

APELLIDOS Y NOMBRES: Pacheco Taboada André Joaquín CÓDIGO: 20222189G

**Programación Paralela**

PRACTICA CALIFICADA Nº 5

05/12/2024

TRABAJO DE INVESTIGACIÓN

Respecto al tema de su proyecto de investigación,

* Control de Avance + Exposiciones Preliminares. *El trabajo de Investigación fue presentado durante las primeras semanas del curso*.
* Realizar la computación serial/paralela de su proyecto de investigación usando un repositorio de almacenamiento en disco (RAD) con organización estructurada, trabajado durante varias semanas en el curso. *Un RAD permite simular la representación de estructura de datos en memoria, tales como arrays, árboles o grafos en memoria, haciendo posible sobre todo el almacenamiento de grandes volúmenes de datos*.
* Descripción detallada de la solución del punto anterior

Ante todo, por la naturalidad del proyecto, se usó la herramienta de versionamiento de código Github para el proyecto. Puede revisar mi proyecto en este link: <https://github.com/A-PachecoT/brain-mri-classification>

Toda la documentación y código está ahí, comentado; incluyendo los detalles del RAD y de las técnicas de paralelismo aplicadas del paper; y por último los pasos para replicar el proyecto. De todas maneras copiaré el contenido en este documento.

Este proyecto se lleva desarrollando desde la solución propuesta en el examen parcial, con el mismo paper: [Parallel and Distributed Graph Neural Networks: An In-Depth Concurrency Analysis](https://arxiv.org/pdf/2205.09702)

# Clasificación de Resonancias Magnéticas Cerebrales

Un proyecto de aprendizaje profundo para clasificar resonancias magnéticas cerebrales y detectar tumores usando TensorFlow.

## Conjunto de Datos

El conjunto de datos utilizado en este proyecto es [Brain MRI Images for Brain Tumor Detection](https://www.kaggle.com/datasets/navoneel/brain-mri-images-for-brain-tumor-detection/data) de Kaggle. Contiene:

- 253 resonancias magnéticas cerebrales en total

- 155 resonancias con tumores (casos positivos)

- 98 resonancias sin tumores (casos negativos)

- Todas las imágenes están en formato JPG

El conjunto de datos está organizado en dos carpetas:

- `yes/`: Contiene resonancias magnéticas con tumores

- `no/`: Contiene resonancias magnéticas sin tumores

Cada resonancia es una imagen en escala de grises que muestra una vista transversal del cerebro. Las imágenes han sido preprocesadas y se ha eliminado el cráneo para enfocarse en el tejido cerebral donde pueden estar presentes los tumores.

## Repositorio de Almacenamiento en Disco (RAD)

El proyecto implementa un sistema de almacenamiento estructurado en disco (RAD) para gestionar eficientemente grandes volúmenes de datos de imágenes médicas. Esta implementación permite simular estructuras de datos en memoria mientras mantiene los datos en disco, optimizando el uso de recursos.

### Características del RAD

#### Estructura de Almacenamiento

* **Formato HDF5**: Utiliza el formato jerárquico HDF5 para almacenar datos multidimensionales
* **Metadatos JSON**: Mantiene un índice de metadatos para acceso rápido y mapeo de datos
* **Almacenamiento Estructurado**:

data/processed/

├── image\_data.h5 # Datos de imágenes y etiquetas

└── metadata.json # Índices y metadatos

#### Gestión de Memoria

* **Carga por Lotes**: Procesamiento de imágenes en mini-batches de 32 muestra
* **Memoria Dinámica**: Datasets redimensionables para crecimiento eficiente
* **Indexación Eficiente**: Sistema de mapeo para acceso rápido a datos

#### Optimizaciones

1. **I/O Eficiente**:

- Escritura secuencial por lotes

- Lectura paralela con ThreadPoolExecutor

- Buffer de prefetch para datos frecuentes

2. **Gestión de Recursos**:

- Liberación automática de memoria

- Cierre seguro de archivos

- Manejo de excepciones robusto

3. **Escalabilidad**:

- Soporte para conjuntos de datos grandes

- Crecimiento dinámico de datasets

- Procesamiento paralelo de E/S

## Detalles Técnicos

### Paralelismo y Optimización

El proyecto implementa múltiples niveles de paralelismo basados en los principios descritos en [Parallel and Distributed Graph Neural Networks: An In-Depth Concurrency Analysis](https://arxiv.org/pdf/2205.09702) , adaptados para el procesamiento de imágenes médicas:

#### Paralelismo de Datos

- **Mini-batch Processing**: Implementación de procesamiento por lotes para optimizar el uso de memoria y mejorar la convergencia

- Tamaño de lote: 32 imágenes

- Prefetch buffer para solapar CPU/GPU

- Caché de datos en memoria para acceso rápido

#### Paralelismo de Hardware

- **Multi-GPU Training**:

- Estrategia `MirroredStrategy` de TensorFlow para distribución automática

- Sincronización de gradientes entre GPUs

- Escalado dinámico del tamaño de lote según GPUs disponibles

#### Optimizaciones de Rendimiento

1. **Carga de Datos**:

- ThreadPoolExecutor para carga paralela de imágenes

- Número de workers igual a CPU cores disponibles

- Pipeline de datos optimizado con `tf.data.AUTOTUNE`

2. **Computación**:

- Precisión mixta (float16/float32) para acelerar cálculos

- Optimización automática de operaciones tensoriales

- Gestión dinámica de memoria GPU

3. **Predicción**:

- Predicción en paralelo usando ThreadPoolExecutor

- Batch processing para inferencia

- Balanceo automático de carga entre GPUs

## Estructura del Proyecto

.

├── artifacts/ # Artefactos generados

│ ├── models/ # Modelos guardados

│ └── plots/ # Gráficos y visualizaciones generadas

├── data/ # Directorio de datos

│ ├── processed/ # Conjunto de datos procesado

│ └── raw/ # Conjunto de datos sin procesar

│ ├── yes/ # Resonancias con tumores

│ └── no/ # Resonancias sin tumores

├── notebooks/ # Notebooks de Jupyter

├── src/ # Código fuente

│ ├── data/ # Carga y preprocesamiento de datos

│ ├── models/ # Arquitectura del modelo

│ ├── utils/ # Funciones de utilidad

│ └── main.py # Script principal

└── requirements.txt # Dependencias del proyecto

## Configuración

1. Crear un entorno virtual:

conda create -n mri-classification python=3.8

conda activate mri-classification

2. Instalar dependencias:

pip install -r requirements.txt

## Uso

Ejecutar el pipeline de entrenamiento:

python src/main.py

## Arquitectura del Modelo

El modelo utiliza una arquitectura CNN con:

- 4 capas convolucionales

- Capas de max pooling

- Dropout para regularización

- Capas densas para clasificación

## Características de Rendimiento

- Soporte para entrenamiento multi-GPU con MirroredStrategy

- Entrenamiento con precisión mixta para cálculos más rápidos

- Carga y preprocesamiento de datos en paralelo

- ThreadPoolExecutor para predicciones en paralelo

- Optimización automática de hardware

## Resultados

El modelo alcanza:

- 75% de precisión para casos sin tumor

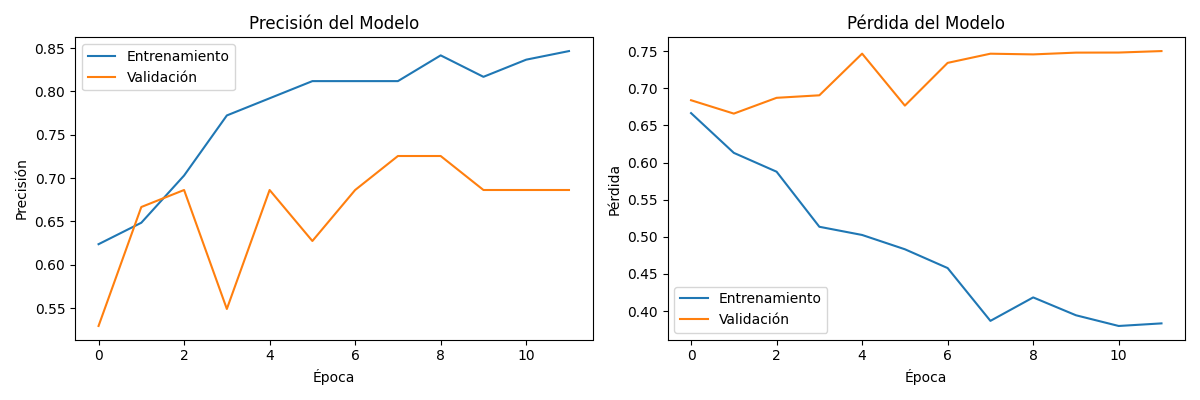
- 71% de precisión para casos con tumor

- 73% de precisión general

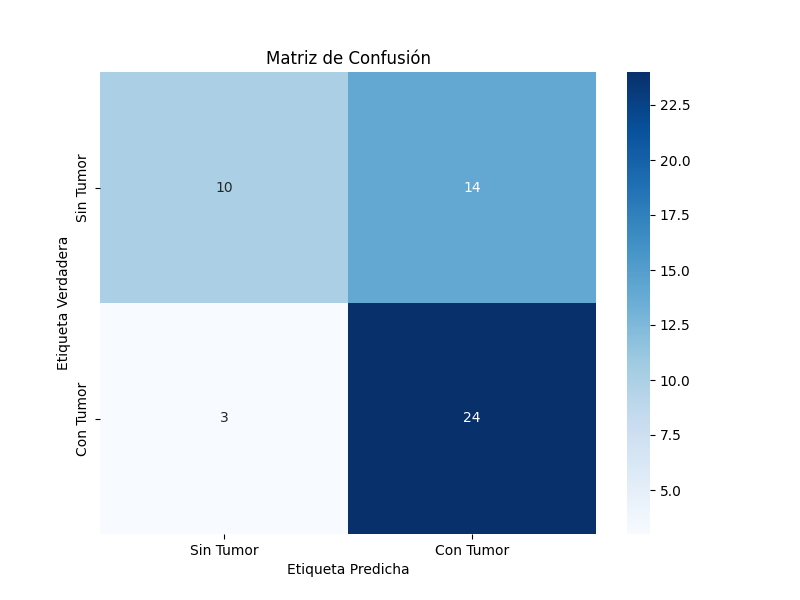
Los resultados se guardan en:

- Pesos del modelo: `artifacts/models/best\_model.h5`

- Gráficos de entrenamiento: `artifacts/plots/training\_history.png`



- Matriz de confusión: `artifacts/plots/confusion\_matrix.png`



## Créditos

Este proyecto está basado en el notebook de Jupyter de [Anirudh Bansal](https://www.kaggle.com/anibansal) con el dataset de Kaggle [Brain MRI Classification](https://www.kaggle.com/code/anibansal/brain-mri-classification) . El trabajo original ha sido reestructurado en un paquete Python apropiado con mejor organización y modularidad; y se optimizaron varios procesos para mejorar el rendimiento y la velocidad, usando técnicas de paralelización y optimización de hardware (ejemplo: multi-threading con GPU)

# Código

main.py

import os

import tensorflow as tf

from data.data\_loader import load\_data, create\_dataset

from models.model import create\_model

from utils.train import train\_model, plot\_training\_history

from utils.evaluate import evaluate\_model, plot\_confusion\_matrix

*# ========================*

*# Configuración Principal*

*# ========================*

def main():

*# Establecer semillas aleatorias para reproducibilidad*

tf.random.set\_seed(42)

*# Habilitar crecimiento de memoria para GPUs*

for gpu in tf.config.list\_physical\_devices("GPU"):

tf.config.experimental.set\_memory\_growth(gpu, True)

*# ========================*

*# Carga de Datos*

*# ========================*

*# Cargar y preprocesar datos con procesamiento paralelo*

print("Loading data...")

