MNIST Data Analysis Project

by

Jingyou Jiang (jj3192)

Columbia University

Department Of Statistics

GR 5241: Statistical Machine Learning

Prof. Xiaofei Shi

Spring 2022

Contents

I. Introduction

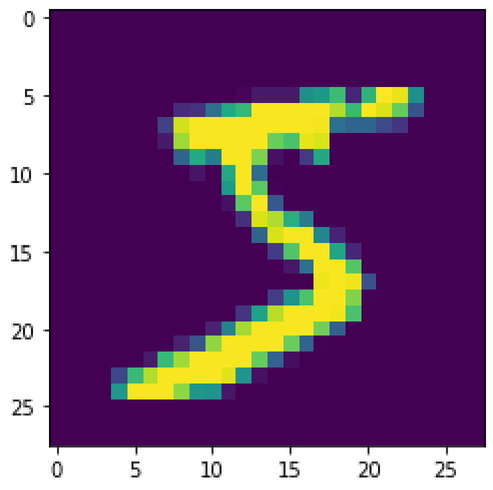
In this report, we're going to work through a series of simple classifiers and neural network architectures and compare their performance on the MNIST handwritten digits dataset. The goal for all the algorithms we examine is the same: take an input image (28*28 pixels) of a handwritten single digit (0–9) and classify the image as the appropriate digit. The MNIST database of handwritten digits is one of the most commonly used datasets 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. It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting.

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 NIST were size normalized. The resulting images contain grey levels because of the anti-aliasing technique used by the normalization algorithm. The images were centered in a 28*28 image by computing the center of mass of the pixels and translating the image to position this point at the center of the 28*28 field.

In this project, we will analyze the dataset MNIST using some classification methods and Deep Learning techniques. We will build different classifiers and neural network structures with MNIST training and test datasets to evaluate their test errors, loss functions and accuracy to determine their performance. Moreover, we will further implement these Machine Learning algorithms with the given datasets "train.txt", "val.txt" and "test.txt" to visualize their performance.

II. Data Processing

The first and most important step in any machine learning task is to prepare the data. It is always a good start to preprocess our data and familiarize with the MNIST data. First download the MNIST database from the *torch.utils.data* package and split the data into training and testing data sets. To be familiar with the data, plot the first sample of X_train to see how the predictor data X look like.



From the graph above, it is easily to recognize the plot as a hand-written number 5 from the 28*28 field. Extract the first Y_train sample to see that the first response case is a numerical number with the value of 5, which matches the plot of corresponding predicting variable. Then, display the dimensions of X_train and X_test are (60000, 28, 28) and (10000, 28, 28) separately. And normalize the training and testing data using *processing.normalize* function in the *sklearn* package.

Moreover, to deal with the labels in the response variable data, use a popular choice of the one-hot embedding to transform the response variable Y_train and Y_test datasets into a dimension of (60000, 10) and (10000,10). The benefit of this transformation is that it doesn't consider an ordinal relationship between labels or complex labels involving characters. The one-hot embedding maps each label to a binary vector. It can make our data more expressive and easier to rescale with a probability-like number for each possible label value. It can make the problem easier for the network to model. It also may offer a more nuanced set of predictions than a single label. After the data transformation and data preprocessing, it is time to proceed with the analysis of MNIST datasets.

III. Simple Classification Models

A key property for a good machine learning algorithm is the reproducibility, meaning that one can repeatedly run the algorithm on certain datasets and obtain the same (or similar) results on a particular project. Historically, many classification methods have been tested with MNIST training and test sets. A supervised machine learning algorithm can rely on labeled input data to learn a function that produces an appropriate output when given new unlabeled data. Thus, before using the deep learning algorithms, there are several classification methods including KNN, AdaBoost.M1/ C4.5, SVM with Gaussian Kernel that can be applied to check the test data.

3.1 KNN Classification

The k-nearest neighbors (KNN) algorithm is a simple, easy-to-implement supervised machine learning algorithm that can be used to solve both classification and regression problems. KNN is a type of instance-based learning, where the function is only approximated locally, and all computation is deferred until classification. It is a non-parametric algorithm wherein it doesn't require training data for inference. Hence training is much faster while inference is much slower when compared to parametric learning algorithm for all obvious reasons. The algorithm is shown as followings:

1. *Load the data*
2. *Initialize the value of k*
3. *To get the predicted class, iterate from 1 to the total number of training data*
4. *Calculate the distance between test data and each row of training data.*
5. *Sort the calculated distances in ascending order based on distance values*
6. *Get top k rows from the sorted array*
7. *Get the most frequent class of these rows*
8. *Return the predicted class*

In this project, we apply the KNN method on our normalized training data by using the built-in function *KNeighborsClassifier* from the *sklearn* package with the parameter k=67 to establish our first classification model. Then, test the model using the test dataset to get a 5.08% test error, which reproduces the 5% test error of Yann LeCun's model established in 1998.

3.2 AdaBoost

Ada-boost or Adaptive Boosting is one of ensemble boosting classifiers, which is also an iterative ensemble method. It builds a strong classifier by combining multiple poorly performing classifiers to increase accuracy. The basic concept behind Adaboost is to set the weights of classifiers and training the data sample in each iteration such that it ensures the accurate predictions of unusual observations. Any machine learning algorithm can be used as base classifier if it accepts weights on the training set. The algorithm works in the following steps:

1. *Adaboost first selects a training subset randomly.*
2. *Iteratively train the model by selecting the training set based on the accurate prediction of the last training.*
3. *Assign the higher weight to wrong classified observations so in the next iteration these observations will get high probability for classification.*
4. *Then, it assigns the weight to the trained classifier according to the accuracy of the classifier. The more accurate classifier will get high weight.*
5. *This process iterates until the complete training data fits without any error or reach to the specified maximum number of estimators.*
6. *To classify, perform a "vote" across all the learning algorithms you built.*

In this project, the AdaBoost. M1/ C4.5 method is applied by using the built-in function *AdaBoostClassifier* and *DecisionTreeClassifier* from the *sklearn* package with the parameter max_depth of decision tree=12 and number of estimators=70. The test error is 3.97%, which is close to the 4.05% test error of Corinna Cortes's model.

3.3 SVM with Gaussian Kernel

A Support Vector Machine (SVM) is a useful machine learning tool that can be used for both regression and classification tasks. An SVM works by mapping the input data into a high dimensional space, and then separating the input using a hyperplane that maximizes the distance between the hyperplane and the categories that it seeks to separate as well as minimizing the number of misclassified examples. The basic steps of the SVM are:

1. *Select two hyperplanes which separate the data with no points between them*
2. *Maximize their distance (the margin)*
3. *The average line will be the decision boundary*

In this project, the SVM method is applied by using the built-in function *SVC* from the *sklearn* package with the parameter Kernel= "rbf" (Gaussian), regularization C=100, and gamma=0.05. The test error is 1.63%, which is close to the 1.4% test error of Christopher J.C. Burges' model.

From the three simple classification methods introduced above, we can see the SVM with Gaussian Kernel performs the best by producing the lowest test error. Trying to implement a better model on the training set and producing a lower test error than the 1.4% record, the SVM with Gaussian Kernel method can be re-used by turning its hyperparameters. If the parameter Kernel= "rbf" (Gaussian), regularization C=11 and gamma=0.1 are chosen, we can get the test error 1.51%, which outperforms pervious results among all classifiers.

IV. Deep Learning

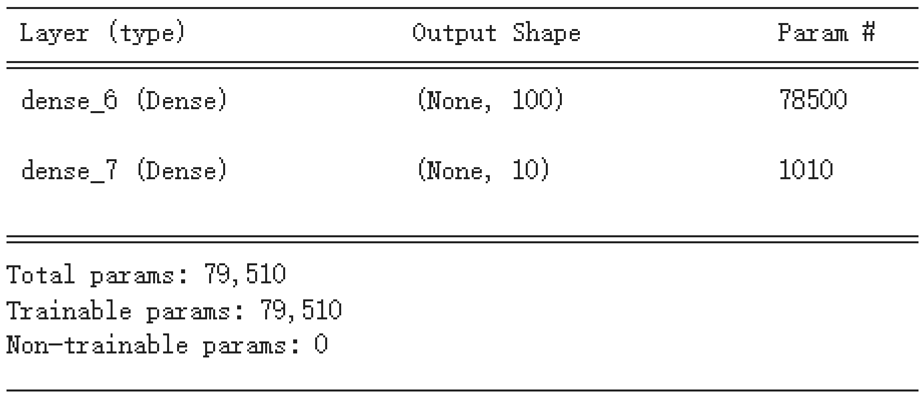
Deep learning is a type of machine learning and artificial intelligence (AI) that imitates the way humans gain certain types of knowledge. It can automate predictive analytics. It is extremely beneficial to data scientists who are tasked with collecting, analyzing, and interpreting large amounts of data. Deep learning neural networks such as recurrent neural networks, convolutional neural networks, and artificial neural networks need massive amounts of data on which to train. Thus, in this project, deep learning methods can make the process of MNIST data analysis faster and easier.

