DLND Your first neural network-Copy1

April 20, 2017

1 Your first neural network

In this project, you'll build your first neural network and use it to predict daily bike rental ridership. We've provided some of the code, but left the implementation of the neural network up to you (for the most part). After you've submitted this project, feel free to explore the data and the model more.

```
In [1]: %matplotlib inline
        %config InlineBackend.figure_format = 'retina'

        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
```

1.1 Load and prepare the data

A critical step in working with neural networks is preparing the data correctly. Variables on different scales make it difficult for the network to efficiently learn the correct weights. Below, we've written the code to load and prepare the data. You'll learn more about this soon!

```
In [2]: data_path = 'Bike-Sharing-Dataset/hour.csv'

        rides = pd.read_csv(data_path)

In [3]: rides.head()
```

```
Out[3]:    instant      dteday  season  yr  mnth  hr  holiday  weekday  workingday  \
        0        1  2011-01-01       1   0     1   0        0        6           0
        1        2  2011-01-01       1   0     1   1        0        6           0
        2        3  2011-01-01       1   0     1   2        0        6           0
        3        4  2011-01-01       1   0     1   3        0        6           0
        4        5  2011-01-01       1   0     1   4        0        6           0

           weathersit  temp   atemp   hum  windspeed  casual  registered  cnt
        0           1  0.24  0.2879  0.81        0.0       3          13   16
        1           1  0.22  0.2727  0.80        0.0       8          32   40
        2           1  0.22  0.2727  0.80        0.0       5          27   32
        3           1  0.24  0.2879  0.75        0.0       3          10   13
        4           1  0.24  0.2879  0.75        0.0       0           1    1
```

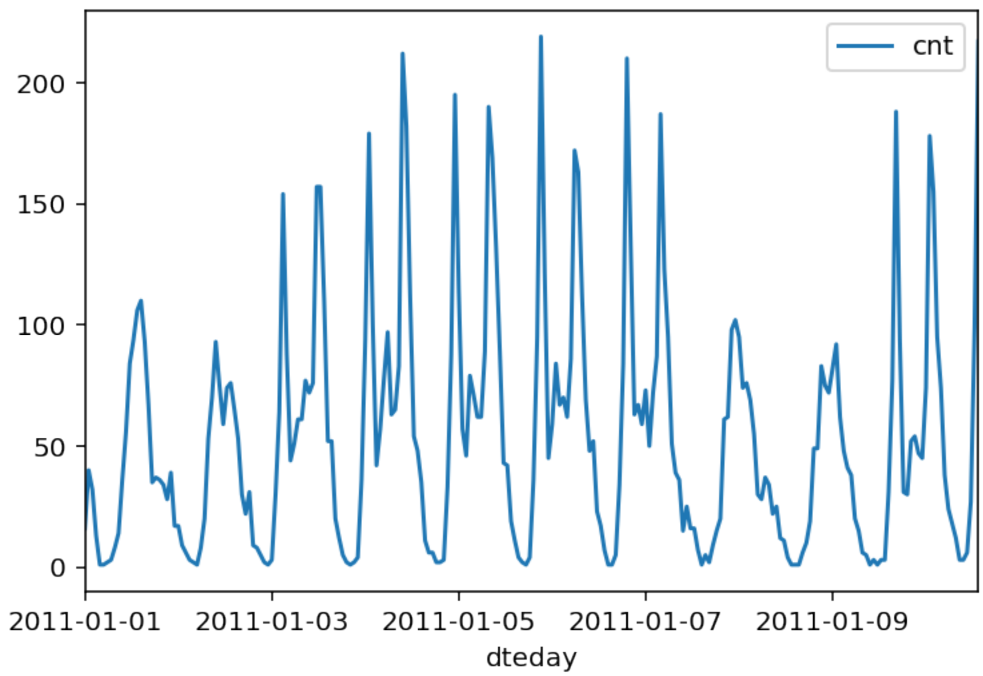
1.2 Checking out the data

This dataset has the number of riders for each hour of each day from January 1 2011 to December 31 2012. The number of riders is split between casual and registered, summed up in the `cnt` column. You can see the first few rows of the data above.

