COMP5329: Assignment1 Report

490005761 zlyu0226

April 8, 2022

1 Introduction

1.1 Aim of study

The aim of this study is to understand the fundamentals of deep learning by implementing simple multilayer perceptron neural networks and related functions such as optimizers, activation functions and loss functions.

1.2 Importance of study

The importance of this study is that the best way to learn the principles of deep learning is to practice them.

However, mature frameworks often hide the process and methods of implementation from the user, wrapping complex processes in simple interfaces.Calling these encapsulated interfaces does not allow people to understand the principles, but merely to become "skilled interface users". Once a problems arises, When a problem arises, those who do not understand the principles are often unable to locate the root cause of the problem, or to deduce how to optimise the model by modifying hyperparameters.

In this study, by implement an independent deep learning framework, subjects will obtain a better understanding towards the underlying principle of deep learning, and thus become a better problem solver in deep learning area.

2 Methodology

2.1 Preprocessing

2.1.1 Data Overview

The given data set is in shape of $(50000, 128)$, with a label of single integer value (0-9). The feature data has a standard deriviation of 1.1688637 and mean -1.59374025. From the data quality perspective, the standard deriviation is very close to 1 and mean is close to 0, which is pretty good for model training.

For the label data, the ground truth of samples are perfectly distributed. There are 10 classes in total, 50000 samples, and 5000 samples for each class precisely.

2.1.2 Preprocess the Feature Data

Although the standard deriviation and mean of the feature data is good, there is still room for improvement. Before training, the feature data was standardized to 0 mean and standard deriviation 1 using the following method:

$$x_{Standardized} = \frac{x - \mu}{\sigma}$$

Where μ stand for mean of x and σ is the standard deriviation of x.

2.1.3 Preprocess the Label Data

The label data is in form of (0-9), which is numerically continuous but discrete in meaning. To make it able to be feed into the model, it must be encoded. Before training, the label data was one hot encoded from single integer to a 10 element vector. For example:

$$OnehotEncode([0, 1, 2]) = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]$$

2.2 Design

The best model is designed with 3 hidden layer, with 64, 32, 10 neuron seperately, initialized by Xavier Normal initialization. Each hidden layer is followed by a batch-normalization layer and a dropout layer with drop rate 0.25. Activation function is ReLU. Specifically, for last layer, the activation function is Softmax. Optimizer is Adam with momentum $\beta_1 = 0.9$ $\beta_2 = 0.999$, learning rate $1e - 3$ and weight decay 0.02. Loss function is Cross Entropy.

2.3 Performance

The best model has obtained 0.5301 accuracy on testset meanwhile 0.1408 loss on trainset. For this result, the model has been trained for 91 epoch in 4min 28s on the test platform with Intel i9-9900K CPU @ 3.60GHz, Windows 11 OS, python 3.8.11, Numpy 1.22.3 and Scipy 1.8.0 through Jupyter lab 3.3.2. No other package or accelerator involved. In the further section, the experiments were performed on this platform as well. The best result given above was obtained at epoch 70.

2.4 Module

2.4.1 Layer

In a multilayer perceptron, the hidden layer is the fully connected layer. Its role is to take the features of the input data and abstract them into another

dimensional space to reveal its more abstracted features, which can be better divided linearly. Multiple hidden layers are multiple levels of abstraction of the input features to achieve a better linear division.

The layer keeps multiple neurons, each one contains two variable, weight and bias. Each neuron for each input X, it does:

$$w^T X + b$$

For a well-trained model, each neuron can be activated by a specific combination of features, outputting a strong signal to the next layer, thus contributing to the final result.

2.4.2 Batch-normalization

Batch-normalization layer is to normalize the batch data during the training process. This layer will compute a moving average of mean and standard deriviation of output from previous layer, and normalize them to reach 0 mean and 1 standard deriviation.

$$\mu_t = \mu_{t-1} * (1 - momentum) + momentum * \mu$$
$$\sigma_t = \sigma_{t-1} * (1 - momentum) + momentum * \mu$$
$$Normalized_t = \frac{input_t - \mu_t}{\sigma_t}$$

After normalization, the batchNorm layer will also do a shifting like hidden layer:

$$out_t = \gamma Normalized_t + \beta$$

Where γ and β are also trainable.

