Artificial Intelligence and Machine Learning with Python

Coursework Module

Dr Naeemullah Khan

Version of: June 16, 2021

May 24-28, Trinity Term 2021
Department of Engineering Science, University of Oxford

Contents

Chapter 1

Introduction

Artificial Intelligence and Machine Learning is fast becoming one of the leading areas of STEM research. Some call it the fourth industrial revolution. So, why is AI and ML becoming so popular and ubiquitous? We will try to answer this question in this course work module.

In this course work module we will try to familiarize ourselves with theoretical and practical aspects of machine learning. The main goals of this course are:

- To learn the basics of `Python` programming for AI and ML research.
- To understand the theoretical aspects of artificial intelligence and machine learning.
- To give a quick introduction to `PyTorch` and `TensorFlow` , two of the most popular machine learning libraries.
- To implement some practical examples of standard machine learning algorithms.

During this week long course `Jupyter Notebooks` will be provided that will consist of the explanation of the daily exercises, examples and lecture notes. We will run these notebooks on `Google Colab` . The first part of each lab will consist of a short lecture explaining the key concepts and goals of each lab, after which the students will work on the assignment.

Exercise 1.1: Exercises will be highlighted like this, and the code monkey will help you find them in the text quickly.

Question: Questions, will be highlighted like this, the "question mark" we help you keep track of questions in the text.

?

Demonstrator(s) will be on hand throughout the course, ask as many questions as you like and don't be shy. For more detailed analysis of ML and AI tools we recommend: Pattern Recognition and Machine Learning , by Christoper Bishop.

PDF notes These notes are available as an online PDF: the demonstrators will tell you where to find this. The PDF will be useful when you want to search for particular keywords.

Course history This coursework module was developed by Dr Naeemullah Khan in 2020.

1.1 Artificial Intelligence and Machine Learning

Humans have an amazing ability to interact with their surroundings and learn complex patterns and rules in the environment. Human civilization is based on the ability of human to learn and create "things".

In the 1940's and 50's researchers started trying to use computers to learn and create ideas like humans do. Of course, the field started with baby steps and very small scale problems were solved. But the field of artificial intelligence was born.

There are quite a few buzzwords associated with the field, below we provide a brief description of some of these concepts.

Artificial Intelligence: is the field of simulating intelligence in machines. If a machine can interact with its surrounding in any way and learn something from this interaction, it is essentially an intelligent entity. The field of designing algorithms for machines to interact and learn from their environment/data is called artificial intelligence.

Machine Learning: is the sub-branch of artificial intelligence where the learning is data driven i.e. the machine does not have tools to directly interact with the environment but it has access to the data of different interactions and corresponding outputs and the under-laying process is learned from this data.

Deep Learning: is the sub-branch of machine learning where the task to be learned are very complex and in order to learn these tasks we need the networks to have very large capacity and a lot of trainable parameters. To achieve this the networks that are used have many layers and are called deep networks and the field is called deep learning.

To be added: More explanation of the the three fields. some examples, 2 figures, supervised and unsupervised learning examples

Chapter 2

Getting Started

Originally for this course work module, we had planned to use `Jupyter Notebooks` on lab computers to do all the exercises and demonstrations. However, because of the current situation, I though it would be more convenient for the students to work with `Google Colab` .

In this chapter [1] we will provide a quick overview of `Google Colab` and how to use it for this course work module.

2.1 `Google Colab` : An Introduction

`Google Colab` provides a free `Jupyter Notebooks` environment that runs entirely on the cloud. `Jupyter Notebooks` provide a flexible interface to the code, documentation and the output through the web browser. It is relatively straight forward to use and allows us to write the `Python` code, text, math expressions, and the output in a single file which is accessed on the web browser. Furthermore, we can share the notebook with collaborators and run our codes on GPUs and TPUs for free! In addition, we don't have to install the common packages used for machine learning and they come pre-installed in the `Google Colab` environment.

2.2 Creating Our First `Google Colab` Notebook

Before we start, note that `Google Colab` is linked to the Google drive, and any `Google Colab` notebooks that you create or import will be stored on your Google drive. If you don't have a Google drive account, make one, and log into it.

Note: For the purpose of this lab, since we are using `Google Colab` , you will require a Google drive account to save your `Google Colab` notebooks.

We will start with opening the following URL in our browser `https://colab.research.google.com/`.
You will see the `Google Colab` home screen shown in Figure 2.2.

Next, we sill select the **NEW NOTEBOOK** option and it will create an empty notebook for us, like the one shown in Figure 2.2.

By default the `Google Colab` notebooks are named **Untitled(x)** where x is a number. You can click on the name of the notebook on the top left corner and rename it. The **ipynb** is the extension for jupyter notebooks.

As with jupyter notebooks, there are two types of cells in `Google Colab` notebooks **Code** and **Text** cells. Anything in the **Code** cell is considered as `Python` code. **Text** cells are for documentation, notes, mathematical expressions etc (Text cells are in markdown language).

[1] This chapter is motivated from `https://www.tutorialspoint.com/google_colab/google_colab_tutorial.pdf`

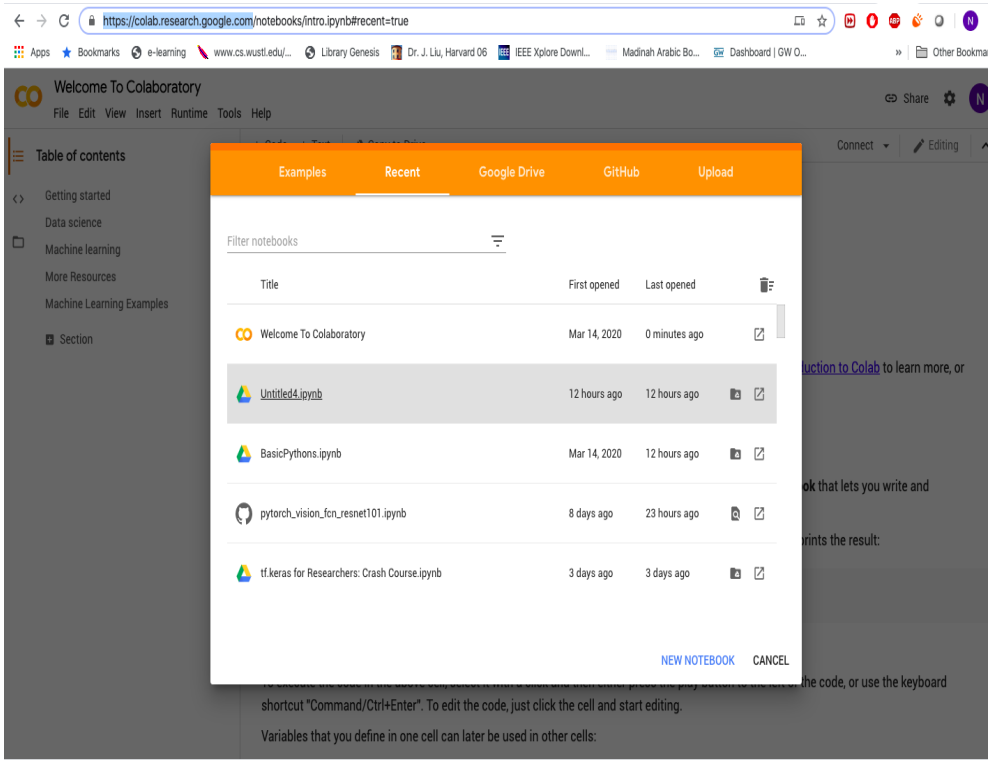

Figure 2.1: Google Colab Homepage

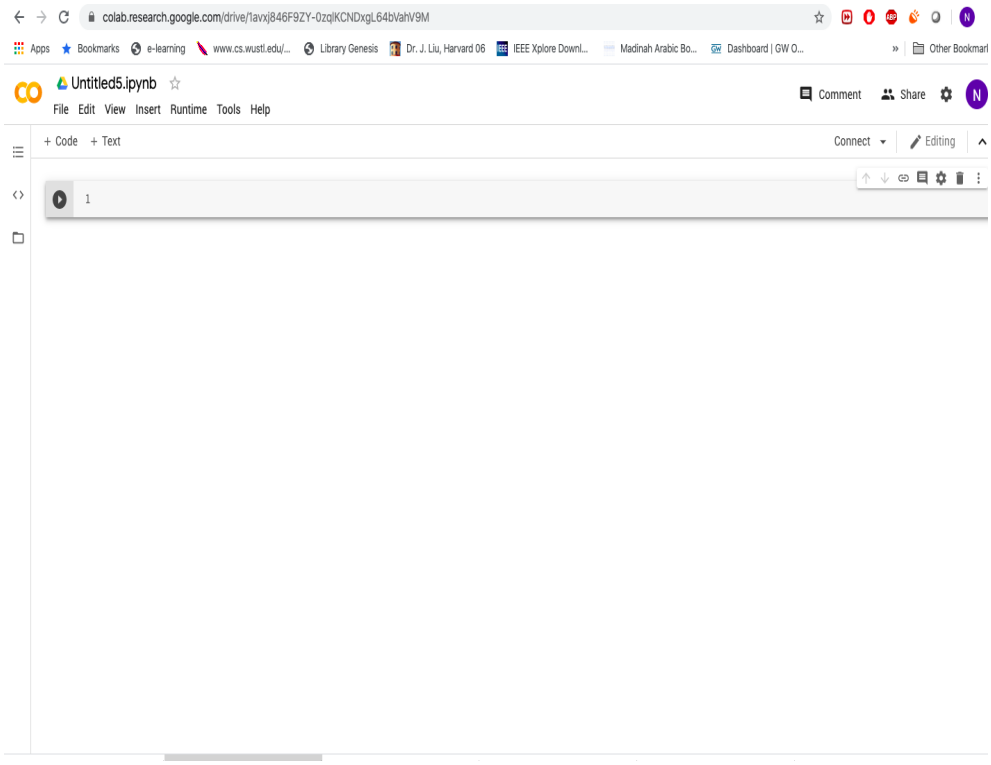

Figure 2.2: Empty Google Colab Notebook

In this chapter we will work on a few test codes, to show how Python codes are manipulated in Google Colab notebook. Don't worry too much about understanding the codes. We will look into the details of Python programming in the following chapter.

2.3 Entering and Executing Code

To enter code in Google Colab notebook click on the only cell in the newly created Google Colab notebook (notice that this cell is a code cell) and type in the following command:

```
print("Hello World from the colab notebook")
```

To execute the code you will click on the play sign on the left of the code cell. This should display "Hello World from the Google Colab notebook" (without the quotes) under the executed cell block. Similarly to execute any cell we need to click on the play sign (Run that cell) on the left of that cell block.

To add new code cells you can use one of the three options: 1) Click on the +Code button on the top left of the Google Colab notebook.2) From **Insert** select **Code Cell**. 3) Hover the mouse around the center of the block and it will give you the option of adding **Code** and **Text** cell below the current cell.

2.4 Runtime Options

For a large Google Colab notebook, we might want to run multiple or all cells at once, to do so, you can use the **Runtime** option of the Google Colab notebook, located on the toolbar on the top left of notebook. To run a group of selected cell you can use the option of **Run Selection** and to run all cells you can select the option **Run all**. There are other options available under **Runtime** as well which you can explore during the course of the course work module.

2.5 Cell Options

In a Google Colab notebook, we might want to move a cell around, delete a cell, etc. To access the cell option click on a cell and the cell options will be displayed on the top right of the selected cell. You will be able to move a cell up or down in your notebook, delete a cell, enter specific comments for a cell (this can be particularly useful to leave notes for changes to be done in future). You will also see the options for link to a cell and to add a form. Additionally, there are options to copy and cut a cell and options related to output of a cell. You can explore these options in detail during the course of the course work module.

2.6 Text Cells

A key strength of the Google Colab notebook is the *Text* cell feature, where we can provide the documentation for the code in the same notebook as the code itself. The text cells use markdown which is a popular markup language and is the super-set of HTML. We can write equations with all the capabilities of latex in the text cells. Some of the capabilities of the text cell are shown in Table 2.6

text cell input	text cell output
This is test	This is test
This is **bold**	This is **bold**
This is *italic*	This is *italic*
This is ~struck~	This is ~~struck~~
$x+y=3$.	$x+y=3$
$\sum _i x _i$	$\sum_i x_i$

Note: To remove the outputs from a `Google Colab` notebook move the mouse to the left of the output block and a cross sign will appear to remove the output.

2.7 Mounting Google Drive to `Google Colab`

At times we might need to handle data in our codes on `Google Colab` . To do so more efficiently we can mount Google Drive to `Google Colab` . The code below mounts Google Drive and then reads an image from a folder in Google Drive

```
from google.colab import drive
drive.mount('/content/drive')

!ls '/content/drive/MyDrive/AICWM'
#lists contents of the directory

from PIL import Image
import matplotlib.pyplot as plt
%matplotlib inline
#displays matplotlib outputs in the notebook


im=Image.open("/content/drive/MyDrive/AICWM/zebra1.png")

plt.imshow(im)
```

Details about using `matplotlib` will follow in Chapter 9.