4.1 Single Hidden Layer Neural Network

A single-layer neural network is the simplest form of artificial neural network. There is only one layer of input nodes that send weighted inputs to a subsequent layer of receiving nodes. It only consists of 3 layers: input, hidden, and output. In our project, this method can be used by training with one 100-hidden-unit layer, an architecture of input (784) → hidden (100) → output (10). The model is going to be initially built by choosing the Stochastic Gradient Descent optimizer with a learning rate of 0.1 and momentum of 0, a batch size of 256, and 150 epochs each time. The model will run repeatedly for 5 times using different random seeds to start with a slightly different initialization of the weights each time. The average cross-entropy error (loss function) and miss-classification error for both training and validation will be plotted each time with the epoch number on the x-axis.

Building the network

Our output layer uses a special activation function called SoftMax. It normalizes the values from the ten output nodes such that all the values are between 0 and 1, and the sum of all ten values is 1. It allows us to treat the ten output values as probabilities, and the largest one is selected as the prediction for the one-hot vector. In machine learning, the SoftMax function is always used when our model output is a one-hot encoded vector. Here is the model summary:

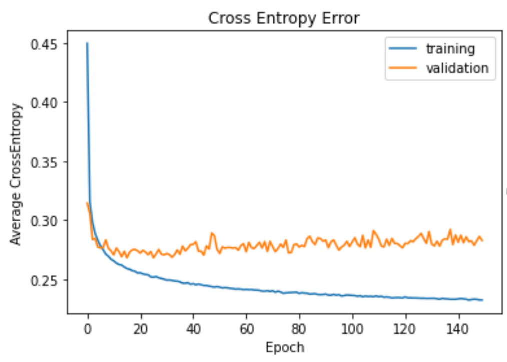


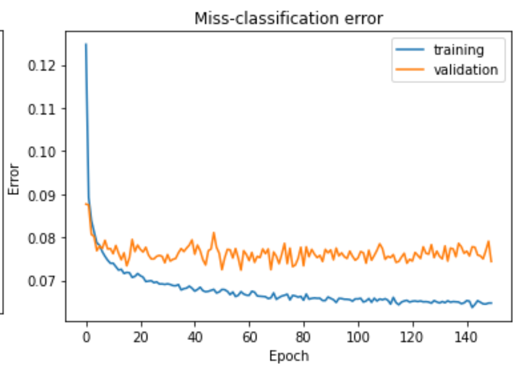
```
Layer (type)                 Output Shape              Param #
=================================================================
dense_6 (Dense)              (None, 100)               78500

dense_7 (Dense)              (None, 10)                1010
=================================================================
Total params: 79,510
Trainable params: 79,510
Non-trainable params: 0
```

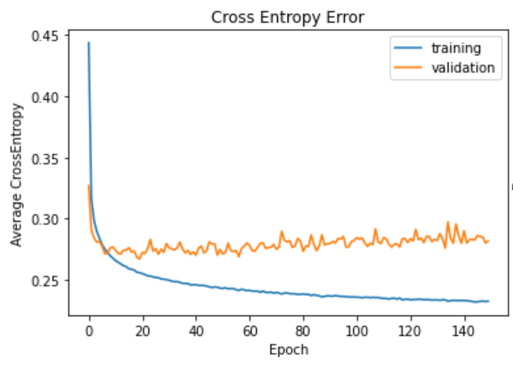
Then, use *Keras* to train and evaluate this model. Here are the results of average cross-entropy error and miss-classification error plot for each run:

