Evolution of Neural Network

Table of contents

Chapter 1

From Perceptron to Feed-Forward Neural Network

1.1 Perceptron Is an Abstracted Mathematical Model of Neural Network

1.1.1 Biology of Neuron

In this section, we introduce neuron from neuroscience perspective. This will not be a thorough introduction, but aiming to build up an abstract mathematical model for representing the network of neurons.[1.1]

Like other kinds of cell, neuron has cell body which includes many kinds of organelles, packaged by cell membrane. It looks like a house with furniture everywhere. Houses all have sofas, lights, and kitchen wares. And cells all have nucleus, Golgi complex, and mitochondria. There are, however, two critical differences between neuron and other kinds of cell. The first difference is that there is electric potential difference between the inside and outside of the cell membrane: the wall of the house is electrostatic. The second property that is unique for neuron is its shape. Typically, a neuron has tree-like extensions from its cell body, called *dendrites*, and an extremely long cable-like extension from its cell body, called *axon*. Altogether, neurons can communicate with each other via electrical signals, called *impulses*.

Let us start our journey in cell body. Electrical signals received from dendrites are collected in the cell body. These signals will change the potential difference of the cell membrane. This change is called *depolarization*. When depolarization exceeds a threshold, impulses will generate in the intersecting area between cell body and axon, called *firing*. Like all cables carry the same voltage, all impulses propagating **along axon** share the same strength, rushing toward the far-end of axon. In the far-end, a cable is separated and connected to each household. So is an axon, separating into many terminals, called *synapses*. Each synapse connects to a dendrite of another neuron, for transmitting electrical signals to it.

Interestingly, the electric signals will not be propagated to the successive neuron directly. Instead, they are converted to chemical signals, called *neurotransmitters*. These neurotransmitters are molecules released from synapse. They are then caught by the receptors on the dendrite of the successive neuron. These receptors will convert the chemical signals back to electrical signals in the dendrite. Why does Nature use such a complex way to propagate electrical signals? Because it is adaptable. The more receptors on the dendrite, the greater strength of electrical signals received on the dendrite. It is potential transformer in neuron! These electrical signals received in the dendrite are then collected in the cell body of the successive neuron, going back to the start of our journey. As a summary, our journey went through: depolarization $\rightarrow$ firing $\rightarrow$ impulses $\rightarrow$ synapses $\rightarrow$ neurotransmitters $\rightarrow$ densities $\rightarrow$ depolarization $\rightarrow$ firing $\rightarrow\cdots$.

1.1. For more details about neurons, see *Principles of Neural Science (6th Edition)* by Eric Kandel and others. In this book, part 2 describes the structure of neurons and how impulses are generated and propagated along axon; and part 3 explains the details of how electrical signals are transmitted from synapse to the dendrite of another neuron.

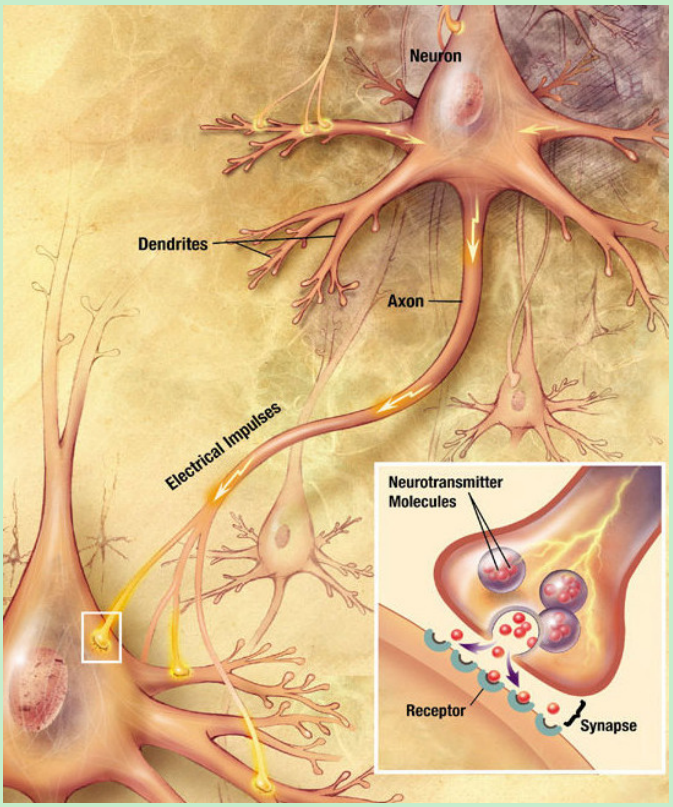

Figure 1.1. This figure illustrates the shape of a neuron, and how impulses propagate from one neuron to another.

1.1.2 Mathematical Abstraction

From this journey, we highlight the following key facts:

- Each neuron receives impulses from other neurons that connect to it via axons.
- Impulses propagating along axon share the same strength.
- The strength of electrical signals is changed when passing from synapses to densities.
- Depolarization is the total effect of the received impulses from other neurons.
- To fire impulses, a neuron has to be sufficiently depolarized, exceeding a threshold.

These facts indicates a mathematical model that simulates the network of neurons. In a neural network with n neurons, the state of neurons, firing or not firing, is characterized by a n-dimensional binary vector, $x \in \{0,1\}^n$, where 0 represents for not firing, and 1 for firing. The state changes with time steps $t = 0, \ldots, T$. At time step t, suppose neuron β fires, thus $x^\beta(t) = 1$. The strength of impulses propagating along axon, sharing the same value, is normalized to 1. When the signal passes from a synapse of neuron β to a dendrite of neuron α, it is adapted by a parameter W_β^α, called *weight*, characterizing the number of receptors on the dendrite. For neuron α, the electrical signal received from neuron β is thus W_β^α, which can be re-written as $W_\beta^\alpha\, x^\beta(t)$ since $x^\beta(t) = 1$. In the next time step, the depolarization of a neuron α is the collection of all (adapted) electrical signals received from its dendrites, thus $\sum_{\beta=1}^n W_\beta^\alpha\, x^\beta(t)$ ($W_\beta^\alpha = 0$ if there is not connection from neuron β to neuron α). The threshold of firing in neuron α is characterized by a parameter b^α, called *bias*. So, we have, in the next time step, the state of neurons comes to be

$$x^\alpha(t+1) = \Theta\left(\sum_{\beta=1}^{n} W_\beta^\alpha\, x^\beta(t) + b^\alpha\right), \tag{1.1}$$

where t represents for the current time step and $t+1$ the next; the Θ, called *activation function*, represents the step function: $\Theta(x<0)\equiv 0$ and $\Theta(x\geqslant 0)\equiv 1$. This model is called *perceptron*.

1.2 Perceptron Should Be Used as Inter-layer

After building up the model, let us use it to simulate a real neural network, such as a brain. Suppose we are given a sequence of observed states of a real neural network, $\{\hat{x}(t)|t=0,\dots,T\}$, as dataset. Our aim is to simulate this sequence with the $x(t)$ in equation (1.1) by adjusting the parameters W and b. The $x(t)$ has initial value $\hat{x}(0)$. From $x(0)$, we compute $x(1)$ by equation (1.1), then $x(2)$, and then $x(3)$, etc. We hope that, there exists proper parameters W and b, such that $x(t)\equiv\hat{x}(t)$ for all time steps.

Unfortunately, our dream may not come true. For example, consider a situation where $\hat{x}^1(t+1)=\mathrm{XOR}(\hat{x}^1(t),\hat{x}^2(t))$ at some time step t. As Marvin Minsky figured out in 1969, a perceptron cannot fit or represent an XOR function.[1.2] So, we cannot have $x^1(t+1)=\mathrm{XOR}(x^1(t),x^2(t))$, thus we cannot hope $x(t)\equiv\hat{x}(t)$.

From 1989 to 1991, researchers found that if we compose a perceptron with a linear transformation, it then obtains the ability to simulate any binary process.[1.3] Explicitly, let

$$z^\alpha=\Theta\left(\sum_{\beta=1}^{m}U^\alpha_\beta\,x^\beta+a^\alpha\right)\quad\text{and}\quad y^\alpha=\Theta\left(\sum_{\beta=1}^{n}W^\alpha_\beta\,z^\beta+b^\alpha\right).\tag{1.2}$$

The first equation represents a perceptron with input x; the second also a perceptron that takes the output of the previous perceptron (namely, the z) as input. Altogether, it becomes a model that accepts x as input and outputs y. The first perceptron is not used for output, but as an inter-layer. The dimension of z (namely, the m) is a hyper-parameter. When m increases, the model has more adaptive components of parameter. Model capacity has benefit from the extensible parameter space (characterized by m). It was proven that this model can simulate any process, including XOR function, as long as the m is large enough.[1.4]

For simulating any process, which may not be binary, we have to generalize perceptron from dealing with binary value to manipulate real number. This means, the activation function of the perceptron (at least for the second perceptron that gives the final output) shall output a continuous spectrum on the real number field. For this purpose, we shall generalize the activation function to be continuous and monotonically increasing. Frequently used activation functions include tanh, sigmoid[1.5], ReLU[1.6], softplus[1.7], and even identity function[1.8]. In the subsequent, the word perceptron refers to that with the generalized activation function.

1.3 Simulation Is a Kind of Data-Fitting

From the previous discussion, we find that the problem of capability of simulation can be converted to the problem of capability of data-fitting. Indeed, the capability of simulating a real neural network was converted to the capability of fitting arbitrary binary dataset by perceptron. To extend our discussion, we have to declare what is the capability of data-fitting precisely.

1.2. See Marvin Minsky's book *Perceptrons: an introduction to computational geometry*.

1.3. A brief history can be found here.

1.4. A wonderful visual proof is given by Michael Nielsen in his book *Neural Networks and Deep Learning*, chapter 4.

1.5. Sigmoid function is a "soft" version of Θ. We have $\mathrm{sigmoid}(x):=1/(1+\exp(-x))$.

1.6. We have $\mathrm{ReLU}(x):=0$ if $x<0$ and $\mathrm{ReLU}(x):=x$ if $x\geqslant 0$.

1.7. We have $\mathrm{softplus}(x):=\ln(1+\exp(x))$. It is the smooth version ReLU.

1.8. Namely, $\mathrm{id}(x)=x$.

First of all, what is data-fitting? A dataset is a collection of input-output pairs. For example, in the handwriting recognition dataset, the input is a handwriting image of a number, and the output is the number. So, a dataset is generally represented as $D := \{(x_i, y_i) | i = 1, \dots, N\}$ where x_i represents for the input and y_i for the output. Data-fitting searches for a model that accepts an input (a new handwriting image) and predicts the output (what is number on the handwriting image) successfully.

Now, what is a model (we have used this word several times without declaration)? Shortly, a model is a parameterized function in a truncated parameter space. As an example, consider the polynomial $f(x;\theta) = \theta_0 + \theta_1 x + \theta_2 x^2 + \cdots$. It is an infinite series. Regarding x as variable and θ as coefficients (or parameter), $f(x;\theta)$ can be seen as a Taylor expansion at $x = 0$, and it can represent any smooth function within the convergence domain. But such parameterized function with infinitely many parameter components is impractical, because computer cannot process infinity. We shall truncate the parameter to some finite m, namely $\theta_n = 0$ for any $n > m$. So, a model is a truncated parameterized function in an infinite-dimensional parameter space.

Then, how to characterize the success of prediction? After accepting an input x, a model gives an output $\hat{y}$. We say the prediction is successful if the error of output is sufficiently small. Precisely, let y the output of x in the dataset, then the error is given by $d(\hat{y}, y)$, where d is a kind of "distance" with $d(\hat{y}, y) \geqslant 0$ and $d(\hat{y}, y) = 0$ only when $\hat{y}$ and y are equal.

