

Caso Práctico Individual

TEORÍA:

MONGODB

La principal diferencia entre las bases de datos relacionales y las no relacionales es cómo almacenan la información. Las bases de datos relacionales almacenan los datos en tablas y se basan en SQL, mientras que las bases de datos no relacionales almacenan los datos en documentos y por tanto tienden a ser más flexibles

Nosotros vamos a estar utilizando Mongo Db, una base de datos no relacional.

Nos guarda estructuras de datos en un formato similar al JSON, permite almacenamiento de muchos tipos de archivos incluyendo imágenes.

Instalación: <https://youtu.be/2cWZ0lFbJoY>

Al iniciar se nos va a asignar un puerto por defecto 27017, al crear la base de datos podemos ver como se nos han creado 3 colecciones grandes por defecto:

- admin: Para almacenar usuarios a los cuales les permitimos el acceso, este acceso puede controlarse por medio de una contraseña
- config: Metemos todas aquellas cosas que gestionan la configuración de la base
- local: Siempre podemos crear una local nueva, aquí recogeremos toda nuestra información.

Desde MongoDB compass podemos ver y trabajar con todo de manera muy visual

The screenshot shows the MongoDB Compass interface. At the top, there's a dark header bar with the title 'MongoDB Compass - localhost:27017'. Below it, a navigation bar with 'Connect', 'Edit', 'View', and 'Help' buttons. The main area has tabs for 'My Queries', 'Databases', and 'Performance', with 'Databases' currently selected. A 'Create database' button and a 'Refresh' button are visible. On the left, a sidebar shows a tree view of databases: 'localhost:27017' (selected), 'My Queries', 'Databases' (selected), and a '+' button. Under 'Databases', there are three entries: 'admin', 'config', and 'local'. Each entry provides storage statistics: 'Storage size:' (20.48 kB for admin, 24.58 kB for config, 77.82 kB for local), 'Collections:' (1 for each), and 'Indexes:' (1 for admin, 2 for config, 3 for local). A 'View' button with a dropdown menu is located at the top right of the main content area.

Con este sistema de gestión yo puedo ver y modificar los datos en tiempo real.

Vamos a ver algunos comandos básicos, vamos a jugar desde la terminal como estamos en windows, tenemos que asegurarnos de previamente haber añadido mongo al path (dentro de nuestras variables de entorno).

Vamos a crear una base de datos dentro de la carpeta que queramos:

```
>> mongod --dbpath "direccion_de_carpeta" -port "n_de_puerto"  
(Si no le asignamos un puerto, mongo nos asigna uno automáticamente)
```

Nos hemos conectado a una nueva base de datos, podemos comprobarlo viendo la carpeta que hemos asignado

Ahora abrimos un nuevo terminal y comprobamos. Iniciamos el shell de mongo desde el directorio en el que estemos trabajando:

```
C:\Users\eherr\OneDrive\Escritorio>mongosh
```

Trabajamos en local

```
test> use local  
switched to db local
```

Para ir familiarizándonos vamos a probar introduciendo algunos datos dentro del local.

Esto lo conseguimos mediante db.collection.**insert("Lo que estemos insertando en formato clave": "valor")**

```
local> db.alumnos.insert({"nombre": "Alberto", "edad": 30})  
DeprecationWarning: Collection.insert() is deprecated. Use insertOne, insertMany, or bulkWrite.  
{  
  acknowledged: true,  
  insertedIds: { '0': ObjectId("641c8513f9df73c7ed31b458") }  
}  
local> db.alumnos.find()  
[  
  {  
    _id: ObjectId("641c8513f9df73c7ed31b458"),  
    nombre: 'Alberto',  
    edad: 30  
}
```

Con db.collection.**find()** nos muestran todos los archivos, añadimos alguna colección mas y con el comando show collections podemos comprobarlas.

```
local> db.asignaturas.insert({ "asignatura": "IA", "temas": ["Tema1", "Tema2"] })
{
  acknowledged: true,
  insertedIds: { '0': ObjectId("641c86b7f9df73c7ed31b45a") }
}
local> show collections
alumnos
asignaturas
```

al comando `.find()` le podemos añadir filtros para hacer una búsqueda más precisa

- `$gt`: mayor que
- `$gte`: mayor o igual que
- `$lt`: menor que
- `$lte`: menor o igual que

```
local> db.alumnos.find({ "edad": {$gte: 30} })
[
  {
    _id: ObjectId("641c8513f9df73c7ed31b458"),
    nombre: 'Alberto',
    edad: 30
  }
]
```

NOTA: Para mas operadores lógicos consultar

<https://www.mongodb.com/docs/manual/reference/operator/query-logical/>

Con el comando `db.collection.drop()` eliminamos la colección que le indiquemos y con `db.collection.remove("dato que queremos eliminar")` eliminamos los datos que le pasemos

Desde la pagina oficial podemos seguir aprendiendo de este lenguaje de consulta, cada base de datos dispone del suyo.

Vamos a interactuar de una tercera forma, vamos a acceder desde python gracias a la librería `pymongo` que nos sirve de conexión entre nuestro script de python y la base de datos.

Este es un ejemplo:

 [pymongo_example.py](#)

1 kB

Si ejecutamos este archivo, podemos comprobar desde el compass como se ha creado.

The screenshot shows the MongoDB Compass interface. The top bar indicates the connection is to 'localhost:27017' and the database is 'local'. The left sidebar shows databases like 'admin', 'config', and 'local', with 'alumnos' selected. The main area is titled 'local.alumnos' and contains tabs for 'Documents', 'Aggregations', 'Schema', 'Explain Plan', 'Indexes', and 'Validation'. Under the 'Documents' tab, there is a search bar and a filter dropdown. Two documents are listed:

```
_id: ObjectId('641c8513f0df73c7ed31b458')
nombre: "Alberto"
edad: 30
```

```
_id: ObjectId('641c8623f0df73c7ed31b459')
nombre: "Fran"
edad: 25
```

PYTORCH

Aquí un breve repaso teórico

Módulo_4_Deep_Learning_Pytorch.... 995 kB

Aquí unos scripts de ejemplo

código_modulo_4.zip 8 kB

Un breve resumen de como creamos una red neuronal:

1. Preparamos los datos:

1. Leemos csv
2. Limpiamos los datos (asegurando que son buenos)
3. Preparamos el formato, transformando nuestro dataset en valores numéricos (Label encoder)
4. Separamos nuestra variable dependiente de las independientes
5. Separamos en train y test (80 - 20)

