Untitled(3)

February 26, 2019

```
In [4]: import pandas as pd
        import numpy as np
        from sklearn.model_selection import train_test_split
        import matplotlib.pyplot as plt
        from sklearn.metrics import accuracy_score
        from scipy.special import xlogy
        import pickle
```

0.1 Class Layer:

- Each layer has many features input_dim, output_dim, activation_function, its derivative, matrix W (weights) and b (biases).
- Each layer also has other parameters like A(input to layer), Z(output from layer), dW(gradient change in W), db(gradient change in b). However, these variables are temporary and removed as soon as their work is done.

0.2 Class Neural Network:

- Variables list of layers, Number of iterations, learning rate(alpha), and batch size)
- It also contain all the activation functions as welll as their derivatives.
- It contain functions for fitting and predicting data.
- There are functions for computing loss, computing error, computing accuracy.
- There are functions which are used while training data like forward_propogation, backward_propogation, update_parameters.

Forward Propogation: For each layer there is input **A[l]** to it. Each layer has **W[l]** and **b[l]** precomputed. Morever there is an activation function **g[l]** already associated to each layer. Two values Z[l] and A[l+1]

$$Z^l = W^l.A^l + b^l$$
$$A^{l+1} = g^{l+1}(Z^l)$$

Then this Al+1 is forwarded as input to next layer. then final output is computed.

Cost function: Cost is computed as Cross entropy which is simpy summation over original value and log of predicted value i.e.

$$L\{Y, A\} = \frac{-1}{m} \sum_{k=1}^{m} (Y * log(A) + ((1 - Y) * log(1 - A))$$

it derivative is given as:

$$dA\{Y, A\} = \frac{-Y}{A} + \frac{1 - Y}{1 - A}$$

Backward propogation For each layer **dAl** is given as input from it **dZl** is computed. Which in then used to compute **dWl** and **dbl**. We also compute **dAl-1**, it is passed as error for the previous layer.

$$dZ^l = dA^l * g^{l'}(Z^l)$$
$$dA^{l-1} = W^l.dZ^l$$
$$dW^l = \frac{1}{m} * dZ^l.A^l$$
$$db^l = \frac{1}{m} * \sum_{k=1}^{m} dZ^l$$
$$W^l = W^l - \alpha * dW^l$$
$$b^l = b^l - \alpha * db^l$$

