## Machine Learning Final Project

Submitted by: Cicy Fan, [cf1311@nyu.edu](mailto:cf1311@nyu.edu), Indu Ramesh, [ir914@nyu.edu](mailto:ir914@nyu.edu)

### Extension 1: LASSO Regularization + Logistic Regression

**Motivation and Theory**

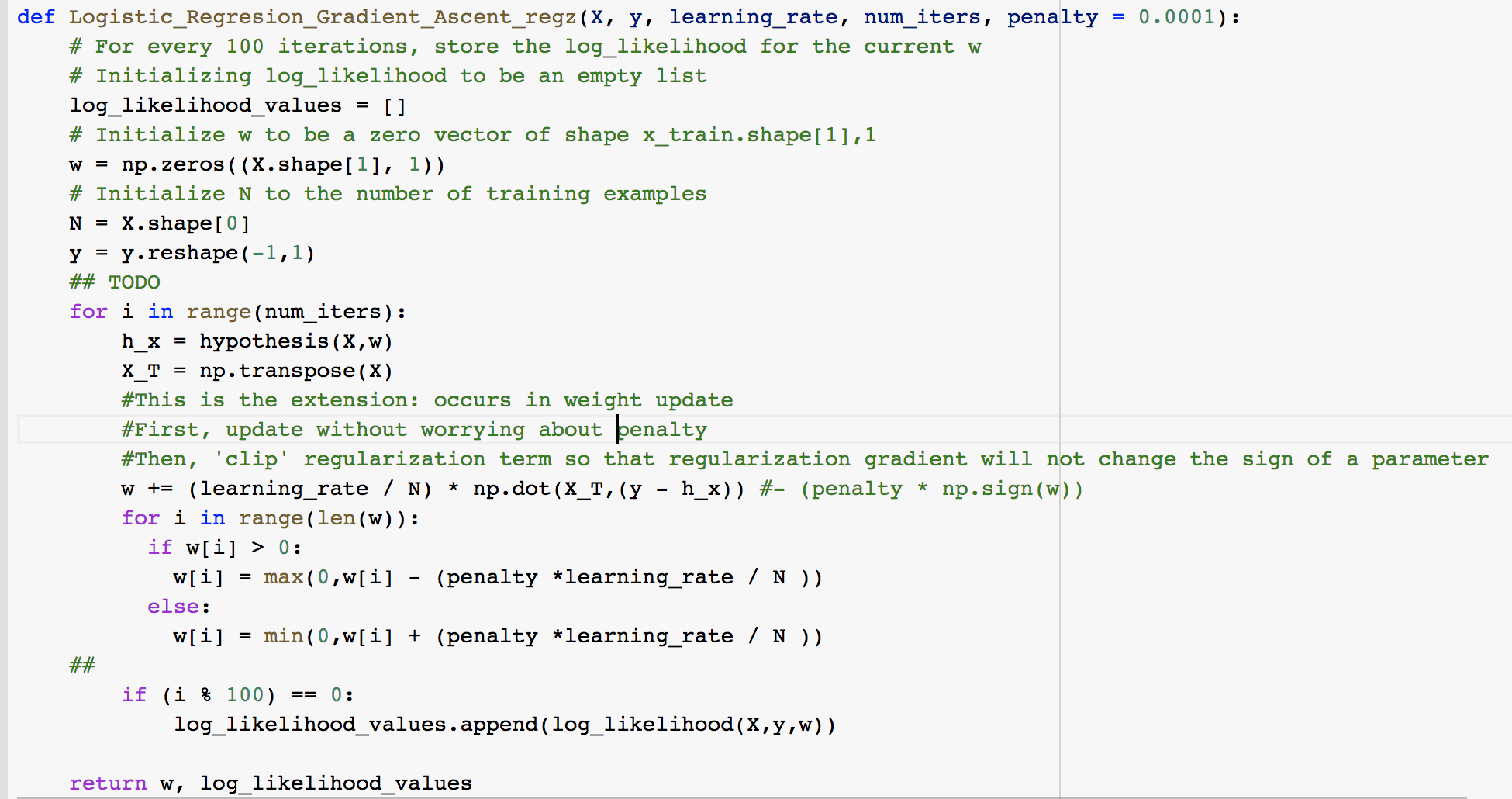
Logistic regression utilizes linear regression to provide a linear boundary for binary classification problems, and then we classify according to the maximum likelihood probability. The cost function of logistic regression is:.

