02_planar_data_classification_with_one_hidden_layer

April 7, 2025

1 Planar data classification with one hidden layer

In this exercise, we will build our first neural network which will have one hidden layer. We'll notice a big difference between this model and the one we implemented previously using logistic regression.

By the end of this assignment, you'll be able to:

- Implement a 2-class classification neural network with a single hidden layer
- Use units with a non-linear activation function, such as tanh
- Compute the cross entropy loss
- Implement forward and backward propagation

1.1 Table of Contents

- 7- Performance on other datasets

1 - Packages

First import all the packages that we will need during this assignment.

- numpy is the fundamental package for scientific computing with Python.
- sklearn provides simple and efficient tools for data mining and data analysis.
- matplotlib is a library for plotting graphs in Python.
- testCases provides some test examples to assess the correctness of your functions
- planar_utils provide various useful functions used in this assignment

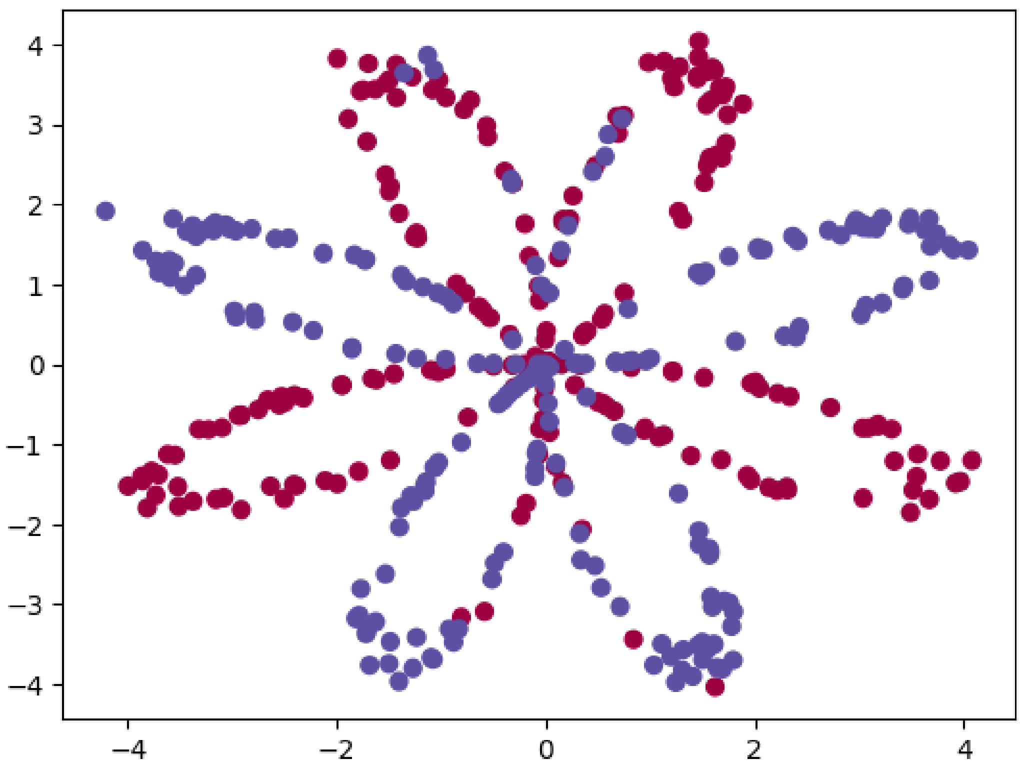
[1]:
```
# Package imports
import numpy as np
import copy
import matplotlib.pyplot as plt
from testCases_v2 import *
from public_tests import *
import sklearn
import sklearn.datasets
import sklearn.linear_model
from planar_utils import plot_decision_boundary, sigmoid, load_planar_dataset,␣
↪load_extra_datasets

%matplotlib inline

%load_ext autoreload
%autoreload 2
```

2 - Load the Dataset

[2]:
```
X, Y = load_planar_dataset()
```

Visualize the dataset using matplotlib. The data looks like a "flower" with some red (label y=0) and some blue (y=1) points. Our goal is to build a model to fit this data. In other words, we want the classifier to define regions as either red or blue.

[3]:
```
# Visualize the data:
plt.scatter(X[0, :], X[1, :], c=Y, s=40, cmap=plt.cm.Spectral);
```



We have the following:

```
- a numpy-array (matrix) X that contains the features (x1, x2)
- a numpy-array (vector) Y that contains the labels (red:0, blue:1).
```

First, let's get a better sense of what our data is like.

Exercise 1

How many training examples do we have? In addition, what is the `shape` of the variables `X` and `Y`?