Note: In `Jupyter Notebooks` we can use `!` to run shell commands. We can use `%` to run magics command in `Jupyter Notebooks` .

Chapter 3

Python Programming

Python is an open source programming language, probably the most widely used programming language for artificial intelligence and machine learning research. In this chapter we will provide an introduction to Python programming.

3.1 A simple Python program

Note: We will be using Python 3 for this course!

We start off by writing a simple Python program.

```
print("Hello World")
```

The above code in Python prints "Hello World" on the screen (without the quotes).

3.2 Variable and Simple Data Types in Python

Variables are the basic place holders for information in programming languages. Below we provide example of common types of variables in Python .

```
text="Hello World!"
print(text)
var_float=4.5
var_int=5
type(text)
```

In the above code 'text' is a variable which holds a string, 'var_float' is a variable which holds a floating type number and 'var_int' is a variable which holds an integer . Notice, in Python we do not need to specify the type of the variable during assignment and the type is automatically set depending on the value on the right hand side. We can check the data type of a variable by using the 'type' command in python.

Note: Variable names can contain only alphabets number and underscore, no spaces. Avoid using python keyword or functions names as variable names.

Good Practice: It is a good practice to use lowercase alphabets for variable names, uppercase letters are used in different names that we will see in the coming chapters.

Strings

A series of characters is called a string. In python anything inside quotes (either single or double quotes) is considered a string.

```
str1="test with double quotes"
str2='test with single quotes'
type(str1)
type(str2)
```

Some interesting combinations of the quotes:

```
str1="single quote ' ' inside double quotes"
str2='double quote " " inside single quote'
print(str1)
print(str2)
```

The following are some of the methods of the string class in Python .

Note: Methods will be formally defined in Chapter 4.


```
name="john doe"
print(name.title())
print(name.upper())
print(name.lower())
```

We can join different string and string variable in Python :

```
print("This is a test" + "to check the + operation for strings")
```

We use f-strings to add strings from string variables

```
str1="part 1"
str2="part 2"
str3="part 3"
cat_str=f"\t{str1}\t{str2}\t{str3}\n"
print(cat_str)
print(cat_str.rstrip())
print(cat_str.lstrip())
print(cat_str.strip())
```

The f_string format can be extended to multiple string variables in Python . Also notice the use of \t and \n and strip commands to add and remove white spaces in Python .

Note: The string.join(iterable) methods provides a flexible way of creting a string from an iterable (strings, lists, tuples etc). Each element of the iterable is sperated by the string. Try the example below:

```
str1='test'
print(str1)
print(','.join(str1))
```

Good Practice: "The most effective debugging tool is still careful thought, coupled with judiciously placed print statements." — Brian W. Kernighan

Integers

As the name suggests integer variables are place holders for integer data. Try the following operations for integers in Python

```
x=4
y=2
print(x+y)
print(x-y)
print(x/y)
#we can use // for integer division
print(x//y)
print(x*y)
print(x**y)
print((x*y)*(x+y)/(x**y))
```

+, -, *, /, ** represent addition, subtraction, multiplication, division, and power operation respectively. Also notice that we can group operations using (), the operation precdence order for un-grouped operations is power, division, multiplication, addition subtraction.

Float

Similar to integers all the above operations are valid for floats as well, with same operation precedence.

Note: If float(s) are involved in an operation Python casts the solution as a float as well. For division opeation the output is a float even if the operands are integers or the result is a whole number.


```
x=4
y=2.0
print(x*y)
```

Additional Features

Some interesting addition features in Python are:

- Underscore can be used in numbers for readability in Python .
- The assignment operation in Python can be used for multiple assignments.

```
x=10_1000
print(x)
x,y,z=1,2,3
print(x,y,z)
```

Good Practice: Python does not have a constant data type. Usually constant variable are labelled with all capital letters

The symbol # is used in Python for comments.

```
#This is the comment
print("The comment part is note executed by python compiler")
```

3.3 Basic Containers

Containers are used in Python to group objects together. We have already seen one type of containers in Python .

Question: What data type that we have seen so far groups together some objects? Hint: the objects that is group together are of character type.

The answer to the above question is strings. Strings are a collection of characters. The containers are grouped into two types in Python 1) mutable, 2) immutable. Mutable containers can be modified after their definition and immutable containers can not be modified after their definition.

Strings

Let's start with strings. As we have mentioned before strings are collection of characters. Strings is an immutable container in Python , that is a string can not be modified after it has been created. Strings are indexed by integers, that is, if you want to access an element (character) from the string you can access by using it's index. Notice that the first element of strings (as well as other indexable containers in Python) start with index 0.

```
str1='test'
str1[0]
str1[1]
str1[3]
str1[4]
str1[1]='b'
```

The last two lines of above code will produce errors. The 5th line is trying to access the 5th element of the string, where the string only has 4 elements. The 6th line tries to modify the 2nd element of the string to a new value and will results in an error as strings are immutable container in Python .

Lists

Lists are the most versatile datatype in Python . A list is defined by comma separating it's elements within square brackets. Elements of a list do not need to be of the same datatype.

```
my_list=[1,2,3,'a','dog',4.5]
```

The above code will create a Python list called my_list which has 6 elements. The first three elements are of integer datatype, followed by a character, a string and a float. Python list are mutable objects and can be modified after definition.

```
my_list=[1,2,3,'a','dog',4.5]
my_list[4]='cat'
del my_list[1]
print(my_list[3])
```

The above is a valid code and the second line will replace the 5th element of the list (dog) with cat. The third line of the code will delete the 2nd element of the list. The fourth line of the code will print the value of the 4th element of the list.
Some common operation and methods for lists are:

```
lst1=[1,2,3]
lst2=['a','b','c']
lst_cat=lst1+lst2
lst_rep=lst1*6
#to check if an element is a member of a list
#we can do x in list, it will return true or false value
print(3 in lst1)
#lists are iteratable objects
for x in lst1:
    print(x)

lst1.append(4)
print(lst1.count(3))
```

[append insert, del, pop, pop position, removing by values] Run the above command and discuss the results with the instructor to make sure that you understand what each command is doing.

Tuples

Tuples is an immutable Python container. Tuple structure is similar to lists with the exception that tuples can not be modified. Tuples can be defined by separating it's elements by comma, inside parenthesis.

```
my_tuple=(1,2,3'a','dog',4.5)
my_tuple[0]
my_tuple[3]=1
```

The first line produces a tuple called my_tuple, notice that similar to lists elements of tuple can be of different datatypes. The 2nd line of code will access the first element of the tuple. The 3rd line will produce and error as we are trying to modify the 4th element of the tuple and tuples are immutable objects.

```
tup1=(1,2,3)
tup2=('a','b','c')
tup_cat=tup1+tup2
tup_rep=tup1*4
del(tup_cat)
print(tup_cat)
len(tup_rep)

#to check if an element is a member of a tuple
#we can do x in tuple, it will return true or false value
print(3 in tup1)
#lists are iteratable objects
for x in tup1:
    print(x)
```

Dictionary

Dictonary object in Python is a collection of elements, where each element has a key and a value. The key is used to access the value. A simple way to understand the concept of dictionaries is to think about real dictionaries, where we can look up the meaning of words. In Python keys correspond to words and the values correspond to meaning of the words. Similar to a dictionary one key can only have one value (like one word has one meaning*). Dictionary are defined with curly brackets.

```
my_dict={'key1':'some value'}
my_dict['key2']='test1'
my_dict['key3']='test2'
print(my_dict)
```

Notice that in the definition of the dictionary, the key and the value are separated by a colon. We can also create an empty dictionary by leaving the curly brackets empty, and we can also create a dictionary with more elements in the definition by separating the elements by comma. Dictionaries are mutable objects and the values assigned to keys can be changed. Notice that by the construction of a dictionary the keys are immutable objects, we can't change a key once it is defined, although we can remove it and add new key values pair.

```
my_dict={'first_name':'N', 'last_name':'K', 'age':'xy'}
print(my_dict)
del(my_dict['age'])
my_dict['AGE']='xy'
my_dict.clear()
print(my_dict)
my_dict['name']='NK'
del(my_dict)
print(my_dict)
```

3.4 Loops

Loops is a programming concept where we want to repeat an action a certain number of times, or until a certain condition is met. For example we might be asked to practice a problem is class until we solve it correctly. Or we might be asked to exercise three times a week. In both these cases an action has to be repeated in the first case until we achieve a condition and in the second case until we achieve a certain number of repetitions.

For Loops

For loops in Python slightly different form other programming languages like C++, basically iterate over a sequence. Rather then being controlled by a counter, they are controlled by the size of the sequence.

```
for(int i=0; i<10; i++)
{
    std::cout<<i<<std::endl;
}
```

The above is a typical for loop in C++, where we start with a varibale i set to 0, i is incerase by 1 every time the loop is run and the opertion is terminated once i is greater then or equal to 10, so the loop will run 10 times.

By contrast in Python the simplest for loop expression will be something like this:

```
for i in range(10):
    print(i)
```

A few important points to notice here, before we start looking into the working of for loop. First is the range function in Python range returns a sequence of number. The more general use of range is range(start, stop, step), where the sequence starts from 'start' the seqeunce stops at 'stop' and the step size is 'step', by default start is 0, and step is 1.

Next, associated with every loop is the concept of the things or action that have be repeated a certain number of times or until a condition is met. The are generally specified in programming language as scope, i.e. scope for a loop is the thing or action that needs to be repeated. Scope in Python is represented by indentation. Hence, it is very important to keep track of indentations in Python .

Note: In Python codes indentation defines the scope of loops.

Now we come back to the for loop, the i variable of the for loop will iterate over (go over) each element of the seqeunce defined by range(10) (numbers from 0 to 9). For each i, we will print the value of i, since it is the scope of the for loop.

Because for loops are iterators in Python they can iterate over containers in Python which is a very desirable property and makes the manipulation of containers very easy.

```
lst1=[1,2,3]
tup1=('a','b','c')
dict1={'fn':'N', 'ln':'K'}

for element in lst1:
    print(element)
```

```
for element in tup1:
    print(element)

for key in dict1:
    print(key , dict1[key])


for c,element in enumerate(lst1):
    print(element)

for c, element in enumerate(tup1):
    print(element)

for c, key in enumerate(dict1):
    print(c, key , dict1[key])
```

While loop

The other type of loop in Python is while loop. While loops correspond to repetition while a condition is true. The syntax of while loop is explained in the example below

```
x=10
while (x>1):
    print(x)
    x=x-1
```

The while loop is run until the condition following the while key word is true. In the above example we have set the value of x to 10 in line 1. so the condition x>1 is true so the commands inside the while loop are performed, the values of x will be printed by line 3 and x will be decreased by 1 on line 3. The process will be repeated until we reach x=1 where the condition in no longer true and we will exit the while loop. Notice that like the for loop, the scope of the while loop is defined by indentation.

A side point to consider here is that with while loop we have to be careful that the conditions we specify are such that we don't get stuck in the loop indefinitely (unless of course this is something that we want in our code).

```
x=10
while (x>1):
    print(x)
    x=x+1
```

The above code will result in an infinite loop since the condition on line 2 is always true and as a result the code will run indefinitely.

Else with loops

Optionally we can use else with for and while loop and the else part of the code is executed after the loops are exited.

```
for x in range(10):
    print(x)
else:
```

```
    print('exited for loop')

x=10
while (x>1):
    print(x)
    x=x-1
else:
    print('x is 1')
```

In both the cases of for loop and while loop the else command is executed once the loop block is exited. Notice that else part of the code runs only once!

3.5 Python Function

One of the Python commands that we have used most frequently so far is print(). We provide some input to the print() command and it is displayed on the screen. But what exactly is this print() command doing? In programming languages such commands are called a functions. So one of the things that we notice right away is that the function print() is **reusable** i.e. we can use it to display different output at different points in the code. Another aspect to note is that use of function renders the code to be more **modular** i.e. different blocks of code do different tasks. In Python we have built in function like print() etc (built in function are those function which we don't define and come with standard Python installation). We can also define custom function in Python . Next we will see an example of a Python function to explore how to use functions in Python .

```
def square(x):
    "this function returns the square of a number"
    return x**2

#Now that the function is defined we can call it
print(square(3))
print(square(4))
```

Notice the definition of a Python function starts with the keyword def, followed by the function name and parentheses. Python function names, like other Python identifiers, can be a combination of letters (both upper and lower case), numbers and underscore. Spaces and special characters(!,*,&) are not allowed in the function name. Function name can not start with a number. Apart from these rules, special care must be observed to not use the names of the built-in functions for user-defined functions, e.g. we should not define a function with name print, since print is already a built-in function in Python .

Insider the parentheses are the parameters or arguments that are passed to the function, i.e. the values on which the function needs to perform certain operations. Parameters can also be defined inside the parentheses.

```
def square(x=3):
    "this function returns the square of a number"
    return x**2

#Now that the function is defined we can call it
print(square())
print(square(3))
print(square(4))
```

In the above example the parameter x is defined inside the parenthese of the function, when no input arguments are passed the value of x is set to 3. When the value of x is passed from the main the value is over-written inside the the function call and the function returns the square of the number passed from main.

