LABORATORY EXPERIENCE #6
(Regression through Neural Networks)

Introduction

This laboratory experience requires to perform **regression** on the dataset of Parkinson disease using Neural Networks. NNs are models that take some data in input and, combining one or more levels of weights, biases and non-linarites through multiplications and sums, reproduce a desired output. Hence, we can use these tools to solve regression, classification or clustering problems.
To implement neural networks we use a tool developed by Google that is called **Tensorflow**. Combined with the numpy library, we are able to develop our assignment using Python coding language. The problem requires to perform the experiment at first with a neural network without hidden layers, and then with a neural network with two hidden layers, using as regressand features 5 and 7.

Data preparation

The input data has to be adjusted in the same way as the first lab, performing normalization and removing some columns corresponding to features that won't be used in the regression problem. The feature on which we want to perform regression has to be isolated in order to create the column vector that we will use as regressand in the training phase.

Algorithms implementation and results

In the case of neural network **without hidden layers**, we only have to define an array of weights, with length 18, as the number of used features, and a bias factor. These parameters are optimized step by step by Tensorflow functions that will perform optimization according to the algorithm and the objective function defined by the user: in our case we decided to use the **Gradient Algorithm** in order to minimize the squared difference between our data rows and the target array, our regressand. The parameters to be set are the learning rate and the number of steps of the optimization algorithm: through a trial and error approach we obtain that performing regression on feature 7 we have very good results with a learning rate of 10^{-5} and 1000 steps.

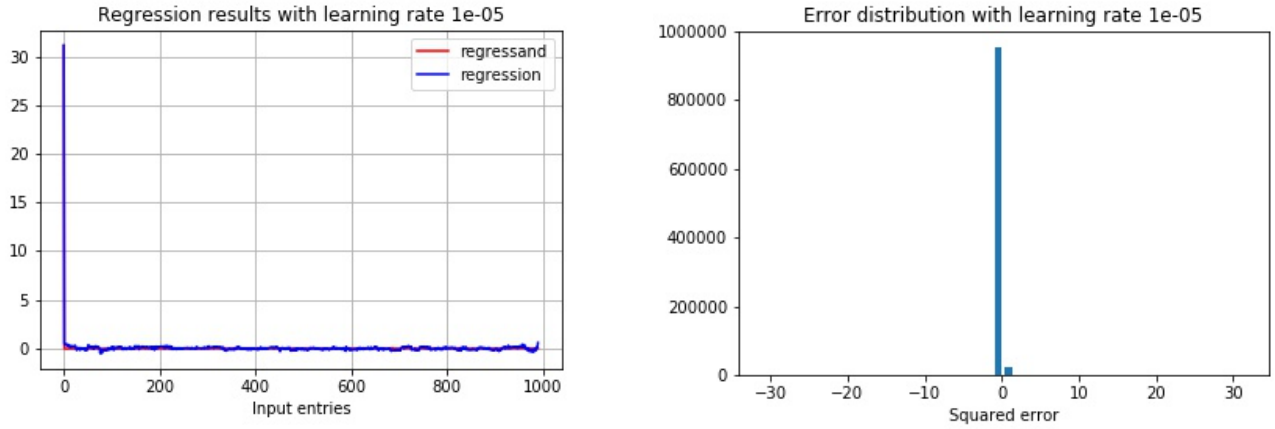

Figure 1: a-Regression results; b-Histogram of the error distribution (feature 7)

Repeating the experiment with feature 5 as regressand, as in the previous laboratory experiences, we experience a larger error in the parameter estimation. Also changing the parameters of the optimization algorithm we are not able to reach the same precision as the feature 7.

