We can prove this as follows:

- Let $\tilde{\beta}$ be any interpolating solution
- Define $\Delta = \tilde{\beta} - \hat{\beta}$
- Note that $\mathbf{x}_i^\top \Delta = 0$ for all i
- By the Pythagorean theorem: $\|\tilde{\beta}\|^2 = \|\hat{\beta}\|^2 + \|\Delta\|^2$
- Therefore $\|\tilde{\beta}\|^2 \geq \|\hat{\beta}\|^2$

That is, gradient descent is a lazy algorithm that travels the minimal distance from the initial point to a solution. This laziness actually provides implicit regularization and may prevent overfitting.

1.4.5 Benefits of Overparameterization

Overparameterization provides several advantages:

- Provides additional degrees of freedom
- Creates a more favorable and smoother optimization landscape
- Makes it easier to find global minima

The implicit regularization effects include:

- Finding minimum L2 norm solutions
- Producing smoother functions
- Improving generalization
- Natural bias toward simpler solutions

1.4.6 Learning as Interpolatable Memorization

In low-noise settings, learning becomes memorization through smooth interpolation:

- Perfect memorization:

$$\mathbf{x}_i^\top \hat{\boldsymbol{\beta}} = y_i \quad (\text{exact fitting})$$

- Smooth interpolation between memorized points:
 - Infinite possible interpolating functions exist
 - Gradient descent selects smoothest interpolant
 - Minimum norm solution: $\hat{\boldsymbol{\beta}} = \arg \min_{\boldsymbol{\beta}} \{\|\boldsymbol{\beta}\|^2 : \mathbf{x}_i^\top \boldsymbol{\beta} = y_i\}$

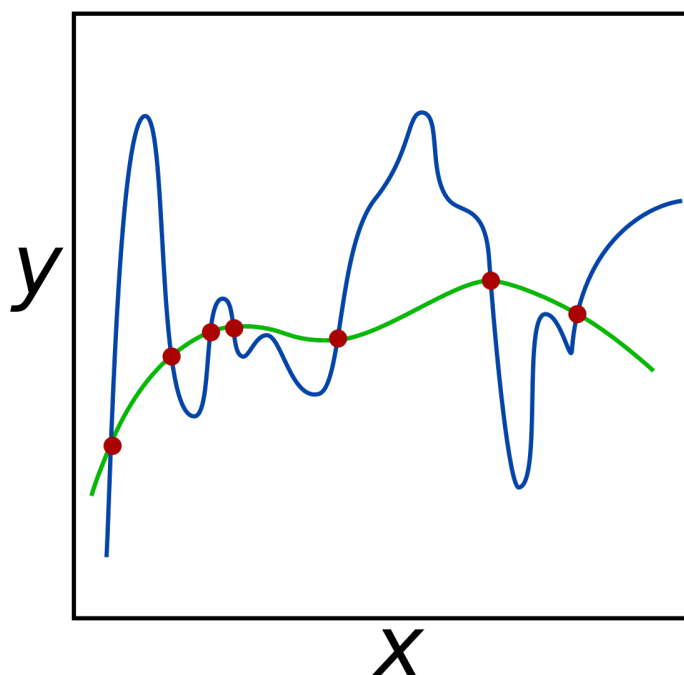


Figure 1.8: Smooth interpolation (blue) versus oscillatory interpolation (red) between memorized points

Success relies on:

- Clean, low-noise data enabling reliable memorization
- Sufficient overparameterization for flexible interpolation
- Implicit bias toward smooth connections between memorized points

This view reframes learning from noise-robust estimation to interpolatable memorization with smooth generalization between points.

1.4.7 Double Descent Phenomenon

The model's behavior exhibits three distinct regimes:

Underparameterized Regime ($p < n$)

When the number of break points is less than observations:

- Cannot achieve zero training loss
- Fitted curve is smooth but misses data points
- Training and test errors decrease with p
- MSE dominated by bias term

Interpolation Threshold ($p \approx n$)

As p approaches n :

- Zero training loss becomes possible
- Fitted curve may show zig-zag patterns
- Test error typically peaks
- Especially when observed x_i don't align with break points

The zig-zag effect occurs because:

$$s(x) = \beta_0 + \sum_{j=1}^p \beta_j \max(0, x - b_j) \quad (\text{piecewise linear}) \quad (1.29)$$

$$y_i = s(x_i) \quad (\text{interpolation constraint}) \quad (1.30)$$

$$\text{but } x_i \neq b_j \text{ typically} \quad (\text{misalignment}) \quad (1.31)$$

Overparameterized Regime ($p \gg n$)

As we further increase break points:

- Maintains zero training loss
- Fitted curve becomes increasingly smooth
- Test error decreases again
- Implicit regularization takes effect

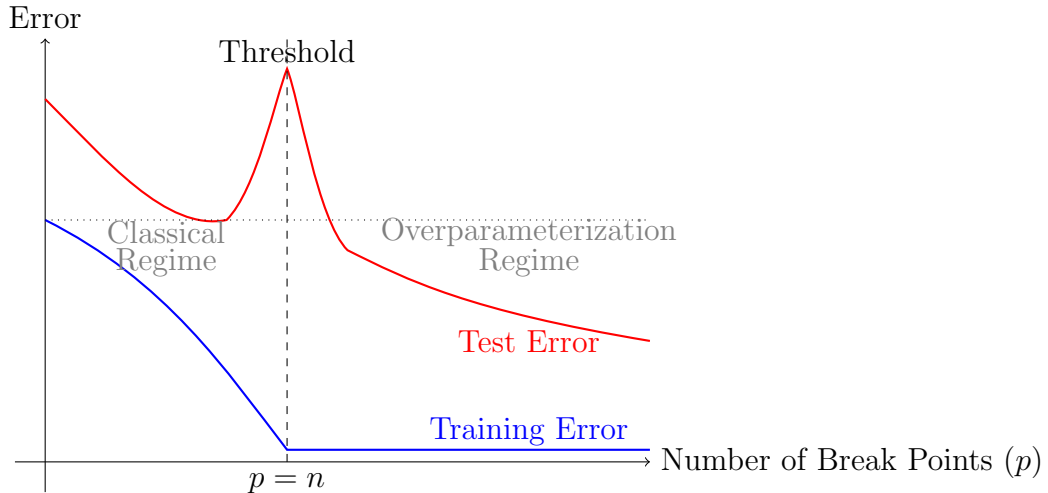


Figure 1.9: Double descent phenomenon: The test error (red) starts above training error (blue) and shows a second descent in the overparameterized regime, eventually achieving lower error than the classical minimum. The training error monotonically decreases to zero at the interpolation threshold.

The smoothing effect in overparameterization occurs because:

- More break points provide finer resolution
- Gradient descent finds minimum norm solution:

$$\|\beta\|^2 = \min\{\|\tilde{\beta}\|^2 : y_i = s(x_i) \text{ for all } i\} \quad (1.32)$$

$$\Rightarrow \text{smoother interpolation} \quad (1.33)$$

- Adjacent break points interact more closely:
 - Smaller spacing between break points
 - no much misalignment with x values
 - More gradual slope changes
 - Better continuity properties

This phenomenon illustrates that:

- More parameters can lead to better generalization
- Interpolation doesn't necessarily mean overfitting
- Implicit regularization plays crucial role
- Sweet spot may lie beyond classical wisdom

1.4.8 Benign Overfitting

The phenomenon of benign overfitting occurs when a model perfectly fits training data (zero training error) yet still generalizes well to test data. This seemingly paradoxical behavior challenges classical statistical wisdom and helps explain the success of modern overparameterized models.

Mathematical Characterization

Consider our piecewise linear model with $p \gg n$ break points:

$$s(x) = \beta_0 + \sum_{j=1}^p \beta_j \max(0, x - b_j) \quad (\text{model}) \quad (1.34)$$

$$y_i = s(x_i) \quad (\text{perfect fit}) \quad (1.35)$$

$$\mathbb{E}[(s(x) - y)^2] \approx \sigma^2 \quad (\text{good generalization}) \quad (1.36)$$

Benign overfitting occurs when:

- Training error reaches zero: $\sum_{i=1}^n (y_i - s(x_i))^2 = 0$
- Test error approaches optimal rate: $\|s - s^*\|_{L^2} = O(\sigma \sqrt{\log n/n})$
- Solution maintains smoothness: $\|\beta\|_2 = O(1)$

Conditions for Benign Overfitting

Several key factors enable benign overfitting:

1. Data Structure:
 - Low-dimensional signal in high-dimensional space
 - Rapidly decaying eigenspectrum of feature covariance
 - Strong correlation among features
2. Noise Characteristics:
 - Low noise level relative to signal strength
 - Noise primarily in directions orthogonal to signal
 - Bounded noise variance: $\mathbb{E}[\epsilon^2] \leq \sigma^2$
3. Model Properties:
 - Sufficient overparameterization: $p \gg n$
 - Implicit regularization from optimization
 - Local adaptivity of piecewise linear functions

Role of Implicit Regularization

Gradient descent plays a crucial role through two mechanisms:

1. Minimum Norm Solution:

$$\hat{\beta} = \arg \min_{\beta} \{\|\beta\|_2 : y_i = s(x_i) \text{ for all } i\} \quad (1.37)$$

2. Early Stopping Effect:

- Initially fits low-frequency components
- Progressively captures higher frequencies
- Natural balance between fit and smoothness

Connection to Double Descent

Benign overfitting helps explain the double descent phenomenon:

- Interpolation regime allows perfect memorization
- Overparameterization enables smooth interpolation
- Implicit regularization prevents harmful overfitting
- Test error continues decreasing beyond interpolation threshold

This relationship is captured by:

$$\text{Test Error} = \underbrace{\text{Estimation Error}}_{\downarrow \text{with } p} + \underbrace{\text{Approximation Error}}_{\downarrow \text{with } p} + \underbrace{\text{Optimization Error}}_{\approx 0 \text{ for } p \gg n} \quad (1.38)$$

Practical Implications

Understanding benign overfitting leads to several insights:

- More parameters can improve generalization
- Perfect training accuracy isn't necessarily harmful
- Traditional bias-variance trade-off needs revision
- Early stopping may be unnecessary with sufficient overparameterization

These principles guide modern deep learning practice:

- Use more parameters than strictly necessary
- Train to zero training error when possible
- Rely on implicit rather than explicit regularization
- Focus on architecture design over regularization

This modern perspective fundamentally changes how we think about model complexity and generalization, suggesting that the classical bias-variance trade-off may be too pessimistic in many practical scenarios.

1.4.9 Connection to Deep Learning

Our piecewise linear model directly connects to modern deep learning through several key aspects:

ReLU Network Interpretation

The model is equivalent to a simple ReLU network:

$$s(x) = \beta_0 + \sum_{j=1}^p \beta_j \max(0, x - b_j) \quad (\text{piecewise linear}) \quad (1.39)$$

$$= \text{Linear}(\text{ReLU}(\text{Linear}(x))) \quad (\text{single hidden layer}) \quad (1.40)$$

This structure generalizes to deep networks:

- ReLU activation creates piecewise linearity
- Multiple layers compose piecewise linear functions
- Break points learned instead of pre-specified

Overparameterization and Memorization

Modern deep learning operates in heavily overparameterized regime:

$$\text{parameters} \gg \text{data points} \quad (\text{overparameterized}) \quad (1.41)$$

$$\text{loss} \approx 0 \quad (\text{perfect memorization}) \quad (1.42)$$

Success depends on:

- Enough parameters to memorize training data
- Gradient descent finding smooth interpolation
- Low-noise, structured training data

Learning as Interpolation

Deep learning succeeds by:

- Memorizing clean training data perfectly
- Interpolating smoothly between memorized points
- Using overparameterization for flexibility
- Relying on optimization's implicit regularization

This view fundamentally shifts from:

- (Underfitting vs overfitting) to (memorization vs. interpolation)
- Parameter counting to implicit smoothness
- Statistical estimation to interpolatable memorization

1.4.10 Reflection: Classical versus Interpolation Paradigms

Nature of Scientific Understanding

The classical paradigm aligns with fundamental scientific principles:

- Newton's laws ($F = ma$):
 - Remarkably simple mathematical form
 - Vast explanatory power
 - True out-of-domain generalization
- Formal logic:
 - Simple deductive rules
 - Universal reasoning principles
 - Domain-independent validity

Limitations of Interpolation

The modern interpolation paradigm faces fundamental constraints:

- Domain Limitation:

$$s(x) = \beta_0 + \sum_{j=1}^p \beta_j \max(0, x - b_j) \quad \text{valid only for } x \in [x_{\min}, x_{\max}] \quad (1.43)$$

- No Extrapolation:
 - Cannot predict beyond training range
 - No principled behavior outside domain
 - Lacks theoretical underpinning for extension
- Complexity versus Simplicity:
 - May miss simple underlying patterns
 - Trades interpretability for flexibility
 - Complex representation of simple phenomena

The success of overparameterized models in specific domains should not overshadow the enduring value of seeking simple, fundamental principles that characterize natural phenomena. True scientific understanding may require bridging the gap between these paradigms, combining the flexibility of modern learning with the profound simplicity of classical physics and logic.

Historical Parallel: From Ptolemy to Deep Learning

The tension between interpolation and fundamental principles has a remarkable historical precedent. Ptolemy’s epicycle model, perhaps the earliest instance of overparameterization, used cycles upon cycles to predict planetary motion:

$$r(t) = \sum_{n=1}^N R_n e^{i\omega_n t} \quad (\text{Fourier decomposition}) \quad (1.44)$$

This approach mirrors modern deep learning:

- Epicycles as basis functions for universal approximation
- Hierarchical structure similar to neural networks
- Perfect interpolation through overparameterization

In contrast, Newton’s simple $F = ma$ explained the same phenomena with remarkable parsimony. This enabled Gauss to predict the position of planetoid Ceres using just seven observations—a triumph of fundamental principles over pure interpolation. Von Neumann later captured this tension: “With four parameters I can fit an elephant, and with five I can make him wiggle his trunk.” This historical parallel challenges us to consider whether modern deep learning, despite its practical success, might sometimes miss simpler underlying patterns in its pursuit of perfect interpolation.

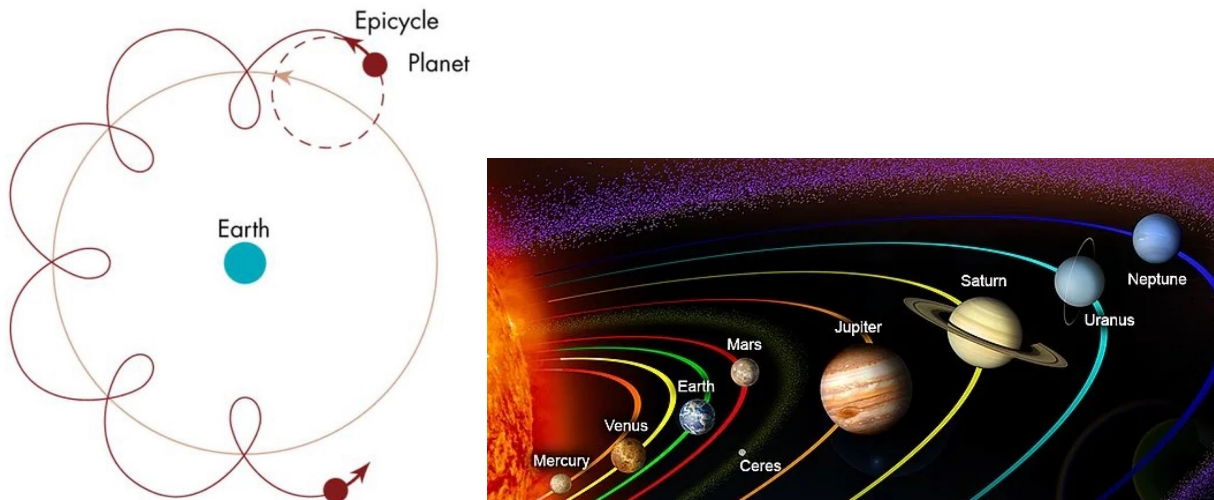


Figure 1.10: Left: Ptolemy's epicycle model assumes Earth is the center of Solar system. Right: Newton's model assumes Sun to be the center, and Ceres is a planetoid.

1.5 Logistic Regression and Classification

Before introducing logistic regression, let's briefly review multiple linear regression. In this setting, we work with multiple input variables and a single output variable:

Observation	x_{i1}	x_{i2}	\cdots	x_{ip}	y_i
1	x_{11}	x_{12}	\cdots	x_{1p}	y_1
2	x_{21}	x_{22}	\cdots	x_{2p}	y_2
\vdots	\vdots	\vdots	\ddots	\vdots	\vdots
n	x_{n1}	x_{n2}	\cdots	x_{np}	y_n

Table 1.5: Multiple linear regression data structure

The model equation is $s_i = \sum_{j=1}^p x_{ij}\beta_j = \mathbf{x}_i^\top \boldsymbol{\beta}$, where $\mathbf{x}_i = (x_{i1}, \dots, x_{ip})^\top$ is the input vector and $\boldsymbol{\beta} = (\beta_1, \dots, \beta_p)^\top$ contains the parameters.

1.5.1 Maximum Likelihood Perspective

From a probabilistic viewpoint, we assume:

$$p(y_i | s_i) \sim \mathcal{N}(s_i, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - s_i)^2}{2\sigma^2}\right) \quad (1.45)$$

The likelihood function for all observations is:

$$\prod_{i=1}^n p(y_i | s_i) = \prod_{i=1}^n \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - s_i)^2}{2\sigma^2}\right) \quad (1.46)$$

Taking the log-likelihood:

$$J = \sum_{i=1}^n \left[-\frac{1}{2} \log(2\pi\sigma^2) - \frac{(y_i - s_i)^2}{2\sigma^2} \right] \quad (1.47)$$

Maximum likelihood estimation seeks $\hat{\boldsymbol{\theta}} = \arg \max_{\boldsymbol{\theta}} J$. This is equivalent to minimizing the sum of squared errors, as $L = -J$ reduces to minimizing $\sum_{i=1}^n (y_i - s_i)^2$.

1.5.2 Logistic Regression Model

Logistic regression extends this framework to binary classification, predicting class probabilities rather than continuous values. We use the Bernoulli distribution $p(y_i|s_i) \sim \text{Bernoulli}(p_i)$ with the sigmoid function:

$$p_i = \text{sigmoid}(s_i) = \frac{e^{s_i}}{1 + e^{s_i}} \quad (1.48)$$

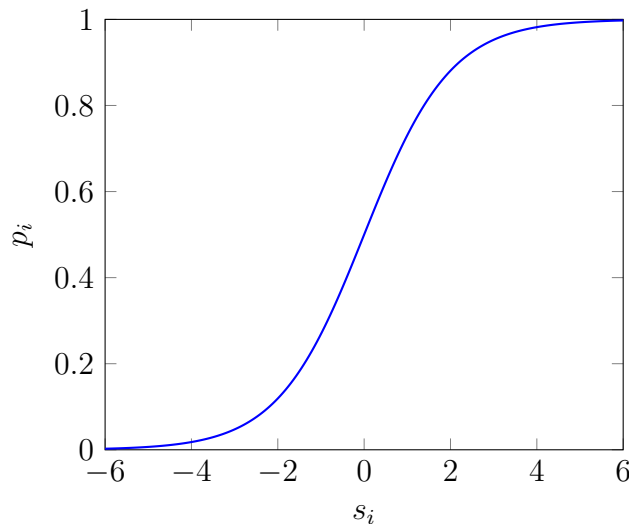


Figure 1.11: Sigmoid function

This gives us class probabilities:

- $p(y_i = 1|s_i) = p_i = \frac{e^{s_i}}{1+e^{s_i}}$
- $p(y_i = 0|s_i) = 1 - p_i = \frac{1}{1+e^{s_i}}$

The logit transformation provides an interpretation:

$$s_i = \log \left(\frac{p_i}{1 - p_i} \right) = \text{logit}(p_i) \quad (1.49)$$

1.5.3 Likelihood and Gradients

For a single observation, the likelihood is:

$$p(y_i|s_i) = \frac{e^{y_i s_i}}{1 + e^{s_i}} \quad (1.50)$$

Taking the log:

$$\log p(y_i|s_i) = y_i s_i - \log(1 + e^{s_i}) \quad (1.51)$$

The gradients are:

- $\frac{\partial \log p(y_i|s_i)}{\partial s_i} = y_i - p_i$
- $\frac{\partial J}{\partial \beta_k} = \sum_{i=1}^n (y_i - p_i) \cdot x_{ik}$
- Vector form: $J'(\beta) = \sum_{i=1}^n \mathbf{x}_i (y_i - p_i)$

This connects to linear regression through error terms:

- Linear regression: $e_i = y_i - s_i$
- Logistic regression: $e_i = y_i - p_i$

Both share the gradient form $J'(\beta) = \sum_{i=1}^n \mathbf{x}_i e_i$, allowing similar optimization techniques.

1.6 Gradient Descent

Gradient descent is a fundamental optimization algorithm widely used in machine learning. We'll focus on its geometric interpretation.

1.6.1 Generic Notation and Taylor Expansion

Consider a d -dimensional vector \mathbf{x} and small change vector $\Delta \mathbf{x}$:

$$\mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_k \\ \vdots \\ x_d \end{pmatrix}, \quad \Delta \mathbf{x} = \begin{pmatrix} \Delta x_1 \\ \vdots \\ \Delta x_k \\ \vdots \\ \Delta x_d \end{pmatrix} \quad (1.52)$$

Using first-order Taylor expansion:

$$f(\mathbf{x} + \Delta \mathbf{x}) \doteq f(\mathbf{x}) + \sum_{k=1}^d \frac{\partial f}{\partial x_k} \Delta x_k \quad (1.53)$$

$$= f(\mathbf{x}) + \langle f'(\mathbf{x}), \Delta \mathbf{x} \rangle \quad (1.54)$$

Where the gradient is:

$$f'(\mathbf{x}) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_k} \\ \vdots \\ \frac{\partial f}{\partial x_d} \end{pmatrix} \quad (1.55)$$

Using inner product properties:

$$f(\mathbf{x} + \Delta\mathbf{x}) \doteq f(\mathbf{x}) + \|f'(\mathbf{x})\| \|\Delta\mathbf{x}\| \cos \theta \quad (1.56)$$

1.6.2 Geometric Interpretation

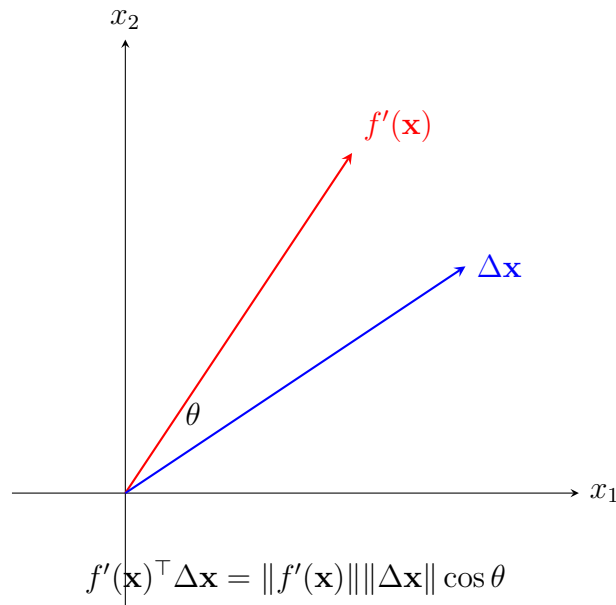


Figure 1.12: Geometric interpretation of gradient descent

The gradient $f'(\mathbf{x})$ points in the steepest ascent direction, while $-f'(\mathbf{x})$ gives the steepest descent direction. The gradient is always perpendicular to the function's contours.

1.6.3 Basic Algorithm

Algorithm 1 Gradient Descent

- 1: Initialize \mathbf{x}_0
 - 2: Choose learning rate $\eta > 0$
 - 3: **for** $t = 0, 1, 2, \dots$ until convergence **do**
 - 4: $\mathbf{x}_{t+1} = \mathbf{x}_t - \eta f'(\mathbf{x}_t)$
 - 5: **end for**
-

Key factors affecting convergence:

- Learning rate η (too small: slow convergence; too large: overshooting)
- Function curvature (high curvature causes oscillation)
- Initial point \mathbf{x}_0 (critical for non-convex functions)

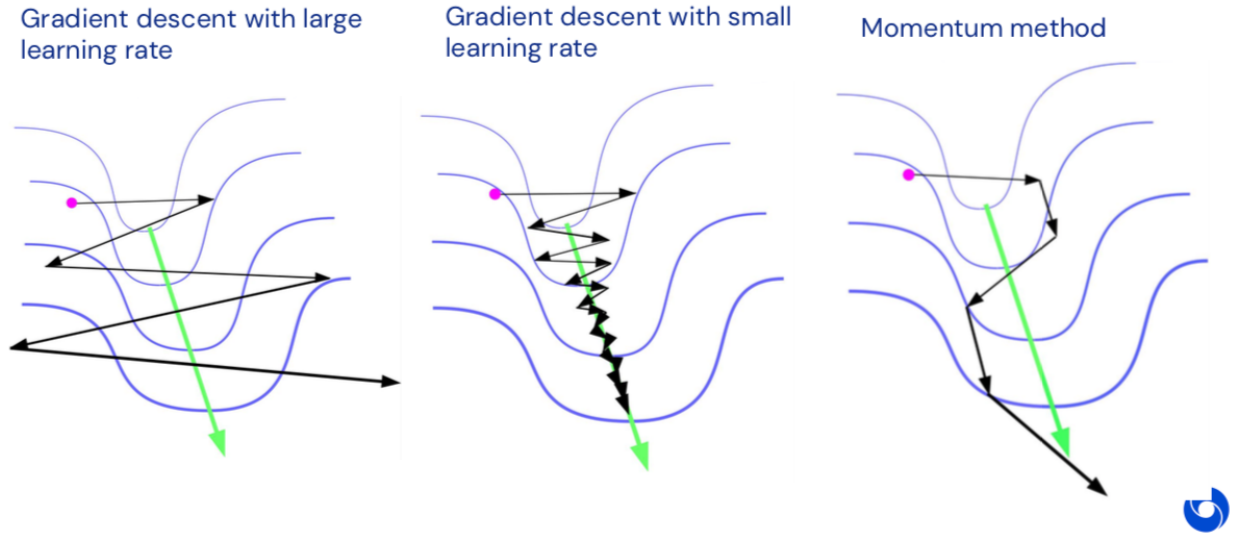


Figure 1.13: Gradient descent behavior with different learning rates and curvatures

1.6.4 Gradient Descent with Momentum

Algorithm 2 Gradient Descent with Momentum

- 1: Initialize $\mathbf{x}_0, \mathbf{v}_0 = \mathbf{0}$
 - 2: Choose learning rate $\eta > 0$ and momentum coefficient $\gamma \in [0, 1)$
 - 3: **for** $t = 0, 1, 2, \dots$ until convergence **do**
 - 4: $\mathbf{v}_t \leftarrow \gamma \mathbf{v}_{t-1} + f'(\mathbf{x}_t)$
 - 5: $\mathbf{x}_{t+1} \leftarrow \mathbf{x}_t - \eta \mathbf{v}_t$
 - 6: **end for**
-

The velocity term follows an exponential moving average:

$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + f'(\mathbf{x}_t) \quad (1.57)$$

$$= \gamma^2 \mathbf{v}_{t-2} + \gamma f'(\mathbf{x}_{t-1}) + f'(\mathbf{x}_t) \quad (1.58)$$

$$= \sum_{i=0}^t \gamma^i f'(\mathbf{x}_{t-i}) \quad (1.59)$$

The momentum coefficient γ has several interpretations:

- Physical: $1 - \gamma$ acts as friction
- Statistical: γ weights past gradients
- Memory: γ determines history length

Momentum provides several advantages:

- Faster convergence ($O(1/t^2)$ vs. $O(1/t)$)
- Reduced oscillation in high curvature regions
- Better escape from local minima
- Natural step size adaptation

Chapter 2

Multi-Layer Perceptron

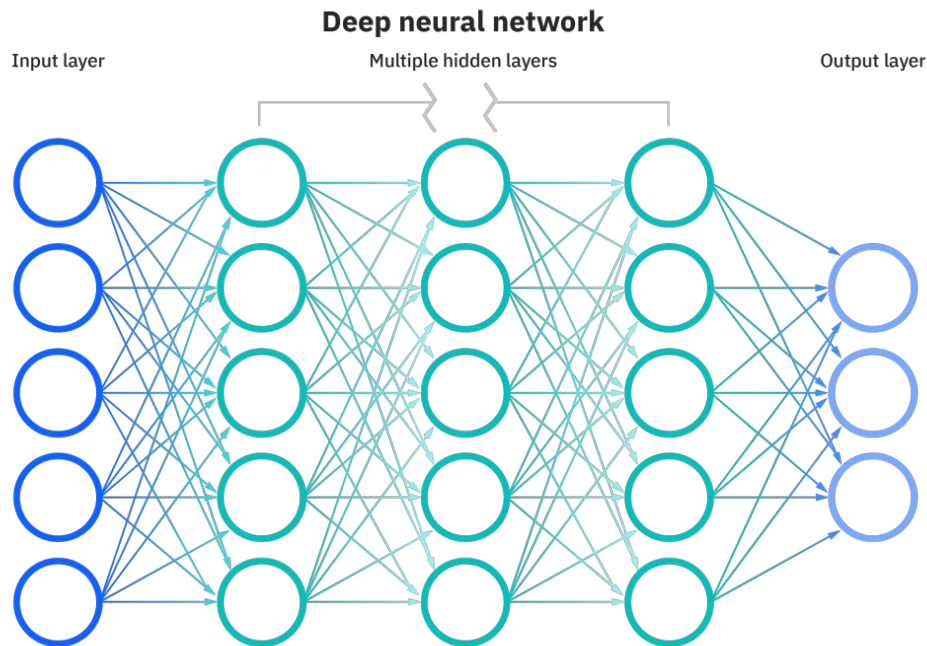


Figure 2.1: Multi-layer perceptron

Chapter Overview

This chapter presents a comprehensive study of Multi-Layer Perceptron (MLP), progressing from fundamental concepts to advanced applications. The material is organized as follows:

2.0.1 Foundation

The chapter begins by establishing the connection between logistic regression and the perceptron model, demonstrating how neural networks naturally extend from logistic regression.

This foundational approach provides a clear conceptual bridge from classical statistical methods to modern neural architectures.

2.0.2 Architecture Development

The architectural concepts are developed in three main stages:

- Single hidden layer networks and their properties
- Progression to general multi-layer architectures
- Detailed explanation of backpropagation for both simple and complex networks

2.0.3 Advanced Topics

The chapter covers several sophisticated aspects of MLPs:

- Multi-class classification implementation and theory
- Word embeddings and their mathematical foundations
- Associative memory and information storage mechanisms
- The concept of superposition in neural representations
- Normalization techniques, with emphasis on RMS normalization
- Dropout as a regularization strategy
- Optimization methods, including:
 - Stochastic Gradient Descent (SGD)
 - Adam optimizer

2.0.4 Notable Features

The chapter is characterized by:

- Rigorous mathematical derivations and proofs
- Detailed diagrams and visual representations
- Practical implementation guidelines
- Connections between theory and real-world applications

2.0.5 Applications

The theoretical concepts are illustrated through several practical applications:

- Natural language processing through word embeddings
- Recommender system design and implementation
- Multi-class classification tasks
- Associative memory systems

This structure effectively bridges theoretical foundations with practical implementations, making the material accessible for both understanding the mathematical underpinnings of MLPs and implementing them in practice.

2.1 Logistic Regression as Perceptron

2.1.1 Notation Comparison

In logistic regression, we write:

$$s_i = \mathbf{x}_i^\top \boldsymbol{\beta} + \beta_0 \quad (2.1)$$

In neural network notation, we write:

$$s_i = \mathbf{x}_i^\top \mathbf{w} + b \quad (2.2)$$

where:

- \mathbf{w} corresponds to $\boldsymbol{\beta}$: weights/coefficients
- b corresponds to β_0 : bias term

The probability is then computed as:

$$p_i = \sigma(s_i) = \frac{1}{1 + e^{-s_i}} \quad (2.3)$$

2.1.2 Network Architecture

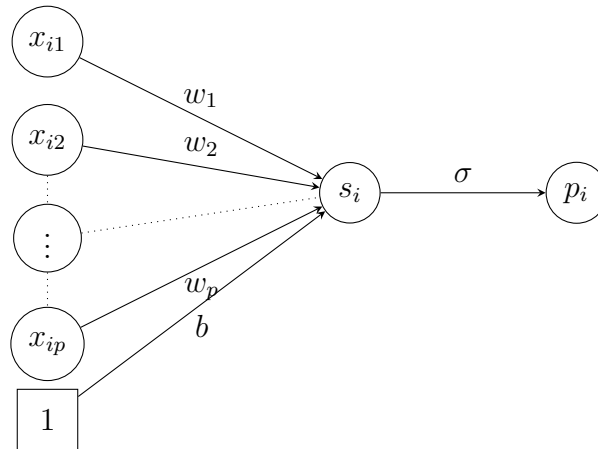


Figure 2.2: One-layer neural network representation of logistic regression

In this architecture:

- Input nodes represent features x_{ij}
- Weights w_j correspond to coefficients β_j
- Bias term b corresponds to intercept β_0
- Sigmoid activation σ transforms score s_i to probability p_i

2.2 One Hidden Layer

The one hidden layer section examines the fundamental building block of modern neural networks through a progressive study of architectures. Beginning with one-dimensional input, it shows how ReLU activations create piecewise linear functions. This extends to two-dimensional inputs where each hidden unit creates a “fold” in the input space, collectively forming piecewise linear surfaces. The section demonstrates that despite its simplicity, this architecture can approximate complex functions through the interplay of linear transformations and non-linear activations. Maximum likelihood estimation provides the training framework, while analysis of overparameterization reveals how gradient descent with zero initialization offers implicit regularization, allowing the network to adapt its complexity to the data while maintaining good generalization properties.

For clarity, we’ll drop the subscript i and describe a generic training example (x, y) or (\mathbf{x}, y) .

2.2.1 One-dimensional Input

The network structure for 1D input:

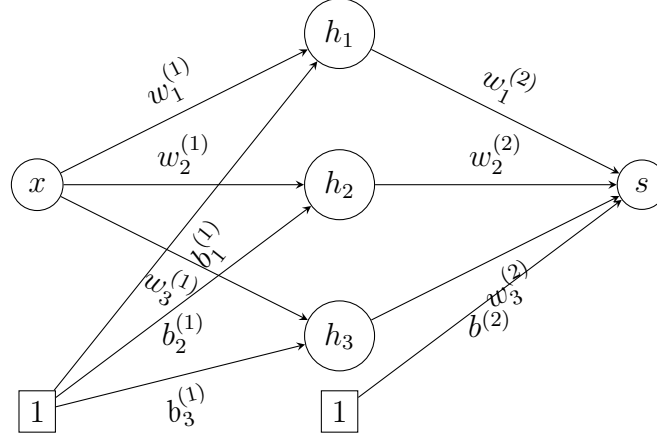


Figure 2.3: One-hidden-layer network with 1D input

For each hidden unit:

$$h_k = \text{ReLU}(w_k^{(1)}x + b_k^{(1)}) = \max(0, w_k^{(1)}x + b_k^{(1)}) \quad (2.4)$$

This is equivalent to our previous non-linear regression model:

$$\text{Previous model: } s = \beta_0 + \sum_j \beta_j \max(0, x - b_j) \quad (2.5)$$

$$\text{Current model: } s = b^{(2)} + \sum_k w_k^{(2)} \max(0, w_k^{(1)}x + b_k^{(1)}) \quad (2.6)$$

$s = f(x)$ is a piecewise linear function.

2.2.2 Two-dimensional Input

The network structure for 2D input:

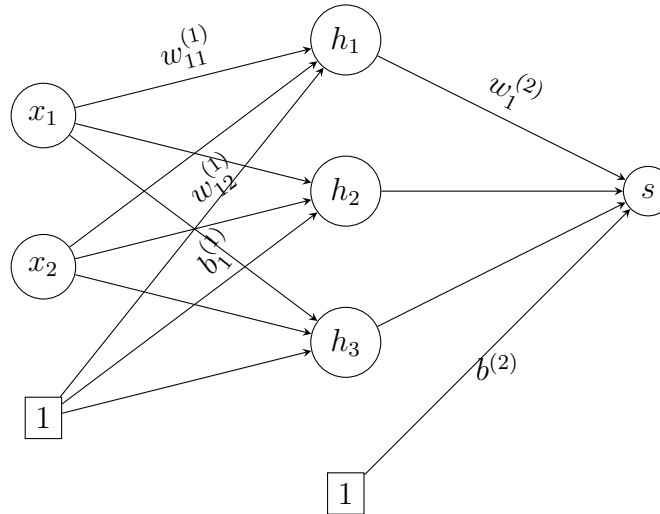


Figure 2.4: One-hidden-layer network with 2D input

The network structure is shown in Figure 2.4. For each hidden unit:

$$h_k = \max(0, w_{k1}^{(1)}x_1 + w_{k2}^{(1)}x_2 + b_k^{(1)}) \quad (2.7)$$

The output is:

$$s = b^{(2)} + \sum_k w_k^{(2)}h_k \quad (2.8)$$

Each hidden unit h_k creates a “fold” in the input space along the line:

$$w_{k1}^{(1)}x_1 + w_{k2}^{(1)}x_2 + b_k^{(1)} = 0 \quad (2.9)$$

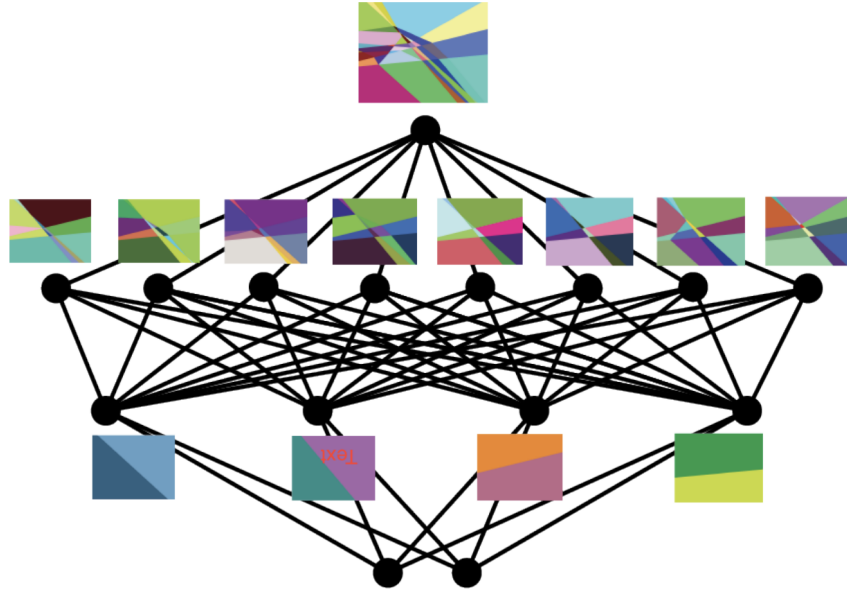


Figure 2.5: Each hidden unit creates a “fold”.

The resulting function $s = f(\mathbf{x})$ is a piecewise linear surface created by these folds. See Figure 2.5.

2.2.3 Maximum Likelihood Estimation

We start with the log-likelihood objective:

$$J = \log p(y|s) \quad (2.10)$$

For regression (Gaussian likelihood):

$$\frac{\partial J}{\partial s} = y - s = \text{error} \quad (2.11)$$

For classification (Bernoulli likelihood):

$$\frac{\partial J}{\partial s} = y - p = \text{error} \quad (2.12)$$

where $p = \sigma(s)$

In both cases, the error signal is the difference between the target and prediction, which is then backpropagated through the network to compute gradients for all parameters.

2.2.4 Chain Rule Backpropagation

Let's derive the gradients step by step using the chain rule. For clarity, let's denote $\frac{\partial J}{\partial s} = e$ (the error signal).

Second Layer Gradients

For the second layer weights:

$$\frac{\partial J}{\partial w_k^{(2)}} = \frac{\partial J}{\partial s} \frac{\partial s}{\partial w_k^{(2)}} = e \cdot h_k \quad (2.13)$$

For the second layer bias:

$$\frac{\partial J}{\partial b^{(2)}} = \frac{\partial J}{\partial s} \frac{\partial s}{\partial b^{(2)}} = e \cdot 1 = e \quad (2.14)$$

First Layer Gradients

For the first layer, we need to backpropagate through the ReLU function. Let's denote:

$$\delta_k = \frac{\partial J}{\partial h_k} = \frac{\partial J}{\partial s} \frac{\partial s}{\partial h_k} = e \cdot w_k^{(2)} \quad (2.15)$$

The ReLU derivative is:

$$\text{ReLU}'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} \quad (2.16)$$

For 1D input:

$$\frac{\partial J}{\partial w_k^{(1)}} = \frac{\partial J}{\partial h_k} \frac{\partial h_k}{\partial w_k^{(1)}} \quad (2.17)$$

$$= \delta_k \cdot \text{ReLU}'(w_k^{(1)} x + b_k^{(1)}) \cdot x \quad (2.18)$$

For 2D input:

$$\frac{\partial J}{\partial w_{kj}^{(1)}} = \frac{\partial J}{\partial h_k} \frac{\partial h_k}{\partial w_{kj}^{(1)}} \quad (2.19)$$

$$= \delta_k \cdot \text{ReLU}'(w_{k1}^{(1)} x_1 + w_{k2}^{(1)} x_2 + b_k^{(1)}) \cdot x_j \quad (2.20)$$

For the first layer bias:

$$\frac{\partial J}{\partial b_k^{(1)}} = \delta_k \cdot \text{ReLU}'(w_{k1}^{(1)} x_1 + w_{k2}^{(1)} x_2 + b_k^{(1)}) \quad (2.21)$$

Graphical Interpretation

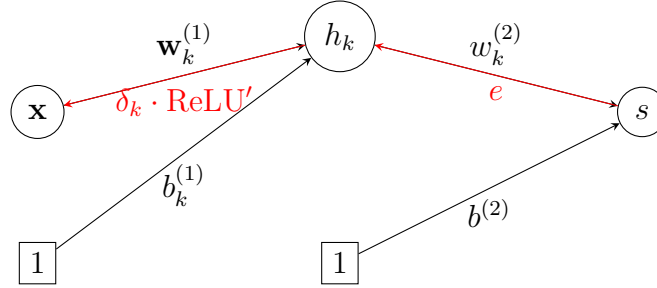


Figure 2.6: Forward and backward signal flow

The backpropagation process has an intuitive graphical interpretation:

1. The error e flows backward through the network.
2. At each ReLU unit:
 - If the unit was inactive (input ≤ 0), no gradient flows back
 - If the unit was active (input > 0), the gradient flows through unmodified
3. The gradients for weights are formed by:
 - Error signal from above (δ_k)
 - Activity state of the ReLU (ReLU')
 - Input to that weight (x or x_j)

This creates a piecewise constant gradient field because of the ReLU function's piecewise linear nature.

2.2.5 Overparameterization and Learning Dynamics

For the one-hidden-layer network, overparameterization occurs when the number of hidden units is large enough that the total number of parameters exceeds the number of training examples:

$$\text{1D case: } (1 + 1)d + (d + 1) > n \quad (2.22)$$

$$\text{2D case: } (2 + 1)d + (d + 1) > n \quad (2.23)$$

where d is the number of hidden units, i.e., dimensionality of h , and n is the number of training examples.

Interpolative Memory

In the overparameterized regime:

- The network can perfectly fit the training data.
- Each hidden unit h_k creates a fold in the space
- The final function $s = f(\mathbf{x})$ interpolates between training points

Implicit Regularization

When trained with gradient descent initialized at zero, the algorithm exhibits implicit regularization:

1. Any parameter configuration that interpolates the data is a global minimum
2. Among all interpolating solutions, gradient descent finds one with:
 - Minimum norm weights
 - Smoother function between data points
 - Better generalization properties

Implications

This behavior has important implications:

- Despite having more parameters than data points, the network can generalize well
- No explicit regularization (like weight decay) is needed
- The choice of initialization and optimization algorithm provides implicit bias
- The learned function tends to be as simple as possible while fitting the data

For the piecewise linear network:

- Each ReLU creates a potential fold
- Gradient descent activates the minimal amounts of folds
- The resulting surface is as flat or as smooth as possible while interpolating the data

This explains why overparameterized neural networks can simultaneously:

- Achieve zero training error (perfect memory)
- Maintain good generalization (smooth interpolation)
- Adapt their complexity to the data (implicit regularization)

2.3 General Multi-layer Perceptron

The general multi-layer perceptron extends the single-hidden-layer architecture to multiple layers of transformations, where each layer processes the output of the previous layer through weights, biases, and non-linear activations. The breakthrough that enables training such deep architectures is backpropagation, an elegant algorithm that efficiently computes parameter gradients by propagating error signals backward through the network. Starting from the output error, backpropagation systematically moves layer by layer in reverse, computing how each parameter contributed to the final error. This process leverages the chain rule of calculus to break down complex gradient calculations into a series of simpler local computations, making the training of deep networks computationally tractable despite their complexity. The algorithm's efficiency and effectiveness have made it the cornerstone of modern deep learning, enabling the training of increasingly deeper architectures.

A general multi-layer perceptron consists of L layers of transformations. Let $h^{(l)} \in \mathbb{R}^{d_l}$ denote the output of layer l , and $s^{(l)} \in \mathbb{R}^{d_l}$ denote the pre-activation values.

For layers $l = 1, \dots, L - 1$:

$$s^{(l)} = \mathbf{W}^{(l)} h^{(l-1)} + b^{(l)} \quad (2.24)$$

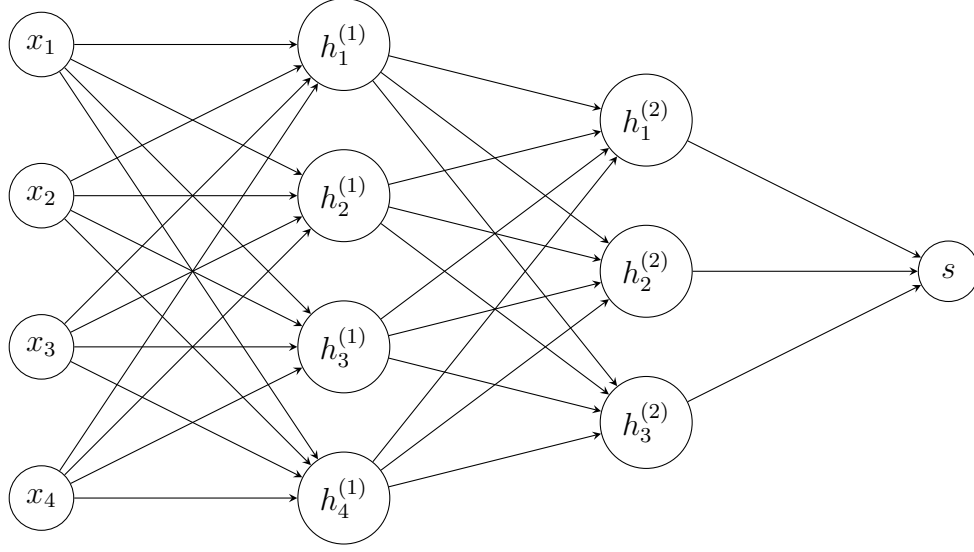
$$h^{(l)} = \sigma(s^{(l)}) \quad (2.25)$$

For the final layer (score):

$$s = h^{(L)} = s^{(L)} = \mathbf{W}^{(L)} h^{(L-1)} + b^{(L)} \quad (2.26)$$

where:

- $\mathbf{W}^{(l)} \in \mathbb{R}^{d_l \times d_{l-1}}$ is the weight matrix
- $b^{(l)} \in \mathbb{R}^{d_l}$ is the bias vector
- $\sigma(\cdot)$ is an element-wise non-linearity (e.g., ReLU)
- $h^{(0)} = x$ is the input

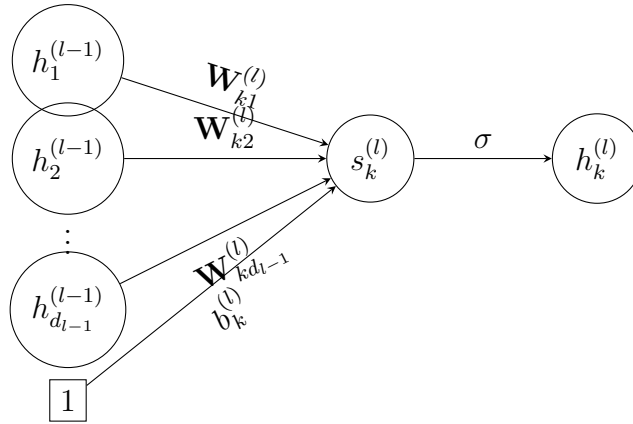
Figure 2.7: MLP architecture with dimensions $d_0 = 4$, $d_1 = 4$, $d_2 = 3$, and scalar output

Each unit in layer l computes:

$$s_k^{(l)} = \sum_{j=1}^{d_{l-1}} \mathbf{w}_{kj}^{(l)} h_j^{(l-1)} + b_k^{(l)} \quad (2.27)$$

$$h_k^{(l)} = \sigma(s_k^{(l)}) \quad (2.28)$$

This computation is a perceptron unit:

Figure 2.8: Single perceptron unit in layer l

Each unit:

- Takes inputs from all units in the previous layer ($h_j^{(l-1)}$)
- Computes a weighted sum plus bias ($s_k^{(l)}$)

- Applies non-linear activation (σ) to produce output ($h_k^{(l)}$)

The multi-layer perceptron is thus composed of many such perceptron units arranged in layers, with the output of each layer serving as input to the next layer.

2.4 Backpropagation for General MLP

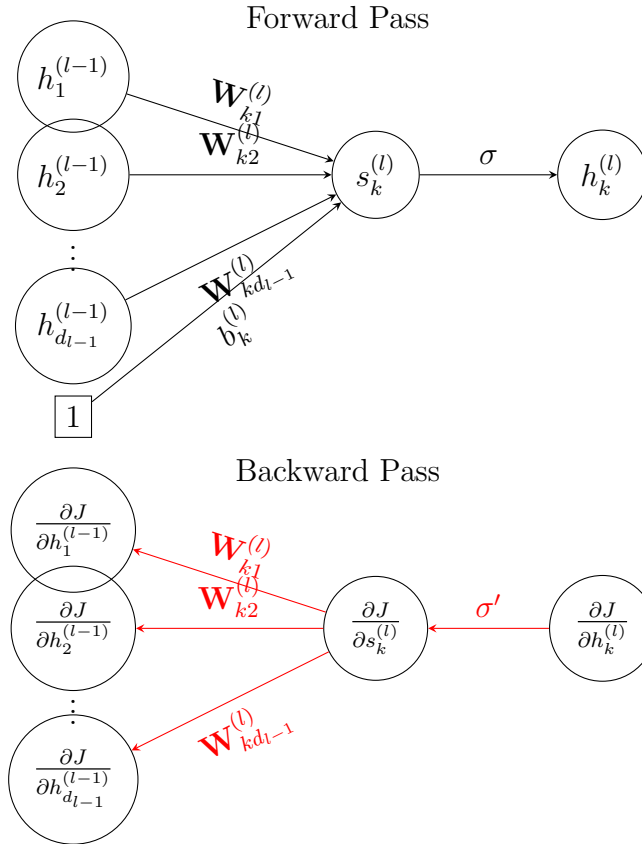


Figure 2.9: Forward and backward propagation through a layer

We start from the objective function:

$$J = \log p(y|s) \quad (2.29)$$

The error at the output layer is:

$$\frac{\partial J}{\partial s} = \frac{\partial J}{\partial h^{(L)}} = \frac{\partial J}{\partial s^{(L)}} = \text{error} \quad (2.30)$$

The backpropagation process through each layer. The complete diagram shows how information flows:

- Forward pass: Computing $h_j^{(l-1)} \rightarrow s_k^{(l)} \rightarrow h_k^{(l)}$

- Backward pass: Computing $\frac{\partial J}{\partial h_k^{(l)}} \rightarrow \frac{\partial J}{\partial s_k^{(l)}} \rightarrow \frac{\partial J}{\partial h_j^{(l-1)}}$
- Weight gradients: Combining values from both passes

2.4.1 Scalar Intuition

First, let's understand backpropagation treating each quantity as a scalar:

Step 1: Through Non-linearity

For any layer l :

$$h^{(l)} = \sigma(s^{(l)}) \quad (2.31)$$

$$\frac{\partial J}{\partial s^{(l)}} = \frac{\partial J}{\partial h^{(l)}} \cdot \frac{\partial h^{(l)}}{\partial s^{(l)}} \quad (2.32)$$

$$= \frac{\partial J}{\partial h^{(l)}} \cdot \sigma'(s^{(l)}) \quad (2.33)$$

Step 2: Through Linear Layer

For the previous layer:

$$s^{(l)} = W^{(l)} h^{(l-1)} \quad (2.34)$$

$$\frac{\partial J}{\partial h^{(l-1)}} = \frac{\partial J}{\partial s^{(l)}} \cdot \frac{\partial s^{(l)}}{\partial h^{(l-1)}} \quad (2.35)$$

$$= \frac{\partial J}{\partial s^{(l)}} \cdot W^{(l)} \quad (2.36)$$

Step 3: Weight and Bias Gradients

For weights and biases:

$$\frac{\partial J}{\partial W^{(l)}} = \frac{\partial J}{\partial s^{(l)}} \cdot \frac{\partial s^{(l)}}{\partial W^{(l)}} \quad (2.37)$$

$$= \frac{\partial J}{\partial s^{(l)}} \cdot h^{(l-1)} \quad (2.38)$$

$$\frac{\partial J}{\partial b^{(l)}} = \frac{\partial J}{\partial s^{(l)}} \quad (2.39)$$

2.4.2 Vector and Matrix Form

Now, we extend to vectors and matrices. We guess that we should transpose some vectors to comply with matrix multiplication rules.

Step 1: Through Non-linearity

$$\frac{\partial J}{\partial s^{(l)}} = \frac{\partial J}{\partial h^{(l)}} \odot \sigma'(s^{(l)}) \quad (2.40)$$

where \odot denotes element-wise multiplication.

Step 2: Through Linear Layer

$$\left(\frac{\partial J}{\partial h^{(l-1)}} \right)^\top = \left(\frac{\partial J}{\partial s^{(l)}} \right)^\top \mathbf{W}^{(l)} \quad (2.41)$$

Note the dimensions for matrix multiplication:

- $\left(\frac{\partial J}{\partial s^{(l)}} \right)^\top$ is $1 \times d_l$
- $\mathbf{W}^{(l)}$ is $d_l \times d_{l-1}$
- $\left(\frac{\partial J}{\partial h^{(l-1)}} \right)^\top$ is $1 \times d_{l-1}$

Step 3: Weight and Bias Gradients

$$\frac{\partial J}{\partial \mathbf{W}^{(l)}} = \frac{\partial J}{\partial s^{(l)}} (h^{(l-1)})^\top \quad (2.42)$$

$$\frac{\partial J}{\partial b^{(l)}} = \frac{\partial J}{\partial s^{(l)}} \quad (2.43)$$

Note that only in Step 2 do we use gradient transpose for convenient matrix multiplication. In other steps, we keep gradients as column vectors.

2.4.3 Detailed Component-wise Verification

Let's verify each step of backpropagation using detailed subscripts.

Step 1: Through Non-linearity

The forward computation is:

$$h_k^{(l)} = \sigma(s_k^{(l)}) \quad (2.44)$$

Therefore:

$$\frac{\partial J}{\partial s_k^{(l)}} = \frac{\partial J}{\partial h_k^{(l)}} \frac{\partial h_k^{(l)}}{\partial s_k^{(l)}} \quad (2.45)$$

$$= \frac{\partial J}{\partial h_k^{(l)}} \sigma'(s_k^{(l)}) \quad (2.46)$$

This matches the vector form with element-wise multiplication:

$$\frac{\partial J}{\partial s^{(l)}} = \frac{\partial J}{\partial h^{(l)}} \odot \sigma'(s^{(l)}) \quad (2.47)$$

Step 2: Through Linear Layer

The forward computation is:

$$s_k^{(l)} = \sum_{j=1}^{d_{l-1}} \mathbf{W}_{kj}^{(l)} h_j^{(l-1)} + b_k^{(l)} \quad (2.48)$$

For component-wise gradient:

$$\frac{\partial J}{\partial h_j^{(l-1)}} = \sum_{k=1}^{d_l} \frac{\partial J}{\partial s_k^{(l)}} \frac{\partial s_k^{(l)}}{\partial h_j^{(l-1)}} \quad (2.49)$$

$$= \sum_{k=1}^{d_l} \frac{\partial J}{\partial s_k^{(l)}} \mathbf{W}_{kj}^{(l)} \quad (2.50)$$

This sum is exactly the j -th component of:

$$\left(\frac{\partial J}{\partial \mathbf{h}^{(l-1)}} \right)^\top = \left(\frac{\partial J}{\partial \mathbf{s}^{(l)}} \right)^\top \mathbf{W}^{(l)} \quad (2.51)$$

Step 3: Weight and Bias Gradients

For weights, using the chain rule:

$$\frac{\partial J}{\partial \mathbf{W}_{kj}^{(l)}} = \frac{\partial J}{\partial s_k^{(l)}} \frac{\partial s_k^{(l)}}{\partial \mathbf{W}_{kj}^{(l)}} \quad (2.52)$$

$$= \frac{\partial J}{\partial s_k^{(l)}} h_j^{(l-1)} \quad (2.53)$$

This matches the matrix form:

$$\frac{\partial J}{\partial \mathbf{W}^{(l)}} = \frac{\partial J}{\partial \mathbf{s}^{(l)}} (\mathbf{h}^{(l-1)})^\top \quad (2.54)$$

For bias:

$$\frac{\partial J}{\partial b_k^{(l)}} = \frac{\partial J}{\partial s_k^{(l)}} \frac{\partial s_k^{(l)}}{\partial b_k^{(l)}} \quad (2.55)$$

$$= \frac{\partial J}{\partial s_k^{(l)}} \cdot 1 \quad (2.56)$$

This matches the vector form:

$$\frac{\partial J}{\partial \mathbf{b}^{(l)}} = \frac{\partial J}{\partial \mathbf{s}^{(l)}} \quad (2.57)$$

The component-wise derivations confirm all three vector/matrix forms:

1. Element-wise multiplication for non-linearity
2. Gradient transpose with weight matrix for chain rule through layer
3. Outer product structure for weight gradients and direct copy for bias gradients

Alternative: Column Vector Gradients

Instead of transposing gradients to row vectors, we can keep them as column vectors and transpose the weight matrix. The forward computation is:

$$s^{(l)} = \mathbf{W}^{(l)} h^{(l-1)} + b^{(l)} \quad (2.58)$$

For component-wise gradient:

$$\frac{\partial J}{\partial h_j^{(l-1)}} = \sum_{k=1}^{d_l} \frac{\partial J}{\partial s_k^{(l)}} \mathbf{W}_{kj}^{(l)} \quad (2.59)$$

$$= \sum_{k=1}^{d_l} \mathbf{W}_{kj}^{(l)} \frac{\partial J}{\partial s_k^{(l)}} \quad (2.60)$$

This sum is exactly the j -th component of:

$$\frac{\partial J}{\partial h^{(l-1)}} = \mathbf{W}^{(l)\top} \frac{\partial J}{\partial s^{(l)}} \quad (2.61)$$

The two forms are equivalent:

- Row vector form: $\left(\frac{\partial J}{\partial h^{(l-1)}}\right)^\top = \left(\frac{\partial J}{\partial s^{(l)}}\right)^\top \mathbf{W}^{(l)}$
- Column vector form: $\frac{\partial J}{\partial h^{(l-1)}} = \mathbf{W}^{(l)\top} \frac{\partial J}{\partial s^{(l)}}$

The column vector form:

- Maintains gradient vectors in same orientation as forward vectors
- Rotates weight matrix instead of gradient vectors
- Makes matrix dimensions more explicit

2.5 Stochastic Gradient Descent

2.5.1 Mini-batch Structure

For a dataset of size N :

- Divide data into mini-batches of size B
- Number of mini-batches per epoch: $T = \lceil N/B \rceil$
- Each mini-batch: $\mathcal{B}_t = \{i_1, \dots, i_B\}$

2.5.2 Gradient Computation

For mini-batch \mathcal{B}_t at time step t :

$$g_t = -\frac{1}{B} \sum_{i \in \mathcal{B}_t} \nabla_{\theta} J_i(\theta) \quad (2.62)$$

where:

- g_t is the average gradient at step t
- $-J_i(\theta)$ is the loss for example i
- θ represents model parameters

2.5.3 Update Rule

The SGD update becomes:

$$\theta_{t+1} = \theta_t - \eta g_t \quad (2.63)$$

where η is the learning rate.

2.5.4 Stochasticity from Mini-batches

The gradient g_t is stochastic because:

- Different mini-batches give different gradients
- g_t is an unbiased estimate of full gradient:

$$\mathbb{E}[g_t] = \frac{1}{N} \sum_{i=1}^N \nabla_{\theta} J_i(\theta) \quad (2.64)$$

- Variance decreases with batch size:

$$\text{Var}[g_t] \propto \frac{1}{B} \quad (2.65)$$

2.5.5 The Concept of Epochs

One epoch means:

- Each training example seen once
- T mini-batch updates
- Full pass through the dataset

2.5.6 Single-Epoch Learning

For large datasets:

- Single epoch = N/B updates
- Can be sufficient if:
 - Dataset is large enough
 - Data has redundancy
 - Task is simple enough
- Benefits:
 - Faster training
 - Fresh examples each update
 - Good for online learning

2.5.7 Practical Considerations

Batch Size Selection

Trade-offs:

- Large B :
 - More stable gradients
 - Better hardware utilization
 - More memory needed
- Small B :
 - More updates per epoch
 - Better regularization
 - More noise in training

Learning Rate Selection

Considerations:

- Scale with batch size: $\eta \propto \sqrt{B}$
- Adjust for gradient variance
- Consider learning rate schedules

Data Shuffling

Between epochs:

- Randomly shuffle dataset
- Creates new mini-batches
- Reduces systematic biases

2.5.8 Advantages of Mini-batch SGD**1. Computational Efficiency:**

- Parallelizable within batch
- Good hardware utilization
- Memory efficient

2. Statistical Efficiency:

- More frequent parameter updates
- Noise helps escape local minima
- Natural regularization

3. Online Learning:

- Can process streaming data
- Adapts to changing patterns
- Memory efficient

2.5.9 Single vs Multiple Epochs

Aspect	Single Epoch	Multiple Epochs
Data size needed	Large	Can be smaller
Learning	One-pass	Iterative refinement
Updates	N/B	$E \cdot N/B$
Memory	Stream data	Full dataset
Convergence	Faster, rougher	Slower, more precise

where E is the number of epochs.

2.6 Adam Optimizer

2.6.1 Recall: Momentum

Momentum accumulates past gradients:

$$v_t = \beta_1 v_{t-1} + g_t \quad (2.66)$$

$$\theta_{t+1} = \theta_t - \eta v_t \quad (2.67)$$

where:

- g_t is current mini-batch gradient
- v_t is velocity (momentum)
- β_1 is momentum coefficient (typically 0.9)

2.6.2 Adaptive Learning Rates

Different parameters may need different learning rates. Adam tracks squared gradients:

$$G_t = \beta_2 G_{t-1} + g_t^2 \quad (2.68)$$

where:

- G_t is accumulated squared gradients
- g_t^2 is element-wise square
- β_2 is decay rate (typically 0.999)

2.6.3 Geometric Intuition

Different dimensions may have:

- Different scales of gradients
- Different curvatures of loss surface
- Different optimal step sizes

2.6.4 Adam Algorithm

Combining momentum and adaptive rates:

Algorithm 3 Adam Optimizer (Simplified)

```

1: Initialize  $\theta_0, v_0 = 0, G_0 = 0$ 
2: for  $t = 1$  to  $T$  do
3:   Compute mini-batch gradient  $g_t$ 
4:    $v_t = \beta_1 v_{t-1} + g_t$  ▷ Update momentum
5:    $G_t = \beta_2 G_{t-1} + g_t^2$  ▷ Update squared grad
6:    $\theta_t = \theta_{t-1} - \eta v_t / \sqrt{G_t + \epsilon}$  ▷ Update
7: end for

```

2.6.5 Geometric Benefits**Momentum Benefits**

- Accumulates consistent gradients
- Dampens oscillations
- Accelerates in flat regions
- Helps escape poor local minima

Adaptive Rate Benefits

- Automatically scales step sizes
- Larger steps for low curvature
- Smaller steps for high curvature
- Handles different parameter scales

For parameter i :

$$\frac{1}{\sqrt{G_{t,i}} + \epsilon} \approx \begin{cases} \text{large} & \text{low curvature/small gradients} \\ \text{small} & \text{high curvature/large gradients} \end{cases} \quad (2.69)$$

2.6.6 Benefits in Practice**1. Robustness:**

- Works well with default hyperparameters
- Handles different scales automatically
- Adapts to dataset characteristics

2. Efficiency:

- Faster convergence

- Fewer hyperparameter tuning needs
- Good for sparse gradients

3. Stability:

- Combines benefits of momentum and adaptive rates
- Handles challenging loss landscapes
- Automatic step size adjustment

2.7 Parameter Initialization

The choice of initial parameters θ_0 significantly impacts training dynamics and final model performance. Proper initialization helps avoid problems like vanishing/exploding gradients and broken symmetry.

2.7.1 Basic Principles

Key initialization objectives:

- Maintain variance across layers
- Break symmetry between units
- Enable efficient gradient flow
- Avoid saturation of non-linearities

2.7.2 Common Initialization Methods

Zero Initialization Setting all weights to zero:

$$\mathbf{W}^{(l)} = \mathbf{0}, \quad b^{(l)} = 0 \quad (2.70)$$

Problems:

- All units compute identical functions
- Network loses expressivity
- Only works for final layer in some cases

Random Normal Initialization Drawing weights from normal distribution:

$$W_{ij}^{(l)} \sim \mathcal{N}(0, \sigma^2) \quad (2.71)$$

Issues:

- Variance grows with layer width
- May lead to vanishing/exploding gradients
- Scale parameter σ needs tuning

Xavier/Glorot Initialization For layers with linear or tanh activation:

$$W_{ij}^{(l)} \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}} + n_{\text{out}}}\right) \quad (2.72)$$

where:

- n_{in} is input dimension
- n_{out} is output dimension
- Factor 2 maintains variance through linear transformation

He Initialization For ReLU networks:

$$W_{ij}^{(l)} \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}}\right) \quad (2.73)$$

Benefits:

- Accounts for ReLU's zero gradient for negative inputs
- Maintains variance through ReLU non-linearity
- Particularly effective for deep networks

2.7.3 Variance Analysis

For a layer with weights \mathbf{W} and input x :

Linear Case

$$\text{Var}[Wx_j] = n_{\text{in}} \text{Var}[W_{ij}] \text{Var}[x_j] \quad (2.74)$$

$$\text{Var}[W_{ij}] = \frac{1}{n_{\text{in}}} \quad (\text{to maintain variance}) \quad (2.75)$$

ReLU Case For ReLU activation $\sigma(x) = \max(0, x)$:

$$\text{Var}[\sigma(x)] = \frac{1}{2} \text{Var}[x] \quad (\text{for zero-mean input}) \quad (2.76)$$

$$\text{Var}[W_{ij}] = \frac{2}{n_{\text{in}}} \quad (\text{to compensate}) \quad (2.77)$$

2.7.4 Practical Guidelines

Best practices for initialization:

1. **Weight Initialization:**

- Use He initialization for ReLU networks
- Use Xavier/Glorot for tanh/sigmoid networks
- Scale carefully based on layer dimensions

2. **Bias Initialization:**

- Generally initialize to zero
- May use small positive values for ReLU
- Consider problem-specific requirements

3. **Special Cases:**

- Initialize skip connections to identity
- Consider orthogonal initialization for RNNs
- Use pretrained weights when available

2.7.5 Impact on Training Dynamics

Good initialization enables:

- Faster convergence
- Better final performance
- More stable training
- Reduced likelihood of bad local minima

Poor initialization can lead to:

- Vanishing/exploding gradients
- Dead ReLU units
- Slow convergence
- Suboptimal solutions

2.7.6 Initialization and Optimization Interplay

The choice of initialization affects:

1. **Learning Rate Selection:**

- Properly scaled initialization allows larger learning rates
- Poor initialization requires more conservative steps

2. **Regularization Effect:**

- Initial weights influence implicit regularization
- SGD with zero initialization has special properties

3. **Optimizer Behavior:**

- Affects momentum dynamics
- Impacts adaptive method scaling
- Influences early training trajectory

This initialization section naturally connects to both the SGD dynamics discussion above and the practical training considerations that follow.

2.8 Multi-class Classification

When y is one-hot over multiple categories, each training example consists of input x and a label indicating one of C possible categories:

$$y = (0, \dots, 0, 1, 0, \dots, 0)^\top \in \{0, 1\}^C \quad (2.78)$$

The network outputs scores $s \in \mathbb{R}^C$, which are then transformed to probabilities:

$$p = \text{softmax}(s), \quad p_c = \frac{e^{s_c}}{\sum_{j=1}^C e^{s_j}} \quad (2.79)$$

2.8.1 Common Examples

Object Recognition

Input x : Image (e.g., $224 \times 224 \times 3$ RGB pixels)

Output y : Object category (one-hot over C classes)

Examples:

- ImageNet: $C = 1000$ object categories
 - dog breeds (Labrador, Golden Retriever, ...)
 - bird species (sparrow, eagle, ...)

- everyday objects (chair, car, book, ...)
- CIFAR-10: $C = 10$ basic categories
 - airplane, automobile, bird, cat, deer
 - dog, frog, horse, ship, truck

Face Recognition

Input x : Face image (e.g., $112 \times 112 \times 3$ RGB pixels)

Output y : Person identity (one-hot over C people)

Examples:

- Large-scale face recognition
 - $C \approx 10^6$ in modern systems
 - Each c represents one person's identity
- Classroom attendance system
 - $C \approx 10^2$ for a school
 - Each c represents one student/teacher

Language Modeling

Input x : Sequence of words (w_1, \dots, w_t)

Output y : Next word w_{t+1} (one-hot over vocabulary)

Examples:

- English language model
 - $C \approx 50,000$ common words
 - Each c represents one word in vocabulary
- Character-level model
 - $C = 26$ (English letters) or
 - $C \approx 128$ (ASCII characters)
- Subword tokenization
 - $C \approx 30,000$ subword units
 - Balances vocabulary size and coverage

2.8.2 Network Architecture

For all these cases, the network:

- Takes domain-specific input x
- Processes through multiple layers $h^{(1)}, \dots, h^{(L-1)}$
- Outputs C -dimensional scores $s = h^{(L)} \in \mathbb{R}^C$
- Applies softmax to get probabilities p

2.8.3 From Logit Scores to Probabilities

The network outputs a vector of logit scores $s \in \mathbb{R}^C$, where each component s_c represents the unnormalized log-probability for category c . These scores are unbounded: $s_c \in (-\infty, \infty)$.

To convert scores to probabilities, we use the softmax function:

$$p_c = \text{softmax}(s)_c = \frac{e^{s_c}}{\sum_{j=1}^C e^{s_j}} \quad (2.80)$$

Why “Softmax”?

The name “softmax” comes from its relationship to the “hardmax” function:

- Hardmax: $\text{hardmax}(s)_c = \begin{cases} 1 & \text{if } s_c > s_j \text{ for all } j \neq c \\ 0 & \text{otherwise} \end{cases}$
- Returns a one-hot vector (winner takes all)
- Not differentiable at boundaries
- Provides no gradient information

Softmax provides a “soft” version of max:

- Higher scores get higher probabilities
- All probabilities are positive and sum to 1
- Differentiable everywhere
- Provides useful gradient information

2.8.4 Loss Function Derivation

For a single training example with one-hot vector y :

$$J = \log p(y|s) \quad (2.81)$$

$$= \log \prod_c p_c^{y_c} \quad (\text{since } y \text{ is one-hot}) \quad (2.82)$$

$$= \sum_{c=1}^C y_c \log p_c \quad (2.83)$$

$$= \sum_{c=1}^C y_c s_c - \log \sum_{j=1}^C e^{s_j} \quad (2.84)$$

Note: $-J$ is commonly used as the loss function, known as cross-entropy loss.

2.8.5 Gradient Derivation

Now let's derive $\frac{\partial J}{\partial s_k}$ for any k :

$$\frac{\partial J}{\partial s_k} = \frac{\partial}{\partial s_k} \left(\sum_{c=1}^C y_c s_c - \log \sum_{j=1}^C e^{s_j} \right) \quad (2.85)$$

$$= \delta_{kc} - \frac{e^{s_k}}{\sum_{j=1}^C e^{s_j}} \quad (2.86)$$

$$= y_k - p_k \quad (2.87)$$

Therefore:

$$\frac{\partial J}{\partial s} = y - p = \text{error} \quad (2.88)$$

2.8.6 Comparison: Binary vs Multi-class

Binary Classification

In binary classification, we typically:

- Choose class 0 as the base class and set its logit score to 0
- Only compute one logit score s for class 1
- Implicitly work with a 2D score vector $(0, s)$

Therefore:

$$p(y = 1|s) = \frac{e^s}{e^0 + e^s} = \sigma(s) \quad (2.89)$$

$$p(y = 0|s) = \frac{e^0}{e^0 + e^s} = 1 - \sigma(s) \quad (2.90)$$

This simplification:

- Reduces parameters (only need one score)
- Makes computation numerically stable
- Is natural when one class is a reference class

Multi-class Classification

For $C > 2$ classes:

- We typically compute all C logit scores
- Setting one class's score to 0 offers little benefit:
 - No clear “reference” class
 - Minimal computational savings
 - Less symmetric treatment of classes
- Full score vector $s \in \mathbb{R}^C$ provides:
 - Symmetric treatment of all classes
 - More natural gradient flow
 - Better numerical stability across all classes

	Binary	Multi-class
Score	$s \in \mathbb{R}$ (implicitly $(0, s)$)	$s \in \mathbb{R}^C$
Activation	$\sigma(s) = \frac{e^s}{1+e^s}$	$\text{softmax}(s)_c = \frac{e^{s_c}}{\sum_j e^{s_j}}$
Label	$y \in \{0, 1\}$	$y \in \{0, 1\}^C$ one-hot
J	$ys - \log(1 + e^s)$	$s_c - \log \sum_j e^{s_j}$
$\frac{\partial J}{\partial s}$	$y - p$	$y - p$

The connection between binary and multi-class cases:

- Binary sigmoid is a special case of softmax with $C = 2$ and $s_0 = 0$
- Both compute normalized exponentials
- Both give similar forms for gradients $(y - p)$
- Both maximize log-likelihood J (or minimize cross-entropy $-J$)

2.8.7 Progressive Abstraction in Hidden Layers

As we move through the layers of a multi-layer perceptron, the representations $h^{(l)}$ become increasingly abstract, progressively discarding nuisance variations that are irrelevant to the classification task.

Hierarchical Abstraction

For a network with L layers, each layer l transforms the representation:

$$h^{(l)} = f(W^{(l)}h^{(l-1)} + b^{(l)}) \quad (2.91)$$

The abstraction process follows:

$$x \rightarrow h^{(1)} \rightarrow h^{(2)} \rightarrow \dots \rightarrow h^{(L-1)} \rightarrow s \quad (2.92)$$

$$\text{concrete} \longrightarrow \text{abstract} \quad (2.93)$$

Nuisance Variation Removal

Consider face recognition where y represents identity. Each layer progressively removes nuisance factors:

- $h^{(1)}$: Local features
 - Retains: edges, textures, local patterns
 - Removes: pixel-level noise, small translations
- $h^{(2)}$: Mid-level features
 - Retains: facial parts, contours
 - Removes: precise edge locations, lighting variations
- $h^{(3)}$: High-level features
 - Retains: facial structure, key identity features
 - Removes: pose variations, expressions
- $h^{(L-1)}$: Identity-specific features
 - Retains: identity-determining characteristics
 - Removes: hairstyle, clothing, background

Manifold Collapse

The representation space progressively "collapses" around class-relevant information:

$$\dim(\text{Var}(h^{(l)}|y)) \searrow \text{ as } l \nearrow \quad (2.94)$$

This collapse has several properties:

- **Within-class Convergence:** As l increases

$$\|h^{(l)}(x_1) - h^{(l)}(x_2)\| \rightarrow 0 \quad \text{for } x_1, x_2 \text{ in same class} \quad (2.95)$$

- **Between-class Separation:** Simultaneously

$$\|h^{(l)}(x_1) - h^{(l)}(x_2)\| \rightarrow d > 0 \quad \text{for } x_1, x_2 \text{ in different classes} \quad (2.96)$$

- **Final Collapse:** At the top layer

$$h^{(L-1)}(x) \approx v_c \quad \text{for all } x \text{ in class } c \quad (2.97)$$

where v_c is a class-specific prototype vector

Information Flow Perspective

The abstraction process can be viewed through information theory:

$$I(h^{(l)}; x) \searrow \text{ as } l \nearrow \quad (\text{decreasing input information}) \quad (2.98)$$

$$I(h^{(l)}; y) \nearrow \text{ as } l \nearrow \quad (\text{increasing class information}) \quad (2.99)$$

where $I(\cdot; \cdot)$ denotes mutual information.

This creates:

- **Information Bottleneck:** Each layer compresses input information while preserving class information
- **Minimal Sufficient Statistics:** Final layers retain only information necessary for classification
- **Maximum Entropy Principle:** Among representations with same class information, network prefers most uniform

Practical Implications

This progressive abstraction has several benefits:

- **Robustness:** Higher layers become invariant to nuisance variations
- **Generalization:** Network learns to ignore task-irrelevant features
- **Efficiency:** Reduced dimensionality in higher layers
- **Interpretability:** Features become more semantic with depth

2.9 Word Embedding

Let's consider predicting the next word given the current word using a simple MLP. We'll use the "Barack Obama" example where we want to predict "Obama" given "Barack".

2.9.1 Model Structure

Input and output are one-hot vectors over vocabulary:

- $x \in \{0, 1\}^{d_{\text{vocabulary}}}$ (e.g., one-hot for "Barack")
- $y \in \{0, 1\}^{d_{\text{vocabulary}}}$ (e.g., one-hot for "Obama")

The model has two layers:

$$h = \mathbf{W}_{\text{embed}} x \quad (\text{embedding}) \quad (2.100)$$

$$s = \mathbf{W}_{\text{unembed}} h \quad (\text{scoring}) \quad (2.101)$$

where:

- $h \in \mathbb{R}^{d_{\text{model}}}$ is the word embedding
- $\mathbf{W}_{\text{embed}} \in \mathbb{R}^{d_{\text{model}} \times d_{\text{vocabulary}}}$
- $\mathbf{W}_{\text{unembed}} \in \mathbb{R}^{d_{\text{vocabulary}} \times d_{\text{model}}}$
- Typically $d_{\text{model}} \ll d_{\text{vocabulary}}$ (e.g., 256 vs 50,000)

2.9.2 Interpretation of Embedding Matrix

The embedding matrix $\mathbf{W}_{\text{embed}}$ contains word embeddings in its columns:

$$\mathbf{W}_{\text{embed}} = \begin{pmatrix} | & | & \cdots & | \\ h_1 & h_2 & \cdots & h_{d_{\text{vocabulary}}} \\ | & | & & | \end{pmatrix} \quad (2.102)$$

When x is a one-hot vector for the k -th word:

- $x = (0, \dots, 0, 1, 0, \dots, 0)^\top$ (1 at position k)
- $h = \mathbf{W}_{\text{embed}} x = k\text{-th column of } \mathbf{W}_{\text{embed}}$
- This selects the embedding vector for the k -th word

2.9.3 Forward Pass Example

For the "Barack Obama" example:

$$x = \text{one-hot("Barack")} \quad (2.103)$$

$$h = \text{embedding of "Barack"} = \mathbf{W}_{\text{embed}} x \quad (2.104)$$

$$s = \text{logit scores for next word} = \mathbf{W}_{\text{unembed}} h \quad (2.105)$$

$$p = \text{softmax}(s) \quad (\text{prediction probabilities}) \quad (2.106)$$

The model should assign high probability to $y = \text{one-hot("Obama")}$.

2.9.4 Gradient Derivation

Starting from $J = \log p(y|s)$, we have:

$$\frac{\partial J}{\partial s} = y - p \quad (2.107)$$

Now let's derive gradients for each layer:

Gradient for Unembedding

$$\frac{\partial J}{\partial \mathbf{W}_{\text{unembed}}} = \frac{\partial J}{\partial s} \cdot \frac{\partial s}{\partial \mathbf{W}_{\text{unembed}}} \quad (2.108)$$

$$= (y - p) h^\top \quad (2.109)$$

This is an outer product: $\mathbb{R}^{d_{\text{vocabulary}}} \times \mathbb{R}^{d_{\text{model}}}$.

Gradient for Hidden Layer

$$\frac{\partial J}{\partial h} = \frac{\partial J}{\partial s} \cdot \frac{\partial s}{\partial h} \quad (2.110)$$

$$= (y - p)^\top \mathbf{W}_{\text{unembed}} \quad (2.111)$$

Note dimensions: $\mathbb{R}^{d_{\text{model}}} = \mathbb{R}^{1 \times d_{\text{vocabulary}}} \times \mathbb{R}^{d_{\text{vocabulary}} \times d_{\text{model}}}$

Gradient for Embedding

$$\frac{\partial J}{\partial \mathbf{W}_{\text{embed}}} = \frac{\partial J}{\partial h} \cdot \frac{\partial h}{\partial \mathbf{W}_{\text{embed}}} \quad (2.112)$$

$$= ((y - p)^\top \mathbf{W}_{\text{unembed}}) x^\top \quad (2.113)$$

This updates only the embedding of the input word:

- Since x is one-hot, $\mathbf{W}_{\text{embed}} x$ selects one column
- Only that column's embedding gets updated
- The gradient is zero for all other word embeddings

2.9.5 Implementation Note

In practice:

- Avoid explicitly constructing one-hot vectors
- Use index operations to select embeddings
- This makes computation efficient despite large vocabulary

2.10 The Profound Idea of Embedding

After establishing neural networks as flexible function approximators through piecewise linear transformations and ReLU rectifications, the concept of embedding emerges as perhaps the most profound idea in deep learning.

2.10.1 From Sparse to Dense Representation

Traditional representations are often sparse:

- One-hot vectors: $(0, \dots, 0, 1, 0, \dots, 0)$
- Binary vectors: $\{0, 1\}^d$
- Few-hot vectors: mostly zeros with few ones

An embedding transforms these into dense vectors:

$$\text{sparse} \in \{0, 1\}^{d_{\text{sparse}}} \xrightarrow{\text{embedding}} \text{dense} \in \mathbb{R}^{d_{\text{dense}}} \quad (2.114)$$

where:

- Each component is a continuous real number
- Typically $d_{\text{dense}} \ll d_{\text{sparse}}$
- All components are potentially meaningful
- Components learn to encode relevant features

2.10.2 Thought Vectors

An embedding can be interpreted as a “thought vector”:

- Each component represents a neuron’s activity
- The full vector represents a distributed pattern
- Similar concepts have similar patterns
- The pattern encodes semantic meaning

For example, a word embedding might encode:

$$\text{“cat”} \rightarrow \begin{pmatrix} 0.82 \text{ (animal)} \\ 0.75 \text{ (pet)} \\ 0.31 \text{ (size)} \\ -0.12 \text{ (aggression)} \\ \vdots \end{pmatrix} \quad (2.115)$$

2.10.3 Vector Operations in Neural Networks

Embeddings are central objects that can undergo various operations:

Linear Transformations

- Matrix multiplication: $\mathbf{W}h$
- Learnable transformations
- Change dimension and perspective

Element-wise Operations

- Non-linear activation: $\sigma(h)$, e.g., ReLU: $\max(0, h)$
- Element-wise product: $h_1 \odot h_2$

Vector Arithmetic

- Addition: $h_1 + h_2$
- Subtraction: $h_1 - h_2$
- Inner product: $h_1^\top h_2$

Famous example in word embeddings:

$$\text{vec}(\text{"king"}) - \text{vec}(\text{"man"}) + \text{vec}(\text{"woman"}) \approx \text{vec}(\text{"queen"}) \quad (2.116)$$

2.10.4 Properties of Embedding Space

The embedding space has meaningful structure:

- **Distance:** Similar concepts are close in the space

$$\|\text{vec}(\text{"cat"}) - \text{vec}(\text{"dog"})\| < \|\text{vec}(\text{"cat"}) - \text{vec}(\text{"car"})\| \quad (2.117)$$

- **Direction:** Differences capture relationships

$$\text{vec}(\text{"Paris"}) - \text{vec}(\text{"France"}) \approx \text{vec}(\text{"Berlin"}) - \text{vec}(\text{"Germany"}) \quad (2.118)$$

2.10.5 Learning Embeddings

Embeddings are learned through:

- Task-specific optimization
- Backpropagation of gradients
- Large-scale training data

The learning process:

- Discovers relevant features automatically
- Creates distributed representations
- Organizes semantic space meaningfully
- Captures complex relationships

2.10.6 Impact on Deep Learning

Embeddings are fundamental because they:

- Transform discrete symbols into continuous vectors
- Enable smooth optimization in neural networks
- Create meaningful semantic spaces
- Allow composition of concepts
- Support transfer learning
- Bridge symbolic and neural computation

This transformation from sparse, discrete representations to dense, continuous vectors is perhaps the key that enables neural networks to process symbolic information and learn meaningful representations from data.

