Introducción a las Redes Neuronales con TensorFlow

Fundamentos de programación en Python

Centro de Investigación en Matemáticas, A.C.

CIMAT

manuel.suarez@cimat.mx

9 de agosto de 2022

Outline

Introducción

- Python es un lenguaje de propósito general lo que significa que puede desarrollarse cualquier tipo de programa en él, desde aplicaciones de bases de datos hasta implementaciones embebidas de inteligencia artificial.
- Su curva de aprendizaje no es muy pronunciada ya que el lenguaje se diseñó para ser sencillo a la vez que muy expresivo en su implementación.
- Para efectos del alcance de este curso en esta sesión vamos a estudiar los componentes básicos del lenguaje necesarios para la construcción de modelos de redes neuronales con TensorFlow.

Objetivos

- Objetivo General.- conocer los fundamentos del lenguaje necesarios para entender la implementación de modelos de redes neuronales con TensorFlow

Criterios de evaluación

Resolución de ejercicios

Características del lenguaje

- Sencillo de utilizar pero expresivo y potente en su formulación.
- Es modular, lo que permite dividir la implementación de un programa en diferentes módulos y reutilizar código.
- Es interpretado por lo que no requiere un ciclo de compilación y enlace de código y bibliotecas.
- Permite escribir programas de manera compacta y expresiva.
- Es extensible y soportado en los principales proveedores de servicios en la nube.

Variables

- En Python no es necesaria la declaración de variables ya que el lenguaje no asigna tipos a ninguna de ellas. Se usa un tipado dinámico de variables en el cuál las operaciones que se aplican a ellas depende del valor asignado en el momento de su evaluación.
- Dependiendo del contenido de una variable puede ser un tipo de dato simple (enteros, caracteres, lógicos) o compuestos (listas, tuplas, diccionarios, clases).

```
# Entero
>>> x = 1
# Punto flotante
>>> y = 1.5
# Cadena de caracteres
>>> z = 'Hola Mundo'
# Lista
>>> lst = ['a', 'b', 'c']
```

Expresiones

- Las operaciones que se pueden realizar dependen del tipo de dato al que se apliquen, en Python muchos operadores matemáticos se encuentran sobrecargados para realizar ciertas operaciones dependiendo del tipo de datos

```
>>> x = 1
>>> y = 2
# Suma y resta
>>> x + y
# Multiplicacion y division
>>> x * y
>>> x / y
# Division entera
>>> x // y
# Modulo
>>> x % y
```

Cadenas de caracteres I

- Manejo de cadenas de caracteres

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> 'doesn\'t'  # use \' to escape the single quote...
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
```

- Uso de la función **print**

Cadenas de caracteres II

```
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
>>> print('"Isn\'t," they said.')
"Isn't," they said.
>>> s = 'First line.\nSecond line.'  # \n means newline
>>> s  # without print(), \n is included in the output
'First line.\nSecond line.'
>>> print(s)  # with print(), \n produces a new line
First line.
Second line.
```

- **raw** strings

Cadenas de caracteres III

```
>>> print('C:\some\name')  # here \n means newline!
C:\some
ame
>>> print(r'C:\some\name')  # note the r before the quo
C:\some\name
```

- String literals

```
print("""\
Usage: thingy [OPTIONS]
     -h                        Display this usage messa
     -H hostname               Hostname to connect to
""")
```

- String interpolation

Cadenas de caracteres IV

```
x = 2
print(f"Value of x={x}")
y = 3
print(f"Value of y={y}")
print(f"Value of x+y={x+y}")
```

- String concatenation

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

- String indexation

Cadenas de caracteres V

```
>>> word = 'Python'
>>> word[0]  # character in position 0
'P'
>>> word[5]  # character in position 5
'n'
>>> word[-1]  # last character
'n'
>>> word[-2]  # second-last character
'o'
>>> word[-6]
'P'
```

- String slicing

Cadenas de caracteres VI