2. Definimos Modelo:

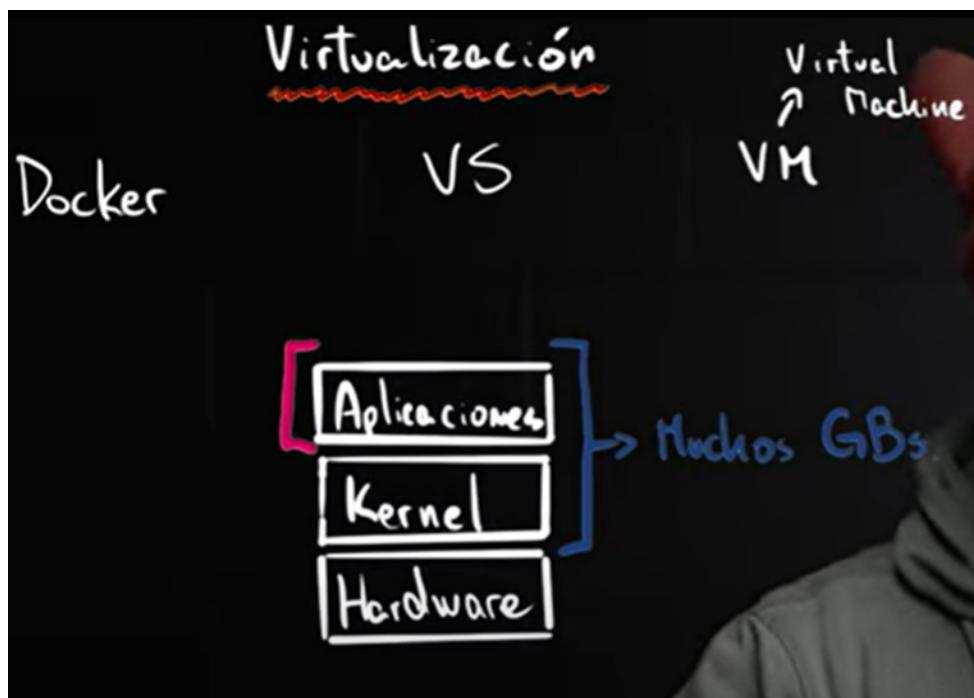
1. Capa i: Linear(n_entradas, n_salidas)
2. Inicialización de los pesos:

1. Kaiming para RELU
 2. Xavier para sigmoide
 3. F. de activación
 4. NOTA: forward nos une entre sí las capas
3. Entrenamos el modelo (con los datos train)
 1. Definimos optimización:
 1. F. de perdida.
 2. Optimizador: Generalmente gradiente desc. SGD(model.parameters(), lr, momentum).
 1. model.parameters(): parámetros necesarios para el entrenamiento
 2. lr: Tasa de aprendizaje. `lr` alto puede hacer que el modelo converja más rápidamente, pero puede dar lugar a que el modelo no converja o a que tenga una solución subóptima. Por otro lado, un valor de `lr` bajo puede hacer que el modelo converja más lentamente, pero puede permitir que el modelo alcance una solución mejor y más estable.
 3. momentum ayuda al algoritmo de optimización a superar los mínimos locales y a acelerar la convergencia a la solución óptima. Un valor de momentum alto puede hacer que el modelo converja más rápidamente, pero también puede hacer que se produzcan actualizaciones de peso inestables y oscilantes. Un valor de momentum bajo puede hacer que el modelo converja más lentamente, pero puede permitir que el modelo alcance una solución mejor y más estable.
 2. Iniciamos un epochs. Veces que pasamos nuestro conjunto de datos por la red. El número de épocas que se ejecutan durante el entrenamiento depende del tamaño y complejidad del conjunto de datos de entrenamiento, así como de la complejidad del modelo. En general, se debe elegir un número suficiente de épocas para permitir que el modelo converja a una solución óptima, pero no demasiadas épocas para evitar el sobreajuste (overfitting) del modelo a los datos de entrenamiento.

1. Separamos en batches (lotes). Usar batches en el entrenamiento del modelo puede tener varias ventajas, como reducir el uso de memoria y acelerar el proceso de entrenamiento, ya que se pueden procesar varias muestras de entrenamiento en paralelo. Además, puede ayudar a mejorar la generalización del modelo y reducir el sobreajuste a los datos de entrenamiento, ya que el modelo actualiza sus pesos en función de los errores de predicción en múltiples muestras en lugar de una sola muestra.
 1. Limpiamos los gradientes del anterior
 2. Se realiza una propagación hacia adelante a través de la red neuronal para generar una predicción, calcula la pérdida (error) de la predicción en comparación con la etiqueta verdadera.
 3. Se realiza una propagación hacia atrás para actualizar los pesos del modelo en función de la pérdida calculada.
 1. Actualizamos los pesos del modelo
4. Evaluamos el modelo (con los datos test)
 1. Sacamos los valores que describan a nuestro problema una mejor optimalidad (MSE, accuracy...)
5. Sacamos predicciones.
 1. Si nuestro modelo es óptimo podemos empezar a introducir nuevos valores

DOCKER

La idea de los contenedores es encapsular una aplicación, docker automatiza el despliegue de las mismas. Esto facilita el control de versiones, dependencias y con el uso de las imágenes permite la automatización del mismo.



A diferencia de una maquina virtual, docker utiliza el kernel (lo que comunica el hardware con las aplicaciones) del sistema operativo donde se está ejecutando por lo que ahorra mucho espacio.

NOTA:

Una virtualización puede ser:

- Paravirtualización
- Virtualización parcial
- Virtualización completa

Docker es muy superior!!

Conceptos básicos:

- Imagen: Representación estática de una aplicación, su configuración y dependencias. <https://hub.docker.com/> -> Pag de referencia
- Contenedor: Instancia de una imagen, pesan muy poco en comparación de las máquinas virtuales
- Registros: Repositorio donde se almacenan las imágenes creadas
- Volúmenes: Permiten persistir datos en los contenedores
- Network: Recurso que conecta los contenedores (incluso si están en diferentes host)

Docker Engine está compuesto por:

1. Docker daemon: es el proceso central de Docker Engine que gestiona los contenedores, imágenes, redes y volúmenes. También se encarga de la comunicación con el cliente Docker a través de una API.

2. Docker CLI: es la interfaz de línea de comandos que se utiliza para interactuar con Docker daemon. Se utiliza para ejecutar comandos como crear y gestionar contenedores, imágenes, networks y volúmenes.
3. API de Docker: es una interfaz de programación de aplicaciones que permite a los desarrolladores interactuar con Docker Engine y crear aplicaciones que utilicen los recursos de Docker.

Vamos a ir probando algunos comandos desde la terminal, para no tener errores nos aseguramos de tener docker desktop corriendo (la aplicación abierta). Además tenemos que tener docker en las variables de entorno de windows.

Estructura de los comandos:

```
docker <command> [-option] <object>
```

Para crear una imagen usamos

```
>> docker pull mongo
```

En este caso descargo una imagen mongo, al no haberle indicado versión tomará la última (latest).

Mongo es una de las imágenes más fáciles de iniciar puesto que no requiere tantas variables de configuración

Si queremos otra versión se lo indicamos (Ej: >> docker pull node:18)

Podemos comprobar que se ha descargado mediante el comando

```
>> docker images
```

C:\Users\eherr>docker images				
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
mongo	latest	9a5e0d0cf6de	10 days ago	646MB
postgres	latest	6e5b7c4abf29	4 months ago	379MB

NOMBRE VERSIÓN IDENTIFICADOR ÚNICO FECHA DE CREACIÓN TAMAÑO

Cada imagen tendrá su propio id

Para eliminar alguna imagen utilizaremos `rm`

Ej:

```
>> docker image rm node:18
```

Vamos a crear nuestro primer contenedor, esto lo haremos con `create`, esto nos devolverá el ID del contenedor que acabamos de crear, el cual necesitaremos para ejecutarlo mediante `start`

Por otro lado `stop` lo detiene.

