Práctica 4.2: Programación distribuida en Python con Ray

Índice

1. Objetivos

El objetivo de esta práctica es iniciarse en la programación distribuida de alto nivel con Python. Se usará para ello el framework Ray, donde se explorará el potencial de las tareas, actores, así como del subsistema Ray Serve.

Para trabajar con esta práctica se utilizará el proyecto de VSCode proporcionado con la práctica anterior, basado en la imagen `ray-os` ya creada.

2. Ejercicios

Ejercicio 1: Paralelización de la clasificación de un conjunto de imágenes con Ray

En este ejercicio se realizarán distintas implementaciones de un proceso de clasificación de imágenes usando un modelo ya entrenado basado en una CNN. El desarrollo se realizará empleando un cuaderno de Jupyter, que se entregará a través del campus virtual. Tanto el conjunto de imágenes para inferencia, como el modelo se descargarán de Hugging Face, plataforma de *machine learning* (ML) y ciencia de datos que facilita a la comunidad el desarrollo, despliegue y entrenamiento de modelos de ML.

Modelo y conjunto de datos

Más concretamente, como modelo para la clasificación se utilizará el Vision Transformer (ViT) de Google, que se aplicará sobre el conjunto `train` del *dataset* tiny-imagenet. Para la carga en memoria del modelo y del *dataset* se hará uso de los módulos `transformers` y `datasets`, respectivamente, proporcionados por Hugging Face. Estos módulos ya están instalados en el entorno de trabajo basado en Docker y VSCode.

Para obtener el conjunto de datos es preciso ejecutar el siguiente código:

```
from datasets import load_dataset
tiny_imagenet = load_dataset('Maysee/tiny-imagenet')
```

Nótese que al ejecutar `load_dataset` por primera vez en una máquina con un *dataset* concreto, dicho *dataset* se descarga en local. Por lo tanto la primera ejecución de `load_dataset` tardará mucho más tiempo que invocaciones posteriores, que solo proceden a cargar en memoria el conjunto de datos (imágenes en este caso).

El objeto devuelto por la llamada anterior es de tipo `DatasetDict`, que puede usarse como un diccionario de Python para acceder a los dos conjuntos de imágenes del *dataset*: `train` (para entrenamiento) y `valid` (para validación). Al imprimir la estructura de datos `tiny_imagenet` desde un *notebook*, observaremos la siguiente salida:

```
DatasetDict({
    train: Dataset({
        features: ['image', 'label'],
        num_rows: 100000
    })
    valid: Dataset({
        features: ['image', 'label'],
        num_rows: 10000
    })
})
```

El conjunto `train` contiene 100000 muestras, frente a las 10000 del conjunto `valid`. Cada muestra incluye una imagen en formato PIL (módulo `pillow`), junto con la etiqueta (clase) de la imagen, que no se usará para el desarrollo del ejercicio.

Para acceder a una imagen aislada (objeto PIL), debemos usar su índice dentro del conjunto deseado, y acceder con la clave `"image"` en la estructura de diccionario. Así, por ejemplo si deseamos acceder a la imagen del conjunto de *train* con índice 2300, tendremos que utilizar la siguiente expresión de Python:

```
tiny_imagenet['train'][2300]['image']
```

Desde el cuaderno de Jupyter podremos visualizar la imagen al usar dicha expresión en una celda:

Para cargar el modelo para clasificación de imágenes, usaremos el módulo `transformers` de Hugging Face:

```
from transformers import pipeline

clf = pipeline("image-classification")
```

La llamada `pipeline` retorna un clasificador para poder inferir la clase de una imagen en formato PIL, entre otros. Así por ejemplo, para obtener la clase de la imagen considerada anteriormente ejecutaremos lo siguiente:

```
In [8]: clf(tiny_imagenet['train'][2300]['image'])

[{'label': 'American alligator, Alligator mississipiensis',
  'score': 0.9436624646186829},
 {'label': 'African crocodile, Nile crocodile, Crocodylus niloticus',
  'score': 0.025410419330000877},
 {'label': 'platypus, duckbill, duckbilled platypus, duck-billed platypus,
  Ornithorhynchus anatinus', 'score': 0.0012438251869753003},
 {'label': 'alligator lizard', 'score': 0.0007204426219686866},
 {'label': 'frilled lizard, Chlamydosaurus kingi',
  'score': 0.0007057994953356683}]
```

Como muestra la salida de la celda, el clasificador retorna una lista de clasificaciones ordenadas por puntuación, siendo en este caso la más acertada “American alligator”, con más de un 94 % de probabilidad.

