MNIST Handwritten Data Modeling

mz2840

Part 1: Background

The MNIST database of handwritten digits is one of the most commonly used dataset for training various image processing systems and machine learning algorithms. MNIST has a training set of 60,000 examples, and a test set of 10,000 examples. MNIST is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image. The original black and white (bilevel) images from MNIST were size normalized.

Part 2: Data Processing

1. Familiarize yourself with the data

1) Here the dimension of the X_train is (60000, 28, 28), and the dimension of the X_test is (10000,28,28).
2) We normalize the X_train and X_test by applying to *normalize* function in the *sklearn.preprocessing* package.
3) Then we use *one-hot* embedding to deal with labels by using the *OneHotEncoder* function in the sklearn.preprocessing package. One-hot encoding can help transform the categorical variables into non-ordered numerical variables. The non-ordered numerical variables make it more reasonable and accurate to generate results.

Part 3: Before Deep Learning

1.KNN method

Grid search is an efficient method to process of performing hyperparameter tuning in order to determine the optimal values for a given model. After applying *GridSearchCV* function in the *sklearn.model_selection* package and *KNeighborsClassifier* function. And the output is when n_neighbors = 5, the model has the highest accuracy, the accuracy is 96.68%.

2. AdaBoost method

Here the weak classifier is C4.5 decision tree. When we set the max_depth = 10, and n_estimators = 70, it can tree the dataset very well, and the error after applying Adaboost is 1-0.9643 = 3.569%.

3. SVM method

As for SVM model, the predicted label should be one dimension array, therefore it is accurate that using the matrix after one hot encoding. Here we apply the SVM model on reshaped normalized X and raw label in the dataset. From the result of GridSearchCV output, when gamma = 0.01, C = 10,

the model has the highest accuracy, the accuracy is 97.42%.

4. Neural Network method

Here I implement tf.keras to develop a neural network, and I also created a sequential model which contains a plain stack of layers. Here I firstly add 3 layers in the sequential and applied *GridSearchCV* to search the best parameters. And the result is that the accuracy is 97.96%. And I would like to increase accuracy by introducing the Convolutional neural network model, it generates a better result, which the accuracy is 98.89%. I will talk more about CNN model later.

Part 4: Deep Learning

Prerequisites

For the deep learning part, make sure you have installed all of the following prerequisites on your development machine:

1) Connect GPU to run the code, use torch.cuda.is_available() to examine whether if you have connected with GPU. And in order to use GPU, be sure to send your model and data into the CUDA device.
2) Torch - PyTorch which is an optimized tensor library primarily used for Deep Learning applications using GPUs and CPUs, used *!pip install torch* or *!conda install torch* to install.

1. A single layer neural network with 100 hidden units

I firstly applied *torchvision.transforms.ToTensor* and *torch.utils.data.DataLoader* to load data and make it feasible to deal with later. Here I create a single layer neural network named *NeuralNet*, and it contains a single hidden layer with 100 units, the input size is 784, the hidden's activation function is the ReLU function, the output size is the number of classes which is 10. The optimizer technique I used is stochastic gradient descent. And I used *torch.manual_seed* to set five different seeds. The first seed is 1, the second seed is 1234, the third seed is 2340, the fourth seed is 100, and the fifth seed is 3450. All the seeds are the random numbers I picked. Figure 1 the screenshot of the single layer neural network model. And I train at least 150 epochs each time until the overfitting.

Figure 1: the single layer neural network model

```
torch.manual_seed(1)    # reproducible, set different seeds
class NeuralNet(nn.Module):
     def __init__(self, input_size, hidden_size, num_classes):
            super(NeuralNet, self).__init__()
            self.input_size = input_size
            self.hidden = nn.Linear(input_size, hidden_size)
            self.relu = nn.ReLU()
            self.out = nn.Linear(hidden_size, num_classes)
     def forward(self, x):
        # nonlinear activation
        x = F.relu(self.hidden(x))
        x = self.out(x)
        return x
net = NeuralNet(input_size, hidden_size, num_classes)
```

```
# optimizing methods
optimizer = torch.optim.SGD(net.parameters(), lr=learning_rate)
# choose loss function
loss_func = torch.nn.CrossEntropyLoss()
```

(a) Plot the average training cross-entropy error on the y-axis vs. the epoch number (x-axis). On the same figure, plot the average validation cross-entropy error function. How does the network's performance differ on the training set versus the validation set during learning? Use the plot of training and testing error curves to support your argument.

Here I applied t*orch.nn.CrossEntropyLoss()* to calculate the cross-entropy error and I created a list called *losses_train* to append each cross-entropy error of training set of each iteration. And created a list called *losses_test* to append each cross-entropy error of testing/validation set of each iteration. Figure 2 is the screenshot of the way of calculating cross-entropy error of five different seeds.

Figure 2: calculating cross-entropy error

```
n_total_steps = len(train_data)
losses_train = []
losses_test = []
errors_train = []
errors_test = []
for epoch in range(num_epochs):
    train_error = 0
    test_error = 0
    train_loss_sum = 0
    test_loss_sum = 0
    for i, (X_train, labels_train) in enumerate(train_loader):
        train = X_train.reshape(-1, 28*28)
        # forward passing
        out = net(train)
        predicted_train = torch.max(out.data,1)[1]
        batch_error = (predicted_train != labels_train).sum()
        train_error += batch_error
        train_loss = loss_func(out, labels_train)
        train_loss_sum += train_loss
        optimizer.zero_grad()
        train_loss.backward()
        optimizer.step()
        if (i+1) % 100 == 0:
            print (f'Epoch [{epoch+1}/{num_epochs}], Step[{i+1}/{n_total_steps}], Loss: {train_loss.item():.4f}')
    losses_train.append(train_loss_sum.item()/batch_size)
    errors_train.append(train_error.item()/batch_size)
```

```
    with torch.no_grad():
        for b,(X_test, labels_test) in enumerate(test_loader):
            test = X_test.reshape(-1, 28*28)
            output = net(test)
            predicted_test = torch.max(output.data,1)[1]
            test_error += (predicted_test != labels_test).sum()
            test_loss = loss_func(output, labels_test)
            test_loss_sum += test_loss
            # correct += pred.eq(target.data.view_as(pred)).sum()
    # test_loss /= len(test_loader.dataset)
    losses_test.append(test_loss_sum.item()/batch_size)
    errors_test.append(test_error.item()/batch_size)
```

Figure 3.1-3.2 are the plots of cross-entropy error of training set and testing set of selected 2 different seeds from 5 seeds.

Figure 3.1: seed = 1's cross-entropy error

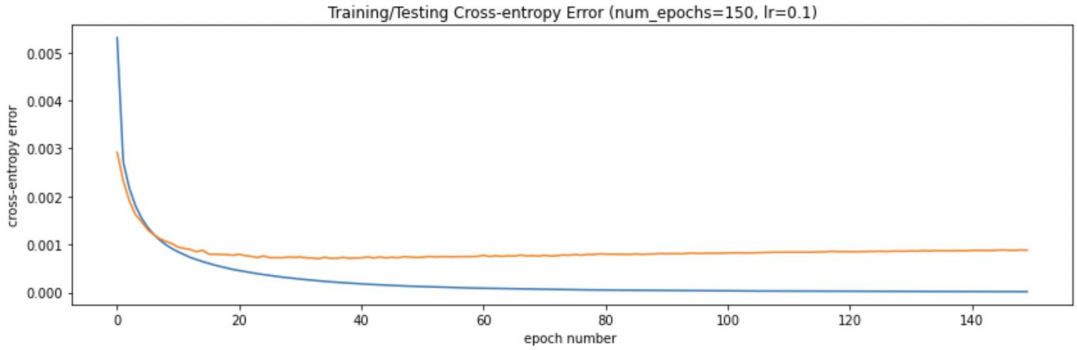
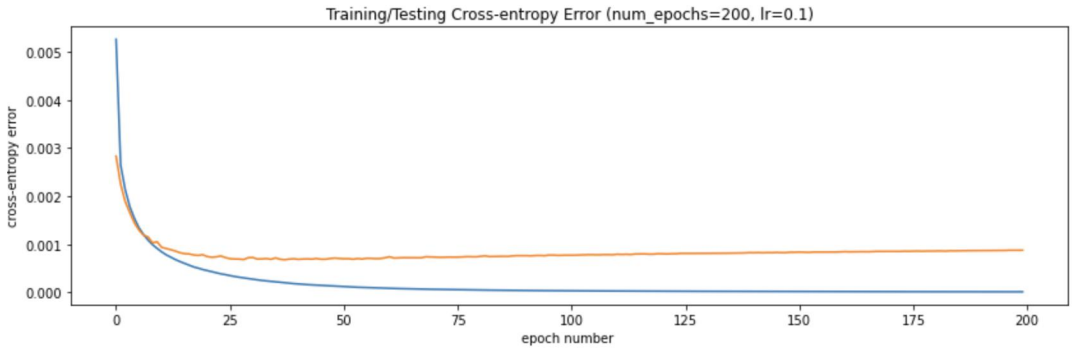


Figure 3.2: seed = 1234's cross-entropy error



Conclusion: What can be concluded from the plots is that after 150 times iterations there is overfitting. And the training cross-entropy error declines from 0.005 to almost 0 after iterations, and the validation cross-entropy error changes from 0.003 to 0.001. Since the testing error shows overfitting, the neural network model can train the dataset well. The training error is larger than the testing error initially which makes sense, since it might due to the model has been trained well after the training error. And the updated model can train the testing set well. And after iterations, testing error cannot reach 0 error like training error.

(b) We could implement an alternative performance measure to the cross entropy, the mean miss-classification error. Plot the classification error vs. number of epochs, for both training and testing. Do you observe a different behavior compared to the behavior of the cross-entropy error function?

I calculated the miss-classification error also in problem (a) by judging whether the predicted_train = torch.max(out.data,1)[1] is the same as labels_train or not. Figure 4 is the main part of the miss-classification error, which has been concluded in Figure 2 as well.

Figure 4: main part of calculating miss-classification error

```
for i, (X_train, labels_train) in enumerate(train_loader):
    train = X_train.reshape(-1, 28*28)
    # forward passing
    out = net(train)
    predicted_train = torch.max(out.data,1)[1]
    batch_error = (predicted_train != labels_train).sum()
    train_error += batch_error
    train_loss = loss_func(out, labels_train)
    train_loss_sum += train_loss
    optimizer.zero_grad()
    train_loss.backward()
    optimizer.step()
    if (i+1) % 100 == 0:
        print (f'Epoch [{epoch+1}/{num_epochs}], Step[{i+1}/{n_total_steps}], Loss: {train_loss.item():.4f}')
losses_train.append(train_loss_sum.item()/batch_size)
errors_train.append(train_error.item()/batch_size)

with torch.no_grad():
    for b,(X_test, labels_test) in enumerate(test_loader):
        test = X_test.reshape(-1, 28*28)
        output = net(test)
        predicted_test = torch.max(output.data,1)[1]
        test_error += (predicted_test != labels_test).sum()
        test_loss = loss_func(output, labels_test)
        test_loss_sum += test_loss
```

Figure 5.1-5.2 are the plots of cross-entropy error of training set and testing set of selected 2 different seeds from 5 seeds. **Note: here the miss-classification error is not in percentage because it is easy for us to compare.**

Figure 5.1: seed = 1's miss-classification error

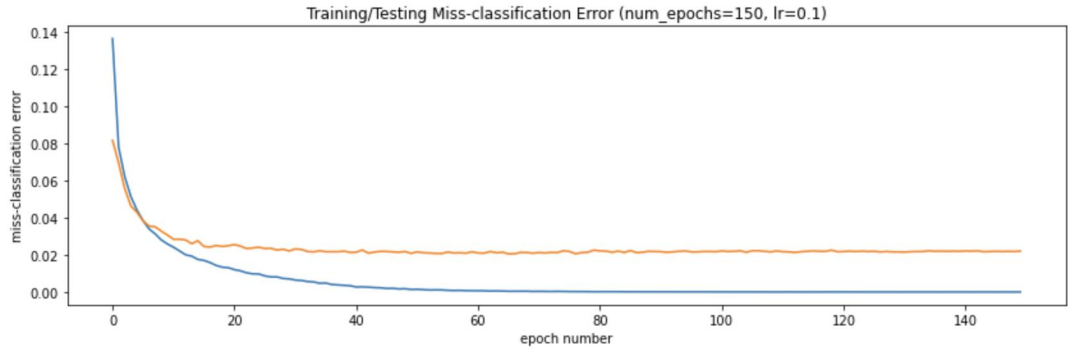
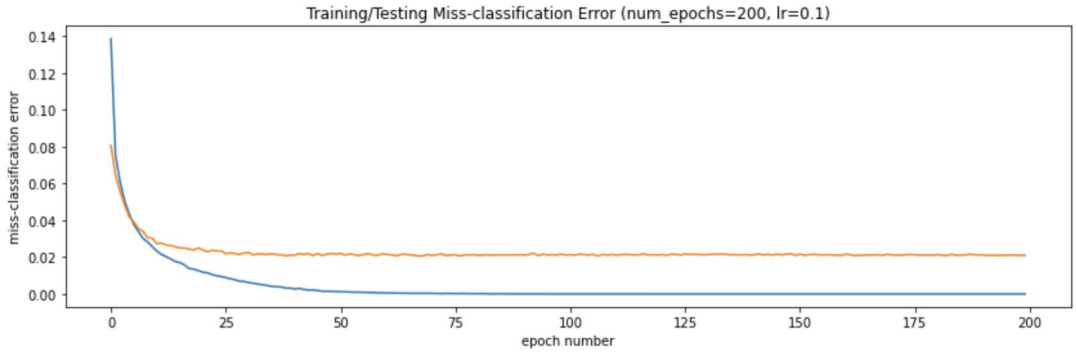


Figure 5.2: seed = 1234's miss-classification error



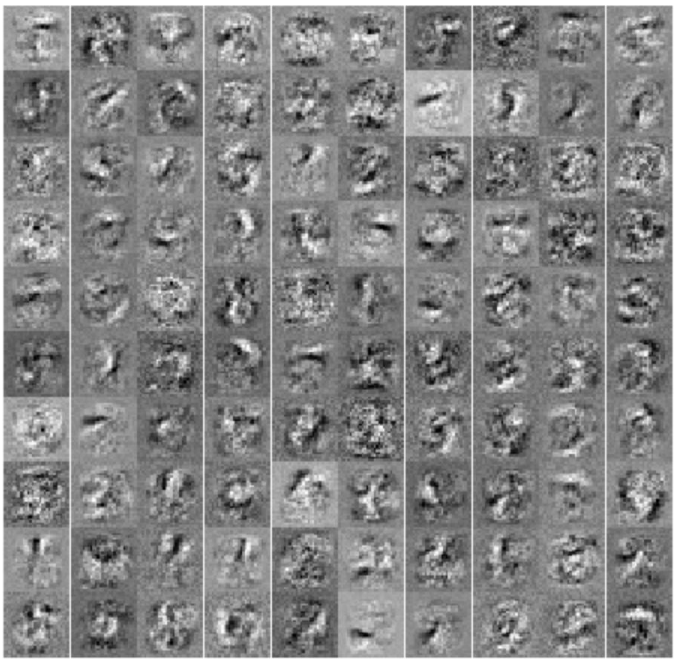
Conclusion: What can be concluded from the plots is that after 150 times iterations there is overfitting. And the training miss-classification error declines from 0.14 to almost 0 after iterations, and the validation testing error changes from 0.08 to 0.02. Since the testing error shows overfitting, the neural network model can train the dataset well. And there are not so much

different behaviors between cross-entropy error and miss-classification error, but due to the different methods of calculating errors, the miss-classification error is a bit larger.
And for a neural network classifier, during training we can use mean squared error or average cross-entropy error, and average cross-entropy error is considered slightly better. If we are using back-propagation, the choice of MSE or ACE affects the computation of the gradient. After training, to estimate the effectiveness of the neural network it's better to use classification error.