In [20]:
```
class layer:
    def __init__(self, output_dim, input_dim, activation_function, derivative):
        self.output_dim = output_dim
        self.input_dim = input_dim
        self.activation_function = activation_function
        self.activation_function_derivative = derivative
        self.W = np.random.randn(output_dim, input_dim)*np.sqrt(2/input_dim)
        self.b = np.zeros((output_dim, 1))

    def print_layer_detail(self):
        print(
            """Input_dim = {0}, output_dim = {1}, activation_function = {2}, W.shape =
            Z.shape = {4}, b.shape = {5}""".format(
                self.input_dim, self.output_dim,
                self.activation_function.__name__, self.W.shape, self.Z.shape,
                self.b.shape))


class NeuralNetwork:
    def __init__(self, iterations=100, alpha=0.01, batch_size=50):
        self.layers = []
        self.iterations = iterations
        self.alpha = alpha
        self.epsilon = 1e-11
```

```
        self.batch_size = batch_size

    def change_iterations(self, iterations):
        self.iterations = iterations

    def change_alpha(self, alpha):
        self.alpha = alpha

    def sigmoid(self, z):
        s = 1 / (1 + np.exp(-z))
        return s

    def sigmoid_derivative(self, z):
        s = self.sigmoid(z)
        s = s*(1-s)
        return s

    def relu(self, z):
#           print(z)
        s = np.maximum(0, z)
        return s

    def relu_derivative(self, z):
        s = (z>0)
        #z[z >= 0] = 1
        #z[z < 0] = 0
        s = s.astype('int')
        assert(s.shape == z.shape)
        return s

    def tanh(self, z):
        return np.tanh(z)

    def tanh_derivative(self, z):
        return (1-np.square(np.tanh(z)))

    def softmax(self, z):
        z = z - z.max(axis=0, keepdims=True)
        y = np.exp(z)
        y = np.nan_to_num(y)
        y = y / y.sum(axis=0, keepdims=True)
        return y

    def softmax_derivative(self, z):
        print(z.shape)
        z = z - z.max(axis=0, keepdims=True)
        y = np.exp(z)
        y = (y * (y.sum(axis=0, keepdims=True) - y)) / np.square(
```

```
            y.sum(axis=0, keepdims=True))
        return y

    def standardize(self, X):
        X_standardized = (X-self.mean)/self.std
        #          display(X_standardized, X_standardized.shape)
        return X_standardized

    def add_layer(self, output_dim, input_dim, activation="relu"):
        activation_function = None
        derivative = None
        if activation == "relu":
            activation_function = self.relu
            derivative = self.relu_derivative
        elif activation == "sigmoid":
            activation_function = self.sigmoid
            derivative = self.sigmoid_derivative
        elif activation == "tanh":
            activation_function = self.tanh
            derivative = self.tanh_derivative
        elif activation == "softmax":
            activation_function = self.softmax
            derivative = self.softmax_derivative
        else:
            raise ("Not a valid error function")
            return
        new_layer = layer(output_dim, input_dim, activation_function,
                          derivative)
        if len(self.layers) == 0:
            self.input_shape = input_dim
        self.output_shape = output_dim
        self.layers.append(new_layer)

    def fit(self, X, y):
        self.mean, self.std = X.mean(), X.std()
        X = self.standardize(X)
        X = np.array(X).T
        y = np.array(y).T
        assert (X.shape[0] == self.input_shape)
        assert (y.shape[0] == self.output_shape)
        assert (X.shape[1] == y.shape[1])
        m = X.shape[1]
        costs = []

        #### X.shape = (number_of_features, number_of_rows)
        #### y.shape = (number_of_labels, number_of_rows)
        full_X = X.T
        full_y = y.T
```

```
        for i in range(self.iterations):
            p = np.random.permutation(m)
            full_X, full_y = full_X[p], full_y[p]
            print("Iteration Number: ",i+1)
            start = 0
            end = self.batch_size
            xxx = 2*m/(self.batch_size*3)
            while end <= m:
                X = full_X.T[:, start:end]
                y = full_y.T[:, start:end]
                start+=self.batch_size
                end+=self.batch_size
                if end%xxx == 0:
                    print("#",end='')
            #### Forward Propogation
                A = X
                for layer_no in range(len(self.layers)):
                    A = self.forward_propogation(layer_no, A)
#                   A = np.nan_to_num(A)
#                   A = A / A.sum(axis=0, keepdims=True)
#                   print(max(A.sum(axis = 1)))
                #### A.shape = (number_of_labels, number_of_rows)
                dZ = A - y
                W = self.layers[-1].W
                A = self.layers[-1].A
                dW = np.dot(dZ, A.T) / m
                #### shape of db = (output_dim, 1)
                db = (1 / m) * np.sum(dZ, axis=1, keepdims=True)
                #### shape of da_new = ((input_dim, output_dim)*(output_dim, number_of_
                #### shape of da_new = (input_dim, number_of_rows)
                dA = np.dot(W.T, dZ)
                #         print("da_new.shape = {0}".format(da_new.shape))
                self.layers[-1].dW = dW
                self.layers[-1].db = db
                #### Backward Propogation
                for layer_no in range(len(self.layers) - 2, -1, -1):
                    dA = self.backward_propogation(layer_no, dA)

                #### Update parameters

                for layer_no in range(len(self.layers)):
                    self.update_parameters(i,layer_no)

            A = full_X.T
            for layer_no in range(len(self.layers)):
                A = self.forward_propogation(layer_no, A)
            cost = self.compute_cost(A, full_y.T)
            costs.append(cost)
```

```
#             print()

        iteration_list = [i for i in range(1, len(costs) + 1)]
        plt.style.use('fivethirtyeight')
        plt.plot(iteration_list, costs)
        plt.ylabel("Loss")
        plt.xlabel("Iterations")
        plt.show()
#         display(costs)

    def forward_propogation(self, layer_no, A):
#         display("Layer number {0}".format(layer_no))
        self.layers[layer_no].A = A
        #### shape of A = (input_dim, number of rows)
        #### shape of W = (output_dim, input_dim)
        W = self.layers[layer_no].W
        b = self.layers[layer_no].b
        #### shape of b = (output_dim, 1)
        g = self.layers[layer_no].activation_function
