Image Classification Using Deep Neural Network

W.M. Nisansala Wickramasinghe
School of Mathematical Sciences,
The University of Texas at Dallas
UTD NetID: wmw190000
Wickramasinghe.Wickramasinghe@UTDallas.edu

Mananage Sanjaya Kumara
School of Mathematical Sciences,
The University of Texas at Dallas
UTD NetID: mmk190004
mananagesanjaya.kumara@utdallas.edu

***Abstract*—This article focuses on building a Deep Neural Network(DNN) from scratch using Python programming language for the purpose of image classification. As the first step, we chose an appropriate activation function which is the sigmoid function because of its ability to transform any real number to number between 0 and 1. Then we implemented forward propagation using two hidden layers, and calculated loss using two different loss functions, i.e. categorical cross entropy and mean squared error. Finally, we tried to minimize the loss by updating weights and biases while back-propagation. We also built a DNN model using Keras and TensorFlow libraries in python. We used the handwritten digit recognition data set (MNIST) to calculate the performances of the model. We carried out hyperparameter tuning to find the optimal parameters that influence image classification. The model with two hidden layers (256, 128 units) and with a learning rate of 0.01 and 100 epochs seems to be the best model with 97% accuracy.**

***Index Terms*—Deep Neural Network, hyperparameter, sigmoid, categorical cross entropy**

I. INTRODUCTION

In the context of machine learning, image classification refers to the task of identifying objects of interest and classifying them using deep learning and machine learning algorithms. Nowadays, image recognition and classification technologies are widely used in different fields like healthcare, marketing, transportation, and e-commerce and it has revolutionized the world with their applications like facial recognition and driverless cars.

Neural networks, also known as Artificial Neural Networks (ANNs) represent a huge breakthrough in image classification. A neural network is a series of algorithms composed of artificial neurons or nodes which can be used to learn and model the relationships between input and output data that are nonlinear and complex. The name and structure of a neural network are inspired by the human brain and it attempts to mimic the structure and function of the human brain. A Deep Neural Network(DNN) is a stacked neural network with multiple hidden layers between the input and output layers. The use of many more hidden layers makes it a 'deep' network and help to learn more complex patterns. Although there are many advantages like adaptive nature, and effective visual analysis sometimes they are highly dependent on the data made available to them.

In literature, there are a lot of studies conducted on image classification, and various statistical and machine learning methods are utilized to model and classify images. For example, Guo, Dong, Li, and Gao have used a convolutional neural network for image classification. In this paper, they have used MINITS dataset to check the performances and they analyzed different methods of learning rates and different optimization algorithms to find the optimal parameters that influence image classification [1]. Moreover, Goh, Chang, and Cheng showed that SVM-based binary classifiers can be effectively combined to perform the multi-class image classification problem [2]. Additionally, other machine learning techniques like K-nearest neighbor(KNN) and Decision trees were successfully used for image classification in literature [3], [4].

In this project, we built a deep neural network for image classification from scratch and tried to compare it with a model built using python libraries Keras and TensorFlow. We tried to use a handwritten digit recognition dataset (MINIST) to compare the performances of two models. The MNIST dataset is an acronym that stands for the Modified National Institute of Standards and Technology data set. This dataset contains a training set of 60,000 28×28 grayscale images of 10 handwritten digits (from 0 to 9), along with a test set of 10,000 images. We built a 2 layer-deep neural network model to classify a given image of a handwritten ten-digit into one of 10 classes representing integer values from 0 to 9, inclusively. This is a multiclass classification problem with 10 output classes.

The whole project can be divided into three phases:

- **Preprocessing:** The data, need to be normalized into a suitable range to avoid large gradient values that could make training difficult. To build the neural network data must be in the correct shape and format. Therefore, we converted digits formatted 0 to 9 labels to a suitable format which consists of vectors with 1's and 0's.
- **Modeling**: In this step, we built a deep neural network from scratch using python and also from python libraries Keras and TensorFlow.
- **Parameter tuning:** Finally, we performed hyperparameter tuning to find the optimal parameters that influence image classification.