X\_train, X\_test, y\_train, y\_test = load\_data(data\_dir="data/raw")

*# Crear datasets optimizados con procesamiento paralelo*

print("Creating datasets...")

train\_dataset = create\_dataset(X\_train, y\_train, is\_training=True)

test\_dataset = create\_dataset(X\_test, y\_test, is\_training=False)

*# ========================*

*# Modelo y Entrenamiento*

*# ========================*

*# Crear y compilar modelo con soporte multi-GPU*

print("Creating model...")

model = create\_model()

model.summary()

*# Entrenar el modelo*

print("\nTraining model...")

history = train\_model(

model, train\_dataset, test\_dataset, checkpoint\_dir="artifacts/models"

)

*# ========================*

*# Visualización y Evaluación*

*# ========================*

*# Graficar historial de entrenamiento*

print("Plotting training history...")

plot\_training\_history(history, save\_path="artifacts/plots/training\_history.png")

*# Evaluar el modelo*

print("\nEvaluating model...")

results = evaluate\_model(model, test\_dataset)

*# Graficar matriz de confusión*

print("Plotting confusion matrix...")

plot\_confusion\_matrix(

results["y\_true"],

results["y\_pred"],

save\_path="artifacts/plots/confusion\_matrix.png",

)

print("\nTraining and evaluation completed!")

print("Check 'artifacts/plots' directory for visualizations.")

if \_\_name\_\_ == "\_\_main\_\_":

main()

data\_loader.py:

import os

import numpy as np

from PIL import Image

from sklearn.model\_selection import train\_test\_split

import tensorflow as tf

from concurrent.futures import ThreadPoolExecutor

import multiprocessing

import h5py

import json

from pathlib import Path

import shutil

*# ========================*

*# Repositorio de Almacenamiento en Disco (RAD)*

*# ========================*

class DiskStorageRepository:

"""

Implementación de un Repositorio de Almacenamiento en Disco (RAD) para gestionar

grandes volúmenes de datos de imágenes médicas de manera eficiente.

"""

def \_\_init\_\_(self, base\_path="data/processed"):

self.base\_path = Path(base\_path)

self.metadata\_file = self.base\_path / "metadata.json"

self.data\_file = self.base\_path / "image\_data.h5"

self.initialize\_storage()

def initialize\_storage(self):

"""Inicializar la estructura del almacenamiento."""

self.base\_path.mkdir(parents=True, exist\_ok=True)

if not self.metadata\_file.exists():

self.\_save\_metadata({"num\_samples": 0, "indices": {}})

def \_save\_metadata(self, metadata):

"""Guardar metadatos en disco."""

with open(self.metadata\_file, "w") as f:

json.dump(metadata, f)

def \_load\_metadata(self):

"""Cargar metadatos desde disco."""

with open(self.metadata\_file, "r") as f:

return json.load(f)

def store\_batch(self, images, labels, batch\_indices):

"""

Almacenar un lote de imágenes y etiquetas en el RAD.

Args:

images: Array de imágenes numpy

labels: Array de etiquetas

batch\_indices: Lista de índices para las imágenes

"""

metadata = self.\_load\_metadata()

current\_size = 0

with h5py.File(self.data\_file, "a") as f:

if "images" not in f:

f.create\_dataset(

"images", data=images, maxshape=(None, \*images.shape[1:])

)

f.create\_dataset("labels", data=labels, maxshape=(None,))

else:

current\_size = f["images"].shape[0]

new\_size = current\_size + len(images)

f["images"].resize(new\_size, axis=0)

f["labels"].resize(new\_size, axis=0)

f["images"][current\_size:] = images

f["labels"][current\_size:] = labels

for idx, orig\_idx in enumerate(batch\_indices):

metadata["indices"][str(orig\_idx)] = current\_size + idx

metadata["num\_samples"] = len(metadata["indices"])

self.\_save\_metadata(metadata)

def get\_batch(self, indices):

"""

Recuperar un lote de imágenes y etiquetas del RAD.

Args:

indices: Lista de índices a recuperar

Returns:

tuple: (imágenes, etiquetas)

"""

metadata = self.\_load\_metadata()

storage\_indices = [metadata["indices"][str(i)] for i in indices]

with h5py.File(self.data\_file, "r") as f:

images = f["images"][storage\_indices]

labels = f["labels"][storage\_indices]

return images, labels

def clear\_storage(self):

"""Limpiar todo el almacenamiento en disco."""

if self.data\_file.exists():

self.data\_file.unlink()

if self.metadata\_file.exists():

self.metadata\_file.unlink()

self.initialize\_storage()

*# ========================*

*# Funciones de Carga*

*# ========================*

def load\_single\_image(args):

"""

Cargar y preprocesar una única imagen.

