

[◀ Return to Classroom](#)[DISCUSS ON STUDENT HUB](#)

Predicting Bike-Sharing Patterns

REVIEW


CODE REVIEW 1


HISTORY

Meets Specifications

Dear Student,

I am really impressed with the amount of effort you've put into the project. You deserve applaud for your hardwork!

 Finally, Congratulations on completing this project. You are one step closer to finishing your Nanodegree.

Wishing you good luck for all future projects 

Some general suggestions

Use of assertions and Logging:

- Consider using [Python assertions](#) for sanity testing - assertions are great for catching bugs. This is especially true of a dynamically type-checked language like Python where a wrong variable type or shape can cause errors at runtime
- Logging is important for long-running applications. Logging done right produces a report that can be analyzed to debug errors and find crucial information. There could be different levels of logging or logging tags that can be used to filter messages most relevant to someone. Messages can be written to the terminal using `print()` or saved to file, for example using the [Logger module](#). Sometimes it's worthwhile to catch and log exceptions during a long-running operation so that the operation itself is not aborted.

Debugging:

- Check out this guide on [debugging in python](#)

Reproducibility:


- Reproducibility is perhaps the biggest issue in machine learning right now. With so many moving parts present in the code (data, hyperparameters, etc) it is imperative that the instructions and code make it easy for anyone to get exactly the same results (just imagine debugging an ML pipeline where the data changes every time and so you cannot get the same result twice).
- Also consider using random seeds to make your data more reproducible.

Optimization and Profiling:

- Monitoring progress and debugging with [Tensorboard](#): This tool can log detailed information about the model, data, hyperparameters, and more. Tensorboard can be used with Pytorch as well.

Code Functionality

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

All unit tests have passed successfully 


Suggestion:

You can export your conda environment into `environment.yaml` file so that you can recreate your conda environment later while practicing on your own system. Use the following command -

```
conda env export -f environment.yaml
```

The sigmoid activation function is implemented correctly

```
self.activation_function = lambda x : 1.0 / (1.0 + np.exp(-x))
```

You've correctly implemented the Sigmoid function via lambda expression 

Read about the differences between [Sigmoid vs Softmax function here](#)

Note - Lambda functions are a concise way to accomplish simple tasks. You can read more about this elegant concept [on this blog post](#)

Forward Pass

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

Forward pass has been correctly implemented in the project.

Note: Generally, in neural networks, the hidden layer activation function is largely chosen to be `ReLU`. Functions like `Sigmoid` and `TanH` are preferred for the final layers but generally not used for hidden layers to avoid vanishing gradient problem.

```
def forward_pass_train(self, x):  
    ''' Implement forward pass here  
  
    Arguments  
    -----  
    x: features batch  
  
    ...  
  
    ##### Implement the forward pass here #####  
    ### Forward pass ###  
    # TODO: Hidden layer - Replace these values with your calculations.  
    hidden_inputs = np.dot(x, self.weights_input_to_hidden) # signals into  
    hidden_outputs = self.activation_function(hidden_inputs) # signals from  
  
    # TODO: Output layer - Replace these values with your calculations.  
    final_inputs = np.dot(hidden_outputs, self.weights_hidden_to_output) #  
    final_outputs = final_inputs # signals from final output layer  
  
    return final_outputs, hidden_outputs
```

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

The `run` method takes the `dot` product of input features and associated weights, passes the result through the activation function.

These steps are repeated again for the final layer. Good job!

💡 Suggestion - Instead of rewriting the same code for `run` function. You could just call the `forward_pass_train` function and return the output

```
def run(self, features):  
  
    final_outputs, _ = self.forward_pass_train(features)  
  
    return final_outputs
```

Backward Pass

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

Excellent work implementing the backward pass.

During the backward pass (also called backpropagation), the networks weights are adjusted based on the gradients at each layer. These gradients are accumulated based on the calculated loss. It tells how much this particular weight affected the error. Then we finally adjust the weights to reduce the loss.

