Playing with MNIST

Author: Manatsu Takahashi (takahashi.manatsu@gmail.com)
Last Modified: 2020-04-10 09:06:13+09:00

Abstract

In this tutorial paper, we learn what is and how to use Neural Network, which is an algorithm in the field of machine learning. We supply both theoretical explanations and a program which learns MNIST dataset to recognize hand-written digits. As for the program, it is written in C++ (not usual Python) without external library and the code structure is explained in detail. The final purpose is to create a web application in which a user draw a line and the program infers what the digit is. It should be easy to apply neural networks to another problem after reading this paper. All the source codes are available in github.com...

Contents

1 Introduction

1.1 Optimization

Optimization is a method to pull the best solution out of all possible solutions [1]. Each solution is tagged with an *energy* which determines how good the solution is. Normally we associate a lower energy with a better solution. Then, if we can somehow decrease energy, we will finally achieve the best solution.

As an example, let's consider a problem where we are required to divide the Japanese Islands except for Hokkaido[*1] as equal as possible into four divisions with respect to their areas. The division should be done in units of prefectures, meaning each prefecture cannot be broken into smaller parts and it as a whole should be stored in a division. This problem can be solved, taking these steps.

1. First we have to define energy. The definition is actually arbitrary, but it is natural we define energy E as

$$E \equiv \sum_i \left| \frac{(\text{total area})}{4} - A_i \right| \tag{1.1}$$

   where A_i is the area of ith division. The lowest energy 0 means the Islands are perfectly equally separated.

2. Then we create an initial state. Let $D \equiv \{1, 2, 3, 4\}$ and d_i be a division number assigned to each prefecture, which means ith prefecture now belong to ith division. By initializing $\{d_i\}$ with random integers whose range is given by D, we get a random initial state; each prefecture is included in a random division.

3. Next we start to decrease the energy E. We randomly pick up a prefecture j and again randomly change the value of its division number d_j. For example, if the current value of d_j is 2, we randomly select an integer n from the set $D \backslash \{d_j\} = \{1, 3, 4\}$ and assign n to d_j. After that, we check the difference $\Delta E \equiv E_{\text{new}} - E_{\text{old}}$, where E_{new} and E_{old} are energies before or after this assignment respectively. If $\Delta E < 0$, that is, if the assignment has decreased the energy, we accept the change. Otherwise, we reject the assignment and revert the change.

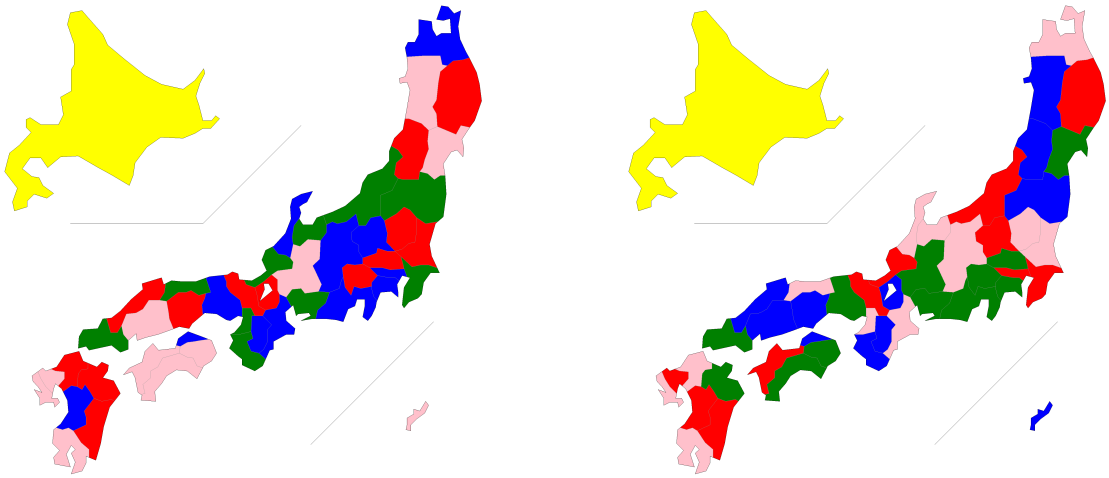
4. We repeat the step 3 until E gets smaller than some small value ϵ. The specific value of ϵ is case-by-case but usually it does not correspond to the minimum possible energy since getting the very lowest energy is difficult especially when we execute numerical (i.e. non-analytic) optimization.

That's all. See Figure 1.1 for an example initial state and an example final state. We also supply the animation `fig_1_1.gif` which illustrates how a calculation proceed.

One of the most important thing is that optimization can be applied to any sorts of problems as far as we can define energy. Even when a problem can be analytically solved, optimization is often adopted as there are quite many optimization algorithms which give a sufficiently good solution in a short time[*2].

[*1] "Hokkaido" is the name of the largest prefecture in Japan. Hokkaido is so large that its area a satisfies $\frac{1}{5} < \frac{a}{A} < \frac{1}{4}$ where A is the total area of the Islands.

[*2] Consider the problem "which is the shortest path from the station A to the airport B?", that a car navigation system usually encounter. Although the analytical solution is needed if we really mean "most short" by "shortest", it is not rare that a sufficiently short path works well. If that's the case, we may want to use an optimization algorithm to get a two-minute-fifteen-second solution in a few hundreds of milliseconds rather than using an analytical algorithm (e.g. Dijkstra's algorithm) to get the shortest two-minute-ten-second solution in a few second.



(a) initial state ($E = 26487.5$)

(b) final state ($E = 0.04$)

Figure 1.1: An example initial state and an example final state. Each color corresponds to an element in the set D (i.e. a division number) and top-left Hokkaido is exceptionally colored yellow. The energy 0.04 means the area difference among divisions is within 0.04 (km^2), which is about the area of *Tokyo Dome* multiplied by 0.86.

1.2 Machine Learning

Machine learning is the scientific study of algorithms and seen as a subset of artificial intelligence. Such algorithms rely on patterns and inference instead of using explicit instructions [2]. Though machine learning handles a wide variety of problems such as *regression problem*, *binary classification problem* and *multiclass classification problem*, let's focus on the last one. Hereafter we assume *supervised learning*, in which the desired outputs (i.e. the correct answers) are given in addition to training data itself.

Multiclass classification is the problem to answer which category an input falls into. Assume we'd like to infer the gender of the person appeared in an input image. This type of problem is solved in these two phases:

1. In *training phase*, for each input image in *training data*, a model implemented in a computer reads it and outputs the inferred gender. By modifying internal parameters according to how the inferred outputs are different from the correct *label*s (i.e. the correct answers), the model learns the patterns found in the images to achieve higher recognition accuracy.

2. In *Inference Phase*, the model reads an arbitrary input image which is normally not included in the training data and outputs the inferred gender while the internal parameters are fixed. It is said to be *overfitting*, which should be avoided, if the model has low accuracy[*3] for inputs read in inference phase whereas it has high accuracy for the training data.

The learning process often boils down to an optimization problem [3], and how it proceeds depends on specific algorithms. In this paper, we mainly deal with *neural networks* among them.

[*3] How can we define accuracy for inputs fed in inference phase though we have no correct answers for them? To define accuracy, we prepare a set of data and the corresponding labels as *testing data* which is similar to training data but is reserved not to be used in training phase.

2 Perceptron

2.1 What is Perceptron?

Before discussing neural networks, we introduce *perceptron*. Perceptron, devised in 1975, is a function which takes one or more boolean values as its arguments and returns a boolean value. Each input x_i is multiplied by the weight w_i bound to a perceptron and the returned value y is determined by the equations

$$y = \begin{cases} 0 & (S' \leq t) \\ 1 & (S' > t) \end{cases} \tag{2.1}$$

$$S' \equiv \sum_i x_i w_i \tag{2.2}$$

where t is some threshold specific to the perceptron. A perceptron is said to be *firing* if $y = 1$. By defining *bias* $b \equiv -t$, the equations are rewritten as

$$y = \begin{cases} 0 & (S \leq 0) \\ 1 & (S > 0) \end{cases} \tag{2.3}$$

$$S \equiv \sum_i x_i w_i + b. \tag{2.4}$$

We can structure a *network*, a.k.a. a *graph*, by arranging multiple perceptrons, which have generally different $\{w_i\}$ and b, and defining the relationship between them. Then each perceptron is called a *neuron* or a *node* and each bridge which connects two different perceptrons is called an *edge*. In this case, an edge has its direction, meaning a pulse is transferred from a *source* node to a *target* node. See Figure 2.1 for an example.

