Neural Networks

In your implementations, please use the function prototype provided (i.e. name of the function, inputs and outputs) in the detailed instructions presented in the remainder of this document. We will be testing your code using a test function that which evokes the provided function prototype. If our testing file is unable to recognize the function prototype you have implemented, you can lose significant portion of your marks.

NumPy Implementation Overview

In this specific implementation, the task of the NN is to identify the probability of the input feature vector $\underline{x}$ supplemented to the NN belonging to one of two classes $\{-1, +1\}$. The output of the NN will be a probability (i.e. a value ranging between 0 and 1) that indicates the probability of the input belonging to class $+1$. The probability of belonging to the other class -1 will be 1 minus the output probability. **The NN with L layers will be composed of the input, output and hidden layers as follow**:

1. The input layer is composed of $d + 1$ inputs where d is the number of features in each input vector $\underline{x}$ and a bias node.

2. The output layer is composed of a single node which outputs the probability of the input vector $\underline{x}$ belonging to class $+1$.

3. The number of hidden layers in between the input layer and output layer along with the number of nodes in each hidden layer are contained within the `hidden_layer_sizes` parameter which is a vector of length $L - 2$ that will be passed as an argument during the construction of the NN. For instance, when `hidden_layer_sizes=[30,100]`, the first hidden layer is composed of 30 nodes (not including the bias node) and the second hidden layer is composed of 100 nodes (also not including the bias node). In this particular case, there are $L = 4$ layers in total (2 are hidden layers).

In your implementation of the NN, you will assume that all activation functions of every node in each hidden layer will be the same and is defined by the function `activation` discussed in Part 2a. The output node will be defined by the function `outputf` also detailed in Part 2a. Every hidden layer will have a bias node that outputs a value of 1.

Part 2a: Activation Functions, Output Functions and Error Functions

In this part, you will be implementing the activation functions, error functions and output functions that will be utilized in the neural network. For this, you will use the `NumPy` library functions. Specifically, you will be implementing six functions which are detailed in the following:

- Function: `def activation(s)`
  - Inputs: `s`
    The input to this function is a single-dimensional real-valued number. You will implement the `ReLU` function here.

- Output: `x`
  The output will be the single-dimensional output of performing a `ReLU` operation on the input `s` (i.e. $x = \theta(s) = ReLU(s)$).
- Function implementation considerations:
  You can use an if-statement to evaluate the conditions in the `ReLU` function.

- Function: `def derivativeActivation(s)`
  - Inputs: `s`
    The input to this function is a single-dimensional real-valued number. You will implement the derivative of the activation function (i.e. `relu` function) here.
  - Output: $\theta'(s)$
    The output of this function is the derivative of the activation function $\theta(s)$.
  - Function implementation considerations:
    You can use an if-statement to evaluate the conditions of the derivative of the activation function.

- Function: `def outputf(s)`
  - Inputs: `s`
    The input to this function is a single-dimensional real-valued number. You will implement the output function which is the logistic regression (i.e. sigmoid) function (i.e. $\frac{1}{1+e^{-s}}$) here.
  - Output: `x_L`
    The output of this function is `x_L` which is evaluated using the logistic regression function. This is a single-dimensional value.
  - Function implementation considerations:
    To evaluate the exponent, you can use the `np.exp` function available in NumPy.

- Function: `def derivativeOutput(s)`
  - Inputs: `s`
    The input to this function is a single-dimensional real-valued number. You will implement the derivative of the output function (i.e. sigmoid function).
  - Output: `x_L`
    The output of this function is derivative of the output function evaluated at s (i.e. derivative of the sigmoid function).
  - Function implementation considerations:
    To evaluate the exponent, you can use the `np.exp` function available in NumPy.
    To evaluate the square, you can use the double asterisk syntax (i.e. `s**2`$=s^2$).

- Function: `def errorf(x_L,y)`
  - Inputs: `x_L` and `y`
    The input to this function is a single-dimensional real-valued number which is `x_L`

and a single dimensional discrete variable `y` which takes values in the set {+1,-1}. `y` is essentially the class that the training data point $\underline{x}_n$ belongs to. `x_L` is the output from the NN model which is obtained by applying forward propagation to $\underline{x}_n$. You will implement the log loss error function that evaluates the error introduced in the output of the NN with respect to the training observation.

