Numerical optimization for large scale problems and Stochastic Optimization

Mattia Delleani
s288854

February 11, 2020

Abstract

This report describes a possible approach to Problem 1, concerning *Static Optimization*. In particular, the proposed approach consists in building a Neural Network able to approximate a function, which values are accessible only through simulations. Then an optimization algorithm has been developed to obtain the argument that maximize the function, directly on the Neural Network.

1 Problem 1: Static Optimization

The Neural Network is composed by:

- *Input layer*: 2 input nodes (s1,s2) and 1 bias node
- *Hidden layer*: 2 hidden nodes and 1 bias node
- *Output layer*: 1 output node

As *activation function* has been used the sigmoid function:

$$h_\theta(x) = \frac{1}{1+e^{-x}}$$

The learning rate used is $\mu = 0.01$ and at each step is multiplied by 0.9999. A *tolerance* of 10^{-6} has been used for the SSE error and the number of epochs is setted at 10000.

1.1 Commented code

The following scripts written in Python language have been used to solve the problem.

[1]:
```
import numpy as np
import pandas as pd
import math
import random
import datetime
import itertools
```

[2]:
```
# function for probabilities
def compute_probabilities(s1, s2):


    p12 = 1/60 + (1/200 - 1/60) * math.sqrt(1-((10-s1)/10)**2)
    p23 = 1/30 + (1/100 - 1/30) * math.sqrt(1-((10-s2)/10)**3)
```

```
    return p12, p23
```

[3]:
```
# function for 60days scheduling
def scheduling(p12, p23, s1, s2):

    #for the 1st day

    n12 = 2000
    B12 = np.random.binomial(n12, p12)


    W1 = B12
    B1 = 0

    N1 = 2000 - W1 - B1

    total_gain = 2000 - (s1+s2)
    last_gain = 0

    #all the other
    for i in range(1, 60):

        W0 = W1
        B0 = B1
        n12 = 2000 - W0- B0
        B12 = np.random.binomial(n12, p12)

        n23 = W0
        B23 = np.random.binomial(n23, p23)

        W1 = W0 + B12 - B23
        B1 = B23

        N1 = 2000 - W1 - B1

        daily_gain = (2000 - B1) * 1 - 90 * B1 -(s1+s2)

        total_gain = total_gain +  daily_gain

        if i== 59:
            last_gain = (2000 - B1) * 1 - 90 * B1



    return total_gain, last_gain
```

1.1.1 Simulation

[4]:
```
now = datetime.datetime.now()

s_list = np.arange(2,11, 2)

combinations = list(itertools.product(s_list, s_list))
```

```
result = []

tot = []

# Start the simulation
for s1, s2 in combinations:

    p12, p23 = compute_probabilities(s1, s2)

    result.append(s1)
    result.append(s2)
    tot.append(s1)
    tot.append(s2)

    i_result = []
    tot_g = []
    for i in range(10000):

        tot_gain, last_gain = scheduling(p12, p23, s1, s2)
        i_result.append(last_gain)
        tot_g.append(tot_gain)
    result.append(i_result)
    tot.append(tot_g)

finish = datetime.datetime.now()

delta = finish - now

print(f"Start: {now}\nFinish: {finish}\nDeltaT:{delta}" )
```

```
Start: 2021-02-10 17:06:32.933865
Finish: 2021-02-10 17:07:53.521874
DeltaT:0:01:20.588009
```

1.1.2 Expected value for each pair (s1, s2)

[5]:
```
expected_values = {}
array = []

final = []
for r, (s1, s2) in zip(tot[2::3], combinations):
    expected_values[(s1,s2)] = sum(r)/10000
    array.append(sum(r)/10000)
```

