Machine Learning for Robot Control

November 25, 2022

Name: Farros Alferro

Student ID: C0TB1706

Supervisor: Assoc. Prof. Shingo Kagami

1 Objective

Robotics, like many other creative technical sectors today, has been and continues to be affected in various ways by machine learning technologies. Robot arms, for example, are a versatile device utilized in a variety of sectors. In this experiment, we use a single-finger robot to throw a ball and then study its behavior.

Nonetheless, predicting the ball's behavior is challenging because it is significantly influenced by various elements, including the robot's motion, ball property, the elasticity of the robot, and air resistance (in this experiment, we only consider the first factor and omit the rest). As a result, we apply AI methods, particularly regression and optimization, to regulate the movement of the balls. Thus, the goal of these experiments is to learn how to employ AI technology in the context of robot control.

2 Methods and Theory

2.1 Regression with MLP

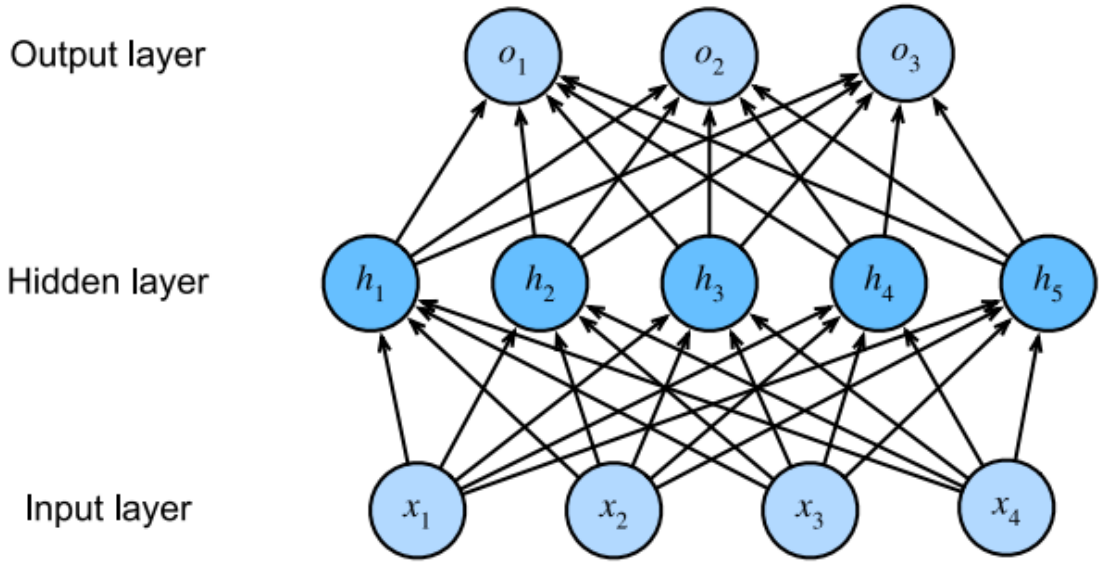
Regression is a technique for determining the relationship between independent variables or features $(x_1, x_2, ..., x_k)$ and a dependent variable or outcome (y). Once the relationship between the independent and dependent variables has been estimated, outcomes can be predicted. It's used in predictive modeling to predict continuous outcomes, therefore it's useful for forecasting and predicting data outputs. In general, machine learning regression entails sketching a line of greatest fit through the data points.

$$y = F(x)$$

This model is then trained so that the errors between each point and the line is minimized.

Model F has a variety of designs. A linear model is the most basic. However, the capability of linear models is restricted; when modeling nonlinear functions, the modeling error becomes considerable. We can overcome the limitations of linear models by incorporating one or more hidden layers. The easiest way to do this is to stack many fully connected layers on top of each other. Each layer feeds

into the layer above it, until we generate outputs. We can think of all the layers except the final layer as our representation and the final layer as our linear predictor. This architecture is commonly called a multilayer perceptron, often abbreviated as MLP. We wil utilized MLP in this experiment [1].



This MLP has 4 inputs, 3 outputs, and its hidden layer contains 5 hidden units. Since the input layer does not involve any calculations, producing outputs with this network requires implementing the computations for both the hidden and output layers; thus, the number of layers in this MLP is 2. Note that both layers are fully connected. Every input influences every neuron in the hidden layer, and each of these in turn influences every neuron in the output layer.

Let us denote by the matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ a minibatch of n examples where each example has d inputs (features). For a one-hidden-layer MLP whose hidden layer has h hidden units, we denote by $\mathbf{H} \in \mathbb{R}^{n \times h}$ the outputs of the hidden layer, which are *hidden representations*. Since the hidden and output layers are both fully connected, we have hidden-layer weights $\mathbf{W}^{(1)} \in \mathbb{R}^{d \times h}$ and biases $\mathbf{b}^{(1)} \in \mathbb{R}^{1 \times h}$ and output-layer weights $\mathbf{W}^{(2)} \in \mathbb{R}^{h \times q}$ and biases $\mathbf{b}^{(2)} \in \mathbb{R}^{1 \times q}$. This allows us to calculate the outputs $\mathbf{O} \in \mathbb{R}^{n \times q}$ of the one-hidden-layer MLP as follows:

$$\mathbf{H} = \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)},$$
$$\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}.$$

Note that after adding the hidden layer, our model now requires us to track and update additional sets of parameters. However, the model does not gain anything even with additional hidden layer. The reason is because the hidden units above are given by an affine function of the inputs, and the outputs (pre-softmax) are just an affine function of the hidden units. An affine function of an affine function is itself an affine function. Moreover, our linear model was already capable of representing any affine function.

To see this formally we can just collapse out the hidden layer in the above definition, yielding an equivalent single-layer model with parameters $\mathbf{W} = \mathbf{W}^{(1)}\mathbf{W}^{(2)}$ and $\mathbf{b} = \mathbf{b}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$:

$$\mathbf{O} = (\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W} + \mathbf{b}.$$

In order to realize the potential of multilayer architectures, we need one more key ingredient: a nonlinear *activation function* σ to be applied to each hidden unit following the affine transformation. For instance, a popular choice is the ReLU (Rectified Linear Unit) activation function $\sigma(x) = \max(0, x)$ operating on its arguments element-wise. The outputs of activation functions $\sigma(\cdot)$ are called *activations*. In general, with activation functions in place, it is no longer possible to collapse our MLP into a linear model:

$$\mathbf{H} = \sigma(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}),$$
$$\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}.$$

Since each row in $\mathbf{X}$ corresponds to an example in the minibatch, with some abuse of notation, we define the nonlinearity σ to apply to its inputs in a row-wise fashion, i.e., one example at a time. Note that we used the same notation for softmax when we denoted a row-wise operation in. Quite frequently the activation functions we use apply not merely row-wise but element-wise. That means that after computing the linear portion of the layer, we can calculate each activation without looking at the values taken by the other hidden units.

