

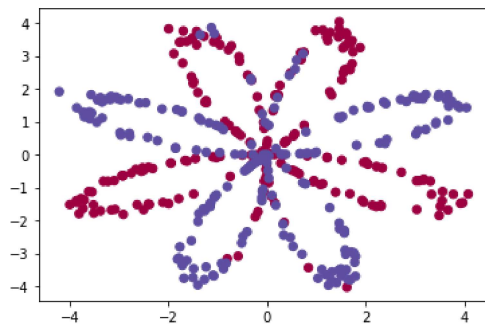
```
# Package imports
import numpy as np
import matplotlib.pyplot as plt
from testCases import *
import sklearn
import sklearn.datasets
import sklearn.linear_model
from planar_utils import plot_decision_boundary, sigmoid, load_planar_dataset, load_extra_datasets
```

```
%matplotlib inline
```

```
np.random.seed(1) # set a seed so that the results are consistent
```

```
X, Y = load_planar_dataset()
```

```
# Visualize the data:
plt.scatter(X[0, :], X[1, :], c=Y, s=40, cmap=plt.cm.Spectral);
```



```
### START CODE HERE ### (~ 3 lines of code)
```

```
shape_X = X.shape
```

```
shape_Y = Y.shape
```

```
m = shape_X[1] # training set size
```

```
### END CODE HERE ###
```

```
print('The shape of X is: ' + str(shape_X))
```

```
print('The shape of Y is: ' + str(shape_Y))
```

```
print('I have m = %d training examples!' % (m))
```

```
The shape of X is: (2, 400)
```

```
The shape of Y is: (1, 400)
```

```
I have m = 400 training examples!
```

```
# Train the logistic regression classifier
```

```
clf = sklearn.linear_model.LogisticRegressionCV();
```

```
clf.fit(X.T, Y.T);
```

```
/usr/local/lib/python3.8/dist-packages/sklearn/utils/validation.py:993: DataConversionWarning: A column-vector y was passed when a
y = column_or_1d(y, warn=True)
```

```
# Plot the decision boundary for logistic regression
```

```
plot_decision_boundary(lambda x: clf.predict(x), X, Y)
```

```
plt.title("Logistic Regression")
```

```
# Print accuracy
```

```
LR_predictions = clf.predict(X.T)
```

```
print('Accuracy of logistic regression: %d ' % float((np.dot(Y,LR_predictions) + np.dot(1-Y,1-LR_predictions))/float(Y.size)*100) +
      '% ' + "(percentage of correctly labelled datapoints)")
```

Accuracy of logistic regression: 47 % (percentage of correctly labelled datapoints)

### Logistic Regression

---

# GRADED FUNCTION: layer\_sizes

```
def layer_sizes(X, Y):
    """
    Arguments:
    X -- input dataset of shape (input size, number of examples)
    Y -- labels of shape (output size, number of examples)

    Returns:
    n_x -- the size of the input layer
    n_h -- the size of the hidden layer
    n_y -- the size of the output layer
    """
    ### START CODE HERE ### (= 3 lines of code)
    n_x = X.shape[0] # size of input layer
    n_h = 4
    n_y = Y.shape[0] # size of output layer
    ### END CODE HERE ###
    return (n_x, n_h, n_y)
```

```
X_assess, Y_assess = layer_sizes_test_case()
(n_x, n_h, n_y) = layer_sizes(X_assess, Y_assess)
print("The size of the input layer is: n_x = " + str(n_x))
print("The size of the hidden layer is: n_h = " + str(n_h))
print("The size of the output layer is: n_y = " + str(n_y))
```

```
The size of the input layer is: n_x = 5
The size of the hidden layer is: n_h = 4
The size of the output layer is: n_y = 2
```

# GRADED FUNCTION: initialize\_parameters

```
def initialize_parameters(n_x, n_h, n_y):
    """
    Argument:
    n_x -- size of the input layer
    n_h -- size of the hidden layer
    n_y -- size of the output layer

    Returns:
    params -- python dictionary containing your parameters:
        W1 -- weight matrix of shape (n_h, n_x)
        b1 -- bias vector of shape (n_h, 1)
        W2 -- weight matrix of shape (n_y, n_h)
        b2 -- bias vector of shape (n_y, 1)
    """
```

```
np.random.seed(2) # we set up a seed so that your output matches ours although the initialization is random.
```

```
### START CODE HERE ### (= 4 lines of code)
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))
### END CODE HERE ###
```

```
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
```

```
n_x, n_h, n_y = initialize_parameters_test_case()
```

```
parameters = initialize_parameters(n_x, n_h, n_y)
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

```

W1 = [[-0.00416758 -0.00056267]
      [-0.02136196  0.01640271]
      [-0.01793436 -0.00841747]
      [ 0.00502881 -0.01245288]]
b1 = [[0.]
      [0.]
      [0.]
      [0.]]
W2 = [[-0.01057952 -0.00909008  0.00551454  0.02292208]]
b2 = [[0.]]

