Image Classification Summary Report

Jue Wang, Keyi Yu

Introduction

In this project, we focused on exploring various deep learning models & techniques on the image classification task. Concretely, we implement and conduct experiments on both the Multilayer Perceptron model and CNN model. The dataset we use is MNIST, which is a very commonly used one. We achieved comparable results to the best we know on this dataset. We implemented with Pytorch. The rest of the report is organized as follows. In the second section, we summarize knowledge points learned in this project, covering both deep learning and computer vision related topics. And then we briefly describe the code we finally delivered in section three, and how we do parameter tuning in section four. Finally is the conclusion of the whole project.

Technical Knowledge Summary

- Model training & evaluation
  Model training means, based on the machine learning algorithm, building a model by examining many examples and trying to find the most proper model which has the minimum loss.
- Loss function
  The loss function is used to evaluate the difference between the predicted value and the real value of the model. The better the loss function is, the better the performance model will be.
- Activation
  To apply a function to another function, such as to bring a linear function into a nonlinear function.
- Learning rate
  Learning rate is a parameter changed in response to the estimated error time when the model weights are updated.
- Batch based training
  Batch size is used to determine the number of examples to work through.
- Optimization
  Optimization is used to minimize the loss function for deeping learning.
- Overfitting
  When a model fits more data than it actually needs, it will catch some inaccuracy which will decrease the efficiency and the accuracy of the model.

- Multilayer Perceptron
  The input layer inputs the information of a picture, and passes the input information to the output layer through the hidden layer. After a series of operations, the output layer gets the desired result.
- CNN
  By constantly changing the length, width and height of the image (converting the handwriting image into a pixel matrix), the important information of the image is filtered again and again, and finally the image is classified.
- MNIST
  A large database with many handwritten digits.
- Pytorch
  An open source machine learning library.

Project Delivery

The file “image_classification _mlp.py” trains the Multilayer Perceptron (MLP) model on MNIST.
This repository is an MLP implementation of classifier on MNIST dataset with PyTorch.
The processes of my code can be broken down into following steps:

1. Define the Network Architecture
2. Define the training & test function for the Network
3. Load and Visualize the data
4. End2end training and testing pipeline

The file “image_classification _cnn.py” trains the Convolution Neural Network (CNN).
Similarly, this repository is an CNN implementation of classifier on MNIST dataset with PyTorch.
The processes of my code (same as previous above) can be broken down into following steps:

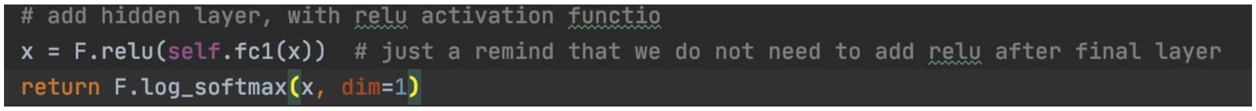
1. Define the Network Architecture
2. Define the training function and the test function
3. Load and Visualize the data
4. End2end training and testing pipeline

Practical Experience in Parameter Tuning

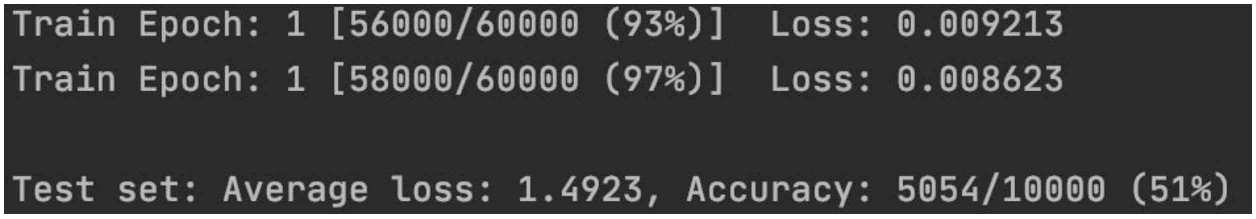
After finishing the code, I also tried to adjust the parameters in the code to find out how these parameters influence the final accuracy, here I select some practical Experience to illustrate:

relu

Before removing the relu after the final layer the accuracy is around 50%

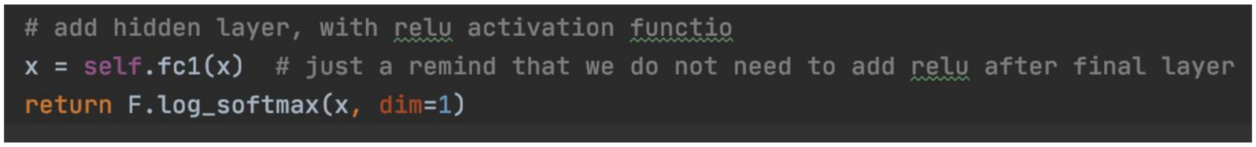


```
# add hidden layer, with relu activation functio
x = F.relu(self.fc1(x))  # just a remind that we do not need to add relu after final layer
return F.log_softmax(x, dim=1)
```

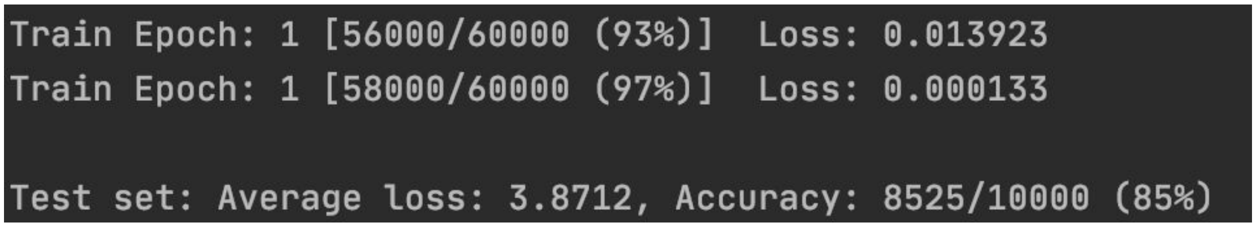


```
Train Epoch: 1 [56000/60000 (93%)]  Loss: 0.009213
Train Epoch: 1 [58000/60000 (97%)]  Loss: 0.008623

Test set: Average loss: 1.4923, Accuracy: 5054/10000 (51%)
```

