A Note on Machine Learning Methods

Ying Nian Wu, UCLA Statistics

Based on M231B lectures, Winter quarter 2022

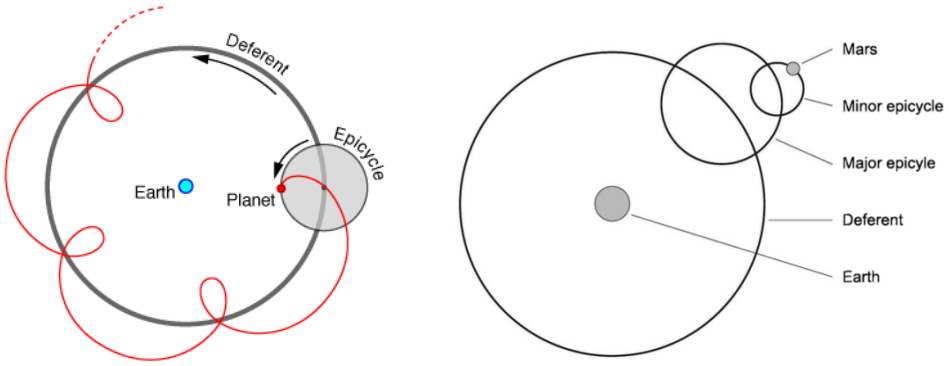
Contents

Credits and acknowledgement

Most of the figures in the note are taken from the internet. Credits and copyrights go to the original authors, together with my gratitude. Materials presented in the note are based on the original papers. References to these papers are to be added. I thank all the authors.

1 Ptolemy's epicycle, modern physics, and machine learning

1.1 Epicycle model



(a) (b)

Figure 1: Epicycle model for planet orbit, with Earth at the center. (a) One epicycle. (b) Two epicycles.

In the ancient time, the most interesting data are the trajectories of celestial bodies, i.e., their positions over time. The study of such data was as fashionable as modern machine learning and artificial intelligence. Ptolemy was perhaps the first machine learner (without using any machine) in history. His epicycle model is perhaps the first machine learning model in history.

The goal of the epicycle model is to model the motion of celestial bodies. The data consist of positions of a planet, such as Mars, over time. The model puts the Earth at the center (i.e., geocentric). The simplest model assumes that the planet moves with a uniform speed on a circle around the Earth, similar to the Moon. This model does not fit the observed data well. We can add an epicycle, so that the center of the epicycle moves on a circle around the Earth, but the planet moves on the epicycle. See Figure 8(a). If this is still not enough, we can add one more epicycle, and so on, until we have a good fit to the observed data. See Figure 8(b).

To be specific, let $(x(t), y(t))$ be the position of a planet on a plane at time t, with Earth being the origin $(0,0)$. Suppose we observe (x_i, y_i) at times t_i, $i = 1, ..., n$. We may consider (t_i) as the input, and (x_i, y_i) as the output. We want to learn a model that can predict the position of the planet at a future time. Of course we also hope to understand the physics of planetary motion.

The simplest model is a circular trajectory,

$$x(t) = r\cos(\omega t),\ y(t) = r\sin(\omega t),$$

where r is the radius and ω is the angular speed. We can also write the model using complex numbers. Let $z(t) = x(t) + iy(t)$ and $z_i = x_i + iy_i$. Then the model becomes

$$z(t) = re^{i\omega t}.$$

Ptolemy found out that a single circle did not fit the data well enough. He then assumed that the planet is moving around a center in a circular motion, and this center itself is moving around the Earth in a circular motion. This is an epicycle model that can be written as

$$z(t) = r_1 e^{i\omega_1 t} + r_2 e^{i\omega_2 t},$$

where $r_1 e^{i\omega_1 t}$ is the original circle, and $r_2 e^{i\omega_2 t}$ is the circle on top of the original circle. See Figure 8(a). If two circles are not enough, we can add the third circle. In general, we may consider a model

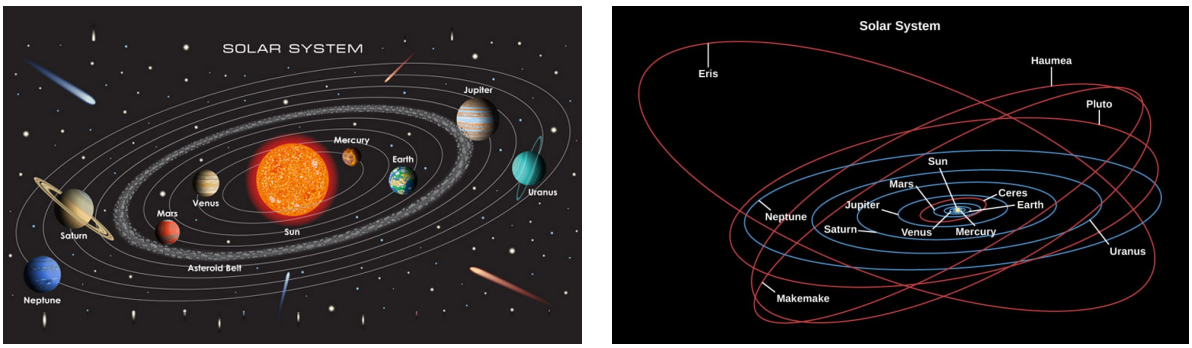
$$z(t) = \sum_{k=1}^{d} r_k e^{i\omega_k t},$$

where d (for dimensionality or degrees of freedom) is the number of circles. It defines the complexity of the model. See Figure 8(b).

This was a stroke of genius. It was a precursor to Fourier analysis. With enough circles, we can fit any trajectory. The above model is by all means a good model, as good as any model we can find in machine learning literature. It is flexible enough to fit all the trajectories. It has a clear geometric meaning.

If we add one circle at a time as was done by Ptolemy, we are actually doing boosting, which is an important learning method based on function expansion. The idea of adding cycles on top of cycles also agrees with the philosophy of neuron networks that add perceptrons on top of perceptrons.

1.2 Newtonian mechanics



(a) (b)

Figure 2: Solar system. (a) Planets. (b) Elliptical orbits of planets and dwarf planets.

Ptolemy's epicycle model turned out to be more general than necessary. Kepler found out that if we put the Sun at the center (i.e., heliocentric), the trajectories of planets relative to the Sun are ellipses. See Figure 2. Newton provided an explanation while inventing Newtonian mechanics and calculus.

According to Newton, gravity is an attractive force between two massive bodies,

$$F = G\frac{m_1 m_2}{r^2},$$

where m_1 and m_2 are the masses of the two bodies, r is the distance between the centers of the two masses, and G is a constant. Newton also assumed that

$$F = ma,$$

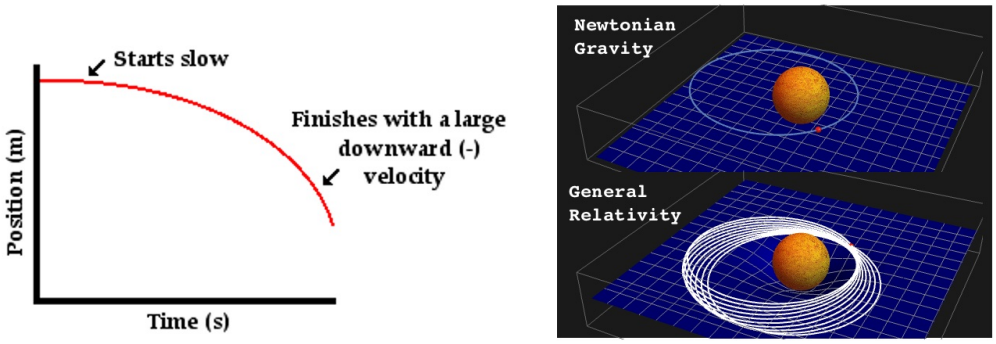
where a is the acceleration. Then Newton was able to show that the trajectories of the planets are ellipses. The model fits the data extremely well.

But what is gravitation? What is force? What is mass? Newton himself did not believe gravitation should act at a distance. Newton's model is just a mathematical model that can fit the observed data, i.e., the observed positions of planets over time. The constant G and the masses of the Sun and the planets are just

parameters and variables of this mathematical model, and these parameters and variables can be learned or inferred from the observed data. The force F is just an intermediate variable. They do not need to be "real". As long as the model fits the past data and predicts the future data, then it is a good model.

Compared to Ptolemy's epicycle model, Newton's model is simpler with a smaller number of parameters and variables, and it fits the observed data better. It is just a better model. But that does not mean Newton's model is more real. It is as much a mathematical hallucination as Ptolemy's model is. Compared to Newton's model, Ptolemy's model is actually a better machine learning model, because it is more general. If the universe is more complex than Newton's model, Ptolemy's model may still be able to fit the data.

1.3 Einstein's general relativity



(a) (b)

Figure 3: General relativity. (a) Space-time trajectory of a falling body. (b) General relativity explains Mercury's motion better than Newton's gravitation.

Einstein had the profound insight that gravity is a geometric property of space-time. If we plot the position of a falling apple over time as in Figure 3(a), the trajectory is a curve. But actually it is a "straight line". It appears curved only because the space-time is curved by the mass of the Earth, and the geodesic (i.e., shortest path or "straight line") in the curved space-time is curved (the curvature is actually not very big, because the time axis is ct, where c is the speed of light). In general, the essence of Einstein's general relativity was summarized by Wheeler as "mass tells the space-time how to curve, and space-time tells the mass how to move". Einstein's field equation is

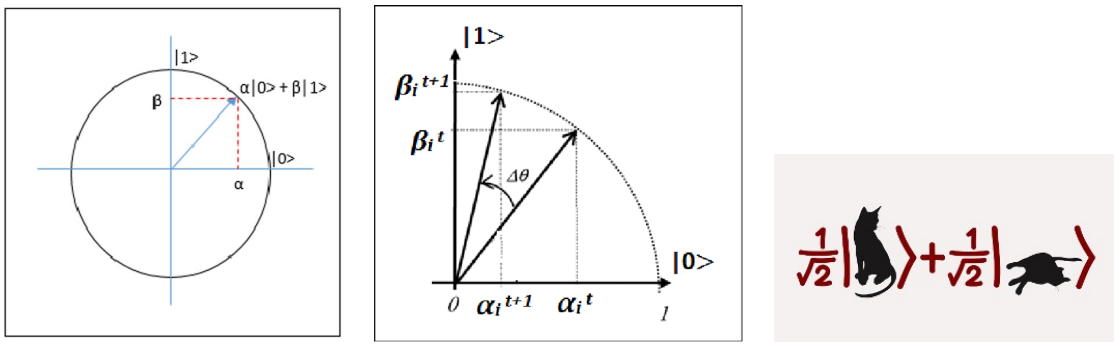
$$R_{\mu\nu} - \frac{1}{2} R g_{\mu\nu} = (8\pi G) T_{\mu\nu}, \tag{1}$$

where the left-hand side is about the curvature of space-time, and the right-hand side is energy-momentum that depends on mass. Einstein's model explains the motion of Mercury better than Newton's model, as shown in Figure 3(b).

As profound and powerful as Einstein's general relativity is, it is still a mathematical model for the observed data. The curvature of space-time is still a mathematical construct which is used to explain the observed data.

1.4 Quantum mechanics

So far we have been emphasizing the fact that the models are mathematical constructs for explaining the observed data, e.g., observed positions of the planets over time. The parameters and variables in the models do not need to be "real". This point of view is even more pronounced in quantum mechanics.



(a) (b) (c)

Figure 4: Quantum mechanics. (a) Vector representation. (b) Vector rotation. (c) Schrodinger cat.

Consider a quantum bit (qubit) that can be 0 or 1. Before we observe or measure it, its state can be represented by a vector, which is a superposition of two orthogonal basis vectors. One basis vector represents 0, which is denoted by $|0\rangle$ using Dirac's notation. The other basis vector represents 1, which is denoted by $|1\rangle$. The two vectors $|0\rangle$ and $|1\rangle$ form an orthonormal basis, and the state can be written as $v = \alpha|0\rangle + \beta|1\rangle$, where α and β are coefficients that are complex numbers. See Figure 4 (a). This also applies to the Schrödinger's cat, where $|0\rangle$ represents dead, and $|1\rangle$ represents alive. See Figure 4(c).

According to quantum mechanics, the vector rotates over time. When we measure the quantum bit using a certain equipment, we either observe 0 or observe 1. The probability of observing 0 is $|\alpha|^2$, and the probability of observing 1 is $|\beta|^2$. For the unit vector v that rotates in the space spanned by $|0\rangle$ and $|1\rangle$, its length is always 1, i.e., $\|v\|^2 = |\alpha|^2 + |\beta|^2 = 1$. That fits perfectly with the fact that the probabilities sum to 1.

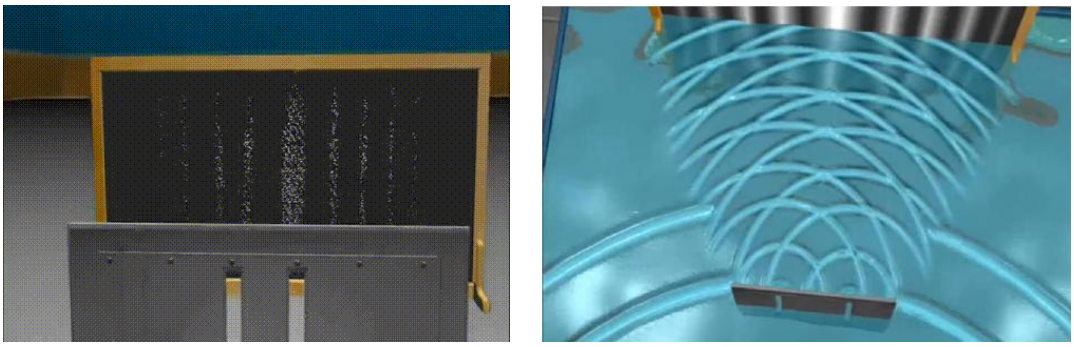
The vector representation is similar to the vector representation in deep learning. It does not exist in "reality". It exists in the mind of the observer, and it is used to predict the probability of the outcome to be observed by the observer. That is, there must be an observer, and the observer makes an observation. Quantum mechanics, in particular, the vector, enables the observer to predict the observed value before making the observation. This vector, similar to the vector representation in deep learning, is a "thought vector".

But exactly what happens before we make the observation, quantum mechanics or at least Copenhagen interpretation says that this is an irrelevant question. Quantum mechanics is only used to predict the probability distribution of the outcome to be observed by the observer. It does not describe reality beyond the observed data. This is in agreement with the goal of machine learning.

We can generalize the above scheme to continuous outcome. For instance, in the double slit experiment, we shot electrons through two slits, and the electrons are detected on the 2D screen, which is how we observe the final positions of the electrons. See Figure 5(a). The observed outcome is the continuous position x. We can write the vector representation informally as $v = \int_x \psi(x)|x\rangle$, where $\psi(x)$ is the coefficient of the vector $|x\rangle$ that represents position x. The (infinite) set of $|x\rangle$ form an orthogonal basis in a (infinite dimensional) Hilbert space (which generalizes finite dimensional Euclidean space). The vector v rotates according to the Schrodinger equation:

$$ih\frac{\partial}{\partial t}\psi_t(x) = \hat{H}\psi_t(x), \tag{2}$$

where h is the Planck constant, and $\hat{H}$ is a differential operator that involves $\partial/\partial x$ and $\partial^2/\partial x^2$ ($\hat{H}$ can be obtained from classical mechanics by changing the momentum p into an operator $-ih\partial/\partial x$, and changing p^2 into $h^2\partial^2/\partial x^2$). When we make measurement of an electron's position using the screen, the probability



(a) (b)

Figure 5: Double slit experiment. (a) Shooting electrons through two slits, and the electors are detected by the screen. (b) A single electron seems to go through two slits like water wave.

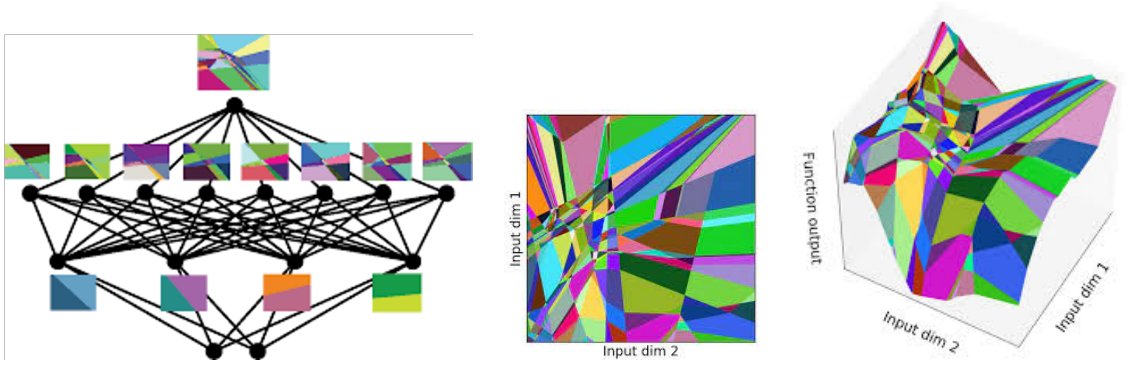
density that we observe position x is $p(x) = |\psi(x)|^2$.

The function $\psi(x)$ plays the role of the coefficients α and β in the discrete case of quantum bit. $\psi(x)$ is a continuous function that behaves like a wave. That is, a single electron can pass through the two slits like water wave, and the waves from the two slits interfere with each other like water waves.

However, the vector v again is a thought vector, and the wave function $\psi(x)$ is not a real wave. It exists only in the mind of the observer, before she observes x. After she observes x, the wave function $\psi(x)$ collapses into a point mass at x. This collapse is also in the mind of the observer. It is the collapse of uncertainty after observing x. This is the Copenhagen interpretation.

Schrodinger and Einstein never accepted the Copenhagen interpretation. After 100 years, people are still arguing about what actually happened in reality before the screen detects the position of the electron. The point is that reality is not knowable beyond observed data. A model only explains the training data and predicts the testing data.

1.5 Neural networks



(a) (b) (c)

Figure 6: ReLU network as piecewise linear function. (a) Two layer network. (b) and (c) Piecewise linear.

Neural networks can be considered epi-perceptron model, where we add perceptrons on top of perceptrons. ResNet is to add residual block on top of residual blocks. Transformer is to add self-attention block

on top of self-attention blocks. For a network with ReLU rectification, the function is piecewise linear with a large number of linear pieces. See Figure 6. It can approximate any non-linear function and can interpolate data even when the input and the output are high dimensional.

The mathematical language of neural network is based on the perceptron model $u = \sigma(Wv + b)$, where v is the input vector, W is the weight matrix, b is bias vector, u is the output vector, and $\sigma()$ is a given non-linear rectification function applied element-wise or coordinate-wise. Such a simple non-linearity creates bending of the flat linear surface. W and b can be learned from the data. We can compose the perceptrons very much like we add up epicycles. Due to the bending created by the coordinate-wise rectification, the resulting function is extremely expressive.

Each vector in the neural network model can be interpreted as activities of a group of neurons, and is a thought vector. A vector can be transformed to other vectors. A vector itself can undergo transformation over time, as in recurrent neural network. The transformation can be a composition of perceptrons. While a vector represents a "noun", the transformation of the vector represents a "verb". A vector can be computed from the input by an encoder, which is again can be a composition of perceptrons. A vector can be used to predict the outcome by a decoder. A vector can also be used to reconstruct the input by a decoder. A decoder can also be a composition of perceptrons. For image, a vector may be placed at a position in the image domain, representing the content of the image around this position, as in convolutional neural network. Sometimes a vector is also called embedding, e.g., we embed a word into a semantic space. In general, we should treat a vector as a whole that encodes some information, whereas the individual elements of the vector may not carry clear meaning. To query certain information contained in a vector, we may use a decoder to decode the information.

We can use the above mathematical language to construct a dynamic model:

$$v_{t+1} = F(v_t, a_t), \tag{3}$$

$$o_t = G(v_t), \tag{4}$$

where v_t is the thought vector at time t, o_t is the observation at time t, and a_t is the action at time t. Both the transformation F and the decoder G can be parametrized by neural networks. We can insert noises into F and G to account for randomness. This model encompasses both Newtonian mechanics and quantum mechanics. In Newtonian mechanics, v_t consists of position and momentum. a_t is acceleration caused by control force. $F(,a_t)$ is a linear mapping of v_t. o_t is the observed position. In quantum mechanics, v_t represents state at time t, $F(,a_t)$ is also a linear mapping or matrix representing a_t. The probability density of observing $o_t = o$ is $|\langle o|v_t\rangle|^2$ as discussed above, i.e., the square of the coefficient of the vector v_t on the basis vector that representing o. While v_t in Newtonian mechanics is low dimensional and has concrete meaning, v_t in quantum mechanics is infinite dimensional and is highly abstract.

If we think about the brain, o_t is the sensory data at time t, such as external data collected by our eyes, ears, touch, etc, as well as internal sensory data including pleasure and pain. a_t is the action that we can control (e.g., how we move around). v_t is an internal representation, i.e., the activities of neurons that form a thought vector. $F(,a)$ represents action a. Due to coordinate-wise non-linear rectification, $F(,a)$ is highly non-linear and more expressive than the linear matrix representation in quantum physics. We plan our actions based on such a model (together with the model for reward, which can be absorbed into observation o_t).

Again, v, $F(,a)$ and G are mental constructs. They do not need to be real at all, but they encode important information and knowledge about the world. Imagine you live in the Matrix, only o_t and a_t are real. As long as the Matrix feeds you with reasonable o_t, you will feel you are living in a real world. The Matrix may use non-Euclidean geometry with highly curved space-time and may use a very strange physics (e.g., the gravity is much weaker, and you can go through a wall) to generate o_t, but your brain will have no problem learning the dynamic model and learn to act. The model in your brain may have much more parameters and

variables than the model of the Matrix, but as long as you train your model with enough training examples, your model should be okay.

While the above discussion focuses on approximation capacity of neural networks and the representational expressiveness of vectors and transformations, the approximation capacity of neural networks also enable them to amortize iterative computations. For instance, given the input, the output may be obtained by an optimization algorithm or by solving a partial differential equation (PDE). The computation is accomplished by an iterative algorithm. But we can learn a neural network that directly maps the input to the output. This is the so-called learned computation, which can be used for both modeling and inference.

1.6 What is the point?

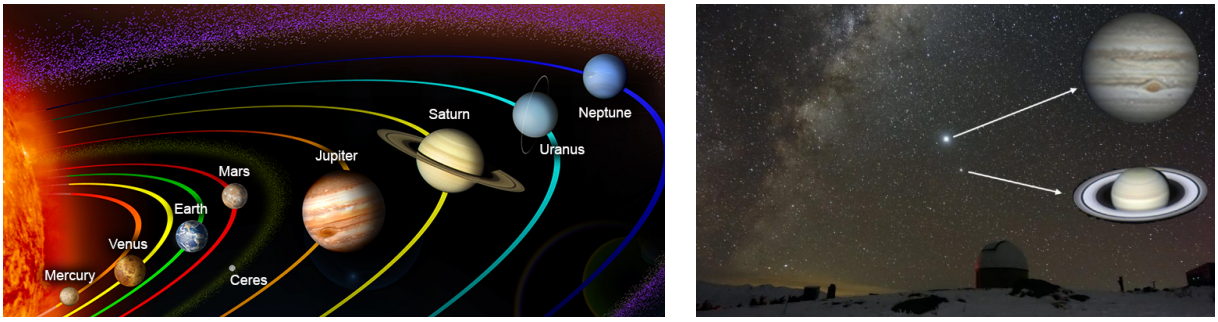
The difference between models in physics and models in machine learning is not that the former are more real or provide understanding, while the latter are just curve fitting and only for prediction. They are all about fitting the observed data. The difference is that models in physics have a smaller number of parameters and variables, and they fit the observed data extremely well. But this is because the data in physics happen to be very simple.

In science, we prefer simple models. As von Neumann said, "Give me three parameters I can fit an elephant, give me four, I can wiggle its trunk." In fact, "adding an epicycle" is synonymous to bad science. In machine learning, we also prefer simple models. But the data in machine learning are often more complex than the data in physics (positions of planets or electrons). It is usually impossible to find simple models to fit the observed data. In that case, we have to adopt models like epicycles. However, even with such models, we still prefer simple models, except that the notion of simplicity is not about counting the number of parameters. A more general notion of simplicity or complexity is to measure how much the model absorbs noises or random coincidences. We may have a model with a lot of parameters, but with explicit or implicit regularization, the model may still be very simple in that it does not overfit the training data.

2 Gauss paradigm

At the time of Euler, Laplace and Gauss, the most interesting data were about motions of celestial bodies. The analysis of such data can be considered the origin of machine learning. In fact, today in machine learning, people are still following the paradigm set up by Gauss.

2.1 Euler's model



(a) (b)

Figure 7: The motion of Jupiter around the Sun is influenced by Saturn.

Euler studied the orbit of Jupiter around the Sun. The motion of Jupiter is influenced by Saturn. By Taylor expansion or perturbation analysis, Euler obtained the following equation:

$$\begin{aligned}\varphi &= \eta - 23525'' \sin q + 168'' \sin 2q^3 2'' \sin 2w - 257'' \sin(w-q) - 243'' \sin(2w-p) + m'' - x'' \sin q + y'' \sin 2q \\ &\quad - z'' \sin(w-p) - u(\alpha + 360v + p)\cos(w-p) + Nu'' - 11405k'' \cos q + (1/600)k'' \cos 2q, \end{aligned} \tag{5}$$

where $(\varphi, \eta, q, w, p, N, v)$ are observed and vary from observation to observation, and $(x, y, m, z, \alpha, k, n, u)$ are unknown parameters. Euler had 75 observations, and by treating $u\alpha$ as γ he had 7 unknown parameters. In other words, he had 75 equations (subject to observational noises) and 7 unknowns.

Let us translate the above equation into more familiar notation in modern statistics. Let

$$y_i = \varphi_i, \ \beta = \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_p \end{pmatrix} = \begin{pmatrix} x \\ y \\ \vdots \\ u \end{pmatrix}, \ \texttt{and} \ x_i = \begin{pmatrix} x_{i1} \\ x_{i2} \\ \vdots \\ x_{ip} \end{pmatrix} = \begin{pmatrix} \sin q_i \\ \sin 2q_i \\ \vdots \\ \sin(w-p) \end{pmatrix}. \tag{6}$$

Then the equation can be written as

$$y_i = x_i^\top \beta = x_{i1}\beta_1 + x_{i2}\beta_2 + \cdots + x_{ip}\beta_p \ \texttt{for} \ i = 1, \cdots, n = 75. \tag{7}$$

The following are some observations: (1) y_i is linear in β, but it can be non-linear in the original variables (q, w, p, N, u). (2) The model is known to be correct *a priori*. While (1) is common in linear models, (2) is rare in machine learning.

Euler did not go very far in solving the above problem.

2.2 Laplace's estimating equation

Laplace proposed the method of combination of equations (for a similar dataset), e.g., we combine the 75 equations into 7 equations, so that we can solve for the 7 unknowns. Specifically we solve β from the following estimating equations:

$$\sum_{i=1}^{n} w_{ik} y_i = \sum_{i=1}^{n} w_{ik} \sum_{j=1}^{p} x_{ij}\beta_j, \ k = 1, ..., p, \tag{8}$$

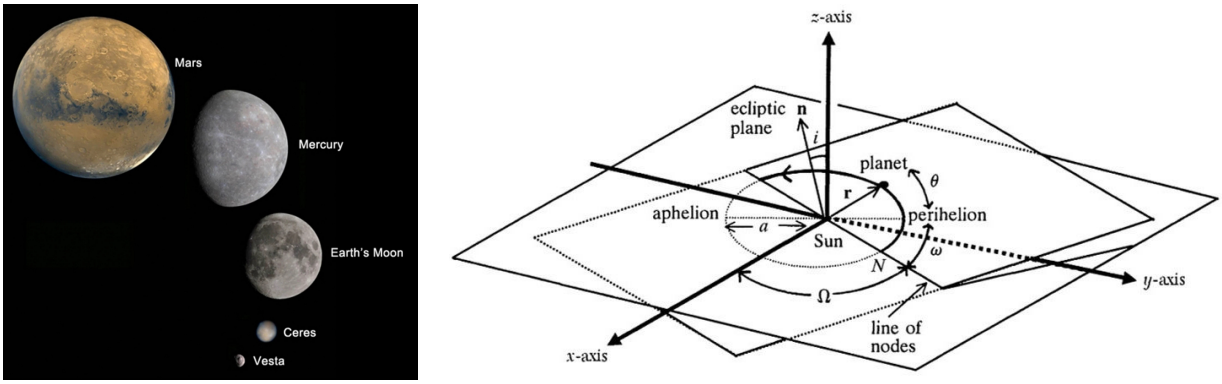
where (w_{ij}) is a set of pre-designed weights. Laplace designed a special set of weights. But he did not give a general principle on how to design the weights.

2.3 Gauss paradigm

Gauss became a celebrity at a young age by accurately predicting the position of the dwarf planet Ceres based on a small number of past observations. He used the least squares method to estimate the unknown parameters that describe the orbit of Ceres.

For linear regression model described above, the least squares method estimates β by minimizing the loss function

$$\mathscr{L}(\beta) = \sum_{i=1}^{n} \left(y_i - \sum_{i=1}^{p} x_{ij}\beta_j \right)^2. \tag{9}$$



(a) (b)

Figure 8: Ceres is a dwarf planet. It requires 7 parameters to describe the orbit of Ceres.

The above loss function can be minimized in closed form by solving the linear equation $L'(\beta) = 0$. This leads to the following estimating equation:

$$\frac{\partial}{\partial \beta_k} \mathscr{L}(\beta) = -2 \sum_{i=1}^{n} x_{ik} \left(y_i - \sum_{i=1}^{p} x_{ij} \beta_j \right) = 0, \; k = 1, ..., p. \tag{10}$$

This estimating equation corresponds to Laplace's estimating equation with $w_{ik} = x_{ik}$.

Gauss did three things that set the paradigm for statistics and machine learning.

(1) Gauss started with a loss function. In machine learning, most of the methods start from loss functions.

(2) Gauss motivated the loss function by a probabilistic formulation. He assumed that

$$y_i = \sum_{j=1}^{p} x_{ij} \beta_j + \varepsilon_i, \; \varepsilon_i \sim \mathrm{N}(0, \sigma^2), \tag{11}$$

independently for $i = 1, ..., n$. Assuming a prior distribution $p(\beta)$, the posterior distribution is

$$p(\beta | (x_i, y_i), i = 1, ..., n) \propto p(\beta) \prod_{i=1}^{n} p(y_i \mid x_i, \beta) \tag{12}$$

$$\propto \exp \left[-\frac{1}{2\sigma^2} \sum_{i=1}^{n} \left(y_i - \sum_{j=1}^{n} x_{ij} \beta_j \right)^2 + \log p(\beta) \right]. \tag{13}$$

Assuming a uniform $p(\beta)$, then maximizing $p(\beta | (x_i, y_i), i = 1, ..., n)$ is equivalent to minimizing the loss function $\mathscr{L}(\beta)$. $\prod_{i=1}^{n} p(y_i \mid x_i, \beta)$ is called likelihood. The least squares estimate is also the maximum likelihood estimate.

(3) Gauss analyzed the property of the least squares estimator $\hat{\beta}_{\mathrm{LS}}$, which is a function of $(x_i, y_i), i = 1, ..., n)$. He used Frequentist thinking, even though the loss function is motivated by Bayesian thinking. Specifically, we assume $(x_i, y_i) \sim p(x, y)$ independently, where $p(x, y)$ is the joint distribution so that the conditional distribution $p(y|x)$ is such that $y_i \sim \mathrm{N}(x_i^\top \beta_{\mathrm{true}}, \sigma^2)$, where β_{true} is the true value of β. If we believe Newtonian mechanics, then such a β_{true} does exist. In Frequentist thinking, we assume β_{true} is fixed but unknown (whereas in Bayesian thinking, we treat β as a random variable). The Frequentist thinking is based on hypothetic repeated sampling, i.e., suppose 100 astronomers collected their own data $(x_i, y_i), i = 1, ..., n) \sim p(x, y)$ independently, and each calculates his or her $\hat{\beta}_{\mathrm{LS}}$ based on least squares, then we have 100

estimated $\hat{\beta}_{\text{LS}}$. We can show that $\hat{\beta}_{\text{LS}}$ fluctuates around β_{true}, i.e., $\mathbb{E}(\hat{\beta}_{\text{LS}}) = \beta_{\text{true}}$. That is, $\hat{\beta}_{\text{LS}}$ is unbiased. The magnitude of fluctuation, i.e., $\text{Var}(\hat{\beta}_{\text{LS}})$ can also be calculated.

As to Laplace estimator based on estimating equation, we can also show that it is unbiased. We can calculate its variance too. Gauss proved that the variance of Laplace estimator is minimized at $w_{ik} = x_{ik}$, i.e., the least squares estimator achieves the minimal variance among all possible Laplace estimators. This result is called the Gauss-Markov theorem, which claims that the least squares estimator is the best linear unbiased estimator (BLUE).

We may summarize Gauss paradigm as follows.

(1) A probabilisitic model of the data.

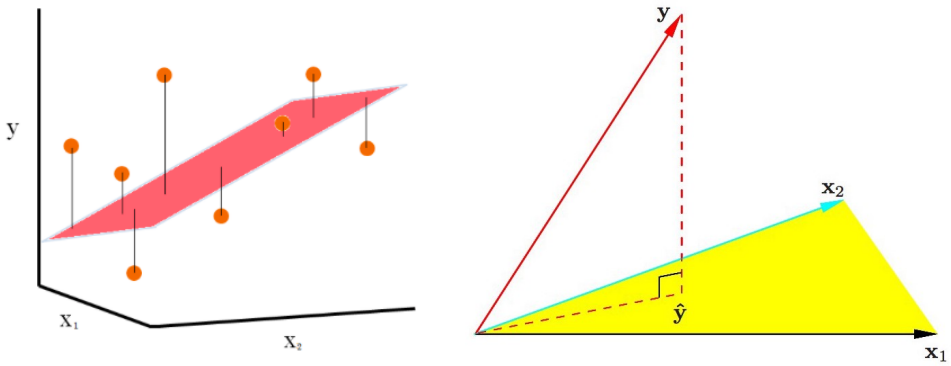
(2) Loss function based on posterior or likelihood.

(3) Analysis within the probabilisitic framework.

It is fair to say that modern statistics and machine learning did not go much beyond Gauss paradigm.

2.4 Linear regression

We now give a more systematic treatment of linear regression and least squares. The following tables illustrate the data frame and the notation of linear regression.



(a) (b)

Figure 9: Geometry of linear regression. (a) Each point is (x_i, y_i). The regression plane is $y = x^\top \beta$. (b) The vectors of $(X_j, j = 1, ..., p)$ and Y. Least squares estimation means projection of Y onto the space spanned by $(X_j, j = 1, ..., p)$.

	input	output
1	$x_{11}, x_{12}, ..., x_{1p}$	y_1
2	$x_{21}, x_{22}, ..., x_{2p}$	y_2
...		
n	$x_{n1}, x_{n2}, ..., x_{np}$	y_n

For the first table, we can write the model as $y_i = \sum_{j=1}^{p} \beta_j x_{ij} + \varepsilon_i$, and $\varepsilon_i \sim \text{N}(0, \sigma^2)$ independently.

	input	output
1	$x_1^\top$	y_1
2	$x_2^\top$	y_2
...		
n	$x_n^\top$	y_n

For the second table, $x_i = (x_{i1}, ..., x_{ip})^\top$, and we can write the model as $y_i = x_i^\top \beta + \varepsilon_i$, where $\beta = (\beta_1, ..., \beta_p)^\top$. The geometry is illustrated by Figure 9(a), where each point is (x_i, y_i), and the regression plane is $y = x^\top \beta$.

input	output
$X = (X_1, X_2, ..., X_p)$	Y

For the third table, $X_j = (x_{1j}, ..., x_{nj})^\top$, $Y = (y_1, ..., y_n)^\top$, and we can write the model as $Y = \sum_{j=1}^p X_j \beta_j + \varepsilon$, or $Y = X\beta + \varepsilon$, where $\varepsilon = (\varepsilon_1, ..., \varepsilon_n)^\top$, and $\varepsilon \sim \mathrm{N}(0, \sigma^2 I_n)$. The geometry is illustrated by Figure 9(b), where we project Y onto the subspace spanned by $(X_j, j = 1, ..., p)$.

2.5 Background: derivatives in matrix form

Suppose $Y = (y_i)_{m\times 1}$, and $X = (x_j)_{n\times 1}$. Suppose $Y = h(X)$. We can define

$$\frac{\partial Y}{\partial X^\top} = \left(\frac{\partial y_i}{\partial x_j}\right)_{m\times n}. \tag{14}$$

To understand the notation, we can treat $\partial Y = (\partial y_i, i = 1, ..., m)^\top$ as a column vector, and $1/\partial X = (1/\partial x_j, j = 1, ..., m)^\top$ as another column vector. Now we have two vectors of operations, instead of numbers. The product of the elements of the two vectors is understood as composition of the two operators, i.e., $\partial y_i (1/\partial x_j) = \partial y_i / \partial x_j$. Then $\partial Y / \partial X^\top$ is a squared matrix according to the matrix multiplication rule.

If $Y = AX$, then $y_i = \sum_k a_{ik} x_k$. Thus $\partial y_i / \partial x_j = a_{ij}$. So $\partial Y / \partial X^\top = A$.

If $Y = X^\top S X$, where S is symmetric, then $\partial Y / \partial X = 2SX$.

If $S = I$, $Y = |X|^2$, $\partial Y / \partial X = 2X$.

The chain rule in matrix form is as follows. If $Y = h(X)$ and $X = g(Z)$, then

$$\frac{\partial y_i}{\partial z_j} = \sum_k \frac{\partial y_i}{\partial x_k} \frac{\partial x_k}{\partial z_j}. \tag{15}$$

Thus

$$\frac{\partial Y}{\partial Z^\top} = \frac{\partial Y}{\partial X^\top} \frac{\partial X}{\partial Z^\top}. \tag{16}$$

2.6 Least squares estimator

For general (X, Y),

$$\mathscr{L}(\beta) = |Y - X\beta|^2. \tag{17}$$

Let $e = Y - X\beta$, then $\mathscr{L}(\beta) = |e|^2$. Applying the chain rule,

$$\frac{\partial \mathscr{L}}{\partial \beta^\top} = \frac{\partial \mathscr{L}}{\partial e^\top} \frac{\partial e}{\partial \beta^\top} = -2e^\top X, \tag{18}$$

hence

$$\mathscr{L}'(\beta) = \frac{\partial \mathscr{L}}{\partial \beta} = -2X^\top (Y - X\beta). \tag{19}$$

Setting $\mathscr{L}'(\beta) = 0$, we get the least squares estimator

$$\hat{\beta} = (X^\top X)^{-1} X^\top Y. \tag{20}$$

We can also derive the above with the more pedestrian approach by writing out the subscripts, as we did above:

$$\frac{\partial}{\partial \beta_k}\mathscr{L}(\beta) = -2\sum_{i=1}^{n} x_{ik}\left(y_i - \sum_{i=1}^{p} x_{ij}\beta_j\right) = \langle X_k, e\rangle = X_k^\top e = 0,\ k = 1,...,p, \tag{21}$$

which leads to $X^\top e = X^\top(Y - X\beta) = 0$.

2.7 Hat matrix and Cauchy-Schwartz inequality

Geometrically, $\hat{Y} = X\hat{\beta}$ is the projection of Y onto the subspace spanned by $X = (X_1,...,X_p)$, so that $e = Y - X\beta$ is perpendicular to X_j at $\hat{\beta}$, i.e., $\langle e, X_j\rangle = X_j^\top e = 0$ for $j = 1,...,p$, i.e., $X^\top(Y - X\beta) = 0$, which leads to the least squares $\hat{\beta}$. The projection is

$$\hat{Y} = X\hat{\beta} = X(X^\top X)^{-1}X^\top Y = HY, \tag{22}$$

where the hat matrix $H = X(X^\top X)^{-1}X^\top$ is the projection operation. It is easy to show that $H = H^\top$ and $H^2 = H$.

The Cauchy-Schwartz inequality is $|Y|^2 \geq |\hat{Y}|^2$, i.e.,

$$Y^\top Y \geq \hat{Y}^\top \hat{Y} = Y^\top HY. \tag{23}$$

2.8 Background: expectation and variance in matrix form

Consider a random matrix X. Suppose X is $m \times n$, and the elements of X are x_{ij}, $i = 1,...,m$ and $j = 1,...,n$. Usually we write $X = (x_{ij})_{m\times n}$ or simply $X = (x_{ij})$. We define $\mathbb{E}(X) = (\mathbb{E}(x_{ij}))$, i.e., taking expectations element-wise. Let A be a constant matrix of appropriate dimension, then $\mathbb{E}(AX) = A\mathbb{E}(X)$. Let B be another constant matrix of appropriate dimension, then $\mathbb{E}(XB) = \mathbb{E}(X)B$.

The above result can be easily understood if we have iid copies $X_1,...,X_n$, so that $\sum_{i=1}^n x_i/n \to \mathbb{E}(X)$, and $\sum_{i=1}^n Ax_i/n \to \mathbb{E}(AX)$, but also $\sum_{i=1}^n Ax_i/n = A\sum_{i=1}^n x_i/n \to A\mathbb{E}(X)$. Thus $\mathbb{E}(AX) = A\mathbb{E}(X)$.

Let X be a random vector. Let $\mu_X = \mathbb{E}(X)$. We define

$$\mathrm{Var}(X) = \mathbb{E}[(X - \mu_X)(X - \mu_X)^\top]. \tag{24}$$

Then the (i,j)-th element of $\mathrm{Var}(X)$ is $\mathrm{Cov}(x_i, x_j)$. The diagonal elements are $\mathrm{Var}(x_i)$.

Let A be a constant matrix of appropriate dimension, then

$$\mathrm{Var}(AX) = A\mathrm{Var}(X)A^\top. \tag{25}$$

This is because

$$\begin{aligned}
\mathrm{Var}(AX) &= \mathbb{E}[(AX - \mathbb{E}(AX))(AX - \mathbb{E}(AX))^\top] && (26)\\
&= \mathbb{E}[(AX - A\mu_X)(AX - A\mu_X)^\top] && (27)\\
&= \mathbb{E}[A(X - \mu_X)(X - \mu_X)^\top A^\top] && (28)\\
&= A\mathbb{E}[(X - \mu_X)(X - \mu_X)^\top]A^\top && (29)\\
&= A\mathrm{Var}(X)A^\top. && (30)
\end{aligned}$$

Note that A does not need to be a square matrix. A can even be a vector, such as $a^\top$, then $\mathrm{Var}(a^\top X) = a^\top \mathrm{Var}(X)a$, which is a quadratic form. $\mathrm{Var}(X)$ is non-negative definite.

