# CPE 646 Final Project

Bike Sharing Prediction Through Neural Network

---

By Zhengyang Zhang & Runan Peng

## Background

Bike sharing systems are new generation of traditional bike rentals where whole process from membership, rental and return back has become automatic. Through these systems, user is able to easily rent a bike from a particular position and return back at another position. Currently, there are about over 500 bike-sharing programs around the world which is composed of over 500 thousands bicycles. Today, there exists great interest in these systems due to their important role in traffic, environmental and health issues.

Apart from interesting real world applications of bike sharing systems, the characteristics of data being generated by these systems make them attractive for the research. Opposed to other transport services such as bus or subway, the duration of travel, departure and arrival position is explicitly recorded in these systems. This feature turns bike sharing system into a virtual sensor network that can be used for sensing mobility in the city. Hence, it is expected that most of important events in the city could be detected via monitoring these data.

## Data Set

Bike-sharing rental process is highly correlated to the environmental and seasonal settings. For instance, weather conditions, precipitation, day of the week, season, hour of the day, etc. can affect the rental behaviors. The core data set is related to the two-year historical log corresponding to years 2011 and 2012 from Capital Bikeshare system, Washington D.C., USA which is publicly available in http://capitalbikeshare.com/system-data. We aggregated the data on two hourly and daily basis and then extracted and added the corresponding weather and seasonal information. Weather information are extracted from http://www.freemeteo.com.

## Data File and Characteristics

There are two data file collected:

- hour.csv : bike sharing counts aggregated on hourly basis. Records: 17379 hours
- day.csv - bike sharing counts aggregated on daily basis. Records: 731 days

Both hour.csv and day.csv have the following fields, except hr which is not available in day.csv

- instant: record index

- dteday : date
- season : season (1:springer, 2:summer, 3:fall, 4:winter)
- yr : year (0: 2011, 1:2012)
- mnth : month ( 1 to 12)
- hr : hour (0 to 23)
- holiday : weather day is holiday or not
- weekday : day of the week
- workingday : if day is neither weekend nor holiday is 1, otherwise is 0.

+ weathersit :
    - 1: Clear, Few clouds, Partly cloudy, Partly cloudy
    - 2: Mist + Cloudy, Mist + Broken clouds, Mist + Few clouds, Mist
    - 3: Light Snow, Light Rain + Thunderstorm + Scattered clouds, Light Rain + Scattered clouds
    - 4: Heavy Rain + Ice Pallets + Thunderstorm + Mist, Snow + Fog

- temp : Normalized temperature in Celsius. The values are divided to 41 (max)
- atemp: Normalized feeling temperature in Celsius. The values are divided to 50 (max)
- hum: Normalized humidity. The values are divided to 100 (max)
- windspeed: Normalized wind speed. The values are divided to 67 (max)
- casual: count of casual users
- registered: count of registered users
- cnt: count of total rental bikes including both casual and registered

## Preliminary Analysis

We will only work with hour.csv file in this project, because the hour data is variate throughout hours in different days, it is a more suitable file for neural network predictions.
Here we read the data and obtain some information:

As we can see there are no missing values in the data set, therefore there will be no missing values task we have to do for this dataset. However, if we want to fit a neural network, we still have to do dummy variable conversion and standard scaler for some columns.

