Multilayer Perceptron Regressor in Python 3

Landon Buell

25 April 2020

1 Introduction

The Multilayer Perceptron is an simple architecture type of neural networks developed by Frank Rosenblatt in the 1950's [1, 5]. It is based off of layers of neurons or nodes , that are connected through synapses and often given an additional intercept as well. Mathematically and computationally, this is modeled though use of column vectors to represents layers, with each entry as the *activation* of a particular neuron [2, 6].

The connections are modeled by elements matrices in matrices, and the intercepts are corresponding indicies inside a *bias vector*. This means that for a fully connected network, every neuron activation in a given layer is the linear combination of all activations in the previous layer, plus an additional intercept. Additionally, an *activation function* is included in this model in order to introduce a degree of non-linearity depending on the nature of the problem [2]. We can mathematically express the formula for a single activation in a layer $(l+1)$ as produced from a layer (l) is:

$$x_i^{(l+1)} = f\Big(\sum_j w_{ij}^{(l)} x_j^{(l)} + b_i^{(l)} \Big) \tag{1}$$

We can express this more compactly, with a consolidated matrix-vector equation. Note that upper indicies indicate the layer number. Layer 0 is the entry layer, where initial vectors of features are given to the network. In a model with L layers, there layer $L-1$ represents the networks final output. We can denote this:

$$\vec{x}^{(l+1)} = f\Big(\hat{W}^{(l)} \vec{x}^{(l)} + \vec{b}^{(l)} \Big) \tag{2}$$

Where $\hat{W}^{(l)}$ is the $(m \times n)$ weighting matrix of the l-th, $\vec{x}^{(l)}$ is the $(n \times 1)$ vector containing the activations for the l-th layer. $\vec{b}^{(l)}$ is the $(m \times 1)$ bias vector which acts as in intercept for the decision boundary, and $\vec{x}^{(l+1)}$

2 Regressor Model

For this project, I have constructed a Multilayer Perceptron Regressor network model, composed of 1 hidden layer. In a k-bins classifier problem, the output of the network, layer $L-1$ would k neurons here, one for each class, and in a regression setting, there is a single neuron, The activation of which is the networks decision [1]. To simplify the complexity of the perceptron model, I have adapted a design build from James Loy's work *Neural Network Projects with Python* [4] that uses a user-built class object as a basis. Note that Loy's model does not support bias vectors in the feed forward equation (2) and uses the *logistic sigmoid* activation function.

I have generalized a class object *"Multilayer_Perceptron_Regressor"* to be able to accept a data set in any (reasonable) dimensional feature space, pass it through a network consisting of a single hidden layer with any number of neurons in that layer. The perceptron then returns an output based on the data which can be compared to the target value for each class. To augment Loy's model, I have included a *momentum* parameter that can scale the gradient in the SGD optimizer algorithm, as well as a method within the object instance to track the value of a Residual Sum of Squares Loss Function with each iteration.

For this project, have chosen to iterate repeatedly over a simple data set also supplied by Loy.

Feature 0	**Feature 1**	**Feature 2**	**Target**
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	0

Admittedly, this creates the trouble of over-fitting on the same set for 1000 epochs [3], but this serves as a proof-of-concept as well. For each model, I have chosen to use 10 neurons in the single hidden layer.

3 Building & Training

The model was built with a simple feed-forward architecture in mind. I hard-coded a system of 1 hidden layer, 2 matrices and no biases vectors. The standard feed forward equation (2) was used twice to produce an output for the training set, and implemented as a method within the MLP object. To update the model parameters, a Gradient Descent algorithm was hard-coded in for simplicity as it's own method. The idea is to compute the gradient of the Loss function (RSS) with respect to each parameter in the weighting matrices. To better do this, we can break the feed-forward equation into two separate pieces:

$$\vec{x}^{(l+1)} = f\left(\vec{z}^{(l+1)}\right) \qquad \vec{z}^{(l+1)} = \hat{W}^{(l)}\vec{x}^{(l)} + \vec{b}^{(l)} \tag{3}$$

Each element of the gradient is then the partial derivative of the loss function with respect to each element in the weight matrices. We can first differentiate the Loss function itself (Recall, RSS) with respect to the output layer $\vec{x}^{(2)}$.

$$\nabla_{x^{(2)}} L = 2(\vec{y} - \vec{x}^{(2)}) \tag{4}$$

Where L is the RSS loss function, $\vec{y}$ is a column vector containing the target outputs for each sample and $\vec{x}^{(2)}$ is the output as predicted by the network [4, 3]. Similarly, we can compute the gradient with respect to each element in the $\hat{W}^{(1)}$ matrix through use of the chain rule from multi-dimensional calculus [2]:

$$\nabla_{W^{(1)}} L = \frac{\partial L}{\partial x^{(2)}} \frac{\partial f}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial W^{(1)}} = 2(\vec{y} - \vec{x}^{(2)}) \odot f'(z^{(2)}) * \vec{x}^{(1)}.T \tag{5}$$

Note that $\frac{\partial L}{\partial x^{(2)}}$ can be treated as an element-wise version of $\nabla_{x^{(2)}} L$ in equation (4). Recall that for F, we have chosen to use the logistic sigmoid function, and ∂f indicates that first derivative of that function as well. Note than in all cases, standard matrix products ($*$) and Hadamard products ($\odot$) must be used appropriately to ensure accuracy and preserve dimensions.