Desarrollo del ejercicio

En este ejercicio se pide desarrollar varias versiones del proceso de clasificación de un subconjunto de imágenes de `tiny-imagenet` usando el modelo previamente introducido. Concretamente, se realizará la inferencia sobre un total de 250 imágenes del conjunto `train`seleccionadas dentro del siguiente rango: `range(0,10000,40)`. El proceso global retornará una lista de tuplas `result` con la clasificación de mayor puntuación (*score*) obtenida para cada imagen, donde cada tupla indicará (clase, puntuación).

El desarrollo se realizará empleando un cuaderno de Jupyter, que se entregará a través del campus virtual. Para cada una de las versiones a desarrollar, su ejecución se incluirá en una celda con la cabecera `%%time` para reportar el tiempo de ejecución, y así poder comparar el rendimiento de las distintas implementaciones. La ejecución de cada versión ha de incluir la carga del *dataset* y del modelo en memoria. Las versiones a desarrollar son las siguientes

- (**Versión A**) Versión secuencial de la implementación, que recorra las imágenes con un bucle. Se aconseja desarrollar una función auxiliar `get_best_class_and_score` , que acepte como parámetros el clasificador y la imagen (objeto PIL), y retorne una tupla con la mejor clasificación y su puntuación.

```
def get_best_class_and_score(classifier,pil_image):
  .... TODO ....
```

- (**Versión B**) Versión paralela de la implementación empleando tareas remotas de Ray. En concreto, se definirá una única tarea para clasificar una imagen, aceptando como parámetro el modelo y el objeto de imagen:

```
@ray.remote
def inference(classifier,image):
    return get_best_class_and_score(classifier,image)
```

  Al invocar esta tarea via `inference.remote()` se pasará como parámetro una referencia `rclf` al clasificador, que se almacenará en el *object store* de Ray:

```
clf = pipeline("image-classification")
rclf=ray.put(clf)
```

- (**Versión C**) Versión paralela de la implementación empleando actores de Ray. Se crearán tantos actores como núcleos tenga el equipo en el que estamos trabajando. Cada actor exportará métodos para realizar la inferencia, y la implementación asignará de forma rotatoria "trabajo" de inferencia a cada actor, equilibrando así la carga entre *workers*. Por motivos de eficiencia, cada actor almacenará como atributo una instancia propia del clasificador, y opcionalmente (a determinar en la construcción del actor), el *dataset* de imagenes. Esto permitirá reducir transferencia de datos entre el *driver* y los *workers* del cluster de Ray. Por simplicidad se proporciona a continuación parte de la implementación del actor:

```
@ray.remote
class InferenceEngine:
    def __init__(self,load_images=True):
        self.classifier = pipeline("image-classification")
        if load_images:
            self.dataset = load_dataset('Maysee/tiny-imagenet')['train']
        else:
            self.dataset= None

    def classify_image(self,i):
        if self.dataset is None or i>=len(self.dataset):
            return ("Error",1)

        ## A completar
        ...

    def classify_image_object(self,image):
        ## A completar
```

```
        ...
```

Como puede observarse el constructor del actor carga una instancia privada del modelo, y hace lo propio con el *dataset* si el parámetro `load_images` es `True`. El actor ofrece dos métodos remotos:

1. `classify_image()` : Retorna la tupla de (clasificación, puntuación) asociada a una imagen, especificada únicamente por su índice en el *dataset*. Esta llamada está pensada para instancias del actor con el dataset almacenado en `self.dataset`
2. `classify_image_object()` : Retorna la tupla de (clasificación, puntuación) asociada a una imagen PIL (objeto) pasada como parámetro. En este caso el actor no hará uso de `self.dataset`

De esta versión con actores se probarán dos variantes. En la primera, cada actor mantendrá el *dataset* de imágenes en memoria, y el proceso de inferencia desde el *driver* invocará el método remoto `classify_image()`. En la segunda variante, el *dataset* se cargará en el driver, y los actores se instanciarán pasando `load_images=False` en su constructor. Asimismo, se usará el método remoto `classify_image_object()` para llevar a cabo la inferencia.

Consideraciones para la implementación:

- La celda con cada variante ha de incluir la creación y la destrucción de actores (`ray.kill(actorRef)`).
- Si en algún caso se muestra error de falta de memoria (`OutOfMemory` error), se aconseja detener y relanzar el cluster de Ray, como sigue:

```
ray.shutdown()
ray.init()
```

Considerando los tiempos de ejecución obtenidos, indicar qué implementación es más eficiente y justificar el motivo. **Pista:** Se ha de tener presente que la transferencia de parámetros entre nodos/workers del cluster de Ray tiene coste no despreciable, especialmente si se trata de objetos de gran tamaño.