[6]:
```
values = np.array(array)
values
from scipy.stats import zscore
from sklearn import preprocessing

scaler = preprocessing.StandardScaler()
mean_training = values.mean()
```

```
std_training = values.std()

max_training = max(values) # max values for normalization
min_training = min(values) # min values for norm.

# Max-min norm.
normalized = [(x - min_training)/(max_training - min_training) for x in values]
#standardized = [(x - mean_training)/(std_training) for x in values]

final = []
# Max-min Normalization for the output of the NN
for (s1, s2) , st_value in zip(combinations, normalized):
    final.append([[s1,s2], [st_value]])
    print(f"Pair:{s1,s2} --> {st_value} ",)
```

```
Pair:(2, 2) --> 0.0
Pair:(2, 4) --> 0.3329668930670624
Pair:(2, 6) --> 0.4894631578876288
Pair:(2, 8) --> 0.543654801035603
Pair:(2, 10) --> 0.5464052791490887
Pair:(4, 2) --> 0.34731989670391716
Pair:(4, 4) --> 0.6048384320283968
Pair:(4, 6) --> 0.7277204108628127
Pair:(4, 8) --> 0.767257188766194
Pair:(4, 10) --> 0.7696519815083037
Pair:(6, 2) --> 0.5596260220409397
Pair:(6, 4) --> 0.7717083302275003
Pair:(6, 6) --> 0.8732868634139147
Pair:(6, 8) --> 0.90598264095888
Pair:(6, 10) --> 0.9049712865254186
Pair:(8, 2) --> 0.6752592855073805
Pair:(8, 4) --> 0.8629338624781059
Pair:(8, 6) --> 0.9523181734809203
Pair:(8, 8) --> 0.9808865654175629
Pair:(8, 10) --> 0.9795814229409096
Pair:(10, 2) --> 0.710386103043338
Pair:(10, 4) --> 0.888936206483296
Pair:(10, 6) --> 0.9739154909636875
Pair:(10, 8) --> 0.999878364208877
Pair:(10, 10) --> 1.0
```

[22]:
```
mean_training
```

[22]:
```
100981.20129999999
```

1.1.3 Neaural Network

[7]:
```
# Normalization of the input
X_train = np.zeros((25,2))
for i, (s1, s2) in enumerate(combinations):
    X_train[i,0] = s1/10
    X_train[i,1] = s2/10
```

```
# output normalized
y_train = normalized
```

[8]:
```
# activation function
def sigmoid(x):
    return 1/(1+ np.exp(-x))

def sigmoid_derivative(x):

    return x*(1-x)
```

[9]:
```
# function: convert normalized output to denormalized gain
def convert_to_gain(x):
    return x*(max_training-min_training) +  min_training
```

[10]:
```
import numpy as np

class NeuralNet():
    def __init__(self):
        np.random.seed(41)
        self.bias = np.random.rand(1) # o lo metto fisso ad 1?
        self.learning_rate = 0.2

    def update_weights(self, weights):
        self.weights = weights

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def sigmoid_der(self, x):
        return self.sigmoid(x)*(1-self.sigmoid(x))

    def training(self, X_train, output, tol):

        #

        bias = [1, 1, 1]  # 2 for the 2 hidden neurons and 1 for the output

        weights = 0.5*np.random.rand(3, 3) # weights initialization
        #print(weights)

        # step lenght coefficient, mu
        coeff = 0.01

        #Initialization: SSE initialized at high value, flag: boolean for early stop␣
↪(reach tollerance), n_epochs
        sse_old = 10000
        flag = False
        n_epocs = 10000


        for i in range(n_epocs):
```

```
            if flag == True:
                print("Reached tol: ", tol)
                break

            out = np.zeros(25)

            numInput = X_train.shape[0]#len(X_train) #25 length of input

            for j in range(numInput):

                x2= [0,0]

                # Feed forward

                # Hidden Layer

                # Hidden node 1
                H1 = bias[0]*weights[0,0]+ X_train[j,0]*weights[0,1] +␣
↪X_train[j,1]*weights[0,2];


                x2[0] = sigmoid(H1);

                # Hidden node 2
                H2 = bias[1]*weights[1,0]+ X_train[j,0]*weights[1,1] +␣
↪X_train[j,1]*weights[1,2];

                x2[1] = sigmoid(H2);

                # Output layer
                x3_1 = bias[2]*weights[2,0] + x2[0]*weights[2,1] + x2[1]*weights[2,2];
                out[j] = sigmoid(x3_1);

                # Backpropagation

                delta3_1 = out[j]*(1-out[j])*(output[j]-out[j]);

                # Propagate the delta backwards into hidden layers
                delta2_1 = x2[0]*(1-x2[0])*weights[2,1]*delta3_1;
                delta2_2 = x2[1]*(1-x2[1])*weights[2,2]*delta3_1;


                # Add weight changes to original weights
                # And use the new weights to repeat process.
                # delta weight = coeff*x*delta
                for k in range(3):# 1:3
                    if k == 0: # Bias cases
                        weights[0,k] = weights[0,k] + 2*coeff*bias[0]*delta2_1;
                        weights[1,k] = weights[1,k] + 2*coeff*bias[1]*delta2_2;
                        weights[2,k] = weights[2,k] + 2*coeff*bias[2]*delta3_1;
                    else: #% When k=2 or 3 input cases to neurons
                        weights[0,k] = weights[0,k] + 2*coeff*X_train[j,0]*delta2_1;
                        weights[1,k] = weights[1,k] + 2*coeff*X_train[j,0]*delta2_2;
```

```
                        weights[2,k] = weights[2,k] + 2*coeff*x2[k-1]*delta3_1;

                # methods: update weights
                self.update_weights(weights)

            # compute the SSE
            sse = (output-out)**2
            print("Error: ",sse.sum())

            # Early stop, reached tollerance
            if abs(sse.sum() -sse_old)< tol:
                print(f"Reached at epoch {i}: ",abs(sse.sum() -sse_old))
                flag = True
                break
            # update sse_old
            sse_old = sse.sum()
            # update coeff
            coeff = coeff*0.9999
        return out

    # methods for prediction
    def think(self, x_valid):

        bias = [1,1,1]

        weights = self.weights

        x2= [0,0]
                # Hidden node 1
        H1 = bias[0]*weights[0,0]+ x_valid[0]*weights[0,1] + x_valid[1]*weights[0,2];


        x2[0] = sigmoid(H1);

                # Hidden node 2
        H2 = bias[1]*weights[1,0]+ x_valid[0]*weights[1,1] + x_valid[1]*weights[1,2];

        x2[1] = sigmoid(H2);

                # Output layer
        x3_1 = bias[2]*weights[2,0] + x2[0]*weights[2,1] + x2[1]*weights[2,2];
        out = sigmoid(x3_1);

        return out
```