2.4.3 Dropout

The dropout layer is used to make the output of the hidden layer sparse and reduce the complex co-adaptation relationship between neurons, thus reducing the probability of overfitting the model.

The principle of it is to randomly set neurons' output to zero. The probabilty of this zerolization is user-defined.

2.4.4 ReLU

ReLU, Rectified Linear Unit, is a kind of activation. It works by non-linearly transforming the linear output of each layer, so that the whole model becomes non-linear and more complex decision boundaries can be formed.

It works by zeroing out negative numbers in the output of the previous layer, leaving values greater than or equal to zero as they are.

2.4.5 Softmax

Softmax maps some inputs to real numbers between 0 and 1, and normalizes them to ensure that the sum is 1. It is designed to be used for multiple classification tasks, since the probability of multiple classes also sums to 1.

For the input $V = [V_1, V_2, \dots V_j]$:

$$Out_i = \frac{e^{V_i}}{\Sigma_j^i e^{V_i}}$$

2.4.6 Adam

Adam optimization is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments. Effective control of the learning rate step and gradient direction through first and second order momentum, preventing oscillation of the gradient and stationarity at the saddle point.

If weight decay is defined, the gradient will be added by the $W * decayRate$ before updating.

$$g_t = g_t + W_{t-1} * decayRate$$

First order moment: a moving average of gradient.

$$m_t = \beta_1 * m_{t-1} + (1 - \beta_1) * g_t$$

Second order moment: a moving average of squared gradient.

$$V_t = \beta_2 * V_{t-1} + (1 - \beta_2) * g_t^2$$

After the moments were computed, update the variable

$$W_t = W_{t-1} - \frac{lr * m_t}{\sqrt{V_t}}$$

2.4.7 CrossEntropy

CrossEntropy loss function is designed to evaluate the prediction performance in multi-classification task. Its derivative is directly depends on the prediction and ground truth, thus it will be a very good loss function when the model is performing bad. But for a better model, the learning speed will decrease.

For prediction p and ground truth y:

$$L = \frac{1}{N} \sum_i L_i = -\frac{1}{N} \sum_i \sum_{c=1}^{M} y_{ic} \log(p_{ic})$$

Where M indicates the total class number and c stand for each class.

3 Experiments and Results

To explore the best model, various hyper-parameter has been attempted. Following tests are performed with an Early-Exit Pipe which will stop the training process when the metric does not come better in certain rounds, (See detail at appendix or dl.utils.EarlyStopping.EarlyStoppingPipe) thus the total epoch may vary.

A set of hyper-parameter has been selected at the begining the test.

- Hidden Layer number: 3
- Batch size: 128
- Preprocessing: Standardization
- Optimizer: SGD
- LR: 1e-3
- Initialization: Kaiming normal initialization
- BatchNorm: Yes
- Dropout: 0.25

After a set of comparison or ablation studies, the best set of hyper-parameter will be selected.

To measure the performance, we have the metrics of training loss, training accuracy, test loss and test accuracy. The training accuracy and test loss does not mean a lot, and since this is a classification task, thus the test accuracy will be the metric which has been weighted the most.

3.1 Hidden Layer

In this experiment, the models with 2, 3, 4 hiddenlayers will be evaluated.

Hidden Layer	Test Accuracy	Epoch	Time Cost/s
2	0.4846	33	101.91
3	0.5068	18	89.39
4	0.4891	72	331.5

Table 1: Performance of different hidden layer number

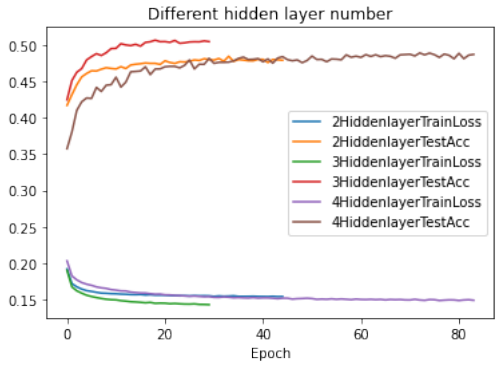

Figure 1: Performance of different hidden layer number

As observed, 3 hidden layer obtained both fast training and higher accuracy.

