UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE
ESCUELA POLITÉCNICA SUPERIOR DE ELCHE
GRADO EN INGENIERÍA INFORMÁTICA EN
TECNOLOGÍAS DE LA INFORMACIÓN

Fine-Tuning de Modelos de Lenguaje para Generación de Código en CODE-2

TRABAJO FIN DE GRADO

Febrero - 2026

AUTOR: Santos Bernabeu Marroquí

DIRECTOR: Francisco Federico García Crespi

"No temo a los ordenadores.
Temo la falta de ellos."

- Isaac Asimov

AGRADECIMIENTOS

Agradezco profundamente a mis profesores y profesoras del Grado en Ingeniería Informática en Tecnologías de la Información por haberme proporcionado las bases necesarias para afrontar este proyecto con confianza y rigor.

A mi familia, por su apoyo incondicional, su paciencia y su constante aliento. Sin su respaldo, este camino no habría sido posible.

Y finalmente, a mis compañeros y compañeras de estudios, por las conversaciones, ideas y motivación compartidas a lo largo de estos años.

Este trabajo es, en parte, fruto de todos ellos.

RESUMEN EJECUTIVO

Este Trabajo Fin de Grado tiene como objetivo el estudio, aplicación y evaluación del modelo **T5-Small** para tareas de *generación automática de código ensamblador* en el entorno académico **CODE-2**[16]. Dicho entorno, ampliamente utilizado en docencia para la enseñanza de arquitectura de computadores[16], presenta una gramática y sintaxis bien definida que lo convierte en un candidato adecuado para la aplicación de modelos de lenguaje generativos.

El trabajo se ha estructurado en torno a dos ejes principales: por un lado, el **fine-tuning** del modelo utilizando técnicas avanzadas como *Prompt-Tuning*, *Adapters*, *Low-Rank Adaptation (LoRA)* y *Full Fine-Tuning* (capítulo 3); y por otro lado, la aplicación de **técnicas de cuantización**, concretamente *Post-Training Quantization (PTQ)*, con el fin de reducir el tamaño del modelo y su coste de inferencia (capítulo 4).

Para la evaluación se ha diseñado un conjunto de pruebas que abarca métricas estándar en la generación de código: **Exact Match**, **BLEU** y **ROUGE-L**, analizando además el consumo de recursos computacionales, tiempos de entrenamiento y desempeño bajo diferentes configuraciones de entrenamiento y hardware.

Los resultados obtenidos confirman que el enfoque de **Full Fine-Tuning** ofrece la mejor calidad y consistencia en la generación de código, alcanzando un Exact Match del 83 % tras optimización de parámetros de decodificación (80 % en el modelo entrenado base con dataset de 36k ejemplos). Aunque técnicas diversas fueron evaluadas (Prompt-Tuning, Adapters, LoRA, Full Fine-Tuning), solo Full Fine-Tuning demostró rendimiento suficiente para aplicaciones educativas donde la corrección semántica es crítica. Para contextos con recursos muy limitados, LoRA constituye una alternativa viable aunque con menor precisión. Además, se documentaron limitaciones fundamentales de técnicas alternativas como QAT (incompatible con Transformers en Windows) y poda agresiva (causa colapso del modelo).

Nota: este trabajo incluye numerosos fragmentos comentados y código reproducible, con el fin de facilitar la comprensión y replicabilidad de los experimentos. Se

recomienda prestar especial atención al capítulo 3, donde se detallan las configuraciones de entrenamiento y al anexo donde se documenta el preprocesamiento del dataset y los scripts utilizados.

Palabras clave: Inteligencia Artificial, Procesamiento del lenguaje natural, modelos Transformer, T5-Small, generación de código, fine-tuning, LoRA, cuantización, CODE-2, evaluación de modelos, arquitectura de computadores.

Keywords: Artificial Intelligence, Natural Language Processing, Transformer models, T5-Small, code generation, fine-tuning, LoRA, quantization, CODE-2, model evaluation, computer architecture.

