UNIVERSITY OF PERADENIYA

DEPARTMENT OF COMPUTER ENGINEERING

CO542: Neural Networks and fuzzy systems

Project Report

APPLICATION OF NEURAL NETWORKS FOR HANDWRITTEN DIGIT RECOGNITION

GROUP 18:

Pankayaraj P (E/14/237)

Sumanasekera Y.D (E/14/337)

De Silva M.D.R.A.M (E/13/058)

Kumarasiri G.M.D (E/13/200)

Navaratne N.M.I (E/13/237)

Code Repository `https://github.com/punk95/NeuralNetworkProject`

Contents

1 INTRODUCTION

The term visual pattern recognition encompasses a wide range of information processing problems of great practical significance, from the classification of handwritten characters and document classification to recognition of images of human faces and identification of fingerprints. Often these are problems which many humans solve in a seemingly effortless fashion. However, the difficulty of visual pattern recognition becomes apparent if one attempts to write a computer program to recognize digits like those illustrated by the following figure.

Figure 1: A few examples of handwritten digits from the MNIST dataset

Simple intuitions humans use to recognize patterns are quite difficult to express algorithmically. However, recent advances in deep learning have made it possible to build neural networks which can recognize objects, text, faces, and even emotions. Neural network uses training examples to automatically infer rules for recognizing objects. Furthermore, by increasing the number of training examples, the network can learn more about visual patterns, and thereby improve its accuracy.

This project was based on implementing a small subsection of object recognition—digit recognition. Using TensorFlow, an open-source Python library, hand-drawn images of the numbers 0-9 were considered and a neural network was built and trained to recognize and predict the correct label for the digit displayed.

2 METHODOLOGY

2.1 Importing the MNIST Dataset

The dataset used in this project was the MNIST dataset, which contains scanned images of handwritten digits, together with their correct classifications. Each image is 28 x 28 pixels in size and each of the 784 pixels making up the image is stored as a value between 0 and 255. This determines the greyscale of the pixel (all the images are presented in black and white only).

The MNIST dataset contains 60,000 images to be used as training data and 10,000 images to be used as test data. Thus each training input is a 784-dimensional vector with each entry in the vector representing the grey value for a single pixel in the image. The corresponding desired output is a 10-dimensional vector.

2.2 Defining the Neural Network Architecture

The architecture of the neural network was defined with three 3 layers. Namely the input, hidden and output layer. The number of neurons (units) in the input and output layer are fixed, as the input is a 28 x 28 image and the output is a 10 x 1 vector representing the class.

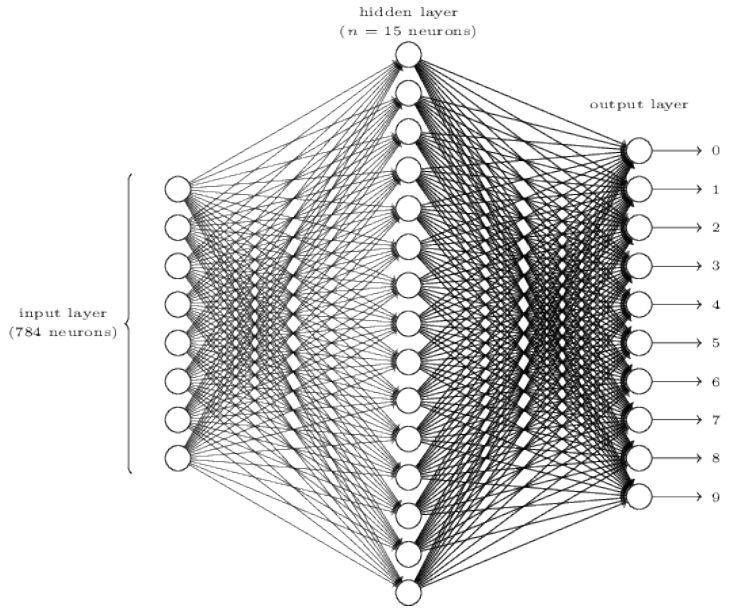


Figure 2: Architecture of a three-layer neural network

Moreover the hyper-parameters of the neural network were also defined. Unlike the parameters that will get updated during training, these values are set initially (before training) and remain constant throughout the process. The defined hyper-parameters are as follows.

- Learning Rate: The learning rate represents how much the parameters will adjust at each step of the learning process.
- Number Of Iterations: The number of training iterations refers to the number of complete passes through the training data set.
- Batch Size: The batch size refers to the number of training examples used at each step when performing gradient descent during training.
- Dropout Variable: The dropout variable represents a threshold at which some units are eliminated at random. Dropout is regularization technique to avoid over fitting (increase the validation accuracy) thus increasing the generalizing power.