3.2 Batch Size

In this experiment, batch size of 64, 128, 256 has been tested. Result as below:

BatchSize	Test Accuracy	Epoch	Time Cost/s
64	0.5168	54	199.2629
128	0.5109	23	74.8191
256	0.5073	18	52.7593

Table 2: Performance of different batch size

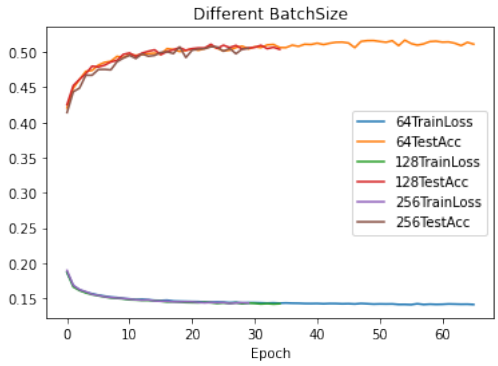

Figure 2: Performance of different batch size

As observed, batch size does not significantly affect the model performance. However,the training time seems like sensitive to it. Batch size 128 model has

stopped converging at the very beginning, but the 64 batch size was osillating until 3 times of 128's best epoch.

Although batch size 64 has obtained the best performance, but it's not significant and a huge over-cost on training time. To sum up, 128 would be the best batch size.

3.3 Preprocessing

Pre-processing the data can often lead to positive feedback. However, given that the distribution of the original data is already good, pre-processing may still have negative effects. The result of raw data and standardized data as below:

Preprocessing	Test Accuracy	Epoch	Time Cost/s
Standardized	0.5115	28	158.7205
Raw	0.5188	40	151.399

Table 3: Performance of different preprocessing

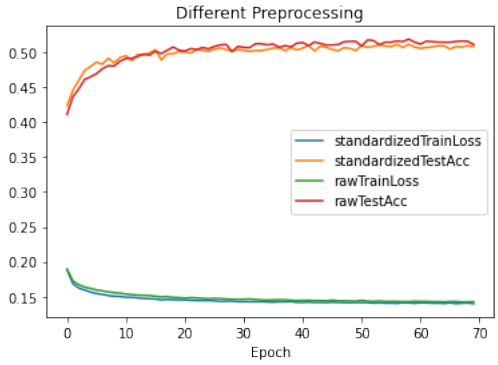

Figure 3: Performance of different preprocessing

It can be observed that there is no significant difference on performance between the models trained by raw data or the standardized data.

This is speculated to be because the distribution of the original data is already good. So in this task, the original data will be chosen for training.

3.4 Optimizer

In this experiment, the different optimizers will be test, which are Adam and SGD.

Optimizer	Test Accuracy	Epoch	Time Cost/s
SGD	0.5114	36	102.9382
Adam	0.5231	40	160.5311

Table 4: Performance of different Optimizer

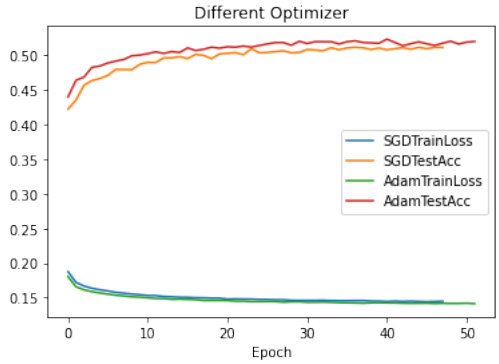

Figure 4: Performance of different Optimizer

As observed, Adam has a slightly better performance.

3.5 Learning Rate

In this experiment, the model will be trained with 0.01, 0.001 and 0.0001 learning rate.