To train a model with logistic regression, we maximize this likelihood function, via gradient ascent. However, maximizing this function can cause overfitting on our training set, especially if the size of the training set is small. A heuristic for observing this kind of overfitting is large weights.

Regularization reels in overfitting by penalizing our weights from growing. There are 2 types: LASSO and Ridge Regression. LASSO regression penalizes the weights via adding a penalty term from the loss function above (since we implement gradient ascent, we subtracting the penalty term rather than adding):

, where is a hyperparameter tuning the penalty.

In LASSO, we will observe features becoming more and more sparse-- some features will converge at 0. Thus LASSO implicitly does feature reduction.

**Our Numpy Implementation** 

The absolute value function doesn’t have a derivative, so there is no closed-form solution for LASSO. Several algorithms exist for adding LASSO regression, including coordinate descent. We implemented a different approach, the “lazy update” approach given in this [paper](https://lingpipe.files.wordpress.com/2008/04/lazysgdregression.pdf) by Carpenter.

Basically, the regularization term is “clipped so that subtracting the regularization gradient will never change the sign of a parameter” (Carpenter 08). We used the pseudocode given in the paper in our Numpy implementation.

**Datasets**

We used the breast cancer dataset from sklearn (from the homework), which has 569 examples with 30 features, and 2 classes. We also used a modified version of the wine dataset from sklearn (from outside of class).

The original dataset has 3 classes, corresponding to 0, 1, and 2. To adapt the dataset for binary classifications, I only kept the data corresponding to classes 1 and 2 and removed the data corresponding to class 0. My modified dataset had 119 examples and 13 features.

**Results**

|  |  |  |  |
| --- | --- | --- | --- |
| **Implementation** | **Dataset** | **Best Result (rounded), w/ hyperparameters (if applicable)** | **Weights for Numpy Logistic Regression + LASSO on Breast Cancer:** (note top weight = bias)  weights = [[ 1.00377823]  [-0.49874581]  [-0.47863449]  [-0.58771095]  [-0.51968986]  [-0.4273466 ]  [-1.76078111]  [-0.53940899]  [-0.87587144]  [-0.57060674]  [ 0.74607842]  [-1.56283208]  [ 0.93023629]  [-0.48512585]  [-1.42674032]  [ 1.24220447]  [-0.45119287]  [ 0.87608049]  [ 1.20180262]  [ 0.25031667]  [ 0.44468964]  [-0.80063973]  [ 0. ]  [-0.5966012 ]  [-0.60910056]  [-0.63456718]  [-0.97096186]  [ 0. ]  [ 0. ]  [-0.00476911]  [-0.97769079]] |
| SciKitLearn Logistic Regression | Breast Cancer | 96.1% |
| SciKitLearn’s Logistic Regression + LASSO Regularization | Breast Cancer | 97.4% |
| Our Numpy Logistic Regression | Breast Cancer | Accuracy: 94.7 %  Optimal learning\_rate = 0.01  Optimal threshold = 0.5  Optimal # of iterations = 632 |
| Our Numpy Logistic Regression + LASSO Regularization | Breast Cancer | Accuracy = 94.9%  Penalty () = 0.1  Optimal learning\_rate = 0.01  Optimal threshold = 0.3  Optimal # of iterations = 31  See right for sparsity results: features 22, 27, 28 went to 0. |
| SciKitLearn Logistic Regression | Wine (Modified slightly, see above) | 93.8% |
| SciKitLearn’s Logistic Regression + LASSO Regularization | Wine | 95.8% |
| Our Numpy Logistic Regression | Wine | Accuracy: 93.75%  Many combos of hyperparameters tested gave this accuracy. Here is one:  threshold = 0.5 learning rate = 0.5: | **Weights for Numpy Logistic Regression + LASSO on Wine:**  weights = [[ 0.09415723]  [ 0.23776728]  [ 0.29354336]  [ 0. ]  [ 0. ]  [ 0. ]  [-0.95845042]  [ 0. ]  [ 0. ]  [ 1.77496089]  [-0.62646242]  [-0.39640343]  [ 0.16180071]] |
| Our Numpy Logistic Regression + LASSO Regularization | Wine | Accuracy: 97.9%  Many combos of hyperparameters tested gave this accuracy. This one matches the above, + penalty: threshold = 0.5  learning rate = 0.5,  Penalty () = 2  See right for sparsity results: features 4,5,6,8,& 9 went to 0. |

