Landon Buell

1 March 2020

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

**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().

**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

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

**Sources**

Géron Aurélien. *Hands-on Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems.* O'Reilly, 2017.

Goodfellow, Ian, et al. *Deep Learning*. MIT Press, 2017.