module-10-revised-programming-only

March 31, 2019

```
In [1]: %matplotlib inline
```

1 Module 10 - Programming Assignment

1.1 Directions

There are general instructions on Blackboard and in the Syllabus for Programming Assignments. This Notebook also has instructions specific to this assignment. Read all the instructions carefully and make sure you understand them. Please ask questions on the discussion boards or email me at `EN605.645@gmail.com` if you do not understand something.

You must follow the directions *exactly* or you will get a 0 on the assignment.

You must submit your assignment as `<jhed_id>.py`.

2 The Problem

When we last left our agent in Module 4, it was wandering around a world filled with plains, forests, swamps, hills and mountains. This presupposes a map with known terrain:

```
......
...**.
...***
..^...
..~^..
```

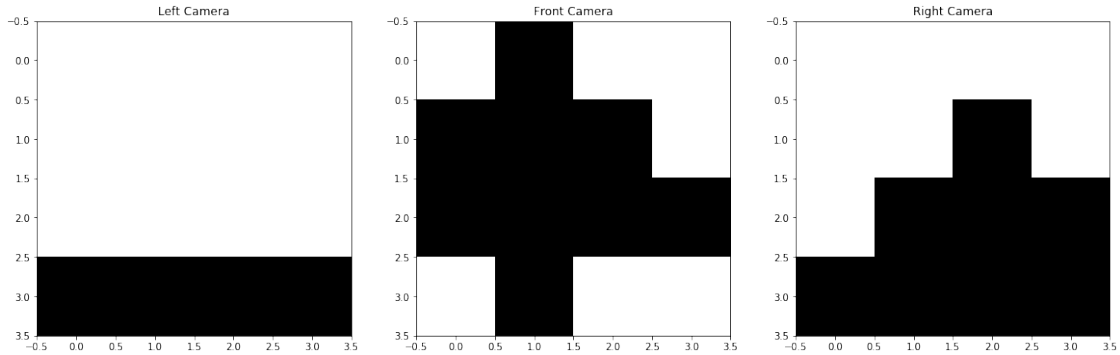
but what if all we know is that we have some area of interest, that we've reduced to a GPS grid:

```
??????
??????
??????
??????
??????
```

and the agent has to determine what kind of terrain is to the left, front and right of it?

Assuming the agent has a very simple visual sensor that constructs a 4x4 grayscale image for each of the three directions, it might it could see something like this:

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        import random

        plain =  [0.0, 0.0, 0.0, 0.0,0.0, 0.0, 0.0, 0.0,0.0, 0.0, 0.0, 0.0,1.0, 1.0, 1.0, 1.0]
        forest = [0.0, 1.0, 0.0, 0.0,1.0, 1.0, 1.0, 0.0,1.0, 1.0, 1.0, 1.0,0.0, 1.0, 0.0, 0.0]
        hills =  [0.0, 0.0, 0.0, 0.0,0.0, 0.0, 1.0, 0.0,0.0, 1.0, 1.0, 1.0,1.0, 1.0, 1.0, 1.0]
        swamp =  [0.0, 0.0, 0.0, 0.0,0.0, 0.0, 0.0, 0.0,1.0, 0.0, 1.0, 0.0,1.0, 1.0, 1.0, 1.0]

        figure = plt.figure(figsize=(20,6))

        axes = figure.add_subplot(1, 3, 1)
        pixels = np.array([255 - p * 255 for p in plain], dtype='uint8')
        pixels = pixels.reshape((4, 4))
        axes.set_title( "Left Camera")
        axes.imshow(pixels, cmap='gray')

        axes = figure.add_subplot(1, 3, 2)
        pixels = np.array([255 - p * 255 for p in forest], dtype='uint8')
        pixels = pixels.reshape((4, 4))
        axes.set_title( "Front Camera")
        axes.imshow(pixels, cmap='gray')

        axes = figure.add_subplot(1, 3, 3)
        pixels = np.array([255 - p * 255 for p in hills], dtype='uint8')
        pixels = pixels.reshape((4, 4))
        axes.set_title( "Right Camera")
        axes.imshow(pixels, cmap='gray')

