Project:

This project was to implement a 3 layer neural network first without regularization, then with L2 regularization and finally with dropout; with increasing accuracy on the test set; but slightly hurting training set performance (since regularization limits the ability of the network to over fit to the training set).

Problem Statement: A football team would like you to recommend positions where their goal keeper should kick the ball so that their team's players can then hit it with their head.

Non regularized model

```
# import packages

import numpy as np

import matplotlib.pyplot as plt

from reg_utils import sigmoid, relu, plot_decision_boundary, initialize_parameters, load_2D_dataset, predict_dec

from reg_utils import compute_cost, predict, forward_propagation, backward_propagation, update_parameters

import sklearn

import sklearn.datasets

import scipy.io

from testCases import *

#load dataset

train_X, train_Y, test_X, test_Y = load_2D_dataset()

#Function

def model(X, Y, learning_rate = 0.3, num_iterations = 30000, print_cost = True, lambd = 0, keep_prob = 1):

    """

    Implements a three-layer neural network: LINEAR->RELU->LINEAR->RELU->LINEAR->SIGMOID.

    Arguments:

    X -- input data, of shape (input size, number of examples)
```

```
Y -- true "label" vector (1 for blue dot / 0 for red dot), of shape (output size, number of examples)
learning_rate -- learning rate of the optimization
num_iterations -- number of iterations of the optimization loop
print_cost -- If True, print the cost every 10000 iterations
lambd -- regularization hyperparameter, scalar
keep_prob - probability of keeping a neuron active during drop-out, scalar.
Returns:
parameters -- parameters learned by the model. They can then be used to predict.
"""
grads = {}
costs = [  ]                    # to keep track of the cost
m = X.shape[1]                  # number of examples
layers_dims = [X.shape[0], 20, 3, 1]
# Initialize parameters dictionary.
parameters = initialize_parameters(layers_dims)

# Loop (gradient descent)
for i in range(0, num_iterations):
   # Forward propagation: LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID.
   if keep_prob == 1:
      a3, cache = forward_propagation(X, parameters)
   elif keep_prob < 1:
      a3, cache = forward_propagation_with_dropout(X, parameters, keep_prob)

   # Cost function
```

```
    if lambd == 0:
        cost = compute_cost(a3, Y)
    else:
        cost = compute_cost_with_regularization(a3, Y, parameters, lambd)

    # Backward propagation.
    assert(lambd==0 or keep_prob==1)    # it is possible to use both L2 regularization and dropout,
                                        # but this assignment will only explore one at a time
    if lambd == 0 and keep_prob == 1:
        grads = backward_propagation(X, Y, cache)
    elif lambd != 0:
        grads = backward_propagation_with_regularization(X, Y, cache, lambd)
    elif keep_prob < 1:
        grads = backward_propagation_with_dropout(X, Y, cache, keep_prob)

    # Update parameters.
    parameters = update_parameters(parameters, grads, learning_rate)

    # Print the loss every 10000 iterations
    if print_cost and i % 10000 == 0:
        print("Cost after iteration {}: {}".format(i, cost))
    if print_cost and i % 1000 == 0:
        costs.append(cost)
return parameters
```

--

L2 regularization

```
def compute_cost_with_regularization(A3, Y, parameters, lambd):
    """
    Implement the cost function with L2 regularization.
    Arguments:
    A3 -- post-activation, output of forward propagation, of shape (output size, number of examples)
    Y -- "true" labels vector, of shape (output size, number of examples)
    parameters -- python dictionary containing parameters of the model

    Returns:
    cost - value of the regularized loss function
    """
    m = Y.shape[1]
    W1 = parameters["W1"]
    W2 = parameters["W2"]
    W3 = parameters["W3"]

    cross_entropy_cost = compute_cost(A3, Y) # This gives you the cross-entropy part of the cost

    L2_regularization_cost =
lambd*(np.sum(np.square(W1))+np.sum(np.square(W2))+np.sum(np.square(W3)))/(2*m)
     cost = cross_entropy_cost + L2_regularization_cost
    return cost
```

--

```
def backward_propagation_with_regularization(X, Y, cache, lambd):
    """
    Implements the backward propagation of our baseline model to which we added an L2 regularization.

    Arguments:
    X -- input dataset, of shape (input size, number of examples)
    Y -- "true" labels vector, of shape (output size, number of examples)
    cache -- cache output from forward_propagation()
    lambd -- regularization hyperparameter, scalar

    Returns:
    gradients -- A dictionary with the gradients with respect to each parameter, activation and pre-activation variables
    """

    m = X.shape[1]
    (Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3, b3) = cache

   dZ3 = A3 - Y
   dW3 = 1./m * np.dot(dZ3, A2.T) + (lambd*W3)/m
    db3 = 1./m * np.sum(dZ3, axis=1, keepdims = True)
    dA2 = np.dot(W3.T, dZ3)
    dZ2 = np.multiply(dA2, np.int64(A2 > 0))
     dW2 = 1./m * np.dot(dZ2, A1.T) + (lambd*W2)/m
    db2 = 1./m * np.sum(dZ2, axis=1, keepdims = True)
    dA1 = np.dot(W2.T, dZ2)
```

```
    dZ1 = np.multiply(dA1, np.int64(A1 > 0))
    dW1 = 1./m * np.dot(dZ1, X.T) + (lambd*W1)/m
    db1 = 1./m * np.sum(dZ1, axis=1, keepdims = True)

    gradients = {"dZ3": dZ3, "dW3": dW3, "db3": db3,"dA2": dA2,
                 "dZ2": dZ2, "dW2": dW2, "db2": db2, "dA1": dA1,
                 "dZ1": dZ1, "dW1": dW1, "db1": db1}

    return gradients
```