After removing the relu after the final layer the accuracy is around 86%



```
# add hidden layer, with relu activation functio
x = self.fc1(x)  # just a remind that we do not need to add relu after final layer
return F.log_softmax(x, dim=1)
```

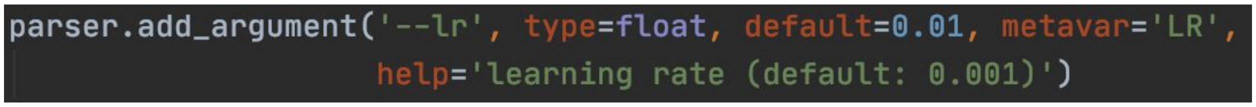


```
Train Epoch: 1 [56000/60000 (93%)]  Loss: 0.013923
Train Epoch: 1 [58000/60000 (97%)]  Loss: 0.000133

Test set: Average loss: 3.8712, Accuracy: 8525/10000 (85%)
```

Another lesson is we don't need to add relu in the final layer.

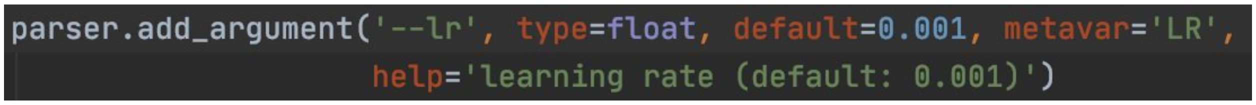
Learning rate

Learning rate changes from 0.1 to 0.01 the accuracy is around 88%

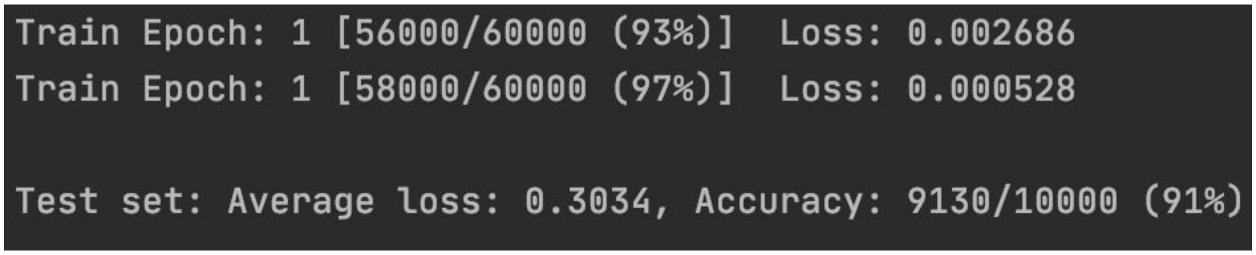


```
parser.add_argument('--lr', type=float, default=0.01, metavar='LR',
                    help='learning rate (default: 0.001)')
```

Learning rate changes from 0.01 to 0.001 the accuracy is around 92%



```
parser.add_argument('--lr', type=float, default=0.001, metavar='LR',
                    help='learning rate (default: 0.001)')
```

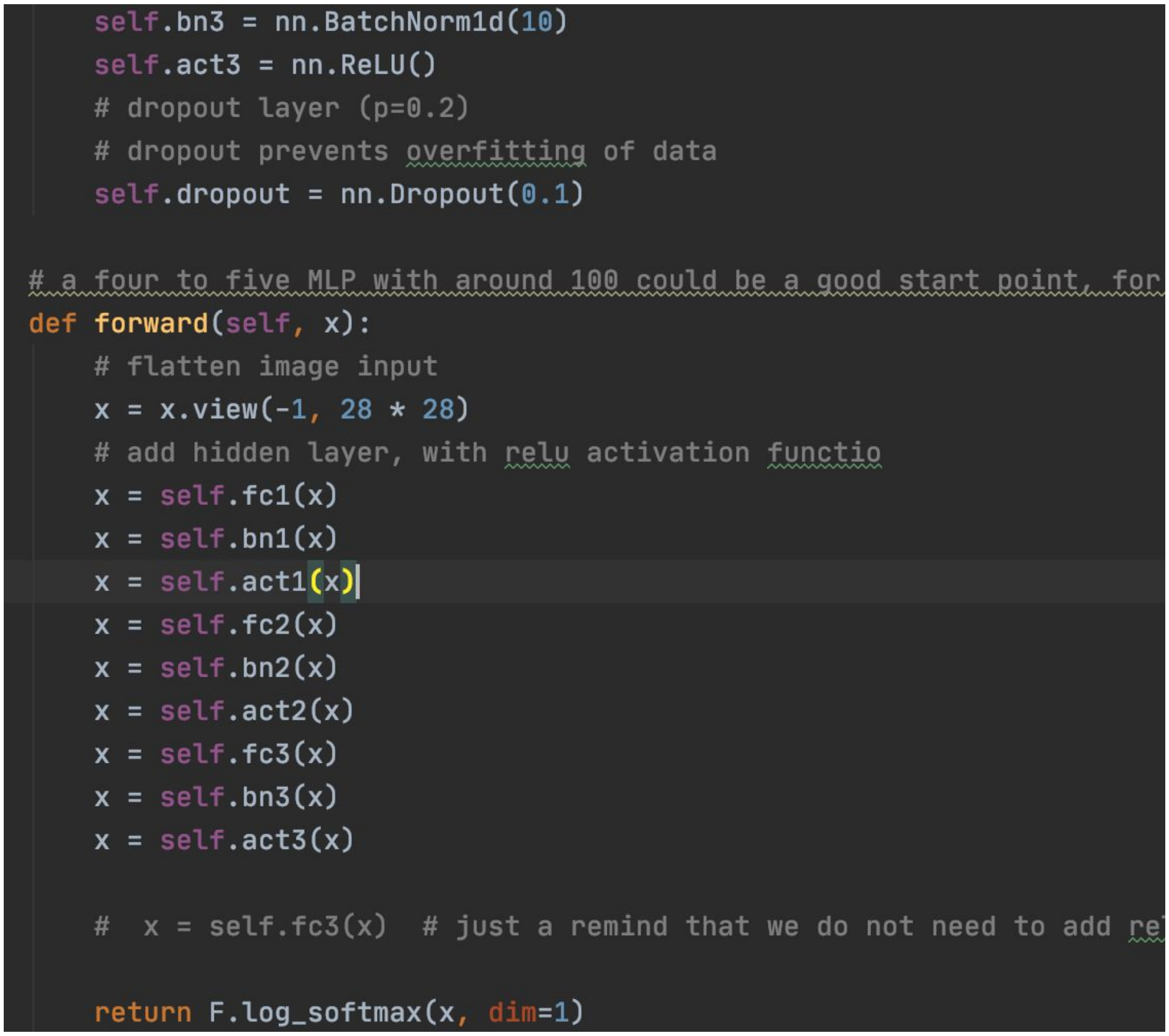


```
Train Epoch: 1 [56000/60000 (93%)]  Loss: 0.002686
Train Epoch: 1 [58000/60000 (97%)]  Loss: 0.000528

Test set: Average loss: 0.3034, Accuracy: 9130/10000 (91%)
```

