Programming with Python

Latheeswara Rao Gedipudi

Written Assignment : DLMDSPWP01

Masters in Computer Sciences

Matriculation: 92120672

Tutor: Lino Antoni Giefer

Date: 31 July 2022

IU International University of

Applied Sciences

Table of Contents

I. List of Figures and Tables :

Introduction: Best approaches to Data operations

In this assignment, I'll be finding 4 ideal fuctions from the 50 ideal functions based on given 4 training functions, and then mapping the four chosen ideal functions to the test x-y pair along with deviations stored in the database table. I'll also be discussing about performing the data operations in python standard library, numpy library and panda library, then concluding the assignment with, which library is suitable for performing specific operations based on time complexity and space complexity, which is nothing but taking the time and memory taken for the data operation.

Time Complexity:

The term "time complexity" in computer science refers to the computational complexity that quantifies the length of time required to execute an algorithm. Counting the number of elementary operations the algorithm performs, assuming that each elementary operation takes a set amount of time to complete, is a standard method for estimating time complexity. As a result, it is assumed that the time required and the number of basic operations carried out by the algorithm are related by a constant factor.

One frequently takes into account the worst-case time complexity, which is the highest amount of time needed for inputs of a particular size, because an algorithm's running time may vary across several inputs of the same size.

Space complexity:

The amount of memory needed to solve a specific instance of the computational issue as a function of the input characteristics is known as the space complexity of an algorithm or computer programme. It is the amount of memory needed by an algorithm to finish its execution.

In many instances, Auxiliary Space is referred to as Space Complexity. The appropriate definitions of Auxiliary Space and Space Complexity are provided below.

The additional or temporary space used by an algorithm is known as auxiliary space.

The amount of space an algorithm uses overall in relation to the size of the input is known as space complexity. Auxiliary space and input space are both included in the definition of space complexity.

For instance, Auxiliary Space would be a better criterion than Space Complexity if we wanted to evaluate common sorting algorithms on the basis of space. Heap sort and Insertion sort both utilise O(1) auxiliary space, but Merge sort uses O(n) auxiliary space. However, all of these sorting algorithms have O(n) space complexity.

Raw information, or data, can exist in any form, whether it is useful or not. Data is readily available to us everywhere in life. Every day, data grows and becomes more varied. As a result, it is difficult to analyse and interpret data in a way that adds value or advances human understanding. Data scientists will need a wide range of abilities, including understanding of the relevant domains, as well as expertise in computer science, artificial intelligence (AI), machine learning (ML), statistics, and mathematics.

Data must first be collected and then processed or arranged for analysis. This procedure is sensitive to performance. Bases on how quickly we can process, how much memory it consumes. We should give considerable thought to this stage while developing a real solution that must analyse massive data.

Best approaches to Data operations:

Python standard library:

Over the past few years, Python has surpassed other languages in the field of data science, and its data structures are essential. There are many different data structures available in Python, but the two we'll be talking about today are array and list.

Two popular Python data structures that share many characteristics are arrays and lists. Both of them may be used to store data, and the indexing technique lets us iterate through, slice, and even access their components.

Even with ease of storing multiple datatypes, and it takes more memory consumption in in lists are more as additional space is dedicated during the initialization of the list.

.

```
%%timeit

size = 1000

l1 = range(size)

l2 = range(size)

result = [(x+y)**2 for x,y in zip(l1, l2)]
```

Here it takes 448 micro seconds to process the result, but takes 28,000 bytes of data.

Thus it holds its advantages and dis-advantages, based upon the requirement, we can use it as follows:

1. When you have distinct data types for your elements, you may store them in a list and access them by just looking at their indices.

2. You can change the size of a list. Consequently, a list is helpful if you are unsure about the amount of elements.

3. Lists are highly recommended when only a little amount of data needs to be saved, since their built-in functions make it easy to manipulate data.

Numpy Library:

The library known as NumPy, or "Numerical Python," contains multidimensional array objects and a selection of procedures for handling those arrays. Arrays can be subjected to logical and mathematical operations using NumPy. In the year 2005, Travis Oliphant developed NumPy. Which is available as a open source project.

Mathematical models of issues in Science and Engineering can be solved on a computer using a variety of tools and approaches are included in NumPy. And also it is a high-performance multidimensional array object, a potent data structure for quickly computing arrays and matrices, is one of these tools. There are numerous high-level mathematical operations that may be performed on these matrices and arrays in order to work with them.

