C++ Assn3
Neural Network for MNIST

Hyeon Jeon

1 Introduction

In this assignment, you will implement a basic neural network consists of 3 layers, which classifies handwritten digits provided by the MNIST dataset. By the assignment, you will understand the following concepts:

- Class and inheritance in cpp
- How neural network & backpropagation works

2 Background

2.1 MNIST dataset

MNIST(Modified National Institute of Standards and Technology database) dataset is a large database of handwritten digits, which is widely used for training and testing image processing systems based on deep learning / machine learning models. Since 1998, lots of computer scientists tried to implement AI, which can classify the images, and recently the researchers from the University of Virginia announced the model, which achieved 99.82% accuracy.

MNIST dataset

MNIST contains 60,000 training images and 10,000 test images, and they're provided in `mnist` directory. Also, accessing and reading files is already implemented; hence you only need to focus on implementing a neural network that receives the data from MNIST as input.

2.2 Brief History

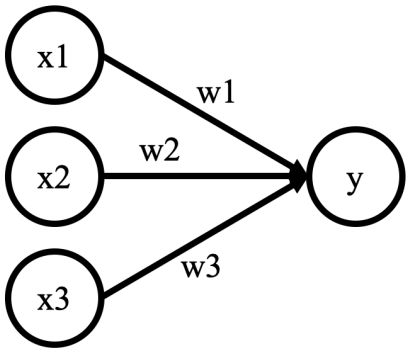
In the early stage of AI(Artificial Intelligence), it was mostly rule-based. Researchers extracted the patterns from data and tried to encode them as rules heuristically. However, the strategy was quite not successful, as it was hard to obtain every underlying pattern from data.

On the other hand, in the 1970-80s, several researchers, including Geoffrey Hinton, David Rumelhart, Yoshua Bengio, tried another approach: Artificial neural network. Their approach mainly aimed to mimic the neuron & synapse structure of the human brain. However, due to its high computation requirement, the approach was extremely inefficient and was discarded for decades; we now call this era as *AI winter*.

However, as the computation power had come a long way, the time and resource usage for Artificial Neural Network decreased significantly, and it showed overwhelming performance. Moreover, as now researchers can collect unlimited data from the internet, it got the opportunity to be adopted in various domains. Therefore, nowadays, AI became one of the most important modern technologies, and learning AI & neural networks is now an essential course for researchers in the engineering or science field.

2.3 Perceptron

Perceptron is an algorithm offered by Frank Rosenblatt in 1957 and now used as a fundamental element in many complex neural networks. The structure is quite simple: it combines several inputs and generates single output.



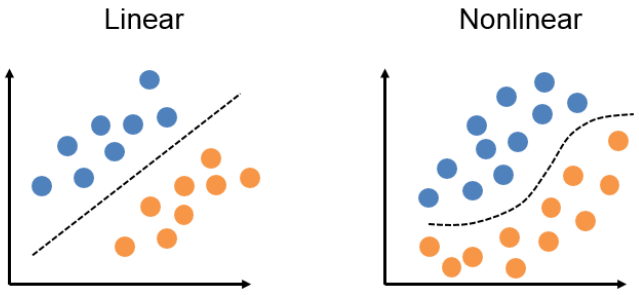
Single perceptron

The y value is determined like this:

$$y = 1 \text{ if } x_1 w_1 + x_2 y_2 + x_3 y_3 > \theta$$
$$y = 0 \text{ if } x_1 w_1 + x_2 y_2 + x_3 y_3 < \theta$$

where θ is a threshold. However, a single-layer perceptron can only classify the space linearly; it cannot be applied to the complex non-linear situation. To solve the problem, researchers constructed a neural network as a sophisticated multi-layer perceptron.

In this assignment, you'll implement a multi-layer perceptron. For the detailed explanation for our neural network, refer next section.