Actually, Adam should use 0.001 as the initial learning rate.

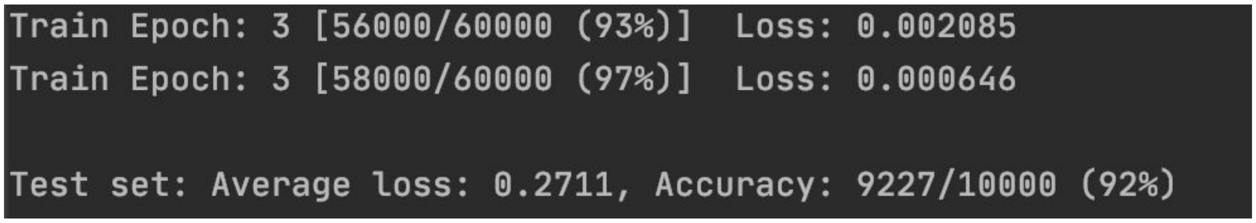
Dropout

Dropout is used to prevent the overfitting of data. We change the dropout from 0.5 to 0.1 after adding more than one layer. The initial dropout is 0.1.



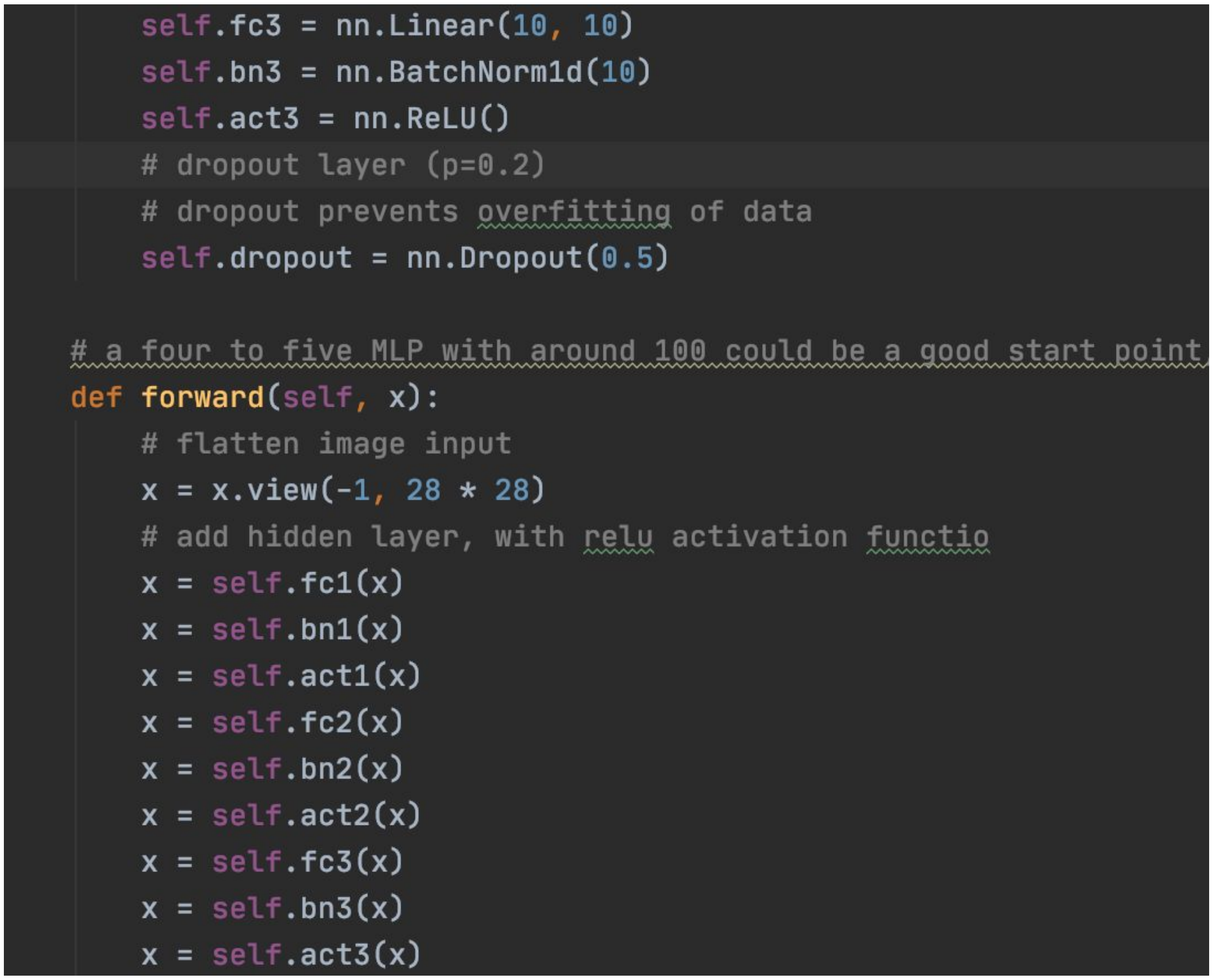
```
        self.bn3 = nn.BatchNorm1d(10)
        self.act3 = nn.ReLU()
        # dropout layer (p=0.2)
        # dropout prevents overfitting of data
        self.dropout = nn.Dropout(0.1)

    # a four to five MLP with around 100 could be a good start point, for
    def forward(self, x):
        # flatten image input
        x = x.view(-1, 28 * 28)
        # add hidden layer, with relu activation functio
        x = self.fc1(x)
        x = self.bn1(x)
        x = self.act1(x)
        x = self.fc2(x)
        x = self.bn2(x)
        x = self.act2(x)
        x = self.fc3(x)
        x = self.bn3(x)
        x = self.act3(x)

        #  x = self.fc3(x)  # just a remind that we do not need to add re

        return F.log_softmax(x, dim=1)
```