(c) Visualize your best results of the learned W as one hundred 28×28 images (plot all filters as one image, as we have seen in class). Do the learned features exhibit any structure?

Here I compare with the five plots in the (a) part, there are almost the same amount five plots. The fifth trial with manual_seed(3450) and num_epochs = 200,batch_size = 100, learning rate = 0.1. The training error is more close to 0, and the testing error shows the overfitting. Therefore, I choose this seed. Figure 6 shows the plot of filters of neural network.

Figure 6: filters of neural network



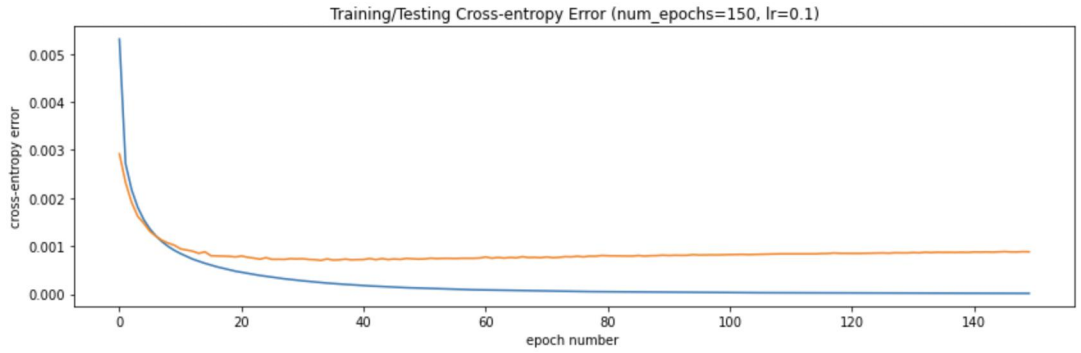
Conclusion: The filter which matches the most with the given input region will produce an output which will be higher in magnitude (compared to the output from other filters). By visualizing filters we get an idea of what pattern each layer has learned to extract from the input.
From the plot, we found that more complex shapes being encoded by the filters. And it can show some of the figures' shape. From the most of the information on numbers is captured, the classification performs well.

(d) Try different values of the learning rate. How do momentum and learning rate affect convergence rate? How would you choose the best value of these parameters?

Here I tried all combinations of learning rates and momentums, and drew the plots of average cross-entropy error and miss-classification error. From Figure 7.1- Figure 7.3 we can conclude the

effect of different learning rates. **Note: here the miss-classification error is not in percentage because it is easy for us to compare.**

Figure 7.1: learning rate = 0.1, momentum = 0



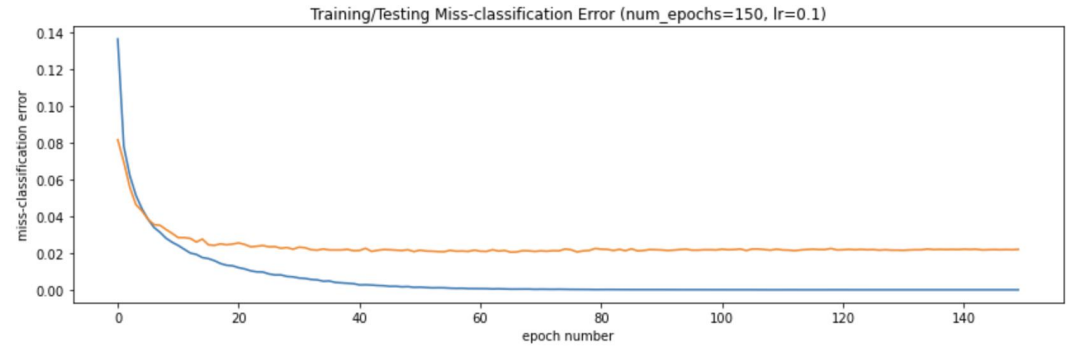
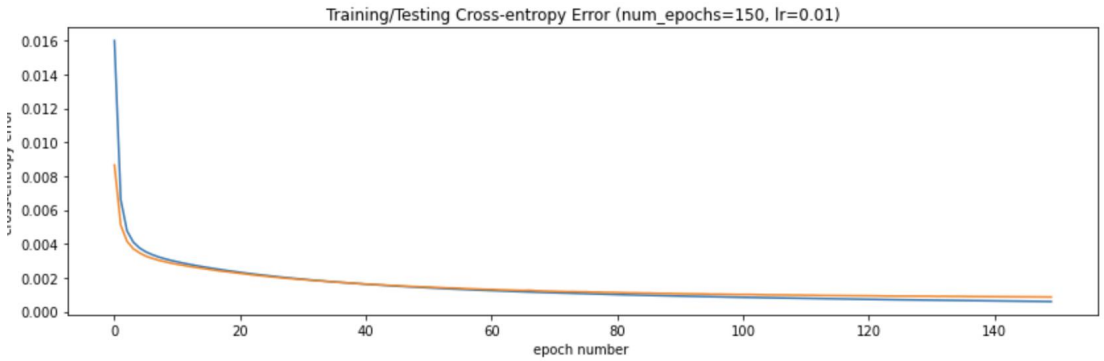


Figure 7.2: learning rate = 0.01, momentum = 0



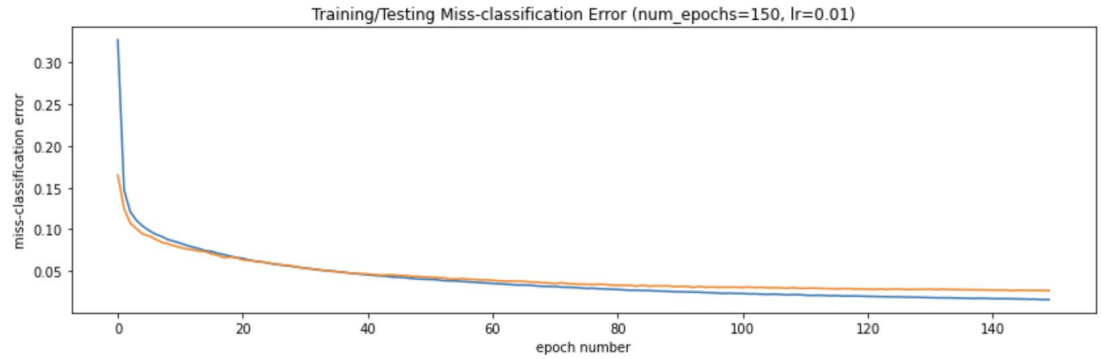
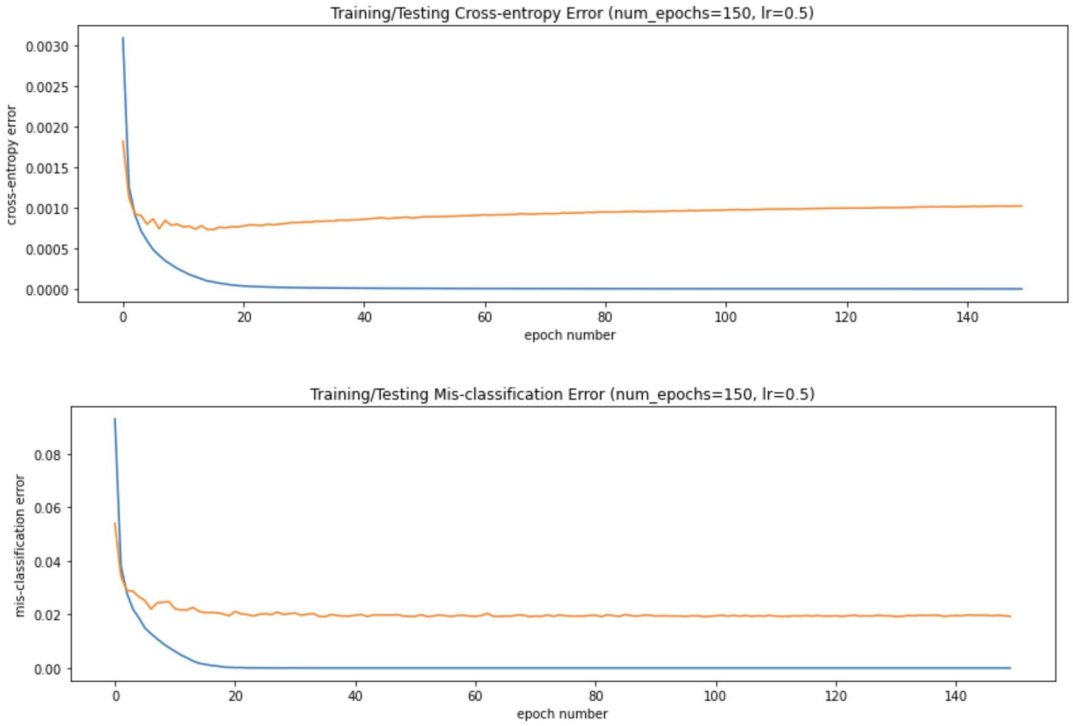


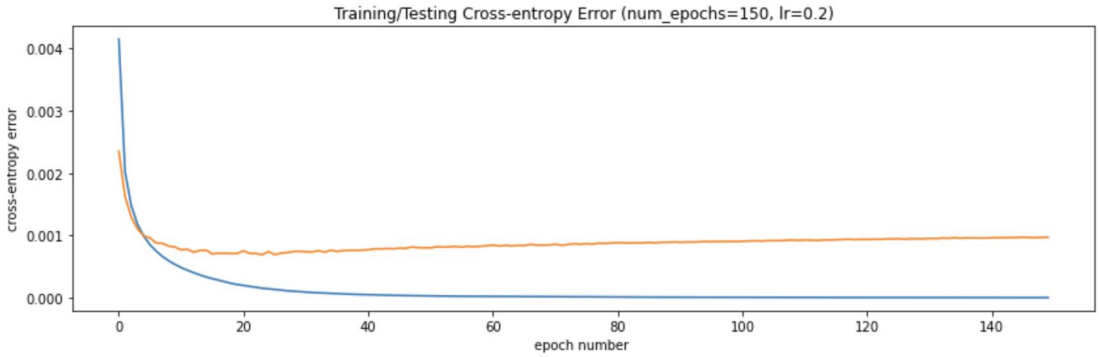
Figure 7.3: learning rate = 0.5, momentum = 0



Conclusion: For learning rate, when the learning rate is too small, like 0.01. It is unlikely to show overfitting after 150 epochs times. And if the learning rate is too big, like 0.5. And it will cause the model to converge too quickly to a suboptimal solution. Therefore, the learning rate should not be too large or too small, when learning rate = 0.1, it performs well.

From Figure 8.1- Figure 8.3 we can conclude the effect of different momentums.

