Your first neural network

In this project, you'll build your first neural network and use it to predict daily bike rental ridership. You are provided some of the code, but left the implementation of the neural network up to you (for the most part).

In [1]:
```
%matplotlib inline
%load_ext autoreload
%autoreload 2
%config InlineBackend.figure_format = 'retina'

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

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	temp
0	1	2011-01-01	1	0	1	0	0	6	0	1	0.24
1	2	2011-01-01	1	0	1	1	0	6	0	1	0.22
2	3	2011-01-01	1	0	1	2	0	6	0	1	0.22
3	4	2011-01-01	1	0	1	3	0	6	0	1	0.24
4	5	2011-01-01	1	0	1	4	0	6	0	1	0.24

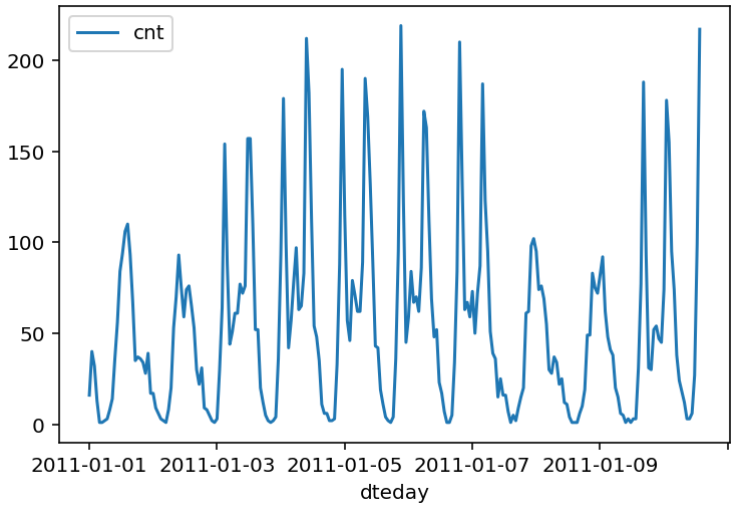
Checking out the data

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]: <AxesSubplot:xlabel='dteday'>



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=True)
    # Nota: he hecho drop_first = True para evitar colinearidad
    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_2	season_3	...	hr_20
0	0	0	0.24	0.81	0.0	3	13	16	0	0	...	0
1	0	0	0.22	0.80	0.0	8	32	40	0	0	...	0
2	0	0	0.22	0.80	0.0	5	27	32	0	0	...	0
3	0	0	0.24	0.75	0.0	3	10	13	0	0	...	0
4	0	0	0.24	0.75	0.0	0	1	1	0	0	...	0

5 rows × 54 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.

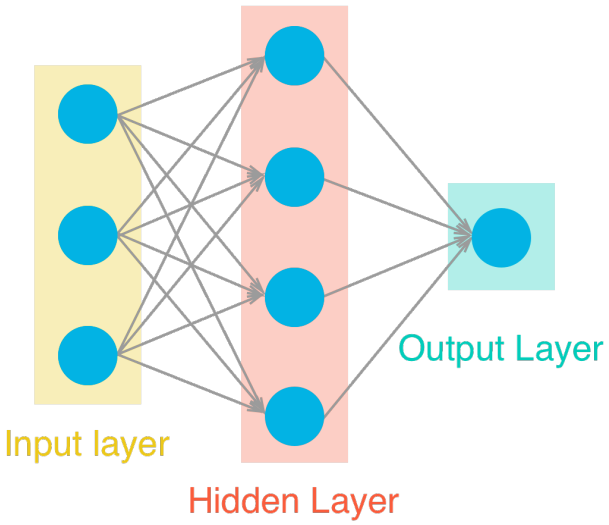
In [7]:
```
# Save data for approximately the last 21 days
test_data = data[-21*24:]

