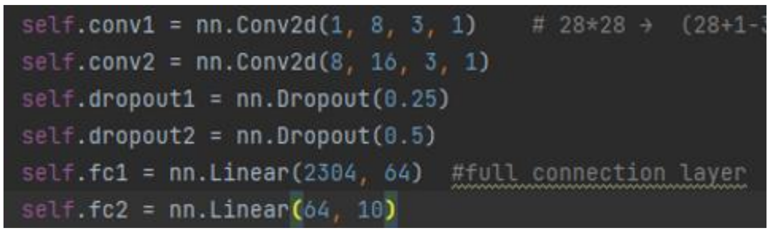
Task 1:

Convolution kernel:

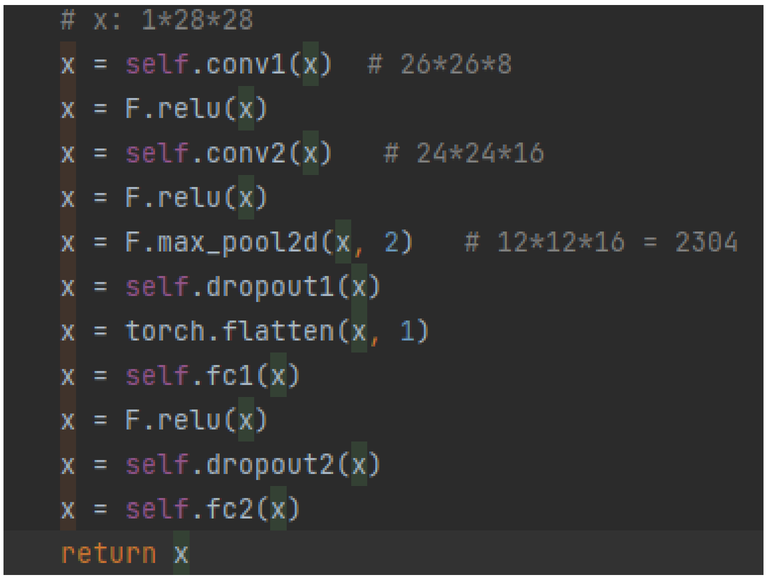
Convolution uses a ‘kernel’ to extract certain ‘features’ from an input image. A kernel is a matrix of weights which are multiplied with the input to extract relevant features. The dimensions of the kernel matrix are how the convolution gets its name. The output dimension calculation outcome is $O = \frac{N-F+2P}{S} + 1$, (O is output matrix, N is input matrix, F is convolutional kernel size, P is padding, and S is strides).

For instance, for the 1st convolutional layer in the framework, the input matrix (N) is 28*28*1, F is 3, P is 0, and S is 1. Computing by substitution, the output (O) is $\frac{(28-3)}{1} + 1 = 26$, and the matrix is 26*26*8.

Then, for the 2nd convolutional layer in the framework, the output (O) is $\frac{(26-3)}{1} + 1 = 24$. The output through conv2 is 24*24*16. After max pooling, the output is 12*12*16. Thus, the input of the full connection layer is 12*12*16, that is, “self.fc1 = nn.Linear(2304, 64)”.



```
self.conv1 = nn.Conv2d(1, 8, 3, 1)    # 28*28 → (28+1-
self.conv2 = nn.Conv2d(8, 16, 3, 1)
self.dropout1 = nn.Dropout(0.25)
self.dropout2 = nn.Dropout(0.5)
self.fc1 = nn.Linear(2304, 64)  #full connection layer
self.fc2 = nn.Linear(64, 10)
```



```
    # x: 1*28*28
    x = self.conv1(x)  # 26*26*8
    x = F.relu(x)
    x = self.conv2(x)   # 24*24*16
    x = F.relu(x)
    x = F.max_pool2d(x, 2)   # 12*12*16 = 2304
    x = self.dropout1(x)
    x = torch.flatten(x, 1)
    x = self.fc1(x)
    x = F.relu(x)
    x = self.dropout2(x)
    x = self.fc2(x)
    return x
```

Loss function:

In mathematical optimization and decision theory, a loss function is a function that maps an event or values of one or more variables onto a real number intuitively representing some "cost" associated with the event. In convolutional neural networks, the loss functions are used to compute the deviation between the output and the label, and then used in the back-propagation process to update the gradient. Through constant training and optimizing the parameters in CNN, the object is to minimize the loss.

The loss functions can be divided into classification loss functions (e.g., cross-entropy or log loss) and regression loss functions. The model in this framework calls F.cross_entropy(output, target) to calculate the cross-entropy loss between inputs and outputs, where the program performs softmax ($Softmax_i = \frac{\mathrm{e}^i}{\sum_j e^j}$, i is the element in the probability distribution) and log operations on the predicted value, and then computes the cross-entropy between the output and the target. The formula of cross-entropy is $-\sum_{i=1}^{n} \mathrm{p}(x_i) \ln{(q(x_i))}$, $p\ (x_i)$ is the real probability distribution and $q\ (x_i)$ is the predicted probability distribution. By reducing the difference between these two probability distributions, the cross-entropy function makes the predicted probability distribution close to the real probability distribution as much as possible.

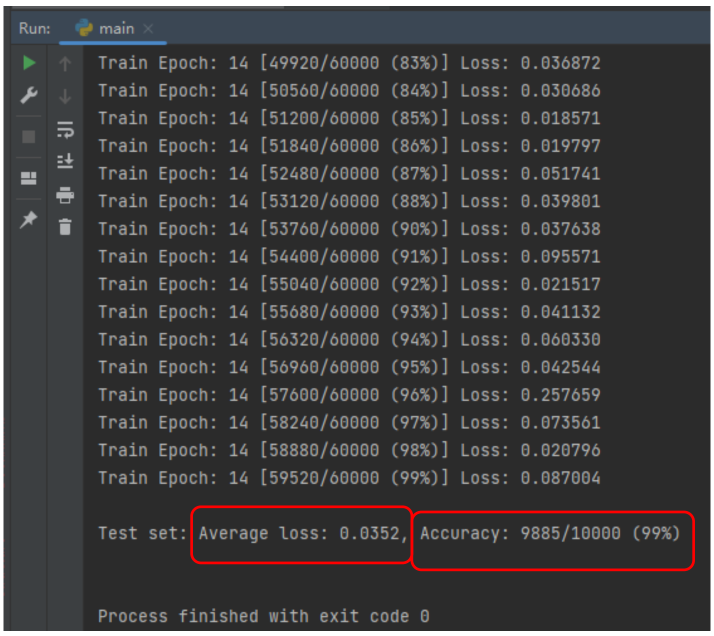
Moreover, the regression loss function aims to measure the degree of inconsistency with residuals. Least absolute deviations $\sum_i^n |Y_i - f(x_i)|$, and least square errors $\sum_i^n (Y_i - f(x_i))^2$, Y_i is the target and $f(x_i)$ is the prediction.

Task 2:

Final accuracy performance:

According to the default setting, the train datasets were trained for 14 epochs. The final accuracy performance is 99% (9885/10000). And the average loss was 0.0352.

The screenshot is showed as below.

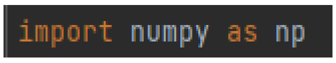


```
Train Epoch: 14 [49920/60000 (83%)] Loss: 0.036872
Train Epoch: 14 [50560/60000 (84%)] Loss: 0.030686
Train Epoch: 14 [51200/60000 (85%)] Loss: 0.018571
Train Epoch: 14 [51840/60000 (86%)] Loss: 0.019797
Train Epoch: 14 [52480/60000 (87%)] Loss: 0.051741
Train Epoch: 14 [53120/60000 (88%)] Loss: 0.039801
Train Epoch: 14 [53760/60000 (90%)] Loss: 0.037638
Train Epoch: 14 [54400/60000 (91%)] Loss: 0.095571
Train Epoch: 14 [55040/60000 (92%)] Loss: 0.021517
Train Epoch: 14 [55680/60000 (93%)] Loss: 0.041132
Train Epoch: 14 [56320/60000 (94%)] Loss: 0.060330
Train Epoch: 14 [56960/60000 (95%)] Loss: 0.042544
Train Epoch: 14 [57600/60000 (96%)] Loss: 0.257659
Train Epoch: 14 [58240/60000 (97%)] Loss: 0.073561
Train Epoch: 14 [58880/60000 (98%)] Loss: 0.020796
Train Epoch: 14 [59520/60000 (99%)] Loss: 0.087004

Test set: Average loss: 0.0352, Accuracy: 9885/10000 (99%)


Process finished with exit code 0
```