Hint: How do we get the shape of a numpy array? (help)

[4]:
```
# The structure of your code in this cell should be as follows:

# shape_X = ...
# shape_Y = ...
# training set size
# m = ...

# YOUR CODE STARTS HERE

shape_X = X.shape       # X is of shape (num_px * num_px * 3, m)
shape_Y = Y.shape       # Y is of shape (1, m)
```

```
m = X.shape[1]          # Number of training examples is the number of columns␣
 ↪in X




# YOUR CODE ENDS 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!
```

Expected Output:

shape of X

(2, 400)

shape of Y

(1, 400)

m

400

3 - Simple Logistic Regression

Before building a full neural network, let's check how logistic regression performs on this problem. We will use sklearn's built-in functions for this. Run the code below to train a logistic regression classifier on the dataset.

[5]:
```
# Train the logistic regression classifier
clf = sklearn.linear_model.LogisticRegressionCV();
clf.fit(X.T, Y.T);
```

```
/home/codespace/.local/lib/python3.12/site-
packages/sklearn/utils/validation.py:1408: DataConversionWarning: A column-
vector y was passed when a 1d array was expected. Please change the shape of y
to (n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
```

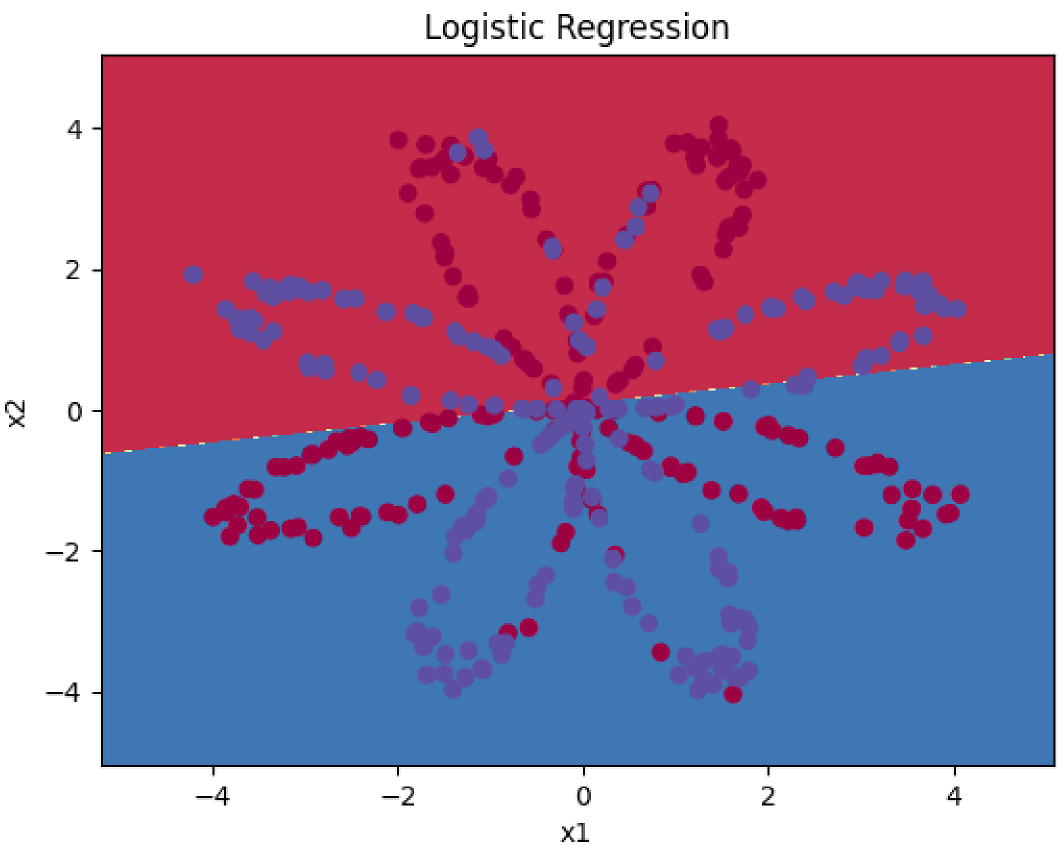
Let's take a look at the decision boundary of this model. Run the code below.

[6]:
```
# 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)")
```

```
/tmp/ipykernel_13494/4242423965.py:7: DeprecationWarning: Conversion of an array
with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you
extract a single element from your array before performing this operation.
(Deprecated NumPy 1.25.)
  print ('Accuracy of logistic regression: %d ' %
float((np.dot(Y,LR_predictions) +
np.dot(1-Y,1-LR_predictions))/float(Y.size)*100) +
```

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



Expected Output:

Accuracy

47%

Interpretation: The dataset is not linearly separable, so logistic regression doesn't perform well. Hopefully a neural network will do better…

4 - Neural Network model

Logistic regression didn't work well on the flower dataset. So, we're going to train a Neural Network with a single hidden layer and see how that handles the same problem.

