Neural Networks

Masahiro Sakuta

Contents

1. A little bit of history

The history of neural networks is like a series of emerging waves. It oscillates between periods of hyped and lost interest from the society like Figure 1.

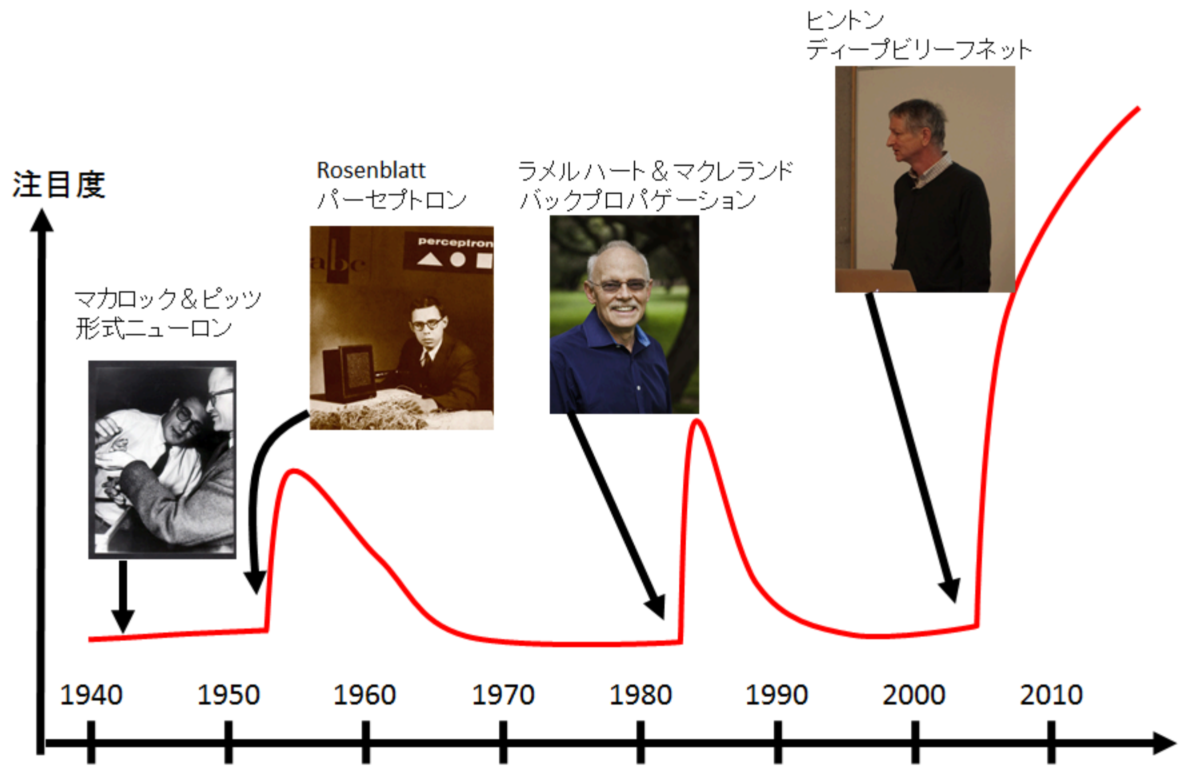

Figure 1: A brief timeline of neural networks

Its origins can be traced back to the 1940s. It is based on the formal neuron model proposed by Warren McCulloch and Walter Pitts, but the most important in history is the perceptron published in 1958 by

Frank Rosenblatt. However, the perceptron was difficult to scale up with the hardware of the time, and efficient learning methods had not been established, so the learning speed did not reach a practical level.

After that, neural networks entered a dark age for a while, but a brief revival occurred in the 1980s when backpropagation using the sigmoid function was invented. However, it still entered another dark age again because it did not produce a practical level of performance.

The tide has changed at the 2009 NIPS Workshop, where the Deep Belief Net implemented by Geoff Hinton produced a better score than any other state-of-the-art algorithms at that time. Around that time, convolutional neural networks were getting popular. They are good at image recognition in particular. Since then, artificial intelligence is dominated by deep learning to the point that just the word AI implies deep learning.