2.11 Associative Memory

Let's revisit the "Barack Obama" example with an associative memory layer. This model captures word associations through an explicit transformation in the embedding space.

2.11.1 Model Structure

The model has three layers:

$$h^{(1)} = \mathbf{W}_{\text{embed}} x \quad (\text{embedding}) \quad (2.119)$$

$$h^{(2)} = \mathbf{W}_{\text{associative}} h^{(1)} \quad (\text{association}) \quad (2.120)$$

$$s = \mathbf{W}_{\text{embed}}^{\top} h^{(2)} \quad (\text{scoring with tied weights}) \quad (2.121)$$

Dimensions:

- $\mathbf{W}_{\text{embed}} \in \mathbb{R}^{d_{\text{model}} \times d_{\text{vocabulary}}}$
- $\mathbf{W}_{\text{associative}} \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$
- $h^{(1)}, h^{(2)} \in \mathbb{R}^{d_{\text{model}}}$
- $x, s \in \mathbb{R}^{d_{\text{vocabulary}}}$

2.11.2 Interpretation

Embedding Layer

When x is one-hot for “Barack”:

- $h^{(1)}$ is the embedding of “Barack”
- This captures semantic features of “Barack”
- $h^{(1)}$ is a column of $\mathbf{W}_{\text{embed}}$

Associative Layer

$\mathbf{W}_{\text{associative}}$ transforms embeddings:

- Maps word meanings to associated word meanings
- “Barack” \rightarrow likely next words like “Obama”
- Learns patterns of word co-occurrence

Tied Weight Layer

Using $\mathbf{W}_{\text{embed}}^{\top}$ for unembedding:

- Same weights for embedding and unembedding
- Reduces parameters
- Forces consistency between input and output spaces

2.11.3 Associative Memory

$\mathbf{W}_{\text{associative}}$ functions as associative memory:

- Maps one thought vector to associated thought vectors
- Learns common patterns in embedding space
- Captures semantic and syntactic relationships

For example:

$$h^{(2)} = \mathbf{W}_{\text{associative}} h^{(1)} \approx \text{embedding of likely next words} \quad (2.122)$$

2.11.4 Gradient Derivation

Starting from $J = \log p(y|s)$ with $\frac{\partial J}{\partial s} = y - p$:

Gradient for Embeddings

Due to weight tying:

$$\frac{\partial J}{\partial \mathbf{W}_{\text{embed}}} = \frac{\partial J}{\partial h^{(1)}} \cdot \frac{\partial h^{(1)}}{\partial \mathbf{W}_{\text{embed}}} + \frac{\partial J}{\partial s} \cdot \frac{\partial s}{\partial \mathbf{W}_{\text{embed}}} \quad (2.123)$$

$$= \left(\frac{\partial J}{\partial h^{(2)}} \mathbf{W}_{\text{associative}} \right) x^\top + (y - p)(h^{(2)})^\top \quad (2.124)$$

Gradient for Association

$$\frac{\partial J}{\partial h^{(2)}} = (y - p)^\top \mathbf{W}_{\text{embed}} \quad (2.125)$$

$$\frac{\partial J}{\partial \mathbf{W}_{\text{associative}}} = \frac{\partial J}{\partial h^{(2)}} \cdot \frac{\partial h^{(2)}}{\partial \mathbf{W}_{\text{associative}}} \quad (2.126)$$

$$= ((y - p)^\top \mathbf{W}_{\text{embed}}) (h^{(1)})^\top \quad (2.127)$$

2.11.5 Learning Dynamics

The model learns two aspects:

1. Word embeddings ($\mathbf{W}_{\text{embed}}$):
 - Semantic features of words
 - Similar words get similar embeddings
2. Associations ($\mathbf{W}_{\text{associative}}$):
 - Patterns of word co-occurrence
 - Transforms meanings to likely next meanings

Weight tying ensures:

- Consistent embedding space
- Same features used for input and output
- More efficient parameter usage

The "Barack Obama" example:

$$x = \text{one-hot("Barack")} \quad (2.128)$$

$$h^{(1)} = \text{embedding of "Barack"} \quad (2.129)$$

$$h^{(2)} = \text{transformation towards "Obama"-like embeddings} \quad (2.130)$$

$$s = \text{scores measuring similarity to all word embeddings} \quad (2.131)$$

2.11.6 Linear Associative Memory

Consider pairs of vectors (a_i, b_i) where $i = 1, \dots, n$. We want to construct a memory that associates a_i with b_i .

Construction

Define the associative matrix as:

$$\mathbf{W}_{\text{associative}} = \sum_{i=1}^n b_i a_i^\top \quad (2.132)$$

If $\{a_i\}$ forms an orthonormal basis:

$$a_i^\top a_j = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \quad (2.133)$$

Perfect Recall

For any basis vector a_k :

$$\mathbf{W}_{\text{associative}} a_k = \sum_{i=1}^n b_i a_i^\top a_k \quad (2.134)$$

$$= \sum_{i=1}^n b_i \delta_{ik} \quad (2.135)$$

$$= b_k \quad (2.136)$$

This shows exact recall of associations for basis vectors.

Example in Word Embeddings

In the “Barack Obama” context:

- a_i : embeddings of input words
- b_i : embeddings of associated next words
- $\mathbf{W}_{\text{associative}}$: learns these associations

2.11.7 Non-linear Associative Memory

We can extend this to a non-linear associative memory using an MLP structure:

$$h^{(1)} = \mathbf{W}_{\text{embed}} x \quad (\text{embedding}) \quad (2.137)$$

$$h^{(2)} = \sigma(\mathbf{W}_{\text{associative}}^{(\text{in})} h^{(1)}) \quad (\text{hidden association}) \quad (2.138)$$

$$h^{(3)} = \mathbf{W}_{\text{associative}}^{(\text{out})} h^{(2)} \quad (\text{output association}) \quad (2.139)$$

$$s = \mathbf{W}_{\text{unembed}} h^{(3)} \quad (\text{scoring}) \quad (2.140)$$

This structure provides:

Non-linear Transformation

- $\sigma(\cdot)$ adds non-linearity to associations
- Each hidden unit creates a “fold” in embedding space with ReLU
- Complex associations can be learned

Memory as Interpolatable Function

The non-linear memory:

- Acts as a piecewise linear function in embedding space
- Can interpolate between learned associations
- Creates smooth transitions between memories
- Generalizes to novel inputs

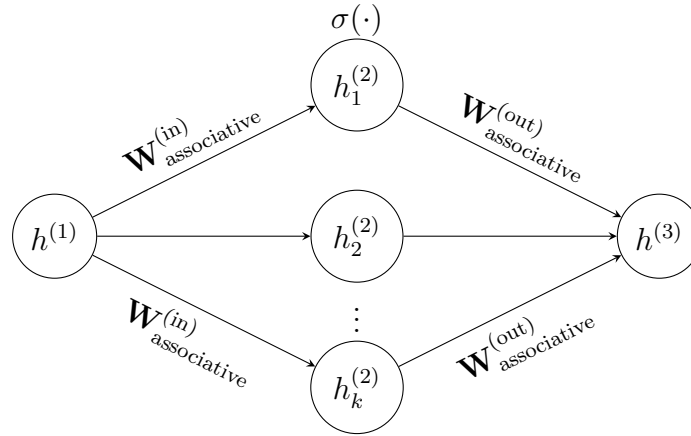


Figure 2.10: Non-linear associative memory structure

Advantages over Linear Memory

1. Capacity:

- Can store more complex associations
- Not limited by input dimension

2. Generalization:

- Learns patterns in associations
- Interpolates between known pairs

3. Flexibility:

- Can capture non-linear relationships
- Adapts to data structure

Learning Dynamics

The network learns:

- $\mathbf{W}_{\text{associative}}^{(\text{in})}$: Extract relevant features
- Hidden layer activations: Create memory regions
- $\mathbf{W}_{\text{associative}}^{(\text{out})}$: Combine features for association

In the “Barack Obama” example:

- $h^{(1)}$: Embedding of “Barack”
- $h^{(2)}$: Non-linear features of association patterns
- $h^{(3)}$: Transformed embedding near “Obama”
- Network learns to associate related concepts

2.12 Embedding for Recommender Systems

2.12.1 Basic Model

For each user i and item j :

- User embedding: $a_i \in \mathbb{R}^d$
- Item embedding: $b_j \in \mathbb{R}^d$
- Rating prediction: $r_{ij} \approx \langle a_i, b_j \rangle$

The prediction model is:

$$r_{ij} = a_i^\top b_j = \sum_{k=1}^d a_{ik} b_{jk} \quad (2.141)$$

2.12.2 Learning from Observations

Given observed ratings $\{r_{ij} : (i, j) \in \Omega\}$, we solve:

$$\min_{a_i, b_j} \sum_{(i,j) \in \Omega} \frac{1}{2} (r_{ij} - \langle a_i, b_j \rangle)^2 \quad (2.142)$$

This learns:

- User preferences as vectors a_i
- Item characteristics as vectors b_j
- Each dimension captures a latent feature

2.12.3 Neural Network Interpretation

From an item-centric view:

- Input: One-hot item vector x_j
- Embedding layer: $b_j = \mathbf{W}_{\text{embed}} x_j$
- User-specific readout: $r_{ij} = a_i^\top b_j$

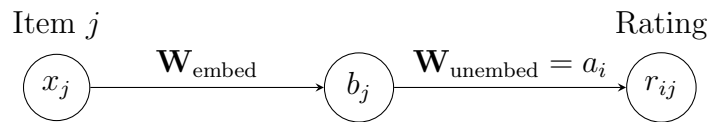


Figure 2.11: Neural network view of recommender system, with user-specific readout weights

2.12.4 Interpretation of User Embeddings

The vector a_i can be interpreted as:

- Neural readout weights in the brain
- Internal representation of preferences
- Control signals for desire/addiction

For example, in video recommendation:

$$a_i = \begin{pmatrix} 0.9 \text{ (action)} \\ 0.2 \text{ (romance)} \\ 0.7 \text{ (sci-fi)} \\ \vdots \end{pmatrix}, \quad b_j = \begin{pmatrix} 0.8 \text{ (action)} \\ 0.1 \text{ (romance)} \\ 0.6 \text{ (sci-fi)} \\ \vdots \end{pmatrix} \quad (2.143)$$

High rating predicted: $\langle a_i, b_j \rangle$ large when preferences align with content.

2.12.5 Mathematical Properties

The inner product structure implies:

- Similar items have similar embeddings b_j
- Similar users have similar embeddings a_i
- Rating is linear in both user and item features

Vector arithmetic works meaningfully:

- $b_{\text{sci-fi}} - b_{\text{drama}}$ captures genre difference
- $a_{\text{teen}} - a_{\text{adult}}$ captures age preference difference
- These differences are consistent across embeddings

2.12.6 Addiction Mechanism

The embedding a_i as neural weights suggests:

- Brain learns to recognize rewarding content
- Embedding adapts through experience
- Strong alignment ($\langle a_i, b_j \rangle \gg 0$) creates addiction

Recommendation systems may:

- Learn and reinforce user preferences
- Create feedback loops strengthening a_i
- Lead to increasingly specific content matching

2.12.7 Extension to Non-linear Models

We can extend to non-linear rating prediction:

$$r_{ij} = f(a_i, b_j) \quad (2.144)$$

where f could be:

- Multi-layer neural network
- Non-linear feature interactions
- Context-dependent transformation

However, the linear model's interpretability makes it valuable:

- Clear meaning of embeddings
- Interpretable feature dimensions
- Simple addition mechanism
- Efficient computation

2.13 Superposition

2.13.1 Beyond Individual Components

Earlier examples of interpreting individual vector components were oversimplified:

$$\text{"cat"} \not\approx \begin{pmatrix} 0.82 \text{ (animal)} \\ 0.75 \text{ (pet)} \\ 0.31 \text{ (size)} \\ \vdots \end{pmatrix} \quad (2.145)$$

In reality, information is distributed across the entire vector through complex superposition patterns.

2.13.2 Basis Representation

An embedding vector h can be expressed in terms of an orthogonal basis:

$$h = \sum_{i=1}^d c_i b_i \quad (2.146)$$

where:

- (b_i) forms an orthogonal basis: $b_i^\top b_j = \delta_{ij}$
- $c_i = h^\top b_i$ are coefficients
- Each c_i may correspond to interpretable features

2.13.3 Subspace Decomposition

More generally, we can decompose h into subspaces:

$$h = \sum_{k=1}^K \mathbf{B}_k \mathbf{C}_k \quad (2.147)$$

where:

- $\mathbf{B}_k \in \mathbb{R}^{d \times d_k}$ defines a subspace
- $\mathbf{C}_k \in \mathbb{R}^{d_k}$ encodes feature values
- $g_k = \mathbf{B}_k \mathbf{C}_k$ represents one aspect of information

2.13.4 Example: Barack Obama Embedding

Consider an embedding vector for "Barack Obama":

$$h_{\text{Barack}} = \sum_{k=1}^K g_k \quad (2.148)$$

where each $g_k = \mathbf{B}_k \mathbf{C}_k$ captures different aspects:

$$g_1 = \text{presidency subspace} \quad (44\text{th President}) \quad (2.149)$$

$$g_2 = \text{ethnicity subspace} \quad (\text{African American}) \quad (2.150)$$

$$g_3 = \text{education subspace} \quad (\text{Harvard Law}) \quad (2.151)$$

$$g_4 = \text{family subspace} \quad (\text{Michelle, Malia, Sasha}) \quad (2.152)$$

$$g_5 = \text{political subspace} \quad (\text{Democratic Party}) \quad (2.153)$$

$$\vdots \quad (2.154)$$

Properties of this decomposition:

- Each g_k lives in its own subspace
- Subspaces may be approximately orthogonal
- Information is distributed but structured
- Features interact through vector addition

2.13.5 Mathematical Properties

Orthogonality

Ideally, subspaces are orthogonal:

$$\mathbf{B}_j^\top \mathbf{B}_k \approx \mathbf{0} \quad \text{for } j \neq k \quad (2.155)$$

This enables:

- Independent representation of features
- Clean separation of information
- Additive composition of meanings

Feature Extraction

To extract feature k :

$$\mathbf{C}_k \approx \mathbf{B}_k^\top h \quad (2.156)$$

This projects h onto subspace k .

2.13.6 Implications

This view of embeddings suggests:

- Information is holistically encoded
- Multiple features coexist through superposition
- Structure emerges from learning
- Interpretation requires subspace analysis

Important considerations:

- Basis vectors/subspaces may not be unique
- Decomposition may be approximate
- Some information may be entangled
- Context may affect interpretation

2.13.7 Neural Network Perspective

In neural networks:

- Hidden layers learn useful subspaces
- Attention mechanisms select relevant subspaces
- Non-linearities enable complex feature interaction
- Training shapes the decomposition structure

This rich representation allows:

- Complex semantic relationships
- Flexible feature composition
- Efficient information storage
- Robust generalization

2.14 Normalization

2.14.1 RMS Normalization

For a vector $h \in \mathbb{R}^d$, RMS normalization is:

$$\text{RMSNorm}(h) = \frac{h}{\sqrt{\frac{1}{d} \sum_{i=1}^d h_i^2}} = \frac{h}{\|h\|/\sqrt{d}} \quad (2.157)$$

Properties:

- Normalizes vector to have RMS = 1
- Preserves direction but standardizes magnitude
- Scale-invariant: $\text{RMSNorm}(\alpha h) = \text{RMSNorm}(h)$ for $\alpha \neq 0$

2.14.2 Geometric Interpretation

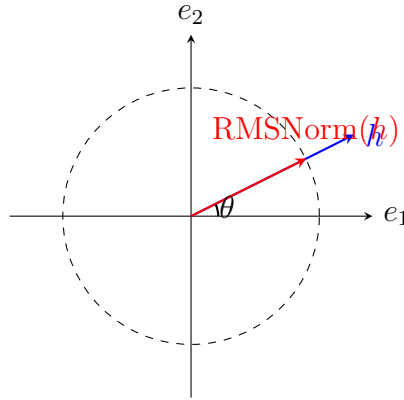


Figure 2.12: RMS normalization projects vectors onto sphere of radius \sqrt{d}

2.14.3 Benefits for Loss Landscape

RMS normalization provides several advantages:

Magnitude Tolerance

- Only direction matters, not magnitude
- Error in $\|h\|$ doesn't affect normalized output
- Loss becomes:

$$L(\theta) = f\left(\frac{h(\theta)}{\|h(\theta)\|}\right) \quad (2.158)$$

Smoother Loss Surface

Consider two vectors h_1 and h_2 :

$$\langle \text{RMSNorm}(h_1), \text{RMSNorm}(h_2) \rangle = \cos(\theta_{12}) \quad (2.159)$$

This means:

- Loss depends on angles between vectors
- Gradients point towards correct direction
- Optimization becomes more geometrically natural

2.14.4 Error Correction Properties

RMS normalization provides fault tolerance:

- **Scale Errors:**

$$\text{RMSNorm}(\alpha h + \epsilon) \approx \text{RMSNorm}(h) \text{ for large } \alpha \quad (2.160)$$

- **Directional Recovery:**

$$\text{RMSNorm}(h + \epsilon_{\perp}) = \frac{h + \epsilon_{\perp}}{\sqrt{\|h\|^2 + \|\epsilon_{\perp}\|^2}/\sqrt{d}} \quad (2.161)$$

where ϵ_{\perp} is noise perpendicular to h

- **Gradient Stability:**

$$\frac{\partial}{\partial h} \text{RMSNorm}(h) \text{ is bounded} \quad (2.162)$$

2.14.5 Cosine Similarity

After RMS normalization, inner products become cosine similarities:

$$\text{sim}(h_1, h_2) = \frac{h_1^{\top} h_2}{\|h_1\| \|h_2\|} = \cos(\theta_{12}) \quad (2.163)$$

This has several implications:

- Measures pure directional alignment
- Bounded in $[-1, 1]$
- Natural for comparing semantic similarity
- Scale-invariant comparison metric

2.14.6 Application in Neural Networks

In practice:

$$\text{out} = \text{RMSNorm}(h)\gamma \quad (2.164)$$

where γ is a learnable scale parameter.

Benefits in neural networks:

- **Training Stability:**

- Consistent scale across layers
- Better gradient flow
- More predictable updates

- **Representation Quality:**

- Focus on directional information
- More robust feature extraction
- Better semantic organization

- **Optimization Behavior:**

- Smoother loss landscape
- More efficient training
- Better convergence properties

2.15 Dropout

2.15.1 Basic Mechanism

For a vector $h \in \mathbb{R}^d$, dropout applies during training:

$$h_{\text{dropout}} = m \odot h \quad (2.165)$$

where:

- $m \in \{0, 1\}^d$ is a random mask
- $P(m_i = 0) = p$ (dropout probability)
- $P(m_i = 1) = 1 - p$ (keep probability)
- \odot denotes element-wise multiplication

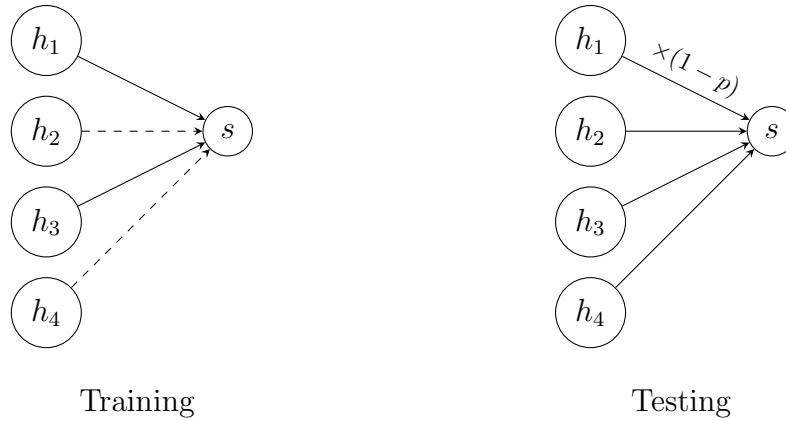


Figure 2.13: Dropout during training and testing phases

2.15.2 Testing Phase Adjustment

During testing, we use the expected value:

$$s = \mathbf{W}_{\text{out}}^{\top} h \cdot (1 - p) \quad (2.166)$$

or equivalently:

$$s = (\mathbf{W}_{\text{out}}^{\top} \cdot (1 - p)) h \quad (2.167)$$

This scaling ensures:

- Expected activation matches training
- No random fluctuations at test time
- Smooth output for deployment

2.15.3 Advantages and Intuitions

Ensemble Effect

Each dropout mask creates a different sub-network:

$$\text{sub-networks} = 2^d \text{ possibilities} \quad (2.168)$$

Benefits:

- Implicit model averaging
- Reduced overfitting
- Better generalization

Feature Co-adaptation Prevention

Without dropout:

$$s = \sum_{i=1}^d w_i h_i \quad (\text{features may co-depend}) \quad (2.169)$$

With dropout:

$$s = \sum_{i=1}^d m_i w_i h_i \quad (\text{features must be robust}) \quad (2.170)$$

This forces:

- Independent feature usefulness
- More robust representations
- Reduced feature co-dependency

Information Distribution

Dropout encourages:

- Distributed representations
- Redundant feature encoding
- Robust information storage

Similar to biological systems:

- Neural redundancy
- Fault tolerance
- Distributed processing

2.15.4 Mathematical Analysis**Training Phase**

For a single forward pass:

$$\mathbb{E}[h_{\text{dropout}}] = (1 - p)h \quad (2.171)$$

$$\text{Var}[h_{\text{dropout}}] = p(1 - p)h \odot h \quad (2.172)$$

Testing Phase

Scaling adjustment ensures:

$$\mathbb{E}_{\text{train}}[s] = \mathbb{E}_m[\mathbf{W}_{\text{out}}^\top (m \odot h)] \quad (2.173)$$

$$= \mathbf{W}_{\text{out}}^\top h \cdot (1 - p) \quad (2.174)$$

$$= s_{\text{test}} \quad (2.175)$$

2.15.5 Implementation Considerations

Common Dropout Rates

Typical values:

- $p = 0.5$ for hidden layers
- $p = 0.2$ for input layer
- Adjust based on layer width

Training vs Testing

Two implementations:

- Scale at test time: $\mathbf{W}_{\text{out}}^\top h \cdot (1 - p)$
- Scale weights: $\mathbf{W}_{\text{out}}^\top \cdot (1 - p)$

2.15.6 Fault Tolerance: RMS Norm vs Dropout Comparison

Both RMS normalization and dropout exhibit distinct but complementary fault tolerance properties, operating at different levels of the network's computation.

Error Types and Correction

For a vector $h \in \mathbb{R}^d$, each method handles different types of errors:

RMS Normalization Handles magnitude errors through projection:

$$\text{RMSNorm}(\alpha h + \epsilon) \approx \text{RMSNorm}(h) \quad \text{for large } \alpha \quad (2.176)$$

For directional noise ϵ_\perp perpendicular to h :

$$\text{RMSNorm}(h + \epsilon_\perp) = \frac{h + \epsilon_\perp}{\sqrt{\|h\|^2 + \|\epsilon_\perp\|^2}/\sqrt{d}} \quad (2.177)$$

Dropout Handles structural errors through masking:

$$h_{\text{dropout}} = m \odot h, \quad m_i \sim \text{Bernoulli}(1 - p) \quad (2.178)$$

Error statistics during training:

$$\mathbb{E}[h_{\text{dropout}}] = (1 - p)h \quad (2.179)$$

$$\text{Var}[h_{\text{dropout}}] = p(1 - p)h \odot h \quad (2.180)$$

Correction Mechanisms

RMS Normalization

- **Continuous Projection:** Maps vectors to constant RMS sphere

$$\|\text{RMSNorm}(h)\| = \sqrt{d} \quad (2.181)$$

- **Gradient Stability:** Bounded derivatives

$$\left\| \frac{\partial}{\partial h} \text{RMSNorm}(h) \right\| \leq C \quad (2.182)$$

- **Scale Invariance:**

$$\text{RMSNorm}(\alpha h) = \text{RMSNorm}(h) \quad \forall \alpha \neq 0 \quad (2.183)$$

Dropout

- **Ensemble Effect:** Implicit averaging over 2^d sub-networks

$$s_{\text{test}} = \mathbb{E}_m[\mathbf{W}_{\text{out}}^\top (m \odot h)] = \mathbf{W}_{\text{out}}^\top h \cdot (1 - p) \quad (2.184)$$

- **Feature Independence:** Each component must be independently useful

$$s = \sum_{i=1}^d m_i w_i h_i \quad (\text{forced robustness}) \quad (2.185)$$

- **Information Distribution:** Redundant encoding across features

$$P(\text{feature}_i \text{ available}) = 1 - p \quad (2.186)$$

Complementary Benefits

The methods provide synergistic fault tolerance:

1. Scale and Structure Protection:

- RMS norm: $h \rightarrow h/\|h\|$ (magnitude correction)
- Dropout: $h \rightarrow m \odot h$ (structural robustness)

2. Error Propagation Control:

- RMS norm prevents magnitude explosion/vanishing
- Dropout prevents co-adaptation and overfitting

3. Combined Guarantees:

$$\|\text{RMSNorm}(m \odot h)\| = \sqrt{d} \quad (\text{stable magnitude}) \quad (2.187)$$

$$\mathbb{E}[m \odot \text{RMSNorm}(h)] = (1 - p) \text{RMSNorm}(h) \quad (\text{preserved direction}) \quad (2.188)$$

This complementary behavior suggests using both methods in practice, as they address different aspects of network robustness and stability.

Chapter 3

Convolutional Neural Networks

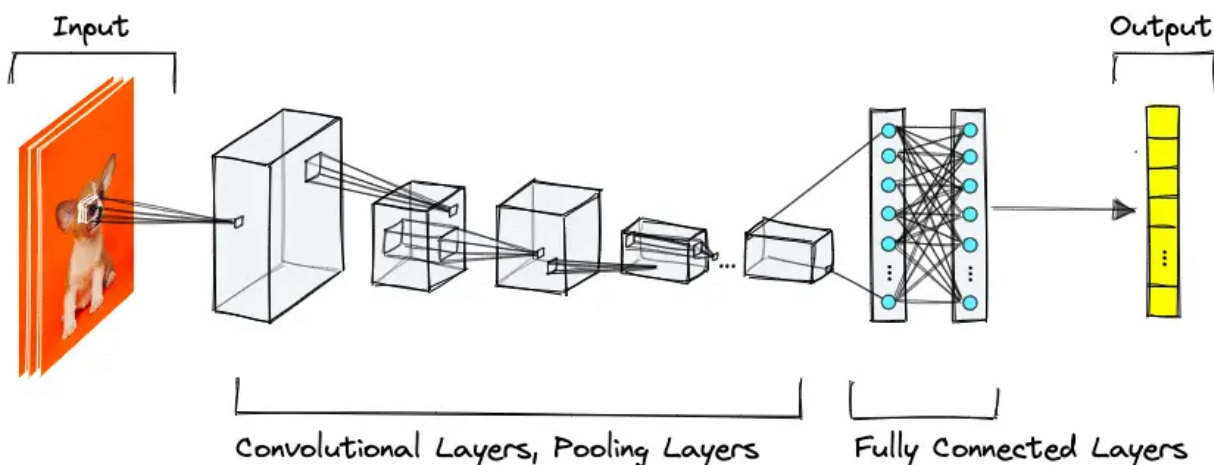


Figure 3.1: Convolutional neural network

Chapter Overview

Convolutional Neural Networks (CNNs) represent a specialized architecture in deep learning, fundamentally conceptualized as computer programs operating in a “neural language” of vectors and transformations. These networks are specifically engineered for processing structured data like images, where information maintains spatial organization. CNNs process images through layers of local feature detectors that share weights across different positions, embedding key assumptions about translation invariance and the importance of local patterns. The architecture progressively transforms input images through convolutional layers that detect local patterns, subsampling operations that reduce spatial dimensions, and fully connected layers that integrate information globally. The implementation details encompass channel organization, 1×1 convolutions, and the specific mechanics of backpropagation through CNN architectures, with additional considerations for efficient parallel processing on modern hardware.

3.1 Neural Networks as Computer Programs

3.1.1 Recall: The Neural Language

The fundamental operation in a multi-layer perceptron is:

$$h^{(l)} = \sigma(\mathbf{W}^{(l)}h^{(l-1)} + b^{(l)}) \quad (3.1)$$

This simple equation defines a basic “neural language” where:

- Objects are vectors (embeddings)
- Operations are learnable transformations
- Information flows through vectors

3.1.2 Basic Operations in Neural Language

The neural language consists of simple vector operations:

1. Linear Transformations:

- Matrix multiplication: $\mathbf{W}h$
- Learnable weights: \mathbf{W} adapts during training
- Bias addition: $+b$

2. Element-wise Operations:

- Rectification: $\sigma(h)$
- Product: $h_1 \odot h_2$
- Addition: $h_1 + h_2$

3. Vector Products:

- Inner product: $h_1^\top h_2$
- Outer product: $h_1 h_2^\top$

3.1.3 Neural Networks as Computer Programs

A neural network can be viewed as:

- A team of vectors passing messages
- Each vector holds a distributed pattern
- Messages transform through learnable operations
- Information flows through vector spaces

The computation flow:

$$h^{(0)} \rightarrow h^{(1)} \rightarrow h^{(2)} \rightarrow \dots \rightarrow h^{(L)} \quad (3.2)$$

where each transition is governed by the neural language operations.

3.1.4 Programming with Vectors

Key aspects of neural programming:

1. **Objects:**

- Vectors as fundamental units
- Each vector represents a pattern
- Projections onto subspaces encode features

2. **Operations:**

- Learnable transformations
- Simple element-wise functions
- Vector arithmetic

3. **Control Flow:**

- Forward propagation of information
- Backward propagation of gradients
- Layer-wise message passing

3.1.5 Learning as Program Writing

The neural program is written by:

- Training data providing examples
- Backpropagation computing updates
- Gradient descent optimizing parameters
- Loss function defining objectives

This process:

- Discovers useful transformations
- Learns meaningful representations
- Adapts to data patterns
- Creates reusable computations

3.1.6 Understanding Neural Programs

To understand a neural network:

- Follow vector messages
- Analyze transformation patterns
- Interpret learned representations
- Study information flow

Key questions:

- What pattern does each vector encode?
- How do transformations modify patterns?
- How does information combine and split?
- What features are extracted and composed?

This vector-centered view provides:

- Clear computational model
- Unified understanding of architectures
- Framework for designing networks
- Basis for analyzing behavior

With this foundation, we can now explore Convolutional Neural Networks as specialized neural programs for processing structured data like images, where vectors will organize in spatial patterns and share transformation parameters.

3.2 Computer Vision

3.2.1 Input Image Structure

At each pixel position (i, j) , the input is a 3D color vector:

$$x_{ij} = \begin{pmatrix} R_{ij} \\ G_{ij} \\ B_{ij} \end{pmatrix} \in \mathbb{R}^3 \quad (3.3)$$

This forms our initial representation:

$$h_{ij}^{(0)} = x_{ij} \quad (3.4)$$

3.2.2 Layers and Representations

As we progress through layers $l = 1, \dots, L$, the network builds increasingly complex representations:

Early Convolutional Layers ($l = 1, 2$)

Local vectors $h_{ij}^{(l)}$ detect primitive patterns:

- **Basic Elements** (spatial range: 3×3 pixels)

$$h_{ij}^{(1)} = \begin{cases} \text{horizontal edges} \\ \text{vertical edges} \\ \text{diagonal bars} \\ \text{corners} \\ \text{color transitions} \end{cases} \quad (3.5)$$

- **Simple Compositions** (spatial range: 5×5 pixels)

$$h_{ij}^{(2)} = \begin{cases} \text{curves} \\ \text{circles} \\ \text{crosses} \\ \text{texture patterns} \\ \text{contour segments} \end{cases} \quad (3.6)$$

Middle Convolutional Layers ($l = 3, 4$)

Local vectors encode object parts:

- **Face Parts** (spatial range: 7×7 to 11×11 pixels)

$$h_{ij}^{(3)} = \begin{cases} \text{eyes} \\ \text{nose bridges} \\ \text{lip curves} \\ \text{eyebrows} \\ \text{ear shapes} \end{cases} \quad (3.7)$$

- **Car Parts** (spatial range: 11×11 to 15×15 pixels)

$$h_{ij}^{(4)} = \begin{cases} \text{wheels} \\ \text{headlights} \\ \text{windows} \\ \text{door handles} \\ \text{side mirrors} \end{cases} \quad (3.8)$$

Late Convolutional Layers ($l = 5, 6$)

Local vectors represent larger compositions:

- **Face Regions** (spatial range: 15×15 to 23×23 pixels)

$$h_{ij}^{(5)} = \begin{cases} \text{eye-nose combinations} \\ \text{mouth-chin regions} \\ \text{cheek-ear areas} \\ \text{forehead-eyebrow segments} \\ \text{complete facial profiles} \end{cases} \quad (3.9)$$

- **Car Sections** (spatial range: 23×23 to 31×31 pixels)

$$h_{ij}^{(6)} = \begin{cases} \text{front grille assemblies} \\ \text{side body panels} \\ \text{rear trunk sections} \\ \text{roof-window combinations} \\ \text{complete car profiles} \end{cases} \quad (3.10)$$

Fully Connected Layers ($l = 7, 8$)

Global vectors integrate complete object information:

- **Object Configuration** ($h^{(7)} \in \mathbb{R}^d$)

$$h^{(7)} = \begin{cases} \text{face arrangement (eyes-nose-mouth)} \\ \text{car structure (front-side-back)} \\ \text{object viewpoint features} \\ \text{global shape descriptors} \\ \text{spatial layout patterns} \end{cases} \quad (3.11)$$

- **Object Identity** ($h^{(8)} \in \mathbb{R}^c$)

$$h^{(8)} = \begin{cases} \text{person identity features} \\ \text{car make/model features} \\ \text{categorical attributes} \\ \text{abstract object properties} \\ \text{classification logits} \end{cases} \quad (3.12)$$

Each layer builds upon previous representations:

$$\text{edges} \xrightarrow{3 \times 3} \text{parts} \xrightarrow{7 \times 7} \text{regions} \xrightarrow{15 \times 15} \text{objects} \xrightarrow{\text{global}} \quad (3.13)$$

Key properties of this hierarchy:

- **Spatial Range:** Receptive field size grows with depth
- **Pattern Complexity:** Features become more sophisticated
- **Semantic Level:** Representations become more abstract
- **Translation Invariance:** Higher layers are more position-robust

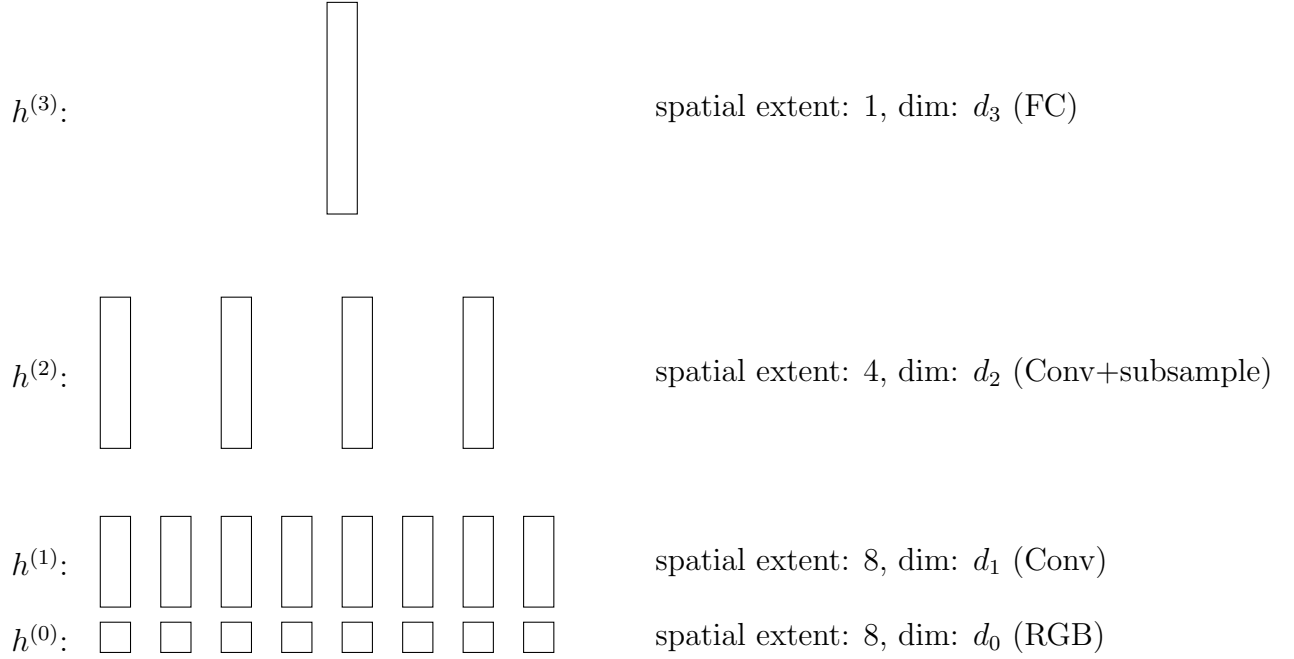


Figure 3.2: CNN structure showing progression from spatially arranged small vectors to single large vector through convolution, subsampling, and fully connected transformations.

3.2.3 Convolutional Layer Computation

At position (i, j) :

$$h_{ij}^{(l)} = \sigma \left(\sum_{\Delta i=-r}^r \sum_{\Delta j=-r}^r \mathbf{W}_{\Delta i, \Delta j} h_{i+\Delta i, j+\Delta j}^{(l-1)} + b^{(l)} \right) \quad (3.14)$$

where:

- r is the range of local shift (typically 1,2,3)
- $\mathbf{W}_{\Delta i, \Delta j}$ is the weight matrix for shift $(\Delta i, \Delta j)$
- Weights are shared across all positions (i, j)

We can also write it as

$$h_{ij}^{(l)} = \sigma \left(\begin{pmatrix} \mathbf{W}_{-r,-r} & \mathbf{W}_{-r,-r+1} & \cdots & \mathbf{W}_{r,r} \end{pmatrix} \begin{pmatrix} h_{i-r,j-r}^{(l-1)} \\ h_{i-r,j-r+1}^{(l-1)} \\ \vdots \\ h_{i+r,j+r}^{(l-1)} \end{pmatrix} + b^{(l)} \right) \quad (3.15)$$

This form shows that:

- Local features are concatenated into a single vector
- One large linear transformation is applied
- The weight matrices are arranged to match the concatenation

The concatenated vector has dimension $d_{l-1}(2r+1)^2$, and the concatenated weight matrix has dimension $d_l \times d_{l-1}(2r+1)^2$.

The convolution operation represents a spatial composition where:

- Each $h_{i+\Delta i, j+\Delta j}^{(l-1)}$ captures local features
- $h_{ij}^{(l)}$ composes these local features
- The composition is learned through $\mathbf{W}_{\Delta i, \Delta j}$

3.2.4 Dimension Considerations

Let the dimensions be:

- $h_{ij}^{(l-1)} \in \mathbb{R}^{d_{l-1}}$ (input features)
- $h_{ij}^{(l)} \in \mathbb{R}^{d_l}$ (output features)

For meaningful composition:

$$d_l > d_{l-1} \quad (3.16)$$

This increase in dimension allows:

- Integration of spatial information
- Detection of more complex patterns
- Composition of lower-level features

Example dimensions:

- Layer 1: $d_1 = 64$ composing RGB ($d_0 = 3$)
- Layer 2: $d_2 = 128$ composing layer 1 features
- Layer 3: $d_3 = 256$ composing layer 2 features

This progressive increase in feature dimensions enables the network to:

- Build hierarchical representations
- Combine local patterns into more complex features
- Maintain rich information about spatial composition

3.2.5 Inductive Bias in Convolution

Understanding Inductive Bias

The term "inductive bias" consists of two key components:

- **Inductive:** Making general conclusions from specific examples
 - Learn patterns from training data
 - Generalize to unseen test data
 - Extrapolate from finite samples to universal rules
- **Bias:** Prior assumptions that guide learning
 - Architectural constraints on possible solutions
 - Built-in preferences for certain patterns
 - Restrictions on model's hypothesis space

Convolution's Inductive Biases

Weight sharing in convolutional layers embeds key assumptions:

$$h_{ij}^{(l)} = f \left(\sum_{\Delta i, \Delta j} \mathbf{W}_{\Delta i, \Delta j} h_{i+\Delta i, j+\Delta j}^{(l-1)} \right) \quad (3.17)$$

These assumptions create three main biases:

1. Translation Invariance

- **What:** Same operation everywhere

$$\mathbf{W}_{ij} = \mathbf{W} \quad \text{for all } (i, j) \quad (3.18)$$

- **Why Inductive:** If a pattern is useful at position (i, j) , it's likely useful at (i', j')
- **Why Biased:** Assumes spatial position doesn't matter for feature detection

2. Local Connectivity

- **What:** Process nearby pixels together

$$\mathbf{W}_{\Delta i, \Delta j} = 0 \quad \text{for } \|\Delta i\|, \|\Delta j\| > k \quad (3.19)$$

- **Why Inductive:** Local patterns in training images generalize to test images
- **Why Biased:** Assumes distant pixels are conditionally independent

3. Hierarchical Composition

- **What:** Stack layers to build complexity

$$\text{receptive field size}(l) > \text{receptive field size}(l-1) \quad (3.20)$$

- **Why Inductive:** Simple patterns combine to form complex ones
- **Why Biased:** Assumes visual world has compositional structure

Benefits of These Biases

These inductive biases provide several advantages:

- **Parameter Efficiency:**

$$\text{params}_{\text{CNN}} \ll \text{params}_{\text{MLP}} \quad \text{for same task} \quad (3.21)$$

- **Sample Efficiency:**

$$\text{training examples needed}_{\text{CNN}} \ll \text{training examples needed}_{\text{MLP}} \quad (3.22)$$

- **Generalization:**

$$\mathbb{E}[\text{test error}_{\text{CNN}}] < \mathbb{E}[\text{test error}_{\text{MLP}}] \quad (3.23)$$

Trade-offs

The strength of these biases creates trade-offs:

- **Advantages:**

- Better generalization on natural images
- Fewer parameters to learn
- More interpretable features

- **Limitations:**

- May not suit non-spatial data
- Restricted in modeling global patterns
- Potentially over-specialized to vision

Therefore, CNN's success stems from matching its inductive biases to the statistical structure of visual data. The "inductive" nature helps generalize from training to test images, while the "biases" encode our prior knowledge about visual world properties.

3.2.6 Subsampling in Convolutional Layers

Subsampling reduces spatial resolution through two steps:

1. Compute convolution as before.
2. Subsample the output:

$$h_{ij}^{(l)} \leftarrow h_{st,sj}^{(l)} \quad (3.24)$$

where:

- s is the stride (typically 2)
- Only keep positions that are multiples of s
- Spatial resolution reduced by factor of s

Progressive Downsampling

With stride $s = 2$:

$$h_{ij}^{(1)} : \text{full resolution} \quad (3.25)$$

$$h_{ij}^{(2)} \leftarrow h_{2i,2j}^{(1)} \quad (3.26)$$

$$h_{ij}^{(3)} \leftarrow h_{2i,2j}^{(2)} \quad (3.27)$$

$$\vdots \quad (3.28)$$

This creates progressive reduction:

- Each layer halves spatial dimensions
- Receptive field doubles relative to input
- Feature dimension typically increases

3.2.7 Fully Connected Layer Computation

Global integration of local features:

$$h^{(l)} = \sigma \left(\sum_{i,j} \mathbf{W}_{ij} h_{ij}^{(l-1)} + b^{(l)} \right) \quad (3.29)$$

where:

- \mathbf{W}_{ij} connects to position (i, j) in previous layer
- $h^{(l)}$ is a global thought vector
- All subsequent layers follow MLP structure

3.2.8 Channel and Kernel View

Let's expand our vector-centered view to show explicit channels. At position (i, j) :

- $h_{ij}^{(l)}$ is a vector of dimension d_l
- Each component k is a channel: $h_{ijk}^{(l)}$
- d_l is the number of channels in layer l

Detailed Convolution Computation

For channel k at position (i, j) in layer l :

$$h_{ijk}^{(l)} = \sigma \left(\sum_{c=1}^{d_{l-1}} \sum_{\Delta i=-r}^r \sum_{\Delta j=-r}^r W_{k,c,\Delta i,\Delta j} h_{i+\Delta i, j+\Delta j, c}^{(l-1)} + b_k^{(l)} \right) \quad (3.30)$$

where:

- $k = 1, \dots, d_l$ is output channel
- $c = 1, \dots, d_{l-1}$ is input channel
- $W_{k,c,\Delta i,\Delta j}$ is kernel weight
- $b_k^{(l)}$ is bias for channel k

3.2.9 1×1 Convolution

A special case occurs when $\Delta i = \Delta j = 0$. This is called a 1×1 convolution:

$$h_{ij}^{(l)} = \sigma(\mathbf{W}_0 h_{ij}^{(l-1)} + b^{(l)}) \quad (3.31)$$

where:

- $\mathbf{W}_0 \in \mathbb{R}^{d_l \times d_{l-1}}$
- No spatial composition is involved
- Each position transformed independently

Interpretation

1×1 convolution performs:

- Feature transformation at each spatial position
- Channel mixing without spatial context
- Pointwise non-linear transformation

This can be viewed as:

- A tiny MLP applied at each position (i, j)
- A learnable projection of the feature vector
- A way to adjust the number of channels

Common Uses

Applications include:

- **Dimension Reduction:**

$$d_l < d_{l-1} \text{ (compression)} \quad (3.32)$$

- **Dimension Expansion:**

$$d_l > d_{l-1} \text{ (enrichment)} \quad (3.33)$$

- **Cross-channel Interaction:**

$$h_{ij}^{(l)} = \text{new combination of channels in } h_{ij}^{(l-1)} \quad (3.34)$$

Computational Efficiency

1×1 convolutions are efficient because:

- Minimal spatial operations
- Simple matrix multiplication at each position
- Fewer parameters than regular convolution
- Can be highly parallelized

3.3 Backpropagation in CNN

Starting from the log-likelihood objective:

$$J = \log p(y|s), \quad s = h^{(L)} = s^{(L)} = \mathbf{W}^{(L)} h^{(L-1)} \quad (3.35)$$

3.3.1 Error Signal

At the top layer:

$$\frac{\partial J}{\partial s} = \frac{\partial J}{\partial h^{(L)}} = \frac{\partial J}{\partial s^{(L)}} = y - p \quad (3.36)$$

3.3.2 Backprop through FC Layers

For a fully connected layer l :

$$h^{(l)} = s^{(l)} = \mathbf{W}^{(l)} h^{(l-1)} \quad (3.37)$$

$$\frac{\partial J}{\partial h^{(l-1)}} = (\mathbf{W}^{(l)})^\top \frac{\partial J}{\partial h^{(l)}} \quad (3.38)$$

$$\frac{\partial J}{\partial \mathbf{W}^{(l)}} = \frac{\partial J}{\partial h^{(l)}} (h^{(l-1)})^\top \quad (3.39)$$

This is standard MLP backprop.

3.3.3 Backprop through Convolutional Layers

For a convolutional layer with spatial positions (i, j) :

$$h_{ij}^{(l)} = \sigma(s_{ij}^{(l)}) \quad (3.40)$$

$$s_{ij}^{(l)} = \sum_{\Delta i, \Delta j} \mathbf{W}_{\Delta i, \Delta j}^{(l)} h_{i+\Delta i, j+\Delta j}^{(l-1)} \quad (3.41)$$

Backprop through non-linearity:

$$\frac{\partial J}{\partial s_{ij}^{(l)}} = \frac{\partial J}{\partial h_{ij}^{(l)}} \odot \sigma'(s_{ij}^{(l)}) \quad (3.42)$$

Backprop to previous layer:

$$\frac{\partial J}{\partial h_{ij}^{(l-1)}} = \sum_{\Delta i, \Delta j} (\mathbf{W}_{\Delta i, \Delta j}^{(l)})^\top \frac{\partial J}{\partial s_{i-\Delta i, j-\Delta j}^{(l)}} \quad (3.43)$$

Weight gradients:

$$\frac{\partial J}{\partial \mathbf{W}_{\Delta i, \Delta j}^{(l)}} = \sum_{i, j} \frac{\partial J}{\partial s_{ij}^{(l)}} (h_{i+\Delta i, j+\Delta j}^{(l-1)})^\top \quad (3.44)$$

3.3.4 Backprop through Subsampling

For a subsampling layer with stride s :

$$h_{ij}^{(l)} \leftarrow h_{si, sj}^{(l)} \quad (3.45)$$

The error signal spreads:

$$\frac{\partial J}{\partial h_{si, sj}^{(l)}} = \frac{\partial J}{\partial h_{ij}^{(l)}} \quad (3.46)$$

3.3.5 Parallelization

FC Layer Parallelism

- Matrix multiplication can be parallelized
- Multiple samples in batch processed in parallel

Conv Layer Parallelism

- All positions (i, j) processed in parallel
- All channels within a position parallel
- Multiple samples in parallel
- Different $(\Delta i, \Delta j)$ parallel

Subsampling Parallelism

- Forward: positions (i, j) independent
- Backward: error signal spreads independently

3.3.6 Implementation Structure

Three levels of parallelism:

Position : (i, j) independent (3.47)

Channel : feature dimensions parallel (3.48)

Batch : samples independent (3.49)

GPU/TPU implementation:

- Forward pass: parallel feature computation
- Backward pass: parallel gradient computation
- Weight updates: parallel accumulation
- Memory access: optimized for parallel ops

Chapter 4

Recurrent Neural Networks

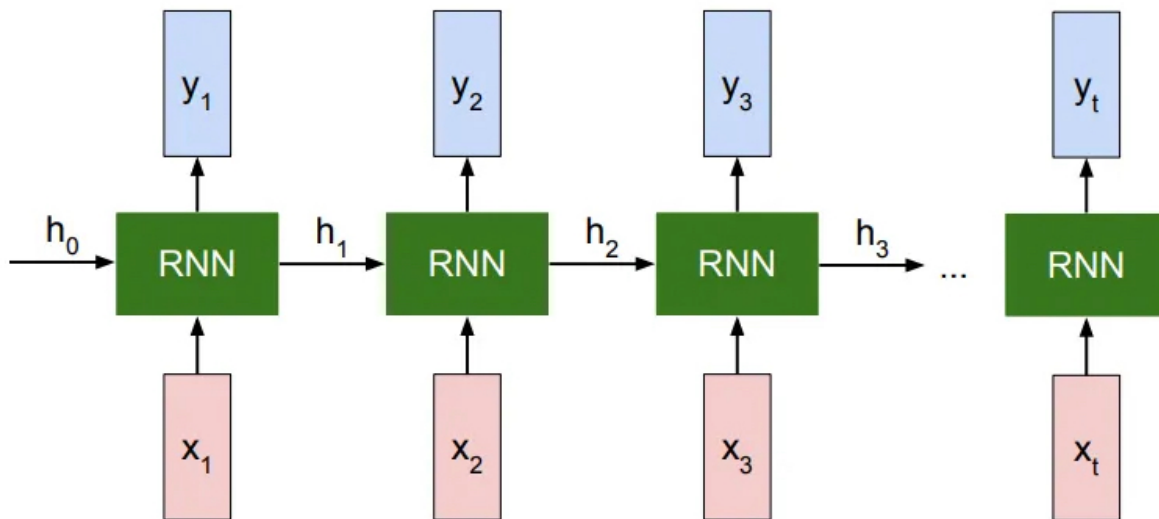


Figure 4.1: Recurrent neural network

Chapter Overview

This chapter explores Recurrent Neural Networks (RNNs) through a vector-centered perspective, focusing on how information evolves over time in neural architectures. It begins by examining the basic structure of RNNs for tasks like next-word prediction, then delves into the critical problem of gradient vanishing in deep sequences. The chapter introduces two fundamental architectural innovations: the memory stream, which allows information to persist over longer time spans horizontally through sequence steps, and the residual stream, which enables information to flow vertically through computational layers. These complementary streams serve as assembly lines - the memory stream accumulates temporal

information while the residual stream builds up computational refinements. The discussion extends to Long Short-Term Memory (LSTM) networks, which organize information across different timescales using cell states and gate mechanisms. The chapter then examines multi-layer RNNs, explaining how information flows both through time and network depth, with residual connections providing crucial paths for gradient flow. It explores the relationship between RNNs and convolutional networks through state space models, which provide a unifying framework for understanding temporal processing. The material advances to cover continuous-time state space models and their discretization, leading to a discussion of the Mamba architecture that introduces selective processing through input-dependent parameters. The chapter concludes with an intriguing parallel between RNNs and quantum mechanics, presenting quantum systems as a special type of RNN where the hidden layer represents fundamental reality and measurements create classical reality as rendered display. Throughout, the chapter emphasizes how the memory streams and residual streams enables deep neural architectures to effectively process both temporal and computational dimensions.

4.1 Vector Evolution over Time

We continue our vector-centered view of neural networks, now dealing with sequences where vectors are indexed by time. The fundamental objects remain vectors (embeddings), but now they evolve over time steps while maintaining the simple neural language of transformations.

At each time step t , we have:

- Input vector: x_t
- Hidden state vector: h_t
- Output vector: y_t

The core recurrent computation is:

$$h_t = \tanh(\mathbf{W}_{\text{recurrent}}h_{t-1} + \mathbf{W}_{\text{embed}}x_t + b) \quad (4.1)$$

The hyperbolic tangent function $\tanh(x)$ is defined as:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1} \quad (4.2)$$

It is used as the activation function for its desirable properties:

- Output range $(-1, 1)$
- Symmetric around zero
- Strong gradients near origin

This vector-centered view emphasizes:

- Vectors as fundamental units of computation
- Same transformations applied at each time step
- Information flow through both time and network layers
- Hidden state vector as evolving memory

4.2 Next Word Prediction

Consider predicting the next word in the phrase “I love machine learning”.

4.2.1 Forward Computation

Starting with zero initial state:

$$h_0 = \mathbf{0} \quad (4.3)$$

$$h_1 = \tanh(\mathbf{W}_{\text{recurrent}}h_0 + \mathbf{W}_{\text{embed}}x_1 + b) \quad \text{input: ‘I’} \quad (4.4)$$

$$h_2 = \tanh(\mathbf{W}_{\text{recurrent}}h_1 + \mathbf{W}_{\text{embed}}x_2 + b) \quad \text{input: ‘love’} \quad (4.5)$$

$$h_3 = \tanh(\mathbf{W}_{\text{recurrent}}h_2 + \mathbf{W}_{\text{embed}}x_3 + b) \quad \text{input: ‘machine’} \quad (4.6)$$

$$s = \mathbf{W}_{\text{unembed}}h_3 \quad \text{logit scores} \quad (4.7)$$

$$p = \text{softmax}(s) \quad \text{next word probabilities} \quad (4.8)$$

The target is $y = x_4 = \text{one-hot vector for “learning”}$.

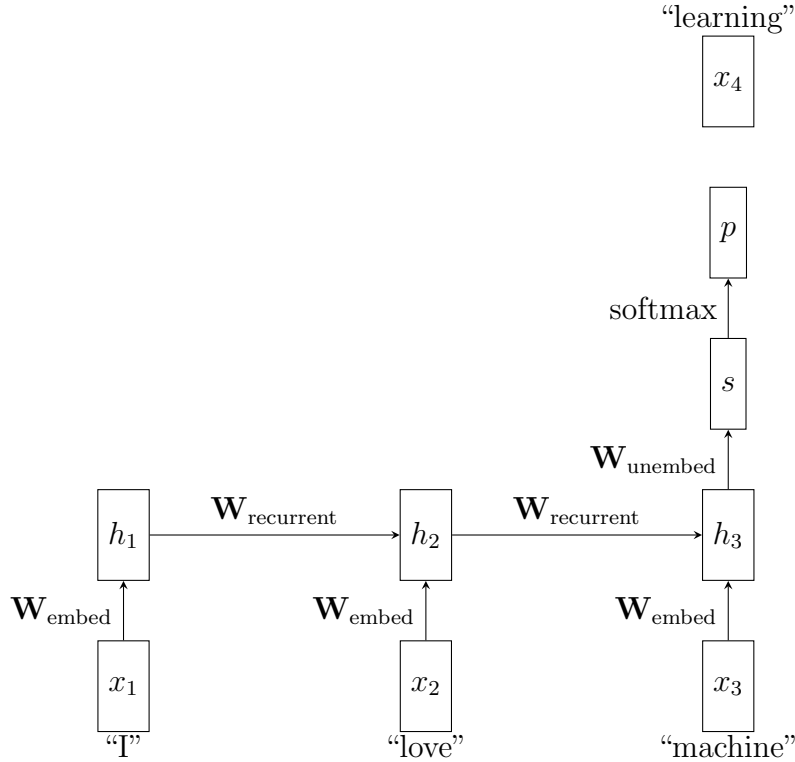


Figure 4.2: RNN predicting next word (x_4) given input sequence (x_1, x_2, x_3)

4.2.2 Meaning of Hidden States

Each hidden state vector h_t encodes:

- Semantic meaning of words seen so far

- Grammatical structure being built
- Context for predicting next word

For example:

- h_1 : understands subject “I”
- h_2 : knows subject-verb “I love”
- h_3 : has complete context “I love machine”

4.2.3 Backpropagation Through Time

Starting from the prediction error:

$$\frac{\partial J}{\partial s} = y - p \quad (4.9)$$

Backprop proceeds backwards in time:

Through Unembedding

$$\frac{\partial J}{\partial h_3} = \mathbf{W}_{\text{unembed}}^\top \frac{\partial J}{\partial s} \quad (4.10)$$

$$\frac{\partial J}{\partial \mathbf{W}_{\text{unembed}}} = \frac{\partial J}{\partial s} h_3^\top \quad (4.11)$$

Through Each Time Step

For $t = 3, 2, 1$:

$$\frac{\partial J}{\partial h_{t-1}} = \mathbf{W}_{\text{recurrent}}^\top \left(\frac{\partial J}{\partial h_t} \odot (1 - h_t^2) \right) \quad (4.12)$$

$$\frac{\partial J}{\partial \mathbf{W}_{\text{recurrent}}} + = \left(\frac{\partial J}{\partial h_t} \odot (1 - h_t^2) \right) h_{t-1}^\top \quad (4.13)$$

$$\frac{\partial J}{\partial \mathbf{W}_{\text{embed}}} + = \left(\frac{\partial J}{\partial h_t} \odot (1 - h_t^2) \right) x_t^\top \quad (4.14)$$

where:

- $(1 - h_t^2)$ is the derivative of \tanh
- $+$ indicates accumulation of gradients
- Gradients flow back through time

4.2.4 Gradient Vanishing

Let's analyze how gradients flow backwards through our example "I love machine learning". The gradient must pass through multiple matrix multiplications and tanh derivatives:

$$\frac{\partial J}{\partial h_1} = (\mathbf{W}_{\text{recurrent}}^\top)^2 \left(\frac{\partial J}{\partial h_3} \odot (1 - h_3^2) \right) \odot (1 - h_2^2) \odot (1 - h_1^2) \quad (4.15)$$

This shows three potential sources of vanishing:

Tanh Derivative

The derivative of tanh has bounded magnitude:

$$\tanh'(x) = 1 - \tanh^2(x) \quad (4.16)$$

$$\leq 1 \text{ for all } x \quad (4.17)$$

$$\approx 0 \text{ when } |x| \text{ is large} \quad (4.18)$$

Therefore:

- Each $(1 - h_t^2)$ factor can significantly reduce gradient
- Especially if hidden states saturate (near -1 or 1)
- Product of many such terms approaches zero

Recurrent Weight Matrix

Multiple multiplications by $\mathbf{W}_{\text{recurrent}}^\top$ can shrink gradients:

- If eigenvalues $|\lambda| < 1$: gradients vanish
- If eigenvalues $|\lambda| > 1$: gradients explode
- Difficult to maintain $|\lambda| \approx 1$ through training

Time Steps

In our example:

- "learning" directly influences h_3
- "machine" influences h_2 through one step
- "love" influences h_1 through two steps
- "I" influences h_1 through three steps

The gradient at each step is approximately:

$$\left\| \frac{\partial J}{\partial h_3} \right\| \approx c \quad (4.19)$$

$$\left\| \frac{\partial J}{\partial h_2} \right\| \approx c\alpha \quad (4.20)$$

$$\left\| \frac{\partial J}{\partial h_1} \right\| \approx c\alpha^2 \quad (4.21)$$

where $\alpha < 1$ is a shrinkage factor from each step, and c is some constant.

Impact on Learning

The vanishing gradient means:

- Recent words (“machine”) are learned effectively
- Distant words (“I”) receive little learning signal
- Long-term dependencies are hard to capture

For example, in predicting “learning”:

- Strong update from “machine”
- Weak update from “love”
- Very weak update from “I”

This makes it difficult for the network to learn patterns like:

- Subject-verb agreement across distance
- Opening and closing parentheses
- Long-term thematic consistency

4.3 LSTM Innovation 1: Memory Stream

In the RNN model, the long term memory is stored in the learned weight matrices, and the short term memory is stored in the hidden vector. Long Short-Term Memory (LSTM) uses memory stream to maintain short-term memory over longer period of time.

We now discuss the first innovation of LSTM, memory stream.

We can reparametrize the RNN update with a skip connection (residual path):

$$h_t = h_{t-1} + \tanh(\mathbf{W}_{\text{recurrent}}h_{t-1} + \mathbf{W}_{\text{embed}}x_t + b) \quad (4.22)$$

4.3.1 Memory Stream

While this formulation is sometimes called a “skip connection”, such terminology understates its importance. It is more illuminating to view it as a “memory stream” or an “assembly line”:

$$h_t = h_{t-1} + \tanh(\mathbf{W}_{\text{recurrent}}h_{t-1} + \mathbf{W}_{\text{embed}}x_t + b) \quad (4.23)$$

In this view, h_{t-1} represents the ongoing memory stream, and we keep adding new information to it at each time step. The $\tanh(\dots)$ term represents the new update to be added to this stream.

Consider our “I love machine learning” example:

$$h_1 = h_0 + \tanh(\mathbf{W}_{\text{recurrent}}h_0 + \mathbf{W}_{\text{embed}}\text{“I”} + b) \quad (4.24)$$

(memory stream now contains “I”)

$$h_2 = h_1 + \tanh(\mathbf{W}_{\text{recurrent}}h_1 + \mathbf{W}_{\text{embed}}\text{“love”} + b) \quad (4.25)$$

(memory stream accumulates “love”)

$$h_3 = h_2 + \tanh(\mathbf{W}_{\text{recurrent}}h_2 + \mathbf{W}_{\text{embed}}\text{“machine”} + b) \quad (4.26)$$

(memory stream adds “machine”)

Like an assembly line, each step takes the current state and adds new information, building up a comprehensive representation. The memory stream maintains all previously accumulated information while incorporating new elements.

This stream view also clarifies the backpropagation dynamics. Gradients can flow freely backward through the memory stream:

$$\frac{\partial h_t}{\partial h_{t-1}} = \mathbf{I} + \mathbf{W}_{\text{recurrent}}^\top \text{diag}(1 - \tanh^2(\dots)) \quad (4.27)$$

The identity matrix \mathbf{I} term is copy-paste operation that allows gradients to flow directly backwards through the memory stream, while the second term represents additional gradient flow through the update computation. This is analogous to being able to trace backwards along the assembly line to see how each addition contributed to the final product.

4.3.2 Example: “I love machine learning”

Using our previous example:

$$h_1 = h_0 + \tanh(\mathbf{W}_{\text{recurrent}}h_0 + \mathbf{W}_{\text{embed}}x_1 + b) \quad (\text{‘I’}) \quad (4.28)$$

$$h_2 = h_1 + \tanh(\mathbf{W}_{\text{recurrent}}h_1 + \mathbf{W}_{\text{embed}}x_2 + b) \quad (\text{‘love’}) \quad (4.29)$$

$$h_3 = h_2 + \tanh(\mathbf{W}_{\text{recurrent}}h_2 + \mathbf{W}_{\text{embed}}x_3 + b) \quad (\text{‘machine’}) \quad (4.30)$$

$$s = \mathbf{W}_{\text{unembed}}h_3 \quad (\text{logit scores}) \quad (4.31)$$

$$p = \text{softmax}(s) \quad (\text{next word probabilities}) \quad (4.32)$$

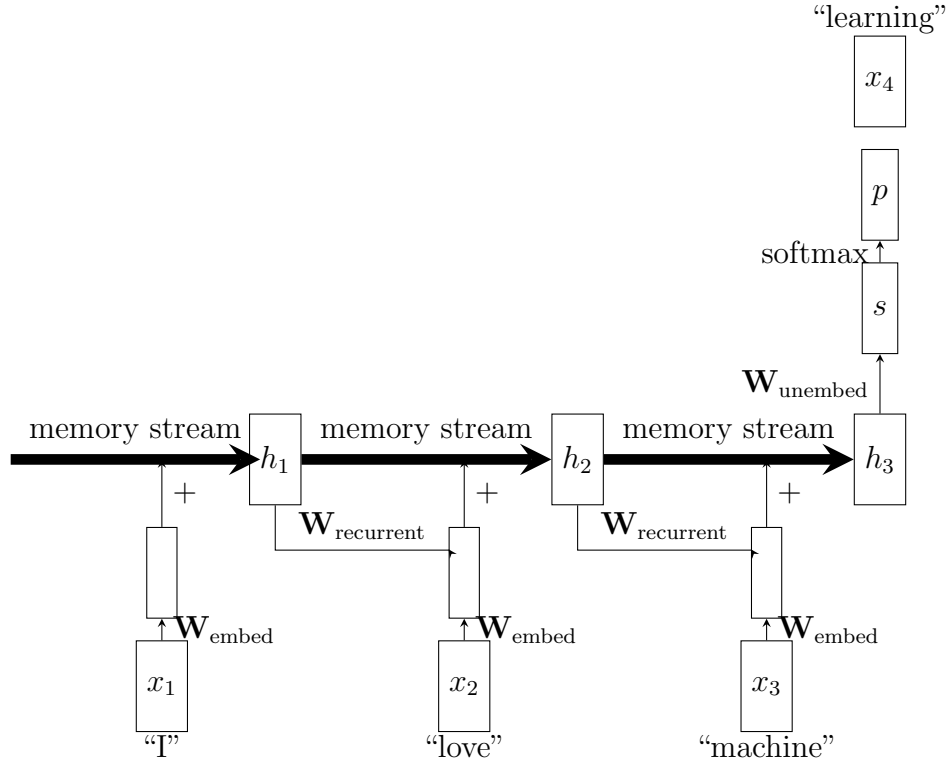


Figure 4.3: RNN with memory stream. Updates are added to the continuous memory stream, with recurrent connections using L-shaped paths.

4.3.3 Superposition in Memory Stream

A key property of the memory stream is that it can accumulate multiple pieces of information without necessarily confounding them, thanks to the high dimensionality of the vector space. When the incremental updates are approximately orthogonal to each other, they can coexist in a state of superposition.

Consider our running example:

$$h_3 = h_0 + \Delta h_1 + \Delta h_2 + \Delta h_3 \quad (4.33)$$

where each update is:

$$\Delta h_1 = \tanh(\mathbf{W}_{\text{recurrent}} h_0 + \mathbf{W}_{\text{embed}} \text{"I"} + b) \quad (4.34)$$

$$\Delta h_2 = \tanh(\mathbf{W}_{\text{recurrent}} h_1 + \mathbf{W}_{\text{embed}} \text{"love"} + b) \quad (4.35)$$

$$\Delta h_3 = \tanh(\mathbf{W}_{\text{recurrent}} h_2 + \mathbf{W}_{\text{embed}} \text{"machine"} + b) \quad (4.36)$$

If these update vectors Δh_t are approximately orthogonal:

$$\Delta h_i^\top \Delta h_j \approx 0 \quad \text{for } i \neq j \quad (4.37)$$

then each component maintains its distinct contribution to the final representation. In high-dimensional spaces, random vectors tend to be nearly orthogonal, making such superposition possible.

This means the final state h_3 can simultaneously encode:

- Subject information from “I”
- Action semantics from “love”
- Object information from “machine”

The network can learn to:

$$h_3 = \alpha_1 v_{\text{subject}} + \alpha_2 v_{\text{action}} + \alpha_3 v_{\text{object}} \quad (4.38)$$

where $v_{\text{subject}}, v_{\text{action}}, v_{\text{object}}$ are nearly orthogonal basis vectors representing different semantic roles, and α_i are learned coefficients.

This superposition property is crucial because:

- It allows the memory stream to maintain multiple aspects of information simultaneously
- Each new update can add information without necessarily destroying previous content
- The network can learn to extract specific components when needed through the unembedding matrix $\mathbf{W}_{\text{unembed}}$

For example, to predict “learning” after “machine”, the model might need to access:

- The subject “I” to ensure semantic coherence
- The verb “love” to maintain the sentiment
- The noun “machine” for immediate context

All these components remain accessible because they exist in superposition in the high-dimensional hidden state vector.

4.3.4 Detailed Gradient Calculation

Given $\frac{\partial J}{\partial h_3}$, let’s calculate $\frac{\partial J}{\partial h_1}$ by considering all possible paths. Let’s use the notation:

$$g_t = \tanh(\mathbf{W}_{\text{recurrent}} h_{t-1} + \mathbf{W}_{\text{embed}} x_t + b) \quad (4.39)$$

so that:

$$h_2 = h_1 + g_2 \quad (4.40)$$

$$h_3 = h_2 + g_3 = (h_1 + g_2) + g_3 \quad (4.41)$$

The gradient $\frac{\partial J}{\partial h_1}$ flows through four paths:

$$\begin{aligned} \frac{\partial J}{\partial h_1} = & \underbrace{\frac{\partial J}{\partial h_3}}_{\text{Path 1}} + \underbrace{\frac{\partial J}{\partial h_3} \cdot \mathbf{W}_{\text{recurrent}}^\top \cdot \text{diag}(1 - g_3^2)}_{\text{Path 2}} + \\ & \underbrace{\frac{\partial J}{\partial h_3} \cdot \mathbf{W}_{\text{recurrent}}^\top \cdot \text{diag}(1 - g_2^2)}_{\text{Path 3}} + \underbrace{\frac{\partial J}{\partial h_3} \cdot (\mathbf{W}_{\text{recurrent}}^\top)^2 \cdot \text{diag}(1 - g_3^2) \cdot \text{diag}(1 - g_2^2)}_{\text{Path 4}} \end{aligned} \quad (4.42)$$

Each path represents:

- Path 1: Direct flow through memory stream
 - $h_1 \rightarrow h_2 \rightarrow h_3$
 - Gradient flows unimpeded through identity connections
- Path 2: Through g_3 computation
 - $h_1 \rightarrow h_2 \rightarrow g_3 \rightarrow h_3$
 - Involves one $\mathbf{W}_{\text{recurrent}}$ transformation
- Path 3: Through g_2 computation
 - $h_1 \rightarrow g_2 \rightarrow h_3$
 - Involves one $\mathbf{W}_{\text{recurrent}}$ transformation
- Path 4: Through both g_2 and g_3 computations
 - $h_1 \rightarrow g_2 \rightarrow h_2 \rightarrow g_3 \rightarrow h_3$
 - Involves two $\mathbf{W}_{\text{recurrent}}$ transformations

The direct path (Path 1) alleviates the gradient vanishing problem because:

- It involves no weight matrices or activation function derivatives
- The gradient flows back unattenuated: $\frac{\partial h_3}{\partial h_1} = \mathbf{I}$
- Even if other paths suffer from vanishing gradients (due to $\mathbf{W}_{\text{recurrent}}$ eigenvalues or tanh derivatives), this path ensures a strong learning signal

For comparison, in a standard RNN without the memory stream, we would only have Path 4:

$$\frac{\partial J}{\partial h_1} = \frac{\partial J}{\partial h_3} \cdot (\mathbf{W}_{\text{recurrent}}^\top)^2 \cdot \text{diag}(1 - g_3^2) \cdot \text{diag}(1 - g_2^2) \quad (4.43)$$

which suffers from:

- Multiple matrix multiplications that can shrink gradients
- Multiple tanh derivatives that are bounded by 1
- No alternative paths for gradient flow

The memory stream thus provides robust gradient flow through Path 1, while the other paths provide additional learning signals when conditions are favorable.

4.3.5 Memory Organization

Long Short-Term Memory (LSTM) organizes information in different time scales:

- Long-term memory: stored in weight matrices \mathbf{W}
 - Learned during training
 - Captures general patterns and knowledge
 - Persists across all sequences
- Short-term memory: stored in cell state vector c_t
 - Evolves during sequence processing
 - Memory stream design helps it persist longer
 - Temporary storage that resets between sequences
- Working memory: stored in hidden state vector h_t
 - Derived from c_t for immediate computation
 - Changes rapidly with each input
 - Interface for reading from memory

4.4 LSTM Innovation 2: Multiplicative Gates

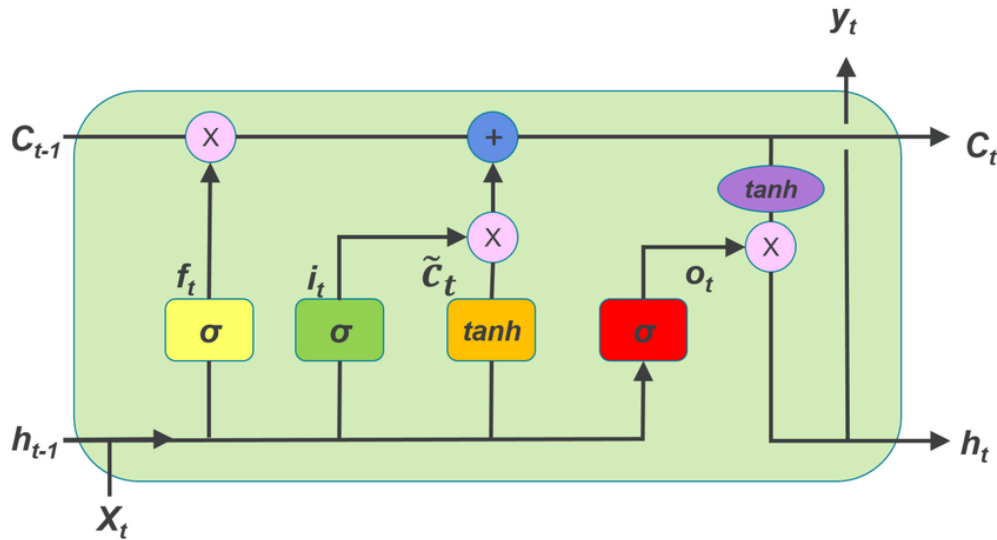


Figure 4.4: Long short-term memory

4.4.1 Key Innovations

LSTM introduces two key innovations over standard RNNs:

Memory Stream for Short-Term Storage

The cell state c_t follows a memory stream architecture:

$$c_t = c_{t-1} + \Delta c_t \quad (4.44)$$

This parametrization allows c_t to maintain information over longer time spans compared to standard RNNs, bridging the gap between immediate and long-term memory. The additive update structure creates paths for both information persistence and gradient flow.

Gate Control Vectors

Three gate vectors modulate information flow:

$$f_t = \sigma(\mathbf{W}_f[h_{t-1}, x_t] + b_f) \quad (\text{forget gate}) \quad (4.45)$$

$$i_t = \sigma(\mathbf{W}_i[h_{t-1}, x_t] + b_i) \quad (\text{input gate}) \quad (4.46)$$

$$o_t = \sigma(\mathbf{W}_o[h_{t-1}, x_t] + b_o) \quad (\text{output gate}) \quad (4.47)$$

These gates use the long-term knowledge stored in \mathbf{W} matrices to control:

- How long information persists in c_t
- What new information enters c_t
- What parts of c_t are exposed through h_t

4.4.2 Memory Update Mechanism

The complete LSTM update demonstrates the interaction between different memory types:

1. Computing potential update using long-term memory:

$$\Delta c_t = \tanh(\mathbf{W}_c[h_{t-1}, x_t] + b_c) \quad (4.48)$$

2. Updating short-term memory through memory stream:

$$c_t = c_{t-1} \odot f_t + \Delta c_t \odot i_t \quad (4.49)$$

3. Generating working memory:

$$h_t = \tanh(c_t) \odot o_t \quad (4.50)$$

The memory stream architecture of c_t allows information to persist by:

- Providing direct paths for information flow ($c_{t-1} \rightarrow c_t$)
- Allowing selective updates through gates
- Maintaining additive accumulation over time

For example, in "I love machine learning":

- \mathbf{W} matrices contain learned patterns about language
- c_t maintains relevant context through the sequence
- h_t provides immediate access to needed information

The combination of long-term memory in weights and enhanced short-term memory in c_t allows LSTM to:

- Apply learned patterns from training (\mathbf{W})
- Maintain context over sequences (c_t)
- Access information as needed (h_t)

It is worth noting that the separation between cell state c_t and hidden state h_t is not strictly necessary. Simpler architectures like Gated Recurrent Units (GRU) successfully combine these roles into a single vector, using a more streamlined gating mechanism while maintaining the key benefits of the memory stream design.

4.5 Multi-layer Recurrent Networks

Multiple recurrent layers can be stacked, where each layer processes the sequence output by the layer below. For a two-layer RNN:

$$h_t^{(1)} = \tanh(\mathbf{W}_{\text{recurrent}}^{(1)} h_{t-1}^{(1)} + \mathbf{W}_{\text{embed}}^{(1)} x_t + b^{(1)}) \quad (\text{Layer 1}) \quad (4.51)$$

$$h_t^{(2)} = \tanh(\mathbf{W}_{\text{recurrent}}^{(2)} h_{t-1}^{(2)} + \mathbf{W}_{\text{associative}}^{(2)} h_t^{(1)} + b^{(2)}) \quad (\text{Layer 2}) \quad (4.52)$$

The final prediction uses the top layer's output:

$$s = \mathbf{W}_{\text{unembed}} h_t^{(2)} \quad (4.53)$$

$$p = \text{softmax}(s) \quad (4.54)$$

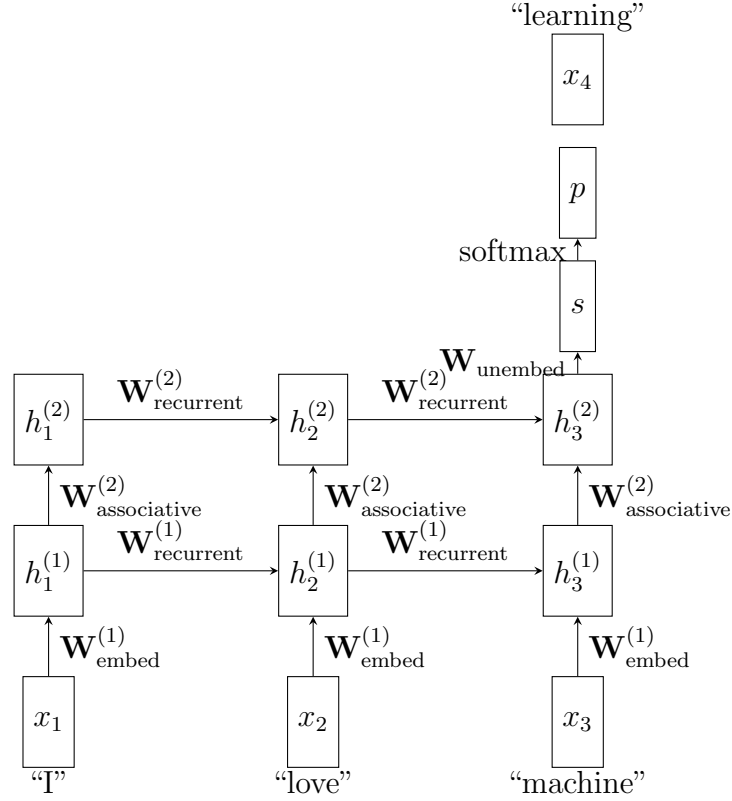


Figure 4.5: Two-layer vanilla RNN with associative connection between layers. Information flows horizontally through time within each layer and vertically through associative memory between layers.

4.5.1 Backpropagation Through Layers and Time

For our two-layer RNN with the final loss J , backpropagation must proceed sequentially through both dimensions:

- Vertically: from Layer 2 down to Layer 1
- Horizontally: from time t back to $t - 1$

Starting with $\frac{\partial J}{\partial s}$ at the output, the computation proceeds as follows:

At Time $t = 3$ (“machine”)

Layer 2 \rightarrow Layer 1:

$$\frac{\partial J}{\partial h^{(2)}_3} = \mathbf{W}^{\top}_{\text{unembed}} \frac{\partial J}{\partial s} \odot (1 - h^{(2)}_3)^2 \quad (4.55)$$

$$\frac{\partial J}{\partial h^{(1)}_3} = \mathbf{W}^{(2)}_{\text{associative}}{}^{\top} \frac{\partial J}{\partial h^{(2)}_3} \odot (1 - h^{(1)}_3)^2 \quad (4.56)$$

At Time $t = 2$ (“love”)

Layer 2:

$$\frac{\partial J}{\partial h_2^{(2)}} = \mathbf{W}_{\text{recurrent}}^{(2)\top} \frac{\partial J}{\partial h_3^{(2)}} \odot (1 - h_2^{(2)^2}) \quad (4.57)$$

Layer 1:

$$\frac{\partial J}{\partial h_2^{(1)}} = \mathbf{W}_{\text{associative}}^{(2)\top} \frac{\partial J}{\partial h_2^{(2)}} \odot (1 - h_2^{(1)^2}) + \mathbf{W}_{\text{recurrent}}^{(1)\top} \frac{\partial J}{\partial h_3^{(1)}} \odot (1 - h_2^{(1)^2}) \quad (4.58)$$

At Time $t = 1$ (“I”)

Layer 2:

$$\frac{\partial J}{\partial h_1^{(2)}} = \mathbf{W}_{\text{recurrent}}^{(2)\top} \frac{\partial J}{\partial h_2^{(2)}} \odot (1 - h_1^{(2)^2}) \quad (4.59)$$

Layer 1:

$$\frac{\partial J}{\partial h_1^{(1)}} = \mathbf{W}_{\text{associative}}^{(2)\top} \frac{\partial J}{\partial h_1^{(2)}} \odot (1 - h_1^{(1)^2}) + \mathbf{W}_{\text{recurrent}}^{(1)\top} \frac{\partial J}{\partial h_2^{(1)}} \odot (1 - h_1^{(1)^2}) \quad (4.60)$$

Key observations:

- We must complete Layer 2 backprop at time t before Layer 1 backprop at time t
- Layer 1 receives gradients from:
 - Layer 2 at the same time step (through $\mathbf{W}_{\text{associative}}^{(2)}$)
 - Its own next time step (through $\mathbf{W}_{\text{recurrent}}^{(1)}$)
- Each step involves both matrix multiplication and tanh derivative
- Gradient vanishing compounds across both dimensions:
 - Through depth (number of layers)
 - Through time (sequence length)

This sequential backpropagation structure means:

- We cannot parallelize across time steps
- We must store all intermediate hidden states
- Learning becomes increasingly difficult for:
 - Deep networks (many layers)
 - Long sequences (many time steps)

4.5.2 Fast Generation/Inference

While training requires backpropagation through both layers and time, generation (also called inference in the literature) can be performed efficiently. For a two-layer network:

At each time step t , we only need to compute:

$$h_t^{(1)} = \tanh(\mathbf{W}_{\text{recurrent}}^{(1)} h_{t-1}^{(1)} + \mathbf{W}_{\text{embed}}^{(1)} x_t + b^{(1)}) \quad (4.61)$$

$$h_t^{(2)} = \tanh(\mathbf{W}_{\text{recurrent}}^{(2)} h_{t-1}^{(2)} + \mathbf{W}_{\text{associative}}^{(2)} h_t^{(1)} + b^{(2)}) \quad (4.62)$$

$$s = \mathbf{W}_{\text{unembed}} h_t^{(2)} \quad (4.63)$$

$$p = \text{softmax}(s) \quad (4.64)$$

Generation is fast because:

- Only forward computation is needed
- Only current time step needs to be stored
- Previous hidden states can be discarded after use
- Computation at each step is constant-time
- Can generate indefinitely with fixed memory

Remark 1. The term “inference” comes from statistical learning, where model parameters are “inferred” during training, and predictions are “inferred” during generation. In deep learning literature, “inference” commonly refers to the generation process, emphasizing that we’re using the trained network to infer what comes next in a sequence.