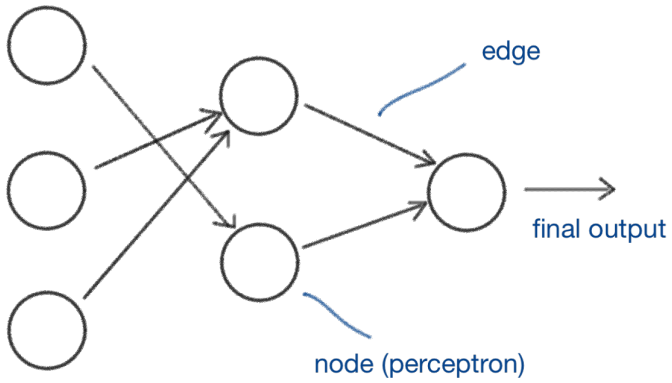


Figure 2.1: An example network which consists of six perceptrons.

2.2 Logic Gate

A *boolean function* is a function of the form $f : \mathbb{B}^k \to \mathbb{B}$, where $\mathbb{B} = \{0, 1\} = \{\texttt{false}, \texttt{true}\}$ is a *boolean domain* [4]. And a *logic gate* is an entity implementing a boolean function [5]. It is clear a perceptron behaves as a boolean function or a logic gate. In this subsection, we see a single perceptron is not quite flexible and also see, fortunately, this fact is not a bad news. Hereafter we mean "a logic gate with $k = 2$" just by "a gate", and only in this subsection refer to "a perceptron with two inputs" just as "a perceptron".

2.2.1 AND Gate

An *AND gate* is a gate which returns `true` if and only if both inputs are `true`. The corresponding *truth table* is shown as Table 2.1.

Input 1	Input 2	Output
0	0	0
0	1	0
1	0	0
1	1	1

Table 2.1: Truth table for an AND gate.

A perceptron becomes an AND gate if

$$w_1 + w_2 > t \tag{2.5}$$
$$w_1 \leq t \tag{2.6}$$
$$w_2 \leq t \tag{2.7}$$
$$0 \leq t. \tag{2.8}$$

There are infinite ways of choosing the parameters which satisfy these conditions (e.g. $(w_1,\ w_2,\ t) \equiv (0.5,\ 0.5,\ 0.7)$).

2.2.2 NAND Gate

A *NAND gate* is a gate which, as its name "Not-AND" implies, returns `false` if and only if both inputs are `true`. The corresponding truth table is shown as Table 2.2.

Input 1	Input 2	Output
0	0	1
0	1	1
1	0	1
1	1	0

Table 2.2: Truth table for an NAND gate.

A perceptron becomes a NAND gate if

$$w_1 + w_2 \leq t \tag{2.9}$$
$$w_1 > t \tag{2.10}$$
$$w_2 > t \tag{2.11}$$
$$0 > t. \tag{2.12}$$

There are infinite ways of choosing the parameters which satisfy these conditions (e.g. $(w_1,\ w_2,\ t) \equiv (-0.5,\ -0.5,\ -0.7)$). In fact, just negating all the parameters of an AND gate gives an NAND gate as far as

$$w_1 \neq t \tag{2.13}$$
$$w_2 \neq t \tag{2.14}$$
$$t \neq 0. \tag{2.15}$$

These restrictions are needed since the original equations eq.(2.5), ..., eq.(2.8) are not of the symmetric forms[*4].

[*4] Reversing "$\leq$" we get not "$\geq$" but "$>$". This gives rise to an asymmetry.

2.2.3 OR Gate

An *OR gate* is a gate which returns `true` if at least one of the inputs is `true`. The corresponding truth table is shown as Table 2.3.

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	1

Table 2.3: Truth table for an OR gate.

A perceptron becomes an OR gate if

$$w_1 + w_2 > t \tag{2.16}$$
$$w_1 > t \tag{2.17}$$
$$w_2 > t \tag{2.18}$$
$$0 \leq t. \tag{2.19}$$

There are infinite ways of choosing the parameters which satisfy these conditions (e.g. $(w_1,\ w_2,\ t) \equiv (0.5,\ 0.5,\ 0.4)$).

Independent of our choice to use a perceptron as an AND gate, a NAND gate or an OR gate, the only different things are the parameters; the structure of the perceptron itself is not changed. In other words, just by changing the values of the parameters, a perceptron's behavior may completely be changed. Further, it is necessary to mention the fact that, when we use a network consisting of perceptrons to calculate something, it is possible and easy to tweak the parameters "during" the calculation, which leads to the change of the way how the calculation proceed. So a perceptron is quite flexible, right? No, it depends.

2.2.4 XOR Gate

Let's consider a gate called an *XOR gate*, or an *Exclusive OR gate*. It returns `true` if only one of the two inputs is `true`. See Table 2.4 for the truth table.

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Table 2.4: Truth table for an XOR gate.

A perceptron becomes an XOR gate if

$$w_1 + w_2 \leq t \tag{2.20}$$
$$w_1 > t \tag{2.21}$$
$$w_2 > t \tag{2.22}$$
$$0 \leq t \tag{2.23}$$

, but it is crystal-clear these conditions are self-contradicting. Contrary to the fact that an XOR operation is used so frequently that many programming languages implement it as a builtin operator (e.g. `^`), it is never be expressed by a single perceptron.

It is possible to invent a more intuitive explanation. As shown in Section 2.1, the output y of a perceptron is given by

$$y = \begin{cases} 0 & (S \leq 0) \\ 1 & (S > 0) \end{cases} \tag{2.24}$$

$$S \equiv w_1 x_1 + w_2 x_2 + b. \tag{2.25}$$

$S = 0$ is an equation of a line which separates the $x_1 - x_2$ plane into two parts and each area is expressed by $S \leq 0$ or $S > 0$. Thus, a specific gate can be implemented by a perceptron if and only if the four points $(0,0)$, $(0,1)$, $(1,0)$ $(1,1)$ can be separated following the truth table. For example, an AND gate can be implemented since it is possible to draw a linear line in such a way that only the point $(1,1)$ out of the four points is above the line. However, there is no way to draw a line such that the points $(0,1)$, $(1,0)$ belong to a division and the other two points belong to the other division. That's why an XOR gate cannot be implemented via an perceptron. Figure 2.2 illustrates this logic.

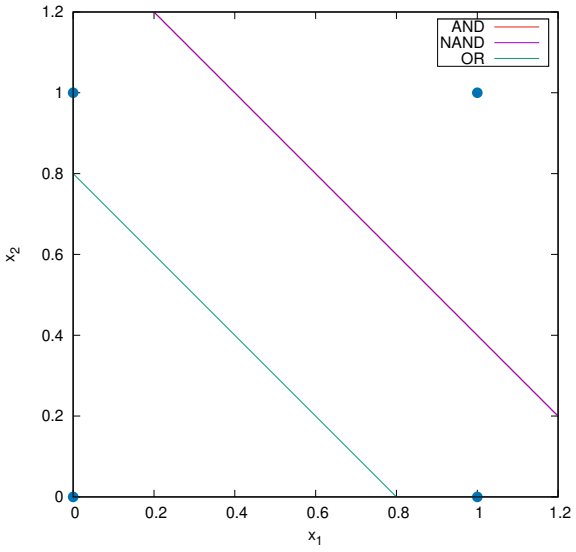


Figure 2.2: An example network which consists of six perceptrons.

Here's a good news: this limitation can be overwhelmed by using a *multilayer perceptron*. It is simply a collection of usual perceptrons arrayed in layers. The network in Figure 2.1 we showed above is an example of a multilayer perceptron. Since OR returns `false` only for $(0,0)$ and NAND returns `false` only for $(1,1)$, by taking AND of their outputs, we get the final result `true` only for $(0,1)$ or $(1,0)$. Thus the multilayer perceptron shown in Figure 2.3 behaves as an XOR gate.

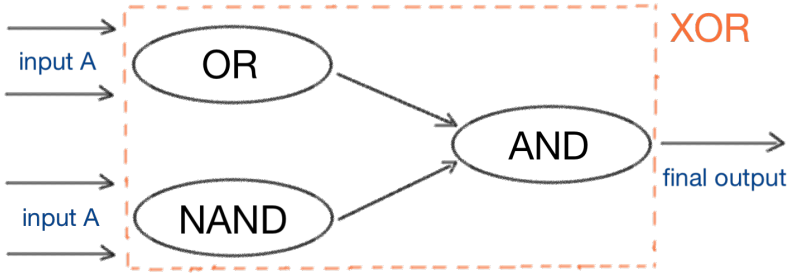


Figure 2.3: An XOR gate expressed by a multilayer perceptron. Note the same input A are fed into both nodes in the first layer.

2.3 summary

In this section, we learned these two properties of perceptron:

1. A perceptron behaves differently for different parameters.
2. A single perceptron cannot execute some sort of calculations. But is may be implemented by employing many perceptrons to create a multilayer perceptron.

