Image Classification of Handwritten Numerals Using Neural Network

Christopher Casey, Sanjeev K Sinha, Shaheen Pouya

December 6, 2022

1 Introduction

One of the most customary and useful applications of artificial intelligence is image classification. Its usage covers a broad area from autonomous vehicles to face recognition in smart phones, from handwriting detection to customer segmentation. These applications also are sided with countless of medical applications while there are tons of researches in this area to include even more uses.
Neural Network is a type of interconnected system using 'neurons' as the nodes to exchange information with each other. This network consists of one input layer, one output layer, and a desirable number of hidden layers in between. The connection between every two nodes from different layers creates a numeric weight that will be tuned when the network is being 'trained'. At the moment, this network helps us to have some of the sharpest models in the machine-learning world when presented with any kind of data with a recognizable pattern. One of the most common uses of neural networks (deep learning) is image classification.
Image classification/ image recognition/pattern recognition is a very challenging problem. The domain of image classification is vast. This project was mainly confined only to classifying images of handwritten digits.
The primary application domain is financial documents, postal services, and phone numbers in handwritten forms/applications that require capturing of handwritten numbers such as amounts in checks, figures in handwritten financial statements, handwritten ZIPs used in addresses, and handwritten

phone numbers on applications or business cards.
In general, the recognition of handwritten numerals is a benchmark problem of Pattern Recognition. In comparison to the problem of recognition of printed numerals, the issue of handwritten numeral recognition is compounded due to variations in the shapes and sizes of handwritten characters.

2 Methodology

This project was mainly built on Python programming language with the usage of some packages that are listed in this report. The task of classifying images was mostly achieved by the generation of testing parameters such as accuracy and f1 score. The main processes for implementation of this project are based on three main stages:

- Data Loading and Pre-Processing
- Model Building, Activation and Back Propagation
- Testing and Reporting

This work is formulated to create a working model of image classification without using any specialized existing packages that are designed for deep learning. As it uses the main regulations in MLP neural networks, the model could be used in any other feasible classification problem with a little modification. Every step in the programming of this project was monitored by Github version control. After the implementation of the whole MLP process in this study, we added noise in the data to check if the model can handle that.

3 Data Loading and Pre-Processing

MNIST dataset

The data used in this research is publicly available MNIST handwritten images (available at: yann.lecun.com/exdb/mnist) including inputs as 28 by 28 images and digits – 0 to 9 - as labels. MNIST is abbreviated by Modified National Institute of Standards and Technology which is a smaller set of NIST dataset.

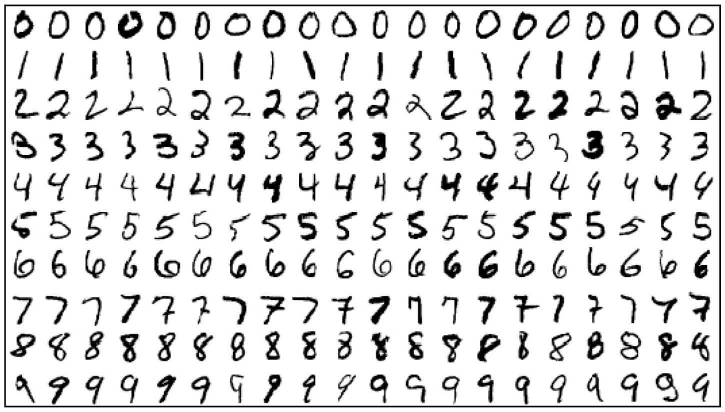
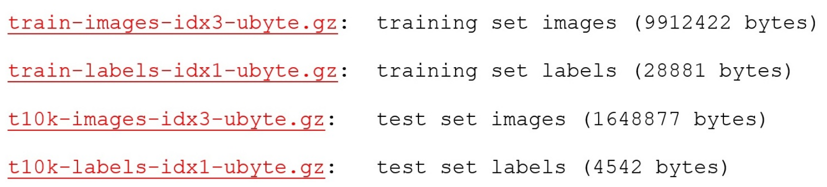


Figure 1: Some instances of MNIST handwritten images.

