CSC 470: Introduction to Neural Networks

Programming Project #1

Implementing a Perceptron

Background

In Module 3 we discussed the **perceptron**, which is the simplest form of a neural network, consisting of only a single neuron. Figure 1 shows a diagram of the classic perceptron.

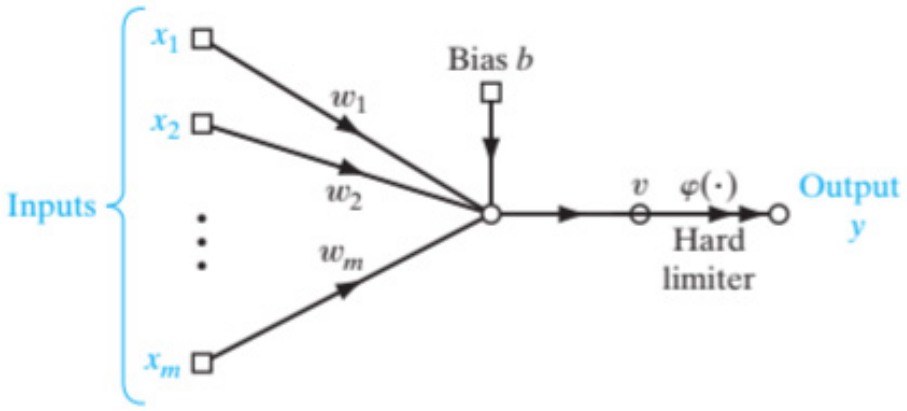

Figure 1. A perceptron as a signal flow diagram.

A perceptron can accept any number of inputs, as well as a bias input, but it can only effectively classify inputs into at most 2 classes. If the inputs can be graphed on a scatter plot, they would form two distributions in the coordinate space, and a well trained perceptron should be able to place a hyperplane between the two distributions and separate them from each other. This assumes, of course, that there are no more than two output classes. If there are more than two output classes, a perceptron can only recognize at most two of them.

Despite its simplicity, a perceptron can be quite powerful. A perceptron's accuracy will depend on a number of factors, including:

- the degree of overlap between the output classes
- choice for the bias value
- choice for the activation function
- calculation of the learning rate parameter
- selection of the training set
- number of batches, iterations, and epochs used in the training

For this project, the goal is for you to implement a functional perceptron and use it on a set of sonar data to see how accurately you can get your perceptron to distinguish between two different classes of objects: rocks vs. metal cylinders. Once you have implemented your perceptron, you will need to train it using a subset of inputs from the dataset, then validate it using the entire dataset. You will need to keep track of how many inputs are correctly identified in order to calculate the accuracy of your perceptron. You will also need to implement a test harness that will read the data, create the training set, initialize the perceptron, train it and then test it.

The Dataset

The set of sonar data is provided as a comma-separated value (csv) file named, "sonar_all-data.csv". This file contains 208 rows of inputs, with each row containing 61 columns. The first 60 columns of each row contains the 60 values for each input, representing the strength of the sonar signal as received from different angles of incidence. Thus, your perceptron will need to accept 60 data inputs. The data signals are floating point values, most of which are in the range 0 to 1. The last column contains the correct identities (or desired outputs) of the objects corresponding to the sonar signals in each row. There are only two different classes of values: 'R' for Rock and 'M' for metal cylinder.

One Because the data are in a csv file, all the values are in plain text format. Your test harness will therefore need to read the data from the file and convert all the string representations of floating point values to actual float type. The desired outputs in the last column of the data file will also need to be converted into two unique integer values, such as 0 and 1.

The Basic Process

Before getting into the details of the perceptron implementation, we need to go over the basic process involved in solving the problem. The steps in the process are outlined below.

Step 1: Acquire the dataset

The data are being provided to you as a csv file, so you just need to read the data from this file and store the data as a matrix. It is fine to leave the desired outputs in the matrix since it is easy enough to read them or bypass them as needed. In total, the matrix should have 208 rows, with each row containing 61 columns (columns 1-60 for the inputs, column 61 for the desired output).

Step 2: Convert the string input values to floats

Since the csv values, when read, will be in string format, the input data in columns 1-60 must be converted to float values. (*Note that programmatically, with 0-based arrays or lists, it will be columns 0-59 that contain the input values.*)

Step 3: Convert the desired outputs to int values (0 or 1)

The string values in column 61 (*column 60 in 0-based notation*) representing the desired outputs ('R' or 'M') must be converted to unique integer values such as 0 and 1.