For example, generating text one word at a time:

- Store only current hidden states $h_t^{(1)}, h_t^{(2)}$
- Sample next word from probability distribution p
- Update hidden states for next step
- Repeat for as many words as needed

This is in contrast to training, which requires:

- Storing all intermediate states
- Computing gradients through all time steps
- Updating all parameters after seeing full sequence

4.5.3 Memory Streams

For a two-layer RNN with memory streams, the updates are:

Layer 1:

$$\Delta h_t^{(1)} = \tanh(\mathbf{W}_{\text{recurrent}}^{(1)} h_{t-1}^{(1)} + \mathbf{W}_{\text{embed}}^{(1)} x_t + b^{(1)}) \quad (\text{compute update}) \quad (4.65)$$

$$h_t^{(1)} = h_{t-1}^{(1)} + \Delta h_t^{(1)} \quad (\text{memory stream 1}) \quad (4.66)$$

Layer 2:

$$\Delta h_t^{(2)} = \tanh(\mathbf{W}_{\text{recurrent}}^{(2)} h_{t-1}^{(2)} + \mathbf{W}_{\text{associative}}^{(2)} h_t^{(1)} + b^{(2)}) \quad (\text{compute update}) \quad (4.67)$$

$$h_t^{(2)} = h_{t-1}^{(2)} + \Delta h_t^{(2)} \quad (\text{memory stream 2}) \quad (4.68)$$

Final prediction:

$$s = \mathbf{W}_{\text{unembed}} h_t^{(2)} \quad (\text{compute logits}) \quad (4.69)$$

$$p = \text{softmax}(s) \quad (\text{get probabilities}) \quad (4.70)$$

Key properties:

- Each layer maintains its own memory stream
- Information flows freely through time in both streams
- Updates are computed using current inputs and previous states
- Memory accumulates additively at each layer

This architecture combines:

- Vertical depth through multiple layers
- Horizontal persistence through memory streams
- Associative mapping between layers

4.6 Residual Stream

4.6.1 Residual Stream Through Layers

For a two-layer RNN with residual stream:

$$h_t^{(1)} = \tanh(\mathbf{W}_{\text{recurrent}}^{(1)} h_{t-1}^{(1)} + \mathbf{W}_{\text{embed}}^{(1)} x_t + b^{(1)}) \quad (\text{Layer 1}) \quad (4.71)$$

$$h_t^{(2)} = h_t^{(1)} + \tanh(\mathbf{W}_{\text{recurrent}}^{(2)} h_{t-1}^{(2)} + \mathbf{W}_{\text{associative}}^{(2)} h_t^{(1)} + b^{(2)}) \quad (\text{Layer 2}) \quad (4.72)$$

The residual stream (vertical identity connection $h_t^{(1)} +$) can be viewed as flowing through computational time:

- Each layer represents one step of computation
- Information flows directly between layers through identity path
- Updates refine the representation at each computational step

This eases backpropagation through layers:

$$\frac{\partial h_t^{(2)}}{\partial h_t^{(1)}} = \mathbf{I} + \mathbf{W}_{\text{associative}}^{(2)\top} \text{diag}(1 - \tanh^2(\cdots)) \quad (4.73)$$

The identity term \mathbf{I} provides:

- Direct gradient path between layers
- Unimpeded flow through computational time
- Protection against gradient vanishing through depth

4.6.2 Residual Stream in MLPs: A Computational Time Perspective

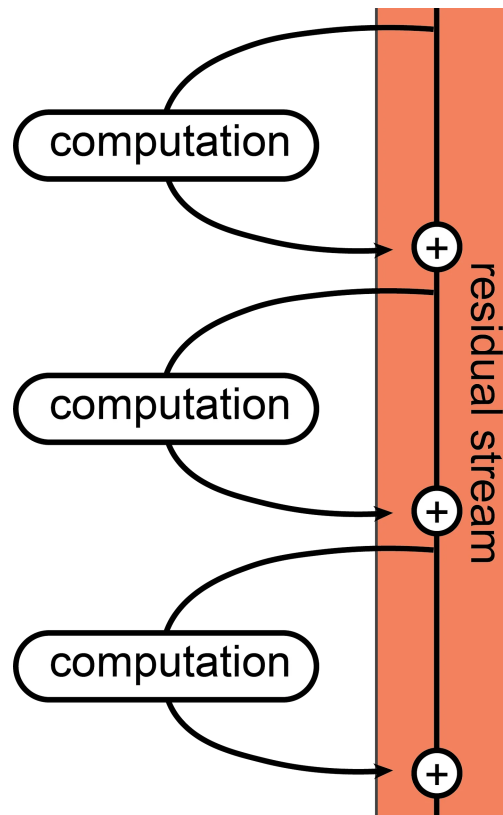


Figure 4.6: Residual Stream

Consider predicting "Obama" given "Barack". With one-hot encoding:

- Input x : one-hot vector for "Barack"
- Output y : one-hot vector for "Obama"

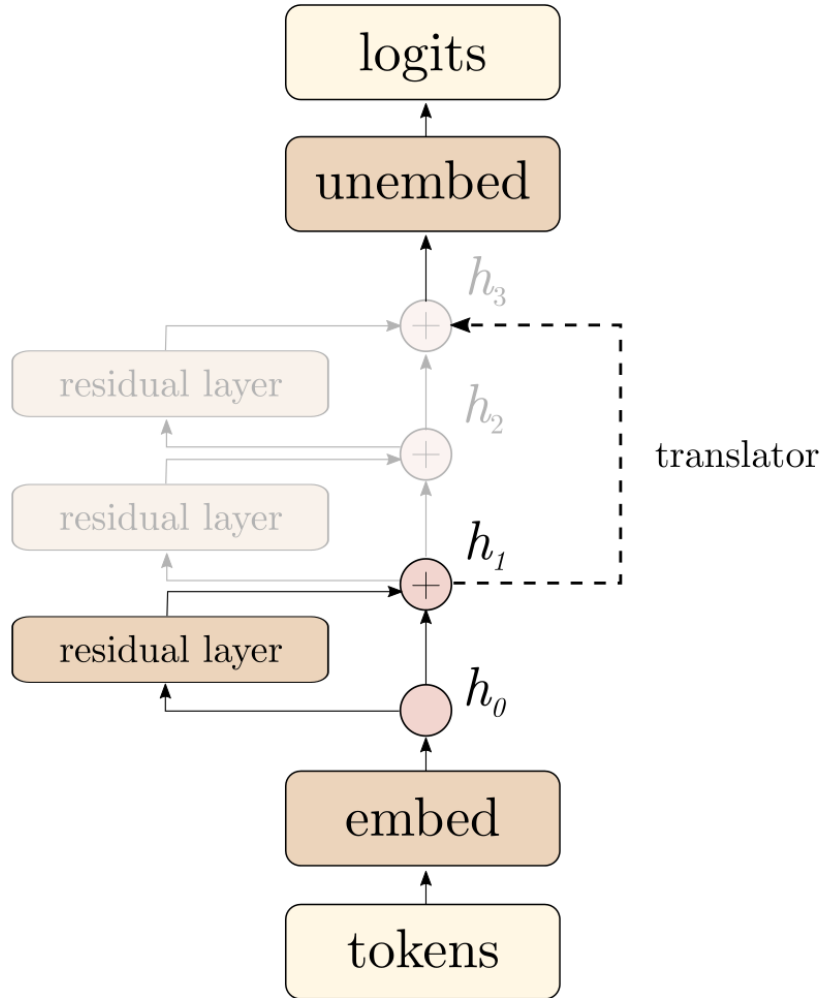


Figure 4.7: Residual Stream

The computation proceeds through layers (or computational time steps):

$$h^{(1)} = \mathbf{W}_{\text{embed}} x \quad (\text{initial embedding}) \quad (4.74)$$

$$h^{(l)} = h^{(l-1)} + \sigma(\mathbf{W}_{\text{residual}}^{(l)} h^{(l-1)} + b^{(l)}) \quad \text{for } l = 2, \dots, L \quad (4.75)$$

$$s = \mathbf{W}_{\text{unembed}} h^{(L)} \quad (\text{logit scores}) \quad (4.76)$$

$$p = \text{softmax}(s) \quad (\text{probabilities}) \quad (4.77)$$

4.6.3 Computational Time Interpretation

We can rewrite this using computational time index t instead of layer index l :

$$h_0 = \mathbf{W}_{\text{embed}}x \quad (\text{initial state}) \quad (4.78)$$

$$h_t = h_{t-1} + \sigma(\mathbf{W}_{\text{residual}}^{(t)}h_{t-1} + b^{(t)}) \quad \text{for } t = 1, \dots, T \quad (4.79)$$

$$s = \mathbf{W}_{\text{unembed}}h_T \quad (\text{final prediction}) \quad (4.80)$$

This highlights that:

- t represents steps of computation, not real time
- Each step refines the representation of "Barack"
- The residual stream $h_{t-1} +$ maintains a direct path to earlier computations

For example, the computation might:

- $t = 0$: Initial embedding captures "Barack" as a name
- $t = 1$: Add features about being a first name
- $t = 2$: Add features about being presidential
- $t = 3$: Refine towards the specific person
- $t = 4$: Further refine for predicting the surname

Each update $\sigma(\mathbf{W}_{\text{residual}}^{(t)}h_{t-1} + b^{(t)})$ adds new features while the residual stream $h_{t-1} +$ preserves existing ones.

4.6.4 Contrast with Real Time

This computational time t differs from real time in several ways:

- Not tied to sequence order (unlike RNNs)
- All steps process the same input ("Barack")
- Steps represent computational refinement
- Could potentially be parallelized
- Number of steps T is a design choice

The residual stream ensures:

- Information from initial embedding persists
- Gradients flow easily through computational steps
- Network can learn how many refinement steps are needed
- Each step can focus on adding new features

4.6.5 Gradient Flow

The gradient flows easily through computational time:

$$\frac{\partial h_t}{\partial h_{t-1}} = \mathbf{I} + \mathbf{W}_{\text{residual}}^{(t)\top} \text{diag}(\sigma'(\cdots)) \quad (4.81)$$

The identity term from the residual stream ensures that:

- Initial embedding can directly influence final prediction
- Gradients can flow back through all computational steps
- Network can learn which refinements are useful

4.6.6 Residual Stream as Assembly Line

The residual stream computation:

$$h_0 = \mathbf{W}_{\text{embed}} x \quad (\text{initial representation}) \quad (4.82)$$

$$h_t = h_{t-1} + \sigma(\mathbf{W}_{\text{residual}}^{(t)} h_{t-1} + b^{(t)}) \quad \text{for } t = 1, \dots, T \quad (4.83)$$

can be viewed as an assembly line where:

- h_{t-1} is the current state of the product
- $\sigma(\mathbf{W}_{\text{residual}}^{(t)} h_{t-1} + b^{(t)})$ is the incremental improvement
- Each step only needs to learn a small refinement

For the "Barack" example:

$$h_0 = \text{basic name embedding} \quad (4.84)$$

$$h_1 = h_0 + \text{first name features} \quad (4.85)$$

$$h_2 = h_1 + \text{presidential features} \quad (4.86)$$

$$h_3 = h_2 + \text{specific person features} \quad (4.87)$$

$$h_4 = h_3 + \text{surname prediction features} \quad (4.88)$$

4.6.7 Learning Simplification

This assembly line structure simplifies learning because:

- Each step has a focused task
 - Only learn the incremental improvement
 - Don't need to reproduce existing features
 - Can specialize in specific refinements

- Default behavior is identity
 - If no improvement needed, set residual to zero
 - Unnecessary computation can be skipped
 - Network can learn how many steps are useful
- Progressive refinement
 - Build features gradually
 - Each step can be small and simple
 - Complex transformations emerge from composition

4.6.8 Parallel with Memory Stream

The assembly line interpretation parallels the memory stream:

Memory Stream (Real Time):

- Flows horizontally through sequence steps
- Each step adds new temporal information
- Default is to maintain current memory
- Gradients flow easily through time

Residual Stream (Computational Time):

- Flows vertically through computational steps
- Each step adds refinements to representation
- Default is to maintain current features
- Gradients flow easily through layers

4.6.9 Gradient Flow as Quality Control

The ease of backpropagation:

$$\frac{\partial h_t}{\partial h_{t-1}} = \mathbf{I} + \mathbf{W}_{\text{residual}}^{(t)\top} \text{diag}(\sigma'(\cdots)) \quad (4.89)$$

can be understood as efficient quality control:

- Error signals flow directly back through assembly line
- Each step gets clear feedback about its contribution
- Identity path ensures no loss of gradient signal

- Easy to identify which refinements were helpful

This parallelism between assembly lines in:

- Real time (memory stream)
- Computational time (residual stream)

provides a unified view of how neural networks can process information efficiently through both temporal and computational dimensions.

4.7 Residual Stream as Learned Iterative Algorithm

4.7.1 Finite Step Iterative Algorithm

The residual stream computation:

$$h_t = h_{t-1} + \sigma(\mathbf{W}_{\text{residual}}^{(t)} h_{t-1} + b^{(t)}) \quad (4.90)$$

is structurally similar to iterative optimization algorithms like gradient ascent:

$$x_t = x_{t-1} + \eta \nabla f(x_{t-1}) \quad (4.91)$$

Key parallels:

- Both update current state by adding an increment
- Both refine solution through multiple steps
- Both maintain the form: state + update

4.7.2 Algorithm Without Explicit Objectives

However, the residual stream differs crucially:

- Traditional iterative algorithms:
 - Know the objective function $f(x)$
 - Compute explicit gradients ∇f
 - Follow a known optimization principle
- Residual stream:
 - No explicit objective function
 - Updates learned from data
 - Discovers its own optimization principle

For the "Barack" example:

- Traditional approach would need:
 - Explicit criterion for good representation
 - Way to measure improvement direction
 - Known update rule
- Residual stream learns:
 - What features to extract
 - How to refine them
 - When to stop refining

4.7.3 Learned Update Rule

The network learns an update function:

$$\Delta h_t = \sigma(\mathbf{W}_{\text{residual}}^{(t)} h_{t-1} + b^{(t)}) \quad (4.92)$$

which can be viewed as:

- A learned gradient-like quantity
- Implicitly optimizing some unknown objective
- Automatically adapting step sizes
- Discovering useful feature refinements

4.7.4 Finite-Step Design

The finite number of steps T means:

- Network must learn efficient updates
- Each step should be maximally useful
- No luxury of asymptotic convergence
- Must reach good solution in fixed steps

This constraint often leads to:

- More aggressive early refinements
- Progressive specialization of steps
- Efficient use of computational budget

4.7.5 Advantages of Learning the Algorithm

This learned iterative process has benefits:

- No need to specify:
 - Objective function
 - Update rules
 - Convergence criteria
- Can discover:
 - Efficient transformation sequences
 - Task-specific refinement patterns
 - Optimal computation paths
- Adapts to:
 - Data distribution
 - Task requirements
 - Computational constraints

The network essentially learns:

- What to optimize
- How to optimize it
- When optimization is sufficient

all implicitly through training on the end task.

4.7.6 Neural Programs with For Loops

The residual stream computation can be viewed as a for loop in a neural program:

$$h_0 = \mathbf{W}_{\text{embed}} x \quad // \text{ initialize} \quad (4.93)$$

$$\text{for } t = 1 \text{ to } T : \quad // \text{ computational steps} \quad (4.94)$$

$$\Delta h_t = \sigma(\mathbf{W}_{\text{residual}}^{(t)} h_{t-1} + b^{(t)}) \quad // \text{ compute update} \quad (4.95)$$

$$h_t = h_{t-1} + \Delta h_t \quad // \text{ update state} \quad (4.96)$$

$$s = \mathbf{W}_{\text{unembed}} h_T \quad // \text{ final output} \quad (4.97)$$

4.8 Neural Programming Language

4.8.1 Neural network as a computer program

This program is written in a simple neural language where:

- Basic objects are vectors
- Basic operations are:
 - Matrix multiplication ($\mathbf{W}h$)
 - Element-wise nonlinearity ($\sigma(\cdot)$)
 - Addition (+)
- Control flow is implicit in network structure
- Parameters are learned from data

4.8.2 Data as the Programmer

The remarkable aspect is that this program is written by data via backpropagation:

- Traditional programming:
 - Human writes explicit instructions
 - Logic must be manually specified
 - Updates must be precisely defined
- Neural programming:
 - Data shapes the transformation matrices
 - Logic emerges from learned patterns
 - Updates are discovered automatically

4.8.3 Role of Residual Stream

The residual stream is crucial for learning deep for loops because:

- Provides stable gradient flow through many steps
- Makes each iteration focus on incremental improvements
- Allows program to build complexity gradually
- Enables learning of very deep computational sequences

4.8.4 Foundation of Digital Intelligence

This ability to learn programs through data underlies the success of digital intelligence:

- Programs emerge from:
 - Simple vector operations
 - Learned transformations
 - Multiple processing steps
- Complex behaviors arise from:
 - Composition of simple operations
 - Data-driven parameter adjustment
 - Iterative refinement
- Intelligence emerges through:
 - Learning computational patterns
 - Discovering useful transformations
 - Building hierarchical processing

The key innovations that enable this are:

- Simple neural language
 - Universal vector representations
 - Learnable matrix transformations
 - Composable operations
- Residual streams
 - Enable deep iteration
 - Stabilize learning
 - Allow complexity to emerge
- Backpropagation
 - Writes programs through data
 - Discovers useful computations
 - Optimizes multi-step processes

This framework allows us to:

- Learn complex programs without explicit programming
- Discover computational patterns from examples
- Build intelligence through iterated refinement
- Scale to deeper and more sophisticated processes

4.9 Parameter Sharing Across Streams

4.9.1 Memory Stream and Residual Stream

Memory Stream (Real Time):

$$h_t = h_{t-1} + \tanh(\mathbf{W}_{\text{recurrent}}h_{t-1} + \mathbf{W}_{\text{embed}}x_t + b) \quad (4.98)$$

where:

- $\mathbf{W}_{\text{recurrent}}$ is shared across all time steps
- Same transformation matrix processes all temporal updates
- Reflects the uniform nature of time evolution

Residual Stream (Computational Time):

$$h_t = h_{t-1} + \sigma(\mathbf{W}_{\text{residual}}^{(t)}h_{t-1} + b^{(t)}) \quad (4.99)$$

where:

- $\mathbf{W}_{\text{residual}}^{(t)}$ can be different for each step t
- Each computational step can learn a specialized transformation
- Reflects the progressive nature of computation

4.9.2 Rationale for the Difference

This distinction arises from different purposes:

- Memory Stream:
 - Processing real sequences that follow consistent rules
 - Same temporal dynamics apply at each step
 - Natural to share parameters across time
- Residual Stream:
 - Each step might need different refinements
 - Early steps might handle basic features
 - Later steps might perform specialized adjustments

For example:

- Memory Stream ("I love machine learning")
 - Same $\mathbf{W}_{\text{recurrent}}$ processes each word

- Temporal relationships follow consistent patterns
- Residual Stream ("Barack" \rightarrow "Obama")
 - $\mathbf{W}_{\text{residual}}^{(1)}$ might extract name features
 - $\mathbf{W}_{\text{residual}}^{(2)}$ might add presidential features
 - $\mathbf{W}_{\text{residual}}^{(3)}$ might refine for surname prediction

4.9.3 Adding Recurrence/Residual to CNNs

Standard convolutional layer:

$$h_{ij}^{(l)} = \sigma\left(\sum_{\Delta i, \Delta j} W_{\Delta i, \Delta j}^{(l)} h_{i+\Delta i, j+\Delta j}^{(l-1)} + b^{(l)}\right) \quad (4.100)$$

We can add computational steps within each layer:

$$h_{ij}^{(l,0)} = \sigma\left(\sum_{\Delta i, \Delta j} W_{\Delta i, \Delta j}^{(l)} h_{i+\Delta i, j+\Delta j}^{(l-1, T_{l-1})} + b^{(l)}\right) \quad (\text{init}) \quad (4.101)$$

$$h_{ij}^{(l,t)} = h_{ij}^{(l,t-1)} + \sigma\left(\sum_{\Delta i, \Delta j} W_{\Delta i, \Delta j}^{(l,t)} h_{i+\Delta i, j+\Delta j}^{(l,t-1)} + b^{(l,t)}\right) \quad \text{for } t = 1, \dots, T_l \quad (4.102)$$

Key features:

- Layer l allocated T_l computational steps
- Residual stream $h_{ij}^{(l,t-1)} +$ maintains information flow
- Different kernels $W^{(l,t)}$ for different steps
- Initial features refined through multiple steps

4.9.4 Computational Structure

The computation proceeds as:

- First get features (e.g., eyes, nose, mouth) through standard convolution
- Then refine these features through T_l steps
- Each step can learn different refinement patterns
- Output feeds to next layer after T_l steps

4.9.5 Advantages

This architecture offers:

- Flexible computation depth per layer
- Learned refinement strategies
- Strong gradient flow through residual stream
- Progressive feature improvement

The network learns:

- Initial feature extraction ($W^{(l)}$)
- Feature refinement patterns ($W^{(l,t)}$)
- How to effectively use allocated steps (T_l)
- Multi-scale feature processing

4.10 Vanilla RNN vs Temporal CNN

4.10.1 With or Without Horizontal Recurrent Connections

Vanilla two-layer RNN:

$$h_t^{(1)} = \tanh(\mathbf{W}_{\text{recurrent}}^{(1)} h_{t-1}^{(1)} + \mathbf{W}_{\text{embed}}^{(1)} x_t + b^{(1)}) \quad (4.103)$$

$$h_t^{(2)} = \tanh(\mathbf{W}_{\text{recurrent}}^{(2)} h_{t-1}^{(2)} + \mathbf{W}_{\text{associative}}^{(2)} h_t^{(1)} + b^{(2)}) \quad (4.104)$$

Temporal CNN version:

$$h_t^{(1)} = \tanh(\mathbf{W}_{\text{embed}}^{(1)} x_t + b^{(1)}) \quad (4.105)$$

$$h_t^{(2)} = \sigma\left(\sum_{\Delta t} \mathbf{W}_{\Delta t} h_{t-\Delta t}^{(1)} + b^{(2)}\right) \quad (4.106)$$

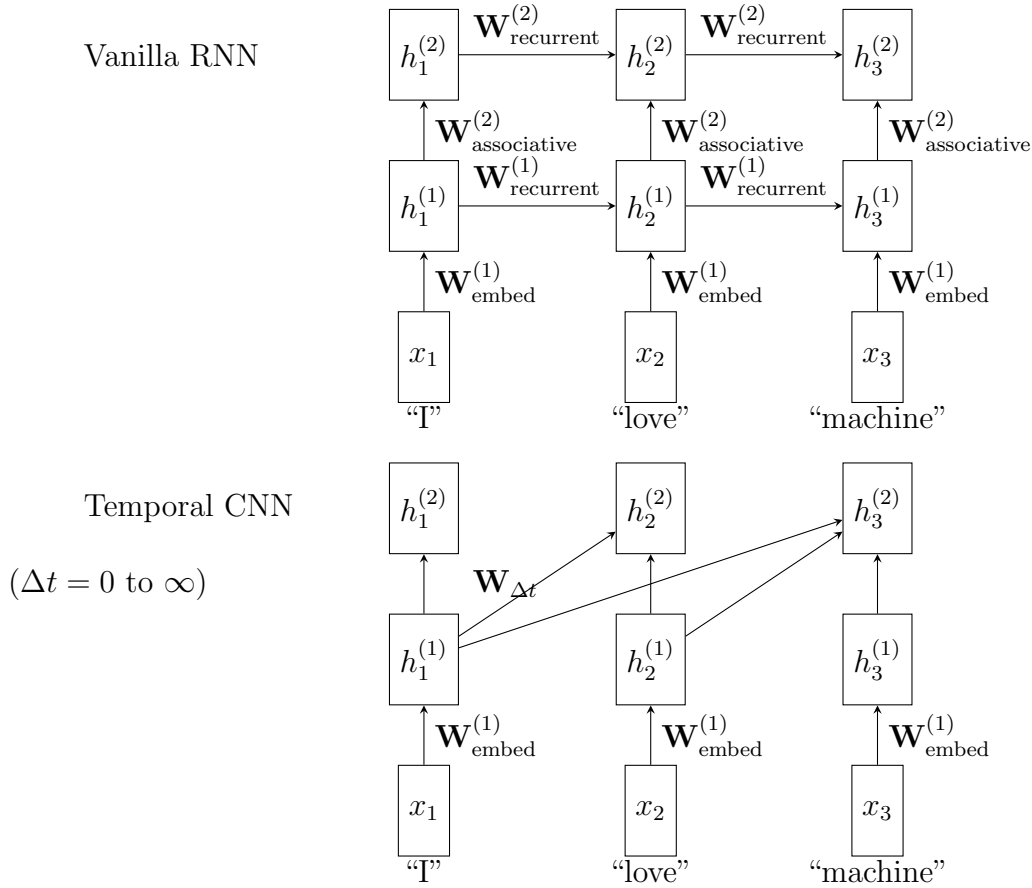


Figure 4.8: Comparison of architectures. Top: Vanilla RNN with recurrent connections. Bottom: Temporal CNN with unlimited receptive field ($h^{(2)}_t$ can attend to all previous $h^{(1)}_{t-\Delta t}$).

4.10.2 Key Differences

Training:

- RNN:
 - Must process sequence sequentially
 - Backprop through time is sequential
 - Needs to maintain all intermediate states
 - Gradient flow through recurrent connections
- Temporal CNN:
 - Can process all positions in parallel
 - Backprop is parallel within kernel width
 - Fixed receptive field size
 - Direct gradient flow through conv weights

Inference:

- RNN:
 - Uses previous hidden states
 - Can handle arbitrary sequence lengths
- Temporal CNN:
 - Uses fixed time window
 - Limited by kernel receptive field

4.11 State Space Models

4.11.1 Basic Formulation

A linear state space model consists of:

$$h_t = \mathbf{A}h_{t-1} + \mathbf{B}x_t \quad (\text{state transition}) \quad (4.107)$$

$$y_t = \mathbf{C}h_t \quad (\text{output mapping}) \quad (4.108)$$

This corresponds to RNN weights:

- $\mathbf{A} = \mathbf{W}_{\text{recurrent}}$: temporal evolution
- $\mathbf{B} = \mathbf{W}_{\text{embed}}$: input projection
- $\mathbf{C} = \mathbf{W}_{\text{unembed}}$: output projection

4.11.2 Unrolled Form

We can unroll the recursion:

$$h_t = \mathbf{A}h_{t-1} + \mathbf{B}x_t \quad (4.109)$$

$$= \mathbf{A}(\mathbf{A}h_{t-2} + \mathbf{B}x_{t-1}) + \mathbf{B}x_t \quad (4.110)$$

$$= \mathbf{A}^2h_{t-2} + \mathbf{A}\mathbf{B}x_{t-1} + \mathbf{B}x_t \quad (4.111)$$

$$= \mathbf{A}^3h_{t-3} + \mathbf{A}^2\mathbf{B}x_{t-2} + \mathbf{A}\mathbf{B}x_{t-1} + \mathbf{B}x_t \quad (4.112)$$

$$= \sum_{\Delta t=0}^{\infty} \mathbf{A}^{\Delta t} \mathbf{B}x_{t-\Delta t} \quad (4.113)$$

Therefore:

$$y_t = \mathbf{C}h_t \quad (4.114)$$

$$= \mathbf{C} \sum_{\Delta t=0}^{\infty} \mathbf{A}^{\Delta t} \mathbf{B}x_{t-\Delta t} \quad (4.115)$$

$$= \sum_{\Delta t=0}^{\infty} \mathbf{W}_{\Delta t} x_{t-\Delta t} \quad (4.116)$$

where $\mathbf{W}_{\Delta t} = \mathbf{C}\mathbf{A}^{\Delta t}\mathbf{B}$ are the convolutional weights.

4.11.3 Unifying Recurrent and Convolutional Views

The state space model unifies:

- Recurrent view:
 - Sequential updates through $h_t = \mathbf{A}h_{t-1} + \mathbf{B}x_t$
 - Constant memory requirement
 - Efficient for autoregressive generation
- Convolutional view:
 - Parallel computation through $y_t = \sum_{\Delta t} \mathbf{W}_{\Delta t} x_{t-\Delta t}$
 - Explicit temporal dependencies
 - Efficient for parallel training

4.11.4 Computational Advantages

This dual nature enables:

- Fast training:
 - Use convolutional form
 - Process all time steps in parallel
 - Direct backpropagation through time
- Fast inference:
 - Use recurrent form
 - Constant memory per step
 - Sequential generation

Moreover, the relationship $\mathbf{W}_{\Delta t} = \mathbf{C}\mathbf{A}^{\Delta t}\mathbf{B}$ provides:

- Parameter efficiency through factorization
- Structured temporal dependencies
- Long-range interactions through powers of \mathbf{A}

4.12 Continuous-Time State Space Model

4.12.1 Memory Stream Form

Starting with continuous-time SSM:

$$h'(t) = \mathbf{A}h(t) + \mathbf{B}x(t) \quad (4.117)$$

For small dt , using first-order approximation:

$$\frac{h(t+dt) - h(t)}{dt} \approx \mathbf{A}h(t) + \mathbf{B}x(t) \quad (4.118)$$

$$h(t+dt) - h(t) \approx (\mathbf{A}dt)h(t) + \mathbf{B}x(t)dt \quad (4.119)$$

Therefore:

$$h(t+dt) = (\mathbf{I} + \mathbf{A}dt)h(t) + \mathbf{B}x(t)dt \quad (4.120)$$

This is in memory stream form where $(\mathbf{I} + \mathbf{A}dt)$ represents the residual update.

4.12.2 Zero-Order Hold (ZOH) Discretization

Consider interval $[t, t + \Delta]$ divided into N steps of size $dt = \Delta/N$:

- Time variable: $\tau \in [t, t + \Delta]$
- Input held constant: $x(\tau) = x(t)$ for $\tau \in [t, t + \Delta]$

Homogeneous Case ($\mathbf{B}x(t) = 0$)

The homogeneous equation:

$$h'(\tau) = \mathbf{A}h(\tau) \quad (4.121)$$

Using small time steps:

$$h(t+dt) = (\mathbf{I} + \mathbf{A}dt)h(t) \quad (4.122)$$

$$h(t+2dt) = (\mathbf{I} + \mathbf{A}dt)h(t+dt) \quad (4.123)$$

$$= (\mathbf{I} + \mathbf{A}dt)^2 h(t) \quad (4.124)$$

Continuing for N steps:

$$h(t+\Delta) = (\mathbf{I} + \mathbf{A}\Delta/N)^N h(t) \quad (4.125)$$

As $N \rightarrow \infty$:

$$h(t+\Delta) = e^{\mathbf{A}\Delta} h(t) \quad (4.126)$$

General Case

For the inhomogeneous equation with constant input:

$$h'(\tau) = \mathbf{A}h(\tau) + \mathbf{B}x(t) \quad (4.127)$$

Let:

$$g(\tau) = h(\tau) - r(t) \quad (4.128)$$

where $r(t)$ is a particular solution satisfying:

$$\mathbf{A}r(t) + \mathbf{B}x(t) = 0 \quad (4.129)$$

Therefore:

$$r(t) = -\mathbf{A}^{-1}\mathbf{B}x(t) \quad (4.130)$$

Then:

$$g'(\tau) = h'(\tau) \quad (4.131)$$

$$= \mathbf{A}h(\tau) + \mathbf{B}x(t) \quad (4.132)$$

$$= \mathbf{A}(g(\tau) + r(t)) + \mathbf{B}x(t) \quad (4.133)$$

$$= \mathbf{A}g(\tau) \quad (4.134)$$

This gives us homogeneous equation for $g(\tau)$, so:

$$g(t + \Delta) = e^{\mathbf{A}\Delta}g(t) \quad (4.135)$$

$$h(t + \Delta) - r(t) = e^{\mathbf{A}\Delta}(h(t) - r(t)) \quad (4.136)$$

Therefore, the general discretization is:

$$h(t + \Delta) = e^{\mathbf{A}\Delta}h(t) + (e^{\mathbf{A}\Delta} - \mathbf{I})\mathbf{A}^{-1}\mathbf{B}x(t) \quad (4.137)$$

This gives us the discrete-time SSM:

$$\mathbf{A}_d = e^{\mathbf{A}\Delta} \quad (4.138)$$

$$\mathbf{B}_d = (e^{\mathbf{A}\Delta} - \mathbf{I})\mathbf{A}^{-1}\mathbf{B} \quad (4.139)$$

such that:

$$h(t + \Delta) = \mathbf{A}_d h(t) + \mathbf{B}_d x(t) \quad (4.140)$$

4.13 Mamba: Selective State Space Model

4.13.1 Key Innovation

Mamba generalizes the linear SSM by making the state matrices input-dependent:

$$\Delta_t, \mathbf{A}_t, \mathbf{B}_t, \mathbf{C}_t = \text{MLPs}(x_t) \quad (\text{selective parameters}) \quad (4.141)$$

$$h_t = e^{\mathbf{A}_t \Delta_t} h_{t-1} + \mathbf{B}_t x_t \quad (\text{state update}) \quad (4.142)$$

$$y_t = \mathbf{C}_t h_t \quad (\text{output}) \quad (4.143)$$

Key features:

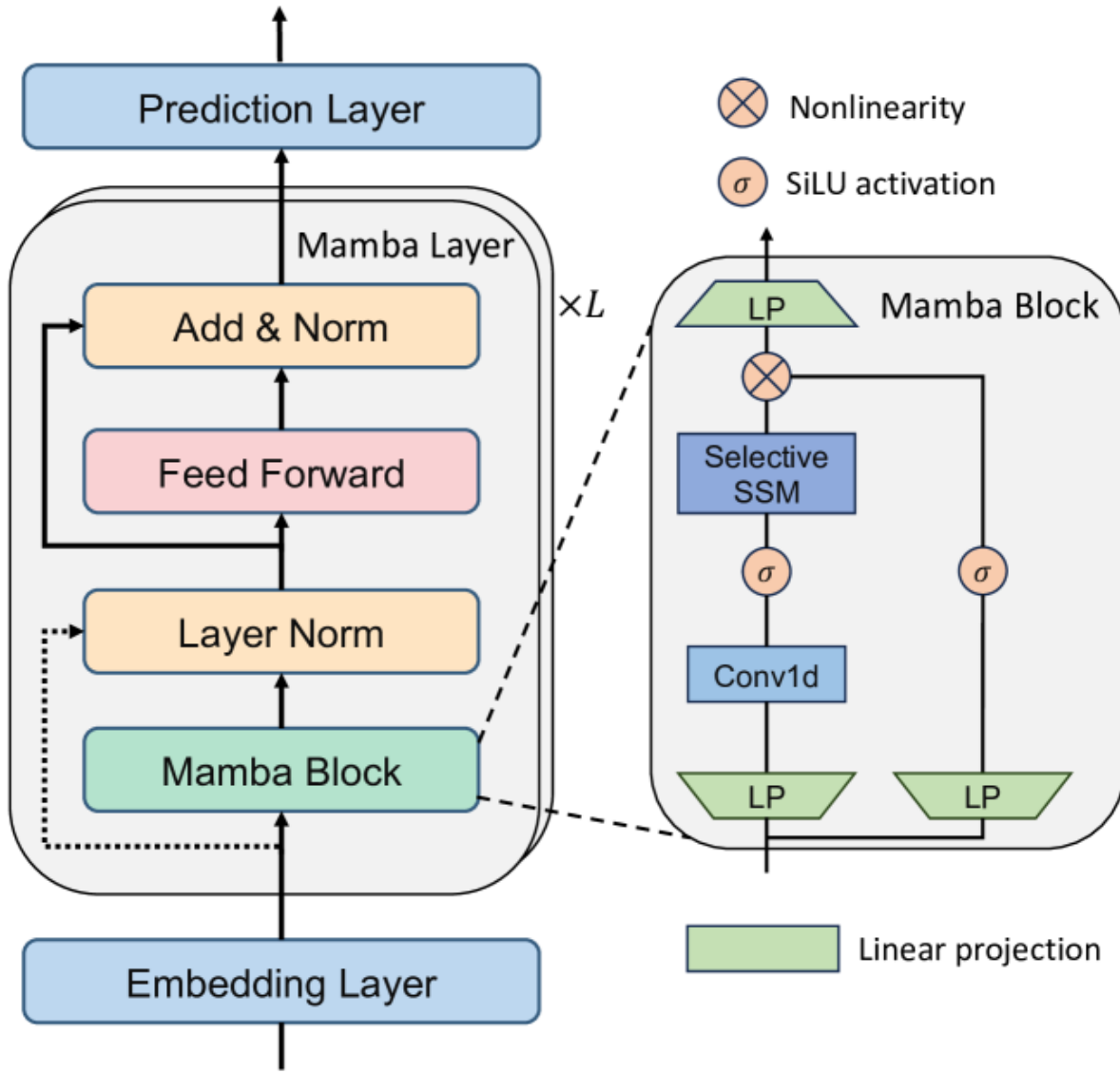


Figure 4.9: Mamba

- Input-dependent discretization Δ_t
 - Different time steps for different inputs
 - Selective processing of sequence elements
- Input-dependent state matrices $\mathbf{A}_t, \mathbf{B}_t, \mathbf{C}_t$
 - Adaptive state evolution
 - Selective information flow
- Efficient hardware implementation
 - Linear recurrence for fast inference

– Parallel computation for training

Mamba combines:

- SSM’s dual recurrent/convolutional nature
- Transformer-like selective processing
- Efficient linear state evolution

4.14 Quantum Mechanics as RNN

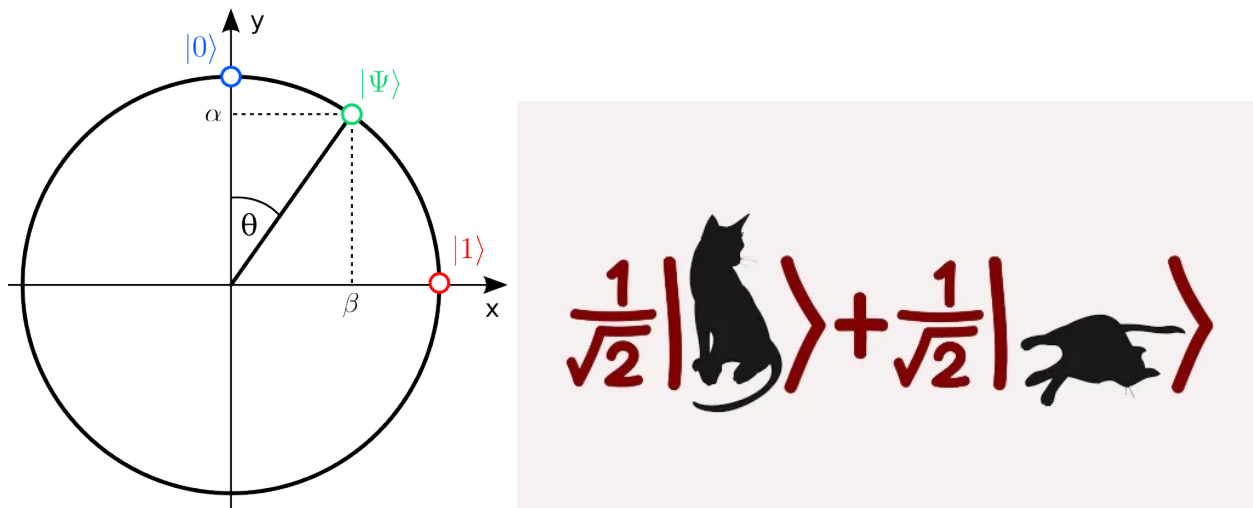


Figure 4.10: Quantum mechanics as linear recurrent neural network. The state vector rotates in the hidden space according to the Schrodinger equation. The state vector is a superposition of orthogonal basis vectors which are embeddings of observable classical states. $|state\rangle$ denotes the embedding of “state”, e.g., 0 and 1 in qubit, or “alive” and “dead” of Schrodinger cat.

Quantum mechanics can be conceptualized as a special type of recurrent neural network (RNN) that does not involve learning but rather explains observational data. The system requires three essential components: an input layer through $\mathbf{W}_{\text{embed}}$ that encodes classical measurements into quantum states (wave function collapse), a hidden layer that evolves according to the Schrödinger equation ($\hbar'(t) = -i\mathbf{H}h(t)$), and an output layer through $\mathbf{W}_{\text{unembed}}$ that projects quantum states back to classical observations via the Born rule (implemented as squared-softmax). The hidden layer represents fundamental reality itself—not a simulation—functioning like a cosmic game engine, while our classical reality emerges as a rendered display through the measurement-collapse cycle. This formulation emphasizes that quantum mechanics is meaningless without an observer who exists outside the universe and interfaces with it through measurements, and that the universe inherently “plays dice” through the probabilistic nature of the Born rule.

4.14.1 Basic Structure

Quantum mechanics can be formulated as a special RNN:

$$h(0) = \mathbf{W}_{\text{embed}}x(0) \quad (\text{Initial observation}) \quad (4.144)$$

$$h'(t) = \mathbf{W}_{\text{recurrent}}h(t) \quad (\text{Schrodinger evolution}) \quad (4.145)$$

$$s(t) = \mathbf{W}_{\text{unembed}}h(t) \quad (\text{Amplitude (logits)}) \quad (4.146)$$

$$p(t) = \text{squared-softmax}(s(t)) \quad (\text{Born rule}) \quad (4.147)$$

$$x(t) \sim p(t) \quad (\text{Universe does plays dice}) \quad (4.148)$$

$$h(t) = \mathbf{W}_{\text{embed}}x(t) \quad (\text{Bohr Wave function collapse}) \quad (4.149)$$

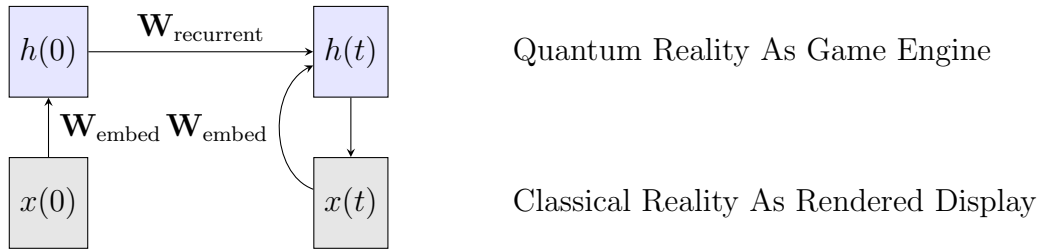


Figure 4.11: Minimal quantum RNN diagram showing the interaction between quantum and classical reality through state transitions.

4.14.2 Special Properties

The weight matrices have specific physical meanings:

Evolution Matrix

$$\mathbf{W}_{\text{recurrent}} = -i\mathbf{H} \quad (4.150)$$

where:

- i is the imaginary unit
- \mathbf{H} is the Hamiltonian operator
- \mathbf{H} is Hermitian: $\mathbf{H}^\dagger = \mathbf{H}$
- Evolution preserves norm: $e^{-i\mathbf{H}t}$ is unitary (rotation), thus $\|s(t)\|^2$ is constant for squared-softmax.
- Eigen-values of \mathbf{H} can be discrete, thus the word “quantum”.

Measurement Basis

$$\mathbf{W}_{\text{embed}} = \mathbf{E} \quad (4.151)$$

$$\mathbf{W}_{\text{unembed}}^\top = \mathbf{E} \quad (4.152)$$

where:

- \mathbf{E} has orthogonal columns: $\mathbf{E}^\top \mathbf{E} = \mathbf{I}$
- Each column is a basis state
- Input encoding and measurement use same basis
- Orthogonality ensures proper measurement probabilities

4.14.3 Squared-Softmax and Born Rule

The squared-softmax function implements Born's rule for quantum measurement:

$$s(t) = \mathbf{E}^\top h(t) \quad (\text{measurement amplitudes}) \quad (4.153)$$

$$p_i(t) = \frac{|s_i(t)|^2}{\|s(t)\|^2} \quad (\text{Born rule}) \quad (4.154)$$

where $\|s(t)\|^2 = \sum_i |s_i(t)|^2$. That is, we use square instead of exponential to calculate probability of outcome i , and that's why we call it squared-softmax.

4.14.4 Norm Conservation in Quantum Measurement

The squared-softmax probability comes from two key properties:

Unitary Evolution

The quantum state evolution preserves norm:

$$h(t) = e^{\mathbf{W}_{\text{recurrent}} t} h(0) = e^{-i\mathbf{H}t} h(0) \quad (4.155)$$

$$\|h(t)\|^2 = h(t)^\dagger h(t) = h(0)^\dagger e^{i\mathbf{H}t} e^{-i\mathbf{H}t} h(0) = \|h(0)\|^2 \quad (4.156)$$

because $e^{-i\mathbf{H}t}$ is unitary (pure rotation in complex space).

Orthogonal Measurement

The measurement basis preserves norm:

$$s(t) = \mathbf{E}^\top h(t) \quad (4.157)$$

$$\|s(t)\|^2 = s(t)^\dagger s(t) = h(t)^\dagger \mathbf{E} \mathbf{E}^\top h(t) = h(t)^\dagger h(t) = \|h(t)\|^2 \quad (4.158)$$

because \mathbf{E} has orthogonal columns ($\mathbf{E} \mathbf{E}^\top = \mathbf{I}$).

Therefore:

- $\|h(t)\|^2$ is constant (unitary evolution)
- $\|s(t)\|^2 = \|h(t)\|^2$ (orthogonal measurement)
- $p_i(t) = |s_i(t)|^2 / \|s(t)\|^2$ gives valid probabilities

This ensures:

- Conservation of probability: $\sum_i p_i(t) = 1$
- Time-independent normalization
- Basis-independent total probability

4.14.5 Hidden Layer as Fundamental Reality

The hidden state $h(t)$ represents the true quantum reality:

- Like a cosmic game engine:
 - Evolution by $-i\mathbf{H}$ between measurements
 - Unitary rotation preserves quantum information
 - Contains complete quantum state information
- Two modes of evolution:
 - Continuous: Schrödinger evolution ($h'(t) = -i\mathbf{H}h(t)$)
 - Discrete: Wave function collapse ($h(t) = \mathbf{E}x(t)$)
- Not a simulation but reality itself:
 - Mathematical description of nature
 - Alternates between evolution and measurement
 - Collapse creates new initial conditions

4.14.6 Interface to Classical Reality

The basis matrix \mathbf{E} provides the quantum-classical interface:

- Measurement process (\mathbf{E}^\top):
 - Projects quantum state to measurement basis
 - Generates measurement amplitudes $s(t)$
 - Probabilities through Born rule
- Wave function collapse (\mathbf{E}):
 - Reinstates pure state after measurement
 - Maps classical outcome to new quantum state
 - Starts new cycle of quantum evolution

4.14.7 The Role of the Observer

The observer plays a central role through measurement:

- Measurement and collapse cycle:
 - Measure: $x(t) \sim \text{squared-softmax}(\mathbf{E}^\top h(t))$
 - Collapse: $h(t) = \mathbf{E}x(t)$
 - Evolution continues from new state
- Basis choice:
 - Observer selects measurement basis \mathbf{E}
 - Choice determines possible outcomes
 - Different bases reveal different aspects
- Creation of classical reality:
 - Measurement actualizes potential outcomes
 - Collapse ensures definite classical states
 - Each measurement creates new branch

4.14.8 Classical Reality as Rendered Display

Classical reality emerges through the measurement-collapse cycle:

- Continuous process:
 - Evolution in hidden layer ($-i\mathbf{H}$)
 - Measurement projects through interface (\mathbf{E}^\top)
 - Collapse reinitializes state (\mathbf{E})
- Creation of classical properties:
 - No definite values until measured
 - Measurement creates classical reality
 - Collapse ensures consistent next evolution

4.14.9 Philosophical Implications

This view of quantum mechanics suggests:

- Fundamental hidden layer:
 - Quantum state evolves unitarily
 - Measurements interrupt evolution

- Collapse restarts cycle
- Interface determines experience:
 - **E** connects quantum and classical
 - Measurement creates classical properties
 - Collapse maintains quantum-classical consistency
- Participatory universe:
 - Observers create reality through measurement
 - Each observation collapses and reinitializes
 - Classical reality emerges through observation cycle

Chapter 5

Transformer and GPT

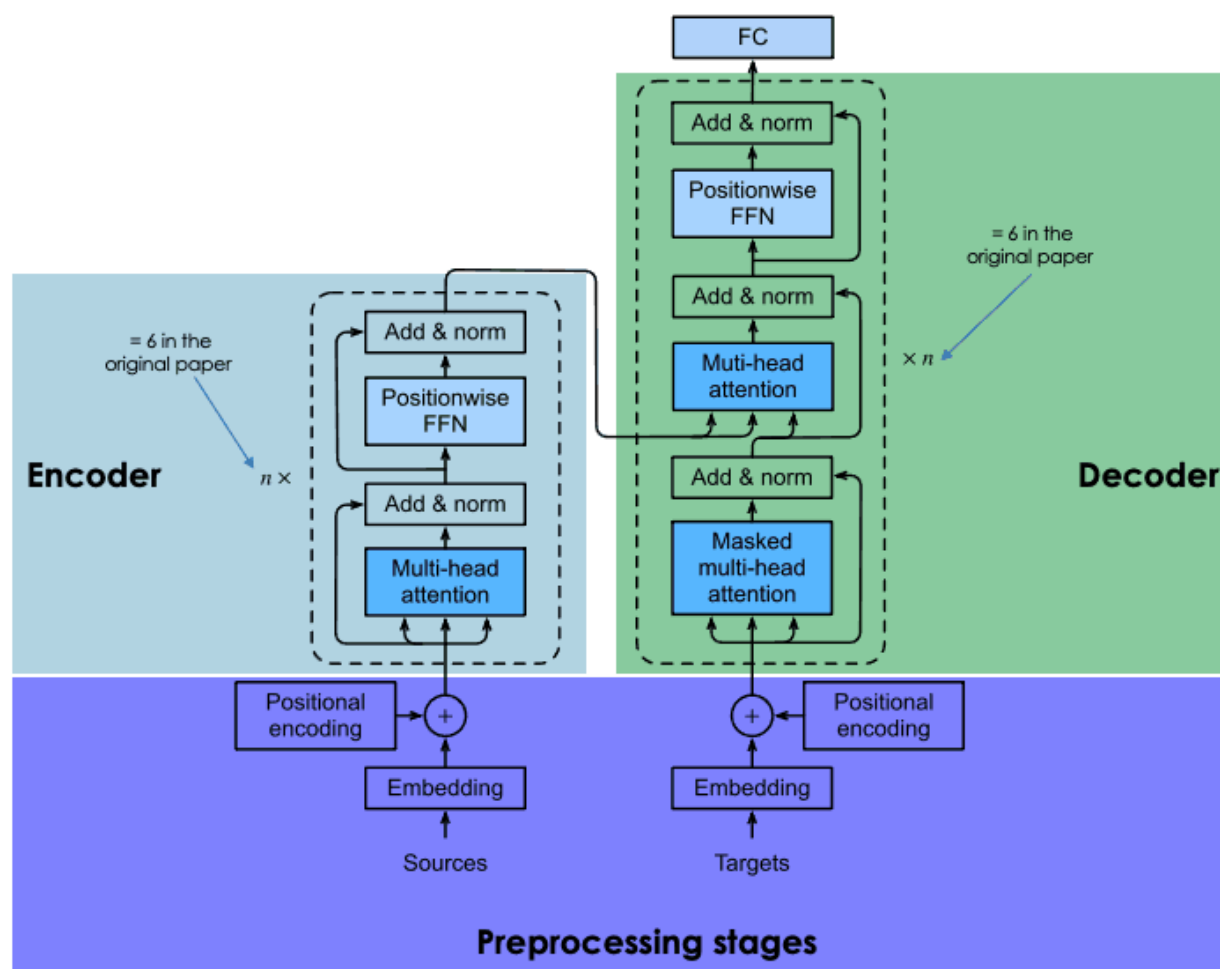


Figure 5.1: Transformer

Chapter Overview

This chapter explores Transformers and GPT architectures through a vector-centered perspective, emphasizing how information is organized through superposition in high-dimensional spaces and processed through complementary retrieval mechanisms. At its core, the chapter explains how neural networks can represent complex information by superimposing multiple aspects in a single vector, allowing concepts like "Barack Obama" to simultaneously encode political, biographical, and temporal information in an efficiently retrievable form. The residual stream serves as a central assembly line where this superposed information is progressively refined and enhanced through alternating retrieval mechanisms: attention retrieves information from context by dynamically weighting relevant parts of the input sequence, while MLPs retrieve learned patterns from their weight matrices acting as associative memory. This dual retrieval process - contextual through attention and memorized through MLPs - enables Transformers to combine document-specific information with general world knowledge. The chapter then examines the training process of Large Language Models, describing the two-stage approach of unsupervised pre-training followed by instruction fine-tuning, where the model learns to align its capabilities with natural language instructions. It details how Reinforcement Learning from Human Feedback (RLHF) further refines model behavior by learning from human preferences, using a reward model trained on human comparisons to guide policy optimization. The material also covers key architectural innovations across the Transformer family, scaling laws governing model performance, and specialized variants for vision and multimodal tasks. Throughout, the chapter emphasizes how the interplay between superposition, residual stream processing, and complementary retrieval mechanisms enables these models to process and generate human-like text while being guided by human feedback.

5.1 Embedding, Thought Vectors and Distributed Representations

We continue our vector-centered view of neural networks, now focusing on the Transformer architecture, specifically the GPT (Generative Pre-trained Transformer) variant for next word prediction. The fundamental objects remain vectors, but with a new mechanism for information flow through attention rather than recurrence.

A neural network can transform a discrete input (e.g., "Barack Obama") into a high-dimensional vector, also called a thought vector or distributed representation or embedding:

$$h = \text{NN}(\text{"Barack Obama"}) \in \mathbb{R}^d \quad (5.1)$$

5.1.1 Superposition Nature

This thought vector can be a superposition of nearly orthogonal components:

$$h = g_1 + g_2 + g_3 + g_4 + \dots \quad (\text{superposition}) \quad (5.2)$$

$$g_i^\top g_j \approx 0 \text{ for } i \neq j \quad (\text{near orthogonality}) \quad (5.3)$$

Examples of components:

- g_1 : Democratic politician
 - Party affiliation
 - Political ideology
 - Legislative history
- g_2 : U.S. President
 - Executive role
 - Time period (2009-2017)
 - Presidential powers
- g_3 : Personal background
 - Born in Hawaii
 - Harvard Law graduate
 - Community organizer
- g_4 : Family relationships
 - Married to Michelle
 - Father of Malia and Sasha
 - Family history

5.1.2 Information Extraction

We can extract specific aspects using projection matrices:

$$g_{\text{party}} = \mathbf{W}_{\text{party}} h \quad (\text{extract political party}) \quad (5.4)$$

$$g_{\text{edu}} = \mathbf{W}_{\text{education}} h \quad (\text{extract education}) \quad (5.5)$$

$$g_{\text{time}} = \mathbf{W}_{\text{timeline}} h \quad (\text{extract temporal info}) \quad (5.6)$$

These extracted features can be converted to human-readable form:

$$s_{\text{party}} = \mathbf{W}_{\text{unembed}} g_{\text{party}} \quad (\text{e.g., "Democrat"}) \quad (5.7)$$

$$s_{\text{edu}} = \mathbf{W}_{\text{unembed}} g_{\text{edu}} \quad (\text{e.g., "Harvard Law"}) \quad (5.8)$$

$$s_{\text{time}} = \mathbf{W}_{\text{unembed}} g_{\text{time}} \quad (\text{e.g., "44th President"}) \quad (5.9)$$

5.1.3 Properties

This distributed representation has several key properties:

- Superposition:
 - Multiple aspects coexist
 - Information adds linearly
 - No interference due to orthogonality
- Extractability:
 - Different matrices extract different aspects
 - Clean separation of features
 - Interpretable outputs
- Compositionality:
 - Components combine naturally
 - Rich internal structure
 - Hierarchical organization

5.1.4 Neural Operations

The network learns to:

- Embed: map inputs to distributed representations
- Transform: manipulate thought vectors
- Extract: project onto relevant subspaces
- Unembed: map back to interpretable outputs

This framework underlies modern neural architectures, where:

- Embeddings create thought vectors
- Hidden layers transform representations
- Attention combines relevant aspects
- Output layers extract specific information

5.1.5 Residual Stream: Building Superposition

The residual stream can be viewed as an assembly line that gradually builds up a superposed representation:

$$h^{(1)} = \mathbf{W}_{\text{embed}}(\text{"Barack Obama"}) \quad (\text{initial embedding}) \quad (5.10)$$

$$h^{(l+1)} = h^{(l)} + g^{(l)} \quad (\text{add new features}) \quad (5.11)$$

$$g^{(l)} = \text{Layer}_l(h^{(l)}) \quad (\text{compute new aspect}) \quad (5.12)$$

5.1.6 Assembly Line Process

Each layer adds new aspects to the representation:

$$h^{(1)} = \text{basic name embedding} \quad (5.13)$$

$$h^{(2)} = h^{(1)} + g_{\text{politician}} \quad (5.14)$$

$$h^{(3)} = h^{(2)} + g_{\text{president}} \quad (5.15)$$

$$h^{(4)} = h^{(3)} + g_{\text{time_period}} \quad (5.16)$$

$$h^{(5)} = h^{(4)} + g_{\text{achievements}} \quad (5.17)$$

Key properties:

- Incremental refinement:
 - Each layer adds specific features
 - Previous information preserved
 - Aspects accumulate naturally
- Near orthogonality:
 - New features $g^{(l)}$ nearly orthogonal
 - Minimal interference
 - Clean superposition
- Assembly line metaphor:
 - Each station adds specific component
 - Previous work preserved
 - Quality control at each step

5.2 Transformer Residual Stream

The Transformer processes information through a residual stream, which acts as an assembly line for building complex representations. Each layer reads from this stream, adds new features, and writes back the enhanced representation.

5.2.1 Dual Retrieval Mechanism

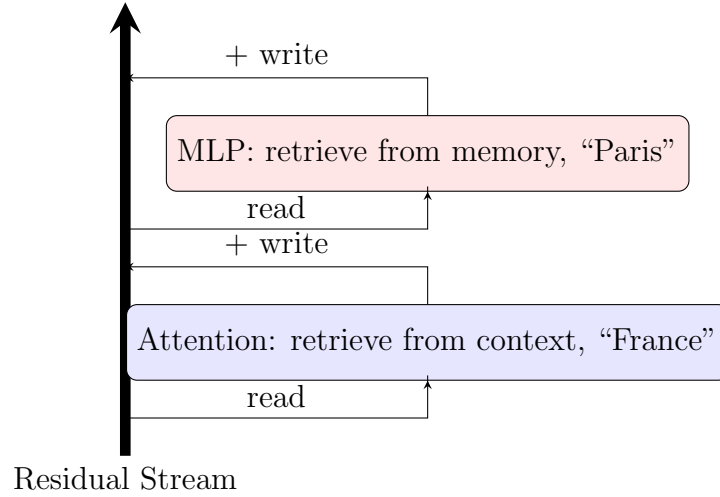


Figure 5.2: The capital of France is ... Each layer of Transformer interacts with the residual stream.

The Transformer alternates between two types of information retrieval:

1. Attention: Retrieves from current context
 - Reads current representation from stream
 - Queries relevant context ("France")
 - Writes back context-enhanced features
 - Like looking up information in current notes
2. MLP: Retrieves from learned knowledge
 - Reads enhanced representation
 - Associates learned patterns ("Paris")
 - Writes back knowledge-enhanced features
 - Like consulting permanent memory

5.2.2 Assembly Line Process

For the query "The capital of France is", the residual stream gradually builds meaning:

$$h^{(l)} = \text{current representation} \quad (5.18)$$

$$h^{(l+\frac{1}{2})} = h^{(l)} + \text{attention}(\text{"France in context"}) \quad (5.19)$$

$$h^{(l+1)} = h^{(l+\frac{1}{2})} + \text{MLP}(\text{"capital relationship"}) \quad (5.20)$$

Each layer adds new aspects through superposition:

- Attention adds:
 - Context-relevant features
 - Relational information
 - Current document knowledge
- MLP adds:
 - Learned associations
 - World knowledge
 - Pattern completions

Example usage:

- Input: "The capital of France is"
- Attention: focuses on "France" in context
- MLP: retrieves learned association "Paris"
- Combined: generates appropriate completion

The residual stream maintains these accumulated features while allowing each layer to add new aspects without disrupting existing information. This assembly line structure enables the Transformer to build increasingly sophisticated representations through successive refinements.

5.2.3 Two Forms of Retrieval

The Transformer alternates between two fundamentally different types of retrieval:

Retrieval from Context

$$h_t^{(l+\frac{1}{2})} = h_t^{(l)} + \text{Attention}(h_1^{(l)}, \dots, h_i^{(l)}, \dots, h_t^{(l)}) \quad (\text{context lookup}) \quad (5.21)$$

$$= h_t^{(l)} + \sum_{i=1}^t \alpha_{ti} \mathbf{W}_v h_i^{(l)} \quad (\text{weighted sum}) \quad (5.22)$$

Key properties:

- Uses multiple vectors $h_i^{(l)}$ from context
- Dynamic weights α_{ti} computed from content
- Like searching through previous notes
- Information flows between positions

Retrieval from Associative Memory

$$h_t^{(l+1)} = h_t^{(l+\frac{1}{2})} + \text{MLP}(h_t^{(l+\frac{1}{2})}) \quad (\text{memory lookup}) \quad (5.23)$$

$$= h_t^{(l+\frac{1}{2})} + \mathbf{W}_2 \sigma(\mathbf{W}_1 h_t^{(l+\frac{1}{2})} + b_1) + b_2 \quad (\text{learned association}) \quad (5.24)$$

Key properties:

- Operates on single vector $h_t^{(l+\frac{1}{2})}$
- Uses fixed learned weights $\mathbf{W}_1, \mathbf{W}_2$
- Like consulting permanent memory
- No interaction with other positions

5.2.4 Complementary Nature

These retrievals serve different purposes:

- Context retrieval:
 - Integrates information across sequence
 - Adapts to current content
 - Handles dynamic relationships
 - Multiple vector operation
- Memory retrieval:
 - Applies learned knowledge
 - Uses fixed associations
 - Position-independent processing
 - Single vector operation

Example for "The capital of France is":

$$\text{Context :} \quad h_t^{(l+\frac{1}{2})} = h_t^{(l)} + \sum_{i=1}^t \alpha_{ti} \mathbf{W}_v h_i^{(l)} \quad (\text{find "France"}) \quad (5.25)$$

$$\text{Memory :} \quad h_t^{(l+1)} = h_t^{(l+\frac{1}{2})} + \text{MLP}(h_t^{(l+\frac{1}{2})}) \quad (\text{recall "Paris"}) \quad (5.26)$$

This alternation between multi-vector context operations and single-vector memory operations allows the Transformer to combine:

- Document-specific information
- General world knowledge
- Dynamic relationships
- Learned patterns

5.2.5 Attention Mechanism

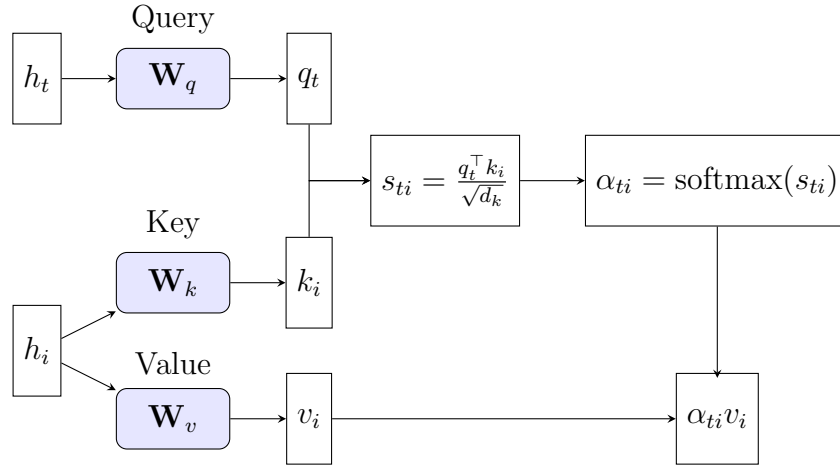


Figure 5.3: Attention mechanism with arrow touching only the upper side of the weighted value box.

The attention weights α_{ti} determine how much information to retrieve from each context position:

Query-Key-Value Computation

$$q_t = \mathbf{W}_q h_t^{(l)} \quad (\text{query: what to look for}) \quad (5.27)$$

$$k_i = \mathbf{W}_k h_i^{(l)} \quad (\text{key: how to find it}) \quad (5.28)$$

$$v_i = \mathbf{W}_v h_i^{(l)} \quad (\text{value: what to retrieve}) \quad (5.29)$$

Attention Weight Computation

$$s_{ti} = \frac{q_t^\top k_i}{\sqrt{d_k}} \quad (\text{scaled dot product}) \quad (5.30)$$

$$\alpha_{ti} = \text{softmax}(s_{ti}) \quad (\text{normalize weights}) \quad (5.31)$$

$$= \frac{\exp(s_{ti})}{\sum_{j=1}^t \exp(s_{tj})} \quad (\text{explicit form}) \quad (5.32)$$

Key components:

- Query-Key interaction:
 - $q_t^\top k_i$ measures relevance
 - Higher when vectors align
 - Captures similarity patterns

- Scaling factor:
 - $\sqrt{d_k}$ prevents extreme gradients
 - Maintains stable optimization
 - Controls attention sharpness
- Softmax normalization:
 - Ensures $\sum_i \alpha_{ti} = 1$
 - Creates probability distribution
 - Sharpens attention focus

Final Retrieval

$$h_t^{(l+\frac{1}{2})} = h_t^{(l)} + \sum_{i=1}^t \alpha_{ti} v_i \quad (\text{weighted combination}) \quad (5.33)$$

$$= h_t^{(l)} + \sum_{i=1}^t \frac{\exp(q_t^\top k_i / \sqrt{d_k})}{\sum_{j=1}^t \exp(q_t^\top k_j / \sqrt{d_k})} v_i \quad (\text{full form}) \quad (5.34)$$

Example mechanism for "The capital of France is":

- Query: looking for "capital of ___"
- Keys: match against each context word
- High α_{ti} when i points to "France"
- Value: retrieve relevant information

Multi-Head Implementation

In practice, use multiple attention heads in parallel:

$$\text{head}_h = \text{Attention}(\mathbf{W}_q^h h_t, \mathbf{W}_k^h h_i, \mathbf{W}_v^h h_i) \quad (5.35)$$

$$\text{MultiHead} = \mathbf{W}_o[\text{head}_1; \dots; \text{head}_H] \quad (5.36)$$

Benefits:

- Different heads can:
 - Focus on different patterns
 - Attend to different positions
 - Capture different relationships
- Parallel computation:
 - Efficient implementation
 - Rich feature combination
 - Multiple viewpoints

5.2.6 Mixture of Experts

Similar to using multiple attention heads, Transformers can employ multiple expert networks for enhanced memory retrieval:

Multi-Expert Computation

$$g_t = \text{Router}(h_t^{(l+\frac{1}{2})}) \quad (\text{compute expert weights}) \quad (5.37)$$

$$= \text{softmax}(\mathbf{W}_r h_t^{(l+\frac{1}{2})}) \quad (\text{routing logits}) \quad (5.38)$$

$$e_t^j = \text{MLP}_j(h_t^{(l+\frac{1}{2})}) \quad (\text{expert computations}) \quad (5.39)$$

$$h_t^{(l+1)} = h_t^{(l+\frac{1}{2})} + \sum_{j=1}^E g_t^j e_t^j \quad (\text{weighted combination}) \quad (5.40)$$

Key properties:

- Multiple expert networks:
 - Each specializes in different patterns
 - Parallel to attention heads
 - Divides computation across experts
- Router network:
 - Learns to select relevant experts
 - Creates sparse routing weights
 - Adapts to input content

Parallel to Multi-Head Attention

The model employs two forms of parallel processing:

$$\text{MultiHead} = \mathbf{W}_o[\text{head}_1; \dots; \text{head}_H] \quad (\text{parallel attention}) \quad (5.41)$$

$$\text{MultiExpert} = \sum_{j=1}^E g_t^j \text{MLP}_j \quad (\text{parallel experts}) \quad (5.42)$$

Complementary benefits:

- Multi-head attention:
 - Parallel processing across positions
 - Different attention patterns
 - Context-based routing

- Multi-expert recall:
 - Parallel processing across experts
 - Specialized knowledge bases
 - Content-based routing

Example for "The capital of France is":

- Attention heads: Find relevant context about France
- Experts: Different experts for:
 - Geographic knowledge
 - Political entities
 - Historical facts
- Combined: Rich multi-perspective retrieval

Implementation Details

Practical considerations:

- Load balancing:
 - Encourage uniform expert utilization
 - Prevent expert collapse
 - Balance computation
- Sparse routing:
 - Select top-k experts only
 - Reduce computation overhead
 - Maintain specialization
- Parallel computation:
 - Efficient hardware utilization
 - Scaled expert capacity
 - Reduced latency

5.3 Complete Transformer Architecture

5.3.1 Token and Position Embeddings

For input token x_t , we combine token and position embeddings:

$$e_{\text{token}} = \mathbf{W}_{\text{embed}} x_t \quad (\text{token embedding}) \quad (5.43)$$

$$e_{\text{pos}} = \mathbf{W}_{\text{pos}} t \quad (\text{position embedding}) \quad (5.44)$$

$$h_t^{(1)} = e_{\text{token}} + e_{\text{pos}} \quad (\text{combined embedding}) \quad (5.45)$$

t is one-hot vector.

Position embedding can be:

- Learned: $\mathbf{W}_{\text{pos}} \in \mathbb{R}^{d \times T_{\text{max}}}$
- Fixed sinusoidal:

$$e_{\text{pos}}(t, 2i) = \sin(t/10000^{2i/d}) \quad (\text{even dimensions}) \quad (5.46)$$

$$e_{\text{pos}}(t, 2i + 1) = \cos(t/10000^{2i/d}) \quad (\text{odd dimensions}) \quad (5.47)$$

5.3.2 Layer Processing

Each layer processes through residual stream:

$$h_t^{(l+\frac{1}{2})} = h_t^{(l)} + \text{Attention}(h_1^{(l)}, \dots, h_t^{(l)}) \quad (\text{context retrieval}) \quad (5.48)$$

$$h_t^{(l+1)} = h_t^{(l+\frac{1}{2})} + \text{MLP}(h_t^{(l+\frac{1}{2})}) \quad (\text{memory retrieval}) \quad (5.49)$$

where attention computes:

$$q_t = \mathbf{W}_q h_t^{(l)} \quad (5.50)$$

$$k_i = \mathbf{W}_k h_i^{(l)} \quad (5.51)$$

$$v_i = \mathbf{W}_v h_i^{(l)} \quad (5.52)$$

$$\alpha_{ti} = \text{softmax}\left(\frac{q_t^\top k_i}{\sqrt{d_k}}\right) \quad (5.53)$$

$$\text{Attention}(h_1^{(l)}, \dots, h_t^{(l)}) = \sum_{i=1}^t \alpha_{ti} v_i \quad (5.54)$$

5.3.3 Output Generation

Final unembedding for prediction:

$$h_t^{(L)} = \text{final representation} \quad (5.55)$$

$$s_t = \mathbf{W}_{\text{unembed}} h_t^{(L)} \quad (\text{logits}) \quad (5.56)$$

$$p_t = \text{softmax}(s_t) \quad (\text{probabilities}) \quad (5.57)$$

5.3.4 Backpropagation and Parallelization

The Transformer architecture enables highly efficient parallel training through two key aspects:

Attention Layer Parallelization

For a sequence of length T , backpropagation through attention follows:

$$\frac{\partial L}{\partial h_t^{(l)}} = \sum_{j=t}^T \frac{\partial L}{\partial h_j^{(l+\frac{1}{2})}} \frac{\partial h_j^{(l+\frac{1}{2})}}{\partial h_t^{(l)}} \quad (\text{backward flow}) \quad (5.58)$$

$$\frac{\partial L}{\partial \alpha_{ti}} = \frac{\partial L}{\partial h_t^{(l+\frac{1}{2})}} v_i \quad (\text{attention weights}) \quad (5.59)$$

$$\frac{\partial L}{\partial v_i} = \sum_{t=1}^T \alpha_{ti} \frac{\partial L}{\partial h_t^{(l+\frac{1}{2})}} \quad (\text{value gradients}) \quad (5.60)$$

Key parallelization features:

- Position-wise operations:
 - Query/Key/Value transformations
 - Gradient computations for $\mathbf{W}_q, \mathbf{W}_k, \mathbf{W}_v$
 - MLP layer gradients
- Matrix operations:
 - Attention score matrix $S = QK^\top$
 - Value weighted sum AV
 - Batch matrix multiplications

Cross-Layer Parallelization

The residual connections enable efficient layer-wise processing:

$$\frac{\partial L}{\partial h^{(l)}} = \frac{\partial L}{\partial h^{(l+1)}} + \frac{\partial L}{\partial \text{Layer}_l} \quad (\text{residual gradient}) \quad (5.61)$$

$$\frac{\partial L}{\partial \text{Layer}_l} = f'(\text{Layer}_l(h^{(l)})) \quad (\text{layer-specific gradient}) \quad (5.62)$$

Parallelization advantages:

- Pipeline parallelism:
 - Different layers on different devices
 - Overlapped forward/backward passes

- Efficient hardware utilization
- Memory efficiency:
 - Gradient checkpointing options
 - Selective activation storage
 - Memory-compute trade-offs

Implementation benefits:

- GPU/TPU optimization:
 - Batched matrix operations
 - Hardware-specific kernels
 - Memory access patterns
- Training acceleration:
 - Multi-GPU data parallelism
 - Distributed training
 - Gradient synchronization

This parallelizable structure contrasts with sequential RNNs, enabling Transformers to efficiently scale to long sequences and large models across multiple accelerators.

5.4 Associative Memory

5.4.1 SVD as Memory Structure

Linear transformation as associative memory:

$$y = \mathbf{W}x \quad (\text{basic transformation}) \quad (5.63)$$

$$\mathbf{W} = \sum_{i=1}^r \lambda_i b_i a_i^\top \quad (\text{SVD decomposition}) \quad (5.64)$$

$$\{a_i\} : \quad a_i^\top a_j = \delta_{ij} \quad (\text{orthonormal questions}) \quad (5.65)$$

$$\{b_i\} : \quad b_i^\top b_j = \delta_{ij} \quad (\text{orthonormal answers}) \quad (5.66)$$

Basic association pairs:

$$\mathbf{W}a_k = \lambda_k b_k \quad (\text{scaled association}) \quad (5.67)$$

$$\mathbf{W} \frac{a_k}{\lambda_k} = b_k \quad (\text{direct association}) \quad (5.68)$$

Interpolative property:

$$x = \sum_i c_i a_i \quad \text{(question combination)} \quad (5.69)$$

$$y = \mathbf{W}x = \sum_i c_i b_i \quad \text{(answer combination)} \quad (5.70)$$

This shows perfect interpolation:

- Input combines question patterns
- Output combines corresponding answers
- Linear coefficients preserved
- Smooth transition between memories

Example interpolation:

$$x = 0.7a_1 + 0.3a_2 \quad \text{(mixed question)} \quad (5.71)$$

$$y = 0.7b_1 + 0.3b_2 \quad \text{(mixed answer)} \quad (5.72)$$

$$\text{Smoothly combines two stored associations} \quad (5.73)$$

Key properties:

- Perfect recall: $\mathbf{W}a_k/\lambda_k = b_k$
- Linear mixing: coefficients preserved
- Continuous interpolation between memories
- Natural generalization to new inputs

5.4.2 MLP as Query Generator

One-hidden-layer MLP structure:

$$h = \sigma(\mathbf{W}_1 x + b_1) \quad \text{(query generation)} \quad (5.74)$$

$$y = \mathbf{W}_2 h + b_2 \quad \text{(answer retrieval)} \quad (5.75)$$

Interpretation:

- \mathbf{W}_1 : maps input to query space
- $\sigma(\cdot)$: selects relevant query components
- \mathbf{W}_2 : retrieves answers from queries

Query components in hidden layer:

$$h_i = \sigma(w_{1i}^\top x + b_{1i}) \quad (\text{query strength}) \quad (5.76)$$

$$y = \sum_i h_i w_{2i} \quad (\text{weighted answers}) \quad (5.77)$$

where:

- w_{1i} : question patterns to look for
- h_i : activation/relevance of each pattern
- w_{2i} : associated answers to each pattern

5.4.3 Memory Cleaning

SVD truncation for output weights:

$$\mathbf{W}_2 = \sum_{i=1}^r \lambda_i b_i a_i^\top \quad (\text{full SVD}) \quad (5.78)$$

$$\mathbf{W}_2^{\text{clean}} = \sum_{i: \lambda_i > \epsilon} \lambda_i b_i a_i^\top \quad (\text{cleaned}) \quad (5.79)$$

Benefits:

- Removes weak associations
- Keeps strong query-answer pairs
- Better generalization
- More reliable retrieval

5.4.4 Memory Editing

Given SVD decomposition:

$$\mathbf{W} = \sum_{i=1}^r \lambda_i b_i a_i^\top \quad (\text{original memory}) \quad (5.80)$$

Direct feature editing:

$$b_k \rightarrow b_k + \Delta b \quad (\text{modify answer}) \quad (5.81)$$

$$\mathbf{W}_{\text{edit}} = \sum_{i \neq k} \lambda_i b_i a_i^\top + \lambda_k (b_k + \Delta b) a_k^\top \quad (\text{edited memory}) \quad (5.82)$$

Localized editing:

$$\Delta \mathbf{W} = \lambda_k \Delta b \cdot a_k^\top \quad (\text{rank-1 update}) \quad (5.83)$$

$$\mathbf{W}_{\text{edit}} = \mathbf{W} + \Delta \mathbf{W} \quad (\text{modified memory}) \quad (5.84)$$

Properties:

- Preserves other associations
- Minimal interference
- Linear update rule
- Controllable modification

5.4.5 Low-Rank Adaptation (LoRA)

Original model structure:

$$h = \sigma(\mathbf{W}_1 x + b_1) \quad (\text{pre-trained query}) \quad (5.85)$$

$$y = \mathbf{W}_2 h + b_2 \quad (\text{pre-trained retrieval}) \quad (5.86)$$

LoRA adds low-rank update:

$$\mathbf{W}_{\text{new}} = \mathbf{W}_{\text{pre}} + \mathbf{B}\mathbf{A} \quad (\text{update}) \quad (5.87)$$

$$\text{rank}(\mathbf{B}\mathbf{A}) = r \ll \min(d_{\text{out}}, d_{\text{in}}) \quad (\text{low rank}) \quad (5.88)$$

Parameter efficiency:

- Original: $d_{\text{out}} \times d_{\text{in}}$ parameters
- LoRA: $r(d_{\text{out}} + d_{\text{in}})$ parameters
- Typical $r = 8$ or 16

Different learning rates for \mathbf{B} and \mathbf{A} can be employed according to recent work of Bin Yu's group.

Interpretation as new memories:

$$\mathbf{B}\mathbf{A} = \sum_{i=1}^r \tilde{\lambda}_i \tilde{b}_i \tilde{a}_i^\top \quad (\text{new associations}) \quad (5.89)$$

$$y = \mathbf{W}_{\text{pre}} h + \sum_{i=1}^r \tilde{\lambda}_i (\tilde{a}_i^\top h) \tilde{b}_i \quad (\text{combined retrieval}) \quad (5.90)$$

Key properties:

- Adds new query-answer pairs
- Preserves pre-trained knowledge
- Efficient fine-tuning
- Linear interpolation still holds

Benefits for adaptation:

- Memory efficient:
 - Small parameter count
 - Share base model
 - Multiple tasks possible
- Training efficient:
 - Few parameters to optimize
 - Better conditioning
 - Faster convergence
- Knowledge preservation:
 - Base model unchanged
 - Add task-specific patterns
 - Clean separation of knowledge

5.4.6 Query-Key-Value Projection as Associative Memory

The projection from hidden state to attention components:

$$q = \mathbf{W}_q h \quad (\text{query projection}) \quad (5.91)$$

$$k = \mathbf{W}_k h \quad (\text{key projection}) \quad (5.92)$$

$$v = \mathbf{W}_v h \quad (\text{value projection}) \quad (5.93)$$

Each projection is an associative memory.

Retrieval process:

1. Initialization Stage

- $h \rightarrow q$: retrieves question patterns
- $h \rightarrow k$: retrieves matching patterns
- $h \rightarrow v$: retrieves answer patterns

2. Context Retrieval Stage

- q, k : compute relevance scores
- v : combine relevant answers

We need associative memory to initialize the context retrieval process.

5.5 Reflection: A Matrix = A Thousand Rules

5.5.1 Matrix as Infinite Association Rules

A learned matrix transformation $\mathbf{W} \in \mathbb{R}^{d \times d}$ operating on vectors $x \in \mathbb{R}^d$ can implement what would require thousands of explicit rules in traditional symbolic systems. This power stems from several fundamental properties:

- Continuous transformation space:

$$y = \mathbf{W}x \quad (\text{basic transformation}) \quad (5.94)$$

$$= \mathbf{W}(\sum_i c_i e_i) \quad (\text{basis decomposition}) \quad (5.95)$$

$$= \sum_i c_i (\mathbf{W}e_i) \quad (\text{linearity}) \quad (5.96)$$

$$= \sum_i c_i r_i \quad (\text{transformed basis}) \quad (5.97)$$

where $\{e_i\}$ are basis vectors and $\{r_i\}$ are their transformations

- Superposition handling:

$$x = g_1 + g_2 + g_3 + \dots \quad (\text{input aspects}) \quad (5.98)$$

$$\mathbf{W}x = \mathbf{W}g_1 + \mathbf{W}g_2 + \mathbf{W}g_3 + \dots \quad (\text{transformed aspects}) \quad (5.99)$$

where each g_i represents a different semantic component

5.5.2 Advantages over Discrete Systems

Traditional rule-based systems and knowledge graphs face inherent limitations:

- Discrete rules: “if A then B”
- Explicit relationships: “A is-a B”
- Manual specification required
- No natural interpolation

In contrast, matrix transformations offer:

- Continuous interpolation:

$$x_\alpha = (1 - \alpha)x_1 + \alpha x_2 \quad (\text{input interpolation}) \quad (5.100)$$

$$\mathbf{W}x_\alpha = (1 - \alpha)\mathbf{W}x_1 + \alpha\mathbf{W}x_2 \quad (\text{smooth transition}) \quad (5.101)$$

- Implicit feature interactions:

$$[\mathbf{W}x]_i = \sum_j W_{ij}x_j \quad (\text{component view}) \quad (5.102)$$

$$= \sum_j w_{ij}^\top x \quad (\text{pattern matching}) \quad (5.103)$$

where each w_{ij} represents a learned pattern

5.5.3 Learnability through Backpropagation

A crucial advantage is the ability to learn optimal transformations through gradient descent:

$$L = \text{Loss}(\mathbf{W}x, y^*) \quad (\text{task objective}) \quad (5.104)$$

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial (\mathbf{W}x)} x^\top \quad (\text{weight gradient}) \quad (5.105)$$

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \eta \frac{\partial L}{\partial \mathbf{W}} \quad (\text{update rule}) \quad (5.106)$$

This enables several powerful capabilities:

- Automatic rule discovery:
 - No manual specification needed
 - Rules emerge from data
 - Complex patterns discovered naturally
- Continuous refinement:
 - Each update slightly improves transformation
 - Smooth optimization landscape
 - Natural handling of uncertainty
- Parallel updates:

$$\Delta W_{ij} = -\eta \frac{\partial L}{\partial W_{ij}} \quad (\text{element update}) \quad (5.107)$$

$$= -\eta \left[\frac{\partial L}{\partial (\mathbf{W}x)} x^\top \right]_{ij} \quad (\text{explicit form}) \quad (5.108)$$

All matrix elements update simultaneously

5.5.4 Compositional Learning

Multiple matrix transformations can compose to learn hierarchical patterns:

$$h^{(1)} = \sigma(\mathbf{W}_1 x) \quad (\text{first layer}) \quad (5.109)$$

$$h^{(2)} = \sigma(\mathbf{W}_2 h^{(1)}) \quad (\text{second layer}) \quad (5.110)$$

$$y = \mathbf{W}_3 h^{(2)} \quad (\text{output layer}) \quad (5.111)$$

Backpropagation enables end-to-end learning:

$$\frac{\partial L}{\partial \mathbf{W}_1} = \frac{\partial L}{\partial h^{(1)}} \frac{\partial h^{(1)}}{\partial \mathbf{W}_1} \quad (\text{chain rule}) \quad (5.112)$$

$$= (\mathbf{W}_2^\top \frac{\partial L}{\partial h^{(2)}}) \odot \sigma'(\mathbf{W}_1 x) x^\top \quad (\text{explicit form}) \quad (5.113)$$

This allows:

- Hierarchical feature learning
- Automatic feature composition
- Complex pattern discovery

5.5.5 Implications

This understanding suggests several key insights:

- Interpretability efforts should focus on understanding transformation spaces rather than extracting discrete rules
- Architecture design should leverage continuous transformations rather than mimicking symbolic reasoning
- The power of neural networks lies in their ability to learn rich continuous mappings that transcend symbolic representations

The combination of expressive power and learnability through backpropagation explains why matrix-based neural architectures have largely superseded traditional rule-based approaches in many domains. One matrix can implement thousands of implicit rules while being automatically tuned through gradient descent, offering a fundamentally more powerful paradigm for artificial intelligence.

