Coursework 1 Report

Neural Computation - Group 8

November 20, 2020

1 Introduction

In this report, we aim to present the process of hyper-parameter tuning on feed-forward neural network trained on course work data set and our final choice of hyper-parameters. This report is consisted of Introduction, Methodology, and Conclusion. The methodology part goes as pre-processing,training time, activation, batch size, learning rate and topology.

2 Methodology

2.1 Pre-processing

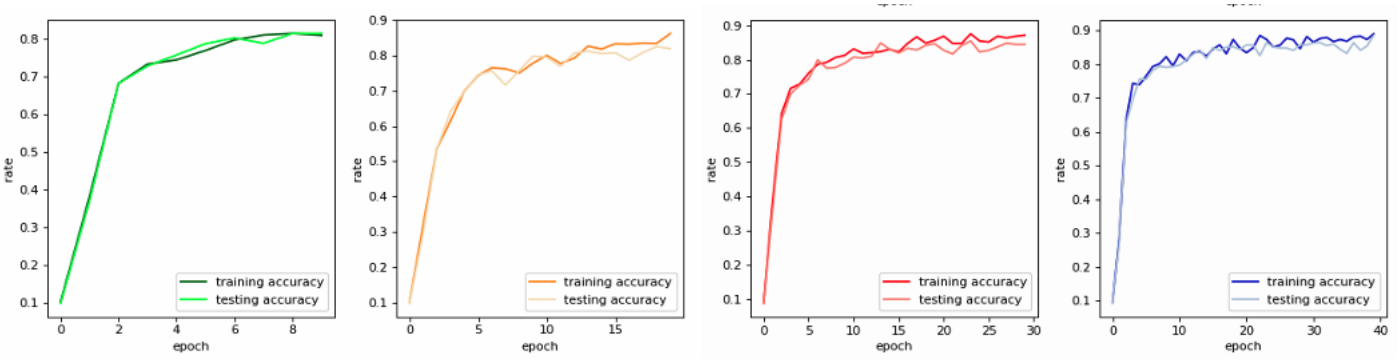
Before feed the data into our neural network, we normalize the input value from [0,255] to [0,1], which can accelerate converge and improve accuracy.

2.2 Training time

One epoch is defined as one complete pass through the whole training set, i.e., it is multiple iterations of gradient descent updates until we show all the data to the neural network, and then a new epoch starts again. So the number of parameters 'epoch' could be translated as the times we feed the training data to the whole neural network. The training time can be expressed by the following equation:

Training time = Number of epoch * Training time for one epoch

So, Intuitively we would find an appropriate training time is equivalent to find the most suitable epoch number for our neural network. If the number of epochs is too small, the accuracy of the training set and the test set is still rising, so it is not suitable to stop training currently. On the contrary, if the number of the epoch is too large, the accuracy of the training set and test set has been stable for a long time, even causing an overfitting phenomenon. Therefore, too large a number of epochs is also not appropriate. In order to achieve a balance between fitting to training data and globalization of the network, we picked a test set of epoch 10, 20, 30, 40 with the rest of the other hyper-parameter unchanged. The following figure describes the accuracy of the training set and test set changes with the number of epochs



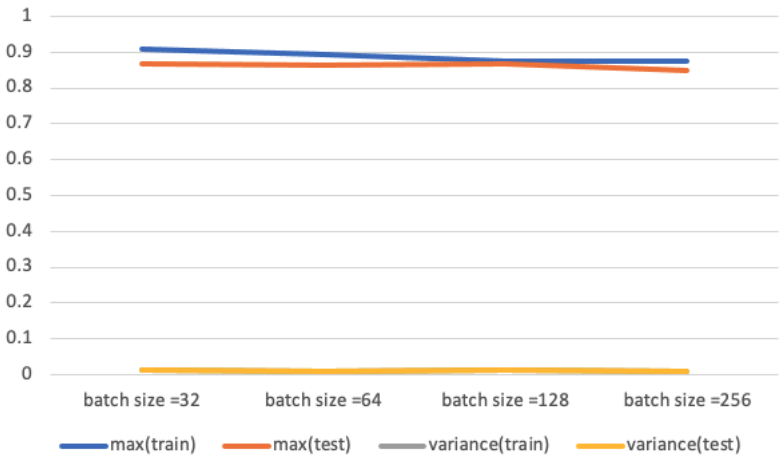
	Epochs=10	Epochs=20	Epochs=25	Epochs=30	Epochs=40
max(train)	0.814	0.862	0.875	0.877	0.885
max(test)	0.814	0.825	0.854	0.854	0.883
variance(train)	0.050	0.035	0.026	0.025	0.027
variance(test)	0.051	0.034	0.025	0.025	0.026

We can see from the first two plots that the accuracy still on its ascent phase, and the table also tell us the variance of 10 and 20 of the epoch are relatively larger than that of other numbers of the epoch, which means the accuracy is not stable. So stop training at the 10 of the epoch is not a good choice. The graph shows that when stepping over 3, the accuracy stays upward trend, which is an improvement we want. However, when over 25, the iterations of data feeding did not show a significant increase in accuracy and variance kept a low level but increase the overfitting rate. To prevent overfitting happens in our model, we would like to choose an **'early stopping'** strategy, i.e. end training early. So from the above analysis, we could figure out that we want the epoch is between [25, 30]. In this interval, the specific epoch value should match the appropriate batch size, the larger batch size matches the smaller epoch, and vice versa. Similarly, according to the formula above training time will be in **[25 * Training time for one epoch, 30 * Training time for one epoch]** for our model.