The accuracy is around 92%



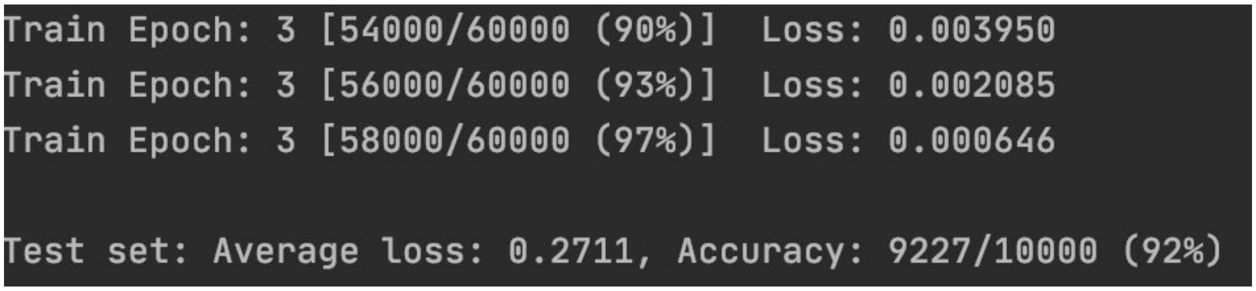
```
Train Epoch: 3 [56000/60000 (93%)]  Loss: 0.002085
Train Epoch: 3 [58000/60000 (97%)]  Loss: 0.000646

Test set: Average loss: 0.2711, Accuracy: 9227/10000 (92%)
```

After changing the dropout into 0.5



```
        self.fc3 = nn.Linear(10, 10)
        self.bn3 = nn.BatchNorm1d(10)
        self.act3 = nn.ReLU()
        # dropout layer (p=0.2)
        # dropout prevents overfitting of data
        self.dropout = nn.Dropout(0.5)

    # a four to five MLP with around 100 could be a good start point
    def forward(self, x):
        # flatten image input
        x = x.view(-1, 28 * 28)
        # add hidden layer, with relu activation functio
        x = self.fc1(x)
        x = self.bn1(x)
        x = self.act1(x)
        x = self.fc2(x)
        x = self.bn2(x)
        x = self.act2(x)
        x = self.fc3(x)
        x = self.bn3(x)
        x = self.act3(x)
```

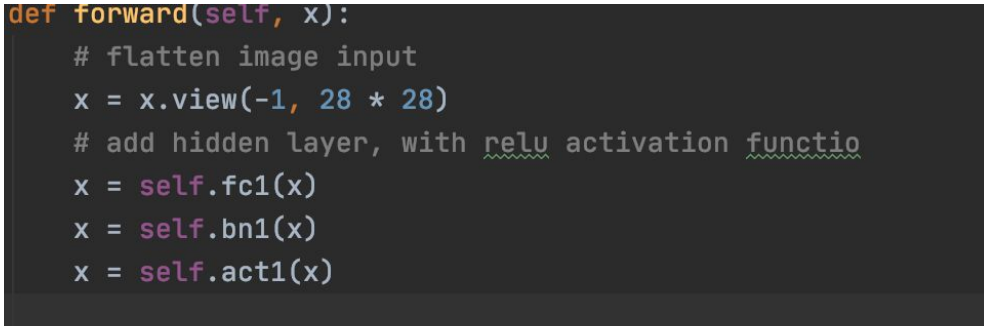
The accuracy with around 92%



```
Train Epoch: 3 [54000/60000 (90%)]  Loss: 0.003950
Train Epoch: 3 [56000/60000 (93%)]  Loss: 0.002085
Train Epoch: 3 [58000/60000 (97%)]  Loss: 0.000646

Test set: Average loss: 0.2711, Accuracy: 9227/10000 (92%)
```

