IA

ENTRENAMIENTO DE MODELOS DE IA

Serie IA Práctica · Leandro Sepúlveda

Cómo Entrenar tu Propio Modelo de IA

Una guía general, sin atajos, para ajustar cualquier modelo de lenguaje a las necesidades de tu empresa

Leandro Juan Sepúlveda Cuevas
Software Engineer · leandrosepulveda.com

2026

Cómo Entrenar tu Propio Modelo de IA

Guía general, sin atajos, para especializar un modelo de lenguaje según las necesidades de tu empresa

Autor: Leandro Juan Sepúlveda Cuevas
Software Engineer — leandrosepulveda.com

Copyright © 2026 Leandro Juan Sepúlveda Cuevas. Todos los derechos reservados.
Queda prohibida la reproducción total o parcial de esta obra sin la autorización expresa del autor.
Primera edición: 2026. Segunda edición revisada y ampliada: 2026.

Este libro enseña un **método general** que aplica a cualquier modelo de lenguaje, abierto o comercial. Los nombres de productos puntuales que aparecen (Mistral, Llama, Qwen, OpenAI, Anthropic/Claude, Hugging Face, etc.) se usan solo como ejemplos concretos para ilustrar cada paso, y pertenecen a sus respectivos dueños. El mercado de modelos e IA cambia muy rápido: los nombres, precios y límites técnicos de este libro son de referencia. Antes de presupuestar un proyecto real, confirma los detalles vigentes en la documentación oficial del proveedor que elijas.

Contenido

01 Por que entrenar tu propio modelo

El techo de los modelos generales y cuando vale la pena cruzarlo

02 Cómo funcionan los modelos de lenguaje

Pre-entrenamiento, fine-tuning y RAG, explicados sin vueltas

03 Los tres caminos para especializar un modelo

Un mapa general antes de elegir herramienta

04 Antes de arrancar: el terreno

Hardware, software y decisiones que definen todo lo demás

05 Los datos: el verdadero trabajo

Como armar, limpiar y validar tu dataset

06 Fine-tuning de un modelo abierto, paso a paso

LoRA, QLoRA y el código completo comentado

07 Fine-tuning con un proveedor por API

De cero a un modelo propio funcionando en producción

08 Cuando el proveedor no ofrece fine-tuning

La ruta del prompt engineering y el caso de Claude

09 Evaluar tu modelo en serio

Métricas, benchmarks y LLM-as-judge

10 Alineación de preferencias

Que hacer después del SFT: DPO y RLHF explicados sin jerga

11 Entrenamiento distribuido

Cuando una sola GPU no alcanza

12 Optimizar el modelo para servir

Cuantización, formatos y motores de inferencia

13 Producción, monitoreo y glosario

Checklist final y los términos que vas a escuchar siempre

Por que entrenar tu propio modelo

CAPÍTULO 01

Si estas leyendo esto es porque ya te diste cuenta de algo: los modelos de lenguaje generales —sin importar cual uses— son impresionantes, pero no resuelven tus problemas puntuales. Saben un poco de todo, pero no son expertos en tu negocio. Y ahí esta el truco: un modelo general nunca va a entender tu jerga interna, tus procesos, tus productos o tu forma de trabajar por más que se lo expliques en el prompt. Siempre hay un techo.

Este libro enseña el método para romper ese techo, sin importar que modelo termines usando. La idea central es esta: existen tres familias de modelos según como puedes acceder a ellos, y cada familia tiene su propio camino de especialización:

- **Modelos de pesos abiertos** (open-weight): descargas los pesos y entrenas tú mismo, en tu propia infraestructura o alquilada. Ejemplos de esta familia: Mistral, Llama, Qwen, Gemma.
- **Modelos comerciales con fine-tuning gestionado**: el proveedor te deja subir tus datos y el entrena una versión privada para ti, sin que veas los pesos. Ejemplo de esta familia: la API de fine-tuning de OpenAI.
- **Modelos comerciales sin fine-tuning público**: el proveedor no ofrece ese servicio al público general, y el camino de especialización pasa por otras técnicas (prompt engineering avanzado, RAG, few-shot). Ejemplo de esta familia: Claude, de Anthropic.

Vas a ver los tres caminos completos en este libro. Los nombres de productos que uso son **ejemplos**, elegidos porque son representativos de cada familia en 2026 —no la única opción. El método que aprendas acá te sirve igual si el año que viene cambias de proveedor o aparece un modelo nuevo, porque lo que cambia es el nombre, no la lógica.

> **ANTES DE ARRANCAR**
>
> Fine-tuning no es magia ni es la primera opción. En el capítulo 3 vas a ver cuando conviene y cuando estas perdiendo el tiempo (y la plata). Si vienes con la idea fija de "quiero entrenar un modelo", te pido que la guardes un rato y sigas leyendo con la cabeza abierta.

Al final de este libro vas a tener un modelo ajustado con tus datos, evaluado con métricas serias y listo para producción —sin importar si termina siendo un modelo abierto entrenado por tú, uno ajustado via API, o un modelo comercial guiado con un buen system prompt. No te prometo que sea fácil, entrenar modelos tiene su parte tediosa, sobre todo la de los datos, pero te prometo que es algo que puedes hacer tú, sin ser un doctorado en machine learning.

Cómo funcionan los modelos de lenguaje

CAPÍTULO 02

La analogía del aprendiz de cocina

Imagina que quieres que un aprendiz aprenda a cocinar en tu restaurante. Primero le muestras miles de recetas generales del mundo entero —eso es el pre-entrenamiento. Después le ensenas las recetas específicas de tu casa, con tus ingredientes y tu estilo —eso es el fine-tuning. Y por ultimo le dices como comportarse con los clientes y que cosas nunca debe hacer —eso es el alignment, el ajuste de comportamiento y seguridad que vemos en el capítulo 10.

Los modelos de lenguaje (LLMs) funcionan prediciendo la siguiente palabra, una y otra vez. Suena trivial, pero cuando el modelo tiene miles de millones de parámetros y fue entrenado con una porción enorme de todo lo escrito en internet, esa capacidad de "adivinar la próxima palabra" se convierte en algo que parece razonar. Esto es cierto para cualquier LLM moderno, no importa quien lo entrene.

