Notes on 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, but being converted to chemical signals, called *neurotransmitters*. These neurotransmitters are molecules released from synapse. They are then caught by 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 the 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 → firing → impulses → synapses → neurotransmitters → densities → depolarization →$\cdots$.

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.

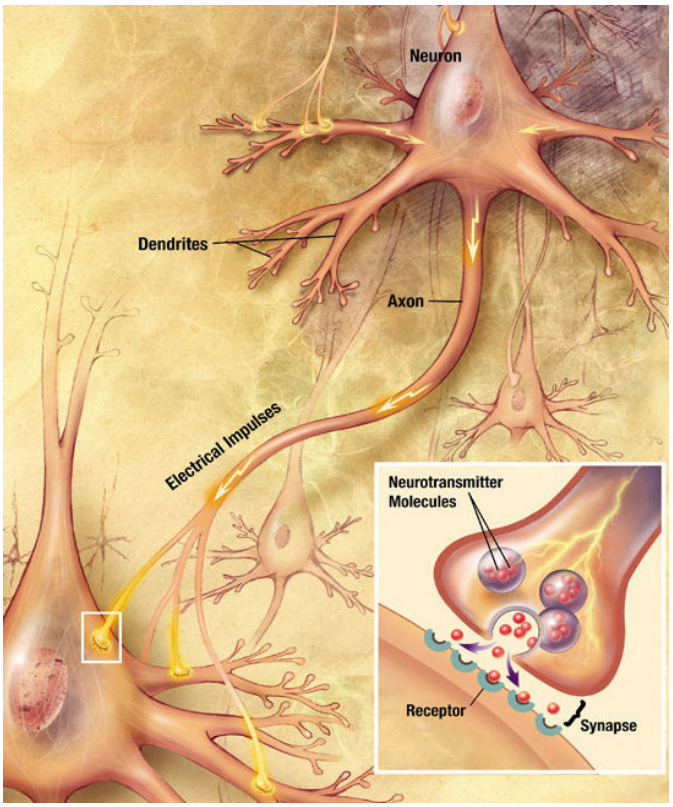

Figure 1.1. This figure illustrates the shape of a neuron, and how impulses propagate from one neuron to another.

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 Has Not Sufficient Capacity

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 $x(t)\equiv\hat{x}(t)$.

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.

Definition 1.1. *[Capability of Data-Fitting] Let $d(\cdot,\cdot)$ represent a distance. Given a dataset $D:=\{(x_i,y_i)|i=1,\dots,N\}$, and a parameterized function (model) $f(x;\theta)$ with x the input and θ the collection of parameters. We say the model $f(x,\theta)$ has the capability of fitting the dataset D, if for any $\varepsilon>0$, we can find a $\theta_\star$ such that*

$$\mathbb{E}_{(x,y)\sim D}[d(f(x;\theta_\star),y)]<\varepsilon,$$

where $\mathbb{E}_{(x,y)\sim D}$ represents the expectation with (x,y) uniformly sampled from D. That is, we can find a set of parameters $\theta_\star$ such that the model can accurately predicts or fits dataset, at arbitrary precision.

So, we claim that perceptron cannot fit the dataset generated by XOR function (the dataset is the truth table of XOR), let alone the dataset generated by an arbitrary function.

1.4 With Hidden Layers, Model Can Fit Any Dataset

1.4.1 Activation Function

To fit any dataset, we have to generalize perceptron from dealing with binary value to manipulate real number. This means, the activation function of perceptron can 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[1.3], ReLU[1.4], softplus[1.5], and even identity function[1.6]. In the subsequent, the word perceptron refers to that with the generalized activation function.

