NN_regression

December 1, 2025

#

UP3, Optimization for machine learning: regression with a Neural Network from scratch

This notebook contains the questions of the practical session along with complementary guidelines and examples. The code is written in Python. The questions are in red.

First import all given code. You are encouraged to have a look at forward_propagation.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from typing import Callable, List

from gradient_descent import gradient_descent
from optim_utilities import print_rec
from test_functions import (
    linear_function,
    ackley,
    sphere,
    quadratic,
    rosen,
    L1norm,
    sphereL1,
    rastrigin,
    michalewicz,
    schwefel
)
from restarted_gradient_descent import restarted_gradient_descent
from random_search import random_opt  # always useful to compare optim algos to␣
 ↪a random search

# auto reload to reload functions imported that have been changed (cf.␣
 ↪test_functions.sphereL1 for lbda)
%load_ext autoreload
%autoreload 2
```

[2]:
```
from forward_propagation import (
    forward_propagation,
    create_weights,
```

```
    vector_to_weights,
    weights_to_vector)
from activation_functions import (
    relu,
    sigmoid,
    linear,
    leaky_relu
)
```

0.0.1 Data structure behind the forward propagation

To start, let's have a look at the structure used to encode the network. Don't worry, more handy utility functions will be introduced soon to manipulate the network globally.

The following network has 2 layers, the first going from 4 input components to the 3 internal neurons, the second going from the 3 internal neurons outputs to the 2 outputs. Don't forget the additional weight for the neurons biases. * `weight[l][i,j]` links entry `j` to output `i` in a layer `l` * the last column of the weights are the biases of the neurons

[3]:
```
# input data
inputs = np.array([[1,2,5,4],[1,0.2,0.15,0.024],[2,1.2,-1.01,-0.4]])
# describe the neural net
weights = [
        np.array(
            [
                [1,0.2,0.5,1,-1],
                [2,1,3,5,0],
                [0.2,0.1,0.6,0.78,1]
            ]
        ),
    np.array(
            [
                [1,0.2,0.5,1],
                [2,1,3,5]
            ]
        )
    ]
activation = sigmoid
# carry out forward propagation
y=forward_propagation(inputs,weights,activation)
# reporting
print(f'This network has {inputs.shape[1]} inputs, {y.shape[1]} outputs and␣
 ↪{inputs.shape[0]} data points')
print('The predictions are:\n',y)
```

```
This network has 4 inputs, 2 outputs and 3 data points
The predictions are:
 [[0.93695121 0.99998324]
```

```
 [0.89266103 0.99991581]
 [0.88231234 0.99982432]]
```

0.0.2 Create a data set

The data set is made of points sampled randomly from a function.

[4]:
```
def simulate_data_target(fun: Callable,
                         n_features: int,
                         n_obs: int,
                         LB: List[float],
                         UB: List[float]) -> dict:

    entry_data = np.random.uniform(low= LB,high=UB,
                                   size=(n_obs, n_features))
    target = np.apply_along_axis(fun, 1, entry_data)

    return {"data": entry_data, "target": target}
```

[5]:
```
used_function = linear_function
n_features = 2
n_obs = 10
LB = [-5] * n_features
UB = [5] * n_features
simulated_data = simulate_data_target(fun = used_function,n_features =␣
 ↪n_features,n_obs=n_obs,LB=LB,UB=UB)
print('x,f(x) =\n',np.
 ↪concatenate((simulated_data["data"],simulated_data["target"].
 ↪reshape(n_obs,1)),axis=1)) # don't print for too large a data set
```