# 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_fields]
test_features, test_targets = test_data.drop(target_fields, axis=1), test_
```

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 se
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. You'll implement both the forward pass and backwards 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]:
```
#############
# In the my_answers.py file, fill out the TODO sections as specified
#############

from my_answers import NeuralNetwork
```

In [10]:
```
def MSE(y, Y):
    return np.mean((y-Y)**2)
```

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.

Nota: para completar el script my_answers.py, en primer lugar hemos recreado los cálculos del gradiente paso a paso en un excel que también entregamos. Esto nos sirvió para entender los cálculos que se tenían que realizar en la bacpropagación:

Modelo que recrea los cálculos para los unit tests

Red neuronal

X	Pesos Capa Hidden (wij)			Input hidden (Ihi)	Output hidden (Ohi)	Pesos Capa Output (ui)	Predicción (y^)	Valor real (y)
0.5	0.1	0.4	-0.3	-0.06	0.485004498	0.3	0.099989239	0.4
-0.2	-0.2	0.5	0.2	-0.18	0.455121108	-0.1		
0.1								

Error € = 0.5*(y - y^)^2	Lambda
0.045003228	0.5

Cálculo pesos hidden_to_output

u1 (updated) = u1 (previous) - 1*grad*lambda	grad= dE/du1	dE/dy^ = -(y - y^)	dy^/du1 = Oh1
0.372753284	-0.145506569	-0.300010761	0.485004498

u2 (updated) = u2 (previous) - 1*grad*lambda	grad = dE/du2	dE/dy^ = -(y - y^)	dy^/du2 = Oh2

In [11]:

```
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.cs

    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

    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 0.013s
```

```
Out[11]: <unittest.runner.TextTestResult run=5 errors=0 failures=0>
```

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 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, this process can have sharply diminishing returns and can waste computational resources if you use too many iterations. You 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.

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.

Choose the 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 you 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.

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. You'll generally find that the best number of hidden nodes to use ends up being between the number of input and output nodes.

In [12]:
```
import sys

####################
### Set the hyperparameters in you myanswers.py file ###
####################
from my_answers import iterations, learning_rate, hidden_nodes, output_node
import time


def train_and_test_network(iterations, learning_rate, hidden_nodes, output_

    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 s
        batch = np.random.choice(train_features.index, size=128)
        X, y = train_features.iloc[batch].values, train_targets.iloc[batch

        network.train(X, y)

        # Printing out the training progress
        train_loss = MSE(network.run(train_features).T, train_targets['cnt
        val_loss = MSE(network.run(val_features).T, val_targets['cnt'].valu
        sys.stdout.write("\rProgress: {:2.1f}".format(100 * ii/float(itera
                         + "% ... 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)

    return losses, network
```

Nota: Para probar diferentes parámetros, hemos decidido hacerlo de una forma un poco más sietmática, introduciendo todos los valores a probar y rentrenando la red con todas las posibles combinaciones. Los resultados se almacenan en la dataframe *results*

In [13]:
```
# Probamos con diferentes combinaciones de parámetros
## TARDA RATO EN EJECUTARSE (en mi caso ~25 mins), para que tarde menos sir
iterations = [100, 1000, 2000]
learning_rate = [0.1, 0.5, 1.0, 3.0]
hidden_nodes = [2, 5, 10]

import itertools
paramateres = [iterations, learning_rate, hidden_nodes]
combinations = list(itertools.product(*paramateres))
results_data = []
for i, param in enumerate(combinations):
    it, lr, hiddn = param[0], param[1], param[2]
    print("\nEntrenando combinación {} de {}".format(int(i), len(combinati
    loss, _ = train_and_test_network(it, lr, hiddn, output_nodes)
    results_data.append([it, lr, hiddn, loss['train'][-1],loss['validation

# Agregamos los resultados
results_columns = ['Iterations', 'Learning rate', 'Hidden nodes', 'Training
results = pd.DataFrame(results_data, columns=results_columns)
```