Figure 8.1: learning rate = 0.2, momentum = 0



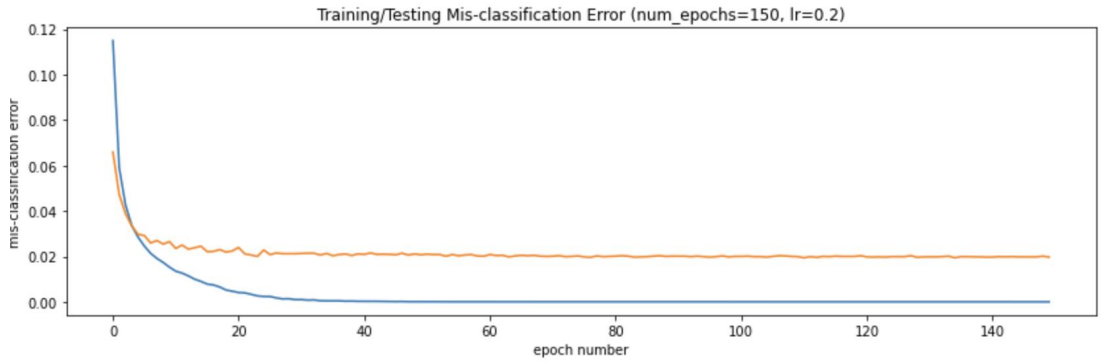


Figure 8.2: learning rate = 0.2, momentum = 0.5

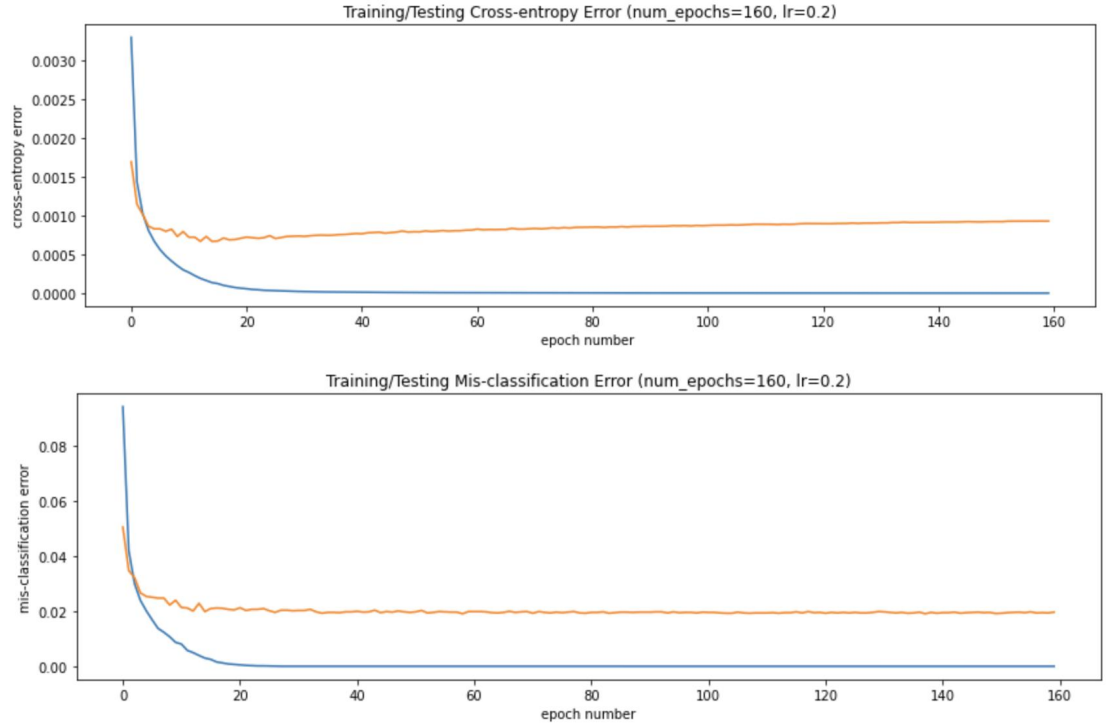
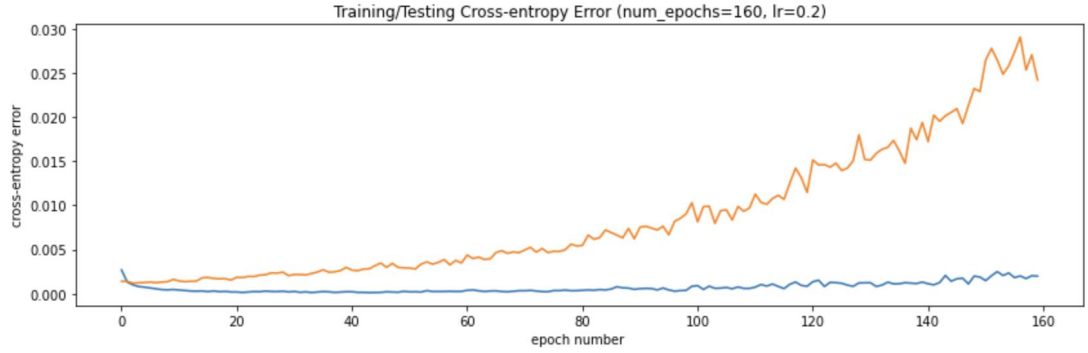
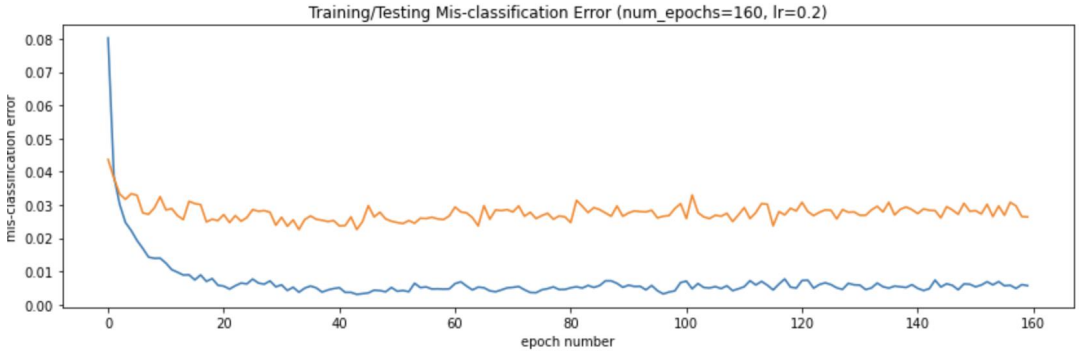


Figure 8.3: learning rate = 0.2, momentum = 0.9





Conclusion: For momentum, when the gradient keeps changing direction, momentum will smooth out the variations. This is particularly useful when the network is not well-conditioned. For most points on the surface, the gradient does not point towards the minimum, and successive steps of gradient descent can oscillate from one side to the other, progressing only very slowly to the minimum. If momentum is too low, the model will still progress slow to the minimum. A high momentum term would lead you in the wrong direction (blow up and always go in the same direction), and/or oscillate around the global minima (making you jump too far). For our case, when momentum = 0.5, it performs well.

2. A CNN i.e. with two 2-D convolutional layers

Here I create two layers neural network named *CNN,* and it contains two 2-D convolutional layers, the first layer's input channel is 1, output channel is 10 and the kernel size is 5, padding used the default value which is 0. Then the second layer's input channel is 10, output channel is 20 and the kernel size is 5. I also added two pooling layer to reduce the size of our image by half on each dimension by taking the maximum of each 2×2 window with a stride of 1. And I applied ReLU nonlinear activation. The output size is the number of classes which is 10. And the optimizer technique I used is stochastic gradient descent. And I used *torch.manual_seed* to set five different seeds. The first seed is 1, the second seed is 1234, the third seed is 2340, the fourth seed is 100, and the fifth seed is 3450. All the seeds are the random numbers I picked. Figure 9 the screenshot of the CNN model i.e. with two 2-D convolutional layers. And I train at least 150 epochs each time until the overfitting.

Figure 9: a CNN i.e. with two 2-D convolutional layers

```
torch.manual_seed(1)

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()

        self.conv_layers = nn.Sequential(
            nn.Conv2d(1, 10, kernel_size=5),
            nn.MaxPool2d(2),
            nn.ReLU(),
            nn.Conv2d(10, 20, kernel_size=5),
            nn.Dropout(),
            nn.MaxPool2d(2),
            nn.ReLU(),
        )
        self.fc_layers = nn.Sequential(
            nn.Linear(320, 50),
            nn.ReLU(),
            nn.Dropout(),
            nn.Linear(50, 10),
            nn.Softmax(dim=1)
        )

    def forward(self, x):
        x = self.conv_layers(x)
        x = x.view(-1, 320)
        x = self.fc_layers(x)
        return x

CNN = CNN()
CNN.cuda()
```

```
optimizer = torch.optim.SGD(CNN.parameters(), lr=learning_rate)
# choose loss function
loss_func = torch.nn.CrossEntropyLoss()
```

(a) Plot the average training cross-entropy error on the y-axis vs. the epoch number (x-axis). On the same figure, plot the average validation cross-entropy error function. How does the network's performance differ on the training set versus the validation set during learning? Use the plot of training and testing error curves to support your argument.

Here I applied t*orch.nn.CrossEntropyLoss()* to calculate the cross-entropy error and I created a list called *losses_train* to append each cross-entropy error of training set of each iteration. And created a list called *losses_test* to append each cross-entropy error of testing/validation set of each iteration. Figure 10 is the screenshot of the way of calculating cross-entropy error of five different seeds.

Figure 10: calculating cross-entropy error

```
n_total_steps = len(train_data)
losses_train = []
losses_test = []
errors_train = []
errors_test = []
for epoch in range(num_epochs):
    train_error = 0
    test_error = 0
    train_loss_sum = 0
    test_loss_sum = 0
    for i, (train, labels) in enumerate(train_loader):
        # forward passing
        X_train = train.cuda()
        Y_labels = labels.cuda()
        out = CNN(X_train)
        predicted_train = torch.max(out.data,1)[1].cuda().data
        batch_error = (predicted_train != Y_labels).sum()
        train_error += batch_error
        train_loss = loss_func(out, Y_labels)
        train_loss_sum += train_loss
        optimizer.zero_grad()
        train_loss.backward()
        optimizer.step()
        if (i+1) % 100 == 0:
            print (f'Epoch [{epoch+1}/{num_epochs}], Step[{i+1}/{n_total_steps}], Loss: {train_loss.item():.4f}')
    losses_train.append(train_loss_sum.item()/batch_size)
    errors_train.append(train_error.item()/batch_size)
```

```
        with torch.no_grad():
            for b,(test, labels) in enumerate(test_loader):
                X_test = test.cuda()
                Y_test = labels.cuda()
                output = CNN(X_test)
                predicted_test = torch.max(output.data,1)[1].cuda().data
                test_error += (predicted_test != Y_test).sum()
                test_loss = loss_func(output, Y_test)
                test_loss_sum += test_loss
                # correct += pred.eq(target.data.view_as(pred)).sum()
        # test_loss /= len(test_loader.dataset)
        losses_test.append(test_loss_sum.item()/batch_size)
        errors_test.append(test_error.item()/batch_size)
```

Figure 11.1-11.2 are the plots of cross-entropy error of training set and testing set of selected 2 different seeds from 5 seeds.

Figure 11.1: seed = 1's cross-entropy error

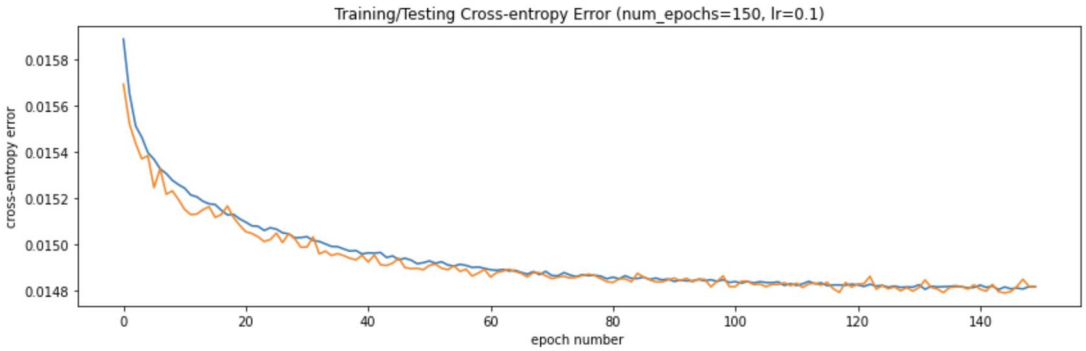
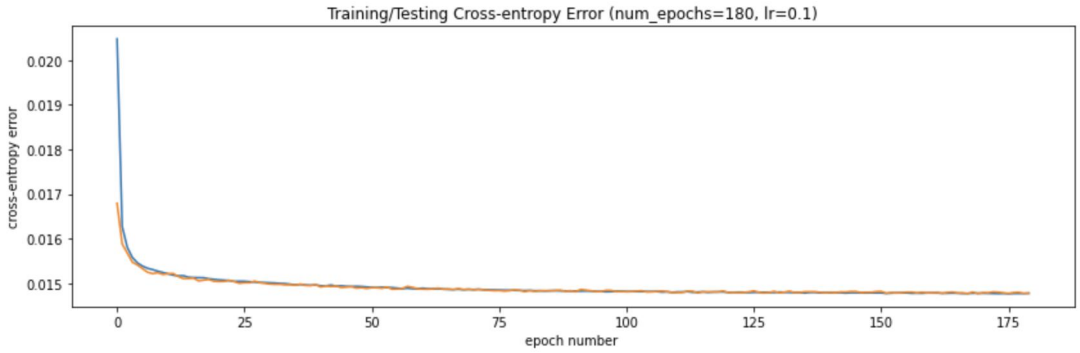


Figure 11.2: seed = 3450's cross-entropy error



Conclusion: In the plot, we can conclude the training error (the blue line) decreases to a sightly larger degree in the first 10 epoch times from 0.02 to 0.015, and then it decreases more stable after 20 epoch times, and the error decreases to almost 0 after 150-time iterations. The testing error (the yellow line) is also declining. And the testing error starts from 0.0165 which is less than 0.02 of the training error initially, it might due to the model has been trained well after the training error. And the updated model can train the testing set well.

(b) We could implement an alternative performance measure to the cross entropy, the mean miss-classification error. Plot the classification error vs. number of epochs, for both training and testing. Do you observe a different behavior compared to the behavior of the cross-entropy error function?

I calculated the miss-classification error also in problem (a) by judging whether the predicted_train = torch.max(out.data,1)[1] is the same as labels_train or not. Figure 12 is the main part of the miss-classification error, which has been concluded in Figure 10 as well.

Figure 12: main part of calculating miss-classification error

```
for epoch in range(num_epochs):
    train_error = 0
    test_error = 0
    train_loss_sum = 0
    test_loss_sum = 0
    for i, (train, labels) in enumerate(train_loader):
        # forward passing
        X_train = train.cuda()
        Y_labels = labels.cuda()
        out = CNN(X_train)
        predicted_train = torch.max(out.data,1)[1].cuda().data
        batch_error = (predicted_train != Y_labels).sum()
        train_error += batch_error
        train_loss = loss_func(out, Y_labels)
        train_loss_sum += train_loss
        optimizer.zero_grad()
        train_loss.backward()
        optimizer.step()
        if (i+1) % 100 == 0:
            print (f'Epoch [{epoch+1}/{num_epochs}], Step[{i+1}/{n_total_steps}], Loss: {train_loss.item():.4f}')
    losses_train.append(train_loss_sum.item()/batch_size)
    errors_train.append(train_error.item()/batch_size)

    with torch.no_grad():
        for b,(test, labels) in enumerate(test_loader):
            X_test = test.cuda()
            Y_test = labels.cuda()
            output = CNN(X_test)
            predicted_test = torch.max(output.data,1)[1].cuda().data
            test_error += (predicted_test != Y_test).sum()
```

Figure 13.1-13.2 are the plots of miss-classification error of training set and testing set of selected 2 different seeds from 5 seeds. **Note: here the miss-classification error is not in percentage because it is easy for us to compare.**

Figure 13.1: seed = 1's miss-classification error

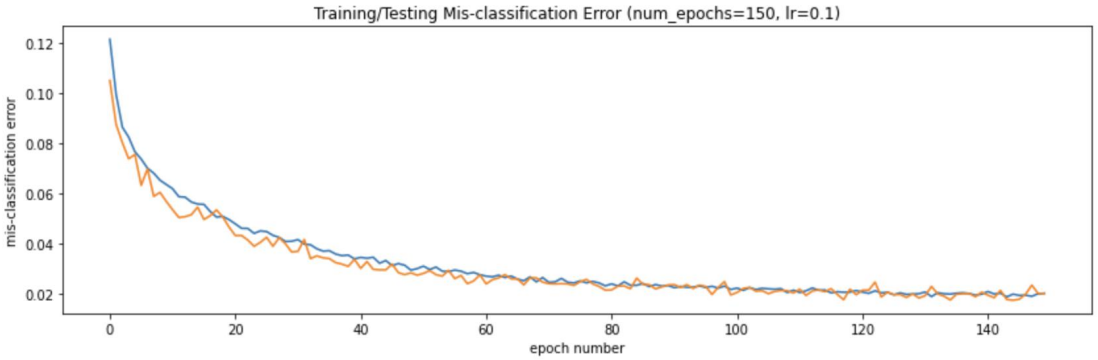
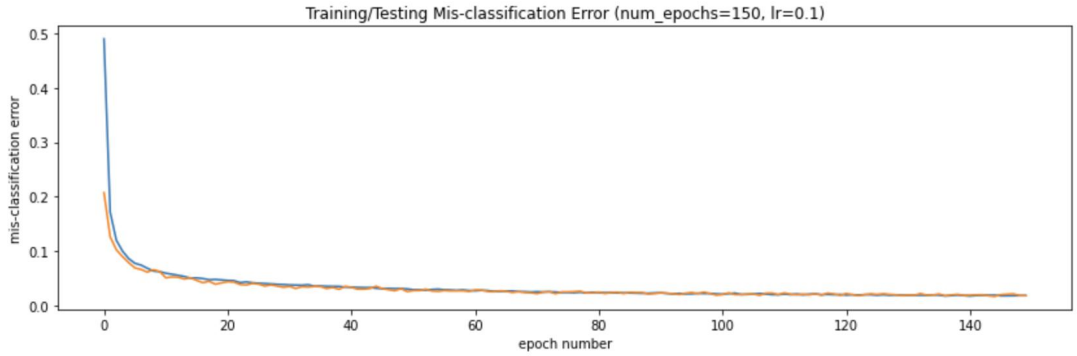


