Assignment 6

B351 / Q351

Due:
Friday, April 17th, 2020 @ 11:59PM

1 Summary

- Gain a basic understanding of artificial neural networks
- Implement the Hopfield update rules, basic vector operations, the perceptron learning algorithm, and the backpropagation algorithm for multi-layer perceptron learning

We will be using Python 3, so be sure you arent using older versions. Code that will compile in Python 2.7 may not compile in Python 3. See our installation guide for help removing old versions and installing Python 3.

To run your program, you will need both the `numpy` and `pandas`. Although `numpy` should already be installed from Assignment 5, it will be automatically installed with `pandas`. Instructions to install `pandas` can be found here.

Please submit your completed files to your private Github repository for this class. You may not make further revisions to your files beyond the above deadline without incurring a late penalty.

This assignment must be completed **individually**. You must submit your own files to your own repository. Any shared code will not be tolerated and those involved will be subjected to the university's cheating and plagiarism policy. **If you discuss ideas, each person must write a comment at the top of the assignment file naming the students with whom ideas were shared.**

2 Background

This assignment covers three different types of artificial neural networks. General information about artificial neural networks can be found here.

2.1 Hopfield Networks

Hopfield networks are a special form of artificial neural networks known as recurrent neural networks. Recurrent neural networks are special in that they allow prior outputs of the network to be used as inputs to some or all of the network. This website provides a good general overview of them. Hopfield

networks have only one layer, and within that layer, all the nodes are fully connected (excluding a connection from a node to itself). Diverging from the majority of other aritfical neural networks, Hopfield network training updates the value at each node, as opposed to the weights in between the nodes. Common uses for them generally involve some sort of pattern recognition (such as digit recognition seen in the lecture slides). This website provides good information regarding the training/update rules for these networks.

2.2 Perceptron

Perceptrons by themselves do not represent a complete artifical neural network, but instead represent a single neuron unit. By themselves, they are simply a linear classifier. As seen in lecture, any data that is linearly separable can be learned by a perceptron. The lecture 16 slides provide very good information on how perceptron learning works and are the best starting point for this portion of the assignment.

2.3 Multi-Layer Perceptron

Multi-layer perceptrons are a type of artificial neural network known as a feedforward network. In this type of network, data only moves in one direction and no cycles exist within the network, differentiating them from recurrent networks (such as the Hopfield network). At a minimum, multi-layer perceptron networks consist of 3 layers: input layer, hidden layer, and output layer. There are no weights to learn in the input layer. The main advantage of a multi-layer over a single perceptron is that the multi-layer can learn data that is not linearly seperable. In fact, the universal approximation theorem tells us that a huge number of functions can be learned by a multi-layer perceptron network with only **one hidden layer**.
Learning data in a multi-layer perceptron network inevitably presents some issues that are not present in a single perceptron. Since there is no longer just one perceptron with weights to update, how the weights of any given node in the network affect the final result is much different from the singular perceptron case. The algorithm used to overcome this obstacle is known as backpropagation. The derivation of this method requires some understanding of multivariate calculus. However, this resource provides a thorough explanation that lowers the barrier somewhat. A careful review of the explanation provided there will help immensely for the implementation in this assignment.

3 Programming Component

3.1 Data Structures

You will utilize the following classes. You should read and understand this section **before** embarking on the assignment.

3.1.1 HopfieldNetwork Class

This class is in the `hopfieldnetwork.py` file.It encapsulates the following data members:

- `nodes` - List of numbers representing the nodes of the network.
- `weights` - 2-d list representing the weights in between the nodes. Weights of i,j is weight between nodes i and j.

The following are HopfieldNetwork's provided object methods:

1. `__init__(self, num_nodes=None, start_nodes=None, target_stable=None)` - constructor for the HopfieldNetwork class.
2. `update_node(self, node)` - Updates the node at index given (node argument) according to the Hopfield Network update rule.
3. `cycle_until_stable(self)` - Updates all nodes in ascending order until a stable state is reached. Uses `update_node`.
4. `print(self, print_weights=false)` - Prints the nodes of the hopfield network. Optionally prints weights according to `print_weights` argument.