To build more general MLPs, we can continue stacking such hidden layers, e.g., $\mathbf{H}^{(1)} = \sigma_1(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})$ and $\mathbf{H}^{(2)} = \sigma_2(\mathbf{H}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)})$, one atop another, yielding ever more expressive models [2].

2.2 Optimization with CMA-ES

Function optimization is the challenge of determining the set of inputs to an objective goal function that results in maximum or minimum output of the function. The objective function must be minimized or maximized because it describes the difference between the actual value of the estimated parameter and what the model predicted. It can be a difficult task because the function has tens, hundreds, thousands, or even millions of inputs, and the structure of the function is unknown and frequently non-differentiable and noisy. Instead of solving it analytically, we can use numerical optimization in this case.

Hill climbing is a simple example of an optimization algorithm. We begin with a random (or predetermined) initial value of x_0 and compute the gradient of the objective function, $E(x)$, with respect to x about x_0.

$$\frac{dE}{dx}(x_0)$$

When x_0 is changed to the gradient's negative direction, the value of the evaluation function at the modified x decreases. We will eventually identify x that minimizes E by iterating these modifications.

Other approaches that use gradients include the gradient descent method and Newton's method. MLP is trained via back-propagation, which is also a gradient approach. One disadvantage of the

gradient approach is that it is frequently trapped by local optima. Consider the case where E has many peak positions. This problem can occur when E employs learning models such as MLP.

We will employ CMA-ES (Covariance Matrix Adaptation Evolution Strategy) in this experiment, which is another strategy that does not require us to provide the gradient of. It employs a population of search points, making it more resistant to noisy functions than gradient approaches. CMA-ES is an evolutionary algorithm, which is based on the principle of biological evolution, namely the repeated interplay of variation (via recombination and mutation) and selection: in each generation (iteration), new individuals (candidate solutions, x) are generated by variation of the current parental individuals, usually in a stochastic way. Then, based on their fitness or objective function value E, some individuals are chosen to be parents in the next generation. Like this, over the generation sequence, individuals with better and better E-values are generated [3][4].

The CMA-ES algorithm employs two fundamental ideas for the adaptation of search distribution parameters:

1. A maximum-likelihood principle that seeks to increase the likelihood of successful candidate solutions and search processes. The distribution's mean is adjusted to optimize the likelihood of previously successful candidate solutions. The distribution's covariance matrix is changed (incrementally) to boost the likelihood of previously successful search steps. Both updates are examples of natural gradient descent. As a result, the CMA performs an iterated principal components analysis of successful search steps while keeping all primary axes intact. The Cross-Entropy Method and distribution algorithm estimation are based on very similar ideas, but they estimate the covariance matrix (non-incrementally) by maximizing the likelihood of successful solution points rather than successful search steps.

2. Two pathways of the time evolution of the strategy's distribution mean are recorded, which are referred to as search or evolution paths. These pathways offer important information about the relationship between successive steps. Specifically, if successive steps in the same direction are done, the evolution routes become long. Two methods are used to exploit the evolution routes. In place of single successful search steps, one path is used for the covariance matrix adaptation technique, allowing for a potentially much faster variance rise of favorable directions. The other path is utilized to do another step-size control. The goal of this step-size control is to make sequential distribution mean movements orthogonal in expectation. The step-size adjustment efficiently prevents premature convergence while still allowing for quick convergence to an optimum.

2.3 Model-based Reinforcement Learning for Throwing Motion

Assume a control parameter x that alters the flight distance of the robot's thrown ball. x, for example, is the torque parameter. Let y be the flight distance. Let f represent the dynamics (the analytical model), thus $y = f(x)$. The ball throwing issue is one in which we must discover a control parameter x_{opt} that achieves $y_{trg} \approx f(x_{opt})$ for a given target of flying distance y_{trg}. It is important to note that we must solve this problem for any y_{trg} [1].

The following is the proposed solution::

1. Training Phase:

   1.1. Throwing the ball using random control parameters (x) and measuring the distances it travels (y). We obtain $\{x, y|k = 1, 2, ..., N\}$. Here, we can choose one of three available

parameters: torque, initial angle, and target angle.

1.2. Using the data obtained above, we train an MLP model.

2. The concept of optimizing the control parameter is as follows: we define the objective function as the squared error of y_{trg} and $f(x)$:

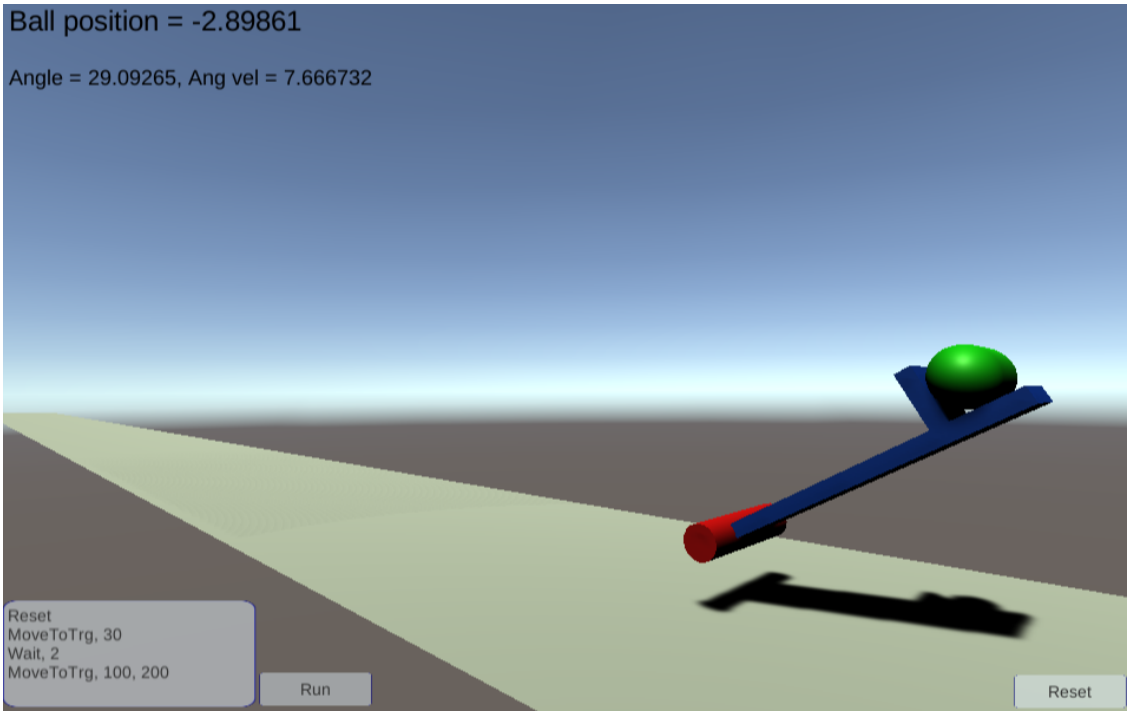
$$f_{error}(x) = (y_{trg} - f(x))^2$$

Then, using CMA-ES, we minimize this objective function in relation to x. The obtained x is x_{opt}, which fulfills $y_{trg} \approx f(x_{opt})$.

3 Experiments and Results

We employ a robot simulator that is similar to the real-life single-finger robot. It is built with the Unity game engine and replicates the physics of a robot and a ball. The following link will lead us to the robot arm:

http://akihikoy.net/p/FingerRobotWebGL/index.html



There is a text box and the Run button visible in the bottom left corner. The text box contains a list of commands for controlling the robot. To run the commands, press the Run button. By default, it goes to 30 degrees, waits 2 seconds, and then moves to 100 degrees with 200 power. The robot will throw the ball, and the ball will land on the ground after a few seconds. The position of the ball is always indicated in the upper left corner. When the ball lands, the "First landing location" is presented, which indicates the ball's flying distance.

3.1 Environment Setup

The experiment results were calculated and evaluated using Google Colaboratory or Jupyter Notebook to make it clearer as we can write code and text. Once a notebook is created, we first clone the library that contains functions necessary to process the experiment results to our local directory. In addition, we also import the **numpy** [5] library to make array-based calculations easier and the **tabulate** [6] library to make table-like output in this notebook.

[1]:
```
!git clone https://github.com/akihikoy/ai_ctrl_1.git

import sys
sys.path.append('ai_ctrl_1/sample')
from libaictrl2 import *
import numpy as np
from tabulate import tabulate
```

```
fatal: destination path 'ai_ctrl_1' already exists and is not an empty
directory.
```

3.2 Regression with MLP

In this exercise, we create synthetic data and use it to train the MLP model. To do this, we use a function from numpy.random module to generate an array that consists of 30 random samples from a uniform distribution over [0, 1).

[2]:
```
# Create artificial samples
np.random.seed(1234)
data_x = np.sort(np.random.rand(30))
print(f'data x: {data_x}')
```

```
data x: [0.01376845 0.07538124 0.19151945 0.27259261 0.27646426 0.31683612
 0.35781727 0.36488598 0.36882401 0.37025075 0.39720258 0.43772774
 0.50099513 0.50308317 0.56119619 0.56809865 0.61539618 0.62210877
 0.65137814 0.68346294 0.71270203 0.77282662 0.77997581 0.78535858
 0.78873014 0.80187218 0.87593263 0.88264119 0.9331401  0.95813935]
```

After that, we create two functions: a linear function and a parabolic function. Moreover, we also added some constants in order to make the data more visible when it is plotted. Then we input the x data to each function to generate y values with their corresponding behaviour (linear and nonlinear).

[3]:
```
# Generate the y value for both linear and nonlinear functions
def linear_fc(x):
    return x + 0.05

def second_order_fc(x):
    return 4 * (x - 0.5) ** 2 + 0.05

data_y = linear_fc(data_x)
data_y_nonlinear = second_order_fc(data_x)
```

```
print(f'linear y: {data_y}\n')
print(f'nonlinear y: {data_y_nonlinear}')
```

```
linear y: [0.06376845 0.12538124 0.24151945 0.32259261 0.32646426 0.36683612
 0.40781727 0.41488598 0.41882401 0.42025075 0.44720258 0.48772774
 0.55099513 0.55308317 0.61119619 0.61809865 0.66539618 0.67210877
 0.70137814 0.73346294 0.76270203 0.82282662 0.82997581 0.83535858
 0.83873014 0.85187218 0.92593263 0.93264119 0.9831401  1.00813935]

nonlinear y: [0.99568448 0.77120436 0.430641   0.25685649 0.24987292 0.18419602
 0.13086371 0.12302319 0.11882857 0.11733947 0.09226924 0.06551134
 0.05000396 0.05003802 0.06497989 0.06854971 0.10326511 0.10964221
 0.14166137 0.18463459 0.23096861 0.34773746 0.36354581 0.37571809
 0.38346038 0.41450725 0.61530138 0.63565712 0.80044139 0.88956667]
```

Then, we train the MLP model for each case and test the model for one sample $x = 0.75$.

[4]:
```
# Train the MLP model
f = TrainMLPR(data_x, data_y)
f_nonlinear = TrainMLPR(data_x, data_y_nonlinear)
print(f'linear data: f(0.75) = {f(0.75)}')
print(f'nonlinear data: f(0.75) = {f_nonlinear(0.75)}')
```