Pre-entrenamiento, fine-tuning y RAG: las tres patas

Pre-entrenamiento: el modelo aprende de una cantidad gigantesca de texto. Toma meses y cuesta millones de dólares en computo. Esto ya lo hacen los laboratorios grandes (sin importar cual elijas) — tú no vas a hacer esto, ni falta que hace.

Fine-tuning: agarras un modelo ya entrenado y lo sigues entrenando, pero ahora con tus datos específicos. Es un ajuste fino de los "pesos" internos del modelo. Esto es lo que vamos a hacer en la mayor parte de este libro, y el método es el mismo sin importar que modelo base elijas.

RAG (Retrieval Augmented Generation): acá no tocas el modelo para nada. En cada consulta, le pasas los documentos relevantes como contexto para que responda con esa información a mano. Es más barato, más rápido de implementar, y más fácil de mantener actualizado —pero depende de que la búsqueda encuentre el documento correcto.

> **REGLA PRÁCTICA**
>
> Si necesitas que el modelo "sepa algo de memoria" —como tu catálogo de productos o tu forma de escribir — piensa en fine-tuning. Si necesita consultar información que cambia todo el tiempo —precios, stock, documentos que se actualizan— usa RAG. En la práctica, la mayoría de los proyectos serios terminan combinando ambos: RAG para los datos frescos, fine-tuning para el tono y el comportamiento.

Que modelo elegir para tu empresa

No existe "el mejor modelo" en abstracto. Existe el mejor modelo para tu caso, tu presupuesto y tus datos. Esta tabla te da un punto de partida genérico según la familia (los nombres son ejemplos representativos, y los precios son aproximados y cambian seguido):

Familia	Como se accede	Costo típico	Ideal para
Open-weight chico (7-9B)	Local (GPU 16GB) o alquilada	Gratis + GPU	Datos sensibles, on-premise, control total
Comercial con fine-tuning gestionado	API del proveedor	Bajo-medio por millón de tokens	Mayoría de casos de empresa, sin gestionar infraestructura
Comercial de razonamiento avanzado	API del proveedor	Medio-alto	Razonamiento complejo, contexto largo
Open-weight orientado a código	Local o alquilada	Gratis-bajo	Tareas de código, uso 100% local

Anota esto porque va a ser importante: no todos los proveedores comerciales ofrecen fine-tuning público tradicional. Algunos si (lo vemos en el capítulo 7), otros no (lo vemos en el capítulo 8). No es un problema, son dos caminos distintos con el mismo destino.

Cuando NO conviene hacer fine-tuning

- Cuando solo necesitas que el modelo recuerde datos factuales que cambian —ahí RAG es más barato y más fácil de mantener.
- Cuando no tienes al menos 100-200 ejemplos de buena calidad para la tarea que quieres enseñar.
- Cuando todavía no probaste a fondo con un buen system prompt —muchas veces alcanza y te ahorras semanas.
- Cuando no tienes forma de medir objetivamente si el fine-tuning mejoro algo (sin métrica, no sabes si funciono —ver capítulo 9).

> **ATENCIÓN**
>
> La regla de oro: primero prueba con prompt engineering. Si no alcanza, prueba con RAG. Si sigue sin alcanzar, ahí sí considera el fine-tuning. Saltar directo a entrenar un modelo sin haber probado lo anterior es la forma más común de quemar tiempo y plata en este tipo de proyectos.

Los tres caminos para especializar un modelo

CAPÍTULO 03

Antes de tocar una línea de código, conviene tener clarísimo cual de los tres caminos te conviene. Esta decisión se toma una sola vez al principio del proyecto, y equivocarse acá cuesta semanas más adelante.

Camino A: modelo de pesos abiertos, entrenado por tú

Descargas un modelo publicado por un laboratorio (Mistral, Llama, Qwen, Gemma son ejemplos típicos en 2026) y lo entrenas con tus datos, en tu propia GPU o una alquilada. Tienes control total: los pesos son tuyos, puedes correrlo sin conexión a internet, y no hay límite de uso más allá de tu hardware. La contra: tú te encargas de todo —infraestructura, mantenimiento, actualizaciones.

Elige este camino si tus datos son sensibles y no pueden salir de tu red, si el volumen de uso es tan alto que pagar por token te sale más caro que tener tu propia GPU, o si necesitas correr el modelo sin conexión.

Camino B: modelo comercial con fine-tuning gestionado

Subes tus datos a la plataforma del proveedor, el entrena una versión privada para ti, y la usas a través de su API con un identificador propio. No ves los pesos, no administras GPU, pero tampoco tienes ese control total del camino A.

Elige este camino si quieres el beneficio de un modelo especializado sin la carga operativa de mantener infraestructura, y tu volumen de uso no justifica todavía comprar o alquilar GPU propia.

Camino C: modelo comercial guiado por prompt (sin fine-tuning)

Algunos proveedores no ofrecen fine-tuning público tradicional. En ese caso, la especialización pasa por otro lado: un system prompt muy bien armado, ejemplos concretos (few-shot), y RAG para el conocimiento que cambia. Sorprendentemente, para la mayoría de las tareas de negocio este camino se acerca mucho al resultado de un fine-tuning real, y es el que menos mantenimiento pide.

> **COMO USAR ESTE LIBRO**
>
> No hace falta que elijas un solo camino para siempre. Los capítulos 6, 7 y 8 muestran cada uno de punta a punta, con el mismo caso de uso de ejemplo (un asistente de atención al cliente), para que puedas comparar resultados y decidir con datos propios, no con suposiciones.

Criterio	Camino A (abierto)	Camino B (API gestionado)	Camino C (prompt)
Control de los datos	Total	Depende del proveedor	Depende del proveedor

Esfuerzo de infraestructura	Alto	Bajo	Ninguno
Costo con volumen alto	Bajo (fijo)	Medio-alto (por uso)	Medio (por uso)
Tiempo hasta el primer resultado	Días	Horas	Minutos
Mantenimiento a largo plazo	Alto	Bajo	Muy bajo

Antes de arrancar: el terreno

CAPÍTULO 04

Hardware: cuanto necesitas de verdad

Este capítulo aplica al Camino A. Para hacer fine-tuning de un modelo abierto de 7 a 9 mil millones de parámetros con QLoRA (la técnica que vamos a usar) necesitas al menos 16GB de VRAM. Si no tienes GPU propia —y la mayoría de la gente no la tiene— puedes alquilar por hora en servicios como RunPod, Lambda Labs, Vast.ai o Google Colab Pro. El costo típico ronda entre 30 y 80 centavos de dólar por hora de GPU, y una corrida de entrenamiento con un dataset chico suele tardar entre 30 minutos y unas pocas horas.