5.5.6 Counter Argument: The Power of Abstract Logic

While continuous matrices excel at learning from concrete instances, abstract logical reasoning offers distinct advantages through universal quantification and symbolic manipulation:

- Universal rules:

$$\forall x : \text{Person}(x) \wedge \text{Mortal}(x) \rightarrow \text{WillDie}(x) \quad (\text{covers all cases}) \quad (5.114)$$

$$\forall x, y, z : \text{Greater}(x, y) \wedge \text{Greater}(y, z) \rightarrow \text{Greater}(x, z) \quad (\text{transitivity}) \quad (5.115)$$

- Perfect generalization:

- No training examples needed
- Zero-shot transfer to new domains
- No distribution shift concerns

- Compositional reasoning:

$$P \rightarrow Q, Q \rightarrow R \quad \therefore P \rightarrow R \quad (\text{valid inference}) \quad (5.116)$$

$$\forall x : [\exists y : \text{Parent}(y, x)] \rightarrow \text{HasParent}(x) \quad (\text{nested quantifiers}) \quad (5.117)$$

Key strengths of symbolic systems include:

- Verifiability: Proofs can be mechanically checked
- Transparency: Clear reasoning chains
- Meta-reasoning: Can reason about reasoning itself

This suggests complementary roles:

- Neural networks: Pattern learning from concrete instances
- Logical systems: Universal rules and abstract reasoning

The future may lie in hybrid systems that combine the pattern-recognition capabilities of matrices with the abstract reasoning power of logic. Each approach offers distinct advantages: matrices handle continuous patterns and learning from examples, while logic enables universal quantification and formal verification.

5.6 Architectural Comparison

5.6.1 Bottom-up Architecture

GPT/Temporal Convolution approach:

$$\text{Temporal Conv :} \quad h_t = \sigma\left(\sum_{\Delta t=0}^t \mathbf{W}_{\Delta t} h_{t-\Delta t}\right) \quad (\text{all past tokens}) \quad (5.118)$$

$$\text{GPT :} \quad h_t = h_t + \sum_{\tau=1}^t \alpha_{t\tau} \mathbf{W}_v h_\tau \quad (\text{attention to past}) \quad (5.119)$$

Key properties:

- Direct access to all previous tokens
- Parallel processing possible
- No information compression
- Like having all past notes visible

5.6.2 Context Access

RNN (working memory):

$$h_t = f(h_{t-1}, x_t) \quad (\text{state update}) \quad (5.120)$$

$$= \tanh(\mathbf{W}_{\text{recurrent}} h_{t-1} + \mathbf{W}_{\text{embed}} x_t) \quad (\text{compress into state}) \quad (5.121)$$

GPT with KV cache (written notes):

$$\text{Cache}_K = [k_1, \dots, k_t] \quad (\text{stored keys}) \quad (5.122)$$

$$\text{Cache}_V = [v_1, \dots, v_t] \quad (\text{stored values}) \quad (5.123)$$

$$h_t = \text{attention}(q_t, \text{Cache}_K, \text{Cache}_V) \quad (\text{look up notes}) \quad (5.124)$$

5.6.3 Memory Metaphor

RNN memory is like working memory:

- Must hold everything in state vector
- Information gets compressed
- Limited capacity
- Easy to forget
- Quick access but fragile

GPT memory is like pencil and paper:

- Writes down each token's representation
- No compression needed
- Unlimited capacity (up to context length)
- Perfect recall
- Can look back at any point

5.6.4 Inference Process

RNN generation:

- Maintain single state h_t
- Update sequentially
- Must remember context in state
- Like keeping everything in mind

GPT generation with KV cache:

- Store all keys and values
- Add to cache at each step
- Look up relevant context
- Like referring to written notes

5.6.5 Trade-offs

RNN advantages:

- Fixed memory usage
- Fast inference (single state)
- Unbounded context length
- Natural sequential processing

GPT/KV cache advantages:

- No information loss
- Selective attention to past
- Parallel training
- Reliable long-term recall

This architectural difference explains why:

- GPT scales better with size
- Handles long-range dependencies better
- More computationally intensive
- Memory usage grows with sequence length

5.7 Original Transformer for Translation

Example translation task:

English : "I love machine learning"

Spanish : "Amo el aprendizaje automático"

5.7.1 Architecture Overview

1. Encoder processes source sentence:

$$e_t^{(1)} = \mathbf{W}_{\text{embed}}^{\text{enc}} x_t \quad (\text{English embedding}) \quad (5.125)$$

$$e_t^{(l+1)} = e_t^{(l)} + \text{BiAttn}(e_t^{(l)}, e_{1:n}^{(l)}) + \text{MLP}(e_t^{(l)}) \quad (\text{encode context}) \quad (5.126)$$

2. Decoder generates translation:

$$h_t^{(1)} = \mathbf{W}_{\text{embed}}^{\text{dec}} y_t \quad (\text{Spanish embedding}) \quad (5.127)$$

$$h_t^{(l+\frac{1}{2})} = h_t^{(l)} + \text{CausalAttn}(h_t^{(l)}, h_{1:t}^{(l)}) \quad (\text{self-attention}) \quad (5.128)$$

$$h_t^{(l+\frac{3}{4})} = h_t^{(l+\frac{1}{2})} + \text{CrossAttn}(h_t^{(l+\frac{1}{2})}, e_{1:n}^{(L)}) \quad (\text{source attention}) \quad (5.129)$$

$$h_t^{(l+1)} = h_t^{(l+\frac{3}{4})} + \text{MLP}(h_t^{(l+\frac{3}{4})}) \quad (\text{process}) \quad (5.130)$$

5.7.2 Three Types of Attention

1. Bidirectional (Encoder):

- Full context: "I love machine learning"
- Each word sees all others
- Builds rich source representations

2. Causal (Decoder self-attention):

- Sequential generation: "Amo..."
- Only sees previous words
- Like GPT's attention

3. Cross (Decoder source-attention):

- Attends to encoded source
- Access to full English sentence
- Guides translation decisions

5.7.3 Encoder Matrix Implementation

For sequence length n :

$$H \in \mathbb{R}^{n \times d} \quad (\text{all tokens}) \quad (5.131)$$

$$Q = HW_Q, K = HW_K, V = HW_V \in \mathbb{R}^{n \times d} \quad (\text{projections}) \quad (5.132)$$

$$A = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V \quad (\text{attention}) \quad (5.133)$$

Advantages:

- Single matrix multiplication
- All positions processed in parallel
- GPU/TPU efficient
- $O(n^2d)$ operations done in parallel

5.7.4 Decoder Masked Attention

Causal masking:

$$M_{ij} = \begin{cases} 0 & \text{if } i \geq j \\ -\infty & \text{if } i < j \end{cases} \quad (\text{mask matrix}) \quad (5.134)$$

$$A = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}} + M\right)V \quad (\text{masked attention}) \quad (5.135)$$

Example for "Amo el":

$$M = \begin{bmatrix} 0 & -\infty & -\infty \\ 0 & 0 & -\infty \\ 0 & 0 & 0 \end{bmatrix} \quad (5.136)$$

This ensures:

- Each position sees only past
- Still parallel computation
- Autoregressive property
- Training matches inference

5.8 Transformer Family: Translation to BERT and GPT

5.8.1 Architectural Heritage

Original Transformer for translation had:

- Encoder: bidirectional attention on source
- Decoder: causal attention + cross-attention to source
- Both parts specialized for translation task

5.8.2 BERT (Bidirectional Encoder Representations from Transformers)

Takes encoder architecture:

- Uses only bidirectional encoder layers
- Removes decoder and cross-attention
- Adapts for general language understanding

Masked Language Modeling:

- Randomly mask 15% of input tokens
- Replace with:

MASK token (80%)

- Random word (10%)
- Original word (10%)

- Model predicts original tokens
- Forces bidirectional understanding

[CLS] Token - The Team Captain:

- Special first token in every input
- Not tied to any input word
- Free to gather relevant information
- Like a captain summarizing team's knowledge
- Used for sentence-level tasks
- Learns to aggregate sequence information

5.8.3 GPT (Generative Pre-trained Transformer)

Takes decoder architecture:

- Uses only causal decoder layers
- Removes encoder and cross-attention
- Focuses on text generation

Key differences:

- No masking tokens needed
- No special [CLS] token
- Pure autoregressive prediction
- Simpler but unidirectional

5.8.4 Core Distinctions

BERT:

- Bidirectional context
- Masked prediction task
- Good for understanding
- Special tokens for tasks
- Team-based information gathering

GPT:

- Unidirectional (left-to-right)
- Next token prediction
- Natural for generation
- No special tokens
- Sequential information processing

This split created two branches:

- Understanding models (BERT-like)
- Generation models (GPT-like)
- Each optimized for their task
- Different pre-training objectives
- Different downstream applications

5.9 Original Transformer Parameters

5.9.1 Core Architecture Parameters

Base model:

$$d_{\text{model}} = 512 \quad (\text{embedding dimension}) \quad (5.137)$$

$$d_{\text{ff}} = 2048 \quad (\text{feed-forward width}) \quad (5.138)$$

$$h = 8 \quad (\text{attention heads}) \quad (5.139)$$

$$d_k = d_v = d_{\text{model}}/h = 64 \quad (\text{per-head dimension}) \quad (5.140)$$

$$N_{\text{enc}} = N_{\text{dec}} = 6 \quad (\text{number of layers}) \quad (5.141)$$

Big model:

$$d_{\text{model}} = 1024 \quad (5.142)$$

$$d_{\text{ff}} = 4096 \quad (5.143)$$

$$h = 16 \quad (5.144)$$

$$N_{\text{enc}} = N_{\text{dec}} = 6 \quad (5.145)$$

5.9.2 Key Design Choices

Dimension ratios:

- Feed-forward: $d_{\text{ff}} = 4d_{\text{model}}$
- Per-head: $d_k = d_v = d_{\text{model}}/h$
- Equal encoder-decoder depth

Training details:

- Dropout: $p = 0.1$
- Label smoothing: $\epsilon = 0.1$
- Warmup steps: 4000
- Adam optimizer: $\beta_1 = 0.9, \beta_2 = 0.98$

5.10 GPT-3 175B Architecture

5.10.1 Key Parameters

Model dimensions:

$$N = 175 \text{ billion parameters} \quad (5.146)$$

$$d = 12,288 \quad (\text{embedding dimension}) \quad (5.147)$$

$$L = 96 \quad (\text{number of layers}) \quad (5.148)$$

$$H = 96 \quad (\text{attention heads}) \quad (5.149)$$

$$d_h = d/H = 128 \quad (\text{dimension per head}) \quad (5.150)$$

5.10.2 Parameter Distribution

Major parameter blocks per layer:

- Attention:

$$\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V \in \mathbb{R}^{d \times d} \quad (3 \text{ matrices}) \quad (5.151)$$

$$\mathbf{W}_O \in \mathbb{R}^{d \times d} \quad (\text{output projection}) \quad (5.152)$$

- MLP:

$$\mathbf{W}_1 \in \mathbb{R}^{4d \times d} \quad (\text{up-projection}) \quad (5.153)$$

$$\mathbf{W}_2 \in \mathbb{R}^{d \times 4d} \quad (\text{down-projection}) \quad (5.154)$$

Total parameters per layer:

$$P_{\text{attention}} = 4d^2 \quad (\text{QKV} + \text{O}) \quad (5.155)$$

$$P_{\text{mlp}} = 8d^2 \quad (\text{up} + \text{down}) \quad (5.156)$$

$$P_{\text{layer}} = 12d^2 \quad (\text{total per layer}) \quad (5.157)$$

5.10.3 Computation Flow

Per sequence position:

- Memory: $O(Ld)$ activations
- Compute: $O(Ld^2)$ operations
- Attention: $O(LH(d/H)^2)$ per key/query

Context window:

- Maximum length: 2048 tokens
- Attention memory: $O(Ld + T^2)$
- Attention compute: $O(LdT + T^2d)$

5.10.4 Design Choices

Key ratios:

- $d_h = 128$ is fixed across scales
- MLP width = $4 \times$ embedding
- H scales with d keeping d_h constant
- L chosen for compute efficiency

Training considerations:

- 8-way tensor parallelism
- Mixed precision training
- Gradient checkpointing
- Careful initialization scale

5.11 Scaling Laws

5.11.1 Power Law Relationships

Language model performance follows predictable power laws with respect to three key variables:

Model Size

Performance improves with parameter count:

$$L = \left(\frac{N_c}{N} \right)^{\alpha_N} + c_1 \quad (5.158)$$

where:

- L is the loss (prediction error)
- N is number of parameters
- $\alpha_N \approx 0.076$ (empirical coefficient)
- N_c is critical model scale

Dataset Size

Performance improves with training data:

$$L = \left(\frac{D_c}{D} \right)^{\alpha_D} + c_2 \quad (5.159)$$

where:

- D is dataset size (tokens)
- $\alpha_D \approx 0.095$ (empirical coefficient)
- D_c is critical data scale

Compute Budget

Performance improves with training compute:

$$L = \left(\frac{C_c}{C} \right)^{\alpha_C} + c_3 \quad (5.160)$$

where:

- C is compute (FLOPs)
- $\alpha_C \approx 0.050$ (empirical coefficient)
- C_c is critical compute scale

5.11.2 Optimal Allocation

Given compute budget C , optimal model size and data size follow:

$$N_{\text{opt}} \propto C^{0.6} \quad (5.161)$$

$$D_{\text{opt}} \propto C^{0.4} \quad (5.162)$$

This means:

- 60% of increased compute should go to model size
- 40% should go to dataset size
- Batch size scales more slowly

5.11.3 Chinchilla Scaling

Recent findings suggest:

- Most models are over-parameterized
- Under-trained on data
- Optimal tokens/parameter ≈ 20
- More compute should go to training steps

For compute budget C :

$$N_{\text{chin}} \propto C^{0.5} \tag{5.163}$$

$$D_{\text{chin}} \propto C^{0.5} \tag{5.164}$$

5.11.4 Implications

These laws guide model development:

- Predictable returns:
 - Can estimate performance gains
 - Plan resource requirements
 - Set realistic targets
- Resource allocation:
 - Balance model vs data size
 - Optimize training time
 - Choose batch size
- Architecture design:
 - Choose model depth vs width
 - Set attention heads
 - Balance MLP sizes
- Training strategy:
 - Data sampling rates
 - Learning rate schedules
 - Optimization choices

5.11.5 Example Scales

Typical scales for different model sizes:

$$\text{Base :} \quad N \approx 125\text{M}, \quad D \approx 2.5\text{B tokens} \quad (5.165)$$

$$\text{Large :} \quad N \approx 350\text{M}, \quad D \approx 7\text{B tokens} \quad (5.166)$$

$$\text{XL :} \quad N \approx 1.3\text{B}, \quad D \approx 26\text{B tokens} \quad (5.167)$$

$$\text{XXL :} \quad N \approx 175\text{B}, \quad D \approx 3.5\text{T tokens} \quad (5.168)$$

5.12 Two-Stage Training

5.12.1 Pre-training Stage

Self-supervised learning on large text corpora:

$$h_t^{(1)} = \mathbf{W}_{\text{embed}} x_t \quad (\text{embed token}) \quad (5.169)$$

$$h_t^{(l+1)} = h_t^{(l)} + \text{Layer}_l(h_t^{(l)}, h_{1:t}^{(l)}) \quad (\text{process context}) \quad (5.170)$$

$$p_t = \text{softmax}(\mathbf{W}_{\text{embed}}^\top h_t^{(L)}) \quad (\text{next token prediction}) \quad (5.171)$$

Training objective:

$$J_{\text{pretrain}} = \sum_t \log p(x_t | x_{<t}) \quad (5.172)$$

Key aspects:

- Large-scale training data
- No human labels needed
- Learns general patterns
- Shared embedding matrix
- World knowledge in weights

5.12.2 Instruction Fine-tuning

Format training data as instruction-response pairs:

Input format:

Instruction: <task description>

Input: <specific instance>

Output: <desired response>

Example:

Instruction: Summarize the following text.

Input: Recent studies show that regular exercise...

Output: The research indicates that physical activity...

Training objective:

$$J_{\text{instruction}} = \sum_t \log p(y_t | x_{\text{instruction}}, x_{\text{input}}, y_{<t}) \quad (5.173)$$

Key aspects:

- Much smaller dataset
- Human-curated examples
- Task-specific format
- Maintains general knowledge
- Aligns capabilities

5.12.3 Training Process

1. Pre-training phase:

- Start with random weights
- Train on internet-scale data
- Learn language patterns
- Acquire world knowledge
- General text prediction

2. Instruction tuning phase:

- Start with pre-trained model
- Train on instruction data
- Learn task formats
- Align with instructions
- Follow user requests

5.12.4 Benefits of Two-Stage Approach

1. Data efficiency:

- Pre-training: billions of tokens
- Instruction tuning: millions of examples
- Transfer of knowledge
- Few-shot learning ability

2. Task flexibility:

- General knowledge base
- Task-specific formatting
- Natural language interface
- Zero-shot generalization

5.12.5 Example Instructions

Different instruction types:

1. Direct command:

"Translate the following English text to French:"

2. Task description:

"You are a helpful assistant that writes emails."

3. Role-based:

"Act as an expert physicist explaining..."

4. Multi-step:

"First analyze the sentiment, then explain why..."

5. Format specification:

"Provide the answer in bullet points..."

This approach enables:

- Natural task specification
- Flexible interaction
- Clear user control
- Task composition

5.13 Data Curation Pipeline

5.13.1 Pre-training Stages

1. Basic Internet Data:

- Web crawls (Common Crawl)
- Basic filtering:
 - Remove spam/bot content
 - Filter duplicate content
 - Language identification
 - Text quality heuristics

2. Higher Quality Sources:

- Books and academic papers
- Wikipedia
- Verified news sources
- Technical documentation

- Quality code repositories

3. Mixed-Quality Training:

- Weight different sources
- More tokens from high-quality sources
- Balance between breadth and quality
- Preserve diverse writing styles

5.13.2 Instruction Fine-tuning

Carefully curated instruction data:

- Human-written examples
- Task diversity:
 - Writing and editing
 - Analysis and reasoning
 - Coding and math
 - Q&A formats
- Clear evaluation criteria
- Quality control standards

5.13.3 RLHF Data

1. Preference Data:

- Human comparisons of outputs
- Consistent ranking criteria
- Coverage of edge cases
- Diverse evaluators

2. Constitutional Rules:

- Safety guidelines
- Ethical principles
- Behavior boundaries
- Response formats

5.13.4 Quality Progression

Data quality requirements increase:

- Pre-training:
 - Billions of tokens
 - Automated filtering
 - Statistical quality metrics
- Instruction tuning:
 - Millions of examples
 - Human curation
 - Task-specific validation
- RLHF:
 - Thousands of comparisons
 - Expert review
 - Rigorous standards

5.13.5 Continuous Improvement

Iterative refinement through:

- User interaction data
- Error analysis
- Bias detection
- Performance monitoring
- Community feedback

5.14 Reinforcement Learning from Human Feedback

5.14.1 Basic Concept

Language models can be viewed as decision-makers:

- Input x : question/instruction (like state in RL)
- Output y : answer/completion (like action in RL)
- Goal: maximize human satisfaction with responses
- Challenge: satisfaction isn't directly programmable

5.14.2 Reward Modeling

Reward learning is sometimes called inverse reinforcement learning, i.e., based on the agent behavior or preference, learn about the agent's reward or value. In comparison, reinforcement learning refers to the optimization of policy given the reward.

Model Architecture

Reward score computation:

$$h = \text{Encoder}(x, y) \quad (\text{process input-output pair}) \quad (5.174)$$

$$r_\phi(x, y) = \mathbf{w}^\top h_{\text{final}} \quad (\text{scalar reward from final embedding}) \quad (5.175)$$

where:

- Same architecture as language model
- Single scalar output
- Learns to score completions

Bradley-Terry Model

Probability that response y_1 is preferred over y_2 given prompt x :

$$P(y_1 \succ y_2 | x; \phi) = \frac{\exp(r_\phi(x, y_1))}{\exp(r_\phi(x, y_1)) + \exp(r_\phi(x, y_2))} \quad (5.176)$$

$$= \sigma(r_\phi(x, y_1) - r_\phi(x, y_2)) \quad (\text{logistic form}) \quad (5.177)$$

Training the reward model:

$$\mathcal{J}_{\text{RM}}(\phi) = \sum_{(x, y_1, y_2) \in \mathcal{D}} \log P(y_1 \succ y_2 | x; \phi) \quad (5.178)$$

$$= \sum_{(x, y_1, y_2) \in \mathcal{D}} \log \sigma(r_\phi(x, y_1) - r_\phi(x, y_2)) \quad (5.179)$$

where:

- $r_\phi(x, y)$: reward model with parameters ϕ
- \mathcal{D} : human comparison data
- (x, y_1, y_2) : prompt and response pairs where y_1 preferred over y_2
- Parameters ϕ learned to match human preferences

Properties

Bradley-Terry model advantages:

- Well-studied for pairwise comparisons
- Natural probabilistic interpretation
- Transitive preferences
- Smooth gradients for learning

Training considerations:

- Consistent human labeling crucial
- Coverage of response space
- Balance between different types of comparisons
- Quality control in human feedback

5.14.3 Bradley-Terry Model

Sports Origins

Originally developed for ranking sports teams:

- Each team has strength score s_i
- Probability team i beats team j :

$$P(i \text{ beats } j) = \frac{\exp(s_i)}{\exp(s_i) + \exp(s_j)} \quad (5.180)$$

- Higher score difference \rightarrow higher win probability
- Learns team strengths from match outcomes

Example interpretation:

- If $s_i = s_j$: $P(\text{win}) = 0.5$
- s_i two points higher: strong favorite
- Can rank all teams on same scale
- Similar to Elo rating system

Adaptation to Response Ranking

Same principle for language model outputs:

$$s_i \rightarrow r_\phi(x, y_i) \quad (\text{response score}) \quad (5.181)$$

$$P(y_1 \succ y_2 | x; \phi) = \frac{\exp(r_\phi(x, y_1))}{\exp(r_\phi(x, y_1)) + \exp(r_\phi(x, y_2))} \quad (5.182)$$

$$= \sigma(r_\phi(x, y_1) - r_\phi(x, y_2)) \quad (5.183)$$

where $r_\phi(x, y)$ plays the role of team strength, scoring how good response y is for prompt x . Analogies:

- Team strength \rightarrow response quality
- Match outcome \rightarrow human preference
- League ranking \rightarrow global quality scale
- Season records \rightarrow preference dataset

Benefits of this approach:

- Well-studied mathematical properties
- Natural handling of transitivity
- Interpretable scores
- Efficient learning from comparisons

5.14.4 Policy Gradient Fine-tuning

Objective and Gradient

The original objective is to maximize expected reward:

$$J(\theta) = \mathbb{E}_{x,y \sim \pi_\theta} [r_\phi(x, y)] \quad (5.184)$$

$$= \sum_x p(x) \sum_y \pi_\theta(y|x) r_\phi(x, y) \quad (5.185)$$

To derive the policy gradient, we take the derivative with respect to θ :

$$\nabla_\theta J(\theta) = \nabla_\theta \sum_x p(x) \sum_y \pi_\theta(y|x) r_\phi(x, y) \quad (5.186)$$

$$= \sum_x p(x) \sum_y \nabla_\theta \pi_\theta(y|x) r_\phi(x, y) \quad (5.187)$$

Using the likelihood ratio trick ($\nabla_\theta \pi_\theta = \pi_\theta \nabla_\theta \log \pi_\theta$):

$$\nabla_\theta J(\theta) = \sum_x p(x) \sum_y \pi_\theta(y|x) \nabla_\theta \log \pi_\theta(y|x) r_\phi(x, y) \quad (5.188)$$

$$= \mathbb{E}_{x,y \sim \pi_\theta} [r_\phi(x, y) \nabla_\theta \log \pi_\theta(y|x)] \quad (5.189)$$

Baseline

The original objective is to maximize expected reward:

$$J(\theta) = \mathbb{E}_{x,y \sim \pi_\theta} [r_\phi(x, y)] \quad (5.190)$$

$$= \sum_x p(x) \sum_y \pi_\theta(y|x) r_\phi(x, y) \quad (5.191)$$

Consider a modified reward with baseline:

$$\tilde{r}_\phi(x, y) = r_\phi(x, y) - b_\phi(x) \quad (5.192)$$

$$\tilde{J}(\theta) = \mathbb{E}_{x,y \sim \pi_\theta} [\tilde{r}_\phi(x, y)] = J(\theta) - \mathbb{E}_{p(x)} [b_\phi(x)] \quad (5.193)$$

Thus $J'(\theta) = \tilde{J}'(\theta)$.

This key result shows we can subtract any state-dependent baseline without changing the expected gradient.

Value Function and Advantage

The value function represents expected reward at state x :

$$V_\phi(x) = \mathbb{E}_{y \sim \pi} [r_\phi(x, y)] \quad (5.194)$$

For instance, for the question x , V measures the difficulty of x . For the task instruction x , V measures the difficulty of the task.

The advantage function measures relative improvement over expected value:

$$A_\phi(x, y) = r_\phi(x, y) - V_\phi(x) \quad (5.195)$$

Since value function is a valid baseline, we can express the policy gradient using advantages:

$$\nabla_\theta J = \mathbb{E}_{x, y \sim \pi_\theta} [r_\phi(x, y) \nabla_\theta \log \pi_\theta(y|x)] \quad (5.196)$$

$$= \mathbb{E}_{x, y \sim \pi_\theta} [A_\phi(x, y) \nabla_\theta \log \pi_\theta(y|x)] \quad (5.197)$$

Variance Analysis

Policy gradient's high variance comes from several sources:

- Reward variation across states
- Long-term credit assignment
- Stochastic policy sampling
- State-dependent reward distributions

Advantage function helps reduce variance because:

- Normalizes rewards relative to state-specific expectations
- Removes state-dependent reward variation:
 - Original: $r_\phi(x, y)$ varies with state
 - Advantage: $A_\phi(x, y)$ measures relative improvement
- Centers the learning signal:
 - $\mathbb{E}_{y \sim \pi_\theta} [A_\phi(x, y)] = 0$
 - Positive advantage: better than average
 - Negative advantage: worse than average

Empirical benefits:

- Smaller gradient variance
- More stable training
- Faster convergence
- Better final performance

Implementation

For each training batch:

1. Sample prompts x from dataset
2. Generate responses $y \sim \pi_\theta(\cdot|x)$
3. Compute rewards $r_\phi(x, y)$
4. Estimate value function $V_\phi(x)$
5. Compute advantages $A_\phi(x, y) = r_\phi(x, y) - V_\phi(x)$
6. Update policy:

$$\theta \leftarrow \theta + \alpha \cdot A_\phi(x, y) \nabla_\theta \log \pi_\theta(y|x) \quad (5.198)$$

Practical considerations:

- Need accurate value estimation
- Balance exploration/exploitation
- Prevent performance collapse
- Maintain language modeling capability

5.14.5 Comparison with Maximum Likelihood

Imitation Learning vs Policy Gradient

Maximum likelihood for imitation learning:

$$J_{\text{MLE}}(\theta) = \mathbb{E}_{(x,y) \sim \mathcal{D}}[\log \pi_\theta(y|x)] \quad (\text{teacher data}) \quad (5.199)$$

$$\nabla_\theta J_{\text{MLE}} = \mathbb{E}_{(x,y) \sim \mathcal{D}}[\nabla_\theta \log \pi_\theta(y|x)] \quad (5.200)$$

Policy gradient with reward:

$$J_{\text{PG}}(\theta) = \mathbb{E}_{x,y \sim \pi_\theta}[r_\phi(x, y)] \quad (\text{self-generated}) \quad (5.201)$$

$$\nabla_\theta J_{\text{PG}} = \mathbb{E}_{x,y \sim \pi_\theta}[r_\phi(x, y) \nabla_\theta \log \pi_\theta(y|x)] \quad (5.202)$$

Key differences:

- Data source:
 - MLE: y generated by a teacher
 - PG: y generated by current policy π_θ
- Learning signal:
 - MLE: Direction from teacher's actions
 - PG: Weighted by reward $r_\phi(x, y)$

Why Self-Imitation Fails

Consider self-imitation (policy gradient with $r_\phi(x, y) = 1$):

$$\nabla_\theta J_{\text{self}} = \mathbb{E}_{x, y \sim \pi_\theta} [\nabla_\theta \log \pi_\theta(y|x)] = \nabla_\theta \mathbb{E}_{x, y \sim \pi_\theta} [1] = 0 \quad (5.203)$$

Therefore:

- Self-imitation provides no learning signal
- Reward breaks symmetry and enables learning, e.g., in RLHF.

5.15 Proximal Policy Optimization (PPO)

5.15.1 Importance Sampling Form

Express objective using old policy π_{old} :

$$J(\theta) = \mathbb{E}_{x, y \sim \pi_\theta} [r_\phi(x, y)] \quad (5.204)$$

$$= \mathbb{E}_{x, y \sim \pi_{\text{old}}} \left[\frac{\pi_\theta(y|x)}{\pi_{\text{old}}(y|x)} r_\phi(x, y) \right] \quad (5.205)$$

Define probability ratio:

$$\rho_\theta(x, y) = \frac{\pi_\theta(y|x)}{\pi_{\text{old}}(y|x)} \quad (5.206)$$

With advantage:

$$J_{\text{IS}}(\theta) = \mathbb{E}_{x, y \sim \pi_{\text{old}}} [\rho_\theta(x, y) A_\phi(x, y)] \quad (5.207)$$

5.15.2 Motivation for Clipping

Challenges with pure importance sampling:

- Large ratios cause high variance
- Excessive policy changes
- Training instability
- Loss of learned behaviors

5.15.3 PPO Clipped Objective

Define clipped objective:

$$J_{\text{PPO}}(\theta) = \mathbb{E}_{x, y \sim \pi_{\text{old}}} [\min(\rho_\theta A_\phi, \text{clip}(\rho_\theta, 1 - \epsilon, 1 + \epsilon) A_\phi)] \quad (5.208)$$

Three cases based on advantage and ratio:

$$J_{\text{PPO}}(\theta) = \mathbb{E}_{x,y \sim \pi_{\text{old}}} \begin{cases} (1 + \epsilon)A_\phi & \text{if } A_\phi > 0 \text{ and } \rho_\theta > 1 + \epsilon \\ (1 - \epsilon)A_\phi & \text{if } A_\phi < 0 \text{ and } \rho_\theta < 1 - \epsilon \\ \rho_\theta A_\phi & \text{otherwise} \end{cases} \quad (5.209)$$

This means:

- Positive advantage ($A_\phi > 0$):
 - Increase π_θ but cap at $(1 + \epsilon)$
 - Prevents too large increases
 - Stabilizes good behaviors
- Negative advantage ($A_\phi < 0$):
 - Decrease π_θ but cap at $(1 - \epsilon)$
 - Prevents too large decreases
 - Maintains exploration
- Within bounds:
 - Use normal policy gradient
 - Allow natural updates
 - Trust region behavior

5.15.4 Implementation Benefits

This formulation provides:

- Stable learning
 - Bounded policy changes
 - Controlled exploration
 - Reliable convergence
- Simple implementation
 - No KL constraints
 - First-order optimization
 - Easy to tune
- Preservation of knowledge
 - Prevents catastrophic changes
 - Maintains learned behaviors
 - Smooth policy evolution

5.15.5 Understanding PPO

Trust Region in Data Space

Proximal Policy Optimization (PPO) implements a trust region in the data space through its clipping mechanism. Unlike TRPO which operates in model space (KL-divergence between policies) or simple gradient methods that work in parameter space, PPO directly controls the ratio of action probabilities $\rho(s, a) = \pi_{\text{new}}(a|s)/\pi_{\text{old}}(a|s)$.

This data-space approach offers several advantages:

1. Direct control over behavioral changes
2. More interpretable constraints
3. Simpler implementation than model-space constraints

Active Learning Perspective

PPO’s clipping mechanism can be viewed as a form of selective learning, where certain state-action pairs are actively removed from the learning process. When the probability ratio $\rho(s, a)$ exceeds the clipping threshold $(1 + \epsilon)$ for positive advantage, $(1 - \epsilon)$ for negative advantage, the gradient is stopped for that example. This is effectively a form of “negative active learning” where the algorithm decides which examples to stop learning from, rather than which examples to learn from.

Exploration-Exploitation Management

The clipping mechanism in PPO naturally manages the exploration-exploitation tradeoff:

For positive advantage $A(s, a) > 0$:

- When $\rho(s, a) > 1 + \epsilon$, gradient flow stops
- This prevents over-exploitation by limiting how strongly the policy can commit to advantageous actions
- Maintains exploration by preventing the policy from becoming too deterministic

For negative advantage $A(s, a) < 0$:

- When $\rho(s, a) < 1 - \epsilon$, gradient flow stops
- This prevents over-suppression of currently disadvantageous actions
- Preserves exploration by maintaining the possibility of revisiting these actions

Implementation Perspective

While PPO is often presented through its objective function:

$$L = \min(\rho(s, a)A(s, a), \text{clip}(\rho(s, a), 1 - \epsilon, 1 + \epsilon)A(s, a)) \quad (5.210)$$

A more natural implementation might use stop-gradient operations:

$$L = \begin{cases} \text{stop_gradient}(\rho(s, a))A(s, a) & \text{if } A(s, a) > 0 \text{ and } \rho(s, a) > 1 + \epsilon \\ \text{stop_gradient}(\rho(s, a))A(s, a) & \text{if } A(s, a) < 0 \text{ and } \rho(s, a) < 1 - \epsilon \\ \rho(s, a)A(s, a) & \text{otherwise} \end{cases}$$

This implementation more directly expresses the algorithm's intention to halt learning when policy changes exceed the trust region bounds.

5.16 Vision Transformer (ViT)

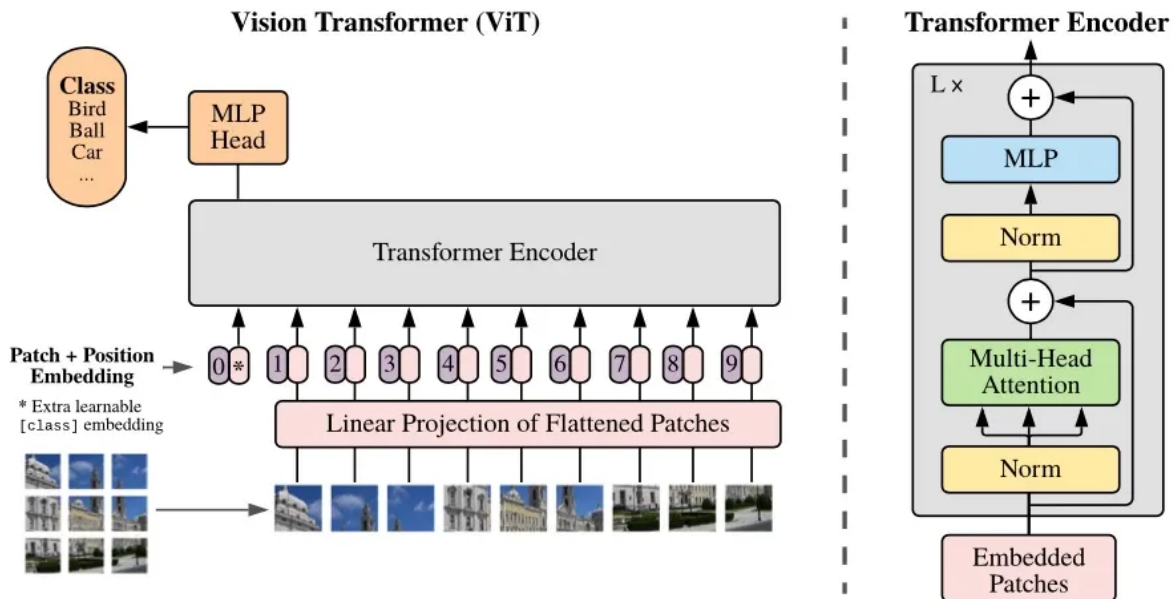


Figure 5.4: ViT

5.16.1 Image to Sequence

Convert 2D image to sequence of patches:

$$\text{Image} \in \mathbb{R}^{H \times W \times C} \quad (\text{original}) \quad (5.211)$$

$$\text{Patches} \in \mathbb{R}^{N \times (P^2 C)} \quad (\text{sequence}) \quad (5.212)$$

$$N = HW/P^2 \quad (\text{sequence length}) \quad (5.213)$$

where:

- $P \times P$: patch size (e.g., 16×16)
- N : number of patches
- Each patch flattened to vector

5.16.2 Patch Embedding

Linear projection of patches:

$$h_i^{(1)} = \mathbf{W}_{\text{patch}} x_i + \mathbf{W}_{\text{pos}} i \quad (\text{patch} + \text{position}) \quad (5.214)$$

$$h_0^{(1)} = \mathbf{W}_{\text{class}} \quad ([\text{CLS}] \text{ token}) \quad (5.215)$$

5.16.3 Processing Architecture

Standard transformer layers:

$$h_i^{(l+\frac{1}{2})} = h_i^{(l)} + \text{Attention}(h_i^{(l)}, h_{0:N}^{(l)}) \quad (\text{global attention}) \quad (5.216)$$

$$h_i^{(l+1)} = h_i^{(l+\frac{1}{2})} + \text{MLP}(h_i^{(l+\frac{1}{2})}) \quad (\text{feature processing}) \quad (5.217)$$

5.16.4 Comparison with CNN

Similarities:

- Hierarchical processing
 - CNN: through layers
 - ViT: through transformer depth
- Feature learning
 - CNN: convolutional filters
 - ViT: attention patterns
- Position handling
 - CNN: built into convolution

- ViT: position embeddings

Key differences:

- Receptive field:
 - CNN: local, grows gradually
 - ViT: global from first layer
- Parameter sharing:
 - CNN: weight tied across space
 - ViT: attention weights dynamic
- Spatial structure:
 - CNN: explicit in convolution
 - ViT: learned through position

5.16.5 Computational Aspects

Memory and compute:

- CNN:
 - $O(HWC)$ memory
 - Linear in image size
 - Efficient for small images
- ViT:
 - $O(N^2D)$ attention memory
 - Quadratic in number of patches
 - Better for larger patches

5.16.6 Practical Considerations

Training requirements:

- CNN:
 - Strong inductive bias
 - Works with less data
 - Stable training
- ViT:

- Needs more data
- Pre-training important
- More flexible patterns

Performance characteristics:

- CNN: better for small datasets
- ViT: scales better with data size
- Hybrid approaches possible
- Task-dependent trade-offs

5.17 CLIP: Contrastive Language–Image Pretraining

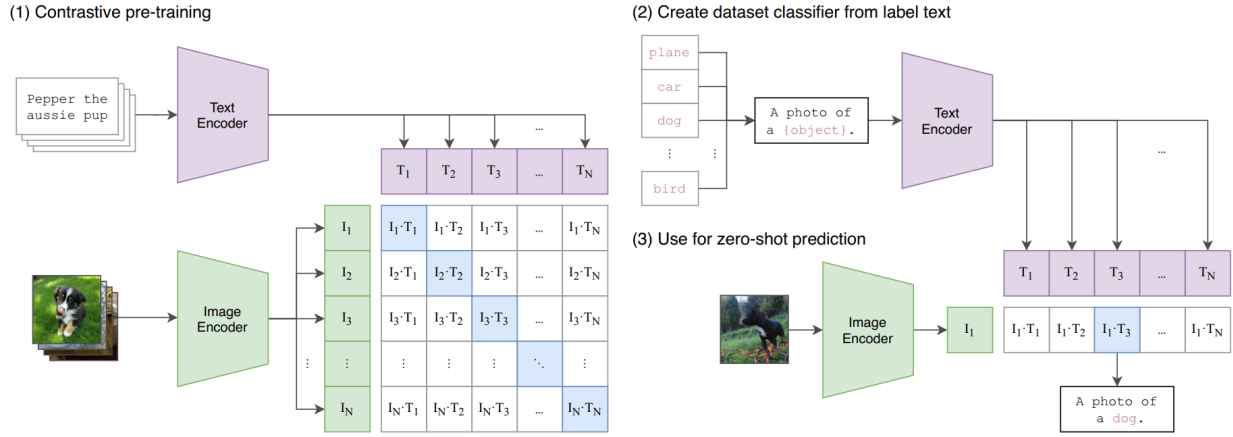


Figure 5.5: CLIP

5.17.1 Dual Encoder Architecture

Two parallel encoders:

$$z_{\text{img}} = f_{\text{img}}(\text{image}) \quad (\text{vision encoder}) \quad (5.218)$$

$$z_{\text{txt}} = f_{\text{txt}}(\text{text}) \quad (\text{text encoder}) \quad (5.219)$$

$$z_{\text{img}}, z_{\text{txt}} \in \mathbb{R}^d \quad (\text{shared space}) \quad (5.220)$$

where:

- f_{img} : Vision Transformer or CNN
- f_{txt} : Text Transformer
- Outputs normalized: $\|z\|_2 = 1$

5.17.2 Contrastive Loss

For batch of N (image, text) pairs:

$$s_{ij} = \frac{z_{\text{img}}^{(i)\top} z_{\text{txt}}^{(j)}}{\tau} \quad (\text{scaled similarity}) \quad (5.221)$$

$$p_i = \frac{\exp(s_{ii})}{\sum_j \exp(s_{ij})} \quad (\text{image to text}) \quad (5.222)$$

$$q_i = \frac{\exp(s_{ii})}{\sum_j \exp(s_{ji})} \quad (\text{text to image}) \quad (5.223)$$

Symmetric cross entropy loss:

$$\mathcal{L} = -\frac{1}{2N} \sum_{i=1}^N (\log p_i + \log q_i) \quad (5.224)$$

5.17.3 Understanding Contrastive Learning

Matrix form of similarities:

$$S = \begin{bmatrix} s_{11} & s_{12} & \cdots & s_{1N} \\ s_{21} & s_{22} & \cdots & s_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ s_{N1} & s_{N2} & \cdots & s_{NN} \end{bmatrix} \quad (5.225)$$

Loss encourages:

- Diagonal terms large (matching pairs)
- Off-diagonal terms small (non-matching)
- Symmetric treatment of modalities
- Hard negative mining within batch

5.17.4 Temperature Scaling

Role of temperature τ :

- Controls similarity sharpness:
 - Lower τ : sharper distinctions
 - Higher τ : softer matching
- Affects training dynamics:
 - Gradient scaling
 - Hard negative emphasis
 - Learning stability

5.17.5 Training Process

Key aspects:

- Large-scale data:
 - Internet-scale image-text pairs
 - Natural language supervision
 - Diverse visual concepts
- Efficient implementation:
 - All pairs computed in parallel
 - GPU-efficient matrix operations
 - Large batch sizes important
- Learned representations:
 - Visual concepts align with language
 - Zero-shot transfer possible
 - Flexible visual reasoning

5.17.6 Applications

Zero-shot capabilities:

- Classification:
 - Encode class names as text
 - Match with image features
 - No task-specific training
- Image retrieval:
 - Text queries to image space
 - Semantic similarity search
 - Open vocabulary search
- Text retrieval:
 - Image queries to text space
 - Image captioning basis
 - Cross-modal understanding

Chapter 6

Diffusion Model

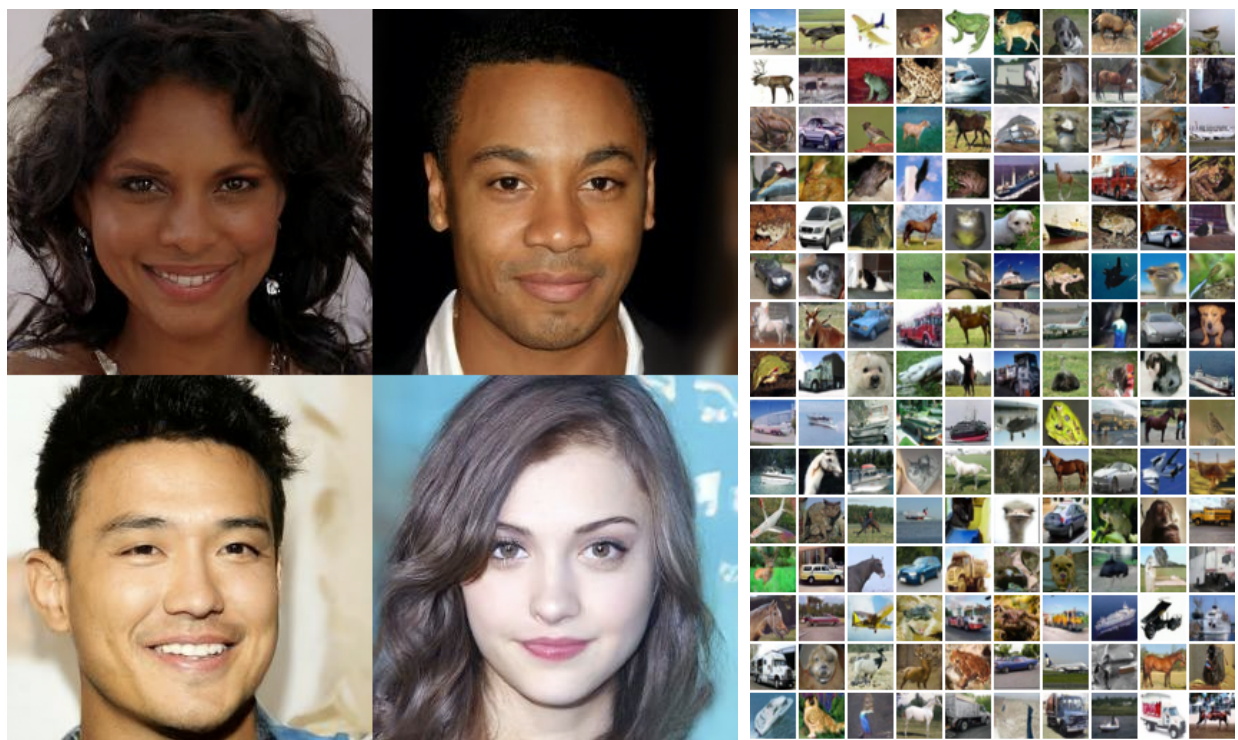


Figure 1: Generated samples on CelebA-HQ 256×256 (left) and unconditional CIFAR10 (right)

Figure 6.1: Denoising Diffusion Probability Model

Chapter Overview

This chapter presents diffusion models, a powerful framework for generating complex high-dimensional data by transforming simple Gaussian noise into structured data through an iterative denoising process. The development begins with fundamental probability concepts, illustrated through the intuitive analogy of one billion particles moving between states and

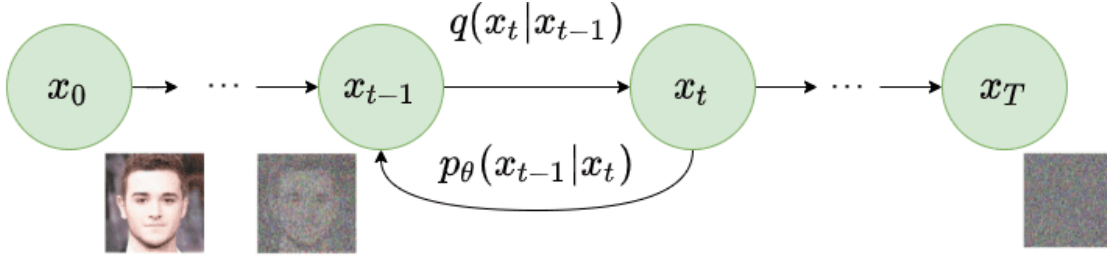


Figure 6.2: q is our P_{data} for generating the trajectory data.

returning to their original distribution. This population movement analogy provides concrete understanding for both discrete and continuous probability rules that underlie diffusion models.

The mathematical framework builds from simple Gaussian trajectories where data points are gradually perturbed with noise until they become pure Gaussian noise, and then learns to reverse this process for generation. Key developments include variance reduction through conditional expectations, alternative formulations based on directly predicting clean data, and connections to classical results like Vincent's identity (Tweedie's formula). The $t - 2$ reasoning leads to deterministic denoising process. Taking limit leads to stochastic and ordinary differential equations. Equivalence between stochastic and deterministic drifts justifies flow matching.

For practical implementation, the framework is realized through neural networks (UNet and Transformer architectures) that learn to gradually denoise data.

6.1 Probability Preliminaries: Counting Population

Understanding probability through population movement provides an intuitive foundation for grasping diffusion models.

6.1.1 Discrete Random Variables: Population Movement Between States

Consider a population of 1 billion people distributed across three states. This setup provides a concrete way to understand fundamental probability concepts:

- $p(x)$: Number (in billions) of people in state x (initial distribution)
- $p(y|x)$: Fraction of people moving from state x to state y (transition probability)
- $\tilde{p}(y)$: Number (in billions) ending up in state y (final distribution)

Three fundamental rules govern population movement:

1. **Chain Rule:** The joint count $p(x, y)$ of people initially in x and ending in y is:

$$p(x, y) = p(x)p(y|x)$$

For example, if state 1 has 0.4 billion people and 30% move to state 2, then $p(1, 2) = 0.4 \times 0.3 = 0.12$ billion people start in state 1 and end in state 2.

2. **Marginalization Rule:** The final population $\tilde{p}(y)$ in state y is:

$$\tilde{p}(y) = \sum_x p(x, y) = \sum_x p(x)p(y|x)$$

This sums up people arriving in y from all possible starting states.

3. **Conditioning Rule:** The fraction $p(x|y)$ of people in state y who came from state x is:

$$p(x|y) = \frac{p(x, y)}{\tilde{p}(y)} = \frac{p(x)p(y|x)}{\sum_x p(x)p(y|x)}$$

These three rules underlie all the probability calculations.

We call $p(y|x)$ the **forward conditional** (where people go) and $p(x|y)$ the **backward conditional** (where people came from). We can interpret x as cause and y as effect, and the conditioning rule is the Bayes rule.

While we can directly interpret our quantities as population counts and fractions, there is an equivalent probabilistic interpretation: if we randomly sample one person from the population of 1 billion people, then $p(x)$ becomes the probability that this person is in state x , and $p(y|x)$ becomes the conditional probability that this person will move to state y given that they are currently in state x . Similarly, $\tilde{p}(y)$ represents the probability that our randomly sampled person ends up in state y . This dual interpretation - as both population counts and probabilities - helps build intuition for probability concepts. We'll examine both discrete and continuous cases.

Let's illustrate with concrete numbers:

State (x)	$p(x)$	Interpretation
1	0.4	0.4 billion people
2	0.3	0.3 billion people
3	0.3	0.3 billion people

Table 6.1: Initial distribution

$p(y x)$	$y = 1$	$y = 2$	$y = 3$
$x = 1$	0.6	0.3	0.1
$x = 2$	0.2	0.5	0.3
$x = 3$	0.1	0.4	0.5

Table 6.2: Forward transition probabilities

Using these numbers, we can calculate:

$$\tilde{p}(1) = 0.4(0.6) + 0.3(0.2) + 0.3(0.1) = 0.33$$

Similar calculations give us $\tilde{p}(2)$ and $\tilde{p}(3)$.

An important property emerges: if we take the $\tilde{p}(y)$ people who ended up in state y and send fraction $p(x|y)$ back to state x , we recover the original distribution $p(x)$. This is like running time backward for one step - while each person may not return to their exact starting point, the overall population distribution returns to its initial state. Mathematically:

$$\sum_y \tilde{p}(y)p(x|y) = p(x)$$

This reversibility property is like a “one-step time machine” for probability distributions, and it is crucial for understanding how diffusion models recover clean data from noisy data. Just as we can reverse one step of diffusion by using the backward conditional probability, diffusion models will learn to reverse multiple steps to recover clean data from noise.

6.1.2 Continuous Random Variables: Population Distribution on a Line

For continuous random variables, we work with probability densities rather than direct probabilities. The key relationship is:

$$\text{probability} = \text{density} \times \text{size}$$

Consider 1 billion people (or particles) distributed along a continuous line where:

- $p(x)\Delta x$ is the number of people in interval $(x, x + \Delta x)$
- $p(y|x)\Delta y$ is the fraction moving from $(x, x + \Delta x)$ to $(y, y + \Delta y)$
- $p(x|y)\Delta x$ is the fraction in $(y, y + \Delta y)$ who came from $(x, x + \Delta x)$

The same probability rules apply in continuous form:

1. Chain Rule:

$$p(x, y)\Delta x\Delta y = p(x)\Delta x \cdot p(y|x)\Delta y$$

$$p(x, y) = p(x)p(y|x)$$

2. Marginalization Rule:

$$\tilde{p}(y)\Delta y = \sum_x p(x, y)\Delta x\Delta y = \sum_x p(x)\Delta x \cdot p(y|x)\Delta y$$

$$\tilde{p}(y) = \int p(x, y)dx = \int p(x)p(y|x)dx$$

3. Conditional Rule:

$$p(x|y)\Delta x = \frac{p(x, y)\Delta x\Delta y}{\tilde{p}(y)\Delta y}$$

$$p(x|y) = \frac{p(x, y)}{\tilde{p}(y)} \propto p(x)p(y|x)$$

The proportionality in the conditional rule or Bayes rule holds when viewing x as the variable and fixing y . The normalizing constant ensures the probabilities sum to 1.

These fundamental probability rules form the mathematical foundation for understanding diffusion models, where we'll see how data points (our "population") spread out under noise and how we can guide them back to their original distribution.

6.2 Noising and Denoising: A Single Step

After understanding basic probability rules, we can now study how data points get perturbed by noise and how we can recover the original data. This forms the foundation for understanding diffusion models.

6.2.1 The Forward Noising Process

Consider a random variable x with probability density function $p(x)$. We add Gaussian noise to create a noisy observation y :

$$y = x + e, \quad \text{where } e \sim \mathcal{N}(0, \sigma^2)$$

Since the noise e is independent of x , the conditional probability of y given x follows a Gaussian distribution:

$$p(y|x) \sim \mathcal{N}(x, \sigma^2)$$

Explicitly:

$$p(y|x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp \left[-\frac{(y-x)^2}{2\sigma^2} \right]$$

This defines our forward process: how clean data x gets transformed into noisy data y .

6.2.2 The Backward Denoising Process

The more interesting question is: given a noisy observation y , what can we say about the original x ? This is answered by the backward conditional $p(x|y)$.

Derivation Using Bayes Rule

From Bayes rule, we know:

$$p(x|y) \propto p(x)p(y|x)$$

Taking the logarithm:

$$\log p(x|y) = \log p(x) + \log p(y|x) + C$$

where C is a normalization constant.

Substituting the Gaussian form of $p(y|x)$:

$$\log p(x|y) = \log p(x) - \frac{(y-x)^2}{2\sigma^2} + C'$$

Taylor Expansion

For small σ^2 , we expect x to be close to y . We can use Taylor expansion of $\log p(x)$ around y :

$$\log p(x) \approx \log p(y) + \nabla \log p(y)(x - y)$$

Substituting this into our expression:

$$\begin{aligned} \log p(x|y) &\approx \log p(y) + \nabla \log p(y)(x - y) - \frac{(y - x)^2}{2\sigma^2} + C' \\ &= \log p(y) - \nabla \log p(y)(y - x) - \frac{(y - x)^2}{2\sigma^2} + C' \end{aligned}$$

Completing the Square

The quadratic terms in $(y - x)$ determine the shape of our Gaussian. Let's complete the square:

$$\begin{aligned} & - \nabla \log p(y)(y - x) - \frac{(y - x)^2}{2\sigma^2} \\ &= -\frac{1}{2\sigma^2} [(y - x)^2 + 2\sigma^2 \nabla \log p(y)(y - x)] \\ &= -\frac{1}{2\sigma^2} [(y - x + \sigma^2 \nabla \log p(y))^2 - (\sigma^2 \nabla \log p(y))^2] \end{aligned}$$

Therefore:

$$\log p(x|y) = -\frac{1}{2\sigma^2} (x - (y + \sigma^2 \nabla \log p(y)))^2 + C''$$

where C'' combines all terms not involving x .

Final Result

This quadratic form in $\log p(x|y)$ implies that $p(x|y)$ is Gaussian with:

$$p(x|y) \sim \mathcal{N}(y + \sigma^2 \nabla \log p(y), \sigma^2)$$

This result has a beautiful interpretation:

- The mean of x given y is $y + \sigma^2 \nabla \log p(y)$
- The first term y represents our noisy observation
- The correction term $\sigma^2 \nabla \log p(y)$ pushes us toward regions of high probability density
- The variance σ^2 represents our uncertainty in this reconstruction

The term $\nabla \log p(y)$ is called the *score function*. It tells us how to modify our noisy observation to better match the clean data distribution. This insight is fundamental to diffusion models, where we'll need to learn this score function to guide noisy samples back to clean data.

6.2.3 Reversibility of the Noising Process

Consider how distributions evolve under noising and denoising:

- **Forward Evolution:**

- Start with distribution $p(x)$
- Add noise: $y = x + e$, where $e \sim \mathcal{N}(0, \sigma^2)$
- End with more spread out distribution $\tilde{p}(y)$

- **Backward Evolution** (Time Machine):

- For each position y , optimal denoising is:

$$p(x|y) \sim \mathcal{N}(y + \sigma^2 \nabla \log p(y), \sigma^2)$$

- The term $\sigma^2 \nabla \log p(y)$ moves particles toward high-density regions
- This movement precisely counters the spreading effect of noise
- Why? Because noise spreads particles away from high-density regions

- **Perfect Recovery:**

- Taking particles at y and moving them by $p(x|y)$ recovers $p(x)$
- Like running time backward for one step
- Individual particles may end up in different positions
- But the overall distribution returns exactly to $p(x)$

Moving toward high density is key: noise spreads particles out from dense regions (like diffusion), so moving back toward density concentrates them again (like anti-diffusion). This is why the score function $\nabla \log p(y)$ enables our probability “time machine” to work.

6.2.4 Why Score Reverses Noising

Consider our billion particles initially concentrated in regions of high probability density. When we add noise, these particles spread out, moving from high-density to low-density regions. Let’s understand why following the score function $\nabla \log p(y)$ precisely reverses this spreading.

Particle Flow Under Noise

Imagine focusing on a small region around position y . The net flow of particles through this region under noise has two components:

- **Influx:** Particles flowing in from nearby positions
- **Outflux:** Particles flowing out to nearby positions

The difference between influx and outflux determines how the particle density changes. In regions of high density:

- More particles are present to flow out
- Fewer particles are in surrounding regions to flow in
- Result: Net loss of particles (density decreases)

Conversely, in regions of low density:

- Fewer particles are present to flow out
- More particles are in surrounding regions to flow in
- Result: Net gain of particles (density increases)

This process continues until the density gradient disappears and particles are uniformly spread out.

Score Function as Anti-Diffusion

The score function $\nabla \log p(y)$ points in the direction of steepest increase in log probability density. For our billion particles:

- $\nabla \log p(y)$ points toward regions where more particles are concentrated
- The magnitude is larger when the density gradient is steeper
- Moving particles along $\sigma^2 \nabla \log p(y)$ creates a flow opposite to noise diffusion

6.2.5 Stochastic Denoising and Deterministic Denoising

When recovering clean data from noisy observations, we have two approaches: stochastic denoising that samples from the posterior distribution, and deterministic denoising that estimates the mean of the posterior. Let's understand their relationship and why deterministic denoising requires a critical half-step correction.

Stochastic Denoising

The posterior distribution $p(x|y)$ is Gaussian with:

$$p(x|y) \sim \mathcal{N}(y + \sigma^2 \nabla \log p(y), \sigma^2)$$

To sample from this distribution, we can write:

$$x = y + \sigma^2 \nabla \log p(y) + \tilde{e}, \quad \tilde{e} \sim \mathcal{N}(0, \sigma^2)$$

Let's understand this process with our billion particles:

- Each noisy particle at position y moves along $\sigma^2 \nabla \log p(y)$
- Additional random noise \tilde{e} is added to each particle
- This maintains uncertainty in our reconstruction
- The distribution of these samples matches $p(x)$

The Problem with Naive Deterministic Denoising

One might think to simply use the mean of the posterior:

$$x_- = y + \sigma^2 \nabla \log p(y)$$

However, this leads to a paradox:

- If we take our stochastic sample x and write it in terms of x_- :

$$x = x_- + \tilde{e}$$

- This reveals that x is just a noisy version of x_-
- Therefore, x_- must be too concentrated
- The distribution of x_- will be more peaked than $p(x)$

Optimal Deterministic Denoising

The solution is to take a half-step along the score direction:

$$x = y + \frac{1}{2} \sigma^2 \nabla \log p(y)$$

To understand why this works, consider our billion particles:

- Stochastic denoising: Each particle takes a full step + random noise
- Full deterministic step: Particles over-concentrate
- Half deterministic step: Particles land with correct spread

We can verify this with a simple example:

- Initial clean distribution: $p(x) = \mathcal{N}(0, 1)$
- After noise: $\tilde{p}(y) = \mathcal{N}(0, 1 + \sigma^2)$
- Score function: $\nabla \log p(y) = -\frac{y}{1 + \sigma^2}$
- Full step result: $\mathcal{N}(0, 1 - \sigma^2)$ (too narrow)
- Half step result: $\mathcal{N}(0, 1)$ (exactly right)

Relationship Between Approaches

The two approaches offer different trade-offs:

- **Stochastic Denoising:**
 - Samples from full posterior distribution
 - Maintains diversity in outputs
 - Useful for generating multiple plausible reconstructions
- **Deterministic Denoising:**
 - Provides single best estimate
 - More computationally efficient
 - Requires crucial half-step correction

Both approaches have their place in practice. Stochastic denoising is often used in generative models where diversity is desired, while deterministic denoising is preferred in applications requiring single, consistent outputs.

6.3 Trajectory-Based Data Augmentation

6.3.1 Motivation and Challenges

A fundamental problem in machine learning is modeling and generating high-dimensional data such as natural images. Let $p_{\text{data}}(\mathbf{x})$ denote the probability distribution of such data. This distribution typically exhibits challenging properties:

- Complex, irregular density landscapes with multiple modes
- Concentration on low-dimensional manifolds within the ambient space
- Sharp transitions between regions of high and low density

Direct modeling of such distributions is challenging due to these characteristics.

6.3.2 Trajectory-Based Approach

Instead of directly modeling $p_{\text{data}}(\mathbf{x})$, we introduce a trajectory-based approach:

1. Start with observed examples $\mathbf{x}_0 \sim p_{\text{data}}(\mathbf{x})$
2. Construct trajectory data $(\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_T)$
3. Design the endpoint \mathbf{x}_T to follow a simple distribution:

$$\mathbf{x}_T \sim p_T(\mathbf{x}_T) \sim \mathcal{N}(0, \sigma_T^2 \mathbf{I}) \quad (6.1)$$

6.3.3 Learning the Generation Process

The key idea is to learn a parametric model $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$ for each step of the reverse process. This model attempts to approximate the true conditional distributions in the trajectory data:

$$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) \approx p_{\text{data}}(\mathbf{x}_{t-1}|\mathbf{x}_t) \quad (6.2)$$

6.3.4 Generation Process

Generation follows a trajectory from \mathbf{x}_T to \mathbf{x}_0 :

1. Sample $\mathbf{x}_T \sim p_T(\mathbf{x}_T)$

2. Iteratively sample:

$$\mathbf{x}_{t-1} \sim p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) \quad \text{for } t = T, \dots, 1 \quad (6.3)$$

3. Obtain the generated sample \mathbf{x}_0

6.3.5 Comparison with Autoregressive Models

Both trajectory-based diffusion models and autoregressive models like GPT share a fundamental principle: decomposing a complex distribution into a sequence of simple steps.

Sequential Decomposition

Both approaches factor the joint distribution into conditional probabilities:

- GPT decomposes text generation as:

$$p(\mathbf{x}) = \prod_{t=1}^T p(x_t|x_{<t}) \quad (6.4)$$

- Trajectory-based models decompose data generation as:

$$p(\mathbf{x}_0) = p(\mathbf{x}_T) \prod_{t=1}^T p(\mathbf{x}_{t-1}|\mathbf{x}_t) \quad (6.5)$$

Key Common Principles

Simplification through Steps Both approaches transform a complex modeling task into a sequence of simpler ones:

- Each step in GPT: model $p(x_t|x_{<t})$ as a simple categorical distribution
- Each step in trajectories: model $p(\mathbf{x}_{t-1}|\mathbf{x}_t)$ as a simple continuous distribution

Chain Structure Both leverage a chain structure:

$$\text{GPT: } x_1 \rightarrow x_2 \rightarrow \cdots \rightarrow x_n \quad (6.6)$$

$$\text{Trajectory: } \mathbf{x}_T \rightarrow \mathbf{x}_{T-1} \rightarrow \cdots \rightarrow \mathbf{x}_0 \quad (6.7)$$

Remark 2. The core insight shared by both approaches is that complex distributions become manageable when broken down into sequences of simple steps, each with a simple conditional distribution.

6.4 Simple Gaussian Trajectory Construction: Noising and Denoising

Let us consider a simple scheme for constructing trajectories through additive Gaussian noise. This construction provides intuitive insight into the trajectory-based approach while maintaining mathematical simplicity.

6.4.1 Forward Process Construction

We construct the trajectory through a simple additive process:

$$\mathbf{x}_t = \mathbf{x}_{t-1} + \mathbf{e}_t \quad (6.8)$$

where:

- $\mathbf{e}_t \sim \mathcal{N}(0, \sigma^2 \mathbf{I})$ is Gaussian noise
- σ^2 is a small variance parameter
- The noise terms \mathbf{e}_t are independent across time steps

6.4.2 Transition Probability

This construction implies a simple transition probability:

$$p_{\text{data}}(\mathbf{x}_t | \mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t | \mathbf{x}_{t-1}, \sigma^2 \mathbf{I}) \quad (6.9)$$

Remark 3. This simple scheme transforms our original data point \mathbf{x}_0 into a sequence of nearby points, creating a gentle path toward a more tractable distribution.

6.4.3 Terminal Distribution Analysis

Let us analyze the distribution of \mathbf{x}_T for large T (e.g., $T = 1000$). By iterating the forward process:

$$\mathbf{x}_T = \mathbf{x}_0 + \sum_{t=1}^T \mathbf{e}_t \quad (6.10)$$

$$= \mathbf{x}_0 + \sum_{t=1}^T \mathcal{N}(0, \sigma^2 \mathbf{I}) \quad (6.11)$$

By independence of the noise terms, we have:

$$\mathbf{x}_T | \mathbf{x}_0 \sim \mathcal{N}(\mathbf{x}_0, T\sigma^2 \mathbf{I}) \quad (6.12)$$

For large T , this implies:

- The variance grows as $T\sigma^2$
- The initial point \mathbf{x}_0 becomes negligible compared to accumulated noise
- The distribution approaches a Gaussian regardless of initial distribution

Remark 4. For $T\sigma^2 \gg \|\mathbf{x}_0\|^2$, the terminal distribution \mathbf{x}_T effectively becomes $\mathcal{N}(0, T\sigma^2 \mathbf{I})$, independent of the starting point \mathbf{x}_0 .

This provides a natural bridge between:

- The complex data distribution at $t = 0$
- A simple Gaussian distribution at $t = T$

6.4.4 Derivation of the Reverse Transition Distribution

Let us derive the crucial result for the reverse process $p_{\text{data}}(\mathbf{x}_{t-1} | \mathbf{x}_t)$.

Bayes Rule Application

By Bayes rule:

$$p_{\text{data}}(\mathbf{x}_{t-1} | \mathbf{x}_t) = \frac{p_{\text{data}}(\mathbf{x}_t | \mathbf{x}_{t-1}) p_{\text{data}}(\mathbf{x}_{t-1})}{p_{\text{data}}(\mathbf{x}_t)} \propto p_{\text{data}}(\mathbf{x}_{t-1}) p_{\text{data}}(\mathbf{x}_t | \mathbf{x}_{t-1}) \quad (6.13)$$

Substituting the forward transition:

$$p_{\text{data}}(\mathbf{x}_{t-1} | \mathbf{x}_t) \propto p_{\text{data}, t-1}(\mathbf{x}_{t-1}) \exp\left(-\frac{\|\mathbf{x}_t - \mathbf{x}_{t-1}\|^2}{2\sigma^2}\right) \quad (6.14)$$

where $p_{\text{data}, t-1}$ is the distribution $p_{\text{data}}(\mathbf{x}_{t-1})$. We add $t - 1$ to the subscript because it is needed to avoid confusion in the next subsection.

First Order Taylor Expansion

For notation simplicity, we temporarily drop the subscript data .

For small σ^2 , we expect \mathbf{x}_{t-1} to be close to \mathbf{x}_t . Let's expand $\log p_{t-1}(\mathbf{x}_{t-1})$ around \mathbf{x}_t :

$$\log p_{t-1}(\mathbf{x}_{t-1}) \approx \log p_{t-1}(\mathbf{x}_t) + \nabla \log p_{t-1}(\mathbf{x}_t)^\top (\mathbf{x}_{t-1} - \mathbf{x}_t) \quad (6.15)$$

Therefore:

$$\log p(\mathbf{x}_{t-1}|\mathbf{x}_t) \approx c + \nabla \log p_{t-1}(\mathbf{x}_t)^\top (\mathbf{x}_{t-1} - \mathbf{x}_t) - \frac{\|\mathbf{x}_t - \mathbf{x}_{t-1}\|^2}{2\sigma^2} \quad (6.16)$$

$$= c - \frac{1}{2\sigma^2} \|\mathbf{x}_{t-1} - \mathbf{x}_t - \sigma^2 \nabla \log p_{t-1}(\mathbf{x}_t)\|^2 \quad (6.17)$$

where c is a constant that has nothing to do with \mathbf{x}_{t-1} .

Key Result

The above quadratic form implies that for small σ^2 :

$$p_{\text{data}}(\mathbf{x}_{t-1}|\mathbf{x}_t) \approx \mathcal{N}(\mathbf{x}_t + \sigma^2 \nabla \log p_{\text{data},t-1}(\mathbf{x}_t), \sigma^2 \mathbf{I}) \quad (6.18)$$

Remark 5. This is a fundamental result showing that:

- The reverse process is Gaussian
- The mean involves the score function $\nabla \log p_{\text{data},t-1}(\mathbf{x}_t)$
- The variance equals the forward process variance σ^2

6.4.5 Uniqueness of Gaussian

A crucial aspect of diffusion models is their reliance on Gaussian distributions. This is not an arbitrary choice but a necessity driven by the curse of dimensionality. In high-dimensional spaces like images, the iid Gaussian is essentially the only distribution that remains tractable. It uniquely allows both simple sampling and efficient density evaluation. Other distributions either become impossible to sample or have intractable normalization constants. Moreover, the Gaussian has the crucial property that sums of independent Gaussians remain Gaussian with closed-form parameters. This property is essential for diffusion models as it allows us to track the distribution of accumulated noise in closed form. This uniqueness of the Gaussian explains why the forward process must use Gaussian noise, why the reverse process must be Gaussian, and why the terminal distribution must be Gaussian. There is simply no other mathematically feasible choice in high dimensions.

6.4.6 Necessity of Small Noise Variance

The requirement that σ^2 be small is fundamental to our diffusion framework. This choice enables three crucial mathematical simplifications:

Taylor Expansion For small σ^2 , we expect \mathbf{x}_{t-1} to be close to \mathbf{x}_t , allowing first-order Taylor expansion:

$$\log p_{\text{data}}(\mathbf{x}_{t-1}) \approx \log p_{\text{data}}(\mathbf{x}_t) + \nabla \log p_{\text{data}}(\mathbf{x}_t)^\top (\mathbf{x}_{t-1} - \mathbf{x}_t) \quad (6.19)$$

This approximation would fail for large σ^2 as higher-order terms become significant.

Gaussian Approximation Small σ^2 ensures the reverse transition remains approximately Gaussian:

$$p_{\text{data}}(\mathbf{x}_{t-1}|\mathbf{x}_t) \approx \mathcal{N}(\mathbf{x}_t + \sigma^2 \nabla \log p_{\text{data}}(\mathbf{x}_t), \sigma^2 \mathbf{I}) \quad (6.20)$$

Without small σ^2 , the reverse distribution could be arbitrarily complex, making modeling and sampling intractable in high dimensions.

Remark 6. These simplifications are not just mathematical conveniences but practical necessities:

- Only Gaussian distributions are tractable in high dimensions
- Only simple parametric forms can be efficiently learned
- Only closed-form sampling procedures are computationally feasible

The small σ^2 assumption thus enables the entire diffusion framework by ensuring we stay within the realm of tractable computations.

6.5 Score-Based Parametrization

Based on our previous derivation, the reverse transition probability takes the form of a simple Gaussian distribution. This makes modeling, learning, and generation very easy. In fact, the simple Gaussian distribution is the only continuous distribution we can handle in high dimension.

6.5.1 Single Neural Network Parametrization

A key insight is that we can model all timesteps with a single neural network:

$$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_t + \sigma^2 s_\theta(\mathbf{x}_t, t), \sigma^2 \mathbf{I}) \quad (6.21)$$

where:

- $s_\theta(\mathbf{x}_t, t)$ is a single neural network
- t is simply an additional input to the network
- The same parameters θ are used for all timesteps

Remark 7. This is a crucial efficiency: rather than learning T separate networks or maintaining T sets of parameters, we learn a single network that handles all timesteps through its time input t . We can embed t into a high-dimensional vector as an input to the network.

6.5.2 Learning the diffusion model

The training loss function becomes:

$$\mathcal{L}(\theta) = \mathbb{E}_{t, \mathbf{x}_{t-1}, \mathbf{x}_t} [\|\mathbf{x}_{t-1} - (\mathbf{x}_t + \sigma^2 s_\theta(\mathbf{x}_t, t))\|^2] \quad (6.22)$$

where t follows the uniform distribution, so that expectation is the average over t .

6.5.3 Generation Process

After learning the score network $s_\theta(\mathbf{x}_t, t)$, generating a single sample is straightforward:

- Start with random noise: $\mathbf{x}_T \sim \mathcal{N}(0, T\sigma^2 \mathbf{I})$
- Iteratively denoise using the learned score:

$$\mathbf{x}_{t-1} = \mathbf{x}_t + \sigma^2 s_\theta(\mathbf{x}_t, t) + \tilde{\mathbf{e}}_t,$$

where $\tilde{\mathbf{e}}_t \sim \mathcal{N}(0, \sigma^2 \mathbf{I})$

- After T steps: \mathbf{x}_0 is our generated sample

The score s_θ guides the noisy point back to high-density regions of the data distribution, like a compass pointing toward the data manifold.

6.5.4 Alternative Loss: Predicting Clean Data

Instead of predicting the previous noisy state, we can aim directly for the clean data:

$$\mathcal{L}(\theta) = \mathbb{E}_{t, \mathbf{x}_0, \mathbf{x}_t} [\|\mathbf{x}_0 - (\mathbf{x}_t + t\sigma^2 s_\theta(\mathbf{x}_t, t))\|^2] \quad (6.23)$$

We shall derive this loss function rigorously later as a variance reduction scheme.

This modification has several advantages:

- Provides cleaner target (\mathbf{x}_0 instead of \mathbf{x}_{t-1})
- Reduces accumulation of errors
- Better signal for learning

We can continue to use the same generation process above.

6.5.5 Noise Prediction

We can rewrite this in terms of noise prediction. Since:

$$\mathbf{x}_t = \mathbf{x}_0 + \boldsymbol{\epsilon}_t, \quad \text{where } \boldsymbol{\epsilon}_t \sim \mathcal{N}(0, t\sigma^2 \mathbf{I}) \quad (6.24)$$

Let $\boldsymbol{\epsilon}_\theta(\mathbf{x}, t) = -t\sigma^2 s_\theta(\mathbf{x}, t)$. Then the loss becomes:

$$\mathcal{L}(\theta) = \mathbb{E}_{t, \mathbf{x}_0, \boldsymbol{\epsilon}_t} [\|\boldsymbol{\epsilon}_t - \boldsymbol{\epsilon}_\theta(\mathbf{x}_0 + \boldsymbol{\epsilon}_t, t)\|^2] \quad (6.25)$$

This reformulation is intuitive: given a noisy observation $\mathbf{x}_0 + \boldsymbol{\epsilon}_t$, predict how much noise $\boldsymbol{\epsilon}_t$ was added.

6.5.6 Generation Process with Noise Prediction

After learning the noise prediction network $\epsilon_\theta(\mathbf{x}, t)$, we can generate samples by iteratively removing predicted noise:

- Start with random noise: $\mathbf{x}_T \sim \mathcal{N}(0, T\sigma^2\mathbf{I})$
- For $t = T, T-1, \dots, 1$:
 - Given \mathbf{x}_t , predict noise: $\epsilon_\theta(\mathbf{x}_t, t)$
 - Remove predicted noise: $\mathbf{x}_{t-1} = \mathbf{x}_t - \frac{\epsilon_\theta(\mathbf{x}_t, t)}{t} + \tilde{\mathbf{e}}_t$
 - Where $\tilde{\mathbf{e}}_t \sim \mathcal{N}(0, \sigma^2\mathbf{I})$
- Final sample is \mathbf{x}_0

Note that this is equivalent to our previous formulation since $\epsilon_\theta(\mathbf{x}, t) = -t\sigma^2 s_\theta(\mathbf{x}, t)$, but the interpretation is more intuitive: we directly predict and remove noise at each step.

6.5.7 Scaling

We can generalize the forward process to have

$$\mathbf{x}_t = c_t \mathbf{x}_{t-1} + \mathbf{e}_t \quad (6.26)$$

where $\mathbf{e}_t \sim \mathcal{N}(0, \sigma_t^2)$, and we scale \mathbf{x}_{t-1} by c_t . We allow σ_t^2 to change over t . A typical choice is $c_t = \sqrt{1 - \sigma_t^2}$. We can also use $\beta_t = \sigma_t^2$, and $\alpha_t = c_t^2$. Such a scaling is not essential either theoretically or practically. For backward sampling, we can first generate $\tilde{\mathbf{x}}_{t-1} = c_t \mathbf{x}_{t-1}$, and then obtain $\mathbf{x}_{t-1} = \tilde{\mathbf{x}}_{t-1}/c_t$.

6.5.8 UNet Parametrization of the Score Network

Architecture Overview

The score network $s_\theta(\mathbf{x}, t)$ or noise predictor $\epsilon_\theta(\mathbf{x}, t)$ is typically implemented using a UNet architecture, which has several advantageous properties for diffusion models:

- Multi-scale processing through downsampling and upsampling
- Feature preservation via skip connections
- Ability to capture both local and global context
- Memory efficiency through progressive feature compression

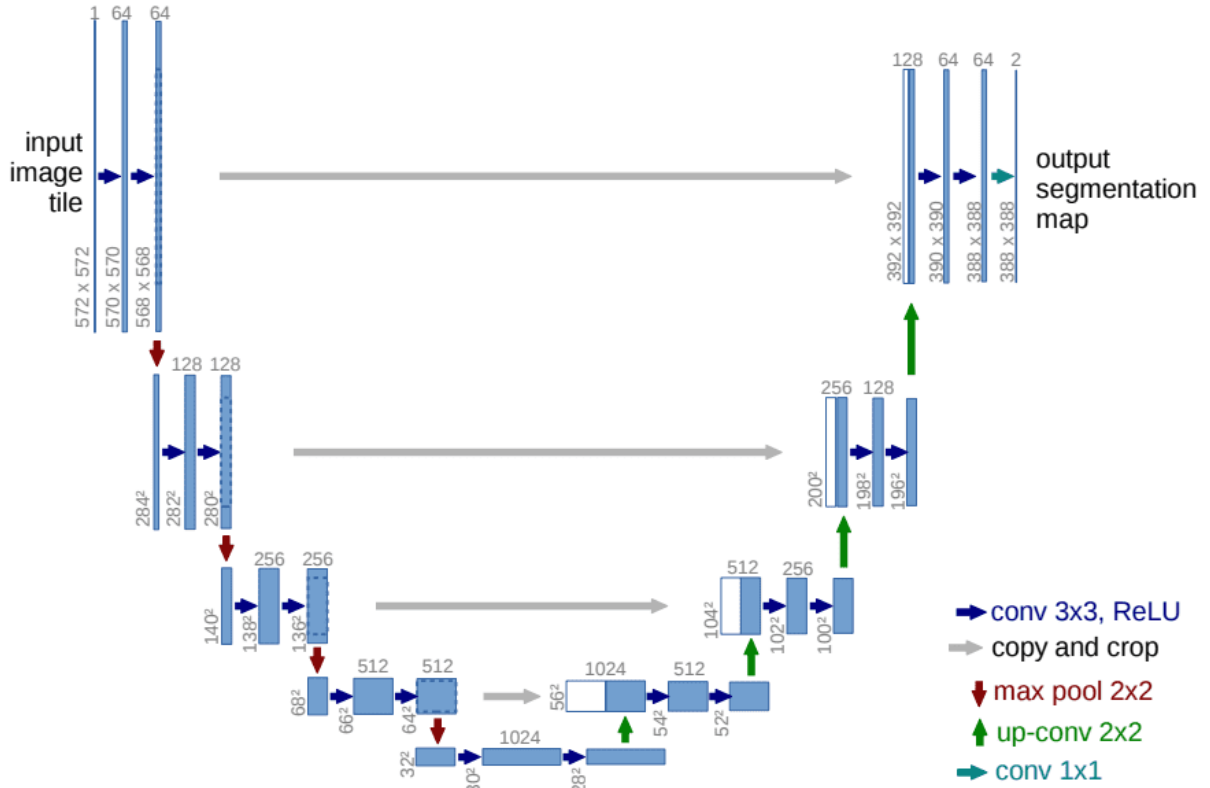


Figure 6.3: UNet

Time Embedding

The timestep t is embedded into a high-dimensional vector through sinusoidal position encoding:

$$\text{emb}(t)_i = \begin{cases} \sin(t/10000^{2i/d}) & \text{if } i \text{ even} \\ \cos(t/10000^{2i/d}) & \text{if } i \text{ odd} \end{cases} \quad (6.27)$$

where:

- d is the embedding dimension
- i indexes the dimension
- The embedding is then processed by an MLP: $\text{emb}_\theta(t) = \text{MLP}(\text{emb}(t))$

Basic Building Blocks

The UNet consists of several key components:

ResNet Block Each resolution level contains ResNet blocks:

$$\text{ResBlock}(\mathbf{h}, \text{emb}) = \mathbf{h} + \text{NN}_\theta(\text{GroupNorm}(\mathbf{h}) + \text{Linear}(\text{emb})) \quad (6.28)$$

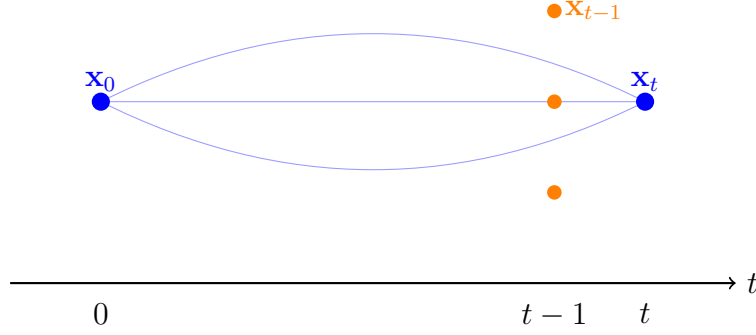


Figure 6.4: Multiple trajectories connecting fixed endpoints \mathbf{x}_0 and \mathbf{x}_t , passing through different possible \mathbf{x}_{t-1} points.

Attention Block Self-attention is applied at lower resolutions:

$$\text{Attention}(\mathbf{h}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \quad (6.29)$$

where $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ are linear projections of the input features \mathbf{h} .

6.6 Variance Reduction via Trajectory Averaging

The training objective involves nested expectations: an outer expectation over data points $\mathbf{x}_0 \sim p_{\text{data}}(\mathbf{x}_0)$, and for each \mathbf{x}_0 , inner expectations over trajectories $\mathbf{x}_1, \dots, \mathbf{x}_T$ sampled from the noising process. In practice, we estimate these expectations using Monte Carlo sampling. However, Monte Carlo estimation introduces variance that can slow down training. A fundamental principle in statistical computing is to replace Monte Carlo averages with closed-form expectations whenever possible, as this eliminates sampling variance (equivalent to using infinitely many Monte Carlo samples for that particular averaging step). In our case, while we must use Monte Carlo sampling for \mathbf{x}_0 from the empirical data distribution, the inner expectations over trajectories have Gaussian structure that we can exploit. This leads us to analyze how we can compute certain trajectory statistics in closed form, thereby reducing the overall variance of our gradient estimates.