Step 4: Create the training set

In order to train the perceptron we need to take a subset of the dataset to use for training. There is no hard and fast rule for how many inputs to use for training, and there are many ways in which the training inputs can be selected. Clearly, if too few inputs are used for training, there is a risk that the perceptron will not be able to accurately classify inputs it did not see during training. Conversely, using too many inputs for the training set may improve the accuracy of the perceptron, but will limit the usefulness of the perceptron if there are only a few inputs left that haven't been seen before.

Step 5: Create the perceptron

With the training set in hand, the next step is to initialize the perceptron. Assuming the activation function has been hardcoded into the perceptron, all that is left to do is initialize all the synaptic weights and set the bias.

Step 6: Train the perceptron

Now it is time to train the perceptron. If the perceptron's synaptic weights and bias have not been initialized yet, this can be done now, or those values can be reset from a previous training. At this point it is necessary to define a couple more values:

Learning rate parameter

This is the amount by which the synaptic weights will be adjusted in the event an input is incorrectly classified. The choice of this value is arbitrary, though some values may prove to be more effective than others. Also, keep in mind that using very small values for the learning rate parameter will only shift the synaptic weights by a small amount. Very large values will shift the weights more rapidly, but may overshoot the value needed for optimal performance.

Epochs, batches, and iterations

You will frequently see these three terms used in the context of training neural networks. The training set is often too large to use all at once to train a neural network, so it must be split into two or more **batches**. For example, let's say we have 10,000 inputs (i.e., rows, not individual inputs) in the training set. If the hardware we're using doesn't have enough memory or processing power to handle the entire training set, we would split it into smaller batches. For example, we could divide the 10,000 inputs into 10 batches of 1000 inputs. Each of the 10 batches is then fed to the neural network. Processing one batch counts as 1 **iteration**, so to process all 10,000 inputs in the training set we would need 10 iterations. One **epoch** is a single processing of all the batches for the training set. So for the example described above:

1 *epoch* = 10 *iterations* (1 *iteration/batch*) for 10,000 training inputs

The sonar dataset we are using for this project is relatively small, at only 208 input rows, so it should be possible to avoid splitting the training set into batches. In this case, 1 epoch would equal 1 iteration for all the training inputs.

The number of epochs to use is arbitrary, and there is no magic number of epochs that gives optimal performance. Too few epochs will result in **underfitting**, while too many epochs may result in **overfitting**. To understand underfitting and overfitting, imagine trying to draw an optimal line through a distribution of points on a graph that best "fits" the majority of the points. With *underfitting*, the line only passes through a small number of points, and the line's equation will not reliably predict the y-coordinate from a given x-coordinate. To visualize *overfitting*, imagine a curved line that accurately hits every point in the training set, but snakes through the spaces between the points. A network that has been overfitted will perform very well on the training set inputs, but will tend to be less accurate for inputs it has never seen before. If you were to graph the accuracy of the perceptron vs. the number of epochs, you would most likely see initially low accuracy that improves as the number of epochs increases, but eventually plateaus, after which point continuing to increase the number of epochs will result in accuracy that decreases, or at best stays at the plateau.

Step 7: Test the trained perceptron

Once the perceptron has been trained, it is ready for use. At this point it should be able to produce the desired outputs for all the inputs in the training set, but its performance on inputs that have never been seen before is still in question. It is usually good practice to approach testing, or **validation** as it is often called, in a layered fashion, as follows:

1. **Test only the training set**
   Testing only the training set serves to confirm the perceptron can accurately predict outputs for the inputs in the training set. If the accuracy here is low, there is little point in testing further, as this indicates the perceptron needs further training. If the accuracy is high, proceed with testing on unseen inputs.

2. **Test the entire dataset (training set + unseen inputs)**
   Assuming the perceptron can accurately predict the outputs for the training set inputs, the next step is to test the perceptron on inputs it has not seen before. This can be done by either testing only unseen inputs, or more commonly, the entire dataset.

Step 8: Tweak, re-train, re-test

Depending on how well the perceptron performs during testing, it may be necessary to make adjustments to the number of epochs, the value of the learning rate parameter, and/or the value of the bias, in order to try to improve the perceptron's performance. It may also be a good idea to select different inputs for the training set, as some subsets of inputs can give better results than others. There is never any guarantee what the upper bound for accuracy will be for a given perceptron on a given dataset. It may be be possible to achieve >99% accuracy, or the accuracy may plateau at say, 75%, no matter what adjustments are made.