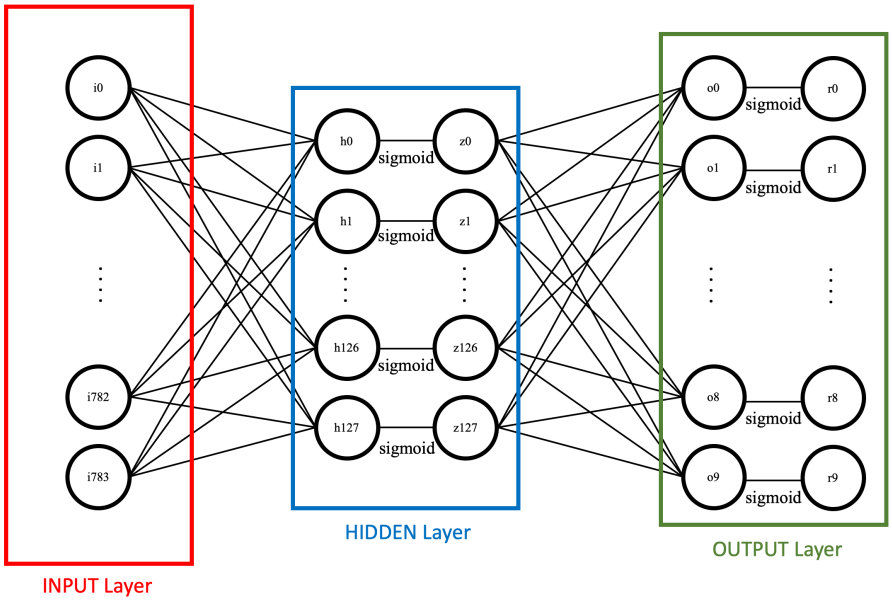


Linear vs. Nonlinear perceptron
(Reference: https://jtsulliv.github.io/perceptron/)

3 Explanation

3.1 Neural network

As I mentioned above, our neural network will consist of 3 layers: input, hidden, and output, which are *fully connected*. The structure overview and specification for each part are as follows.



Neural network structure overview

As the MNIST image size is $28 \times 28 = 784$, There are total 784 nodes in the input layer. The hidden layer initially has 128 nodes, but you can modify it freely. Last, the output layer has ten nodes, where each represents the corresponding digit (0-9).

3.2 Activation function

You might notice that both the hidden layer and the output layer have a single edge named *Sigmoid*. It is an *Activation function* for our neural network. So what is activation function? To

understand the activation function, you must recall that the single-layer perceptron generates linear classification, and a multi-layer approach had arrived to solve the problem. However, if we overlap multiple linear perceptrons successively, it still produces linear classification. A simple example might help you: overlapping $y = ax$ and $z = by$ just generates $z = b \cdot ax$, which is still linear.

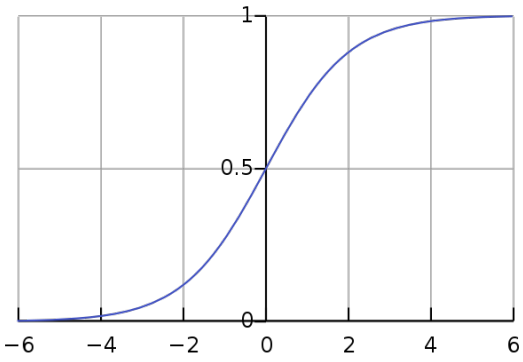
Therefore, a non-linear function should be located between linear perceptrons, and the activation function serves the role. There exist various modern non-linear activation functions: ReLU, Leaky ReLU, tanh, Softmax, Swish....

Activation function has one more important property: it controls the power of the signal. Let's remind that the basic perceptron generates only two kinds of signal: 0 to 1 (transfer or not). However, researchers found that transferring the signal value is more effective in solving complex problems. But to transfer continuous value, the output of the perceptron should be post-processed to maintain consistency. The activation function helps the neural network to achieve the goal. For instance, Sigmoid normalizes the input value to a value between 0 and 1.

Sigmoid

Sigmoid, or logistic regression, which you'll implement, is quite an old strategy. The function and its graph are like this:

$$\text{Sigmoid}(x) = \frac{e^x}{1 + e^x}$$



As represented in the graph, Sigmoid sends a large positive number to 1 and the opposite to -1. Sigmoid has some critical problems: exponential function inside it requires high computation, and sometimes immensely slows down the training. Nevertheless, it is easy to implement and also fits our problem. You'll notice the benefit of Sigmoid in **backpropagation section**.