The first line of the functions above has a string (which is called a docstring) and is optional. Docstring, if used, should contain some useful information about the function, i,e. what does the function do, what are the input parameter, what is returned from the function etc. The user can access the docstring by either using help() function or __doc__ attribute of the object (the definition of attributes and object will follow in the next chapter).

```
def square(x=3):
    "this function returns the square of a number"
    return x**2

#Now that the function is defined we can call it
square.__doc__
help(square)
```

The code block of the function start with a colon sign and the scope of the function is defined by indentation. The output of the function is passed back through the return statement. If the return is used without any arguments in front of it the function does not return any value. Return keyword also indicates that the function has ended it's execution.

Note: To quit Python help for a function press q.

Python Recursive Function

A recursive function is a function that call's itself. This is helpful in problem's when there is an inherent structure in the function that we can exploit. Let's take the example of factorial function to explain recursion. The factorial function of a natural number n is defined as $f(n) = n*(n-1)*(n-2)*...*3*2*1 = n*f(n-1)$.

```
def fact(n):
  if n==1:
    return 1
  else:
    return n*fact(n-1)
```

List Comprehension

In Python list comprehension provide a concise way to apply operations to elements of an iterable object. For example if want to square each element of a list or pass it to a function or to check if they meet certain conditions:

```
x=[1,2,3,4,5]
y=[i*i for i in x]
print(y)
print(x)
#print(i)
```

The above code performs operation on each element of "x" in a concise manner.

Dictionary Comprehension

Similar to list comprehension Python also provides dictionary comprehension to perform operations on dictionary elements in a concise manner.

```
dict1={'key1':'val1', 'key2':'val2'}

new_dict={key:val+' new' for (key,val) in dict1.items()}
print(new_dict)
```

The above code access each "key" and "value" in a Python dictionary and appends " new" to the each value of the dictionary.

Scope and Namespace

Try the below code and discuss the concept of scope and namespace with the instructor.

```
x="global"
def fun_scope():
    def fun_change_local():
        x = "local changed"

    def fun_change_nonlocal():
        nonlocal x
        x = "nonlocal changed"

    def fun_change_global():
        global x
        x = "global changed"

    x = "scope_fun scope"
    fun_change_local()
    print("After local assignment:", x)
    fun_change_nonlocal()
    print("After nonlocal assignment:", x)
    fun_change_global()
    print("After global assignment:", x)

fun_scope()
print(x)
print("In global scope:", x)
```

The nonlocal keyword is used to work with variables inside nested functions, where the variable should not belong to the inner function.

Exercises

Exercise 3.1: Write a program that asks the user to input, their name, title, age and occupation and outputs a string, "[title] [name] is [age] years old and works as a [job]."

Exercise 3.2: Write a program that takes a positive integer as an input and return's it's length. Return 0 if the number is not a positive integer.

Exercise 3.3: In a single line of code extract all numbers from a list that are greater than 100.

Exercise 3.4: In a single line of code capitalize the first character of the value strings in a `Python` dictionary.

Exercise 3.5: Write a program that checks if a 5 digit positive integer is palindrome. Bonus: extent it to any positive integer.

Exercise 3.6: Write a program which will find all such numbers which are divisible by 7 but are not a multiple of 4, between 100 and 1500 (both included). The numbers obtained should be printed in a comma-separated sequence on a single line.

Exercise 3.7: Write a code that inputs two integers and returns True if they are anagram of each other, else returns False.

Exercise 3.8: Write a program that takes a number and it's base (<36) and returns the value in decimal system. Assume number greater than 9 are represented by alphabets [a-z].

Exercise 3.9: Text based calculator: Write a program that take's in a string "No1 operator No2". and returns the output of the operation in string. Allowed operations are {+,-,*,÷}.

Exercise 3.10: Write a program that takes an input N and returns fibonacci sequence to N^{th} number in the sequence with and without using recursion.

Exercise 3.11: Write a program that takes a positive integer input N and returns all palindrome's uptill and including N.

Exercise 3.12: Write a program that return's the smallest of the integers in a list.

Exercise 3.13: Write a program that return's the largest number in a list.

Exercise 3.14: Write a program that removes all vowels from a string.

Exercise 3.15: Write a program that take's an input N (odd integer) and drawn an isosceles triangle with N stars on the base and the other sides of equal size.
Example: N=5

```
  *
 ***
*****
```

Exercise 3.16: Write a program that removes repeated words next to each other from a string.

Chapter 4

Object Oriented Programming

Programming methodologies can be broadly divided into two types 1) Functional Programming 2) Object Oriented Programming. In this chapter we will present an overview of object oriented programming as it will come in handy when using artificial intelligence and machine learning tools in `Python` .

4.1 What is Objected Oriented Programming?

Object oriented programming is a paradigm of programming where data and tools to manipulate it, are grouped together and this grouped structures is called an object (hence the name object oriented programming). Let's motivate the object oriented programming (OOP) with an example, say we want to design a system for an organization to manage it's employees. We have to store employee information and functions to handle employee pay raises, transfers etc.

The procedural approach to do this will be to write some functions which handle the pay raises and transfers, etc, and separately write data structures to hold employee data, when the functions are called be on the data structure we will call them by passing the data structure to them and transferring the output back to the data structure.

The OOP approach to this problem is much simpler: here, we define an object called the employee. This object will have all the data associated with the employee and the functions that control the data manipulation for the employee. If we need to call a function for an employee (object) we don't need to specifically pass any data around as the object has access to it's data.

4.2 Classes

On the surface the above idea of grouping data and methods in an object seem quite good but how do we actually do it in programming. Say for the above example if we have a couple thousand employees, how can we write the data and methods for each one of them in an object? If we have to do it for each employee individually it will be excessively prohibitive and virtually impossible for large number of objects.

In OOP similar objects are grouped together in classes. Classes can be thought of as templates to which object's belong. Objects can be generated by using the blueprint of a class. Below is the code for generating a new class in `Python` .

```
class Employee:
    pass
```

The above code generates a class called Employee. Right now the class is empty, we have used the keyword pass to prevent the error which is generated when you don't add any code under the class definition

Good Practice: Class names use CamelCase notation starting with a capital letter in Python to differentiate them from objects.

Note: Class definition in Python 2 is slightly different.

```
class Employee(object):
    pass
```

Notice that we have again used Python keyword pass here, which is used as a a place holder in Python , pass is where the code will go eventually.

Let's add some methods and attributes to the class Employee that we have defined above

```
class Employee:
    "a simple class"
    def __init__(self, fn, ln):
        self.first_name=fn
        self.last_name=ln

staff1234=Employee('Naeem', 'Khan')
print(staff1234.first_name, staff1234.last_name)
```

Those of you who are familiar with object oriented programming in other languages like C++ will see the similarities between constructors in C++ and the __init __ method in Python . I like to give the simplest definition that every time you create an object of a class the __init __ of that class is called and basically acts as an interface between the class and your code. Another important concept to understand is that of self (we can use a different name for this as well) variable of classes in Python . The variable self represents a particular instance object of the class inside the class definition. For example, let's say we create an instance object1 of a a particular class, then when the __init __ function is executed self would imply object1 and we can access the attributes and methods of object1 inside the class using the keyword self.

In the above code the 5th line of the code generates an instance of class Employee called staff1234. Naeem and Khan are passed as fn and ln to the __init __ function of the class employee and staff1234 is passed to self. The __init __ function assigns the value Naeem to the first_name attribute of staff1234 and assigns the value khan to the last _name attribute of staff1234.

Note: The . operator is used to access attributes and methods in Python . Try printing __doc __attribute for the above class and see what do you get.

Good Practice: Usually the instance object variable in methods inside the class is labelled self.

Note: If a class does not have an __doc __method instantiation the class creates an empty object.

Note: Instance object is passed as the first argument to the methods in a class.

```
def full_name(first_name, last_name):
    return first_name+ ' '+last_name

class Employee:

    def __init__(self, fn, ln):
        self.first_name=fn
        self.last_name=ln

    def full_name(self):
        return self.first_name+ ' '+self.last_name

staff1234=Employee ('Naeem', 'Khan')
print(staff1234.first_name, staff1234.last_name)
print(staff1234.full_name())
print(full_name(staff1234.first_name,staff1234.last_name))
```

The real advantage of the object oriented programming is the ability to the different methods to connect to objects in a seamless manner. For example for the above example if we wanted a function to join the first and last name of each employee we would have to provide the first name and last name of each object to that function and that function would have then returned the result to us. But using the object oriented programming we can define a method inside the class which will have access to the attributes of the object and we don't need to explicitly provide the data to the function, rather we just call the method for the instance of the class and it returns the desired results.

```
def full_name(first_name, last_name):
    return first_name+ ' '+last_name

class Employee:

    company='AI Course Work Module'

    def __init__(self, fn, ln):
        self.first_name=fn
        self.last_name=ln

    def full_name(self):
        return self.first_name+ ' '+self.last_name

staff1234=Employee ('Naeem', 'Khan')
print(staff1234.first_name, staff1234.last_name)
print(staff1234.full_name())
print(full_name(staff1234.first_name,staff1234.last_name))

print(staff1234.company)
```

In the above code we have introuduced a new attribute of the class Employee called company. Notice unlike the attriubutes like first_name and last_name the attribute company is not an attribute of each instance uniquely rather it is common among all attribute, such attributes are called class attributes.

Note: Check the __class__attribute of the names in your code and see what it returns.

Inheritance

Inheritance is the concept in object oriented programming where we derive a class from a different class. Let's work with the above example for Employee class, suppose we have different departments in an organization and we want to write separate class for each of these departments. Since these departments still consist of the employees of the organization, Do we need to re-define the methods we have defined in Employee class again for each department? The answer is no. The class for each department can inherit from the Employee class. Let's try to put this somewhat convoluted explanation in code and it would make things clearer.

```
#inheritance
class Employee:

    company='AI Course Work Module'

    def __init__(self, fn, ln):
        self.first_name=fn
        self.last_name=ln

    def full_name(self):
        return self.first_name+ ' '+self.last_name

class Dept1(Employee):
    department='Department 1'


staff1234=Dept1('Naeem','Khan')
print(staff1234.full_name())
```

In the above example the class Dept1 inherits the methods __init __and full_name from Employee. Notice that we have not defined these methods in the body of Dept1 but when we create an instance of class Dept1 it is using these methods from class Employee.
Now, let's run this code

```
#inheritance
class Employee:

    company='AI Course Work Module'

    def __init__(self, fn, ln):
        self.first_name=fn
        self.last_name=ln


    def full_name(self):
```

```
        return self.first_name+ ' '+self.last_name

class Dept1(Employee):
    dpeartment='Department 1'

    def __init__(self, fn, ln):
        self.first_name_dept=fn
        self.last_name_dept=ln


staf1234=Dept1('Naeem','Khan')
print(staf1234.full_name())
```

In the above code you will get an error when using the method full_name. The reason is that now our child class Dept1 has it's own __init __method which takes preference over the __init __of the parent class and hence attributes first _name and last _name are not defined when we call the full _name method, because the __init __of the parent class (Employee)is not executed.

But this seems to be a problem if this is the case then we will not be able to use the methods of the parent class. The answer to this problem is simple and we will explain it with an example below.

```
#inheritence
class Employee:

    company='AI Course Work Module'

    def __init__(self, fn, ln):
        self.first_name=fn
        self.last_name=ln


    def full_name(self):
        return self.first_name+ ' '+self.last_name

class Dept1(Employee):
    dpeartment='Department 1'

    def __init__(self, fn, ln):
        super().__init__(fn,ln)
        self.first_name_dept=fn
        self.last_name_dept=ln


staf1234=Dept1('Naeem','Khan')
print(staf1234.full_name())
```

The above code will work fine, despite the fact that we have method __init __in both parent and the child class(and we need to run both of these). Notice that the only change is the super().__init __command. The keyword super refers to the super class of the class and in the above code it will call the __init __method of the Employee class and fix the first _name and last _name so then we are able to use the methods of the Employee class. The keyword super can be used in OOP inheritance and extend a parent class.

Good Practice: It is better to keep the class attributes and instance attributes separate.

Note: The lookup for an attribute in an object follows the following rule:

- Arrtibute is looked for in the class first.
- if not found in above it is looked for in the parent-list from left to right in the class definition.
- In the parent class, it is looked recursively in the parents of parent class.
- if not found in above it is looked for in next parent and so on.

TO DO: isinstance issubclass

Exercises

Exercise 4.1: Write a child class with 2 parents and validate the above scheme of attribute search in instance of a class.

Exercise 4.2: Write a program (without using OOP) which takes in $(x, y) \in \mathbb{R}^+2$ a tuple of positive number representing the size of rectangular grid. It then takes an arbitrary number of parameters $(x, y), r$, where the (x, y) represents the center of a circle and r represents the radius. For each entry the program returns a "True" if the circle can be placed on the grid without it intersecting with any other circles and then places the circle on the grid. Else "False" if the circle can not be placed on the grid because it intersects with other previously placed circle.

Exercise 4.3: Re-implement the above with OOP.

Exercise 4.4: Re-implement the above in "n" dimensional space. where "n" is the input of the class constructor.

