UDACITY

‹ Return to Classroom

Predicting Bike-Sharing Patterns

REVIEW

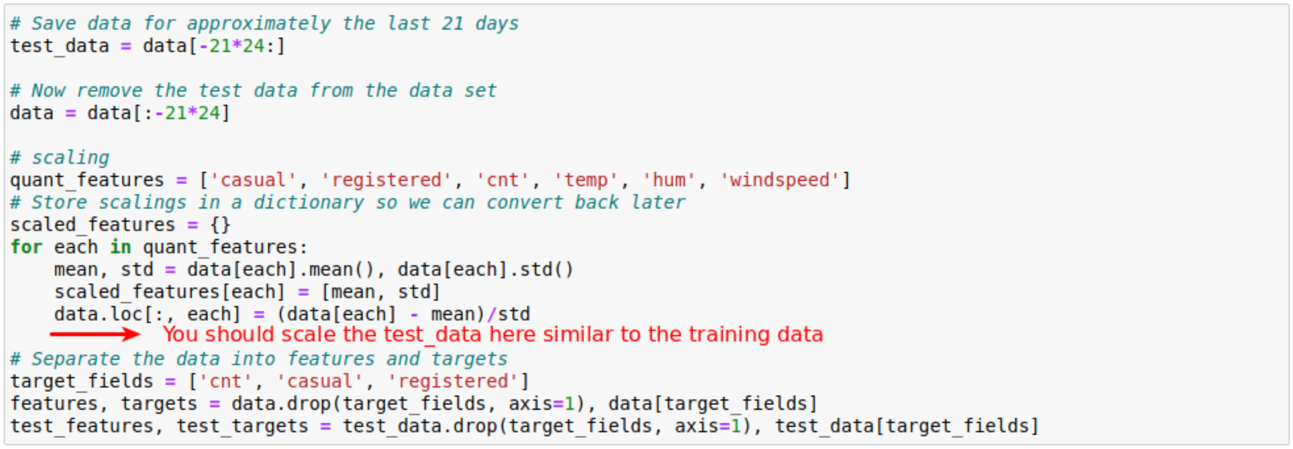
CODE REVIEW

HISTORY

Meets Specifications

Well Done !!! This is a fantastic submission. Congratulations on completing the project !!!

Regarding your predictions, you will see that your predictions come out to be flat even after the network has converged to the recommended levels. This is because you have forgotten to scale the test data.



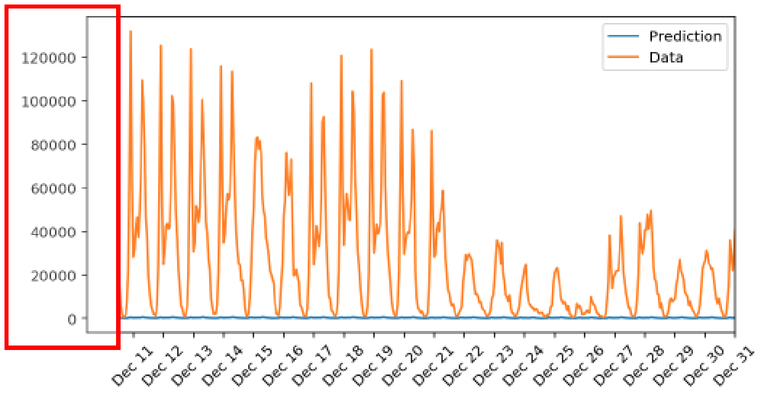
```
# Save data for approximately the last 21 days
test_data = data[-21*24:]

# Now remove the test data from the data set
data = data[:-21*24]

# scaling
quant_features = ['casual', 'registered', 'cnt', 'temp', 'hum', 'windspeed']
# Store scalings in a dictionary so we can convert back later
scaled_features = {}
for each in quant_features:
    mean, std = data[each].mean(), data[each].std()
    scaled_features[each] = [mean, std]
    data.loc[:, each] = (data[each] - mean)/std
```

→ You should scale the test_data here similar to the training data

```
# Separate the data into features and targets
target_fields = ['cnt', 'casual', 'registered']
features, targets = data.drop(target_fields, axis=1), data[target_fields]
test_features, test_targets = test_data.drop(target_fields, axis=1), test_data[target_fields]
```

Since you missed to scale the targets for test data, the predictions seem to be flatlined. It is also evident from the y-axis of your final predictions.

