IMAGE RECOGNITION MODEL

***** The Code for the Image Recognition Model can be accessed from my Github Link :** https://github.com/jonygeta/Data622FinalProject/blob/master/MNIST.py) ***

Introduction

Image recognition refers to the task of inputting an image into a neural network and having it output some kind of label for that image. The label that the network outputs will correspond to a pre-defined class. There can be multiple classes that the image can be labeled as, or just one. If there is a single class, the term "recognition" is often applied, whereas a multi-class recognition task is often called "classification". [1] In this regard It is advisable to mention machine learning work flow as it will enable us to build image recognition model . There are generally 4 work flows : data preparation, creating the model, training the model, model evaluation and testing the model**.**

In this Project I have followed the work flows and have tested the accuracy level at the end . The project is based on the Minst data set sourced from http://yann.lecun.com/exdb/mnist/.

The MNIST database (Modified National Institute of Standards and Technology database) of handwritten digits consists of a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. Additionally, the black and white images from NIST were size-normalized and centered to fit into a 28x28 pixel bounding box and anti-aliased, which introduced grayscale levels.

This database is well liked for training and testing in the field of machine learning and image processing. It is a remixed subset of the original NIST datasets. One half of the 60,000 training images consist of images from NIST's testing dataset and the other half from Nist's training set. The 10,000 images from the testing set are similarly assembled.

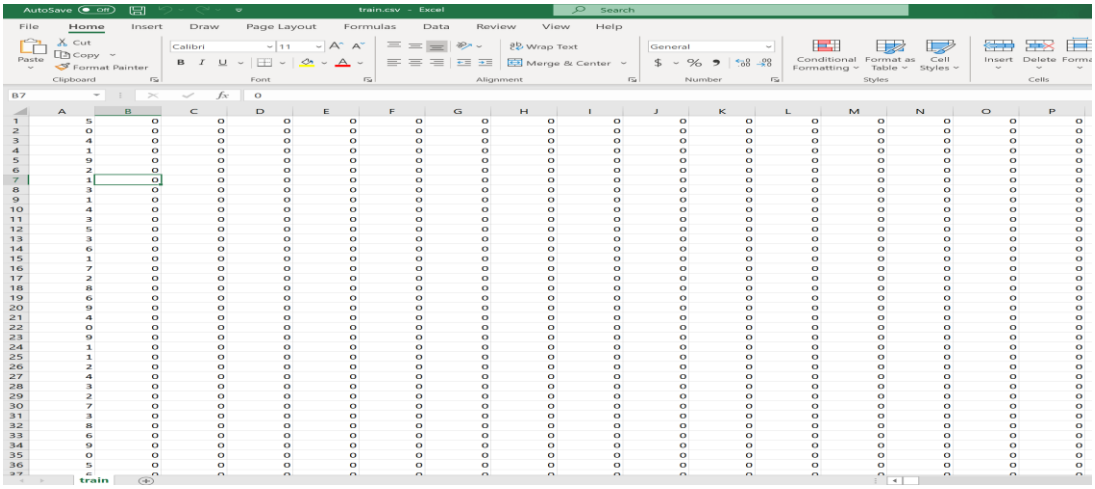
Steps taken in building the Model :

1. Importing Libraries
2. Loading and Preprocessing Data
3. Creating a validation set
4. Defining the model structure
5. Training the model
6. Making predictions

[1] https://stackabuse.com/image-recognition-in-python-with-tensorflow-and-keras/. Date accessed December 10, 2019.

Reading the MNIST data set

Glimpse of the data set .



```
import numpy as np
import matplotlib.pyplot as plt
image_size = 28 # width and length
no_of_different_labels = 10 #  i.e. 0, 1, 2, 3, ..., 9
image_pixels = image_size * image_size
data_path = "data/mnist/"
train_data = np.loadtxt(data_path + "mnist_train.csv",
delimiter=",")
test_data = np.loadtxt(data_path + "mnist_test.csv",
delimiter=",")
test_data[:10]
```