Exercise 4.5: Re-implement the above in 2 dimensional where we can now draw, circle, squares, triangles, and rectangles. The format for the shapes is:
circle: x, y, r where x, y is center of circle and r is it's radius
square: x, y, d where x, y is center of square and d is it's side length
square: x, y, l, w where x, y is center of square and l, w are it's length and width
triangle: $x_1, y_1, x_2, y_2, x_3, y_3$ where x_i, y_i represents the i^{th} corner of the triangle

Chapter 5

Regression

In this chapter, we will start looking into machine learning techniques. To start with we, look into regression.

5.1 Functions and Basis

Recall from calculus that we can approximate a function with a series, the most popular one being Taylor series approximation, where we approximate a function with it's projections on a polynomial basis:

$$f(x) = \sum_{n=0}^{N} \frac{f^n(x_0)(x - x_0)^n}{n!} \tag{5.1}$$

where, $f(x)$ is the function we want to approximate, index n denotes nth function in the basis and it ranges from 0 to N, $f^n(.)$ denotes the n^{th} derivative of the function, and $n!$ is the factorial of n.

A question to ask here is why does this approximation or any approximation of this type is good? To answer this question we need to understand the concept of basis. We can approximate any function if we choose a large/rich set of basis. "Basis" can be thought of as component function in a function space with the condition that none of the component function can be obtained through the linear combination of all other functions.

Note: A basis is a set of functions on a function space, if every continuous function on that space can be written as a linear combination of these basis. Elements of basis are linearly independent of each other. Can you guess why?

Note: The set of all functions that can be written as a linear combination of basis functions is called the span of the basis.

5.2 Regression from Data

In the above section, we decomposed a function in polynomial basis when we can calculate all the derivative so the function. What do we do, when we don't have access to the derivative of the function but only values of function at certain points? Think in term of real life, we can get samples of the under lying relation between variables but we don't always have access to the expression of this relation. Suppose, we want to find the relation between change in resistance of a conductor with temperature. We can vary the temperature of the resistors and measure the resistance. Now, we have a set of values $(x_j, r(x_j))$. Can we find as expression for the underlying function $r(x)$ given this data?

The answer is yes, we can try to fit a function to the data. In a way similar to the Taylor series approximation for analytic functions. We start out with an assumption that a certain

number of polynomial basis N will be sufficient for the approximation of the underlying function. Then we can setup a loss function on the data and minimise the loss for the optimum projections on these basis.

$$L(\boldsymbol{a}) = \frac{1}{2}\sum_{j=0}^{M}(r(x_j) - \sum_{n=0}^{N} a_n x_j^n)^2 \tag{5.2}$$

Here, $\boldsymbol{a} = [a_0, a_1, ..., a_N]$ are the unknown weights of the polynomial basis, that we are trying to find. $r(x_j)$ and x_j^n are the values of resistance and the values of temperature raised to nth power.

Any ideas as to how we can minimise this loss function? Simple calculus to the rescue :). We need to find the gradient of the loss function a find the points where it is zero, this will give us the critical points of the loss function, we can then look at the points where the Hessian matrix is positive definite.

It is much more convenient to use the matrix notation, but just to give an idea of the method, we will first derive expression for each variable independently. The derivative of the loss function with respect to the k^{th} element of vector $\mathbf{a}$ is

$$\frac{\partial L(\ \mathbf{a})}{\partial a_k} = \frac{1}{2}\sum_{j=0}^{M}(2(r(x_j) - \sum_{i=0}^{N} a_i x_j^i)x_j^k) \tag{5.3}$$

we can write the expression in terms of dot product and eliminate the sum over data samples

$$\frac{\partial L(\ \mathbf{a})}{\partial a_k} = \langle \boldsymbol{r}, \boldsymbol{x^k} \rangle - \sum_{i=0}^{N} a_i \langle \boldsymbol{x^i}, \boldsymbol{x^k} \rangle \tag{5.4}$$

where $\langle ., . \rangle$ represents dot product, $\boldsymbol{x}^i = [x_0^i, x_1^i, ..., x_M^i]^T$, and $\boldsymbol{r} = [r_0^i, r_1^i, ..., r_M^i]^T$. If we set equation 5.4 equal to 0 and rearrange the terms we get,

$$\sum_{i=0}^{N} a_i \langle \boldsymbol{x^i}, \boldsymbol{x^k} \rangle = \langle \boldsymbol{r}, \boldsymbol{x^k} \rangle \tag{5.5}$$

Now, we have an equation in $N+1$ variables, how do we solve it? At the very least we will need $N+1$ equations to solve for these $N+1$ variables. Any ideas, where we will get these extra N equations? We get it by taking the derivative with respect to the other components of $\boldsymbol{a}$, essentially the above expression for all values of $k \in \{0, 1, 2, ..., N\}$.

Let us the write the above, in matrix notation which is more convenient in our notation. We can rewrite 5.4 as

$$[\boldsymbol{x^k}]^T \boldsymbol{X a} = [\boldsymbol{x^k}]^T \boldsymbol{r} \tag{5.6}$$

where $\boldsymbol{X} = [\boldsymbol{x}^0, \boldsymbol{x}^1, ..., \boldsymbol{x}^N]$ and $\boldsymbol{a} = [a_0, a_1, ..., a_N]^T$. We can combine for all k equations, and the expression now becomes.

$$\boldsymbol{X}^T \boldsymbol{X a} = \boldsymbol{X}^T \boldsymbol{r} \tag{5.7}$$

Now this looks like the an expression we can solve, notice that $\boldsymbol{X}$ is a matrix of sampled basis at value of $x = x_j, j \in \{0, 1, , M\}$ the data points where we samples our original function $r(x)$. The column vector $\boldsymbol{r}$ are the value of function $r(x)$ at sampled points. Here we go now, to find vector $\boldsymbol{a}$ which are projections on our basis functions we can solve the above system of linear equations.

Some important things to note here, $\mathbf{X}^T\ \mathbf{X}$ is a square matrix and is full-ranked if $M \geq N$so we can find the inverse for this matrix. Only issue to look into is that matrix is well-conditioned, in order for us to invert it. Keep this point in mind, we will tie it up with the concept of regularisation.

5.3 Under-fitting, Over-fitting, and Regularisation

Now that we have introduced the tools for setting up a regression problem. Let's look into some important concepts associated with regression. The most import part of setting up a regression problem is the choice of the basis. How many basis do we use in regression? are fewer basis better, or are more basis better? and what is the right number of basis?

If we have a very small number of basis, we get an under fit where the basis are not sufficiently diverse to capture the properties of the training data (of course it would not work on test data as well). Simplest example of an under-fit problem is if we are trying to fit a data from quadratic equation with constant and linear basis. Of course a projection on these two basis will not be able to capture the quadratic data properly.

However, what happens if we have too many basis? Imagine if we have 100 data points and we try 100 basis. The regression function will match the data completely but will it be a good fit to the underlying distribution of the data. The answer is NO. Data from real life scenarios is almost always prone to some noise, typically what it means is that any good regression method should be immune to this noise. So a complete fit to the data might not be a good idea. We want the regression solution to fit the underlying distribution while being immune to the noise in the process. Thus, we have to be careful about the selection of the right number of basis.

There are detailed examples in the `Google Colab` notebook, to explain in detail the under-fitting and over-fitting scenarios.

Regularisation

The main idea of regularisation is to add a term to the energy, which penalises the projection on the basis which are not related to data.

$$L(\boldsymbol{a}) = \frac{1}{2}\sum_{j=0}^{M}(r(x_j) - \sum_{i=0}^{N} a_i x_j^i)^2 + \lambda \sum_{i=0}^{N} a_i^2 \tag{5.8}$$

Now, if we do the same steps for this equation that we did for the unregulariszed version of the equation we will get the expression for optimum weighs as

$$[\boldsymbol{X}^T\boldsymbol{X} + \lambda \boldsymbol{I}]\boldsymbol{a} = [\boldsymbol{X}]^T\boldsymbol{r} \tag{5.9}$$

Some important observations, The regularisation term puts a penalty on norm of the weight vector. Here we are using the second norm of the weight vector because it is the simplest norm to understand and implement. However, in practice it might not be the best regularisation to use in all case. Another popular norm used in regularization is the first norm of the weight vector. It has the desirable property of sparsity.

Note: To understand why the L2 regularization works. Assume that 2 columns of X are correlated then for we can have many values of projections on these two components for which the residual will be minimum (both components can cancel each other). Putting a penalty on weight of the projection ensures that the values for projection on correlation are not complementary.

Note: Another interpretation of the L2 regularization is this. If columns in X are highly correlated then the condition number for X^TX is very high (X^TX is not well conditioned). Adding the weight penalty converts the term into $X^TX + \lambda I$ which is well conditioned if λ is sufficiently large.

5.4 Logistic Regression

So far, we have seen how to regress data to a continuous function. In many applications in real life, however, we require to output discrete categories for data. The techniques that we have studied above will not work in these cases as we will not be able to define the derivative for function with discrete outputs.

To solve problem like these we use logistic regression. The core idea of logistic regression is: instead of finding the discrete class labels, we instead find the probability of data belonging to a certain class.

In this chapter we will setup a simple binary classification problem with logistic regression and provide some examples. In order to convert a linear regression model to output probability values for a class we use sigmoid function, defined as:

$$\sigma(x) = \frac{1}{1 + exp(-x)} \tag{5.10}$$

The output of the sigmoid function lies in the range (0,1), so it can be interpreted as the probability of belonging to a class.

Next we setup a linear logistic model. Let's assume we have data points $(\boldsymbol{x}_i, y_i)$ where $\boldsymbol{x_i}$ are input points and y_i is the corresponding class label. In the case of binary classification $y_i \in \{0, 1\}$. Our model is

$$\hat{y} = \sigma(h_\theta(x)) \tag{5.11}$$

where,

$$h_\theta(x) = \sum_{j=0}^{N} a_j x^j \tag{5.12}$$

Note: In our example, we are assuming x to be one-dimensional. It can be multi-dimensional as well.

Similar to the procedure for linear regression we can minimize the residual of the logistic linear regression

$$\sum_i (y_i - \hat{y}_i)^2 = \sum_i (y_i - \sigma(h_\theta(x_i)))^2 \tag{5.13}$$

To find the minima we set the derivative of the residual equal to zero and solve for the unknowns.

$$\frac{\partial}{\partial a_j} \sum_i (y_i - \hat{y}_i)^2 = 2 \sum_i (y_i - \hat{y}_i) * \sigma'(h_\theta(x)) * x^j \tag{5.14}$$

Kullback–Leibler Divergence and Cross-Entropy Loss

Since, the output of the sigmoid is interpreted as the probability of belonging to a certain class. For most application we will use metrics on probability like Kullback-Leibler divergence (KL divergence). Let's assume we have two discrete probability distributions P(x) and Q(x). The Kullback–Leibler divergence $D_{KL}(P||Q)$ is defined as:

$$D_{KL}(P||Q) = -\sum_x P(x) log \frac{Q(x)}{P(x)} \tag{5.15}$$

Some interesting properties of KL divergence:

- KL divergence is always non-negative
- In general $D_{KL}(P||Q) \neq D_{KL}(Q||P)$

- $D_{KL}(P||Q) = 0$ if and only if P and Q are identical
- KL divergence in convex in P and Q

Note: What happens to the value $P(x)log\frac{Q(x)}{P(x)}$ as $P(x) \rightarrow 0^+$? Why aren't we interested in $P(x) \rightarrow 0^-$?

In most machine learning applications, we will focus on the expression of cross entropy (a special case of KL divergence)

$$L_{CE}(y, \hat{y}) = -\sum_i (\sum_k y_i^k log(\hat{y}_i^k)) \tag{5.16}$$

where, i represents the data sample, k represents the number of out class $y^k = 1$ if y belongs to class k and is zero otherwise.

Note: Can you show that cross entropy loss is a special case to KL divergence.

