Assessment Report

Jiaxu Li

April 2, 2024

Abstract

This assignment tasks was undertaken the full machine learning pipeline, including data handling and processing, model construction and training, and evaluation of the developed model. 3-hidden layers neural network was built to address the classification task only using core Python and NumPy.

1 Data Preparation

1.1 Import the dataset

The given dataset has 178 records about chemical properties of food product, which 13 features and 1 label for each record. The labels 1, 2, 3 represent 3 different manufacturers, and 13 features representing 13 different materials that contained in the product. The features of the dataset are the following:

- Amino$_a cid–Thetotalpercentagecontentofanimoacid.$
- Malic$_a cid–Thepercentagecontentofmalicacid.$
- Ash – The measure of ash present in the product.
- Alc – The alcalinity of ash present.
- Mg – The measure of magnesium.
- Phenols – The total measure of phenols.
- Flavanoids – The measure of flavonoid phenols in the product.
- Nonflavanoid$_p henols–Themeasureofnon-flavonoidphenolsintheproduct.$
- Proanth – Proanthocyanins measure.
- Colo$_i nt–Thecolorintensity.$
- Hue – Hue of the color.
- OD – The protein content of the product.
- Proline – The measure of proline amino acids.

The code to load the dataset into python as below

```
import panda as pd
def load_dataset(self,data_path):
    self.dataset = pd.read_csv(data_fpath)
```

1.2 Preprocess the dataset

Data cleaning and data preprocessing play a crucial role in Machine Learning. In this task, a Python function `null_check` was used to check the null values in the dataset. After splitting the dataset in to 6:2:2 train, validation and test set , the `data_analysis` function was used to plot the boxplot to observe the outliers in each feature. A heatmap was drawn by the `feature_importance` function to show the correlation between the features and the target.

```python
from sklearn.model_selection import train_test_split
def check_null(self):
    #Check
    null_values = self.dataset.isnull().sum()
    return null_values

def data_analysis(self):
    features = ["Amino_acid", "Malic_acid", "Ash", "Acl", "Mg",
        "Phenols", "Flavanoids", "Nonflavanoid_phenols",
        "Proanth", "Colo_int", "Hue", "OD", "Proline"]
    plt.figure(figsize=(15, 10))

    # plot boxplot for each feature
    for i, feature in enumerate(features):
        plt.subplot(4, 4, i + 1)
        sns.boxplot(x=self.dataset[feature])
        plt.title(f"{feature} Distribution")
    plt.tight_layout()
    plt.show()

def feature_importance(self):
    # compute the correlation matrix
    correlation_matrix = self.dataset.corr()
    # plot the heatmap
    plt.figure(figsize=(10, 8))
    sns.heatmap(correlation_matrix, annot=True,
        cmap="coolwarm", fmt=".2f")
    plt.title("Correlation Heatmap")

    # show the plot
    plt.show()
```

The result find out, there are no null value in the dataset.

According to the boxplot (fig 1), we find out there are some outliers in the dataset, and the distribution of features are various. So I decide to use the L2 normalization to to scale data while maintaining robustness against outliers. The Heatmap (fig 2)shows there are relative high correlation between target and other 12 feature, except for "ash" feature, which only get -0.05 correlation. However, considering the given dataset is very samll, so I kept this feature.

```python
def normalize(self, data_x):
    # Use L2 norm to normalize the data
    norm = np.linalg.norm(data_x, axis=0)
    data_x = data_x / norm
    return data_x
```