Below is a plot showing the number of bike riders over the first 10 days or so in the data set. (Some days don't have exactly 24 entries in the data set, so it's not exactly 10 days.) You can see the hourly rentals here. This data is pretty complicated! The weekends have lower over all ridership and there are spikes when people are biking to and from work during the week. Looking at the data above, we also have information about temperature, humidity, and windspeed, all of these likely affecting the number of riders. You'll be trying to capture all this with your model.

```
In [4]: rides[:24*10].plot(x='dteday', y='cnt')
```

```
Out[4]: <matplotlib.axes._subplots.AxesSubplot at 0x7f4a1b61db38>
```



1.2.1 Dummy variables

Here we have some categorical variables like season, weather, month. To include these in our model, we'll need to make binary dummy variables. This is simple to do with Pandas thanks to `get_dummies()`.

```
In [273]: dummy_fields = ['season', 'weathersit', 'mnth', 'hr', 'weekday']
          for each in dummy_fields:
              dummies = pd.get_dummies(rides[each], prefix=each, drop_first=False)
              rides = pd.concat([rides, dummies], axis=1)
```

```python
fields_to_drop = ['instant', 'dteday', 'season', 'weathersit',
                  'weekday', 'atemp', 'mnth', 'workingday', 'hr']
data = rides.drop(fields_to_drop, axis=1)
data.head(50)
```

Out[273]:

	yr	holiday	temp	hum	windspeed	casual	registered	cnt	season_1 \
0	0	0	0.24	0.81	0.0000	3	13	16	1
1	0	0	0.22	0.80	0.0000	8	32	40	1
2	0	0	0.22	0.80	0.0000	5	27	32	1
3	0	0	0.24	0.75	0.0000	3	10	13	1
4	0	0	0.24	0.75	0.0000	0	1	1	1
5	0	0	0.24	0.75	0.0896	0	1	1	1
6	0	0	0.22	0.80	0.0000	2	0	2	1
7	0	0	0.20	0.86	0.0000	1	2	3	1
8	0	0	0.24	0.75	0.0000	1	7	8	1
9	0	0	0.32	0.76	0.0000	8	6	14	1
10	0	0	0.38	0.76	0.2537	12	24	36	1
11	0	0	0.36	0.81	0.2836	26	30	56	1
12	0	0	0.42	0.77	0.2836	29	55	84	1
13	0	0	0.46	0.72	0.2985	47	47	94	1
14	0	0	0.46	0.72	0.2836	35	71	106	1
15	0	0	0.44	0.77	0.2985	40	70	110	1
16	0	0	0.42	0.82	0.2985	41	52	93	1
17	0	0	0.44	0.82	0.2836	15	52	67	1
18	0	0	0.42	0.88	0.2537	9	26	35	1
19	0	0	0.42	0.88	0.2537	6	31	37	1
20	0	0	0.40	0.87	0.2537	11	25	36	1
21	0	0	0.40	0.87	0.1940	3	31	34	1
22	0	0	0.40	0.94	0.2239	11	17	28	1
23	0	0	0.46	0.88	0.2985	15	24	39	1
24	0	0	0.46	0.88	0.2985	4	13	17	1
25	0	0	0.44	0.94	0.2537	1	16	17	1
26	0	0	0.42	1.00	0.2836	1	8	9	1
27	0	0	0.46	0.94	0.1940	2	4	6	1
28	0	0	0.46	0.94	0.1940	2	1	3	1
29	0	0	0.42	0.77	0.2985	0	2	2	1
30	0	0	0.40	0.76	0.1940	0	1	1	1
31	0	0	0.40	0.71	0.2239	0	8	8	1
32	0	0	0.38	0.76	0.2239	1	19	20	1
33	0	0	0.36	0.81	0.2239	7	46	53	1
34	0	0	0.36	0.71	0.2537	16	54	70	1
35	0	0	0.36	0.66	0.2985	20	73	93	1
36	0	0	0.36	0.66	0.1343	11	64	75	1
37	0	0	0.36	0.76	0.1940	4	55	59	1
38	0	0	0.34	0.81	0.1642	19	55	74	1
39	0	0	0.34	0.71	0.1642	9	67	76	1
40	0	0	0.34	0.57	0.1940	7	58	65	1