The model:

Mathematically:

For one example $x^{(i)}$:

$$z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1]} \tag{1}$$

$$a^{[1](i)} = \tanh(z^{[1](i)}) \tag{2}$$

$$z^{[2](i)} = W^{[2]} a^{[1](i)} + b^{[2]} \tag{3}$$

$$\hat{y}^{(i)} = a^{[2](i)} = \sigma(z^{[2](i)}) \tag{4}$$

$$y^{(i)}_{prediction} = \begin{cases} 1 & \text{if } a^{[2](i)} > 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Given the predictions on all the examples, you can also compute the cost J as follows:

$$J = -\frac{1}{m} \sum_{i=0}^{m} \left(y^{(i)} \log\left(a^{[2](i)}\right) + (1-y^{(i)}) \log\left(1-a^{[2](i)}\right) \right) \tag{6}$$

Reminder: The general methodology to build a Neural Network is to:

```
1. Define the neural network structure ( # of input units,  # of hidden units, etc).
2. Initialize the model's parameters
3. Loop:
    - Implement forward propagation
    - Compute loss
    - Implement backward propagation to get the gradients
    - Update parameters (gradient descent)
```

In practice, we'll often build helper functions to compute steps 1-3, test each function to make sure each one is working properly, and then merge them into one function called `nn_model()`. Once we've built `nn_model()` and learned the right parameters, we can make predictions on new data.

4.1 - Defining the neural network structure

Exercise 2 - layer_sizes

Define three variables: - n_x: the size of the input layer - n_h: the size of the hidden layer (**set this to 4, only for this Exercise 2**) - n_y: the size of the output layer

Hint: Use shapes of X and Y to find n_x and n_y. Also, hard code the hidden layer size to be 4.

[7]:
```
# 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
    """


    # YOUR CODE STARTS HERE



    n_x = X.shape[0]  # Number of features in the input layer
    n_h = 4           # You can choose the size of the hidden layer (here,␣
↪arbitrarily set to 4)
    n_y = Y.shape[0]  # Number of outputs

    # YOUR CODE ENDS HERE
    return (n_x, n_h, n_y)
```

[8]:
```
t_X, t_Y = layer_sizes_test_case()
(n_x, n_h, n_y) = layer_sizes(t_X, t_Y)
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))

layer_sizes_test(layer_sizes)
```

```
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
All tests passed!
```

Expected output

```
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
All tests passed!
 All tests passed.
```

4.2 - Initialize the model's parameters

Exercise 3 - initialize_parameters

Implement the function `initialize_parameters()`.

Instructions: - Make sure the parameters' sizes are right. Refer to the neural network figure above if needed. - Initialize the weights matrices with random values. - Use: `np.random.randn(a,b) * 0.01` to randomly initialize a matrix of shape (a,b). - Initialize the bias vectors as zeros. - Use: `np.zeros((a,b))` to initialize a matrix of shape (a,b) with zeros.

[9]:
```
# 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)
    """

    # YOUR CODE STARTS HERE
    W1 = np.random.randn(n_h, n_x) * 0.01
    b1 = np.zeros((n_h, 1))
    W2 = np.random.randn(n_y, n_h) * 0.01
    b2 = np.zeros((n_y, 1))




    # YOUR CODE ENDS HERE

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

    return parameters
```

[10]:
```
np.random.seed(2)
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"]))

initialize_parameters_test(initialize_parameters)
```

```
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.]]
All tests passed!
```

Expected output

```
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.]]
All tests passed!
```

4.3 - The Loop

Exercise 4 - forward_propagation

Implement `forward_propagation()` using the following equations:

$$Z^{[1]} = W^{[1]}X + b^{[1]} \tag{1}$$

$$A^{[1]} = \tanh(Z^{[1]}) \tag{2}$$

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]} \tag{3}$$

$$\hat{Y} = A^{[2]} = \sigma(Z^{[2]}) \tag{4}$$

Instructions:

- Check the mathematical representation of the classifier in the figure above.
- Use the function `sigmoid()`. It's built into this notebook, (i.e., imported from the planar_utils.py module).
- Use the function `np.tanh()` from the numpy library.
- Implement using these steps:

1. Retrieve each parameter from the dictionary "parameters" (which is the output of `initialize_parameters()` by using `parameters[".."]`.
2. Implement Forward Propagation. Compute $Z^{[1]}, A^{[1]}, Z^{[2]}$ and $A^{[2]}$ (the vector of all the predictions on all the examples in the training set).