Learning Rate	Test Accuracy	Epoch	Time Cost/s
0.01	0.5018	39	162.8090
0.001	0.5266	91	254.2694
0.0001	0.5140	153	492.1130

Table 5: Performance of different Learning Rate

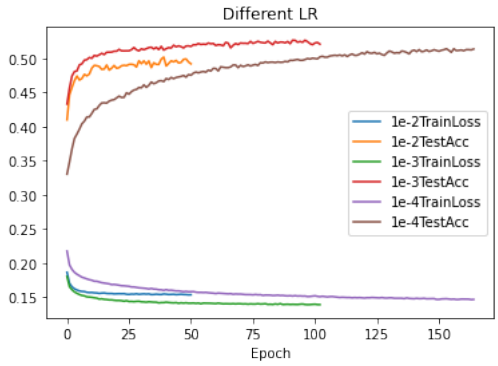

Figure 5: Performance of different Learning Rate

As observed, a learning rate of 0.01 performs pretty bad. It stopped converging very early and did not reach an accuracy as high as the others. 0.001 seems pretty good, with the best performance with a feasible converging time. As for 0.0001 lr, it takes too much time and did not reach the best performance. This is probably due to stucking in the local optimal.

3.6 Initialization

In this experiment, various layer initializations will be evaluated. Include Xavier Normal/Uniform and Kaiming Normal/Uniform.

Initialization	Test Accuracy	Epoch	Time Cost/s
KaimingNormal	0.5208	34	114.6010
KaimingUniform	0.5154	34	142.7880
XavierNormal	0.5239	61	206.2885
XavierUniform	0.5236	46	139.4836

Table 6: Performance of different Initialization

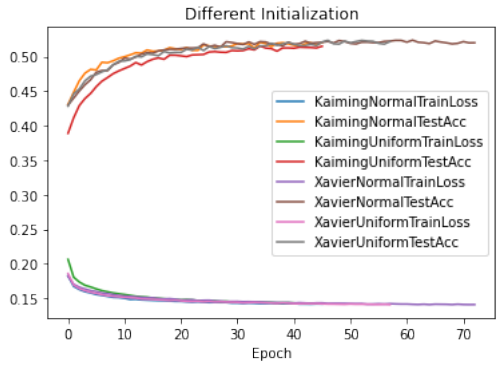

Figure 6: Performance of different Initialization

As observed, Xavier initializations are better than Kaiming in this task. The Xavier Normal is slightly better than the Xavier Uniform.

3.7 Batch Normalization

To explore the influence of Batch Normalization, this experiment will evaluate the models with/without batchNorm.

Result as follows:

BatchNorm	Test Accuracy	Epoch	Time Cost/s
With	0.5241	64	203.3823
Without	0.5248	84	252.7695

Table 7: Performance of models with/without BatchNorm

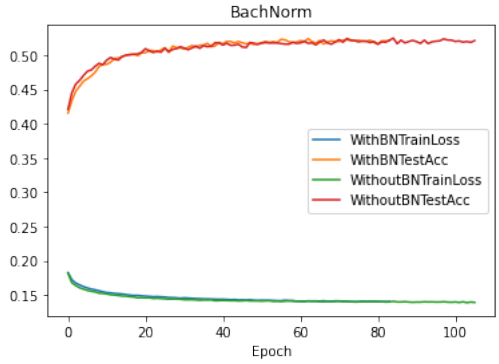

Figure 7: Performance of models with/without BatchNorm

It can be observed for this task, bachnorm layers is not significantly affecting the performance.

3.8 Dropout

To explore the influence of Dropout, this experiment will evaluate the models with/without Dropout.

Result as follows:

Dropout	Test Accuracy	Epoch	Time Cost/s
With	0.5224	81	245.4308
Without	0.5247	13	71.3339

Table 8: Performance of models with/without Dropout

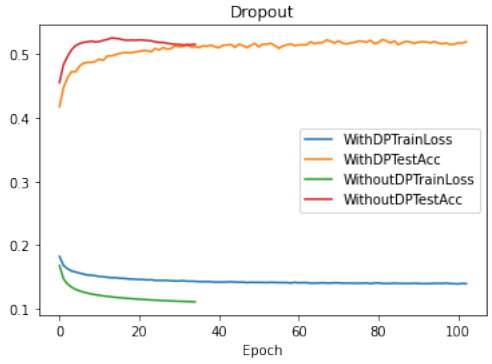

Figure 8: Performance of models with/without Dropout

It can be observed that without dropout the loss was dropping dramtically, but soon the accuracy on testset began to drop which can be defined as overfitting. For the model with dropout, the loss curve was not dropping fast, but the accuracy on testset was keep raising, though there were some ocillation. It can be seen that Dropout can indeed prevent overfitting.