2.9 Gauss-Markov theorem

Gauss justified the least squares estimator by proving that among all the linear unbiased estimators, including Laplace's estimating equation, the least squares estimator has the minimal variance.

In Gauss' analysis, he adopted the frequentist framework, by assuming $Y = X\beta_{\text{true}} + \varepsilon$, with $\mathbb{E}(\varepsilon) = 0$ and $\text{Var}(\varepsilon) = \sigma^2 I_n$. We assume X is fixed. For least squares estimate $\hat{\beta}_{\text{LS}}$,

$$\hat{\beta}_{\text{LS}} = (X^\top X)^{-1} X^\top Y = (X^\top X)^{-1} X^\top (X\beta_{\text{true}} + \varepsilon) = \beta_{\text{true}} + X^{-1}\varepsilon, \tag{31}$$

therefore

$$\mathbb{E}[\hat{\beta}_{\text{LS}}] = \mathbb{E}[\beta_{\text{true}} + X^{-1}\varepsilon] = \beta_{\text{true}} + X^{-1}\mathbb{E}[\varepsilon] = \beta_{\text{true}}, \tag{32}$$

and

$$\text{Var}(\hat{\beta}_{\text{LS}}) = \text{Var}(\beta_{\text{true}} + X^{-1}\varepsilon) = \text{Var}(X^{-1}\varepsilon) \tag{33}$$

$$= X^{-1}\text{Var}(\varepsilon)X^{-\top} = \sigma^2 (X^\top X)^{-1}. \tag{34}$$

For a general linear estimator $\hat{\beta} = AY$, where A may depend on X. If $\mathbb{E}(\hat{\beta}) = \beta_{\text{true}}$, then

$$\mathbb{E}[\hat{\beta}] = AX\beta_{\text{true}} = \beta_{\text{true}} \Rightarrow AX = I_p. \tag{35}$$

Further,

$$\text{Var}(\hat{\beta}) = \text{Var}(AY) = A\text{Var}(\varepsilon)A^\top = \sigma^2 AA^\top. \tag{36}$$

According to Cauchy-Schwartz inequality,

$$\text{Var}(\hat{\beta}) = \sigma^2 AA^\top \geq \sigma^2 AHA^\top = \sigma^2 AX(X^\top X)^{-1}X^\top A^\top = \sigma^2 (X^\top X)^{-1} = \text{Var}(\hat{\beta}_{\text{LS}}), \tag{37}$$

or more carefully, for any new testing input x,

$$\text{Var}(x^\top \hat{\beta}) = \sigma^2 x^\top AA^\top x \geq \text{Var}(x^\top \hat{\beta}_{\text{LS}}) = \sigma^2 x^\top (X^\top X)^{-1} x, \tag{38}$$

i.e., the least squares estimate gives us the most accurate prediction. This proves the Gauss-Markov theorem, i.e., the least squares method is the best linear unbiased estimator (BLUE).

2.10 Continuing Gauss paradigm

Gauss assumed that the prior $p(\beta)$ is a uniform distribution (within a big finite range). We can also assume $p(\beta) \sim \text{N}(0, \tau^2 I_p)$, where I_p is the p-dimensional identity matrix. This leads to the ridge regression, which maximizes the posterior $p(\beta|(x_i, y_i), i = 1, ..., n)$, or equivalently, minimizes the penalized least squares

$$\sum_{i=1}^{n} \left(y_i - \sum_{i=1}^{p} x_{ij}\beta_j \right)^2 + \lambda \sum_{j=1}^{p} \beta_j^2.$$

The penalty term leads to regularization of estimation, which is important for generalization. It is particularly important if we have a lot of variables, such as $p > n$. The ridge regression leads to kernel machine.

We may also perform a full Bayesian analysis, i.e., making multiple guesses of β by sampling from the posterior $p(\beta|(x_i, y_i), i = 1, ..., n)$, and when we make prediction, use each of the multiple guesses to predict, and average over the multiple predictions. This leads to the Gaussian process treatment.

For some problems, it may be appropriate to assume that β is sparse, i.e., only a small number of components of β are non-zero. We can use a prior distribution such as a mixture of a point mass at 0 and a normal distribution with a big variance. We can also use the sparsity inducing ℓ_1 regularization to minimize

$$\sum_{i=1}^{n}\left(y_i - \sum_{i=1}^{p} x_{ij}\beta_j\right)^2 + \lambda \sum_{j=1}^{p} |\beta_j|.$$

This is the Lasso (Least Absolute Shrinkage and Selection Operator) estimator.

2.11 Bayesian, Frequentist, variational

We can write $(x_i, y_i) \sim p_\theta(x_i, y_i)$ for $i = 1, ..., n$. In supervised learning, we let $p_\theta(x, y) = p_\theta(y|x)p(x)$, where we learn $p_\theta(y|x)$ and we leave $p(x)$ alone. In unsupervised learning, we do not observe y, and we model $p_\theta(x)$ instead.

In the following, we focus on supervised learning. In Bayesian framework, we treat θ as a random variable. We assume its marginal distribution to be $p(\theta)$. It is called the prior distribution. The learning is based on posterior distribution

$$p(\theta \mid (x_i, y_i), i = 1, ..., n) \propto p(\theta) \prod_{i=1}^{n} p(y_i \mid x_i, \theta), \tag{*}$$

where we write $p_\theta(y|x)$ as $p(y|x, \theta)$ to emphasize that θ is a random variable to be conditioned upon (we will derive the above proportionality that the end of this section). $\prod_{i=1}^{n} p(y_i \mid x_i, \theta)$ is called likelihood. $l(\theta|(x_i, y_i), i = 1, ..., n) = \sum_{i=1}^{n} \log p(y_i \mid x_i, \theta)$ is called the log-likelihood.

If we estimate θ by maximizing $p(\theta \mid (x_i, y_i), i = 1, ..., n)$ over θ, we get the so-called Maximum A Posteriori (MAP) estimate.

$$\log p(\theta \mid (x_i, y_i), i = 1, ..., n) = l(\theta|(x_i, y_i), i = 1, ..., n) + \log p(\theta).$$

If $p(\theta)$ is uniform within a range, MAP becomes maximum likelihood estimate (MLE). For non-uniform $p(\theta)$, MAP is penalized or regularized likelihood.

MAP only captures the maximum or mode of the posterior distribution but misses the uncertainty in the posterior distribution. To capture the uncertainty, we may draw multiple samples $\theta_m \sim p(\theta \mid (x_i, y_i), i = 1, ..., n)$ for $m = 1, ..., M$. This can be accomplished by Monte Carlo, such as Markov chain Monte Carlo (MCMC). These θ_m are the multiple guesses of θ.

The posterior $p(\theta \mid (x_i, y_i), i = 1, ..., n)$ is often not tractable in the sense that we cannot calculate the normalizing constant of $p(\theta) \prod_{i=1}^{n} p(y_i \mid x_i, \theta)$ to make it a probability distribution. In variational inference, we find a simpler distribution $q_\phi(\theta)$ to approximate $p(\theta \mid (x_i, y_i), i = 1, ..., n)$, where ϕ is the variational parameter that we choose to minimize the divergence from $q_\phi(\theta)$ to $p(\theta \mid (x_i, y_i), i = 1, ..., n)$.

For a testing example (x_0, y_0), if we know θ, we can predict y_0 based on $p(y_0 \mid x_0, \theta)$, where θ can be MAP or MLE. But it is better to take into account the uncertainty in θ by averaging over multiple guesses, i.e.,

$$\begin{aligned} p(y_0 \mid x_0, (x_i, y_i), i = 1, ..., n) &= \int p(y_0 \mid x_0, \theta) p(\theta \mid (x_i, y_i), i = 1, ..., n) \\ &= \mathbb{E}_{p(\theta|(x_i,y_i),i=1,...,n)}[p(y_0 \mid x_0, \theta)] \approx \frac{1}{M} \sum_{m=1}^{M} p(y_0 \mid x_0, \theta_m). \end{aligned}$$

This avoids overfitting, or as the slogan goes: Bayesian does not overfit. For some models such as Gaussian process, $p(y_0 \mid x_0, (x_i, y_i), i = 1, ..., n)$ can be obtained directly without explicitly integrating out θ.

The Frequentist thinking treats $(x_i, y_i) \sim p_{\text{data}}(x, y)$, where p_{data} is an unknown distribution that generates the data. If the model is correct, then there is a true value θ_{true}, so that $p_{\text{data}}(x, y) = p_{\text{true}}(x, y)$. We can then study the property of MLE, MAP, or Bayesian prediction within this frequentist framework.

In Gauss paradigm, he used Bayesian to motivate the least squares method, and he justified the least squares method within the Frequentist framework.

More on Learning Based on Posterior Distribution

This section provides derivation details on the conclusion that "learning is based on posterior distribution", i.e.,

$$p(\theta \mid (x_i, y_i), i = 1, ..., n) \propto p(\theta) \prod_{i=1}^{n} p(y_i \mid x_i, \theta), \tag{39}$$

According to Bayes' theorem, we can write $p(\theta \mid (x_i, y_i), i = 1, ..., n)$ as

$$p(\theta \mid (x_i, y_i), i = 1, ..., n) = \frac{p((x_i, y_i), i = 1, ..., n \mid \theta) p(\theta)}{p((x_i, y_i), i = 1, ..., n)}$$

where we treat the denominator, $p((x_i, y_i), i = 1, ..., n)$, as a constant. Therefore, we obtain

$$p(\theta \mid (x_i, y_i), i = 1, ..., n) \propto p((x_i, y_i), i = 1, ..., n \mid \theta) p(\theta)$$

where

$$p((x_i, y_i), i = 1, ..., n \mid \theta) = \prod_{i=1}^{n} p(x_i, y_i \mid \theta).$$

For $i = 1, ..., n$, we have

$$\begin{aligned} p(x_i, y_i \mid \theta) &= \frac{p(y_i \cap x_i \cap \theta)}{p(\theta)} \\ &= \frac{\frac{p(y_i \cap x_i \cap \theta)}{p(x_i \cap \theta)}}{\frac{p(\theta)}{p(x_i \cap \theta)}} \\ &= p(y_i \mid x_i, \theta) p(x_i \mid \theta) \end{aligned}$$

Here we assume x_i and θ are independent, i.e. $p(x_i \mid \theta) = p(x_i)$. Now we obtain

$$\begin{aligned} p(x_i, y_i \mid \theta) &= p(y_i \mid x_i, \theta) p(x_i) \\ p((x_i, y_i), i = 1, ..., n \mid \theta) &= \prod_{i=1}^{n} p(y_i \mid x_i, \theta) p(x_i) \end{aligned}$$

We again treat $\prod_{i=1}^{n} p(x_i)$ as a constant, and therefore we obtain

$$p(\theta \mid (x_i, y_i), i = 1, ..., n) \propto p(\theta) \prod_{i=1}^{n} p(y_i \mid x_i, \theta), \tag{*}$$

2.12 Logistic regression

We can generalize Gauss' treatment of linear regression to logistic regression and classification.

Consider a dataset with n training examples, where $x_i^\top = (x_{i1}, \cdots, x_{ip})$ consists of p predictors and $y_i \in \{0, 1\}$ is the outcome or class label.

We assume $[y_i|x_i, \beta] \sim \text{Bernoulli}(p_i)$, i.e., $\Pr(y_i = 1|x_i, \beta) = p_i$, and we assume

$$\text{logit}(p_i) = \log \frac{p_i}{1 - p_i} = s_i = x_i^\top \beta.$$

Then

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

where the sigmoid function is the inverse of the logit function.

2.13 Classification and perceptron

For logistic regression, we want to learn β either for the purpose of explanation or understanding, or for the purpose of classification or prediction. In the context of classification, we usually let $y_i \in \{+1, -1\}$ instead of $y_i \in \{1, 0\}$. Those x_i with $y_i = +1$ are called positive examples, and those x_i with $y_i = -1$ are called negative examples.

We may call β a classifier. $s_i = x_i^\top \beta = \langle x_i, \beta \rangle$ is the projection of x_i on the vector β, so the vector β is the direction that reveals the difference between positive x_i and negative x_i. Thus β should be aligned with positive x_i and negatively aligned with negative x_i, i.e., β should point from the negative examples to the positive examples.

According to the previous subsection,

$$\Pr(y_i = +1|x_i, \beta) = \frac{1}{1 + \exp(-s_i)}.$$

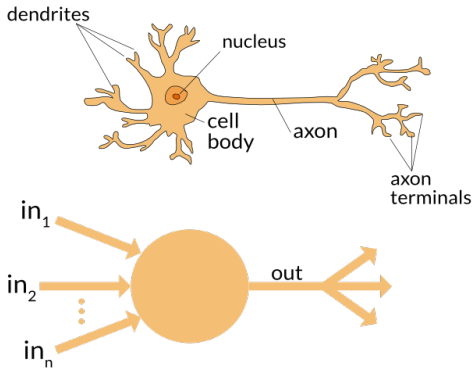


Figure 10: A neuron takes inputs, and generates outputs.

A deterministic version of the logistic regression is the perceptron model

$$y_i = \text{sign}(s_i),$$

where $\text{sign}(s) = +1$ if $s \geq 0$, and $\text{sign}(s) = -1$ if $s < 0$.

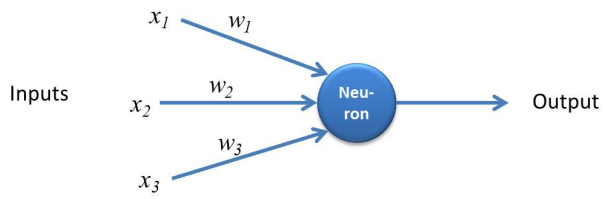


Figure 11: A perceptron is a simple model of a neuron. It computes a weighted sum of the inputs (plus a bias), and outputs the sign of the weighted sum.

The perceptron model is inspired by neuroscience. See Figure 10. It can be considered an over-simplified model of a neuron, which takes input x_i, and emits output y_i. See Figure 11.

The perceptron model can be generalized to neural networks, support vector machines, as well as adaboost, which are three major tools for classification.

Notationally, in machine learning literature, the perceptron is often written as

$$y_i = \text{sign}\left(\sum_{j=1}^{p} w_j x_{ij} + b\right) = \text{sign}(x_i^\top w + b),$$

where $w = (w_j, j = 1, ..., p)^\top$ are the connection weights and b is the bias term. (w, b) corresponds to β.

2.14 Loss functions

In order to learn β (or (w, b)) from the training data $\{(x_i, y_i), i = 1, ..., n\}$, we can minimize the loss function

$$\mathscr{L}(\beta) = \sum_{i=1}^{n} L(y_i, s_i),$$

where $L(y_i, s_i)$ is the loss for each training example (x_i, y_i). We need to define $L(y_i, s_i)$.

Loss function for least squares regression

For linear regression, we usually use the least squares loss,

$$L(y_i, s_i) = (y_i - s_i)^2.$$

Loss function for robust linear regression

We may also use the mean absolute value loss,

$$L(y_i, s_i) = |y_i - s_i|,$$

which penalizes large differences between y_i and $s_i = x_i^\top \beta$ to a less degree than the least squares loss, thus the estimated β is less affected by the outliers.

Loss function for logistic regression with 0/1 responses

For logistic regression, we usually maximize the likelihood function, which is

$$\text{Likelihood}(\beta) = \prod_{i=1}^{n} \Pr(y_i|x_i,\beta).$$

That is, we want to find β to maximize the probability of the observed $(y_i, i = 1,...,n)$ given $(x_i, i = 1,...,n)$. The maximum likelihood estimate gives the most plausible explanation to the observed data.

For $y_i \in \{0,1\}$,

$$\Pr(y_i = 1|x_i,\beta) = \text{sigmoid}(s_i) = \frac{\exp(s_i)}{1+\exp(s_i)},$$

$$\Pr(y_i = 0|x_i,\beta) = 1 - \Pr(y_i = 1|x_i,\beta) = \frac{1}{1+\exp(s_i)},$$

We can combine the above two equations by

$$\Pr(y_i|x_i,\beta) = \frac{\exp(y_i s_i)}{1+\exp(s_i)}.$$

The log-likelihood is

$$\text{LogLikelihood}(\beta) = \sum_{i=1}^{n} \log \Pr(y_i|x_i,\beta) = \sum_{i=1}^{n} \left[y_i s_i - \log(1+\exp(s_i))\right].$$

We can define the loss function as the negative log-likelihood

$$L(y_i, s_i) = -\left[y_i s_i + \log(1+\exp(s_i))\right].$$

Loss function for logistic regression with $\pm$ responses

If $y_i \in \{+1,-1\}$, we have

$$\Pr(y_i = +1|x_i,\beta) = \frac{1}{1+\exp(-s_i)},$$

and

$$\Pr(y_i = -1|x_i,\beta) = \frac{1}{1+\exp(s_i)}.$$

Combining them, we have

$$p(y_i|x_i,\beta) = \frac{1}{1+\exp(-y_i s_i)}.$$

The log-likelihood is

$$\sum_{i=1}^{n} \log \Pr(y_i|x_i,\beta) = -\sum_{i=1}^{n} \log\left(1+\exp(-y_i s_i)\right).$$

We define the loss function as the negative log-likelihood. Thus

$$L(y_i, s_i) = \log\left[1+\exp(-y_i s_i)\right].$$

This loss is called the logistic loss.

The least squares loss for linear regression can also be derived from the log-likelihood if we assume the errors follow a normal distribution.

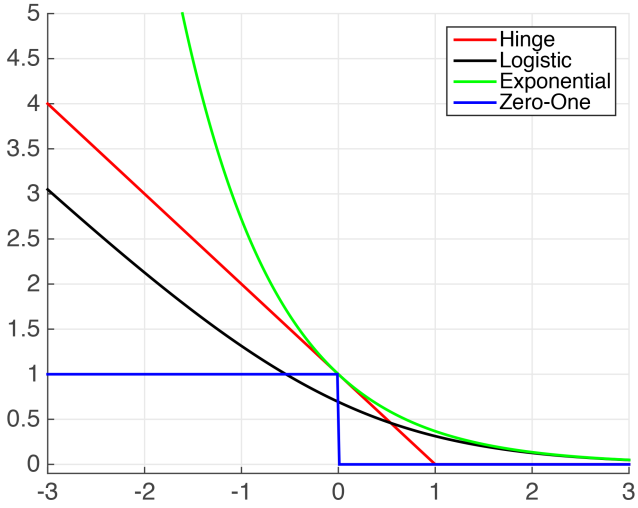


Figure 12: Loss functions for classification. The horizontal axis is $m_i = y_i x_i^\top \beta$. The vertical axis is $L(y_i, x_i^\top \beta)$. The exponential loss and the hinge loss can be considered approximations to the logistic loss. These loss functions penalize negative m_i. The more negative m_i is, the bigger the loss. The loss functions also penalize small positive m_i, e.g., those $m_i < 1$. Such loss functions encourage correct and confident classifications.

Loss functions for classification

The following summarizes several possible choices for the loss function $L(y_i, s_i)$ for classification. See Figure 12.

$$\begin{aligned}
&\text{Logistic loss} = \log\left(1 + \exp(-y_i s_i)\right), \\
&\text{Exponential loss} = \exp\left(-y_i s_i\right), \\
&\text{Hinge loss} = \max\left(0, 1 - y_i s_i\right), \\
&\text{Zero-one loss} = 1\left(y_i s_i > 0\right)
\end{aligned}$$

Both the exponential and hinge losses can be considered approximations to the logistic loss. The logistic loss is used by logistic regression. The exponential loss is used by adaboost. The hinge loss is used by support vector machines. The zero-one loss is to count the number of mistakes. It is not differentiable and is not used for training.

All the above loss functions are based on $m_i = y_i s_i$. We call m_i the margin for example (y_i, x_i). We want y_i and $s_i = x_i^\top \beta$ to be of the same sign for correct classification. If $y_i = +1$, we want $s_i = x_i^\top \beta$ to be very positive. If $y_i = -1$, we want $s_i = x_i^\top \beta$ to be very negative. We want the margin m_i to be as large as possible for confident classification.

2.15 Regularization

If we have many parameters, i.e., β is a high dimensional vector, we need to regularize β and use the following objective function,

$$\mathscr{L}(\beta) = \sum_{i=1}^{n} L(y_i, s_i) + \lambda \rho(\beta),$$

where $\rho(\beta)$ is a regularization function and λ is a regularization constant to be carefully selected or tuned.

Popular choices of $\rho(\beta)$ include: (1) ℓ_2 regularization, where $\rho(\beta) = \|\beta\|_{\ell_2}^2$, which is the sum of squares of the components of β, giving us the ridge regression; and (2) ℓ_1 regularization, where $\rho(\beta) = \|\beta\|_{\ell_1}$, which is the sum of the absolute values of the components of β. This gives us the Lasso. Both regularization functions are convex. While ℓ_2 induces shrinkage, ℓ_1 induces variable selection. Usually we do not penalize the intercept or bias. In machine learning literature, we often write β in terms of (w,b), and we only regularize w, because b only tells us the overall location of $\{y_i\}$, and it does not tell us anything about the relationship between y_i and x_i.

2.16 Gradient descent: learning from errors

With the loss function $\mathscr{L}(\beta)$ defined, we can estimate β by minimizing $\mathscr{L}(\beta)$. We can use the first order method such as gradient descent. We can also use the second order method such as Newton-Raphson (if we can afford it).

The gradient descent algorithm updates β by

$$\beta_{t+1} = \beta_t - \eta \mathscr{L}'(\beta_t),$$

where η is the step size or learning rate. It may change over time. $\mathscr{L}'(\beta)$ is the gradient or derivative of $\mathscr{L}(\beta)$, which gives us the steepest direction for descent, and

$$\mathscr{L}'(\beta) = \sum_{i=1}^{n} \frac{\partial}{\partial \beta} L_i(y_i, s_i) + \lambda \rho'(\beta).$$

For least squares loss,

$$\frac{\partial}{\partial \beta} L_i(y_i, s_i) = -2(y_i - s_i)x_i = -2e_i x_i,$$

where

$$e_i = y_i - s_i$$

is the error made by the current β.

For logistic regression with $y_i \in \{0,1\}$,

$$\begin{aligned} \frac{\partial}{\partial \beta} L_i(y_i, s_i) &= -y_i x_i + \frac{\exp(s_i)}{1+\exp(s_i)} x_i \\ &= -(y_i - p_i)x_i = -e_i x_i, \end{aligned}$$

where

$$e_i = y_i - p_i$$

is the error made by the current β. If $y_i = 0$, p_i should be close to 0. If $y_i = 1$, p_i should be close to 1. Otherwise there will be an error which causes the change in β. The gradient descent algorithm learns from the errors, by incorporating $e_i x_i$ into β, allowing β memorize those x_i on which it had made mistakes. In the classification setting, we want β to be a vector that points from negative examples to the positive examples, and the gradient descent algorithm achieves that.

2.17 Newton-Raphson

A more efficient method is to update β using Newton-Raphson algorithm. Suppose we want to solve $h(x) = 0$. At x_t, we take the first order Taylor expansion

$$h(x) \doteq h(x_t) + h'(x_t)(x - x_t).$$

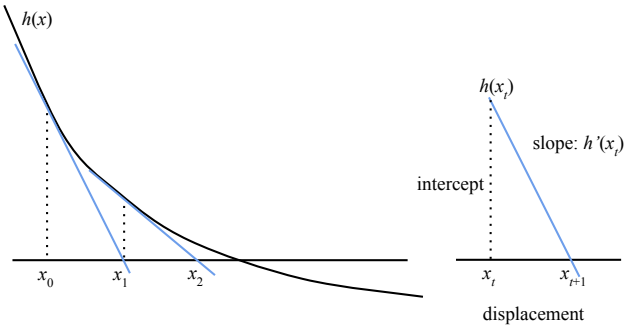


Figure 13: Newton-Raphson

Each iteration, we find the root of the linear surrogate function, which is the above first order Taylor expansion,

$$x_{t+1} = x_t - \frac{h(x_t)}{h'(x_t)}.$$

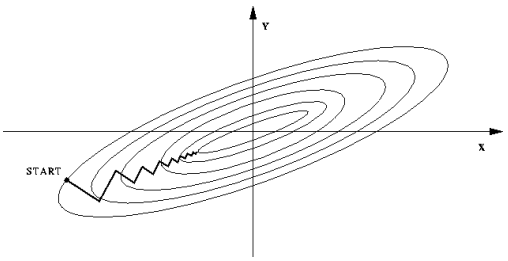


Figure 14: Gradient descent.

Suppose we want to find the mode of $f(x)$, we can solve $f'(x) = 0$. Using Newton-Raphson, we have

$$x_{t+1} = x_t - \frac{f'(x_t)}{f''(x_t)}.$$

Each iteration maximizes a quadratic approximation to the original function at x_t,

$$f(x) \doteq f(x_t) + f'(x_t)(x - x_t) + \frac{1}{2} f''(x_t)(x - x_t)^2.$$

$f''(x_t)$ is the curvature of f at x_t. If the curvature is big, the step size should be small. If the curvature is small, the step size can be made larger.

If the variable is a vector $x = (x_1, x_2, \cdots, x_n)^\top$, let

$$f'(x) = \left(\frac{\partial f}{\partial x_i}\right)_{n\times 1} \quad f''(x) = \left(\frac{\partial f}{\partial x_i \partial x_j}\right)_{n\times n},$$

$f''(x_t)$ is called the Hessian matrix, we have

$$x_{t+1} = x_t - f''(x_t)^{-1} f'(x_t).$$

$f''(x_t)$ tells us the local shape of f around x_t. $f''(x_t)^{-1}f'(x_t)$ gives us better direction than $f'(x_t)$ as shown in the above figure. The Newton-Raphson is a second order algorithm.

2.18 Iterated reweighed least squares

For maximum likelihood estimate of β in logistic regression, let $l(\beta)$ be the log-likelihood,

$$l(\beta) = \text{LogLikelihood}(\beta) = \sum_{i=1}^{n} \log \Pr(y_i|x_i,\beta) = \sum_{i=1}^{n} \left[y_i s_i - \log(1+\exp(s_i))\right].$$

To find the maximum of $l(\beta)$, we first calculate the gradient

$$l'(\beta) = \sum_{i=1}^{n} \left[y_i x_i - \frac{e^{x_i^\top \beta}}{1+e^{x_i^\top \beta}} x_i \right] = \sum_{i=1}^{n} (y_i - p_i) x_i.$$

The second derivative of the log likelihood function is

$$l''(\beta) = -\sum_{i=1}^{n} p_i(1-p_i) x_i x_i^\top.$$

We can update β by

$$\beta^{(t+1)} = \beta^{(t)} + l''(\beta^{(t)})^{-1} l'(\beta^{(t)}).$$

Let $w_i = p_i(1-p_i)$, we can rewrite the update equation as

$$\begin{aligned}
\beta^{(t+1)} &= \beta^{(t)} + \left[\sum_{i=1}^{n} p_i(1-p_i) x_i x_i^\top\right]^{-1} (y_i - p_i) x_i \\
&= \left(\sum_{i=1}^{n} w_i x_i x_i^\top\right)^{-1} \left[\sum_{i=1}^{n} w_i x_i x_i^\top \beta^{(t)} + (y_i - p_i) x_i\right] \\
&= \left(\sum_{i=1}^{n} w_i x_i x_i^\top\right)^{-1} \left[\sum_{i=1}^{n} w_i x_i \left(x_i^\top \beta^{(t)} + \frac{y_i - p_i}{w_i}\right)\right].
\end{aligned}$$

Let

$$\hat{y}_i = x_i^\top \beta^{(t)} + \frac{y_i - p_i}{w_i},$$

let $\tilde{x}_i = x_i\sqrt{w_i}, \tilde{y}_i = \hat{y}_i\sqrt{w_i}$, we can rewrite the equation above as follows:

$$\begin{aligned}
\beta^{(t+1)} &= \left(\sum_{i=1}^{n} w_i x_i x_i^\top\right)^{-1} \left(\sum_{i=1}^{n} w_i x_i \hat{y}_i\right) \\
&= \left(\sum_{i=1}^{n} \tilde{x}_i \tilde{x}_i^\top\right)^{-1} \left(\sum_{i=1}^{n} \tilde{x}_i \tilde{y}_i\right).
\end{aligned}$$

The flow is

$\beta^{(t)} \to \eta_i = x_i^\top \beta^{(t)} \to p_i = \sigma(\eta_i) \to w_i = p_i(1-p_i) \to \hat{y}_i = \eta_i + \frac{y_i - p_i}{w_i} \to \tilde{x}_i = x_i\sqrt{w_i}, \tilde{y}_i = \hat{y}_i\sqrt{w_i} \to \beta^{(t+1)}$.

XGboost can be considered a variation of iterated reweighed least squares.

2.19 Three modes of learning

Supervised learning

The table below displays the dataset for supervised learning. $h_i = (h_{ik}, k = 1, ..., d)^\top$ is the d-dimensional

	input	features	output
1	$x_1^\top$	$h_1^\top$	y_1
2	$x_2^\top$	$h_2^\top$	y_2
...			
n	$x_n^\top$	$h_n^\top$	y_n

vector of features or hidden variables. y_i follows a linear model on h_i. Euler's model can be written in this form, where x_i are the raw input variables $(\varphi, \eta, q, w, p, N, v)$, and $h(x_i)$ are derived from perturbation analysis based on Newtonian mechanism, such as $(\sin q_i, \sin 2q_i, ..., \sin(w-p))$.

The supervised learning can be represented by the diagram below,

$$\begin{array}{rc} \text{output}: & y_i \\ & \uparrow \\ \text{features}: & h_i \\ & \uparrow \\ \text{input}: & x_i \end{array}$$

where the vector of features h_i is computed from x_i via $h_i = h(x_i)$.

Encoder and decoder: In the above diagram, the transformation $x_i \to h_i$ is called an encoder, and the transformation $h_i \to y_i$ is called a decoder.

Both classification and regression are about supervised learning because for each input x_i, an output y_i is provided as supervision. In regression, y_i is continuous. In classification, y_i is categorical. We can represent y_i by a one-hot vector, i.e., if y_i denotes the k-th category, then y_i is a vector where the k-th element is 1 and all the other elements are 0.

Unsupervised learning

In unsupervised learning, the dataset is as below, where y_i are not provided as supervision.

	input	hidden	output
1	$x_1^\top$	$h_1^\top$	?
2	$x_2^\top$	$h_2^\top$	?
...			
n	$x_n^\top$	$h_n^\top$	?

In a generative model, the vector h_i is not a vector of features extracted from the signal x_i. h_i is a vector of hidden variables that is used to generate x_i, as illustrated by the following diagram:

$$\begin{array}{rc} \text{hidden}: & h_i \\ & \downarrow \\ \text{input}: & x_i \end{array}$$

The components of the d-dimensional h_i are variably called factors, sources, components or causes. The prototype example is factor analysis or principal component analysis.

Auto-encoder: h_i is also called a code in the auto-encoder illustrated by the following diagram:

$$\begin{array}{ll} \text{code}: & h_i \\ & \uparrow\downarrow \\ \text{input}: & x_i \end{array}$$

The direction from h_i to x_i is called the decoder, and the direction from x_i to h_i is called the encoder.

Distributed representation and disentanglement: $h_i = (h_{ik}, k = 1, ..., d)$ is called a distributed representation of x_i. Usually the components of h_i, $(h_{ik}, k = 1, ..., d)$, are assumed to be independent, and (h_{ik}) are said to disentangle the variations in x_i.

Embedding: h_i can also be considered the coordinates of x_i, if we embed x_i in a low-dimensional space, as illustrated by the following diagram:

$$\begin{array}{c} \leftarrow h_i \rightarrow \\ | \\ \leftarrow x_i \rightarrow \end{array}$$

In the training data, we find a h_i for each x_i, so that $\{h_i, i = 1, ..., n\}$ preserve the relative relations between $\{x_i, i = 1, ..., n\}$. The prototype example of embedding is multi-dimensional scaling, where we want to preserve the Euclidean distances between the examples.

Reinforcement learning

Reinforcement learning is similar to supervised learning except that the guidance is in the form of reward. Here x_i is the state. y_i can be the action taken at this state. y_i can also be the value of this state, where value is defined as the accumulated reward.

3 Stein Estimator

3.1 Bias and variance tradeoff

While Gauss showed that $\hat{\beta}_{\text{LS}}$ has minimal variance among all the linear unbiased estimators, one may ask whether a biased estimator may be even better than the least squares estimator. For an estimator $\hat{\beta}$, let $\mu = \mathbb{E}(\hat{\beta})$. The mean squares error can be decomposed into bias and variance terms,

$$\begin{aligned} \mathbb{E}\|\hat{\beta} - \beta_{\text{true}}\|^2 &= \mathbb{E}\|(\hat{\beta} - \mu) + (\mu - \beta_{\text{true}})\|^2 \\ &= \mathbb{E}\|\hat{\beta} - \mu\|^2 + \|\mu - \beta_{\text{true}}\|^2 = \textit{variance} + \textit{bias}^2. \end{aligned}$$

The figure below illustrates the bias and variance tradeoff. It is possible to introduce some bias so as to reduce the variance.

3.2 Shrinkage estimator

Stein shocked the world of statistics in 1960s by showing that as long as $p > 2$, you can always do better than least squares estimator. While Gauss-Markov theorem states that least squares method is the best linear unbiased estimator, Stein's estimator is neither linear nor unbiased.

For simplicity, let us assume that $X^\top X = I_p$, i.e., $(X_j, j = 1, ..., p)$ form an orthonormal basis. Then

$$\hat{\beta}_{\text{LS}} = X^\top (X\beta_{\text{true}} + \varepsilon) = \beta_{\text{true}} + X^\top \varepsilon = \beta_{\text{true}} + \delta,$$

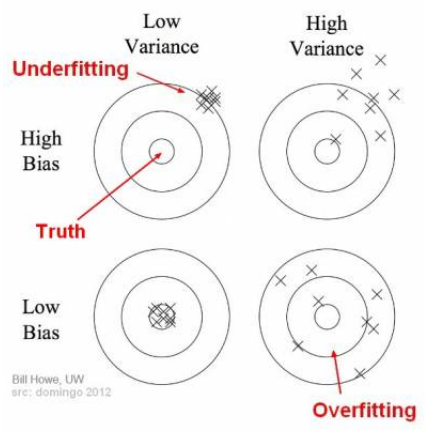


Figure 15: Bias and variance tradeoff.

where $\delta = X^\top \varepsilon$ is the model fitting to the noise ε. Suppose $\varepsilon \sim \mathrm{N}(0, \sigma^2 I_n)$, then $\delta \sim \mathrm{N}(0, \sigma^2 I_p)$, because $\mathrm{Var}(\delta) = X^\top \mathrm{Var}(\varepsilon) X = \sigma^2 I_p$. Thus $\hat{\beta}_{\mathrm{LS}} \sim \mathrm{N}(\beta_{\mathrm{true}}, \sigma^2 I_p)$.

Stein's estimator is defined as

$$\hat{\beta}_{\mathrm{Stein}} = \left(1 - \frac{(p-2)\sigma^2}{\|\hat{\beta}_{\mathrm{LS}}\|^2}\right) \hat{\beta}_{\mathrm{LS}}.$$

Stein proved that for $p > 2$

$$\mathbb{E}\|\hat{\beta}_{\mathrm{Stein}} - \beta_{\mathrm{true}}\|^2 \leq \mathbb{E}\|\hat{\beta}_{\mathrm{LS}} - \beta_{\mathrm{true}}\|^2.$$

Stein's estimator is based on the observation that $\|\hat{\beta}_{\mathrm{LS}}\|^2 = \|\beta_{\mathrm{true}}\|^2 + p\sigma^2$. That is, the vector $\hat{\beta}_{\mathrm{LS}}$ is longer than the vector β_{true}. Thus we may want to shrink $\hat{\beta}_{\mathrm{LS}}$ to bring it closer to β_{true}. Stein's estimator is an example of shrinkage estimator.

3.3 Stein lemma

For $z \sim \mathrm{N}(\mu, \sigma^2)$, $\mathbb{E}[(z-\mu)g(z)] = \sigma^2 \mathbb{E}[g'(z)]$.
proof Integral by parts,

$$\begin{aligned}
\mathbb{E}[(z-\mu)g(z)] &= \int (z-\mu)g(z) \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(z-\mu)^2}{2\sigma^2}} dz \\
&= -\sigma^2 g(z) \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(z-\mu)^2}{2\sigma^2}} \Big|_{-\infty}^{\infty} + \sigma^2 \int g'(z) \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(z-\mu)^2}{2\sigma^2}} \\
&= \sigma^2 \mathbb{E}\left[g'(z)\right].
\end{aligned}$$

3.4 Proof of Stein's result

For simplicity, we denote $\hat{\beta}_{\mathrm{LS}}$ by X, and denote β_{true} by θ, so that $X \sim \mathrm{N}(\theta, \sigma^2 I_p)$.

$$\mathbb{E}\|X - \theta\|^2 = \mathbb{E}\left[\sum_{i=1}^{p} (X_i - \theta_i)^2\right] = p\sigma^2.$$

$$\mathbb{E}\left[\left|\left|\left(1-\frac{(p-2)\sigma^2}{\|X\|^2}\right)X-\theta\right|\right|^2\right] = E\left[\left|\left|(X-\theta)-\frac{(p-2)\sigma^2}{\|X\|^2}X\right|\right|^2\right]$$
$$= \mathbb{E}\|X-\theta\|^2+\mathbb{E}\left[\frac{(p-2)^2\sigma^4}{\|X\|^2}\right]-2\mathbb{E}\left[\langle X-\theta,\frac{(p-2)\sigma^2}{\|X\|^2}X\rangle\right]=(*).$$

$\mathbb{E}\left[\langle X-\theta,\frac{(p-2)\sigma^2}{\|X\|^2}X\rangle\right]$ can be simplified as follows

$$\mathbb{E}\left[\sum_{i=1}^{p}(X_i-\theta_i)\frac{(p-2)\sigma^2}{\|X\|^2}X_i\right] = \mathbb{E}\left[\sum_{i=1}^{p}(X_i-\theta_i)\frac{(p-2)\sigma^2}{X_i^2+\sum_{j\neq i}^{p}X_j^2}X_i\right].$$

Letting

$$\frac{(p-2)\sigma^2}{X_i^2+\sum_{j\neq i}^{p}X_j^2}X_i=g(x_i), \tag{40}$$

we can use Stein's lemma. Hence

$$\begin{aligned}\mathbb{E}\left[\sum_{i=1}^{p}(X_i-\theta_i)\frac{(p-2)\sigma^2}{X_i^2+\sum_{j\neq i}^{p}X_j^2}X_i\right] &= \sigma^2\sum_{i=1}^{p}\mathbb{E}\left[\left(\frac{(p-2)\sigma^2}{X_i^2+\sum_{j\neq i}^{p}X_j^2}X_i\right)'\right]\\ &= \sigma^2\sum_{i=1}^{p}\mathbb{E}\left[\frac{(p-2)\sigma^2}{\|X\|^2}-\frac{(p-2)\sigma^2 2X_i^2}{\|X\|^2}\right]\\ &= \sigma^2\mathbb{E}\left[\frac{p(p-2)\sigma^2}{\|X\|^2}-\frac{2(p-2)\sigma^2}{\|X\|^2}\right]\\ &= \mathbb{E}\left[\frac{(p-2)^2\sigma^4}{\|X\|^2}\right].\end{aligned}$$

Therefore,

$$\begin{aligned}(*) &= \mathbb{E}\left[\|X-\theta\|^2+\frac{(p-2)^2\sigma^4}{\|X\|^2}-2\frac{(p-2)^2\sigma^4}{\|X\|^2}\right]\\ &= \mathbb{E}\left[\|X-\theta\|^2-\frac{(p-2)^2\sigma^4}{\|X\|^2}\right]\leq\mathbb{E}\left[\|X-\theta\|^2\right]=p\sigma^2.\end{aligned}$$

3.5 Stein estimator as empirical Bayes

$X\sim\mathrm{N}(\theta,\sigma^2 I_p)$ can be written as $X=(x_i,i=1,...,p)^\top$, and $x_i\sim\mathrm{N}(\theta_i,\sigma^2)$. We can adopt a Bayesian treatment, by assuming a prior distribution $\theta_i\sim\mathrm{N}(0,\tau^2)$. Marginally, if we integrate out θ_i, we have $x_i\sim\mathrm{N}(0,\sigma^2+\tau^2)$, and the posterior distribution of each θ_i is

$$p(\theta_i\mid X)\sim\mathrm{N}\left(x_i\frac{1/\sigma^2}{1/\sigma^2+1/\tau^2},\frac{1}{1/\sigma^2+1/\tau^2}\right),$$

and

$$\mathbb{E}(\theta_i\mid X)=x_i\left(1-\frac{\sigma^2}{\sigma^2+\tau^2}\right).$$

Since $x_i\sim\mathrm{N}(0,\sigma^2+\tau^2)$, we have

$$\mathbb{E}(1/\|X\|^2)=1/[(\sigma^2+\tau^2)(p-2)].$$

Thus we can estimate $1/(\sigma^2+\tau^2)$ by $(p-2)/\|X\|^2$. This leads to Stein's estimator.

Gaussian process is a generalization of this idea. The estimation of the hyper-parameters in Gaussian process is similar to the estimation of τ^2. It is called empirical Bayes because τ^2 or $1/(\sigma^2+\tau^2)$ is estimated from empirical data. A full Bayesian treatment would put a prior on τ^2.

4 Model Complexity and Overfitting

4.1 Regression

4.1.1 Model bias

In Gauss' analysis, he assumed the model is true, or the Newtonian mechanics is true. According to George Box: "All models are wrong, but some are useful." Thus it is better to assume that the model is wrong or biased.