Ejercicio 2: Creación de servicio de información sobre imágenes basado en Ray Serve

En este ejercicio guiado se empleará Ray Serve para servir por HTTP un modelo que detecte imágenes enviadas remotamente por el usuario, identifiqué qué objeto, animal o cosa representa la imagen, y reporte información sobre ello consultando la Wikipedia.

La realización de este ejercicio requiere la instalación de dos paquetes adicionales con `pip` dentro del contenedor Docker. Para ello abriremos una terminal desde VSCode y ejecutaremos el siguiente comando:

```
(venv) /workspaces/.../PythonProject# pip install python-multipart wikipedia
```

A continuación se muestra información detallada sobre el procedimiento de envío de la petición, el modo de procesamiento de la misma en el servidor, y las partes a desarrollar de la práctica.

Envío de peticiones POST desde el cliente

El usuario realizará una petición POST HTTP a la URL donde se exporte el servicio de inferencia, incrustando la imagen en la petición HTTP. Para ello se utilizará el módulo `requests` de Python. A continuación se muestra un código de ejemplo, que realiza una petición para obtener información sobre la imagen de un martillo (mostrada más abajo), que se asume almacenada en el fichero *example-image.png* del cliente:

```
import requests

image_path = "example-image.png"
url = "http://127.0.0.1:8000/whatis/"
```

```
with open(image_path, "rb") as f:
    files = {"file": (image_path, f, "image/png")}
    response = requests.post(url, files=files)

print("Server response:", response.json())
```

Figura 1: Imagen de muestra

El servicio plenamente funcional, en ejecución, y accesible en el endpoint http://127.0.0.1:8000/whatis/ desde *localhost*, debería devolver la siguiente salida, codificada en JSON:

```
Server response: {'prediction': 'hammer', 'wikipedia info': 'A  hammer is a tool, most often
  a hand tool, consisting of a weighted "head" fixed to a long handle that is swung to deliver
  an impact to a small area of an object. This can be, for example, to drive nails into wood,
  to shape metal (as with a forge), or to crush rock. Hammers are used for a wide range of
  driving, shaping, breaking and non-destructive striking applications. Traditional disciplines
  include carpentry, blacksmithing, warfare, and percussive musicianship (as with a gong).
  \nHammering i'...}
```

Cabe destacar que las imágenes en formato PIL pueden almacenarse como ficheros PNG. De este modo, si empleamos un *dataset* de imágenes con muestras en formato PIL, podemos todavía hacer uso del servicio con una conversión previa a archivo PNG. El siguiente ejemplo pone de manifiesto cómo hacer la conversión con una imagen del *dataset* `tiny-imagenet` empleado en el ejercicio 1:

```
from datasets import load_dataset
from PIL import Image

## Carga del dataset
tiny_imagenet = load_dataset('Maysee/')

## Elección de una de las imágenes (formato PIL)
sample_image=tiny_imagenet['train'][2300]['image']

## Conversión de la imagen a PNG y almacenamiento en un fichero
sample_image.save("sample.png", format="PNG")
```

Procesamiento de peticiones

Para procesar peticiones en el servidor se usarán las siguientes tecnologías:

- Ray Serve, como framework básico para exposición del servicio
- Vision Transformer (ViT) de Google, para la clasificación de la imagen, usando dicho modelo desde el módulo `transformers` de Hugging Face, como en el ejercicio 1.

- Módulo `wikipedia` de Python, para hacer búsquedas de términos concretos en Wikipedia, y descargar contenido de páginas específicas.

Para ilustrar el uso básico del módulo `wikipedia`, consideremos el siguiente ejemplo. Supongamos que se desea buscar las páginas más frecuentes sobre "Barcelona". Para ello, se ha de usar la función `search()` de `wikipedia`, como en el siguiente ejemplo, donde se obtienen las 10 entradas más comunes.

```
In [2]: import wikipedia

In [3]: results = wikipedia.search("Barcelona")

In [4]: results
Out[4]:
 ['Barcelona',
 'FC Barcelona',
 '2024-25 FC Barcelona season',
 'Barcelona Open (tennis)',
 'Barcelona (disambiguation)',
 'RCD Espanyol',
 'FC Barcelona 6-1 Paris Saint-Germain FC',
 '1992 Summer Olympics',
 'Vicky Cristina Barcelona',
 '2025 Barcelona Open Banc Sabadell - Singles']
```