3.1.2 Perceptron

This class is in the `perceptron.py` file. This class contains all the needed methods for classification using the perceptron algorithm, as well as methods to perform training. It encapsulates the following data members:

- `inputs` - 2-d list of values representing the training input data.
- `targets` - List of values representing the target classifications for the training data.
- `input_size` - Number of attributes for each input instance.
- `output_size` - Length of `targets`.
- `num_inputs` - Number of training inputs provided.
- `weights` - Weights of the perceptron of length `input_size` + 1.

The following are Perceptrons's provided object methods:

1. `__init__(self, inputs, targets)` - constructor for the Perceptron class.
2. `train(self)` - Trains perceptron using `train_sample`. Returns weights.
3. `train_sample(self, input, target)` - Performs one interation of training using provided input and target. Returns true if weights are updated, false otherwise.

4. `predict(self, input)` - Classifies a given input using the network.
5. `activation(self, n` - Applies threshold activation function to `n` and returns result.
6. `dot(self, a, b)` - Returns dot product of vectors `a` and `b`.
7. `add(self, a, b` - Returns `a` + `b` (vector addition).
8. `sub(self, a, b)` - Returns `a` - `b` (vector subtraction).

3.1.3 MLP

This class is in the `backpropgation.py` file. It represents a multi-layer perceptron network and contains methods for both training and classification. It encapsulates the following data members:

- `eta` - learning rate of the network.
- `num_layers` - number of layers in the network.
- `input_size` - number of attributes for each input.
- `output_size` - number of output nodes.
- `params` - dictionary containing weights and activation function for each layer.

The following are MLP's provided object methods:

1. `__init__(self, nn_architecture, seed=0, eta=0.5)` - Constructor for the MLP class.
2. `initialize(self, nn_architecture, seed=0)` - Initializes network parameters according to `nn_architecture`.
3. `feed_forward(self, inputs)` - Feeds the provided inputs through the network and returns the predictions for them.
4. `back_propogation(self, targets, activations)` - Performs back-propagation algorithm for updating weights according to the error between the target and activation values.
5. `calc_output_layer_delta(self, activation, targets)` - Calculates the derivative of the error w.r.t. the output values.
6. `calc_layer_delta(self, weight, activation, delta)` - Calculates the derivative of the error w.r.t. the output of the specified layer.
7. `update_layer_weights(self, layer, gradient)` - Update the weights of the specified layer using the gradient and learning rate (`eta`) of the network.

8. `update_layer_bias(self, layer, gradient)` - Update the bias of the specified layer using the bias gradient and learning rate (`eta`) of the network.

9. `calc_layer_gradient(self, activation, delta)` - Calculates the gradient of the weights based on the activation values and delta associated with the weights.

10. `train_network(self, epochs, train)` - Trains the network using iterative SGD (weight update for each training point) on the provided training data (`train`) for the specified number of epochs.

11. `test_network(self, test)` - Calculates the accuracy of the network on the provided testing data (`test`).

3.1.4 Other Functions

These functions are not contained in within the `MLP` class, but are present in the `backpropagation.py` file.

1. `five_fold_cross_val(nn_architecture)` - Performs 5-fold cross validation on the specified network architecture. Returns the mean accuracy.

2. `sigmoid(x)` - Applies the sigmoid activation function to `x`. Returns the result.

3. `softmax(x)` - Applies the softmax activation function to `x`. Returns the result.