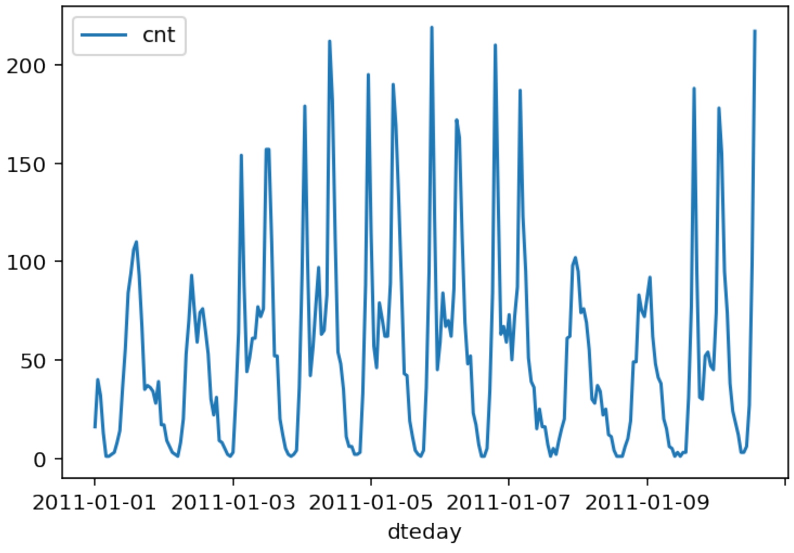
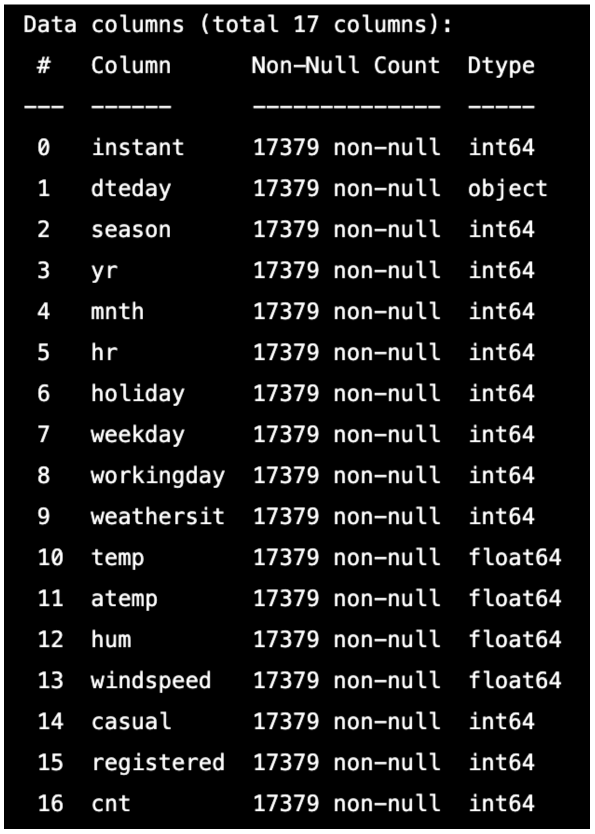


Figure 1



```
Data columns (total 17 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   instant     17379 non-null  int64
 1   dteday      17379 non-null  object
 2   season      17379 non-null  int64
 3   yr          17379 non-null  int64
 4   mnth        17379 non-null  int64
 5   hr          17379 non-null  int64
 6   holiday     17379 non-null  int64
 7   weekday     17379 non-null  int64
 8   workingday  17379 non-null  int64
 9   weathersit  17379 non-null  int64
 10  temp        17379 non-null  float64
 11  atemp       17379 non-null  float64
 12  hum         17379 non-null  float64
 13  windspeed   17379 non-null  float64
 14  casual      17379 non-null  int64
 15  registered  17379 non-null  int64
 16  cnt         17379 non-null  int64
```

data.info

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.

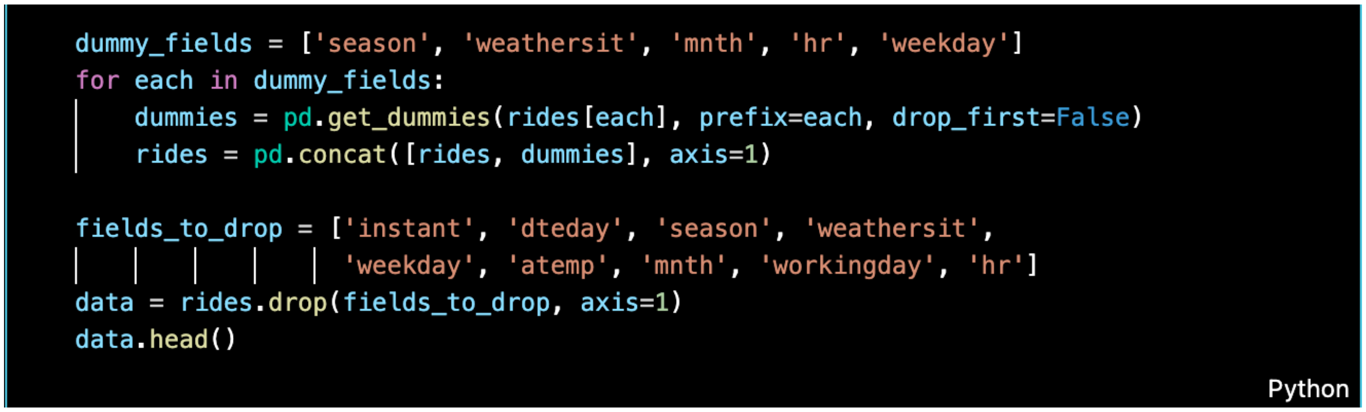
Figure 1 is a plot showing the number of bike riders over the first 10 days or so in the data set. (Some days don't have exactly 24 entries in the data set, so it's not exactly 10 days.) The weekends have lower over all ridership and there are spikes when people are biking to and from work during the week.