Figure 2: Probably the most famous historical photo of AI history, a portrait of Frank Rosenblatt

2. The model of the Perceptron

The activation function of the first model of perceptron proposed by Rosenblatt was a step function like Equation 1

$$S(x) = \begin{cases} 1 & 0 < x \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

This was because the model of a neuron was thought to have active and inactive states like Figure 3.

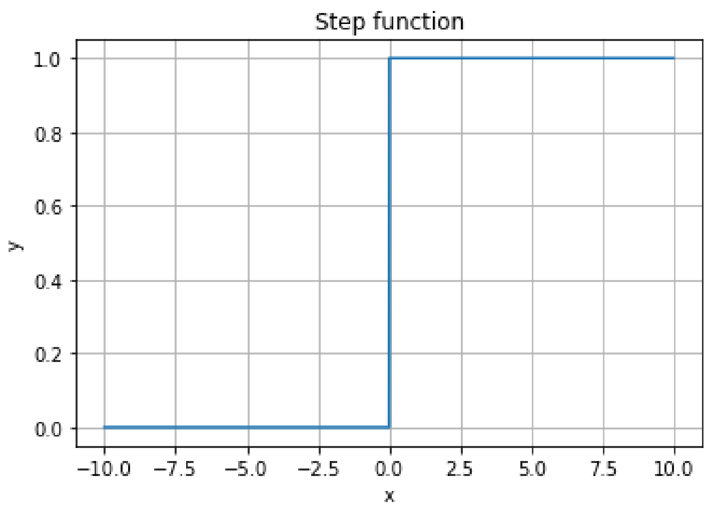

Figure 3: A step function

However, sigmoid function became popular for an activation function since it became clear that derivative is necessary for backpropagation. This function is advantageous for activation function since it has derivatives over the whole range of input variable, but it doesn't have a basis on real neurons.

A sigmoid function is defined like Equation 2 and it looks like Figure 4.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

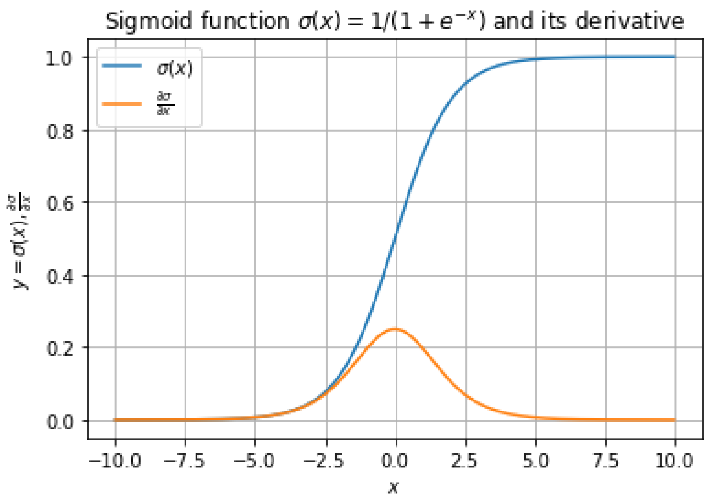

Figure 4: A sigmoid function

The sigmoid function is good and smooth, but the output range is fixed between 0 and 1, so it cannot represent an arbitrary function and has an issue called vanishing gradient. Therefore, the sigmoid function is not without its shortcomings.

