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).

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

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

```
/Users/raghurajah/anaconda/lib/python3.5/site-packages/matplotlib/fo
nt_manager.py:273: UserWarning: Matplotlib is building the font cach
e using fc-list. This may take a moment.
  warnings.warn('Matplotlib is building the font cache using fc-list
. This may take a moment.')
/Users/raghurajah/anaconda/lib/python3.5/site-packages/matplotlib/fo
nt_manager.py:273: UserWarning: Matplotlib is building the font cach
e using fc-list. This may take a moment.
  warnings.warn('Matplotlib is building the font cache using fc-list
. This may take a moment.')
```

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	weathersit	ter
0	1	2011-01-01	1	0	1	0	0	6	0	1	0.2
1	2	2011-01-01	1	0	1	1	0	6	0	1	0.2
2	3	2011-01-01	1	0	1	2	0	6	0	1	0.2
3	4	2011-01-01	1	0	1	3	0	6	0	1	0.2
4	5	2011-01-01	1	0	1	4	0	6	0	1	0.2

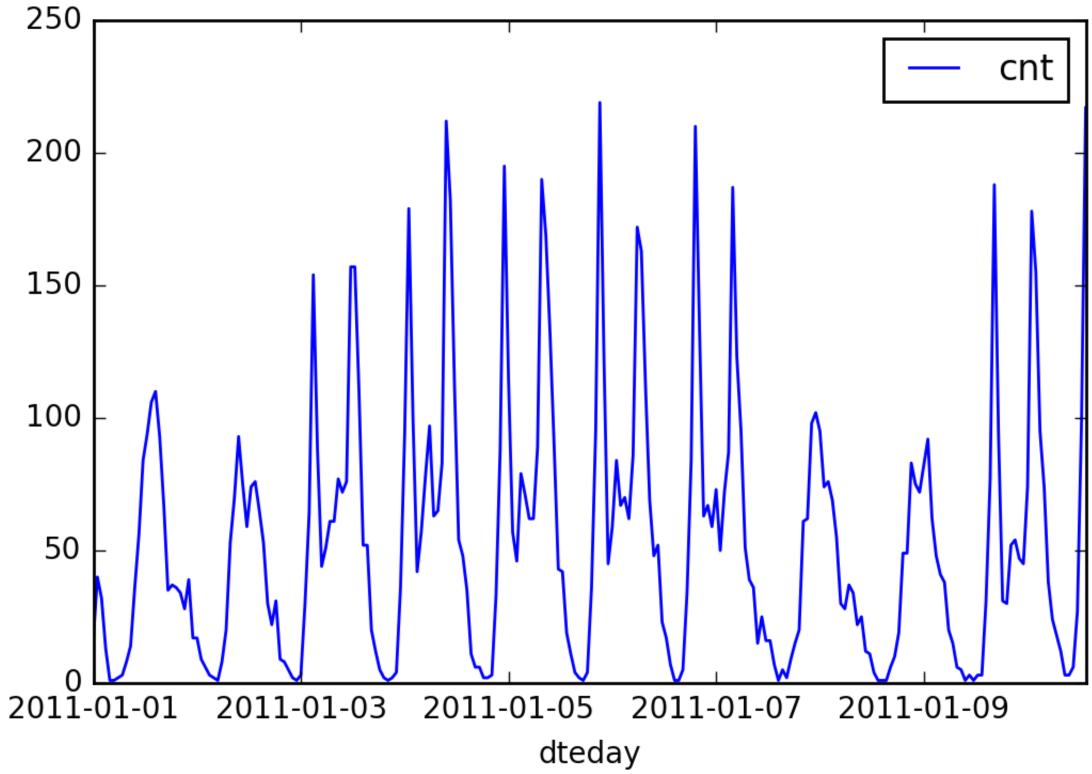
Checking out the data

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 0x11e651f28>



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 [5]:
```
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)

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

Out[5]:

	yr	holiday	temp	hum	windspeed	casual	registered	cnt	season_1	season_2	...
0	0	0	0.24	0.81	0.0	3	13	16	1.0	0.0	...
1	0	0	0.22	0.80	0.0	8	32	40	1.0	0.0	...
2	0	0	0.22	0.80	0.0	5	27	32	1.0	0.0	...
3	0	0	0.24	0.75	0.0	3	10	13	1.0	0.0	...
4	0	0	0.24	0.75	0.0	0	1	1	1.0	0.0	...

5 rows × 59 columns

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
```

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_fiel
ds]
test_features, test_targets = test_data.drop(target_fields, axis=1), t
est_data[target_fields]
```

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 validatio
n set
train_features, train_targets = features[:-60*24], targets[:-60*24]
val_features, val_targets = features[-60*24:], targets[-60*24:]
```