6.6.1 Multiple Trajectories Perspective

Consider a fixed data point \mathbf{x}_0 , there can be a lot of trajectories, e.g., 1 trillion trajectories starting from this \mathbf{x}_0 . There may be 1 million of them go through the same point \mathbf{x}_t at time t . Each such trajectory goes through a \mathbf{x}_{t-1} , so there are 1 million \mathbf{x}_{t-1} .

The average loss among these $M = 1$ million $\mathbf{x}_t^{(m)}$, $m = 1, \dots, M$ is

$$\frac{1}{M} \sum_{m=1}^M \frac{\|\mathbf{x}_{t-1}^{(m)} - (\mathbf{x}_t + \sigma^2 s_\theta(\mathbf{x}_t, t))\|^2}{2\sigma^2} = \frac{\|\bar{\mathbf{x}}_{t-1} - (\mathbf{x}_t + \sigma^2 s_\theta(\mathbf{x}_t, t))\|^2}{2\sigma^2} + \text{constant} \quad (6.30)$$

where

$$\bar{\mathbf{x}}_{t-1} = \frac{1}{M} \sum_{m=1}^M \mathbf{x}_{t-1}^{(m)} \rightarrow \mathbb{E}[\mathbf{x}_{t-1} | \mathbf{x}_0, \mathbf{x}_t] \quad (6.31)$$

Using $\bar{\mathbf{x}}_{t-1}$ instead of \mathbf{x}_{t-1} as target, we squeeze out the variance among the $\mathbf{x}_{t-1}^{(m)}$.

For our Gaussian process, we can derive $\mathbb{E}[\mathbf{x}_{t-1} | \mathbf{x}_0, \mathbf{x}_t]$ in closed form:

Lemma 8 (Conditional Expectation). *For the process $\mathbf{x}_s = \mathbf{x}_{s-1} + \mathbf{e}_s$ where $\mathbf{e}_s \sim \mathcal{N}(0, \sigma^2 \mathbf{I})$:*

$$\bar{\mathbf{x}}_{t-1} = \mathbb{E}[\mathbf{x}_{t-1} | \mathbf{x}_0, \mathbf{x}_t] = \frac{1}{t} \mathbf{x}_0 + \left(1 - \frac{1}{t}\right) \mathbf{x}_t \quad (6.32)$$

6.6.2 Variance-Reduced Loss with Conditional Mean

Let's define the cumulative noise:

$$\boldsymbol{\epsilon}_t = \sum_{i=1}^t \mathbf{e}_i = \mathbf{x}_t - \mathbf{x}_0 \sim \mathcal{N}(0, t\sigma^2 \mathbf{I}) \quad (6.33)$$

where each $\mathbf{e}_i \sim \mathcal{N}(0, \sigma^2 \mathbf{I})$ independently.

Recall the conditional mean:

$$\bar{\mathbf{x}}_{t-1} = \frac{1}{t} \mathbf{x}_0 + \left(1 - \frac{1}{t}\right) \mathbf{x}_t \quad (6.34)$$

6.6.3 Deriving Alternative Loss via Conditional Mean

The original loss using $\bar{\mathbf{x}}_{t-1}$ is:

$$\mathcal{L}(\theta) = \mathbb{E}_{t, \mathbf{x}_{t-1}, \mathbf{x}_t} [\|\bar{\mathbf{x}}_{t-1} - (\mathbf{x}_t + \sigma^2 s_\theta(\mathbf{x}_t, t))\|^2]$$

Plugging in $\bar{\mathbf{x}}_{t-1} = \frac{1}{t} \mathbf{x}_0 + \left(1 - \frac{1}{t}\right) \mathbf{x}_t$:

$$\mathcal{L}(\theta) = \mathbb{E}_{t, \mathbf{x}_0, \mathbf{x}_t} \left[\left\| \frac{1}{t} \mathbf{x}_0 + \left(1 - \frac{1}{t}\right) \mathbf{x}_t - (\mathbf{x}_t + \sigma^2 s_\theta(\mathbf{x}_t, t)) \right\|^2 \right]$$

Rearranging:

$$\mathcal{L}(\theta) = \mathbb{E}_{t, \mathbf{x}_0, \mathbf{x}_t} \left[\left\| \frac{1}{t} (\mathbf{x}_0 - \mathbf{x}_t) - \sigma^2 s_\theta(\mathbf{x}_t, t) \right\|^2 \right]$$

Scaling by t^2 (which doesn't change the optimal solution):

$$\mathcal{L}(\theta) = \mathbb{E}_{t, \mathbf{x}_0, \mathbf{x}_t} [\|\mathbf{x}_0 - (\mathbf{x}_t + t\sigma^2 s_\theta(\mathbf{x}_t, t))\|^2]$$

This is exactly our alternative loss targeting \mathbf{x}_0 directly.

6.7 Connection to Denoising Auto-Encoder and Vincent Identity

6.7.1 Denoising Auto-Encoder

Our score-based diffusion model has a deep connection to denoising auto-encoders:

- The score network prediction $\mathbf{x}_t + t\sigma^2 s_\theta(\mathbf{x}_t, t)$ is actually predicting $\mathbb{E}[\mathbf{x}_0|\mathbf{x}_t]$
- This matches exactly the form of Vincent's identity (also known as Tweedie formula):

$$\mathbb{E}[\mathbf{x}_0|\mathbf{x}_t] = \mathbf{x}_t + t\sigma^2 \nabla \log p_t(\mathbf{x}_t) \quad (6.35)$$

- Therefore, when $s_\theta(\mathbf{x}_t, t)$ is optimal, it must equal the true score $\nabla \log p_t(\mathbf{x}_t)$

This connection reveals that our diffusion model is essentially learning optimal denoising. When we predict \mathbf{x}_0 from noisy \mathbf{x}_t , we're performing denoising auto-encoding, with the score function naturally emerging from the optimal denoising solution.

Remark 9. This equivalence explains why:

- Our model learns to denoise gradually
- The score function guides samples toward high density regions
- Predicting \mathbf{x}_0 directly is a natural objective

6.7.2 Proof of Vincent Identity

Setup

For a noisy observation \mathbf{x}_t , we want to compute:

$$\mathbb{E}[\mathbf{x}_0|\mathbf{x}_t] = \int \mathbf{x}_0 p(\mathbf{x}_0|\mathbf{x}_t) d\mathbf{x}_0 \quad (6.36)$$

Proof Steps

Step 1: Bayes Rule

$$p(\mathbf{x}_0|\mathbf{x}_t) = \frac{p(\mathbf{x}_t|\mathbf{x}_0)p(\mathbf{x}_0)}{p_t(\mathbf{x}_t)} \quad (6.37)$$

Step 2: Gaussian Noise Model

$$p(\mathbf{x}_t|\mathbf{x}_0) = \frac{1}{(2\pi t\sigma^2)^{d/2}} \exp\left(-\frac{\|\mathbf{x}_t - \mathbf{x}_0\|^2}{2t\sigma^2}\right) \quad (6.38)$$

Step 3: Compute Mean

$$\mathbb{E}[\mathbf{x}_0|\mathbf{x}_t] = \int \mathbf{x}_0 \frac{p(\mathbf{x}_t|\mathbf{x}_0)p(\mathbf{x}_0)}{p_t(\mathbf{x}_t)} d\mathbf{x}_0 \quad (6.39)$$

$$= \mathbf{x}_t + t\sigma^2 \frac{\int \frac{\mathbf{x}_0 - \mathbf{x}_t}{t\sigma^2} p(\mathbf{x}_t|\mathbf{x}_0)p(\mathbf{x}_0) d\mathbf{x}_0}{p_t(\mathbf{x}_t)} \quad (6.40)$$

$$= \mathbf{x}_t + t\sigma^2 \nabla_{\mathbf{x}_t} \log p_t(\mathbf{x}_t) \quad (6.41)$$

Remark 10. Key points in this derivation:

- Uses proper conditioning through Bayes rule
- Explicitly handles the Gaussian form of $p(\mathbf{x}_t|\mathbf{x}_0)$
- Score function emerges from the gradient of log marginal

6.8 Noise Prediction Parameterization

Based on our analysis of the conditional mean, we can parameterize the reverse process through noise prediction:

$$\epsilon_\theta(\mathbf{x}_t, t) \approx \epsilon_t = \mathbf{x}_t - \mathbf{x}_0 \quad (6.42)$$

where:

- $\epsilon_\theta(\mathbf{x}_t, t)$ is a neural network that predicts the cumulative noise
- $\epsilon_t = \sum_{i=1}^t \mathbf{e}_i$ is the true cumulative noise
- The same parameters θ are used for all timesteps

6.8.1 Loss Function

The training objective is a simple mean squared error between predicted and actual noise:

$$\mathcal{L}(\theta) = \mathbb{E}_{\mathbf{x}_0, \epsilon_t} \|\epsilon_t - \epsilon_\theta(\mathbf{x}_0 + \epsilon_t, t)\|^2 \quad (6.43)$$

Remark 11. This formulation has several advantages:

- Direct prediction of the noise component
- Simple MSE loss without scaling factors
- Natural interpretation as denoising
- Good numerical conditioning

6.8.2 Training Algorithm

Algorithm 4 Training Algorithm

Input: Training data $\{\mathbf{x}_0^{(i)}\}_{i=1}^N$, number of timesteps T , noise scale σ^2

while not converged do:

- (a) Sample minibatch $\{\mathbf{x}_0^{(i)}\}_{i=1}^B$ from training data
- (b) Sample timesteps $t \sim \text{Uniform}(1, T)$ for each sample
- (c) Sample noise $\boldsymbol{\epsilon}_t \sim \mathcal{N}(0, t\sigma^2\mathbf{I})$ for each sample
- (d) Compute noisy samples $\mathbf{x}_t = \mathbf{x}_0 + \boldsymbol{\epsilon}_t$
- (e) Compute loss $\mathcal{L} = \frac{1}{B} \sum_{i=1}^B \|\boldsymbol{\epsilon}_t^{(i)} - \epsilon_\theta(\mathbf{x}_t^{(i)}, t^{(i)})\|^2$
- (f) Update θ using gradient descent on \mathcal{L}

end while

6.8.3 Sampling Algorithm

Given a trained noise prediction model $\epsilon_\theta(\mathbf{x}, t)$, we can generate samples by reversing the diffusion process:

Algorithm 5 Reverse Diffusion Sampling

Input: Noise prediction model ϵ_θ , time steps T , noise scale σ^2

Initialize: Sample $\mathbf{x}_T \sim \mathcal{N}(0, T\sigma^2\mathbf{I})$

For each $t = T, T-1, \dots, 1$:

- (a) Sample noise: $\mathbf{z}_t \sim \mathcal{N}(0, \mathbf{I})$
- (b) Denoise: $\mathbf{x}_{t-1} = \mathbf{x}_t - \frac{\sigma^2}{t} \epsilon_\theta(\mathbf{x}_t, t) + \sigma \mathbf{z}_t$

Output: Generated sample \mathbf{x}_0

Remark 12. In the sampling process:

- The predicted noise is scaled by σ^2/t
- Random noise \mathbf{z}_t is added for stochasticity
- The process gradually denoises the sample from Gaussian noise to a data sample

6.9 Maximum Likelihood and Kullback-Leibler Divergence

6.9.1 General Setting

Let P_{data} denote the true data distribution and P_{θ} denote our model distribution. The maximum likelihood objective is:

$$\max_{\theta} \mathbb{E}_{\mathbf{x} \sim P_{\text{data}}} [\log P_{\theta}(\mathbf{x})] \quad (6.44)$$

In practice, the expectation $\mathbb{E}_{\mathbf{x} \sim P_{\text{data}}}$ is approximated by averaging over training data.

This is equivalent to minimizing the Kullback-Leibler divergence:

$$D_{KL}(P_{\text{data}} \| P_{\theta}) = \mathbb{E}_{\mathbf{x} \sim P_{\text{data}}} \left[\log \frac{P_{\text{data}}(\mathbf{x})}{P_{\theta}(\mathbf{x})} \right] \quad (6.45)$$

$$= \mathbb{E}_{\mathbf{x} \sim P_{\text{data}}} [\log P_{\text{data}}(\mathbf{x})] - \mathbb{E}_{\mathbf{x} \sim P_{\text{data}}} [\log P_{\theta}(\mathbf{x})] \quad (6.46)$$

$$= H(P_{\text{data}}) - \mathbb{E}_{\mathbf{x} \sim P_{\text{data}}} [\log P_{\theta}(\mathbf{x})] \quad (6.47)$$

where $H(P_{\text{data}})$ is the entropy of the data distribution.

Remark 13. Since $H(P_{\text{data}})$ is constant with respect to θ , maximizing likelihood is equivalent to minimizing $D_{KL}(P_{\text{data}} \| P_{\theta})$.

6.9.2 Extension to Trajectories

For diffusion models, we consider trajectories $\mathbf{x} = (\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_T)$. The distributions become:

- $P_{\text{data}}(\mathbf{x})$: Distribution over trajectories starting from real data
- $P_{\theta}(\mathbf{x})$: Model distribution over trajectories

The objective remains:

$$\min_{\theta} D_{KL}(P_{\text{data}} \| P_{\theta}) = \min_{\theta} \mathbb{E}_{\mathbf{x} \sim P_{\text{data}}} \left[\log \frac{P_{\text{data}}(\mathbf{x})}{P_{\theta}(\mathbf{x})} \right] \quad (6.48)$$

6.9.3 Trajectory Distributions

For diffusion models, we consider trajectories $\mathbf{x} = (\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_T)$. The distributions decompose as:

$$P_{\text{data}}(\mathbf{x}) = p_{\text{data}}(\mathbf{x}_0) \prod_{t=1}^T p_{\text{data}}(\mathbf{x}_t | \mathbf{x}_{t-1}) \quad (6.49)$$

$$P_{\theta}(\mathbf{x}) = p(\mathbf{x}_T) \prod_{t=1}^T p_{\theta}(\mathbf{x}_{t-1} | \mathbf{x}_t) \quad (6.50)$$

Maximum Likelihood Decomposition

The log-likelihood of a trajectory becomes:

$$\log P_\theta(\mathbf{x}) = \log p(\mathbf{x}_T) + \sum_{t=1}^T \log p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) \quad (6.51)$$

Taking expectation over the data distribution:

$$\begin{aligned} \mathbb{E}_{\mathbf{x} \sim P_{\text{data}}}[\log P_\theta(\mathbf{x})] &= \mathbb{E}_{\mathbf{x}_T \sim p_{\text{data}}(\mathbf{x}_T)}[\log p(\mathbf{x}_T)] + \sum_{t=1}^T \mathbb{E}_{\mathbf{x} \sim P_{\text{data}}}[\log p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)] \\ &= \mathbb{E}_{\mathbf{x}_T \sim p_{\text{data}}(\mathbf{x}_T)}[\log p(\mathbf{x}_T)] + \sum_{t=1}^T \mathbb{E}_{(\mathbf{x}_{t-1}, \mathbf{x}_t) \sim p_{\text{data}}(\mathbf{x}_{t-1}, \mathbf{x}_t)}[\log p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)] \end{aligned} \quad (6.52)$$

(6.53)

Remark 14. The key insight is that:

- The expectation over full trajectories decomposes into expectations over pairs $(\mathbf{x}_{t-1}, \mathbf{x}_t)$
- Each term involves the marginal distribution $p_{\text{data}}(\mathbf{x}_{t-1}, \mathbf{x}_t)$
- This allows us to optimize each conditional separately

6.9.4 Learning the diffusion model

The log-likelihood becomes:

$$\log p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = -\frac{\|\mathbf{x}_{t-1} - (\mathbf{x}_t + \sigma^2 s_\theta(\mathbf{x}_t, t))\|^2}{2\sigma^2} + \text{const} \quad (6.54)$$

The training loss function becomes:

$$\mathcal{L}(\theta) = \mathbb{E}_{t, \mathbf{x}_{t-1}, \mathbf{x}_t} [\|\mathbf{x}_{t-1} - (\mathbf{x}_t + \sigma^2 s_\theta(\mathbf{x}_t, t))\|^2] \quad (6.55)$$

where t follows the uniform distribution, so that expectation is the average over t .

Remark 15. This formulation has several advantages:

- Direct optimization of log-likelihood
- Simple Gaussian form with learned mean
- Score function captured by neural network

6.9.5 Connection to KL Divergence

The negative log-likelihood becomes:

$$\begin{aligned}
-\mathbb{E}_{\mathbf{x} \sim P_{\text{data}}} [\log P_{\theta}(\mathbf{x})] &= -\mathbb{E}_{\mathbf{x}_T} [\log p(\mathbf{x}_T)] - \sum_{t=1}^T \mathbb{E}_{p_{\text{data}}(\mathbf{x}_{t-1}, \mathbf{x}_t)} [\log p_{\theta}(\mathbf{x}_{t-1} | \mathbf{x}_t)] \\
&= D_{KL}(p_{\text{data}}(\mathbf{x}_T) \| p(\mathbf{x}_T)) + \sum_{t=1}^T \mathbb{E}_{\mathbf{x}_t} [D_{KL}(p_{\text{data}}(\mathbf{x}_{t-1} | \mathbf{x}_t) \| p_{\theta}(\mathbf{x}_{t-1} | \mathbf{x}_t))] + \text{const}
\end{aligned} \tag{6.56}$$

$$(6.57)$$

This decomposition suggests two key properties:

- The terminal distribution $p(\mathbf{x}_T)$ should match $p_{\text{data}}(\mathbf{x}_T)$
- Each reverse conditional $p_{\theta}(\mathbf{x}_{t-1} | \mathbf{x}_t)$ should match $p_{\text{data}}(\mathbf{x}_{t-1} | \mathbf{x}_t)$

6.9.6 Trajectory Distribution Factorization

A key insight in our development is to factorize the trajectory distribution $P_{\text{data}}(\mathbf{x})$ in a specific way that exploits our previous closed-form calculations. While the forward process naturally suggests the factorization:

$$P_{\text{data}}(\mathbf{x}) = p_{\text{data}}(\mathbf{x}_0) \prod_{t=1}^T p_{\text{data}}(\mathbf{x}_t | \mathbf{x}_{t-1}) \quad (\text{forward}) \tag{6.58}$$

we instead choose:

$$P_{\text{data}}(\mathbf{x}) = p_{\text{data}}(\mathbf{x}_0) p_{\text{data}}(\mathbf{x}_T | \mathbf{x}_0) \prod_{t=1}^T p_{\text{data}}(\mathbf{x}_{t-1} | \mathbf{x}_t, \mathbf{x}_0) \quad (\text{conditional}) \tag{6.59}$$

This choice is motivated by our previous variance reduction analysis, where we found that $\mathbb{E}_{p_{\text{data}}(\mathbf{x}_{t-1} | \mathbf{x}_t, \mathbf{x}_0)} [\|\mathbf{x}_{t-1} - (\mathbf{x}_t + \frac{\sigma_t^2}{t} \epsilon_{\theta}(\mathbf{x}_t, t))\|^2]$ has a closed-form solution. By introducing $p_{\text{data}}(\mathbf{x}_{t-1} | \mathbf{x}_t, \mathbf{x}_0)$ in our factorization, we can directly utilize this closed-form expectation in the subsequent KL divergence calculations.

The model distribution remains:

$$P_{\theta}(\mathbf{x}) = p(\mathbf{x}_T) \prod_{t=1}^T p_{\theta}(\mathbf{x}_{t-1} | \mathbf{x}_t) \tag{6.60}$$

6.9.7 KL Divergence Decomposition

The KL divergence becomes:

$$D_{KL}(P_{\text{data}} \| P_{\theta}) = \mathbb{E}_{\mathbf{x} \sim P_{\text{data}}} \left[\log \frac{p_{\text{data}}(\mathbf{x}_0) p_{\text{data}}(\mathbf{x}_T | \mathbf{x}_0) \prod_{t=1}^T p_{\text{data}}(\mathbf{x}_{t-1} | \mathbf{x}_t, \mathbf{x}_0)}{p(\mathbf{x}_T) \prod_{t=1}^T p_{\theta}(\mathbf{x}_{t-1} | \mathbf{x}_t)} \right] \tag{6.61}$$

$$\begin{aligned}
&= \mathbb{E}_{\mathbf{x}_0} \left[\log \frac{p_{\text{data}}(\mathbf{x}_0) p_{\text{data}}(\mathbf{x}_T | \mathbf{x}_0)}{p(\mathbf{x}_T)} \right] + \sum_{t=1}^T \mathbb{E}_{\mathbf{x}_0, \mathbf{x}_t} [D_{KL}(p_{\text{data}}(\mathbf{x}_{t-1} | \mathbf{x}_t, \mathbf{x}_0) \| p_{\theta}(\mathbf{x}_{t-1} | \mathbf{x}_t))] \\
&\tag{6.62}
\end{aligned}$$

6.9.8 Local KL Terms

Each local KL term is:

$$D_{KL}(p_{\text{data}}(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) \| p_{\theta}(\mathbf{x}_{t-1}|\mathbf{x}_t)) \quad (6.63)$$

$$= \mathbb{E}_{\mathbf{x}_{t-1} \sim p_{\text{data}}(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)} \left[\log \frac{p_{\text{data}}(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)}{p_{\theta}(\mathbf{x}_{t-1}|\mathbf{x}_t)} \right] \quad (6.64)$$

Since both distributions are Gaussian with same variance $\sigma^2 \mathbf{I}$:

$$D_{KL}(p_{\text{data}}(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) \| p_{\theta}(\mathbf{x}_{t-1}|\mathbf{x}_t)) = \frac{1}{2\sigma^2} \|\mathbb{E}[\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0] - (\mathbf{x}_t - \frac{\sigma^2}{t} \epsilon_{\theta}(\mathbf{x}_t, t))\|^2 \quad (6.65)$$

6.9.9 Final Objective

Using $\mathbb{E}[\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0] = \mathbf{x}_t - \frac{1}{t} \epsilon_t$, we get:

$$\mathcal{L}(\theta) = \frac{1}{2} \sum_{t=1}^T \mathbb{E}_{\mathbf{x}_0, \epsilon_t} [\|\epsilon_t - \epsilon_{\theta}(\mathbf{x}_0 + \epsilon_t, t)\|^2] \quad (6.66)$$

Remark 16. Key insights:

- The conditional formulation naturally incorporates \mathbf{x}_0 information
- KL terms directly involve conditional means
- No expectation over \mathbf{x}_{t-1} is needed
- Gives same objective but from a different perspective

6.10 Deterministic Sampling: $t - 2$ Reasoning

Consider omitting the noise term \mathbf{z}_t in the sampling process:

$$\tilde{\mathbf{x}}_{t-1} = \mathbf{x}_t + \sigma^2 s_{\theta}(\mathbf{x}_t, t) \quad (6.67)$$

where $\tilde{\mathbf{x}}_{t-1}$ denotes our deterministic update.

For the discussion below, assuming the score network $s_{\theta}(\mathbf{x}_t, t)$ estimates the score function $\nabla \log p_{\text{data}}(\mathbf{x}_t)$ exactly for all t , and let us write

$$s(\mathbf{x}_t, t) = \nabla \log p(\mathbf{x}_t),$$

where we drop the subscripts θ and data for simplicity.

Now let us assume $\mathbf{x}_t \sim p(\mathbf{x}_t)$. A key observation is that, given \mathbf{x}_t ,

$$\tilde{\mathbf{x}}_{t-1} + \mathbf{e}_t = \mathbf{x}_t + \sigma^2 \nabla \log p(\mathbf{x}_t) + \mathbf{e}_t \sim p(\mathbf{x}_{t-1}|\mathbf{x}_t) \quad (6.68)$$

where $\mathbf{e}_t \sim \mathcal{N}(0, \sigma^2 \mathbf{I})$, and is independent of \mathbf{x}_t , so that it is also independent of $\tilde{\mathbf{x}}_{t-1}$.

Thus marginally, $\tilde{\mathbf{x}}_{t-1} + \mathbf{e}_t \sim p(\mathbf{x}_{t-1})$.

This implies:

$$\tilde{\mathbf{x}}_{t-1} \sim p(\mathbf{x}_{t-2}) \quad (6.69)$$

because $\mathbf{x}_{t-1} = \mathbf{x}_{t-2} + \mathbf{e}_t$.

That is, $\tilde{\mathbf{x}}_{t-1}$ has the same marginal distribution as $p(\mathbf{x}_{t-2})$.

Therefore, our deterministic update:

$$\mathbf{x}_{t-2} = \mathbf{x}_t + \sigma^2 s(\mathbf{x}_t, t) \quad (6.70)$$

directly gives us samples from the distribution two steps earlier.

Remark 17. This reveals that:

- The deterministic process naturally takes double steps
- Adding noise \mathbf{z}_t is unnecessary for correct marginal distributions
- We could adjust the time indexing to reflect this two-step nature

6.11 Continuous Time Analysis

Let us rescale the time interval $[0, T]$ to $[0, 1]$ for clearer analysis. Let us redefine our new σ^2 to be the old $\sigma_T^2 = T\sigma^2$. Define:

$$\Delta t = \frac{1}{T} \quad (6.71)$$

which will play the role of dt as $T \rightarrow \infty$. Thus our original σ^2 becomes $\sigma^2/T = \sigma^2 \Delta t$.

6.11.1 Forward Process

In discrete time:

$$\mathbf{x}_{t+\Delta t} = \mathbf{x}_t + \mathbf{e}_t, \quad \mathbf{e}_t \sim \sigma \sqrt{\Delta t} \mathbf{z}_t \quad (6.72)$$

where $\mathbf{z}_t \sim \mathcal{N}(0, \mathbf{I})$, and is independent of \mathbf{x}_t , thus $\mathbf{e}_t \sim \mathcal{N}(0, \sigma^2 \Delta t \mathbf{I})$

Taking $T \rightarrow \infty$ (i.e., $\Delta t \rightarrow 0$), this naturally becomes:

$$d\mathbf{x}_t = \sigma d\mathbf{w}_t \quad (6.73)$$

where \mathbf{w}_t is a standard Brownian motion, and

$$d\mathbf{w}_t \approx \sqrt{\Delta t} \mathbf{z}_t$$

Please note the $\sqrt{\Delta t}$ scaling. It is the cause of the zig-zag path because the velocity

$$d\mathbf{w}_t/dt \approx \sqrt{\Delta t} \mathbf{z}_t / \Delta t = \mathbf{z}_t / \sqrt{\Delta t} \rightarrow \infty$$

6.11.2 Stochastic Differential Equation (SDE) Backward

Moving backward in discrete time:

$$\mathbf{x}_{t-\Delta t} = \mathbf{x}_t + \sigma^2 \Delta t s_\theta(\mathbf{x}_t, t) + \mathbf{e}_t, \quad \mathbf{e}_t \sim \sigma \sqrt{\Delta t} \mathbf{z}_t \quad (6.74)$$

where $\mathbf{z}_t \sim \mathcal{N}(0, \mathbf{I})$, and is independent of \mathbf{x}_t .

As $\Delta t \rightarrow 0$, this becomes:

$$d\mathbf{x}_t = -\sigma^2 s_\theta(\mathbf{x}_t, t) dt + \sigma d\mathbf{w}_t. \quad (6.75)$$

This is called stochastic differential equation (SDE) because of the $d\mathbf{w}_t$ term.

6.11.3 Deterministic Ordinary Differential Equation (ODE) Backward

From our $t - 2$ property, the deterministic backward step is:

$$\mathbf{x}_{t-2\Delta t} = \mathbf{x}_t + \sigma^2 \Delta t s_\theta(\mathbf{x}_t, t) \quad (6.76)$$

Taking the differential form:

$$\frac{\mathbf{x}_{t-2\Delta t} - \mathbf{x}_t}{2\Delta t} = \frac{1}{2} \sigma^2 s_\theta(\mathbf{x}_t, t) \quad (6.77)$$

$$-\frac{d\mathbf{x}_t}{dt} = \frac{1}{2} \sigma^2 s_\theta(\mathbf{x}_t, t) \quad (6.78)$$

Therefore:

$$\frac{d\mathbf{x}_t}{dt} = -\frac{1}{2} \sigma^2 s_\theta(\mathbf{x}_t, t) \quad (6.79)$$

This is ordinary differential equation (ODE), which is deterministic.

Remark 18. Key insights from this analysis:

- The noise term $\sqrt{\Delta t} \mathbf{z}_t$ gives proper scaling for continuous-time limit $d\mathbf{w}_t$
- The forward process converges to Brownian motion with variance σ^2 as $\Delta t \rightarrow 0$
- The backward SDE reverses time and adds a score-based drift term
- The $\frac{1}{2}$ factor in ODE arises from taking double steps
- Both SDE and ODE preserve the marginal distributions

6.11.4 Understanding Continuous Time Through Movies

The discrete difference equations with explicit Δt and $\sqrt{\Delta t}$ terms provide clearer intuition than their differential counterparts:

- **Forward Process as Movie Frames:**
 - Each step: $\mathbf{x}_{t+\Delta t} = \mathbf{x}_t + \sigma\sqrt{\Delta t}\mathbf{z}_t$
 - Δt represents time between movie frames
 - $\sqrt{\Delta t}$ term makes noise scaling explicit
 - Like filming milk diffusing in coffee frame by frame
- **Critical $\sqrt{\Delta t}$ Scaling:**
 - $\sqrt{\Delta t}$ term requires second-order Taylor expansion
 - Creates characteristic zig-zag paths in diffusion
 - Makes instantaneous velocity infinite as $\Delta t \rightarrow 0$
 - This “roughness” is fundamental to Brownian motion
- **Backward Process as Reversed Movie:**
 - Stochastic: $\mathbf{x}_{t-\Delta t} = \mathbf{x}_t + \sigma^2\Delta ts_\theta(\mathbf{x}_t, t) + \sigma\sqrt{\Delta t}\mathbf{z}_t$
 - Like playing movie backward, frame by frame
 - Score function guides particles back to structure
 - Maxwell’s demon knowing which way to push particles

Remark 19. While rigorous SDE theory requires measure theory to properly count continuous trajectories, the movie analogy with discrete frames of Δt spacing provides intuitive understanding of:

- Why $\sqrt{\Delta t}$ scaling is natural
- How forward and backward processes relate
- Why paths are necessarily rough
- The essence of reversing diffusion

The measure-theoretic details, while mathematically important, are not essential for understanding the core principles.

6.12 Stochastic Noising and Deterministic Denoising

6.12.1 Distribution Preservation

Consider two processes:

Forward (Noising)

$$\mathbf{x}_t = \mathbf{x}_{t-\Delta t} + \mathbf{e}_t, \quad \mathbf{e}_t = \sigma\sqrt{\Delta t}\mathbf{z}_t \quad (6.80)$$

where $\mathbf{z}_t \sim \mathcal{N}(0, \mathbf{I})$, and is independent of $\mathbf{x}_{t-\Delta t}$.

Backward (Denoising)

$$\mathbf{x}_{t-\Delta t} = \mathbf{x}_t + \frac{\sigma^2 \Delta t}{2} \nabla \log p_t(\mathbf{x}_t) \quad (6.81)$$

From our previous derivation of the $t - 2$ property, we know that if $\mathbf{x}_{t-\Delta t}$ in the two above equations have the same marginal distribution.

6.12.2 Intuitive Understanding

Consider one million particles $\{\mathbf{x}_{t-\Delta t}^{(i)}\}_{i=1}^{1,000,000}$ that follow some distribution. The forward and backward processes can be understood through their effects on this particle cloud:

Forward (Diffusion) When we add noise to each particle:

$$\mathbf{x}_t^{(i)} = \mathbf{x}_{t-\Delta t}^{(i)} + \sigma\sqrt{\Delta t}\mathbf{z}_t^{(i)} \quad (6.82)$$

the particle cloud becomes more diffused, as each particle randomly moves away from its neighbors.

Backward (Condensation) The gradient step on log density:

$$\mathbf{x}_{t-\Delta t}^{(i)} = \mathbf{x}_t^{(i)} + \frac{\sigma^2 \Delta t}{2} \nabla \log p_t(\mathbf{x}_t^{(i)}) \quad (6.83)$$

moves each particle toward regions of higher particle density, as $\nabla \log p_t$ points in directions where there are more particles. This condensation exactly counteracts the previous diffusion.

Remark 20. This balance explains why:

- Random noise spreads out the particle cloud
- Gradient flow concentrates particles
- These opposing forces preserve the marginal distribution

6.12.3 Langevin Dynamics for Equilibrium Sampling

The balance between diffusion and condensation suggests the Langevin dynamics for sampling from a target distribution $\pi(\mathbf{x})$:

Equilibrium Langevin

$$\mathbf{x}_{t+\Delta t} = \mathbf{x}_t + \frac{\sigma^2 \Delta t}{2} \nabla \log \pi(\mathbf{x}_t) + \sigma \sqrt{\Delta t} \mathbf{z}_t, \quad \mathbf{z}_t \sim \mathcal{N}(0, \mathbf{I}) \quad (6.84)$$

The coefficient $\frac{1}{2}$ ensures perfect balance between:

- Forward diffusion spreading particles
- Backward condensation concentrating particles

Thus if $\mathbf{x}_t \sim \pi$, then $\mathbf{x}_{t+1} \sim \pi$ as well.

6.12.4 Non-equilibrium Sampling

The denoising process is non-equilibrium because it transport p_t to $p_{t-\Delta t}$.

Non-equilibrium SDE

$$\mathbf{x}_{t-\Delta t} = \mathbf{x}_t + \sigma^2 \Delta t \nabla \log p_t(\mathbf{x}_t) + \sigma \sqrt{\Delta t} \mathbf{z}_t \quad (6.85)$$

The coefficient 1 instead of $\frac{1}{2}$ means:

Non-equilibrium ODE

$$\mathbf{x}_{t-\Delta t} = \mathbf{x}_t + \frac{\sigma^2 \Delta t}{2} \nabla \log p_t(\mathbf{x}_t) \quad (6.86)$$

Remark 21. This comparison shows:

- Only equilibrium Langevin preserves the target distribution
- SDE and ODE change the distribution from p_t to $p_{t-\Delta t}$
- The $\frac{1}{2}$ coefficient is crucial for proper sampling

6.13 General Forward Process with Drift

6.13.1 Forward Process Analysis

Consider a more general forward process:

$$\mathbf{x}_{t+\Delta t} = \mathbf{x}_t + \mathbf{f}(\mathbf{x}_t, t) \Delta t + \sigma_t \sqrt{\Delta t} \mathbf{z}_t, \quad \mathbf{z}_t \sim \mathcal{N}(0, \mathbf{I}) \quad (6.87)$$

The two terms have different scaling:

- Drift term: $\mathbf{f}(\mathbf{x}_t, t) \Delta t \sim O(\Delta t)$
- Diffusion term: $\sigma_t \sqrt{\Delta t} \mathbf{z}_t \sim O(\sqrt{\Delta t})$

As $\Delta t \rightarrow 0$, the diffusion dominates the drift since $\sqrt{\Delta t} \gg \Delta t$. This creates the characteristic zig-zag pattern.

6.13.2 Backward Processes

Backward SDE Following our previous derivation but with the general forward process:

$$\mathbf{x}_{t-\Delta t} = \mathbf{x}_t + [-\mathbf{f}(\mathbf{x}_t, t) + \sigma_t^2 \nabla \log p_t(\mathbf{x}_t)] \Delta t + \sigma_t \sqrt{\Delta t} \mathbf{z}_t \quad (6.88)$$

where:

- $-\mathbf{f}(\mathbf{x}_t, t) \Delta t$ reverses the drift
- $\sigma_t^2 \nabla \log p_t(\mathbf{x}_t) \Delta t$ provides score-based correction
- $\sigma_t \sqrt{\Delta t} \mathbf{z}_t$ maintains the diffusion

Backward ODE The deterministic version becomes:

$$\mathbf{x}_{t-\Delta t} = \mathbf{x}_t + [-\mathbf{f}(\mathbf{x}_t, t) + \frac{\sigma_t^2}{2} \nabla \log p_t(\mathbf{x}_t)] \Delta t \quad (6.89)$$

Remark 22. Key observations:

- The SDE and ODE differ in both noise and score coefficient (1 vs 1/2)
- The score term scales with σ_t^2 as noise variance changes
- The $\sqrt{\Delta t}$ scaling of noise remains crucial for continuous-time limit
- The drift terms use Δt scaling for smooth paths

6.14 Random Drift Process

6.14.1 Process Comparison

Consider two stochastic processes on time interval $[0, 1]$:

Random Drift

$$\mathbf{x}_{t+\Delta t} = \mathbf{x}_t + \mathbf{f}(\mathbf{x}_t, t) \Delta t + \sigma(\mathbf{x}_t, t) \Delta t \mathbf{z}_t \quad (6.90)$$

where $\mathbb{E}[\mathbf{z}_t] = \mathbf{1}$, $\text{Var}[\mathbf{z}_t] = \mathbf{I}$, and $\{\mathbf{z}_t\}$ are independent.

Diffusion Process

$$\mathbf{x}_{t+\Delta t} = \mathbf{x}_t + \mathbf{f}(\mathbf{x}_t, t) \Delta t + \sigma(\mathbf{x}_t, t) \sqrt{\Delta t} \mathbf{z}_t \quad (6.91)$$

The key difference is the scaling: Δt vs $\sqrt{\Delta t}$.

6.14.2 Accumulated Variance Analysis

For the random drift, the total accumulated variance:

$$\sum_{k=1}^N \sigma^2(\mathbf{x}_{t_k}, t_k) (\Delta t)^2 \rightarrow \int_0^1 \sigma^2(\mathbf{x}_t, t) dt \cdot \Delta t \rightarrow 0 \text{ as } \Delta t \rightarrow 0 \quad (6.92)$$

This is the Law of Large Number effect.

For the diffusion:

$$\sum_{k=1}^N \sigma^2(\mathbf{x}_{t_k}, t_k) \Delta t \rightarrow \int_0^1 \sigma^2(\mathbf{x}_t, t) dt \text{ remains finite} \quad (6.93)$$

This is the Central Limit Theorem effect.

This integral view shows:

- Random drift variance has extra Δt factor, vanishing in limit
- Diffusion variance converges to a proper time integral
- The path-dependent nature through $\sigma^2(\mathbf{x}_t, t)$

6.14.3 Deterministic Equivalence

For random drift, we can replace the stochastic process with its deterministic mean:

$$\mathbf{x}_{t+\Delta t} = \mathbf{x}_t + \mathbf{f}(\mathbf{x}_t, t) \Delta t \quad (6.94)$$

This preserves the marginal distributions.

6.14.4 Backward Process

We can reverse the process with a deterministic ODE:

$$\mathbf{x}_{t-\Delta t} = \mathbf{x}_t - \mathbf{f}(\mathbf{x}_t, t) \Delta t \quad (6.95)$$

No random term is needed since the forward process randomness vanishes in the limit.

Remark 23. This analysis reveals:

- Random drift averages out (Law of Large Numbers)
- Diffusion accumulates (Central Limit Theorem)
- Δt vs $\sqrt{\Delta t}$ scaling determines limiting behavior
- Only deterministic drift needs to be reversed

6.15 Fokker-Planck Analysis

The $t - 2$ property we derived earlier is remarkably powerful and provides complete intuition about diffusion processes. Here we present Fokker-Planck analysis as a complementary tool, focusing first on univariate case for clarity. While we could work purely in continuous or discrete time, we maintain the Δt and $\sqrt{\Delta t}$ formulation throughout as it provides the clearest window into drift versus diffusion behavior.

6.15.1 Test Function Perspective

The key is to analyze via test function. Consider a smooth test function $h(x_t)$ with fast decaying tails. This can be viewed as a smoothed version of an indicator function:

- Indicator: $\mathbf{1}(x_t \in [a, b])$ for interval $[a, b]$
- Smooth test function: $h(x_t)$ approximates indicator
- $\mathbb{E}[h(x_t)]$ captures marginal distribution of x_t : $\mathbb{E}[\mathbf{1}(x_t \in [a, b])] = P((x_t \in [a, b]))$

6.15.2 SDE Analysis

Consider the scalar SDE:

$$x_{t+\Delta t} = x_t + f(x_t, t)\Delta t + \sigma(x_t, t)\sqrt{\Delta t}z_t, \quad z_t \sim \mathcal{N}(0, 1) \quad (6.96)$$

Evolution of test function expectation:

$$\mathbb{E}[h(x_{t+\Delta t})] - \mathbb{E}[h(x_t)] = \mathbb{E}[h(x_t + f\Delta t + \sigma\sqrt{\Delta t}z_t) - h(x_t)] \quad (6.97)$$

$$= \mathbb{E}[h'(x_t)(f\Delta t + \sigma\sqrt{\Delta t}z_t) + \frac{1}{2}h''(x_t)\sigma^2\Delta t] + o(\Delta t) \quad (6.98)$$

The $\sqrt{\Delta t}$ term requires second-order Taylor because:

- First-order: $\mathbb{E}[\sqrt{\Delta t}z_t] = 0$
- Second-order: $\mathbb{E}[\Delta t z_t^2] = \Delta t$
- Higher-order terms vanish as $\Delta t \rightarrow 0$

This gives Fokker-Planck equation:

$$\frac{\partial p_t}{\partial t} = -\frac{\partial}{\partial x}(fp_t) + \frac{1}{2}\frac{\partial^2}{\partial x^2}(\sigma^2 p_t) \quad (6.99)$$

6.15.3 ODE Analysis

For deterministic ODE:

$$x_{t+\Delta t} = x_t + v(x_t, t)\Delta t \quad (6.100)$$

First-order Taylor suffices:

$$\mathbb{E}[h(x_{t+\Delta t})] - \mathbb{E}[h(x_t)] = \mathbb{E}[h'(x_t)v\Delta t] + o(\Delta t) \quad (6.101)$$

Giving continuity equation:

$$\frac{\partial p_t}{\partial t} = -\frac{\partial}{\partial x}(vp_t) \quad (6.102)$$

6.15.4 SDE-ODE Equivalence

Setup Given SDE with time-dependent noise:

$$x_{t+\Delta t} = x_t + f(x_t, t)\Delta t + \sigma(t)\sqrt{\Delta t}z_t \quad (6.103)$$

Find equivalent ODE:

$$x_{t+\Delta t} = x_t + v(x_t, t)\Delta t \quad (6.104)$$

Solution The equivalent ODE velocity simplifies to:

$$v(x_t, t) = f(x_t, t) + \frac{1}{2}\sigma^2(t)\frac{\partial \log p_t}{\partial x} \quad (6.105)$$

6.15.5 Random Drift Analysis

For random drift:

$$x_{t+\Delta t} = x_t + f(x_t, t)\Delta t + \sigma(x_t, t)\Delta t z_t, \quad \mathbb{E}[z_t] = 0 \quad (6.106)$$

First-order Taylor suffices:

$$\mathbb{E}[h(x_{t+\Delta t})] - \mathbb{E}[h(x_t)] = \mathbb{E}[h'(x_t)f\Delta t] + o(\Delta t) \quad (6.107)$$

where $\sigma\Delta t z_t$ term vanishes due to $\mathbb{E}[z_t] = 0$.

6.15.6 Extension to Multivariate Case

The multivariate case follows the same principles with:

- Gradient ∇ replacing derivative $\frac{\partial}{\partial x}$
- Divergence $\nabla \cdot$ for flux terms
- Laplacian ∇^2 for diffusion terms
- Matrix $\sigma\sigma^\top$ for diffusion coefficient

Remark 24. Key insights remain the same:

- $\sqrt{\Delta t}$ noise requires second-order expansion
- Random drift needs only first-order terms
- Test functions capture distribution evolution
- Dimensionality affects notation but not principles

6.16 Flow Matching with Straight Trajectories

6.16.1 Design Principle

Unlike diffusion models that add noise incrementally, flow matching takes a fundamentally different approach to constructing trajectories. The key idea is to first sample the endpoints independently:

- Start point $\mathbf{x}_0 \sim p_{\text{data}}$
- End point $\mathbf{x}_1 \sim \mathcal{N}(0, \mathbf{I})$

and then connect them with straight lines. This design choice has profound implications:

- No diffusion term ($\sqrt{\Delta t}$ noise) is needed
- Only random drift (Δt scaling) appears
- The randomness comes entirely from the random endpoints
- The path between endpoints is deterministic

6.16.2 Non-Markovian Trajectory Data

The straight-line trajectories are non-Markovian:

- Each \mathbf{x}_t depends on both \mathbf{x}_0 and \mathbf{x}_1
- Knowing \mathbf{x}_t alone doesn't determine \mathbf{x}_{t-1}
- Full trajectory is determined by endpoints

Multiple Path Perspective Consider one trillion possible straight paths:

- Each path connects some $\mathbf{x}_0 \sim p_{\text{data}}$ to $\mathbf{x}_1 \sim \mathcal{N}(0, \mathbf{I})$
- At time t , a point \mathbf{x} might have 1000 different paths passing through it
- Each path suggests a different $\mathbf{x}_{t-\Delta t}$ given $\mathbf{x}_t = \mathbf{x}$

Markovian Backward Process Despite the non-Markovian nature of trajectories, we can use a Markovian backward process:

$$p_\theta(\mathbf{x}_{t-\Delta t}|\mathbf{x}_t) \quad (6.108)$$

This corresponds to:

- Given \mathbf{x}_t , randomly selecting one of the paths through it
- Following that path to get $\mathbf{x}_{t-\Delta t}$
- Forgetting which path was chosen

Marginal Distribution Preservation Key property: If a population of particles follows this backward process:

- Their distribution at each time t matches the marginal of trajectory data
- This holds even though individual trajectories may differ
- The randomness in path selection preserves the ensemble behavior

Remark 25. This reveals that:

- Non-Markovian data can be modeled with Markovian process
- Only marginal distributions matter for generation
- Path-level details can be forgotten
- Ensemble behavior is preserved through random path selection

6.16.3 Setup

Consider independent variables:

- $\mathbf{x}_1 \sim \mathcal{N}(0, \mathbf{I})$
- $\mathbf{x}_0 \sim p_{\text{data}}$

Define the linear interpolation:

$$\mathbf{x}_t = (1 - t)\mathbf{x}_0 + t\mathbf{x}_1 \quad (6.109)$$

6.16.4 Backward Process Analysis

For this linear interpolation, the backward transition is:

$$\mathbf{x}_{t-\Delta t} = \mathbf{x}_t + \Delta t \cdot (\mathbf{x}_0 - \mathbf{x}_1) = \mathbf{x}_t + \Delta t \mathbf{v}(\mathbf{x}_t, t) \quad (6.110)$$

where $\mathbf{v}(\mathbf{x}_t, t) = \mathbf{x}_0 - \mathbf{x}_1$ is random (depends on random $\mathbf{x}_0, \mathbf{x}_1$) but:

- Has Δt scaling (not $\sqrt{\Delta t}$)
- Therefore is a random drift process
- Can be replaced by its expectation by our previous analysis

6.16.5 Flow Matching Learning

The goal is to learn velocity field $\mathbf{v}_\theta(\mathbf{x}_t, t)$ such that:

$$\mathbf{v}_\theta(\mathbf{x}_t, t) \approx \mathbb{E}[\mathbf{v}(\mathbf{x}_t, t) | \mathbf{x}_t] \quad (6.111)$$

The learning objective is:

$$\min_{\theta} \mathbb{E}_{\mathbf{x}_0, \mathbf{x}_1, t} [\|\mathbf{v}_\theta(\mathbf{x}_t, t) - \mathbf{v}(\mathbf{x}_t, t)\|^2] \quad (6.112)$$

where:

- $\mathbf{x}_t = (1 - t)\mathbf{x}_0 + t\mathbf{x}_1$ is the interpolated point
- $\mathbf{v}(\mathbf{x}_t, t) = \mathbf{x}_0 - \mathbf{x}_1$ is the true velocity
- Training samples are pairs $(\mathbf{x}_0, \mathbf{x}_1)$ with random t

Remark 26. Key insights:

- The straight-line interpolation gives random drift
- Random drift can be replaced by deterministic drift
- Learning matches the expected velocity field
- No diffusion term needed due to Δt scaling

6.16.6 Connection to Noise and Score Prediction

When $\mathbf{x}_1 \sim \mathcal{N}(0, \mathbf{I})$, we can rewrite:

$$\mathbf{x}_t = (1 - t)\mathbf{x}_0 + t\mathbf{x}_1 = \mathbf{x}_0 + t\mathbf{x}_1 = \mathbf{x}_0 + t\boldsymbol{\epsilon} \quad (6.113)$$

where $\boldsymbol{\epsilon} = \mathbf{x}_1 \sim \mathcal{N}(0, \mathbf{I})$.

The velocity becomes:

$$\mathbf{v}(\mathbf{x}_t, t) = \mathbf{x}_0 - \mathbf{x}_1 = (\mathbf{x}_t - t\mathbf{x}_1) - \mathbf{x}_1 = \mathbf{x}_t - (t + 1)\mathbf{x}_1 = \mathbf{x}_t - (t + 1)\boldsymbol{\epsilon} \quad (6.114)$$

Therefore:

$$\|\mathbf{v}_\theta(\mathbf{x}_t, t) - \mathbf{v}(\mathbf{x}_t, t)\|^2 = \|\mathbf{v}_\theta(\mathbf{x}_t, t) - [\mathbf{x}_t - (t + 1)\boldsymbol{\epsilon}]\|^2 \quad (6.115)$$

$$= \|[\mathbf{v}_\theta(\mathbf{x}_t, t) - \mathbf{x}_t] + (t + 1)\boldsymbol{\epsilon}\|^2 \quad (6.116)$$

$$\propto \left\| \frac{\mathbf{v}_\theta(\mathbf{x}_t, t) - \mathbf{x}_t}{t + 1} + \boldsymbol{\epsilon} \right\|^2 \quad (6.117)$$

This shows:

- Velocity prediction \mathbf{v}_θ is equivalent to noise prediction $\epsilon_\theta = -\frac{\mathbf{v}_\theta - \mathbf{x}_t}{t + 1}$
- Score estimation $s_\theta = \nabla \log p_t = -\frac{\epsilon}{t}$ connects through $\mathbf{v}_\theta = \mathbf{x}_t + (t + 1)s_\theta$

- All three views (velocity, noise, score) are equivalent up to scaling

Remark 27. This reveals that:

- Flow matching with Gaussian endpoints is a form of score matching
- The straight line property provides simple velocity-score relationship
- Training objectives differ only by time-dependent scaling

6.17 Variance Scheduling

In our previous formulation, we added fixed-variance Gaussian noise at each step: $\mathbf{x}_t = \mathbf{x}_{t-1} + \mathbf{e}_t$. An alternative approach is to scale the previous state while adding noise, ensuring controlled variance growth. This leads us to introduce a variance schedule with parameters $\{\alpha_t, \beta_t\}_{t=1}^T$ that carefully balances signal preservation and noise addition.

6.17.1 Forward Process Construction

We start by defining a single step of the forward process:

Definition 28 (Forward Step). Given position \mathbf{x}_{t-1} , the next position is:

$$p_{\text{data}}(\mathbf{x}_t | \mathbf{x}_{t-1}) = \mathcal{N}(\sqrt{\alpha_t} \mathbf{x}_{t-1}, \beta_t \mathbf{I}) \quad (6.118)$$

i.e., $\mathbf{x}_t = \sqrt{\alpha_t} \mathbf{x}_{t-1} + \mathbf{e}_t$, where $\mathbf{e}_t \sim \mathcal{N}(0, \beta_t)$. where:

- $\beta_t \in (0, 1)$ controls the noise variance
- $\alpha_t = 1 - \beta_t$ controls the scaling

6.17.2 Deriving the Marginal Distribution

Let's derive $q(\mathbf{x}_t | \mathbf{x}_0)$ step by step:

Lemma 29 (Two-Step Distribution). *After two steps:*

$$\mathbf{x}_2 | \mathbf{x}_0 = \sqrt{\alpha_2}(\sqrt{\alpha_1} \mathbf{x}_0 + \sqrt{\beta_1} \boldsymbol{\epsilon}_1) + \sqrt{\beta_2} \boldsymbol{\epsilon}_2 \quad (6.119)$$

$$= \sqrt{\alpha_1 \alpha_2} \mathbf{x}_0 + \sqrt{\alpha_2 \beta_1} \boldsymbol{\epsilon}_1 + \sqrt{\beta_2} \boldsymbol{\epsilon}_2 \quad (6.120)$$

where $\boldsymbol{\epsilon}_1, \boldsymbol{\epsilon}_2 \sim \mathcal{N}(0, \mathbf{I})$ independently.

Lemma 30 (Two-Step Variance). *The variance after two steps is:*

$$\alpha_2 \beta_1 + \beta_2 = 1 - \alpha_1 \alpha_2 \quad (6.121)$$

Theorem 31 (General Marginal). *For any time t , we have:*

$$q(\mathbf{x}_t | \mathbf{x}_0) = \mathcal{N}(\sqrt{\bar{\alpha}_t} \mathbf{x}_0, (1 - \bar{\alpha}_t) \mathbf{I}) \quad (6.122)$$

where:

$$\bar{\alpha}_t = \prod_{i=1}^t \alpha_i \quad (6.123)$$

Proof. By induction on t :

1. Base case ($t = 1$): Directly from definition
2. Inductive step: Assume true for $t - 1$
3. For step t :

$$\mathbf{x}_t | \mathbf{x}_0 = \sqrt{\alpha_t} \mathbf{x}_{t-1} + \sqrt{\beta_t} \boldsymbol{\epsilon}_t \quad (6.124)$$

$$= \sqrt{\alpha_t} (\sqrt{\bar{\alpha}_{t-1}} \mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_{t-1}} \boldsymbol{\epsilon}_{t-1}) + \sqrt{\beta_t} \boldsymbol{\epsilon}_t \quad (6.125)$$

4. Collecting terms:

- Mean: $\sqrt{\alpha_t \bar{\alpha}_{t-1}} \mathbf{x}_0 = \sqrt{\bar{\alpha}_t} \mathbf{x}_0$
- Variance: $\alpha_t (1 - \bar{\alpha}_{t-1}) + \beta_t = 1 - \bar{\alpha}_t$

□

The marginal can be written in terms of a single noise variable:

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t} \mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t} \boldsymbol{\epsilon} \quad (6.126)$$

where $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$

Remark 32. This construction ensures:

- At $t = 0$: $\bar{\alpha}_0 = 1$, so \mathbf{x}_0 is unchanged
- As t increases: $\bar{\alpha}_t$ decreases
- At $t = T$: If $\bar{\alpha}_T \approx 0$, then $\mathbf{x}_T \sim \mathcal{N}(0, \mathbf{I})$
- The variance is always normalized: $\bar{\alpha}_t + (1 - \bar{\alpha}_t) = 1$

6.17.3 Training and Sampling

We can derive the training objective and sampling process by proper scaling:

Training Objective:

- Learn to predict the noise: $\epsilon_\theta(\mathbf{x}_t, t) \approx \epsilon$
- Simple L2 loss: $\mathcal{L}(\theta) = \mathbb{E}_{t, \mathbf{x}_0, \epsilon} \|\epsilon - \epsilon_\theta(\mathbf{x}_t, t)\|^2$
- Sample t uniformly, $\epsilon \sim \mathcal{N}(0, \mathbf{I})$

Sampling Process:

- Start: $\mathbf{x}_T \sim \mathcal{N}(0, \mathbf{I})$
- For $t = T, \dots, 1$:
- $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1-\alpha_t}}\epsilon_\theta(\mathbf{x}_t, t)) + \sqrt{\beta_t}\mathbf{z}_t$
- where $\mathbf{z}_t \sim \mathcal{N}(0, \mathbf{I})$

This schedule ensures smooth transformation between data and noise while maintaining tractable Gaussian form at each step. The implementation remains simple despite the more sophisticated variance control.

6.17.4 Forward Process SDE

Start with the variance scaling step:

$$\mathbf{x}_t = \sqrt{\alpha_t}\mathbf{x}_{t-1} + \sqrt{\beta_t}\epsilon_t, \quad \epsilon_t \sim \mathcal{N}(0, \mathbf{I}) \quad (6.127)$$

To connect to continuous time, we:

1. Replace discrete index with $t + \Delta t$:

$$\mathbf{x}_{t+\Delta t} = \sqrt{\alpha(t)}\mathbf{x}_t + \sqrt{\beta(t)}\epsilon_t \quad (6.128)$$

2. Write $\alpha(t) = 1 - \beta(t)$ and expand square root:

$$\mathbf{x}_{t+\Delta t} = \sqrt{1 - \beta(t)}\mathbf{x}_t + \sqrt{\beta(t)}\epsilon_t \quad (6.129)$$

$$= (1 - \frac{\beta(t)}{2})\mathbf{x}_t + \sqrt{\beta(t)}\epsilon_t + O(\beta(t)^2) \quad (6.130)$$

3. Set $\beta(t) = \tilde{\beta}(t)\Delta t$ for some function $\tilde{\beta}(t)$:

$$\mathbf{x}_{t+\Delta t} = \mathbf{x}_t - \frac{\tilde{\beta}(t)}{2}\mathbf{x}_t\Delta t + \sqrt{\tilde{\beta}(t)\Delta t}\epsilon_t + O(\Delta t^2) \quad (6.131)$$

This gives forward SDE:

$$d\mathbf{x}_t = -\frac{\tilde{\beta}(t)}{2}\mathbf{x}_t dt + \sqrt{\tilde{\beta}(t)}d\mathbf{w}_t \quad (6.132)$$

$$= \mathbf{f}(\mathbf{x}_t, t)dt + \sigma(t)d\mathbf{w}_t \quad (6.133)$$

where:

- Drift: $\mathbf{f}(\mathbf{x}_t, t) = -\frac{\tilde{\beta}(t)}{2}\mathbf{x}_t$
- Diffusion: $\sigma(t) = \sqrt{\tilde{\beta}(t)}$

Remark 33. Key points:

- $\beta(t)$ must scale with Δt
- Drift comes from Taylor expansion of $\sqrt{1 - \beta(t)}$
- Higher order terms vanish as $\Delta t \rightarrow 0$
- Original discrete steps emerge when $\Delta t = 1$

6.17.5 Backward Processes

Starting from forward SDE:

$$d\mathbf{x}_t = -\frac{\tilde{\beta}(t)}{2}\mathbf{x}_t dt + \sqrt{\tilde{\beta}(t)}d\mathbf{w}_t \quad (6.134)$$

Backward SDE The backward SDE is:

$$d\mathbf{x}_t = [\frac{\tilde{\beta}(t)}{2}\mathbf{x}_t + \tilde{\beta}(t)\nabla \log p_t(\mathbf{x}_t)]dt + \sqrt{\tilde{\beta}(t)}d\mathbf{w}_t \quad (6.135)$$

In discrete time (Δt steps):

$$\mathbf{x}_{t-\Delta t} = \mathbf{x}_t + [\frac{\tilde{\beta}(t)}{2}\mathbf{x}_t + \tilde{\beta}(t)\nabla \log p_t(\mathbf{x}_t)]\Delta t + \sqrt{\tilde{\beta}(t)}\Delta t\mathbf{z}_t \quad (6.136)$$

Connection to DDPM Denoising Diffusion Probabilistic Models (DDPM) uses:

- Noise prediction: $\epsilon_\theta = -\nabla \log p_t(\mathbf{x}_t)$
- $\beta(t) = \tilde{\beta}(t)\Delta t$

This gives DDPM update:

$$\mathbf{x}_{t-\Delta t} = \frac{1}{\sqrt{\alpha(t)}}(\mathbf{x}_t - \frac{\beta(t)}{\sqrt{1 - \bar{\alpha}(t)}}\epsilon_\theta(\mathbf{x}_t, t)) + \sqrt{\beta(t)}\mathbf{z}_t \quad (6.137)$$

Backward ODE The backward ODE takes drift term from SDE:

$$\frac{d\mathbf{x}_t}{dt} = \frac{\tilde{\beta}(t)}{2}\mathbf{x}_t + \frac{\tilde{\beta}(t)}{2}\nabla \log p_t(\mathbf{x}_t) \quad (6.138)$$

In discrete time:

$$\mathbf{x}_{t-\Delta t} = \mathbf{x}_t + \left[\frac{\tilde{\beta}(t)}{2}\mathbf{x}_t + \frac{\tilde{\beta}(t)}{2}\nabla \log p_t(\mathbf{x}_t) \right] \Delta t \quad (6.139)$$

Connection to DDIM Denoising Diffusion Implicit Models (DDIM) maintains:

$$\mathbf{x}_t = \sqrt{\bar{\alpha}(t)}\mathbf{x}_0 + \sqrt{1 - \bar{\alpha}(t)}\boldsymbol{\epsilon} \quad (6.140)$$

Using predicted noise $\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)$, DDIM update is:

$$\mathbf{x}_{t-\Delta t} = \sqrt{\frac{\bar{\alpha}(t-\Delta t)}{\bar{\alpha}(t)}}\mathbf{x}_t - \sqrt{1 - \frac{\bar{\alpha}(t-\Delta t)}{\bar{\alpha}(t)}}\sqrt{1 - \bar{\alpha}(t)}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t) \quad (6.141)$$

Remark 34. This comparison shows:

- DDPM follows stochastic backward SDE
- DDIM follows deterministic backward ODE
- Both preserve marginal distributions
- SDE provides exploration, ODE gives deterministic paths
- Different time discretizations give different practical algorithms

6.18 Applications of Diffusion Models

6.18.1 Text-to-Image Generation

Diffusion models can be extended to generate images from text descriptions:

- **Basic Idea:**
 - Condition the denoising process on text input
 - Convert text into embeddings using transformers
 - Guide the noise removal based on text features
- **Key Components:**
 - Text Encoder: Converts text to meaningful embeddings
 - Cross-Attention: Connects text and image features
 - UNet Architecture: Modified to incorporate text condition



Figure 6.5: Text to image/video generation

- **Classifier-Free Guidance:**

- Enhances text alignment during generation
- Balances between quality and text adherence
- Controls strength of text influence

6.18.2 Diffusion Transformer

An alternative architecture replacing UNet with transformers:

- **Motivation:**

- Better handles long-range dependencies
- More flexible architecture
- Improved scaling properties

- **Key Features:**

- Divides images into patches
- Uses global attention over entire image
- Incorporates time information at each layer
- Processes patches with transformer blocks

- **Advantages:**
 - Captures global image structure better
 - More natural for incorporating conditions
 - Simpler architectural design

Both approaches demonstrate how diffusion models can be adapted and enhanced for specific applications while maintaining their core denoising principles.

Chapter 7

VAE and GAN

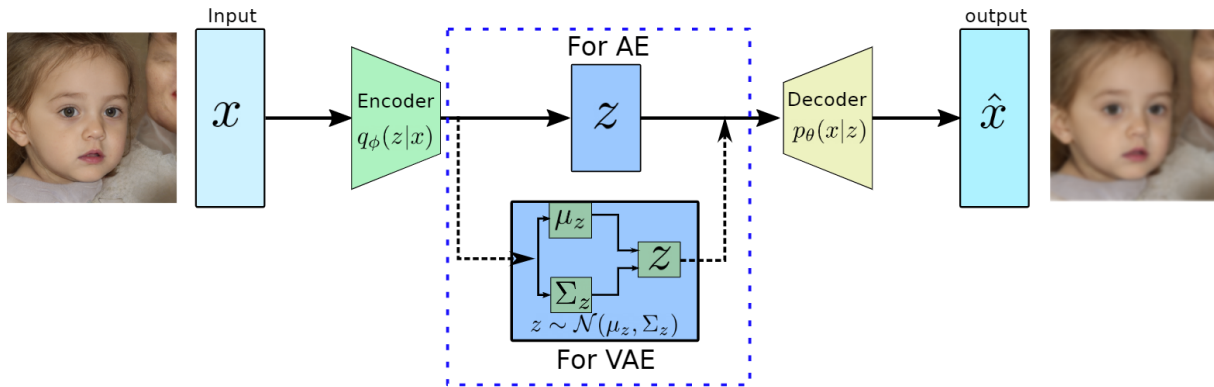


Figure 7.1: VAE

Chapter Overview

This chapter presents variational autoencoders (VAEs) through a natural progression from classical statistical principles to modern deep generative models. We begin with maximum likelihood estimation and its connection to KL divergence minimization. We then introduce latent variables as a form of data augmentation, where observed data are viewed as effects and latent variables as underlying causes.

The Evidence Lower BOund (ELBO) emerges naturally from considering the joint KL divergence between data and model distributions. This perspective reveals deep connections to classical methods: the EM algorithm appears as a special case where the inference model is tied to the current model's posterior. VAEs then emerge by making the inference model learnable, leading to joint optimization of both generative and inference components.

The chapter concludes by connecting VAEs to diffusion models, showing how the latter achieve simpler training and tighter bounds by replacing learned inference with a fixed forward process. This development reveals how modern deep generative models both build upon and strategically deviate from classical statistical principles.

We then turn to Generative Adversarial Networks (GANs), which take a radically different approach by formulating generative modeling as a two-player game between a generator and a discriminator. This game-theoretic perspective culminates in Wasserstein GAN, which recasts the discriminator as a critic that scores image quality, providing a more stable and interpretable training framework.

7.1 Maximum Likelihood and KL-Divergence

7.1.1 Empirical Distribution and Log-likelihood

Consider independent and identically distributed observations:

$$x_1, \dots, x_n \sim p_{\text{data}}(x)$$

The log-likelihood for a parametric model p_θ is:

$$\mathcal{L}(\theta) = \frac{1}{n} \sum_{i=1}^n \log p_\theta(x_i)$$

As $n \rightarrow \infty$, this converges to:

$$\mathcal{L}(\theta) \rightarrow \mathbb{E}_{p_{\text{data}}}[\log p_\theta(x)]$$

7.1.2 True Model Log-likelihood and Entropy

For the true data distribution, we define:

$$\mathcal{L}(p_{\text{data}}) = \mathbb{E}_{p_{\text{data}}}[\log p_{\text{data}}(x)] = -H(p_{\text{data}})$$

where $H(p_{\text{data}})$ is the entropy of the data distribution. This has two interpretations:

- $-\mathcal{L}(p_{\text{data}})$ measures data complexity through entropy
- Higher entropy means data is more random/complex
- Lower entropy means data has more structure/simplicity

7.1.3 KL Divergence as Log-likelihood Gap

The gap between true and model log-likelihoods is:

$$\begin{aligned} \mathcal{L}(p_{\text{data}}) - \mathcal{L}(\theta) &= \mathbb{E}_{p_{\text{data}}}[\log p_{\text{data}}(x) - \log p_\theta(x)] \\ &= D_{\text{KL}}(p_{\text{data}} \| p_\theta) \end{aligned}$$

Key insights:

- Maximum likelihood \equiv minimum KL divergence
- The gap is always non-negative: $D_{\text{KL}} \geq 0$
- Gap = 0 if and only if $p_\theta = p_{\text{data}}$
- Best achievable likelihood depends on data entropy

7.1.4 Information Geometric Interpretation

In the space of probability distributions:

- Each distribution is a point in an infinite-dimensional space
- Model family $\{p_\theta : \theta \in \Theta\}$ forms a manifold
- p_{data} is typically off this manifold
- Maximum likelihood finds the "closest" point on manifold to p_{data}
- "Closest" is measured by KL divergence
- Like projection, but KL is asymmetric

This geometric view reveals:

- Model capacity = manifold complexity
- Optimization = finding shortest path on manifold
- Model misspecification = non-zero minimum KL
- Local minima = multiple projections possible

7.1.5 Implications

This framework provides deep insights:

- Links statistical estimation to geometry
- Shows how model complexity relates to manifold dimension
- Explains why simpler data (low entropy) is easier to model
- Provides geometric intuition for optimization algorithms

7.2 Deconvolution Network with Latent Space

7.2.1 Structured Latent Representation

For objects like chairs or cars, we can decompose the latent code into:

$$z = [z_{\text{type}}, z_{\text{pose}}]$$

where:

- $z_{\text{type}} \in \{0, 1\}^K$: One-hot encoding of chair type
 - K different chair categories

- e.g., office chair, dining chair, armchair
- $z_{\text{pose}} \in \mathbb{R}^d$: Continuous camera pose parameters
 - Azimuth angle $\theta \in [0, 360]$
 - Elevation angle $\phi \in [-90, 90]$
 - Distance r from object

7.2.2 Deconvolution Network Architecture

Generator $G(z)$ maps latent code to image through progressive upsampling:

1. Input processing:

- Project z_{type} through embedding layer
- Process z_{pose} through MLP
- Concatenate and reshape to initial feature map (h_0, w_0, c_0)

2. Progressive upsampling layers:

$$(h_i, w_i, c_i) \rightarrow (2h_i, 2w_i, c_{i+1})$$

Each block contains:

- Transposed convolution: stride 2 for spatial upsampling
- BatchNorm for training stability
- ReLU activation
- Skip connections from pose parameters

3. Final output layer:

- Conv layer to get desired channels (e.g., 3 for RGB)
- Tanh activation for $[-1, 1]$ output range

7.2.3 Training

Minimize reconstruction error:

$$\min_G \mathbb{E}_{(x,z) \sim \text{data}} \|x - G(z)\|^2$$

where:

- (x, z) pairs come from labeled dataset
- z contains ground truth type and pose
- $\|\cdot\|^2$ is pixel-wise squared error

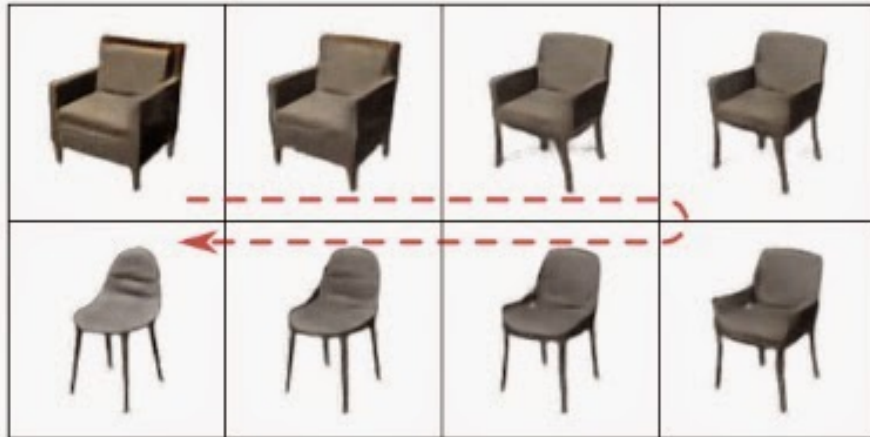


Figure 1. Interpolation between two chair models (original: top left, final: bottom left). The generative convolutional neural network learns the manifold of chairs, allowing it to interpolate between chair styles, producing realistic intermediate styles.

Figure 7.2: Latent space interpolation

7.2.4 Latent Space Interpolation

The structured latent space enables controlled interpolation:

- Type interpolation:

- Can crossfade between types:

$$z_t = [(1 - t)z_{\text{type}}^1 + tz_{\text{type}}^2, z_{\text{pose}}]$$

- Reveals learned manifold between categories

- Pose interpolation:

- Smooth rotation: interpolate azimuth

$$\theta_t = (1 - t)\theta_1 + t\theta_2$$

- View change: interpolate elevation

$$\phi_t = (1 - t)\phi_1 + t\phi_2$$

- Distance variation: interpolate r

- Joint interpolation:

- Can simultaneously vary type and pose
- Reveals understanding of 3D structure
- Shows disentanglement of factors

7.2.5 Applications

This architecture enables:

- Novel view synthesis
- Category morphing
- Controlled object manipulation
- Understanding of learned 3D representations

7.3 Latent Variable Models: From Effect to Cause

7.3.1 Data Augmentation with Latent Variables

When modeling complex data x (e.g., images), we often observe:

- The distribution $p_{\text{data}}(x)$ is highly multi-modal
- Data concentrates on a low-dimensional manifold
- Direct modeling of $p_{\text{data}}(x)$ is challenging

Instead of modeling x directly, we augment each observation with latent variables z :

- z represents underlying causes (pose, lighting, style)
- x represents observed effects (actual images)
- For now, assume we know how to augment x with z (will be addressed in VAE)

7.3.2 Generative Model Structure

We decompose the joint distribution as:

$$p_{\theta}(x, z) = p(z)p_{\theta}(x|z)$$

where:

- Prior $p(z) = \mathcal{N}(0, I)$: simple, unimodal distribution
- Conditional $p_{\theta}(x|z) = \mathcal{N}(G_{\theta}(z), \sigma^2 I)$
 - $G_{\theta}(z)$ is the deconvolution network
 - Maps latent causes to observed effects
 - Includes learnable parameters θ

7.3.3 Manifold Learning Perspective

This model has several key properties:

- Maps simple $p(z)$ to complex $p_\theta(x)$
 - Input: unimodal Gaussian
 - Output: multi-modal data distribution
 - G_θ learns to "fold" space to create modes
- Learns data manifold structure
 - $G_\theta(z)$ traces out the data manifold
 - z provides coordinates on this manifold
 - Dimension of z controls manifold complexity

7.3.4 Historical Connection: Factor Analysis

This approach has deep roots in psychometrics:

- Factor analysis (early 20th century)
 - z : underlying factors (intelligence, personality)
 - x : observed test scores
 - Linear $G_\theta(z) = Wz + b$
- Modern neural generative models
 - Same principle but nonlinear G_θ
 - Much richer transformations possible
 - Can capture complex manifold structure

7.4 From Marginal to Joint KL Divergence

7.4.1 Log-likelihood and KL Divergence

Let us start with the marginal distribution:

- Model log-likelihood:

$$\mathcal{L}(\theta) = \mathbb{E}_{p_{\text{data}}(x)}[\log p_\theta(x)]$$

- True data log-likelihood:

$$\mathcal{L}(p_{\text{data}}) = \mathbb{E}_{p_{\text{data}}(x)}[\log p_{\text{data}}(x)]$$

This represents the best achievable log-likelihood by any model.

- Marginal KL divergence:

$$\begin{aligned} D_{\text{KL}}(p_{\text{data}}(x) \| p_{\theta}(x)) &= \mathbb{E}_{p_{\text{data}}(x)} [\log p_{\text{data}}(x) - \log p_{\theta}(x)] \\ &= \mathcal{L}(p_{\text{data}}) - \mathcal{L}(\theta) \end{aligned}$$

7.4.2 Extension to Complete Data

When we augment x with latent variable z :

- Complete data model:

$$p_{\theta}(x, z) = p(z)p_{\theta}(x|z)$$

- Complete data distribution:

$$p_{\text{data}}(x, z) = p_{\text{data}}(x)p_{\text{data}}(z|x)$$

- Joint KL divergence:

$$D_{\text{KL}}(p_{\text{data}}(x, z) \| p_{\theta}(x, z)) = \mathbb{E}_{p_{\text{data}}(x, z)} \left[\log \frac{p_{\text{data}}(x)p_{\text{data}}(z|x)}{p_{\theta}(x, z)} \right]$$

7.4.3 Key Decomposition

Based on the decomposition

$$p_{\theta}(x, z) = p_{\theta}(x)p_{\theta}(z|x)$$

The joint KL can be decomposed:

Theorem 35 (Joint KL Decomposition).

$$\begin{aligned} D_{\text{KL}}(p_{\text{data}}(x, z) \| p_{\theta}(x, z)) &= D_{\text{KL}}(p_{\text{data}}(x) \| p_{\theta}(x)) \\ &\quad + \mathbb{E}_{p_{\text{data}}(x)} [D_{\text{KL}}(p_{\text{data}}(z|x) \| p_{\theta}(z|x))] \end{aligned}$$

This leads to:

$$\begin{aligned} D_{\text{KL}}(p_{\text{data}}(x, z) \| p_{\theta}(x, z)) &= \mathcal{L}(p_{\text{data}}(x)) - \mathbb{E}_{p_{\text{data}}(x)} [\log p_{\theta}(x)] + \mathbb{E}_{p_{\text{data}}(x)} [D_{\text{KL}}(p_{\text{data}}(z|x) \| p_{\theta}(z|x))] \\ &= \mathcal{L}(p_{\text{data}}(x)) - \mathbb{E}_{p_{\text{data}}(x)} [\log p_{\theta}(x) - D_{\text{KL}}(p_{\text{data}}(z|x) \| p_{\theta}(z|x))] \\ &= \mathcal{L}(p_{\text{data}}(x)) - \mathbb{E}_{p_{\text{data}}(x)} [\text{ELBO}(x|\theta)] \end{aligned}$$

where

$$\text{ELBO}(x|\theta) = \log p_{\theta}(x) - D_{\text{KL}}(p_{\text{data}}(z|x) \| p_{\theta}(z|x))$$

ELBO means evidence lower bound, which is a lower bound of the log-likelihood $\log p_{\theta}(x)$, because $D_{\text{KL}} \geq 0$.

Define

$$\text{ELBO}(\theta) = \mathbb{E}_{p_{\text{data}}(x)} [\text{ELBO}(x|\theta)]$$

Then

$$D_{\text{KL}}(p_{\text{data}}(x, z) \| p_{\theta}(x, z)) = \mathcal{L}(p_{\text{data}}) - \text{ELBO}(\theta)$$

7.4.4 Two Forms of ELBO

The above form of ELBO can be called “conceptual form” of ELBO. It is based on the decomposition $p_\theta(x, z) = p_\theta(x)p_\theta(z|x)$, where

$$p_\theta(x) = \int p_\theta(x, z)dz = \int p(z)p_\theta(x|z)dz$$

which is not intractable. As a result

$$p_\theta(z|x) = \frac{p_\theta(x, z)}{p_\theta(x)} = \frac{p(z)p_\theta(x|z)}{\int p(z)p_\theta(x|z)dz}$$

is also not tractable.

However, the above form of ELBO shows clearly that it is a lower bound of log-likelihood.

For the “computational form” of ELBO, we can use the tractable decomposition

$$p_\theta(x, z) = p(z)p_\theta(x|z).$$

Then

$$\begin{aligned} D_{\text{KL}}(p_{\text{data}}(x, z) \| p_\theta(x, z)) &= \mathbb{E}_{p_{\text{data}}(x, z)} \left[\log \frac{p_{\text{data}}(x)p_{\text{data}}(z|x)}{p(z)p_\theta(x|z)} \right] \\ &= \mathcal{L}(p_{\text{data}}(x)) - \mathbb{E}_{p_{\text{data}}(x)} \mathbb{E}_{p_{\text{data}}(z|x)} [\log p(z) + \log p_\theta(x|z) - \log p_{\text{data}}(z|x)] \\ &= \mathcal{L}(p_{\text{data}}(x)) - \mathbb{E}_{p_{\text{data}}(x)} \left[\mathbb{E}_{p_{\text{data}}(z|x)} (\log p_\theta(x|z)) - D_{\text{KL}}(p_{\text{data}}(z|x) \| p(z)) \right] \\ &= \mathcal{L}(p_{\text{data}}(x)) - \mathbb{E}_{p_{\text{data}}(x)} [\text{ELBO}(x|\theta)] \end{aligned}$$

where

$$\text{ELBO}(x|\theta) = \mathbb{E}_{p_{\text{data}}(z|x)} (\log p_\theta(x|z)) - D_{\text{KL}}(p_{\text{data}}(z|x) \| p(z))$$

This form is used in computation. We will interpret it later.

Thus, the ELBO has two equivalent forms:

$$\text{ELBO}(x|\theta) = \log p_\theta(x) - D_{\text{KL}}(p_{\text{data}}(z|x) \| p_\theta(z|x)) \quad (\text{Form 1})$$

$$= \mathbb{E}_{p_{\text{data}}(z|x)} [\log p_\theta(x|z)] - D_{\text{KL}}(p_{\text{data}}(z|x) \| p(z)) \quad (\text{Form 2})$$

Form 1 (Conceptual):

- Shows ELBO as log-likelihood minus inference gap
- Reveals why it’s a lower bound: $D_{\text{KL}} \geq 0$
- Not computable because:
 - $\log p_\theta(x)$ requires intractable integration
 - $p_\theta(z|x)$ requires Bayes’ rule: $p_\theta(z|x) = \frac{p_\theta(x|z)p(z)}{p_\theta(x)}$
- Provides theoretical understanding of the bound’s tightness

Form 2 (Computational):

- Involves only tractable components:
 - $p_\theta(x|z)$: decoder network
 - $p(z)$: prior (e.g., $\mathcal{N}(0, I)$)
 - $p_{\text{data}}(z|x)$: inference model (to be learned)
- Can be directly optimized
- Suggests practical network architecture
- Leads naturally to implementation strategy

This duality is crucial:

- Form 1 helps us understand what we're trying to achieve
- Form 2 shows us how to actually achieve it
- Together they bridge theory and practice

7.4.5 Analysis of Gaps

Inference Gap: The difference between model likelihood and ELBO

$$\begin{aligned}
 \mathcal{L}(\theta) - \text{ELBO}(\theta) &= \mathbb{E}_{p_{\text{data}}(x)}[\log p_\theta(x)] - \mathbb{E}_{p_{\text{data}}(x)}[\text{ELBO}(x|\theta)] \\
 &= \mathbb{E}_{p_{\text{data}}(x)}[\log p_\theta(x) - \text{ELBO}(x|\theta)] \\
 &= \mathbb{E}_{p_{\text{data}}(x)}[D_{\text{KL}}(p_{\text{data}}(z|x) \| p_\theta(z|x))]
 \end{aligned}$$

Model Gap: The difference between optimal and model likelihood

$$\begin{aligned}
 \mathcal{L}(p_{\text{data}}) - \mathcal{L}(\theta) &= \mathbb{E}_{p_{\text{data}}(x)}[\log p_{\text{data}}(x)] - \mathbb{E}_{p_{\text{data}}(x)}[\log p_\theta(x)] \\
 &= \mathbb{E}_{p_{\text{data}}(x)}[\log \frac{p_{\text{data}}(x)}{p_\theta(x)}] \\
 &= D_{\text{KL}}(p_{\text{data}}(x) \| p_\theta(x))
 \end{aligned}$$

Total Gap: The joint KL divergence

$$\mathcal{L}(p_{\text{data}}) - \text{ELBO}(\theta) = D_{\text{KL}}(p_{\text{data}}(x, z) \| p_\theta(x, z))$$

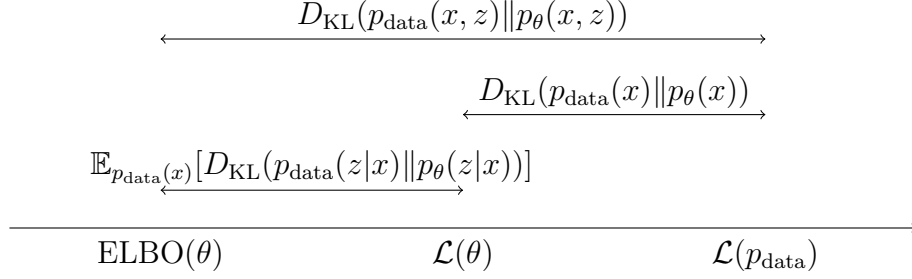


Figure 7.3: Three KL divergences measuring different gaps

7.5 Inference Model

7.5.1 From Data Augmentation to Learnable Inference

The joint KL divergence involves $p_{\text{data}}(z|x)$, which represents how latent variables are assigned to observed data. We can view this in two equivalent ways:

- As data augmentation: $p_{\text{data}}(z|x)$ augments each observation x with latent variables z
- As missing data imputation: $p_{\text{data}}(z|x)$ infers the missing latent variables for each x

Instead of fixing this conditional distribution, we can make it learnable:

$$p_{\text{data}}(z|x) \rightarrow q_{\phi}(z|x)$$

This leads to a more general joint KL divergence:

$$D_{\text{KL}}(p_{\text{data}}(x)q_{\phi}(z|x)||p(z)p_{\theta}(x|z))$$

7.5.2 Joint Optimization

The problem becomes a joint optimization over both the model θ and the inference model ϕ :

$$\min_{\theta, \phi} D_{\text{KL}}(p_{\text{data}}(x)q_{\phi}(z|x)||p(z)p_{\theta}(x|z))$$

This has several theoretical implications:

- The inference model $q_{\phi}(z|x)$ is learned along with the generative model $p_{\theta}(x|z)$
- Both models are treated symmetrically in the joint KL divergence
- The optimization naturally balances:
 - Quality of latent variable inference (through $q_{\phi}(z|x)$)
 - Quality of data generation (through $p_{\theta}(x|z)$)
- The joint KL directly gives the negative ELBO (up to constant $H(p_{\text{data}})$):

$$D_{\text{KL}}(p_{\text{data}}(x)q_{\phi}(z|x)||p(z)p_{\theta}(x|z)) = \mathcal{L}(p_{\text{data}}) - \mathbb{E}_{p_{\text{data}}(x)}[\text{ELBO}(x|\theta, \phi)]$$

7.5.3 Evidence Lower Bound with Learnable Inference

For each observation x , the ELBO with learnable inference model $q_\phi(z|x)$ has two equivalent forms:

Definition 36 (ELBO with Learnable Inference).

$$\text{ELBO}(x|\theta, \phi) = \log p_\theta(x) - D_{\text{KL}}(q_\phi(z|x) \| p_\theta(z|x)) \quad (\text{Form 1})$$

$$= \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{\text{KL}}(q_\phi(z|x) \| p(z)) \quad (\text{Form 2})$$

The full objective is:

$$\text{ELBO}(\theta, \phi) = \mathbb{E}_{p_{\text{data}}(x)}[\text{ELBO}(x|\theta, \phi)]$$

This gives us:

$$D_{\text{KL}}(p_{\text{data}}(x)q_\phi(z|x) \| p(z)p_\theta(x|z)) = \mathcal{L}(p_{\text{data}}) - \text{ELBO}(\theta, \phi)$$

Therefore, minimizing the joint KL divergence over both θ and ϕ is equivalent to maximizing the ELBO:

$$\max_{\theta, \phi} \text{ELBO}(\theta, \phi) \equiv \min_{\theta, \phi} D_{\text{KL}}(p_{\text{data}}(x)q_\phi(z|x) \| p(z)p_\theta(x|z))$$

7.5.4 Interpreting Form 1 of the ELBO

The first form of the ELBO directly shows its relationship to the log-likelihood:

$$\text{ELBO}(x|\theta, \phi) = \underbrace{\log p_\theta(x)}_{\text{log-likelihood}} - \underbrace{D_{\text{KL}}(q_\phi(z|x) \| p_\theta(z|x))}_{\text{inference gap}}$$

Since KL divergence is always non-negative, this form immediately reveals that ELBO is indeed a lower bound on the log-likelihood:

$$\text{ELBO}(x|\theta, \phi) \leq \log p_\theta(x)$$

with equality if and only if $q_\phi(z|x) = p_\theta(z|x)$.