So, we say $f(x;\theta)$ has the capability of fitting the dataset D, if for any $\varepsilon > 0$, we can find a truncated parameter $\theta_\star$ in the parameter space such that

$$\sum_{(x,y)\in D} d(f(x;\theta_\star), y) < \varepsilon. \tag{1.3}$$

In the previous Taylor expansion example, we have known from calculus that the error (residue) decreases when we increase the truncation m. Intuitively, the capacity of the model $f(x;\theta)$ increases with the increment of m.

Going back to perceptrons, the parameter is (U, W, a, b), and the model is given by equation 1.2. Here, m serves as the truncation. The single perceptron 1.1 does not fit this definition, since its parameter space has been bounded by the input and output dimensions.

Chapter 2
Gradient Based Optimization

2.1 The Objective Is the Expected Distance Between Prediction and Truth

As discussed in section 1.3, the performance of simulation or, generally, data-fitting can be numerically characterized. Recall in that section, a dataset $D := \{(x_i, y_i) | i = 1, \dots, N\}$ is given, as well as a parameterized function (called a model) $f(x; \theta)$ with x the input and θ the collection of parameters. For each x_i in the dataset, we have $\hat{y}_i := f(x_i; \theta)$ which can be explained as the prediction of the model when its parameter is θ. If the model has good performance, we should expect that its prediction $\hat{y}_i$ is very close to the truth or target y_i. This can be characterized by a "distance" $d(\cdot, \cdot)$ for which $d(x, y) \geqslant 0$ and $d(x, y) = 0$ if and only if $x = y$. That is, $d(y_i, \hat{y}_i)$ shall be small enough. This is the demand for one datum, (x_i, y_i). Since each datum is equally weighted (we suppose so), we shall characterize the total performance by $\sum_{(x_i, y_i) \in D} d(y_i, f(x_i; \theta))$ which should be small enough.

But, when the size of dataset D is extremely large, computing $\sum_{(x_i, y_i) \in D} d(y_i, f(x_i; \theta))$ will become difficult. In practice, we sample sufficient many data from D, and compute the expectation of $d(y, f(x; \theta))$ on these samples instead. These sampled data are called **mini-batch**. (The whole dataset D is thus named by **full-batch**.) Thus, a proper quantity that characterizes performance is

$$L_D(\theta) := \mathbb{E}_{(x,y) \sim D}[d(y, f(x; \theta))], \tag{2.1}$$

where $\mathbb{E}_{(x,y) \sim D}$ represents the expectation with (x, y) uniformly sampled from D.

L_D is called a *loss function*, since the best model parameter $\theta_\star$ is that which minimizes the expected distance L_D. Thus,[2.1]

$$\theta_\star = \operatorname{argmin} L_D.$$

The important thing is that, in the usual situation of neural network, the function L_D is a smooth function of θ, and that θ is on a finite-dimensional Euclidean space. This means, we can use gradient descent method to find the minimum of L_D without any constraint.

2.2 Moving Average Helps Avoid Stochastic Disturbance

2.2.1 Stochastic Disturbance in Loss Function

Since the loss function is computed on mini-batch instead of the whole dataset D, there must be stochastic disturbance. Let B a random subset of D, a mini-batch. By central limit theorem, the mean value of $d(y, f(x; \theta))$ over B approximately obeys a normal distribution. The expectation of this distribution is the mean value of $d(y, f(x; \theta))$ over D, the loss function on full-batch. And the variance is proportional to $1/|B|$, where $|B|$ denotes the number of elements in B. So, we can re-write the loss function L_D as

$$L_D(\theta) = \hat{L}_D(\theta) + \delta L_D(\theta),$$

2.1. This is not the whole story. In fact, $\theta_\star = \operatorname{argmin} L_D$ is not what we want. Generally, the performance is examined not on the training dataset D, but on another similar dataset T, called test dataset. While the loss function $L_D(\theta)$ decreases in the training process, the loss function on test dataset, $L_T(\theta)$ may start to increase at some point θ_{ES} (ES for "early-stopping"). That is,

$$\nabla L_D(\theta_{\mathrm{ES}}) \cdot \nabla L_T(\theta_{\mathrm{ES}}) = 0.$$

We must stop training and use θ_{ES} as the best-fit parameters. In this book, we try to keep things simple. Thus, we omit this complicate thing, only thinking about decreasing the loss function L_D.

where $\hat{L}_D$ represents the mean value over the full-batch D, and $\delta L_D(\theta)$ is the random disturbance, which approximately obeys a normal distribution with zero mean. For the same reason (central limit theorem), the gradient $\nabla\delta L_D(\theta)$ also approximately obeys a normal distribution with zero mean and variance proportional to $1/|B|$.

2.2.2 Gradient Descent Method

For minimizing a function $h\colon \mathbb{R}^n \to \mathbb{R}$, standard gradient descent method computes the gradient of h, and iterates along the negative direction of gradient so as to decrease h at each iteration. Explicitly, let $t \in \mathbb{N}$ denotes the step of iteration, thus the variable at step $t+1$ is given by

$$x_{t+1} = x_t - \eta \nabla h(x_t), \tag{2.2}$$

where the η is a fixed positive number, called **learning rate**. If h is smooth, then we have $h(x_{t+1}) < h(x_t)$ as long as the learning rate is sufficiently small and $\nabla h(x_t) \neq 0$. Thus, the iteration (2.2) always decrease h until the (maybe local) minimum.

Problems arise when applying gradient descent method directly to minimize L_D because of the random disturbance δL_D. What we really want to minimize is the deterministic $\hat{L}_D$, the loss function on the full-batch, but what we can obtain is the L_D instead of $\hat{L}_D$. We hope that, iterated by the gradient descent method 2.2, the trajectory $(\theta_0, \theta_1, \dots)$ generated by ∇L_D (what we can compute) and the $(\hat{\theta}_0, \hat{\theta}_1, \dots)$ by $\nabla \hat{L}_D$ (what we expect to compute but cannot) share the same limit $\theta_\star$, the real best-fit value. Only when $\nabla L_D(\theta)$ is sufficiently close to $\nabla \hat{L}_D(\theta)$ can this be done, which indicates that we have to reduce the randomness from $\nabla\delta L_D$.

2.2.3 Moving Average of Gradient

An efficient method for reducing randomness is averaging. Let $\{X_i | i = 1, \dots, n\}$ a set of i.i.d. random variables, each having variance $\mathrm{Var}[X]$. By central limit theorem, the variance of the averaged, $(1/n)\sum_i X_i$, is decreased by a factor $1/n$, thus $\mathrm{Var}[X]/n$. The same, we cache the most recent n gradients $\{\nabla L_D(\theta_{t-n+1}), \dots, \nabla L_D(\theta_t)\}$ while iterating the θ (in the same way as the x_t in previous) at step t. Then, average over the cache to get the gradient used for iteration, $(1/n)\sum_{i=t}^{t-n+1} \nabla L_D(\theta_i)$. In this way, the variance of randomness caused by $\nabla\delta L_D$ is decreased by a factor $1/n$. By adjusting the value of n, the randomness can be limited sufficiently.

This "bare" average calls for caching the most recent gradients. It is very memory intensive when the dimension of θ goes high. A smarter one is **moving average**: given $\gamma \in [0, 1]$, the moving average of $\nabla L_D(\theta_t)$, denoted by g_t, is computed by iteration

$$g_t = \gamma g_{t-1} + (1-\gamma)\nabla L_D(\theta_t), \tag{2.3}$$

with initialization $g_0 = 0$. The γ, called **decay factor** or **forgetting factor**, determines how many old information of gradient, g_{t-1}, is to be "forgotten", and how many new information of gradient, $\nabla L_D(\theta_t)$, is to be "memorized". The g_t can be seen as a weighted average of $\{\nabla L_D(\theta_0), \dots, \nabla L_D(\theta_t)\}$, where the recent gradients have greater weights and the remote have less. Then, we iterate the θ_t by g_t instead of $\nabla L_D(\theta_t)$, as

$$\theta_{t+1} = \theta_t - \eta g_t. \tag{2.4}$$

2.2.4 Finetune Decay Factor and Learning Rate

When γ is close to 0, the moving average g_t is easy to forget the old information of ∇L_D. Indeed, the factor $\gamma \ll 1$ in equation (2.3) implies $g_t \approx \nabla L_D(\theta_t)$, no averaging at all. So, for make the moving average effective, γ shall not be too small.

On the contrary, when γ is close to 1, the moving average g_t is hard to "accept" new information of ∇L_D. Indeed, the factor $(1-\gamma) \ll 1$ in equation (2.3) implies $g_t \approx g_{t-1}$ for all t. So, the moving average is hard to be modified when γ is close to 1. This is equivalent to a large learning rate, leading to an ascend of loss function instead of descent.

So, for efficiently and safely using moving average, the γ shall be moderate. And if, in practice, the $\nabla\delta L_D$ is so large that the moving average can be effective only when γ is close to 1, then we shall accordingly tune the learning rate η to be smaller, so as to decrease the loss function safely.

2.2.5 History and Remark

Moving average of gradient was first applied to gradient descent in 1986.[2.2] Later, the efficiency of moving average was explained as avoiding getting stucked by local minima. They compared moving average of gradient to the momentum in physics: the "heavy ball" rushes out of a local minimum with large "momentum". But, in a space with extremely high dimension, it is rare to encounter a local minimum, but saddle points instead. So, this explanation cannot be true. The underlying insight is that it is easy for the eigenvalues of random symmetric matrix (Hessian matrix can be seen as one) to be all positive when it is two dimensional, but being exponentially harder when the dimension increases. In fact, the eigenvalue of a random symmetric matrix obeys the Wigner semicircle distribution, when the dimension goes to infinity. This distribution is parity symmetric.

2.2.6 Implementation

We summarize this section by a `Numpy` implementation.

```
def moving_average(loss_gradient, initial_theta, decay_factor,
                   learning_rate, steps):
    # Initialization:
    theta = initial_theta
    g = np.zeros(np.shape(theta))
    # Iteration:
    for t in range(steps):
        g = decay_factor * g + (1-decay_factor) * loss_gradient(theta)
        theta = theta - learning_rate * g
    return theta
```

2.3 Gradient Direction May Not Be Optimal (TODO)

2.3.1 Estimation of Gradients at Different Layer

Iterating along the negative direction of the (moving average of) gradient ∇L_D may not be optimal. To address this issue, let us consider a simple instance and compute the gradients at different layers.