Modified code:

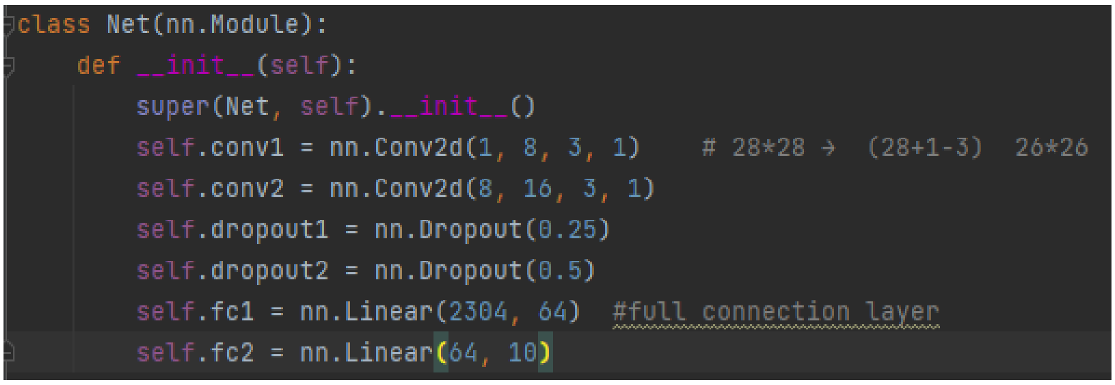
For all codes, you can feel free to refer to the **zip** file.

I import *numpy* as np to visualize the value in the beginning of the source code.



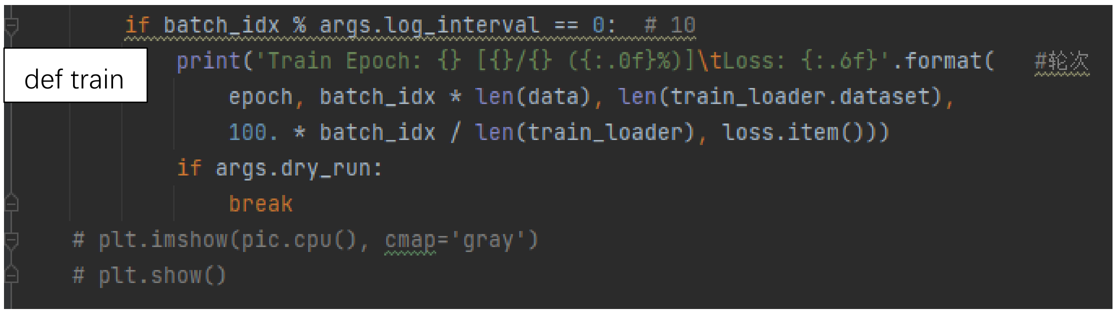
```
import numpy as np
```

For the "class Net" part, I didn't change the convolution kernel matrix value because I have tested it would decrease if I changed, even I have seen some teaching videos that they are using other number such as "self.conv1d = nn.Conv2d(1, 20, 5, 1)" and self.conv2d = nn.Conv2d(20, 50, 5, 1)"



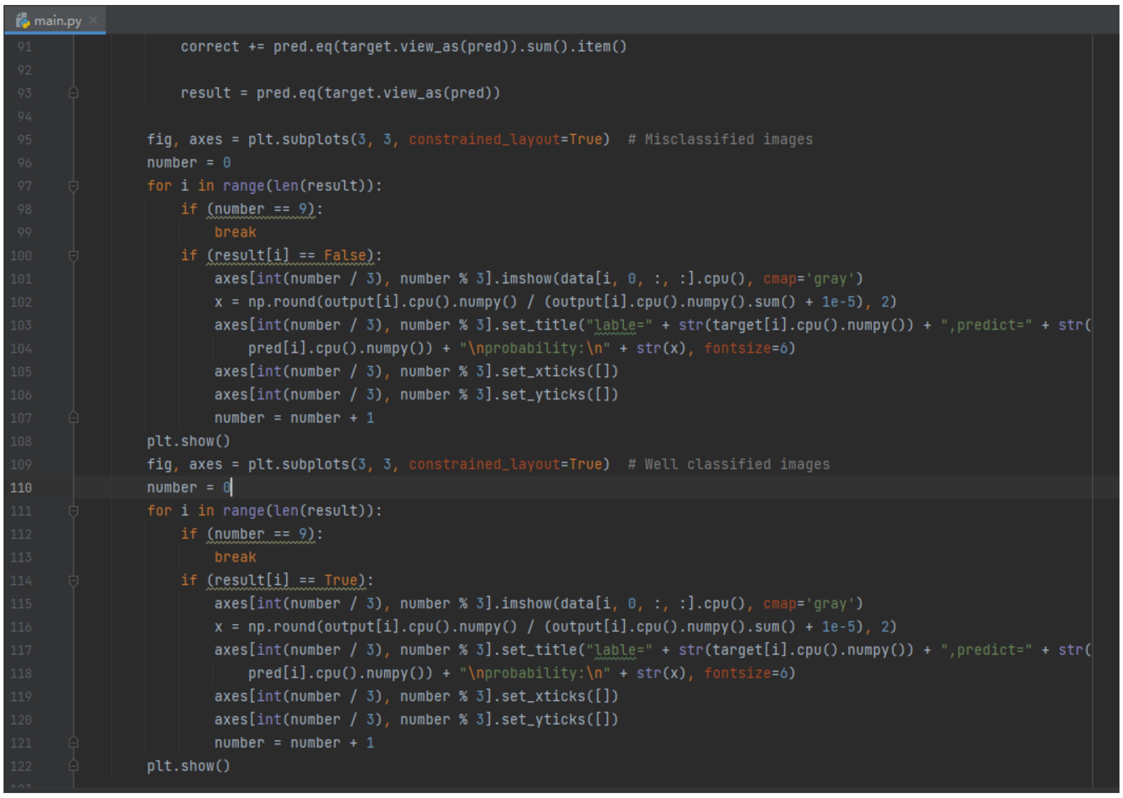
```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 8, 3, 1)     # 28*28 → (28+1-3)  26*26
        self.conv2 = nn.Conv2d(8, 16, 3, 1)
        self.dropout1 = nn.Dropout(0.25)
        self.dropout2 = nn.Dropout(0.5)
        self.fc1 = nn.Linear(2304, 64)  #full connection layer
        self.fc2 = nn.Linear(64, 10)
```

I didn't change the "def train" part, just annotate the "plt" part.



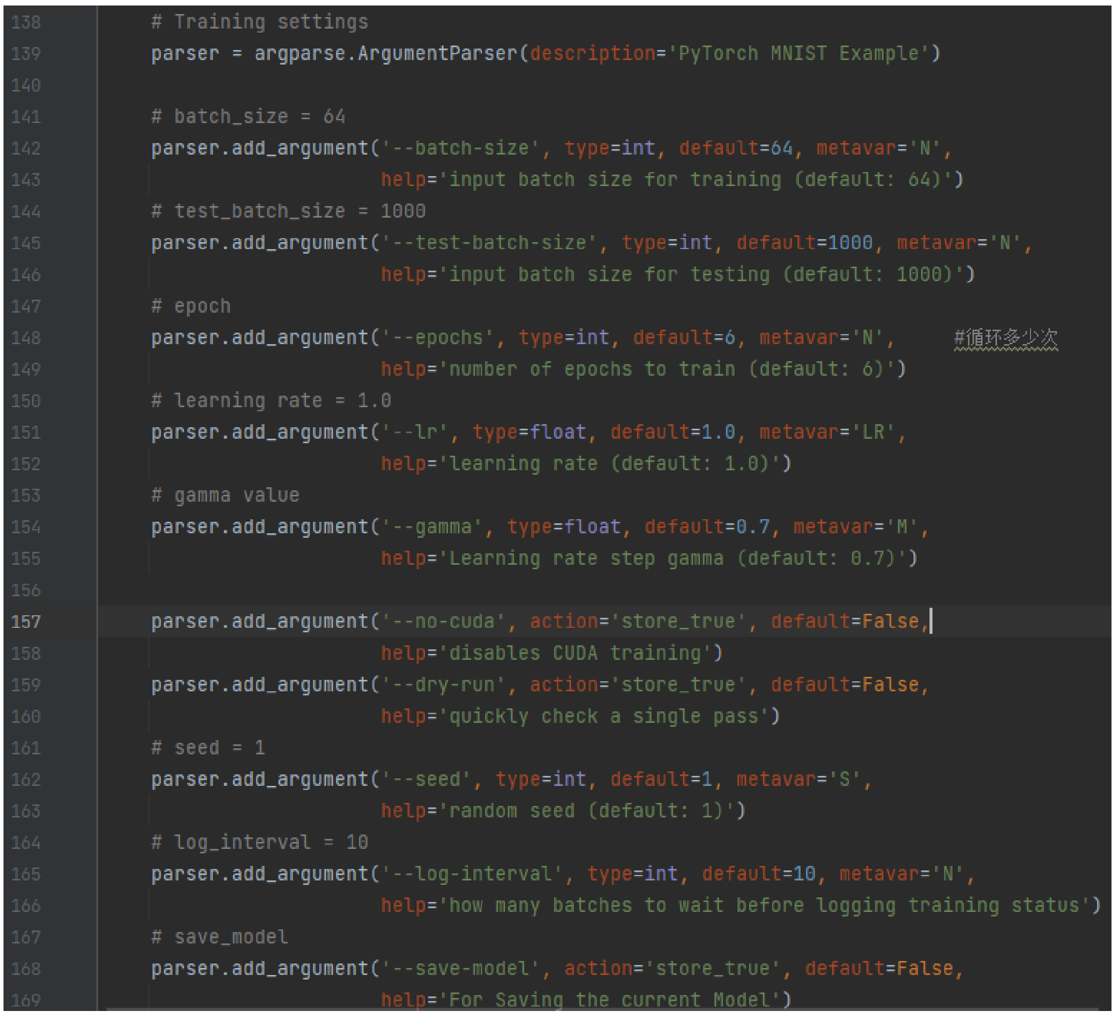
```
            if batch_idx % args.log_interval == 0:  # 10
                print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(   #轮次
                    epoch, batch_idx * len(data), len(train_loader.dataset),
                    100. * batch_idx / len(train_loader), loss.item()))
                if args.dry_run:
                    break
        # plt.imshow(pic.cpu(), cmap='gray')
        # plt.show()
```

