import numpy as np import matplotlib.pyplot as plt from PIL import Image import glob, os, random

---------- helpers ----------

def cargar\_datos(path\_healthy, path\_parkinson, max\_imgs=300, tam=(64, 64)): healthy = glob.glob(os.path.join(path\_healthy, ".png"))[:max\_imgs] park = glob.glob(os.path.join(path\_parkinson, ".png"))[:max\_imgs]

X, y = [], []  
for f in healthy:  
 img = Image.open(f).convert("L").resize(tam)  
 X.append(np.asarray(img, dtype=np.float32).flatten()) # sin dividir por 255  
 y.append(0)  
for f in park:  
 img = Image.open(f).convert("L").resize(tam)  
 X.append(np.asarray(img, dtype=np.float32).flatten())  
 y.append(1)  
  
print(f"Total imágenes: {len(X)}")  
return np.array(X), np.array(y)

def dividir\_datos(X, y, test\_ratio=0.2): idx = np.arange(len(X)) np.random.shuffle(idx) n\_test = int(len(idx) \* test\_ratio) return X[idx[n\_test:]], X[idx[:n\_test]], y[idx[n\_test:]], y[idx[:n\_test]]

def normalizar\_datos(X\_train, X\_test): min\_vals = np.min(X\_train, axis=0) max\_vals = np.max(X\_train, axis=0) range\_vals = max\_vals - min\_vals range\_vals[range\_vals == 0] = 1 X\_train\_norm = (X\_train - min\_vals) / range\_vals X\_test\_norm = (X\_test - min\_vals) / range\_vals return X\_train\_norm, X\_test\_norm

---------- función f ----------

def f\_wb(x, w, b): z = np.dot(w, x) + b return (np.tanh(z) + 1) / 2

---------- pérdida y exactitud ----------

def loss(w, b, X, D): preds = np.array([f\_wb(xi, w, b) for xi in X]) return np.mean((preds - D) \*\* 2)

def accuracy(w, b, X, D): preds = np.array([f\_wb(xi, w, b) for xi in X]) preds\_bin = (preds >= 0.5).astype(int) return np.mean(preds\_bin == D)

---------- gradientes ----------

def grad\_L(w, b, X, D): grad\_w = np.zeros\_like(w) grad\_b = 0.0 for x\_i, d\_i in zip(X, D): z = np.dot(w, x\_i) + b tanh\_z = np.tanh(z) f = 0.5 \* (1 + tanh\_z) common\_term = (1 - tanh\_z\*\*2) \* (f - d\_i) grad\_w += common\_term \* x\_i grad\_b += common\_term return grad\_w, grad\_b

---------- descenso por gradiente ----------

def gradient\_descent(X\_train, D\_train, X\_test, D\_test, alpha, iterations=1000): K = X\_train.shape[1] w = np.random.randn(K) \* 0.01 b = 0.0

history = {  
 'train\_loss': [],  
 'train\_acc': [],  
 'test\_loss': [],  
 'test\_acc': []  
}  
  
for it in range(iterations):  
 grad\_w, grad\_b = grad\_L(w, b, X\_train, D\_train)  
 w -= alpha \* grad\_w  
 b -= alpha \* grad\_b  
  
 if it % 10 == 0 or it == iterations - 1:  
 train\_loss = loss(w, b, X\_train, D\_train)  
 train\_acc = accuracy(w, b, X\_train, D\_train)  
 test\_loss = loss(w, b, X\_test, D\_test)  
 test\_acc = accuracy(w, b, X\_test, D\_test)  
  
 history['train\_loss'].append(train\_loss)  
 history['train\_acc'].append(train\_acc)  
 history['test\_loss'].append(test\_loss)  
 history['test\_acc'].append(test\_acc)  
  
 print(f"Iter {it}: Train Loss={train\_loss:.6f}, Train Acc={train\_acc:.4f}, Test Loss={test\_loss:.6f}, Test Acc={test\_acc:.4f}")  
  
return w, b, history

---------- main ----------

def main(): path\_h = "../Healthy" path\_p = "../Parkinson"