It consists of 60,000 small images with square 28 by 28-pixel dimensions in grayscale. It is known as the most used dataset for learning image recognition of handwriting as it has almost unique images in every frame as could be seen in figure 1 above. Each of the handwritten images has its correct numerical label attached to it. This makes creating the training algorithm simpler for recognizing the correct label for each image.

Downloading data

The link for MNIST dataset contains 4 files in ".gz" zipped format which would be downloaded and used in the model as could be seen in figure 1:



```
train-images-idx3-ubyte.gz:  training set images (9912422 bytes)
train-labels-idx1-ubyte.gz:  training set labels (28881 bytes)
t10k-images-idx3-ubyte.gz:   test set images (1648877 bytes)
t10k-labels-idx1-ubyte.gz:   test set labels (4542 bytes)
```

Figure 2: The files in MNIST website.

Data Loading packages

The main packages used in the data loading class were:

- Struct: Extracting data into arrays, to convert data into python values.
- Numpy and Pandas: Data process and management, changing arrays and storing them.
- Matplotlib: Top generate graphical outputs (for checking data loader).

Building data loader model

There are mainly four steps in the data loading class/model.

1. Reading from files and storing them as arrays

```
def read_image_labels(self, labels_path, images_path):
    labels = []
    images = []
    with open(labels_path, 'rb') as labels_file:
        magic, size = struct.unpack(">II", labels_file.read(8))
        labels_data = array("B", labels_file.read())

    with open(images_path, 'rb') as images_file:
        magic, size, rows, cols = struct.unpack(">IIII", images_file.read(16))
        image_data = array("B", images_file.read())
```

2. Resizing the arrays into 28*28

```
for i in range(size):
    image = np.array(image_data[i*rows*cols: (i+1)*rows*cols])
    image = image.reshape(28, 28)
    images[i][:] = image
```

3. Defining train and test loading functions

```
def get_train_data(self):
    x_train, y_train = self.read_image_labels(self.training_images_path, self.training_labels_path)
    return x_train, y_train

def get_test_data(self):
    x_test, y_test = self.read_image_labels(self.test_images_path, self.test_labels_path)
    return x_test, y_test
```

4. Defining test functions to see if loading works properly

```
def simple_show(self):
    images, _ = self.get_train_data()
    for i in range(9):
        plt.subplot(330+1 + i)
        plt.imshow(images[i], Decmap=plt.get_cmap('gray'))
    plt.show()
```

Visualizing one instance of the dataset

After defining the data loader model as a class we can use it on the main model by one single line of creating an object. Using the above (simple show) function we can see some frames from the data which could be seen in figure 1 below. After the loading process, the data is ready to be used in the main model.

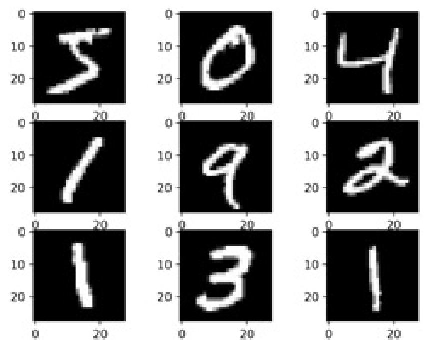


Figure 3: Some hand writings that are derived from data set

4 Multi-Layer Perceptron

For this project, a strict adherence to an MLP structure was maintained. Meaning that there are 3 layers that make up this projects model, an input layer, hidden layer, and output layer. The input layer has 784 neurons representing the 28x28 pixel grid of gray-scale values making up each of the character images being interpreted. Next, the hidden layer, made up of 256 neurons in this project, this layer directs values through an activation function and performs non-linear transformations on the data. Finally, the output layer, contains the probabilistic value that each image is a specific 0-9 classification. The neuron with the highest value decides what classification the network thinks the character image is. A single neuron represents a single classification. There being 10 characters, 0-9, that an image can be classified as, this projects model had 10 neurons in the output layer.