Most importantly, in the "def test" part, I write the code below to **demonstrate some well classified and misclassified images**, and with their **corresponding classification confidence value.** Other codes in the "def test" part, I didn't change.



```
            correct += pred.eq(target.view_as(pred)).sum().item()

            result = pred.eq(target.view_as(pred))

        fig, axes = plt.subplots(3, 3, constrained_layout=True)  # Misclassified images
        number = 0
        for i in range(len(result)):
            if (number == 9):
                break
            if (result[i] == False):
                axes[int(number / 3), number % 3].imshow(data[i, 0, :, :].cpu(), cmap='gray')
                x = np.round(output[i].cpu().numpy() / (output[i].cpu().numpy().sum() + 1e-5), 2)
                axes[int(number / 3), number % 3].set_title("lable=" + str(target[i].cpu().numpy()) + ",predict=" + str(
                    pred[i].cpu().numpy()) + "\nprobability:\n" + str(x), fontsize=6)
                axes[int(number / 3), number % 3].set_xticks([])
                axes[int(number / 3), number % 3].set_yticks([])
                number = number + 1
        plt.show()
        fig, axes = plt.subplots(3, 3, constrained_layout=True)  # Well classified images
        number = 0
        for i in range(len(result)):
            if (number == 9):
                break
            if (result[i] == True):
                axes[int(number / 3), number % 3].imshow(data[i, 0, :, :].cpu(), cmap='gray')
                x = np.round(output[i].cpu().numpy() / (output[i].cpu().numpy().sum() + 1e-5), 2)
                axes[int(number / 3), number % 3].set_title("lable=" + str(target[i].cpu().numpy()) + ",predict=" + str(
                    pred[i].cpu().numpy()) + "\nprobability:\n" + str(x), fontsize=6)
                axes[int(number / 3), number % 3].set_xticks([])
                axes[int(number / 3), number % 3].set_yticks([])
                number = number + 1
        plt.show()
```

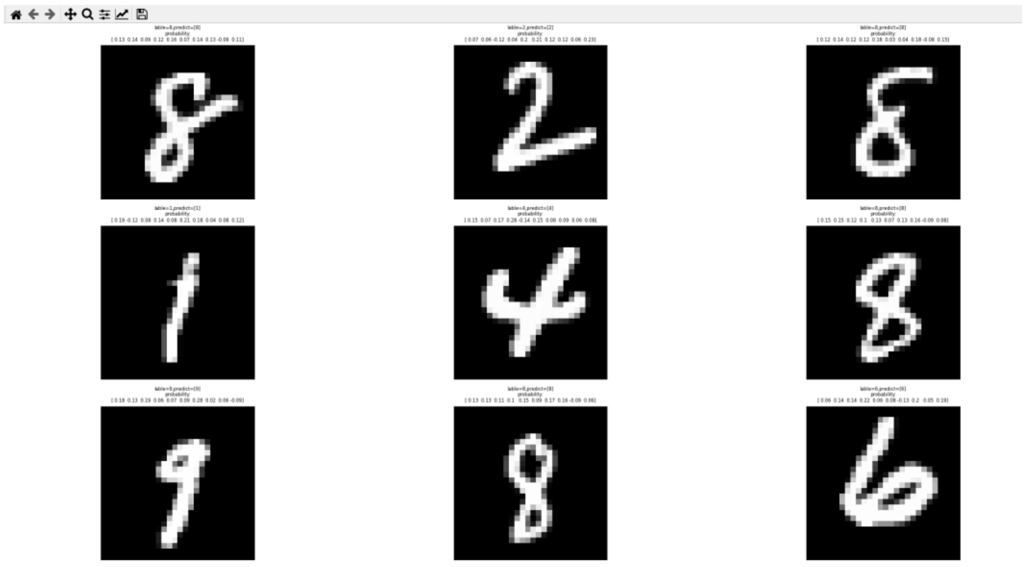
For “main” part, I didn’t change a lot. Keeping the default setting: batch size = 1, test_batch_size = 1000, epoch = 14, learning rate = 1.0, gamma value = 0.7, log interval = 10



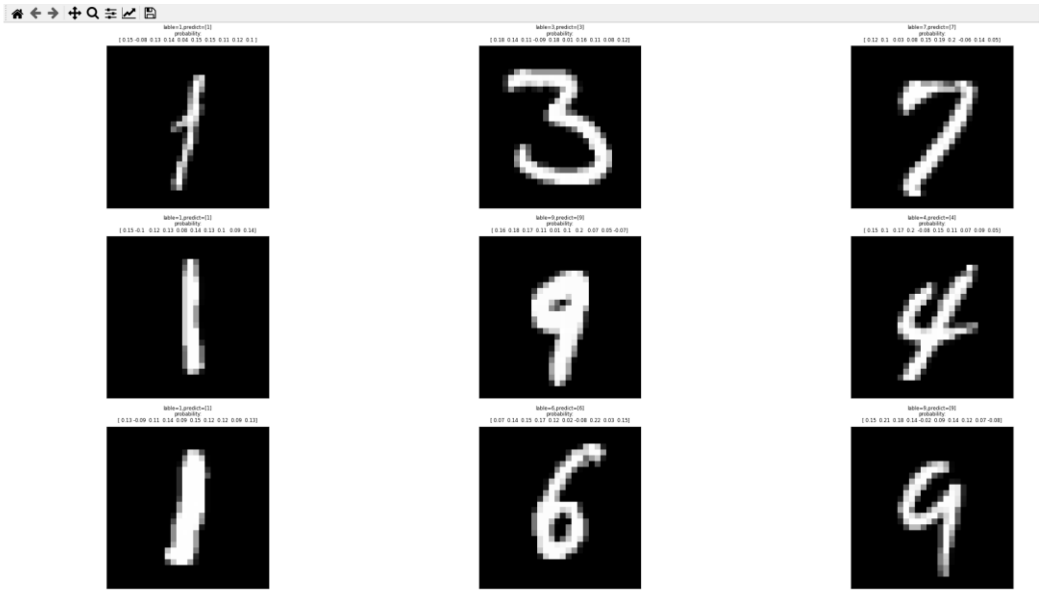
```python
    # Training settings
    parser = argparse.ArgumentParser(description='PyTorch MNIST Example')

    # batch_size = 64
    parser.add_argument('--batch-size', type=int, default=64, metavar='N',
                        help='input batch size for training (default: 64)')
    # test_batch_size = 1000
    parser.add_argument('--test-batch-size', type=int, default=1000, metavar='N',
                        help='input batch size for testing (default: 1000)')
    # epoch
    parser.add_argument('--epochs', type=int, default=6, metavar='N',     #循环多少次
                        help='number of epochs to train (default: 6)')
    # learning rate = 1.0
    parser.add_argument('--lr', type=float, default=1.0, metavar='LR',
                        help='learning rate (default: 1.0)')
    # gamma value
    parser.add_argument('--gamma', type=float, default=0.7, metavar='M',
                        help='Learning rate step gamma (default: 0.7)')

    parser.add_argument('--no-cuda', action='store_true', default=False,
                        help='disables CUDA training')
    parser.add_argument('--dry-run', action='store_true', default=False,
                        help='quickly check a single pass')
    # seed = 1
    parser.add_argument('--seed', type=int, default=1, metavar='S',
                        help='random seed (default: 1)')
    # log_interval = 10
    parser.add_argument('--log-interval', type=int, default=10, metavar='N',
                        help='how many batches to wait before logging training status')
    # save_model
    parser.add_argument('--save-model', action='store_true', default=False,
                        help='For Saving the current Model')
```

Run the “main.py”

Here are some of **well classified images** and the **corresponding classification confidence value**.