        self.layers[layer_no].Z = np.dot(W, A) + b
        #### shape of Z = (output_dim, number_of_rows)
        A = g(self.layers[layer_no].Z)
        #### shape of A = (output_dim, number_of_rows)
        return A

    def compute_cost(self, prediction, target):
        #         display(y_hat.max())
        # shape of prediction (number_of_labels, number of training rows)
        m = prediction.shape[1]
        clipped = np.clip(prediction, self.epsilon, 1 - self.epsilon)
        cost = target * np.log(clipped) + (1 - target) * np.log(1 - clipped)
        return -np.sum(cost)/m

    def compute_error(self, prediction, target):
        denominator = np.maximum(prediction - prediction ** 2, self.epsilon)
        delta = (prediction - target) / denominator
#         delta = -np.nan_to_num(np.divide(target, prediction)) + np.nan_to_num(np.divi
        assert (delta.shape == target.shape == prediction.shape)
        return delta

    def backward_propogation(self, layer_no, dA):
        #### shape of dA = (output_dim, number_of_rows)
        #### shape of W = (output_dim, input_dim)
        #### shape of Z = (output_dim, number_of_rows)
        #### shape of A = (input_dim, number_of_rows)
        W = self.layers[layer_no].W
        g_der = self.layers[layer_no].activation_function_derivative
        Z = self.layers[layer_no].Z
```

```
        A = self.layers[layer_no].A
        m = A.shape[1]
        #### shape of dZ = (output_dim, number_of_rows)
        #         print(
        #             "dA.shape = {0}, Z.shape = {1}, activation_function = {2}".format
        #                 dA.shape, Z.shape, g_der.__name__))
        dZ = dA * g_der(Z)
        #### shape of dW = ((output_dim, number_of_rows)*(number_of_rows, input_dim))
        #### shape of dW = (output_dim, input_dim)
        dW = np.dot(dZ, A.T) / m
        #### shape of db = (output_dim, 1)
        db = (1 / m) * np.sum(dZ, axis=1, keepdims=True)
        #### shape of da_new = ((input_dim, output_dim)*(output_dim, number_of_rows))
        #### shape of da_new = (input_dim, number_of_rows)
        da_new = np.dot(W.T, dZ)
        #         print("da_new.shape = {0}".format(da_new.shape))
        self.layers[layer_no].dW = dW
        self.layers[layer_no].db = db
        self.layers[layer_no].A = self.layers[layer_no].Z = None
        return da_new

    def update_parameters(self, iteration, layer_no):
        #### shape of W, dW = (output_dim, input_dim)
        #### shape of b, db = (output_dim, 1)
        W = self.layers[layer_no].W
        b = self.layers[layer_no].b
        dW = self.layers[layer_no].dW
        db = self.layers[layer_no].db
        alph = self.alpha/(1+iteration)
        W = W - alph * dW
        b = b - alph * db
        self.layers[layer_no].W = W
        self.layers[layer_no].b = b
        self.layers[layer_no].dW = self.layers[layer_no].db = None
#         display("W", W)
#         display("b", b)

    def predict(self, X):
        X = np.array(self.standardize(X))
        X = np.array(X).T
        A = X
        for layer_no in range(len(self.layers)):
            A = self.forward_propogation(layer_no, A)
        A = A.T
#         for i in range(len(A)):
#             display(A[i])
        return A
    def calculate_accuracy(self, y_pred, y_test):
```

```
            y_hat = np.argmax(y_pred, axis=1).flatten()
            y__test = np.array(y_test)
            y__test = np.argmax(y__test, axis=1).flatten()
#             print(np.unique(y_hat, return_counts=True))
            count = 0
            for yh, y in zip(y_hat, y__test):
                if (yh == y):
                    count += 1
            return (count /  len(y_hat))
```

```
In [19]: data = pd.read_csv("Apparel/apparel-trainval.csv")
         data = data.astype('float64')
```

```
In [7]: display(data.head())
```

```
   label  pixel1  pixel2  pixel3  pixel4  pixel5  pixel6  pixel7  pixel8  \
0    2.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
1    9.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
2    6.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     5.0
3    0.0     0.0     0.0     0.0     1.0     2.0     0.0     0.0     0.0
4    3.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0

   pixel9  ...  pixel775  pixel776  pixel777  pixel778  pixel779  pixel780  \
0     0.0  ...       0.0       0.0       0.0       0.0       0.0       0.0
1     0.0  ...       0.0       0.0       0.0       0.0       0.0       0.0
2     0.0  ...       0.0       0.0       0.0      30.0      43.0       0.0
3     0.0  ...       3.0       0.0       0.0       0.0       0.0       1.0
4     0.0  ...       0.0       0.0       0.0       0.0       0.0       0.0

   pixel781  pixel782  pixel783  pixel784
0       0.0       0.0       0.0       0.0
1       0.0       0.0       0.0       0.0
2       0.0       0.0       0.0       0.0
3       0.0       0.0       0.0       0.0
4       0.0       0.0       0.0       0.0

[5 rows x 785 columns]
```

```
In [8]: y = pd.get_dummies(data['label'])
```

```
In [9]: X = data.drop('label', axis=1)
```

```
In [10]: display(X.head())
         display(y.head())
```

```
   pixel1  pixel2  pixel3  pixel4  pixel5  pixel6  pixel7  pixel8  pixel9  \
0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
1     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
```

```
2     0.0     0.0     0.0     0.0     0.0     0.0     0.0     5.0     0.0
3     0.0     0.0     0.0     1.0     2.0     0.0     0.0     0.0     0.0
4     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0

   pixel10  ...  pixel775  pixel776  pixel777  pixel778  pixel779  pixel780  \
0      0.0  ...       0.0       0.0       0.0       0.0       0.0       0.0
1      0.0  ...       0.0       0.0       0.0       0.0       0.0       0.0
2      0.0  ...       0.0       0.0       0.0      30.0      43.0       0.0
3      0.0  ...       3.0       0.0       0.0       0.0       0.0       1.0
4      0.0  ...       0.0       0.0       0.0       0.0       0.0       0.0

   pixel781  pixel782  pixel783  pixel784
0       0.0       0.0       0.0       0.0
1       0.0       0.0       0.0       0.0
2       0.0       0.0       0.0       0.0
3       0.0       0.0       0.0       0.0
4       0.0       0.0       0.0       0.0

[5 rows x 784 columns]


   0.0  1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0
0    0    0    1    0    0    0    0    0    0    0
1    0    0    0    0    0    0    0    0    0    1
2    0    0    0    0    0    0    1    0    0    0
3    1    0    0    0    0    0    0    0    0    0
4    0    0    0    1    0    0    0    0    0    0
```

