Landon Buell

1 March 2020

A Breakdown of the Source Code for Scikit- Learn’s Multilayer Perceptron Classifier Object

Dependencies:

sklearn (0.22.1)

scipy

numpy

**Introduction**

A *Multi-Layer Perceptron* (MLP) is one of the most common neural network architectures due to it’s simplicity and general applicability [Geron]. A perceptron is built by connecting layers of *neurons* or nodes that hold floating point values, usually compressed between 0 and +1. The first layer of an MLP network is usually called *layer 0* or the *input layer* and is where features are entered in the form of a numerical vector. A series of matrix multiplications then transforms those features between the *hidden layers* of the network. The final transformation brings the features into the last layer of the network, called the *output layer*. This layer is another numerical vector that is represents the final output or decision of that network.

Python’s open-source library *scikit learn* (sklearn) provides a built-in class object that implements a linear Multilayer Perceptron model. The version that will be explored in this paper is the MLPClassifer object from sklearn 0.22.1. To understand the object, how it works and how to modify it to suit needs, we will produce a short program that implements it in Python and proceed through the source code.

Scikit-Learn Documentation Home page:

<https://scikit-learn.org/stable/>

Multilayer Perceptron Classifier Documentation page:

<https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html>

The documentation – string for the MLPClassifer object indicates that the Multilayer Perceptron classifier operates by optimizing the log – loss function by using *Limited Memory Broyden-Fletcher-Goldfarb-Shanno* (LBFGS) or *Stochastic Gradient Descent* (SGD). For the analysis of this algorithm, we will use the SGD method as set by the parameter *solver.*

**MPL Classifier Implementation**

**MPL Classifier Instance Initialization**

An instance of the MLPClassifier is created with the line:



Here we create the instance, and the class’ \_\_init\_\_() method is called. We pass in the argumnets to indicate 4 hidden layers in the model, with 20 neurons in each layer. We establish a *relu* activation function, which is short of *rectified linear unit*. As previously indicated we have also chosen to use the stochastic gradient descent model as a solver. The parameter *max\_iter* sets the maximum iterations to test before the gradient descent solver will move to the next training sample, even it it does not converge to a minimum. The parameter *tol* indicates the tolerance for that gradient descent uses to establish the minimum. Once this value is reached, the parameters for a specific sample converges and the algorithm moves to the next sample. Finally, we set *random\_state* for reproducible results. In a formal implementation this parameter would not be set.

**MPL Classifier Instance Fitting Method**

The MLP Classifier is trained with the MLPClassifier.fit() method, which takes the matrix *X* of training samples and features and the vector *y* of training labels. This in turn calls the parent class BaseMultilayerPerceptron and the method BaseMutlilayerPerceptron.\_fit(). This method then checks the parameters from the class instance and validates them. Errors are raised where needed. Since we have set the solver to use a stochastic gradient method, the fitting function for the SGD solver is called in BaseMultilayerPerceptron.\_fit\_stochastic(). Given the specific use of SGD, as opposed to another stochastic solver (“adam”), the method, another child class is created called SGDOptimizer().

**Activations (line 357)**

The term *activation* refers to the value contained within each neuron of a given layer. For the linear, matrix-vector description of the MLP object, an activation is simply the numerical value stored within each entry of an array. Thus, the array labeled “activations” is the state of each neuron in a particular layer. The dimension of the activation array is always some amount of rows, by 1 column. For the input layer of a network, the activations are the raw features, given as is – a single feature to a single neuron. The output layer for a k-classes classifier has k neurons in it.

**Stochastic Fitting (line 474)**

The stochastic method then takes the full data set and target vector as given by the X and y arrays and permutes them according to the random state parameter. An internal function splits the full dataset into *mini batches*, which are then iterated through (line 517). This is where the processes of forward feeding and then back propagation begins. Under each loop, a batch is used to index a subset of the X and y arrays, and is given to a back propagation algorithm: BaseMultilayerPerceptron.\_backprop().

**Back Propagration (line 181)**

The documentation string for this method indicates that it computed the loss function for the multilayer perceptron. Additionally, the local derivative of each of each parameter is computed. This is the numerical equivalent to the gradient operation, which gives the SGD algorithm it’s name. Back propagation takes the X and y subset arrays and the additional arguments:

A list of activations - one element corresponds the activation of that layer (length of L-1)

A list of coefficient Gradients – Each element is the amount of change to update a coefficient parameter for an iteration

A list of intercept Gradients – Each element is the amount of change to update a intercept parameter for an iteration

This function is called underneath a “for” loop, one iteration for each “mini batch” of training samples. A mini batch is a subset of rows from the larger feature matrix. Each subset is then fed into the back propagation method. For that subset of samples, the value of the loss function, the coefficient gradients, and intercepts gradients are returned. The loss function for that mini batch subset is added to larger total. The coefficient and intercept gradients are also used to update the gradient vector. These parameter are used to update the parameters in the “optimizer” child class instance, which in out case is the SGDOptimizer() class (“\_stochastic\_optimizers.py”, line 74)

For back propagation to update these parameters, the network must first implement a *forward pass* of the data subset. This is the equivalent of taking each training sample and feeding it through the MLP network. The result of a forward pass is a prediction by the classifier given an array of features.

**Forward Pass (line 91)**

This method performs a forward pass through the network by computing values of neurons in each layer up to and including the output layer. The only argument required it to take a list of activations, where the i-th element is the activation for the i-th layer. To forward pass, the function loops over the number of layers, minus 1. For each iteration, the i-th activation and i-th coefficients and the i-th intercepts are all used to compute the coefficients and intercepts for the i+1-th layer.



In line 102, a ‘for’ loop is initiated to iterate through each layer of the network. The i+1 layer is computed by a method called “safe\_sparse\_dot”. This function is defined in the script “extmath.py”, line 118. It computes the dot product between the i-th layer activations, and the i-th layer coefficients.



This method has a great deal of built-in error handling. If the entries *a* and *b* are always guaranteed to be less than 2D, then most of the error handling in the function is obsolete. This function could be replaced by another method that similarly compute that matrix of dot product between the arrays and then returns the result. This method would be and excellent candidate to replace with some approximate computation method.

The intercepts for the i-th row are then also added into the list of the the layer i+1 activations. This each time the loop it iterated through, a new row, for each layer is added into a list of lists. Each list contains the activations and intercepts for that particular layer. The final output of the function is a list containing N\_layers – 1 rows, each row has the activations and incercepts for a layer.

**Back Propagation Continued**

Now that the activation lists have been computed, the back propagation algorithm resumes. The last row of the activations list of the final output of the network. This is essentially it’s decision as to what class a sample or subset of samples belongs to. We can compare this last layer to the target output. The algorithm then feeds both the predicted output and the target output into a defined loss function, which can also be manipulated by the user. The result of this is a variable “loss” which is the quantity that we seek to minimize is stochastic gradient descent. The loss function is also modified with the L2 penalty (also called *alpha parameter*). (lines 228 – 231)

Now, the method can move on the computing the loss function’s gradient. It uses the index of the last row, the number of samples given, the activations list, deltas and the coefficient and intercept gradients. The final lines of this function compute the gradient of the cost function given all of the layers of the network:



**Computing the Loss Function Gradient**

The gradient for the final layer of the network is computed first.