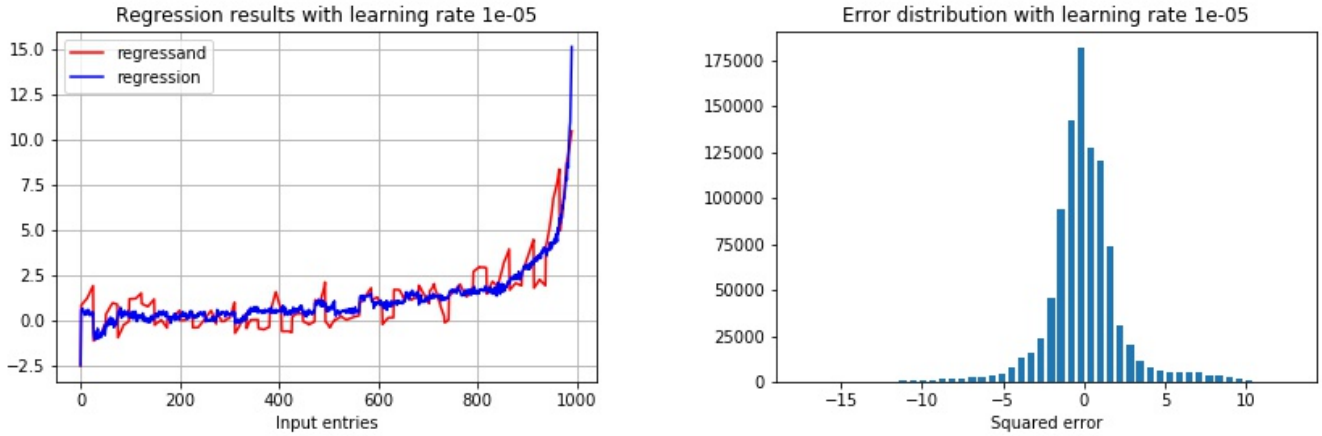

Figure 2: a-Regression results; b-Histogram of error distribution (feature 5)

In the example with **two hidden layers**, we have to define 3 sets of weights and biases and we have to add a non-linearity of type *'tanh'* after each activation.
The results are slightly better compared to the ones obtained with the structure without hidden layers. In particular we have optimum results regressing on feature 7 with learning rate equal to 10^{-4}. Also in this case, the result of the regression on feature 5 is not good.

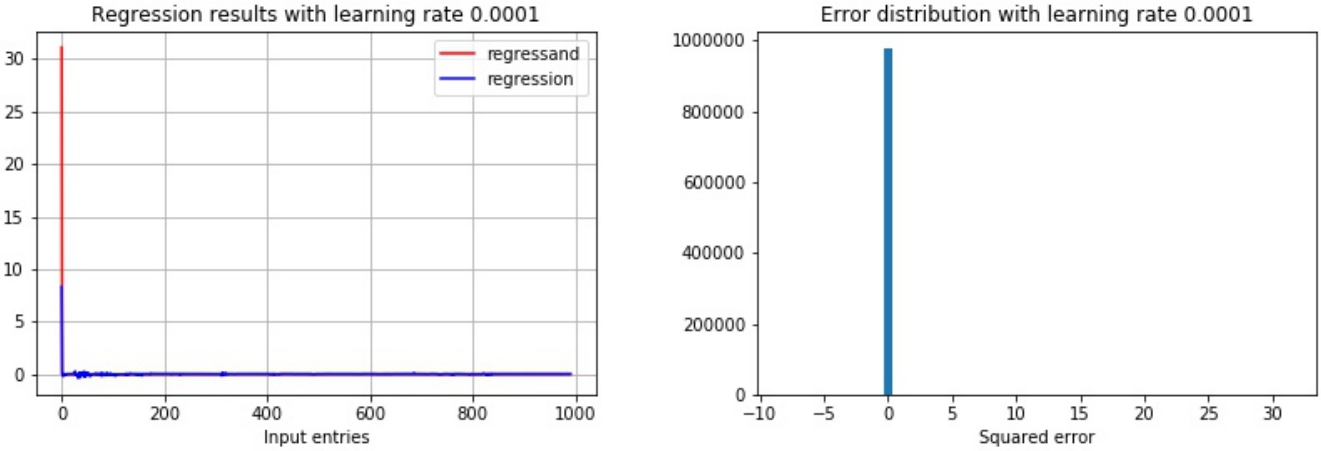

Figure 3: a-Regression results; b-Histogram of the error distribution (feature 7)

