Notebook

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import tensorflow as tf
import cv2
from sklearn import preprocessing


sns.set()


def initialize_parameters(n_x, n_h, n_y):

    #

    W1 = np.random.randn(n_h, n_x) * 0.01
    b1 = np.zeros(shape=(n_h, 1))
    W2 = np.random.randn(n_y, n_h) * 0.01
    b2 = np.zeros(shape=(n_y, 1))

    assert (W1.shape == (n_h, n_x))
    assert (b1.shape == (n_h, 1))
    assert (W2.shape == (n_y, n_h))
    assert (b2.shape == (n_y, 1))

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters


def forward_pass(X, parameters):
    #print(parameters)
    # to make forward pass calculations we need W1 and W2 so we will extract them from dic
    W1 = parameters['W1']
    W2 = parameters['W2']
    b1 = parameters['b1']
    b2 = parameters['b2']
    #print(parameters)
    #print(W1)
    #print(b1)
    # first layer calculations - hidden layer calculations
    Z1 = np.dot(W1, X) + b1
    #print(Z1)
    # for i in range(len(Z1)):
    #   print(i)
    #   print(Z1[i])
    A1 = tanh(Z1)  # activation in the first layer is tanh
    #print(A1)
```

```
    # output layer calculations
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)# A2 are predictions, y_hat

    # cache values for backpropagation calculations
    cache = {'Z1':Z1,
             'A1':A1,
             'Z2':Z2,
             'A2':A2
            }
    # print(A2.shape)
    return A2, cache


def backward_pass(parameters, cache, X, Y):

    # unpack paramaeters and cache to get values for backpropagation calculations
    W1 = parameters['W1']
    W2 = parameters['W2']

    Z1 = cache['Z1']
    A1 = cache['A1']
    Z2 = cache['Z2']
    A2 = cache['A2']

    m = X.shape[1] # number of examples in a training set
    #print(A2)
    #print(A2.shape)
    dZ2= A2 - Y


    dW2 = (1 / m) * np.dot(dZ2, A1.T)
    db2 = (1 / m) * np.sum(dZ2, axis=1, keepdims=True)   # keepdims - prevents python to c

    dZ1 = np.multiply(np.dot(W2.T, dZ2), 1 - np.power(A1, 2)) # we use tanh activation fur
    dW1 = (1 / m) * np.dot(dZ1, X.T)
    db1 = (1 / m) * np.sum(dZ1, axis=1, keepdims=True)

    grads = {"dW1": dW1,
             "db1": db1,
             "dW2": dW2,
             "db2": db2}

    return grads


def initialize_adam(parameters) :

    L = len(parameters) // 2 # number of layers in the neural networks
    v = {}
    s = {}

    # v- exponentially weighted average of the gradient
    # s -exponentially weighted average of the squared gradient
    for l in range(L):
        v["dW1" ] = np.zeros(parameters["W1" ].shape)
```

```
        v["db1" ] = np.zeros(parameters["b1" ].shape)
        s["dW2" ] = np.zeros(parameters["W2" ].shape)
        s["db2" ] = np.zeros(parameters["b2" ].shape)

    return v, s


def update_parameters_with_adam(parameters, grads, v, s, t, learning_rate = 0.01,
                                beta1 = 0.9, beta2 = 0.999,  epsilon = 1e-8):



    L = len(parameters) // 2                 # number of layers in the neural networks
    v_corrected = {}                         # Initializing first moment estimate, python
    s_corrected = {}                         # Initializing second moment estimate, pythor


    for l in range(L):
        v["dW1"] = beta1*v["dW1" ] + (1-beta1)*grads['dW1' ]
        v["db1" ] = beta1*v["db1" ] + (1-beta1)*grads['db1' ]


        # Compute bias-corrected first moment estimate.

        v_corrected["dW1" ] = v["dW1" ]/(1 - beta1**t)
        v_corrected["db1" ] = v["db1" ]/(1 - beta1**t)