X, y = cargar\_datos(path\_h, path\_p, max\_imgs=300, tam=(64, 64))  
X\_tr, X\_te, y\_tr, y\_te = dividir\_datos(X, y)  
  
print("\n🔴 Entrenando con datos SIN normalizar...")  
w, b, hist = gradient\_descent(X\_tr, y\_tr, X\_te, y\_te, alpha=1e-6, iterations=1000)  
  
print("\n🔵 Entrenando con datos NORMALIZADOS (min-max)...")  
X\_tr\_norm, X\_te\_norm = normalizar\_datos(X\_tr, X\_te)  
w\_n, b\_n, hist\_norm = gradient\_descent(X\_tr\_norm, y\_tr, X\_te\_norm, y\_te, alpha=1e-3, iterations=1000)  
  
# ---------- visualización ----------  
steps = np.arange(0, 1000, 10).tolist()  
if 999 not in steps:  
 steps.append(999) # incluir última iteración  
  
plt.figure(figsize=(16, 10))  
  
# Pérdida  
plt.subplot(2, 2, 1)  
plt.plot(hist['train\_loss'], label="Loss (sin normalizar)", color='red')  
plt.plot(hist\_norm['train\_loss'], label="Loss (normalizado)", color='blue')  
plt.title("Pérdida - Entrenamiento")  
plt.xlabel("Iteraciones (cada 10)")  
plt.ylabel("Loss")  
plt.legend()  
plt.grid(True)  
  
# Exactitud entrenamiento  
plt.subplot(2, 2, 2)  
plt.plot(hist['train\_acc'], label="Accuracy Train (sin normalizar)", color='red')  
plt.plot(hist\_norm['train\_acc'], label="Accuracy Train (normalizado)", color='blue')  
plt.title("Exactitud - Entrenamiento")  
plt.xlabel("Iteraciones (cada 10)")  
plt.ylabel("Accuracy")  
plt.ylim(0, 1)  
plt.legend()  
plt.grid(True)  
  
# Exactitud test  
plt.subplot(2, 2, 3)  
plt.plot(hist['test\_acc'], label="Accuracy Test (sin normalizar)", color='red')  
plt.plot(hist\_norm['test\_acc'], label="Accuracy Test (normalizado)", color='blue')  
plt.title("Exactitud - Test")  
plt.xlabel("Iteraciones (cada 10)")  
plt.ylabel("Accuracy")  
plt.ylim(0, 1)  
plt.legend()  
plt.grid(True)  
  
plt.tight\_layout()  
plt.show()  
  
# Últimos valores para el informe  
print("\n--- Comparación final ---")  
print(f"Sin normalizar - Test Accuracy final: {hist['test\_acc'][-1]:.4f}")  
print(f"Normalizado - Test Accuracy final: {hist\_norm['test\_acc'][-1]:.4f}")

if **name** == "**main**": main()

**Corrección conceptual menor:** en la línea:

python

CopiarEditar

X\_tr\_norm = X\_tr / 255.0  
X\_te\_norm = X\_te / 255.0

Ya habías normalizado antes al cargar las imágenes (/ 255.0), así que esto está redundante. Mejor usar normalizar\_datos() que hiciste:

python

CopiarEditar

X\_tr\_norm, X\_te\_norm = normalizar\_datos(X\_tr, X\_te)

------------------------

**Sugerencia adicional para análisis**:  
 Agregá un print final al main() con las últimas métricas para comparar fácil:

print("\n--- Comparación final ---")  
print(f"Normalizado - Test Accuracy final: {acc\_te\_hist\_norm[-1]:.4f}")  
print(f"Sin Normalizar - Test Accuracy final: {acc\_te\_hist[-1]:.4f}")

import matplotlib.pyplot as plt

1. Normalizar datos

X\_train\_norm = X\_train / 255.0 X\_test\_norm = X\_test / 255.0

2. Entrenar con datos normalizados

w\_norm, b\_norm, train\_loss\_norm, train\_acc\_norm, test\_acc\_norm = gradient\_descent( X\_train\_norm, D\_train, X\_test\_norm, D\_test, alpha=1e-7, iterations=1000 )

