Abstract

Fully connected networks are the backbone of deep learning which are used for thousands of applications. This report serves to explain the implementation behind a fully connected neural network using only Numpy (no automatic gradient). The report shows the step down approach behind training and testing fully connected neural networks on MNIST handwritten digit dataset. Derivation behind forward propagation, backward propagation and error gradient computation is also discussed in this report.

Introduction

Technology becomes a crucial part of our daily life where artificial intelligence is playing a very important role in technological advancement. Image classification is the application of such future technology that changed the way we used to see the world. It is the task of assigning an input image with one class from a fixed set of categories. Deep learning excels in classifying objects in image, since it is implemented using three or more layers of neural networks, where each layer is capable of extracting one or more features of the image. Neural networks are a set of algorithms, which are loosely based on models after the human brain, to perform a recognition.

The architecture of neural networks is arranged into layers, where each layer consists of many simple processing units called nodes. A node is just a perceptron which takes an input data, performs some computation and then passes through a node's activation function to show that up to what context signal progress proceeds through the network to perform classification. For each input provided to the perceptron neural network, a weight is also associated with it as shown in Figure 1. Weights in a network represent the strength of a particular node when making a decision. They also decide how fast the activation function will trigger.

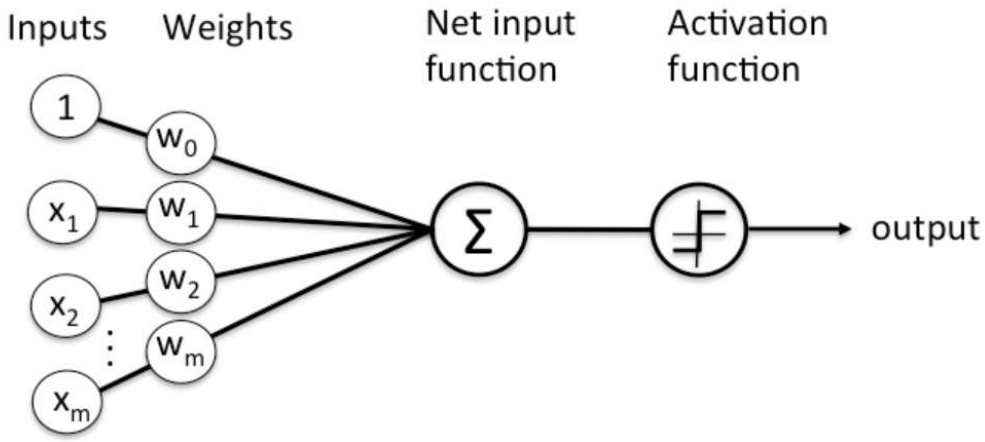

Figure 1. Single Perceptron Neural Network.[3]

Fully connected neural networks (FCN) are networks in which adjacent network layers are fully connected to one another, in other words each neuron in the network is connected to

every neuron in the adjacent layer. This network is widely used and is also incorporated in many convolution neural networks. Handwritten digits recognition is the ability of computers to recognize written digits. The task of classifying handwritten digits is not simple since different people have different writing styles, but with use of MNIST dataset and deep learning models it can be easily completed.

Related work

Digit classification systems are used to recognize the digits from different types of sources such as emails, bank cheques, papers, images, etc. Some of the widely used applications of handwritten image classification systems are implemented in banks to read amounts on bank checks to process them, recognize number plates of the vehicle from the video and images, convert numeric entries in forms filled up by hand to digital, and so on. Combined handwritten digits and character classification is also implemented in the U.S. Postal Service (USPS), for address recognition to automate the letter sorting [1].

Data description model

The MNIST (Modified National Institute of Standards and Technology database) dataset of handwritten digits from [2] were used to train and test the network built for recognising the handwritten digits. Yann Lecun, Corinna Cortes, and Christopher Burges developed this MNIST dataset for evaluating and improving machine learning models on the handwritten digit classification problem. MNIST is a subset of a larger dataset from NIST hand printed forms and characters database. MNIST dataset contains the images of handwritten digits written by the high school students and the employee of the United States Census Bureau. It is one of the largest dataset of handwritten digits, which contains a training set of 60,000 and a testing set of 10,000 gray scale images. Each gray scale image is size 28 X 28, representing the digits 0 to 9. MNIST is a widely used dataset for learning techniques and pattern recognition methods on real-world data, since the images are already pre processed. The images of the digits already have been size-normalized and centered in fixed size images.

