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

4 SPECIFICATIONS REQUIRE CHANGES

Great submission. I can see that you've already spent time tuning up the hyper parameters. Being a great deep learning engineer is all about tweaking the parameters and learning an intuition towards each of the parameter and being able to identify which knob to turn. Your off to a great start. I hope that you would spend some more time with each of the parameters and see for yourself some of the behaviour when you intentionally push the knob too far in any of the ways.

Your interpretation of the model is correct -- in fact, if you investigated the dataset is more detail, you would find that there was a lack of examples around the holiday season!

Code Functionality

All the code in the notebook runs in Python 3 without failing.

Great. Everything checks out.

The sigmoid activation function is implemented correctly

Nice work.

The activation function can also be implemented with the lambda function.

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

All unit tests must be passing

Great work. We hope that you've come to realize how effective unit tests can be. 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 (there are 3 links) 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.

Forward Pass

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

Nice job implementing the matrix multiplication of the weights and inputs. You could also include a bias term in the layer if you wanted to.

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

Good job. I hope you've understood the significance of this small piece of code in your model. For fun, you can try to remove this activation function (change the gradients correspondingly) and try running your model. The non-linearity that this activation adds into the model is the essence of what enables our model to search a solution space that's much more than what a purely linear model could, and it's these activation functions that has made deep learning so powerful.

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.

This is a regression model, so the output is simply the raw input to the output layer. Nice work!

Backward Pass

The network output error is implemented correctly

The error propagated back to the hidden layer is implemented correctly

Fantastic work! Most students tend to struggle here because understanding how back-prop works is quite difficult. You've masterfully implemented this. Spending time to really understanding this will really lay the foundation to your future as a deep learning engineer.

Updates to both the weights are implemented correctly.

Nice work implementing back propagation correctly. This is a hard part of the project, and may look messy depending on how you defined some of your earlier variables. Great job!

Hidden layer gradient(hidden_grad) is calculated correctly.

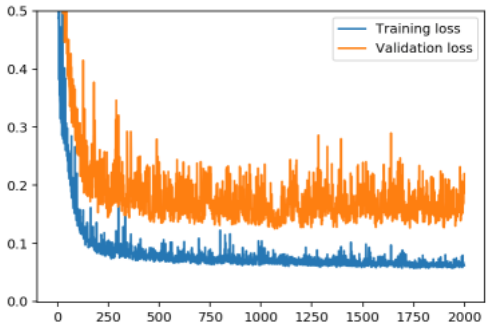
While your code works correctly, the formula used to calculate `hidden_grad` is not the gradient. It is the backpropagated error multiplied by the gradient. While this doesn't affect the output, this will make it difficult for you to understand what the code does when you revisit the code later on.

Please change the hidden_grad to match the actual definition, which is the gradient of the hidden units activation function;
`hidden_grad = hidden_outputs * (1.0 - hidden_outputs)`. Remember to change the weight updates step too after making these changes!

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.

The following is the loss graph that I saw when I ran your model for 2000 epochs at my end.



There are signs of mild overfitting starting at around 1000 epochs, and cutting the training short at 400 epochs and calling it a day is probably the best your model can do. But the large swings in the validation loss curve suggest their are other issues that need to be addressed.

The fluctuations might be caused by too high of a learning rate, which is pushing the weights around by too much, preventing the model weights from converging to a good minima. Or they might be the result of a model that is not complex enough to learn the functions it needs to, i.e. it's solution space just does't contain a good solution to the problem it is trying to solve.

See the next sections for more.

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.

8 hidden units is not enough for this project.

In order for the network to be able to generalize it has to be at least 10.

A good rule of thumb is the halfway in between the number of input and output units.

How many input units are there?
How many output units?

There's a good answer here for how to decide the number of nodes in the hidden layer.
https://www.quora.com/How-do-I-decide-the-number-of-nodes-in-a-hidden-layer-of-a-neural-network

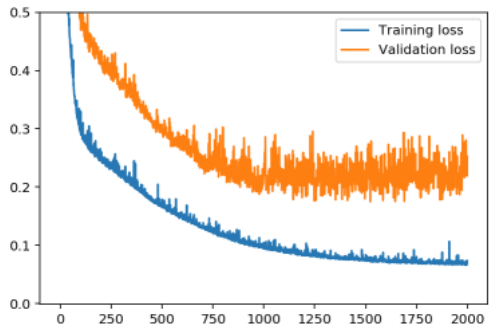
The learning rate is chosen such that the network successfully converges, but is still time efficient.