This form provides distinct interpretations for optimizing ϕ and θ :

For the inference parameters ϕ , the objective is clear: we want the learned inference model $q_\phi(z|x)$ to approximate the true posterior $p_\theta(z|x)$ under the current generative model. This is achieved by minimizing the KL divergence term, which measures how far our inference model is from the true posterior.

For the generative parameters θ , the objective reveals an interesting interplay. The first term $\log p_\theta(x)$ encourages the model to assign high probability to the observed data. However, the second term creates an additional constraint: the model's posterior $p_\theta(z|x)$ should be close to the learned inference model $q_\phi(z|x)$. This means the generative model must “bend” itself to accommodate the inference model — it must not only explain the data well but do so in a way that makes the approximate inference accurate.

This dual optimization creates a cooperative learning dynamic: while the inference model tries to match the true posterior, the generative model simultaneously adjusts its structure to make this inference task easier. In other words, θ is optimized not just for data likelihood but also for inference accuracy, leading to models that are both powerful and amenable to approximate inference.

This interpretation highlights why the ELBO is such an effective objective for learning both generative and inference models. It naturally balances the competing goals of model accuracy (high likelihood) and inference quality (low KL divergence), leading to models that are both expressive and tractable. The fact that the generative model adapts itself to make inference easier is particularly important in practice, as it means we can learn models that are specifically structured to work well with our approximate inference procedures.

7.5.5 Interpreting Form 2 of the ELBO

The second form of the ELBO consists of two terms:

$$\text{ELBO}(x|\theta, \phi) = \underbrace{\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)]}_{\text{reconstruction term}} - \underbrace{D_{\text{KL}}(q_\phi(z|x)||p(z))}_{\text{regularization term}}$$

The inference model $q_\phi(z|x)$ can be interpreted as producing a “fuzzy” point estimate of the latent variable z conditioned on the observation x . Rather than outputting a single deterministic value, it provides a distribution over possible values of z . The two terms in Form 2 create competing objectives that balance each other:

The first term, $\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)]$, is the expected log-likelihood of reconstructing x given samples of z drawn from $q_\phi(z|x)$. This term encourages the inference model to produce values of z that can effectively explain or reconstruct the observation x through the generative model $p_\theta(x|z)$. Crucially, because $q_\phi(z|x)$ produces a distribution rather than a point estimate, $p_\theta(x|z)$ must learn to explain x for a range of possible z values rather than a single optimal z . If instead we were to use a deterministic point estimate for z , it could encode too much information about x , leaving $p_\theta(x|z)$ with little to learn. By maintaining uncertainty in z , we force $p_\theta(x|z)$ to learn meaningful patterns rather than relying on z to capture all the details of x .

The second term, $D_{\text{KL}}(q_\phi(z|x)||p(z))$, acts as a regularization term by measuring how much the inferred distribution $q_\phi(z|x)$ deviates from the prior $p(z)$. This KL divergence term encourages the inference model to maintain uncertainty and avoid overly concentrated or deterministic estimates. Importantly, this term can never become exactly zero unless $q_\phi(z|x)$ exactly matches the prior $p(z)$ for all x , which would make the inference model independent of the input and therefore useless. The non-zero KL divergence reflects the fundamental trade-off between maintaining uncertainty in the latent space while still extracting meaningful information from the observations.

This interpretation reveals how the ELBO naturally balances between two competing objectives: making the latent representations informative enough to enable accurate reconstruction while preventing them from becoming too specialized or deterministic. The regularization effect of the KL term is crucial for learning robust and generalizable representations, as it maintains a degree of “fuzziness” in the latent space that can help prevent

overfitting and enable better generalization. This fuzziness is not just a mathematical nicety but serves a crucial role in learning: it prevents the latent variables from becoming a mere lookup table for x and instead forces the model to learn meaningful, generalizable patterns in the data.

7.5.6 Mode Covering versus Mode Seeking Behavior

An important distinction exists between two different KL divergence minimizations in variational inference:

$$\min_{\theta} D_{\text{KL}}(p_{\text{data}}(x) \| p_{\theta}(x)) \quad \text{versus} \quad \min_{\phi} D_{\text{KL}}(q_{\phi}(z|x) \| p_{\theta}(z|x))$$

These objectives exhibit fundamentally different behaviors due to the asymmetric nature of KL divergence:

Model Learning: Mode Covering When minimizing $D_{\text{KL}}(p_{\text{data}}(x) \| p_{\theta}(x))$, the objective becomes:

$$\mathbb{E}_{p_{\text{data}}(x)} [\log p_{\text{data}}(x) - \log p_{\theta}(x)]$$

The crucial term $-\log p_{\theta}(x)$ becomes arbitrarily large when $p_{\theta}(x)$ approaches zero for any x where $p_{\text{data}}(x) > 0$. This leads to mode-covering behavior: the learned model $p_{\theta}(x)$ must assign non-negligible probability to all regions where the data distribution has mass, even if this means placing probability mass in regions between modes. This results in a model that may generate samples that don't look like real data points, as it tries to "cover" all modes of the data distribution.

Inference: Mode Seeking In contrast, when minimizing $D_{\text{KL}}(q_{\phi}(z|x) \| p_{\theta}(z|x))$, the objective becomes:

$$\mathbb{E}_{q_{\phi}(z|x)} [\log q_{\phi}(z|x) - \log p_{\theta}(z|x)]$$

Here, the expectation is taken with respect to $q_{\phi}(z|x)$. The inference model $q_{\phi}(z|x)$ can avoid regions where $p_{\theta}(z|x)$ is small by setting $q_{\phi}(z|x)$ to zero in those regions, incurring no penalty. This leads to mode-seeking behavior: $q_{\phi}(z|x)$ tends to concentrate around a single mode of $p_{\theta}(z|x)$, potentially ignoring other modes entirely. The inference model will typically underestimate the uncertainty in the posterior, producing overly confident predictions.

Implications This asymmetry has important practical implications:

- The generative model $p_{\theta}(x)$ will tend to produce "blurry" samples as it tries to cover all modes of the data distribution
- The inference model $q_{\phi}(z|x)$ will tend to be overconfident, potentially missing important alternative explanations for the data
- This tension is inherent in the variational framework and helps explain some common failure modes of VAEs, such as blurry reconstructions

Understanding these different behaviors is crucial for both model design and interpretation. For instance, if precise uncertainty quantification is important, one might need to use more expressive inference models or alternative divergence measures that better capture multimodal posteriors. Similarly, if sharp sample generation is desired, one might need to modify the generative objective to avoid the mode-covering behavior.

7.5.7 Connection to EM Algorithm

The EM algorithm corresponds to setting the inference model to the current posterior:

$$q_\phi(z|x) = p_{\theta_t}(z|x)$$

where θ_t is the parameter value at iteration t .

For each observation x , the ELBO takes two forms:

$$\begin{aligned} \text{ELBO}(x|\theta, \theta_t) &= \log p_\theta(x) - D_{\text{KL}}(p_{\theta_t}(z|x) \| p_\theta(z|x)) \\ &= \mathbb{E}_{p_{\theta_t}(z|x)}[\log p_\theta(x|z)] - D_{\text{KL}}(p_{\theta_t}(z|x) \| p(z)) \end{aligned}$$

Key properties:

- At $\theta = \theta_t$, the first KL term vanishes:

$$\text{ELBO}(x|\theta_t, \theta_t) = \log p_{\theta_t}(x)$$

- For any θ :

$$\log p_\theta(x) \geq \text{ELBO}(x|\theta, \theta_t)$$

- EM iterations guarantee monotonic improvement:

$$\begin{aligned} \log p_{\theta_{t+1}}(x) &\geq \text{ELBO}(x|\theta_{t+1}, \theta_t) \\ &\geq \text{ELBO}(x|\theta_t, \theta_t) \\ &= \log p_{\theta_t}(x) \end{aligned}$$

The algorithm proceeds as:

- E-step: Use current θ_t to compute posterior

$$p_{\theta_t}(z|x) = \frac{p_{\theta_t}(x|z)p(z)}{p_{\theta_t}(x)}$$

- M-step: Maximize second form of ELBO

$$\theta_{t+1} = \arg \max_{\theta} \mathbb{E}_{p_{\text{data}}(x)} \mathbb{E}_{p_{\theta_t}(z|x)} [\log p_\theta(x|z)]$$

Note that $D_{\text{KL}}(p_{\theta_t}(z|x) \| p(z))$ is constant w.r.t. θ

This shows:

- EM provides monotonic improvement via tight bound at θ_t
- The M-step only needs to optimize the expected complete log-likelihood
- The algorithm alternates between:
 - Making the bound tight (E-step)
 - Optimizing the tight bound (M-step)

VAE generalizes EM by:

- Allowing learned inference model independent of θ
- Trading bound tightness for tractability
- Enabling joint optimization of model and inference

7.6 Variational Autoencoder Implementation

7.6.1 Neural Network Parametrization

We implement both distributions using neural networks:

- Inference model (encoder) $q_\phi(z|x)$:

$$q_\phi(z|x) = \mathcal{N}(z; \mu_\phi(x), \text{diag}(\sigma_\phi^2(x)))$$

where $\mu_\phi(x)$ and $\sigma_\phi^2(x)$ are neural networks

- Generative model (decoder) $p_\theta(x|z)$:

- For continuous data:

$$p_\theta(x|z) = \mathcal{N}(x; \mu_\theta(z), \text{diag}(\sigma_\theta^2(z)))$$

- For binary data:

$$p_\theta(x|z) = \text{Bernoulli}(x; \mu_\theta(z))$$

- Prior remains fixed:

$$p(z) = \mathcal{N}(z; 0, I)$$

7.6.2 The Reparametrization Trick

To enable backpropagation through random sampling, we reparametrize the sampling process:

- Instead of directly sampling:

$$z \sim q_\phi(z|x) = \mathcal{N}(\mu_\phi(x), \text{diag}(\sigma_\phi^2(x)))$$

- We reparametrize using an auxiliary random variable:

$$\epsilon \sim \mathcal{N}(0, I)$$

$$z = \mu_\phi(x) + \sigma_\phi(x) \odot \epsilon$$

- This makes the sampling process differentiable:
 - ϵ provides randomness
 - $\mu_\phi(x)$ and $\sigma_\phi(x)$ are differentiable transforms
 - \odot denotes element-wise multiplication

7.6.3 Computing the ELBO

For a single observation x , we can now compute:

$$\begin{aligned} \text{ELBO}(x|\theta, \phi) &= \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{\text{KL}}(q_\phi(z|x) \| p(z)) \\ &= \mathbb{E}_{\epsilon \sim \mathcal{N}(0, I)}[\log p_\theta(x|\mu_\phi(x) + \sigma_\phi(x) \odot \epsilon)] - \\ &\quad \frac{1}{2} \sum_{j=1}^d (1 + \log \sigma_{\phi,j}^2(x) - \mu_{\phi,j}^2(x) - \sigma_{\phi,j}^2(x)) \end{aligned}$$

where:

- First term is estimated using Monte Carlo sampling
- KL term has closed form for Gaussian distributions
- d is the dimension of the latent space

7.6.4 Training Algorithm

For each minibatch:

1. Sample x from training data
2. Sample $\epsilon \sim \mathcal{N}(0, I)$

3. Compute $z = \mu_\phi(x) + \sigma_\phi(x) \odot \epsilon$
4. Compute ELBO estimate using:
 - Reconstruction term: $\log p_\theta(x|z)$
 - KL term (closed form)
5. Update θ and ϕ using gradient ascent

7.6.5 Practical Considerations

- KL annealing: Gradually increase weight of KL term during training

$$\mathcal{L}_\beta = \mathbb{E}[\log p_\theta(x|z)] - \beta D_{\text{KL}}(q_\phi(z|x) \| p(z))$$

- Reconstruction scaling: Balance terms for high-dimensional x

$$\mathcal{L}_\alpha = \alpha \mathbb{E}[\log p_\theta(x|z)] - D_{\text{KL}}(q_\phi(z|x) \| p(z))$$

- Multiple samples: Reduce variance using more ϵ samples

$$\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] \approx \frac{1}{L} \sum_{l=1}^L \log p_\theta(x|z^{(l)})$$

7.6.6 Generation and Reconstruction

After training:

- Generation: Sample $z \sim \mathcal{N}(0, I)$, then compute $\mu_\theta(z)$
- Reconstruction: Compute $\mu_\phi(x)$, then compute $\mu_\theta(\mu_\phi(x))$
- Interpolation: Interpolate in z space between encodings of two images

7.7 Comparison with Diffusion Models

7.7.1 Latent Variable Structure

VAE and diffusion models can be unified under the same latent variable framework:

- VAE:
 - Single latent vector: $z \in \mathbb{R}^d$
 - Joint distribution: $p_\theta(x, z) = p(z)p_\theta(x|z)$
 - Requires learned inference: $q_\phi(z|x)$
- Diffusion:
 - Sequence of latents: $z = (x_1, \dots, x_T)$ where $x_0 = x$
 - Forward process: $q(x_t|x_{t-1})$ is fixed Gaussian
 - Reverse process: $p_\theta(x_{t-1}|x_t)$ is learned

7.7.2 Key Distinction: Fixed vs Learned Inference

The fundamental difference lies in inference:

- VAE:
 - Must learn inference model $q_\phi(z|x)$
 - Optimizes joint KL: $D_{\text{KL}}(p_{\text{data}}(x)q_\phi(z|x)||p(z)p_\theta(x|z))$
 - Challenging optimization over both θ and ϕ
- Diffusion:
 - Forward process $q(x_t|x_{t-1})$ is fixed
 - Results in supervised learning problem
 - No variational inference needed

7.7.3 Theoretical Guarantees

Diffusion models offer stronger theoretical guarantees:

- Prior approximation:
 - VAE: Assumes fixed $\mathcal{N}(0, I)$ prior
 - Diffusion: x_T provably converges to $\mathcal{N}(0, I)$ as $T \rightarrow \infty$
- Transition probabilities:
 - VAE: Complex, learned $p_\theta(x|z)$
 - Diffusion: $p_\theta(x_{t-1}|x_t)$ provably approaches Gaussian for small noise
- ELBO tightness:
 - VAE: Gap depends on quality of learned $q_\phi(z|x)$
 - Diffusion: Tighter bounds due to:
 - * Gaussian x_T (exact prior)
 - * Near-Gaussian transitions
 - * No learned variational approximation

7.7.4 Philosophical Perspective

While diffusion models take the mathematical form of VAEs:

- They discard the core VAE principle of learned inference
- Replace variational learning with supervised learning

- Trade flexibility of learned inference for:
 - Theoretical guarantees
 - Easier optimization
 - Tighter bounds

This reveals a fundamental insight:

Diffusion models succeed not by embracing VAE principles, but by carefully constructing a scenario where they can be avoided while maintaining the generative capability.

7.8 Generative Adversarial Networks

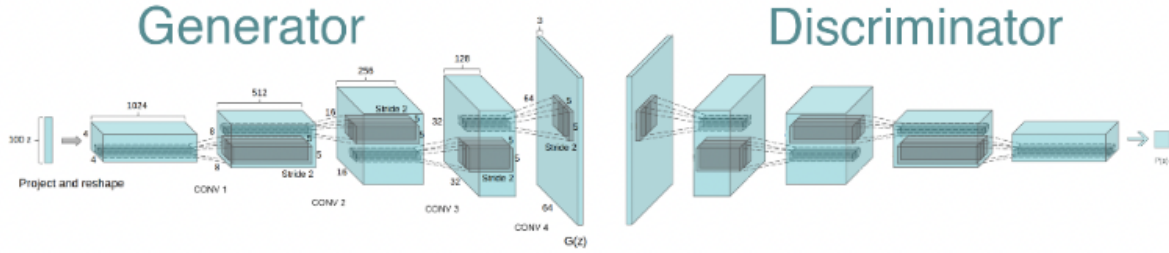


Figure 7.4: GAN

7.8.1 Data Structure

Consider a training dataset structured as follows:

Source	Data	Label	Size
Real data	$x_i \sim p_{\text{data}}(x)$	$y_i = 1$	$i = 1, \dots, n$
Generated data	$\tilde{x}_i = G(z_i), z_i \sim \mathcal{N}(0, I)$	$\tilde{y}_i = 0$	$i = 1, \dots, n$

Table 7.1: Binary classification structure of GAN

7.8.2 Learning the Discriminator

The discriminator $D(x)$ models $p(y = 1|x)$, the probability that x is real. The log-likelihood for D is:

$$\begin{aligned}\mathcal{L}(D) &= \sum_{i=1}^n [\log D(x_i) + \log(1 - D(\tilde{x}_i))] \\ &= \sum_{i=1}^n [\log D(x_i) + \log(1 - D(G(z_i)))]\end{aligned}$$

As $n \rightarrow \infty$, this approaches:

$$\begin{aligned}\mathcal{L}(D) &= \mathbb{E}_{x \sim p_{\text{data}}} [\log D(x)] + \mathbb{E}_{z \sim \mathcal{N}(0, I)} [\log(1 - D(G(z)))] \\ &= \mathbb{E}_{x \sim p_{\text{data}}} [\log D(x)] + \mathbb{E}_{\tilde{x} \sim p_G} [\log(1 - D(\tilde{x}))]\end{aligned}$$

where p_G is the distribution induced by $G(z)$ with $z \sim \mathcal{N}(0, I)$.

7.8.3 Game-Theoretic Perspective

This log-likelihood becomes a value function $V(D, G)$ in a zero-sum game:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{\text{data}}} [\log D(x)] + \mathbb{E}_{z \sim \mathcal{N}(0, I)} [\log(1 - D(G(z)))]$$

where:

- Discriminator D aims to maximize $V(D, G)$:
 - Increase $D(x)$ for real data
 - Decrease $D(G(z))$ for generated data
- Generator G aims to minimize $V(D, G)$:
 - Make $D(G(z))$ close to 1
 - Make generated samples indistinguishable from real data

This formulation:

- Turns generative modeling into a binary classification game
- Does not require explicit density estimation
- Leads to an adversarial training dynamic

7.8.4 Implementation Form

In practice, minimizing $\log(1 - D(G(z)))$ provides weak gradients early in training when D easily rejects poor samples. Instead, we use the equivalent form:

- For discriminator D , maximize:

$$\mathcal{L}_D = \mathbb{E}_{x \sim p_{\text{data}}} [\log D(x)] + \mathbb{E}_{z \sim \mathcal{N}(0, I)} [\log(1 - D(G(z)))]$$

- For generator G , maximize:

$$\mathcal{L}_G = \mathbb{E}_{z \sim \mathcal{N}(0, I)} [\log D(G(z))]$$

Training algorithm:

1. For each iteration:
 - Sample minibatch of real data $\{x_i\}_{i=1}^m$
 - Sample minibatch of noise $\{z_i\}_{i=1}^m$
 - Update D using gradient ascent on \mathcal{L}_D
 - Sample new noise $\{z_i\}_{i=1}^m$
 - Update G using gradient ascent on \mathcal{L}_G

7.8.5 Wasserstein GAN

W-GAN reframes the generator-discriminator relationship into an actor-critic setup:

- Critic $f(x)$:
 - Assigns real-valued scores to images
 - Higher scores for "better" (more realistic) images
 - No sigmoid constraint (unlike GAN's discriminator)
 - Relationship to GAN: $D(x) = \text{sigmoid}(f(x))$
- Actor $G(z)$:
 - Tries to generate images that receive high scores from critic
 - Gets clearer feedback through continuous scores
 - No longer needs to "fool" a binary classifier

- Objective:

$$\min_G \max_{f \in \mathcal{F}_L} \mathbb{E}_{x \sim p_{\text{data}}} [f(x)] - \mathbb{E}_{z \sim \mathcal{N}(0, I)} [f(G(z))]$$

where \mathcal{F}_L is the set of 1-Lipschitz functions

- Interpretation:

- Critic tries to assign high scores to real images, low scores to generated ones
- Generator tries to create images that receive high critic scores
- Lipschitz constraint ensures scores don't become arbitrarily large
- More like a rating system than a binary real/fake classifier

This actor-critic perspective:

- Provides more informative learning signal than binary classification
- Makes the relationship between generator and critic more collaborative
- Better reflects the continuous nature of image quality
- Helps explain why W-GAN training is more stable

7.8.6 Mode Collapse

Mode collapse represents a significant challenge in GAN training where the generator G learns to map different latent vectors z to a limited subset of the data distribution's modes, while ignoring others. This phenomenon can be understood as follows:

- Problem Definition:
 - Generator G maps Gaussian noise $z \sim \mathcal{N}(0, I)$ to only a few major modes of $p_{\text{data}}(x)$
 - Minor modes in the true data distribution are systematically ignored
 - Multiple different inputs z_1, z_2, \dots, z_k may map to the same or very similar outputs

- Mathematical Perspective:

$$p_G(x) \ll p_{\text{data}}(x) \text{ for minor modes}$$

$$p_G(x) \approx p_{\text{data}}(x) \text{ for a few major modes}$$

- Causes:
 - Discriminator optimization may focus on major modes first
 - Generator can maximize $\mathbb{E}_{z \sim \mathcal{N}(0, I)} [\log D(G(z))]$ by concentrating on these modes
 - No explicit penalty for lack of diversity in generated samples

Consider a simple example where $p_{\text{data}}(x)$ is a mixture of Gaussians:

$$p_{\text{data}}(x) = \sum_{i=1}^k \pi_i \mathcal{N}(\mu_i, \Sigma_i)$$

Mode collapse occurs when G learns to generate samples primarily from components with larger mixing coefficients π_i , while ignoring components with smaller coefficients.

Several approaches have been proposed to address mode collapse:

- Minibatch discrimination: Compare generated samples within a minibatch
- Unrolled GANs: Look ahead several discriminator steps
- Multiple discriminators: Provide diverse feedback signals
- Modified objectives: Include terms that explicitly encourage diversity

The severity of mode collapse can be measured through metrics such as:

- Coverage: Fraction of true modes captured by G
- Quality-diversity tradeoff: Relationship between sample quality and diversity
- Birthday paradox test: Detecting duplicate samples in generated data

This challenge highlights a fundamental tension in GAN training between quality and diversity of generated samples.

Chapter 8

Deep Reinforcement Learning

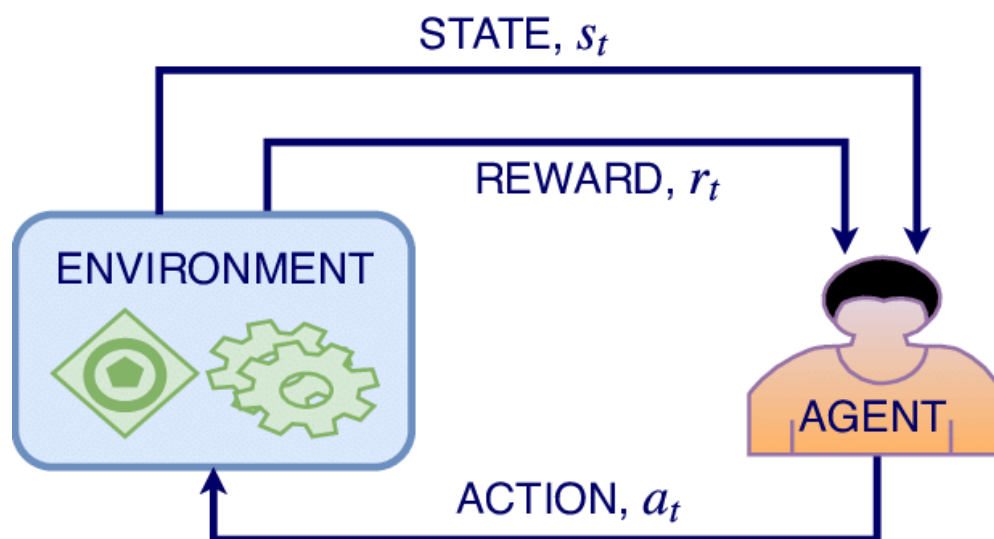


Figure 8.1: Reinforcement Learning

Chapter Overview

This chapter examines deep reinforcement learning through systematic comparisons of fundamental paradigms, using AlphaGo and Atari game mastery as illustrative case studies. These domains serve as ideal counterpoints to highlight key dichotomies in reinforcement learning approaches:

- **Model-Based vs Model-Free:** AlphaGo leverages Go's perfect model for Monte Carlo Tree Search, while Atari agents learn directly from experience without explicit modeling. This contrast illuminates when and why model-based planning proves advantageous over pure model-free learning.

- **Value-Based vs Policy-Based Methods:** DQN’s success in Atari demonstrates value-based learning with dense rewards, while policy gradient methods like PPO show alternative advantages in policy space exploration. AlphaGo’s hybrid approach, combining both paradigms, reveals their complementary strengths.
- **On-Policy vs Off-Policy Learning:** The trade-offs between on-policy methods (PPO) and off-policy approaches (DQN) highlight fundamental tensions between sample efficiency and stability. These differences manifest distinctly in both domains, informing algorithm choice.
- **Dense vs Sparse Rewards:** Atari’s frequent feedback enables direct bootstrapping and temporal difference learning, while Go’s sparse terminal rewards necessitate sophisticated planning and value estimation. This reward structure profoundly impacts algorithm design and effectiveness.
- **Policy vs Planning:** Pure policy methods (DQN, PPO) contrast with planning-centric approaches (MCTS), revealing how environment characteristics influence the balance between reactive policies and explicit planning.

Through these comparisons, we explore:

- How environment properties guide algorithm selection
- When to prefer different learning paradigms
- Trade-offs between computational complexity and performance
- Integration strategies combining multiple approaches

Modern frameworks like MuZero increasingly blur these distinctions, suggesting a unified view where these apparent dichotomies represent different regions in a continuous space of algorithms. This comparative lens provides deeper insight into reinforcement learning’s fundamental principles and their practical manifestation in different domains.

8.1 Theoretical Foundations of Sequential Decision Making

Since we already have some exposure to reinforcement learning in Chapter 5, where we studied reinforcement learning from human feedback (RLHF), we shall begin this chapter with theoretical foundation of Markov decision process (MDP). Readers without any prior exposure to RL is encouraged to read Chapter 5 on RLHF first, or read about AlphaGo and Atari first, before reading this section.

8.1.1 Basic Setup

Core components of an MDP:

- State space S : Set of all possible states
- Action space A : Set of all possible actions
- Transition model $P(s'|s, a)$: Dynamic model
- Reward model $R(s, a, s')$: Immediate reward
- Discount factor $\gamma \in [0, 1]$: Future reward weighting

8.1.2 Key Functions

Policy

A policy π maps states to actions:

- Deterministic: $\pi : S \rightarrow A$
- Stochastic: $\pi(a|s)$ probability distribution over A

Value Functions

Two equivalent perspectives:

- State-value function:

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s \right] \quad (8.1)$$

- Action-value function:

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a \right] \quad (8.2)$$

Optimal Functions

$$V^*(s) = \max_{\pi} V^\pi(s) \quad (8.3)$$

$$Q^*(s, a) = \max_{\pi} Q^\pi(s, a) \quad (8.4)$$

$$\pi^*(s) = \arg \max_a Q^*(s, a) \quad (8.5)$$

The goal of RL is to find the optimal policies.

8.1.3 Model-Based vs Model-Free Paradigms

Model-Based Approach

When transition model is known/learned:

- Can simulate future states: $s' \sim P(\cdot|s, a)$
- Enables planning and trajectory optimization
- Examples: MCTS, MPC, Dynamic Programming

Model-Free Approach

When model is unknown/complex:

- Learn directly from experience
- No explicit model required
- Examples: Q-learning, Policy Gradient

8.2 Fundamental Theorems in Reinforcement Learning

8.2.1 Policy Gradient Theorem

Setup

Objective function:

$$J(\theta) = \mathbb{E}_{\pi_\theta} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right] \quad (8.6)$$

We aim to prove:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(a|s) Q^{\pi_\theta}(s, a)] \quad (8.7)$$

Proof

Start with state distribution $\rho^\pi(s)$:

$$J(\theta) = \sum_s \rho^\pi(s) \sum_a \pi_\theta(a|s) Q^{\pi_\theta}(s, a) \quad (8.8)$$

Taking gradient:

$$\nabla_\theta J(\theta) = \sum_s \rho^\pi(s) \sum_a [\nabla_\theta \pi_\theta(a|s) Q^{\pi_\theta}(s, a) + \pi_\theta(a|s) \nabla_\theta Q^{\pi_\theta}(s, a)] \quad (8.9)$$

$$= \sum_s \rho^\pi(s) \sum_a \pi_\theta(a|s) \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} Q^{\pi_\theta}(s, a) + \text{second term} \quad (8.10)$$

Key insight: Second term sums to zero due to compatible value function approximation. Using $\nabla_{\theta} \log \pi_{\theta}(a|s) = \frac{\nabla_{\theta} \pi_{\theta}(a|s)}{\pi_{\theta}(a|s)}$:

$$\nabla_{\theta} J(\theta) = \sum_s \rho^{\pi}(s) \sum_a \pi_{\theta}(a|s) \nabla_{\theta} \log \pi_{\theta}(a|s) Q^{\pi_{\theta}}(s, a) \quad (8.11)$$

Which gives our expectation form:

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\pi_{\theta}} [\nabla_{\theta} \log \pi_{\theta}(a|s) Q^{\pi_{\theta}}(s, a)] \quad (8.12)$$

8.2.2 Fundamental Relationships in Value-Based RL

Return and its Recursive Structure

The total return (reward-to-go) from time t is:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \dots \quad (8.13)$$

This infinite sum has a fundamental recursive structure. At its simplest:

$$R_t = r_t + \gamma R_{t+1} \quad (8.14)$$

More generally, for any horizon m :

$$R_t = r_t + \gamma r_{t+1} + \dots + \gamma^{m-1} r_{t+m-1} + \gamma^m R_{t+m} \quad (8.15)$$

This recursive structure is the foundation for bootstrap-based methods in RL.

Value Functions and Their Definitions

For a policy π , we define two types of value functions:

$$v_{\pi}(s) = \mathbb{E}_{\pi}[R_t | s_t = s] \quad (8.16)$$

$$Q_{\pi}(s, a) = \mathbb{E}_{\pi}[R_t | s_t = s, a_t = a] \quad (8.17)$$

These expectations incorporate:

- Future rewards under policy π
- State transition dynamics $P(s'|s, a)$
- Discount factor γ

State-Action Value Relationships

Value functions under policy π are related:

v_{π} in terms of Q_{π} :

$$v_{\pi}(s) = \sum_a \pi(a|s) Q_{\pi}(s, a) \quad (8.18)$$

This averages Q -values over policy's action choices.

Q_{π} in terms of v_{π} :

$$Q_{\pi}(s, a) = r(s, a) + \gamma \sum_{s'} P(s'|s, a) v_{\pi}(s') \quad (8.19)$$

This shows immediate reward plus discounted future value.

Bellman Equations for Policy Evaluation

Combining these relationships yields the Bellman equation for v_π :

$$v_\pi(s) = \sum_a \pi(a|s) \left[r(s, a) + \gamma \sum_{s'} P(s'|s, a) v_\pi(s') \right] \quad (8.20)$$

And for Q_π :

$$Q_\pi(s, a) = r(s, a) + \gamma \sum_{s'} P(s'|s, a) \sum_{a'} \pi(a'|s') Q_\pi(s', a') \quad (8.21)$$

Optimal Value Functions

The optimal value functions are defined as:

$$v^*(s) = \max_\pi v_\pi(s) \quad (8.22)$$

$$Q^*(s, a) = \max_\pi Q_\pi(s, a) \quad (8.23)$$

These are related by:

$$v^*(s) = \max_a Q^*(s, a) \quad (8.24)$$

$$Q^*(s, a) = r(s, a) + \gamma \sum_{s'} P(s'|s, a) v^*(s') \quad (8.25)$$

Bellman Optimality Equation

Combining these yields the Bellman optimality equation:

$$Q^*(s, a) = r(s, a) + \gamma \sum_{s'} P(s'|s, a) \max_{a'} Q^*(s', a') \quad (8.26)$$

Define the Bellman optimality operator \mathcal{T} :

$$(\mathcal{T}Q)(s, a) = r(s, a) + \gamma \sum_{s'} P(s'|s, a) \max_{a'} Q(s', a') \quad (8.27)$$

Fixed Point Analysis

For any Q_1, Q_2 :

$$\|\mathcal{T}Q_1 - \mathcal{T}Q_2\|_\infty \leq \gamma \|Q_1 - Q_2\|_\infty \quad (8.28)$$

This contraction mapping property implies:

- Unique fixed point Q^*
- Value iteration convergence: $\|\mathcal{T}^n Q_0 - Q^*\|_\infty \leq \gamma^n \|Q_0 - Q^*\|_\infty$
- Rate of convergence controlled by γ

Broader Implications

These relationships underpin major RL algorithms:

- **Policy Evaluation:** Uses Bellman equation for v_π
- **Q-Learning:** Approximates Bellman optimality
- **Actor-Critic:** Leverages both v_π and Q_π
- **MCTS:** Uses recursive structure of returns

Understanding these relationships helps explain why:

- Temporal difference methods work
- Bootstrap methods are efficient
- Value iteration converges
- Function approximation can be challenging

8.2.3 Implications

These theorems underpin:

- Policy Gradient: Direct policy optimization
- Q-Learning: Value iteration convergence
- Actor-Critic: Combines both insights

8.2.4 Core Algorithm Derivations

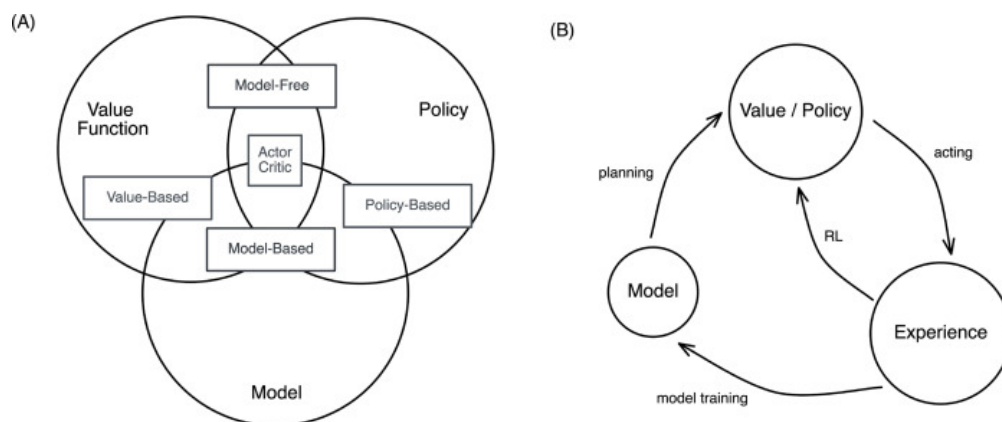


Figure 8.2: RL algorithms

Policy Gradient

Objective: Maximize expected return

$$J(\theta) = \mathbb{E}_{\pi_\theta} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right] \quad (8.29)$$

Policy Gradient Theorem:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(a|s) Q^{\pi_\theta}(s, a)] \quad (8.30)$$

Please refer to Chapter 5 RLHF part for a detailed comparison between policy gradient learning and maximum likelihood learning (imitation learning).

Q-Learning

Based on Bellman optimality:

$$Q^*(s, a) = R(s, a) + \gamma \mathbb{E}_{s'} \left[\max_{a'} Q^*(s', a') \right] \quad (8.31)$$

Update rule:

$$Q_{k+1}(s, a) = (1 - \alpha) Q_k(s, a) + \alpha \left[r + \gamma \max_{a'} Q_k(s', a') \right] \quad (8.32)$$

Model Predictive Control

Optimize H-step trajectory:

$$\max_{a_t, \dots, a_{t+H-1}} \sum_{k=0}^{H-1} \gamma^k r(s_{t+k}, a_{t+k}) \quad (8.33)$$

subject to:

$$s_{t+k+1} = f(s_{t+k}, a_{t+k}) \quad (8.34)$$

8.2.5 Advanced Methods

Actor-Critic

Combines policy gradient with value estimation:

- Actor: Updates policy using critic's value

$$\Delta\theta \propto \nabla_\theta \log \pi_\theta(a|s) A(s, a) \quad (8.35)$$

- Critic: Estimates advantage function

$$A(s, a) = Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s) \quad (8.36)$$

Bootstrap Principle

Key idea: Update estimates using other estimates

- Temporal Difference (TD) Learning:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_t + \gamma V(s_{t+1}) - V(s_t)] \quad (8.37)$$

- n -step returns:

$$G_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n V(s_{t+n}) \quad (8.38)$$

- Trade-off between bias and variance

This theoretical foundation explains why:

- AlphaGo combines model-based planning (MCTS) with learned values
- Atari agents use model-free methods with bootstrapping
- MPC relies on explicit model and receding horizon

8.3 The Game of Go

8.3.1 Game Complexity

Go is an ancient board game that has challenged human intellect for over 2,500 years. Played on a 19×19 grid, it presents a computational challenge of remarkable scale. The game tree complexity of Go is estimated to be approximately 10^{170} , far exceeding that of chess (10^{120}). This vast search space made Go particularly resistant to traditional game-playing algorithms that had succeeded in other domains.

8.3.2 Formal Game Definition

The game of Go can be formally defined as follows:

- **Board State** (s): A state s represents a configuration of black and white stones on the 19×19 board, along with the game history required by the ko rule. The state space includes all legal board positions.
- **Legal Actions** (a): At each state s , a player can either place a stone at any empty intersection (subject to the rules of Go) or pass. We denote the set of legal actions at state s as $A(s)$.
- **Terminal States**: A game terminates when both players pass consecutively. At this point, the reward z is computed.
- **Reward** (z): The terminal reward z is $+1$ for a black win, -1 for a white win, and 0 for a draw, from black's perspective. The winner is determined by counting territory and captured stones according to Chinese rules.



Figure 8.3: Alpha Go

8.3.3 Rules and Gameplay

Players alternate placing black and white stones on board intersections. The objective is to surround and capture opponent stones while securing territory. Key rules include:

1. Stones must be placed on empty intersections
2. Groups of stones must maintain at least one liberty (adjacent empty intersection)
3. The ko rule prevents immediate repetition of board positions
4. Captured stones are removed from the board

This combination of simple rules and vast complexity made Go an ideal challenge for advancing artificial intelligence beyond traditional game-playing approaches.

8.4 Neural Network Architecture

8.4.1 Policy Network

The policy network $p_{\sigma}(a|s)$ outputs a probability distribution over all legal moves a in position s . It is parameterized by weights σ and consists of:

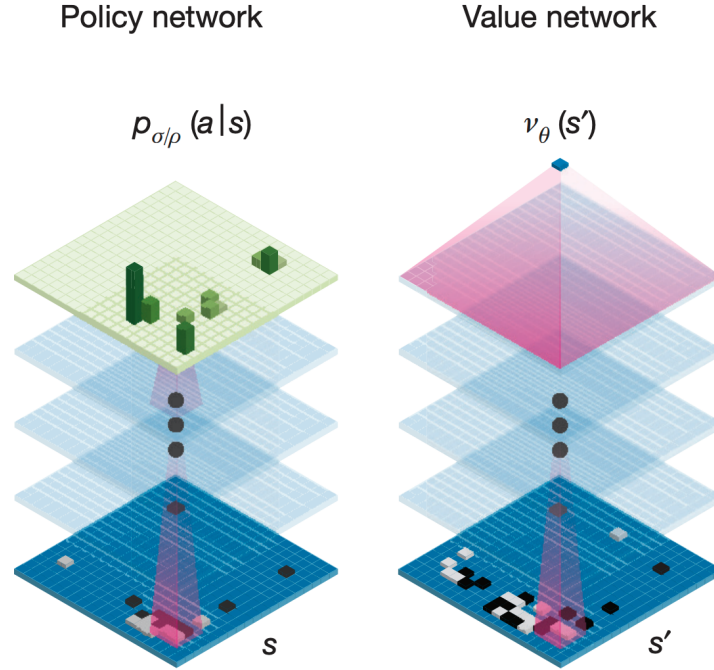


Figure 8.4: Policy and value networks

- Input: $19 \times 19 \times 48$ image stack representing current and historical board positions
- Architecture: 13-layer convolutional neural network
- Output: Probability distribution over all $19 \times 19 + 1$ possible moves (including pass)

8.4.2 Value Network

The value network $v_{\theta}(s)$ outputs a scalar value estimating the expected outcome from position s . It is parameterized by weights θ and shares a similar architecture:

- Input: Same $19 \times 19 \times 48$ image stack as policy network
- Architecture: Similar convolutional structure to policy network
- Output: Scalar value predicting expected outcome $\in [-1, 1]$

8.5 Training Methodology

8.5.1 Supervised Learning of Policy Network

The policy network was initially trained on expert human moves using stochastic gradient descent to maximize the likelihood of the expert move a played in state s :

$$\Delta\sigma \propto \frac{\partial \log p_\sigma(a|s)}{\partial \sigma} \quad (8.39)$$

This resulted in the supervised learning (SL) policy network p_σ . A faster but less accurate rollout policy p_π was also trained similarly for use during Monte Carlo tree search.

8.5.2 Reinforcement Learning of Policy Network

The policy network was further improved through self-play reinforcement learning. Games were played between the current policy network and random previous iterations. The weights were updated to maximize the expected outcome:

$$\Delta\rho \propto \frac{\partial \log p_\rho(a_t|s_t)}{\partial \rho} z_t \quad (8.40)$$

where z_t is the terminal reward at the end of the game. This policy gradient approach led to the reinforcement learning (RL) policy network, which played more strongly than the SL policy network.

8.5.3 Training the Value Network

The value network was trained using a novel approach to improve accuracy:

1. Generate self-play games using the RL policy network
2. For each position s , record the terminal reward z from player's perspective
3. Train value network by regression to minimize mean squared error:

$$\Delta\theta \propto \frac{\partial (z - v_\theta(s))^2}{\partial \theta} \quad (8.41)$$

This approach differs from traditional reinforcement learning value networks in two key ways:

- Uses self-play positions from strong policy network
- Regresses towards actual game outcomes rather than bootstrapped estimates

The resulting value network provides more accurate position evaluation than Monte Carlo rollouts, especially in complex tactical situations where reading to the end of the game is infeasible.

8.6 Progressive Introduction to Monte Carlo Tree Search

8.6.1 From Simple Policy to Look-ahead Search

At any given state s_0 , we have multiple ways to select an action:

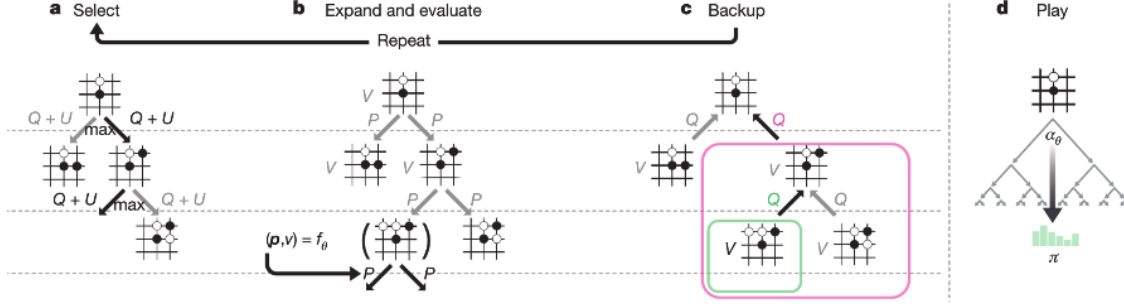


Figure 8.5: Monte Carlo Tree Search

Direct Policy Approach

The simplest approach is to directly sample from the learned policy:

$$a_0 \sim p(a_0|s_0) \quad (8.42)$$

One-Step Value Maximization

Alternatively, we can choose the action that maximizes the immediate next state's value:

$$a_0 = \arg \max_{a_0} v(s_1) \quad (8.43)$$

where s_1 follows from taking action a_0 in state s_0 .

8.6.2 Basic Monte Carlo Look-ahead

We can improve upon these simple approaches by looking deeper into the future through Monte Carlo sampling. Here's a basic version that illustrates the core concept:

Forward Simulation

For a fixed depth $T = 20$:

1. Start from state s_0
2. Sample action sequence using policy:

$$a_t \sim p(a_t|s_t) \quad \text{for } t = 0, 1, \dots, 19 \quad (8.44)$$

3. Generate corresponding states:

$$s_{t+1} = f(s_t, a_t) \quad \text{for } t = 0, 1, \dots, 19 \quad (8.45)$$

4. Evaluate final state using value network:

$$v(s_{20}) \quad (8.46)$$

Q-Value Estimation

Repeat the forward simulation N times. For each first action a_0 , compute:

$$Q(s_0, a_0) = \mathbb{E}[v(s_{20})|s_0, a_0] \quad (8.47)$$

where the expectation is approximated by averaging over all simulations that started with a_0 .

Action Selection

In actual play, select the action with highest estimated Q-value:

$$a_0^* = \arg \max_{a_0} Q(s_0, a_0) \quad (8.48)$$

8.6.3 Advantage of Looking Ahead

This simple Monte Carlo approach already illustrates a key insight: evaluating deeper positions (s_{20}) provides more reliable information than evaluating immediate positions (s_1). This occurs for two fundamental reasons:

Future Sight Advantage

When evaluating $v(s_{20})$ versus $v(s_1)$:

- $v(s_1)$ must implicitly predict the next 19 moves
- $v(s_{20})$ sees their actual realization
- Tactical sequences that were uncertain at s_1 have played out in s_{20}

Averaging Advantage

The Q-value computation:

- Aggregates many evaluations of $v(s_{20})$ for each a_0
- Reduces variance through averaging
- Provides more robust assessment than single evaluation
- Captures different possible developments from each a_0

8.6.4 Foundation for Full MCTS

This simple version lays the groundwork for understanding full MCTS by introducing:

- The concept of Q-values from forward simulation
- The advantage of deeper evaluation
- The role of Monte Carlo averaging
- The basic loop of:
 - Forward simulation
 - Terminal evaluation
 - Backward value aggregation

The full MCTS algorithm, which we'll examine next, builds upon these concepts by adding:

- Adaptive action selection during simulation
- Tree structure to store statistics
- Value backpropagation through the tree
- Balance between exploration and exploitation

8.6.5 Q-value Update on the Whole Branch

Building upon our basic look-ahead search, we now track statistics for all state-action pairs along each simulation path, not just at the root state.

Statistics Tracking

For each state-action pair (s, a) encountered in our simulations, we maintain:

- $N(s, a)$: Visit count for this state-action pair
- $W(s, a)$: Cumulative value from all visits
- $Q(s, a) = \frac{W(s, a)}{N(s, a)}$: Average value estimate

Simulation Process

Each simulation consists of:

1. **Forward Pass:**

- Start from s_0
- Sample actions: $a_t \sim p(a_t|s_t)$ for $t = 0, \dots, 19$
- Generate states: $s_{t+1} = f(s_t, a_t)$
- Store path: $(s_0, a_0), (s_1, a_1), \dots, (s_{19}, a_{19})$

2. **Leaf Evaluation:**

$$v_{leaf} = v(s_{20}) \quad (8.49)$$

3. **Backward Update:** For each (s_t, a_t) in the path, update:

$$N(s_t, a_t) \leftarrow N(s_t, a_t) + 1 \quad (8.50)$$

$$W(s_t, a_t) \leftarrow W(s_t, a_t) + v_{leaf} \quad (8.51)$$

$$Q(s_t, a_t) \leftarrow \frac{W(s_t, a_t)}{N(s_t, a_t)} \quad (8.52)$$

Progressive Refinement

This approach provides several advantages:

• **Multi-level Evaluation:**

- Q-values computed at all depths
- Statistics improve with more simulations
- Different actions explored at each state

• **Dynamic Updates:**

- Q-values refine over time
- More visited paths get better estimates
- Natural balance of exploration/exploitation

• **Information Reuse:**

- Same leaf evaluation updates multiple Q-values
- Common subsequences benefit from multiple paths
- Efficient use of each simulation

Example Update

Consider a single simulation path:

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \cdots \xrightarrow{a_{19}} s_{20} \quad (8.53)$$

After evaluating $v(s_{20}) = v_{leaf}$, we update:

$$\begin{aligned} &\text{For } t = 19 \text{ down to } 0 : \\ &N(s_t, a_t) \leftarrow N(s_t, a_t) + 1 \\ &W(s_t, a_t) \leftarrow W(s_t, a_t) + v_{leaf} \\ &Q(s_t, a_t) \leftarrow W(s_t, a_t)/N(s_t, a_t) \end{aligned}$$

Comparison with Basic Version

Improvements over the simpler approach:

- **Information Capture:**
 - Before: Only root Q-values stored
 - Now: Q-values at all depths
 - Better use of each simulation
- **Statistical Accuracy:**
 - Before: Separate averages for each a_0
 - Now: Integrated statistics across tree
 - More efficient value estimation
- **Search Structure:**
 - Before: Independent root simulations
 - Now: Building blocks for tree search
 - Foundation for guided exploration

This progressive refinement of Q-values sets the stage for the final MCTS version, where we'll introduce:

- UCT-style action selection
- Policy-guided exploration
- Tree structure maintenance
- Adaptive simulation strategies

8.6.6 Policy-Guided Action Selection

We now improve our 20-step lookahead by making action selection more sophisticated. Instead of purely sampling from the policy $p(a|s)$, we select actions that balance exploiting our accumulated knowledge (Q-values) with exploring promising actions suggested by the policy.

Selection Formula

At each state s_t during the simulation, select action:

$$a_t = \arg \max_a \left(Q(s_t, a) + cp(a|s_t) \frac{\sqrt{\sum_b N(s_t, b)}}{1 + N(s_t, a)} \right) \quad (8.54)$$

where:

- $Q(s_t, a)$: Exploitation term (using accumulated knowledge)
- $p(a|s_t)$: Policy prior (guiding exploration)
- $\frac{\sqrt{\sum_b N(s_t, b)}}{1 + N(s_t, a)}$: Visit count bonus term
- c : Exploration constant (typically around 1.0 to 5.0)

Term Analysis

- **Exploitation Term** $Q(s_t, a)$:
 - Represents accumulated knowledge
 - Favors actions that have worked well
 - More reliable as $N(s_t, a)$ increases
 - Drives convergence to best actions
- **Exploration Term** $cp(a|s_t) \frac{\sqrt{\sum_b N(s_t, b)}}{1 + N(s_t, a)}$:
 - Encourages trying promising actions
 - Decays with increasing visits
 - Weighted by policy prior $p(a|s_t)$
 - Scales with total visits $\sqrt{\sum_b N(s_t, b)}$

Dynamic Balance

The formula automatically adjusts exploration-exploitation balance:

- **Early in Search:**
 - Small $N(s_t, a)$ values
 - Exploration term dominates
 - Actions selected mainly based on policy
 - Broad exploration of promising paths
- **Later in Search:**
 - Larger $N(s_t, a)$ values
 - Q-values become more reliable
 - Exploitation term gains importance
 - Focus narrows to best-performing actions

Role of Policy Prior

The learned policy $p(a|s_t)$ serves multiple purposes:

- Guides initial exploration
- Focuses search on promising actions
- Reduces effective branching factor
- Particularly valuable in large action spaces

Visit Count Scaling

The $\sqrt{\sum_b N(s_t, b)}$ term provides important properties:

- Increases exploration bonus with more total visits
- Ensures continued exploration of alternatives
- Balances exploration across different tree depths
- Theoretically motivated by UCT analysis

Example Scenarios

Consider a state s_t with two actions:

Early in search:

- Action 1: $Q = 0.0$, $N = 1$, $p = 0.8$
- Action 2: $Q = 0.0$, $N = 1$, $p = 0.2$
- Total visits = 2
- Exploration term dominates, Action 1 preferred due to higher policy probability

Later in search:

- Action 1: $Q = 0.3$, $N = 50$, $p = 0.8$
- Action 2: $Q = 0.6$, $N = 20$, $p = 0.2$
- Total visits = 70
- Q-values more reliable, Action 2 may be preferred despite lower policy probability

Implementation Benefits

This selection strategy provides several advantages:

- Automatic exploration-exploitation trade-off
- Efficient use of learned policy knowledge
- Theoretical guarantees from UCT framework
- Natural transition from exploration to exploitation
- Scalable to large action spaces

This improved action selection prepares us for the final step of full MCTS, where we'll add:

- Explicit tree structure
- Node expansion criteria
- Adaptive simulation depth
- More sophisticated backup strategies

8.6.7 Full MCTS with Dynamic Tree Growth

We now remove the fixed 20-step constraint and allow the tree to grow naturally through node expansion. This leads to the complete MCTS algorithm that adapts its search depth based on the most promising paths.

Tree Structure

Each node in the tree stores:

- State s
- Visit statistics for all actions a :
 - $N(s, a)$: visit count
 - $W(s, a)$: cumulative value
 - $Q(s, a) = W(s, a)/N(s, a)$: average value
- Children nodes (if expanded)

MCTS Steps

Each simulation consists of four phases:

1. Selection:

- Start at root node s_0
- While at expanded node s_t , select action:

$$a_t = \arg \max_a \left(Q(s_t, a) + cp(a|s_t) \frac{\sqrt{\sum_b N(s_t, b)}}{1 + N(s_t, a)} \right) \quad (8.55)$$

- Continue until reaching leaf node s_L

2. Expansion:

- For leaf node s_L , create child nodes for all legal actions, these child nodes will become leaf nodes in the next round of selection
- Initialize statistics:

$$N(s_L, a) = 0 \quad (8.56)$$

$$W(s_L, a) = 0 \quad (8.57)$$

$$Q(s_L, a) = 0 \quad (8.58)$$

3. Evaluation:

- Evaluate leaf position using value network:

$$v_{leaf} = v_{\theta}(s_L) \quad (8.59)$$

4. Backup:

- For each state-action pair (s_t, a_t) on path to root:

$$N(s_t, a_t) \leftarrow N(s_t, a_t) + 1 \quad (8.60)$$

$$W(s_t, a_t) \leftarrow W(s_t, a_t) + v_{leaf} \quad (8.61)$$

$$Q(s_t, a_t) \leftarrow W(s_t, a_t)/N(s_t, a_t) \quad (8.62)$$

Advantages Over Fixed-Depth Search

- **Adaptive Depth:**
 - Tree grows deeper along promising paths
 - Natural allocation of computation
 - No artificial depth limit
 - Better handling of variable-length sequences
- **Efficient Memory Use:**
 - Only stores nodes that have been visited
 - Memory grows with actual search effort
 - Focuses resources on relevant parts of tree
 - Avoids exponential memory growth
- **Better Value Estimation:**
 - Values backed up from various depths
 - Naturally weights different horizons
 - Combines shallow and deep evaluations
 - More robust position assessment

Tree Growth Properties

The tree expands asymmetrically:

- **Promising Paths:**
 - Higher visit counts
 - Deeper expansion
 - More refined value estimates
- **Unpromising Paths:**
 - Lower visit counts
 - Shallower expansion
 - Less computational investment

Comparison with Fixed-Depth Version

Key differences:

- **Depth:**

$$\text{Depth} = \begin{cases} 20 & \text{Fixed version} \\ \text{Variable} & \text{Full MCTS} \end{cases} \quad (8.63)$$

- **Memory:**

$$\text{Nodes} = \begin{cases} O(20 \times |\mathcal{A}|) & \text{Fixed version} \\ O(N_{\text{sim}} \times \text{avg_depth}) & \text{Full MCTS} \end{cases} \quad (8.64)$$

- **Value Estimation:**

$$v_{\text{leaf}} = \begin{cases} v_{\theta}(s_{20}) & \text{Fixed version} \\ v_{\theta}(s_L) & \text{Full MCTS} \end{cases} \quad (8.65)$$

This full version of MCTS represents the algorithm as used in AlphaGo and similar systems. The natural tree growth and adaptive search depth make it a powerful approach for finding strong moves in complex games like Go.

8.6.8 Complementary Roles of Policy and Value for Search

The policy and value networks play distinct but complementary roles in controlling the MCTS search space:

Policy Network: Reducing Search Breadth

The policy network $p(a|s)$ helps control horizontal expansion:

- **Without Policy:**

- Must consider all legal moves equally
- Branching factor = full legal move set (≈ 250 in Go)
- Search breadth grows exponentially
- Most computation wasted on poor moves

- **With Policy:**

- Focus search on promising moves
- Effectively reduces branching factor
- $p(a|s)$ guides exploration term:

$$cp(a|s) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)} \quad (8.66)$$

- Computation concentrated on relevant variations

Value Network: Reducing Search Depth

The value network $v(s)$ helps control vertical expansion:

- **Without Value:**
 - Must search to game termination
 - Need deep rollouts for position evaluation
 - Search depth = remaining game length
 - Impractical in games like Go
- **With Value:**
 - Can evaluate leaf nodes directly
 - Shallower search trees suffice
 - More accurate than rollout results
 - Better handling of tactical positions

Combined Effect

Together, these networks dramatically reduce the search space:

$$\text{Search Space} = \text{Breadth}^{\text{Depth}} \xrightarrow{\text{Policy, Value}} (\text{Effective Breadth})^{\text{Effective Depth}} \quad (8.67)$$

Where:

- Policy reduces effective breadth from ≈ 250 to perhaps ≈ 50
- Value reduces effective depth from ≈ 150 to perhaps ≈ 20
- Combined reduction: $250^{150} \rightarrow 50^{20}$
- Makes deep strategic search feasible

Search Quality

The reduction in search space actually improves search quality:

- **Policy Impact:**
 - More visits to relevant moves
 - Better statistics for key variations
 - Reduced noise from irrelevant moves
- **Value Impact:**
 - More accurate leaf evaluation

- Better handling of tactical positions
- Reduced variance compared to rollouts

This efficient division of labor between policy and value networks is a key reason for MCTS's success in complex games like Go, where the raw search space is astronomically large. By controlling both breadth and depth of search, these networks make it possible to perform meaningful strategic planning in otherwise intractable game trees.

8.6.9 Value Network and Bootstrap Principle

The value network enables powerful bootstrapping in MCTS, which is crucial for reducing search depth while maintaining evaluation accuracy.

Basic Bootstrap Concept Value bootstrapping means:

- Each state value builds on future state values
- No need to search to terminal states
- Chain of value estimates:

$$v(s_t) \approx r_t + \gamma v(s_{t+1}) \quad (8.68)$$

Traditional MCTS Without Value Network Without bootstrapping:

- Must play out to game end
- Value only from terminal states z
- Long rollouts required
- High variance in estimates
- Computation grows with game length

MCTS With Value Network With bootstrapping:

- Can stop at leaf node s_L
- Use value estimate $v(s_L)$
- Short search trees suffice
- Lower variance estimates
- Computation independent of game length

Bootstrap Chain Effect Creates a chain of increasingly reliable estimates:

$$Q(s_0, a_0) \leftarrow r_0 + \gamma Q(s_1, a_1) \leftarrow r_1 + \gamma Q(s_2, a_2) \leftarrow \cdots \leftarrow r_L + \gamma v(s_L) \quad (8.69)$$

Each step in the chain:

- Incorporates actual game dynamics
- Accumulates real intermediate rewards
- Benefits from search at future nodes
- Terminates with learned value estimate

Why Bootstrap Works Bootstrap is effective because:

- Value network trained on full game outcomes
- Each tree node aggregates many leaf evaluations
- Multiple simulations reduce estimation variance
- Search refines raw value predictions
- Shorter paths have less compounded error

Implementation Benefits This leads to practical advantages:

- Fixed-depth search trees possible
- Predictable computation per move
- Better tactical evaluation
- More simulations in same time
- Efficient memory usage

The bootstrap principle transforms MCTS from a terminal-state-focused search to an efficient, bounded-depth procedure that can still capture long-term strategic considerations through the value network’s learned knowledge.

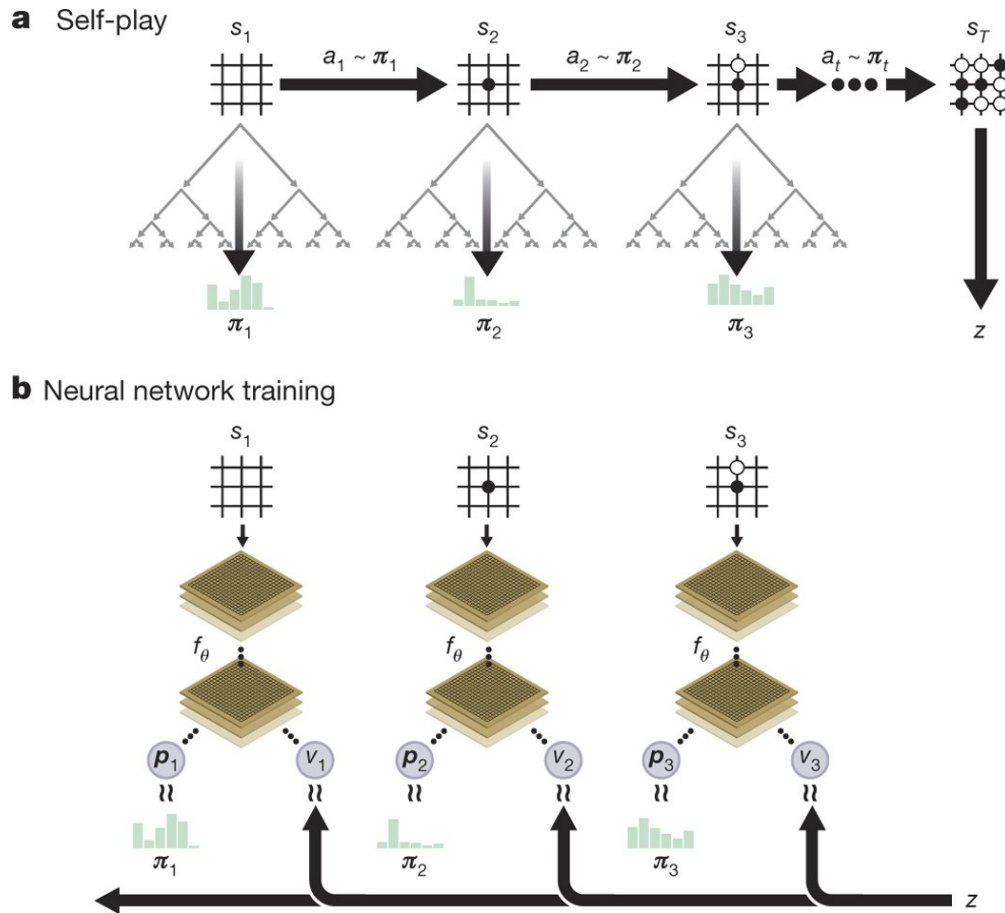


Figure 8.6: AlphaGo Zero

8.7 From AlphaGo to AlphaGo Zero

8.7.1 Original AlphaGo Architecture

Training in London

The original AlphaGo system had two phases in different locations:

- **In London (Training Phase):**
 - Train policy network π_θ on human expert games
 - Train value network v_θ using self-play games
 - Freeze the networks once training is complete
- **In Seoul (Playing Phase):**
 - Use the fixed networks from London
 - Combine them with MCTS for actual play
 - No further training or updates

8.7.2 The Key Insight

Looking at this setup, a crucial realization emerges:

- MCTS with the networks plays better than the raw networks
- This suggests a natural improvement:
 - Why not use MCTS self-play games to train the networks?
 - The stronger play from MCTS could provide better training data
 - This could be done right in London, no need for separate locations

8.7.3 The Natural Evolution

This insight leads to a progression of ideas:

Step 1: Use MCTS Self-Play

Instead of just human games:

- Let MCTS play against itself
- Use these games to train policy network:

$$\pi_{\theta} \leftarrow \text{moves chosen by MCTS} \quad (8.70)$$

dwq21

- Use game outcomes to train value network:

$$v_{\theta} \leftarrow \text{actual game results} \quad (8.71)$$

Step 2: Remove Human Knowledge

Then comes the radical thought:

- Why start with human games at all?
- Could start from random networks
- Let MCTS compensate for initial poor play
- Use game outcomes as ground truth

8.7.4 Birth of AlphaGo Zero

This evolution naturally leads to AlphaGo Zero:

- Start with random networks
- Use MCTS to play games
- Train networks on MCTS games
- Repeat, creating a learning cycle:

$$\text{Random} \xrightarrow{\text{MCTS}} \text{Better Play} \xrightarrow{\text{Training}} \text{Better Networks} \xrightarrow{\text{MCTS}} \text{Even Better Play} \quad (8.72)$$

8.7.5 Why This Works

The system can bootstrap from zero because:

- MCTS provides improvement over raw network performance
- Game outcomes give reliable training signals
- Each component helps improve the others:
 - Better value network \rightarrow better MCTS evaluation
 - Better MCTS \rightarrow finds stronger moves
 - Stronger moves \rightarrow better training data
 - Better training data \rightarrow improved networks

This progression from original AlphaGo to AlphaGo Zero shows how the desire to improve an existing system, combined with some key insights, can lead to a simpler yet more powerful approach. The "Zero" in AlphaGo Zero represents not just the absence of human knowledge, but the elegance of learning everything from first principles through self-play.

8.8 Reflections: System 1 and System 2

AlphaGo Zero's architecture provides remarkable insights into the nature of intelligence and learning, particularly when viewed through the lens of dual-process theory of cognition.

8.8.1 System 1 and System 2 in AlphaGo Zero

MCTS as System 2

Monte Carlo Tree Search embodies characteristics of System 2 thinking:

- Deliberate, step-by-step planning
- Explicit consideration of alternatives

- Resource-intensive computation
- Conscious-like sequential reasoning
- Look-ahead simulation of consequences

Neural Networks as System 1

The policy and value networks mirror System 1 characteristics:

- Fast, intuitive responses
- Pattern-based recognition
- Low computational overhead
- Subconscious-like immediate judgments
- Learned from experience

8.8.2 The Consciousness Parallel

Planning as Conscious Thought

MCTS exhibits key features of conscious processing:

- Sequential, one-step-at-a-time analysis
- Explicit representation of possibilities
- Working memory-like tree structure
- Deliberate evaluation of options
- Awareness-like focus of computation

Fast vs Slow Thinking

The system naturally implements Kahneman's two speeds:

- **Fast (System 1):**
 - Neural networks provide immediate evaluations
 - Policy network suggests moves instantly
 - Value network gives quick position assessments
- **Slow (System 2):**
 - MCTS performs careful search
 - Explicitly considers move sequences
 - Accumulates evidence through simulation

8.8.3 Learning as Memorization

System 1 Learning

A key insight emerges about neural network training:

- Networks primarily learn to memorize results of MCTS
- No need for complex RL algorithms
- Simple supervised learning suffices
- Goal is to memorize System 2's conclusions
- Policy gradient/PPO not actually necessary

Symbiotic Relationship

The two systems support each other:

$$\text{System 2 (MCTS)} \xrightarrow{\text{provides training data}} \text{System 1 (Networks)} \xrightarrow{\text{guides search}} \text{System 2 (MCTS)} \quad (8.73)$$

8.8.4 Primacy of Planning

Planning vs Reinforcement Learning

This analysis suggests a profound insight:

- Planning, not RL, is the key to intelligence
- Representation should be learned to facilitate planning
- RL could be replaced by simple distillation and memorization
- Planning provides ground truth for learning
- Planning is more fundamental than policy optimization

Evolutionary Implications

This suggests a compelling evolutionary story:

- Evolution needed to discover only two core algorithms:
 - A universal planning mechanism (likely gradient descent)
 - A simple learning algorithm (also gradient descent)
- Planning provides the conscious workspace
- Learning distills planning results into fast responses
- Combination creates powerful general intelligence

8.8.5 Generalization and Transfer

Planning Algorithm Generality

Planning has broad applicability:

- Works across diverse domains
- Requires only basic environment model
- Adapts automatically to new situations
- More generalizable than learned policies

Learning Algorithm Simplicity

Neural network training can be straightforward:

- Simple supervised learning from planning
- No complex RL machinery needed
- Focus on memorization and distillation
- Supports but doesn't replace planning

This perspective suggests that AlphaGo Zero's architecture might be more than just a good design for playing Go - it might reflect fundamental principles about the nature of intelligence, consciousness, and learning. The distinction between planning-based System 2 and learning-based System 1, along with their symbiotic relationship, could be a blueprint for understanding both biological and artificial intelligence.

8.9 Deep Q-Learning for Atari Games

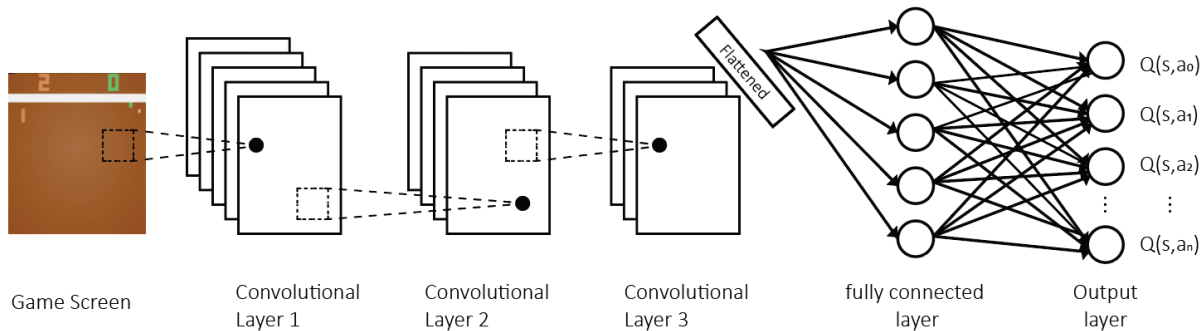


Figure 8.7: Deep Q learning for Atari

8.9.1 The Atari Environment

Unlike Go, Atari games present a distinctly different MDP structure:

- State (s): Raw pixel frames (typically 84×84 grayscale)
- Actions (A): Discrete joystick movements (typically 4-18 actions)
- Reward (r): Game score changes (dense and intermediate)
- Transition: Frame-to-frame dynamics (deterministic)

8.9.2 Q-Learning Formulation

Q-learning aims to learn the optimal action-value function:

$$Q^*(s, a) = \mathbb{E}[r + \gamma \max_{a'} Q^*(s', a')] \quad (8.74)$$

In Deep Q-Network (DQN), we approximate Q^* using a neural network $Q_\theta(s, a)$ with parameters θ .

8.9.3 Key Components

Experience Replay

To break correlation between consecutive samples:

- Store transitions (s_t, a_t, r_t, s_{t+1}) in replay buffer D
- Sample random minibatches for training
- Buffer size typically 1 million transitions

Target Network

To reduce moving target problem:

- Maintain separate target network Q_{θ^-} with parameters θ^-
- Update θ^- to θ every C steps (e.g., $C = 10,000$)
- Use θ^- for computing target values

8.9.4 Training Process

For each step:

1. Select action using ϵ -greedy policy:

$$a_t = \begin{cases} \text{random action} & \text{with probability } \epsilon \\ \arg \max_a Q_\theta(s_t, a) & \text{otherwise} \end{cases} \quad (8.75)$$

2. Execute action, observe reward r_t and next state s_{t+1}
3. Store transition (s_t, a_t, r_t, s_{t+1}) in D
4. Sample random minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from D
5. Compute target values:

$$y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} Q_{\theta^-}(s_{j+1}, a') & \text{otherwise} \end{cases} \quad (8.76)$$

6. Update θ by gradient descent on loss:

$$L(\theta) = \mathbb{E}_{(s,a,r,s') \sim D} [(y - Q_\theta(s, a))^2] \quad (8.77)$$

8.9.5 Contrast with AlphaGo

Key differences from AlphaGo include:

- No search component (pure Q-learning)
- Single-step updates vs game outcome
- Experience replay vs self-play games
- ϵ -greedy exploration vs PUCT
- Direct Q-value learning vs policy-value network

8.9.6 Practical Considerations

Critical implementation details:

- Frame stacking: Input 4 consecutive frames
- Reward clipping: Clip to $[-1, 1]$ range
- Gradient clipping: Prevent exploding gradients
- Decreasing ϵ schedule: $1.0 \rightarrow 0.1$ over first million frames

8.9.7 Q-Learning and MCTS: Shared Principles

Having examined Q-values and bootstrapping in MCTS, we can now see striking parallels with Q-learning, despite their different applications and implementations.

Q-Value Similarities

Both methods maintain state-action values $Q(s, a)$:

- **MCTS Q-values:**
 - $Q(s, a) = W(s, a)/N(s, a)$
 - Averages over tree search results
 - Updated within each search tree
 - Temporary, discarded after move selection
- **DQN Q-values:**
 - $Q(s, a)$ from neural network
 - Learned from experience replay
 - Updated through gradient descent
 - Permanent, reused across episodes

Bootstrap Principle

Both methods use bootstrapping for value estimation:

- **MCTS Bootstrap:**

$$Q(s_t, a_t) \leftarrow \text{average of } [v(s_L)] \quad (8.78)$$

where s_L is a leaf node and averaging is over multiple simulations

- **Q-Learning Bootstrap:**

$$Q(s_t, a_t) \leftarrow r_t + \gamma \max_{a'} Q(s_{t+1}, a') \quad (8.79)$$

Key Differences

Despite these similarities, important differences exist:

- **Planning vs Learning:**
 - MCTS: Plans ahead using known model
 - DQN: Learns from past experience
- **Value Sources:**
 - MCTS: Value network $v(s_L)$ at leaves

- DQN: Actual rewards plus bootstrapped future values
- **Update Scope:**
 - MCTS: Local to current search tree
 - DQN: Global across all experiences
- **Exploration Strategy:**
 - MCTS: Policy-guided UCT selection
 - DQN: ϵ -greedy or other direct exploration

Complementary Strengths

Each approach has advantages in different contexts:

- **MCTS Advantages:**
 - Better when model available
 - Can look ahead explicitly
 - More focused exploration
 - Natural handling of large action spaces
- **DQN Advantages:**
 - Works without environment model
 - Learns from actual experience
 - Reuses past knowledge
 - Efficient with dense rewards

Understanding these connections and differences helps appreciate how similar principles can manifest in different algorithms. While MCTS and DQN might appear very different at first glance, they share fundamental ideas about value estimation and bootstrapping, adapted to their respective domains and requirements.

8.10 Policy Gradient Methods for Atari Games

Please refer to Chapter 5 RLHF part for a detailed introduction and explanation of policy gradient.

8.10.1 Core Idea

Unlike Q-learning which learns action-values, policy gradient directly optimizes the policy:

- Policy $\pi_\theta(a|s)$: Neural network outputs action probabilities
- Objective: Expected sum of rewards $J(\theta) = \mathbb{E}_{\pi_\theta}[\sum_t r_t]$
- Update: Follow gradient of $J(\theta)$ to improve policy

8.10.2 Policy Gradient Theorem

The key theoretical result states:

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\pi_{\theta}} [\nabla_{\theta} \log \pi_{\theta}(a_t | s_t) Q^{\pi_{\theta}}(s_t, a_t)] \quad (8.80)$$

Leading to the update rule:

$$\theta \leftarrow \theta + \alpha \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) R_t \quad (8.81)$$

where $R_t = \sum_{k=t}^T r_k$ is the observed return.

8.10.3 REINFORCE Algorithm

Basic implementation:

1. Run policy π_{θ} for one episode:
 - Collect (s_t, a_t, r_t) for $t = 1, \dots, T$
 - Calculate returns $R_t = \sum_{k=t}^T r_k$

2. Update policy:

$$\theta \leftarrow \theta + \alpha \sum_t \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) R_t \quad (8.82)$$

8.10.4 Variance Reduction

Key improvements to reduce variance:

Baseline Subtraction

Use advantage instead of raw returns:

$$A(s_t, a_t) = R_t - V(s_t) \quad (8.83)$$

where $V(s_t)$ is learned state-value function (baseline).

Actor-Critic Architecture

- Actor: Policy network $\pi_{\theta}(a|s)$
- Critic: Value network $V_w(s)$
- Update rules:

$$\begin{aligned} \theta &\leftarrow \theta + \alpha_{\theta} \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) A(s_t, a_t) \\ w &\leftarrow w + \alpha_w \nabla_w (R_t - V_w(s_t))^2 \end{aligned}$$

8.10.5 Practical Implementation

For Atari games:

1. **Network Architecture:**

- Convolutional layers process $84 \times 84 \times 4$ input frames
- Split into policy head (softmax over actions) and value head (scalar)

2. **Training Process:**

- Collect fixed-length trajectories (e.g., 128 steps)
- Compute advantages using Generalized Advantage Estimation (GAE):

$$A^{GAE}(s_t, a_t) = \sum_{l=0}^{\infty} (\gamma \lambda)^l \delta_{t+l} \quad (8.84)$$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$

- Update both policy and value networks

8.10.6 Comparison with Q-Learning

Advantages of policy gradient:

- Better handles continuous action spaces
- Can learn stochastic policies
- More stable learning in many cases
- Natural extension to actor-critic methods

Challenges:

- Higher variance in updates
- Sensitive to hyperparameter choices
- Often requires more samples
- Can converge to local optima

8.11 Value-Based versus Policy-Based Methods

After examining both Q-learning and policy gradient approaches, we can now systematically compare these two fundamental paradigms in reinforcement learning. This comparison helps understand why actor-critic methods, which we will discuss next, aim to combine the advantages of both approaches.

8.11.1 Fundamental Differences

Core Learning Target

The two approaches differ in their primary learning objective:

- **Value-Based Methods:**

- Learn action-value function $Q(s, a)$
- Policy is implicit: $\pi(s) = \arg \max_a Q(s, a)$
- Optimize through Bellman equation:

$$Q(s, a) = \mathbb{E}[r + \gamma \max_{a'} Q(s', a')] \quad (8.85)$$

- **Policy-Based Methods:**

- Learn policy $\pi_\theta(a|s)$ directly
- No explicit value function required
- Optimize expected return:

$$J(\theta) = \mathbb{E}_{\pi_\theta} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right] \quad (8.86)$$

8.11.2 Key Properties

Action Space Handling

Different capabilities in action space:

- **Value-Based:**

- Natural for discrete actions
- Challenging for continuous actions (requires discretization)
- Must evaluate all actions for max operation
- Harder to represent mixed strategies

- **Policy-Based:**

- Works for both discrete and continuous actions
- Can learn stochastic policies
- Natural parameterization of continuous actions
- Efficient in large/infinite action spaces

8.11.3 Learning Characteristics

Sample Efficiency

Efficiency comparison:

- **Value-Based:**
 - Generally more sample efficient
 - Can reuse samples through experience replay
 - Better bootstrapping through value estimates
 - More efficient with dense rewards
- **Policy-Based:**
 - Generally less sample efficient
 - Often requires on-policy samples
 - Higher variance in gradient estimates
 - Better with sparse rewards

Convergence Properties

Different convergence characteristics:

- **Value-Based:**
 - Can oscillate due to max operator
 - Susceptible to overestimation bias
 - Guaranteed convergence in tabular case
 - May be unstable with function approximation
- **Policy-Based:**
 - Generally more stable learning
 - Converges to local optimum
 - Gradient estimates may have high variance
 - Better theoretical guarantees with approximation

8.11.4 Implementation Aspects

Memory Requirements

Storage needs:

$$\text{Memory} = \begin{cases} O(|\mathcal{S}| \times |\mathcal{A}|) & \text{Value-Based (table)} \\ O(|\theta|) & \text{Policy-Based (parameters)} \end{cases} \quad (8.87)$$

Computational Complexity

Action selection cost:

$$\text{Computation} = \begin{cases} O(|\mathcal{A}|) & \text{Value-Based (max operation)} \\ O(1) & \text{Policy-Based (direct output)} \end{cases} \quad (8.88)$$

8.11.5 Practical Trade-offs**When to Use Value-Based Methods**

Preferred conditions:

- Discrete, manageable action space
- Dense reward structure
- Sample efficiency is priority
- Off-policy learning desired
- Deterministic optimal policy exists

When to Use Policy-Based Methods

Favorable scenarios:

- Continuous or large action space
- Stochastic policy required
- Sparse reward structure
- Stability is priority over sample efficiency
- Natural gradient updates desired

8.11.6 Empirical Results in Atari**DQN Performance**

Value-based characteristics:

- Strong performance in games with clear action values
- Efficient learning of game mechanics
- Good at exploitation of learned strategies
- May struggle with exploration-heavy games

Policy Gradient Performance

Policy-based characteristics:

- Better exploration through stochastic policies
- More stable learning curves
- Often slower to reach peak performance
- Superior in games requiring mixed strategies

8.11.7 Motivation for Hybrid Approaches

The complementary strengths and weaknesses of these approaches naturally motivate hybrid methods:

- **Value-Based Strengths:**

- Sample efficiency
- Better bootstrapping
- Experience replay
- Strong exploitation

- **Policy-Based Strengths:**

- Stability
- Continuous actions
- Stochastic policies
- Better exploration

This motivates actor-critic methods, which we will examine next, as they attempt to combine the advantages of both approaches while mitigating their respective weaknesses. The actor-critic architecture represents a natural synthesis of these two fundamental paradigms, leading to more robust and flexible algorithms.

8.12 Actor-Critic Methods for Atari Games

Actor-critic methods represent a powerful hybrid approach that combines the advantages of both value-based and policy-based methods. For Atari games, these methods have proven particularly effective due to their ability to handle large discrete action spaces while maintaining stable learning.

8.12.1 Core Architecture

The actor-critic architecture consists of two main components:

Actor (Policy Network)

The actor $\pi_\theta(a|s)$ directly learns the policy:

- Takes game frames as input (typically $84 \times 84 \times 4$)
- Outputs action probabilities for each possible Atari action
- Updated to maximize expected advantage:

$$\max_{\theta} \mathbb{E}[\log \pi_{\theta}(a|s) A(s, a)] \quad (8.89)$$

Critic (Value Network)

The critic $V_{\phi}(s)$ learns state values:

- Shares convolutional layers with actor
- Outputs scalar state value estimate
- Updated to minimize TD error:

$$\min_{\phi} \mathbb{E}[(r + \gamma V_{\phi}(s') - V_{\phi}(s))^2] \quad (8.90)$$

8.12.2 Advantage Estimation

A crucial component is accurate advantage estimation:

One-Step Advantage

The simplest form uses one-step TD error:

$$A(s_t, a_t) = r_t + \gamma V_{\phi}(s_{t+1}) - V_{\phi}(s_t) \quad (8.91)$$

Generalized Advantage Estimation (GAE)

GAE provides better advantage estimates through exponentially-weighted TD errors:

$$A^{GAE}(s_t, a_t) = \sum_{l=0}^{\infty} (\gamma \lambda)^l \delta_{t+l} \quad (8.92)$$

where:

$$\delta_t = r_t + \gamma V_{\phi}(s_{t+1}) - V_{\phi}(s_t) \quad (8.93)$$

The parameter $\lambda \in [0, 1]$ controls the bias-variance trade-off:

- $\lambda = 0$: One-step TD (low variance, high bias)
- $\lambda = 1$: Monte Carlo (high variance, low bias)
- Typical values: $\lambda = 0.95$

8.12.3 Implementation for Atari

Network Architecture

Shared convolutional backbone:

- Input: $84 \times 84 \times 4$ stacked frames
- Conv layers: Same as DQN architecture
- Split into policy and value heads
- Policy head: Softmax over actions
- Value head: Single scalar output

Training Process

For each iteration:

1. Collect trajectories using current policy
2. Compute advantages using GAE
3. Update actor (policy) network:

$$\theta \leftarrow \theta + \alpha_{\theta} \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) A^{GAE}(s_t, a_t) \quad (8.94)$$

4. Update critic (value) network:

$$\phi \leftarrow \phi + \alpha_{\phi} \nabla_{\phi} (G_t - V_{\phi}(s_t))^2 \quad (8.95)$$

where G_t is the discounted sum of rewards

8.12.4 Key Advantages for Atari

Stability Improvements

Actor-critic methods provide several stability benefits:

- Reduced variance through value bootstrapping
- Shared feature learning between policy and value
- Natural curriculum through value estimation
- Stable credit assignment through GAE

Sample Efficiency

Better sample efficiency through:

- Value-guided policy updates
- Efficient advantage estimation
- Reuse of experience for both networks
- Faster learning of action preferences

8.12.5 Practical Considerations

Hyperparameters

Critical parameters include:

- Learning rates: α_θ (actor), α_ϕ (critic)
- GAE parameter: λ (typically 0.95)
- Discount: γ (typically 0.99)
- Value loss coefficient (balances losses)
- Batch size and optimization epochs

Implementation Tips

Key considerations:

- Normalize advantages batch-wise
- Use separate optimizers for actor and critic
- Clip gradient norms for stability
- Monitor value loss and policy entropy
- Consider frame stacking and reward scaling

8.12.6 Comparison to Other Methods

Versus Pure Policy Gradient

Advantages over policy gradient:

- Lower variance updates
- Better credit assignment
- Faster learning in early stages
- More stable optimization

Versus DQN

Advantages over DQN:

- Can learn stochastic policies
- No replay buffer required
- Better exploration through policy entropy
- More natural action probability outputs

Actor-critic methods have become a cornerstone of modern deep RL, particularly for challenging domains like Atari games. Their ability to combine the strengths of both policy-based and value-based methods, while addressing the weaknesses of each, makes them a powerful and practical choice. The use of advantage estimation and shared architecture provides a robust foundation for learning complex behaviors in high-dimensional state spaces with large discrete action sets.