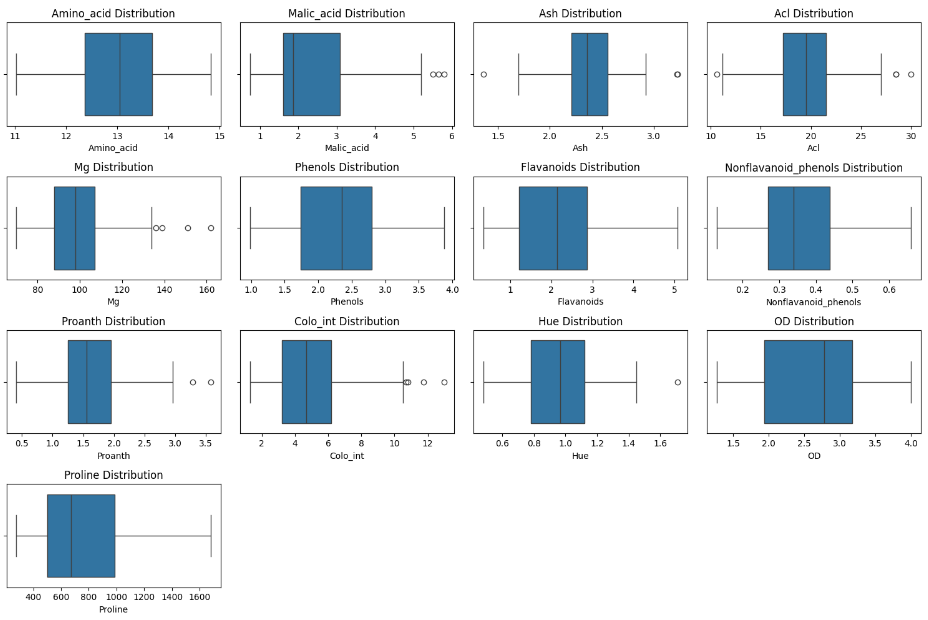


Figure 1: Box plot

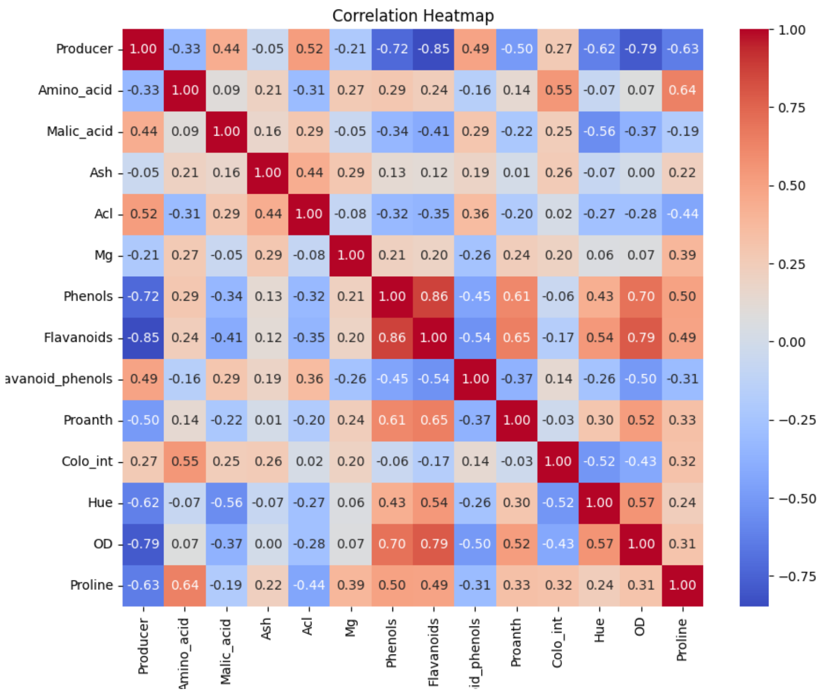


Figure 2: Correlation Heat Map

2 Model Construction

2.1 Loss function

In order to solve the task of multi-class classification, Softmax cross entropy loss was used to be the loss function in my neural network, by maximising the likelihood of the softmax regression. softmax cross entropy mathematical expression:

$$L(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} y_{i,k}\log(\hat{y}_{i,k}) \tag{1}$$

N number of sample
K number of class
$y_{i,k}$ real label that sample i in class k
$\hat{y}_{i,k})$ predict probability that sample i in class k

```
def get_loss(self, label):  #
        self.batch_size = self.output.shape[0]
        self.label = label
        self.label_onehot = np.zeros_like(self.output)
        label = label.flatten().astype(int)
        # print(self.label_onehot)
        # print(label)
        self.label_onehot[np.arange(self.batch_size), label-1]
        = 1.0 #onhot to represent the label(true value)
        entropy_loss = -np.sum(np.log(self.output)
        * self.label_onehot) / self.batch_size
        total_loss = entropy_loss + 0.01 * self.weights
        return total_loss
```

2.2 Network Design

In this task python and numpy were only used to bulid a 3-hidden layer network, which I think would be complicated enough for learning features from the data. Consider to avoid over-fitting and the features are not that complicated, the hidden layers size were structured as 16-32-8. In order to explain my network as clearly as I can, I drew some original handcrafts to visualize the network.

