Two-Layer Neural Network for Digit Classification on MNIST

Kexin Qin, 21210980117

Abstract

This project implements a Neural Network for the purpose of digit classification. The network contains two layers and is implemented in python without using any advanced machine learning libraries. The network parameters are learned though the backpropagation algorithm with the SGD optimizer. MNIST dataset is used to train and test the network. We implement L2 regularization as well as the Learning rate decay (lrDecay) technique. We have also selected a “best" model via parameter tuning.

1 Introduction

Digit classification is the task of classifying a given image into one of the digits from 0 to 9. This is of key importance in modern day machine intelligence and has many practical applications such as Natural Language Processing (to autograde), Self Driving Cars (to detect speed limits) etc. In this project, we have done the task of digit classification using a two-layer neural network.

The full project can be found on https://github.com/Aria-qin/Two-Layer-Neural-Network-for-MNIST and the trained model may be downloaded at Baidu Netdisk link, pwd: woen.

2 Training Process

Neural network is inspired from the way that biological nervous systems process information. For instance, human nervous system contains millions of neurons and each neuron propagates the information forward to another neuron. Based on the input signal that a neuron receives, it is either activated or not.

Similar to the biological system, a neural network consists of an input layer, any number of hidden layers (including 0), and an output layer. Each layer has a collection of neurons and all the layers are connected in an acyclic manner. In other words, the outputs of some neurons in a layer will be the inputs for other neurons in the above layer.The model is stored in the weights and biases, where weights are the parameters that are used to weigh the incoming inputs to a neuron, thus suppressing some of the inputs while enhancing the others. Then the bias is added, and it is passed through an activation function which decides whether the neuron should be activated or not.

Thus, fundamentally training a network can be reformulated to the problem of learning these weights such that the predicted results at the output layer are as close to the actual results. In this project, we set the loss function and the activation function as follows:

- **Loss Function:** in this project, we use the Mean Squared Error as the loss function to learn the parameters. Assume a sample input $\{(x, y\}, y = (y^1, ..., y^m)$, and the predicted result given by the neural network $\hat{y}^i, i = 1, ...m$, then the loss function is defined as:

$$L(W, b) = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^i - y^i)^2 \tag{1}$$

  To prevent over-fitting, we use L2 regularization with strength λ. The above loss function can be rewritten as:

$$L(W, b) = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^i - y^i)^2 + \frac{\lambda}{2}||W||_2^2 \tag{2}$$

- **Activation Function:** we choose ReLU (Rectified Linear Unit) as the activation function in this project, which is faster than sigmoid to compute because of its simplicity. . ReLU function is defined as the positive part of its argument (see **Figure 1**):

$$ReLU(x) = x^+ = \max(0, x). \tag{3}$$

  Moreover, we add the Softmax function in the last layer, which turns the outputs into a probability distribution.

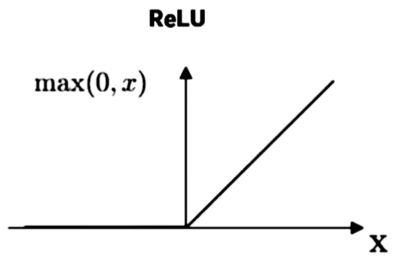

Figure 1: ReLU

Given the loss function, we use an optimizer to update the weights and bias so that we could reduce the loss by iteration. In this project, we use Stochastic gradient descent (SGD) as our optimizer, which is a stochastic approximation of the gradient descent optimization and an iterative method for minimizing the loss function. It replaces the actual gradient (calculated from the entire data set) by an estimate gradient (calculated from a randomly selected sample of the data).

The optimizing process involves two main steps: forward pass and backpropagation, which will be discussed in the next sections.

2.1 Forward Pass

we feed a neural network with data, and it goes through all the layers. Each activation pattern in one layer causes the activation pattern in the next one to eventually get a prediction in the output layer.

$$z_l = W_l \cdot a_{l-1} + b_l \tag{4}$$

$$a_l = \text{activation}(z_l), \tag{5}$$

where l represents layers, W represents layers, and b represents bias. In our project, there are only two activation functions: ReLU for the first layer and softmax for the second layer.

