Name	Nimesha Asintha Muthunayake
Reg No	209359G
Data Set	MNIST (Standard)
Git Repository	https://github.com/nimeshacs/Handwritten-digit-classifiers

Image Classifier for the MNIST

Develop a deep learning model for image classification. Include the following in your report. (5 marks)

a. Explanation to your model, design decisions, training-test dataset descriptions and what other factors were considered to improve your model.
b. Accuracy of the model at the end of each epoch.

Answer: (1:a)

The **MNIST** (Modified National Institute of Standards and Technology database) handwritten digit classification problem is a standard dataset used in machine learning It has a training set of 60,000 examples and a test set of 10,000 examples. This data set contains 28x28 grayscale images of the 10 digits.

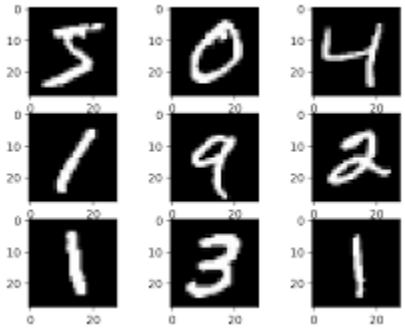


Figure 1.0: MINST Handwriting digits

The task is to classify images of a handwritten digit into one of 10 classes representing integer values from 0 to 9.For this project **Keras API** which is tightly integrated into the TensorFlow ecosystem.

Implementation

1.0 Data preparation

As we already know that each image only contains a hand-drawn digits, that the images all have the same square size of 28×28 pixels (Width and Hight), and all the images are gray scaled by default. Therefore, once we loaded the images into the program, we have to reshape data array to keep data in a single-color channel.

In this data set there are 10 classes which are representing integer values therefore one hot encoding is used for the class elements to convert integers into binary vectors.

Each image pixel values are unsigned integers in the range between black(0) and white (255) ,therefore we should use some scaling technique to rescale the images before modeling them. Images are rescaled to the range 0 and 1 using below technique.

```
train_float = train.astype('float32')
test_float = test.astype('float32')

train_float = train_float / 255.0
test_float = test_float / 255.0
```

1.1 Model Preparation

Convolutional neural network(CNN) is used as molding technique for MNIST classification problem. As we already know convolutional layer and pooling layer are the main two components of the CNN. convolutional layer is used for future extracting layer while pooling layer is used for progressively reducing the spatial size of the representation to reduce the number of parameters and computation in the network.

In this implementation of CNN , 3 * 3 filter is used for the convolutional layer and 32 filters used for max pooling layer. Since this problem based on 10 classes of digits , we need output layer with 10 nodes .added dense layer between feature extractor and the output layer to interpret the features of the images in the model.

ReLU activation function is used for all the convolutional and pooling layers

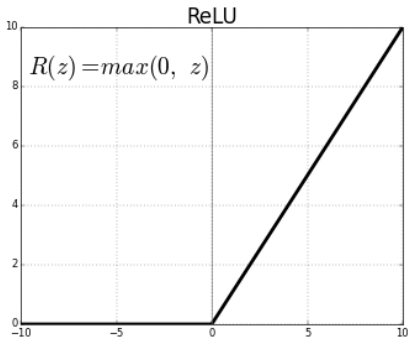


Figure 1.0: ReLU Activation Function

Softmax layer is typically the final output layer in a neural network that performs multi-class classification therefore **Softmax layer** is applied to the network for mapping the non-normalized output of a network to a probability distribution over predicted output classes.

1.2 Taring and Evaluating the Model

Used **20%**(12,000) training data from the data set for training process and data set is shuffled prior to being split. We used **10** training **epochs** with a default batch size of 32 examples. for each fold will be used to evaluate the model both during each epoch.

Next, a summary of the model performance is calculated. We can see in this case, the model has an estimated skill of about **98.68%,** which is reasonable.

```
Accuracy for the CNN model = 98.680
```

b). Accuracy of the model at the end of each epoch.

Answer: (1:b)

