Applied Deep Learning

Review of Some Python & NumPy Features

Jupyter (Ipython) Notebook/Lab

- Jupyter Notebook: open-source web application that allows create and share documents that contain live code, equations, visualizations and narrative text.
- May use it as an Python interpreter/runtime environment
- Jupyter Lab: a web-based interactive development environment for Jupyter notebooks, code, and data.
- Lab computers have notebook/lab installed through anaconda python

Mutable and Immutable Objects

- Everything is an objects
  - def foo(x): <-- function object
  - class bar: <-- class object
  - x = bar()   <-- instance object of bar
  - Everything is a dictionary (too). Use dir() to see its attributes.
- Some objects, e.g., integer, float, string, and tuple are "immutable."
  - Distinguish variable and the object referred to by the variable.
  - For immutable objects, we cannot change the object but we can redirect the variable to refer to a different object, e.g.:

```
x = 3
x += 1   # now x refers to the object with value 4
```

  - Dictionary keys must be immutable

Class Example

```
class Stack:
    def __init__(self):          # constructor
        self.items = []
    def push(self, x):
        self.items.append(x)
    def pop(self):
        x = self.items[-1]                  # what happens if it’s empty?
        del self.items[-1]
        return x
    def empty(self):
        return len(self.items) == 0     # Boolean result
```

Class and Subclass

```
class FancyStack(Stack):
  def __init__(self):
     self.name='FancyStack'
     super().__init__()

  def peek(self, n):
     size = len(self.items)
     assert 0 <= n < size                          # test precondition
     return self.items[size-1-n]

  def pop(self):                          # override base/parent class method
     x = self.items[0]
     del self.items[0]
     return x
```

Interface Class and Abstract Method

```
from abc import ABC, abstractmethod

class BaseModel(ABC):
    @abstractmethod
    def train(self, x):
        pass

m = BaseModel()

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-53f55fa572ca> in <module>
      6         pass
      7
----> 8 m = BaseModel()

TypeError: Can't instantiate abstract class BaseModel with abstract methods train
```

Enforcing override

Callable

- A callable is anything you can call, using parenthesis, and possibly passing arguments.
  - Functions are callables
  - We *call* a class to create a new instance of the class
  - Class methods are callables
  - Instances of classes can be callables
- Which is a function and which is a class (object)?
  - zip,
  - len,
  - int
- Callable object
  - Instance of a class that implements __call__ method

```
class MyFunc:
    def __call__(self, x):
        return x*x

square = MyFunc()

print(square(4))
```

`callable(MyFunc)`	True/False
`callable(square)`	True/False
`callable(5)`	True/False

In Python, everything is an object, even functions

The `for` loop

- **`for` loop**: Repeats a set of statements over a group of values.
  - Syntax:

    for ***variableName*** in ***groupOfValues***:
        statements

    - We indent the statements to be repeated with tabs or spaces.
    - ***variableName*** gives a name to each value, so you can refer to it in the ***statements***.
    - ***groupOfValues*** can be a range of integers, specified with the `range` function.
  - Example:

```
for x in range(1, 6):
    print x, "squared is", x * x
```

Output:

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
```

`range`

- The `range` function specifies a range of integers:
  - `range(`***start***`,` ***stop***`)`

    \- the integers between ***start*** (inclusive) and ***stop*** (exclusive)

- It can also accept a third value specifying the change between values.
  - `range(`***start***`,` ***stop***`,` ***step***`)` - the integers between ***start*** (inclusive) and ***stop*** (exclusive) by ***step***

- Example:

```
for x in range(5, 0, -1):
    print(x)
```

Output:

```
5
4
3
2
1
```

Iterating Loop

- Looping by iterating an iterable:

```
for x in [1, 2, 3]:
    print(x)
```

```
x = iter([1, 2, 3])
while True:
    try:
        print(next(x))
    except StopIteration:
        break
```

Iterator

- An iterator is an object that can be iterated upon, i.e., you can traverse through all the values hold by the object.
- Often used in for-loop
- An iterator object implements the iterator protocol, which consist of the methods
  - __iter__() sets up the iteration
  - __next__() each call to this function returns one next value
  - raise StopIteration to stop iteration

```
class NumberIter:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    if self.a < 11:
        x = self.a
        self.a += 1
        return x
    else:
        raise StopIteration
```

```
>>> ni = NumberIter()
>>> for i in ni:
        print(i)