[11]:
```
## Training NN
net = NeuralNet()

tol = 10**(-6)
out= net.training(X_train, y_train, tol)
```

```
Error:  1.5644879423255782
Error:  1.5576685772752479
Error:  1.5514248395511812
```

```
.............................
Error:  0.19349412373873698
Error:  0.19349302940839494
Error:  0.1934919393631943
Error:  0.19349085359693105
Error:  0.19348977210341087
Error:  0.19348869487644724
Error:  0.19348762190986288
Error:  0.1934865531974897
Error:  0.19348548873316726
Error:  0.19348442851074535
Error:  0.19348337252408127
Error:  0.1934823207670417
Error:  0.19348127323350126
Error:  0.1934802299173446
Error:  0.19347919081246392
Error:  0.1934781559127608
Error:  0.19347712521214466
Error:  0.1934760987045349
Error:  0.19347507638385755
Error:  0.1934740582440495
Error:  0.19347304427905512
Error:  0.19347203448282715
Error:  0.1934710288493268
Error:  0.19347002737252555
Error:  0.19346903004640098
Reached at epoch 5998:  9.973261245743714e-07
```

1.1.4 Comparison

[12]:
```
print("True  -  Predicted")
tot_error = 0
for i in range(len(out)):
    print(f"{y_train[i]:.4f} - {out[i]:.4f}, Error: {y_train[i] - out[i]}")
    tot_error += (y_train[i] - out[i])**2

print("SSE: ",tot_error)
```

```
True  -  Predicted
0.0000 - 0.2090, Error: -0.20902028020959426
0.3330 - 0.3172, Error: 0.015803499460750103
0.4895 - 0.4779, Error: 0.011553428883318617
0.5437 - 0.6527, Error: -0.10906014144621357
0.5464 - 0.7851, Error: -0.23870814356660597
0.3473 - 0.3322, Error: 0.015158083030664793
0.6048 - 0.4972, Error: 0.10767995919793438
0.7277 - 0.6698, Error: 0.05789414322410291
0.7673 - 0.7961, Error: -0.02889105805020975
0.7697 - 0.8688, Error: -0.09910371861850176
0.5596 - 0.5172, Error: 0.04247012374716641
0.7717 - 0.6866, Error: 0.08508326115273324
0.8733 - 0.8066, Error: 0.06671997953380426
0.9060 - 0.8744, Error: 0.031594375733392854
0.9050 - 0.9100, Error: -0.004992214684545138
```

```
0.6753 - 0.7029, Error: -0.02767100042908699
0.8629 - 0.8163, Error: 0.04664437249871456
0.9523 - 0.8795, Error: 0.07277696839614012
0.9809 - 0.9127, Error: 0.0682050595991267
0.9796 - 0.9303, Error: 0.04924477603949895
0.7104 - 0.8255, Error: -0.11508107321442052
0.8889 - 0.8843, Error: 0.004620425835072273
0.9739 - 0.9152, Error: 0.05874111403114113
0.9999 - 0.9317, Error: 0.06818579680965797
1.0000 - 0.9409, Error: 0.059082056805168426
SSE:  0.19346903004640092
```

