Universidad Internacional de Valencia

Matemáticas para la IA

Ejercicios de aplicación

Jose Antonio Gómez Vargas

https://github.com/jagvgithub/Matematicas_IA/blob/main/Ejercicios.ipynb

Ejercicio 1 (2 puntos). Desarrollo de Laplace.

1. Deducir de la definición 4 el determinante en dimensión 0, 1 y 2.
2. A partir de la definición 4, expresar el determinante de una matriz cuadrada recursivamente en función de determinantes la matrices cuadradas de dimensión inferior. Indicación: para cada $n \in$ N, distribuir (por linealidad en las columnas) sobre la descomposición de una matriz cuadrada de dimensión n + 1, siendo $n \in$ N y

- λ un coeficiente real,
- v un vector de dimensión n (una columna de n coeficientes reales),
- ω un covector de la misma dimensión (una fila de n coeficientes),
- A una matriz cuadrada de la misma dimensión (con n 2 coeficientes), luego proceder del mismo modo con los demás coeficientes de esa primera columna, con cada columna.

3. Implementar en Python la definición así obtenida.

$$\begin{pmatrix} \lambda & w \\ v & A \end{pmatrix} = \begin{pmatrix} \lambda * 1 + 0 & w \\ \lambda * 0 + v & A \end{pmatrix}$$

$Ejercicio\ 1_1$

Dimensión 0

En dimensión 0, una matriz es una matriz vacía, ya que no tiene filas ni columnas. Por convención, el determinante de una matriz vacía es 1, porque no hay vectores involucrados, y la única función que puede cumplir con las propiedades dadas (linealidad y antisimetría) es la que asigna el valor 1.

$$\det(\text{matriz vacía}) = 1$$

Dimensión 1

En dimensión 1, una matriz 1×1 es simplemente un escalar a. La única función lineal antisimétrica de un solo vector que cumple la propiedad que el determinante de la matriz identidad sea 1 es el propio valor del único elemento de la matriz.

Sea $A = (\, a \,)$, entonces:

$\det(A) = a$

Para la matriz identidad $I = (\, 1 \,)$:

$\det(I) = 1$

Dimensión 2

En dimensión 2, una matriz 2×2 tiene la forma:

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

Para deducir el determinante, usaremos las propiedades de linealidad y antisimetría. El determinante debe ser una función lineal de los vectores columna y cambiar de signo si intercambiamos dos vectores columna.

Para la matriz identidad

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

, el determinante debe ser 1. Además, si intercambiamos dos filas o columnas, el signo del determinante debe cambiar.

La única función que satisface estas propiedades en dimensión 2 es:

$$\det(A) = ad - bc$$

Podemos verificar que esta fórmula cumple con las propiedades requeridas:

1. **Linealidad**: El determinante es lineal en cada columna.
2. **Antisimetría**: Si intercambiamos las dos columnas, el determinante cambia de signo:
$$\det \begin{pmatrix} b & a \\ d & c \end{pmatrix} = bc - ad = -(ad - bc)$$
3. **Determinante de la matriz identidad**:
$$\det \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = 1 \cdot 1 - 0 \cdot 0 = 1$$

$Ejercicio\ 1_2$

Deducción Recursiva del Determinante

Definición 4 (Determinante)

El determinante de una matriz en tanto lista de vectores es la única función lineal antisimétrica de estos vectores tal que la matriz de la función identidad tenga determinante uno.

Expansión por Cofactores

Para una matriz cuadrada (A) de dimensión (n+1):

$$A = \begin{pmatrix} \lambda & \omega \\ v & A \end{pmatrix}$$

donde:

- λ es un escalar.
- v es un vector columna de dimensión n.
- ω es un vector fila de dimensión n.
- A es una matriz cuadrada de dimensión $n \times n$.

La fórmula recursiva para el determinante es:

$$\det(A) = \sum_{j=1}^{n+1} (-1)^{1+j} a_{1j} \det(A_{1j})$$

donde a_{1j} es el elemento en la primera fila y j-ésima columna de A , y A_{1j} es la submatriz de A que resulta de eliminar la primera fila y la j-ésima columna.