3.2 Objective

Your goal is to provide the implementations to the following functions:

1. `HopfieldNetwork.update_node`

2. `HopfieldNetwork.cycle_until_stable`

3. `Perceptron.train`

4. `Perceptron.train_sample`

5. `Perceptron.predict`

6. `Perceptron.activation`

7. `Perceptron.dot`

8. `Perceptron.add`

9. `Perceptron.sub`

10. `MLP.back_propagation`
11. `MLP.calc_output_layer_delta`
12. `MLP.calc_layer_delta`
13. `MLP.update_layer_weights`
14. `MLP.calc_layer_gradient`

to the following specifications:

3.2.1 HopfieldNetwork.update_node(self, node)

Write a function to update the value of a specified node according to the hopfield update rule. Returns true if the value at the node changed, false otherwise.

- `node`: index of the node to update

3.2.2 HopfieldNetwork.cycle_until_stable(self)

Write a function to sequentially update the values in the network in ascending order (0, 1,...,`len(self.nodes-1)`) until the network has reached a stable state. A stable state is one where node values have converged and any further training does not change them. This function should call `HopfieldNetwork.update_node`.

3.2.3 Perceptron.train(self)

Write a function to train the perceptron according to the perceptron learning algorithm. Using the `Perceptron.train_sample` method, the network should train on each data point in the data set until all points are correctly classified.

3.2.4 Perceptron.train_sample(self, input, target)

Write a function that performs the perceptron training algorithm step for a single data point. This consists of:

1. Calculate the output for this sample using `Perceptron.dot`
2. Apply the activation function to the output
3. If the prediction is correct, return False
4. Otherwise, update the weights in the network using either `Perceptron.add` or `Perceptron.sub` and return True

- `input & target`: input data and target prediction for the data

3.2.5 Perceptron.predict(self, input)

Write a function that predicts the class of a data point. To do this, use `Perceptron.dot` with the input data and network weights as well as `Perceptron.activation`. Returns the prediction.

3.2.6 Perceptron.activation(self, n)

Write a function that applies the perceptron threshold function to `n`. Returns the result (1 or 0).

- `n`: input value

3.2.7 Perceptron.dot/add/sub(self, a, b)

Write a function that accepts two vectors and returns:

1. `dot`: the dot product of `a` and `b`
2. `add`: the element-wise addition of `a` and `b`
3. `sub`: the element-wise subtraction of `a` and `b`

The vectors must have the same dimension for each operation

- `a` and `b`: input vectors

3.2.8 MLP.backpropagation(self, targets, activations)

Write a function that implements the backpropagation algorithm. This is done as follows:

1. Calculate $\frac{\partial Err}{\partial out}$ (the change in error with respect to the output of the network) by applying the derivative of the sigmoid to `targets` and `activations`. Use `MLP.calc_output_layer_delta`
2. Calculate $\frac{\partial Err}{\partial w}$ where w is the weights of the output. This can be interpreted as the change in the error with respect to the weights of the output layer. To do this, the activation values from the layer preceding the output and $\frac{\partial Err}{\partial out}$ from step 1 should be used with `MLP.calc_layer_gradient`
3. Calculate $\frac{\partial Err}{\partial b}$ where b is the bias of the output. There is a single bias value for each node in a given layer. To calculate $\frac{\partial Err}{\partial b_i}$ (where b_i is the bias for node i in a given layer), you calculate the average of $\frac{\partial Err}{\partial w_i}$ (where w_i are the weights for node i in a given layer)
4. Using $\frac{\partial Err}{\partial w}$ and `eta`, update the output layer's weights using `MLP.update_layer_weights`

5. Using $\frac{\partial Err}{\partial b}$ and `eta`, update the output layer bias using `MLP.update_layer_bias`

6. Starting with the layer preceding the output, repeat the following in reverse order for each layer:

   (a) Calculate $\frac{\partial Err}{\partial out}$ (the change in error with respect to the output of the **current layer**). To do this, use the weights of current layer + 1, the activation values of the current layer, and $\frac{\partial Err}{\partial out}$ from current layer + 1. For example, if the calculation is being done for for the 3rd layer of some network, the weights and $\frac{\partial Err}{\partial out}$ from layer 4 would be used with the activation values from layer 3. Be sure to not discard $\frac{\partial Err}{\partial out}$ after performing the weight update, since it must be reused. Use `MLP.calc_layer_update`

