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
```

```
Note: not using Google CoLab
```

Part 1.3: Python Lists, Dictionaries, Sets, and JSON

Like most modern programming languages, Python includes Lists, Sets, Dictionaries, and other data structures as built-in types. The syntax appearance

of both of these is similar to JSON. Python and JSON compatibility is discussed later in this module. This course will focus primarily on Lists, Sets, and Dictionaries. It is essential to understand the differences between these three fundamental collection types.

- **Dictionary** - A dictionary is a mutable unordered collection that Python indexes with name and value pairs.
- **List** - A list is a mutable ordered collection that allows duplicate elements.
- **Set** - A set is a mutable unordered collection with no duplicate elements.
- **Tuple** - A tuple is an immutable ordered collection that allows duplicate elements.

Most Python collections are mutable, meaning the program can add and remove elements after definition. An immutable collection cannot add or remove items after definition. It is also essential to understand that an ordered collection means that items maintain their order as the program adds them to a collection. This order might not be any specific ordering, such as alphabetic or numeric.

Lists and tuples are very similar in Python and are often confused. The significant difference is that a list is mutable, but a tuple isn't. So, we include a list when we want to contain similar items and a tuple when we know what information goes into it ahead of time.

Many programming languages contain a data collection called an array. The array type is noticeably absent in Python. Generally, the programmer will use a list in place of an array in Python. Arrays in most programming languages were fixed-length, requiring the program to know the maximum number of elements needed ahead of time. This restriction leads to the infamous array-overrun bugs and security issues. The Python list is much more flexible in that the program can dynamically change the size of a list.

The next sections will look at each collection type in more detail.

Lists and Tuples

For a Python program, lists and tuples are very similar. Both lists and tuples hold an ordered collection of items. It is possible to get by as a programmer using only lists and ignoring tuples.

The primary difference that you will see syntactically is that a list is enclosed by square braces [], and a tuple is enclosed by parenthesis (). The following code defines both list and tuple.

In [1]:
```
l = ['a', 'b', 'c', 'd']
t = ('a', 'b', 'c', 'd')
```

```
print(l)
print(t)
```

```
['a', 'b', 'c', 'd']
('a', 'b', 'c', 'd')
```

The primary difference you will see programmatically is that a list is mutable, which means the program can change it. A tuple is immutable, which means the program cannot change it. The following code demonstrates that the program can change a list. This code also illustrates that Python indexes lists starting at element 0. Accessing element one modifies the second element in the collection. One advantage of tuples over lists is that tuples are generally slightly faster to iterate over than lists.

In [2]:
```
l[1] = 'chaged'
#t[1] = 'changed' # This would result in an error

print(l)
```

```
['a', 'chaged', 'c', 'd']
```

Like many languages, Python has a for-each statement. This statement allows you to loop over every element in a collection, such as a list or a tuple.

In [4]:
```
# Iterate over a collection.
for s in l:
    print(s)
```

```
a
changed
c
d
```

The **enumerate** function is useful for enumerating over a collection and having access to the index of the element that we are currently on.

In [5]:
```
# Iterate over a collection, and know where your index.  (Python is zero-bas
for i,l in enumerate(l):
    print(f"{i}:{l}")
```

```
0:a
1:changed
2:c
3:d
```

A **list** can have multiple objects added, such as strings. Duplicate values are allowed. **Tuples** do not allow the program to add additional objects after definition.

In [6]:
```
# Manually add items, lists allow duplicates
c = []
c.append('a')
c.append('b')
```

```
c.append('c')
c.append('c')
print(c)
```

```
['a', 'b', 'c', 'c']
```

Ordered collections, such as lists and tuples, allow you to access an element by its index number, as done in the following code. Unordered collections, such as dictionaries and sets, do not allow the program to access them in this way.

In [7]:
```
print(c[1])
```

```
b
```

A **list** can have multiple objects added, such as strings. Duplicate values are allowed. Tuples do not allow the program to add additional objects after definition. The programmer must specify an index for the insert function, an index. These operations are not allowed for tuples because they would result in a change.

In [8]:
```
# Insert
c = ['a', 'b', 'c']
c.insert(0, 'a0')
print(c)
# Remove
c.remove('b')
print(c)
# Remove at index
del c[0]
print(c)
```

```
['a0', 'a', 'b', 'c']
['a0', 'a', 'c']
['a', 'c']
```

Sets

A Python **set** holds an unordered collection of objects, but sets do *not* allow duplicates. If a program adds a duplicate item to a set, only one copy of each item remains in the collection. Adding a duplicate item to a set does not result in an error. Any of the following techniques will define a set.

In [9]:
```
s = set()
s = { 'a', 'b', 'c'}
s = set(['a', 'b', 'c'])
print(s)
```