The learning rate is too high. The weight updates are throwing the accuracy of the model around and we see wild fluctuations in the accuracy of model from one step to the next. Because the learning rate is too high the weights are not able to converge.

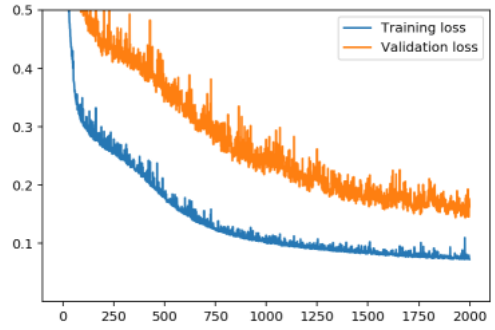
Decreasing the learning rate will allow the model to train longer and it will be better equipped to converge to a local minimum. It will also require more epochs, but we have already addressed that, and are ready to increase the epoch count again if we see it as necessary.

Here's a sequence of loss graphs showing what happened when I decreased the learning rate to 0.01 and ran with increasing numbers of hidden units.

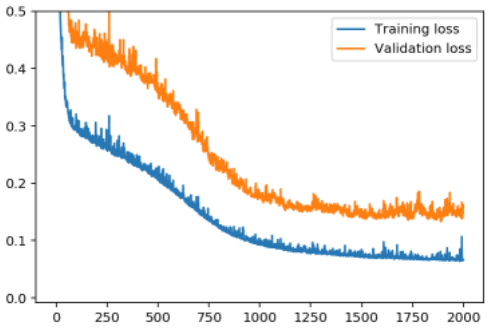
8 hidden units

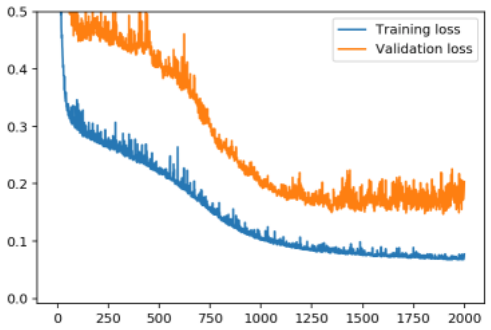


10 hidden units

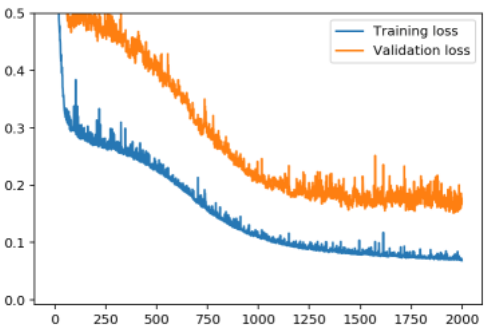


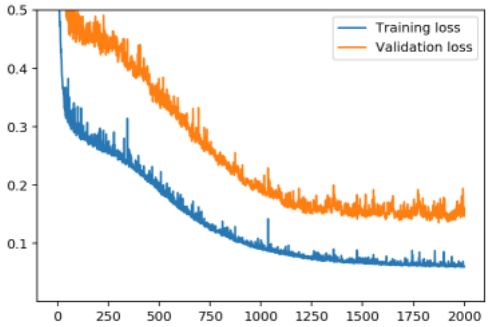
12 hidden units - 2 runs





14 hidden units - 2 runs





The graphs for 12 and 14 hidden units show that sometimes the validation losses can still exhibit fluctuations near the end of training. This indicates there is room to reduce the learning rate some more, or the number of hidden units should be increased, or a combination of both.

That is as far as I go. Next you should try increasing the number of hidden units and experiment with different learning rates and hidden unit combinations.

Here is a great reference for evaluating learning rate, from this source

RESUBMIT

DOWNLOAD PROJECT

Learn the best practices for revising and resubmitting your project.

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

RETURN TO PATH

Student FAQ