Prometheus Project Democratic Algorithm Proposal

Shadman Sakib Asraf
June 11, 2020

1 Objective

To implement a democratic algorithm that takes inputs from a stack of models (currently some combination of NNs and PCA algorithms) and outputs a set of percentages in decimal form. These percentages will correspond to the likeliness the algorithm believes that an image is a certain thing.

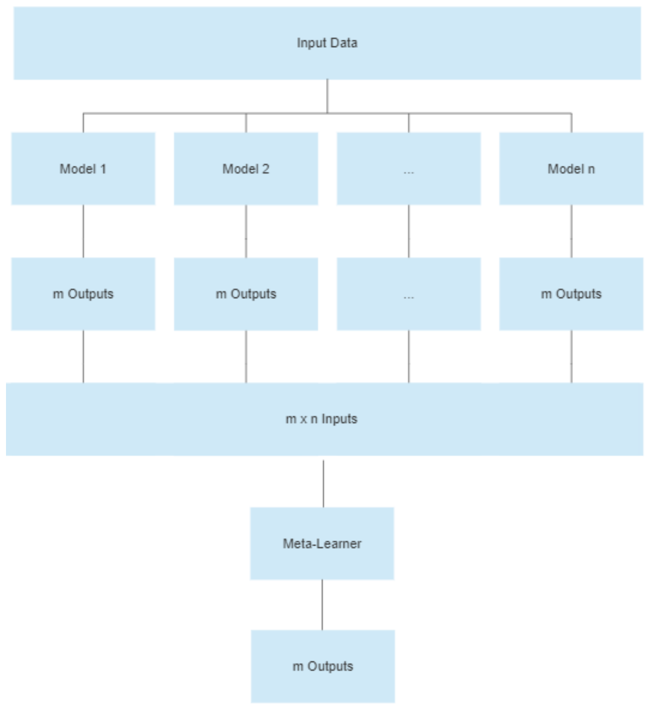
2 General Design

The vision system will have two main parts: a stack of models, and a meta-learner which in our case is an additional neural network. There are a number of ways to implement ensemble learning, however one of the easiest, most efficient, and most accurate is to implement stacking - stacking another neural network on top of our stack of models.

This is the implementation I'm choosing to use. The neural network itself will be a simple MLP. To start, the MLP will have a single hidden layer with 12 neurons. The number of neurons may be subject to change depending on how many algorithms are fed into the MLP - it seems to be consensus that a rule of thumb when choosing the number of neurons in your hidden layer to be about 2/3 of the sum of the neurons in your input and output layers. In my case, I'm assuming there will be 4 algorithms feeding into my neural network (with 4 outputs each), and that my algorithm will output 4 neurons.

In the output layer of the meta-learner, I'll be implementing a softmax layer. This will allow the meta-learner to produce percentages in decimal form as outputs so that the outputs correspond with the percentage likeliness of the image being the thing that the neuron corresponds to.

2.1 General Diagram of Vision System



2.2 Overview of Democratic Algorithm

VisionSystem Class
A class containing the ModelStack and MetaLearner. Takes an input image, plugs it into a ModelStack then plugs the ModelStack's output into the MetaLearner and returns the MetaLearner' output.

ModelStack Class
The stack of models will be implemented by creating a class in Java called, "ModelStack". Initialization of a ModelStack object will initialize a set of pretrained models (such as our CNN and PCA algorithms). A processImage function will return an array containing the outputs of all of the function in the ModelStack.

MetaLearner Class
The democratic algorithm itself will contained in a class called "MetaLearner". When initialized with the filepath of a file containing preexisting weights is passed in, a MetaLearner object will initialize its weights as those passed in weights. When nothing is passed in, a MetaLearner will randomly initialize weights.
The MetaLearner and its subclasses will be designed following this tutorial. Since this tutorial is in Python, I will be using it as a guide to create my MetaLearner, though there may be a few differences in implementation.

A MetaLearner will contain the following variables in its field:

- a private NeuralNetwork object
- a private final int numberOfInputs equal to the number of inputs the MetaLearner should be receiving from the ModelStack

A MetaLearner will contain the following functions:

- save() which saves the weights of the MetaLearner
- backProp() which runs the back propogation algorithm
- feedForward() which takes in an array of doubles (the outputs from the ModelStack)

NeuralNetwork Class
A NeuralNetwork object will contain a fully constructed neural network with input, output, and hidden layers.

A NeuralNetwork will contain the following variables in its field:

- An array of Layers
- A function lossFunction
- A function lossDerivative

A NeuralNetwork will contain the following functions:

- add() which appends a layer passed into the function
- setLoss() which sets the loss function of the NeuralNetwork
- feedForward() which takes in inputData and passes it through the layers and returns the final layer's output
- train() which takes in inputs, expectedOutputs, epochs, and a learningRate and then trains the NeuralNetwork.

Layer Class
A Layer is an abstract class that is used to model a neuronLayer as well as an activationLayer

A Layer will contain the following variables in its field:

- a double[] input
- an double[] output

A Layer will contain the following functions:

- feedForward() which feeds the input through the layer and returns an output
- backProp() which takes in an outputError and a learningRate then returns the inputError

NeuronLayer Class
A NeuronLayer is a Layer representing the neurons of the NeuralNetwork.

A NeuronLayer will contain the following variables in its field:

- An inputSize x outputSize array of weights
- A 1 x outPutsize array of biases
- a double[] input
- a double[] output

A NeuronLayer will contain the following functions:

- feedForward() which feeds the input through the layer and returns an output
- backProp() which takes in a previously calculated outputError and a learningRate then returns the calculated inputError (error of the input)

ActivationLayer
An ActivationLayer takes in an array of inputs and passes it through an activation function.

An ActivationLayer will contain the following variables in its field:

- an array of inputs
- an array of outputs
- a passed in activationFunction
- a passed in activationFunctionDerivative

ActivationFunctions Class
The ActivationFunctions class will contain functions that model certain activation functions (in our case the sigmoid and softmax functions) as well as their derivatives.

The ActivationFunctions class contains the following functions:

- sigmoid() which applies and returns the sigmoid function on an input
- sigmoidDerivative() which applies and returns the derivative of the sigmoid function on the input
- softmax() which applies and returns the softmax function on an input
- softmaxDerivative() which applies and returns the derivative of the softmax function on an input

LossFunction Class
The LossFunction class contains the loss function and the derivative of the loss function used to calculate the error/derivative of the error with respect to the output of the final layer.

The LossFunction class contains the following functions:

- meanSquaredError() which returns the mean squared error given an actual and expected value
- meanSquaredErrorDerivative() which returns the derivative of the mean squared error given an actual and expected value

Matrix Operations
In addition to the above classes, I will be using the ApacheCommons Linear Algebra Library to implement operations such as the dot product or transpose of matrices.

3 Tasks

1. Create the framework for all of the classes used in/by the ModelStack and MetaLearner
2. Implement ModelStack
3. Test ModelStack to ensure it’s working as intended
4. Implement the MetaLearner without softmax layer
5. Test MetaLearner on a small basic dataset to ensure it’s working properly
6. Test MetaLearner on data output by ModelStack (without training) to ensure no errors
7. Train MetaLearner using ModelStack
8. Test Trained MetaLearner
9. Implement a softmax layer in the output layer of the MetaLearner