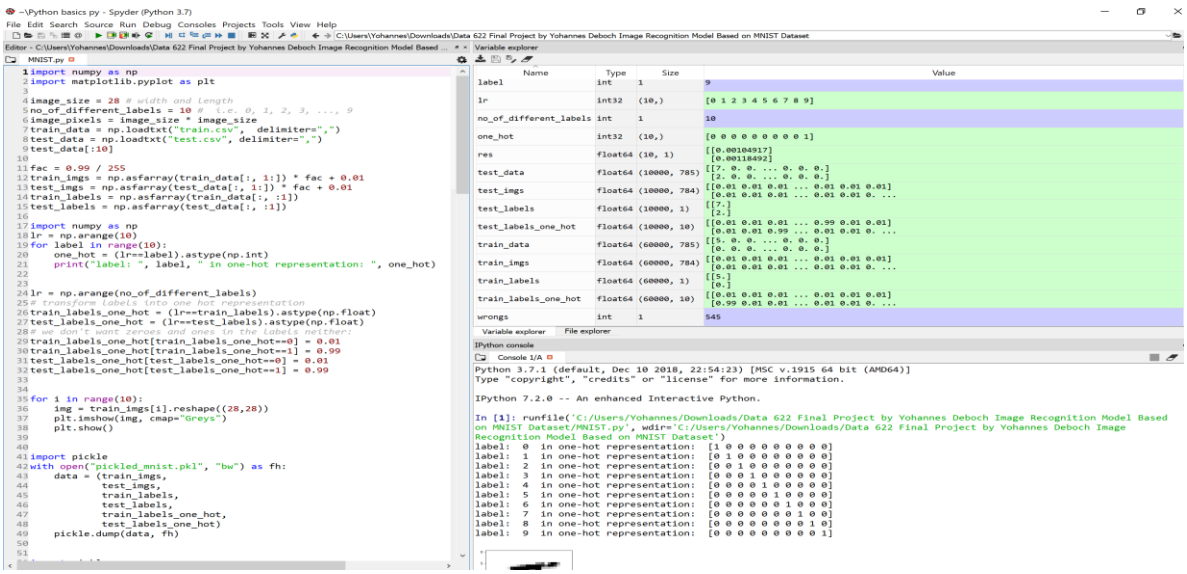
The images of the MNIST dataset are greyscale and the pixels range between 0 and 255 including both bounding values. We will map these values into an interval from [0.01, 1] by multiplying each pixel by 0.99 / 255 and adding 0.01 to the result.

```
fac = 0.99 / 255
train_imgs = np.asfarray(train_data[:, 1:]) * fac + 0.01
test_imgs = np.asfarray(test_data[:, 1:]) * fac + 0.01
train_labels = np.asfarray(train_data[:, :1])
test_labels = np.asfarray(test_data[:, :1])
```

We need the labels in our calculations in a one-hot representation. We have 10 digits from 0 to 9, i.e. lr = np.arange(10).

```
lr = np.arange(10)
for label in range(10):
one_hot = (lr==label).astype(np.int)
print("label: ", label, " in one-hot representation: ", one_hot)
```

Screenshot Image from the Model Testing



Dumping the Data for Faster Reload

We will save the data in binary format with the dump function from the pickle module:

```
import pickle
with open("data/mnist/pickled_mnist.pkl", "bw") as fh:
   data = (train_imgs,
           test_imgs,
           train_labels,
           test_labels,
           train_labels_one_hot,
```

```
        test_labels_one_hot)
    pickle.dump(data, fh)
```

Classifying the Data

Sigmoid:

The forward pass on a single example x executes the following computation:

$$\hat{y}=\sigma(w^Tx+b).$$

Here σ is the sigmoid function:

$$\sigma(z)=11+e^{-z}.$$

The main reason why we use sigmoid function is because it exists between (0 to 1). Therefore, it is especially used for models where we have to predict the probability as an output. Since probability of anything exists only between the range of 0 and 1, sigmoid is the right choice. The function is differentiable.

Back Propagation:

For backpropagation, we'll need to know how L changes with respect to each component w_j of w. That is, we must compute each $\partial L/\partial w_j$

Focusing on a single example will make it easier to derive the formulas we need. Holding all values except w_j fixed, we can think of L as being computed in three steps: $w_j \to z \to \hat{y} \to L$. The formulas for these steps are:

$$z\hat{y}L(y,\hat{y})=w^Tx+b,=\sigma(z),=-y\log(\hat{y})-(1-y)\log(1-\hat{y}).$$

And the chain rule tells us:

$$\partial L\partial w_j=\partial L\partial\hat{y}\partial\hat{y}\partial z\partial z\partial w_j.$$

Looking at $\partial L/\partial\hat{y}$ first:

$$\partial L\partial\hat{y}=\partial\partial\hat{y}(-y\log(\hat{y})-(1-y)\log(1-\hat{y}))=-y\partial\partial\hat{y}\log(\hat{y})-(1-y)\partial\partial\hat{y}\log(1-\hat{y})=-y\hat{y}+(1-y)1-\hat{y}=\hat{y}-y\hat{y}(1-\hat{y}).$$