For binary classification we will get

$$L = -\sum_i y_i^k log(\hat{y}_i^k) + (1 - y_i^k) log(1 - \hat{y}_i^k) \tag{5.17}$$

The derivative of the loss is

$$\frac{\partial L}{\partial a_j} = -\frac{y_i}{\hat{y_i^k}} * \sigma'(h_\theta(x)) * x^j + \frac{1 - y_i^k}{1 - \hat{y_i^k}} * \sigma'(h_\theta(x)) * x^j \tag{5.18}$$

which can be further simplified as:

$$\frac{\partial L}{\partial a_j} = -(y - h_\theta(x)) * x_j \tag{5.19}$$

Now, that we have defined the derivative of the loss function with respect to the unknowns, we can setup an iterative scheme for finding the minima of the problem. The details of the optimization tools to setup such schemes follow in the next chapter.

to be added: appropriate figures and exercises

Exercises

Exercise 5.1: Perform linear regression on data provided for the exercise. You can access the data by using the following commands:

```
!git clone https://github.com/naeem872/AICWM.git
import pandas as pd
import matplotlib.pyplot as plt
data=pd.read_csv('AICWM/Data/Q1trainData.csv')
x=data['x'].to_numpy()
y=data['y'].to_numpy()
plt.plot(x,y)
```

- using the above code snippet load the training data from Q1trainData.csv

- using the above code snippet load the test data from Q1testData.csv
- plot both the test and train data on the same plot
- perform the linear regression with 2 up to 10 basis and show the test error as a function of the number of basis
- can you observe over-fitting and under-fitting?
- perform the l_2 regularization and see how it changes the results

Exercise 5.2: Let's try to write a code to produce noisy version of a function and then try to perform polynomial regression on it:

```
x=np.linspace(0,10,1000)
x=x.T
f=0.2
pi=np.pi
m,b=1,1
y=np.sin(2*pi*f*x)+np.random.rand(x.size)
plt.plot(x,y)
```

- use the above code to generate test and train samples for the following functions: $\exp x$, $\log x$ (do we need to change the domain of the function), $u(x)$step function at $x = 5$, $\delta(x)$ delta function at $x = 5$

Exercise 5.3: Linear Regression for images: In this exercise we will show that our formulation of linear regression can be applied to images as well. First we create a test image:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

im=np.zeros((28,28))
im[10:15,10:15]=1

plt.imshow(im)
```

Next we define basis for linear regression. We will use the following basis:

$$b_{i,j} = \cos(\frac{\pi i x}{N})\cos(\frac{\pi j y}{M}) \tag{5.20}$$

where N, M is the size of the image. x, y represent the pixel coordinate. Vary the number of basis from $i, j = \{5, 10, 15, 25\}$ and see how the resultant image changes.

Exercise 5.4: Logistic Regression: Perform logistic regression on the data generated by the following code:

```
num_data=100 # data points per class

x1=np.random.randn(2,num_data)+1
x0=np.random.randn(2,num_data)

y1=np.ones((1,num_data))
y0=np.zeros((1,num_data))

X=np.concatenate((x1,x0),axis=1)
y=np.concatenate((y1,y0), axis=1)

print(X.shape)
print(y.shape)


plt.plot(X[0,:100],X[1,:100],'b*')
plt.plot(X[0,100:],X[1,100:],'r*')
X=X.T
y=y.T
```

Chapter 6

Optimization: Review

In chapter 5, we have introduced the simplest machine learning algorithms, where the learned function is linear in the unknown weights. Even for these simple cases we can't find a closed form solution for logistic regression. What happens if we use more complicated function of the unknown weights. Of course, in most cases we won't be able to solve for the minima in closed forms. In this chapter we provide some background of the optimization techniques we will use to find the minima of non-linear loss functions for AI and ML applications.

Question: Is the span of fixed basis sufficient to capture complex natural phenomenons?

In real life applications, in most cases, we can not write the unknown function as a linear combination of fixed basis. In fact, we are mostly interested in learning the basis jointly with the weights in the form of a highly non-linear function.

So, can we solve for the optimum of these highly non-linear functions in closed form. The simple answer is, no, we can not. But, we have some iterative techniques to solve for the optimum that we implement for most of these learning approaches and which work very well.

6.1 Review: Finding the Stationary Points of a Non-linear Function

Let's assume that we have a function $g(x)$ the derivative of which $f(x) = g'(x)$ is non-linear in x and we want to find stationary points of the function $g(x)$ what can we do?

One way to solve this problem will be to use Newton-Raphson method for finding the roots of $f(x)$. Newton-Raphson method is an iterative scheme where

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{6.1}$$

The above converges when $f(x) = 0$, notice that $f(x)$ corresponds to the roots of the gradient and will give us a stationary points of the function $g(x)$.

Question: Try to derive the expression above for Newton-Raphson method. Hint: Write the first-order Taylor series approximation of $f(x)$ and solve for Δx.

6.2 Gradient Descent

Note: Gradient is the direction of maximum change for a multi-variable function. For a function $f(x, y, z)$ gradient is defined as: $\nabla f = [\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}]^T$.

Note: Notice, that for we have defined gradient decent algorithm for a function of single variable but it can be easily extended to function of multiple variables.

Generally speaking, we are not interested in all types of stationary points. In optimization problems we are interested in either maxima or minima. The problem for finding maxima and minima are easily interchangeable by multiplying the function with negative 1. So, in general if have an algorithm for finding minima we can solve typical optimization problems.

The simplest algorithm in optimization for finding local minima is gradient decent. Gradient decent algorithm translates into taking a step locally in a direction where the function value is decreasing the most (i.e. negative gradient direction), and repeating the process until we are at a location of zero gradient (or very small gradient).

Let's assume that $f(x)$ is a function and we want to find a local minima for this function. Assume that we can calculate the function value and it's gradient and that we start with a point x_0, the gradient decent step will be

$$x_1 = x_0 - \mu f'(x_0) \tag{6.2}$$

where, μ controls the step size and f' denotes the gradient of the function f.

We can repeat the above step until we reach a point where $f' = 0$ where the algorithm will converge. Notice the $f' = 0$ corresponds to the critical points of the function f and since we are going in decent direction of function value (with very high probability) we will converge to a local minima.

Algorithm 1: Gradient Decent Algorithm

Result: Local minima
initialize x_0;
while $f'(x_n) > \epsilon$ **do**
  $x_{n+1} = x_n - \mu f'(x_n)$;
end

6.3 Gradient Decent with Backtracking:

In Algorithm 1 we provide simple gradient decent algorithm where step-size in each iteration is controlled by the hyperparameter μ. Ideally speaking at each iteration we would the value of μ which takes us closest to a local minima. Will a bigger value of μ always takes us closer to a minima?

Note: Even in the simplest case of quadratic complex function, for a sufficiently large value of μ we can increase the value of $f(x)$ in algorithm 1. Unless of course we are at a minima where the algorithm will converge.

There are algorithm which ensures that we tune the value of μ at step so that we can ensure sufficient decrease of the function $f(x)$ at each iteration. One such algorithm is Backtracking summarized in Algorithm 2

Algorithm 2: Gradient Decent with Backtracking Algorithm

Result: Local minima
initialize x_0, $\mu = 1$ $\alpha = 0.1$ $\beta = 0.5$;
while $f'(x_n) > \epsilon$ **do**
  while $f(x_n) - \mu * f'(x_n) > f(x_n) - \alpha * f'(x_n)$ **do**
    $\mu = \mu * \beta$;
  end
  $x_{n+1} = x_n - \mu f'(x_n)$;
end

Notice, that in backtracking, we start with a large value of μ and decrease it until a sufficient decrease condition is met.

Good Practice: In both `PyTorch` and `TensorFlow` , we should use decaying learning rates for training networks.

6.4 Stochastic Gradient Decent

So far, we have seen methods which are dependent on calculation of the gradient of the loss function for finding local minima. Assuming that our loss functions are differentiable can we always find the gradient?

Notice, that the loss functions are summation over all samples of data. For deep learning application we have a huge number of training samples, running beyond millions in some cases. This makes the computation of gradient very expensive. A way around this issue is to use stochastic gradient decent where rather then computing the gradient of the loss function for the entire dataset we calculate the gradient for randomly selected batch of data in each iteration.

Let's assume that our loss function is defined as

$$L(\boldsymbol{W}) = \sum_{x_i \in D} l_{\boldsymbol{W}}(y_i, \hat{y}(x_i)) \tag{6.3}$$

where, l is a loss function (x_i, y_i) is a training sample and D is the training data set. The stochastic gradient decent algorithm is provide below:

Algorithm 3: Stochastic Gradient Decent Algorithm

Result: Local minima
initialize $\boldsymbol{W}_0$;
while $\underline{||\boldsymbol{W}_{i+1} - \boldsymbol{W}_i||_2^2 > \epsilon}$ **do**
  $B \leftarrow$ random subset of D;
  $L(\boldsymbol{W}) = \sum_{x_j \in B} l_{\boldsymbol{W}}(y_j, \hat{y}(x_j))$;
  $\boldsymbol{W}_{n+1} = \boldsymbol{W}_n - \mu \nabla_{\boldsymbol{W}} L(\boldsymbol{W})$;
end

Stochastic gradient decent is the bread and butter of machine learning and artificial intelligence algorithm. Familiarize yourself well with this algorithm as you will be using it frequently.

6.5 Second Order Methods: Newton Method

This section is just for providing an overview and can be skipped All of the techniques we discussed above work with the first order Taylor series approximation of the functions. We can also use higher order approximation of the function for setting up our iterative schemes. The most common second order technique is Newton method. The algorithm for Newton method is summarized in Algorithm 4

Algorithm 4: Newton Method Algorithm

Result: Local minima
initialize x_0;
while $\underline{f'(x_n) > \epsilon}$ **do**
  $x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$;
end

Note: Newton method converges in fewer iterations to the local minima.

Question: Is there any relation between Newton-Raphson method for root finding and Newton method for finding minima?

Question: Does the function value decrease in each iteration of the Newton method?

Question: Can you prove using the second order Taylor series that the Newton method step is the best step we can take?

Exercises

Exercise 6.1: Write a python script to perform gradient decent on a function. Also implement the Newton method. Compare the number of iterations of convergence in both cases.

Chapter 7

Neural Networks

We have studied, linear and logistic regression models in Chapter 5. In linear regression models we fixed the basis and then try to find an optimal combination of these basis. Linear Regression model do not work well in more complex cases. One solution around these problem is to have a fixed number of adaptive basis where the basis and their combination is jointly learned during training. The most popular method of this type is Neural Network. In this chapter we will study the Neural Networks and important considerations for their implementation.

7.1 Basic Blocks of Neural Networks

The linear regression models we have seen so far are liner functions of some basis functions of the form:

$$f(x) = \sum_{i=0}^{N} w_i x^i \tag{7.1}$$

More generally we can write,

$$f(x) = \sum_{i=0}^{N} w_i \phi_i(x) \tag{7.2}$$

where ϕ_i are the fixed basis functions.

In neural networks, we write the basis as a composition of linear operation, where each linear operation is followed by a non-linearity. Each of these linear operation followed by a non-linearity is called a layer. Below we provide the linear operation in the first layer

$$a = Wx + b \tag{7.3}$$

where, x is in the input, W is a weight matrix, b is a vector of biases, and a is the output, known as activation. The non-linearity in the neural network is a non-linear activation function. There are several activation functions, the most popular are rectified linear unit (RELU) and sigmoids and softmax (details of activation function will follow below)

$$z = h(a) \tag{7.4}$$

where $h(.)$ is the non-linear activation function and it is applied element-wise to the input a and z is the output of the first layer.

More generally er can write, the first layer as $f^{[1]}(x) = h(W^{[1]}x + b^{[1]})$, then an N layer neural network would be, the composition of N layers

$$F(x) = f^{[N]} \circ f^{[N-1]} \circ ... \circ f^{[2]} \circ f^{[1]}(x). \tag{7.5}$$

where each layer $f^{(}[i])$ is parameterised by unknown parameters $W^{[i]}, b^{[i]}$.

Question: Can we remove the non-linearities in the neural networks? If we remove them what will happen?

?

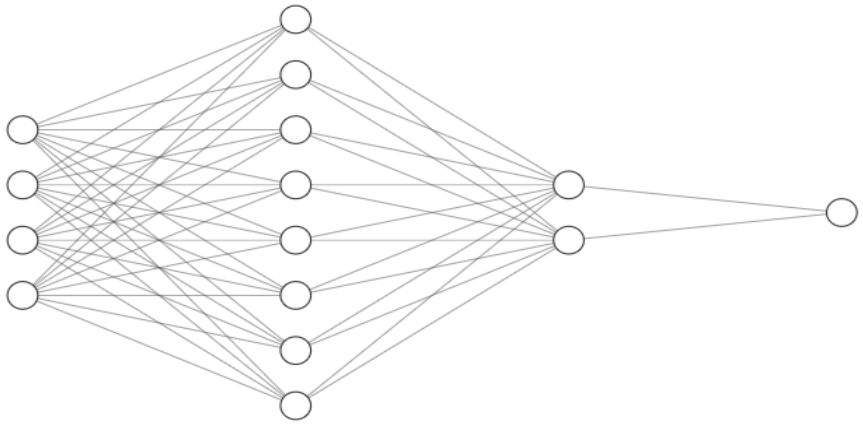

Figure 7.1: A three layer neural network

7.2 Activation Function

Activation functions are the non-linearities that are applied to the output of the linear operation of each layer. Some of the most commonly used activation functions are:

Rectified Linear Unit

Rectified linear unit is one of the most commonly used activation functions in machine learning and it is one of the reasons that we are able to train deep networks (network with many layers). A rectified linear unit layer is defined as

$$f(x) = max(0, x) \tag{7.6}$$

The derivative of the Relu layer is defined as

$$f'(x) = \begin{cases} 0 & x < 0 \\ 1 & x > 0 \end{cases} \tag{7.7}$$

Notice that the derivative is not defined at $x = 0$.

Sigmoid Function

Neural networks have proved very successful in classification applications, where the desired outputs are discrete class labels rather than continuous values. Instead of the network output being discrete labels (calculating derivatives for which will be difficult as the output will not be continuous), we design the network such that the outputs are the probability of input belonging to a class. Sigmoid function is useful in these scenarios. A sigmoid is defined as:

$$f(x) = \frac{1}{1 + exp(-x)} \tag{7.8}$$

The derivative of the sigmoid function is defined as

$$f'(x) = f(x) * (1 - f(x)) \tag{7.9}$$

The important property of the sigmoid function is that it's output lies in [0,1) which can be interpreted as probability of belonging to a class.

Tanh Function

Another common non-linearity used in the neural networks is tanh. Tanh is defined as:

$$f(x) = \frac{exp(x) - exp(-x)}{exp(x) + exp(-x)} \tag{7.10}$$

The derivative of the tanh function is defined as

$$f'(x) = 1 - f^2(x) \tag{7.11}$$

Softmax function

In most classification applications rather then dividing the problem into a set of binary classification problems (where we can use sigmoid), we want the output to be probability of belonging to each class, where the sum over all probability over classes is equal to 1. For these applications we used softmax function. Softmax function is defined as:

$$f(x_i) = \frac{exp(x_i)}{\sum_j exp(x_j)} \tag{7.12}$$

The derivative of the softmax function is defined as:

$$\frac{\partial f(x_i)}{x_i} = \frac{exp(x_i)}{\sum_j exp(x_j)} - \frac{exp(x_i)^2}{(\sum_j exp(x_j))^2} \tag{7.13}$$

$$\frac{\partial f(x_i)}{x_k} = -\frac{exp(x_i) * exp(x_k)}{(\sum_j exp(x_k))^2} \tag{7.14}$$

for $k \neq i$

7.3 Back-propagation and Weights Update in Neural Networks

Using equation (7.5) and the nonlinear functions defined above we can design a neural network of any shape and size. Calculating the output of the neural network amount to composition of layers in order as in (7.5). But what are the right values of the weights of the neural network and how can we train for the right values of the parameters of the network, given that we can have an arbitrarily complex network.

To answer this question we will first give a brief introduction of the chain rule we used in our calculus courses. Generally speaking we can define a derivative over composition of function as:

$$\frac{d}{dx} f(g(x)) = \frac{df(g(x))}{dg(x)} * \frac{dg(x)}{dx} \tag{7.15}$$

Calculating, the gradients of the loss function of the neural network with respect to the weights of the networks, translate of application of the chain rule.

Let's show it with and example. Let's assume that our neural network is defined as

$$F(x) = s \circ f^{[4]} \circ f^{[3]} \circ f^{[2]} \circ f^{[1]}(x). \tag{7.16}$$

where, $f^{[i]} = r * W^{[i]} + b^{[i]}$, and let's assume that we want to find the gradient of the loss function of the network with respect to $W^{[2]}$. Let's assume that the loss function is some function $L(F(x))$, here we will not concern our-self with what L is, we just know that the derivative of L is defined.

Now, Let's apply the chain rule to the loss function and try to find the gradient with respect to $W^{[2]}$

$$\frac{\partial L(F(x))}{\partial W^{[2]}} = \frac{\partial L(F(x))}{F(x)} * \frac{\partial F(x)}{\partial F^4(x)} * \frac{\partial F^4(x)}{\partial F^3(x)} * \frac{\partial F^3(x)}{\partial F^2(x)} * \frac{\partial F^2(x)}{\partial W^{[2]}} \tag{7.17}$$

where $F^i(x) = f^{[i]} \circ f^{[i-1]} ... \circ f^{[2]} \circ f^{[1]}(x)$ is the output of the ith layer. All the partial derivative in the above expression are either derivative of a linear function or of a non-linearity. We have calculated the derivatives of both types of functions above. So we have all the tools necessary to calculate the gradient of the loss function with respect to the parameters of the neural network.

Note: Notice, that in practical implementation, we don't need to store the output before and after all the activation functions, it saves us memory.

To Do: Some networks for typical problems, figure and exercises

Exercises

Exercise 7.1: Write a numpy based program to compute the forward and backward pass of a two layer neural network. Use this code to train a classifier for 2D data.

Exercise 7.2: Implement the above using PyTorch using SGD optimizer.

Chapter 8

Convolutional Neural Networks

Most of the data generated by users these days is images and videos and there is a growing need for machines to smartly interact with this data. Some common examples would be security applications, where we want to detect certain events or objects in images and videos. Image classification and detection, which is particularly useful in image search etc. Object recognition, where we want to recognize objects in videos, this is useful in robotics, where robot have to interact with their environment.

The fundamental building block of the data for all of the above applications is an image (videos are just a sequence of images). In order to extract useful features from images we use convolutions in artificial intelligence and machine learning applications.

What is Convolution?

Mathematically convolution is defined as:

$$y(\mathbf{x}) = \int_{\mathbf{t}} x(\mathbf{t}) f(\mathbf{t}-\mathbf{x}) d\mathbf{t} \tag{8.1}$$

where **x** is the input (2 dimensional in case of an image) **f** if the filter (2D in case of an image). The mathematical definition might seem a bit complex, a less formal but easy to understand definition that I like to use is as follows: The output of the convolution operation at a point is the sum of the product of element of x and f around that point (the values of f and the size of f characterizes each unique filter).

Why Convolution?

A good question to ask here is why do we need convolutions? Why can't we just use the extended formulation of neural networks here, and should that be sufficient for images? The answer is, yes, we can extend the neural networks to images, where we just flatten the image and use it as an input vector. This kind of architecture will be able to learn important feature in images but there are a few major drawbacks of this approach that we highlight below.

Drawbacks of Neural Networks for Images:

- Using neural network setup for image will require an excruciatingly large number of parameters. Suppose we have an image of size 256×256 and we want to extract 10 channels from this image each where the image is down-sampled by a factor of two. The first layer of our neural network will have 256*256*10*128*128 $\approx$ 10,000,000,000 parameters. This if order of magnitudes larger then the state-of-the-art CNNs we use in the field.
- The reduction in parameters in the CNN's don't necessarily come with a decline in performance. The reason for this is that in images we might have repeated patterns, hence we can use same filter at multiple points in the image to extract these patterns (neural networks are incapable of doing this, CNNs do it).
- CNNs provide better in-variance properties in images then NNs.

The details expressions of convolutions and the gradients of convolution operations are beyond the scope of this course work module. However, these are simple extensions of what we did for simple neural networks.

Chapter 9

Numpy: Review

Numpy short for numerical Python , is a Python package that is used for multidimensional array manipulation and operations. It is a library for scientific computing in Python . Numpy operations are very efficient in terms of time and memory and speed up the processing by quite a large margin. Below we will present some basic numpy codes for a quick introduction of this useful Python library [1].