```
>>> word[0:2]  # characters from position 0 (included)
'Py'
>>> word[2:5]  # characters from position 2 (included)
'tho'
>>> word[:2]   # character from the beginning to positio
'Py'
>>> word[4:]   # characters from position 4 (included)
'on'
>>> word[-2:]  # characters from the second-last (inclu
'on'
```

Estructuras de bifurcación

- La bifuración nos permite controlar el flujo de ejecución de un programa por medio de la evaluación de una condición, si el resultado de esta condición es verdadera se ejecutará de manera condicional un bloque de código, en caso de ser falsa se podrá ejecutar otro.

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

Estructuras de repetición I

- Las estructuras de repetición nos permiten evaluar repetidamente una sección de código
- A diferencia de otros lenguajes la sentencia **for** itera sobre los elementos de una lista:

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

- Para iterar sobre una secuencia de números utilizamos la función **range**

Estructuras de repetición II

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4

>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
```

Estructuras de repetición III

```
4 lamb
```

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i, elem in enumerate(a):
...     print(i, elem)
...
```

break, **continue**, **else** Clauses on Loops I

- **break** detiene la ejecución de un ciclo, **else** proporciona una alternativa de flujo en la ejecución del ciclo al terminar de iterar sobre todos los elementos de la lista

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # loop fell through without finding a facto
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
```

break, **continue**, **else** Clauses on Loops II

```
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

- **continue** ejecuta el siguiente ciclo

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Found an even number", num)
...         continue
...     print("Found an odd number", num)
...
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
```

break, **continue**, **else** Clauses on Loops III

```
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

Sentencia **pass**

- La sentencia **pass** no realiza ninguna ejecución, se utiliza cuando se requiere la declaración sintacticamente correcta de alguna sentencia sin implementar nada en concreto

```
>>> while True:
...     pass  # Busy-wait for keyboard interrupt (Ctrl+
...

>>> class MyEmptyClass:
...     pass
...

>>> def initlog(*args):
...     pass   # Remember to implement this!
...
```

Sentencia **match** I

- La sentencia **match** se emplea para hacer *pattern matching* lo cuál compara una expresión con determinado valor para ejecutar una sección de código

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

- En Python, podemos utilizar esta sentencia para realizar la asignación de valores a las variables:

Sentencia **match** II

```
# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")
```

Declaración de funciones I

- Para definir una función se usa la palbra reservada **def** seguida del identificador de la función y la lista de argumentos formales entre paréntesis

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

- Argumentos por defecto

Declaración de funciones II

```
def ask_ok(prompt, retries=4, reminder='Please try again
while True:
    ok = input(prompt)
    if ok in ('y', 'ye', 'yes'):
        return True
    if ok in ('n', 'no', 'nop', 'nope'):
        return False
    retries = retries - 1
    if retries < 0:
        raise ValueError('invalid user response')
    print(reminder)
```

- Keywork arguments

Declaración de funciones III

```
def parrot(voltage, state='a stiff', action='voom', type
    print("— This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("— Lovely plumage, the", type)
    print("— It's", state, "!")
# 1 positional argument
parrot(1000)
# 1 keyword argument
parrot(voltage=1000)
# 2 keyword arguments
parrot(voltage=1000000, action='VOOOOOM')
# 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)
# 3 positional arguments
parrot('a million', 'bereft of life', 'jump')
# 1 positional, 1 keyword
parrot('a thousand', state='pushing up the daisies')
```

Declaración de funciones IV

- Argument lists

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

```
>>> def concat(*args, sep="/"):
...     return sep.join(args)
...
>>> concat("earth", "mars", "venus")
'earth/mars/venus'
>>> concat("earth", "mars", "venus", sep=".")
'earth.mars.venus'
```

- Unpacking arguments

Declaración de funciones V

```
>>> list(range(3, 6))                # normal call with sep
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))               # call with arguments
[3, 4, 5]

>>> def parrot(voltage, state='a stiff', action='voom')
...     print("-- This parrot wouldn't", action, end=' '
...     print("if you put", voltage, "volts through it.'
...     print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin'
>>> parrot(**d)
```

Lambda Expressions I

- Las expresiones **lambda** nos permiten definir funciones sin asignarles un identificador