II. DEEP NEURAL NETWORK

A deep neural network is a collection of different layers. The first layer is the input layer and it picks up the input

signals and passes them to the next layer. The next layers are hidden layers. They perform nonlinear transformations of the inputs entered into the network and pass the result to the final layer. The last layer which delivers the final result is called the output layer.

A model with an input layer, k number of hidden layers, and an output layer can be written as

$$X \to \boldsymbol{h}^{(1)}(X) \to \cdots \to \boldsymbol{h}^{(k)}\{\boldsymbol{h}^{(k-1)}\} \to a\{\boldsymbol{h}^{(k)}\} \to \hat{\boldsymbol{Y}}$$

implying the target f is assumed have the form

$$f(X;\theta) = a\{\boldsymbol{h}^{k}\{\boldsymbol{h}^{(k-1)}\{\ldots\{\boldsymbol{h}^{(1)}(X)\}\ldots\}\}\}$$

with specified activation function a. Parameter θ consists only of biases and weights. Figure 1 represents a simple neural network with one hidden layer.

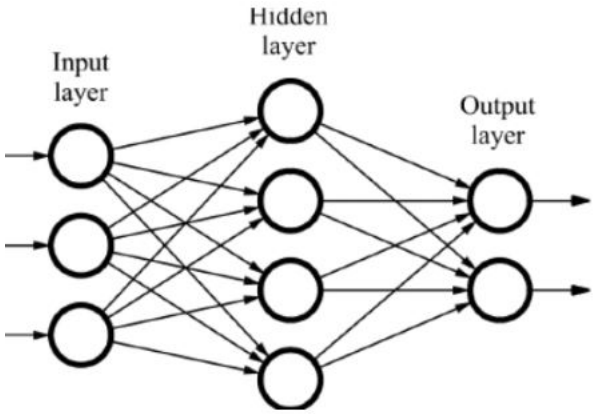

Fig. 1. A simple neural network with one hidden layer

We built our deep neural network using the following steps:

- Choosing activation function
- Forward propagation
- Choosing loss (cost) function
- Backward propagation
- Parameter tuning

A. Activation function

One of the first steps in building a neural network is finding the appropriate activation function. The activation function takes the sum of weighted input as an argument and returns the output of the neuron.

The sigmoid function's ability to transform any real number to one between 0 and 1 is useful in this case as it allows the network to learn non-linear relationships between the data. The sigmoid function can be represented as

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

derivative of sigmoid would be

$$\frac{\partial \sigma}{\partial x} = \sigma(x)(1-\sigma(x))$$

B. Forward Propagation

Forward propagation means we accept an input $\boldsymbol{X}$ and produce an output $\boldsymbol{y}$ and information flows forward through the network. The input $\boldsymbol{X}$ provides the initial information that then propagates up to the hidden units at each layer and finally produces $\hat{\boldsymbol{y}}$.

The forward propagation equations for a neural network with one hidden layer can be formulated as:

$$\begin{aligned} \boldsymbol{Z}_1 &= \boldsymbol{W}_1^T \boldsymbol{X} + \boldsymbol{b}_1 \\ \boldsymbol{A}_1 &= \sigma(\boldsymbol{Z}_1) \\ \boldsymbol{Z}_{out} &= \boldsymbol{W}_{out}^T \boldsymbol{A}_1 + \boldsymbol{b}_{out} \\ \boldsymbol{O} &= \sigma(\boldsymbol{Z}_{out}) \end{aligned}$$

where, $\boldsymbol{X}, \boldsymbol{W}, \boldsymbol{b}, a, \boldsymbol{O}$ represents corresponding input, weights, biases, activation function, and output.

C. Calculating loss

Next, we compare the result we obtained from the forward propagation with the actual output. The task is to minimize the loss. Each of the neurons in the network is contributing some error to the final output.