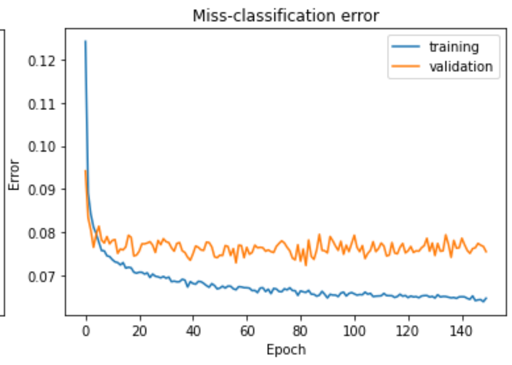
Seed=100



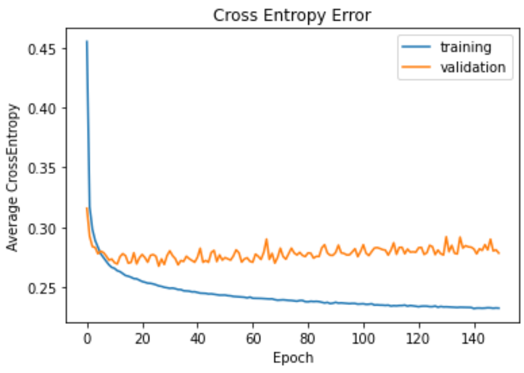


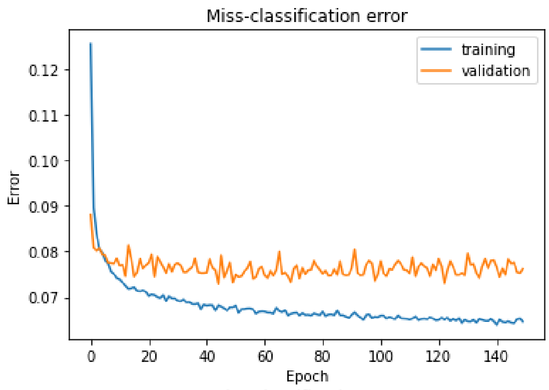
Seed=10



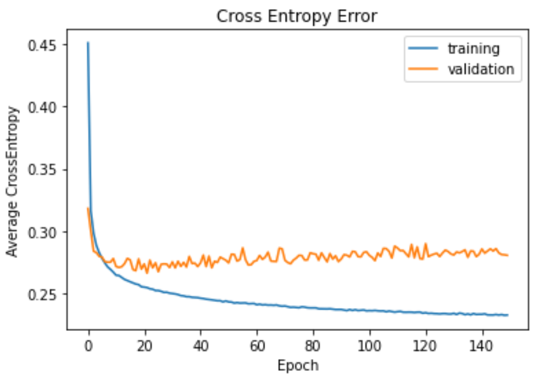


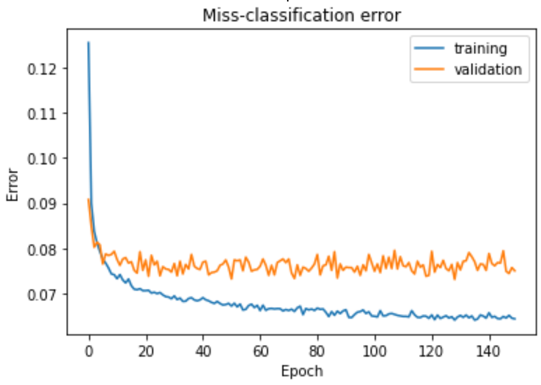
Seed=300



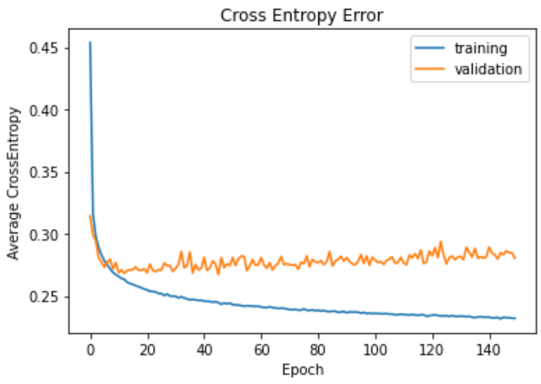


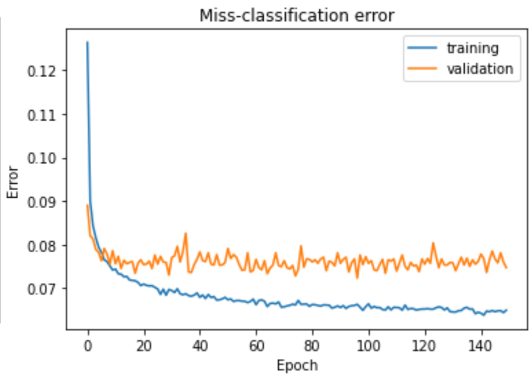
Seed=500





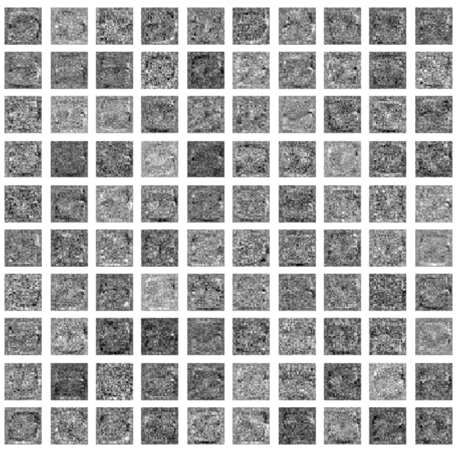
Seed=1000





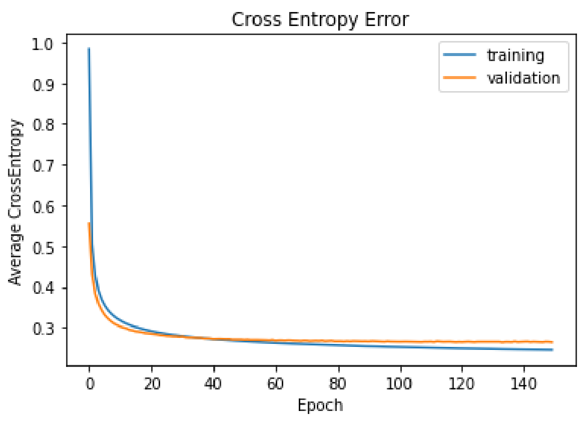
Then we get our best Single Hidden Layer Neural Network model with the test loss 0.28, and accuracy 0.926 after several runs.

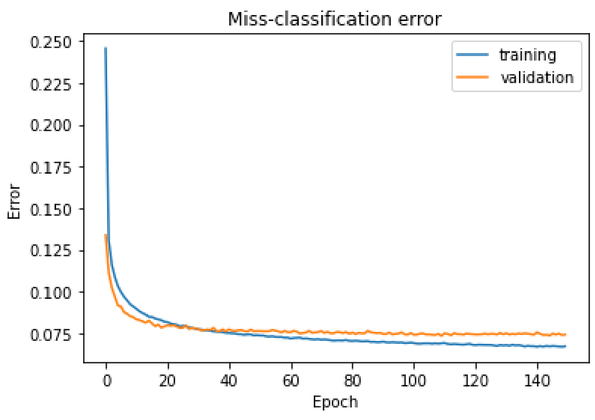
Visualize the best results of the learned W using one hundred 28×28 images:



Then, modify different parameters including learning rate (0.01, 0.2, 0.5) and momentum (0.5, 0.9) to see any change.

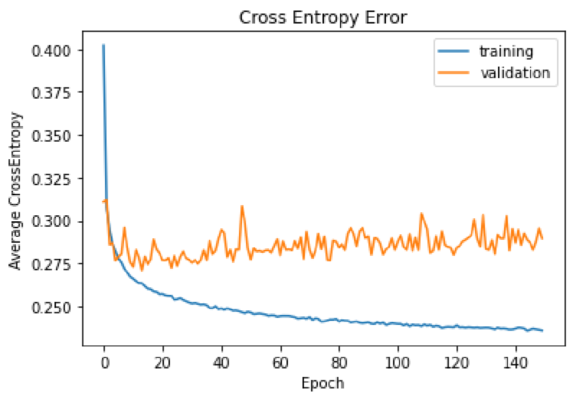
Seed=100, Learning rate=0.01, Momentum=0