```
>>> def make_incrementor(n):
...         return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

- Estas expresiones pueden ser utilizadas para evitar la definición de una función por medio de un nombre sino asignar directamente la expresión a utilizar en donde se requiera una función

Lambda Expressions II

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4,
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

Documentation Strings

- Las cadenas de documentación permiten asignar un texto descriptivo a las funciones en Python que se mostrará en los editores de código como parte de la definición de la función. Son útiles como documentación

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
...
>>> print(my_function.__doc__)
Do nothing, but document it.

    No, really, it doesn't do anything.
```

Listas I

Métodos para trabajar con listas

- list.append(x)
- list.extend(iterable)
- list.insert(i, x)
- list.remove(x)
- list.pop([i])
- list.clear()
- list.index(x[, start[, end]])
- list.count(x)
- list.sort(*, key=None, reverse=False)
- list.reverse()
- list.copy()

Listas II

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi',
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)  # Find next banana starting
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'ora
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'ora
>>> fruits.sort()
>>> fruits
```

Listas III

```
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'or
>>> fruits.pop()
'pear'
```

Listas por comprensión

Las listas por comprensión permiten definir nuevas listas a partir de la definición de las propiedades que deben cumplir los elementos de las listas.

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

squares = list(map(lambda x: x**2, range(10)))

squares = [x**2 for x in range(10)]

>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Tuplas y secuencias I

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
```

Tuplas y secuencias II

```
>>> v
([1, 2, 3], [3, 2, 1])

>>> t = (12345, 54321, 'hello!')
>>>
>>> x, y, z = t
```

Sets I

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange',
>>> print(basket)                      # show that duplicat
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership te
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but n
{'r', 'd', 'b'}
```

Sets II

```
>>> a | b                                    # letters in a or b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                                    # letters in both a
{'a', 'c'}
>>> a ^ b                                    # letters in a or b
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Diccionarios

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
```

Reading and Writing Files I

- Apertura de un archivo

```
>>> f = open('workfile', 'w', encoding="utf-8")
```

- Bloques

```
>>> with open('workfile', encoding="utf-8") as f:
...     read_data = f.read()

>>> # We can check that the file has been automatically
>>> f.closed
True
```

- Lectura

Reading and Writing Files II

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

- Escritura

```
>>> f.write('This is a test\n')
15
```

Archivos Zip

```
# importing required modules
from zipfile import ZipFile

# specifying the zip file name
file_name = "my_python_files.zip"

# opening the zip file in READ mode
with ZipFile(file_name, 'r') as zip:
    # printing all the contents of the zip file
    zip.printdir()

    # extracting all the files
    print('Extracting all the files now...')
    zip.extractall()
    print('Done!'
```

Class Objects

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'

x = MyClass()
```

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

Class and Instance Variables

```
class Dog:

    kind = 'canine'         # class variable shared by all

    def __init__(self, name):
        self.name = name    # instance variable unique to e

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # shared by all dogs
'canine'
>>> e.kind                  # shared by all dogs
'canine'
>>> d.name                  # unique to d
'Fido'
>>> e.name                  # unique to e
'Buddy'
```

Inheritance

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Módulos

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

Paquetes

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversi
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
```

Módulos de la biblioteca estándar

https://docs.python.org/3/tutorial/stdlib.html

Conclusiones

- Las características del lenguaje hacen que la programación en Python sea mas sencilla y compacta.
- La disponibilidad de bibliotecas de clases y funciones permite implementar casi cualquier funcionalidad en el lenguaje de una manera simple y flexible.
- El soporte por parte de los principales proveedores de servicios en la nube lo hacen un lenguaje adecuado para implementar diferentes programas de aplicación.
- Existe una diversidad muy amplia de bibliotecas de implementación de cómputo científico, aprendizaje automatizado, visión computacional, aprendizaje profundo, etc.