
Classification of Fashion MNIST Data using Neural Networks

Assignment 2
2808ict Intelligent Systems
Bennett Taylor
S5095512

Introduction:

Neural networks work the same way as new born babies. They are created ignorant to the world and this ignorance is slowly revised when they are exposed to world. Neural networks see the world through large sets of labelled data. It is by training the neural networks that its ignorance gets decreased.

Training a neural network is commonly done with backpropagation. When data is feed into the network, It produces an output of what it thinks the right answer is. If the network gets the wrong answer then backpropagation is used to change its structure (by changing weights and biases) by 'rewiring' it to choose the right answer. It is another form to optimization to minimize the error (difference between the network output and the correct output) of the network.

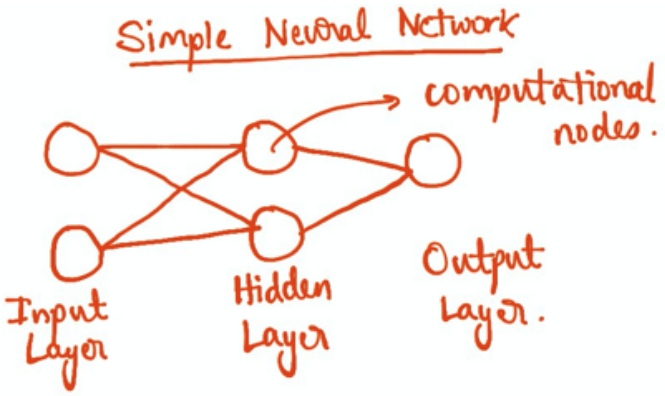

Figure 1 displays the structure of a neural network. It is made up of the input layer (which is raw input data feed into the network), the hidden layer (which attempts to breakdown the problem) and the output layer (which displays how likely each answer is). Each layer is made up of nodes/neurons which are connected to its neighbouring layers (another concepts stolen from biology). The paths that connect all these nodes together are called weights.

This project aims to classify images in the Fashion MNIST Dataset with 10 classes. The classes are as follows:

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

The dataset consists of 28x28 pixel grayscale images of 10 items of clothing. The image data is converted to a flattened csv table containing 60,000 rows and 784 columns.

Our Neural network will follow the same structure in figure 1 but contain 784 nodes in the input layer, 30 nodes in the hidden layer and 10 nodes in the output layer.

Architecture:

Since the target variables have a single value between 0 and 9 (each representing the 10 classes), hot encoding is used to convert the single value into a one hot vector with 10 values.

- Sigmoid Activation Function: This function is used to convert a real value into one that can be interpreted as a probability (between 0 and 1). It is a linear equation that incorporates the weights and bias's to give a prediction. This Activation function is applied to all the layers of the neural network except for the output layer. This is demonstrated in the Sigmoid(x) function.

$$a = \frac{1}{1 + e^{-z}}$$

- SoftMax function: This function is used in the output layer to give out values as a vector. The sum of all the values of the output vector (output nodes) add up to 1 and the class with the highest value is considered the prediction.

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

- Loss function:The loss function represents how bad the classification performance of the algorithm is. Below is the Categorical Cross Entropy loss function used but Gradient Descent for optimization of weights and bias.

$$-\sum_{c=1}^{M} y_{o,c} \log(p_{o,c})$$

- Feed Forward: the feed forward function calculates the network's current weights and bias to give output to the next layer. It moves layer by layer from the input layer to the output layer to give a prediction.

```
def feed_forward(self, X, y):
    #calculate all activations
    Activation_func = x
    Activation_list = [x] #list of all activations
    nets_of_nodes = [] #stores all the nets/in sum of each node
    for b, w in zip(self.biases, self.weights):
        net = numpy.dot(w,Activation_func) + b
        nets_of_nodes.append(net)
        Activation_func = Sigmoid(net)
        Activation_list.append(Activation_func)
```

- Backpropagation: This function is what helps the neural network grow to give better predictions. After the feed forward function reaches the output layer, backpropagation uses gradient descent to improve the weights and biases of the network. The process of feed forward and backpropagation in repeated until the network give a sound prediction.

```
def back_prop(self, X, Y):
    Activation_list, nets_of_nodes = self.feed_forword(x, y)

    delta = self.cost_derivative(Activation_list[-1], y) * Sigmoid_derivative(nets_of_nodes[-1])
    b = [numpy.zeros(bias.shape) for bias in self.biases] #layed numpy array
    w = [numpy.zeros(weight.shape) for weight in self.weights] #layed numpy array
    b[-1] = delta
    w[-1] = numpy.dot(delta, Activation_list[-2].transpose())

    for layer in xrange(2, self.num_of_layers):
        net = nets_of_nodes[-layer]
        delta = numpy.dot(self.weights[-layer+1].transpose(), delta) * Sigmoid_derivative(net)
        b[-layer] = delta
        w[-layer] = numpy.dot(delta, Activation_list[-layer-1].transpose())

    return b, w
```

- Stochastic GD Function: This is the learning part of the network which implements stochastic gradient decent. The ‘ins_outs’ parameter represents the training inputs and corresponding correct outputs. The parameter ‘epochs’ tells the function the number of epochs to train for. The function partitions the training data into mini batches (a collection of training tests) to try and reach the global minimum faster. This is done by applying a step of gradient decent for each mini batch. The update function updates the networks weights and biases in each gradient decent step.

- Network Class: This class is the main bulk of the algorithm and contains all the functions integral to the inner working of the network. Such as stochastic gradient decent, calculating node activations, backpropagation, etc. When it is first called it sets up all the layers, weights and biases included in the network. The weights and biases are initialised as randomly generated numbers (using the numpy random.random function) and put into a matrix respectively. This helps with efficiency later in the code as we can just use vector multiplication to calculate activations. The

parameter ‘sizes’ tells the network how many nodes the network contains in each respective layer. For example, ‘NN = Network([2,3,2])’ tells the network that it will have 2 nodes in the first layer, 3 in the second and 2 in the final.

Conclusion:

This project exhibits how flexible and powerful neural networks can be for the computing world. It also conveys how developing specific algorithms for specific classification tasks could become outdated. They are very well suited for the modern world with their fast response/computational times and adaptability to a wide range of disciplines.