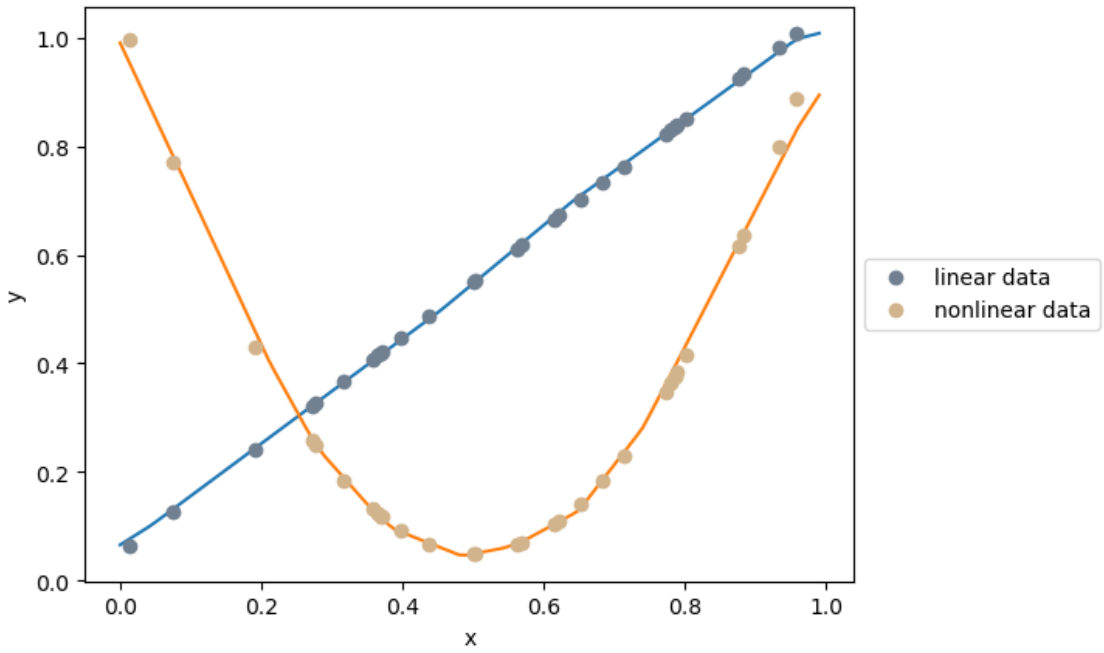
```
linear data: f(0.75) = 0.8020389724749504
nonlinear data: f(0.75) = 0.3055841752437347
```

Lastly, we plot the model's predicted value for both the linear and nonlinear data. As seen from the figure below, the MLP model can predict the function of the generated data: linear function for the gray-colored samples and 2th polynomial function for the tan-colored samples. This shows that the MLP model is able to approximate nonlinear data, a feature that cannot be found in linear regression model.

[5]:
```
# Line plot the output of the model
plot = PlotF(f, xmin=0.0, xmax=1.0, show=False)
plot_nonlinear = PlotF(f_nonlinear, xmin=0.0, xmax=1.0, show=False)

# Scatter plot the data samples
plot.plot(data_x, data_y, 'o', c='slategray', label='linear data')
plot.plot(data_x, data_y_nonlinear, 'o', c='tan', label='nonlinear data')

# Show the labels, legends, and the figure
plot.xlabel('x')
plot.ylabel('y')
plot.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plot.show()
```



3.3 Optimization with CMA-ES

In this exercise, we are going to find an x point on the previous plot with the nearest value to the target value y_{trg}. To do this, first, we define an objective function of each case, which is the squared error of y_{trg} and $f(x)$ learned in the previous section.

[6]:
```
# Define the objective function
def f_error(x):
  y_trg = 0.75
  return (y_trg - f(x))**2

def f_error_nonlinear(x):
  y_trg = 0.75
  return (y_trg - f_nonlinear(x))**2
```

Next, we optimize the objective function by inputting init to the *Fmin* function, which is an optimizer function that utilizes the CMA-ES algorithm and return the x value that optimizes the input function. In the below code, we use $x = 0.5$ as the initial value with a search range between 0.0 and 1.0.

[7]:
```
# Finding x that minimizes the objective function
x_opt = FMin(f_error, 0.5, 0.0, 1.0)
x_opt_nonlinear = FMin(f_error_nonlinear, 0.5, 0.0, 1.0)
```

We can observe from the upper code output that the *Fmin* function is able to obtain the x_{opt} value after 44 and 50 iterations for both linear and nonlinear cases, respectively. To test this, we input back the obtained value to the MLP model, and as shown below, the *Fmin* does a great

job of acquiring the x value that optimizes the previous objective function. Moreover, we also add the coordinate $(x_{opt},\ f_{error}(x_{opt}))$ to the previous plot, indicated by the 'x' symbol for the linear function and '*' for the nonlinear function.

[8]:
```
# Print the result
print('Linear solution:', x_opt, f(x_opt), f_error(x_opt))
print('Nonlinear solution:', x_opt_nonlinear, f_nonlinear(x_opt_nonlinear),␣
↪f_error_nonlinear(x_opt_nonlinear))
```

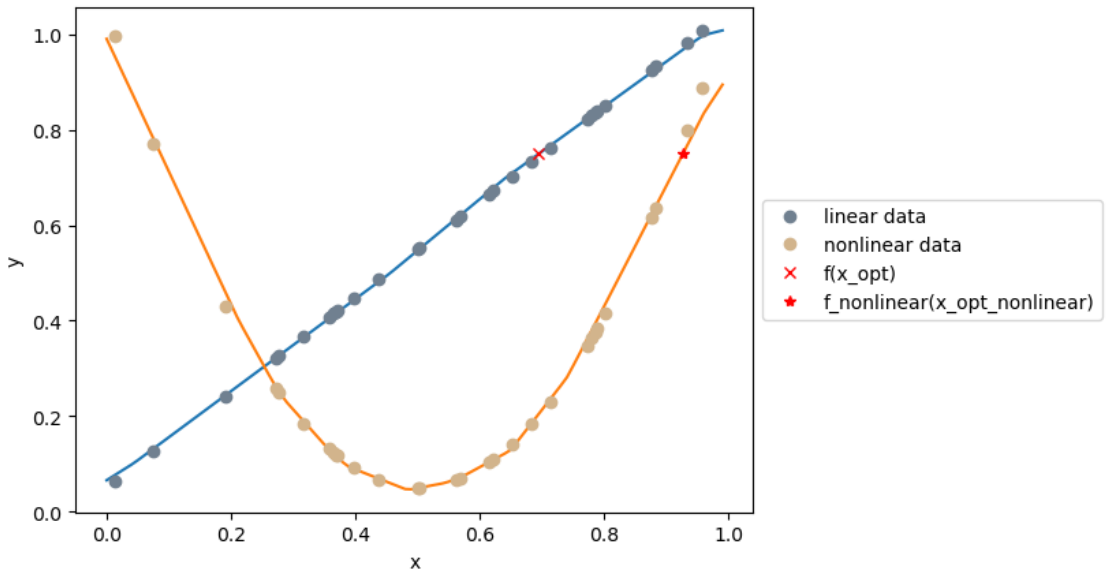
```
Linear solution: 0.694797190714189 0.7500000005504405 3.029847120938653e-19
Nonlinear solution: 0.9263573207190897 0.7500000008232756 6.777826457778079e-19
```