ÍNDICE

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

ÍNDICE DE CÓDIGOS

1. INTRODUCCIÓN

1.1. Contexto y motivación

En los últimos años, el avance de los modelos de lenguaje basados en la arquitectura **Transformer** ha transformado profundamente el campo del procesamiento del lenguaje natural (PLN)[12]. Esta clase de modelos ha logrado resultados punteros en tareas como traducción automática, resumen de texto y generación de código. Entre estos modelos destaca **T5 (Text-To-Text Transfer Transformer)**, desarrollado por Google, cuya arquitectura convierte cualquier tarea de PLN en un problema de entrada y salida de texto (*text-to-text*). Su versión reducida, **T5-Small**, proporciona un equilibrio eficaz entre precisión y eficiencia computacional.

Por otro lado, el entorno **CODE-2**[16] constituye una herramienta didáctica ampliamente empleada en la enseñanza de arquitectura de computadores. Este entorno permite trabajar con programas escritos en un subconjunto de ensamblador simplificado, facilitando el aprendizaje del funcionamiento interno del procesador a nivel de instrucciones, registros y ciclos de ejecución. CODE-2 dispone de un *ensamblador* y un *emulador* integrados, lo que permite una verificación rápida y visual del comportamiento de los programas escritos.

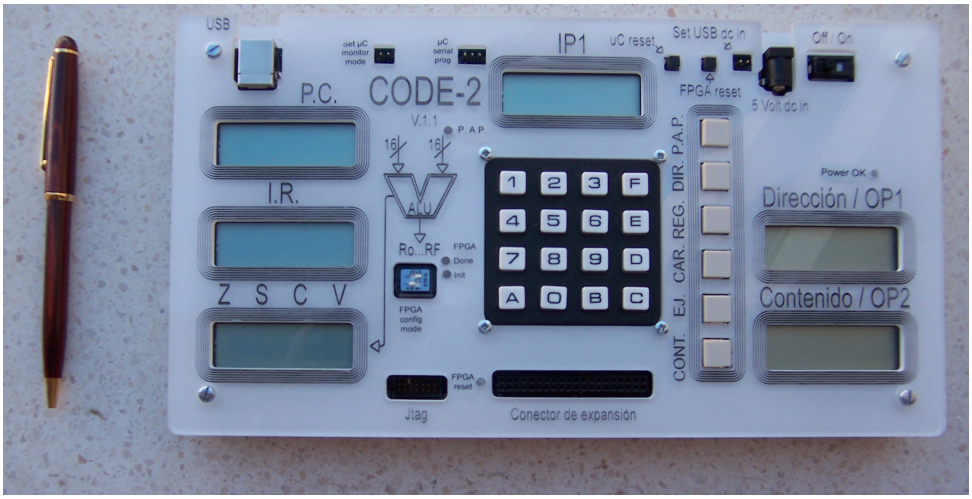

Figura 1.1: Fotografía de la versión de CODE-2 comercializada por la empresa Seven Solutions.

La generación automática de código en CODE-2 representa un desafío interesante: traducir descripciones funcionales en lenguaje natural a instrucciones en ensamblador válidas y ejecutables. Este problema es representativo de tareas más generales de síntesis de código, pero con restricciones más marcadas y una finalidad pedagógica.

El objetivo principal de este trabajo es aplicar técnicas de *fine-tuning* sobre T5-Small para adaptar su comportamiento a la tarea de generación de código ensamblador en CODE-2. Para ello, se exploran distintas metodologías de ajuste fino —incluyendo *Prompt-Tuning*, *Adapters*, *LoRA* y *Full Fine-Tuning*— y se evalúa el rendimiento obtenido mediante métricas estándar en generación automática de código, como BLEU, ROUGE-L y Exact Match.

Además, se consideran técnicas de compresión como la **cuantización**, con el objetivo de reducir el tamaño y el consumo del modelo, facilitando así su implementación en entornos educativos con recursos computacionales limitados[22]. Esta optimización es clave para el desarrollo de asistentes didácticos capaces de integrarse en sistemas accesibles y reutilizables.

