

AFH

Fine Tuning



Fine Tuning

Mario Coca Atienza

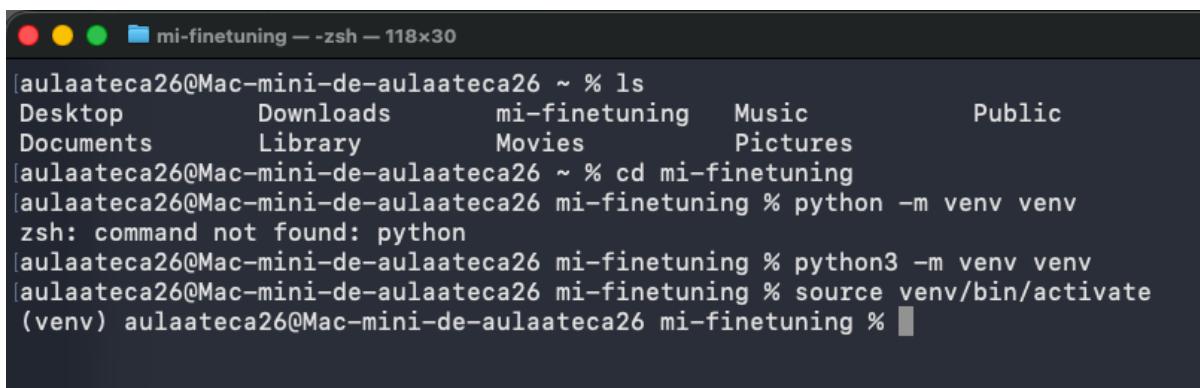
Índice

Práctica	2
Punto 1	2
Punto 2	2
Punto 3	3
Punto 4	3
Punto 5	4
Punto 6	4
Punto 7	5
Punto 8	5

Práctica

Punto 1

Lo primero que haremos es crear el directorio “mi-finetuning” y entraremos en él. Después se ejecuta “python3 -m venv venv”, y crea la carpeta “venv” dentro del proyecto, donde se guardará el entorno virtual de Python. Por último, se activa el entorno virtual con source venv/bin/activate; se comprueba que está activo porque en el prompt de la terminal aparece el prefijo (venv) antes del nombre del equipo, señalando que todos los paquetes que se instalen a partir de ahora se guardarán en este entorno aislado.

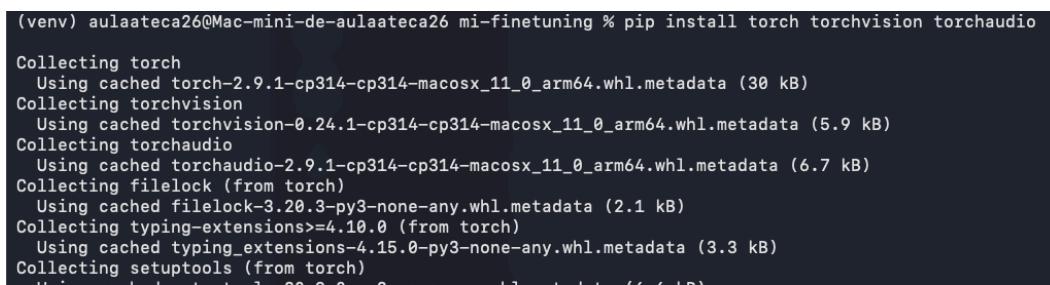


```
● ● ● mi-finetuning ~ zsh ~ 118x30
[aulaateca26@Mac-mini-de-aulaateca26 ~ % ls
Desktop Downloads mi-finetuning Music Public
Documents Library Movies Pictures
[aulaateca26@Mac-mini-de-aulaateca26 ~ % cd mi-finetuning
[aulaateca26@Mac-mini-de-aulaateca26 mi-finetuning % python -m venv venv
zsh: command not found: python
[aulaateca26@Mac-mini-de-aulaateca26 mi-finetuning % python3 -m venv venv
[aulaateca26@Mac-mini-de-aulaateca26 mi-finetuning % source venv/bin/activate
(venv) aulaateca26@Mac-mini-de-aulaateca26 mi-finetuning %
```

Punto 2

En esta captura se muestra la instalación de PyTorch en el entorno virtual del proyecto. Con el prefijo (venv) visible en la terminal, se ejecuta “pip install torch torchvision torchaudio” para instalar el framework de deep learning junto con sus módulos de visión y audio.