4.1 Layer Initialization

In order to implement an MLP programmatically, it has to be understood how data flows throughout the model to produce values indicative of a final classification. This flow and process will be discussed thoroughly in the following section. However, understand that the only values stored in this model are the weights that represent the connections from layer to layer. All the other data is calculated in real time when the initial input values are fed to the network. Every neuron in a layer has a weighted connection to each of the neurons in the next layer. There being 3 layers, 2 matrices are necessary to store the weights of the connection between them. For example, this model has an input layer of 784 neurons and a hidden layer of 256 neurons. So, the size of the matrix holding the weighted connections is (784, 256). The size of the second matrix would be calculated the same way with the hidden layer and the output layer. For this model we randomly and uniformly initialize these matrices with starting values between -1 and 1. Further the initial value is divided by the square root. Cautious initialization is very important when working with neural networks as result of the stochastic gradient descent functions. In order to avoid the exploding gradient descent function and the vanishing gradient descent problem, these initialization formulas and functions were used. This was all taken into account when initializing the weights and the layers they represented.

4.2 Forward Pass

For this section, a discussion on how data flows through the MLP model as it attempts to make classifications will be taken primarily from the perspective of how it is implemented in Python. Passing in the array of 784 gray-scale values, the dot product is taken with the first weight matrix. This resultant array is then passed into the hidden layers activation function, the sigmoid function. Again, the dot product of the activation values, from the hidden layer, and next matrix of weights is taken. This result is then fed into the softmax function to finally give the normalized and probabilistic value that each image is a certain classification. So an array of 10 outputs is the result of the neural network, and the max value in the array of size 10 decides what classification it is. More of the theoretical math is portrayed in the next section.

5 Backpropagation

The backpropagation of a neural network is the backbone of the model. It starts from the end point of the neural network, that is, output. The backpropagation aims to adjust the weights of nodes of the neural network in such a manner that the desired relationship between the input and output is achieved to the extent possible depending upon iterations and hyperparameters. Neural network uses two steps – feedforward and backpropagation. During feedforward, input is fed to the network. These inputs are processed by the nodes at each layer according to the activation function, weights and biases. The output produced at each output node by the network in this manner is compared to the expected output. The difference between the target and the actual output is error. Since the output is based on the weights, biases and activation function and each connected node contributed to the final output, the error is also part of each node's contribution. We want the neural network to deliver the expected output. This essentially means that whatever error has been contributed by each node during feedforward step must be nullified. This exercise is taken care of by backpropagation. In backpropagation, the error determined at the output is used to modify the weights at each node without changing the activation function. This process is carried out for each training instance so that the overall error is minimized. We will explain the backpropagation on the basis of a typical network. Let us look at the following network:

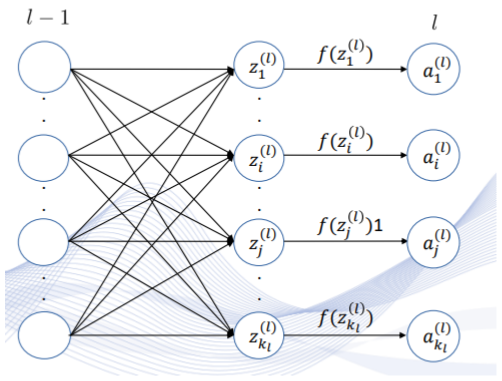
In a neural network, in general, we can write values at each node on layer l as -

$$z_j^{(l)} = \mathbf{w}_j^{(l)T}\mathbf{a}^{(l-1)}$$
$$a_j^{(l)} = f(\mathbf{w}_j^{(l)T}\mathbf{a}^{(l-1)})$$

This can be written in more compact form using vector notation for all the nodes at one layer –

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)T}\mathbf{a}^{(l-1)})$$
$$\mathbf{a}^{(l)} = f(\mathbf{W}^{(l)T}\mathbf{a}^{(l-1)})$$