3.3 Training

The training step of typical neural networks, including our network, follows the process:

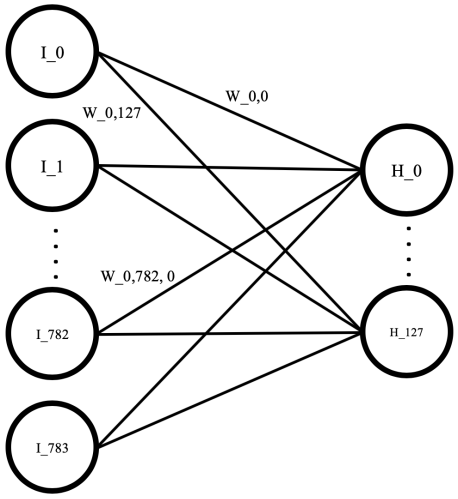
1. Get current input
2. Perform feed-forward computation through the network

3. Calculate Loss
4. Backpropagate the loss and update weights
5. Go back to step 1 with next input

Let's see the detail with our neural network.

Feed-Forward

Our network starts with the connection between the Input layer and the Hidden layer.



Let's denote each input value(node) as I_i and hidden layer input value as H_i. Then the overall computation across Input and Hidden layer is as follows:

$$H_i = W_{0,i}I_0 + W_{1,i}I_1 + \cdots + W_{782,i}I_{782} + W_{783,i}I_{783} + b_i$$

where i: 0 to 127 (128 hidden nodes), $W_{a,b}$ corresponds to the edge connecting I_a and H_b, and b_i denotes the bias.

Hidden layer also contains Sigmoid. Let's denote Sigmoid as ϕ and it's result as Z_i. Then we can just say $\phi(H_i) = Z_i$ where i: 0 to 127.

Now only the connection between Hidden / Output layer and the final Sigmoid are left. These computations are like this:

$$\begin{aligned} O_i &= W'_{0,i}Z_0 + W'_{1,i}Z_1 + \cdots + W'_{126,i}Z_{126} + W'_{127,i}Z_{127} + b'_i \\ R_i &= \phi(O_i) \qquad\qquad i : 0 \text{ to } 9 \end{aligned}$$

where O_i: Output layer input value, and R_i: Final result value.

Loss function

As final result R_is are the output of Sigmoid, their value is in between zero and one. Next step is to calculate loss. In this assignment, you'll implement *MSE(Mean Squared Error)*:

$$MSE = \frac{1}{2}\Sigma_{i=0}^{9}(\hat{R}_i - R_i)^2$$
$$\hat{R}_i = 0 \text{ if } label == i \text{ else } 0$$

as loss function, where $\hat{R}_i$ denotes the real result. The value of real result vector $\hat{R}$ is defined as above. For example, if the input image represents digit 7, $\hat{R}$ will be $(0, 0, 0, 0, 0, 0, 0, 1, 0, 0)$.

Therefore, if our neural network assigns a high value to the index corresponds to the answer and low value to the others, the loss will be small. Therefore, we interpret the final result value vector of R as follows: answer $= \arg\max R_i$.

Backpropagation

Now we learned how our neural network generates the answer and the way to calculate the loss. Then, how can we *train* our neural network? Backpropagation, which was first suggested by Rumelhart et al. in their paper published in *Nature*, provides a polished way to achieve the goal.

The main concept of backpropagation is to send back the generated loss(or error) toward the input and underlying weights. While sending back the error, the gradient for each weight will be calculated successively, and the weights will be updated due to their gradients. So how can we calculate gradient *successively*? The theoretical background for this step is quite simple: *chain rule*.

Maybe you had learned partial derivative and chain rule in the calculus class. For who did not or who forgot the concepts, I'll briefly explain them. The former is quite a straightforward concept: a partial derivative of a function of several variables is its derivative concerning one of those variables, with the others held constant. For example, partial derivative $\partial f/\partial x$ of the function $f(x, y) = ax^2 + bxy + cy^2$ will be:

$$\frac{\partial f}{\partial x} = 2ax + by.$$

This is the only thing you need to know about partial derivative.

