UDACITY

< Back to Deep Learning

Predicting Bike-Sharing Patterns

REVIEW

CODE REVIEW

HISTORY

Meets Specifications

Overall this is an outstanding submission ! Well done on completing your first neural network :)

The model is a bit overfitted to correctly classify the holiday period. One reasonable answer is the model fails to generalize the Christmas period (aka Holiday period) because it has seen this period only once before (Dec 2011).

Code Functionality

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

Great work. Unit testing is one of the most reliable methods to ensure that your code is free from all bugs without getting confused with the interactions with all the other code. If you are interested, you can read up more and I hope that you will continue to use unit testing in every module that you write to keep it clean and speed up your development.

But always keep in mind, that unit tests cannot catch every issue in the code. So your code could have bugs even though unit tests pass.

The sigmoid activation function is implemented correctly

Awesome, your sigmoid is implemented correctly!

Further, to understand the drawbacks of `sigmoid` and the popular choice `ReLU`, refer to this link.
Understanding the drawbacks of sigmoid and why it's not used anymore will help you develop better models.

Forward Pass

The forward pass is correctly implemented for the network's training.

Nice implementation of the matrix multiplication of the input-to-hidden weights. You can also introduce bias term in your work.
Bias is a value that allows you to to shift the activation function to the left or right.
Here is one of the many resources that you can look up to know more about the bias term.

You can implement the bias by initializing with small constant value and add to the matrix multiplication of the input-to-hidden weights and the inputs as follows:

```
# Initialize the bias term
self.bias = 0.01

### In both the train and run methods you can add the bias term
np.dot(self.weight_input_to_hidden, inputs) + self.bias
```

The run method correctly produces the desired regression output for the neural network.

You didn't apply an activation here. Which is correct.

Sometimes you do need an activation on the output. Can you think of when? Think about a self-driving car. If you are trying to predict the steering angle, you want to make sure that it is bounded so that the car won't try and steer too sharply. In this case, you would use an activation function. Such as the tanh, to make sure that it is bounded. Just food for thought

Backward Pass

The network correctly implements the backward pass for each batch, correctly updating the weight change.

Woah ! Really an awesome job done implementing the backward pass. Being one of the most difficult to understand topics in Neural Networks, you implemented it perfectly

understand topics in Neural Networks, you implemented it perfectly.

Updates to both the input-to-hidden and hidden-to-output weights are implemented correctly.

Congrats! Updating the weights correctly is the most important part of the algorithm 👏🏽

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.

Perfect ! As per the current architecture of the network, ~ 2500 `iterations` performs well.

Pro Tip: Epoch tuning can also be done using **early stopping**. This basically means stopping the training process when a certain validation score has been reached. This helps prevent overfitting. Check it out.

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.

16 is a good choice for `hidden_nodes`.

Although there is no mathematical explanation for how many hidden units should be used, there have been many empirical studies which suggest that a good rule of thumb is to try an average of the number of input and output nodes. So, for this problem, it should be around 25 (as average is around 28). However, this should not be taken as a hard and fast rule and should only be used as a guideline to design the architecture.

The optimal number of hidden units can be arrived at by using cross-validation along with grid/randomized search. You can read this link to further understand the process of hyperparameter optimization. In the same link, there is a section called Babysitting the learning process which gives a guideline on how to train a network. It is an excellent read packed with tonnes of good stuff.

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

Given the network size, 0.8 is a fairly good learning rate.

Pro Tip: Another approach generally employed is **gradual decay of learning rate** after each (another hyperparameter) epoch. This way, when weights are close to the minima, we take a small step in direction of minima and not overshoot it.

The number of output nodes is properly selected to solve the desired problem.

The training loss is below 0.09 and the validation loss is below 0.18.

DOWNLOAD PROJECT

RETURN TO PATH

Rate this review