--

Dropout regularization

```
def forward_propagation_with_dropout(X, parameters, keep_prob = 0.5):
    """
    Implements the forward propagation: LINEAR -> RELU + DROPOUT -> LINEAR -> RELU + DROPOUT -> LINEAR -> SIGMOID.
    Arguments:
    X -- input dataset, of shape (2, number of examples)
    parameters -- python dictionary containing your parameters "W1", "b1", "W2", "b2", "W3", "b3":
                  W1 -- weight matrix of shape (20, 2)
                  b1 -- bias vector of shape (20, 1)
                  W2 -- weight matrix of shape (3, 20)
                  b2 -- bias vector of shape (3, 1)
                  W3 -- weight matrix of shape (1, 3)
                  b3 -- bias vector of shape (1, 1)
    keep_prob - probability of keeping a neuron active during drop-out, scalar
```

```
    Returns:
    A3 -- last activation value, output of the forward propagation, of shape (1,1)
    cache -- tuple, information stored for computing the backward propagation
    """
    # retrieve parameters
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    W3 = parameters["W3"]
    b3 = parameters["b3"]

    # LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID
    Z1 = np.dot(W1, X) + b1
    A1 = relu(Z1)
    D1 = np.random.randn(A1.shape[0],A1.shape[1])                                  # Step 1: initialize matrix D1 = np.random.rand(..., ...)
    D1 = D1<keep_prob                                 # Step 2: convert entries of D1 to 0 or 1 (using keep_prob as the threshold)
    A1 = A1*D1                                # Step 3: shut down some neurons of A1
    A1 = A1/keep_prob                                 # Step 4: scale the value of neurons that haven't been shut down
    Z2 = np.dot(W2, A1) + b2
    A2 = relu(Z2)
     D2 = np.random.randn(A2.shape[0],A2.shape[1])                                  # Step 1: initialize matrix D2 = np.random.rand(..., ...)
```

```
    D2 = D2<keep_prob                                       # Step 2: convert entries of D2 to 0 or 1 (using keep_prob as the threshold)
    A2 = A2*D2                                              # Step 3: shut down some neurons of A2
    A2 = A2/keep_prob                                       # Step 4: scale the value of neurons that haven't been shut down
    Z3 = np.dot(W3, A2) + b3
    A3 = sigmoid(Z3)


    cache = (Z1, D1, A1, W1, b1, Z2, D2, A2, W2, b2, Z3, A3, W3, b3)
    return A3, cache
--
def backward_propagation_with_dropout(X, Y, cache, keep_prob):
    """
    Implements the backward propagation of our baseline model to which we added dropout.

    Arguments:
    X -- input dataset, of shape (2, number of examples)
    Y -- "true" labels vector, of shape (output size, number of examples)
    cache -- cache output from forward_propagation_with_dropout()
    keep_prob - probability of keeping a neuron active during drop-out, scalar

    Returns:
    gradients -- A dictionary with the gradients with respect to each parameter, activation and pre-activation variables
    """

    m = X.shape[1]
```

```
    (Z1, D1, A1, W1, b1, Z2, D2, A2, W2, b2, Z3, A3, W3, b3) = cache


    dZ3 = A3 - Y

    dW3 = 1./m * np.dot(dZ3, A2.T)

    db3 = 1./m * np.sum(dZ3, axis=1, keepdims = True)

    dA2 = np.dot(W3.T, dZ3)

   dA2 = dA2*D2            # Step 1: Apply mask D2 to shut down the same neurons as during the forward
propagation

    dA2 = dA2/keep_prob           # Step 2: Scale the value of neurons that haven't been shut down

    dZ2 = np.multiply(dA2, np.int64(A2 > 0))

    dW2 = 1./m * np.dot(dZ2, A1.T)

    db2 = 1./m * np.sum(dZ2, axis=1, keepdims = True)

    dA1 = np.dot(W2.T, dZ2)

    dA1 = dA1*D1           # Step 1: Apply mask D1 to shut down the same neurons as during the forward
propagation

    dA1 = dA1/keep_prob           # Step 2: Scale the value of neurons that haven't been shut down

    dZ1 = np.multiply(dA1, np.int64(A1 > 0))

    dW1 = 1./m * np.dot(dZ1, X.T)

    db1 = 1./m * np.sum(dZ1, axis=1, keepdims = True)


    gradients = {"dZ3": dZ3, "dW3": dW3, "db3": db3,"dA2": dA2,

                 "dZ2": dZ2, "dW2": dW2, "db2": db2, "dA1": dA1,

                 "dZ1": dZ1, "dW1": dW1, "db1": db1}

    return gradients
```