And after performing the operation it took 11 micro seconds to perform the same operation as above.

```
 %%timeit

import numpy as np

size = 1000

a1 = np.arange(size)

a2 = np.arange(size)

result = (a1 + a2)**2
```

And it took 4000 bytes of memory allocation for storing the an array of 1000 elements, which is very very less than python standard library. Why so? , to explain this the below figure and explantion as to how the memory allocation system works in numpy, which is the main added advantage of being stored as an array, than in standard python libraries

As the object paradigm in Python might results in wasteful memory access.

Moving from a C integer to a Python integer exposes an additional type information layer. Imagine having a large number of these integers and wishing to perform a batch operation on them. While you might use the default List object in Python, you would probably use a buffer-based array in C.

In its most basic form, a NumPy array is a Python object created from a C array. It has a pointer to a values-only contiguous data buffer, in other words. In contrast, a Python list has a pointer to a contiguous buffer of pointers that each point to a Python object that in turn holds references to its contents (in this case, integers). Here is a possible schematic representation of the two:

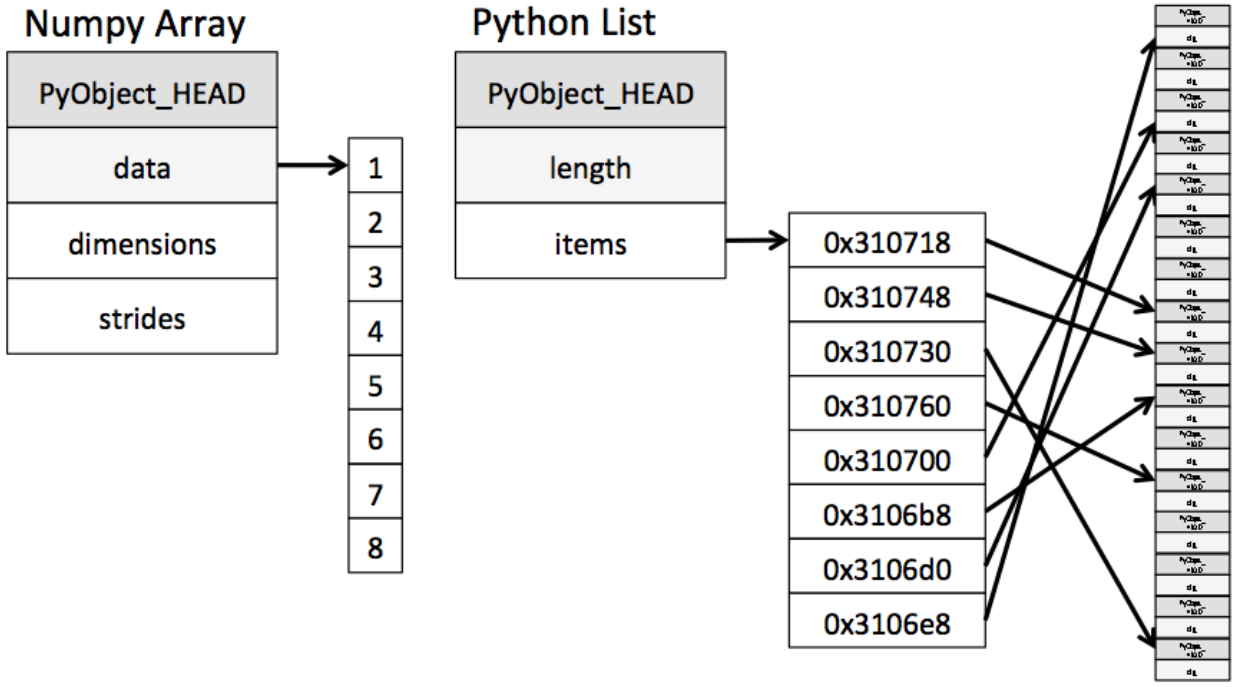

Figure: A

This figure represents how the memory is being allocated by segregation in numpy arrays, resulting in less memory consumption, faster data processing and operations, convenient.

Pandas:

Pandas have incredible strength. They give you access to a vast array of crucial features and instructions that are utilised to quickly evaluate your data. Pandas can be used to carry out a variety of activities, like filtering your data in accordance with specific criteria or segmenting and separating the data in accordance with preferences.

One of the best benefits of pandas is this. With the help of Pandas, what would have required several lines of Python code in the absence of any support libraries may be completed in just one or two lines. Thus, employing Pandas speeds up the data handling process. We may concentrate more on data analysis algorithms with the time saved.

