UDACITY

PROJECT

Your first neural network

A part of the Deep Learning Nanodegree Foundation Program

PROJECT REVIEW

CODE REVIEW

NOTES

SHARE YOUR ACCOMPLISHMENT!

Requires Changes

1 SPECIFICATION REQUIRES CHANGES

Great job overall! You are very close to having your first neural network implemented!
All you have to work on now is tunning your hyperparameters.
Please aim for a validation loss that is between 0.2-0.15.

I can't wait to see your next submission!

Code Functionality

All the code in the notebook runs in Python 3 without failing, and all unit tests pass.

Good job passing all the unit tests!
This is good practice for Test Driven Development, where you write your tests out before you write the code, to make sure that your code behaves as you intend once you've written it! this is especially applicable in difficult programming exercises like this one, where a small syntax or mathematical error would be hard to find.

The sigmoid activation function is implemented correctly

SUGGESTION

Great job! I like that you used the lambda functions in pythong! You can also implement this function separately, which will make it easier for you to change the activation function or use it somewhere else in another layer of your network

```
def sigmoid(x):
     return 1/(1+np.exp(-x))

def sigmoid_prime(x):
     return sigmoid(x) * (1.0 - sigmoid(x))
```

This will make it easier to change the activation function or reuse it in your network!

Forward Pass

The input to the hidden layer is implemented correctly in both the train and run methods.

Good work! Everything looks correct!

The output of the hidden layer is implemented correctly in both the `train` and `run` methods.

SUGGESTION

Perhaps try to have more consistency between your `train` and `run` functions.

The input to the output layer is implemented correctly in both the train and run methods.

The output of the network is implemented correctly in both the train and run methods.

Backward Pass

The network output error is implemented correctly

AWESOME

Backpropagation is one of the hardest parts of this project, and you implemented it perfectly. Good job!

Updates to both the weights are implemented correctly.

Hyperparameters

The number of epochs is chosen such the network is trained well enough to accurately make predictions but is not overfitting to the training data.

I strongly recommend that you increase the number of epochs until you see a clear plateau in your validation loss.

A good way to choose the epoch number is to observe the plot of training and validation loss. You want an epoch number that will allow the neural network to learn to its full (or close to full) potential. Graphically, this means that you want to see your learning and validation loss plateau. At the same time, you do not want to over-train your network, which will lead to overfitting to your training sample. An indication of overfitting is your validation loss beginning to increase after the plateau.

To make the fluctuation in the validation loss a bit clearer, you can set the y-axis to be between 0 and 0.5
`_ = plt.ylim(ymax=0.5)`

The number of hidden units is chosen such that the network is able to accurately predict the number of bike riders, is able to generalize, and is not overfitting.

The number of hidden nodes to include in a neural network is still an open question and is more a matter of art and field knowledge than science at this point. With that being said, there are some good rules of thumb regarding the hidden layer node numbers, one of which is that the number of your nodes should usually be somewhere between the number of input (52) and output (1) nodes.

The fewer nodes you include, the harder it will be for your network to generalize.
The more nodes you include, the more data you will need to train the network.

I would recommend at least 10 nodes in the hidden layer for this problem. I tried a few numbers myself, and I found that numbers around 10-13 give the best results. Of course, this is only from experimental results, and you should try some of your own!

Here is a good thread that talks about the number of nodes in a hidden layer.

The learning rate is chosen such that the network successfully converges, but is still time efficient.

The learning rates for neural networks tend to be between 0.01 and 0.001, with some neural networks having much smaller learning rates (or variable learning rates!). In this case, we divide the learning rate by `n_records` when we update the weights.

```
self.weights_hidden_to_output += self.lr * delta_weights_h_o / n_records
self.weights_input_to_hidden += self.lr * delta_weights_i_h / n_records
```

Since `n_records` is 128, we effectively reduce the learning rate by a factor of 100. This is why the real learning rate in this case is the quotient of `learning_rate/n_records`. So a learning rate of 0.2 is actually closer to ~0.002 due to this division. This is a side effect of using gradient descent (batch processing of inputs).

Here's a thread that you might enjoy: Neural Network Learning Rate and Batch Weight Update. It covers things that we don't need to know for this project, so you can skip that kind of stuff.

RESUBMIT