Our model classifies 10 handwritten digits (from 0 to 9). Therefore, this can be framed as a multi-class classification problem.

1) Cross entropy: : Formula for calculating cross-entropy loss:

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1-y_i) \cdot log(1-p(y_i))$$

The cross-entropy loss is then calculated by using the cross-entropy formula and adding up all the losses: $c = \sum_{c=0}^{m} \text{cross entropy}(x, y)$ where m is the number of classes.

2) Mean squared error(MSE): : Formula for calculating cross-entropy loss:

$$\text{MSE} = \frac{1}{N}\sum(y-\hat{y})^2$$

where $\hat{y}$ is the predicted value.

D. Backward propagation

Now we try to minimize the loss by updating weights and biases while traveling back. This process is known as "Backward Propagation". In order to minimize the error, we can use a common algorithm known as "Gradient Descent". In gradient descent, we update weights using the rule

$$\boldsymbol{W} \to \boldsymbol{W} - \frac{\partial \boldsymbol{E}}{\partial \boldsymbol{W}}$$

Here we consider two cases,

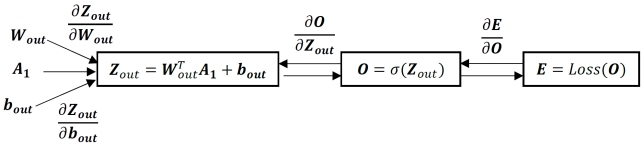

Fig. 2. updating weights and bias at output layer

1) updating weights and bias at output layer: Our goal is to update the weights ($\boldsymbol{W}_{out}$) and bias ($\boldsymbol{b}_{out}$) of the output layer. Let's take sigmoid as our activation and MSE as our loss function. Refer figure 2. The values of $\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{W}_{out}}$ and $\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{b}_{out}}$ can be calculated as

$$\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{W}_{out}} = \frac{\partial \boldsymbol{Z}_{out}}{\partial \boldsymbol{W}_{out}} \times \left[\frac{\partial \boldsymbol{O}}{\partial \boldsymbol{Z}_{out}} \cdot \frac{\partial \boldsymbol{E}}{\partial \boldsymbol{O}}\right]^T$$

and

$$\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{b}_{out}} = \frac{\partial \boldsymbol{b}_{out}}{\partial \boldsymbol{W}_{out}} \times \left[\frac{\partial \boldsymbol{O}}{\partial \boldsymbol{Z}_{out}} \cdot \frac{\partial \boldsymbol{E}}{\partial \boldsymbol{O}}\right]^T$$

where,

$$\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{O}} = -(\boldsymbol{Y} - \boldsymbol{O}), \quad \frac{\partial \boldsymbol{O}}{\partial \boldsymbol{Z}_{out}} = \boldsymbol{O}(1 - \boldsymbol{O})$$
$$\frac{\partial \boldsymbol{Z}_{out}}{\partial \boldsymbol{W}_{out}} = \boldsymbol{A}_1, \quad \frac{\partial \boldsymbol{Z}_{out}}{\partial \boldsymbol{b}_{out}} = \boldsymbol{1}$$

$\times$ represent dot product and $\cdot$ represent element wise multiplication.

then we can update weights using

$$\boldsymbol{W}_{out} \to \boldsymbol{W}_{out} - \frac{\partial \boldsymbol{E}}{\partial \boldsymbol{W}_{out}}, \quad \boldsymbol{b}_{out} \to \boldsymbol{b}_{out} - \frac{\partial \boldsymbol{E}}{\partial \boldsymbol{b}_{out}}$$

2) updating weights and bias at hidden layers: Our goal is to update the weight($\boldsymbol{W}_{out}$) and bias($\boldsymbol{b}_{out}$) of the hidden layer. Refer figure 3. The values of $\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{W}_1}$ and $\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{b}_1}$ can be calculated as