2.2 Backpropagation

Backpropagation is the process of propagating the loss that has been calculated at the end of forward propagation back through the network and update the network parameters so that the loss is minimized. To do so, we make use of the chain rule to calculate the derivative of the loss with respect to parameters. Once we have the gradients of loss with respect to each of the network parameters, we update the parameter with a learning rate. This means that we change the values of the parameters in the deepest direction in which the loss reduces. The parameter update formula for this two-layer neural network is shown below.

$$\begin{aligned}
dz_2 &= a_2 - y, \\
dW_2 &= \frac{1}{m}(dz_2)a_1^T + \frac{\lambda}{m}W_2, \\
db_2 &= \frac{1}{m}\sum(dz_2), \\
dz_1 &= W_2^T(dz_2) \times ReLU'(z_1), \\
dW_1 &= \frac{1}{m}(dz_1)x^T + \frac{\lambda}{m}W_1, \\
db_1 &= \frac{1}{m}\sum(dz_1),
\end{aligned}$$

where m is the batch size.

3 Implementation

3.1 Architecture

In this project, we have implemented a neural network for the purpose of digit classification. The dataset that we have used in this project is the MNIST dataset in which each image is represented by 28 x 28 pixels. Each pixels value ranging from 0 to 255. We use these 784-pixel values to represent the input and to train the network. Therefore, we have 784 neurons in the input layer and 10 neurons (labels from 0 to 9) in the output layer. And we add a hidden layer to the neural network. The structure of our neural network as shown in **Figure 2**.

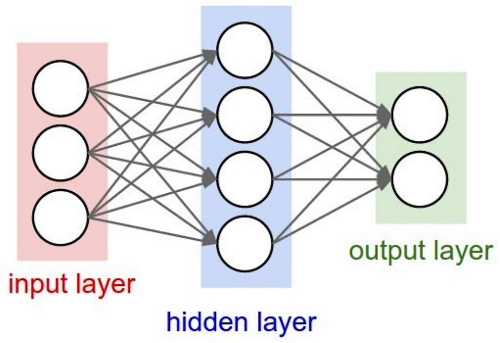

Figure 2: Structure of the two-layer neural network

3.2 Preparing the training data

MNIST database of handwritten digits is used to train the network and to test the accuracy. I download the data into the csv data files: *mnist_train.csv* and *mnist_test.csv*, they contain 60000 and 10000 images, respectively. Each image in the dataset is in the form of a list of 784 features corresponding to the 784 pixels with a value from 0 to 255. The label corresponding to that image is a number from 0 to 9.

For preparing the training data, we encode the labels as one-hot vectors. For instance, a label of “2" has been encoded as a list of 10 in which the second index is 1 and the rest are zeros i.e. [0,1,0,0,0,0,0,0,0,0]. Besides, we have normalized each pixel value to be in the range [0.01,1] by dividing each value by 255/0.99 and plus 0.01.

3.3 Parameters

3.3.1 Initialization

I use “He Normalization" method proposed by with normal distribution to initialize the weight and bias parameters with mean = 0 and standard deviation = sqrt((2 / (Number of neurons in the previous layer))). The rate of learning rate decay is set as 0.95, the batch size is 64. And we will run the training for 50 epochs. We shuffle the training data at the beginning of every epoch.

3.3.2 Parameter Tuning

In this project, we mainly test 3 hyperparameters: the size of hidden layer, initial learning rate and regularization strength.

- **The size of hidden layer:** First, I choose the size of hidden layer as (128, 256, 512) respectively, and the results show that the hidden layer with 512 neurons performs best. Then I randomly choose 6 integers form [480, 540], the hidden layer with 512 neurons still has the best performance, while the differences are minimal.

- **Learning rate:** I conduct a exploration of learning rates on a logarithmic scale, the values are set as $10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}$. The neural network with initial learning rate 10^{-1} performs surprisingly well, reaching a training accuracy 99.9%.
- **Regularization strength:** Similarly, I conduct a exploration of regularization strength on a logarithmic scale, the values are set as $10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}$. The results show that when the regularization strength is set as 10^{-4}, the model has the best performance. But there is not much difference, 10^{-4} beat the others just at 0.1% level in test accuracy.