Pandas offer incredibly efficient ways to represent data. This aids in improved data analysis and comprehension. Better outcomes for data science efforts are facilitated by simpler data representation.

But when it comes to data processing and operations, pandas is slower than python standard library and numpy combined.

Which is 2.29 milli seconds and 7900 bytes of memory taken for storing the same 1000 elements in a dataframe.

```
%%timeit

import pandas as pd

size = 1000

l1 = range(size)

l2 = range(size)

df_1 = pd.DataFrame(l1)

df_2 = pd.DataFrame(l2)

test = df_1.add(df_2)

test = test**2

print(test)
```

Pandas offers some tools for managing memory allocation, including: You can handle a dataset that is too large to fit in memory by employing a batch machine learning algorithm, which only processes a portion of the data at once. If you only need a sample of the data, using a batch technique makes sense. You can actually load the data in parts because of Python. Since the dataset flows continuously into a DataFrame or other data structure during this procedure, it is also known as data streaming.

Memory consumption:

To understand the memory consumption of standard python library , numpy and pandas. And the results were taken in the conclusion table.

```
import pandas as pds

import numpy as np

import timeit

import time

import sys


list = range(1000)

print(sys.getsizeof(1)*len(list))

array = np.arange(1000)

print(array.size * array.itemsize)

df = pds.DataFrame(list)

print(df.info(memory_usage='deep'))
```

Conclusion:

After approaching different libraries for a data operation, and got the result

	Memory(bytes)	Time_Taken
Python standard library	28000	448 micro seconds
Numpy	4000	11 micro seconds
Pandas	7900	2.29 milli seconds

Although Python's Standard Library utilities have the highest memory use, they performed better with the data operation but not better than numpy

The numpy layout will be significantly more efficient than the Python layout if you're performing an operation that steps over data in sequence, both in terms of the cost of storage and the cost of access.

Even though Pandas is generally took more time for the operations than any other library used in here, but took less memory than standard python library. Due to its extensive range of data operation methods, which we did not require for the current assignment, it was also simpler to use and may be of great value.

To summarize, numpy is a solid option for working with numerical data, and it also took lesser runtime and space complexity. Thus, it can be widely used for numerical and scientific computing than standard python library and panda, if you care about performance and space complexity

While pandas is the ideal solution for huge datasets, especially if they need a lot of preprocessing, such as filling empty values with defaults, because pandas handles memory so effectively,

List Of Abbreviations:

Random-access memory, or RAM. Regardless of the order (and hence location) in which they were recorded, certain contents can be accessed (read or written) immediately by the central processor unit in the main memory of a computer in a relatively short amount of time. Random-access circuits can be used to create static RAM (SRAM) and dynamic RAM, respectively (DRAM)

Machine learning, or ML, Computers are able to learn without being explicitly programmed because to this branch of study.

Artificial intelligence, or AI. When a system is created as a computer programme that autonomously decides on behalf of humans with regard to a task without being expressly programmed, the term artificial intelligence is used.

References:

1. Official Python Documentation. https://docs.python.org/3/library/

2. Official Python Documentation. https://docs.python.org/3/library/timeit.html

3. https://en.wikipedia.org/wiki/Time_complexity

4. https://en.wikipedia.org/wiki/Space_complexity

5. https://www.geeksforgeeks.org/g-fact-86/

6. https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html

7. https://medium.com/analytics-vidhya/introduction-to-numpy-279bbc88c615

8. https://data-flair.training/blogs/advantages-of-python-pandas/

9. http://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/

Appendix:

GitHub_link :- https://github.com/latheeswaraRao/latheeswaraRao

Main Code:

```
import pandas as pds


from utility.sqlite_utils import SQLLiteUtils


# establishing the local database connection

Conn_Conncetion_String = “sqlite:///data.db”


# CSV Path for ideal, train, test datasets

train_csv_path = “data_sets/train.csv”

ideal_csv_path = “data_sets/ideal.csv”

test_csv_path = “data_sets/test.csv”


def dataframe(filename):

    “””

    @return: Read CSV File and Convert to DF

    “””

    return pds.read_csv(filename)


def empty_dataframe():
```

```
    “””
    @return: Empty DataFrame
    “””
    return pds.DataFrame()


# Identifying the 4 ideal ideal_df from the 50 ideal_df
def ideal_function_finder(tr_df, idl_df):
    “””
    In this function we take the training and ideal function as an arguments
    And then the algorithm executes to find the ideal_df
    And returns the ideal ideal_df and maximum deviation
    “””
    # Changing the col indexes for compatibility
    tr_df.columns = tr_df.columns.str.replace(‘y’, ‘tr_y’)
    idl_df.columns = idl_df.columns.str.replace(‘y’, ‘idl_y’)

    # The both datasets are merged (trained data and ideal data) by using pds based on ‘x’.
    merged_dataframe = pds.merge(tr_df, idl_df, how=’inner’, right_on=’x’, left_on=’x’)

    # Creating empty dataframe for keeping final resulting values.
    Generate_ideal_func = empty_dataframe()
    generate_max = empty_dataframe()
```

```
# iterate the combined dataframe
column_ = [_col_ for _col_ in merged_dataframe.columns if ‘tr_’ in _col_]
# here empty df is used to extract the ideal_ideal_df, for temporary

for col_, i in enumerate(column_):

    temp_df = empty_dataframe()
    cols = [co_ for co_ in merged_dataframe.columns if ‘idl_’ in co_]
    for k in cols:
        temp_df[f”{k}_ls”] = (merged_dataframe[i] – merged_dataframe[k]) ** 2

    _win_ = str(temp_df.sum().idxmin()).split(“_”)[1]

    generate_ideal_func[[_win_]] = merged_dataframe[[“idl_” + _win_]]

    # Generating the max deviations and storing it n a data frame
    temp_df_max_value = temp_df[f”idl_{_win_}_ls”].max()
    generate_max[_win_] = [temp_df_max_value ** (0.50)]

generate_ideal_func.insert( loc=0, column=’x’, value = merged_dataframe[‘x’])

return {  ‘max’ : generate_max  , ‘ideal’: generate_ideal_func   }
```

```
def mapping_function(_test_df, ideal_, max_d):
    “””
    In mapping_function function, the test and ideal values are passed as the arguments
    And then the process begins for finding the mapping
    The condition is given below
    “””
    # The both datasets are merged (test data and ideal data) by using pds.
    _test_df[‘ideal_func’] = None
    merged_dataframe = _test_df.merge(ideal_, how=’left’, on=[‘x’])
    r1 = merged_dataframe.iterrows()

    for index, row in r1:

        # Assigning the float minimum and none every time
        min_delta_y = float(‘inf’)
        i_func = None
        r2 = max_d.T.iterrows()

        for _j, _row_ in r2:

            delta_y = abs(row[‘y’] – row[_j])
```

```
        # So to assign any targeted ideal_df to any test data point,
        # the delta value should be less than or equals when we perform max deviation by square(2) factor

        delta_bool = min_delta_y > delta_y
        delta_value = _row_[0] * (2 ** (0.50)) >= delta_y

        if delta_value and delta_bool:
            i_func = _j
            min_delta_y = delta_y


    _test_df.loc[index, 'ideal_func'] = i_func

    # Keeping Default value to None
    _test_df.at[index, 'delta_y'] = None
    _test_df.at[index, 'idl_y'] = None

    # Checking if min_delta_y is greater than inf.
    If float('inf') > min_delta_y:
        # if so reassign to min_delta_y, thus you assign the minimum values
       _test_df.at[index, 'delta_y'] = min_delta_y
```

```
        # Checking for i_func is exit. If so reassign to i_func
        if i_func:
            _test_df.at[index, 'idl_y'] = merged_dataframe[i_func][index]

    return _test_df


if __name__ == "__main__":
    # Connecting the database and initiating the connection
    sqlite_utils_ = SQLLiteUtils(db_str_conn=Conn_Conncetion_String)

    # Creating table_1: Loading the train dataset and sending to store in the database:
    sqlite_utils_.put_data(df=dataframe(train_csv_path), table_name='train',
db_str_conn=Conn_Conncetion_String)

    # Creating table_2: Loading the ideal dataset and sending to store in the database:
    sqlite_utils_.put_data(df=dataframe(ideal_csv_path), table_name='ideal',
db_str_conn=Conn_Conncetion_String)

    # Creating table_3: Loading the train dataset and sending to store in the database:
    sqlite_utils_.put_data(df=dataframe(test_csv_path), table_name='test',
db_str_conn=Conn_Conncetion_String)

    # identifying the ideal ideal_df and maximum deviations
```

```
    func_df = ideal_function_finder(dataframe(train_csv_path), dataframe(ideal_csv_path))


    # Mapping the ideal ideal_df to the data
    _mapping_ = mapping_function(dataframe(test_csv_path), func_df['ideal'], func_df['max'])


    # Final step : Final matching created along with mapping and deviations
    _mapping_ =   _mapping_[['x', 'y', 'delta_y', 'ideal_func']]
    print(f"Ideal Function: \n {func_df['ideal']}")
    print(f"Max Deviations: \n {func_df['max']}")


    sqlite_utils_.put_data(df=_mapping_, table_name='test_map',
db_str_conn=Conn_Conncetion_String)
```