The Dataset is downloaded from the website “http://yann.lecun.com/exdb/mnist/”, where four different compressed files are available. The files contain the training and testing images and labels. The MNIST data is downloaded and stored, by using a simple Python script. The python script downloads the four compressed files using the provided base url, and then reads the compressed files and stores this content into a python pickle module for later use. In the main program, data are loaded from the Python pickle module into four numpy arrays; “x_train”, “y_train” , “x_test ”, and “y_test”. Training images and testing images were flattened and stored in 60,000 X 784 sized “x_train” and 10,000 X 784 sized “x_test” numpy arrays respectively. For training and testing images the corresponding true labels are stored in 1 X 60,000 “y_train” and 1 X 10,000 “y_test” respectively.

Model Description

The structure of the handwritten digits classification model contains one input layer, two hidden layers, and one output layer, as shown in Figure 2. The input layer contains 784 input neurons, reason being that each image in MNIST dataset is 28 X 28, which is stretched to a length 784 vector. Output layer contains 10 neurons, since the model classifies between 10 types of digits. First and second hidden layer contain 200 and 50 neutrons respectively. The overall structure is 784-200-50-10.

The two hidden layers are followed by the ReLU (Rectified Linear Unit) activation layers in the model, to determine which neurons need to be activated and which to be deactivated. An activation function provides nonlinearity to the model which improves the performance. The ReLU is the most used activation function in deep learning models, especially in the fully connected and convolution neural networks. The ReLU function is shown in Figure 3, which is a piecewise linear function that will output the input directly if it's positive otherwise outputs zero. The output of the last layer is connected with the softmax activation function. It is a form of logistic regression that normalizes an input value into a vector of values that follows a probability distribution whose total sums up to 1. The softmax layer allows the neural network to run a multi-class function.

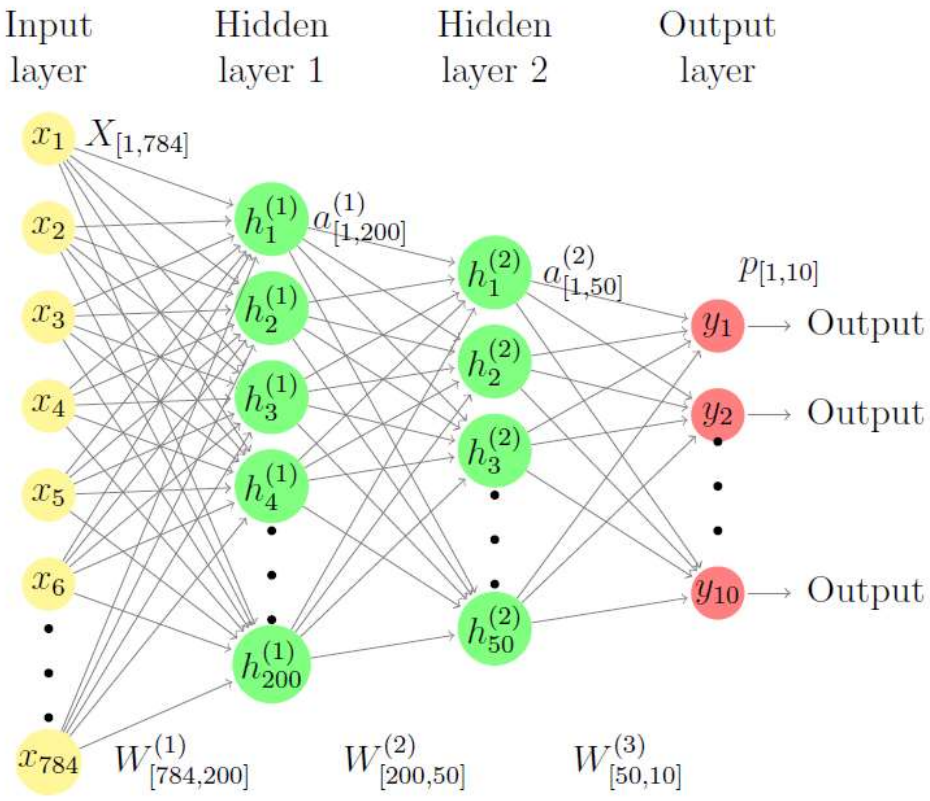

Figure 2. The fully-connected architecture for handwritten digit classification on MNIST dataset