Consequently, if it is possible automatically to change the parameters in response to inputs and the corresponding outputs, a multilayer perceptron becomes an automatically configured and dynamically growing processor.

3 Neural Network

3.1 What is Neural Network?

As described in Section 2.1, the output y of a perceptron follows eq.(2.3) and eq.(2.4). These equations can be rewritten as

$$y = h(S) \tag{3.1}$$

$$S \equiv \sum_i x_i w_i + b \tag{3.2}$$

$$h(x) \equiv \begin{cases} 0 & (x \leq 0) \\ 1 & (x > 0) \end{cases} \tag{3.3}$$

where h is a *step function*. In other words, the output is determined by taking the dot product $\boldsymbol{x} \cdot \boldsymbol{w}$, adding the bias b to it and finally applying the *activation function* h. A perceptron uses a step function as the activation function.

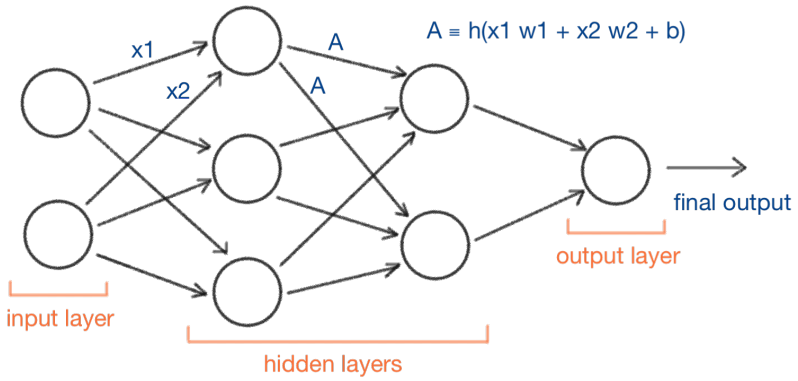


Figure 3.1: An example of a neural network. Though the network is drawn as a *fully connected* graph, meaning each node in a layer reads from all the nodes in the previous layer [6], an edge disappears when the corresponding weight is zero. The output layer need not consist of a single node.

A *neural network* is similar to a multilayer perceptron but uses arbitrary activation functions, which means each node in the network generally reads and outputs real (i.e. non-boolean) values. Layers in a neural network are fallen into the following categories.

1. *Input layer*: the first layer
2. *Hidden layers*: one or more layers placed in the middle of a network
3. *Output layer*: the last layer

See Figure 3.1 for an example structure.

3.2 Activation Functions

Which activation function is to be used is dependent of a model and a target problem. It is usual we use some activation function for hidden layers and another one for the output layer. In this subsection, we introduce a few activation functions frequently used.

3.2.1 Step Function

Step function is a function defined by the formula

$$h(x) \equiv \begin{cases} 0 & (x \leq 0) \\ 1 & (x > 0) \end{cases} . \tag{3.4}$$

As we've seen before, this function is used as the activation function for a perceptron. See Figure 3.2 for the plot.

3.2.2 Sigmoid Function

Sigmoid function is a function defined by the formula

$$h(x) \equiv \frac{1}{1 + \exp(-x)}. \tag{3.5}$$

This is the smoother version of step function and still returns a value in $[0, 1]$. See Figure 3.2 for the plot.

3.2.3 ReLU

Recently, *ReLU (rectified linear unit)* is often used as an activation function. It is defined as

$$h(x) \equiv \max(0, x). \tag{3.6}$$

See Figure 3.2 for the plot.

3.2.4 Identity Function

Identity function is a function which returns its input as-is.

$$h(x) \equiv x \tag{3.7}$$

It is used as the activation function for the output layer for regression problems.

3.2.5 Softmax Function

Finally, we introduce *softmax function* which is defined by the formula

$$h(\{x_i\}) \equiv \frac{\exp(\{x_i\})}{\sum_i \exp(x_i)} \tag{3.8}$$

where exp() should be applied element-wise. It is used as the activation function for the output layer for multiclass classification problems, which will be dug into in Section 4.

The most important property of softmax function is that the elements in a returned set ranges over $[0, 1]$ and sums up to unity. Therefore we may interpret each element as a probability. Since the function does not affect the magnitude relationship among inputs, and since calculating exp() is heavy, we usually replace the function with identity function in the inference phase.

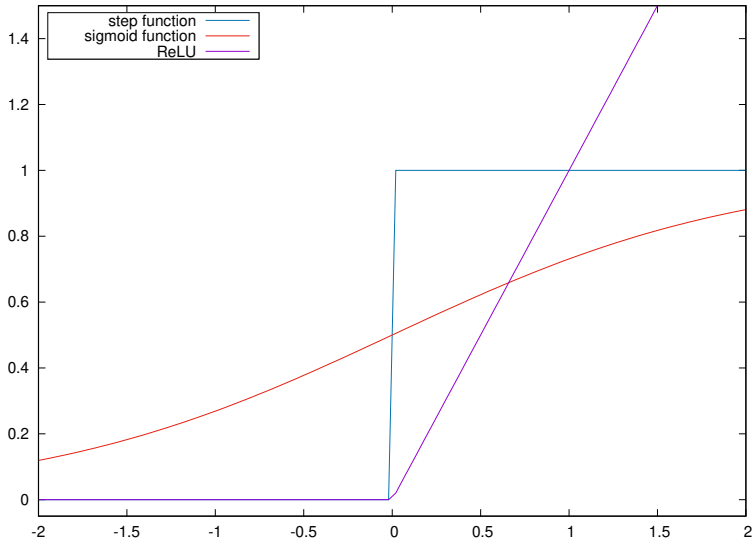


Figure 3.2: Plot of step function, sigmoid function and ReLU.

3.3 Loss Functions

labelsection:3.5

To perform machine learning using a neural network, as described in Section 1.2, we shall compare the network's outputs and the labels (i.e. desired outputs), and modify live the parameters such as weights and biases according to the difference. Such comparisons are done by *loss functions*. A loss function reads both each output from the nodes in the output layer and the corresponding label to return a single scalar value indicating how the current outputs are bad. Thus one of the purposes of the learning phase is to minimize the loss function our model adopt.

In this subsection, we simply introduce some loss functions. How we modify internal parameters by using the returned values is discussed in Section 3.5.

3.3.1 Mean Squared Error

Mean squared error is defined by the formula

$$E \equiv \frac{1}{2} \sum_k (y_k - t_k)^2 \tag{3.9}$$

where y_i is ith element of a network's output $\boldsymbol{y}$ and t_i is ith element of the corresponding label $\boldsymbol{t}$.

The problem is that a label is sometimes scalar while an array is output from a network. Assume we have the output "the input image is

$$\begin{pmatrix} \text{a dog with a probability of 0.2} \\ \text{a cat with a probability of 0.5} \\ \text{a rat with a probability of 0.3} \end{pmatrix}.\text{"}$$

and the label is just "the input image is a cat". In such cases, we forcibly convert a label to an array of the same dimensions. For example, "the input image is a cat" is rephrased as "the input image is

$$\begin{pmatrix} \text{a dog with a probability of 0} \\ \text{a cat with a probability of 1.0} \\ \text{a rat with a probability of 0} \end{pmatrix}.\text{"}$$

This is called *one-hot representation*.

3.3.2 Cross Entropy Error

Cross entropy error is defined by the formula

$$E \equiv -\sum_k t_k \ln(y_k) \tag{3.10}$$

where the meanings of t_k and y_k are described in Section 3.3.1.

If $\boldsymbol{t}$ is an array of probabilities as in the example in Section 3.3.1 and if $t_m = 1$, eq.(3.10) is simplified as

$$\begin{aligned} E &= -t_m \ln(y_m) \\ &= -\ln(y_m) \end{aligned} \tag{3.11}$$

since $t_n = 0$ for all $n(\neq m)$. In this case, E is dependent only of y_m and takes a larger value as y_m goes further away of 1.0. Figure 3.3 illustrates this behavior.