41	0	0	0.36	0.46	0.3284	10	43	53	1
42	0	0	0.32	0.42	0.4478	1	29	30	1
43	0	0	0.30	0.39	0.3582	5	17	22	1
44	0	0	0.26	0.44	0.3284	11	20	31	1
45	0	0	0.24	0.44	0.2985	0	9	9	1
46	0	0	0.22	0.47	0.1642	0	8	8	1
47	0	0	0.22	0.44	0.3582	0	5	5	1
48	0	0	0.20	0.44	0.4179	0	2	2	1
49	0	0	0.16	0.47	0.3881	0	1	1	1

	season_2	...	hr_21	hr_22	hr_23	weekday_0	weekday_1	weekday_2	\
0	0	...	0	0	0	0	0	0	
1	0	...	0	0	0	0	0	0	
2	0	...	0	0	0	0	0	0	
3	0	...	0	0	0	0	0	0	
4	0	...	0	0	0	0	0	0	
5	0	...	0	0	0	0	0	0	
6	0	...	0	0	0	0	0	0	
7	0	...	0	0	0	0	0	0	
8	0	...	0	0	0	0	0	0	
9	0	...	0	0	0	0	0	0	
10	0	...	0	0	0	0	0	0	
11	0	...	0	0	0	0	0	0	
12	0	...	0	0	0	0	0	0	
13	0	...	0	0	0	0	0	0	
14	0	...	0	0	0	0	0	0	
15	0	...	0	0	0	0	0	0	
16	0	...	0	0	0	0	0	0	
17	0	...	0	0	0	0	0	0	
18	0	...	0	0	0	0	0	0	
19	0	...	0	0	0	0	0	0	
20	0	...	0	0	0	0	0	0	
21	0	...	1	0	0	0	0	0	
22	0	...	0	1	0	0	0	0	
23	0	...	0	0	1	0	0	0	
24	0	...	0	0	0	1	0	0	
25	0	...	0	0	0	1	0	0	
26	0	...	0	0	0	1	0	0	
27	0	...	0	0	0	1	0	0	
28	0	...	0	0	0	1	0	0	
29	0	...	0	0	0	1	0	0	
30	0	...	0	0	0	1	0	0	
31	0	...	0	0	0	1	0	0	
32	0	...	0	0	0	1	0	0	
33	0	...	0	0	0	1	0	0	
34	0	...	0	0	0	1	0	0	
35	0	...	0	0	0	1	0	0	
36	0	...	0	0	0	1	0	0	

37	0	...	0	0	0	1	0	0
38	0	...	0	0	0	1	0	0
39	0	...	0	0	0	1	0	0
40	0	...	0	0	0	1	0	0
41	0	...	0	0	0	1	0	0
42	0	...	0	0	0	1	0	0
43	0	...	0	0	0	1	0	0
44	0	...	1	0	0	1	0	0
45	0	...	0	1	0	1	0	0
46	0	...	0	0	1	1	0	0
47	0	...	0	0	0	0	1	0
48	0	...	0	0	0	0	1	0
49	0	...	0	0	0	0	1	0

	weekday_3	weekday_4	weekday_5	weekday_6
0	0	0	0	1
1	0	0	0	1
2	0	0	0	1
3	0	0	0	1
4	0	0	0	1
5	0	0	0	1
6	0	0	0	1
7	0	0	0	1
8	0	0	0	1
9	0	0	0	1
10	0	0	0	1
11	0	0	0	1
12	0	0	0	1
13	0	0	0	1
14	0	0	0	1
15	0	0	0	1
16	0	0	0	1
17	0	0	0	1
18	0	0	0	1
19	0	0	0	1
20	0	0	0	1
21	0	0	0	1
22	0	0	0	1
23	0	0	0	1
24	0	0	0	0
25	0	0	0	0
26	0	0	0	0
27	0	0	0	0
28	0	0	0	0
29	0	0	0	0
30	0	0	0	0
31	0	0	0	0
32	0	0	0	0

```
33          0          0          0          0
34          0          0          0          0
35          0          0          0          0
36          0          0          0          0
37          0          0          0          0
38          0          0          0          0
39          0          0          0          0
40          0          0          0          0
41          0          0          0          0
42          0          0          0          0
43          0          0          0          0
44          0          0          0          0
45          0          0          0          0
46          0          0          0          0
47          0          0          0          0
48          0          0          0          0
49          0          0          0          0

[50 rows x 110 columns]
```

1.2.2 Scaling target variables

To make training the network easier, we'll standardize each of the continuous variables. That is, we'll shift and scale the variables such that they have zero mean and a standard deviation of 1.