In [11]:
```python
X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42)
```

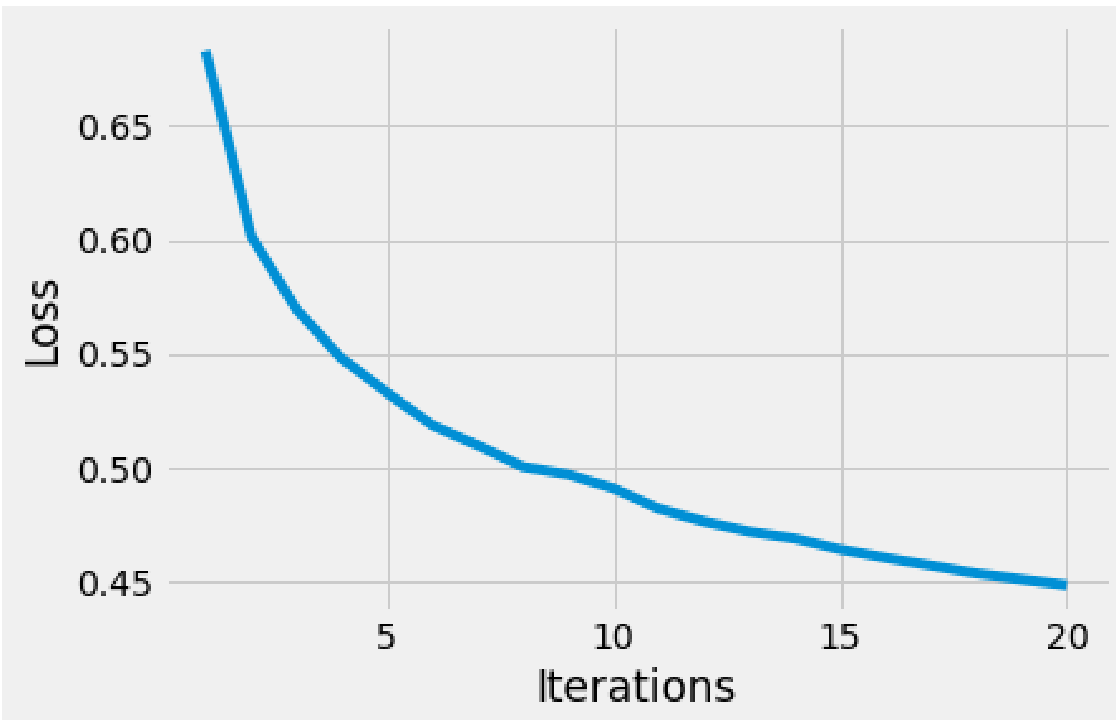
In [21]:
```python
model_tanh = NeuralNetwork(iterations=20, alpha=0.05, batch_size=50)
```

In [22]:
```python
model_tanh.add_layer(256, 784, "tanh")
model_tanh.add_layer(128, 256, "tanh")
model_tanh.add_layer(64, 128, "tanh")
# model.add_layer(50, 100, "sigmoid")
model_tanh.add_layer(10, 64, "softmax")
```

In [23]:
```python
model_tanh.fit(X_train, y_train)
```

```
Iteration Number:  1
###############Iteration Number:  2
###############Iteration Number:  3
###############Iteration Number:  4
###############Iteration Number:  5
###############Iteration Number:  6
###############Iteration Number:  7
```

```
###############Iteration Number:  8
###############Iteration Number:  9
###############Iteration Number:  10
###############Iteration Number:  11
###############Iteration Number:  12
###############Iteration Number:  13
###############Iteration Number:  14
###############Iteration Number:  15
###############Iteration Number:  16
###############Iteration Number:  17
###############Iteration Number:  18
###############Iteration Number:  19
###############Iteration Number:  20
###############
```




```
In [24]: with open('tanh.pkl', 'wb') as output:
             pickle.dump(model_tanh, output, pickle.HIGHEST_PROTOCOL)
```

```
In [25]: y_pred = model_tanh.predict(X_test)
         print("Accuracy : ",model_tanh.calculate_accuracy(y_pred, y_test)*100)
```

