Open in Colab

T81-558: Applications of Deep Neural Networks

Module 1: Python Preliminaries

- Instructor: Jeff Heaton, McKelvey School of Engineering, Washington University in St. Louis
- For more information visit the class website.

Module 1 Material

- Part 1.1: Course Overview [Video] [Notebook]
- Part 1.2: Introduction to Python [Video] [Notebook]
- Part 1.3: Python Lists, Dictionaries, Sets and JSON [Video] [Notebook]
- Part 1.4: File Handling [Video] [Notebook]
- **Part 1.5: Functions, Lambdas, and Map/Reduce** [Video] [Notebook]

Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

In [1]:
```
try:
    from google.colab import drive
    %tensorflow_version 2.x
    COLAB = True
    print("Note: using Google CoLab")
except:
    print("Note: not using Google CoLab")
    COLAB = False
```

Part 1.5: Functions, Lambdas, and Map/Reduce

Functions, **lambdas**, and **map/reduce** can allow you to process your data in advanced ways. We will introduce these techniques here and expand on them in the next module, which will discuss Pandas.

Function parameters can be named or unnamed in Python. Default values can also be used. Consider the following function.

In [2]:
```
def say_hello(speaker, person_to_greet, greeting = "Hello"):
    print(f'{greeting} {person_to_greet}, this is {speaker}.')

say_hello('Jeff', "John")
say_hello('Jeff', "John", "Goodbye")
say_hello(speaker='Jeff', person_to_greet="John", greeting = "Goodbye")
```

```
Hello John, this is Jeff.
Goodbye John, this is Jeff.
Goodbye John, this is Jeff.
```

A function is a way to capture code that is commonly executed. Consider the following function that can be used to trim white space from a string capitalize the first letter.

In [3]:
```
def process_string(str):
    t = str.strip()
    return t[0].upper()+t[1:]
```

This function can now be called quite easily.

In [4]:
```
str = process_string("  hello  ")
print(f'"{str}"')
```

```
"Hello"
```

Python's **map** is a very useful function that is provided in many different programming languages. The **map** function takes a **list** and applies a function to each member of the **list** and returns a second **list** that is the same size as the first.

In [5]:
```
l = ['   apple  ', 'pear ', 'orange', 'pine apple  ']
list(map(process_string, l))
```

Out[5]:
```
['Apple', 'Pear', 'Orange', 'Pine apple']
```

Map

The **map** function is very similar to the Python **comprehension** that we previously explored. The following **comprehension** accomplishes the same task as the previous call to **map**.

In [6]:
```
l = ['   apple  ', 'pear ', 'orange', 'pine apple  ']
l2 = [process_string(x) for x in l]
print(l2)
```

```
['Apple', 'Pear', 'Orange', 'Pine apple']
```

The choice of using a **map** function or **comprehension** is up to the programmer. I tend to prefer **map** since it is so common in other programming languages.

Filter

While a **map function** always creates a new **list** of the same size as the original, the **filter** function creates a potentially smaller **list**.

In [7]:
```
def greater_than_five(x):
    return x>5

l = [ 1, 10, 20, 3, -2, 0]
l2 = list(filter(greater_than_five, l))
print(l2)
```

```
[10, 20]
```

Lambda

It might seem somewhat tedious to have to create an entire function just to check to see if a value is greater than 5. A **lambda** saves you this effort. A lambda is essentially an unnamed function.

In [8]:
```
l = [ 1, 10, 20, 3, -2, 0]
l2 = list(filter(lambda x: x>5, l))
print(l2)
```

```
[10, 20]
```

Reduce

Finally, we will make use of **reduce**. Like **filter** and **map** the **reduce** function also works on a **list**. However, the result of the **reduce** is a single value. Consider if you wanted to sum the **values** of a **list**. The sum is implemented by a **lambda**.

In [9]:
```
from functools import reduce

l = [ 1, 10, 20, 3, -2, 0]
result = reduce(lambda x,y: x+y,l)
print(result)
```

```
32
```