The scaling factors are saved so we can go backwards when we use the network for predictions.

```
In [6]: quant_features = ['casual', 'registered', 'cnt', 'temp', 'hum', 'windspeed']
        # Store scalings in a dictionary so we can convert back later
        scaled_features = {}
        for each in quant_features:
            mean, std = data[each].mean(), data[each].std()
            scaled_features[each] = [mean, std]
            data.loc[:, each] = (data[each] - mean)/std
```

1.2.3 Splitting the data into training, testing, and validation sets

We'll save the data for the last approximately 21 days to use as a test set after we've trained the network. We'll use this set to make predictions and compare them with the actual number of riders.

```
In [7]: # Save data for approximately the last 21 days
        test_data = data[-21*24:]

        # Now remove the test data from the data set
        data = data[:-21*24]

        # Separate the data into features and targets
        target_fields = ['cnt', 'casual', 'registered']
```

```
features, targets = data.drop(target_fields, axis=1), data[target_fields]
test_features, test_targets = test_data.drop(target_fields, axis=1), test_data[target_fi
```

We'll split the data into two sets, one for training and one for validating as the network is being trained. Since this is time series data, we'll train on historical data, then try to predict on future data (the validation set).

In [8]:
```
# Hold out the last 60 days or so of the remaining data as a validation set
train_features, train_targets = features[:-60*24], targets[:-60*24]
val_features, val_targets = features[-60*24:], targets[-60*24:]
```

1.3 Time to build the network

Below you'll build your network. We've built out the structure and the backwards pass. You'll implement the forward pass through the network. You'll also set the hyperparameters: the learning rate, the number of hidden units, and the number of training passes.

The network has two layers, a hidden layer and an output layer. The hidden layer will use the sigmoid function for activations. The output layer has only one node and is used for the regression, the output of the node is the same as the input of the node. That is, the activation function is $f(x) = x$. A function that takes the input signal and generates an output signal, but takes into account the threshold, is called an activation function. We work through each layer of our network calculating the outputs for each neuron. All of the outputs from one layer become inputs to the neurons on the next layer. This process is called *forward propagation*.

We use the weights to propagate signals forward from the input to the output layers in a neural network. We use the weights to also propagate error backwards from the output back into the network to update our weights. This is called *backpropagation*.

> **Hint:** You'll need the derivative of the output activation function ($f(x) = x$) for the backpropagation implementation. If you aren't familiar with calculus, this function is equivalent to the equation $y = x$. What is the slope of that equation? That is the derivative of $f(x)$.

Below, you have these tasks: 1. Implement the sigmoid function to use as the activation function. Set `self.activation_function` in `__init__` to your sigmoid function. 2. Implement the forward pass in the `train` method. 3. Implement the backpropagation algorithm in the `train` method, including calculating the output error. 4. Implement the forward pass in the `run` method.