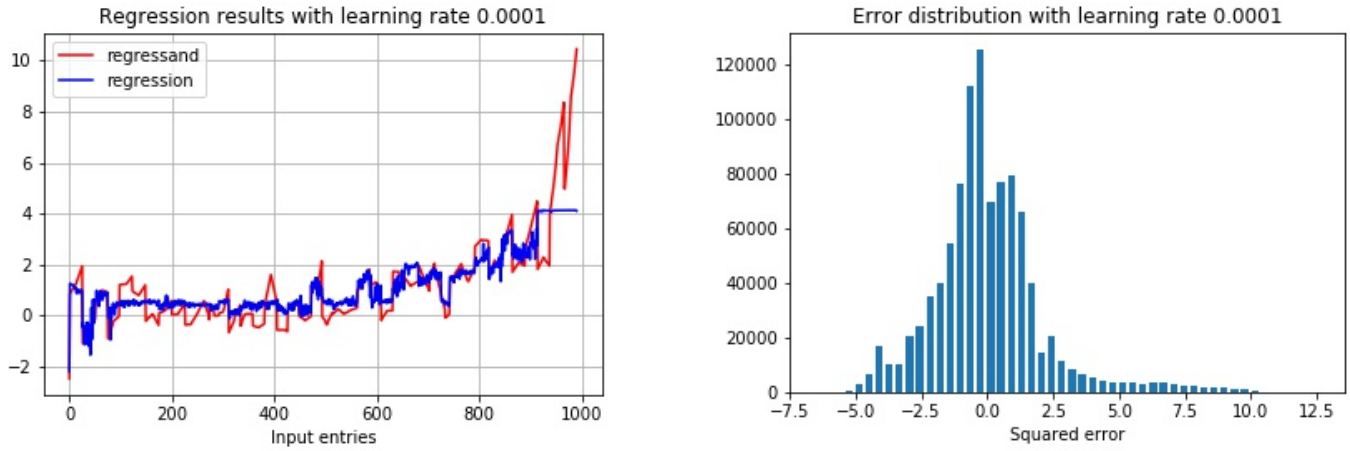

Figure 4: a-Regression results; b-Histogram of the error distribution (feature 7)

LAB_06 – No hidden nodes

```
import scipy.io as scio
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

# importing the input data
mat_file=scio.loadmat('parkinson.mat')
data=mat_file.get('new_matrix')

FEATURE = 5 # the feature we want to use as regressand

(N, F)=np.shape(data)
#normalizing data
x_norm=data
for i in range(N):
   x_norm[i] = (data[i]-np.mean(data,0))/np.std(data,0)

# removing the column that we use as regressand
regressand=data[:,FEATURE-1]

x_norm=np.delete(x_norm, [1-1,4-1,6-1,FEATURE-1], 1) # deleting some unused features
(N, F)=np.shape(x_norm)


# initializing the neural network graph
tf.set_random_seed(1234)
learning_rate = 1e-5

x = tf.placeholder(tf.float64, [N, F])
t = tf.placeholder(tf.float64, [N, 1])

w1 = tf.Variable(tf.random_normal(shape=[F, 1], mean=0.0, stddev=1.0, dtype=tf.float64,
name="weights"))
b1 = tf.Variable(tf.random_normal(shape=[1, 1], mean=0.0, stddev=1.0, dtype=tf.float64,
name="biases"))
y = tf.matmul(x, w1) + b1

#implementation of gradient algorithm
cost = tf.reduce_sum(tf.squared_difference(y, t, name="objective_function"))
optim = tf.train.GradientDescentOptimizer(learning_rate, name="GradientDescent")
optim_op = optim.minimize(cost, var_list=[w1, b1])

init = tf.global_variables_initializer()

#--- run the learning machine
sess = tf.Session()
```

```
sess.run(init)

xval = x_norm.reshape(N,F)
tval = regressand.reshape(N, 1)
for i in range(1000):
   # generate the data
   # train
   input_data = {x: xval, t: tval}
   sess.run(optim_op, feed_dict = input_data)
   if i % 100 == 0:# print the intermediate result
      print(i,cost.eval(feed_dict=input_data,session=sess))


#--- print the final results
print(sess.run(w1), sess.run(b1))
a = sess.run(w1)
yval= y.eval(feed_dict = input_data, session=sess)

plt.plot(regressand,'r', label='regressand')
plt.plot(yval,'b', label='regression')
plt.xlabel('Input entries')
plt.grid(which='major', axis='both')
plt.legend()
plt.title('Regression results with learning rate '+ str(learning_rate))
plt.savefig('yval_vd_ytrue_noHN_7jpg',format='jpg')
plt.show()

hist, bins = np.histogram((regressand-yval), bins=50)
width = 0.7 * (bins[1] - bins[0])
center = (bins[:-1] + bins[1:]) / 2
plt.bar(center, hist, align='center', width=width)
plt.xlabel('Squared error')
plt.title('Error distribution with learning rate '+str(learning_rate))
plt.savefig('squared_error_noHN_7'+ str(learning_rate)+'.jpg',format='jpg')
plt.show()
```