Consider a feed-forward neural network with a single hidden layer. The M-dimensional model output is

$$y^\alpha := f^\alpha(x;\theta) = \sum_{\beta=1}^{H} U_\beta^\alpha z^\beta + c^\alpha$$

with the H-dimensional output of hidden layer

$$z^\beta := \sigma\left(\sum_{\gamma=1}^{N} W_\gamma^\beta x^\gamma + b^\beta\right),$$

where $x \in \mathbb{R}^N$ is model input, $\theta := (U, W, b, c)$ is the collection of model parameters, and σ is the activation function. Thus, we have gradient

$$\frac{\partial L_D}{\partial U_\beta^\alpha} = \frac{\partial L_D}{\partial y^\alpha}\frac{\partial y^\alpha}{\partial U_\beta^\alpha},$$

2.2. *Learning representations by back-propagating errors*, by David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams, 1986.

and

$$\frac{\partial L_D}{\partial W_\beta^\alpha} = \sum_{\gamma=1}^{M} \frac{\partial L_D}{\partial y^\gamma} \frac{\partial y^\gamma}{\partial z^\alpha} \frac{\partial z^\alpha}{\partial W_\beta^\alpha}.$$

We are to estimate the relation of orders between $\mathrm{Var}[\partial L_D / \partial U]$ and $\mathrm{Var}[\partial L_D / \partial W]$. Generally, the output of each layer is properly normalized, which is ensured by initialization and normalization (we will discuss normalization in section 3.7.3). Thus, $\partial y^\alpha / \partial U_\beta^\alpha = z^\beta$ and $\partial z^\alpha / \partial W_\beta^\alpha = \sigma'(\sum_\gamma W_\gamma^\alpha x^\gamma + b^\alpha) x^\beta$ share the same order which is $\mathcal{O}(1)$. (For example, when σ represents ReLU activation function, $\sigma'(\cdot) \in \{0, 1\}$ is roughly estimated as $1/2$, thus $\partial z^\alpha / \partial W_\beta^\alpha = \sigma'(\cdots) x^\beta = \mathcal{O}(1)$.) So, we have a rough estimation

$$\begin{aligned}
\mathrm{Var}\left[\frac{\partial L_D}{\partial U}\right] &\sim \mathrm{Var}\left[\frac{\partial L_D}{\partial y}\right]; \\
\mathrm{Var}\left[\frac{\partial L_D}{\partial W}\right] &\sim M\,\mathrm{Var}\left[\frac{\partial L_D}{\partial y}\right] \mathrm{Var}\left[\frac{\partial y}{\partial z}\right] \\
&= M\,\mathrm{Var}\left[\frac{\partial L_D}{\partial y}\right] \mathrm{Var}[U],
\end{aligned}$$

where, in the last line, we used the relation $\partial y / \partial z = U$. The variance is computed over components. The M factor is estimated by central limit theorem where we roughly regard $\partial L_D / \partial W$ and U as i.i.d. random variables. Generally, U is initialized with variance $6/(M+H)$.[2.3] Thus, $M\,\mathrm{Var}[U] \sim M/(M+H) = (1 + H/M)$. When $H \gg M$, indicating that the hidden dimension is much larger than the output dimension, which is usually the situation we encounter, we have $M\,\mathrm{Var}[U] \ll 1$, leading to $\mathrm{Var}[\partial L_D / \partial U] \gg \mathrm{Var}[\partial L_D / \partial W]$. So, in some general situations, gradient varies greatly across layers.

2.3.2 Large Difference of Gradients May Slow Down Optimization

The large difference between the order of $\partial L_D / \partial U$ and $\partial L_D / \partial W$ may slow down the optimization. We are to explain the reason by estimating some orders.

Let U_0 the initial values of U, and $U_\star$ the best-fit. They share the same order, since we initialize the parameters by (very roughly) estimating the order of best-fit, so as to speed up the optimization. That is, $U_0 \sim U_\star \sim (U_\star - U_0)$. The same for W, and we have $W_0 \sim W_\star \sim (W_\star - W_0)$. Explicity, U_0 has order $1/(M+H)$, and W_0 has order $1/(H+N)$. Generally, the hidden dimension is much greater than the input and output dimensions, or say $H \gg N, M$. Thus $U_0 \sim W_0 \sim 1/H$, so is $(U_\star - U_0) \sim (W_\star - W_0)$.

If, as previously estimated, $\mathrm{Var}[\partial L_D / \partial U] \gg \mathrm{Var}[\partial L_D / \partial W]$, then there will be much more steps for W to iterate than for U, because W walks much slower than U even though they have to walk the same (order of) distance. The dillydallying W slows down the optimization.

2.3.3 Rescale by Standard Derivation

One way to reduce the large difference between the order of $\partial L_D / \partial U$ and of $\partial L_D / \partial W$ is dividing by their standard derivations. That is, using $(\partial L_D / \partial U) / \sqrt{\mathrm{Var}[\partial L_D / \partial U]}$ and $(\partial L_D / \partial W) / \sqrt{\mathrm{Var}[\partial L_D / \partial W]}$ instead of $\partial L_D / \partial U$ and $\partial L_D / \partial W$ for gradient descent. Then,

$$\mathrm{Var}\left[\frac{\partial L_D}{\partial U} \mathrm{Var}\left[\frac{\partial L_D}{\partial U}\right]^{-1/2}\right] = \mathrm{Var}\left[\frac{\partial L_D}{\partial U}\right] \mathrm{Var}\left[\frac{\partial L_D}{\partial U}\right]^{-1} = 1,$$

and the same for $\partial L_D / \partial W$. Now, the orders are equal.

2.3. This is the Glorot initialization. We will not discuss the initialization techniques, but refer the reader to the paper by Xavier Glorot and Yoshua Bengio: *Understanding the difficulty of training deep feed-forward neural networks*.

Generally, denote the parameters in each layer as θ_l. Then θ is the concatenation of the θ_ls for all layers. While iterating θ by gradient descent method (2.2), replace $\partial L_D / \partial \theta_l$ by

$$\frac{\partial L_D}{\partial \theta_l^\alpha} \to \frac{\partial L_D}{\partial \theta_l^\alpha} \operatorname{Var}\left[\frac{\partial L_D}{\partial \theta_l}\right]^{-1/2}, \tag{2.5}$$

where $\operatorname{Var}[\partial L_D / \partial \theta_l]$ is computed over components, which consist of $\partial L_D / \partial \theta_l^\alpha$ for all α.

2.3.4 Implementation

We summarize this section by a `Numpy` implementation. In this implementation, we also use moving average to reduce the stochastic disturbance of gradient.

```
def rescale_gradient(loss_gradient, initial_theta_list, decay_factor,
                     learning_rate, steps):
    """The initial_theta_list contains the initial parameters for each layer."""
    # Initialization:
    theta_list = [theta for theta in initial_theta_list]
    g_list = [np.zeros(np.shape(theta)) for theta in theta_list]
    # Iteration:
    for t in range(steps):
        for i, theta in enumerate(theta_list):
            # Moving average
            g = decay_factor * g + (1-decay_factor) * loss_gradient(theta)
            # Iterate with rescaled gradient
            theta_list[i] = theta - learning_rate * g / np.std(g)
    return theta_list
```

2.4 Using the Sign of Gradient (TODO)

2.4.1 History and Remark

The idea of using the sign of gradient is proposed by Martin Riedmiller and Heinrich Braun in 1992. In their `Rprop` (short for resilient back-propagation) algorithm, TODO. But, `Rprop` algorithm cannot deal with mini-batch which indicates stochastic gradient. TODO

Later in 2012, James Martens and Ilya Sutskever generalized the `Rprop` algorithm to stochastic gradient and made it much simpler. The new algorithm is called `RMSprop`. As it is named, it employs root mean square (RMS for short) for computing the sign of gradient, which helps stabilize the stochastic gradient. TODO The s^α decreases slowly when $|g^\alpha|$ decreases, but increases quickly when $|g^\alpha|$ increases.

Finally, in 2014, by adding moving average to the gradient, RMSprop algorithm is further improved, now called Adam algorithm.

2.4.2 Implementation

We summarize this section by a `Numpy` implementation.

```
def signprop(loss_gradient, initial_theta, decay_factor,
             learning_rate, steps):
    # Initialization:
    theta = initial_theta
    g = np.zeros(np.shape(theta))
    # Iteration:
    for t in range(steps):
        # Moving average
        g = decay_factor * g + (1-decay_factor) * loss_gradient(theta)
        # Iterate by sign of gradient
        theta = theta - learning_rate * np.sign(g)
    return theta
```

2.5 Gradient Is Computed by Vector-Jacobian Product * [2.4]

2.5.1 From Feed-Forward Neural Network to General Composition

Recall in section ?, we have shown that a feed-forward neural network is a composition of multiple perceptrons. For example, for a two-layer feed-forward neural network, we have

$$z^\alpha = h\left(\sum_{\beta=1}^{n} U_\beta^\alpha x^\beta + c^\alpha\right)$$

and

$$y^\alpha = g\left(\sum_{\beta=1}^{m} W_\beta^\alpha z^\beta + b^\alpha\right).$$

If we use an f to denote the model, say $f(x;\theta)$, then θ represents for the collection (U, c, W, b), and

$$f(x;U,c,W,b) = g\left(\sum_{\beta=1}^{m} W_\beta^\alpha h\left(\sum_{\beta=1}^{n} U_\beta^\alpha x^\beta + c^\alpha\right) + b^\alpha\right),$$

which is a composition of two parameterized functions. In fact, almost all existing neural networks, from computer vision to natural language process, are compositions of simple parameterized functions, like f. So, we shall consider general composition like

$$f_\theta := g_{\theta_n} \circ \cdots \circ g_{\theta_2} \circ g_{\theta_1},$$

where, for simplicity, we have placed parameters onto the subscripts, thus $g_\theta(x) := g(x;\theta)$ and $f_\theta(x) := f(x;\theta_1,\ldots,\theta_n)$.

2.5.2 Vector-Jacobian Product

Computer calculates the derivative such as $\partial L_D / \partial \theta_i$ by *vector-Jacobian product* (VJP for short). For declare what vector-Jacobian-product is, let us consider a simple example. We are to define the vector-Jacobian product of sigmoid function, which is defined by $y^\alpha = 1/(1+\exp(-x^\alpha))$. By chain-rule, we have

$$\frac{\partial y^\alpha}{\partial x^\beta} = y^\alpha (1 - y^\alpha)\, \delta_\beta^\alpha$$

And for any vector v^β, we have

$$\sum_\beta v^\beta \frac{\partial y^\alpha}{\partial x^\beta} = v^\alpha y^\alpha (1 - y^\alpha).$$

The vector-Jacobian product of sigmoid function has `Numpy` implementation as

```
def sigmoid_vjp(x):
    y = 1 / (1 + np.exp(-x))
    def grad(dy):
        return dy * y * (1-y)
    return y, grad
```

It returns the output of sigmoid function, as well as a function `grad`. It can be recognized that `grad(dy)` encodes the $\sum_\beta v^\beta \partial y^\alpha / \partial x^\beta$, where v is represented by the `dy`. So, a vector-Jacobian product of a function f returns two parts, one is the output value of the function $f(x)$, the other is a function that computes $\sum_\beta v^\beta (\partial f^\alpha / \partial x^\beta)(x)$. If f has multiple variables, then vector-Jacobian product shall return a function $\sum_\beta v^\beta (\partial f^\alpha / \partial x_i^\beta)(x_1,\ldots,x_n)$ for each variable x_i.

Using matrix format is convenient. Thus re-write

$$\sum_\beta v^\beta \frac{\partial f^\alpha}{\partial x^\beta}(x) \to v \cdot \frac{\partial f}{\partial x}(x).$$

2.4. You can skip this section if you are not care about how computer calculates derivative.

2.5.3 Forward Propagation

Recall that $f_\theta := g_{\theta_n} \circ \cdots \circ g_{\theta_2} \circ g_{\theta_1}$. Thus, for computing the value of $f_\theta(x)$ for a datum $(x, y) \in D$, we shall first compute $g_{\theta_1}(x)$. But, if we have defined the vector-Jacobian product of g_{θ_1}, we will get $z_1 := g_{\theta_1}(x)$, as well as functions $\mathrm{d}g_{\theta_1}(v) := v \cdot (\partial z_1 / \partial x)$ and $\mathrm{d}'g_{\theta_1}(w) := w \cdot (\partial z_1 / \partial \theta_1)$. The same for g_{θ_i}, we get $z_i := g_{\theta_i}(z_{i-1})$ as well as $\mathrm{d}g_{\theta_i}(v) := v \cdot (\partial z_i / \partial z_{i-1})$ and $\mathrm{d}'g_{\theta_i}(w) := w \cdot (\partial z_i / \partial \theta_i)$. And the model prediction $\hat{y} := g_{\theta_n}(z_{n-1})$ as well as $\mathrm{d}g_{\theta_n}(v) := v \cdot (\partial \hat{y} / \partial z_{n-1})$ and $\mathrm{d}'g_{\theta_n}(w) := w \cdot (\partial \hat{y} / \partial \theta_n)$. Finally, if we have also defined the vector-Jacobian product of the distance d, then we will get $l = d(\hat{y}, y)$ where y is the truth, as well as $\mathrm{d}l(v) := v \cdot (\partial l / \partial \hat{y})$ and $\mathrm{d}'l(v) := v \cdot (\partial l / \partial y)$. The l is the loss for this datum (x, y). This process, computing vector-Jacobian products from input to output, is called *forward propagation*.