In [222]:
```
class NeuralNetwork(object):
    def __init__(self, input_nodes, hidden_nodes, output_nodes, learning_rate):
        # Set number of nodes in input, hidden and output layers.
        self.input_nodes = input_nodes
        self.hidden_nodes = hidden_nodes
        self.output_nodes = output_nodes

        # Initialize weights
        self.weights_input_to_hidden = np.random.normal(0.0, self.input_nodes**-0.3,
                                        (self.input_nodes, self.hidden_nodes))
        #print("Weights input to hidden",self.weights_input_to_hidden)
        self.weights_hidden_to_output = np.random.normal(0.0, self.hidden_nodes**-0.3,
```

```
                                        (self.hidden_nodes, self.output_nodes))
        #print("Weights hidden to output",self.weights_hidden_to_output)
        self.lr = learning_rate

        def sigmoid(x):
            return 1 / (1 + np.exp(-x))
        self.activation_function = sigmoid

################################################################################
#
#  Point of this whole damn section belowis to allow the network to backpropogate, by
#  doing a forward pass, calculating the error, then back-propogating through the net.
#
################################################################################
    def train(self, features, targets):
        ''' Train the network on batch of features and targets.

            Arguments
            ---------

            features: 2D array, each row is one data record, each column is a feature
            targets: 1D array of target values

        '''
        n_records = features.shape[0]
        delta_weights_i_h = np.zeros(self.weights_input_to_hidden.shape)
        delta_weights_h_o = np.zeros(self.weights_hidden_to_output.shape)
        for X, y in zip(features, targets):
            ###############################################
            #Implement the forward pass here ####
            ### Forward pass ###
            ###############################################

            hidden_inputs = X @ self.weights_input_to_hidden #found this python 3.5 sc
            #np.dot(X, self.weights_input_to_hidden) # signals into hidden layer

            hidden_outputs = self.activation_function(hidden_inputs) # signals from hi

            #hidden_outputs = hidden_inputs * self.activation_function(hidden_inputs)

            final_inputs = hidden_outputs @ self.weights_hidden_to_output
            #np.dot(hidden_outputs, self.weights_hidden_to_output) # signals into fina
            final_outputs = final_inputs # signals from final output layer


            ###############################################
            #### Implement the backward pass here ####
            ### Backward pass ###
```

```
                ##############################################

                # Output layer error is the difference between desired, "y", and actual ou
                error = y - final_outputs
                grad_term = 1
                output_error_term = error * grad_term #gradient of error

                hidden_error = self.weights_hidden_to_output @ output_error_term
                hidden_error_term = hidden_error * hidden_outputs * ( 1 - hidden_outputs)

                #print("hidden_error_term:", hidden_error_term, "shape:", hidden_error_ter

                # Weight step (input to hidden)
                delta_weights_i_h += hidden_error_term * X[:, None]
                delta_weights_h_o += output_error_term * hidden_outputs[:, None]

            ################################## Update Weights ######################
            self.weights_hidden_to_output += self.lr * delta_weights_h_o / n_records # upd

            self.weights_input_to_hidden += self.lr * delta_weights_i_h / n_records # upda

    ####################################################################

        def run(self, features):
            ''' Run a forward pass through the network with input features

                Arguments
                ---------
                features: 1D array of feature values
            '''

            #### Implement the forward pass here ####

            hidden_inputs = features @ self.weights_input_to_hidden
            #np.dot(features,self.weights_input_to_hidden) # signals into hidden layer
            hidden_outputs = self.activation_function(hidden_inputs) # signals from hidden

            final_inputs = hidden_outputs @ self.weights_hidden_to_output
            #np.dot(hidden_outputs,self.weights_hidden_to_output) # signals into final out

            final_outputs = final_inputs # signals from final output layer

            return final_outputs
```