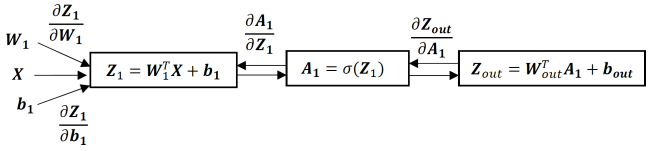

Fig. 3. updating weights and bias at output layer

$$\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{W}_1} = \frac{\partial \boldsymbol{Z}_1}{\partial \boldsymbol{W}_1} \times \left[\frac{\partial \boldsymbol{A}_1}{\partial \boldsymbol{Z}_1} \cdot \frac{\partial \boldsymbol{Z}_{out}}{\partial \boldsymbol{A}_1} \times \left[\frac{\partial \boldsymbol{O}}{\partial \boldsymbol{Z}_{out}} \cdot \frac{\partial \boldsymbol{E}}{\partial \boldsymbol{O}}\right]\right]^T$$

and

$$\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{b}_1} = \frac{\partial \boldsymbol{Z}_1}{\partial \boldsymbol{b}_1} \times \left[\frac{\partial \boldsymbol{A}_1}{\partial \boldsymbol{Z}_1} \cdot \frac{\partial \boldsymbol{Z}_{out}}{\partial \boldsymbol{A}_1} \times \left[\frac{\partial \boldsymbol{O}}{\partial \boldsymbol{Z}_{out}} \cdot \frac{\partial \boldsymbol{E}}{\partial \boldsymbol{O}}\right]\right]^T$$

where,

$$\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{O}} = -(\boldsymbol{Y} - \boldsymbol{O}), \quad \frac{\partial \boldsymbol{O}}{\partial \boldsymbol{Z}_{out}} = \boldsymbol{O}(1 - \boldsymbol{O}),$$
$$\frac{\partial \boldsymbol{Z}_{out}}{\partial \boldsymbol{A}_1} = \boldsymbol{W}_{out}, \quad \frac{\partial \boldsymbol{A}_1}{\partial \boldsymbol{Z}_1} = \boldsymbol{A}_1(1 - \boldsymbol{A}_1),$$
$$\frac{\partial \boldsymbol{Z}_1}{\partial \boldsymbol{W}_1} = \boldsymbol{X}, \quad \frac{\partial \boldsymbol{Z}_1}{\partial \boldsymbol{b}_1} = \boldsymbol{1}.$$

we can update weights using

$$\boldsymbol{W}_1 \to \boldsymbol{W}_1 - \frac{\partial \boldsymbol{E}}{\partial \boldsymbol{W}_1}, \quad \boldsymbol{b}_1 \to \boldsymbol{b}_1 - \frac{\partial \boldsymbol{E}}{\partial \boldsymbol{b}_1}$$

III. Application to MNITS data

MNIST data set is a collection of 10 grayscale handwritten digits varying from the number 0 to 9. The data consist of separate training set with 60,000 images and a test set of 10,000 images that are classified into corresponding classes or labels. We used MNIST dataset in Keras package. Figure 4 represents a random sample of 9 MNIST digits.

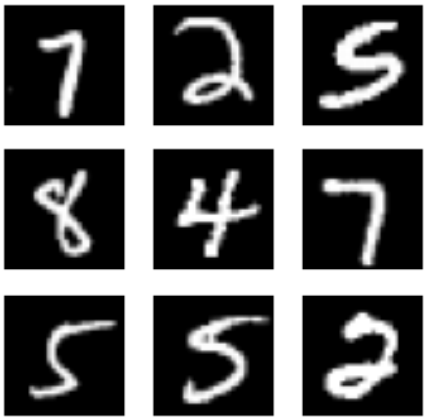

Fig. 4. 9 random MNIST digits

First, we normalized data into a suitable range to avoid large gradient values which could make training difficult. To build the neural network, data must be in the correct shape and format. Therefore, we converted digits formatted 0 to 9 labels to a suitable format which consists of vectors with 1's and 0's.