2.3 Batch size

The batch size defines the number of samples that will be propagated through the network.

From the perspective of epochs, the best range is between 25 and 30, and the batch size is also relatively good in this range compared with other batch sizes.Second, if the sample size is large, the normal mini-batch size is 64 to 512. Given the way the computer memory is set up and used, the code will run faster if the mini-batch size is 2^n.



(Figure 1)

epchos=30	batch size = 32	batch size = 64	batch size = 128	batch size = 256
max(train)	0.907	0.893	0.874	0.874
max(test)	0.869	0.865	0.869	0.85
variance(train)	0.014	0.01	0.012	0.011
variance(test)	0.013	0.008	0.012	0.01

(Table 1)

Because we can find from the epoch when the epoch between **25** and **30** is the best. We considered the impact of different batch sizes on the results when **epochs** =30.As we can see from **Figure 1**, when the epochs value is fixed, we change the batch size. We adjust the batch size between **32** and **256**.We found little change in accuracy and variance, but **256** is considered the optimal solution among these four values.

In general, depending on your GPU's video memory, set it to the maximum, and generally require a multiple of 8 (say, 32,128), so that parallelism inside the GPU is the most efficient. The CPU has prefetching, and the GPU has merge access, which requires not only the length to be 2 to the power, but also the memory address. In addition, It is said that GPU can give better performance to batch power of 2, so the performance when the batch sets to 16, 32, 64, and 128 is usually better than when the batch is set to multiples of 10 and

100.

In summary, since our data was run on the CPU, the data tended to be stable without little difference, but in fact, the larger the batch size, the better the result.According to our experimental proof, we finally choose **256** as our optimal solution. We know that the larger the batch size is, the better the final result will be produced. However, the larger batch size is not the better, and the decision will be made according to the actual situation.

2.4 Activation function

Sigmoid function:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Relu function:

$$Relu(z) = max(0, z)$$

Before we start to do some experiments to compare their classification accuracy. We can make a theoretical prediction first. We guess that the classification accuracy of the relu function is better than the sigmoid function. We can compare the formulas of these two activation functions. Although the sigmoid function is a smooth function that's easy to differentiate, its output is not a zero-means. This means that the neurons in the next layer will get the signal of the non-zero mean output from the previous layer as input, which will change the original distribution of the data as the number of network layers increases. In addition, the range of values of the sigmoid function is from **0** to **0.25**. Then the gradient vanishing may happen because of the "chain reaction" when the neural network does back-propagation. We can find that our neural network is a five-layer network structure with three hidden layers, so the sigmoid function seems to be less effective when applied to the multiplayer network structure.

The gradient of the relu function is constant in the part of greater than **0**, so there is no gradient vanishing phenomenon. Also, the derivative of the relu function is calculated faster, so when using gradient descent, it will converge much faster than sigmoid function. We should notice the "dead relu problem", because the relu function forces the output of the part **x¡0** to be set to **0**, which may cause the network model to fail to learn effective features of data. As a result, if the learning rate is set too high, it may cause most of the neurons in the network to be in a "dead" state.

Combining the above analysis and the comparison of different learning rates, we set the learning rate to **0.0001**. And we set the batch size(**30**) and epochs(**200**) to the most suitable parameters through experiments before. We found that the classification accuracy of the relu function applied to the current model is stable between **80%** and **85%**. Meanwhile, the final fitted curve is very stable and there is no over-fitting or under-fitting phenomenon. But when we put sigmoid function in the current model, it can only achieve approximately **20%** accuracy. Although the classification accuracy will be improved when we try to reduce the number of hidden layers, the sigmoid function still cannot achieve a good classification accuracy in the current model compared by relu function. And the following table records the classification accuracy by using sigmoid function with different number of hidden layers.

	3 hidden layer	2 hidden layer	1 hidden layer
accuracy for sigmoid function	20%	30-40%	45-60%

As a result, when the neural network model is more layered structure, we prefer to choose relu as the activation function.