En resumen, este Trabajo Fin de Grado explora la intersección entre inteligencia artificial, generación de código y docencia, proponiendo un enfoque práctico y reproducible para la generación de código en CODE-2 mediante modelos generativos entrenados específicamente para ello.

1.2. Objetivos

El propósito general de este Trabajo Fin de Grado es explorar la capacidad del modelo **T5-Small**, en combinación con técnicas modernas de *fine-tuning* y compresión, para generar automáticamente código ensamblador en el entorno **CODE-2**, con un enfoque centrado en la eficiencia y la aplicabilidad educativa.

Objetivo general

Desarrollar y evaluar un modelo de generación automática de código en CODE-2, basado en T5-Small, que sea preciso, eficiente y adecuado para su uso como herramienta de apoyo en entornos educativos.

Objetivos específicos

- Aplicar y comparar distintas estrategias de *fine-tuning*, incluyendo *Prompt-Tuning*, *Adapters*, *LoRA* y *Full Fine-Tuning*, para adaptar T5-Small a la generación de código ensamblador.
- Diseñar, ampliar y preprocesar un conjunto de datos sintético basado en lenguaje natural y sus correspondientes programas en ensamblador CODE-2.
- Implementar técnicas de *cuantización* para reducir el tamaño del modelo final y facilitar su despliegue en entornos de recursos limitados.
- Evaluar el rendimiento de los modelos entrenados mediante métricas estándar en generación de código: **Exact Match**, **BLEU** y **ROUGE-L**.
- Analizar el uso de estos modelos como **herramientas de apoyo al aprendizaje**, capaces de generar soluciones didácticas y comprensibles a partir de consignas funcionales.
- Documentar el proceso experimental para facilitar la reproducibilidad y reutilización del sistema en futuras implementaciones académicas.

1.3. Alcance del proyecto

El presente proyecto se enmarca en el ámbito de la generación automática de código mediante modelos de lenguaje preentrenados, con especial foco en su aplicación a entornos educativos. El alcance del trabajo abarca desde la preparación del conjunto de datos hasta la evaluación y compresión del modelo, **delimitando de forma precisa las tareas a realizar y los recursos utilizados**.