In [209]:
```
def MSE(y, Y):
    return np.mean((y-Y)**2)
```

1.4 Unit tests

Run these unit tests to check the correctness of your network implementation. This will help you be sure your network was implemented correctly before you starting trying to train it. These tests must all be successful to pass the project.

```
In [130]: %pdb
```

```
Automatic pdb calling has been turned OFF
```

```
In [216]: import unittest
          print("train_features.shape[1]", train_features.shape)
          inputs = np.array([[0.5, -0.2, 0.1]])
          print("test:inputs ",inputs,inputs.shape)
          targets = np.array([[0.4]])
          print("test:targets ",targets, targets.shape)
          test_w_i_h = np.array([[0.1, -0.2],
                                 [0.4, 0.5],
                                 [-0.3, 0.2]])
          test_w_h_o = np.array([[0.3],
                                 [-0.1]])

          class TestMethods(unittest.TestCase):

              ##########
              # Unit tests for data loading
              ##########

              def test_data_path(self):
                  # Test that file path to dataset has been unaltered
                  self.assertTrue(data_path.lower() == 'bike-sharing-dataset/hour.csv')

              def test_data_loaded(self):
                  # Test that data frame loaded
                  self.assertTrue(isinstance(rides, pd.DataFrame))

              ##########
              # Unit tests for network functionality
              ##########

              def test_activation(self):
                  network = NeuralNetwork(3, 2, 1, 0.5)
                  # Test that the activation function is a sigmoid
                  self.assertTrue(np.all(network.activation_function(0.5) == 1/(1+np.exp(-0.5))))

              def test_train(self):
                  # Test that weights are updated correctly on training
                  network = NeuralNetwork(3, 2, 1, 0.5)
```

```
                network.weights_input_to_hidden = test_w_i_h.copy()
                network.weights_hidden_to_output = test_w_h_o.copy()

                network.train(inputs, targets)
                self.assertTrue(np.allclose(network.weights_hidden_to_output,
                                            np.array([[ 0.37275328],
                                                      [-0.03172939]])))
                self.assertTrue(np.allclose(network.weights_input_to_hidden,
                                            np.array([[ 0.10562014, -0.20185996],
                                                      [0.39775194, 0.50074398],
                                                      [-0.29887597, 0.19962801]])))
            def test_run(self):
                # Test correctness of run method
                network = NeuralNetwork(3, 2, 1, 0.5)
                network.weights_input_to_hidden = test_w_i_h.copy()
                print("test: weights_input_to_hidden: ", network.weights_input_to_hidden,netwo
                network.weights_hidden_to_output = test_w_h_o.copy()
                print("test: weights_hidden_to_output: ", network.weights_hidden_to_output,net
                self.assertTrue(np.allclose(network.run(inputs), 0.09998924))

        suite = unittest.TestLoader().loadTestsFromModule(TestMethods())
        unittest.TextTestRunner().run(suite)
```

```
...

train_features.shape[1] (15435, 56)
test:inputs  [[ 0.5 -0.2  0.1]] (1, 3)
test:targets  [[ 0.4]] (1, 1)
test: weights_input_to_hidden:  [[ 0.1 -0.2]
 [ 0.4  0.5]
 [-0.3  0.2]] (3, 2)
test: weights_hidden_to_output:  [[ 0.3]
 [-0.1]] (2, 1)
run function's hidden_inputs: [[-0.06 -0.18]]
run function's hidden_outputs:  [[ 0.4850045   0.45512111]]
run function's final outputs:  [[ 0.09998924]]
Train: Delta_weights_i_h:  [[ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]] (3, 2)
Train: Delta_weights_h_o:  [[ 0.]
 [ 0.]] (2, 1)
Train: Shape of hidden inputs is:  [-0.06 -0.18] (2,)
Train: Shape of hidden outputs is:  [ 0.4850045   0.45512111] (2,)
Train: shape of final outputs:  [ 0.09998924]
self.weights_input_to_hidden  [[ 0.10562014 -0.20185996]
 [ 0.39775194  0.50074398]
 [-0.29887597  0.19962801]]
```

```
----------------------------------------------------------------------
Ran 5 tests in 0.013s

OK
```

```
Out[216]: <unittest.runner.TextTestResult run=5 errors=0 failures=0>
```

1.5 Training the network

Here you'll set the hyperparameters for the network. The strategy here is to find hyperparameters such that the error on the training set is low, but you're not overfitting to the data. If you train the network too long or have too many hidden nodes, it can become overly specific to the training set and will fail to generalize to the validation set. That is, the loss on the validation set will start increasing as the training set loss drops.

You'll also be using a method know as Stochastic Gradient Descent (SGD) to train the network. The idea is that for each training pass, you grab a random sample of the data instead of using the whole data set. You use many more training passes than with normal gradient descent, but each pass is much faster. This ends up training the network more efficiently. You'll learn more about SGD later.

1.5.1 Choose the number of iterations

This is the number of batches of samples from the training data we'll use to train the network. The more iterations you use, the better the model will fit the data. However, if you use too many iterations, then the model with not generalize well to other data, this is called overfitting. You want to find a number here where the network has a low training loss, and the validation loss is at a minimum. As you start overfitting, you'll see the training loss continue to decrease while the validation loss starts to increase.

1.5.2 Choose the learning rate

This 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.

1.5.3 Choose the number of hidden nodes

The more hidden nodes you have, the more accurate predictions the model will make. Try a few different numbers and see how it affects the performance. You can look at the losses dictionary for a metric of the network performance. If the number of hidden units is too low, then the model won't have enough space to learn and if it is too high there are too many options for the direction that the learning can take. The trick here is to find the right balance in number of hidden units you choose.