Software que vas a necesitar

- Python 3.10 o superior instalado
- pip, el gestor de paquetes de Python
- Git
- Una cuenta en Hugging Face (es gratis)
- Opcional: cuenta en Weights & Biases (wandb) para monitorear el entrenamiento en tiempo real

Instalando el entorno paso a paso

Primero, crea un entorno virtual de Python para no mezclar dependencias con otros proyectos:

```
python -m venv venv_training
venv_training\Scripts\activate # en Windows
source venv_training/bin/activate # en Mac/Linux

pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu121
pip install transformers accelerate peft bitsandbytes trl
pip install datasets wandb scikit-learn
```

Si no tienes GPU y vas a alquilar una en la nube, instala la versión de PyTorch sin CUDA en tu maquina local (solo para preparar los datos) y usa el entorno con GPU solo para entrenar:

```
pip install torch torchvision torchaudio
pip install transformers accelerate peft trl datasets wandb
```

Verifica que todo este funcionando antes de seguir:

```
python -c "import torch; print(f'CUDA disponible: {torch.cuda.is_available()}')"
```

Descargando un modelo base

El código de este libro usa un modelo genérico de 7B parámetros como ejemplo —piensa que en tu proyecto real podrías reemplazar el nombre por Mistral, Llama, Qwen o el que corresponda, el flujo de trabajo es igual para todos:

```
pip install huggingface_hub
huggingface-cli login

# En Python:
from huggingface_hub import snapshot_download
snapshot_download(
    repo_id="mistralai/Mistral-7B-Instruct-v0.3",  # ejemplo: reemplaza por tu modelo
elegido
    local_dir="./models/modelo-base"
)
```

> **NOTA**
>
> Algunos modelos en Hugging Face son "gated": tienes que aceptar una licencia en la web antes de poder descargarlos con tu token. Si te da error de permisos, ese suele ser el motivo.

Formatos de datos según donde entrenes

Cada modelo y cada plataforma espera los datos de entrenamiento en un formato particular. Para la mayoría de los modelos abiertos tipo "instruct":

```
[INST] {instruccion del usuario} [/INST]
{respuesta esperada}</s>
```

Para plataformas comerciales con fine-tuning gestionado, el formato suele ser de mensajes tipo chat, en JSONL:

```
{"messages": [
  {"role": "system", "content": "Eres un asistente experto en..."},
  {"role": "user", "content": "Pregunta del usuario"},
  {"role": "assistant", "content": "Respuesta esperada"}
]}
```

Guardar bien este detalle te ahorra dolores de cabeza más adelante: mezclar formatos es de los errores más comunes de gente que arranca con esto.

Los datos: el verdadero trabajo

CAPÍTULO 05

El ingrediente secreto: datos de calidad

Los datos son lo más importante de todo el proceso, más que la GPU, más que el código, más que el modelo base que elijas. Puedes tener la mejor infraestructura del mundo, pero si tus datos son malos, el resultado va a ser malo. No hay atajo para esto, y esto es cierto sin importar el camino (A, B o C) que hayas elegido en el capítulo 3.

Un buen dataset tiene tres características:

- **Relevante:** cada ejemplo representa exactamente lo que el modelo va a enfrentar en producción, no casos inventados o forzados.
- **Consistente:** el tono, el formato y el estilo son parejos en todos los ejemplos.
- **Suficiente:** no hay un número mágico universal, pero entre 200 y 500 ejemplos de alta calidad es un buen piso para arrancar a ver resultados.

Armando tu primer dataset desde cero

Supongamos que trabajas en una empresa de logística y quieres un modelo que responda preguntas sobre el seguimiento de paquetes. Un ejemplo en formato instruct:

```
[INST] Donde esta mi paquete con tracking AR123456789? [/INST]
El paquete AR123456789 salio de nuestro centro de distribucion en Buenos Aires
y esta programado para llegar el 15 de marzo.</s>
```

Y el mismo tipo de ejemplo en formato JSONL de mensajes:

```
{"messages": [
  {"role": "system", "content": "Eres un asistente de logistica."},
  {"role": "user", "content": "Donde esta mi paquete?"},
  {"role": "assistant", "content": "Necesito el numero de tracking para consultar el
estado."}
]}
```

Limpieza y validación: la parte aburrida que nadie se salta

- Elimina duplicados: dos ejemplos casi iguales confunden al modelo más de lo que ayudan.
- Corrige errores ortográficos: el modelo los va a aprender y repetir tal cual.
- Normaliza formatos: todas las fechas de la misma forma, todos los montos con el mismo formato de moneda.

- Saca cualquier dato personal o sensible de clientes reales antes de entrenar con eso.
- Balancea las categorías: si tienes 100 ejemplos de tracking y solo 5 de devoluciones, el modelo va a ser malísimo respondiendo sobre devoluciones.

Cuando tienes pocos ejemplos: aumento de datos

Si no llegas a los 100 ejemplos mínimos, puedes generar variaciones sintéticas usando un modelo grande para que te ayude a redactar más ejemplos a partir de los que ya tienes:

```
# Prompt para generar datos sinteticos
prompt = f"""
Genera 10 variaciones de esta conversacion para fine-tuning:
Pregunta: {pregunta_original}
Respuesta: {respuesta_original}
Cada variacion debe tener una redaccion distinta pero el mismo significado.
"""
```

> **CUIDADO**
>
> Los datos sintéticos son una herramienta, no un atajo total: el modelo grande que usas para generarlos puede alucinar o meter información incorrecta sin que te des cuenta. Siempre revisa a mano lo que se genera antes de sumarlo al dataset final —nunca lo cargues directo sin mirar.