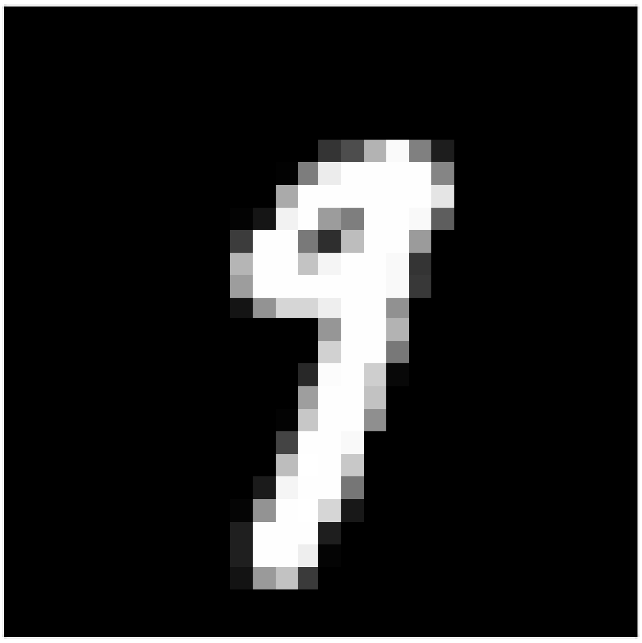
```
lable=8,predict=[8]
probability:
[ 0.13  0.14  0.09  0.12  0.16  0.07  0.14  0.13 -0.09  0.11]

lable=2,predict=[2]
probability:
[ 0.07  0.06 -0.12  0.04  0.2   0.21  0.12  0.12  0.06  0.23]

lable=8,predict=[8]
probability:
[ 0.12  0.14  0.12  0.12  0.18  0.03  0.04  0.18 -0.08  0.15]

lable=1,predict=[1]
probability:
[ 0.19 -0.12  0.08  0.14  0.08  0.21  0.18  0.04  0.08  0.12]

lable=4,predict=[4]
probability:
[ 0.15  0.07  0.17  0.28 -0.14  0.15  0.08  0.09  0.06  0.08]

lable=8,predict=[8]
probability:
[ 0.15  0.15  0.12  0.1   0.13  0.07  0.13  0.16 -0.09  0.08]

lable=9,predict=[9]
probability:
[ 0.18  0.13  0.19  0.06  0.07  0.09  0.28  0.02  0.06 -0.09]

lable=8,predict=[8]
probability:
[ 0.13  0.13  0.11  0.1   0.15  0.09  0.17  0.16 -0.09  0.06]

lable=6,predict=[6]
probability:
[ 0.06  0.14  0.14  0.22  0.06  0.08 -0.13  0.2   0.05  0.19]
```



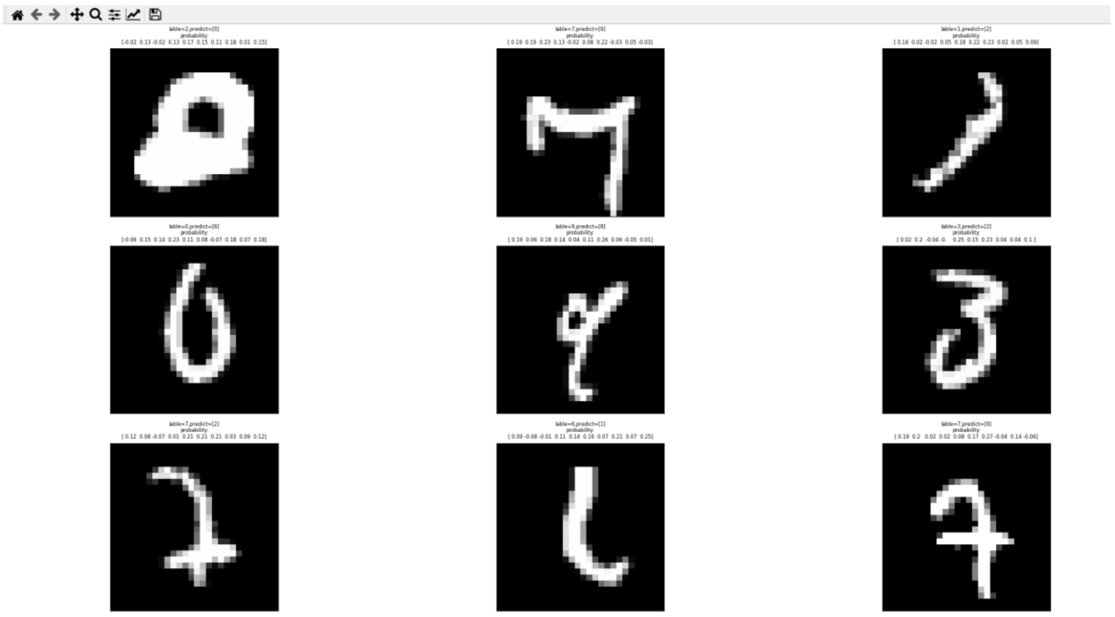
```
lable=1,predict=[1]
probability:
[ 0.15 -0.08  0.13  0.14  0.04  0.15  0.15  0.11  0.12  0.1 ]

lable=3,predict=[3]
probability:
[ 0.18  0.14  0.11 -0.09  0.18  0.01  0.16  0.11  0.08  0.12]

lable=7,predict=[7]
probability:
[ 0.12  0.1   0.03  0.08  0.15  0.19  0.2  -0.06  0.14  0.05]

lable=1,predict=[1]
probability:
[ 0.15 -0.1   0.12  0.13  0.08  0.14  0.13  0.1   0.09  0.14]

lable=9,predict=[9]
probability:
[ 0.16  0.18  0.17  0.11  0.01  0.1   0.2   0.07  0.05 -0.07]

lable=4,predict=[4]
probability:
[ 0.15  0.1   0.17  0.2  -0.08  0.15  0.11  0.07  0.09  0.05]

lable=1,predict=[1]
probability:
[ 0.13 -0.09  0.11  0.14  0.09  0.15  0.12  0.12  0.09  0.13]

lable=6,predict=[6]
probability:
[ 0.07  0.14  0.15  0.17  0.12  0.02 -0.08  0.22  0.03  0.15]

lable=9,predict=[9]
probability:
[ 0.15  0.21  0.18  0.14 -0.02  0.09  0.14  0.12  0.07 -0.08]
```

If the confidence value of well classified images is not clear in the above images, I post one of them below to **zoom in** for clearly checking.