```
x,f(x) =
 [[-4.48730839 -3.33877101 -8.16485041]
 [ 1.32834258  4.7588702  13.84608298]
 [ 0.34236662 -0.66041405  2.02153853]
 [-0.65700742 -4.4972725  -6.65155243]
 [ 3.88777305  1.04482474  8.97742254]
 [-1.39616902  2.24346348  6.09075795]
 [ 4.50601315  4.23574379 15.97750072]
 [-1.59851179  0.54658539  2.49465899]
 [ 1.60275284  0.95612586  6.51500457]
 [ 2.50839555 -2.22507965  1.05823625]]
```

0.0.3 Make a neural network, randomly initialize its weights, propagate input data

Create a NN with 1 layer, 2 inputs, 1 output, thus 1 neuron in the layer. Propagate the data inputs through it.

[6]:
```
used_network_structure = [2,1]
used_activation = leaky_relu
weights = create_weights(used_network_structure)
weights_as_vector,_ = weights_to_vector(weights)
dim = len(weights_as_vector)
print("weights=",weights)
print("nb. of NN parameters to learn, dim=",dim)
```

```
weights= [array([[-1.65849458,  0.17016686,  0.30139831]])]
nb. of NN parameters to learn, dim= 3
```

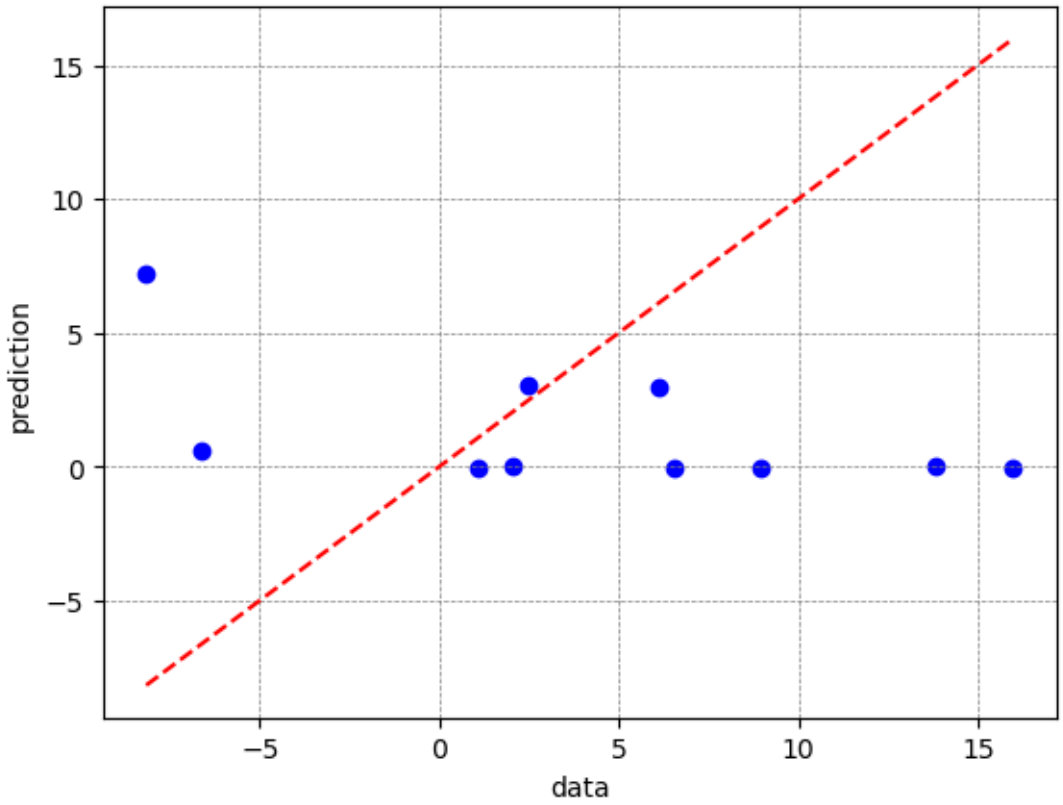
[7]:
```
predicted_output =␣
  ↪forward_propagation(inputs=simulated_data["data"],weights=weights,activation_functions=used
```

Compare the data and the prediction of the network. Of course, at this point, no training is done so they are different. They just have the same format (provided a `reshape` is done).

[8]:
```
print("data vs prediction\n")
print(np.append(simulated_data["target"].reshape(-1,1),predicted_output,axis=1))
plt.scatter(simulated_data["target"], predicted_output, label='pred vs. data',␣
  ↪color='blue')
min_value = simulated_data["target"].min()
max_value = simulated_data["target"].max()
plt.plot([min_value, max_value], [min_value, max_value], color='red',␣
  ↪linestyle='--', label='')
plt.xlabel('data')
plt.ylabel('prediction')
plt.grid(color = 'gray', linestyle = '--', linewidth = 0.5)
```