Separando entrenamiento, validación y prueba

Un error muy común de gente que arranca: entrenar con el 100% de los datos y después "probar" con los mismos ejemplos que ya vio el modelo. Eso no mide nada real. Separa siempre tu dataset en tres partes antes de entrenar:

- **Entrenamiento (70-80%):** lo que el modelo efectivamente aprende.
- **Validación (10-15%):** lo que usas durante el entrenamiento para chequear que no este memorizando de más (overfitting).
- **Prueba (10-15%):** datos que el modelo nunca vio, reservados exclusivamente para la evaluación final del capítulo 9.

```
from sklearn.model_selection import train_test_split
import json

with open("datos_completos.jsonl") as f:
    datos = [json.loads(linea) for linea in f]

train, temp = train_test_split(datos, test_size=0.25, random_state=42)
val, test = train_test_split(temp, test_size=0.4, random_state=42)

print(f"Train: {len(train)} | Val: {len(val)} | Test: {len(test)}")
```

Fine-tuning de un modelo abierto, paso a paso

CAPÍTULO 06

Este capítulo desarrolla el Camino A completo. El modelo que usamos como ejemplo es Mistral 7B Instruct, pero el código funciona igual, cambiando solo el `repo_id`, para Llama, Qwen, Gemma o cualquier otro modelo abierto compatible con la librería `transformers`.

La técnica LoRA (Low-Rank Adaptation)

Entrenar un modelo completo desde cero (lo que se llama "full fine-tuning") es caro y lento, y requiere una cantidad de GPU que la mayoría de las empresas no tiene ni necesita. LoRA es una técnica que congela el modelo original y le agrega pequeños "adaptadores" entrenables por encima. El resultado: entrenas con muchísima menos GPU, más rápido, y con resultados comparables para la mayoría de los casos de uso reales.

Piénsalo así: en vez de reformar toda la cocina, cambias solo los cuchillos y las sartenes. El resto de la cocina —el modelo base— queda intacto.

QLoRA es una versión todavía más eficiente que cuantiza el modelo a 4 bits antes de entrenar. Con QLoRA puedes hacer fine-tuning de un modelo de 7B con apenas 12-16GB de VRAM, algo que hace unos años hubiera sido impensado sin una GPU de servidor.

El script completo de entrenamiento

```
import torch
from transformers import (
    AutoModelForCausalLM, AutoTokenizer,
    BitsAndBytesConfig, TrainingArguments
)
from peft import LoraConfig, get_peft_model, prepare_model_for_kbit_training
from trl import SFTTrainer
from datasets import load_dataset

# 1. Configuración de cuantización (así ahorramos memoria de GPU)
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.float16,
    bnb_4bit_use_double_quant=True,
)

# 2. Cargar el modelo base y el tokenizador (cambia el repo_id por tu modelo elegido)
modelo = AutoModelForCausalLM.from_pretrained(
    "mistralai/Mistral-7B-Instruct-v0.3",
    quantization_config=bnb_config,
    device_map="auto",
)
tokenizer = AutoTokenizer.from_pretrained("mistralai/Mistral-7B-Instruct-v0.3")
tokenizer.pad_token = tokenizer.eos_token

# 3. Configurar LoRA sobre el modelo cuantizado
modelo = prepare_model_for_kbit_training(modelo)
lora_config = LoraConfig(
    r=16, lora_alpha=32,
    target_modules=["q_proj", "k_proj", "v_proj", "o_proj"],
    lora_dropout=0.05, bias="none", task_type="CAUSAL_LM",
)
modelo = get_peft_model(modelo, lora_config)

# 4. Como formateamos cada ejemplo del dataset
def formatear(ejemplo):
    return f'[INST] {ejemplo["instruction"]} [/INST]\n{ejemplo["response"]}</s>'

# 5. Configurar y lanzar el entrenamiento
args = TrainingArguments(
    output_dir="./modelo-ajustado",
    num_train_epochs=3,
    per_device_train_batch_size=4,
    gradient_accumulation_steps=4,
    warmup_steps=10, logging_steps=10,
    learning_rate=2e-4, fp16=True,
    eval_strategy="steps", eval_steps=50,
)
dataset = load_dataset("json", data_files="datos_train.jsonl", split="train")
dataset_val = load_dataset("json", data_files="datos_val.jsonl", split="train")

trainer = SFTTrainer(
    model=modelo, args=args,
    train_dataset=dataset, eval_dataset=dataset_val,
    tokenizer=tokenizer, formatting_func=formatear, max_seq_length=1024,
)
trainer.train()
```

```
trainer.save_model("./modelo-ajustado-final")
print("Entrenamiento completado")
```

Corriendo el entrenamiento

```
python entrenar.py
```

El entrenamiento puede tardar desde 30 minutos (con 200 ejemplos en una GPU tipo RTX 4090) hasta varias horas (con 1000 o más ejemplos, o GPUs más modestas). Vas a ver algo así en la consola:

```
Epoch 1/3: 100%|##########| 50/50 [12:30<00:00, loss=0.82, eval_loss=0.79]
Epoch 2/3: 100%|##########| 50/50 [12:15<00:00, loss=0.51, eval_loss=0.55]
Epoch 3/3: 100%|##########| 50/50 [12:20<00:00, loss=0.38, eval_loss=0.61]
```

NOTA

El "loss" es el número que te dice que tan lejos esta la predicción del modelo de la respuesta correcta — cuanto más bajo, mejor. Fíjate que en el ejemplo de arriba el `eval_loss` (el loss medido contra datos de validación) empieza a subir en la época 3 mientras el loss de entrenamiento sigue bajando: eso es una señal clásica de overfitting, el modelo esta memorizando en vez de aprender. Cuando veas esto, quédate con el checkpoint de la época anterior.

Fine-tuning con un proveedor por API

CAPÍTULO 07

Este capítulo desarrolla el Camino B: un proveedor comercial que ofrece fine-tuning gestionado. Usamos la API de OpenAI como ejemplo representativo de esta familia, porque en 2026 es una de las más usadas, pero el flujo (subir datos, lanzar un job, monitorear, usar el modelo resultante) es el mismo que vas a encontrar en cualquier otro proveedor que ofrezca este servicio.