At the latter part of the project Convolution neural network architecture (CNN) was tried on the dataset as well. The idea behind the CNN is that an image can be represented as a function of several edges of different angles. So in order to find the major edges that affect the image formulation a function can be formulated between them via a dense layer and obtain the results. In the feedforward architecture the hidden layers provides an easier general representation of classification. Here, due to the definition of filters the representations presented are the edges itself. In the face of higher dimension of input vectors, a dense connection may result in millions of connections while the usage of convolution filters actually reduce the number of connections while providing meaningful interpretation of the data. Apart from the convolution kernels, the pooling layers are also used. While they hold no parameters to be learned, they can be used to highlight or average a set of values used in the convoluted results.

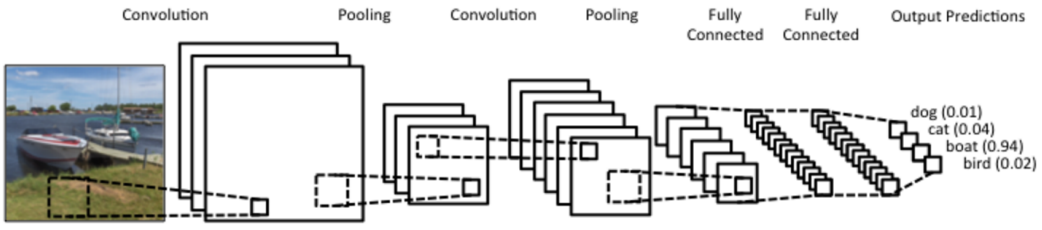


Figure 3: Architecture of a Convolution neural network

2.3 Building the Model

In order to build the model, the network was set up as a computational graph for TensorFlow to execute. The core concept of TensorFlow is the tensor, a data structure representing any multidimensional array. A data graph flow essentially maps the flow of information via the interchange between a tensor and a node. Once the graph is complete, the model is executed and the output is computed.

The parameters updated by the network during the training process are the weight and bias values. These values are essentially where the network does its learning, as they are used in the activation functions of the neurons, representing the strength of the connections between units. As the initial values of these parameters have a significant impact on the final accuracy of the model, their values were initialized (as opposed to assigning empty placeholders). Random values from a truncated normal distribution were chosen for the weights. A small constant value was initially assigned for the bias which is also learned as time progress so as to ensure that the tensors activate in the initial stages and therefore contribute to the propagation. The weights and bias tensors were stored in dictionary objects for ease of access.

The final step in building the graph involved defining the loss function that was to be optimized. Here, cross-entropy was incorporated as it is a popular choice of loss function in Tensor-Flow programs. The cross-entropy function quantifies the difference between two probability distributions (the predictions and the labels). A perfect classification would result in a cross-entropy of 0, with the loss completely minimized. Cross entropy also removes the dependency derivative of the output layer neuron off the derivative of the error. Since output layer's single is going to be a sigmoid function if it stagnates then the vanishing gradient descent problem happens early and makes most of the network ineffective. Thus this removal of dependency helps us increases the accuracy a lot.

Moreover gradient descent was chosen as the optimization algorithm which was used to minimize the loss function. Gradient descent optimization is a common method for finding the local minimum of a function by taking iterative steps along the gradient in a negative direction. Since gradient descent's parameters are static and minimalistic, further algorithms such as Momentum and Adam gradient descent were also used.

2.4 Training the Model

Training the neural network model consisted of feeding the training data to the model and optimizing the loss function. Every time the network iterates through a batch of more training images, it updates the parameters to reduce the loss in order to more accurately predict the digits shown.

As mentioned, the essence of the training process is to optimize the loss function. Thus the aim at this stage was to minimize the difference between the predicted labels of the images, and the true labels of the images. The process involved four steps which were repeated for a set number of iterations.

- Propagate values forward through the network
- Compute the loss
- Propagate values backward through the network
- Update the parameters

At each training step, the parameters were adjusted slightly in an attempt to reduce the loss for the next step. As the learning progressed, a reduction in loss was observed, and eventually the training was stopped and the subsequent network was used as the model for testing new data.

2.5 Evaluating the Model

Once training was completed, an evaluation of the model was carried out in order to determine its accuracy. For this, a comparison was carried out to determine how the model performs on the MNIST test dataset.

The accuracy on the test dataset was lower than the accuracy on the training dataset as expected. This gap between training accuracy and test accuracy is due to over-fitting. Data visualization was done using tensorboard.