Perceptron Implementation

I will leave the implementation details of your perceptron to you, but I strongly recommend that you model your perceptron using the components we have covered in lecture. These components are:

The set of synaptic weights

You will need 60 weights to correspond to the 60 input values for each row of input data, so it is advisable to store the weights in a list as opposed to 60 individual variables.

The bias value

I would recommend storing the bias in its own variable, as opposed to just adding it to the list of synaptic weights.

Function for computing the weighted sum of the inputs

This function simply multiplies each synaptic weight by its corresponding input, and returns the sum of all the products.

Function for computing the induced local field

This function adds the bias value to the weighted sum of the inputs, and returns this sum. Often this is included as part of the function for computing the weighted sum of the inputs, but it's a good idea to decouple the two computations for the sake of modularity.

Activation function

This function quantizes the value of the induced local field so the perceptron will produce a binary output, either 0 or 1. In the case of a perceptron, the two possible return values must match the integer values used to represent the two possible classes of inputs.

Predict function

I'm not sure whether "predict" is the best name for this function, but it is commonly used so I will stick with it. This function just takes an input vector (a row of 60 inputs for the sonar dataset), calculates the sum of the weighted inputs, adds the bias to produce the induced local field, then passes the result to the activation function and returns the result; e.g., either a 0 or a 1.

Train function

This function should accept a training set, the learning rate parameter, and the number of epochs to use. The function needs to do the following:

1. Reset the synaptic weights (usually to 0.0) to erase the perceptron's memory so it can be trained anew.

2. For each epoch:
    For each row in the training set:
        Predict the output for the current row
        Calculate the error signal; i.e., deviation from the desired output
        Adjust the synaptic weights, and bias, accordingly*

The asterisk marking the adjustment step means how you do this is arbitrary. You can decide by how much to adjust the synaptic weights given the error signal, and optionally you can also adjust the value of the bias. You will need to train your perceptron under varying conditions until you find a set of conditions that works best.

Technically, the train function does not have to be part of the perceptron itself. If you wish, you can place all the training logic in your test harness.

Test function

This function just feeds the perceptron all the input vectors in the dataset and returns the result. No adjustments are made and no epochs are involved, the perceptron is simply predicting the output for each of the inputs. As I mentioned, you can include only the training set here to get an idea of how accurate the perceptron is on the training set. But the main focus will be to feed the perceptron input vectors it has not seen before to see how accurate it is based on the training it has received. As with the train function, this function does not have to be part of a perceptron itself, it can be part of the test harness. Either way, it should return useful information about the test, such as the number of correct predictions, so the accuracy can be calculated.

Test Harness Implementation

For those of you who may not be familiar with the term, "test harness", a test harness is simply a component that serves to feed inputs to a model (e.g., a perceptron), retrieve the outputs, and display useful information about the results. A test harness is also a convenient component to include code for loading data from data files, doing any pre-processing (e.g., converting strings to floats), and initializing all the parameters before creating the model. Test harness code is often included along with the model code in the same file, but I discourage this practice for the sake of modularity. The model should not need to know anything about how it is tested, and the test harness should not have direct access to the model, but rather just the model's public interface. That said, for this project you may include your test harness code in a separate file or in the same file as your perceptron, whichever you prefer.

Programming Language

Your perceptron should be implemented in Python, even though this project can be completed without using any third-party Python libraries. Later projects will require the use of Python, so this is an excellent project to get you started. In the last part of the Python primer we introduced the concept of classes, but did go into details about what a Python class looks like. I will introduce the basics of writing Python classes here, using the perceptron for this project as an example.

A `Perceptron` Class

As we mentioned, a perceptron can be modeled as having a number of components, as listed previously, and summarized below:

- A collection of synaptic weights
- Bias value
- Weighted sum function
- Induced local field function
- Activation function
- Predict function
- Train function
- Test function

How best to implement these components depends on what type of problem is being solved, whether or not reusability is important, and design preferences. I will offer one possible design, but you are free to use it, adapt it, or ignore it and come up with your own design.

Looking at the above list, our Perceptron class will have 2 attributes, the set of synaptic weights and the bias value; and 6 functions. We know each input vector from the sonar dataset contains 60 input values, so we will need 60 synaptic weights, one per input in the intput vector. For such a large number of weights, storing them in a collection of some sort would seem to be the best option. Python's list data structure would serve well for this. To allow the greatest flexibility, it wouls be best to assume the synaptic weights will be `float` values.

