Planar data classification with one hidden layer v5

July 20, 2019

1 Planar data classification with one hidden layer

Welcome to your week 3 programming assignment. It's time to build your first neural network, which will have a hidden layer. You will see a big difference between this model and the one you implemented using logistic regression.

You will learn how 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 1 - Packages

Let's first import all the packages that you 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

```
In [1]: # Package imports
        import numpy as np
        import matplotlib.pyplot as plt
        from testCases_v2 import *
        import sklearn
        import sklearn.datasets
        import sklearn.linear_model
        from planar_utils import plot_decision_boundary, sigmoid, load_planar_dataset, load_extr

        %matplotlib inline

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

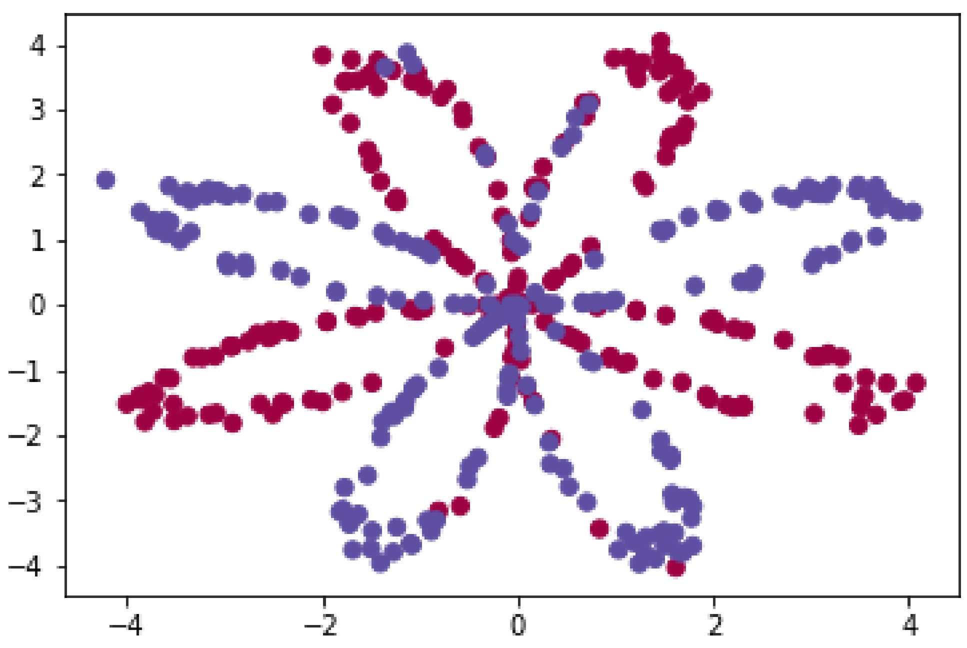
1.2 2 - Dataset

First, let's get the dataset you will work on. The following code will load a "flower" 2-class dataset into variables X and Y.

```
In [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. Your goal is to build a model to fit this data.

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



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

Lets first get a better sense of what our data is like.

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

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

```
In [4]: ### START CODE HERE ### ( 3 lines of code)
        shape_X = X.shape
        shape_Y = Y.shape
        m = X.shape[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!
```

Expected Output:
shape of X
(2, 400)
shape of Y
(1, 400)
m
400

1.3 3 - Simple Logistic Regression

Before building a full neural network, lets first see how logistic regression performs on this problem. You can use sklearn's built-in functions to do that. Run the code below to train a logistic regression classifier on the dataset.