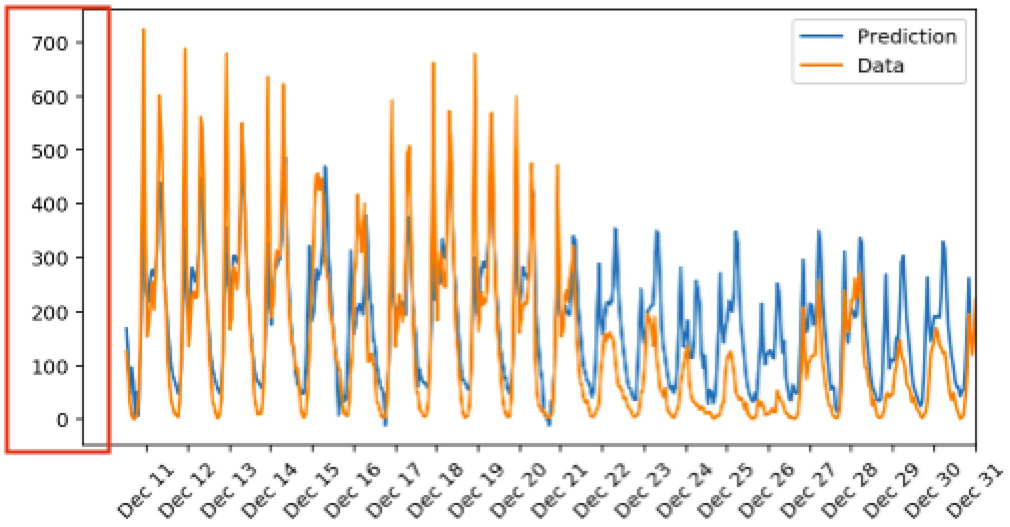
```
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:10: DeprecationWarning:
.ix is deprecated. Please use
.loc for label based indexing or
.iloc for positional indexing

See the documentation here:
http://pandas.pydata.org/pandas-docs/stable/indexing.html#ix-indexer-is-deprecated
  # Remove the CWD from sys.path while we load stuff.
```



To correct the issue, you can simply add the following line in the for loop.

```
test_data.loc[:, each] = (test_data[each] - mean) / std
```

Once corrected, you should get predictions like below (see, how the scale also changes):



There is one suggestion I would like to make though. It is always a good idea to think about the predictions made by your model. This is true with machine learning in general and with black-box models like neural networks in particular as these models do not provide any insight into how they are arriving at their decision. So, what observations can you make about your prediction? Is the model prediction good? What about the predictions in the latter part of the month? Why does the model seem to overpredict the number of bike riders? These are all good questions we should be thinking about.

To understand how backpropagation works on an intuitive level, I will recommend that you watch this excellent talk by Tariq Rashid. He also has a book that you can check out if you want to look at the details in the talk. Another important resource that I would like to suggest about understanding backprop from a mathematical standpoint is

the online book by Michael Nielsen. I find that it is easier to understand and go through after going through the talk though.

All the best for your next submission. Keep learning !!!

Code Functionality

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

Good Job !!! All unit tests pass for implementation.

The sigmoid activation function is implemented correctly

Good Job !!! If you want to learn about other kinds of activation functions, please refer to this.

Forward Pass

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

Good Job!!! The forward pass has been implemented correctly. Please note that we are using an identity function as an activation function for the final output layer as the task is a regression one.

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

The `run` method is basically the same thing as the forward pass. It should simply take the input features and pass them through the trained model to get the regression outputs. It performs the same weighted sum calculation which you are implementing using the dot product.

