Homework 1 - Deep Learning (ECE 60146)

Souradip Pal
PUID 0034772329

January 16, 2023

1 Introduction

In this homework, a general introduction to Python Object-Oriented(OO) concepts were provided and a series of programming tasks were performed using Python OO. The programming tasks were completed with the help of the ideas provided in the Python OO lecture presented by Prof. Kak.

2 Implementation and Results

2.1 Task 1

A **Sequence** class containing an instance variable *array* was created as follows:

```
class Sequence(object):
    def __init__(self, array):
        self.array = array
```

2.2 Task 2

Using the **Sequence** class as the base class, a new subclass called **Fibonacci** was created which takes two input parameters i.e. *first_value* and *second_value*. These two inputs serve as the first two numbers of the Fibonacci sequence. The constructor of the base class was called using a list of these two input parameters.

```
class Fibonacci(Sequence):
    def __init__(self, first_value, second_value):
        super(Fibonacci, self).__init__([first_value, second_value])
```

2.3 Task 3

To make the instances of the **Fibonacci** class *callable* with an input parameter *length*, the *__call__* method was modified. The method computes the Fibonacci sequence of that length, stores in the variable *array* and returns it.

```
def __call__(self, length):
    if not length is None and length >= 2:
        for i in range(length - 2):
            self.array.append(self.array[-1]+self.array[-2])
    else:
        raise ValueError('Invalid length.')
    return self.array
```

The results of the code snippet are shown below:

```
>> FS = Fibonacci(1, 2)
>> FS(length = 5)
>> [1, 2, 3, 5, 8]
```

2.4 Task 4

In this task, the **Sequence** class definition was modified to make its instances as an iterator. An *index* variable was added to keep track of the iteration index and pre-defiend *__next__* method was modifed. Also, the pre-defined methods *__iter__* and *__len__* methods were modified accordingly.

```
class Sequence(object):
    def __init__(self, array):
        self.array = array
        self.index = -1

    # Task 4: Modification to use Sequence instance as an iterator
    def __iter__(self):
        return self

    def __next__(self):
        self.index += 1
        if self.index < len(self.array):
            return self.array[self.index]
        else:
            raise StopIteration
    next = __next__

    def __len__(self):
        return len(self.array)
```

Running the code snippet in the task gives the following outputs:

```
>> FS = Fibonacci(first_value = 1, second_value = 2)
>> FS(length = 5)
>> [1, 2, 3, 5, 8]
>> print(len(FS))
>> 5
```

```
>> print([n for n in FS])
>> [1, 2, 3, 5, 8]
```

2.5 Task 5

Another subclass called **Prime** was created using the base **Sequence** class and its definition was modified to make its instances *callable* and used as an *iterator*. The *__call__* method take the *length* parameter as input and computes the consecutive prime numbers of that length.

```
class Prime(Sequence):
    def __init__(self):
        super(Prime, self).__init__([])

    # Task 5: Making the Prime class instances callable
    def __call__(self, length):
        self.__init__()
        if not length is None:
            if length == 1:
                self.array.append(2)

            if length == 2:
                self.array.append(2)
                self.array.append(3)

            if length > 2:
                self.array.append(2)
                self.array.append(3)
                n = 5
                while len(self.array) < length:
                    if not any([n % x == 0 for x in self.array]):
                        self.array.append(n)
                    n+=1;
        else:
            raise ValueError('Invalid length.')

        return self.array
```

Running the code snippet in the task gives the following outputs:

```
>> PS = Prime()
>> PS(length = 8)
>> [2, 3, 5, 7, 11, 13, 17, 19]
>> print(len(PS))
>> 8
>> print([n for n in PS])
>> [2, 3, 5, 7, 11, 13, 17, 19]
```

2.6 Task 6

In this task, the > operator was overloaded by modifying the pre-defined *__gt__* method to compare element-wise the two arrays of same length and return the number of elements in first array greater than the corresponding elements in the other array. If the lengths of the sequences compared are not same, then a *ValueError* was thrown.

```
def __gt__(self, other):
    if len(self.array) == len(other.array):
        gt = [x > other.array[i] for i,x in enumerate(self.array)]
        return gt.count(True)
    else:
        raise ValueError('Two arrays are not equal in length!')
```

Running the code snippet in the task gives the following outputs:

```
>> FS = Fibonacci(first_value = 1, second_value = 2)
>> FS(length = 8)
>> [1, 2, 3, 5, 8, 13, 21, 34]
>> PS = Prime()
>> PS(length = 8)
>> [2, 3, 5, 7, 11, 13, 17, 19]
>> print (FS > PS)
>> 2
>> PS(length = 5)
>> [2, 3, 5, 7, 11]
>> print(FS > PS)
```

```
ValueError                                Traceback (most recent call last)
<ipython-input-38-716a0b9fee04> in <module>
----> 1 print(FS > PS)

<ipython-input-24-ad6ff5627760> in __gt__(self, other)
     26             return gt.count(True)
     27         else:
---> 28             raise ValueError('Two arrays are not equal in length!')

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

3 Conclusion

In conclusion, Python OO is a powerful tool which provide a range of functionalities like callables, iterable, error handling, etc. to make deep-learning codes in Pytorch adhere to the software best practices.