```
1 import numpy as np
2 print(np.__version__)
```

The convention is to import numpy module with np alias. We can use the functions, definitions and statements etc in numpy module using the `.` operator. For example in the above code we access the `__version__` to print the version of numpy.

```
print(np.array(3).shape)
print(np.array([3]).shape)
print(np.array([[3]]).shape)
print(np.array([[[3]]]).shape)
```

Numpy arrays are the primary data processing containers in numpy. Numpy arrays are defined by using np.array command. The [] brackets are used to group data into vectors and matrices. The above code has same data in all arrays but the shape of the arrays is different owing to the [] we have used in each line of code.

```
import time as time
x_list=[i for i in range(100000)]
x=np.array(x_list)

st=time.process_time()
x_list=[i+10 for i in x_list]
ed=time.process_time()
print(round(ed-st,5))


st=time.process_time()
x+=10
ed=time.process_time()
print(round(ed-st,5))
```

[1] Parts of this chapter is motivated from https://cs231n.github.io/python-numpy-tutorial/

One of the main advantage of numpy arrays over python containers like lists etc is it's speed. Operations can be performed on numpy arrays much faster than on Python lists etc. The above code times the increment by constant operation for numpy arrays and Python lists and displays the time. Notice that for numpy arrays the operation is much faster.

```
x=np.array([1,2,3,4])
print(type(x))
print(x[0],x[1],x[2],x[3])
x[1]=x[2]
print(x)
print(x.shape)

y=np.array([[1,2,3],[4,5,6],[7,8,9]])
print(y.shape)
print(y.size)
print(y[1,2],y[0,1], y[2,2])

#indexing with negative indices
print(y[1,-1])
```

Numpy arrays can be initialized with multidimensional (or one dimensional) Python lists. All elements of the numpy array are of the same data type and the data type can be accessed with the type command of Python . Numpy array are indexed in the form x[obj] where x is the numpy array and obj it the type of indexing which can be field access, basic slicing and advanced slicing.

```
x=np.random.random((4,3))
print(x)

print('printing first column of x', x[:,0:1])
print('printing first row of x', x[0:1,:])
print('printing ? row ? cols', x[:1,2:])
print('printing ? row ? cols', x[1:3,1:3])
print('printing ? row ? cols', x[1:2,1:2])

y=x[1:3,1:3]
y[0,0]=0
#slice is a view into original array modifying it will
#modify original array as well
print(x)
```

Using index we can access rows or columns of the numpy array and even multiple rows and columns of the numpy array. The above code extracts rows and columns and specific combinations of rows and columns of numpy arrays. Here, we would like to convey some important distinctions between integer slicing and basic slicing. The syntax for basic slicing for numpy arrays is start:stop[:step] for each dimension. The slice obtained will have the elements in the corresponding dimension starting from "start" with step size of "step" and ending one place before "end". "start" and "step" argument are optional and by default their value is None. Basic slicing in Python gets a slice of the original array without copying it to a new memory location. Any changes made to the slice will result in changes in the array as well.

On the other hand integer indexing gets us a copy of the indices we have specified, and changes to them do not effect the original array.

```
#difference of integer indexing and slicing

x=np.random.random((5,5))
print(x)
y1=x[:,1]
y2=x[:,1:2]
print(y1,y1.shape)
print(y2,y2.shape)


z1=x[:,1]
z2=x[:,1:2]
print(z1,z1.shape)
print(z2,z2.shape)


print(x[[1,2,3],:].shape)
```

The above codes shows the difference of integer indexing and basic slicing for numpy arrays. Notice that integer indexing and basic slicing are accessing the same elements in the array but the output of integer indexing is a one dimensional array and the output of the basic slicing has the same number of dimensions as the original array albeit the size of each dimension is fixed by the slicing operation.

Note: Notice that a combination of integer slicing and basic slicing in numpy will give you a slice from the main array and changing it will cause changes in the main array.


```
#integer slicing tricks: one use of the integer index is to
#modify an elements in rows or columns on an np array

x=np.random.random([5,3])
#let's divide an element for each row by 2, first
row_ind=[1,0,2,0,1]
r,c= x.shape
print(x[np.arange(r),row_ind])
print(x)
x[np.arange(r),row_ind]/=2
print(x)

#let's multiply an element from each column by 5

col_ind=[3,4,1]
x[col_ind,np.arange(c)]*=5
print(x)
```

The above code uses integer slicing to modify a specific element in each row or column of the the array. Of course a similar scheme can be used to modify any random subset of elements of an array as long as we know the the indices of all the elements that needs to be changed.

```
x=np.random.random((6,6))
y=x[[1,2,1,3,0],[2,3,4,5,1]]

print(y)
print (x)
y[0]=1
print(x)
print(y)
```

As suggested in the note above integer indexing generates a new array as opposed to getting a slice from the original array so changing the new array does not change elements in the original array.

```
x=np.random.random([3,3])
sel_ind=x>0.5
print(sel_ind)
print(x[sel_ind])
```

In many applications we need to extract certain elements from a numpy array that satisfies a certain condition. For example we might be interested to find the days of the month in December where temperature was below zero degrees. This can be done in numpy with Boolean indexing. Boolean indexing essentially translates to a True of False value for an index, if the value is True the element at that index is selected else it is not.

Note: Similar to integer indexing Boolean indexing create a copy of the array elements and changing the copy does not change the original array.


```
x = np.zeros((3,2))
print(x)

y = np.ones((2,2))
print(y)

iden = np.eye(2)
print(iden)

r = np.random.random((2,2))
print(r)

c = np.full((2,2), 3)
print(c)
```

Many a times in programming exercises we need to generate arrays with particular type of values. For example zeros or ones, identity matrix or matrix with random values sampled from a particular distribution. Numpy has functions for generating these types of matrices. We can generate metrics with fixed values, identity matrices and random matrices with values distributed from uniform and normal distributions and random integers uniformly distributed from a range.

Numpy Datatypes

```
x=np.array([1,2,3])
y=np.array([1., 2., 3])
print(x.dtype)
print(y.dtype)
z=x+y
print(z.dtype)
```

Numpy allows us to use arrays of multiple datatypes. The ability to work with multiple datatypes is very useful. Imagine two applications where in one application you have to assign a unique ID to each student in a class and in the other application you are recording the temperatures observed in a city each day. In the first application you will probably not need Real numbers and positive integers would be sufficient for your need, where as in the second application you will need to work with real numbers. Using real numbers in the first application will add useless complexity to your problem and using integers in the second example will result in data loss. Therefore, it is nice to have the ability to use multiple datatype while programming the solution for a particular problem.