3. Comparar con resultados originales

Supongamos que ya tenías estos arrays del ejercicio 3:

train\_loss\_orig, train\_acc\_orig, test\_acc\_orig

4. Graficar comparación de Loss (entrenamiento)

plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1) plt.plot(train\_loss\_orig, label="Loss Train (sin normalizar)") plt.plot(train\_loss\_norm, label="Loss Train (normalizado)") plt.xlabel("Iteraciones (cada 10)") plt.ylabel("Loss") plt.title("Pérdida durante el entrenamiento") plt.legend()

5. Graficar comparación de Accuracy (testing)

plt.subplot(1, 2, 2) plt.plot(test\_acc\_orig, label="Accuracy Test (sin normalizar)") plt.plot(test\_acc\_norm, label="Accuracy Test (normalizado)") plt.xlabel("Iteraciones (cada 10)") plt.ylabel("Accuracy") plt.title("Exactitud en testeo") plt.legend()

plt.tight\_layout() plt.show()

Exactitud (accuracy): porcentaje de predicciones correctas

def accuracy(w, b, X, D): preds = np.array([f\_wb(xi, w, b) for xi in X]) preds\_labels = (preds >= 0.5).astype(int) return np.mean(preds\_labels == D)

Descenso por gradiente con tracking de métricas

def gradient\_descent\_track(X\_train, D\_train, X\_test, D\_test, alpha=1e-7, iterations=1000): K = X\_train.shape[1] w = np.random.randn(K) \* 0.01 b = 0.0

# Para guardar historia  
train\_losses, test\_losses = [], []  
train\_accuracies, test\_accuracies = [], []  
  
for \_ in range(iterations):  
 grad\_w, grad\_b = grad\_L(w, b, X\_train, D\_train)  
 w -= alpha \* grad\_w  
 b -= alpha \* grad\_b  
  
 # Guardar métricas  
 train\_losses.append(loss(w, b, X\_train, D\_train))  
 test\_losses.append(loss(w, b, X\_test, D\_test))  
 train\_accuracies.append(accuracy(w, b, X\_train, D\_train))  
 test\_accuracies.append(accuracy(w, b, X\_test, D\_test))  
  
return w, b, train\_losses, test\_losses, train\_accuracies, test\_accuracies

Visualización

def plot\_metrics(train\_losses, test\_losses, train\_acc, test\_acc): iterations = len(train\_losses)

plt.figure(figsize=(14, 5))  
  
# MSE  
plt.subplot(1, 2, 1)  
plt.plot(range(iterations), train\_losses, label="Train MSE")  
plt.plot(range(iterations), test\_losses, label="Test MSE")  
plt.title("Error cuadrático durante entrenamiento")  
plt.xlabel("Iteraciones")  
plt.ylabel("MSE")  
plt.legend()  
  
# Accuracy  
plt.subplot(1, 2, 2)  
plt.plot(range(iterations), train\_acc, label="Train Accuracy")  
plt.plot(range(iterations), test\_acc, label="Test Accuracy")  
plt.title("Accuracy durante entrenamiento")  
plt.xlabel("Iteraciones")  
plt.ylabel("Accuracy")  
plt.legend()  
  
plt.tight\_layout()  
plt.show()

### **¿Cómo lo usás?**

Luego de definir tus X\_train, D\_train, X\_test, D\_test, ejecutás:

python

Copiar código

w, b, train\_losses, test\_losses, train\_acc, test\_acc = gradient\_descent\_track(  
 X\_train, D\_train, X\_test, D\_test, alpha=1e-7, iterations=1000  
)  
  
plot\_metrics(train\_losses, test\_losses, train\_acc, test\_acc)