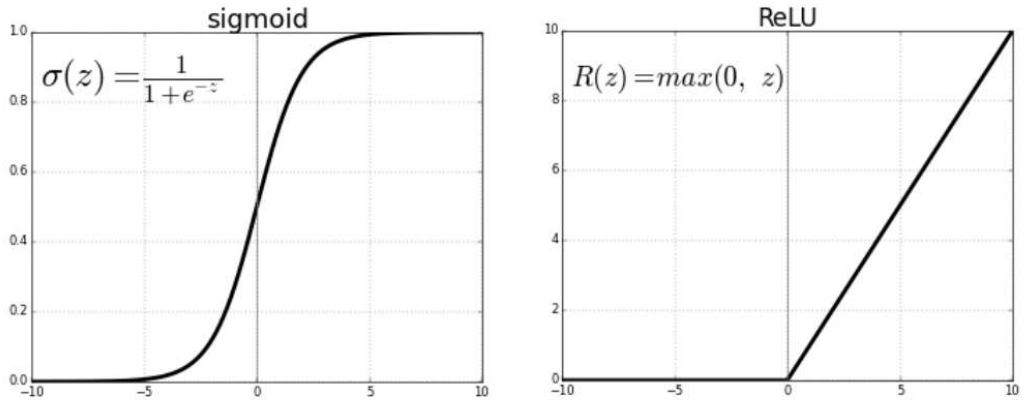

Figure 3. Sigmoid and ReLU activation functions[6]

Method Description

Handwritten digits classification model uses a fully connected neural network with two hidden layers to predict the class of the input image. Neurons are "transparent" in the input layer, which means the output of the input layer neuron is input itself. The hidden and output layer neurons are made up of accumulation of product and the activation function similar to perceptron shown in Figure 1. The perceptron works on three basic steps [3]. First, it multiplies inputs with their respective weights. Then it adds all the multiplied values and biases together, after that it passes the weighted sum to the correct activation function to produce an output. The equation below shows the matrix form representation of the function that perceptron uses to compute output.

$$output = ActivationFunction(\sum_{n} x_n w_n + bias)$$

Mechanism behind the fully connected networks is straightforward with two main processes: forward and backward propagation. Forward propagation or forward pass is a set of operations which transforms network input into output. During the testing stage the neural network relies entirely on the forward pass. Since the neural network is just multiple perceptrons put together, therefore forward pass computation is similar to perceptron. Equations for forward pass for handwritten digit classification model shown in Figure 2 is computed as,

$$z_1 = XW^{(1)} + b1$$

$$a^{(1)} = ReLU(z_1)$$

$$z_2 = a^{(1)}W^{(2)} + b2$$

$$a^{(1)} = ReLU(z_2)$$

$$z_3 = a^{(2)}W^{(3)} + b3$$

$$p = Softmax(z_3)$$

Where X is a vector representation of input feature vectors of size 1 X 784, W represents weight matrices and b are biases for each layer. $a^{(1)}$, $a^{(2)}$, and p are the functions learned on the hidden layer 1, hidden layer 2 and output respectively. z represents the accumulation of product between input and weights at each layer. p returns the probability distribution for 10 classes, from which the label for input image is predicted.

The training of the model is required for the program to correctly predict the class of input image. Training allows the model to learn and update the neuron weights and biases that are important in distinguishing between classes. During the training the network takes an input and passes it through forward propagation to produce an output. The network used predicted output and ideal output to compute the error and start adjusting the weights to produce more accurate output next time, starting from the output layer and going backward until reaching the input layer. The backpropagation is a method which calculates error gradients with respect to each network variables (weights and biases), which are used to update them correspondingly.