Next we want $\partial\hat{y}/\partial z$:

$$\partial\partial z\sigma(z)=\partial\partial z(11+e^{-z})=-1(1+e^{-z})2\partial\partial z(1+e^{-z})=e^{-z}(1+e^{-z})2=11+e^{-z}e^{-z}1+e^{-z}=\sigma(z)e^{-z}1+e^{-z}=\sigma(z)(1-11+e^{-z})=\sigma(z)(1-\sigma(z))=\hat{y}(1-\hat{y}).$$

Lastly we tackle $\partial z/\partial w_j$:

$$\partial\partial w_j(w^Tx+b)=\partial\partial w_j(w_0x_0+\ldots+w_nx_n+b)=w_j.$$

Finally we can substitute into the chain rule to find:

$$\partial L \partial w_j = \partial L \partial \hat{y} \partial \hat{y} \partial z \partial z \partial w_j = \hat{y} - y \hat{y}(1-\hat{y})\hat{y}(1-\hat{y}) w_j = (\hat{y} - y) w_j.$$

In vectorized form with m training examples this gives us:

$$\partial L \partial w = 1m X(\hat{y} - y)^T.$$

What about $\partial L / \partial b$? A very similar derivation yields, for a single example:

$$\partial L \partial b = (\hat{y} - y).$$

Which in vectorized form amounts to:

$$\partial L \partial b = 1m \sum_{i=1m} (\hat{y}_{(i)} - y_{(i)}).$$

In our code we'll label these gradients according to their denominators, as dW and db. So for backpropagation we'll compute

dW = (1/m) * np.matmul(X, (A-Y).T) and db = (1/m) * np.sum(A-Y, axis=1, keepdims=True).

