Artificial Neural Network Homework 1: MNIST Classification Based on MLP

Jinyi Hu
Department of Computer Science and Technology, Tsinghua University
hujy369@163.com

Abstract

We implement the multi-layer perception neural network based on the code provided by TA. The model is tested on the dataset of MNIST, a large database of handwritten digits. The result shows good performance of our model.

1. Model

1.1. Model description

Basically, we implement a very simple neural network, known as multilayer perceptron(MLP). MLP is a fully connected neural network, consisting of at least three layers of nodes: an input layer, a hidden layer and an output layer. And except for the input nodes, every node in the layer is a neuron with a nonlinear activation function. To train the MLP network, we utilize supervised learning technique, computing the loss between the output and the given label and then backpropagating to adjust the weight.

1.2. Activation function

Inspired by the biological machanism, neurons use a nonlinear activation function to distinguish data which cannot be linearly separated. In this experiment, we utilize two kinds of nonlinear activation function: sigmoids and ReLU, which are described by, respectively,

$$Sigmoid : y(x_i) = \frac{1}{1+e^{-x_i}}$$

$$ReLU : y(x_i) = max(x_i, 0).$$

Here the x_i is the input of the neuron and y is the output of the neuron. In these cases, the derivative function of these activation functions can be computed and represented simply as

$$Sigmoid : y'(x_i) = y(x_i)(1-y(x_i))$$

$$ReLU : y'(x_i) = \begin{cases} 0, & x_i < 0 \\ 1, & x_i \geq 0 \end{cases}$$

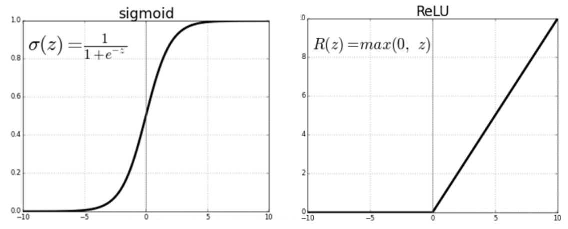

Figure 1. The graphs of two kinds of activation functions

Besides, there are other activation functions such as $tanh$, which is described by

$$y(x_i) = tanh(x_i).$$

We will not discuss these activation functions in this experiment.

1.3. Loss function

In this project, we utilize two kind of loss function: euclidean loss and softmax cross-entropy loss, and compare the performance between them. In this classification problem, the output is a distribution of 10 classes. We will compute distance between the output distribution and the given label, which will be transformed to one-hot vector. For the Euclidean loss, we directly compute the euclidean distance between the output and the one-hot label vector, which can be described by

$$E = \frac{1}{2}\sum_{i=1}^{k}(t_k - y_k)^2.$$

For the Softmax cross-entropy loss, it will first softmax the output of the final layer and the compute the cross-entropy between the softmaxed output and the one-hot label vector, which can be described by