Then how about chain rule? Astonishingly, you had noticed the basic concept of the chain rule in high school. Recall the famous derivative rule $(f(g(x))' = f'(g(x)) \cdot g'(x)$, which can be written as

$$\frac{df \circ g}{dx} = \frac{df}{dg} \cdot \frac{dg}{dx}$$

by the Leibniz's notational system. This directly shows the main idea of the chain rule: the derivative can be calculated through the consecutive sequence of inner derivatives. A simple example will help you. Let's think about the composite function $f(g(h(i(x))))$. The derivative of the func-

tion can be calculated through the sequence:

$$\begin{aligned}(f(g(h(i(x)))))' &= f'(g(h(i(x)))) \cdot (g(h(i(x))))' \\ &= f'(g(h(i(x)))) \cdot g'(h(i(x))) \cdot (h(i(x)))' \\ &= f'(g(h(i(x)))) \cdot g'(h(i(x))) \cdot h'(i(x)) \cdot i'(x)\end{aligned}$$

which can be expressed as

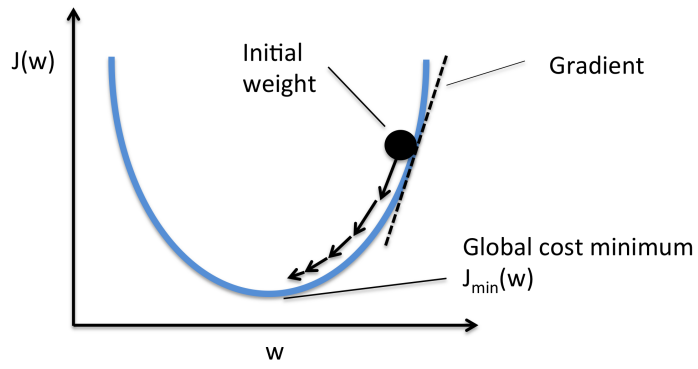
$$\frac{df}{dx} = \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{di} \cdot \frac{di}{dx},$$

which is the standard form of the chain rule. Of course, chain rule can be applied to partial derivative too.

Then how can we apply these concepts to update weights during backpropagation? The main purpose of the backpropagation is to reduce the loss. And our error, MSE, can be expressed as the function of variables W, b, I, where each denotes the set of weight, bias, and input values. Therefore, it is evident that we can calculate the partial derivative of MSE due to each weight and bias, and the value can be evaluated using the chain rule. After calculating the gradient, the weight can be updated toward the direction, which reduces the loss. The equation below explains the work.

$$w = w - \eta \frac{\partial MSE}{\partial w}$$

In the equation, w denotes a certain weight, and η is the learning rate. If the learning rate is low, the training's precision will become high, but it will take much time to arrive at the point. In contrast, a high learning rate will ensure both fast training and low precision.



Weight update due to its gradient

Now its time to follow the backpropagation step. Let's first recall the entire feed-forward process.

$$
\begin{aligned}
H_i &= W_{0,i}I_0 + W_{1,i}I_1 + \cdots + W_{782,i}I_{782} + W_{783,i}I_{783} + b_i \\
Z_i &= \phi(H_i) && i : 0 \text{ to } 127 \\
O_j &= W'_{0,j}Z_0 + W'_{1,j}Z_1 + \cdots + W'_{126,j}Z_{126} + W'_{127,j}Z_{127} + b'_j \\
R_j &= \phi(O_j) && j : 0 \text{ to } 9 \\
MSE &= \frac{1}{2}\Sigma_{k=0}^{9}(\hat{R}_k - R_k)^2 && \hat{R}_k = 0 \text{ if } label == k \text{ else } 0
\end{aligned}
$$

The parameters that should be updated are the *weights* and *bias* of the neural network. Let's first investigate $W'_{i,j}$. It is obvious that

$$
\frac{\partial MSE}{\partial W'_{i,j}} = \frac{\partial MSE}{\partial R_j}\frac{\partial R_j}{\partial O_j}\frac{\partial O_j}{\partial W'_{i,j}},
$$

and each term of the right-hand side is as follows:

$$
\begin{aligned}
\frac{\partial MSE}{\partial R_j} &= \frac{1}{2}\frac{\partial}{\partial R_j}(\hat{R}_j - R_j)^2 = (\hat{R}_j - R_j) \\
\frac{\partial R_j}{\partial O_j} &= \frac{\partial \phi(O_j)}{\partial O_j} \\
\frac{\partial O_j}{\partial W'_{i,j}} &= Z_i
\end{aligned}
$$