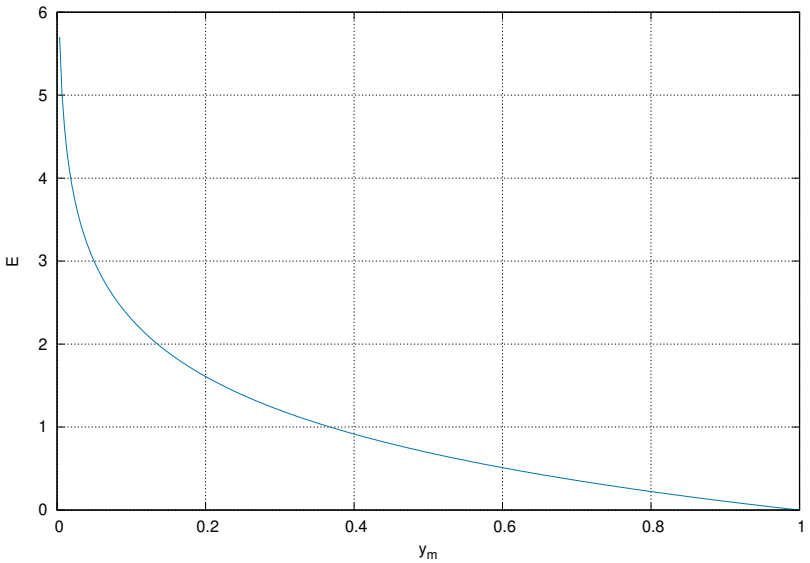


Figure 3.3: The behavior of cross entropy error when $t_m = 1$ and $t_n = 0$ for all $n(\neq m)$.

3.4 Forward Propagation

When we do a calculation in a forward order, that is, from the input layer to the output layer, this process is called a *forward propagation*. It is mathematically written down by matrix operations.

Assume we have two consecutive layers l_1 and l_2. Let N be the number of the nodes in l_1, M the number of the nodes in l_2, I a $1 \times N$ matrix whose elements work as inputs, W an $N \times M$ matrix whose ith column represents the weights of inputs for ith node in l_2, $\boldsymbol{b}$ a $1 \times M$ matrix whose ith element represents the bias of ith node in l_2, and h the activation function. The structure is illustrated in Figure 3.4. Then the output L of the layer l_2 is calculated by

$$L = h(IW + \boldsymbol{b}). \tag{3.12}$$

The operation $IW + \boldsymbol{b}$ is called a *affine transformation*. Since L is of the length M, this itself is used as the input to the next layer l_3. Therefore, one forward propagation is processed just by repeatedly calculating eq.(3.12).

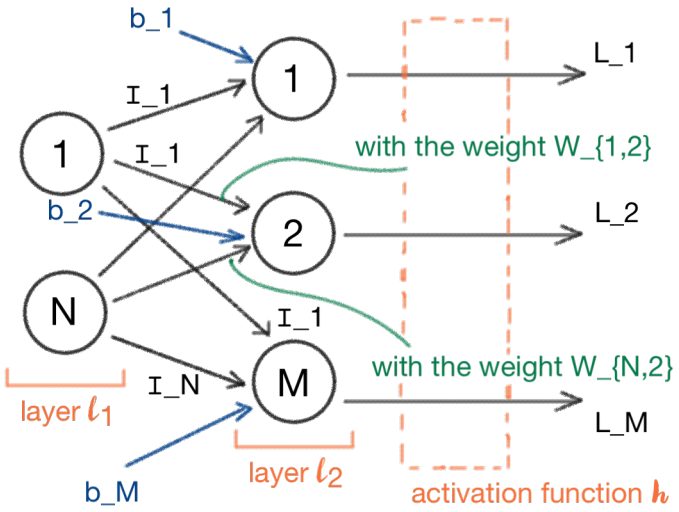


Figure 3.4: Illustration of the matrix operation in eq.(3.12).

This formulation is easily extended to deal with multiple Is at a time. When the number of I is H, we can prepare the new $H \times N$ matrix I by vertically aligning each of the original Is. By this, L also becomes an $H \times M$ matrix, and the definitions of part of activation/loss functions (e.g. softmax function) are naturally to be modified to process their inputs row-wise. Though this extension does not change the *time complexity*, we will benefit from it because matrix arithmetic for larger matrices is handled fairly efficiently by techniques like *multithreaded programming*, *vectorization*[*5] or *GPGPU*[*6]. The final problem we need tackle on is how to re-define the bias $\boldsymbol{b}$ to calculate the now broken $IW + \boldsymbol{b}$. Actually re-defining $\boldsymbol{b}$ is unnatural since the number of biases are not changed unless the number of nodes in a layer changes. Rather, it is more preferable that we *overload* the plus operator `+`; the addition of the $N \times M$ matrix A and the $1 \times M$ matrix $\boldsymbol{b}$ shall be calculated by performing the element-wise additions of $\boldsymbol{b}$ and each of the rows of A. For example,

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} + \begin{pmatrix} -1 & -2 & -3 \end{pmatrix} \coloneqq \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} + \begin{pmatrix} -1 & -2 & -3 \\ -1 & -2 & -3 \end{pmatrix}$$

[*5] Vectorization makes it possible to process multiple values via a single instruction. Normally it is explicitly implemented by users, but with the `-O3` flag `g++` enables automatic vectorization.

[*6] That is so-called GPU programming.

$$= \begin{pmatrix} 0 & 0 & 0 \\ 3 & 3 & 3 \end{pmatrix}. \tag{3.13}$$

Even when we have H' inputs, it is usual we feed only H of them, which is called a *minibatch*, into a network in a single forward propagation. One of the reasons is the *spatial complexity*. Finally, $\frac{H'}{H}$ forward propagations are called an *epoch*. Completing calculations for an epoch we can say the H' inputs are fully scanned except when we create each minibatch by randomly choosing H inputs from them, in which case the certain input may not be scanned or may be so twice or more.

3.5 Steepest Descent

As previously noticed in Section 3.5, now we explain one of the ways how to modify the internal parameters of a network in response to outputs of a loss function.

Gradient is an array of partial derivatives with respect to each variable. For example, in $x-y$ plane, the gradient of $f(x,y) = y^3 + (x+y)^2 + 3x$ is given as

$$\Delta f \equiv \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right) \tag{3.14}$$

$$= \left(2(x+y)+3,\ 3y^2 + 2(x+y)\right). \tag{3.15}$$

The gradient can be interpreted as the "direction and rate of fastest increase" [7]. Thus by calculating the gradient of the current position, moving a little in the opposite direction of the gradient, and repeating these two steps, we can find a local minimum. This algorithm is called *steepest descent* or *gradient descent*.

Using *central difference*, the partial derivative $\frac{\partial f}{\partial x}$ is numerically calculated by

$$\frac{\partial f}{\partial x} \simeq \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon} \tag{3.16}$$

where ϵ is a small constant (e.g. 0.01). By "moving a little in the opposite direction of the gradient $\Delta f(x,y)$", we mean the following assignments

$$x \leftarrow x - t\frac{\partial f}{\partial x} \tag{3.17}$$

$$y \leftarrow y - t\frac{\partial f}{\partial y} \tag{3.18}$$

where t is again a small value[*7] called a *learning rate*.

Since a loss function is a function of all of the internal parameters (i.e. the weights and the biases) of a network, we can modify the parameters by the steps shown in Algorithm 3.1. And this is actually "learning". Whilst the parameters are automatically tweaked, the value of a learning rate have to be set by hand. Such an entity is called a *hyperparameter*. Even one execution of the algorithm is very heavy since it includes $2n$ forward propagations, each of which has the complexity of $O(N^3)$ according to eq.(3.12), where n is the number of the internal parameters. One important note is that it is usual n is of the order 10^3 or 10^4 even for a simple network with a single hidden layer, making it difficult to use the algorithm practically for larger networks. In the next section, we introduce another method to calculate a gradient, which is much faster and whose result is almost the same as that given by central difference.

[*7] It is said t should not be too small. This is natural because, if we choose t so, we move little in each step and thus tend to be caught in a shallow local minimum. We can jump over such minima by preparing a larger value for t, though.

Algorithm 3.1 Steepest Descent by Central Difference

$\epsilon \leftarrow$ (a small value)
$t \leftarrow$ (a learning rate)
$P \leftarrow$ (an array of all the internal parameters)
$P' \leftarrow$ (an empty array)

for p in P **do**
  $p \leftarrow p + \epsilon$
  $E1 \leftarrow$ (the loss function) ▷ $f(x+\epsilon)$
  $p \leftarrow p - 2\epsilon$
  $E2 \leftarrow$ (the loss function) ▷ $f(x-\epsilon)$
  $p \leftarrow p + \epsilon$
  Push $\left(p - t\frac{E_1 - E_2}{2\epsilon}\right)$ to P'.
end for

$P \leftarrow P'$

3.6 Backward Propagation

3.6.1 Chain Rule

Theorem 3.1. (*Chain Rule*) When we have $f = f(q_1, ..., q_M)$ and $q_i = q_i(s_1, ..., s_N)$, the formula below is satisfied.

$$\frac{\partial f}{\partial s_i} = \sum_{k=1}^{M} \frac{\partial f}{\partial q_k} \frac{\partial q_k}{\partial s_i} \tag{3.19}$$

□

Proof 3.1. Omitted. ■

See Theorem 3.2 for the matrix version of this theorem.