Figure 13.2: seed = 3450's miss-classification error

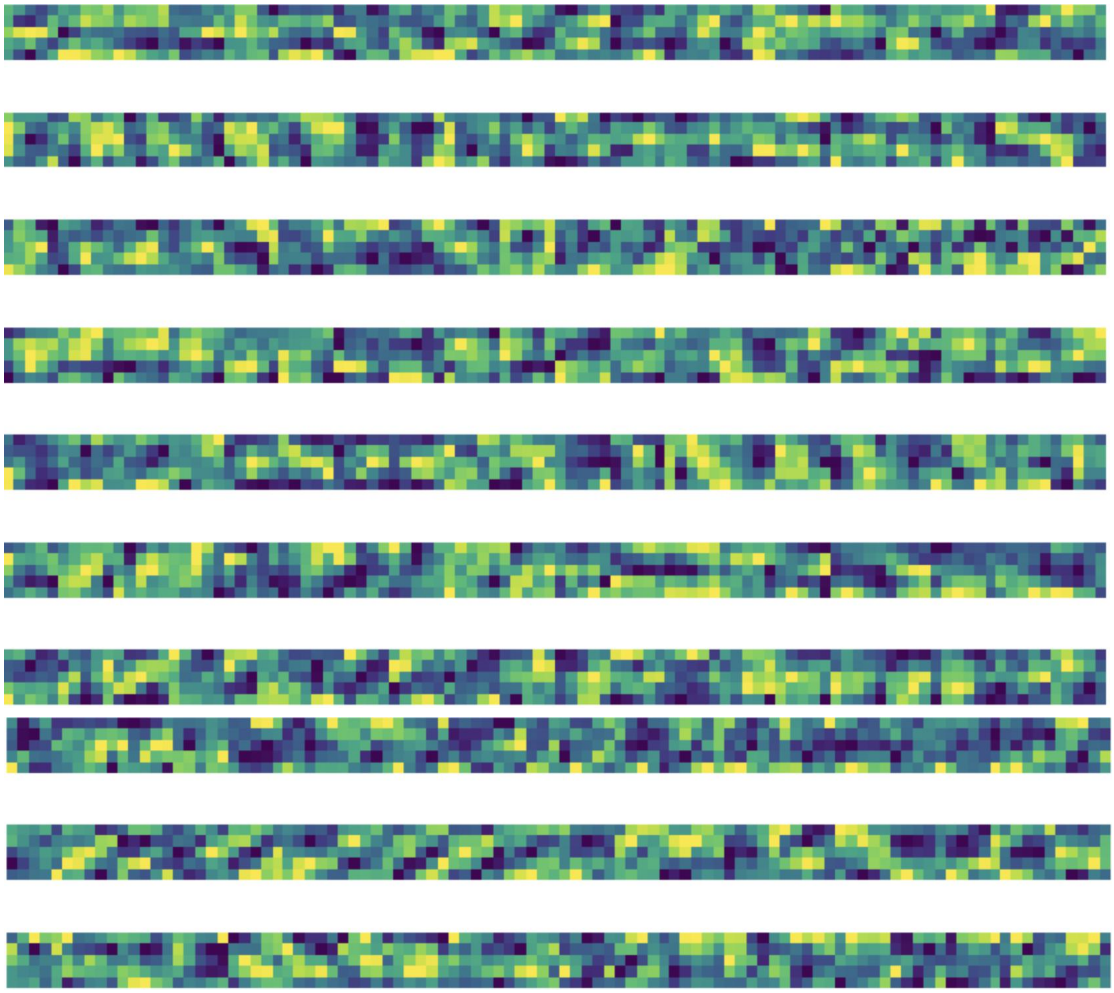


Conclusion: What can be concluded from the plots is that after 150 times iterations there is overfitting. And the training miss-classification error declines from 0.5 to almost 0.01 after iterations, and the validation testing error changes from 0.2 to 0.02. Since the testing error shows overfitting, the neural network model can train the dataset well. And there are not so much different behaviors between cross-entropy error and miss-classification error, but due to the different methods of calculating errors, the miss-classification error is a bit larger.

(c) Visualize your best results of the learned W as one hundred 28×28 images (plot all filters as one image, as we have seen in class). Do the learned features exhibit any structure?

Here I compare with the five plots in the (a) part, there are almost the same amount five plots. The fifth trial with manual_seed(3450) and num_epochs = 150, batch_size = 100, learning rate = 0.1. The training error is more close to 0, and the testing error shows the overfitting. Therefore, I choose this seed. Figure 14 shows the plot of filters of the second layer convolutional neural network.

Figure 14: plot of filters of the second layer convolutional neural network.



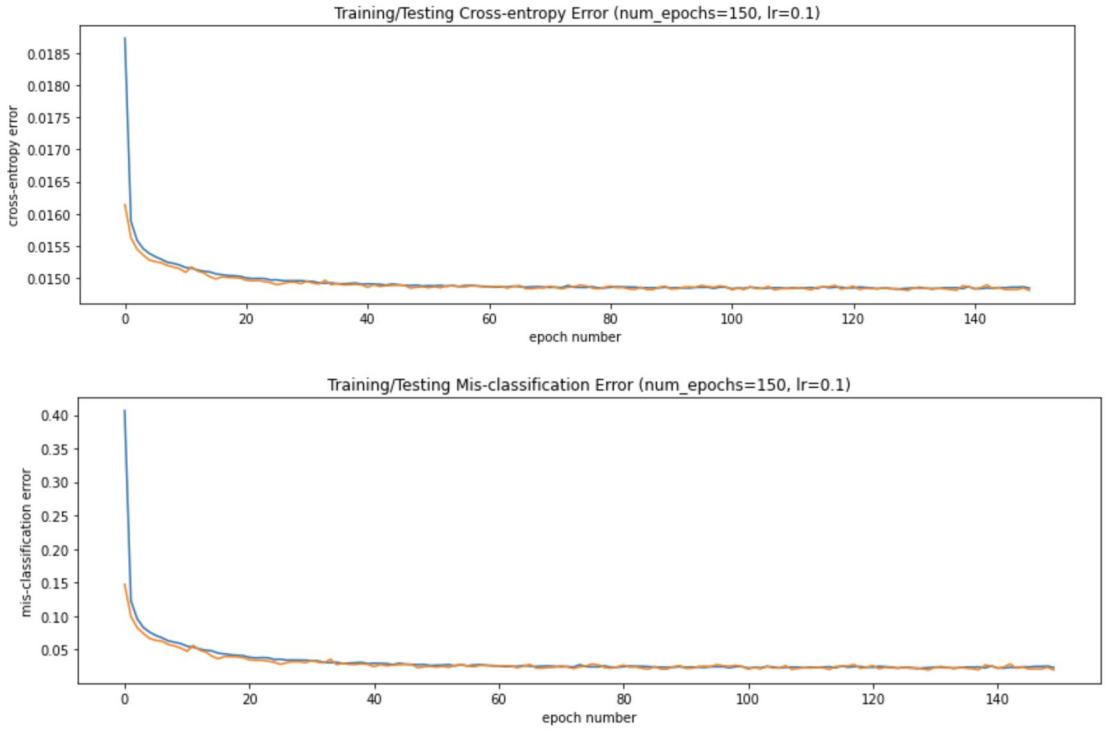
Conclusion: The plot creates a figure with 100 images, one row for each filter and one column for each channel.The dark squares indicate small or inhibitory weights and the light squares represent large or excitatory weights. Using this intuition, we can see that the filters on the first row detect a gradient from light in the top left to dark in the bottom right.

(d) Try different values of the learning rate. How do momentum and learning rate affect convergence rate? How would you choose the best value of these parameters?

Conclusion: Here I tried some combinations of learning rates and momentums, and drew the plots of average cross-entropy error and miss-classification error. The code explicitly introduces different combinations of learning rates and momentums. And the effect of different learning rates and momentums is the same as the one-layer simple neural network, which has already introduced before. If learning rate is too small, it is unlikely to show overfitting after 150 epochs times. And if the learning rate is too big, it will cause the model to converge too quickly to a suboptimal solution. If momentum is too low, the model will still progress slow to the minimum. A high momentum term would lead you in the wrong direction (blow up and always go in the same direction), and/or oscillate around the global minima (making you jump too far).

Figure 15 shows the best combinations of learning rates and momentums, which learning rate = 0.1 and momentum = 0.5. **Note: here the miss-classification error is not in percentage because it is easy for us to compare.**

Figure 15: the best combinations: learning rate = 0.1 and momentum = 0.5



3. Favorite deep learning architecture
introducing batch normalization, introducing dropout in training

Redo part 3(a) - 3(d) with your favorite deep learning architecture (e.g., introducing batch normalization, introducing dropout in training) to beat the performance of SVM with Gaussian Kernel, i.e., to have a test error rate lower than 1.4%.

Here I create three layers neural network named *Net,* and it contains three 2-D convolutional layers, the first layer's input channel is 1, output channel is 32 and the kernel size is 3, the stride is 1. Then the second layer's input channel is 32, output channel is 64 and the kernel size is 3, the stride is 1. The third layer's input channel is 64, output channel is 128, and the stride is also 1. I added Batch Normalization to standardize the inputs to a layer for each batch and Dropout rate prevent overfitting. And I applied ReLU nonlinear activation. The output size is the number of classes which is 10. And the optimizer technique I used is stochastic gradient descent. And I used *torch.manual_seed* to set five different seeds. The first seed is 1, the second seed is 1234, the third seed is 2340, the fourth seed is 100, and the fifth seed is 3450. All the seeds are the random numbers I picked. Figure 16 the screenshot of the CNN model i.e. with three 2-D convolutional layers. And I train at least 150 epochs each time until the overfitting.

Figure 16: CNN model i.e. with three 2-D convolutional layers.

```
torch.manual_seed(1)
class Net(nn.Module):
  def __init__(self):
    super(Net,self).__init__()
    self.conv1=nn.Conv2d(1,32,3,1)
    self.conv1_bn=nn.BatchNorm2d(32)

    self.conv2=nn.Conv2d(32,64,3,1)
    self.conv2_bn=nn.BatchNorm2d(64)

    self.conv3=nn.Conv2d(64,128,3,1)
    self.conv3_bn=nn.BatchNorm2d(128)

    self.dropout1=nn.Dropout(0.25)

    self.fc1=nn.Linear(15488,128)
    self.fc1_bn=nn.BatchNorm1d(128)

    self.fc2=nn.Linear(128,64)
    self.fc3=nn.Linear(64,10)
```

```
    def forward(self,x):
      x=self.conv1(x)
      x=F.relu(self.conv1_bn(x))

      x=self.conv2(x)
      x=F.relu(self.conv2_bn(x))

      x=self.conv3(x)
      x=F.relu(self.conv3_bn(x))

      x=F.max_pool2d(x,2)
      x=self.dropout1(x)

      x=torch.flatten(x,1)

      x=self.fc1(x)
      x=F.relu(self.fc1_bn(x))

      x=self.fc2(x)
      x=self.fc3(x)
      output=F.log_softmax(x,dim=1)
      return output
  net = Net()
  net.cuda()
```

```
# optimizing methods
optimizer = torch.optim.SGD(net.parameters(), lr=learning_rate)
# choose loss function
loss_func = torch.nn.CrossEntropyLoss()
```

(a) Plot the average training cross-entropy error on the y-axis vs. the epoch number (x-axis). On the same figure, plot the average validation cross-entropy error function. How does the network's performance differ on the training set versus the validation set during learning? Use the plot of training and testing error curves to support your argument.

Here I applied t*orch.nn.CrossEntropyLoss()* to calculate the cross-entropy error and I created a list called *losses_train* to append each cross-entropy error of training set of each iteration. And created a list called *losses_test* to append each cross-entropy error of testing/validation set of each iteration. Figure 17 is the screenshot of the way of calculating cross-entropy error of five different seeds.

Figure 17: calculating cross-entropy error

```
n_total_steps = len(train_data)
losses_train = []
losses_test = []
errors_train = []
errors_test = []
for epoch in range(num_epochs):
    train_error = 0
    test_error = 0
    train_loss_sum = 0
    test_loss_sum = 0
    for i, (train, labels) in enumerate(train_loader):
        X_train = train.cuda()
        Y_labels = labels.cuda()
        # forward passing
        out = net(X_train)
        predicted_train = torch.max(out.data,1)[1].cuda().data
        batch_error = (predicted_train != Y_labels).sum()
        train_error += batch_error
        train_loss = loss_func(out, Y_labels)
        train_loss_sum += train_loss
        optimizer.zero_grad()
        train_loss.backward()
        optimizer.step()
        if (i+1) % 100 == 0:
            print (f'Epoch [{epoch+1}/{num_epochs}], Step[{i+1}/{n_total_steps}], Loss: {train_loss.item():.4f}')
    losses_train.append(train_loss_sum.item()/batch_size)
    errors_train.append(train_error.item()/batch_size)
```

```
        with torch.no_grad():
            for b,(test, labels) in enumerate(test_loader):
                X_test = test.cuda()
                Y_test = labels.cuda()
                output = net(X_test)
                predicted_test = torch.max(output.data,1)[1].cuda().data
                test_error += (predicted_test != Y_test).sum()
                test_loss = loss_func(output, Y_test)
                test_loss_sum += test_loss
                # correct += pred.eq(target.data.view_as(pred)).sum()
        # test_loss /= len(test_loader.dataset)
        losses_test.append(test_loss_sum.item()/batch_size)
        errors_test.append(test_error.item()/batch_size)
```

Figure 18.1-18.2 are the plots of cross-entropy error of training set and testing set of selected 2 different seeds from 5 seeds.