Ejemplo para $n = 2$

Para una matriz 3×3:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

El determinante de A es:

$$\det(A) = a_{11} \det \begin{pmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix} - a_{12} \det \begin{pmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{pmatrix} + a_{13} \det \begin{pmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix}$$

Cada determinante de submatriz 2×2 se calcula como:

$$\det \begin{pmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix} = a_{22}a_{33} - a_{23}a_{32}$$

$$\det \begin{pmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{pmatrix} = a_{21}a_{33} - a_{23}a_{31}$$

$$\det \begin{pmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} = a_{21}a_{32} - a_{22}a_{31}$$

$Ejercicio\ 1_3$

```
import numpy as np

def determinant(matrix):
    # Base case for 1x1 matrix
    if len(matrix) == 1:
        return matrix[0, 0]

    # Base case for 2x2 matrix
    if len(matrix) == 2:
        return matrix[0, 0] * matrix[1, 1] - matrix[0, 1] * matrix[1, 0]

    # Recursive case for larger matrices
    det = 0
    for j in range(len(matrix)):
        submatrix = np.delete(np.delete(matrix, 0, axis=0), j, axis=1)
        cofactor = (-1) ** j * matrix[0, j]
        det += cofactor * determinant(submatrix)

    return det

# Ejemplo de uso:
A = np.array([
    [1, 2, 3],
    [0, 1, 4],
    [5, 6, 0]
])

print("El determinante de A es:", determinant(A))
```

```
El determinante de A es: 1
```

Ejercicio 2 (2 puntos). Eliminación de Gauss–Jordan.

1. Deducir de la definición 4 el efecto que tiene en el determinante de una matriz sumar a una de sus columnas una combinación lineal de las demás.
2. A partir de la definición 4, proponer una estrategia para triangularizar una matriz sin cambiar su determinante e implementar en Python una definición alternativa del determinante. Indicación: descomponer similarmente al ejercicio anterior.
3. implementar en Python la definición así obtenida.

Parte 1: Efecto en el Determinante al Sumar una Combinación Lineal de las Demás Columnas

---

De la definición 4, sabemos que el determinante es una función lineal antisimétrica. Esto significa que el determinante de una matriz cambia de signo si intercambiamos dos columnas y es lineal respecto a cada columna. Ahora, consideremos el efecto de sumar a una columna una combinación lineal de las demás columnas.

Sea A una matriz cuadrada y supongamos que sumamos una combinación lineal de las columnas $C_2, C_3, \ldots, C_n$ a la primera columna C_1. Es decir, la nueva columna C_1' es:

$$C_1' = C_1 + a_2 C_2 + a_3 C_3 + \ldots + a_n C_n$$

donde $a_2, a_3, \ldots, a_n$ son coeficientes escalares.

Por la linealidad del determinante, podemos escribir el determinante de la nueva matriz como la suma de determinantes de matrices donde cada matriz tiene una de las columnas $C_1, a_2 C_2, a_3 C_3, \ldots, a_n C_n$ en la primera columna y las demás columnas de A. Sin embargo, todas estas matrices con una columna múltiplo de otra tendrán determinante 0 debido a la propiedad de linealidad, excepto la matriz con la columna original C_1.

Por lo tanto, sumar una combinación lineal de otras columnas a una columna no cambia el determinante de la matriz.

Parte 2: Estrategia para Triangularizar una Matriz sin Cambiar su Determinante

---

Para triangularizar una matriz sin cambiar su determinante, podemos usar las siguientes operaciones elementales que no alteran el valor del determinante:

1. **Intercambiar dos filas o columnas**: Cambia el signo del determinante.
2. **Multiplicar una fila o columna por un escalar (k)**: Multiplica el determinante por (k).

3. **Sumar a una fila (o columna) una combinación lineal de las otras filas (o columnas)**: No cambia el determinante.

Usando la eliminación de Gauss-Jordan, nos limitaremos a usar la tercera operación (sumar combinaciones lineales de otras filas o columnas) para triangularizar la matriz. Esto garantiza que el determinante se mantenga invariante durante el proceso.

Parte 3: Implementación en python

```
import numpy as np

def gauss_jordan(matrix):
    n = len(matrix)
    for i in range(n):
        # Make the diagonal contain all 1's
        if matrix[i, i] == 0:
            for j in range(i+1, n):
                if matrix[j, i] != 0:
                    matrix[[i, j]] = matrix[[j, i]]
                    break

        # Normalize the pivot row
        pivot = matrix[i, i]
        matrix[i] = matrix[i] / pivot

        # Make all rows below the pivot have 0 in current column
        for j in range(i+1, n):
            matrix[j] -= matrix[j, i] * matrix[i]

    return matrix

def determinant(matrix):
    matrix = matrix.astype(float)  # Ensure floating-point division
    n = len(matrix)

    if n == 1:
        return matrix[0, 0]

    # Triangularize the matrix using Gauss-Jordan
    triangular_matrix = gauss_jordan(matrix)

    # The determinant is the product of the diagonal elements
    det = 1
    for i in range(n):
        det *= triangular_matrix[i, i]

    return det

# Ejemplo de uso:
A = np.array([
    [1, 2, 3],
    [0, 1, 4],
    [5, 6, 0]
])

print("El determinante de A es:", determinant(A))
```

```
El determinante de A es: 1.0
```

Ejercicio 3 (2 puntos). Comparación.