2.2.1 The Forward Pass

As shown in the figure3, I write down the matrix size of Weight and bias in each layer. The activation function Relu and Softmax are used in the forward propagation, which forward mathematical formula were written below.

The main reason why I choose Relu as activation function is it can avoid gradient decent compared with Sigmoid.The code below shows how to express the mathematical expression in python.
This is code snippets of forward pass:

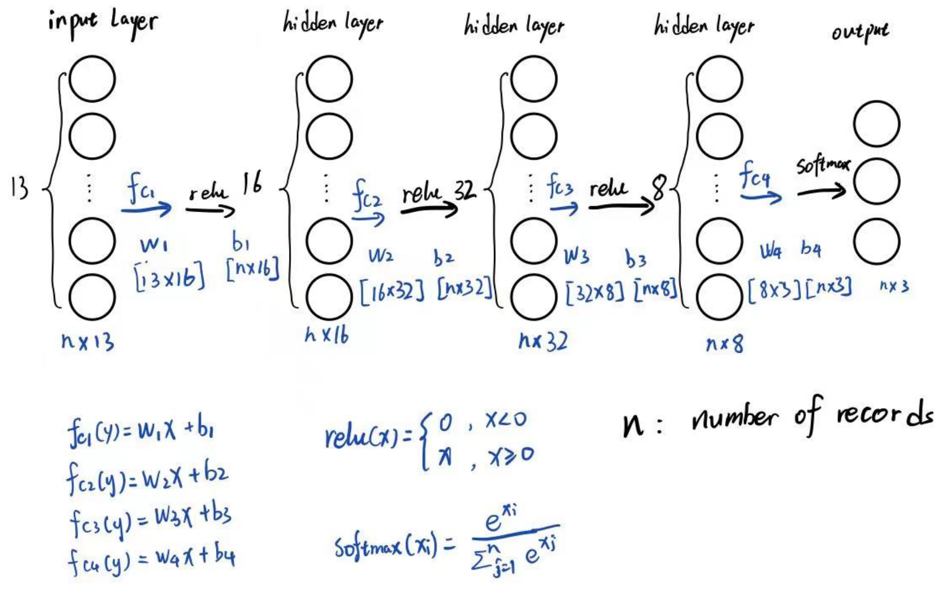


Figure 3: forward handcraft

```python
class FullyConnectedLayer:
    def forward(self,input):
        self.input = input
        self.output = np.matmul(input,self.weight) + self.bias
        return self.output

class ReLu:
    def forward(self,input):
        self.input = input
        self.output = np.maximum(0,input)
        return self.output

class Softmax:
    def forward(self, prob):
        self.output = np.exp(prob) /
        np.sum(np.exp(prob), axis=1, keepdims=True)
        return self.output

def forward(self, input):  # forward propagation
    h1 = self.fc1.forward(input)
    h1 = self.relu1.forward(h1)
    h2 = self.fc2.forward(h1)
    h2 = self.relu2.forward(h2)
    h3 = self.fc3.forward(h2)
    h3 = self.relu3.forward(h3)
    prob = self.fc4.forward(h3)
    prob = self.softmax.forward(prob)
    return prob
```