8.12.7 Proximal Policy Optimization (PPO)

Please refer to Chapter 5 RLHF part for a detailed introduction and explanation of PPO. The key idea of PPO is data filtering, i.e., in each learning step, stop gradient on those examples where the current policy already assigns high probabilities to the advantageous actions or low probabilities to the disadvantageous actions. The goal for doing this is to avoid over-exploitation of the advantageous actions and under-exploration around the disadvantageous actions. The “proximal” treatment is applied directly in data space, instead of policy space, still less in parameter space.

PPO addresses the key challenge in policy gradient methods: how to determine the largest possible improvement step without destroying the policy. It combines the sample efficiency of TRPO with simpler implementation.

Clipped Objective

PPO’s key innovation is the clipped objective function:

$$L^{CLIP}(\theta) = \mathbb{E}_t[\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)A_t)] \quad (8.96)$$

where:

- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ is the probability ratio
- ϵ is the clip parameter (typically 0.2)
- A_t is the advantage estimate

8.12.8 Actor-Critic Implementation in PPO

PPO builds upon the actor-critic framework while introducing key innovations for stability. Understanding its actor-critic implementation is crucial for grasping how PPO achieves its performance improvements.

Network Architecture

PPO typically employs a shared-backbone architecture:

- **Shared Layers:**
 - Common feature extractor (e.g., CNN for Atari)
 - Parameter sharing improves learning efficiency
 - Learns representations useful for both policy and value
- **Policy Head (Actor):**
 - Outputs action probabilities $\pi_\theta(a|s)$
 - Includes clipped objective:

$$L^{CLIP}(\theta) = \mathbb{E}_t[\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)A_t)] \quad (8.97)$$

$$\text{where } r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

- **Value Head (Critic):**
 - Outputs state value estimate $V_\phi(s)$
 - Often includes clipped value objective:

$$L^{VF}(\phi) = \max((V_\phi(s_t) - R_t)^2, (V_{clipped}(s_t) - R_t)^2) \quad (8.98)$$

$$\text{where } V_{clipped}(s_t) = V_{\phi_{old}}(s_t) + \text{clip}(V_\phi(s_t) - V_{\phi_{old}}(s_t), -\epsilon, \epsilon)$$

Combined Loss Function

PPO optimizes a combined objective:

$$L^{TOTAL}(\theta, \phi) = L^{CLIP}(\theta) - c_1 L^{VF}(\phi) + c_2 L^{ENT}(\theta) \quad (8.99)$$

where:

- c_1 balances value loss (typically 0.5)
- c_2 controls entropy bonus (typically 0.01)
- L^{ENT} is the policy entropy term for exploration

Advantage Estimation

PPO uses GAE for advantage estimation:

$$A^{GAE}(s_t, a_t) = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} \quad (8.100)$$

where:

$$\delta_t = r_t + \gamma V_{\phi}(s_{t+1}) - V_{\phi}(s_t) \quad (8.101)$$

This requires maintaining both:

- Current policy for action selection
- Value function for advantage computation

Training Process

The actor-critic components are updated together:

1. Collect trajectories using current policy π_{θ}
2. Compute advantages using critic V_{ϕ}
3. For K epochs:
 - Sample mini-batches of experience
 - Update both actor and critic using L^{TOTAL}
 - Ensure updates stay within trust region

Key Innovations Beyond Basic Actor-Critic

PPO introduces several improvements:

- **Trust Region Enforcement:**
 - Clipping in both policy and value updates
 - Prevents destructively large updates
 - Maintains proximity to old policy
- **Multiple Update Epochs:**
 - Reuses each batch of experience
 - More stable than single-pass updates
 - Better sample efficiency
- **Adaptive Advantage Estimation:**
 - GAE with learned value function
 - Better credit assignment
 - Reduced variance in updates

Implementation Considerations

Practical aspects of the actor-critic implementation:

- **Shared Parameters:**
 - Early layers share parameters
 - Separate output layers for policy and value
 - Backpropagation through both heads
- **Update Ordering:**
 - Synchronous updates to both networks
 - Gradient scaling for different objectives
 - Advantage normalization before updates
- **Memory Management:**
 - Store trajectories for multiple epochs
 - Maintain old policy for ratio computation
 - Track value predictions for GAE

This actor-critic implementation in PPO represents a sophisticated evolution of the basic actor-critic framework, with additional mechanisms for stability and efficiency. The combination of clipped objectives, value function clipping, and entropy regularization helps achieve stable learning while maintaining the advantages of the actor-critic architecture.

8.13 Bootstrapping in Dense-Reward Settings

8.13.1 Core Bootstrap Concept

The fundamental idea of bootstrapping is to update current estimates using subsequent estimates:

- Don't wait for final outcome
- Use next state's estimate as surrogate target
- Chain together dense rewards

8.13.2 One-Step Bootstrap

Basic form across different algorithms:

Q-Learning Form

$$\text{Target} = r_t + \gamma \max_{a'} Q(s_{t+1}, a') \quad (8.102)$$

- Immediate reward r_t is known and reliable
- Next state's value estimated by $\max_{a'} Q(s_{t+1}, a')$
- Only one step of real reward used

Actor-Critic Form

$$\text{Target} = r_t + \gamma V(s_{t+1}) \quad (8.103)$$

- Same structure as Q-learning
- $V(s_{t+1})$ replaces $\max_{a'} Q(s_{t+1}, a')$
- Still one-step bootstrap

8.13.3 Multi-Step Bootstrap

Extend to n steps for better trade-off:

$$\text{Target} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots + \gamma^n V(s_{t+n}) \quad (8.104)$$

Benefits:

- Uses more real rewards
- Reduces reliance on value estimates
- Still maintains bootstrap advantage

8.13.4 Why Bootstrap Works in Dense Rewards**Key Properties**

- **Frequent Feedback:**
 - Many $r_t \neq 0$ in sequence
 - Each reward provides real information
 - Bootstrapped estimates mix real and predicted rewards
- **Error Reduction:**
 - Value errors decrease with frequent updates
 - Dense rewards provide constant correction
 - Bootstrap target becomes increasingly accurate

Advantage over Monte Carlo

Compared to waiting for episode end:

- **Faster Learning:**

$$\text{Updates per step} = \begin{cases} 1 & \text{Monte Carlo} \\ \text{episode length} & \text{Bootstrap} \end{cases} \quad (8.105)$$

- **Lower Variance:**

- Short-term predictions more reliable
- Dense rewards reduce uncertainty
- Frequent corrections prevent error accumulation

8.13.5 Implementation Considerations**Trade-offs in Steps**

Choice of bootstrap length:

- Short (1-step):
 - Lower variance
 - More bias from value estimates
 - Faster learning in early stages
- Long (n-step):
 - Higher variance
 - More real rewards used
 - Better final performance

TD(λ) Solution

Blend multiple step lengths:

$$\text{Target} = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} \left(\sum_{k=1}^n \gamma^{k-1} r_{t+k} + \gamma^n V(s_{t+n}) \right) \quad (8.106)$$

Benefits:

- Combines advantages of all step lengths
- Automatically adjusts based on value accuracy
- Practical implementation via eligibility traces

8.13.6 Success in Practice

Bootstrap effectiveness shown by:

- **DQN Performance:**
 - Stable learning across many games
 - Efficient value propagation
 - Robust to hyperparameters
- **A2C/A3C Results:**
 - Fast learning with n-step returns
 - Good final performance
 - Parallelizable updates

8.13.7 Dense-Reward vs MCTS Bootstrapping

Having explored bootstrapping in both contexts, we can now analyze their similarities and differences.

Common Bootstrap Principle Both approaches build future values from estimates:

- **Core Idea:**
 - Use estimated values of future states
 - Chain value predictions together
 - Avoid waiting for terminal outcomes
 - Propagate information backwards

Key Differences

- **Reward Structure:**
 - **Dense Rewards (Atari):**

$$Q(s_t, a_t) \leftarrow r_t + \gamma Q(s_{t+1}, a_{t+1}) \quad (8.107)$$

where r_t provides frequent real feedback

- **MCTS (AlphaGo):**

$$Q(s_t, a_t) \leftarrow \text{average of } [v(s_L)] \quad (8.108)$$

where intermediate rewards are zero

- **Information Source:**

- **Dense Rewards:**
 - * Real rewards provide constant correction
 - * Each step gives meaningful feedback
 - * Value errors decrease with frequent updates
- **MCTS:**
 - * Value network provides leaf estimates
 - * No intermediate feedback
 - * Multiple simulations reduce variance
- **Update Mechanism:**
 - **Dense Rewards:**
 - * Temporal difference updates
 - * Learning across episodes
 - * Permanent value estimates
 - **MCTS:**
 - * Monte Carlo averaging within tree
 - * Reset after each move
 - * Temporary search statistics

Complementary Strengths Each form of bootstrapping excels in its domain:

- **Dense-Reward Advantages:**
 - Real feedback at each step
 - Natural learning progression
 - Clear credit assignment
 - Rapid value propagation
- **MCTS Advantages:**
 - Explicit look-ahead
 - Multiple evaluation paths
 - Value accuracy through averaging
 - Focused exploration

Implementation Impact The differences affect implementation:

- **Dense Rewards:**

$$\text{Updates per step} = 1 \tag{8.109}$$

Single update from each real experience
- **MCTS:**

$$\text{Updates per step} = \text{number of simulations} \tag{8.110}$$

Multiple updates from simulated paths

Sample Efficiency Different approaches to improving estimates:

- **Dense Rewards:**
 - Experience replay for data reuse
 - Multi-step returns
 - TD(λ) combinations
- **MCTS:**
 - Multiple simulations per state
 - Policy-guided exploration
 - Value network refinement

This comparison reveals how the same fundamental principle of bootstrapping adapts to different contexts. While dense-reward settings leverage frequent feedback for continuous learning, MCTS uses bootstrapping for efficient tree search with sparse rewards.

8.14 Temporal Difference Learning

Temporal Difference (TD) learning represents one of the most fundamental ideas in reinforcement learning, combining Monte Carlo ideas with dynamic programming's bootstrapping.

8.14.1 The TD Learning Principle

Basic TD Update

The core TD learning update:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_t + \gamma V(s_{t+1}) - V(s_t)] \quad (8.111)$$

where:

- $r_t + \gamma V(s_{t+1})$ is the TD target
- r_t is the immediate reward
- $V(s_{t+1})$ is the bootstrapped future value
- α is the learning rate

The Bootstrap Mechanism

TD learning bootstraps in two key ways:

- **Value Bootstrapping:**
 - Uses $V(s_{t+1})$ as proxy for future returns
 - No need to wait for episode end
 - One estimate builds on another
- **Time Bootstrapping:**
 - Updates occur at each time step
 - Each experience provides learning
 - Quick propagation of value information

8.14.2 Comparison with Other Methods

Monte Carlo Methods

No bootstrapping:

$$V(s_t) \leftarrow V(s_t) + \alpha[G_t - V(s_t)] \quad (8.112)$$

where $G_t = \sum_{k=t}^T \gamma^{k-t} r_k$ is the actual return

- **Advantages:**
 - Unbiased estimates
 - No bootstrap error
 - Works without model
- **Disadvantages:**
 - High variance
 - Must wait for episode end
 - Slower learning

Dynamic Programming

Full bootstrap:

$$V(s) \leftarrow \sum_{s'} P(s'|s, a)[r(s, a, s') + \gamma V(s')] \quad (8.113)$$

- **Advantages:**
 - Full information usage
 - Systematic updates

- Lower variance
- **Disadvantages:**
 - Requires model
 - Computationally expensive
 - Bootstrap bias

8.14.3 TD Learning Properties

Bias-Variance Trade-off

TD learning balances:

- **Bias:**
 - From bootstrapping
 - Initial estimates affect learning
 - Generally decreases over time
- **Variance:**
 - Lower than Monte Carlo
 - Better sample efficiency
 - More stable learning

Online Learning

TD enables true online learning:

- Updates at each step
- No need to store episodes
- Immediate feedback incorporation
- Natural for continuing tasks

8.14.4 Variants and Extensions

n-step TD

Bridges TD and Monte Carlo:

$$G_t^{(n)} = r_t + \gamma r_{t+1} + \dots + \gamma^{n-1} r_{t+n-1} + \gamma^n V(s_{t+n}) \quad (8.114)$$

- $n = 1$: Regular TD
- $n = \infty$: Monte Carlo
- Intermediate n : Balance bias/variance

TD(λ)

Combines multiple n-step returns:

$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)} \quad (8.115)$$

Advantages:

- Unified view of TD and MC
- Adaptive bias-variance trade-off
- Efficient computation through eligibility traces

8.14.5 Connection to Other Concepts**Q-Learning Connection**

Q-learning is TD for state-action values:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \quad (8.116)$$

MCTS Connection

Similar bootstrap principle:

- Both build estimates from future values
- TD: Through temporal sequence
- MCTS: Through tree structure
- Both reduce reliance on terminal outcomes

TD learning represents a fundamental advance in reinforcement learning by introducing bootstrapping in the temporal domain. Its ability to learn online and balance bias-variance trade-offs makes it a cornerstone of modern RL algorithms.

8.15 On-Policy versus Off-Policy Learning**8.15.1 Fundamental Definitions**

- **Behavior Policy** $\mu(a|s)$: Policy generating data
- **Target Policy** $\pi(a|s)$: Policy being learned
- **On-Policy**: $\mu = \pi$
- **Off-Policy**: $\mu \neq \pi$

8.15.2 Mathematical Formulation

On-Policy Updates

Direct expectation under current policy:

$$J(\theta) = \mathbb{E}_{\pi_\theta}[G_t] = \mathbb{E}_{s \sim \rho^\pi, a \sim \pi_\theta}[Q^\pi(s, a)] \quad (8.117)$$

Off-Policy Updates

Uses importance sampling ratio:

$$\rho_t = \frac{\pi(a_t|s_t)}{\mu(a_t|s_t)} \quad (8.118)$$

Corrected expectation:

$$J(\theta) = \mathbb{E}_\mu[\rho_t G_t] = \mathbb{E}_{s \sim \rho^\mu, a \sim \mu} \left[\frac{\pi_\theta(a|s)}{\mu(a|s)} Q^\pi(s, a) \right] \quad (8.119)$$

8.15.3 Algorithm Examples

On-Policy Methods

- **SARSA:**

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \quad (8.120)$$

- **PPO/TRPO:**

$$\min_{\theta} \mathbb{E} \left[\min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} A(s, a), \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}, 1 - \epsilon, 1 + \epsilon\right) A(s, a) \right) \right] \quad (8.121)$$

Off-Policy Methods

- **Q-Learning:**

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)] \quad (8.122)$$

- **DQN:**

$$\Delta\theta \propto (r + \gamma \max_{a'} Q_{\theta-}(s', a') - Q_\theta(s, a)) \nabla_\theta Q_\theta(s, a) \quad (8.123)$$

8.15.4 Key Trade-offs

Sample Efficiency

- **On-Policy:**

- Must discard old data
- Requires fresh samples
- Lower sample efficiency

- **Off-Policy:**

- Can reuse old data
- Experience replay
- Higher sample efficiency

Stability

- **On-Policy:**

- More stable learning
- No distribution shift
- Better convergence properties

- **Off-Policy:**

- Potential instability
- Distribution mismatch
- Need stabilization techniques

8.15.5 Implementation Considerations

On-Policy Implementation

Key requirements:

- Collect new data each iteration
- Update policy using only recent data
- Parallel environment sampling helps
- Example memory requirement: $O(|\text{batch size}|)$

Off-Policy Implementation

Key requirements:

- Maintain replay buffer
- Handle distribution shift
- Target networks for stability
- Example memory requirement: $O(|\text{replay buffer}|)$

8.15.6 Unified View

Both approaches optimize:

$$\max_{\theta} \mathbb{E}_{s \sim \rho, a \sim \mu} \left[\frac{\pi_{\theta}(a|s)}{\mu(a|s)} A(s, a) \right] \quad (8.124)$$

where:

- On-policy: $\mu = \pi_{\theta_{old}}$, recent data
- Off-policy: $\mu \neq \pi_{\theta}$, replay buffer

8.15.7 Application Examples

- **AlphaGo:**
 - Initial SL policy: Off-policy (human data)
 - RL improvement: On-policy (self-play)
- **Atari:**
 - DQN: Off-policy with replay buffer
 - A3C/PPO: On-policy with parallel actors

8.16 Dense versus Sparse Rewards

8.16.1 Reward Characteristics

Atari Games - Dense Rewards

Key properties of Atari reward structure:

- **Frequency:** Rewards obtained frequently during gameplay
 - Points for collecting items
 - Scores for hitting targets
 - Continuous feedback for progress
- **Temporal Structure:**

$$R_{total} = \sum_{t=1}^T r_t \quad \text{where many } r_t \neq 0 \quad (8.125)$$

- **Intermediate Feedback:**
 - Rewards signal immediate action quality
 - Clear correlation between good actions and score increase
 - Multiple reward scales (small points vs bonus points)

Go - Sparse Rewards

Characteristics of Go's reward structure:

- **Frequency:** Single reward at game end

$$R_{total} = z \quad \text{where } z \in \{-1, 0, +1\} \quad (8.126)$$

- **Temporal Structure:**

- No intermediate rewards ($r_t = 0$ for all $t < T$)
- Only terminal state provides feedback
- Binary outcome (win/loss)

8.16.2 Implications for Learning

Impact on Q-Learning

- **Atari (Dense):**
 - Q-values updated frequently
 - Clear temporal difference signals
 - Equation: $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$
 - Many informative updates due to $r_t \neq 0$
- **Go (Sparse):**
 - Most Q-updates only propagate zero rewards
 - Long credit assignment chains
 - Difficult bootstrapping due to delayed feedback
 - Need for auxiliary value estimation

Impact on Policy Gradient

- **Atari (Dense):**

$$\nabla_{\theta} J(\theta) = \mathbb{E}[\nabla_{\theta} \log \pi_{\theta}(a_t | s_t) (\sum_{k=t}^T r_k)] \quad (8.127)$$

- Frequent reward signals guide policy updates
- Lower variance in gradient estimates
- Natural curriculum through score progression

- **Go (Sparse):**

$$\nabla_{\theta} J(\theta) = \mathbb{E}[\nabla_{\theta} \log \pi_{\theta}(a_t | s_t) z] \quad (8.128)$$

- Same reward z applied to all actions in episode
- Higher variance in gradient estimates
- Need for sophisticated value bootstrapping

8.16.3 Solution Approaches

Atari Solutions

Leverage dense reward structure:

- Direct reinforcement learning (DQN, A2C, PPO)
- Experience replay for sample efficiency
- Reward clipping to manage scale differences
- Frame stacking for temporal context

Go Solutions

Overcome sparse rewards:

- Monte Carlo tree search for lookahead
- Value network to predict long-term outcomes
- Self-play for training data generation
- Policy network to guide search

8.16.4 Architectural Implications

The reward structure fundamentally shapes algorithm design:

- **Atari:**
 - Focus on efficient online learning
 - Emphasis on exploration-exploitation
 - Direct value estimation feasible
- **Go:**
 - Need for explicit search
 - Importance of self-play curriculum
 - Reliance on learned value approximation

8.17 Model-Based vs Model-Free Approaches

8.17.1 Model Definition

A model provides the environment dynamics:

- State transition: $P(s_{t+1}|s_t, a_t)$
- Reward function: $R(s_t, a_t, s_{t+1})$

8.17.2 Analysis by Game Type

Go Model Characteristics

- **Perfect Model Available:**
 - Deterministic transitions following game rules
 - Next state exactly computable
 - Zero intermediate rewards
 - Model is computationally cheap
- **AlphaGo's Model Usage:**
 - MCTS uses perfect model for lookahead
 - Each simulation computes exact next states
 - No need to learn dynamics
 - Value network complements perfect rollouts

Atari Model Characteristics

- **Complex Visual Dynamics:**
 - High-dimensional pixel space ($84 \times 84 \times 4$)
 - Complex object interactions
 - Game physics and collision detection
 - Visual effects and animations
- **Learning the Model:**

$$\hat{s}_{t+1}, \hat{r}_t = f_\phi(s_t, a_t) \quad (8.129)$$

where f_ϕ is a learned neural network

8.17.3 Algorithmic Approaches

Model-Free Methods

Learn directly from experience:

- **Q-Learning (DQN):**

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \quad (8.130)$$

- **Policy Gradient (PPO):**

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t | s_t) A_t \quad (8.131)$$

- **Advantages:**

- No model required
- Works with any environment
- Simple implementation

- **Disadvantages:**

- Sample inefficient
- No planning capability
- Limited transfer learning

Model-Based Methods

Use or learn environment dynamics:

- **Go (MCTS):**

- Use perfect model for tree search
- Combine with learned value/policy
- Plan optimal sequences
- Sample efficient due to perfect model

- **Atari (World Models):**

- Learn approximate dynamics model
- Use model for planning/imagination
- Challenge: model errors compound
- High computational cost

8.17.4 Hybrid Approaches

MuZero Architecture

Unifies model-based and model-free:

- Learn implicit dynamics model
- MCTS with learned model
- Works for both Go and Atari
- Key innovation: predict only relevant features

Learning Components

MuZero learns three functions:

- Dynamics: $s_{t+k} = g_{\theta}(s_t, a_t, \dots, a_{t+k-1})$
- Policy: $\pi_{\theta}(a|s_t)$
- Value: $v_{\theta}(s_t)$

8.17.5 Trade-offs Summary

- **Go:**
 - Perfect model available
 - Model-based methods excel
 - MCTS provides strong planning
- **Atari:**
 - Complex visual dynamics
 - Model-free methods more common
 - Model learning is challenging
- **Universal Approach (MuZero):**
 - Learn task-relevant dynamics
 - Combine search and learning
 - Bridge model-based/model-free gap

8.18 Model Predictive Control (MPC)

8.18.1 Core Concept

MPC combines model-based planning with receding horizon control:

- Plan actions for H-step horizon
- Execute only first action
- Replan at next step with updated state

8.18.2 Mathematical Formulation

At each time t , solve:

$$\max_{a_t, \dots, a_{t+H-1}} \sum_{k=0}^{H-1} \gamma^k r(s_{t+k}, a_{t+k}) \quad (8.132)$$

subject to:

$$s_{t+k+1} = f(s_{t+k}, a_{t+k}) \quad (8.133)$$

where:

- H is planning horizon
- f is dynamics model (learned or known)
- Only a_t is actually executed

8.18.3 Algorithm Structure

Repeat at each timestep:

1. State Estimation:

- Get current state s_t
- Update model if learning-based

2. Planning:

- Generate trajectories for horizon H
- Optimize action sequence
- Can use various optimization methods:
 - Random shooting
 - Cross-entropy method (CEM)
 - Gradient-based optimization

3. Execution:

- Apply first action a_t
- Observe next state s_{t+1}
- Repeat process

8.18.4 Key Advantages

- **Robustness:**
 - Continuous replanning handles uncertainty
 - Adapts to model errors
 - Recovery from disturbances
- **Flexibility:**
 - Works with learned or known models
 - Can incorporate constraints
 - Adjustable planning horizon

8.18.5 Comparison to Other Methods

- vs Pure Planning:
 - More robust to model errors
 - Lower computational cost
- vs Model-Free RL:
 - More sample efficient
 - Better online adaptation
 - Requires good model

8.18.6 Unifying View: MPC and AlphaGo Planning

Both MPC and AlphaGo exemplify model-based planning, though in different domains. Their core similarities reveal fundamental principles of planning in sequential decision making:

Planning Structure

- **Look-ahead Tree/Trajectory:**

- AlphaGo: Tree expansion with MCTS

$$s_t \rightarrow \{(a, s_{t+1}) \rightarrow (a', s_{t+2}) \rightarrow \dots\} \quad (8.134)$$

- MPC: Trajectory optimization over horizon H

$$s_t \rightarrow s_{t+1} \rightarrow \dots \rightarrow s_{t+H} \quad (8.135)$$

- **Receding Horizon:**

- AlphaGo: New MCTS tree at each state
- MPC: New trajectory optimization at each step
- Both: Execute first action, then replan

Value Estimation

- **Terminal Value:**
 - AlphaGo: $v_\theta(s_L)$ at leaf nodes
 - MPC: Terminal cost $h(s_{t+H})$ or None
- **Cumulative Reward:**
 - AlphaGo: $\sum_t r_t + v_\theta(s_L)$
 - MPC: $\sum_{k=0}^{H-1} r(s_{t+k}, a_{t+k}) + h(s_{t+H})$

Model Usage

- **Transition Model:**

$$s_{next} = \begin{cases} \text{Game rules} & \text{AlphaGo (perfect model)} \\ f_\theta(s, a) \text{ or } f(s, a) & \text{MPC (learned/known model)} \end{cases} \quad (8.136)$$

- **Model Application:**
 - AlphaGo: Exact next states in tree search
 - MPC: State predictions over horizon

Optimization Strategy

- **Search Space:**
 - AlphaGo: Tree of discrete actions
 - MPC: Sequence of continuous actions
- **Selection/Optimization:**

$$a_t = \begin{cases} \arg \max_a \text{Upper Confidence Bound}(s_t, a) & \text{AlphaGo} \\ \arg \max_{a_{t:t+H-1}} \sum_{k=0}^{H-1} r(s_{t+k}, a_{t+k}) & \text{MPC} \end{cases} \quad (8.137)$$

Key Principles Both Share

- **Planning Horizon:**
 - Limited-depth forward planning
 - Balance computation vs horizon length
 - Replan after each execution
- **Uncertainty Handling:**
 - Continuous replanning compensates for errors

- Value estimation for beyond-horizon effects
- Model accuracy most critical in near term

- **Computation Allocation:**

- More computation for immediate actions
- Decreasing precision for future steps
- Real-time computation constraints

This comparison reveals that while AlphaGo and MPC were developed in different communities (discrete games vs continuous control), they embody the same fundamental principles of model-based planning: look-ahead with limited horizon, value estimation for long-term effects, and continuous replanning for robustness.

8.19 Planning versus Policy Approaches

8.19.1 Fundamental Distinction

Policy Definition

A policy directly maps states to actions:

- **Mapping:** $\pi_{\theta}(a|s)$ parameterized by θ
- **Computation:** Single forward pass through network
- **Memory:** Fixed size independent of decision
- **Characteristics:** Can be stochastic or deterministic

Planning Definition

Planning searches over future trajectories:

- **Search Space:** Tree or trajectory of depth H
- **Computation:** Variable based on search depth/width
- **Memory:** Grows with search space
- **Characteristics:** Usually deterministic given compute budget

8.19.2 Computational Properties

Policy Computation

Analysis of policy computational characteristics:

- **Time Complexity:** $O(1)$ per decision
- **Space Complexity:** $O(|\theta|)$ for parameters θ
- **Scaling:** Fixed with state/action space size
- **Parallelization:** Highly parallelizable batch inference

Planning Computation

Analysis of planning computational requirements:

- **Time Complexity:** $O(b^d)$ for breadth b , depth d
- **Space Complexity:** $O(b^d)$ for tree storage
- **Scaling:** Exponential with horizon length
- **Parallelization:** Tree expansion can be parallelized

8.19.3 Information Usage

Policy Information Processing

How policies utilize available information:

- **Historical Data:** Learns patterns from past experiences
- **Future Consideration:** Implicit in learned parameters
- **Uncertainty:** Encoded in policy distribution
- **Model Dependence:** Can operate model-free

Planning Information Processing

How planning leverages information:

- **Historical Data:** Only through value estimates
- **Future Consideration:** Explicit trajectory evaluation
- **Uncertainty:** Handled through tree exploration
- **Model Dependence:** Requires accurate transition model

8.19.4 Decision Quality

Policy Decisions

Quality characteristics of policy-based decisions:

- **Optimality:** Bounded by training experience
- **Consistency:** Deterministic if policy is deterministic
- **Generalization:** Handles novel states within training distribution
- **Recovery:** Must learn recovery strategies explicitly

Planning Decisions

Quality characteristics of planning-based decisions:

- **Optimality:** Improves with computation budget
- **Consistency:** May vary with search parameters
- **Generalization:** Perfect within model bounds
- **Recovery:** Natural through replanning

8.19.5 Hybrid Approaches

AlphaGo Integration

Combines policy and planning through:

- Policy network guides MCTS exploration
- Planning improves over policy decisions
- Value network evaluates leaf nodes
- Mathematical framework:

$$P(a|s) = \text{softmax}(Q(s, a)/\tau) \quad (8.138)$$

where

$$Q(s, a) = \frac{W(s, a)}{N(s, a)} + c\pi(a|s) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)} \quad (8.139)$$

MuZero Advancement

Further integration through:

- Learned implicit model for planning
- Joint policy, value, and dynamics learning
- Planning used in both training and inference
- Unified model-free and model-based approaches

8.19.6 Domain-Specific Considerations

Policy-Favorable Conditions

Prefer policy when:

- Real-time decisions required
- Large/continuous action spaces present
- Accurate model unavailable
- Limited computation budget
- Environment highly stochastic

Planning-Favorable Conditions

Prefer planning when:

- Computation time available
- Accurate model exists
- Critical decisions required
- Environment near-deterministic
- Safety constraints important

Hybrid-Favorable Conditions

Use hybrid approach when:

- Domain complexity high (e.g., Go)
- Variable computation budget available
- Both speed and accuracy needed
- Rich state/action structure exists
- Long-term consequences significant

8.19.7 Implementation Considerations

Policy Implementation

Key steps in policy implementation:

1. Design network architecture
2. Choose policy gradient algorithm
3. Configure experience collection
4. Set training hyperparameters
5. Implement inference pipeline

Planning Implementation

Key steps in planning implementation:

1. Define state transition model
2. Select search algorithm
3. Configure expansion strategy
4. Implement backup operations
5. Optimize computation allocation

8.19.8 Future Trends

The field increasingly moves toward:

- Dynamic computation allocation
- Adaptive planning depth
- Improved model learning
- Better policy-planning integration
- Hardware-aware algorithm design

This comparison reveals that modern deep reinforcement learning systems often benefit from combining both approaches, using policies for rapid approximate decisions and planning for refined decision-making when computation permits. The success of systems like AlphaGo and MuZero demonstrates the power of properly balancing these complementary approaches.

8.20 Relationship Between Planning and Control

8.20.1 Core Definitions and Distinctions

Planning Characteristics

Planning is characterized by:

- Finding sequences of actions to reach desired states
- Operating primarily in state/action space
- Emphasis on look-ahead and trajectory generation
- Example: MCTS in AlphaGo finding sequences of moves

Control Characteristics

Control focuses on:

- Regulating system behavior to achieve/maintain desired states
- Operating in both state and dynamics space
- Emphasis on feedback and stability
- Example: PID controller maintaining robot joint angles

8.20.2 Mathematical Formulations

Planning Formulation

The planning problem can be stated as:

$$\max_{a_1, \dots, a_n} \sum_{i=1}^n r(s_i, a_i) \quad (8.140)$$

subject to:

$$s_{i+1} = f(s_i, a_i) \quad (8.141)$$

where:

- a_i are discrete or continuous actions
- s_i are system states
- $r(s_i, a_i)$ is the reward function
- f is the transition function

Control Formulation

The control problem is typically formulated as:

$$\min_{u(t)} \int_0^T L(x(t), u(t)) dt \quad (8.142)$$

subject to:

$$\dot{x}(t) = f(x(t), u(t)) \quad (8.143)$$

where:

- $u(t)$ is the control input
- $x(t)$ is the system state
- L is the cost function
- f represents system dynamics

8.20.3 Key Distinctions

Time Horizon

- **Planning:**
 - Discrete-time formulation
 - Finite horizon typically
 - Coarser time discretization
- **Control:**
 - Continuous-time or high-frequency discrete
 - Often infinite horizon
 - Fine-grained time discretization

State Space Treatment

- **Planning:**
 - Discrete or discretized states
 - Focus on reachability
 - Often deals with combinatorial spaces
- **Control:**
 - Continuous state spaces
 - Focus on stability
 - Usually deals with smooth spaces

8.20.4 Model Predictive Control: A Bridge

MPC represents a fusion of planning and control approaches:

Key Components

- **Planning Aspect:**

$$\min_{u_{t:t+H}} \sum_{k=t}^{t+H} L(x_k, u_k) \quad (8.144)$$

- **Control Aspect:**

- Apply only first control input
- Continuous replanning
- Feedback incorporation

MPC Algorithm

At each timestep:

1. Measure current state x_t
2. Solve optimization problem:

$$u_{t:t+H}^* = \arg \min_{u_{t:t+H}} \sum_{k=t}^{t+H} L(x_k, u_k) \quad (8.145)$$

subject to:

$$x_{k+1} = f(x_k, u_k) \quad (8.146)$$

$$x_k \in \mathcal{X} \quad (8.147)$$

$$u_k \in \mathcal{U} \quad (8.148)$$

3. Apply u_t^*
4. Repeat at next timestep

8.20.5 Comparative Analysis

Computational Aspects

- **Planning:**
 - Higher computational cost
 - Better optimality for complex tasks
 - Handles discontinuous objectives

- Harder real-time guarantees
- **Control:**
 - Lower computational cost
 - Better stability guarantees
 - Continuous operation
 - Easier real-time implementation

Uncertainty Handling

- **Planning:**
 - Through search tree expansion
 - Scenario-based planning
 - Replanning on deviations
- **Control:**
 - Through robust control methods
 - Adaptive control
 - Continuous feedback

8.20.6 Modern Integration

Hierarchical Framework

Modern systems often employ a hierarchical approach:

- **High Level:** Planning (slow update)
 - Long-term trajectory generation
 - Discrete decision making
- **Mid Level:** MPC (medium update)
 - Trajectory optimization
 - Constraint handling
- **Low Level:** Control (fast update)
 - Trajectory tracking
 - Disturbance rejection

Learning Integration

Modern approaches incorporate learning:

- Neural network dynamics models
- Learned value functions
- Policy networks for initialization
- End-to-end differentiable planning

8.20.7 Future Directions

The field moves toward:

- Unified planning and control frameworks
- Learning-augmented MPC
- Multi-timescale integration
- Hardware-aware algorithms

While planning and control share the goal of decision-making in dynamic systems, they represent complementary approaches optimized for different aspects of the problem. Modern systems increasingly recognize that these approaches are not mutually exclusive but rather can be integrated effectively at different levels of the control hierarchy.

8.21 Policy and Value Functions in Planning and Control

8.21.1 Fundamental Roles

Policy Functions

A policy π maps states to actions:

- **Deterministic Policy:** $\pi : S \rightarrow A$
- **Stochastic Policy:** $\pi(a|s)$ giving probability distribution over A
- **Parameterized Form:** $\pi_\theta(a|s)$ with parameters θ

Value Functions

Value functions estimate expected returns:

- **State Value:** $V^\pi(s) = \mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s]$
- **Action Value:** $Q^\pi(s, a) = \mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a]$
- **Advantage:** $A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$

8.21.2 Integration in Planning

Policy-Guided Planning

Policy networks enhance planning efficiency:

$$\text{PUCT}(s, a) = Q(s, a) + c\pi_\theta(a|s) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)} \quad (8.149)$$

where:

- $Q(s, a)$ is the action value from search
- $\pi_\theta(a|s)$ guides exploration
- $N(s, a)$ counts state-action visits
- c balances exploration and exploitation

Value-Guided Planning

Value functions enhance planning depth:

$$V_{plan}(s) = \max_a \left\{ r(s, a) + \gamma \max_{a'} [\lambda V_\theta(s') + (1 - \lambda) V_{tree}(s')] \right\} \quad (8.150)$$

where:

- $V_\theta(s)$ is the learned value function
- $V_{tree}(s)$ is the tree search value
- λ balances learned and searched values

8.21.3 Integration in Control

Policy-Based Control

Neural network policies for control:

$$u(t) = \pi_\theta(x(t)) + \epsilon(t) \quad (8.151)$$

where:

- $u(t)$ is the control input
- $x(t)$ is the system state
- $\epsilon(t)$ is exploration noise
- π_θ is a neural network controller

Value-Based Control

Value functions guide optimal control:

$$u^*(t) = \arg \max_u [r(x, u) + \gamma V(f(x, u))] \quad (8.152)$$

where:

- $f(x, u)$ is the system dynamics
- $V(x)$ approximates optimal value function
- $r(x, u)$ is immediate reward/cost

8.21.4 Hybrid Architectures

Model Predictive Control with Policy and Value

MPC enhanced by learned components:

$$J(u_{t:t+H}) = \sum_{k=t}^{t+H-1} r(x_k, u_k) + V_\theta(x_{t+H}) \quad (8.153)$$

subject to:

$$u_k = \pi_\theta(x_k) + \Delta u_k \quad (8.154)$$

$$x_{k+1} = f(x_k, u_k) \quad (8.155)$$

$$\|\Delta u_k\| \leq \epsilon \quad (8.156)$$

where:

- $V_\theta(x_{t+H})$ provides terminal cost
- $\pi_\theta(x_k)$ initializes optimization
- Δu_k allows deviation from policy
- ϵ constrains policy deviation

AlphaGo-Style Planning with Control

Combining search and control:

$$a_t = \arg \max_a [Q_{mcts}(s_t, a) + \alpha Q_{control}(s_t, a)] \quad (8.157)$$

where:

- Q_{mcts} comes from tree search
- $Q_{control}$ from control optimization
- α balances planning and control

8.21.5 Learning Mechanisms

Policy Learning

Multiple approaches to policy improvement:

- **Policy Gradient:**

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\pi_{\theta}} [\nabla_{\theta} \log \pi_{\theta}(a|s) A^{\pi}(s, a)] \quad (8.158)$$

- **Trust Region:**

$$\max_{\theta} \mathbb{E} \left[\frac{\pi_{\theta}(a|s)}{\pi_{\theta_{old}}(a|s)} A^{\pi}(s, a) \right] \text{ s.t. } D_{KL}(\pi_{\theta_{old}}, \pi_{\theta}) \leq \delta \quad (8.159)$$

Value Learning

Multiple approaches to value estimation:

- **Temporal Difference:**

$$V_{\theta}(s_t) \leftarrow V_{\theta}(s_t) + \alpha [r_t + \gamma V_{\theta}(s_{t+1}) - V_{\theta}(s_t)] \quad (8.160)$$

- **Monte Carlo:**

$$V_{\theta}(s_t) \leftarrow V_{\theta}(s_t) + \alpha [G_t - V_{\theta}(s_t)] \quad (8.161)$$

8.21.6 Implementation Considerations

Architecture Design

Key design decisions include:

- Network architectures for policy and value
- Integration depth in planning/control
- Balance between learned and optimized components
- Computational budget allocation

Training Strategy

Effective training requires:

- Off-policy data collection
- Experience replay management
- Value and policy updates scheduling
- Exploration strategy design

8.21.7 Future Directions

Emerging trends include:

- End-to-end differentiable planning
- Meta-learning for policy adaptation
- Multi-task value functions
- Hierarchical policy architectures
- Uncertainty-aware value estimates

Both policy and value functions serve crucial roles in modern planning and control systems. While traditionally these components were often used separately, contemporary approaches increasingly recognize their complementary nature and leverage both for enhanced performance. The integration of these learned components with classical planning and control methods represents a promising direction for future research and development.

8.22 Online versus Offline Reinforcement Learning

8.22.1 Fundamental Distinctions

Online RL

Characterization of online learning:

- **Environment Access:** Direct interaction during training
- **Data Collection:** Agent actively gathers experiences
- **Policy Update:** Immediate updates from new experiences
- **Exploration:** Agent controls data collection strategy

Offline RL

Key characteristics of offline learning:

- **Environment Access:** No interaction during training
- **Data Collection:** Fixed dataset \mathcal{D} of past experiences
- **Policy Update:** Limited to existing data distribution
- **Exploration:** Must leverage existing data coverage

8.22.2 Mathematical Formulation

Online RL Objective

Standard RL objective with data collection:

$$\max_{\theta} \mathbb{E}_{\pi_{\theta}} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right] \quad (8.162)$$

where experiences are collected using current policy:

$$(s_t, a_t, r_t, s_{t+1}) \sim \pi_{\theta} \quad (8.163)$$

Offline RL Objective

Modified objective using fixed dataset:

$$\max_{\theta} \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right] \quad (8.164)$$

subject to distributional constraints:

$$D(\pi_{\theta}(\cdot|s) \parallel \pi_{\beta}(\cdot|s)) \leq \epsilon \quad (8.165)$$

where:

- \mathcal{D} is the fixed dataset
- π_{β} is the behavior policy that generated the data
- D is a divergence measure
- ϵ constrains deviation from data distribution

8.22.3 Key Challenges

Online RL Challenges

- **Sample Efficiency:**

$$\text{Samples Needed} \propto \frac{|\mathcal{S}| \times |\mathcal{A}|}{\epsilon^2} \quad (8.166)$$

- **Exploration-Exploitation:**

$$a_t = \arg \max_a [Q(s_t, a) + c \sqrt{\frac{\log t}{N(s_t, a)}}] \quad (8.167)$$

- **Safety During Learning:**

- Risk of poor actions during exploration
- Need for safe exploration strategies

Offline RL Challenges

- **Distribution Shift:**

$$\mathbb{E}_{\pi_\theta}[Q(s, a)] \neq \mathbb{E}_{\mathcal{D}}[Q(s, a)] \quad (8.168)$$

- **Value Overestimation:**

$$Q(s, a) \approx r + \gamma \max_{a' \in \text{supp}(\mathcal{D})} Q(s', a') \quad (8.169)$$

- **Limited State Coverage:**

- Unknown state-action combinations
- Uncertainty in unexplored regions

8.22.4 Modern Algorithms

Online Methods

Representative approaches:

- **PPO:** Trust region policy optimization

$$L^{CLIP}(\theta) = \mathbb{E}_t[\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)A_t)] \quad (8.170)$$

- **SAC:** Maximum entropy RL

$$J(\theta) = \mathbb{E}_{\pi_\theta} \left[\sum_{t=0}^{\infty} \gamma^t (r_t + \alpha H(\pi_\theta(\cdot | s_t))) \right] \quad (8.171)$$

- **DrQ:** Data-regularized Q-learning

$$L(\theta) = \mathbb{E}_{(s, a, r, s') \sim \mathcal{B}} [(r + \gamma \max_{a'} Q_{\theta'}(\text{aug}(s'), a') - Q_\theta(\text{aug}(s), a))^2] \quad (8.172)$$

Offline Methods

Key algorithms:

- **BCQ:** Behavior-Constrained Q-learning

$$a^* = \arg \max_{a_i \sim G_\omega(s)} Q_\theta(s, a_i) \quad (8.173)$$

- **CQL:** Conservative Q-Learning

$$L_{CQL}(\theta) = \alpha \mathbb{E}_{s \sim \mathcal{D}} [\log \sum_a \exp(Q_\theta(s, a)) - \mathbb{E}_{a \sim \mathcal{D}} [Q_\theta(s, a)]] \quad (8.174)$$

- **TD3+BC:** TD3 with Behavior Cloning

$$\pi_\phi(s) = \arg \max_a [\lambda Q_\theta(s, a) + (1 - \lambda) \pi_\beta(a | s)] \quad (8.175)$$

8.22.5 Implementation Considerations

Online Implementation

Key requirements:

- **Environment Interface:**
 - Real-time interaction capability
 - Reset functionality
 - State/reward observation
- **Experience Collection:**
 - Parallel environment instances
 - Replay buffer management
 - Exploration strategy
- **Training Loop:**

$$\text{while not converged: } \begin{cases} \text{Collect experiences} \\ \text{Update policy} \\ \text{Evaluate performance} \end{cases} \quad (8.176)$$

Offline Implementation

Essential components:

- **Dataset Management:**
 - Efficient data storage
 - Batch sampling
 - Data preprocessing
- **Policy Constraints:**
 - Support estimation
 - Uncertainty quantification
 - Distribution matching
- **Training Process:**

$$\text{repeat until convergence: } \begin{cases} \text{Sample batch from } \mathcal{D} \\ \text{Update with constraints} \\ \text{Validate on held-out data} \end{cases} \quad (8.177)$$

8.22.6 Applications

Online RL Applications

Suitable domains:

- Simulated environments
- Game playing
- Robotic control in safe settings
- Continuous learning systems

Offline RL Applications

Appropriate use cases:

- Healthcare decision making
- Industrial process control
- Autonomous driving
- Recommendation systems

8.22.7 Future Directions

Emerging trends:

- **Hybrid Approaches:**
 - Offline pre-training with online fine-tuning
 - Selective environment interaction
 - Data-driven exploration
- **Theoretical Advances:**
 - Tighter bounds on offline learning
 - Uncertainty quantification
 - Causal inference integration
- **Practical Improvements:**
 - Better support estimation
 - More efficient constraints
 - Scalable implementations

The distinction between online and offline RL represents a fundamental trade-off between data collection flexibility and real-world applicability. While online RL offers the potential for continuous improvement through interaction, offline RL provides a path to leveraging existing datasets in scenarios where direct environment interaction is impractical or unsafe. Modern approaches increasingly explore combinations of both paradigms, seeking to leverage their complementary strengths.

8.23 Summary

8.23.1 Core Components in Deep RL

All approaches can be understood through four fundamental components:

- **Policy Network** $\pi_\theta(a|s)$:
 - Direct action selection
 - Can be deterministic or stochastic
 - Works with or without planning
- **Value Network** $V_\theta(s)$ or $Q_\theta(s, a)$:
 - Long-term outcome prediction
 - Guides policy improvement
 - Enables bootstrapping
- **Dynamics Model** $f_\theta(s, a)$:
 - State transition prediction
 - Either perfect (Go) or learned (Atari)
 - Enables planning when available
- **Planning Module**:
 - Action sequence optimization
 - Uses model for lookahead
 - Can combine with policy/value

8.23.2 Algorithm Classification

Different approaches emphasize different components:

Algorithm	Policy	Value	Model	Planning
AlphaGo	✓	✓	Perfect	MCTS
DQN	Implicit	✓	No	No
PPO	✓	✓	No	No
MPC	Optional	Optional	✓	Horizon-H
MuZero	✓	✓	Learned	MCTS

8.23.3 Key Trade-offs

Model-Based vs Model-Free

Choice depends on:

- Model availability/accuracy
- Computational budget
- Sample efficiency needs
- Planning horizon length

Planning vs Direct Policy

Balance between:

- Computation per decision
- Required optimality
- Environment predictability
- Real-time constraints

8.23.4 Unified Learning Framework

Most algorithms optimize some combination of:

$$\mathcal{L}(\theta) = \underbrace{\mathbb{E}[(r + \gamma V_{\text{target}} - V_{\theta})^2]}_{\text{value loss}} + \underbrace{\mathbb{E}[\log \pi_{\theta} A]}_{\text{policy loss}} + \underbrace{\mathbb{E}[\|f_{\theta}(s, a) - s'\|^2]}_{\text{model loss}} \quad (8.178)$$

Where:

- Value target comes from bootstrapping or planning
- Advantage A comes from value or planning
- Model loss only present in model-based methods

8.23.5 Domain-Specific Insights

Different problems emphasize different components:

- **Perfect Model + Sparse Reward (Go):**
 - Heavy planning (MCTS)
 - Policy for search guidance
 - Value for leaf evaluation

- **No Model + Dense Reward (Atari):**

- Direct policy/value learning
- Bootstrapped updates
- Experience replay

- **Learned Model + Control (MPC):**

- Short-horizon planning
- Continuous replanning
- Model adaptation

This unified framework reveals that success in deep RL comes from appropriately combining these components based on problem characteristics, computational resources, and performance requirements. The field's complexity arises not from the individual components, but from the many ways they can be effectively combined.

Chapter 9

Trees and Boosting

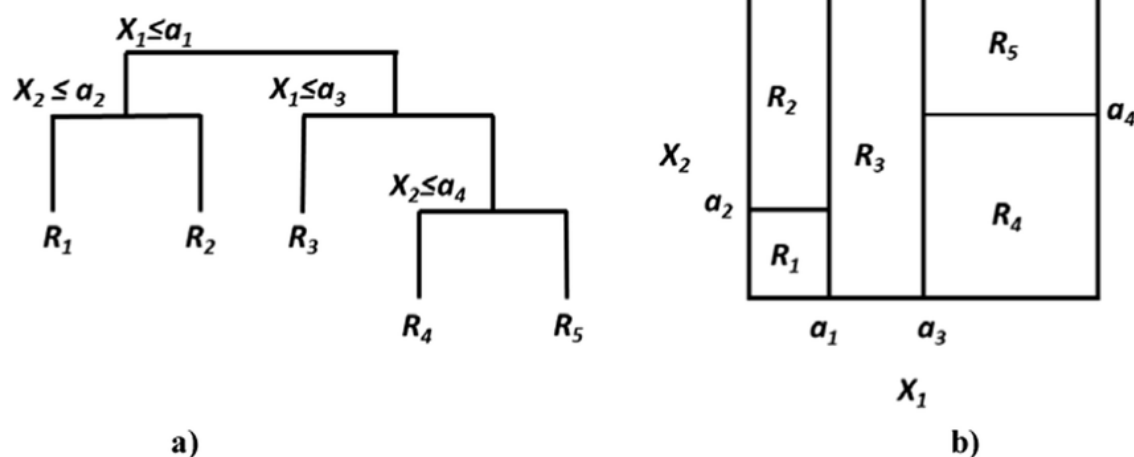


Figure 9.1: Tree

Chapter Overview

This chapter builds from regression trees to modern boosting methods, starting with the fundamental concept of recursive binary partitioning for piecewise-constant function approximation. From this foundation, we develop L2 boosting as a natural extension where regression trees serve as base learners in iterative residual fitting, providing a simple yet powerful framework for gradient descent in function space. This leads to XGBoost, which enhances the boosting framework through second-order approximation of arbitrary loss functions, elegantly transforming each iteration into a weighted least squares problem where weights come from the Hessian and the working response is the negative gradient-to-Hessian ratio. Having established this modern perspective, we then look back at two influential historical developments: AdaBoost, which introduced the key ideas of exponential loss and multiplicative weight updates, and Random Forests, which take a parallel rather than sequential approach to ensemble building through bootstrap aggregation and random feature

selection. This organization emphasizes the mathematical progression from simple regression trees to sophisticated boosting algorithms while acknowledging the historical developments that shaped the field.

9.1 Incremental Model Improvement: From Deep Learning to Trees

9.1.1 The Principle of Incremental Learning

In the previous chapters, we studied deep learning models trained through gradient descent and back-propagation. The core idea is simple yet powerful: start with an initial model and incrementally improve it through small updates. Specifically, at each iteration t :

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta} L(\theta_t) \quad (9.1)$$

where θ_t represents the model parameters and η is the learning rate. Each update slightly adjusts the model to better fit the training data.

9.1.2 Three Paradigms of Incremental Improvement

This chapter introduces two new approaches to machine learning - decision trees and boosting - that also follow the principle of incremental improvement, albeit in different ways:

1. **Gradient-Based Neural Networks:** - Model: $f(x; \theta)$ parameterized by weights θ - Increment: Small parameter updates via gradient descent - Update rule: $\theta_{t+1} = \theta_t - \eta \nabla_{\theta} L(\theta_t)$
2. **Decision Trees:** - Model: Piecewise constant function on regions $\{R_m\}$ - Increment: Binary splitting of existing regions - Update rule: $R_m \rightarrow \{R_{m1}, R_{m2}\}$ via optimal splits
3. **Boosting:** - Model: Additive ensemble $f_M(x) = \sum_{m=1}^M h_m(x)$ - Increment: Addition of new base learners (trees) - Update rule: $f_M = f_{M-1} + h_M$

9.1.3 Geometric Interpretation

These three approaches can be understood geometrically:

1. **Gradient Descent:** - Moves continuously in the parameter space - Each step follows the direction of steepest descent - Local linear approximation of the loss surface
2. **Tree Growing:** - Refines the partition of feature space - Each split creates a new decision boundary - Piecewise constant approximation gets finer
3. **Boosting:** - Moves in function space - Each step adds a new function - Residual fitting guides the improvement

9.1.4 Common Mathematical Structure

Despite their different appearances, these methods share a common mathematical structure:

1. **Optimization Objective:**

$$\min_f \sum_{i=1}^n L(y_i, f(x_i)) + \text{complexity}(f) \quad (9.2)$$

2. **Iterative Improvement:** - Start with simple model - Repeatedly apply local improvements - Stop when improvements become small

3. **Bias-Variance Trade-off:** - Initial model is biased but stable - Each increment reduces bias - Too many increments increase variance

9.1.5 The Role of Gradients

Gradients play a central role in all three approaches:

1. **Neural Networks:** - Direct gradient descent in parameter space - Back-propagation computes exact gradients - Learning rate controls step size

2. **Decision Trees:** - Split criterion approximates gradient - Greedy optimization at each split - Binary decisions approximate continuous gradients

3. **Gradient Boosting:** - Explicit gradient descent in function space - Trees fit negative gradients - Learning rate scales tree contributions

This unifying view helps us understand these seemingly different approaches as variations on the same theme: incremental model improvement guided by some form of gradient information.

9.1.6 Looking Ahead

The rest of this chapter will explore trees and boosting in detail, keeping in mind their connection to the gradient-based learning we've studied previously. We'll see how these methods:

- Offer different trade-offs between bias and variance
- Handle the curse of dimensionality
- Manage computational complexity
- Combat overfitting
- Achieve state-of-the-art performance in many applications

By understanding these methods through the lens of incremental improvement, we can better appreciate their strengths, limitations, and relationships to deep learning.

9.2 Decision Trees

9.2.1 A Motivating Example

Consider a simple yet illustrative problem: classifying individuals as male or female based on their height and weight measurements. While seemingly straightforward, this example encapsulates the key ideas behind decision trees and their geometric interpretation.

Let $(X_1, X_2) \in \mathbb{R}^2$ represent the height and weight measurements respectively, and $Y \in \{0, 1\}$ denote the gender (0 for female, 1 for male). Given a training dataset $\{(x_{i1}, x_{i2}, y_i)\}_{i=1}^n$, we aim to construct a predictor for unseen individuals.

9.2.2 Decision Rules and Tree Structure

A decision tree makes predictions through a sequence of binary questions. For our example, a simple tree structure might be:

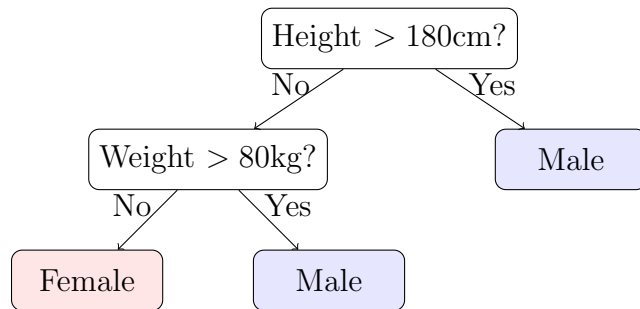


Figure 9.2: Decision Tree Structure

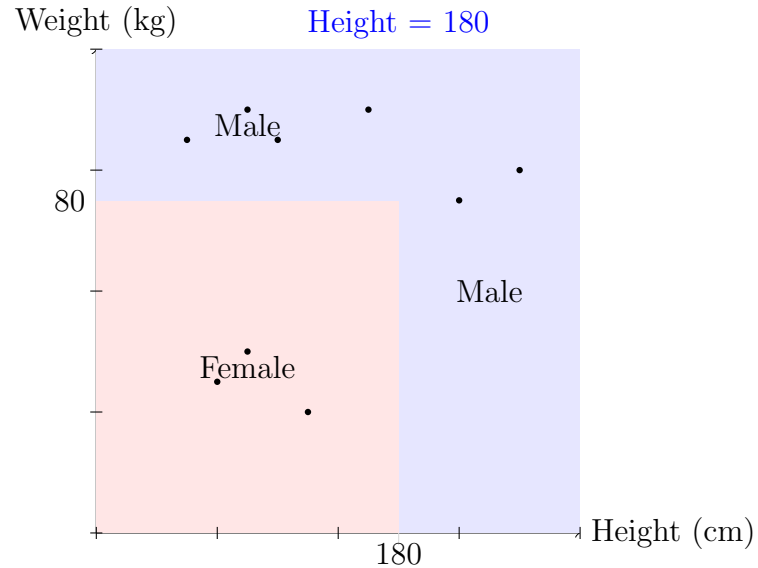


Figure 9.3: Feature Space Partition

- First split: Height > 180cm → Male
- For Height ≤ 180cm:
 - Weight > 80kg → Male
 - Weight ≤ 80kg → Female

9.2.3 The Concept of Purity

The quality of a partition is measured by the “purity” of the resulting regions. For a region R containing n_R observations, let $p_k(R)$ denote the proportion of class k observations in R . Common purity measures include:

1. Misclassification error:

$$\text{error}(R) = 1 - \max_k p_k(R) \quad (9.3)$$

2. Gini index:

$$\text{Gini}(R) = 1 - \sum_k p_k(R)^2 \quad (9.4)$$

3. Cross-entropy:

$$\text{Entropy}(R) = - \sum_k p_k(R) \log_2(p_k(R)) \quad (9.5)$$

For binary classification ($K = 2$), all these measures are convex functions of the class proportion $p_1(R)$, reaching their maximum at $p_1(R) = 0.5$ and minimum at $p_1(R) \in \{0, 1\}$.

9.2.4 Splitting Criterion

When considering a split s that partitions region R into R_L and R_R , we compute the reduction in impurity:

$$\Delta I(s, R) = I(R) - \frac{|R_L|}{|R|} I(R_L) - \frac{|R_R|}{|R|} I(R_R) \quad (9.6)$$

where $I(\cdot)$ is any of the impurity measures defined above. The optimal split s^* maximizes this reduction:

$$s^* = \arg \max_s \Delta I(s, R) \quad (9.7)$$

9.2.5 From Classification to Regression

This framework extends naturally to regression problems where $Y \in \mathbb{R}$. The key modifications are:

1. The prediction in each region R becomes the mean of the response values:

$$\hat{c}_R = \frac{1}{|R|} \sum_{i: x_i \in R} y_i \quad (9.8)$$

2. The impurity measure becomes the mean squared error:

$$I(R) = \frac{1}{|R|} \sum_{i: x_i \in R} (y_i - \hat{c}_R)^2 \quad (9.9)$$

This connection between classification and regression trees provides a unified framework for understanding tree-based methods, which we will explore in subsequent sections.

9.3 Regression Trees

A regression tree constructs a piecewise constant prediction function through recursive partitioning of the feature space.

9.3.1 Mathematical Framework

Let (X, Y) be a random pair where $X \in \mathcal{X} \subset \mathbb{R}^p$ represents the feature vector and $Y \in \mathbb{R}$ is the response variable. Given training data $\{(x_i, y_i)\}_{i=1}^n$, our goal is to estimate the regression function $f(x) = \mathbb{E}[Y|X = x]$.

A regression tree partitions \mathcal{X} into M disjoint regions $\{R_m\}_{m=1}^M$ and fits a constant value in each region:

$$f(x) = \sum_{m=1}^M c_m \mathbb{I}\{x \in R_m\} \quad (9.10)$$

where $c_m \in \mathbb{R}$ is the prediction value for region R_m and $\mathbb{I}\{\cdot\}$ denotes the indicator function.

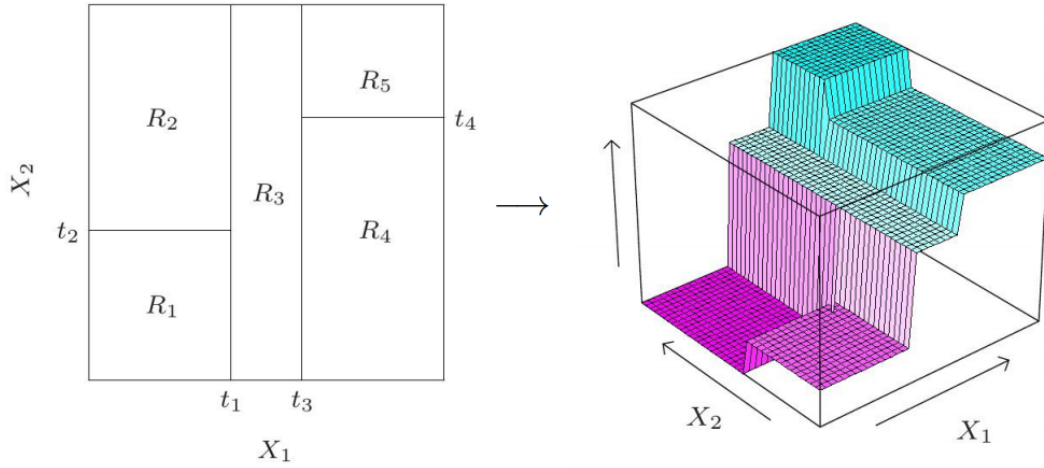


Figure 9.4: Regression tree

9.3.2 Optimization Problem

The estimation procedure aims to minimize:

$$\min_{\{R_m\}_{m=1}^M, \{c_m\}_{m=1}^M} \left\{ \sum_{i=1}^n (y_i - \sum_{m=1}^M c_m \mathbb{I}\{x_i \in R_m\})^2 + \lambda M \right\} \quad (9.11)$$

where λM serves as a complexity penalty controlling the total number of regions.

9.3.3 Recursive Binary Splitting

The optimization proceeds through recursive binary splitting. For a given region R , we consider splitting it into two subregions based on a feature j and split point s :

$$\begin{aligned} R_L(j, s) &= \{x \in R : x_j \leq s\} \\ R_R(j, s) &= \{x \in R : x_j > s\} \end{aligned}$$

The optimal constant prediction within any region R is the mean of the responses:

$$\hat{c}_R = \frac{1}{|I_R|} \sum_{i \in I_R} y_i \quad (9.12)$$

where $I_R = \{i : x_i \in R\}$ is the set of indices for observations in region R .

9.3.4 Split Selection Algorithm

The algorithm for finding the optimal split at each node proceeds through nested loops:

Algorithm 6 Optimal Split Selection

```

1: Input: Region  $R$ , data  $\{(x_i, y_i)\}_{i \in I_R}$ 
2: Output: Optimal feature  $j^*$  and split point  $s^*$ 
3: Initialize  $L^* \leftarrow \infty$  ▷ Best loss so far
4: for  $j = 1$  to  $p$  do ▷ Loop over features
5:   Sort observations in  $R$  by feature  $j$ 
6:   for  $i \in I_R \setminus \{\max(I_R)\}$  do ▷ Loop over potential splits
7:     if  $x_{ij} < x_{i+1,j}$  then ▷ Unique split points only
8:        $s \leftarrow (x_{ij} + x_{i+1,j})/2$ 
9:       Compute  $I_L \leftarrow \{k \in I_R : x_{kj} \leq s\}$ 
10:      Compute  $I_R \leftarrow \{k \in I_R : x_{kj} > s\}$ 
11:       $\hat{c}_L \leftarrow \frac{1}{|I_L|} \sum_{k \in I_L} y_k$ 
12:       $\hat{c}_R \leftarrow \frac{1}{|I_R|} \sum_{k \in I_R} y_k$ 
13:       $L(j, s) \leftarrow \sum_{k \in I_L} (y_k - \hat{c}_L)^2 + \sum_{k \in I_R} (y_k - \hat{c}_R)^2$ 
14:      if  $L(j, s) < L^*$  then
15:         $L^* \leftarrow L(j, s)$ 
16:         $j^* \leftarrow j$ 
17:         $s^* \leftarrow s$ 
18:      end if
19:    end if
20:  end for
21: end for return  $(j^*, s^*)$ 

```

9.3.5 Tree Growing Procedure

The full tree is grown through the following steps:

1. Start with all data in a single region $R_1 = \mathcal{X}$
2. For each current terminal region R_m :
 - (a) Find optimal split (j^*, s^*) using Algorithm 1
 - (b) If the decrease in loss exceeds λ , create two new regions
 - (c) Otherwise, leave R_m as a terminal node
3. Repeat step 2 until no further splits are beneficial

9.3.6 Statistical Properties

For any region R_m , the mean squared error can be decomposed as:

$$\frac{1}{|I_m|} \sum_{i \in I_m} (y_i - \hat{c}_m)^2 = \frac{1}{|I_m|} \sum_{i \in I_m} (y_i - c_m^*)^2 + (c_m^* - \hat{c}_m)^2 \quad (9.13)$$

where $c_m^* = \mathbb{E}[Y|X \in R_m]$ is the conditional expectation in region R_m .

9.4 Least Squares Boosting

9.4.1 Basic Framework

Let's begin with the standard additive model framework but now explicitly incorporating regularization terms:

$$f(x) = \sum_{m=1}^M h_m(x) \quad (9.14)$$

where each base learner $h_m(x)$ is a regression tree that can be written as:

$$h_m(x) = \sum_{j=1}^{J_m} c_{mj} \mathbb{I}\{x \in R_{mj}\} \quad (9.15)$$

Here:

- R_{mj} are disjoint regions partitioning the feature space
- c_{mj} are the coefficients for each region
- J_m is the number of regions in the m -th tree

9.4.2 Regularized Optimization

At iteration m , given the current model $f_{m-1}(x)$, we solve:

$$\min_{h_m} \left\{ \sum_{i=1}^n (r_i - h_m(x_i))^2 + \lambda \sum_{j=1}^{J_m} c_{mj}^2 + \gamma J_m \right\} \quad (9.16)$$

where:

- $r_i = y_i - f_{m-1}(x_i)$ are the current residuals
- λ controls L2 regularization on region coefficients
- γ penalizes the number of regions

9.4.3 Tree Construction with Regularization

For a candidate split that divides region R into R_L and R_R :

1. Optimal coefficients with L2 regularization:

$$c_L^* = \frac{\sum_{i \in R_L} r_i}{|R_L| + \lambda} \quad (9.17)$$

$$c_R^* = \frac{\sum_{i \in R_R} r_i}{|R_R| + \lambda} \quad (9.18)$$

2. Split gain with both regularization terms:

$$\Delta L = \frac{(\sum_{i \in R_L} r_i)^2}{|R_L| + \lambda} + \frac{(\sum_{i \in R_R} r_i)^2}{|R_R| + \lambda} - \frac{(\sum_{i \in R} r_i)^2}{|R| + \lambda} - \gamma \quad (9.19)$$

9.4.4 Extension to Weighted Least Squares

Now let's extend this to include observation weights w_i . The optimization becomes:

$$\min_{h_m} \left\{ \sum_{i=1}^n w_i (r_i - h_m(x_i))^2 + \lambda \sum_{j=1}^{J_m} c_{mj}^2 + \gamma J_m \right\} \quad (9.20)$$

This leads to modified formulas:

1. Optimal coefficients:

$$c_L^* = \frac{\sum_{i \in R_L} w_i r_i}{\sum_{i \in R_L} w_i + \lambda} \quad (9.21)$$

$$c_R^* = \frac{\sum_{i \in R_R} w_i r_i}{\sum_{i \in R_R} w_i + \lambda} \quad (9.22)$$

2. Split gain:

$$\Delta L = \frac{(\sum_{i \in R_L} w_i r_i)^2}{\sum_{i \in R_L} w_i + \lambda} + \frac{(\sum_{i \in R_R} w_i r_i)^2}{\sum_{i \in R_R} w_i + \lambda} - \frac{(\sum_{i \in R} w_i r_i)^2}{\sum_{i \in R} w_i + \lambda} - \gamma \quad (9.23)$$

9.4.5 Complete Algorithm

Algorithm 7 Regularized Weighted L2 Boosting

- 1: **Initialize:** $f_0(x) = 0$
 - 2: **for** $m = 1$ to M **do**
 - 3: Compute residuals: $r_i = y_i - f_{m-1}(x_i)$
 - 4: Fit regularized tree $h_m(x)$:
 1. Start with single region
 2. For each leaf node:
 - Find best split using regularized gain
 - Split if gain > 0
 3. Compute regularized coefficients for each region
 - 5: Update: $f_m(x) = f_{m-1}(x) + h_m(x)$
 - 6: **end for**
 - 7: **return** $f_M(x)$
-

9.4.6 Connection to XGBoost

This formulation directly connects to XGBoost by observing that:

1. For general loss functions, the second-order Taylor expansion leads to a weighted least squares problem where:

- Weights: $w_i = \frac{\partial^2 l}{\partial f^2} \Big|_{f=f_{m-1}(x_i)}$
- Working response: $r_i = -\frac{\partial l / \partial f}{w_i} \Big|_{f=f_{m-1}(x_i)}$

2. The regularization terms remain the same, providing:

- L2 regularization on leaf weights (λ term)
- Complexity penalty on tree structure (γ term)

This formulation shows how L2 boosting with explicit regularization naturally generalizes to XGBoost's framework for arbitrary loss functions through second-order approximation.

9.5 XGBoost for Logistic Regression

9.5.1 The Logistic Model

Consider binary classification where $y_i \in \{0, 1\}$. The logistic model estimates probability $p(x) = P(Y = 1 | X = x)$ through:

$$p(x) = \frac{1}{1 + e^{-f(x)}} \quad (9.24)$$

where $f(x)$ is our additive model:

$$f(x) = \sum_{m=1}^M h_m(x) \quad (9.25)$$

9.5.2 Loss Function Analysis

For each observation (x_i, y_i) , the negative log-likelihood loss is:

$$l(y_i, f) = -y_i \log(p) - (1 - y_i) \log(1 - p) \quad (9.26)$$

$$= -y_i \log\left(\frac{1}{1 + e^{-f}}\right) - (1 - y_i) \log\left(\frac{e^{-f}}{1 + e^{-f}}\right) \quad (9.27)$$

$$= y_i \log(1 + e^{-f}) + (1 - y_i)(f + \log(1 + e^{-f})) \quad (9.28)$$

$$= f(1 - y_i) + \log(1 + e^{-f}) \quad (9.29)$$

The first and second derivatives are:

$$r_i = \frac{\partial l}{\partial f} = (1 - y_i) - \frac{1}{1 + e^f} = p - y_i \quad (9.30)$$

$$w_i = \frac{\partial^2 l}{\partial f^2} = \frac{e^f}{(1 + e^f)^2} = p(1 - p) \quad (9.31)$$

9.5.3 Adding a New Tree

Having computed r_i and w_i , adding a new tree amounts to solving a weighted least squares problem:

$$\min_{h_m} \left\{ \sum_{i=1}^n w_i (z_i - h_m(x_i))^2 + \lambda \sum_{j=1}^{J_m} c_{mj}^2 + \gamma J_m \right\} \quad (9.32)$$

where:

- $z_i = -r_i/w_i = -(p_i - y_i)/(p_i(1 - p_i))$ is the working response
- λ controls L2 regularization on leaf weights
- γ penalizes tree complexity

Tree Structure

The tree $h_m(x)$ takes the form:

$$h_m(x) = \sum_{j=1}^{J_m} c_{mj} \mathbb{I}\{x \in R_{mj}\} \quad (9.33)$$

where:

- R_{mj} are disjoint regions partitioning the feature space
- c_{mj} are the leaf weights
- J_m is the number of leaves

Optimal Leaf Weights

For any region R_j , the optimal weight with regularization is:

$$c_j^* = -\frac{\sum_{i \in R_j} r_i}{\sum_{i \in R_j} w_i + \lambda} = \frac{\sum_{i \in R_j} w_i z_i}{\sum_{i \in R_j} w_i + \lambda} \quad (9.34)$$

This has an intuitive interpretation:

- Numerator sums the weighted working responses
- Denominator includes regularization λ
- More weight (w_i) means more influence on leaf value

Split Finding

For a candidate split dividing region R into R_L and R_R , the gain is:

$$\Delta L = \frac{(\sum_{i \in R_L} r_i)^2}{2(\sum_{i \in R_L} w_i + \lambda)} + \frac{(\sum_{i \in R_R} r_i)^2}{2(\sum_{i \in R_R} w_i + \lambda)} - \frac{(\sum_{i \in R} r_i)^2}{2(\sum_{i \in R} w_i + \lambda)} - \gamma \quad (9.35)$$

The split finding algorithm proceeds as:

Algorithm 8 Finding Optimal Split

```

1: for each feature  $k$  do
2:   Sort instances by feature  $k$  value
3:   for each possible split point  $s$  do
4:     Calculate  $r_L = \sum_{x_{ik} \leq s} r_i$ ,  $w_L = \sum_{x_{ik} \leq s} w_i$ 
5:     Calculate  $r_R = \sum_{x_{ik} > s} r_i$ ,  $w_R = \sum_{x_{ik} > s} w_i$ 
6:     Calculate gain  $\Delta L$  using formula above
7:     if  $\Delta L$  is maximum so far then
8:       Update best split
9:     end if
10:  end for
11: end for

```

Tree Building Process

The complete tree is built recursively:

Algorithm 9 Building Tree h_m

```

1: Start with all data in single region
2: Function BuildTree(Region  $R$ ):
3: Find best split of  $R$  using Algorithm 1
4: if gain  $> 0$  then
5:   Split  $R$  into  $R_L$  and  $R_R$ 
6:   BuildTree( $R_L$ )
7:   BuildTree( $R_R$ )
8: else
9:   Make  $R$  a leaf with weight  $c_R^*$ 
10: end if

```

Practical Considerations

Additional constraints are often added:

- Maximum tree depth d
- Minimum sum of weights in each node
- Maximum number of leaves

These help prevent overfitting and ensure computational efficiency.

9.5.4 Geometric Interpretation

The derivatives provide crucial geometric information:

1. **First Derivative** (r_i):

- Represents the slope of the loss function
- $r_i = p - y_i$ is the prediction error
- Large magnitude indicates we're far from optimal
- Sign indicates direction of needed adjustment

2. **Second Derivative** (w_i):

- Represents the curvature of the loss function
- $w_i = p(1 - p)$ is highest at $p = 0.5$ (equals 0.25)
- Approaches 0 as p approaches 0 or 1
- Indicates how "certain" our current prediction is

9.5.5 The Golf Analogy

This interplay between r_i and w_i parallels different stages of a golf game:

1. **Early Predictions** ($p \approx 0.5$):

- Large $w_i \approx 0.25$ means high curvature
- Working response $z_i = -r_i/w_i$ is moderate
- Like a mid-range shot: balanced distance and precision

2. **Very Wrong Predictions** ($p \approx 0$ for $y_i = 1$ or $p \approx 1$ for $y_i = 0$):

- Small $w_i \approx 0$ means low curvature
- Large working response due to small denominator
- Like the first stroke: need big correction, less precision

3. **Nearly Correct Predictions** ($p \approx y_i$):

- Moderate w_i but very small r_i
- Small working response due to small numerator
- Like the final putt: small, precise adjustments

9.5.6 Connection to Error Back-propagation

The gradient r_i in XGBoost is analogous to the error term e_i in neural network back-propagation:

1. **Neural Networks:**

- Error e_i is back-propagated through layers
- Weights are updated based on back-propagated error
- Each layer learns to reduce the propagated error

2. **XGBoost:**

- Gradient r_i is the "error" to be corrected
- New tree $h_m(x)$ learns to predict $-r_i/w_i$
- Each tree reduces the residual error

9.5.7 Tree Learning as Back-propagation

When fitting a new tree $h_m(x)$, we are effectively:

1. Computing the error signal: $r_i = p_i - y_i$
2. Scaling by curvature: $z_i = -r_i/w_i$
3. Learning a tree to predict this scaled error
4. Updating the model: $f_m = f_{m-1} + h_m$

This is analogous to neural network training where:

1. Forward pass computes predictions
2. Back-propagation computes error gradients
3. Network weights are updated to reduce error

The key difference is that instead of updating weights in a fixed architecture, we:

- Grow a new tree optimized for current errors
- Scale updates by local curvature (w_i)
- Add the tree to our ensemble

This interpretation unifies gradient boosting with classical back-propagation, viewing both as iterative error correction methods in different architectural paradigms.

I'll convert the content into LaTeX format, creating a proper subsection that would fit seamlessly into your chapter.

9.5.8 Connection to Iterative Reweighted Least Squares

XGBoost's treatment of logistic regression has deep connections to the classical iterative reweighted least squares (IRLS) algorithm. This connection helps explain why XGBoost's second-order approximation is particularly effective and provides additional insight into its optimization strategy.

IRLS for Regular Logistic Regression

In standard logistic regression, we minimize the negative log-likelihood:

$$L(\beta) = \sum_{i=1}^n [-y_i x_i^T \beta + \log(1 + e^{x_i^T \beta})] \quad (9.36)$$

The Newton-Raphson update takes the form:

$$\beta^{(t+1)} = \beta^{(t)} - [H(\beta^{(t)})]^{-1} g(\beta^{(t)}) \quad (9.37)$$

where:

- $g(\beta) = \sum_{i=1}^n (p_i - y_i) x_i$ is the gradient
- $H(\beta) = \sum_{i=1}^n p_i (1 - p_i) x_i x_i^T$ is the Hessian
- $p_i = \frac{1}{1 + e^{-x_i^T \beta}}$ is the predicted probability

This can be rewritten as a weighted least squares problem:

$$\beta^{(t+1)} = \arg \min_{\beta} \sum_{i=1}^n w_i^{(t)} (z_i^{(t)} - x_i^T \beta)^2 \quad (9.38)$$

where:

- $w_i^{(t)} = p_i^{(t)} (1 - p_i^{(t)})$ are the weights
- $z_i^{(t)} = x_i^T \beta^{(t)} + \frac{y_i - p_i^{(t)}}{p_i^{(t)} (1 - p_i^{(t)})}$ is the working response

XGBoost as Functional IRLS

XGBoost generalizes this idea to function space. Instead of updating linear coefficients β , it updates the function f by adding trees:

$$f_{m+1}(x) = f_m(x) + h_m(x) \quad (9.39)$$

The second-order Taylor expansion of the loss around f_m gives:

$$L(f_m + h) \approx \sum_{i=1}^n [l(y_i, f_m(x_i)) + g_i h_m(x_i) + \frac{1}{2} h_i h_m(x_i)^2] \quad (9.40)$$

where:

- $g_i = p_i - y_i$ is the gradient
- $h_i = p_i(1 - p_i)$ is the Hessian diagonal

This leads to the weighted least squares problem for finding the new tree:

$$h_m = \arg \min_h \sum_{i=1}^n w_i (z_i - h(x_i))^2 + \Omega(h) \quad (9.41)$$

where:

- $w_i = p_i(1 - p_i)$ are the weights (exactly as in IRLS)
- $z_i = -\frac{g_i}{h_i} = -\frac{p_i - y_i}{p_i(1 - p_i)}$ is the working response
- $\Omega(h)$ is the regularization on tree complexity

Theoretical Insights

This connection reveals several important theoretical insights:

1. **Functional Generalization:** XGBoost extends IRLS from parameter space to function space, replacing linear updates with tree additions. The fundamental structure of the optimization remains the same, but the search space becomes much richer.
2. **Weight Interpretation:** The weights $w_i = p_i(1 - p_i)$ maintain their classical interpretation:
 - Small for confident predictions (p_i near 0 or 1)
 - Large for uncertain predictions (p_i near 0.5)
 - Automatically handle extreme probabilities
3. **Working Response:** The working response z_i represents the desired shift in the function value:
 - Magnitude increases for misclassified points
 - Direction determined by $y_i - p_i$
 - Naturally scaled by prediction certainty
4. **Regularization:** XGBoost adds explicit regularization $\Omega(h)$ that:
 - Controls tree complexity
 - Prevents overfitting

Implementation Benefits

This IRLS perspective explains several practical advantages of XGBoost:

1. **Numerical Stability:** Like IRLS, the method naturally handles extreme probabilities through the weighting scheme, preventing numerical instabilities in updates.
2. **Efficient Updates:** The quadratic approximation allows efficient tree building through weighted least squares, making the optimization computationally tractable.
3. **Natural Scaling:** The Hessian weights provide automatic scaling of updates based on prediction confidence, similar to the role of the Fisher information in classical maximum likelihood estimation.
4. **Interpretable Steps:** Each tree addition can be viewed as a Newton step in function space, providing theoretical guarantees on convergence similar to those in classical IRLS.

This connection to IRLS provides both theoretical understanding and practical insights into XGBoost's effectiveness for logistic regression tasks, while the extension to function space and the addition of regularization terms explain its superior performance compared to classical approaches.

9.6 Surrogate Loss Functions and Incremental Learning

9.6.1 The Role of Surrogate Losses

Many machine learning algorithms can be understood through the lens of surrogate loss functions - simpler, more tractable functions that we optimize in place of our original objective. This perspective provides another unifying view of gradient descent, boosting, and other incremental learning methods.

9.6.2 Gradient Descent as Surrogate Minimization

Consider minimizing a function $L(\theta)$. At each iteration, gradient descent can be viewed as minimizing a quadratic surrogate function:

$$Q_t(\theta) = L(\theta_t) + \nabla L(\theta_t)^T(\theta - \theta_t) + \frac{1}{2\eta} \|\theta - \theta_t\|^2 \quad (9.42)$$

The three terms in this surrogate have clear interpretations:

- $L(\theta_t)$ is the current loss value
- $\nabla L(\theta_t)^T(\theta - \theta_t)$ provides the first-order approximation
- $\frac{1}{2\eta} \|\theta - \theta_t\|^2$ is a proximal term limiting the step size

The minimizer of this quadratic surrogate is precisely the gradient descent update:

$$\theta_{t+1} = \arg \min_{\theta} Q_t(\theta) = \theta_t - \eta \nabla L(\theta_t) \quad (9.43)$$

This shows that gradient descent can be interpreted as constructing a surrogate quadratic function at each step, finding its minimum exactly, and using that minimum as the next iterate.

9.6.3 Boosting and Surrogate Losses

Different boosting algorithms can be understood through their choice of surrogate loss functions. The most common choices include:

- AdaBoost: Uses exponential loss

$$L_{\text{exp}}(y, f) = \exp(-yf) \quad (9.44)$$

- LogitBoost: Uses logistic loss

$$L_{\text{log}}(y, f) = \log(1 + \exp(-yf)) \quad (9.45)$$

- L2Boosting: Uses squared error loss

$$L_2(y, f) = \frac{1}{2}(y - f)^2 \quad (9.46)$$

9.6.4 XGBoost's Second-Order Surrogate

XGBoost takes this further by using a second-order surrogate function. At each step, for adding a new tree $h_m(x)$, it minimizes:

$$Q_m(h) = \sum_{i=1}^n [g_i h(x_i) + \frac{1}{2} h_i h(x_i)^2] + \Omega(h) \quad (9.47)$$

where:

- $g_i = \partial_{f_{m-1}} l(y_i, f_{m-1}(x_i))$ is the gradient
- $h_i = \partial_{f_{m-1}}^2 l(y_i, f_{m-1}(x_i))$ is the Hessian
- $\Omega(h)$ is a regularization term

This surrogate provides several advantages: it incorporates second-order information, admits efficient optimization for tree structure, and provides natural regularization.

