UDACITY

PROJECT

Your first neural network

A part of the Deep Learning Nanodegree Foundation Program

PROJECT REVIEW

CODE REVIEW

NOTES

SHARE YOUR ACCOMPLISHMENT!

Meets Specifications

DISCLAIMER: I'm not the previous grader, and I reviewed all your submission for your benefit

Excellent job on the coding part of the assignment! You seem to have understood how the algorithm works and how to efficiently implement it in Python. Congratulations!

Also good job choosing the hyperparameters for your model. There's still room for improvement, if you are up for a challenge, follow the guidelines mentioned above and try to lower the validation loss and reduce the training time (spoiler: I've seen values as low as 0.120 and your training time shouldn't take more than 1-2 minutes tops).

Concerning your final comments, you are on the right track: The reason why you see that behaviour is because we are training the model with data from one distribution (bike riding data from "normal" months), and validating with data from other distribution (bike riding data from the last weeks of december). That's why the model can't "generalize" what he learned (from typical yearly demand) to a very specific season of the year where the demand is very different.

What was done with this assignment is conceptually wrong (but done for pedagogic purposes) - when training a model we should procure that the distribution (the characteristics) of the data used for that, must be as similar as possible as the data we are going to do the predictions.

As a final bonus, I'm sharing this reading about the bias-variance tradeoff, if you are interested. Hope you like it!
http://www.holehouse.org/mlclass/10_Advice_for_applying_machine_learning.html

Keep up with the good work!

Code Functionality

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

Good job! Well done

The sigmoid activation function is implemented correctly

Ok, correctly implemented! Notice that for conciseness you could have also create a lambda (anonymous) function: `self.activation_function = lambda x: 1/(1+np.exp(-x))`

Forward Pass

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

You got this right. Good use of numpy.dot for the matrix multiplication.

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

Correct, here you are activating the inputs with the sigmoid (since this is a regression task)

Correct, here you are activating the inputs with the sigmoid (since this is a regression task)

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

Nice! as you can see the dot function and the data structures simplify coding.

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

You got this right, since we are working on a regression model, the final output is just the raw input to the output layer.

Backward Pass

The network output error is implemented correctly

Good! As simple as that, just a subtraction of numpy arrays.

Updates to both the weights are implemented correctly.

Awesome, you avoided a lot of problems here, and also your code is elegant, using the np.dot function and the .T function. This takes advantage of the fact that the line inputs = np.array(inputs_list, ndmin=2).T ensures that `inputs` has two dimensions.

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.

Good job! Your plot shows how training and validation loss go decreasing. Also you took care of overfitting, your validation curve doesn't seem to go up. On the other hand, you should notice on your model that around epoch 15000, although your training loss goes down, the validation loss keeps stabilized (on average, since it seems that its variance increases). Once you get to a level of accuracy for your validation set, no more additional training will benefit your model - you will be losing precious time training.

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.

Good choice of number of hidden units! Basically you need a number not so low (since you want a number of nodes enough to create interactions between features), but also not so high that your model would be "tied" to training data, not being able to generalize to unseen data. A good rule of thumb is the half way in between the number of input and output units. See here for more info - https://www.quora.com/How-do-I-decide-the-number-of-nodes-in-a-hidden-layer-of-a-neural-network

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

A good range for experimenting learning rates is between 0.001 and 0.1 When it's too large, the network doesn't converge since the weight update steps are too large. On the other hand a lower learning rate just makes the network slower to converge without adding any benefit. Read more about it here: https://www.quora.com/In-an-artificial-neural-network-algorithm-what-happens-if-my-learning-rate-is-wrong-too-high-or-too-low

In your case, take care of this: you picked a very high number of epochs, and a very slow learning rate. It takes lot of time to learn! Consider that the metric of a good model, many times is not only its accuracy, but also its training time. Maybe you could try reducing the epochs and increasing the learning rate and compare results.

DOWNLOAD PROJECT

Have a question about your review? Email us at review-support@udacity.com and include the link to this review.

RETURN TO PATH

Rate this review

Student FAQ