        plt.show()
        plt.close()
```



which would be plains, forest and hills respectively.

2.1 The Assignment

For this programming assignment your task is to write an artificial neural network that determines what kind of terrain it is. This is a multi-class problem.

For a starting point, you can refer to **module-10-pseudocode.pdf** and the Self-Check.

2.2 Data

As before, we have clean examples of the different types of terrain but based on the location, the registration can be a bit off for some of the types and the visual sensor is often blurry.

Here are the clean examples with different registrations:

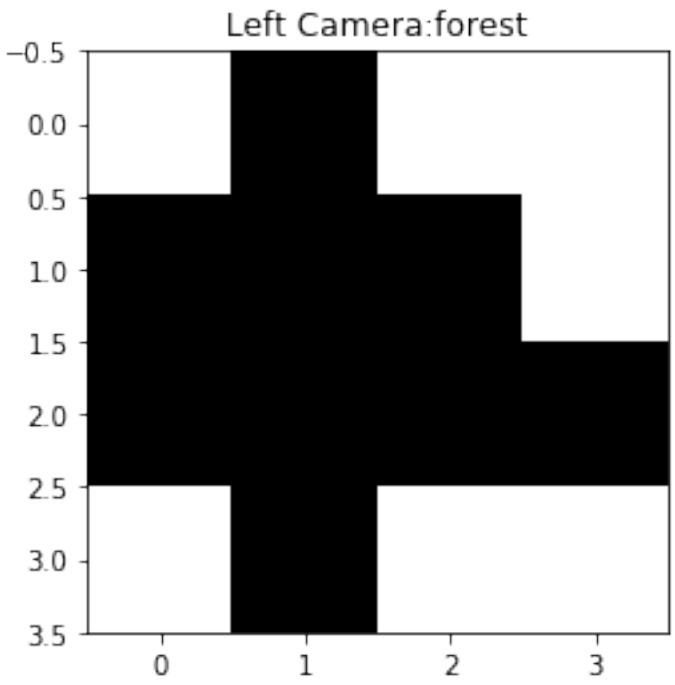
```
In [3]: clean_data = {
            "plains": [
                [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0
            ],
            "forest": [
                [0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 0.0, 0.0
                [0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0
                [1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0
                [0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0
            ],
            "hills": [
                [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0
                [0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 1.0
                [0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0
                [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0
            ],
            "swamp": [
                [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 1.0, 1.0, 1.0
                [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0
            ]
        }
```

Let's create a function that allows us to view any of these:

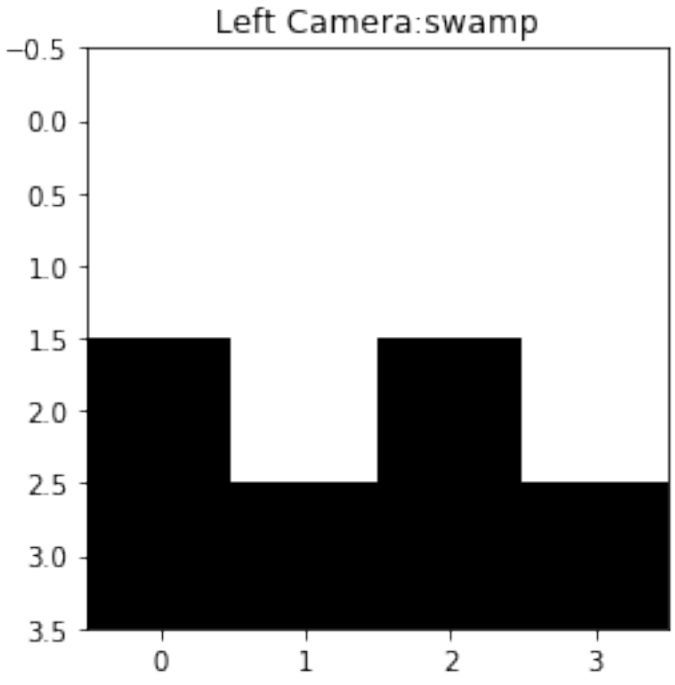
```
In [4]: def view_sensor_image( data):
            figure = plt.figure(figsize=(4,4))
            axes = figure.add_subplot(1, 1, 1)
            pixels = np.array([255 - p * 255 for p in data[:-1]], dtype='uint8')
            pixels = pixels.reshape((4, 4))
            axes.set_title( "Left Camera:" + data[-1])
            axes.imshow(pixels, cmap='gray')
            plt.show()
            plt.close()
```

"I think that I shall never see a thing so lovely as a tree."