These values and functions are then cached, waiting for the next process, called backward propagation.

2.5.4 Backward Propagation

Backward propagation is for computing the derivatives $\partial l / \partial \theta_{n-i}^\alpha$ for each $i = 0, \ldots, n-1$. By chain-rule

$$\frac{\partial l}{\partial \theta_i^\alpha} = \sum_\beta \frac{\partial l}{\partial \hat{y}^\beta} \sum_\gamma \frac{\partial \hat{y}^\beta}{\partial z_{n-1}^\delta} \sum_\delta \frac{\partial z_{n-1}^\gamma}{\partial z_{n-2}^\delta} \cdots \sum_\varphi \frac{\partial z_{n-i+1}^\varepsilon}{\partial z_{n-i}^\varphi} \frac{\partial z_{n-i}^\varphi}{\partial \theta_{n-i}^\alpha},$$

or in matrix format

$$\frac{\partial l}{\partial \theta_i} = \frac{\partial l}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_{n-1}} \cdot \frac{\partial z_{n-1}}{\partial z_{n-2}} \cdot \cdots \cdot \frac{\partial z_{n-i+1}}{\partial z_{n-i}} \cdot \frac{\partial z_{n-i}}{\partial \theta_{n-i}},$$

where, on the right hand side, the first term is a vector, while the others are all Jacobian matrices.

We recognize that this is a chain of vector-Jacobian products. First, the first two terms, $(\partial l / \partial \hat{y}) \cdot (\partial \hat{y} / \partial z_{n-1})$, are in fact the $\epsilon_n := \mathrm{d}g_{\theta_n}(\partial l / \partial \hat{y})$, which is a vector. Thus, the first three terms are $\epsilon_{n-1} := \mathrm{d}g_{\theta_{n-1}}(\epsilon_n)$, which is another vector. Repeating this process, we find that

$$\begin{aligned}
\epsilon_n &:= \mathrm{d}g_{\theta_n}(\partial l / \partial \hat{y}); \\
\epsilon_{n-1} &:= \mathrm{d}g_{\theta_{n-1}}(\epsilon_n); \\
\epsilon_{n-2} &:= \mathrm{d}g_{\theta_{n-2}}(\epsilon_{n-1}); \\
&\cdots\cdots \\
\epsilon_{n-i+1} &:= \mathrm{d}g_{\theta_{n-i+1}}(\epsilon_{n-i+2}); \\
\frac{\partial l}{\partial \theta_{n-i}} &= \mathrm{d}'g_{n-i}(\epsilon_{n-i-1}).
\end{aligned}$$

So, by computing the value of the function obtained from vector-Jacobian product along the backward direction, from output toward input, we get the $\partial l / \partial \theta_{n-i}$. This is called *backward propagation*. Notice that the ϵ_i is in fact the derivative $\partial l / \partial z_{i-1}$.

After backward propagation, the cached values and functions can be freed.

As a summary, the whole process is: first computing and caching vector-Jacobian products forwardly, and then applying the functions given by vector-Jacobian products to derivatives backwardly.

2.6 TODO

2.6.1 Initialization

Let us consider a single layer, for each component $\alpha = 1, \ldots, M$,

$$y^\alpha = \sigma\left(\sum_{\beta=1}^N W_\beta^\alpha x^\beta + b^\alpha \right).$$

The bias b is usually initialized as zero vector, thus we can neglect b when consider initialization. The weight W is usually initialized by uniformly sampling from a range $(-\omega,\omega)$, or by sampling from normal distribution $\mathcal{N}(0,\omega^2)$, for some $\omega>0$. We are to give a proper value of ω. Assume that x has been properly normalized, so that $\mathbb{E}(x^\alpha)=0$ and $\mathrm{Var}[x^\alpha]=1$ for each component α, where the expectations are taken over mini-batch. Otherwise, replace $x^\alpha\to(x^\alpha-\mathbb{E}[x^\alpha])/\sqrt{\mathrm{Var}[x^\alpha]}$.

One extreme is $\omega\gg 1$. In this situation, the randomness in W will pollute the information in the input x, thus y cannot gain any correlation with x. That is, information cannot "flow through" the perceptron in the forward propagation. To make this explicit, we first omit the σ function and bias b, thus consider $z^\alpha:=\sum_{\beta=1}^N W_\beta^\alpha x^\beta$. If W was deterministic, then we would have $\mathrm{Var}[z]=0$. Thus, the variance of z reflects the randomness of W. We have $\mathrm{Var}[z^\alpha]=\mathrm{Var}[\sum_{\beta=1}^N W_\beta^\alpha x^\beta]=\sum_{\beta=1}^N (x^\beta)^2\,\mathrm{Var}[W_\beta^\alpha]=(\omega^2/3)\sum_{\beta=1}^N (x^\beta)^2=(\omega^2/3)\|x\|_2^2$ for each α, where in the second equality, we used the formula $\mathrm{Var}(aX+bY)=a^2\,\mathrm{Var}(X)+b^2\,\mathrm{Var}(Y)+2ab\,\mathrm{Cov}(X,Y)$ and the fact that W_β^α and $W_{\beta'}^\alpha$ $(\beta\neq\beta')$ are independent, and in the third equality, we used the variance of uniform distribution. So, standard derivative of z^α, or the randomness introduced by W to output z, is proportional to ω and L_2-norm of x. The greater ω, the greater pollution.[2.5]

The other extreme is $\omega\ll 1$ (thus $\omega\approx 0$). In this situation, y will be very insensitive to x, thus cannot properly react to a critical change in x (for example, changing from 2 to 3 in the MNIST dataset). This relates to backward propagation of information. So, consider the vector-Jacobian product $\sum_{\alpha=1}^M v_\alpha(\partial y^\alpha/\partial x^\gamma)=\sum_{\alpha=1}^M \sigma'(\sum_{\beta=1}^N W_\beta^\alpha x^\beta+b^\alpha)W_\gamma^\alpha v_\alpha$ for some vector v. For simplicity, now omit the $\sigma'(\ldots)$ term (for ReLU it is unit) and examine the rest, $\delta^\alpha:=\sum_{\alpha=1}^M W_\gamma^\alpha v_\alpha$. Following the same process, we have $\mathrm{Var}[\delta^\alpha]=(\omega^2/3)\|v\|_2^2$ for each α. So, standard derivative of δ^α, or the scale of backward propagation, is proportional to ω and L_2-norm of v. The smaller ω, the smaller backward information progagation.

2.6.2 Normalized Vector-Jacobian Product

We find the optimized model has the property that, in the backward propagation, $\mathrm{Var}[\sum_{\alpha=1}^{N_{l+1}} v_\alpha(\partial z_{l+1}^\alpha/\partial z_l^\beta)(z_l)]=\mathrm{Const}$ for different l, where expectation is taken over component β, *even though* $\mathrm{Var}[z_l]$ *varies greatly in different layers*. This may relate to the singular values of the Jacobian $(\partial z_{l+1}^\alpha/\partial z_l^\beta)(z_l)$.

Let $v\in\mathbb{R}^M$, $J\in\mathbb{R}^{M\times N}$. If suppose $\mathbb{E}[vJ]=0$, then

$$\begin{aligned}\mathrm{Var}[vJ]&=\mathbb{E}[(vJ)^2]\\&=\frac{1}{N}\sum_{\beta=1}^N\left[\sum_{\alpha=1}^M v_\alpha J_\beta^\alpha\right]\left[\sum_{\gamma=1}^M v_\gamma J_\beta^\gamma\right]\\&=\sum_{\alpha,\gamma=1}^M v_\alpha v_\gamma\left(\frac{1}{N}\sum_{\beta=1}^N J_\beta^\alpha J_\beta^\gamma\right).\end{aligned}$$

Define $K^{\alpha\gamma}:=(1/N)\sum_{\beta=1}^N J_\beta^\alpha J_\beta^\gamma$ which is a $M\times M$ real positive-definite symmetric matrix. Thus, it has eigenvalues $((\sigma^1)^2,\ldots,(\sigma^M)^2)$. In a proper coordinate where K is diagonalized, we have $v\to\tilde{v}$. Thus,

$$\mathrm{Var}[vJ]=\langle v|K|v\rangle=\sum_{\alpha=1}^M \tilde{v}_\alpha^2(\sigma^\alpha)^2,$$

compared with $\mathrm{Var}[v]$ which (assume $\mathbb{E}[v]=0$) is $\mathrm{Var}[v]=(1/M)\sum_{\alpha=1}^M v_\alpha^2=(1/M)\sum_{\alpha=1}^M \tilde{v}_\alpha^2$.

For example, when $\tilde{v}_\alpha\equiv\phi$ for some $\phi\in\mathbb{R}$ for all α, we have $\mathrm{Var}[v]=\phi^2$, thus

$$\frac{\mathrm{Var}[vJ]}{\mathrm{Var}[v]}=\sum_{\alpha=1}^M(\sigma^\alpha)^2=\mathrm{tr}(K).$$

2.5. This may not be true. The values of W, once sampled, has been determined. From information perspective, as long as the W is non-degenerate, then the mutual information between z and x will not be lost because of W. But, W is usually non-degenerate, since W obeys the Wigner semicircle theorem which states that the eigenvalues of a random matrix obey a Wigner semicircle distribution which is zero-measure on zero.

The same for Jacobian-vector product. Let $u \in \mathbb{R}^N$. If suppose $\mathbb{E}[Ju] = 0$, then, following the same steps,

$$\mathrm{Var}[Ju] = \sum_{\alpha,\gamma=1}^{N} \left(\frac{1}{M} \sum_{\beta=1}^{M} J_\alpha^\beta J_\gamma^\beta \right) u^\alpha u^\gamma.$$

Define $L_{\alpha\gamma} := (1/M) \sum_{\beta=1}^{M} J_\alpha^\beta J_\gamma^\beta$ which is a $N \times N$ real positive-definite symmetric matrix. Thus, it has eigenvalues $(\psi_1^2, \ldots, \psi_N^2)$. In a proper coordinate where L is diagonalized, we have $u \to \tilde{u}$. Thus,

$$\mathrm{Var}[Ju] = \langle u|L|u\rangle = \sum_{\alpha=1}^{N} (\tilde{u}^\alpha)^2 \psi_\alpha^2,$$

compared with $\mathrm{Var}[u]$ which (assume $\mathbb{E}[u] = 0$) is $\mathrm{Var}[u] = (1/N) \sum_{\alpha=1}^{N} (u^\alpha)^2 = (1/N) \sum_{\alpha=1}^{N} (\tilde{u}^\alpha)^2$.

2.6.3 Criticality

Remind of the criticality in Ising model. An improper parameter (T that away from critical T_c) blocks the propagation of **local perturbation**. Now, we are to find a similar concept in deep neural network.

A deep neural network has the general structure ($l = 1, \ldots, L$)

$$z_l = f_l(z_{l-1}; \theta_l),$$

with $z_0 := x$ as the model input, and $y := z_L$ as the model output. Thus, the model becomes the composition $f(\cdot; \theta) = f_L(\cdot; \theta_L) \circ \cdots \circ f_1(\cdot; \theta_1)$, where $\theta = \theta_1 \oplus \cdots \oplus \theta_L$. The depth l plays the role of time in Ising model.