El camino más simple de los tres

Si el Camino A te pareció con demasiadas piezas moviéndose a la vez, este es la versión simplificada: no administras GPU, no instalas nada pesado, todo pasa a través de la API. A cambio, pagas por uso y el modelo ajustado vive dentro de la infraestructura del proveedor, no en la tuya.

> **NOTA SOBRE MODELOS DISPONIBLES**
>
> Que modelos están habilitados para fine-tuning cambia seguido —los proveedores van rotando que familia de modelos ofrecen para esto. Antes de arrancar un proyecto real, revisa siempre la documentación oficial de fine-tuning de tu proveedor elegido: esta es el area que más rápido cambia de todo el libro.

Preparando los datos

El formato es el mismo que vimos en el capítulo 4, mensajes tipo chat en JSONL:

```
{"messages": [
  {"role": "system", "content": "Eres un asistente de logistica."},
  {"role": "user", "content": "Donde esta mi paquete?"},
  {"role": "assistant", "content": "Necesito el numero de tracking para consultar el
estado."}
]}
```

Creando el fine-tuning con la API

```
import os
from openai import OpenAI

client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))

# 1. Subir el archivo de entrenamiento
resp = client.files.create(
    file=open("datos_openai.jsonl", "rb"),
    purpose="fine-tune"
)
file_id = resp.id
print(f"Archivo subido: {file_id}")

# 2. Crear el job de fine-tuning
resp = client.fine_tuning.jobs.create(
    training_file=file_id,
    model="gpt-4.1-mini-2025-04-14",  # confirma el modelo vigente en la doc oficial
    hyperparameters={"n_epochs": 3},
)
job_id = resp.id
print(f"Fine-tuning iniciado: {job_id}")

# 3. Monitorear el progreso
import time
while True:
    estado = client.fine_tuning.jobs.retrieve(job_id)
    print(f"Estado: {estado.status}")
    if estado.status in ["succeeded", "failed", "cancelled"]:
        break
    time.sleep(30)

# 4. Usar el modelo ya ajustado
modelo_fine = estado.fine_tuned_model
resp = client.chat.completions.create(
    model=modelo_fine,
    messages=[{"role": "user", "content": "Como devuelvo un producto?"}]
)
print(resp.choices[0].message.content)
```

NOTA

El fine-tuning gestionado no modifica el modelo base para todo el mundo: crea una versión privada, con su propio identificador, que solo tú puedes usar con tu API key.

Cuanto cuesta esto en la práctica

El costo de fine-tuning gestionado tiene dos partes: el entrenamiento en si (se cobra por token procesado durante el entrenamiento) y después el uso normal del modelo ya ajustado, que suele salir un poco más caro por token que el modelo base equivalente. Para un dataset de 200-500 ejemplos cortos, el entrenamiento en si suele costar unos pocos dólares. Lo que realmente pesa en el presupuesto es el uso en producción, no el entrenamiento.

Cuando el proveedor no ofrece fine-tuning

CAPÍTULO 08

Este capítulo desarrolla el Camino C. Usamos Claude, de Anthropic, como ejemplo representativo, porque en 2026 es uno de los modelos comerciales de referencia que no ofrece fine-tuning tradicional al público a través de su API estándar —pero el mismo enfoque aplica a cualquier proveedor en esa misma situación.

Por que este camino existe y por que importa menos de lo que parece

Existen opciones limitadas de personalización para clientes empresariales grandes a través de convenios especiales o de plataformas en la nube de terceros, pero no es algo que tú como desarrollador vayas a activar con una llamada a la API un martes por la tarde.

La buena noticia es que esto importa menos de lo que parece. Los modelos de esta familia suelen tener una ventana de contexto enorme y son muy buenos siguiendo instrucciones largas y detalladas, así que un system prompt bien escrito —combinado con ejemplos concretos— puede acercarse muchísimo al resultado de un fine-tuning para la mayoría de las tareas de negocio.

Como armar un system prompt que realmente funcione

```
system_prompt = """
Eres un asistente de logistica para [Empresa X].

## Tu personalidad
- Eres profesional pero cercano
- Usas un tono amable
- Respondes siempre en español

## Reglas estrictas
1. NUNCA inventes numeros de tracking
2. Si un cliente esta enojado, disculpate primero
3. Siempre ofrece una solución, no solo una explicación

## Información que debes saber
- Política de devoluciones: 30 días, producto sin abrir
- Horarios de atención: Lunes a Viernes de 9 a 18
- Tracking: disponible en la web o en la app
"""
```

Few-shot prompting y cache de contexto

Además del system prompt, puedes incluir ejemplos concretos de conversaciones bien resueltas en cada llamada —eso se llama few-shot prompting. Y muchos proveedores de esta familia ofrecen un

mecanismo de cache de prompts, que reduce el costo de reenviar un system prompt largo en cada llamada porque no hay que volver a procesarlo desde cero cada vez:

```
respuesta = client.messages.create(
    model="claude-sonnet-4-6",  # confirma el identificador vigente en la doc oficial
    system=[{
        "type": "text",
        "text": system_prompt,
        "cache_control": {"type": "ephemeral"}
    }],
    messages=[{"role": "user", "content": consulta}]
)
```

> **NOTA**
>
> El nombre exacto del modelo (el "identificador") cambia cada vez que el proveedor saca una versión nueva. Revisa siempre la documentación oficial de la API antes de poner un identificador fijo en producción.

Comparando los tres caminos con tu propio caso

Haz una prueba honesta con 20-30 preguntas reales de tu caso de uso y compara las tres opciones: un modelo comercial guiado con system prompt y few-shot (Camino C), un modelo comercial fine-tuneado con tus datos (Camino B), y un modelo abierto fine-tuneado corriendo local (Camino A). En la práctica: para tareas simples, con menos de 10 reglas de negocio, un buen system prompt suele alcanzar y sale más barato de mantener. Para tareas complejas —más de 50 reglas, o conocimiento muy específico que no entra cómodo en un prompt— el fine-tuning empieza a ganar terreno, sobre todo en costo por consulta cuando el volumen es alto.

