Redes Neuronales (2025)

Ayudas e indicaciones para el Trabajo Práctico N°1 (Trabajo en progreso)[*]

Índice

[*]Reportar errores a: tristan.osan@unc.edu.ar

1. Familiarizándose con Google Colab y Python

1.1. Ejercicio 1: Importando librerías

En Python, las librerías son conjuntos de funciones y herramientas que podemos usar para ampliar las capacidades del lenguaje. Para trabajar con arreglos numéricos y gráficos, usaremos:

- `numpy` para operar con arrays.
- `matplotlib.pyplot` para graficar.

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

Explicación:

- `import numpy as np`: Importa la librería `numpy` y le asigna el alias `np` para escribir menos.
- `import matplotlib.pyplot as plt`: Importa el módulo `pyplot` de `matplotlib` y le asigna el alias `plt`.

1.2. Ejercicio 2: Practicando instrucciones básicas

1. Comentarios:

```
# Esto es un comentario
```

2. Operaciones con enteros

```
3 + 2
5 * 4
10 // 3  # Division entera
```

3. Operaciones con flotantes:

```
3.5 + 2.1
5.0 * 4.2
10.0 / 3.0
```

4. String:

```
"Hola mundo"
```

5. Docstring:

```
"""Este es un docstring,
sirve para documentar el codigo."""
```

6. Concatenar strings:

```
"Hola" + " " + "mundo"
```

1.3. Ejercicio 3: Practicando con variables

Las variables guardan valores para reutilizarlos.

```
x = 3
x

y = 4
x + y

z = x * (2 + y)
z

s1 = "hola"
s2 = "mundo"
s = s1 + s2
s
```

1.4. Ejercicio 4: Practicando operaciones lógicas: igualdades y desigualdades

```
1 == 1
1 == 2
1 != 1
1 != 2
"hola" == "hola"
"hola" != "hola"
"hola" == "mundo"
"hola" != "mundo"
True == True
False == True
False != True
False == False
1 < 2
1 > 2
1 <= 2
1 >= 2
```

Nota: == compara igualdad, `!=` compara desigualdad, y <, >, <=, >= comparan valores numéricos.

1.5. Ejercicio 5: Practicando con la función print

La función `print()` muestra información en pantalla.

```
print(42)
print("Hola")
print(3, 7)
print("Hola", "mundo")
print("Hola", "mundo", sep="")
print(25, "a\~nos")
```

2. Ejercicio 6: Practicando con condicionales if, else y elif

En Python, las estructuras condicionales nos permiten tomar decisiones en el código.

```
x = 10

if x > 0:
    print("x es positivo")
elif x == 0:
    print("x es cero")
else:
    print("x es negativo")
```

Explicación:

- `if`: se ejecuta si la condición es verdadera.
- `elif`: "else if", se evalúa si la condición anterior fue falsa.
- `else`: se ejecuta si todas las condiciones anteriores son falsas.

3. Listas y tuplas

3.1. Introducción

En Python, las **listas** y **tuplas** son estructuras de datos que permiten almacenar colecciones de elementos. La diferencia principal es que:

- Las **listas** son **mutables**, es decir, pueden modificarse después de su creación.
- Las **tuplas** son **inmutables**, es decir, no pueden cambiarse una vez creadas.

3.2. Listas en Python

Creación de listas

Una lista se crea usando corchetes `[ ]` y separando los elementos con comas.

```
animales = ["perro", "gato", "loro", "caballo"]
numeros = [1, 2, 3, 4, 5]
mixta = ["hola", 42, 3.14, True]
```

Acceso a elementos

Los elementos se indexan desde 0.

```
print(animales[0])   # "perro"
print(numeros[2])    # 3
```

Modificación de listas

```
animales[1] = "gato domestico"
print(animales)
```

Slicing (sublistas)

```
print(animales[0:2])  # Desde indice 0 hasta 1
print(animales[-2:])  # Ultimos dos elementos
```

Métodos útiles de listas

```
animales.append("lobo")  # Agregar al final
animales.remove("loro")  # Eliminar elemento
print(len(animales))     # Largo de la lista
```

3.3. Tuplas en Python

Creación de tuplas

Una tupla se crea usando paréntesis `( )`.

```
coordenadas = (10, 20)
colores = ("rojo", "verde", "azul")
```

Acceso a elementos

```
print(coordenadas[0])  # 10
print(colores[2])      # "azul"
```

Inmutabilidad de las tuplas

```
coordenadas[0] = 15  # Error: las tuplas no se pueden modificar
```

3.4. Diferencias entre una Lista y una Tupla en Python

En Python, tanto las **listas** como las **tuplas** permiten almacenar colecciones de elementos, pero tienen diferencias clave:

- **Lista**: mutable, se pueden agregar, eliminar o modificar elementos después de creada. Se definen con corchetes `[ ]`.
- **Tupla**: inmutable, no se pueden cambiar sus elementos una vez creada. Se definen con paréntesis `( )`.