For the changes with respect to the hidden layer, we can compute:

$$\nabla_{x^{(1)}} L = \frac{\partial L}{\partial x^{(2)}} \frac{\partial f}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial x^{(1)}} = 2(\vec{y} - \vec{x}^{(2)}) \odot f'(z^{(2)}) * \hat{W}^{(1)}.T \tag{6}$$

Finally, we can compute the gradient with respect to each element in the $\hat{W}^{(0)}$ matrix.

$$\nabla_{W^{(0)}} L = \frac{\partial L}{\partial x^{(1)}} \frac{\partial f}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial W^{(0)}} = \frac{\partial L}{\partial x^{(1)}} \odot f'(z^{(1)}) * \vec{x}^{(0)}.T \tag{7}$$

Where $\frac{\partial L}{\partial x^{(1)}}$ is given in equation (6).

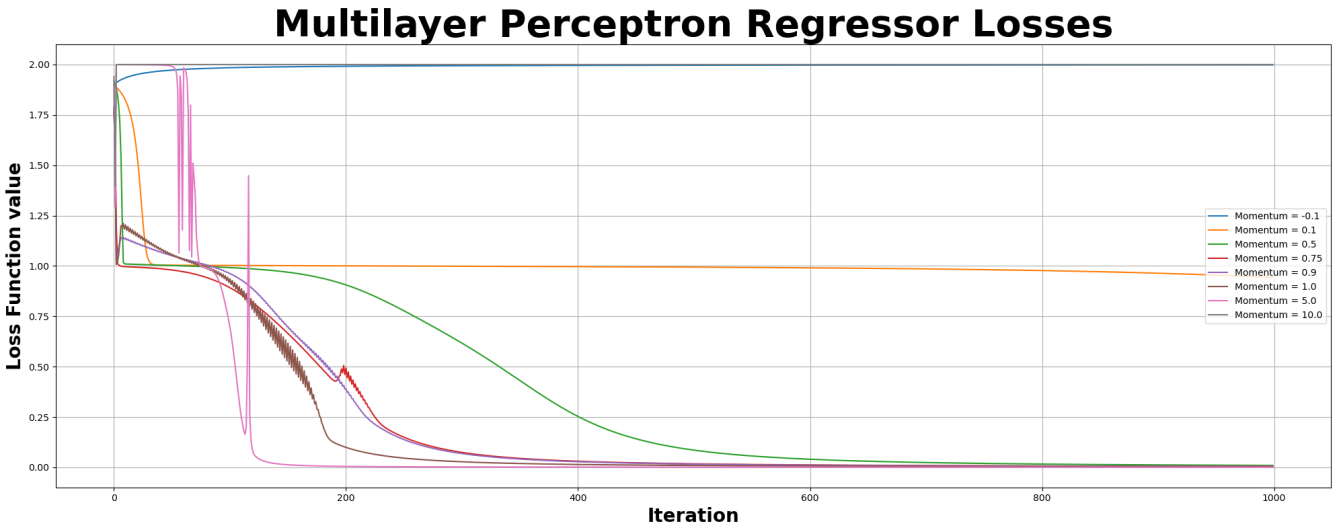
Each of these equations has been implemented computationally in a back propagation method. Once each gradient for the weights is computed, (7) and (5), the arrays are multiplied by the momentum scalar, and then added to the existing weights to update them. This process repeates for a for a number given by a "max_iters" class attribute.

4 Adjusting the Momentum Parameter

Since my earliest attempts at creating this model from scratch were unsuccessful, My model is mostly constructed on top of Loy's. To add more depth to this project, I experiment with adjusting a static momentum parameter between eight different models, in each model, the entries in the gradient vector are scaled by that constant. I have included the final output for each of the four samples below, as well as the value for the final loss function iteration.

Model #	0	1	2	3	4	5	6	7
Momentum	-0.1	+0.1	+0.5	0.75	+0.9	+1.0	+5.0	+10.0
Sample 1	0.9990	0.2969	0.0271	0.0203	0.0149	0.0210	0.0081	0.9999
Sample 2	0.9997	0.6421	0.9518	0.9654	0.9674	0.9748	0.9911	0.9999
Sample 3	0.9997	0.5296	0.9525	0.9636	0.9668	0.9743	0.9908	0.9999
Sample 4	0.9998	0.5553	0.0582	0.0431	0.0397	0.0284	0.0101	0.9999
Final Loss	1.9981	0.7518	0.0097	0.0039	0.0037	0.0031	0.0003	1.9999

For each model, the current RSS loss function value is stored in an array. To compare the differences in how the loss function behaves (ideally approaches 0 as the number of epochs approaches $+\infty$), I have provided visualizations below comparing the models and their loss function after each iteration.



5 Conclusion

From this experiment with a simple Multilayer Perceptron Regression Model, we can observe the following from tracking the loss function of each model with respect to each passing training epoch.

- A model with a 'negative' momentum will diverge from a minimum and seek a maximum loss value - i.e. the gradient is pointing 'uphill'. Additionally, a sufficiently large enough momentum will cause an overshoot and diverge to a maximum as well.
- A model with sufficiently low momentum will approach a minimum but may not reach a suitable minimum, even after many iterations
- A model with a 'medium' momentum (around +1) will converge sufficiently quickly
- A model with a greater momentum may converge faster than those with a smaller momentum, but often, the loss value experience more volatile changes incating an overshoot of a possible loss local minimum.
- In general, models show behavior that the loss function asymptotically approaches zero, which is desirable to to discourage over-fitting.

References

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

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

[3] James, Gareth, et al. *An Introduction to Statistical Learning with Applications in R*. Springer, 2017.

[4] Loy, James. *Neural Network projects with Python*. Packt Publishing, 2019

[5] McCulloch, Warren S., and Walter Pitts. "A Logical Calculus of the Ideas Immanent in Nervous Activity." *The Bulletin of Mathematical Biophysics*, vol. 5, no. 4, 1943, pp. 115–133.

[6] Petrik, Marek. "Deep Neural Networks." Machine Learning. 27 April. 2020, Durham, New Hampshire.