Project 01A Instructions
DSCI 39001: Neural Networks

In this assignment, you will begin to implement your own neural network classifier. It will support an arbitrary number of inputs, hidden layers, nodes per layer, and classes. We will assume that the classes are encoded as consecutive integers 0, 1, 2, …, K – 1.

1. Start by opening Spyder (or a different editor of your choice), and create a script titled **ANN_YourLastName.py**.

2. Import numpy.

3. Define a class called `ANN` (which stands for Artifical Neural Network).

4. The constructor `__init__()` should take six parameters: `self`, `X`, `y`, `layer_sizes`, `activation`, and `weights`. The `X` and `y`, parameters will store the training data. The `layer_sizes` parameter will be a list of integers containing the number of non-bias nodes in each of the layers (including the input and output layers). The length of this list will determine the number of layers in the network. The `activation` parameter will set the activation of the hidden layers. It will expect a string that is either `'sigmoid'` or `'relu'`. Set its default value to `'relu'`. Finally, the `weights` parameter will specify a set of weights to be used in our model, and should have a default value of **None**. Our first pass at this implementation will not involve any training, so this parameter will provide us with the option of setting our weights manually.

5. The constructor should perform the following tasks:

   a) Store `X`, `y`, `layer_sizes`, `activation` as class attributes.

   b) Create the following attributes: `self.N` should contain the number of training samples, `self.M` should contain the number of predictors, `self.D` should contain the depth of the network (i.e. the number of non-input layers), and `self.K` should contain the number of classes.

   c) You should `assert` that the first entry of `layer_sizes` is equal to `self.M`, and that the last entry of `layer_sizes` is equal to `self.K`.

   d) Create an attribute called `self.a`. This should store the an activation function, either sigmoid or relu. Not that it should store the function itself, and not just the name of the function. Use numpy functions so that these functions can be applied to arrays. Lambda functions provide a convient tool for defining `self.a`, but you are not required to use them if you don't want to. I recommend it, though.

   e) If the `weights` parameter is equal to **None**, then randomly initialize your weight matrices, storing them in a list called `self.weights`. You will need to create a total of `self.D` weight matrices. A given weight matrix will be stored as a 2D array, with a number of rows equal to the total number of nodes in the previous layer, and a number of columns equal to the number of non-bias nodes in the subsequent layer. Each weight should be randomly initialized to be drawn from a uniform distribution on the interval [-1,1]. Such a matrix can be created using this code: `wts = np.random.uniform(low=-1, high=1, size=(rows, cols))`

   f) If the `weights` parameter is not equal to **None**, then set `self.weights` to be equal to the weight list provided in this parameter.

   g) Create a 2D Matrix with shape `(self.N, self.K)` called `self.T`. If observation `i` is in class **k**, then `self.T[i,k]` should be equal to 1. All other entries should be 0. This matrix is called an indicator matrix. We will not use it in this project, but it will be required for training later. I used a simple loop for this, but there is likely a slick way to do this using numpy code.

6. Create a method called `predict_proba()`. This should take two parameters: `self` and `X`. The parameter `X` will store a feature array. This method should return a 2D array with the same number of rows as `X` and a number of columns equal to `self.K`. The entries in a given row should be the estimated probabilities that the observation is in each of the K classes.

   Pseudo-code for this method is provided below:

   ```
   Create array named ones with shape (number of rows in X, 1)
   Set A equal to X
   Create a list called self.activations. It should initially contain only A.

   for each non-input layer:
       A a column of ones to the front of A
       Z = matrix product of A and layer weights

       if in output layer:
           A = softmax(Z)
       else:
           A = self.a(Z)

       Append A to self.activations

   return A
   ```

   To provide two points of clarification: The column of ones is to account for the bias node in each non-output layer. Also, we store the output of each layer in `self.activations` because we will need them layer to calculate our weight updates later when training.

7. Create a method called `predict()`. This should take two parameters: `self` and `X`. The parameter `X` will store a feature array. This method should return a 1D array with a number of entries equal to the number of rows in `X`. The entries should be integers indicated the predicted class for each observation in `X`. This can be accomplished with 3 (or fewer) lines of code using `predict_proba()` and `np.argmax()`.

8. Create a method called `score()` that takes three parameters: `self`, `X`, and `y`. The method should return a list or tuple containing two values: The model's loss and the model's accuracy, as calculated on `X`, and `y`. This method should make use of `predict_proba()` and `predict()`.

Pending: A Jupyter Notebook will be provided later with some instructions for testing your code.