**Discussion**

Adding LASSO regularization produced an improvement in both sklearn’s implementation on both datasets and our implementation for the wine dataset.

For the breast cancer data set, we saw a very small (~0.2% improvement) with LASSO. The reasoning might lie behind the observation that not many features converged at 0 (just 3/30 were “removed”); more features were “necessary” to diagnose breast cancer, thus a lower penalty was needed to penalize the weights and the effect of the improvement was small. However, with the wine dataset there was a much larger improvement (~4%), and a larger portion of features were removed (5/13). Both of these results agree with the theory of LASSO regularization, mentioned above -- the weights are penalized slightly to prevent overfitting to the training set, and thus the test set accuracy improves (higher bias but less variance).

Our results were similar to scikitlearn’s, except that the final accuracy of our Numpy Logistic Regression + LASSO on the wine was actually better than scikitlearn’s. Why might this be? First off, scikitlearn’s Logistic Regression does not support batch gradient descent. The closest I could get to our homework interpretation that supported LASSO was ‘saga,’ a “fast incremental gradient method’ that adapts and optimizes on stochastic gradient descent with an ‘averaging strategy’, outlined in this [paper](https://arxiv.org/abs/1407.0202). Our implementation used batch gradient descent, and coupled with the LASSO algorithm above, produced good results on both datasets (especially with some hyperparameter tuning). Additionally, scikitlearn’s version did not converge, even after 5000 iterations, even though I scaled the data via preprocessing. While the overall accuracy was still good for scikitlearn, our implementation converged when scikitlearn’s didn’t.

### 

### 

### 

### 

### 

### 

### 

### 

### 

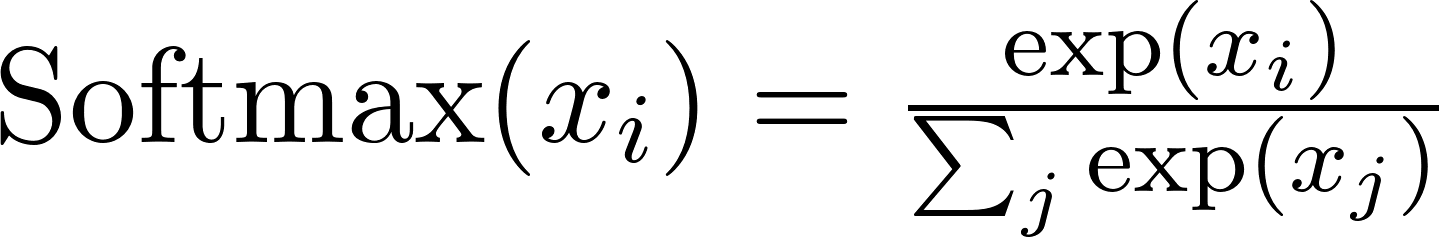
### 

### Extension 2 - Neural Network with Softmax activation

**Motivation and Theory**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  |  | | --- | --- | | Output Layer Activation | Accuracy | | softmax | 97.56% | | sigmoid | 98.00% | | hard\_sigmoid | 52.67% | | tanh | 10.22% | | softsign | 11.56% | | softplus | 96.89% | | relu | 50.00% | | selu | 16.44% | | elu | 42.22% | |