The bias value is a single value, so it can be stored separately from the synaptic weights. Alternatively, it can be inserted at the beginning of the collection of synaptic weights, as the lecture from Module 3 mentioned. I prefer to keep the bias separate, but feel free to add it to the collection of synaptic weights if you would like. As with the synaptic weights, to allow the most flexibility it would be wise to make the bias a `float` value.

For the functions, we need to decide on the arguments and return types. The function that computes the weighted sum of the inputs needs to be given an input vector as an argument. As with the synaptic weights, a list would be appropriate here, since each input vector will have 60 values. Care must be take to ensure that the indexes into both lists are synchronized, in order to correctly map each synaptic weight to the input it receives. This function should return the sum of all the inputs multiplied by their corresponding synaptic weights. This should be a `float` value.

The function that calculates the value for the induced local field simply adjust the sum of the weighted inputs by the bias value. Therefore, this function needs the sum of all the weighted inputs and the bias value as inputs, either directly or indirectly. The return value should be a `float` value.

The activation function quantizes the value of the induced local field to ensure the output of the perceptron is always one of two possible values. This function will need to determine where the cutoff is for the induced local field such that any induced local field value above or equal to the cutoff returns one of the two possible outputs, while any induced local field value below the cutoff returns the other output. It is common to use 0 and 1 as the two possible outputs, but any two unique values may be used. The values should be integers, though.

The predict function is the function that best encompasses the behavior of the perceptron. It takes a single input vector as an argument, and returns the value of the activation function after the inputs have been processed. Once the perceptron has been sufficiently trained and tested, this will be the primary function that is called when the perceptron is actually used.

The train function, as mentioned previously, may be included either as part of the perceptron or as part of the test harness. Either approach will work. For the design I am proposing I will make the train function part of the perceptron. This function needs to be given a training set consisting of an aribitrary number of input vectors, the learning rate parameter, and the number of epochs to use for training. The implementation of this function is critical, as this will be where the perceptron attempts to predict the output for a given input vector, and adjust its synaptic weights as needed to make the perceptron converge towards being able to correctly classify inputs. A return value here is not strictly required if the perceptron's synaptic weights are modified in place, which can be done if the train function is part of the perceptron itself. If the train function is placed externally to the perceptron, it will need to return the collection of adjusted synaptic weights so the adjustments can be passed on to the perceptron.

The test function may also be included either as part of the perceptron, or it can be placed externally in a test harness. The argument for this function is a dataset. This can be the entire dataset, only the training set, or only the dataset minus the training set inputs. It is not necessary to pass the learning rate parameter or the number of epochs since no synaptic weight adjustments will be made by this function. Since the purpose of this function is to determine the accuracy of the perceptron, especially with respect to inputs that were not part of the training set, this function should return some useful value, such as a collection of the actual outputs for all the input vectors. The actual outputs can then be compared to the desired outputs, and the accuracy of the perceptron can be calculated as:

$$accuracy = \#\ correctly\ predicted\ outputs\ /\ \#\ of\ input\ vectors\ in\ the\ dataset\ *\ 100\%$$

The next page shows skeleton code for a Perceptron class containing the components we just described. I've also included this as a Python script, `Perceptron.py`.

```
class Perceptron(object):

        # Create a new Perceptron
        #
        # Params:     bias - arbitrarily chosen value that affects the overall output
        #                    regardless of the inputs
        #
        #             synaptic_weights - list of initial synaptic weights
        def __init__(self, bias, synaptic_weights):
                self.bias = bias
                self.synaptic_weights = synaptic_weights


        # Activation function
        #      Quantizes the induced local field
        #
        # Params:     z - the value of the indiced local field
        #
        # Returns:    an integer that corresponds to one of the two possible output
        #             values (usually 0 or 1)
        def activation_function(self, z):



        # Compute and return the weighted sum of all inputs (not including bias)
        #
        # Params:     inputs -      a single input vector (which may contain multiple
        #                           individual inputs)
        #
        # Returns:    a float value equal to the sum of each input multiplied by its
        #             corresponding synaptic weight
        def weighted_sum_inputs(self, inputs):


        # Compute the induced local field (the weighted sum of the inputs + the bias)
        #
        # Params:     inputs -      a single input vector (which may contain multiple
        #                           individual inputs)
        #
        # Returns:    the sum of the weighted inputs adjusted by the bias
        def induced_local_field(self, inputs):


        # Predict the output for the specified input vector
        #
        # Params:     input_vector - a vector or row containing a collection of
        #             individual input values
        #
        # Returns:    an integer value representing the final output, which must be one
                      of the two possible output values (usually 0 or 1)
        def predict(self, input_vector):
```

```
    # Train this Perceptron
    #
    # Params:     training_set - a collection of input vectors that represents a
    #                            subset of the entire dataset
    #
    #             learning_rate_parameter -  the amount by which to adjust the
    #                    synaptic weights following an incorrect prediction
    #
    #             number_of_epochs -  the number of times the entire training set is
    #                                 processed by the perceptron
    #
    # Returns:    no return value
    def train(self, training_set, learning_rate_parameter, number_of_epochs):


    # Test this Perceptron
    # Params:     test_set -    the set of input vectors to be used to test the
    #                           perceptron after it has been trained
    #
    # Returns:    a collection or list containing the actual output (i.e.,
    #             prediction) for each input vector
    def test(self, test_set):
```