Output:

Table 1:

	x	y1	y2	y3	y4
0	-20.0	17.820736	19.649096	-23995.145	289.27176
1	-19.9	16.901337	20.001790	-23636.730	285.71735
2	-19.8	15.903314	19.850693	-23282.309	282.06284
3	-19.7	14.707388	19.782515	-22931.451	278.97427
4	-19.6	13.092855	19.341375	-22583.979	276.01590

.. … … … … …

395 19.5 12.102674 19.967306 22249.402 506.30853

396 19.6 13.137722 19.168580 22593.998 510.98904

397 19.7 15.080357 19.592160 22940.775 515.29706

398 19.8 16.062035 19.661098 23291.934 519.42530

399 19.9 16.809252 19.503643 23646.380 524.47095

[400 rows x 5 columns]

Table 2:

	x	y1	y2	y3	…	y47	y48	y49	y50
0	-20.0	-0.912945	0.408082	9.087055	…	-5.298317	-0.186278	0.912945	0.396850
1	-19.9	-0.867644	0.497186	9.132356	…	-5.293305	-0.215690	0.867644	0.476954
2	-19.8	-0.813674	0.581322	9.186326	…	-5.288267	-0.236503	0.813674	0.549129
3	-19.7	-0.751573	0.659649	9.248426	…	-5.283204	-0.247887	0.751573	0.612840
4	-19.6	-0.681964	0.731386	9.318036	…	-5.278115	-0.249389	0.681964	0.667902
..	…	…	…	…	…	…	…	…	…
395	19.5	0.605540	0.795815	10.605540	…	-5.273000	0.240949	0.605540	0.714434
396	19.6	0.681964	0.731386	10.681964	…	-5.278115	0.249389	0.681964	0.667902
397	19.7	0.751573	0.659649	10.751574	…	-5.283204	0.247887	0.751573	0.612840
398	19.8	0.813674	0.581322	10.813674	…	-5.288267	0.236503	0.813674	0.549129
399	19.9	0.867644	0.497186	10.867644	…	-5.293305	0.215690	0.867644	0.476954

[400 rows x 51 columns]

Table 3:

	x	y
0	18.0	19.108044
1	12.4	-3.141064
2	-1.3	-0.049534
3	10.1	173.023530
4	-12.4	-2.871425
..	…	…
95	-18.9	17.519545
96	-12.2	11.691954
97	16.9	-16.938326
98	-7.4	8848.531000
99	-15.4	-10951.686000

[100 rows x 2 columns]

Ideal Functions: pair

y1 y10

y2 y31

y3 y25

y4 y20

Table 4: mapping

	x	y	delta_y	ideal_func
0	18.0	19.108044	NaN	None
1	12.4	-3.141064	NaN	None
2	-1.3	-0.049534	NaN	None
3	10.1	173.023530	NaN	None
4	-12.4	-2.871425	NaN	None
..	…	…	…	…
95	-18.9	17.51954	NaN	None
96	-12.2	11.69195	0.5080	y31
97	16.9	-16.938326	NaN	None
98	-7.4	8848.531	NaN	None
99	-15.4	-10951.68	0.1060	y25

[100 rows x 4 columns]

Figures:

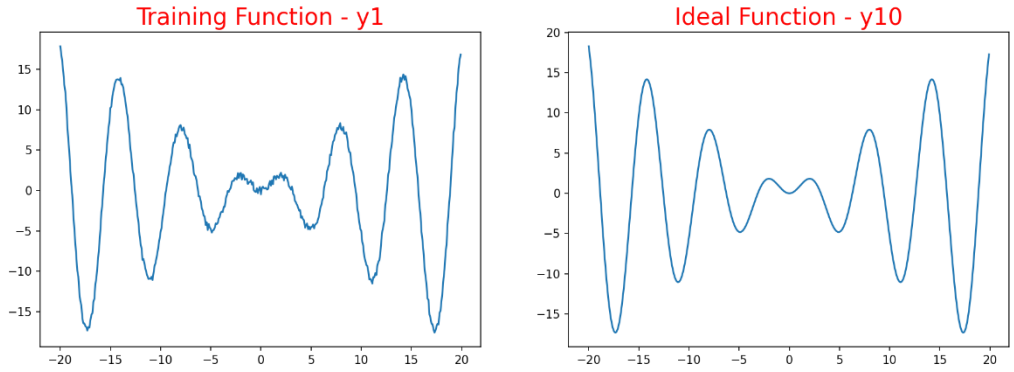

Figure 1

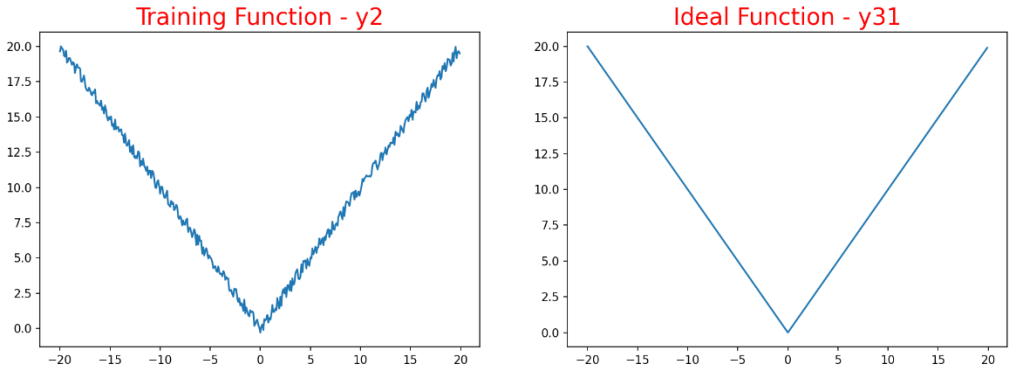

Figure 2

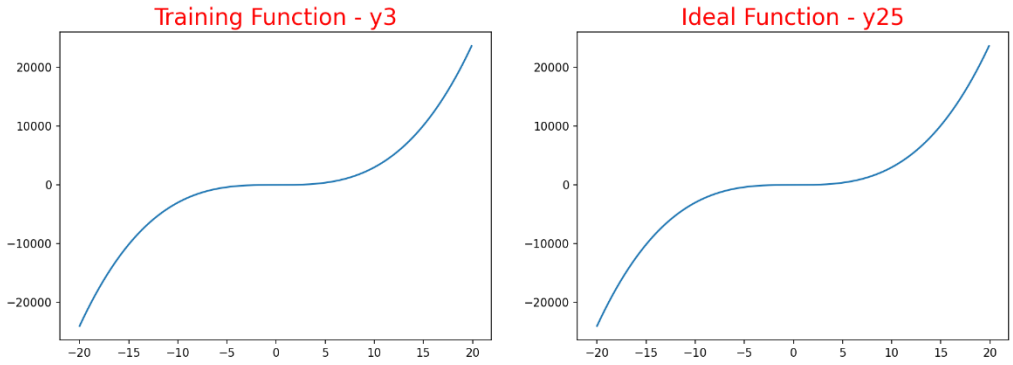

Figure 3

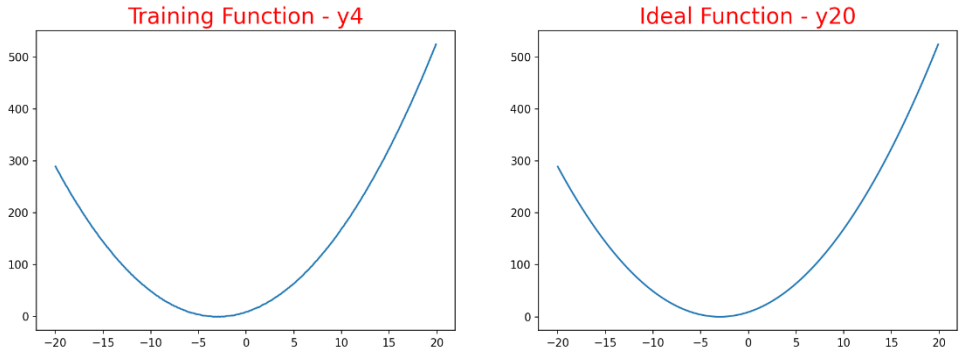

Figure 4