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

```
def update_weights(self, delta_weights_i_h, delta_weights_h_o, n_records):  
    ''' Update weights on gradient descent step  
  
    Arguments  
    -----  
    delta_weights_i_h: change in weights from input to hidden layers  
    delta_weights_h_o: change in weights from hidden to output layers  
    n_records: number of records  
  
    ...  
  
    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 #
```

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.

Epochs = 3600

A decent choice for the number of epochs to train the model.

Note - Higher epochs should help with learning, although at one point the learning would stop as the model becomes really good at the training data.

From then on, there is a risk that model will start fitting on training noise, therefore leading to a overfit model that performs well on training data but fails at producing sensible results for external data.

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.

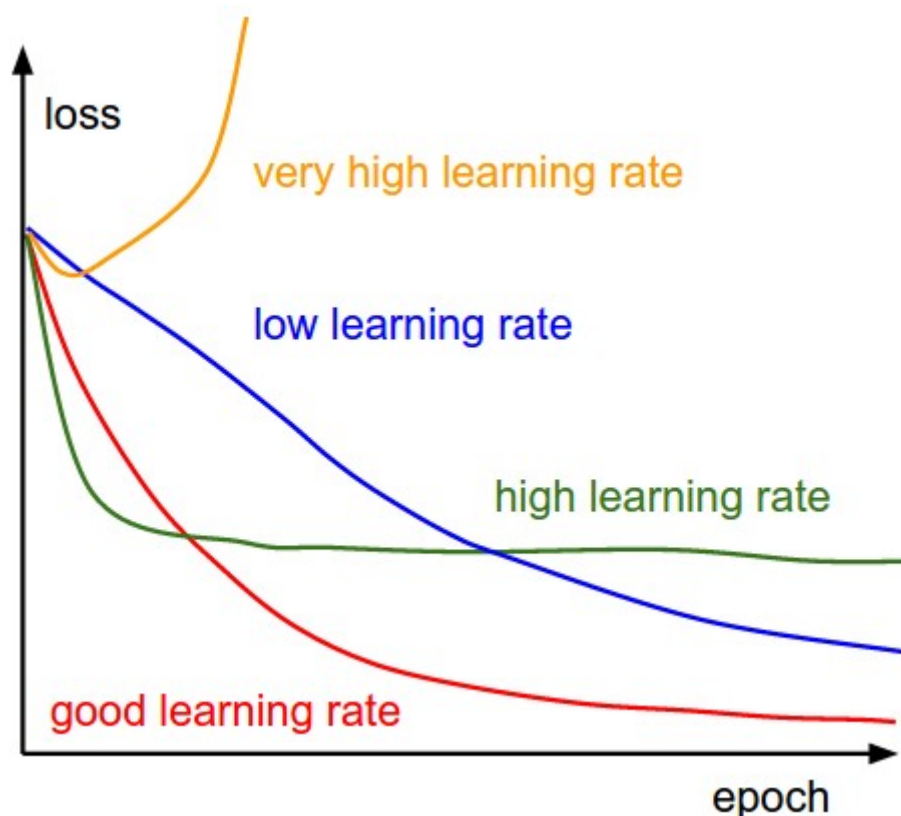
Good choice of hidden units.

Hidden units in a range of to work well for this problem.

When it comes to hyperparameters, there is no universal answer to what works well. Therefore, it's best to experiment with a range of different values to check which hyperparameters result in the best model. You can implement Cross Validation techniques such as GridSearchCV or RandomSearchCV that automate the task of finding the best parameters among a list of parameters that the user specifies.

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

Here's a curve of loss vs epochs for different learning rates. You can see that for high learning rates the loss might decrease initially but stagnates or rapidly goes up again as the number of epochs increases.



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

The number of is set to 1.

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

Progress: 100.0% ... Training loss: 0.082 ... Validation loss: 0.172

Both training and validation loss meet the required criteria. Good job!

Training loss: 0.082

Validation loss: 0.172

Note - If you wish to develop a solid understanding of Neural Network fundamentals, I highly recommend watching [3Blue1Brown's videos on Neural Networks](#)

 [DOWNLOAD PROJECT](#)

1 [CODE REVIEW COMMENTS](#)



RETURN TO PATH

Rate this review

START