3.3.3 Final Setting

The parameters of the best model we find are as follows:

Table 1: Parameters of the best model

Hidden layer size	512
Learning rate	0.1
Regularization strength	10^{-4}

4 Results

With the parameters shown in table 1, our two-layer neural network can achieve a train accuracy **99.87**% and test accuracy **98.17**%.

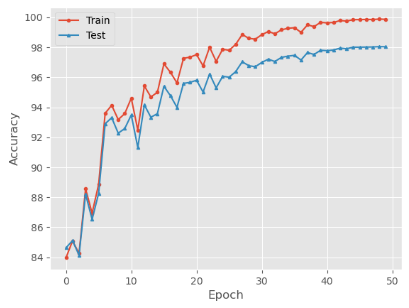

Figure 3: Accuracy against epochs

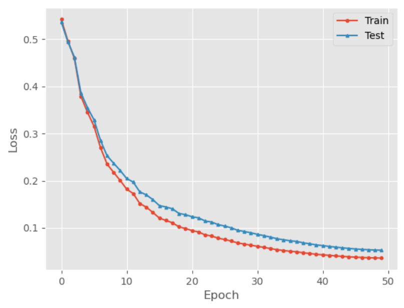

Figure 4: Loss against epochs

The accuracy and loss values against the number of epochs are plotted in **Figure 3** and **Figure 4** , respectively. We can see from **Figure 3** that both the train accuracy and the test accuracy fluctuate violently at the first epochs because of the stochastic property of SGD, then change slightly after 30 epochs, . The train accuracy reaches a value about 99.9% at the end, while the test accuracy is a little bit lower, ended with a value of 98.17%. This indicates that the network might be overfitted.

As for the change of loss against epochs, the loss of train set keeps decreasing along 50 epochs. The loss of train set achieves about 0.035 at the end, the loss of test set is higher, reaching about 0.053 at the end.

Besides, we have also visualized the weight matrix in each layer. The shape of W_1 is 784×512, we reshape each column of it to an image of (28, 28), which will produce 512 images representing 512 features in the first layer. To show the results more clearly we just choose the first 100 images and plot them in **Figure 5**.