Se pueden observar las ruedas específicas para macOS ARM64 (macosx_11_0_arm64.whl) que pip está descargando desde caché, optimizadas para aprovechar el chip M-series del Mac. Este paso es esencial para preparar el entorno antes del fine-tuning, ya que PyTorch servirá como backend para cargar el modelo Gemma 3 y ejecutar el entrenamiento con MLX y LoRA.

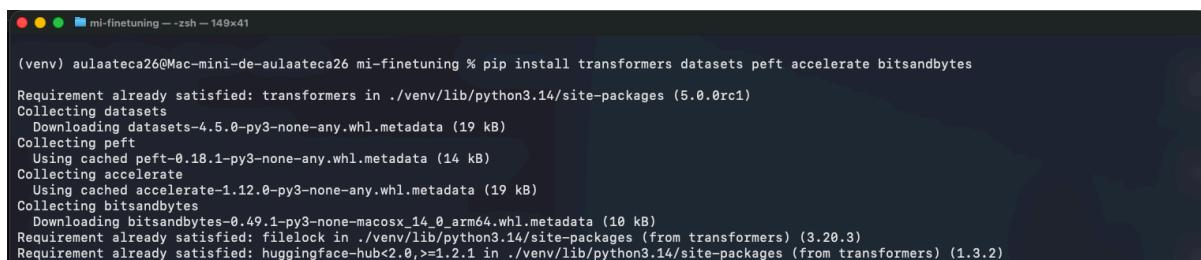


```
(venv) aulaateca26@Mac-mini-de-aulaateca26 mi-finetuning % pip install torch torchvision torchaudio
Collecting torch
  Using cached torch-2.9.1-cp314-cp314-macosx_11_0_arm64.whl.metadata (30 kB)
Collecting torchvision
  Using cached torchvision-0.24.1-cp314-cp314-macosx_11_0_arm64.whl.metadata (5.9 kB)
Collecting torchaudio
  Using cached torchaudio-2.9.1-cp314-cp314-macosx_11_0_arm64.whl.metadata (6.7 kB)
Collecting filelock (from torch)
  Using cached filelock-3.20.3-py3-none-any.whl.metadata (2.1 kB)
Collecting typing_extensions>=4.10.0 (from torch)
  Using cached typing_extensions-4.15.0-py3-none-any.whl.metadata (3.3 kB)
Collecting setuptools (from torch)
  Using cached setuptools-65.5.0-py3-none-any.whl (87 kB)
```

Punto 3

Ahora, hay que instalar las librerías esenciales para el fine-tuning de modelos de lenguaje. En el entorno virtual activo (venv), se ejecuta pip install transformers datasets peft accelerate bitsandbytes para instalar los paquetes de Hugging Face Transformers, gestión de datasets, PEFT (Parameter-Efficient Fine-Tuning), Accelerate y BitsAndBytes.

Pip está resolviendo las dependencias y descargando paquetes optimizados para macOS ARM64. Estas librerías permitirán cargar el modelo Gemma 3 desde Hugging Face, procesar los datos de entrenamiento en formato JSONL y aplicar LoRA para ajustar eficientemente solo una fracción de los parámetros del modelo.