Looking at the data above, we also have information about temperature, humidity, and windspeed, all of these likely affecting the number of riders.

## Preprocessing

### DUMMY VARIABLE

Here we believe factors like season, weather, month, hour and day of the week have an effect on the number of rides during that particular hour on that particular day. 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() function.



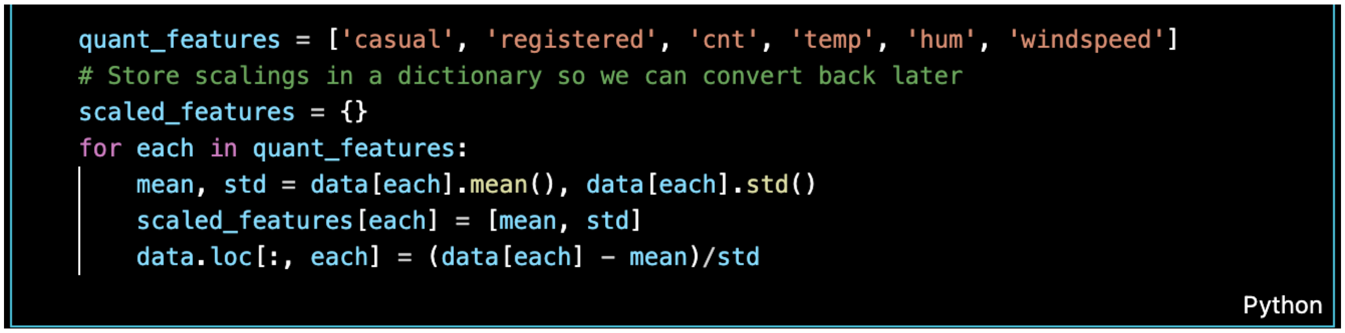
```python
dummy_fields = ['season', 'weathersit', 'mnth', 'hr', 'weekday']
for each in dummy_fields:
    dummies = pd.get_dummies(rides[each], prefix=each, drop_first=False)
    rides = pd.concat([rides, dummies], axis=1)

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

Dummy variable

### SCALING TARGET VARIABLE

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.



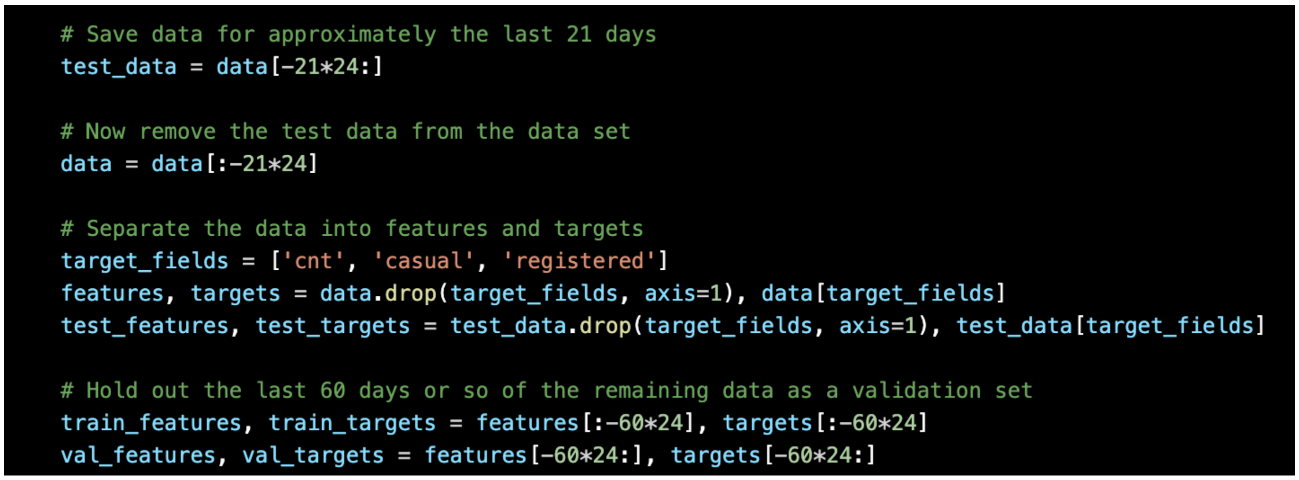
```python
quant_features = ['casual', 'registered', 'cnt', 'temp', 'hum', 'windspeed']
# Store scalings in a dictionary so we can convert back later
scaled_features = {}
for each in quant_features:
    mean, std = data[each].mean(), data[each].std()
    scaled_features[each] = [mean, std]
    data.loc[:,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