For hidden layers, sigmoid function and for the last layer, softmax func-



tions in the network are as follows -

$$\text{Sigmoid: } \sigma(x) = \frac{1}{1+e^{(-x)}}$$

$$\text{Softmax: } \hat{y}_i = g(a_i) = \frac{e^{a_i}}{\sum_k e^{a_k}}$$

Error is determined using actual output and expected output as below:

$$\text{Error: } J = \hat{y}_i - y_i$$

Differentiation $\frac{\partial J}{\partial \mathbf{W}} = 0$ provides us points corresponding to minima. Since there is no closed form solution to the problem at hand, numerical method is resorted to solve the problem.

We search optimal values of W iteratively using the following relationship:

$$\mathbf{W}(t+1) = \mathbf{W}(t) + \Delta\mathbf{W}$$

On using gradient descent with learning rate μ, we have the following expression:

$$\mathbf{W}(t+1) = \mathbf{W}(t) - \mu\nabla_{\mathbf{w}} J(\mathbf{W})$$

This provides us the equation for updating j^{th} neuron in the l^{th} layer as below:

$$\mathbf{w}_j^{(l)}(t+1) = \mathbf{w}_j^{(l)}(t) - \mu\frac{\partial J}{\partial \mathbf{w}_j^{(l)}}$$

We use following equations for updating deltas on the basis of derivative of activation functions:

$$\boldsymbol{\delta}^{(l)} = \left[\mathbf{w}^{(l+1)}\boldsymbol{\delta}^{(l+1)}\right] \cdot \frac{df(\mathbf{z}^{(l)})}{d\mathbf{z}^{(l)}}$$

$$\frac{\partial J}{\partial w_{jk}^{(l)}} = \frac{\partial J}{\partial z_j^{(l)}}\frac{\partial z_k^{(l)}}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} a_k^{(l-1)}$$

In order to use the above updations, we use following derivatives:

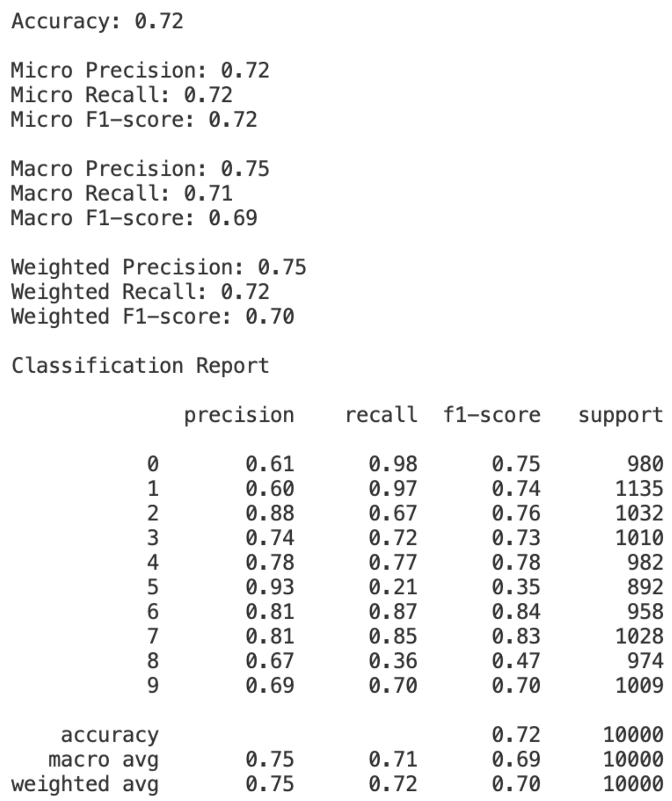
$$\text{The derivative of the sigmoid: } \frac{d\sigma(x)}{dx} = \sigma(x)(1-\sigma(x))$$

The derivative of the softmax: $\frac{\partial}{\partial x_k}\sigma(\mathbf{x}, i) = \sigma(\mathbf{x}, i)(\delta_{ik} - \sigma(\mathbf{x}, k))$ where δ_{ik} is Kronecker delta.