Si consultamos la página correspondiente al primer resultado de búsqueda con `page()` obtenemos un objeto de tipo `WikipediaPage`, cuyo contenido en texto plano podemos obtener usando el atributo `content`:

```
In [5]: page=wikipedia.page('Barcelona')
In [6]: page
Out[6]: <WikipediaPage 'Barcelona'>
In [7]: page.content
Out[7]: 'Barcelona ... is a city on the northeastern coast of Spain. It is the capital and
largest city of the autonomous community of Catalonia, as well as the second-most populous
municipality of Spain. With a population of 1.6 million within city limits, its urban area
extends to numerous neighbouring municipalities within the province of Barcelona and is home
to around 5.3 million people, making it the fifth most populous urban area of the European
Union after Paris, the Ruhr area, Madrid and Milan. It is one of the largest metropolises on
the Mediterranean Sea, located on the coast between the mouths of the rivers Llobregat and Besòs,
bounded to the west by the Serra de Collserola mountain range.\nAccording to tradition, Barcelona
was founded by either the Phoenicians or the Carthaginians, who had trading posts along the
Catalonian coast. In the Middle Ages, Barcelona became the capital of the County of Barcelona.
After joining with the Kingdom of Aragon to form the composite monarchy of the Crown of Aragon,
Barcelona, which continued to be the capital of the Principality of Catalonia, became the most
important city in the Crown of Aragon and its main economic and administrative centre, only to
be overtaken by Valencia, wrested from Moorish control by the Catalans, shortly before the dynastic
union between the Crown of Castile and the Crown of Aragon in 1516. Barcelona became the centre of
Catalan separatism, briefly becoming part of France during the 17th century Reapers\' War and again
in 1812 until 1814 under Napoleon. Experiencing industrialization and several workers movements
during the 19th and early 20th century, it became the capital of autonomous Catalonia in 1931 and it
was the epicenter of the revolution experienced by Catalonia during the Spanish Revolution of 1936,
until its capture by the fascists in 1939...
```

El servidor a implementar con Ray Serve deberá utilizar las citadas funciones, para obtener la información sobre la predicción con mayor puntuación que haya devuelto el clasificador de imágenes. Nótese que si la categoría (*label*) de mayor puntuación que retorna el modelo incluye la coma (por devolver varios sinónimos), será preciso quedarse con el primer término. Así por ejemplo, si la imagen se reconoce como `'American alligator, Alligator mississipiensis'`, debemos usar `'American alligator'` al realizar la búsqueda en Wikipedia.

Desarrollo del ejercicio

Desarrollar el servidor anteriormente descrito para servir el modelo de inferencia y consulta de información de imágenes con Ray Serve. A continuación se proporciona un código Python base (despliegue de Ray Serve) a completar para construir el servidor. El código incluye una serie de comentarios explicativos de las funciones/métodos a desarrollar y completar:

```
from PIL import Image
from io import BytesIO
from fastapi import FastAPI, File, UploadFile
from ray import serve
from transformers import pipeline
import ray
import wikipedia

def fetch_wikipedia_page(search_term,chars=500):
    ## Completar
    ## Si página no encontrada o no hay entradas asociadas
    ## a ese termino de búsqueda devolver "No pages found".
    ## En otro caso, retornar un string con los chars primeros
    ## caracteres de la página que aparece en primer lugar
    ## al hacer la búsqueda en Wikipedia


app = FastAPI()

@serve.deployment()
class WikipediaSearch:
    async def __call__(self, input_text):
        contents = fetch_wikipedia_page(input_text)
        return contents

@serve.deployment()
@serve.ingress(app)
class ImageClassifier:
    def __init__(self, wikipedia_search):
        ## Inicializar atributos
        self._classifier = ...
        self._wiki= ...

    def classify_image(self,image):
                ## Completar
        ## Ejecutar modelo self._classifier() sobre imagen.
        ## Si no hay resultados o puntuación (score) es < 0.80
        ## devolver None
        ## En caso contrario quedarse con el mejor resultado
        ## y con primer término (en caso de que hayq varios sinónimos
        ## separados por comas)

    @app.post("/whatis")
    async def classify(self, file: UploadFile = File(...)):
        ## Extrae fichero de imagen de petición HTTP
        contents = await file.read()
        ## Convierte contenido a imagen PIL
        image = Image.open(BytesIO(contents)).convert("RGB")
        ## Invocar modelo para obtener término de búsqueda
        search_item=self.classify_image(image)

        if search_item is None:
            return { "prediction": "unknown"}
```

```
        ## Completar conexión con modelo de WikipediaSearch
        page_contents = ...

        return {"prediction": search_item, "wikipedia info": page_contents }



wiki=WikipediaSearch.bind()
info=ImageClassifier.bind(wiki)
```