Evaluar tu modelo en serio

CAPÍTULO 09

"Le pregunte tres cosas y anduvo bien" no es una evaluación, es una anécdota. Este capítulo te da las herramientas para medir de forma objetiva si tu modelo —sin importar el camino que hayas elegido— realmente mejoro, y cuanto.

Métricas automáticas: un primer filtro rápido

Perplexity: mide que tan "sorprendido" esta el modelo por el texto real de tu set de prueba. Cuanto más baja, mejor —el modelo predice con más confianza lo que efectivamente viene después. Es rápida de calcular pero no te dice nada sobre si la respuesta es correcta o útil, solo sobre la fluidez.

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer

modelo = AutoModelForCausalLM.from_pretrained("./modelo-ajustado-final")
tokenizer = AutoTokenizer.from_pretrained("./modelo-ajustado-final")

def calcular_perplexity(texto):
    entradas = tokenizer(texto, return_tensors="pt")
    with torch.no_grad():
        salida = modelo(**entradas, labels=entradas["input_ids"])
    return torch.exp(salida.loss).item()

textos_prueba = [ej["response"] for ej in dataset_test]
perplexities = [calcular_perplexity(t) for t in textos_prueba]
print(f"Perplexity promedio: {sum(perplexities)/len(perplexities):.2f}")
```

BLEU y ROUGE: comparan la respuesta generada por el modelo contra una respuesta "de referencia" escrita por un humano, midiendo cuanto se solapan las palabras y frases. BLEU es más estricto con el orden exacto, ROUGE es más permisivo. Son útiles para tareas donde existe una respuesta esperada relativamente fija, como resúmenes o traducciones, y menos útiles para conversaciones abiertas donde hay muchas formas validas de responder bien.

```
from rouge_score import rouge_scorer

scorer = rouge_scorer.RougeScorer(['rouge1', 'rougeL'], use_stemmer=True)

def evaluar_rouge(generada, referencia):
    return scorer.score(referencia, generada)

resultado = evaluar_rouge(respuesta_modelo, respuesta_esperada)
print(f"ROUGE-L: {resultado['rougeL'].fmeasure:.3f}")
```

LLM-as-judge: cuando la respuesta correcta no es única

Para la mayoría de las tareas de negocio (atención al cliente, redacción, análisis) hay muchas formas validas de responder bien, y BLEU/ROUGE no capturan eso. La técnica que se volvió estándar en 2026 es usar un modelo grande como "juez" que compare las respuestas según criterios que tú defines:

```
prompt_juez = f"""
Eres un evaluador experto. Compara estas dos respuestas a la misma pregunta
y decida cuál es mejor según: precision, tono, completitud y utilidad.

Pregunta: {pregunta}
Respuesta A (modelo base): {respuesta_a}
Respuesta B (modelo ajustado): {respuesta_b}

Responde SOLO con JSON: {{"ganador": "A"|"B"|"empate", "razon": "..."}}
"""
# Envia este prompt a un modelo grande (distinto del que estas evaluando)
# y agrega resultados sobre tu set de prueba completo (50-100 casos)
```

ATENCIÓN

Usa siempre un modelo distinto (idealmente más grande o de otra familia) para juzgar, nunca el mismo modelo que estas evaluando —tiende a favorecerse a si mismo. Y hace la comparación en ambos ordenes (A primero, después B primero) para descartar sesgos de posición.

Benchmarks estándar: cuando compararte con el resto del mundo

Si tu modelo va a hacer tareas de conocimiento general además de tu tarea específica, tiene sentido correrlo contra benchmarks públicos como MMLU (conocimiento general), HumanEval (código) o GSM8K (razonamiento matemático), usando frameworks de evaluación como `lm-evaluation-harness`. Esto te dice si el fine-tuning empeoró capacidades generales que el modelo tenía antes —un efecto colateral real llamado "olvido catastrófico".

```
# Instalacion y uso básico de lm-evaluation-harness
pip install lm-eval

lm_eval --model hf \
    --model_args pretrained=./modelo-ajustado-final \
    --tasks mmlu,gsm8k \
    --device cuda:0 \
    --batch_size 8
```

El checklist de evaluación completo

- ☐ Perplexity calculada sobre el set de prueba (nunca sobre datos de entrenamiento)
- ☐ ROUGE/BLEU si tu tarea tiene respuestas de referencia razonablemente fijas
- ☐ Evaluación LLM-as-judge comparando modelo base vs. modelo ajustado, en al menos 50 casos

- ☐ Benchmark de conocimiento general para detectar olvido catastrófico, si aplica a tu caso
- ☐ Revisión manual de al menos 20 respuestas por una persona del equipo

Alineación de preferencias

CAPÍTULO 10

Que hacer después del SFT

Todo lo que vimos hasta acá es fine-tuning supervisado (SFT): le muestras al modelo ejemplos de "pregunta correcta → respuesta correcta". Eso funciona muy bien para enseñarle conocimiento y formato, pero tiene un límite: el modelo aprende a imitar tus ejemplos, no necesariamente a elegir la mejor respuesta entre varias posibles cuando hay matices de estilo o de juicio.

Para ese ultimo ajuste fino de comportamiento existe una familia de técnicas de alineación de preferencias. Las dos más conocidas en 2026 son RLHF y DPO.

RLHF: la versión clásica, en tres pasos

1. **SFT:** el paso que ya viste, entrenar con ejemplos de buena calidad.
2. **Modelo de recompensa:** entrenas un segundo modelo que aprende a puntuar respuestas, mostrándole pares donde un humano eligió cual respuesta prefiere.
3. **Optimización por refuerzo:** ajustas el modelo principal usando ese modelo de recompensa como guía, con un algoritmo como PPO, para que genere cada vez respuestas mejor puntuadas.

RLHF funciona muy bien pero es caro y complejo: necesitas entrenar dos modelos y el proceso de optimización por refuerzo es inestable y difícil de depurar. Por eso, en la práctica, la mayoría de los equipos que no son un laboratorio grande usan DPO.

DPO: la alternativa que se volvió el estándar práctico

DPO (Direct Preference Optimization) llega al mismo resultado sin entrenar un modelo de recompensa separado ni usar aprendizaje por refuerzo. En cambio, ajusta el modelo directamente usando pares de respuestas (una preferida, una rechazada) con una función de perdida matemática que empuja al modelo hacia las respuestas preferidas. Es mucho más simple de implementar y más estable de entrenar.