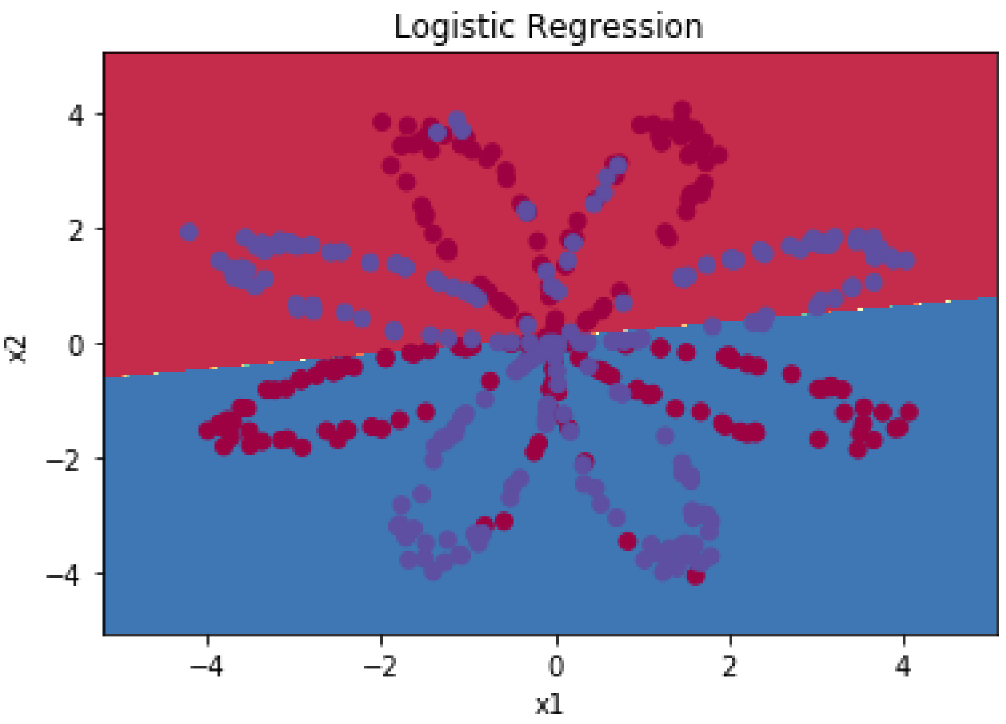
```
In [5]: # Train the logistic regression classifier
        clf = sklearn.linear_model.LogisticRegressionCV();
        clf.fit(X.T, Y.T);
```

You can now plot the decision boundary of these models. Run the code below.

```
In [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
               '% ' + "(percentage of correctly labelled datapoints)")
```

```
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. Let's try this now!

1.4 4 - Neural Network model

Logistic regression did not work well on the "flower dataset". You are going to train a Neural Network with a single hidden layer.

Here is our 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)

You often build helper functions to compute steps 1-3 and then merge them into one function we call `nn_model()`. Once you've built `nn_model()` and learnt the right parameters, you can make predictions on new data.

1.4.1 4.1 - Defining the neural network structure

Exercise: Define three variables: - n_x: the size of the input layer - n_h: the size of the hidden layer (set this to 4) - 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.

```
In [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
            """
            ### 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)
```

```
In [8]: 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
```

Expected Output (these are not the sizes you will use for your network, they are just used to assess the function you've just coded).

n_x
5

n_h
4
n_y
2

1.4.2 4.2 - Initialize the model's parameters

Exercise: Implement the function `initialize_parameters()`.

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

```
In [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)
            """

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

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

```
In [10]: 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.]]
```

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.]]

1.4.3 4.3 - The Loop

Question: Implement `forward_propagation()`.

Instructions: - Look above at the mathematical representation of your classifier. - You can use the function `sigmoid()`. It is built-in (imported) in the notebook. - You can use the function `np.tanh()`. It is part of the numpy library. - The steps you have to implement are: 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 your 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.

```
In [11]: # 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 initializatio

    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
```

```
In [12]: 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
```

```
0.262818640198 0.091999045227 -1.30766601287 0.212877681719
```

Expected Output:
0.262818640198 0.091999045227 -1.30766601287 0.212877681719

Now that you have computed $A^{[2]}$ (in the Python variable "A2"), which contains $a^{[2](i)}$ for every example, you can compute the cost function as follows:

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

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

Instructions: - There are many ways to implement the cross-entropy loss. To help you, we give you how we would have implemented $-\sum_{i=0}^{m} y^{(i)} \log(a^{[2](i)})$:

```
logprobs = np.multiply(np.log(A2),Y)
cost = - np.sum(logprobs)                # no need to use a for loop!
```

(you can use either `np.multiply()` and then `np.sum()` or directly `np.dot()`).

```
In [37]: # 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 = -(Y*np.log(A2) + (1-Y)*np.log(1-A2))
             cost = np.sum(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
```

```
In [38]: A2, Y_assess, parameters = compute_cost_test_case()

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

```
cost = 0.693058761039
```

Expected Output:

cost

0.693058761…

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

Question: Implement the function `backward_propagation()`.

Instructions: Backpropagation is usually the hardest (most mathematical) part in deep learning. To help you, here again is the slide from the lecture on backpropagation. You'll want to use the six equations on the right of this slide, since you are building a vectorized implementation.

- Tips:
  - To compute dZ1 you'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 you can compute $g^{[1]'}(Z^{[1]})$ using `(1 - np.power(A1, 2))`.

In [43]:
```
# 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 para
    """
    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 ab
    dZ2 = A2-Y #dZ2=(n_y, m)
    dW2 = (np.dot(dZ2, A1.T))/m  #dZ2=(n_y, m), A1=(n_h, m), W2=(n_y, n_h)
    db2 = np.sum(dZ2, axis=1, keepdims=True)/m
```