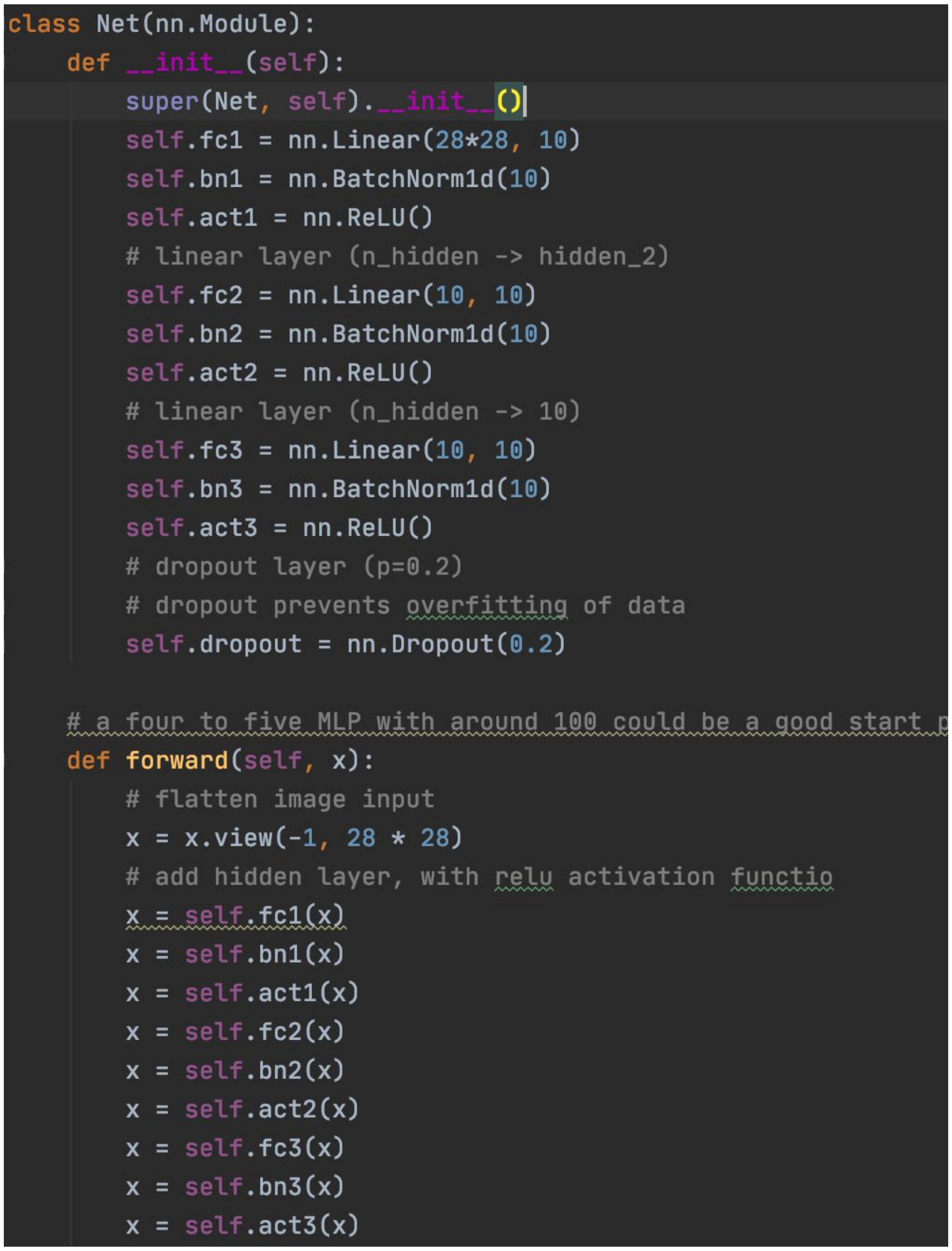
Layers

The number of layer is changed from 1 to 3, here is the result for one layer.



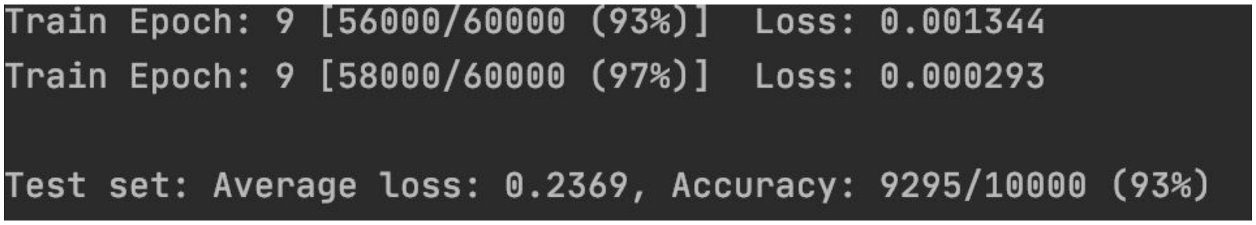
```
def forward(self, x):
    # flatten image input
    x = x.view(-1, 28 * 28)
    # add hidden layer, with relu activation functio
    x = self.fc1(x)
    x = self.bn1(x)
    x = self.act1(x)
```

Three layers



```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(28*28, 10)
        self.bn1 = nn.BatchNorm1d(10)
        self.act1 = nn.ReLU()
        # linear layer (n_hidden -> hidden_2)
        self.fc2 = nn.Linear(10, 10)
        self.bn2 = nn.BatchNorm1d(10)
        self.act2 = nn.ReLU()
        # linear layer (n_hidden -> 10)
        self.fc3 = nn.Linear(10, 10)
        self.bn3 = nn.BatchNorm1d(10)
        self.act3 = nn.ReLU()
        # dropout layer (p=0.2)
        # dropout prevents overfitting of data
        self.dropout = nn.Dropout(0.2)

    # a four to five MLP with around 100 could be a good start p
    def forward(self, x):
        # flatten image input
        x = x.view(-1, 28 * 28)
        # add hidden layer, with relu activation functio
        x = self.fc1(x)
        x = self.bn1(x)
        x = self.act1(x)
        x = self.fc2(x)
        x = self.bn2(x)
        x = self.act2(x)
        x = self.fc3(x)
        x = self.bn3(x)
        x = self.act3(x)
```