# GRADED FUNCTION: forward_propagation

def forward_propagation(X, parameters):
    """
    Argument:
    X -- input data of size (n_x, m)
    parameters -- python dictionary containing your parameters (output of initialization function)

    Returns:
    A2 -- The sigmoid output of the second activation
    cache -- a dictionary containing "Z1", "A1", "Z2" and "A2"
    """
    # Retrieve each parameter from the dictionary "parameters"
    ### START CODE HERE ### (= 4 lines of code)
    W1 = parameters['W1']
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']
    ### END CODE HERE ###

    # Implement Forward Propagation to calculate A2 (probabilities)
    ### START CODE HERE ### (= 4 lines of code)
    Z1 = np.dot(W1, X) + b1
    A1 = np.tanh(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)
    ### END CODE HERE ###

    assert(A2.shape == (1, X.shape[1]))

    cache = {"Z1": Z1,
            "A1": A1,
            "Z2": Z2,
            "A2": A2}

    return A2, cache

X_assess, parameters = forward_propagation_test_case()

A2, cache = forward_propagation(X_assess, parameters)

# Note: we use the mean here just to make sure that your output matches ours.
print(np.mean(cache['Z1']), np.mean(cache['A1']), np.mean(cache['Z2']), np.mean(cache['A2']))

-0.0004997557777419913 -0.000496963353231779 0.00043818745095914653 0.500109546852431

# GRADED FUNCTION: compute_cost

def compute_cost(A2, Y, parameters):
    """
    Computes the cross-entropy cost given in equation (13)

    Arguments:
    A2 -- The sigmoid output of the second activation, of shape (1, number of examples)
    Y -- "true" labels vector of shape (1, number of examples)
    parameters -- python dictionary containing your parameters W1, b1, W2 and b2

    Returns:
    cost -- cross-entropy cost given equation (13)
    """

    m = Y.shape[1] # number of example

    # Compute the cross-entropy cost
    ### START CODE HERE ### (= 2 lines of code)
    logprobs = np.dot(Y, np.log(A2).T) + np.dot(1 - Y, np.log(1 - A2).T)
    cost = np.float64(-logprobs / m)
    ### END CODE HERE ###

    cost = np.squeeze(cost)      # makes sure cost is the dimension we expect.

```

```

        # E.g., turns [[17]] into 17
    assert(isinstance(cost, float))

    return cost

A2, Y_assess, parameters = compute_cost_test_case()

print("cost = " + str(compute_cost(A2, Y_assess, parameters)))

cost = 0.6929198937761264

# GRADED FUNCTION: backward_propagation

def backward_propagation(parameters, cache, X, Y):
    """
    Implement the backward propagation using the instructions above.

    Arguments:
    parameters -- python dictionary containing our parameters
    cache -- a dictionary containing "Z1", "A1", "Z2" and "A2".
    X -- input data of shape (2, number of examples)
    Y -- "true" labels vector of shape (1, number of examples)

    Returns:
    grads -- python dictionary containing your gradients with respect to different parameters
    """
    m = X.shape[1]

    # First, retrieve W1 and W2 from the dictionary "parameters".
    ### START CODE HERE ### (≈ 2 lines of code)
    W1 = parameters['W1']
    W2 = parameters['W2']
    ### END CODE HERE ###

    # Retrieve also A1 and A2 from dictionary "cache".
    ### START CODE HERE ### (≈ 2 lines of code)
    A1 = cache['A1']
    A2 = cache['A2']
    ### END CODE HERE ###

    # Backward propagation: calculate dW1, db1, dW2, db2.
    ### START CODE HERE ### (≈ 6 lines of code, corresponding to 6 equations on slide above)
    dZ2 = A2 - Y
    dW2 = np.dot(dZ2, A1.T) / m
    db2 = np.sum(dZ2, axis=1, keepdims=True) / m
    dZ1 = np.dot(W2.T, dZ2) * (1 - np.power(A1, 2))
    dW1 = np.dot(dZ1, X.T) / m
    db1 = np.sum(dZ1, axis=1, keepdims=True) / m
    ### END CODE HERE ###