1.1.5 Evaluate the result

[13]:
```
# Start the evaluation
now = datetime.datetime.now()
result = []
tot = []

for s1, s2 in zip([2, 3, 1], [5, 9, 9]):

    p12, p23 = compute_probabilities(s1, s2)

    result.append(s1)
    result.append(s2)
    tot.append(s1)
    tot.append(s2)

    i_result = []
    tot_g = []
    for i in range(10000):

        tot_gain, last_gain = scheduling(p12, p23, s1, s2)
        i_result.append(last_gain)
        tot_g.append(tot_gain)
    result.append(i_result)
    tot.append(tot_g)

finish = datetime.datetime.now()

delta = finish - now

print(f"Start: {now}\nFinish: {finish}\nDeltaT:{delta}" )
```

```
Start: 2021-02-10 17:07:57.607211
Finish: 2021-02-10 17:08:06.618197
DeltaT:0:00:09.010986
```

[21]:
```
# compute the expected values
expected_values = {}
array = []

final = []
for r in (tot[2::3]):
```

```
    #expected_values[(s1,s2)] = sum(r)/10000
    array.append(sum(r)/10000)

## Print result
i = 0
for s1, s2 in zip([0.2, 0.3, 0.1], [0.5, 0.9, 0.9]):
    out= net.think([s1,s2])
    gain = convert_to_gain(out)

    print(f"Expected gain from {s1}-{s2}: {array[i]:.2f} | Predicted: {gain:.2f}|␣
 ↪AbsolutError: {abs(array[i]- gain)} ")
    i = i+1
```

```
Expected gain from 0.2-0.5: 94921.77 | Predicted: 94195.57| AbsolutError:
726.1974872872524
Expected gain from 0.3-0.9: 99904.25 | Predicted: 102283.53| AbsolutError:
2379.281943903261
Expected gain from 0.1-0.9: 93874.53 | Predicted: 99301.67| AbsolutError:
5427.143470283074
```

[15]:
```
# Example of prediction: given an input it get the prediction normalized. Then␣
 ↪denormalized.
result = net.think([0.5,0.9])
print("Result normalized:", result)
print("Gain:",convert_to_gain(result))
```

```
Result normalized: 0.87173016070081
Gain: 103922.46083616714
```

1.1.6 Optimization

[16]:
```
# Box projection: input constraints [0, 1]
def box_projection(x):
    mins = [0,0]
    maxs = [1,1]
    xhat = np.maximum(np.minimum(x, maxs), mins);
    return xhat
```

[17]:
```
# MU and C for Spall method
def mu(m):

    A = 10
    B = 100
    l = 1
    return A/((B+m)**l)

def c(m):
    C=0.1
    t = 0.5
    return C/(m)**t

def spall_gradf(x, c):
    gradf = np.zeros(2)
```

```
    h = (2*np.random.randint(0,2,size=(2))-1) # h: random vector -1, 1
    x_plus = x.copy() + c* h
    x_minus = x.copy() - c* h

    gradf = (net.think(x_minus.tolist())- net.think(x_plus.tolist()))/(2*h)


    return gradf
```

[18]:
```
# Armijo function
def farmijo(fk, alpha, xk, pk, h):
    farm = fk - c1* alpha*np.dot(gradf(xk, h),pk)
    return farm
```

[19]:
```
# Finite difference
def gradf(x, h):
    gradf = np.zeros(2)

    # compute gradf
    for i in range(2):

        # centered finite difference
        x_plus = x.copy()
        x_minus = x.copy()
        x_plus[i]+= h
        x_minus[i]-=h
        #gradf[i] = (net.think(x_plus.tolist())- net.think(x_minus.tolist()))/(2*h)
        gradf[i] = (net.think(x_minus.tolist())- net.think(x_plus.tolist()))/(2*h)

    return gradf
```

