MANUAL DE INSTALACION

Integrantes: Dany Molina, Kevin Gómez, Joseph Jiménez

Descripción del Proyecto

El repositorio Blackbox S es una implementación en C de una "caja negra" numérica (Black-Box) diseñada para encapsular y ejecutar modelos o algoritmos sin revelar su interior.

1. Prerrequisitos

Antes de comenzar, se debe tener instalado:

- Python (versión 3.x recomendada)
- pip (gestor de paquetes de Python)

Verifica las versiones ejecutando:

```
python –version
pip –versión
```

2. Preparación del Entorno.

Es buena práctica usar un entorno virtual para aislar las dependencias del proyecto:

Para Windows:

```
python -m venv venv
venv\Scripts\activate
```

3. Ubicación del Archivo requirements.txt

Navega al directorio del proyecto donde se encuentra el archivo requirements.txt:

```
cd ruta/al/proyecto
```

4. Instalación de Dependencias

Ejecuta el siguiente comando para instalar todas las dependencias listadas en el archivo:

```
pip install -r requirements.txt
```

4.1. Contenido del Archivo

El archivo requirements.txt especifica las siguientes dependencias con sus versiones exactas:

```
keras==3.9.2
numpy==2.2.6
```

Dependencias:

4.1.1. Keras 3.9.2

Keras es una API de redes neuronales de alto nivel, escrita en Python y capaz de ejecutarse sobre TensorFlow.

4.1.2. NumPy 2.2.6

Descripción: Biblioteca fundamental para computación científica en Python.

5. Verificación de la Instalación

```
pip show keras numpy
```

Análisis del Código: Modelo de Caja Negra (Blackbox)

Objetivo general del código:

Este programa carga un modelo entrenado en Keras (blackbox_S.keras) para realizar predicciones sobre entradas numéricas. Está diseñado como un sistema de **caja negra**, es decir, se desconoce el modelo interno, pero se puede observar su comportamiento frente a distintas entradas.

Secciones del Código y Su Funcionalidad

1. Importación de librerías necesarias

```
import logging

from sys import stdout

from datetime import datetime

import os

import numpy as np

from tensorflow import keras
```

logging, datetime: permiten registrar eventos e información útil (para depuración o trazabilidad).

os: interactúa con el sistema de archivos.

numpy: manipulación de vectores/matrices, esencial para procesar datos de entrada.

keras: se utiliza para cargar y ejecutar el modelo de red neuronal previamente entrenado.

2. **Configuración de Logging**

```
import __main__

logging.basicConfig(
   level=logging.INFO,
   format="[%(asctime)s][%(levelname)s] %(message)s",
   stream=stdout,
   datefmt="%m-%d %H:%M:%S",
)
```

Esta sección configura el formato y nivel de los mensajes de registro.

Si se ejecuta en un script, muestra el nombre del archivo; si es un entorno interactivo (como Jupyter), lo indica también.

El datetime.now() registra la fecha y hora actual de ejecución.

3. **Funciones para trabajar con el modelo**

load_model

```
def load_model(model_path: str = "Blackbox/blackbox_S.keras") -> keras.Sequential:
   logging.debug(f"Loading model from {model_path}")
   print("Current working directory:", os.getcwd())
   return keras.models.load_model(model_path)
```

Carga un modelo entrenado desde la ruta especificada.

Usa os.getcwd() para verificar desde qué ruta se está ejecutando el script.

predict_point

```
def predict_point(model: keras.Sequential, x1: float, x2: float) -> float:
    # Validación de tipos
    if not isinstance(x1, (int, float)):
        raise TypeError("x1 must be a number")
    if not isinstance(x2, (int, float)):
        raise TypeError("x2 must be a number")

    # Predicción para un único punto
    result = model.predict(np.array([[x1, x2]], dtype=np.float32))
    return float(np.round(result).flatten()[0])
```

Recibe dos entradas x1 y x2, las combina en un arreglo y las envía al modelo.

Retorna un valor **predicho y redondeado**, que representa la salida de la red neuronal para esa entrada específica.

predict_batch

```
def predict_batch(model: keras.Sequential, x1s: list, x2s: list) -> list:
    if not isinstance(x1s, list) or not isinstance(x2s, list):
        raise TypeError("Inputs must be lists")

    array_input = np.array([x1s, x2s], dtype=np.float32).T
    return model.predict(array_input).round().flatten().tolist()
```

Procesa varios pares de entrada simultáneamente.

Combina dos listas x1s y x2s en una matriz de forma [(x1, x2), (x1, x2), ...] y predice para todos ellos.

Devuelve una lista de predicciones redondeadas.

Configuraciones especiales para entorno Jupyter

```
%load_ext autoreload

%autoreload 2
```

Estos comandos son exclusivos de Jupyter.

Permiten que las modificaciones hechas a los módulos importados (como Blackbox) se recarguen automáticamente sin tener que reiniciar el kernel.

5. Carga del modelo y pruebas de predicción

```
model = keras.models.load_model('Blackbox/blackbox_S.keras')

predict_point(model, 0.2, 0.4)

predict_point(model, 1.1, 1.3)

predict_batch(model, [0.2, 1.1], [0.4, 1.3])
```

Se carga el modelo entrenado.

Se hacen predicciones puntuales y por lotes.

¿Qué hace el modelo?

Dado que es una caja negra, no se conoce su estructura interna, pero podemos inferir que:

- Recibe dos entradas numéricas (x1 y x2).
- Devuelve un valor que probablemente representa una clasificación binaria (0 o 1), ya que se utiliza .round() para redondear las salidas.

Qué es un archivo .keras?

Es un archivo que guarda un modelo completo de Keras, incluyendo:

- La arquitectura del modelo (secuencia de capas).
- Los pesos entrenados.
- La configuración de entrenamiento (optimizador, función de pérdida, métricas, etc.).

- Puede ser cargado directamente sin necesidad de redefinir el modelo manualmente.

¿Cómo lo analizas en la máquina?

```
from tensorflow import keras

# Cargar el modelo (usar la ruta correcta)

modelo = keras.models.load_model("Blackbox/blackbox_S.keras")

# Mostrar resumen del modelo

modelo.summary()

# Ver tipo y forma de entrada esperada

print("Entrada esperada:", modelo.input_shape)

print("Salida esperada:", modelo.output_shape)

# Ver los nombres de capas (opcional)

for i, layer in enumerate(modelo.layers):

  # Usar layer.output.shape para obtener la forma de salida de manera segura

  output_shape = getattr(layer, "output_shape", None)

  if output_shape is None and hasattr(layer, "output"):

    output_shape = getattr(layer.output, "shape", "Desconocido")

  print(f"Capa {i}: {layer.name} - {layer.__class__.__name__} - salida: {output_shape}")
```

```
puntos = [[0.1, 0.2], [0.5, 0.6], [1.0, 0.8]]

predicciones = modelo.predict(puntos)
```

Si la salida es un valor entre 0 y 1, probablemente sea una clasificación binaria y puedas usar:

```
clases = (predicciones > 0.5).astype(int)
```

MODELO

Layer (type)	Output Shape	Param #
dense_100 (Dense)	(None, 23)	69
dense_101 (Dense)	(None, 23)	552
dense_102 (Dense)	(None, 21)	504
dense_103 (Dense)	(None, 19)	418
dense_104 (Dense)	(None, 12)	240
dense_105 (Dense)	(None, 6)	78
dense_106 (Dense)	(None, 6)	42
dense_107 (Dense)	(None, 5)	35
dense_108 (Dense)	(None, 1)	6

Detalles importantes

- **Entrada esperada:** Debería ser un vector de 2 valores, por ejemplo [x1, x2]. Esto se deduce porque la primera capa (dense_100) tiene 69 parámetros, y con 2 entradas + 1 sesgo × 23 neuronas = 69.

  (2+1) x 23=69

- **Salida:** Un solo valor por muestra → el modelo predice una probabilidad (de clase 1), y luego se puede usar un umbral de 0.5 para clasificar como 0 o 1.

- **Código equivalente del modelo**
  Este es un código Python **reconstruido** que representa muy probablemente tu modelo original:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense


model = Sequential([
   Dense(23, input_shape=(2,), activation='relu'),  # dense_100
   Dense(23, activation='relu'),              # dense_101
   Dense(21, activation='relu'),              # dense_102
   Dense(19, activation='relu'),              # dense_103
   Dense(12, activation='relu'),              # dense_104
   Dense(6, activation='relu'),              # dense_105
   Dense(6, activation='relu'),              # dense_106
   Dense(5, activation='relu'),              # dense_107
   Dense(1, activation='sigmoid')              # dense_108
])
```

¿Cómo usarlo para predicción?

```
import numpy as np
def predict_batch(model, x1_list, x2_list):
    data = np.column_stack((x1_list, x2_list))
    predictions = model.predict(data)
    return (predictions > 0.5).astype(int).flatten().tolist()
```

Usa predict_point

```
def predict_point(model, x1, x2):
data = np.array([x1, x2])
pred = model.predict(data)[0][0]
return int(pred > 0.5)
```