3.6.2 Backpropagation

Say we have the network shown in Figure 3.5 and would have $\frac{\partial z}{\partial a}$, $\frac{\partial z}{\partial b}$ and $\frac{\partial z}{\partial c}$. Using central difference, these derivatives are calculated completely independently. But are they truly independent? No. Taking into account d, e, f are independent of a, b, c, the derivatives are broken by chain rule as

$$\begin{aligned} \frac{\partial z}{\partial a} &= \frac{\partial z}{\partial y}\frac{\partial y}{\partial a} \\ &= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial a} \end{aligned} \tag{3.20}$$

$$\frac{\partial z}{\partial b} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial b} \tag{3.21}$$

$$\frac{\partial z}{\partial c} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial c}. \tag{3.22}$$

Since they have $\frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$ as the common factor, we can just calculate the factor once and reuse it to be multiplied by the local derivatives $\frac{\partial x}{\partial a}$, $\frac{\partial x}{\partial b}$ and $\frac{\partial x}{\partial c}$. And note $\frac{\partial z}{\partial d} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial d}$ also includes the factor $\frac{\partial z}{\partial y}$ we've just seen.

In general, the derivative of the intermediate/final output of a network with respect to some variable is calculated by moving backward with the initial value 1 while calculating local derivatives and multiplying

each result to the current value. In the example of $\frac{\partial x}{\partial a}$ above, the calculation proceeds as $1 \to 1 \cdot \frac{\partial z}{\partial y} \to 1 \cdot \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x} \to 1 \cdot \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x} \cdot \frac{\partial x}{\partial a}$. By memorizing intermediate results, they can be reused to calculate other derivatives. These processes are called *backpropagation* since the current result propagates backward, that is, in the direction from the output layer to the input layer.

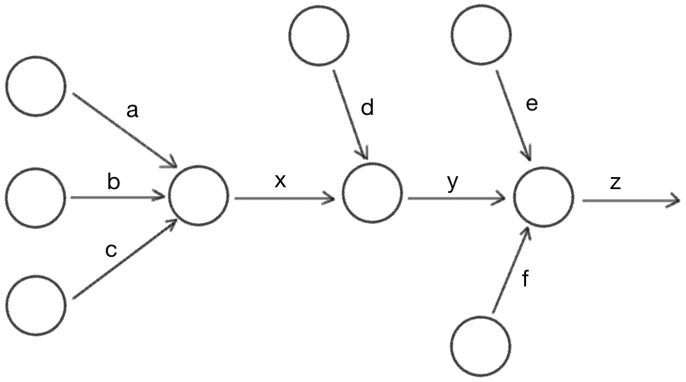


Figure 3.5: A sample network.

Actually we can even calculate the derivative with respect to the variables in a layer at the same time. Just aligning eq.(3.20) to eq.(3.22) we get

$$\frac{\partial z}{\partial \boldsymbol{a}} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial \boldsymbol{a}} \tag{3.23}$$

where $\boldsymbol{a} \equiv (a, b, c)$. Here we defined for any $N \times M$ matrix A and a scalar L,

$$\frac{\partial L}{\partial A} \equiv \begin{pmatrix} \frac{\partial}{\partial A_{11}} & \frac{\partial}{\partial A_{12}} & \cdots & \frac{\partial}{\partial A_{1M}} \\ \frac{\partial}{\partial A_{21}} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ \frac{\partial}{\partial A_{N1}} & \cdots & \cdots & \frac{\partial}{\partial A_{NM}} \end{pmatrix} L. \tag{3.24}$$

3.6.3 Chain Rule (Matrix Version)

As described in Section 3.4, we use matrix arithmetic to process propagations in a network rather than scalar arithmetic appeared in Section 3.6.2. Thus it is useful if we extend Theorem 3.1 to the matrix version.

Theorem 3.2. (*Chain Rule (Matrix Version)*) When we have a scalar function $g = g(Z(Y(X)))$ where X, Y, Z are matrices, the formula below is satisfied.

$$\left(\frac{\partial g}{\partial X}\right)_{lm} = \sum_{p,q} \frac{\partial g}{\partial Y_{pq}} \frac{\partial Y_{pq}}{\partial X_{lm}} \tag{3.25}$$

□

Proof 3.2.

$$\begin{aligned} \left(\frac{\partial g}{\partial X}\right)_{lm} &= \frac{\partial g}{\partial X_{lm}} \quad (\because \text{eq.(3.24)}) \\ &= \sum_{i,j} \frac{\partial g}{\partial Z_{ij}} \frac{\partial Z_{ij}}{\partial X_{lm}} \quad (\because \text{chain rule}) \end{aligned}$$

$$
\begin{aligned}
&= \sum_{i,j} \frac{\partial g}{\partial Z_{ij}} \sum_{p,q} \frac{\partial Z_{ij}}{\partial Y_{pq}} \frac{\partial Y_{pq}}{\partial X_{lm}} \quad (\because \text{chain rule}) \\
&= \sum_{p,q} \left(\sum_{i,j} \frac{\partial g}{\partial Z_{ij}} \frac{\partial Z_{ij}}{\partial Y_{pq}} \right) \frac{\partial Y_{pq}}{\partial X_{lm}} \\
&= \sum_{p,q} \frac{\partial g}{\partial Y_{pq}} \frac{\partial Y_{pq}}{\partial X_{lm}} \quad (\because \text{chain rule}) \qquad (3.26)
\end{aligned}
$$

■

3.6.4 Gradient Check

Although backpropagation is by far the faster than central difference and the two methods theoretically give the same results, the latter is still used to check if an implementation of the former is correct since backpropagation is relatively complicated. This comparison is called a *gradient check*.

3.7 Using Backward Propagation

Now let's put backward propagation into practice in a neural network. Hereafter we assume the network structure shown in Figure 3.6.

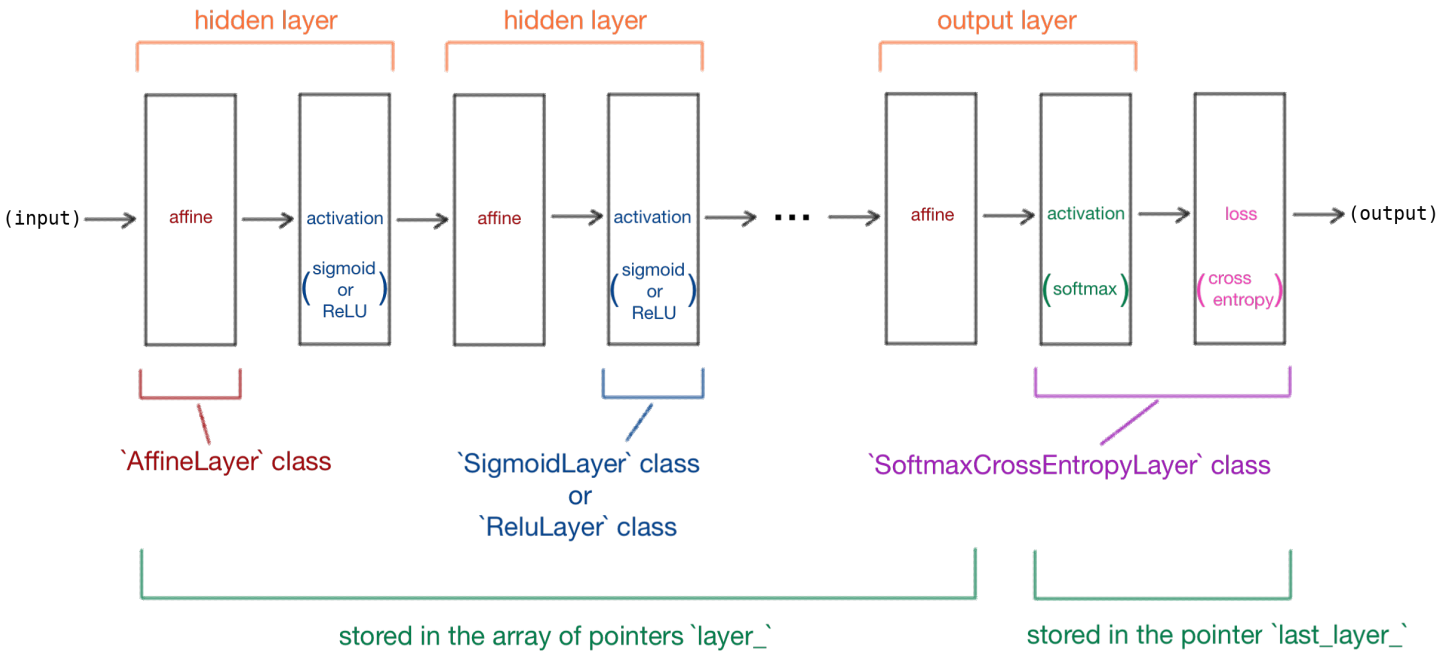


Figure 3.6: The network used in Section 3.7. Information about the pointers (i.e. the last line in the figure) will be presented later in Section 4.2.