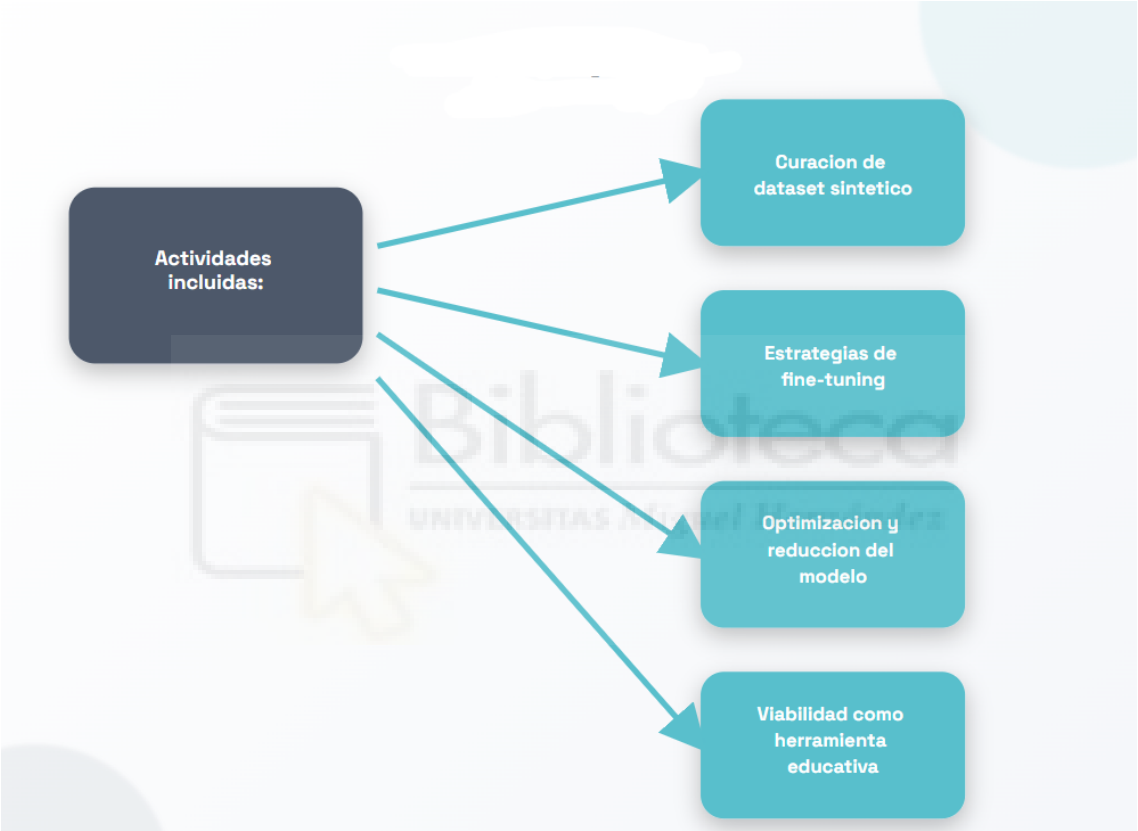

Figura 1.2: Actividades incluidas en el alcance del proyecto.

Entre las actividades incluidas en el alcance del proyecto se encuentran:

- El diseño y curación de un **dataset sintético** basado en descripciones funcionales y su traducción al subconjunto de instrucciones definido por CODE-2.
- La aplicación de diferentes **estrategias de fine-tuning** sobre el modelo T5-Small para adaptar su comportamiento a la tarea de generación de código ensamblador.
- La integración de técnicas de **optimización y reducción del modelo**, como la cuantización post-entrenamiento, para facilitar su despliegue en dispositivos con recursos limitados.
- La **evaluación cuantitativa** del modelo mediante métricas de generación estándar, así como el análisis de rendimiento y uso de recursos durante el entrenamiento e inferencia.
- El análisis de la **viabilidad del modelo como herramienta educativa**, valorando su capacidad para generar código comprensible y funcional a partir de descripciones simples.

Quedan fuera del alcance de este proyecto aspectos como la implementación de interfaces gráficas, el uso de modelos de mayor escala (como T5-Base o T5-Large), o la adaptación del sistema a entornos productivos o industriales.

1.4. Organización del documento

Este documento se estructura en ocho capítulos, organizados de forma que permiten una comprensión progresiva del trabajo realizado, desde los fundamentos teóricos hasta los resultados experimentales y las conclusiones extraídas. A continuación, se describe brevemente el contenido de cada uno:

- **Capítulo 1: Introducción.** Presenta el contexto general del trabajo, los objetivos perseguidos, el alcance del proyecto y la organización del documento.

- **Capítulo 2: Marco Teórico.** Describe los conceptos fundamentales necesarios para el desarrollo del trabajo, incluyendo la arquitectura Transformer, el modelo T5, el entorno CODE-2, y las técnicas de fine-tuning y cuantización empleadas.
- **Capítulo 3: Metodología.** Expone el proceso seguido para la creación del dataset, las estrategias de entrenamiento aplicadas al modelo T5-Small y las configuraciones experimentales utilizadas.
- **Capítulo 4: Resultados y Discusión.** Presenta los resultados obtenidos tras el entrenamiento y evaluación de los modelos, incluyendo métricas cuantitativas, tiempos de ejecución y análisis comparativo entre técnicas.
- **Capítulo 5: Despliegue del Sistema.** Describe la implementación práctica del modelo final en un entorno real accesible mediante web, detallando la arquitectura técnica utilizada y validando la viabilidad del sistema desarrollado.
- **Capítulo 6: Conclusiones y Trabajos Futuros.** Resume los principales hallazgos del trabajo, valora su impacto potencial y plantea líneas de investigación o desarrollo para futuras mejoras.
- **Capítulo 7: Bibliografía.** Recoge las fuentes bibliográficas consultadas y citadas a lo largo del documento.
- **Capítulo 8: Anexos.** Incluye material complementario como fragmentos de código, tablas completas, configuraciones experimentales y otros elementos relevantes que apoyan la reproducibilidad del trabajo.