In order to start calculating error gradients the loss of the model is computed using the cross entropy loss function which is shown in the equation below,

$$L = -\sum_{j}^{M} y_j \ln p_j$$

Where M is the number of classes, p is the vector of the network output and y is the vector of true labels. In order to find error gradients with respect to each variable, the method of chain rule is intensively used. The following shows the complex backward propagation derivation for the handwritten digit classification model shown in Figure 2 .

Starting from the last layer, the change in loss with respect to the weights 3 is determined by,

$$\frac{\partial L}{\partial W^{(3)}} = \frac{\partial L}{\partial z_3}\frac{\partial z_3}{\partial W^{(3)}}$$

For the softmax and cross entropy loss, loss respect to accumulation of product z3 is

$$\delta z_3 = \frac{\partial L}{\partial z_3} = p - y$$

Where y is the true output. So the gradient for loss layer can be derived as,

$$\frac{\partial L}{\partial W^{(3)}} = \delta z_3 * (a^{(2)})^T$$

Preceding with layer 2:

$$\frac{\partial L}{\partial W^{(2)}} = \frac{\partial L}{\partial z_2}\frac{\partial z_2}{\partial W^{(2)}}$$

Where dL/dz_2 is

$$\delta z_2 = \frac{\partial L}{\partial z_3} = \frac{\partial L}{\partial a^{(2)}}\frac{\partial a^{(2)}}{\partial z_2}$$

$$= \frac{\partial L}{\partial z_3}\frac{\partial z_3}{\partial a^{(2)}}\frac{\partial ReLU(z_2)}{\partial z_2}$$

$$= \delta z_3 * W^{(3)} * \frac{\partial ReLU(z_2)}{\partial z_2}$$

therefore the loss respect to weights 2 is

$$\frac{\partial L}{\partial W^{(2)}} = \delta z_2 * (a^{(1)})^T$$

Preceding to Layer 1:

$$\frac{\partial L}{\partial W^{(1)}} = \frac{\partial L}{\partial z_1}\frac{\partial z_1}{\partial W^{(1)}}$$

Where dL/dz_1 is

$$\delta z_1 = \frac{\partial L}{\partial z_2} = \frac{\partial L}{\partial a^{(1)}}\frac{\partial a^{(1)}}{\partial z_1}$$

$$= \frac{\partial L}{\partial z_2}\frac{\partial z_2}{\partial a^{(1)}}\frac{\partial ReLU(z_1)}{\partial z_1}$$

$$= \delta z_2 * W^{(2)} * \frac{\partial ReLU(z_1)}{\partial z_1}$$

therefore the loss due to weight 1 is:

$$\frac{\partial L}{\partial W^{(1)}} = \delta z_1 * X$$

Similarly the biases are also calculated for each layers, following are the calculations for error due to biases:

$$\frac{\partial L}{\partial b_3} = \frac{\partial L}{\partial z_3}\frac{\partial z_3}{\partial b_3} = \frac{\partial L}{\partial b_3} = \delta z_3$$

$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial z_2}\frac{\partial z_2}{\partial b_2} = \frac{\partial L}{\partial b_2} = \delta z_2$$

$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial z_1}\frac{\partial z_1}{\partial b_1} = \frac{\partial L}{\partial b_1} = \delta z_1$$

Having the above equations, the error gradient with respect to each weight/bias can be calculated. To reduce the loss of the model, the weights and biases are updated in the opposite direction of the gradient. The general formula for which gradient can be updated is defined as follows:

$$x_{t+1} = x_t - \alpha \frac{\partial L_t}{x_t}$$

Where x is any trainiable variable either weights or B, t is the algorithm iteration index, and alpha is a learning rate. This process of training is repeated for many iterations until the error between the ideal output and the predicted output is small enough.

Experimental Procedure

The experiment of the handwritten digits classification was done entirely using the numpy, no deep learning framework was used. Code was generated using the techniques and function discussed in the method description section. In the Python script a class named “FC_NN_MNIST” is created to store all the data and the functions to train and test fully connected neural networks. At the start of the program, the variables are declared and the weights and biases are created.