- Values needed in the backpropagation are stored in "cache". The cache will be given as an input to the backpropagation function.

[11]:
```python
# 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".  These are the␣
 ↪W and b parameters from each
    # layer of our neural network that will allow forward prop from X to A2.

    # YOUR CODE STARTS HERE



    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]


    # YOUR CODE ENDS HERE

    # Implement Forward Propagation to calculate A2 (probabilities).  These are␣
 ↪the calculations of the nodes in
    # layers 1 and 2 of our neural network yielding our Z and A values for each␣
 ↪of the layers, ending at A2.

    # YOUR CODE STARTS HERE


    # First layer computations
    Z1 = np.dot(W1, X) + b1        # Linear step for layer 1
    A1 = np.tanh(Z1)               # Activation using tanh for layer 1
```

```
    # Second layer computations
    Z2 = np.dot(W2, A1) + b2      # Linear step for layer 2
    A2 = sigmoid(Z2)              # Activation using sigmoid for output layer



    # YOUR CODE ENDS HERE

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

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

    return A2, cache
```

[12]:
```
t_X, parameters = forward_propagation_test_case()
A2, cache = forward_propagation(t_X, parameters)
print("A2 = " + str(A2))

forward_propagation_test(forward_propagation)
```

```
A2 = [[0.21292656 0.21274673 0.21295976]]
All tests passed!
```

Expected output

```
A2 = [[0.21292656 0.21274673 0.21295976]]
All tests passed!
 All tests passed.
```

4.4 - Compute the Cost

Now that we've computed $A^{[2]}$ (in the Python variable "`A2`"), which contains $a^{[2](i)}$ for all examples, we can compute the cost function as follows:

$$J = -\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} \log\left(a^{[2](i)}\right) + (1-y^{(i)}) \log\left(1-a^{[2](i)}\right) \right) \tag{13}$$

Exercise 5 - compute_cost

Implement `compute_cost()` to compute the value of the cost J.

Instructions: - There are many ways to implement the loss. This is one way to implement one part of the equation without for loops: $-\sum\limits_{i=1}^{m} y^{(i)}\log(a^{[2](i)})$:

```
logprobs = np.multiply(np.log(A2),Y)
cost = - np.sum(logprobs)
```

- Use that to build the whole expression of the cost function.

Notes:

- Use either 1. `np.multiply()` and then `np.sum()` or 2. `np.dot()`
- If you use `np.multiply` followed by `np.sum` the end result will be a type `float`, whereas if you use `np.dot`, the result will be a 2D numpy array.

- You can use `np.squeeze()` to remove redundant dimensions (in the case of single float, this will be reduced to a zero-dimension array).
- You can also cast the array as a type `float` using `float()`.

[13]:
```python
# GRADED FUNCTION: compute_cost

def compute_cost(A2, Y):
    """
    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)

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

    """

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

    # Compute the cost
    # YOUR CODE STARTS HERE

    cost = -(1/m) * np.sum(Y * np.log(A2) + (1 - Y) * np.log(1 - A2))




    # YOUR CODE ENDS HERE

    cost = float(np.squeeze(cost))  # makes sure cost is the dimension we␣
 ↪expect.
                                    # E.g., turns [[17]] into 17

    return cost
```

[14]:
```python
A2, t_Y = compute_cost_test_case()
cost = compute_cost(A2, t_Y)
```

```
print("cost = " + str(compute_cost(A2, t_Y)))

compute_cost_test(compute_cost)
```

```
cost = 0.6930587610394646
All tests passed!
```

Expected output

```
cost = 0.6930587610394646
All tests passed!
 All tests passed.
```

4.5 - Implement Backpropagation

Using the cache computed during forward propagation, we can now implement backward propagation.

Exercise 6 - backward_propagation

Implement the function `backward_propagation()`.

Instructions: Backpropagation is usually the hardest (most mathematical) part in deep learning. Below are the formulas for backpropagation. Use the six equations on the right side since we are building a vectorized implementation.

Figure 1: Backpropagation. Use the six equations on the right.

- Tips:
  - To compute dZ1 we'll need to compute $g^{[1]'}(Z^{[1]})$. Since $g^{[1]}(.)$ is the tanh activation function, if $a = g^{[1]}(z)$ then $g^{[1]'}(z) = 1 - a^2$. So we can compute $g^{[1]'}(Z^{[1]})$ using `(1 - np.power(A1, 2))`.