Mediante docker ps vemos los contenedores que están corriendo y añadiendo - letra podemos:

- -a muestra todos, no solo los que están corriendo
- -l muestra el último contenedor creado
- -n [N] muestra los últimos N contenedores
- -q solo muestra los ids
- -f "key=value" muestra los contenedores filtrados por clave valor (id, name, status...)

```
C:\Users\eherr>docker create mongo
fbe2fd4a238f25266864c6a16792c8e5b091b79abe1a3d7e9b4ca4403f22b995

C:\Users\eherr>docker start fbe2fd4a238f25266864c6a16792c8e5b091b79abe1a3d7e9b4ca4403f22b995
fbe2fd4a238f25266864c6a16792c8e5b091b79abe1a3d7e9b4ca4403f22b995

C:\Users\eherr>docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
fbe2fd4a238f mongo "docker-entrypoint.s..." 2 minutes ago Up 21 seconds 27017/tcp vigilant_moore
```

Observamos como el Id que se nos muestra es una abreviatura del primero, con el cual también podemos hacer referencia a nuestro contenedor. También podemos referenciarlo mediante el nombre que fue asignado de manera aleatoria por docker, el cual podremos cambiar a nuestro gusto (container rename).

El puerto 27017 es el utilizado por mongo por defecto, aunque si intentamos acceder notaremos que no podemos, de esta manera introducimos el concepto de port mapping.

Probemos a crear un contenedor donde mapearemos los puertos:

```
C:\Users\eherr>docker create -p27017:27017 --name Contenedor_de_prueba mongo
6c5e34c8c95bdb57a46ce576eeb71c914c82d467c22c40c3d44ffa41544bb01
C:\Users\eherr>docker start Contenedor_de_prueba
Contenedor_de_prueba

C:\Users\eherr>docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
6c5e34c8c95b mongo "docker-entrypoint.s..." 30 seconds ago Up 6 seconds 0.0.0.0:27017->27017/tcp Contenedor_de_prueba
fbe2fd4a238f mongo "docker-entrypoint.s..." 28 minutes ago Up 26 minutes 27017/tcp vigilant_moore
```

El primer puerto que le indicamos es el puerto del host y el segundo es el puerto del contenedor que vamos a crear y queremos mapear, es decir, cuando el contenedor Docker esté en ejecución, cualquier aplicación que quiera acceder al servicio de Mongo que se ejecuta dentro del contenedor, deberá conectarse al primer puerto indicado (del host) en lugar de al segundo puerto (del contenedor).
En caso de no especificar puertos, docker lo hará por nosotros

Con el comando de docker logs nombre_contenedor podemos ver si se está ejecutando de manera correcta y para hacer una escucha continua añadimos --follow. (control + c para deterner)

>> docker logs --follow nombre_contenedor

Para simplificar todo el proceso de creación de una imagen (descargarla en caso de no tenerla), creación un contenedor e iniciar el mismo, disponemos de `run`

Ej:

```
>> docker run -name Contenedor_de_prueba -p 27017:27017 -d mongo
```

-d es para no realizar un follow

Cuando estemos creando nuestro container en mongo vamos a necesitar ciertas variables de entorno las cuales en mongo son

`MONGO_INITDB_ROOT_USERNAME=` y `MONGO_INITDB_ROOT_PASSWORD=`
Viene siendo un usuario y contraseña

Ej:

```
>> docker create -p27017:27017 --name Contenedor_de_prueba -e  
MONGO_INITDB_ROOT_USERNAME=Enrique -e  
MONGO_INITDB_ROOT_PASSWORD=mi_contraseña
```

Existen frameworks y librerías que permiten enlazar nuestros programas en python.
Nosotros utilizaremos pymongo para interactuar con nuestro contenedor.
Ahora este enlace lo hacemos mediante un archivo llamado necesariamente `dockefile`, donde vamos a escribir las instrucciones de nuestro contenedor para poder crearse.

```
FROM node:18

RUN mkdir -p /home/app

COPY . /home/app

EXPOSE 3000

CMD ["node", "/home/app/index.js"]
```

Este es un ejemplo típico de dockerfile

- FROM: Indicamos la imagen y su versión

- RUN mkdir -p: Indicamos la ruta donde vamos a meter el código fuente de nuestra aplicación
- COPY: nos permite acceder a nuestros archivos anfitrío y copiarlos en la ruta de destino que le hemos pasado
- EXPOSE: Exponemos un puerto en el que se levanta la aplicación, para que el resto de contenedores se conecten a través de este puerto
- CMD: Indicamos el comando que se tiene que ejecutar para que la aplicación corra, además de los argumentos

Aún no podemos crear el contenedor porque no disponemos de una red, necesitamos una red donde todos los contenedores que formen parte de esa red podrán conectarse entre ellos.

Antes de haber creado ninguna podemos ver que ya docker nos proporciona las siguientes edes

```
C:\Users\eherr>docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
2da349335fa3   bridge    bridge      local
80612b03581f   host      host       local
612aeea49e35   none      null       local
```

Vamos a crear una red de manera muy similar a las imágenes (para trabajar con ellas es similar)

```
C:\Users\eherr>docker network create mired
269c928197843438bfc64b8a902b6001e1f8190ee79aa602a10024fe309b7f0c

C:\Users\eherr>docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
2da349335fa3   bridge    bridge      local
80612b03581f   host      host       local
269c92819784   mired     bridge      local
612aeea49e35   none      null       local
```

Conociendo esto podemos añadir un contenedor a la red, incluso desde la propia creación del contenedor simplemente añadiendo --network "nombre_de_la_red"

Ahora bien, gracias a la herramienta docker-compose podemos automatizar aún más el despliegue, desde nuestro editor de texto crearemos un archivo docker-compose.yml (.yaml es una extensión propia de las configuraciones) Esta herramienta es para la orquestación de contenedores docker, permite configurar diferentes servicios, esto permite tener muchos entornos en un solo host con diferentes variables de entorno.