2.6.4 RNN: Boundary between Order and Chaos

Consider a deep neural network with residual structure. When all the layers are the same function $f_l(\cdot; \theta_l) \equiv f_{l'}(\cdot; \theta_{l'})$ for each $l, l' = 1, \ldots, L$, then we have

$$z_{l+1} = f(z_l; \theta),$$

with $z_0 = x$ and $z_L = y$ again. For a perturbation $x' = x + \delta x$, we have, for each layer, $\delta z_{l+1} = (\partial f / \partial z_l)(z_l; \theta)\, \delta z_l$. If $\|(\partial f / \partial z_l)(z_l; \theta)\, \delta z_l\| < \|\delta z_l\|$, we find $|\delta y| \ll |\delta x|$ as $L \gg 1$. Conversely, if $\|(\partial f / \partial z_l)(z_l; \theta)\, \delta z_l\| > \|\delta z_l\|$, we have $|\delta y| \gg |\delta x|$ as $L \gg 1$. The first is in order, while the second is chaos. Thus, the boundary $\|(\partial f / \partial z_l)(z_l; \theta)\, \delta z_l\| = \|\delta z_l\|$ characterizes the criticality, which is the same as the bifurcation of logistic map.

Chapter 3
When Neural Network Becomes Deep

3.1 Enlarging Model Increases Performance

In this section, we investigate how the model performance benefits from increasing the number of adaptive parameters. We are to show that this almost always results in an increment of performance.

Consider a model $f(x;\theta)$ with m parameters (or components of parameter). In the infinite-dimensional parameter space (section 1.3), it is written as $f(x;(\theta^1,\dots,\theta^m,0,\dots))$. For convienence, we omit the innermost parentheses, so it is denoted by $f(x;\theta^1,\dots,\theta^m,0,\dots)$. Given a dataset $D=\{(x_i,y_i)|i=1,\dots,N\}$, suppose that we have trained the model, resulting in the best-fit model $f(x;\theta^1_\star,\dots,\theta^m_\star,0,\dots)$. *After* the training, we append n adaptive parameters, with $n\ll m$. For convienence, we denote the later n parameters by φ, thus the model changes from m parameters

$$f(x;\theta_\star)=f(x;\theta^1_\star,\dots,\theta^m_\star,0,\dots)$$

to $(m+n)$ parameters

$$f(x;\theta_\star,\varphi)=f(x;\theta^1_\star,\dots,\theta^m_\star,\theta^{m+1},\dots,\theta^{m+n},0,\dots).$$

Practically, we can safely suppose that all the best-fit parameters are small.[3.1] Together with $n\ll m$, it implies that, while continue the training, we can freeze θ to $\theta_\star$ and only optimize φ, since the feedback from φ to θ can be negligible. So, the loss function used for fine-tuning the model becomes

$$L(\varphi)=\mathbb{E}_{(x,y)\sim D}[d(f(x;\theta_\star,\varphi),y)].$$

Minimizing $L(\varphi)$ results in a best-fit $\varphi_\star$. Then we find the best-fit of the model as $(\theta_\star,\varphi_\star)$. We are to show that the fine-tuning always decrease the loss. That is, enlarging model always benefits its performance.

As an example, consider $d(x,y)=(x-y)^2/2$. Thus, the output is a real number. We Taylor expand f by φ at $\varphi=0$, as

$$f(x;\theta_\star,\varphi)=f(x;\theta_\star)+\sum_{\alpha=1}^{n}\varphi^\alpha\frac{\partial f}{\partial\varphi^\alpha}(x;\theta_\star)+o(\varphi).$$

Inserting into the loss function gives

$$L(\varphi)=\frac{1}{2}\mathbb{E}_{(x,y)\sim D}[(f(x;\theta_\star,\varphi)-y)^2]=L(\varphi)=\frac{1}{2}\mathbb{E}_{(x,y)\sim D}\left[\left(\sum_{\alpha=1}^{n}\varphi^\alpha\frac{\partial f}{\partial\varphi^\alpha}(x;\theta_\star)-\delta y+o(\varphi)\right)^2\right],$$

where $\delta y:=y-f(x;\theta_\star)$, the residue left by $f(x;\theta_\star)$. Since φ is small, we can omit the $o(\varphi)$, resulting in a linear model that fits the residue.[3.2] It can be solved analytically. We have

$$\frac{\partial L}{\partial\varphi^\alpha}(\varphi)=\sum_{\beta=1}^{n}\mathbb{E}_{(x,y)\sim D}\left[\frac{\partial f}{\partial\varphi^\beta}(x;\theta_\star)\frac{\partial f}{\partial\varphi^\alpha}(x;\theta_\star)\right]\varphi^\beta-\mathbb{E}_{(x,y)\sim D}\left[\delta y\frac{\partial f}{\partial\varphi^\alpha}(x;\theta_\star)\right].$$

3.1. This is generally true, for several reasons. One is that small parameters increases robustness in generalization. Another may trace to the initialization strategy and optimization algorithm. Temporally, we will not discuss these aspects, but simply assume the fact.

3.2. The same situation as `xgboost`. For detail, see *XGBoost: A Scalable Tree Boosting System* by Tianqi Chen and Carlos Guestrin, 2016. arXiv: 1603.02754

The matrix

$$\mathbb{E}_{(x,y)\sim D}\left[\frac{\partial f}{\partial \varphi^\beta}(x;\theta_\star)\frac{\partial f}{\partial \varphi^\alpha}(x;\theta_\star)\right]$$

is the covariance of random variable $(\partial f/\partial\varphi^\alpha)(X;\theta_\star)$, where we use capital character X for indicating randomness. Thus it is positive definite unless for component α, $(\partial f/\partial\varphi^\alpha)(x;\theta_\star)$ vanishes for all x, meaning that φ^α is a useless parameter (this shall not happen). So, it has inverse and $(\partial L/\partial\varphi^\alpha)(\varphi_\star)=0$ gives

$$\varphi^\beta_\star = \sum_{\alpha=1}^{n} \mathbb{E}_{(x,y)\sim D}\left[\delta y \frac{\partial f}{\partial \varphi^\alpha}(x;\theta_\star)\right]\left\{\mathbb{E}_{(x,y)\sim D}\left[\frac{\partial f}{\partial \varphi}(x;\theta_\star)\frac{\partial f}{\partial \varphi}(x;\theta_\star)\right]^{-1}\right\}^{\alpha\beta}.$$

And $\varphi_\star=0$ only when

$$\mathbb{E}_{(x,y)\sim D}\left[\delta y \frac{\partial f}{\partial \varphi^\alpha}(x;\theta_\star)\right]=0$$

for any α. Notice that

$$\frac{\partial L}{\partial \varphi^\alpha}(0) = -\mathbb{E}_{(x,y)\sim D}\left[\delta y \frac{\partial f}{\partial \varphi^\alpha}(x;\theta_\star)\right].$$

It means $\varphi_\star=0$ has been the best-fit of φ.

In this situation $\varphi_\star=0$, indicating that introducing φ does not increase the performance, even though it does not decrease the performance also.

3.2 (TODO)

Introducing new parameters to a model may increase its representability, but not always so. Consider the feed-forward neural network with single hidden layer (see, section ?),

$$f^\alpha(x;\theta) := U^\alpha_\beta\,\sigma(W^\beta_\gamma x^\gamma + b^\beta) + c^\alpha,$$

where $\theta := (U, W, b, c)$, $x\in\mathbb{R}^n$, $W\in\mathbb{R}^{m\times n}$. and σ denotes the activation function. Now, introduce a new parameter $V\in\mathbb{R}^{m\times n}$ in such a way that

$$f^\alpha(x;\theta) \to f^\alpha_{\text{ext}}(x;\theta_{\text{ext}}) := U^\alpha_\beta\,\sigma((V^\beta_\gamma + W^\beta_\gamma)\,x^\gamma + b^\beta) + c^\alpha,$$

where $\theta_{\text{ext}} := (U, V, W, b, c)$. It is apparently that the new parameter may not help increase the representability. In fact, we can combine $V+W$ as a whole, $\tilde{W} := V+W$. Thus, f_{ext} goes back to $U^\alpha_\beta\,\sigma(\tilde{W}^\beta_\gamma x^\gamma + b^\beta) + c^\alpha$, which is the same as f. So, introducing new parameters in a "wrong" way will not increase the representability of the model.

This example is still too complicated to investigated. Let us simply it further. Consider a model

$$g(x;\theta) := a\,\sigma(bx),$$

where $\theta := (a, b)$ and $x, a, b\in\mathbb{R}$. You can recognize that this model g is the extremely simplified version of the model f. The same, introduce a "wrong" parameter $c\in\mathbb{R}$ such that

$$g_{\text{ext}}(x;\theta_{\text{ext}}) := a\,\sigma((b+c)\,x),$$

where $\theta_{\text{ext}} := (a, b, c)$ is the extended collection of parameters; and defining $\tilde{b} := b+c$ makes g_{ext} go back to g. The key is that, after defining $\tilde{b}$, g_{ext} has the same *form* as g. This definition can be seem as a coordinates transform on the parameter space $(a, b, c)\to(a, \tilde{b}, c)$ after which some coordinate (the c) is absent. Notice that this coordinates transform is only for θ_{ext}, thus is independent of x. Contrarily, if we introduce the parameter c in such a way that

$$g'_{\text{ext}}(x;\theta_{\text{ext}}) := a\,\sigma(bx + c),$$

then (recall that σ is a non-linear function) we cannot find an x-independent coordinate transform of θ_{ext} by which some coordinate becomes absent.

3.3 Draft

As discussed in section 2.1, given a dataset D, the performance of data-fitting is characterized by a loss function $L_D(\theta)$ where θ is the collection of model parameters. These parameters are like the knobs on an instrument. By tuning these knobs, the value displayed on the instrument screen varies. But, unlike the knobs, not all the parameters are independent. As an example, consider a loss function $L_D(\theta_1, \theta_2) = g(\theta_1 + \theta_2)$ on 2-dimensional parameter space, where $g: \mathbb{R} \to \mathbb{R}$. It seems that L_D has two knobs, but letting the coordinates transform as $\theta \to \tilde{\theta}$ where $\tilde{\theta}_1 := (\theta_1 + \theta_2)/2$ and $\tilde{\theta}_2 := (\theta_1 - \theta_2)/2$, which is apparently bijective, we have $\tilde{L}_D(\tilde{\theta}_1, \tilde{\theta}_2) := L_D(\theta_1(\tilde{\theta}_1, \tilde{\theta}_2), \theta_2(\tilde{\theta}_1, \tilde{\theta}_2)) = g(2\tilde{\theta}_1)$. Tuning $\tilde{\theta}_2$ does not change the loss function; it is a useless knob. Precisely, there may exist a global coordinate transformation $\theta \to \tilde{\theta}$ on the parameter space such that some component $\tilde{\theta}_\alpha$ is absent in the transformed loss function $\tilde{L}_D(\tilde{\theta}) := L_D(\theta(\tilde{\theta}))$. That is,

$$\sum_\beta \frac{\partial L_D}{\partial \theta_\beta}(\theta) \frac{\partial \theta_\beta}{\partial \tilde{\theta}_\alpha}(\tilde{\theta}(\theta)) = 0$$

holds for all θ. This indicates that not all parameters are equally effective for decreasing the loss function. Parameters like this are called **degenerate**.

3.4 Enlarging Model Is Efficient for Increasing Its Representability (TODO)

As long as the model is not degenerate, all its "knobs" are effective for tuning. We may expect that the more parameters it has, the better performance it will be. So, for increasing the representability of a model (neural network), so as to fit more and more data with sufficient flexibility, we can enlarge it by increasing the number of the trainable parameters. Even though there are many other ways of increasing representability, such as changing the model architecture, simply increasing the number of trainable parameters will be the most cheap, safe, and efficient.

3.5 Increasing Depth Is More Efficient for Enlarging Model

3.5.1 Simple Baseline Model

There are mainly two ways to increasing the number of trainable parameters: increasing width or increasing depth. By increasing width, we enlarge the dimension of hidden layers. And by increasing depth, we add more hidden layers to the model. The problem is, which way of enlarging model is more computational efficient, having less complexity.