Scaling variable

**TRAIN/TEST/VALIDATION SET SPLIT**

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.

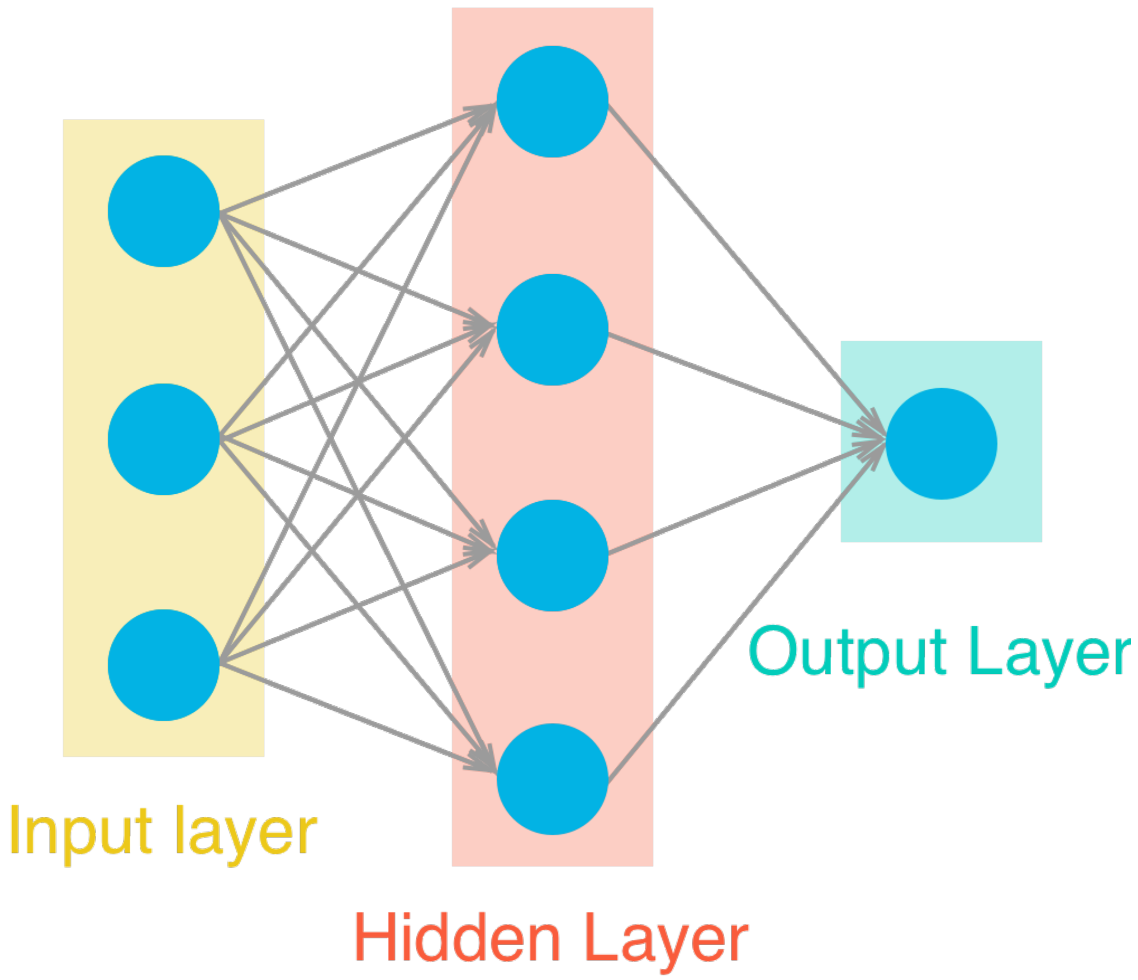
We'll split the data into two sets, one for training and one for validating as the network is being trained. Since this is time series data, we'll train on historical data, then try to predict on future data (the validation set).



```python
# 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_fields]

# 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:]
```