[15]:
```
# 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".
```

```
    # YOUR CODE STARTS HERE


    W1 = parameters["W1"]
    W2 = parameters["W2"]


    # YOUR CODE ENDS HERE

    # Next, retrieve A1 and A2 from dictionary "cache".
    # YOUR CODE STARTS HERE


    A1 = cache["A1"]
    A2 = cache["A2"]


    # YOUR CODE ENDS HERE

    # Finally, implment backpropagation: calculate dW1, db1, dW2, db2.
    # YOUR CODE STARTS HERE

    dZ2 = A2 - Y                                        #derivative for output␣
↪layer
    dW2 = (1/m) * np.dot(dZ2, A1.T)                     #derivative for W2
    db2 = (1/m) * np.sum(dZ2, axis=1, keepdims=True)   #derivative for b2

    dA1 = np.dot(W2.T, dZ2)                             #backprop into hidden␣
↪layer
    dZ1 = dA1 * (1 - np.power(A1, 2))                   #derivative for tanh␣
↪activation (1-tanh^2)
    dW1 = (1/m) * np.dot(dZ1, X.T)                      #derivative for W1
    db1 = (1/m) * np.sum(dZ1, axis=1, keepdims=True)   #derivative for b1



    # YOUR CODE ENDS HERE

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

    return grads
```

[16]:
```
parameters, cache, t_X, t_Y = backward_propagation_test_case()

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

backward_propagation_test(backward_propagation)
```

```
dW1 = [[ 0.00301023 -0.00747267]
 [ 0.00257968 -0.00641288]
 [-0.00156892  0.003893  ]
 [-0.00652037  0.01618243]]
db1 = [[ 0.00176201]
 [ 0.00150995]
 [-0.00091736]
 [-0.00381422]]
dW2 = [[ 0.00078841  0.01765429 -0.00084166 -0.01022527]]
db2 = [[-0.16655712]]
All tests passed!
```

Expected output

```
dW1 = [[ 0.00301023 -0.00747267]
 [ 0.00257968 -0.00641288]
 [-0.00156892  0.003893  ]
 [-0.00652037  0.01618243]]
db1 = [[ 0.00176201]
 [ 0.00150995]
 [-0.00091736]
 [-0.00381422]]
dW2 = [[ 0.00078841  0.01765429 -0.00084166 -0.01022527]]
db2 = [[-0.16655712]]
All tests passed!
 All tests passed.
```

4.6 - Update Parameters

Exercise 7 - update_parameters

Implement the update rule. Use gradient descent. We have to use (dW1, db1, dW2, db2) in order to update (W1, b1, W2, b2).

General gradient descent rule: $\theta = \theta - \alpha \frac{\partial J}{\partial \theta}$ where α is the learning rate and θ represents a parameter.

Figure 2: The gradient descent algorithm with a good learning rate (converging) and a bad learning rate (diverging). Images courtesy of Adam Harley.

Hint

- Use `copy.deepcopy(...)` when copying lists or dictionaries that are passed as parameters to functions. It avoids input parameters being modified within the function.

[17]:
```python
# 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 a copy of each parameter from the dictionary "parameters". Use␣
 ↪copy.deepcopy(...) for W1 and W2
    # YOUR CODE STARTS HERE

    W1 = copy.deepcopy(parameters["W1"])
    W2 = copy.deepcopy(parameters["W2"])
    b1 = copy.deepcopy(parameters["b1"])
    b2 = copy.deepcopy(parameters["b2"])



    # YOUR CODE ENDS HERE

    # Retrieve each gradient from the dictionary "grads"
    # YOUR CODE STARTS HERE


    dW1 = grads["dW1"]
    db1 = grads["db1"]
    dW2 = grads["dW2"]
    db2 = grads["db2"]


    # YOUR CODE ENDS HERE

    # Implement the gradient descent update for the parameters W and b␣
 ↪parameter for each of the 2 layers.
    # YOUR CODE STARTS HERE

    W1 = W1 - learning_rate * dW1
    b1 = b1 - learning_rate * db1
```

```python
    W2 = W2 - learning_rate * dW2
    b2 = b2 - learning_rate * db2



    # YOUR CODE ENDS HERE

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

    return parameters
```

[18]:
```python
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"]))

update_parameters_test(update_parameters)
```

```
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]]
All tests passed!
```

Expected output

```
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]]
```

```
All tests passed!
 All tests passed.
```

4.7 - Integration

Integrate the functions we've built above in `nn_model()`

Exercise 8 - nn_model

Build the neural network model in `nn_model()`.

Instructions: The neural network model has to use the previous functions in the right order.

