# Neural Networks for Multi-Class Classification

**Sachin Venugopal**
University at Buffalo
sachinve@buffalo.edu

## Abstract

This project shows the implementation of a single-layer Neural Network, a multi-layer Neural Network and a Convolutional Neural Network, each used to solve a multi-class classification problem. We train the models to learn from Fashion MNIST dataset of clothing images and classify a given image into one of 10 classes.

## 1 Introduction

The task of designing and training a model on a large dataset of images, such as the Fashion MNIST dataset, and implementing it to identify the correct class to which any such given image may belong, involves complex computations and would be a difficult task to achieve using simple Logistic Regression. Such a model can be implemented efficiently using a Neural Network.

Neural Networks or Artificial Neural Networks are complex models, which try to mimic the way the human brain develops classification rules. A Neural Network consists of many different layers of neurons, with each layer receiving inputs from previous layers, and passing outputs to further layers.

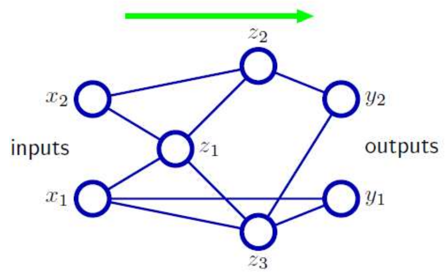


Fig 1a. Shows the example of a Neural Network

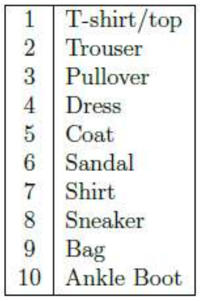
Each neuron in the Neural Network receives an input value, to which it multiplies its weight and adds the bias before applying an activation function (such as Sigmoid, ReLU, etc.) and passes the output to some or all the neurons in the successive layer. For regular neural networks, the most common layer type is the fully-connected layer, in which neurons between two adjacent layers are fully pairwise connected, but neurons within a single layer share no connections. The output layer of the Neural Network (last layer) is used to represent the class scores for the classification task.

## 2 Dataset

We use the Fashion-MNIST dataset for the purpose of training and testing our classifier. Fashion-MNIST is a dataset of Zalando's article images—consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image – 28 pixels in height and 28 pixels in width and each pixel has a single pixel value associated with where a higher value indicates a darker image.

The training and the test datasets have 785 columns each with the first column holding the class label specifying the category to which the article of clothing belongs.

Each image is associated with a label from 10 classes shown in the table below:



| | |
|---|---|
| 1 | T-shirt/top |
| 2 | Trouser |
| 3 | Pullover |
| 4 | Dress |
| 5 | Coat |
| 6 | Sandal |
| 7 | Shirt |
| 8 | Sneaker |
| 9 | Bag |
| 10 | Ankle Boot |

Fig 2a. Labels for Fashion-MNIST dataset

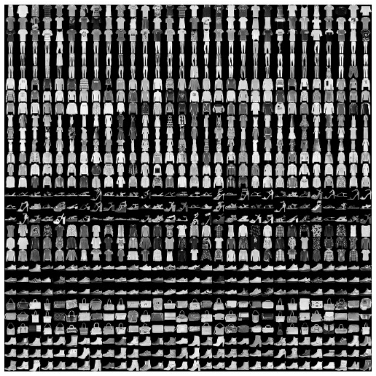


Fig.2b Data sample from Fashion-MNIST

## 3 Pre-Processing

The Fashion-MNIST dataset for the project is downloaded and partitioned into two NumPy arrays – one containing the 784 features and the other containing the class category for each of the entries in the dataset respectively. The data is also partitioned into 60000 examples for training and 10000 examples for testing. The data is already pre-processed and can be used in the Python program using the load_mnist function defined in util_mnist_reader.

The provided data needs to be converted in the following steps before being used:

1. **Normalization**: The input test and training arrays containing the 784 features needs to be normalized by dividing by 255.0 so that the pixel values remain in the range of 0 to 1.
2. **Convert the output array to a One-hot vector**: The output test and training array containing the class categories of each example is processed by one-hot encoding each row corresponding to each example. The array this becomes an array of 0s and 1s with 1s indexed by the category of the example.

## 4 Architecture

### 4.1. Neural Network

A Neural Network is a computational model inspired by the way biological neural networks in the human brain process information. It consists of a network of computational units, neurons, organized as layers, where data is processed parallelly in each layer and is passed on to the next layer without any interdependencies among the neurons of a particular layer.