Args:

args (tuple): (img\_path, img\_size, label)

Returns:

tuple: (image\_array, label)

"""

img\_path, img\_size, label = args

try:

img = Image.open(img\_path).convert("RGB")

img = img.resize(img\_size)

img\_array = np.array(img) / 255.0

return img\_array, label

except Exception as e:

print(f"Error al cargar la imagen {img\_path}: {e}")

return None, None

def load\_data(data\_dir="images", img\_size=(128, 128), batch\_size=32):

"""

Cargar y preprocesar imágenes de resonancia magnética usando procesamiento paralelo

y almacenamiento en disco.

"""

*# Inicializar el RAD*

rad = DiskStorageRepository()

rad.clear\_storage() *# Limpiar almacenamiento anterior*

*# Preparar rutas de imágenes y etiquetas*

image\_data = []

*# Verificar que el directorio existe*

if not os.path.exists(data\_dir):

raise FileNotFoundError(f"El directorio {data\_dir} no existe")

*# Recolectar rutas para imágenes positivas (con tumor)*

yes\_path = os.path.join(data\_dir, "yes")

if os.path.exists(yes\_path):

for img\_name in os.listdir(yes\_path):

image\_data.append((os.path.join(yes\_path, img\_name), img\_size, 1))

*# Recolectar rutas para imágenes negativas (sin tumor)*

no\_path = os.path.join(data\_dir, "no")

if os.path.exists(no\_path):

for img\_name in os.listdir(no\_path):

image\_data.append((os.path.join(no\_path, img\_name), img\_size, 0))

if not image\_data:

raise ValueError(f"No se encontraron imágenes en {data\_dir}")

*# Procesar y almacenar imágenes en lotes usando RAD*

num\_workers = multiprocessing.cpu\_count()

for i in range(0, len(image\_data), batch\_size):

batch\_data = image\_data[i : i + batch\_size]

with ThreadPoolExecutor(max\_workers=num\_workers) as executor:

results = list(executor.map(load\_single\_image, batch\_data))

*# Filtrar resultados válidos*

valid\_results = [(img, label) for img, label in results if img is not None]

if valid\_results:

batch\_images, batch\_labels = zip(\*valid\_results)

batch\_images = np.array(batch\_images)

batch\_labels = np.array(batch\_labels)

batch\_indices = range(i, i + len(valid\_results))

rad.store\_batch(batch\_images, batch\_labels, batch\_indices)

*# Cargar todos los datos del RAD*

metadata = rad.\_load\_metadata()

all\_indices = list(range(metadata["num\_samples"]))

if not all\_indices:

raise ValueError("No se pudieron cargar imágenes válidas")

X, y = rad.get\_batch(all\_indices)

*# Dividir los datos*

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, y, test\_size=0.2, random\_state=42

)

return X\_train, X\_test, y\_train, y\_test

*# ========================*

*# Creación de Dataset*

*# ========================*

def create\_dataset(X, y, batch\_size=32, is\_training=True):

"""

Crear un dataset de TensorFlow optimizado con procesamiento paralelo.

Args:

X (np.ndarray): Datos de imágenes

y (np.ndarray): Etiquetas

batch\_size (int): Tamaño del lote

is\_training (bool): Si es un dataset de entrenamiento

Returns:

tf.data.Dataset: Dataset de TensorFlow optimizado

"""

*# Calcular el número de llamadas paralelas*

AUTOTUNE = tf.data.AUTOTUNE

*# Crear el dataset*

dataset = tf.data.Dataset.from\_tensor\_slices((X, y))

if is\_training:

*# Cachear el dataset en memoria para mejor rendimiento*

dataset = dataset.cache()

*# Mezclar con un buffer lo suficientemente grande para asegurar buena aleatorización*

dataset = dataset.shuffle(buffer\_size=1000)

*# Configurar procesamiento por lotes en paralelo*

dataset = dataset.batch(batch\_size)

*# Precargar siguiente lote mientras se procesa el actual*

dataset = dataset.prefetch(AUTOTUNE)

return dataset

model.py:

import tensorflow as tf

from tensorflow.keras import layers, models

*# ========================*

*# Definición del Modelo*

*# ========================*

def create\_model(input\_shape=(128, 128, 3)):

"""

Crear un modelo CNN con soporte multi-GPU.

