Machine Learning Project 2: Design Document

Brett Layman, Kirby Overman, Carsen Ball

October 2nd, 2017

1 Description of the Problem

The goal of this project is to approximate the Rosenbrock function,

$$f(x) = f(x_1, x_2, ..., x_n) = \Sigma_{i=1}^{n-1}[(1 - x_i)^2 + 100(x_{i+1} - x_i^2])^2$$

when n = 2,3,4,5,6. We are going to implement a multi-layer feed forward neural network with back-propagation and a Radial Basis Function network (RBF). The feed forward NN will be able to be created with an arbitrary number of inputs, an arbitrary number of hidden layers (between zero and three), arbitrary number of nodes per hidden layer, and an arbitrary number of nodes in the output layer. All of these arbitrary numbers will be passed in as arguments on the start of our algorithm.

The RBF will have an arbitrary number of inputs, Gaussian Basis functions (hidden layer), and output neurons; much like the feed-forward net. For both networks we would like to note that in approximating the Rosenbrock function, we only need one output node to approximate the value of the function, y, given an input vector X. ε.

2 Software Architecture

2.1 Back-propagation

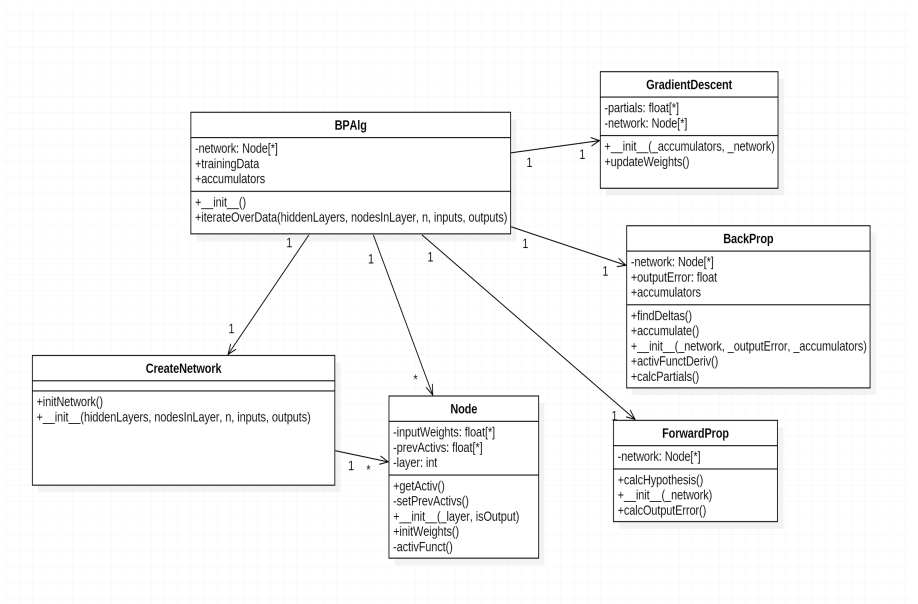
Our back-propagation algorithm is broken into 4 main processes controlled by 4 classes: CreateNetwork, ForwardProp, BackProp, and GradientDescent. The running of these processes is controlled by the class BPAlg. We also will use a Node class to store data about nodes in our network.

The CreateNetwork class initializes our network of nodes. This entails creating a two dimensional list of nodes, where each sublist represents a layer in the network, and each element in the sublist represents a node in that layer. When each Node is initialized, its input weights will be randomly initialized to a number in this range: [-.1,.1]. CreateNetwork will only be used once, as opposed to the other classes, which will be used multiple times until gradient descent converges.

ForwardProp is used to calculate our hypothesis, and the error of that hypothesis (our output delta). This will be accomplished by looping through each node in each layer, and making use of methods in our node class such as setPrevActivs() (which retrieves and assigns the activations in the previous layer to an instance variable), and activFunct(), which takes our weighted sum and passes it into our activation function. Our final output layer will not use the activation function, because we are trying to perform regression, not classification.