$$E = \sum_{i=1}^{k} t_i \ln h_i$$

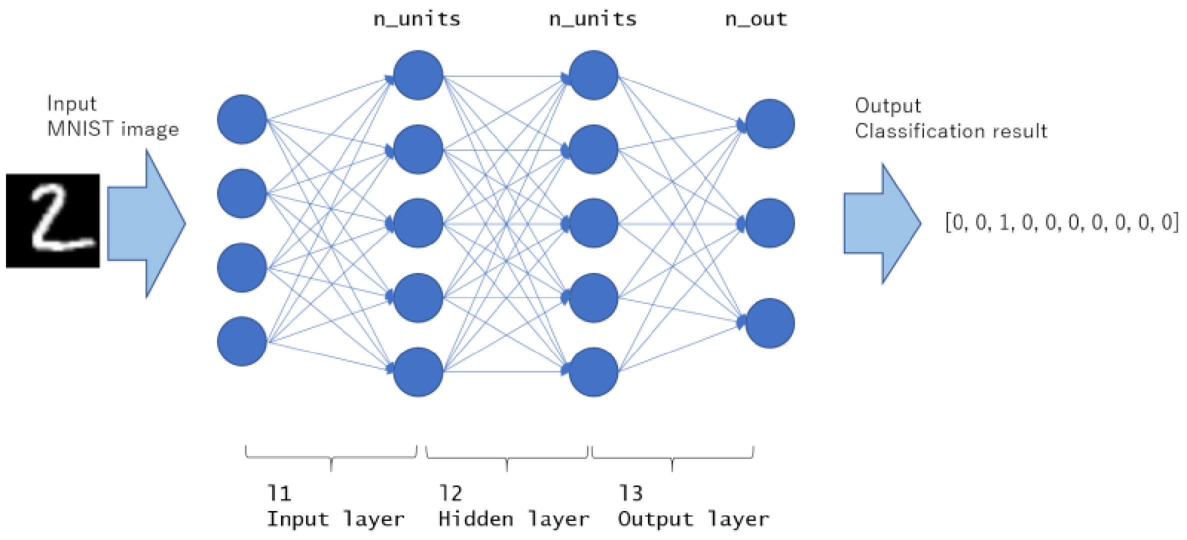

Figure 2. MLP network architecture

where

$$h_i = \frac{\exp x_i}{\sum_{j=1}^{k} \exp x_j}$$

Cross-entropy can reflect the distance between two distribution. This loss function is commonly used in classification problems.

For the derivative of the loss function, we omit the deduction process and give the final result. The derivative of Euclidean loss is relatively trival, which can be described by

$$\frac{\partial E^{(n)}}{\partial x^{(n)}} = x^{(n)} - t^{(n)}$$

While the deduction process of the derivative of the softmax cross-entropy loss is a little tedious, the result is quite simple. It's in the following formula:

$$\frac{\partial E^{(n)}}{\partial x^{(n)}} = Softmax(x^{(n)}) - t^{(n)}$$

$t^(n)$ in the two fomula above means the one-hot label vector for the n^{th} sample and $x^(n)$ means the output vector for the n^{th} sample.

Two points here need to be emphasized. Firstly, the format above looks like computing the derivative of a vector. Actually, it's for the convienence of programming implementation. In computer, all the derivatives are computed in a vectorized format. Its format may not mathematically precise enough.

Secondly, when training the model, we often use a mini-batch of data to finish one adjustment. So we should compute the average of loss in a batch. Also, when computing the gradient, the final result derived from the formula above should be divided by batch size. So actually, the gradient of loss should be in the format as following:

$$Euclidean\ loss: \frac{\partial E^{(n)}}{\partial x^{(n)}} = (x^{(n)} - t^{(n)})/batch\_size$$

$$cross\ entropy: \frac{\partial E^{(n)}}{\partial x^{(n)}} = (Softmax(x^{(n)}) - t^{(n)})/batch\_size$$

1.4. Optimization

To accerlerate the convergence, some optimization methods are used. Commonly, we use Stochastic Gradient Descent(SGD) and Momentum machanism. In this way, the adjustment process can be formated like:

$$\Delta w_{ji}(new) = \alpha \Delta w_{ji}(old) - \eta \delta_i(n) y_j(n)$$

According to the formula, there seems to be some tiny mistakes in the code provided by TA.

2. Experimental design

We design 8 different experiments in total and compare the performance ammong model with different layers, different loss function and activation function. The structure of these networks are similar. Table 1 and Table 5 shows the details of model.

We fixed than total iterations as 60,000, which means $max\ epoch \times batch\ size \equiv 60.000$

2.1. Dataset

We train out model on the dataset of MNIST, a famous dataset of handwritten digits. In this dataset, there 60,000

Layer	Type	Input Size	Output Size
1	Linear	784	256
2	ReLU(sigmoid)	256	256
3	Linear	256	10
4	ReLU(sigmoid)	10	10

Table 1. Details of the one hidden layer model

training samples in total and 10,000 test samples. After training, the accuracy rate of recognizing the handwritten digits can reach above 98%

2.2. Device

All the experiments are done on my own MacBool Pro, with 6 Intel Core i7 2.6 GHz CPUs.

2.3. Experiment 1: one hidden layer with ReLU and euclidean loss

In this case, we add one hidden layer with ReLU activation function and utilize euclidean loss function.The details of hyperparameters are shown in Table 2.

Type	Value
batch size	30
learning rate	0.01
weight decay	1e-4
momentum	0.9
epoch	30

Table 2. Details of the hyperparameters in Experiment 1

2.4. Experiment 2: one hidden layer with sigmoid and euclidean loss

In this case, we add one hidden layer with sigmoid activation function and utilize euclidean loss function. The details of hyperparameters are shown in Table 3.