```
x=np.random.randint(0,9,(1,4))
y=np.random.randint(2,5,(1,4))
print(x,y)
print(x+y)
print(np.add(x,y))

print(x-y)
print(np.subtract(x,y))

print(x*y)
print(np.multiply(x,y))

print(x/y)
print(np.divide(x,y))

print("I have intentionally started the range of y from value
greater than zero, Can you think or a reason why?")

print(np.sqrt(x),np.sqrt(y))
```

The above code provides some of the operations we can perform on numpy array. Notice that all of the above operations are element-wise operations in numpy and performed element wise. $*$ is not matrix multiplication rather element wise multiplication between elements of the two arrays.

Good Practice: Always make sure that there are no divisions by zero in your code!

```
x=np.random.randint(0,9,(4,))
y=np.random.randint(0,9,(4,))


print(x.shape, y.shape)

print(x.dot(y.T))
```

```
print(np.dot(x,y.T))

print(np.matmul(x,y))

x_f=np.random.randint(0,9,(1,4))
y_f=np.random.randint(0,9,(4,1))


print(x.shape, y.shape)
print(x_f.dot(y_f))
print(np.dot(x_f,y_f))
print(np.matmul(x_f,y_f))


w=np.random.randint(0,9,(3,4))
z=np.random.randint(0,9,(4,3))

print(w.dot(z))
print(np.dot(w,z))
print(np.matmul(w,z))
```

For matrix multiplication (2D arrays) in numpy we can use either *np.dot* or *np.matmul* (np.matmul is preferred). Notice that for using matrix multiplications the two arrays should be compatible i.e. the number of columns of the first array should be equal to the number or rows of the second array. For 1D arrays np.dot returns the dot product. If both arrays in np.dot are multi-dimensional np.dot returns the product sum over the last dimension of the first array and second to last dimension of the second array.

Note: For complex conjugate dot product use np.vdot.

```
x=np.random.randint(0,9,(3,4,2))

print("x is :",x)
print("sum of x is:", np.sum(x))
print(np.sum(x,axis=0))
print(np.sum(x,axis=1))
print(np.sum(x,axis=2))


print("max of x is:",np.max(x))
print(np.max(x,axis=0))
print(np.max(x,axis=1))
print(np.max(x,axis=2))

print("min of x is:",np.min(x))
print(np.min(x,axis=0))
print(np.min(x,axis=1))
print(np.min(x,axis=2))
```

The above command demonstrates how to get the sum of a numpy array and the sum along a particular dimension of an array. Similarly we can also get the maximum or minimum values of a numpy array and maximum/minimum values along particular dimensions.

```
x=np.random.randint(0,9,(2,2))
y=np.random.randint(0,9,(2,2))
print(x)
print(y)
np.copyto(x,y)
print(x)
print(y)
```

Note: If we want to make a copy of a numpy array which can be modified independently of the original array, we can use np.copyto command.


```
x=np.random.randint(0,9,(4,3))
y=np.reshape(x,(2,6))
print(x)
print(y)

print(np.ravel(x,order='C'))    #try order 'C','F', 'A', 'K'

v=x.flat
print(v[6])

u=x.flatten()
print(u)
```

If we want to reshape an array we can use np.reshape. The size of the original array and the reshaped array should be same (by size we mean the number of elements), you can check the number of elements of a numpy array by using the command np.ndarray.size, you will see this type of notation quite a lot in numpy documentation what it means is the size is and attribution of ndarray which is a numpy object.

```
v=np.random.randint(0,9,(1,4))
print(v,v.T)
x=np.random.randint(0.9,(2,3))
print(x)
print(x.T)

z=np.zeros((2,3,4))
print(z.shape)
print(np.moveaxis(z,0,1).shape)
print(np.moveaxis(z,0,2).shape)


w=np.ones((5,3,4))
print(w.shape)
print(np.swapaxes(w,0,2).shape)

print(np.transpose(w, axes=(2,1,0)).shape)
print(np.transpose(w, axes=(0,1,2)).shape)
```

Transposition or moving dimensions is an important operation for array manipulation and comes in quite handy when you are doing matrix calculus. Numpy has transpose and swapaxes functions that can be used for these type of operations.

Complete list of array manipulation can be found here https://docs.scipy.org/doc/numpy/reference/routines.array-manipulation.html

Numpy Broadcasting

Broadcasting is a tool in numpy which allows for operation on numpy arrays without the need for looping over all the elements of the numpy array. Starting from the trailing dimensions of the numpy arrays the rules for numpy broadcasting are as follows:

- The size of the n^{th} trailing dimension for both arrays is equal
- The size of the n^{th} trailing dimension for one of the arrays is equal to one
- if at least one the above two condition is satisfied for all the trailing dimensions where shape is non-zero for both arrays. The array with larger shape dimensions can have any value where the smaller array has zero shape.

Example: if we have two arrays of shape $l \times r \times d \times c$ and c. The arithmetic operations between them is compatible and the output shape will be $l \times r \times d \times c$.

Example: if we have two arrays of shape $l \times r \times 1 \times c$ and $l \times 1 \times d \times c$. The arithmetic operations between them is compatible and the output shape will be $l \times r \times d \times c$.

The dimension that is 1 is copied the number of time corresponding to the size on the same dimension of the other array.

```
import numpy as np

x=np.ones((1,2,3))
y=0.5*np.ones((3))

print(x*y)
print((x*y).shape)


x=np.ones((1,2,3))
y=0.5*np.ones((5,1,3))

print(x*y)
print((x*y).shape)
```

9.1 Plotting in Python

For plotting in Python we use MatPlotLib library. Below are a few sample codes explaining the use of matplotlib .

```
import matplotlib.pyplot as plt
import numpy as np

%matplotlib inline

x = np.linspace(-1, 1, 100)
y = x*x + 1
```

```
plt.plot(x, y)
plt.show()
```

```
import matplotlib.pyplot as plt
import numpy as np

%matplotlib inline


x = np.linspace(-1, 1, 100)

for i in range(6):
  y = i*x + 1
  plt.subplot(2,3,i+1)
  plt.plot(x, y)
plt.show()
```

Exercises

Exercise 9.1: Time different operations for numpy arrays and `Python` lists and display the output.

Exercise 9.2: If you are given a random variable $x \sim U[0,1]$ can you generate a random variable $y \sim U[a,b]$ where $b > a$?

Exercise 9.3: Test np.dot(x,y) and np.matmul(x,y) on the following arrays.

- x and y are 1D
- x is 2D y is 1D
- x is 1D and 2D y is scalar
- x is 3D and y is 1D
- x is 3D and y is 3D

Exercise 9.4: Write a code which takes in the row and columns wise maximum and minimum values of two 2D arrays of same size and returns the upper and lower bounds of the maximum and minimum values of the sum of the two arrays.

Exercise 9.5: Try the following:

- np.transpose for 1D and 2D array
- np.transpose for 3D and 4D array
- np.swapaxes for 2D and 3D array

Exercise 9.6: Write a program that returns the Frobenius norm of a square matrix.

Exercise 9.7: Write a program to implement Newotn method to find the solution of $p(x,n)=0$, where n is the order of the polynomial p. The input of the program is an array of the coefficent of the polynomials. Output is the solution x_f of the equation. Example: input $[2,1,6]$ mean's we have to solve $2x^2+x+6=0$. Newton method is an iterative technique for solving non-linear equations with $x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)}$

Exercise 9.8: Write a program that takes in image (I) of shape $l\times w\times c$ and mask (m) of $l\times w$, where the mask has elements in $R=\{1,2,...,N\}$. We need to calculate $E=\sum_c\sum_{r\in R}\sum_{i,j\in l,w}(I[i,j,c])-a[r,c])^2*(m[i,j]==r)$. Where $a[r,c]=\sum_{i,j\in l,w}I[i,j,c])*(m[i,j]==r)/\sum_{i,j\in l,w}(m[i,j]==r)$. Do the above using for loops and without using for loops using numpy broadcasting.

Use the following code to generate the image and the mask.

```
im=np.zeros((32,32))
im[:16,:]=1

mask=np.zeros((32,32))
mask[:,:16]=1

plt.imshow(mask)
```

Chapter 10

PyTorch Basics

PyTorch is one of the most widely used package for artificial intelligence and machine learning. PyTorch is maintained by Facebook. PyTorch by design has been developed to be as close to numpy for an easy transition to PyTorch for those who are familiar with scientific computation in Python .

The building block for PyTorch and for that matter most of deep learning tools is a Tensor. Tensor is similar to numpy array. Let's start with importing torch and torchvision modules.

```
import torch
import torchvision
import torch.nn as nn
import numpy as np
import torchvision.transforms as transforms
```

To start off, we start by defined zero dimensional, one and two dimensional tensors. Notice that similar to numpy, associated with a PyTorch tensor is the notion of datatype, shape and size.

```
x=torch.tensor(2.)
print(x)
print(x.dtype)
print(x.shape)


y=torch.tensor([1,2,3])
print(y)
print(y.dtype)
print(y.shape)


z=torch.tensor([[1,2],[3,4]])
print(z)
print(z.dtype)
print(z.shape)
```

Similar to array in numpy the basic object for PyTorch is a tensor. Tensor can be thought of as a multi-dimensional array. We can generate different types of tensor in PyTorch .

```
#computational graph
x=torch.tensor(2.,requires_grad=True)
y=x*x +2*x+3
```

```
y.backward()
print(x.grad)
```

The fundamental block of computation defined in PyTorch is a computation graph. The forward pass in the computational graph is just going from input to output. The backward() command computes the gradient with respect to all the variables for which the "require_grad" flag is set to "True" the default value of this flag is "True".

```
x=torch.tensor(2.,requires_grad=True)

lr=torch.tensor([1e-2])
for i in range(1000):
  print('iteraion is:', i)
  x=torch.tensor(x,requires_grad=True)
  y=x*x +2*x+3
  y.backward()
  with torch.no_grad():
    x=x-lr*x.grad.data
  print(x,y)
```

As a first example in PyTorch , we will implement gradient descent algorithm on a simple function. Notice, how convenient it is to implement the algorithm in PyTorch . We don't have to compute the gradients explicitly rather PyTorch keeps track of the gradients for us. All we have to do is invoke the "backward()" command for the function and then we can use the gradient with respect to independent variable to update the variable until we reach the local minima.

Question: Implement the gradient descent using PyTorch for the following functions:

```
 dx_in, hl1, dy_out, n = 32, 50, 2 , 1000

x = torch.randn(dx_in, n)
y = torch.randn(dy_out, n)


w1 = torch.randn(hl1, x_in, requires_grad=True)
w2 = torch.randn(y_out, hl1,requires_grad=True)

learning_rate = 1e-6
for t in range(100):
  z=w1.mm(x).clamp(min=0)
  y_pred = w2.mm(z)


  loss = (y_pred - y).pow(2).sum()
  print(t, loss.item())

  loss.backward()


  with torch.no_grad():
    w1 -= learning_rate * w1.grad
```

```
    w2 -= learning_rate * w2.grad

    w1.grad.zero_()
    w2.grad.zero_()
```

The above example implements a two layer simple neural network in PyTorch . You can select the input dimensions, data size and the number of hidden unit and output units on the first lime of the code.

Let's take a deeper look at the code and try to see what the code is doing exactly. We are trying to setup a regression problem and hence you can see that both x and y are sampled from normal distribution. If it were a classification we would have sample y from Bernoulli distribution.

Similarly, we can see that weights of both layers also sampled from normal distribution. Notice that the require_grad flag has been set to "True" because we need to optimize the network with respect to these weights.

Note: Notice, that we have to initialize the weights of the network randomly, otherwise we will that gradient of the loss function for multiple weights of the network will be exactly same and they will evolve to same values.

In this example we are setting the learning rate of $1e-6$. Some good question to ask here are: Is this the best learning weight we can select for this problem? Should we keep the learning rate constant during the learning process?

Example motivated from https://github.com/jcjohnson/pytorch-examples#pytorch-autograd

```
 x = torch.randn(1000, 10)
y = torch.randn(1000, 2)

# Build a fully connected layer.
linear1 = nn.Linear(10, 50)
linear2 = nn.Linear(50,2)


# Build loss function and optimizer.
criterion = nn.MSELoss()
optimizer1 = torch.optim.SGD(linear1.parameters(), lr=0.01)
optimizer2 = torch.optim.SGD(linear2.parameters(), lr=0.01)


for i in range(100):
    optimizer1.zero_grad()
    optimizer2.zero_grad()
    h1 = linear1(x).clamp(min=0)
    pred = linear2(h1)
    loss = criterion(pred, y)
    loss.backward()
    optimizer1.step()
    optimizer2.step()
    print('loss after step optimization: ',i, loss.item())
```

```
 # Fully connected neural network with one hidden layer
class Net(nn.Module):
```

```
    def __init__(self, input_size, hidden_size, out_size):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, out_size)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out


x = torch.randn(1000, 10)
y = torch.randn(1000, 2)

net=Net(10,50,2)

criterion = nn.MSELoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.01)

for i in range(100):
    optimizer.zero_grad()
    pred = net(x)
    loss = criterion(pred, y)
    loss.backward()
    optimizer.step()
    print('loss after step optimization: ',i, loss.item())
```

Examples motivated form https://github.com/yunjey/pytorch-tutorial

10.1 Data Loading and Transformation

An import aspect of deep learning is the manipulation and augmentation of data. Notice, that in most cases the datasets are huge and can't be loaded in memory, not to mention the fact that we have to select random batches for the data to implement methods like stochastic gradient decent. Both `PyTorch` and `TensorFlow` provide tools for data manipulation and augmentation. In this section we will see an overview of these methods in `PyTorch` and see how these methods work.