```
Entrenando combinación 0 de 36
Progress: 99.0% ... Training loss: 0.929 ... Validation loss: 1.336
Entrenando combinación 1 de 36
Progress: 99.0% ... Training loss: 0.885 ... Validation loss: 1.337
Entrenando combinación 2 de 36
Progress: 99.0% ... Training loss: 0.686 ... Validation loss: 1.220
Entrenando combinación 3 de 36
Progress: 99.0% ... Training loss: 0.488 ... Validation loss: 0.791
Entrenando combinación 4 de 36
Progress: 99.0% ... Training loss: 0.516 ... Validation loss: 0.852
Entrenando combinación 5 de 36
Progress: 99.0% ... Training loss: 0.492 ... Validation loss: 0.771
Entrenando combinación 6 de 36
Progress: 99.0% ... Training loss: 0.360 ... Validation loss: 0.552
Entrenando combinación 7 de 36
Progress: 99.0% ... Training loss: 0.358 ... Validation loss: 0.582
Entrenando combinación 8 de 36
Progress: 99.0% ... Training loss: 0.412 ... Validation loss: 0.639
Entrenando combinación 9 de 36
Progress: 99.0% ... Training loss: 0.341 ... Validation loss: 0.543
Entrenando combinación 10 de 36
Progress: 99.0% ... Training loss: 0.326 ... Validation loss: 0.773
Entrenando combinación 11 de 36
Progress: 99.0% ... Training loss: 0.791 ... Validation loss: 1.254
Entrenando combinación 12 de 36
Progress: 99.9% ... Training loss: 0.359 ... Validation loss: 0.577
Entrenando combinación 13 de 36
Progress: 99.9% ... Training loss: 0.333 ... Validation loss: 0.518
Entrenando combinación 14 de 36
Progress: 99.9% ... Training loss: 0.330 ... Validation loss: 0.515
Entrenando combinación 15 de 36
Progress: 99.9% ... Training loss: 0.275 ... Validation loss: 0.547
Entrenando combinación 16 de 36
Progress: 99.9% ... Training loss: 0.273 ... Validation loss: 0.494
Entrenando combinación 17 de 36
Progress: 99.9% ... Training loss: 0.264 ... Validation loss: 0.483
Entrenando combinación 18 de 36
Progress: 99.9% ... Training loss: 0.267 ... Validation loss: 0.437
Entrenando combinación 19 de 36
Progress: 99.9% ... Training loss: 0.223 ... Validation loss: 0.458
Entrenando combinación 20 de 36
Progress: 99.9% ... Training loss: 0.231 ... Validation loss: 0.451
Entrenando combinación 21 de 36
```