As a result, ReLU (Regularized Linear Unit) which is defined as Equation 3 is often used these days. It looks like Figure 5. This function does not require a relatively expensive exponential function, since it is simple combination of identity and constant functions. Another advantage of it is that it has a value range from 0 to ∞, and the derivative can be defined over the entire range of the variable. At this point, the analogy of biological neuron is like thrown out of the window.

$$R(x) = \begin{cases} x \; 0 < x \\ 0 \text{ otherwise} \end{cases} \tag{3}$$

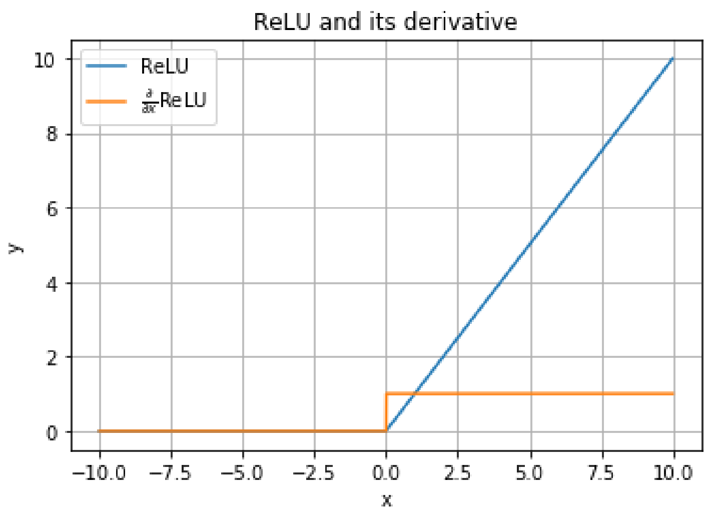

Figure 5: A ReLU function

Note that you could think of the simplest activation function as the identity transformation $f(x) = x$. If you use it, however, no matter how deep the neural network architecture you are using, it can be collapsed into a single linear transformation, so it would be incapable of representing nonlinearity. Also, since the entire network can be collapsed into one linear transformation, there would be no point having intermediate layers.

There are a lot of variants of ReLU, in attempt to overcome the limitation of flat derivative part. One of them is SiLU (Sigmoid Linear Unit), a sigmoid multiplied by a proportional function. It looks like ReLU as a whole, but the derivative is not zero over the whole value range.

$$\sigma(x) = \frac{x}{1 + e^{-x}} \tag{4}$$

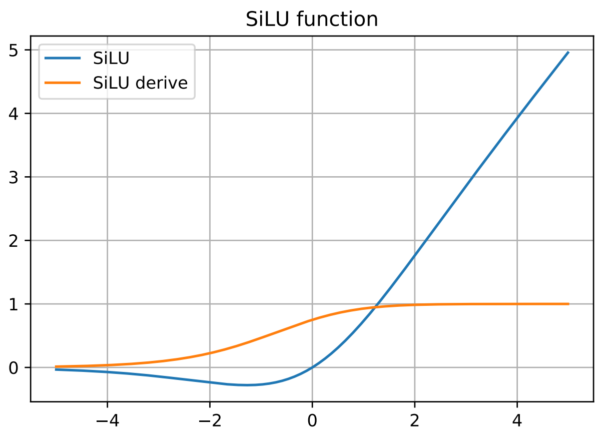

Figure 6: A SiLU function

3. Feed Forward Neural Network

Feedforward simply means transmitting a signal in the forward direction.

For example, consider a simple neural network like Figure 7.

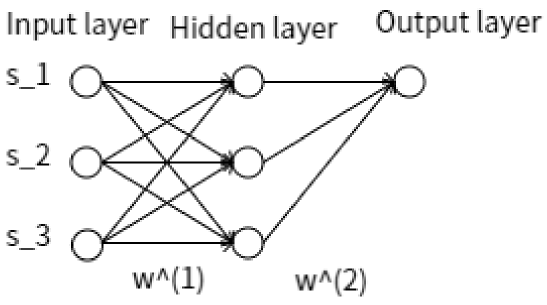

Figure 7: Feedforward neural network

The input is represented as $s_i(i = 1, 2, ..., n(1))$, the weights of the first layer as $w_{ij}^{(1)}(j = 1, 2, ..., n(2))$, then the signal from the input layer to the hidden layer $s_j^{(1)}$ is transmitted as Equation 5.

$$s_j^{(1)} = f\left(\sum_{i=1}^{n} w_{ij}^{(1)} s_i\right) \tag{5}$$

here $f(\cdot)$ is the activation function described in the previous section.