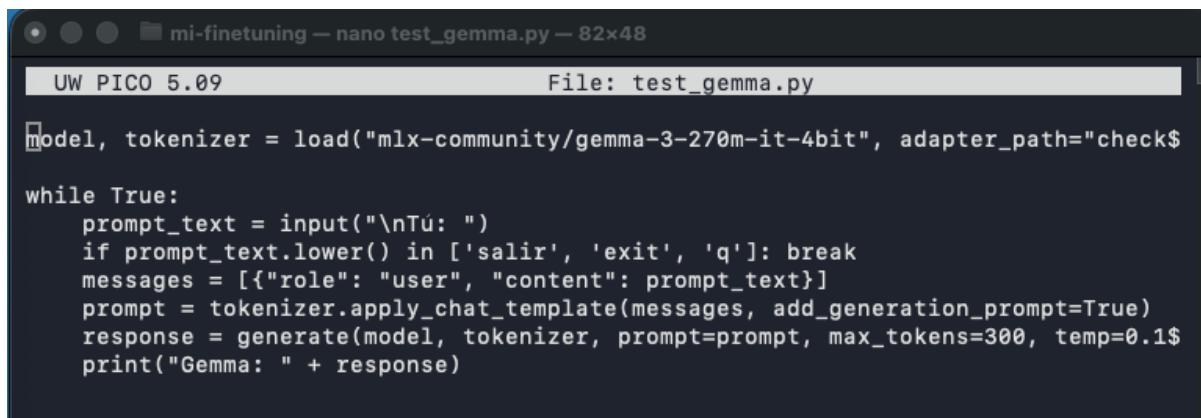


```
(venv) aulaateca26@Mac-mini-de-aulaateca26 mi-finetuning % pip install transformers datasets peft accelerate bitsandbytes
Requirement already satisfied: transformers in ./venv/lib/python3.14/site-packages (5.0.0rc1)
Collecting datasets
  Downloading datasets-4.5.0-py3-none-any.whl.metadata (19 kB)
Collecting peft
  Using cached peft-0.18.1-py3-none-any.whl.metadata (14 kB)
Collecting accelerate
  Using cached accelerate-1.12.0-py3-none-any.whl.metadata (19 kB)
Collecting bitsandbytes
  Downloading bitsandbytes-0.49.1-py3-none-macosx_14_0_arm64.whl.metadata (10 kB)
Requirement already satisfied: filelock in ./venv/lib/python3.14/site-packages (from transformers) (3.20.3)
Requirement already satisfied: huggingface-hub<2.0,>=1.2.1 in ./venv/lib/python3.14/site-packages (from transformers) (1.3.2)
```

Punto 4

Esta captura muestra el editor de código con el archivo “test_gemma.py”, que sirve para probar el modelo después del fine-tuning. Se observa el código cargando el modelo “mlx-community/gemma-3-270m-it-4bit” desde Hugging Face junto con los adaptadores LoRA generados durante el entrenamiento, usando la función load() con la ruta adapter_path.

Debajo aparece el bucle principal while True: que permite interactuar con el modelo: lee el texto del usuario, construye la lista de mensajes con roles "user" y "assistant", aplica la plantilla de chat del tokenizador con “apply_chat_template()” y genera la respuesta con “generate()”, mostrándola precedida de "Gemma:". Este script será clave para verificar que el fine-tuning ha funcionado correctamente una vez entrenado el modelo.

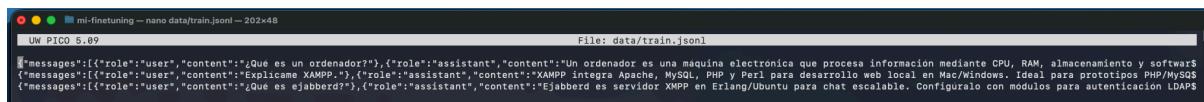


```
UW PICO 5.09                                     File: test_gemma.py
model, tokenizer = load("mlx-community/gemma-3-270m-it-4bit", adapter_path="check$"
while True:
    prompt_text = input("\nTú: ")
    if prompt_text.lower() in ['salir', 'exit', 'q']: break
    messages = [{"role": "user", "content": prompt_text}]
    prompt = tokenizer.apply_chat_template(messages, add_generation_prompt=True)
    response = generate(model, tokenizer, prompt=prompt, max_tokens=300, temp=0.1$print("Gemma: " + response)
```

Punto 5

Esta captura muestra un archivo de datos de entrenamiento en formato JSONL que contiene una conversación completa sobre gestión de correo electrónico y seguridad informática. Se observa la estructura típica de mensajes con campos "role" (user/assistant) y "content", donde el usuario pregunta sobre problemas comunes con correos y el asistente responde con explicaciones técnicas detalladas sobre dominios, certificados SSL y configuración de servidores de correo.