---------------------------------------------------------------------------------

def grad\_L(w, b, X, D): grad\_w = np.zeros\_like(w) grad\_b = 0.0

for x\_i, d\_i in zip(X, D):  
 z = np.dot(w, x\_i) + b  
 tanh\_z = np.tanh(z)  
 f = 0.5 \* (1 + tanh\_z) # f\_w,b(x\_i)  
 common\_term = (1 - tanh\_z\*\*2) \* (f - d\_i)  
   
 grad\_w += common\_term \* x\_i  
 grad\_b += common\_term  
  
return grad\_w, grad\_b

**La podés usar así dentro del bucle de entrenamiento:**

**dw, db = grad\_L(w, b, X\_train, D\_train)**  
**w -= alpha \* dw**  
**b -= alpha \* db**

import numpy as np

# Definir la función f\_w,b (la función de activación)

def f\_wb(x, w, b): z = np.dot(w, x) + b return (np.tanh(z) + 1) / 2

# Definir la función de pérdida cuadrática

def loss(w, b, X, D): preds = np.array([f\_wb(xi, w, b) for xi in X]) return np.sum((preds - D) \*\* 2)

# Calcular gradientes de w y b

def gradients(w, b, X, D): grad\_w = np.zeros\_like(w) grad\_b = 0.0

for xi, di in zip(X, D):  
 z = np.dot(w, xi) + b  
 f = (np.tanh(z) + 1) / 2  
 error = f - di  
 df\_dz = 0.5 \* (1 - np.tanh(z) \*\* 2)  
   
 grad\_w += 2 \* error \* df\_dz \* xi  
 grad\_b += 2 \* error \* df\_dz  
  
return grad\_w, grad\_b

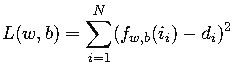
# Método de descenso por gradiente

def gradient\_descent(X, D, alpha=1e-7, iterations=1000): K = X.shape[1] w = np.random.randn(K) \* 0.01 # Inicialización aleatoria pequeña b = 0.0

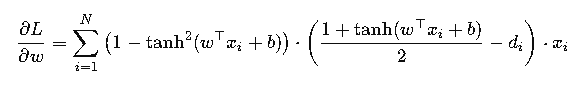
for it in range(iterations):  
 grad\_w, grad\_b = gradients(w, b, X, D)  
 w -= alpha \* grad\_w  
 b -= alpha \* grad\_b  
   
return w, b

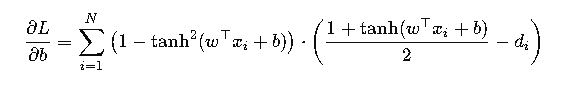
**Parte 1: Descenso de gradiente**

**Ejercicio 1. Calcular las derivadas parciales de la función L con respecto a w y b (pueden asistirse mediante** [**www.matrixcalculus.org**](http://www.matrixcalculus.org)**)**

  
donde:







**Ejercicio 2. Implementar el método de descenso por gradiente y optimizar los parámetros de la función f para el conjunto de datos de entrenamiento. Para esto les recomendamos que trabajen con un subconjunto de los datos que tenga una cantidad parecida de imágenes de dibujos de personas con y sin Párkinson.**

Para ejercicio 2 (sin normalizar), te dejo una versión mejorada del código, que incluye cálculo de error cuadrático y accuracy en cada iteración (opcional, para facilitar ejercicio 3):

import numpy as np

def f\_wb(x, w, b):

z = np.dot(w, x) + b

return (np.tanh(z) + 1) / 2

def loss(w, b, X, D):

preds = np.array([f\_wb(xi, w, b) for xi in X])

return np.mean((preds - D) \*\* 2)

def accuracy(w, b, X, D):

preds = np.array([f\_wb(xi, w, b) for xi in X])

preds\_bin = (preds >= 0.5).astype(int)

return np.mean(preds\_bin == D)

def gradients(w, b, X, D):

grad\_w = np.zeros\_like(w)

grad\_b = 0.0

for xi, di in zip(X, D):

z = np.dot(w, xi) + b

f = (np.tanh(z) + 1) / 2

error = f - di

df\_dz = 0.5 \* (1 - np.tanh(z)\*\*2)

grad\_w += 2 \* error \* df\_dz \* xi

grad\_b += 2 \* error \* df\_dz

grad\_w /= len(D)

grad\_b /= len(D)

return grad\_w, grad\_b

def gradient\_descent(X\_train, D\_train, X\_test, D\_test, alpha=1e-7, iterations=1000):