```
data vs prediction

[[-8.16485041e+00  7.17542676e+00]
 [ 1.38460830e+01 -1.09184866e-02]
 [ 2.02153853e+00 -3.78795461e-03]
 [-6.65155243e+00  6.25754805e-01]
 [ 8.97742254e+00 -5.96865768e-02]
 [ 6.09075795e+00  2.99870019e+00]
 [ 1.59775007e+01 -6.45101685e-02]
 [ 2.49465899e+00  3.04553216e+00]
 [ 6.51500457e+00 -2.19405766e-02]
 [ 1.05823625e+00 -4.23739693e-02]]
```



0.1 Error functions

Utility functions to transform weights into a vector and vice versa. This is used in the calculation of the error function (the vector is transformed into NN weights, ...).

[9]:
```
# play with weight transformation functions
weights = create_weights(used_network_structure)
print('weights=\n',weights)
weights_as_vector, _ = weights_to_vector(weights)
print('weights_as_vector=\n',weights_as_vector)
w2 = vector_to_weights(weights_as_vector, used_network_structure)
print('weights back=\n',w2)
```

```
weights=
 [array([[ 0.43839343, -1.28002019,  0.62233895]])]
weights_as_vector=
 [ 0.43839343 -1.28002019  0.62233895]
weights back=
 [array([[ 0.43839343, -1.28002019,  0.62233895]])]
```

We define 2 error functions, one for regression is the mean square error, the other is the cross-entropy error for classification.

[10]:
```
# mean squared error
def cost_function_mse(y_predicted: np.ndarray,y_observed: np.ndarray):
    error = 0.5 * np.mean((y_predicted - y_observed)**2)
    return error
```

[11]:
```
# entropy
def cost_function_entropy(y_predicted: np.ndarray,y_observed: np.ndarray):
    n = len(y_observed)
    term_A = np.multiply(np.log(y_predicted),y_observed)
    term_B = np.multiply(1-y_observed,np.log(1-y_predicted))
    error = - (1/n)*(np.sum(term_A)+np.sum(term_B))
    return(error)
```

[12]:
```
def error_with_parameters(vector_weights: np.ndarray,
                          network_structure: List[int],
                          activation_function: Callable,
                          data: dict,
                          cost_function: Callable,
                          regularization: float = 0) -> float:

    weights = vector_to_weights(vector_weights,network_structure)
    predicted_output =␣
 ↪forward_propagation(data["data"],weights,activation_function)
    predicted_output = predicted_output.reshape(-1,)

    error = cost_function(predicted_output,data["target"]) + regularization *␣
 ↪np.sum(np.abs(vector_weights))

    return error
```

[13]:
```
# used_network_structure and used_activation defined above
used_data = simulated_data
used_cost_function = cost_function_mse


def neural_network_cost(vector_weights):

    cost = error_with_parameters(vector_weights,
                                 network_structure = used_network_structure,
                                 activation_function = used_activation,
                                 data = used_data,
                                 cost_function = used_cost_function)