In our neural network homework, we implemented a simple neural network with activation functions such as sigmoid, tanh, and relu, and we used the same function in input and output layers. Since the digits dataset is a multiclass classification problem, my first thought for an extension was to apply the softmax activation function to the output layer. However, I also wanted to see what other activation functions worked well. Therefore, I iterated through 9 different activation functions available with the [Keras](https://keras.io/activations/) library for the output layer, while holding the input layer activation function fixed as sigmoid. For a neural net with 100 epochs, the sigmoid output layer activation function performed best at 98%, with softmax at 97.56% and softplus at 96.89% with the digits dataset. As a result, I chose softmax as my extension. As defined [here](https://towardsdatascience.com/epoch-vs-iterations-vs-batch-size-4dfb9c7ce9c9), an epoch is when an entire dataset is passed forward and backward through the network once, which is similar to the idea of iterations used in our neural network homework, given that the batch size is the size of the dataset. Otherwise, iterations is the number of batches needed to complete one epoch.

Mathematically, softmax is written as: [](https://www.codecogs.com/eqnedit.php?latex=%5Ctext%7BSoftmax%7D(x_%7Bi%7D)%20%3D%20%5Cfrac%7B%5Cexp(x_i)%7D%7B%5Csum_j%20%5Cexp(x_j)%7D#0). This is an extension of logistic regression that normalizes the result of the sigmoid function, where each class is a one-hot-encoded vector. The output of softmax is a probability distribution; since the outputs have been normalized, they all add up to 1. Recall that in sigmoid; the outcome is between 0 and 1 (but may not necessarily sum to 1), but in softmax we get a probability distribution (that sums to 1).

**Results**

With the [second dataset](https://www.kaggle.com/rtatman/did-it-rain-in-seattle-19482017), I chose a binary classification problem and the softmax function still performed well, but not as well as the digits dataset. I performed the same experiment as above, but with 50 epochs to reduce runtime. The result was still impressive: sigmoid 97.6%, softmax 97.4%, and softplus 97.2%.

Our first dataset, the digits, is multi-class classification, where each example has only one correct answer that corresponds to one of 10 digits. Softmax will increase the probability of 1 class and decrease the others. In this scenario for the digits dataset, this is exactly what we want, which may explain the reason behind the improvement we saw with Softmax vs. Sigmoid.

On the second dataset, Rain, we saw a smaller improvement. Note that this is a binary classification problem, and the only difference between sigmoid and Softmax here is that Softmax will output a probability distribution, and sigmoid will simply output the value between 0 and 1. This is still a dataset where each example has a correct answer out of two (True or False), but since there are not as many classes, the probability distribution output does not have as significant an effect on the result.

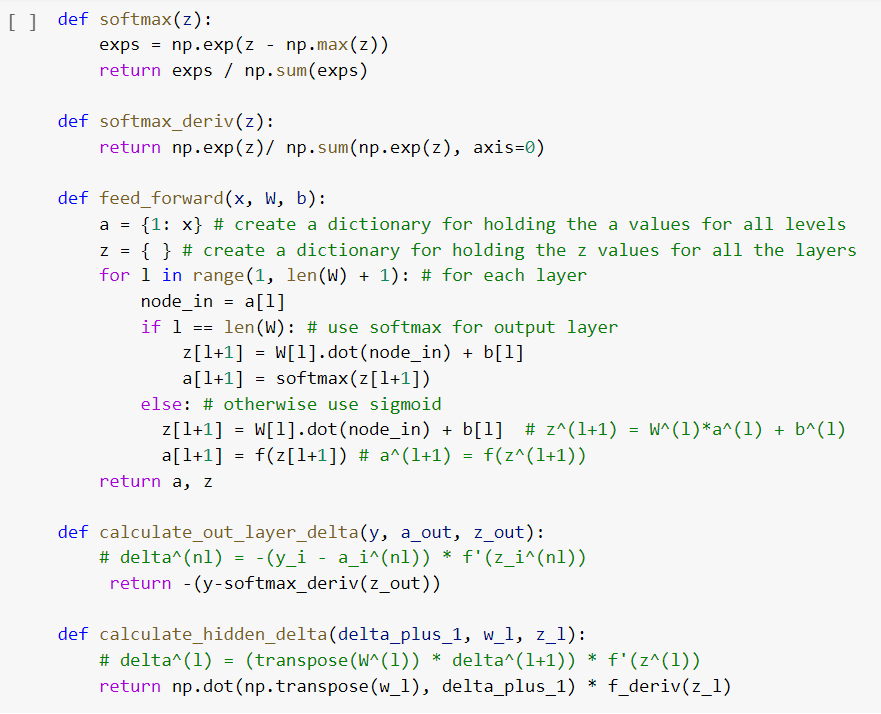
Interestingly, the numpy implementation of the neural networks actually performed better than the ones implemented with Keras in some cases, especially when the softmax function is added. I suspect this is due to parameter selection of values such as alpha or lambda. I did not find a straightforward method to specify parameters of the activation functions with Keras, and this issue was well documented across Github and Stackoverflow. Another reason could be that the Numpy implementation initialized weights uniformly drawn from [0, 1), while the Keras library might be using another distribution.

