UNIVERSIDAD NACIONAL DE INGENIERÍA
FACULTAD DE CIENCIAS
ESCUELA PROFESIONAL DE CIENCIA DE LA COMPUTACION

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

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 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] , 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

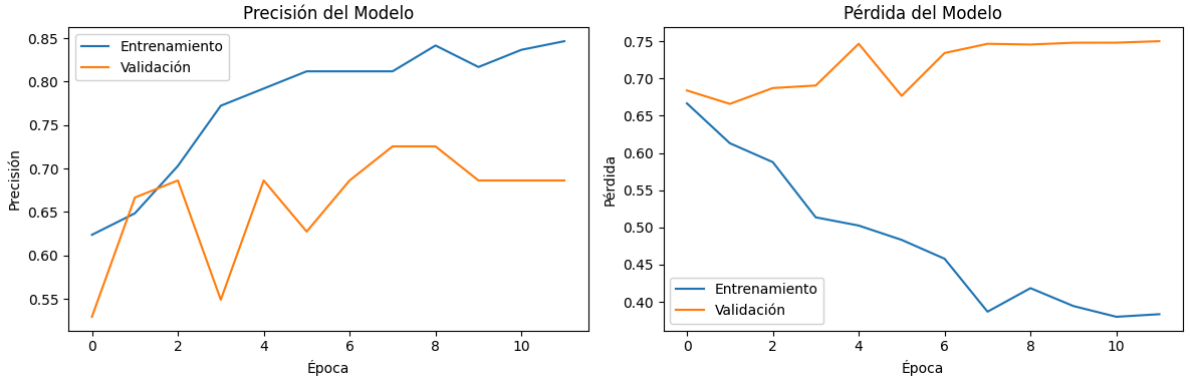
Características de Rendimiento

- 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

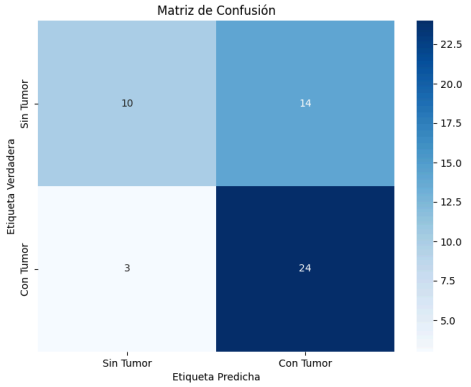
Resultados

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 con el dataset de Kaggle 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

```python
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
# ========================
```

```python
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:

```python
import tensorflow as tf
from tensorflow.keras import layers, models

# ========================
# Definición del Modelo
```

```python
# ========================


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
```

```python
# ========================
# 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()
```