Sreeram Nagappa
snagapp@purdue.edu

hw1-report

January 16, 2024

1 Introduction

The goal of this homework is to improve your understanding of the Python Object-Oriented (OO) code in general, especially with regard to how it is used in PyTorch. This is the only homework you will get on general Python OO programming. Future homework assignments will be specific to using PyTorch classes directly or your own extensions of those classes for creating your DL solutions.

2 Task 1

[1]:
```
# Task 1

class Sequence(object):
    def __init__(self, array):
        self.array = array                # Variable to store the array
```

Task 1 asked to create a class named Sequence with an instance variable named array, which was done by having a class and a constructor that took in a parameter 'array' and stored it in a variable 'self.array'.

3 Task 2

[2]:
```
# Task 2

class Sequence(object):
    def __init__(self, array):
        self.array = array                # Variable to store the array

class Arithmetic(Sequence):
    def __init__(self, start, step):
        super().__init__(self)            # Super method to inherit Sequence␣
 ↪base class
        self.start = start                # Variable to store 'start' input␣
 ↪parameter
        self.step = step                  # Variable to store 'step' input␣
 ↪parameter
```

Task 2 asked to create a subclass named Arithmetic that extended from the Sequence class, which was done by having a class and a constructor that took in parameters 'start' and 'step' and stored in variables 'self.start' and 'self.step'. Also, in the constructor, the super method is used to inherit the Sequence class in the Arithmetic class and be able to access the variable 'self.array'.

4 Task 3

[3]:
```
# Task 3

class Sequence(object):
    def __init__(self, array):
        self.array = array                # Variable to store the array

class Arithmetic(Sequence):
    def __init__(self, start, step):
        super().__init__(self)            # Super method to inherit Sequence␣
 ↪base class
        self.start = start                # Variable to store 'start' input␣
 ↪parameter
        self.step = step                  # Variable to store 'step' input␣
 ↪parameter

    def __call__(self, length):
        self.array = [self.start + (self.step * i) for i in range(length)]      ␣
 ↪  # Create the arithmetic array based on inputs
        print(self.array)
```

[4]:
```
# Task 3 Required Outputs

AS = Arithmetic(start=1, step=2)
AS(length=5)
```

```
[1, 3, 5, 7, 9]
```

[5]:
```
# Task 3 Own Outputs

AS = Arithmetic(start=1, step=3)
AS(length=7)
```

```
[1, 4, 7, 10, 13, 16, 19]
```

Task 3 asked to make the Arithmetic class' instances callable, which was done by having a **call** method to invoke the creation of an arithmetic array using the 'self.start' and 'self.step' variables and an input parameter 'length'. The algorithm for the arithmetic array uses a for loop that executes a certain number of times based on the length and adds the step value each time to the previous calculated value beginning with the start value. Finally, the array is printed after the instance call, as shown in the required and own test cases above.

5 Task 4

[6]:
```
# Task 4

class Sequence(object):
    def __init__(self, array):
        self.array = array                # Variable to store the array

    def __len__(self):
        return len(self.array)            # Method to return length of the array

    def __iter__(self):
        self.index = 0                    # Variable to store iterator starting␣
 ↪at 0
        return self                       # Method to make Sequence be used as␣
 ↪an iterator

    def __next__(self):
        if self.index < len(self.array):
            result = self.array[self.index]
            self.index += 1
            return result                 # Method to iterate through the␣
 ↪Sequence instance
        else:
            raise StopIteration


class Arithmetic(Sequence):
    def __init__(self, start, step):
        super().__init__(self)            # Super method to inherit Sequence␣
 ↪base class
        self.start = start                # Variable to store 'start' input␣
 ↪parameter
        self.step = step                  # Variable to store 'step' input␣
 ↪parameter

    def __call__(self, length):
        self.array = [self.start + (self.step * i) for i in range(length)]      ␣
 ↪  # Create the arithmetic array based on inputs
        print(self.array)
```

[7]:
```
# Task 4 Required Outputs

AS = Arithmetic(start=1, step=2)
AS(length=5)
print(len(AS))
print([n for n in AS])
```

```
[1, 3, 5, 7, 9]
5
[1, 3, 5, 7, 9]
```

[8]:
```
# Task 4 Own Outputs

AS = Arithmetic(start=1, step=3)
AS(length=7)
print(len(AS))
print([n for n in AS])
```

```
[1, 4, 7, 10, 13, 16, 19]
7
[1, 4, 7, 10, 13, 16, 19]
```