2. MARCO TEÓRICO

2.1. Introducción a los modelos de lenguaje y Transformers

Los modelos de lenguaje tienen como objetivo principal modelar la probabilidad de aparición de secuencias de texto, aprendiendo representaciones estadísticas de patrones lingüísticos a partir de grandes corpus. Desde los modelos n-grama hasta redes neuronales recurrentes (RNN), la evolución de estas arquitecturas ha culminado en los **Transformers**[21], que se han convertido en el estándar de facto para el procesamiento del lenguaje natural[12].

La arquitectura Transformer está basada en mecanismos de **autoatención**[21], que permiten al modelo ponderar la importancia relativa de cada token de entrada en función del contexto global de la secuencia. A diferencia de las RNN, los Transformers procesan las secuencias en paralelo, lo que mejora drásticamente la eficiencia en entrenamiento y generación.

El **entrenamiento** de un modelo Transformer se realiza típicamente en dos fases: preentrenamiento y ajuste fino (*fine-tuning*). En la primera, el modelo aprende representaciones generales del lenguaje a partir de tareas auto-supervisadas como la predicción de tokens ocultos o la reordenación de frases. Posteriormente, se ajusta a tareas específicas mediante *fine-tuning*, reutilizando los pesos previamente aprendidos.

Cada modelo Transformer se compone de millones —o incluso billones— de **parámetros**, que representan los pesos de las conexiones entre neuronas. En particular, T5-Small, el modelo empleado en este trabajo, cuenta con aproximadamente 60 millones de parámetros entrenables, lo que permite un compromiso aceptable entre precisión y consumo de recursos.

El procesamiento del texto comienza con la conversión de las frases de entrada en unidades mínimas llamadas **tokens**, que pueden corresponder a palabras, subpalabras o caracteres. En el caso de T5, se utiliza el tokenizador *SentencePiece*[9], que

divide el texto en subunidades de significado optimizando la cobertura del vocabulario con un número limitado de tokens posibles.

Gracias a estas características, los Transformers han permitido el desarrollo de modelos versátiles y adaptables, capaces de abordar tareas diversas bajo un paradigma unificado de *text-to-text*[18], como se verá en la siguiente sección con el modelo T5.

2.1.1. Entrenamiento

El entrenamiento de un modelo de lenguaje consiste en ajustar sus parámetros para minimizar una función de pérdida que evalúa la calidad de las predicciones generadas a partir de entradas textuales. En el caso de modelos Transformer como T5, este proceso se realiza mediante aprendizaje auto-supervisado, utilizando pares de entrada y salida textuales generados automáticamente.

El modelo recibe una secuencia de entrada tokenizada (por ejemplo, una descripción funcional) y aprende a generar la secuencia de salida correspondiente (como una instrucción en lenguaje ensamblador). Durante cada iteración de entrenamiento, se calculan los gradientes de error mediante retropropagación, y se actualizan los pesos internos del modelo utilizando optimizadores como `Adam` o `Adafactor`[20], siendo este último más eficiente en modelos grandes como T5.

2.1.2. Parámetros

Los parámetros de un modelo Transformer son los pesos que se ajustan durante el entrenamiento. Estos parámetros están distribuidos entre:

- Embeddings (tokens y posiciones)
- Pesos de las capas de atención
- Pesos de las capas feed-forward
- Parámetros de normalización y sesgo

T5-Small contiene alrededor de 60 millones de parámetros, lo que representa un buen equilibrio entre capacidad expresiva y eficiencia computacional. A mayor número de parámetros, mayor es la capacidad del modelo para aprender patrones complejos, pero también aumenta el riesgo de sobreajuste y los requisitos de hardware.