"""

*# Configurar estrategia de distribución*

strategy = (

tf.distribute.MirroredStrategy()

if len(tf.config.list\_physical\_devices("GPU")) > 1

else tf.distribute.get\_strategy()

)

with strategy.scope():

*# ========================*

*# Arquitectura CNN*

*# ========================*

model = models.Sequential(

[

*# Capas convolucionales y de pooling*

layers.Conv2D(32, (3, 3), activation="relu", input\_shape=input\_shape),

layers.MaxPooling2D((2, 2)),

layers.Conv2D(64, (3, 3), activation="relu"),

layers.MaxPooling2D((2, 2)),

layers.Conv2D(64, (3, 3), activation="relu"),

layers.MaxPooling2D((2, 2)),

layers.Conv2D(128, (3, 3), activation="relu"),

layers.MaxPooling2D((2, 2)),

*# Capas densas y dropout*

layers.Flatten(),

layers.Dense(128, activation="relu"),

layers.Dropout(0.5),

layers.Dense(64, activation="relu"),

layers.Dropout(0.3),

layers.Dense(1, activation="sigmoid"),

]

)

*# Usar precisión mixta para cálculos más rápidos*

optimizer = tf.keras.optimizers.Adam()

optimizer = tf.keras.mixed\_precision.LossScaleOptimizer(optimizer)

model.compile(

optimizer=optimizer, loss="binary\_crossentropy", metrics=["accuracy"]

)

return model

train.py

import tensorflow as tf

from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping

import matplotlib.pyplot as plt

import os

*# ========================*

*# Configuración GPU*

*# ========================*

def setup\_multi\_gpu():

"""

Configurar estrategia multi-GPU si está disponible.

Returns:

tf.distribute.Strategy: Estrategia de distribución

"""

try:

*# Verificar GPUs disponibles*

gpus = tf.config.list\_physical\_devices("GPU")

if len(gpus) > 1:

strategy = tf.distribute.MirroredStrategy()

print(f"Entrenando usando {len(gpus)} GPUs")

else:

strategy = tf.distribute.get\_strategy() *# Estrategia por defecto*

print("Entrenando usando estrategia por defecto")

return strategy

except:

return tf.distribute.get\_strategy() *# Estrategia por defecto*

*# ========================*

*# Entrenamiento*

*# ========================*

def train\_model(

model, train\_dataset, val\_dataset, epochs=50, checkpoint\_dir="artifacts/models"

):

"""

Entrenar el modelo con soporte de procesamiento paralelo.

"""

*# Crear directorio de checkpoints si no existe*

os.makedirs(checkpoint\_dir, exist\_ok=True)

*# Habilitar entrenamiento con precisión mixta para cálculos más rápidos*

tf.keras.mixed\_precision.set\_global\_policy("mixed\_float16")

*# Optimizar rendimiento del dataset*

AUTOTUNE = tf.data.AUTOTUNE

train\_dataset = train\_dataset.prefetch(AUTOTUNE)

val\_dataset = val\_dataset.prefetch(AUTOTUNE)

*# Callbacks con procesamiento paralelo*

checkpoint\_path = os.path.join(checkpoint\_dir, "best\_model.h5")

callbacks = [

ModelCheckpoint(

checkpoint\_path,

monitor="val\_accuracy",

save\_best\_only=True,

mode="max",

verbose=1,

),

EarlyStopping(

monitor="val\_loss", patience=10, restore\_best\_weights=True, verbose=1

),

tf.keras.callbacks.ReduceLROnPlateau(

monitor="val\_loss", factor=0.2, patience=5, min\_lr=1e-6

),

]

*# Entrenar el modelo*

history = model.fit(

train\_dataset,

validation\_data=val\_dataset,

epochs=epochs,

callbacks=callbacks,

workers=os.cpu\_count(), *# Carga de datos en paralelo*

use\_multiprocessing=True,

)

return history

*# ========================*

*# Visualización*

*# ========================*

def plot\_training\_history(history, save\_path="artifacts/plots/training\_history.png"):

"""

Graficar historial de entrenamiento mostrando curvas de precisión y pérdida.