```
{'c', 'a', 'b'}
```

A **list** is always enclosed in square braces [], a **tuple** in parenthesis (), and similarly a **set** is enclosed in curly braces. Programs can add items to a **set** as they run. Programs can dynamically add items to a **set** with the **add** function. It

is important to note that the **append** function adds items to lists, whereas the **add** function adds items to a **set**.

In [5]:
```
# Manually add items, sets do not allow duplicates
# Sets add, lists append.  I find this annoying.
c = set()
c.add('a')
c.add('b')
c.add('c')
c.add('c')
c.add('gc')
print(c)
```

```
{'a', 'b', 'gc', 'c'}
```

Maps/Dictionaries/Hash Tables

Many programming languages include the concept of a map, dictionary, or hash table. These are all very related concepts. Python provides a dictionary that is essentially a collection of name-value pairs. Programs define dictionaries using curly braces, as seen here.

In [8]:
```
d = {'name': "Jeff", 'address':"123 Main"}
print(d)
print(d['name'])

if 'name' in d:
    print("Name is defined")
    print(d['name'])
    print(d['address'])

if 'age' in d:
    print("age defined")
else:
    print("age undefined")
```

```
{'name': 'Jeff', 'address': '123 Main'}
Jeff
Name is defined
Jeff
123 Main
age undefined
```

Be careful that you do not attempt to access an undefined key, as this will result in an error. You can check to see if a key is defined, as demonstrated above. You can also access the dictionary and provide a default value, as the following code demonstrates.

In [12]:
```
d.get('unknown_key', 'default')
```

Out[12]:
```
'default'
```

You can also access the individual keys and values of a dictionary.

In [13]:
```
d = {'name': "Jeff", 'address':"123 Main"}
# All of the keys
print(f"Key: {d.keys()}")

# All of the values
print(f"Values: {d.values()}")
```

```
Key: dict_keys(['name', 'address'])
Values: dict_values(['Jeff', '123 Main'])
```

Dictionaries and lists can be combined. This syntax is closely related to JSON. Dictionaries and lists together are a good way to build very complex data structures. While Python allows quotes (") and apostrophe (') for strings, JSON only allows double-quotes ("). We will cover JSON in much greater detail later in this module.

The following code shows a hybrid usage of dictionaries and lists.

In [14]:
```
# Python list & map structures
customers = [
    {"name": "Jeff & Tracy Heaton", "pets": ["Wynton", "Cricket",
        "Hickory"]},
    {"name": "John Smith", "pets": ["rover"]},
    {"name": "Jane Doe"}
]

print(customers)

for customer in customers:
    print(f"{customer['name']}:{customer.get('pets', 'no pets')}")
```

```
[{'name': 'Jeff & Tracy Heaton', 'pets': ['Wynton', 'Cricket', 'Hickory']},
{'name': 'John Smith', 'pets': ['rover']}, {'name': 'Jane Doe'}]
Jeff & Tracy Heaton:['Wynton', 'Cricket', 'Hickory']
John Smith:['rover']
Jane Doe:no pets
```

The variable **customers** is a list that holds three dictionaries that represent customers. You can think of these dictionaries as records in a table. The fields in these individual records are the keys of the dictionary. Here the keys **name** and **pets** are fields. However, the field **pets** holds a list of pet names. There is no limit to how deep you might choose to nest lists and maps. It is also possible to nest a map inside of a map or a list inside of another list.

More Advanced Lists

Several advanced features are available for lists that this section introduces. One such function is **zip**. Two lists can be combined into a single list by the **zip**

command. The following code demonstrates the **zip** command.

In [9]:
```
a = [1,2,3,4,5]
b = [5,4,3,2,1]

print(zip(a,b))
```

```
<zip object at 0x7fc5d8419e60>
```

To see the results of the **zip** function, we convert the returned zip object into a list. As you can see, the **zip** function returns a list of tuples. Each tuple represents a pair of items that the function zipped together. The order in the two lists was maintained.

In [12]:
```
a = [1,2,3,4,5,9]
b = [5,4,3,2,1,3]

print(list(zip(a,b)))
```

```
[(1, 5), (2, 4), (3, 3), (4, 2), (5, 1), (9, 3)]
```

The usual method for using the **zip** command is inside of a for-loop. The following code shows how a for-loop can assign a variable to each collection that the program is iterating.

In [17]:
```
a = [1,2,3,4,5]
b = [5,4,3,2,1]

for x,y in zip(a,b):
    print(f'{x} - {y}')
```

```
1 - 5
2 - 4
3 - 3
4 - 2
5 - 1
```

Usually, both collections will be of the same length when passed to the **zip** command. It is not an error to have collections of different lengths. As the following code illustrates, the **zip** command will only process elements up to the length of the smaller collection.

In [18]:
```
a = [1,2,3,4,5]
b = [5,4,3]

print(list(zip(a,b)))
```

```
[(1, 5), (2, 4), (3, 3)]
```

Sometimes you may wish to know the current numeric index when a for-loop is iterating through an ordered collection. Use the **enumerate** command to track the index location for a collection element. Because the **enumerate** command

deals with numeric indexes of the collection, the zip command will assign arbitrary indexes to elements from unordered collections.

Consider how you might construct a Python program to change every element greater than 5 to the value of 5. The following program performs this transformation. The enumerate command allows the loop to know which element index it is currently on, thus allowing the program to be able to change the value of the current element of the collection.

In [19]:
```
a = [2, 10, 3, 11, 10, 3, 2, 1]
for i, x in enumerate(a):
    if x>5:
        a[i] = 5
print(a)
```

