Feed-Forward-NN

August 21, 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 [ ]: from IPython.core.interactiveshell import InteractiveShell
        InteractiveShell.ast_node_interactivity = "all"
```

```
In [1]: %matplotlib notebook
        %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')
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Out[4]: <matplotlib.axes._subplots.AxesSubplot at 0x7fb0b440a7f0>
```

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 [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  \
        0   0        0  0.24  0.81        0.0       3          13   16         1
        1   0        0  0.22  0.80        0.0       8          32   40         1
        2   0        0  0.22  0.80        0.0       5          27   32         1
        3   0        0  0.24  0.75        0.0       3          10   13         1
        4   0        0  0.24  0.75        0.0       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

   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 rows x 59 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_
```

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 [22]: 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**-0.5,
                                                (self.input_nodes, self.hidden_nodes))

                 self.weights_hidden_to_output = np.random.normal(0.0, self.hidden_nodes**-0.5
                                                (self.hidden_nodes, self.output_nodes))
                 self.lr = learning_rate
```

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

        ### If the lambda code above is not something you're familiar with,
        # You can uncomment out the following three lines and put your
        # implementation there instead.
        #
        #def sigmoid(x):
        #    return 0  # Replace 0 with your sigmoid calculation here
        #self.activation_function = sigmoid


    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

        '''
        #No. of training records in the batch, to do the final update of the derivati
        n_records = features.shape[0]
        #Initialize derivative accumulators to zero
        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 ###
            #TODO: Hidden layer - Replace these values with your calculations.

            #++ MY_NOTE: Attaching to the convention of Phi x X
            #++ where Phi is the weight matrix with dims (no.hidden-units x no.inputs
            # ++ hidden inputs is denoted as z(X) and hiiden_ouptuts as f(z) ++

            #Ensure inputs are 2 dimensional and in the form of column vectors
            init_inputs = np.array(X, ndmin=2).T
            expected_vals = np.array(y, ndmin=2).T

            #++ dim: s_{l+1} x 1 , no.hidden-units x 1 = s_{l+1} x 1
            hidden_inputs =  np.matmul(self.weights_input_to_hidden.T,init_inputs)# s
```

```
            hidden_outputs = self.activation_function(hidden_inputs) # signals from h

            # TODO: Output layer - Replace these values with your calculations.

            #++ dim: no.outputs x hidden_units , hidden_units x 1 = no.outputs x 1
            final_inputs = np.matmul(self.weights_hidden_to_output.T,hidden_outputs) #
            #++ MY_NOTE: The last layer only does regression, f(z) = z ++
            final_outputs = final_inputs.copy() # signals from final output layer

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

            # ++ MY_NOTE: (1) Compute last layer error
            # (2) Propagate the error to hidden layer E(l) = Weights(l) x d(l+1) .* a

            #++ dim: no.outputs x 1
            # ++ MY_NOTE: The multiplication times 1 is just for reference, this shou
            # ++ derivative of the activation function used in the last layer ++
            last_layer_error = expected_vals - final_outputs # Output layer error
            #++ dim: hidden_units x outputs X  outputs x 1 = hidden_units x 1
            hidden_layer_error = (np.matmul(self.weights_hidden_to_output, last_layer_
                                  self.activation_function_d(hidden_inputs))

            # ++ MY_NOTE: Update the derivatives, multiplying by the respective input
            #++ dims: hidden-units x outputs = hidden-units x x outputs
            delta_weights_h_o += np.matmul(hidden_outputs,last_layer_error.T)
            #++ dims: inputs x hidden-units = inputs x hidden-units
            delta_weights_i_h += np.matmul(init_inputs,  hidden_layer_error.T)

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

        # TODO: Update the weights - Replace these values with your calculations.
        self.weights_hidden_to_output += (self.lr * delta_weights_h_o/n_records) # up
        self.weights_input_to_hidden += (self.lr * delta_weights_i_h/n_records) # upd

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

            Arguments
            ---------
            features: 1D array of feature values
        '''
        #Ensure inputs are 2 dimensional
        init_vals = np.array(features, ndmin=2).T
        #### Implement the forward pass here ####
        # TODO: Hidden layer - replace these values with the appropriate calculations
        hidden_inputs =  np.matmul(self.weights_input_to_hidden.T,init_vals)# signals
        hidden_outputs = self.activation_function(hidden_inputs) # signals from hidde
```

```
        # TODO: Output layer - Replace these values with the appropriate calculations
        final_inputs = np.matmul(self.weights_hidden_to_output.T,hidden_outputs) # si
        final_outputs = final_inputs.T # signals from final output layer

        return final_outputs
```

In [23]:
```
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 befor you starting trying to train it. These tests must all be successful to pass the project.

In [24]:
```
import unittest

inputs = np.array([[0.5, -0.2, 0.1]])
targets = np.array([[0.4]])
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)))
```

```python
    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()
        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 0.004s

OK
```

```
Out[24]: <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. Normally a good choice to start at is 0.1; however, if you effectively divide the learning rate by n_records, try starting out with a learning rate of 1. In either case, 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 [33]:
```
import sys

### Set the hyperparameters here ###
iterations = 6000
learning_rate = 0.87
hidden_nodes = 11
output_nodes = 1

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']

    # ++ MY_NOTE: Reduce learning rate after some time to reduce the possibility of
    # jumping around the local-minima
    if ii > 3000:
        network.learning_rate = .06
```

```
        network.train(X, y)

        # Printing out the training progress
        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)) \
                         + "% ... Training loss: " + str(train_loss)[:5] \
                         + " ... Validation loss: " + str(val_loss)[:5])
        sys.stdout.flush()

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

```
Progress: 100.0% ... Training loss: 0.053 ... Validation loss: 0.140
```

1.6 MY OBSERVATIONS - Training Log

- Less number of hidden units require higher number of iterations, as they have to learn how to encode the information "well enough."
- To choose num. of hidden units between num. of inputs and outputs is a good start. Never close to the number of inputs because then no important information will be extracted from the data and it is more likely to overfit.
- At the beginning of the traning, a higher learning rate is better, otherwise convergence time is too long. It is a good idea to reduce the learning rate afterwards when weights are closer to a local minima.

```
In [34]: plt.figure("Train and Validatino Loss")
         plt.plot(losses['train'], label='Training loss')
         plt.plot(losses['validation'], label='Validation loss')
         plt.legend()
         _ = plt.ylim()
```

```
<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>
```

1.7 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 [35]: 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)

    # ++ MY_NOTE: Compute test error to see if it agrees apprximately with the validation
    pred = network.run(test_features).T
    test_loss = MSE(pred[0], test_targets['cnt'].values)
    sys.stdout.write("Test Error: " + str(test_loss))
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Test Error: 0.214983393682
```

1.8 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

- Approximately the model has a 15-20% error on the test data. Which is slightly above the validation error. Based on this and the previous graph we can say that the model predicts well the data, at least on working days. And it also catches the peak hours for bike rental.
- The network generally looses accuracy after the data from DEC 21, where the number of customers in a day decreases significantly in comparison to the previous days. The network overestimates the output from this point forward.
- A hypothesis for the loss of accuracy on these dates could be that from DEC 22 there are no working days. There is a weekend previous to the Christmas holidays. It is reasonable to say that the system has a lot more examples from working days that from holidays, so it cannot generalize as well. And although DEC 22 and DEC 23 are not holidays but part of a weekend, people could have start their holiday activities since then. And unfortunately at this point, the network does not take into account the history of its inputs i.e. use inputs from previous timesteps to make a prediction about the current timestep.

```
In [ ]:
```