Furthermore, the weights of the second layer are $w_{jk}^{(2)}$, the signal in the output layer $s_k^{(2)}$ becomes

$$s_k^{(2)} = f\left(\sum_{j=1}^{n} w_{jk}^{(2)} s_j^{(1)}\right). \tag{6}$$

From the discussion so far, it is obvious that the number of hidden layers can be increased arbitrarily.

You could also use matrix multiplication to represent Equation 5 or Equation 6.

$$\begin{aligned} \boldsymbol{s}^{(1)} &= f\left(W^{(1)}\boldsymbol{s}\right) \\ \boldsymbol{s}^{(2)} &= f\left(W^{(2)}\boldsymbol{s}^{(1)}\right) \end{aligned} \tag{7}$$

Here, W is a matrix and $\boldsymbol{s}$ is a column vector.

Parallel computing utilizing GPU is really good at this kind of computation. There are many math libraries that can do this, such as MATLAB, SciPy, NumPy, and Pandas.

4. Bias term

In practice, we often put a constant input in each layer, unaffected by previous layers. These inputs are called biases.

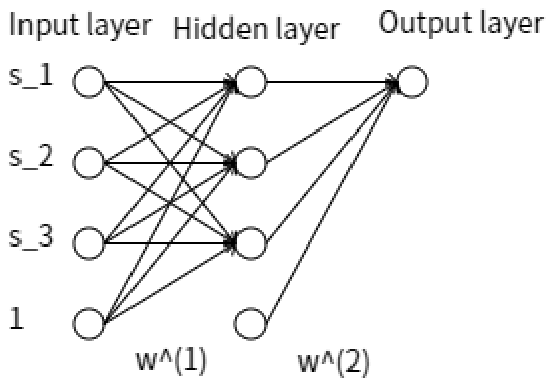

Figure 8: A neural network with a bias term

The usefulness of this is to have constant degrees of freedom in the arguments of the activation function. For example, Let the last signal in the input layer s_n be a constant value 1. Then we can transform Equation 5 into

$$s_j^{(1)} = f\left(w_{0j} + \sum_{i=1}^{n} w_{ij}^{(1)} s_i\right) \tag{8}$$

This is effectively the same as shifting the origin of the activation function by $-w_{0j}$. In this way, the offset of the origin can be included in the weight matrix and the offset need not be considered as a separate optimization target. In Rosenblatt's perceptron, the activation threshold is a separate parameter from the weight, which increases the complexity of multi-layering.

We can also add a bias term to the intermediate layer.

$$s_k^{(2)} = f\left(w_{0k}^{(2)} + \sum_{j=1}^{n} w_{jk}^{(2)} s_j^{(1)}\right) \tag{9}$$

A more concise way to put it is to include a constant term in the input signal. So $\boldsymbol{s}$ will be $(s_1, s_2, ...s_n, 1)$ an we can simply write

$$\begin{aligned} \boldsymbol{s}_j^{(1)} &= f\left(W^{(1)}\boldsymbol{s}\right) \\ \boldsymbol{s}_j^{(2)} &= f\left(W^{(2)}\boldsymbol{s}^{(\mathbf{1})}\right) = f\left(W^{(2)} f\left(W^{(1)} s\right)\right) \end{aligned} \tag{10}$$

5. Backpropagation

When training a neural network, a technique called backpropagation is used.