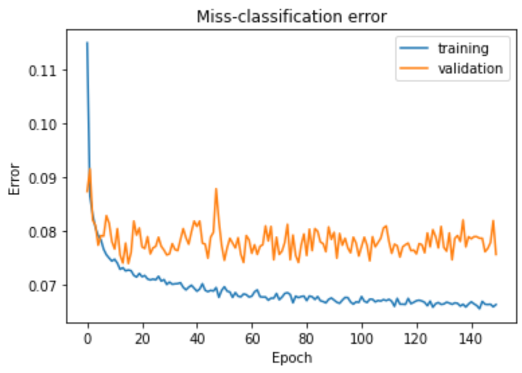




Test loss: 0.264, Test accuracy: 0.926

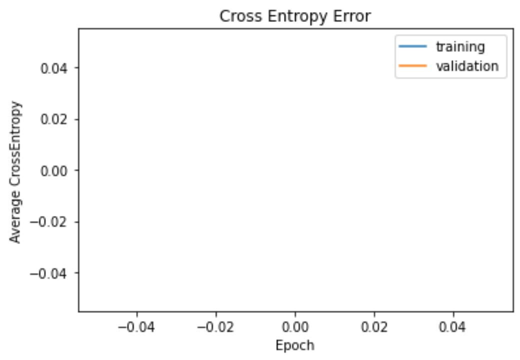
Seed=100, Learning rate=0.2, Momentum=0

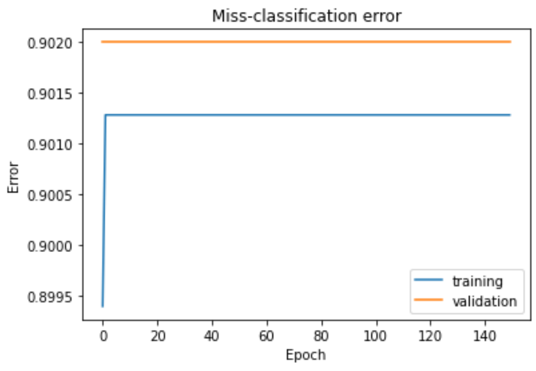




Test loss: 0.289, Test accuracy: 0.924

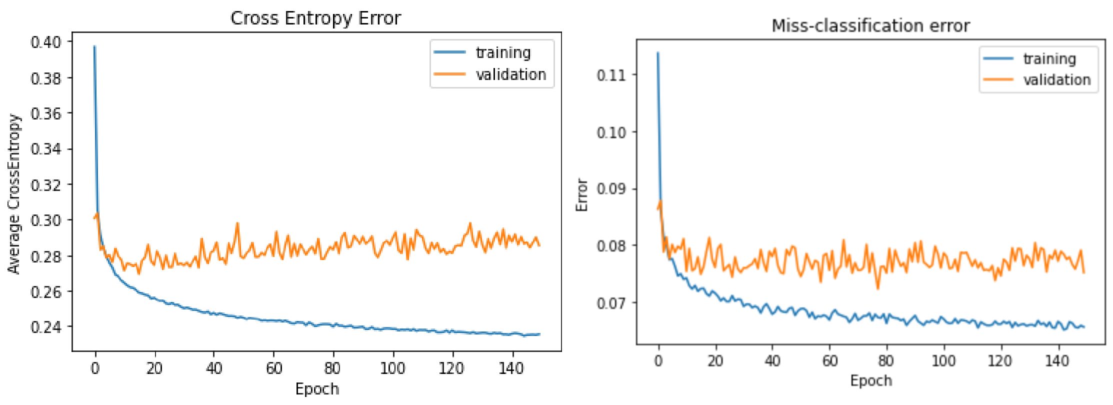
Seed=100, Learning rate=0.5, Momentum=0





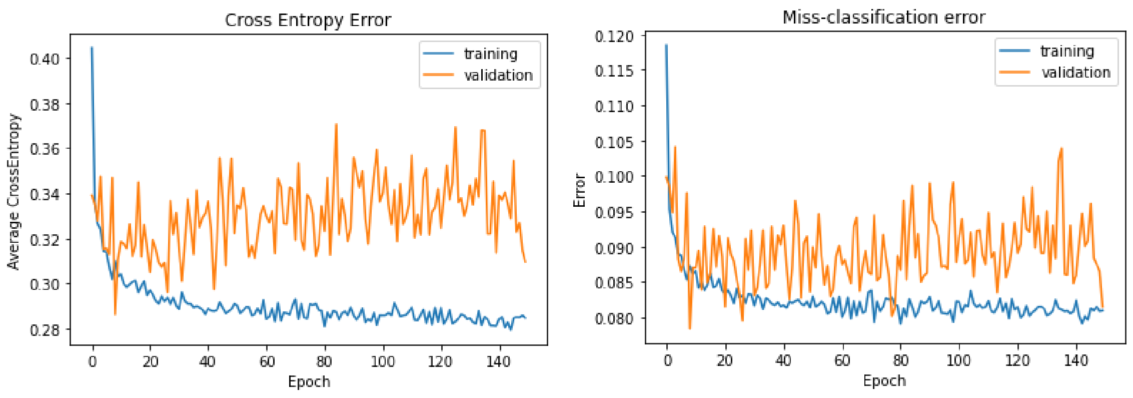
Test loss: NA, Test accuracy: 0.098

Seed=100, Learning rate=0.1, Momentum=0.5



Test loss: 0.285, Test accuracy: 0.925

Seed=100, Learning rate=0.1, Momentum=0.9



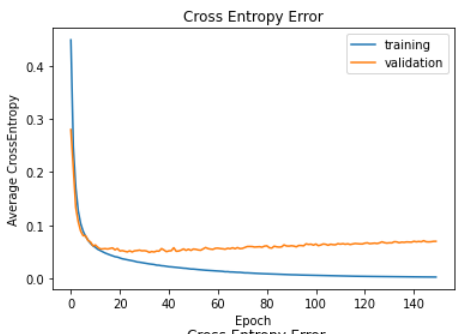
Test loss: 0.31, Test accuracy: 0.919

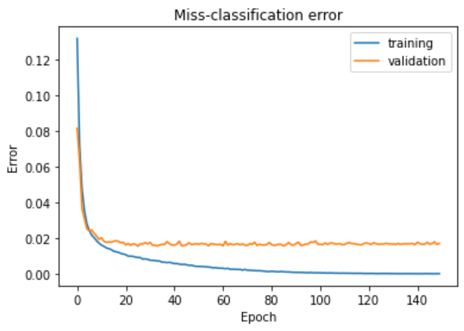
Thus, we can see that momentum can accelerate training and learning rate schedules can help to converge the optimization process. Choosing a value of learning rate too small may result in a long training process that could get stuck, whereas choosing a value too large may result in learning a sub-optimal set of weights too fast or an unstable training process. Thus, in this model, from the obtained results above, the best value of parameters for this model will be 0.1 learning rate and 0.5 momentum.

4.2 Convolutional Neural Network

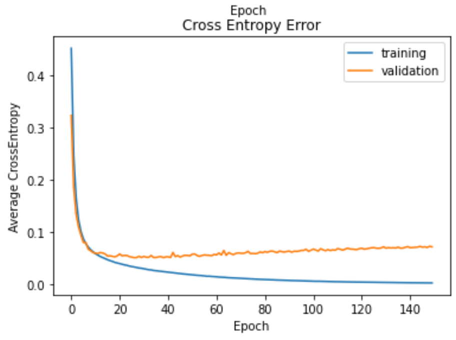
A Convolutional Neural Network (CNN) is a Deep Learning algorithm which can take in an input image, assign weights and biases to various aspects/objects in the image and differentiate one from the other. The pre-processing required in a CNN is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, CNN can automatically learn these filters. In this project, we start with one 2-D convolutional layer with 32 numbers of filters and kernel size 3, adding the Relu activation, adding a Maxpooling 2-D layer with pool size (2,2) to build our CNN architecture. The model is going to be initially built by choosing the Stochastic Gradient Descent optimizer with a learning rate of 0.1 and momentum of 0, a batch size of 256, and 150 epochs each time. The model will run repeatedly using 5 different seeds to start with different initialization of the weights each time. The average cross-entropy error and miss-classification error will be plotted each time with the epoch number on the x-axis. Here are the results:

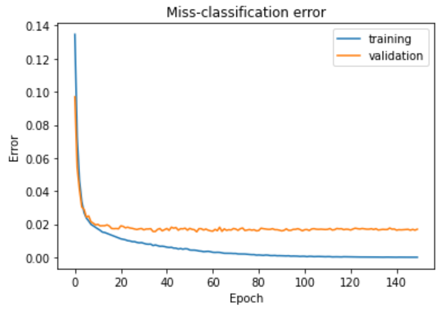
Seed=100



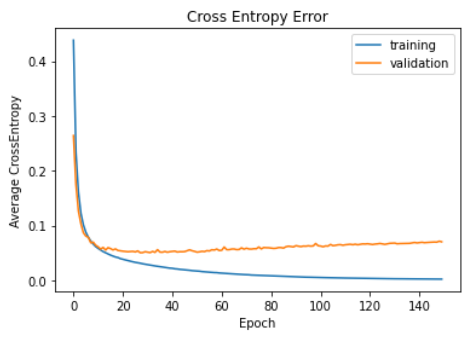


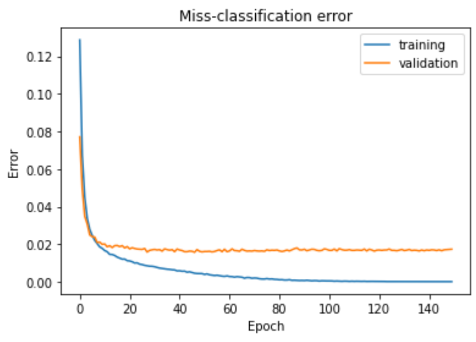
Seed=10



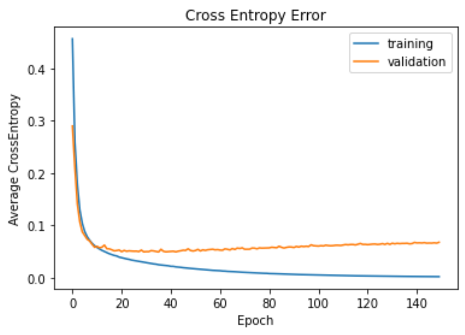


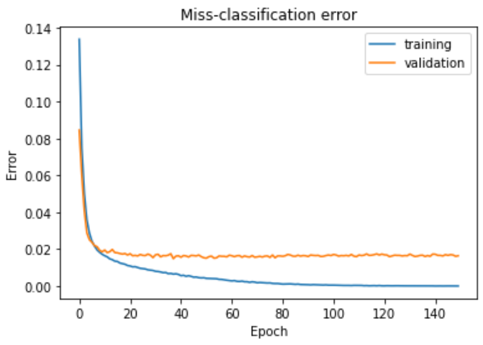
Seed=300



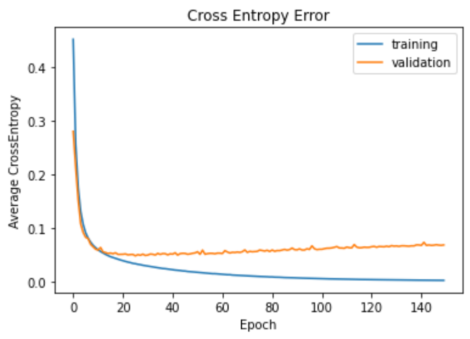


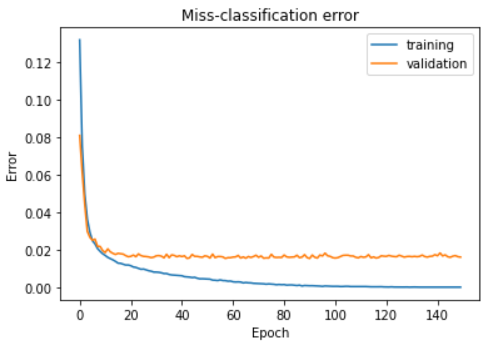
Seed=500





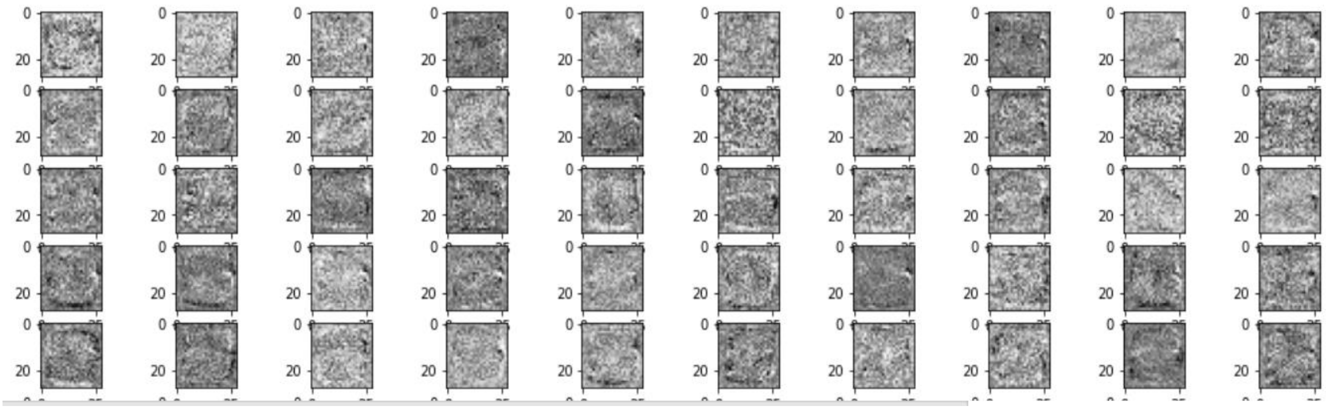
Seed=1000





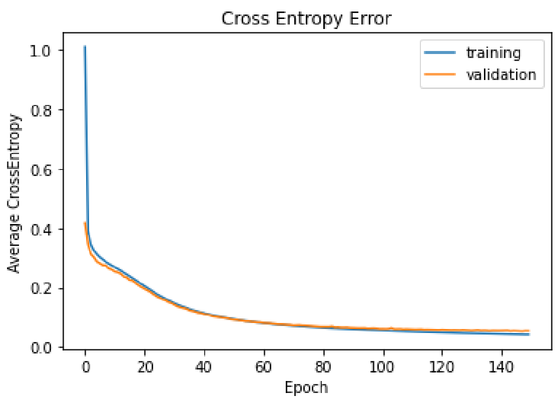
Then we get our best Convolutional Neural Network model with the test loss 0.0679, and accuracy 0.984 after several runs.

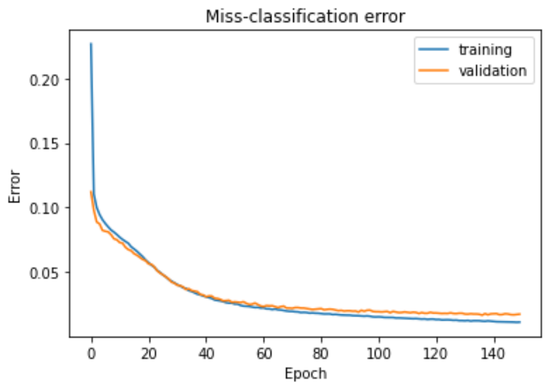
Visualize the best results of the learned W using one hundred 28×28 images:



Then, modify different parameters including learning rate (0.01, 0.2, 0.5) and momentum (0.5, 0.9) to see any change.

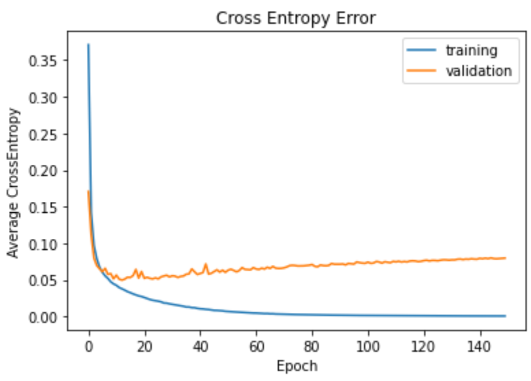
Seed=100, Learning rate=0.01, Momentum=0