Figure 18.1: seed = 1's cross-entropy error

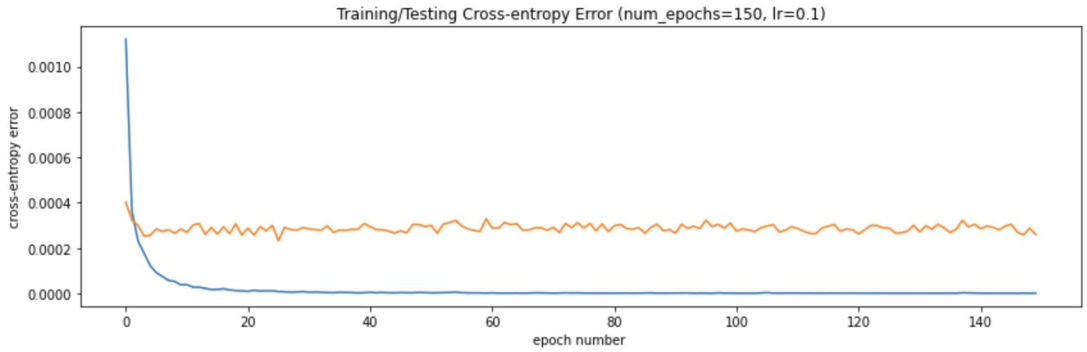
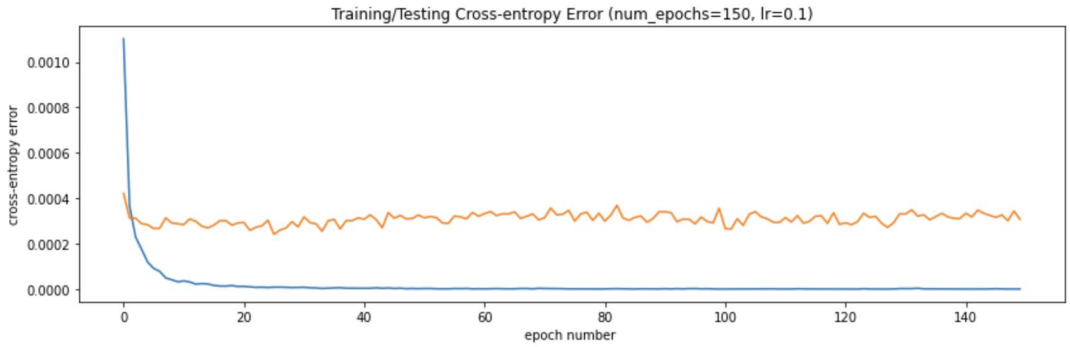


Figure 18.2: seed = 100's cross-entropy error



Conclusion: What can be concluded from the plots is that after 150 times iterations there is overfitting. And the training cross-entropy error declines from 0.0010 to almost 0 after iterations, and the validation cross-entropy error changes from 0.0004 to 0.0002. Since the testing error shows overfitting, the neural network model can train the dataset well. The training error is larger than the testing error initially which makes sense, since it might due to the model has been trained well after the training error. And the updated model can train the testing set well.

(b) We could implement an alternative performance measure to the cross entropy, the mean miss-classification error. Plot the classification error vs. number of epochs, for both training and testing. Do you observe a different behavior compared to the behavior of the cross-entropy error function?

I calculated the miss-classification error also in problem (a) by judging whether the predicted_train = torch.max(out.data,1)[1] is the same as labels_train or not. Figure 19 is the main part of the miss-classification error, which has been concluded in Figure 17 as well.

Figure 19: main part of calculating miss-classification error

```
    for i, (train, labels) in enumerate(train_loader):
        X_train = train.cuda()
        Y_labels = labels.cuda()
        # forward passing
        out = net(X_train)
        predicted_train = torch.max(out.data,1)[1].cuda().data
        batch_error = (predicted_train != Y_labels).sum()
        train_error += batch_error
        train_loss = loss_func(out, Y_labels)
        train_loss_sum += train_loss
        optimizer.zero_grad()
        train_loss.backward()
        optimizer.step()
        if (i+1) % 100 == 0:
            print (f'Epoch [{epoch+1}/{num_epochs}], Step[{i+1}/{n_total_steps}], Loss: {train_loss.item():.4f}')
    losses_train.append(train_loss_sum.item()/batch_size)
    errors_train.append(train_error.item()/batch_size)

    with torch.no_grad():
        for b,(test, labels) in enumerate(test_loader):
            X_test = test.cuda()
            Y_test = labels.cuda()
            output = net(X_test)
            predicted_test = torch.max(output.data,1)[1].cuda().data
            test_error += (predicted_test != Y_test).sum()
            test_loss = loss_func(output, Y_test)
            test_loss_sum += test_loss
            # correct += pred.eq(target.data.view_as(pred)).sum()
    # test_loss /= len(test_loader.dataset)
    losses_test.append(test_loss_sum.item()/batch_size)
    errors_test.append(test_error.item()/batch_size)
```

Figure 20.1-20.2 are the plots of miss-classification error of training set and testing set of selected 2 different seeds from 5 seeds. **Note: here the miss-classification error is not in percentage because it is easy for us to compare.**

Figure 20.1: seed = 1's cross-entropy error

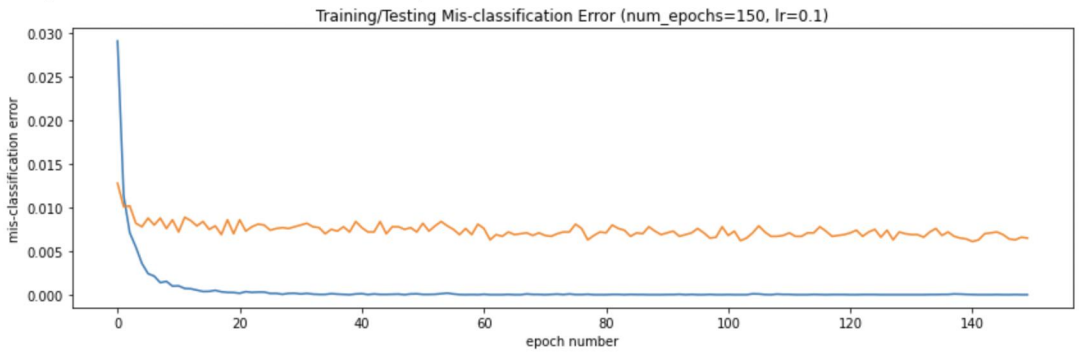
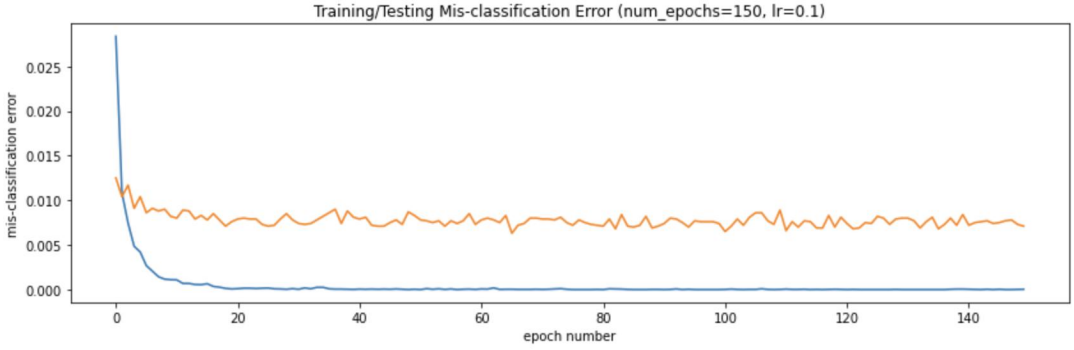


Figure 20.2: seed = 100's cross-entropy error

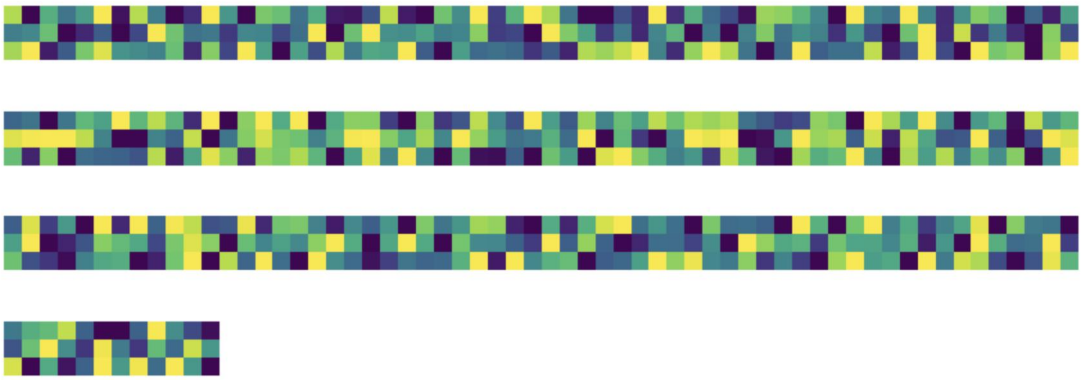


Conclusion: What can be concluded from the plots is that after 150 times iterations there is overfitting. And the training miss-classification error declines from 0.025 to almost 0 after

iterations, and the validation testing error to 0.01 which is less than 1.4% of **SVM with Gaussian Kernel**. Since the testing error shows overfitting, the neural network model can train the dataset well. And there are not so much different behaviors between cross-entropy error and miss-classification error, but due to the different methods of calculating errors, the miss-classification error is a bit larger.

(c) Visualize your best results of the learned W as one hundred 28×28 images (plot all filters as one image, as we have seen in class). Do the learned features exhibit any structure?

Figure 21: plot of filters of the second layer convolutional neural network.



Conclusion: The plot creates a figure with second filter layer images, one row for each filter and one column for each channel.The dark squares indicate small or inhibitory weights and the light squares represent large or excitatory weights. Using this intuition, we can see that the filters on the first row detect a gradient from light in the top left to dark in the bottom right.

(d) Try different values of the learning rate. How do momentum and learning rate affect convergence rate? How would you choose the best value of these parameters?

Conclusion: Here I tried some combinations of learning rates and momentums, and drew the plots of average cross-entropy error and miss-classification error. The code explicitly introduces different combinations of learning rates and momentums. And the effect of different learning rates and momentums is the same as the one-layer simple neural network, which has already introduced before. Figure 22 - Figure 23 shows the best combinations of learning rates and momentums, which learning rate = 0.01 and momentum = 0.5. And learning rate = 0.1 is also a good choice. **Note: here the miss-classification error is not in percentage because it is easy for us to compare.**

Figure 22: the best combinations learning rate = 0.01 and momentum = 0.5

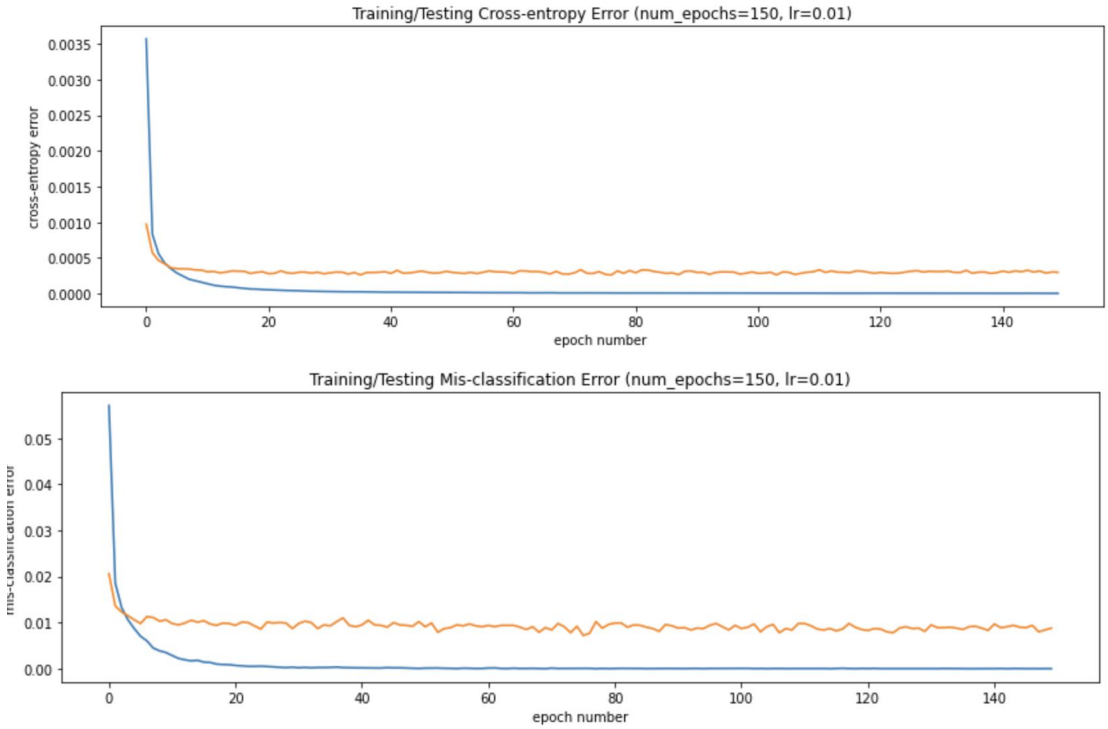
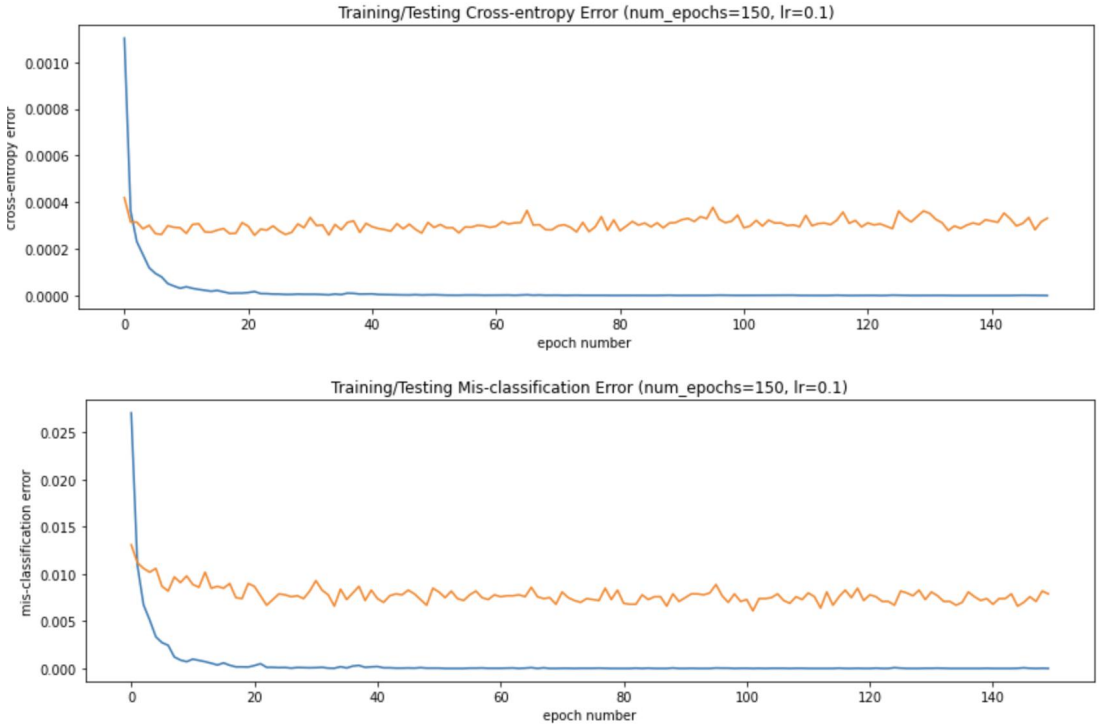