[9]:
```
# Line plot the output of the model
plot = PlotF(f, xmin=0.0, xmax=1.0, show=False)
plot_nonlinear = PlotF(f_nonlinear, xmin=0.0, xmax=1.0, show=False)

# Scatter plot the data samples
plot.plot(data_x, data_y, 'o', c='slategray', label='linear data')
plot.plot(data_x, data_y_nonlinear, 'o', c='tan', label='nonlinear data')

# Plot x_opt
plot.plot([x_opt], [f(x_opt)], 'x', c='red', label='f(x_opt)')
plot.plot([x_opt_nonlinear], [f_nonlinear(x_opt_nonlinear)], '*', c='red',␣
↪label='f_nonlinear(x_opt_nonlinear)')

# Show the labels, legends, and the figure
plot.xlabel('x')
plot.ylabel('y')
plot.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plot.show()
```



3.4 Learning Dynamics of Throwing Robot

Data that will be used to train the MLP model is obtained by manually inputting the independent variable x and assigning the first landing position as the dependent variable y. As we use a robot simulator, commands are needed to move the robot with the desired parameters. Here are the commands used to obtain the data:

```
Reset
MoveToTrg, 20
Wait, 2
MoveToTrg, 110, 100
Wait, 7
...
Reset
MoveToTrg, 30
Wait, 2
MoveToTrg, 100, 840
Wait, 7
```

Here, we set the torque value as the control parameter and vary it from 110 to 840, while the initial and target angles are set to be constant of 20° and 110° respectively. The reason we chose the aforementioned configuration is that after trial and error, the following conclusions were drawn:

- The initial angle can only be set to a value between $10 - 40$. If its lower, the ball will move a little bit to the upper part of the launcher, shifting ball's initial position. If its higher, the ball will be launched before it reaches the `Wait, 2` command. Thus, we set the initial angle to 20°.
- The target angle is set to 110° so that the robot arm covers at least a quarter of a circle and passes the 90°. There is no specific reason for setting this value; as long as it is not too low or not too high, the trajectory will be fine.
- The torque values have been adjusted to remove those that yield identical flying distances. Moreover, the interval is raised from 10 to 20 to observe how greater torque affects the trajectory.

Moreover, we stack several commands in a row to obtain many samples in one run. First, the robot will move to the initial angle; then it will wait for 2 seconds until it moves to the target angle with the stated torque. Here, we add an additional command `Wait, 7` to instruct the simulator to stay still for 7 seconds while we input the data to the excel file. Then, the robot will reset to its initial condition and do the same operations as before again and again until the end of the commands.

After obtaining the data, we scale the x and y values. This is done since these algorithms occasionally assume a range of values, in which learning and optimization will be impacted if x and y values are too high or too small.

[10]:
```
# Data obtained from the robot simulator
```

```
data_x = [100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200, 240, 280, 320,␣
↪360,
          400, 440, 480, 520, 560, 600, 640, 680, 720, 760, 800, 840] # Torque
data_y = [24.21888, 25.51657, 27.52911, 29.60749, 31.53964, 32.64215, 35.23895,␣
↪36.65649, 36.77268,
          38.07054, 42.09496, 41.95324, 45.77739, 46.68586, 46.01189, 49.48092,␣
↪47.90132, 50.36226,
          47.09709, 53.56269, 50.58345, 48.33881, 49.23299, 54.17227, 52.17052,␣
↪50.1124 , 50.11637] # Flying distance

# Scaling
data_x = np.array(data_x) * 0.001
data_y = np.array(data_y) * 0.01
```

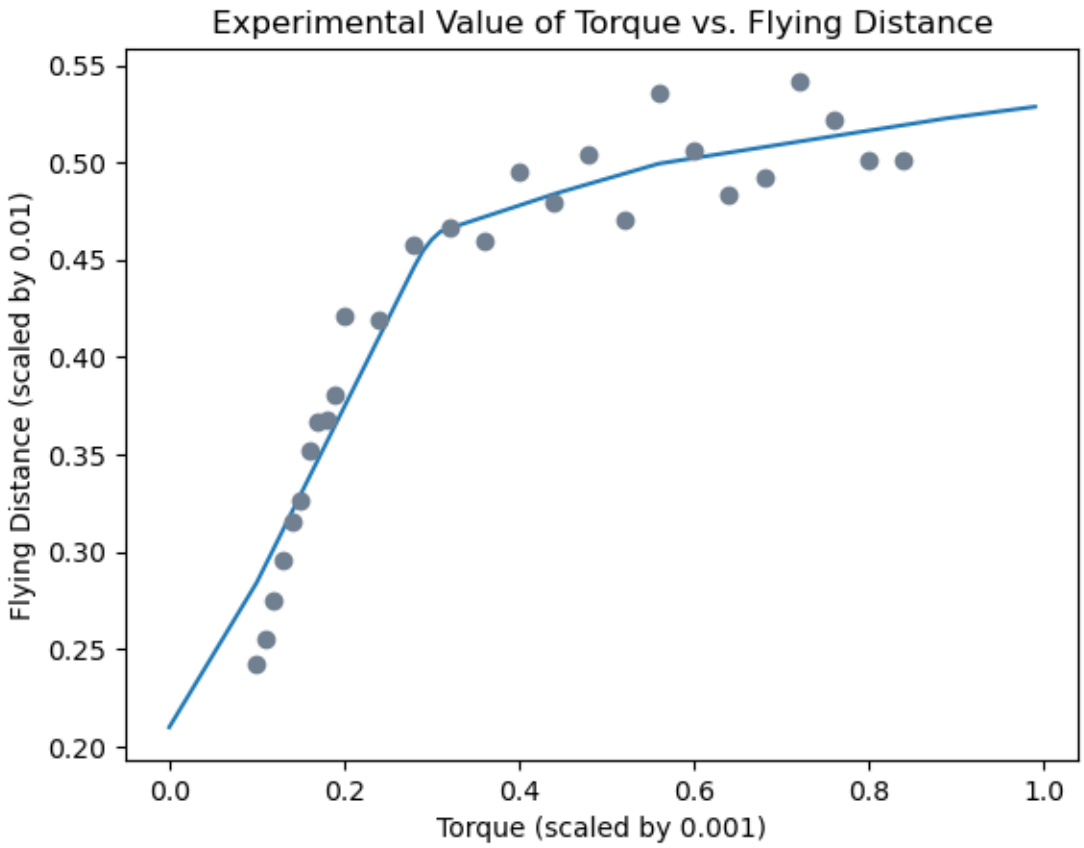
Using the acquired data, we train the MLP model and plot the generated function. The graph shows that the MLP model correctly fits the data (it is not underfitted or overfitted), implying that the function successfully learns the data. We can also see that the function for our calculated control variables is not linear (which makes sense since the trajectory is parabolic).