```
from trl import DPOTrainer, DPOConfig
from datasets import Dataset

# Cada ejemplo tiene: prompt, respuesta preferida, respuesta rechazada
datos_preferencias = Dataset.from_dict({
    "prompt": ["Como cancelo mi pedido?"],
    "chosen": ["Entra a 'Mis pedidos' y toca 'Cancelar'. Es inmediato y sin costo dentro de
las 2 horas."],
    "rejected": ["No se puede cancelar un pedido una vez hecho."],
})

config = DPOConfig(
    output_dir="./modelo-dpo",
    per_device_train_batch_size=2,
    learning_rate=5e-6,
    num_train_epochs=1,
    beta=0.1,  # que tanto se aleja el modelo ajustado del modelo original
)

trainer = DPOTrainer(
    model=modelo,  # el modelo ya pasado por SFT en el capitulo 6
    args=config,
    train_dataset=datos_preferencias,
    tokenizer=tokenizer,
)
trainer.train()
```

De donde salen los pares de preferencia

- Generar dos respuestas distintas con el mismo modelo (variando la temperatura) y que un humano elija cual prefiere.
- Usar respuestas reales buenas de tu equipo de atención vs. respuestas genéricas o incorrectas que quieras evitar.
- Usar un modelo grande como juez (la misma técnica del capítulo 9) para generar el ranking inicial, y validarlo después con revisión humana en una muestra.

> **TIP**
>
> No necesitas miles de pares de preferencia para notar mejoras: unos pocos cientos, bien elegidos alrededor de los casos donde tu modelo SFT falla más seguido, suelen alcanzar. DPO brilla justo para corregir esos "casi bien, pero no del todo" que el SFT solo no resuelve.

Entrenamiento distribuido

CAPÍTULO 11

Cuando una sola GPU no alcanza

Todo el código de los capítulos anteriores asume una sola GPU. Eso te sirve para modelos de 7-9B con LoRA/QLoRA. Pero si tu modelo base es más grande, tu dataset es enorme, o quieres hacer full fine-tuning en vez de LoRA, vas a necesitar repartir el trabajo entre varias GPUs. Esto se llama entrenamiento distribuido.

Las tres estrategias principales

Data Parallelism (DDP): la más simple. Cada GPU tiene una copia completa del modelo, y cada una procesa un lote distinto de datos en paralelo. Al final de cada paso, se promedian los gradientes entre todas las GPUs. Funciona muy bien cuando el modelo entero entra en una sola GPU pero quieres acelerar el entrenamiento repartiendo los datos.

```
# Lanzar entrenamiento DDP con accelerate en 4 GPUs
accelerate config  # respondes las preguntas: multi-GPU, 4 procesos, etc.
accelerate launch entrenar.py
```

Model Parallelism / Tensor Parallelism: cuando el modelo es tan grande que ni siquiera entra en una sola GPU, se parte el modelo mismo entre varias GPUs —cada una tiene una porción de las capas o de las matrices internas.

ZeRO / FSDP (Fully Sharded Data Parallel): la técnica que se volvió el estándar práctico para entrenar modelos grandes con recursos limitados. En vez de que cada GPU tenga una copia completa del modelo, el optimizador, y los gradientes, esa información se reparte ("shardea") entre todas las GPUs disponibles, liberando muchísima memoria.

Configurando DeepSpeed con ZeRO

```
# deepspeed_config.json
{
  "zero_optimization": {
    "stage": 2,
    "offload_optimizer": { "device": "cpu" }
  },
  "fp16": { "enabled": true },
  "train_batch_size": 16,
  "gradient_accumulation_steps": 4
}
```

```
# Lanzar con DeepSpeed integrado a TrainingArguments
args = TrainingArguments(
    output_dir="./modelo-grande",
    deepspeed="deepspeed_config.json",
    per_device_train_batch_size=2,
    num_train_epochs=3,
)
```

> **NOTA**
>
> El "stage" de ZeRO indica cuanto se reparte: stage 1 reparte solo el estado del optimizador, stage 2 también los gradientes, y stage 3 también los propios pesos del modelo. A más stage, más memoria ahorras pero más comunicación entre GPUs necesitas —stage 2 es un buen punto de partida para la mayoría de los casos.

Cuando de verdad necesitas esto

Para el 90% de los proyectos de fine-tuning de una empresa —con modelos de 7-9B y LoRA— una sola GPU alcanza y sobra, y este capítulo no aplica. Considera entrenamiento distribuido solo si: tu modelo base supera los 13-30B de parámetros, necesitas hacer full fine-tuning en vez de LoRA, o tu dataset es tan grande que el tiempo de entrenamiento en una sola GPU se vuelve impráctico (varios días o más).

Optimizar el modelo para servir

CAPÍTULO 12

Entrenar es solo la mitad del trabajo

Tener un modelo ajustado y evaluado no significa que este listo para atender usuarios reales a buen costo y velocidad. El modelo que acabas de entrenar, en su formato original, es pesado y lento de servir. Este capítulo cubre como optimizarlo, aplicable sobre todo al Camino A (modelos abiertos que tú sirves).

Cuantización para inferencia: GGUF y AWQ

- **GGUF:** formato pensado para correr modelos en CPU o hardware modesto, muy usado con el motor `llama.cpp` y herramientas como Ollama. Ideal para desarrollo local o despliegues sin GPU dedicada.
- **AWQ / GPTQ:** técnicas de cuantización pensadas para servir en GPU con buena velocidad, manteniendo la calidad muy cerca del modelo original en 16 bits.