Task 4 asked to make the Sequence class' instance be used as an iterator, which was done by having a **iter** method and a **next** method. In the **iter** method, a variable 'self.index' stored the initial value of 0 to be used as the iterator in the **next** method and returns self. In the **next** method, if the current index value is less than the length of the array, the value stored at that index of the array is returned and the 'self.index' value is incremented by one, else a 'StopIteration' exception is raised to prevent iterating when the end of the array is reached. Furthermore, to return the length of the array, a **len** method is created in the Sequence class. These methods are inherited by the subclasses. The expected length and iteration of values in the array are returned, as shown in the required and own test cases above.

6 Task 5

[9]:
```
# Task 5

class Sequence(object):
    def __init__(self, array):
        self.array = array                # Variable to store the array

    def __len__(self):
        return len(self.array)            # Method to return length of the array

    def __iter__(self):
        self.index = 0                    # Variable to store iterator starting␣
 ↪at 0
        return self                       # Method to make Sequence be used as␣
 ↪an iterator

    def __next__(self):
        if self.index < len(self.array):
            result = self.array[self.index]
            self.index += 1
```

```
            return result                # Method to iterate through the␣
 ↪Sequence instance
        else:
            raise StopIteration


class Arithmetic(Sequence):
    def __init__(self, start, step):
        super().__init__(self)           # Super method to inherit Sequence␣
 ↪base class
        self.start = start               # Variable to store 'start' input␣
 ↪parameter
        self.step = step                 # Variable to store 'step' input␣
 ↪parameter

    def __call__(self, length):
        self.array = [self.start + (self.step * i) for i in range(length)]     ␣
 ↪  # Create the arithmetic array based on inputs
        print(self.array)


class Geometric(Sequence):
    def __init__(self, start, ratio):
        super().__init__(self)           # Super method to inherit Sequence␣
 ↪base class
        self.start = start               # Variable to store 'start' input␣
 ↪parameter
        self.ratio = ratio               # Variable to store 'ratio' input␣
 ↪parameter

    def __call__(self, length):
        self.array = [self.start * (self.ratio ** i) for i in range(length)]   ␣
 ↪  # Create the geometric array based on inputs
        print(self.array)
```

[10]:
```
# Task 5 Required Outputs

GS = Geometric(start=1, ratio=2)
GS(length=8)
print(len(GS))
print([n for n in GS])
```

```
[1, 2, 4, 8, 16, 32, 64, 128]
8
[1, 2, 4, 8, 16, 32, 64, 128]
```

[11]:
```
# Task 5 Own Outputs

GS = Geometric(start=1, ratio=3)
GS(length=7)
print(len(GS))
print([n for n in GS])
```

```
[1, 3, 9, 27, 81, 243, 729]
7
[1, 3, 9, 27, 81, 243, 729]
```

Task 5 asked to create a subclass named Geometric that extended from the Sequence class, which was done by having a class and a constructor that took in parameters 'start' and 'ratio' and stored in variables 'self.start' and 'self.ratio'. Also, in the constructor, the super method is used to inherit the Sequence class in the Geometric class and be able to access the variable 'self.array'. Furthermore, Task 5 asked to make the Geometric class' instances callable, which was done by having a **call** method to invoke the creation of a geometric array using the 'self.start' and 'self.ratio' variables and an input parameter 'length'. The algorithm for the geometric array uses a for loop that executes a certain number of times based on the length and multiplies the ratio raised to the current iterator value to the previous calculated value beginning with the start value. Finally, the array is printed after the instance call, as shown in the required and own test cases above. In addition, Task 5 asked to make the Sequence class' instance be used as an iterator, which was done by having a **iter** method and a **next** method. In the **iter** method, a variable 'self.index' stored the initial value of 0 to be used as the iterator in the **next** method and returns self. In the **next** method, if the current index value is less than the length of the array, the value stored at that index of the array is returned and the 'self.index' value is incremented by one, else a 'StopIteration' exception is raised to prevent iterating when the end of the array is reached. Furthermore, to return the length of the array, a **len** method is created in the Sequence class. These methods are inherited by the subclasses. The expected length and iteration of values in the array are returned, as shown in the required and own test cases above.