Although conceptually each hidden layer consists both of an affine transformation and an activation function, and a loss function is not included in the definition of a network, for convenience we implement the following classes and assemble them to create a network.

1. `AffineLayer` class deals with only the part of an affine transformation (Section 3.4).
2. `SigmoidLayer` class deals with sigmoid function (Section 3.2.2) as an activation function.

3. `ReluLayer` class deals with ReLU (Section 3.2.3) as an activation function.

4. `SoftmaxCrossEntropyLayer` class deals with softmax function (Section 3.2.5) as the activation function for an output layer and cross entropy error (Section 3.3.2) as a loss function.

Let's dig into each of these classes in the reverse (i.e. backward) order.

3.7.1 SoftmaxCrossEntropyLayer class

In this subsubsection, only Theorem 3.4 is essential. Only eager readers may want to catch up Theorem 3.3 and Theorem 3.5.

Theorem 3.3. Let an array X be a raw[*8] output of a network, an array T a training data in one-hot representation, h softmax function and g cross entropy error.

$$h(A) = \frac{\exp(A)}{\sum_i \exp(A_i)} \quad (\because \text{eq.}(3.8)) \tag{3.27}$$

$$g(A,B) = -\sum_k B_k \ln(A_k) \quad (\because \text{eq.}(3.10)) \tag{3.28}$$

Then $\frac{\partial}{\partial X} g(h(X),T)$ is given by

$$\frac{\partial}{\partial X} g(h(X),T) = h(X) - T. \tag{3.29}$$

□

Note the rhs represents the difference between the final output from a network and the training data. This natural result is not a product of chance. In fact, this is why we use cross entropy error with softmax function. Similarly, to get the same result, we use mean squared error when identity function is applied instead of softmax function. See Theorem 3.5 for the detail.

Proof 3.3. We calculate the lth element of $\frac{\partial}{\partial X} g(h(X),T)$.

$$\begin{aligned}
\left(\frac{\partial g}{\partial X}\right)_l &= \frac{\partial g}{\partial X_l} \\
&= \sum_i \left(\frac{\partial g}{\partial h_i}\frac{\partial h_i}{\partial X_l} + \frac{\partial g}{\partial T_i}\frac{\partial T_i}{\partial X_l}\right) \quad (\because \text{chain rule}) \\
&= \sum_i \left(\frac{\partial g}{\partial h_i}\frac{\partial h_i}{\partial X_l} + \frac{\partial g}{\partial T_i}\cdot 0\right) \\
&= \sum_i \frac{\partial g}{\partial h_i}\frac{\partial h_i}{\partial X_l} \\
&= \sum_i \frac{\partial g}{\partial h_i}\frac{\partial}{\partial X_l}\frac{\exp(X_i)}{\sum_k \exp(X_k)} \quad (\because \text{eq.}(3.27)) \\
&= \sum_i \frac{\partial g}{\partial h_i}\left\{\delta_{il}\frac{\exp(X_i)}{\sum_k \exp(X_k)} - \frac{\exp(X_i)\exp(X_l)}{\left(\sum_k \exp(X_k)\right)^2}\right\} \\
&= \sum_i \left(-\frac{\partial}{\partial h_i}\sum_j T_j \ln(h_j)\right)\left\{\delta_{il}\frac{\exp(X_i)}{\sum_k \exp(X_k)} - \frac{\exp(X_i)\exp(X_l)}{\left(\sum_k \exp(X_k)\right)^2}\right\} \quad (\because \text{eq.}(3.28))
\end{aligned} \tag{3.30}$$

[*8] The final output is created by applying softmax function (or whatever) to the "raw" output.

$$
\begin{aligned}
&= \sum_i \left(-T_i \frac{1}{h_i}\right) \left\{ \delta_{il} \frac{\exp(X_i)}{\sum_k \exp(X_k)} - \frac{\exp(X_i)\exp(X_l)}{\left(\sum_k \exp(X_k)\right)^2} \right\} \\
&= \sum_i \left(-T_i \frac{\sum_j \exp(X_j)}{\exp(X_i)}\right) \left\{ \delta_{il} \frac{\exp(X_i)}{\sum_k \exp(X_k)} - \frac{\exp(X_i)\exp(X_l)}{\left(\sum_k \exp(X_k)\right)^2} \right\} \quad (\because \text{eq.}(3.27)) \\
&= -T_l \frac{\sum_j \exp(X_j)}{\exp(X_l)} \frac{\exp(X_l)}{\sum_k \exp(X_k)} + \sum_i T_i \frac{\sum_j \exp(X_j)}{\exp(X_i)} \frac{\exp(X_i)\exp(X_l)}{\left(\sum_k \exp(X_k)\right)^2} \\
&= -T_l + \sum_i T_i \frac{\exp(X_l)}{\sum_k \exp(X_k)} \\
&= -T_l + \frac{\exp(X_l)}{\sum_k \exp(X_k)} \quad \left(\because \sum_i T_i = 1\right) \\
&= (h(X))_l - T_l \quad (\because \text{eq.}(3.27)) \qquad (3.31)
\end{aligned}
$$

Thus we get

$$
\frac{\partial}{\partial X} g(h(X), T) = h(X) - T. \qquad (3.32)
$$

■

Theorem 3.4. Let a matrix X be an array of raw outputs of a network, a matrix T an array of training data in one-hot representation, h softmax function and g cross entropy error.

$$
\begin{aligned}
\{h(A)\}_{ij} &= \frac{\exp(A_{ij})}{\sum_k \exp(A_{ik})} \quad (\because \text{eq.}(3.8)) \qquad (3.33) \\
g(A, B) &= -\frac{1}{N} \sum_{i,j} B_{ij} \ln(A_{ij}) \quad (\because \text{eq.}(3.10)) \qquad (3.34)
\end{aligned}
$$

where N is the number of the rows of A (or B). Note g is scalar.

Then $\frac{\partial}{\partial X} g(h(X), T)$ is given by

$$
\frac{\partial}{\partial X} g(h(X), T) = \frac{1}{N} \left(h(X) - T\right). \qquad (3.35)
$$

□

This theorem is the matrix version of Theorem 3.3.

Proof 3.4. We calculate the (l, m) element of the matrix $\frac{\partial}{\partial X} g(h(X), T)$.

$$
\begin{aligned}
\left(\frac{\partial g}{\partial X}\right)_{lm} &= \frac{\partial g}{\partial X_{lm}} \\
&= \sum_{i,j} \left(\frac{\partial g}{\partial h_{ij}} \frac{\partial h_{ij}}{\partial X_{lm}} + \frac{\partial g}{\partial T_{ij}} \frac{\partial T_{ij}}{\partial X_{lm}} \right) \quad (\because \text{chain rule}) \\
&= \sum_{i,j} \left(\frac{\partial g}{\partial h_{ij}} \frac{\partial h_{ij}}{\partial X_{lm}} + \frac{\partial g}{\partial T_{ij}} \cdot 0 \right) \\
&= \sum_{i,j} \frac{\partial g}{\partial h_{ij}} \frac{\partial h_{ij}}{\partial X_{lm}}
\end{aligned}
$$