Suppose $Y = g+\varepsilon$, where g is the vector of ground truth, e.g., the true positions of a planet over time, ε is the noise vector. For simplicity, let us also assume that $X^\top X = I_p$, i.e., $(X_j, j = 1,...,p)$ form an orthonormal basis. Then

$$\hat{\beta} = X^\top(g+\varepsilon) = X^\top g + X^\top \varepsilon = \beta^* + \delta,$$

where $\beta^* = X^\top g$ is the "best" value of β, i.e., the best the model can do to fit the ground truth, and $\delta = X^\top \varepsilon$ is the model fitting to the noise ε. $\hat{\beta}_j = \beta^*_j + \delta_j$, for $j = 1,...,p$.

Let $\hat{g} = X\beta^*$, then $\hat{g}$ is the projection of g onto the subspace spanned by X. We call $|g-\hat{g}|^2$ the model bias.

Let $\hat{\varepsilon} = X\delta$, then $\hat{\varepsilon}$ is the projection of ε onto the space X.

Suppose $\varepsilon \sim \mathrm{N}(0,\sigma^2 I_n)$, then $\delta \sim \mathrm{N}(0,\sigma^2 I_p)$, because $\mathrm{Var}(\delta) = X^\top \mathrm{Var}(\varepsilon) X = \sigma^2 I_p$, and $\|X\delta\|^2 = \delta^\top X^\top X\delta = \|\delta\|^2 = p\sigma^2$.

4.1.2 Training and testing errors, overfitting

The training error is

$$\begin{aligned}\mathbb{E}|Y-\hat{Y}|^2 &= |g-\hat{g}|^2 + \mathbb{E}|\varepsilon-\hat{\varepsilon}|^2 \\ &= |g-\hat{g}|^2 + \mathbb{E}|\varepsilon|^2 - \mathbb{E}|\hat{\varepsilon}|^2 \\ &= |g-\hat{g}|^2 + (n-p)\sigma^2.\end{aligned}$$

Suppose X is fixed, and the test data is $\tilde{Y} = g+\tilde{\varepsilon}$, where $\tilde{\varepsilon}$ is independent of ε. Then the testing error is

$$\begin{aligned}\mathbb{E}|\tilde{Y}-\hat{Y}|^2 &= |g-\hat{g}|^2 + \mathbb{E}|\tilde{\varepsilon}-\hat{\varepsilon}|^2 \\ &= |g-\hat{g}|^2 + \mathbb{E}|\tilde{\varepsilon}|^2 + \mathbb{E}|\hat{\varepsilon}|^2 \\ &= |g-\hat{g}|^2 + (n+p)\sigma^2.\end{aligned}$$

The error of β^* is $|g-\hat{g}|^2$.

Thus overfitting is defined as the testing error minus the training error, which is $2p\sigma^2$.

We may also interpret overfitting as the error of $\hat{\beta}$ minus the error of β^*, which is $p\sigma^2$. On the training data, the learned $\hat{\beta}$ does even better than the ground truth β^*. However, on the testing data, the former does worse than the latter.

When we increase the model complexity p, the training error will keep decreasing. The testing error will decrease and then start to increase, that is, the testing error has a U-shape. See Figure 16.

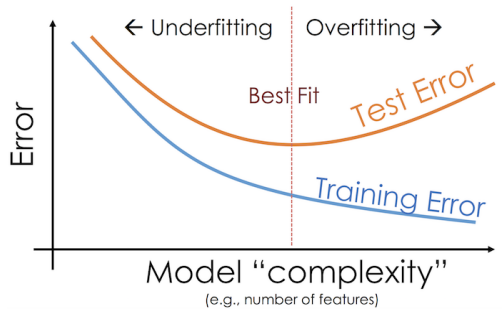


Figure 16: Training and testing errors over model complexity.

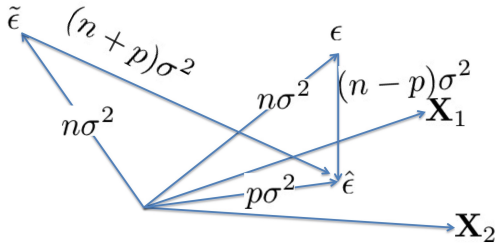


Figure 17: Overfitting due to learning from noise. Training error is $\mathbb{E}|\varepsilon - \hat{\varepsilon}|^2 = (n-p)\sigma^2$, and testing error is $\mathbb{E}|\tilde{\varepsilon} - \hat{\varepsilon}|^2 = (n+p)\sigma^2$.

4.1.3 Learning from noise

If $g = 0$, then the model learns from noise, and it becomes clear how the model overfits. See Figure 17.

The training error is $\mathbb{E}|\varepsilon - \hat{\varepsilon}|^2$, and the testing error is $\mathbb{E}|\tilde{\varepsilon} - \hat{\varepsilon}|^2$. While $\hat{\varepsilon}$ is the projection of ε onto X, it is not the projection of $\tilde{\varepsilon}$ onto X.

4.1.4 Effective degrees of freedom

Consider a general estimator $\hat{\beta}$ such as the ridge estimator. The training error is

$$\mathbb{E}|Y - X\hat{\beta}|^2 = \mathbb{E}|g + \varepsilon - X\hat{\beta}|^2.$$

The testing error is

$$\mathbb{E}|\tilde{Y} - X\hat{\beta}|^2 = \mathbb{E}|g + \tilde{\varepsilon} - X\hat{\beta}|^2.$$

The overfitting = testing error - training error is

$$2\mathbb{E}\langle \varepsilon, X\hat{\beta} \rangle,$$

which is how much $\hat{\beta}$ absorbs the noise ε.

Recall for the least squares $\hat{\beta}$ overfitting is $2p\sigma^2$. We can define the effective degrees of freedom of $\hat{\beta}$ to be

$$\hat{p} = \frac{\mathbb{E}\langle \varepsilon, X\hat{\beta} \rangle}{\sigma^2}.$$

With stronger regularization, e.g., bigger λ in the ridge regression, we have smaller $\hat{p}$ and less overfitting.

For a general estimator, the testing error can be decomposed into model bias, estimation bias, estimation variance, and noise,

$$\mathbb{E}|\tilde{Y} - X\hat{\beta}|^2 = |g - \hat{g}|^2 + |\mathbb{E}(\hat{\beta}) - \beta^*|^2 + \mathbb{E}|\hat{\beta} - \mathbb{E}(\hat{\beta})|^2 + n\sigma^2.$$

As we increase model complexity, the model bias decreases, the estimation bias decreases, but the variance increases.

4.2 Classification

4.2.1 Error and score

Suppose the classifier is $\hat{y} = f(x) \in \{-1, +1\}$. The classification error is $1(f(x) \neq y)$, and the classification score is $f(x)y$. Clearly score = 1 - 2 $\times$ error. We shall analyze the training score and testing score. Note that here we treat $f(x)$ as the binary-valued function, unless in other parts of the note.

4.2.2 Coin flipping

We shall first study the model fitting of "coin flippings," i.e., we assume $y = \varepsilon \sim$ Bernoulli$(1/2)$, i.e., $\Pr(\varepsilon = 1) = \Pr(\varepsilon = -1) = 1/2$.

A property of coin flipping is that for any Bernoulli random variable $Z \sim$ Bernoulli$(p) \in \{+1, -1\}$ that is independent of ε, we have $\varepsilon Z \sim$ Bernoulli$(1/2)$.

4.2.3 Learning from coin flipping and Rademacher complexity

For any classifier $f(x)$, the testing score is $\mathbb{E}[\varepsilon f(x)] = 0$.

Suppose our dataset is $(x_i, y_i), i = 1, ..., n$ where $y_i = \varepsilon_i \sim$ Bernoulli(1/2) that follows a coin flipping process. For a classifier $f(x)$, the training score is $\frac{1}{n}\sum_{i=1}^n y_i f(x_i)$. However, in the training stage, we can select $f \in \mathscr{F}$, which is the class of all possible classifiers. For instance, if $f(x) = \text{sign}(x^\top \beta)$ is a linear classifier, then $\mathscr{F} = \{f(x) = \text{sign}(x^\top \beta), \forall \beta\}$. Then the training score of the optimal classifier is

$$R = \mathbb{E}\left[\max_{f \in \mathscr{F}} \frac{1}{n}\sum_{i=1}^n \varepsilon_i f(x_i)\right].$$

The above is the Rademacher complexity of the class $\mathscr{F}$. It is similar to the effective degrees of freedom in regression.

4.2.4 Growth number and VC-dimension

Let $N = |\{(f(x_1), ..., f(x_n)), \forall f \in \mathscr{F}\}|$ be the number of all possible predicted sequences $(f(x_1), ..., f(x_n))$ if we vary f in $\mathscr{F}$. Clearly $N \leq 2^n$. N is called the growth number. The growth number plays a role in the union bound of the training error, i.e.,

$$\Pr\left(\max_{f \in \mathscr{F}} \frac{1}{n}\sum_{i=1}^n \varepsilon_i f(x_i) > t\right) \leq N \Pr\left(\frac{1}{n}\sum_{i=1}^n \varepsilon_i > t\right) \leq N \exp(-2nt^2).$$

The union bound is that $P(A \cup B) \leq P(A) + P(B)$. Consider taking an exam N times and suppose each time the probability of passing the exam is p. If you take the maximal score, then you increase your chance of passing the exam. But this chance is still bounded by Np. The exponential term above is due to the concentration inequality.

Recall $N \leq 2^n$. The VC-dimension is the largest n, so that $N = 2^n$, i.e., the number of coin flippings that can be perfectly explained by selecting f from $\mathscr{F}$. Recall in the linear regression model, the dimension p is the number of noises we can perfect explain by our model $f(x) = x^\top \beta$. Letting p be the VC-dimension of the class $\mathscr{F}$, one can prove that

$$N \leq \left(\frac{en}{p}\right)^p,$$

i.e., the dimension of the space $\{(f(x_1), ..., f(x_n)), \forall f \in \mathscr{F}\}$ is p.

The Rademacher complexity R is bounded by the growth number N. R is your maximal score if you take the exam N times.

$$R \leq \sqrt{\frac{2\log N}{n}}.$$

4.2.5 Symmetrization

In general $(x,y) \sim p(x,y)$, where $[y|x] \sim p(y|x)$, and y does not follow Bernoulli$(1/2)$. In the regression setting, g or $|g - \hat{g}|^2$ gets cancelled in comparing the training error and testing error, so that what matters is the noise vector ε. This is also the case with classification, due to the following symmetrization trick.

Suppose the training data is $(x_i, y_i) \sim p(x,y)$, $i = 1, ..., n$. For any f, consider

$$\frac{1}{n}\sum_{i=1}^n \varepsilon_i y_i f(x_i) = \frac{1}{n}\left[\sum_{i:\varepsilon_i=+1} y_i f(x_i) - \sum_{i:\varepsilon_i=-1} y_i f(x_i)\right],$$

which can be interpreted as cross-validation version of overfitting, i.e., randomly assign half of the training data as "training", and the other half as "testing". However, $\varepsilon_i y_i \sim$ Bernoulli$(1/2)$, thus the cross-validation overfitting has the same distribution as $\frac{1}{n}\sum_{i=1}^n \varepsilon_i f(x_i)$, which is about learning from noise.

During testing, suppose that we have $(\tilde{x}_i, \tilde{y}_i) \sim p(x,y), i = 1, ..., n$. For classifier $f(x)$, the testing score is $\frac{1}{n}\sum_{i=1}^n \tilde{y}_i f(\tilde{x}_i)$. The bound of overfitting can be calculated by

$$\begin{aligned}
\mathbb{E}\left[\frac{1}{n}\sum_{i=1}^n y_i f(x_i) - \frac{1}{n}\sum_{i=1}^n \tilde{y}_i f(\tilde{x}_i)\right] &\leq \mathbb{E}\left[\max_{f\in\mathscr{F}}\left(\frac{1}{n}\sum_{i=1}^n y_i f(x_i) - \frac{1}{n}\sum_{i=1}^n \tilde{y}_i f(\tilde{x}_i)\right)\right] \\
&= \mathbb{E}\left[\max_{f\in\mathscr{F}}\left(\frac{1}{n}\sum_{i=1}^n (y_i f(x_i) - \tilde{y}_i f(\tilde{x}_i))\,\varepsilon_i\right)\right],\ \varepsilon_i \sim \text{Bernoulli(1/2)} \\
&\leq \mathbb{E}\left[\max_{f\in\mathscr{F}} \frac{1}{n}\sum_{i=1}^n y_i \varepsilon_i f(x_i) + \max_{f\in\mathscr{F}} \frac{1}{n}\sum_{i=1}^n (-\tilde{y}_i)\varepsilon_i f(\tilde{x}_i)\right] \\
&= 2R.
\end{aligned}$$

That is, the overfitting is bounded by $2R$.

5 Kernel Regression

In kernel machines, $f(x) = h(x)^\top \beta$, and $h(x)$ is a high-dimensional or infinite dimensional vector. It is implicit. We only need to know $K(x,x') = \langle h(x), h(x')\rangle$. $K(x,x')$ is the kernel.

For the beginner, it is best to stick to the form $h(x)^\top \beta$. Gradually, one can think more in terms of reproducing kernels.

5.1 Ridge regression

In order to reduce the model bias, we want the number of parameters to be large. However, this will cause overfitting if we continue to use the least squares estimator. We may reduce overfitting by using a biased estimator such as ridge regression.

The ridge regression minimizes

$$\mathscr{L}(\beta) = |Y - X\beta|^2 + \lambda|\beta|^2,$$

for $\lambda \geq 0$. Similar to the derivation in least square estimator, we have

$$\mathscr{L}'(\beta) = -2X^\top(Y - X\beta) + 2\lambda\beta$$
$$\mathscr{L}''(\beta) = 2(X^\top X + \lambda I)$$

Setting

$$\mathscr{L}'(\beta) = -2X^\top(Y - X\beta) + 2\lambda\beta = 0,$$

we have

$$\hat{\beta} = (X^\top X + \lambda I_p)^{-1} X^\top Y.$$

5.2 Linear spline

In the one-dimensional case, the linear spline model is of the form

$$f(x) = \beta_0 + \sum_{k=1}^{d} \beta_k (x - \alpha_k)_+,$$

where $(x-\alpha_k)_+ = \max(0, x-\alpha_k)$, $\alpha_k, k = 1, ..., d$ are the knots, and β_k is the change of slope at knot α_k. We can learn the model from the training data (x_i, y_i), $i = 1, ..., n$ by ridge regression which minimizes

$$\mathscr{L}(\beta) = \sum_{i=1}^{n} \left[y_i - \beta_0 - \sum_{k=1}^{d} \beta_k (x_i - \alpha_k)_+ \right]^2 + \lambda \sum_{k=1}^{d} \beta_k^2,$$

where $\sum_{k=1}^{d} \beta_k^2$ measures the smoothness of $f(x)$.

Let $x_{ik} = (x_i - \alpha_k)_+$, $x_{i0} = 1$. The objective function is

$$\mathscr{L}(\beta) = |Y - X\beta|^2 + \beta^\top D \beta,$$

where X is the $n \times (p+1)$ matrix, D is $(p+1) \times (p+1)$ diagonal matrix, with $D_{kk} = \lambda$, except $D_{11} = 0$ because we do not penalize β_0. Then

$$\hat{\beta} = (X^\top X + D)^{-1} X^\top Y.$$

We may choose (α_k) to be equally spaced within the range of x, usually taken to be $[0, 1]$. If we use a large number of knots, and if λ is small, then the spline tends to interpolate the observations and overfit the noise. On the other hand, if λ is too big, the model may be close to a horizontal line and may fail to capture the signal.

Modern neural networks can be considered a multivariate generalization of the linear spline. Let x be a p-dimensional input. A two layer rectified neural network is of the following form

$$f(x) = \beta_0 + \sum_{k=1}^{d} \beta_k h_k,$$
$$h_k = \max\left(0, \alpha_{k0} + \sum_{j=1}^{p} \alpha_{kj} x_j\right).$$

The model can be considered a multivariate linear spline. We can learn both $\alpha = (\alpha_{kj}, j = 0,1,...,p, k = 1,...,d)$ and $\beta = (\beta_k, k = 0,1,...,d)$ from training data $\{(x_i, y_i), i = 1,...,n\}$. We will study this model later.

5.3 Representer theorem

We can replace x by $h(x)$, where $h(x) = (h_k(x), k = 1,...,d)$ and the dimension d can be infinite. Then the model becomes

$$f(x) = \sum_{k=1}^{d} \beta_k h_k(x) = h(x)^\top \beta,$$

where $h(x) = (h_k(x), k = 1,...,d)^\top$ is the hidden vector.

For a loss function

$$\sum_{i=1}^{n} L(y_i, h(x_i)^\top \beta) + \frac{1}{2}\lambda |\beta|^2.$$

The estimator that minimizes the loss function should be in the form of

$$\hat{\beta} = \sum_{i=1}^{n} c_i h(x_i).$$

That is, $\hat{\beta}$ lies in the sub-space spanned by $(h(x_i), i = 1,...,n)$. This is the representer theorem.

We can prove it by contradiction. Suppose the minimizer of the loss function $\tilde{\beta}$ does not lie in the sub-space spanned by $(h(x_i), i = 1,...,n)$. Then we can write

$$\tilde{\beta} = \hat{\beta} + \Delta,$$

where

$$\langle \Delta, h(x_i) \rangle = 0, \forall i.$$

Then

$$h(x_i)^\top \tilde{\beta} = h(x_i)^\top (\hat{\beta} + \Delta) = h(x_i)^\top \hat{\beta}.$$

But

$$|\tilde{\beta}|^2 = |\hat{\beta}|^2 + |\Delta|^2.$$

Thus $\tilde{\beta}$ has bigger loss than $\hat{\beta}$, unless $\Delta = 0$. This proves the representer theorem.

5.4 Kernel regression

For the penalized least squares loss,

$$L(\beta) = \sum_{i=1}^{n} \left(y_i - h(x_i)^\top \beta \right)^2 + \lambda |\beta|^2,$$

applying the representer theorem by plugging in $\beta = \sum_{j=1}^{n} c_j h(x_j)$, we have

$$\begin{aligned} L &= \sum_{i=1}^{n} \left(y_i - h(x_i)^\top \beta \right)^2 + \lambda |\beta|^2 \\ &= \sum_{i=1}^{n} \left(y_i - h(x_i)^\top \sum_{j=1}^{n} c_j h(x_j) \right)^2 + \lambda \left| \sum_{i=1}^{n} c_i h(x_i) \right|^2 \\ &= \sum_{i=1}^{n} \left(y_i - \sum_{j=1}^{n} c_j K(x_i, x_j) \right)^2 + \lambda \sum_{i,j} c_i c_j K(x_i, x_j) \\ &= |Y - \mathbf{K}c|^2 + \lambda c^\top \mathbf{K} c. \end{aligned}$$

Taking derivative with respect to c, and setting it to zero,

$$-\mathbf{K}(Y-\mathbf{K}c)+\lambda\mathbf{K}c=0,$$

then

$$c=(\mathbf{K}+\lambda I_n)^{-1}Y=(c_i,i=1,...,n)^\top$$

and

$$\hat{f}(x)=\sum_{i=1}^n c_iK(x_i,x).$$

5.5 Reproducing kernel Hilbert space

Suppose we have two functions defined on the same set of features, $f(x)=h(x)^\top\beta$, $g(x)=h(x)^\top\gamma$. We can define the inner product

$$\langle f,g\rangle_{\mathscr{H}}=\langle\beta,\gamma\rangle.$$

The norm then becomes

$$|f|^2_{\mathscr{H}}=\langle f,f\rangle_{\mathscr{H}}=\langle\beta,\beta\rangle=|\beta|^2,$$

and $\mathscr{H}$ consists of all the functions with finite norm.

The inner product defines a Hilbert space, which is a generalization of Euclidean space, where the notions such as orthogonality and orthogonal decomposition still hold.

The key property is the reproducing property:

$$\begin{aligned}\langle f,K(x',\cdot)\rangle_{\mathscr{H}} &= \langle h(x)^\top\beta,\, h(x)^\top h(x')\rangle_{\mathscr{H}}\\ &= h(x')^\top\beta=f(x').\end{aligned}$$

5.6 Kernel version of representer theorem

We can define the loss function as

$$\sum_{i=1}^n L(y_i,f(x_i))+\frac{\lambda}{2}|f|^2_{\mathscr{H}}.$$

This becomes non-parametric regression.

The minimizer of the loss function must be of the following form

$$\hat{f}(x)=\sum_{i=1}^n c_iK(x,x_i).$$

We proved this in the last subsection. Now we give a more direct proof. Suppose the minimizer does not lie in the subspace spanned by $(k(\cdot,x_i),i=1,...,n)$, then we can write it as

$$\tilde{f}(x)=\sum_{i=1}^n c_iK(x,x_i)+\Delta(x),$$

with

$$\langle\Delta(x),K(x_i,x)\rangle_{\mathscr{H}}=0.$$

Then on the observed x_i,

$$\begin{aligned}\tilde{f}(x_i) &= \langle\tilde{f}(x),K(x,x_i)\rangle_{\mathscr{H}}\\ &= \langle\hat{f}(x)+\Delta(x),K(x,x_i)\rangle_{\mathscr{H}}\\ &= \hat{f}(x_i)+0=\hat{f}(x_i).\end{aligned}$$

In the above, we used the reproducing property $\langle \hat{f}(x), K(x,x_i)\rangle = \hat{f}(x_i)$. The two estimator have the same values on the observed data and therefore share the same value for the loss term $\sum_{i=1}^n L(y_i, f(x_i))$. However,

$$\begin{aligned} |\tilde{f}|^2_{\mathscr{H}} &= |\hat{f}+\Delta|^2_{\mathscr{H}} \\ &= |\hat{f}|^2_{\mathscr{H}} + |\Delta|^2_{\mathscr{H}} \\ &\geq |\hat{f}|^2_{\mathscr{H}}. \end{aligned}$$

Therefore, adding any $\Delta(x)$ will increase our loss function and thus the best estimator should be $\sum_{i=1}^n c_i K(x,x_i)$.

With a reproducing kernel K and the associated norm, we can avoid dealing with $h(x)$ and β explicitly.

5.7 Mercer theorem

For a kernel $K(x,x')$, we may consider it an infinite matrix with continuous index (x,x'). If it is positive definite, we will have the spectral decomposition

$$K(x,x') = \sum_{k=1}^{\infty} \lambda_k q_k(x) q_k(x').$$

Then we can let

$$h_k(x) = q_k(x)\lambda_k^{1/2};\ h(x) = (h_k(x), k = 1, ..., \infty)^\top.$$

The point of Mercer theorem is that as long as the kernel K is positive definite, then there always exists the feature function h. But we do not need to know h explicitly.

A commonly used positive definite kernel is the Gaussian kernel or radial basis function

$$K(x,x') = \exp(-\gamma|x-x'|^2).$$

It should not be confused with the Gaussian distribution. The corresponding $h_k(x)$ are the Fourier basis, such as those in the Ptolemy epicycle model.

5.8 Nearest neighbors interpolation

The kernel model computes the score by

$$s = h(x)^\top \beta = \sum_{i=1}^n c_i K(x_i, x),$$

where $h(x)$ and β are implicit but useful for theoretical understanding, while c_i and K are explicit and are used in computing. The model is an interpolation of the observed examples. For each x, we can find nearby x_i, and then average their y_i values. The kernel model essentially does this in an optimal way. In this sense, the model does not provide much understanding of the data.

6 Gaussian Process

The Bayesian interpretation of linear regression assumes a prior distribution on the coefficient vector β. With $f(x) = h(x)^\top \beta$, $f(x)$ becomes a random function or a stochastic process over the domain of x. With a Gaussian prior distribution for β, $f(x)$ becomes a Gaussian process.

Similar to the kernel version of representer theorem, we can work with the kernel $K(x,x') = \langle h(x), h(x')\rangle$ directly, where $K(x,x')$ takes the form of covariance matrix. We do not need to deal with $h(x)$ and β explicitly, although they help us understand the model.

6.1 Background: multivariate normal

Joint distribution

We start from $Z = (z_1, ..., z_n)^\top$, where $z_i \sim \mathrm{N}(0,1)$ independently. Then $\mathbb{E}(Z) = 0$, and $\mathrm{Var}(Z) = I$. We denote $Z \sim \mathrm{N}(0, I)$. The density of Z is

$$\begin{aligned} f_Z(Z) &= \frac{1}{(2\pi)^{n/2}} \exp\left[-\frac{1}{2}\sum_i z_i^2\right] \\ &= \frac{1}{(2\pi)^{n/2}} \exp\left[-\frac{1}{2} Z^\top Z\right]. \end{aligned}$$

Let $X = \mu + \Sigma^{1/2} Z$, then $Z = \Sigma^{-1/2}(X - \mu)$, which is a matrix version of standardization. Then

$$\begin{aligned} f_Y(Y) &= \frac{1}{(2\pi)^{n/2}} \exp\left[-\frac{1}{2}(X-\mu)^\top \Sigma^{-1}(X-\mu)\right] / |\Sigma^{1/2}| \\ &= \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left[-\frac{1}{2}(X-\mu)^\top \Sigma^{-1}(X-\mu)\right]. \end{aligned}$$

Moreover, because $X = \mu + \Sigma^{1/2} Z$, we have $\mathbb{E}(X) = \mu$ and $\mathrm{Var}(X) = \Sigma$. We denote $X \sim \mathrm{N}(\mu, \Sigma)$.

In general, if $X \sim \mathrm{N}(\mu, \Sigma)$, and $Y = AX$, then $Y \sim \mathrm{N}(A\mu, A\Sigma A^\top)$. A does not need to be a square matrix.

Conditional distribution

Assuming $\mathbb{E}(X) = 0$ (otherwise we can always let $X \leftarrow X - \mathbb{E}(X)$). Let us partition X into (X_1, X_2).

$$\begin{pmatrix} X_1 \\ X_2 \end{pmatrix} \sim \mathrm{N}\left(\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{pmatrix}\right).$$

Let $\varepsilon = X_2 - AX_1$. We choose A to make $\mathrm{Cov}(\varepsilon, X_1) = 0$,

$$\begin{aligned} \mathrm{Cov}(\varepsilon, X_1) &= \mathrm{Cov}(X_2 - AX_1, X_1) \\ &= \mathrm{Cov}(X_2, X_1) - A\mathrm{Cov}(X_1, X_1) \\ &= \Sigma_{21} - A\Sigma_{11} = 0, \end{aligned}$$

so $A = \Sigma_{21}\Sigma_{11}^{-1}$, and $X_2 = AX_1 + \varepsilon$. This can be considered a regression of X_2 on X_1. The residual variance is

$$\begin{aligned} \mathrm{Var}(\varepsilon) &= \mathrm{Cov}(\varepsilon, \varepsilon) \\ &= \mathrm{Cov}(X_2 - AX_1, \varepsilon) \\ &= \mathrm{Cov}(X_2, \varepsilon) \\ &= \mathrm{Cov}(X_2, X_2 - AX_1) \\ &= \mathrm{Cov}(X_2, X_2) - \mathrm{Cov}(X_2, AX_1) \\ &= \Sigma_{22} - \Sigma_{21}\Sigma_{11}^{-1}\Sigma_{12}. \end{aligned}$$

Thus

$$\begin{pmatrix} X_1 \\ \varepsilon \end{pmatrix} = \begin{pmatrix} I_1 & 0 \\ -A & I_2 \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \end{pmatrix} \sim \mathrm{N}\left(\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} \Sigma_{11} & 0 \\ 0 & \Sigma_{22} - \Sigma_{21}\Sigma_{11}^{-1}\Sigma_{12} \end{pmatrix}\right).$$

So the marginal distribution

$$X_1 \sim \mathrm{N}(0, \Sigma_{11}),$$

the mean and the variance of the conditional distribution $P(X_2|X_1)$ are

$$\begin{aligned}\mathbb{E}(X_2|X_1) &= A\mathbb{E}(X_1)+\mathbb{E}(\varepsilon)\\ &= \Sigma_{21}\Sigma_{11}^{-1}X_1,\end{aligned}$$

$$\begin{aligned}\mathrm{Var}(X_2|X_1) &= \mathbb{E}((X_2-\mathbb{E}(X_2|X_1))^2|X_1)\\ &= \mathbb{E}(\varepsilon^2|X_1)\\ &= \mathbb{E}(\varepsilon^2)-\mathbb{E}(\varepsilon)^2\\ &= \mathrm{Var}(\varepsilon),\end{aligned}$$

thereby the conditional distribution

$$[X_2|X_1] \sim \mathrm{N}(\Sigma_{21}\Sigma_{11}^{-1}X_1, \Sigma_{22}-\Sigma_{21}\Sigma_{11}^{-1}\Sigma_{12}),$$

which is a multivariate linear regression. The geometric intuition is least squares projection.

6.2 Bayesian interpretation of ridge regression

Suppose $Y = X\beta+\varepsilon$, where $\beta \sim \mathrm{N}(0,\tau^2 I_p)$, $\varepsilon \sim \mathrm{N}(0,\sigma^2 I_n)$, and ε is independent of β. The joint distribution is

$$\begin{bmatrix}Y\\ \beta\end{bmatrix} \sim \mathrm{N}\left(\begin{bmatrix}0\\ 0\end{bmatrix}, \begin{bmatrix}\tau^2XX^\top+\sigma^2I_n & \tau^2X\\ \tau^2X^\top & \tau^2I_p\end{bmatrix}\right).$$

The posterior of β is

$$[\beta|Y,X] = \mathrm{N}(\tau^2X^\top(\tau^2XX^\top+\sigma^2I_n)^{-1}Y, \tau^2I_p-\tau^2X^\top(\tau^2XX^\top+\sigma^2I_n)^{-1}\tau^2X)$$

The posterior mean is the same as the ridge regression. In fact, we can write

$$\begin{aligned}p(\beta|Y,X) &\propto p(\beta)p(Y|X,\beta)\\ &\propto \exp\left(-\frac{1}{2\tau^2}|\beta|^2\right)\exp\left(-\frac{1}{2\sigma^2}|Y-X\beta|^2\right)\\ &= \exp\left(-\frac{1}{2}\left[\frac{1}{\sigma^2}|Y-X\beta|^2+\frac{1}{\tau^2}|\beta|^2\right]\right),\end{aligned}$$

which clearly shares the form of ridge regression.

It is tempting to apply the kernel trick by promoting x to $h(x)$, and replacing $\langle x,x'\rangle$ by $\langle h(x),h(x')\rangle = K(x,x')$. But there is a more direct way where we do not need to deal with $h(x)$ and β explicitly, and there is no need to explicitly integrating out β.

6.3 Kernel as covariance

Assuming $f(x) = h(x)^\top\beta$, and $\beta \sim \mathrm{N}(0,\tau^2 I_d)$, $f(x)$ is a stochastic process in the domain of x, and for any pair of points (x,x') in the domain,

$$\begin{aligned}\mathrm{Cov}(f(x),f(x')) &= \mathrm{Cov}(h(x)^\top\beta, h(x')^\top\beta)\\ &= \mathbb{E}[h(x)^\top\beta\beta^\top h(x')]\\ &= \tau^2h(x)^\top h(x') = K(x,x').\end{aligned}$$

Thus the kernel becomes the covariance matrix.

We may use Gaussian kernel, but it has nothing to do with the word "Gaussian" in the Gaussian process, which comes from the fact that for any $(x_1,...,x_n)$, $(f(x_1),...,f(x_n))$ follows a multivariate Gaussian distribution since β follows a Gaussian prior distribution.

6.4 Prediction by conditioning

For the training data, we can write:

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{bmatrix} = \begin{bmatrix} f(x_1) \\ f(x_2) \\ \dots \\ f(x_n) \end{bmatrix} + \varepsilon,$$

where $\varepsilon \sim \mathrm{N}(0, \sigma^2 I_n)$.

Marginally, we have

$$Y \sim \mathrm{N}(0, \mathbf{K} + \sigma^2 I_n),$$

where $\mathbf{K}_{ij} = K(x_i, x_j)$ is an $n \times n$ matrix. For a testing example x_0, we have $\mathbb{E}[f(x_0)] = 0$, $\mathrm{Var}[f(x_0)] = K(x_0, x_0)$, and $\mathrm{Cov}(Y, f(x_0)) = K(X, x_0)$, which is an $n \times 1$ vector, whose i-th element is $K(x_i, x_0)$. The joint distribution is

$$\begin{bmatrix} Y \\ f(x_0) \end{bmatrix} = \mathrm{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \mathbf{K} + \sigma^2 I_n & K(X, x_0) \\ K(x_0, X) & K(x_0, x_0) \end{bmatrix}_{(n+1)\times(n+1)} \right).$$

The prediction of $f(x_0)$ is then based on

$$[f(x_0)|Y, X] \sim \mathrm{N}(K(x_0, X)^\top (\mathbf{K} + \sigma^2 I_n)^{-1} Y, K(x_0, x_0) - K(x_0, X)(\mathbf{K} + \sigma^2 I_n)^{-1} K(x_0, X)^\top),$$

where the posterior mean

$$\hat{f}(x_0) = K(x_0, X)^\top (\mathbf{K} + \sigma^2 I_n)^{-1} Y,$$

which is the same as kernel regression.

6.5 Posterior interval

In addition to finding the point estimate as in kernel regression, we also obtain the uncertainty of the estimate with

$$V = K(x_0, x_0) - K(x_0, X)(\mathbf{K} + \sigma^2 I_n)^{-1} K(x_0, X)^\top.$$

We can construct a $(1 - \alpha)$ posterior interval

$$\hat{f}(x_0) \pm z_{\alpha/2} V^{1/2}.$$

6.6 Marginal likelihood for hyper-parameter

The marginal distribution of $Y \sim \mathrm{N}(0, \mathbf{K}_\gamma + \sigma^2 I_n)$, where $\mathbf{K} = (K_{ij} = K(x_i, x_j))$. There are several hyper-parameters we need to choose for GP. To name a few, we have τ^2 in the kernel function $K(x, x') = \tau^2 h(x)^\top h(x')$, σ^2 in $P(Y)$ and γ, which is the parameter of the Gaussian kernel $K(x, x') = \tau^2 \exp(-\gamma |x - x'|^2)$.

The marginal likelihood of these hyper-parameters is

$$L(\tau^2, \sigma^2, \gamma) = \frac{1}{(2\pi)^{n/2} |\Sigma_{\tau^2, \sigma^2, \gamma}|^{1/2}} \exp\left(-\frac{1}{2} Y^\top \Sigma^{-1}_{\tau^2, \sigma^2, \gamma} Y \right),$$

where $\Sigma_{\tau^2, \sigma^2, \gamma} = \mathbf{K}_{\tau^2, \gamma} + \sigma^2 I_n$.

The log-marginal-likelihood for determining τ^2, σ^2 and γ is then

$$l(\tau^2, \sigma^2, \gamma) = -\frac{1}{2} Y^\top \Sigma^{-1}_{\tau^2, \sigma^2, \gamma} Y - \frac{1}{2} \log |\Sigma_{\tau^2, \sigma^2, \gamma}|.$$

It is simpler than cross-validation for choosing τ^2, σ^2 and γ.

7 Kernel SVM

We now study kernel classification, which parallels the kernel regression. Again we begin with the linear classifier and apply the kernel trick, where we use the hinge function as the loss function. Then we explain the original idea of support vector machine as the max margin classifier. Both the hinge and margin formulations involve rewriting the primal problem as min-max, and then changing it to the max-min problem which is the dual problem.

7.1 Max margin

Consider the perceptron $y_i = \text{sign}(x_i^\top \beta)$, which separates the positive examples and negative examples by projecting the data on vector β, or by a hyperplane that is perpendicular to β. If the positive examples and negative examples are separable, there can be many separating hyperplanes. We want to choose the one with the maximum margin in order to guard against the random fluctuations in the unseen testing examples.

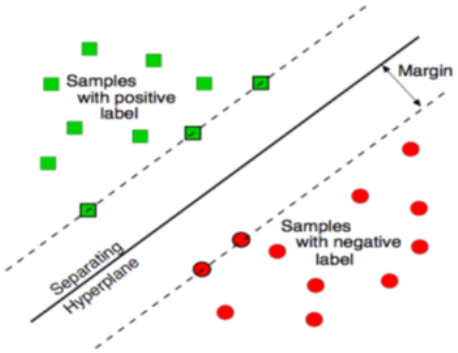


Figure 18: Max margin. We want to separate the positive examples and negative examples by a hyperplane that has the maximum margin.

The idea of support vector machines (SVM) is to find the β so that,
(1) for positive examples $y_i = +$, $x_i^\top \beta \geq 1$ and,
(2) for negative examples $y_i = -$, $x_i^\top \beta \leq -1$.
Here we use +1 and -1, because we can always scale β.

The decision boundary is decided by the training examples that lie on the margin. Those are the support vectors. Let u be a unit vector that has the same direction as β, $u = \dfrac{\beta}{|\beta|}$. Suppose x_i is an example on the margin (i.e., support vector), the projection of x_i on u is

$$\langle x_i, u \rangle = \langle x_i, \frac{\beta}{|\beta|} \rangle = \frac{\langle \beta, x_i \rangle}{|\beta|} = \frac{\pm 1}{|\beta|}.$$

So the margin is $1/|\beta|$. In order to maximize the margin, we should minimize $|\beta|$ or $|\beta|^2$. Hence, the SVM can be formulated as an optimization problem as follows:

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}|\beta|^2, \\ \text{subject to} \quad & y_i x_i^\top \beta \geq 1, \forall i. \end{aligned}$$

Recall $x_i^\top \beta$ is the score, and $y_i x_i^\top \beta$ is the individual margin of observation i. This is the primal form of SVM.

7.2 Primal dual

Primal as min-max

We first translate the constrained form of the primal into an unconstrained form with Lagrangian

$$L(\beta,\alpha)=\frac{1}{2}|\beta|^2+\sum_{i=1}^n\alpha_i\left(1-y_ix_i^\top\beta\right),$$

where $\alpha_i\geq 0$, and $\alpha=(\alpha_i,i=1,...,n)$. This rewrite results in

$$\min_\beta\max_{\alpha\geq 0}L(\beta,\alpha).$$

Note that in this unconstrained form, the constraints are automatically satisfied. Otherwise, for the optimal β, if there exists an i such that $1-y_ix_i^\top\beta>0$, then we can let $\alpha_i\to\infty$ in maximization, which will push $L(\beta,\alpha)\to\infty$. Apparently, such a β cannot be a minimizer. This contradiction proves the statement that constraints are automatically satisfied.

Dual as max-min

The dual problem is $\max_\alpha\min_\beta L(\beta,\alpha)$. This is analytically more friendly, since for fixed α and $\min_\beta L$, we have

$$\begin{aligned}L(\beta,\alpha) &= \frac{1}{2}|\beta|^2+\sum_{i=1}^n\alpha_i-\left\langle\beta,\sum_{i=1}^n\alpha_iy_ix_i\right\rangle\\ &= \frac{1}{2}\left|\beta-\sum_{i=1}^n\alpha_iy_ix_i\right|^2-\frac{1}{2}\left|\sum_{i=1}^n\alpha_iy_ix_i\right|^2+\sum_{i=1}^n\alpha_i,\end{aligned}$$

with the representer

$$\hat\beta=\sum_{i=1}^n\alpha_iy_ix_i.$$

That is, β is along the direction from the negative support vectors to the positive support vectors. Substitute $\hat\beta$ back, we have

$$\begin{aligned}q(\alpha) &= \sum_{i=1}^n\alpha_i-\frac{1}{2}\left|\sum_{i=1}^n\alpha_iy_ix_i\right|^2\\ &= \sum_{i=1}^n\alpha_i-\frac{1}{2}\sum_{i,j}\alpha_i\alpha_jy_iy_j\langle x_i,x_j\rangle.\end{aligned}$$

$q(\alpha)$ can be maximized by dual coordinate ascent.

The complementary slackness is such that

$$\alpha_i\left(1-y_ix_i^\top\beta\right)=0$$

for $i=1,...,n$. If $y_ix_i^\top\beta>1$, so that x_i is well classified, then $\alpha_i=0$. If $\alpha_i>0$, then $y_ix_i^\top\beta=1$, i.e., x_i is on the boundary and is a support vector. The learned $\beta=\sum_{i=1}^n\alpha_iy_ix_i$ is decided by the support vectors i whose $\alpha_i>0$.

7.3 Dual coordinate ascent

The dual problem can be solved by stochastic coordinate descent. At each time step t, we randomly pick an index i and compute the optimal one-variable update:

$$\delta^* = \underset{\delta: 0 \le \alpha_i + \delta \le C}{\arg\max} \; q(\alpha + \delta e_i),$$

where e_i is the one-hot vector where the i-th element is 1.

$$\begin{aligned} q(\alpha + \delta e_i) &= \frac{1}{2}(\alpha + \delta e_i)^\top Q(\alpha + \delta e_i) - (\alpha + \delta e_i)^\top \mathbf{1} \\ &= \frac{Q_{ii}}{2}\delta^2 + \alpha^\top Q_i \delta - \delta, \end{aligned}$$

where Q_i is the i-th column of Q, and Q_{ii} is the i-th diagonal element of Q. By differentiating the above with respect to δ and setting it to be zero, we have

$$\delta = \frac{1 - \alpha^\top Q_i}{Q_{ii}}.$$

However, we require $0 \le \alpha_i + \delta \le C$, so the optimal solution is

$$\delta^* = \max\left(-\alpha_i, \min\left(C - \alpha_i, \frac{1 - \alpha^\top Q_i}{Q_{ii}}\right)\right).$$

7.4 Max margin = min distance

The primal form of SVM is max margin, and the dual form of SVM is min distance.

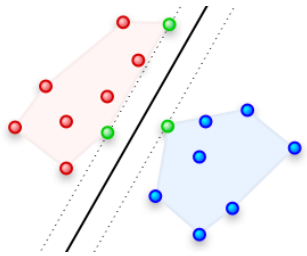


Figure 19: max margin = min distance. In order to find the max marginal separating hyperplane, we can find the minimum distance between the convex hulls of positive and negative examples. The separating hyperplane is in the middle of the minimum distance.

Consider the convex hulls of the positive and negative examples. The margin between the two sets is defined by the minimum distance between two. Let $X_+ = \sum_{i \in +} c_i x_i$ and $X_- = \sum_{i \in -} c_i x_i$ ($c_i \ge 0, \sum_{i \in +} c_i =$

$1, \sum_{i\in-} c_i = 1$) be two points in the positive and negative convex hulls. The margin is $\min |X_+ - X_-|^2$.

$$\begin{aligned} |X_+ - X_-|^2 &= \left| \sum_{i\in+} c_i x_i - \sum_{i\in-} c_i x_i \right|^2 \\ &= \left| \sum_i y_i c_i x_i \right|^2 \\ &= \sum_{i,j} c_i, c_j y_i y_j \langle x_i, x_j \rangle, \\ \text{subject to} \quad & c_i \geq 0, \sum_{i\in+} c_i = 1, \sum_{i\in-} c_i = 1. \end{aligned}$$

This problem is essentially the same as the dual problem. After we solve for c_i, the non-zeros c_i's are support vectors, i.e., examples on the boundary.