2.1.3. Tokens

Los tokens son las unidades mínimas de texto que el modelo procesa, resultantes de una tokenización previa. En el caso de T5, se emplea el tokenizador `SentencePiece`, que permite dividir el texto en subpalabras o fragmentos frecuentemente usados.

Por ejemplo, la palabra `\programador"` puede dividirse en los tokens `\pro"`, `\gram"`, `\ador"`, dependiendo del vocabulario aprendido. Este enfoque mejora la cobertura lingüística y permite trabajar con vocabularios fijos de tamaño reducido (en T5, 32,000 tokens).

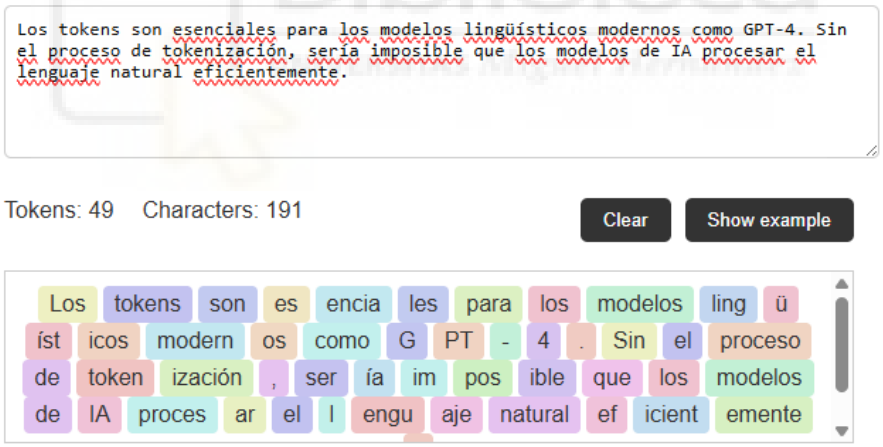

Figura 2.1: Ejemplo de tokenización utilizando SentencePiece. La frase de entrada se divide en tokens subpalabra que el modelo puede procesar.

Durante la inferencia o entrenamiento, el modelo genera secuencias token a token, y la calidad de la salida final depende tanto del entrenamiento como de la elección correcta de tokens intermedios.

2.2. Arquitectura del modelo T5-Small

El modelo T5 (Text-to-Text Transfer Transformer), propuesto por Google Research en 2020[18], representa un enfoque unificado en el que toda tarea de Procesamiento de Lenguaje Natural (PLN) se reformula como un problema de entrada/salida textual. Este marco de trabajo permite reutilizar la misma arquitectura para traducción, clasificación, respuesta automática, resumen, y también para tareas como la generación automática de código.

T5-Small es la versión compacta del modelo, con aproximadamente 60 millones de parámetros, lo que lo hace idóneo para tareas que requieren eficiencia computacional o despliegue en entornos con recursos limitados, como el contexto educativo abordado en este trabajo.

2.2.1. Estructura general

La arquitectura de T5-Small se basa en el esquema Transformer encoder-decoder[21]. Cada bloque de encoder y decoder está compuesto por las siguientes capas fundamentales:

- **Capa de embeddings:** convierte los tokens de entrada en vectores de dimensión fija, combinando embeddings de posición y vocabulario.
- **Atención multi-cabezal (Multi-Head Attention):** permite al modelo centrarse en diferentes posiciones de la secuencia simultáneamente, capturando múltiples relaciones semánticas.
- **Normalización y conexiones residuales:** estabilizan y aceleran el entrenamiento al permitir el flujo directo de gradientes.
- **Capas Feed-Forward:** redes densas aplicadas a cada token de forma independiente, con activación ReLU.

El encoder procesa la entrada textual (una descripción funcional, por ejemplo) y genera una representación latente que captura su semántica. El decoder toma esa

representación y genera secuencialmente los tokens de salida, en este caso, líneas de código en lenguaje ensamblador CODE-2.