When evaluated with optimal hyper-parameters, it was observed that the accuracy gradually increases to a stagnating value while the loss gradually decreases as expected. This behaviour is illustrated in the following figures.

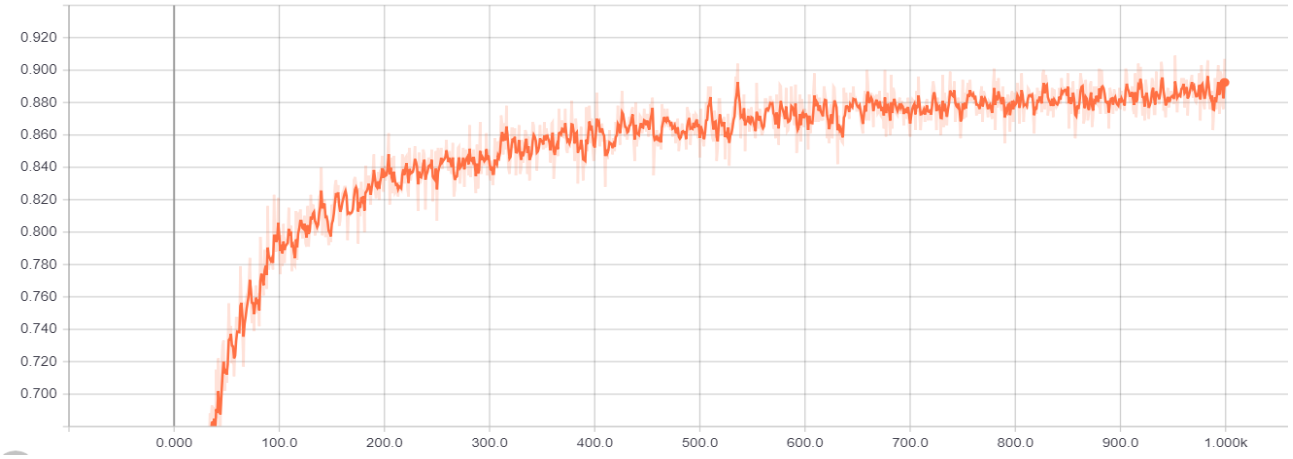


Figure 4: Variation of accuracy

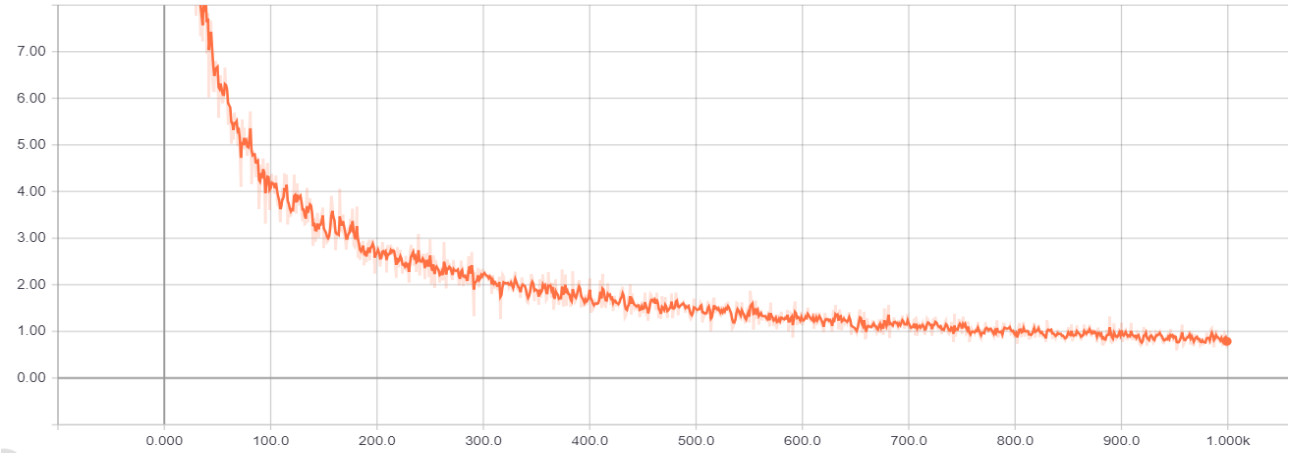


Figure 5: Variation of loss

3 RESULTS AND DISCUSSION

In order to improve the accuracy of the model and to further learn about the impact of tuning hyper-parameters, the effect of changing the learning rate, the dropout threshold, the batch size, and the number of iterations were analysed.