7.5 Kernel trick

We can play the kernel trick. For each x_i, we transform it to a high dimensional (or infinite dimensional) feature vector $h(x_i)$, and we use $h(x_i)$ instead of x_i for classification. Then we need to replace $\langle x_i, x_j \rangle$ by $\langle h(x_i), h(x_j) \rangle$. The kernel trick is such that we do not need to specify $h()$ explicitly. We only need to define the kernel function

$$K(x_i, x_j) = \langle h(x_i), h(x_j) \rangle.$$

One commonly used kernel function is the Gaussian kernel,

$$K(x, x') = \exp\left(-\gamma |x - x'|^2\right)$$

which measures the similarity between x and x'. With such a kernel, we can continue to minimize $q(\alpha)$. After finding the optimal α, we have

$$\beta = \sum_{i=1}^{n} \alpha_i y_i h(x_i).$$

For a testing data x, we classify it by

$$\begin{aligned} \hat{y} &= \text{sign}(h(x)^\top \beta) \\ &= \text{sign}\left(\sum_{i=1}^{n} \alpha_i y_i \langle h(x_i), h(x) \rangle \right) \\ &= \text{sign}\left(\sum_{i=1}^{n} \alpha_i y_i K(x_i, x) \right). \end{aligned}$$

In the above classifier, α_i are non-zero only for the support vectors. The classifier compares the testing example x with the support vectors, which serve as exemplars. If x is more similar to the positive exemplars, we will classify x to be positive. Otherwise we classify x to be negative.

7.6 Support vectors and nearest neighbors template matching

For those examples with $\alpha_i > 0$, they are called the support vectors, and they determine the classifier, which can be interpreted as a sophisticated version of nearest neighbor template matching. Again we only need to know the kernel function without knowing the feature function h.

7.7 Non-separable case and slack variables

The primal form of SVM for the non-separable case is

$$\begin{aligned}&\text{minmize } \frac{1}{2}|\beta|^2 + C\sum_{i=1}^{n}\xi_i,\\ &\text{subject to } y_i x_i^\top \beta \geq 1-\xi_i, \text{ and } \xi_i \geq 0,\ \forall i,\end{aligned}$$

where ξ_i are slack variables to accommodate the non-separable examples. We want the number of non-zero ξ_i to be as small as possible, i.e., we penalize $\sum_i \xi_i$.

The dual problem in this case is similar to the separable case. We only need to replace the constraint $\alpha_i \geq 0$ by the box constraint $\alpha_i \in [0,C], \forall i$. Specifically, we first translate the contrained primal to unconstrained form, with Lagrangian

$$L(\beta,\alpha,\mu) = \frac{1}{2}|\beta|^2 + C\sum_{i=1}^{n}\xi_i + \sum_{i=1}^{n}\alpha_i\left(1-\xi_i - y_i x_i^\top\beta\right) + \sum_{i=1}^{n}\mu_i(1-\xi_i),$$

where $\alpha_i \geq 0$, and $\alpha = (\alpha_i, i = 1,...,n)$, $\mu_i \geq 0$, and $\mu = (\mu_i, i = 1,...,n)$.

Then we have the primal dual transformation:

$$\min_{\beta,\xi_i}\max_{\alpha_i\geq 0,\mu_i\geq 0,\forall i} L(\beta,\alpha,\mu) \to \max_{\alpha_i\geq 0,\mu_i\geq 0,\forall i}\min_{\beta,\xi_i} L(\beta,\alpha,\mu).$$

That is, we have a dual form

$$\max_{\alpha_i\geq 0,\mu_i\geq 0,\forall i}\min_{\beta,\xi_i} L(\beta,\alpha,\mu) = \frac{1}{2}|\beta|^2 + \sum_{i=1}^{n}\alpha_i\left(1-y_i x_i^\top\beta\right) + \sum_{i=1}^{n}(C-\mu_i-\alpha_i)\xi_i.$$

It is easy to see that the term $\frac{1}{2}|\beta|^2 + \sum_{i=1}^{n}\alpha_i\left(1-y_i x_i^\top\beta\right)$ is the same as the one in the separable case. For the extra term $\sum_{i=1}^{n}(C-\mu_i-\alpha_i)\xi_i$, it has to be zero to reach the equilibrium. Thus we have

$$0 \leq \alpha_i = C - \mu_i \leq C.$$

That is, for the non-separable case, we solve the same dual form as the separable case with on extra constraint $\alpha_i \in [0,C]\forall i$.

7.8 Bias and sequential minimal optimization (SMO)

With a bias term, SVM solves the optimization problem,

$$\min_{(w,b)}\sum_{i=1}^{n}\max_{\alpha_i\in[0,C]}\alpha_i(1-y_i(w^\top x_i+b)) + \frac{1}{2}|w|^2,$$

which is equivalent to

$$\max_{\alpha_i\in[0,C]}\min_{(w,b)}\frac{1}{2}\left|w-\sum_{i=1}^{n}y_i\alpha_i x_i\right|^2 + \sum_{i=1}^{n}\alpha_i + \sum_{i=1}^{n}y_i\alpha_i b - \frac{1}{2}\left|\sum_{i=1}^{n}\alpha_i y_i x_i\right|^2.$$

The constraint $\sum_{i=1}^{n}y_i\alpha_i = 0$ must be satisfied. Otherwise if $\sum_{i=1}^{n}y_i\alpha_i > 0$, then we can let $b \to -\infty$, and if $\sum_{i=1}^{n}y_i\alpha_i < 0$, then $b \to \infty$, so that we will not have minimum. Thus our task is reduced to solving the dual problem,

$$\max_{\alpha_i\in[0,C]}\sum_{i=1}^{n}\alpha_i - \frac{1}{2}\alpha^\top Q\alpha,$$

subject to $\sum_{i=1}^{n}y_i\alpha_i = 0$. Instead of coordinate ascent, we need to update α_i in pairs to make sure the constraint is satisfied. This is usually implemented by sequential minimal optimization (SMO).

7.9 Primal form with hinge loss

In order to make ξ_i as small as possible, if $y_i x_i^\top \beta \geq 1$, we can take $\xi_i = 0$. Otherwise, we can let $\xi_i = 1 - y_i x_i^\top \beta$. Combine the two conditions, the optimal ξ_i is

$$\xi_i = \max(0, 1 - y_i x_i^\top \beta).$$

Then the primal problem becomes an unconstrained minimization of the loss function

$$\frac{1}{2}|\beta|^2 + C\max(0, 1 - y_i x_i^\top \beta),$$

or let $C = 1/\lambda$, the minimization becomes

$$\max(0, 1 - y_i x_i^\top \beta) + \frac{\lambda}{2}|\beta|^2.$$

This is similar to ridge regression with ℓ_2 regularization. In general, we can use loss function

$$\sum_{i=1}^{n} L(y_i x_i^\top \beta) + \frac{\lambda}{2}|\beta|^2,$$

where $L()$ can be hinge loss or logistic loss.

7.10 Hinge loss and linear SVM

For classification where $y_i \in \{+1, -1\}$, we can classify y_i by the perceptron model:

$$\hat{y}_i = \text{sign}(x_i^\top \beta).$$

To estimate β, we can modify the loss function of ridge regression $\mathscr{L}(\beta) = |Y - X\beta|^2 + \lambda|\beta|^2$ to

$$\mathscr{L}(\beta) = \sum_{i=1}^{n} \max(0, 1 - y_i x_i^\top \beta) + \frac{\lambda}{2}|\beta|^2.$$

This is the objective function of linear SVM.

7.11 Re-representing hinge loss

For the hinge loss $\max(0, 1 - m)$, it is non-linear in m, we can represent it as

$$\max(0, 1 - m) = \max_{\alpha \in [0,1]} \alpha(1 - m),$$

which is linear in m.

7.12 Primal min-max to dual max-min

The optimization problem becomes

$$\hat{\beta} \leftarrow \min_{\beta} \sum_{i=1}^{n} \max_{\alpha_i \in [0,1]} \alpha_i (1 - y_i x_i^\top \beta) + \frac{\lambda}{2}|\beta|^2.$$

We can change min max to max min, so that we change the primal problem into the dual problem.

$$\hat{\beta} \leftarrow \max_{\alpha_i \in [0,1] \forall i} \min_{\beta} \sum_{i=1}^{n} \alpha_i (1 - y_i x_i^\top \beta) + \frac{1}{2}|\beta|^2$$

For convenience, let $C = 1/\lambda$:

$$\hat{\beta} \leftarrow \max_{\alpha_i \in [0,1] \forall i} \min_{\beta} \sum_{i=1}^{n} \alpha_i (1 - y_i x_i^\top \beta) + \frac{1}{2C}|\beta|^2$$

which is equivalent to

$$\hat{\beta} \leftarrow \max_{\alpha_i \in [0,1] \forall i} \min_{\beta} C \left[\sum_{i=1}^{n} \alpha_i (1 - y_i x_i^\top \beta) + \frac{1}{2C}|\beta|^2 \right]$$

Let $\alpha_i \leftarrow C\alpha_i$, and now we obtain

$$\begin{aligned}
\hat{\beta} &\leftarrow \max_{\alpha_i \in [0,C] \forall i} \min_{\beta} \sum_{i=1}^{n} \alpha_i (1 - y_i x_i^\top \beta) + \frac{1}{2}|\beta|^2 \\
&= \max_{\alpha_i \in [0,C] \forall i} \min_{\beta} \left[\frac{1}{2} \left| \beta - \sum_{i=1}^{n} \alpha_i y_i x_i \right|^2 - \frac{1}{2} \left| \sum_{i=1}^{n} \alpha_i y_i x_i \right|^2 + \sum_{i=1}^{n} \alpha_i \right].
\end{aligned}$$

The minimization gives us the representer

$$\hat{\beta} = \sum_{i=1}^{n} \alpha_i y_i x_i.$$

The dual problem is $\max_{\alpha_i \in [0,C], \forall i} q(\alpha)$,

$$q(\alpha) = \left[\sum_{i=1}^{n} \alpha_i - \frac{1}{2} \left| \sum_{i=1}^{n} \alpha_i y_i x_i \right|^2 \right] = \frac{1}{2} \alpha^\top Q \alpha - \alpha^\top \mathbf{1},$$

where $\mathbf{1}$ is a vector of 1's.

7.13 Kernelize

For kernel SVM, we a assume feature vector $h(x)$, and classify y_i by

$$\hat{y}_i = \text{sign}(h(x_i)^\top \beta).$$

The loss function is

$$\mathscr{L}(\beta) = \sum_{i=1}^{n} \max(0, 1 - y_i h(x_i)^\top \beta) + \frac{\lambda}{2}|\beta|^2.$$

The representer is

$$\hat{\beta} = \sum_{i=1}^{n} \alpha_i y_i h(x_i).$$

The dual problem is $\max_{\alpha_i \in [0,C], \forall i} q(\alpha)$,

$$\begin{aligned} q(\alpha) &= \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j \langle h(x_i), h(x_j) \rangle \\ &= \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j K(x_i, x_j). \end{aligned}$$

Again the problem can be solved by dual coordinate ascent.

After solving α_i, we get $\hat{\beta}$ from the representer, and the estimated function is

$$\begin{aligned} \hat{f}(x) &= h(x)^\top \hat{\beta} \\ &= \sum_{i=1}^{n} \alpha_i y_i \langle h(x_i), h(x) \rangle \\ &= \sum_{i=1}^{n} \alpha_i y_i K(x_i, x). \end{aligned}$$

8 Lasso Regression

8.1 ℓ_1 regularization

The Lasso regression estimates β by

$$\hat{\beta}_\lambda = \arg\min_\beta \left[\frac{1}{2}\|Y - X\beta\|_{\ell_2}^2 + \lambda\|\beta\|_{\ell_1} \right],$$

where $\|\beta\|_{\ell_1} = \sum_{j=1}^p |\beta_j|$. Lasso stands for "least absolute shrinkage and selection operator." There is no closed form solution for general p.

We do have closed form solution for $p = 1$, where X is an $n \times 1$ vector,

$$\hat{\beta}_\lambda = \begin{cases} (\langle Y, X\rangle - \lambda)/\|X\|_{\ell_2}^2, & \text{if } \langle Y, X\rangle > \lambda; \\ (\langle Y, X\rangle + \lambda)/\|X\|_{\ell_2}^2, & \text{if } \langle Y, X\rangle < -\lambda; \\ 0 & \text{if } |\langle Y, X\rangle| < \lambda. \end{cases}$$

We can write it as

$$\hat{\beta}_\lambda = \text{sign}(\hat{\beta}) \max(0, |\hat{\beta}| - \lambda/\|X\|_{\ell_2}^2),$$

where $\hat{\beta} = \langle Y, X\rangle/\|X\|_{\ell_2}^2$ is the least squares estimator. The above transformation from $\hat{\beta}$ to $\hat{\beta}_\lambda$ is called soft thresholding.

Compare Lasso with ridge regression in one-dimensional situation, the latter being $\hat{\beta}_\lambda = \langle Y, X\rangle/(\|X\|_{\ell_2}^2 + \lambda)$, the behavior of Lasso is richer, including both shrinkage (by subtracting λ) and selection (via thresholding at λ).

The reason for the fact that $\hat{\beta}$ can be zero is that the left and right derivatives of $|\beta|$ at 0 are not the same, so that the function $\|Y - X\beta\|_{\ell_2}^2/2 + \lambda\|\beta\|_{\ell_1}$ may have a negative left derivative and a positive right derivative at 0, so that 0 can be the minimum. For $|\beta|^{1+\delta}$ with $\delta > 0$, its derivative at 0 is 0, so that $\hat{\beta}$ cannot be zero in general. For $|\beta|^{1-\delta}$, there is a sharp turn at 0, but it is not convex anymore. $|\beta|$ or piecewise linear function in general is the only choice that has a sharp turn at 0 but is still barely convex.

Thus the Lasso regression prefers sparse β, i.e., only a small number of components of β are non-zero.

8.2 Primal form of Lasso

The primal form of Lasso is $\min \|Y - X\beta\|_{\ell_2}^2/2$ subject to $\|\beta\|_{\ell_1} \le t$. The lagrangian is

$$\begin{aligned} L(\beta, \lambda) &= \frac{1}{2}\|Y - X\beta\|_{\ell_2}^2 + \lambda(\|\beta\|_{\ell_1} - t) \\ &= \frac{1}{2}\|Y - X\beta\|_{\ell_2}^2 + \lambda\|\beta\|_{\ell_1} - \lambda t \end{aligned}$$

Minimizing $\frac{1}{2}\|Y - X\beta\|_{\ell_2}^2 + \lambda\|\beta\|_{\ell_1} - \lambda t$ over β is equivalent to minimizing $\frac{1}{2}\|Y - X\beta\|_{\ell_2}^2 + \lambda\|\beta\|_{\ell_1}$

Therefore, the dual form of Lasso is $\min \|Y - X\beta\|_{\ell_2}^2/2 + \lambda\|\beta\|_{\ell_1}$. The two forms are equivalent with a one-to-one correspondence between t and λ. If $\hat{\beta}_\lambda$ is the solution to the dual form, then it must be the solution to the primal form with $t = \|\hat{\beta}_\lambda\|_{\ell_1}$. The reason is that if a different $\hat{\beta}$ is the solution to the primal form, then $\hat{\beta}$ is a better solution to the dual form than $\hat{\beta}_\lambda$, which results in contradiction.

The primal form also reveals the sparsity inducing property of ℓ_1 regularization in that the ℓ_1 ball has low-dimensional corners, edges, and faces, but is still barely convex.

The above is the well known figure of Lasso. Take the left plot for example. The blue region is $\|\beta\|_{\ell_1} \le t$. The red curves is the contour plot, where each red elliptical circle consists of those β that have the same

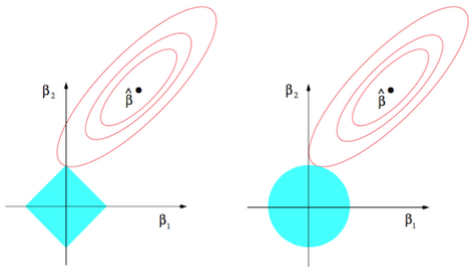


Figure 20: Lasso in primal form.

value of $\|Y - X\beta\|_{\ell_2}^2$. The circle on the outside has bigger $\|Y - X\beta\|_{\ell_2}^2$ than the circle inside. The solution to the problem of $\min \|Y - X\beta\|_{\ell_2}^2$ subject to $\|\beta\|_{\ell_1} \leq t$ is where the red circle touches the blue region. Any other points in the blue region will be outside the outer red circle and thus have bigger values of $\|Y - X\beta\|_{\ell_2}^2$. The reason that the ℓ_1 regularization induces sparsity is that it is likely for the red circle to touch the blue region at a corner, which is a sparse solution. If we use ℓ_2 regularization, as is the case with the plot on the right, then the solution is not sparse in general.

8.3 Coordinate descent for Lasso solution path

Recall that our Lasso dual form is

$$\min_{\beta} L(\beta) = \min_{\beta} \frac{1}{2}\|Y - X\beta\|_{\ell_2}^2 + \lambda\|\beta\|_{\ell_1}$$

For multi-dimensional $X = (X_j, j = 1, ..., p)$, we can use the coordinate descent algorithm to compute $\hat{\beta}_\lambda$.The algorithm updates one component at a time.

Given the current values of $\beta = (\beta_j, j = 1, ..., p)$, for $j = 1, ..., p$, we treat the loss function $L(\beta)$ as $L(\beta_j)$, where we fix β_k where $k \neq j$. Let $R = Y - \sum_{j=1}^{p} X_j\beta_j$ and $R_j = Y - \sum_{k\neq j} X_k\beta_k$.

We then update β_j such that $\frac{\partial L}{\partial \beta_j} = 0$, from which we can derive $\beta_j = \text{sign}(\hat{\beta}_j)\max(0, |\hat{\beta}_j| - \lambda/\|X\|_{\ell_2}^2)$, where $\hat{\beta}_j = \langle R_j, X_j\rangle / \|X_j\|_{\ell_2}^2$ is the least square estimator obtained from coordinate descent.

We can find the solution path of Lasso by starting from a big λ so that all of the estimated β_j are zeros. Then we gradually reduce λ. For each λ, we cycle through $j = 1, ..., p$ for coordinate descent until convergence, and then we lower λ. This gives us $\hat{\beta}(\lambda)$ for the whole range of λ. The whole process is a forward selection process, which sequentially selects new variables and occasionally removes selected variables.

8.4 Least angle regression

Let $R = Y - \sum_{j=1}^{p} X_j\beta_j$ and $R_j = Y - \sum_{k\neq j} X_k\beta_k$. Then, $R = R_j - X_j\beta_j$. In the above coordinate descent algorithm, at any given λ and each β_j, we want to minimize $L(\beta_j)$. If β is the Lasso solution, we have

$$\frac{\partial L}{\partial \beta_j} = -R^\top X_j + \lambda s = 0 \quad \text{where} \quad s = \begin{cases} 1, & \text{if } \beta_j > 0, \\ -1, & \text{if } \beta_j < 0, \\ (-1, 1) & \text{if } \beta_j = 0. \end{cases}$$

then

$$\langle R, X_j\rangle = \begin{cases} \lambda, & \text{if } \beta_j > 0, \\ -\lambda, & \text{if } \beta_j < 0, \\ s\lambda & \text{if } \beta_j = 0. \end{cases}$$

where $|s| < 1$. Thus in the above process, for all of those selected X_j, the algorithm maintains that $\langle R, X_j \rangle$ to be λ or $-\lambda$, for all selected X_j. If we interpret $|\langle R, X_j \rangle|$ in terms of the angle between R and X_j, then we may call the above process the equal angle regression or the least angle regression (LARS). In fact, the solution path is piecewise linear, and the LARS computes the linear pieces analytically instead of gradually reducing λ as in coordinate descent.

8.5 Stagewise regression or epsilon-boosting

The stagewise regression iterates the following steps. Given the current $R = Y - \sum_{j=1}^{p} X_j \beta_j$, find j with the maximal $|\langle R, X_j \rangle|$. Then update $\beta_j \leftarrow \beta_j + \varepsilon \langle R, X_j \rangle$ for a small ε. This is similar to the matching pursuit but is much less greedy. Such an update will change R and reduce $|\langle R, X_j \rangle|$, until another X_j catches up. So overall, the algorithm ensures that all of the selected X_j to have the same $|\langle R, X_j \rangle|$, which is the case with the algorithm in the above two sections. The stagewise regression is also called ε-boosting.

We can also view the stagewise regression from the perspective of the primal form of the Lasso problem: minimize $\|Y - X\beta\|_{\ell_2}^2$ subject to $\|\beta\|_{\ell_1} \leq t$. If we relax the constraint by increasing t to $t + \Delta t$, then we want to update β_j with the maximal $|\langle R, X_j \rangle|$ in order to maximally reducing $\|Y - X\beta\|_{\ell_2}^2$.

9 Boosting

In trees and boosting machines, $f(x) = h(x)^\top \beta = \sum_{m=1}^{d} h_m(x) \beta_m$, where $h_m(x)$ are base functions or base learners. In trees, $h_m(x)$ are indicator functions of rectangle regions. In boosting machines, $h_m(x)$ are themselves shallow trees. We learn $h_m(x)$ by greedy search or forward selection.

9.1 Classification and regression trees (CART)

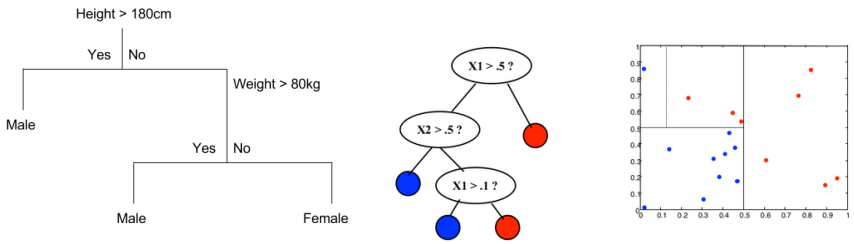


Figure 21: Decision tree. Left: an example of a decision tree that classifies a person as male or female based on height and weight. Right: the process of learning CART is to recursively partitioning each rectangle region into two sub-regions. Each partitioning should increases the purity of the y_i within each region.

Figure 21 illustrates the basic idea of classification and regression tree (CART). The figure on the left shows a simple example of a decision tree, to decide whether a person is male or female based on the height and weight of the person. The classification and regression tree is to learn such a decision rule from the data. The figure on the right illustrates the process of learning, by recursively partitioning the data space. Specifically, for the whole data set, we go through all the variables, $j = 1, ..., p$, and for each variable, we go though all the observed values $(x_{ij}, i = 1, ..., n)$. For each $t = x_{ij}$, we consider dividing the data space into two parts based on whether $x_j \geq t$ or $x_j < t$. We choose the best j and t according to a certain criterion. Thus we split the whole dataset into two subsets. For each subset of data, we repeat what we did for the whole

dataset. In this way, we obtain a recursive binary partition of the data space, which corresponds to a binary decision tree. We can design a criterion to determine when to stop this process.

First, what is the criterion to choose j and t for binary division? We want to divide the whole region into two regions, so that examples within the same region tend to share the same responses. In other words, the examples within each region display strong purity.

Formally, the process of recursive partitioning eventually divides the whole region into M sub-regions $R_m, m = 1, ..., M$. For a generic example (x, y), we model

$$f(x) = \sum_{m=1}^{M} c_m \mathbf{1}(x \in R_m),$$

where $\mathbf{1}()$ is an indicator function, which is 1 if $x \in R_m$, and 0 otherwise. This is a piecewise constant function. Suppose y_i is continuous, and we want to predict y by $f(x)$. We estimate $f(x)$ by minimizing the least squares loss function

$$\mathscr{L} = \sum_{i=1}^{n} (y_i - f(x_i))^2.$$

Consider the first step that partitions the whole region into two sub-regions R_1 and R_2. Let $\hat{c}_1$ be the average of y_i with $x_i \in R_1$, and let $\hat{c}_2$ be the average of y_i with $x_i \in R_2$. Then the loss function is

$$\mathscr{L} = \sum_{x_i \in R_1} (y_i - \hat{c}_1)^2 + \sum_{x_i \in R_2} (y_i - \hat{c}_2)^2,$$

which may be considered a measure of purity in the case of regression. We want to find j and t to minimize the above $\mathscr{L}$. We can repeat this procedure recursively.

To be more concrete, suppose we would like to choose the m-th region $R_m = \{x | x_j \leq t\}$ with cutoff position x_j and cutoff value $t \in [\min_i x_{ij}, \max_i x_{ij}]$, we opt to find j and t to minimize the risk below

$$x_j, t = \arg\min_{x_j, t} \sum_{i=1}^{n} \left(r_i - c_m 1\left(x \in R_m\right)\right)^2,$$

where

$$\begin{aligned} r_i &= y_i - \sum_{m'=1}^{m-1} c_{m'} 1\left(x \in R_{m'}\right), \\ c_m &= \frac{\sum_{i=1}^{n} r_i 1\left(x_i \in R_m\right)}{\sum_{i=1}^{n} 1\left(x_i \in R_m\right)}, \end{aligned}$$

are the estimation (mean as the best estimator) of c_m and the i-th residual of the previous model with $m-1$ regions respectively.

As to when to stop, we may consider penalizing the complexity of the tree, e.g., we use the following regularized loss function

$$\mathscr{L} = \sum_{i=1}^{n} (y_i - f(x_i))^2 + \lambda M,$$

where $\lambda > 0$ is the cost for adding one region. Then we can stop the partitioning process if the reduction in the loss cannot cover the cost of adding a region.

We may also include some regularization on the partitions. We can further introduce weights for each datapoint in the training set. Then the objective becomes

$$\mathscr{L} = \sum_{i=1}^{n} w_i \left(y_i - f\left(x_i\right)\right)^2 + \gamma \sum_{m=1}^{M} |c_m|^2 + \lambda M,$$

and the criterion for selecting partitioning parameters j and t also changes accordingly

$$x_j, t = \arg\min_{x_j,t} \sum_{i=1}^{n} w_i \left(r_i - c_m 1\left(x \in R_m\right)\right)^2 + \gamma |c_m|^2 + \lambda M,$$

where r_i is identical to the case without data weights. The estimation to c_m can still be obtained by taking derivative of $\mathscr{L}$, which turns out to be

$$c_m = \frac{\sum_{i=1}^{n} w_i r_i 1\left(x_i \in R_m\right)}{\gamma + \sum_{i=1}^{n} w_i 1\left(x_i \in R_m\right)}.$$

The stop condition is still based on whether $\mathscr{L}$ decreases or not.

Finally, we consider the binary classification problem. We can simply replace the least square loss in the regression problem with logistic regression loss, then we can directly apply such a tree growing method to it.

For a more generic classification setting, suppose there are K categories, ideally, we hope that within each region, all the examples belong to the same category. More realistically, let $\hat{p}_k$ be the proportion of these examples that belong to category k. Let $\hat{k}$ be the category that has the most examples, i.e., $\hat{k} = \arg\max_k \hat{p}_k$. We may define the purity in terms of $\hat{p}_{\hat{k}} = \max_k \hat{p}_k$. If we stop splitting this region, we will classify the examples in this region to category $\hat{k}$. There are other definitions of purity, such as Gini index, which is

$$G = \sum_k \hat{p}_k (1 - \hat{p}_k).$$

9.2 Random forest

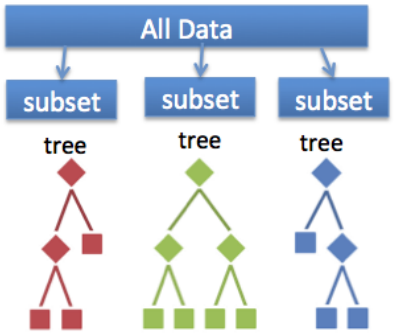


Figure 22: Random forest. We creates new datasets by resampling the original training dataset with replacement. For each new dataset, we learn a tree.

The idea of random forest is simpler than Adaboost. For the dataset $\{(x_i, y_i), i = 1, ..., n\}$, we create a new dataset by randomly sampling the examples with replacement. For this new dataset, we grow a tree. See Figure 22. For each split, instead of going over all the p variables j, we randomly sample $\sqrt{p}$ variables, and let j run through this subset of variables to decide on which variable we split. We then repeat this procedure a large number of times to grow a large number of trees, i.e., a forest. In the testing stage, for each input x, each tree produces a predicted y. We then take the average of the predicted values in the case of regression, or take the majority vote by the trees in the case of classification. See Figure 23.

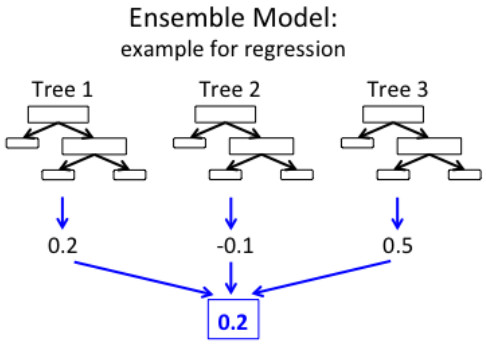


Figure 23: Random forest. The prediction by a random forest is an average of the predictions by the trees in the forest in the case of regression, and is the majority vote in the case of classification.

9.3 Adaboost

Committee of weak classifiers

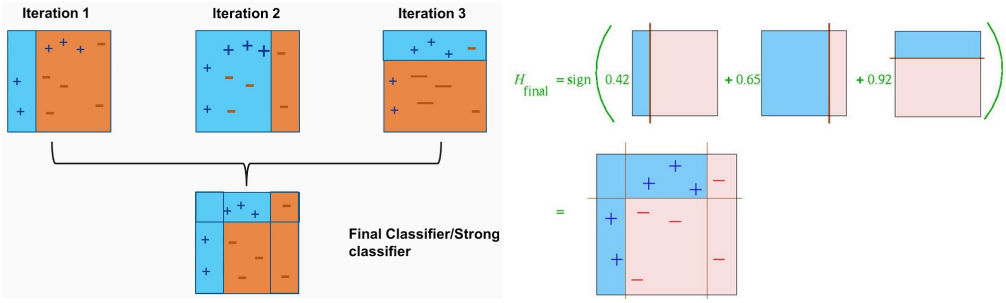


Figure 24: Adaboost. Each iteration adds a new classifier to the committee, and the final committee is a linear combination of the selected classifiers. Each new classifier is trained on reweighed dataset, where those examples that are not classified correctly by the current committee receive bigger weights, which are illustrated by bigger sizes of the + and - signs. The voting weight or coefficient of each added classifier is based on its performance on the reweighed dataset.

Adaboost is a committee machine or ensemble machine for classification, which consists of a number of weak classifiers $h_k(x_i) \in \{+,-\}$, $k = 1,...,d$. The final classification is a perceptron based on the weak classifiers,

$$y_i = \text{sign}\left(\sum_{k=1}^{d} \beta_k h_k(x_i)\right),$$

where β_k can be interpreted as the weight of vote of classifier k.

You may compare the above committee machine to the two-layer neural net we studied before, where $h_k(x_i)$ plays the role of h_{ik}.

Adding one classifier at a time

In Adaboost, the based learners are weak classifiers. As illustrated by Figure 24, we want to learn a committee to separate the positive examples and negative examples, where the weak classifiers are based on

thresholding either x_1 or x_2. Each iteration adds a new weak classifier, and then assigns bigger weights to the examples that are not classified well by the current committee. Then in the next iteration, we learn a new classifier on the reweighed dataset that focuses on mistakes. Whenever a new classifier is added, it is assigned a voting weight in the committee. The final classifier is a weighted sum of the selected weak classifiers. The Adaboost algorithm is another example of learning from errors.

Even if the classifiers $\{h_k\}$ may be weak, it is still possible to boost them into a strong classifier.

When training an Adaboost classifier, we sequentially add members to the committee.

Suppose the current committee has $m-1$ classifiers,

$$F_{m-1}(x_i) = \sum_{k=1}^{m-1} \beta_k h_k(x_i),$$

and we want to add a new member $h_m()$ to boost the current classifier to

$$F_m(x_i) = F_{m-1}(x_i) + \beta_m h_m(x_i).$$

We need to select the new classifier $h_m()$ and assign its weight β_m.

Exponential loss

The exponential loss is an upper bound of the training error, because

$$1(y_i \neq \text{sign}(f_i)) \leq \exp(-y_i f(x_i)).$$

We use this loss to guide the selection of the next weak classifier,

$$\begin{aligned} \mathscr{L}(h_m, \beta_m) &= \sum_{i=1}^{n} \exp\left[-y_i \left(F_{m-1}(x_i) + \beta_m h_m(x_i)\right)\right] \\ &\propto \sum_{i=1}^{n} D_i \exp\left[-\beta_m y_i h_m(x_i)\right], \end{aligned}$$

where

$$D_i \propto \exp\left[-\left(y_i F_{m-1}(x_i)\right)\right],$$

and we usually normalize D_i so that $\sum_{i=1}^{n} D_i = 1$. $\{D_i\}$ is a distribution over the training examples $\{(x_i, y_i\}$. If (x_i, y_i) is well classified by the current $F_{m-1}()$, then $y_i F_{m-1}(x_i)$ is large, and D_i is small. Thus the distribution $\{D_i\}$ focuses on those examples that are not classified well, i.e., it focuses on the errors or mistakes, so that the training algorithm can learn from them.

$y_i h_m(x_i)$ can only be $+1$ or -1. If $y_i h_m(x_i) = 1$, then $h_m(x_i) = y_i$, meaning that $h_m()$ classifies x_i correctly. Otherwise if $y_i h_m(x_i) = -1$, then $h_m()$ makes an error on x_i. Therefore

$$\mathscr{L}(h_m, \beta_m) = \sum_{i:h_m(x_i)=y_i} D_i e^{-\beta} + \sum_{i:h_m(x_i)\neq y_i} D_i e^{\beta} = (1-\varepsilon)e^{-\beta} + \varepsilon e^{\beta},$$

where $\varepsilon = \sum_{i:h_m(x_i)\neq y_i} D_i$ is the error of $h_m()$ on the reweighed or refocused dataset.

Recall that for positive a and b, $a + b \geq 2\sqrt{ab}$, because $a + b - \sqrt{ab} = (\sqrt{a} - \sqrt{b})^2 \geq 0$, with equality achieved when $a = b$, thus for a fixed $h_m()$,

$$(1-\varepsilon)e^{-\beta} + \varepsilon e^{\beta} \geq 2\varepsilon(1-\varepsilon),$$

with the equality achieved by

$$(1-\varepsilon)e^{-\beta} = \varepsilon e^{\beta},$$

so that

$$\beta = \frac{1}{2}\log\frac{1-\varepsilon}{\varepsilon}.$$

Therefore we want to choose $h_m()$ to minimize $\varepsilon(1-\varepsilon)$. Since $\varepsilon \leq 1/2$ (otherwise we can always flip $h_m()$), $\varepsilon(1-\varepsilon)$ is an increasing function over $\varepsilon \in [0,1/2]$. Thus we only need to choose $h_m()$ to minimize the training error ε.

9.4 Gradient boosting

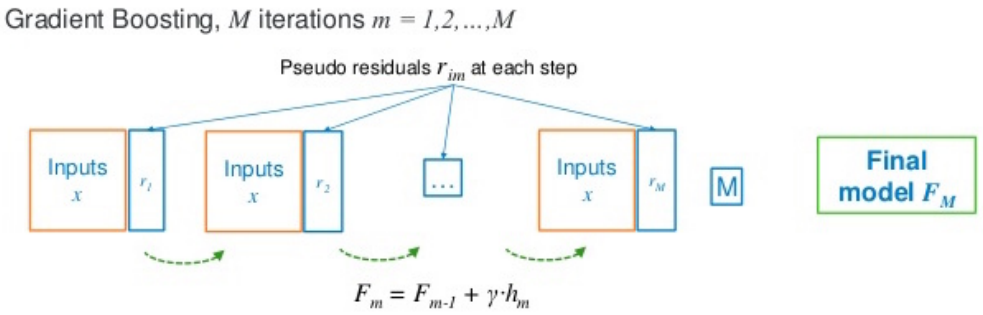


Figure 25: Gradient boosting. Each iteration learns a tree from the errors of the current model. The final model is a linear combination of the learned trees.

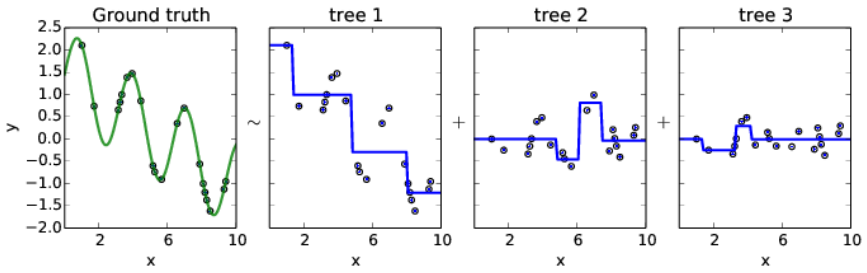


Figure 26: Gradient boosting for regression in one dimensional situation. Each tree is a piecewise constant function.

We want to learn a function

$$f(x) = \sum_{k=1}^{d} h_k(x)$$

for the purpose of regression or classification, where $h_k(x)$ are base functions that are not necessarily classifiers, i.e., $h_k(x)$ may return continuous values.

Let $F_{m-1}(x) = \sum_{k=1}^{m-1} h_k(x)$ be the current function. We want to add a base function $h_m(x)$ so that

$$F_m(x) = F_{m-1}(x) + h_m(x).$$

For regression, the least squares loss is

$$\mathscr{L} = \sum_{i=1}^{n} [y_i - (F_{m-1}(x_i) + h_m(x_i))]^2 = \sum_{i=1}^{n} (r_i - h_m(x_i))^2,$$

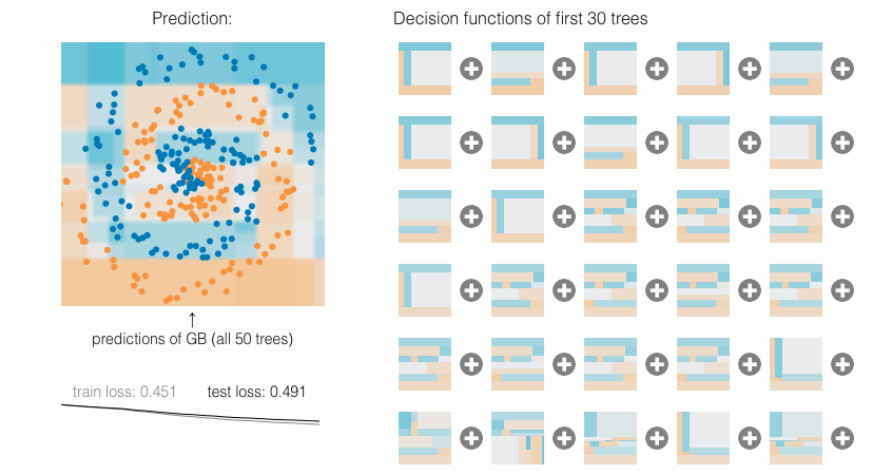


Figure 27: Gradient boosting for classification in two dimensional situation. Each tree is a piecewise constant function.

where

$$r_i = y_i - F_{m-1}(x_i)$$

is the residual error for observation i. We can then treat $\{(x_i, r_i), i = 1, ..., n\}$ as our new dataset, and learn a regression tree $h_m(x)$ from this new dataset.

For a general loss function, e.g., the mean absolution deviation,

$$\mathscr{L} = \sum_{i=1}^{n} |y_i - (F_{m-1}(x_i) + h_m(x_i))|,$$

which penalizes large deviations less than the least squares loss and is thus more robust to outliers, we may learn a new tree $h_m(x)$ by minimizing $|r_i - h_m(x_i)|$. However we need to rewrite the code for learning the tree. It is better to continue to use the code for least squares regression tree.

Let

$$s_i = F_{m-1}(x_i) + h_m(x_i),$$

for $i = 1, ..., n$, and let $\hat{s}_i = F_{m-1}(x_i)$ be the prediction by F_{m-1}. Let the loss function be $\mathscr{L} = \sum_{i=1}^{n} L(y_i, s_i)$. In the case of mean absolute deviation, $\mathscr{L}(y_i, s_i) = |y_i - s_i|$. Let

$$r_i = -\frac{\partial L(y_i, s_i)}{\partial s_i}\Big|_{\hat{s}_i},$$

Then $r = (r_i, i = 1, ..., n)^\top$ is the direction to change $f = (f_i, i = 1, ...,)$ for the steepest descent of $\mathscr{L}$. r_i becomes the residual error in the least squares regression. In general, r_i means the lack of fitness for example i because if r_i is close to 0, there is no need to change s_i to fit the data. Otherwise we need to make big change in f_i to fit the data.

We can fit a tree to approximate r by minimizing

$$\sum_{i=1}^{n} (r_i - h_m(x_i))^2$$

using the code of least squares regression tree.

9.5 Extreme gradient boosting (XGB) as iterated reweighed least squares

A more principled implementation of gradient boosting is to expand $L(y_i, s_i)$ by the second order Taylor expansion,

$$L(y_i, s_i) \approx L(y_i, \hat{s}_i) + L'(y_i, \hat{s}_i) h_m(x_i) + \frac{1}{2} L''(y_i, \hat{s}_i) h_m(x_i)^2,$$

where $L'(y,s) = \frac{\partial}{\partial s} L(y,s)$, and $L''(y,s) = \frac{\partial^2}{\partial s^2} L(y,s)$. Let $r_i = L'(y_i, \hat{s}_i)$ and $w_i = L''(y_i, \hat{s}_i)$, then we can write

$$\begin{aligned} L(y_i, s_i) &= \frac{w_i}{2} \left[h_m(x_i)^2 - 2 \frac{r_i}{w_i} h_m(x_i) \right] + \text{const} \\ &= \frac{w_i}{2} \left[\frac{r_i}{w_i} - h_m(x_i) \right]^2 + \text{const} \\ &= \frac{w_i}{2} \left[\tilde{y}_i - h_m(x_i) \right]^2 + \text{const}. \end{aligned}$$

Thus we can learn h_m by weighted least squares, with working response $\tilde{y}_i = r_i / w_i$, and working weight w_i. Compared to the original gradient boosting, we take into account the curvature w_i. The bigger the curvature it, the more important the observation is. We divide r_i by w_i because if the curvature is big, then a small step step will cause big decrease.This is similar to the Newton-Raphson algorithm for fitting generalized linear models such as logistic regression.