        # Moving average of the squared gradients.

        s["dW2"] = beta2*s["dW2" ] + (1-beta2)*(grads['dW2' ])**2
        s["db2" ] = beta2*s["db2" ] + (1-beta2)*(grads['db2' ])**2


        # Compute bias-corrected second raw moment estimate.

        s_corrected["dW2" ] = s["dW2" ]/(1 - beta2**t)
        s_corrected["db2" ] = s["db2" ]/(1 - beta2**t)


        parameters["W1" ] = parameters["W1" ]- (learning_rate*v_corrected["dW1" ])/(np.sqr
        parameters["b1" ] = parameters["b1" ]- (learning_rate*v_corrected["db1" ])/(np.sqr

    return parameters, v, s


def NN_model(X,Y,n_h, num_iterations, learning_rate):

    n_x = X.shape[0] # size of an input layer = number of features
    n_y = Y.shape[0] # size of an output layer
    parameters = initialize_parameters(n_x, n_h, n_y)

    #unpack parameters
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
```

```
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    #print(parameters)
    param = {}
    for i in range(num_iterations):
      #print(i)
      A2, cache = forward_pass(X, parameters)
      grads = backward_pass(parameters, cache, X, Y)
      v,s = initialize_adam(parameters)
      param = update_parameters_with_adam(parameters,grads,v,s , t = 2 )
      print(i,param)
    #print(A2)
    # print(cache)
    return parameters


def predict(parameters, X):

    A2, cache = forward_pass(X, parameters)
    predictions = np.round(A2)

    return predictions


def unpickle(file):
    import pickle
    with open(file, 'rb') as fo:
        dict = pickle.load(fo, encoding='bytes')
    return dict


def loadbatch(batchname):
    batch = unpickle('/content/drive/MyDrive/Colab_Notebooks'+"/"+batchname)
    return batch


def loadlabelnames():
    meta = unpickle('/content/drive/MyDrive/Colab_Notebooks'+"/"+'batches.meta')
    return meta[b'label_names']


batch1 = loadbatch('data_batch_1')
print("Number of items in the batch is", len(batch1))

# Display all keys, so we can see the ones we want
print('All keys in the batch:', batch1.keys())
```

```
Number of items in the batch is 4
All keys in the batch: dict_keys([b'batch_label', b'labels', b'data', b'filenames'])
```

```
data = batch1[b'data'][:,:1023]
labels = batch1[b'labels']
print ("size of data in this batch:", len(data), ", size of labels:", len(labels))
print (type(data))
# print(data.shape)
# print(labels)
label = []
```

```
data_class = []
names = loadlabelnames()
```

```
size of data in this batch: 10000 , size of labels: 10000
<class 'numpy.ndarray'>
```

```
for i in range(len(labels)):
  if labels[i] == 1 or labels[i]==6:
    label.append(labels[i])
    data_class.append(data[i])
```

```
df = np.array(data_class)
df2 = np.array(label)
```

```
num_train = int(.70 * len(df))
num_test = int(0.15 * len(df))

x_train, y_train = data_class[:num_train], label[:num_train]
x_test, y_test = data_class[num_test:], label[num_test:]

x_train  = np.array(x_train)
y_train  = np.array(y_train)
x_test  = np.array(x_test)
y_test  = np.array(y_test)
```

```
arr = [y_train.tolist()]
y_train = np.array(arr)

arr1 = [y_test.tolist()]
y_test = np.array(arr1)
```

```
num_iterations = 20000
learning_rate = 0.01
n_h = 4
parameters_final = NN_model(x_train.T,y_train,n_h, num_iterations, learning_rate)
```

```
Y_predictions_test = predict(parameters_final, x_test.T)
Y_predictions_train = predict(parameters_final, x_train.T)
```

```
acc = np.mean(y_train == Y_predictions_train)
acc
```

```
0.47360912981455067
```

Resources -

1. https://towardsdatascience.com/how-to-implement-an-adam-optimizer-from-scratch-76e7b
2. https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a29
3. https://machinelearningmastery.com/tour-of-optimization-algorithms/