2.2.2. Hiperparámetros principales de T5-Small

- Número de capas (encoder/decoder): 6
- Dimensión de embeddings: 512
- Número de cabezas de atención: 8
- Dimensión interna del feed-forward: 2048
- Vocabulario: 32,000 tokens subpalabra (con SentencePiece)

2.2.3. Flujo de inferencia

Durante la inferencia, el modelo sigue un esquema auto-regresivo: genera un token, lo concatena con los anteriores y vuelve a generar el siguiente, hasta completar la secuencia o alcanzar el token especial `<eos>` (end of sequence). Este proceso es sensible a hiperparámetros como la *temperatura*, *top-k*, o *top-p*, que permiten controlar la aleatoriedad y creatividad del modelo.

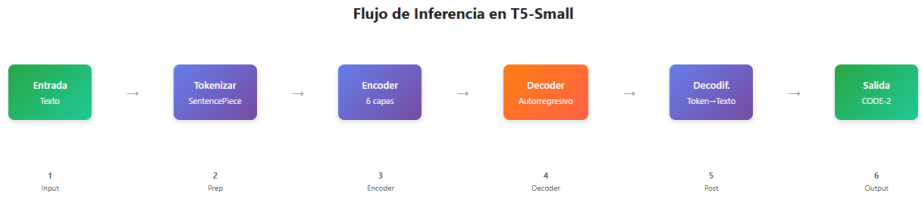

Figura 2.2: Flujo de inferencia en T5-Small. El encoder procesa la entrada y el decoder genera tokens de salida secuencialmente.

2.2.4. Aplicabilidad en generación de código

Gracias a su arquitectura flexible, T5-Small puede ser entrenado con pares de entrada/salida donde el texto de entrada describe una operación (por ejemplo, “suma de dos registros y almacenamiento en R3”) y la salida corresponde a la instrucción

CODE-2 apropiada. El encoder captura el significado semántico de la descripción, mientras que el decoder aprende a mapearlo a una secuencia sintácticamente válida y funcional dentro del repertorio de instrucciones del ensamblador CODE-2.

2.3. Características del CODE-2

El lenguaje CODE-2[16] es un ensamblador educativo diseñado para introducir a los estudiantes en los fundamentos de la arquitectura de computadores y la programación a bajo nivel. Su diseño está inspirado en arquitecturas reducidas (RISC), con un conjunto limitado y didáctico de instrucciones, permitiendo centrarse en los conceptos esenciales del procesamiento secuencial, operaciones aritméticas, control de flujo y manipulación de registros.

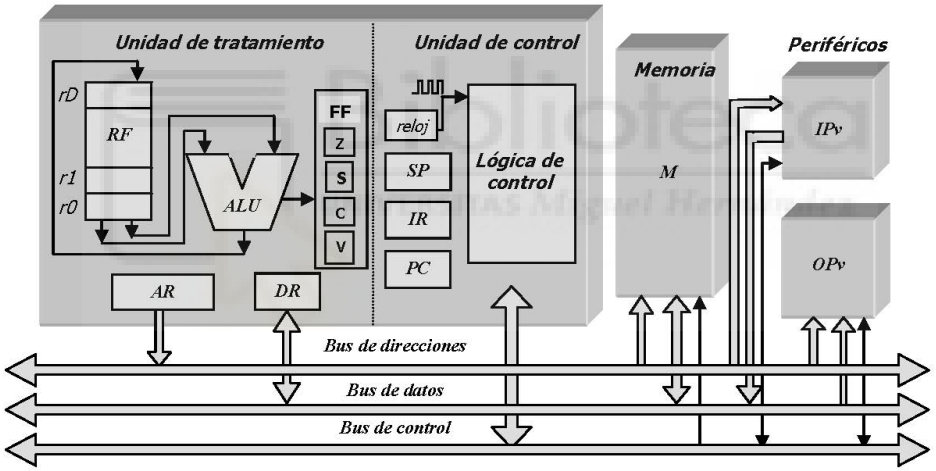

Figura 2.3: Estructura general del entorno CODE-2, mostrando la CPU simulada, registros, memoria y el flujo de ejecución de instrucciones.

