

**School of Information Technology & Engineering**

**Course Project Report**

**PYTHON PROGRAMMING (SWE530)**

**Title: Bike Sharing Analysis**

**Faculty: Prof Selva Rani B**

**Slot: C1**

Team Members

-------------------------

D Kalyan Kumar (13MSE0270)

K Vamsi Krishna (13MSE0375)

**1. Problem Statement**

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. The problem in the bike sharing dataset is to predict the bike-usage by using the following attributes 'season’, ‘weather sit ', 'month', 'hour' and 'weekday' by using the neural networks algorithm. The neural network algorithm for this scenario is multi-layer perceptron which uses an activation function. We will measure the difference between the predicted value (algorithm) and the real data.

**2. Introduction**

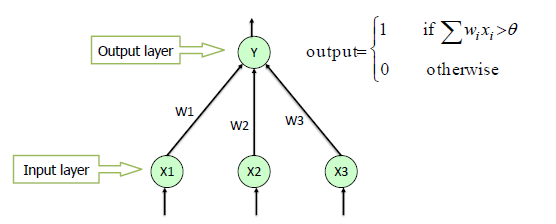
**Artificial neural networks** (**ANNs**) are a computational model used in computer science and other research disciplines, which is based on a large collection of simple neural units called artificial neurons, loosely analogous to the observed behaviour of a biological brain's axons. Each neural unit is connected with many others, and links can enhance or inhibit the activation state of adjoining neural units.  Each individual neural unit computes using summation function. There may be a threshold function or limiting function on each connection and on the unit itself, such that the signal must surpass the limit before propagating to other neurons. These systems are self-learning and trained, rather than explicitly programmed, and excel in areas where the solution or feature detection is difficult to express in a traditional computer program.

Neural networks typically consist of multiple layers or a cube design, and the signal path traverses from the first (input), to the last (output) layer of neural units. Back propagation is the use of forward stimulation to reset weights on the "front" neural units and this is sometimes done in combination with training where the correct result is known. More modern networks are a bit freer flowing in terms of stimulation and inhibition with connections interacting in a much more chaotic and complex fashion.  Dynamic neural networks are the most advanced, in that they dynamically can, based on rules, form new connections and even new neural units while disabling others.

The goal of the neural network is to solve problems in the same way that the human brain would, although several neural networks are more abstract. Modern neural network projects typically work with a few thousand to a few million neural units and millions of connections, which is still several orders of magnitude less complex than the human brain and closer to the computing power of a worm.

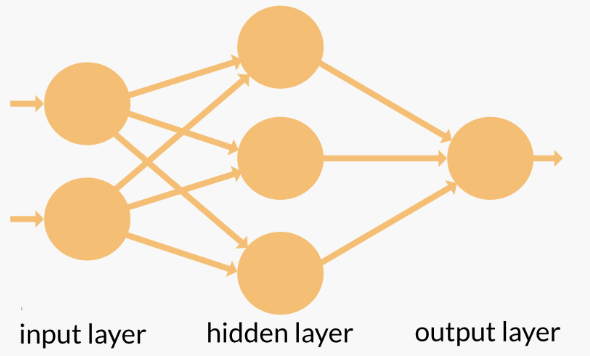
**3. Perceptron**

 Perceptron is a feed-forward network based on a threshold transfer function. SLP is the simplest type of artificial neural networks and can only classify linearly separable cases with a binary target (1, 0).