1. Obtener la complejidad computacional de cada una de estas dos implementaciones.
2. Generar matrices aleatoriamente en dimensión $n \in \{ 2, 3, \cdots, 9, 10 \}$ y comparar el tiempo de ejecución de cada una de estas dos implementaciones con la función numpy.linalg.det (la función determinante de la extensión numérica de Python al álgebra lineal). Indicación: se puede utilizar la función numpy.random.rand para generar los coeficientes aleatorios de sus matrices.

1. Complejidad Computacional

Implementación del Determinante por Expansión de Cofactores:

- La complejidad de esta implementación es $O(n!)$, donde n es el tamaño de la matriz cuadrada. Esto se debe a que el algoritmo utiliza una expansión de cofactores para calcular el determinante, lo que implica calcular determinantes de submatrices recursivamente. Como cada submatriz de tamaño $n \times n$ requiere calcular el determinante de una matriz de tamaño $(n-1) \times (n-1)$, el número total de operaciones crece factorialmente con n.

Implementación del Determinante por Gauss-Jordan:

- La complejidad de esta implementación es $O(n^3)$, donde n es el tamaño de la matriz cuadrada. La eliminación de Gauss-Jordan implica operaciones de pivoteo, normalización y cero en la matriz, que tienen una complejidad de $O(n^3)$.

2. Comparación de Tiempos de Ejecución

Para comparar el tiempo de ejecución de estas implementaciones con `numpy.linalg.det`, podemos generar matrices aleatorias de distintos tamaños y calcular el tiempo que tarda cada método en calcular el determinante de estas matrices.

```
import numpy as np
import time

def determinant_expansion(matrix):
    # Implementación del determinante por expansión de cofactores
    pass

def determinant_gauss_jordan(matrix):
    # Implementación del determinante por Gauss-Jordan
    pass

# Definir los tamaños de las matrices
sizes = range(2, 11)

# Repetir para cada tamaño de matriz
for n in sizes:
    # Generar matriz aleatoria
    matrix = np.random.rand(n, n)

    # Calcular tiempo de ejecución para la implementación de expansión de cofactores
    start_time = time.time()
    det_expansion = determinant_expansion(matrix)
    expansion_time = time.time() - start_time

    # Calcular tiempo de ejecución para la implementación de Gauss-Jordan
    start_time = time.time()
    det_gauss_jordan = determinant_gauss_jordan(matrix)
    gauss_jordan_time = time.time() - start_time

    # Calcular tiempo de ejecución para la función numpy.linalg.det
    start_time = time.time()
    det_numpy = np.linalg.det(matrix)
    numpy_time = time.time() - start_time

    # Imprimir resultados
    print(f"Tamaño de la matriz: {n}x{n}")
    print(f"Tiempo de ejecución (Expansión de Cofactores): {expansion_time:.6f} segundos")
    print(f"Tiempo de ejecución (Gauss-Jordan): {gauss_jordan_time:.6f} segundos")
    print(f"Tiempo de ejecución (NumPy): {numpy_time:.6f} segundos")
    print(f"Determinante (Expansión de Cofactores): {det_expansion}")
    print(f"Determinante (Gauss-Jordan): {det_gauss_jordan}")
    print(f"Determinante (NumPy): {det_numpy}")
    print()
```

```
Tamaño de la matriz: 2x2
Tiempo de ejecución (Expansión de Cofactores): 0.000005 segundos
Tiempo de ejecución (Gauss-Jordan): 0.000002 segundos
Tiempo de ejecución (NumPy): 0.004244 segundos
Determinante (Expansión de Cofactores): None
Determinante (Gauss-Jordan): None
Determinante (NumPy): -0.12621965147995126
```

```
Tamaño de la matriz: 3x3
Tiempo de ejecución (Expansión de Cofactores): 0.000003 segundos
Tiempo de ejecución (Gauss-Jordan): 0.000002 segundos
Tiempo de ejecución (NumPy): 0.000112 segundos
Determinante (Expansión de Cofactores): None
Determinante (Gauss-Jordan): None
Determinante (NumPy): -0.042464249137946074

Tamaño de la matriz: 4x4
Tiempo de ejecución (Expansión de Cofactores): 0.000002 segundos
Tiempo de ejecución (Gauss-Jordan): 0.000001 segundos
Tiempo de ejecución (NumPy): 0.000084 segundos
Determinante (Expansión de Cofactores): None
Determinante (Gauss-Jordan): None
Determinante (NumPy): -0.10975808496133849

Tamaño de la matriz: 5x5
Tiempo de ejecución (Expansión de Cofactores): 0.000003 segundos
Tiempo de ejecución (Gauss-Jordan): 0.000001 segundos
Tiempo de ejecución (NumPy): 0.000079 segundos
Determinante (Expansión de Cofactores): None
Determinante (Gauss-Jordan): None
Determinante (NumPy): 0.024453411149382858

Tamaño de la matriz: 6x6
Tiempo de ejecución (Expansión de Cofactores): 0.000001 segundos
Tiempo de ejecución (Gauss-Jordan): 0.000001 segundos
Tiempo de ejecución (NumPy): 0.000033 segundos
Determinante (Expansión de Cofactores): None
Determinante (Gauss-Jordan): None
Determinante (NumPy): 0.0001226965810081374

Tamaño de la matriz: 7x7
Tiempo de ejecución (Expansión de Cofactores): 0.000001 segundos
Tiempo de ejecución (Gauss-Jordan): 0.000001 segundos
Tiempo de ejecución (NumPy): 0.000025 segundos
Determinante (Expansión de Cofactores): None
Determinante (Gauss-Jordan): None
Determinante (NumPy): -0.07582678653114466

Tamaño de la matriz: 8x8
Tiempo de ejecución (Expansión de Cofactores): 0.000002 segundos
Tiempo de ejecución (Gauss-Jordan): 0.000000 segundos
Tiempo de ejecución (NumPy): 0.000029 segundos
Determinante (Expansión de Cofactores): None
Determinante (Gauss-Jordan): None
Determinante (NumPy): -0.08848228596631084

Tamaño de la matriz: 9x9
Tiempo de ejecución (Expansión de Cofactores): 0.000001 segundos
```