We are to exam this problem by considering a sufficiently simple neural network with one-dimensional input, one-dimension output, one hidden layer, and without biases. Suppose we have a baseline neural network $y = f(\sum_{\alpha=1}^n U_\alpha z^\alpha)$ with $z^\alpha = f(W^\alpha x)$, where $n \gg 1$. There are n Us and Ws respectively, thus $2n$ parameters. What is the computational complexity? We have to go through the processes that computes y from x and derivatives $\partial y / \partial U$, $\partial y / \partial W$.

For computing z, we have to do n multiplications for $W^\alpha x$, and n activations by f. For computing y from z, we make n multiplications for $U_\alpha z^\alpha$, and one activation by f. For computing $\partial y / \partial U$, we have

$$\frac{\partial y}{\partial U^\alpha} = f'\left(\sum_{\alpha=1}^n U_\alpha z^\alpha\right) z^\alpha.$$

Notice that the term $\sum_{\alpha=1}^{n} U_\alpha z^\alpha$ and z has been computed previously. There is no need to compute a quantity more than once; we shall cache them when computing y from x. This occupies memory for $1+n$ float numbers. Now, to compute $\partial y / \partial U$, we simply need one activation by f' and n multiplications. Finally, for computing $\partial y / \partial W$, we have

$$\frac{\partial y}{\partial W_\alpha} = \frac{\partial y}{\partial z^\alpha}\frac{\partial z^\alpha}{\partial W_\alpha} = f'\left(\sum_{\alpha=1}^{n} U_\alpha z^\alpha\right) U_\alpha \times f'(W^\alpha x)\, x.$$

Again, $W^\alpha x$ has been computed previously, thus shall be cached, occupying memory for n float numbers. In addition, we need n activations by f' for $f'(W^\alpha x)$, one multiplication for $f'(\sum_{\alpha=1}^{n} U_\alpha z^\alpha)$ times x, n multiplications for $U_\alpha \times f'(W^\alpha x)$, and n multiplications for the final result. Totally, we have to cache about $2n$ float numbers, which is the spatial complexity; and about n activations by f, n activations by f', and $5n$ multiplications, which is the temporal complexity.

3.5.2 Increasing Depth

If we add a new hidden layer (without bias) between y and z, say z', with dimension n, then $y = f(\sum_{\alpha=1}^{n} U_\alpha z'^\alpha)$, $z'^\alpha = f(\sum_{\beta=1}^{n} V_\beta^\alpha z^\beta)$, and $z^\alpha = f(W^\alpha x)$. there will be n^2 additional parameters (the weight V), thus $2n+n^2$ parameters in total.

What is the complexity? Again, we have to go through the processes that computes y from x and derivatives $\partial y / \partial U$, $\partial y / \partial V$, $\partial y / \partial W$.

For compute y from x, we have known the computation from x to z and from z' to y, which need about n activations by f and $2n$ multiplications in total. All left to do is figuring out the process that computes z' from z. This needs n activations by f and n^2 multiplications. Computing $\partial y / \partial U$ is the same as before, which caches about n float numbers and needs n multiplications. For $\partial y / \partial V$, we have

$$\frac{\partial y}{\partial V_\alpha^\beta} = \frac{\partial y}{\partial z'^\beta}\frac{\partial z'^\beta}{\partial V_\alpha^\beta} = f'\left(\sum_{\alpha=1}^{n} U_\alpha z'^\alpha\right) U_\beta \times f'\left(\sum_{\alpha=1}^{n} V_\alpha^\beta z^\alpha\right) z^\alpha.$$

Again, the terms $\sum_{\alpha=1}^{n} U_\alpha z'^\alpha$, $\sum_{\alpha=1}^{n} V_\alpha^\beta z^\alpha$, and z^α has been computed, we shall cache them for reusing. This occupies a memory of $1+2n$ float numbers. Thus, we just need to compute n activations by f' for $f'(\sum_{\alpha=1}^{n} V_\alpha^\beta z^\alpha)$, n multiplications for $f'(\sum_{\alpha=1}^{n} U_\alpha z'^\alpha)\, z^\alpha$, n multiplications for $U_\beta \times f'(\sum_{\alpha=1}^{n} V_\alpha^\beta z^\alpha)$, and n^2 multiplications for the final result. For $\partial y / \partial W$, we have

$$\frac{\partial y}{\partial W_\alpha} = \sum_{\beta=1}^{n} \frac{\partial y}{\partial z'^\beta}\frac{\partial z'^\beta}{\partial z^\alpha}\frac{\partial z^\alpha}{\partial W_\alpha} = \sum_{\beta=1}^{n} f'\left(\sum_{\alpha=1}^{n} U_\alpha z'^\alpha\right) U_\beta \times f'\left(\sum_{\alpha=1}^{n} V_\alpha^\beta z^\alpha\right) V_\alpha^\beta \times f'(W^\alpha x)\, x.$$

We shall read $W^\alpha x$ from cache, which occupies a memory of n float numbers. In addition, we need to compute n activations by f', one multiplication for $f'(\sum_{\alpha=1}^{n} U_\alpha z'^\alpha)\, x$, n multiplications for $U_\beta \times f'(\sum_{\alpha=1}^{n} V_\alpha^\beta z^\alpha)$, and $2n^2$ multiplications for the final result. Totally, we have to cache about $4n$ float numbers, which is the spatial complexity; and about $2n$ activations by f, $2n$ activations by f', and $4n^2$ multiplications, which is the temporal complexity.

3.5.3 Increasing Width

If we are to increase the width of the baseline neural network so as to obtain the same number of parameters of that which adds a new hidden layer, we shall extend the z to m-dimension, such that $2m = 2n+n^2$. Thus, based on the computation in section 3.5.1, we have to cache about $2m \approx n^2$ float numbers, which is the spatial complexity; and about $m \approx 0.5n^2$ activations by f, $m \approx 0.5n^2$ activations by f', and $5m \approx 2.5n^2$ multiplications, which is the temporal complexity.

3.5.4 Summary: Increasing Depth v.s. Increasing Width

Now, we find that, for obtaining the same number of trainable parameters after enlarging, increasing depth is much more efficient in memory than increasing width. And increasing width is almost as the same efficiency as increasing depth in computing time. Recall that this result is obtained when $n \gg 1$. In this situation, increasing depth is much more efficient in computation than increasing width.

3.6 Increasing Depth Makes It Hard to Control the Gradients

Even though increasing depth is more efficient for enlarging the model capacity, it increases the difficulty of training. To declare this problem, consider a feed-forward neural network with L layers. It can be expressed as

$$z_l^\alpha = f_l\left(\sum_{\beta=1}^{n_l} (W_l)_\beta^\alpha\, z_{l-1}^\beta + b_l^\alpha\right), \tag{3.1}$$

where $l=1,\dots,L$ represents the number of layer, z_0 is model input, and z_L is model output. The W_l and b_l are the trainable parameters of the perceptron at layer l, and f_l its activation function. For training by gradient descent methods, we have to compute the derivative, such as:

$$\frac{\partial z_L}{\partial W_1}(z_0) = \frac{\partial z_L}{\partial z_{L-1}}(z_{L-1})\cdot\frac{\partial z_{L-1}}{\partial z_{L-2}}(z_{L-2})\cdot\cdots\cdot\frac{\partial z_2}{\partial z_1}(z_1)\cdot\frac{\partial z_1}{\partial W_1}(z_0), \tag{3.2}$$

where for simplicity we employed matrix multiplication format, in which each partial derivative is an Jacobian matrix.

For stabilizing the training, we hope that $\|\partial z_L/\partial W_1\|\sim 1$ during the whole training process before approaching the best fit.[3.3] But, when the model becomes depth, it cannot be guaranteed. Indeed, we have the rough estimation

$$\left\|\frac{\partial z_L}{\partial W_1}(z_0)\right\| \sim \left\|\frac{\partial z_L}{\partial z_{L-1}}(z_{L-1})\right\|\left\|\frac{\partial z_{L-1}}{\partial z_{L-2}}(z_{L-2})\right\|\cdots\left\|\frac{\partial z_2}{\partial z_1}(z_1)\right\|\left\|\frac{\partial z_1}{\partial W_1}(z_0)\right\|,$$

from which we can see that, when $L\gg 1$, we have to carefully tune the $\partial z_l/\partial z_{l-1}$ for all $l=2,\dots,L$ to balance the long sequence of products on the right hand side, so as to make $\|\partial z_L/\partial W_1\|\sim 1$. This, however, cannot be done since the training process is quite complicated and unpredictable, let alone fine-tuning the $\partial z_l/\partial z_{l-1}$s. As a result, when the model becomes quite deep, the training process will be extremely unstable: the gradients of parameters jump back and forth over a wide range.

3.7 Techniques Are Combined for Controlling the Gradients

3.7.1 Residual Structure

As the chain-rule (3.2) indicated, the problem of bounding the gradients by parameters, such as $\partial z_L/\partial W_1$, can be converted to bound the $\|\partial z_{l+m}/\partial z_l\|$ for each $1\leqslant l<l+m\leqslant L$, hoping for $\|\partial z_{l+m}/\partial z_l\|\sim 1$.

The trick is doing perturbation. Explicitly, instead of using perceptron to represent the function that computes z_l out of z_{l-1} as before, we use it for the difference between z_l and z_{l-1}. That is, we re-define the $z_l(z_{l-1})$ from

$$z_l^\alpha = f_l\left(\sum_{\beta=1}^{n_l} (W_l)_\beta^\alpha\, z_{l-1}^\beta + b_l^\alpha\right)$$

to

$$z_l^\alpha = z_{l-1}^\alpha + f_l\left(\sum_{\beta=1}^{n_l} (W_l)_\beta^\alpha\, z_{l-1}^\beta + b_l^\alpha\right)$$

for all $l=1,\dots,L$. The term

$$r_{l-1}^\alpha(z_{l-1}) := f_l\left(\sum_{\gamma=1}^{n_l} (W_l)_\gamma^\alpha\, z_{l-1}^\gamma + b_l^\alpha\right)$$

is called *residual*. We then have

$$\frac{\partial z_l^\alpha}{\partial z_{l-1}^\beta} = \delta_\beta^\alpha + \frac{\partial r_{l-1}^\alpha}{\partial z_{l-1}^\beta}(z_{l-1}).$$

3.3. By saying $|x|\sim 1$ for some x, we mean that its absolute value will not be extremely large or tiny, such as 10^{10} or 10^{-10}.

As long as the $\|\partial r / \partial z\|$ is small enough (perturbative), we will have $\|\partial z_{l+1}/\partial z_l\| \sim 1$.

Now, we apply this to calculate $\partial z_{l+m}/\partial z_l$. First, we try the simplest case where $m=2$:

$$\begin{aligned}\frac{\partial z_{l+2}^{\alpha}}{\partial z_l^{\beta}}(z_l) &= \sum_{\gamma=1}^{n_{l+1}} \frac{\partial z_{l+2}^{\alpha}}{\partial z_{l+1}^{\gamma}}(z_{l+1}) \frac{\partial z_{l+1}^{\gamma}}{\partial z_l^{\beta}}(z_l) \\ &= \sum_{\gamma=1}^{n_{l+1}} \left(\delta_{\gamma}^{\alpha} + \frac{\partial r_{l+1}^{\alpha}}{\partial z_{l+1}^{\gamma}}(z_{l+1})\right)\left(\delta_{\beta}^{\gamma} + \frac{\partial r_l^{\gamma}}{\partial z_l^{\beta}}(z_l)\right) \\ &= \delta_{\beta}^{\alpha} + \frac{\partial r_l^{\alpha}}{\partial z_l^{\beta}}(z_l) + \frac{\partial r_{l+1}^{\alpha}}{\partial z_{l+1}^{\beta}}(z_{l+1}) + o\left(\left\|\frac{\partial r}{\partial z}\right\|\right).\end{aligned}$$

By repeating this process, we will find

$$\frac{\partial z_{l+m}^{\alpha}}{\partial z_l^{\beta}}(z_l) = \delta_{\beta}^{\alpha} + \frac{\partial r_l^{\alpha}}{\partial z_l^{\beta}}(z_l) + \cdots + \frac{\partial r_{l+m-1}^{\alpha}}{\partial z_{l+m-1}^{\beta}}(z_{l+m-1}) + o\left(\left\|\frac{\partial r}{\partial z}\right\|\right).$$

Now, if we can further bound the $\|\partial r / \partial z\|$, then problem is solved. Recall that $r_{l-1}^{\alpha}(z_{l-1}) := f_l(\sum_{\gamma=1}^{n_l} (W_l)_{\gamma}^{\alpha} z_{l-1}^{\gamma} + b_l^{\alpha})$, thus

$$\frac{\partial r_{l-1}^{\alpha}}{\partial z_{l-1}^{\beta}}(z_{l-1}) = (W_l)_{\beta}^{\alpha} \times f_l'\left(\sum_{\gamma=1}^{n_l} (W_l)_{\gamma}^{\alpha} z_{l-1}^{\gamma} + b_l^{\alpha}\right).$$

The problem now is converted to bound $(W_l)_{\beta}^{\alpha}$ and $f_l'(\sum_{\gamma=1}^{n_l} (W_l)_{\gamma}^{\alpha} z_{l-1}^{\gamma} + b_l^{\alpha})$.