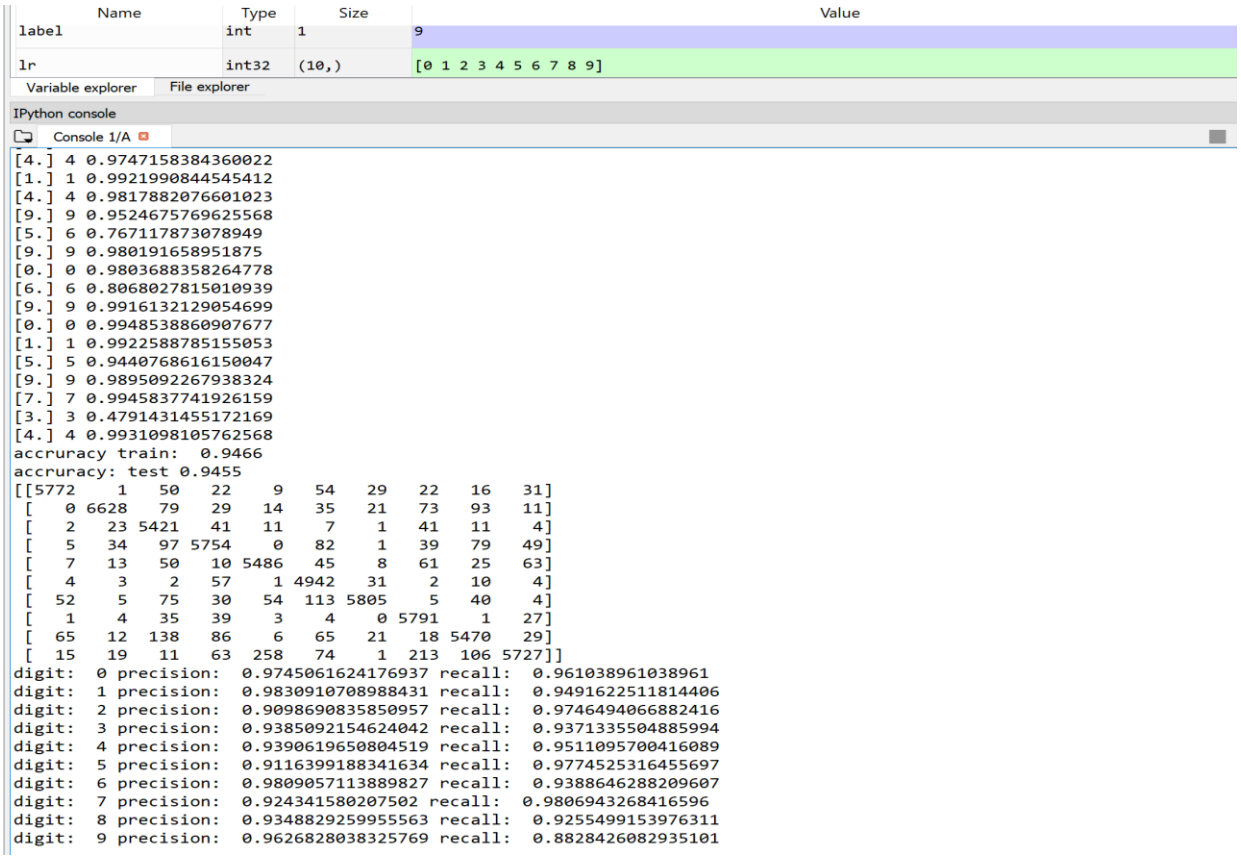
```
def sigmoid(x):
   return 1 / (1 + np.e ** -x)
activation_function = sigmoid
from scipy.stats import truncnorm
def truncated_normal(mean=0, sd=1, low=0, upp=10):
   return truncnorm((low - mean) / sd,
                    (upp - mean) / sd,
                    loc=mean,
                    scale=sd)
class NeuralNetwork:

   def __init__(self,
                no_of_in_nodes,
                no_of_out_nodes,
                no_of_hidden_nodes,
                learning_rate):
      self.no_of_in_nodes = no_of_in_nodes
      self.no_of_out_nodes = no_of_out_nodes
      self.no_of_hidden_nodes = no_of_hidden_nodes
      self.learning_rate = learning_rate
      self.create_weight_matrices()

   def create_weight_matrices(self):
      """
      A method to initialize the weight
      matrices of the neural network
      """
      rad = 1 / np.sqrt(self.no_of_in_nodes)
      X = truncated_normal(mean=0,
                           sd=1,
                           low=-rad,
                           upp=rad)
      self.wih = X.rvs((self.no_of_hidden_nodes,
                              self.no_of_in_nodes))
      rad = 1 / np.sqrt(self.no_of_hidden_nodes)
```

```
        X = truncated_normal(mean=0, sd=1, low=-rad, upp=rad)
        self.who = X.rvs((self.no_of_out_nodes,
                          self.no_of_hidden_nodes))


    def train(self, input_vector, target_vector):
        """
        input_vector and target_vector can
        be tuple, list or ndarray
        """

        input_vector = np.array(input_vector, ndmin=2).T
        target_vector = np.array(target_vector, ndmin=2).T

        output_vector1 = np.dot(self.wih,
                                input_vector)
        output_hidden = activation_function(output_vector1)

        output_vector2 = np.dot(self.who,
                                output_hidden)
        output_network = activation_function(output_vector2)

        output_errors = target_vector - output_network
        # update the weights:
        tmp = output_errors * output_network \
              * (1.0 - output_network)
        tmp = self.learning_rate  * np.dot(tmp,
                                           output_hidden.T)
        self.who += tmp
        # calculate hidden errors:
        hidden_errors = np.dot(self.who.T,
                               output_errors)
        # update the weights:
        tmp = hidden_errors * output_hidden * \
              (1.0 - output_hidden)
        self.wih += self.learning_rate \
                    * np.dot(tmp, input_vector.T)



    def run(self, input_vector):
        # input_vector can be tuple, list or ndarray
        input_vector = np.array(input_vector, ndmin=2).T
        output_vector = np.dot(self.wih,
                               input_vector)
        output_vector = activation_function(output_vector)

        output_vector = np.dot(self.who,
                               output_vector)
        output_vector = activation_function(output_vector)

        return output_vector

    def confusion_matrix(self, data_array, labels):
        cm = np.zeros((10, 10), int)
        for i in range(len(data_array)):
```

```
            res = self.run(data_array[i])
            res_max = res.argmax()
            target = labels[i][0]
            cm[res_max, int(target)] += 1
        return cm
    def precision(self, label, confusion_matrix):
        col = confusion_matrix[:, label]
        return confusion_matrix[label, label] / col.sum()

    def recall(self, label, confusion_matrix):
        row = confusion_matrix[label, :]
        return confusion_matrix[label, label] / row.sum()


    def evaluate(self, data, labels):
        corrects, wrongs = 0, 0
        for i in range(len(data)):
            res = self.run(data[i])
            res_max = res.argmax()
            if res_max == labels[i]:
                corrects += 1
            else:
                wrongs += 1
        return corrects, wrongs

ANN = NeuralNetwork(no_of_in_nodes = image_pixels,
                    no_of_out_nodes = 10,
                    no_of_hidden_nodes = 100,
                    learning_rate = 0.1)


for i in range(len(train_imgs)):
    ANN.train(train_imgs[i], train_labels_one_hot[i])
for i in range(20):
    res = ANN.run(test_imgs[i])
    print(test_labels[i], np.argmax(res), np.max(res))
```