  - Output: e_n
    The output of this function is e_n which is evaluated via the log loss error function: $e_n(x\_L, y) = -\mathbb{I}_{y=+1}\log(x\_L) - \mathbb{I}_{y=-1}\log(1 - x\_L)$. The indicator function is denoted as $\mathbb{I}_{condition}$ which returns 1 if the condition in the subscript is true and 0 otherwise.
  - Function implementation considerations:
    To evaluate the log, you can use the `np.log` function available in NumPy. An if-statement can be utilized to account for the indicator function.

- Function: `def derivativeError(x_L,y)`

  - Inputs: `x_L` and `y`
    The input to this function is a single-dimensional real-valued number which is `x_L` and a single dimensional discrete variable `y` which takes values in the set {+1,-1}. `y` is essentially the class that the training data point $\underline{x}_n$ belongs to. `x_L` is the output from the NN model which is obtained by applying forward propagation to $\underline{x}_n$. You will implement the derivative of the error function (i.e. log loss function) with respect to the input x_L.
  - Output: $\frac{\partial e_n}{\partial x_L}$
    The output of this function is the derivative of the error function evaluated at `x_L` (i.e. derivative of the log loss function).
  - Function implementation considerations:
    An if-statement can be utilized to account for the indicator function.

The following is the mark breakdown for Part 2a:

- Test file successfully runs all six implemented functions: 12 marks
- Outputs of all six functions are close to the expected output: 12 marks
- Code content is organized well and annotated with useful comments: 14 marks

Part 2b: Training the Neural Network

In this part, you will implement four functions that will train your NN. Details are provided in the following:

- Function: `def forwardPropagation(x, weights)`
  In this function, you will implement the forward propagation algorithm that computes the vector of outputs x^l at each layer l of the NN and the vector of sum s^l that are passed as input into each layer l in the NN.

- Inputs: `x, weights`
  The input `x` is $d+1$ dimensional. The $1st$ component of `x` is 1 to account for the bias node. The remaining d components represent input feature vector $\underline{x}_n$ (i.e, $\underline{x}_n = [\underline{x}_{n,1}, \underline{x}_{n,2}, ..., \underline{x}_{n,d}]$, x= $[1, \underline{x}_{n,1}, \underline{x}_{n,2}, ..., \underline{x}_{n,d}]$). The `weights` variable is a list where each element l is a matrix representing the weights of edges between layer l and $l+1$. The dimension of a w^l matrix is $\{d^l+1\} \times d^{l+1}$ as there are weights propagating out of d^l+1 nodes (including the bias node) into d^{l+1} nodes in the next layer (not including the bias node). There are $L-1$ matrices in the `weights` list.
- Output: X, S
  For each layer (except the input layer), you will compute the sum s_j^l of all weights on edges that are multiplied by the outputs of the previous layer $l-1$ that are entering into node j located i layer l along with the output x_j^l resulting from applying the activation function to s_j^l. The input into node j located in layer l is $s_j^l = \sum_{i=1}^{d^{l-1}+1} w_{ij}^{l-1} x_i^{l-1}$ and the output is the application of the activation function to the input $x_j^l = \theta(s_j^l)$. This function will return two outputs.
  1. `X` is the output at all nodes residing in all layers of the NN. Thus, `X` is a list that is composed of elements where the l^{th} element is a vector of size d^l+1 **(i.e. output of each node in layer l plus the bias node)** with the exception of the last element which is a single-dimensional vector representing the output node. `X` is composed of L elements where L is the total number of layers in the NN (input, output and hidden layers).
  2. `S` is the input into all nodes located in all layers of the NN **(with the exception of the input layer)**. `S` is another list which is composed of $L-1$ elements. Element l in `S` represents a vector of d^{l+1} dimensions where each component s_j^l represents the inputs into node j at layer $l+1$.
- Function implementation considerations:
  You will utilize the `activation` function and `outputf` function that you had implemented in Part 2a to calculate the outputs belonging to nodes in the hidden and output layers respectively.