3.4.1. Ejemplo de Lista

```
# Lista: se pueden modificar los elementos
frutas = ["manzana", "banana", "cereza"]
print(frutas)       # ['manzana', 'banana', 'cereza']

frutas[1] = "naranja"  # Modificamos la segunda fruta
print(frutas)       # ['manzana', 'naranja', 'cereza']

frutas.append("pera")  # Agregamos una nueva fruta
print(frutas)       # ['manzana', 'naranja', 'cereza', 'pera']
```

3.4.2. Ejemplo de Tupla

```
# Tupla: los elementos no se pueden modificar
colores = ("rojo", "verde", "azul")
print(colores)      # ('rojo', 'verde', 'azul')

# Intentar modificar una tupla provoca un error:
colores[1] = "amarillo"  # TypeError: 'tuple' object does not support
   item assignment
```

3.5. Resumen rápido

	Lista	Tupla
Mutable	Sí	No
Sintaxis	[ ]	( )
Velocidad	Más lenta	Más rápida
Uso típico	Datos que cambian	Datos fijos

3.6. Ejercicio 7: Practicando con listas

Las listas almacenan múltiples valores en una sola variable.

```
animales = ["perro", "gato", "loro", "caballo", "llama"]

# Acceder al 3er elemento
print(animales[2])

# Slicing
print(animales[:3])    # primeros 3
print(animales[-3:])   # últimos 3
print(animales[1:4])   # desde el segundo hasta el cuarto

# List comprehension
con_l = [a for a in animales if "l" in a]
print(con_l)

# Lista de n\'umeros 1 al 10
a = list(range(1, 11))
```

```
# Modificar entradas
a[5] = "hola"
print(a)

a[2:6] = ["perro", "gato", "loro", "caballo"]
print(a)
```

3.7. Ejercicio 8: Practicando con tuplas

Las tuplas son similares a las listas, pero **inmutables**.

```
# Igualdades
print((1,2,3) == (1,2,3))
print((1,2,3) == (1,2,3,4))

# Indexaci\'on
t = (1,2,3)
print(t[0], t[1], t[2])

# Intentar modificar (error)
# t[0] = 10

# Comparaci\'on con lista
print((1,2,3) == [1,2,3])

# Desempaquetado
x, y, z = ("hola", "mundo", 100)
print(x, y, z)
```

4. Ejercicio 9: Practicando con loops

1. **For con números pares:**

```
for i in range(0, 11, 2):
    print(i)
```

2. **Conjetura de Collatz:**

```
n = 10
contador = 0

while n != 1:
    if n % 2 == 0:
        n = n // 2
    else:
        n = 3 * n + 1
    contador += 1
    print(n)

print("Iteraciones:", contador)
```

3. **Iterar sobre lista:**

```
for animal in ["perro", "gato", "loro", "caballo", "llama"]:
    print(animal)
```

5. Ejercicio 10: Definiendo y evaluando funciones

1. **Función suma:**

```
def suma(a, b):
    return a + b

print(suma(3, 4))
```

2. **Parámetro opcional:**

```
def opcional(a, b=1):
    return a - b

print(opcional(5))
print(opcional(5, 3))
```

3. **Factorial:**

```
def fact(n):
    if isinstance(n, int) and n >= 0:
        resultado = 1
        for i in range(1, n+1):
            resultado *= i
        return resultado
    else:
        print("El n\'umero n no es un entero no negativo.")
        return None

print(fact(5))
print(fact(-2))
```

6. Uso básico de la librería NumPy para manipular arrays

La librería `NumPy` es fundamental para el cálculo numérico en Python, ya que permite manejar arreglos multidimensionales (arrays) y realizar operaciones matemáticas eficientes.