Now you might ask; "How can we calculate $\frac{\partial \phi(O_j)}{\partial O_j}$" ? Recall that

$$
\phi(x) = \text{Sigmoid}(x) = \frac{e^x}{1+e^x} = \frac{1}{1+e^{-x}}.
$$

We can easily calculate the derivative using simple derivation rule:

$$
\begin{aligned}
\phi'(x) &= ((1+e^{-x})^{-1})' \\
&= (1+e^{-x})^{-2} \cdot (1+e^{-x})' \\
&= (1+e^{-x})^{-2} \cdot (-e^{-x}) \\
&= \frac{1}{1+e^{-x}} \cdot \frac{-e^{-x}}{1+e^{-x}} \\
&= (1-\phi(x))\phi(x)
\end{aligned}
$$

Therefore, we can rewrite the second derivative $\frac{\partial R_j}{\partial O_j}$.

$$
\frac{\partial R_j}{\partial O_j} = \frac{\partial \phi(O_j)}{\partial O_j} = (1-\phi(O_j))\phi(O_j) = (1-R_j)R_j
$$

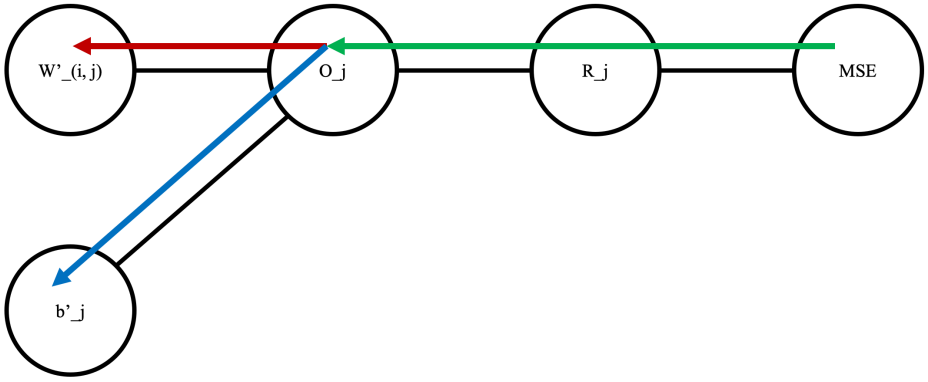
It is easy to notice that $\frac{\partial MSE}{\partial W_{i,j}}$ can be calculated only by using $\hat{R}_j, R_j, Z_i$, which are the values that can be evaluated during the feed-forward step.

Then How about the bias b'_j? We can simply write the gradient like:

$$\frac{\partial MSE}{\partial b'_j} = \frac{\partial MSE}{\partial R_j}\frac{\partial R_j}{\partial O_j}\frac{\partial O_j}{\partial b'_j},$$

Now you might found the issue; we already calculated $\frac{\partial MSE}{\partial R_j}$ and $\frac{\partial R_j}{\partial O_j}$ while dealing with $W_{i,j}$. Therefore the only thing we need to consider is $\frac{\partial O_j}{\partial b'_j}$, which is 1.

This fact is one of the most critical points during backpropagation. We can reuse the sequence of partial gradients. The above situation can be illustrated like this.



As we already calculated the gradients corresponding to a green arrow while going toward a red arrow, we don't need to do it again while going toward a blue arrow. This technique reduces computation a lot and therefore accelerates training.

4 Assignments

4.1 Calculation

You may be understood how our neural network works. However, I didn't explain everything, so you must first fill up the missing links.

Solve the following problems. It is highly recommended to write the entire process with LaTeX.

1. In section **3**, we only calculated the gradients for the second layer. Calculate the gradients for the first layer.
   a) Calculate $\frac{\partial MSE}{\partial W_{i,j}}$.
   b) Calculate $\frac{\partial MSE}{\partial b_j}$.
2. Recall that our neural network updates the weight by applying