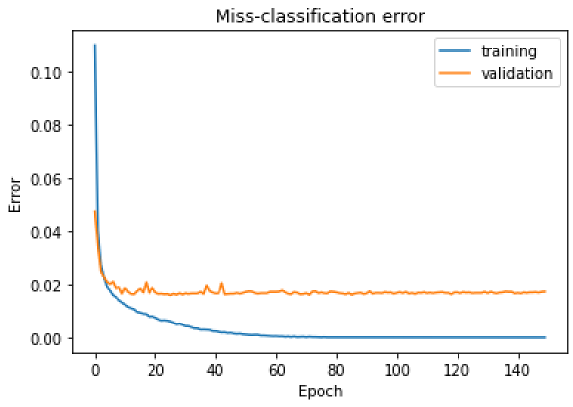




Test loss: 0.0542, Test accuracy: 0.983

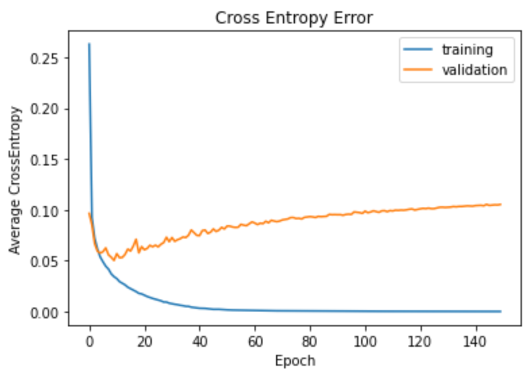
Seed=100, Learning rate=0.2, Momentum=0

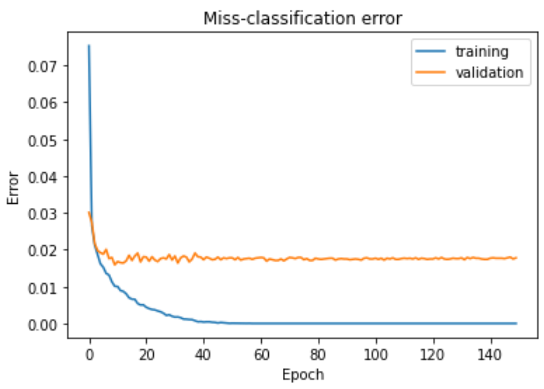




Test loss: 0.0798, Test accuracy: 0.983

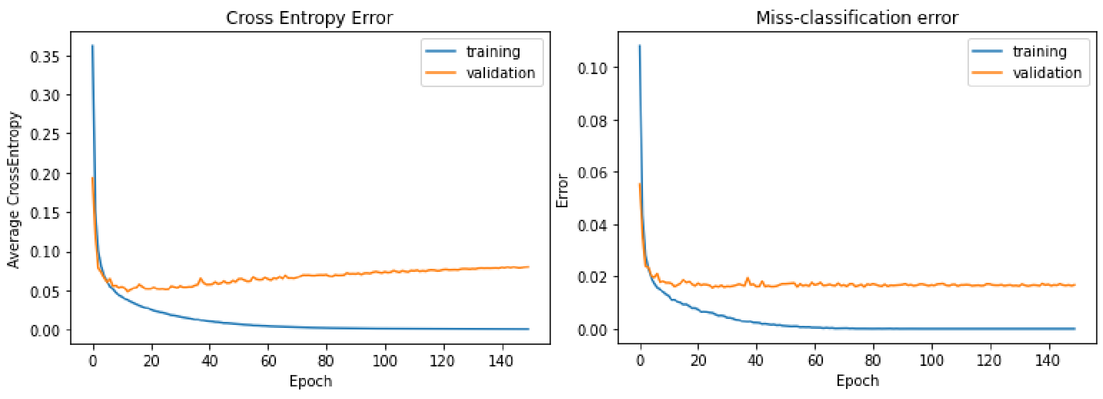
Seed=100, Learning rate=0.5, Momentum=0





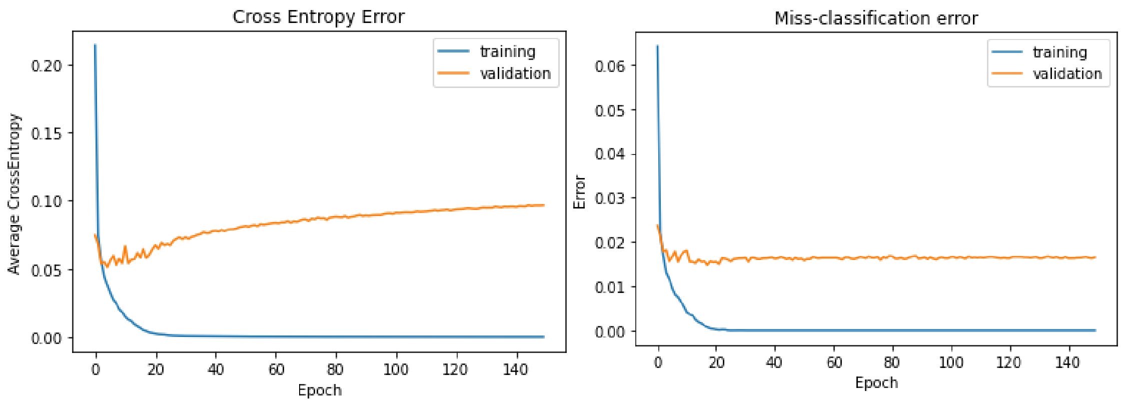
Test loss: 0.105, Test accuracy: 0.982

Seed=100, Learning rate=0.1, Momentum=0.5



Test loss: 0.0799, Test accuracy: 0.983

Seed=100, Learning rate=0.1, Momentum=0.9

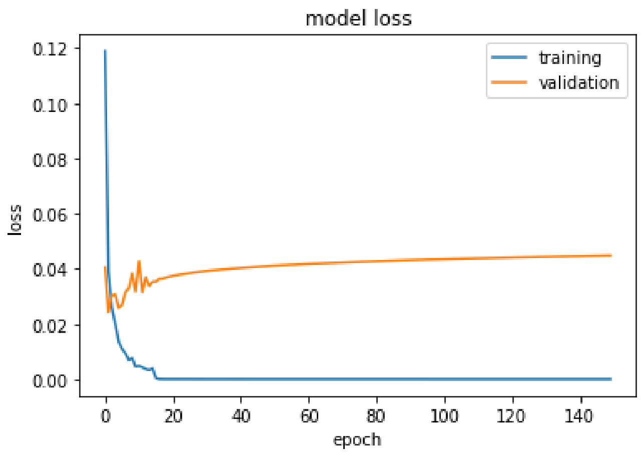


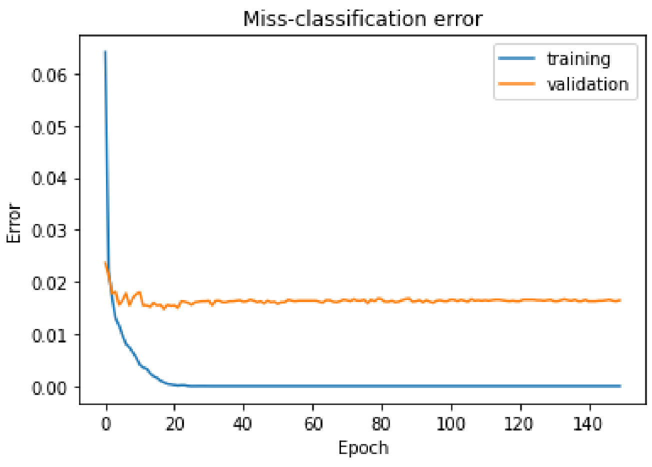
Test loss: 0.0964, Test accuracy: 0.984

Thus, we can see that momentum can accelerate training and learning rate schedules can help to converge the optimization process. Choosing a value of learning rate too small may result in a long training process that could get stuck, whereas choosing a value too large may result in learning a sub-optimal set of weights too fast or an unstable training process. Thus, in this model, from the obtained results above, the best value of parameters for this model will be 0.1 learning rate and 0.5 momentum.

4.3 CNN Performance Improvement

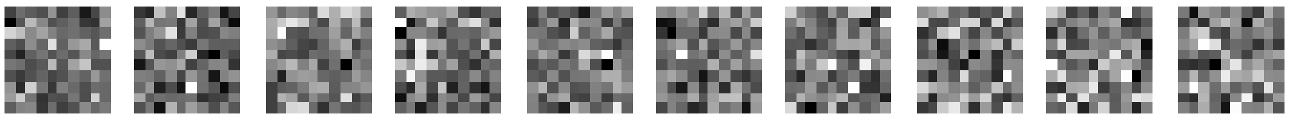
To further develop the convolutional neural network (CNN) structure have a test error rate lower than 1.4%, we build the model first with a single convolutional layer with a small filter size (3,3) and a modest number of filters (32) followed by a max-pooling layer. Then, increase the depth of the feature extractor part of the model, following a VGG-like pattern of adding more convolutional and pooling layers with the same sized filter, while increasing the number of filters to 64 each with another max-pooling layer. The filter maps can then be flattened to provide features to the classifier. Between the feature extractor and the output layer, we can add a dense layer with 100 nodes to interpret the features. All layers will use the *ReLU* activation function and the *He* weight initialization scheme. We will use a conservative configuration for the stochastic gradient descent optimizer with a learning rate of 0.1 and a momentum of 0.5. Then, the model will be evaluated using five-fold cross-validation. We will train the baseline model for a modest 150 training epochs with a default batch size of 32 examples. Here are the results of average cross-entropy error and miss-classification error plot:





Then we get our best Convolutional Neural Network model with the test loss 0.0448, and test accuracy 0.993 after several runs. It means we get the test error around 0.7%, which beats the performance of SVM with Gaussian Kernel, which has a test error rate of 1.4%.

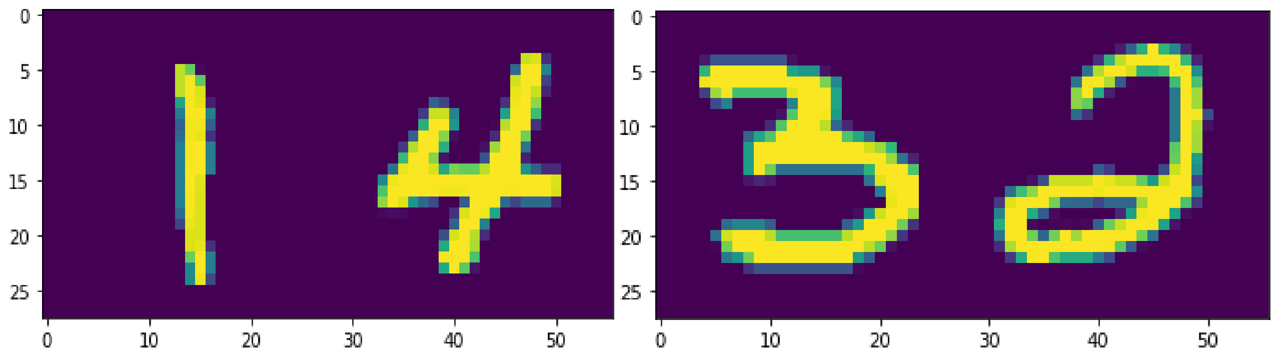
Visualize the best results of the learned W using one hundred 28×28 images:



V. More about Deep Learning

5.1. Data Processing

In this part, we are working with the given data train.txt, val.txt and test.txt. In particular, train.txt contains 20,000 lines and val.txt and test.txt contains 5000 lines in the same format. Each line contains 1569 coordinates, with the first 784 real-valued numbers correspond to the 784 pixel values for the first digit, next 784 real valued numbers correspond to the pixel values for the second digit. To begin with, load the



data and plot a few examples to decide if the pixels were scanned out in row-major or column-major order. The first two cases look like:

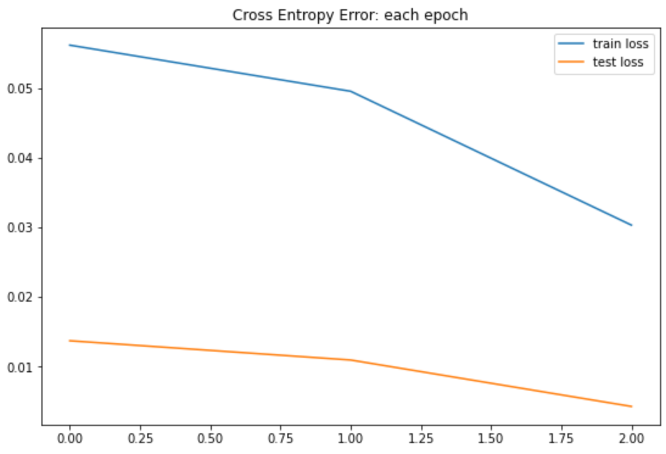
The pixels were scanned out in a row-major order. We can see the first 1568 digits form a 28 * 56 picture that contains two numbers with 784 pixel values each, and the last digit, the 1569th coordinate is the sum of the two numbers in a numerical form (i.e. number 5 in first two samples).