First, let's define the cost function L, which is a indication of the magnitude of the deviation from ground truth answer. This is defined in terms of the ground truth A_k and the predicted answer $s_k^{(2)}$ as Equation 11. Here, the reason why the deviation from the ground truth is squared isn't entirely clear, but it will become clear in a later section.

$$L = \frac{1}{2} \sum_{k\left(A_k - s_k^{(2)}\right)}^{2} \tag{11}$$

5.1. Gradient of the Weights on the First Layer

First, let's think about how we can modify the output layer weights $w_{jk}^{(2)}$ so that the prediction get closer to the correct answer. It means finding the direction of motion in $\boldsymbol{w}$ space that decreases L, so we can take gradient of L with respect to each element of $\boldsymbol{w}$. In other words, compute $\frac{\partial L}{\partial w_{jk}^{(2)}}$. We can use this gradient and the learning rate parameter δ to define the update rule for the $\boldsymbol{w}$ as Equation 12. The weight of the next step can be obtained by subtracting the gradient multiplied by δ.

$$w_{jk}^{(2)}\Big|_{t+1} = w_{jk}^{(2)}\Big|_{t} - \delta \frac{\partial L}{\partial w_{jk}^{(2)}}\Bigg|_{t} \tag{12}$$

This corresponds to the gradient descent method in the optimization problem, and we will discuss later in dropout whether it falls into a local solution in a multifaceted solution space.

Calculating the second term on the right-hand side of Equation 12, from the chain rule of composite functions,

$$\begin{aligned} -\delta \frac{\partial L}{\partial w_{jk}^{(2)}} &= -\delta \frac{\partial L}{\partial s_k^{(2)}} \frac{\partial s_k^{(2)}}{\partial w_{jk}^{(2)}} \\ &= \delta \left(A_k - s_k^{(2)}\right) \frac{\partial s_k^{(2)}}{\partial w_{jk}^{(2)}} \end{aligned} \tag{13}$$

and can be easily calculated from only the difference between the teacher signal answer and the predicted answer.

Let's focus on the second half $\frac{\partial s_k^{(2)}}{\partial w_{jk}^{(2)}}$. Since the variables are related with the activation function f through Equation 6 like below. we can simplify the calculation by placing an intermediate variable $p_k^{(2)}$, which is the argument given to f.

$$p_k^{(2)} \equiv \sum_{j=1}^{n} w_{jk}^{(2)} s_j^{(1)} \tag{14}$$

Here we can use the chain rule again to yield

$$\frac{\partial s_k^{(2)}}{\partial w_{jk}^{(2)}} = \frac{\partial s_k^{(2)}}{\partial p_k^{(2)}} \frac{\partial p_k^{(2)}}{\partial w_{jk}^{(2)}}. \tag{15}$$

$\frac{\partial s_k^{(2)}}{\partial p_k^{(2)}}$ is the derivative of the activation function f'.

Also we can use

$$\begin{aligned} \frac{\partial p_k^{(2)}}{\partial w_{jk}^{(2)}} &= \frac{\partial}{\partial w_{jk}^{(2)}} \sum_{j=1}^{n} w_{jk}^{(2)} s_j^{(1)} \\ &= s_j^{(1)} \end{aligned} \tag{16}$$

to represent

$$\frac{\partial s_k^{(2)}}{\partial w_{jk}^{(2)}} = s_j^{(1)} f'\left(p_k^{(2)}\right) \tag{17}$$

Once again we write the entire Equation 12 as

$$w_{jk}^{(2)}\Big|_{t+1} = w_{jk}^{(2)}\Big|_{t} + \delta\left(A_k - s_k^{(2)}\right)s_j^{(1)} f'\left(p_k^{(2)}\right) \tag{18}$$

From here, it is obvious that backpropagation requires that the activation function can be derived.

Here we can use a little trick when the activation function is a sigmoid function.

$$\begin{aligned}
\frac{\partial \sigma}{\partial x} &= \frac{\partial}{\partial x}\frac{1}{1+e^{-x}} \\
&= e^{-x}\frac{1}{\left(1+e^{-x}\right)^2} \\
&= \frac{1+e^{-x}-1}{1+e^{-x}}\frac{1}{1+e^{-x}} \\
&= \left(1 - \frac{1}{1+e^{-x}}\right)\frac{1}{1+e^{-x}} \\
&= (1-\sigma)\sigma
\end{aligned} \tag{19}$$