K = X\_train.shape[1]

w = np.random.randn(K) \* 0.01

b = 0.0

history = {

'train\_loss': [],

'train\_acc': [],

'test\_loss': [],

'test\_acc': []

}

for it in range(iterations):

grad\_w, grad\_b = gradients(w, b, X\_train, D\_train)

w -= alpha \* grad\_w

b -= alpha \* grad\_b

if it % 10 == 0 or it == iterations - 1:

train\_loss = loss(w, b, X\_train, D\_train)

train\_acc = accuracy(w, b, X\_train, D\_train)

test\_loss = loss(w, b, X\_test, D\_test)

test\_acc = accuracy(w, b, X\_test, D\_test)

history['train\_loss'].append(train\_loss)

history['train\_acc'].append(train\_acc)

history['test\_loss'].append(test\_loss)

history['test\_acc'].append(test\_acc)

print(f"Iter {it}: Train Loss={train\_loss:.4f}, Train Acc={train\_acc:.4f}, Test Loss={test\_loss:.4f}, Test Acc={test\_acc:.4f}")

return w, b, history

Comentarios importantes:

* El learning rate (α\alphaα) para datos sin normalizar suele ser mucho más pequeño porque los valores de entrada pueden ser grandes (hasta 255 por pixel), por eso puse un valor pequeño 1×10−71 \times 10^{-7}1×10−7 para que no explote el entrenamiento.
* En el ejercicio 4 con datos normalizados, ese α\alphaα se puede aumentar (por ejemplo a 10−310^{-3}10−3 o 10−410^{-4}10−4) y el entrenamiento será más rápido y estable.
* El parámetro history sirve para guardar datos y poder graficar la evolución de la pérdida y exactitud (ideal para ejercicio 3).

**Ejercicio 3. Calcular el error cuadrático y la exactitud (accuracy) durante la optimización para el conjunto de entrenamiento y para el conjunto de testing. Generar las visualizaciones correspondientes.**

Supuestos previos

* Ya tenés implementado el descenso por gradiente para entrenar el modelo (ejercicio 2).
* Tenés divididos los datos en conjunto de entrenamiento (X\_train, D\_train) y conjunto de testing (X\_test, D\_test).
* En la función de descenso por gradiente que te di en el ejercicio 2, ya guardamos en history las métricas para entrenamiento y testing.

Paso 1: Ejecutar el entrenamiento guardando métricas

Asumiendo que ya tenés los datos cargados y procesados en matrices X\_train, D\_train, X\_test, D\_test:

w\_opt, b\_opt, history = gradient\_descent(X\_train, D\_train, X\_test, D\_test, alpha=1e-7, iterations=1000)

Aquí history tiene:

* history['train\_loss'] → lista con pérdidas MSE en entrenamiento.
* history['train\_acc'] → lista con accuracies en entrenamiento.
* history['test\_loss'] → lista con pérdidas MSE en testing.
* history['test\_acc'] → lista con accuracies en testing.

Paso 2: Graficar las curvas de error y accuracy

Usamos matplotlib para visualizar:

import matplotlib.pyplot as plt

def plot\_metrics(history):

iterations = range(0, len(history['train\_loss'])\*10, 10) # porque guardamos cada 10 iteraciones

plt.figure(figsize=(14,6))

# Plot error cuadrático (MSE)

plt.subplot(1, 2, 1)

plt.plot(iterations, history['train\_loss'], label='Entrenamiento')

plt.plot(iterations, history['test\_loss'], label='Testing')

plt.title('Error cuadrático medio (MSE) durante la optimización')

plt.xlabel('Iteraciones')

plt.ylabel('MSE')

plt.legend()

plt.grid(True)