   $$w = w - \eta\frac{\partial MSE}{\partial w}.$$

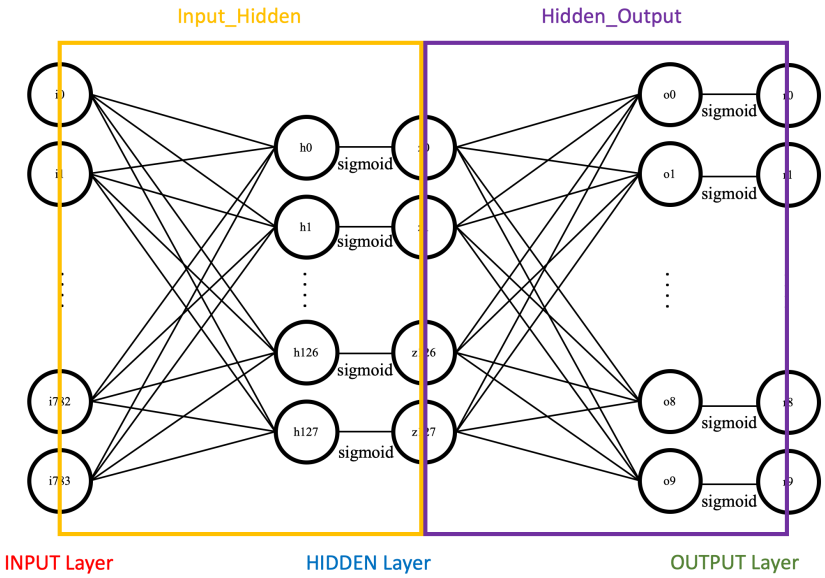
   Prove that this updating strategy ensures the decrease of MSE (*hint: focus on* ΔMSE).

4.2 Implementation

The template code is available at the link. The training and testing code are already implemented, so you only need to focus on implementing the feed-forward and backpropagation step.

The functions that you need to implement is like this:

- `perceptron.cpp`
  - `Perceptron(int num)`
    The constructor of a single perceptron, which gets `num` inputs. Allocate the initial value to the `num` weights correspond to each input, and a single bias. It is recommended to assign random value to weights / bias using `rand_0to1` function in `helper.cpp`.
  - `forward(double* input)`
    Feed-forward step for a single perceptron.
  - `backprop(double delta, double *current_input)`
    Backpropagation step for a single perceptron. `delta` denotes the former gradients which are propagated from the fore, and `current_input` literally denotes the current input vector for the layer, which contains the perceptron.
- `sigmoid.cpp`
  - `forward(double input)`
  - `backprop()`
- `bridge.cpp`
  This class represents the bridge between a layer to the other layer. It is consists of the set of perceptrons and sigmoids, according to the input / output structure explained in section **3.1**.
  - `Bridge(int input_num, int output_num)`
    Gets input / output number as arguments, and allocates perceptrons and sigmoids due to the values.
  - `forward(double* input)`
- `hidden_output.cpp`
  inherites `Bridge` class, and represents the bridge from the input layer to the hidden layer.
  - `backprop(double* delta)`
- `input_hidden.cpp`
  inherites `Bridge` class, and represents the bridge from the input layer to the hidden layer.
  - `backprop(double* delta)`
    `Input_Hidden :: backprop()` should get the result of `Hidden_Output :: backprop()` as a argument.
- `train.cpp`
  - `loss_function(double* delta)`
    `delta` denotes the array consists of $\hat{R}_i - R_i$, where $i = 0$ to 9.



The area of `Hidden_Output` and `Input_Hidden` are like this.

After implementation, you can generate executable `program` by `make run` command. If the program output is like this, it verifies your implementation.

```
❯ ./program
============================
Neural Network model for MNIST
EPOCH: 100
Learning rate: 0.01
============================
TRAINING START!!
Total dataset size: 60000
Sample ~3000: Average loss is 0.402518
Sample ~6000: Average loss is 0.258591
Sample ~9000: Average loss is 0.217464
Sample ~12000: Average loss is 0.195215
Sample ~15000: Average loss is 0.208591
Sample ~18000: Average loss is 0.178943
Sample ~21000: Average loss is 0.155885
Sample ~24000: Average loss is 0.155108
Sample ~27000: Average loss is 0.151913
Sample ~30000: Average loss is 0.147741
Sample ~33000: Average loss is 0.154779
Sample ~36000: Average loss is 0.123361
Sample ~39000: Average loss is 0.128606
Sample ~42000: Average loss is 0.131559
Sample ~45000: Average loss is 0.126316
Sample ~48000: Average loss is 0.1355
Sample ~51000: Average loss is 0.127829
Sample ~54000: Average loss is 0.113296
Sample ~57000: Average loss is 0.103901
Sample ~60000: Average loss is 0.080278
============================
TESTING...
============================
Test RESULT
Test set size: 10000
Correct: 8877
Wrong: 1123
Accuracy: 0.8877
```

4.3 Tips

- Use the constants in `constants.h`. For fast test & verification, I recommend you to change the `EPOCH`
- Remember that you only need to implement about 50 lines.