Using this, Equation 12 can be further rewritten as follows:

$$w_{jk}^{(2)}\Big|_{t+1} = w_{jk}^{(2)}\Big|_{t} + \delta\left(A_k - s_k^{(2)}\right)s_j^{(1)}\left(1 - s_k^{(2)}\right)s_k^{(2)} \tag{20}$$

5.2. Backpropagating to the first layer

Now, let's focus on the hidden layer weights $w_{ij}^{(1)}$. We can replace the weight of $w_{ij}^{(1)}$ in Equation 12 like below.

$$w_{ij}^{(1)}\Big|_{t+1} = w_{ij}^{(1)}\Big|_{t} - \delta\frac{\partial L}{\partial w_{ij}^{(1)}}\Bigg|_{t} \tag{21}$$

However, the method of applying the chain rule will change. Since there is an index variable k which is neither the final indices i, j, the whole derivative becomes total derivative, not the partial derivative.

$$\begin{aligned}
\frac{\partial L}{\partial w_{ij}^{(1)}} &= \sum_k \frac{\partial L}{\partial s_k^{(2)}}\frac{\partial s_k^{(2)}}{\partial w_{ij}^{(1)}} \\
&= \sum_k \frac{\partial L}{\partial s_k^{(2)}}\frac{\partial s_k^{(2)}}{\partial s_j^{(1)}}\frac{\partial s_j^{(1)}}{\partial w_{ij}^{(1)}} \\
&= \sum_k \underbrace{\frac{\partial L}{\partial s_k^{(2)}}}_{(1)}\underbrace{\frac{\partial s_k^{(2)}}{\partial p_k^{(2)}}}_{(2)}\underbrace{\frac{\partial p_k^{(2)}}{\partial s_j^{(1)}}}_{(3)}\underbrace{\frac{\partial s_j^{(1)}}{\partial w_{ij}^{(1)}}}_{(4)}
\end{aligned} \tag{22}$$

Wow, there are a lot of factors! Let's disect them into pieces.

(1) we already calculated $\frac{\partial L}{\partial s_k^{(2)}} = A_k - s_k^{(2)}$.

(2) we already calculated $\frac{\partial s_k^{(2)}}{\partial p_k^{(2)}} = f'\left(p_k^{(2)}\right)$.

(3) can be calculated using Equation 14.

$$\begin{aligned} \frac{\partial p_k^{(2)}}{\partial s_j^{(1)}} &= \frac{\partial}{\partial s_j^{(1)}} \sum_{j=1}^{n} w_{jk}^{(2)} s_j^{(1)} \\ &= w_{jk}^{(2)} \end{aligned} \tag{23}$$

In summary, this whole summation can be reused to reduce the computation in the next layer. Let's call it $L_j^{(1)}$ for it being the "loss function" for the next layer.[1] Note that it is a vector of j elements.

$$\begin{aligned} L_j^{(1)} &\equiv \sum_{k=1}^{n_k} \frac{\partial L}{\partial s_k^{(2)}} \frac{\partial s_k^{(2)}}{\partial p_k^{(2)}} \frac{\partial p_k^{(2)}}{\partial s_j^{(1)}} \\ &= \sum_{k=1}^{n_k} \left(A_k - s_k^{(2)}\right) f'\left(p_k^{(2)}\right) w_{jk}^{(2)} \end{aligned} \tag{24}$$

This is the crucial part of the backpropagation. We are considering a neural network with 2 layers, but you can imagine extending this logic to many layers indefinitely.