# Plot exactitud (accuracy)

plt.subplot(1, 2, 2)

plt.plot(iterations, history['train\_acc'], label='Entrenamiento')

plt.plot(iterations, history['test\_acc'], label='Testing')

plt.title('Exactitud (Accuracy) durante la optimización')

plt.xlabel('Iteraciones')

plt.ylabel('Accuracy')

plt.legend()

plt.grid(True)

plt.show()

# Llamar función para graficar

plot\_metrics(history)

Resumen

* Ejecutamos el descenso por gradiente (ejercicio 2) para obtener los pesos y bias optimizados.
* En cada paso guardamos MSE y accuracy para train y test.
* Graficamos esas métricas para observar evolución, posibles sobreajustes, convergencia, etc.

Resultado esperado:

* El error debe disminuir con el tiempo.
* La accuracy debe aumentar hasta estabilizarse.

**Ejercicio 4. Compare el resultado anterior si se entrena el modelo con datos normalizados, es decir, con valores entre 0 y 1.**

1. Normalizar los datos

Si tus datos están en RGB (0 a 255), primero pasalos a escala de grises o bien simplemente dividí todos los valores por 255 para que estén entre 0 y 1:

X\_train\_normalized = X\_train / 255.0

X\_test\_normalized = X\_test / 255.0

Si ya transformaste las imágenes a escala de grises o vectorizaste, la división se hace igual.

2. Reentrenar el modelo con datos normalizados

Usá la misma función de entrenamiento que antes, pero ahora con X\_train\_normalized y X\_test\_normalized:

w\_norm, b\_norm, history\_norm = gradient\_descent(

X\_train\_normalized, D\_train, X\_test\_normalized, D\_test, alpha=1e-6, iterations=1000

)

Nota que acá podés probar un learning rate (α) más grande porque la normalización suele hacer que el gradiente sea más estable.

3. Graficar ambas curvas juntas para comparer

import matplotlib.pyplot as plt

iterations = range(0, len(history['train\_loss'])\*10, 10) # asumiendo guardado cada 10 iteraciones

plt.figure(figsize=(14, 6))

# Error cuadrático medio (MSE)

plt.subplot(1, 2, 1)

plt.plot(iterations, history['train\_loss'], label='Entrenamiento sin normalizar')

plt.plot(iterations, history['test\_loss'], label='Testing sin normalizar')

plt.plot(iterations, history\_norm['train\_loss'], '--', label='Entrenamiento normalizado')

plt.plot(iterations, history\_norm['test\_loss'], '--', label='Testing normalizado')

plt.title('MSE - Sin normalizar vs Normalizado')

plt.xlabel('Iteraciones')

plt.ylabel('MSE')

plt.legend()

plt.grid(True)

# Exactitud (Accuracy)

plt.subplot(1, 2, 2)

plt.plot(iterations, history['train\_acc'], label='Entrenamiento sin normalizar')

plt.plot(iterations, history['test\_acc'], label='Testing sin normalizar')

plt.plot(iterations, history\_norm['train\_acc'], '--', label='Entrenamiento normalizado')

plt.plot(iterations, history\_norm['test\_acc'], '--', label='Testing normalizado')

plt.title('Accuracy - Sin normalizar vs Normalizado')

plt.xlabel('Iteraciones')

plt.ylabel('Accuracy')

plt.legend()

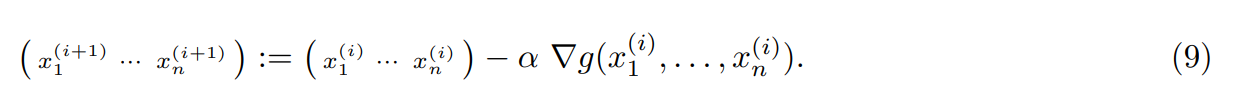
plt.grid(True)

plt.show()

4. Qué deberías observar

* Con datos normalizados:
  + Las curvas de error (MSE) disminuyen más rápido.
  + La exactitud (accuracy) sube más rápido y puede ser igual o mejor que sin normalizar.
  + Se puede usar un learning rate más grande sin que el entrenamiento “explote” o diverja.
  + Generalmente, el entrenamiento es más estable y converge mejor.