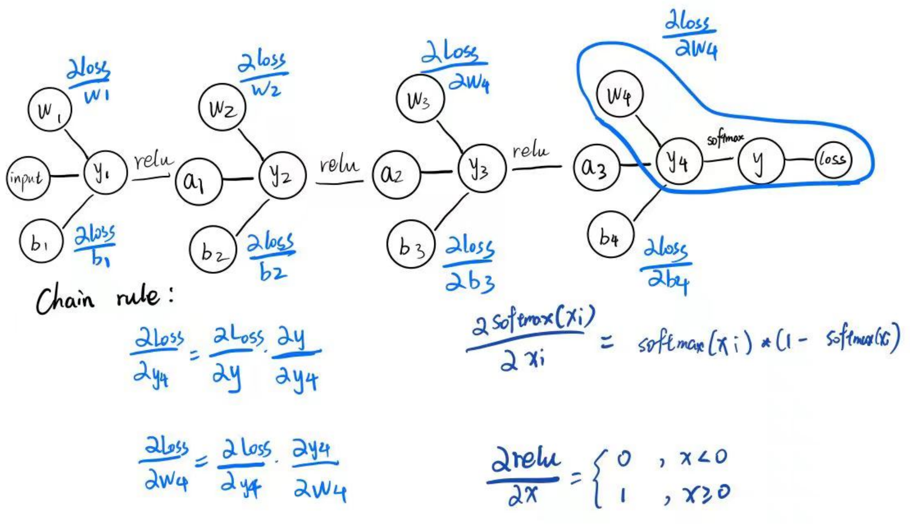

∂loss/w1
w1
input
y1
b1
∂loss/b1
relu
a1
w2
∂loss/w2
y2
b2
∂loss/b2
relu
a2
w3
∂loss/∂w4
y3
b3
∂loss/∂b3
relu
a3
∂loss/∂w4
w4
y4
softmax
y
loss
b4
∂loss/∂b4
Chain rule:
∂loss/∂y4 = ∂loss/∂y · ∂y/∂y4
∂loss/∂w4 = ∂loss/∂y4 · ∂y4/∂w4
∂softmax(xi)/∂xi = softmax(xi) * (1 - softmax(xi))
∂relu/∂x = { 0 , x<0 ; 1 , x≥0

Figure 4: Backward handcraft

2.2.2 The Backward Pass

In the backward pass, the error is propagated backward through the network to update the weights. As you can see in figure 4, the gradient of the error with respect to each weight and bias in the network are calculated using the chain rule of calculus. The figure also give an example to show, how the chain rule works.

The code below shows how to express the mathematical expression in python, some of them are already been simplify with the chain rule
This is code snippets of backward pass:

```
class FullyConnectedLayer:
    def backward(self,output_grad):
        self.d_weight = np.dot(self.input.T,output_grad) + 0.01 * 2
        * self.weight #dL/dw = X.T * dL/dy
        self.d_bias = np.sum(output_grad,axis=0)
        #dL/db = 1 + dL/dy
        self.d_input = np.dot(output_grad,self.weight.T)
        #dL/dX = dL/dy*W.T
        return self.d_input

class ReLu:
    def backward(self,output_grad):
        self.d_input = output_grad * (self.input > 0)
        return self.d_input


class Softmax:
    def backward(self):
        diff_softmax = (self.output - self.label_onehot)
        / self.batch_size #get deviation of output by chain rule
        return diff_softmax
```

2.3 Gradient Descent

Mini batch stochastic gradient descent is used to achieve this task. The dataset is divided into smaller subsets of fixed size (mini-batches). Each mini-batch contains a subset of the training data .For each mini-batch, the gradient of the loss function with respect to the model parameters is computed. Then the gradient of each layers parameters will be used to update the parameter.

backward function shows how the gradient pass and compute, `create_mini_batches`function I reference the code in the pratical and do shuffle in each epoch when create the mini batches.

The code snippets shows below:

```


# backward propagation to compute gradient
def backward(self):
    dloss = self.softmax.backward()
    dh4 = self.fc4.backward(dloss)
    dh3 = self.relu3.backward(dh4)
    dh3 = self.fc3.backward(dh3)
    dh2 = self.relu2.backward(dh3)
    dh2 = self.fc2.backward(dh2)
    dh1 = self.relu1.backward(dh2)
    dh1 = self.fc1.backward(dh1)
    return

class FullyConnectedLayer:
    # update the parameter of each layer
    self.update_layer_list = [h1,h2,h3,h4]

    def update(self, lr):
        for layer in self.update_layer_list:
            layer.update_param(lr)

    def update_param(self,lr):
        self.weight = self.weight - lr * self.d_weight
        self.bias = self.bias - lr * self.d_bias

class my_neural_network:
    def train(self):
        print('Start training...')_
        # reset train_losses and value_losses every train
        self.train_losses = []
        self.val_losses = []
        for epoch in range(self.max_epoch):
            batches = self.create_mini_batches(self.batch_size)
            #create batch
            for batch_x, batch_y in batches:
                self.forward(batch_x)
                train_loss = self.softmax.get_loss(batch_y)
                self.backward()
                self.update(self.lr)
```