3.1 Error versus Batch Size

The general approach to determine the error is to separately calculate the error for each example in the training data and then average them. However such an approach is computationally expensive and results in a significantly longer time to train models on large datasets. In contrast stochastic gradient descent estimates the expected error for a small sample of randomly chosen training inputs as the actual error. Provided that the sample size is large enough, we expect that the average error value of the sample to be roughly equal to the average over all the errors of the training inputs. To observe this process, an experiment was conducted by varying the batch size and stopping the model after a certain number of iterations (while keeping the other hyper-parameters constant) and sampling the loss 50 times. Then the variance between those errors was computed and plotted against the batch size. The graphical results were as expected.

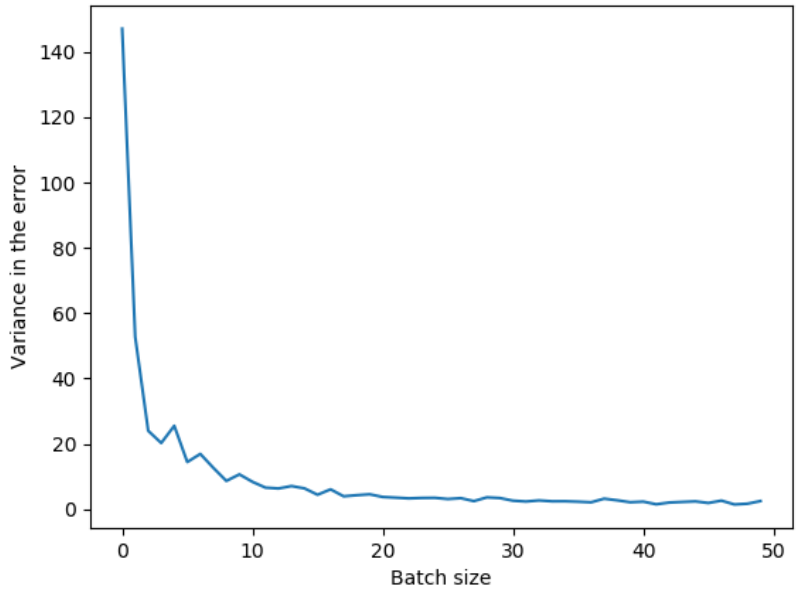


Figure 6: Variation of error with batch size

3.2 Accuracy versus Hyper-parameters of the Feedforward Network

3.2.1 Batch Size

As explained in the previous section, the batch size increases the actual accuracy of the error, thus changes made in the weights become more accurate. As expected, the accuracy showed a gradual increase with increasing batch size. Other parameters were kept constant and near optimal in order to get a clear view of this relationship between the two values.

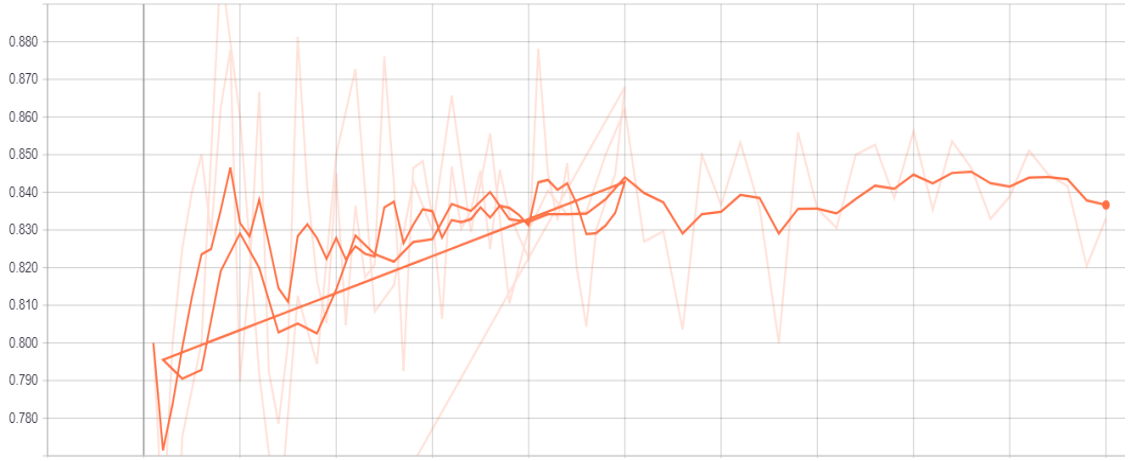


Figure 7: Variation of accuracy with batch size

3.2.2 Learning Rate

Learning rate is a parameter that normally holds a parabolic relationship with accuracy. Thus, when it is too low, it fails to reach the near minima of the cost function with a particular number of iterations (i.e. the model will converge too slowly). Similarly when it is too large, it may never reach the minima (i.e. the gradient descent can overshoot the minima and thus fail to converge). Even though the optimal learning rate is expected to be small since the relu function is used as activation (thus dying out of activation is prevented) in the presence of a limited number of iterations the optimal learning rate is expected to be a little higher

This relationship can be observed upon changing the learning rate while the other parameters are kept near optimal. Here the x axis denotes *1/learning rate* as it is

easier to visualize using a graph. Thus lower values of x corresponds to a higher learning rate while the higher values of x corresponds to a lower learning rate.