Backprop is used to calculate our hidden layers' delta values. It uses accumulators to create partial derivatives with respect to our weights, as we iterate over all of the points in our training dataset.

Finally, GradientDescent uses those partial derivatives to update our weights for each node in each layer, using simultaneous update. It then checks if all of the partial derivatives are approximately equal to zero, and if so, the algorithm is trained. If not, it repeats ForwardProp and BackProp with the new set of weights.



2.2 Radial-Basis

The radial basis algorithm has 4 classes controlled by the main class RadialBasis. The RadialBasis class will store the data points to be used in the algorithm and their expected outputs, the k Value which represents how many clusters we will be creating, and the number of output nodes that we will be using. The errors float will accumulate the error for each data point, and then be divided by the total number of data points used to train, to calculate the average error.

From the RadialBasis class the algorithm will initialize the kMeans class. The kMeans class will perform the kMeans clustering of the data in the cluster() function for a specified K. This function will create a list of μ values that represent the means of the clustered data. The calcSigma() will be called after the data has been clustered, to calculate the variance value for every hidden node to be used in the Gaussian function.

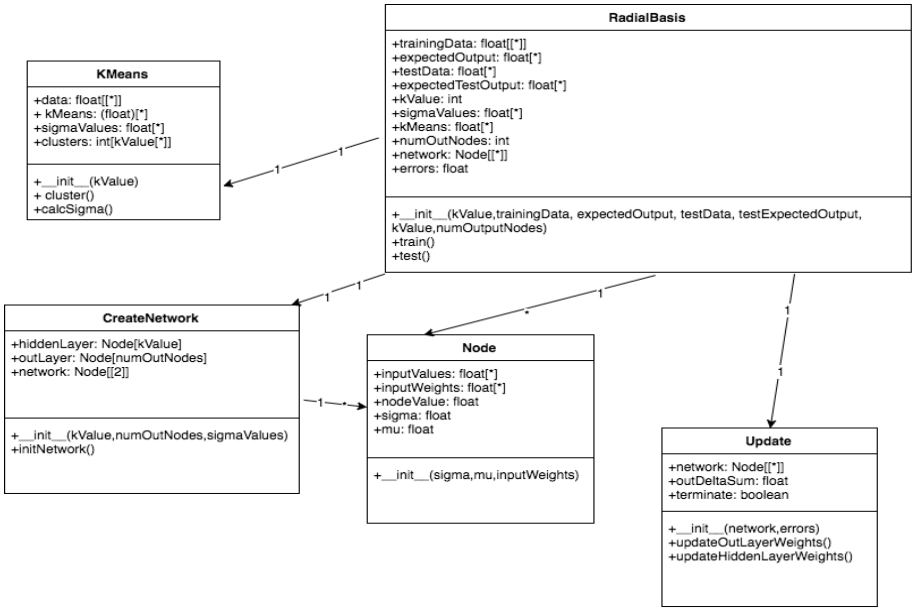
Once the data is clustered the algorithm will initialize the createNetwork() class. This will create kValue + numOutNodes instances of the node class. Each node will be initialized with a list of random weight values in the range:[-.1,.1], representing the incoming weights to the node. The hidden layer nodes will hold a sigma and mu value to be used for the Gaussian function. The network array will hold the hiddenLayer node list, and the outLayer node list.

The train() function, called from the RadialBasis class, will update the input values for each node in the hidden layer while the terminate value is false. From there it will calculate the value for each hidden layer node from the Gaussian function, with the

input into the Gaussian function being the input values · weights for the node. Once the hidden layer values are calculated, the output layer values will be calculated from the hidden layer values and corresponding weights. The train() function will then compare the sum of the output from the output layers, to the expected output for the point from the data using mean squared error. The mean squared error will then be added to the errors float. This process will be repeated for every data point in the training data set.