10 Neural Networks

In neural networks, we still have $f(x) = h(x)^\top \beta$, but $h(x)$ is not designed in terms of kernels and trees, instead $h(x)$ is itself expressed as linear models of features at lower layers. Thus a neural network is a linear model on top of linear models, with coordinate-wise non-linear transformation such as the sigmoid in logistic regression. If the non-linear transformation is the so-called rectified linear unit, $f(x)$ is a piecewise linear function, where the pieces are recursively partitioned, similar to trees.

10.1 Two layer perceptron

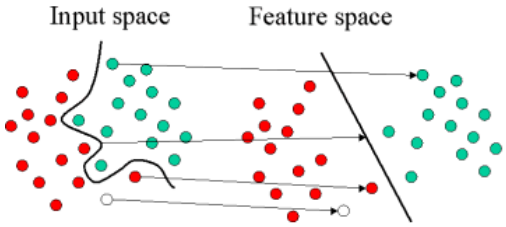


Figure 28: The positive examples and negative examples cannot be separated by a hyperplane in the original space. We can transform each x_i into a feature h_i, so that the examples can be separated by a hyperplane in the feature space.

obs	input	hidden	output
1	$X_1^\top$	$h_1^\top$	y_1
2	$X_2^\top$	$h_2^\top$	y_2
...			
n	$X_n^\top$	$h_n^\top$	y_n

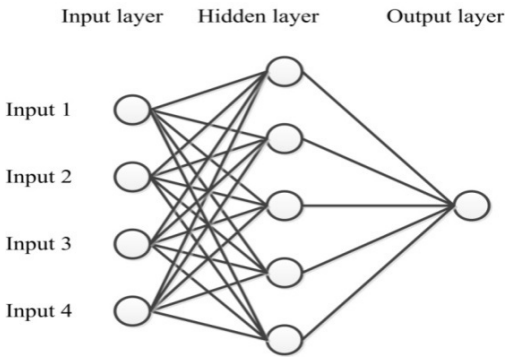


Figure 29: A two layer feedforward neural network. The output follows a logistic regression on the hidden vector. Each component of the hidden vector in turn follows a logistic regression on the input vector.

A perceptron seeks to separate the positive examples and negative examples by projecting them onto a vector β, or in other words, separating them using a hyperplane that is perpendicular to β. If the data are not linearly separable, a perceptron cannot work. We may need to transform the original variables into some other feature space so that the examples can be linearly separated. See Figure 28. One way to solve this

problem is to generalize the perceptron into multi-layer perceptron. This structure is also called feedforward neural network. See Figure 29.

The neural network is logistic regression on top of logistic regressions. $y_i \in \{0,1\}$ follows a logistic regression on $h_i = (h_{ik}, k = 1,...,d)^\top$, and each h_{ik} follows a logistic regression on $x_i = (x_{ij}, j = 1,...,p)^\top$,

$$
\begin{aligned}
& y_i \sim \text{Bernoulli}(p_i), \\
& p_i = \text{sigmoid}(h_i^\top \beta) = \text{sigmoid}\left(\sum_{k=1}^{d} \beta_k h_{ik}\right), \\
& h_{ik} = \text{sigmoid}(x_i^\top \alpha_k) = \text{sigmoid}\left(\sum_{j=1}^{p} \alpha_{kj} x_{ij}\right).
\end{aligned}
$$

Rectified linear unit (ReLU)

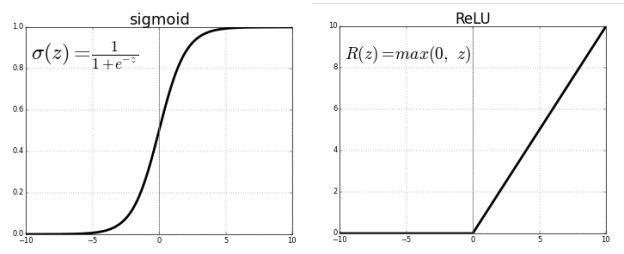


Figure 30: The left is the sigmoid function, and the right is the ReLU function. The sigmoid function saturates at the two ends, causing the gradient to vanish. The ReLU does not saturate for big positive input.

In modern neural nets, the non-linearity is often achieved through rectified linear units (ReLU) $\max(0,a)$. See Figure 30.

$$
\begin{aligned}
& y_i \sim \text{Bernoulli}(p_i), \\
& p_i = \text{sigmoid}(h_i^\top \beta) = \text{sigmoid}\left(\sum_{k=1}^{d} \beta_k h_{ik}\right), \\
& h_{ik} = \max(x_i^\top \alpha_k, 0) = \max\left(\sum_{j=1}^{p} \alpha_{kj} x_{ij}, 0\right).
\end{aligned}
$$

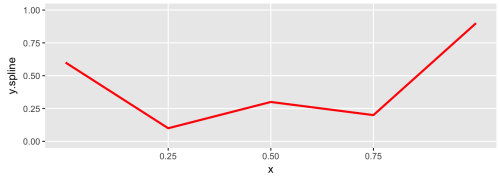


Figure 31: Spline is a continuous piecewise linear function, where at each knot, the spline makes a turn by changing the slope. The neural net can be viewed a high dimensional spline with exponentially many linear pieces.

Recall the linear spline model $f(x_i) = \beta_0 + \sum_{j=1}^{p} \beta_j \max(0, x_i - a_j)$, see Figure 31, where a_j are the knots where the spline makes a turn, i.e., a change of slope by β_j. The neural net with ReLU can be viewed as a

high-dimensional spline or piecewise linear mapping. If there are many layers in the neural net, the number of linear pieces is exponential in the number of layers. It can approximate highly non-linear mapping by patching up the large number of linear pieces.

Back-propagation

When the non-linear rectification is sigmoid, the log-likelihood is

$$\mathscr{L}(\beta,\alpha)=\sum_{i=1}^{n}\left\{y_i\sum_{k=1}^{d}\beta_k h_{ik}-\log\left[1+\exp\left(\sum_{k=1}^{d}\beta_k h_{ik}\right)\right]\right\}.$$

This time we use $\mathscr{L}(\beta,\alpha)$ to denote the log-likelihood function, which is to be maximized.

The gradient is

$$\frac{\partial\mathscr{L}}{\partial\beta}=\sum_{i=1}^{n}(y_i-p_i)h_i,$$
$$\frac{\partial\mathscr{L}}{\partial\alpha_k}=\frac{\partial\mathscr{L}}{\partial h_k}\frac{\partial h_k}{\partial\alpha_k}=\sum_{i=1}^{n}(y_i-p_i)\beta_k h_{ik}(1-h_{ik})x_i$$

$\partial\mathscr{L}/\partial\alpha_k$ is calculated by the chain rule. Again the gradient descent learning algorithm learns from the mistake or error y_i-p_i. The chain rule back-propagates the error to assign the blame to β and α in order for β and α to update. If the current network makes a mistake on the i-th example, β will change to be more aligned with h_i, while each α_k also changes to be more aligned with x_i. The amount of change depends on β_k as well as $h_{ik}(1-h_{ik})$, which measures how big a role played by $x^\top\alpha_k$ in predicting y_i. If $x^\top\alpha_k$ plays a big role, then α_k should receive much blame and change.

For ReLU $\max(0,a)$, we should replace $h_{ik}(1-h_{ik})$ in the back-propagation by $1(h_{ik}>0)$, which is a binary detector. The sigmoid function saturates at the two ends, where the derivatives are close to zero. This can cause the vanishing gradient problem in back-propagation, so that the network cannot learn from the error. The ReLU function does not saturate for big positive input, which indicates the existence of a certain patten. This help avoids the vanishing gradient problem.

10.2 Implicit regularization by gradient descent

For simplicity, let us assume x is one-dimensional. $h_k=\max(0,a_kx+b_k)$, and $s=f(x)=\sum_{k=1}^{d}\beta_k h_k$. As discussed above, $f(x)$ is a linear spline.

Suppose we initialize at $(a_k,b_k)\sim p_0(a,b)$ independently for $k=1,...,d$ and freeze them, and we only learn β_k by gradient descent, starting from $\beta_k^{(0)}=0$. For training examples $\{x_i,y_i,i=1,...,n\}$, the loss function is

$$\mathscr{L}(\beta)=\frac{1}{n}\sum_{i=1}^{n}(y_i-h(x_i)^\top\beta)^2.$$

If $d\le n$, there is a single minimum, and gradient descent converges to this minimum.

When $d>n$, the model is over-parametrized, i.e., the number of parameters is greater than the number of training examples. Then there are infinitely many β that gives $L(\beta)=0$. Gradient descent will converge to one such β.

Gradient descent is

$$\Delta\beta\propto\frac{1}{n}\sum_{i=1}^{n}(y_i-h(x_i)^\top\beta)h(x_i).$$

It will converge to a $\hat{\beta} = \sum_i c_i h(x_i)$ that satisfies $y_i = h(x_i)^\top \hat{\beta}$ for all i. Suppose there is another $\tilde{\beta}$ that satisfies $y_i = h(x_i)^\top \tilde{\beta}$ for all i. Let $\tilde{\beta} = \hat{\beta} + \Delta$, then $\Delta \perp h(x_i)$ for all i, i.e., $\Delta \perp \hat{\beta}$. Thus $\|\tilde{\beta}\|^2 = \|\hat{\beta}\|^2 + \|\Delta\|^2$. Thus gradient descent will converge to $\hat{\beta}$ with minimal ℓ_2 norm. This is implicit regularization, and it underlies the double descent behavior as illustrated by Figure 32. When $p \le n$, there is only one minimum of $\mathscr{L}(\beta)$, and there is no regularization, thus the overfitting becomes severe as p approaches n. But as $p > n$, there are infinitely many β with $\mathscr{L}(\beta) = 0$, and gradient descent provides implicit regularization, so that testing error starts to decreases again.

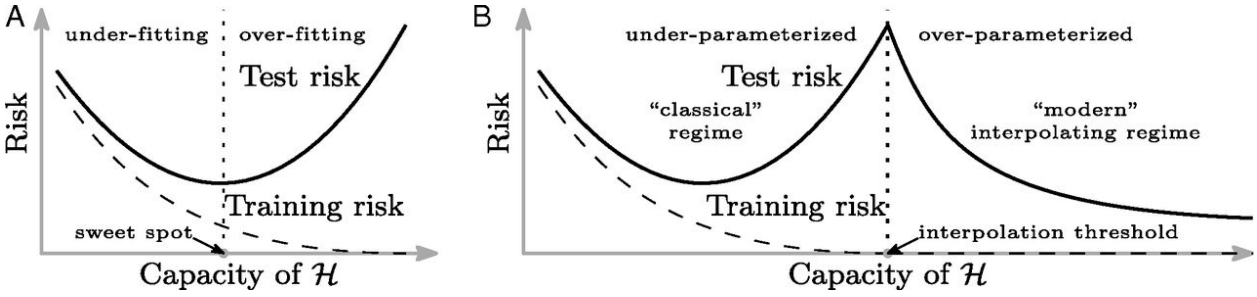


Figure 32: Double descent. Left: testing error for classical statistical model. Right: testing error for modern over-parametrized neural network.

10.3 Connection to kernel machine

Suppose we freeze (a_k, b_k), and only learn β by gradient descent, then it is equivalent to the kernel machine

$$K(x,x') = \langle h(x), h(x') \rangle = \sum_{k=1}^{d} h_k(x) h_k(x') = \frac{1}{d} \sum_{k=1}^{d} \tilde{h_k}(x) \tilde{h_k}(x'),$$

where $\tilde{h_k} = \sqrt{d} h_k$. Assuming $(\sqrt{d}a_k, \sqrt{d}b_k) \sim \tilde{p}_0(a,b)$, where $\tilde{p}_0$ is independent of d, then for large d, by law of large number,

$$K(x,x') \to \mathbb{E}_{\tilde{p}_0(a,b)}[\tilde{h}_k(x)\tilde{h}_k(x')].$$

So this model is equivalent to kernel regression. Even if we free (a_k, b_k) and learn them together with β_k, the learning dynamics is still similar to kernel machine, where, by first-order Taylor expansion, $h(x) = \frac{\partial}{\partial \theta} f_\theta(x)$, i.e., the first derivative evaluated at the initialization θ_0. The kernel is called neural tangent kernel.

10.4 Connection to Gaussian process

We can adopt a Bayesian framework, by assuming a prior distribution $(a_k, b_k, \beta_k) \sim p(a,b)p(\beta)$ independently. Then

$$f(x) = \sum_{k=1}^{d} \beta_k h_k(x) = \frac{1}{\sqrt{d}} \sum_{k=1}^{d} \tilde{\beta}_k h_k(x),$$

where $\tilde{\beta}_k = \sqrt{d}\beta_k$. Assuming $\tilde{\beta} \sim \tilde{p}(\beta)$ independently, where $\tilde{p}(\beta)$ is independent of d, and assuming $\mathbb{E}(\hat{\beta}) = 0$, then according to the central limit theorem, as $d \to \infty$,

$$(f(x_i), i = 1, ..., n)^\top \sim \mathrm{N}(0, K),$$

where $K_{ij} = K(x_i, x_j)$, and $K(x,x') = \mathbb{E}(f(x)f(x'))$ is the kernel. Thus the model is equivalent to the Gaussian process.

10.5 Multi-layer network

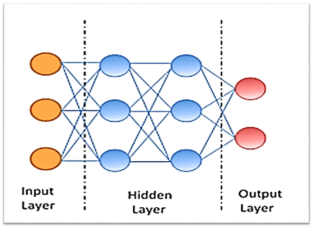


Figure 33: Multi-layer perceptron (two hidden layers here) or feedforward neural network. There is an input layer (or an input vector), an output layer, and multiple hidden layers (or hidden vectors). Each layer is a linear transformation of the layer beneath, followed by an element-wise non-linear transformation.

See Figure 33 for an illustration of multi-layer perceptron or a feedforward neural network. It consists an input layer, an output layer, and multiple hidden layers in between. Each layer can be represented by a vector, which is obtained by multiplying the layer below it by a weight matrix, plus a bias vector, and then transforming each element of the resulting vector by sigmoid or ReLU etc.

More formally, the network has the following recursive structure:

$$\begin{aligned} h_l &= f_l(s_l), \\ s_l &= W_l h_{l-1} + b_l, \end{aligned}$$

for $l = 1, ..., L$, where l denotes the layer, with $h_0 = X$, which is the input vector, and h_L is used to predict Y, which is the output, based on a log-likelihood function $L(Y, h_L)$. Here h_L corresponds to $X^\top \beta$ in the previous section. W_l and b_l are the weight matrix and bias vector respectively. f_l is an element-wise transformation, i.e., each component of h_l is obtained by a non-linear transformation of the corresponding component of s_l, such that $h_{lk} = f_l(s_{lk})$ for each k.

10.6 Multi-layer back-propagation

We may write the forward pass as

$$\begin{array}{ccccccccccc} x & \to & h_1 & \to & \ldots & \to & h_{l-1} & \to & h_l & \ldots \to L. \\ & & \uparrow & & & & \uparrow & & \uparrow & \\ & & W_1 & & \ldots & & W_{l-1} & & W_l & \end{array}$$

The back-propagation pass is

$$\begin{array}{ccccccccccc} x & \leftarrow & h_1 & \leftarrow & \ldots & \leftarrow & h_{l-1} & \leftarrow & h_l & \ldots \leftarrow L. \\ & & \downarrow & & & & \downarrow & & \downarrow & \\ & & W_1 & & \ldots & & W_{l-1} & & W_l & \end{array}$$

This is a process of assigning blame. If the loss function L finds something wrong, he will blame h_L and W_{L+1}. This layer is usually the soft-max layer. h_L will then blame h_{L-1} and W_L, and so on. This process

follows the chain rule:

$$\begin{aligned}\frac{\partial L}{\partial h_{l-1}^\top} &= \sum_{k=1}^{d} \frac{\partial L}{\partial h_{l,k}} \frac{\partial h_{l,k}}{\partial s_{l,k}} \frac{\partial s_{l,k}}{\partial h_{l-1}^\top} \\ &= \sum_{k=1}^{d} \frac{\partial L}{\partial h_{l,k}} f_l'(s_{l,k}) W_{l,k} \\ &= \frac{\partial L}{\partial h_l^\top} f_l' W_l,\end{aligned}$$

where $W_{l,k}$ is the k-th row of W_l, and $f_l' = \text{diag}(f_l'(s_{l,k}), k = 1, ..., d)$.

$$\begin{aligned}\frac{\partial L}{\partial W_{l,k}} &= \frac{\partial L}{\partial h_{l,k}} \frac{\partial h_{l,k}}{\partial s_{l,k}} \frac{\partial s_{l,k}}{\partial W_{l,k}} \\ &= \frac{\partial L}{\partial h_{l,k}} f_l'(s_{l,k}) h_{l-1}^\top,\end{aligned}$$

thus,

$$\frac{\partial L}{\partial W_l} = f_l' \frac{\partial L}{\partial h_l} h_{l-1}^\top.$$

Similarly,

$$\frac{\partial L}{\partial b_l} = f_l' \frac{\partial L}{\partial h_l}.$$

10.7 Stochastic gradient descent (SGD)

Mini-batch

Let the training data be (x_i, y_i), $i = 1, ..., n$. Let $L_i(\theta) = L(y_i, x_i; \theta)$ be the loss caused by (x_i, y_i). For regression, $L(y_i, x_i; \theta) = (y_i - f(x_i))^2$, where $f(x_i)$ is parametrized by a neural network with parameters θ. For classification, $L(y_i, x_i; \theta) = -\log p(y_i|x_i)$ where $p(y_i|x_i)$ is modeled by a neural network with parameters θ, with a softmax layer at the top. Let $\mathscr{L}(\theta) = \frac{1}{n}\sum_{i=1}^n L_i(\theta)$ be the overall loss averaged over the whole training dataset. The gradient descent is

$$\theta_{t+1} = \theta_t - \eta \mathscr{L}'(\theta_t),$$

where η is the step size or learning rate, and $\mathscr{L}'(\theta)$ is the gradient.

The above gradient descent may be time consuming because we need to compute $\mathscr{L}'(\theta) = \frac{1}{n}\sum_{i=1}^n L_i'(\theta)$ by summing over all the examples. If the number of examples is large, the computation can be time consuming. We may use the following stochastic gradient descent algorithm. At each step, we randomly select i from $\{1, 2, ..., n\}$. Then we update θ by

$$\theta_{t+1} = \theta_t - \eta_t L_i'(\theta_t),$$

where η_t is the step size or learning rate, and $L_i'(\theta_t)$ is the gradient only for the i-th example. Because i is randomly selected, the above algorithm is called the stochastic gradient descent algorithm.

Instead of randomly selecting a single example, we may randomly select a mini-batch, and replace $L_i'(\theta_t)$ by the average of this mini-batch.

In order for the algorithm to converge to a local minimum, we need the following conditions. (1) $\sum_{t=1}^\infty \eta_t = \infty$. (2) $\sum_{t=1}^\infty \eta_t^2 < \infty$. The first condition ensures that the algorithm can go the distance toward the minimum. The second condition ensures that the algorithm will not run away from the local minimum once it arrives. One simple example is $\eta_t = c/t$ for a constant c. In practice, we need more sophisticated schemes for choosing η_t. For instance, reducing η_t after a certain number of steps, or reducing η_t if the training error stops decreasing.

Momentum, Adagrad, RMSprop, Adam

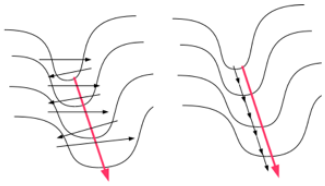


Figure 34: Momentum. The left figure illustrates the original gradient descent algorithm. The black arrow is the gradient direction, and the red arrow is the preferred direction. The right figure illustrates the gradient descent algorithm with momentum, which is along the red arrow.

The gradient descent algorithm goes downhills in the steepest direction in each step. However, the steepest direction may not be the best direction, as illustrated by Figure 34. In the left figure, the black arrows are the gradient direction. The red arrows are the preferred direction, which is the direction of momentum. It is better to move along the direction of the momentum, as illustrated by the right figure. We want to accumulate the momentum, and let it guide the descent. The following is the stochastic gradient descent with momentum:

$$v_t = \gamma v_{t-1} + \eta_t g_t,$$
$$\theta_t = \theta_{t-1} - v_t.$$

where g_t is the average gradient computed from the current mini-batch, and v_t is the momentum or velocity. γ is usually set at .9, for accumulating the momentum, and θ is updated based on the momentum.

Adagrad modifies the gradient descent algorithm in another direction. The magnitudes of the components of g_t may be very uneven, and we need to be adaptive to that. The Adagrad let

$$G_t = G_{t-1} + g_t^2,$$
$$\theta_{t+1} = \theta_t - \eta_t \frac{g_t}{\sqrt{G_t + \varepsilon}},$$

where ε is a small number to avoid dividing by 0. In the above formula, g_t^2 and $g_t/\sqrt{G_t+\varepsilon}$ denote component-wise square and division.

In Adagrad, G_t is the sum over all the time steps. It is better to sum over the recent time steps. Adadelta and RMSprop use the following scheme:

$$G_t = \beta G_{t-1} + (1-\beta) g_t^2,$$

where β can be set at .9. It can be shown that

$$G_t = (1-\beta)(\beta^{t-1} g_1^2 + \beta^{t-2} g_2^2 + ... + \beta g_{t-1}^2 + g_t^2),$$

which is a sum over time with decaying weights.

The Adam optimizer combines the idea of RMSprop and the idea of momentum.

$$v_t = \gamma v_{t-1} + (1-\gamma) g_t,$$
$$G_t = \beta G_{t-1} + (1-\beta) g_t^2,$$
$$v_t \leftarrow v_t/(1-\gamma), G_t \leftarrow G_t/(1-\beta),$$
$$\theta_{t+1} = \theta_t - \eta_t \frac{v_t}{\sqrt{G_t + \varepsilon}}.$$

10.8 Convolutional neural networks (CNN, ConvNet)

Convolution, kernels, filters

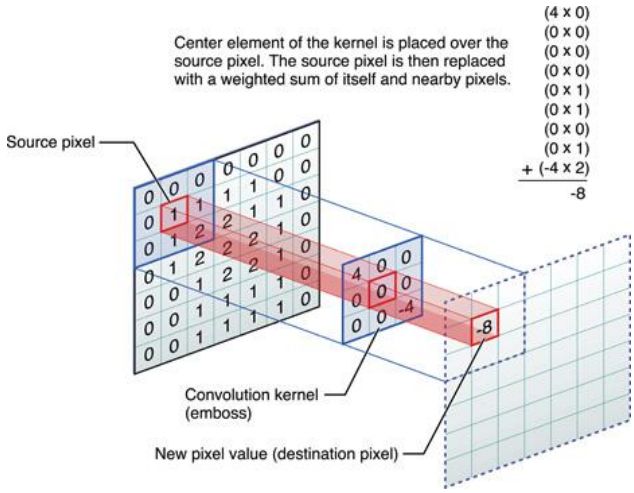


Figure 35: Local weighted summation. Here the filter or kernel is 3×3. It slides over the whole input image or feature map. At each pixel, we compute the weighted sum of the 3×3 patch of the input image, where the weights are given by the filter or kernel. This gives us an output image or filtered image or feature map.

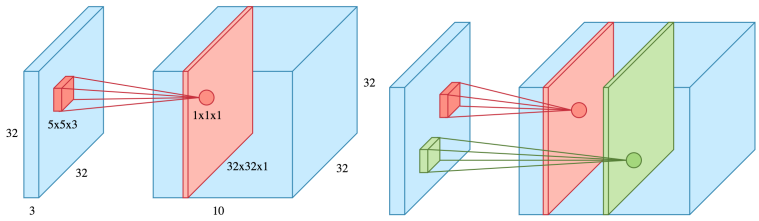


Figure 36: Convolution. The input may consist of multiple channels, illustrated by a rectangle box. Each input feature map is a slice of the box. The local weighted summation in convolution is also over the channels. If the spatial range of the filter is 3×3 and the input has 3 channels, then the filter or kernel is $3 \times 3 \times 3$ box. The spatial range can also be 1×1, then the filter involves weighted summation of the 3 channels at the same pixel. For the input image, there are 3 channels corresponding to 3 colors: red, green, black. For the hidden layers, each layer may consist of hundreds of channels. Each channel is obtained by a filter. For instance, in the figure on the right, the red feature map is obtained by the red filter, and the green feature map is obtained by the green filter.

In the neural network, $h_l = f_l(W_l h_{l-1} + b_l)$, the linear transformation $s_l = W_l h_{l-1} + b_l$ that maps h_{l-1} to s_l can be highly structured. One important structure is the convolutional neural network, where W_l and b_l have a convolutional structure.

Specifically, a convolutional layer h_l is organized into a number of feature maps. Each feature map, also called a channel, is obtained by a filter or a kernel operating on h_{l-1}, which is also organized into a number of feature maps. Each filter is a local weighted summation, plus a bias term, followed by a non-linear transformation.

Figure 35 explains the local weighted summation. We convolve an input feature map with a 3×3 filter to obtain an output feature map. The value of each pixel of the output feature map is obtained by a weighted

summation of pixel values of the 3×3 patch of the input feature map around this pixel. We apply the summation around each pixel, using the same weights, to obtain the output feature map.

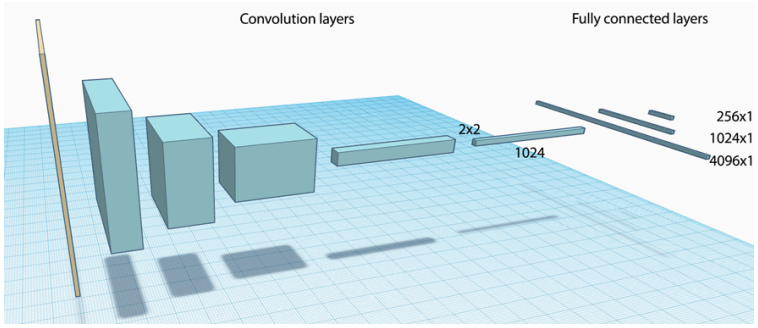


Figure 37: Convolutional neural network (CNN or ConvNet). The input image has 3 channels (R, G, B). Each subsequent layer consists of multiple channels, and is illustrated by a box. Sub-sampling is performed after some layers, so that the the box at the higher layers is smaller than the box at the lower layer in the spatial extent. Meanwhile, the box at the higher layer may have more channels than the box at the lower layer, illustrated by the fact that the box at the higher layer is longer than the box at the lower layer. A fully connected layer consists of 1×1 feature maps, and is illustrated by a horizontal line.

If there are multiple input feature maps, i.e., multiple input channels, the weighted summation is also over the multiple feature maps. We can apply different filters to the same set of input feature maps. Then we will get different output feature maps. Each output feature map corresponds to one filter. See Figure 36 for illustration.

After the weighted summation, we may also add a bias and apply a non-linear transformation such as sigmoid or ReLU. The filter becomes non-linear.

After obtaining the feature maps in h_l, we may perform max-pooling, e.g., for each feature map, at each pixel, we replace the value of this pixel by the maximum of the 3×3 patch around this pixel. We may also do average pooling, i.e., at each pixel, we replace the value of this pixel by the average of the 3×3 patch around this pixel.

The mapping from h_l to h_{l-1} may also involve sub-sampling to reduce the size of feature maps. For instance, after obtaining a feature map by a filter, we can partition the feature map into 2×2 blocks, and within each block, we only keep the upper left pixel. This will reduce the width and height of the feature map by half. h_{l-1} may have more channels than h_l, because each element of h_l covers a bigger spatial extent than each element of h_{l-1}, and there are more patterns of bigger spatial extent.

The output feature map may also be 1×1, whose value is a weighted sum of all the elements in h_{l-1}. h_l may consists of a number of such 1×1 maps. It is called the fully connected layer because each element of h_l is connected to all the elements in h_{l-1}.

A convolutional network consists of multiple layers of convolutional and fully connected layers, with max pooling and sub-sampling between the layers. See Figure 37. The network can be learned by back-propagation.

Alex net, VGG net, inception net

The Alex net is the neural network that achieved the initial breakthrough on object recognition for the ImageNet dataset. It goes through several convolutional layers, followed by fully connected layers. It has 5 convolutional layers and 2 fully connected layers, plus a softmax output layer.

The VGG net is an improvement on the Alex net. There are two versions, VGG16 and VGG19, which

consist of 16 hidden layers and 19 hidden layers respectively. The VGG19 has 144 million parameters. The filters of the VGG nets are all 3×3. Like Alex net, it has two fully connected layers, plus a softmax output layer.

The inception net took its name from the movie "Inception," which has a line "we need to go deeper." The network makes extensive use of 1×1 filters, i.e., for each feature map in h_l, each pixel value is a weighted summation of the pixel values of all the feature maps in h_{l-1} at the same pixel, plus a bias and a non-linear transformation. The 1×1 filters serve to fuse the channels in h_{l-1} at each pixel. The feature maps at each layer of the inception net are obtained by filters of sizes 1×1, 3×3 and 5×5, as well as max pooling.

Object detection, semantic segmentation, deformable CNN

CNN is commonly used for computer vision. Two prominent tasks are object detection and semantic segmentation. Object detection is to impose bounding boxes on the objects in the image and output the object categories. Semantic segmentation is to segment the image into different regions, with each region corresponding to an object whose category is labeled.

In object detection, we can use CNN to predict the possible bounding boxes and the shifts of the bounding boxes.

In deformable CNN, we can use CNN to predict the deformations of the grids that support the convolutional kernels.

10.9 Batch normalization

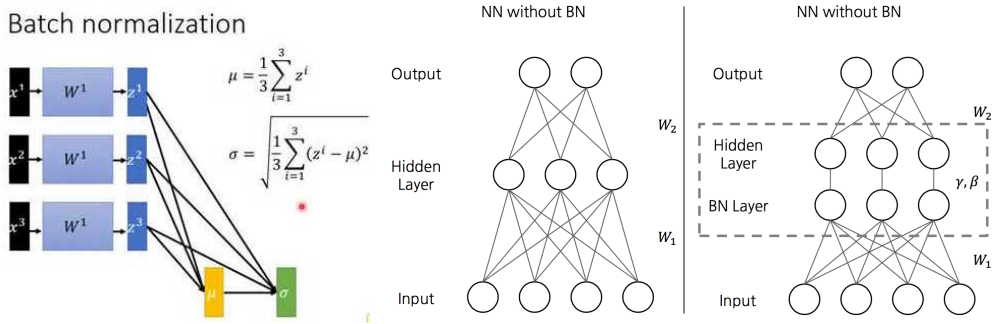


Figure 38: Batch normalization. Suppose we have a batch of 3 examples. For each element of each layer, we compute the mean μ and standard deviation σ by pooling over the 3 examples. We then normalize the element for each example, followed by a linear transformation to be learned from the data. In back-propagation, we need to treat μ and σ as a layer and compute the derivatives of μ and σ with respect to the whole batch. It is as if the whole batch becomes a single example because μ and σ are computed from this whole batch.

When training the neural net by back-propagation, the distribution of h_l keeps changing because the parameters keep changing. This may cause a problem in training. We can stabilize the distribution by batch normalization. That is, between h_{l-1} and h_l, we can add a batch normalization layer. For simplicity, let x be the input to the batch normalization layer, and let y be the output of the batch normalization layer. For instance, x is h_{l-1}, and y becomes the normalized version of h_{l-1} to be fed into the layer for computing h_l.

With slight abuse of notation, we let x be an element of h_{l-1}, and we apply the batch normalization for each element of h_{l-1}. Suppose we have a batch of n training examples, so that we have $\{x_i, i = 1, ..., n\}$. The

batch normalization layer is defined as follows:

$$\begin{aligned}
\mu &= \frac{1}{n}\sum_{i=1}^{n} x_i; \\
\sigma^2 &= \frac{1}{n}\sum_{i=1}^{n}(x_i-\mu)^2; \\
\hat{x}_i &= \frac{x_i-\mu}{\sigma}; \\
y_i &= \beta+\gamma\hat{x}_i.
\end{aligned}$$

This way, we stablize the distribution of each element of h_{l-1} during the training process. See Figure 38 for an illustration.

Note that in the batch normalization layer, there are parameters β and γ to be learned. More importantly, μ and σ^2 are functions of the whole batch. When we do chain rule calculations for back-propagation, we need to compute the derivatives of μ and σ^2 with respect to all the $(x_i, i=1,...,n)$. It is as if the whole batch becomes a single training example.

10.10 Residual net

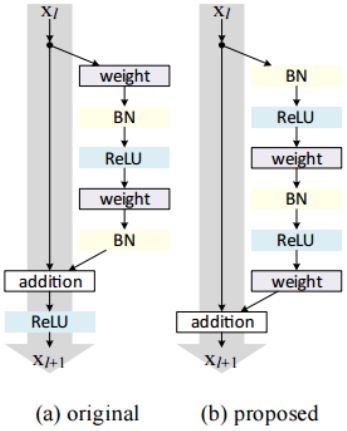


Figure 39: Residual net. The figure on the left is the original version of a residual block. The figure on the right is the revised version, which is in the form of $x_{l+1} = x_l + F(x_l)$, where F consists of two rounds of weighted sum, batch normalization, and ReLU.

A residual block in the residual net is as follows. Let x_l be the input to the residual block. Let x_{l+1} be the output of the residual block. Let $F(x_l)$ be the transformation of x_l that consists of two rounds of weighted summation, batch normalization, and ReLU. We let

$$x_{l+1} = x_l + F(x_l),$$

as illustrated by the right plot of Figure 40. $F(x_l)$ models the residual of the mapping from x_l to x_{l+1}, on top of the identity mapping. The following are some rationales for such a residual block.

(1) If we model $x_{l+1} = F(x_l)$, mathematically $F(x_l)$ parametrized by a set of weights may be the same as $x_l + F(x_l)$ parametrized by a different set of weights, computationally it can be more difficult for gradient descent to learn the former than the latter. It can be much easier for stochastic gradient descent to find a

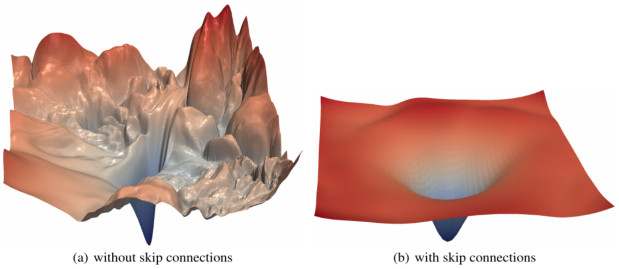


Figure 40: Visualizing loss function. Left: regular network. Right: residual net.

good $F(x_l)$ in the residual form than in the original form. Specifically, the loss function with residual net parametrization is much smoother.

(2) The mapping $x_{l+1} = x_l + F(x_l)$ may model an iteration of an iterative algorithm or dynamic process, where l actually denotes the time step t instead of the real layer l. The mapping models the iterative refinement of the same layer over time.

(3) With multiple residual blocks, we implicitly have an ensemble of networks. For instance, consider a simple network $y = w_2 w_1 x$, where all the symbols are scalers. If we adopt a residual form $y = (1+w_2)(1+w_1)x$, we can expand it as $y = x + w_1 x + w_2 x + w_2 w_1 x$. Thus the residual form is an ensemble of 4 networks. We may also think of the residual form as an expansion, like the Taylor expansion.

10.11 Recurrent neural networks (RNN), LSTM, GRU

Vanilla RNN

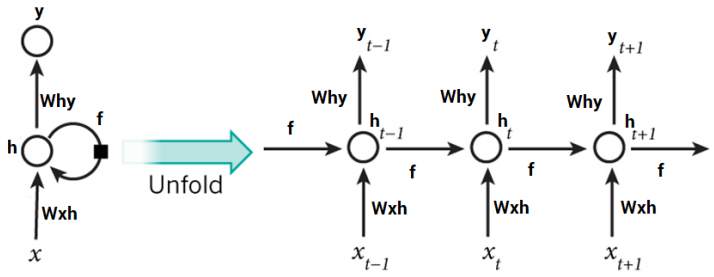


Figure 41: Recurrent neural network. The latent vector summarizes all the information about the past.

Figure 41 shows the vanilla version of the recurrent neural network, which can be written as

$$h_t = f(W(h_{t-1}, x_t)$$
$$y_t = f(Wh_t),$$

where we use W as a generic notation for weight matrices, and f as a generic notation for element-wise non-linear transformations. They are different at each occurrence. (h_{t-1}, x_t) is the concatenated vector. We omit the bias, which can be absorbed into the weight matrix if we add 1 to the vector.

In the case of prediction, $y_t = x_{t+1}$. There are different forms of inputs and outputs for different applications of RNN, as illustrated by Figure 42.

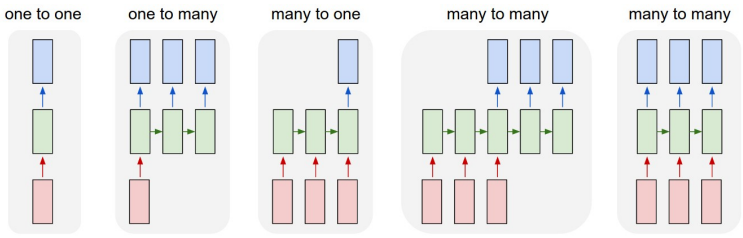


Figure 42: Different forms of inputs and outputs in RNN.

Long short term memory (LSTM)

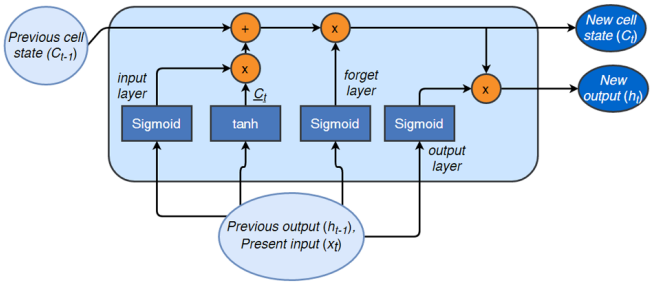


Figure 1. Architecture in a LSTM network

Figure 43: Long short term memory. There is a memory vector, and three gates, forget, input, and output.

The training of RNN is again based on back-propagation, except that we need to back-propagate over time. There is a vanishing (or exploding) gradient problem. The LSTM was designed to overcome this problem by introducing memory cells.

Figure 43 illustrates the architecture of LSTM, which can be written as

$$
\begin{aligned}
(f_t, i_t, o_t, \Delta c_t) &= f(W(h_{t-1}, x_t)), \\
c_t &= c_{t-1} f_t + \Delta c_t i_t, \\
h_t &= o_t f(c_t), \\
y_t &= f(W h_t).
\end{aligned}
$$

The gates are like if-then statements in a computer program. They are made continuous by sigmoid f so that they are differentiable. The residual net is a special case of LSTM.

Gated recurrent units (GRU)

The gated recurrent unit is a simplification of LSTM, by merging c_t and h_t:

$$
\begin{aligned}
(z_t, r_t) &= f(W(h_{t-1}, x_t)), \\
\tilde{h}_t &= f(W(x_t, r_t h_{t-1})), \\
h_t &= (1 - z_t) h_{t-1} + z_t \tilde{h}_t, \\
y_t &= f(W h_t).
\end{aligned}
$$

z_t is a gate to decide whether to renew or not, and r_t is a gate to decide whether to remember or not.

10.12 Encoder-decoder, thought vector

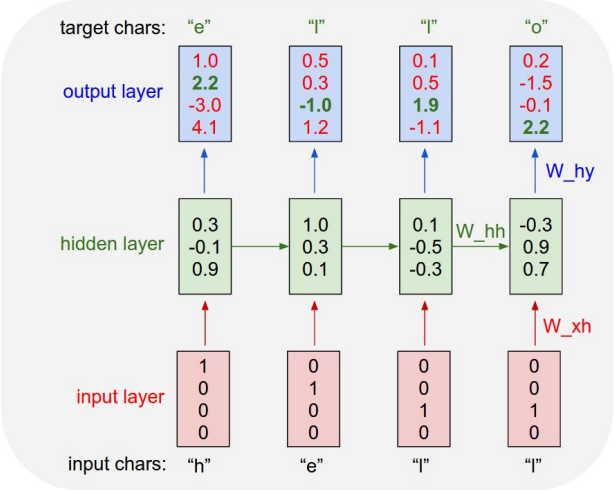


Figure 44: An RNN for sequence generation.

Figure 44 shows an RNN for sequence generation, where x_t is the current letter, and y_t is the next letter. Each letter can be represented by a one-hot vector. The RNN maps the one-hot vector into a hidden vector, and the hidden vector is used to generate the next letter by softmax probabilities. We can think of $x_t \to h_t$ as the encoder, and $h_t \to y_t$ as the decoder. Because h_t also depends on h_{t-1}, the h_t is a thought vector that encodes all the relevant information of the past. That is, we have the following scheme: input $\to$ thought vector $\to$ output.

Translation

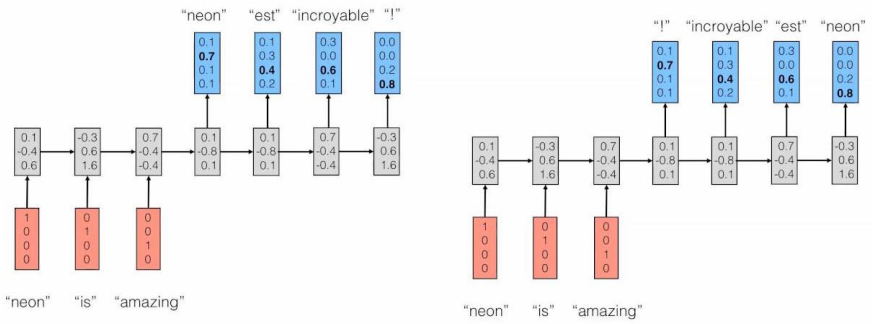


Figure 45: Machine translation. The encoding RNN encodes the input sentence into a thought vector, and the decoding RNN decodes the thought vector into an output sentence.

Figure 45 shows the RNN for machine translation. This time, each word is represented by a one-hot vector. We encode the input sentence into a thought vector by an encoding RNN. The thought vector is fed into a decoding RNN to generate the output sentence.