    return cost
```

Below, the cost function associated to the neural network is calculated from a simple vector in a manner similar to $f(x)$, therefore prone to optimization. The translation of the vector

into as many weight matrices as necessary is done thanks to the * `used_network_structure`, * `used_activation`, * `used_data` and * `used_cost_function` defined above and passed implicitely thanks to Python's scoping rules.

[14]:
```
# call to the NN cost at a random point = random initialization of the weights␣
 ↪and biases
random_weights_as_vect = np.random.uniform(size=dim)
neural_network_cost(random_weights_as_vect)
```

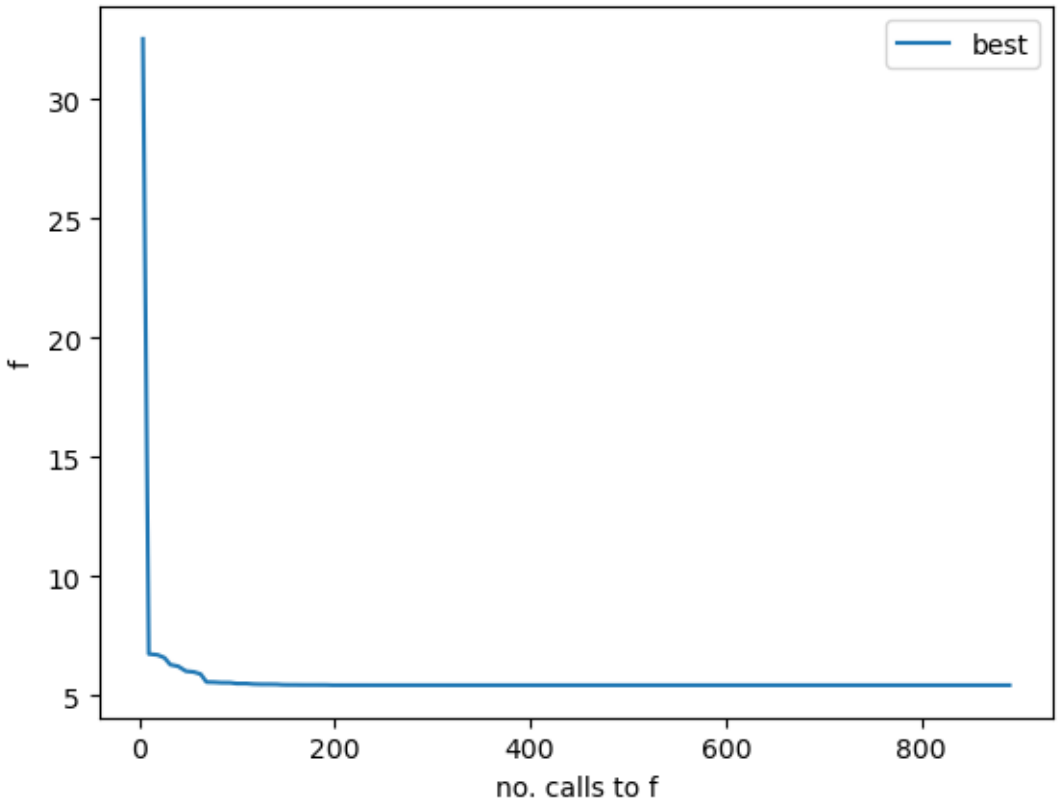
[14]:
```
27.175728123830034
```

0.1.1 Learn the network by gradient descent

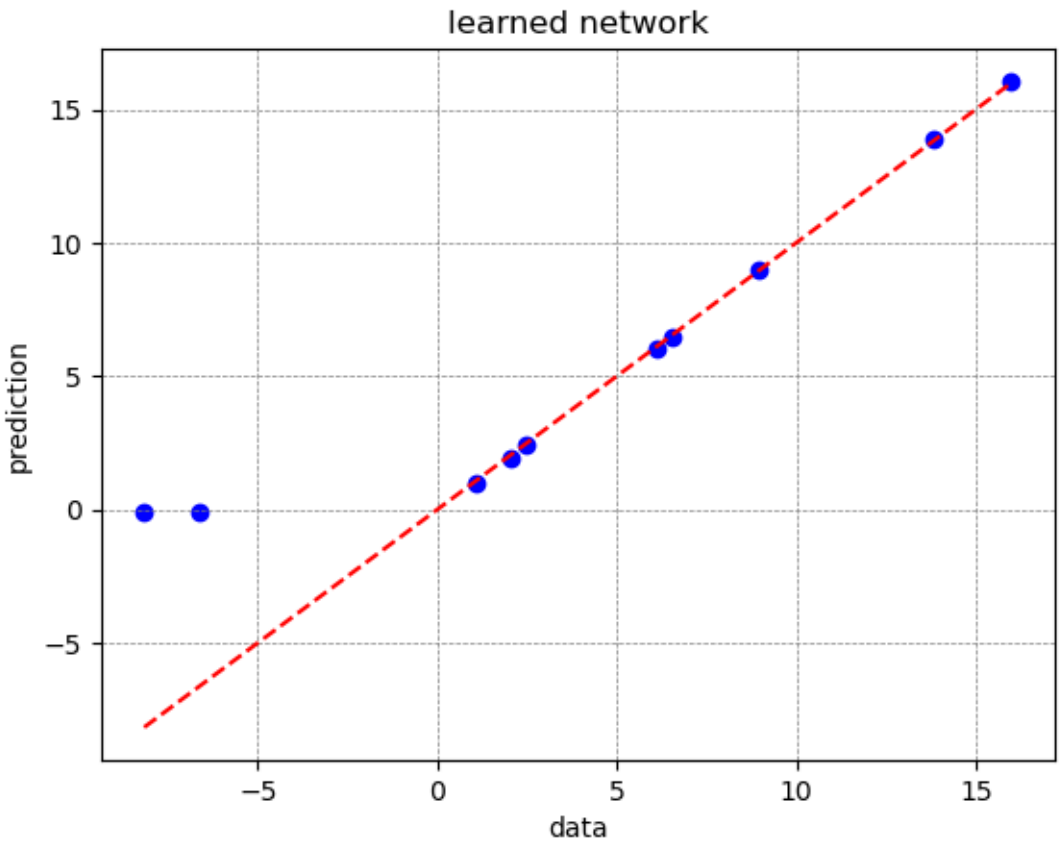
[15]:
```
LB = [-5] * 3
UB = [5] * 3
printlevel = 1
res = gradient_descent(func = neural_network_cost,
                start_x = np.array([0.28677805, -0.07982693,  0.37394315]),
                LB = LB, UB = UB,budget = 1000,printlevel=printlevel)
```

[16]:
```
# reporting
# report optimization convergence
print_rec(res=res, fun=neural_network_cost, dim=len(res["x_best"]),
          LB=LB, UB=UB , printlevel=printlevel, logscale = False)