Args:

history: Historial de entrenamiento de model.fit()

save\_path (str): Ruta para guardar el gráfico

"""

*# Crear directorio si no existe*

os.makedirs(os.path.dirname(save\_path), exist\_ok=True)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

*# Gráfico de precisión*

ax1.plot(history.history["accuracy"], label="Entrenamiento")

ax1.plot(history.history["val\_accuracy"], label="Validación")

ax1.set\_title("Precisión del Modelo")

ax1.set\_xlabel("Época")

ax1.set\_ylabel("Precisión")

ax1.legend()

*# Gráfico de pérdida*

ax2.plot(history.history["loss"], label="Entrenamiento")

ax2.plot(history.history["val\_loss"], label="Validación")

ax2.set\_title("Pérdida del Modelo")

ax2.set\_xlabel("Época")

ax2.set\_ylabel("Pérdida")

ax2.legend()

plt.tight\_layout()

plt.savefig(save\_path)

plt.close()

evaluate.py

import numpy as np

import matplotlib.pyplot as plt

from sklearn.metrics import confusion\_matrix, classification\_report

import seaborn as sns

import os

import tensorflow as tf

from concurrent.futures import ThreadPoolExecutor

*# ========================*

*# Funciones de Evaluación*

*# ========================*

def parallel\_predict(model, images\_batch):

"""

Realizar predicciones en paralelo para un lote de imágenes.

"""

return model.predict(images\_batch)

def evaluate\_model(model, test\_dataset):

"""

Evaluar el modelo en datos de prueba con procesamiento paralelo.

"""

AUTOTUNE = tf.data.AUTOTUNE

test\_dataset = test\_dataset.prefetch(AUTOTUNE)

*# Obtener predicciones usando procesamiento paralelo*

y\_pred = []

y\_true = []

*# Crear un pool de hilos para predicción paralela*

with ThreadPoolExecutor(max\_workers=os.cpu\_count()) as executor:

futures = []

for images, labels in test\_dataset:

futures.append(executor.submit(parallel\_predict, model, images))

y\_true.extend(labels.numpy())

*# Recolectar resultados*

for future in futures:

predictions = future.result()

y\_pred.extend(predictions.flatten() > 0.5)

*# Convertir a arrays numpy*

y\_pred = np.array(y\_pred)

y\_true = np.array(y\_true)

*# Imprimir reporte de clasificación*

print("\nReporte de Clasificación:")

print(

classification\_report(y\_true, y\_pred, target\_names=["Sin Tumor", "Con Tumor"])

)

return {"y\_true": y\_true, "y\_pred": y\_pred}

*# ========================*

*# Visualización*

*# ========================*

def plot\_confusion\_matrix(

y\_true, y\_pred, save\_path="artifacts/plots/confusion\_matrix.png"

):

"""

Graficar matriz de confusión.

Args:

y\_true (np.ndarray): Etiquetas verdaderas

y\_pred (np.ndarray): Etiquetas predichas

save\_path (str): Ruta para guardar el gráfico

"""

*# Crear directorio si no existe*

os.makedirs(os.path.dirname(save\_path), exist\_ok=True)

cm = confusion\_matrix(y\_true, y\_pred)

plt.figure(figsize=(8, 6))

sns.heatmap(

cm,

annot=True,

fmt="d",

cmap="Blues",

xticklabels=["Sin Tumor", "Con Tumor"],

yticklabels=["Sin Tumor", "Con Tumor"],

)

plt.title("Matriz de Confusión")

plt.ylabel("Etiqueta Verdadera")

plt.xlabel("Etiqueta Predicha")

plt.savefig(save\_path)

plt.close()