Image captioning

Image captioning can be considered a special case of machine translation, where the encoding RNN is replaced by a CNN (e.g., VGG). We can take a certain layer of CNN as the thought vector to be fed into the decoding RNN to generate the caption.

VQA

The visual question and answering can also be considered a special case of machine translation, where we have an encoder for images and an encoder for questions. We then concatenate the thought vectors of the image and question, and use the concatenated thought vector to generate the answer.

10.13 Memory and attention, query and key

For text based QA, e.g., answering questions based on wikipedia, we can encode the sentences in wikipedia into thought vectors. These vectors serve as memory. For an input question, we can encode it into a thought vector. By matching the thought vector of the question to the thought vectors of the memory, we can decide which sentence we want to pay attention to, using a softmax probability distribution over the sentences. The weighted sum of the thought vectors of the sentences weighted by the attention probabilities then becomes a combined thought vector, which, together with the thought vector of the question, is decoded into the answer.

Specifically, let $m_i, i = 1, ..., N$ be the vectors in the memory. Let Q be the question or query. We can let $(key_i, value_i) = Wm_i$. Then we can let

$$A = \sum_{i=1}^{N} attention_i \times value_i,$$

where

$$attention_i = \frac{\exp(\langle Q, key_i \rangle)}{\sum_{i'=1}^{N} \exp(\langle Q, key_{i'} \rangle)}.$$

We can then let A emit the answer by soft-max classification.

10.14 Word2vec for semantic embedding

Word embedding is at the foundation of modern Natural language processing (NLP). We embed the words into an Euclidean space. Suppose there are K words in the dictionary. We can represent each word x as a K-dimensional one-hot vector. Suppose we want to embed the words in the d-dimensional space. We can let $h = Wx$, so that the k-th column of W, i.e., w_k, is the embedding of word k, whose x is a one-hot vector with k-th element being 1. This is a linear encoding. We can decode it by a linear soft-max classifier, i.e., $p(x|h) \propto \exp(\langle x, \tilde{W}h \rangle)$, where $\tilde{W}$ is the decoding matrix, and $\tilde{W}h$ is the K-dimension score vector. We may train this model to predict the words within the context of each word. For instance the probability that word j is within the context of word i is

$$Q_{ij} = \frac{\exp(\langle w_i, \tilde{w}_j \rangle)}{\sum_k \exp(\langle w_i, \tilde{w}_k \rangle}.$$

The GloVe (Global Vector) is similar to word2vec, where $\log Q_{ij} = \langle w_i, \tilde{w}_j \rangle + b_i + \tilde{b}_j$, where b_i and $\tilde{b}_j$ are bias terms. This is similar to matrix factorization in recommender system.

It is likely that we use both vector and symbolic representations. In a certain context and for a short term task, the number of words or symbols can be made very small, so that it may be convenient to use symbols for reasoning. For the long term accumulation of knowledge, the dictionary can be very large, and the vector

representation may be useful to properly represent the semantics of the words. This is like the particle/wave duality in quantum physics, where the one-hot representation is on the side of particle, and the dense vector embedding is on the side of wave.

10.15 Self-attention, Transformer, GPT, BERT

For a sentence that consists of words $x_1, ..., x_M$, we can embed the words as $h_1, ..., h_M$, and process them using LSTM or its multi-layer bi-directional versions. A more efficient and flexible sequence is to update the vectors by self-attention mechanism. Specifically, we can let $(query, key, value)_m = Wh_m$. Then we update h_m by $h_m \leftarrow h_m + \Delta h_m$, so that

$$\Delta h_m = \sum_{k=1}^{M} attention_{m \to k} \times value_k,$$

where

$$attention_{m \to k} = \frac{\exp(\langle query_m, key_k \rangle}{\sum_{k'=1}^{M} \exp(\langle query_m, key_{k'} \rangle)}.$$

which involves soft-max attention based on the matching between query and key. We can also use multiple heads, so that each head has its own system of query, key and value, and then add up multiple heads. The above update can be iterated for multiple steps. Such a system was called Transformer, which was trained for machine translation with an encoder and a decoder, both involving self-attention mechanism. Later the decoder part is adopted by GPT (generative pre-training), and the encoder part is adopted by BERT (bi-directional encoder representations from transformers). BERT is trained by self-supervision task of filling in masked words in the sentences.

11 Representation Learning with Deep Nets

Similar to supervised learning such as kernel machines, boosting machines, and neural networks, which are based on $f(x) = h(x)^\top \beta$, in representation learning, we represent the input x by a representation h. However, without an output y to predict, we want the hidden vector h to capture key patterns and structures in x, and we want h to be simple, such as being of low dimension, being sparse, or having independent components, or being error resistant. We may think of h as an encoding of x, and we want to be able to decode x from the code h. We may also think of h as an embedding of x in an Euclidean space, and we want h to mirror the relationship between x, e.g., the relation between (h, h') mirrors the relation between (x, x'), especially for nearby x and x'.

11.1 Three lessons of deep learning

The following are three lessons offered by deep learning.

(1) Multi-layer function approximator.

(2) Vector representation. The vector can be interpreted as a code, a thought vector, or an embedding.

(3) Learned computation. The inference or sampling algorithm can be approximated by a deep network.

While (1) is most important and enables (2) and (3), (2) can be quite important in itself, and does not have to involve deep network. The representation does not need to be a vector. It can be a matrix or more structured object. (3) may be considered learning from the results of on-line computation, i.e., learning from internal data produced by on-line computation. It is like muscle memory.

11.2 Unsupervised learning

In unsupervised learning, the dataset is as below where y_i are not provided as supervision. In a generative

	input	hidden	output
1	$x_1^\top$	$h_1^\top$	?
2	$x_2^\top$	$h_2^\top$	?
...			
n	$x_n^\top$	$h_n^\top$	?

model, the vector h_i is not a vector of features extracted from the signal x_i. h_i is a vector of hidden variables that is used to generate x_i, as illustrated by the following diagram:

$$\begin{array}{rc} \text{hidden}: & h_i \\ & \downarrow \\ \text{input}: & x_i \end{array}$$

The components of the d-dimensional h_i are variably called factors, sources, components or causes.

11.3 Thought vector, auto-encoding, embedding, disentangling

Auto-encoder: h_i is also called a code in the auto-encoder illustrated by the following diagram:

$$\begin{array}{rc} \text{code}: & h_i \\ & \uparrow\downarrow \\ \text{input}: & x_i \end{array}$$

The direction from h_i to x_i is called the decoder, and the direction from x_i to h_i is called the encoder.

Distributed representation and disentanglement: $h_i = (h_{ik}, k = 1, ..., d)$ is called a distributed representation of x_i. Usually the components of h_i, $(h_{ik}, k = 1, ..., d)$, are assumed to be independent, and (h_{ik}) are said to disentangle the variations in x_i.

Embedding: h_i can also be considered the coordinates of x_i, if we embed x_i in a low-dimensional space, as illustrated by the following diagram:

$$\begin{array}{c} \leftarrow h_i \rightarrow \\ | \\ \leftarrow x_i \rightarrow \end{array}$$

In the training data, we find a h_i for each x_i, so that $\{h_i, i = 1, ..., n\}$ preserve the relative relations between $\{x_i, i = 1, ..., n\}$. The prototype example of embedding is multi-dimensional scaling, where we want to preserve the Euclidean distances between the examples.

11.4 Kullback-Leibler divergence

The Kullback-Leibler divergence from p to q is defined as

$$\mathrm{KL}(p|q) = \mathbb{E}_p[-\log q(x)] - \mathbb{E}_p[-\log p(x)] = \mathbb{E}_p[\log(p(x)/q(x)].$$

According to Jensen inequality, for the concave function $\log(x)$,

$$\mathbb{E}[\log(x)] \leq \log(\mathbb{E}(x)),$$

so

$$\mathbb{E}_p[\log(q(x)/p(x))] \leq \log \mathbb{E}_p[q(x)/p(x)] = \log \int [\frac{q(x)}{p(x)} p(x)]dx = \log 1 = 0.$$

Thus $\mathrm{KL}(p|q) \geq 0$.

The KL-divergence is not a metric or distance in that it does not satisfy the triangle inequality.

For conditional distribution, KL involves expectation with respect to the random variable being conditioned on:

$$\mathrm{KL}(p(y|x)|q(y|x)) = \mathbb{E}_{p(x,y)}\left[\log \frac{p(y|x)}{q(y|x)}\right] = \mathbb{E}_{p(x)}\mathbb{E}_{p(y|x)}\left[\log \frac{p(y|x)}{q(y|x)}\right].$$

Let $p(x)$ and $q(x)$ be the marginal distributions, then

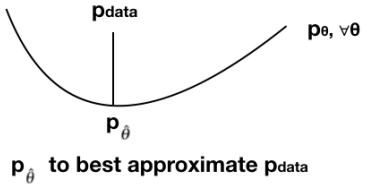
$$\begin{aligned}
\mathrm{KL}(p(x,y)|q(x,y)) &= \mathbb{E}_p\left[\log \frac{p(x,y)}{q(x,y)}\right] \\
&= \mathbb{E}_p\left[\log \frac{p(x)p(y|x)}{q(x)q(y|x)}\right] \\
&= \mathbb{E}_p\left[\log \frac{p(x)}{q(x)}\right] + \mathbb{E}_p\left[\log \frac{p(y|x)}{q(y|x)}\right] \\
&= \mathrm{KL}(p(x)|q(x)) + \mathrm{KL}(p(y|x)|q(y|x)).
\end{aligned}$$

Two sources of KL-divergence are: (1) Large deviation, where the KL-divergence plays the role of the exponential rate of probability. (2) Coding theory, where the KL-divergence measures the coding redundancy.

KL divergence is adopted in information theory for redundancy measurement, in probabilistic theory for rate of probability measurement. And in statistics, it is used for maximum likelihood estimation.

MLE vs KL divergence

Suppose we observe training examples $x_i \sim p_{\text{data}}(x)$ independently for $i = 1, ..., n$. For big n, we have $\frac{1}{n}\sum_{i=1}^n f(x_i) \doteq \mathbb{E}_{p_{\text{data}}}[f(x)]$. Consider a model $p_\theta(x)$. We can learn θ by maximizing the log-likelihood $L(\theta) = \frac{1}{n}\sum_{i=1}^n \log p_\theta(x_i)$. As $n \to \infty$, $L(\theta) = \frac{1}{n}\sum_{i=1}^n \log p_\theta(x_i) \to E_{p_{\text{data}}}[\log p_\theta(x)]$. Since $\mathrm{KL}(p_{\text{data}}|p_\theta) = \mathbb{E}_{p_{\text{data}}}[\log p_{\text{data}}(x)] - \mathbb{E}_{p_{\text{data}}}[\log p_\theta(x)]$, $\hat{\theta} = \arg\max_\theta L(\theta) = \arg\min_\theta \mathrm{KL}(p_{\text{data}}|p_\theta)$. The maximum likelihood estimate can be seen as the most plausible explanation of the data.



Inclusive and exclusion KL

For MLE, we minimize $\mathrm{KL}(p_{\text{data}}|p_\theta)$, which is expectation with respect to p_{data}. p_θ seeks to cover all the modes of p_{data}. This will cause the mode inclusion behavior and the over-dispersion of the learned p_θ.

Sometimes we want p_θ to approximate a given target distribution, p_{target}, which we only need to know up to a normalizing constant. We can minimize $\mathrm{KL}(p_\theta|p_{\text{target}})$, which is expectation with respect to p_θ. This is variational approximation. Thus p_θ can ignore some minor modes and only focus on major modes. This will cause the mode exclusion or mode collapsing behavior. Figure 46 illustrates the basic idea.

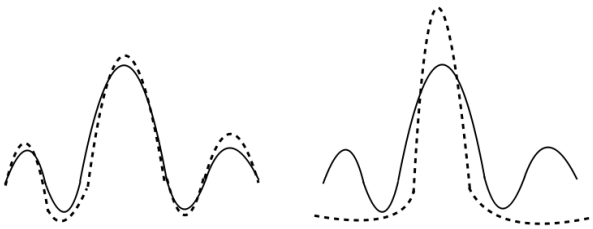


Figure 46: Left: $\min_\theta \mathrm{KL}(p_{\text{data}}|p_\theta)$ vs Right: $\min_\theta \mathrm{KL}(p_\theta|p_{\text{target}})$. The dotted line is p_θ.

11.5 Decoder

Let h be the latent variables. We can decode it into an image $x = g_\theta(h)$, where g_θ is a neural network with parameters θ. This is in the reverse direction from neural network for unsupervised learning, which maps x to h. We call the neural network from x to h a bottom-up encoder network, and the network from h to x the top-down decoder network. While the bottom-up network may be a convolutional network, the top-down network is sometimes called the deconvolutional network.

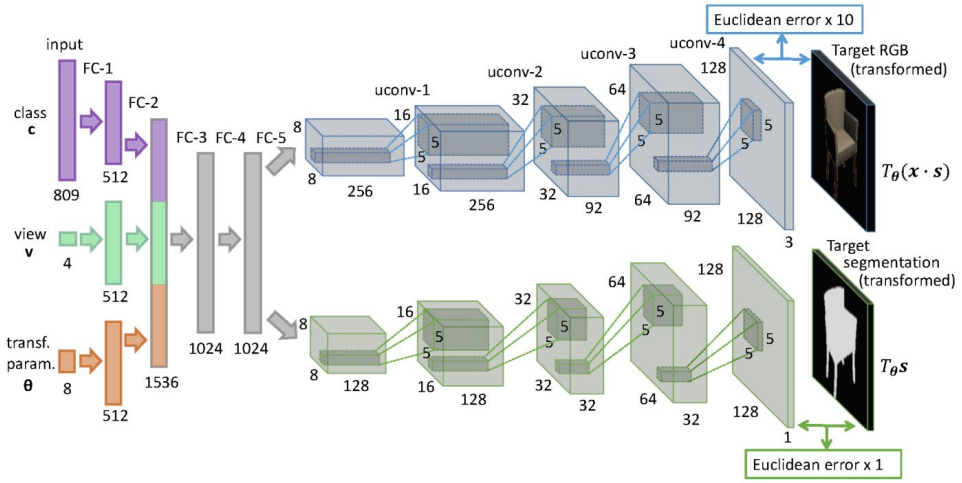


Figure 47: A top-down decoder network, where h consists of class (one-hot), view and transform parameters of chair.

Figure 47 shows the architecture of a decoder network, which is trained on chair images, where h consists of class, which is a one-hot vector, view and transform parameters, and x is the image of the corresponding chair. We can learn the decoder by minimizing $|x - g_\theta(h)|^2$ if we have training data $\{h_i, x_i\}$. After training the decoder network, we can interpolate the h vectors of two different chairs, and generate the interpolated chair by the learned decoder. You can see that the learned network has amazing interpolative ability.

For image processing, such as style transfer, we can again use an encoder and decoder scheme. We can encode the input image into a thought vector. We can also encode the style as another vector. We can then concatenate the two vectors, and feed it into a decoder, to generate the output image.

11.6 Generative adversarial networks (GAN)

We can learn the decoder model in unsupervised manner, by treating h as a latent vector. We can also assume a simple known prior distribution on h, $h \sim p(h)$. Then the decoder defines a generative model. We call such a model a generator model, which is a non-linear generalization of the factor analysis model.

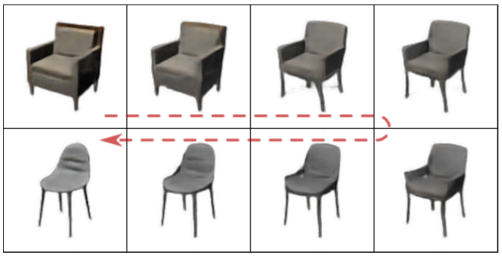


Figure 48: Interpolating the one-hot vectors of two types of chairs.

The model can be learned by generative adversarial networks (GAN), where we pair the generator model G with a discriminator model D, where for an image x, $D(x)$ is the probability that x is a true image instead of a generated image.

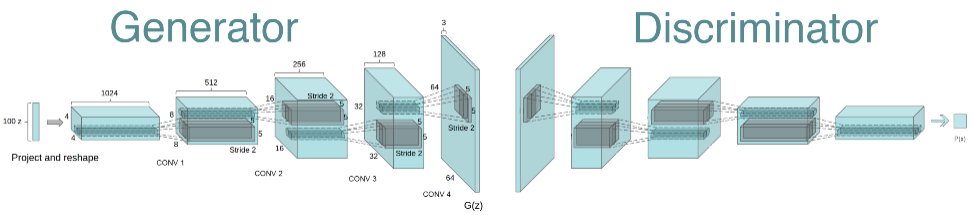


Figure 49: Generative adversarial networks (GAN) consists of a generator and a discriminator.

We can train the pair of (G,D) by an adversarial, zero-sum game. Specifically, let $G(h) = g_\theta(h)$ be a generator. Let

$$V(D,G) = \mathbb{E}_{¶}[\log D(X)] + \mathbb{E}_{h\sim p(h)}[\log(1-D(G(h))],$$

where $\mathbb{E}_{¶}$ can be approximated by averaging over the observed examples, and $\mathbb{E}_h$ can be approximated by Monte Carlo average over the faked examples generated by the model.

Thus, we learn D and G by $\min_G \max_D V(D,G)$. $V(D,G)$ is the log-likelihood for D, i.e., the log-probability of the real and faked examples. However, $V(D,G)$ is not a very convincing objective for G. In practice, the training of G is usually modified into maximizing $\mathbb{E}_{h\sim p(h)}[\log D(G(h))]$ to avoid the vanishing gradient problem.

For a given θ, let p_θ be the distribution of $g_\theta(h)$ with $h \sim p(h)$. Assuming a perfect discriminator according to the Bayes rule $D(x) = ¶(x)/(¶(x)+p_\theta(x))$ (assuming equal numbers of real and fake examples). Then θ minimizes Jensen-Shannon

$$\mathrm{JSD}(¶|p_\theta) = \mathrm{KL}(p_\theta|p_{\mathrm{mix}}) + \mathrm{KL}(¶|p_{\mathrm{mix}}),$$

where $p_{\mathrm{mix}} = (¶+p_\theta)/2$. As a result, GAN has mode collapsing behavior.

11.7 Variational auto-encoder (VAE) as alternating projection

The generator model is a decoder. We can pair it with an encoder.

The decoder model is $h \sim p(h) \sim \mathrm{N}(0,I_d)$, and $[x|h] \sim p_\theta(x|h) \sim \mathrm{N}(g_\theta(h),\sigma^2 I)$. It defines a joint distribution $p_\theta(h,x) = p(h)p_\theta(x|h)$.

The encoder model is $[h|x] \sim q_\phi(h|x) \sim \mathrm{N}(\mu_\phi(x),V_\phi(x))$, where V is a diagonal matrix. Together with $¶(x)$, we have a joint distribution $q_\phi(h,x) = ¶(x)q_\phi(h|x)$.

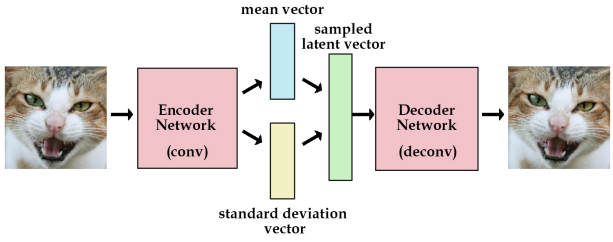


Figure 50: VAE: The decoder is a top-down latent variable model. The encoder approximates the posterior distribution of latent vector given observed image.

We learn both the decoder θ and encoder ϕ by

$$\min_{\theta,\phi} \mathrm{KL}(q_\phi(h,x)|p_\theta(h,x)).$$

The expectation with respect to $q_\phi(h|x)$ can be based on the re-parametrization

$$h = \mu_\phi(x) + V_\phi^{1/2} z,\ z \sim \mathrm{N}(0, I_d),$$

so that the expectation is with respect to z.

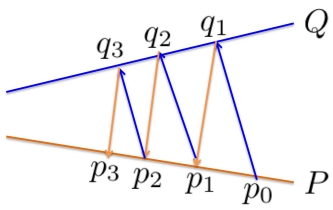


Figure 51: VAE as alternating projection.

Let $Q = \{q_\phi(h,x), \forall\phi\}$ and let $P = \{p_\theta(h,x), \forall\theta\}$. The VAE problem is to $\min_{p\in P, q\in Q} \mathrm{KL}(p|q)$. Starting from p_0, the VAE iterates the following two steps:

(1) $q_{t+1} = \arg\min_{q\in Q} \mathrm{KL}(q|p_t)$.

(2) $p_{t+1} = \arg\min_{p\in P} \mathrm{KL}(q_{t+1}|p)$.

This is alternating projection. The minimization can also be replaced by gradient descent.

The wake-sleep algorithm is the same as VAE in Step (2). However, in Step (1), it is $q_{t+1} = \arg\min_{q\in Q} \mathrm{KL}(p_t|q)$, where we generate dream data from p_t, and learn $q_\phi(h|X)$ from the dream data.

Maximum likelihood vs variational, bias and regularization

The VAE objective function

$$\mathrm{KL}(q_\phi(h,x)|p_\theta(h,x)) = \mathrm{KL}(¶(x)|p_\theta(x)) + \mathrm{KL}(q_\phi(h|x)|p_\theta(h|x)).$$

The first term $\mathrm{KL}(¶(x)|p_\theta(x))$ is the objective function of the maximum likelihood. The VAE divergence is an upper bound of the MLE divergence, which is computationally intractable because $p_\theta(x) = \int p_\theta(h,x)dh$

is an intractable integral. The VAE objective function is tractable as long as the encoder or the inference model $q_\phi(h|x)$ is in closed form.

The accuracy of VAE relative to MLE is determined by the second divergence, which governs the learning of the inference model, i.e., given θ,

$$\hat{\phi}_{\text{VAE}} = \arg\min_\phi \text{KL}(q_\phi(h|x)|p_\theta(h|x)).$$

Since q_ϕ is on the left side of KL-divergence, i.e., the exclusive KL, it tends to chase the major mode of p_θ while ignoring the minor modes.

While $q_\phi(h|x)$ seeks to get closer to $p_\theta(h|x)$ in learning ϕ, in the learning of θ, the model $p_\theta(h,x)$ tends to bias itself from MLE so that $p_\theta(h|x)$ becomes closer to $q_\phi(h|x)$. This bias is actually beneficial to the inference model q_ϕ, and the bias induced by q_ϕ also provides some regularization of the model.

Variational inference vs maximum *a posteriori* (MAP)

The maximum *a posteriori* (MAP) estimate of h is

$$\hat{h}_{\text{MAP}} = \arg\max_h \log p_\theta(h|x).$$

$q_{\hat{\phi}}(h|x)$ is close to the point mass at $\hat{h}_{\text{MAP}}$ because of the aforementioned mode chasing behavior. The difference is that $q_{\hat{\phi}}(h|x)$ is a probability distribution that account for the uncertainty of $p_\theta(h|x)$. In fact, the variational divergence is

$$\text{KL}(q_\phi(h|x)|p_\theta(h|x)) = -H(q_\phi(h|x)) - \mathbb{E}_{q_\phi(h|x)}[\log p_\theta(h|x)],$$

where

$$H(q_\phi(h|x)) = -\mathbb{E}_{q_\phi(h|x)}[\log q_\phi(h|x)]$$

is the entropy of $q_\phi(h|x)$. The variational approximation is to maximize the posterior $\mathbb{E}_{q_\phi(h|x)}[\log p_\theta(h|x)]$ like MAP, but it also seeks to maximize the entropy $H(q_\phi(h|x))$. For variational inference, $H(q_\phi(h|x))$ should be in closed form.

In learning θ, we should not use $\hat{h}_{\text{MAP}}$ to infer h because it does not account for uncertainty. $\hat{h}_{\text{MAP}}$ amounts to overfitting. It explains away too much of x so that there is not much left for θ to explain. As a result, we cannot learn θ accurately, and the learned θ cannot generate realistic x because it can generate realistic x only for those preferred h that are similar to $\hat{h}_{\text{MAP}}$.

Local vs global variational

There is a slight ambiguity in $\text{KL}(q(h|x)|p(h|x))$. We can interpret it as the KL-divergence with x fixed, or with x averaged out by ¶. In the previous subsection, it is the former. In VAE, it is the latter.

In VAE, $\mathbb{E}_¶$ is approximated by averaging over the training data $(x_i, i = 1,...,n)$, so that ϕ is estimated by

$$\min_\phi \frac{1}{n}\sum_{i=1}^n \text{KL}(q_\phi(h_i|x_i)|p_\theta(h_i|x_i)).$$

Here ϕ is shared by all $(x_i, i = 1,...,n)$, and it is the global variational parameter. For each x_i, $q_\phi(h_i|x_i) \sim \text{N}(\mu_\phi(x_i), V_\phi(x_i))$.

We may also estimate (μ_i, V_i) for each x_i by

$$\min_{\mu_i,V_i} \text{KL}(q_{\mu_i,V_i}(h_i|x_i)|p_\theta(h_i|x_i)),$$

where $q_{\mu_i,V_i}(h_i|x_i) \sim \mathrm{N}(\mu_i, V_i)$. (μ_i, V_i) is called the local variational parameter.

For the local variational parameter, we need an iterative inference algorithm to compute it for each x_i. This can be time consuming. For the global parameter, we learn an inference network to map x_i directly to (μ_i, V_i). This amounts to distilling the inference algorithm to the inference network, i.e., the inference network or the inference model is the learned computation.

Traditional variational methods employ local parameters, while VAE uses global parameters by taking advantage of the flexibility of neural nets.

11.8 VAE and EM

In the EM algorithm, we assume $q_\phi(h|x)$ to be infinitely flexible. In the alternating projection of VAE,

(1) $q_{t+1} = \arg\min_{q \in Q} \mathrm{KL}(q|p_t)$, so that $q_{t+1}(h|x) = p_{\theta_t}(h|x)$.

(2) $p_{t+1} = \arg\min_{p \in P} \mathrm{KL}(q_{t+1}|p)$, i.e.,

$$\begin{aligned} \theta_{t+1} &= \arg\min_\theta \mathrm{KL}(¶ p_{\theta_t}(h|x)|p_\theta(h,x)) \\ &= \arg\max_\theta \mathbb{E}_¶ \mathbb{E}_{p_{\theta_t}(h|x)}[\log p_\theta(h,x)]. \end{aligned}$$

The EM algorithm requires that $p_\theta(h|x)$ is in closed form, which means $p_\theta(x)$ should be in closed form.

q is the distribution of data. ¶ is the distribution of observed data x, and $q_\phi(h|x)$ is the multiple imputation of the missing data h. $q_\phi(h,x)$ is the distribution of the complete data (h,x).

p is the distribution of the model. $p(h)p_\theta(x|h) = p_\theta(h,x)$ is the complete-data model, and $p_\theta(x) = \int p_\theta(h,x)dh$ is the observed data model.

11.9 Flow-based model

Similar to generator model, in the flow-based model, we have $h \sim \mathrm{N}(0, I_D)$, and $x = T_\theta(h)$. The difference is that h has the same dimensionality as x, which is D, and T_θ is a one-to-one differentiable transform. Suppose T_θ maps a small region Δh around h to a small region Δx around x. Then

$$q(h)|\Delta h| = p(x)|\Delta x|,$$

where $q(h)$ is the density of h, and $p(x)$ is the density of x. $|\Delta h|$ is the volume of the small region Δh, and Δx is the volume of the small region Δx. To understand this point, you can imagine $q(h)$ as a cloud of points in the space of h, and $p(x)$ is the cloud of points in the space of x. Then the points in Δh are mapped to the points in Δx, and the number of points in Δh is the same as the number of points in Δx.

$$|\Delta x|/|\Delta h| = |\det(T'_\theta(h))|,$$

where $T_\theta(h) = dx/dh^\top$ is the $D \times D$ Jacobian matrix, and $|\det(T')|$ is the absolute value of the determinant of Jacobian. Locally T is a linear transformation, and T' is the matrix of this linear transformation. For any matrix A, it maps the cube $[0,1]^D$ to a parallelogram formed by the column vectors of A. The volume of this parallelogram is the determinant of A. Thus the density $p(x)$ can be obtained in closed form. This is different from the generator model, where the marginal density involves integrating out h.

We can design T to consists of a sequence (or a flow) of simple transformations that are invertible and whose Jacobians can be easily computed. For instance, consider a transformation $x \to y$. We can let $x = (x_1, x_2)$, where x_1 is d_1-dimensional, and x_2 is $d_2 = D - d_1$ dimensional. We can let $y_1 = x_1$, and $y_2 = \mu(x_1) + \sigma(x_1)x_2$, where $\mu(x_1)$ is a d_2 dimensional vector and $\sigma(x_1)$ is a d_2 dimensional diagonal matrix.

The flow-based model can be used as a generative model. It can also be used as an inference model.

12 Representation Learning without Deep Nets

12.1 Factor analysis and generalizations

The generator model in GAN and VAE can be traced back to the factor analysis model.

The model

The model is as follows:

$$x_i = Wh_i + \varepsilon_i,$$

for $i = 1, ..., n$, where W is a $p \times d$ dimensional matrix (p is the dimensionality of x_i and d is the dimensionality of h_i), and ε_i is a p-dimensional residual vector. The following are the interpretations of W:

(1) Loading matrix: Let $W = (w_{jk})_{p\times d}$. $x_{ij} \approx \sum_{k=1}^{d} w_{jk} h_{ik}$, i.e., each component of x_i, x_{ij}, is a linear combination of the latent factors. w_{jk} is the loading weight of factor k on variable j.

(2) Basis vectors: Let $W = (W_k, k = 1, ..., d)$, where W_k is the k-th column of W. $x_i \approx \sum_{k=1}^{d} h_{ik} W_k$, i.e., x_i is a linear superposition of the basis vectors (W_k), where h_{ik} are the coefficients.

(3) Matrix factorization: $(x_1, ..., x_n) \approx W(h_1, ..., h_n)$, where the $p \times n$ matrix $(x_1, ..., x_n)$ is factorized into the $p \times d$ matrix W and the $d \times n$ matrix $(h_1, ..., h_n)$.

In factor analysis, we have $h_i \sim \mathrm{N}(0, I_d)$, $x_i = Wh_i + \varepsilon_i$, $\varepsilon_i \sim \mathrm{N}(0, \sigma^2 I_p)$, and ε_i is independent of h_i. The dimensionality of h_i, which is d, is smaller than the dimensionality of x_i, which is p. The factor analysis is very similar to the principal component analysis (PCA), which is a popular tool for dimension reduction. The difference is that in factor analysis, the column vectors of W do not need to be orthogonal to each other.

The factor analysis model originated from psychology, where x_i consists of the test scores of student i on p subjects. h_i consists of the verbal intelligence and the analytical intelligence of student i ($d = 2$). Another example is the decathlon competition, where x_i consists of the scores of athlete i on $p = 10$ sports, and h_i consists of athlete i's speed, strength and endurance ($d = 3$).

Generalizing prior assumption: ICA, sparse coding, NMF, recommender, K-means

In independent component analysis, we assume $p = d$, and $h_k \sim p_k$, which is a heavy tailed distribution. It is also called blind source separation, where h_k are the sources mixed by W.

In sparse coding, we assume $d > p > d_0$, where d_0 is the number of h_k that are non-zero. $W = (W_k, k = 1, ..., d)$ forms a dictionary, and the sparse h selects the most meaningful words from this dictionary.

In non-negative matrix factorization, we assume $h_k \geq 0$, to emphasize the fact that the parts should add positively.

In a recommender system, $(x_1, ..., x_n) \approx W(h_1, ..., h_n)$, where x_{ij} is user i's rating of movie j, and $x_{ij} = \langle w_j, h_i \rangle$, where w_j is the j-row of W. We can interpret h_i as user i's desires in various aspects, and w_j as movie j's desirabilities in the corresponding aspects. x_i are incomplete, but we can complete x_i by learning h_i and w_j from the observed data, so that we can recommend user i the movies that may receive high ratings from him or her.

If h_i is a one-hot vector, then $x_i = Wh_i + \varepsilon_i$ means that $x_i = W_k + \varepsilon_i$, where k is the element of h_i that is 1, and W_k is the k-th column of W. This is the K-means model.

The sparse vector lies in between one-hot vector and dense vector.

Generalizing linear transformation to non-linear transformation

The factor analysis model is $x = Wh + \varepsilon$ (here we drop the subscript i for convenience). We can generalize it to $x = g_\theta(h) + \varepsilon$, where g_θ can be parametrized by a deep network. This is the decoder model or the

generator model in GAN and VAE.

12.2 K-means as one-hot vector representation

Let $(x_i, i = 1, ..., n)$ be the observed data. The K-means method partition the data into K different clusters. Each cluster has a mean μ_k, $k = 1, ..., K$. The K means algorithm iterates the following two steps:

(1) Assignment. We assign each x_i to a cluster k by minimizing $|x_i - \mu_k|^2$ over k.

(2) Update. For each cluster k, we compute μ_k as the mean of the examples x_i that belong to cluster k.

K-means can be considered a special case of encoding, where for each x_i, the corresponding h_i is one hot, i.e., only one component of h_i is 1, and the other components are all 0.

Word2vec is the opposite of K-means, where we encode one-hot vector (word) to a dense vector.

12.3 Spectral embedding and clustering

The spectral embedding and clustering method starts from an adjacency matrix

$$A_{ij} = \exp\left(-\frac{1}{2\sigma^2}|x_i - x_j|^2\right),$$

which measures the similarity between i and j. A is an $n \times n$ matrix with elements (A_{ij}), where we assume $A_{ii} = 0$ on the diagonal. A_{ij} does not care about distant relationship because it is practically zero if x_i is far away from x_j.

The spectral embedding is to find h_i for each x_i, by minimizing

$$\sum_{i \neq j} A_{ij}|h_i - h_j|^2 = 2\text{trace}(H^\top (D - A)H),$$

where $D_i = \sum_j A_{ij}$, and D is the diagonal matrix whose i-th diagonal element is D_i. $H^\top = (h_i, i = 1, ..., n)$. trace is the sum of the diagonal elements. H can be obtained by the smallest d eigen vectors of $D - A$, where d is the dimensionality of h_i.

The basic idea is that the embedding (h_i) should capture the nearby relations, while ignoring distant relations. If A_{ij} is big, we want h_i to be close to h_j.

The spectral clustering method is similar to spectral embedding. It computes the Laplacian

$$L = D^{-1/2} A D^{-1/2},$$

and finds $H^\top = (h_i, i = 1, ..., n)$ as the largest d eigen vectors of L. Then perform K-means on $(h_i, i = 1, ..., n)$. Such clustering pays more attention to nearby relations, while ignoring distant relations, because the definition of A. $L_{ij} = A_{ij}/\sqrt{D_i D_j}$, where we normalize the adjacency by D, i.e., a close friendship to a popular person with a lot of good friends is not really as close as it appears to be.

12.4 Multi-dimensional scaling as mirroring

The earliest embedding method is multi-dimensional scaling, which seeks to preserve the Euclidean distance by

$$\min_{\{h_i\}} \sum_{ij} (|x_i - x_j| - |h_i - h_j|)^2.$$

Later embedding methods focus more on preserving the adjacencies of nearby points, i.e., the embedding lets friends stay close, but is not concerned with relations between strangers.

12.5 t-Stochastic neighborhood embedding (tSNE) as energy-based model

tSNE seeks to mirror the friendship relationships. For each example i, we view other examples from its perspective, and define

$$p_{j|i} = \exp\left(-\frac{1}{2\sigma_i^2}|x_i - x_j|^2\right) / \sum_{k\neq i} \exp\left(-\frac{1}{2\sigma_i^2}|x_i - x_k|^2\right).$$

We may choose σ_i so that it distinguishes $\{x_j, j \neq i\}$ in terms of their relationships to x_i.

For $(h_i, i = 1, ..., n)$ in the embedded space, we mirror $(x_i, i = 1, ..., n)$ by defining

$$q_{j|i} = \exp\left(-\frac{1}{2\sigma_i^2}|h_i - h_j|^2\right) / \sum_{k\neq i} \exp\left(-\frac{1}{2\sigma_i^2}|h_i - h_k|^2\right).$$

Both p and q are conditional distributions of j given i, i.e., if i feels lonely and wants to call someone, then j comes to his or her mind with probability $p_{j|i}$ or $q_{j|i}$.

We may optimize the embedding by

$$\min_{\{h_i\}} \mathrm{KL}(p|q) = \min_{\{h_i\}} \sum_{ij} (\log p_{j|i} - \log q_{j|i}).$$

In tSNE, we replace the normal density in $q_{j|i}$ by the t distribution or the Cauchy distribution, so that h_i can be more spread out. We also symmetrize $p_{j|i}$ by defining $p_{ij} = (p_{j|i} + p_{i|j})/2n$, and we define

$$q_{ij} = (1 + |h_i - h_j|^2)^{-1} / \sum_{k\neq l} (1 + |h_k - h_j|^2)^{-1}.$$

We then find $h = \{h_i\}$ by minimizing

$$L(h) = \mathrm{KL}(p|q) = \sum_{ij} p_{ij}(\log p_{ij} - \log q_{ij}).$$

Now p and q are joint distributions of (i, j), i.e., if we want to sample a pair of friends, we get (i, j) with probability p_{ij} or q_{ij}.

$L(h)$ is actually the negative log-likelihood, and h is the parameter. p serves as the data distribution, and q is the model distribution. q is actually an energy-based model of the form

$$q_\theta(x) = \frac{1}{Z(\theta)} \exp[f_\theta(x)],$$

where $\theta = h$, and $x = (i, j)$, and

$$f_\theta(x) = -\log(1 + |h_i - h_j|^2),$$

It can be shown that

$$-\frac{\partial}{\partial \theta} \mathrm{KL}(p|q_\theta) = \mathbb{E}_p\left[\frac{\partial}{\partial \theta} f_\theta(x)\right] - \mathbb{E}_q\left[\frac{\partial}{\partial \theta} f_\theta(x)\right].$$

For tSNE,

$$\frac{\partial f}{\partial h_i} = -2(1 + |h_i - h_j|^2)^{-1}(h_i - h_j).$$

The gradient is

$$\frac{\partial L}{\partial h_i} = 4 \sum_{j\neq i} (p_{ij} - q_{ij})(1 + |h_i - h_j|^2)^{-1}(h_i - h_j).$$

The factor 4 is due to the symmetry between (i, j) and (j, i). We can obtain (h_i) by gradient descent.

The basic idea is that (p_{ij}) captures the nearby relations among (x_i), and (q_{ij}) captures the nearby relations among (h_i), while allowing (h_i) to be scattered out according to the heavy-tailed t distribution. We want to find (h_i) so that Q approaches P.

12.6 Local linear embedding (LLE)

LLE seeks to mirror the local linear relationship.

First, for each i, we find its neighbors j, and we denote $j \sim i$.

Then for each i, we extract the local linear relationship by minimizing

$$\min_{(w_{ij})} \left| x_i - \sum_{j \sim i} w_{ij} x_j \right|^2.$$

We impose the constraint $\sum_j w_{ij} = 1$, so that the learned w_{ij} is invariant with the shift $x_i \leftarrow x_i + c$ for a constant c.

Then we find $\{h_i\}$ by minimizing

$$\min_{(h_i)} \sum_i \left| h_i - \sum_{j \sim i} w_{ij} h_j \right|^2,$$

which can be minimized by eigen computation.

Unlike tSNE, where the mirroring is in explicitly defined friendship relationship, in LLE, the mirroring is in the form of minimization.

12.7 Topic model and latent Dirichelet allocation (LDA)

The topic model is used to analyze documents, e.g., news reports or medical papers, to identify the topics and their key words. Each report may contain multiple topics, and each topic generates a distribution of words. Suppose there are K topics. For a document, let $\rho = (\rho_k, k = 1, ..., K)$ be the probability distribution of topics. Even though K may be large, only a small number of topics have big probabilities. Suppose $f_k(x)$ be the distribution of word x in each topic. Even though the number of words is large, only a small number of words have big probabilities. Then the overall distribution

$$f(x) = \sum_k \rho_k f_k(x),$$

which can be written in the matrix factorization form.

In latent Dirichlet allocation, we assume that ρ follows a Dirichelet distribution. Given ρ, the topic follows a multinomial distribution. Given the topic k, the words follow a multinomial distribution according to f_k. We assume that the words are unordered in the document, i.e., each document is a bag of words. The model can be written as $\rho \to h \to x$, where h is the one-hot hidden vector for topic, and x is the one-hot vector for word. There are two layers of latent vectors that generate each word in each document, namely ρ at the document level, and h at the word level.

The model can be learned by local variational method, with variational inference models $q(h)$ and $q(\rho)$, by minimizing $\mathrm{KL}(q(h)q(\rho)|p(h,\rho|x)$, where we assume the inference distributions $q(h)$ and $q(\rho)$ to be independent. This leads to the mean field method where we update the variational parameters for $q(h)$ and $q(\rho)$ alternatively. It is similar to the Gibbs sampler for sampling $p(h,\rho|x)$ by alternating the sampling of $p(h|\rho,x)$ and $p(\rho|h,x)$. The sampling in MCMC is replaced by optimizing the variational parameters. For each word x, $q(h)$ tells us the topic. For each document, $q(\rho)$ tells us the distribution of the topics.

13 Reinforcement Learning

The reinforcement learning lies between supervised learning and unsupervised learning. In supervised learning, the representation $h(x)$ is used to predict y. In unsupervised learning, the representation h is used to

encode and mirror x. In reinforcement learning, the representation h is used to predict the action or value. In imitation learning, the action is observed, so the learning is supervised. In reinforcement learning, the action is not observed, and the training signal is in the form of cumulated reward or value.

13.1 Alpha Go

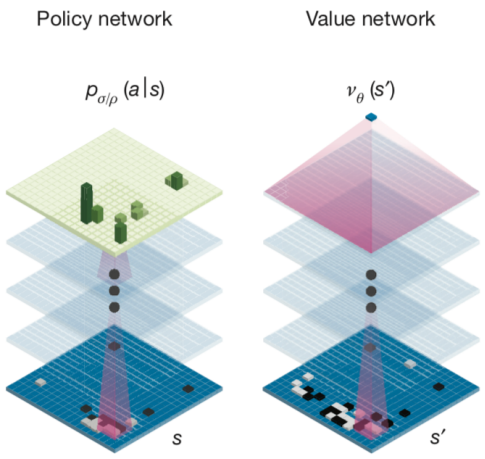


Figure 52: Policy network and value network. The input is a 19×19 image. The policy network is a classification network. The value network is a regression network.