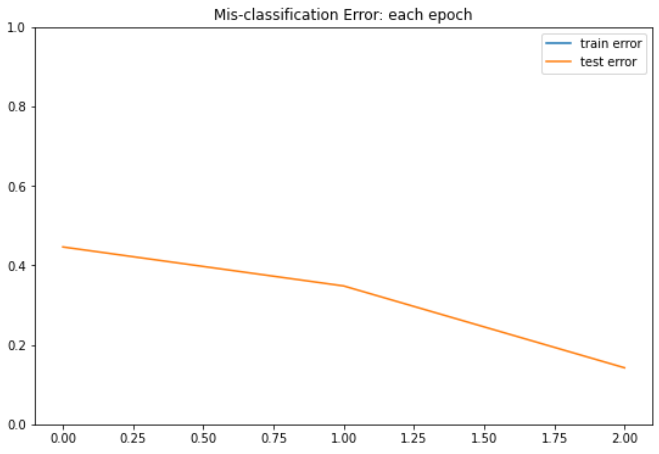
5.2. Deep Learning Models

To work on the analysis of these given data sets, we decide to use Deep Learning models to train the model, test the validation error, report the generalization error so that we can determine the performance of the Deep Learning architecture that we build and choose the best model, trying to obtain a test error lower than 1%.

Based on the performance of pervious models that we build on the MNIST data, we decide to build a multi-layer Convolutional Neural Network model with and without the dropout in training to check their performance on the given datasets.

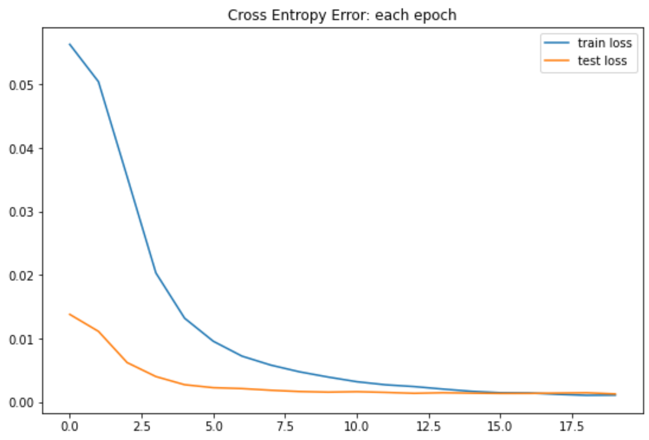
First, we build the CNN with three convolutional layers with number of filters 1*32, 32*64, 64*64 and kernel size 3 followed by a max-pooling layer with size 2. The filter maps can then be flattened to provide features to the classifier. All layers will use the *ReLU* activation function and the *He* weight initialization scheme. Two linear transformation function will then be added into the model to transform the input and output data size. We will use a conservative configuration for the stochastic gradient descent optimizer with a learning rate of 0.1 and a momentum of 0. We will train the baseline model for a modest 150 training epochs with a default batch size of 32 examples. Here are the results of average cross-entropy error and miss-classification error plot:

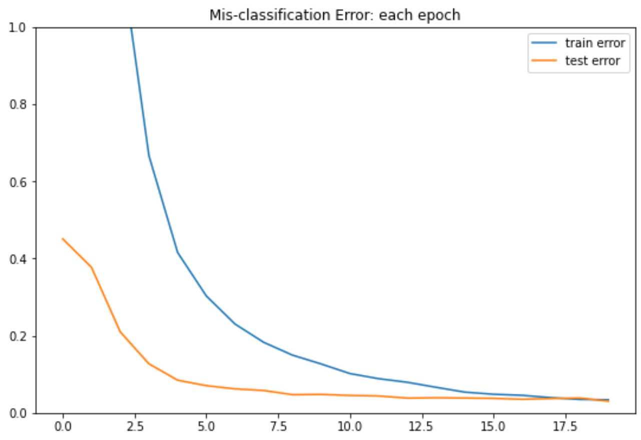




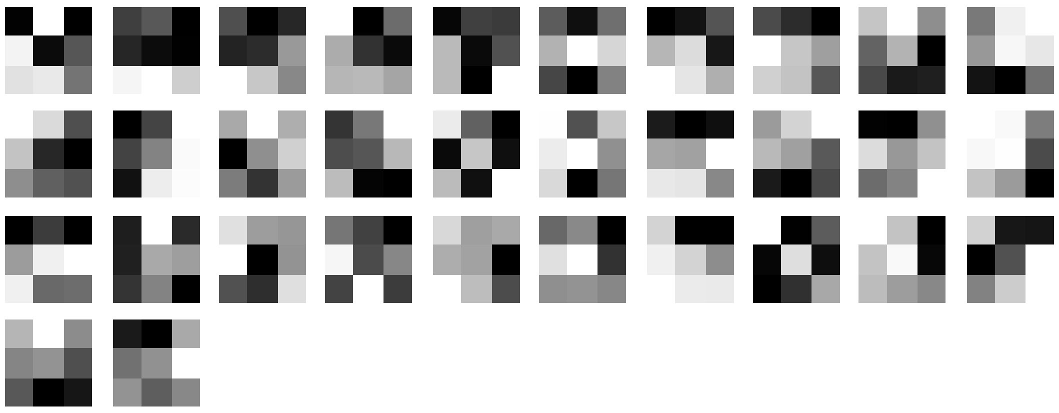
Visualize the best results of the learned W using one hundred 28×28 images:

Then, we build another CNN model with the same structure as the above one. But then we add on a dropout function to dropout the 0.5 probability of an element to be zeroed. After that, we can run the training function with the stochastic gradient descent optimizer with a learning rate of 0.1, no momentum, epoch number of 150, and default batch size. Here are the results of average cross-entropy error and miss-classification error plot:





Then, we can see a very low test loss and test error that is much lower than 0.01 and 1% respectively so we can conclude that the multi-layer CNN model with dropout is the best model that perform a test error lower than 1% on the given datasets. Visualize the best results of the learned W using one hundred 28×28 images:



VI. Results & Conclusion

In this project, we first analyze the dataset MNIST using some classification methods including KNN, AdaBoost with Decision Tree, SVM with Gaussian Kernel. And we get the test error shown in the following table:

Methods	**Test Error**	**Parameters**
KNN	5.08%	K=67
AdaBoost.M1 / C4.5	3.97%	Max depth=12, n=70
SVM with Gaussian Kernel	1.63%	C=100, gamma=0.05
SVM with Gaussian Kernel	1.51%	C=11, gamma=0.1

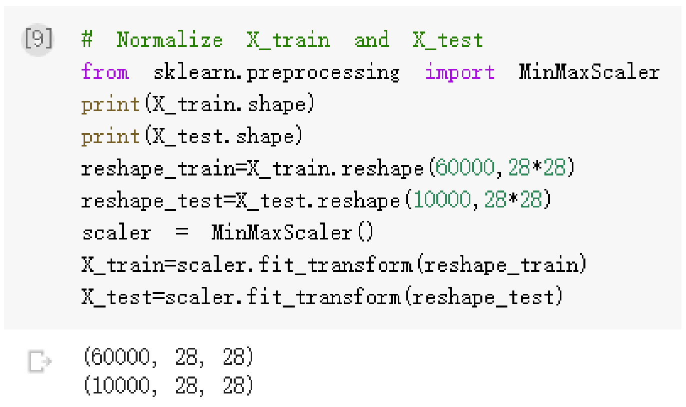
Then, we apply the Deep Learning techniques to build neural networks such as Single Hidden Layer Neural Network and CNN to train the MNIST data and test the loss and accuracy to determine the performance of each network model. After several runs of evaluation and validation, we obtain the Single Hidden Layer Neural Network model with the test loss around 0.28, and accuracy around 0.926 and the Convolutional Neural Network model with the test loss around 0.0679, and accuracy around 0.984. Eventually, a multi-layer CNN model performs the test loss 0.0448, and test accuracy 0.993 after several runs. It means we get the test error around 0.7%, which beats the performance of SVM with Gaussian Kernel, which has a test error rate of 1.4%.

To work with the given datasets “train.txt”, “val.txt” and “test.txt”, we further implement Deep Learning algorithms like multi-layer CNN structures with and without Dropout to visualize their performance. As a result, we find the best classifier with multi-layer CNN model with dropout that performs a test error lower than 1% on the given datasets. The result is close to MNIST data we had in the pervious part.