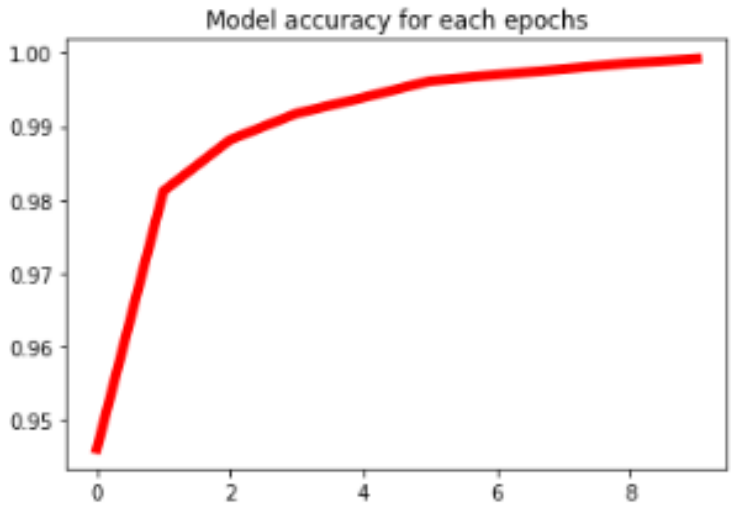


Figure 1.2: Accuracy for each epoch

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] - 37s 616us/step - loss: 0.1636 - accuracy: 0.9496 - val_loss: 0.0796 - val_accuracy: 0.9748
Epoch 2/10
60000/60000 [==============================] - 36s 608us/step - loss: 0.0571 - accuracy: 0.9833 - val_loss: 0.0527 - val_accuracy: 0.9844
Epoch 3/10
60000/60000 [==============================] - 38s 630us/step - loss: 0.0373 - accuracy: 0.9885 - val_loss: 0.0500 - val_accuracy: 0.9845
Epoch 4/10
60000/60000 [==============================] - 37s 611us/step - loss: 0.0268 - accuracy: 0.9915 - val_loss: 0.0484 - val_accuracy: 0.9846
Epoch 5/10
60000/60000 [==============================] - 36s 605us/step - loss: 0.0181 - accuracy: 0.9943 - val_loss: 0.0547 - val_accuracy: 0.9812
Epoch 6/10
60000/60000 [==============================] - 37s 609us/step - loss: 0.0129 - accuracy: 0.9964 - val_loss: 0.0428 - val_accuracy: 0.9862
Epoch 7/10
60000/60000 [==============================] - 39s 649us/step - loss: 0.0093 - accuracy: 0.9974 - val_loss: 0.0411 - val_accuracy: 0.9869
Epoch 8/10
60000/60000 [==============================] - 36s 603us/step - loss: 0.0060 - accuracy: 0.9984 - val_loss: 0.0433 - val_accuracy: 0.9872
Epoch 9/10
60000/60000 [==============================] - 36s 602us/step - loss: 0.0045 - accuracy: 0.9991 - val_loss: 0.0444 - val_accuracy: 0.9875
Epoch 10/10
60000/60000 [==============================] - 36s 605us/step - loss: 0.0028 - accuracy: 0.9996 - val_loss: 0.0457 - val_accuracy: 0.9868
Accuracy for the CNN model = 98.680
```

2. Add random noise to the training and test datasets and report the accuracy. You may use the following sample code to add random noise. You may vary the noise_factor to control the amount of noise added to the datasets and report the results. (3 marks)

Answer:(2)

Added gaussian random noise to the images, below two images show the digits before adding nose and after adding noise to the data set

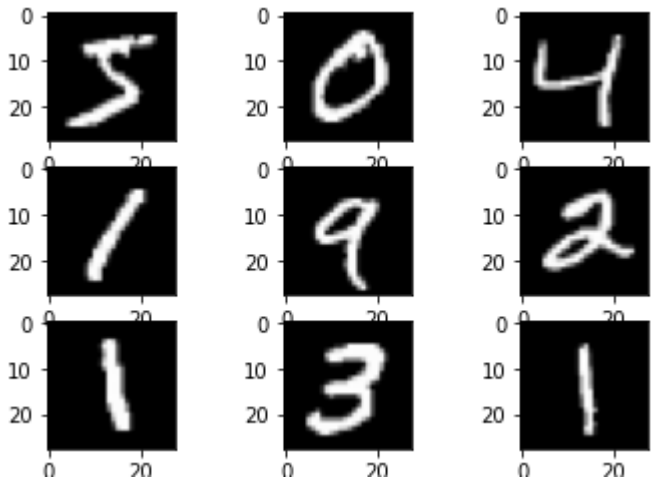


Figure 1.3: Before adding noise

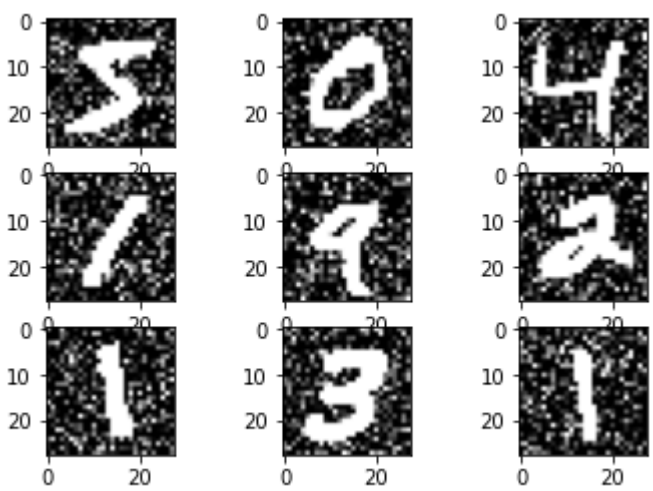


Figure 1.4: After adding noise