A basic unit of computation in a neural network is the neuron or a node or unit. It receives input from some other nodes or from an external source and computes an output. Each input has an associated weight (**w**), and a neuron applies a function (***f***), called **activation function**, on the weighted sum of inputs. The purpose of the activation function is to introduce non-linearity into the output of a neuron.

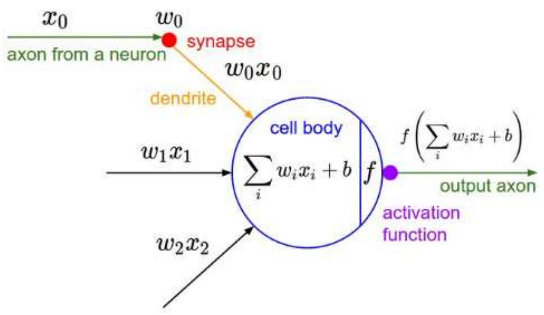


Fig 3. Mathematical model of a neuron

Every activation function operates on a single member to perform a certain fixed mathematical operation on it. There are several activation functions one may use in a neural network:

- **Sigmoid**: It takes a real-valued input and squashes it to range between 0 and 1.

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

- **tanh**: It takes a real-valued input and squashes it to the range [-1, 1].

$$\tanh(x) = 2\sigma(2x) - 1$$

- **ReLU**: ReLU stands for Rectified Linear Unit. It takes a real-valued input and thresholds it at zero (replaces negative values with zero)

$$f(x) = \max(0, x)$$

### 4.2. Feed-Forward Network Function

A feed-forward neural network is an artificial neural network where the connections between the units do not form a cycle. In such a network, information moves forward, from the input nodes, through the hidden nodes and then to the output nodes.

An example of a feed forward network is a multi-layer perceptron shown in the figure below.

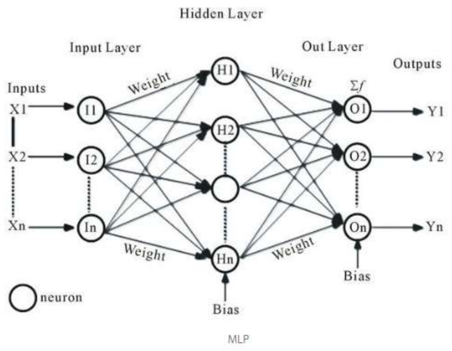


Fig 4. Multi-layer perceptron

Neural networks use basis functions that follow the form:

$$y(\mathbf{x}, \mathbf{w}) = f\left(\sum_{j=1}^{M} w_j \phi_j(\mathbf{x})\right)$$

So that each basis function is itself a nonlinear combination of the inputs, where the coefficients in the linear combination are adaptive parameters. This leads to the basic neural network model which can be described as a series of functional transformations.

First, we construct M linear combinations of the input variables $x_1$ to $x_D$ in the form,

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}$$

Where j = 1,…,M and the superscript (1) indicates the first layer of the neural network. Parameters $w_{ji}^{(1)}$ represent the weights and parameters $w_{j0}^{(1)}$ represent biases.

Each of them is then transformed using a differentiable, nonlinear activation function *h(.)*. These are again linearly combined to give output unit activations given by,

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)}$$

where k = 1,…..,K and K is the total number of outputs. This transformation corresponds to the second layer of the network. Finally, the output unit activations are transformed using an appropriate activation function to give a set of network outputs $y_k$.

We can combine these various stages to give the overall network function that, for sigmoid output unit activation functions takes the form,

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) + w_{k0}^{(2)}\right)$$

### 4.3 Back-Propagation

Back propagation is a method of fine-tuning some of the hyper-parameters of a neural network (for example, weights) based on the error rate obtained in the previous iteration(epoch). Proper tuning of the weights allows you to reduce error rates and to make the model reliable by increasing its generalization.

In Back Propagation of errors, we calculate the total error at the output nodes and propagate these errors back through the network using back-propagation to calculate the gradients. Then we use an optimization method such as Gradient Descent to adjust all weights in the network with an aim of reducing the error at the output layer.