```
In [5]: view_sensor_image( clean_data[ "forest"][0])
```



In [6]: `view_sensor_image( clean_data["swamp"][0])`



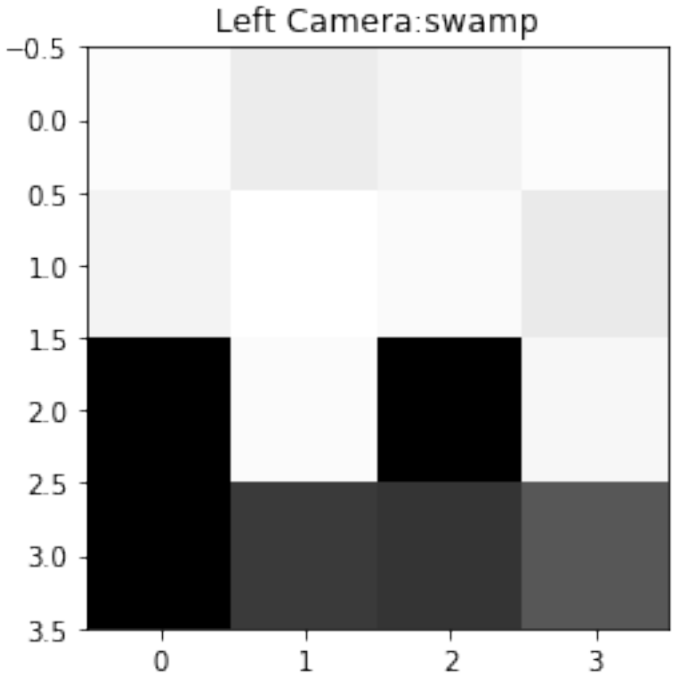
The data that comes in, however, is noisy. The values are never exactly 0 and 1. In order to mimic this we need a `blur` function.

We will assume that noise is normally distributed. For values that should be 0, the noisy values are distributed $N(0.10, 0.05)$. For values should be 1, the noisy values are distributed $N(0.9, 0.10)$.

```
In [7]: def blur( data):
            def apply_noise( value):
                if value < 0.5:
                    v = random.gauss( 0.10, 0.05)
                    if v < 0.0:
                        return 0.0
                    if v > 0.75:
                        return 0.75
                    return v
                else:
                    v = random.gauss( 0.90, 0.10)
                    if v < 0.25:
                        return 0.25
                    if v > 1.00:
                        return 1.00
                    return v
            noisy_readings = [apply_noise( v) for v in data[0:-1]]
            return noisy_readings + [data[-1]]
```

We can see how this affects what the agent *actually* sees.

```
In [8]: view_sensor_image( blur( clean_data["swamp"][0]))
```



You are going to want to write three (3) functions:

1. `learn_model`
2. `apply_model`
3. `evaluate_model`

But first we need one more function...

2.2.1 `generate_data`

With the clean examples and the `blur` function, we have an unlimited amount of data for training and testing our classifier, an ANN that determines if a sensor image is hills, swamp, forest or plains.

In classification, there is a general problem called the "unbalanced class problem". In general, we want our training data to have the same number of classes for each class. This means you should probably generate training data with, say, 100 of each type.

But what do we do about the class label with the neural network?

In this case, we can do "one hot". Instead of `generate_data` outputing a single 0 or 1, it should output a vector of 0's and 1's so that y is now a vector as well as x. We can use the first position for hill, the second for swamp, the third for forest and the fourth for plains:

```
[0, 1, 0, 0]
```

what am I? swamp.

Unlike logistic regression, you should set the *biases* inside the neural network (the implict $x_0 = 1$) because there are going to be lot of them (one for every hidden and output node).

`generate_data` now only needs to take how many you want of each class:

`generate_data( clean_data, 100)`

generates 100 hills, 100 swamp, 100 forest, 100 plains and transforms y into the respective "one hot" encoding. It will return a List of training data examples where each example is a Tuple of Lists (xs) and Lists (ys).

2.2.2 `learn_model`

`learn_model` is the function that takes in training data and actually learns the ANN. If you're up to it, you can implement a vectorized version using Numpy but you might start with the loopy version first.

In the lecture, I mentioned that you should usually mean normalize your data but you don't need to do that in this case because the data is already on the range 0-1.

You should add a parameter to indicate how many nodes the hidden layer should have.

When verbose is True, you should print out the error so you can see that it is getting smaller.

When developing your algorithm, you need to watch the error so you'll set verbose=True to start. You should print it out every iteration and make sure it is declining. You'll have to experiment with both epsilon and alpha; and it doesn't hurt to make alpha adaptive (if the error increases, make alpha = alpha / 10).

When you know that your algorithm is working, change your code so that the error is printed out only every 1,000 iterations (it takes a lot of iterations for this problem to converge, depending on your parameter values--start early).

`learn_model` returns the neural network. The hidden layer will be one vector of thetas for each hidden node. And the output layer will have its own thetas, one for each output (4 in this case). Return it as a Tuple: (List of List, List of List).

2.2.3 `apply_model`

`apply_model` takes the ANN (the model) and either labeled or unlabeled data. If the data is unlabeled, it will return predictions for each observation as a List of Tuples of the inferred value (0 or 1) and the actual probability (so something like (1, 0.73) or (0, 0.19) so you have [(0, 0.30), (1, 0.98), (0, 0.87), (0, 0.12)]. Note that unlike the logistic regression, the threshold for 1 is not 0.5 but which value is largest (0.98 in this case).

If the data is labeled, you will return a List of List of Tuples of the actual value (0 or 1) and the predicted value (0 or 1). For a single data point, you'll have the pairs of actual values [(0, 1), (0, 0), (0, 0), (1, 0)] is a misclassification and [(0, 0), (0, 0), (1, 1), (0, 0)] will be a correct classification. Then you have a List of *those*, one for each observation.

2.2.4 simple evaluation

We have an "unlimited" supply of data so we'll just generate a training set and then a test set and see how well our neural network does. Use the error rate (correct classifications/total examples) for your evaluation metric. We'll learn about more sophisticated

1. Assuming we have 4 hidden nodes, draw out on paper what you think the network looks like. (You don't need to submit this but you should have a good idea of what the network looks like before you start programming it).
2. generate training set.
3. learn a model
4. generate test set
5. apply model to test set.
6. evaluate the results.

As always when working with Lists or Lists of Lists, be very careful when you are modifying these items in place that this is what you intend (you may want to make a copy first)

Put your helper functions above here.

2.3 Main Functions

Everything you need is already in `module10.py`

Use `learn_model` to learn a ANN model for classifying sensor images as hills, swamps, plains or forest. Use your `generate_data` function to generate a training set with 100 examples for each. **Set debug to True**

```
In [11]: def learn_model( data, hidden_nodes, debug=False):
             pass

         train_data = generate_data( clean_data, 100)
         model = learn_model( train_data, 4, True)
```

Use `generate_data` to generate 100 blurred examples of each terrain and use this as your test data.

```
In [ ]: test_data = generate_data( clean_data, 100)

        def apply_model( model, test_data, labeled=False):
            pass

        results = apply_model( model, test_data)

In [ ]: error_rate = evaluate_results(results)
        print(f"The error rate is {error_rate}")
```