Figure 23: the best combinations learning rate = 0.1 and momentum = 0.5



Part 5: More about Deep Learning

1. Familiarize with the data

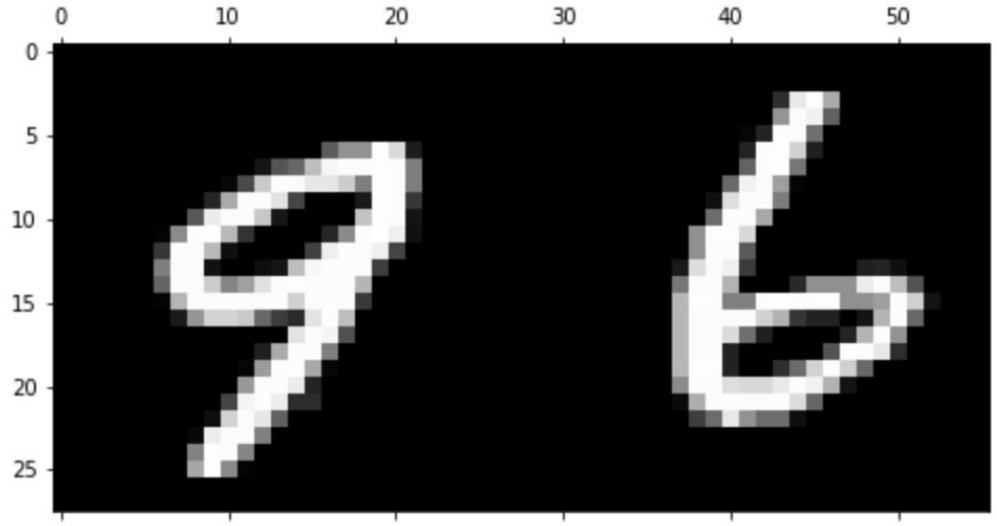
As a warm up question, load the data and plot a few examples. Decide if the pixels were scanned out in row-major or column-major order. What is the relationship between the 2 digits and the last coordinate of each line?

After loading the data, and visualizing some examples, it is essential to know that the pixels were scanned out in row-major order. And from the three examples we plot, Figure 24 is one of the selected examples, the last coordinate of each line is the sum of the 2 digits.

Figure 24: visualizing the digit

```
print(train.iloc[100][1568])
plt.matshow(pixel_mat(100), cmap=plt.cm.gray)
```

```
15.0
<matplotlib.image.AxesImage at 0x7f2f91fbfad0>
```



2. Data Pre-processing

Since txt file is not suitable to use Pytorch to train model later, here we need to process it and transform it to the torch tensors data and then applied *torch.utils.data.TensorDataset* to load the data. Figure 25 is the screenshot of processing data.

Figure 25: processing data

```
train_x_reshape = np.array(train.iloc[:,:1568]).reshape(20000,28,56)
test_x_reshape = np.array(test.iloc[:,:1568]).reshape(5000,28,56)
val_x_reshape = np.array(val.iloc[:,:1568]).reshape(5000,28,56)
```

```
train_y_reshape = np.array(train.iloc[:,1568])
test_y_reshape = np.array(test.iloc[:,1568])
val_y_reshape = np.array(val.iloc[:,1568])
```

```
train_x_tensor = torch.tensor(train_x_reshape)
test_x_tensor = torch.tensor(test_x_reshape)
val_x_tensor = torch.tensor(val_x_reshape)
```

```
train_y_tensor = torch.tensor(train_y_reshape)
test_y_tensor = torch.tensor(test_y_reshape)
val_y_tensor = torch.tensor(val_y_reshape)
```

```
train_dataset = torch.utils.data.TensorDataset(train_x_tensor, train_y_tensor)
test_dataset = torch.utils.data.TensorDataset(test_x_tensor, test_y_tensor)
val_dataset = torch.utils.data.TensorDataset(val_x_tensor, val_y_tensor)
```

```
train_loader = torch.utils.data.DataLoader(dataset = train_dataset,batch_size = batch_size,shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset = test_dataset,batch_size = batch_size,shuffle=False)
val_loader = torch.utils.data.DataLoader(dataset = val_dataset,batch_size = batch_size,shuffle=False)
```

3. **Model training**

Repeat part 3(a) - 3(d) with at least two of your favorite deep learning architecture (e.g., introducing batch normalization, introducing dropout in training) with respect to with train.txt, val.txt and test.txt.

Model 1: A three 2-D convolutional layers of CNN mode

Here I create three layers neural network named *cnn,* and it contains three 2-D convolutional layers, the first layer's input channel is 1, output channel is 36 and the kernel size is 3, the stride is 1. Then the second layer's input channel is 32, output channel is 72 and the kernel size is 3, the stride is 1. The third layer's input channel is 72, output channel is 144, and the stride is also 1. And I applied ReLU nonlinear activation. The output size is the number of classes which is 19 (0-18). And the optimizer technique I used is stochastic gradient descent. And I used *torch.manual_seed* to set five different seeds. The first seed is 1, the second seed is 1234, the third seed is 2340, the fourth seed is 100, and the fifth seed is 3450. All the seeds are the random numbers I picked. Figure 26 the screenshot of the CNN model i.e. with three 2-D convolutional layers. And I train at least 150 epochs each time until the overfitting.

Figure 26: CNN model i.e. with three 2-D convolutional layers.

```
torch.manual_seed(1)
class cnn(torch.nn.Module):
    def __init__(self):
      super(cnn, self).__init__()
      self.conv1 = nn.Sequential(
        nn.Conv2d(1, 36, 3, 1,1),
        nn.ReLU(),
        nn.MaxPool2d(2)
        )
      self.conv2 = nn.Sequential(
        nn.Conv2d(36, 72, 3, 1,1),
        nn.ReLU(),
        nn.MaxPool2d(2)
        )
      self.conv3 = nn.Sequential(
        nn.Conv2d(72, 144, 3, 1,1),
        nn.ReLU(),
        nn.Flatten()
        )
      self.l1 = nn.Linear(144*7*14, 512)
      self.l2 = nn.Linear(512, 19)
    def forward(self, X):
      X = self.conv1(X)
      X = self.conv2(X)
      X = self.conv3(X)
      X = nn.functional.relu(self.l1(X))
      X = self.l2(X)
      return X
net = cnn()
net.cuda()
```

```
# optimizing methods
optimizer = torch.optim.SGD(net.parameters(), lr=learning_rate)
# choose loss function
loss_func = torch.nn.CrossEntropyLoss()
```

(a) Plot the average training cross-entropy error on the y-axis vs. the epoch number (x-axis). On the same figure, plot the average validation cross-entropy error function. How does the network's performance differ on the training set versus the validation set during learning? Use the plot of training and testing error curves to support your argument.

Here I applied t*orch.nn.CrossEntropyLoss()* to calculate the cross-entropy error and I created a list called *losses_train* to append each cross-entropy error of training set of each iteration. And created a list called *losses_test* to append each cross-entropy error of validation set of each iteration. Figure 27 is the screenshot of the way of calculating cross-entropy error of five different seeds.

Figure 27: calculating cross-entropy error of five different seeds

```
n_total_steps = len(train_data)
losses_train = []
losses_val = []
errors_train = []
errors_val = []
for epoch in range(num_epochs):
    train_error = 0
    val_error = 0
    train_loss_sum = 0
    val_loss_sum = 0
    for i, (train,labels) in enumerate(train_loader):
        # forward passing
        x_train = train.float().reshape(100,1,28,56)
        X_train = x_train.cuda()
        Y_labels = labels.cuda()
        out = net(X_train)
        predicted_train = torch.max(out.data,1)[1].cuda().data
        batch_error = (predicted_train != Y_labels).sum()
        train_error += batch_error
        train_loss = loss_func(out, Y_labels)
        train_loss_sum += train_loss
        optimizer.zero_grad()
        train_loss.backward()
        optimizer.step()
        if (i+1) % 100 == 0:
            print (f'Epoch [{epoch+1}/{num_epochs}], Step[{i+1}/{n_total_steps}], Loss: {train_loss.item():.4f}')
    losses_train.append(train_loss_sum.item()/batch_size)
    errors_train.append(train_error.item()/batch_size)
```

```
    with torch.no_grad():
        for b, (test,labels) in enumerate(val_loader):
            x_test = test.float().reshape(100,1,28,56)
            X_test = x_test.cuda()
            Y_test = labels.cuda()
            output = net(X_test)
            predicted_test = torch.max(output.data,1)[1].cuda().data
            val_error += (predicted_test != Y_test).sum()
            val_loss = loss_func(output, Y_test)
            val_loss_sum += val_loss
            # correct += pred.eq(target.data.view_as(pred)).sum()
    # test_loss /= len(test_loader.dataset)
    losses_val.append(val_loss_sum.item()/batch_size)
    errors_val.append(val_error.item()/batch_size)
```

Figure 28.1-28.2 are the plots of cross-entropy error of training set and testing set of selected 2 different seeds from 5 seeds.

Figure 28.1: seed = 1234's cross-entropy error

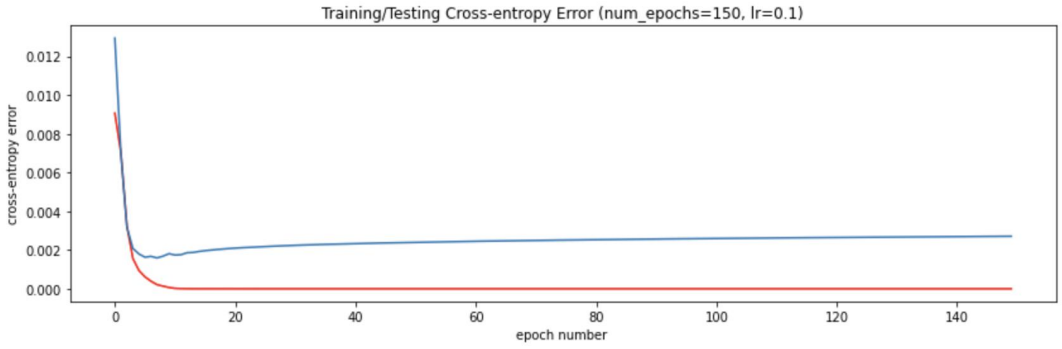
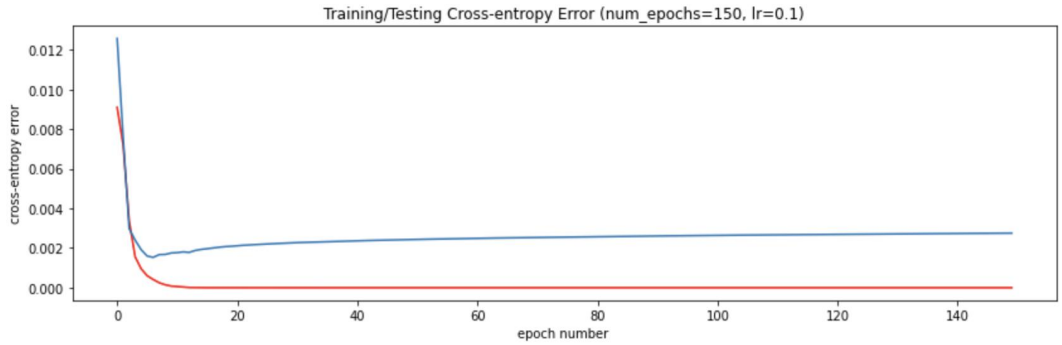


Figure 28.2: seed = 3450's cross-entropy error



Conclusion: In the plot, we can conclude the training error (the red line) decreases to a sightly larger degree in the first 10 epoch times from 0.012 to 0.002, and then it decreases more stable after 20 epoch times, and the error decreases to almost 0 after 150-time iterations. The testing error (the blue line) is also declining. And the testing error starts from 0.009 which is less than 0.012 of the training error initially, and decreases to 0.002.

(b) Plot the classification error vs. number of epochs, for both training and testing. Do you observe a different behavior compared to the behavior of the cross-entropy error function?

Figure 29.1-29.2 are the plots of miss-classification error of training set and testing set of selected 2 different seeds from 5 seeds. **Note: here the miss-classification error is not in percentage because it is easy for us to compare.**

Figure 29.1: seed = 1234's miss-classification error

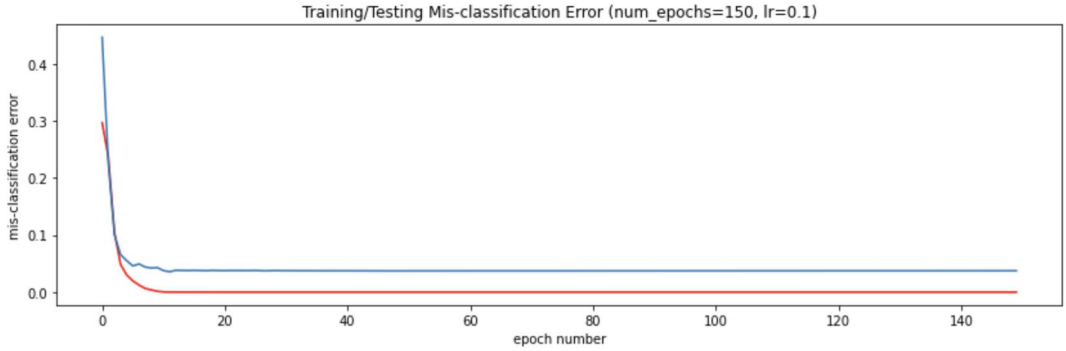
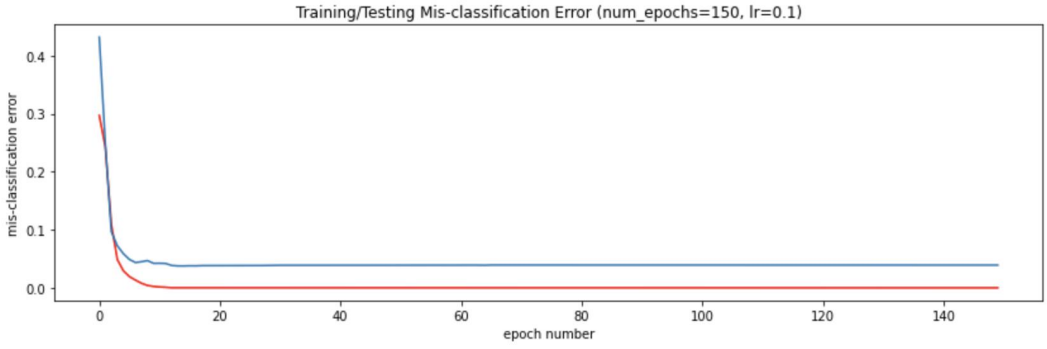


