Parameters

```
import numpy as np
import pandas as pd

#x_train,y_train,x_test,y_test,classes = load_data()
x_train = np.array([[0.1,0.3]])
y_train = np.array([1])

# Parameters
input_ = x_train
target_ = y_train

num_hidden_layers = 1
num_input_dim = 2
num_target_dim = 1  # binary classifier

print_loss_ = True
epochs = 2000 # forward pass + backward pass
epsilon = 0.01 # learning rate for gradient descent
reg_lambda = 0.001 # regularization strength
layer_dims = [2,2,1] # neurons in each layer of the network. Input
layer followed by 3 hidden layers and one target layer
```

Weights

```
def initialize_weights_n_biases(layer_dims):
      # Initialize the parameters to random values. We need to learn
these.
      parameters = {}
      parameters['W'+str(1)] = np.array([[-0.1,0.2],[0.2,-0.3]])
      parameters['b'+str(1)] = np.array([0,0])
      parameters['W'+str(2)] = np.array([[0.3],[-0.1]])
      parameters['b'+str(2)] = np.array([0])

      return parameters
```

Forward pass

```
def linear_forward(z, w, b):
      Z = z.dot(w)
      Z = Z + b
      stash = (z,w,b)
```

```
    return Z,stash

def activation_function(activation, z):
    if activation == "relu":
        a = np.maximum(0,z)
        stash = z
    elif activation == "sigmoid":
        a = 1/(1+np.exp(-1*z))
        stash = z
    return a,stash

def forward_pass(X, params):
    stashes = []
    A = X
    L = len(params) // 2
    for l in range(1,L):
        Z_prev = A
        Z,linear_stash = 
linear_forward(Z_prev,params['W'+str(l)],params['b'+str(l)])
        A,activation_stash = activation_function("relu",Z)
        stash = (linear_stash,activation_stash)
        stashes.append(stash)

    Z_,linear_stash = 
linear_forward(A,params['W'+str(L)],params['b'+str(L)])
    A_,activation_stash = activation_function("sigmoid",Z_)
    stash = (linear_stash,activation_stash)
    stashes.append(stash)
    return A_,stashes

def compute_loss(loss_func, A, Y,i):
    m = Y.shape[0]
    if loss_func == "binary_crossentropy":
        A = A.flatten()
        cost = np.sum((-(Y*np.log(A))-((1-Y)*np.log(1-
A))),axis=0,keepdims = 1)
        cost = np.squeeze(cost)
    return cost
```

Backward pass

```
def intermediate_differentiation(dEA, activation, stash):
    if activation == "relu":
        z = stash
        dAZ = 1
        dEZ = np.array(dEA,copy=True)
        dEZ[z<=0] = 0
```

```
    elif activation == "sigmoid":
        z = stash
        tmp = 1/(1+np.exp(-z))
        dAZ = tmp*(1-tmp) # differentiation of output w.r.t input
dA/dZ
        # multiply dE/dA * dA/dZ
        dEZ = dEA*dAZ
    return dEZ

def error_rate_calc(dEZ, stash):
    z,w,b = stash
    m = z.shape[0]
    dZW = z.T # rate of change of input w.r.t weight, dZ/dW = z
    dEW = np.dot(dZW,dEZ)/m # rate of change of error w.r.t weight,
dE/dW = dE/dZ * dZ/dW
    dEb = np.sum(dEZ,axis=0,keepdims=1)/m # rate of change of error
w.r.t bias, dE/db = dE/dZ * dZ/db
    dA_prev = np.dot(dEZ,w.T) # error propagated backward

    return dA_prev,dEW, dEb

def linear_backward(dEA, stash, activation):
    linear_stash, activation_stash = stash
    dEZ =
intermediate_differentiation(dEA,activation,activation_stash) # dE/dZ
    dA_prev,dW, db = error_rate_calc(dEZ, linear_stash)
    return dA_prev,dW,db

def backward_pass(A,Y,stashes):
    grads = {}
    L =len(stashes)
    Y = Y.reshape(A.shape)

    dEA = -(np.divide(Y,A)-np.divide(1-Y,1-A)) # differentiation of
error w.r.t output dE/dA
    current_stash = stashes[L-1]
    grads['dA'+str(L-1)],grads['dW'+str(L)],grads['db'+str(L)] =
linear_backward(dEA,current_stash,"sigmoid")
    for l in reversed(range(L-1)):
        current_stash = stashes[l]

    grads['dA'+str(l)],grads['dW'+str(l+1)],grads['db'+str(l+1)] =
linear_backward(grads['dA'+str(l+1)],current_stash,"relu")

    return grads
```

Inference

```
def update_parameters(parameters, grads):
      L = len(parameters) // 2
      for l in range(L):
            parameters['W'+str(l+1)] = parameters['W'+str(l+1)] -
epsilon * grads['dW'+str(l+1)] # W = W - lr*(dE/dW)
            parameters['b'+str(l+1)] = parameters['b'+str(l+1)] -
epsilon * grads['db'+str(l+1)] # b = b - lr*(dE/db)
      return parameters

# Build Sequential model
def build_sequential_model(X, Y, layer_dims, print_loss = False):
      params = initialize_weights_n_biases(layer_dims)

      costs = []
      for i in range(0,epochs):

            # Forward Propagation
            A,caches = forward_pass(X, params)
            # Error Calculation
            cost = compute_loss("binary_crossentropy",A,Y,i)
            # Backward Propagation
            grads = backward_pass(A, Y, caches)
            # Update parameters
            params = update_parameters(params,grads)
            costs.append(cost)
            if(print_loss == True and i%100 == 0):
                  print("cost at iteration {} is {}".format(i,cost))

      return params


model = build_sequential_model(input_, target_, layer_dims,
print_loss_)

def predict(X,Y,model):
      m = X.shape[0]
      res = np.zeros(m)
      probabs, stashes = forward_pass(X,model)
      for i in range(0,probabs.shape[0]):
            if probabs[i][0] > 0.5:
                  res[i] = 1
            else:
                  res[i] = 0
      print("Accuracy: "+str(np.sum(res == Y)/m))
      return res

train_data_prediction = predict(x_train,y_train,model)
```

```
cost at iteration 0 is 0.6856753052962775
cost at iteration 100 is 0.47182510825630847
cost at iteration 200 is 0.3364123438272523
cost at iteration 300 is 0.2460693926791892
cost at iteration 400 is 0.18382909759742236
cost at iteration 500 is 0.1401625468799544
cost at iteration 600 is 0.10910931593674726
cost at iteration 700 is 0.08670714721935192
cost at iteration 800 is 0.07027752249258523
cost at iteration 900 is 0.05800965371689404
cost at iteration 1000 is 0.048679277571707286
cost at iteration 1100 is 0.04145443577125768
cost at iteration 1200 is 0.03576411571293002
cost at iteration 1300 is 0.031211189721155382
cost at iteration 1400 is 0.027515290012331866
cost at iteration 1500 is 0.02447535200652837
cost at iteration 1600 is 0.021944909598761914
cost at iteration 1700 is 0.019815637527586548
cost at iteration 1800 is 0.018006251531653517
cost at iteration 1900 is 0.01645492005463617
Accuracy: 1.0
```