The following are the steps in Error Backpropagation Algorithm:

1. Apply input vector xn to the network and forward propagate through the network using:

   $$a_j = \sum_i w_{ji} z_i$$

   And z = h(aj)

2. Evaluate δk for all the outputs using

   δk =yk-tk

3. Backpropagate the δs using

   δj = h'(aj)∑whj δh

   to obtain δj for each hidden unit

4. Use $\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i$ to evaluate required derivatives.

### 4.4 Softmax Function

The Softmax regression 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 output values are between the range [0, 1].
The Softmax function is given by,

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \text{ for } i = 1, \ldots, K \text{ and } \mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K$$

### 4.5 Model Implementation

#### 4.5.1 Single-Layer Neural Network

**A. Feed-forward:**
We initialize the weights to random values and biases to 0.

For the input layer of the neural network,
We have

$$Z_1 = W_1X + b_1$$

Where,
$Z_1$ is the response matrix of the input layer of the neural network,
$W_1$ is the weight matrix of arbitrary values,
$b_1$ is the bias of arbitrary value,
X is the input feature matrix

Next, the output $Z_1$ is passed to the first sigmoid activation function to give us,

$$A_1 = \sigma(Z_1)$$

We now use the output of the input layer of the neural network as the input to the hidden layer given by,

$$Z_2 = W_2A_1 + b_2$$

Where,
$Z_2$ is the response matrix of the hidden layer of the neural network,
$W_2$ is the weight matrix of arbitrary values,
$b_2$ is the bias of arbitrary value,
$A_1$ is the output of the first activation function

The output $Z_2$ is passed to the second softmax activation function to get $A_2$,

$$A_2 = \text{Softmax}(Z_2)$$

We determine the prediction of our neural network using $A_2$ by converting it into a one-hot vector by taking argmax across each row.

## B. Backward Propagation:

We use the value of $A_2$ produced by the Softmax function to calculate the error value using Cross-Entropy Loss Function given by,

$$Loss = -\frac{1}{m} * \sum y \log(A_2)$$

Where,
y is the corresponding values of the training data
m is the total number of examples

We then use the following derivatives to update the Weight matrices and bias values,

$$\nabla Z_2 = A_2 - Y$$
$$\nabla W_2 = \nabla Z_2 * A_1$$
$$\nabla b_2 = \nabla Z_2$$
$$\nabla Z_1 = \nabla Z_2 * W_2$$
$$\nabla W_1 = \nabla Z_2 * X$$
$$\nabla b_1 = \nabla Z_1$$

Weight and bias are updated as,

$$W_1 = W_1 - \alpha * \nabla W_1$$
$$b_1 = b_1 - \alpha * \nabla b_1$$
$$W_2 = W_2 - \alpha * \nabla W_2$$
$$b_2 = b_2 - \alpha * \nabla b_2$$

Where $\alpha$ is the Learning Rate

We tune the hyperparameters such as $\alpha$ (learning rate), number of epochs (iterations) and number of neurons to get optimal performance.

### 4.5.2 Multi-Layer Neural Network

We use Keras, a high-level Neural Network library, to implement a multilevel Neural Network consisting of the following hidden layers:

1) 300 neurons using a sigmoid activation function with a dropout of 0.5
2) 150 neurons using a sigmoid activation function with a dropout of 0.5
3) 64 neurons using a ReLU activation function
4) 10 neurons using Softmax activation

We use a Stochastic gradient descent as the optimizer and calculate the loss using Cross Entropy Loss Function.

### 4.5.3 Convolutional Neural Network

A convolutional neural network (CNN) is a class of deep neural networks, most commonly applied to analyzing visual imagery.

The name “convolutional neural network” indicates that the network employs a mathematical operation called convolution. Convolution is a specialized kind of linear operation. Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.

A convolutional neural network consists of an input and an output layer as well as multiple hidden layers. The hidden layers typically consist of a series of convolutional layers that ‘convolve’ with a multiplication or other dot product. The activation function is commonly a RELU layer, and is subsequently followed by additional convolutions such as pooling layers, fully connected layers and normalization layers, referred to as hidden layers because their inputs and outputs are masked by the activation function and final convolution.