Type	Value
batch size	200
learning rate	0.01
weight decay	1e-4
momentum	0.95
epoch	200

Table 3. Details of the hyperparameters in Experiment 2

2.5. Experiment 3: one hidden layer with ReLU and softmax loss

In this case, we add one hidden layer with ReLU activation function and utilize softmax loss function. The details of hyperparameters are shown in Table 4.

Type	Value
batch size	100
learning rate	0.02
weight decay	1e-4
momentum	0.95
epoch	100

Table 4. Details of the hyperparameters in Experiment 3

Layer	Type	Input Size	Output Size
1	Linear	784	256
2	ReLU(sigmoid)	256	256
3	Linear	256	128
4	ReLU(sigmoid)	128	128
5	Linear	128	10
6	ReLU(sigmoid)	10	10

Table 5. Details of two hidden layers model

2.6. Experiment 4: one hidden layer with sigmoid and softmax loss

In this case, we add one hidden layer with sigmoid activation function and utilize softmax loss function. The details of hyperparameters are shown in Table 6.

Type	Value
batch size	200
learning rate	0.02
weight decay	1e-4
momentum	0.95
epoch	200

Table 6. Details of the hyperparameters in Experiment 4

2.7. Experiment 5: two hidden layers with ReLU and euclidean loss

In this case, we add one hidden layer with sigmoid activation function and utilize softmax loss function. The details of hyperparameters are shown in Table 7.

Type	Value
batch size	100
learning rate	0.01
weight decay	1e-4
momentum	0.95
epoch	100

Table 7. Details of the hyperparameters in Experiment 5

Metric	ReLU+euclidean	sigmoid+euclidean	sigmoid+softmax
Train Loss	0.012	0.0482	1.5186
Train Acc.	99.47%	95.16%	95.34%
Test Loss	0.02269	0.04873	1.5194
Test Acc.	98.17%	94.91%	95.19%
Train Time	98s	254s	285s

Table 8. Evaluation data of **one** hidden layer model

Metric	ReLU+euclidean	sigmoid+euclidean	sigmoid+softmax
Train Loss	0.0052	0.0343	1.5053
Train Acc.	99.73%	96.79%	96.661%
Test Loss	0.01437	0.03595	1.50851
Test Acc.	98.40%	96.61%	96.294%
Train Time	136s	368s	491s

Table 9. Evaluation data of **two** hidden layers model

2.8. Experiment 6: two hidden layers with sigmoid and euclidean loss

In this case, we add one hidden layer with sigmoid activation function and utilize softmax loss function. The details of hyperparameters are shown in Table 10.

Type	Value
batch size	300
learning rate	0.03
weight decay	1e-4
momentum	0.95
epoch	300

Table 10. Details of the hyperparameters in Experiment 6

2.9. Experiment 7: two hidden layers with ReLU and softmax loss

In this case, we add one hidden layer with sigmoid activation function and utilize softmax loss function. The details of hyperparameters are shown in Table 11.

Type	Value
batch size	200
learning rate	0.02
weight decay	1e-4
momentum	0.95
epoch	200

Table 11. Details of the hyperparameters in Experiment 7

2.10. Experiment 8: two hidden layers with sigmoid and softmax loss

In this case, we add one hidden layer with sigmoid activation function and utilize softmax loss function. The details of hyperparameters are shown in Table 12.

Type	Value
batch size	300
learning rate	0.035
weight decay	1e-4
momentum	0.95
epoch	300

Table 12. Details of the hyperparameters in Experiment 8

3. Evaluation and Analysis

Although we have done 8 experiments, the performance of two of them, one hidden layer and two hidden layers with ReLU function and softmax activation function fluctuate greatly. The seasons for that will be analysed in the following part. Here, we only evaluate 6 of them, all of which perform in stable.

3.1. Hidden layers

Generally, two hidden layers model is better than one hidden layer model, while it takes longer to train. This result conforms to common sense that model with more parameters can fit the function better and, yet need more time to train these parameters.

