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

Nice job on this project! On the last question: You got it! We don't have enough examples of holidays for the network to learn about them (and the `holiday` variable is incorrectly labeled for the days around Christmas and Thanksgiving). For dealing with holidays with the small amount of training data available over the holiday time periods, we could use a RNN as you mentioned, time-lagged features or fix the variable for holiday/not holiday (they only label the 22nd and 25th of December a holiday). There's also the possibility of oversampling the data on Christmas and Thanskgiving holidays. The network also over-predicts for Thanksgiving if you look back further into the validation set.

Code Functionality

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

The sigmoid activation function is implemented correctly

Great! Scipy also has a function for this.

All unit tests must be passing

Forward Pass

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

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

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.

Nice work! This is by no means required, but just something nice to know for the future: the entire forward pass can actually be incorporated into a function, then you can call it from both the train and run functions. It would then follow the DRY principle (which as a programmer, I bet you already know about), and reduce chances for errors. If you see something like that in the future in a project, feel free to change it to improve upon the code.

Backward Pass

The network output error is implemented correctly

Updates to both the weights are implemented correctly.

Nice job overall here, you got the big-picture math correct! One small detail: In this case, we're calling the hidden gradient just the derivative of the sigmoid function, so `hidden_grad` should be just the derivative of the sigmoid function. That mean `output_grad` would just be 1.

Other people sometimes call the entire derivative of the error with respect to the weight the gradient.

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.

Nice job! One way to make it easier to visually see if the losses are increasing is by adding gridlines:

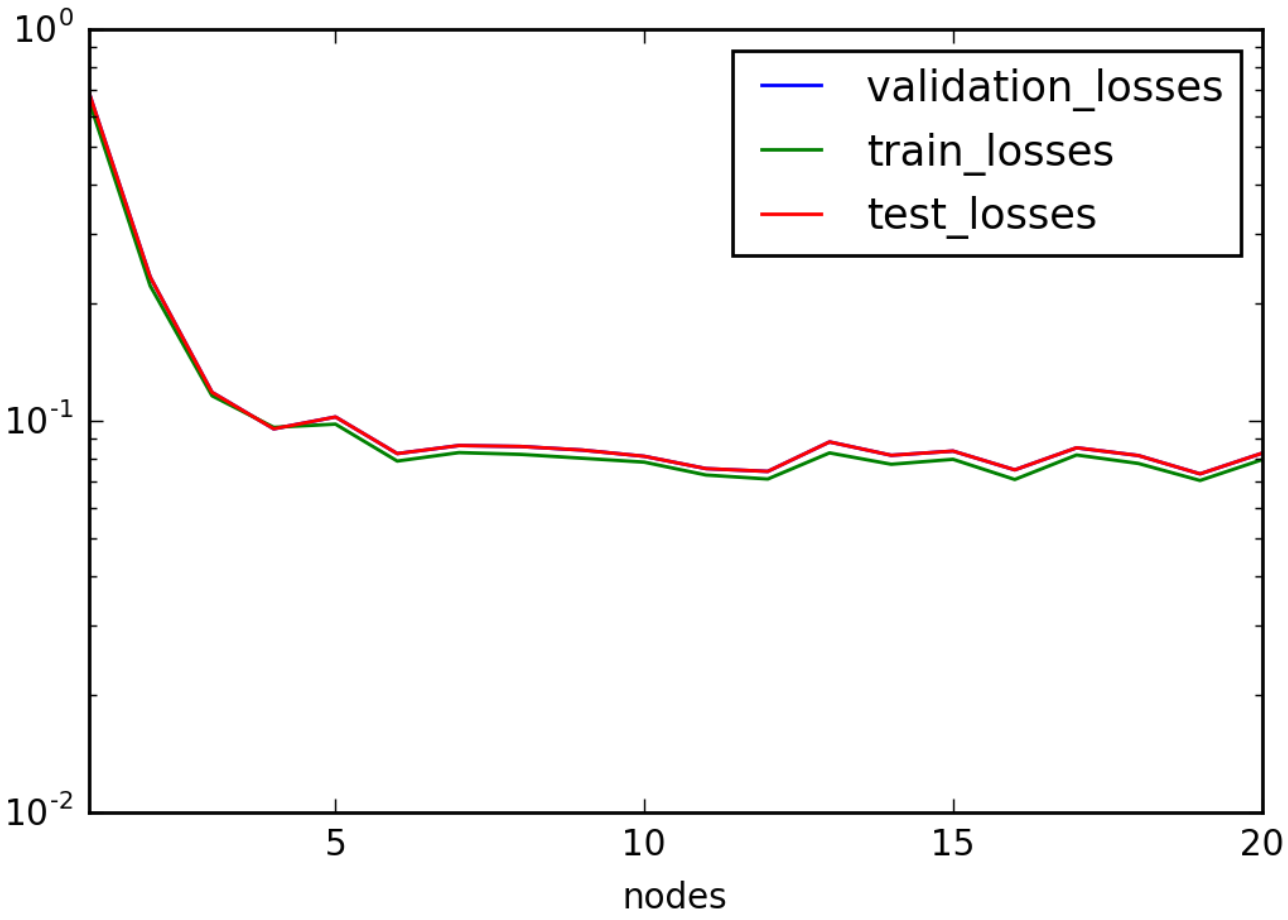
```
plt.plot(losses['train'], label='Training loss')
plt.plot(losses['validation'], label='Validation loss')
ax = plt.gca()
ax.grid(True)
plt.legend()
plt.ylim(ymax=0.5)
```

Here's an example of where to stop during training (the 'early stopping' section with the plot; the 'Early stopping point' is the ideal place to stop 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.

Great job tuning hidden units! Anything above 10 is acceptable by the specifications, but actually 6 is the minimum I go by (see the plot below). A good rule of thumb for number of hidden units is half way between the number of input and output units. Another rule of thumb is to stick to numbers less than twice the number of input units (112 in this case). Here's another resource on the subject.

If you're curious, try different numbers of hidden units, recording the validation error for each, and plot the number of hidden units on the x-axis and validation loss on the y-axis (or try it on another neural net project). That way you can *quantitatively* judge what the optimum number of hidden units is. For reproducibility, consider setting the random seed. You should end up seeing that the losses flatten out after around 5/6 hidden nodes, which is much smaller than the rule of thumb value of 28. Most likely this is because all the dummy variables are not independent of each other. In actuality, there's only about 13 independent variables.



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

For your learning rate, 0.1 is on the edge of being too high because sometimes it won't converge (and the losses are very jumpy in each training step), but below 0.001 will take a long time to converge. Usually 0.01 is a good starting point, and depending on your situation, you can increase it until it doesn't converge (in other words, the training and validation loss never settle out to a flat, low number), or the validation error is increased compared with 0.01. I think you may be able to get a lower loss with a smaller learning rate. You should see the jumpiness of the validation loss decrease.

Here's more info on tuning the learning rate.

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

Student FAQ