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 [9]:
```
class NeuralNetwork(object):
    def __init__(self, input_nodes, hidden_nodes, output_nodes, learni
ng_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.hidd
en_nodes**-0.5,
                                        (self.hidden_nodes, self.input_
nodes))

        self.weights_hidden_to_output = np.random.normal(0.0, self.out
```

```
put_nodes**-0.5,
                                        (self.output_nodes, self.hidden
_nodes))
        self.lr = learning_rate

        #### Set self.activation_function to your implemented sigmoid
function ####
        #
        # Note: in Python, you can define a function with a lambda exp
ression,
        # as shown below.
        self.activation_function = lambda x : 1 / (1 + np.exp(-x))  #
Replace 0 with your sigmoid calculation.



    def train(self, inputs_list, targets_list):
        # Convert inputs list to 2d array
        inputs = np.array(inputs_list, ndmin=2).T
        targets = np.array(targets_list, ndmin=2).T

        #### Implement the forward pass here ####
        ### Forward pass ###
        # Hidden layer - Replace these values with your calculations.
        hidden_inputs = np.dot(self.weights_input_to_hidden, inputs) #
signals into hidden layer
        hidden_outputs = self.activation_function(hidden_inputs) # sig
nals from hidden layer

        # TODO: Output layer - Replace these values with your calculat
ions.
        final_inputs = np.dot(self.weights_hidden_to_output, hidden_ou
tputs) # signals into final output layer
        final_outputs = final_inputs # signals from final output layer

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

        # Output error - Replace this value with your calculations.
        output_errors = (targets - final_outputs) # Output layer error
is the difference between desired target and actual output.
        delta_w_hidden_output = np.dot(output_errors, hidden_outputs.T
)

        # Backpropagated error - Replace these values with your calcul
ations.
        hidden_errors = np.dot(self.weights_hidden_to_output.T, output
_errors) # errors propagated to the hidden layer
        hidden_grad = hidden_errors * hidden_outputs * (1 - hidden_out
puts)  # hidden layer gradients
        delta_w_input_hidden = np.dot(hidden_grad, inputs.T)

        # Update the weights - Replace these values with your calculat
```

```
ions.
        self.weights_hidden_to_output += (self.lr * delta_w_hidden_out
put) # update hidden-to-output weights with gradient descent step
        self.weights_input_to_hidden += (self.lr * delta_w_input_hidde
n) # update input-to-hidden weights with gradient descent step


    def run(self, inputs_list):
        # Run a forward pass through the network
        inputs = np.array(inputs_list, ndmin=2).T

        #### Implement the forward pass here ####
        # Hidden layer - replace these values with the appropriate cal
culations.
        hidden_inputs = np.dot(self.weights_input_to_hidden, inputs) #
signals into hidden layer
        hidden_outputs = self.activation_function(hidden_inputs) # sig
nals from hidden layer

        # Output layer - Replace these values with the appropriate cal
culations.
        final_inputs = np.dot(self.weights_hidden_to_output, hidden_ou
tputs) # signals into final output layer
        final_outputs = final_inputs # signals from final output layer

        return final_outputs
```

In [10]:

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

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.

Choose the number of epochs

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.

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.

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.

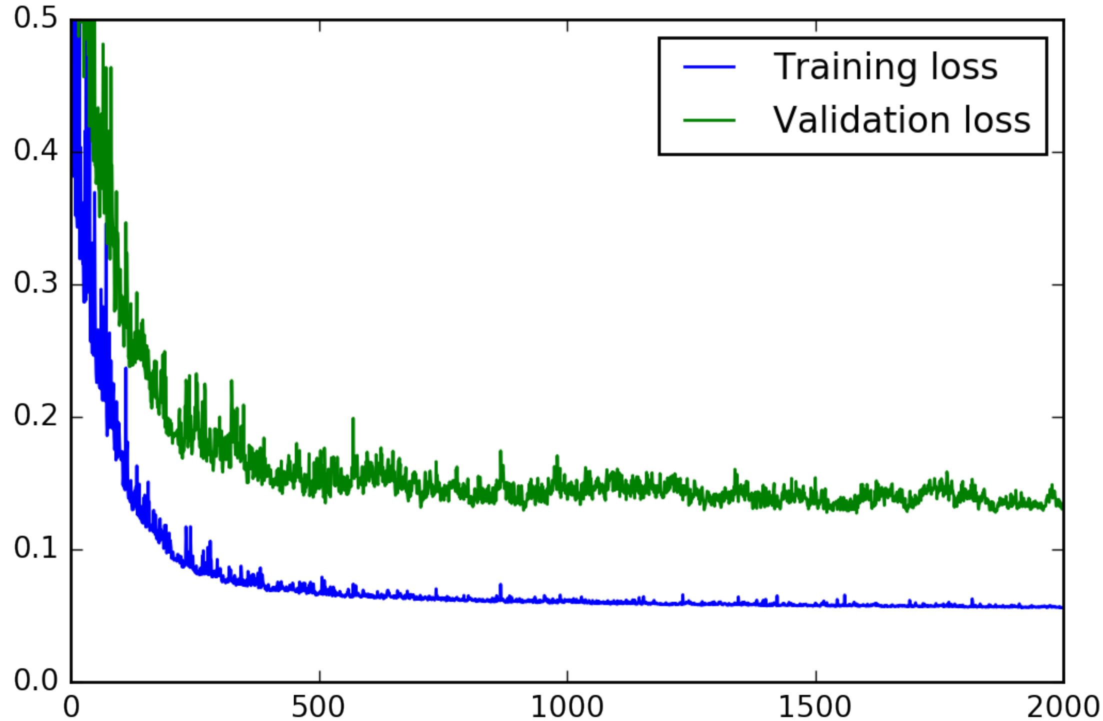
In [11]:

```
import sys

### Set the hyperparameters here ###
epochs = 2000
learning_rate = 0.1
hidden_nodes = 25
output_nodes = 1

N_i = train_features.shape[1]
network = NeuralNetwork(N_i, hidden_nodes, output_nodes, learning_rate
)

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

    # Printing out the training progress
    train_loss = MSE(network.run(train_features), train_targets['cnt']
.values)
    val_loss = MSE(network.run(val_features), val_targets['cnt'].value
s)
    sys.stdout.write("\rProgress: " + str(100 * e/float(epochs))[:4] \
                     + "% ... Training loss: " + str(train_loss)[:5] \
                     + " ... Validation loss: " + str(val_loss)[:5])

    losses['train'].append(train_loss)
    losses['validation'].append(val_loss)

    # Get fancier - reduce learning rate with time
    network.lr = learning_rate / (int(e/100) + 1)
```

```
Progress: 99.9% ... Training loss: 0.055 ... Validation loss: 0.133
```