Ejercicio 4 (4 puntos).

Con el propósito de aproximar un mínimo local de una función real de varias variables reales, el método de descenso de gradiente consiste en iterar una marcha (positivamente) proporcional al (opuesto del) gradiente desde un valor inicial, con la intuición de 'seguir el agua' hasta dar con el valle.

1. Implementar en Python un algoritmo de descenso de gradiente (con un máximo de m = 10**5 iteraciones) a partir de los siguientes argumentos tomados en ese orden: la función f cuyo mínimo local se propone aproximar, el valor inicial x desde el que empieza la marcha, la razón geométrica o coeficiente de proporcionalidad y, el parámetro de tolerancia z para finalizar cuando el gradiente de la función f caiga dentro de esa tolerancia. Indicación: empezar por implementar el gradiente grad(f) de la función f.
2. Calcular formalmente { $t \in$ R. f ' (t) = 0 } para $f : t \mapsto\longrightarrow 3t$ 4+4t 3−12t 2+7.
3. Con una tolerancia z = 10**-12 y un valor inicial de x = 3 aplicar su algoritmo con razón y = 10**-1, 10**-2, 10**-3 luego hacer lo mismo con x = 0. Interpretar el resultado.
4. Repetir estos dos últimos apartados con $f : (s, t) \mapsto\longrightarrow s$ 2 + 3st + t 3 + 1 y los valores iniciales x = [-1,1], [0,0].

$Ejercicio\ 4_1$

```
import numpy as np

def gradient_descent(f, grad_f, x_initial, learning_rate, tolerance, max_iterations=10**5):
    x = x_initial
    iterations = 0

    while iterations < max_iterations:
        # Calcular el gradiente de f en el punto x
        gradient = grad_f(x)

        # Actualizar el valor de x
        x -= learning_rate * gradient

        # Verificar la condición de parada
        if np.linalg.norm(gradient) < tolerance:
            break

        iterations += 1

    return x, iterations

# Ejemplo de uso:
def f(x):
    return x**2

def grad_f(x):
    return 2 * x

x_initial = 5
learning_rate = 0.1
tolerance = 1e-6

x_min, num_iterations = gradient_descent(f, grad_f, x_initial, learning_rate, tolerance)
print("Mínimo local aproximado:", x_min)
print("Número de iteraciones:", num_iterations)
```

```
Mínimo local aproximado: 3.3699933333938316e-07
Número de iteraciones: 73
```