Para lanzar el servidor, se ha de ejecutar el comando `serve run` en una terminal dentro del contenedor gestionado por VSCode, pasando como parámetros el nombre del fichero Python (sin la extensión) y la variable que contiene el modelo principal que actúa de *frontend*, ambos valores separados por ":". Así, por ejemplo, si el código se incluyera en el fichero *ray-server.py* almacenado en el directorio actual, el comando a ejecutar sería: `serve run ray-server:info`. Más abajo se proporciona la salida:

```
(venv) /workspaces/.../PythonProject# serve run ray-server:info
2025-04-25 11:40:35,155 INFO scripts.py:494 -- Running import path: 'ray-server:info'.
2025-04-25 11:40:38,256 WARNING services.py:2070 -- WARNING: The object store is using /tmp
instead of /dev/shm because /dev/shm has only 67108864 bytes available. This will harm
performance! You may be able to free up space by deleting files in /dev/shm. If you are inside
a Docker container, you can increase /dev/shm size by passing '--shm-size=1.20gb' to 'docker run'
(or add it to the  run_options list in a Ray cluster config). Make sure to set this to more than
30% of available RAM. 2025-04-25 11:40:38,378 INFO worker.py:1843 -- Started a local Ray instance.
View the dashboard at http://127.0.0.1:8265 (ProxyActor pid=11132) INFO 2025-04-25 11:40:39,364
proxy 172.17.0.2 -- Proxy starting on node 21e4dd4e6bfa894e78d75a767bf7656b9cdc33b4df53d7ad9cf78801
(HTTP port: 8000).
INFO 2025-04-25 11:40:39,398 serve 10882 -- Started Serve in namespace "serve".
INFO 2025-04-25 11:40:39,406 serve 10882 -- Connecting to existing Serve app in namespace "serve".
New http options will not be applied.
(ProxyActor pid=11132) INFO 2025-04-25 11:40:39,384 proxy 172.17.0.2 -- Got updated endpoints: {}.
(ServeController pid=11135) INFO 2025-04-25 11:40:39,486 controller 11135 -- Deploying new version
of Deployment(name='WikipediaSearch', app='default') (initial target replicas: 1).
(ServeController pid=11135) INFO 2025-04-25 11:40:39,487 controller 11135 -- Deploying new version
of Deployment(name='ImageClassifier', app='default') (initial target replicas: 1).
(ProxyActor pid=11132) INFO 2025-04-25 11:40:39,489 proxy 172.17.0.2 -- Got updated endpoints:
{Deployment(name='ImageClassifier', app='default'): EndpointInfo(route='/',
app_is_cross_language=False)}.
(ProxyActor pid=11132) INFO 2025-04-25 11:40:39,517 proxy 172.17.0.2 --
Started <ray.serve._private.router.SharedRouterLongPollClient object at 0xffff840e6bd0>.
(ServeController pid=11135) INFO 2025-04-25 11:40:39,592 controller 11135 --
Adding 1 replica to Deployment(name='WikipediaSearch', app='default').
(ServeController pid=11135) INFO 2025-04-25 11:40:39,593 controller 11135 --
Adding 1 replica to Deployment(name='ImageClassifier', app='default').
(ServeReplica:default:ImageClassifier pid=11141) No model was supplied, defaulted to
google/vit-base-patch16-224 and revision 3f49326 (https://huggingface.co/google/vit-base-patch16-224).
(ServeReplica:default:ImageClassifier pid=11141) Using a pipeline without specifying a model name
and revision in production is not recommended.
(ServeReplica:default:ImageClassifier pid=11141) Device set to use cpu
INFO 2025-04-25 11:40:43,526 serve 10882 -- Application 'default' is ready at http://127.0.0.1:8000/.
[... Teclear CTRL+C para matar el servidor ...]
```

A partir de este momento, el servidor estará listo para recibir peticiones a través del puerto 8000 (puerto por defecto en Ray Serve). En particular, la URI `http://127.0.0.1:8000/whatis` es la que acepta peticiones POST para obtención de información de imágenes.

Una vez desarrollado y lanzado el servidor, el estudiante ha de generar peticiones de prueba usando un *notebook*, que deberá entregarse junto con el código servidor. Para hacer las peticiones, se aconseja hacer uso de imágenes pertenecientes a *datasets* disponibles en Hugging Face, como el utilizado en el Ejercicio 1.