- Function: `def backPropagation(X,` y_n`, S, weights)`
  You will implement the back propagation algorithm here that computes the error gradient $\frac{\delta e_n}{w_{ij}^l}$ for every weight in the NN around the training point $\underline{x}_n$.
  - Inputs: `X,` y_n`, S, weights`
    `X` and `S` will be the outputs of applying the forward propagation algorithm for training point $\underline{x}_n$. y_n represents the class label observed for the input vector $\underline{x}_n$. The `weights` input is defined in the same manner as that in the `forwardPropagation` function.
  - Output: g
    This function will output a list `g` which contains all the error gradients computed for all the weights in the NN. This list has the same dimensions as the `weight` variable.

- Function implementation considerations:
  As discussed in the lecture, you will need to begin from the last layer (i.e. the output layer) and then compute the gradients recursively for the earlier layers using the notion of forward and backward messages. The forward messages are already available in `X`. To compute the backward messages, you will need to utilize `S` and derivatives of the activation and output functions. For this, you can use the `derivativeError`, `derivativeOutput` and `derivativeActivation` that you had implemented in Part 2a. To store the backward messages as you move along the NN from the last layer to the input layer, you will need to insert the current computations to the beginning of the list. For this, you can use the `insert` function to insert at the beginning of the list.

- Function: `def updateWeights(weights,g,alpha)`
  In this function, you will apply the stochastic gradient descent update to improve the weights based on the error gradients computed using the back propagation algorithm.

  - Inputs: `weights,g,alpha`
    `weights` represent the current weights assigned to all edges in the NN. This list has the same dimensions as the `weight` parameter passed as an argument in the previous function. `g` is a list that contains all the gradients of the weights as computed by the `backPropagation` function. `alpha` is the step-size of the weight update. Thus, each weight is updated according to the SGD update: $w_{ij}^l \leftarrow w_{ij}^l - \alpha \frac{\delta e_n}{\delta w_{ij}^l}$.
  - Output: `nW`
    The output `nW` will have the same dimensions as the input `weights` parameter. `nW` will contain the updated weights.
  - Function implementation considerations:
    You may utilize the `list` features in Python to perform the weight updates.

- Function: `def errorPerSample(X,`y_n`)`
  This function computes e_n which is the error contributed by the training point $\underline{x}_n, y_n$.

  - Inputs: `X,`y_n
    The inputs that are passed are the same as the `X` and y_n arguments passed to the `backPropagation` function.
  - Output: eN
    This is the error contributed at the last layer `x_L`. This is a single-dimensional output.
  - Function implementation considerations:
    You will utilize the `errorf` implemented in Part 2a.

- Function: `def fit_NeuralNetwork(X_train, y_train, alpha, hidden_layer_sizes, epochs)`
  In this function, you will implement and train the NN using all of the afore-mentioned functions.

- Inputs: `X_train, y_train, alpha, hidden_layer_sizes, epochs`
  `X_train` is the training dataset that is composed of N, d-dimensional vectors. Each training point $\underline{x}_n$ is a d-dimensional vector of which the transpose is taken and forms the n^{th} row in the `X_train` matrix. `y_train` is an $N - dimensional$ vector that consists of the corresponding class observations for each training vector in `X_train`. `alpha` is the step-size used to update the weights in the NN. `hidden_layer_sizes` contains information about the number of nodes in each hidden layer (as specified in the introduction of this assignment). `epochs` represents the number of times the training process will pass through the entire training set to tune the weight parameters.
- Output: `err, weights`
  `weights` contains the final weights obtained from training the NN using the back propagation algorithm. The dimensions of this list will be the same as the input argument to the `backpropagation` algorithm. *err* is a list that contains the average error computed at each epoch. This will be a list containing `epoch` number of elements.
- Function implementation considerations:
  In this function, you will need to initialize the weight list. Be careful with how you assign the indices as this is a common cause of error.
  1. For initializing the weights, you can set every weight to be a random Gaussian with zero mean and 0.1 standard deviation. You can use the function `numpy.random.normal` or `numpy.random.randn` for this.
  2. At each epoch, you may shuffle the training dataset so that you train the weights using a randomly selected training point which is consistent with the SGD algorithm. For shuffling the dataset, you can use the `np.random.shuffle` function in Python.
  3. For each point selected in the training dataset, you will apply the forward propagation algorithm, and back propagation algorithm. You will update the weights using the outcome of these functions.
  4. You will also accumulate the error for each point as you pass these to tune the weight parameters. At the end of each epoch, you will take the average of the error accrued at that epoch. This is repeated until the training takes place for the specified number of epochs.

