BME646 and ECE60146: Homework 1

Spring 2025
Due Date: Monday, Jan 20, 2025, 11:59pm
TA: Akshita Kamsali (akamsali@purdue.edu)

A policy document related to homework programming assignments and the submission of your solutions is being updated and will be posted at BrightSpace before the end of the first week of the semester. Please hold off on any policy related questions until then.

1 Introduction

This homework is designed to strengthen your understanding of Python Object-Oriented Programming (OOP), with a particular focus on its applications in PyTorch. It is the only assignment that focuses on general OOP principles.

Future assignments will involve the Python classes from the PyTorch platform and your homework will involve either using them or extending them in the object-oriented sense.

In this homework, you'll apply OOP concepts to model exponential growth related to a hypothetical exercise in population dynamics.

Note that you should use Python 3.x and NOT Python 2.x for this and all future programming assignments. For the Python-related knowledge required in this homework, refer to Prof. Kak's tutorial on OO Python [1].

2 Programming Tasks (100 points)

1. Create a class named `BioModel` with an instance variable named `sequence` as shown below:

```python
class BioModel(object):
    def __init__(self, sequence):
        self.sequence = sequence
```

   The input parameter `sequence` is expected to be a list of numbers, *e.g.* `[0, 1, 2]`. This class will serve as the base class for the subclasses later in this assignment.

2. Now, extend your `BioModel` class into a subclass called `ExponentialGrowthModel` with its `__init__` method defined with two input parameters: `start`

and `rate`. These two values will serve as the start and the rate of the growth model.

3. Further expand your `ExponentialGrowthModel` class to make its instances *callable*.

   More specifically, when an instance of ExponentialGrowthModel is called with a parameter named `length`, that should generate a sequence of length values based on the formula:

   $$\text{next value} = \text{current value} \times (1 + \text{rate})$$

   In addition, calling the instance should cause the computed sequence to be printed. Shown below is a demonstration of the expected behaviour described so far:

```
GM = ExponentialGrowthModel(start=100, rate=0.1)
GM(length=5)  # [100, 110, 121, 133.1, 146.41]
print(len(GM))  # 5
```

4. Modify your class definitions so that your `BioModel` instance can be used as an *iterator*. For example, when iterating through an instance of `ExponentialGrowthModel`, the numbers should be returned one-by-one.

   The snippet below illustrates the expected behavior:

```
GM = ExponentialGrowthModel(start=100, rate=0.1)
GM(length=5)  # [100, 110, 121, 133.1, 146.41]
print(len(GM))  # 5
print([n for n in GM])  # [100, 110, 121, 133.1, 146.41]
```

5. Now, we simualte a decay model. Make another subclass of the `BioModel` class named `ExponentialDecayModel`. As the name suggests, the new class is identical to `ExponentialGrowthModel` except that it is a decay model where rate will simply have a negative sign. When an instance of `ExponentialDecayModel` is called with a parameter named `length`, that should generate a sequence of length values based on the formula:

   $$\text{next value} = \text{current value} \times (1 - \text{rate})$$

   What is shown below illustrates the expected behavior:

```python
DM = ExponentialDecayModel(start=100, rate=0.2)
DM(length=5)  # [100, 80, 64, 51.2, 40.96]
print(len(DM))  # 5
print([n for n in DM])  # [100, 80, 64, 51.2, 40.96]
```

6. Modify `BioModel` to allow for a comparison between two instances using the `==` operator. If the sequences are of the same length, compare them element-wise and return the count of matching elements. Otherwise, raise a `ValueError`. Shown below is an example:

```python
GM = ExponentialGrowthModel(start=100, rate=0.1)
GM(length=5)  # [100, 110, 121, 133.1, 146.41]

GM2 = ExponentialGrowthModel(start=100, rate=0.2)
GM2(length=5)  # [100, 120, 144, 172.8, 207.36]

print(GM == GM2)  # 1

GM3 = ExponentialGrowthModel(start=100, rate=0.2)
GM3(length=3) # [100, 120, 144]

print(GM == GM3)  # will raise an error
# Traceback (most recent call last):
#    ...
#    ValueError: Two arrays are not equal in length!
```

   Show it for both the growth and the decay models.

7. Once you have finished all the tasks. Now, choose your own values for start and rate parameters and present the results.

3 Bonus (20 points)

1. **Combined Model (10 points):** Create a new subclass called `CombinedBioModel`. This class will combine the effects of growth and decay. It will:

   - Generate a growth sequence using `ExponentialGrowthModel`.
   - Generate a decay sequence using `ExponentialDecayModel`.
   - Combine these sequences by multiplying the corresponding values from both sequences element-wise.

```python
CBM = CombinedBioModel(growth_start=100, growth_rate=0.1,
                               decay_start=1.0,
                               decay_rate=0.05)
```

```
CBM(length=5) # [100.0, 104.50, 109.20, 114.12, 119.25]
```

2. **Visualization (10 points):**
   - Plot the generated growth, decay and combined sequences for two growth and two decay rates visualize growth and decay dynamics.
   - Ensure that all 4 plots have the same y-axis limits to allow for consistent comparison.
   - Clearly label each plot with a title that specifies the growth and decay rates used.
   - Add legends in each plot to indicate which line corresponds to growth and which corresponds to decay.
   - Use consistent colors for growth and decay lines across all plots to maintain clarity (e.g., blue for growth, red for decay).
   - Make sure the lines in the plots are clearly visible, and use line styles (solid, dashed, etc.) if needed to distinguish between growth and decay.
   - Ensure that all axes are labeled appropriately (x-axis for time or iteration, y-axis for value), and the legend should be easy to read.

4 Submission Instructions

Include a typed report explaining how you solved the given programming tasks. You may refer to the homework solutions posted at the class website for the previous years for examples of how to structure your report

1. Turn in a PDF file and mark all pages on **gradescope**. Rename .pdf file as hw1_<First Name><Last Name>.pdf

2. Submit your code files(s) as zip file. Rename the .zip file as hw1_<First Name><Last Name>.zip and follow the same file naming convention for your pdf report too. **Not adhering to the above naming convention will lead to you receiving an automatic zero for the homework.**

3. For this homework, you are encouraged to use `.ipynb` for development and the report. If you use `.ipynb`, please convert code to `.py` and submit that as source code. **Do NOT submit .ipynb notebooks.**

4. You can resubmit a homework assignment as many times as you want up to the deadline. Each submission will overwrite any previous submission. **If you are submitting late, do it only once.** Otherwise, we cannot guarantee that your latest submission will be pulled for grading and will not accept related regrade requests.

5. The sample solutions from previous years are for reference only. **Your code and final report must be your own work.**

6. Your pdf must include a description of

   - Outputs from your implementation for the parameter values in the snippet.
   - Outputs for each of the provided snippets above whith input parameters of your choice.
   - Your source code. Make sure that your source code files are adequately commented and cleaned up. You may refer to the homework solutions posted at the class website for the previous years for examples for reference.

References

[1] Python OO for DL. URL https://engineering.purdue.edu/DeepLearn/pdf-kak/PythonOO.pdf.