$Ejercicio\ 4_2$

---

2. Cálculo Formal de { $t \in \mathbb{R}$. f '(t) = 0 } para $f : t \mapsto 3t^4 + 4t^3 - 12t^2 + 7$.

Para encontrar los puntos donde $f'(t) = 0$ para $f(t) = 3t^4 + 4t^3 - 12t^2 + 7$, primero calculamos la derivada de $f(t)$ con respecto a t:

$$f(t) = 3t^4 + 4t^3 - 12t^2 + 7$$

Calculamos la derivada con respecto a t, denotada como $f'(t)$:

$$f'(t) = \frac{d}{dt}(3t^4 + 4t^3 - 12t^2 + 7) = 12t^3 + 12t^2 - 24t$$

Para encontrar los puntos donde $f'(t) = 0$, igualamos la derivada a cero y resolvemos para t:

$$12t^3 + 12t^2 - 24t = 0$$

Factorizamos t y obtenemos:

$$t(12t^2 + 12t - 24) = 0$$

Resolvemos la ecuación cuadrática $12t^2 + 12t - 24 = 0$ para t utilizando la fórmula cuadrática:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

donde $a = 12$, $b = 12$ y $c = -24$. Calculamos:

$$t = \frac{-12 \pm \sqrt{12^2 - 4 \cdot 12 \cdot (-24)}}{2 \cdot 12} = \frac{-12 \pm \sqrt{144 + 1152}}{24} = \frac{-12 \pm \sqrt{1296}}{24} = \frac{-12 \pm 3(}{24}$$

Esto nos da dos soluciones posibles:

$$t_1 = \frac{-12 + 36}{24} = \frac{24}{24} = 1$$
$$t_2 = \frac{-12 - 36}{24} = \frac{-48}{24} = -2$$

Por lo tanto, los puntos donde $f'(t) = 0$ son $t = 1$ y $t = -2$.

$Ejercicio\ 4_3$

```
import numpy as np

# Definición de la función y su derivada
def f(x):
    return 3 * x**4 + 4 * x**3 - 12 * x**2 + 7

def grad_f(x):
    return 12 * x**3 + 12 * x**2 - 24 * x

# Algoritmo de descenso de gradiente
def gradient_descent(f, grad_f, x_initial, learning_rate, tolerance, max_iterations=10**5):
    x = x_initial
    iterations = 0

    while iterations < max_iterations:
        gradient = grad_f(x)
        x -= learning_rate * gradient

        if np.abs(gradient) < tolerance:
            break

        iterations += 1

    return x, iterations

# Parámetros
tolerance = 1e-12
initial_values = np.array([3, 0], dtype=np.float128)
learning_rates = [1e-1, 1e-2, 1e-3]

# Aplicar el algoritmo para diferentes valores iniciales y tasas de aprendizaje
for x_initial in initial_values:
    print(f"\nValor inicial de x: {x_initial}\n")
    for learning_rate in learning_rates:
        x_min, num_iterations = gradient_descent(f, grad_f, x_initial, learning_rate, tolera
        print(f"Tasa de aprendizaje: {learning_rate}")
        print(f"Mínimo local aproximado: {x_min}")
        print(f"Número de iteraciones: {num_iterations}\n")
```

```
Valor inicial de x: 3.0

<ipython-input-7-3dddbf74f825>:8: RuntimeWarning: overflow encountered in scalar power
  return 12 * x**3 + 12 * x**2 - 24 * x
<ipython-input-7-3dddbf74f825>:8: RuntimeWarning: invalid value encountered in scalar ad
  return 12 * x**3 + 12 * x**2 - 24 * x
Tasa de aprendizaje: 0.1
Mínimo local aproximado: nan
Número de iteraciones: 100000

Tasa de aprendizaje: 0.01
```

```
Mínimo local aproximado: -1.9999999999999967
Número de iteraciones: 31

Tasa de aprendizaje: 0.001
Mínimo local aproximado: 1.0000000000000264
Número de iteraciones: 831
```

Ejercicio 4_4

```
Tasa de aprendizaje: 0.1
```

```
import numpy as np

# Definición de la función y su gradiente
def f(x):
    s, t = x
    return s**2 + 3 * s * t + t**3 + 1

def grad_f(x):
    s, t = x
    grad_s = 2 * s + 3 * t
    grad_t = 3 * s**2 + 3 * t**2
    return np.array([grad_s, grad_t])

# Algoritmo de descenso de gradiente
def gradient_descent(f, grad_f, x_initial, learning_rate, tolerance, max_iterations=10**5):
    x = x_initial.astype(np.float64)  # Asegurar que los valores iniciales sean del tipo flo
    iterations = 0

    while iterations < max_iterations:
        gradient = grad_f(x)
        x -= learning_rate * gradient

        if np.linalg.norm(gradient) < tolerance:
            break
```