Now, we want to calculate (4), but to make them simplier, let's define an intermediate variable like before:

$$p_j^{(1)} = \sum_{i=1}^{n} w_{ij}^{(1)} s_i \tag{25}$$

Using this, we can write (4) as

$$\begin{aligned} \frac{\partial s_j^{(1)}}{\partial w_{ij}^{(1)}} &= \frac{\partial s_j^{(1)}}{\partial p_j^{(1)}} \frac{\partial p_j^{(1)}}{w_{ij}^{(1)}} \\ &= f'\left(p_j^{(1)}\right) \frac{\partial}{\partial w_{ij}^{(1)}} \sum_{i=1}^{n} w_{ij}^{(1)} s_i \\ &= s_i f'\left(p_j^{(1)}\right) \end{aligned} \tag{26}$$

The whole expression becomes

$$\frac{\partial L}{\partial w_{ij}^{(1)}} = L_j^{(1)} s_i f'\left(p_j^{(1)}\right) \tag{27}$$

With a neural network of only few layers, we cannot really feel the benefits of reusing the previous layer's computation. However, the number of repeated computations grow exponentially as the layers get deeper, so this technique is essential to the large scale deep learning. The idea of this algorithm is a bit like FFT butterfly operation.

Calculation is performed in order from the output side to the input side, so it is called backpropagation.

6. Batch training

Loss function given in Equation 11 was the result for only one sample. If we calculate all N samples together, we get the following (we are running out of space for subscripts, but the superscript $[l]$ indicates the lth sample):

[1] Technically, it is a *derivative* of the loss function, but we don't want to put any more fancy notations.

$$L = \frac{1}{2}\sum_{l=1}^{N}\sum_{k}\left(A_k^{[l]} - s_k^{(2)[l]}\right)^2 \tag{28}$$

Since the partial derivatives are independent for each sample, the second term of the right side of Equation 12 is as follows.

$$\begin{aligned}
-\delta\frac{\partial L}{\partial w_{jk}^{(2)}} &= -\delta\sum_{l=1}^{N}\frac{\partial L}{\partial s_k^{(2)[l]}}\frac{\partial s_k^{(2)[l]}}{\partial w_{jk}^{(2)}} \\
&= -\delta\sum_{l=1}^{N}\left(A_k - s_k^{(2)[l]}\right)\frac{\partial s_k^{(2)[l]}}{\partial w_{jk}^{(2)}} \\
&= -\delta\sum_{l=1}^{N}\left(A_k - s_k^{(2)[l]}\right)s_j^{(1)[l]}f'\left(\sum_{j=1}^{N}w_{jk}^{(2)}s_j^{(1)[l]}\right)
\end{aligned} \tag{29}$$

It is obvious that the gradient of the hidden layer can be calculated in the same way.

Batch training is a method of determining the gradient direction from the loss functions of multiple samples at once. On the other hand, as we saw in the previous section, calculating the gradient direction from each sample one by one is called online training or stochastic gradient descent. There is also a method called mini-batch, where you train repeatedly with a sample size in bite-sized chunks. Each has advantages and disadvantages as described below.

- Batch training has high stability and fast convergence because it descends a gradient that smoothes the data with variability.
- Online training is swayed by noise in individual data, so convergence is poor, but it can be applied to very large training data that cannot be stored in memory.

7. Appendix

7.1. All You Need to Know about Derivatives

You should know a little calculus to understand how neural networks work (or any machine learning algorithms in that regard). Your high school should have taught you about it, but the world is a diverse place now and I don't know if you had that opportunity.[2]

I have tried to teach how the machine learning works to several people, and found out that some people have really hard time understanding it. Right now my theory why is that these people didn't understand basic math including calculus.

7.1.1. Ordinary derivatives

The definition of derivative is as follows:

$$\frac{df}{dx} = \lim_{\varepsilon\to 0}\frac{f(x+\varepsilon) - f(x)}{\varepsilon} \tag{30}$$

The notion like $\frac{df}{dx}$ was invented by Leibniz, who is also credited as one of the inventors of the calculus. The other person in credit that invented calculus at the same time (how that could happen was an interesting story for another time) is Newton. He also invented his version of notation, like $\dot{f}$. It is sometimes written as f' too, although I don't know who invented it.

[2]Surprisingly many people don't have a clear idea of basic calculus, even in the field of AI research. I hope the reader is not one of them, or quit being an imposter like them.