Alpha Go is a good starting point to learn reinforcement learning. Let s be the current state, which is a 19×19 image. Let a be the action, i.e., where to place the stone. There are 19×19 choices (although some are forbidden by the rule). We want to decide the action a.

Game tree and minimax solution

Starting from the initial state, i.e., an empty board, the black and white players alternatively place black and white stones on the board. This generates a game tree, whose breadth and depth are both big. We can find minimax solution by back-propagation from the leaf nodes, so that each player chooses the moves that maximizes its final result, i.e., a win in this case.

For Go, it is impractical to go through the whole game tree. We may use Monte Carlo tree search (MCTS) to go over the promising moves. But pure MCTS is still quite challenging to implement. We use policy network to reduce the breadth of the tree search, and the value network to reduce the depth of tree search.

Supervised learning or behavior cloning

We can learn a policy network $p_\sigma(a|s)$, with parameter σ (stands for supervised) from the data $\{(s,a)\}$ collected from human players. See Figure 52 for an illustration of the policy network. This is a classification problem and is a supervised learning problem. It is also called behavior cloning. The learning rule is

$$\Delta\sigma \propto \frac{\partial}{\partial\sigma} \log p_\sigma(a|s),$$

which maximizes the log-likelihood $\log p_\sigma(a|s)$ over σ.

We can also learn a simpler roll out network to be used in Monte Carlo tree search.

Reinforcement learning by policy gradient

After learning $p_\sigma(a|s)$, we can learn another policy network $p_\rho(a|s)$ by reinforcement learning. Starting from $\rho = \sigma$, we let p_ρ play against p_σ, until the end. Let $z \in \{+1, -1\}$ be whether ρ wins the game. We update ρ by

$$\Delta\rho \propto \frac{\partial}{\partial\rho} \log p_\rho(a|s) z.$$

The above updating rule is called policy gradient. It is similar to the above maximum likelihood updating rule, except for the following two aspects: (1) the derivative of the log-likelihood is weighted by the reward z. (2) the action a is generated by the current policy instead of human expert.

The above updating rule maximizes $\mathbb{E}_\rho[z]$. It is called REINFORCE algorithm. We will justify this algorithm later on.

Value network

After learning $p_\rho(a|s)$, we can let ρ play with itself from a random starting state s, until we reach the end and get z. We then update the value network $v_\theta(s)$ by

$$\Delta\theta \propto \frac{\partial}{\partial\theta} [z - v_\theta(s)]^2.$$

See Figure 52 for an illustration of the value network.

The learned v can also be used as a baseline for policy gradient

$$\Delta\rho \propto \frac{\partial}{\partial\rho} \log p_\rho(a|s)[z - v_\theta(s)],$$

where $v_\theta(s)$ serves to reduce the variance of the gradient. Here v_θ is the critic, and p_ρ is the actor.

Monte Carlo tree search

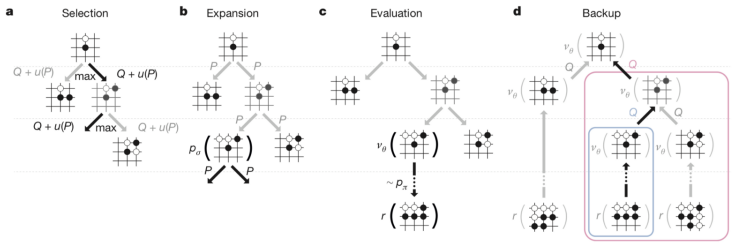


Figure 53: Monte Carlo tree search grows a tree by repeating the following four steps: selection, expansion, evaluation, and backup.

After learning the policy networks p_σ, p_ρ, and value network v_θ, we can use either the policy network or the value network to play the game. But this will not work well. Instead we use them to help us look ahead and plan the next action using Monte Carlo tree search (MCTS). Specifically, we wan to estimate $Q(s,a)$, the value of action a at state s.

Treating the current state as the root, MCTS grows a tree by repeating the following four steps: selection, expansion, evaluation, and backup. For each node s of the tree, and each action a from this node, we record the number of visits $N(s,a)$ and the action value $Q(s,a)$. During each pass of MCTS, starting from the root

state, we go down the tree until we get to a leaf node. At each non-leaf node s, we choose an action a that balances $Q(s,a)$ and $N(s,a)$. We want to choose a with a high $Q(s,a)$ for exploitation, meanwhile we also want to choose a with low $N(s,a)$ for exploration. When we come to a leaf node, we expand the tree using all the possible moves from this node. Then we choose an expanded node, and use a roll out policy to play the game until the end. Finally we backup the roll out result. For all the branches (s,a) we go through in this pass of MCTS, we increase the visit count $N(s,a)$ by 1, and increase or decrease the total value of (s,a) by 1 according to the roll out result. $Q(s,a)$ is then updated as the current total value divided by the current visit count. MCTS is expected to converge to the minimax solution to the game.

The policy network can help guide the selection step. For each node s, we select the action according to $Q(s,a)+Cp_\sigma(a|s)/\sqrt{N(s,a)}$ where the constant C balances the exploration and exploitation. We use the supervised p_σ instead of reinforcement learning p_ρ, because p_σ is more diverse.

The value network can help avoid the reliance on roll out. For an expanded node s, instead of rolling out to the end, and then backup, we can simply backup $v_\theta(s)$ by adding the total value of a traveled branch (s,a) the value $v_\theta(s)$. The Alpha Go uses a linear combination of roll out result from s and $v_\theta(s)$. Since s is closer to the end of the game than the root node, the value $v_\theta(s)$ can be a more precise estimate than the value at the root node.

After many passes of MCTS, at the current root node s, we choose a with the maximal $Q(s,a)$. We can also choose a with the maximum $N(s,a)$.

The MCTS is a planning process. Its purpose is to select the next move a from the current root node s. After we make the move a in real game play, we can then discard the tree. When we need to choose the next move in real game play, we start to grow another tree.

13.2 Alpha Go Zero

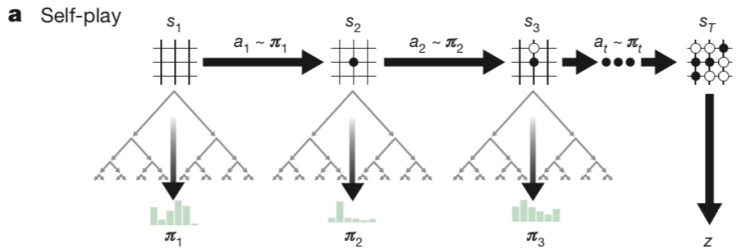


Figure 54: Alpha Go Zero. In self-play, we use MCTS to plan each move, where MCTS is guided by policy and value networks. After the game is over, the game result is used to train the policy and value networks.

In Alpha Go, after learning the policy and value networks, we will not update it, and they are used for MCTS for playing the game. This is a big waste, because when we use MCTS to play the game, the results of the game play should be used to further update the policy and value networks. In fact, the policy and value networks can be learned solely from the games where the moves are planned by MCTS.

In Alpha Go Zero, the learning of p_σ, p_ρ and v_θ before MCTS is discarded. Instead, p and v are learned from the results of the self-play, and p and v in turn guides the MCTS in planning each move of the self-play. Specifically, we iterate the following two steps:

(1) Given p and v, play a game where each move is planned by MCTS. Specifically, at each state s of the game, we grow a tree in order to evaluate $N(s,a)$ and $Q(s,a)$ (in fact, $N(s,a)$ and $Q(s,a)$ are computed for all the nodes in the tree). We let $\pi(a|s) \propto N(s,a)^{1/\tau}$, where τ is the temperature parameter. We the discard the tree and use $\pi(a|s)$ to select the move a to play the game. We play the game until the end.

(2) We use $\pi(a|s)$ to train the policy $p(a|s)$. We use the end result $z \in \{+1,-1\}$ to train the value network $v(s)$ for all the states of the game.

In Alpha Go Zero, the policy and value networks share the same body, which consists of many residual blocks. Each network has a head. Such a design makes sense because the two networks are consistent with each other.

In Alpha Go and Zero, the policy network is like impulse or habit, and the value network is like gut feeling. They are like emotions. The Monte Carlo tree search is to think through and plan. The reason we have consciousness is that we need to know our emotions (as well as perceptions, i.e., what we see and hear) in order to plan our actions.

13.3 Atari by Q learning

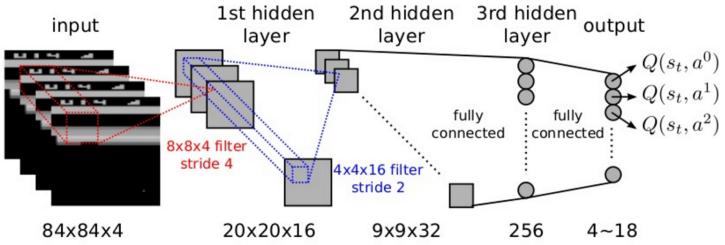


Figure 55: Atari. The action value network $Q_\theta(s,a)$.

Atari is a video game. It is different from Go in the following two aspects. (1) In Atari we have immediate rewards in terms of points earned at each step, while in Go the reward is delayed to the end. (2) In Atari, a player does not know the dynamics or the detailed rules that drive the game, although the player can play out the game to observe the change of the state and the reward earned, while in Go, a player knows the rule of game.

We can learn to play Atari by Q learning, by learning the Q function $Q_\theta(s,a)$ by playing out the game. $Q(s,a)$ is the cumulative reward to go, i.e., the total increase of score if we take action a at the current state s, and then play out the game to the best of our ability. Suppose the cumulative reward to go is R. Then we can update θ by

$$\Delta\theta \propto -\frac{\partial}{\partial\theta}(R - Q_\theta(s,a))^2..$$

In order to obtain R, we need to play out the game by taking the best action at each step. However, this will be time consuming. After taking the action a, we get the immediate reward r, and get to the next state s'. Then we estimate R by $r + \gamma \max'_a Q_\theta(s',a')$, where $\max_{a'} Q_\theta(s',a')$ is the cumulative reward to go at state s' if we do our best. γ is the discount factor, usually taken to be close to 1. Then we can use the estimated R to update θ.

The above method of predicting R is referred to as a bootstrap method, where we use the current model $Q_\theta(s,a)$ to help predict R. The difference between the predicted R and $Q_\theta(s,a)$ is called temporal difference error.

At each s, how do we take the action a? The greedy policy is to take a that maximize $Q_\theta(s,a)$. Usually we use the ε-greedy policy, where with ε probability, we take a random action, and with probability $1-\varepsilon$, we take the greedy action.

Q-learning is a model free method, where we do not need to know the dynamics $s_{t+1} = f(s_t,a_t)$ and the reward function $r_t = r(s_t,a_t)$, i.e., we do not know $f()$ and $r()$, but at state s_t, we can try the action a_t to observe s_{t+1} and r_t.

13.4 Markov decision process (MDP)

Model

The model has two parts.

Part 1: The dynamic model is $p(s_{t+1}|s_t, a_t)$. In the deterministic case, $s_{t+1} = f(s_t, a_t)$.

Part 2: The reward model is $p(r_t|s_t, a_t, s_{t+1})$. In the deterministic case, $r_t = r(s_t, a_t, s_{t+1})$, as well as $\tilde{r}(s_T)$ if there is a terminating state.

The dynamics model and the reward model are Markovian, in that we do not need to know the past history before (s_t, a_t).

The trajectory is $\tau = ((s_t, a_t), t = 0, ...)$.

The return is

$$R(\tau) = \sum_{t=0}^{T-1} r(s_t, a_t, s_{t+1}) + \tilde{r}(s_T).$$

For infinite horizon (i.e., no terminal state),

$$R = \sum_{t=0}^{\infty} \gamma^t r(s_t, a_t, s_{t+1}),$$

where γ is the discount factor. It is introduced so that R is finite. It also makes practical sense because we are more certain about near future and want to collect rewards sooner.

Policy and value

A policy is $\pi(a_t|s_t)$. Again it is Markovian. For deterministic policy, $a_t = \pi(s_t)$.

Together with dynamics, the distribution of a trajectory τ is

$$p(\tau) = \prod_{t=0}^{T-1} \pi(a_t|s_t) p(s_{t+1}|s_t, a_t).$$

Define the return to go at time t to be

$$R_t(\tau) = \sum_{k=0}^{\infty} \gamma^k r_{t+k},$$

where $r_t = r(s_t, a_t, s_{t+1})$.

The state value is

$$v^\pi(s) = \mathbb{E}_{p(\tau)}[R_t \mid s_t = s],$$

The state-action value (or quality) is

$$Q^\pi(s, a) = \mathbb{E}_{p(\tau)}[R_t \mid s_t = s, a_t = a].$$

Usually we write $\mathbb{E}_{p(\tau)} = \mathbb{E}_\pi$ because the distribution of trajectory depends on π, with the dynamics fixed.

The optimal value is

$$v^*(s) = \max_\pi v^\pi(s).$$

$$Q^*(s, a) = \max_\pi Q^\pi(s, a).$$

A policy π^* is optimal if $v^{\pi*}(s) \geq v^\pi(s)$ for all π and all s.

For MDP, one can always find a deterministic policy to be optimal.

13.5 Model-based planning: play out in imagination

We can learn the dynamic model $p(s_{t+1}|s_t,a_t)$ and the reward model $p(r_t|s_t,a_t,s_{t+1})$, and plan the optimal sequence of actions $(a_t,t=0,...,T-1)$ to maximize the total reward. This is model-based planning or optimal control. It is usually solved by dynamic programming. The value and policy are involved in the calculation of dynamic programming. The Monte Carlo tree search is model-based planning.

In the computation of model-based planning, we may generate trajectories in our mind by imagining playing out a sequence of actions. For instance, the Monte Carlo tree search is computed in the mind, not in real play. After Monte Carlo tree search, we only play the first move in the real game.

13.6 Model-free learning: play out in real life

In model-free learning, we do not learn the dynamic and the reward models. Instead we take actions a_t at s_t in real life, and observe s_{t+1} and collect r_t.

In value-based learning such as Q-learning, we estimate $Q(s,a)$, and then use greedy policy that maximize $Q(s,a)$ in testing.

In policy-based learning such as policy gradient, we roll out the trajectory and update the policy.

Model-based learning requires a small amount of data to learn the dynamic and reward models. Planning does not require more data. The policy and value are computed from simulated data in the mind.

Model-free learning requires more data to estimate the value or policy from real life experience. We need to repeatedly play out the actions in real life and observe the trajectories and rewards.

13.7 Temporal difference bootstrap

To reduce the data requirement in model-free learning, we do not need to play out the actions until the end. In order to obtain the total return of the whole trajectory, we may play out for a few steps, and then stop early, and use the current value function to estimate the rest of the return. The value function is expected to be more accurate as we are closer to the end. This is the temporal difference bootstrap estimate. This is a bit like imaginary playing out in model-based method, i.e., at the early stop, we ask ourselves: if we continue to play until the end, what would be the return?

13.8 Q learning

The Q-learning is model-free value-based learning using the bootstrap idea. Let $Q_\theta(s,a)$ be the value network. Let s' be the state observed after playing a at state s. Let R be the return if we play out as best as we can. Then we can update θ by

$$\Delta\theta \propto \frac{\partial}{\partial\theta}[R-Q_\theta(s,a)]^2.$$

We use the bootstrap method to get

$$R = r+\gamma\max_{a'} Q_\theta(s',a').$$

At s, how should we choose a? Usually we use ε-greedy policy, i.e., with probability $1-\varepsilon$, we take $a=\arg\max_a Q_\theta(s,a)$, and with probability ε, we take random action. This is the off-policy approach, i.e., in the learning stage, we do not take the greedy policy according to Q, because we need to explore other actions. After learning, we can switch to the greedy policy in testing.

Q-learning is essentially dynamic programming, except that r is observed in real play, i.e., it is empirical version of dynamic programming.

13.9 REINFORCE, MLE, re-parametrization

For a policy network $\pi_\theta(a|s)$, if we play out this policy in real life, we can obtain a trajectory τ, and the distribution of τ is

$$p_\theta(\tau) = \prod_t \pi(a_t|s_t)p(s_{t+1}|s_t,a_t).$$

The return is $R(\tau) = \sum_t \gamma^t r_t$, where $r_t = r(s_t,a_t,s_{t+1})$.

We want to find θ to maximize $\mathbb{E}_{p(\tau)}[R(\tau)]$.

REINFORCE

The gradient is

$$\begin{aligned}
\frac{\partial}{\partial\theta}\mathbb{E}_{p_\theta(\tau)}[R(\tau)] &= \int R(\tau)\frac{\partial}{\partial\theta}p_\theta(\tau)d\tau \\
&= \int \left[R(\tau)\frac{\partial}{\partial\theta}\log p_\theta(\tau)\right] p_\theta(\tau)d\tau \\
&= \mathbb{E}_{p_\theta(\tau)}\left[R(\tau)\frac{\partial}{\partial\theta}\log p_\theta(\tau)\right].
\end{aligned}$$

The stochastic gradient (or empirical gradient) is to sample τ by playing out the policy π_θ, and update θ by

$$\Delta\theta \propto R(\tau)\frac{\partial}{\partial\theta}\log p_\theta(\tau).$$

This is the REINFORCE algorithm.

REINFOCE vs maximum likelihood

If we observe an expert who plays out his policy to generate a trajectory, we can learn θ by maximum likelihood, and the stochastic gradient is

$$\Delta\theta \propto \frac{\partial}{\partial\theta}\log p_\theta(\tau).$$

The differences between REINFORCE and maximum likelihood are as follows.

(1) In REINFORCE, τ is generated by the self policy. In MLE, τ is generated by the expert.

(2) In REINFORCE, there is $R(\tau)$ in the gradient. In MLE, there is no $R(\tau)$ term.

If we set $R(\tau) = c$, a constant, then

$$\mathbb{E}_{p_\theta(\tau)}\left[c\frac{\partial}{\partial\theta}\log p_\theta(\tau)\right] = \frac{\partial}{\partial\theta}\mathbb{E}_{p_\theta(\tau)}[c] = 0.$$

This means one cannot learn from himself.

REINFORCE vs re-parametrization

In VAE, the inference model $q_\phi(h|x)$ is like a policy, where x is the state, and h is the action. We want to maximize $\mathbb{E}_{q_\phi(h|x)}[R(h)]$ for a function $R(h)$. We can use REINFORCE algorithm to optimize ϕ. In the parametrization trick, we write $h = \mu_\phi(x) + V_\phi(x)^{1/2}e$, where $e \sim \mathrm{N}(0, I_d)$, so that $\mathbb{E}_{q_\phi(h|x)}[R(h)] = \mathbb{E}_e[R(\mu_\phi(x) + V_\phi^{1/2}e)]$, which can be maximized by gradient ascent.

Mapping the situation to reinforcement learning, if we can reparametrize the policy $p_\theta(\tau)$ by $\tau = \pi_\theta(e)$, where $e \sim p(e)$ independent of θ, and π is now a deterministic function, then we can maximize $\mathbb{E}_e[R(\pi_\theta(e)]$, whose gradient is $\mathbb{E}_e\left[\frac{\partial}{\partial\theta}R(\pi_\theta(e))\right]$. This is the same as the REINFORCE gradient $\mathbb{E}_{p_\theta(\tau)}\left[R(\tau)\frac{\partial}{\partial\theta}\log p_\theta(\tau)\right]$. However, $\mathrm{Var}_e\left[\frac{\partial}{\partial\theta}R(\pi_\theta(e))\right]$ can be much smaller than $\mathrm{Var}_{p_\theta(\tau)}\left[R(\tau)\frac{\partial}{\partial\theta}\log p_\theta(\tau)\right]$. In the former, we directly improve each sampled trajectory $\tau = \pi_\theta(e)$. In the latter, however, we improve the probability of those sampled trajectories that have large R. For instance, suppose you want to learn how to shoot the basket ball. In the reparametrization method, you try to improve each attempt by aiming carefully. In REINFORCE, you randomly shoot the ball blindly, hoping that you may get lucky occasionally, and then you learn from these lucky attempts. As a result, you need to try a lot of times.

13.10 Policy gradient: actor and critic

Independent of dynamics

The gradient is

$$\begin{aligned}
\frac{\partial}{\partial\theta}\mathbb{E}_{p_\theta(\tau)}[R(\tau)] &= \mathbb{E}_{p_\theta(\tau)}\left[R(\tau)\frac{\partial}{\partial\theta}\log p_\theta(\tau)\right] \\
&= \mathbb{E}_{p_\theta(\tau)}\left[\left(\sum_{t=0}^{\infty} r_t\right)\frac{\partial}{\partial\theta}\log\left(\prod_{t=0}^{\infty}\pi_\theta(a_t|s_t)p(s_{t+1}\mid s_t,a_t)\right)\right] \\
&= \mathbb{E}_\theta\left[\left(\sum_{t=0}^{\infty} r_t\right)\left(\sum_{t=0}^{\infty}\frac{\partial}{\partial\theta}\log\pi_\theta(a_t|s_t)\right)\right] \\
&= \mathbb{E}_\theta\left[\sum_{t=0}^{\infty}\left(\frac{\partial}{\partial\theta}\log\pi_\theta(a_t|s_t)\sum_{t'=0}^{\infty} r_{t'}\right)\right],
\end{aligned}$$

where we ignore the discount factor for simplicity.

Independent of past

$$\begin{aligned}
\frac{\partial}{\partial\theta}\mathbb{E}_\theta[R] &= \mathbb{E}_\theta\left[\sum_{t=0}^{\infty}\left(\frac{\partial}{\partial\theta}\log\pi_\theta(a_t|s_t)\sum_{t'=0}^{\infty} r_{t'}\right)\right] \\
&= \sum_{t=0}^{\infty}\mathbb{E}_\theta\left[\left(\frac{\partial}{\partial\theta}\log\pi_\theta(a_t|s_t)\left(\sum_{t'=0}^{t-1} r_{t'} + \sum_{t'=t}^{\infty} r_{t'}\right)\right)\right] \\
&= \sum_{t=0}^{\infty}\mathbb{E}_\theta\left[\frac{\partial}{\partial\theta}\log\pi_\theta(a_t|s_t)(R_t + C)\right] \qquad \text{where } C = \sum_{t'=0}^{t-1} r_{t'} \\
&= \sum_{t=0}^{\infty}\mathbb{E}_\theta\left[\frac{\partial}{\partial\theta}\log\pi_\theta(a_t|s_t)R_t\right] + \sum_{a_t}\pi_\theta(a_t|s_t)\left[\frac{\partial}{\partial\theta}\log\pi_\theta(a_t|s_t)C\right] \\
&= \sum_{t=0}^{\infty}\mathbb{E}_\theta\left[\frac{\partial}{\partial\theta}\log\pi_\theta(a_t|s_t)R_t\right] + \frac{\partial}{\partial\theta}\left[C\sum_{a_t}\pi_\theta(a_t|s_t)\right] \\
&= \sum_{t=0}^{\infty}\mathbb{E}_\theta\left[\frac{\partial}{\partial\theta}\log\pi_\theta(a_t|s_t)R_t\right] + \frac{\partial}{\partial\theta}C \\
&= \sum_{t=0}^{\infty}\mathbb{E}_\theta\left[\frac{\partial}{\partial\theta}\log\pi_\theta(a_t|s_t)R_t\right]
\end{aligned}$$

where

$$R_t = \sum_{t'=t}^{\infty} r_{t'}$$

is the reward to go after action a_t, whereas $\sum_{t'=0}^{t-1} r_{t'}$ is constant relative to a_t, because a_t can only cause change in the future, but it cannot change the past. Therefore we can remove it as a baseline.

Advantage

$$\frac{\partial}{\partial \theta}\mathbb{E}_\theta[R] \quad = \quad \sum_{t=0}^{\infty} \mathbb{E}_\theta\left[\frac{\partial}{\partial \theta}\log \pi_\theta(a_t|s_t)(R_t - V(s_t))\right]$$

for any baseline $V(s_t)$, which cannot be changed by a_t. Here we take

$$V(s_t) = V^\pi(s_t) = \mathbb{E}[R_t \mid s_t],$$

which is the value of s_t, averaged over all possible actions $a_t \sim \pi(a|s_t)$.

$$A_t = R_t - V(s_t)$$

is called the advantage of taking action a_t relative to other possible actions at s_t. Subtracting $V(s_t)$ as a baseline helps reduce the variance of the gradient. Again consider learning a policy to shoot the basket ball. If you are very close to the basket and you hit the basket, it is not a big deal. But if you are beyond the three-pointer arc and you hit the basket, you should learn from such attempts.

When computing the gradient, we fix π at the current policy.

We can learn θ by stochastic gradient

$$\Delta\theta \propto \sum_{t=0}^{\infty}\left[\frac{\partial}{\partial \theta}\log \pi_\theta(a_t|s_t)(R_t - V_\alpha(s_t))\right]$$

where we run the policy π_θ to the end. We also need to learn a value network $V_\alpha(s)$ to approximate $V^\pi(s)$. The above update pushes the policy to favor the above average actions.

The policy $\pi_\theta(a|s)$ is called an actor. The value $V_\alpha(s)$ is called a critic.

Temporal difference

We may also estimate R_t based on

$$V^\pi(s_t) = \mathbb{E}[R_t] = \mathbb{E}\left[\sum_{k=0}^{n} r_{t+k} + V^\pi(s_{t+n+1})\right].$$

Thus we can estimate R_t without running the policy to the very end, and we estimate R_t by

$$\hat{R}_t = \sum_{k=0}^{n} r_{t+k} + V_\alpha(s_{t+n+1}),$$

which consists of two parts:

(1) Monte Carlo unrolling in real life: run the policy π_θ for n steps to accumulate $\sum_{k=0}^{n} r_{t+k}$.

(2) Bootstrapping in imagination: prediction the rest of the accumulated reward by the current value network $V_\alpha(s_{t+n+1})$.

This is a temporal difference scheme.

Thus we can use gradient descent

$$\Delta\theta \propto \sum_{t=0}^{\infty}\left[\frac{\partial}{\partial\theta}\log\pi_\theta(a_t|s_t)\left(\sum_{k=0}^{n} r_{t+n}+V_\alpha(s_{t+n+1})-V_\alpha(s_t)\right)\right].$$

Meanwhile we can update α by

$$\Delta\alpha \propto \left[\sum_{k=0}^{n} r_{t+n}+V_\alpha(s_{t+n+1})-V_\alpha(s_t)\right]\frac{\partial V}{\partial\alpha},$$

which seeks to approach the fixed point

$$V^\pi(s_t)=\mathbb{E}\left[\sum_{k=0}^{n} r_{t+k}+V^\pi(s_{t+n+1})\right]$$

by gradient descent on

$$[\hat{R}_t - V_\alpha(s_t)]^2.$$

The above is not a real loss function because the target $\hat{R}_t=\sum_{k=0}^{n} r_{t+k}+V_\alpha(s_{t+n+1})$ is based on the policy network V_α itself, and this bootstrapped target keeps changing. Without Monte Carlo, it becomes a self-fulfilling prophecy, and there is nothing for the value network to learn.

Q learning is similar to the learning of the value network $V_\alpha(s)$.

13.11 Partially observed MDP (POMDP)

Instead of directly observing s, we may observe $o\sim\rho(o|s)$, which is the observation model. Together with the dynamics, it becomes a hidden Markov model or state space model. We can update the belief $b_t=p(s_t\mid o_{\le t})$ by Bayes rule. In POMDP, we treat the belief as the state. Then it can still be formulated as a MDP.

13.12 Multi-agent reinforcement learning

The Alpha Go is a two agent RL. In multi-agent RL, we need to define policy and value for each player. In the cooperative setting, the players have a joint value function to maximize. In the competitive setting, the players play zero-sum or non-zero sum game. In real life, the situation can be both cooperative and competitive.

13.13 Inverse reinforcement learning (IRL)

In inverse reinforcement learning, an expert demonstrates (s_t,a_t). We can learn a policy $\pi_\theta(a|s)$ by supervised learning from the demonstrations by maximum likelihood. This is called behavior cloning. We can also learn a reward or value function $r_\theta(s,a)$ from demonstrations. This is called inverse reinforcement learning. The value function turns to be more generalizable than the policy.

13.14 Energy-based model

An energy-based model is of the following form

$$p_\theta(x)=\frac{1}{Z(\theta)}\exp(f_\theta(x)),$$

where $Z(\theta) = \int \exp(f_\theta(x))$ is the normalizing constant. This model originated from statistical mechanics, where $-f_\theta(x)$ is called the energy function of x.

Suppose we observe $x_i \sim p_{\text{data}}(x)$ independently for $i = 1, ..., n$. The log-likelihood is

$$l(\theta) = \frac{1}{n} \sum_{i=1}^{n} \log p_\theta(x_i),$$

whose gradient is

$$l'(\theta) = \frac{1}{n} \sum_{i=1}^{n} \frac{\partial}{\partial \theta} f_\theta(x_i) - \mathbb{E}_\theta \left[\frac{\partial}{\partial \theta} f_\theta(x) \right] \to \mathbb{E}_{p_{\text{data}}} \left[\frac{\partial}{\partial \theta} f_\theta(x) \right] - \mathbb{E}_\theta \left[\frac{\partial}{\partial \theta} f_\theta(x) \right],$$

and $\mathbb{E}_{\theta_t}$ can be approximated by Monte Carlo samples from $p_{\theta_t}(x)$.

IRL is a good context to introduce the energy-based model, where x is the action (we make the state implicit for simplicity), and $f_\theta(x)$ is the reward or value of action x.

The generator model or the flow model can be considered the policy model or actor, which can generates the samples directly, whereas the energy-based model can be considered a critic or evaluator. Sampling from energy-based model usually requires iterative Markov chain Monte Carlo (MCMC). It is also related to optimal control or planning, where $f_\theta(x)$ is an objective function.

Suppose we have a policy model $\pi_\alpha(x)$ such as a flow model, we can train it together with the energy-based model p_θ by adversarial contrastive divergence

$$\min_\theta \max_\alpha [\text{KL}(p_{\text{data}}|p_\theta) - \text{KL}(\pi_\alpha|p_\theta)],$$

where π_α is the actor and p_θ is the critic.

14 Background: Convex Optimization and Duality

SVM brought convex optimization to machine learning. We will give a systematic treatment to this topic. The easiest starting point is to write the primal problem as a min-max problem, and then change it to the dual max-min problem, which is often easier to solve. We also give other geometric interpretations.

14.1 von Neumann minimax theorem

von Neumann's game theory

There are two players, X and Y. Assume the loss function is $F(x,y)$ from X's perspective, where x is X's action and y is Y's action. From Y's perspective, his or her loss function is $-F(x,y)$. This is a zero sum game.

For each action x of X, the worse case scenario is $\max_y F(x,y)$. X wants to minimize the loss of the worse case, i.e., $\min_x \max_y F(x,y)$. Similarly, Y wants to $\max_y \min_x F(x,y)$. The solution to the game exists if

$$\min_x \max_y F(x,y) = \max_y \min_x F(x,y).$$

Convex-concave function and saddle point

$F(x,y)$ is a convex-concave function if $F(x,y)$ is convex in x given any fixed y, and $F(x,y)$ is concave in y given any fixed x. For example, $F(x,y) = x^2 - y^2$ is such a convex-concave function. For convex-concave

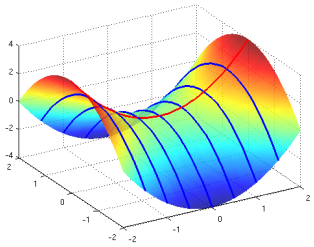


Figure 56: Saddle point of convex-concave function

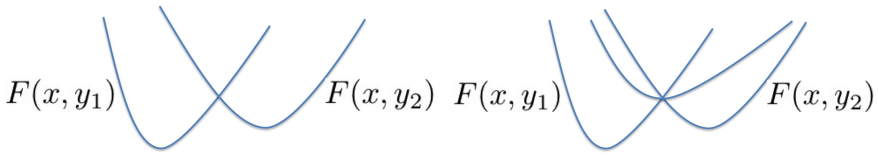


Figure 57: The functions $F(x,y_i)$ live on different slices of y_i. We project them onto the same plane. Left: min-max $\geq$ max-min. Right: For convex-concave F, max-min $\geq$ min-max.

$F(x,y)$, the solution to the above min-max and max-min problem exists, and is the saddle point. If $(x_\star,y_\star)$ is the saddle point of $F(x,y)$, then $F(x,y_\star) \geq F(x_\star,y_\star) \geq F(x_\star,y)$. See Figure 56.

The existence of the solution can be understood as follows. For $F(x,y)$, we can consider a discrete set of y, i.e., $\{y_1,...,y_n\}$. We can then plot the curves $F(x,y_i)$ for $i = 1,...,n$. Each $F(x,y_i)$ is a function of x. Then $\max_i F(x,y_i)$ is the upper envelop of these functions. $\min_x \max_i F(x,y_i)$ is the minimum of this upper envelop. Now consider $\min_x F(x,y_i)$, it is the minimum of the curve $F(x,y_i)$. All the n minima are below the upper envelop. $\max_i \min_x F(x,y_i)$ is the maximum of these n minima, and is also below the upper envelop. Thus $\min_x \min_i F(x,y_i) \geq \max_i \min_x F(x,y_i)$.

Now we show that the converse is also true for convex-concave function. Consider $F(x,y_1)$ and $F(x,y_2)$. Suppose we let y move on the line segment from y_1 to y_2. Suppose $y = \lambda y_1 + (1-\lambda)y_2$ for $\lambda \in [0,1]$. Because $F(x,y)$ is concave in y, we have

$$F(x,y) \geq \lambda F(x,y_1) + (1-\lambda)F(x,y_2) \geq \min_{i\in\{1,2\}} F(x,y_i),$$

where $\min_{i\in\{1,2\}} F(x,y_i)$ is the lower envelop. Let $m(y) = \min_x F(x,y)$ be the minimum of $F(x,y)$ for each y. As y moves from y_1 to y_2, the minimum must be above the lower envelop. Since $F(x,y)$ is convex for each y, thus at a certain point y_0, we must have $m(y_0)$ to be above $\min_x \max_{i\in\{1,2\}} F(x,y_i)$, which is on the lower envelop. Thus we have $\min_x \min_i F(x,y_i) \geq \max_i \min_x F(x,y_i)$.

Thus we have min-max = max-min.

14.2 Constrained optimization and Lagrange multipliers

Rewrite constrained optimization as min-max

The primal problem is

$$\text{minimize } f(x), \text{ subject to } g(x) \leq 0.$$

We can form the Lagrangian

$$L(x,\lambda) = f(x) + \lambda g(x),\ \lambda \geq 0.$$

The primal problem then becomes

$$\min_x \max_{\lambda \geq 0} L(x, \lambda).$$

Even though we remove the constraint $g(x) \leq 0$ in the above min-max problem, the constraint is automatically satisfied. This is because for all x such that $g(x) > 0$, we can let $\lambda \to \infty$, so that $L(x, \lambda) \to \infty$. Thus the solution to the min-max problem must satisfy the constraint $g(x) \leq 0$.

Complementary slackness

If (x, λ) is the solution to the above min-max problem, then we must have

$$\lambda g(x) = 0.$$

This is because $\lambda g(x) \leq 0$, since $g(x) \leq 0$ and $\lambda \geq 0$.

If $g(x) < 0$ strictly, then $\lambda = 0$. If $\lambda > 0$, then we must have $g(x) = 0$. This is the complementary slackness, which is part of the KKT condition.

Dual problem

The dual problem is $\min_{\lambda \geq 0} \min_x L(x, \lambda)$.

Let

$$q(\lambda) = \min_x L(x, \lambda).$$

Then the dual problem becomes $\max_{\lambda \geq 0} q(\lambda)$. After we find the optimal $\lambda_\star$, we can then go back to get the optimal $x_\star$ by retrieving the solution to the above $\min_x$ problem.

The dual problem can be simpler than the primal problem if the dimensionality of λ is much smaller than the dimensionality of x. The Lagrange multiplier λ may also have a meaningful interpretation.

Dual is concave

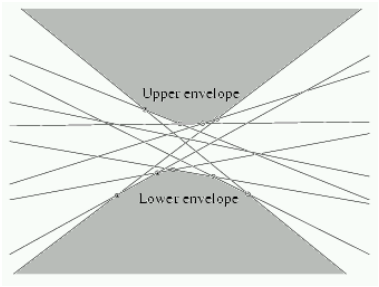


Figure 58: For each λ, $q(\lambda)$ is the minimum of all the lines $\{f(x) + \lambda g(x), \forall x\}$, thus the function $q(\lambda)$ lower envelops these lines, and is a concave function. The dual problem is to maximize $q(\lambda)$ over $\lambda \geq 0$.

.

$q(\lambda)$ is a concave function. For any fixed x, $L(x, \lambda)$ is a linear function of λ with slope $g(x)$ and intercept $f(x)$. For all x, we have a set of lines. For each λ, we take the minimum of all the lines. This gives us the lower envelop illustrated in Figure 58. The lower envelop is concave whether $f(x)$ and $g(x)$ are convex or not.

Two important dualities in statistics and machine learning are: (1) The duality between maximum entropy and maximum likelihood in exponential family model. (2) The duality between maximum margin

and minimal distance in support vector machine (SVM). In addition to the general geometric interpretations provided here, each specific duality has its own special geometric interpretation.

Strong and weak duality

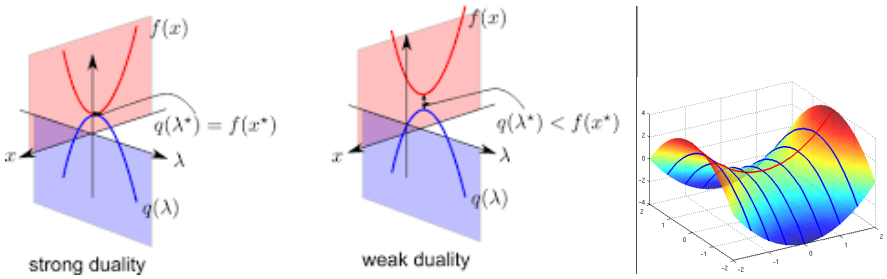


Figure 59: $q(\lambda)$ lower bounds $f(x)$. In the left plot, $q(\lambda)$ kisses $f(x)$ at $(x_\star, \lambda_\star)$, and there is strong duality. In the right plot, there is a duality gap between $q(\lambda)$ and $f(x)$, and there is weak duality.

The function

$$q(\lambda) = \min_x L(x,\lambda) \leq \min_{x:g(x)\leq 0} L(x,\lambda) \leq f(x),$$

for $\lambda \geq 0$, thus $q(\lambda)$ lower bounds $f(x)$, and $q(\lambda_\star) \leq f(x_\star)$. If $q(\lambda_\star) = f(x_\star)$, i.e., $q(\lambda)$ kisses $f(x)$, we have strong duality. If $q(\lambda_\star) < f(x_\star)$, there is a duality gap and we have weak duality.

Contour plot

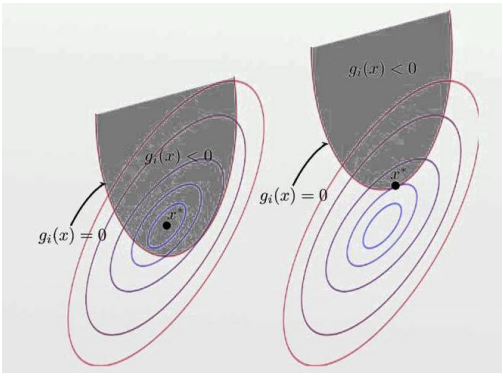


Figure 60: In the left plot, the minimum of $f(x)$ is within the constrained region $g(x) \leq 0$. Thus $\lambda_\star = 0$ and $g(x_\star) < 0$. In the right plot, the minimum of $f(x)$ is outside the constrained region, and a contour of $f(x)$ touches the contour $g(x) = 0$ at $x_\star$. The two contours are co-tangent at $x_\star$, and the gradients of $f(x)$ and $g(x)$ at $x_\star$ are of opposite directions, so that $f'(x_\star) = \lambda g'(x_\star)$ for a $\lambda > 0$.

The situation is illustrated by Figure 60. In the left plot, the minimum of $f(x)$ is within the constraint $g(x) < 0$. Thus $g(x_\star) < 0$ and $\lambda_\star = 0$. In the right plot, the minimum of $f(x)$ is outside the constraint $g(x) \geq 0$. Thus the constrained minimum is achieved when the contour (level set) of $f(x)$ touches the contour $g(x) = 0$ at $x_\star$, where the two contours are co-tangent, and the gradients of $f(x)$ and $g(x)$ at $x_\star$ are of the opposite directions. Thus $f'(x_\star) = -\lambda g'(x_\star)$ for some $\lambda > 0$, and the derivative of $f(x) + \lambda g(x)$ is equal to 0 at $x_\star$, which minimizes $f(x) + \lambda g(x)$.

Range plot

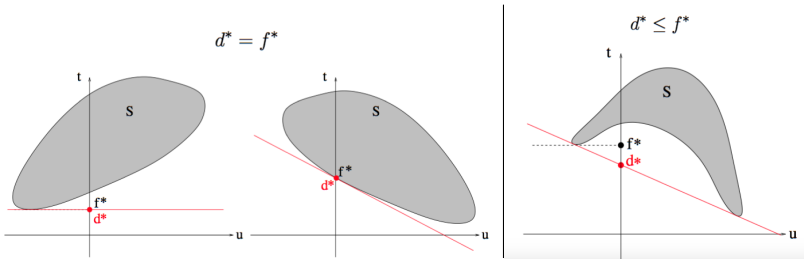


Figure 61: Left: the shaded region is $S = \{u = g(x), t = f(x)\}$. The constrained minimum is the lowest point of the left half of the region with $u \le 0$. For each $\lambda \ge 0$, the minimum $q(\lambda)$ of $f(x) + \lambda g(x) = t + \lambda u$ is the intercept of the supporting line of S with non-positive slope $-\lambda$. The maximum of $q(\lambda)$ is the highest intercept among all the supporting lines of S. In the left plot, the minimum is attained for $u < 0$, and $\lambda_\star = 0$. In the right plot, the minimum is attained at $u = 0$, and $\lambda_\star < 0$. Right: The maximum of the dual problem $d^\star$ is less than the minimum of the primal problem $f^\star$, so we have weak duality.