The training of the handwritten digit classification model was done using the mini-batch SGD (Stochastic Gradient Descent) with batch size set as 128. The learning rate for the model was set to 0.01. Stochastic gradient descent is an iterative method for optimizing an objective function with suitable smoothness properties. During the training process the training dataset is divided into mini 128 image set sizes and iteratively passed through the forward and backward pass to update weights and biases of the model. The forward pass and backward pass is calculated using the formula derived in the method description section. Figure 4 and Figure 5 shows the forward propagation and backward propagation implemented in the python program

respectively. Testing of the program is done using the passing test data images to “Forward_Propogation” function to get the clasificacion result as output.

```python
def Forward_Propogation(self, Input):
    z1 = np.dot(Input, self.weight0)  + self.bias0
    Hidden_Layer1 = self.Relu(z1)
    z2 = np.dot(Hidden_Layer1, self.weight1) + self.bias1
    Hidden_Layer2 = self.Relu(z2)
    z3 = np.dot(Hidden_Layer2, self.weight2) + self.bias2
    Output =  self.SoftMax(z3)
    return Hidden_Layer1, Hidden_Layer2, Output
```

Figure 4. Forward Propagation implemented in python program

```python
# Input --w0--> [H1:  (z1 = Input*weight0+bias0 | Hidden_Layer1 = Relu(z1))] --w1-->
#   [H2:  (z2 = Hidden_Layer1*weight1+bias1 | Hidden_Layer2 = Relu(z2))] --w2-->
#   [Out: (z3 = Hidden_Layer2*weight2 +bias2 | Output  = SoftMax(z3))]
#Compute the gradiant on output probability:
#dL/dz3 = Derivative of Cross CrossEntropyLoss respect to softmax * Derivative of softmax respect to z3
dz3 = self.Compute_Gradiant(Output, True_Output) # [dL/d(Output)]*[d(Output)/d(z3)] = dL/d(z3)
#H2:
dHidden_layer2 = np.dot(dz3, self.weight2.T) # dL/d(Hidden_Layer2) = [dL/d(z3)]*[d(z3)/d(Hidden_Layer2)]
self.weight2 -= self.learning_rate*np.dot(Hidden_Layer2.T, dz3) #dL/dw2 = [dL/d(Output)]*[d(Output)/d(z3)]*[d(z3)/d(dw2)]
self.bias2 -= self.learning_rate *np.sum(dz3,axis=0, keepdims=True) #bias2
dz2 = dHidden_layer2*self.Relu(Hidden_Layer2,derv=True) #dL/d(z2) = [dL/d(Hidden_Layer2)]*[d(Hidden_Layer2)/d(z2)]

#H1:
dHidden_Layer1 = np.dot(dz2, self.weight1.T) # dL/d(Hidden_Layer1) = [dL/dz2]*[dz2/d(Hidden_Layer1)]
self.weight1 -= self.learning_rate*np.dot(Hidden_Layer1.T, dz2) # dL/dw1 = [dL/d(Hidden_Layer2)]*[d(Hidden_Layer2)/d(z2)]*[d(z2)/dw2]
self.bias1 -= self.learning_rate*np.sum(dz2, axis=0)#bias1
dz1 = dHidden_Layer1*self.Relu(Hidden_Layer1, derv=True) #dL/dz1 = [dL/d(Hidden_Layer1)]*[d(Hidden_Layer1)/dz1]

#Input:
self.weight0 -= self.learning_rate*np.dot(Input.T, dz1)#dL/dw0 = [dL/dz1]*[dz1/dw0]
self.bias0 -= self.learning_rate*np.sum(dz1, axis=0)# bias0
```

Figure 5. Backward Propagation implemented in python program

Results

Training Data Accuracy :

- Out of 60,000 images 59021 images classified correctly which provided the accuracy of 98.368 %

Testing Data Accuracy:

- Out of 10,000 images 9706 images classified correctly therefore the resulting accuracy of model is 97.06 %

To train the model for 10 epochs, the program took around 4.30 minutes.

