Austin Eversole

Neural Networks - Experimentation

We are going to use the MLP Classifier: Multi-Layer Perceptron Classifier, from the SKLearn classifier. It's a very nice Neural Network!

Parameter Experimentation:

I tried Experimenting with the Learning Rate first. I found that with the constant learning rate, the car dataset got an accuracy of 91. Changing the Learning Rate to 'invscaling' keeps the accuracy still the same. SKLearn's documentation about 'invscaling' says: "*invscaling' gradually decreases the learning rate learning_rate_ at each time step 't' using an inverse scaling exponent of 'power_t'. effective_learning_rate = learning_rate_init / pow(t, power_t)"*. Changing the Learning rate to 'adaptive' also doesn't seem to change the accuracy at all. The documentation says the following about adaptive Learning rates: *"'adaptive' keeps the learning rate constant to 'learning_rate_init' as long as training loss keeps decreasing. Each time two consecutive epochs fail to decrease training loss by at least tol, or fail to increase validation score by at least tol if 'early_stopping' is on, the current learning rate is divided by 5.*"Intersting!

Next I tried experimenting with 'hidden_layer_sizes'. This one was more interesting! The parameter that changes the "number of neurons in the ith hidden layer" is a tuple. When the tuple is (5,2), it is an accuracy of 91. With a tuple of (5,10), the accuracy goes up to 97%! With (1,2), the accuracy goes down to 72%. And with a tuple of (50,50) the accuracy is 98%. I noticed that the larger that either number is, the more accurate the neural net will be.

Lastly I experimented with the Activation function. What we used by default is *'relu', the rectified linear unit function, returns f(x) = max(0, x)*. We will try all the other activation functions:

- 'relu', the rectified linear unit function, returns f(x) = max(0, x)
  - Accuracy: 91%
- 'identity', no-op activation, useful to implement linear bottleneck, returns f(x) = x
  - Accuracy: 79%
- 'logistic', the logistic sigmoid function, returns f(x) = 1 / (1 + exp(-x)).
  - Accuracy: 89%
- tanh', the hyperbolic tan function, returns f(x) = tanh(x).
  - Accuracy: 86%

Datasets for Experimentation

I found 4 data sets that will be good to experiment on. The Overwatch Data set and Cars dataset from previous projects, and 2 new ones: The US Census, and Analytics from the Google Play Store.

After a lot of preprocessing and trying to get the datasets to work, I wasn't able to get the Google Play Store dataset to work. The US Census dataset goes through the Neural Network okay, but something went wrong and the Prediction is always 0%. I decided to focus my efforts on experimentation on the

Cars data set instead, and played around with getting that to work. The Overwatch dataset worked too, although which a much lower accuracy (Cars data set had 75% accurate predictions, while Overwatch dataset had around 57% accuracy.

Using the Overwatch Data Set

Back when we were using Decision Trees, I used a dataset that I acquired from Kaggle that contains data from matches in the Video Game "Overwatch". I decided to use the Overwatch Data set for the deep experimentation because it is the most reliable and also the least accurate of the choices that I had. I found that when using the Decision Tree classifier, the data set predictions had an accuracy of 47%, while it had about 60% when using the Neural Network. I changed the dataset a little bit: the Sklearn Decision Tree wasn't able to take n-ary categorical datasets, and so I did a lot of preprocessing to make the Overwatch dataset into something binary that I could still make valuable predictions with. For the Sklearn Neural Network, I made one of the variables (Team Stack) n-ary again. So I may be able to make better predictions this way. Now we will start with the experimentations

With a tuple of (5,2) for Hidden Layer Size, activation function "relu" and a "constant" learning rate, we got 60% accuracy.

After experimenting with higher and lower Layer #'s on both sides of the tuple, (5,2) is the best solution I could come with. Next I decided to experiment with the activation function.

- 'relu', the rectified linear unit function, returns f(x) = max(0, x)
  - Accuracy: 60%
- 'identity', no-op activation, useful to implement linear bottleneck, returns f(x) = x
  - Accuracy: 59%
- 'logistic', the logistic sigmoid function, returns f(x) = 1 / (1 + exp(-x)).
  - Accuracy: 61%
- tanh', the hyperbolic tan function, returns f(x) = tanh(x).
  - Accuracy: 62%

Well awesome, we just found not one but Two activation functions that improved our overall accuracy! The function f(x) = tanh(x) worked best as an activation function.

Okay now with the f(x) = tanh(x) Activation function and (5,2) tuple as our layer size, we will try to find the highest accuracy for a Learning rate.

Learning Rate:

- 'constant' - constant learning rate
  - 62%
- 'invscaling'- gradually decreases the learning rate at each time step 't'
  - 62%
- 'adaptive'- constant learning rate, but wach time two consecutive epochs fail to decrease training loss by at least tol, or fail to increase validation score by at least tol if 'early_stopping' is on, the current learning rate is divided by 5.

        - 62%

Finally, we will experiment with different types of solvers. We have used solver='lbfgs' so far in our experiments. We will revert the Learning Rate back to the constant type.

- 'Lbfgs' - an optimizer in the family of quasi-Newton methods.
    - 62%
- 'sgd' - stochastic gradient descent.
    - 45%
    - 60%. With changing the learning_rate_init (and making the learning rate adaptive)
        - "Learning_rate_init" is the initial learning rate used. It controls the step-size in updating the weights.
        - When increasing the learning_rate_init from the defalt 0.001, the accuracy gradually gets better and better. At learning_rate_init = 0.551, the accuracy is at 60%. However at 0.553, the accuracy drops down to 47% and then continues to go up and down but never past 60% from what I can tell.
- 'adam' – a special stochastic gradient-based optimizer
    - 56%

From what I have found, the best way to improve the accuracy with this data set is through a hyperbolic tan function as the activation function. 62% isn't much better than 60%, but I was impressed I was even able to improve the accuracy.