In conclusion, softmax is a powerful output layer activation function for not only multiclass classification problems, but can also help to improve the accuracy of binary classification problems. It is easy to implement and achieves strong results. Softplus also achieved pretty good results in our experiment, and might be another activation function worth looking into.

|  |  |  |
| --- | --- | --- |
| Digits | Naive - sigmoid | Extended - Softmax |
| Result Graph (Keras) |  |  |
| Accuracy (Keras) | 93.3% | 95.8% |
| Accuracy (Numpy) | 92.9% | **97.5%** |

|  |  |  |
| --- | --- | --- |
| Rain | Naive - sigmoid | Extended - Softmax |
| Result Graph (Keras) |  |  |
| Accuracy (Keras) | 90.6% | **91.8%** |
| Accuracy (Numpy) | 86.2% | 90.8% |

**Our Numpy Implementation: Code Screenshot**



### 

### 

### 

### 

### Extension 3: Momentum + Neural Networks

**Motivation and Theory**

Neural Networks allow for much more complicated decision boundaries to be created to separate data, allowing us to solve more difficult classification problems with more accuracy. The trade-off is that the cost function we need to minimize becomes complicated, and we can no longer assume that it’s convex. There may be several local minima we get stuck in when descending the gradient of the function.

Momentum is a method to avoid getting stuck in a local minimum by gathering “momentum” as we descend. Thus if we are stuck in a local minimum, we have “momentum” to carry us out. The “velocity” from the previous gradient helps us build momentum. Mathematically, momentum is a hyperparameter between 0 and 1 that we have to tune to avoid overshooting.

Here is a definition of momentum, from a [paper](http://proceedings.mlr.press/v28/sutskever13.pdf) by Sutskever et al. Essentially, the weight update is given a velocity vector to keep track of. Given a function f(to minimize, momentum is defined as:

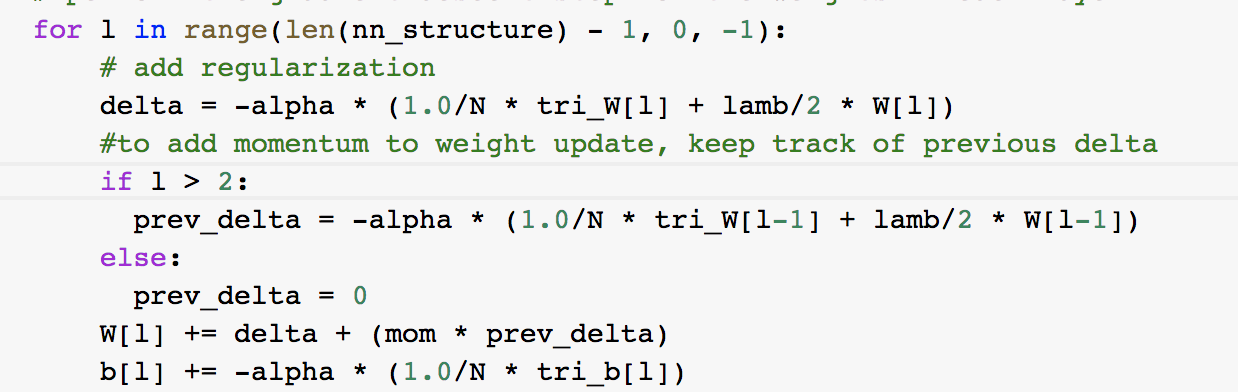
where represents the learning rate, andis a momentum hyperparameter between 0 and 1 (Sutskeveer et al 13).