Backward Pass

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

The basic idea of the backward pass is to tweak the model parameters based on the error encountered during the forward pass. So the network is basically a function of the weights. When we get the error (in this project, a mean squared loss), we try to attribute its source to the corresponding weights (as inputs and outputs into the model are fixed during the whole training phase). So, we adjust these weights based on how much they actually contributed to the error. You correctly calculated the derivatives for both the output layer and the hidden layer. Well Done !!!

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

Nice Work !!! You have correctly updated the weights. Please note that we are using mini-batch SGD and accumulating the errors for all records in the batch before making an update. That is why we are scaling the final weight updates by the number of records. The learning rate decides how quickly the network learns. in this case, both the learning rate and the number of records determine the final step size for the weight update.

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 !!! The currently chosen number of epochs does the job.

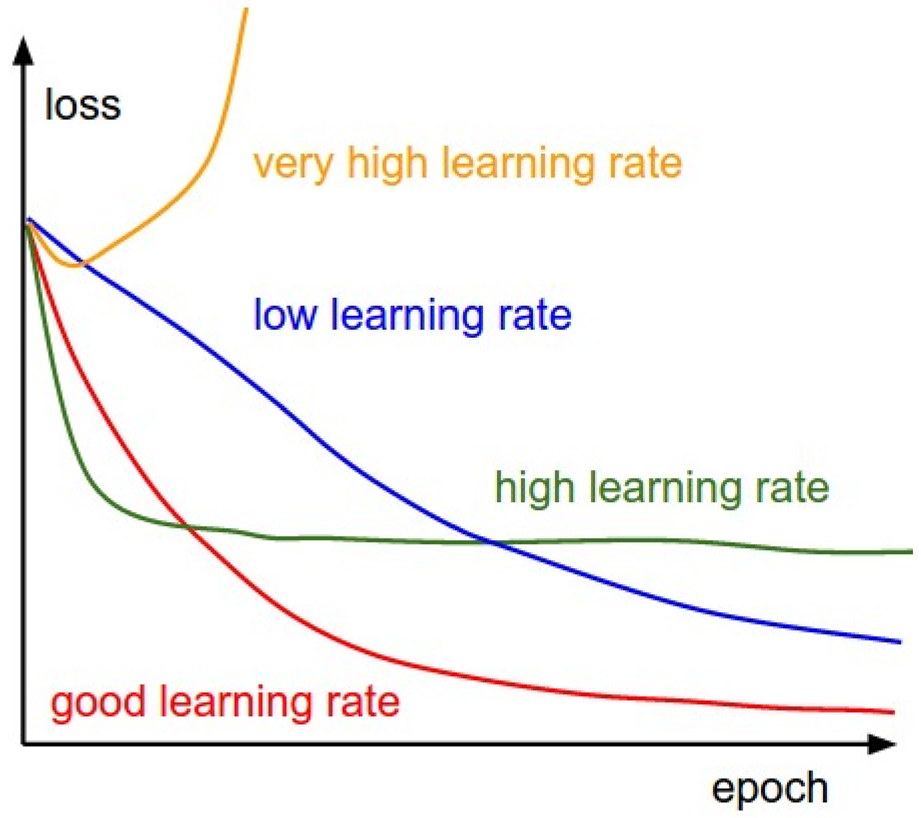
The number of hidden units is chosen such that the network is able to accurately predict the number of bike riders, is able to generalize, and is not overfitting.

You have chosen a good number of hidden units. Well Done !!!

Anything between 8 and 20 works quite well for this problem. By now, you must have understood that the number of hidden units to be used in a network is fairly open and there is no mathematical formula that provides the exact size of the hidden layer. However, there have been empirical studies that suggest some heuristics that can be used to help make this decision. One of the most common heuristics is to use the average of the input and output nodes. You can take a look at the whole discussion.

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

The learning rate that you are using is high. This can be seen from the learning curve where the error decreases rapidly initially and then flattens out with a neck. Below is a cartoon representation of different learning rates taken from cs231n class notes that should help you choose the right learning rate always.



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

Well Done !!! Since this is a regression problem, we only need a single output node.

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

The training and validation loss are well below the acceptable limits. Well Done !!!

DOWNLOAD PROJECT

RETURN TO PATH

Rate this review

START