```
Progress: 99.9% ... Training loss: 0.283 ... Validation loss: 0.505
Entrenando combinación 22 de 36
Progress: 99.9% ... Training loss: 0.303 ... Validation loss: 0.312
Entrenando combinación 23 de 36
Progress: 1.1% ... Training loss: 12092 ... Validation loss: 64034
/home/llbernat/PycharmProjects/MUSI-AP3/MUSI-AP/Project1-bikesharing/my_ans
wers.py:23: RuntimeWarning: overflow encountered in exp
  self.activation_function = lambda x : 1.0 / (1.0 + np.exp(-1.0*x))  # Rep
lace 0 with your sigmoid calculation.
Progress: 29.6% ... Training loss: inf ... Validation loss: inf6366
/home/llbernat/anaconda3/lib/python3.8/site-packages/numpy/core/_methods.p
y:160: RuntimeWarning: overflow encountered in reduce
  ret = umr_sum(arr, axis, dtype, out, keepdims)
<ipython-input-10-2ce1de1832f0>:2: RuntimeWarning: overflow encountered in
square
  return np.mean((y-Y)**2)
Progress: 61.1% ... Training loss: nan ... Validation loss: nan
/home/llbernat/PycharmProjects/MUSI-AP3/MUSI-AP/Project1-bikesharing/my_ans
wers.py:109: RuntimeWarning: overflow encountered in add
  delta_weights_h_o += np.expand_dims(output_error_term*-1.0, axis=1)
Progress: 99.9% ... Training loss: nan ... Validation loss: nan
Entrenando combinación 24 de 36
Progress: 100.0% ... Training loss: 0.306 ... Validation loss: 0.502
Entrenando combinación 25 de 36
Progress: 100.0% ... Training loss: 0.303 ... Validation loss: 0.497
Entrenando combinación 26 de 36
Progress: 100.0% ... Training loss: 0.312 ... Validation loss: 0.488
Entrenando combinación 27 de 36
Progress: 100.0% ... Training loss: 0.240 ... Validation loss: 0.420
Entrenando combinación 28 de 36
Progress: 100.0% ... Training loss: 0.217 ... Validation loss: 0.391
Entrenando combinación 29 de 36
Progress: 100.0% ... Training loss: 0.235 ... Validation loss: 0.461
Entrenando combinación 30 de 36
Progress: 100.0% ... Training loss: 0.223 ... Validation loss: 0.337
Entrenando combinación 31 de 36
Progress: 100.0% ... Training loss: 0.147 ... Validation loss: 0.287
Entrenando combinación 32 de 36
Progress: 100.0% ... Training loss: 0.181 ... Validation loss: 0.257
Entrenando combinación 33 de 36
Progress: 100.0% ... Training loss: 0.255 ... Validation loss: 0.344
Entrenando combinación 34 de 36
Progress: 100.0% ... Training loss: 0.243 ... Validation loss: 0.284
Entrenando combinación 35 de 36
Progress: 100.0% ... Training loss: 0.304 ... Validation loss: 0.514
```

In [14]:
```
results.sort_values(by=['Validation loss'],axis=0)
```

Out[14]:

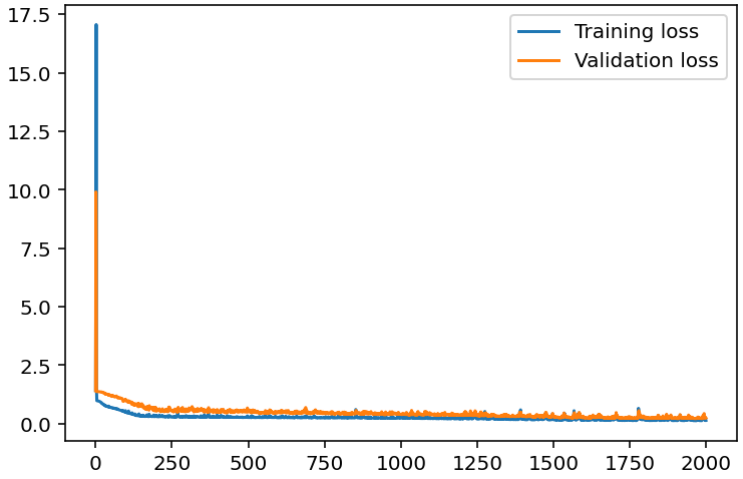
	Iterations	Learning rate	Hidden nodes	Training loss	Validation loss
32	2000	1.0	10	0.181685	0.257015
34	2000	3.0	5	0.243874	0.284570
31	2000	1.0	5	0.147856	0.287883
22	1000	3.0	5	0.303417	0.312352
30	2000	1.0	2	0.223232	0.337299
33	2000	3.0	2	0.255184	0.344569
28	2000	0.5	5	0.217276	0.391918
27	2000	0.5	2	0.240758	0.420648

	Iterations	Learning rate	Hidden nodes	Training loss	Validation loss
18	1000	1.0	2	0.267113	0.437300
20	1000	1.0	10	0.231040	0.451095
19	1000	1.0	5	0.223444	0.458289
29	2000	0.5	10	0.235787	0.461423
17	1000	0.5	10	0.264721	0.483031
26	2000	0.1	10	0.312745	0.488319
16	1000	0.5	5	0.273406	0.494263
25	2000	0.1	5	0.303906	0.497532
24	2000	0.1	2	0.306943	0.502440
21	1000	3.0	2	0.283330	0.505712
35	2000	3.0	10	0.304107	0.514536
14	1000	0.1	10	0.330748	0.515432
13	1000	0.1	5	0.333293	0.518149
9	100	3.0	2	0.341734	0.543755
15	1000	0.5	2	0.275960	0.547800
6	100	1.0	2	0.360808	0.552476
12	1000	0.1	2	0.359986	0.577678
7	100	1.0	5	0.358352	0.582868
8	100	1.0	10	0.412362	0.639792
5	100	0.5	10	0.492500	0.771710
10	100	3.0	5	0.326792	0.773799
3	100	0.5	2	0.488453	0.791836
4	100	0.5	5	0.516548	0.852093
2	100	0.1	10	0.686232	1.220109
11	100	3.0	10	0.791959	1.254436
0	100	0.1	2	0.929022	1.336176
1	100	0.1	5	0.885594	1.337466

In [15]:
```
losses, network = train_and_test_network(2000, 3.0, 5, output_nodes) # Da
```