6.1. Importar NumPy

Para usar `NumPy` primero debemos importarla, usualmente con el alias `np`:

```
import numpy as np
```

6.2. Crear arrays

- Crear un array a partir de una lista de Python:

```
a = np.array([1, 2, 3, 4])
print(a)            # Salida: [1 2 3 4]
```

- Crear un array 2D (matriz):

```
b = np.array([[1, 2], [3, 4]])
print(b)
# Salida:
# [[1 2]
#  [3 4]]
```

- Crear arrays con valores iniciales predeterminados:

```
ceros = np.zeros(5)        # Array de 5 ceros
unos = np.ones((2,3))      # Matriz 2x3 de unos
```

- Crear arrays con valores equiespaciados:

```
c = np.linspace(0, 10, 5)  # 5 valores entre 0 y 10
print(c)                    # [ 0.   2.5  5.   7.5 10. ]
```

6.3. Operaciones básicas con arrays

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

print(x + y)     # Suma elemento a elemento: [5 7 9]
print(x * y)     # Multiplicacion elemento a elemento: [4 10 18]
print(x - y)     # Resta: [-3 -3 -3]
print(x / y)     # Division: [0.25 0.4 0.5]
```

6.4. Indexación y slicing

```
arr = np.array([10, 20, 30, 40, 50])

print(arr[0])    # Primer elemento: 10
print(arr[1:4])  # Sub-array: [20 30 40]
print(arr[-2:])  # Ultimos dos elementos: [40 50]
```

6.5. Funciones útiles

```
arr = np.array([1, 2, 3, 4, 5])

print(np.sum(arr))     # Suma de todos los elementos: 15
print(np.mean(arr))    # Promedio: 3.0
print(np.max(arr))     # Maximo: 5
print(np.min(arr))     # Minimo: 1
```

NumPy facilita la manipulación y el cálculo con grandes conjuntos de datos numéricos gracias a su estructura eficiente de arrays y sus funciones optimizadas.

6.6. Ejercicio 11: Practicando con arrays (NumPy)

Los `arrays` son estructuras similares a listas, pero optimizadas para operaciones matemáticas. Usaremos la librería `NumPy`.

```
import numpy as np

# 1) Array de enteros
b = np.array([1, 2, 3])
print(b)

# 2) Array 2D
a = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(a)

# 3) Acceder a fila 1, columna 2 (\'indices desde 0)
print(a[1, 2])

# 4) Array de ceros (float64 por defecto)
zeros64 = np.zeros(10, dtype=np.float64)
print(zeros64)

# 5) Array de valores 3.14 (float32)
pi_array = np.full(10, 3.14, dtype=np.float32)
print(pi_array)
```

7. Ejercicio 12: Practicando con clases

En Python, las clases nos permiten crear nuestros propios tipos de datos con atributos y métodos.

```
class MiClase:
    def __init__(self, nombre="mi clase"):
        self.nombre = nombre

    def __str__(self):
        return self.nombre

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

    def cambiar_nombre(self, nuevo_nombre):
        self.nombre = nuevo_nombre

# Crear objeto
mi_objeto = MiClase()

# Test __str__()
print(str(mi_objeto))
```

```
# Test __len__()
print(len(mi_objeto))

# Cambiar nombre
mi_objeto.cambiar_nombre("nueva clase")
print(str(mi_objeto))
```

Explicación:

- `__init__`: método constructor.
- `__str__`: representación en texto.
- `__len__`: define la función `len()`.
- `self`: referencia al propio objeto.

8. Gráficos con Python

8.1. Introducción

`matplotlib` es una librería popular para crear gráficos en Python. Su módulo `pyplot` provee una interfaz sencilla para generar visualizaciones de datos.

8.2. Importar matplotlib.pyplot

La convención es importar `pyplot` como `plt`:

```
import matplotlib.pyplot as plt
```

8.3. Primer gráfico sencillo

Graficar la función $y = 2x + 1$:

```
x = [0, 1, 2, 3, 4, 5]
y = [1, 3, 5, 7, 9, 11]

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

8.4. Agregar etiquetas y título

```
plt.plot(x, y)
plt.xlabel("Eje X")
plt.ylabel("Eje Y")
plt.title("Gráfico de y = 2x + 1")
plt.show()
```

8.5. Personalizar líneas y marcadores

```
plt.plot(x, y, color="red", linestyle="--", marker="o")
plt.show()
```

8.6. Gráficos con múltiples curvas

```
import numpy as np

x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)

plt.plot(x, y1, label="sin(x)", color="blue")
plt.plot(x, y2, label="cos(x)", color="green", linestyle="--")
plt.xlabel("x")
plt.ylabel("y")
plt.title("Funciones seno y coseno")
plt.legend()
plt.grid(True)
plt.show()
```

8.7. Gráficos de dispersión (scatter)

```
x = [1, 2, 3, 4]
y = [10, 20, 25, 30]

plt.scatter(x, y, color="purple", marker="x")
plt.xlabel("Eje X")
plt.ylabel("Eje Y")
plt.title("Gráfico de dispersión")
plt.show()
```

8.8. Histogramas

```
data = [1,1,2,3,3,3,4,4,5,6,7,8,8,9]

plt.hist(data, bins=5, color="orange", edgecolor="black")
plt.xlabel("Valor")
plt.ylabel("Frecuencia")
plt.title("Histograma de datos")
plt.show()
```

8.9. Guardar gráficos

```
plt.plot(x, y)
plt.savefig("grafico_lineal.png")
plt.close()
```

8.10. Ejercicio 13: Graficando (Matplotlib)

Para graficar funciones matemáticas usaremos `NumPy` y `Matplotlib`.