```
from keras.datasets import mnist
from matplotlib import pyplot
import numpy as np

(trainX, trainy), (testX, testy) = mnist.load_data()

print('Train: X=%s, y=%s' % (trainX.shape, trainy.shape))
print('Test: X=%s, y=%s' % (testX.shape, testy.shape))

noise_factor = 0.5
x_train_noisy = trainX + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=trainX.shape)
x_train_noisy = np.clip(x_train_noisy, 0., 1.)

#Show images before adding noise
for i in range(9):
  pyplot.subplot(330 + 1 + i)
  pyplot.imshow(trainX[i], cmap=pyplot.get_cmap('gray'))
pyplot.show()

#Show images after adding noise
for i in range(9):
  pyplot.subplot(330 + 1 + i)
  pyplot.imshow(x_train_noisy[i], cmap=pyplot.get_cmap('gray'))
pyplot.show()
```

```
noise_factor = 0.5
X_train = X_train + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=X_train.shape)
X_test  = X_test + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=X_test.shape)
X_train = np.clip(X_train, 0., 1.)
X_test  = np.clip(X_test, 0., 1.)
```

After adding noise as noise_factor = 0.5 , It reduced the model accuracy to **94.0**

```
Accuracy for the CNN model = 94.080
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] - 13s 223us/step - loss: 0.4376 - accuracy: 0.8582 - val_loss: 0.2381 - val_accuracy: 0.9254
Epoch 2/10
60000/60000 [==============================] - 7s 120us/step - loss: 0.1944 - accuracy: 0.9392 - val_loss: 0.2010 - val_accuracy: 0.9363
Epoch 3/10
60000/60000 [==============================] - 7s 121us/step - loss: 0.1263 - accuracy: 0.9592 - val_loss: 0.1922 - val_accuracy: 0.9406
Epoch 4/10
60000/60000 [==============================] - 7s 121us/step - loss: 0.0796 - accuracy: 0.9741 - val_loss: 0.1841 - val_accuracy: 0.9449
Epoch 5/10
60000/60000 [==============================] - 7s 121us/step - loss: 0.0466 - accuracy: 0.9861 - val_loss: 0.2016 - val_accuracy: 0.9425
Epoch 6/10
60000/60000 [==============================] - 7s 121us/step - loss: 0.0226 - accuracy: 0.9943 - val_loss: 0.2183 - val_accuracy: 0.9421
Epoch 7/10
60000/60000 [==============================] - 7s 121us/step - loss: 0.0098 - accuracy: 0.9984 - val_loss: 0.2273 - val_accuracy: 0.9462
Epoch 8/10
60000/60000 [==============================] - 7s 120us/step - loss: 0.0037 - accuracy: 0.9998 - val_loss: 0.2576 - val_accuracy: 0.9425
Epoch 9/10
60000/60000 [==============================] - 7s 121us/step - loss: 0.0017 - accuracy: 1.0000 - val_loss: 0.2595 - val_accuracy: 0.9467
Epoch 10/10
60000/60000 [==============================] - 7s 120us/step - loss: 0.0011 - accuracy: 1.0000 - val_loss: 0.2684 - val_accuracy: 0.9466
Accuracy for the CNN model = 94.660
```