The situation can also be illustrated by Figure 64. Let $(u = g(x), t = f(x))$ be a point in the space of (u,t). The range of (u,t) is illustrated by the shaded region $S = \{u = g(x), t = f(x), \forall x\}$. The minimum of $t = f(x)$ is taken on the half of the shaded area to the left of the vertical axis, i.e., $u = g(x) \le 0$. For each λ, the minimum of $f(x) + \lambda g(x)$, which is $q(\lambda)$, is the intercept c of the line $t = -\lambda u + c$ with a negative slope $-\lambda$ and the line supports the whole shaded region. We want to find $\lambda \ge 0$ that achieves the highest intercept, which is $q(\lambda_\star)$. If the shaded area of (t,u) is concave, the highest intercept $q(\lambda_\star)$ equals $f(x_\star)$, and we have the strong duality. If $u_\star = g(x_\star) < 0$, the line is horizontal, and we have $\lambda_\star = 0$. If $u_\star = g(x_\star) = 0$, we have $\lambda_\star > 0$, and the line is tangent to the shaded region at $u = g(x) = 0$. If the shaded area is not concave, we have weak duality, as illustrated by the right panel of Figure 64.

Equality constraint

The equality constraint can be similarly treated. Suppose we want to minimize $f(x)$ subject to $g(x) = 0$. The Lagrangian is $L(x,\lambda) = f(x) + \lambda g(x)$, where λ can be any real number.

If we want to minimize $f(x)$ subject to $g_0(x) = 0$ and $g_1(x) \le 0$. The Lagrangian is $L(x,\lambda_0,\lambda_1) = f(x) + \lambda_0 g_0(x) + \lambda_1 g_1(x)$, with $\lambda_1 \ge 0$.

14.3 Legendre-Fenchel convex conjugate

In constrained optimization, we can rewrite the problem into a min-max problem using the Lagrangian. We can also rewrite an unconstrained minimization problem as a min-max problem, like what we did for the hinge loss.

min-max = max-min again

Suppose the primal problem is

$$\min_x [f(x) + \rho(x)],$$

e.g., $\rho(x) = |x|^2/2$, and $f(x)$ is the loss function such as the hinge loss. We can write

$$f(x) = \max_\lambda [\lambda x - f^*(\lambda)].$$

Then the primal problem becomes

$$\min_x \max_\lambda [\lambda x - f^*(\lambda) + \rho(x)].$$

Its dual problem is

$$\max_\lambda \min_x [\lambda x + \rho(x) - f^*(\lambda)].$$

Let

$$q(\lambda) = \min_x [\lambda x + \rho(x) - f^*(\lambda)],$$

which is particularly simple if $\rho(x) = |x|^2/2$ and, in such a case, the dual problem is to maximize $q(\lambda)$.

Supporting lines and upper envelop

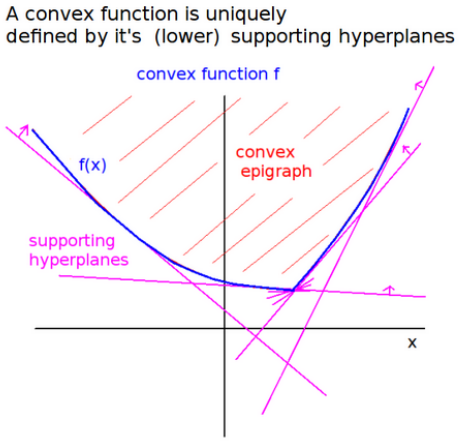


Figure 62: The lower support lines and the upper envelop.

f^* is the Legendre-Fenchel transform or the convex conjugate of f. For each λ, $\lambda x - f^*(\lambda)$ is the lower supporting line of $f(x)$ with slope λ. As a result, $f(x)$ is the upper envelop of all the lower supporting lines.

Specifically, for a function $f(x)$, let

$$f^*(\lambda) = \max_x [\lambda x - f(x)].$$

Geometrically, $f^*(\lambda)$ is the maxima gap between λx and $f(x)$, i.e., if we drop the line λx by $f^*(\lambda)$, so that it becomes $\lambda x - f^*(\lambda)$, then this line is a supporting line of the function $f(x)$. Stated differently, this line is below $f(x)$, but touches $f(x)$ at $x_\star$ that maximizes $\lambda x - f(x)$.

We can represent $f(x)$ as the envelop of its lower supporting lines, i.e.,

$$f(x) = \max_\lambda [\lambda x - f*(\lambda)].$$

For convex f, $f^{**} = f$. Otherwise, f^{**} is the lower convex envelop of f.

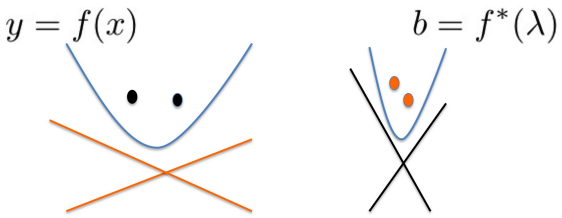


Figure 63: Point line duality. A point in one plane corresponds to a line in another plane, and vice versa.

Point-line duality

The same form of calculating lower support line and upper envelop hints at a symmetry or duality. It is the point-line duality. Specifically, for a point (λ, b) in one plane, there is a line $y = \lambda x - b$ in the plane (x, y). Conversely for a point (x, y), there is a line $b = \lambda x - y$ in the plane (λ, b). Two points forming a line means two lines meet at a point. For a convex function $y = f(x)$, any point above $y = f(x)$ corresponds to a line below $b = f^*(\lambda)$. Similarly, for any point (λ, b) above the curve $b = f^*(\lambda)$, there is a line below $y = f(x)$.

Complementary regions

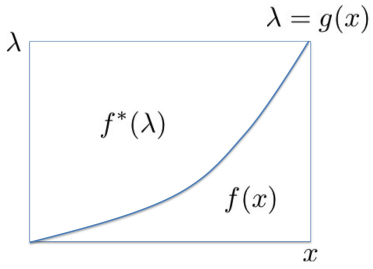


Figure 64: Complementary regions within the rectangle $[0, x] \times [0, \lambda]$. $f(x)$ is the area of the region below $\lambda = g(x)$, and $f^*(\lambda)$ is the area of the region above $\lambda = g(x)$.

Another interpretation is based on complementary regions. Consider the curve $\lambda = g(x)$ within the rectangle $[0, x] \times [0, \lambda]$. Let $f(x) = \int_0^x g(x)dx$ be the area below $g(x)$ up to x. Let $f^*(\lambda) = \int_0^\lambda g^{-1}(\lambda)d\lambda$ be the area above $g(x)$ up to λ. The total area of the rectangle is λx. Then $f(x) = \max_\lambda [\lambda x - f^*(\lambda)]$ and $f^*(\lambda) = \max_x [\lambda x - f(x)]$.

15 Background: Maximum Likelihood

The theory of maximum likelihood is an extension of the theory of Gauss on the optimality of least squares.

15.1 REINFORCE

The REINFORCE algorithm is based on the following identity

$$\begin{aligned}\frac{\partial}{\partial\theta}\mathbb{E}_\theta[h(x)] &= \frac{\partial}{\partial\theta}\int h(x)p_\theta(x)dx \\ &= \int h(x)\frac{\partial}{\partial\theta}p_\theta(x)dx \\ &= \int\left[h(x)\frac{\partial}{\partial\theta}\log p_\theta(x)\right]p_\theta(x)dx \\ &= \mathbb{E}_\theta\left[h(x)\frac{\partial}{\partial\theta}\log p_\theta(x)\right].\end{aligned}$$

$h(x)$ is the cumulated reward. x is the action. The above gradient is similar to maximum likelihood, but the gradient of the log-likelihood is weighted by the reward $h(x)$.

15.2 Expectation of score is zero

In the above, let $h(X) = 1$ or constant. Then

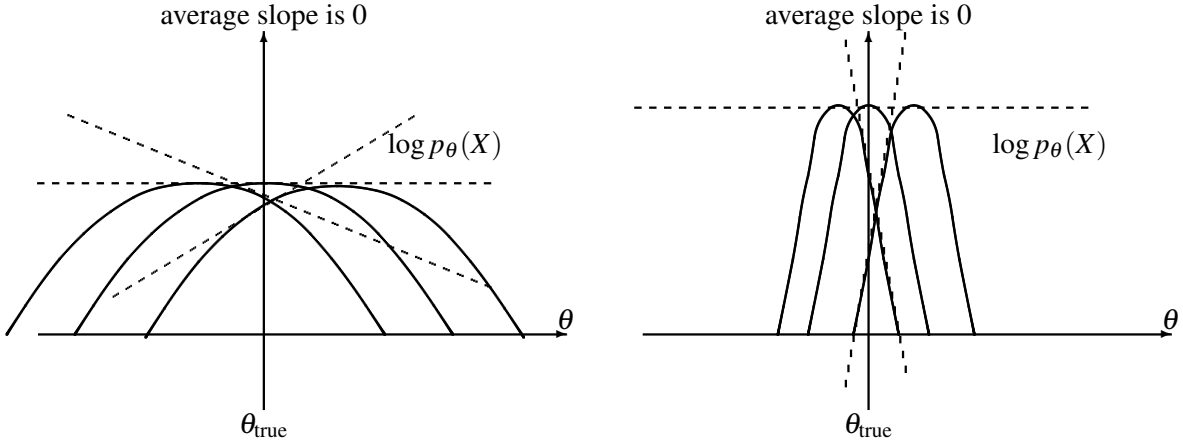
$$\frac{\partial}{\partial\theta}\mathbb{E}_\theta[1] = 0 = \mathbb{E}_\theta\left[\frac{\partial}{\partial\theta}\log_\theta(x)\right].$$

Or more directly,

$$\frac{\partial}{\partial\theta}\int p_\theta(x)dx = 0 = \int\left[\frac{\partial}{\partial\theta}\log p_\theta(x)\right]p_\theta(x)dx = \mathbb{E}_\theta\left[\frac{\partial}{\partial\theta}\log p_\theta(x)\right].$$

$\frac{\partial}{\partial\theta}\log p_\theta(x)$ is called score function.

Two interpretations



Interpretation 1: In the above figure, $x \sim p_{\theta_{\text{true}}}(x)$ for a fixed θ_{true}. Since x is random, $\log p_\theta(x)$ is a random function of θ. The derivative of this random function at θ_{true} can be positive, negative or zero. On average, the derivative is zero.

Interpretation 2: If $x \sim p_{\theta_{\text{true}}}(x)$ for a fixed θ_{true}. Suppose we want to learn θ by maximizing the log-likelihood $\log p_\theta(x)$ over θ. Suppose we are already at θ_{true}, then the gradient at θ_{true} should be zero on average, i.e., the stochastic gradient algorithm should just stay around θ_{true} without a clear drift.

Although we compare REINFORCE with maximum likelihood, we need to be careful that the action x is generated from the current policy $p_\theta(x)$. The action x is not observed from an expert or a demonstrator. Therefore, there is nothing for the model to learn from the x generated by itself, i.e.,

$$\mathbb{E}_\theta \left[\frac{\partial}{\partial \theta} \log_\theta(x) \right] = 0,$$

according to the interpretation 2 above. The reason that REINFORCE works is because of the reward $h(x)$, which tells which action x is relatively better. Here $h()$ serves to compare different x, i.e., $h()$ is meaningful only in terms of relative comparison.

15.3 Estimating equation

The log-likelihood

$$l(\theta) = \frac{1}{n} \sum_{i=1}^{n} \log p_\theta(x_i)$$

measures the plausibility of θ in explaining $(x_i, i = 1, ..., n)$. To find the maximum likelihood estimate, we can maximize $l(\theta)$ by solving $l'(\theta) = 0$, i.e., we solve the estimating equation

$$\frac{1}{n} \sum_{i=1}^{n} \frac{\partial}{\partial \theta} \log p_\theta(x_i) = 0.$$

Let $\hat{\theta}$ be the solution. $\hat{\theta}$ is random because x is random.

This equation is true on average, e.g.,

$$\mathbb{E}_\theta \left[\frac{\partial}{\partial \theta} \log p_\theta(x) \right] = 0, \forall \theta,$$

or with more clear notation,

$$\mathbb{E}_{\theta_{\text{true}}} \left[\frac{\partial}{\partial \theta} \log p_\theta(x) \big|_{\theta_{\text{true}}} \right] = 0, \forall \theta_{\text{true}}.$$

Thus $\hat{\theta}$ fluctuates around θ_{true}, and $\hat{\theta} \to \theta_{\text{true}}$ as $n \to \infty$, because according to the law of large number, the estimating equation converges to

$$\mathbb{E}_{\theta_{\text{true}}} \left[\frac{\partial}{\partial \theta} \log p_\theta(x) \right] = 0,$$

whose solution is θ_{true} no matter what θ_{true} is. This is called consistency of $\hat{\theta}$.

In general, suppose we have a function $h_\theta(x)$ so that

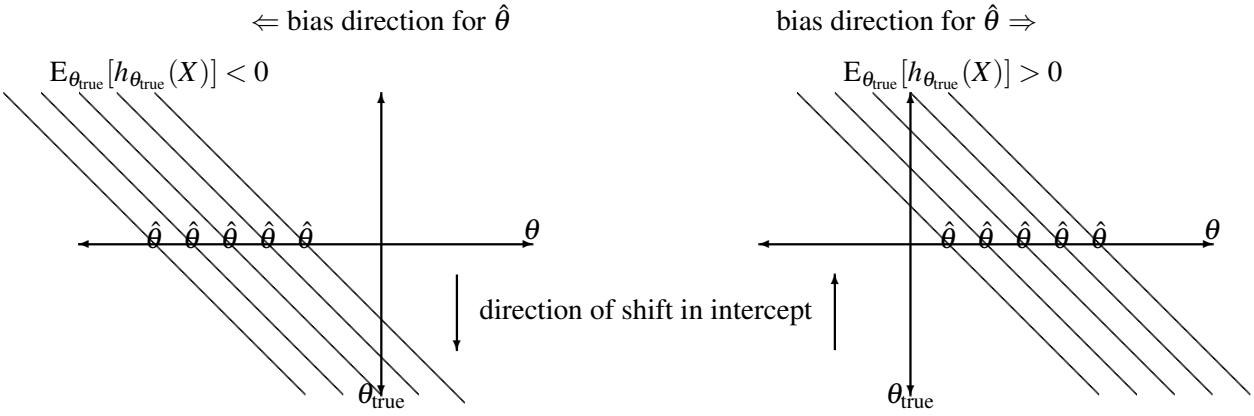
$$\mathbb{E}_\theta[h_\theta(x)] = 0, \forall \theta,$$

then we can estimate θ by solving the estimating equation

$$\frac{1}{n} \sum_{i=1}^{n} h_\theta(x_i) = 0.$$

15.4 Unbiased estimator

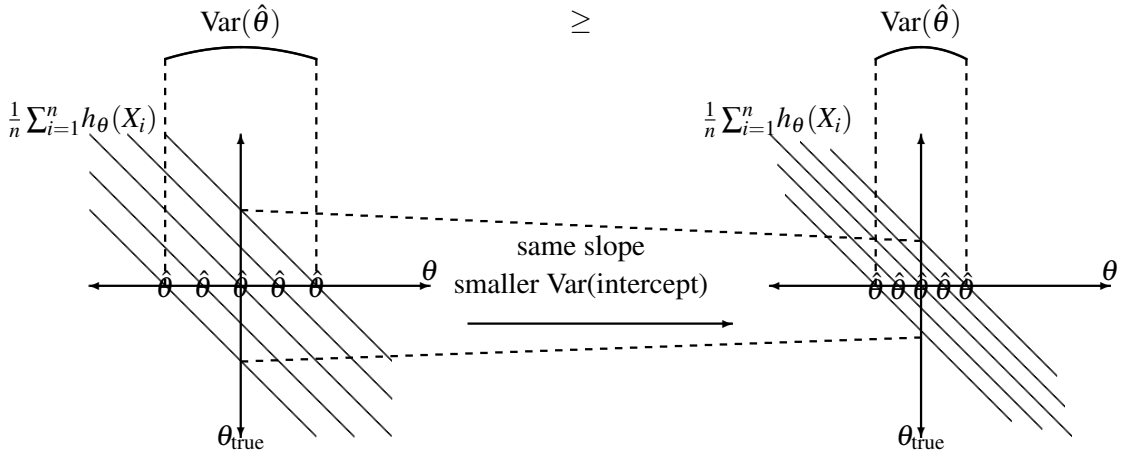
The requirement that $\mathbb{E}_{\theta_{\text{true}}}[h_{\theta_{\text{true}}}(x)] = 0$ is to ensure that there is no bias.



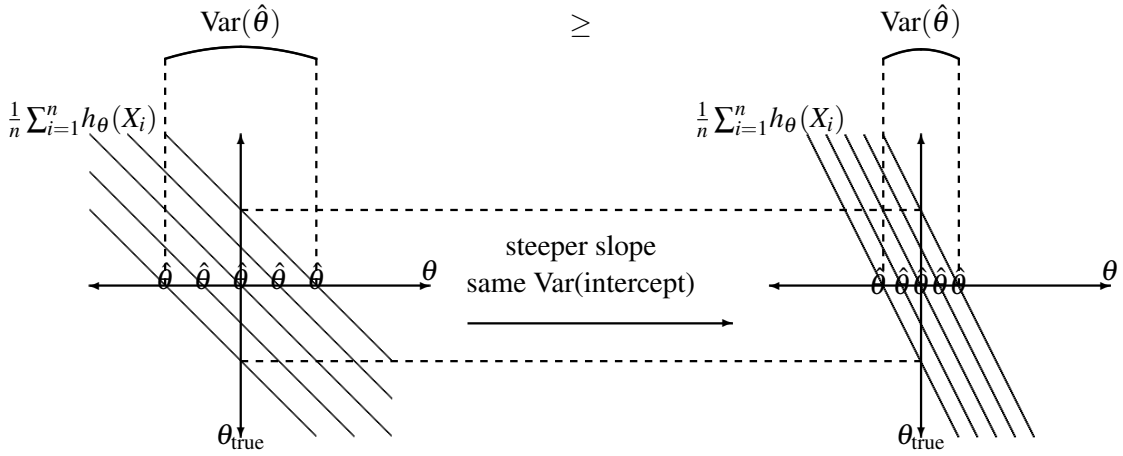
Let $x_i \sim p_{\theta_{\text{true}}}(x)$ for $i = 1, ..., n$. The dataset $(x_i, i = 1, ..., n)$ is random, thus $\frac{1}{n}\sum_{i=1}^{n} h_\theta(x_i)$, illustrated as a line, is also random. $\frac{1}{n}\sum_{i=1}^{n} h_{\theta_{\text{true}}}(x_i)$ is the intercept. If the intercept does not fluctuate around zero, the estimated $\hat{\theta}$ will not fluctuate around θ_{true}. Thus we should have $\mathbb{E}_{\theta_{\text{true}}}[h_{\theta_{\text{true}}}(x)] = 0$ for any θ_{true}. Such $h()$ leads to the unbiased estimator.

15.5 Variance of estimator

The variance of the estimator depends on the variance of intercept as well as the magnitude of the slope of $h_{\theta_{\text{true}}}(x)$.



If we keep the slope fixed, then the smaller the variance of the intercept, the smaller the variance of the estimator.



If we keep the variance of the intercept fixed, then the bigger the slope, the smaller the variance of the estimator.

15.6 Asymptotic distribution

More formally, the above explanation is based on the first order Talyor around θ_{true},

$$\frac{1}{n}\sum_{i=1}^{n} h_{\theta_{\text{true}}}(x_i) + \frac{1}{n}\sum_{i=1}^{n} \frac{\partial}{\partial\theta} h_\theta(x_i)\big|_{\theta_{\text{true}}} (\theta - \theta_{\text{true}}) = 0,$$

Thus

$$\sqrt{n}(\hat{\theta} - \theta_{\text{true}}) = -\frac{\frac{1}{\sqrt{n}}\sum_{i=1}^{n} h_{\theta_{\text{true}}}(x_i)}{\frac{1}{n}\sum_{i=1}^{n} \frac{\partial}{\partial\theta} h_\theta(x_i)\big|_{\theta_{\text{true}}}}.$$

One may think of it as a Newton-Raphson step from θ_{true}.

According to the central limit theorem,

$$\frac{1}{\sqrt{n}}\sum_{i=1}^{n} h_{\theta_{\text{true}}}(x_i) \sim \mathrm{N}(0, \mathrm{Var}_{\theta_{\text{true}}}[h_{\theta_{\text{true}}}(x)]).$$

According to the law of large number,

$$\frac{1}{n}\sum_{i=1}^{n} \frac{\partial}{\partial\theta} h_\theta(x_i)\big|_{\theta_{\text{true}}} \to \mathbb{E}_{\theta_{\text{true}}}\left[\frac{\partial}{\partial\theta} h_\theta(x)\big|_{\theta_{\text{true}}}\right].$$

Thus,

$$\sqrt{n}(\hat{\theta} - \theta_{\text{true}}) = \mathrm{N}\left(0, \frac{\mathrm{Var}_{\theta_{\text{true}}}[h_{\theta_{\text{true}}}(x)]}{\mathbb{E}_{\theta_{\text{true}}}\left[\frac{\partial}{\partial\theta} h_\theta(x)\big|_{\theta_{\text{true}}}\right]^2}\right).$$

The variance of $\hat{\theta}$ is determined by the variance of the intercept and the magnitude of the slope.

15.7 Optimality of MLE

For the best unbiased estimator, we need to minimize the variance of the estimator. Since

$$\mathbb{E}_\theta[h_\theta(x)] = \int h_\theta(x) p_\theta(x) dx = 0, \ \forall\theta,$$

$$\frac{\partial}{\partial\theta}\mathbb{E}_\theta[h_\theta(x)] = \frac{\partial}{\partial\theta}\int h_\theta(x)p_\theta(x)dx = 0,$$

$$\begin{aligned}\frac{\partial}{\partial\theta}\int h_\theta(x)p_\theta(x)dx &= \int \frac{\partial}{\partial\theta}h_\theta(x)p_\theta(x)dx + \int\left[h_\theta(x)\frac{\partial}{\partial\theta}\log p_\theta(x)\right]p_\theta(x)dx \\ &= \mathbb{E}_\theta\left[\frac{\partial}{\partial\theta}h_\theta(x)\right] + \mathbb{E}_\theta\left[h_\theta(x)\frac{\partial}{\partial\theta}\log p_\theta(x)\right] \\ &= \mathbb{E}_\theta\left[\frac{\partial}{\partial\theta}h_\theta(x)\right] + \mathrm{Cov}_\theta\left[h_\theta(x), \frac{\partial}{\partial\theta}\log p_\theta(x)\right] = 0\end{aligned}$$

Thus

$$\mathbb{E}_\theta\left[\frac{\partial}{\partial\theta}h_\theta(x)\right] = -\mathrm{Cov}_\theta\left[h_\theta(x), \frac{\partial}{\partial\theta}\log p_\theta(x)\right]$$

$$\begin{aligned}\mathbb{E}_\theta\left[\frac{\partial}{\partial\theta}h_\theta(x)\right]^2 &= \mathrm{Cov}_\theta\left[h_\theta(x), \frac{\partial}{\partial\theta}\log p_\theta(x)\right]^2 \\ &\leq \mathrm{Var}_\theta[h_\theta(x)]\mathrm{Var}_\theta\left[\frac{\partial}{\partial\theta}\log p_\theta(x)\right].\end{aligned}$$

$$\frac{\mathrm{Var}_\theta[h_\theta(x)]}{\mathbb{E}_\theta\left[\frac{\partial}{\partial\theta}h_\theta(x)\right]^2} \geq \left(\mathrm{Var}_\theta\left[\frac{\partial}{\partial\theta}\log p_\theta(x)\right]\right)^{-1}.$$

$$\mathrm{Var}_{\theta_{\text{true}}}(\hat{\theta}) \geq \frac{1}{n}\left(\mathrm{Var}_{\theta_{\text{true}}}\left[\frac{\partial}{\partial\theta}\log p_\theta(x)\big|_{\theta_{\text{true}}}\right]\right)^{-1}.$$

The minimum is achieved by

$$h_\theta(x) \propto \frac{\partial}{\partial\theta}\log p_\theta(x),$$

which leads to the maximum likelihood estimating equation.

15.8 Fisher information

$$\mathbb{E}_\theta\left[\frac{\partial}{\partial\theta}h_\theta(x)\right] = -\mathrm{Cov}_\theta\left[h_\theta(x), \frac{\partial}{\partial\theta}\log p_\theta(x)\right]$$

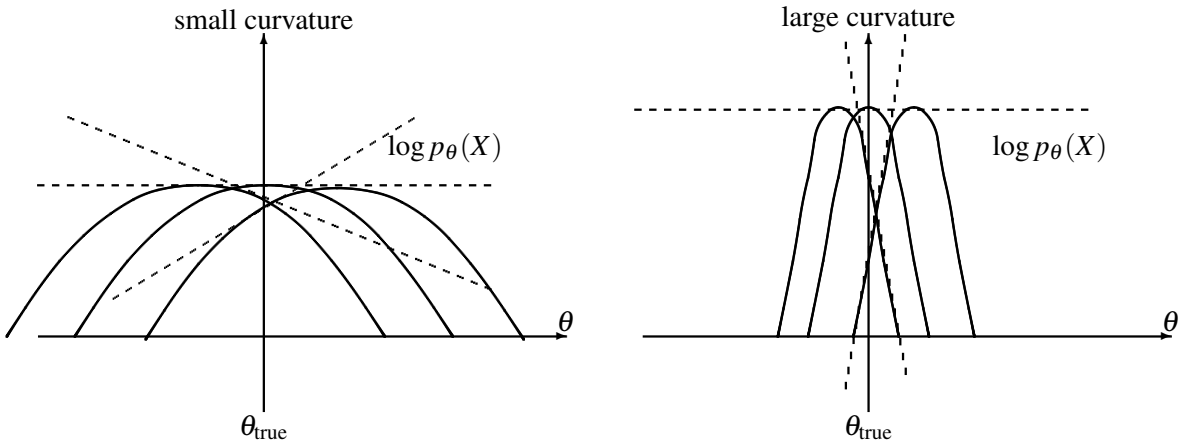
If $h_\theta(x) \propto \frac{\partial}{\partial\theta}\log p_\theta(x)$, then

$$\begin{aligned}-\mathbb{E}_\theta\left[\frac{\partial^2}{\partial\theta^2}\log p_\theta(x)\right] &= \mathrm{Cov}_\theta\left[\log p_\theta(x), \frac{\partial}{\partial\theta}\log p_\theta(x)\right] \\ &= \mathrm{Var}_\theta\left[\frac{\partial}{\partial\theta}\log p_\theta(x)\right] = I(\theta),\end{aligned}$$

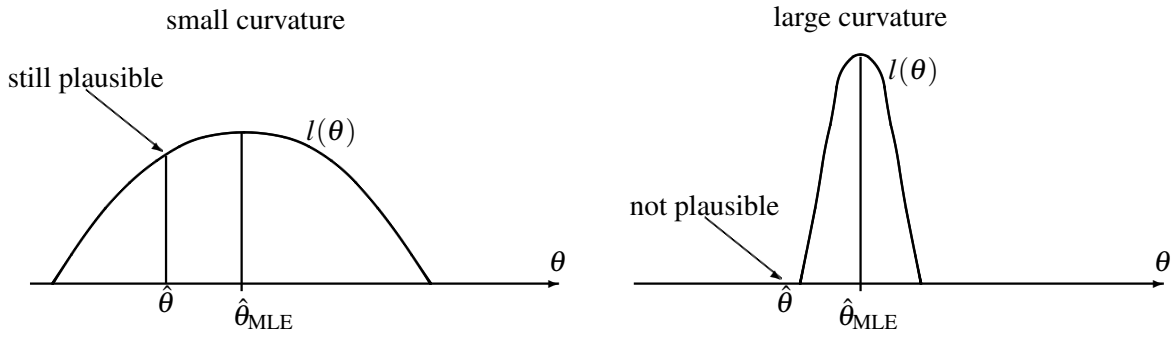
which is the Fisher information. Thus for MLE,

$$\sqrt{n}(\hat{\theta} - \theta_{\text{true}}) = \mathrm{N}\left(0, I(\theta_{\text{true}})^{-1}\right).$$

The bigger the Fisher information, the smaller the variance.



Fisher information measures the curvature of the log-likelihood as well as the variance of the slope of the log-likelihood. Both the curvature and variance measure the change of the slope around θ_{true}.



The log-likelihood $l(\theta) = \frac{1}{n}\sum_{i=1}^{n} \log p_\theta(x_i)$ measures the plausibility of θ in explaining $(x_i, i = 1, ..., n)$. If $l(\theta)$ is of high curvature, then a small deviation from MLE will make θ not plausible. Thus the dataset gives us strong information about θ. If $l(\theta)$ is of low curvature, the dataset does not tell us much information about θ.

15.9 Information geometry

A family of distributions $\{p_\theta\}$ forms a manifold. What is a natural metric? How do we measure the distance locally, i.e., between p_θ and $p_{\theta+\Delta\theta}$? A natural choice is Kullback-Leibler. Using the second order Talyor, we have

$$\begin{aligned}
\text{KL}(p_\theta | p_{\theta+\Delta\theta}) &= \mathbb{E}_\theta[\log p_\theta(x) - \log p_{\theta+\Delta\theta}(x)] \\
&= \mathbb{E}_\theta\left[\log p_\theta(x) - (\log p_\theta(X) + \langle \frac{\partial}{\partial\theta}\log p_\theta(x), \Delta\theta\rangle + \frac{1}{2}\Delta\theta^\top \frac{\partial^2}{\partial\theta^2}\log p_\theta(x)\Delta\theta + o(|\Delta\theta|^2))\right] \\
&= \frac{1}{2}\Delta\theta^\top I(\theta)\Delta\theta
\end{aligned}$$

where the Fisher information $I(\theta)$ serves as the metric. Recall any symmetric matrix is a diagonal matrix from a rotated viewpoint Q which consists of eigen vectors. Viewed from this perspective, when computing the distance, we scale the components of $\Delta\theta$ in this viewpoint differently. Therefore, a circle around θ in the KL distance is an ellipse in the Euclidean distance.

The KL divergence is invariant to re-parametrization. If we change θ to $\gamma = f(\theta)$ by a one-to-one mapping f, we will not change the KL-divergence.

15.10 Natural gradient

The gradient is the steepest ascent we can achieve if we move a fixed small distance. Measured in Euclidean distance,

$$\begin{aligned}\log p_{\theta+\Delta\theta}(x) &= \log p_\theta(x) + \langle \Delta\theta, \frac{\partial}{\partial\theta}\log p_\theta(x)\rangle + o(|\Delta\theta|) \\ &\leq \log p_\theta(x) + |\Delta\theta|\big|\frac{\partial}{\partial\theta}\log p_\theta(x)\big|,\end{aligned}$$

with equality achieved by $\Delta\theta \propto \frac{\partial}{\partial\theta}\log p_\theta(x)$, which is the steepest direction for fixed $|\Delta\theta|$. If we measure the squared distance $\Delta\theta$ by $\Delta\theta^\top I(\theta)\Delta\theta$, i.e., by $|I(\theta)^{1/2}\Delta|^2$, then

$$\begin{aligned}\log p_{\theta+\Delta\theta}(x) &= \log p_\theta(x) + \langle \Delta\theta, \frac{\partial}{\partial\theta}\log p_\theta(x)\rangle \\ &= \log p_\theta(x) + \langle I(\theta)^{1/2}\Delta\theta, I(\theta)^{-1/2}\frac{\partial}{\partial\theta}\log p_\theta(x)\rangle \\ &\leq \log p_\theta(x) + |I(\theta)^{1/2}\Delta\theta|\big|I(\theta)^{-1/2}\frac{\partial}{\partial\theta}\log p_\theta(x)\big|,\end{aligned}$$

with the equality achieved by $I(\theta)^{1/2}\Delta\theta \propto I(\theta)^{-1/2}\frac{\partial}{\partial\theta}\log p_\theta(x)$, thus

$$\Delta\theta \propto I(\theta)^{-1}\frac{\partial}{\partial\theta}\log p_\theta(x)$$

which is the steepest direction for fixed $\mathrm{KL}(p_\theta|p_{\theta+\Delta\theta}) = |I(\theta)^{1/2}\Delta|^2$. $I(\theta)^{-1}\frac{\partial}{\partial\theta}\log p_\theta(x)$ is the natural gradient, which is invariant under re-parametrization.

16 Background: Second Order Taylor Expansion

16.1 Vector Form Second Order Taylor Expansion

Suppose $f:\mathbb{R}^{d\times 1}\to\mathbb{R}$ and $x\in\mathbb{R}^{d\times 1}$, then

$$y = f(x) = f(x_1, x_2, \ldots, x_d)$$

and

$$\begin{aligned}f'(x) &= \left(\frac{\partial f}{\partial x_i}\right)^\top_{i=1,\ldots,d} \\ f''(x) &= \left(\frac{\partial^2 f}{\partial x_i \partial x_j}\right)_{i=1,\ldots,d,j=1,\ldots,d}\end{aligned}$$

where $f'(x)\in\mathbb{R}^{d\times 1}$ and $f''(x)\in\mathbb{R}^{d\times d}$.

We consider the second order Taylor expansion of $f(x) = f(x_0+tu)$ at x_0, where u is a $d\times 1$ vector, and we can understand u as a unit length vector: suppose $x = x_0+\Delta x$, let $t = |\Delta x|$, then $u = \frac{\Delta x}{|\Delta x|}$. Let $F:\mathbb{R}\to\mathbb{R}$ with $F(t) = f(x)$, the second order Taylor expansion can be written as

$$F(t) = F(0) + F'(0)t + \frac{1}{2}F''(0)t^2 + o(t^2).$$

Next, we derive the vector form second order Taylor expansion. First, we rewrite $F'(t)$ and $F''(t)$ as function of x by the chain rule:

$$F'(t) = \frac{\partial}{\partial t} f(x_0 + tu) = \sum_{i=1}^{d} \frac{\partial f(x)}{\partial x_i} \frac{\partial x_i}{\partial t} = \langle f'(x), u \rangle$$

since $f(x)$ depends on t via x_i and $x_i = u_i t + x_{0i}$ for $i = 1 \dots d$. Then,

$$\begin{aligned}
F''(t) &= \frac{\partial}{\partial t} F'(t) \\
&= \frac{\partial}{\partial t} \sum_{i=1}^{d} \frac{\partial f(x)}{\partial x_i} \frac{\partial x_i}{\partial t} \\
&= \frac{\partial}{\partial t} \sum_{i=1}^{d} \frac{\partial f(x)}{\partial x_i} u_i \\
&= \sum_{i=1}^{d} \frac{\partial}{\partial t} \frac{\partial f(x)}{\partial x_i} u_i \\
&= \sum_{i=1}^{d} \sum_{j=1}^{d} \frac{\partial^2 f(x)}{\partial x_i \partial x_j} \frac{\partial x_j}{\partial t} u_i \\
&= \sum_{i=1}^{d} \sum_{j=1}^{d} \frac{\partial f(x)}{\partial x_i \partial x_j} u_j u_i \\
&= u^\top f''(x) u
\end{aligned}$$

In fact, we can derive the formula compactly using the vector notation:

$$F'(t) = \frac{\partial f}{\partial x^\top} \frac{\partial x}{\partial t} = \langle f'(x), u \rangle = u^\top f'(x)$$

and

$$F''(t) = \frac{\partial^2 f}{\partial t^2} = u^\top \frac{\partial}{\partial t} f'(x) = u^\top \frac{\partial f'(x)}{\partial x^\top} \frac{\partial x}{\partial t} = u^\top f''(x) u$$

However, this derivation involves multi-variable calculus and the first way of the derivation is easier to understand.

Now we get the vector form second order Taylor expansion:

$$\begin{aligned}
f(x) &= F(t) \\
&= F(0) + F'(0)t + \frac{1}{2} F''(0) t^2 + o(t^2) \\
&= f(x_0) + \langle f'(x), u \rangle t + \frac{1}{2} u^\top f''(x_0) u t^2 + o(t^2) \\
&= f(x_0) + \langle f'(x_0), \Delta x \rangle + \frac{1}{2} \Delta x^\top f''(x_0) \Delta x + o(|\Delta x|^2)
\end{aligned}$$

16.2 Geometric Understanding of Second Order Taylor Expansion

Let $g = f'(x_0)$ and $H = f''(x_0)$ be the Hessian matrix. We first investigate the geometric understanding of the first order term:

Note that $\langle g, \Delta x \rangle = |g||\Delta x| \cos\theta$, where θ denotes the angle between g and Δx. Then when $\Delta x \propto g$, the magnitude of $\langle g, \Delta x \rangle = |g||\Delta x|$ is largest; while the magnitude of $\langle g, \Delta x \rangle = 0$ is smallest when $\Delta x \perp g$. This explains why we update in the direction of the first derivative in gradient descent.

Then we consider the geometric understanding of the second order term:

By diagonalizing the hessian matrix H, we get $H = Q\Lambda Q^\top$, where $\Lambda = \text{diag}(\lambda_i)_{i=1,\dots,d}$ is a diagonal matrix, and $Q = (q_i)_{i=1,\dots,d}$ and $\langle q_i, q_j \rangle = 1$ if $i = j$; $\langle q_i, q_j \rangle = 0$ otherwise. Let $\Delta z = Q^\top \Delta x$, we get

$$\Delta x^\top H \Delta x = \Delta x^\top Q \Lambda Q^\top \Delta x = \Delta z^\top \Lambda \Delta z = \sum_{i=1}^{d} \lambda_i \Delta z_i^2$$

For fixed value of the second order term, i.e. when $\sum_{i=1}^{d} \lambda_i \Delta z^2 = \Delta x^\top H \Delta x = C$ for constant C, we can rewrite the equation as $\sum_{i=1}^{d} \frac{\Delta z^2}{1/\lambda_i} = C$. For H that is semi-positive definite, $\sum_{i=1}^{d} \lambda_i \Delta z_i^2 \geq 0$, which means $\lambda_i \geq 0 \quad \forall i$. In this case, $\sum_{i=1}^{d} \frac{\Delta z^2}{1/\lambda_i} = C$ is an ellipsis with axis z_i for $i \in \{1, \dots, d\}$. Also note that large λ_i corresponds to the short axis, or the minor axis of the ellipsis in the 2D case. This will make more sense when we relate it to the Newton-Raphson algorithm. When some $\lambda_i < 0$, the second order term would characterize a saddle point, with the direction of Δz_i to be concave for $\lambda_i < 0$ and convex for $\lambda_i > 0$.

It's also worth mentioning that the projection of Δx to the subspace spanned by $\{q_i : i \in \{1, \dots, d\}\}$ is called analysis, while the reverse to be synthesis.

16.3 Relation to the Newton-Raphson Algorithm

Assume $H > 0$ and $g \neq 0$. We consider the second order Taylor expansion of $f(x)$ around x_0.

Let $\Delta = z - z_0 = Q^\top (x - x_0)$. Note that Q is orthonormal, so $QQ^\top = I$; then we have

$$\begin{aligned}
f(x) &\approx f(x_0) + \langle f'(x_0), x - x_0 \rangle + \frac{1}{2}(x - x_0)^\top f''(x_0)(x - x_0) \\
&= f(x_0) + \langle f'(x_0), QQ^\top (x - x_0) \rangle + \frac{1}{2}(x - x_0)^\top Q \Lambda Q^\top (x - x_0) \\
&= f(x_0) + \langle f'(x_0), Q(z - z_0) \rangle + \frac{1}{2}(z - z_0)^\top \Lambda (z - z_0) \\
&= f(x_0) + \langle Q^\top f'(x_0), (z - z_0) \rangle + \frac{1}{2}(z - z_0)^\top \Lambda (z - z_0) \\
&= f(x_0) + \sum_{i=1}^{d} (Q^\top f'(x_0))_i \Delta_i + \frac{1}{2} \sum_{i=1}^{d} \lambda_i \Delta_i^2 \\
&= \frac{1}{2} \sum_{i=1}^{d} \lambda_i (\Delta_i + (Q^\top f'(x_0))_i / \lambda_i)^2 + f(x_0) - \sum_{i=1}^{d} (Q^\top f'(x_0)_i / \lambda_i)^2
\end{aligned}$$

Since $H > 0$, $f(x)$ is convex at the neighborhood of x_0, and so it reaches the local minimum at the center of the ellipsis (the first term above is of the form $\sum \lambda_i \alpha_i^2$, which characterizes an ellipsis with axes to be α_i's). Therefore, at the local minimum, we have $\Delta_i + (Q^\top f'(x_0))_i / \lambda_i = 0$ for all i. Pack the equations together, $\Delta + \Lambda^{-1} Q^\top f'(x_0) = 0$, or $\Lambda Q^\top (x - x_0) + Q^\top f'(x_0) = 0$. Therefore, we get $Q\Lambda Q^\top (x - x_0) + f'(x_0) = 0$ ($QQ^\top = I$), or $f''(x_0)(x - x_0) + f'(x_0) = 0$, so

$$x_1 = x_0 - f''(x_0)^{-1} f'(x_0),$$

where x_1 is the minimum of the second order Taylor expansion. Note that this is of the same form with the Newton-Raphson algorithm's update.

From the update rule above, the geometric understanding of the first and second order terms of the Taylor expansion makes sense. We update x in the direction of $f'(x)$, and adjust the step size according to the curvature $f''(x)$. For the steep direction with large curvature, we want to adjust our step size to be smaller to prevent overshoot; for the direction with small curvature, where the function changes gradually, we want to adjust the step size to be larger.

16.4 Second Order Taylor Expansion as the Surrogate Function

In some optimization problem, optimizing the original problem would be very hard. In these scenarios, we might want to use the second order Taylor expansion to construct a surrogate function to approximate the local optimal point.

For example, in the gradient descent algorithm, we minimize the surrogate function

$$s(x) = f(x_0) + \langle f'(x_0), x - x_0 \rangle + \frac{1}{2\eta}|x - x_0|^2$$

to obtain x_1. We know that $s(x) = f(x_0) + \langle f'(x_0), x - x_0 \rangle + \frac{1}{2\eta}|x - x_0|^2 = \frac{1}{2\eta}|x - (x_0 - \eta f'(x_0))|^2 + constant$, so x should be updated to $x = x_0 - \eta f'(x_0)$ in the gradient descent iteration.

Note that comparing with Newton-Raphson algorithm, we fix the step size here since the hessian matrix H is computationally expensive. In fact, in some variants of the gradient descent, the momentum can be understood to be an approximation of the curvature.

In the boosting algorithms, we also use the second order Taylor expansion as a surrogate. The surrogate function can be rearranged to be a weighted least square loss, which is familiar to us and relatively easier to optimize.