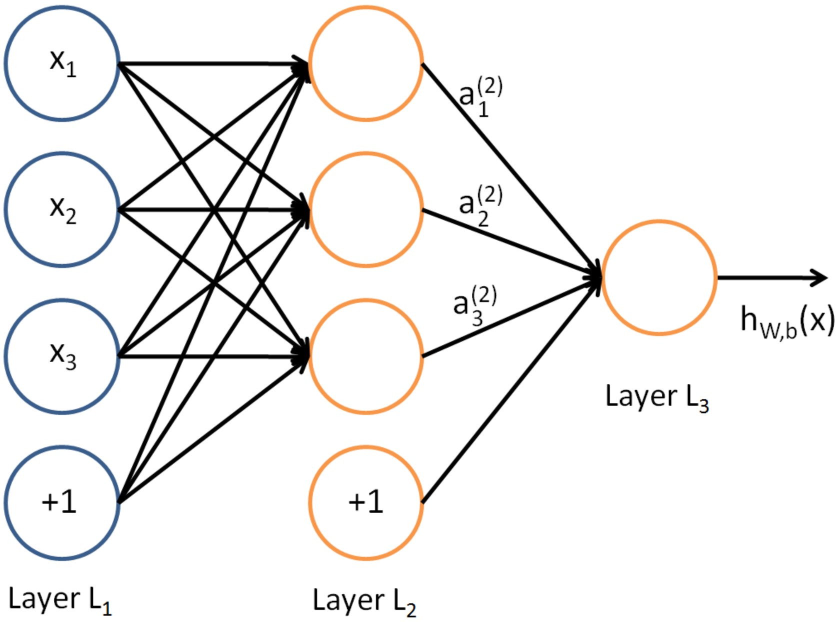
# DEVNAGRI TEXT CLASSIFICTION

**Objective :** Build and train a neural networks to classify Devanagari Handwritten Characters.

**Theory** :

Multi-layer Neural Network approach is used , in which , a input , one hidden layer and one output layer is used , as shown below .




The Hidden Layer is fully connected (except the bias term) with the Input Layer and the Output Layer is fully connected with Hidden Layer.

Let's start with The Neural Network:

1) We will start at INPUT LAYER , we forward propagate the patterns of training data through the network to generate the an putput.
2) Based on the network's output , we calculate the error that we want to minimize (eg L2/L2 or MSE) using cost function.
3) We backpropagate the error, find its derivative with respect to each weight in the network , and update the model.

   Finally , after repeating the steps for multiple epochs(eg:500,1000) and learning the weights of the Neural Network , we will use forward propagation to calculate the network output and apply a threshold function to obtain the predicted class labels .

   Since each unit is connected to all previous unit , we first calculate the activation $a_1^{(2)}$ ,

   $$z_1^{(2)} = a_0^{(1)}\ w_{1,0}^{(1)} + a_1^{(1)}\ w_{1,1}^{(1)} + .... + a_m^{(1)}\ w_{1,m}^{(1)}$$

   $a_1^{(2)} = \Phi\ (Z_1^{(2)})$ ------- Activation function

$Z_1^{(2)}$ , is net input and $\Phi(z)$ is activation function (sigmoid function), which has to be differentiable to learn the weights that connects the neurons using a gradient-based approach.

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

We are using a feedforward approach , which is , each layer serves as input to next layer without loops .

**Code Description :**

Class NeuralNet is defined with following arguments ,

1) n_output : no. of classes ( =46 , no. of Devangri class)
2) n_feature : no. of features (32x32 , =1024 )
3) n_hidden : no. of hidden layer input (=50 )
4) l1 : L1 regularization parameter (=0.0)
5) l2 : L2 regularization parameter (= 0.0)
6) epochs : time period (=1000)
7) eta : learning parameter (=.001)
8) alpha :
9) decrease_const : adaptive learning parameter
10) minibathces :

**functions defines inside class NEURALNET :**

1) **def _init_ :**
Initialises the class variables
2) **def _encode_lables :**
Encoding the class label as integer values , to avoid technical glitches. One hot encoding is used , it creates a new dummy feature for each unique value in nominal feature column.