**Ejercicio 5. Analizar el impacto del par´ametro α (Ecuaci´on 9) en la convergencia del m´etodo. Tomar un rango de 5 valores posibles y analizar la convergencia para el conjunto de testing para los distintos valores de α. Graficar resultados**



1. Elegir un rango de valores de α\alphaα

Por ejemplo:

alphas = [1e-7, 1e-6, 5e-6, 1e-5, 5e-5]

Estos valores cubren tasas de aprendizaje desde muy pequeñas hasta medianas.

2. Para cada α\alphaα:

* Entrenar el modelo usando el descenso por gradiente con ese α\alphaα.
* Guardar la evolución del error cuadrático medio (MSE) para el conjunto de testing.

Ejemplo de pseudocódigo:

results = {}

for alpha in alphas:

w, b, history = gradient\_descent(

X\_train\_normalized, D\_train,

X\_test\_normalized, D\_test,

alpha=alpha, iterations=1000

)

results[alpha] = history['test\_loss'] # Guardamos la evolución del error de testing

3. Graficar la evolución del MSE en testing para cada α\alphaα

import matplotlib.pyplot as plt

plt.figure(figsize=(10, 6))

for alpha in alphas:

plt.plot(results[alpha], label=f'α = {alpha}')

plt.title('Convergencia del descenso por gradiente para distintos valores de α')

plt.xlabel('Iteraciones')

plt.ylabel('MSE en testing')

plt.legend()

plt.grid(True)

plt.show()

4. Qué observar

* Para valores muy pequeños de α, la curva disminuirá lentamente, el entrenamiento será muy lento.
* Para valores óptimos, la curva disminuirá rápido y convergerá bien.
* Para valores demasiado grandes, el error puede no disminuir o incluso explotar (subir de forma abrupta), indicando divergencia.

**Ejercicio 6. ¿Cómo impacta el tamaño del escalado de las imágenes en la efectividad del método? ¿Y en el tiempo de cómputo? Realizar los experimentos y gráficos acordes para estudiar estas limitaciones.**

Plan para abordar el ejercicio:

1. Escalar las imágenes a diferentes tamaños

Por ejemplo, podés elegir distintos tamaños cuadrados:

* 16x16
* 32x32
* 64x64
* 128x128

(Depende del tamaño original de tus imágenes)

Para cada tamaño:

* Redimensionar todas las imágenes (train y test) a ese tamaño.
* Vectorizar las imágenes para que queden como vectores de características.

2. Entrenar el modelo con cada tamaño de imagen

Usando la misma configuración (por ejemplo, misma tasa de aprendizaje, mismo número de iteraciones).

Registrar:

* Métrica de efectividad (accuracy, error cuadrático medio) en train y test.
* Tiempo que tarda en entrenar.

3. Graficar resultados

* Accuracy / Error vs tamaño de imagen (mostrar cómo cambia la efectividad).
* Tiempo de cómputo vs tamaño de imagen (mostrar cómo crece el costo computacional).

Ejemplo de código (conceptual)

import time

from skimage.transform import resize # Para redimensionar imágenes

sizes = [16, 32, 64, 128]

accuracies = []

times = []

for size in sizes:

# Redimensionar imágenes (asumiendo X\_train, X\_test son arrays de imágenes)

X\_train\_resized = np.array([resize(img, (size, size), anti\_aliasing=True) for img in X\_train])

X\_test\_resized = np.array([resize(img, (size, size), anti\_aliasing=True) for img in X\_test])

# Vectorizar

X\_train\_vec = X\_train\_resized.reshape(len(X\_train\_resized), -1)

X\_test\_vec = X\_test\_resized.reshape(len(X\_test\_resized), -1)