[19]:
```python
# 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 the parameters.
    # YOUR CODE STARTS HERE
    parameters = initialize_parameters(n_x, n_h, n_y)




    # YOUR CODE ENDS HERE


    # Loop (gradient descent)

    for i in range(0, num_iterations):


        # Implement the following steps by calling the functions we wrote above:
 ↪
```

```
        # 1. Forward propagation. Inputs: "X, parameters". Outputs: "A2, cache".
↪
        # 2. Cost calculation. Inputs: "A2, Y". Outputs: "cost".
        # 3. Backpropagation. Inputs: "parameters, cache, X, Y". Outputs:␣
↪"grads".
        # 4. Gradient descent parameter update. Inputs: "parameters, grads".␣
↪Outputs: "parameters".

        # YOUR CODE STARTS HERE


        A2, cache = forward_propagation(X, parameters)
        cost = compute_cost(A2, Y)
        grads = backward_propagation(parameters, cache, X, Y)
        parameters = update_parameters(parameters, grads)


        # YOUR CODE ENDS HERE

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

    return parameters
```

[20]:
```
nn_model_test(nn_model)
```

```
Cost after iteration 0: 0.693086
Cost after iteration 1000: 0.000220
Cost after iteration 2000: 0.000108
Cost after iteration 3000: 0.000072
Cost after iteration 4000: 0.000054
Cost after iteration 5000: 0.000043
Cost after iteration 6000: 0.000036
Cost after iteration 7000: 0.000030
Cost after iteration 8000: 0.000027
Cost after iteration 9000: 0.000024
W1 = [[ 0.71392202  1.31281102]
 [-0.76411243 -1.41967065]
 [-0.75040545 -1.38857337]
 [ 0.56495575  1.04857776]]
b1 = [[-0.0073536 ]
 [ 0.01534663]
 [ 0.01262938]
 [ 0.00218135]]
W2 = [[ 2.82545815 -3.3063945  -3.16116615  1.8549574 ]]
b2 = [[0.00393452]]
```

```
All tests passed!
```

Expected output

```
Cost after iteration 0: 0.693198
Cost after iteration 1000: 0.000219
Cost after iteration 2000: 0.000108
...
Cost after iteration 8000: 0.000027
Cost after iteration 9000: 0.000024
W1 = [[ 0.56305445 -1.03925886]
 [ 0.7345426  -1.36286875]
 [-0.72533346  1.33753027]
 [ 0.74757629 -1.38274074]]
b1 = [[-0.22240654]
 [-0.34662093]
 [ 0.33663708]
 [-0.35296113]]
W2 = [[ 1.82196893  3.09657075 -2.98193564  3.19946508]]
b2 = [[0.21344644]]
All tests passed!
 All tests passed.
```

5 - Test the Model

5.1 - Predict

Exercise 9 - predict

Predict with the model by building `predict()`. Use forward propagation to predict results.

Reminder: predictions $= y_{prediction} = \mathbb{1}\text{activation} > 0.5 = \begin{cases} 1 & \text{if } activation > 0.5 \\ 0 & \text{otherwise} \end{cases}$

As an example, if you would like to set the entries of a matrix X to 0 and 1 based on a threshold you would do: `X_new = (X > threshold)`

[21]:
```python
# 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)
    """
```

```
    # Compute probabilities using forward propagation, and classifies to 0/1␣
↪using 0.5 as the threshold.

    # YOUR CODE STARTS HERE
    #Compute probabilities using forward propagation
    A2, _ = forward_propagation(X, parameters)
    #Convert probabilities A2 to actual predictions using 0.5 as the threshold
    predictions = (A2 > 0.5).astype(int)

    # YOUR CODE ENDS HERE

    return predictions
```

[22]:
```
parameters, t_X = predict_test_case()

predictions = predict(parameters, t_X)
print("Predictions: " + str(predictions))

predict_test(predict)
```

```
Predictions: [[1 0 1]]
All tests passed!
```

Expected output

```
Predictions: [[ True False  True]]
All tests passed!
 All tests passed.
```