$$
\begin{aligned}
&= \sum_{i,j} \frac{\partial g}{\partial h_{ij}} \frac{\partial}{\partial X_{lm}} \frac{\exp(X_{ij})}{\sum_k \exp(X_{ik})} \quad (\because \text{eq.}(3.33)) \\
&= \sum_{i,j} \frac{\partial g}{\partial h_{ij}} \left\{ \delta_{il}\delta_{jm} \frac{\exp(X_{ij})}{\sum_k \exp(X_{ik})} - \frac{\exp(X_{ij}) \frac{\partial}{\partial X_{lm}} \sum_k \exp(X_{ik})}{\left(\sum_k \exp(X_{ik})\right)^2} \right\} \\
&= \sum_{i,j} \frac{\partial g}{\partial h_{ij}} \left\{ \delta_{il}\delta_{jm} \frac{\exp(X_{ij})}{\sum_k \exp(X_{ik})} - \frac{\exp(X_{ij})\delta_{il} \exp(X_{lm})}{\left(\sum_k \exp(X_{ik})\right)^2} \right\} \\
&= \sum_{i,j} \frac{\partial}{\partial h_{ij}} \left(-\frac{1}{N} \sum_{p,q} T_{pq} \ln(h_{pq}) \right) \left\{ \delta_{il}\delta_{jm} \frac{\exp(X_{ij})}{\sum_k \exp(X_{ik})} - \frac{\exp(X_{ij})\delta_{li} \exp(X_{lm})}{\left(\sum_k \exp(X_{ik})\right)^2} \right\} \quad (\because \text{eq.}(3.34)) \\
&= \sum_{i,j} \left(-\frac{1}{N} T_{ij} \frac{1}{h_{ij}} \right) \left\{ \delta_{il}\delta_{jm} \frac{\exp(X_{ij})}{\sum_k \exp(X_{ik})} - \frac{\exp(X_{ij})\delta_{li} \exp(X_{lm})}{\left(\sum_k \exp(X_{ik})\right)^2} \right\} \\
&= \sum_{i,j} \left(-\frac{1}{N} T_{ij} \frac{\sum_r \exp(X_{ir})}{\exp(X_{ij})} \right) \left\{ \delta_{il}\delta_{jm} \frac{\exp(X_{ij})}{\sum_k \exp(X_{ik})} - \frac{\exp(X_{ij})\delta_{li} \exp(X_{lm})}{\left(\sum_k \exp(X_{ik})\right)^2} \right\} \quad (\because \text{eq.}(3.33)) \\
&= -\frac{1}{N} \left(T_{lm} - \sum_{i,j} T_{ij} \frac{\delta_{li} \exp(X_{lm})}{\sum_k \exp(X_{ik})} \right) \\
&= -\frac{1}{N} \left(T_{lm} - \sum_{j} T_{lj} \frac{\exp(X_{lm})}{\sum_k \exp(X_{lk})} \right) \\
&= -\frac{1}{N} \left(T_{lm} - \frac{\exp(X_{lm})}{\sum_k \exp(X_{lk})} \right) \quad \left(\because \sum_j T_{lj} = 1 \right) \\
&= -\frac{1}{N} \left(T_{lm} - h_{lm} \right) \quad (\because \text{eq.}(3.33)) \\
&= \frac{1}{N} \left(h_{lm} - T_{lm} \right) \qquad (3.36)
\end{aligned}
$$

Thus we get

$$
\frac{\partial}{\partial X} g(h(X), T) = \frac{1}{N} \left(h(X) - T \right). \qquad (3.37)
$$

■

Theorem 3.5. Let an array X be a raw output of a network, an array T a training data in one-hot representation, h identity function and g mean squared error.

$$
\begin{aligned}
h(A) &= A \quad (\because \text{eq.}(3.7)) && (3.38) \\
g(A, B) &= \frac{1}{2} \sum_k (A_k - B_k)^2 \quad (\because \text{eq.}(3.9)) && (3.39)
\end{aligned}
$$

Then $\frac{\partial}{\partial X} g(h(X), T)$ is given by

$$
\frac{\partial}{\partial X} g(h(X), T) = h(X) - T. \qquad (3.40)
$$

□

The result is the same as eq.(3.29).

Proof 3.5. We calculate the lth element of $\frac{\partial}{\partial X} g(h(X), T)$.

$$\begin{aligned}
\left(\frac{\partial g}{\partial X}\right)_l &= \sum_i \frac{\partial g}{\partial h_i}\frac{\partial h_i}{\partial X_l} \quad (\because \text{eq.}(3.30)) \\
&= \sum_i \frac{\partial g}{\partial h_i}\frac{\partial X_i}{\partial X_l} \quad (\because \text{eq.}(3.38)) \\
&= \frac{\partial g}{\partial h_l} \\
&= \frac{\partial}{\partial h_l}\frac{1}{2}\sum_k (h_k - T_k)^2 \quad (\because \text{eq.}(3.39)) \\
&= \frac{1}{2}\sum_k 2(h_k - T_k)\delta_{lk} \\
&= h_l - T_l
\end{aligned} \tag{3.41}$$

Thus we get

$$\frac{\partial}{\partial X} g(h(X), T) = h(X) - T. \tag{3.42}$$

■

3.7.2 SigmoidLayer class

Theorem 3.6. Let g be the final output of a network, I an input to an instance of `SigmoidLayer` class and Y the corresponding output of the instance of `SigmoidLayer` class. Then the formula below is satisfied.

$$\left(\frac{\partial g}{\partial I}\right)_{lm} = \frac{\partial g}{\partial Y_{lm}} Y_{lm}(1 - Y_{lm}) \tag{3.43}$$

□

Proof 3.6.

$$\begin{aligned}
\left(\frac{\partial g}{\partial I}\right)_{lm} &= \frac{\partial g}{\partial I_{lm}} \\
&= \sum_{i,j} \frac{\partial g}{\partial Y_{ij}}\frac{\partial Y_{ij}}{\partial I_{lm}} \quad (\because \text{eq.}(3.25)) \\
&= \sum_{i,j} \frac{\partial g}{\partial Y_{ij}}\frac{\partial}{\partial I_{lm}}\frac{1}{1 + \exp(-I_{ij})} \quad (\because \text{eq.}(3.5)) \\
&= \sum_{i,j} \frac{\partial g}{\partial Y_{ij}}(-1)\frac{-\exp(-I_{ij})\delta_{il}\delta_{jm}}{(1 + \exp(-I_{ij}))^2} \\
&= \frac{\partial g}{\partial Y_{lm}}\frac{\exp(-I_{lm})}{(1 + \exp(-I_{lm}))^2} \\
&= \frac{\partial g}{\partial Y_{lm}} Y_{lm}\frac{\exp(-I_{lm})}{1 + \exp(-I_{lm})} \quad (\because \text{eq.}(3.5)) \\
&= \frac{\partial g}{\partial Y_{lm}} Y_{lm}(1 - Y_{lm}) \quad (\because \text{eq.}(3.5))
\end{aligned} \tag{3.44}$$

■

3.7.3 ReluLayer class

Theorem 3.7. Let g be the final output of a network, I an input to an instance of `ReluLayer` class and Y the corresponding output of the instance of `ReluLayer` class. Then

$$\left(\frac{\partial g}{\partial I}\right)_{lm} = \frac{\partial g}{\partial Y_{lm}} f_{lm} \tag{3.45}$$

where

$$f_{ij} \equiv \begin{cases} 1 & (I_{ij} > 0) \\ 0 & (I_{ij} \leq 0) \end{cases} . \tag{3.46}$$

□

Proof 3.7.

$$\begin{aligned} \left(\frac{\partial g}{\partial I}\right)_{lm} &= \frac{\partial g}{\partial I_{lm}} \\ &= \sum_{i,j} \frac{\partial g}{\partial Y_{ij}} \frac{\partial Y_{ij}}{\partial I_{lm}} \quad (\because \text{eq.}(3.25)) \\ &= \sum_{i,j} \frac{\partial g}{\partial Y_{ij}} \delta_{il} \delta_{jm} f_{ij} \quad (\because \text{eq.}(3.6)) \\ &= \frac{\partial g}{\partial Y_{lm}} f_{lm} \end{aligned} \tag{3.47}$$

■

3.7.4 AffineLayer class

Theorem 3.8. Let g be the final output of a network, I an input to an instance of `AffineLayer` class, W the weights of the layer, $\boldsymbol{b}$ the biases of the layer and Y the corresponding output of the layer. Then the formulae below are satisfied.

$$\frac{\partial g}{\partial W} = {}^t I \frac{\partial g}{\partial Y} \tag{3.48}$$

$$\frac{\partial g}{\partial I} = \frac{\partial g}{\partial Y} {}^t W \tag{3.49}$$

$$\left(\frac{\partial g}{\partial \boldsymbol{b}}\right)_l = \sum_i \frac{\partial g}{\partial Y_{il}} \tag{3.50}$$

□

Proof 3.8. eq.(3.48) is proved as follows.

$$\begin{aligned} \left(\frac{\partial g}{\partial W}\right)_{lm} &= \sum_{i,j} \frac{\partial g}{\partial Y_{ij}} \frac{\partial Y_{ij}}{\partial W_{lm}} \quad (\because \text{eq.}(3.25)) \\ &= \sum_{i,j} \frac{\partial g}{\partial Y_{ij}} \frac{\partial}{\partial W_{lm}} (IW + \boldsymbol{b})_{ij} \quad (\because \text{eq.}(3.12)) \\ &= \sum_{i,j} \frac{\partial g}{\partial Y_{ij}} \frac{\partial}{\partial W_{lm}} (IW)_{ij} \end{aligned}$$

$$
\begin{aligned}
&= \sum_{i,j} \frac{\partial g}{\partial Y_{ij}} \frac{\partial}{\partial W_{lm}} \sum_k I_{ik} W_{kj} \\
&= \sum_{i,j} \frac{\partial g}{\partial Y_{ij}} \delta_{jm} I_{il} \\
&= \sum_i \frac{\partial g}{\partial Y_{im}} I_{il} \\
&= \left({}^t I \frac{\partial g}{\partial Y} \right)_{lm}
\end{aligned} \tag{3.51}
$$

eq.(3.49) is proved as follows.

