PROJECT

Your first neural network

A part of the Deep Learning Nanodegree Foundation Program

PROJECT REVIEW | CODE REVIEW | NOTES

Meets Specifications

SHARE YOUR ACCOMPLISHMENT

Congrats!! 🎉 Outstanding work!

You corrected the issues suggested by the previous reviewer and the network seems to be working wonderfully! With a fair number of hidden units and an appropriate number of epochs for the learning rate, you achieve good validation error (this predicts how the network will behave with new data).

The number of records is used to compensate the iterations we do in the *for* loop over the number of feature in the train dataset X. You can see further details in the review.

I tried to give you some insights and resources to go a little beyond and I really hope I was able to help you out.

You have built a solid knowledge for going deeper into the exciting deep learning world! Good journey!! ⛵

> Please, be sure to rate my work! Hope you enjoy the rest of the course! Cheers! 🍻

Code Functionality

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

Your code passed the automatic unit tests. Great job! 👏

✓ **The sigmoid activation function is implemented correctly**

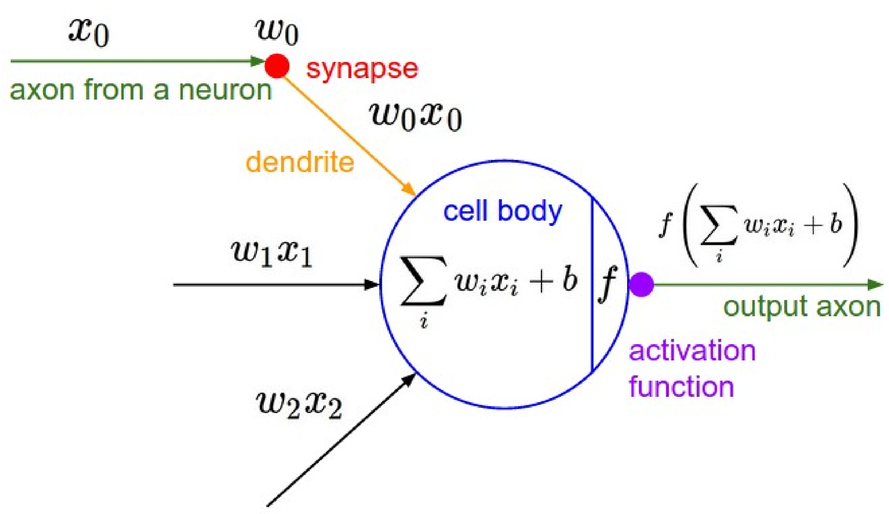
The sigmoid is well defined and implemented as a python `lambda` function, which helps to get shorter and more generic code. Well done here! 😎

Forward Pass

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

Nicely done! Student implemented the first hidden layer using numpy function `dot()` for matrix multiplication! The bias term could also be added to have a comprehensive a scheme.

Remember a layer of a neural network consists of a number of neurons connecting one layer to the next one, each one like this:



There you see how a simple bias term can be added before passing the input to the activation function.

Reference: Stanford CS231 Convolutional Neural Networks

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

Good! 👏 Both in `train` and `run` methods the output of the layers is obtained from the activation function, in the form `self.activation_function(hidden_inputs)`, which calls the sigmoid function defined in the constructor of the `NeuralNetwork` class.

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

The input to the output layer is correctly implemented, as done with the input to the hidden layer 👍. Same comments as in that part, bias could be added in this layer too.

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

Good work!! 👍 Since we are dealing with a regression problem and not classification, the output does not need to be limited to [0,1] interval. The output is the raw input to the output layer, with no function applied.

Backward Pass

✓ **The network output error is implemented correctly**

Output errors calculated correctly!! 🎉

The output error is calculated as the target minus the output of the network.

✓ **Updates to both the weights are implemented correctly.**

Everything is perfectly implemented 👏

- `hidden_error` term is well implemented, as the backpropagation of the output error.
- `hidden_error_term` is correct, taking into account the derivative of the sigmoid is sigmoid(x)(1-sigmoid(x)).
- `output_error_term` is correct, it is just the output `error`.
- The weights increments for the batch `delta_weights_i_h` and `delta_weights_h_o` are correct, taking into account the dimensions.
- Also, both `self.weights_hidden_to_output` and `self.weights_inputs_to_hidden` weights are updated properly and the number of records is considered. Well done in this part 👍

> **The operation of dividing by the number of records is important since we have iterated over the n_features (or the number of elements in the training dataset X), and hence, updated `delta_weights_*` n_features times. This is why it is needed to compensate it in the weights update.**