  You will use all of the afore-mentioned functions in this part to implement the training process of the NN. The `List` class and the `hstack` function will be handy here as well.

The following is the mark breakdown for Part 2b:

- Test file successfully runs all four implemented functions: 8 marks
- Outputs of all four functions are close to the expected output: 8 marks
- Code content is organized well and annotated with useful comments: 10 marks

Part 2c: Performance Evaluation

You will be implementing four functions to evaluate the performance of your NN and these are detailed as follows:

- Function: `def pred(x,weights)`
  In this function, for a given input vector `x` ($\underline{x}_n$ with a bias node), you will compute the output class using the NN that has been implemented.
  - Inputs: `x,weights`
    `x` and `weights` is defined in a similar manner as the previous `forwardPropagation` function.
  - Output: `c`
    You will output the class the input belongs to. As the output of the NN is the probability of the input belonging to class +1, you will apply a threshold of 0.5 to identify which class the input belongs to. If the probability is greater than or equal to 0.5, then it is class +1. Otherwise, it is class -1. This will be the output of the function.
  - Function implementation considerations:
    You will utilize the `forwardPropagation` function implemented in the previous part.
- Function: `def confMatrix(X_train,y_train,w)` This function evaluates the confusion matrix (**feel free to use scikit-learn to compute the confusion matrix**). `X_train,y_train` are composed of the training points and corresponding observations. `w` is the final weight list computed via the training of the NN. It will be implemented in a similar manner as the previous assignment. You will use the `pred` function to evaluate the output of the NN in this implementation. The output of this function will be a two-by-two matrix. The first row, first column will contain a count of how many testing samples actually belong to class -1 that the NN identified as belonging to class -1. The first row, second column will contain a count of how many testing samples actually belong to class -1 that the NN identified as belonging to class +1. The second row, first column will contain a count of how many testing samples actually belong to class 1 that the NN identified as belonging to class -1. The second row, second column will contain a count of how many testing samples actually belong to class +1 that the NN identified as belonging to class +1.
- Function: `def plotErr(e,epochs)`
  - Inputs: `e,epochs`
    `e` is the error list containing the average error at each epoch of the training process. This is output by the `fit_NeuralNetwork` function. `epochs` is a single-dimensional parameter that denotes the number of iterations sweeping through the entire training set that is utilized to tune the NN weight parameters.
  - Output: N/A
    This function will plot the error over the training epochs. There will be no outputs.

  - Function implementation considerations:
    You will use the `plt.plot` function and `plt.show` function to plot the error at each epoch in the y-axis and the epoch number in the x-axis.

- Function: `def test_SciKit(X_train, X_test, Y_train, Y_test)`
  You will use the built-in `MLPClassifier` to train the NN and evaluate the performance of this NN using the `confusion_matrix` function available in the `scikit-learn` library.

  - Inputs: `X_train, X_test, Y_train, Y_test`
    These inputs are the same as the previous assignment for the PLA algorithm.
  - Output: cM
    This function will output the confusion matrix obtained for the test dataset.
  - Function implementation considerations:
    You will utilize the `MLPClassifier` to instantiate the NN. You will use the following input parameters when you initialize the MLPClassifier: `ADAM` solver, `alpha`=10^{-5}, `hidden_layer_sizes=(30, 10)`, `random_state=1`. Then you will evoke the `fit` function to train the NN using the training dataset `X_train,Y_train`. You will then use the `predict` function to obtain the output of the NN for the test input data `X_test`. Finally, you will use the `confusion_matrix` to identify how many test points have been correctly predicted by the NN.

Answer the following question(s) Remember to submit this in the `pdf` file together with your code.

- In your scikit-learn implementation, wet `hidden_layer_sizes` to the following values `(5,5),(10,10) ,(30,10)` and keep other parameters the same or as default. Report their training and testing accuracy using the same train-test split in the `test()`.

The following is the mark breakdown for Part 2c:

- Test file successfully runs all four implemented functions: 8 marks
- Outputs of all four functions are close to the expected output: 8 marks
- Code content is organized well and annotated with useful comments: 10 marks
- Question answered correctly: 10 marks