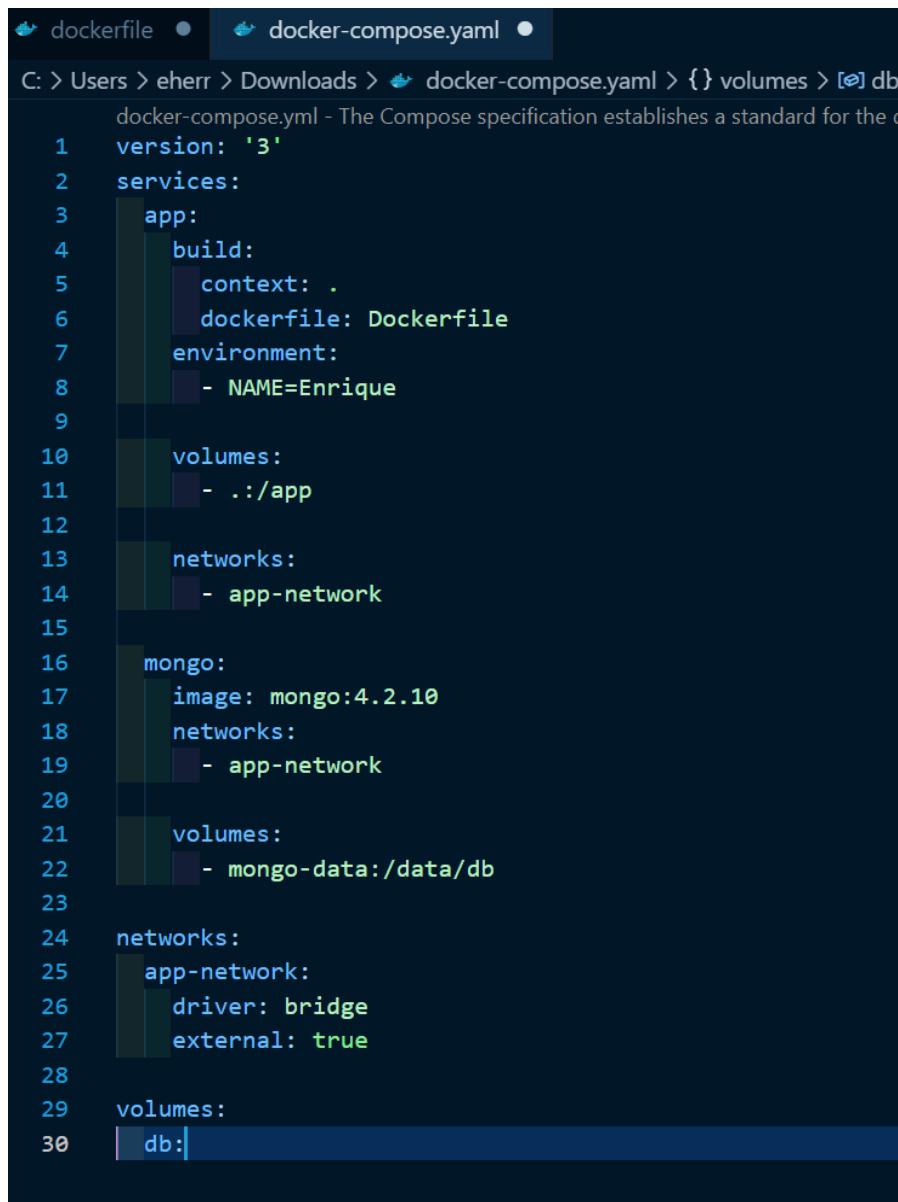
Comandos básicos:

- Build: construye las imágenes de los servicios
- Up: crea los contenedores de los servicios, redes y volúmenes
- Start: Inicia un contenedor existente
- Stop: Para un contenedor
- Down Para y borra los contenedores, redes y volúmenes

Un archivo de configuración está dividido en varios bloques con diferentes propiedades

- Servicios:
 - Build: Especificamos el contexto y el dockerfile para crear la imagen.
 - Image: Nombre de la imagen
 - Volumes: Definición de los volúmenes del contenedor
 - Environment: Definición de variables de entorno
 - Ports: Mapeado de puertos
 - Networks: lista de redes conectadas
- Redes: Definimos las redes personalizadas
 - Driver: Bridge(por defecto), host, overlay...
 - External (Booleano): Utiliza una red ya existente
- Volúmenes: Define los volúmenes con nombre
 - Name: Volume
 - External (Booleano): Si el volume es para todos los contenedores del host o solo para los servicios del docker-compose

Ejemplo:



The screenshot shows a terminal window with two tabs: 'dockerfile' and 'docker-compose.yaml'. The current tab is 'docker-compose.yaml'. The file content is as follows:

```
C: > Users > eherr > Downloads > docker-compose.yaml > {} volumes > db
  docker-compose.yml - The Compose specification establishes a standard for the d
1  version: '3'
2  services:
3    app:
4      build:
5        context: .
6        dockerfile: Dockerfile
7        environment:
8          - NAME=Enrique
9
10   volumes:
11     - ./app
12
13   networks:
14     - app-network
15
16   mongo:
17     image: mongo:4.2.10
18     networks:
19       - app-network
20
21   volumes:
22     - mongo-data:/data/db
23
24   networks:
25     app-network:
26       driver: bridge
27       external: true
28
29   volumes:
30     db:
```

Ahora ejecutando un par de comandos desde la terminal:

1 Construimos la aplicación:

>> docker-compose build

2 Levantamos la aplicación:

>> docker-compose up

PRÁCTICA:

ENUNCIADO:

1 – Clonar repositorio del proyecto (este contiene el código de entrenamiento y predicción así como el modelo previamente entrenado):

2 – Crear una nueva rama en el proyecto con el siguiente formato (tu nombre de usuario en la plataforma): NombreApellido1Apellido2

3 – Desarrollar el código Python para entrenamiento de una red neuronal con el dataset proporcionado ([6_training.py](#)).

4 – Entrenamiento, evaluación de resultados y creación de código para realizar predicciones con el modelo entrenado ([7_predict.py](#)). ***Nota: si el entrenamiento lleva mucho tiempo o no se tienen recursos suficientes ya se proporciona el modelo entrenado para su uso.

5 – Almacenamiento de las predicciones en una base de datos Mongo (empleando pymongo en Python).

6 – Encapsulamiento del proyecto en un contenedor Docker que tomará unos datos de entrada de una carpeta y realizará las tareas del script para predicciones.

7 – Añadir nuevos ficheros y cambios (scripts modificados, dockerfile...) al repositorio local (commit) y subir la nueva rama creada al repositorio en Github (push).

8 – Realizar una pequeña memoria (no más de una página) con los pasos realizados y subirla a la tarea del módulo habilitada para tal efecto.

RESOLUCIÓN:

Tenemos un Dataset que contiene 25.000 imágenes de perros y gatos con sus respectivas etiquetas por lo que estamos hablando de un aprendizaje supervisado. Haremos una red convolucional de clasificación.

Este data set nos servirá para entrenar un modelo con el que podremos predecir de cualquier foto si es un perro o un gato.

Los scripts proporcionados van a ser muy parecidos a los adjuntos en los ejemplos de pytorch.

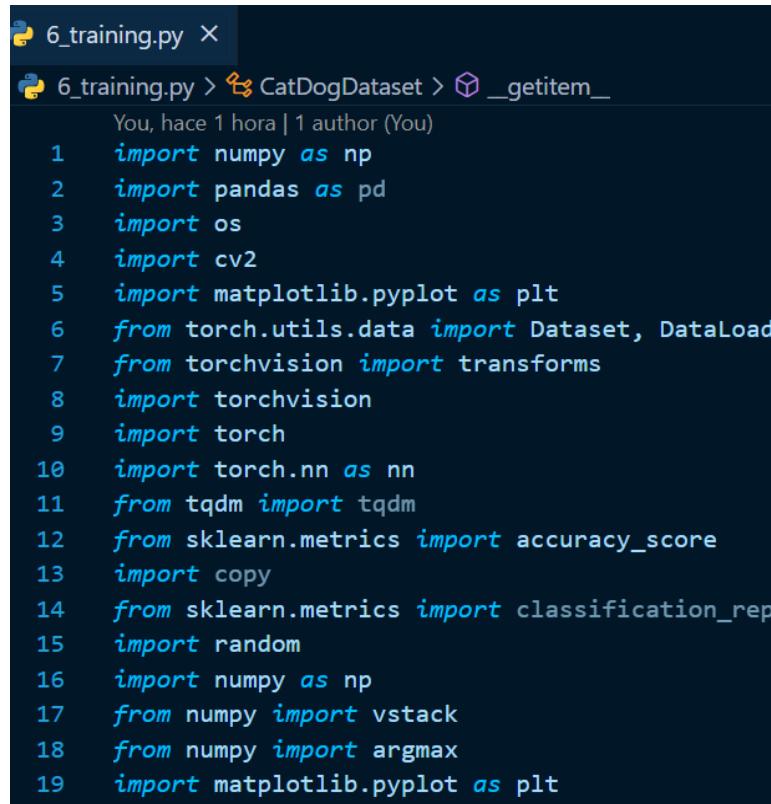
Disponemos de dos Scripts:

- 6_training.py : Aquí entrenaremos nuestro modelo a partir de nuestro dataset y lo guardaremos en formato .pth

- [7_predict.py](#) : Cargaremos el modelo ya entrenado y podremos predecir imágenes de perros y gatos

6_training.py

Empezamos importando las librerías que vamos a utilizar:



```
You, hace 1 hora | 1 author (You)
1 import numpy as np
2 import pandas as pd
3 import os
4 import cv2
5 import matplotlib.pyplot as plt
6 from torch.utils.data import Dataset, DataLoader
7 from torchvision import transforms
8 import torchvision
9 import torch
10 import torch.nn as nn
11 from tqdm import tqdm
12 from sklearn.metrics import accuracy_score
13 import copy
14 from sklearn.metrics import classification_report
15 import random
16 import numpy as np
17 from numpy import vstack
18 from numpy import argmax
19 import matplotlib.pyplot as plt
```

- PASO 1:

Vamos a crear una clase CatDogDataset que hereda de la clase Dataset de Pytorch, la utilizaremos para cargar imágenes de perros y gatos con sus etiquetas correspondientes.

```

class CatDogDataset(Dataset):
    def __init__(self, train_dir, img_list, transform = None):
        self.train_dir = train_dir
        self.transform = transform
        self.images = img_list

    def __len__(self):
        return len(self.images)

    def __getitem__(self, index):
        image_path = os.path.join(self.train_dir, self.images[index])
        label = self.images[index].split(".")[0]
        label = 0 if label == 'cat' else 1
        img = cv2.imread(image_path)
        if self.transform:
            img = self.transform(img)
        return img, torch.tensor(label)

```

En el constructor definimos la ruta al directorio de las imágenes, la lista de nombres y la transformación(opcional) que se le aplicará a las imágenes.

El método len nos devuelve el numero total de imágenes y el método getitem nos devuelve la imagen (con su etiqueta) que pidamos, le tenemos que indicar el indice.

- PASO 2:

El objetivo de este paso es crear un conjunto de datos de entrenamiento y validación (train y test).

```

41
42     ## Step 2: Create training dataset
43     data_transform = transforms.Compose([
44         transforms.ToTensor(),
45         transforms.Resize((256, 256)),
46         transforms.ColorJitter(),
47         transforms.RandomCrop(224),
48         transforms.RandomHorizontalFlip()
49     ])
50
51     # Create dataloaders
52     train_dir = 'dogs-vs-cats/train'
53     images_paths = os.listdir(train_dir)
54     random.shuffle(images_paths)
55     train_img_list = images_paths[0:2000]
56     val_img_list = images_paths[2000:25000]
57     train_img_list = images_paths[0:2000]
58     val_img_list = images_paths[2000:2500]
59     train_dataset = CatDogDataset(train_dir, train_img_list, transform = data_transform)
60     val_dataset = CatDogDataset(train_dir, val_img_list, transform = data_transform)
61     train_dataloader = DataLoader(train_dataset, batch_size = 64, shuffle=True)
62     val_dataloader = DataLoader(val_dataset, batch_size = 64, shuffle=False)
63
64     # Visualize images in the dataset
65     samples, labels = iter(train_dataloader).next()
66     plt.figure(figsize=(16,32))
67     grid_imgs = torchvision.utils.make_grid(samples[:32])
68     np_grid_imgs = grid_imgs.numpy()
69     # in tensor, image is (batch, width, height), so you have to transpose it to (width, height, batch) in numpy to show it.
70     plt.imshow(cv2.cvtColor(np.transpose(np_grid_imgs, (1,2,0)), cv2.COLOR_BGR2RGB))
71     plt.show()

```

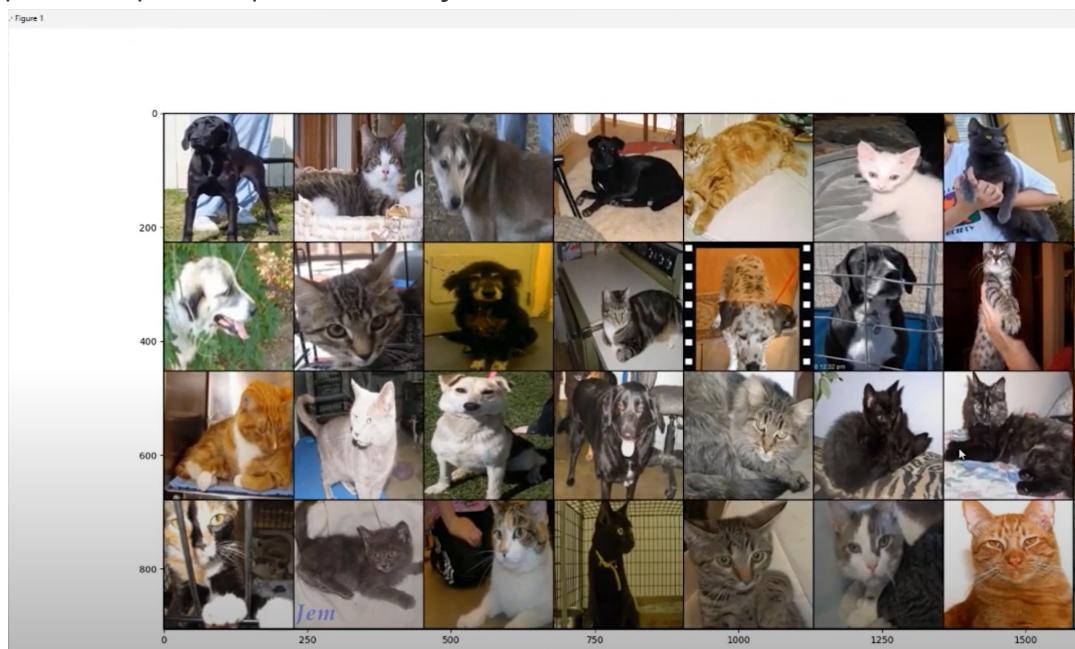
Definimos la transformación que se aplicará a cada imagen para mejorar el rendimiento del modelo, este debe recibir todas las imágenes con el mismo tamaño (nxn, es decir, mismo ancho que largo). Haremos además recortes aleatorios para

que haya modificaciones en las imágenes, de esta manera aumentaremos la variedad.

Crearemos dos conjuntos de datos utilizando la clase que definimos anteriormente

Por cada uno de estos conjuntos crearemos un dataloader (train_dataloader y val_dataloader), esto lo utilizaremos para poder cargar los datos en lotes durante el entrenamiento y la validación. El tamaño del lote lo establecemos en 64 y los únicamente mezclaremos los datos de entrenamiento aleatoriamente.

La última parte de este código nos visualizará algunas imágenes del entrenamiento para comprobar que estamos ejecutando correctamente.