[11]:
```
# Training the MLP model
f = TrainMLPR(data_x, data_y)

#Line plot the output of the model
plot = PlotF(f, xmin=0.0, xmax=1.0, show=False)

# Scatter plot the data samples
plot.plot(data_x, data_y, 'o', c='slategray')

# Show the labels, legends, and the figure
plot.xlabel('Torque (scaled by 0.001)')
plot.ylabel('Flying Distance (scaled by 0.01)')
plot.title('Experimental Value of Torque vs. Flying Distance')
plot.show()
```



3.5 Control the Robot to Throw the Ball to a Target Location

Similar to section 4.2, we will optimize an objective function (square error function) to find x_{opt} that minimizes the objective function. Moreover, we also created an array of y_{trg} values ranging from 0.24 to 0.60 with an interval of 0.02 (24 m to 60 m with an interval of 2 m) by implementing *np.arange* function. After that, we create a dictionary where its keys-values contain the measurement variables and a list consisting of measurement values, respectively. Then, we start optimizing for each y_{trg} and record the obtained x_{opt} values to our dictionary.

[12]:
```
def f_error(x, y_trg):
    return np.square(f(x) - y_trg)

y_trgs = np.arange(0.240, 0.620, 0.020)

output = {'ytrgs':y_trgs, 'x_opts':[], 'f_x_opts':[], 'f_error_x_opts':[]}

for y_trg in y_trgs:
    x_opt = FMin(lambda x: f_error(x, y_trg), 0.5, 0.0, 5.0)
    f_x_opt = f(x_opt)
    f_error_x_opt = f_error(x_opt, y_trg)
```

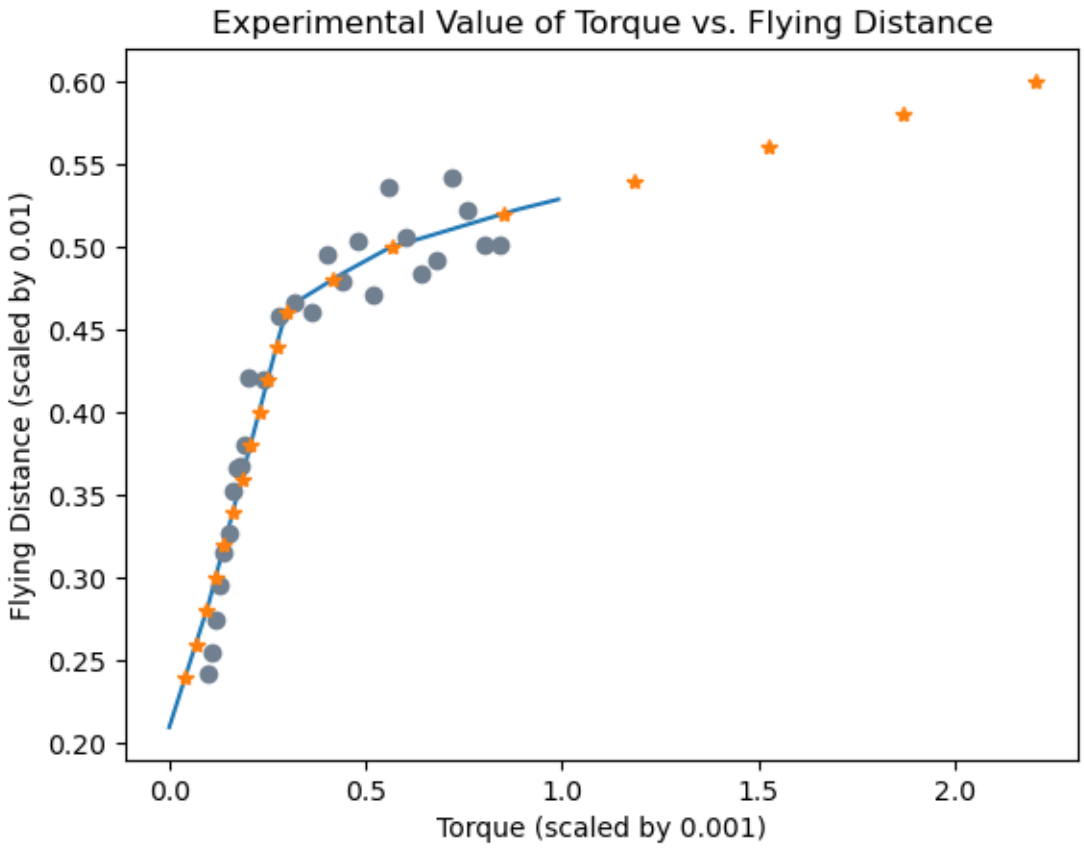
```
    output['x_opts'].append(x_opt)
    output['f_x_opts'].append(f_x_opt)
    output['f_error_x_opts'].append(f_error_x_opt)
```

Next, we use the *tabulate* library to print the dictionary created before in the form of a table and plot the estimated values. From the plot, the estimated x_{opt} values for small y_{trg} values are not accurate (underfitting) as their distances to the actual values are considerably large, considering the scale of the data. However, this does not happen to large y_{trg} value since the function does not overfit nor underfit our data. Moreover, we can see that the function is able to predict x_{opt} values outside the range of our x and y data (the line plot is not drawn due to our data range).

[13]:
```
print(tabulate(output, headers='keys', tablefmt='github'))

plot = PlotF(f, xmin=0.0, xmax=1.0, show=False)
plot.plot(data_x, data_y, 'o', c='slategray')
plot.plot(output['x_opts'], output['f_x_opts'], '*')
plot.xlabel('Torque (scaled by 0.001)')
plot.ylabel('Flying Distance (scaled by 0.01)')
plot.title('Experimental Value of Torque vs. Flying Distance')
plot.show()
```

ytrgs	x_opts	f_x_opts	f_error_x_opts
0.24	0.0406614	0.24	2.92812e-17
0.26	0.0675719	0.26	7.69901e-21
0.28	0.0944823	0.28	4.89291e-20
0.3	0.11765	0.3	2.68266e-17
0.32	0.139842	0.32	3.03876e-20
0.34	0.162035	0.34	6.99283e-17
0.36	0.184228	0.36	2.23959e-24
0.38	0.206421	0.38	2.90963e-19
0.4	0.228613	0.4	9.46586e-20
0.42	0.250806	0.42	1.71004e-19
0.44	0.272999	0.44	1.72719e-19
0.46	0.299479	0.46	7.03545e-17
0.48	0.414633	0.48	6.94235e-21
0.5	0.568767	0.5	2.45916e-18
0.52	0.85146	0.52	2.74855e-18
0.54	1.18419	0.54	7.11219e-19
0.56	1.52521	0.56	3.37046e-19
0.58	1.86623	0.58	1.70315e-20
0.6	2.20309	0.6	9.92462e-19



After that, we multiply the x_{opt} values by 1000 to obtain the torque values. Then we input these torque values into the robot simulator and record the flying distance using the same command as in previous experiments. In order to ease the comparison, we plot the MLP-estimated results and the true y values.