# Normalizar

X\_train\_vec = X\_train\_vec / 255.0

X\_test\_vec = X\_test\_vec / 255.0

start\_time = time.time()

w, b, history = gradient\_descent(

X\_train\_vec, D\_train, X\_test\_vec, D\_test,

alpha=1e-6, iterations=1000

)

elapsed = time.time() - start\_time

# Guardar tiempo y accuracy final en test

times.append(elapsed)

accuracies.append(history['test\_acc'][-1]) # Última accuracy del test

# Graficar

import matplotlib.pyplot as plt

plt.figure(figsize=(12,5))

plt.subplot(1, 2, 1)

plt.plot(sizes, accuracies, marker='o')

plt.title('Accuracy vs Tamaño de imagen')

plt.xlabel('Tamaño de imagen (px)')

plt.ylabel('Accuracy en test')

plt.grid(True)

plt.subplot(1, 2, 2)

plt.plot(sizes, times, marker='o', color='red')

plt.title('Tiempo de cómputo vs Tamaño de imagen')

plt.xlabel('Tamaño de imagen (px)')

plt.ylabel('Tiempo (segundos)')

plt.grid(True)

plt.show()

¿Qué deberías esperar?

* A mayor tamaño de imagen:
  + Mayor cantidad de información, por lo que la accuracy podría mejorar hasta cierto punto.
  + Sin embargo, a partir de cierto tamaño, puede haber pérdidas marginales (sobreajuste, ruido).
  + El tiempo de cómputo crecerá cuadráticamente (porque el número de píxeles crece como tamaño²).
* Por eso, existe un compromiso entre precisión y tiempo computacional.

**Ejercicio 7. Para el valor de α que tenga mejor valor de convergencia, generar la matriz de confusión y analizar brevemente la efectividad del método.**

Paso 1: Usar el valor de α\alphaα con mejor convergencia

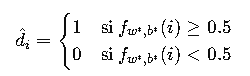
Del análisis del ejercicio 5, seleccionás el valor de α\alphaα que mostró la mejor y más rápida disminución del error sin explotar.

Paso 2: Entrenar el modelo con ese α\alphaα hasta convergencia o número fijo de iteraciones

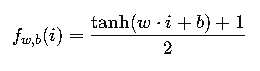
Guardá los parámetros finales w∗,b∗w^\*, b^\*w∗,b∗.

Paso 3: Predecir las etiquetas para el conjunto de testing

La función para predecir la clase binaria sería algo así:



Donde



Paso 4: Calcular la matriz de confusión

La matriz de confusión tiene la forma:

|  | Predicho Positivo (1) | Predicho Negativo (0) |
| --- | --- | --- |
| Verdadero Positivo (1) | TP | FN |
| Verdadero Negativo (0) | FP | TN |

Podés usar sklearn para calcularla y mostrarla:

from sklearn.metrics import confusion\_matrix, ConfusionMatrixDisplay

import matplotlib.pyplot as plt

# Predicciones binarias para test

y\_pred = (f\_wb(X\_test, w\_star, b\_star) >= 0.5).astype(int)

# Matriz de confusión

cm = confusion\_matrix(D\_test, y\_pred)

# Mostrar matriz

disp = ConfusionMatrixDisplay(confusion\_matrix=cm, display\_labels=[0,1])

disp.plot(cmap=plt.cm.Blues)

plt.title(f'Matriz de confusión para α = {alpha\_best}')

plt.show()

Paso 5: Análisis breve

* TP (Verdaderos Positivos): Pacientes con Parkinson correctamente detectados.
* TN (Verdaderos Negativos): Personas sanas correctamente detectadas.
* FP (Falsos Positivos): Personas sanas clasificadas erróneamente como con Parkinson.
* FN (Falsos Negativos): Pacientes con Parkinson no detectados.

Comentarios para el análisis:

* Si hay muchos FN, el método pierde pacientes con Parkinson (peligroso clínicamente).
* Si hay muchos FP, genera alarmas innecesarias (puede aumentar costos y ansiedad).
* Un buen método tiene TP y TN altos, y FP y FN bajos.