Programming Test: Learning Activations in Neural Networks

Abstract—The choice of Activation Functions (AF) has proven to be an important factor that affects the performance of an Artificial Neural Network (ANN). Use a 1-hidden layer neural network model that adapts to the most suitable activation function according to the data-set. The ANN model can learn for itself the best AF to use by exploiting a flexible functional form, $k0 + k1 * x$ with parameters k0, k1 being learned from multiple runs. You can use this code-base for implementation guidelines and help.

My Report is on iris Dataset

Data

Before the actual implementation, we need to read and process our data. Here i am using "**iris.csv**" Dataset.

The data is providing the information about the 3 different types of flower here it is named as **"species"** and we have **four** pieces of data for each sample and the data set has 4 features (sepal length, sepal width, petal length and petal width).

By using Scikit Learn we are splitting the data as train, test and validation

- Here as the 1st step I am splitting the data set as train and test.
- Then the training data will be further split into train and validation.

Implementation

Here we are going to build a simple neural network that supports multiple layers and validation. The main function is Neural Network, which will train the network for the specified number of epochs. At first, the weights of the network will get randomly initialized by Initialize Weights. Then, in each epoch, the weights will be updated by Train and finally, every 20 epochs

accuracy both for the training and validation sets will be printed by the Accuracy function. As input the function receives the following:

- X_train, Y_train: The training data and target values.
- X_val, Y_val: The validation data and target values. These are optional parameters.
- epochs: Number of epochs. Defaults at 10.
- nodes: A list of integers. Each integer denotes the number of nodes in each layer. The length of this list denotes the number of layers. That is, each integer in this list corresponds to the number of nodes in each layer.
- lr: The learning rate of the back-propagation training algorithm. Defaults at 0.15.

The weights of the network are initialized randomly in the range [-1, 1] by Initialize Weights. This function takes as input nodes and returns a multi-dimensional array, weights. Each element in the weights list represents a hidden layer and holds the weights of connections from the previous layer (including the bias) to the current layer. So, element i in weights holds the weights of the connections from layer i-1 to layer i. Note that the input layer has no incoming connections so it is not present in weights.

For example, let's say we have four features (as is the case with the Iris dataset) and the hidden layers have 5, 10 and 3 (for the output, one for each class) nodes. Thus, nodes == [4, 5, 10, 3] Then, the connections between the input layer and the first hidden layer will be (4+1)*5 = 25. After augmenting the input with the bias (in this case the bias has a constant value of 1), the input layer has 5 nodes. By fully connecting this layer to the next (each node in the input layer is connected will every node of the hidden layer), we get that the total number of connections is 25. Similarly, we get that the connections between the first hidden layer and the second one will be (5+1)*10 = 60 and between the second hidden layer with the output we have (10+1)*3 = 33 connections.

In the implementation, numpy is used to generate a random number in the [-1, 1] range for each connection.

With the weights of the network at hand, we want to continuously adjust them across the epochs so that (hopefully) our network becomes more accurate. The training of the weights is accomplished via the popular (Forward) Back-

Propagation algorithm. In this technique, the input first passes through the whole network and the output is calculated. Then, according to the error of this output, the weights of the network are updated from last to first. The error is propagated *backwards*, hence the name of the titular algorithm. Let's get into more detail about these two steps:

Forward Propagation:

- Each layer receives an input and computes an output. The output is computed by first calculating the dot product between the input and the weights of the layer and then passing this dot product through an activation function (in this case, the sigmoid function).
- The output of each layer is the input of the next.
- The input of the first layer is the feature vector.
- The output of the final layer is the prediction of the network.

Backward Propagation:

- Calculate error at final output.
- Propagate error backwards through the layers and perform corrections.
    - Calculate Delta: Back-propagated error of current layer *times* Sigmoid derivation of current layer activation.
    - Update Weights between current layer and previous layer: Multiply delta with activation of previous layer and learning rate, and add this product to weights of previous layer.
    - Calculate error for current layer. Remove the bias from the weights of the previous layer and multiply the result with delta to get error.

In our implementation we will pass each sample of our dataset through the network, performing first the forward pass and then the weight updating via the back-propagation algorithm. Finally, the newly calculated weights will be returned.

Neural networks need an activation function to pass the dot product of each layer through to get the final output (as well as to get to get the delta in back-

propagation). In this tutorial, we will use the sigmoid function and its derivative. Other activation functions are available, like the famous ***ReLU***. Also, sometimes layers don't use the same activation function, and there are times where the output doesn't have an activation function at all (for example, in the case of regression).

When we want to make a prediction for an item, we need to first pass it through the network. The output of the network (in the case of three different classes, as in the Iris problem) will be in the form [x, y, z] where x, y, z are real numbers in the range [0, 1]. The higher the value of an element, the more confident the network is that it is the correct class. We need to convert this output to the proper one-hot format we mentioned earlier. Thus, we will take the largest of the outputs and set the corresponding index to 1, while the rest are set to 0. This means the predicted class is the one the network is most confident in (i.e. the greatest value).

So, a prediction involves the forward propagation and the conversion of the output to one-hot encoding, with the 1 denoting the predicted class.

Finally, we need a way to evaluate our network. For this, we will write the Accuracy function which, given the computed weights, predicts the class of each object in its input and checks it against the actual class, returning the percentage of correct predictions.

Instead of the percentile accuracy, other accuracy metrics can be employed, but for this tutorial this simple method will do.

Submitted by:

Kirtika . V