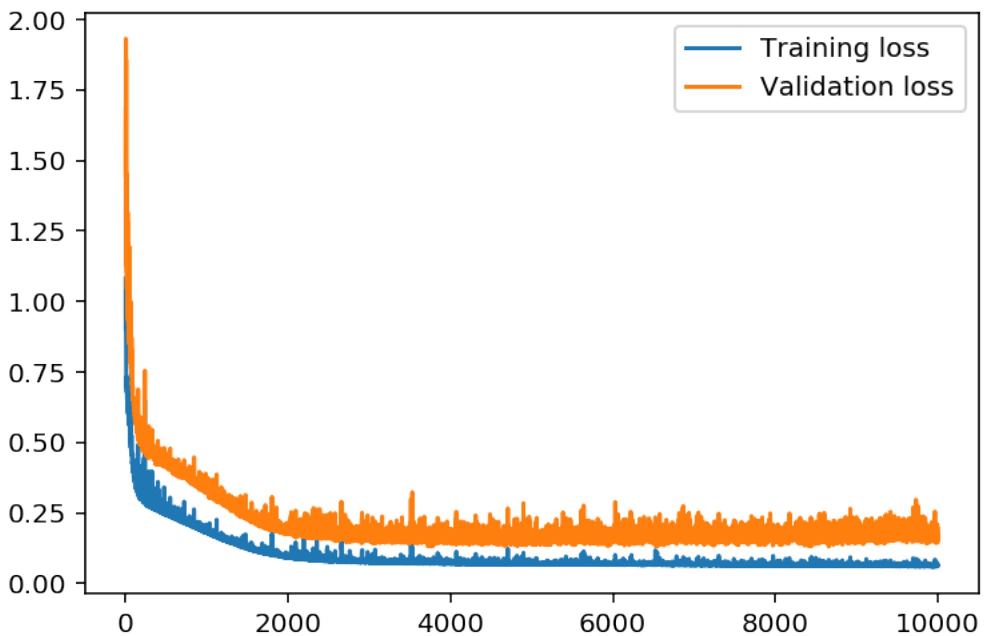
```
In [270]: import sys
          #for hn in range(3, 1000, 1): #So that I can test ranges
```

```
#for lr in range(.1, .9, .01) Checking different learning rates
    ### Set the hyperparameters here ###
iterations = 10000
learning_rate = .7
#learning_rate = lr
#learning_rate = (int(hn)100) #rough method for setting learning rate as a function of
#hidden_nodes = hn
hidden_nodes = 9
output_nodes = 1
print("Learning rate: ",learning_rate)
print(" hidden nodes ",hn)
N_i = train_features.shape[1]
network = NeuralNetwork(N_i, hidden_nodes, output_nodes, learning_rate)

losses = {'train':[], 'validation':[]}
for ii in range(iterations):
    # Go through a random batch of 128 records from the training data set
    batch = np.random.choice(train_features.index, size=128)
    X, y = train_features.ix[batch].values, train_targets.ix[batch]['cnt']

    network.train(X, y) #This is where the actual training part takes place.


    # Printing out the training progress. This feeds the values in then tests accuracy
    train_loss = MSE(network.run(train_features).T, train_targets['cnt'].values)
    val_loss = MSE(network.run(val_features).T, val_targets['cnt'].values)
    sys.stdout.write("\rProgress: {:2.1f}".format(100 * ii/float(iterations))
    sys.stdout.flush()

    losses['train'].append(train_loss)
    losses['validation'].append(val_loss)
''' #Turned on in-line graph drawing
    fig, ax = plt.subplots(figsize=(8,4))

    mean, std = scaled_features['cnt']
    predictions = network.run(test_features).T*std + mean
    ax.plot((test_targets['cnt']*std + mean).values, label='Data')
    ax.plot(predictions[0], label='Prediction')

    ax.set_xlim(right=len(predictions))
    ax.legend()

    dates = pd.to_datetime(rides.ix[test_data.index]['dteday'])
    dates = dates.apply(lambda d: d.strftime('%b %d'))
    ax.set_xticks(np.arange(len(dates))[12::24])
    _ = ax.set_xticklabels(dates[12::24], rotation=45)

'''
```

```
Learning rate:  0.7
 hidden nodes  53
Progress: 100.0% ... Training loss: 0.064 ... Validation loss: 0.148
```