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

1.3. Sigmoid function can be seen as a shifted tanh. We have $\mathrm{sigmoid}(x):=1/(1+\exp(-x))$.

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

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

1.6. Forsooth, it is $y=x$.

1.4.2 Hidden Layer

From 1989 to 1991, researchers found that if we compose multiple perceptrons (with arbitrary activation function) together, it then obtains the ability to fit any dataset.[1.7] Explicitly, we use more than one perceptron, and use the output of a perceptron as the input of another perceptron. For example, let $z^\alpha = f(\sum_{\beta=1}^{n} U_\beta^\alpha x^\beta + c^\alpha)$ and $y^\alpha = g(\sum_{\beta=1}^{m} W_\beta^\alpha z^\beta + b^\alpha)$, where f and g represents the (generalized) activation function. The first equation represents a perceptron with input x; the second another perceptron that takes the output of the previous perceptron, z, as input. Altogether, it becomes a model with x as input and y as output. This composed model is named as *feed-forward neural network* or *multi-layer perceptron*, and each perceptron inside is called a *layer*. The layer that furnishes model output is called *output layer*; while other layers are named as *hidden layers*. In our example, we have a neural network with two layers, one of which is hidden layer.

In fact, as it was proven, when the g is identity function, feed-forward neural network can fit any dataset.[1.8] This is true even it has only one hidden layer. So, hidden layer is the key to success.

1.7. A brief history can be found here.

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

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 $d(\cdot, \cdot)$. That is, $d(\hat{y}_i, y_i)$ 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 $\sum_{(x_i, y_i) \in D} d(f(x_i; \theta), y_i)$ which should be small enough.

But, when the size of dataset D is extremely large, computing $\sum_{(x_i, y_i) \in D} d(f(x_i; \theta), y_i)$ will become difficult. In practice, we sample sufficient many data from D, and compute the expectation of $d(f(x; \theta), y)$ on these samples instead. Thus, a proper quantity that characterizes performance is

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

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

$$\theta_\star = \operatorname{argmin} L_D(\theta).$$

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 based methods to find the minimum of L_D without any constraint.

2.2 Going Along Gradient Direction May Not Be Optimal (TODO)

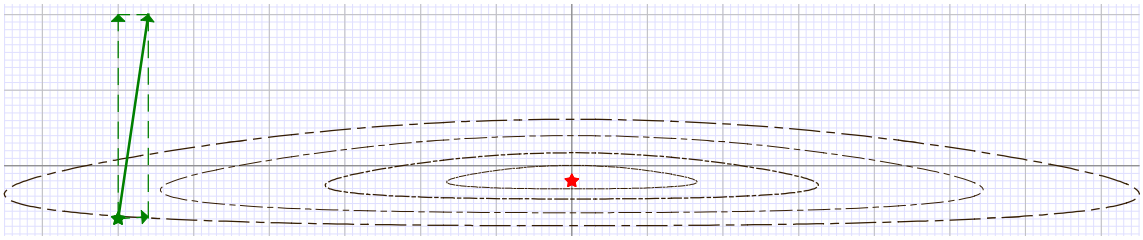

Figure 2.1. The black dash curves represents the contour map of a function $f: \mathbb{R}^2 \to \mathbb{R}$. The red star is the top or argmin f. The green star is where you are. The green lines represent the negative direction of gradient based on the contour map and its decomposition along horizontal and vertical directions.

The critical problem of gradient based optimization is how to reach the optimum with the minimum number of iteration steps. But, going along the negative direction of the gradient of loss function may not be optimal. Figure 2.1 shows a typical example where the negative direction of gradient (the solid green arrow) does not point to the optimum (the red star), but a direction which is almost irrelevant to the optimum.

To deal with this situation, we have to increase the horizontal component of the green vector, and decrease the vertical one.

$(g_{t-n}, g_{t-n+1}, \dots, g_t)$. $s_t^\alpha := (g_{t-n}^\alpha)^2 + \cdots + (g_t^\alpha)^2$. $\hat{g}^\alpha := g^\alpha / \sqrt{s^\alpha + \epsilon}$.
$s_{t+1}^\alpha \to (1-\gamma)\, s_t^\alpha + \gamma\, (g_{t+1}^\alpha)^2$.