LAB 6 – 2 Hidden Layers

```
import scipy.io as scio
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

# importing the input data
mat_file=scio.loadmat('parkinson.mat')
data=mat_file.get('new_matrix')

FEATURE = 5 # the feature we want to use as regressand

(N, F)=np.shape(data)
#normalizing data
x_norm=data
for i in range(N):
   x_norm[i] = (data[i]-np.mean(data,0))/np.std(data,0)

# removing the column that we use as regressand
regressand=data[:,FEATURE-1]

x_norm=np.delete(x_norm, [1-1,4-1,6-1,FEATURE-1], 1) # deleting some unused features (in
python the indexes start from 0)
(N, F)=np.shape(x_norm)


# initializing the neural network graph
tf.set_random_seed(1234)
learning_rate = 1e-4
n_hidden_nodes_1=F
n_hidden_nodes_2=10

x = tf.placeholder(tf.float64, [N, F])
t = tf.placeholder(tf.float64, [N, 1])

# first layer
w1 = tf.Variable(tf.random_normal(shape=[F, n_hidden_nodes_1], mean=0.0, stddev=1.0,
dtype=tf.float64, name="weights"))
b1 = tf.Variable(tf.random_normal(shape=[1, n_hidden_nodes_1], mean=0.0, stddev=1.0,
dtype=tf.float64, name="biases"))
a1 = tf.matmul(x, w1) + b1
z1 = tf.nn.tanh(a1)

# second layer
w2 = tf.Variable(tf.random_normal(shape=[n_hidden_nodes_1, n_hidden_nodes_2], mean=0.0,
stddev=1.0, dtype=tf.float64, name="weights2"))
```

```
b2 = tf.Variable(tf.random_normal(shape=[1, n_hidden_nodes_2], mean=0.0, stddev=1.0,
dtype=tf.float64, name="biases2"))
a2 = tf.matmul(z1, w2) + b2
z2 = tf.nn.tanh(a2)

# second layer
w3 = tf.Variable(tf.random_normal(shape=[n_hidden_nodes_2, 1], mean=0.0, stddev=1.0,
dtype=tf.float64, name="weights3"))
b3 = tf.Variable(tf.random_normal(shape=[1, 1], mean=0.0, stddev=1.0, dtype=tf.float64,
name="biases3"))
y = tf.matmul(z2, w3) + b3

#implementation of gradient algorithm
cost = tf.reduce_sum(tf.squared_difference(y, t, name="objective_function"))
optim = tf.train.GradientDescentOptimizer(learning_rate, name="GradientDescent")
optim_op = optim.minimize(cost, var_list=[w1,b1,w2,b2,w2,b3])

init = tf.global_variables_initializer()

#--- run the learning machine
sess = tf.Session()
sess.run(init)

xval = x_norm.reshape(N,F)
tval = regressand.reshape(N, 1)
for i in range(1000):
   # generate the data
   # train
   input_data = {x: xval, t: tval}
   sess.run(optim_op, feed_dict = input_data)
   if i % 100 == 0:# print the intermediate result
      print(i,cost.eval(feed_dict=input_data,session=sess))


#--- print the final results
print(sess.run(w1),sess.run(b1))
print(sess.run(w2),sess.run(b2))
print(sess.run(w3),sess.run(b3))

a = sess.run(w1)
yval= y.eval(feed_dict = input_data, session=sess)

print np.shape(yval)



plt.plot(regressand,'r', label='regressand')
plt.plot(yval,'b', label='regression')
```

```
plt.xlabel('Input entries')
plt.grid(which='major', axis='both')
plt.legend()
plt.title('Regression results with learning rate '+str(learning_rate))
plt.savefig('yval_vd_ytrue_2hn_5.jpg',format='jpg')
plt.show()

hist, bins = np.histogram((regressand-yval), bins=50)
width = 0.7 * (bins[1] - bins[0])
center = (bins[:-1] + bins[1:]) / 2
plt.bar(center, hist, align='center', width=width)
plt.xlabel('Squared error')
plt.title('Error distribution with learning rate '+str(learning_rate))
plt.savefig('squared_error_2hn_5'+ str(learning_rate)+'.jpg',format='jpg')
plt.show()
```