3. Explain how the accuracy of the image classifier can be improved for the scenario where the dataset includes noise as in part 2 above. You may implement a new model with the improvements. (7 marks)

Answer :(3)

Data Augmentation and Denoising Autoencoder

Data Augmentation and Denoising Autoencoder can be used to handle noisy data augmentation is used to avoid overfitting problem and Autoencoder can be used to compress and reduce dimensionality of data.

Adding noise means expands the size of the training dataset. Each time a training sample is exposed to the model, random noise is added to the input variables making them different every time it is exposed to the model. In this way, adding noise to input samples is a simple form of augmentation. Data augmentation is used in order to avoid overfitting problem of the MNIST data.

" Injecting noise in the input to a neural network can also be seen as a form of data augmentation."

— Page 241, Deep Learning, 2016. (Adaptive Computation and Machine Learning series) by Ian Goodfellow (Author), Yoshua Bengio (Author), Aaron Courville (Author)

)

Adding noise means that the network is less able to memorize training samples because they are changing all the time, resulting in smaller network weights and a more robust network that has lower generalization error. Image data augmentation is a technique that can be used to artificially expand the size of a training dataset by creating modified versions of images in the dataset.

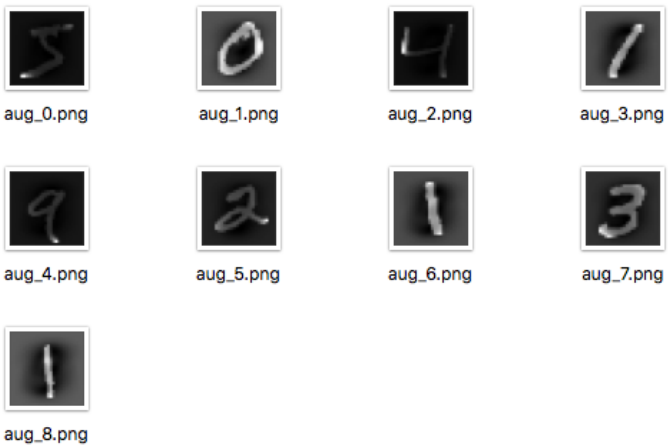
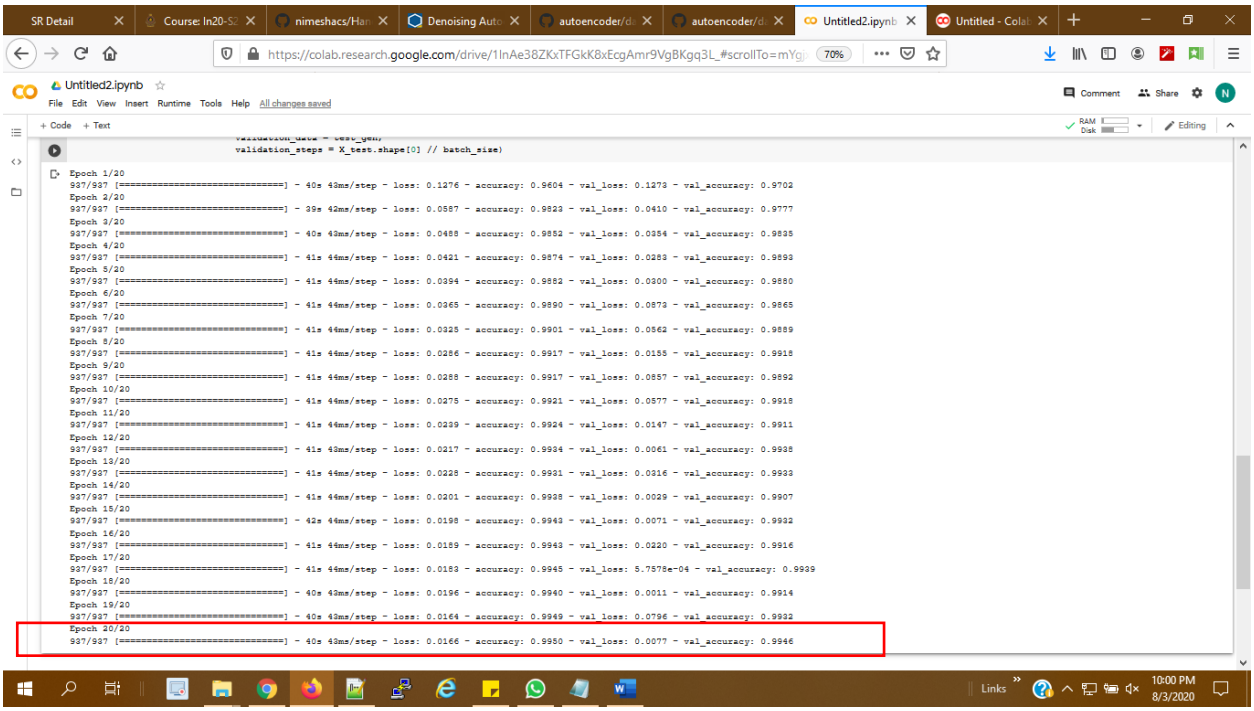


Figure 1.5: Augmented Images

Accuracy After augmentation is 99.46

```
val_accuracy: 0.9946
```



Autoencoder

The most important solution here is to insert a *bottleneck layer* in the middle of the network. This bottleneck layer is given a dimensionality much smaller than the input and output. This forces the network to not just pass the input through to the output because the bottleneck layer is intentionally **too small to contain all the information in the input image**.

Autoencoders are made up of two parts Encoders and Decoders as in above chart. Autoencoders decrease the number of dimentions required to store the Information in the model, and Decoders try to get this information back from the compressed form. This method can be used to improve our model for MNIST with noise.

Note : After adding noise to the images the accuracy was reduces to **94%** when Data ***Augmentation*** is added to the model accuracy was increased to **99.46%** therefore didn't apply Autoencoder to the code base but as a future work I am planning to add ***Autoencoder*** to compress and reduce dimensionality of data for better accuracy.