[25]:
```
x0 = np.random.rand(2) # random starting point between 0 and 1
print("Starting point:", x0)
print(f"Gain at x0: {convert_to_gain(net.think(x0))}")

# Init

h = 10**(-12)*np.linalg.norm(x0) # h value for finite difference
gamma = 0.8 #  factor that multiplies the descent direction before the (possible)␣
 ↪projection
tolx = 10e-12 # a real scalar value characterizing the tolerance with respect to the
              #norm of  |xk+1 - xk| in order to stop the method;
alpha0 = 0.7 #the initial factor that multiplies the descent direction at each
             #iteration;
c1 = 1e-4 # the factor of the Armijo condition that must be a scalar in (0,1);
rho = 0.8 # fixed factor, lesser than 1, used for reducing alpha0;
btmax = 50 # maximum number of steps for updating alpha during the backtracking␣
 ↪strategy.
tollgrad = 1e-12 #value used as stopping criterion w.r.t. the norm of the gradient;

x = x0.copy()
xk = box_projection(x) # projection
kmax = 10000 # max number of iterations
```

```
k = 0 # init k for iterations

fk = net.think(xk)

gradfk_norm = np.linalg.norm(gradf(xk, h))
deltaxk_norm = tolx +1

spall = False # set True: use Spall Method, False: finite difference
while k<kmax and gradfk_norm >= tollgrad and deltaxk_norm >= tolx:


    if spall == True:
        # gradient
        pk = -spall_gradf(xk, c(k+1))
        gamma = mu(k+1)
        #update x and step-length
        xbark = xk + gamma*pk

        # box proj. if exceed the constraint
        xhatk = box_projection(xbark)

        xnew = xhatk
        fnew = net.think(xnew)
        deltaxk_norm = np.linalg.norm(xnew - xk)
        xk = xnew
        fk = fnew
        gradfk_norm = np.linalg.norm(spall_gradf(xk, c(k+1)));
    else:
        # gradient
        pk = -gradf(xk, h)
        #update x and step-length
        xbark = xk + gamma*pk

        # box proj. if exceed the constraints

        xhatk = box_projection(xbark)


        # Reset the value of alpha for Armijo
        alpha = alpha0

        #compute the candidate xk
        pik = xhatk - xk

        xnew = xk + alpha*pik

        # compute the function f in xnew
        fnew = net.think(xnew)

        #Backtracing strategy with Armijo condition
        bt = 0

        while bt<btmax and fnew < farmijo(fk,alpha, xk, pik,h):
```

```
            #Reduce the value of alpha
            alpha = rho * alpha
            # Update xnew and fnew w.r.t. the reduced alpha
            xnew = xk + alpha * pik
            fnew = net.think(xnew)


            # increase the counter
            bt = bt + 1


        # Update xk, fk, gradfk_norm, deltaxk_norm
        deltaxk_norm = np.linalg.norm(xnew - xk)

        xk = xnew
        fk = fnew
        gradfk_norm = np.linalg.norm(gradf(xk, h));


    # Increase the step by one
    k = k + 1;


print(f"-------------------------------------\nEnd point at iteration K:{k+1} ,xk:
 ↪{xk}")
if spall == True:
    print("Spall method")
else:
    print("Finite difference")

res = net.think(xk)
print(f"Result:{fnew}")

print(f"Optimized gain:{convert_to_gain(fnew)}")
```

```
Starting point: [0.17768291 0.96347716]
Gain at x0: 101473.22560120237
-------------------------------------
End point at iteration K:30 ,xk:[1. 1.]
Finite difference
Result:0.9409237302420774
Optimized gain:105326.29016417477
```

[ ]:

[ ]:

1.2 Neural Network results

The quality of the approximation has been evaluated at this three further values (s1 = 0.2; s2 = 0.5), (s1 = 0.3; s2 = 0.9), (s1 = 0.1; s2 = 0.9) and compared them with the prevision of the network.
The following table provides the results obtained.