In theory, momentum allows for faster convergence and a more optimal final result. In practice, according to the [scikitlearn documentation for neural networks](https://scikit-learn.org/stable/modules/neural_networks_supervised.html), SGD with momentum can perform better than Adam or L-BFGS optimizations, converging faster and providing more optimal solutions.

We experimented with momentum values and momentum = 0.9 provided the best improvement, which is what we have implemented below.

**Our Numpy Implementation**

The implementation of the above approach is relatively simple. In the weight update, normally, a *delta* term is subtracted from the weights, that is equal to the learning rate \* the derivative of the cost function we intend to minimize. To implement momentum, we add the *delta* from the *previous* weight update to this term, multiplied by a momentum hyperparameter. Thus momentum is “gained” when we descend, expanding from the previous iteration. Our code is below, from the weight update in our backpropagation function (from our original homework on neural nets):



**Datasets**

We used the MNIST toy dataset from sklearn (Digits), which has 569 examples with 30 features, and 10 classes. We also used the iris dataset, which has 150 examples with 4 features, and 3 classes.

**Results**

|  |  |  |
| --- | --- | --- |
| **Implementation** | **Dataset** | **Best Result (rounded), w/ hyperparameters (if applicable)** |
| SciKitLearn neural network | MNIST (scikitlearn) | 97.6% |
| SciKitLearn’s neural network + momentum (0.9) | MNIST | 98.2% |
| Numpy Neural Network | MNIST | alpha: 0.25 lambda: 0.01  gradient descent for 1500 iterations  CPU times: user 2min 33s, sys: 36.9 ms, total: 2min 34s  Wall time: 2min 34s  Accuracy: 84.4% |
| Numpy Neural Network + Momentum (0.9) | MNIST | alpha: 0.25 lambda: 0.01  gradient descent for 1500 iterations  CPU times: user 2min 34s, sys: 41.9 ms, total: 2min 34s  Wall time: 2min 34s  Accuracy: 85.6% |
| SciKitLearn neural network | Iris | 92.1% |
| SciKitLearn’s neural network + momentum (0.9) | Iris | 97.3% |
| Numpy Neural Network | Iris | alpha: 0.25 lambda: 0.01  gradient descent for 1500 iterations  CPU times: user 12.3 s, sys: 26 ms, total: 12.3 s  Wall time: 12.3 s  Accuracy: 81.6% |
| Numpy Neural Network + Momentum (0.9) | Iris | alpha: 0.25 lambda: 0.01  gradient descent for 1500 iterations  CPU times: user 12.3 s, sys: 22 ms, total: 12.3 s  Wall time: 12.3 s  Accuracy: 86.8% |

**Discussion**

Momentum provided an accuracy improvement in both datasets we tested, both in scikitlearn and numpy (~1% for Digits (MNIST) in our implementation, and ~5% for Iris).. This result agrees with the theory above-- the addition allowed the weights to converge faster in an optimum gradient (global minimum). Both datasets corresponded to nonbinary classification with more complicated decision boundaries, and momentum theoretically improves convergence and optimizes weights in such scenarios.

Additionally, our results have lower accuracy than scikitlearn’s--slightly lower for Digits and lower for Iris. This may be because in scikitlearn, iterations to convergence are measured in epochs, and the specific number of iterations cannot be controlled (see documentation).

Finally, below are the gradient descent convergence graphs for Iris for our Numpy implementation (left is without momentum, right is with momentum). Notice how the addition of momentum allows the weights to converge in fewer iterations, as well as the increased accuracy for both datasets reported above. This makes momentum an effective implementation-- relatively simple code, but much more efficient and with noticeable improvements in accuracy.