$$
\begin{aligned}
\left(\frac{\partial g}{\partial I} \right)_{lm} &= \sum_{i,j} \frac{\partial g}{\partial Y_{ij}} \frac{\partial Y_{ij}}{\partial I_{lm}} \quad (\because \text{eq.}(3.25)) \\
&= \sum_{i,j} \frac{\partial g}{\partial Y_{ij}} \frac{\partial}{\partial I_{lm}} (IW + \boldsymbol{b})_{ij} \quad (\because \text{eq.}(3.12)) \\
&= \sum_{i,j} \frac{\partial g}{\partial Y_{ij}} \frac{\partial}{\partial I_{lm}} (IW)_{ij} \\
&= \sum_{i,j} \frac{\partial g}{\partial Y_{ij}} \frac{\partial}{\partial I_{lm}} \sum_k I_{ik} W_{kj} \\
&= \sum_{i,j} \frac{\partial g}{\partial Y_{ij}} \delta_{il} W_{mj} \\
&= \sum_j \frac{\partial g}{\partial Y_{lj}} W_{mj} \\
&= \left(\frac{\partial g}{\partial Y} {}^t W \right)_{lm}
\end{aligned} \tag{3.52}
$$

eq.(3.50) is proved as follows.

$$
\begin{aligned}
\left(\frac{\partial g}{\partial \boldsymbol{b}} \right)_l &= \sum_{i,j} \frac{\partial g}{\partial Y_{ij}} \frac{\partial Y_{ij}}{\partial \boldsymbol{b}_l} \quad (\because \text{eq.}(3.25)) \\
&= \sum_{i,j} \frac{\partial g}{\partial Y_{ij}} \frac{\partial}{\partial \boldsymbol{b}_l} (IW + \boldsymbol{b})_{ij} \quad (\because \text{eq.}(3.12)) \\
&= \sum_{i,j} \frac{\partial g}{\partial Y_{ij}} \frac{\partial}{\partial \boldsymbol{b}_l} \boldsymbol{b}_j \quad (\because \text{eq.}(3.13)) \\
&= \sum_i \frac{\partial g}{\partial Y_{il}}
\end{aligned} \tag{3.53}
$$

■

4 Hand-Written Digit Recognition

Finally let's implement a neural network to do hand-written digit recognition.

4.1 MNIST Dataset

MNIST(Modified National Institute of Standards and Technology) is a dataset of hand-written digits. This is the most famous dataset in the field of machine learning and often seen in theses as a dataset for testing. MNIST consists of images of digits $0, 1, 2, ..., 9$. 60000 images are supplied for training and 10000 images for testing. The dimensions of each image are 28×28 and it has only one channel (i.e. a black-and-white image). Each pixel has a value ranged over $[0..255]$ and the label like $7, 2$ or 1 which indicates the correct answer is also supplied for each image.

Figure 4.1: Sample images from MNIST dataset. Cited from [8].

See `read_mnist/README.md` for the detail of the dataset structure.

4.2 Implementations

We implement a network whose structure is shown in Figure 3.6. We set the number of the nodes in the input layer to $28 \times 28 = 784$ and that of the output layer to 10 since there are 10 kinds of digits. We use the classes whose structure is shown in Figure 4.2 and handle all the layers though pointers of the type `Layer *` stored in `layer_` array except the instance of `SoftmaxCrossEntropyLayer` which is exceptionally pointed by `last_layer_` of the type `LastLayer *`.

4.3 Backpropagation vs Central Difference

To see how backpropagation is faster than central difference, we did a simple speed test. The parameters used are shown in Table 4.2. The results are shown in Table 4.1 and Figure 4.3. As Table 4.1 indicates, backpropagation was surprisingly but expectedly about 1850 times faster than central difference. And Figure 4.3 tells the results given by the two methods correspond exactly to each other, and tells overfitting (Section 1.2) didn't occur. It should also be noted the accuracy reached around good 85 (%) even though only a single hidden layer with only 10 nodes were used.

method	elapsed time (sec)
backpropagation	22.299
central difference	41171.548

Table 4.1: The result of the speed test. Backpropagation is much faster.

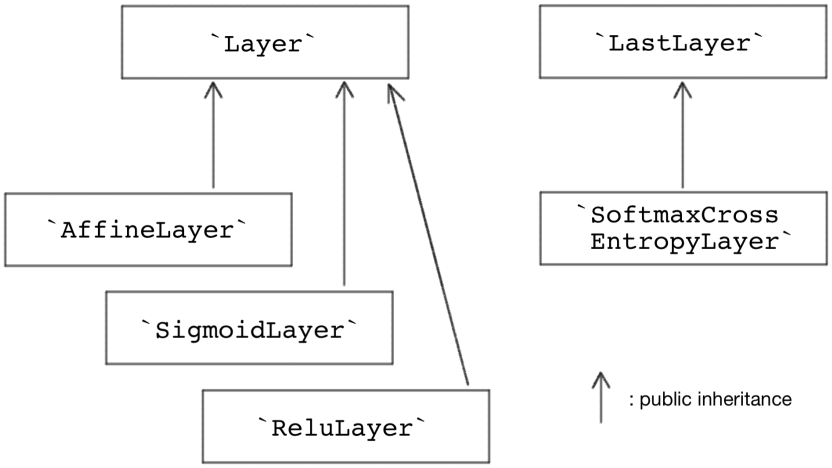


Figure 4.2: The structure of the classes.

# of hidden layers	1
# of nodes in each hidden layer	10
# of images in a minibatch	100
# of epochs calculated	16
dx (used only by central difference)	10^{-2}
learning rate	10^{-1}

Table 4.2: The parameters used for the speed test of backpropagation and central difference.

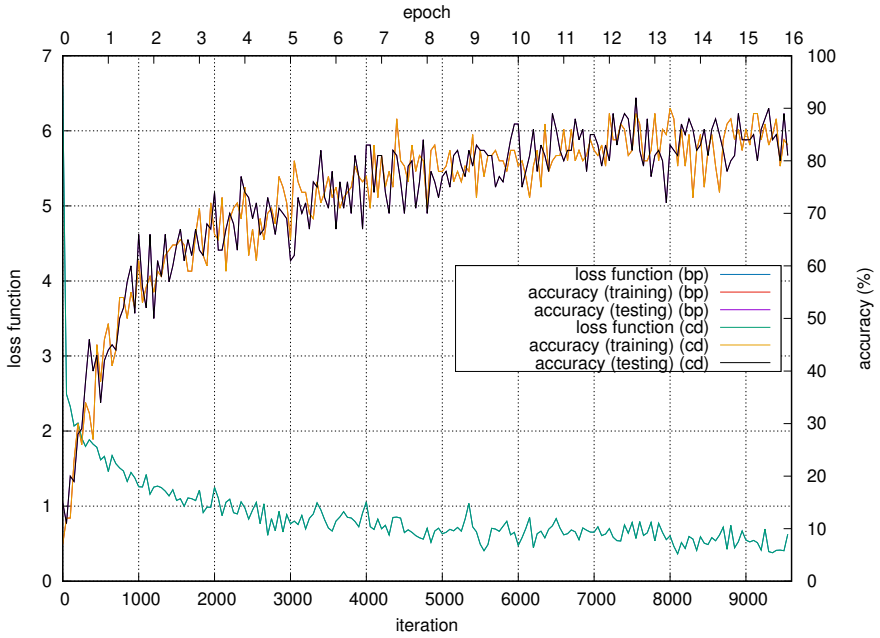


Figure 4.3: The result of the speed test. The two methods give the same result.

4.4 Backpropagation with More Hidden Nodes

5 References

[1] "Optimization problem" (https://en.wikipedia.org/wiki/Optimization_problem)

[2] "Machine learning" (https://en.wikipedia.org/wiki/Machine_learning)

[3] "What is the difference between machine learning and optimization?" (https://www.quora.com/What-is-the-difference-between-machine-learning-and-optimization)

[4] "Boolean function" (https://en.wikipedia.org/wiki/Boolean_function)

[5] "Logic gate" (https://en.wikipedia.org/wiki/Logic_gate)

[6] "Artificial neural network" (https://en.wikipedia.org/wiki/Artificial_neural_network#Organization)

[7] "Gradient" (https://en.wikipedia.org/wiki/Gradient)

[8] "MNIST dataset" (https://en.wikipedia.org/wiki/MNIST_database)