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

# 1) Valores x
valores_x = np.linspace(0, 5, 100)

# 2) y1 = cos(x)
valores_y1 = np.cos(valores_x)

# 3) y2 = sin(x)
valores_y2 = np.sin(valores_x)

# 4) Graficar
plt.plot(valores_x, valores_y1, label="cos(x)", linestyle="-")
plt.plot(valores_x, valores_y2, label="sin(x)", linestyle="--")

# 5) Ajustes del gráfico
plt.xlabel("Eje X")
plt.ylabel("Eje Y")
plt.title("Funciones seno y coseno en [0,5]")
plt.legend()
plt.xlim(0, 5)
plt.grid(True)

# Mostrar
plt.show()
```

Notas:

- `np.linspace`: genera valores equidistantes.
- `plt.plot`: dibuja curvas.
- `label`: nombre para la leyenda.
- `linestyle="`: línea continua; `-"`: punteada.

8.11. Gráficos avanzados con `matplotlib.pyplot`

8.11.1. Múltiples gráficos en una figura (subplots)

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

x = np.linspace(0, 2*np.pi, 400)
y = np.sin(x**2)

fig, axs = plt.subplots(2, 2, figsize=(10,8))

axs[0, 0].plot(x, np.sin(x))
axs[0, 0].set_title('Seno')

axs[0, 1].plot(x, np.cos(x), 'tab:orange')
axs[0, 1].set_title('Coseno')

axs[1, 0].plot(x, y, 'tab:green')
axs[1, 0].set_title('Seno de x^2')

axs[1, 1].plot(x, np.tan(x), 'tab:red')
axs[1, 1].set_ylim([-10, 10])  # Limitar eje y
axs[1, 1].set_title('Tangente')

plt.tight_layout()  # Ajusta para que no se superpongan
plt.show()
```

Explicación: `plt.subplots` crea una figura con una grilla de ejes (aquí 2x2). Cada `axs[i,j]` es un gráfico independiente donde se puede dibujar. `figsize` define el tamaño en pulgadas. `tight_layout()` mejora la distribución para que las etiquetas no se corten.

8.11.2. Cambiar estilos y temas predefinidos

```
import matplotlib.pyplot as plt

print(plt.style.available)  # Lista de estilos disponibles

plt.style.use('ggplot')  # Cambia el estilo global a 'ggplot'

x = [1,2,3,4,5]
y = [1,4,9,16,25]

plt.plot(x, y)
plt.title('Ejemplo con estilo ggplot')
plt.show()
```

Explicación: `plt.style.use` cambia el tema general del gráfico para que luzca diferente sin cambiar código específico.

8.11.3. Personalizar colores, tamaños y fuentes

```
plt.plot(x, y, color='purple', linewidth=3, linestyle='-.', marker='s'
    , markersize=8)

plt.title('Título con fuente grande', fontsize=16, fontweight='bold',
    color='navy')
plt.xlabel('Eje X', fontsize=12)
plt.ylabel('Eje Y', fontsize=12)
plt.grid(True, linestyle=':', linewidth=0.7)
plt.show()
```

Opciones comunes:

- `color`: color de la línea.
- `linewidth`: grosor de la línea.
- `linestyle`: estilo de línea ('-', '–', '-.', ':').
- `marker`: símbolo en puntos ('o', 's', 'x', etc).
- `markersize`: tamaño del marcador.
- `fontsize`, `fontweight`, `color`: opciones para texto.

8.11.4. Anotar puntos específicos

```
x = np.linspace(0, 10, 100)
y = np.sin(x)

plt.plot(x, y)

# Anotar el máximo local
max_x = np.pi / 2
max_y = 1
plt.annotate('Máximo local',
             xy=(max_x, max_y), xycoords='data',
             xytext=(max_x+1, max_y-0.5), textcoords='data',
             arrowprops=dict(arrowstyle='->', color='red'))

plt.show()
```

Explicación: `plt.annotate` agrega texto con una flecha que apunta a una coordenada del gráfico.