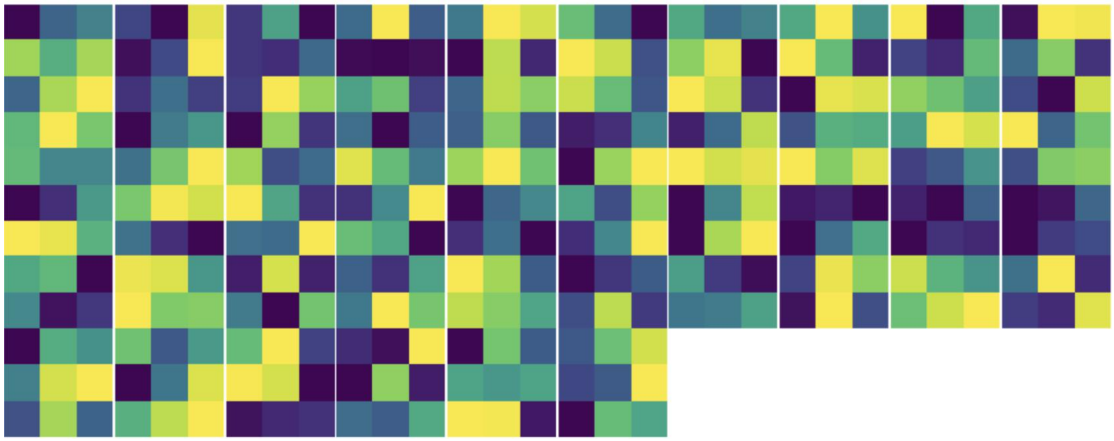
Figure 29.2: seed = 3450's miss-classification error



Conclusion: What can be concluded from the plots is that after 150 times iterations there is overfitting. And the training miss-classification error declines from 0.4 to almost 0 after iterations, and the validation testing error decreases from 0.3 to 0.05. Since the testing error shows overfitting, the fist model can train the dataset well. And there are not so much different behaviors between cross-entropy error and miss-classification error, but due to the different methods of calculating errors, the miss-classification error is a bit larger.

(c) Visualize your best results of the learned W as one hundred 28×28 images (plot all filters as one image, as we have seen in class). Do the learned features exhibit any structure?

Figure 30: plot of filters of the second layer convolutional neural network.



Conclusion: The plot creates a figure with second filter layer images, one row for each filter and one column for each channel.The dark squares indicate small or inhibitory weights and the light squares represent large or excitatory weights. Using this intuition, we can see that the filters on the first row detect a gradient from light in the top left to dark in the bottom right.

(d) Try different values of the learning rate. How do momentum and learning rate affect convergence rate? How would you choose the best value of these parameters?

Conclusion: Here I tried some combinations of learning rates and momentums, and drew the plots of average cross-entropy error and miss-classification error. The code explicitly introduces different combinations of learning rates and momentums. And the effect of different learning rates and momentums is the same as the one-layer simple neural network, which has already introduced before. Figure 31 - Figure 32 shows different learning rates and momentums effect. Here we will choose learning rate = 0.1 and momentum = 0 or 0.5 is the best case. **Note: here the miss-classification error is not in percentage because it is easy for us to compare.**

Figure 31: the best combinations learning rate = 0.1 and momentum = 0.5

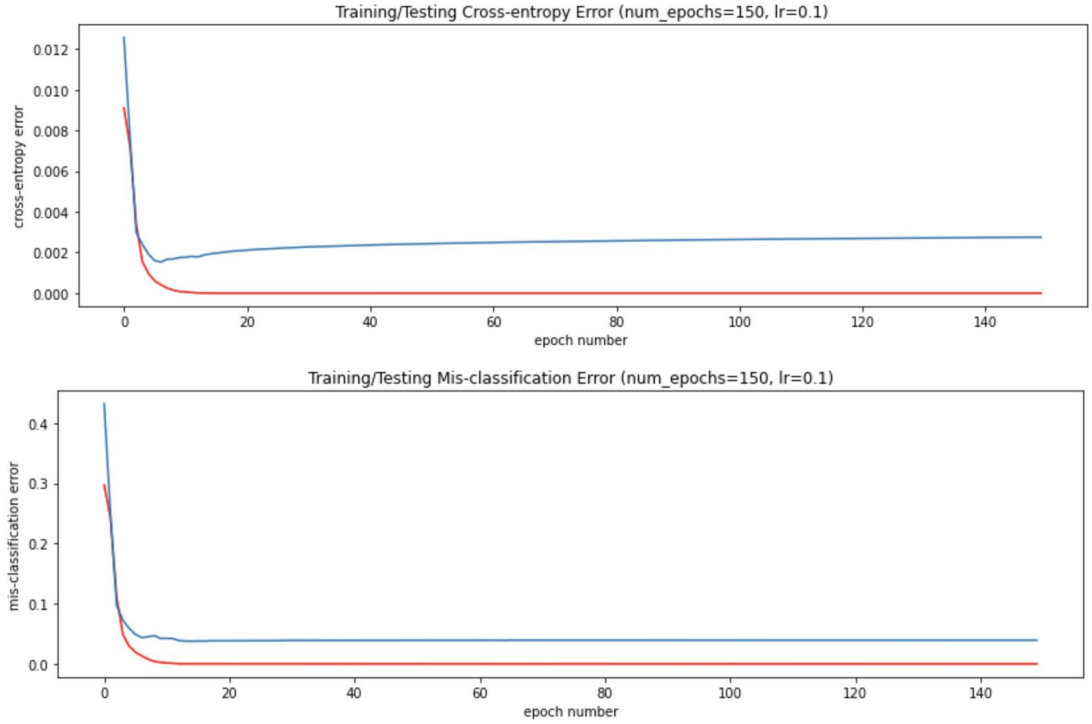
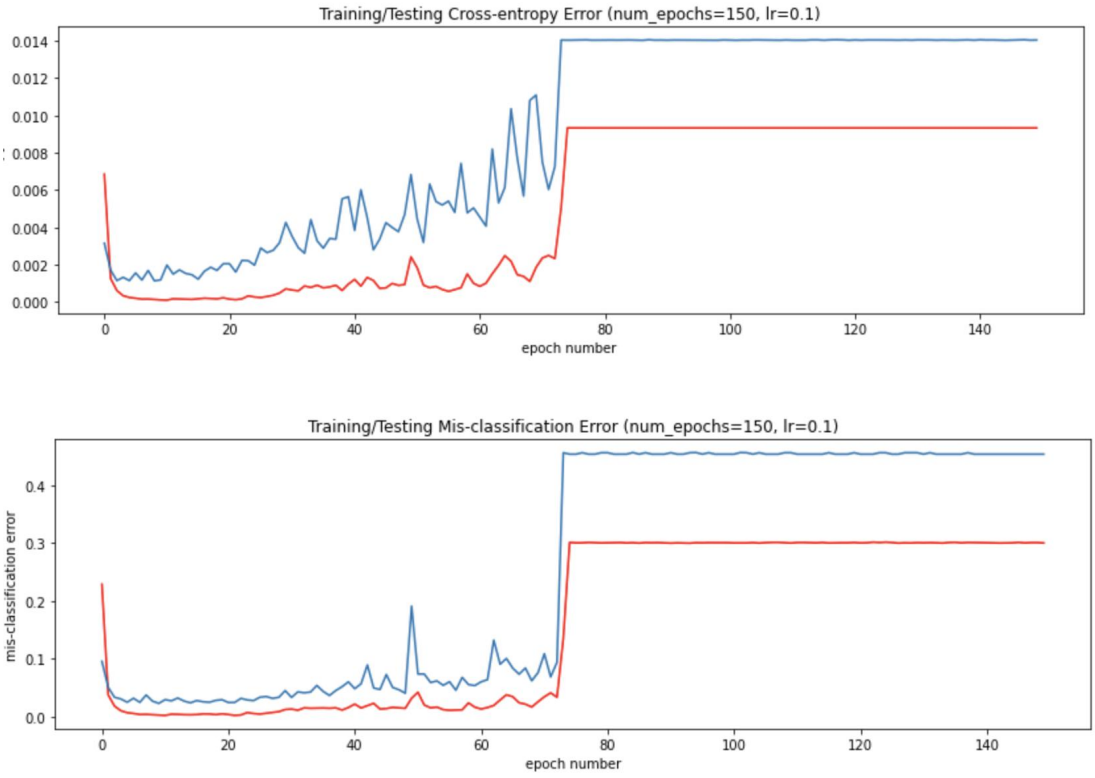


Figure 32: learning rate = 0.1 and momentum = 0.9 (to compare)



Model 2: A three 2-D convolutional layers of CNN model with BatchNorm and Dropout

Here I create three layers neural network named *Net,* and it contains three 2-D convolutional layers, the first layer's input channel is 1, output channel is 32 and the kernel size is 3, the stride is 1. Then the second layer's input channel is 32, output channel is 64 and the kernel size is 3, the stride is 1. The third layer's input channel is 64, output channel is 128, and the stride is also 1. And I applied ReLU nonlinear activation. The output size is the number of classes which is 19 (0-18). I added Batch Normalization to standardize the inputs to a layer for each batch and Dropout rate prevent overfitting. And the optimizer technique I used is stochastic gradient descent. And I used *torch.manual_seed* to set five different seeds. The first seed is 1, the second seed is 1234, the third seed is 2340, the fourth seed is 100, and the fifth seed is 3450. All the seeds are the random numbers I picked. Figure 33 the screenshot of the CNN model i.e. with three 2-D convolutional layers. And I train at least 150 epochs each time until the overfitting.

Figure 33: CNN model i.e. with three 2-D convolutional layers.

```
torch.manual_seed(1)
class Net(nn.Module):
  def __init__(self):
    super(Net,self).__init__()
    self.conv1=nn.Conv2d(1,32,3,1)
    self.conv1_bn=nn.BatchNorm2d(32)

    self.conv2=nn.Conv2d(32,64,3,1)
    self.conv2_bn=nn.BatchNorm2d(64)

    self.conv3=nn.Conv2d(64,128,3,1)
    self.conv3_bn=nn.BatchNorm2d(128)

    self.dropout1=nn.Dropout(0.25)

    self.fc1=nn.Linear(35200,100)
    self.fc1_bn=nn.BatchNorm1d(100)

    self.fc2=nn.Linear(100,64)
    self.fc3=nn.Linear(64,19)
```

```
      def forward(self,x):
        x=self.conv1(x)
        x=F.relu(self.conv1_bn(x))

        x=self.conv2(x)
        x=F.relu(self.conv2_bn(x))

        x=self.conv3(x)
        x=F.relu(self.conv3_bn(x))

        x=F.max_pool2d(x,2)
        x=self.dropout1(x)

        x=torch.flatten(x,1)

        x=self.fc1(x)
        x=F.relu(self.fc1_bn(x))

        x=self.fc2(x)
        x=self.fc3(x)
        output=F.log_softmax(x,dim=1)
        return output
    net = Net()
    net.cuda()
```

```
# optimizing methods
optimizer = torch.optim.SGD(net.parameters(), lr=learning_rate)
# choose loss function
loss_func = torch.nn.CrossEntropyLoss()
```

(a) Plot the average training cross-entropy error on the y-axis vs. the epoch number (x-axis). On the same figure, plot the average validation cross-entropy error function. How does the network's performance differ on the training set versus the validation set during learning? Use the plot of training and testing error curves to support your argument.

Here I applied t*orch.nn.CrossEntropyLoss()* to calculate the cross-entropy error and I created a list called *losses_train* to append each cross-entropy error of training set of each iteration. And created a list called *losses_test* to append each cross-entropy error of validation set of each iteration. The calculation process is the same as before. Figure 34.1-34.2 are the plots of cross-entropy error of training set and testing set of selected 2 different seeds from 5 seeds.

Figure 34.1: seed = 1234's cross-entropy error

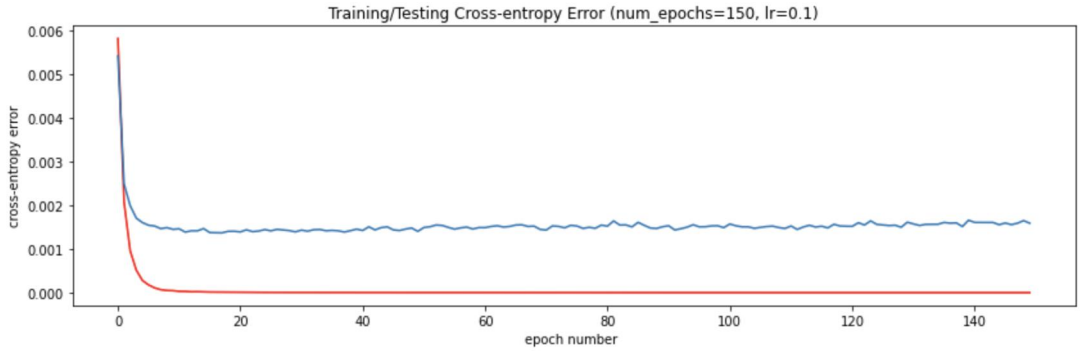
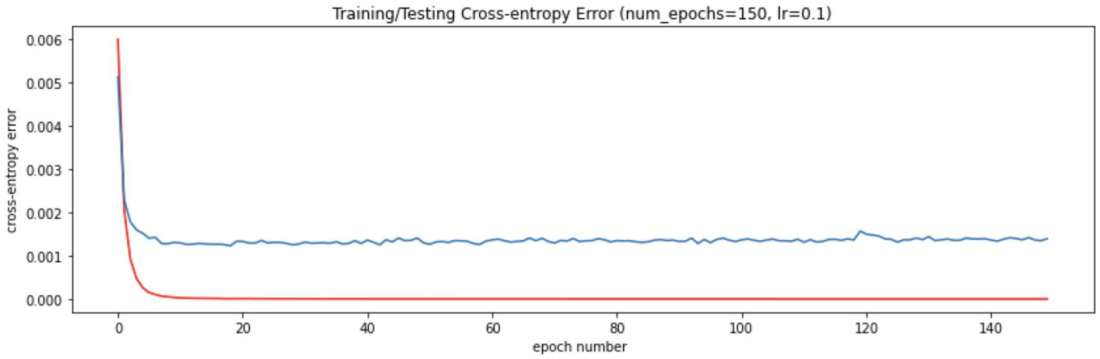


Figure 34.2: seed = 3450's cross-entropy error



Conclusion: In the plot, we can conclude the training error (the red line) decreases to a sightly larger degree in the first 10 epoch times from 0.006 to 0.001, and then it decreases more stable after 20 epoch times, and the error decreases to almost 0 after 150-time iterations. The testing error (the red line) is also declining. And the testing error starts from 0.005 which is less than 0.006 of the training error initially, and decreases to 0.0015.