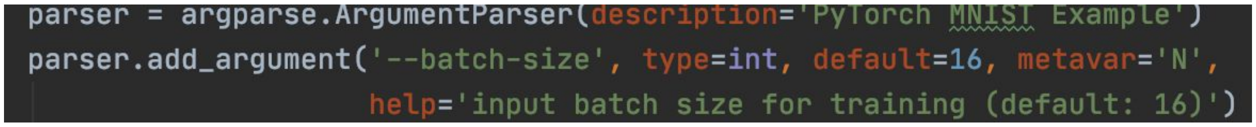
The accuracy is around 93%



```
Train Epoch: 9 [56000/60000 (93%)]  Loss: 0.001344
Train Epoch: 9 [58000/60000 (97%)]  Loss: 0.000293

Test set: Average loss: 0.2369, Accuracy: 9295/10000 (93%)
```

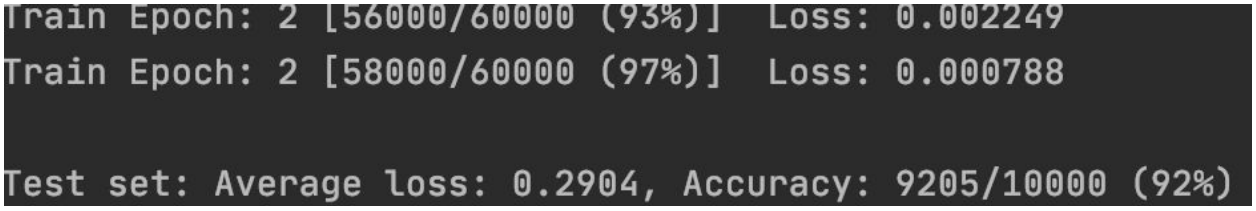
Batch size

Initial batch size is 16



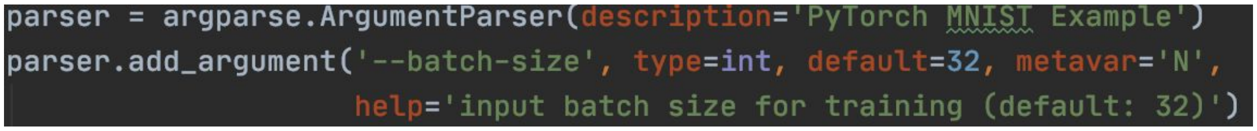
```
parser = argparse.ArgumentParser(description='PyTorch MNIST Example')
parser.add_argument('--batch-size', type=int, default=16, metavar='N',
                    help='input batch size for training (default: 16)')
```

The accuracy is around 92%



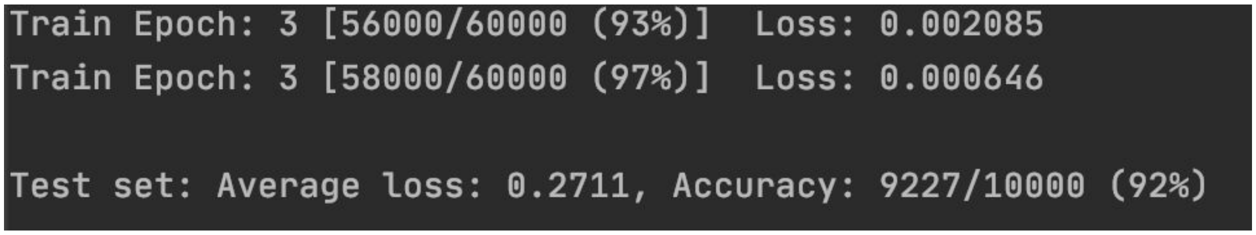
```
Train Epoch: 2 [56000/60000 (93%)]  Loss: 0.002249
Train Epoch: 2 [58000/60000 (97%)]  Loss: 0.000788

Test set: Average loss: 0.2904, Accuracy: 9205/10000 (92%)
```

Batch size changes from 16 to 32



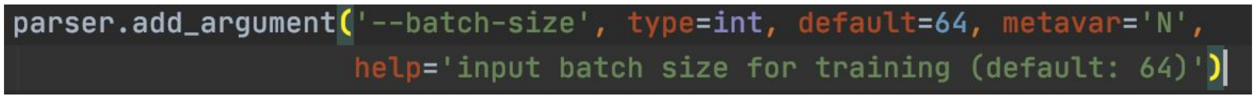
```
parser = argparse.ArgumentParser(description='PyTorch MNIST Example')
parser.add_argument('--batch-size', type=int, default=32, metavar='N',
                    help='input batch size for training (default: 32)')
```

The accuracy us around 92%



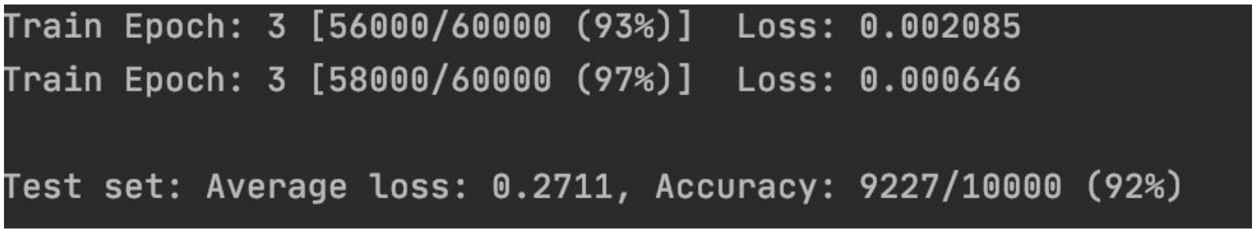
```
Train Epoch: 3 [56000/60000 (93%)]  Loss: 0.002085
Train Epoch: 3 [58000/60000 (97%)]  Loss: 0.000646

Test set: Average loss: 0.2711, Accuracy: 9227/10000 (92%)
```