```
[2, 5, 3, 5, 5, 3, 2, 1]
```

The comprehension command can dynamically build up a list. The comprehension below counts from 0 to 9 and adds each value (multiplied by 10) to a list.

In [20]:
```
lst = [x*10 for x in range(10)]
print(lst)
```

```
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
```

A dictionary can also be a comprehension. The general format for this is:

```
dict_variable = {key:value for (key,value) in
dictonary.items()}
```

A common use for this is to build up an index to symbolic column names.

In [21]:
```
text = ['col-zero','col-one', 'col-two', 'col-three']
lookup = {key:value for (value,key) in enumerate(text)}
print(lookup)
```

```
{'col-zero': 0, 'col-one': 1, 'col-two': 2, 'col-three': 3}
```

This can be used to easily find the index of a column by name.

In [22]:
```
print(f'The index of "col-two" is {lookup["col-two"]}')
```

```
The index of "col-two" is 2
```

An Introduction to JSON

Data stored in a CSV file must be flat; it must fit into rows and columns. Most people refer to this type of data as structured or tabular. This data is tabular because the number of columns is the same for every row. Individual rows may

be missing a value for a column; however, these rows still have the same columns.

This data is convenient for machine learning because most models, such as neural networks, also expect incoming data to be of fixed dimensions. Real-world information is not always so tabular. Consider if the rows represent customers. These people might have multiple phone numbers and addresses. How would you describe such data using a fixed number of columns? It would be useful to have a list of these courses in each row that can be variable length for each row or student.

JavaScript Object Notation (JSON) is a standard file format that stores data in a hierarchical format similar to eXtensible Markup Language (XML). JSON is nothing more than a hierarchy of lists and dictionaries. Programmers refer to this sort of data as semi-structured data or hierarchical data. The following is a sample JSON file.

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    },
    {
      "type": "mobile",
      "number": "123 456-7890"
    }
  ],
  "children": [],
  "spouse": null
}
```

The above file may look somewhat like Python code. You can see curly braces that define dictionaries and square brackets that define lists. JSON does require

there to be a single root element. A list or dictionary can fulfill this role. JSON requires double-quotes to enclose strings and names. Single quotes are not allowed in JSON.

JSON files are always legal JavaScript syntax. JSON is also generally valid as Python code, as demonstrated by the following Python program.

In [23]:
```
jsonHardCoded = {
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": True,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    },
    {
      "type": "mobile",
      "number": "123 456-7890"
    }
  ],
  "children": [],
  "spouse": None
}
```

Generally, it is better to read JSON from files, strings, or the Internet than hard coding, as demonstrated here. However, for internal data structures, sometimes such hard-coding can be useful.

Python contains support for JSON. When a Python program loads a JSON the root list or dictionary is returned, as demonstrated by the following code.

In [24]:
```
import json

json_string = '{"first":"Jeff","last":"Heaton"}'
obj = json.loads(json_string)
print(f"First name: {obj['first']}")
print(f"Last name: {obj['last']}")
```

```
First name: Jeff
Last name: Heaton
```

Python programs can also load JSON from a file or URL.

In [25]:
```
import requests

r = requests.get("https://raw.githubusercontent.com/jeffheaton/"
                 +"t81_558_deep_learning/master/person.json")
print(r.json())
```

```
{'firstName': 'John', 'lastName': 'Smith', 'isAlive': True, 'age': 27, 'addr
ess': {'streetAddress': '21 2nd Street', 'city': 'New York', 'state': 'NY',
'postalCode': '10021-3100'}, 'phoneNumbers': [{'type': 'home', 'number': '21
2 555-1234'}, {'type': 'office', 'number': '646 555-4567'}, {'type': 'mobil
e', 'number': '123 456-7890'}], 'children': [], 'spouse': None}
```

Python programs can easily generate JSON strings from Python objects of dictionaries and lists.

In [26]:
```
python_obj = {"first":"Jeff","last":"Heaton"}
print(json.dumps(python_obj))
```

```
{"first": "Jeff", "last": "Heaton"}
```

A data scientist will generally encounter JSON when they access web services to get their data. A data scientist might use the techniques presented in this section to convert the semi-structured JSON data into tabular data for the program to use with a model such as a neural network.

In [ ]:
```

```