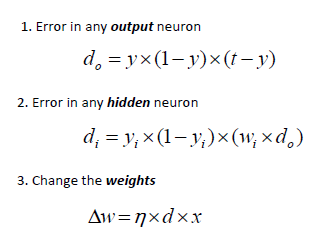


**Multi-Level Perceptron**

A multi-layer perceptron (**MLP**) has the same structure of a single layer perceptron with one or more hidden layers. The backpropagation algorithm consists of two phases: the forward phase where the activations are propagated from the input to the output layer, and the backward phase, where the error between the observed actual and the requested nominal value in the output layer is propagated backwards in order to modify the weights and bias values.



|  |
| --- |
| **Forward propagation:** |
| Propagate inputs by adding all the weighted inputs and then computing outputs using sigmoid threshold. |
| **Backward propagation:** |
| Propagates the errors backward by apportioning them to each unit according to the amount of this error  the unit is responsible for. |



**Perceptron Learning Rule**

1. Given input x = ( I1, I2, .., In). Perceptron produces output y. We are told correct output **O**.
2. For all i:   
       wi := wi + C (**O**-y) Ii
3. t := t - C (**O**-y)

**4. Modules**

**1. Scaling target variables**

To make training the network, 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.

**2. Splitting the data into training, testing, and validation sets**

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

**3.  Build the network**

Built out the structure and the backwards pass. Then implement the forward pass through the network. Set the hyper parameters: 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)=xf(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.

Tasks:

1. Implement the sigmoid function to use as the activation function. Set 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.

**4. Training the network**

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.

A 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. 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.

**5. 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. Choose enough epochs to train the network well but not too many or you'll be overfitting.

**6. 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.

**7. 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. 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.

**8. Check out your predictions**

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.

**5. Code**

**Import Packages**

%matplotlib inline

import numpy as np

import pandas as pd

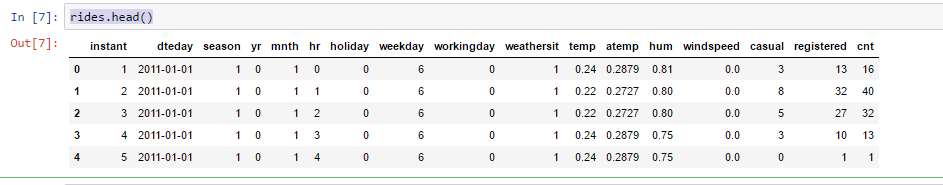
import matplotlib.pyplot as plt

**Data set path and import data**

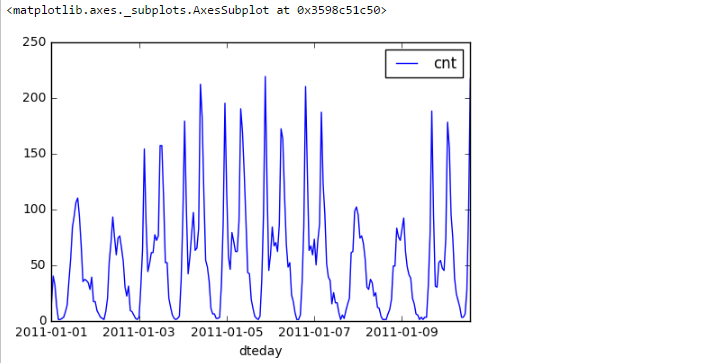
data\_path = 'Bike-Sharing-Dataset/hour.csv'

rides = pd.read\_csv(data\_path)

rides.head()



**Plotting Graph**



**Scaling Target Variables**

dummy\_fields = ['season', 'weathersit', 'mnth', 'hr', 'weekday']

for each in dummy\_fields:

dummies = pd.get\_dummies(rides[each], prefix=each, drop\_first=False)

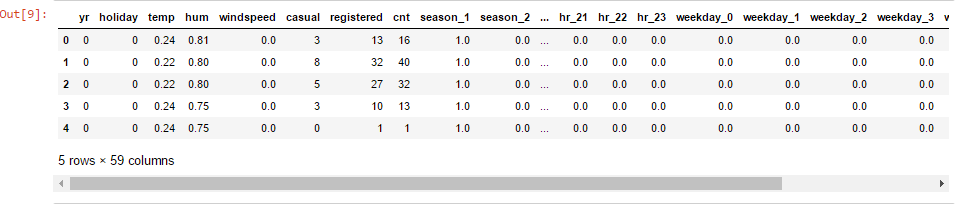
rides = pd.concat([rides, dummies], axis=1)

fields\_to\_drop = ['instant', 'dteday', 'season', 'weathersit',

'weekday', 'atemp', 'mnth', 'workingday', 'hr']

data = rides.drop(fields\_to\_drop, axis=1)

data.head()