```
Progress: 100.0% ... Training loss: 0.134 ... Validation loss: 0.236
```

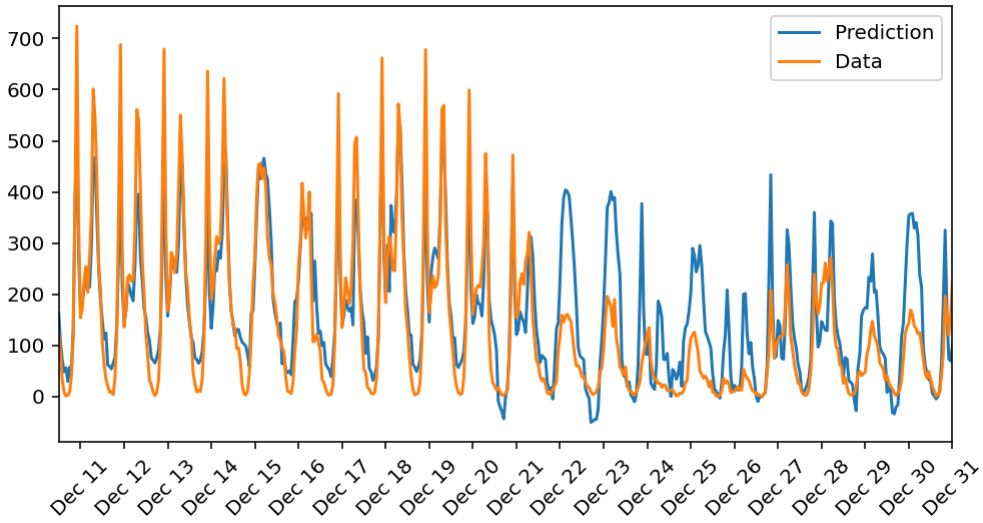
In [16]:
```
plt.plot(losses['train'], label='Training loss')
plt.plot(losses['validation'], label='Validation loss')
plt.legend()
_ = plt.ylim()
```



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 [17]:
```
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(0, right=len(predictions))
ax.legend()

dates = pd.to_datetime(rides.iloc[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.

(1) How well does the model predict the data?

Con los parámetros finalmente elegidos, el modelo predice bastante bien el número de bicis que se alquilan. Aunque sigue habiendo cierto error, y sigue siendo mayor el error en validación que en training (aunque la diferencia no es muy alta), por lo que sigue habiendo cierto overfitting.

(2) Where does it fail?

Identificamos dos puntos concretos en los que el modelo falla:

1. En algunos casos el modelo predice un **número negativo** de bicis alquiladas, lo que obviamente no tiene sentido.

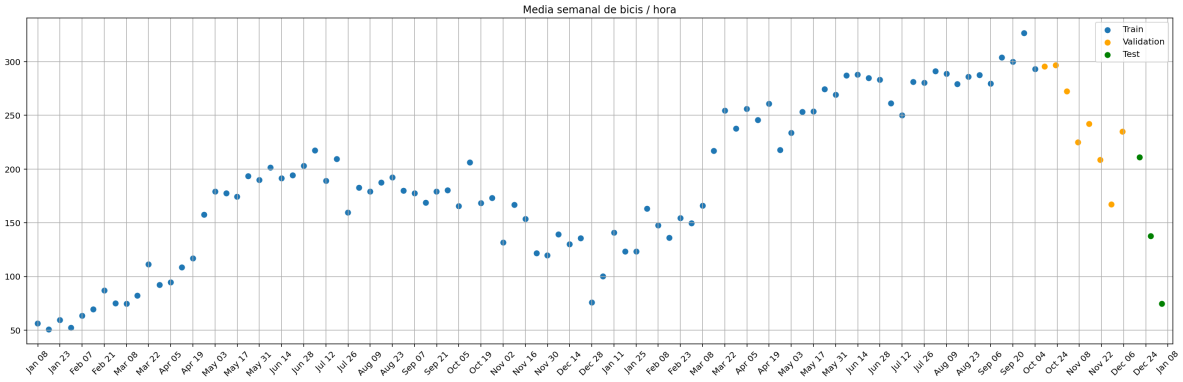
2. Es posible que el modelo no es capaz de capturar la **singularidad de algunos días** festivos y/o el efecto de estacionalidad. En nuestro set de test esos días serían 25 y 26 de diciembre. Son días en los que se observa un número extraordinariamente bajo de alquileres porque probablemente la gente está celebrando con sus familias (el 26 de diciembre en muchos países es *boxing day*, un día festivo familiar). Aunque hay una *feature* que es *holiday*, probablemente no sea suficente para capturar la singularidad de esos días.