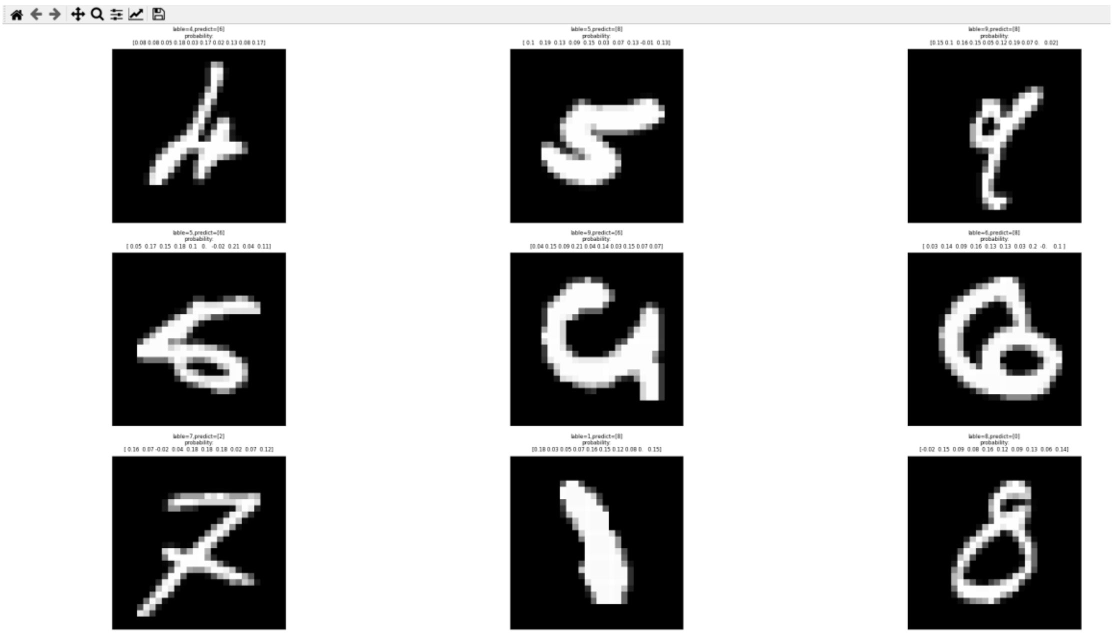
```
lable=9,predict=[9]
probability:
[ 0.19  0.12  0.19  0.05  0.07  0.09  0.3   0.01  0.07 -0.08]
```



Here are some of **misclassified images** and the **corresponding classification confidence value**.



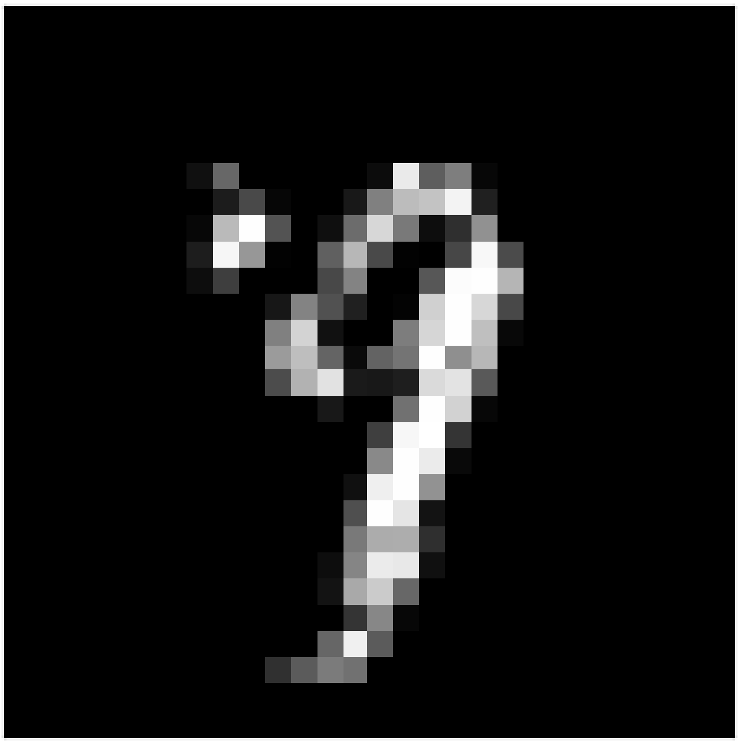
```
lable=2,predict=[0]
probability:
[-0.02  0.13 -0.02  0.13  0.17  0.15  0.11  0.18  0.01  0.15]

lable=7,predict=[9]
probability:
[ 0.19  0.19  0.23  0.13 -0.02  0.08  0.22 -0.03  0.05 -0.03]

lable=1,predict=[2]
probability:
[ 0.16  0.02 -0.02  0.05  0.18  0.22  0.23  0.02  0.05  0.09]

lable=0,predict=[6]
probability:
[-0.06  0.15  0.14  0.23  0.11  0.08 -0.07  0.18  0.07  0.18]

lable=9,predict=[8]
probability:
[ 0.19  0.06  0.18  0.14  0.04  0.11  0.26  0.06 -0.05  0.01]

lable=3,predict=[2]
probability:
[ 0.02  0.2  -0.04 -0.    0.25  0.15  0.23  0.04  0.04  0.1 ]

lable=7,predict=[2]
probability:
[ 0.12  0.08 -0.07  0.01  0.21  0.21  0.21  0.03  0.09  0.12]

lable=6,predict=[1]
probability:
[ 0.09 -0.08 -0.01  0.11  0.14  0.16  0.07  0.21  0.07  0.25]

lable=7,predict=[9]
probability:
[ 0.19  0.2   0.02  0.02  0.08  0.17  0.27 -0.04  0.14 -0.06]
```



```
lable=4,predict=[6]
probability:
[0.08 0.08 0.05 0.18 0.03 0.17 0.02 0.13 0.08 0.17]

lable=5,predict=[8]
probability:
[ 0.1   0.19  0.13  0.09  0.15  0.03  0.07  0.13 -0.01  0.13]

lable=9,predict=[8]
probability:
[0.15 0.1  0.16 0.15 0.05 0.12 0.19 0.07 0.   0.02]

lable=5,predict=[6]
probability:
[ 0.05  0.17  0.15  0.18  0.1   0.   -0.02  0.21  0.04  0.11]

lable=9,predict=[6]
probability:
[0.04 0.15 0.09 0.21 0.04 0.14 0.03 0.15 0.07 0.07]

lable=6,predict=[8]
probability:
[ 0.03  0.14  0.09  0.16  0.13  0.13  0.03  0.2  -0.    0.1 ]

lable=7,predict=[2]
probability:
[ 0.16  0.07 -0.02  0.04  0.18  0.18  0.18  0.02  0.07  0.12]

lable=1,predict=[8]
probability:
[0.18 0.03 0.05 0.07 0.16 0.15 0.12 0.08 0.   0.15]

lable=8,predict=[0]
probability:
[-0.02  0.15  0.09  0.08  0.16  0.12  0.09  0.13  0.06  0.14]
```

If the confidence value of well classified images is not clear in the above images, I post one of them below to **zoom in**.

lable=9,predict=[7]
probability:
[0.17 0.18 0.08 0.01 0.1   0.16 0.28 -0.06 0.15 -0.06]



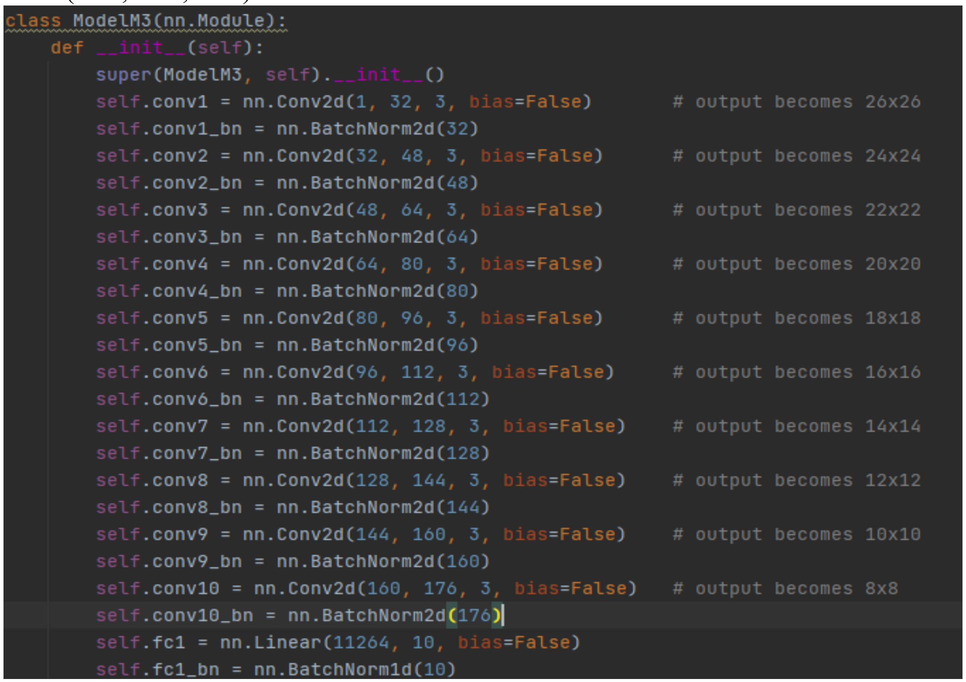
Task 3:

According to the statistic, it shows that using k-nearest neighbors (KNN), random forests, support vector machines (SVM), and simple neural network models could easily achieve 97-98% accuracy performance. For our example code, it used basic CNN to achieve 98.85%, as the screenshot shown in the Task 2.