From the graph, we can see that for small y_{trg} values, the x_{opt} obtained from the trained-MLP and optimizer yields a really small flying distance value in the actual situation. On the other hand, when the y_{trg} values are large, the x_{opt} values tend to go up linearly while the true value reaches an asymptote value of around 50.11 m. Overall, the estimated values are not really accurate compared with the real flying distance.

Next, we calculate the Root Mean Squared Error (RMSE) metric as it is the standard method for deep learning techniques. The result of the RMSE value is considerably high. Several reasons can cause this high error:

- Limited data. The number of samples used to train the model is very small, considering the extensive range of possible y values. Like the usual deep learning model, a large amount of data is needed in order to properly train the model. The limited number of data also highlights the inability of our MLP model to predict the asymptote trend.
- The model's hyperparameter: activation function, optimizer function, number of layers, number of neurons in each layer, learning rate, momentum, and others.

- The scale number used to scale the data might not be the best one.

Lastly, we can improve the model's performance by addressing the above issues. This can be done by trial and error or using existing algorithms that can improve a model's efficiency and accuracy.

[14]:
```
# Rescaling the data to obtain the torque
torque = np.array(output['x_opts']) * 1000
y_ideal = np.array([10.8489, 18.85292, 22.97134, 26.86658, 31.23996, 35.99181,␣
 ↪41.32508, 44.21941, 46.17176,
                    46.18808, 44.28088, 46.84426, 47.7033, 52.74706, 50.1126, 50.
 ↪11637, 50.11638, 50.11226, 50.11239])

print(tabulate(zip(torque, y_ideal, y_trgs * 100), headers=['torque', 'y_ideal',␣
 ↪'y_target'], tablefmt='github'))
```

torque	y_ideal	y_target
40.6614	10.8489	24
67.5719	18.8529	26
94.4823	22.9713	28
117.65	26.8666	30
139.842	31.24	32
162.035	35.9918	34
184.228	41.3251	36
206.421	44.2194	38
228.613	46.1718	40
250.806	46.1881	42
272.999	44.2809	44
299.479	46.8443	46
414.633	47.7033	48
568.767	52.7471	50
851.46	50.1126	52
1184.19	50.1164	54
1525.21	50.1164	56
1866.23	50.1123	58
2203.09	50.1124	60

4 Discussion

In order to ease the comparison, we plot the MLP-estimated results and the true y values in the following figure.

From the graph, we can see that for small y_{trg} values, the x_{opt} obtained from the trained-MLP and optimizer yields a really small flying distance value in the actual situation. On the other hand, when the y_{trg} values are large, the x_{opt} values tend to go up linearly while the true value reaches an asymptote value of around 50.11 m. Overall, the estimated values are not really accurate compared with the real flying distance.

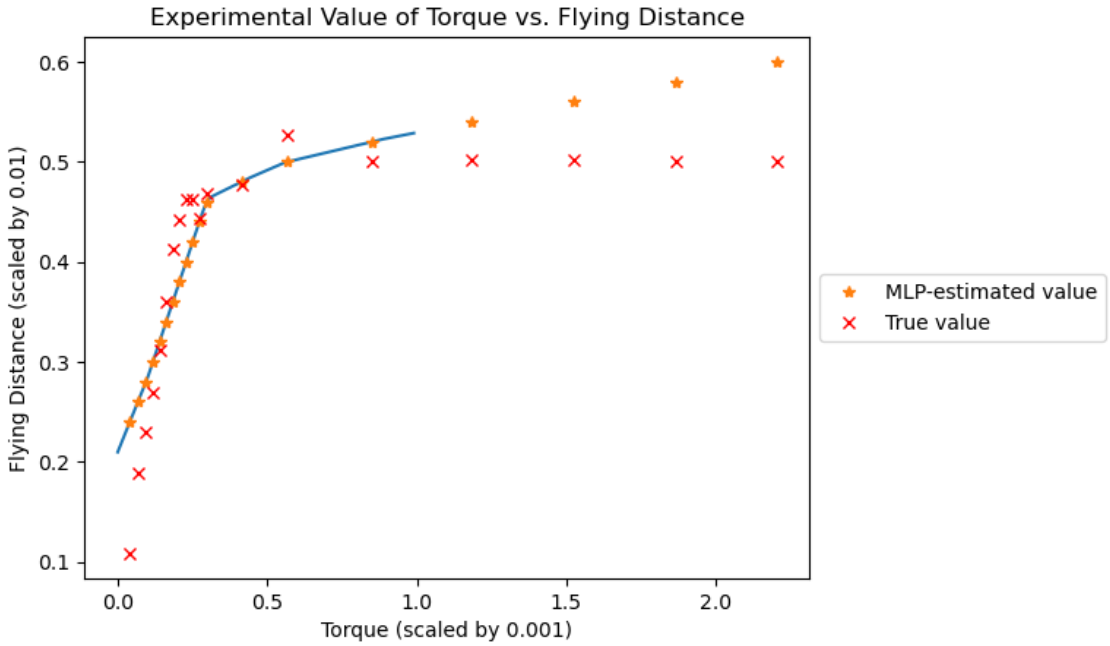
Next, we calculate the Root Mean Squared Error (RMSE) metric as it is the standard method for

deep learning techniques [7]. The result of the RMSE value is considerably high. Several reasons can cause this high error:

- Limited data. The number of samples used to train the model is very small, considering the extensive range of possible y values. Like the usual deep learning model, a large amount of data is needed in order to properly train the model. The limited number of data also highlights the inability of our MLP model to predict the asymptote trend.
- The model's hyperparameter: activation function, optimizer function, number of layers, number of neurons in each layer, learning rate, momentum, and others.
- The scale number used to scale the data might not be the best one.
- Others

Lastly, we can improve the model's performance by addressing the above issues. This can be done by trial and error or using existing algorithms that can improve a model's efficiency and accuracy.