Figure 8: Variation of accuracy with learning rate

3.2.3 Optimization Algorithm

Three gradient descent algorithms were experimented with in this project. Gradient descent algorithms are optimization algorithms which based on a convex function, tweaks it's parameters iteratively to minimize a given function to its local minimum.

Stochastic gradient descent (SGD) is similar to the vanilla gradient descent where instead of having to consider the expectation over the whole dataset, it considers the expectation over only a small number of mini-batch of data points.

However gradient descent has trouble navigating ravines, that is, areas where the surface curves much more steeply in one dimension than in another, which are common around local optima. In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum. **Momentum** is a method that helps accelerate SGD in the relevant direction and dampens oscillations. It does this by adding a fraction γ of the update vector of the past time step to the current update vector.

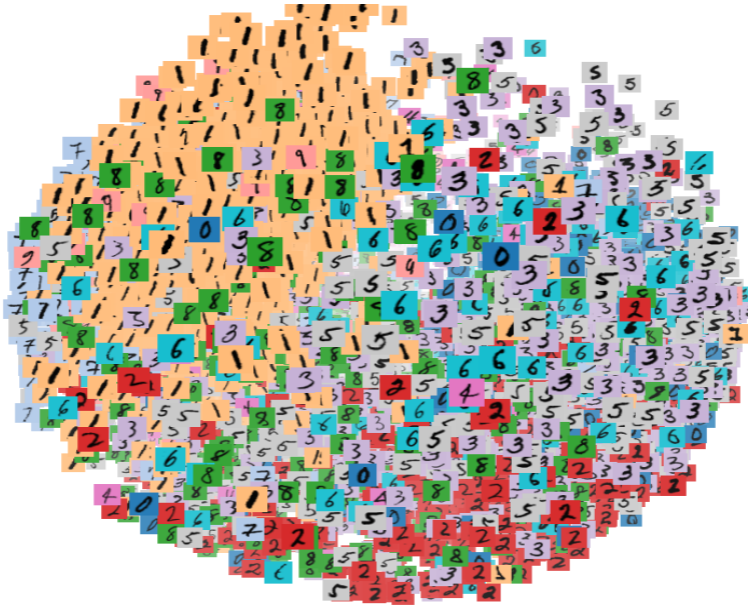
Adam gradient descent is an adaptive learning rate algorithm. Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface.

Optimization Algorithm	Accuracy
Stochastic gradient descent (learning rate = 0.01)	0.7597
Momentum gradient descent (learning rate = 0.01, momentum = 0.5)	0.8242
Adam gradient descent (learning rate = 0.01)	0.926

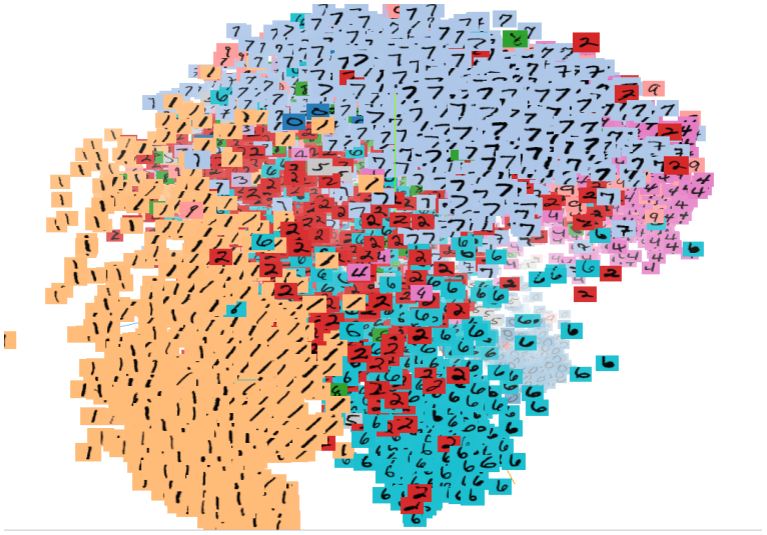
3.2.4 Number of Hidden Layers

The effect of using three hidden layers was determined at this stage. The dimension reduced clusters of each layer's output were plotted to observe how easy it is to visualize the output of each hidden layer. The clusters obtained were as follows.

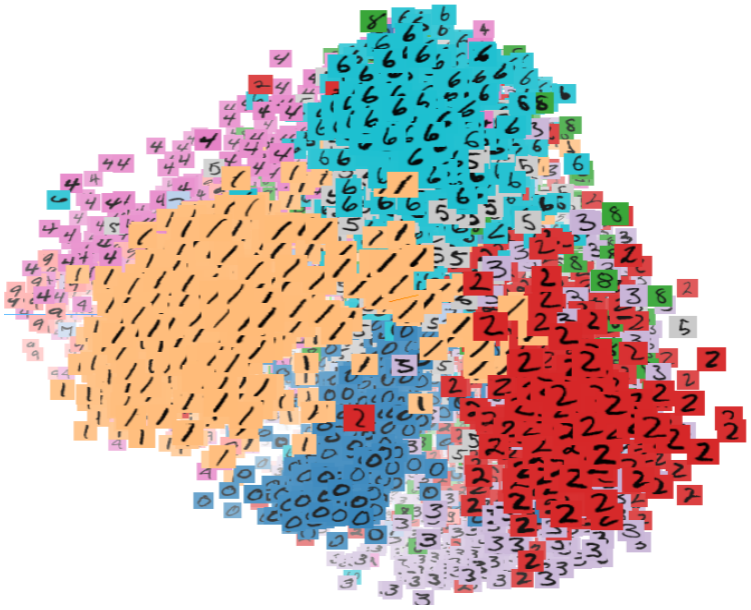
Layer 1's output



Layer 2's output



Layer 3's output



As illustrated by the figures above, with each proceeding layer, the clusters become more easily classifiable. This signifies the idea of representation learning. That is, as we increase the number of layers, each layer learns a new representation or an embedding from the output of the previous layer, thus that representation may be easily classifiable than its predecessor. Therefore, the idea of how the neural network reduces its complexity as it proceeds can be observed by these diagrams.

3.3 Comparison of Architectures

3.3.1 CNN versus Feedforward Networks

In order to determine the effect of a particular type of neural network, both the Convolutional neural network (CNN) and feedforward networks were run with almost the same hyper-parameters. That is, *hidden layer = 1024*, *learning rate = 0.001*, *batch size = 700*, *no of iterations = 1000* , *activation = relu*. While we understand that the parameters may affect the performance slightly differently in both cases, the selection of the same parameter says some information about the performance of both networks. Thus, the only difference between the architecture being feedforward layer is to find a representation learning from dense connections to each pixel of the image for the final layer to process, while CNN used a 32 and 64 filter which via sparse connection were left to find several edges for the final layer to process. Max pooling was used to highlight the found edge features thus bringing down the representation for CNN in the final layer to be the same as the one for a feedforward network. In the process of image detection, this idea of using a mechanism to explicitly find edges and form a functional representation between them evidently worked better.