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

    return grads

parameters, cache, X_assess, Y_assess = backward_propagation_test_case()

grads = backward_propagation(parameters, cache, X_assess, Y_assess)
print ("dW1 = " + str(grads["dW1"]))
print ("db1 = " + str(grads["db1"]))
print ("dW2 = " + str(grads["dW2"]))
print ("db2 = " + str(grads["db2"]))

dW1 = [[ 0.01018708 -0.00708701]
 [ 0.00873447 -0.0060768 ]
 [-0.00530847  0.00369379]
 [-0.02206365  0.01535126]]
db1 = [[-0.00069728]
 [-0.00060606]
 [ 0.000364 ]
 [ 0.00151207]]
dW2 = [[ 0.00363613  0.03153604  0.01162914 -0.01318316]]
db2 = [[0.06589489]]

# GRADED FUNCTION: update_parameters

def update_parameters(parameters, grads, learning_rate = 1.2):
    """

```

Updates parameters using the gradient descent update rule given above

Arguments:

parameters -- python dictionary containing your parameters  
grads -- python dictionary containing your gradients

Returns:

parameters -- python dictionary containing your updated parameters  
"""

# Retrieve each parameter from the dictionary "parameters"  
### START CODE HERE ### (= 4 lines of code)

```
W1 = parameters['W1']  
b1 = parameters['b1']  
W2 = parameters['W2']  
b2 = parameters['b2']  
### END CODE HERE ###
```

# Retrieve each gradient from the dictionary "grads"  
### START CODE HERE ### (= 4 lines of code)

```
dW1 = grads['dW1']  
db1 = grads['db1']  
dW2 = grads['dW2']  
db2 = grads['db2']  
## END CODE HERE ###
```

# Update rule for each parameter  
### START CODE HERE ### (= 4 lines of code)

```
W1 = W1 - learning_rate * dW1  
b1 = b1 - learning_rate * db1  
W2 = W2 - learning_rate * dW2  
b2 = b2 - learning_rate * db2  
### END CODE HERE ###
```

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

return parameters

```
parameters, grads = update_parameters_test_case()  
parameters = update_parameters(parameters, grads)
```

```
print("W1 = " + str(parameters["W1"]))  
print("b1 = " + str(parameters["b1"]))  
print("W2 = " + str(parameters["W2"]))  
print("b2 = " + str(parameters["b2"]))
```

```
W1 = [[-0.00643025  0.01936718]  
      [-0.02410458  0.03978052]  
      [-0.01653973 -0.02096177]  
      [ 0.01046864 -0.05990141]]  
b1 = [[-1.02420756e-06]  
      [ 1.27373948e-05]  
      [ 8.32996807e-07]  
      [-3.20136836e-06]]  
W2 = [[-0.01041081 -0.04463285  0.01758031  0.04747113]]  
b2 = [[0.00010457]]
```

# GRADED FUNCTION: nn\_model

```
def nn_model(X, Y, n_h, num_iterations = 10000, print_cost=False):  
    """
```

Arguments:

X -- dataset of shape (2, number of examples)  
Y -- labels of shape (1, number of examples)  
n\_h -- size of the hidden layer  
num\_iterations -- Number of iterations in gradient descent loop  
print\_cost -- if True, print the cost every 1000 iterations

Returns:

parameters -- parameters learnt by the model. They can then be used to predict.  
"""

```
np.random.seed(3)  
n_x = layer_sizes(X, Y)[0]  
n_y = layer_sizes(X, Y)[2]
```

# Initialize parameters, then retrieve W1, b1, W2, b2. Inputs: "n\_x, n\_h, n\_y". Outputs = "W1, b1, W2, b2, parameters".

```
### START CODE HERE ### (= 5 lines of code)  
parameters = initialize_parameters(n_x, n_h, n_y)  
W1 = parameters['W1']  
b1 = parameters['b1']
```

```

W2 = parameters['W2']
b2 = parameters['b2']
### END CODE HERE ###

# Loop (gradient descent)
for i in range(0, num_iterations):

    ### START CODE HERE ### (≈ 4 lines of code)
    # Forward propagation. Inputs: "X, parameters". Outputs: "A2, cache".
    A2, cache = forward_propagation(X, parameters)

    # Cost function. Inputs: "A2, Y, parameters". Outputs: "cost".
    cost = compute_cost(A2, Y, parameters)

    # Backpropagation. Inputs: "parameters, cache, X, Y". Outputs: "grads".
    grads = backward_propagation(parameters, cache, X, Y)

    # Gradient descent parameter update. Inputs: "parameters, grads". Outputs: "parameters".
    parameters = update_parameters(parameters, grads)