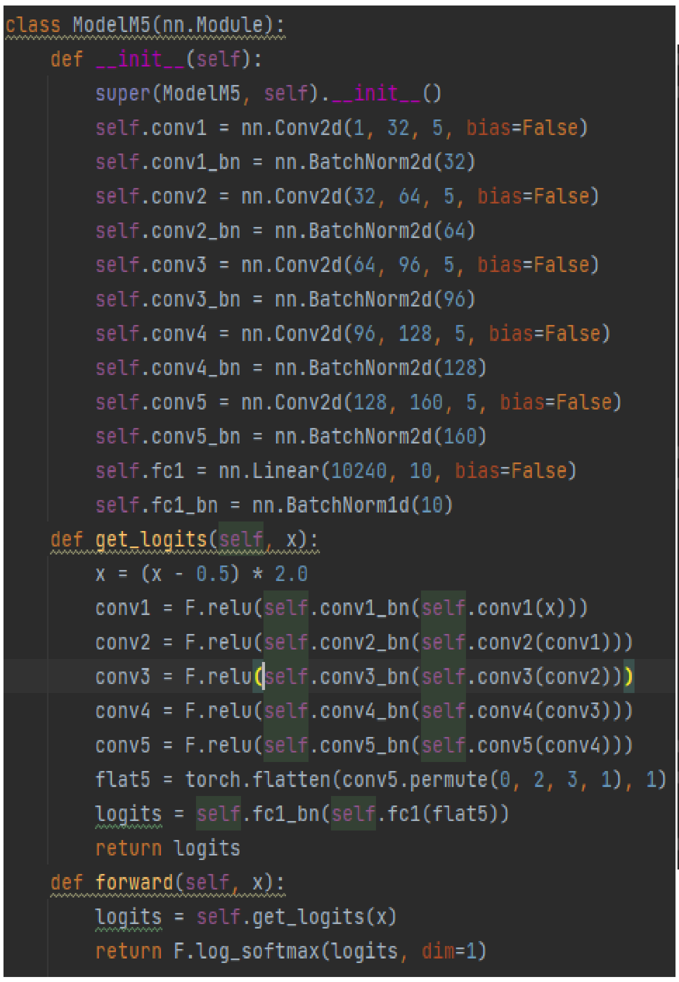
However, the last nearly 1% accuracy performance is extremely difficult to optimize. As the screenshots in the Task 2, even a person cannot recognize some digital images. It requires us to careful tuning the hyperparameters in the example code default setting, such as learning rate (lr) and batch_size.

For improvement of the accuracy performance, *a new method which use three different models with 3×3, 5×5, 7×7 kernel size is going to be introduced*. Each model consists of a set of convolution layers followed by a single fully connected layer (fc). Every convolution layer uses batch normalization and ReLU activation. Additionally, a file named "transforms" are used for translation and rotation to augment training data. The three models have similar architectures, using majority voting when obtaining the final prediction.

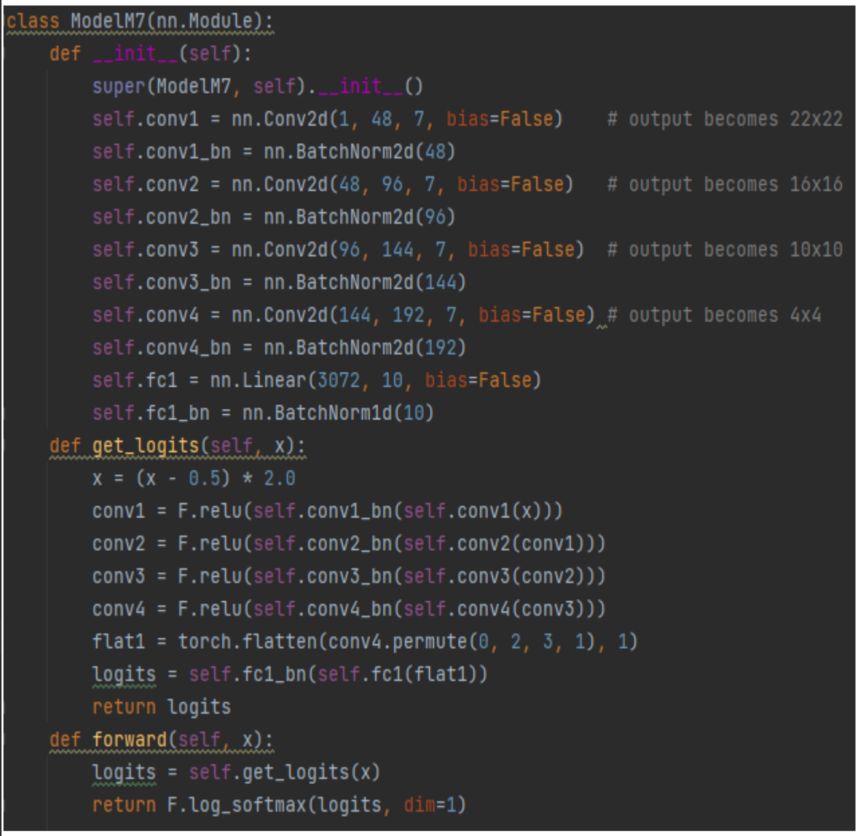
For **network design**, if we use a 3×3 kernel, the width and height of the image is reduced by two after each convolution layer. The number of channels is increased after each layer in order to account for reduction in feature map size. Once the feature map size becomes small enough, a fully-connected layer connects the feature map to the final output. The networks differ only in the kernel sizes (3×3, 5×5, 7×7).



```
class ModelM3(nn.Module):
    def __init__(self):
        super(ModelM3, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, bias=False)       # output becomes 26x26
        self.conv1_bn = nn.BatchNorm2d(32)
        self.conv2 = nn.Conv2d(32, 48, 3, bias=False)      # output becomes 24x24
        self.conv2_bn = nn.BatchNorm2d(48)
        self.conv3 = nn.Conv2d(48, 64, 3, bias=False)      # output becomes 22x22
        self.conv3_bn = nn.BatchNorm2d(64)
        self.conv4 = nn.Conv2d(64, 80, 3, bias=False)      # output becomes 20x20
        self.conv4_bn = nn.BatchNorm2d(80)
        self.conv5 = nn.Conv2d(80, 96, 3, bias=False)      # output becomes 18x18
        self.conv5_bn = nn.BatchNorm2d(96)
        self.conv6 = nn.Conv2d(96, 112, 3, bias=False)     # output becomes 16x16
        self.conv6_bn = nn.BatchNorm2d(112)
        self.conv7 = nn.Conv2d(112, 128, 3, bias=False)    # output becomes 14x14
        self.conv7_bn = nn.BatchNorm2d(128)
        self.conv8 = nn.Conv2d(128, 144, 3, bias=False)    # output becomes 12x12
        self.conv8_bn = nn.BatchNorm2d(144)
        self.conv9 = nn.Conv2d(144, 160, 3, bias=False)    # output becomes 10x10
        self.conv9_bn = nn.BatchNorm2d(160)
        self.conv10 = nn.Conv2d(160, 176, 3, bias=False)   # output becomes 8x8
        self.conv10_bn = nn.BatchNorm2d(176)
        self.fc1 = nn.Linear(11264, 10, bias=False)
        self.fc1_bn = nn.BatchNorm1d(10)
```



```python
class ModelM5(nn.Module):
    def __init__(self):
        super(ModelM5, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 5, bias=False)
        self.conv1_bn = nn.BatchNorm2d(32)
        self.conv2 = nn.Conv2d(32, 64, 5, bias=False)
        self.conv2_bn = nn.BatchNorm2d(64)
        self.conv3 = nn.Conv2d(64, 96, 5, bias=False)
        self.conv3_bn = nn.BatchNorm2d(96)
        self.conv4 = nn.Conv2d(96, 128, 5, bias=False)
        self.conv4_bn = nn.BatchNorm2d(128)
        self.conv5 = nn.Conv2d(128, 160, 5, bias=False)
        self.conv5_bn = nn.BatchNorm2d(160)
        self.fc1 = nn.Linear(10240, 10, bias=False)
        self.fc1_bn = nn.BatchNorm1d(10)
    def get_logits(self, x):
        x = (x - 0.5) * 2.0
        conv1 = F.relu(self.conv1_bn(self.conv1(x)))
        conv2 = F.relu(self.conv2_bn(self.conv2(conv1)))
        conv3 = F.relu(self.conv3_bn(self.conv3(conv2)))
        conv4 = F.relu(self.conv4_bn(self.conv4(conv3)))
        conv5 = F.relu(self.conv5_bn(self.conv5(conv4)))
        flat5 = torch.flatten(conv5.permute(0, 2, 3, 1), 1)
        logits = self.fc1_bn(self.fc1(flat5))
        return logits
    def forward(self, x):
        logits = self.get_logits(x)
        return F.log_softmax(logits, dim=1)
```



```python
class ModelM7(nn.Module):
    def __init__(self):
        super(ModelM7, self).__init__()
        self.conv1 = nn.Conv2d(1, 48, 7, bias=False)    # output becomes 22x22
        self.conv1_bn = nn.BatchNorm2d(48)
        self.conv2 = nn.Conv2d(48, 96, 7, bias=False)   # output becomes 16x16
        self.conv2_bn = nn.BatchNorm2d(96)
        self.conv3 = nn.Conv2d(96, 144, 7, bias=False)  # output becomes 10x10
        self.conv3_bn = nn.BatchNorm2d(144)
        self.conv4 = nn.Conv2d(144, 192, 7, bias=False) # output becomes 4x4
        self.conv4_bn = nn.BatchNorm2d(192)
        self.fc1 = nn.Linear(3072, 10, bias=False)
        self.fc1_bn = nn.BatchNorm1d(10)
    def get_logits(self, x):
        x = (x - 0.5) * 2.0
        conv1 = F.relu(self.conv1_bn(self.conv1(x)))
        conv2 = F.relu(self.conv2_bn(self.conv2(conv1)))
        conv3 = F.relu(self.conv3_bn(self.conv3(conv2)))
        conv4 = F.relu(self.conv4_bn(self.conv4(conv3)))
        flat1 = torch.flatten(conv4.permute(0, 2, 3, 1), 1)
        logits = self.fc1_bn(self.fc1(flat1))
        return logits
    def forward(self, x):
        logits = self.get_logits(x)
        return F.log_softmax(logits, dim=1)
```

