Payton Howell May 2021

Bio-Inspired Artificial Intelligence

Matrix-Form Back Propagation Network

Background and Goals

For this project, my motivation came from a need to understand the mathematics behind the back propagation algorithm. After learning about the matrix forms of forward and backward propagation, I wanted to code a network that would utilize the NumPy library to do the matrix calculations behind these processes. I wanted this code to be able to handle any size network needed and be able to correctly handle back propagation to get to a set of weights that accurately classifies the data.

Methodology

This code creates a neural network capable of back propagation- allowing the user to create a network capable of being trained to any data the user wants to fit the network to. Inputs to the created network must be in the form of a NumPy array with N rows of 1 input each [N x 1]. This allows the network to correctly propagate the information to produce an output of Y x 1. The system uses sigmoid activation functions for both layers in the network, as it is straightforward to produce the delta matrices of these functions.

The main functions of this class are the self.propagate() and self.adjust_weights() functions. The propagate function goes through forward propagation, while the adjust_weights function goes through back propagation to find and then apply the weight changes necessary to increase the accuracy of the program. These two functions are utilized to the fullest extent in the self.train() function which performs both to train the network on the patterns and targets entered.

Other functions also aid in the process of training such as the self.error() function which takes in sets of patterns and targets and returns the sum squared error and the fraction of outputs within tolerance of the target. For patterns with several outputs, the output in question is within tolerance only if all outputs are within tolerance of the targets.

While coding, my goal was to keep everything generalized to ensure any size neural network could be created with the code. This meant that I had to test several different sized networks to ensure it runs correctly and can handle networks of any size. Using sets of weights I knew would match the targets of a 2-2-1 network trying to approximate the AND operation, I was able to ensure the network processed inputs correctly through propagation. I also used that set of weights to test back propagation to ensure the weight changes caused an increase in accuracy with each epoch.

Creating larger networks was more difficult for testing cause I had to ensure not only my code could handle networks of these sizes but also that the functions were being accomplished correctly. Several times propagation would occur yet the network would not be learning the targets or improving upon them.

Results and Conclusions

The most difficult aspect of this project was testing and refining the code. Figuring out what went wrong and why was definitely the part of this that kept me on my toes even once I was pretty confident in it working. Once I tested the propagate function and was sure it worked, coding the adjust-weight function took a lot of work. This required making sure all the functions were using the correct type of matrix multiplication- especially because there's a huge difference between element-wise multiplication and matrix multiplication. In back propagation most functions require element-wise multiplication, making the times where matrix multiplication is used much more important to define.

Once I was confident it was working, using it on the data sets was straightforward, the hardest part was ensuring everything ran smoothly. Testing the number of epochs required by a network

to learn the data was also difficult as I did not have the network produce a history to be able to see how the network's accuracy changed over time. To learn a good limit, I instead had to work incrementally and run several tests until I found a number of epochs that produced a good accuracy on both the test and training data.

I first tested the network by applying the 2-2-1 network we coded during class- seeing how my code compared in the number of epochs required and ensuring it was able to learn weights both from an initialized state and after being trained on a different set of targets. To learn the AND operation the network took a comparable number of epochs in comparison to the 2-2-1 network created during class(both taking just over 800 epochs on average with a learning rate of 0.5).

I also tested the code with the MNIST data set, seeing if I could get a decent level of accuracy after training it over many epochs. In the end I trained the network over 180 epochs with a learning rate or 0.5 to an accuracy of 98.3% on the testing data and 95.4% on the testing data. This increase in the number of epochs from what was seen in class comes from the fact that the code updates the weights after propagating every pattern, rather than in randomized batches and does not include momentum. Adding momentum into the system and randomizing the order of the patterns as they enter the network, which will decrease the number of epochs necessary to train the network on data sets, is definitely something I would like to add.

In the future I'd also like to go back and make sure everything runs in the most efficient way and everything can be clearly understood in the code/structure of the program. This would include saving the accuracy of the network every epoch to be able to measure its change over time.

How to Run It

To create a network you must define a NeuralNetwork object, with the parameters being the number of inputs, hidden layer nodes, and outputs in that order. The learning rate and tolerance parameters can also be defined at this step. From here you receive a network with a randomized set of weights.

To train the network, the self.train() function needs the list of inputs and targets in the correct format as parameters. To limit the training to a maximum on the number of epochs, mx_Epochs, can be changed to cap the number of epochs the network will go through to that value. This value can also be set very high if the goal of the network is to get 100% of all patterns to within tolerance. This function will print out the number of epochs it has gone through and will adjust the weights as it runs.

To propagate a single input pattern, the self.propagate() function will run the entered pattern through the network's current set of weights and return the output as a np array vector. this will also update the variables holding the output and hidden layer activations (self.output and self.hidden, respectively) which can be called on for the most recent pattern propagated.

To reinitialize, self.initialize() must be called with the same parameters of number of inputs, hidden nodes, and outputs, and in the creation of the network. This will reproduce the two weight matrices of the network with random entries between -0.1 and +0.1 .

A Colab notebook with the code and some examples that can be run is at: Matrix-Form Colab Notebook

Link: https://colab.research.google.com/drive/1S24CSNP6F7bvddANqzcNVUTfIsw_RLYw?usp=sharing