Screenshot Image showing the Accuracy testing of the model :

```
accruracy train: 0.9466
accruracy: test 0.9455
```



Name	Type	Size	Value
label	int	1	9
lr	int32	(10,)	[0 1 2 3 4 5 6 7 8 9]

Variable explorer | File explorer

IPython console

Console 1/A

```
[4.] 4 0.9747158384360022
[1.] 1 0.9921990844545412
[4.] 4 0.9817882076601023
[9.] 9 0.9524675769625568
[5.] 6 0.767117873078949
[9.] 9 0.980191658951875
[0.] 0 0.9803688358264778
[6.] 6 0.8068027815010939
[9.] 9 0.9916132129054699
[0.] 0 0.9948538860907677
[1.] 1 0.9922588785155053
[5.] 5 0.9440768616150047
[9.] 9 0.9895092267938324
[7.] 7 0.9945837741926159
[3.] 3 0.4791431455172169
[4.] 4 0.9931098105762568
accruracy train:  0.9466
accruracy: test 0.9455
[[5772    1   50   22    9   54   29   22   16   31]
 [   0 6628   79   29   14   35   21   73   93   11]
 [   2   23 5421   41   11    7    1   41   11    4]
 [   5   34   97 5754    0   82    1   39   79   49]
 [   7   13   50   10 5486   45    8   61   25   63]
 [   4    3    2   57    1 4942   31    2   10    4]
 [  52    5   75   30   54  113 5805    5   40    4]
 [   1    4   35   39    3    4    0 5791    1   27]
 [  65   12  138   86    6   65   21   18 5470   29]
 [  15   19   11   63  258   74    1  213  106 5727]]
digit:  0 precision:  0.9745061624176937 recall:  0.961038961038961
digit:  1 precision:  0.9830910708988431 recall:  0.9491622511814406
digit:  2 precision:  0.9098690835850957 recall:  0.9746494066882416
digit:  3 precision:  0.9385092154624042 recall:  0.9371335504885994
digit:  4 precision:  0.9390619650804519 recall:  0.9511095700416089
digit:  5 precision:  0.9116399188341634 recall:  0.9774525316455697
digit:  6 precision:  0.9809057113889827 recall:  0.9388646288209607
digit:  7 precision:  0.924341580207502 recall:  0.9806943268416596
digit:  8 precision:  0.9348829259955563 recall:  0.9255499153976311
digit:  9 precision:  0.9626828038325769 recall:  0.8828426082935101
```

A neural network by definition consists of more than just 1 cell. Since we want to recognize 10 different handwritten digits our network needs 10 cells, each representing one of the digits 0-9.

The target output is the binary representation of our desired output. Since our desired output (provided via the MNIST label file) is always a number between 0-9 the target output is modeled as vector of 10 values, each either 0 or 1.

The network is trained by looping through all 60,000 images and making the network classify each image. Its classification is then compared with the correct answer (given in the label file) and cell weights are adjusted according to the difference between the two (the error).

The optimal number of hidden units could easily be smaller than the number of inputs, there is no rule like multiply the number of inputs with N. If there are a lot of training examples, you can use multiple hidden units here I have used 100 hidden units. Usually one hidden layer is used for simple tasks. In order to secure the ability of the network to generalize the number of nodes has to be kept as low as possible. If you have a large excess of nodes, you network becomes a memory bank and less efficient.

Future prospects of Image Recognition model

The image recognition market is estimated to reach USD 38.9 billion by 2021, compared to USD 15.9 billion in 2016 — more than 100% increase for the projected period, according to data by Markets and Markets research. Companies in e-commerce, automotive, content sharing, healthcare and security are rapidly implementing image recognition and driving the adoption of the technology. Other contributing factors are the sophistication of facial recognition, content moderation and other algorithms, the rise in popularity of media cloud services, the surge in mobile devices equipped with cameras and hardware technological advancements.[2]

I personally believe there is a vast opportunity in the image recognition industry. I believe there is a huge potential of innovation in this regards by considering market based and application based approaches .

Image Recognition Market Based on Types:

- Optical Character Recognition
- Code Recognition
- Digital Image Processing
- Pattern Recognition
- Object Recognition
- Facial Recognition

Image Recognition Market Based on Applications:

- Scanning & Imaging
- Image Search
- Security & Surveillance
- Augmented Reality
- Marketing & Advertising

[2] https://imagga.com/blog/image-recognition-trends-in-2020/.