As might be expected the predicted result for [s1,s2] = [0.2, 0.5] has a relative low absolute error because

s1	s2	Simulation	Predicted	AbsError
0.2	0.5	94921.77	94195.57	726.197
0.3	0.9	99904.25	102283.53	2379.28
0.1	0.9	93874.53	99301.67	5427.14

the Neural Network has been trained with even numbers, so (s1= 0.2) has already been seen. On the other hand, pair of odd numbers do not belong to the training set on which our model has been trained, so the approximations have higher absolute errors.

2 Optimization

This section explain a possible approach for the optimization task, which aim is to maximize the gain through the built model. In order to do that, we adapted the steepest descent method to work in this setting. However, since the function is not known in any explicit form, the gradient is not known and needs to be estimated from direct evaluations of the function f. Stochastic gradient method has been implemented with two approaches:

- Centered finite differences (CFD)
- Spall method

Spall method does simultaneous perturbation and requires only 2 function evaluations for each iteration, indeed the numerator is the same for each component, whereas the Centered FD requires $2 * N$ function evaluation. In this problem, CFD works well since $N = 2$ so there are no problem about the computational cost.
A box projection function has been implemented in order to maintain the constraint: $0 \leq s1, s2 \leq 10$.

2.1 Tuning parameter and Result

The initial point has been choosen randomly between 0 and 1 (since training inputs had been normalize with a decimal normalization). The explanation of the parameters is in section 1.1.6 Optimization at [25]. Obviously, precise values change at each trial, since there are variables (e.g. x0, h for Spall..) that are initialized as random.

2.1.1 Results: Spall

For the implementation of $\mu : A = 10, B = 100, l = 1$ have been choosen.
For the implementation of $c : C = 0.1, t = 0.5$ have been chosen.

tollgrad	tolx	x	Point	Gain	Iteration
10^{-12}	10^{-12}	x0:	[0.80272798 0.63937322]	104262	0
		xk:	[0.86038796 0.69303146]	104651	10000
10^{-12}	10^{-6}	x0:	[0.31179984 0.17112321]	91383	0
		xk:	[0.49885129 0.34086332]	97259	520
10^{-8}	10^{-6}	x0:	[0.23022621 0.77827517]	99672	0
		xk:	[0.35835989 0.89395774]	102831	288

As tolerances get smaller the number of iterations increases. In fact, as we can see in the first row of the table above the maximum number of iteration are reached. On the contrary the errors increase the number of iterations decreases, and an early stop criterion is reached. The expected maximum [1 1] is never reached, as a consequence the initial parameter μ and c can be tuned in order to get better results (with μ computed with: $A = 1000, B = 500, l = 0.6$ the maximum is reached after 25-50 iterations, depending on the starting point).

2.1.2 Results: Finite Difference

Case 1 :

For the implementation: $\gamma : 0.8$ as step-length, $h : 10^{-6} * \|x\|$ value for finite difference.
For the Armijo's condition: $\alpha : 0.7$, $\rho : 0.8$, *btmax*: 50 (max number of backtracking operations for Armijo).

tollgrad	tolx	x	Point	Gain	Iteration
10^{-10}	10^{-10}	x0:	[0.04258496 0.76779936]	95810	0
		xk:	[1. 1.]	105326	26
10^{-8}	10^{-6}	x0:	[0.28295119 0.64437429]	98404	0
		xk:	[0.99999799 0.99999997]	105326	15

As ttolerances get smaller the number of iterations increases and the maximum [1 1] has been reached in the first row with just 26 iterations, whereas with a large tolerance an early stop occur after 15 iterations and it's very close to the optimum.

Case 2 :
For the implementation: $\gamma : 0.9$ as step-length, $h : 10^{-6} * \|x\|$ value for finite difference.
For the Armijo's condition: $\alpha : 0.75$, $\rho : 0.8$, *btmax*: 50 (max number of backtracking operations for Armijo).

tollgrad	tolx	x	Point	Gain	Iteration
10^{-10}	10^{-10}	x0:	[0.70125869 0.13970315]	97685	0
		xk:	[1. 1.]	105326	19
10^{-8}	10^{-6}	x0:	[0.88571771 0.09344163]	100331	0
		xk:	[1. 0.99999897]	105326	15

In this case, both α and γ are increased. As a result, the function converge faster towards the optimum for both *tolerances* choice.
On the contrary, if α and γ are decreased the number of iterations increase.
Other tuning test could be done, even on the Armijo's parameter.

3 Plot

Here two plots of the points of the surfaces of the training set (left) and on the predicted data by the Network (right).