```
import torch
import numpy as np
import os
import imageio as io
from torch.utils.data import Dataset, DataLoader
import matplotlib.pyplot as plt
from torchvision import transforms, utils
from skimage import io, transform
```

For the dataloader we will use the above packages. We will explaing the packages as they will be used in the code below.

```
%matplotlib inline
```

```
dataDir='' #directory here
D=os.listdir(dataDir)
print(len(D))
Dr=[]
for name in D:
    if not 'jpg' in name:
        Dr.append(name)
        name

for name in Dr:
    name
    D.remove(name)
```

In the code above we are trying to make a list of all jpg images in the `dataDir` folder. The command `os.listdir` gives a list of all the files in the folder. Notice that we might have other type of files in the folder as well. Just to remove the chances of any error we remove the non jpg extension files from the list. We loop over the directory `D` and collect all the non jpg images in a list `Dr` and in the last part of the code we remove elements of `Dr` from `D`.

```
class KaustDataset(Dataset):
    """Our dataset ."""

    def __init__(self,img_list, root_dir, transform=None):

        self.images = img_list
        self.root_dir = root_dir
        self.transform = transform

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        if torch.is_tensor(idx):
            idx = idx.tolist()

        img_name = os.path.join(self.root_dir,
                                self.images[idx])
        image = io.imread(img_name)
        sample = {'image': image}

        if self.transform:
            sample = self.transform(sample)

        return sample

kaust_dataset = KaustDataset(img_list=D,
                                    root_dir=dataDir)
```

In the code above we extend the `Dataset` class of `PyTorch` and define a new child class called `KaustDataset`. We then define three methods for this class. The first method is the `__init__` method which is used to pass the directory and transformation information that we need for our data pipeline.

The next method that we need to define is the `__len__` method which return the number of elements in our dataset. And lastly we define a method `__getitem__` which gets us an element of the dataset. Everything else is handled by the `Dataset` which is the parent class of our class. Notice that we have an if condition to convert the variable `id` to list type if it is not a list and we apply transform to the item if a transform is provided, by default the value of `transform` is `None`.

In the last part of the above code we define an object of the class `KaustDataset` with no transforms.

```
for i in range(len(kaust_dataset)):
    print(D[i])
    sample = kaust_dataset[i]

    print(i, sample['image'].shape)

    plt.imshow(sample['image'])


    plt.show()
```

The above code displays all images in the dataset.

Next, we want to apply some transformation to the dataset in the data loader pipeline. We define two classes of transformation, one to resize the image and the other to normalize the image. Notice that these type of transforms are very important in data pipelines and are used for data pre-processing and adjustment.

```
class Resize(object):
    """Resize the image."""

    def __init__(self, output_size):
        assert isinstance(output_size, (int, tuple))
        self.output_size = output_size

    def __call__(self, sample):
        image = sample['image']

        h, w = image.shape[:2]
        img = transform.resize(image,
        (self.output_size, self.output_size))


        return {'image': img}
```

To use PyTorch `transform` we define classes for our transformation, where we have two methods, `__init__` for passing arguments to the class and `__call__` to perform the transformation on a sample of the data. In the above code we resize the image to the input size.

```
class Normalize(object):
    """Normalize the image"""
```

```
    def __init__(self):
        print("init")
    def __call__(self, sample):
        image = sample['image']

        h, w,c = image.shape
        img=copy.copy(image)
        for i in range(c):
            mean=sum(sum(image[:,:,i]))/(h*w)
            std=(image[:,:,i]-mean)
            std=abs(std)
            img[:,:,i]=(image[:,:,i]-mean)/std



        return {'image': img}
```

Similar to the Resize class for Normalize has two methods __init__ for passing arguments to the class and __call__ to perform the transformation on a sample of the data.

Exercise 10.1: We have used Python 2 convention for class definitions. Try Python 3 class definitions and see if it is still works.

```
transformed_dataset = KaustDataset(img_list=D,
                                   root_dir=dataDir,
                                   transform=transforms.Compose
                                   ([
                                           Resize(124),
                                           Normalize()
                                                          ]))



print(len(transformed_dataset))

for i in range(len(transformed_dataset)):
    sample = transformed_dataset[i]
    print(sample['image'].shape)
```

In the above code we generate an instance of the dataset from KaustDataset but this time we use the transform argument as well and ask for composition of resize and normalize operation.

```
dataloader = DataLoader(transformed_dataset, batch_size=4,
                        shuffle=True, num_workers=4)

print(dataloader)
for batch in dataloader:
    print(batch['image'].shape)
    for i in range(batch['image'].shape[0]):
```

```
        plt.imshow(batch['image'][i,:,:,:])
        plt.show()
```

As a final step for the application of the stochastic gradient dataset we need batches of the dataset rather then the entire dataset. To do this we can use the DataLoader of PyTorch and it will generate the batch of the size provided.

Question: Modify the data loader code to add a horizontal flip and to add label from mask directory for each figure.

?

Exercises

Note: Below codes use the following plotting function:

```
def plotClass(X,y,p):
    plt.figure()
    for i in range(y.shape[0]):
        if y[i]==0:
            plt.plot(X[i,0],X[i,1],'r'+p)
        else:
            plt.plot(X[i,0],X[i,1],'b'+p)

    plt.show()
```

!

Exercise 10.2: Use simple neural networks to train a binary classifier for the following data. Study the effect of changing the value of offset the number of layers and hidden units on the classification result.

```
num_data=100 # data points per class
offset=5
x1=np.random.randn(2,num_data)+offset
x0=np.random.randn(2,num_data)

y1=np.ones((1,num_data))
y0=np.zeros((1,num_data))

x=np.concatenate((x1,x0),axis=1)
y=np.concatenate((y1,y0), axis=1)

print(x.shape)
print(y.shape)


plt.plot(x[0,:100],x[1,:100],'b*')
plt.plot(x[0,100:],x[1,100:],'r*')
x=x.T
```

```
x=torch.from_numpy(x).float()
y=torch.from_numpy(y.T).float()
```

Exercise 10.3: Use simple neural networks to train a binary classifier for the following data. Study the effect of changing the number of layers and hidden units on the classification result.

```
num_data=200

x=np.random.randn(2,num_data)
x=np.random.randn(2,num_data)

y=(x[0,:]**2+x[1,:]**2)>0.5
y.astype(int)
x=x.T
y=y.T
y=np.reshape(y,(num_data,1))
plotClass(x,y,'o')

x=torch.from_numpy(x).float()
y=torch.from_numpy(y).float()
```

Exercise 10.4: Use simple neural networks to train a binary classifier for the following data. Study the effect of changing the number of layers and hidden units on the classification result.

```
num_data=200
x=np.random.randn(2,num_data)


y=(x[0,:]*x[1,:])>0
y.astype(int)
x=x.T
y=y.T
y=np.reshape(y,(num_data,1))
plotClass(x,y,'o')

x=torch.from_numpy(x).float()
y=torch.from_numpy(y).float()
```

Exercise 10.5: Use simple neural networks to train a binary classifier for the following data. Study the effect of changing the number of layers and hidden units on the classification result.

```
num_data=500

x=np.random.uniform(-5,5,(2,num_data))


y=(np.floor(x[0,:]%2)!=np.floor(x[1,:]%2))>0
y.astype(int)
x=x.T
y=y.T
y=np.reshape(y,(num_data,1))
plotClass(x,y,'o')

x=torch.from_numpy(x).float()
y=torch.from_numpy(y).float()
```

Chapter 11

TensorFlow Basics

TensorFlow is the most popular deep learning library maintained by Google. In this course we will be working with TensorFlow 2.0 which comes with eager execution mode which allows for very Python like code behavior and is fairly easy to use. In TensorFlow 2, Keras a top level API comes incorporated and provides very convenient tools for creating and executing deep networks.
As a first step, we will start by importing TensorFlow . As a convention TensorFlow is imported as tf.

```
import tensorflow as tf
print(tf.__version__)
```

The above code prints the version of TensorFlow , notice that the version of TensorFlow is ≥ 2.

```
print(f"Eager execution is: {tf.executing_eagerly()}")
print(f"Keras version: {tf.keras.__version__}")
```

The above will print whether eager execution is enabled or not and the version of the Keras. Notice that Keras is part of the tf module in TensorFlow 2.

Note: Notice that by default eager execution is enabled in TensorFlow 2.


```
var = tf.Variable(1)
if tf.config.list_physical_devices('GPU'):
  print('Running on GPU')
else:
  print('Running on CPU')
```

The strength of the top machine learning libraries comes with their ability to leverage the capability of running the codes on GPU, which are much faster than CPU for matrix operations. The above code will print weather the code is running on GPU or CPU.

Note: You can enable the GPU for you Google Colab notebook, under Edit/ Notebook Settings.

Like any programming tool, the fundamental building block of TensorFlow are variables and operation. Below we will create TensorFlow variables of different rank,

```
import numpy as np

t_var0=tf.Variable(1)
t_var1=tf.Variable([1,2,3])
t_var2=tf.Variable([[1,2],[3,4]])

n_var3=np.random.random((3,3,3))
t_var3=tf.Variable(n_var3)
print(t_var0, t_var1, t_var2, t_var3)
```

The above code creates TensorFlow variables of rank 0, 1, 2, and 3. Notice that the variable creation in TensorFlow is similar to numpy array creation.

MNIST Classification with TensorFlow

In the section we will write our first complete machine learning code for performing MNIST classification with TensorFlow .

```
import tensorflow as tf
import tensorflow.keras as keras
import numpy as np
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
%matplotlib inline
%load_ext tensorboard
```

To create and run the model for MNIST classification we will need the above packages. We will explain each package as it is used in the code.

```
#importing data
(x_train, y_train),(x_test, y_test) =
tf.keras.datasets.mnist.load_data()
print(x_train.shape, y_train.shape, x_test.shape, y_test.shape)
```

The first step is to prepare the data for the training and testing. We use the data from tf.keras.datasets . Notice that load _data returns two tuples one with train data and label and the other with test data and label. We will print the shape of the four numpy arrays so that we can see how much data we have available and how are the samples stacked in these arrays i.e. the samples are stacked along the first dimension or otherwise.

```
(_, im_h, im_w)=x_train.shape
im_c=1      #gray scale image only 1 channel
max_train_val=np.max(x_train)
x_train=x_train/ np.float32(max_train_val)

min_test_val=np.max(x_test)
x_test=x_test/np.float32(min_test_val)
```

Next, we extract the image size from the data. Notice that `np.ndarray.shape` returns the tuple of shape, we extract the second and third elements for the tuple for height and width of the image.

Note: If are not interested in the value of a variable from a function or in unpacking we can assign it to a dummy variable name _.


```
#model design
hidden_units=64
num_classes=10
model = keras.models.Sequential()
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(hidden_units, activation='relu'))
model.add(keras.layers.Dense(num_classes, activation='softmax'))
```

Next we define our model, our model is simple neural network with one hidden layer. We specify the number of hidden units and the number of output class (which is 10 in this case since we are classifying digits). Number of hidden units is a hyper-parameter for which we can try different values and select the best one for our need.

Notice, that model is an instance of the `keras.model.Sequential` and we use the `add` method to add layers to the model. The first part of the model is the flatten operation which flattens the image into a vector. Next we add a fully connected layer to our model using `keras.layers.Dense` and specify the number of outputs of the layer and the activation, (Notice that keras keeps track of the inputs of the layer). Next, we add another fully connected layer with ten outputs with softmax activation. Since we are interested in the probabilities of an image belonging to a certain class we have used the softmax layer.

```
#test validation split
x_train,x_val, y_train, y_val=
train_test_split(x_train,y_train,test_size=0.10)
```

Next, we use the `train _test _split` to divide the training set into a training and a validation set.

```
#training the model
num_epochs=10
model.compile(optimizer='sgd',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
callbacks = [keras.callbacks.TensorBoard('./logdata')]
model.fit(x_train, y_train,
          epochs=num_epochs, verbose=1,
          validation_data=(x_val, y_val),
          callbacks=callbacks)
```

The above code performs the training of our model with the training data for 10 epochs.

```
# Evaluate the model
print(' Evaluate on test data')
results = model.evaluate(x_test, y_test, batch_size=64)
print('test loss, test acc:', results)

# Generate predictions
print('\n# Generate predictions for 5 samples')
predictions = model.predict(x_test[:5])
print(predictions)

for i in range(predictions.shape[0]):
  plt.imshow(np.reshape(x_test[i,:,:],(28,28)))
  label=tf.argmax(predictions[i,:])
  plt.title(str(label))
  plt.show()
```

The above code performs the evaluation on the test set.

```
%tensorboard --logdir logdata/
```

The above code opens the tensorboard to provide the details of the training and validation of the model.