3.7.2 Regularization

Regularization techniques, including L_2-regularization and `AdamW` optimizer (that is, `Adam` optimizer with weight decay), help bound the $\|W_l\|$ and $\|b_l\|$. Explicitly, we add a penalty term to loss (which is to be minimized by training) as

$$\lambda \sum_{l=1}^{L} (\|W_l\|_2 + \|b_l\|_2),$$

where λ is a hyper-parameter that weights the penalty, and $\|\cdot\|_2$ represents the L_2-norm. As the result of regularization, $\|W_l\|$ and $\|b_l\|$ are bounded to be small.

3.7.3 Normalization

Normalization techniques, such as batch normalization and layer normalization, serve for bounding the second term. Explicitly, it passes $N(z_{l-1})$ instead of z_{l-1} to the perceptron at layer l, where $N(z)$ is defined as

$$N(z) := \frac{z - \mathbb{E}[z]}{\sqrt{\mathrm{Var}[z]}}.$$

For batch normalization, The expectation $\mathbb{E}[z]$ and variance $\mathrm{Var}[z]$ are counted by data-batch (we put multiple data into the model in parallel, called *batch*, thus z is a batch). For layer normalization, they are counted by neurons (that is, by the index of z^{α}). Anyway, normalization techniques shift and rescale z_{l-1} so that $\|z_{l-1}\| \sim 1$.

3.7.4 Summary: Gradients Are Bounded by the Techniques Altogether

Since $\|W_l\|$ and $\|b_l\|$ have been bounded to be small and $\|z_{l-1}\|$ is bounded to unit, the term $|\sum_{\gamma=1}^{n_l} (W_l)_{\gamma}^{\alpha} z_{l-1}^{\gamma} + b_l^{\alpha}|$ is properly bounded for each α. Assuming that f' is not singular, which is held for all commonly used activation functions, then $|f_l'(\sum_{\gamma=1}^{n_l} (W_l)_{\gamma}^{\alpha} z_{l-1}^{\gamma} + b_l^{\alpha})| \lesssim 1$ for each α. Thus, for each α and β,

$$\left|\frac{\partial r_{l-1}^{\alpha}}{\partial z_{l-1}^{\beta}}(z_{l-1})\right| = |(W_l)_{\beta}^{\alpha}| \times \left|f_l'\left(\sum_{\gamma=1}^{n_l} (W_l)_{\gamma}^{\alpha} z_{l-1}^{\gamma} + b_l^{\alpha}\right)\right|$$

is bounded to be small. Recall that $\|\partial r_{l-1}/\partial z_{l-1}\| \ll 1$ implies $\|\partial z_{l+m}/\partial z_l\| \sim 1$. The gradients $\|\partial z_L/\partial W_1\|$, and all other gradients by parameters, are thus bounded.[3.4]

3.8 A Little History about Depth

3.8.1 Regularization

Regularization has a long history (comparing with the history of modern neural network). L_2-regularization was first suggested by a Soviet Russian mathematician Andrey Tikhonov in 1943, which is also called "ridge regression". Besides L_2-regularization, commonly used regularization techniques are L_1-regularization, proposed by Fadil Santosa and William Symes in 1986.

The general L_n regularization is adding a penalty term $\lambda \|\theta\|_n$ to loss function, where θ represents the model parameters and $\|\cdot\|_n$ the n-norm. The hyper-parameter λ is in $(0, +\infty)$. To obtain a proper value of hyper-parameter, researchers have to search in an infinite range, which is difficult. So, in 2014, another kind of regularization method called dropout was proposed by *Nitish Srivastava* and others. Dropout also has a hyper-parameter, the dropout probability p. But p is in $(0, 1)$, a finite range. Searching for a proper hyper-parameter becomes much easier.

Regularization via optimizer was first proposed by Ilya Loshchilov and Frank Hutter in 2017. They added weight decay to the `Adam` optimizer for regularizing the model parameters during training.

Regularization techniques were invented, not for dealing with depth, but for avoiding overfitting.

3.8.2 Recurrent Neural Network

Interestingly, the problem that it is hard to control the gradients of deep feed-forward neural network was first encountered in 1991, not in a very deep architecture, but a shallow one. It was the recurrent neural network, which has an intrinsic property that is equivalent to depth.

Explicitly, recurrent neural network was invented for manipulating sequential data such as text (a sequence of words). A sequential datum is described by $(x_1, \dots, x_T)$ with each $x_t \in \mathbb{R}^n$. For manipulating it, recurrent neural network was designed as

$$z_t^\alpha = f(W_\beta^\alpha z_{t-1}^\beta + U_\beta^\alpha x_t^\alpha + b),$$

where z_t with $t = 1, \dots, T$ and $z_0 = 0$ is recognized as the (sequential) output of a hidden layer, parameterized by W, U, and b. So, z_0 and x_1 are used to compute z_1; then z_1 and x_2 are used to compute z_2; $\dots$; finally z_{T-1} and x_T are used to compute z_T. Usually, z_T is then passed to an output layer, usually a perceptron, to make a prediction, such as the emotion that the input text contains. So, this model consists of a hidden layer and an output layer, thus is very shallow.

But, it is intrinsically very deep, since z_t depends on z_{t-1}, thus on z_{t-2}, thus on z_{t-3}, etc. It is like the feed-forward neural network. When $T \gg 1$, it becomes very deep.

3.8.3 Long Short-Term Memory

To solve the problem of recurrent neural network caused by it "depth", in 1997, a complex structure called *long short-term memory* (LSTM for short) was proposed by Sepp Hochreiter and Jürgen Schmidhuber. Even though not solved, but the severity of uncontrollable gradients can be effectively reduced.

The basic idea underlying long short-term memory is dynamically omitting some intermediate z_t for computing z_T. By saying "dynamically", we mean that the model determines which z_t is to be omitted based on the current input. In this way, z_T does not depend on some z_t. In other words, the "depth" is reduced.

3.4. We examined this theoretical analysis by numerical experiments on the fashion-MNIST dataset (using TensorFlow). The result surprisingly supports our analysis. For details, see the Jupyter notebook `depth-fashion-mnist.ipynb`.

3.8.4 Highway

Inspired by long short-term memory, in the May of 2015, Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber suggested a similar method for feed-forward neural network that dynamically omits some hidden layers.[3.5] The effective depth is thus reduced. Again, the model is trained to determine which hidden layers are to be omitted based on the current input.

3.8.5 Batch Normalization

Another technique that contributes to controlling gradients is normalization. But, it was initially motivated by another problem called *internal covariate shift*. During the training process, the trainable parameters and model input are dynamically changing. The mean value of hidden layer output may shift away from zero. This change, however, is hard to control, especially when the model becomes deep. In this situation, the shift of the hidden layer output will accumulate layer by layer, thus become larger and large, making the model unstable.

To solve this problem, in the February of 2015, Sergey Ioffe and Christian Szegedy proposed a simple method that regularizes the hidden layer output, called *batch normalization*. With a batch of input x, we get a batch of z_l, the hidden layer output at layer l. For each component α, we normalize z_l^α to be

$$\hat{z}_l^\alpha := \frac{z_l^\alpha - \mathbb{E}_{z_l\sim\mathcal{B}}[z_l^\alpha]}{\sqrt{\mathrm{Var}_{z_l\sim\mathcal{B}}[z_l^\alpha]}} \tag{3.3}$$

where the expectation and variance are taken on batch (denoted by $\mathcal{B}$). Then use the $\hat{z}_l^\alpha$ as the input of layer $l+1$.[3.6]

3.8.6 Layer Normalization

Later on, in the July of 2016, Jimmy Lei Ba and others proposed another normalization technique called *layer normalization*. In this method, the expectation and variance are computed on the neurons on the same layer. Explicitly, they modified the expectation $\mathbb{E}_{z_l\sim\mathcal{B}}[z_l^\alpha]$ to be $\mathbb{E}_\alpha[z_l]$, which is the expectation taken on the index α. The same for variance. In this way, the normalization can be established even when data are not grouped into batches.[3.7]

3.8.7 Residual Neural Network

Parallel to the normalization technique, the highway method was continually improved. In the December of 2015, Kaiming He and others simplified the original highway structure. They dropped the gate, which was used for determining which hidden layers are to be omitted. Before passing the output of hidden layer l to the successive hidden layer as input, the input of hidden layer l is added to its output, and the addition as a whole is passed to the successive hidden layer. Explicitly, using the language of feed-forward neural network, it is

$$z_l^\alpha = z_{l-1}^\alpha + f_l\left(\sum_{\beta=1}^{n_l} (W_l)_\beta^\alpha\, z_{l-1}^\beta + b_l^\alpha\right).$$

In addition, they employed batch normalization for the input of each layer. Thus, the whole story becomes

$$z_l^\alpha = z_{l-1}^\alpha + f_l\left(\sum_{\beta=1}^{n_l} (W_l)_\beta^\alpha\, \hat{z}_{l-1}^\beta + b_l^\alpha\right),$$

where $\hat{z}_{l-1}$ is given by equation (3.3). With this *residual structure*, a model can have hundreds of hidden layers. While building the model, they also employed batch normalization.

3.5. To be precisely, the hidden layer is not completely omitted, but gated. For details, see *Highway Networks*.

3.6. In fact, this is not the whole story. The authors introduced two additional trainable parameters γ and β, initialized to 1 and 0 respectively, such that

$$\hat{z}_l^\alpha := \gamma\,\frac{z_l^\alpha - \mathbb{E}[z_l^\alpha]}{\sqrt{\mathrm{Var}[z_l^\alpha]}} + \beta.$$

This, however, does not affect the basic idea of normalization. For details, see the original paper.

3.7. The normalization layer was initially designed for recurrent neural network, where batch normalization cannot be properly used. For details, see the original paper.

Chapter 4
Natural Language Processing

4.1 Representation of Words

4.1.1 Knowing a Word by the Company It Keeps

The theme of natural language processing is teaching machine to understand human languages. Human languages are consist of sentences, which in turn are consist of words. So, the basic task is teaching machine to understand the meaning of a word.

First of all, we have to figure out how to explicitly represent the meaning of a word. In *Philosophical Investigations*, published in 1953, the Austrian philosopher Ludwig Wittgenstein claimed: "One cannot guess how a word functions. One has to look at its use, and learn from that." Later in 1957, the English linguist John Rupert Firth developed this idea, suggested in his little book *Studies in Linguistic Analysis* (section 4): "You shall know a word by the company it keeps". He then illustrated that the meaning of "ass" is indicated by the collocations like "you silly ass", "what an ass he is", and "don't be such an ass". Words in collocation support each other, apprehend each other.