    ### END CODE HERE ###

    # Print the cost every 1000 iterations
    if print_cost and i % 1000 == 0:
        print ("Cost after iteration %i: %f" %(i, cost))

return parameters

X_assess, Y_assess = nn_model_test_case()

parameters = nn_model(X_assess, Y_assess, 4, num_iterations=10000, print_cost=False)
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))

<ipython-input-14-ccd59e612351>:20: RuntimeWarning: divide by zero encountered in log
logprobs = np.dot(Y, np.log(A2).T) + np.dot(1 - Y, np.log(1 - A2).T)
/content/planar_utils.py:34: RuntimeWarning: overflow encountered in exp
s = 1/(1+np.exp(-x))
W1 = [[-4.18497132  5.33206757]
 [-7.53803917  1.20755745]
 [-4.19300026  5.32616513]
 [ 7.53798166 -1.20759037]]
b1 = [[ 2.32932729]
 [ 3.81001689]
 [ 2.33008569]
 [-3.81011712]]
W2 = [[-6033.82357224 -6008.14296509 -6033.08780459  6008.07954278]]
b2 = [[-52.67922744]]

# GRADED FUNCTION: predict

def predict(parameters, X):
    """
    Using the learned parameters, predicts a class for each example in X

    Arguments:
    parameters -- python dictionary containing your parameters
    X -- input data of size (n_x, m)

    Returns
    predictions -- vector of predictions of our model (red: 0 / blue: 1)
    """

    # Computes probabilities using forward propagation, and classifies to 0/1 using 0.5 as the threshold.
    ### START CODE HERE ### (≈ 2 lines of code)
    A2, cache = forward_propagation(X, parameters)
    predictions = (A2 > .5)
    ### END CODE HERE ###

    return predictions

parameters, X_assess = predict_test_case()

predictions = predict(parameters, X_assess)
print("predictions mean = " + str(np.mean(predictions)))

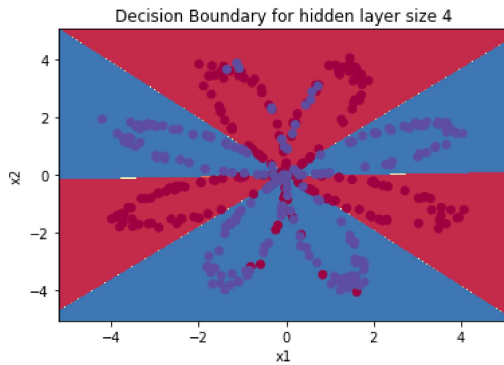
predictions mean = 0.6666666666666666

```

```
# Build a model with a n_h-dimensional hidden layer
parameters = nn_model(X, Y, n_h = 5, num_iterations = 10000, print_cost=True)
```

```
# Plot the decision boundary
plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
plt.title("Decision Boundary for hidden layer size " + str(4))
```

```
Cost after iteration 0: 0.693252
Cost after iteration 1000: 0.283771
Cost after iteration 2000: 0.270689
Cost after iteration 3000: 0.263510
Cost after iteration 4000: 0.258455
Cost after iteration 5000: 0.250679
Cost after iteration 6000: 0.224348
Cost after iteration 7000: 0.219895
Cost after iteration 8000: 0.180043
Cost after iteration 9000: 0.175090
Text(0.5, 1.0, 'Decision Boundary for hidden layer size 4')
```

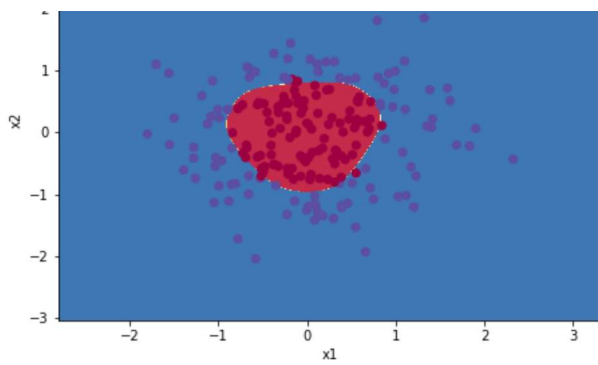


```
# Print accuracy
predictions = predict(parameters, X)
print ('Accuracy: %d' % float((np.dot(Y,predictions.T) + np.dot(1-Y,1-predictions.T))/float(Y.size)*100) + '%')
```

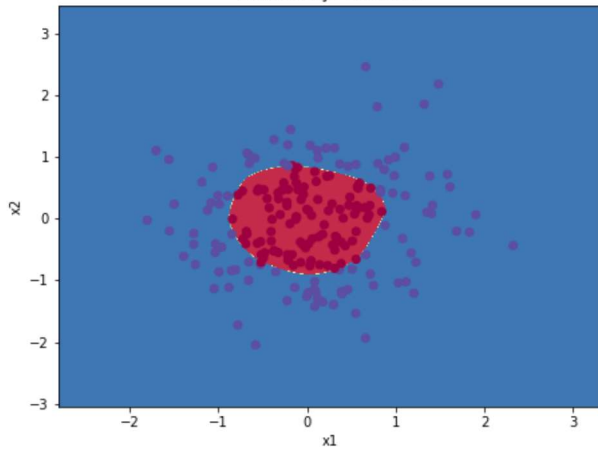
```
Accuracy: 91%
```