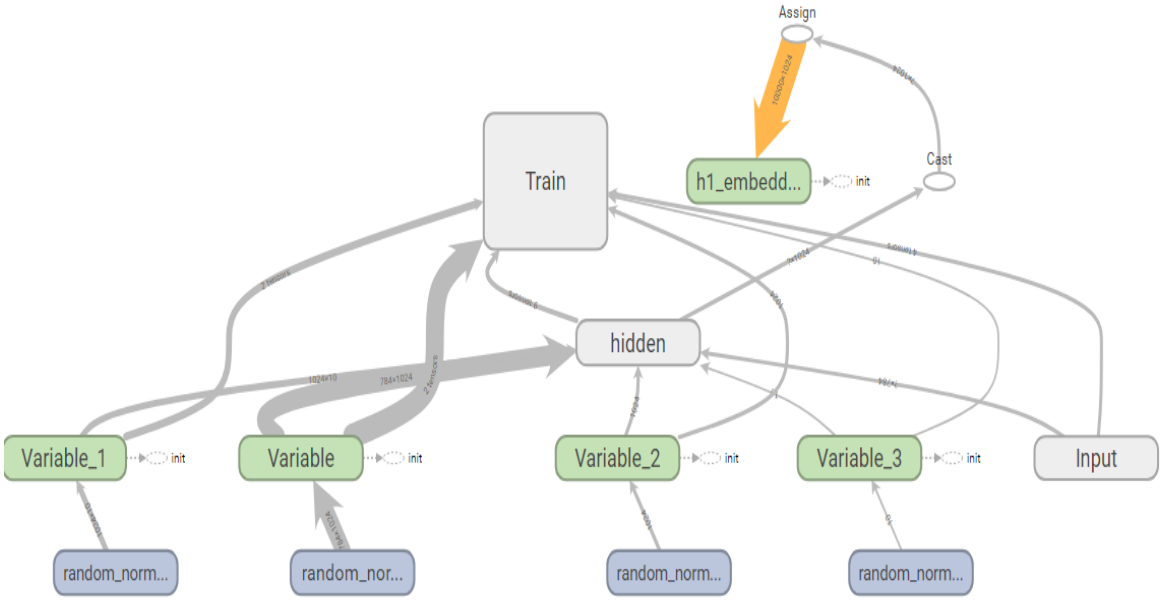


Figure 9: Feedforward network architecture

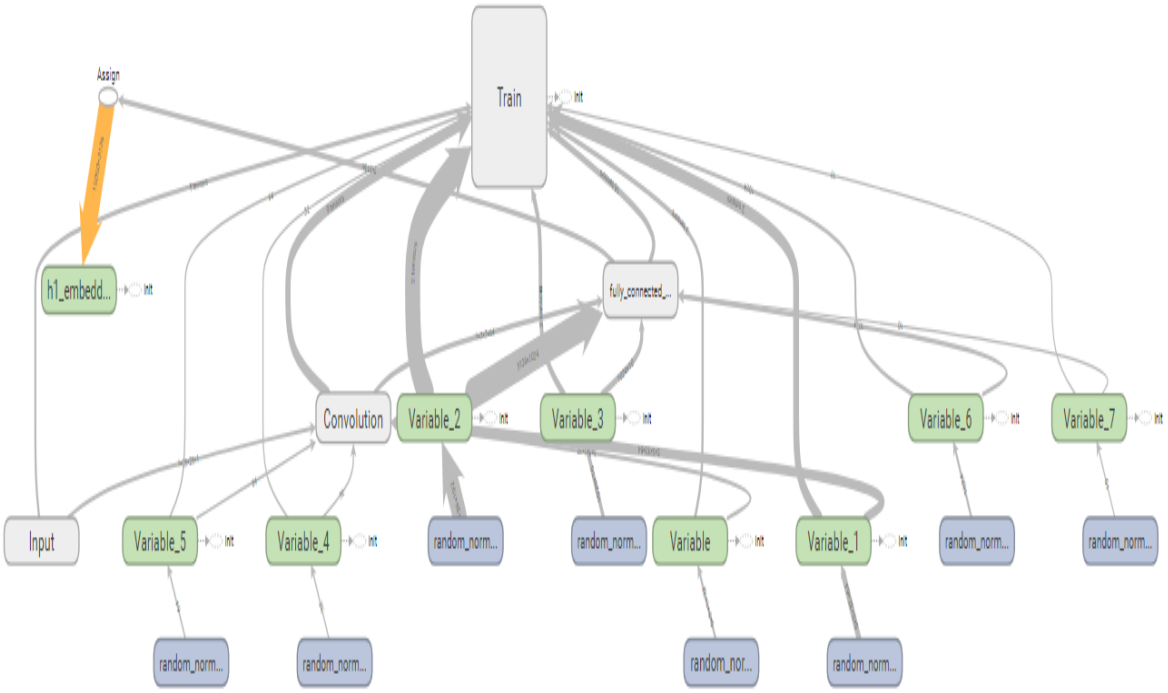


Figure 10: Convolutional neural network (CNN) architecture

The following figures illustrate the variation of accuracy and loss in FFN and CNN. Here the blue curve represents the feedforward network while the orange represents the CNN.