In [13]:
```
plt.plot(losses['train'], label='Training loss')
plt.plot(losses['validation'], label='Validation loss')
plt.legend()
plt.ylim(ymax=0.5)
```

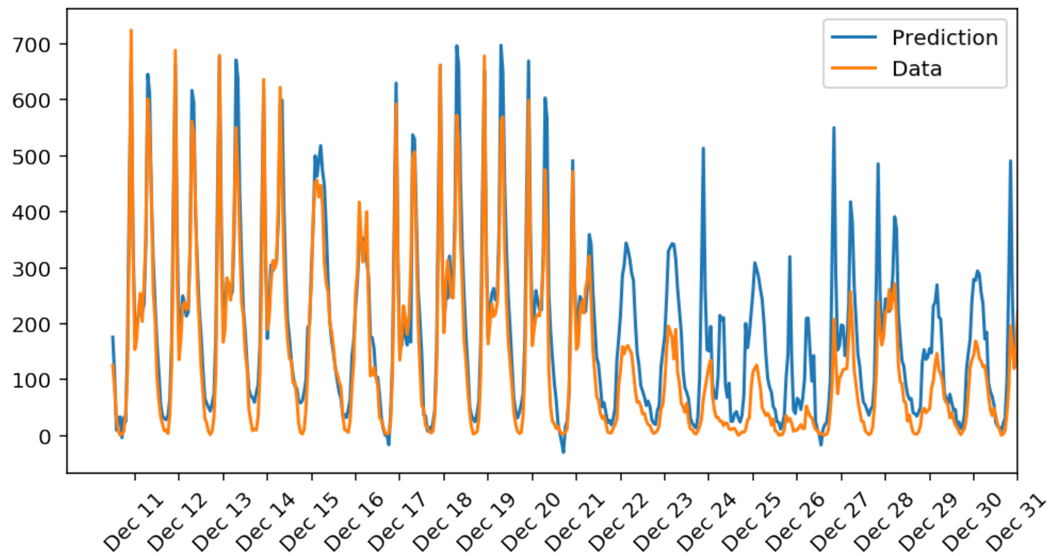
Out[13]: (0.0, 0.5)



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 [33]: fig, ax = plt.subplots(figsize=(8,4))

mean, std = scaled_features['cnt']
predictions = network.run(test_features)*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)
```



OPTIONAL: Thinking about your results.

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

Answer

Unit tests

Run these unit tests to check the correctness of your network implementation. These tests must all be successful to pass the project.

In [14]:

```
import unittest

inputs = [0.5, -0.2, 0.1]
targets = [0.4]
test_w_i_h = np.array([[0.1, 0.4, -0.3],
                       [-0.2, 0.5, 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/hou
r.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, -0.0317293
9]])))
        self.assertTrue(np.allclose(network.weights_input_to_hidden,
                                    np.array([[ 0.10562014,  0.3977519
4, -0.29887597],
                                              [-0.20185996,  0.5007439
8,  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()
        network.weights_hidden_to_output = test_w_h_o.copy()
```

```
        self.assertTrue(np.allclose(network.run(inputs), 0.09998924))

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

```
.....
----------------------------------------------------------------------
--
Ran 5 tests in 0.003s

OK
```

Out[14]:
```
<unittest.runner.TextTestResult run=5 errors=0 failures=0>
```