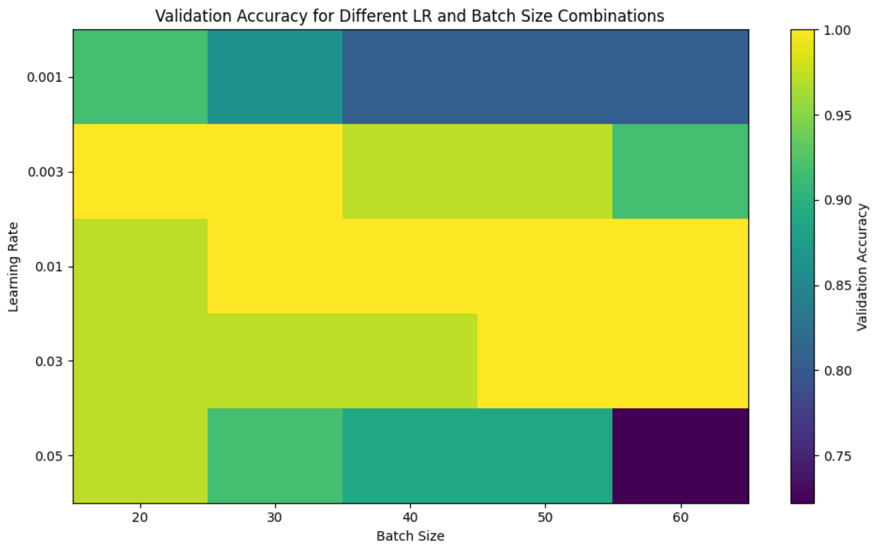


Figure 5: grid search

3 Model Training

As mentioned in 1.2 Preprocess the dataset previously, I seperate the 178 records into 6:2:2 size, which means the validation set have 35 records.

3.1 Model Training

With the aim of finding the proper hyper parameters, a grid search was used by observing the performance accuracy on validation set. According to the heat map shown in figure 5, validation accuracy performs well when hyper parameters are [lr:0.003, batchsize:20], [lr:0.003, batchsize:30], [lr:0.01, batchsize:20], [lr:0.01, batchsize:40], [lr:0.01, batchsize:50], [lr:0.01, batchsize:60],[lr:0.03, batchsize:50], [lr:0.01, batchsize:60].

3.2 Module Regularisation

In order to prevent overfitting by penalizing large weights in the model, L2 regularization term was imply to encourages the model to learn simpler patterns and prevents it from relying too heavily on any single feature, thus improving its generalization performance on unseen data.

$$L_{\mathrm{L2}} = \lambda \sum_{i=1}^{n} w_i^2$$

The code snippets shows below:

```
slef.weights = sum(sum(self.fc1.weight**2)) +
sum(sum(self.fc2.weight**2)) +
sum(sum(self.fc3.weight**2)) +
sum(sum(self.fc4.weight**2))
```

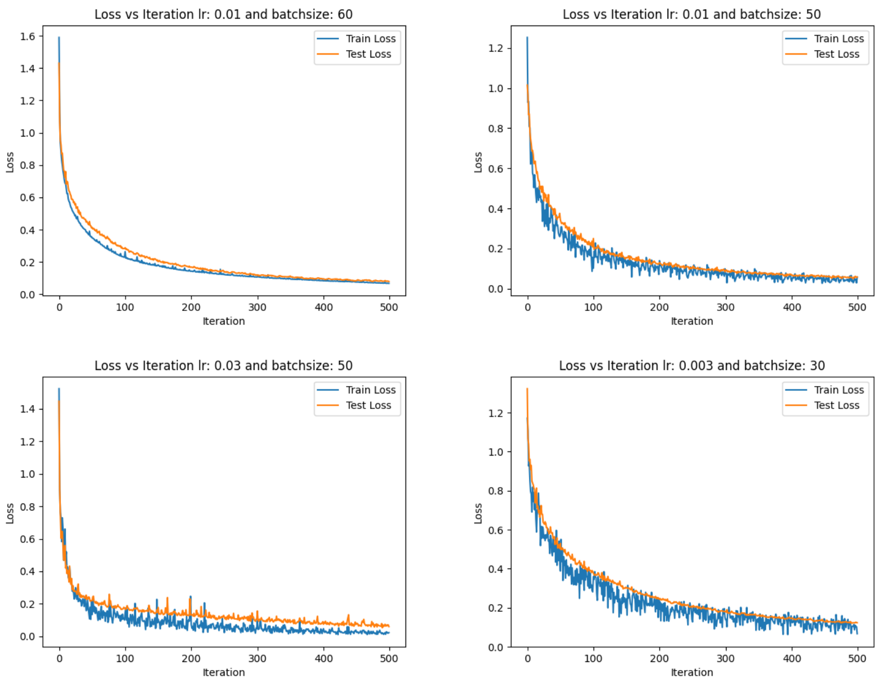
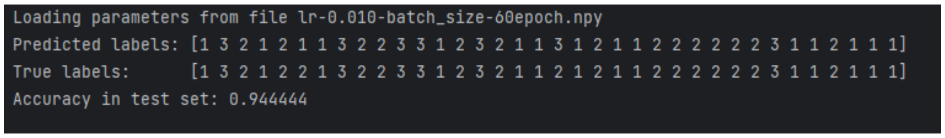