Throughout the project, we find out that the Deep Learning algorithms perform generally well on prediction and classification, especially the CNN model, which automatically learn the filters. We can further improve its performance by adding more layers, but it will also cause a long runtime, which is an expensive and time costly approach. We can also tune the hyperparameters in each classifier and adjust the training parameters like seed, epoch number, learning rate and momentum to perform the model with different initialization, convergence rate, and running time. We also cannot add too many layers since we need to avoid the overfitting problem. In this project, all the deep learning structures built perform well overall and there is basic no overfitting problem on the model mentioned above.

VII. Appendix

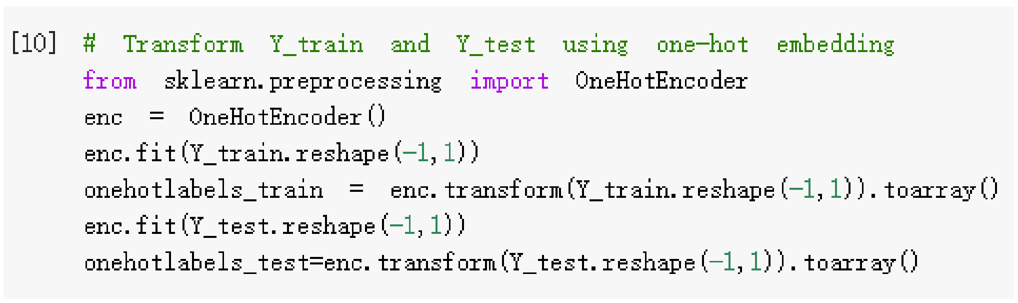
1. Data Normalization



```
[9] # Normalize X_train and X_test
    from sklearn.preprocessing import MinMaxScaler
    print(X_train.shape)
    print(X_test.shape)
    reshape_train=X_train.reshape(60000,28*28)
    reshape_test=X_test.reshape(10000,28*28)
    scaler = MinMaxScaler()
    X_train=scaler.fit_transform(reshape_train)
    X_test=scaler.fit_transform(reshape_test)
```

```
(60000, 28, 28)
(10000, 28, 28)
```

2. One-hot Embedding



```
[10] # Transform Y_train and Y_test using one-hot embedding
     from sklearn.preprocessing import OneHotEncoder
     enc = OneHotEncoder()
     enc.fit(Y_train.reshape(-1,1))
     onehotlabels_train = enc.transform(Y_train.reshape(-1,1)).toarray()
     enc.fit(Y_test.reshape(-1,1))
     onehotlabels_test=enc.transform(Y_test.reshape(-1,1)).toarray()
```

3. Single Hidden Layer Neural Network model

```
from tensorflow import keras
# Set seed for producing results with different weights
tf.random.set_seed(100)
model = Sequential()
# The input layer requires the special input_shape parameter which should match the shape of our training data.
# Define a model with only one layer with 100 hidden units
model.add(keras.layers.Dense(100))
model.add(keras.layers.Dense(10,activation="softmax"))

# Loss is cross-entropy and optimizer is SGD
model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.1,momentum=0), loss='categorical_crossentropy', metrics=['accuracy'])
history = model.fit(X_train, onehotlabels_train, batch_size=128, epochs=150, verbose=False, validation_data=(X_test,onehotlabels_test))
loss, accuracy = model.evaluate(X_test, onehotlabels_test, verbose=False)
```

4. Visualize Results

```
params = model.layers[0].get_weights()[0]
plt.figure(figsize=(8, 8))
for i in range(params.shape[0]):
    plt.subplot(10, 10, i + 1) # Since we know it is a 10 x 10 grid
    x = params[:,i]
    plt.imshow(x.reshape((28, 28)), cmap = "gray", interpolation = "nearest")
    plt.axis("off")
plt.subplots_adjust(wspace=0, hspace=0)
plt.savefig("<filename>.png")
```

5. CNN model

```
from tensorflow import keras
# Set seed for producing results with different weights
tf.random.set_seed(100)
model = Sequential()
# The input layer requires the special input_shape parameter which should match the shape of our training data.
# Define a model with only one layer with 100 hidden units
model.add(Conv2D(32,kernel_size=(3,3),activation="relu",input_shape=(28,28,1)))
model.add(MaxPooling2D(2,2))
model.add(Flatten())
model.add(Dense(10,activation="softmax"))

# Loss is cross-entropy and optimizer is SGD
model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.1,momentum=0), loss='categorical_crossentropy', metrics=['accuracy'])
history = model.fit(X_train.reshape(-1,28,28), onehotlabels_train, batch_size=128, epochs=150, verbose=False, validation_data=(X_test.reshape(-1,28,28),onehotlabels_test))
loss, accuracy = model.evaluate(X_test.reshape(-1,28,28), onehotlabels_test, verbose=False)
```

6. Best CNN model

```
def define_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', input_shape=(28, 28, 1)))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform'))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(100, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(10, activation='softmax'))
    # compile model
    opt = SGD(learning_rate=0.1, momentum=0.5)
    model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
    return model


kfold = KFold(n_splits=5, shuffle=True, random_state=1)
model = define_model()
cnn = model.fit(X_train.reshape(-1,28,28), onehotlabels_train, epochs=150, batch_size=32, validation_data=(X_test.reshape(-1,28,28), onehotlabels_test), verbose=0)
loss, accuracy = model.evaluate(X_test.reshape(-1,28,28), onehotlabels_test, verbose=False)
```

7. Multi-Layer CNN model

```
class  mcn(torch.nn.Module):
        def  __init__(self):
            super(mcn,  self).__init__()
            self.conv1  =  nn.Sequential(
                nn.Conv2d(1,  32,  3,  1,  1),
                nn.ReLU(),
                nn.MaxPool2d(2)
                )
            self.conv2  =  nn.Sequential(
                nn.Conv2d(32,  64,  3,  1,  1),
                nn.ReLU(),
                nn.MaxPool2d(2)
                )
            self.conv3  =  nn.Sequential(
                nn.Conv2d(64,  64,  3,  1,  1),
                nn.ReLU(),
                nn.Flatten()
                )
            self.l1  =  nn.Linear(64*7*14,  512)
            self.l2  =  nn.Linear(512,  20)
        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
```

8. Train-Test

```
def  train_test(mod,  epoch_num,  learning_rate,  momentum,  trn_data,  tst_data):

    loss_fn  =  nn.CrossEntropyLoss()
    optimizer  =  torch.optim.SGD(mod.parameters(),  lr  =  learning_rate,  momentum  =  momentum)

    for  epoch  in  range(epoch_num):
        tr_e  =  0
        ts_e  =  0
        tr_l  =  0
        ts_l  =  0
        for  i,  (data_train,  target_train)  in  enumerate(trn_data):
            trn  =  data_train.float().reshape(100,1,28,56)
            out  =  mod(trn)
            train_pred  =  torch.max(out.data,  1)[1]
            tr_e  +=  (train_pred  !=  target_train).sum()
            batch_l  =  loss_fn(out,  target_train)
            tr_l  +=  batch_l
            optimizer.zero_grad()
            batch_l.backward()
            optimizer.step()
        train_loss.append(tr_l.item()  /  batch_size)
        train_error.append(tr_e.item()  /  batch_size)

        with  torch.no_grad():
            for  j,  (data_test,  target_test)  in  enumerate(tst_data):
                tst  =  data_test.float().reshape(100,1,28,56)
                out_test  =  mod(tst)
                test_pred  =  torch.max(out_test.data,  1)[1]
                ts_e  +=  (test_pred  !=  target_test).sum()
                batch_l_test  =  loss_fn(out_test,  target_test)
                ts_l  +=  batch_l_test
        test_loss.append(ts_l.item()  /  batch_size)
        test_error.append(ts_e.item()  /  batch_size)
```

9. Multi-Layer CNN model with Dropout

```
class mcndrop(torch.nn.Module):
        def __init__(self):
            super(mcndrop, self).__init__()
            self.conv1 = nn.Sequential(
                nn.Conv2d(1, 32, 3, 1, 1),
                nn.ReLU(),
                nn.MaxPool2d(2)
                )
            self.conv2 = nn.Sequential(
                nn.Conv2d(32, 64, 3, 1, 1),
                nn.ReLU(),
                nn.MaxPool2d(2)
                )
            self.conv3 = nn.Sequential(
                nn.Conv2d(64, 64, 3, 1, 1),
                nn.ReLU(),
                nn.Flatten()
                )
            self.l1 = nn.Linear(64*7*14, 512)
            self.drop = nn.Dropout(p=0.5)
            self.l2 = nn.Linear(512, 20)
        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.drop(X)
            X = self.l2(X)
            return X
```