**3) def _initilaize_weights:**

Randomly defining weights , between the value of (-1,1)

3) **def _sigmoid :** activation function , defined in theory section
4) **def _sigmoid_gradient:** return gradient of activation function , essential for backpropagation in order to calculate the gradients.
5) **def _add_bias_unit:** extra neuron is added to each pre-output layer that store the value 1 , it is not connected to any previous layer and it does not represent activity.
6) **def _get_cost :** Cost function , which has to be minimized in order to get the optimal weights. Logistic cost function is preferred over MSE , since it is a classification problem and not regression problem.
7) **def _get_gradient:** Calculates the gradient , (backpropagation step) , in order to update the weights.
8) **def _feedforward:** Feed-forward network , based on equations given in the theory section , using input features , and sigmoid activation function.
9) **def _L2_reg:** L2 regularization parameter for error calculation.
10) **def _L1_reg:** L1 regularization parameter for error calculation.
11) **def fit:** it fit's the training data , it first shuffle the training data (in order to obtain more accurate data) , adaptive learning rate is used in order to train the data and the weights are updated at each epochs.
12) **def predict:** it takes the testing or unclassified features value in order to predict the class of Devnagri text.

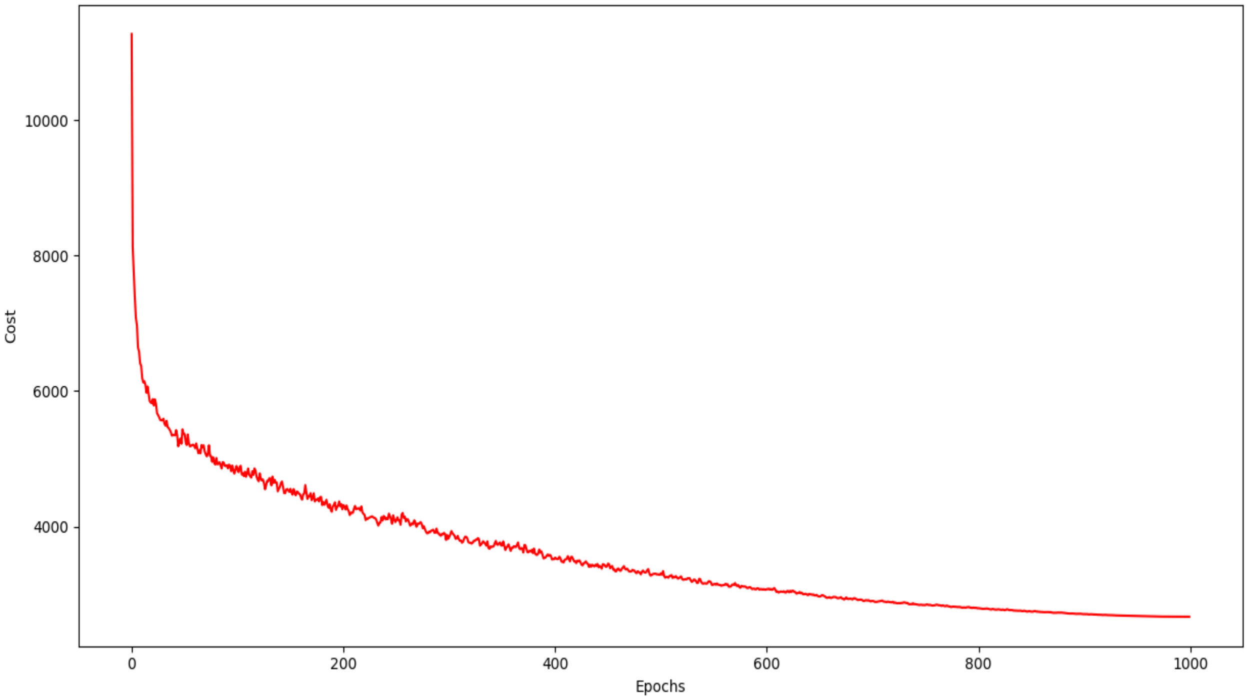
Rest of code , shows how to call the class NeuralNet ,plot cost vs epochs graph and predict values.