Figure 6: Top4: Train loss and validation loss

```
def get_loss(self, label):  #
        self.batch_size = self.output.shape[0]
        self.label = label
        self.label_onehot = np.zeros_like(self.output)
        label = label.flatten().astype(int)
        # print(self.label_onehot)
        # print(label)
        self.label_onehot[np.arange(self.batch_size), label-1]
        = 1.0 #onhot to represent the label(true value)
        entropy_loss = -np.sum(np.log(self.output)
        * self.label_onehot) / self.batch_size
        total_loss = entropy_loss + 0.01 * self.weights /
        regularization = 0.01 * self.weights / (2*self.batch_size)
        total_loss = entropy_loss + regularization
        return total_loss
```

3.3 Model inference

Among the well perform hyper parameter pair,s I choose the top 4 both well perform and large batch, considering the larger batch is, the quicker the loss converge. Base on the figure 6 shows, I choose [lr : 0.01, batchsize: 60] as my final hyper parameter.



```
Loading parameters from file lr-0.010-batch_size-60epoch.npy
Predicted labels: [1 3 2 1 2 1 1 3 2 2 3 3 1 2 3 2 1 1 3 1 2 1 1 2 2 2 2 2 2 3 1 1 2 1 1 1]
True labels:      [1 3 2 1 2 2 1 3 2 2 3 3 1 2 3 2 1 1 2 1 2 1 1 2 2 2 2 2 2 3 1 1 2 1 1 1]
Accuracy in test set: 0.944444
```

Figure 7: performance on test set

4 Evaluation

4.1 Present Results

By using the well trained model as shown in figure 7 and figure 8, our model predict the data in the test set, and get a 94.444 percent accuracy.

```python
def evaluate(self,data):
        prob = self.forward(data[:,:-1])
        pred_labels = np.argmax(prob, axis=1)
        accuracy = np.mean(pred_labels+1 == data[:, -1])
        print("Predicted labels:",  pred_labels+1)
        print("True labels:     ",  data[:, -1].astype(int))
        print('Accuracy in test set: %f' % accuracy)
        return accuracy


dp = Data_preprocessing()
train_data, test_data, val_data = dp.process()
mlp = run_model(0.01, 60)
accuracy = mlp.evaluate(test_data)  # get the validation accuracy
```

4.2 PLot

The image at the upper side of figure 9, the fluctuant structure both validation loss and train loss represent the unnormal gradient decent. This may because the hyper parameters is unfit for the model, the learning rate is too high in this case. The well perform structure of loss decreation is the image at the left bottom in the figure 9.

As we can see the right bottom image in figure 9, the validation loss is consistently lower than training loss, this may because overfitting to the training data. Overfitting occurs when the model learns to memorize the training data rather than generalize to new, unseen data. In this case, the model performs well on the training set but poorly on the validation set.

The addition of regularisation do help mitigate the overfitting , as the figure 10 shows , the validation loss more close to train loss and become converge after adding regularisation.

5 Explain Results

In general, by choosing a relatively optimal combination of hyper parameters for my network model, and made the training loss converge as early as possible. The accuracy of the final prediction result with the trained model reached 94.44 percent.