The experiment above was not involving batchNorm. There is another result for with/without Dropout when Batchnorm is presented:

Dropout	Test Accuracy	Epoch	Time Cost/s
With	0.5259	78	272.3730
Without	0.5084	166	562.3526

Table 9: Performance of models with/without Dropout

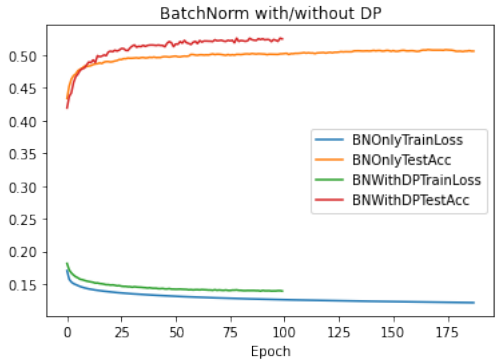

Figure 9: Performance of models with/without Dropout

It can be seen that DP will be better if work with BatchNorm layer. Another found is that Batchnorm without dropout will greatly reduce the converging speed and lower the running speed. This reveals another important function of Dropout, which is reducing the workload of both forward/backward proporgation.

4 Discussion and Conclusion

In this task, an understanding of the underlying logic of deep learning was gained by implementing a multi-classifier from scratch.

The implemented multi-classifier has been experimented with several times, applying well-established techniques and substituting different hyperparameters, without obtaining satisfactory results, and the reasons for this are worth considering. One possible reason for this is that the decision boundaries between the different categories in the dataset intersect, resulting in the model not learning each category correctly. From this persepctive, it reveals a very important fact, which is that all amazing machine learning techniques are based on data, and only when the quality and quantity of the data is high enough can the model achieve a good performance.

From some noteworthy experiments, it has been observed that a slight change on some hyper-parameters will have a very significant impact on the performance of the model, like learning rate. Also observed is the power of certain machine learning techniques, such as dropout, to mitigate overfitting while improving the speed of training. This indicated that care should be taken when choosing hyperparameters, and the best way to get the best set of hyperparameters is through continuous experimentation.

5 Appendix

5.1 Manual

The module implementation has been finished in a form of python package named "dl" in the directory "dl" of the project root.

The best model is trained in the model.ipynb in the root of this project. It can be directly opened in jupyter lab/notebook and run in the same directory with the package (if the package has not been installed). You can use it as the sample of your own usage.

5.1.1 Prerequests

python $\geq$ 3.8, Numpy $\geq$ 1.22 and Scipy $\geq$ 1.8

A lower version pakage is likely to be working, but not guarenteed.

5.1.2 Setup(optional)

If an installation is prefered, run the following code in the root of this project to install the package:

```
1  cd /path/to/package/and/setup.py
2  pip install .
```

5.1.3 importing

There are 3 options to use this package.

If the package has been installed:

```
1  import dl
```

at the top of your python script directly.

If the package has not been installed, you may add following lines to use the package:

```
1  import sys
2  sys.path.append("path/to/package")
3  import dl
```

Or if you create any python script under the root of this project, you can import the package directly as well.

5.1.4 Usage

For detailed usage, checkout the comment or run:

```
1  pydoc -p [port]
```

in the command line. Replace [port] by your preference. Then open the website "localhost:[port]", search for package "dl".

To train a model:

1. Read your data set, preprocess them.
2. Construct a dl.dataset.Dataset object by your data set.
3. Construct a dl.nn.Module object to define your network structure and forward behaviour.
4. Define an optimizer using classes in dl.optimizer. You may define your own by inherit class dl.optimizer.Optim.
5. Define a lossfunction using classes in dl.metrics. You may define your own by inherit class dl.metrics.LossFunction.
6. Define you training process. For each iteration of your training, you should:
   (a) Call the model object to process feature data, get the returned yhat.
   (b) Call the loss function with yhat and ground truth to calculate the loss.
   (c) Call the loss.backward() to perform a back proporgation.
   (d) Call the optimizer.step() to update the parameters.