There are several important things to note from this code. The "`object`" reference in the first line indicates the `Perceptron` class inherits from the base `object` class. This can be actually be omitted in this case, since all classes automatically inherit form the `object` class.

The `__init__` function is inherited by all Python classes, and it can be overridden to provide the desired initialization for a custom class. It is analogous to the constructor method in Java. In this case we are initializing the Perceptron class with values for the bias and the synaptic weights. Class attributes in Python are not declared as they are in a language such as Java. Instead, they are declared at the time of first use, and the attribute name must be preceded by the word, "**`self`**". Use of **`self`** distinguishes an attribute variable from a local function variable. **`self`** refers to the class instance itself.

Regarding the functions, note that all function declarations must begin with the keyword, **`def`**, which is shorthand for "define". We also encounter the word **`self`** again. For classes, each function must have "**`self`**" as the first argument. This is important, as **`self`** must be used to when a class calls its own functions or accesses its own attributes. It is similar to the `this` keyword in Java, except its use is not optional. If you try to have a class call one of its own functions or access one of its own attributes without using **`self`**, you will get an error.

The Test Harness

The test harness for this project is responsible for reading the dataset from a file, converting the string values from the file to the appropriate data type (e.g., float or int), and creating the training set. This script also instantiates a Perceptron, initializes the values needed for training, then trains the perceptron. After training, the script should test, or validate the perceptron by feeding the perceptron the entire dataset. Alternatively, the perceptron can be tested separately on the training set and the dataset inputs that were not part of the training set, in order to see how well the perceptron did against the known vs. the unknown inputs. The final number of correct and incorrect outputs should be displayed, along with the calculation of the perceptron's accuracy. I have provided a skeleton test harness script, `Perceptron_Tester.py`, you may use or adapt to suit your needs. I have included a function for reading the dataset from the CSV file for convenience, along with other accessory functions for manipulating the types of the data values. You are left with writing code to create the training set, as well as instantiating the Perceptron, training it, and testing it.

You will need to choose values for the learning rate parameter and the number of epochs to use. I suggest starting with a learning rate parameter of 0.01 and a small number of epochs (e.g., 500). See if you can tweak these values to obtain higher accuracy, paying attention to the possibility of overfitting.

What You Need to Turn In

Code:

Since the implementation details of both the Perceptron class and the test harness are being left to you, you will need to submit both your Perceptron class script and your test harness script. They can be in separate files or both can be in a single file, either way is fine.

Results:

You also need to submit a brief summary of your results consisting of the best accuracy you were able to achieve, and the values you chose for the learning rate parameter and the bias. Also, briefly describe how you chose the training set you used and the number of epochs you used for training.

Point Distribution

I have based the point distribution on the design I have proposed above. As long as your perceptron performs all of these functions correctly, you will get the points for those functions. it does not matter whether you have each function implemented separately or not. The train function will be the most challenging to implement, and is the most critical function of the perceptron, so it is worth significantly more points.

The goal is to get your perceptron to function consistently with at least 70% accuracy. The baseline accuracy prior to any training at all is approximately 53%. This is simply because there are only 2 possible classes of inputs, so if the perceptron always outputs a 0 or always outputs a 1, it will by chance correctly classify however many inputs correspond to the perceptron's default output. I will award an additional 5 points extra credit if you can get your perceptron to correctly classify at least 95% of the inputs.

Item	Points Possible	Points
Computation of sum of weighted inputs	3	
Computation of induced local field	3	
Activation function	3	
Predict function	3	
Train function	15	
Test function	3	
Perceptron works and achieves >= 70% accuracy (*+5 points extra credit if you can get the accuracy to >= 95%*)	10	
Total:	**40**	