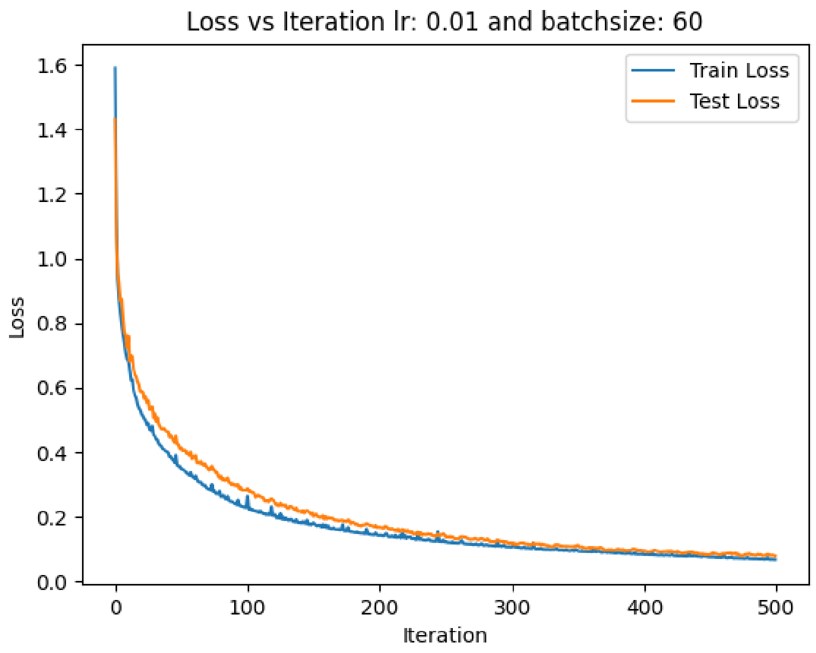


Figure 8: Loss of the final Model I choose

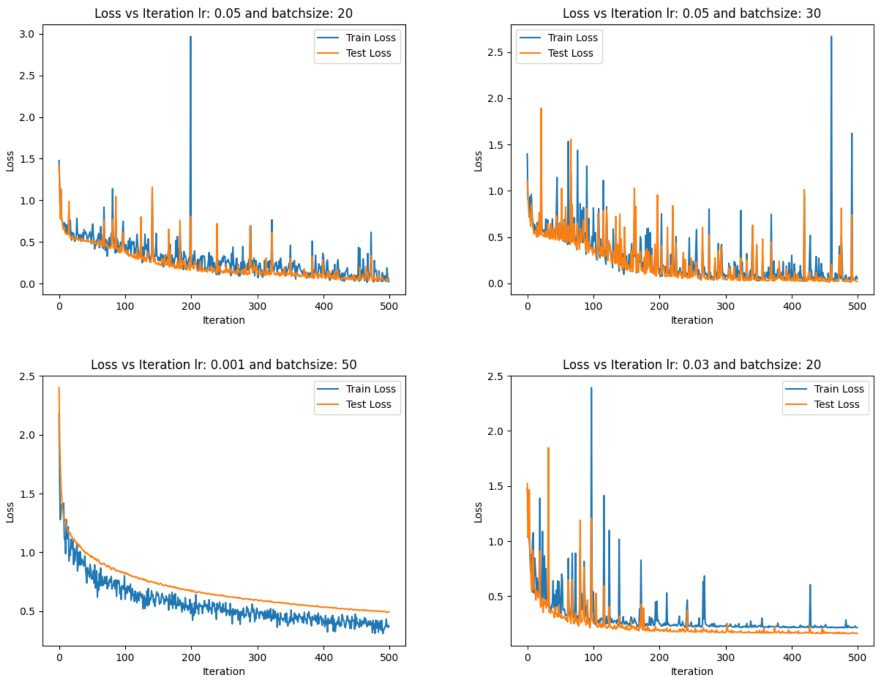


Figure 9: Other hyper parameters pairs performance

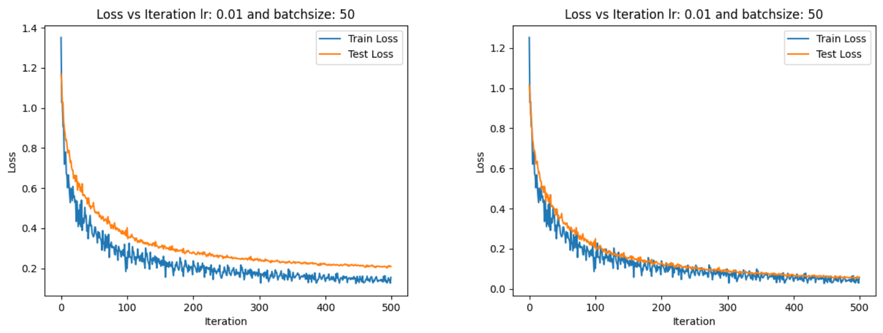


Figure 10: left one is before regularisation, right one is after regularisation

However, there are some things that can be improved in the experiment, for example, I did not use the optimizer adam to optimize the model. The data set is too small, which may affect the final effect of the model. The design of neural networks of different sizes also needs experimental research.