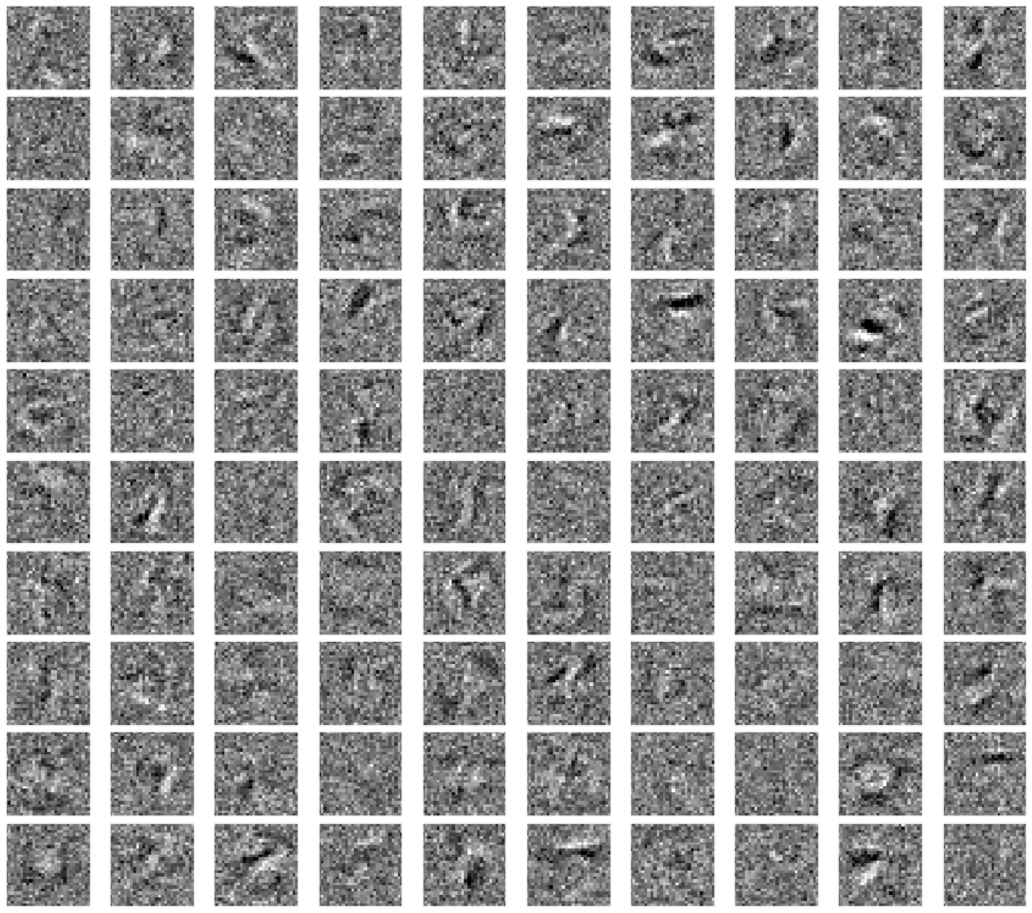

Figure 5: Visualization of the weight matrix of the first layer, W_1

From **Figure 5**, we can see that the first layer of neural network extract some specific combinations of the 784 input features, and map them into a reduced space with 512 features.

For the weight matrix W_2 of the second layer, it has the shape of 512×10. I tried two ways to visualize W_2:

1. visualize W_2: 10 images, each image is constructed from a column of W_2. The shape of each image is

32.

2. visualize $W_1 \cdot W_2$: $W_1 \cdot W_2$ has the shape of 784×10. Similar to the way we visualize W_1, we split the matrix into 10 columns, and plot each column to a 28×28 image.

These two visualization methods are shown in **Figure 6** and **Figure 7**, respectively.

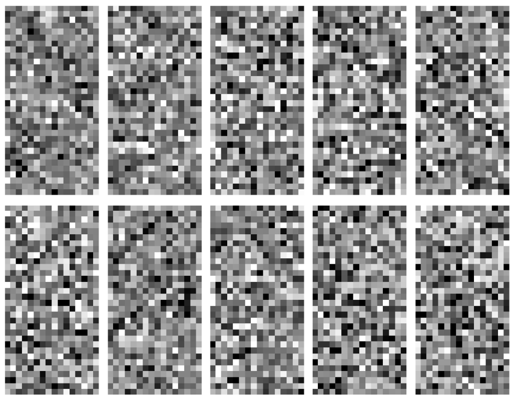

Figure 6: Visualization of the weight matrix of the second layer, W_2

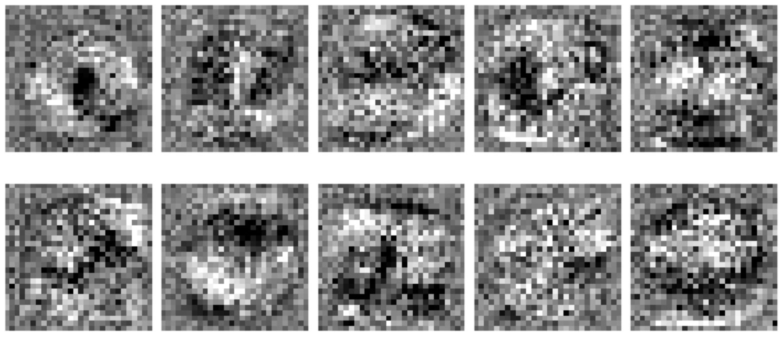

Figure 7: Visualization of the product of two weight matrices, $W_1 \cdot W_2$

Figure 6 gives little information about the extracted features since its shape is irregular, while in **Figure 7**, we can find depict a general shape of 10 figures. The difference between **Figure 7** and the real shape of figures is brought about by bias.

References