In our project, the object is to build a deep neural network to classify the images in MNIST dataset. So we developed a two-layer neural network with an output layer. We considered two model architectures. We designed the first model architecture with a single hidden layer with 128 units and an output layer. For the next model, we used two hidden layers with 256 and 128 units respectively. We chose the 'sigmoid' activation function in each hidden layer and output layer.

In a compiler, we used 'categorical cross entropy as a loss function and we used an 'adam' optimizer with two different learning rates 0.1 and 0.01. We also choose two different epoch sizes$(100, 200)$ for our network setting.

The loss and accuracy of each epoch for training and validation data were stored for each different setting. The plots of accuracy vs epochs and loss vs epochs were drawn for both training and validation data.

Fig. 5. Training Accuracy for 200 epochs

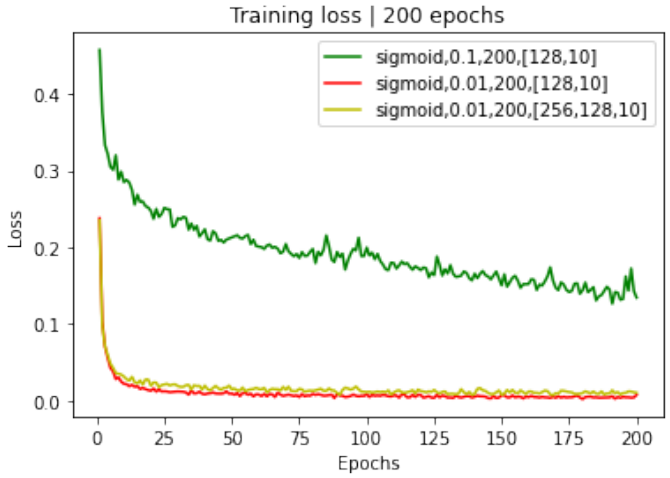

Fig. 6. Training Loss for 200 epochs

Figure 5 represents the plot of training accuracy vs epochs for three different hyper-parameters settings. The model with a single hidden layer with 128 units and with a 0.01 learning rate gives the highest training accuracy. At the same time, it gives the lowest training loss for 200 epochs. (See Fig 6).

Figures 7 & 8 represent the accuracy vs epochs and loss vs epochs for the validation data set. When it comes to validation data the model with two hidden layers (256, 128 units) and with a learning rate of 0.01 gives the best validation accuracy and validation loss.