The Leibniz notation has an advantage that it indicates a variable that derive f with respect to. It is especially handy with partial derivatives. The Newton notation implies that the function is derived with respect to time variable, so it has limits when you want to apply it to other variables.

Let's do some exercies with basic functions. Consider a function Equation 31.

$$f(x) = x^2 \tag{31}$$

Let's put it into Equation 30.

$$\frac{df}{dx} = \lim_{\varepsilon \to 0} \frac{(x + \varepsilon)^2 - x^2}{\varepsilon} \tag{32}$$

You can calculate the denominator inside the limit as follows:

$$\begin{aligned} (x + \varepsilon)^2 - x^2 &= x^2 + 2\varepsilon x + \varepsilon^2 - x^2 \\ &= \varepsilon(2x + \varepsilon) \end{aligned} \tag{33}$$

Now, we can reduce the ε to 0, since nothing is in the factor of ε.

$$\frac{df}{dx} = \lim_{\varepsilon \to 0}(2x + \varepsilon) = 2x \tag{34}$$

You won't calculate the derivatives like this every time, but you can find tables of derivatives of common functions on the web. Basic ones like polynomials are worth memorizing since they are not too complex.

$$\frac{dx^n}{dx} = nx^{n-1} \tag{35}$$

Another important technique to calculate derivatives is the chain rule. Suppose we had a function $f(x)$ which is in turn a variable of another function $g(f(x))$. Then we can calculate the derivative of g with respect to x using this "chain of derivatives".

$$\frac{dg}{dx} = \frac{dg}{df}\frac{df}{dx} \tag{36}$$

It is not very clear with this abstract notation, so let's use an example functions like below.

$$\begin{aligned} g(f) &= f^2 \\ f(x) &= x + a \end{aligned} \tag{37}$$

Here, a is a constant.

We can calculate the derivative of each function like below.

$$\begin{aligned} \frac{dg}{df} &= 2f \\ \frac{df}{dx} &= x \end{aligned} \tag{38}$$

So the result of the whole derivative is:

$$\frac{dg}{dx} = 2x(x + a) \tag{39}$$

Remember that you can define the intermediate function however you like. So if your function is given like this:

$$g(x) = (x + a)^2 \tag{40}$$

you can see that you can define an intermediate function $f(x) = x + a$. This is one of the most basic technique to derive a complicated function.

There are few other techniques like derivative of products, but I won't go into details.

7.1.2. Partial derivatives

Partial derivatives are derivatives on a function with multiple independent variables. For example, suppose we have a function with 2 variables, $f(x, y)$. The notion of each partial derivative with respect to each variable is like below.

$$\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \tag{41}$$

When you compute a partial derivative, you fix the other variables as if they were constants. For example, let's take a function like below.

$$f(x, y) = x^2 + y^2 \tag{42}$$

Partial derivatives are calculated like below.

$$\frac{\partial f}{\partial x} = 2x, \frac{\partial f}{\partial y} = 2y \tag{43}$$

7.1.3. Total derivatives

A less known variant of derivatives is called total derivatives. It is a derivative of a function with multiple variables, like partial derivatives, but it incorporates both variables.

$$df = \frac{\partial f}{\partial x} dx + \frac{\partial f}{\partial y} dy \tag{44}$$

The notable thing about total derivatives is that it doesn't specify a derived variable. In a sense, it is a template of derivatives that can adapt to any variable.

It is useful when the variables x, y are not independent. For example, if they are both dependent on the third variable, z, we can write like $x(z), y(z)$. Then, we can calculate like below:

$$\frac{df}{dz} = \frac{\partial f}{\partial x}\frac{dx}{dz} + \frac{\partial f}{\partial y}\frac{dy}{dz} \tag{45}$$

Another way to put it is that it is kind of a version of chain rule with multiple variables.

7.1.4. Conclusion

As you can see, the value of derivative is that it reduces the idea of taking limits (which drives minds crazy) into mere manipulation of symbols. It is one of the most successful mathematical tools and still a valuable tool in the age of computers to analyze complicated models in the world, as you are witnessing with deep learning.