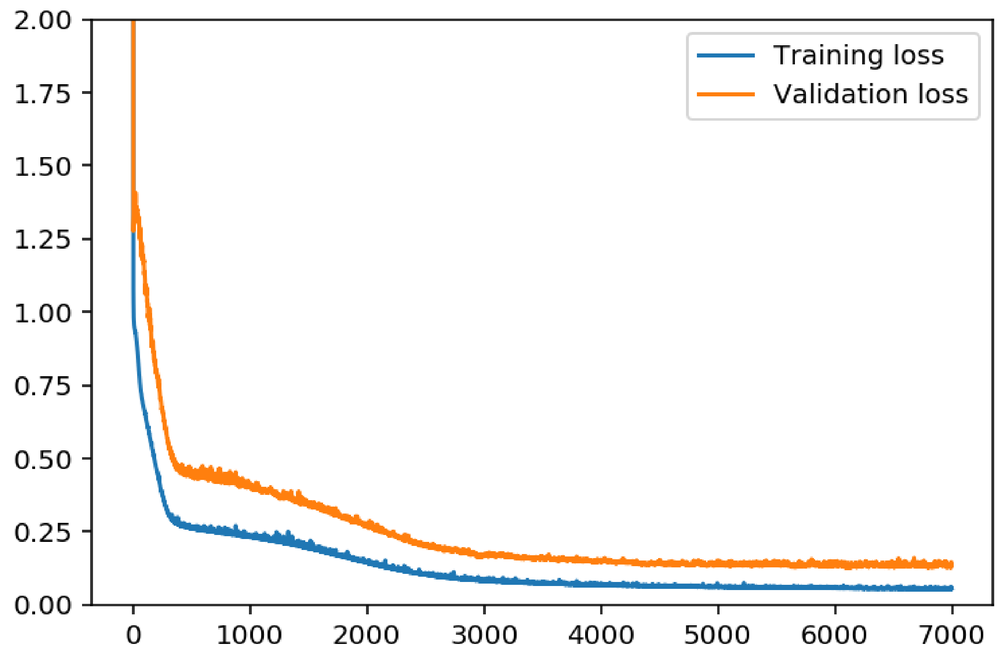
Hyperparameters

✓ **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.**

Awesome! 👌 The number of epochs is enough and there is no overfitting (the testing loss curve follows the training loss curve!). Both training and validation loss are quite low! The network has reached almost a stationary state and is stable. You could even push harder and achieve a better value of performance.

The training set error is low and the validation error is not increasing! Validation loss is below 0.2, that's wonderful!

> 👉 In order to be able to judge better, it is good set the vertical limits to a reasonable range (i.e. `plt.ylim(0,2)`) to focus on the important part and be able to compare between curves with detail. You set ymax=0.5, but that may limit the vertical span of the graph too much and does not set a lower limit (losses cannot be less than 0).
> I did set the limits and this is how your graph looks like now:



We can see much clearly now that the loss curves have reached a stationary value 😎.

✓ **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.**

The number of hidden units is appropriate for the case. A good rule of thumb is to use no more than twice the number of input units, while trying to keep this number as low as possible so the network can generalize, so probably at least 8 units.

Always keep in mind not to overload a hidden layer with more units than necessary.

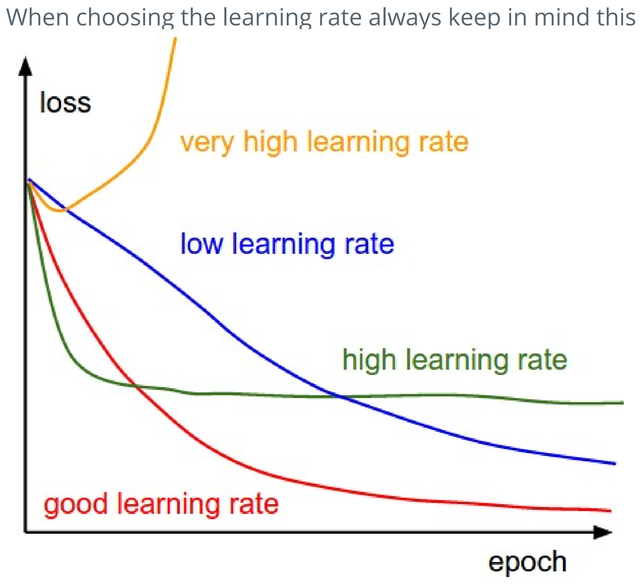
25 units is a fair number. Well done here!!! 👏

You can have more information on how to choose a proper number of hidden units in this link from a Quora discussion.

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

Well done here! Good compromise value for the learning rate 👌

We have to be careful here, since high learning rate may result in the network failing to converge while small learning rates may guarantee the convergence but at the expense decreasing the speed of convergence.

> When choosing the learning rate always keep in mind this graph as a reference:
>
> 
>
>
> Reference Image 1

The value of the learning rate depends on the network, more complex networks (higher number of hidden units) can put up with higher learning rates. A simpler network may miss a lot with high learning rates. You can tell that a learning rate is high when there is a big step at the beginning and sometimes, when spikes appear in the curve, due to the network having some difficulties to find the minimum.
Values range from 0.5-0.01. It is not a fixed value, it is design. You have a lot of options here 😉.

❕ **0.5 can be a good value for a neural network with 25 hidden units paired up with a good number of epochs 👌.**

Usually, the procedure to get a good performance is the following: once the number of hidden units is set, we try to change the learning rate so we achieve a curve like the red one. Then, we need to Increase the number of epochs and then see what is the minimum validation loss we get when the curve is stable.

- If it is more than 0.2: then we should change the learning rate so we can go further and start the process again.
  Sometimes the learning rate has to be decreased and other times increased. One thing you can do to know that, is to try first to decrease the learning rate and see if the minimum loss achieved is lower or not than before. If it is higher, then you should have increased the learning rate instead.
- Once you improve the validation loss, see what is the best value achieved in the stable state, and if it is less than 0.2, we're finished! 🙌

> You have more info in this link from the CS231 course on Convolutional Neural Networks from Stanford.

DOWNLOAD PROJECT

RETURN TO PATH