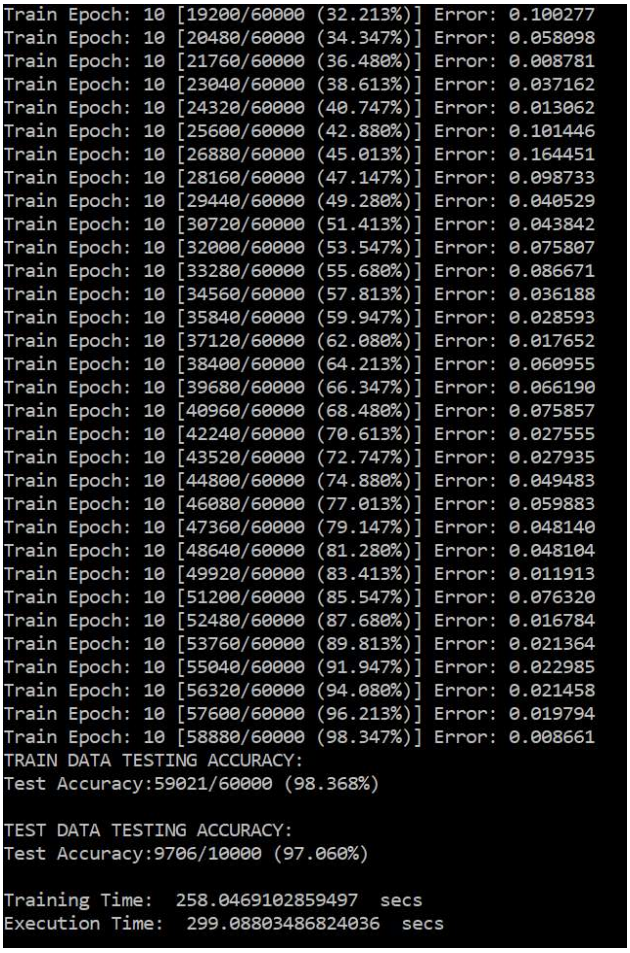

Figure 6. The results of fully connected handwritten digits classification model

Conclusion

Deep Learning is progressing very fast, due to a large number of useful libraries like TensorFlow, Pytorch, caffe, and others. Due to the availability of these libraries, implementation of neural networks does not require deep seated understanding of the area. However, knowing what's going on inside a network can be quite useful, as the complexity of tasks grows. The knowledge can help with the selection of activation functions, weights initialization, learning rates, expand the understanding of many advanced topics and many more. Handwritten digit classification using the MNIST is a great way to understand many types of deep learning architectures. Fully connected networks are one of the most important types of architectures to understand, since many of the modern and advanced neural networks are based on it.

The implementation of handwritten digit classification in Python using only numpy performed really well. The outstanding testing accuracy of 97.06 % was archived using the fully connected model. To archive the high accuracy, the SGD with the batch size of 128 was used. SGD performs the weight update of the network for every set of learning samples provided to the network instead of for the whole training data set. The process allows the gradient descent to find

the values of the parameters of a function that minimizes the cost function as much as possible. Overall the implementation of handwritten digits classification using a fully connected network was a great success, it performed outstandingly, and provided great knowledge in working behind the neural network architecture.

References

[1] “Post“Postal address recognition” *SRI International*, https://www.sri.com/hoi/postal-adress-recognition/

[2] Y. LeCun, C.Cortes, and C.Burges. "THE MNIST DATABASE of handwritten digits" http://yann.lecun.com/exdb/mnist/

[3] S.Sharma. “What the Hell is Perceptron? The Fundamentals of Neural Networks” *towards data science,* 2017, https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53

[4]A.Sakryukin. “Under The Hood of Neural Networks.” *towards data science,* 2018, https://towardsdatascience.com/under-the-hood-of-neural-networks-part-1-fully-connected-5223b7f78528

[5] C.Nicholson “A Beginner's Guide to Neural Networks and Deep Learning” *pathmind,* https://pathmind.com/wiki/neural-network

[6] S.Sharma. “Activation Functions in Neural Networks” *towards data science, 2017,* https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6