2.3 Validation Helps Avoid the Instability of Optimization (TODO)

2.4 Moment Helps Avoid Stochastic Disturbance (TODO)

$\hat{g}_{t+1}^\alpha = (1-\gamma)\, \hat{g}_t + \gamma g_{t+1}$.
$v_{t+1}^\alpha = (1-\gamma_1)\, v_t^\alpha + \gamma g_{t+1}$.
$s_{t+1}^\alpha \to (1-\gamma)\, s_t^\alpha + \gamma\, (g_{t+1}^\alpha)^2$
$\hat{g}_{t+1}^\alpha := v_{t+1}^\alpha / \sqrt{s_{t+1}^\alpha + \epsilon}$

2.5 A Little History about Optimizer (TODO)

2.6 Gradient Is Computed by Vector-Jacobian Product [2.1]

2.6.1 From Feed-Forward Neural Network to General Composition

Recall in section 1.4, 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

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

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

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

2.6.2 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

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

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 pseudo-code as

```
def sigmoid_vjp(x: Vector):
    y: Vector = 1 / (1 + exp(-x))

    def grad(dy: Vector) -> Vector:
        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

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

2.6.3 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.

2.6.4 Backward Propagation

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

$$\frac{\partial l}{\partial \theta^\alpha_i} = \sum_\beta \frac{\partial l}{\partial \hat{y}^\beta} \sum_\gamma \frac{\partial \hat{y}^\beta}{\partial z^\delta_{n-1}} \sum_\delta \frac{\partial z^\gamma_{n-1}}{\partial z^\delta_{n-2}} \cdots \sum_\varphi \frac{\partial z^\varepsilon_{n-i+1}}{\partial z^\varphi_{n-i}} \frac{\partial z^\varphi_{n-i}}{\partial \theta^\alpha_{n-i}},$$

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.

Chapter 3
When Neural Network Becomes Deep

3.1 Enlarging Model Is Efficient for Increasing Its Representability

When fitting data using a function with trainable parameters, it is well-known that the more parameters it has, the better representability it will be, as long as these parameters are independent (or non-degenerate).[3.1]

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.

3.2 Increasing Depth Is More Efficient for Enlarging Model

3.2.1 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

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

3.1. This viewpoint has been examined by scaling law. Researchers in OpenAI found that, when the model has been large enough, the validation loss has a strong dependence with the number of trainable parameters, but is insensitive to the hyper-parameters of architecture such as width-depth-ratio (see section 3.1 of the paper).

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.

3.2.2 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.

3.2.3 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 3.2.1, 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.

3.2.4 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.

3.3 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.[3.2] 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.

3.4 Techniques Are Combined for Controlling the Gradients

3.4.1 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

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

3.2. 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 $f_l'(\sum_{\gamma=1}^{n_l} (W_l)_{\gamma}^{\alpha} z_{l-1}^{\gamma} + b_l^{\alpha})$.

3.4.2 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.

3.4.3 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 $\|z_{l-1}\| \sim 1$.

3.4.4 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.[3.3]

3.5 A Little History about Depth

3.5.1 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.

3.5.2 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 $(x_1, \ldots, x_T)$ 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 $t = 1, \ldots, T$ 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; $\ldots$; 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.

3.5.3 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.

3.3. 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`.

3.5.4 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.[3.4] 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.

3.5.5 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 $l+1$.[3.5]

3.5.6 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.[3.6]

3.5.7 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.

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

3.5. 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.