5.2 - Test the Model on the Planar Dataset

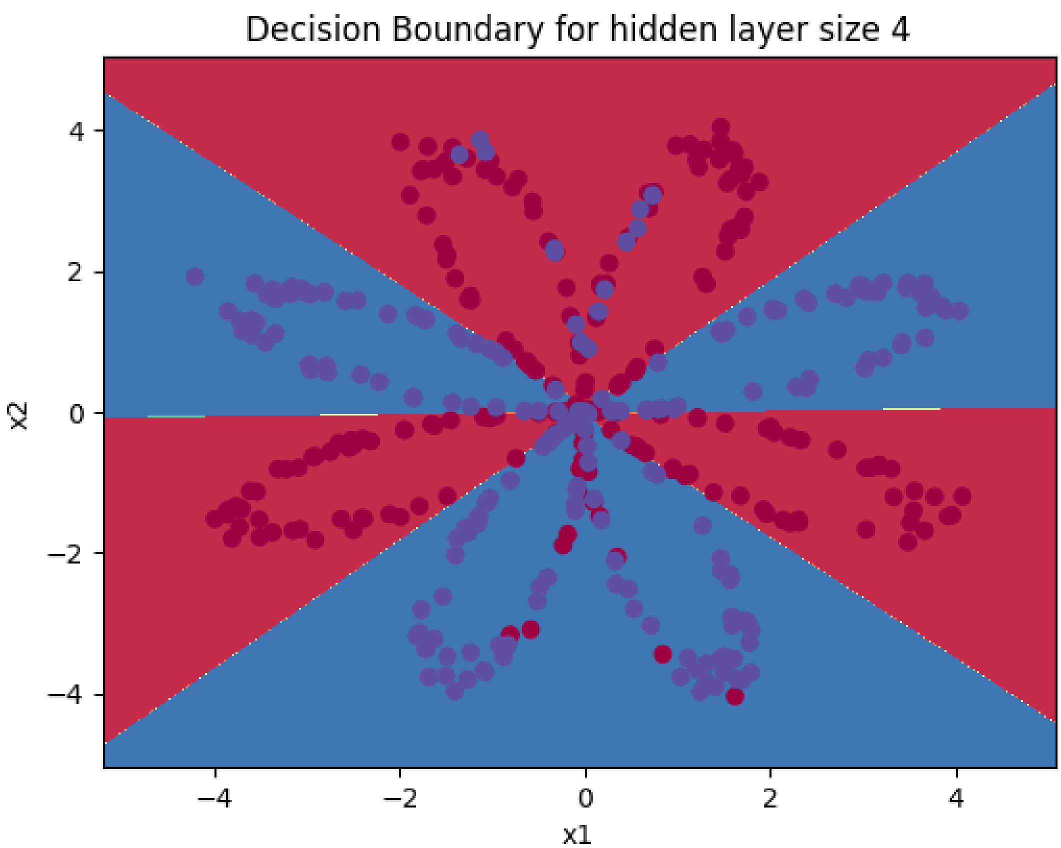
It's time to run the model and see how it performs on a planar dataset. Run the following code to test your model with a single hidden layer of n_h hidden units!

[23]:
```
# Build a model with a n_h-dimensional hidden layer
parameters = nn_model(X, Y, n_h = 4, 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: 0.693162
Cost after iteration 1000: 0.258625
Cost after iteration 2000: 0.239334
Cost after iteration 3000: 0.230802
Cost after iteration 4000: 0.225528
Cost after iteration 5000: 0.221845
Cost after iteration 6000: 0.219094
Cost after iteration 7000: 0.220668
```

```
Cost after iteration 8000: 0.219411
Cost after iteration 9000: 0.218486
```

[23]: Text(0.5, 1.0, 'Decision Boundary for hidden layer size 4')



[24]:
```
# 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: 90%
```

```
/tmp/ipykernel_13494/1304927518.py:3: DeprecationWarning: Conversion of an array
with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you
extract a single element from your array before performing this operation.
(Deprecated NumPy 1.25.)
  print ('Accuracy: %d' % float((np.dot(Y, predictions.T) + np.dot(1 - Y, 1 -
predictions.T)) / float(Y.size) * 100) + '%')
```

Expected Output:

Accuracy

90%

Accuracy is really high compared to Logistic Regression. The model has learned the patterns of the flower's petals! Unlike logistic regression, neural networks are able to learn even highly non-linear decision boundaries.

1.1.1 Congrats - you're done!

Here's a quick recap of all you just accomplished:

- Built a complete 2-class classification neural network with a hidden layer
- Made good use of a non-linear unit
- Computed the loss
- Implemented forward and backward propagation

You've created a neural network that can learn patterns! Excellent work. Below is some code that shows some techniques for trying out different sizes for the hidden layer.