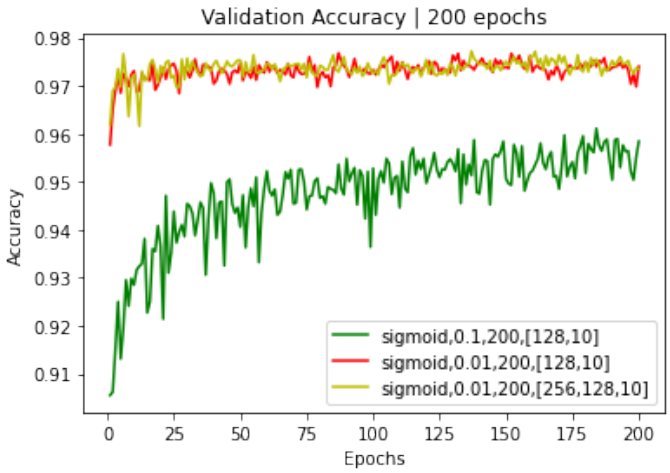

Fig. 7. Validation Accuracy for 200 epochs

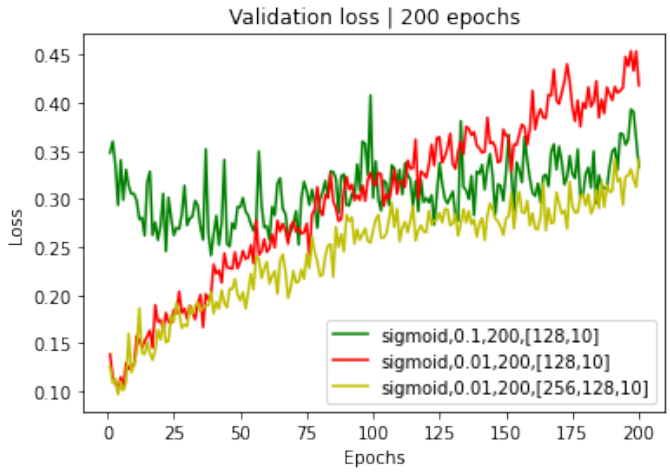

Fig. 8. Validation Loss for 200 epochs

Hyperparameters	Training Accuracy	Training Error
[Sigmoid ,0.1,[128,10],100]	96.0150%	0.19682
[Sigmoid ,0.01 ,[128,10],100]	99.9467%	0.00164
[Sigmoid ,0.01 ,[256,128,10],100]	99.7667%	0.00760
[Sigmoid ,0.1 ,[128,10],100]	97.7717%	0.10794
[Sigmoid ,0.01 ,[128,10],100]	99.9400%	0.00240
[Sigmoid ,0.01 ,[256,128,10],100]	99.7917%	0.00951

TABLE I
TRAINING ACCURACY AND TRAINING ERROR FOR DIFFERENT HYPERPARAMETERS

Hyperparameters	Training Accuracy	Training Error
[Sigmoid ,0.1,[128,10],100]	94.92%	0.33344
[Sigmoid ,0.01 ,[128,10],100]	97.06%	0.33802
[Sigmoid ,0.01 ,[256,128,10],100]	97.31%	0.25512
[Sigmoid ,0.1 ,[128,10],100]	95.84%	0.33343
[Sigmoid ,0.01 ,[128,10],100]	97.40%	0.41833
[Sigmoid ,0.01 ,[256,128,10],100]	97.34%	0.34110

TABLE II
TEST ACCURACY AND TEST ERROR FOR DIFFERENT HYPERPARAMETERS

The training accuracy and the training error for each hy-

perparameter setting are shown in table I and the results for test data are shown in table II. The highest training accuracy and the lowest training error are received by the model with a single hidden layer, 0.01 learning rate, and 100 epochs. Overall, the best test accuracy and the minimum error are achieved by the model with two hidden layers, 0.01 learning rate, and 100 epochs. So this model can be taken as the best model to classify the MNIST image data.

Discussion and Conclusion

This article focuses on building a Deep Neural Network(DNN) from scratch using Python programming language for the purpose of image classification. We also built a model using Keras and TensorFlow libraries and carried out the parameter tuning. The model we built using scratch performed well for smaller data sets. But when it comes to the larger data sets and with more hidden layers the performances are poor and this could be because it gets stuck at a sub-optimal solution. All the steps and matrix manipulations we used during the model building were discussed in the section under Deep Neural Network. For the model built using libraries, we performed hyperparameter tuning to find the optimal parameters that influence image classification. The model with two hidden layers (256, 128 units) and with a learning rate of 0.01 seems to be the best model with 97% accuracy.

References

[1] T. Guo, J. Dong, H. Li and Y. Gao, "Simple convolutional neural network on image classification," 2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA), 2017, pp. 721-724

[2] K. Goh, E. Chang, and K. Cheng, "SVM binary classifier ensembles for image classification". In Proceedings of the tenth international conference on Information and knowledge management (CIKM '01), 2001

[3] C. Yang, S. Prasher, P. Enright, C. Madramootoo, M. Burgess, P. Goel, I. Callum, "Application of decision tree technology for image classification using remote sensing data", Agricultural Systems, Volume 76, Issue 3, 2003, pp 1101-1117

[4] K. Huang, S. Li, X. Kang et al. "Spectral–Spatial Hyperspectral Image Classification Based on KNN". Sens Imaging, Volume 17, Issue 1, 2016.