3.6. 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 Vector 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

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

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 parameter t indicates the position of word w in context. For example, for predicting fox out of its company, we shall compute $p(w|\mathsf{the}, \mathsf{quick}, \mathsf{brown}, \mathsf{jumps}, \mathsf{over}, \mathsf{the}, \mathsf{lazy}, \mathsf{dog}; 4)$ for each word w in $(\mathsf{the}, \mathsf{quick}, \mathsf{brown}, \mathsf{fox}, \mathsf{jumps}, \mathsf{over}, \mathsf{lazy}, \mathsf{dog}, \dots)$, 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

$$v_{w_t} := f(w_1, \dots, w_{t-1}, w_{t+1}, \dots, w_T; t, \theta), \tag{4.2}$$

where $f(\dots; t, \theta): \mathcal{V}^{T-1} \to \mathbb{R}^n$ 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 $\mathrm{softmax}_\alpha(x) := \exp(x_\alpha)/\sum_\beta \exp(x_\beta)$. Since softmax is positive definite and $\sum_\alpha \mathrm{softmax}_\alpha(x) = 1$, 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 $\mathrm{softmax}_2$ 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 $\mathcal{V}_{\mathsf{quick}} := \mathcal{V}_2 = \mathsf{quick}$. 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,

$$p(w|w_1, \dots, w_{t-1}, w_{t+1}, \dots, w_T; t) = \mathrm{softmax}_w(Uv_{w_t}).$$

We further 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 $(u_{\mathsf{quick}})_\alpha := U_{2\alpha}$ for $\alpha = 1, \dots, n$. This results in

$$p(w|w_1, \dots, w_{t-1}, w_{t+1}, \dots, w_T; t) = \frac{\exp(u_w \cdot v_{w_t})}{\sum_{x\in\mathcal{V}} \exp(u_x \cdot v_{w_t})}, \tag{4.3}$$

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.

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]

$$L(\theta, U) := -\mathbb{E}_{(w_1,\dots,w_T)\sim\mathrm{corpus}, t\sim(1,\dots,T)}\left[\ln \frac{\exp(u_{w_t} \cdot v_{w_t})}{\sum_{x\in\mathcal{V}} \exp(u_x \cdot v_{w_t})}\right], \tag{4.4}$$

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.

As a summary, we build 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 $f(\dots; t, \theta)$ in equation (4.2). Different researchers propose different forms of $f(\dots; t, \theta)$, with different complexities, resulting in different performances.

4.1.3 Example: Bidirectional Encoder Representations from Transformers

4.3

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.

4.3. The original paper can be found in arXiv.

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. So, the task is classify the word into any of these classes.

4.2 Representation of Sentences

4.3 Machine Translation

4.4 Language Modeling

4.5 Drafts

4.5.1 `Word2vec` Is a Simple Model that Implements the Firth's Idea

In 2013, Tomas Mikolov and others proposed a simple model that encodes the words into n-dimensional real vectors.[4.4] The underlying idea is that of Firth: words that company a word give the meaning of the word. There are two questions: what does company mean explicitly, and how these words give the meaning of a word. The answers are:

- the words surround a word in a sentence company the word; and
- from these words, the central word can be predicted.

For example, consider the sentence: the quick brown fox jumps over the lazy dog. If we take into account four neighbors, then the word fox will have neighbors quick, brown, jumps, and over. These four words company the central fox.

Then, to predict fox out of its four neighbors, the authors employed a simple probabilistic model. First, they assumed that the probability is "separable", as

$$p(\text{fox}|\text{quick},\text{brown},\text{jumps},\text{over}) = p(\text{fox}|\text{quick})\,p(\text{fox}|\text{brown})\,p(\text{fox}|\text{jumps})\,p(\text{fox}|\text{over}).$$

This indicates that the positional relationship is omitted. For modeling each conditional probability, they assigned two n-dimensional real vectors for each word in a given vocabulary. For example, let vocabulary $\mathcal{V} = (\text{the}, \text{quick}, \text{brown}, \text{fox}, \text{jumps}, \text{over}, \text{lazy}, \text{dog}, \dots)$ in which each word is represented by two vectors $u, v \in \mathbb{R}^n$. They then built a simple model for predicting fox out of quick, brown, etc, as

$$p(w|w') = \frac{\exp(u_w \cdot v_{w'})}{\sum_{x\in\mathcal{V}} \exp(u_x \cdot v_{w'})}. \tag{4.5}$$