Este tipo de datos estructurados es exactamente lo que necesita MLX LoRA para el fine-tuning: conversaciones reales que enseñan al modelo Gemma 3 el estilo de respuesta técnica y el nivel de detalle que se espera en las respuestas del asistente entrenado.

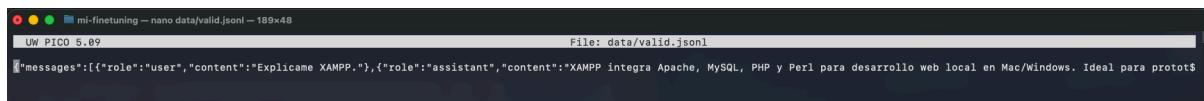


```
File: data/train.jsonl
{"messages": [{"role": "user", "content": "¿Qué es un ordenador?"}, {"role": "assistant", "content": "Un ordenador es una máquina electrónica que procesa información mediante CPU, RAM, almacenamiento y software."}, {"role": "user", "content": "Expícame XAMPP."}, {"role": "assistant", "content": "XAMPP integra Apache, MySQL, PHP y Perl para desarrollo web local en Mac/Windows. Ideal para prototipos PHP/MySQL."}, {"role": "user", "content": "¿Qué es ejabberd?"}, {"role": "assistant", "content": "Ejabberd es servidor XMPP en Erlang/Ubuntu para chat escalable. Configúralo con módulos para autenticación LDAP."}]
```

Punto 6

Esta captura muestra un segundo archivo de datos de entrenamiento en formato JSONL que contiene una conversación sobre XAMPP. Se observa la misma estructura de mensajes con campos "role" (user/assistant) y "content", donde el usuario pregunta qué es XAMPP y el asistente responde explicando que es un paquete que integra Apache, MySQL, PHP y Perl para desarrollar aplicaciones web en local.

Este ejemplo complementa el archivo anterior sobre correo electrónico, proporcionando al modelo Gemma 3 variedad en temas técnicos (desarrollo web local vs. seguridad de correo). La consistencia en el formato JSONL de ambos archivos asegura que MLX LoRA pueda procesarlos correctamente durante el fine-tuning para entrenar respuestas técnicas especializadas.



```
File: data/valid.jsonl
{"messages": [{"role": "user", "content": "Expícame XAMPP."}, {"role": "assistant", "content": "XAMPP integra Apache, MySQL, PHP y Perl para desarrollo web local en Mac/Windows. Ideal para prototipos PHP/MySQL."}]
```

Punto 7

En esta captura, se muestra el comando exacto que se ejecuta en la terminal para comenzar el proceso de fine-tuning. Se utiliza mlx_lm_lora.train con el modelo base mlx-community/gemma-3-27b-it-4bit (Gemma 3 27B cuantizado a 4 bits), especificando la ruta del dataset JSONL con --data, 100 iteraciones de entrenamiento (--iters 100), tamaño de lote 1 (--batch-size 1) y la carpeta checkpoints_ordenador donde se guardarán los adaptadores LoRA generados (--adapter-path).