**Result :**

Training Accuracy : 77.75 %

```
1000Training accuracy: 77.75%
```

Cost v/s epoch



**Code : (PYTHON BASED) :**

```
import numpy as np

from scipy.special import expit # check and replace

import sys

import matplotlib.pyplot as plt

class NeuralNet(object):

  def
__init__(self,n_output,n_feature,n_hidden=30,l1=0.0,l2=0.0,epochs=500,eta=0.001,alpha=0.0,decrease_const=0.0,s
huffle=True,minibatches=1,random_state=None):

    np.random.seed(random_state)

    self.n_output = n_output

    self.n_feature = n_feature

    self.n_hidden = n_hidden

    self.w1,self.w2 = self._initialize_weights()

    self.l1=l1

    self.l2=l2

    self.epochs = epochs

    self.eta= eta

    self.alpha= alpha

    self.decrease_cost = decrease_const

    self.shuffle = shuffle

    self.minibatches = minibatches


  def _encode_labels(self,y,k):

    onehot = np.zeros((k,y.shape[0]))

    for idx,val in enumerate(y):

      onehot[val][idx]=1.0

    return onehot


  def _initialize_weights(self):

    w1 = np.random.uniform(-1.0,1.0,size=self.n_hidden*(self.n_feature+1))

    w1 = w1.reshape(self.n_hidden,self.n_feature+1)

    w2 = np.random.uniform(-1.0,1.0,size=self.n_output*(self.n_hidden+1))

    w2 = w2.reshape(self.n_output,self.n_hidden +1)

    return w1,w2
```

```
def _sigmoid(self,z):
    Act = 1.0/(1.0 + np.exp(-z))
    return Act

def _sigmoid_gradient(self,z):
    sg=self._sigmoid(z)
    return sg*(1-sg)

def _add_bias_unit(self,X,how='column'):
    if how=='column':
        X_new = np.ones((X.shape[0],X.shape[1]+1))
        X_new[:,1:]=X
    elif how=='row':
        X_new = np.ones((X.shape[0]+1,X.shape[1]))
        X_new[1:,:] = X
    else :

        raise AttributeError('how must be column or row ')

    return X_new

def _feedforward(self,X,w1,w2):
    a1 = self._add_bias_unit(X,how='column')
    z2 = w1.dot(a1.T)
    a2 = self._sigmoid(z2)
    a2 = self._add_bias_unit(a2,how='row')
    z3 = w2.dot(a2)
    a3 = self._sigmoid(z3)
    return a1,z2,a2,z3,a3

def _L2_reg(self,lambda_,w1,w2):
    return (lambda_/2.0)*(np.sum(w1[:,1:]**2)+np.sum(w2[:,1:]**2))
```

```
def _L1_reg(self,lambda_,w1,w2):
    return (lambda_/2.0)*(np.abs(w1[:,1:]).sum() + np.abs(w2[:,1:]).sum())


def _get_cost(self,y_enc,output,w1,w2):
    term1 = -y_enc*(np.log(output))
    term2 = (1-y_enc)*np.log(1-output)
    cost = np.sum(term1 - term2)
    L1_term = self._L1_reg(self.l1,w1,w2)
    L2_term = self._L2_reg(self.l2,w1,w2)
    cost = cost + L1_term + L2_term
    return cost


def _get_gradient(self, a1, a2, a3, z2, y_enc, w1, w2):
    sigma3 = a3 - y_enc                    #backpropogation
    z2 = self._add_bias_unit(z2, how='row')
    sigma2 = w2.T.dot(sigma3) * self._sigmoid_gradient(z2)


    sigma2 = sigma2[1:, :]
    grad1 = sigma2.dot(a1)
    grad2 = sigma3.dot(a2.T)


    #regularize
    grad1[:, 1:] += (w1[:, 1:] * (self.l1 + self.l2))
    grad2[:, 1:] += (w2[:, 1:] * (self.l1 + self.l2))