6 Output

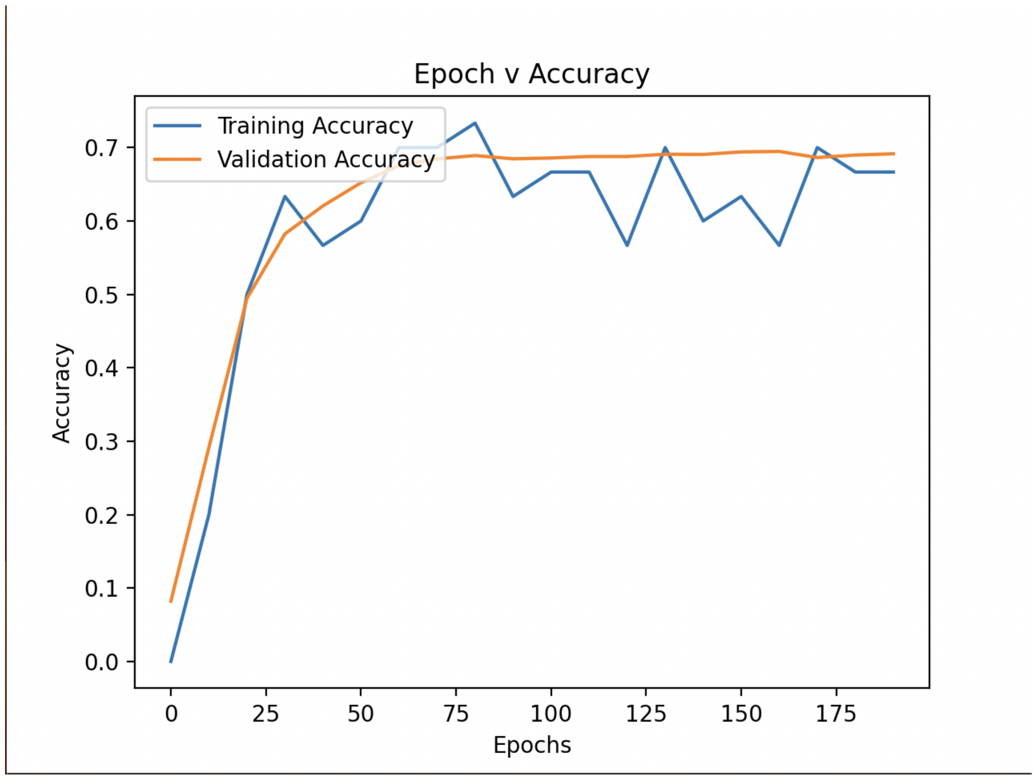
Upon completion of training and testing the MLP, this project spits out a number of different plots and statistical analyses. As mentioned in the proposal presentation analyses that are included is F1 Score, Precision, and Recall. Some plot that are also included are Epoch vs Accuracy plots and Noise sigma vs Accuracy for the added Gaussian noise analytics performed.

The F1-score, precision, and recall are all important metrics to consider when evaluating the performance of a neural network. The F1-score is a measure of the balance between precision and recall, and is calculated by taking the harmonic mean of the two. Precision is a measure of the accuracy of the classifier, and is calculated as the number of true positives divided by the total number of samples predicted as positive. Recall, on the other hand, is a measure of the completeness of the classifier, and is calculated as the number of true positives divided by the total number of samples that are actually positive. Together, these metrics provide a comprehensive view of the performance of a neural network, and are critical in determining its effectiveness at solving a given problem. This is included in the output for analyzing the model with Gaussian noise and without Gaussian noise. Attached below is an example of this output:

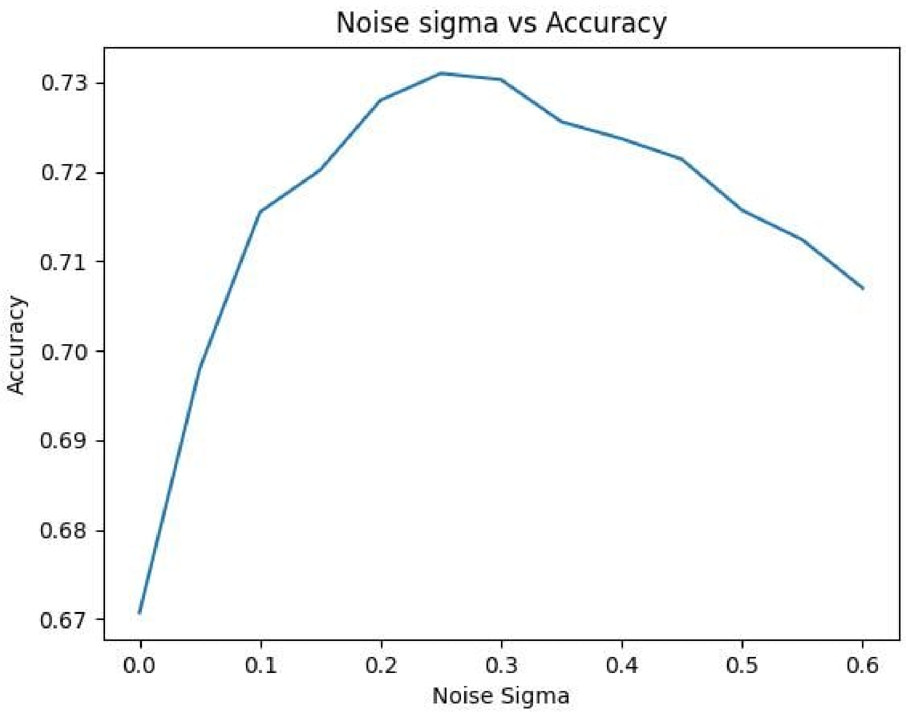


```
Accuracy: 0.72

Micro Precision: 0.72
Micro Recall: 0.72
Micro F1-score: 0.72

Macro Precision: 0.75
Macro Recall: 0.71
Macro F1-score: 0.69

Weighted Precision: 0.75
Weighted Recall: 0.72
Weighted F1-score: 0.70

Classification Report

              precision    recall  f1-score   support

           0       0.61      0.98      0.75       980
           1       0.60      0.97      0.74      1135
           2       0.88      0.67      0.76      1032
           3       0.74      0.72      0.73      1010
           4       0.78      0.77      0.78       982
           5       0.93      0.21      0.35       892
           6       0.81      0.87      0.84       958
           7       0.81      0.85      0.83      1028
           8       0.67      0.36      0.47       974
           9       0.69      0.70      0.70      1009

    accuracy                           0.72     10000
   macro avg       0.75      0.71      0.69     10000
weighted avg       0.75      0.72      0.70     10000
```

Accuracy is a measure of how well a neural network is able to correctly classify a given input, while epoch refers to a single iteration through the entire data-set during training. As the network trains, its accuracy on the training data is likely to improve with each epoch. However, it is also possible for the network to over-fit to the training data, resulting in decreased accuracy on new, unseen data. Therefore, it is important to monitor the accuracy of the network over time, both on the training data and on a separate validation data-set, in order to ensure that it is learning effectively and generalizing well to new data. Included below is an example of this output.



Adding Gaussian noise to a neural network can be a useful regularization technique, which can help to improve the network's performance on unseen data and reduce over-fitting. Gaussian noise is a type of random noise with a normal or Gaussian distribution, which can be added to the input or output of a neural network layer. This added noise forces the network to learn a more robust and generalized representation of the data, rather than over-fitting to the specific training examples.



7 Project Guide

For repository containing the project, see Here or go to the next url: `https://github.com/ChrisACas/MLProject-Neural_Tigers`.

Simply follow the .readme to import the dependencies necessary and run the script with the command 'python test.py'. If the script appears frozen, try closing the plots so that the program may continue.

In order to see the 'Noise Sigma vs Accuracy Experiment' run 'python SanjeevsFinalTest.py'