De este set aleatorio de nuestro dataloader podemos comprobar que efectivamente se ha realizado el resize

- PASO 3:

Definimos la clase scratch_nn utilizando la API de pytorch.

```

## Step 3: Define Deep Learning model
...

class scratch_nn(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=100, kernel_size=5, stride=1, padding=0)
        self.conv2 = nn.Conv2d(100, 200, 3, stride=1, padding=0)
        self.conv3 = nn.Conv2d(200, 400, 3, stride=1, padding=0)
        self.mpool = nn.MaxPool2d(kernel_size=3)
        self.relu = nn.ReLU()
        self.linear1 = nn.Linear(19600, 1024)
        self.linear2 = nn.Linear(1024, 512)
        self.linear3 = nn.Linear(512, 2)
        self.classifier = nn.Softmax(dim=1)

    def forward(self,x):
        x = self.mpool( self.relu(self.conv1(x)) )
        x = self.mpool( self.relu(self.conv2(x)) )
        x = self.mpool( self.relu(self.conv3(x)) )
        x = torch.flatten(x, start_dim=1)
        x = self.linear1(x)
        x = self.linear2(x)
        x = self.linear3(x)
        x = self.classifier(x)
        return x

```

Utilizaremos convoluciones 2D y capas lineales para procesar imágenes de entrada y producir una salida de clasificación de dos clases.

La salida de las tres primeras capas (Convoluciones 2D), se ve afectada por un maxPool (kernel = 3). La activación es RELU.

Las siguientes tres capas son lineales, teniendo la última dos salidas (lógico pues estamos haciendo clasificación binaria), a estas últimas le aplicaremos una activación tipo SoftMax.

Encontramos también en nuestra clase el método forward, que define la secuencia de operaciones para procesar la entrada x a través de la red.

- PASO 4:

Definiremos dos funciones:

- train_step: La utilizaremos para entrenar el modelo.

```

## Step 4: Define train_step and predict functions
def train_step(train_loader, model, optimizer, criterion, device):
    # define the optimization
    avg_loss = []
    predictions, actuals = list(), list()
    # enumerate mini batches
    for i, (inputs, targets) in enumerate(train_loader):
        inputs, targets = inputs.to(device), targets.to(device)
        # clear the gradients
        optimizer.zero_grad()
        # compute the model output
        yhat = model(inputs)
        # calculate loss
        loss = criterion(yhat, targets)
        # credit assignment
        loss.backward()
        # update model weights
        optimizer.step()
        avg_loss.append(loss.item())
        # Get accuracy
        actual = targets.cpu().numpy()
        yhat = argmax(yhat.detach().cpu().numpy(), axis=1)
        # reshape for stacking
        actual = actual.reshape(len(actual), 1)
        yhat = yhat.reshape(len(yhat), 1)
        # store
        predictions.append(yhat)
        actuals.append(actual)

    predictions, actuals = vstack(predictions), vstack(actuals)
    # calculate accuracy
    acc = accuracy_score(actuals, predictions)
    return sum(avg_loss)/len(avg_loss), acc

```

Recibimos cuatro parámetros: train_loader que es el dataloader de entrenamiento, model que es la arquitectura de la red neuronal, optimizer que es el optimizador que se utiliza para ajustar los pesos de la red neuronal y criterion que es la función de pérdida.

En primer lugar, se recorren los lotes del dataloader y se envían los datos y las etiquetas al dispositivo (CPU o GPU). Después, se establecen los gradientes a cero, se calcula la salida de la red neuronal, se calcula la función de pérdida, se propagan los gradientes hacia atrás y se actualizan los pesos del modelo. Finalmente, se calcula la precisión del modelo en el conjunto de entrenamiento.

- evaluation_step: La utilizaremos para evaluar el modelo en el conjunto de validación.

```

# evaluate the model
def evaluation_step(val_loader, model, criterion, device):
    avg_loss = []
    predictions, actuals = list(), list()
    for i, (inputs, targets) in enumerate(val_loader):
        inputs, targets = inputs.to(device), targets.to(device)
        # evaluate the model on the test set
        yhat = model(inputs)
        loss = criterion(yhat, targets)
        avg_loss.append(loss.item())
        # retrieve numpy array
        yhat = yhat.detach().cpu().numpy()
        actual = targets.cpu().numpy()
        # convert to class labels
        yhat = argmax(yhat, axis=1)
        # reshape for stacking
        actual = actual.reshape(len(actual), 1)
        yhat = yhat.reshape(len(yhat), 1)
        # store
        predictions.append(yhat)
        actuals.append(actual)

    predictions, actuals = vstack(predictions), vstack(actuals)
    # calculate accuracy
    acc = accuracy_score(actuals, predictions)
    return sum(avg_loss)/len(avg_loss), acc

```

Recibimos cuatro parámetros: val_loader (dataloader de validación), model (arquitectura de la red neuronal), criterion (función de pérdida) y device (dispositivo en el que se realizará la evaluación).

En primer lugar, se recorremos los lotes del dataloader y enviamos los datos y etiquetas al dispositivo. Después, calculamos la salida de la red neuronal y la función de pérdida. Finalmente, calculamos la precisión del modelo en el conjunto de validación.

- PASO 5:

La siguiente función es el bucle principal de entrenamiento de nuestro modelo de red neuronal.

```

## Step 5: Define main train function
def train(model, train_dataloader, val_dataloader, optimizer, criterion, device):
    num_epoch = 5#300
    train_loss_list, train_acc_list, val_loss_list, val_acc_list = [], [], [], []
    for epoch in range(1, num_epoch + 1):
        train_loss, train_acc = train_step(train_dataloader, model, optimizer, criterion, device)
        train_loss_list.append(train_loss)
        train_acc_list.append(train_acc*100)
        print(f"Train: Loss at epoch {epoch} is {train_loss} and accuracy is {train_acc}%")
        val_loss, val_acc = evaluation_step(val_dataloader, model, criterion, device)
        val_loss_list.append(val_loss)
        val_acc_list.append(val_acc*100)
        print(f"Validation: Loss at epoch {epoch} is {val_loss} and accuracy is {val_acc}%")
        torch.save(model.state_dict(), model_file_name)
    return model, train_loss_list, train_acc_list, val_loss_list, val_acc_list

```

Tomamos como entrada el modelo, los dataloaders de entrenamiento y validación, el optimizador, la función de pérdida y el dispositivo donde realizaremos el entrenamiento (CPU o GPU).

Comenzamos definiendo el número de épocas que se utilizarán para entrenar la red neuronal.

En cada una llamamos a la función train_step para realizar el entrenamiento. La pérdida y la precisión se almacena en las listas train_loss_list y train_acc_list.

En cada una de las épocas tambien evaluamos el modelo, printeando la perdida y precisión.

Acabamos guardando el modelo

- PASO 6:

Cargamos los datos y utilizamos las funciones definido previamente

```

# Step 6: Training parameters, model declaration and training/validation process
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = scratch_nn()
model = model.to(device)

lr = 0.001
weight_dec = 0.001
model_file_name = "dogs_cats_model.pth"
optimizer = torch.optim.Adam(model.parameters(), lr=lr, weight_decay=weight_dec)
criterion = nn.CrossEntropyLoss()
model, train_loss, train_acc, val_loss, val_acc = train(model, train_dataloader, val_dataloader, optimizer, criterion, device)

```

El modelo se entrena durante cinco épocas y los resultados de pérdida y precisión se almacenan en listas para su análisis y visualización en el paso 7.