(b) Plot the classification error vs. number of epochs, for both training and testing. Do you observe a different behavior compared to the behavior of the cross-entropy error function?

Figure 35.1-35.2 are the plots of miss-classification error of training set and testing set of selected 2 different seeds from 5 seeds. **Note: here the miss-classification error is not in percentage because it is easy for us to compare.**

Figure 35.1: seed = 1234's miss-classification error

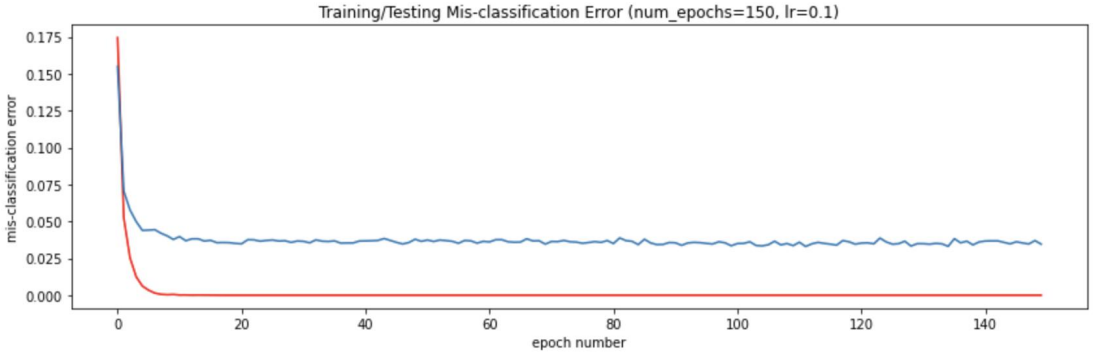
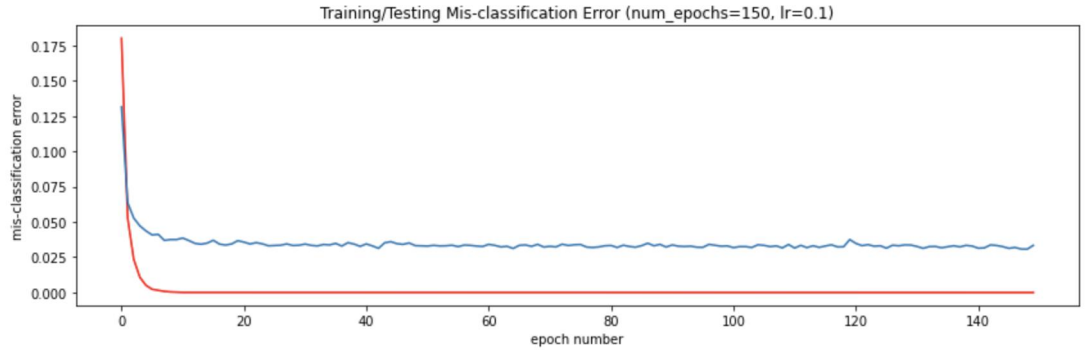


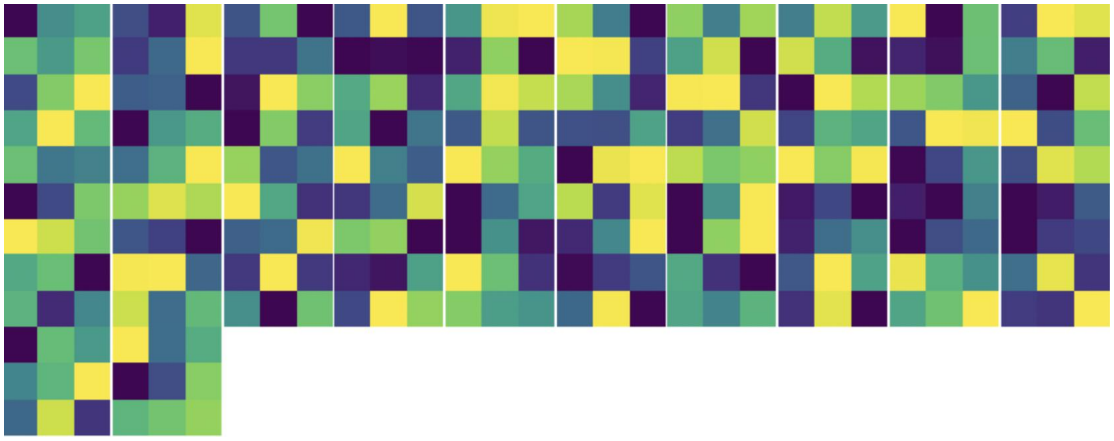
Figure 35.2: seed = 3450's miss-classification error



Conclusion: What can be concluded from the plots is that after 150 times iterations there is overfitting. And the training miss-classification error declines from 0.175 to almost 0 after iterations, and the validation testing error decreases from 0.150 to 0.025. Since the testing error shows overfitting, the fist model can train the dataset well. And there are not so much different behaviors between cross-entropy error and miss-classification error, but due to the different methods of calculating errors, the miss-classification error is a bit larger.

(c) Visualize your best results of the learned W as one hundred 28×28 images (plot all filters as one image, as we have seen in class). Do the learned features exhibit any structure?

Figure 36: plot of filters of the second layer convolutional neural network.



(d) Try different values of the learning rate. How do momentum and learning rate affect convergence rate? How would you choose the best value of these parameters?

Conclusion: Here I tried some combinations of learning rates and momentums, and drew the plots of average cross-entropy error and miss-classification error. The code explicitly introduces different combinations of learning rates and momentums. And the effect of different learning rates and momentums is the same as the one-layer simple neural network, which has already introduced before. Figure 37 - Figure 38 shows different learning rates and momentums effect. Here we will choose learning rate = 0.1 and momentum = 0 or 0.5 is the best case. **Note: here the miss-classification error is not in percentage because it is easy for us to compare.**

Figure 37: the best combinations learning rate = 0.1 and momentum = 0.5

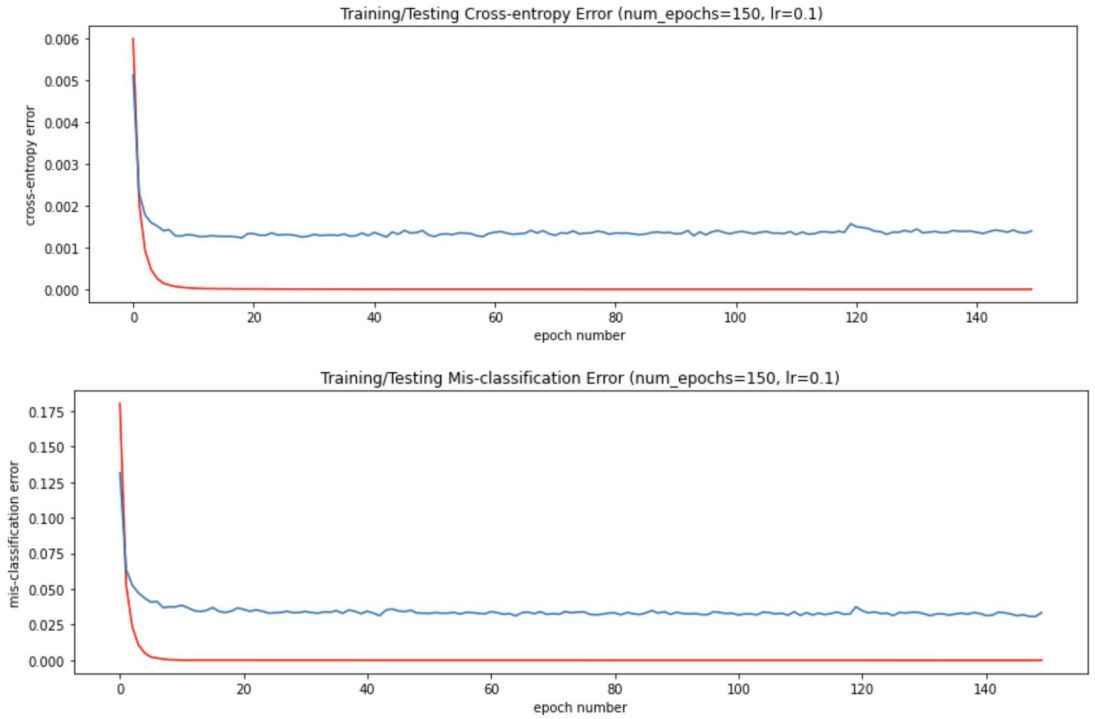
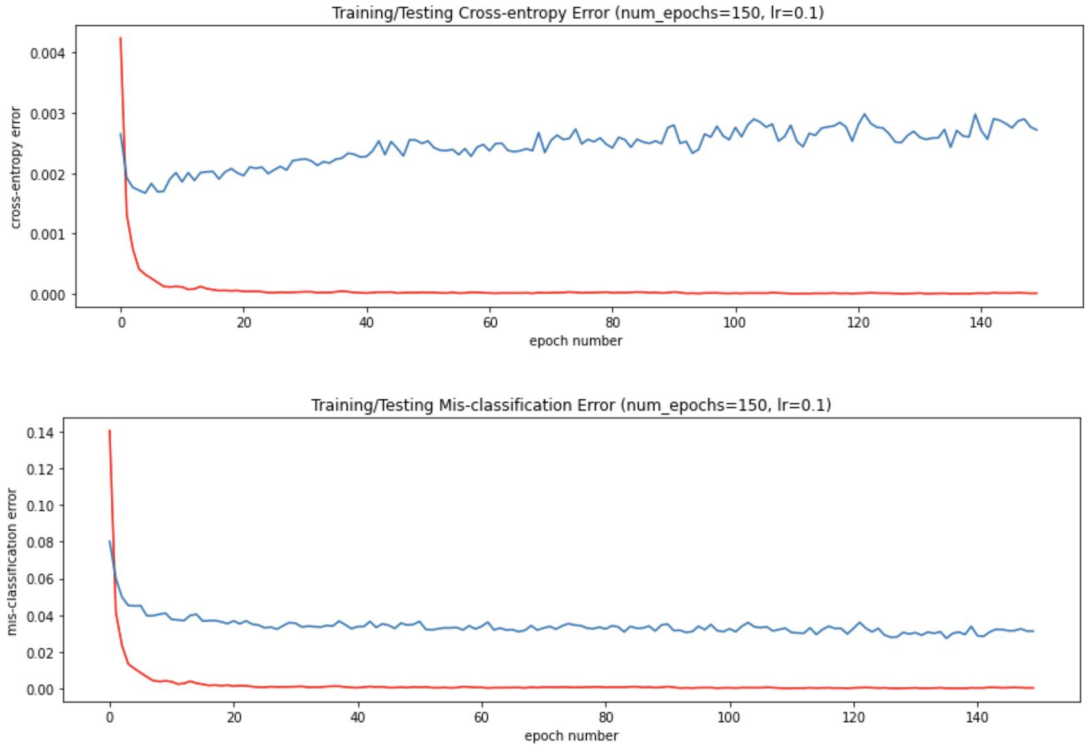


Figure 38: learning rate = 0.1 and momentum = 0.9 (to compare)



(b) Using the validation error (i.e., the performance on val.txt) to select the best model.

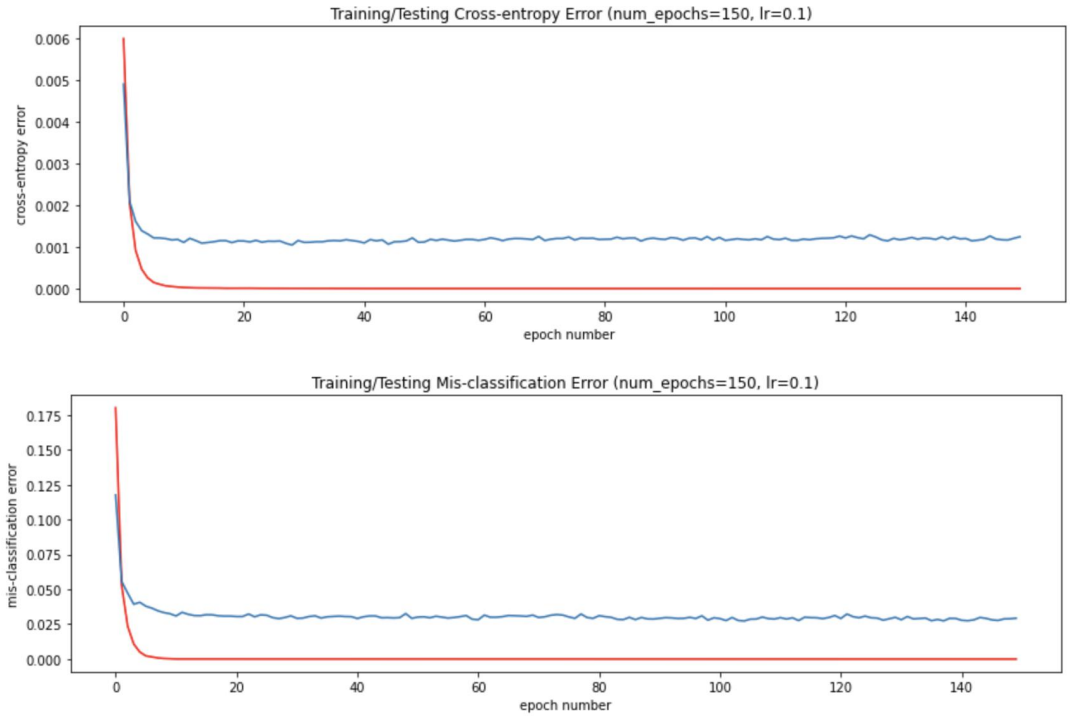
After training the model in the proceeded dataset, and visualize the average training cross-entropy error and the average validation cross-entropy error as well as miss-classification error of training set and validation set. The first model's validation error is about 0.05 after iterations. And the second model's validation error is 0.025 after iterations.

Therefore, **here I choose the second model.**

(c) Report the generalization error (i.e., the performance on test.txt) for the model you picked. How would you compare the test errors you obtained with respect to the original MNIST data? Explain why you cannot obtain a test error lower than 1%.

Then I used the second model to train on the test.txt, and the generalization miss-classification error is 0.025. Figure 39 shows the result. The generalization error is slightly larger than the error of the original MNIST data (Neural network:0.02, CNN: approximately 0.005).

Figure 39: generalization error



In my point of view, I think it might due to the complex dataset compared with the MNIST original dataset. Since the original MNIST dataset only have 10 digits and each of them are quite different each other. But for this dataset with real-valued numbers and the first digit, the data are more complex and the model. And it also might that we have not so much samples as the original MNIST dataset. This dataset has 20000 training samples and the original MNIST dataset has 60000 training samples. It also due to the model, we might need a more accurate model with more layers to train our dataset.