```
# Convertir un modelo fine-tuneado a GGUF con llama.cpp
python convert_hf_to_gguf.py ./modelo-ajustado-final --outfile modelo.gguf --outtype q4_k_m

# Servirlo localmente con Ollama
ollama create mi-modelo -f Modelfile
ollama run mi-modelo
```

Motores de inferencia: como servir a muchos usuarios a la vez

Correr `model.generate()` directo con `transformers` (lo que usamos para entrenar) funciona para probar, pero es ineficiente para producción con varios usuarios simultáneos. Para eso existen motores de inferencia especializados:

Motor	Fortaleza	Cuando usarlo
vLLM	Alto throughput, batching continuo	Servir muchas requests simultaneas en GPU
llama.cpp / Ollama	Corre en CPU, formato GGUF	Desarrollo local o despliegues sin GPU
TGI (Text Generation Inference)	Integración directa con Hugging Face	Servir modelos de HF en producción

```
# Servir tu modelo LoRA con vLLM
pip install vllm

vllm serve ./modelo-ajustado-final \
    --enable-lora \
    --max-model-len 4096 \
    --port 8000

# Consultarlo como si fuera la API de OpenAI (vLLM expone el mismo formato)
curl http://localhost:8000/v1/chat/completions \
    -H "Content-Type: application/json" \
    -d '{"model": "modelo-ajustado-final", "messages": [{"role": "user", "content":
"Hola"}]}'
```

TIP

Un adaptador LoRA es liviano (unos pocos cientos de MB) comparado con el modelo base completo (varios GB). Muchos motores de inferencia, incluido vLLM, te dejan servir el modelo base una sola vez en memoria y cargar varios adaptadores LoRA distintos encima, lo que te permite tener varios "modelos especializados" corriendo con el costo de uno solo.

Producción, monitoreo y glosario

CAPÍTULO 13

Desplegando tu modelo en producción

Para un modelo abierto entrenado localmente y ya optimizado (capítulo 12), una forma simple de exponerlo como servicio es con FastAPI delante de vLLM o de un pipeline directo:

```
from fastapi import FastAPI
from pydantic import BaseModel
from transformers import pipeline

app = FastAPI()
pipe = pipeline("text-generation", model="./modelo-ajustado-final")

class Consulta(BaseModel):
    mensaje: str

@app.post("/chat")
def chat(consulta: Consulta):
    resultado = pipe(f"[INST] {consulta.mensaje} [/INST]")
    return {"respuesta": resultado[0]["generated_text"]}
```

Para un modelo fine-tuneado con un proveedor gestionado, simplemente apuntas al identificador que te devolvió el job de entrenamiento:

```
respuesta = client.chat.completions.create(
    model="ft:gpt-4.1-mini-2025-04-14:tu-empresa::ABC123",
    messages=[{"role": "user", "content": pregunta}]
)
```

Monitoreo y mejora continua

- Loggear todas las consultas y respuestas (respetando privacidad de datos, claro).
- Tener un botón de "esto no sirvió" o feedback en la interfaz.
- Revisar semanalmente los casos donde el modelo fallo.
- Agregar esos casos fallidos al dataset y re-entrenar cada cierto tiempo, o sumarlos como pares de preferencia para una ronda de DPO (capítulo 10).

Problemas comunes y como resolverlos

El loss no baja. Causas posibles: datos mal formateados, learning rate muy alto, o muy pocos ejemplos. Solución: revisa el formato línea por línea, baja el learning rate a 1e-4, y asegúrate de tener al menos 100 ejemplos de calidad.

El modelo repite siempre lo mismo. Causa: el modelo aprendió un patrón dominante que aparece demasiado seguido en tus datos. Solución: balancea mejor las categorías del dataset y reduce el número de épocas de entrenamiento.

El modelo alucina (inventa información). Causa: nunca aprendió los límites de lo que sabe, porque en el dataset nunca aparece un ejemplo donde la respuesta correcta sea "no tengo esa información". Solución: agrega explícitamente esos ejemplos.

Error de memoria (Out of Memory / OOM). Solución: reduce el batch_size, usa gradient_accumulation_steps más alto, o pasa a QLoRA en 4 bits si todavía no lo estabas usando, o considera ZeRO/FSDP (capítulo 11) si el modelo simplemente no entra.

Checklist final antes de pasar a producción

- ☐ Tienes al menos 200 ejemplos de entrenamiento de buena calidad?
- ☐ Separaste entrenamiento, validación y prueba (capítulo 5)?
- ☐ Evaluaste con métricas automáticas y LLM-as-judge, no solo con la vista (capítulo 9)?
- ☐ Comparaste el resultado contra el modelo base sin ajustar?
- ☐ Optimizaste el modelo para servir si el volumen de uso lo justifica (capítulo 12)?
- ☐ Tienes algún sistema de monitoreo de las respuestas en producción?
- ☐ Definiste que debe hacer el modelo cuando no sabe la respuesta?

Glosario rápido

Termino	Que significa
LLM	Large Language Model, un modelo de lenguaje entrenado con grandes cantidades de texto.
Fine-tuning	Reentrenar un modelo ya entrenado con datos propios, más chicos y específicos.
LoRA / QLoRA	Técnicas que permiten ajustar un modelo entrenando solo una fracción de sus parámetros, ahorrando memoria y tiempo.
RAG	Retrieval Augmented Generation: buscar documentos relevantes y pasárselos al modelo en el prompt, en vez de reentrenarlo.
DPO / RLHF	Técnicas de alineación de preferencias: ajustan el modelo para que prefiera unas respuestas sobre otras.
Perplexity	Métrica que mide que tan bien predice el modelo un texto real; más baja es mejor.
LLM-as-judge	Usar un modelo grande para evaluar y comparar respuestas de otro modelo.
ZeRO / FSDP	Técnicas de entrenamiento distribuido que reparten el modelo y sus estados entre varias GPUs.

Cuantización	Reducir la precisión numérica de los pesos de un modelo para que ocupe menos memoria y sea más rápido.
System prompt	Instrucciones fijas que definen el comportamiento del modelo antes de cada conversación.
Overfitting	Cuando el modelo memoriza los datos de entrenamiento en vez de aprender un patrón general.

. . .

Si llegaste hasta acá, ya tienes todo lo que necesitas para entrenar, evaluar en serio y poner en producción un modelo de IA ajustado a tu empresa —sin importar si el camino que elegiste fue entrenar un modelo abierto, ajustar uno gestionado por API, o guiar un modelo comercial con un buen prompt. No es magia, es trabajo bien hecho: datos buenos, código correcto, evaluación honesta y paciencia para iterar. Con eso alcanza y sobra.

Leandro Juan Sepúlveda Cuevas
Software Engineer · leandrosepulveda.com