For example, w and w' denote fox and quick respectively. The vector u represents for the center word, which is to be predicted, and the vector v for one of the neighbors. In this model, the vectors u and v for each word in vocabulary $\mathcal{V}$ are trainable parameters. They are adjusted so as to maximize the probabilities like $p(\text{fox}|\text{quick},\text{brown},\text{jumps},\text{over})$. In this way, fox is "known", or predicted, by the words quick, brown, jumps, and over that company it. This model is called word-to-vector, or word2vec for short. Either u_w or v_w is employed as the vector representation of a word w.[4.5]

4.4. The original paper can be found in arXiv.

4.5. In the paper, the authors employed the vector v as representation.

What does the equation (4.5) really mean? 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. So, if we collect the u vectors for all the words in the vocabulary and concatenate them together as a $|\mathcal{V}| \times n$ matrix U, then equation (4.5) can be re-written as

$$p(w|w') = \text{softmax}_w(Uv_{w'}),$$

where the $Uv_{w'}$ is a matrix-vector multiplication, and the softmax_w is the w-component of the categorical probability. Noticing that $\mathcal{V}$ is a list, the w-component represents the position of the word w in $\mathcal{V}$. Thus, softmax_w is the probability of encountering the word w (given its neighbor w'). Now the model is found to be a linear transformation of the vector $v_{w'}$ with an additional softmax/probabilistic output. Can you find a simpler model for the same task?

4.5.2 `Word2vec` in Practice

After building vocabulary, we iterate over the corpus again to fetch each word w and its neighbors $\{w_1, \ldots, w_N\}$ where N is a hyper-parameter. We assign an n-dimensional real vector u_w to the word w and v_{w_i} for each neighbor w_i. Thus, the loss function for this datum can be written as

$$l(u, v_{w_1}, \ldots, v_{w_N}) := -\ln\left(\prod_{i=1}^{N} \frac{\exp(u_w \cdot v_{w_i})}{\sum_{x\in\mathcal{V}} \exp(u_x \cdot v_{w_i})}\right).$$

As usually, the total loss function will be

$$L(u, v) := \mathbb{E}_{(w,\{w_1,\ldots,w_N\})\sim\text{copus}}[l(u, v_{w_1}, \ldots, v_{w_N})].$$

The authors of `word2vec` argued that, when the vocabulary size becomes large, the term $\sum_{x\in\mathcal{V}} \exp(x \cdot v_{w_i})$ in loss function will become very compute intensive.

4.5.3 Lost and Found

There are, however, many drawbacks of `word2vec`. The first is the omission of positional information. In many situations, this will not be a problem. For example, if we exchange the positions of fox and quick, it becomes: the fox brown quick jumps over the lazy dog. Because of the flexion of words, a reader can understand it without any difficulty. Another problem is much more serious: words can be polysemy. As an example, ass has the meanings of fool, hip, and being sexy, but it has at most two vectors of representation.

Even though `word2vec` is not a faithful modeling of the Firth's idea, many interesting results are revealed. First of all, the words with similar meanings are similar in their vector representations, characterized by the cosine similarity between the vectors. A more interesting result is the relative relation between words, like $v_{\text{king}} - v_{\text{man}} \approx v_{\text{queen}} - v_{\text{woman}}$, or $v_{\text{big}} - v_{\text{biggest}} \approx v_{\text{small}} - v_{\text{smallest}}$.

4.6 The Second Example: People in a Park

Now, for vector representation of words, the main problem turns to be determining the explicitly form of the model $f(w_1, \ldots, w_{t-1}, w_{t+1}, \ldots, w_T; \theta)$ in equation (?).

Recall that, in `word2vec`, the positional information of the words in context is omitted. But, in fact, position can also be encoded. So, we can regard w_t as a pair (w_t, t) where the second variable labels the position of the word w_t in the context $(w_1, \ldots, w_T)$.

We first assign each w_t and t n-dimensional real vectors respectively, and then add them together, resulting in one n-dimensional real vector for the pair (w_t, t), denoted by x_t. Then, these vectors, each for a word, are mutually mixed. Explicitly, it computes the "proximity" from $x_{t'}$ to x_t for each $t, t' = 1, \ldots, T$ with $t \neq t'$, determined by

$$W_{tt'} := \frac{\exp(x_t \cdot (K^T K) \cdot x_{t'})}{\sum_{t''=1}^{T} \exp(x_{t''} \cdot (K^T K) \cdot x_{t'})},$$

and $w_{tt}=0$, where K is an $m \times n$ matrix with m arbitrary. Then, compute the weighted output

$$z_t := \sum_{t'=1}^{T} W_{tt'} x_{t'}.$$

The greater the $W_{tt'}$ is, the more contribution from $x_{t'}$ to z_t.

Next, let FFN_θ a feed-forward neural network parameterized by θ, the output is $v_{w_t} := \mathrm{FFN}_\theta(z_t)$.