Batch size changes from 32 to 64



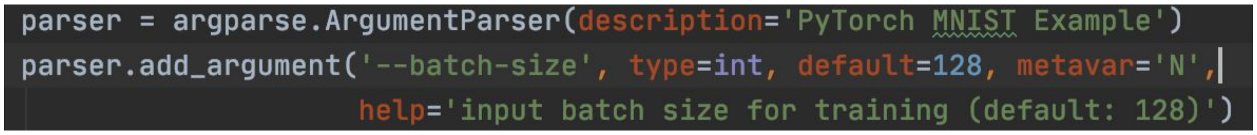
```
parser.add_argument('--batch-size', type=int, default=64, metavar='N',
                    help='input batch size for training (default: 64)')
```

The accuracy is 92%



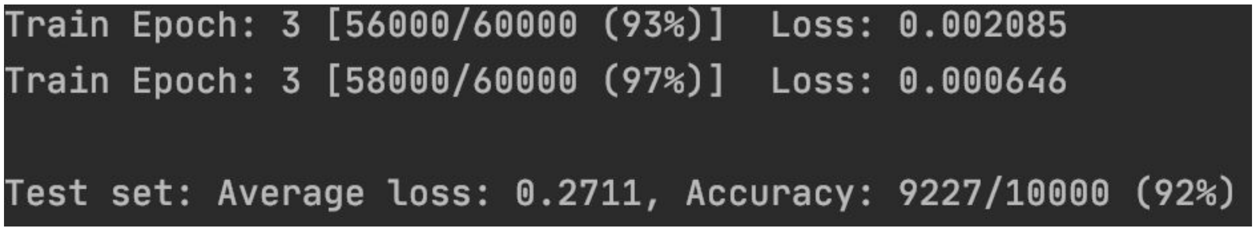
```
Train Epoch: 3 [56000/60000 (93%)]   Loss: 0.002085
Train Epoch: 3 [58000/60000 (97%)]   Loss: 0.000646

Test set: Average loss: 0.2711, Accuracy: 9227/10000 (92%)
```

Batch size changes from 64 to 128



```
parser = argparse.ArgumentParser(description='PyTorch MNIST Example')
parser.add_argument('--batch-size', type=int, default=128, metavar='N',
                    help='input batch size for training (default: 128)')
```

