Assignment 2 Full ~~[pre-release summary of certain relevant parts only!]~~

Submission Note (as explained in class): In order to help you prepare for the midterm, this assignment was released in two parts:

- Part 1 should be submitted by **Saturday 8 Feb 2020 at 9pm**. As long as you submit a functioning "Option 1" for Part 1(a) and Part 1(c) by this date, you will get at least 50% on the entire Part 1 (see below to read exactly what the different options include). Note that as mentioned in class, more aspects will be added to Part 1 as well, but they are not relevant to the Saturday submission opportunity.
- The rest of the assignment is now due on ~~Tues 18 Feb 2020, 9pm~~ **Sun 23 Feb 2020, 9pm**. At that time, you *entire* assignment will be marked (so even if you did not complete Part 1 previously, you could still get 100%).
- **New:** Due to the delays in releasing the final version, there will not be a "Part 2" to this assignment involving PyTorch.

Introduction

In Part 1 of this assignment you will implement a basic neural net in `numpy`. You are not to use any libraries such as `sklearn` or `keras`, although you are always welcome to check if there is a certain library that you think would make sense for you to use!

[20pts Total] Part 1: A Feedforward Neural Network

As discussed in lab, we will follow some of the examples shown in PlayGround.

a) **[3pts] Create the data set.** Choose from the following ~~two options (for now)~~ three options:

[1pt] Option 1 : Create and visualize a two-class data set consisting of two Gaussians, similarly scaled as the ones in the bottom left data set in playground.

[2pts] Option 2 : Create and visualize both: (i) a two-class data set as in Option 1, as well as (ii) a two class data set of concentric circles, as in the upper left data set in playground.

[3pts] Option 3 [**new**] : Create and visualize both Gaussian clusters and concentric circles, but this time allow the user to specify how many different clusters/circles the distribution includes. For example, if `(numClasses==3)` then that will generate three Gaussians, or three concentric circles, corresponding to three distinct classes. The sample function below has been modified accordingly, with a default value, so that if you are only able to handle two classes, it will still be called the same way. New.

Note that your data sets must be randomly generated, so that running the code multiple times with different seeds will give different (but similar looking) results each time. Also note the args have couple small differences from before. Your data generation should be run with the function:

```
X = generateData( numExamples, distributionShape, numClasses = 2, numOutputs=1 )
    # X is just the name of the matrix containing the generated dataset
    # numExamples is the number of examples in the dataset, with each row corresponding to one
        example
    # distributionShape can either be 'g' (gaussian) or 'c' (circles)
    # if numClasses==2 then numOutputs can be 1, assuming a sigmoid unit, or 2, corresponding to
        softmax.
    # Otherwise, numOutputs must be the same as numClasses.
```

---

b) **[1pt] Add noise [new] :** Add an option that allows you to add label noise with a given probability.

c) **[7pts] Train a small feedforward neural network [new] :**

Starting with the example done in lab, you will continue to implement a small feedforward network in `numpy`. At this stage, you will write two functions:

```
train( X, numInput, numHiddenUnits, numOutput, activationType, numIter) \\
    // X is the data matrix, one example per row
    // numInput is the number of columns of input in X
    // numOutput is the number of columns of output in X
    // activationType is either 'linear' or 'sigmoid' or 'reLU';
    //          it specifies only the activation function of the single hidden layer
    // numIter is the number of iterations for which your training algorithm should run.
    //
    // Return:
    //  The function should return (W1, W2), a 2-tuple of weight matrices for the
    //  input-to-hidden and hidden-to-output layers, similarly to the sample code
    //  done in lab.
}
```

If $numOutput == 1$, then the output unit is a sigmoid (no matter what activation function the hidden units have). If $numOutput > 1$, then the output units should be a softmax layer.

Your second function should take as input a data matrix (which might contain test data), as well as the trained weights, and the same set of architecture parameters (i.e. numInput, numHiddenUnits, numOutput, activationType) and also a data matrix , and a verbosity parameter, and it should output the results of the trained weights applied to the given data.

```
predict( X, W1, W2, numInput, numHiddenUnits, numOutput, activationType, verbosity )
    // X, W1, W2, numInput, numHiddenUnits, numOutput, activationType are all as before.
    //
    // Return:
    //  This function returns a matrix containing with the same number of rows
    //  as X, and with a total of (numOutput+1) columns:
    //  the first numOutput columns contain the predicted values for the given input X. //  The
        last column contains the cross-entropy error for that prediction, given the
    //  correct answer as defined in X.
    //
```