After every training data point has been processed, the Update will average the errors. The updateOutLayerWeights() function will calculate the delta values for each output node, and change the corresponding weights. The weight · delta for each node will be summed and held in outDeltaSum. The updateHiddenLayers() function will utilize the outDeltaSum to calculate the delta value for each hidden layer node. The weights will be updated correspondingly for each hidden layer node. If the terminating condition has been met, the boolean value terminate which is initialized as false, will be changed to true.

The test() function will be called once the network has been trained. It will use the test data set points to test the accuracy of the network.



3 Design Decisions

3.1 Back propagation Neural Network

In designing our Back-propagation implementation, we decided to split the algorithm into classes that corresponded to important inputs and outputs at each stage of the algorithm. ForwardProp produces the error (output delta). BackProp takes in the error and produces the partial derivatives of the weights with respect to the cost function. Finally GradientDescent takes in the partial derivatives, and alters the weights. Each of these major steps has sub-steps that can be represented as methods in those classes. Also, the steps can be repeated by having the driver class, BPAlg, reuse those classes. BPAlg is also useful for storing data across multiple steps, namely the network representation.

Network representation is an important part of creating an efficient algorithm. In this case, we chose to use a two dimensional list of nodes, where the first dimension corresponds to the layer the node is in, and the second corresponds to the index of the node in that layer. By creating a Node class, we were able to encapsulate a significant amount of information about the network in each node. For instance, we store the information about previous layer activations, and corresponding weights in each node, so that the nodes activation can be calculated and accessed by nodes in the subsequent layer. We think that this kind of modularized weight storage will make the weights easier to manage.

3.2 Radial Basis Neural Network

For the Radial Basis Neural Network we decided to take an object oriented approach. We wanted to have the network be created of instances of nodes. Because the hidden layer nodes and the output layer nodes will have different functions called, we put the two lists of nodes into a list. This will keep the layers separated, and easy to access the hidden nodes or the output nodes. Since we only have one hidden layer, we can create a list of size 2. We chose to keep the expected outputs separate from the data points to avoid issues when looping through the data points.

To be able to have unique sigmas corresponding the cluster, we decided to create a list of the σ values in the kMeans class. This list then is utilized when creating the hidden nodes. The node will hold the μ value from the cluster and the corresponding σ value. We chose to insert the values into each node to make accessing the values easy when calculating the overall node value. As well, the input values leading to the node, and the corresponding weights are held in lists in the node class to allow for easy access when calculating the Gaussian function value.

4 Experimental Design

For our experiment, we will be training the back-propagation and radial basis algorithms using a grid of points from the Rosenbrock function. Our grid will make use of exponentially greater intervals to capture different scales of the function. For example, using base two, we could set x1 = ... 2,4,8,16,32... The number of points along each dimension of the grid will depend on the grid size, because increasing the number of dimensions exponentially increases the number of points in the grid. Specifically we will set the number of points per dimension equal to: $P^{\frac{1}{d}}$, where P is the total number of points we want in the grid (probably around 10,000), and d represents the number of inputs ($|X|$), which corresponds to the number of dimensions.

For testing the algorithms, we will be randomly sampling within a range of values for each attribute. This range will extend beyond the grid size, to account for how well the algorithms apply outside of their training data range. For each of the test points we will compute the mean squared error between our hypothesis and the actual value of the function. We will then sum these errors to get a total error for that algorithm. To compare our two algorithms, we will use a two sided t-test with p value of .05. We will compare our algorithms across different numbers of inputs, different parameters of the backprop algorithm (number of layers, nodes per layer), and parameters of our RBF (number of Gaussians).

References

[1] Miroslav Kubat *An Introduction to Machine Learning* Springer. 2015. Chapter 5, Artificial Neural Networks.

[2] Rudolf Kruse & Christian Borgelt & Christian Braune & Sanaz Mostaghim & Matthias Steinbrecher *Computational Intelligence* Second Edition. Springer.2016

[3] Andries P. Engelbrecht *Computational Intelligence* Second Edition. John Wiley & Sons, Ltd. 2007.