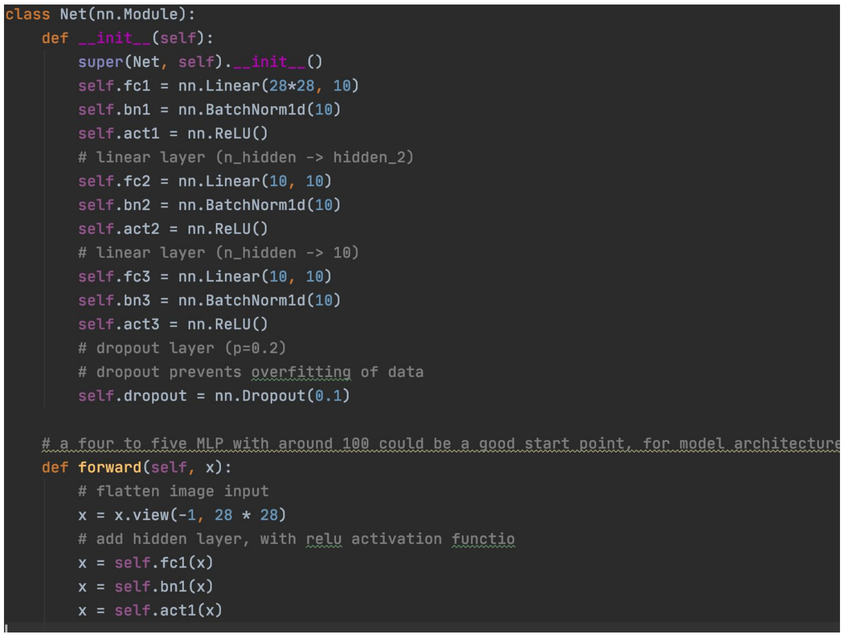
The accuracy is 92%



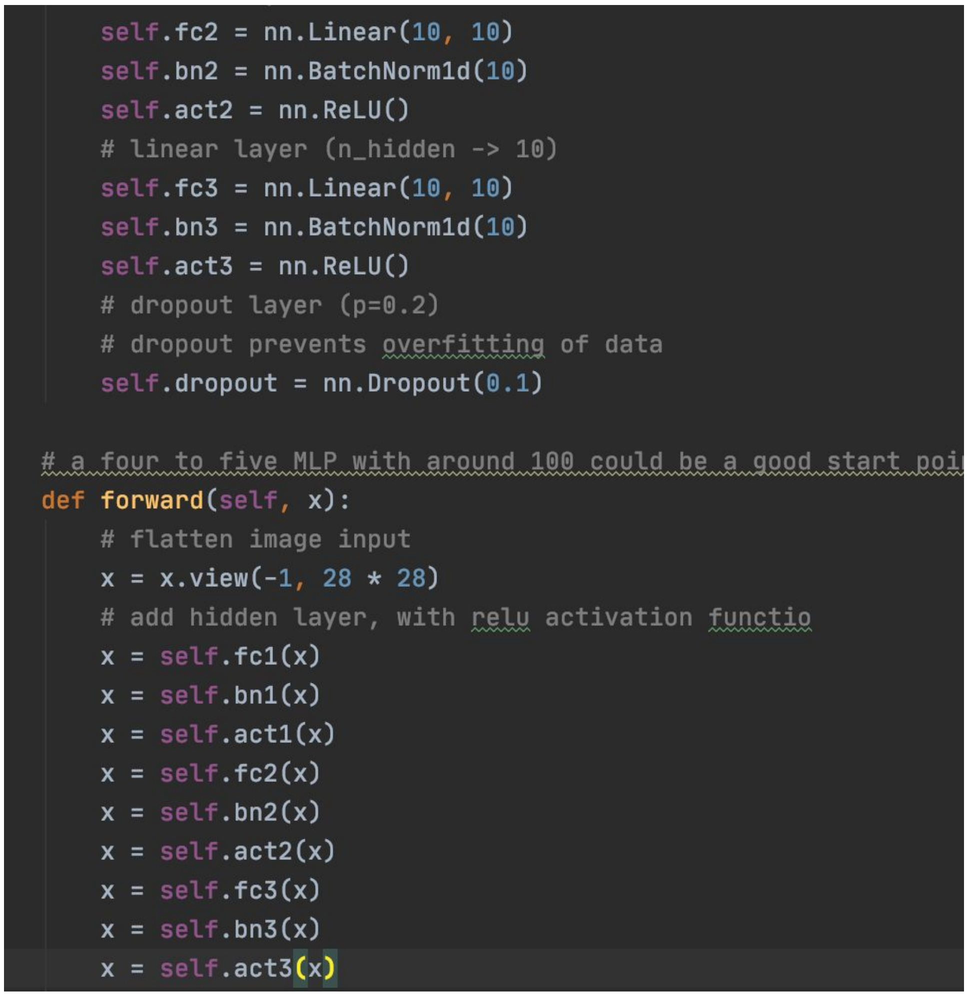
```
Train Epoch: 3 [56000/60000 (93%)]   Loss: 0.002085
Train Epoch: 3 [58000/60000 (97%)]   Loss: 0.000646

Test set: Average loss: 0.2711, Accuracy: 9227/10000 (92%)
```

It seems like the batch size has little influence on the accuracy, however, the batch size is the last parameter I adjusted, the accuracy can't change much anymore. In other problems, it sometimes also matters.

Change the Net

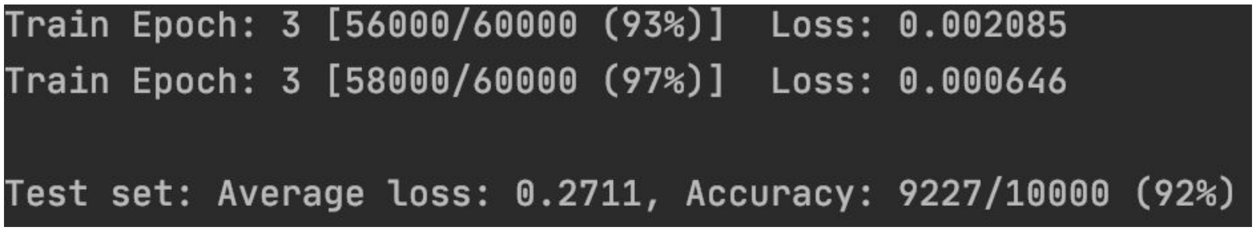


```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(28*28, 10)
        self.bn1 = nn.BatchNorm1d(10)
        self.act1 = nn.ReLU()
        # linear layer (n_hidden -> hidden_2)
        self.fc2 = nn.Linear(10, 10)
        self.bn2 = nn.BatchNorm1d(10)
        self.act2 = nn.ReLU()
        # linear layer (n_hidden -> 10)
        self.fc3 = nn.Linear(10, 10)
        self.bn3 = nn.BatchNorm1d(10)
        self.act3 = nn.ReLU()
        # dropout layer (p=0.2)
        # dropout prevents overfitting of data
        self.dropout = nn.Dropout(0.1)

    # a four to five MLP with around 100 could be a good start point, for model architecture
    def forward(self, x):
        # flatten image input
        x = x.view(-1, 28 * 28)
        # add hidden layer, with relu activation functio
        x = self.fc1(x)
        x = self.bn1(x)
        x = self.act1(x)
```



```
        self.fc2 = nn.Linear(10, 10)
        self.bn2 = nn.BatchNorm1d(10)
        self.act2 = nn.ReLU()
        # linear layer (n_hidden -> 10)
        self.fc3 = nn.Linear(10, 10)
        self.bn3 = nn.BatchNorm1d(10)
        self.act3 = nn.ReLU()
        # dropout layer (p=0.2)
        # dropout prevents overfitting of data
        self.dropout = nn.Dropout(0.1)

    # a four to five MLP with around 100 could be a good start poi
    def forward(self, x):
        # flatten image input
        x = x.view(-1, 28 * 28)
        # add hidden layer, with relu activation functio
        x = self.fc1(x)
        x = self.bn1(x)
        x = self.act1(x)
        x = self.fc2(x)
        x = self.bn2(x)
        x = self.act2(x)
        x = self.fc3(x)
        x = self.bn3(x)
        x = self.act3(x)
```

The accuracy is around 92%



```
Train Epoch: 3 [56000/60000 (93%)]  Loss: 0.002085
Train Epoch: 3 [58000/60000 (97%)]  Loss: 0.000646

Test set: Average loss: 0.2711, Accuracy: 9227/10000 (92%)
```

Conclusion

This is the first time for me to get close contact with deep learning, after learning the Multilayer Perceptron (MLP) model on MNIST and the Convolution Neural Network(CNN) model. I have been more familiar with the basic deep learning knowledge which is a good start for me to continue learning. I hope I could learn more models in the future.