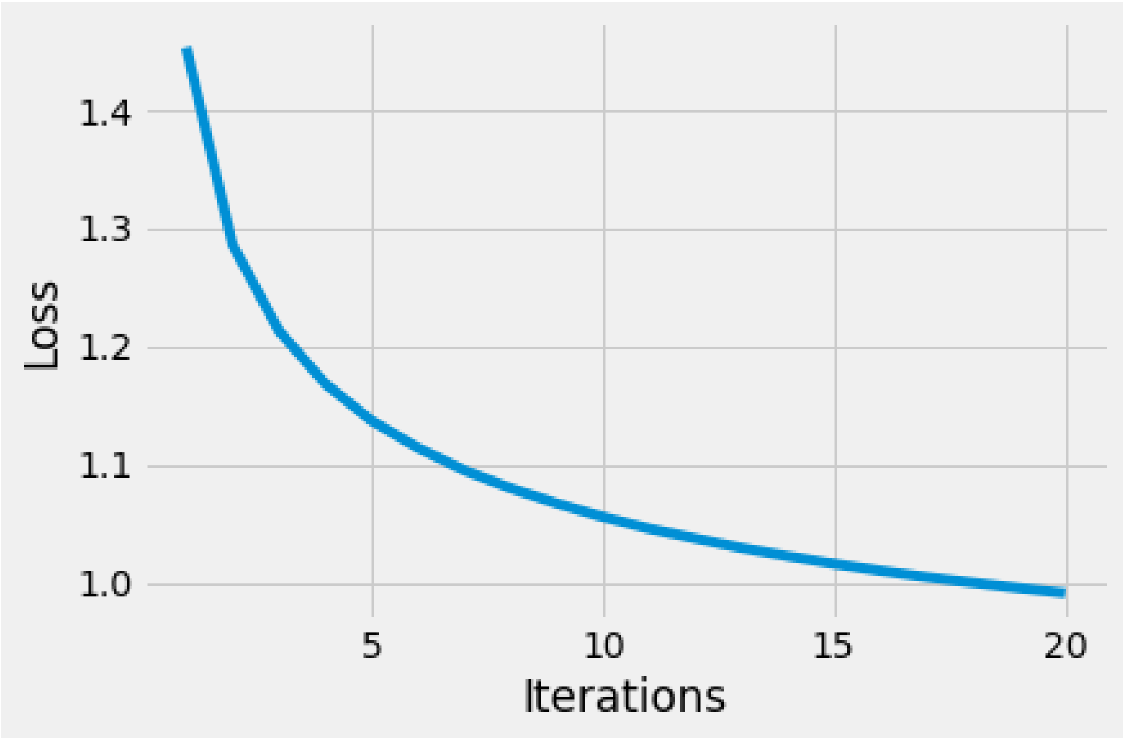
```
Accuracy :  88.36666666666667
```

```
In [26]: model_sigmoid = NeuralNetwork(iterations=20, alpha=0.2, batch_size=50)
         model_sigmoid.add_layer(256, 784, "sigmoid")
```

```
model_sigmoid.add_layer(128, 256, "sigmoid")
model_sigmoid.add_layer(64, 128, "sigmoid")
# model.add_layer(50, 100, "sigmoid")
model_sigmoid.add_layer(10, 64, "softmax")
```

In [27]:
```
model_sigmoid.fit(X_train, y_train)
```

```
Iteration Number:  1
###############Iteration Number:  2
###############Iteration Number:  3
###############Iteration Number:  4
###############Iteration Number:  5
###############Iteration Number:  6
###############Iteration Number:  7
###############Iteration Number:  8
###############Iteration Number:  9
###############Iteration Number:  10
###############Iteration Number:  11
###############Iteration Number:  12
###############Iteration Number:  13
###############Iteration Number:  14
###############Iteration Number:  15
###############Iteration Number:  16
###############Iteration Number:  17
###############Iteration Number:  18
###############Iteration Number:  19
###############Iteration Number:  20
###############
```



```
In [28]: with open('sigmoid.pkl', 'wb') as output:
             pickle.dump(model_sigmoid, output, pickle.HIGHEST_PROTOCOL)
```

```
In [29]: y_pred = model_sigmoid.predict(X_test)
         print("Accuracy : ",model_sigmoid.calculate_accuracy(y_pred, y_test)*100)
```