```
# This may take about 2 minutes to run
```

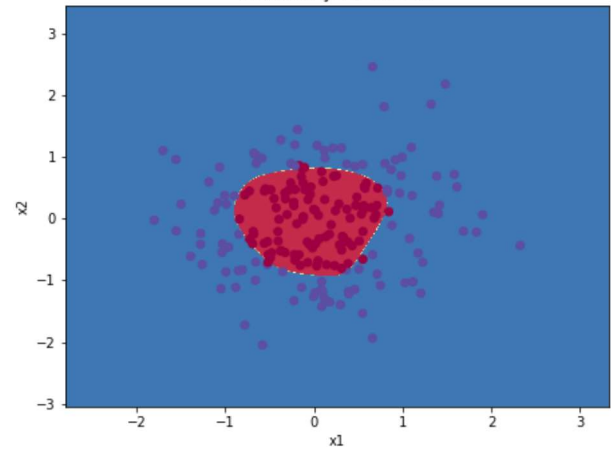
```
plt.figure(figsize=(16, 32))
hidden_layer_sizes = [1, 2, 3, 4, 5, 20, 40]
for i, n_h in enumerate(hidden_layer_sizes):
    plt.subplot(5, 2, i+1)
    plt.title('Hidden Layer of size %d' % n_h)
    parameters = nn_model(X, Y, n_h, num_iterations = 5000)
    plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
    predictions = predict(parameters, X)
    accuracy = float((np.dot(Y,predictions.T) + np.dot(1-Y,1-predictions.T))/float(Y.size)*100)
    print ("Accuracy for {} hidden units: {} %".format(n_h, accuracy))
```



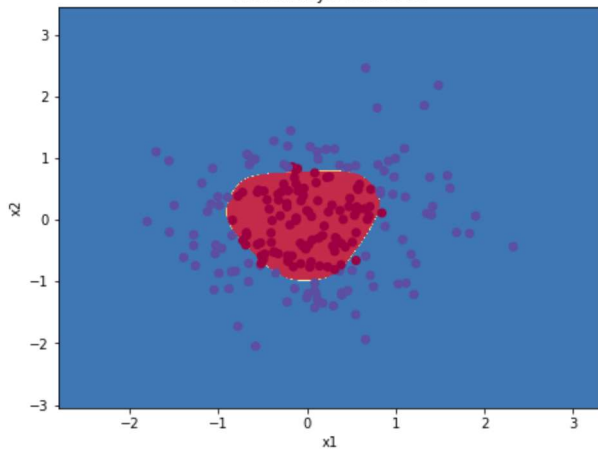
Hidden Layer of size 5



Hidden Layer of size 20



Hidden Layer of size 40



```
# Datasets
noisy_circles, noisy_moons, blobs, gaussian_quantiles, no_structure = load_extra_datasets()
```

```
datasets = {"noisy_circles": noisy_circles,
            "noisy_moons": noisy_moons,
            "blobs": blobs,
            "gaussian_quantiles": gaussian_quantiles}
```

```
### START CODE HERE ### (choose your dataset)
dataset = "gaussian_quantiles"
### END CODE HERE ###
```

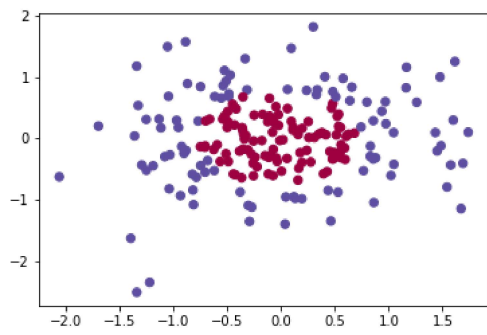
```
X, Y = datasets[dataset]
X, Y = X.T, Y.reshape(1, Y.shape[0])
```

```
# make blobs binary
if dataset == "blobs":
    Y = Y%2
```

```
# Visualize the data
plt.scatter(X[0, :], X[1, :], c=Y, s=40, cmap=plt.cm.Spectral);
```







✓ 0s completed at 12:34 PM