# look at the network learned
weights_best = vector_to_weights(res["x_best"],used_network_structure) #␣
 ↪extract weights of best network found
print("Best NN weights:\n",weights_best)
predicted_output =␣
 ↪forward_propagation(used_data["data"],weights_best,used_activation)
plt.scatter(used_data["target"], predicted_output, label='pred vs. data',␣
 ↪color='blue')
min_value = used_data["target"].min()
max_value = used_data["target"].max()
plt.plot([min_value, max_value], [min_value, max_value], color='red',␣
 ↪linestyle='--', label='')
plt.xlabel('data')
plt.ylabel('prediction')
plt.title('learned network')
plt.grid(color = 'gray', linestyle = '--', linewidth = 0.5)
```

```
search stopped after 890 evaluations of f because of too small step too small
gradient
best objective function = 5.433791117304952
best x = [1.0144296  2.01767394 2.93742074]
```




```
Best NN weights:
 [array([[1.0144296 , 2.01767394, 2.93742074]])]
```



0.1.2 Question 4: make your own network for regression

3. Generate 100 data points with the sphere function in 2 dimensions.
4. Create a network with 2 inputs, 5 ReLU neurons in the hidden layer, and 1 output.
5. Learn it on the quadratic data points you generated. Plot some results, discuss them.

0.1.3 Answer 4: make your own network for regression

Your code, your explanations

1 THE END

[ ]: