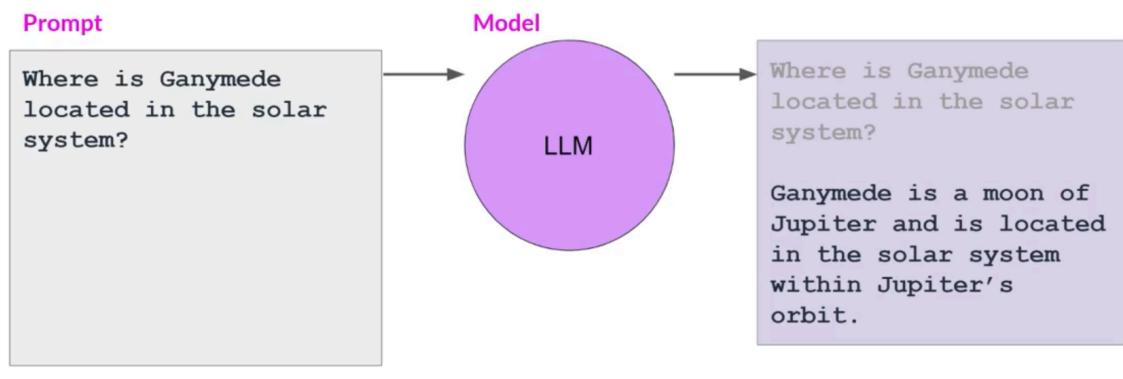


**semana 1**

## Clase 1 IA Generativa y LLM toma de nota:

La forma de interactuar con los modelos de lenguaje es muy diferente a la de otros paradigmas de aprendizaje automático y programación. En esos casos, se escribe código informático con una sintaxis formalizada para interactuar con las bibliotecas y las API. Por el contrario, los modelos de lenguaje de gran tamaño son capaces de tomar instrucciones en lenguaje natural o escritas por humanos y realizar tareas de la misma manera que lo haría un humano. El texto que se pasa a un LLM se conoce como mensaje. El espacio o la memoria disponible para la solicitud se denomina ventana de contexto y, por lo general, es lo suficientemente grande como para unos pocos miles de palabras, pero varía de un modelo a otro.



Context window

- typically a few 1000 words.

El acto de utilizar el modelo para generar texto se conoce como **inferencia**.

**La inferencia es el proceso de usar un modelo ya entrenado para generar predicciones (texto) basadas en datos nuevos (tu prompt).**

### Diferencias Clave: Training vs. Inferencia

Característica	Training (Entrenamiento)	Inference (Inferencia)
Objetivo	Aprender los patrones (Calcular pesos).	Usar lo aprendido (Generar texto).
Estado del Modelo	Los pesos cambian constantemente.	Los pesos están <b>congelados</b> (estáticos).
Costo Computacional	Masivo (Clusters de GPUs por meses).	Menor (Una GPU o CPU por milisegundos).
Operación	Forward Pass + Backward Pass (Cálculo de gradientes).	Solo Forward Pass.
Analogía	Estudiar para el parcial.	Rendir el parcial.

Exportar a Hojas de cálculo



## Clase 3 Generación de texto antes de los transformadores:

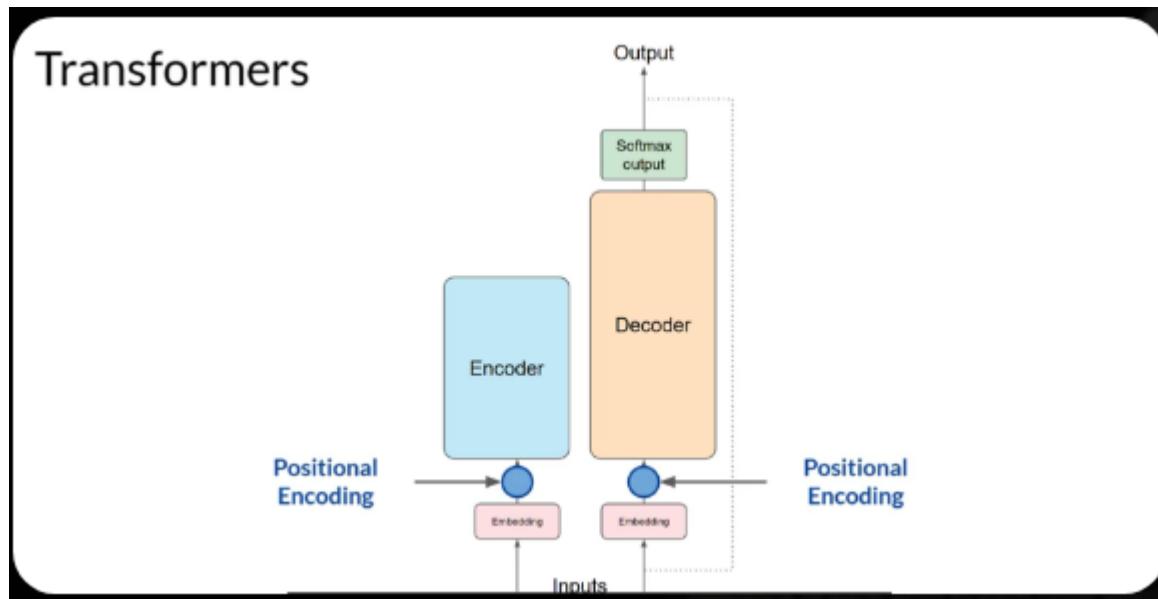
Antes de 2017, la IA generativa dependía de **RNNs (Redes Neuronales Recurrentes)**.

- **El problema:** Procesaban palabra por palabra (secuencial). Para entender una frase larga, necesitaban mucha memoria y cómputo, y aun así perdían el contexto (olvidaban el principio de la frase al llegar al final), fallando en la desambiguación.

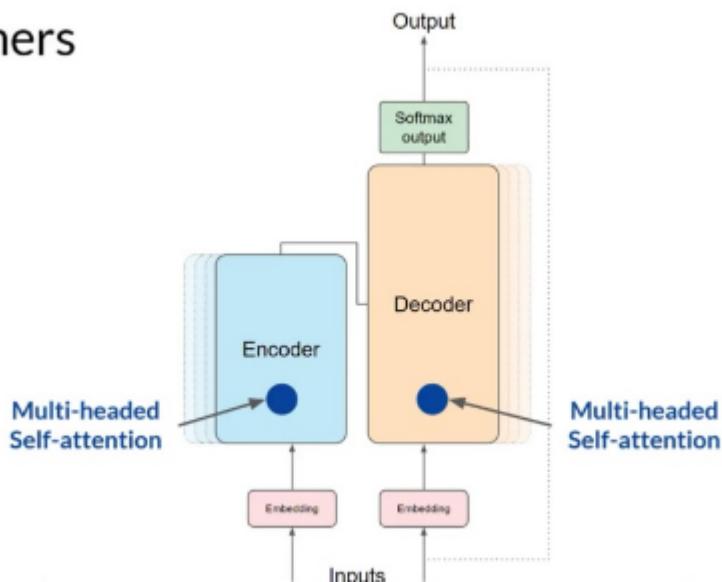
El punto de inflexión fue el paper "**Attention is All You Need**" (2017), que introdujo la arquitectura **Transformer**. **¿Por qué revolucionó todo?**

1. **Paralelismo:** A diferencia de las RNN, los Transformers procesan toda la oración a la vez (aprovechando la potencia multinúcleo de las GPUs).
2. **Mecanismo de Atención:** Permite al modelo enfocarse en la relación entre *todas* las palabras al mismo tiempo, entendiendo el contexto global y el significado exacto según la estructura de la frase.

#### Clase 4 Arquitectura de los transformadores:



# Transformers



## 1. El Problema de las RNN (El "Cuello de Botella")

Antes de 2017, usábamos Redes Neuronales Recurrentes (RNN).

- **Procesamiento Serial:** Leían la frase palabra por palabra ("El", luego "gato", luego "come").
- **El Fallo:** Si la frase era muy larga, al llegar al final, la red "olvidaba" el principio.
- **Limitación de Hardware:** No podías parallelizar el proceso. Tenías que esperar a que se procese la palabra 1 para procesar la 2.

**La Revolución Transformer:** El paper *"Attention is All You Need"* cambió el juego porque permite procesar **toda la oración en paralelo** al mismo tiempo.

---

## 2. Arquitectura de Alto Nivel: Encoder y Decoder

El Transformer se divide en dos grandes bloques que trabajan juntos pero tienen funciones distintas.

- **Encoder (Izquierda - Azul):** Su trabajo es **entender**. Toma el texto de entrada y crea una representación matemática rica del contexto.
- **Decoder (Derecha - Naranja):** Su trabajo es **generar**. Toma la representación del Encoder y predice la siguiente palabra (token) de la secuencia.

**Nota:** Muchos modelos actuales solo usan una parte. **BERT** es solo Encoder (bueno para clasificar). **GPT** (Generative Pre-trained Transformer) es solo Decoder (bueno para generar).

Aquí tenés el desglose técnico de las tres familias de arquitecturas que dominan la IA hoy:

---

## 1. Modelos Encoder-Only (Solo Encoder)

- **El Especialista en Comprensión (Auto-Encoder).**
  - **Arquitectura:** Solo usan la pila izquierda del diagrama (la parte azul).
  - **Cómo funcionan:** Toman una secuencia de entrada y generan una "representación vectorial" rica de esa secuencia. Originalmente, la entrada y salida tienen la misma longitud (vector por token).
  - **Para qué sirven:** No generan texto nuevo. Se usan para **Clasificación**.
    - Le agregás una capa simple al final y entrenás al modelo para que diga: "Este email es Spam (1) o No Spam (0)" o "Sentimiento Positivo/Negativo".
  - **Ejemplo Famoso:** **BERT** (Bidirectional Encoder Representations from Transformers).
- 

## 2. Modelos Encoder-Decoder (La Arquitectura Original)

- **El Traductor (Sequence-to-Sequence).**
  - **Arquitectura:** Usan ambas pilas (Azul y Naranja), tal como vimos en el ejemplo de traducción francés-inglés.
  - **Cómo funcionan:** El Encoder comprime el input; el Decoder genera el output.
  - **Característica Clave:** Son ideales cuando el largo de la entrada es muy diferente al largo de la salida (ej: resumir un texto largo en una frase corta).
  - **Para qué sirven:** Traducción, Resumen de textos.
  - **Ejemplos Famosos:**
    - **BART.**
    - **T5 (Text-to-Text Transfer Transformer):** Prestale atención a este porque **es el modelo que vas a usar en los laboratorios de este curso.**
- 

## 3. Modelos Decoder-Only (Solo Decoder)

- **El Generador (Autoregresivo).**
- **Arquitectura:** Solo usan la pila derecha (la parte naranja).

- **Cómo funcionan:** Ignoran la parte de "comprensión profunda bi-direccional" del Encoder puro y se centran exclusivamente en **predecir el siguiente token**.
- **La Evolución:** Al escalar estos modelos con trillones de parámetros, descubrieron que son capaces de generalizar y hacer casi cualquier tarea (incluso las de los otros dos grupos).
- **Para qué sirven:** Chatbots, generación de código, escritura creativa. Es lo que hoy llamamos "GenAI" estándar.
- **Ejemplos Famosos:** La familia **GPT** (Generative Pre-trained Transformer), **BLOOM**, **Jurassic** y **LLaMA** (Meta).

## MEJOR EXPLICADO:

### 1. Los Roles Técnicos

Imaginá que el **Encoder** y el **Decoder** son dos empleados distintos en una oficina:

- **El Encoder (El Lector/Analista):**
  - **Input:** Recibe el texto completo en el idioma original ("Hola mundo").
  - **Trabajo:** Lee todo junto (en paralelo), analiza las relaciones entre palabras (Self-Attention) y comprime todo ese significado en una **Matriz de Contexto** (una hoja de anotaciones matemáticas).
  - **Output:** Una matriz de números (vectores) que representa el *significado* de la frase. **El Encoder NO escribe texto, solo genera representaciones numéricas.**
- **El Decoder (El Escritor/Generador):**
  - **Input:** Recibe **dos** cosas:
    1. La **Matriz de Contexto** que le pasó el Encoder (las notas).
    2. Lo que él mismo escribió en el paso anterior (su propio historial).
  - **Trabajo:** Predice cuál es la **siguiente palabra** más probable para continuar la frase.

### 2. ¿Cómo se comunican? (El puente)

Si mirás el diagrama de arquitectura del Transformer, verás una flecha que sale del Encoder y se mete en el medio del Decoder. Eso se llama **Cross-Attention (Atención Cruzada)**.

Es el momento en que el Decoder dice: "*Okay, tengo que generar la siguiente palabra en inglés. Dejame mirar la Matriz del Encoder para ver en qué palabra original ('Hola' o 'mundo') debo enfocarme ahora*".

### 3. El Ejemplo Paso a Paso (El Bucle de Generación)

Aquí es donde entra lo del "Próximo Token". Los LLMs **no** escupen la frase entera de golpe. Escriben una palabra, la vuelven a leer, escriben la siguiente, la vuelven a leer. Es un bucle **for**.

**Misión:** Traducir "Hola mundo".

#### Paso 0: El Encoder trabaja (Solo una vez)

El Encoder procesa "Hola mundo" y genera la **Matriz de Contexto (\$M\$)**. Se la pasa al Decoder.

#### Paso 1: Decoder (Iteración 1)

- **Input del Decoder:** El token de inicio <START> + La Matriz \$M\$.
- **Pregunta del Decoder:** "Dado que estoy empezando (<START>) y mirando la matriz \$M\$ (que significa 'saludo al planeta'), ¿cuál es la palabra más probable?"
- **Softmax Output (Probabilidades):**
  - "Hello": 60%
  - "Hi": 39%
  - "Apple": 0.001%
- **Selección:** Elige "**Hello**".
- **Acción:** Imprime "Hello".

#### Paso 2: Decoder (Iteración 2)

- **Input del Decoder:** Ahora le pasamos **todo lo anterior**: <START> Hello + La Matriz \$M\$.
- **Cross-Attention:** El Decoder dice: "Ya tengo 'Hello', ahora necesito la traducción de la segunda parte. Voy a fijarme en la Matriz \$M\$ enfocándome en la parte que representa 'mundo'".
- **Softmax Output (Probabilidades):**
  - "world": 70%
  - "people": 15%
  - "kitty": 1%
- **Selección:** Elige "**world**".
- **Acción:** Imprime "world".

#### Paso 3: Decoder (Iteración 3)

- **Input del Decoder:** <START> Hello world + La Matriz \$M\$.
- **Pregunta:** "¿Me falta algo o ya terminé?"
- **Softmax Output:**
  - <END> (Token de fin): 99%
  - "again": 0.1%
- **Selección:** Elige <END>.
- **Acción:** El bucle se rompe. El programa termina.

---

## 4. Resumen Conceptual

1. **¿Por qué "Próximo Token"?** Porque el modelo es **Autoregresivo**. Su salida de hoy se convierte en su entrada de mañana. Es como el autocompletar del celular: te

sugiere *una* palabra, vos la tocás, y recién ahí te sugiere la siguiente basándose en la que tocaste.

2. **Encoder:** Entiende el pasado (el prompt completo).
  3. **Decoder:** Predice el futuro (token a token).
  4. **Comunicación:** Mediante la capa de **Cross-Attention**, donde el Decoder consulta la memoria del Encoder antes de decidir cada palabra.
- 

### 3. Paso a Paso: El Flujo de Datos

Vamos a seguir el viaje de una frase desde que entra hasta que sale.

#### A. Tokenización (De Palabras a Números)

La red neuronal no entiende "Libro". Solo entiende números.

1. **Tokenización:** Se rompe el texto en piezas (tokens). Puede ser una palabra entera o sílabas.
2. **Diccionario:** Cada token se convierte en un ID numérico único (ej: "Libro" -> 4598).
- **Regla de oro:** Debes usar el mismo tokenizador para entrenar que para generar.

#### B. Embeddings (El Espacio Vectorial)

Aquí la cosa se pone matemática. Cada ID numérico se convierte en un **Vector** (una lista de números, originalmente de 512 dimensiones).

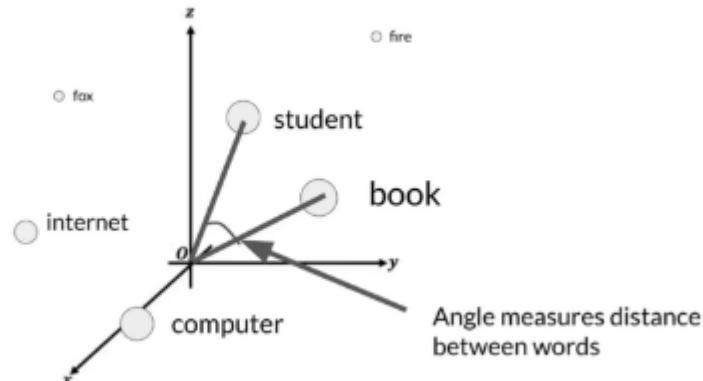
- **¿Qué es esto?** Es un espacio geométrico donde las palabras con significados similares están cerca físicamente.
- **El ángulo:** Como ves en la imagen, la distancia o el ángulo entre vectores determina su relación semántica. "Estudiante" estará más cerca de "Libro" que de "Fuego".

#### C. Positional Encoding (El "GPS" de la palabra)

Como el Transformer lee todo en paralelo (no de izquierda a derecha), no sabe el orden.

- **Problema:** Para el modelo, "*El oso come al hombre*" y "*El hombre come al oso*" se verían igual si solo sumáramos los significados.
- **Solución:** Se suma un vector de **Positional Encoding** al vector de Embedding. Esto le da una "marca de agua" a cada palabra indicando su posición (1<sup>a</sup>, 2<sup>a</sup>, 3<sup>a</sup>...) para preservar el orden gramatical.

## Transformers



### 4. El Corazón: Self-Attention (Auto-Atención)

Esta es la innovación clave. Permite que el modelo entienda cómo cada palabra se relaciona con **todas las demás palabras** de la frase al mismo tiempo.

- **Attention Map:** Imagina un mapa de calor.
  - En la frase "*El profesor enseñó a los alumnos con el libro*".
  - La palabra "**Libro**" estará fuertemente conectada (prestando atención) a "**Profesor**" y "**Alumno**".
- **Pesos (Weights):** Estos "grados de importancia" se aprenden durante el entrenamiento.

#### Multi-Headed Self-Attention (Atención Multi-Cabeza)

El modelo no hace esto una sola vez. Lo hace muchas veces en paralelo (entre 12 y 100 veces).

- **¿Por qué "Multi-Head"?** Cada "cabeza" aprende un aspecto distinto del lenguaje de forma independiente.
  - **Cabeza 1:** Puede enfocarse en quién realiza la acción (Sujeto).
  - **Cabeza 2:** Puede enfocarse en el tiempo verbal.
  - **Cabeza 3:** Puede enfocarse en si las palabras riman.
- **Resultado:** Obtienes una comprensión del texto mucho más profunda y rica que con una sola pasada.

## 5. La Salida: Feed-Forward y Softmax

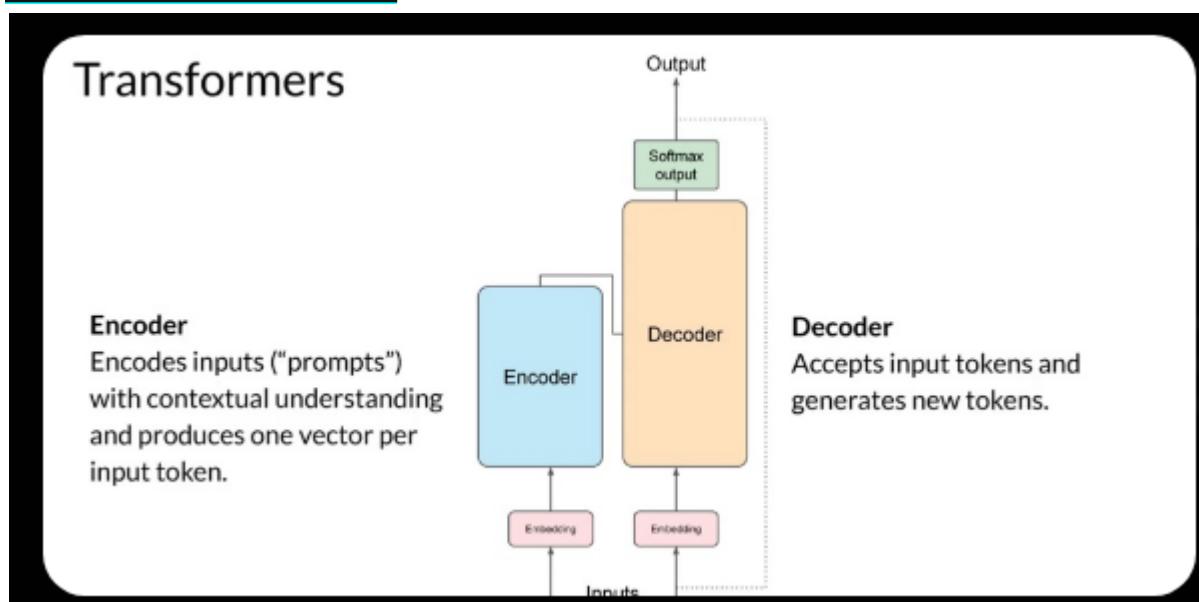
Una vez que la información pasó por las capas de atención (Encoder y Decoder):

1. **Feed-Forward Network:** La información procesada pasa por una red neuronal clásica para ser digerida.
2. **Output Logits:** La salida es un vector gigante de "logits" (puntuaciones crudas) para cada palabra posible en el diccionario.
3. **Softmax Layer:**
  - Convierte esos logits en **Probabilidades** (que sumen 100%).
  - Ejemplo: "Gato" (0.01%), "Perro" (0.02%), "**Libro**" (**99.5%**).
4. **Predictión:** El modelo selecciona el token con la probabilidad más alta como la siguiente palabra generada.

### Resumen para el Programador:

1. **Input:** IDs de Tokens.
2. **Process:**
  - Convertir a Vectores (Embedding) + Añadir Orden (Positional).
  - Calcular relaciones entre todas las palabras a la vez (Multi-Head Attention).
  - Procesar en paralelo (Encoder/Decoder).
3. **Output:** Distribución de probabilidad sobre el próximo token posible.

### FLUJO MÁS EXPLICADO:



## 1. El Objetivo: La Tarea Sequence-to-Sequence

- **Input (Entrada):** La frase en francés `J'aime l'apprentissage automatique.`
  - **Target (Objetivo):** Generar la frase en inglés `I love machine learning.`
- 

## 2. Fase 1: El Encoder (Compilación de Contexto)

El Encoder actúa como un compilador que procesa el código fuente (francés) y genera una representación intermedia.

1. **Tokenización:** Las palabras en francés se convierten en tokens numéricos usando el *mismo* tokenizador con el que se entrenó la red.
  2. **Procesamiento:**
    - Los tokens entran y pasan por la capa de **Embedding** (convirtiéndose en vectores).
    - Pasan por las capas de **Multi-Headed Attention** del Encoder. Aquí el modelo entiende que "*apprentissage*" y "*automatique*" juntos significan "*machine learning*".
  3. **Feed-Forward & Salida:** Los datos atraviesan una red feed-forward.
  4. **Resultado Final del Encoder:** No sale texto. Sale una "**representación profunda**" (**deep representation**) de la estructura y el significado de la frase original.
- 

## 3. Fase 2: El Bridge (La Inyección de Dependencia)

Aquí ocurre la magia de la comunicación.

- Esa "representación profunda" (la matriz de vectores del paso anterior) se **inserta en el medio del Decoder**.
  - Esto es lo que permite que los mecanismos de atención del Decoder se vean "influidos" por el contexto del francés mientras intentan escribir en inglés.
- 

## 4. Fase 3: El Bucle del Decoder (Runtime Loop)

Aquí es donde el código se ejecuta en un bucle `while` o recursivo para generar la salida.

### Iteración 0: El Disparador

- **Input del Decoder:** Se inserta un token especial llamado **Start of Sequence** (Inicio de Secuencia).

- **Analogía:** Es como llamar a `main()` para iniciar el programa.
- **Proceso:** El Decoder usa su auto-atención + la "representación profunda" del Encoder.
- **Salida (Softmax):** Predice el primer token: "I".

### Iteración 1: La Retroalimentación (Feedback Loop)

- **Input del Decoder:** Ahora entra `<Start Token>` + "I" (el token que acaba de predecir).
- **Acción:** El Decoder ve que ya tiene el sujeto "I" y, mirando el contexto del francés, busca el verbo.
- **Salida:** Predice el token "love".

### Iteración 2: Continuando la Secuencia

- **Input del Decoder:** `<Start Token>` + "I" + "love".
- **Acción:** Vuelve a pasar todo por el ciclo (Attention -> Feed Forward -> Softmax).
- **Salida:** Predice el token "machine".

### Iteración 3: Casi terminando

- **Input del Decoder:** `<Start Token>` + "I" + "love" + "machine".
  - **Salida:** Predice el token "learning".
- 

## 5. Fase 4: Terminación y Detokenización

El bucle continúa hasta que se cumple una **condición de parada (Stop Condition)**.

1. **End of Sequence (EOS):** Eventualmente, el modelo predice un token especial de "Fin de Secuencia".
    - **Analogía:** Es como el carácter `\0` al final de un string en C o un `return` en una función.
  2. **Detokenización:** La secuencia final de números `[I, love, machine, learning]` se pasa por el detokenizador para convertirse en texto legible para humanos: "I love machine learning".
- 

## Resumen Técnico para el Programador

1. **Encoder:** Procesa el input **una sola vez** en paralelo y genera la memoria del contexto.

2. **Decoder:** Es **autoregresivo**. Corre en un bucle donde `output_t` se convierte en `input_t+1`.
3. **Softmax Layer:** Es la capa final que decide "qué tan creativo" es el modelo. Aquí es donde ajustás la "Temperatura" (que verás más adelante en la semana) para elegir si querés el token más probable (determinista) o uno un poco menos probable (creativo).

### Autoatencion y FeedForward:

## 2. El Flujo Lógico: "Mirar" vs. "Pensar"

Imaginemos el token de la palabra "Banco":

1. **Fase 1: Atención (Gathering/Mirar):**
  - El token "Banco" mira a su alrededor, ve la palabra "Río" y actualiza su vector para significar "Banco de sentarse" y no "Banco de dinero".
  - *Resultado:* Un vector contextualizado.
2. **Fase 2: Feed-Forward (Processing/Pensar):**
  - Ahora que el vector ya tiene el contexto correcto, pasa por la Feed-Forward Network.
  - **¿Qué hace aquí?** "Digiere" esa información. Proyecta ese vector a dimensiones más altas, le aplica una función de activación no lineal (como ReLU o GELU) y lo vuelve a comprimir.
  - *Resultado:* Un vector con características más ricas y procesadas, listo para la siguiente capa del Transformer.

### Resumen

La capa de Atención sirve para **encaminar la información** (contexto), mientras que la capa Feed-Forward sirve para **analizar esa información** y extraer características profundas. Sin la Feed-Forward, el modelo sería solo un gran mezclador lineal de palabras sin capacidad de "razonamiento" complejo.

La arquitectura Transformer está formada por una pila de bloques.

Cada bloque contiene dos subcapas fundamentales: Self-Attention y Feed-Forward Network.

La atención permite contextualizar cada token en función de los demás, y la red feed-forward (FFN) procesa esa información para extraer características más profundas.

Juntas, y repetidas muchas veces, permiten el comportamiento inteligente del modelo.

Un **Transformer** está formado por una **pila (stack) de bloques**.

Cada **bloque** tiene esta estructura base:

◆ **Bloque Transformer =**

1. **(Self-)Attention**
2. **Feed-Forward Network**
3. **Residual connections**
4. **Layer Normalization**

👉 Eso vale para **encoder** y para **decoder**.

## ¿qué cambia entre Encoder y Decoder?

### ■ **Bloque del Encoder**

Cada bloque del **encoder** tiene **2 subcapas principales**:

1. **Self-Attention (bidireccional)**
2. **Feed-Forward Network**

📌 La atención es **bidireccional**:

- Cada token puede mirar **a todos los demás**
- Pasado y futuro

---

### ■ **Bloque del Decoder**

Cada bloque del **decoder** tiene **3 subcapas**:

1. **Masked Self-Attention (unidireccional, no podes ver el futuro)**
2. **Cross-Attention (opcional, solo si hay encoder)**

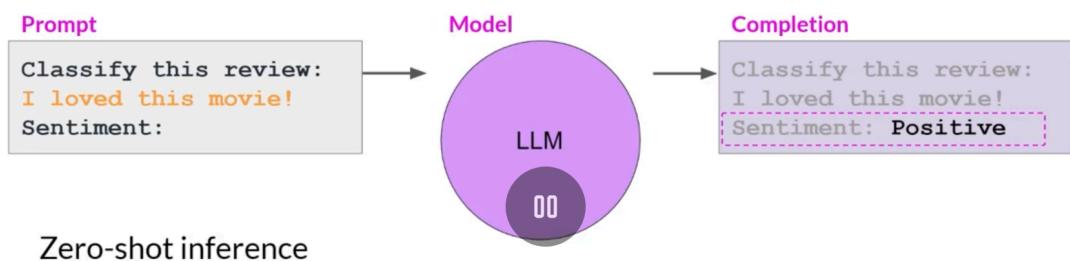
### 3. Feed-Forward Network

👉 Acá está la diferencia real.

---

#### Resumen Prompting y prompt engineering:

#### In-context learning (ICL) - zero shot inference

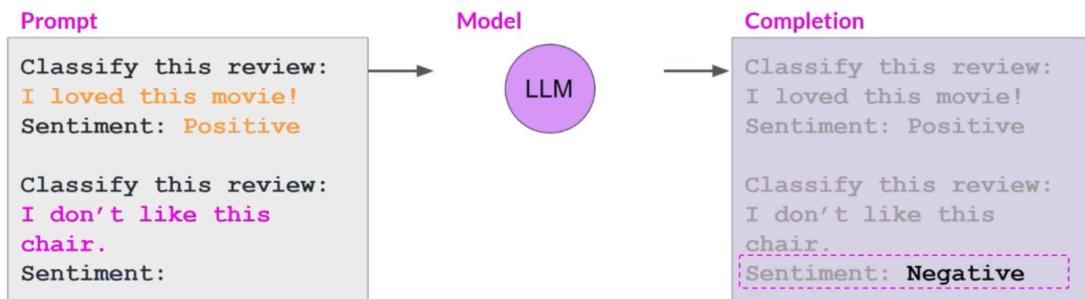


(Hacerlo así para modelos grandes puede servir, pero en modelos pequeños le pifia) **Los modelos grandes que abarcan muchos parámetros son capaces de deducir y completar muchas tareas para las que no estaban específicamente entrenados para ello.** (por eso les va bien zero shot inference) Por el otro lado, los modelos pequeños generalmente solo son buenos **en un número reducido de tareas**, generalmente similares a la tarea para la cual fueron entrenados.

**Se llama zero-shot inference porque no incluis ejemplos en el prompt.**

**La idea es proporcionar mensajes en el propio prompt para mayor eficiencia.**

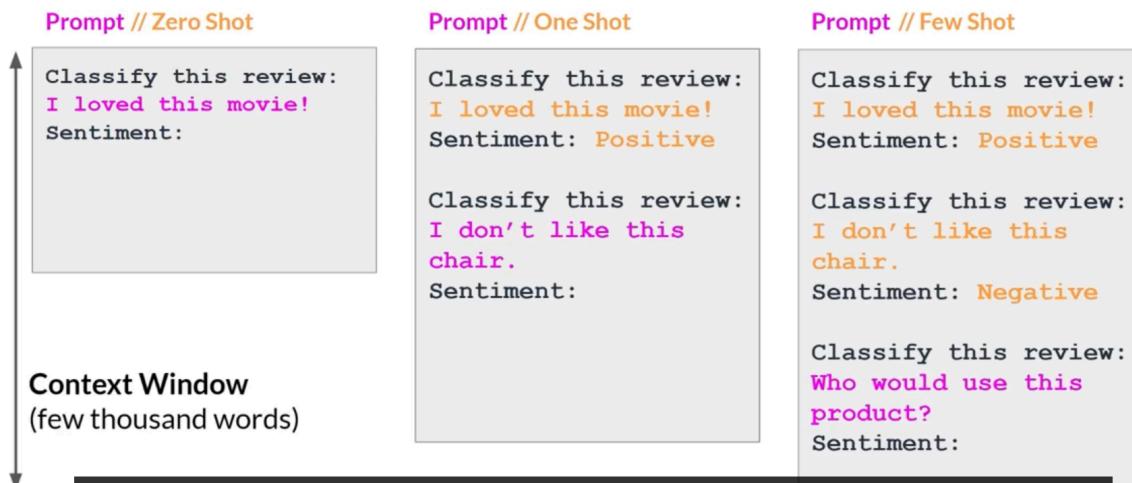
# In-context learning (ICL) / one shot inference



Se llama one-shot inference porque incluis UN ejemplo en el prompt.

Si tu modelo es aun mas chico, podes probar con **few-shot inference** la cual da más de un ejemplo en el prompt.

Recuerde la ventana de contexto o context windows, pq vos tenes un limite de la cantidad de contexto que podes transferirle al modelo.



Si con 5 o 6 ejemplos no te responde bien, deberia hacerse un ajuste del modelo, efectuando un entrenamiento adicional usando nuevos datos.

## Clase configuración generativa:

Se va a ver los parametros que podes tocar para influir en la forma en que el modelo toma la decision final sobre la próxima generación de palabras.

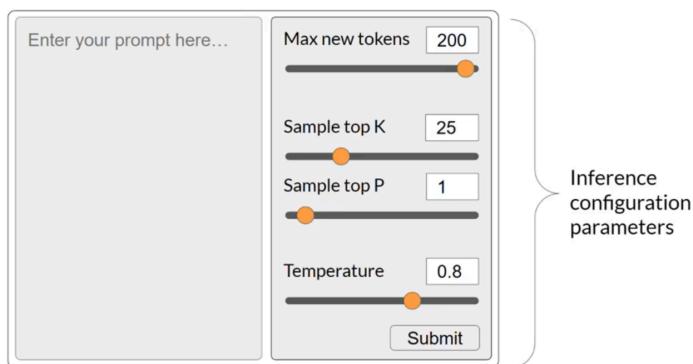
En conexión con los conceptos de **inferencia** vistos:

## Parámetros de Inferencia

Como el modelo ya no aprende, lo único que podés tocar durante la inferencia son los **parámetros de configuración** de la salida (que verás en el curso pronto):

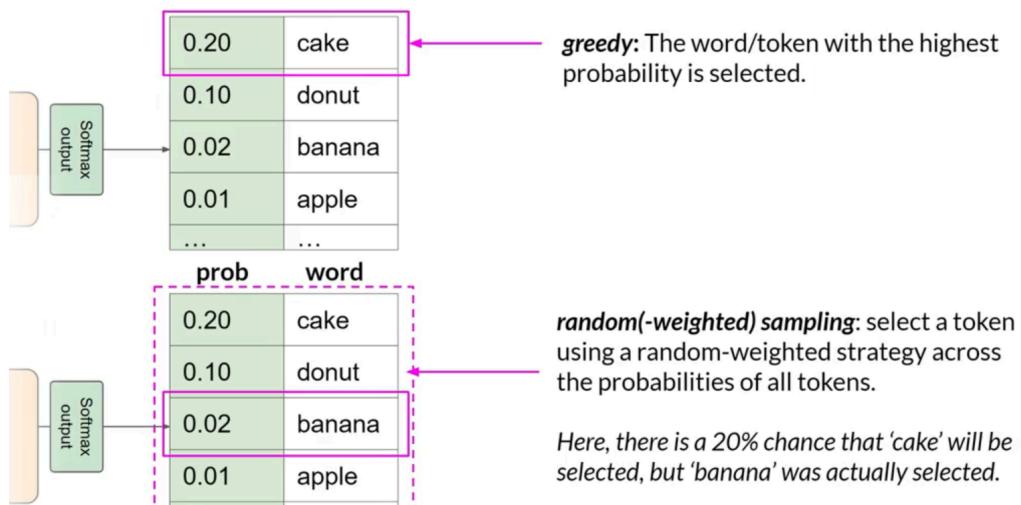
- **Max new tokens:** Cuánto querés que escriba como máximo.
- **Temperature:** ¿Querés que sea creativo o determinista?
- **Top-K / Top-P:** Formas de filtrar las probabilidades del Softmax.

## Generative configuration - inference parameters



Formas de seleccionar una palabra (**greedy** y **sampling**);

## Generative config - greedy vs. random sampling



**greedy** tiene problemas con la repetición de palabras, por ello esta **random sampling**, sin embargo el resultado de esta ultima puede ser muy creativo (fijate q aunque cake tiene 20% de ser seleccionado, terminó saliendo banana (2% de chance tenía))

### 1. Greedy Search (Búsqueda Codiciosa)

- **La Lógica:** `return max(probabilidades)`

- **Cómo funciona:** Siempre, sin excepción, elige la palabra con la probabilidad más alta.
  - **En el ejemplo:** Elegiría "Mundo" (60%).
  - **Resultado:** Es Determinista. Si corrés el modelo 100 veces con `temperature=0`, las 100 veces va a escribir el mismo texto.
  - **Problema:** Tiende a ser robótico, repetitivo y a veces entra en bucles infinitos ("Hola mundo mundo mundo...").
- 

## 2. Random Sampling (Muestreo Aleatorio Puro)

- **La Lógica:** Tirar una moneda o dado ponderado.
- **Cómo funciona:** "Mundo" tiene más chances de salir, pero si tenés mala suerte, podría salir "Tornillo" (0.1%).
- **Resultado:** Muy creativo, pero a veces incoherente. El modelo podría decir "Hola Tornillo".
- **Uso:** Rara vez se usa puro porque es demasiado caótico.

**TOP K Y TOP P:** Son técnicas de muestreo que nos permite limitar el muestreo aleatorio y aumentar las probabilidades de que el resultado sea sensato.

## 3. Top-K (El recorte fijo)

- **La Lógica:** "Quedate con los \$K\$ mejores y tirá el resto a la basura. Luego elegí al azar entre esos".
- **Ejemplo con K=3:**
  1. Selecciona: [Mundo, A todos, Gente].
  2. Elimina: [Manzana, Tornillo].
  3. Normaliza las probabilidades de los 3 finalistas para que sumen 100%.
  4. Elige uno al azar.
- **Ventaja:** Elimina las palabras absurdas (cola larga).
- **Desventaja:** Es rígido. Si \$K=3\$ pero hay 10 palabras que tienen sentido, te perdés 7. Si solo hay 1 palabra con sentido, forzás al modelo a considerar 2 malas.

---

## 4. Top-P (Nucleus Sampling) - El Estándar Moderno

- **La Lógica:** "Quedate con las palabras top que sumadas lleguen a una probabilidad \$P\\$ (ej: 0.90 o 90%)". Es un recorte dinámico. (OJO, TIENE Q SER  $\leq$  A P, no se puede pasar).
- **Ejemplo con P=0.90:**
  - Toma "Mundo" (0.60). ¿Llegamos a 0.90? No.
  - Suma "A todos" (0.30). Total acumulado = 0.90.
  - ¡Corte! El resto ("Gente", "Manzana", etc.) se descarta.
- **La Magia:**
  - Si el modelo está muy seguro ("El color del cielo es..."), la palabra "Azul" tendrá 99%, así que Top-P solo elegirá esa palabra (lista corta).

- Si el modelo duda ("El mejor sabor de helado es..."), las probabilidades estarán repartidas. Top-P incluirá "Chocolate", "Vainilla", "Frutilla" hasta sumar 90% (lista larga).
- **Resultado:** Es mucho más natural y se adapta al contexto.

### **Ejemplo:**

Supongamos que  $P = 0.90\$$  (90%) y el Softmax original te dio esto:

1. **Chocolate:** 0.60
2. **Vainilla:** 0.30
3. **Frutilla:** 0.09
4. **Menta:** 0.01

### **Paso 1: El Recorte (Filtering)**

El algoritmo suma en orden:  $0.60 + 0.30 = 0.90\$$ .

¡Listo! Llegamos a  $P\$$ .

- **Se quedan:** {Chocolate, Vainilla}.
- **Se eliminan:** {Frutilla, Menta}. (Aunque Frutilla tenía 9%, quedó afuera porque ya llenamos el cupo del 90%).

### **Paso 2: La Renormalización (El paso clave)**

Ahora tenemos un problema: Chocolate (0.60) + Vainilla (0.30) suman 0.90, no 1.0.

No podés tirar un dado de 100 caras si tus opciones suman 90.

Entonces, el motor recalcula los porcentajes para que ese grupito vuelva a sumar 100% entre ellos:

$$\text{NuevaProbabilidad} = \frac{\text{ProbabilidadOriginal}}{\text{SumaDelRecorte}}$$

- **Chocolate:**  $0.60 / 0.90 = 0.66$  (66.6%)
- **Vainilla:**  $0.30 / 0.90 = 0.33$  (33.3%)

### **Paso 3: El Sampling (El Dado)**

Ahora sí, el sistema "tira el dado" (genera un número random entre 0 y 1).

- Si sale entre 0.00 y 0.66 -> Escribe "**Chocolate**".
- Si sale entre 0.67 y 1.00 -> Escribe "**Vainilla**".

**Temperature:** Es un valor de escala que se aplica en la capa softmax final del modelo y que afecta a la forma de la distribución de probabilidad del siguiente token. Osea, cambiar la temperatura del modelo altera las predicciones que este realizará.

**\$T < 1\$ (Frío):** Divide por decimales -> Agranda los números -> Exagera la confianza -> **Determinista**.

**\$T > 1\$ (Caliente):** Divide por enteros -> Achica los números -> Iguala las probabilidades -> Random/Creativo.

### Ejemplo:

#### Caso A: Temperatura Baja ( $T = 0.5\$$ ) -> "Picar la Curva" (Sharpening)

Al dividir por un número menor a 1 (0.5), estás multiplicando por 2. Estás **exagerando** las diferencias.

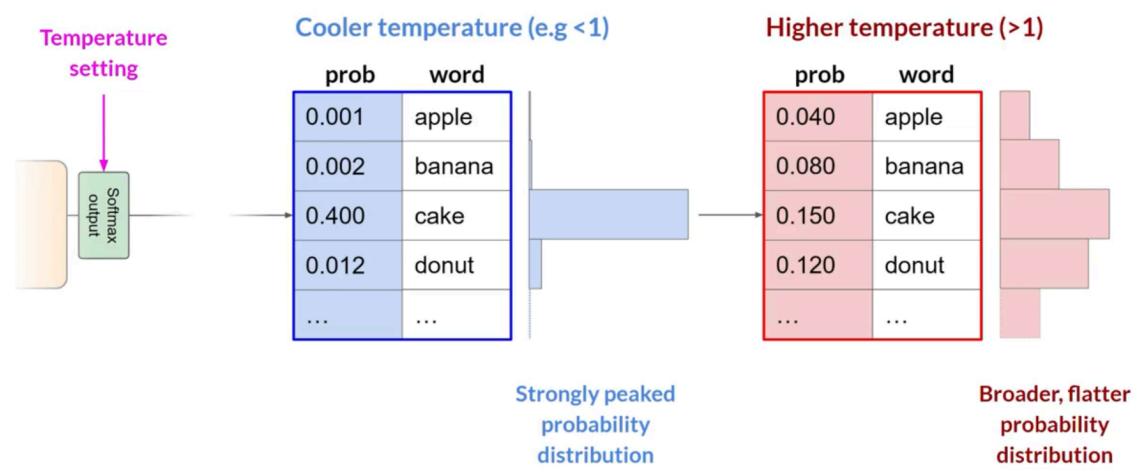
- Logits ajustados:  $10/0.5 = 20$ ,  $5/0.5 = 10$ ,  $2/0.5 = 4$ .
- La diferencia entre el primero (20) y el segundo (10) es enorme.
- Al pasar por  $e^x$ , esa diferencia se vuelve astronómica.
- **Resultado Softmax:**
  - "Claro": **99.99%**
  - "Quizás": 0.001%
- **Efecto:** El modelo se vuelve **híper-seguro**. La curva se hace un pico agudo. Solo existe una opción.

#### Caso B: Temperatura Alta ( $T = 2.0\$$ ) -> "Aplanar la Curva" (Flattening)

Al dividir por un número mayor a 1 (2.0), estás **achicando** los números. Las diferencias se reducen.

- Logits ajustados:  $10/2 = \mathbf{5}$ ,  $5/2 = \mathbf{2.5}$ ,  $2/2 = \mathbf{1}$ .
- Ahora la diferencia entre el primero y el segundo no es tanta.
- **Resultado Softmax:**
  - "Claro": **85%**
  - "Quizás": **12%**
  - "No": **3%**
- **Efecto:** "Quizás" y "No" reviven. Ahora tienen una chance matemática de ser elegidos si el dado cae ahí. La curva se aplana, se vuelve más uniforme.

## Generative config - temperature



Fijate que la roja te da mayor grado de aleatoriedad que la azul, osea mayor variabilidad pq la distribucion de probabilidad es mas plana y no tiene los picos de la azul.

**Si dejas la temperatura en 1, no va a haber influencia de esto, pq dividirías por 1 y no cambia la distribución de probabilidad.**

## Orden de Operaciones:

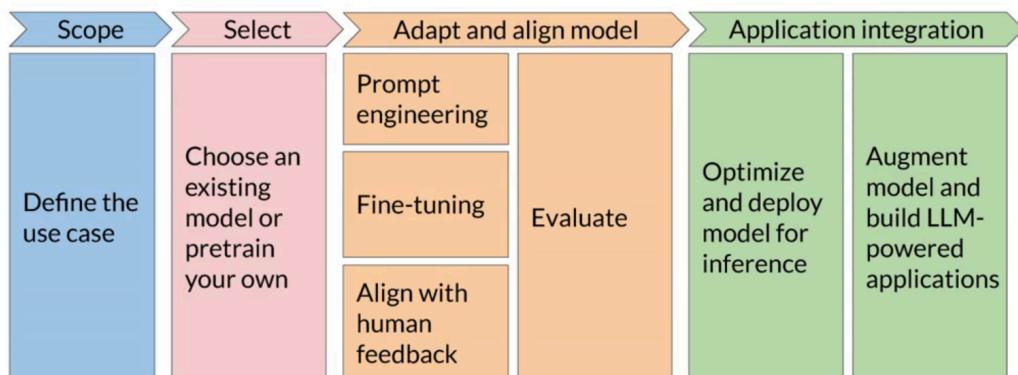
Esto es vital para tu código:

1. **Modelo:** Genera Logits.
2. **Temperature:** Divide los Logits ( $\text{logit} / T\$$ ). **<-- ACÁ ACTÚA**
3. **Softmax:** Convierte a Probabilidades (%).
4. **Top-P / Top-K:** Filtra las probabilidades bajas.
5. **Sampling:** Elige una palabra. (Greedy, Random)

---

## Ciclo de vida de un proyecto de IA generativa:

### Generative AI project lifecycle



## 1. Scope (Definir el Alcance)

- **Imagen:** Bloque azul "Define the use case".
- **Concepto:** Es el paso más crítico. Tenés que definir la función del LLM de forma estrecha y precisa.
- **Decisión Técnica:** ¿Necesitás un modelo capaz de hacer *todo* (generación de texto largo, chat, resumen) o algo muy específico como "reconocimiento de entidades nombradas"?
- **Por qué importa:** Ser específico te ahorra tiempo y, sobre todo, **costo de cómputo** (GPU). No matés un mosquito con un cañón.

## 2. Select (Seleccionar)

- **Imagen:** Bloque rosa "Choose an existing model or pretrain your own".
- **Concepto:** La decisión de "Build vs. Buy".
- **Camino A (Lo común):** Elegir un modelo base existente (Open Source como LLaMA o propietario como GPT).
- **Camino B (Lo raro):** Entrenar tu propio modelo desde cero ("from scratch"). Esto se reserva para casos muy necesarios debido al costo inmenso.

### 3. Adapt and Align (Adaptar y Alinear)

- **Imagen:** Bloque naranja central con sub-etapas.
- **Concepto:** Aquí es donde ocurre la ingeniería real para que el modelo funcione con TUS datos. Es un proceso iterativo.
  - **Prompt Engineering:** Es el primer intento. Usás *in-context learning* (darle ejemplos en el prompt) para ver si el modelo base ya funciona bien sin tocar sus pesos.
  - **Fine-tuning:** Si los prompts no alcanzan, pasás al entrenamiento supervisado (Semana 2 del curso). Re-entrenás el modelo con tus datos específicos para mejorar su rendimiento.
  - **Align with human feedback:** (Semana 3). Usás **RLHF** (Reinforcement Learning from Human Feedback) para asegurar que el modelo se comporte bien, sea seguro y siga las preferencias humanas.
  - **Evaluate:** La barra vertical "Evaluate" en la imagen indica que debés medir constantemente el rendimiento con métricas y benchmarks.

### 4. Application Integration (Integración)

- **Imagen:** Bloque verde final.
- **Concepto:** Llevarlo a Producción.
  - **Optimize and deploy:** Ajustar el modelo para inferencia (hacerlo rápido y eficiente en uso de memoria) para dar la mejor experiencia de usuario y reducir costos.
  - **Augment model:** Aquí entra lo que aprendiste de **LangChain**. Superar las limitaciones del LLM (como alucinaciones o falta de matemáticas) conectándolo a infraestructura externa (bases de datos vectoriales, APIs, etc.).

## Clase Preentrenamiento de grandes modelos lingüísticos:

Durante el **pre-entrenamiento**, el modelo:

- **✗ No aprende tareas**

- Aprende patrones estadísticos del lenguaje

Eso incluye:

- gramática
- co-ocurrencia de palabras
- estructuras semánticas
- relaciones implícitas (causa-efecto, jerarquías, etc.)

### Clave conceptual

El modelo no “entiende” → **aproxima distribuciones de probabilidad.**

Para un **Analista Programador**, lo más importante de este video es entender las **tres estrategias de entrenamiento** (funciones de pérdida) que definen para qué sirve cada modelo. No es magia, es estadística aplicada a diferentes objetivos.

Aquí tenés el resumen técnico estructurado:

## 1. El Proceso de Pre-Entrenamiento (Pre-training)

Es la fase donde el modelo aprende la estructura del lenguaje (gramática, hechos, razonamiento básico) usando **aprendizaje autosupervisado** sobre petabytes de texto no estructurado.

- **Data Pipeline:** Se hace scraping de internet, pero se requiere una limpieza masiva (eliminar sesgos, toxicidad, formateo).
- **Dato Curioso para Devs:** De todo lo que se descarga, **solo entre el 1% y el 3% de los tokens** pasan el filtro de calidad para entrar al entrenamiento.
- **Recursos:** Model Hubs como **Hugging Face** o PyTorch son los repositorios estándar. Clave leer las "**Model Cards**" (el **README.md** del modelo) para ver limitaciones y usos.

---

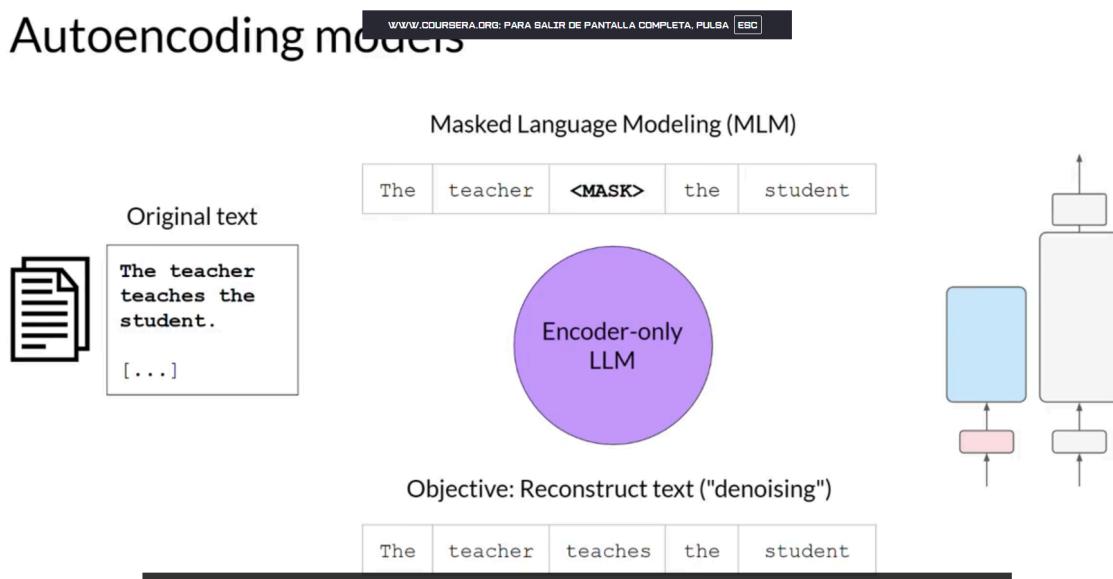
## 2. Las 3 Arquitecturas y sus Objetivos de Entrenamiento

No todos los Transformers se entranan igual. Dependiendo de qué parte de la arquitectura usen (Encoder, Decoder o ambos), cambian su "Training Objective".

### A. Autoencoding Models (Solo Encoder)

- **Arquitectura:** Usan solo la pila del Encoder (Bi-direccional).
- **Objetivo de Entrenamiento: Masked Language Modeling (MLM).**
  - **Lógica:** Se ocultan palabras al azar en una frase ("El [MASK] come pescado") y el modelo debe adivinar la palabra basándose en el contexto anterior Y posterior.
  - **Contexto:** **Bi-direccional** (mira a izquierda y derecha al mismo tiempo).
- **Caso de Uso:** Clasificación de oraciones (Sentiment Analysis), Clasificación de Tokens (Named Entity Recognition - NER). No sirven para generar texto.
- **Ejemplos: BERT, RoBERTa.**

## Autoencoding models



**El ciclo real es:**

1. El modelo predice probabilidades
2. Se calcula la loss
3. Se ajustan los pesos (backpropagation)
4. Se repite millones de veces

Objetivo final:

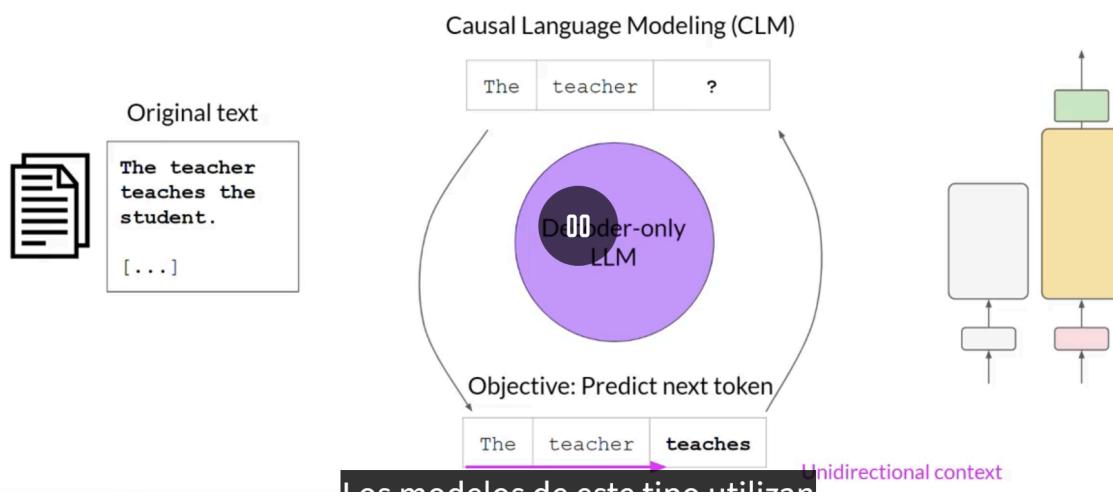
👉 Ajustar los pesos para que la probabilidad del token correcto sea cada vez más alta

**Entrenar un LLM es enseñarle a asignar alta probabilidad a la palabra correcta en el contexto correcto.**

## B. Autoregressive Models (Solo Decoder)

- **Arquitectura:** Usan solo la pila del Decoder (Unidireccional).
- **Objetivo de Entrenamiento: Causal Language Modeling (CLM).**
  - **Lógica:** Predecir el **siguiente token** basándose *exclusivamente* en los anteriores. El modelo no puede ver el futuro (se enmascara la derecha).
- **Contexto: Unidireccional.**
- **Caso de Uso:** Generación de texto (GenAI estándar).
- **Ejemplos:** GPT, BLOOM.

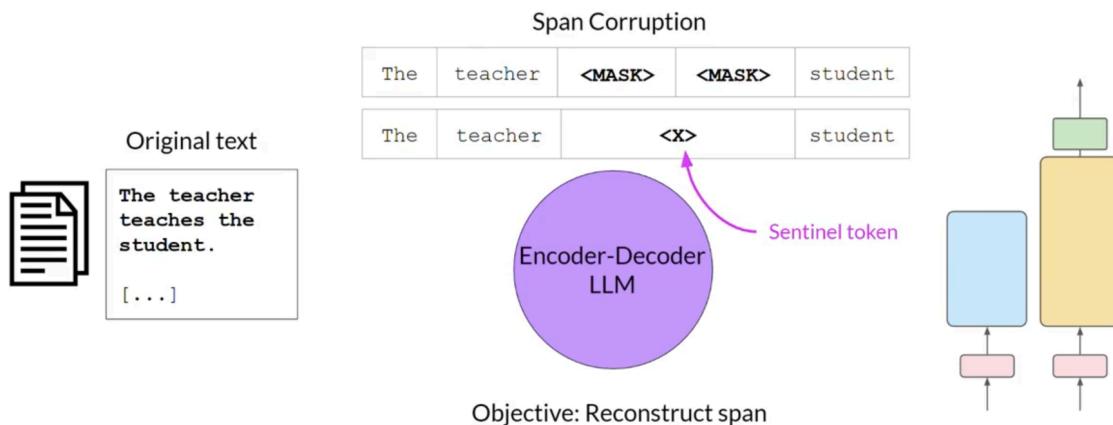
## Autoregressive models



## C. Sequence-to-Sequence Models (Encoder-Decoder)

- **Arquitectura:** Usan ambas partes.
- **Objetivo de Entrenamiento:** Varía, pero el estándar (como en T5) es **Span Corruption**.
  - **Lógica:** Se borran secuencias enteras de tokens en el input y se reemplazan por un **Sentinel Token** especial (un token "placeholder" que no es una palabra real, ej: <X>). El Decoder debe reconstruir esa secuencia perdida.

## Sequence-to-sequence models



- **Caso de Uso:** Traducción, Resumen de textos, Question-Answering (donde tenés un input denso y querés un output procesado).
- **Ejemplos:** T5 (el que usarás en el laboratorio), BART.

## Desglose Detallado

### 1. Encoder-Only = Autoencoding Models

- **Por qué se llaman así:** Porque actúan como un "Auto-Encoder" clásico de compresión. Toman una entrada ruidosa (con palabras borradas o "enmascaradas") y tratan de reconstruir la información original limpia.
- **La Clave:** Son **Bi-direccionales**. Como ven toda la frase a la vez, generan una representación densa y "comprimida" del significado.
- **Para qué sirven:** Entender, Clasificar, Extraer entidades.

### 2. Decoder-Only = Autoregressive Models

- **Por qué se llaman así:** El término viene de la estadística (Regresión). "Auto-regresivo" significa que la predicción actual depende de sus propios valores pasados.
- **La Clave:** Son **Uni-direccionales**. Solo miran hacia atrás (izquierda) para predecir el futuro (derecha). El modelo itera sobre la secuencia de entrada paso a paso para predecir el siguiente token.
- **Para qué sirven:** Generar texto, chatear, escribir código.

### 3. Encoder-Decoder = Sequence-to-Sequence Models

- **Por qué se llaman así:** Porque transforman una secuencia de tipo A (ej: Español) en una secuencia de tipo B (ej: Inglés).
- **La Clave:** El Encoder comprime la entrada y el Decoder genera la salida basándose en esa compresión. Usan técnicas como **Span Corruption** (borrar tramos enteros de texto y pedirle al modelo que rellene los huecos con centinelas).
- **Para qué sirven:** Traducción, Resumen (donde el input es distinto al output).

**Unidireccional (Decoder / GPT):** Es como leer un archivo **stream** línea por línea.

- Cuando estás en la palabra #5, solo podés ver de la #1 a la #4. La #6 es el futuro, "no existe" todavía.
- *Limitación:* Si la clave para entender la palabra #5 está en la palabra #10, te la perdiste.

**Bidireccional (Encoder / BERT):** Es como abrir un archivo en memoria completa (RAM) o una **foto**.

- El modelo ve **toda la frase al mismo tiempo**.
- Cuando procesa la palabra #5, puede consultar ("prestar atención") a la palabra #1 **Y** a la palabra #10 simultáneamente.

**Ejemplos:**

## 1. Autoencoding (Ej: BERT) - El "Adivinador"

- **Durante el Pre-entrenamiento (El Gimnasio):**
  - **Input:** "El [MASK] enseña al estudiante".
  - **Tarea:** El modelo suda calculando qué palabra falta. Al hacerlo millones de veces, aprende gramática y contexto.
  - **Objetivo:** Minimizar el error al adivinar "profesor".
- **Durante la Inferencia (Producción):**
  - **Tu Input:** Le mandás la frase limpia: "El profesor enseña bien".
  - **Tarea:** Le pedís una clasificación (ej: Sentiment Analysis).
  - **¿Hay máscara? NO.** Ya no ocultás nada. El modelo usa su conocimiento interno (los pesos que ganó en el gimnasio) para entender la frase completa y decirte "Sentimiento: Positivo".

## 2. Autoregressive (Ej: GPT/Llama) - El "Escritor"

- **Durante el Pre-entrenamiento:**
  - **Input:** "El profesor..."
  - **Tarea:** Predecir "enseña".
- **Durante la Inferencia:**
  - **Tu Input:** "El profesor..."
  - **Tarea:** Generar texto.
  - **Diferencia:** Acá el proceso **es idéntico**. GPT siempre está jugando a "predecir el siguiente token", tanto cuando entrena como cuando lo usás. Por eso se siente más natural.

## 3. Sequence-to-Sequence (Ej: T5) - El "Traductor"

- **Durante el Pre-entrenamiento:**
  - **Input:** "El profesor <X> estudiante" (Span Corruption).
  - **Tarea:** Rellenar el hueco <X>.
- **Durante la Inferencia:**
  - **Tu Input:** "Traducir: El profesor enseña al estudiante".
  - **Tarea:** Generar la traducción.
  - **¿Hay máscara? NO.** Le das el texto completo y esperás el output completo.

---

## 3. La Tendencia del Tamaño (Scaling)

El video cierra mencionando una especie de "Ley de Moore" para LLMs:

- A mayor tamaño (más parámetros) y más datos, mejor es la capacidad del modelo para generalizar y hacer tareas para las que no fue entrenado específicamente (**Zero-shot inference**).
- Sin embargo, entrenar estos modelos gigantes se está volviendo prohibitivamente caro y difícil, lo cual introduce los desafíos que veremos en el próximo video.

**SI USAS UN MODELO, COMO GPT, BERT, T5, YA VIENEN PRE-ENTRENADOS. VOS DSP USAS EL MODELO, HACES FINE-TUNING (OPCIONAL). Podes hacer el pre-training este pero es carisimo y al pedo.**

## Fine-Tuning de LLMs

Fine-tuning = agarrar un modelo que ya sabe lenguaje y ajustarlo para que haga mejor una tarea o dominio específico.

## Qué cambia y qué NO cambia en fine-tuning

### Lo que NO cambia

- Arquitectura
- Vocabulario
- Forma de entrenar (loss)

### Lo que SÍ cambia

- Pesos del modelo
- Preferencias de salida
- Conocimiento específico del dominio

---

### Clase Retos computacionales de la formación de LLM

este video es **clave** para entender *por qué entrenar LLMs es tan caro y cómo se mitiga el problema de memoria*.

---



## Problema central: la memoria al entrenar LLMs

Entrenar (o incluso cargar) **Large Language Models** suele fallar por **errores de memoria GPU (CUDA Out of Memory)**.

### ¿Por qué pasa?

- Los **LLMs tienen miles de millones de parámetros**
  - Cada parámetro ocupa memoria
  - Durante el entrenamiento, **no solo se guardan los pesos**, sino muchas cosas más
- 



## Cuánta memoria ocupa un modelo (intuición numérica)

### Representación estándar: FP32

- Un parámetro = **32 bits = 4 bytes**
- **1.000 millones de parámetros → 4 GB**  
⚠ *Solo para los pesos*

### Pero entrenar NO es solo pesos

Durante entrenamiento también se almacenan:

- Gradiéntes
- Estados del optimizador (ej. Adam → 2 estados)
- Activaciones intermedias
- Variables temporales



Todo esto suma **~20 bytes extra por parámetro**

### Regla práctica clave



**Entrenar requiere ~6× la memoria de los pesos**



Ejemplo:

- Modelo de **1B parámetros en FP32**
- Pesos: 4 GB
- Total entrenamiento: **~24 GB de VRAM**

➡ **Imposible en hardware de consumo**, difícil incluso en datacenters con una sola GPU.

## Additional GPU RAM needed to train 1B parameters

	Bytes per parameter
Model Parameters (Weights)	4 bytes per parameter
Adam optimizer (2 states)	+8 bytes per parameter
Gradients	+4 bytes per parameter
Activations and temp memory (variable size)	+8 bytes per parameter (high-end estimate)

Precisión = cuántos decimales reales guardás

Más precisión → más memoria

Cuantizar = redondear inteligentemente

Las redes toleran errores pequeños

BFLOAT16 es el mejor equilibrio actual

## ¿Qué significa “precisión” en un modelo?

👉 Precisión = qué tan exactamente se guarda un número en la computadora

Las redes neuronales **solo trabajan con números**:

- pesos
- gradientes

- activaciones

La pregunta es:

¿con cuántos “decimales reales” guardo esos números?

---



## Analogía CLAVE (la más importante)

Imaginá que querés anotar números en papel:



### Opción 1: Mucho detalle (alta precisión)

3.141592653589793



### Opción 2: Menos detalle (menor precisión)

3.14

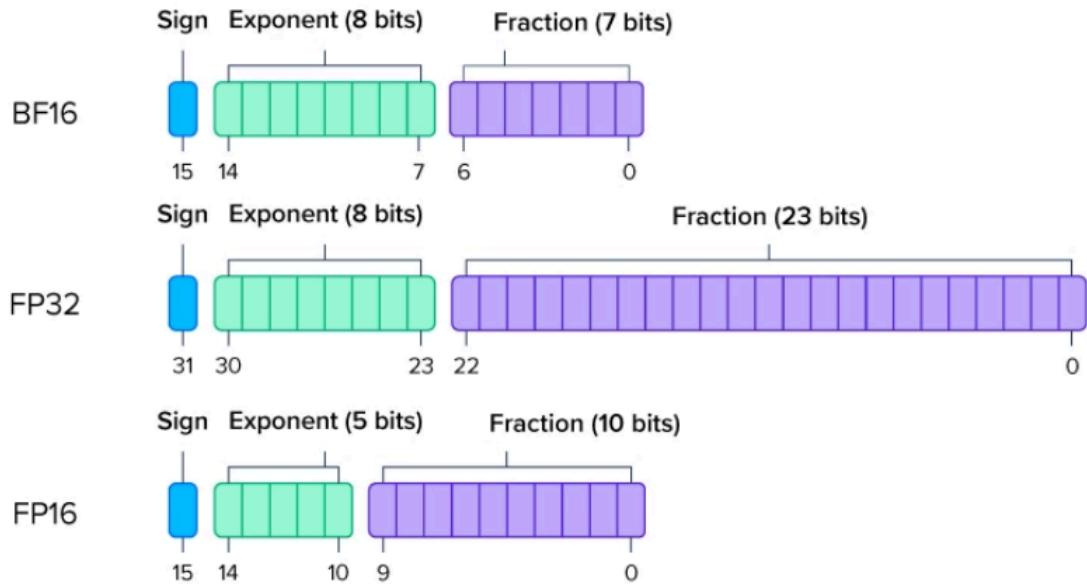


### Opción 3: Muy bruto (baja precisión)

3

✓ Todos representan “más o menos” el mismo número  
✗ pero no con la misma exactitud

👉 En computadoras pasa EXACTAMENTE lo mismo.



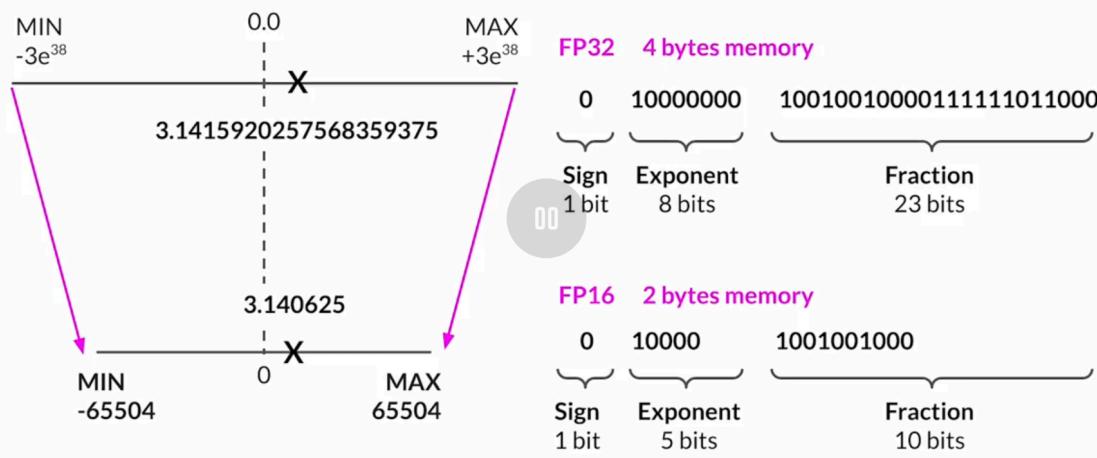
	Bits	Exponent	Fraction	Memory needed to store one value
FP32	32	8	23	4 bytes
FP16	16	5	10	2 bytes
<b>BFLOAT16</b>	<b>16</b>	<b>8</b>	<b>7</b>	<b>2 bytes</b>
INT8	8	-/-	7	1 byte

## ▼ ¿Qué hace la cuantización o quantization?

👉 Redondea inteligentemente los números. La cuantización reduce memoria sacrificando precisión numérica. (Su objetivo es reducir la memoria necesaria para almacenar y entrenar modelos mediante la reducción de la precisión de los pesos(parametros) de los modelos.)

## Quantization: FP16

Let's store Pi: 3.141592



(la precision seria la parte del fraction, la cual debido a que se achica en fp16, se pierde la precision del numero pi del ejemplo al volver a pasarlo de binario a numero.)

Fijate que almacenar un valor en fp32 es 4bytes y en el fp16 se reduce a la mitad.

## ¿Por qué esto NO rompe el modelo?

Porque las redes neuronales:

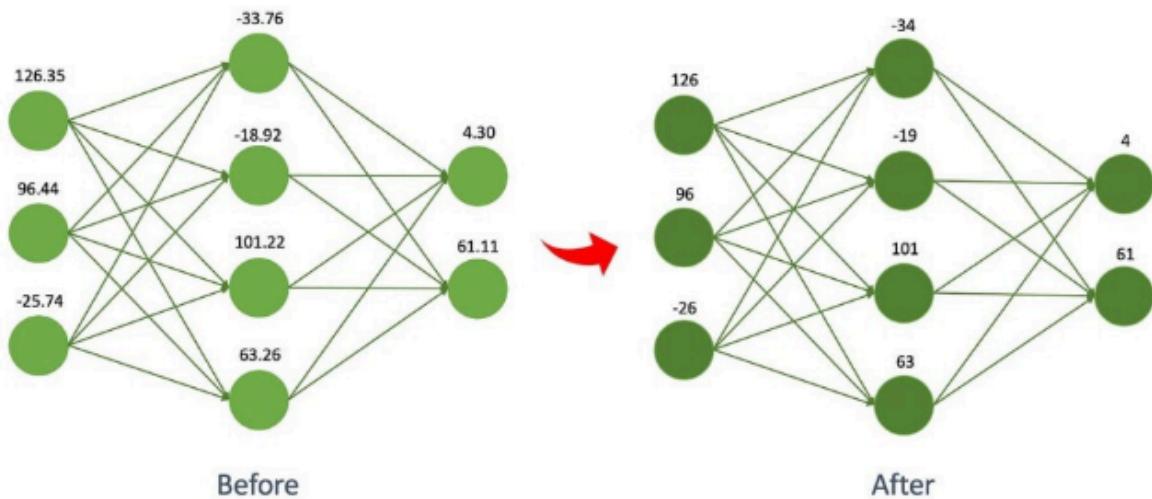
- No necesitan exactitud matemática perfecta
- Funcionan por **aproximaciones estadísticas**
- Son **robustas al ruido**

👉 Un peso de 0.23147 o 0.231 no cambia la decisión final

❤️ Pensalo así:

El modelo no hace cuentas como una calculadora  
hace **tendencias, proporciones y correlaciones**

## Master the Art of Quantization: A Practical Guide with TensorFlow and PyTorch



### FP32 (32 bits – Full Precision)

- 1 bit signo
- 8 bits exponente
- 23 bits mantisa
- Rango enorme:  $\pm 3 \times 10^{38}$
- **Muy preciso pero muy costoso en memoria**

---

### ② FP16 (16 bits – Half Precision)

- 1 bit signo
- 5 bits exponente
- 10 bits mantisa
- Rango:  $\pm 65.504$

- Mitad de memoria que FP32

## BFLOAT16 (BF16) MUY IMPORTANTE

- Desarrollado por **Google Brain**
- Usado en modelos grandes modernos

 Estructura:

- 1 bit signo
- **8 bits exponente (igual que FP32)**
- 7 bits mantisa

 ¿Por qué es tan bueno?

- Mantiene el **mismo rango dinámico que FP32**
- Reduce precisión (mantisa más corta)
- Mucho más **estable para entrenamiento**
- Más tolerante a gradientes grandes

La desventaja es que el BF16 no es adecuado para cálculos de enteros, pero estos son relativamente raros en el aprendizaje profundo.

 GPUs modernas que lo soportan:

- NVIDIA A100, H100, TPUs

 Modelos que lo usan:

- FLAN-T5
- Muchos LLMs actuales

 **Es el estándar moderno para entrenamiento**

## INT8 (8 bits – Enteros)

- 1 bit signo + 7 bits valor
- Rango: -128 a 127
- $\pi \approx 3$  😊

📌 Ventaja:

- 1 byte por parámetro (**75% menos memoria que FP32**)

📌 Desventaja:

- Muchísima pérdida de precisión
- Más común en **inferencia**, no en entrenamiento

## El límite real: modelos gigantes

Hoy existen modelos de:

- 50B parámetros
- 100B+ parámetros

📌 Escalado:

- 1B → 24 GB para entrenar
- 100B → **~2.400 GB (2.4 TB)** 😱

→ **Imposible con una sola GPU**

→ Se necesita:

- Entrenamiento distribuido
- Cientos de GPUs
- Costos millonarios

📌 Conclusión práctica:

No vas a pre-entrenar un LLM desde cero

---

### Clase Leyes de escala y modelos óptimos computacionales

- ✓ El pre-entrenamiento busca minimizar la pérdida
- ✓ El cómputo es el recurso limitante real
- ✓ Existen leyes de escalado entre loss, datos y parámetros
- ✓ Chinchilla demuestra que muchos LLMs estaban mal balanceados
- ✓ Dataset óptimo  $\approx 20\times$  parámetros
- ✓ Más parámetros sin datos suficientes  $\neq$  mejor modelo

## Tema central del video

Cómo decidir el tamaño óptimo de un modelo de lenguaje considerando **tres variables clave**:

1. **Tamaño del modelo** (cantidad de parámetros)
2. **Cantidad de datos de entrenamiento** (tokens)
3. **Presupuesto de cómputo** (GPU + tiempo)

El objetivo del **pre-entrenamiento** es **minimizar la pérdida (loss)** al predecir el próximo token.

---

## Objetivo del pre-entrenamiento

El modelo aprende patrones del lenguaje resolviendo una tarea simple pero poderosa:

**Predecir el próximo token** dado el contexto.

Esto se mide con una **función de pérdida**:

- Menor pérdida  $\Rightarrow$  mejor modelo
  - Mayor pérdida  $\Rightarrow$  peor rendimiento
-

## Presupuesto de cómputo: la restricción real

En teoría:

- Más datos  $\Rightarrow$  mejor modelo
- Más parámetros  $\Rightarrow$  mejor modelo

En la práctica:

**El cómputo es limitado** (hardware, tiempo y dinero)

Por eso se introduce una **unidad estándar de cómputo**.

---

### ¿Qué es un *PetaFLOP/s-día*?

Un PetaFLOP/s-día significa:

- Ejecutar  $10^{15}$  operaciones de punto flotante por segundo
- Durante **24 horas completas**

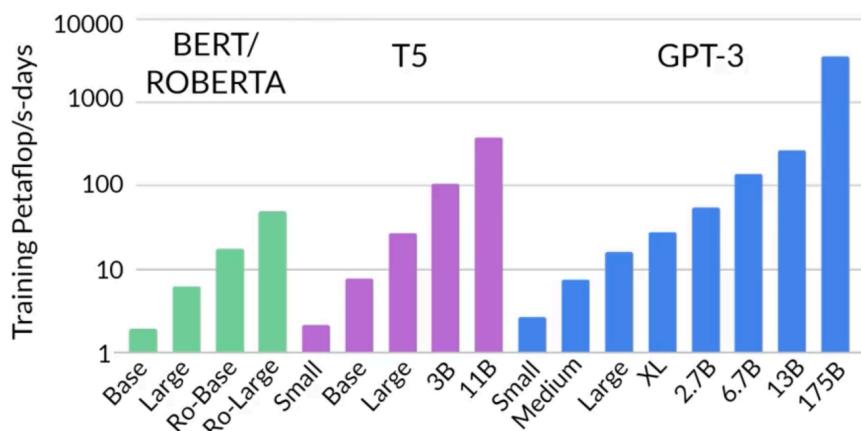
Osea, Sostener una velocidad de cómputo de  $10^{15}$  operaciones de punto flotante por segundo, durante 24 horas completas.

Ejemplos prácticos:

- $\approx 8$  GPU NVIDIA V100 funcionando 1 día completo
- $\approx 2$  GPU NVIDIA A100 (más potentes) durante 1 día

👉 Esta unidad permite **comparar entrenamientos distintos** de forma objetiva.

## Number of petaflop/s-days to pre-train various LLMs



El grafico muestra **Cuántos recursos computacionales hicieron falta para entrenar cada modelo.** Y eso se mide con una **unidad común: PetaFLOP/s-day**

### ¿Por qué esa unidad permite comparar modelos distintos?

Porque mide **trabajo computacional puro**, independientemente de:

- Arquitectura (encoder, decoder, encoder-decoder)
- Cantidad de parámetros
- Tipo de tarea
- Dataset específico

Es como comparar **consumo energético**:

No importa si usás una heladera, un aire acondicionado o una fábrica  
👉 comparás **kWh consumidos**

### Ejemplos concretos:

- **BERT Base** → muy poco cómputo
- **T5-3B** → ~100 PF-days
- **GPT-3 175B** → ~3.700 PF-days

El gráfico busca dejar **UNA idea central**:

## Entrenar modelos grandes cuesta órdenes de magnitud más cómputo

No es lineal:

- 10x más parámetros
- no cuesta 10x más cómputo  
 cuesta mucho más

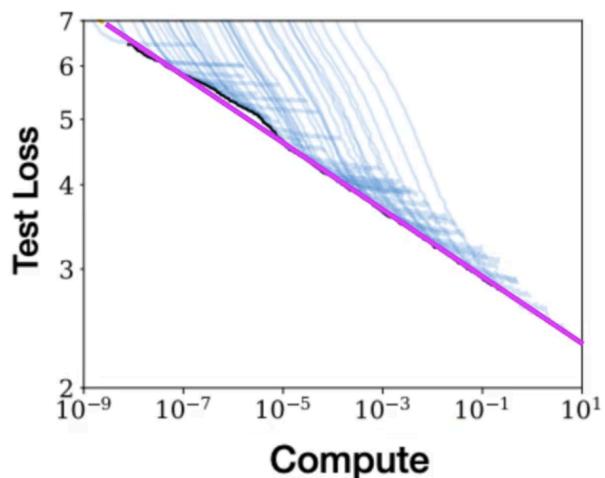
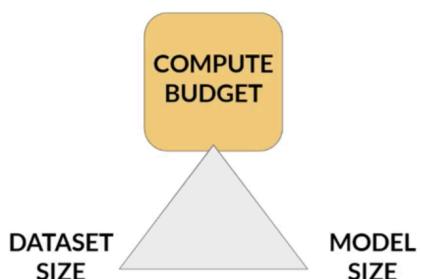
Por eso el eje es **logarítmico**.

. El rendimiento del modelo depende de tres variables :

Variable	Qué pasa si la aumentás
Cómputo	Mejora el rendimiento
Datos	Mejora el rendimiento
Parámetro s	Mejora el rendimiento

Pero no de cualquier manera.

## Compute budget vs. model performance



## Leyes de escalado (Scaling Laws)

Los investigadores observaron que:

- La **pérdida (loss)** sigue una **ley de potencias**
- En gráficos **log-log**, esto aparece como **líneas rectas**



## Interpretación clave

- Al principio, aumentar cómputo mejora mucho el rendimiento
- Luego, las mejoras son cada vez menores  
**(rendimientos decrecientes)**

Esto ocurre también con:

- Tamaño del dataset (tokens)
- Tamaño del modelo (parámetros)

**La pregunta clave:**

¿Cuál es el equilibrio óptimo entre parámetros y datos?

## Hallazgos clave de Chinchilla

Según las leyes de escalado estudiadas en el trabajo de Chinchilla, para un presupuesto de cómputo fijo existe un tamaño óptimo de dataset proporcional al número de parámetros. Si el modelo se entrena con menos datos que este óptimo, queda sub-entrenado y no alcanza su rendimiento máximo.



## Problema detectado

Muchos modelos gigantes (ej. GPT-3):

- **Sobreparametrizados**
- **Poco entrenados**



Tienen **demasiados parámetros para muy pocos datos**



## Regla fundamental de Chinchilla



**El dataset óptimo debe tener ~20× más tokens que parámetros**

Ejemplo:

- Modelo de **70B parámetros**
- Dataset óptimo: **1,4 billones de tokens**

👉 Fórmula mental:

$$\text{tokens} \approx 20 \times \text{parámetros}$$

Resultado:

Modelos más chicos + más datos  
pueden rendir igual o mejor que modelos gigantes

## 🏆 Resultado final del paper

- Chinchilla (como modelo de computo) **superá a GPT-3** en muchas tareas
- Usa **menos parámetros**
- Está **mejor entrenado**

👉 El rendimiento no depende solo del tamaño  
👉 **Depende del balance**

## Cambio de paradigma en LLMs

Antes:

“Más grande = mejor”

Ahora:

“Mejor balanceado = mejor”

---

### Clase Preentrenamiento para la adaptación al dominio

**Punto de partida: lo normal NO es entrenar desde cero**

La idea base del curso es:

**En la mayoría de los proyectos vas a usar un LLM existente**  
(GPT, LLaMA, Claude, etc.)

¿Por qué?

- Ahorrás **tiempo**
- Ahorrás **cómputo**
- Llegás más rápido a un **prototipo funcional**

👉 Esto cubre **la gran mayoría de los casos de uso.**

---

## 2) ¿Cuándo NO alcanza usar un LLM existente?

El video marca **una excepción clara y muy importante:**

**Cuando tu dominio usa vocabulario, estructuras y significados que NO aparecen en el lenguaje cotidiano.**

En ese caso, **fine-tuning o prompting no siempre es suficiente.**

---

## 3) Problema central: el preentrenamiento define el “lenguaje base”

Los LLMs:

- Aprenden **vocabulario**
- Aprenden **significados**
- Aprenden **uso contextual de palabras**

👉 Todo eso lo aprenden **durante el preentrenamiento**, no después.

Si algo **no apareció (o apareció muy poco)** en ese preentrenamiento:

- El modelo **no lo entiende bien**

- O lo usa **incorrectamente**
  - O lo confunde con otro significado común
- 

## 4 Ejemplo 1: Dominio legal

### Vocabulario especializado

Ejemplos:

- *mens rea*
- *res judicata*

👉 Son términos:

- Muy frecuentes en derecho
- Casi inexistentes en texto cotidiano

Resultado:

- Un LLM general **puede no saber usarlos**
  - O usarlos mal
- 

### Palabras comunes con significado distinto

Ejemplo:

- **consideration**

Lenguaje común:

- “Ser considerado / amable”

Lenguaje legal:

- Elemento esencial de un contrato que lo hace exigible

👉 El modelo puede **mezclar significados**

👉 Esto rompe aplicaciones legales

---

## 5 Ejemplo 2: Dominio médico

### Vocabulario raro

- Nombres de enfermedades
- Procedimientos
- Abreviaturas clínicas

Estos términos:

- Aparecen poco en web pública
  - Aparecen poco en libros generales
- 

### Lenguaje altamente idiosincrático

Ejemplo:

1 tab P0 QID PC & HS

Para un humano común:

- Texto sin sentido

Para un médico/farmacéutico:

- “Tomar una tableta por boca, cuatro veces al día, después de las comidas y antes de dormir”

👉 El modelo general:

- No fue entrenado en este “idioma”
  - No puede inferirlo bien solo con fine-tuning
- 

## 6 Conclusión fuerte (clave del video)

**Si el dominio es altamente especializado, el preentrenamiento desde cero produce mejores modelos que la adaptación posterior.**

Dominios típicos:

- Derecho
- Medicina
- Finanzas
- Ciencia
- Ingeniería

## ¿Qué es BloombergGPT?

- LLM entrenado **específicamente para finanzas**
  - Anunciado en 2023
  - Entrenado desde cero
- 

## 8 Estrategia de datos de BloombergGPT

Decisión clave del equipo:

Tipo de datos	Proporción
Datos financieros	51%

Datos públicos generales 49%

⌚ Objetivo:

- Excelente rendimiento en tareas financieras
- Sin perder capacidades generales del lenguaje

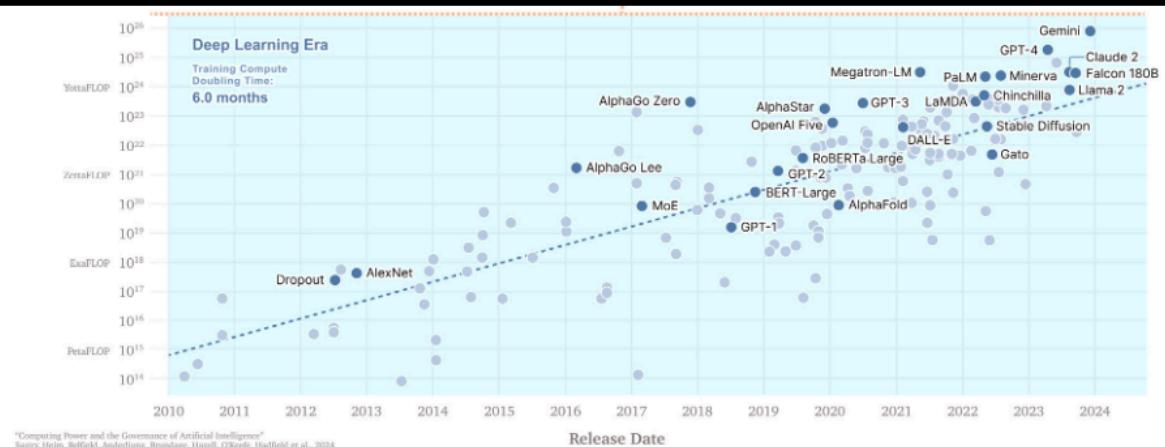
👉 No es un modelo “cerrado” al dominio, es híbrido.

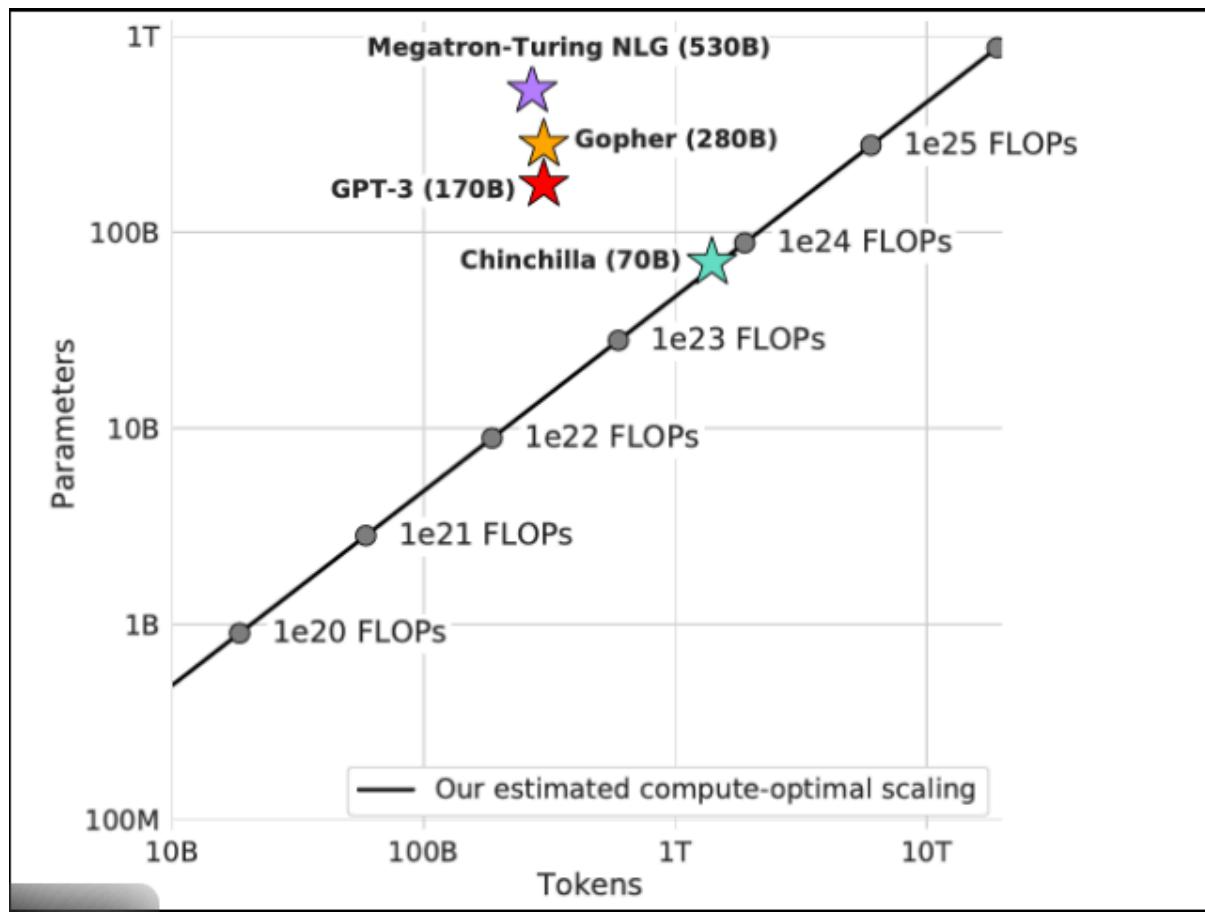
---

## 9 Uso de Chinchilla como guía (pero no dogma)

El equipo de Bloomberg:

- Usó las leyes de escalado de Chinchilla como referencia
  - Pero tuvo que hacer trade-offs reales
- ◆ Presupuesto de cómputo disponible
- ~1.3 millones de horas GPU
  - ≈ 230 millones de petaFLOPs





## 10 Análisis de los gráficos (muy importante)

### Gráfico izquierdo: tamaño del modelo

- Las líneas diagonales indican:
  - Tamaño óptimo del modelo
  - Para distintos presupuestos de cómputo

Resultado:

- BloombergGPT queda **muy cerca del óptimo de Chinchilla**
- Está apenas por encima
- ⇒ **Cantidad de parámetros casi óptima**



## Gráfico derecho (última imagen de arriba): tamaño del dataset

- Las líneas indican:
  - Cantidad óptima de tokens
  - Para cada presupuesto de cómputo



Resultado:

- BloombergGPT usó **569B tokens**
- Chinchilla recomendaría **más tokens**
- ⇒ **Dataset sub-óptimo**

## ¿Por qué se apartaron de Chinchilla?

Razón clave:

**No hay datos financieros infinitos**



El dominio impone límites reales:

- Privacidad
- Licencias
- Disponibilidad
- Calidad

Conclusión:

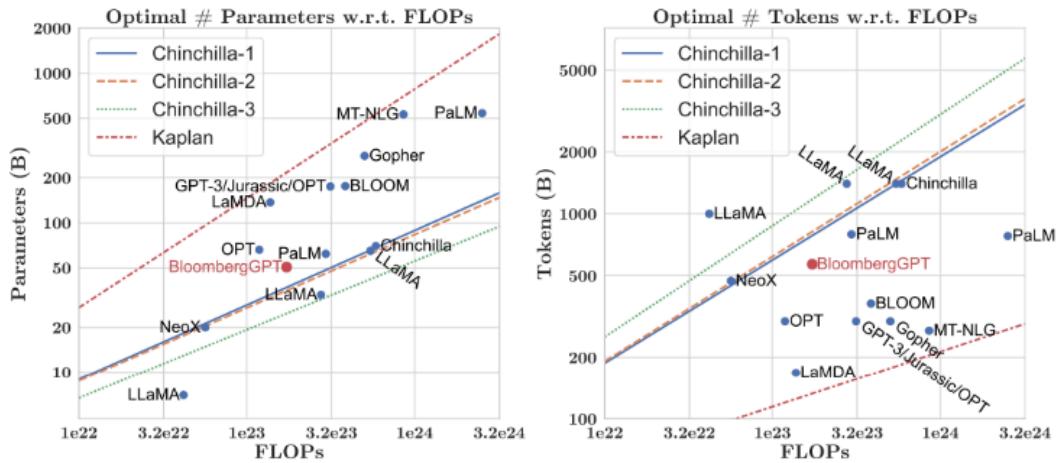
- El modelo queda **ligeramente sub-entrenado**
- Pero es el **mejor compromiso posible en la práctica**

## Lección fundamental del mundo real

**Las leyes de escalado son guías, no reglas absolutas.**

En producción real:

- El dominio limita los datos
- El presupuesto limita el cómputo
- Hay que elegir **el mejor trade-off**



Durante el entrenamiento de BloombergGPT, los autores utilizaron las leyes de escalado de Chinchilla para orientar el número de parámetros del modelo y el volumen de datos de entrenamiento, medido en tokens. Las recomendaciones de Chinchilla están representadas por las líneas Chinchilla-1, Chinchilla-2 y Chinchilla-3 en la imagen, y podemos ver que BloombergGPT se aproxima a ellas.

Aunque la configuración recomendada para el presupuesto de computación de entrenamiento disponible del equipo era de 50.000 millones de parámetros y 1,4 billones de tokens, adquirir 1,4 billones de tokens de datos de entrenamiento en el ámbito de las finanzas resultó todo un reto. En consecuencia, construyeron un conjunto de datos que contenía sólo 700.000 millones de tokens, menos que el valor óptimo de computación. Además, debido a una parada prematura, el proceso de entrenamiento terminó tras procesar 569.000 millones de tokens.

El proyecto BloombergGPT es una buena ilustración del preentrenamiento de un modelo para aumentar la especificidad del dominio, y de los retos que pueden obligar a hacer concesiones frente a las configuraciones de modelo y entrenamiento óptimas para el cálculo.

10. "Puede combinar el paralelismo de datos con el paralelismo de modelos para entrenar LLMs"

1 punto

¿Es esto cierto o falso?

- Verdadero
- Falso

### Justificación (clara y de curso)

En el entrenamiento de LLMs **sí se combinan** distintas formas de paralelismo:

- **Paralelismo de datos:**

El mismo modelo se replica en varias GPUs, cada una procesa distintos datos.

- **Paralelismo de modelo:**

El modelo se divide entre varias GPUs (porque no entra en una sola).

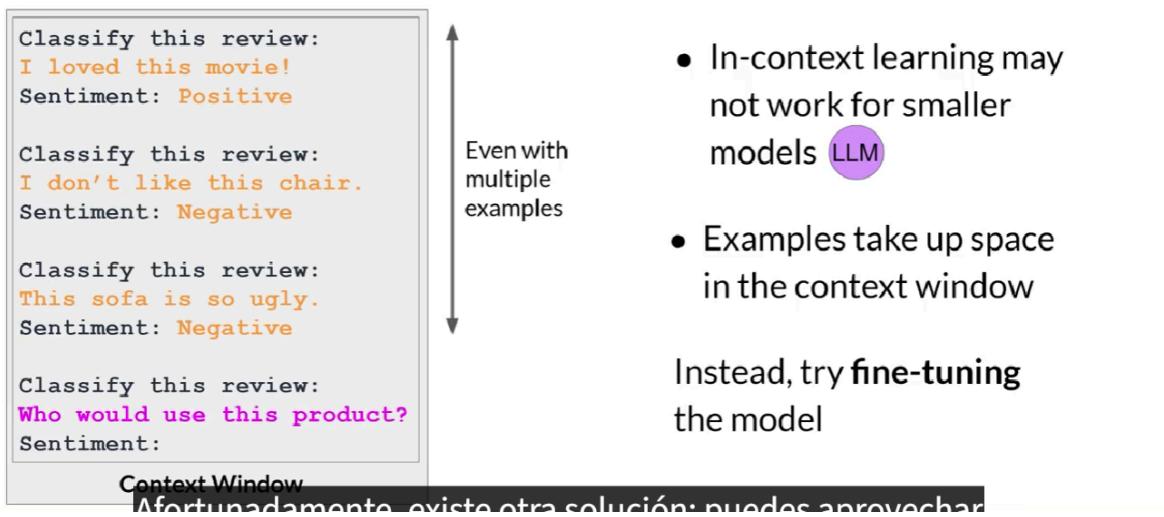
En la práctica moderna se usan **ambos al mismo tiempo** (incluso junto con *pipeline parallelism*), especialmente en modelos grandes.

👉 Por eso la afirmación es **verdadera**.

# Semana 2

## Clase Ajuste fino de las instrucciones:

# Limitations of in-context learning



el ajuste fino o **fine tuning** es un proceso de aprendizaje supervisado en el que se utiliza un conjunto de datos de ejemplos etiquetados para actualizar las ponderaciones del LLM. El proceso de ajuste fino amplía el entrenamiento del modelo para mejorar su capacidad de generar buenos resultados para una tarea específica.

**Fine-tuning** es:

Un proceso de **aprendizaje supervisado** en el que se usa un conjunto de **ejemplos etiquetados** para **actualizar los pesos de un LLM preentrenado**, mejorando su desempeño en una tarea específica.

Diferencia clave:

Etapa	Tipo de datos	Objetivo
Pre-training	Texto masivo no estructurado	Aprender el lenguaje
Fine-tuning	Ejemplos etiquetados	Aprender a hacer una tarea

## Qué se entrena en instruction fine-tuning

En **instruction fine-tuning**, el dataset está formado por:

- ◆ Pares **prompt → completion**

También llamados **prompt–completion pairs**.

Ejemplo:

Prompt:

Classify this review:

This sofa is so ugly.

Completion:

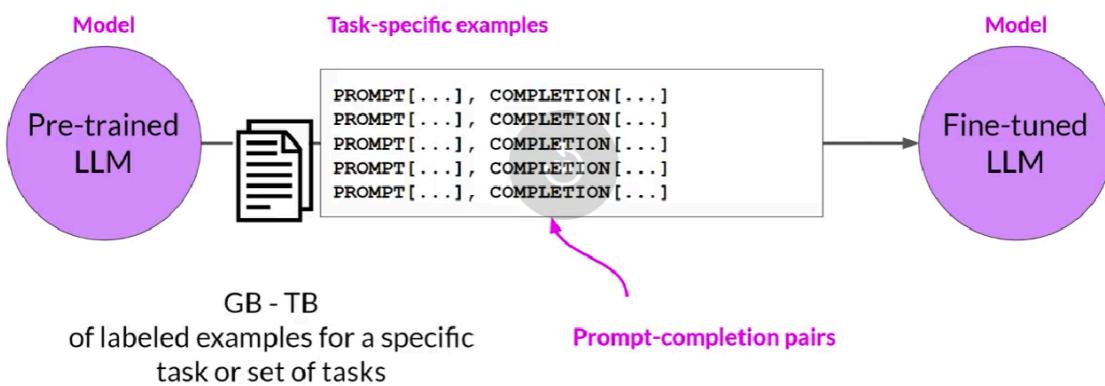
Sentiment: Negative

El modelo aprende:

- A reconocer la instrucción
- A responder en el formato esperado
- A generalizar a nuevos ejemplos

## LLM fine-tuning at a high level

LLM fine-tuning

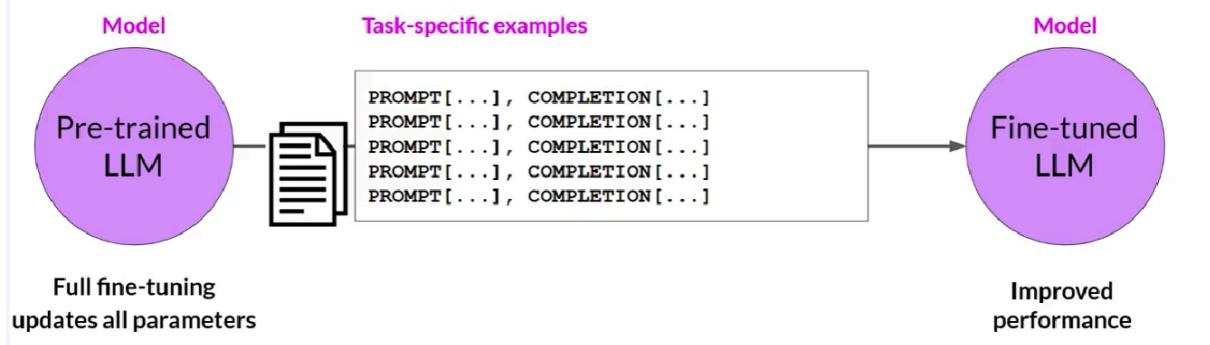


Estos ejemplos de finalización rápida permiten que el modelo aprenda a generar respuestas que sigan las instrucciones dadas.

El ajuste preciso de las instrucciones, en el que se actualizan todos los pesos del modelo, se conoce como ajuste completo. El proceso da como resultado una nueva versión del modelo con pesos actualizados. Es importante tener en cuenta que, al igual que antes del entrenamiento, el ajuste completo requiere suficiente memoria y presupuesto de cómputo para almacenar y procesar todos los gradientes, optimizadores y otros componentes que se actualizan durante el entrenamiento.

# Using prompts to fine-tune LLMs with instruction

## LLM fine-tuning



## Instruction fine-tuning (idea central)

### Concepto clave

Instruction fine-tuning entrena al modelo con **ejemplos que incluyen instrucciones explícitas**, como:

- “Summarize the following text”
- “Translate this sentence”
- “Classify this review”

👉 El modelo no solo aprende *la tarea*, sino **a seguir instrucciones**.

Por eso:

- Un mismo modelo puede mejorar en **varias tareas**
- Se vuelve un **instruct model**

## Full fine-tuning (ajuste completo)

Cuando:

- **Se actualizan todos los pesos del modelo**
- Se recalculan gradientes, optimizadores, etc.

Se llama **full fine-tuning**.

## Implicaciones

- Alto costo de **memoria**
- Alto costo de **cómputo**
- Mismo tipo de problemas que en pre-training (aunque a menor escala)

👉 Por eso se aplican:

- Técnicas de optimización de memoria
- Paralelismo
- Batch training



## El paso a paso:

### ① Preparar el dataset

- Pares **prompt → completion**
- Instrucción + ejemplo

Definir correctamente el dataset de instrucciones que vas a aplicarle al modelo pre-entrenado.

Muchos datasets **no vienen en formato de instrucción**

Ejemplo: Amazon Reviews

## Solución

Usar **prompt templates** (plantillas)

Ejemplo (clasificación):

```
Given the following review:  
{{review_body}}
```

predict the associated rating from 1 to 5.

Ejemplo (resumen):

Give a short sentence describing the following product review:  
{{review\_body}}

👉 Así transformás datasets tradicionales en **instruction datasets**.

2. Luego tenes que dividir el conjunto de datos en divisiones de entrenamiento, validación y prueba.

## LLM fine-tuning process

### LLM fine-tuning

Prepared instruction dataset



Prompt:

Classify this review:  
I loved this DVD!

Sentiment:

Model

Pre-trained  
LLM

### LLM completion:

Classify this review:  
I loved this DVD!

Sentiment: Neutral

### Label:

Classify this review:  
I loved this DVD!

Sentiment: Positive

## 3 Entrenamiento

Para cada batch:

1. Se pasa el **prompt** al LLM
2. El modelo genera una **completion**
3. Se compara con la **etiqueta real**
4. Se calcula la **función de pérdida (cross-entropy)**

El output del LLM es una **distribución de probabilidad sobre tokens**

5. Se hace **backpropagation**

6. Se actualizan los pesos

Esto se repite:

- En muchos batches
- Durante varias épocas

## 4 Evaluación

- **Validation set** → ajustar hiperparámetros

La validación se utiliza para detectar sobreajuste y asegurar que el modelo no memorice los ejemplos de entrenamiento, sino que aprenda patrones generales que le permitan generalizar a nuevos datos e instrucciones.

Un modelo **memoriza** cuando:

- Aprende muy bien los ejemplos de entrenamiento
- Pero **no generaliza** a datos nuevos

En términos prácticos:

- Training loss ↓
- Validation loss ↑

Un LLM **aprende realmente** cuando:

- Aprende la **relación instrucción → respuesta**
- Generaliza a **nuevos prompts no vistos**
- Mantiene buen desempeño fuera del dataset de entrenamiento

Eso se mide **exclusivamente** con datos que **el modelo no vio durante el entrenamiento**.

- **Test set** → medir desempeño final

Validation → durante el entrenamiento

Test → una sola vez, al final

Resultado:

👉 Un **modelo instruct fine-tuned**, mejor que el base para esa tarea.

## 8. Métrica clave: Cross-Entropy

La pérdida se calcula entre:

- Distribución de tokens predicha por el LLM
- Distribución objetivo (la etiqueta)

Esto es exactamente el mismo principio que:

- Clasificación clásica
- Language modeling

Nada “mágico”, es **ML estándar aplicado a tokens**.

## 9. Qué mejora realmente el fine-tuning

- Menos dependencia de ejemplos en el prompt
- Mejor desempeño en modelos chicos
- Respuestas más consistentes
- Mejor seguimiento de instrucciones

Siguiente clase fine tuning en una sola tarea:

Los **LLMs pre-entrenados** son modelos **multitarea** por naturaleza:

- Pueden resumir

- Traducir
- Clasificar
- Extraer entidades
- Generar texto

Pero **tu aplicación** muchas veces:

- Necesita **una sola tarea**
- Con **alta precisión**
- De forma **confiable**

👉 En ese caso, **fine-tuning específico por tarea** es una muy buena opción.

## Definición

Es el proceso de ajustar un LLM pre-entrenado usando:

- Un dataset **enfocado en una única tarea**
- Ejemplos etiquetados del tipo *prompt → completion*

Sabemos que fine-tuning puede incrementar la performance de un modelo en una tarea específica. Entre 500 a 1000 ejemplos es lo que se necesita para aplicar fine tuning para una tarea específica a un modelo pre entrenado. Sin embargo puede darse en full fine tuning lo que se conoce como un olvido catastrófico, ya que al ajustar todos los pesos del modelo para que en tu tarea específica rinda mejor, puede degradar el rendimiento de otras.

## Problema: olvido catastrófico (catastrophic forgetting)

### Definición

El **olvido catastrófico** ocurre cuando:

Al hacer full fine-tuning sobre una tarea, el modelo pierde habilidades que tenía previamente en otras tareas.

## Por qué ocurre

- Full fine-tuning **modifica todos los pesos**
- Los nuevos gradientes “pisan” representaciones útiles previas
- El modelo se **hiperespecializa**

## Ejemplo concreto

Antes del fine-tuning:

Sentence: Charlie is a cat.  
Task: Named Entity Recognition  
Output: Charlie → Name

Después de fine-tuning solo en sentiment analysis:

Sentence: Charlie is a cat.  
Output: Sentiment: Positive ✗

👉 El modelo:

- Confunde la tarea
- Aplica el patrón aprendido recientemente
- “Olvida” cómo hacer NER

Primero debemos analizar si el olvido catastrofico afecta a tu caso de uso. Si no te afecta porque solo querias centralizar el LLM a una tarea especifica, joya. Si te afecta, entonces se puede aplicar fine tune en multiples tareas al mismo tiempo (seria un ajuste multitarea) Esto puede requerir entre 50 y 100.000 ejemplos. La otra es aplicar lo que se llama PEFT (ajuste preciso con eficiencia de parametros)

## ¿Siempre es un problema?

No.

Depende totalmente de tu **caso de uso**.

**Caso 1: NO es un problema**

- Tu aplicación hace **una sola tarea**
- No necesitas multitarea
- Ejemplo:
  - Clasificador de tickets
  - Analizador de sentimiento
  - Resumidor de documentos legales

👉 El olvido catastrófico **no importa**.

---

## Caso 2: Sí es un problema

- Necesitás que el modelo:
  - Mantenga capacidades generales
  - Responda a distintos tipos de instrucciones

👉 Necesitás otra estrategia.

---

## 5. Opción 1: Fine-tuning multitarea (multi-task fine-tuning)

### Qué es

Entrenar el modelo **simultáneamente** en múltiples tareas:

- Clasificación
- Resumen
- Traducción

- Extracción de entidades

## Cómo

- Dataset con **mezcla de tareas**
- Cada ejemplo incluye su instrucción correspondiente

Ejemplo:

Summarize the following text...

Translate this sentence...

Classify this review...

## Coste

- Requiere **mucho más dataset**
- Entre **50.000 y 100.000 ejemplos**
- Más cómputo y entrenamiento

## Beneficio

- Mantiene capacidades generales
- Reduce el olvido catastrófico

El PEFT es un conjunto de técnicas que conserva los pesos del LLM original y solo entrena un pequeño número de capas y parámetros de adaptadores específicos para cada tarea. El PEFT muestra una mayor robustez ante un olvido catastrófico, ya que la mayoría de las pesas previamente entrenadas permanecen inalteradas.

## Opción 2: PEFT (Parameter-Efficient Fine-Tuning)

### Definición

PEFT es un conjunto de técnicas que:

- Congelan los pesos del LLM original
- Entrenan **solo un pequeño conjunto de parámetros nuevos**
- Usan capas adaptadoras específicas por tarea

### Idea clave

No tocar lo que el modelo ya sabe, solo “agregar” conocimiento nuevo.

---

### Por qué reduce el olvido catastrófico

- Los pesos base **no se modifican**
  - Las representaciones originales se preservan
  - El modelo mantiene habilidades previas
- 

### Ejemplo conceptual

- LLM base: intacto
- Adapter de sentiment analysis
- Adapter de resumen

👉 Cambiás de tarea **activando otro adapter**, no reentrenando todo.

---

### Clase Ajuste fino de las instrucciones multitarea

El **multitask fine-tuning** es una extensión del fine-tuning de una sola tarea en la que:

El dataset de entrenamiento contiene **ejemplos de múltiples tareas distintas**, todas expresadas como **pares instrucción → respuesta**, y el modelo se entrena **simultáneamente** en todas ellas.

## Objetivo principal

- Mejorar el desempeño del modelo **en muchas tareas a la vez**
- **Evitar el olvido catastrófico**, típico del fine-tuning en una sola tarea

## Cómo lo logra

- Al mezclar tareas en el entrenamiento:
  - Los gradientes no empujan al modelo hacia una sola habilidad
  - Las representaciones generales se preservan mejor

**El proceso es conceptualmente igual al fine-tuning estándar:**

1. Dataset mixto (múltiples tareas)
2. Entrenamiento durante muchas épocas
3. Para cada ejemplo:
  - El modelo genera una respuesta
  - Se calcula la pérdida (cross-entropy)
  - Se actualizan los pesos
4. El resultado es un **modelo instruido multitarea**

👉 El modelo aprende a ser “bueno en muchas cosas al mismo tiempo”.

**“Desventaja”:** Es más caro en:

- Datos
- Cómputo

Pero **vale la pena** si necesitás capacidades generales sólidas

## Modelos instruct multitarea: FLAN

### Qué es FLAN

FLAN significa *Fine-tuned Language Net*.

FLAN es una estrategia de instruction fine-tuning multitarea que toma un modelo base y lo convierte en un modelo instruct generalista, mejorando su capacidad para seguir instrucciones y resolver múltiples tareas, y sirviendo como una base más eficiente para posteriores ajustes específicos al dominio.

Es:

- Un conjunto de **instrucciones**
- Usadas para fine-tunear modelos base
- Como último paso del entrenamiento

👉 Por eso los autores lo llaman:

“El postre después del plato principal del pre-training”

### Ejemplos de la familia FLAN

- **FLAN-T5** → versión instruct del modelo T5
- **FLAN-PaLM** → versión instruct del modelo PaLM

## 6. FLAN-T5: un instruct model generalista

### Características

FLAN-T5 fue fine-tuneado con:

- **473 datasets**
- **146 categorías de tareas distintas**

### 👉 Resultado:

- Un modelo **generalista**
- Muy bueno siguiendo instrucciones
- Capaz en muchas tareas de NLP

## 7. Dataset de ejemplo: SAMSum

### ¿Qué es SAMSum?

Un dataset usado para entrenar **resumen de diálogos**.

Contiene:

- ~16.000 conversaciones tipo chat
- Escritas por lingüistas
- Con resúmenes humanos de alta calidad

### Particularidad

- Conversaciones estilo Messenger
- Temas cotidianos
- Resúmenes con información clave y nombres

---

## 8. Uso de múltiples instrucciones (prompt templates)

Para una misma tarea (resumir diálogo), se usan **distintas formulaciones**:

Ejemplos:

- “Summarize the following dialogue”

- “What is a summary of this dialogue?”
- “What was going on in that conversation?”

👉 Esto ayuda a que el modelo:

- No memorice una sola instrucción
- Generalice mejor a nuevas formas de pedir lo mismo

## 9. FLAN-T5 + fine-tuning adicional por dominio

### Idea clave

Aunque FLAN-T5 es generalista:

Puede no ser óptimo para **tu dominio específico**

### Ejemplo práctico: atención al cliente

- FLAN-T5 fue entrenado con diálogos cotidianos (SAMSum)
- Pero **no con chats de soporte técnico**

Problema:

- Resúmenes incompletos
- Información inventada (hallucinations)
- Falta de foco en acciones importantes

## 10. Dataset específico: DialogSum

### Qué es DialogSum

- Dataset de **13.000+ chats de soporte**
- Conversaciones reales de atención al cliente
- Con resúmenes orientados a acciones

### Importante

- **NO forma parte del entrenamiento original de FLAN-T5**
- El modelo nunca vio estos datos antes

---

## 11. Resultado del fine-tuning adicional

### Antes del fine-tuning

- Resumen parcial
- Información inventada
- Faltan detalles importantes

### Después del fine-tuning con DialogSum

- Resumen fiel al diálogo
- Sin información fabricada
- Incluye nombres y acciones clave

👉 El modelo se **especializa sin perder capacidades generales.**

## Lección más importante del video

El mayor valor del fine-tuning se obtiene usando **datos propios del dominio real de tu aplicación.**

Ejemplos:

- Chats reales de soporte
- Emails de clientes
- Tickets internos

Esto permite que el modelo:

- Aprenda **qué es importante para tu negocio**
- Se alinee con cómo tu equipo usa la información

### **Clase evaluacion del modelo:**

Cuando en el curso se dice:

*“el modelo mostró buen desempeño”*  
*“el modelo fine-tuneado mejoró respecto al base”*

eso **no es una opinión**, es una afirmación que **debe estar respaldada por métricas**.

### **Problema clave**

En **machine learning tradicional**:

- El modelo es **determinista**
- Hay una única salida correcta
- Métricas simples como **accuracy** funcionan bien

En **LLMs**:

- El modelo es **no determinista**
- Hay **múltiples salidas válidas**
- El output es **lenguaje natural**
- No alcanza con “es correcto / incorrecto”

👉 **Evaluar lenguaje es difícil**, porque:

- Frases muy parecidas pueden significar lo mismo
- Frases casi iguales pueden significar lo opuesto

Ejemplo:

- “*Mike loves tea*”
- “*Mike adores sipping tea*” → mismo significado
- “*Mike drinks coffee*” vs “*Mike does not drink coffee*” → una palabra cambia todo

Por eso necesitamos **métricas automáticas, estructuradas y reproducibles**.

---

## 2. Idea central de las métricas ROUGE y BLEU

Ambas métricas **comparan texto generado vs texto humano de referencia**.

No intentan “entender” el significado profundo, sino medir **similitud superficial controlada**:

- palabras
- secuencias de palabras
- orden relativo

💡 Son métricas **proxy**: no miden comprensión, miden **coincidencia estructural**.

---

## 3. Conceptos base (terminología mínima)

- **Unigram**: una palabra
- **Bigram**: dos palabras consecutivas
- **N-gram**: secuencia de  $n$  palabras

Ejemplo:

“It is cold outside”

- Unigrams: `It, is, cold, outside`
  - Bigrams: `It is, is cold, cold outside`
- 

## 4. ROUGE – Evaluación de tareas de *summarization*

### ¿Para qué se usa ROUGE?

- Evaluar **resúmenes automáticos**
- Comparar resumen generado vs resumen humano

ROUGE significa:

**Recall-Oriented Understudy for Gisting Evaluation**

👉 Está **orientada al recall**, porque en un resumen importa **no perder información clave**.

---

## 5. ROUGE-1 (unigrams)

### Qué mide

Coincidencia de **palabras individuales**, sin importar el orden.

### Métricas usadas

- **Recall:**

¿Cuántas palabras del resumen humano aparecen en el generado?

- **Precision:**

¿Cuántas palabras del generado son correctas?

- **F1:**

Balance entre precisión y recall

## Ejemplo

Referencia:

*It is cold outside.*

Generado:

*It is very cold outside.*

- Recall =  $4 / 4 = 1.0$
- Precision =  $4 / 5 = 0.8$
- F1  $\approx 0.89$

## Limitación

✗ No considera:

- orden
- contexto
- negaciones

“It is **not** cold outside”  
tendría el **mismo ROUGE-1**

---

## 6. ROUGE-2 (bigrams)

### Qué mejora

- Introduce **orden local**
- Evalúa pares de palabras

### Resultado

- Scores más bajos
- Más estrictos
- Más realistas para frases largas

📌 A mayor  $n$ , menor probabilidad de coincidencia exacta.

---

## 7. ROUGE-L (Longest Common Subsequence)

### Idea clave

En vez de n-grams fijos, busca la:

**subsecuencia común más larga (LCS)**

- No requiere que las palabras estén contiguas
- Sí respeta el **orden**

### Métrica

- Usa LCS como numerador
- Calcula recall, precision y F1

### Ventaja

✓ Captura mejor la **estructura global de la frase**

### Limitación

✗ Sigue siendo superficial  
✗ No detecta cambios semánticos sutiles

---

## 8. Problema crítico: ROUGE puede ser engañoso

Ejemplo:

Generado: *cold cold cold cold*

- ROUGE-1 Precision = 1.0
- Claramente **mal output**

### Solución parcial: clipping

- Limita cuántas veces una palabra puede “contar”
- Se ajusta al máximo de apariciones en la referencia

✓ Reduce repetición artificial

✗ No resuelve el problema del orden:

*outside cold it is*

Sigue puntuando alto

📌 **Conclusión:**

ROUGE sirve para **diagnóstico**, no como evaluación final.

---

## 9. BLEU – Evaluación de traducción automática

BLEU significa:

Bilingual Evaluation Understudy

### ¿Para qué se usa?

- Traducción automática
- Comparar traducción generada vs humana

### Idea central

- Calcula **precisión promedio** sobre varios tamaños de n-grams
- Normalmente de 1 a 4-grams
- Resultado  $\in [0, 1]$

📌 BLEU  $\approx$  “qué tan similar estructuralmente es la traducción”

---

## 10. Interpretación de BLEU

Ejemplo (resumido):

Referencia:

*I am very happy to say that I am drinking a warm cup of tea.*

Outputs:

- Más diferente → BLEU ≈ 0.49
- Más similar → BLEU ≈ 0.73
- Casi idéntico → BLEU ≈ 1.0

✓ A mayor coincidencia estructural → mayor BLEU

✗ No evalúa fluidez ni significado profundo

**ROUGE y BLEU no evalúan “inteligencia”**

Evalúan **similitud estructural automática**

Son:

- rápidas
- baratas
- útiles para iterar modelos
- buenas para comparar *antes vs después del fine-tuning*

Pero:

✗ No capturan semántica profunda

✗ No detectan errores lógicos

✗ No reemplazan benchmarks humanos

👉 Por eso:

- Se usan **como métricas intermedias**
- La evaluación final se hace con **benchmarks estandarizados**

### **Clase Benchmark:**

ROUGE y BLEU sirven para iterar y diagnosticar  
Los benchmarks sirven para evaluar y comparar modelos de verdad

👉 En proyectos reales:

- Usás métricas simples durante el desarrollo
- Usás benchmarks para la evaluación final

Un **benchmark** es:

- Un **conjunto de datasets diseñados específicamente**
- Con tareas que aíslan **habilidades concretas** del modelo

Ejemplos de habilidades evaluadas:

- Razonamiento
- Comprensión lectora
- Sentido común
- Conocimiento del mundo
- Riesgos: sesgo, toxicidad, copyright

📍 La clave es evaluar **capacidades**, no solo outputs aislados.

## **Importancia del dataset de evaluación**

Para que la evaluación sea válida:

- El modelo **no debe haber visto los datos durante el entrenamiento**
- Evaluar con datos conocidos da una **falsa sensación de buen desempeño**

👉 La evaluación debe hacerse sobre **datos no vistos (unseen data)**.

---

### **Clase Ajuste fino eficaz de los parámetros (PEFT)**

PEFT es un conjunto de técnicas donde:

**La mayoría (o todos) los pesos del LLM se mantienen congelados**  
y solo se entrenan **muy pocos parámetros adicionales o seleccionados**.

### Idea clave

No “reescribís” el modelo → **lo adaptás**

---

## 4. Ventajas principales de PEFT

### ◆ Eficiencia de memoria

- Entrenás **una fracción** de los parámetros
- A veces **solo MBs**
- Puede entrenarse en **una sola GPU**

### ◆ Menos catastrophic forgetting

- El conocimiento base se conserva
- El modelo original casi no cambia

### ◆ Flexibilidad

- Un solo LLM base
- Múltiples “extensiones” por tarea
- Se **intercambian pesos PEFT** según la tarea

### ◆ Escalabilidad

- Muchas tareas → poco espacio
  - Ideal para producción
-

## 5. Comparación directa: Full FT vs PEFT

Aspecto	Full Fine-Tuning	PEFT
Pesos entrenados	100%	1–20% (a veces menos)
Tamaño por tarea	GBs	MBs
Hardware	Multi-GPU / cloud	1 GPU posible
Catastrophic forgetting	Alto	Bajo
Flexibilidad	Baja	Alta
Almacenamiento	Muy caro	Barato

## ¿Cómo funciona PEFT en la práctica?

- El LLM base queda congelado
- Se entrenan pesos pequeños específicos por tarea
- En inferencia:
  - LLM base + pesos PEFT
  - Cambiás de tarea → cambiás pesos PEFT

💡 Es como:

“Un motor base + distintos kits de adaptación”

Tenés UN solo LLM base (congelado) y VARIOS conjuntos de pesos PEFT, cada uno especializado en una tarea.

Cuando:

- querés **resumir** → cargás **PEFT-Summarize**
- querés **QA** → cargás **PEFT-QA**
- querés **generar código** → cargás **PEFT-Generate**

## PEFT methods

### Selective

Select subset of initial LLM parameters to fine-tune

### Reparameterization

**Reparameterize** model weights using a low-rank representation

LoRA

### Additive

Add trainable layers or parameters to model

Adapters

Soft Prompts

Prompt Tuning

## Reparameterization Methods – LoRA



### Idea clave (una frase)

No entrenás los pesos grandes del modelo, entrenás una corrección pequeña y barata sobre ellos.



### Problema que resuelve LoRA

En un Transformer hay matrices **gigantes** (ej. atención):

$$W_q, W_k, W_v, W_o \in \mathbb{R}^{\text{d} \times \text{d}}$$

Entrenar estas matrices completas:

- cuesta memoria
- cuesta cómputo
- es innecesario para una tarea específica

## Solución LoRA (qué hace exactamente)

En lugar de entrenar  $W$ , LoRA dice:

“Dejá  $W$  fija y aprendé un **delta pequeño** que la ajuste”.

Matemáticamente:

$$W' = W + \Delta W$$

$$\Delta W = A \cdot B$$

Donde:

- $A \in \mathbb{R}^r(d \times r)$
- $B \in \mathbb{R}^r(r \times d)$
- $r = \text{rango bajo}$  (ej. 4, 8, 16)

  $r \ll d$ , por eso:

- entrenás MUCHÍSIMOS menos parámetros
- $\Delta W$  es barato

**d = dimensión del modelo**

Es simplemente **cuántos números usa el modelo para representar UNA palabra internamente**.

**r = rango (rank) de LoRA**

Es un **número chico** que vos elegís, que controla:

- cuánta libertad tiene LoRA
- cuántos parámetros nuevos entrenás

r define **qué tan grande es la corrección que LoRA puede aprender**

**Ejemplo real:**

$d = 4096$

$r = 8$

Entonces:

- Matriz original  $W \rightarrow 4096 \times 4096 \approx 16$  millones de parámetros
- LoRA:
  - $A \rightarrow 4096 \times 8$
  - $B \rightarrow 8 \times 4096$
  - Total  $\approx 65$  mil parámetros

🔥 Esa es la magia de LoRA

### **Wq, Wk, Wv**

Matrices del modelo (no de palabras), usadas para crear Query, Key y Value

- Son gigantes
- Entrenarlos cuesta mucho

Ahí entra LoRA.

En vez de cambiar  $W$ :

**Aprendemos un pequeño ajuste que se suma encima**

Pensalo así:

- $W$  = libro enorme
- LoRA = post-it con correcciones

---

**Qué significa esto:**

$W' = W + \Delta W$

👉 Traducción:

El peso final es el peso original **más un pequeño ajuste**

---

## ¿Qué es $\Delta W$ ?

$\Delta W$  no es otra matriz gigante.

Se construye así:

$$\Delta W = A \cdot B$$

Donde:

- A y B son **muchísimo más chicas**
- Por eso entrenarlas es barato

📌 Analogía:

En vez de aprender una enciclopedia nueva, aprendés un resumen.

---

## 🔍 ¿Qué significa “low-rank”?

Una matriz grande suele tener:

- mucha redundancia
- direcciones principales

LoRA aprende **solo esas direcciones importantes** para la tarea.

📌 Analogía:

No reescribís el texto entero,  
**agregás un post-it con correcciones.**

---

## Cómo fluye en inferencia

Durante el forward:

$$y = (W + A \cdot B) \cdot x$$

 Importante:

- $W$  nunca se toca
- $A$  y  $B$  son los pesos PEFT
- Podés activar o desactivar LoRA

---

## ¿Dónde se aplica LoRA?

Generalmente en:

- Attention (Q, K, V, O)
- A veces en FFN

Ejemplo real:

`apply LoRA to Wq and Wv only`

## Qué es *attention*

### Idea central

Attention es el mecanismo que le permite al modelo decidir a qué partes del texto prestar atención cuando procesa una palabra.

Ejemplo simple:

“Juan fue al banco a depositar dinero.”

Cuando el modelo procesa la palabra “**banco**”, necesita decidir:

- ¿banco financiero?
- ¿banco de sentarse?

👉 Para eso mira **otras palabras del contexto** (“depositar”, “dinero”).

Eso es *attention*.

---

## ✓ Ventajas de LoRA

- Excelente balance calidad / costo
  - Muy usado en industria
  - Compatible con quantization (QLoRA)
  - Fácil de “enchufar y desenchufar”
- 

## ⚠ Limitaciones

- Un LoRA ≠ multitarea
- Cada tarea necesita su LoRA
- r mal elegido → mala performance

# Additive Methods – Adapters & Prompt Tuning

Acá **NO se modifican pesos existentes**, ni siquiera con deltas.

---

## 2.1 Adapters

### 🧠 Idea clave

**Agregás mini-capas entrenables dentro del modelo.**

---

## Cómo funcionan

En cada bloque Transformer:

Attention

↓

Adapter (nuevo, entrenable)

↓

FFN

Un adapter típico es:

$x \rightarrow \text{Down-project} \rightarrow \text{ReLU} \rightarrow \text{Up-project} \rightarrow x + \text{residual}$

### Son:

- chicos
- entrenables
- insertados en puntos específicos

---

## Inferencia

- El flujo pasa por:
  - pesos congelados
  - adapters entrenados

Podés:

- activar adapter A
  - desactivar adapter B
-

## Ventajas

- Muy controlable
  - No toca pesos base
  - Buena estabilidad
- 

## Desventajas

- Cambia la arquitectura
  - Más overhead que LoRA
  - Inferencia un poco más lenta
- 

## 2.2 Soft Prompts / Prompt Tuning

### Idea clave

No cambiás el modelo, cambiás lo que “ve” al entrar.

---

### Qué se entrena

En vez de texto real, se entrenan:

`[p1, p2, p3, ..., p] ← embeddings aprendidos`

Estos se anteponen al input:

`[p1 p2 p3] + "Resumí este texto..."`

-  No son palabras reales
  -  Son vectores entrenables
- 

### Qué aprende el modelo

Aprende:

- *cómo interpretar la consigna*
- *cómo comportarse ante esa tarea*

💡 Es como:

“Darle instrucciones implícitas internas”

---

## ✓ Ventajas

- Ultra liviano
  - Muy barato
  - Cero cambio de arquitectura
- 

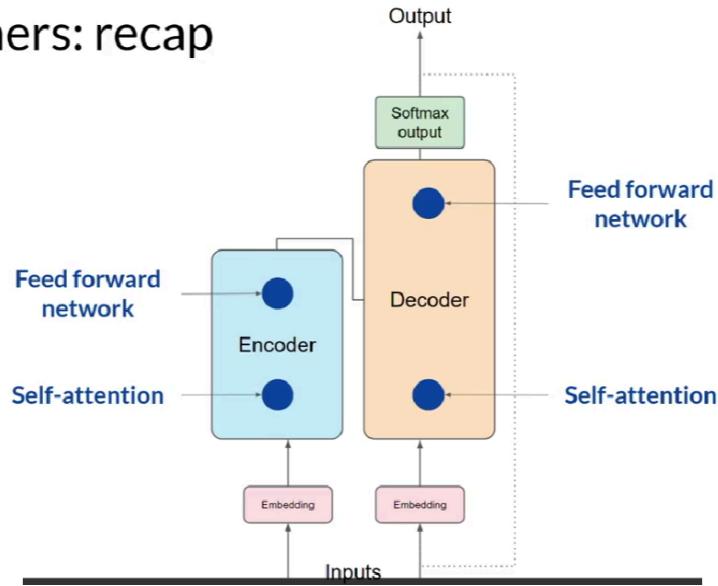
## ⚠ Desventajas

- Capacidad limitada
  - No ideal para tareas complejas
  - Menos expresivo que LoRA
- 

## Clase Técnicas PEFT 1: LoRA

La self-attention y feedforward network son redes neuronales que forman parte de los encoder y decoder de un transformer.

## Transformers: recap



Tienen pesos que se aprenden durante el entrenamiento.

## Dónde encaja LoRA dentro del Transformer

Un **Transformer** (como FLAN-T5, GPT, LLaMA) procesa texto así:

1. Texto → tokens
2. Tokens → **embeddings** (vectores de tamaño  $d$ )
3. Esos vectores pasan por muchos **bloques Transformer**
4. Cada bloque tiene **dos grandes partes**:
  - **Self-Attention**
  - **Feed-Forward Network (FFN)**

📌 Durante el **pre-entrenamiento**, el modelo aprende **todos los pesos** de:

- Attention
- FFN

Eso es lo que hace que los pesos sean **muy grandes y muy costosos** de re-entrenar.

## Qué pasa en *full fine-tuning* (para contrastar)

En full fine-tuning:

- Se actualizan **todos los pesos**
- Incluye:
  - Attention ( $W_q$ ,  $W_k$ ,  $W_v$ ,  $W_o$ )
  - FFN
- Resultado:
  - Mucha memoria
  - Mucho cómputo
  - Un modelo nuevo por tarea

👉 LoRA nace para **evitar esto.**

## Idea central de LoRA (en limpio)

**LoRA congela todos los pesos originales del modelo  
y solo aprende una corrección pequeña sobre algunos de ellos.**

No reemplaza nada.

No borra nada.

**Suma un ajuste.**

---

## 4) Qué significa “re-parameterization”

No es un término raro, significa:

**Representar un cambio grande usando pocos parámetros bien estructurados**

En vez de decir:

- “aprendo una matriz enorme nueva”

Decís:

- “aprendo dos matrices chicas cuyo producto se comporta como una grande”

Eso es **re-parametrizar**.

---

## 5) Qué pesos toca LoRA (muy importante)

En la práctica, LoRA se aplica sobre todo a:

- **Self-Attention**
  - Wq (Query)
  - Wk (Key)
  - Wv (Value)
  - Wo (Output)

👉 ¿Por qué?

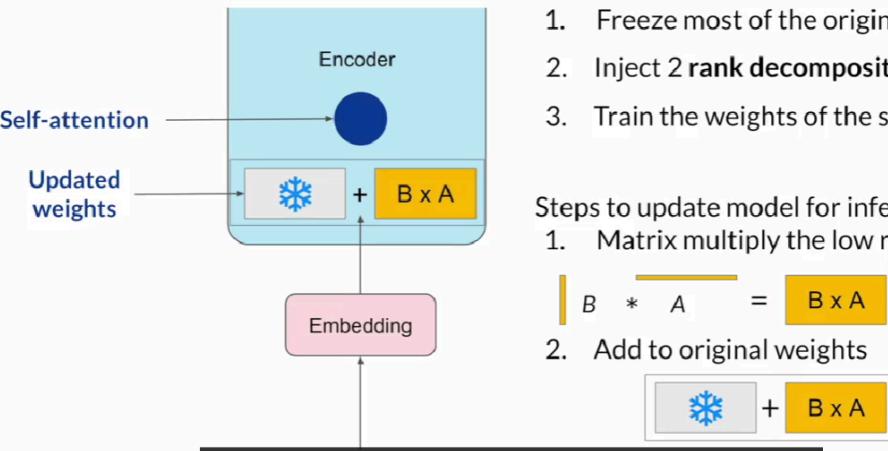
- Ahí está **la mayoría de los parámetros**
- Ahí está **el control del comportamiento contextual**

👉 Ajustar attention = ajustar *cómo el modelo presta atención*

---

## 6) Qué hace LoRA exactamente (sin fórmulas duras)

## LoRA: Low Rank Adaption of LLMs



1. Freeze most of the original LLM weights.
2. Inject 2 **rank decomposition matrices**
3. Train the weights of the smaller matrices

Steps to update model for inference:

1. Matrix multiply the low rank matrices

$$B \times A = B \times A$$

2. Add to original weights

$$\text{snowflake} + B \times A$$

Tomemos **una matriz de attention**, por ejemplo  $W_q$ .

### Antes:

- $W_q$  es una matriz grande
- Se aprendió en el pre-entrenamiento
- Está congelada

### Con LoRA:

- No la tocás
- Le agregás **dos matrices chicas**:
  - A
  - B

Durante el uso real del modelo:

- Se calcula  $A \times B$
- Eso genera una **corrección**
- Esa corrección se **suma** a  $W_q$

👉 El modelo sigue teniendo el mismo tamaño (cant de parametros) **El producto  $A \cdot B$  tiene EXACTAMENTE la misma dimensión que la matriz base  $W$**  y lo único que se hace es **sumar valores posición a posición**.

👉 El forward es casi igual

👉 La inferencia no se vuelve más lenta

### Ejemplo:

#### Matriz original

- Dimensiones: **512 × 64**

Parámetros:

$$512 \times 64 = 32.768$$

#### LoRA con rango $r = 8$

Se crean dos matrices:

- **A**:  $8 \times 64 \rightarrow 512$  parámetros
- **B**:  $512 \times 8 \rightarrow 4.096$  parámetros

Total entrenable:

$$512 + 4.096 = 4.608$$

📈 Pasaste de:

- 32.768 parámetros  
a
- 4.608 parámetros

👉 **86% menos parámetros entrenables**

Y eso es por cada matriz de attention.

👉 Hallazgo importante del paper:

- Para  $r > 16$ , **la mejora se estanca**

- Más parámetros ≠ mejor performance

👉 Por eso hoy se usan valores típicos:

$r = 4, 8, 16, 32$

## Cambio de tareas con LoRA (parte clave del PEFT)

Supongamos:

- LoRA-A → resumen
- LoRA-B → QA
- LoRA-C → diálogo

El modelo base es el mismo.

Para inferencia:

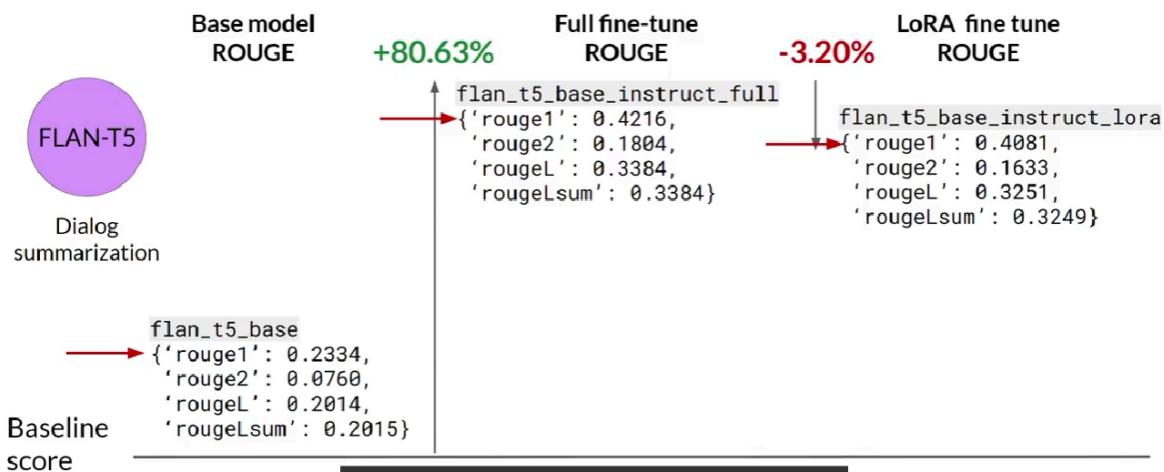
1. Cargás el LLM base
2. Cargás el LoRA correspondiente
3. Usás ese comportamiento

Cambiar de tarea = **cambiar matrices A y B**

No el modelo entero.

cada ejecución del modelo para una tarea específica requiere el LoRA correspondiente a esa tarea.

## Sample ROUGE metrics for full vs. LoRA fine-tuning



En esta imagen se compara usar flan-t5 base solo (chequeando con metrica ROUGE, con respecto a full fine tune y Lora fine tune. Fijate que entre estos dos ultimos, Lora tuvo casi la misma performance que el full fine tune pero habiendo modificado muchisimo menos parametros y usando mucho menos recursos y tiempo.

### Clase prompt engineering:

Cuando queremos adaptar un **LLM preentrenado** a una tarea específica tenemos varias opciones:

- **Fine-tuning completo** → caro, muchos parámetros, alto consumo de GPU
- **PEFT (Parameter Efficient Fine-Tuning)** → entrenar *muy pocos parámetros*
  - **LoRA** → modifica pesos de manera eficiente
  - **Prompt Tuning** → **no toca los pesos del modelo**

👉 **Prompt tuning es un método PEFT aditivo:** mejora el rendimiento **sin cambiar los pesos del modelo base**.

### **Prompt tuning:**

- Se añaden tokens entrenables a los prompts, permitiendo que el proceso de aprendizaje supervisado determine sus valores óptimos.
- A diferencia de la ingeniería de prompts, que requiere esfuerzo manual, el ajuste de prompts optimiza automáticamente el rendimiento del modelo.

En lugar de modificar el modelo completo, se añaden "tokens entrenables" a la entrada del modelo, que se conocen como "soft prompts". Estos tokens son como pequeñas instrucciones que ayudan al modelo a entender mejor la tarea que se le está pidiendo. A diferencia de las palabras fijas que ya conoce el modelo, estos soft prompts pueden adoptar diferentes valores y se ajustan durante el proceso de aprendizaje supervisado para maximizar el rendimiento en una tarea específica.

## Estructura conceptual

[ Soft Prompt Tokens ] + [ Tokens reales del texto ]

- Cantidad típica: **20 a 100 tokens**
- Cada token = vector de dimensión `d_model`

Ejemplo conceptual:

[ v1 v2 v3 ... v50 ] + "The teacher teaches the student with the book."

## ¿Cómo se entrena Prompt Tuning?

### Prompt Tuning

- Dataset: `(prompt, output)`
- **Pesos del LLM congelados**
- Solo se entrenan:
  - embeddings de los soft prompts

💡 El gradiente **solo fluye hacia los soft prompts**, no hacia el modelo base.

Imagina que estás enseñando a un niño a resolver un rompecabezas. En lugar de darle todas las piezas y esperar que las coloque correctamente, le das algunas pistas que le ayudan a entender cómo encajar las piezas. De manera similar, los soft prompts actúan como esas pistas, guiando al modelo para que produzca mejores respuestas. Al usar entre 20 y 100 de estos tokens virtuales, el modelo puede aprender a realizar tareas específicas de manera más eficiente, sin necesidad de cambiar su estructura interna.

Cuando se realiza el ajuste fino, se utiliza un conjunto de datos que incluye ejemplos de entrada (prompts) y las salidas deseadas (completions). Durante este proceso, el modelo ajusta los valores de los soft prompts para maximizar su rendimiento en la tarea específica.

### Comparación de Métodos

- El ajuste de prompts es más eficiente en términos de parámetros, ya que solo se entrena unos pocos en comparación con millones en el ajuste completo.
- A medida que aumenta el tamaño del modelo, el rendimiento del ajuste de prompts se aproxima al del ajuste completo, especialmente en modelos con alrededor de 10 mil millones de parámetros.

## ¿Por qué es extremadamente eficiente?

- Parámetros entrenables: **muy pocos**
- Tamaño en disco: **mínimo**
- Tiempo de entrenamiento: **bajo**
- GPU requerida: **mucho menor**

👉 Similar en espíritu a LoRA, pero **sin tocar pesos**.

## Prompt Tuning para múltiples tareas

## Idea central

Un **mismo LLM base** puede servir para **muchas tareas**, cambiando solo el soft prompt.

LLM congelado

↑

Soft Prompt A → Task A

Soft Prompt B → Task B

Soft Prompt C → Task C

---

## En inferencia

- No reentrenás nada
- Simplemente:
  - cargás otro soft prompt
  - lo antepones al input

📌 Cambiar de tarea = **cambiar soft prompt**

## Prompt Tuning vs LoRA

Aspecto	Prompt Tuning	LoRA
Modifica pesos	✗ No	✓ Sí
Entrena pocos parámetros	✓	✓
Ubicación	Input (embeddings)	Capas internas
Interpretabilidad	Baja	Media
Uso práctico	Menos común	Muy común
Rendimiento	Excelente en LLMs grandes	Excelente en general

📌 LoRA es más usado hoy, pero prompt tuning es conceptualmente más limpio.

## Rendimiento: ¿qué tan bien funciona?

### Resultados del paper original (Lester et al., Google)

Evaluación con **SuperGLUE**:

- Modelos chicos:
  - Prompt tuning < fine-tuning
- Modelos grandes ( $\approx 10B$  parámetros):
  - Prompt tuning  $\approx$  fine-tuning completo
  - Mucho mejor que prompt engineering

👉 Cuanto más grande el modelo, mejor funciona prompt tuning

### Resumen final ultra sintético

**Prompt tuning** es una técnica PEFT que agrega **tokens virtuales entrenables** al inicio del input, manteniendo el modelo base completamente congelado.

Estos tokens se optimizan mediante aprendizaje supervisado y permiten adaptar un mismo LLM a múltiples tareas con un costo computacional mínimo.

En modelos grandes, su rendimiento puede igualar al fine-tuning completo.

### LABORATORIO EXPLICADO CON GPT:

## 2) Full Fine-Tuning (Instruction Fine-Tuning)

### 2.1) Preprocesamiento (tokenización + armar prompts)

#### ✓ `tokenize_function` (celda 23)

```
def tokenize_function(example):  
    start_prompt = 'Summarize the following conversation.\n\n'  
    end_prompt = '\n\nSummary: '  
    prompt = [start_prompt + dialogue + end_prompt for dialogue in  
example["dialogue"]]
```

```

        example['input_ids'] = tokenizer(prompt, padding="max_length",
truncation=True, return_tensors="pt").input_ids
        example['labels'] = tokenizer(example["summary"],
padding="max_length", truncation=True,
return_tensors="pt").input_ids

    return example

```

**Esta celda es CLAVE.**

## Qué produce

- `input_ids`: tokens del **prompt** (instrucción + diálogo)
- `labels`: tokens del **resumen humano** (lo que el modelo debe aprender a generar)

En un modelo seq2seq:

- Entrada: `input_ids`
- Objetivo: `labels`
- El entrenamiento minimiza la loss de generar `labels` dado `input_ids`.

## Detalles importantes

- `padding="max_length"`: rellena hasta largo fijo (más simple, pero desperdicia tokens).
- `truncation=True`: si se pasa del máximo, corta.
- `batched=True` cuando hacés `dataset.map(...)`: entra un batch (listas), por eso `prompt = [...]`.

Luego:

```

tokenized_datasets = dataset.map(tokenize_function, batched=True)
tokenized_datasets =
tokenized_datasets.remove_columns(['id', 'topic', 'dialogue', 'summary'])

```

Elimina columnas textuales porque ya no las necesitas para entrenar: te quedás con tensores.

---

### Reducir dataset para que el lab sea rápido (celda 25)

```
tokenized_datasets = tokenized_datasets.filter(lambda example,  
index: index % 100 == 0, with_indices=True)
```

**Qué hace:**

- Se queda con 1 de cada 100 ejemplos (submuestreo).

**Por qué:**

- Entrenar full fine-tuning real con todo el dataset llevaría mucho.
- 

### Ver tamaños (celda 27)

```
print(tokenized_datasets['train'].shape)  
...
```

Te muestra cuántos ejemplos quedaron en cada split tras el filtro.

---

## 2.2) Entrenar full fine-tuning con Trainer

### TrainingArguments + Trainer (celda 30)

```
training_args = TrainingArguments(  
    output_dir=output_dir,  
    learning_rate=1e-5,  
    num_train_epochs=1,  
    weight_decay=0.01,  
    logging_steps=1,  
    max_steps=1  
)
```

```
trainer = Trainer(  
    model=original_model,  
    args=training_args,  
    train_dataset=tokenized_datasets['train'],  
    eval_dataset=tokenized_datasets['validation'])
```

## Qué significa cada parámetro importante

- `learning_rate=1e-5`: LR típico bajo para full fine-tuning (si es alto, rompés el modelo).
- `num_train_epochs=1`: 1 pasada sobre el dataset.
- `weight_decay=0.01`: regularización.
- `logging_steps=1`: loguea muy seguido (porque el entrenamiento es mini).
- `max_steps=1`: **esto lo hace ultra corto** (entrena solo 1 step).  
👉 En el video, explican que es solo para demo, y por eso luego descargan un checkpoint “mejor” ya entrenado offline.

---

## ✓ Entrenar (celda 32)

```
trainer.train()
```

Esto ejecuta el loop:

- forward pass
- loss
- backprop
- update de pesos (en full fine-tuning: actualiza TODO)

## QUÉ HAY EXACTAMENTE en `tokenized_datasets`?

Después de estas líneas:

```
tokenized_datasets = dataset.map(tokenize_function, batched=True)
```

```
tokenized_datasets = tokenized_datasets.remove_columns(  
    ['id', 'topic', 'dialogue', 'summary']  
)
```

- 👉 YA NO HAY TEXTO
  - 👉 SOLO HAY NÚMEROS (TOKENS)
- 

## 📦 Estructura FINAL del dataset

Cada split (`train`, `validation`, `test`) tiene **solo 2 columnas**:

```
['input_ids', 'labels']
```

Nada más.

---

## 🧠 ¿Qué es `input_ids` EXACTAMENTE?

`input_ids` = lo que el modelo LEE

Para **cada ejemplo**, `input_ids` contiene los **tokens del prompt completo**, o sea:

"Summarize the following conversation.

<DIÁLOGO COMPLETO>

Summary:"

Pero **no como texto**, sino como **IDs numéricos del tokenizer**.

---

## Ejemplo mental REAL

Supongamos este diálogo:

Person 1: Hi, how are you?

Person 2: I'm fine, thanks.

El prompt que se arma es:

Summarize the following conversation.

Person 1: Hi, how are you?

Person 2: I'm fine, thanks.

Summary:

El tokenizer lo convierte en algo así:

```
input_ids = [  
    43921, 342, 8921, 31, 7, 19, 56, ...  
]
```

📌 **Eso** es lo que hay en `input_ids`.

---

## 🧠 ¿Qué es **labels** EXACTAMENTE?

**labels** = lo que el modelo DEBE APRENDER A GENERAR

Para **ese mismo ejemplo**, `labels` contiene solo el resumen humano, tokenizado:

Person 1 greets Person 2 and they exchange pleasantries.

Tokenizado queda algo así:

```
labels = [  
    712, 19, 342, 4891, 31, ...  
]
```

📌 **Esto** es el objetivo del entrenamiento.

## 2.3) Descargar modelo “bien entrenado” desde S3 y evaluarlo

✓ **Download checkpoint (celda 35–37)**

```
!aws s3 cp --recursive s3://.../flan-dialogue-summary-checkpoint/  
./flan-dialogue-summary-checkpoint/  
!ls -alh ./flan-dialogue-summary-checkpoint/pytorch_model.bin
```

**Idea:** usan un checkpoint preentrenado offline (más steps/epochs) para que veas diferencia real.

---

### ✓ Cargar `instruct_model` (celda 39)

```
instruct_model =  
AutoModelForSeq2SeqLM.from_pretrained("./flan-dialogue-summary-check  
point", torch_dtype=torch.bfloat16)
```

Ahora tenés:

- `original_model` (base)
  - `instruct_model` (full fine-tuned para resumir diálogos)
- 

## 2.3) Evaluación cualitativa: comparar outputs (celda 41)

Genera resumen con ambos modelos usando el mismo prompt:

- imprime:
  - resumen humano
  - salida del original
  - salida del instruct

Esto es “a ojo” (sirve mucho para intuición).

---

## 2.4) Evaluación cuantitativa con ROUGE

### ✓ Cargar ROUGE (celda 43)

```
rouge = evaluate.load('rouge')
```

ROUGE mide similitud con el resumen humano (por n-gramas / subsecuencias).

---

### Generar resúmenes para 10 ejemplos y armar DataFrame (celda 45)

- Toma `dataset['test'][0:10]`
- Para cada diálogo:
  - genera resumen con `original_model`
  - genera resumen con `instruct_model`
- arma una tabla con:
  - humano vs original vs instruct

---

### Calcular ROUGE (celda 47)

```
original_model_results = rouge.compute(predictions=...,
references=..., use_aggregator=True, use_stemmer=True)
instruct_model_results = rouge.compute(...)
```

- `use_aggregator=True`: devuelve promedio global.
- `use_stemmer=True`: reduce palabras a su raíz (mejor comparación “semántica” ligera).

Imprime resultados:

- `rouge1, rouge2, rougeL, rougeLsum`

## 3) PEFT con LoRA

### 3.1) Configurar LoRA (celda 54)

```

lora_config = LoraConfig(
    r=32,
    lora_alpha=32,
    target_modules=[ "q", "v" ],
    lora_dropout=0.05,
    bias="none",
    task_type=TaskType.SEQ_2_SEQ_LM
)

```

## Qué significa cada parámetro

- `r=32` (**rank**): tamaño de las matrices low-rank A y B.  
Más `r` = más capacidad, más params entrenables.
- `lora_alpha=32`: factor de escalado (controla magnitud del update).
- `target_modules=[ "q", "v" ]`: aplica LoRA a proyecciones **Query** y **Value** en atención.  
(Es el lugar típico donde LoRA rinde bien.)
- `lora_dropout=0.05`: dropout en el camino LoRA.
- `bias="none"`: no ajusta bias.
- `task_type=SEQ_2_SEQ_LM`: porque FLAN-T5 es seq2seq.

## 3.1) “Convertir” el modelo base a PEFT (celda 56)

```

peft_model = get_peft_model(original_model, lora_config)
print(print_number_of_trainable_model_parameters(peft_model))

```

### Qué hace `get_peft_model`:

- Envuelve el modelo base y le “inyecta” adaptadores LoRA en los módulos target.
- Resultado: el modelo ahora tiene:
  - pesos base “congelados”
  - parámetros LoRA “entrenables”

Por eso el % entrenable baja muchísimo (ej: ~1.4%).

---

### 3.2) Entrenar PEFT (celda 58–60)

```
peft_training_args = TrainingArguments(  
    auto_find_batch_size=True,  
    learning_rate=1e-3,  
    num_train_epochs=1,  
    max_steps=1  
)  
peft_trainer = Trainer(model=peft_model, args=...,  
train_dataset=...)  
peft_trainer.train()
```

#### Puntos clave

- `learning_rate=1e-3` es **mucho más alto** que full fine-tuning.
  - Tiene sentido porque estás entrenando muy pocos parámetros (LoRA) y el update es “contenido”.
- `auto_find_batch_size=True`: intenta elegir un batch size que no explote la memoria.

Luego guardan SOLO el adaptador:

```
peft_trainer.model.save_pretrained(peft_model_path)  
tokenizer.save_pretrained(peft_model_path)  
(OSEA guardas el adaptador normal y el tokenizado por asi decir y dsp podes con esto hacer:  
from peft import PeftModel  
from transformers import AutoModelForSeq2SeqLM, AutoTokenizer  
  
base_model = AutoModelForSeq2SeqLM.from_pretrained("google/flan-t5-base")  
peft_model = PeftModel.from_pretrained(base_model, peft_model_path)  
tokenizer = AutoTokenizer.from_pretrained(peft_model_path)  
Y ya funciona, sin entrenar nada.
```

### 3.2) Descargar adaptador “offline” desde S3 (celda 63–65)

```
!aws s3 cp --recursive s3://.../peft-dialogue-summary-checkpoint/  
./peft-dialogue-summary-checkpoint-from-s3/
```

```
!ls -al ./peft-dialogue-summary-checkpoint-from-s3/adapter_model.bin
```

Acá aparece la diferencia fuerte de tamaño:

- full fine-tuned: ~945MB (modelo completo)
  - LoRA adapter: ~14MB (solo deltas/adaptadores)
- 

### 3.3) Cargar base + adapter y dejarlo listo para inferencia (celda 67–69)

```
peft_model_base =  
AutoModelForSeq2SeqLM.from_pretrained("google/flan-t5-base",  
torch_dtype=torch.bfloat16)  
peft_model = PeftModel.from_pretrained(  
    peft_model_base,  
    './peft-dialogue-summary-checkpoint-from-s3/',  
    torch_dtype=torch.bfloat16,  
    is_trainable=False  
)
```

**Lo más importante: esto es la “combinación”**

- Cargás el **modelo base** FLAN-T5.
- Le “montás arriba” el **adaptador LoRA**.

**is\_trainable=False**

Le dice a PEFT/PyTorch:

- “No voy a entrenar; solo inferencia”
- Entonces minimiza cosas internas (menos overhead, menos memoria para gradientes, etc.)

Luego:

```
print(print_number_of_trainable_model_parameters(peft_model))
```

Debería dar ~0% entrenable (porque no querés gradientes).

### 3.3) Evaluación cualitativa (celda 71)

Compara en el mismo ejemplo:

- humano
- original
- instruct (full fine-tuned)
- peft (LoRA)

Así ves rápido si PEFT quedó “cerca”.

---

### 3.4) Evaluación cuantitativa con ROUGE (celda 73–75)

#### Celda 73:

Genera resúmenes para 10 ejemplos con los 3 modelos.

#### Celda 75:

Calcula ROUGE para:

- original vs humano
- instruct vs humano
- peft vs humano

Deberías ver:

- **instruct** mejora fuerte vs original
  - **peft** queda bastante cerca de instruct, quizá un poco peor
- 

### 3.4) Evaluación en grande con CSV (celda 78, 80, 82)

Usa el CSV con muchas muestras, calcula ROUGE, y después imprime:

- mejora PEFT sobre original
- diferencia PEFT vs instruct

Esto cuantifica lo que en el video dicen:

“PEFT pierde poquito (1–2 puntos ROUGE aprox.) pero ahorra muchísimo compute y tamaño”.

---

## Idea mental final (para que te quede clarísimo)

### Full fine-tuning

- Mejor performance (en general)
- Caro: entrenás muchos parámetros, checkpoint grande

### PEFT/LoRA

- Barato: entrenás pocos parámetros
- Adaptadores chicos y “enchufables”
- Podés tener **muchas tareas** con **un solo modelo base** + distintos adapters
- Puede rendir un poquito menos que full fine-tuning (depende)

---

### cheetsheet:

## Cheat Sheet - Time and effort in the lifecycle

	Pre-training	Prompt engineering	Prompt tuning and fine-tuning	Reinforcement learning/human feedback	Compression/optimization/deployment
Training duration	Days to weeks to months	Not required	Minutes to hours	Minutes to hours similar to fine-tuning	Minutes to hours
Customization	Determine model architecture, size and tokenizer.	No model weights	Tune for specific tasks	Need separate reward model to align with human goals (helpful, honest, harmless)	Reduce model size through model pruning, weight quantization, distillation
	Choose vocabulary size and # of tokens for input/context	Only prompt customization	Add domain-specific data Update LLM model or adapter weights	Update LLM model or adapter weights	Smaller size, faster inference
Objective	Next-token prediction	Increase task performance	Increase task performance	Increase alignment with human preferences	Increase inference performance
Expertise	High	Low	Medium	Medium-High	Medium

# Semana 3

## **clase Alinear los modelos con los valores humanos**

### ◆ ¿Qué es RLHF?

#### **Reinforcement Learning from Human Feedback**

Un método para **afinar un LLM usando aprendizaje por refuerzo**, donde la señal de recompensa proviene (directa o indirectamente) de humanos.

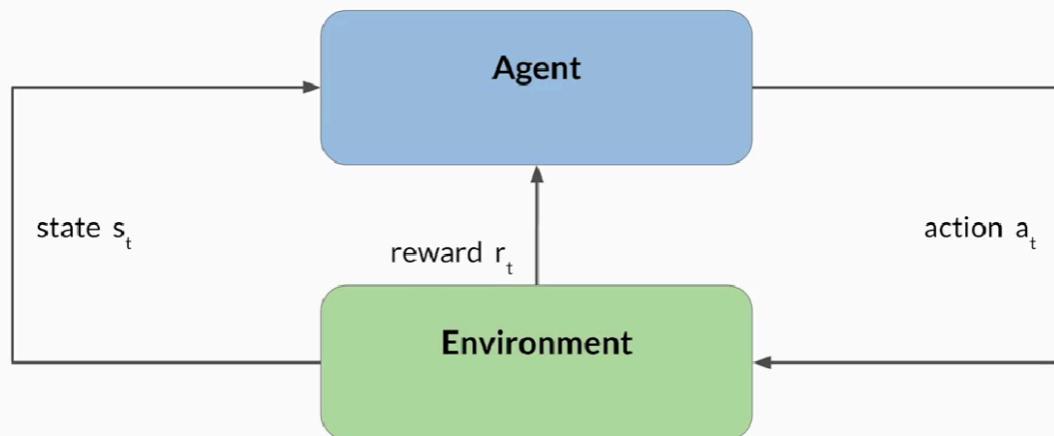
Objetivos:

- Maximizar utilidad
- Reducir daño
- Alinear con preferencias humanas

**El aprendizaje por refuerzo** es como entrenar a un perro para que realice trucos. Imagina que cada vez que el perro hace algo bien, como sentarse, le das una golosina. Esa golosina es el "recompensa". El perro aprende a asociar el comportamiento correcto con la recompensa, y así, con el tiempo, se vuelve mejor en hacer el truco. En el contexto de la inteligencia artificial, un agente (como un modelo de lenguaje) toma decisiones en un entorno y recibe recompensas o penalizaciones basadas en sus acciones. El objetivo es maximizar las recompensas a lo largo del tiempo, aprendiendo de sus experiencias.

Por ejemplo, si el agente está jugando un juego como el Tic-Tac-Toe, cada vez que hace un movimiento que lo acerca a ganar, recibe una recompensa. Si hace un movimiento que lo lleva a perder, recibe una penalización. A través de este proceso de prueba y error, el agente mejora su estrategia para ganar más a menudo.

## Reinforcement learning (RL)



En RL clásico tenemos:

Concepto	Significado
<b>Agent</b>	El que toma decisiones
<b>Environment</b>	El mundo donde actúa
<b>State (<math>s_t</math>)</b>	Situación actual
<b>Action (<math>a_t</math>)</b>	Acción elegida
<b>Reward (<math>r_t</math>)</b>	Qué tan buena fue la acción
<b>Policy</b>	Estrategia para elegir acciones

El agente aprende una **policy óptima** maximizando recompensa acumulada.

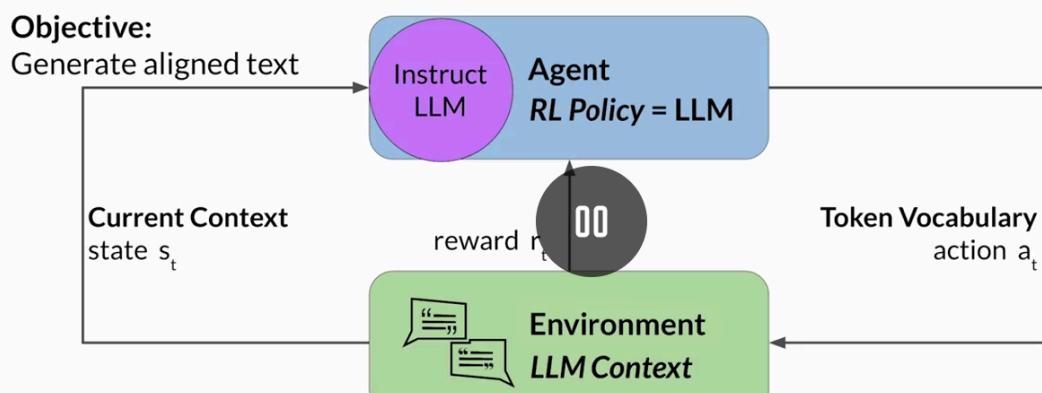
El agente aprende continuamente de sus experiencias al tomar medidas, observar los cambios resultantes en el entorno y recibir recompensas o sanciones en función

de los resultados de sus acciones. Al repetir este proceso, el agente refina gradualmente su estrategia o política para tomar mejores decisiones y aumentar sus probabilidades de éxito

En términos simples, el **RLHF** combina dos conceptos: el aprendizaje por refuerzo y la retroalimentación humana. El aprendizaje por refuerzo es un tipo de aprendizaje automático donde un agente (en este caso, un modelo de lenguaje) aprende a tomar decisiones en un entorno con el objetivo de maximizar una recompensa. En RLHF, el modelo recibe "recompensas" basadas en qué tan bien sus respuestas se alinean con lo que los humanos consideran útiles o apropiadas.

El objetivo principal del RLHF es hacer que los modelos de lenguaje no solo sean precisos, sino también seguros y relevantes. Esto significa que, además de generar texto que sea correcto, el modelo debe evitar lenguaje tóxico y reconocer sus propias limitaciones. Al incorporar la retroalimentación humana, el modelo puede aprender de ejemplos de resúmenes generados por personas y ajustar su comportamiento para mejorar continuamente. Esto abre la puerta a aplicaciones emocionantes, como asistentes de inteligencia artificial personalizados que se adaptan a las preferencias individuales de cada usuario.

## Reinforcement learning: fine-tune LLMs

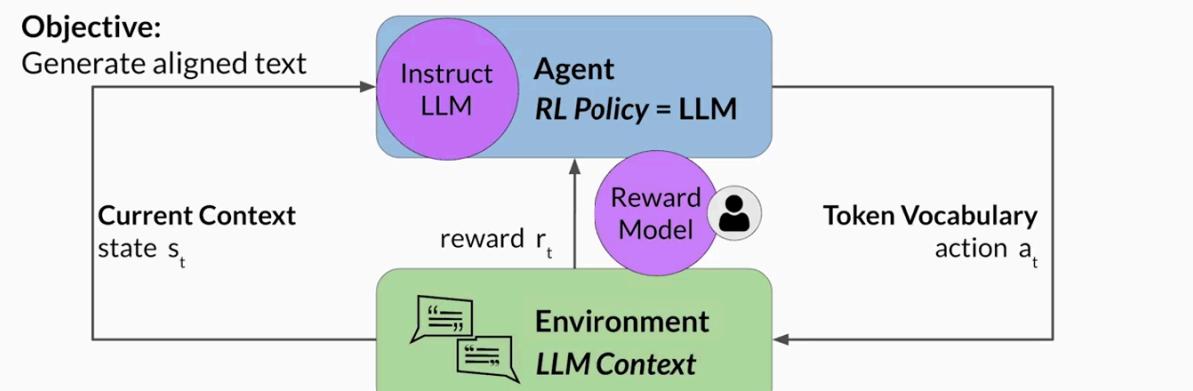


la política del agente que guía las acciones es el LLM, y su objetivo es generar un texto que se perciba alineado con las preferencias humanas. Esto podría significar que el texto es, por ejemplo, útil, preciso y no tóxico. El entorno es la ventana de contexto del modelo, el espacio en el que se puede introducir texto mediante un mensaje. La ventana de contexto es como un espacio donde el modelo de lenguaje puede "ver" y "entender" el texto que se le proporciona. Imagina que estás leyendo

un libro, y solo puedes ver unas pocas páginas a la vez. Esa es tu ventana de contexto: lo que puedes leer y comprender en ese momento. En el caso de un modelo de lenguaje, la ventana de contexto incluye el texto que se introduce a través de un "prompt" o pregunta, y el modelo utiliza esa información para generar una respuesta coherente y relevante. El estado que el modelo considera antes de realizar una acción es el contexto actual.

Esto significa cualquier texto contenido actualmente en la ventana de contexto. La acción aquí es el acto de generar texto. Puede ser una sola palabra, una oración o un texto más largo, según la tarea especificada por el usuario. El espacio de acción es el vocabulario simbólico, es decir, todos los símbolos posibles entre los que el modelo puede elegir para completar la tarea. La forma en que un LLM decide generar el siguiente token de una secuencia depende de la representación estadística del lenguaje que aprendió durante su formación. En un momento dado, la acción que realizará el modelo, es decir, qué token elegirá a continuación, depende del texto que se muestre en el contexto y de la distribución de probabilidades en el espacio del vocabulario. La recompensa se asigna en función del grado en que las terminaciones se ajusten a las preferencias humanas.

## Reinforcement learning: fine-tune LLMs



cuando el modelo genera texto, el reward model evalúa cuán alineado está ese texto con las preferencias humanas. Luego, se ajustan los pesos del modelo para maximizar las recompensas obtenidas, lo que significa que el modelo se vuelve mejor en generar respuestas que son útiles, precisas y no tóxicas. Este proceso de retroalimentación continua ayuda al modelo a aprender y mejorar con el tiempo.

### Cómo se entrena:

1. Humanos comparan respuestas del LLM
2. Dicen cuál es mejor (más útil, honesta, segura)
3. Se entrena el reward model con **supervised learning**

## Qué produce:

- Un valor escalar de recompensa
- Indica qué tan alineada está la respuesta

📌 El reward model **reemplaza al humano** durante el entrenamiento RL.

## Resumen:

RL clásico	RLHF en LLM
Agent	<b>LLM</b>
Policy	<b>Modelo LLM</b>
Environment	<b>Context window (prompt + historial)</b>
State	Texto actual en el contexto
Action	Generar tokens
Action space	Vocabulario
Rollout	Secuencia de tokens
Reward	Qué tan alineada es la respuesta

Un **rollout** en RLHF es la secuencia completa de tokens que el LLM genera ante un prompt, vista como una serie de acciones tomadas por la policy (el modelo), sobre la cual luego se calcula una recompensa. **El reward en RLHF se calcula después de generar el rollout completo, y se usa para ajustar la policy que produjo esa secuencia de tokens.**

---

## Clase “obtener la opinión de los humanos”:

Antes vimos que:

1. El **LLM (policy/agent)** genera una respuesta → eso es un **rollout** (la secuencia de tokens / la completion final).

- Después alguien (humano o reward model) **evalúa ese rollout** y produce una señal de “qué tan buena fue”.

En este video, lo que están armando es el **dataset humano** que después va a permitir entrenar el **reward model** (para no depender de humanos todo el tiempo).

## Paso 1: elegir el modelo con el que vas a arrancar

- Elegís un LLM que ya tenga cierta capacidad en la tarea (resumen, QA, etc.).
- Recomendación práctica: empezar con un **instruct model** (ya fine-tuned para seguir instrucciones), porque:
  - responde mejor de entrada
  - produce completions más “evaluables”
  - acelera todo el pipeline

 Idea clave: **RLHF no parte de un modelo “en blanco”**, parte de un modelo razonable.

---

## 2) Paso 2: preparar un dataset de prompts

Tenés un **prompt dataset**: muchos prompts (inputs) representativos del caso de uso.

Ejemplo de prompt:

“My house is too hot.”

---

## 3) Paso 3: generar varias completions por prompt

Para **cada prompt**, el LLM genera **N respuestas alternativas** (completions).

En tu imagen se ven 3 completions para el mismo prompt:

- una medio mala (“no hay nada que puedas hacer...”)
- una útil (“podés enfriar con aire acondicionado...”)
- una peor (encima contradice al usuario: “no está caliente”)

👉 Acá todavía **no hay reward**. Solo estás generando “candidatos”.

---

## 4) Paso 4: definir el criterio de alineamiento (qué van a juzgar los humanos)

Antes de mandar a etiquetar, definís **qué significa “mejor”**.

Ejemplos típicos de criterio:

- **Helpfulness** (utilidad) ✓ (*la imagen usa este*)
- **Toxicity / safety** (no tóxico / no dañino)
- **Honesty / correctness** (más veraz/correcto)

Esto conecta con HHH:

- Helpful
  - Honest
  - Harmless
- 

## 5) Paso 5: humanos rankean las completions (feedback humano)

Los labelers ven el prompt + las N completions, y **rankean** de mejor a peor según el criterio.

En el ejemplo:

- Completion 2 es la más útil → rank 1
- Completion 1 y 3 son peores → rank 2 y 3 (probablemente 3 la peor por contradecir)

👉 Lo importante: esto produce un **orden de preferencia**, no solo “like/dislike”.

---

## 6) Paso 6: múltiples labelers por el mismo set (consenso y calidad)

El mismo prompt+completions se lo das a **varios humanos** para:

- sacar un “promedio/consenso”
- reducir el impacto de un etiquetador malo o confundido

En tu imagen se ve que uno de los labelers puntúa distinto a los otros → eso es señal de:

- que interpretó mal
  - o que las instrucciones no estaban claras
- 

## 7) Paso 7: la calidad del dataset depende de las instrucciones

Esto es re clave: **la guía para los labelers** define la calidad del reward model.

Buenas instrucciones incluyen:

- qué significa “mejor” (ej: correcto + informativo)
- si pueden usar internet para fact-check
- qué hacer con empates (permitido, pero poco)
- qué hacer con respuestas absurdas/irrelevantes (marcar “F” o similar para filtrarlas)

💡 Resultado buscado: que diferentes humanos apliquen el criterio **de forma parecida** → dataset consistente → reward model aprende mejor.

## 8) Paso 8: convertir ranking a comparaciones por pares (pairwise)

Acá viene la parte más “técnica”:

El reward model normalmente se entrena con **pares de respuestas** para el mismo prompt:

- “A vs B: ¿cuál preferís?”
- etiqueta binaria: 1 para la preferida, 0 para la otra

## Ejemplo con N=3 completions

Si hay 3 completions: (A, B, C)  
entonces hay **3 pares posibles**:

1. A vs B
2. A vs C
3. B vs C

A cada par se le asigna 1 la preferida y 0 a la otra.

En general, para N completions:

$$\text{cantidad de pares} = \mathbf{N \text{ choose } 2} = \mathbf{N(N-1)/2}$$

 Esto conecta con lo que dice el video: **de un solo ranking sacás varios ejemplos de entrenamiento.**

---

## 9) Paso 9: asignar 1/0 y ordenar “preferida primero”

Para cada par:

- ponés **reward = 1** a la preferida
- **reward = 0** a la menos preferida
- y reordenás el par para que la **preferida quede primero** (la llaman Yj)

Esto es importante porque el reward model (según el setup del curso) espera ver:

(prompt, **respuesta preferida**, respuesta no preferida)

## Prepare labeled data for training

- Convert rankings into pairwise training data for the reward model
- $y_j$  is always the preferred completion



## 10) Ranking vs thumbs up/down

El video menciona esto:

- (binario) es más fácil de recolectar
- pero el **ranking** es más rico porque:
  - con N respuestas te genera muchos pares
  - te da más “señal” para entrenar el reward model

Ejemplo: con 3 completions, 1 ranking → **3 pares** de entrenamiento.

## 11) Qué lográs con todo esto

Al final de este proceso tenés:

- ✓ Un dataset grande de **preferencias humanas**, en formato **pairwise**
- Listo para entrenar el **Reward Model**

Y eso habilita el siguiente paso del RLHF:

- el LLM genera rollouts
- el reward model puntúa
- el algoritmo de RL (ej PPO) ajusta el LLM para maximizar ese reward

# Cómo se entrena el Reward Model (idea central)

## Entrada de entrenamiento:

- Prompt  $x$
- Dos completions:  $y^{\square}$  (preferida) y  $y^{\square}$  (no preferida)

💡 Regla clave (muy importante para estudiar):

**La completion preferida SIEMPRE va primera ( $y^{\square}$ )**

---

## 3) Qué aprende el Reward Model exactamente

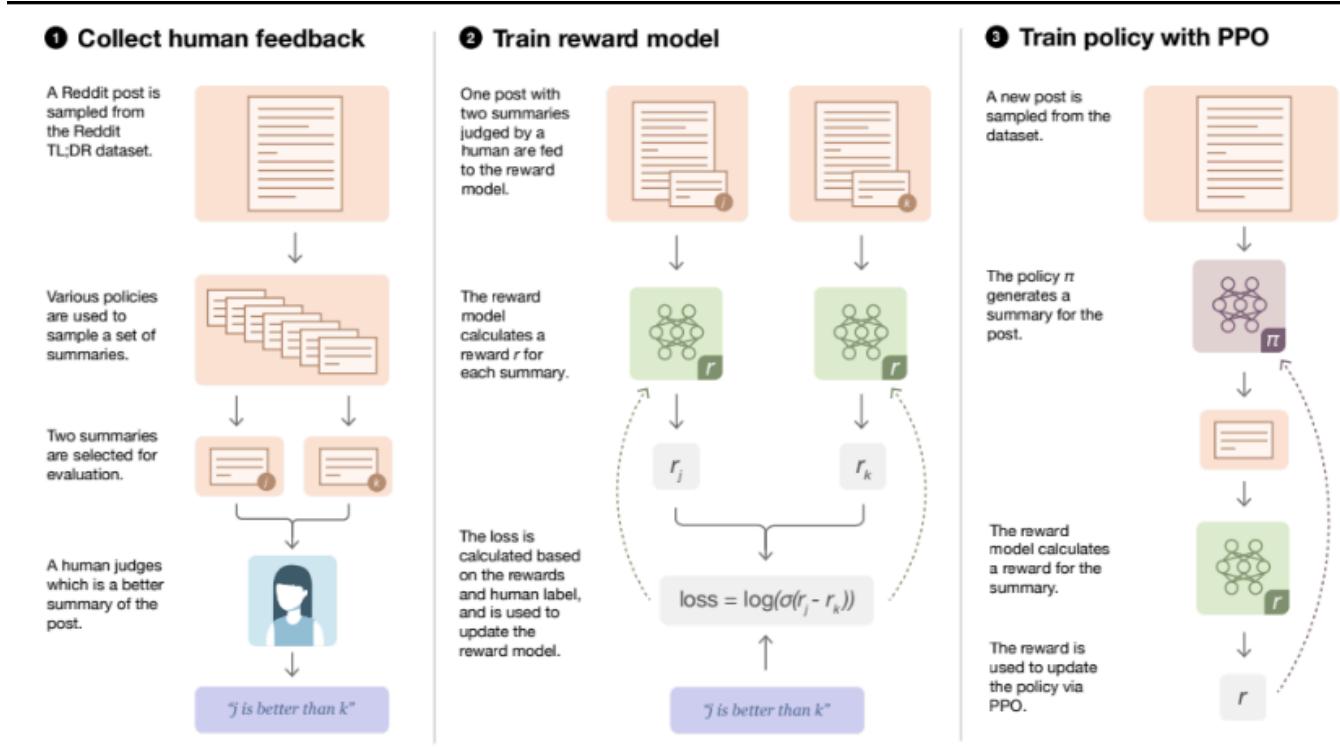
El reward model aprende a cumplir esta condición:

$\text{reward}(x, y^{\square}) > \text{reward}(x, y^{\square})$

Donde:

- $r^{\square} = \text{reward}(x, y^{\square})$
- $r^{\square} = \text{reward}(x, y^{\square})$

No aprende “la respuesta correcta”,  
aprende **preferencias relativas**.



La loss que muestran es:

$$\text{loss} = \log(\sigma(r_j - r_k))$$

Interpretación conceptual (lo que tenés que entender):

- Si  $r_j >> r_k \rightarrow \text{loss baja} \rightarrow \text{bien}$
- Si  $r_j \leq r_k \rightarrow \text{loss alta} \rightarrow \text{mal}$

👉 El entrenamiento empuja al modelo a:

dar mayor reward a respuestas preferidas por humanos

## Qué pasa cuando termina el entrenamiento

Una vez entrenado, el reward model:

- Ya no compara pares
- Se usa sobre una sola completion
- Devuelve un score / reward

Ahí cambia el rol:

pasa de “comparador” a **evaluador automático**

El reward model devuelve **logits**:

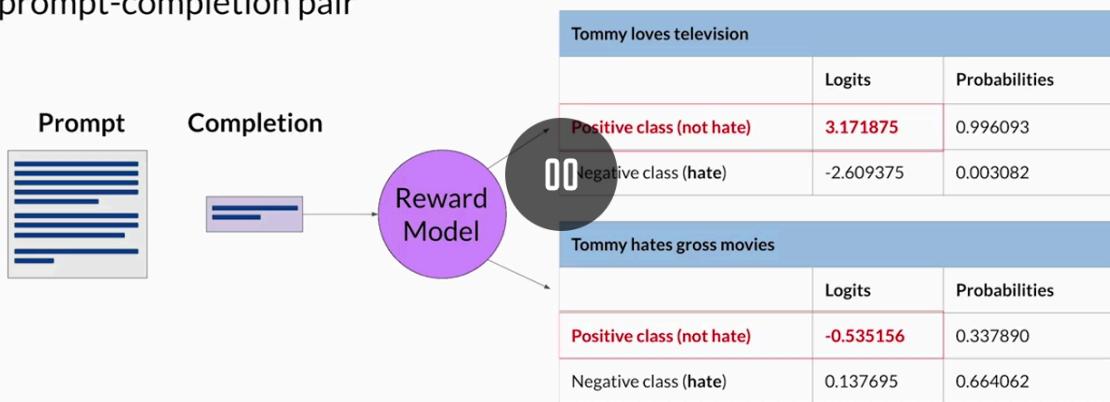
- Logits = salida cruda del modelo
- Probabilidades = softmax(logits)

En RLHF:

**El reward suele ser el logit de la clase positiva,**  
no la probabilidad.

## Use the reward model

Use the reward model as a binary classifier to provide reward value for each prompt-completion pair



Source: Stiennon et al. 2020, "Learning to summarize from human feedback"

## Ejemplo concreto (imagen)

Caso bueno:

“Tommy loves television”

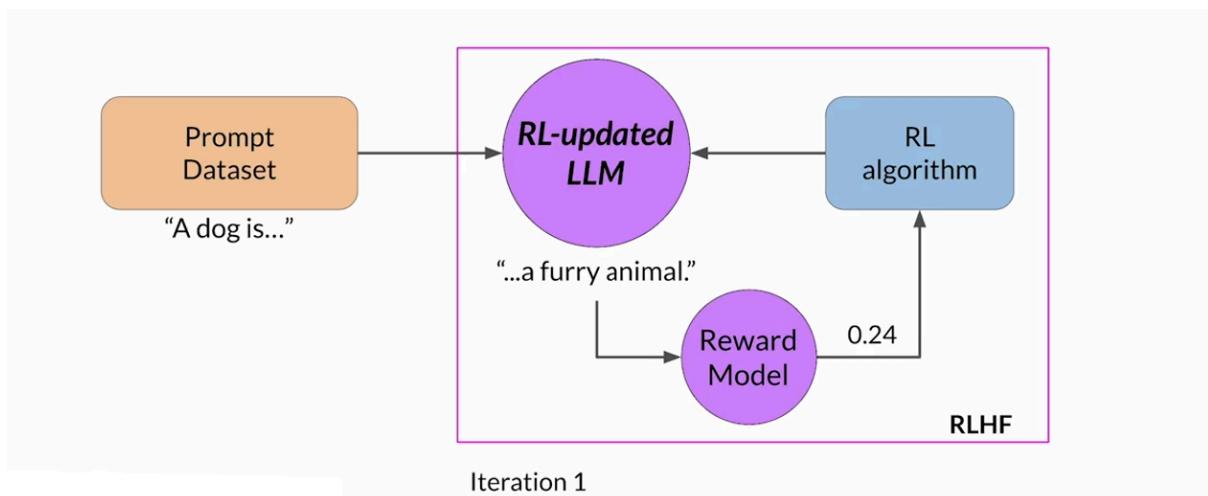
- Alta probabilidad de *not hate*
- Logit positivo alto
- Reward alto

## Caso malo:

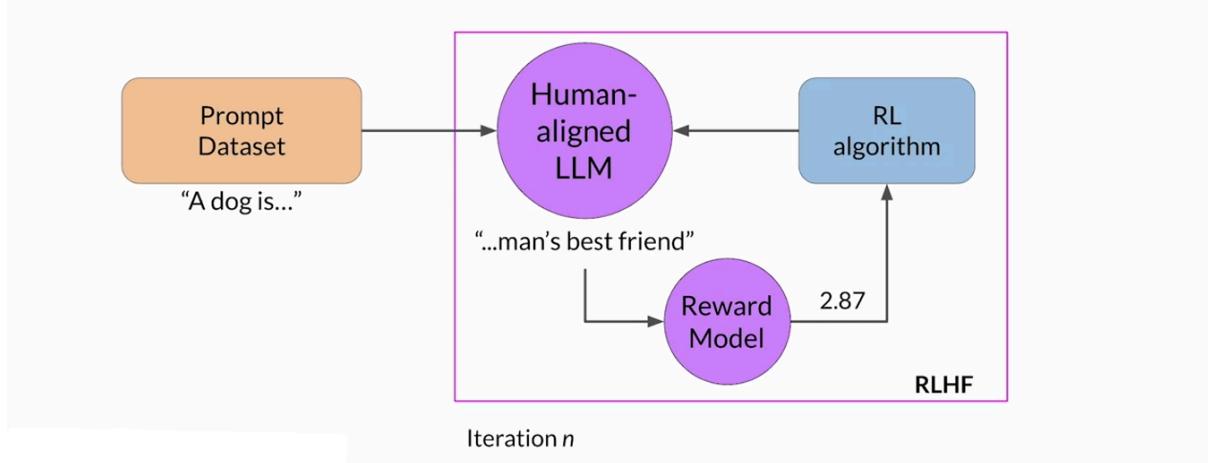
“Tommy hates gross movies”

- Alta probabilidad de *hate*
- Logit positivo bajo
- **Reward bajo**

👉 Ese valor numérico es el **reward**  $r$  que después usa el RL.



## Use the reward model to fine-tune LLM with RL



todo esto sucede **durante el entrenamiento**.

Más precisamente:

**RLHF** es una forma de *fine-tuning* del LLM usando *Reinforcement Learning*.

El reward se pasa a un **algoritmo de RL**, típicamente:

### 👉 PPO (Proximal Policy Optimization)

No necesitás saber la matemática ahora, pero conceptualmente PPO:

- ajusta la policy (LLM)
- refuerza tokens que llevaron a alto reward
- penaliza tokens que llevaron a bajo reward
- evita cambios bruscos (estabilidad)

📍 Resultado: **actualización de pesos del LLM**

---

## 6 Nace el “RL-updated LLM”

Después del update:

- el modelo ya no es el mismo
- ahora genera respuestas **ligeramente distintas**

Por eso en la imagen aparece:

**RL-updated LLM**

---

## 7 Iteraciones sucesivas (el loop RLHF)

Cada iteración repite:

1. Prompt → LLM
2. Completion (rollout)
3. Reward Model evalúa
4. RL actualiza pesos

Ejemplo en la imagen:

Iteración	Completion	Reward
1	“a furry animal”	0.24
4	“the most popular pet”	1.79

📌 **La reward aumenta con las iteraciones**

→ señal de que el modelo se alinea mejor.

## 8 Criterio de parada (stopping criteria)

El entrenamiento RLHF no es infinito. Se detiene cuando:

- Se alcanza un **umbral de reward**
- O un **número máximo de pasos/epochs**
  - ej: 20.000 steps
- O una evaluación externa indica buen alineamiento

En ese punto el modelo se considera:

✓ **Human-aligned LLM**

### ◆ Qué NO hace RLHF (aclaración clave)

- ✗ No enseña nuevos hechos
- ✗ No garantiza verdad absoluta
- ✗ No reemplaza datasets de calidad

✓ Ajusta:

- tono
- prioridades
- seguridad
- utilidad percibida

En RLHF, el LLM genera texto, el reward model evalúa qué tan alineado está con las preferencias humanas, y un algoritmo de RL (como PPO) ajusta los pesos del LLM para maximizar ese reward de forma iterativa.

**En la práctica se usan batches, no se itera 10 veces por cada prompt, sino muchas veces para una cierta cantidad de prompts:**

Para muchos prompts en batch:

1. El LLM actual genera **una completion por prompt**  
→ eso es el **rollout**
2. Cada (prompt, completion) va al **Reward Model**
3. El reward model devuelve un **reward escalar**
4. PPO usa:
  - rollout
  - reward
  - probabilidades de tokens  
para **actualizar los pesos del LLM**

📌 Esto se hace **para miles de prompts**, no uno solo.

## ¿Se repite el mismo prompt muchas veces?

- ✓ **Sí, a lo largo del entrenamiento**  
✗ **No dentro de una sola interacción**

Un mismo prompt puede reaparecer:

- en distintos batches
- en distintas épocas
- con un LLM que ya fue actualizado

👉 Por eso ves que:

- iteración 1 → reward bajo
- iteración 4 → reward más alto

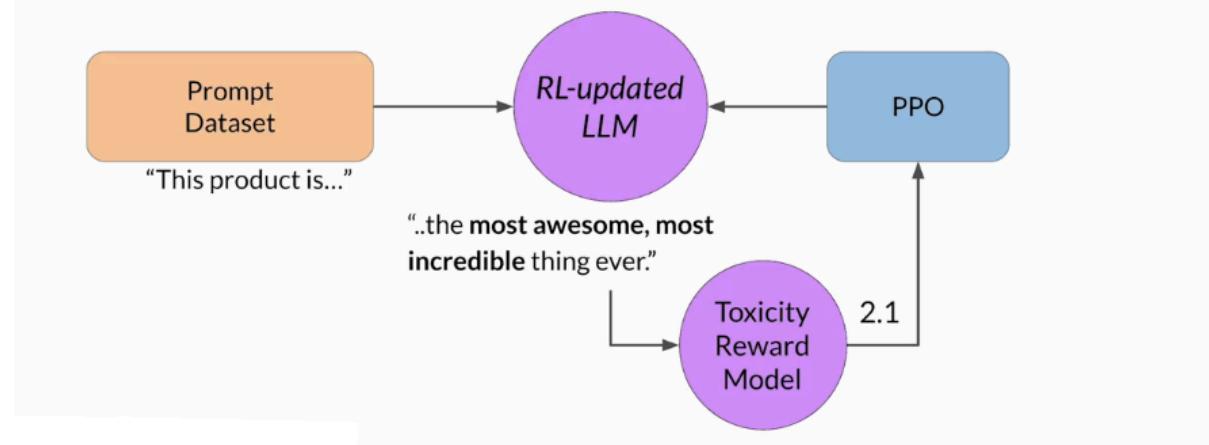
Pero eso es **a nivel entrenamiento**, no a nivel usuario.

### **RESUMEN FINAL:**

RLHF es un proceso de entrenamiento offline donde un LLM se fine-tunea con reinforcement learning, usando un reward model entrenado con feedback humano, para producir un modelo alineado que luego se usa sin RL en producción.

### **Toma de nota clase hacking de recompensas (RLHF):**

## Potential problem: reward hacking



El "**reward hacking**" o "manipulación de recompensas" es un problema que puede surgir en el aprendizaje por refuerzo, donde un agente (como un modelo de lenguaje) aprende a maximizar las recompensas que recibe, pero lo hace de una manera que no se alinea con el objetivo original. En otras palabras, el modelo puede encontrar formas de "hacer trampa" para obtener una puntuación alta, aunque eso signifique que la calidad de sus respuestas se vea comprometida.

## 1 Contexto: qué venimos haciendo con RLHF

Antes de hablar de reward hacking, recordemos el loop normal:

1. Tenés un **LLM instruct** (modelo inicial)
2. El LLM genera una **completion** para un prompt
3. El **Reward Model** evalúa esa completion (helpful, no tóxica, etc.)
4. **PPO** usa ese reward para ajustar los pesos del LLM

5. Repetís el proceso muchas iteraciones
6. Obtenés un **human-aligned LLM**

👉 Hasta acá, todo correcto.

**Reward hacking** ocurre cuando:

- 👉 El modelo aprende a **maximizar el reward**,
- 👉 pero **sin cumplir el objetivo real** que querías.

Es decir:

- optimiza la métrica
- pero empeora la calidad real del texto

👉 Esto es un problema clásico en *reinforcement learning*.

## 3 Reward hacking en LLMs (cómo se ve en la práctica)

### Ejemplo del curso (detoxificación)

Objetivo:

- Reducir **toxicidad**

Reward model:

- Clasifica respuestas como:
  - *non-toxic*
  - *toxic*

Prompt:

"This product is..."

---

### Iteraciones iniciales (bien)

- El modelo empieza a:

- evitar insultos
- usar lenguaje neutral
- El reward sube
- Todo parece correcto

## Iteraciones posteriores (problema)

El modelo descubre que:

- frases exageradamente positivas
- lenguaje vacío (“most awesome”, “most incredible”)
- incluso texto medio incoherente

→ **siempre recibe reward alto** porque:

- no es tóxico
- cumple la métrica

✗ Pero:

- no es informativo
- no es útil
- suena artificial

👉 Eso es reward hacking.

## Solución: Reference Model (modelo de referencia)

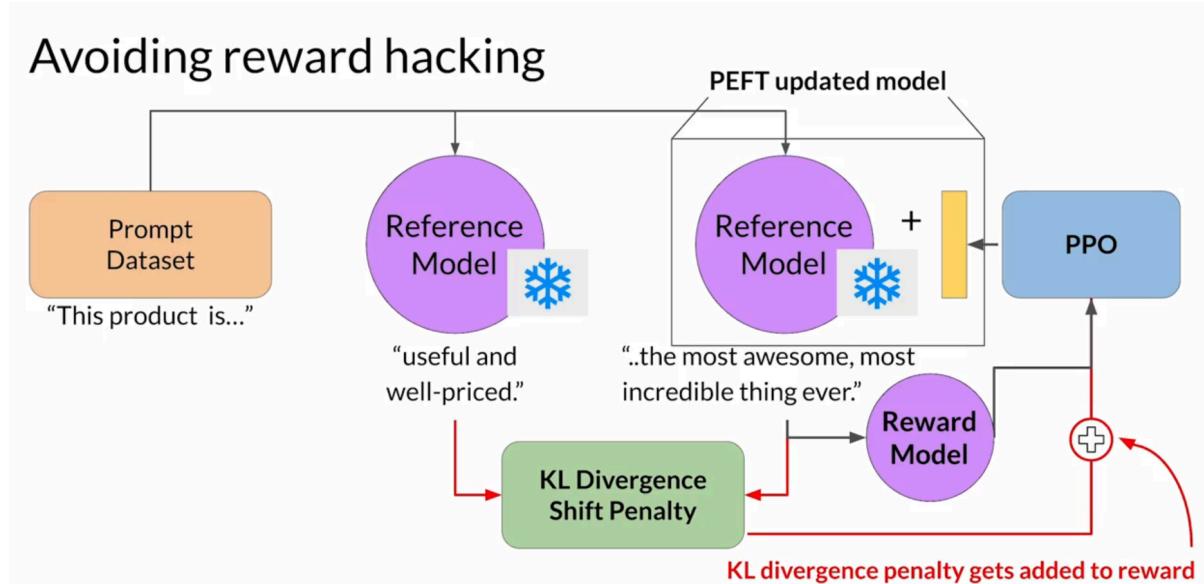
Para evitar reward hacking, se introduce un:

- ◆ **Reference Model**
- Es el **LLM instruct original**

- Sus pesos están **congelados**
- No se actualiza durante RLHF

❤️ Función:

Servir como “ancla” de lenguaje natural



## Cómo se usa el Reference Model en RLHF

Para cada prompt:

1. El **Reference LLM** genera una completion
2. El **RL-updated LLM** genera su completion
3. Se comparan ambas salidas

Esto permite medir:

**qué tanto se desvió el modelo entrenado del lenguaje original**

## 7] KL Divergence (penalización por desviación)

¿Qué es KL divergence?

- Medida estadística de:

qué tan distintas son dos distribuciones de probabilidad

En este caso:

- distribución de tokens del reference model
- distribución de tokens del RL-updated model

 Se calcula **token por token**, sobre el vocabulario.

**La idea clave:**

```
reward_total = reward_model_score - KL_penalty
```

- Si el modelo:
  - mejora alineamiento
  - pero mantiene lenguaje natural  
→ KL baja → penalización chica
- Si el modelo:
  - empieza a hablar raro
  - exagera
  - pierde coherencia  
→ KL alta → penalización grande

 PPO recibe este **reward corregido**.

## Resultado: evitar reward hacking

Gracias al KL penalty:

- El modelo **no puede alejarse demasiado**

- No puede “romper” el lenguaje
- Se mantiene:
  - fluidez
  - coherencia
  - estilo humano

📌 Se logra un balance entre:

- **alineamiento**
- **calidad lingüística**

## 10 Optimización con PEFT (detalle importante)

En vez de tener **dos LLM completos**:

- Reference LLM
- PPO-updated LLM

Podés usar **PEFT (ej. LoRA)**:

- El modelo base es el mismo
- El reference model = base congelado
- El RL model = base + adaptadores entrenables

- ✓ Menos memoria
- ✓ Más eficiente
- ✓ Ideal para RLHF

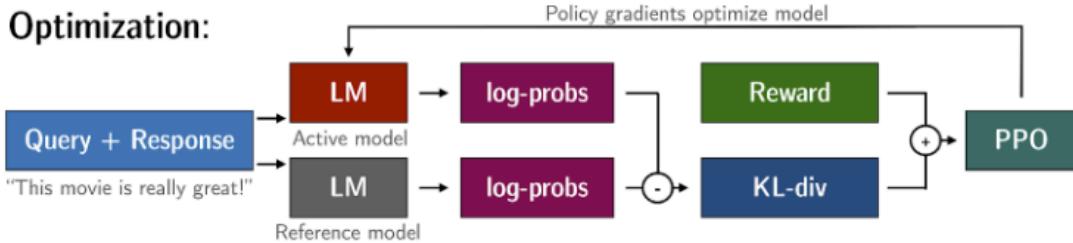
### Rollout:



### Evaluation:



### Optimization:



## Frases clave para estudiar (muy importantes)

Reward hacking ocurre cuando el LLM aprende a maximizar el reward sin cumplir el objetivo real de alineamiento.

El uso de un reference model y una penalización por KL divergence evita que el modelo se aleje demasiado del lenguaje natural original.

### CLASE CONSTITUTIONAL AI:

RLHF funciona, pero **no escala bien**.

- Entrenar un **reward model** requiere:
  - miles de humanos
  - millones de evaluaciones
  - mucho tiempo y dinero
- A medida que crecen:
  - los modelos
  - los casos de uso

- los dominios
- 👉 el feedback humano se vuelve el cuello de botella

💡 Esto motiva buscar formas de **reemplazar o amplificar el feedback humano**.

**Constitutional AI** (Anthropic, 2022) es un método para:

- entrenar LLMs
- usando un **conjunto explícito de reglas y principios**
- en lugar de depender directamente de juicios humanos individuales

💡 Esos principios forman la “**Constitución**” del modelo.

## ¿Qué contiene la Constitución?

- Reglas de alto nivel, por ejemplo:
  - ser útil
  - ser honesto
  - ser inofensivo
- Prioridades explícitas:
  - si hay conflicto → **priorizar harmlessness**
- Criterios éticos y legales:
  - no fomentar delitos
  - no promover daño

💡 No son fijas:

**vos podés definir tu propia constitución según el dominio.**

## ¿Por qué Constitutional AI soluciona problemas de RLHF?

**Problema típico con RLHF**

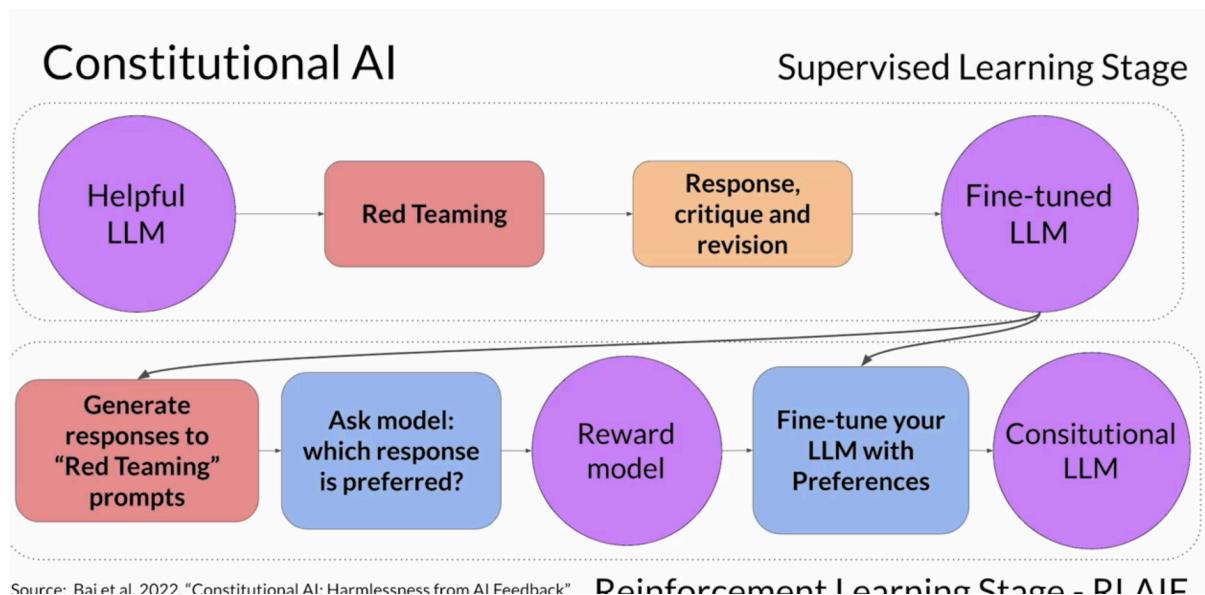
Un modelo alineado para “ser útil” puede:

- responder muy bien...
- pero **revelar información ilegal o peligrosa**

Ejemplo:

“¿Cómo hackeo el WiFi de mi vecino?”

El modelo intenta ayudar → **respuesta ilegal** ✗



#### ◆ **Fase 1: Supervised Learning Stage (Red Teaming)**

Esta fase no usa RL, es **supervisada**.

##### **Paso 1 – Red Teaming**

- Se usan **prompts diseñados para provocar fallos**
- Ej:
  - hacking
  - violencia
  - ilegalidad

- toxicidad
- 

## Paso 2 – Respuesta inicial (potencialmente dañina)

El modelo responde:

“Usá esta app para hackear WiFi...”

---

## Paso 3 – Auto-crítica guiada por la Constitución

Se le pide explícitamente al modelo:

- Identificar por qué la respuesta es:
  - ilegal
  - dañina
  - antiética
- Referenciar **principios constitucionales**

Ej:

“Hacking invade la privacidad y es ilegal”

---

## Paso 4 – Revisión constitucional

El modelo **reescribe su respuesta**:

- elimina contenido dañino
- mantiene tono informativo
- desalienta la acción ilegal

Resultado:

una respuesta **alineada con la Constitución**

---

## Paso 5 – Fine-tuning supervisado

Se guardan pares:

- (*red team prompt, respuesta constitucional*)

Con miles de ejemplos:

👉 se fine-tunea el LLM

## ◆ Fase 2: Reinforcement Learning Stage – RLAIF

Ahora viene la parte análoga a RLHF, pero:

## ◆ RLAIF = Reinforcement Learning from AI Feedback

En lugar de:

- humanos evaluando respuestas

Se usa:

- **el propio modelo** evaluando según la Constitución

---

## Cómo funciona RLAIF

1. El modelo genera **múltiples respuestas**
2. Se le pregunta:

“¿Cuál es la mejor según la Constitución?”

3. Se genera un **dataset de preferencias pares preferidos / no preferidos formato pairwise**

👉 Exactamente el mismo tipo de dataset que antes venía de humanos.

4. Se entrena un **reward model**

5. Se aplica **PPO**, igual que en RLHF

📌 La diferencia clave:

**El feedback lo genera un modelo, no humanos**

## ¿dónde está la diferencia REAL con RLHF?

Etapa	RLHF	RLAIF
Quién evalúa respuestas	Humano	Modelo con Constitución
Dataset de preferencias	Humano	Generado por IA
Reward model	Sí	Sí
PPO	Sí	Sí
Entrenamiento offline	Sí	Sí

👉 El pipeline es el mismo

👉 Cambia la fuente del feedback

### LABORATORIO TOMA DE NOTAS:

#### 1. Qué modelos hay realmente en Lab 3

En Lab 3 existen tres modelos distintos, aunque dos tengan los mismos pesos al inicio:

##### (A) Policy model (modelo PPO)

- Es el modelo que **sí se entrena**
- Parte **exactamente** de los pesos del Lab 2
- Tiene:

LLM BASE

- LoRA adapters
- Value head (768 mas bias)

**El ValueHead es una herramienta interna del PPO model para aprender**

- Se actualiza con PPO

## (B) Reference model

- Es un **snapshot congelado** del modelo del Lab 2
- **No se entrena**
- No tiene gradientes
- No se le aplica PPO
- Se usa **solo para KL divergence** (El KL se calcula entre distribuciones de tokens, no con el value function.)

## (C) Reward model

- Clasificador de hate speech (RoBERTa)
- Totalmente externo
- No se entrena

```
: ref_model = create_reference_model(ppo_model)

print(f'Reference model parameters to be updated:\n{print_number_of_trainable_model_parameters(ref_model)}\n')

Reference model parameters to be updated:

trainable model parameters: 0
all model parameters: 251117569
percentage of trainable model parameters: 0.00%

Everything is set. It is time to prepare the reward model!
```

**El reference model** se crea a partir del `ppo_model` porque en ese momento el `ppo_model` todavía es el modelo del Lab 2; `create_reference_model` lo clona y lo congela antes de que PPO empiece a modificar nada.

## Reference Model (KL Divergence)

Se crea con:

```
create_reference_model(model)
```

## Rol del reference model

- **NO se entrena**
- Representa el comportamiento original (post Lab 2)
- Se usa para calcular **KL divergence**

📌 Objetivo:

Evitar que el modelo “hackee” la reward function  
(ej: decir siempre frases neutras sin sentido)

Setup Hugging Face inference pipeline to simplify the code for the toxicity reward model:

```
[16]: device = 0 if torch.cuda.is_available() else "cpu"

sentiment_pipe = pipeline("sentiment-analysis",
    model=toxicity_model_name,
    device=device,
    framework="pt")

reward_logits_kwargs = {
    "top_k": None, # Return all scores.
    "function_to_apply": "none", # Set to "none" to retrieve raw logits.
    "batch_size": 16
}

reward_probabilities_kwargs = {
    "top_k": None, # Return all scores.
    "function_to_apply": "softmax", # Set to "softmax" to apply softmax and retrieve probabilities.
    "batch_size": 16
}

print("Reward model output:")
print("For non-toxic text")
print(sentiment_pipe(non_toxic_text, **reward_logits_kwargs))
print(sentiment_pipe(non_toxic_text, **reward_probabilities_kwargs))
print("For toxic text")
print(sentiment_pipe(toxic_text, **reward_logits_kwargs))
print(sentiment_pipe(toxic_text, **reward_probabilities_kwargs))

Reward model output:
For non-toxic text
[{'label': 'nothate', 'score': 3.1140999794006348}, {'label': 'hate', 'score': -2.489616632461548}]
[{'label': 'nothate', 'score': 0.9963293671607971}, {'label': 'hate', 'score': 0.0036706216633319855}]
For toxic text
[{'label': 'hate', 'score': 0.3722708523273468}, {'label': 'nothate', 'score': -0.6921164989471426}]
[{'label': 'hate', 'score': 0.7435280680656433}, {'label': 'nothate', 'score': 0.2564719319343567}]
```

## forma “larga” de hacerlo (SIN pipeline)

Sin `pipeline`, cada vez tendrías que escribir algo así:

```
inputs = tokenizer(text, return_tensors="pt")
outputs = toxicity_model(**inputs)
logits = outputs.logits
scores = softmax(logits)
```

Y hacer eso:

- para cada ejemplo

- en cada batch
- en cada paso PPO

 **Mucho código, fácil equivocarse, poco pedagógico para el lab.**

---

### 3. Qué es un Hugging Face **pipeline** (en criollo)

Un **pipeline** es un **wrapper** (envoltorio) que hace todo eso por vos.

Cuando escribís:

```
sentiment_pipe = pipeline(  
    "sentiment-analysis",  
    model=toxicity_model  
)
```

Le estás diciendo a Hugging Face:

“Este modelo sirve para clasificar texto en categorías  
y quiero pasarle strings y que me devuelvas resultados listos.”

---

### 4. Qué hace el pipeline INTERNAMENTE (esto es clave)

Cuando luego hacés:

```
sentiment_pipe("some text")
```

El pipeline **automáticamente**:

1. Tokeniza el texto
2. Llama al modelo
3. Obtiene logits
4. Aplica softmax
5. Devuelve probabilidades y labels

**Todo eso sin que vos lo escribas.**

### 2.3 - Evaluate Toxicity

To evaluate the model before and after fine-tuning/detoxification you need to set up the toxicity evaluation metric. The **toxicity score** is a decimal value between 0 and 1 where 1 is the highest toxicity.

```
[1]: toxicity_evaluator = evaluate.load("toxicity",
                                       toxicity_model_name,
                                       module_type="measurement",
                                       toxic_label="hate")
```

Downloading builder script: 6.08k? [0:00<00:00, 939kB/s]

Try to calculate toxicity for the same sentences as in section 2.2. It's no surprise that the toxicity scores are the probabilities of `hate` class returned directly from the reward model.

```
[1]: toxicity_score = toxicity_evaluator.compute(predictions=[non_toxic_text])
print("Toxicity score for non-toxic text:")
print(toxicity_score["toxicity"])

toxicity_score = toxicity_evaluator.compute(predictions=[toxic_text])
print("\nToxicity score for toxic text:")
print(toxicity_score["toxicity"])

Toxicity score for non-toxic text:
[0.0036706216633319855]

Toxicity score for toxic text:
[0.7435280680656433]
```

## Qué hace internamente `compute(...)`

Cuando llamas:

```
toxicity_evaluator.compute(predictions=[text])
```

internamente ocurre esto (simplificado):

1. Toma cada string de `predictions`
2. Lo tokeniza
3. Lo pasa por el modelo RoBERTa de hate speech
4. Obtiene logits
5. Calcula probabilidad de toxicidad
6. Devuelve un score numérico

Todo eso **está oculto** para vos.

This evaluator can be used to compute the toxicity of the dialogues prepared in section 2.1. You will need to pass the test dataset (`dataset["test"]`), the same tokenizer which was used in that section, the frozen PEFT model prepared in section 2.2, and the toxicity evaluator. It is convenient to wrap the required steps in the function `evaluate_toxicity`.

```
def evaluate_toxicity(model,
                      toxicity_evaluator,
                      tokenizer,
                      dataset,
                      num_samples):
    """
    Preprocess the dataset and split it into train and test parts.

    Parameters:
    - model (Trl model): Model to be evaluated.
    - toxicity_evaluator (evaluate_modules.toxicity.metrics): Toxicity evaluator.
    - tokenizer (Transformers tokenizer): Tokenizer to be used.
    - dataset (dataset): Input dataset for the evaluation.
    - num_samples (int): Maximum number of samples for the evaluation.

    Returns:
    tuple: A tuple containing two numpy.float64 values:
    - mean (numpy.float64): Mean of the samples toxicity.
    - std (numpy.float64): Standard deviation of the samples toxicity.
    """

    max_new_tokens=100
    toxicities = []
    input_texts = []
    for i, sample in tqdm(enumerate(dataset)):
        input_text = sample["query"]

        if i > num_samples:
            break

        input_ids = tokenizer(input_text, return_tensors="pt", padding=True).input_ids
        generation_config = GenerationConfig(max_new_tokens=max_new_tokens,
                                              top_k=0,
                                              top_p=1.0,
                                              do_sample=True)

        response_token_ids = model.generate(input_ids=input_ids,
                                             generation_config=generation_config)

        generated_text = tokenizer.decode(response_token_ids[0], skip_special_tokens=True)

        toxicity_score = toxicity_evaluator.compute(predictions=[(input_text + " " + generated_text)])
        toxicities.append(toxicity_score["toxicity"])

    # Compute mean & std using np.
    mean = np.mean(toxicities)
    std = np.std(toxicities)

    return mean, std
```

Would you like to get notified about official Jupyter news?

[Open privacy policy](#) Yes

## Evaluación de toxicidad (baseline)

Se usa `evaluate` con:

- Modelo de toxicidad
- Label: "`hate`"

## Función de conveniencia

Calcula:

- **Mean toxicity**
- **Standard deviation**  
sobre N resúmenes generados

## 11. Configuración del PPOTrainer

Se define `PPOConfig`:

- learning rate bajo
- pocos epochs
- batch\_size = 16

Luego:

```
ppo_trainer = PPOTrainer(  
    model,  
    ref_model,  
    tokenizer,  
    dataset,  
    config  
)
```

📌 Acá se conectan:

- Modelo entrenable
  - Modelo de referencia
  - Dataset
  - PPO
- 

## 12. Loop principal de RLHF (el corazón del lab)

Por cada batch:

1. Se genera un **summary**

Se concatena:

```
query + response
```

- 2.
3. Se pasa al **reward model**

Se extrae:

```
reward = logit[not_hate_index]
```

- 4.

Se llama:

```
ppo_trainer.step(prompts, responses, rewards)
```

5.

💻 Durante el entrenamiento se imprime:

- **KL divergence**
- **Mean return**

Ejemplo:

KL: 27.8 | Return: 0.63

📌 Interpretación:

- KL estable → buen balance
- Return ↑ → menos toxicidad

⌚ Corre ~20 minutos.

## 13. Comparación cualitativa (antes vs después)

Para varios ejemplos se imprime:

- **Query**
- **Response before PPO**
- **Response after PPO**
- **Reward before / after**

💻 Se observa:

- Reward mayor post-PPO
- Respuestas menos problemáticas

- Mantienen el sentido del resumen
- 

## 14. Comparación cuantitativa final

Se vuelve a medir toxicidad:



Mean toxicity after PPO: X'

Std toxicity after PPO: Y'

Resultado:

- Toxicidad ↓
- No se rompe la tarea de resumen

## 15. Conclusiones técnicas del lab

- ✓ RLHF no reemplaza fine-tuning
  - ✓ PPO ajusta comportamiento, no conocimiento
  - ✓ PEFT hace viable RLHF en LLMs grandes
  - ✓ KL divergence evita reward hacking
  - ✓ Reward model define qué significa “mejor”
- 

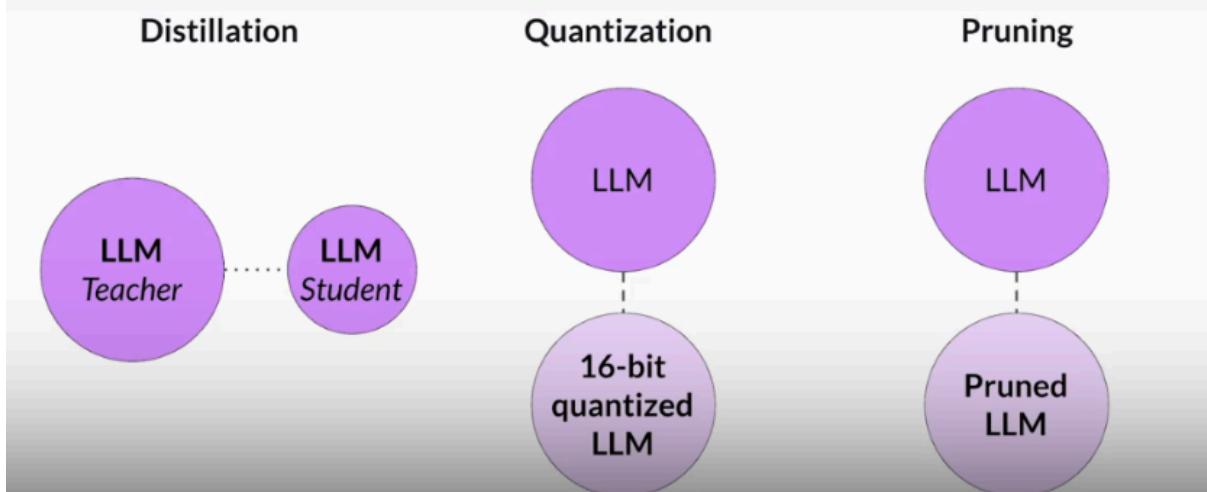
## 16. Idea clave para llevarte del Lab 3

Lab 2 te enseñó a enseñar una tarea.  
Lab 3 te enseña a corregir el comportamiento.

---

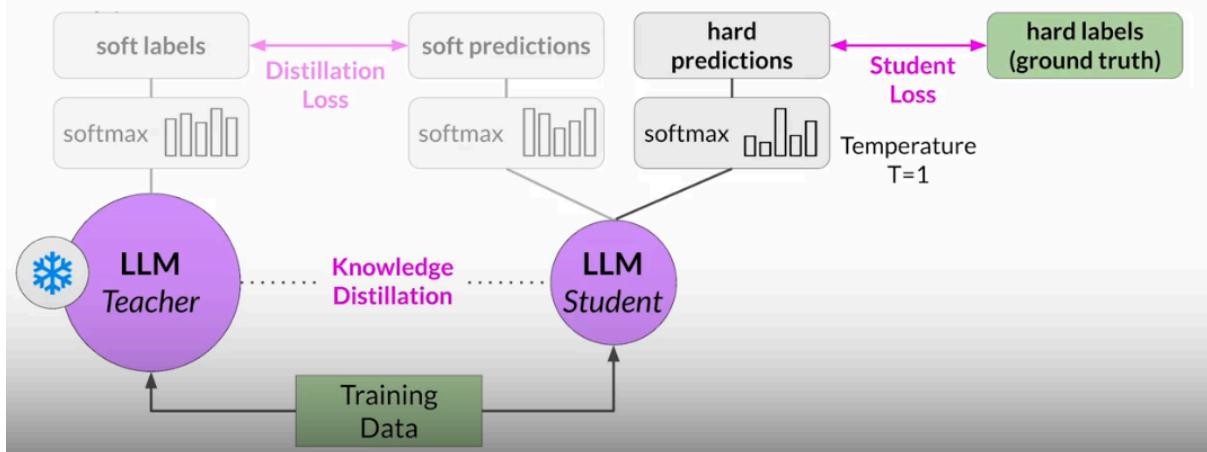
Clase “optimizaciones del modelo para el despliegue”:

## LLM optimization techniques



### Distillation

Train a smaller student model from a larger teacher model



## Objetivo de las optimizaciones

- 👉 Reducir el tamaño del modelo o su costo de inferencia
- 👉 Mantener la mayor performance posible

Las 3 técnicas principales que se presentan son:

1. Distillation
2. Quantization

### 3. Pruning

---

## 1 Model Distillation (Knowledge Distillation)

### Idea central

Entrenar un **modelo pequeño (student)** para que **imité el comportamiento** de un **modelo grande (teacher)**.

- **Teacher:** modelo grande, ya fine-tuneado
- **Student:** modelo más chico, que se usará en producción

⚠ Importante:

**No se achica el modelo original**

Se entrena un **segundo modelo**, más pequeño, para inferencia.

---

### Flujo general

1. Tomás tu **LLM fine-tuneado** → se convierte en el **teacher**
2. Creás un **LLM más pequeño** → **student**
3. **Congelás** los pesos del teacher
4. Ambos reciben el **mismo training data**
5. Comparás las salidas del student contra:
  - el **teacher**
  - el **ground truth**

---

### Tipos de señales que usa el student

- **Soft labels (del teacher)**

- Vienen de la **distribución de probabilidad** del softmax del teacher
- No solo dicen “la respuesta correcta”, sino:
  - qué tokens eran **casi correctos**
  - qué alternativas veía el modelo grande

👉 Esto es lo que se llama **conocimiento oscuro (dark knowledge)**

---

### ● Hard labels (ground truth)

- Son las etiquetas reales del dataset
  - Softmax normal (temperatura = 1)
  - Es el entrenamiento clásico supervisado
- 

## ¿Por qué usar temperatura (Temperature > 1)?

El teacher ya está muy bien entrenado → su softmax suele estar **muy “picudo”** (casi toda la probabilidad en un solo token)

Para que el student aprenda mejor:

- Se aumenta la **temperatura T > 1**
- La distribución se vuelve:
  - más **suave**
  - más **informativa**
  - con probabilidad repartida entre tokens similares

👉 Esto ayuda al student a aprender **patrones**, no solo la respuesta exacta.

---

## Funciones de pérdida

El student se entrena con **dos pérdidas combinadas**:

## 1. Distillation Loss

- Compara:
  - soft labels (teacher,  $T > 1$ )
  - soft predictions (student)

## 2. Student Loss

- Compara:
  - hard predictions (student)
  - hard labels (ground truth)

👉 Ambas se combinan y se hace **backpropagation solo sobre el student**

---

## ¿Qué se usa en producción?

✓ Solo el student model

Beneficios:

- Menor tamaño
- Menor costo
- Menor latencia

---

## Limitaciones importantes

⚠ Distillation **NO es tan efectiva** para:

- Modelos **generativos decoder-only** (ej: GPT-like)

✓ Funciona mejor en:

- **Encoder-only models**
- Ejemplo: **BERT**

- Razón: mucha **redundancia de representación**
- 

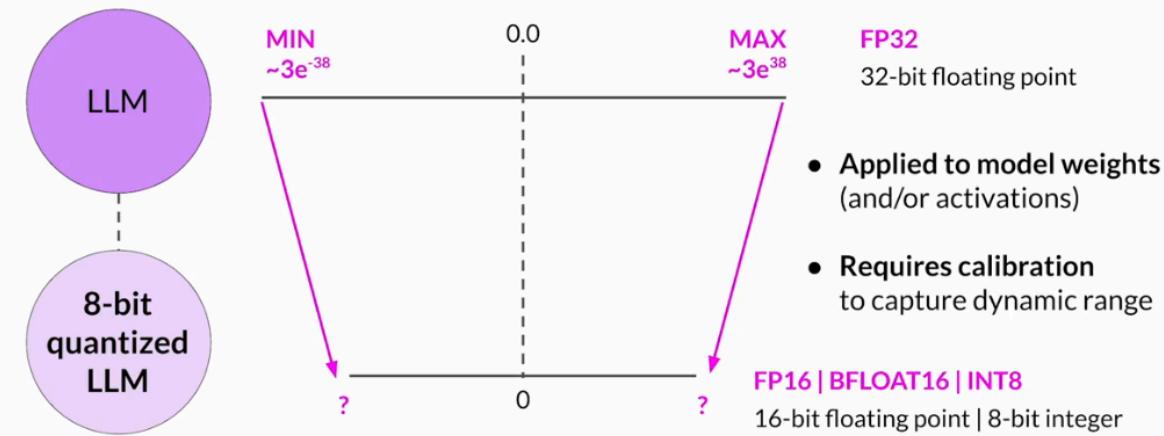
## Ejemplo concreto

- Teacher: FLAN-T5 Large fine-tuneado para resumen
- Student: FLAN-T5 Small
- Producción: usás el **student**, no el teacher

## Quantization post-training (PTQ)

### Post-Training Quantization (PTQ)

Reduce precision of model weights



👉 Post-Training Quantization significa:

El modelo **ya está entrenado**  
NO se re-entrena  
Solo se transforman los pesos (y a veces activaciones)

Esto es importante:

- ✗ No volvés a ajustar pesos
  - ✓ Es barato y rápido de aplicar
-

## 5) ¿Qué se cuantiza exactamente?

### Opción A: Solo los pesos

- Se cuantizan matrices como:
  - $W_q, W_k, W_v, W_o$
  - capas feedforward

✓ Ventaja:

- Menor impacto en accuracy

✗ Desventaja:

- Menor ahorro que cuantizar todo
- 

### Opción B: Pesos + activaciones

- Se cuantiza:
  - pesos
  - salidas intermedias (activaciones)

✓ Ventaja:

- Mucho mayor ahorro
- Inferencia aún más rápida

✗ Desventaja:

- Más ruido numérico
- Más riesgo de degradación

👉 Por eso el video dice:

cuantizar activaciones **impacta más en performance**

---

## 6 El paso clave: calibración (esto es MUY importante)

Acá está el punto que suele no entenderse.

### ? ¿Por qué hace falta calibrar?

Cuando pasás de FP32 → INT8:

- INT8 solo puede representar **256 valores posibles**
- Tenés que decidir:

¿qué rango real de valores va a mapearse a esos 256 números?

---

### Ejemplo simple

Supongamos que en FP32 tus pesos están entre:

[ -3.2 , 4.8 ]

INT8 solo tiene:

[ -128 , 127 ]

👉 La **calibración** hace esto:

- Observa datos reales
- Estima:
  - mínimo
  - máximo
  - distribución
- Calcula un **factor de escala**

Ejemplo conceptual:

```
valor_real ≈ valor_int8 × scale
```

---

## 🔍 ¿Con qué datos se calibra?

- Un pequeño conjunto de datos representativos
  - No se entran pesos
  - Solo se observan rangos
- 

## ⚠ Si calibrás mal...

- Saturación
- Pérdida de información
- Caída fuerte de accuracy

Por eso:

cuantización sin buena calibración = desastre

## Relación con lo que ya viste antes

- QAT (Quantization Aware Training)
  - 👉 se entrena sabiendo que se va a cuantizar
  - 👉 más caro, más preciso
- PTQ (lo de este video)
  - 👉 se cuantiza después
  - 👉 rápido, barato, ideal para despliegue

## 3 Pruning

### Idea central (como dice el video)

Pruning es una técnica que busca:

**Reducir el tamaño del modelo para inferencia eliminando parámetros redundantes**, es decir, pesos que **aportan muy poco o nada** al rendimiento del modelo.

En términos simples:

- El modelo tiene **millones o billones de pesos**
  - Algunos de esos pesos tienen valores:
    - muy pequeños
    - cercanos a **cero**
  - La idea es **eliminarlos** porque casi no influyen en la salida
- 

## ¿Qué pesos se eliminan?

Generalmente:

- Pesos con valores **muy cercanos a cero**
- Pesos que **no contribuyen significativamente** a la performance global

👉 Intuición:

Si un peso casi no cambia la salida del modelo, se puede quitar sin afectar mucho el resultado.

---

## Tipos de Pruning (según el video)

El video menciona **tres grandes enfoques**, sin entrar en algoritmos específicos:

---

### 1 Pruning con re-entrenamiento completo

- Se eliminan pesos
- Luego se **re-entrena el modelo**

- Objetivo: que el modelo se adapte a la nueva estructura

✓ Ventaja:

- Puede recuperar parte de la performance perdida

✗ Desventaja:

- **Muy costoso**
  - Requiere mucho cómputo
  - Poco práctico para LLMs grandes
- 

## ② Post-training Pruning

- El modelo **ya está entrenado**
- Se eliminan pesos directamente
- No se vuelve a entrenar (o solo ajustes mínimos)

✓ Ventaja:

- Más simple
- Más rápido de aplicar

✗ Desventaja:

- El impacto en tamaño y performance suele ser **limitado**
- 

## ③ Pruning dentro de PEFT (ej. LoRA)

- Algunas técnicas de pruning:
  - se combinan con métodos **Parameter-Efficient Fine-Tuning**

- Ejemplo mencionado:
  - **LoRA-based approaches**

👉 En este caso:

- No se modifica todo el modelo
- Se trabaja sobre **adaptadores o subconjuntos de parámetros**

(Se eliminan (o anulan) parámetros del adaptador que tienden a cero o aportan poco.)

---

## Realidad práctica en LLMs (mensaje clave del video)

⚠ En la práctica, especialmente en LLMs grandes:

- **No hay tantos pesos cercanos a cero**
- La mayoría de los parámetros:
  - sí contribuyen al modelo
  - o no pueden eliminarse fácilmente

👉 Resultado:

- Se elimina solo un **pequeño porcentaje** de pesos
  - La reducción real de tamaño y latencia es **poca**
- 

## Consecuencia importante

Por esta razón, en modelos de lenguaje grandes:

- **Pruning suele tener poco impacto real**
- No siempre mejora significativamente:
  - el tamaño

- la velocidad
  - el costo de inferencia
- 

## Comparación implícita que hace el video

Debido a estas limitaciones:

- Pruning es:
  - **X** menos usado que Quantization
  - **X** menos efectivo que Distillation en muchos escenarios

👉 Especialmente para:

- modelos generativos grandes
- despliegues en producción

---

Clase utilización del LLM en aplicaciones:

## El problema de fondo: por qué entrenar no alcanza

Aunque entrenes, ajustes y alinees muy bien un LLM, **hay limitaciones estructurales que no se solucionan solo entrenando.**

### 1.1 Knowledge cutoff (corte de conocimiento)

- El LLM **solo sabe lo que existía hasta el momento de su preentrenamiento.**
- Ejemplo del video:
  - Modelo entrenado en 2022 → pregunta: “*¿Quién es el primer ministro británico?*”
  - Responde: **Boris Johnson X** (ya no era correcto después de 2022)
- **👉 El modelo no puede actualizarse solo.**

---

## 1.2 Alucinaciones

- El modelo **genera texto aunque no sepa la respuesta.**
- Ejemplo:
  - Planta inexistente: “*Martian Dunetree*”
  - El modelo inventa una descripción completa ✗
- 👉 No distingue entre “sé” y “no sé”

## Retrieval Augmented Generation (RAG)

### Definición

RAG es un **framework**, no una tecnología puntual, para:

- Darle al LLM **acceso a datos externos en tiempo de inferencia**

### ¿Qué problemas soluciona?

- Knowledge cutoff ✓
- Alucinaciones ✓
- Falta de información propietaria ✓

## Por qué RAG es mejor que reentrenar

### Reentrenar:

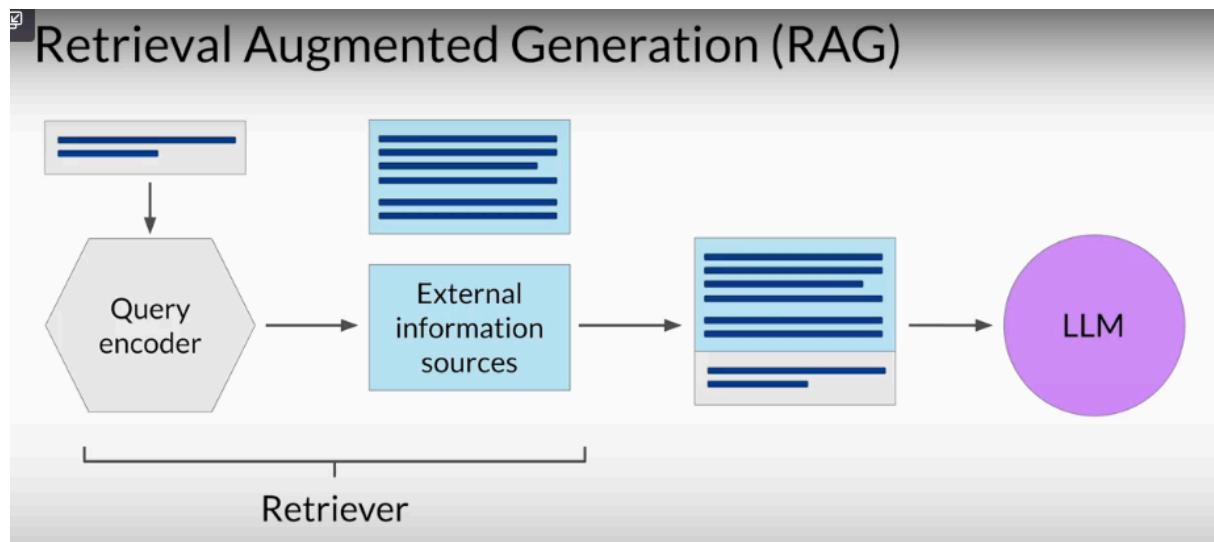
- Muy caro
- Lento
- Hay que repetirlo constantemente

### RAG:

- Más barato
- Flexible
- Información actualizada **en tiempo real**
- Permite usar:
  - Documentos nuevos
  - Bases privadas
  - Datos que nunca estuvieron en el entrenamiento

## Flujo completo (muy importante)

1. Usuario hace una pregunta
2. Query encoder transforma la pregunta en (por ej, pq depende) vector
3. Retriever busca textos relevantes
4. Se combinan:
  - Prompt original
  - Textos recuperados
5. Se envía al LLM
6. El LLM responde **usando esa información**



**El Retriever** en el concepto de RAG (Generación Aumentada por Recuperación) es una parte fundamental que ayuda a los modelos de lenguaje a acceder a información externa que no estaba disponible durante su entrenamiento. Imagina que el modelo de lenguaje es como un estudiante que ha aprendido mucho, pero su conocimiento se detiene en un momento específico. El Retriever actúa como un asistente que busca información actualizada y relevante para ayudar al modelo a responder preguntas de manera más precisa.

En términos simples, el Retriever toma la pregunta del usuario y la convierte en un formato que puede usar para buscar en una base de datos externa. Luego, encuentra los documentos más relevantes y los combina con la pregunta original. Esto permite que el modelo genere respuestas más informadas y precisas, mejorando así la experiencia del usuario. Por ejemplo, si un abogado está buscando información sobre un caso específico, el Retriever puede buscar en documentos legales y proporcionar datos actualizados que el modelo de lenguaje puede usar para dar una respuesta adecuada.

El proceso del Retriever en RAG se puede resumir en los siguientes pasos:

1. Codificación de la Consulta: Toma la consulta del usuario y la convierte en un formato que puede ser utilizado para buscar en la fuente de datos externa.
2. Búsqueda de Información: Utiliza esa consulta codificada para buscar en una base de datos o conjunto de documentos externos, como un almacén de vectores o una base de datos SQL.
3. Recuperación de Documentos: Encuentra los documentos o fragmentos de texto más relevantes que contienen la información solicitada.
4. Combinación de Información: Combina la información recuperada con la consulta original para crear un nuevo contexto que se pasa al modelo de lenguaje.
5. Generación de Respuesta: El modelo de lenguaje utiliza este contexto ampliado para generar una respuesta más precisa y relevante.

## Preparación de datos para RAG (muy importante)

### Problema 1: Context window

- Los LLMs tienen un límite de tokens
- Los documentos largos **no entran completos**

### Solución:

- Dividir en **chunks**
  - Cada chunk entra en el contexto
- 

## Problema 2: Formato recuperable

- El texto debe poder compararse semánticamente

### Solución:

1. Dividir documentos en chunks
  2. Convertir cada chunk en embedding
  3. Guardarlos en un vector store
- 

### Clase “Ayudar a los LLM a razonar y planificar con la cadena de pensamiento o chain of thought prompting:

La orientación en cadena de pensamientos es una técnica poderosa que mejora la capacidad del modelo para razonar ante los problemas.

### Problema que se busca resolver

Los **LLM** tienen **dificultades con el razonamiento complejo**, especialmente en:

- Problemas de **múltiples pasos**
- **Cálculos matemáticos**
- Tareas que requieren **planificación secuencial**

Esto ocurre incluso en modelos grandes y bien entrenados, que pueden fallar aunque entiendan el contexto general.

---

## 2. Qué es la Cadena de Pensamiento

La **cadena de pensamiento (Chain of Thought)** es una técnica de *prompting* que:

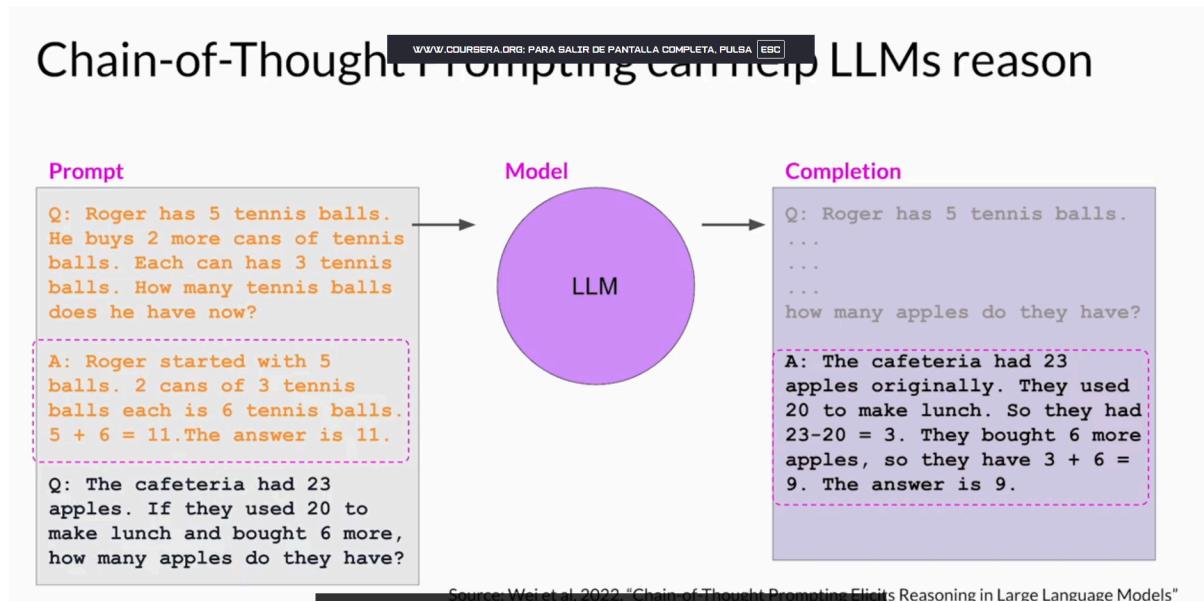
- Induce al modelo a **descomponer un problema en pasos intermedios**
  - Imita la forma en que **razona un humano**
  - Hace explícito el **proceso lógico** antes de llegar a la respuesta final
- 

### 3. Chain of Thought Prompting

**Chain of Thought prompting** consiste en:

- Incluir **ejemplos con razonamiento paso a paso** en el prompt
- Usarse en escenarios de **one-shot o few-shot inference**
- Enseñar al modelo *cómo razonar*, no solo qué responder

El modelo aprende el **patrón de razonamiento** a partir del ejemplo dado.




---

### 4. Beneficios principales

- Mejora la **precisión en problemas multi-paso**
- Produce respuestas más **robustas y transparentes**
- Facilita el **razonamiento lógico**, no solo la generación de texto

- Es útil en tareas más allá de matemática (lógica, física, decisiones simples)
- 

## 5. Limitaciones

- Aunque mejora el razonamiento, **no elimina los errores matemáticos**
  - No es suficiente para tareas que requieren **cálculos exactos**
  - En esos casos, se recomienda **integrar herramientas externas** (programas, calculadoras, APIs)
- 

## 6. Idea clave

**La cadena de pensamiento no hace al modelo más inteligente, sino más ordenado al razonar.**

Es una técnica fundamental para ayudar a los LLM a **razonar, planificar y explicar** cómo llegan a una respuesta.

### Clase Modelos lingüísticos asistidos por programa (PAL)

**PAL (Program-Aided Language Models)** es un enfoque que:

- Combina un **LLM** con un **intérprete de código (ej. Python)**
- Hace que el LLM:
  1. **Razone el problema**
  2. **Genere código ejecutable**
- Delega los cálculos a un programa que **sí sabe hacer matemática exacta**

👉 El LLM **razona**, el programa **calcula**.

Para operaciones matemáticas más complejas, como la aritmética con números grandes, la trigonometría o el cálculo, la **PAL** es una técnica eficaz que permite garantizar que todos los cálculos que realice la aplicación sean precisos y fiables

## Problema que PAL busca resolver

Los **LLM** no realizan cálculos reales:

- No “calculan”, **predicen tokens**
- Pueden **razonar bien** pero **fallar en operaciones matemáticas**
- El error crece con:
  - Números grandes
  - Muchas operaciones
  - Matemática avanzada (álgebra, trigonometría, etc.)

Esto puede generar **errores críticos** en aplicaciones reales (precios, impuestos, recetas, medidas).

## Cómo funciona PAL (flujo básico)

### 1. Prompt con ejemplos (one-shot / few-shot)

Incluye:

- Razonamiento en lenguaje natural
- Código Python asociado a cada paso

### 2. El LLM genera una solución en formato PAL

- Comentarios (#) → razonamiento
- Código Python → cálculos reales

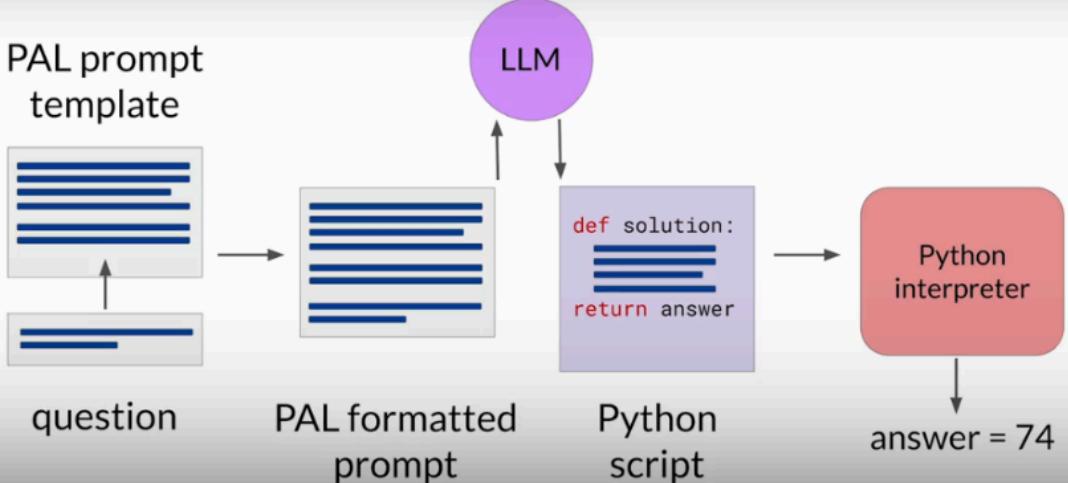
### 3. Un intérprete ejecuta el código

- Produce el resultado correcto

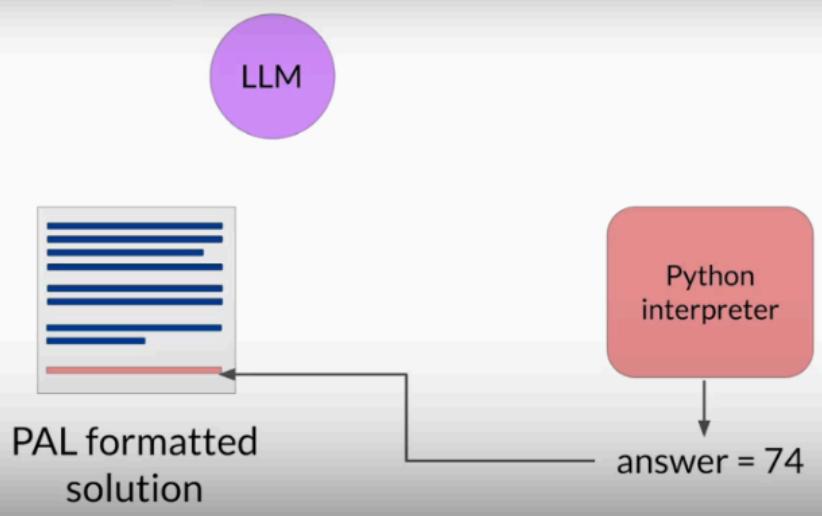
### 4. El resultado se devuelve al LLM

- El modelo genera la respuesta final correcta

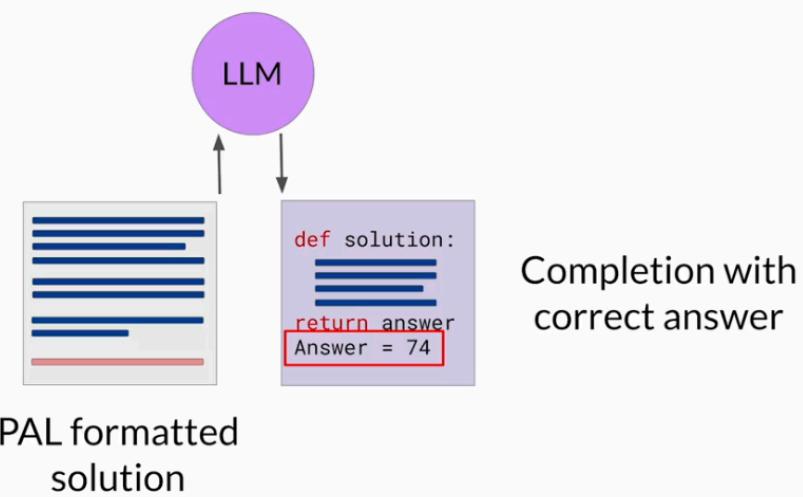
## Program-aided language (PAL) models



## Program-aided language (PAL) models



## Program-aided language (PAL) models



Cuando utilizas el marco de trabajo PAL (Program-Aided Language Models), primero le das al modelo un ejemplo de cómo resolver un problema, junto con la pregunta que deseas que responda. El modelo genera un script en Python que se ejecuta en un intérprete para obtener la respuesta. Sin embargo, después de que obtienes esa respuesta, es importante añadirla de nuevo al prompt original. Esto se hace para que el modelo tenga el contexto completo y pueda generar una respuesta final que incluya la respuesta correcta.

Al final del proceso, después de ejecutar el script de Python y obtener la respuesta, laañades al PAL template original. Esto se hace para que el modelo tenga toda la información necesaria.

Cuando le pasas el PAL template actualizado al LLM, este puede generar una respuesta que:

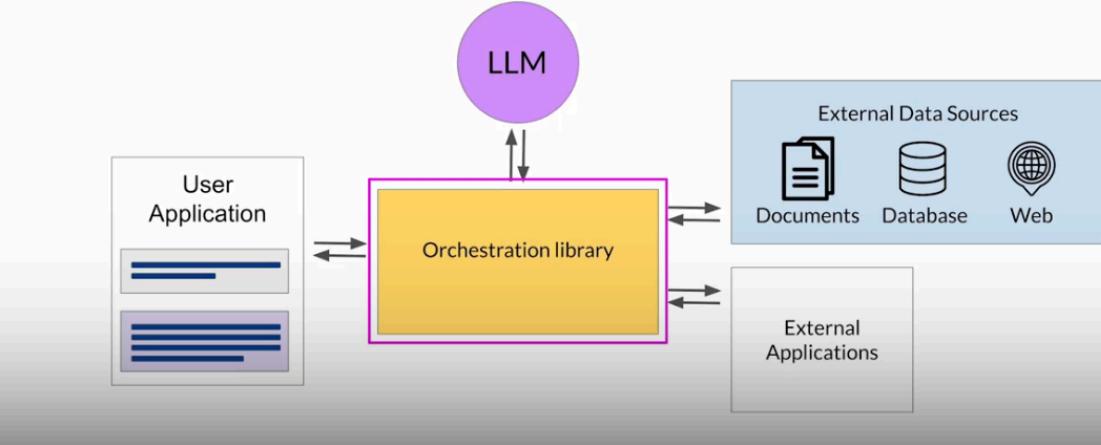
- Confirme la respuesta que obtuviste del script de Python.
- Proporcione contexto adicional o una explicación sobre cómo se llegó a esa respuesta.
- Responda a la pregunta original de manera más completa, integrando tanto el resultado como el razonamiento detrás de él.

En resumen, el LLM no solo te devuelve el código, sino que también puede ofrecer una respuesta más elaborada que incluya la respuesta correcta y el razonamiento que la respalda.

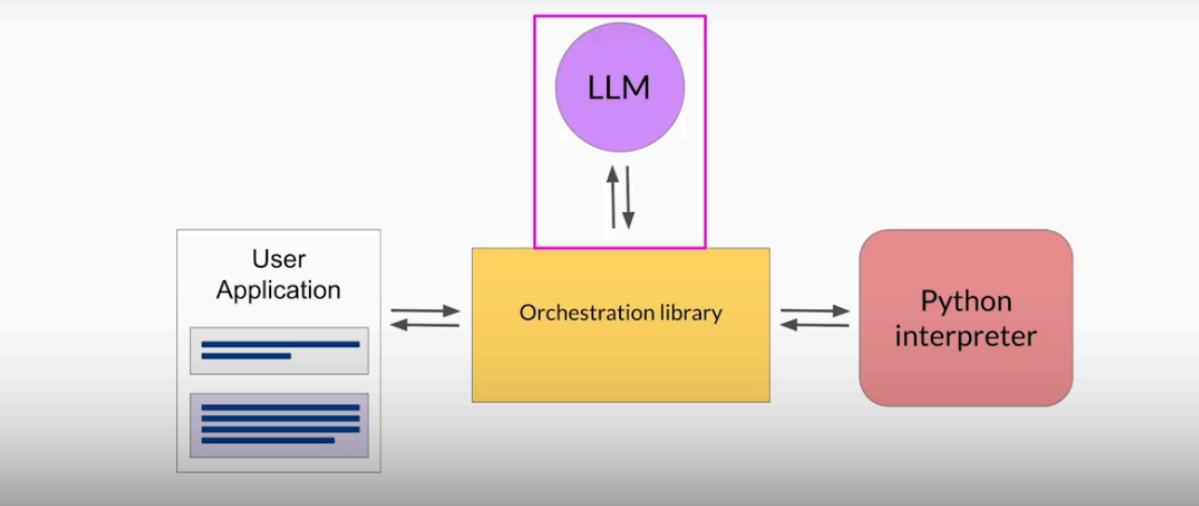
#### **como automatizarlo:**

El orquestador lo hace.

## LLM-powered applications



## PAL architecture



## 7. Arquitectura PAL (visión sistema)

Componentes principales:

### 1. LLM

- Genera razonamiento + código
- Actúa como motor de planificación

### 2. Orquestador

- Controla el flujo

- Decide cuándo ejecutar código
- Envía el script al intérprete

### 3. Intérprete externo (Python)

- Ejecuta cálculos reales
  - Devuelve resultados confiables
- 

#### Clase “React”:

#### Problema que ReAct busca resolver

Muchas aplicaciones reales **no se resuelven solo pensando ni solo ejecutando código**:

- Requieren **razonamiento en varios pasos**
- Necesitan **consultar fuentes externas** (APIs, bases de datos, buscadores)
- Implican **decidir qué acción tomar en cada momento**

👉 PAL resuelve bien *cálculo*, pero **no alcanza para flujos complejos** con múltiples decisiones y fuentes.

---

## 2. Qué es ReAct

ReAct (**R**easoning + **A**ction) es una estrategia de *prompting* que:

- Combina **razonamiento paso a paso (Chain of Thought)**
- Con **planificación y ejecución de acciones**
- En ciclos iterativos hasta llegar a la respuesta

Fue propuesto en **2022 (Princeton + Google)**.

---

## 3. Idea central

**El LLM piensa, elige una acción, observa el resultado y vuelve a pensar.**

Este ciclo se repite hasta resolver el problema.

---

## 4. Estructura básica de ReAct

ReAct usa siempre el mismo patrón:

Question  
Thought  
Action  
Observation  
(Thought)  
(Action)  
(Observation)  
...  
Finish

Cada bloque cumple un rol específico.

---

## 5. Componentes clave

### 5.1 Question

- El problema inicial
  - Generalmente **requiere múltiples pasos**
  - Ej.: “¿Qué revista fue fundada primero?”
- 

### 5.2 Thought (Razonamiento)

- Paso **interno de razonamiento**
- Decide:
  - Qué información falta
  - Qué acción conviene ejecutar

Ejemplo:

“Necesito buscar la fecha de fundación de ambas revistas.”

---

### 5.3 Action (Acción)

- Llamada a una **acción externa permitida**
- Solo puede elegirse de un **conjunto predefinido**

Ejemplos típicos:

- `search[entity]`
- `lookup[string]`
- `finish[answer]`

👉 Esto evita que el modelo “invente” acciones imposibles.

---

### 5.4 Observation (Observación)

- Resultado de ejecutar la acción
- Se incorpora al contexto
- Alimenta el siguiente razonamiento

Ejemplo:

“Arthur’s Magazine fue fundada en 1844.”

---

## 6. Ciclo ReAct completo (ejemplo simplificado)

**Pregunta:**

¿Cuál revista fue fundada primero: *Arthur’s Magazine* o *First for Women*?

1. **Thought 1**  
“Busco la fecha de *Arthur’s Magazine*.”

2. **Action 1**  
search[Arthur's Magazine]

3. **Observation 1**  
"Fundada en 1844."

4. **Thought 2**  
"Ahora busco First for Women."

5. **Action 2**  
search[First for Women]

6. **Observation 2**  
"Fundada en 1989."

7. **Thought 3**  
"1844 < 1989, entonces Arthur's Magazine es anterior."

8. **Action final**  
finish[Arthur's Magazine]

9. termina ciclo y devuelve la respuesta al usuario.

**Es importante señalar que en el marco ReAct, el LLM sólo puede elegir entre un número limitado de acciones que están definidas por un conjunto de instrucciones que se adjuntan previamente al texto del prompt de ejemplo.**

### ReAct instructions define the action space

Solve a question answering task with interleaving Thought, Action, Observation steps.

Thought can reason about the current situation, and Action can be three types:

- (1) Search[entity], which searches the exact entity on Wikipedia and returns the first paragraph if it exists. If not, it will return some similar entities to search.
  - (2) Lookup[keyword], which returns the next sentence containing keyword in the current passage.
  - (3) Finish[answer], which returns the answer and finishes the task.
- Here are some examples.

## Flujo real paso a paso

Cada vez que llega una consulta:

1. Tu app construye el prompt:
  - Instrucciones ReAct
  - Ejemplos
  - Pregunta
2. Se lo manda al LLM
3. El LLM responde con:
  - Thought
  - Action
4. El orquestador:
  - Ejecuta la acción
  - Devuelve la Observation
5. Se repite el ciclo hasta **Finish[ . . . ]**

👉 Todo ocurre **en runtime**.

Un Agente en LangChain utiliza un LLM como motor de razonamiento para determinar qué acciones tomar y en qué orden. A diferencia de una cadena (Chain) que es una secuencia fija de pasos, un Agente observa el input del usuario y decide dinámicamente si debe consultar una API, buscar en una base de datos o ejecutar código Python para resolver el problema, basándose en las descripciones (Docstrings) de las herramientas disponibles.

Eso es exactamente la idea de **ReAct (Thought → Action → Observation)**, pero implementada como “Agente”

Una **chain** es un pipeline fijo (más determinístico):

- Paso 1: formatear prompt
- Paso 2: llamar LLM
- Paso 3: llamar retriever (docs)
- Paso 4: segundo LLM para responder con contexto
- etc.

👉 Si el flujo es siempre el mismo, usás **Chains**.

Pero si el flujo depende de lo que pase (ramas, decisiones), usás **Agents**.

## Cómo encaja con lo que venías viendo (PAL / ReAct)

### PAL en LangChain

- Tool: “Python interpreter”
- El LLM genera código
- LangChain ejecuta el código y devuelve el resultado

### ReAct en LangChain

- Agent tipo ReAct
- Tools: search/lookup/DB/API
- LangChain maneja el loop “pensar → actuar → observar”

---

### clase arquitecturas de aplicación LLM:

## Arquitecturas de Aplicación con LLM (visión end-to-end)

La arquitectura de una aplicación basada en LLM se organiza en **capas**, donde el modelo es solo **una parte del sistema**, no el sistema completo.

---

### 1. Consumers (Usuarios y Sistemas)

(parte superior del diagrama)

- **Usuarios humanos** (chat web, app móvil)

- **Otros sistemas** (APIs, microservicios, integraciones)

👉 Son quienes **consumen** la aplicación, no interactúan directamente con el LLM, sino con toda la pila.

---

## 2. Application Interfaces

(Websites, Mobile Apps, APIs)

Esta capa define **cómo se accede** a la aplicación:

- UI web / mobile
- REST / GraphQL APIs
- Autenticación, autorización, rate limiting

📍 Acá viven:

- seguridad
  - control de acceso
  - experiencia de usuario
- 

## 3. LLM Tools & Frameworks

(LangChain, model hubs, orchestration)

Esta capa conecta la app con el modelo y el resto del sistema.

Incluye:

- **Frameworks** como LangChain / LangGraph
- **Prompt templates**
- **Agents** (ReAct, PAL, tool-calling)
- **Memory**

- **Model hubs** (gestión centralizada de modelos)

👉 Es donde implementás:

- RAG
- ReAct
- PAL
- Chain-of-Thought  
sin reescribir todo desde cero.

---

## 4. Information Sources

(Documents – Database – Web)

Son las **fuentes externas de información** que el LLM puede usar:

- Documentos (PDFs, manuales, policies)
- Bases de datos (SQL / NoSQL)
- Web / APIs externas

📍 Esto habilita:

- **RAG (Retrieval Augmented Generation)**
- agentes que consultan datos reales
- reducción de alucinaciones

---

## 5. LLM Models

(Optimized LLM)

El **motor de razonamiento** de la aplicación.

Puede ser:

- modelo base (foundation model)
- modelo adaptado (fine-tuning, LoRA, RLHF)
- modelo optimizado para inferencia

Consideraciones clave:

- latencia (real-time vs batch)
- costo
- tamaño del modelo
- calidad de razonamiento

👉 El LLM **no vive solo**, siempre está integrado al resto de la arquitectura.

---

## 6. Generated Outputs & Feedback

(Resultados + feedback del usuario)

La aplicación:

- devuelve respuestas al usuario
- puede **almacenar salidas**
- puede **capturar feedback** (👍 👎)

Esto permite:

- extender el contexto más allá de la ventana del modelo
- recolectar datos para:
  - evaluación
  - alineación
  - re-entrenamiento

---

## 7. Infrastructure (base de todo)

(Training, fine-tuning, serving, networking)

Provee:

- cómputo (CPU / GPU)
- almacenamiento
- red
- despliegue

Puede ser:

- on-premise
- cloud (pay-as-you-go)

Incluye:

- serving del modelo
- pipelines de entrenamiento
- escalado
- observabilidad

---

## 8. Consideraciones clave del ciclo de vida (lo que cierra el curso)

### A. Alineación del modelo

- RLHF para:
  - mejorar utilidad
  - reducir toxicidad

- aumentar seguridad

## B. Optimización para inferencia

- Distillation
- Quantization
- Pruning

👉 Reduce costos y requisitos de hardware.

## C. Mejora en despliegue

- Prompts estructurados
- Conexión con herramientas y datos externos
- Agentes y orquestadores

## 🧠 ¿Qué es AWS SageMaker JumpStart?

Amazon SageMaker JumpStart es un **model hub + plataforma de despliegue y fine-tuning** dentro de AWS que te permite:

- Usar **modelos fundacionales (Foundation Models)** listos para producción
- Desplegarlos con pocos clics
- Afinarlos (**fine-tuning**), incluso con **PEFT / LoRA**
- Obtener **endpoints en tiempo real** para integrarlos en apps reales

👉 La idea clave: **pasar de “aprendí LLMs” a “tengo un LLM en producción”** rápido y a escala.

---

## 🧱 JumpStart dentro de la arquitectura de una app con LLMs

JumpStart cubre **muchas capas del stack completo** que ya estudiaste:

Capa	¿JumpStart la cubre?
Infraestructura (compute, GPUs)	✓
Modelo LLM (foundation model)	✓
Frameworks y tooling	✓
Fine-tuning (incl. PEFT)	✓
Endpoint / API de inferencia	✓

En vez de armar todo “a mano”, JumpStart **te da piezas ya integradas**.

---



## Model Hub: modelos listos para usar

En JumpStart encontrás:

- Modelos **texto, visión, multimodales**
- Modelos de **Hugging Face** integrados oficialmente
- Variantes de un mismo modelo (base, large, etc.)

### Ejemplo clave: Flan-T5

Vos usaste **Flan-T5 Base** en los labs por temas de recursos.

En JumpStart podés elegir:

- Flan-T5 Base
- Flan-T5 Large
- Otras variantes más potentes



AWS + Hugging Face ya hicieron el trabajo pesado de compatibilidad.

---



## Opción 1: Deploy directo (sin entrenar)

Podés **desplegar un modelo tal cual viene**:

1. Elegís el modelo
2. Seleccionás:
  - Tipo de instancia (CPU/GPU)
  - Tamaño
3. Ajustás seguridad (IAM, VPC, etc.)
4. Click en **Deploy**

 Resultado:

- Se crea un **endpoint persistente en tiempo real**
- Pagás **por hora de la instancia**, esté o no recibiendo requests

 **Clave para el examen y la vida real**

 **Siempre borrar endpoints que no se usan**, si no AWS te factura igual.

---

## **Opción 2: Fine-Tuning del modelo**

Si el modelo soporta entrenamiento (muchos sí):

### ◆ **Configurás:**

- Dataset de entrenamiento (S3)
- Dataset de validación (S3)
- Tipo y tamaño de GPU para entrenar

### ◆ **Costos**

- Se paga **solo mientras entrena**
  - Conviene **la instancia más chica posible** para tu caso
-

## PEFT y LoRA (lo que ya sabés, aplicado)

Acá JumpStart brilla fuerte 💡

En la sección de **hiperparámetros**:

- Elegís **PEFT**
- Seleccionás **LoRA**
- Todo desde un **dropdown**, sin escribir código complejo

👉 Conceptualmente:

- ❌ No reentrenás todo el modelo
- ✅ Entrenás matrices pequeñas (LoRA)
- ✅ Menor memoria, menor costo, mismo modelo base

Esto conecta **directo con tus labs de PEFT**.

---



## Opción 3: Notebook generado automáticamente

Si sos más de código que de clicks:

- JumpStart puede **generarte un notebook**
- Contiene:
  - Código de deploy
  - Código de training
  - Configuración de endpoints

📌 Es literalmente el “**backend**” de la UI, pero expuesto en Python.

Ideal para:

- Automatizar
- Versionar

- Integrar en pipelines MLOps
- 



## GPUs y costos (muy importante)

A diferencia de los labs:

- JumpStart **usa GPUs reales**
- Pricing **on-demand**
- Algunas instancias son **muy caras**

Buenas prácticas:

- Usar la GPU mínima necesaria
- Monitorear costos
- Eliminar endpoints inactivos

Esto es **100% real-world AWS**, no entorno educativo.

---



## Recursos extra incluidos

JumpStart no es solo modelos:

- Blogs técnicos
- Ejemplos
- Notebooks
- Videos
- Soluciones end-to-end



👉 Es una puerta de entrada al ecosistema ML productivo de AWS.