As the screenshots showed on the above, we can easily get that:

For an input image (28*28, 1ch) for Model M3, the process is:

Conv2d(3×3, 32ch), BatchNorm2d(32),ReLU

↓

Conv2d(3×3, 48ch), BatchNorm2d(48),ReLU

↓

Conv2d(3×3, 64ch), BatchNorm2d(64),ReLU

↓

Conv2d(3×3, 80ch), BatchNorm2d(80),ReLU

↓

Conv2d(3×3, 96ch), BatchNorm2d(96),ReLU

↓

Conv2d(3×3, 112ch), BatchNorm2d(112),ReLU

↓

Conv2d(3×3, 128ch), BatchNorm2d(128),ReLU

↓

Conv2d(3×3, 144ch), BatchNorm2d(144),ReLU

↓

Conv2d(3×3, 160ch), BatchNorm2d(160),ReLU

↓

Conv2d(3×3, 176ch), BatchNorm2d(176),ReLU

↓

Linear(11264, 10), BatchNorm1d(10)

For Model M5, the process:

Conv2d(5×5, 32ch), BatchNorm2d(32),ReLU

↓

Conv2d(5×5, 64ch), BatchNorm2d(64),ReLU

↓

Conv2d(5×5, 96ch), BatchNorm2d(96),ReLU

↓

Conv2d(5×5, 128ch), BatchNorm2d(128),ReLU

↓

Conv2d(5×5, 160ch), BatchNorm2d(160),ReLU

↓

For Model M7, the process:

Conv2d(7×7, 48ch), BatchNorm2d(48),ReLU

↓

Conv2d(7×7, 96ch), BatchNorm2d(96),ReLU

↓

Conv2d(7×7, 144ch), BatchNorm2d(144),ReLU

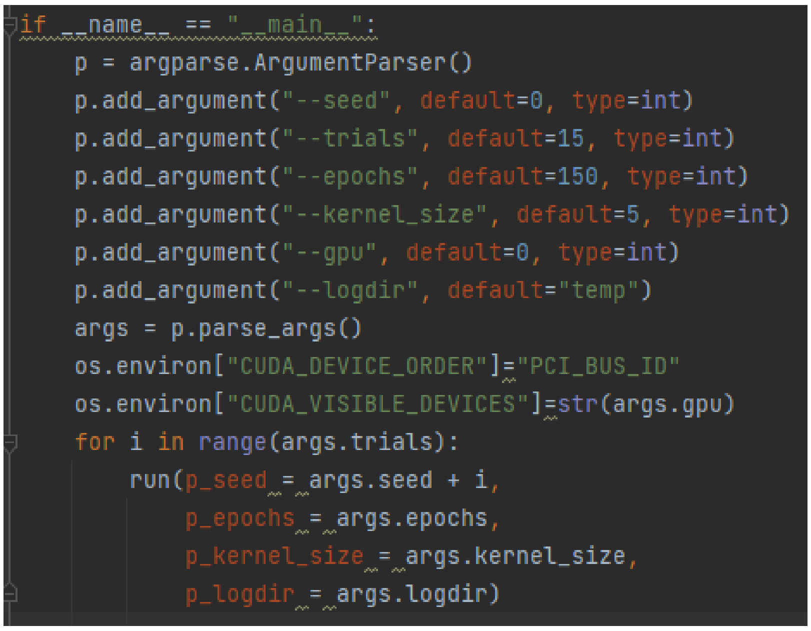
↓

Conv2d(7×7, 192ch), BatchNorm2d(192),ReLU

↓

Linear(3072, 10), BatchNorm1d(10)

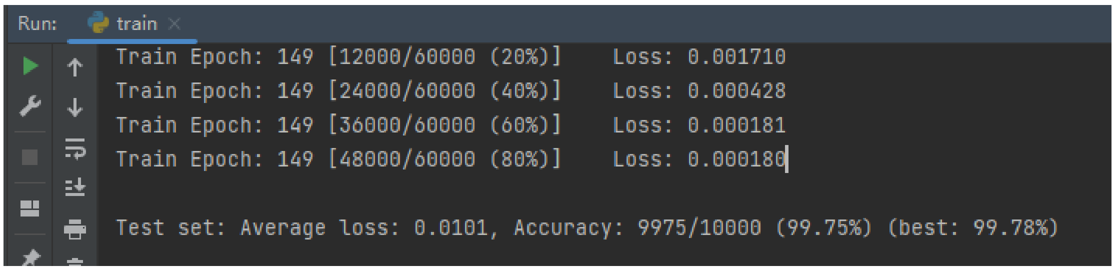
Linear(10240, 10), BatchNorm1d(10)



```python
if __name__ == "__main__":
    p = argparse.ArgumentParser()
    p.add_argument("--seed", default=0, type=int)
    p.add_argument("--trials", default=15, type=int)
    p.add_argument("--epochs", default=150, type=int)
    p.add_argument("--kernel_size", default=5, type=int)
    p.add_argument("--gpu", default=0, type=int)
    p.add_argument("--logdir", default="temp")
    args = p.parse_args()
    os.environ["CUDA_DEVICE_ORDER"]="PCI_BUS_ID"
    os.environ["CUDA_VISIBLE_DEVICES"]=str(args.gpu)
    for i in range(args.trials):
        run(p_seed = args.seed + i,
            p_epochs = args.epochs,
            p_kernel_size = args.kernel_size,
            p_logdir = args.logdir)
```

Here is the default argument and one of the train results

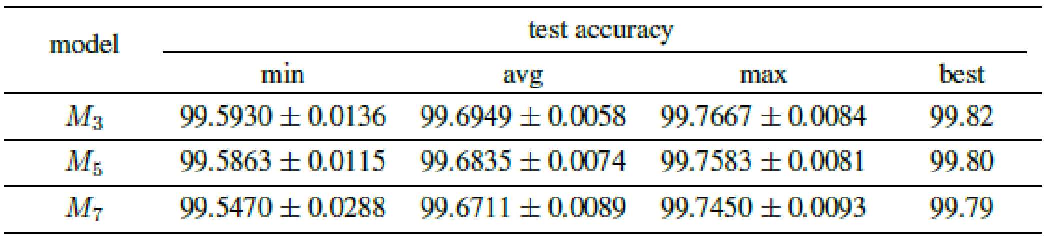
It trains 150 epochs (initial number is 0). The accuracy performance is **99.75% (best: 99.78%)**, much higher than the example code. The average loss is **0.0101**.



```
Train Epoch: 149 [12000/60000 (20%)]    Loss: 0.001710
Train Epoch: 149 [24000/60000 (40%)]    Loss: 0.000428
Train Epoch: 149 [36000/60000 (60%)]    Loss: 0.000181
Train Epoch: 149 [48000/60000 (80%)]    Loss: 0.000180

Test set: Average loss: 0.0101, Accuracy: 9975/10000 (99.75%) (best: 99.78%)
```

For each type of network, we have trained 30 networks with different initial parameters. Each network was trained for 150 epochs, since the test accuracy hardly improved after that point.