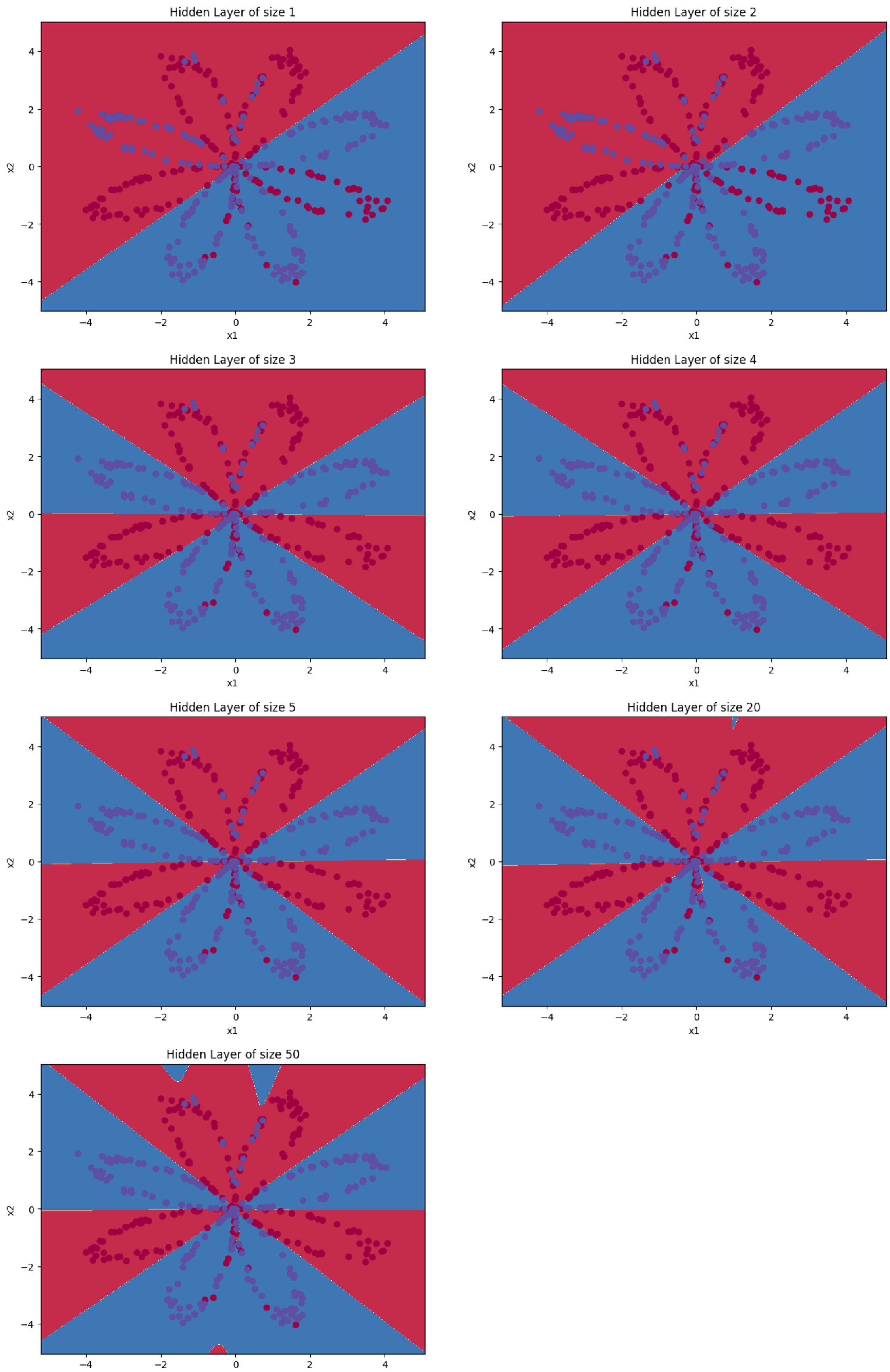
6 - Tuning hidden layer size (optional/ungraded exercise)

Run the following code (it may take 1-2 minutes). Then, observe different behaviors of the model for various hidden layer sizes.

[25]:
```
# This may take about 2 minutes to run

plt.figure(figsize=(16, 32))
hidden_layer_sizes = [1, 2, 3, 4, 5, 20, 50]
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))
```

```
/tmp/ipykernel_13494/164830050.py:11: DeprecationWarning: Conversion of an array
with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you
extract a single element from your array before performing this operation.
(Deprecated NumPy 1.25.)
  accuracy = float((np.dot(Y,predictions.T) + np.dot(1 - Y, 1 - predictions.T))
/ float(Y.size)*100)
```

```
Accuracy for 1 hidden units: 67.5 %
Accuracy for 2 hidden units: 67.25 %
Accuracy for 3 hidden units: 90.75 %
Accuracy for 4 hidden units: 90.5 %
Accuracy for 5 hidden units: 91.25 %
Accuracy for 20 hidden units: 90.75 %
Accuracy for 50 hidden units: 90.25 %
```



Hidden Layer of size 1
Hidden Layer of size 2
Hidden Layer of size 3
Hidden Layer of size 4
Hidden Layer of size 5
Hidden Layer of size 20
Hidden Layer of size 50
x1
x2

Interpretation: - The larger models (with more hidden units) are able to fit the training set better, until eventually the largest models overfit the data. - The best hidden layer size seems to be around n_h = 5. Indeed, a value around here seems to fits the data well without also incurring noticeable overfitting. - Later, we'll use regularization in neural networks which allows using large models (such as n_h = 50) without much overfitting.

References:

- http://cs231n.github.io/neural-networks-case-study/

[ ]:

```

```