9.6.5 A Unified View Through Surrogate Functions

These methods share a common structure. The original problem seeks to minimize some loss:

$$\min_f L(f) \tag{9.48}$$

The iterative solution at each step t takes the form:

$$f_{t+1} = f_t + \arg \min_h Q_t(h) \tag{9.49}$$

where Q_t is a surrogate function. Good surrogate functions share several key properties:

- Upper bound the original loss: $Q_t(h) \geq L(f_t + h)$
- Match at the current point: $Q_t(0) = L(f_t)$
- Are easier to optimize than the original loss

9.6.6 Design Principles for Surrogate Functions

When designing surrogate functions, several key principles must be balanced:

1. **Approximation Quality** The surrogate should:

- Closely approximate the original loss
- Provide useful gradients for optimization
- Capture key properties like convexity

2. **Optimization Ease** The surrogate should have:

- Simple analytical form
- Efficient minimization algorithms
- Well-behaved derivatives

3. **Statistical Properties** The surrogate should ensure:

- Proper scoring rules
- Fisher consistency
- Robustness to noise

9.6.7 Connection to Earlier Sections

This surrogate function perspective complements our earlier view of incremental learning. Consider how each method employs both perspectives:

1. Gradient Descent

- Incremental view: Parameter updates
- Surrogate view: Quadratic approximation

2. Decision Trees

- Incremental view: Region splitting
- Surrogate view: Piecewise approximation

3. Boosting

- Incremental view: Function addition
- Surrogate view: Loss upper bound

Understanding both perspectives helps inform algorithm design, hyperparameter tuning, convergence analysis, and performance optimization. This dual view reveals the deep connections between seemingly different approaches to machine learning, all unified by the principles of incremental improvement and surrogate optimization.

9.7 The “Lazy” Nature of Boosting and Implicit Regularization

Boosting methods, particularly gradient boosting with small step sizes, exhibit a surprising resistance to overfitting despite the capacity to create arbitrarily complex functions. This behavior parallels the success of overparametrized neural networks trained with gradient descent, and both can be understood through the lens of algorithmic “laziness”.

9.7.1 Gradient Flow and Function Space

Consider the continuous-time gradient flow in function space:

$$\frac{\partial f_t}{\partial t} = -\nabla L(f_t) \tag{9.50}$$

For both neural networks and boosting:

- The space of possible functions is extremely rich
- Many different solutions can achieve zero training error
- The actual trajectory taken by the algorithm matters

9.7.2 The Principle of Least Action

Gradient-based methods follow a “lazy” path in the sense that they:

1. Take the path of steepest descent locally
2. Make minimal changes necessary to reduce the loss
3. Implicitly prefer “simpler” functions early in training

For boosting, this manifests in several ways:

$$f_{m+1} = f_m - \eta \nabla L(f_m) \quad (9.51)$$

where:

- Small learning rate η ensures small steps
- Each tree focuses on current residuals
- Early trees capture “easy” patterns

9.7.3 Spectral Bias in Function Learning

Both boosting and neural networks exhibit a form of spectral bias:

1. Low-Frequency First:

- Early iterations learn smooth, global patterns
- Complex, high-frequency components emerge later
- This ordering provides implicit regularization

2. Mathematical Characterization: If we decompose the target function in frequency components:

$$f^*(x) = \sum_k a_k \phi_k(x) \quad (9.52)$$

The learning dynamics tend to prioritize lower-frequency basis functions $\phi_k(x)$ earlier in training.

9.7.4 Early Stopping as Complexity Control

The “lazy” nature of boosting means that:

$$\text{Complexity}(f_m) \approx O(m\eta) \quad (9.53)$$

This leads to several important consequences:

1. Early stopping effectively controls model complexity
2. The number of iterations acts as a complexity penalty
3. The learning rate η scales the complexity growth

9.7.5 Implicit Regularization Through Optimization

The optimization algorithm itself provides regularization through:

1. **Incremental Learning:**

$$\Delta f_m = -\eta \nabla L(f_{m-1}) \quad (9.54)$$

Each step makes minimal changes needed to reduce the loss. Copy

2. **Local Smoothness:**

$$\|\Delta f_m\|_{\mathcal{H}} \leq \eta \|\nabla L(f_{m-1})\|_{\mathcal{H}} \quad (9.55)$$

Changes are bounded in an appropriate function space norm.

3. **Gradient Flow Path:** The discrete updates approximate a continuous flow:

$$f_m \approx f_0 - \eta \int_0^m \nabla L(f_t) dt \quad (9.56)$$

9.7.6 Comparison with Neural Networks

The parallels with neural networks include:

Property	Boosting	Neural Networks
Space	Function space	Parameter space
Updates	Add new trees	Adjust weights
Laziness	Small learning rate	Small learning rate
Early learning	Global patterns	Low-frequency components
Late learning	Details	High-frequency components

Table 9.1: Comparison of lazy learning properties

9.7.7 Practical Implications

This theoretical understanding suggests several practical guidelines:

1. Use small learning rates to promote "lazy" learning
2. Monitor validation performance for early stopping
3. Expect early iterations to capture main effects
4. Allow longer training for fine-grained patterns

The "lazy" nature of boosting thus provides an elegant explanation for its empirical success, connecting it to broader principles of implicit regularization in modern machine learning. This perspective helps explain why boosting can work well even without explicit regularization, particularly when combined with early stopping.

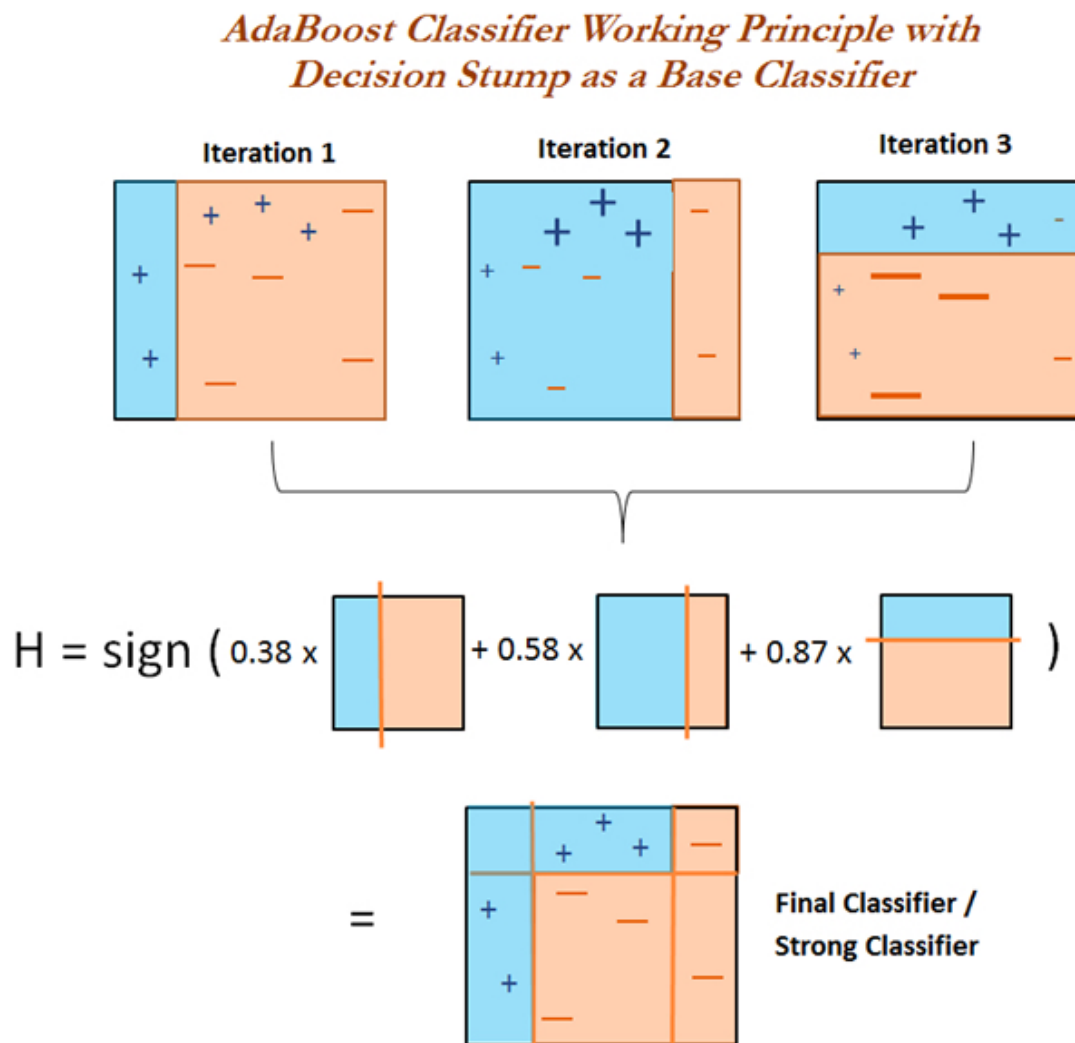


Figure 9.5: Adaboost

9.8 AdaBoost

9.8.1 The Exponential Loss Framework

Consider binary classification where $y_i \in \{-1, 1\}$. We seek an additive model of the form:

$$f_M(x) = \sum_{m=1}^M \alpha_m h_m(x) \quad (9.57)$$

where $h_m(x) \in \{-1, 1\}$ are base classifiers and α_m are their weights. AdaBoost minimizes the exponential loss:

$$L(f) = \frac{1}{n} \sum_{i=1}^n \exp(-y_i f(x_i)) \quad (9.58)$$

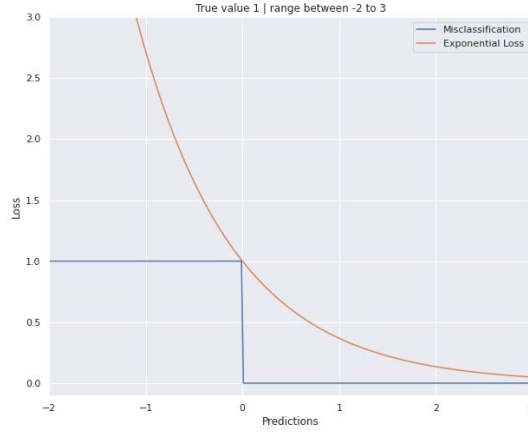


Figure 9.6: Exponential loss upper bounds 0/1 loss

9.8.2 Properties of Exponential Loss

For binary classification where $y \in \{-1, 1\}$, AdaBoost minimizes the exponential loss:

$$L(f) = \frac{1}{n} \sum_{i=1}^n \exp(-y_i f(x_i)) \quad (9.59)$$

Relationship to Zero-One Loss

The zero-one loss is defined as:

$$L_{0-1}(f) = \frac{1}{n} \sum_{i=1}^n \mathbb{I}\{y_i f(x_i) < 0\} \quad (9.60)$$

The exponential loss has several important relationships with zero-one loss:

Proposition 37 (Upper Bound). *For any $y \in \{-1, 1\}$ and $f \in \mathbb{R}$:*

$$\mathbb{I}\{yf < 0\} \leq \exp(-yf) \quad (9.61)$$

Proof. Consider two cases:

- If $yf \geq 0$: Then $\mathbb{I}\{yf < 0\} = 0 \leq \exp(-yf)$
- If $yf < 0$: Then $\mathbb{I}\{yf < 0\} = 1 < \exp(-yf)$

Therefore, exponential loss always upper bounds zero-one loss. \square

Smoothness Properties

The exponential loss has several advantageous properties:

1. **Differentiability:** The loss is infinitely differentiable with:

$$\frac{\partial}{\partial f} \exp(-yf) = -y \exp(-yf) \quad (9.62)$$

$$\frac{\partial^2}{\partial f^2} \exp(-yf) = \exp(-yf) \quad (9.63)$$

2. **Strong Convexity:** The second derivative is always positive:

$$\frac{\partial^2}{\partial f^2} \exp(-yf) = \exp(-yf) > 0 \quad (9.64)$$

3. **Gradient Magnitude:** For misclassified points ($yf < 0$):

$$\left| \frac{\partial}{\partial f} \exp(-yf) \right| = \exp(-yf) > 1 \quad (9.65)$$

This ensures strong gradients for correcting mistakes.

Statistical Implications

The properties of exponential loss lead to important consequences:

1. **Population Minimizer:** The minimizer of expected exponential loss is:

$$f^*(x) = \frac{1}{2} \log \frac{P(Y = 1|X = x)}{P(Y = -1|X = x)} \quad (9.66)$$

which is a monotone transformation of the optimal Bayes classifier.

2. **Margin Maximization:** The exponential penalty on negative margins encourages:

- Correct classification ($yf > 0$)
- Large margins (large $|f|$ when confident)
- Focus on hard examples (large loss when wrong)

These properties help explain why AdaBoost often shows good generalization despite optimizing a seemingly aggressive loss function. The exponential loss provides:

- A computationally tractable upper bound on misclassification error
- Smooth gradients for optimization
- Strong penalties that drive margin maximization
- Theoretical connection to optimal Bayes classifier

9.8.3 Forward Stagewise Additive Modeling

At iteration m , given current model $f_{m-1}(x)$, we seek:

$$(\alpha_m, h_m) = \arg \min_{\alpha, h} \sum_{i=1}^n \exp(-y_i[f_{m-1}(x_i) + \alpha h(x_i)]) \quad (9.67)$$

This can be rewritten as:

$$\sum_{i=1}^n w_i^m \exp(-\alpha y_i h(x_i)) \quad (9.68)$$

where $w_i^m = \exp(-y_i f_{m-1}(x_i))$ are the current weights.

9.8.4 Optimal Base Classifier

For fixed α , the optimal h_m minimizes:

$$\sum_{i=1}^n w_i^m \exp(-\alpha y_i h(x_i)) \quad (9.69)$$

Since $h(x_i) \in \{-1, 1\}$, this is equivalent to minimizing:

$$e^\alpha \sum_{y_i \neq h(x_i)} w_i^m + e^{-\alpha} \sum_{y_i = h(x_i)} w_i^m \quad (9.70)$$

Define the weighted error:

$$\epsilon_m = \frac{\sum_{y_i \neq h_m(x_i)} w_i^m}{\sum_{i=1}^n w_i^m} \quad (9.71)$$

9.8.5 Optimal Weight Coefficient

Given h_m , differentiating with respect to α yields:

$$\frac{\partial}{\partial \alpha} [e^\alpha \epsilon_m + e^{-\alpha} (1 - \epsilon_m)] = 0 \quad (9.72)$$

Solving this equation:

$$\alpha_m = \frac{1}{2} \log \left(\frac{1 - \epsilon_m}{\epsilon_m} \right) \quad (9.73)$$

9.8.6 Weight Update Rule

The weights for the next iteration become:

$$w_i^{m+1} = \exp(-y_i f_m(x_i)) \quad (9.74)$$

$$= \exp(-y_i [f_{m-1}(x_i) + \alpha_m h_m(x_i)]) \quad (9.75)$$

$$= w_i^m \exp(-\alpha_m y_i h_m(x_i)) \quad (9.76)$$

After normalization:

$$w_i^{m+1} \leftarrow \frac{w_i^{m+1}}{\sum_{j=1}^n w_j^{m+1}} \quad (9.77)$$

9.8.7 The Complete Algorithm

These derivations lead naturally to the AdaBoost algorithm:

Algorithm 10 AdaBoost via Exponential Loss Minimization

- 1: **Initialize:** $w_i^1 = \frac{1}{n}$ for $i = 1, \dots, n$
 - 2: **for** $m = 1$ to M **do**
 - 3: Fit $h_m(x)$ to minimize weighted error using weights w_i^m
 - 4: Compute $\epsilon_m = \sum_{i=1}^n w_i^m \mathbb{I}\{y_i \neq h_m(x_i)\} / \sum_{i=1}^n w_i^m$
 - 5: Set $\alpha_m = \frac{1}{2} \log \left(\frac{1-\epsilon_m}{\epsilon_m} \right)$
 - 6: Update $w_i^{m+1} = w_i^m \exp(-\alpha_m y_i h_m(x_i))$
 - 7: Normalize weights
 - 8: **end for**
 - 9: **return** $f_M(x) = \text{sign} \left(\sum_{m=1}^M \alpha_m h_m(x) \right)$
-

9.8.8 Comparison with XGBoost

Loss Functions and Their Properties

The fundamental difference between AdaBoost and XGBoost begins with their loss functions:

1. AdaBoost: Exponential Loss

$$L_{\text{Ada}}(y, f) = \exp(-yf) \quad (9.78)$$

2. XGBoost: Logistic Loss (for binary classification)

$$L_{\text{XGB}}(y, f) = \log(1 + \exp(-yf)) \quad (9.79)$$

Key implications:

- AdaBoost's exponential loss grows exponentially with negative margins, making it more sensitive to outliers
- XGBoost's logistic loss is bounded, providing better robustness
- XGBoost's second derivative allows Newton-like updates

Aspect	AdaBoost	XGBoost
Base Learner Type	Classification Trees	Regression Trees
Tree Depth	Usually stumps (depth=1)	Typically 3-6 levels
Node Prediction	$\{-1, 1\}$	Real-valued
Update Strategy	Multiplicative weights	Newton step

Table 9.2: Comparison of tree characteristics

Feature	AdaBoost	XGBoost
Regularization	Implicit through early stopping	Explicit L1/L2 terms plus tree structure penalties
Parallelization	Limited	Extensive (feature parallel, node parallel)
Memory Usage	Lower (simpler trees)	Higher (deeper trees, gradient info)
Hyperparameters	Few (mainly iterations and learning rate)	Many (tree depth, regularization, sampling rates, etc.)
Typical Use Cases	Simple binary classification	General-purpose ML (classification, regression, ranking)

Table 9.3: Modern implementation comparison

Base Learners and Their Updates

Modern Implementation and Usage

XGBoost is generally more suitable for modern machine learning applications, especially when:

- Dataset is large or high-dimensional
- Problem requires fine-tuned performance
- Computing resources allow for parallel processing
- Problem involves missing data or sparse features

However, AdaBoost remains valuable when:

- Interpretability is crucial (due to simpler trees)
- Computing resources are limited
- Problem is a straightforward binary classification
- Quick prototyping is needed

9.9 Random Forests

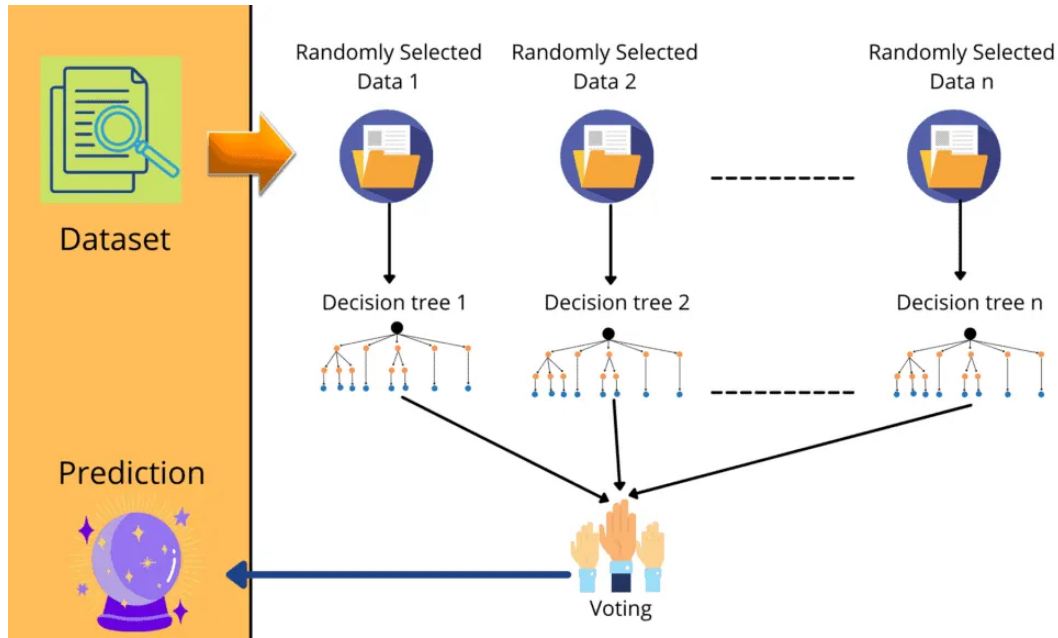


Figure 9.7: Random forest

9.9.1 Ensemble Framework

While boosting builds an additive ensemble sequentially, Random Forests construct an ensemble of trees in parallel:

$$f_M(x) = \frac{1}{M} \sum_{m=1}^M T_m(x) \quad (9.80)$$

where $\{T_m\}_{m=1}^M$ are individual decision trees, each built on a bootstrap sample of the training data with a randomized feature selection process.

9.9.2 Sources of Randomization

Bootstrap Aggregating (Bagging)

For each tree m :

- Draw bootstrap sample $\mathcal{D}_m = \{(x_i^*, y_i^*)\}_{i=1}^n$ from training data
- Probability of observation i being selected:

$$P(i \in \mathcal{D}_m) = 1 - \left(1 - \frac{1}{n}\right)^n \approx 0.632 \quad (9.81)$$

Random Feature Selection

At each node split:

- Randomly sample k features from the full set of p features
- Typical choices:

$$k = \begin{cases} \lfloor \sqrt{p} \rfloor & \text{for classification} \\ \lfloor p/3 \rfloor & \text{for regression} \end{cases} \quad (9.82)$$

9.9.3 Tree Construction

For each tree T_m :

Algorithm 11 Random Forest Tree Construction

- 1: Draw bootstrap sample \mathcal{D}_m
- 2: Initialize: single node tree
- 3: **while** nodes can be split **do**
- 4: **for** each terminal node R **do**
- 5: Sample k features randomly
- 6: Find best split s^* among k features:

$$s^* = \arg \min_s \left[\sum_{i \in R_L(s)} (y_i - \bar{y}_L)^2 + \sum_{i \in R_R(s)} (y_i - \bar{y}_R)^2 \right] \quad (9.83)$$

- 7: Split node if improvement exceeds threshold
 - 8: **end for**
 - 9: **end while**
-

9.9.4 Statistical Properties

Variance Reduction

For i.i.d. trees with variance σ^2 , the ensemble variance is:

$$\text{Var}(f_M) = \rho \sigma^2 + \frac{1 - \rho}{M} \sigma^2 \quad (9.84)$$

where ρ is the correlation between trees.

Out-of-Bag Error Estimation

For observation i , let \mathcal{M}_i be the set of trees not containing i in their bootstrap sample:

$$\text{OOB}_i = \frac{1}{|\mathcal{M}_i|} \sum_{m \in \mathcal{M}_i} T_m(x_i) \quad (9.85)$$

The OOB error estimate:

$$\text{Err}_{\text{OOB}} = \frac{1}{n} \sum_{i=1}^n L(y_i, \text{OOB}_i) \quad (9.86)$$

9.9.5 Variable Importance Measures

Mean Decrease in Impurity (MDI)

For variable j :

$$\text{MDI}_j = \frac{1}{M} \sum_{m=1}^M \sum_{t \in T_m} p(t) \Delta i(s_t) \mathbb{I}(v(s_t) = j) \quad (9.87)$$

where:

- $p(t)$ is the proportion of samples reaching node t
- $\Delta i(s_t)$ is the impurity decrease at split s_t
- $v(s_t)$ is the variable used in split s_t

Mean Decrease in Accuracy (MDA)

For variable j :

$$\text{MDA}_j = \text{Err}_{\text{OOB}} - \text{Err}_{\text{OOB}}^j \quad (9.88)$$

where $\text{Err}_{\text{OOB}}^j$ is the OOB error after randomly permuting variable j .

9.9.6 Theoretical Results

Consistency

Under suitable conditions on the individual trees:

Theorem 38 (Random Forest Consistency). *As $n, M \rightarrow \infty$:*

$$\mathbb{E}[(f_M(X) - f^*(X))^2] \rightarrow 0 \quad (9.89)$$

where f^* is the true regression function.

Rate of Convergence

For regression with bounded response:

Theorem 39 (Convergence Rate). *With high probability:*

$$\|f_M - f^*\|_2^2 = O\left(\frac{\log n}{n}\right) \quad (9.90)$$

provided the trees are grown to an appropriate depth.

Aspect	Random Forest	AdaBoost	XGBoost
Ensemble Building	Parallel	Sequential	Sequential
Tree Independence	Independent	Dependent	Dependent
Weight Updates	Equal weights	Exponential	Gradient-based
Overfitting Risk	Low	Moderate	Moderate
Parameter Tuning	Simple	Moderate	Complex

Table 9.4: Comparison of tree ensemble methods

9.9.7 Comparison with Boosting Methods

9.9.8 Implementation Considerations

1. Key Parameters:

- Number of trees M
- Number of features k at each split
- Minimum node size
- Maximum tree depth

2. Computational Aspects:

- Trivially parallelizable across trees
- Memory requirements scale linearly with M
- Can be updated online with new trees

Chapter 10

Support Vector Machine

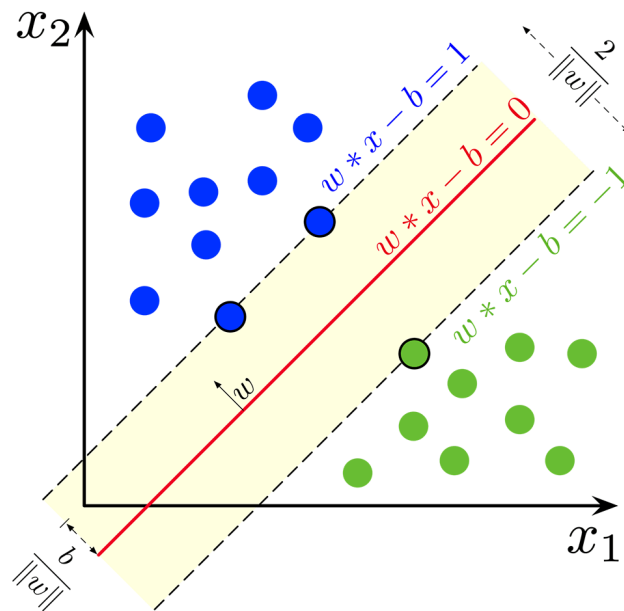


Figure 10.1: SVM

Chapter Overview

Support Vector Machines (SVM) provide a geometric approach to classification by seeking a maximum-margin separating hyperplane between classes, which can be understood as finding the minimal distance between class convex hulls. The method begins with a simple geometric intuition of projecting data onto a direction vector and maximizing the separation between classes, which leads to a convex quadratic optimization problem. Through the introduction of slack variables and the hinge loss, SVMs elegantly handle non-separable data, while the kernel trick enables nonlinear classification by implicitly mapping data to a high-dimensional feature space where linear separation becomes possible. The optimization problem can be solved in its dual form, revealing that the solution depends only on support

vectors (points on or violating the margin), making the method computationally efficient for sparse solutions. This framework combines several fundamental machine learning principles: geometric intuition, convex optimization, kernel methods, and regularization in reproducing kernel Hilbert spaces, which continue to influence modern machine learning approaches despite limitations in scaling to very large datasets.

10.1 Primal Problem: Max Margin

10.1.1 The Geometric Intuition

Consider a binary classification problem with data points $\{(x_i, y_i)\}_{i=1}^n$ where $x_i \in \mathbb{R}^p$ and $y_i \in \{-1, +1\}$. Instead of immediately introducing a separating hyperplane, let us begin with a simpler geometric concept:

Definition 40 (Linear Separability via Projection). Two sets of points are linearly separable if there exists a direction (unit vector) $u \in \mathbb{R}^p$, $\|u\| = 1$, such that their projections onto u are completely separated.

For any unit vector u , define:

$$a_+ = \min_{i:y_i=+1} \langle x_i, u \rangle \quad (10.1)$$

$$a_- = \max_{i:y_i=-1} \langle x_i, u \rangle \quad (10.2)$$

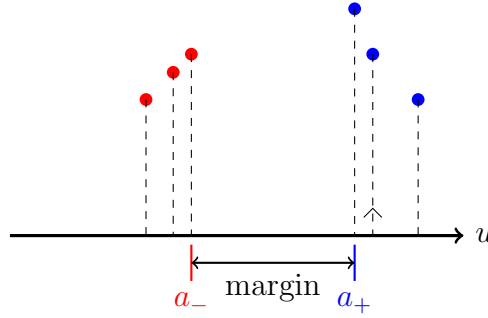


Figure 10.2: Projection onto unit vector u . The margin $a_+ - a_-$ is the separation between the minimum projection of positive (blue) points and maximum projection of negative (red) points.

10.1.2 The Separation Problem

The data is linearly separable if and only if there exists a unit vector u such that:

$$a_- < a_+ \quad (10.3)$$

The quality of separation can be measured by the gap:

$$\text{margin} = a_+ - a_- \quad (10.4)$$

This leads to a natural optimization problem:

$$\max_{\|u\|=1} (a_+ - a_-) \quad (10.5)$$

10.1.3 Connection to Standard SVM Formulation

Let us define the midpoint and half-margin:

$$\bar{a} = \frac{a_+ + a_-}{2} \quad (\text{midpoint}) \quad (10.6)$$

$$\Delta = a_+ - \bar{a} = \bar{a} - a_- \quad (\text{half-margin}) \quad (10.7)$$

To transform our geometric formulation into the standard SVM, we set:

$$w = \frac{u}{\Delta} \quad (10.8)$$

$$b = -\frac{\bar{a}}{\Delta} \quad (10.9)$$

This construction has several important properties:

- For any point x with projection $a = \langle x, u \rangle$:

$$\langle w, x \rangle + b = \frac{a}{\Delta} - \frac{\bar{a}}{\Delta} = \frac{a - \bar{a}}{\Delta}$$

- For positive points: $a \geq a_+$, so:

$$\langle w, x \rangle + b \geq \frac{a_+ - \bar{a}}{\Delta} = 1$$

- For negative points: $a \leq a_-$, so:

$$\langle w, x \rangle + b \leq \frac{a_- - \bar{a}}{\Delta} = -1$$

- The norm of w is inversely proportional to the margin:

$$\|w\| = \frac{\|u\|}{\Delta} = \frac{1}{\Delta}$$

Therefore, maximizing the margin 2Δ is equivalent to minimizing $\|w\|$, or more conveniently, minimizing $\frac{1}{2}\|w\|^2$. The squared norm is preferred because:

- It avoids the square root in the objective function
- Its derivatives are simpler (linear in w)
- It maintains the convexity of the optimization problem

This gives us the standard primal SVM problem:

$$\min_{w,b} \quad \frac{1}{2}\|w\|^2 \quad (10.10)$$

$$\text{s.t.} \quad y_i(\langle w, x_i \rangle + b) \geq 1, \quad i = 1, \dots, n \quad (10.11)$$

The constraints ensure that all points are correctly classified with a margin of at least $1/\|w\|$, while minimizing $\|w\|^2$ maximizes this margin.

10.2 From Primal to Dual: MinMax = MaxMin

10.2.1 The Lagrangian Formulation

Starting from our primal problem:

$$\min_{w,b} \quad \frac{1}{2} \|w\|^2 \quad (10.12)$$

$$\text{s.t.} \quad y_i(\langle w, x_i \rangle + b) \geq 1, \quad i = 1, \dots, n \quad (10.13)$$

We introduce Lagrange multipliers $\alpha_i \geq 0$ for each constraint:

$$\mathcal{L}(w, b, \alpha) = \frac{1}{2} \|w\|^2 - \sum_{i=1}^n \alpha_i [y_i(\langle w, x_i \rangle + b) - 1] \quad (10.14)$$

10.2.2 The Minimax Problem

The primal problem is equivalent to:

$$\min_{w,b} \max_{\alpha \geq 0} \mathcal{L}(w, b, \alpha) \quad (10.15)$$

Under strong duality (which holds due to Slater's condition), this equals:

$$\max_{\alpha \geq 0} \min_{w,b} \mathcal{L}(w, b, \alpha) \quad (10.16)$$

10.2.3 Equivalence of Max-Min Lagrangian to Primal Problem

The equivalence between the primal problem and its Lagrangian dual formulation relies on strong duality. Let's establish this rigorously.

Theorem 41 (Equivalence of Primal and Max-Min Lagrangian). *For the SVM optimization problem, the following equality holds:*

$$\min_{w,b} \frac{1}{2} \|w\|^2 \quad \text{s.t.} \quad y_i(\langle w, x_i \rangle + b) \geq 1 = \min_{w,b} \max_{\alpha \geq 0} \mathcal{L}(w, b, \alpha) \quad (10.17)$$

where $\mathcal{L}(w, b, \alpha) = \frac{1}{2} \|w\|^2 - \sum_{i=1}^n \alpha_i [y_i(\langle w, x_i \rangle + b) - 1]$

Proof. First, observe that for any feasible (w, b) :

$$\max_{\alpha \geq 0} \mathcal{L}(w, b, \alpha) = \begin{cases} \frac{1}{2} \|w\|^2 & \text{if } y_i(\langle w, x_i \rangle + b) \geq 1 \ \forall i \\ +\infty & \text{otherwise} \end{cases} \quad (10.18)$$

This is because:

- If any constraint is violated, say $y_k(\langle w, x_k \rangle + b) < 1$, then α_k can be made arbitrarily large, making $\mathcal{L} \rightarrow +\infty$

- If all constraints are satisfied, the maximum occurs at $\alpha_i = 0$ for all i

Therefore:

$$\min_{w,b} \max_{\alpha \geq 0} \mathcal{L}(w, b, \alpha) = \min_{w,b} \begin{cases} \frac{1}{2} \|w\|^2 & \text{if } y_i(\langle w, x_i \rangle + b) \geq 1 \ \forall i \\ +\infty & \text{otherwise} \end{cases} \quad (10.19)$$

$$= \min_{w,b} \frac{1}{2} \|w\|^2 \quad \text{s.t.} \quad y_i(\langle w, x_i \rangle + b) \geq 1 \quad (10.20)$$

The last equality holds because any (w, b) violating the constraints would give an infinite value, and thus cannot be optimal. \square

Remark 42. This equivalence is crucial because it:

1. Justifies the use of the Lagrangian formulation
2. Shows that the constraints are properly enforced
3. Enables the transition to the dual problem

Corollary 43 (Strong Duality). *Under Slater's condition (which holds for SVM), we have:*

$$\min_{w,b} \max_{\alpha \geq 0} \mathcal{L}(w, b, \alpha) = \max_{\alpha \geq 0} \min_{w,b} \mathcal{L}(w, b, \alpha) \quad (10.21)$$

This explains why:

- We can solve the dual problem instead of the primal
- The optimal values of both problems coincide
- The KKT conditions are both necessary and sufficient

10.2.4 Saddle Point and Max-Min Equality

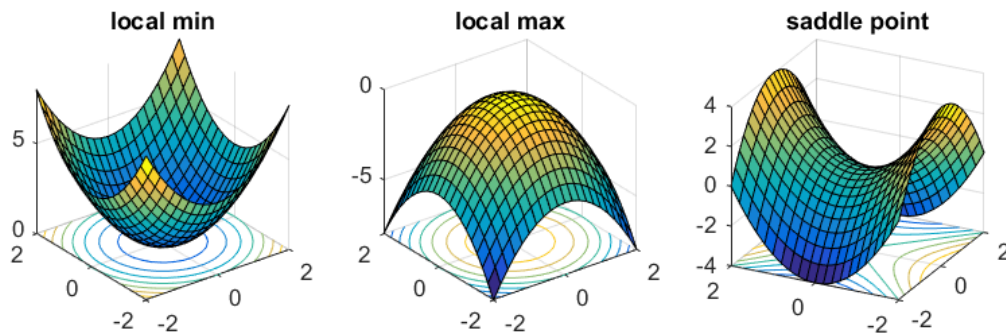


Figure 10.3: Left: local minimum, e.g., $x^2 + y^2$. Middle: local maximum, e.g., $-(x^2 + y^2)$. Right: saddle point, e.g., $x^2 - y^2$.

The equality between $\max_{\alpha \geq 0} \min_{w,b} \mathcal{L}(w, b, \alpha)$ and $\min_{w,b} \max_{\alpha \geq 0} \mathcal{L}(w, b, \alpha)$ is not a coincidence but follows from the special structure of the Lagrangian. Let's explore this deeply.

Definition 44 (Saddle Point). A point (w^*, b^*, α^*) is called a saddle point of \mathcal{L} if for all feasible (w, b, α) :

$$\mathcal{L}(w^*, b^*, \alpha) \leq \mathcal{L}(w^*, b^*, \alpha^*) \leq \mathcal{L}(w, b, \alpha^*) \quad (10.22)$$

Definition 45 (Concave-Convex Function). A function $\mathcal{L}(x, y)$ is called concave-convex if:

- For fixed y , $x \mapsto \mathcal{L}(x, y)$ is convex
- For fixed x , $y \mapsto \mathcal{L}(x, y)$ is concave

Lemma 46 (Structure of SVM Lagrangian). *The SVM Lagrangian $\mathcal{L}(w, b, \alpha)$ is:*

1. *Strictly convex in (w, b) for any fixed α*
2. *Linear (hence concave) in α for any fixed (w, b)*
3. *Defined on a convex domain*

Proof. 1. For fixed α , the Hessian with respect to (w, b) is:

$$H = \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix} \quad (10.23)$$

which is positive semidefinite, making \mathcal{L} convex in (w, b) .

2. \mathcal{L} is linear in α as it appears only in terms of the form $\alpha_i(\cdot)$.

3. The domains $\{(w, b)\} = \mathbb{R}^{d+1}$ and $\{\alpha : \alpha_i \geq 0\}$ are convex. \square

Theorem 47 (Saddle Point Existence). *Under the conditions above, if (w^*, b^*, α^*) is a KKT point of the SVM optimization, then it is a saddle point of \mathcal{L} .*

Proof. At the KKT point:

1. $\nabla_{w,b} \mathcal{L}(w^*, b^*, \alpha^*) = 0$ implies (w^*, b^*) minimizes $\mathcal{L}(\cdot, \cdot, \alpha^*)$
2. Complementary slackness and primal feasibility imply α^* maximizes $\mathcal{L}(w^*, b^*, \cdot)$

Therefore:

$$\mathcal{L}(w^*, b^*, \alpha) \leq \mathcal{L}(w^*, b^*, \alpha^*) \leq \mathcal{L}(w, b, \alpha^*) \quad (10.24)$$

\square

Corollary 48 (Max-Min Equality). *For the SVM problem:*

$$\max_{\alpha \geq 0} \min_{w, b} \mathcal{L}(w, b, \alpha) = \min_{w, b} \max_{\alpha \geq 0} \mathcal{L}(w, b, \alpha) \quad (10.25)$$

Proof. By the concave-convex property and convex domains:

$$\max_{\alpha \geq 0} \min_{w, b} \mathcal{L}(w, b, \alpha) \leq \mathcal{L}(w^*, b^*, \alpha^*) \quad (10.26)$$

$$\leq \min_{w, b} \max_{\alpha \geq 0} \mathcal{L}(w, b, \alpha) \quad (10.27)$$

The reverse inequality always holds, so equality follows. \square

Remark 49 (Geometric Interpretation). The saddle point property provides a geometric interpretation:

- The optimal solution sits at a "mountain pass"
- It's a minimum along the primal directions (w, b)
- It's a maximum along the dual directions α
- This geometry ensures the solution is unique and stable

Remark 50 (Algorithmic Implications). This structure suggests various algorithms:

- Primal-dual methods can alternate between max and min steps
- Gradient descent on (w, b) can be combined with ascent on α
- The saddle point structure guarantees convergence under appropriate conditions

10.2.5 Game Theoretic Interpretation of Max-Min Equality

The equality between max min and min max can be elegantly understood through the lens of two-player zero-sum games, as formalized by von Neumann.

Definition 51 (Two-Player Zero-Sum Game). A two-player zero-sum game consists of:

- Player 1 (minimizer) chooses strategy $x \in X$
- Player 2 (maximizer) chooses strategy $y \in Y$
- Payoff function $f(x, y)$ where:
 - Player 1 pays Player 2 amount $f(x, y)$
 - Player 1 wants to minimize payment
 - Player 2 wants to maximize payment

Remark 52 (Order of Play). Two scenarios are possible:

1. Player 1 moves first:
 - Player 2 can observe and react optimally
 - Results in payoff $\max_y f(x, y)$
 - Player 1 anticipates this: $\min_x \max_y f(x, y)$
2. Player 2 moves first:
 - Player 1 can observe and react optimally
 - Results in payoff $\min_x f(x, y)$
 - Player 2 anticipates this: $\max_y \min_x f(x, y)$

Theorem 53 (von Neumann’s Minimax Theorem). *If X and Y are compact convex sets, and $f(x, y)$ is continuous and concave-convex, then:*

$$\min_{x \in X} \max_{y \in Y} f(x, y) = \max_{y \in Y} \min_{x \in X} f(x, y) \quad (10.28)$$

Remark 54 (Implications). This remarkable equality implies:

1. The order of play doesn’t matter
2. Neither player can benefit from moving second
3. There exists an equilibrium strategy pair (x^*, y^*)
4. The value $v^* = f(x^*, y^*)$ is the game value

Example 55 (Application to SVM). For SVMs, we can interpret:

- Player 1: Primal variables (w, b)
- Player 2: Dual variables α
- Payoff: Lagrangian $\mathcal{L}(w, b, \alpha)$
- Game value: Optimal objective value

Remark 56 (Mixed Strategies). Even when pure strategy equilibria don’t exist:

- Players can randomize their choices
- Mixed strategy equilibria always exist
- The expected payoff satisfies max-min equality

Theorem 57 (Nash Equilibrium in Zero-Sum Games). *In a two-player zero-sum game satisfying von Neumann’s conditions:*

1. A Nash equilibrium always exists
2. All Nash equilibria have the same value
3. The value equals $\min \max = \max \min$

Remark 58 (Computational Aspects). The game theoretic view suggests:

- Iterative algorithms simulating alternating play
- Learning by repeated game playing
- Convergence to equilibrium strategies
- Connection to online learning and regret minimization

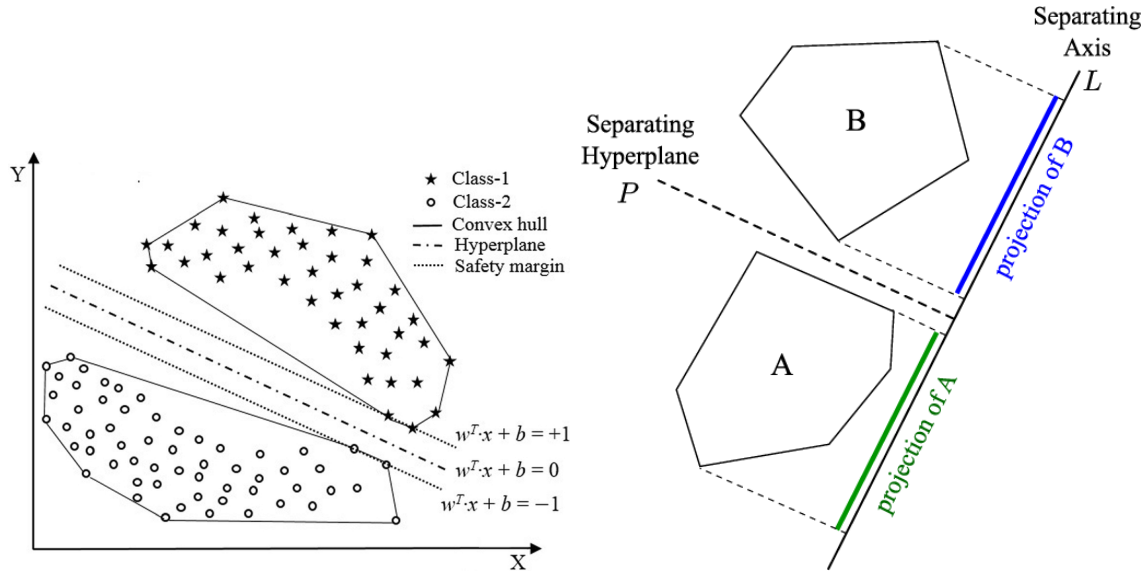


Figure 10.4: Convex hulls

10.3 Dual Problem: Min Distance

10.3.1 Initial Dual Derivation

Starting from the primal problem:

$$\min_{w,b} \frac{1}{2} \|w\|^2 \quad (10.29)$$

$$\text{s.t. } y_i(\langle w, x_i \rangle + b) \geq 1, \quad i = 1, \dots, n \quad (10.30)$$

The Lagrangian is:

$$\mathcal{L}(w, b, \alpha) = \frac{1}{2} \|w\|^2 - \sum_{i=1}^n \alpha_i [y_i(\langle w, x_i \rangle + b) - 1] \quad (10.31)$$

Taking derivatives:

$$\frac{\partial \mathcal{L}}{\partial w} = w - \sum_{i=1}^n \alpha_i y_i x_i = 0 \quad \implies \quad w = \sum_{i=1}^n \alpha_i y_i x_i \quad (10.32)$$

$$\frac{\partial \mathcal{L}}{\partial b} = - \sum_{i=1}^n \alpha_i y_i = 0 \quad (10.33)$$

Substituting back gives our initial dual form:

$$\max_{\alpha} \sum_{i=1}^n \alpha_i - \frac{1}{2} \left\| \sum_{i=1}^n \alpha_i y_i x_i \right\|^2 \quad (10.34)$$

$$\text{s.t. } \alpha_i \geq 0, \quad i = 1, \dots, n \quad (10.35)$$

$$\sum_{i=1}^n \alpha_i y_i = 0 \quad (10.36)$$

Later, we can expand

$$\left\| \sum_{i=1}^n \alpha_i y_i x_i \right\|^2 = \langle \alpha_i y_i x_i \alpha_j y_j x_j \rangle \quad (10.37)$$

$$= \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j \langle x_i, x_j \rangle \quad (10.38)$$

when we want to employ the kernel trick.

10.3.2 Geometric Interpretation via frontal points

Let's interpret $w = \sum_{i=1}^n \alpha_i y_i x_i$ geometrically by separating positive and negative classes:

$$w = \sum_{i:y_i=+1} \alpha_i x_i - \sum_{i:y_i=-1} \alpha_i x_i = x_+ - x_- \quad (10.39)$$

where:

$$x_+ = \sum_{i:y_i=+1} \alpha_i x_i \quad (\text{frontal positive point}) \quad (10.40)$$

$$x_- = \sum_{i:y_i=-1} \alpha_i x_i \quad (\text{frontal negative point}) \quad (10.41)$$

This gives:

$$\|w\|^2 = \|x_+ - x_-\|^2 \quad (10.42)$$

10.3.3 The Distance Interpretation

The dual objective becomes:

$$\max_{\alpha} \sum_{i=1}^n \alpha_i - \frac{1}{2} \|x_+ - x_-\|^2 \quad (10.43)$$

The constraint $\sum_{i=1}^n \alpha_i y_i = 0$ implies that the total weights for positive and negative points are equal:

$$\sum_{i:y_i=+1} \alpha_i = \sum_{i:y_i=-1} \alpha_i = \lambda \quad (10.44)$$

for some $\lambda > 0$.

Definition 59 (Normalized Weights). Define:

$$\gamma_i^+ = \frac{\alpha_i}{\lambda} \quad \text{for } y_i = +1 \quad (10.45)$$

$$\gamma_i^- = \frac{\alpha_i}{\lambda} \quad \text{for } y_i = -1 \quad (10.46)$$

This gives:

$$\sum_{i:y_i=+1} \gamma_i^+ = 1 \quad (10.47)$$

$$\sum_{i:y_i=-1} \gamma_i^- = 1 \quad (10.48)$$

$$\gamma_i^+, \gamma_i^- \geq 0 \quad (10.49)$$

This reveals that $\bar{x}_+ = x_+/\lambda$ and $\bar{x}_- = x_-/\lambda$ are convex combinations of points in their respective classes:

$$\bar{x}_+ \in \text{conv}\{x_i : y_i = +1\} \quad (10.50)$$

$$\bar{x}_- \in \text{conv}\{x_i : y_i = -1\} \quad (10.51)$$

10.3.4 The Minimum Distance Problem

Therefore, the SVM dual is equivalent to finding the minimum distance between the convex hulls of the positive and negative classes:

$$\min_{\bar{x}_+, \bar{x}_-} \|\bar{x}_+ - \bar{x}_-\| \quad (10.52)$$

$$\text{s.t. } \bar{x}_+ \in \text{conv}\{x_i : y_i = +1\} \quad (10.53)$$

$$\bar{x}_- \in \text{conv}\{x_i : y_i = -1\} \quad (10.54)$$

Theorem 60 (Geometric Interpretation). *The following properties hold:*

1. w points from \bar{x}_- to \bar{x}_+
2. The margin is twice the distance between convex hulls
3. Support vectors are the points with non-zero coefficients in these convex combinations
4. \bar{x}_+ and \bar{x}_- are on the frontal faces of the positive and negative convex hulls respectively

10.3.5 Projections and Separation

When we project any point x onto $w = x_+ - x_-$, we get:

$$\frac{\langle x, w \rangle}{\|w\|} = \frac{\langle x, x_+ - x_- \rangle}{\|x_+ - x_-\|} \quad (10.55)$$

Corollary 61 (Projection Properties). *The projection reveals:*

1. Support vectors from each class project to the same positions on w
2. Non-support vectors project beyond these positions
3. The margin width is $\frac{2}{\|w\|} = \frac{2}{\|x_+ - x_-\|}$

10.3.6 Karush-Kuhn-Tucker (KKT) Conditions

The complete set of KKT conditions:

1. **Primal feasibility:**

$$y_i(\langle w, x_i \rangle + b) \geq 1 \quad (10.56)$$

2. **Dual feasibility:**

$$\alpha_i \geq 0 \quad (10.57)$$

3. **Complementary slackness:**

$$\alpha_i[y_i(\langle w, x_i \rangle + b) - 1] = 0 \quad (10.58)$$

The complementary slackness condition reveals that:

- If $\alpha_i > 0$: point lies exactly on the margin ($y_i(\langle w, x_i \rangle + b) = 1$)
- If $\alpha_i = 0$: point is either correctly classified beyond the margin or misclassified

10.4 Dual Coordinate Ascent

10.4.1 Dual Problem with $b = 0$

We first study a simple case where we assume $b = 0$. This is a reasonable assumption for the kernel version.

With $b = 0$, our dual problem simplifies to:

$$\max_{\alpha} \quad \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j \langle x_i, x_j \rangle \quad (10.59)$$

$$\text{s.t.} \quad \alpha_i \geq 0, \quad i = 1, \dots, n \quad (10.60)$$

Note that we no longer have the constraint $\sum_{i=1}^n \alpha_i y_i = 0$.

10.4.2 Coordinate-wise Optimization

For updating α_i , fix all other α_j ($j \neq i$). Let's write the objective in terms of α_i :

$$f(\alpha_i) = \alpha_i + \sum_{j \neq i} \alpha_j - \frac{1}{2} \alpha_i^2 \langle x_i, x_i \rangle - \alpha_i \sum_{j \neq i} \alpha_j y_i y_j \langle x_i, x_j \rangle - \text{const} \quad (10.61)$$

$$= \alpha_i - \frac{1}{2} Q_{ii} \alpha_i^2 - \alpha_i \sum_{j \neq i} y_i y_j Q_{ij} \alpha_j + \text{const} \quad (10.62)$$

where $Q_{ij} = \langle x_i, x_j \rangle$ is the kernel matrix.

10.4.3 Optimal Update

Taking the derivative with respect to α_i and setting to zero:

$$\frac{\partial f}{\partial \alpha_i} = 1 - Q_{ii}\alpha_i - \sum_{j \neq i} y_i y_j Q_{ij} \alpha_j = 0 \quad (10.63)$$

Define the gradient component:

$$g_i = 1 - y_i \sum_{j=1}^n y_j Q_{ij} \alpha_j = 1 - y_i \langle w, x_i \rangle \quad (10.64)$$

Then the optimal value for α_i is:

$$\alpha_i^{\text{new}} = \max \left\{ 0, \frac{y_i g_i}{Q_{ii}} \right\} \quad (10.65)$$

10.4.4 Algorithm

Algorithm 12 Dual Coordinate Ascent for SVM

```

1: Initialize:  $\alpha = 0, w = 0$ 
2: while not converged do
3:   for  $i = 1$  to  $n$  do
4:      $g_i \leftarrow 1 - y_i \langle w, x_i \rangle$ 
5:      $\alpha_i^{\text{old}} \leftarrow \alpha_i$ 
6:      $\alpha_i \leftarrow \max\{0, y_i g_i / Q_{ii}\}$ 
7:      $w \leftarrow w + (\alpha_i - \alpha_i^{\text{old}}) y_i x_i$ 
8:   end for
9: end while

```

10.4.5 Implementation Details

1. **Maintaining w :**

- Keep $w = \sum_{i=1}^n \alpha_i y_i x_i$ in memory
- Update incrementally after each α_i change
- Reduces computation of g_i from $O(n)$ to $O(p)$

2. **Convergence Check:**

- Monitor maximum KKT violation:

$$\max_i |y_i g_i| \cdot \min\{\alpha_i, |1 - y_i g_i|\} \leq \epsilon \quad (10.66)$$

3. **Shrinking Strategy:**

- Skip updates for $\alpha_i = 0$ when $y_i g_i < 1$
- Skip updates for $\alpha_i > 0$ when $y_i g_i = 1$

10.5 The Kernel Trick

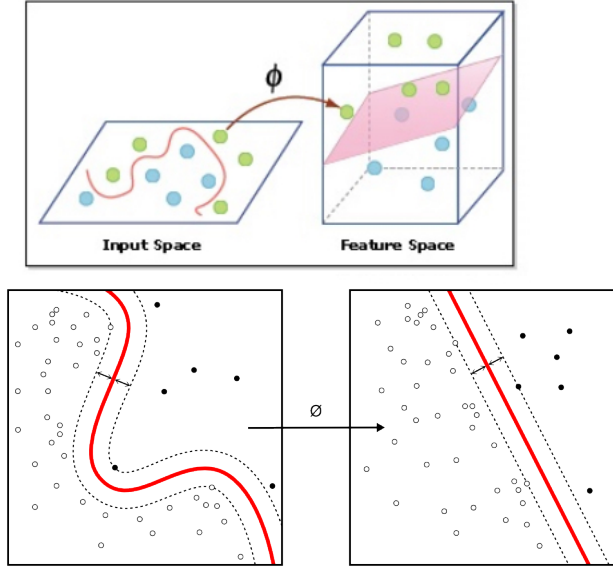


Figure 10.5: The kernel trick maps data to a feature space where linear separation is possible

10.5.1 Motivation

Observe that in our dual formulation and coordinate descent algorithm, the data x_i appears only through inner products $\langle x_i, x_j \rangle$. This suggests we can implicitly work in a higher-dimensional feature space without explicitly computing the transformations.

Specifically, the original dual optimization problem is:

$$\max_{\alpha} \quad \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j \langle x_i, x_j \rangle \quad (10.67)$$

$$\text{s.t.} \quad \alpha_i \geq 0, \quad i = 1, \dots, n \quad (10.68)$$

This optimization problem only depends on $\langle x_i, x_j \rangle$. After optimization, we obtain $(\alpha_i, i = 1, \dots, n)$, and

$$w = \sum_{i=1}^n \alpha_i y_i x_i \quad (10.69)$$

The decision function is

$$f(x) = \langle w, x \rangle = \sum_{i=1}^n \alpha_i y_i \langle x_i, x \rangle \quad (10.70)$$

so that x is classified as positive if $f(x) \geq 0$. The decision function $f(x)$ only depends on $\langle x_i, x \rangle$.

If we cannot separate the positive and negative examples in the original x space, we can map x to a higher dimensional $\Phi(x)$ so that they can be separated in the Φ space.

10.5.2 Kernel Function

Define a feature map $\Phi : \mathcal{X} \rightarrow \mathcal{F}$ into a high (possibly infinite) dimensional feature space. A kernel function computes inner products in this space:

$$K(x, z) = \langle \Phi(x), \Phi(z) \rangle_{\mathcal{F}} \quad (10.71)$$

10.5.3 Kernelized Dual Problem

With the feature map, we can replace x by $\Phi(x)$. Then the dual optimization problem becomes

$$\max_{\alpha} \quad \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j \langle \Phi(x_i), \Phi(x_j) \rangle \quad (10.72)$$

$$\text{s.t.} \quad \alpha_i \geq 0, \quad i = 1, \dots, n \quad (10.73)$$

The above is equivalent to

$$\max_{\alpha} \quad \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j K(x_i, x_j) \quad (10.74)$$

$$\text{s.t.} \quad \alpha_i \geq 0, \quad i = 1, \dots, n \quad (10.75)$$

The decision function is:

$$f(x) = \sum_{i=1}^n \alpha_i y_i \langle \Phi(x_i), \Phi(x) \rangle = \sum_{i=1}^n \alpha_i y_i K(x_i, x) \quad (10.76)$$

10.5.4 Common Kernel Functions

1. **Polynomial Kernel** (degree d):

$$K(x, z) = (1 + \langle x, z \rangle)^d \quad (10.77)$$

Feature space includes all monomials up to degree d

2. **Gaussian RBF Kernel:**

$$K(x, z) = \exp(-\gamma \|x - z\|^2) \quad (10.78)$$

Infinite-dimensional feature space

3. **Sigmoid Kernel:**

$$K(x, z) = \tanh(\kappa \langle x, z \rangle + c) \quad (10.79)$$

10.5.5 Mercer's Theorem

Theorem 62 (Mercer's Condition). *A symmetric function $K(x, z)$ is a valid kernel if and only if its Gram matrix is positive semidefinite for any finite set $\{x_i\}_{i=1}^n$:*

$$\sum_{i=1}^n \sum_{j=1}^n c_i c_j K(x_i, x_j) \geq 0 \quad \forall c_i \in \mathbb{R} \quad (10.80)$$

This guarantees the existence of a feature space \mathcal{F} and map Φ where $K(x, z) = \langle \Phi(x), \Phi(z) \rangle_{\mathcal{F}}$.

10.5.6 Kernelized Coordinate Descent

The coordinate descent updates become:

1. Gradient computation:

$$g_i = 1 - y_i \sum_{j=1}^n \alpha_j y_j K(x_i, x_j) \quad (10.81)$$

2. Optimal update:

$$\alpha_i^{\text{new}} = \max \left\{ 0, \frac{y_i g_i}{K(x_i, x_i)} \right\} \quad (10.82)$$

10.5.7 Implementation Considerations

1. **Kernel Cache:**

- Store frequently used rows of kernel matrix
- Trade-off between memory and computation

2. **Kernel Matrix Properties:**

- Symmetric: $K(x_i, x_j) = K(x_j, x_i)$
- Diagonal elements constant for normalized kernels

3. **Numerical Stability:**

- Add small constant to diagonal ($K(x, x) + \epsilon$)
- Maintain numerical precision in kernel computations

10.5.8 Reproducing Kernel Hilbert Space

Construction from Feature Maps

Let $\Phi : \mathcal{X} \rightarrow \mathbb{R}^D$ (possibly infinite-dimensional) be a feature map. Consider functions of the form:

$$f(x) = \Phi(x)^\top \beta \quad (10.83)$$

$$g(x) = \Phi(x)^\top \gamma \quad (10.84)$$

where β, γ are vectors in \mathbb{R}^D .

Inner Product Structure

Define an inner product between functions through their parameters:

$$\langle f, g \rangle_{\mathcal{H}} = \langle \beta, \gamma \rangle = \beta^\top \gamma \quad (10.85)$$

This induces a norm:

$$\|f\|_{\mathcal{H}}^2 = \langle f, f \rangle_{\mathcal{H}} = \|\beta\|^2 \quad (10.86)$$

Kernel Function

The kernel function arises naturally as:

$$K(x, z) = \Phi(x)^\top \Phi(z) \quad (10.87)$$

For any fixed x , define the function:

$$K_x(\cdot) = K(\cdot, x) = \Phi(\cdot)^\top \Phi(x) \quad (10.88)$$

The Reproducing Property

Consider the inner product between f and K_x :

$$\langle f, K_x \rangle_{\mathcal{H}} = \langle \Phi(\cdot)^\top \beta, \Phi(\cdot)^\top \Phi(x) \rangle_{\mathcal{H}} \quad (10.89)$$

$$= \beta^\top \Phi(x) \quad (10.90)$$

$$= f(x) \quad (10.91)$$

This is the reproducing property:

$$\boxed{\langle f, K(\cdot, x) \rangle_{\mathcal{H}} = f(x)} \quad (10.92)$$

Implications

1. Evaluation Functional:

- For fixed x , the map $f \mapsto f(x)$ is continuous
- $|f(x)| \leq \|f\|_{\mathcal{H}} \sqrt{K(x, x)}$ by Cauchy-Schwarz

2. Feature Expansion: Any function $f \in \mathcal{H}$ can be written as:

$$f = \sum_{i=1}^n \alpha_i K(\cdot, x_i) \quad (10.93)$$

for some n , coefficients $\{\alpha_i\}$, and points $\{x_i\}$

3. Norm Computation: For $f = \sum_{i=1}^n \alpha_i K(\cdot, x_i)$:

$$\|f\|_{\mathcal{H}}^2 = \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j K(x_i, x_j) \quad (10.94)$$

Properties of the RKHS

1. Uniqueness:

Theorem 63. *For a positive definite kernel K , there exists a unique RKHS \mathcal{H} with reproducing kernel K .*

2. Completeness: \mathcal{H} is complete under the norm $\|\cdot\|_{\mathcal{H}}$

3. Dense Subset: The span of $\{K(\cdot, x) : x \in \mathcal{X}\}$ is dense in \mathcal{H}

10.5.9 Example: Gaussian RBF Kernel

For $K(x, z) = \exp(-\gamma\|x - z\|^2)$:

1. **Feature Map:**

$$\Phi(x) = \exp(-\gamma\|x\|^2) \left(1, \sqrt{\frac{2\gamma}{1!}}x_1, \sqrt{\frac{2\gamma}{1!}}x_2, \sqrt{\frac{2\gamma^2}{2!}}x_1^2, \dots \right) \quad (10.95)$$

2. **Function Space:**

$$\mathcal{H} = \left\{ f : \sum_{|\alpha| \geq 0} \frac{|\alpha|!}{\gamma^{|\alpha|}} \|f_\alpha\|^2 < \infty \right\} \quad (10.96)$$

where f_α are coefficients in the Taylor expansion

Connection to SVM

The SVM optimization in RKHS becomes:

$$\min_{f \in \mathcal{H}} \frac{1}{2} \|f\|_{\mathcal{H}}^2 + C \sum_{i=1}^n \max(0, 1 - y_i f(x_i)) \quad (10.97)$$

$$f(x) = \sum_{i=1}^n \alpha_i y_i K(x_i, x) \quad (10.98)$$

The RKHS norm naturally penalizes complexity:

$$\|f\|_{\mathcal{H}}^2 = \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j K(x_i, x_j) \quad (10.99)$$

10.6 Soft Margin SVM

10.6.1 Motivation

When classes are not linearly separable, we introduce slack variables $\xi_i \geq 0$ to allow for violations of the margin constraints:

- $\xi_i = 0$: point is correctly classified and outside the margin
- $0 < \xi_i \leq 1$: point is correctly classified but inside the margin
- $\xi_i > 1$: point is misclassified

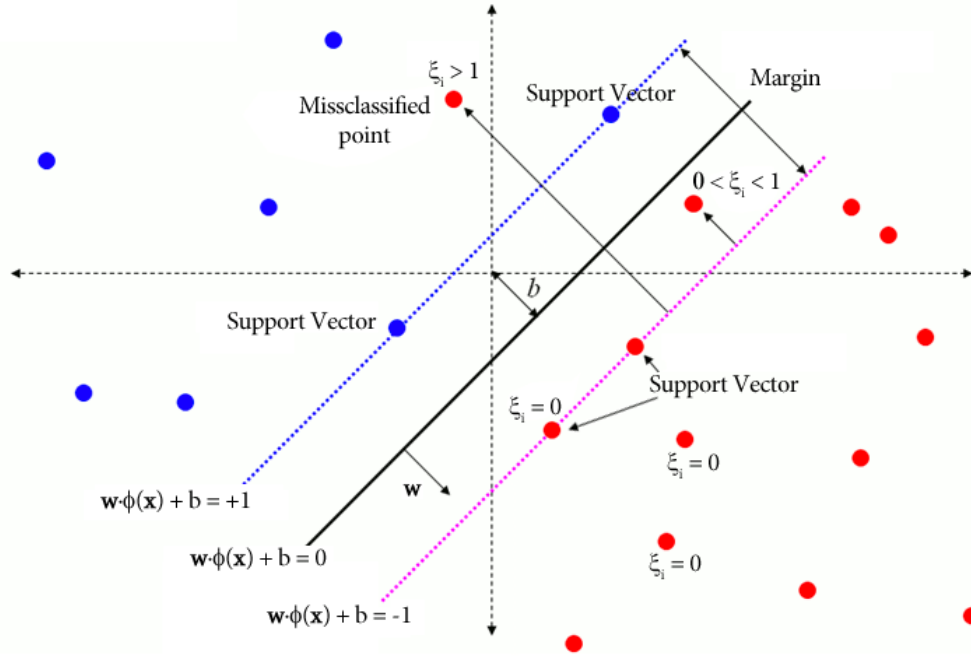


Figure 10.6: Slack variables

10.6.2 Primal Problem

The optimization problem becomes:

$$\min_{w, b, \xi} \quad \frac{1}{2} \|w\|^2 + C \sum_{i=1}^n \xi_i \quad (10.100)$$

$$\text{s.t.} \quad y_i(\langle w, x_i \rangle + b) \geq 1 - \xi_i, \quad i = 1, \dots, n \quad (10.101)$$

$$\xi_i \geq 0, \quad i = 1, \dots, n \quad (10.102)$$

where:

- $C > 0$ is the regularization parameter
- Larger C penalizes violations more heavily
- Smaller C allows for a wider margin

10.6.3 Lagrangian

The Lagrangian with dual variables $\alpha_i \geq 0$ and $\mu_i \geq 0$:

$$\mathcal{L} = \frac{1}{2} \|w\|^2 + C \sum_{i=1}^n \xi_i - \sum_{i=1}^n \alpha_i [y_i(\langle w, x_i \rangle + b) - 1 + \xi_i] - \sum_{i=1}^n \mu_i \xi_i \quad (10.103)$$

10.6.4 KKT Conditions

$$\frac{\partial \mathcal{L}}{\partial w} = w - \sum_{i=1}^n \alpha_i y_i x_i = 0 \quad (10.104)$$

$$\frac{\partial \mathcal{L}}{\partial b} = - \sum_{i=1}^n \alpha_i y_i = 0 \quad (10.105)$$

$$\frac{\partial \mathcal{L}}{\partial \xi_i} = C - \alpha_i - \mu_i = 0 \quad (10.106)$$

From the last equation and $\mu_i \geq 0$:

$$0 \leq \alpha_i \leq C \quad (10.107)$$

10.6.5 Dual Problem

The dual optimization becomes:

$$\max_{\alpha} \quad \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j \langle x_i, x_j \rangle \quad (10.108)$$

$$\text{s.t.} \quad 0 \leq \alpha_i \leq C, \quad i = 1, \dots, n \quad (10.109)$$

$$\sum_{i=1}^n \alpha_i y_i = 0 \quad (10.110)$$

10.6.6 Support Vector Cases

The KKT conditions reveal three types of points:

1. **Non-support vectors** ($\alpha_i = 0$):

- Correctly classified outside margin
- $\xi_i = 0$

2. **Margin support vectors** ($0 < \alpha_i < C$):

- Exactly on margin
- $\xi_i = 0$

3. **Bounded support vectors** ($\alpha_i = C$):

- Inside margin or misclassified
- $\xi_i > 0$

10.6.7 Bias Term Computation

For any margin support vector ($0 < \alpha_i < C$):

$$b = y_i - \sum_{j=1}^n \alpha_j y_j \langle x_j, x_i \rangle \quad (10.111)$$

In practice, average over all margin support vectors:

$$b = \frac{1}{|S|} \sum_{i \in S} \left(y_i - \sum_{j=1}^n \alpha_j y_j \langle x_j, x_i \rangle \right) \quad (10.112)$$

where $S = \{i : 0 < \alpha_i < C\}$.

10.6.8 Model Selection

The parameter C controls the trade-off between:

- Margin width (regularization)
- Training errors (empirical risk)

Typically chosen through cross-validation over a grid:

$$C \in \{2^{-5}, 2^{-3}, \dots, 2^{13}, 2^{15}\} \quad (10.113)$$

10.7 Sequential Minimal Optimization (SMO)

10.7.1 Problem Structure

Recall the dual optimization problem:

$$\max_{\alpha} \quad \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i,j=1}^n \alpha_i \alpha_j y_i y_j K(x_i, x_j) \quad (10.114)$$

$$\text{s.t.} \quad 0 \leq \alpha_i \leq C, \quad i = 1, \dots, n \quad (10.115)$$

$$\sum_{i=1}^n \alpha_i y_i = 0 \quad (10.116)$$

Let $Q_{ij} = y_i y_j K(x_i, x_j)$. Define the objective function:

$$f(\alpha) = \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i,j=1}^n \alpha_i \alpha_j Q_{ij} \quad (10.117)$$

10.7.2 Two-Variable Subproblem

Theorem 64 (SMO Subproblem). *Given a feasible α , consider updating components i and j while keeping others fixed. Then:*

1. *The equality constraint reduces to:*

$$\alpha_i y_i + \alpha_j y_j = \gamma \quad \text{where } \gamma = - \sum_{k \neq i, j} \alpha_k y_k \quad (10.118)$$

2. *The subproblem in α_j becomes:*

$$\min_{\alpha_j} \quad \frac{1}{2}(\alpha_i^2 Q_{ii} + 2\alpha_i \alpha_j Q_{ij} + \alpha_j^2 Q_{jj}) - (\alpha_i + \alpha_j) \quad (10.119)$$

$$\text{s.t.} \quad 0 \leq \alpha_j \leq C \quad (10.120)$$

$$\alpha_i = \frac{\gamma - \alpha_j y_j}{y_i} \quad (10.121)$$

Proof. The first statement follows directly from the equality constraint. For the second, substitute the expression for α_i into the objective and collect terms. \square

10.7.3 Analytical Solution

Theorem 65 (Optimal Update). *Let $g_i = \sum_{k=1}^n \alpha_k Q_{ik} - 1$ be the negative gradient component. The optimal update for α_j is:*

$$\alpha_j^{new} = \alpha_j^{old} + \frac{y_j(g_i - g_j)}{Q_{ii} + Q_{jj} - 2Q_{ij}} \quad (10.122)$$

subject to box constraints.

Proof. 1. Substitute α_i expression into objective to get quadratic in α_j 2. Take derivative and set to zero 3. Express gradient components using g_i and g_j 4. Solve for update \square

10.7.4 Constraint Handling

Lemma 66 (Box Constraints). *The solution must satisfy $L \leq \alpha_j^{new} \leq H$ where:*

If $y_i = y_j$:

$$L = \max(0, \alpha_i + \alpha_j - C) \quad (10.123)$$

$$H = \min(C, \alpha_i + \alpha_j) \quad (10.124)$$

If $y_i \neq y_j$:

$$L = \max(0, \alpha_j - \alpha_i) \quad (10.125)$$

$$H = \min(C, C + \alpha_j - \alpha_i) \quad (10.126)$$

Proof. Apply the box constraints $0 \leq \alpha_i, \alpha_j \leq C$ to the equality constraint equation. \square

10.7.5 Algorithm and Convergence

Algorithm 13 Sequential Minimal Optimization

```

1: Initialize  $\alpha = 0$ 
2: while not converged do
3:   Select index  $i$  violating KKT conditions
4:   Select second index  $j \neq i$ 
5:    $\gamma \leftarrow -\sum_{k \neq i,j} \alpha_k y_k$ 
6:   Update  $\alpha_j$  using optimal update formula
7:   Clip  $\alpha_j$  to  $[L, H]$ 
8:   Update  $\alpha_i$  to maintain  $\sum_k \alpha_k y_k = 0$ 
9: end while
  
```

Theorem 67 (Convergence). *The SMO algorithm converges to the global optimum of the dual problem.*

Proof Sketch. 1. Each update strictly improves the objective unless at optimum 2. The objective is bounded above (due to constraints) 3. The feasible set is compact 4. There are finitely many possible working sets \square

10.7.6 Connection to Coordinate Methods

SMO can be viewed as a special case of block coordinate ascent where:

1. Block size is always 2
2. One variable (i) chosen by KKT violation
3. Second variable (j) chosen to maximize progress
4. Updates maintain feasibility exactly

This contrasts with standard coordinate descent which:

1. Updates one variable at a time
2. May temporarily violate equality constraint
3. Requires projection back to feasible set

10.7.7 Comparison of Incremental Learning Strategies

Both SMO and coordinate descent for SVMs can be viewed through the lens of incremental model improvement, similar to how trees and boosting build models. Let's examine these connections:

Model Representation

Each method maintains and incrementally improves a model:

- **SVM (SMO/Coordinate Descent):**

$$f(x) = \sum_{i=1}^n \alpha_i y_i K(x_i, x) \quad (10.127)$$

- **Boosting:**

$$f(x) = \sum_{m=1}^M \beta_m h_m(x) \quad (10.128)$$

- **Trees:**

$$f(x) = \sum_{l=1}^L c_l \mathbb{I}[x \in R_l] \quad (10.129)$$

Update Strategies

The key difference lies in how these models are updated:

1. **SMO:** Updates pairs (α_i, α_j) while maintaining $\sum \alpha_i y_i = 0$
2. **Dual Coordinate Ascent:** Updates single α_i at a time
3. **Boosting:** Adds new weak learners $h_m(x)$ sequentially
4. **Trees:** Greedily splits regions R_l to improve fit

Geometric Interpretation

Each method can be viewed as searching in different spaces:

- **SVM:** Moves along feasible line segments in the dual space to improve margin
- **Boosting:** Moves in the direction of steepest descent in function space
- **Trees:** Partitions feature space to minimize loss

Trade-offs

The methods balance different concerns:

- **SVM Updates:** Exact optimization of sub-problems, constrained movement
- **Boosting:** Approximate optimization, unconstrained growth
- **Trees:** Greedy local decisions, hierarchical structure

10.8 From Slack Variables to Hinge Loss

The soft-margin SVM objective:

$$\min_{w,b,\xi} \frac{1}{2} \|w\|^2 + C \sum_{i=1}^n \xi_i \quad \text{s.t.} \quad y_i(\langle w, x_i \rangle + b) \geq 1 - \xi_i, \quad \xi_i \geq 0 \quad (10.130)$$

can be rewritten by eliminating ξ_i :

$$\min_{w,b} \frac{1}{2} \|w\|^2 + C \sum_{i=1}^n \max(0, 1 - y_i(\langle w, x_i \rangle + b)) \quad (10.131)$$

This reveals the hinge loss:

$$L_{\text{hinge}}(z) = \max(0, 1 - z) \quad (10.132)$$

10.8.1 Three Major Loss Functions

For margin $z = yf(x)$:

1. **Hinge Loss** (SVM):

$$L_{\text{hinge}}(z) = \max(0, 1 - z) \quad (10.133)$$

2. **Logistic Loss** (Logistic Regression):

$$L_{\text{log}}(z) = \log(1 + e^{-z}) \quad (10.134)$$

3. **Exponential Loss** (AdaBoost):

$$L_{\text{exp}}(z) = e^{-z} \quad (10.135)$$

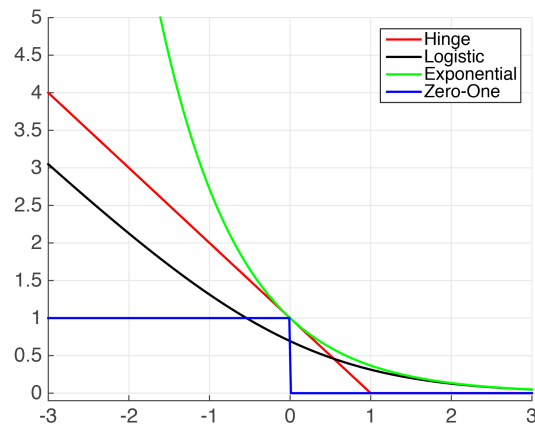


Figure 10.7: Loss functions

10.8.2 Properties

1. Hinge Loss:

- Piecewise linear
- Zero loss for correctly classified points beyond margin
- Linear penalty for margin violations
- Not differentiable at $z = 1$
- Sparse solutions (many zero α_i)

2. Logistic Loss:

- Smooth and differentiable everywhere
- Never exactly zero
- Can be interpreted as log-likelihood
- Probabilistic interpretation: $P(Y = 1|X = x) = \frac{1}{1+e^{-f(x)}}$
- Non-sparse solutions

3. Exponential Loss:

- Smooth and differentiable
- Grows exponentially for negative margins
- More sensitive to outliers
- Connection to AdaBoost's multiplicative updates

10.8.3 Derivatives

1. Hinge Loss:

$$\frac{\partial L_{\text{hinge}}}{\partial z} = \begin{cases} -1 & \text{if } z < 1 \\ 0 & \text{if } z > 1 \end{cases} \quad (10.136)$$

2. Logistic Loss:

$$\frac{\partial L_{\text{log}}}{\partial z} = -\frac{1}{1 + e^z} \quad (10.137)$$

3. Exponential Loss:

$$\frac{\partial L_{\text{exp}}}{\partial z} = -e^{-z} \quad (10.138)$$

10.8.4 Statistical Interpretation

1. Population Minimizers:

- Logistic Loss: $f^*(x) = \log \frac{P(Y=1|X=x)}{P(Y=-1|X=x)}$
- Exponential Loss: $f^*(x) = \frac{1}{2} \log \frac{P(Y=1|X=x)}{P(Y=-1|X=x)}$
- Hinge Loss: $f^*(x) = \text{sign}(P(Y = 1|X = x) - \frac{1}{2})$

2. **Fisher Consistency:** All three losses are Fisher consistent for binary classification:

$$\text{sign}(f^*(x)) = \text{sign}(2P(Y = 1|X = x) - 1) \quad (10.139)$$

10.8.5 Practical Considerations

1. Choice of Loss:

- Hinge: When sparsity is desired
- Logistic: When probability estimates are needed
- Exponential: When strong emphasis on hard examples is wanted

2. Robustness:

- Hinge and logistic more robust to outliers
- Exponential loss sensitive to outliers
- Hinge loss most robust to label noise

10.9 A Unified View of Modern Learning Methods

10.9.1 General Framework

All three methods can be viewed as:

$$f(x) = \sum_{k=1}^K \beta_k h_k(x) \quad (10.140)$$

where:

- $h_k(x)$ are hidden layer features
- β_k are output layer weights
- K is the dimension of hidden layer (possibly infinite)

Aspect	XGBoost	Kernel SVM	MLP
Features $h_k(x)$	Tree outputs	$\Phi(x)$ components	$\sigma(w_k^\top x + b_k)$
Number K	$\# \text{trees} \times \# \text{leaves}$	∞ (implicit)	User-specified
Design	Learned sequentially	Fixed a priori	Learned in parallel
Interpretability	High	Low	Low

Table 10.1: Comparison of hidden layer characteristics

10.9.2 Hidden Layer Characteristics

10.9.3 Feature Construction

1. XGBoost:

$$h_k(x) = \sum_{m=1}^M \sum_{l=1}^L \beta_{ml} \mathbb{I}\{x \in R_{ml}\} \quad (10.141)$$

where:

- R_{ml} are regions defined by tree splits
- Features learned adaptively via greedy splitting
- Piecewise constant approximation

2. Kernel SVM:

$$h_k(x) = [\Phi(x)]_k \quad \text{with} \quad K(x, z) = \sum_{k=1}^{\infty} h_k(x) h_k(z) \quad (10.142)$$

where:

- Features defined implicitly through kernel
- Fixed feature map (e.g., RBF, polynomial)
- Universal approximation capability

3. MLP:

$$h_k(x) = \sigma(w_k^\top x + b_k) \quad (10.143)$$

where:

- σ is activation function (e.g., ReLU)
- Features learned through backpropagation
- Compositional structure possible (deep networks)

10.9.4 Learning Paradigms

1. XGBoost:

- Sequential feature construction
- Gradient-guided splitting
- Second-order optimization
- Additive model building

2. Kernel SVM:

- Fixed feature space
- Convex optimization
- Kernel trick avoids explicit features
- Maximum margin principle

3. MLP:

- Parallel feature learning
- End-to-end gradient descent
- Backpropagation
- Deep composition possible

10.9.5 Model Complexity Control

$$\text{XGBoost: } \lambda \|w\|^2 + \gamma \# \text{leaves} + \alpha \sum_{m,l} |\beta_{ml}|$$

$$\text{Kernel SVM: } \frac{1}{2} \|f\|_{\mathcal{H}}^2 = \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j K(x_i, x_j)$$

$$\text{MLP: } \lambda \sum_l \|W_l\|_F^2 + \text{dropout/batchnorm}$$

10.9.6 Advantages and Trade-offs

1. XGBoost:

- + Interpretable features
- + Handles mixed data types
- + Natural handling of missing values

- - Limited smoothness
- - Sequential training

2. **Kernel SVM:**

- + Convex optimization
- + Theoretical guarantees
- + Flexible feature space
- - Scaling with dataset size
- - Limited interpretability

3. **MLP:**

- + End-to-end learning
- + Compositional features
- + Parallel training possible
- - Local optima
- - Requires more data

10.9.7 Practical Considerations

1. **When to Use Each:**

- XGBoost: Structured data, mixed types
- Kernel SVM: Small-medium datasets, complex boundaries
- MLP: Large datasets, raw features (images, text)

2. **Computational Scaling:**

- XGBoost: $O(nd \log n)$ per tree
- Kernel SVM: $O(n^2)$ to $O(n^3)$
- MLP: $O(ndh)$ per epoch (h = hidden size)

10.10 Model Complexity and Regularization

10.10.1 Fundamental Principle: Explaining Away Noise

The core principle underlying model complexity can be understood through the lens of noise explanation. Different complexity measures essentially quantify how well a model class can explain pure noise:

Definition 68 (Model Complexity - Informal). The complexity of a model class is its capacity to fit random patterns that contain no true underlying structure.

This manifests differently in classification and regression:

- **Classification:** Noise appears as random coin flips (± 1 labels)
- **Regression:** Noise appears as Gaussian perturbations around true values

10.10.2 Three Views of Complexity

VC Dimension

The VC dimension has a simple interpretation through noise:

Definition 69 (VC Dimension). The VC dimension h of a model class is the maximum number of points for which the model can fit any arbitrary assignment of binary labels.

Key properties:

- Measures capacity to explain random coin flips
- For linear classifiers in \mathbb{R}^d : VC dimension = $d + 1$
- Larger h implies more capacity to memorize random noise

Rademacher Complexity

Rademacher complexity provides a more nuanced, data-dependent view:

Definition 70 (Empirical Rademacher Complexity). For a function class \mathcal{F} and sample $S = \{x_1, \dots, x_n\}$, the empirical Rademacher complexity is:

$$\hat{\mathcal{R}}_n(\mathcal{F}) = \mathbb{E}_\sigma \left[\sup_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^n \sigma_i f(x_i) \right] \quad (10.144)$$

where σ_i are independent random signs (± 1).

Properties:

- Measures correlation between function class and random noise
- More data-dependent than VC dimension
- Provides tighter generalization bounds
- Natural extension to real-valued functions

Regression Complexity

In regression, complexity manifests through function smoothness:

- Degree- d polynomial can fit $d + 1$ random points perfectly
- Function smoothness limits oscillations fitting noise
- RKHS norm controls the magnitude of coefficients

10.10.3 Controlling Complexity

Margin's Role

The maximum margin principle provides natural complexity control:

1. Without margin: Linear classifier in \mathbb{R}^d has VC dimension $d + 1$
2. With margin γ : VC dimension becomes $\approx \min(R^2/\gamma^2, d)$
3. Rademacher complexity decreases proportionally to $1/\gamma$

This shows how margin constraints reduce noise-fitting capacity:

- Larger margin means fewer random labelings can be perfectly separated
- Rademacher complexity bound tightens with margin size
- Natural connection to regularization

L2 Regularization

L2 regularization provides a more general view of complexity control:

1. Linear Case:

- $\|w\|^2$ small implies large margin
- Directly limits ability to fit random labels

2. Kernel Case:

$$\|f\|_{\mathcal{H}}^2 = \sum_{i,j} \alpha_i \alpha_j K(x_i, x_j) \quad (10.145)$$

- Controls smoothness of decision boundary
- Limits oscillations that could fit random noise

10.10.4 Unified Understanding

All these views describe resistance to noise:

- VC dimension: Cannot memorize too many random labels
- Rademacher complexity: Cannot correlate strongly with noise
- RKHS norm: Cannot fit too many random points

10.10.5 Theoretical Guarantees

This leads to fundamental generalization bounds:

Theorem 71 (Rademacher Generalization Bound). *With probability $1 - \delta$, for all $f \in \mathcal{F}$:*

$$\mathbb{E}[L(f)] \leq \hat{L}(f) + 2\hat{\mathcal{R}}_n(\mathcal{F}) + \sqrt{\frac{\ln(1/\delta)}{2n}} \quad (10.146)$$

where $\hat{L}(f)$ is the empirical risk.

10.10.6 Modern Perspectives

Recent developments add nuance to this understanding:

- **Double Descent:** Overparameterization can decrease effective complexity
- **Implicit Regularization:** SGD provides algorithmic complexity control
- **Neural Tangent Kernel:** Connects deep learning to kernel methods

10.10.7 Practical Implications

This unified view guides practical decisions:

1. Model Selection:

- Choose complexity appropriate for dataset size
- Use cross-validation to detect noise-fitting

2. Regularization Choice:

- L2 for smooth functions
- L1 for sparse solutions
- Early stopping to prevent noise memorization

3. Algorithm Design:

- Balance empirical risk and complexity
- Adapt regularization to data properties
- Monitor effective model capacity