NOTA:

Se utiliza el optimizador Adam para actualizar los parámetros del modelo durante el entrenamiento.

- PASO 7:

Ahora podemos visualizar los resultados del entrenamiento.

```

# Step 7: Show results
plt.figure(1)
plt.plot(train_loss)
plt.plot(val_loss)
plt.title("Loss in training and validation")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend(["Train", "Validation"])

plt.figure(2)
plt.plot(train_acc)
plt.plot(val_acc)
plt.title("Accuracy in training and validation")
plt.xlabel("Epochs")
plt.ylabel("Accuracy (%)")
plt.legend(["Train", "Validation"])

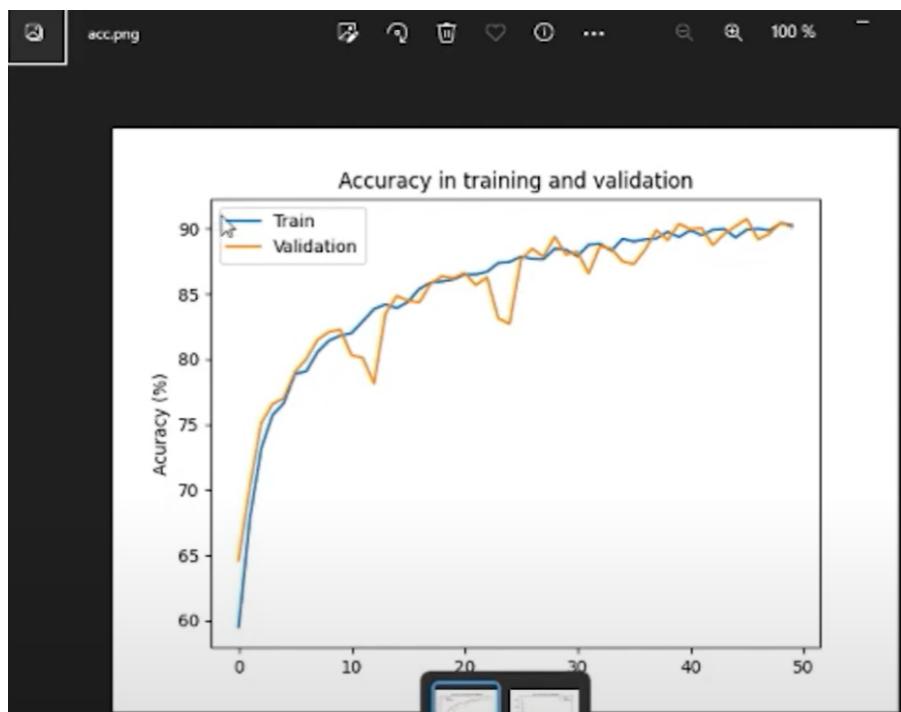
plt.show()

```

La primera gráfica muestra la pérdida en el conjunto de entrenamiento y validación en función del número de épocas.



La segunda gráfica muestra la precisión en el conjunto de entrenamiento y validación en función del número de épocas.



Podemos ver que la red neuronal comienza a aprender rápidamente y, después de alrededor de 5 épocas, se estabiliza. La precisión en el conjunto de validación alcanza aproximadamente el 90%.

7_predict.py

Este script carga un modelo previamente entrenado y lo utiliza para realizar predicciones de clasificación de imágenes de perros y gatos. La red neuronal utilizada es una red convolucional con tres capas de convolución y una capa de clasificación final. La red está definida en la clase scratch_nn. Se carga el modelo entrenado previamente y se define la transformación de preprocesamiento de imagen para escalar la imagen a 224x224 y convertirla en tensor.

```

# Definir modelo
You, hace 8 horas | 1 author (You)
class scratch_nn(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=100, kernel_size=5, stride=1, padding=0)
        self.conv2 = nn.Conv2d(100, 200, 3, stride=1, padding=0)
        self.conv3 = nn.Conv2d(200, 400, 3, stride=1, padding=0)
        self.mpool = nn.MaxPool2d(kernel_size=3)
        self.relu = nn.ReLU()
        self.linear1 = nn.Linear(19600,1024)
        self.linear2 = nn.Linear(1024,512)
        self.linear3 = nn.Linear(512,2)
        self.classifier = nn.Softmax(dim=1)

    def forward(self,x):
        x = self.mpool( self.relu(self.conv1(x)) )
        x = self.mpool( self.relu(self.conv2(x)) )
        x = self.mpool( self.relu(self.conv3(x)) )
        x = torch.flatten(x, start_dim=1)
        x = self.linear1(x)
        x = self.linear2(x)
        x = self.linear3(x)
        x = self.classifier(x)
        return x

# Cargar modelo entrenado
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = scratch_nn()      You, hace 8 horas • predict
model.load_state_dict(torch.load("dogs_cats_model.pth", map_location=device))

model.eval()
model = model.to(device)

# Definir procesados de la imagen
data_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Resize((224, 224)),
])

```

Para cada imagen en la carpeta predict_cat_dog, se lee la imagen, se preprocesa y se pasa por la red neuronal. El resultado es una distribución de probabilidad sobre las dos clases. La etiqueta con la mayor probabilidad se selecciona como la predicción y se almacena en la variable output.

```

# Realizar la predicción de todas las imágenes en la carpeta
labels = ["Cat", "Dog"]
for image_path in glob.glob("predict_cat_dog/*.jpg"):
    img_orig = cv2.imread(image_path)
    img = data_transform(img_orig).unsqueeze(0).to(device)
    outputs = model(img)
    outputs = outputs.detach().cpu().numpy()
    output = argmax(outputs, axis=1)[0]
    print("Predicted label: "+labels[output])
    cv2.imshow("Predicted label: "+labels[output], img_orig)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

```

Acabamos mostrando la imagen en una ventana con la etiqueta predicha y se almacena la predicción en una base de datos MongoDB con la ruta de la imagen y la etiqueta predicha.



Podemos observar en cada una de las 10 etiquetas que se ha cumplido, por lo que estamos prediciendo bien.

Damos por válido el modelo

DOCKERIZACIÓN

Empezamos creando en el directorio que estamos trabajando un archivo dockerfile, que ya sabemos por la teoría que solo se puede llamar de esta manera.