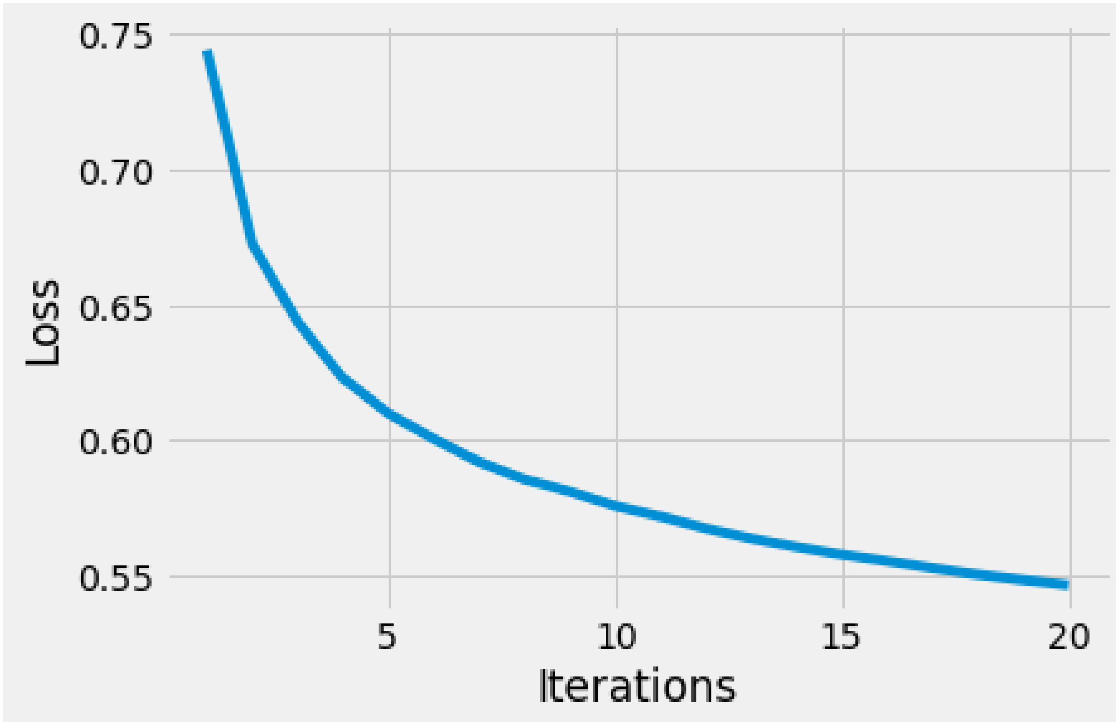
```
Accuracy :  84.38333333333333
```

```
In [30]: model_relu = NeuralNetwork(iterations=20, alpha=0.01, batch_size=50)
```

```
In [31]: model_relu.add_layer(256, 784, "relu")
         model_relu.add_layer(128, 256, "relu")
         model_relu.add_layer(64, 128, "relu")
         model_relu.add_layer(10, 64, "softmax")
```

```
In [32]: model_relu.fit(X_train, y_train)
```

```
Iteration Number:  1
###############Iteration Number:  2
###############Iteration Number:  3
###############Iteration Number:  4
###############Iteration Number:  5
###############Iteration Number:  6
###############Iteration Number:  7
###############Iteration Number:  8
###############Iteration Number:  9
###############Iteration Number:  10
###############Iteration Number:  11
###############Iteration Number:  12
###############Iteration Number:  13
###############Iteration Number:  14
###############Iteration Number:  15
###############Iteration Number:  16
###############Iteration Number:  17
###############Iteration Number:  18
###############Iteration Number:  19
###############Iteration Number:  20
###############
```




```
In [33]: with open('relu.pkl', 'wb') as output:
             pickle.dump(model_relu, output, pickle.HIGHEST_PROTOCOL)

In [34]: y_pred = model_relu.predict(X_test)
         print("Accuracy : ",model_relu.calculate_accuracy(y_pred, y_test)*100)

Accuracy :  87.26666666666667
```

0.3 Best performing Architecture for Neural Network

model_tanh.add_layer(10, 64, "softmax") - Best performing architecture has 3 hidden layer - All these layer have tanh as activation function - Loss function used is Cross Entropy loss - Dimensions of layer - - Hidden Layer 1 : 784 * 256 - - Hidden Layer 2 : 256 * 128 - - Hidden Layer 3 : 128 * 64 - - Output Layer : 64 * 10