3.2. Loss function

The computation of softmax loss function is more complicated than euclidean loss function, which lead to more

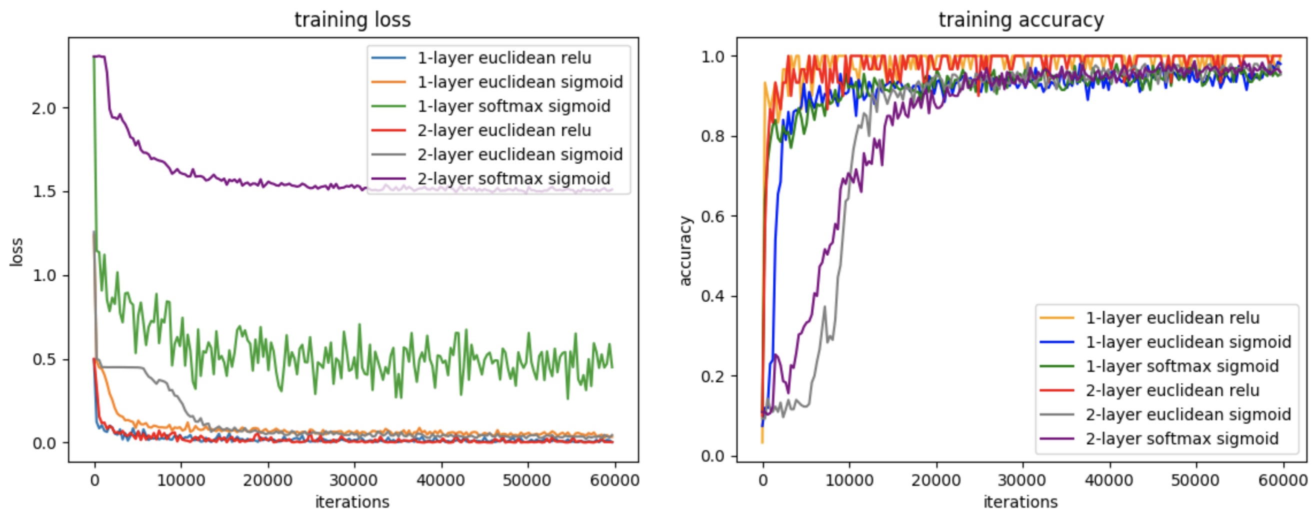

Figure 3. Loss and accuracy for each iterations during training

time consuming on training. In the expect of performance, softmax function performs better in one hidden layer model but worse in two hidden layers model.

3.3. Activation function

Comparing the performance between ReLU and sigmoid, we can tell the fact the ReLU activation function performs much better than sigmoid activation function in both convergence speed and accuracy. We analyse in two aspects.

3.3.1 Gradient

At the positive and negtive saturation region, the gradient of sigmoid is close to 0, which may lead to gradient vanish. However, the gradient of ReLU is constant. Therefore, ReLU function will avoid the problem of gradient disappearance. In some deep neural network, this flaw inn sigmoid may be more striking.

3.3.2 Speed of convergence

The derivative of ReLU function is much easier to compute than the derivative of sigmoid. Therefore, the rate of convergence of ReLU is faster than sigmoid function.

3.3.3 What happened in Experiment 3 and 7

ReLU seems perfect from the analyse above. However, from experiments, we still can discover some weaknesses of is. ReLU will turn all less-than-zero value of gradient into zero, which actually disables some neurons because their gradients are constantly zero. In some setting where learning rate is relatively large, ReLU is prone to be 'dead'. That's why, as for as I'm concerned, the results of experiments 3 and 7 is in fluctuation. In the same hyperparameters setting, sometimes the training accuracy can reach 98% quickly, while sometimes the model can reach limit at 90%. The results in this experimentshave much to do with initializing weight.

4. Some skills in adjusting hyperparameters

- When batch size is increased, the learning rate can be increased slightly with it.
- If batch size is too large, model is prone to fall in local minimum, which if too small, the model will fluctuate greatly. So batch size should be proper. When adjusting batch size, we can also make a try to slightly adjust learning rate.

4.1. Conclusion and future work

In this experiment, we implement MLP neural network and train it on MNIST. The result is satisfactory. We discover ReLU perform much better than sigmoid and two hidden layer model is better than one layer model.

In the future, we can add dropout machanism and try CNN model on the dataset.