    return grad1, grad2


def predict(self, X):
    a1, z2, a2, z3, a3 = self._feedforward(X, self.w1, self.w2)


    y_pred = np.argmax(z3, axis=0)
    return y_pred


def fit(self, X, y, print_progress=False):
```

```
        self.cost_ =[]
        X_data ,y_data = X.copy(),y.copy()
        y_enc = self._encode_labels(y,self.n_output)

        delta_w1_prev = np.zeros(self.w1.shape)
        delta_w2_prev = np.zeros(self.w2.shape)

        for i in range(self.epochs):
            #adaptive learning rate
            self.eta/= (1+self.decrease_cost*i)

            if print_progress:
                sys.stderr.write('\rEpoch: %d/%d' % (i+1, self.epochs))
                sys.stderr.flush()

            if self.shuffle:
                idx =  np.random.permutation(y_data.shape[0])
                X_data, y_data = X_data[idx], y_data[idx]

            mini = np.array_split(range(y_data.shape[0]), self.minibatches)

            for idx in mini:
                #feedforward
                a1, z2, a2, z3, a3 = self._feedforward(X[idx], self.w1, self.w2) #errore
                cost = self._get_cost(y_enc=y_enc[:, idx],
output=a3,w1=self.w1,w2=self.w2)
                self.cost_.append(cost)
                #compute gradient via backpropagation
                grad1, grad2 = self._get_gradient(a1=a1, a2=a2,a3=a3, z2=z2,
y_enc=y_enc[:, idx],w1=self.w1,w2=self.w2)

                #update weights
                delta_w1, delta_w2 = self.eta * grad1,self.eta * grad2
                self.w1 -= (delta_w1 + (self.alpha * delta_w1_prev))
```

```
            self.w2 -= (delta_w2 + (self.alpha * delta_w2_prev))
            delta_w1_prev, delta_w2_prev = delta_w1, delta_w2
        return self



X_train = np.genfromtxt(r'C:\Users\Rag9704\Documents\devnagri_image.csv',dtype=int,delimiter=',')
y_train = np.genfromtxt(r'C:\Users\Rag9704\Documents\Devnagri_label.csv',dtype=int,delimiter=',')
X_test = np.genfromtxt(r'C:\Users\Rag9704\Documents\devnagri_test_imagee.csv',dtype=int,delimiter=',')
y_test = np.genfromtxt(r'C:\Users\Rag9704\Documents\devnagri_test_label.csv',dtype=int,delimiter=',')

nn = NeuralNet(n_output=46,n_feature=X_train.shape[1],n_hidden=50,l2=0.1,l1 = 0.0,epochs = 10,alpha =
0.001,decrease_const = 0.00001,shuffle = True,minibatches=50,random_state=1)

nn.fit(X_train,y_train,print_progress=True)

plt.plot(range(len(nn.cost_)),nn.cost_)
#plt.ylim([0,2000])
plt.ylabel('Cost')
plt.xlabel('Epochs*50')
plt.tight_layout()
plt.show()

batches =np.array_split(range(len(nn.cost_)), 1000)
cost_ary = np.array(nn.cost_)
cost_avgs = [np.mean(cost_ary[i]) for i in batches]
plt.plot(range(len(cost_avgs)),cost_avgs,color='red')
#plt.ylim([0,2000])
plt.ylabel('Cost')
plt.xlabel('Epochs')
plt.tight_layout()
plt.show()

y_train_pred = nn.predict(X_train)
acc = np.sum(y_train == y_train_pred, axis=0) / X_train.shape[0]
```

```
print('Training accuracy: %.2f%%' % (acc * 100))
```