0.4 Effect of various activation function in hidden layer:

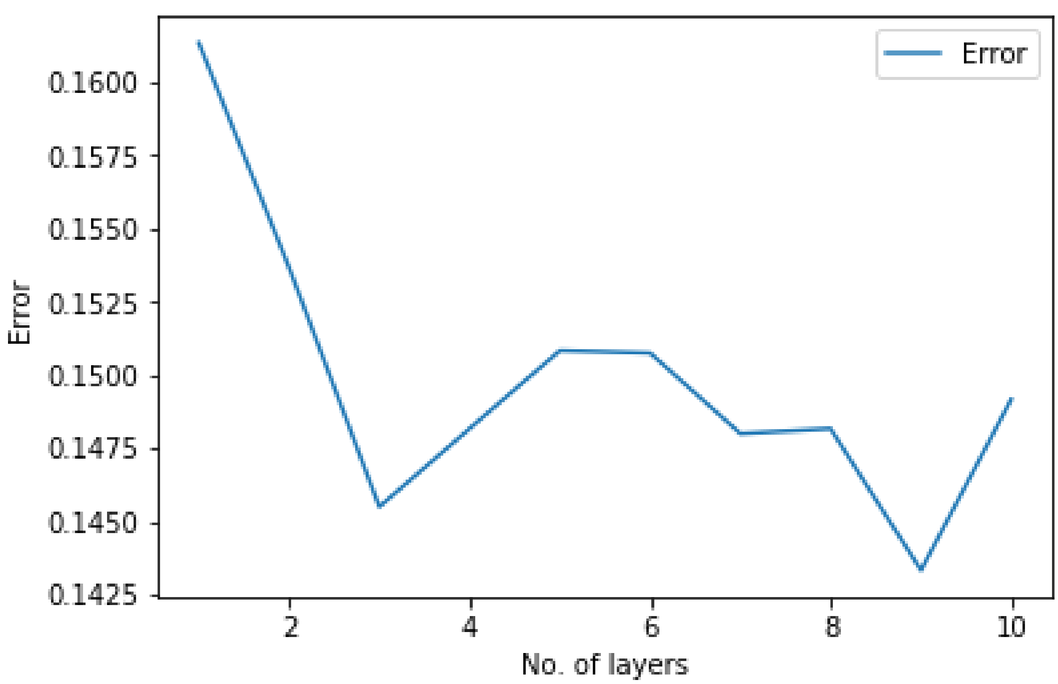
Different layers more or less perform decent. However, tanh seems to outperform others. Sigmoid is a slow learner function so with less number of iteration it is predicting comparitevely less accurately. However the fastest convergence is observent in Relu. #### Results on validation data: - Tanh 88.11% - sigmoid 83.75% - Relu 86.90%

0.5 Effect on Loss and Error of validation data with increasing number of layers

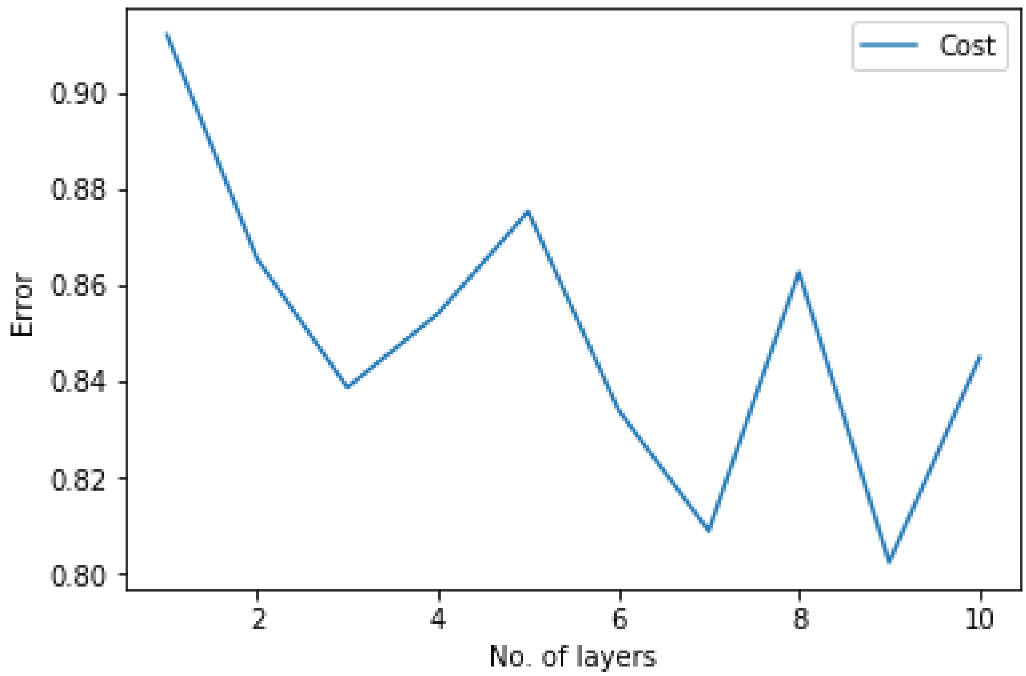
- With increasing number of layers the error and loss decrease in general. Thus model is able to fit the data.
- However after large number of layers it is increasing because of overfitting.
- Error function is smooth as compare to loss function.

```
In [21]: loss = []
         error = []
         for n_layers in range(1,11,1):
             model = NeuralNetwork(iterations=20, alpha=0.01, batch_size=50)
             model.add_layer(32, 784, "tanh")
             for i in range(n_layers):
                 model.add_layer(32, 32, "tanh")
             model.add_layer(10, 32, "softmax")
             model.fit(X_train, y_train)
             y_pred = model.predict(X_test)
             loss.append(model.compute_cost(y_pred.T, (y_test.values).T))
         #     print(loss)
             error.append(1.0 - model.calculate_accuracy(y_pred, y_test))
```

```
In [22]: plt.plot([i for i in range(1,11,1)], error, label ="Error")
         plt.legend(loc = "best")
         plt.ylabel("Error")
         plt.xlabel("No. of layers")
         plt.show()
```



```
In [23]: plt.plot([i for i in range(1,11,1)], loss, label ="Cost")
         plt.legend(loc = "best")
         plt.ylabel("Error")
         plt.xlabel("No. of layers")
         plt.show()
```