7 Task 6

[12]:
```
# Task 6

class Sequence(object):
    def __init__(self, array):
        self.array = array                # Variable to store the array

    def __len__(self):
        return len(self.array)            # Method to return length of the array

    def __iter__(self):
        self.index = 0                    # Variable to store iterator starting␣
 ↪at 0
        return self                       # Method to make Sequence be used as␣
 ↪an iterator
```

```
    def __next__(self):
        if self.index < len(self.array):
            result = self.array[self.index]
            self.index += 1
            return result                # Method to iterate through the␣
↪Sequence instance
        else:
            raise StopIteration

    def __eq__(self, other):
        if len(self.array) != len(other.array):
            raise ValueError("Two arrays are not equal in length!")

        count = 0                        # Variable to store number of equal␣
↪elements
        for i in range(len(self.array)):
            if self.array[i] == other.array[i]:
                count += 1

        return count                     # Method to check if two sequence␣
↪instances are equal element-wise


class Arithmetic(Sequence):
    def __init__(self, start, step):
        super().__init__(self)           # Super method to inherit Sequence␣
↪base class
        self.start = start               # Variable to store 'start' input␣
↪parameter
        self.step = step                 # Variable to store 'step' input␣
↪parameter

    def __call__(self, length):
        self.array = [self.start + (self.step * i) for i in range(length)]     ␣
↪  # Create the arithmetic array based on inputs
        print(self.array)


class Geometric(Sequence):
    def __init__(self, start, ratio):
        super().__init__(self)           # Super method to inherit Sequence␣
↪base class
        self.start = start               # Variable to store 'start' input␣
↪parameter
```

```
        self.ratio = ratio                # Variable to store 'ratio' input␣
↪parameter

    def __call__(self, length):
        self.array = [self.start * (self.ratio ** i) for i in range(length)]  ␣
↪  # Create the geometric array based on inputs
        print(self.array)
```

[13]:
```
# Task 6 Required Outputs

AS = Arithmetic(start=1, step=2)
AS(length=5)
GS = Geometric(start=1, ratio=2)
GS(length=5)
print(AS == GS)

GS(length=8)
print(AS == GS)
```

```
[1, 3, 5, 7, 9]
[1, 2, 4, 8, 16]
1
[1, 2, 4, 8, 16, 32, 64, 128]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[13], line 10
      7 print(AS == GS)
      9 GS(length=8)
---> 10 print(AS == GS)

Cell In[12], line 24, in Sequence.__eq__(self, other)
     22 def __eq__(self, other):
     23     if len(self.array) != len(other.array):
---> 24         raise ValueError("Two arrays are not equal in length!")
     26     count = 0
     27     for i in range(len(self.array)):

ValueError: Two arrays are not equal in length!
```

[14]:
```
# Task 6 Own Outputs

AS = Arithmetic(start=2, step=3)
AS(length=7)
GS = Geometric(start=1, ratio=3)
GS(length=7)
```

```
print(AS == GS)

GS(length=4)
print(AS == GS)
```

```
[2, 5, 8, 11, 14, 17, 20]
[1, 3, 9, 27, 81, 243, 729]
0
[1, 3, 9, 27]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[14], line 10
      7 print(AS == GS)
      9 GS(length=4)
---> 10 print(AS == GS)

Cell In[12], line 24, in Sequence.__eq__(self, other)
     22 def __eq__(self, other):
     23     if len(self.array) != len(other.array):
---> 24         raise ValueError("Two arrays are not equal in length!")
     26     count = 0
     27     for i in range(len(self.array)):

ValueError: Two arrays are not equal in length!
```

Task 6 asked to make the Sequence class be able to check if two sequence instances are equal element-wise, which was done by having a **eq** method. The algorithm for this method first checks if the two sequences are equal in length, and if not, throw a 'ValueError'. If the sequences are equal, then initalize the variable 'count' and set it equal to 0, which will keep track of the number of equal elements. Then with a for loop, iterate through the length of the sequence and at each index, check if the value at that index in both sequences are equal and if so, increment count by one. At end, return the count. In the first test case, since 1 is the same at index 0 of both sequences and none of the other indexes have the same value, the output is 1. In the second test case, since the first sequence has a length of 5 and second sequence has a length of 8, a 'ValueError' is thrown. The newly created sequences and the number of equal elements are returned if the sequences are equal in length or a 'ValueError' is thrown if the sequences are not equal in length, as shown in the required and own test cases above.