1
2
3
4
5
6
7
8
9
10
```

Generator

- Generator functions allow you to declare a function that behaves like an iterator, i.e. it can be used in a for loop.
- Simplify the common pattern, no need to define
  - __iter__()
  - __next__()

```
def NumberGen():
    a = 1
    while a < 11:
        yield a
        a += 1
```

```
>>> for i in NumberGen():
        print(i)

1
2
3
4
5
6
7
8
9
10
```

Python is lazy!

List vs Iterator/Generator

- We can do

```
for i in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    print(i)
```

- Why do we need iterator/generator after all?
  [Hint: Python is *lazy*!!]

Lazy evaluation can be Good

- Lazy evaluation is an evaluation strategy which delays the evaluation of an expression until its value is needed.
- Compute/construct/fetch the object when it is needed.
- In our case, saving memory space.

List Comprehension

```
[expression for element in list]
```

- Applies the expression to each element in the list
- You can have 0 or more for or if statements
- If the expression evaluates to a tuple it must be in parenthesis

List Comprehension (2)

```
vec = [2, 4, 6]
>>> [3*x for x in vec]
    [6, 12, 18]

>>> [3*x for x in vec if x > 3]
    [12, 18]

>>> [3*x for x in vec if x < 2]
    []

>>> [[x,x**2] for x in vec]
    [[2, 4], [4, 16], [6, 36]]

>>> [x, x**2 for x in vec]
    # error - parens required for tuples
```

Slicing: Return Copy of a Subsequence

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

- Start copying at the first index, and stop copying before the second index.

```
>>> t[1:4]
('abc', 4.56, (2,3))
```

- can also use negative indices

```
>>> t[1:-1]
('abc', 4.56, (2,3))
```

- Omit the first index to make a copy starting from the beginning

```
>>> t[:2]
(23, 'abc')
```

- Omit the second index to make a copy starting at the first index and going to the end

```
>>> t[2:]
(4.56, (2,3), 'def')
```

Numpy indexing

- Single-dimension indexing is accomplished as usual.

```
>>> x = np.arange(10)
>>> x[2]
2
>>> x[-2]
8
```

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{bmatrix}$$

- Multi-dimensional arrays support multi-dimensional indexing.

```
>>> x.shape = (2,5) # now x is 2-dimensional
>>> x[1,3]
8
>>> x[1,-1]
9
```

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 & 9 \end{bmatrix}$$

Numpy Indexing (2)

- Using fewer dimensions to index will result in a subarray.

```
>>> x[0]
array([0, 1, 2, 3, 4])
```

- This means that `x[i, j]` is the same as `x[i][j]` but the second method is less efficient.

Numpy Indexing (3)

- Slicing is possible just as it is for typical Python sequences.

```
>>> x = np.arange(10)
>>> x[2:5]
array([2, 3, 4])
>>> x[:-7]
array([0, 1, 2])
>>> x[1:7:2]
array([1, 3, 5])
>>> y = np.arange(35).reshape(5,7)
>>> y[1:5:2,::3]
array([[ 7, 10, 13], [21, 24, 27]])
```

Numpy Indexing (4)

- Multi-dimensional index and slicing

```
>>> y = np.arange(35).reshape(5,7)
>>> y
array([[ 0,  1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12, 13],
       [14, 15, 16, 17, 18, 19, 20],
       [21, 22, 23, 24, 25, 26, 27],
       [28, 29, 30, 31, 32, 33, 34]])

>>> y[np.array([0,2,4]), np.array([0,1,2])]
array([ 0, 15, 30])

>>> y[np.array([0,2,4]), 1]
array([ 1, 15, 29])

>>> y[np.array([0,2,4]), 1:3]
array([[ 1,  2],
       [15, 16],
       [29, 30]])
```

Broadcasting

```
a = a + 1 # add one to every element
```

When operating on multiple arrays, broadcasting rules are used.

Each dimension must match, from right-to-left

1. Dimensions of size 1 will broadcast (as if the value was repeated).
2. Otherwise, the dimension must have the same shape.
3. Extra dimensions of size 1 are added to the left as needed.

Broadcasting example

Suppose we want to add a color value to an image

a.shape is 100, 200, 3

b.shape is 3

a + b will pad b with two extra dimensions so it has an effective shape of 1 x 1 x 3.

So, the addition will broadcast over the first and second dimensions.

Broadcasting failures

If a.shape is 100, 200, 3 but b.shape is 4 then a + b will fail. The trailing dimensions must have the same shape (or be 1)