Meets Specifications

Congratulations! I can see you've put a lot of time and thought into your solutions, and provided some good observations about your results.

Additionally, when you are tuning the hyper parameters it is important that you experiment a little so that you can observe the affects of the hyper parameters on the learning curve. A little trial and error may be needed to get the network performing well!

A really great start to your Deep Learning Nanodegree Foundation! Awesome work 

Looking forward to your next project submission!

PS: If you would like to read more, the following is a great article with some additional resources for moving forward with deep learning: http://blog.deepgram.com/how-to-get-a-job-in-deep-learning/

Code Functionality

All the code in the notebook runs in Python 3 without failing.
Your code passed the automatic unit tests. Great job!
The sigmoid activation function is implemented correctly
Great job!

ADDITIONAL NOTE

Alternatively, you could implement the activation function with a lambda function:
`self.activation_function = lambda x: 1/(1+np.exp(-x))`

All unit tests must be passing
Great work getting all the unit tests to pass.

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.

 Great job working out that the output should be the raw input to the output layer!

Backward Pass

The network output error is implemented correctly

The error propagated back to the hidden layer is implemented correctly
Well done. Your solution works perfectly!

ADDITIONAL NOTE

Alternatively, given there is only one output unit in this case, you could have implemented like this:
`hidden_errors = self.weight_hidden_to_output * output_errors`

Additionally, your solution works because you only have one output unit... should you have more than one output unit you would need to slightly adjust you solution by applying the transpose function to `self.weights_hidden_to_output`:
`hidden_errors = np.dot(self.weight_hidden_to_output.T, output_errors)`
Updates to both the weights are implemented correctly.
Super! Great work applying the transpose to the inputs... this is really important here and can be a little tricky to work out.

ADDITIONAL NOTE

For this particular network (being a single sample) your solution of a mixture of element-wise updating and matrix multiplication on the weights works. However, this would not always be the case, so it is better practice to solve this by doing matrix multiply:

`self.weights_input_to_hidden += self.learning_rate * np.dot(hidden_errors * hidden_grad, inputs.T)`
Hidden layer gradient(hidden_grad) is calculated correctly.

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.
A good starting point!

SUGGESTION

Please note that this is the number of times the dataset will pass through the network, each time updating the weights. As the number of epochs increases, the network becomes better and better at predicting the targets in the training set. You'll need to choose enough epochs to train the network well but not too many or you'll be overfitting.

You chose 100 as your number of epochs, which is a good start but a relatively small amount of epochs. Try increasing the number in order to improve your network. Perhaps experiment with values around a thousand and then beyond and observe the affect on the learning curve.
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.
Perfect!

SUGGESTION

A good rule of thumb for deciding the number of nodes is halfway in between the number of input and output nodes. For this network the number of hidden nodes tend to range from 10 up to 30.

A great answer to how to decide the number of nodes in the hidden layer can be found in this post: 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.
This is a great starting point!

SUGGESTION

Please note that the learning rate scales the size of weight updates. If this is too big, the weights tend to explode and the network fails to fit the data. A good choice to start at is 0.1. If the network has problems fitting the data, try reducing the learning rate. Note that the lower the learning rate, the smaller the steps are in the weight updates and the longer it takes for the neural network to converge.

You chose 0.1 as your learning rate, which is a good starting point. However, sometimes, the network doesn't converge when it is around 0.1. The weight update steps are too large with this learning rate and the weights end up not converging. Try reducing the rate in order to improve your network by getting the learning rate just low enough to get the network to converge.

Try values around 0.01 and observe the affect on the learning curve.