3. Como se puede ver en el gráfico de abajo, en los dos años de datos que tenemos hay un **incremento general del volumen de bicis alquilado en el segundo año con respecto al primero**. Las diferencias relativas se mantienen (ej. en ambos casos hay un descenso en los meses de noviembre y diciembre) pero los valores absolutos no. Las variables meteorológicas son similares en ambos años, por lo que este incremento se debe a razones no capturadas en nuestras variables (puede ser simplemente que hay una mayor adopción del uso de la bici por las razones que sean)

En el siguiente apartado explicamos las causas de esots dos puntos y posibles soluciones.

In [18]:

```
fig, ax = plt.subplots(figsize=(15,4))

# Media semanal
roll_hours = 24*7
train_targets_avg = train_targets['cnt'].rolling(window=roll_hours).mean()
val_targets_avg = val_targets['cnt'].rolling(window=roll_hours).mean().ilo
test_targets_avg = test_targets['cnt'].rolling(window=roll_hours).mean().i

mean, std = scaled_features['cnt']
ax.scatter(train_targets_avg.index, (train_targets_avg*std + mean).values,
ax.scatter(val_targets_avg.index - roll_hours, (val_targets_avg*std + mean
ax.scatter(test_targets_avg.index - roll_hours, (test_targets_avg*std + mea

total_len = len(train_targets_avg) + len(val_targets_avg) + len(test_targe
ax.set_xlim(0, right=max(test_targets_avg.index) + 1)
ax.legend()
ax.figure.set_size_inches(24.0,7.0)
plt.gca()
ax.grid(True)

indexes = train_targets_avg.index.union(val_targets_avg.index).union(test_
dates = pd.to_datetime(rides.iloc[indexes]['dteday'])
dates = dates.apply(lambda d: d.strftime('%b %d'))
dates = dates.to_list()
dates.append('Jan 08')
_ = ax.set_xticks(list(np.arange(max(test_targets_avg.index))[7*24::2*7*24
_ = ax.set_xticklabels(dates[0::2], rotation=45)
_ = ax.set_title('Media semanal de bicis / hora')
```



(3) Why does it fail where it does?

1. El modelo produce números negativos porque el *output* de la red no está limitado inferiormente, de modo que el modelo con menor error puede ser uno que en algunos casos produzca valores negativos. Una posible solución sería usar una **función de ativación** en la última neurona que impidiera que las predicciónes, una vez desescaladas, tuvieran un valor negativo. Se podría usar una tangente hiperbólica, cuya imagen está acotada en [-1.0, 1.0], que tras desescalar los datos, queda en [8.0, 370]. Si realmente se quiere acotar en 0, la función de activación a utilizar podría ser tanh(x) · (1.0445), ya que *(0.0 - mean) / std = -1.0445*. Esta función de activación "personalizada" se puede usar sin problema ya que es derivable.

   2 y 3. En el caso del punto 2 y 3, el fallo se debe a que, al cambiar número de bicis por factores no capturados en las variables, el modelo no "tiene manera" de ajustar mejor. Sin embargo, sí que podría mitigarse este efecto si se hubiera hecho el **split en *train,***

***validation, test* de una forma más aleatorizada**, donde se mezclaran datos de diferentes días, meses y años en los tres sets.

En el curso de Deep Learning A-Z de Juan Gabriel Gomila se utiliza una red Neuronal Recurrente para predecir el resultado de un *stock*. Esto nos hace pensar que se podría mejorar la predicción, aplicando una red de este tipo.