Train Test Validation

## Build Neural Network



Neural Network

To demonstrate our understanding of the Artificial Neural Network, we decided to implement the neural network from scratch rather use packages like PyTorch and Tensorflow. More details please refer to the **Neural_Network.py** file in our submission.

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***.

In this neural network, we will use **MSE** (Mean Square Error) to evaluate the model performance.

## Train Neural Network

Here we're 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.

We'll also be using Stochastic Gradient Descent (SGD) to train the network. The idea is that for each training pass, you select 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.

**NUMBER OF ITERATION**

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, this process can have sharply diminishing returns and can waste computational resources if we use too many iterations. We want to find a number here where the network has a low training loss, and the validation loss is at a minimum. The ideal number of iterations would be a level that stops shortly after the validation loss is no longer decreasing.

**LEARNING RATE**

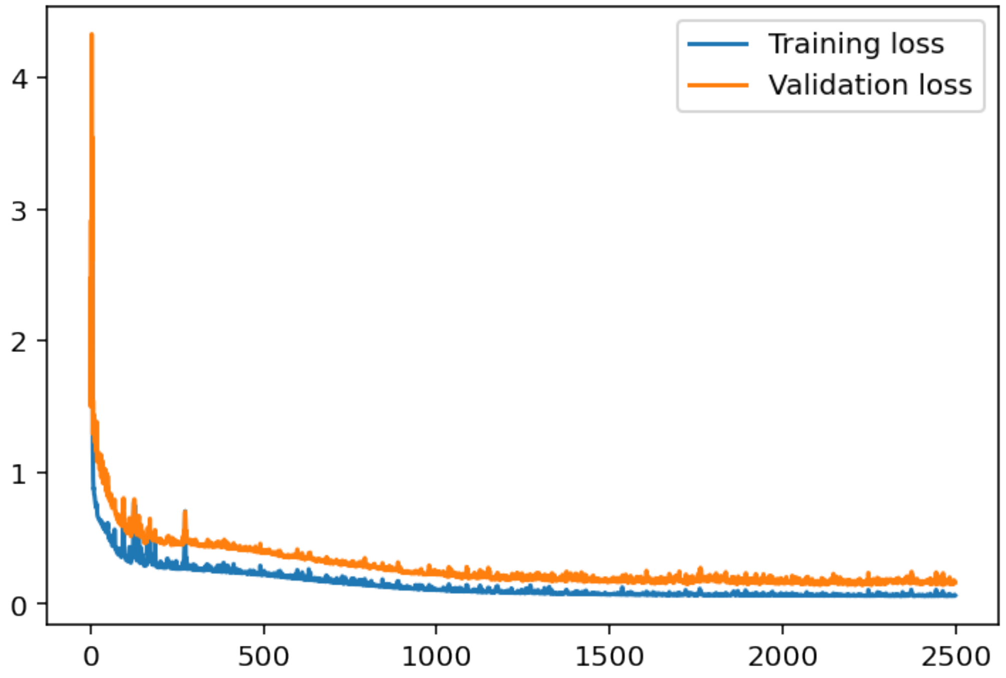
This scales the size of weight updates. If this is too big, the weights tend to explode and the network fails to fit the data. Normally a good choice to start at is 0.1; however, if we effectively divide the learning rate by n_records, try starting out with a learning rate of 1. In either case, if the network has problems fitting the data, we will try reducing the learning rate. However, 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.