[15]:
```
# Plot to make comparison easier
plot = PlotF(f, xmin=0.0, xmax=1.0, show=False)
plot.plot(output['x_opts'], output['f_x_opts'], '*', label='MLP-estimated value')
plot.plot(output['x_opts'], y_ideal * 0.01, 'x', c='r', label='True value')
plot.xlabel('Torque (scaled by 0.001)')
plot.ylabel('Flying Distance (scaled by 0.01)')
plot.title('Experimental Value of Torque vs. Flying Distance')
plot.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plot.show()
```



[16]:
```
RMSE = np.sqrt(np.mean((y_ideal - y_trgs * 100) ** 2))
print(f'Root Mean Square Error: {RMSE}')
```

```
Root Mean Square Error: 5.656948331877966
```

5 Conclusion

In this experiment, we learned how to utilize machine learning to control the robot's arm behaviour. We also discussed machine learning terms like the multi-layer perceptron model, regression, and optimization function. We train the model by creating a dataset consisting of a control parameter (torque) and the dependent variable (flying distance). After training the model, we evaluate the model's performance by observing the generated plot. Moreover, we also utilized the CMA-es algorithm to optimize an objective function, which allows us to obtain a torque value by inputting the desired flying distance to the objective function. However, the result is considered inaccurate, and several hypotheses were proposed that might explain the bad result.

6 Reference

[1] Machine Learning for Robot Control (Lab Experiment II) — Machine Learning for Robot Control (Lab Experiment II). http://www.ic.is.tohoku.ac.jp/%7Eswk/lecture/exp2/

[2] Zhang, A. (2022). Dive into Deep Learning: 5. Multilayer Perceptrons. https://d2l.ai/chapter_multilayer-perceptrons/index.html

[3] Hansen, N. (2016). The CMA evolution strategy: A tutorial. https://doi.org/10.48550/arXiv.1604.00772.

[4] Wikipedia contributors. (2022, November 22). CMA-ES. Wikipedia. https://en.wikipedia.org/wiki/CMA-ES

[5] NumPy documentation — NumPy v1.23 Manual. (n.d.). https://numpy.org/doc/stable/

[6] tabulate. (2022, October 6). PyPI. https://pypi.org/project/tabulate/

[7] Brownlee, J. (2021, January 2021). Regression Metrics for Machine Learning. https://machinelearningmastery.com/regression-metrics-for-machine-learning/

7 Addition

This part is only a little exploration that I did on how to utilize all available parameters (torque, initial angle, target angle) by implementing the cma package. In my opinion, adding some features to the contol parameters allow the model to learn more appropiately, resulting a better accuracy. Moreover, I also utilized more samples in this exploration as there are greater number of possible combinations compared to the case where we only tweak one independent variables.

[17]:
```
# Importing necessary packages
from sklearn.neural_network import MLPRegressor
import pandas as pd
from sklearn.model_selection import train_test_split
```

[19]:
```
sys.path.pop(-1)
# We download and install cma package from the internet as I have tried using␣
 ↪cma.py module in 'ai_ctrl_1'
# directory and it is not working for >=1 dimension parameter.
! pip install cma
import imp
imp.reload(cma)
```

[20]:
```
# Importing the data. The file 'Data.xlsx' is located in my local directory
df = pd.read_excel('Data.xlsx', index_col=None, header=0)
print(df.head())
df.shape
```

```
   Initial Angle  Final Angle  Torque  First Landing Position
0             30           80     100                11.01115
1             30           80     110                11.54333
2             30           80     120                12.13412
3             30           80     130                12.48524
4             30           80     140                12.06657
```

[20]:
```
(196, 4)
```

[21]:
```
# We scale the dataset then split it into two: training and test dataset
x = np.array([df.iloc[i, :3] for i in range(df.shape[0])]) * 0.001
print(f'x: {x}, {x.shape}')
y = np.array(df.iloc[:, 3]) * 0.01
print(f'y: {y}, {y.shape}')
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.05,␣
 ↪random_state=1234)
```

```
x: [[0.03 0.08 0.1 ]
 [0.03 0.08 0.11]
 [0.03 0.08 0.12]
 [0.03 0.08 0.13]
 ...
 [0.02 0.11 0.84]], (196, 3)
y: [0.1101115 0.1154333 0.1213412
 ...
 0.1248524 0.5011637], (196,)
```

[22]:
```
# We train the model. The model's hyperparamters is the same as the one used in␣
 ↪TrainMLPR
model = MLPRegressor(activation='relu', hidden_layer_sizes=(20, 20),␣
 ↪solver='adam', tol=1e-6, alpha=0.01, max_iter=200000, random_state=2)
model.fit(X_train, y_train)
```

[22]:
```
MLPRegressor(alpha=0.01, hidden_layer_sizes=(20, 20), max_iter=200000,
             random_state=2, tol=1e-06)
```

[23]:
```
# Defining the objective function
def f_error(x, y_trg):
    return np.square(model.predict(x) - y_trg)
```

[24]:
```
# The CMA-es algorithm to optimize the objective function. It is similar as FMin␣
 ↪in previous exercise,
# but it accepts multi-dimensional input
def FMin(f, x_init, x_min, x_max):
  options= {'bounds':[x_min, x_max], 'verb_log':0,'popsize':4}
  res = cma.fmin(lambda x:f(x), x_init, 0.01, options)
  return res
```

[25]:
```
# Start optimizing the objective function and obtain x_opt for each test y value.
x_init = np.array([0.03, 0.1, 0.15])
x_min = np.array([0.02, 0.08, 0.1])
x_max = np.array([0.04, 0.12, 0.2])
x_opts = [FMin(lambda x: f_error(x.reshape(1, -1), y_trg), x_init, x_min,␣
 ↪x_max)[0] for y_trg in y_test]
```

[26]:
```
# Defining Root Mean Square Error function between the prediction and the true␣
 ↪value.
def RMSE(y_predict, y_real):
  return np.sqrt(np.sum(np.square(y_predict - y_real)))
```

[27]:
```
# Print the error
error = RMSE(model.predict(x_opts), y_test.reshape(-1,1))
print(f'RMSE error: {error}')
```

```
RMSE error: 1.3417644148814405
```

As we can see, the error reduces from 5.65 to 1.34, which is quite significant. We can further improve the model's performance as already explained in the Discussion section.