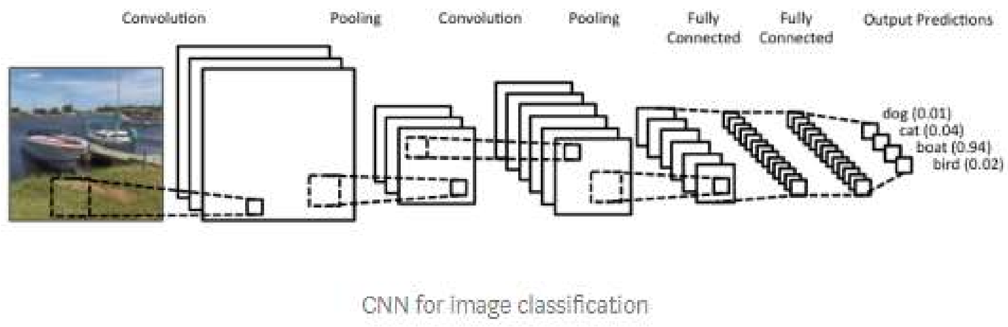


Fig.5 Example of Convolutional Neural Network Architecture

In our implementation of a CNN, we have created a model with 3 convolution layers each followed by a Maxpool layer:

- A 32-neuron convolution layer with a kernel size of 5X5 and Maxpool of pool size 2X2
- A64-neuron convolution layer with a e=kernel size of 5X5 and a Maxpool of pool size 2X2 using a sigmoid activation function
- A 64-neuron convolution layer with a e=kernel size of 3X3 and a Maxpool of pool size 2X2 Using a ReLU activation function

The classification layer includes:

- A 1024-neuron fully-connected layer that uses a ReLU activation function.
- A 10-neuron layer that uses Softmax to produce the final output of the CNN.

## 5 Results

### 5.1 Single-Layer Neural Network

Setting the hyperparameters as:

- Epochs: 150
- Learning Rate: 0.28
- Number of Neurons in the hidden layer: 342

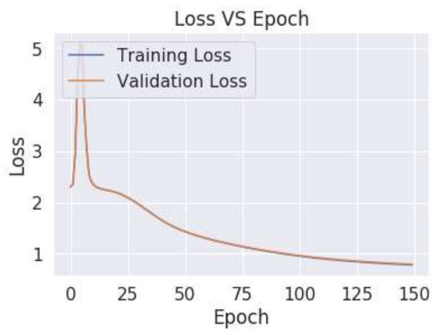


Fig6(a) Loss vs Epoch

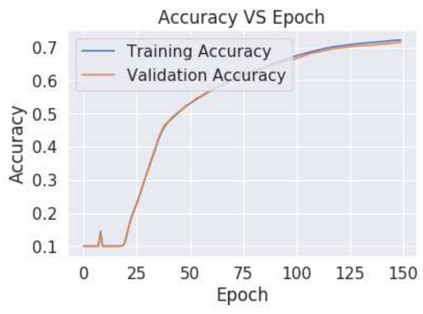


Fig6(b) Accuracy vs Epoch

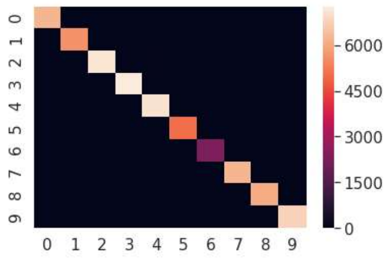
Here the training and validation data graphs overfit for hyperparameters which give optimal accuracy.

$$Accuracy = \frac{N_{correct}}{N}$$

Accuracy over testing data: 0.72195

Loss over testing data: 0.7778533517230872

Heatmap of Confusion Matrix:



### 5.2 Mutli-Layer Neural Network

Setting the hyperparameters as:

- Epochs: 60
- Learning Rate: 0.35
- No. of layers: 6

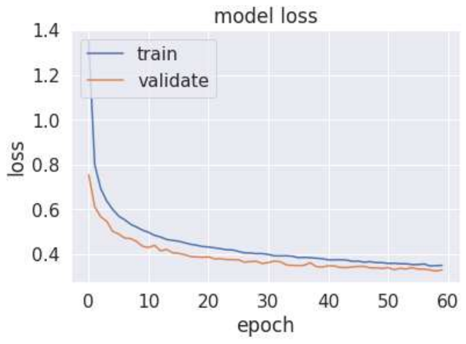


Fig 7(a). Loss VS Epoch

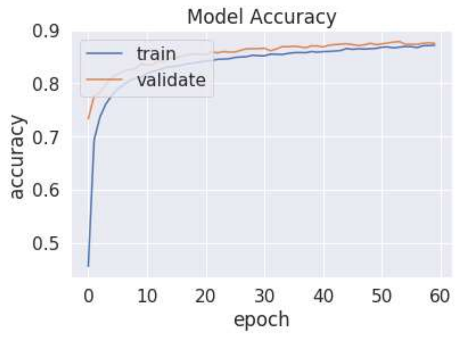
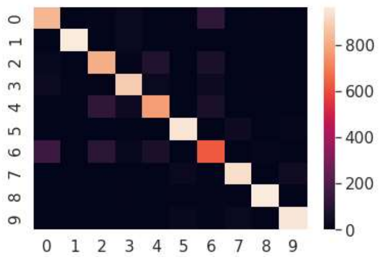


Fig 7(b). Accuracy VS Epoch

$$Accuracy = \frac{N_{correct}}{N}$$

Accuracy over testing data: `0.8692000007629395`
Loss over testing data: `0.35677031114697455`

Heatmap of Confusion Matrix:



We infer that with an increase in the number of hidden layers, performance increases significantly.

### 5.3 Convolutional Neural Network

Setting the hyperparameters as:

- Epochs: 100
- Learning Rate: 0.23
- No. of layers: 11

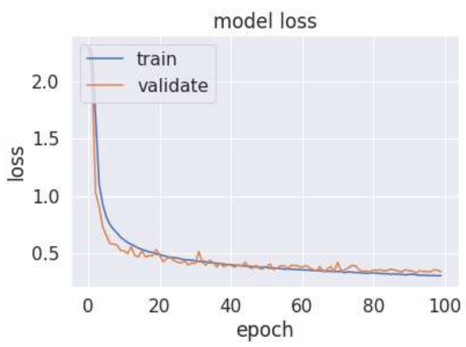


Fig 8(a). Loss VS Epoch

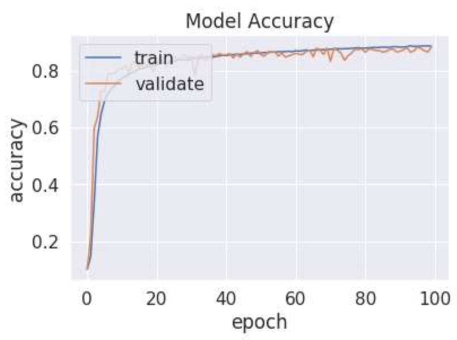


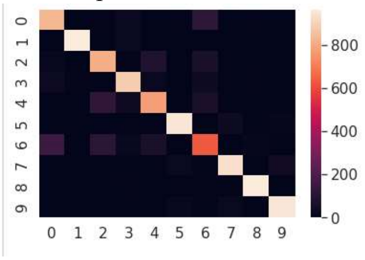
Fig 8(b). Accuracy VS Epoch

$$Accuracy = \frac{N_{correct}}{N}$$

Accuracy over testing data: `0.8791`

Loss over testing data: `0.3552879317522049`

Heatmap of Confusion Matrix:



## 6 Conclusion

We use 3 different kinds of Neural Networks in order to train, validate and test models to learn from Fashion-MNIST dataset containing 768 features. We see that that the single-layer Neural network takes a larger number of epochs and a higher learning rate to produce greater accuracy. Also, we notice that though the graphs overfit for the training and validation data, the model does not perform too well on the test data. One way of improving the performance of such a Neural Network would be to introduce regularization to improve its generalization.

With multi-layer Neural Network, as the number of layers and neurons per layer increase, the features extracted increase. This does not necessarily guarantee improved performance as the hyper parameters would still require tuning to find a combination of hyperparameters with optimal performance.

Finally, the Convolutional Neural Network performs better than the multi-layer Neural Network in extracting features from the images. The performance however does not increase with an increase in the number of convolutions and requires careful tuning of hyperparameters to achieve optimal performance.

### References

[1] https://skymind.ai/wiki/neural-network/

[2] https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks/

[3] http://cs231n.github.io/neural-networks-1/

[4] https://towardsdatascience.com/a-gentle-introduction-to-neural-networks-series-part-1-2b90b87795bc

[5] https://en.wikipedia.org/wiki/Convolutional_neural_network

[6] 5.3 Back Propagation, Sargur N. Srihari, University at Buffalo, State University of New York

[7] Logistic Regression and Machine Learning book, Christopher M. Bishop

[8] http://cs229.stanford.edu/notes/