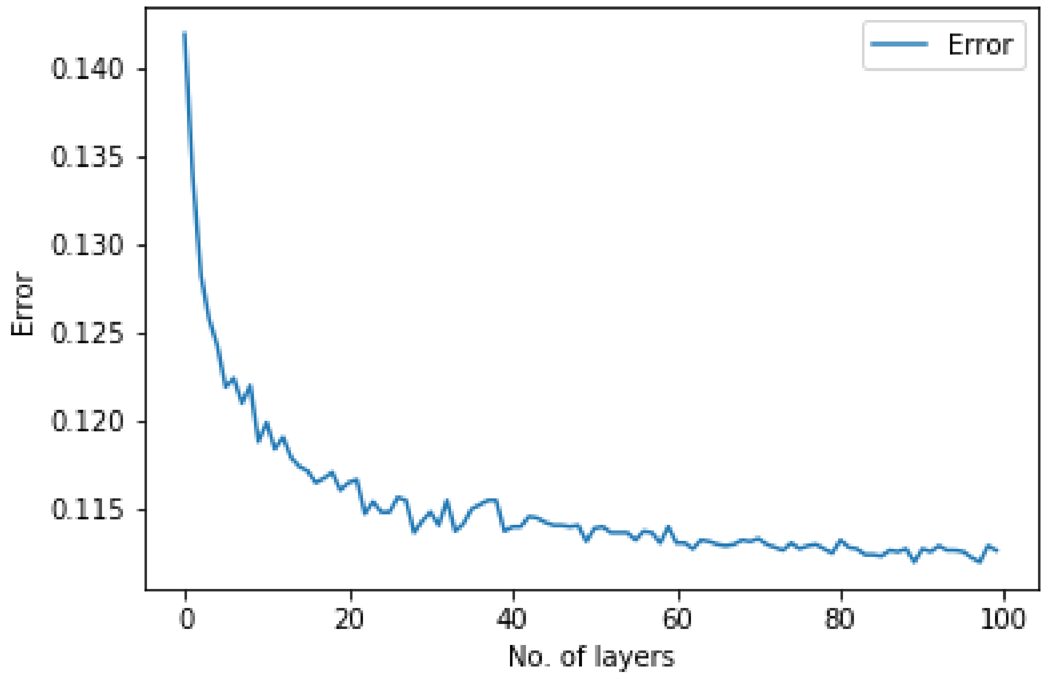


0.6 Error on validation data with increasing Epochs

- With increasing epochs the error is decreasing.
- After 50 iterations the error seems to become constant.
- It is because after certain iterations the model is not learning new things.

```
In [24]: model = NeuralNetwork(iterations=1, alpha=0.05, batch_size=50)
         model.add_layer(256, 784, "tanh")
         model.add_layer(128, 256, "tanh")
         model.add_layer(64, 128, "tanh")
         # model.add_layer(50, 100, "sigmoid")
         model.add_layer(10, 64, "softmax")
         alpha = 0.05
         error = []
         for iteration in range(100):
             model.change_alpha(alpha/(1+iteration))
             model.fit(X_train, y_train)
             y_pred = model.predict(X_test)
             error.append(1.0 - model.calculate_accuracy(y_pred, y_test))
```

```
In [25]: plt.plot([i for i in range(100)], error, label ="Error")
         plt.legend(loc = "best")
         plt.ylabel("Error")
         plt.xlabel("No. of layers")
         plt.show()
```




```
In [35]: TEST_DATA = pd.read_csv("apparel-test.csv")
         y_tanh = np.argmax(model_tanh.predict(TEST_DATA), axis=1).flatten()
         y_relu = np.argmax(model_relu.predict(TEST_DATA), axis=1).flatten()
         y_sigmoid = np.argmax(model_sigmoid.predict(TEST_DATA), axis=1).flatten()
         output = pd.DataFrame()
         output['tanh'] = y_tanh
         output['relu'] = y_relu
         output['sigmoid'] = y_sigmoid

In [38]: np.savetxt("output.csv", np.int32(y_tanh))

In [42]: tanh_file = open('tanh.pkl',"rb")
         model_loaded = pickle.load(tanh_file)
         y_pred_loaded = model_loaded.predict(X_test)
         print(model_loaded.calculate_accuracy(y_pred_loaded, y_test))

0.8836666666666667
```

1 Question 2:

- Since it is a regression problem, by keeping the output activation function as f(x) = x will give a linear combination of the hidden layer note, hence giving linear regression with Neural networks.

The hidden layer activations can be anything, but it would be preferable to use the function f(x) = x as it is more intuitive for a linear regression, i.e linear combination of input features.

However one thing to note is that price of house can't be negative so we can use RELU activation fucntion at output layer as it is essentially a linear function of values > 0.