**NUMBER OF HIDDEN NODES**

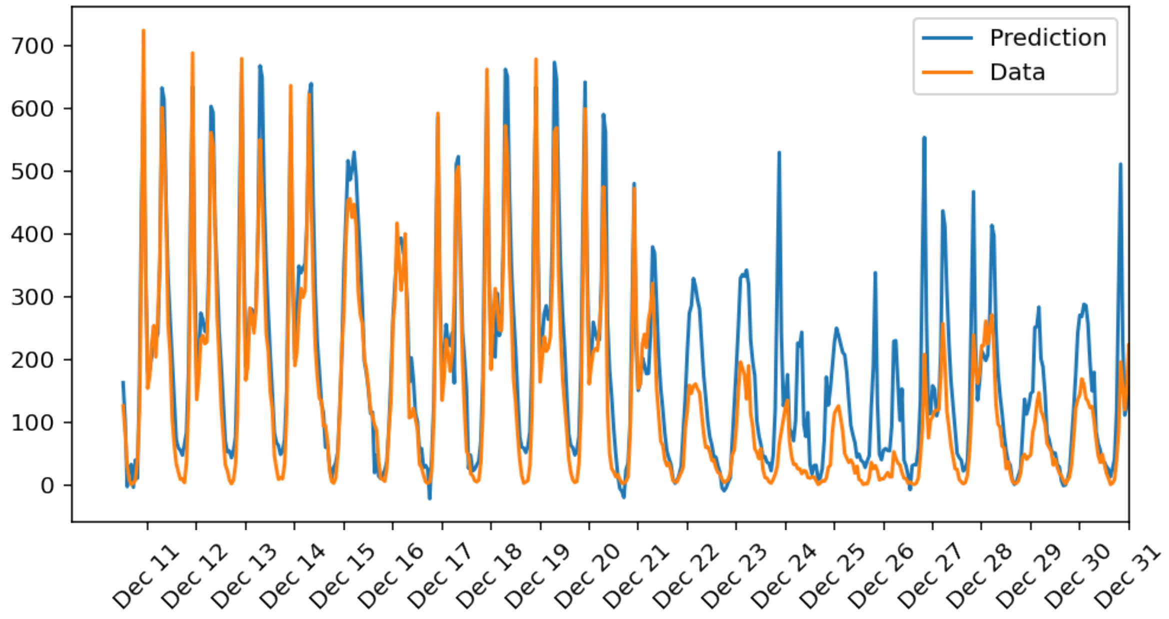
In a model where all the weights are optimized, the more hidden nodes you have, the more accurate the predictions of the model will be.  (A fully optimized model could have weights of zero, after all.) However, the more hidden nodes we have, the harder it will be to optimize the weights of the model, and the more likely it will be that suboptimal weights will lead to overfitting. With overfitting, the model will memorize the training data instead of learning the true pattern, and won't generalize well to unseen data.

**OUR SETUP**
After couple of tries, we have find the parameter we believe to be the best as follows:
**Iteration: 2500, Learning rate: 1, Hidden_Nodes: 10**

## Result & Discussion



Training Loss vs. Validation Loss



Prediction

As we can see from the two graph above. In the first graph, the error graph shows promising result. The training and validation result show similar pattern that decrease over epochs, that means the data is not being overfitted. This is also an indication we have chosen some promising hyperparameter.

However, if we take a look at the prediction graph above, we can see the prediction is quite promising till December 20th ish. As we can see from the yellow line in the graph, the bike sharing usage suddenly shows abnormal usage pattern after that date.

We may have no idea after December 20th for this Bike Sharing data, but it’s definitely worth further investigation in the future. But if we talk from the Neural Network’s perspective, we would say the model performs quite well on the data till its bike sharing usage pattern drastically changed.