This part should work for an arbitrary number of input units, an arbitrary number of hidden units (keeping just a single layer), and for the three activation functions (linear, sigmoid, reLU). ~~Later parts of this question will ask you to write this in an object-oriented format and add additional features. For Saturday, you are simply asked to write this for an arbitrary number of input units, an arbitrary number of hidden units (keeping just a single layer), and for three activation functions (linear, sigmoid, reLU). You should try to allow for multiple output units, but they do not need to function well: there are some numerical precision issues that you might be unable to solve on your own at this point.~~

d) **[5pts] Refactor [new] :** Refactor your existing code into a cleaner object-oriented form. For now, you can assume the default values as shown in the listing below. That is, you need create a class for a model that has a single sigmoid output, with two hidden layers (in addition to the input), with each hidden layer having two ReLU units.

```
class Model:
    def __init__(self, numInputs=2, numOutputs=1, layerSize=2, numHiddenLayers=2, activationType
        ='R'):
        // numInputs: number of inputs to the net
        // numOuputs: number of output units in the output
        // layerSize: the number of units in each hidden layer
        // activationType: either 'L' (linear), 'S' (sigmoid), 'R'(reLU)
        //          This is the activation type of all hidden layers.
```

```
        //          Note that the output should automatically be a softmax layer if
        //          numOutputs is greater than 1, and a sigmoid otherwise.
        //          (So the default output is a single sigmoid unit).

        self.numInputs = numInputs
        self.numOuputs = numOuputs
        self.layerSize = layerSize
        self.numHiddenLayers = numHiddenLayers
        self.activationType = activationType
```

Create a function `setWeights( value )` that allows us to set all the weights to a constant value (for debugging). You should also have a function `initWeights(mean,stdev)` that initializes all the weights to be randomly chosen with the provided mean and standard deviation.

We should be able to call your model as follows:

```
X = generateData( 100, 'g', 2, 1 )  # use 2 gaussians to generate
                # 100 data points, with a single target value (0 or 1)

np.random.shuffle(X)   # make sure the examples are shuffled
X_train = X[:90]        # create a 90/10 train/test split
X_test = X[90:]

net = Model()
net.setInput(X_train)
net.setTest(X_test)
net.initWeights(0.0,0.01)  # initialize weights with mean 0 and standard deviation 0.01
trainError = net.train(100, 0.1)   # train for 100 iterations with a step size of 0.1, this
    should return 100x2 array containing the training and test error at each of the 100
    iterations

testError = net.test() # return the error on the test set

Y = net.predict(X1) # assuming X1 is a valid input matrix, return the array of predictions for X1
```

For this part, you only need to have it work for the given default values.

e) **[1pt] [new] :** Allow a variable number of hidden units. E.g.

```
net = Model(2,1,5,2,'R') #same as above but with 5 hidden units per layer
```

f) **[1pt] [new] :** Allow the various possible activation types. E.g.

```
net = Model(2,1,2,2,'S') #same as above but with sigmoid hidden units
```

g) **[1pt] [new] :** Allow multiple output units, using softmax and a cross-entropy error.

```
net = Model(2,3,2,2,'S') #3 softmax output units
```

h) **[1pt] [new] :** Allow multiple hidden layers.

```
net = Model(2,1,2,5,'R')  # 5 hidden layers
```

In this assignment, your code will primarily be marked by running it. It is possible that the marker will look at the code itself, but not guaranteed. The marking will be done based on whether the test scripts run properly and give correct answers. We may provide some additional informal suggestions over the next couple of days, but this document should provide sufficient specification in its current form for completing the assignment.

General Marking Notes and Tips

- In some assignments (such as this one), you will be marked by having the markers run test scripts as specified, and they will examine the output of those test scripts. That means that your code must follow the specified format exactly (e.g. any function templates), or the test scripts may not run.
- You will be marked for accuracy, correctness and also clarity of presentation where relevant.
- Be selective in the experimental results you present. You don't need to present every single experiment you carried out. It is best to summarize some of your work and present only the interesting results, e.g. where the behaviour of the ML model behaves differently.
- In your answers, you want to demonstrate skill in using any required libraries to experiment with ML models, and good understanding / interpretation of the results. *Make it easy for the marker to see your skill and your understanding.*
- Justify your meta-parameter choices where appropriate.
- Except where you are explicitly asked to implement a particular algorithm/step yourself, if there is a procedure in sklearn that does the task, you are free to find it, read its documentation and use it. If you are not sure, it is best to on the shared notes or in class!
- **You should submit one python notebook with your answers.**
- Liberally add markdown cells to explain your code and experimental results. Make good use of the formatting capabilities of markdown to make your notebook highly readable.
- Add links to all online resources you use (markdown syntax is: [URL](anchor_text)). Avoid explicit long URLs in your markdown cells.