The picture below shows the minimum, average, maximum accuracy of 30 networks between 50 and 150 epochs, in the 95% confidence range. The accuracy of M3 is slightly higher followed by M5 and M7, but the difference is not too obvious (less than 0.02%). Between 50 and 150 epochs of 30 networks, the highest test accuracy observed from M3, M5, M7 was 99.82%, 99.80%, and 99.79% respectively.



model	test accuracy			
	min	avg	max	best
M_3	99.5930 ± 0.0136	99.6949 ± 0.0058	99.7667 ± 0.0084	99.82
M_5	99.5863 ± 0.0115	99.6835 ± 0.0074	99.7583 ± 0.0081	99.80
M_7	99.5470 ± 0.0288	99.6711 ± 0.0089	99.7450 ± 0.0093	99.79

In the before method, we don’t use pooling and dropout. However, we know when building a CNN, a common practice is to use pooling, such as max pooling or average pooling. A commonly used CNN model consists of a set of convolution layers where each convolution layer is followed by a pooling layer, and one or multiple fully connected layers at the end. Some networks have two convolution layers before the pooling layer.

Take an example, C1 C2 and C3:

Input image (28*28, 1ch)

C1:

Conv2d(5×5, 64ch), BatchNorm2d(64),ReLU

↓

MaxPool(2×2)

↓

Conv2d(5×5, 128ch), BatchNorm2d(128),ReLU

↓

MaxPool(2×2)

↓

Linear(6272, 10), BatchNorm1d(10)

↓

C2:

Conv2d(5×5, 64ch), BatchNorm2d(64),ReLU

↓

MaxPool(2×2)

↓

Conv2d(5×5, 128ch), BatchNorm2d(128),ReLU

↓

MaxPool(2×2)

↓

Linear(6272, 10), BatchNorm1d(10)

↓

Linear(100, 10), BatchNorm1d(10)

C3:

Conv2d(5×5, 32ch), BatchNorm2d(32),ReLU

↓

Conv2d(5×5, 64ch), BatchNorm2d(64),ReLU

↓

MaxPool(2×2)

↓

Conv2d(5×5, 96ch), BatchNorm2d(96),ReLU

↓

Conv2d(5×5, 128ch), BatchNorm2d(128),ReLU

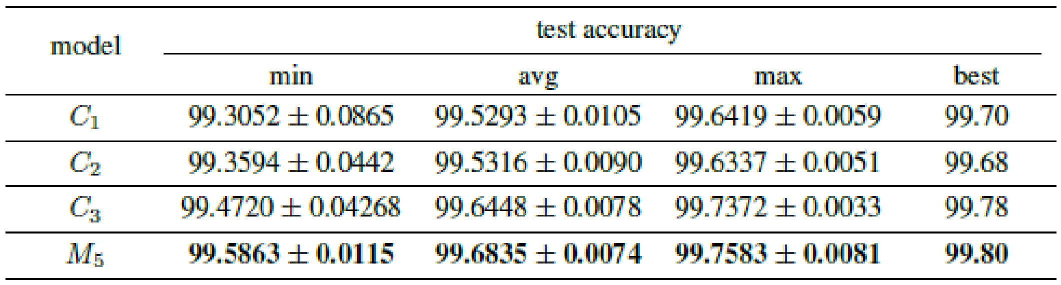
↓

MaxPool(2×2)

↓

Linear(6272, 10), BatchNorm1d(10)

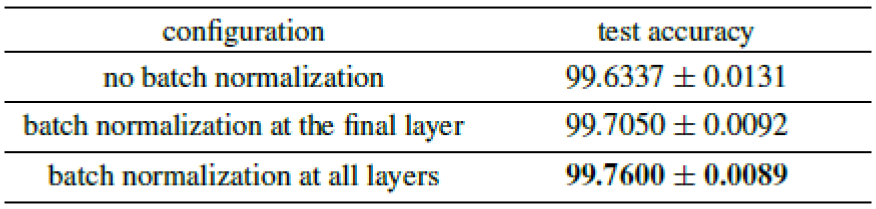
It can be observed that for networks using max pooling, the test accuracy goes through oscillations in the early stage of training. On the other hand, the test accuracy of M5 increases in a more stable manner.



model	test accuracy			
	min	avg	max	best
C_1	99.3052 ± 0.0865	99.5293 ± 0.0105	99.6419 ± 0.0059	99.70
C_2	99.3594 ± 0.0442	99.5316 ± 0.0090	99.6337 ± 0.0051	99.68
C_3	99.4720 ± 0.04268	99.6448 ± 0.0078	99.7372 ± 0.0033	99.78
M_5	**99.5863 ± 0.0115**	**99.6835 ± 0.0074**	**99.7583 ± 0.0081**	**99.80**

The average test accuracy of C3 and M5 is better than that of C1 and C2, which means using more convolution layers could result in better feature learning. Having more fully connected layers at the end did not help, as can be seen from the accuracy of C1 and C2. Between C3 and M5, M5 achieves higher accuracy in general, and can reach higher accuracy in the best case.

As for batch normalization, it is a technique to improve performance of the network as well as stability and speed of training. Compared three configurations: the first model uses no batch normalization at all, the second model uses batch normalization only at the fully connected layer, and the third model uses batch normalization at all layers



configuration	test accuracy
no batch normalization	99.6337 ± 0.0131
batch normalization at the final layer	99.7050 ± 0.0092
batch normalization at all layers	$\mathbf{99.7600 \pm 0.0089}$

It is evident that using batch normalization helps improve the performance of neural network models. The best performance is achieved when batch normalization is used at each convolution and fully connected layer.

In conclusion, MNIST dataset is a very classical dataset used for testing the accuracy performance. The example code has achieved a high accuracy performance, nearly 99%. However, many other methods could improve it (besides Pytorch, TensorFlow is also effective). Task 3 introduces a method by using 2D batch normalization and ReLU activation after each convolution layer. It has improved the accuracy performance into nearly 99.75% (9975/10000). We know there are still 25 handwritten images cannot be recognized by the computer. Hope in the future, the accuracy performance will improve higher.

References:

[1]. Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," in Proceedings of the IEEE, vol. 86, no. 11, pp. 2278-2324, Nov. 1998, doi: 10.1109/5.726791.

[2]. An, S., Lee, M., Park, S., Yang, H., & So, J. (2020). An ensemble of simple convolutional neural network models for mnist digit recognition. arXiv preprint arXiv:2008.10400.

[3]. https://towardsdatascience.com/types-of-convolution-kernels-simplified-f040cb307c37

[4]. https://programmathically.com/understanding-convolutional-filters-and-convolutional-kernels/

[5]. https://blog.csdn.net/gary101818/article/details/122458932

[6].https://www.bilibili.com/video/BV1eM411C7oF/?p=5&share_source=copy_web&vd_source=971c4d26440f07ed1fd2c5e76eee8fb2

[7].https://www.bilibili.com/video/BV1WT4y177SA/?spm_id_from=333.788&vd_source=7a20c867dbfe33b0dfae6eca6623a999

[8]. A. Comeau and C. McDonald, "Analyzing Decision Trees to Understand MNIST Misclassification," *2019 IEEE MIT Undergraduate Research Technology Conference (URTC)*, 2019, pp. 1-4, doi: 10.1109/URTC49097.2019.9660504.

[9]. S. Huang, "Influence of Different Convolutional Neural Network Settings on the Performance of MNIST Handwritten Digits Recognition," *2020 International Conference on Artificial Intelligence and Education (ICAIE)*, 2020, pp. 1-6, doi: 10.1109/ICAIE50891.2020.00008

[10]. Y. Lecun, C. Cortes, and C. J. Burges. “MNIST handwritten digit database”. In: ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist (2010)

[11]. A. Byerly, T. Kalganova, and I. Dear, “A branching and merging convolutional network with homogeneous filter capsules,” arXiv preprint arXiv:2001.09136 (2020)

[12]. S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” arXiv preprint arXiv:1502.03167 (2015)

[13]. https://blog.csdn.net/bublebee/article/details/88768381

[14]. https://en.wikipedia.org/wiki/Loss_function