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

#split last 21 days of data

test\_data = data[-21\*24:]

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]

#last 60 days of data as validation set

train\_features, train\_targets = features[:-60\*24], targets[:-60\*24]

val\_features, val\_targets = features[-60\*24:], targets[-60\*24:]

**Build the network**

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.hidden\_nodes\*\*-0.5,

(self.hidden\_nodes, self.input\_nodes))

self.weights\_hidden\_to\_output = np.random.normal(0.0, self.output\_nodes\*\*-0.5,

(self.output\_nodes, self.hidden\_nodes))

self.lr = learning\_rate

# Activation function is the sigmoid function

self.activation\_function = lambda x: 1 / ( 1 + np.exp(-x))

def train(self, inputs\_list, targets\_list):

inputs = np.array(inputs\_list, ndmin=2).T

targets = np.array(targets\_list, ndmin=2).T

# Hidden layer

hidden\_inputs = np.dot(self.weights\_input\_to\_hidden, inputs)

hidden\_outputs = self.activation\_function(hidden\_inputs)

# Output layer

final\_inputs = np.dot(self.weights\_hidden\_to\_output, hidden\_outputs)

final\_outputs = final\_inputs

### Backward pass ###

# Output error

output\_errors = targets - final\_outputs

# Backpropagated error

hidden\_errors = np.dot(self.weights\_hidden\_to\_output.T, output\_errors)

hidden\_grad = hidden\_outputs \* (1 - hidden\_outputs)

# output\_grad = learning\_rate \* hidden\_errors \* inputs

# Update the weights

output\_grad= 1

self.weights\_hidden\_to\_output += self.lr \* np.dot(output\_errors, hidden\_outputs.T)

# update hidden-to-output weights with gradient descent step

self.weights\_input\_to\_hidden += self.lr \* np.dot( hidden\_grad \* hidden\_errors , inputs.T)

# 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

#forward pass

# Hidden layer

hidden\_inputs = np.dot(self.weights\_input\_to\_hidden, inputs)

hidden\_outputs = self.activation\_function(hidden\_inputs)

# Output layer

final\_inputs = np.dot(self.weights\_hidden\_to\_output, hidden\_outputs)

final\_outputs = final\_inputs

return final\_outputs

def MSE(y, Y):

return np.mean((y-Y)\*\*2)

**Training the data**

1)

import sys

# Set hyperparameters here

epochs = 2000

learning\_rate = 0.075

hidden\_nodes = 30

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):

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)

train\_loss = MSE(network.run(train\_features), train\_targets['cnt'].values)

val\_loss = MSE(network.run(val\_features), val\_targets['cnt'].values)

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)



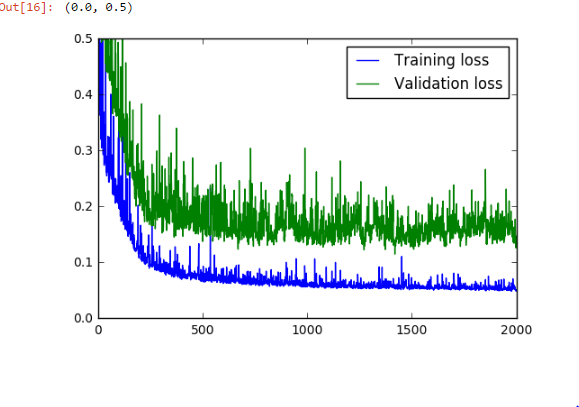
2)

plt.plot(losses['train'], label='Training loss')

plt.plot(losses['validation'], label='Validation loss')

plt.legend()

plt.ylim(ymax=0.5)



**Predictions**

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)