The screenshot shows a code editor interface with two panes. On the left, the file tree (Explorador) shows a folder named 'CASO_PRACTICO_INDIVIDUAL' containing several files: predict_cat_dog, Teoría_previa, .gitignore, 6_training.py, 7_predict.py, docker-compose.yaml, and req.txt. The file 'req.txt' is currently selected. On the right, the code editor pane displays the Dockerfile content:

```

FROM bitnami/pytorch:latest
ADD req.txt .
RUN pip install -r req.txt
WORKDIR /app
ADD 7_predict.py .
ADD dogs_cats_model.pth .
CMD ["python", "7_predict.py"]

```

Le vamos a añadir las secuencias que encontramos en la imagen

- FROM: Nombre y versión de la imagen que vamos a crear, en nuestro caso hemos buscado una que tuviera pytorch incluido
- ADD: Añadimos los requisitos, este es un archivo .txt en el que indicamos las librerías que debemos instalar

The screenshot shows a code editor pane displaying the 'req.txt' file content:

```

pymongo
pillow

```

NOTA:

pytorch no necesitamos instalarla puesto que viene con la imagen

Los puntos en las líneas de ADD indican el contexto, que es el directorio en el que nos encontramos

- RUN: Instalamos estas dos librerías mediante un pip install -r
- WORKDIR: Indicamos el directorio de trabajo
- ADD: Indicamos los archivos que queremos abrir
- ADD: El modelo también
- CMD: Indicamos al terminal que vamos a ejecutar en python el archivo [7_predict.py](#)

Vamos a construir nuestro contenedor mediante el comando `build`

```
(base) C:\Users\eherr\OneDrive\Escritorio\UAX\3er Curso\Inteligencia artificial\Caso_Practico_individual
> * Historial restaurado

Microsoft Windows [Versión 10.0.22621.1413]
(c) Microsoft Corporation. Todos los derechos reservados.

=> [internal] load .dockerignore                                         0.0s
=> => transferring context: 2B                                         0.0s
=> [internal] load metadata for docker.io/bitnami/pytorch:latest      2.0s
=> [auth] bitnami/pytorch:pull token for registry-1.docker.io          0.0s
=> [1/6] FROM docker.io/bitnami/pytorch:latest@sha256:5d66a2c048b4806de6979962ce98a5bec79cd875f9 0.0s
=> [internal] load build context                                       0.0s
=> => transferring context: 101B                                       0.0s
=> CACHED [2/6] ADD      req.txt .                                     0.0s
=> CACHED [3/6] RUN      pip install -r req.txt                      0.0s
=> CACHED [4/6] WORKDIR /app                                         0.0s
=> CACHED [5/6] ADD      7_predict.py .                                0.0s
=> CACHED [6/6] ADD      dogs_cats_model.pth .                         0.0s
=> exporting to image                                                 0.0s
=> => exporting layers                                              0.0s
=> => writing image sha256:327cb58cad2497fb8bc266c890e90afcfcae30001cb09892f0a90f16e90e380d 0.0s
```

Ahora vamos a crear nuestro docker-compose para enlazar nuestros resultado a una base de mongo.

```
You, hace 6 horas | 1 author (You)
version: "3"
You, hace 6 horas | 1 author (You)
services:
  You, hace 6 horas | 1 author (You)
    predict:
      You, hace 6 horas | 1 author (You)
        build:
          context: .
        volumes:
          - ./predict_cat_dog/:/predict_cat_dog
        networks:
          - predict-public

You, hace 6 horas | 1 author (You)
mongo:
  image: mongo:4.2.10
  container_name: mongo
  networks:
    - predict-public
  volumes:
    - db
  ports:
    - 27017:27017

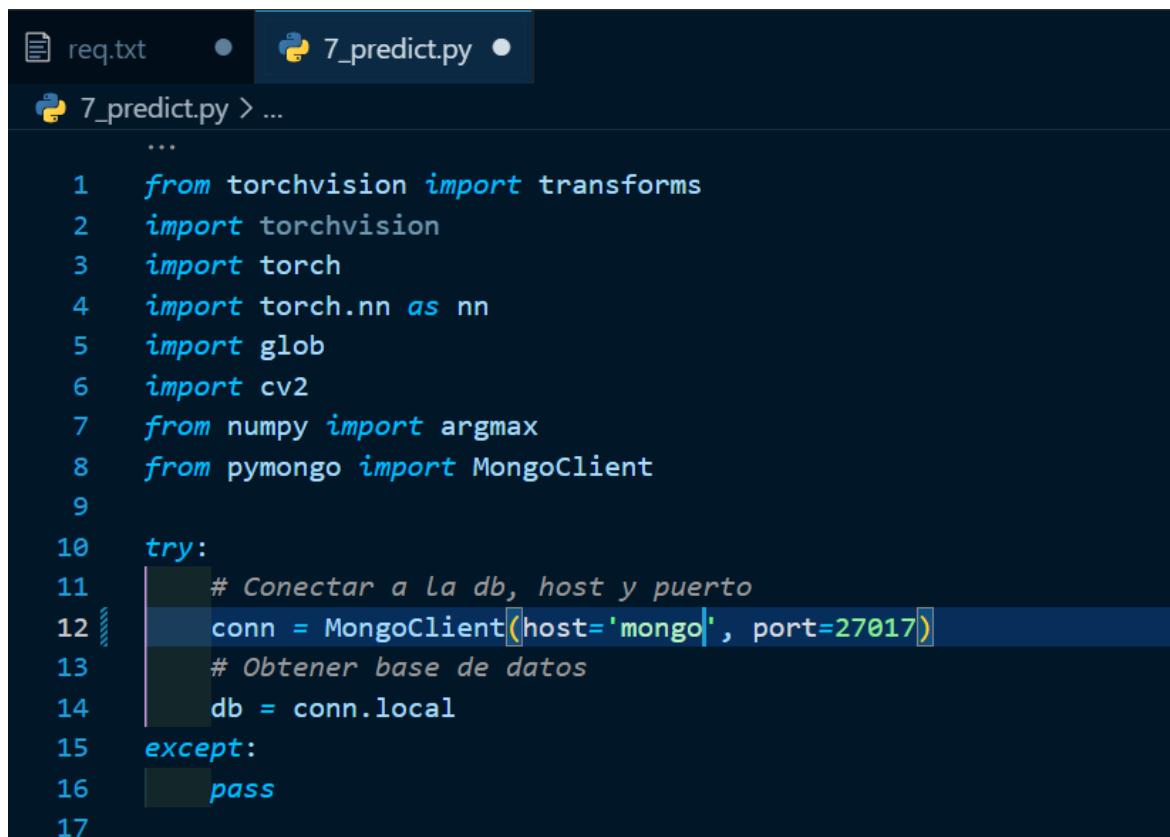
You, hace 6 horas | 1 author (You)
networks:
  You, hace 6 horas | 1 author (You)
    predict-public:
      driver: bridge

You, hace 6 horas | 1 author (You)
volumes:
  |   db:
```

1. Indicamos versión
 2. Indicamos los servicios, ya vimos antes que eran 3:
 1. build: le indicamos donde construimos

2. volumes: Le ponemos un nombre y le indicamos la carpeta donde guardamos, en nuestro caso será el mismo nombre
3. networks: Ponemos nombre de la red
3. Construimos la imagen de mongo
 1. Indicamos version
 2. Nombre
 3. Indicamos red (que va a ser la misma)
 4. Volumen
 5. Exponemos los puertos
4. Creamos la red en global
 1. Con el nombre que hemos indicado y el driver por defecto
5. Creamos el volumen en global

Con el comando `docker-compose build` construiremos nuestra imagen, ya solo quedaría ejecutarlo



```

req.txt • 7_predict.py •
7_predict.py > ...
...
1  from torchvision import transforms
2  import torchvision
3  import torch
4  import torch.nn as nn
5  import glob
6  import cv2
7  from numpy import argmax
8  from pymongo import MongoClient
9
10 try:
11     # Conectar a La db, host y puerto
12     conn = MongoClient(host='mongo', port=27017)
13     # Obtener base de datos
14     db = conn.local
15 except:
16     pass
17

```

No olvidarnos de indicar también el nombre del host y la dirección del puerto a nuestro archivo python