La variable de entorno TOKENIZERS_PARALLELISM=false al inicio evita mensajes de advertencia del tokenizador durante el entrenamiento. Este es el comando principal que une el modelo preentrenado con los datos personalizados para crear la versión ajustada del modelo.

```
(venv) aulaateca26@Mac-mini-de-aulaateca26 mi-finetuning % TOKENIZERS_PARALLELISM=false mlx_lm.lora --model mlx-community/gemma-3-270m-it-4bit --train --data data --iters 1000 --batch-size 1 --adapter-path checkpoints_ordenador --save-every 50
```

Punto 8

Esta captura muestra el log completo del entrenamiento del modelo Gemma 3 27B en 4-bit usando MLX. Se observa la carga del modelo y dataset, seguido del progreso por iteraciones con métricas como Train loss (que disminuye progresivamente de 1.88 a 0.84), Peak mem, Tokens/sec y uso de memoria. Cada 50 iteraciones se guardan los adaptadores LoRA en checkpoints_ordenador/adapters.safetensors. Al final aparece el mensaje "Saved final weights" confirmando que el fine-tuning ha terminado exitosamente y los adaptadores están listos para usar con el script test_gemma.py.

```
(venv) aulaateca26@Mac-mini-de-aulaateca26 mi-finetuning % TOKENIZERS_PARALLELISM=false mlx_lm.lora --model mlx-community/gemma-3-270m-it-4bit --train --data data --iters 100 --batch-size 1 --adapter-path checkpoints_ordenador --save-every 50

Loading pretrained model
Fetching 10 files: 100% [██████████] | 10/10 [00:00<00:00, 128266.18it/s]
Download complete: : 0.00B [00:00, ?B/s] | 0/10 [00:00<?, ?it/s]
Loading datasets
Training
Trainable parameters: 0.629% (1.688M/268.098M)
Starting training..., iters: 100
Calculating loss...: 100% [██████████] | 1/1 [00:00<00:00, 8.54it/s]
Iter 1: Val loss 9.081, Val took 0.119s
Iter 10: Train loss 4.417, Learning Rate 1.000e-05, It/sec 10.006, Tokens/sec 497.281, Trained Tokens 497, Peak mem 0.370 GB
Iter 20: Train loss 1.042, Learning Rate 1.000e-05, It/sec 27.101, Tokens/sec 1336.075, Trained Tokens 998, Peak mem 0.370 GB
Iter 30: Train loss 0.308, Learning Rate 1.000e-05, It/sec 27.015, Tokens/sec 1323.753, Trained Tokens 1480, Peak mem 0.370 GB
Iter 40: Train loss 0.245, Learning Rate 1.000e-05, It/sec 27.038, Tokens/sec 1343.799, Trained Tokens 1977, Peak mem 0.370 GB
Iter 50: Train loss 0.201, Learning Rate 1.000e-05, It/sec 27.038, Tokens/sec 1332.955, Trained Tokens 2470, Peak mem 0.370 GB
Iter 50: Saved adapter weights to checkpoints_ordenador/adapters.safetensors and checkpoints_ordenador/0000050_adapters.safetensors.
Iter 60: Train loss 0.162, Learning Rate 1.000e-05, It/sec 26.936, Tokens/sec 1319.844, Trained Tokens 2960, Peak mem 0.379 GB
Iter 70: Train loss 0.132, Learning Rate 1.000e-05, It/sec 27.011, Tokens/sec 1331.620, Trained Tokens 3453, Peak mem 0.379 GB
Iter 80: Train loss 0.099, Learning Rate 1.000e-05, It/sec 27.018, Tokens/sec 1342.801, Trained Tokens 3950, Peak mem 0.379 GB
Iter 90: Train loss 0.069, Learning Rate 1.000e-05, It/sec 27.010, Tokens/sec 1323.467, Trained Tokens 4440, Peak mem 0.379 GB
Calculating loss...: 100% [██████████] | 1/1 [00:00<00:00, 58.03it/s]
Iter 100: Val loss 0.029, Val took 0.019s
Iter 100: Train loss 0.045, Learning Rate 1.000e-05, It/sec 26.987, Tokens/sec 1330.449, Trained Tokens 4933, Peak mem 0.379 GB
Iter 100: Saved adapter weights to checkpoints_ordenador/adapters.safetensors and checkpoints_ordenador/000100_adapters.safetensors.
Saved final weights to checkpoints_ordenador/adapters.safetensors.
(venv) aulaateca26@Mac-mini-de-aulaateca26 mi-finetuning %
```