   (b) Calculate $\frac{\partial Err}{\partial w}$ (the change in error with respect to the weights of the **current layer**. This is done by using $\frac{\partial Err}{\partial out}$ and the values that were inputted to the current layer (which would be the activation values of current layer - 1). For example, for the first layer, the initial input values would be used. For the 3rd layer, the output of the second layer would be used. Use `MLP.calc_layer_gradient`

   (c) Calculate $\frac{\partial Err}{\partial b}$ for the current layer using the same method as step 3

   (d) Update the weights of the current layer. Use $\frac{\partial Err}{\partial w}$ and `eta` to update the weights for the current layer.

   (e) Using $\frac{\partial Err}{\partial b}$ and `eta`, update the current layer bias using `MLP.update_layer_bias`

- `targets`: target classification for each data point
- `activations`: activation values from output layer

3.2.9 MLP.calc_output_layer_delta(self, activation, targets)

Write a function the calculates and returns $\frac{\partial Err}{\partial out}$ for the output (step 1 in `MLP.backpropagation`), using the network output and the expected outputs. The equation for this is $(a - t)$

- `activation`: activation values from the output layer
- `targets`: target classifications for the inputs

3.2.10 MLP.calc_layer_delta(self, weight, activation, delta)

Write a function that calculates $\frac{\partial Err}{\partial out}$ for any non-output layer. The equation for this is: $(\text{weight}^T \cdot \text{delta}) \times \text{activation} \times (1 - \text{activation})$

- `weight`: weights from current layer + 1

- `activation`: activation values from current layer
- `delta`: $\frac{\partial Err}{\partial out}$ from current layer + 1

3.2.11 MLP.update_layer_weights(self, layer, gradient)

Write a function that updates the weights for a layer using $w_{i+1} = w_i - (\eta \times \frac{\partial Err}{\partial w})$ (η is `eta`)

- `layer`: String in format of "Wi", where i is the layer number (0 indexed)
- `gradient` $\frac{\partial Err}{\partial w}$ for the current layer

3.2.12 MLP.update_layer_bias(self, layer, gradient

Write a function that update the bias for a layer using $w_{i+1} = w_i - (\eta \times \frac{\partial Err}{\partial b})$ (η is `eta`)

- `layer`: String in format "Bi", where i is the layer number (0 indexed)
- `gradient`: $\frac{\partial Err}{\partial b}$ for the current layer

3.2.13 MLP.calc_layer_gradient(self, activation, delta)

Write a function that calculates $\frac{\partial Err}{\partial w}$ (the gradient) for layer corresponding to `delta`. The equation for this is $\frac{\partial Err}{\partial out} \cdot (\text{activation})^T$

- `activation`: input to the current layer, which are the activation values of current layer - 1
- `delta`: $\frac{\partial Err}{\partial out}$ for the current layer

4 Grading

Problems are weighted according to the percentage assigned in the problem title. For each problem, points will be given based on the criteria met in the corresponding table on the next page and whether your solutions pass the test cases on the online grader.

In addition, the students in the tool-assisted group will have their solutions assessed for clarity (15 points), control flow and time complexity (5 points), and making use of clean and suitable syntax features (5 points).

Finally, remember that this is an individual assignment.

4.1 HopfieldNetwork.update_node (20%)

4.2 HopfieldNetwork.cycle_until_stable (20%)

4.3 Perceptron.train (15%)

4.4 Perceptron.train_sample (15%)

4.5 Perceptron.predict (5%)

4.6 Perceptron.activation (2%)

4.7 Perceptron.dot (1%)

4.8 Perceptron.add (1%)

4.9 Perceptron.sub (1%)

4.10 MLP.backpropogation (20%)