4.1.2 Context-Dependent/Independent Vector Representation

Machine cannot understand a string (a word) unless it is properly encoded. The code shall preserve the meaning of the word. In this section, we seek for the general approach to encode a word by n-dimensional vector based on the Firth's idea: you shall know a word by the company it keeps.

As a preparation, we shall collect a corpus which consists of contexts. A context is a list of words in human languages. It can be a sentence, a paragraph, or a document. Generally, it is a sequence of words $(w_1, w_2, \ldots, w_T)$, where the T varies with each context. In addition, we shall prepare a vocabulary $\mathcal{V}$, which is an (ordered) list of words.[4.1]

For example, consider the sentence (context): the quick brown fox jumps over the lazy dog. The word fox keeps a company consisting of the rest of the words: the, quick, brown, jumps, over, the, lazy, and dog. Notice that the word the appears twice in different positions. A vocabulary can be $\mathcal{V} := (\text{the}, \text{quick}, \text{brown}, \text{fox}, \text{jumps}, \text{over}, \text{lazy}, \text{dog}, \ldots)$, but the is unique in $\mathcal{V}$.

Now, about Firth's idea: given a context, the company that a word keeps are the other words in the same context; and out of its company, the word can be predicted (machine knows the meaning of a word if the word can be correctly predicted). So, the Firth's idea that you shall know a word by the company it keeps can be restated as predicting the word in a context by the other words in the same context.

Generally, given a context $(w_1, w_2, \ldots, w_T)$, a word w_t keeps company $(w_1, \ldots, w_{t-1}, w_{t+1}, \ldots, w_T)$. We are to build a model for the conditional probability

$$p(w|w_1, \ldots, w_{t-1}, \texttt{<?>}, w_{t+1}, \ldots, w_T), \tag{4.1}$$

4.1. In practice, we build the vocabulary $\mathcal{V}$ by word-counting. That is, we iterate over the corpus and count the frequency of each word. Then sort the words by their frequency in descending. For limiting the size of vocabulary, we simply omit the words with extremely low frequencies. Whenever these words are encountered, they are replaced by the "word" <OOV> in vocabulary, short for out-of-vocabulary.

for each word w in vocabulary $\mathcal{V}$, such that, at least statistically, w_t has the maximal probability. The word <?>, which is a specific element in vocabulary $\mathcal{V}$, is a placeholder used for indicating the position of word w in context sequence. For example, for predicting fox out of its company, we shall compute $p(w|\text{the}, \text{quick}, \text{brown}, \text{<?>}, \text{jumps}, \text{over}, \text{the}, \text{lazy}, \text{dog})$ for each word w in (the, quick, brown, fox, jumps, over, lazy, dog, ...), and expect the probability of fox to be maximum.

Since the ultimate goal is to find a vector representation for each word in vocabulary, the conditional probability (4.1) shall be computed out of the vector representation of the word w_t, the meaning of w_t that a machine can understand. The vector representation of w_t, which is context-dependent, can be generally expressed by

$$v_{w_t} := f(w_1, \ldots, w_{t-1}, \text{<?>}, w_{t+1}, \ldots, w_T; \theta), \tag{4.2}$$

where $f(\ldots;\theta): \mathcal{V}^T \to \mathbb{R}^n$ represents a general model parameterized by θ, such as a neural network. Regardless of the explicit form of f, the point is that the vector representation v_{w_t} shall depend on the company that w_t keeps in the context.

In machine learning, categorical probability is represented by the output of softmax function. Recall that softmax function is defined by $\text{softmax}_\alpha(x) := \exp(x_\alpha)/\sum_\beta \exp(x_\beta)$. Since softmax is positive definite and $\sum_\alpha \text{softmax}_\alpha(x) = 1$, its output is usually interpreted as categorical probability, in which the α-index represents the probability of the word in the α position of vocabulary, $\mathcal{V}_\alpha$. In the previous example, the component softmax_2 represents the probability of $\mathcal{V}_2$, which is the word quick. For simplicity, we also use the word for its position in vocabulary, thus $\mathcal{V}_{\text{quick}} := \mathcal{V}_2 = \text{quick}$. The input of softmax function thus shall be a $|\mathcal{V}|$-dimensional vector; and we shall transform the n-dimensional vector v_{w_t} to be $|\mathcal{V}|$-dimensional. The simplest way to do so is linear transformation Uv_{w_t}, where U is a trainable $|\mathcal{V}| \times n$ matrix. Thus,

$$p(w|w_1, \ldots, w_{t-1}, \text{<?>}, w_{t+1}, \ldots, w_T) = \text{softmax}_w(Uv_{w_t}).$$

For w_t can be correctly predicted out of its context, we have to maximize this probability. Thus, the loss function shall be[4.2]

$$L(\theta, U) := -\mathbb{E}_{(w_1,\ldots,w_T)\sim\text{corpus}, t\sim(1,\ldots,T)}[\ln p(w_t|w_1, \ldots, w_{t-1}, \text{<?>}, w_{t+1}, \ldots, w_T)], \tag{4.3}$$

where we have applied the whole process to all the contexts in the corpus and all the words in each context. The whole model has two parts of trainable parameters, the θ used for computing the context-dependent vector representation v_{w_t}, and the matrix U for computing probability. This is how to know a word by the company it keeps for a machine.

The matrix U has an intrinsic interpretation. If we denote each row-vector of the matrix U by u_w, where the w indicates the position of the word w in vocabulary. Thus, in the previous example, we have $(u_{\text{quick}})_\alpha := U_{2\alpha}$ for $\alpha = 1, \ldots, n$. This results in

$$p(w|w_1, \ldots, w_{t-1}, \text{<?>}, w_{t+1}, \ldots, w_T) = \frac{\exp(u_w \cdot v_{w_t})}{\sum_{x\in\mathcal{V}} \exp(u_x \cdot v_{w_t})}, \tag{4.4}$$

where, for each $w \in \mathcal{V}$, $u_w \in \mathbb{R}^n$ is a trainable parameter. The u_w, for each word w in vocabulary, can be viewed as a context-independent (or "absolute") vector representation of w. Indeed, a word w has different context-dependent vectors v_ws in contexts, but the u_w is the context-independent vector that is closest, at least statistically, to these v_ws.

As a summary, we have built a general model for teaching machine to understand the meaning of words, by representing each word by two n-dimensional real vectors. One is context-dependent (the v_w vector); and the other is context-independent or absolute (the u_w vector). All that left is determining the explicit form of $f(\ldots;\theta)$ in equation (4.2). Different researchers propose different forms of $f(\ldots;\theta)$, with different complexities, resulting in different performances.

4.1.3 Example: Bidirectional Encoder Representations from Transformers (BERT)

4.2. In practice, when computing the term $\sum_{x\in\mathcal{V}} \exp(u_x \cdot v_{w_t})$, instead of running over the vocabulary $\mathcal{V}$, it is preferred to sample limited number of words from $\mathcal{V}$ and compute the summation on the samples.

In 2018, Jacob Devlin and others built a model for the vector representation, equation (4.2).[4.3] Given a context $(w_1, w_2, \ldots, w_T)$, a word w_t keeps company $(w_1, \ldots, w_{t-1}, w_{t+1}, \ldots, w_T)$, to model the parameterized function $f(w_1, \ldots, w_{t-1}, \texttt{<?>}, w_{t+1}, \ldots, w_T; \theta)$, they considered a neural network which is a stack of multiple self-attention layers and feed-forward layers.

The first layer in the neural network is called **embedding**. In this layer, each word w_t and its position t are assigned with two n-dimensional real vectors respectively. Then add the two vectors together, resulting in one n-dimensional real vector for the pair (w_t, t), denoted by x_t. So, the input $(w_1, \ldots, w_{t-1}, \texttt{<?>}, w_{t+1}, \ldots, w_T)$ is "embedded" as $(x_1, \ldots, x_T) \in \mathbb{R}^{n \times T}$.

Upon the embedding layer, it is stacked by self-attention layers and feed-forward layers in turn. **Self-attention** layer mixes the information of the "embedded words" mutually. It accepts the sequence $(x_1, \ldots, x_T)$ as input. The portion of information propagating from position t' to t for each $t, t' = 1, \ldots, T$ is determined by a real valued function $q(x_t, x_{t'}; \varphi)$, parameterized by φ. The information of x_t is mixed by others via

$$\bar{x}_t := \frac{\sum_{t'=1}^{T} \exp(q(x_t, x_{t'}; \varphi))\, x_{t'}}{\sum_{t''=1}^{T} \exp(q(x_t, x_{t''}; \varphi))}.$$

If we regard $q(x_t, x_{t'}; \varphi)$ as the logits of a categorical probability $Q(t')$, that is $Q(t') := \mathrm{softmax}_{t'}(q(x_t, x_\cdot; \varphi))$, then $\bar{x}_t$ can be realized as the expectation

$$\bar{x}_t = \mathbb{E}_{t' \sim Q}[x_{t'}],$$

and $Q(t')$ becomes the portion of information propagating from t' to t. This is why we use the bar-notation, which is usually assigned for mean or expectation. The output of self-attention layer is thus the "mixed embedded words" $(\bar{x}_1, \ldots, \bar{x}_T) \in \mathbb{R}^{n \times T}$. Next, the **feed-forward** layer update each $\bar{x}_t$ individually. The feed-forward layer, as it is named, is a shallow feed-forward neural network $g(x; \psi)$, parameterized by ψ. The output is thus the sequence $(g(\bar{x}_1; \varphi), \ldots, g(\bar{x}_T; \varphi))$. This sequence is in turn sent to the next self-attention layer, then next feed-forward layer. Repeating this process multiple times, the final output will be the output of $f(\ldots; \theta)$, the context-dependent vector representation of words. The θ comes to be the collection of parameters in all layers, including embedding, self-attention layers, and feed-forward layers. [4.4]

4.1.4 Application: Named-Entity Recognition

By the vector representation of a word, computer can understand the meaning of the word. Then, it can recognize named-entities in a sentence. Named-entity is a word for a specific entity, such as person, city, or time. For example, consider the sentence: Jim bought 300 shares of Acme Corp. in 2006. We can recognize Jim as a person, Acme Corp. as an organization, and 2006 as time.

Named-entity recognition (NER for short) is an important topic of natural language processing. But since words in a sentence have been represented by vectors, NER becomes a simple classification task. Precisely, suppose we have a set of named-entity classes, like `person`, `orgnization`, `time`, etc. We also have a specific class, labeled by `<O>`, for words that are not named-entity, such as is, walk, and good. The task turns to be classifying the word into any of these classes. This can be done by sending the context-dependent vector representation to a feed-forward neural network appended by a softmax function.

4.3. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding* by Jacob Devlin and others, 2018.

4.4. This is a sketch of how BERT works. The $q(x_t, x_{t'}; \varphi)$ is implemented by

$$q(x_t, x_{t'}; \varphi) := (W_K x_t)^T \cdot (W_Q x_{t'}),$$

where W_K and W_Q are both $n \times n$ matrices, and $\varphi := (W_K, W_Q)$. In practice, before sending to feed-forward layer, the $\bar{x}_t$ is linearly transformed as $W_V \bar{x}_t$ where W_V is a $n \times n$ matrix. Residual structure and normalization (discussed in section 3.7) are employed for controlling gradients. They also employ techniques "multi-head attention" for enriching the information propagation and "word-piece" for avoiding the out-of-vocabulary problem. As an example for illustrating how vector representation of words is made in practice, we simply omit these details, referring the reader to their original paper.

4.2 Representation of Sentences

4.2.1 From Words to Sentences: Firth's Idea Continued

4.5

4.2.2 Human Languages Are Recursive (Maybe)

4.2.3 Application: Textual Similarity

4.2.4 Application: Machine Translation

4.3 Language Modeling

4.5. *Universal Sentence Encoder* by Daniel Cer and others, 2018. A better explanation of universal sentence encoder can be found here.