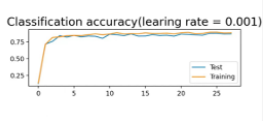
2.5 Learning rate

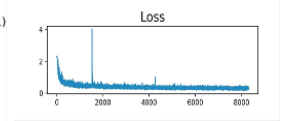
The learning rate represents the step length adjusted by the weight parameter in the inverse gradient direction. The smaller the learning rate, the better the model effect, but it will bring extremely slow speed and fall into the local optimum. So, we need to increase the learning rate, but too large a learning rate will cause the function to fail to converge, or jump out of the best point, and the model training effect becomes worse. Therefore, choosing an appropriate learning rate is a very critical indicator for training the model.

In the current research, we set the learning rate manually. First, set a larger learning rate to make the loss value of the network drop quickly, and then reduce the learning rate a little bit as the number of iterations increases to prevent the global optimal solution from being exceeded. In the process, we keep **epochs=30** and **batch-size=200** unchanged in order to control variables.

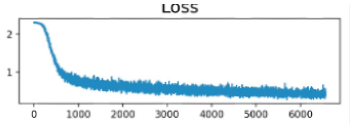
learning rate = 0.01: From the figure1 below, we can find that the accuracy of both train and test is particularly low and the fitting effect is particularly poor.

learning rate = 0.001: From Figure 2 we can see that when the learning rate is 0.001, the accuracy rate is greatly improved to about 90 and the fitting effect is also good, but we can see from Figure 3 that the loss function drops particularly fast at 1800 and 4200 At that time, there was a problem with convex convergence, so we need to further reduce the learning rate.

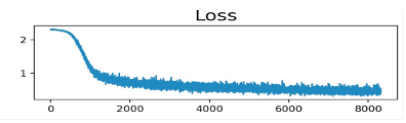




leading rate = 0.0001: From Figure 4, we can see that the fitting effect and accuracy of the two curves have not changed much compared to when the learning rate is 0.001, but the loss function graph in Figure 5 has changed significantly, the bulge disappears and decreases Slowdown.



leading rate = 0.00005: From Figure 6, we can see that the curve fitting effect is still good, but the accuracy rate drops to only 82. In the loss function graph of Fig. 7, the drop is very slow and consumes a lot of training time.



In summary, whether it is accuracy, loss function and training time, the learning rate is more accurate at around 0.0001.

2.6 Topology

The topology or architecture of neural network involves 3 layers, i.e. input layer, hidden layers, and output layer. In this question to classify pictures into categories, the input layer is designed by size of picture which is 28 * 28, and the output layer is decided by the number of categories which is 10. So, the only one left to be tuned is the hidden layer. Why we need more than one hidden layer? From Deep Learning, 2016, we know that a single hidden layer feed forward network would perform well on a linear separable model. But if we need to solve image classification problem which fits non-linear model, we need to increase the hidden layers. In our case, we think that we need more than 2 hidden layers to extract image features to adjust to the activation function we picked. So, we picked our hidden layers size ¿= 2, and we design the topology samples to be 2 hidden layers or 3 hidden layers. Still we would have a one-layer topology as comparison.

With activation set as “relu”. Topology could be viewed as “independent” to the other hyper-parameters and here is a shot fo r specific number.

```
hyper_parameters = {'activation_function' : "relu",
                    'learning_rates' : [0.0001],
                    'batch_sizes' : [200],
                    'epochs' : [30], }
```

With these hyper-parameters, we trained our model with all the sample in our topology samples and the last sample is provided as default.

```
topology_samples = [[784, 40, 10],
                    [784, 300, 40, 10],
                    [784, 30, 20, 20, 10]]
```

Here is a table for Highest test accuracy, its overfit rate *i.e.*

$$overfitRate = testacc \div trainacc$$

$$overfitRate \geq 100\% : underfit$$

$$overfitRate < 100\% : overfit$$

and on which epoch this was trained.

	[784, 40, 10]	[784, 300, 40, 10]	[784, 20, 20, 10]	[784, 30, 20, 20, 10]
Max test accuracy	0.856	0.864	0.863	0.864
Overfit rate at max test acc	98.28%	100.8%	99.1%	101.8%
Epoch no. at max test acc (out of 30)	27	24	26	19

The table displays that with the increase of hidden layers from 1 to 2, the network shows slight increase in max test accuracy. Also, we could see that the network learned the best result faster. However, this topology failed to completely learn the training data and had underfitting problem. When we increase hidden layers from 2 to 3, we have no increase in test accuracy, but the network learned the result too fast and did not fit the training data well enough. It is possible the network operates too many gradient decreases and slipped away from the minimum loss point which resulted in the underfitting problem.

From the data and analysis we did above, we choose a 2-hidden layer topology and, after a couple of tests we set it to be [784, 20, 20, 10] as this one performs well (86.3%) in the test accuracy and does not suffer sufficiently in underfitting problem.

3 Conclusion

From experiment in the methodology part, we have considered multiple discriminating factors such as maximised test accuracy, accuracy variance and *etc.*. Finally we could bring the hyper-parameters to

- epoch: 30
- activation function : Relu
- batch size: 256
- learning rate: 0.0001
- topology: [784, 20, 20, 10]