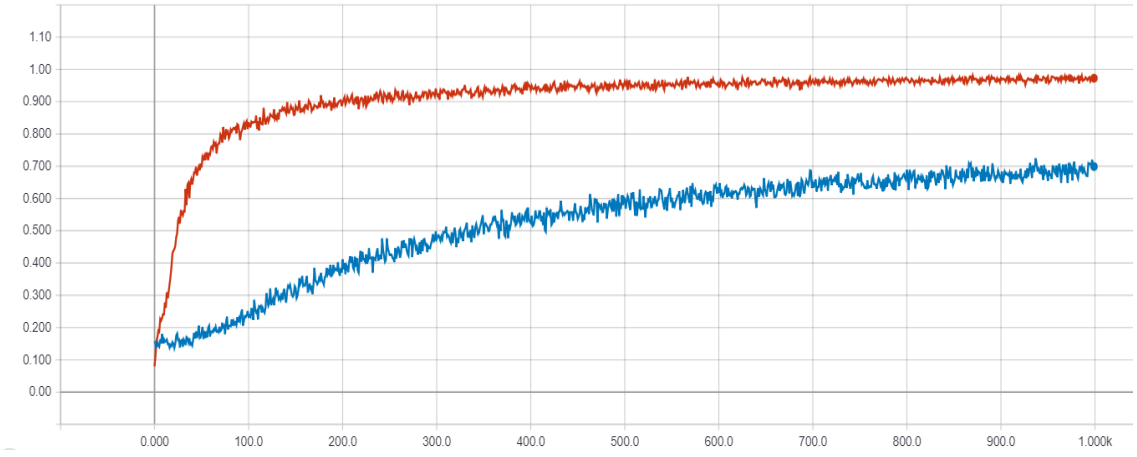


Figure 11: Variation of accuracy of the Feedforward network versus CNN

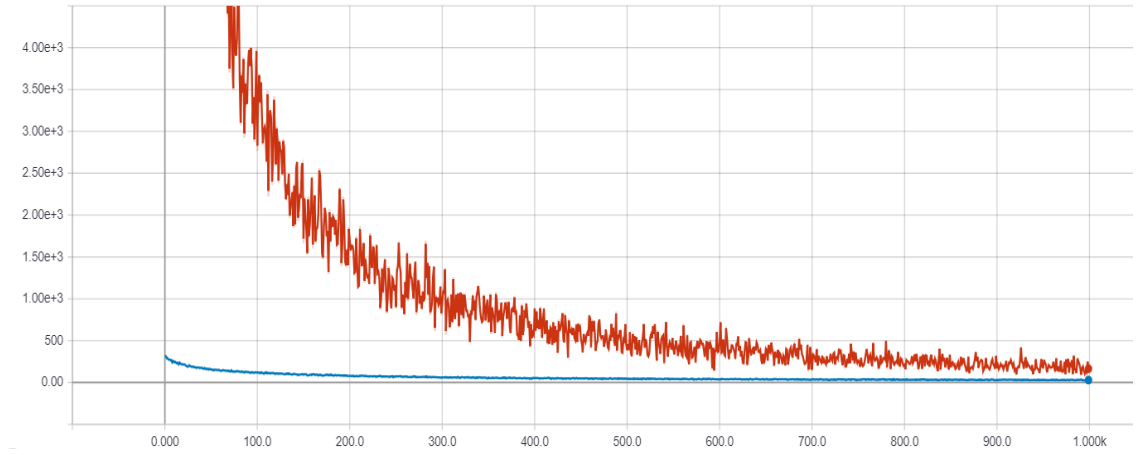


Figure 12: Variation of loss of the Feedforward network versus CNN

A Principal Component Analysis (PCA) for dimensionality reduction was carried out and the respective clusters for FFN and CNN were observed. The observations are as follows.

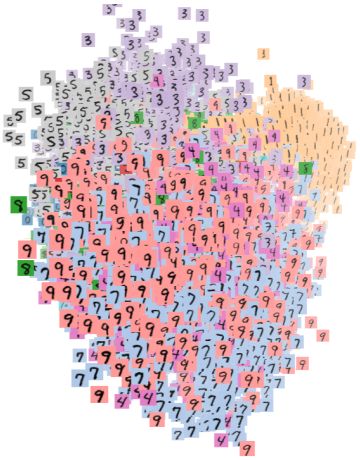


Figure 13: Clusters of the Feedforward network

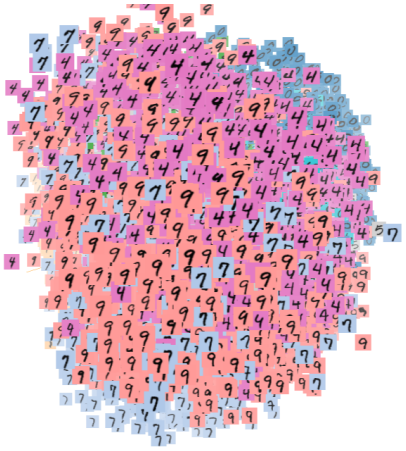


Figure 14: Clusters of the CNN

In handwritten image classification, the digits 4 and 9 look quite similar and that is where usually the classifier makes some errors in prediction. As illustrated by Figure 13, for the FFN those two numbers were interleaved in the cluster while they are more separated within the cluster for the CNN (Figure 14) . This depicts the effectiveness of CNNs on the image recognition over FFN while maintaining the parameters to be learned low.