Out[270]: " #Turned on in-line graph drawing\n   fig, ax = plt.subplots(figsize=(8,4))\n   \n

In [271]:
```
plt.plot(losses['train'], label='Training loss')
plt.plot(losses['validation'], label='Validation loss')
plt.legend()
_ = plt.ylim()
```

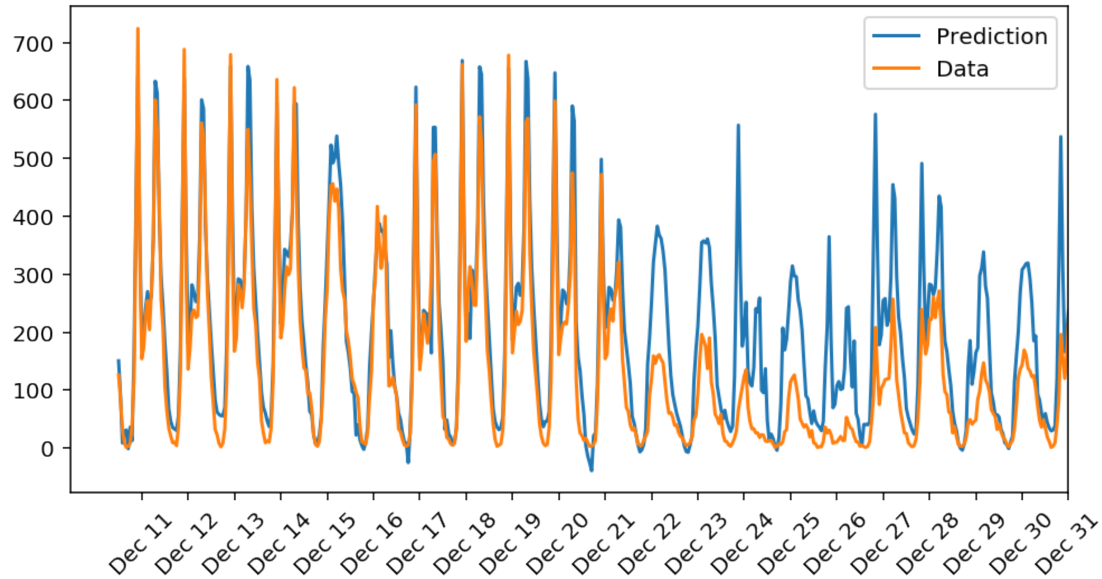


1.6 Check out your predictions

Here, use the test data to view how well your network is modeling the data. If something is completely wrong here, make sure each step in your network is implemented correctly.

In [274]:
```
fig, ax = plt.subplots(figsize=(8,4))

mean, std = scaled_features['cnt']
predictions = network.run(test_features).T*std + mean
ax.plot(predictions[0], label='Prediction')
ax.plot((test_targets['cnt']*std + mean).values, label='Data')


ax.set_xlim(right=len(predictions))
ax.legend()
```

```
dates = pd.to_datetime(rides.ix[test_data.index]['dteday'])
dates = dates.apply(lambda d: d.strftime('%b %d'))
ax.set_xticks(np.arange(len(dates))[12::24])
_ = ax.set_xticklabels(dates[12::24], rotation=45)
```



1.7 OPTIONAL: Thinking about your results(this question will not be evaluated in the rubric).

Answer these questions about your results. How well does the model predict the data? Where does it fail? Why does it fail where it does?

> **Note:** You can edit the text in this cell by double clicking on it. When you want to render the text, press control + enter

Your answer below On a standard day, the model does pretty darn well, nearly matching maxs and mins.

The model fails on holidays. This seems to be making generalizations about what the peaks should be, which varies with actual bike usage data. Maybe it was good weather that day, but the model didn't take day of the week into perspective, etc. Alternatively, maybe some dates have great weather, but the model doesn't favor holidays as much as wind/weather/day of week. Maybe it could be provided some guidelines about human behaviour on religious holidays, to add more negative bias to the system.

```
In [ ]:
```