CODE-2 es la base del entorno utilizado en este trabajo para entrenar modelos de generación automática de código. La tarea del modelo consiste en generar instrucciones CODE-2 a partir de descripciones en lenguaje natural que especifican operaciones funcionales (por ejemplo: “restar el contenido del registro 1 al registro 2 y guardar el resultado en el registro 3”).

2.3.1. Estructura y repertorio de instrucciones

El repertorio de instrucciones de CODE-2 está compuesto por un conjunto reducido pero completo de instrucciones ensamblador, clasificadas en los siguientes grupos:

- **Transferencia de datos:** `LOAD`, `STORE`, `MOV`
- **Operaciones aritméticas:** `ADD`, `SUB`, `MUL`, `DIV`
- **Operaciones lógicas:** `AND`, `OR`, `NOT`
- **Control de flujo:** `JMP`, `JZ`, `JNZ`, `CALL`, `RET`
- **Gestión de registros y memoria:** instrucciones de acceso y manipulación directa

Cada instrucción está compuesta por un opcode seguido de uno o más operandos, que pueden ser registros (`R1`, `R2`, etc.), direcciones de memoria, etiquetas o constantes.

Codop		Nombre	Nemónico	Parámetros	Formato	Explicación	Nº ciclos
binario	Hex						
0000	0	Cargar	LD	rx,[v]	F3	rx←M(rD+v)	9
0001	1	Almacenar	ST	[v],rx	F3	M(rD+v)←rx	9
0010	2	Carga inmediata baja	LLI	rx,v	F3	rx(15:8)←H'00; rx(7:0)←v	6
0011	3	Carga inmediata alta	LHI	rx,v	F3	rx(15:8)←v	8
0100	4	Entrada	IN	rx,IPv	F3	rx←IPv	8
0101	5	Salida	OUT	OPv,rx	F3	OPv←rx	8
0110	6	Suma	ADDS	rx,rs,ra	F4	rx←rs+ra	7
0111	7	Resta	SUBS	rx,rs,ra	F4	rx←rs-ra	7
1000	8	NAND	NAND	rx,rs,ra	F4	rx←(rs·ra)'	7
1001	9	Desplaza izquierda	SHL	rx	F1	C←rx(15), rx(i)←rx(i-1), i=15,...,1; rx(0)←0	6
1010	A	Desplaza derecha	SHR	rx	F1	C←rx(0), rx(i)←rx(i+1), i=0,...,14; rx(15)←0	6
1011	B	Desplaza arit. dcha.	SHRA	rx	F1	C←rx(0), rx(i)←rx(I+1), i=0,...,14	6
1100	C	Salto	B-	cnd	F2	Si cnd se cumple, PC←rD	6
1101	D	Subrutina	CALL-	cnd	F2	Si cnd se cumple, rE←rE-1, M(rE)←PC, PC←rD	6/10
1110	E	Retorno	RET	-	F0	PC← M(rE); rE←rE+1	8
1111	F	Parar	HALT	-	F0	Parar	6

Figura 2.4: Ejemplo de instrucciones y sintaxis en CODE-2. Cada instrucción consta de un opcode y sus operandos correspondientes

Ejemplo de instrucción en CODE-2:

```
ADD R1, R2, R3
```

significa que se suma el contenido de registro 2 y registro 3, y el resultado se almacena en registro 1.

El lenguaje está diseñado para tener una curva de aprendizaje gradual y facilitar su integración con herramientas de simulación, como las que se describen a continuación.

2.3.2. Herramientas disponibles (emulador, ensamblador, etc.)

El ecosistema CODE-2 incluye herramientas que permiten validar y ejecutar el código ensamblador generado, facilitando el desarrollo de datasets y la evaluación de modelos de generación automática. Entre las principales herramientas destacan:

- **Ensamblador:** traduce el código fuente en instrucciones binarias o pseudocódigo ejecutable por el emulador. Detecta errores sintácticos y verifica operandos.
- **Emulador:** simula el comportamiento de la máquina CODE-2, permitiendo la ejecución paso a paso de programas, visualización de registros y control de flujo.

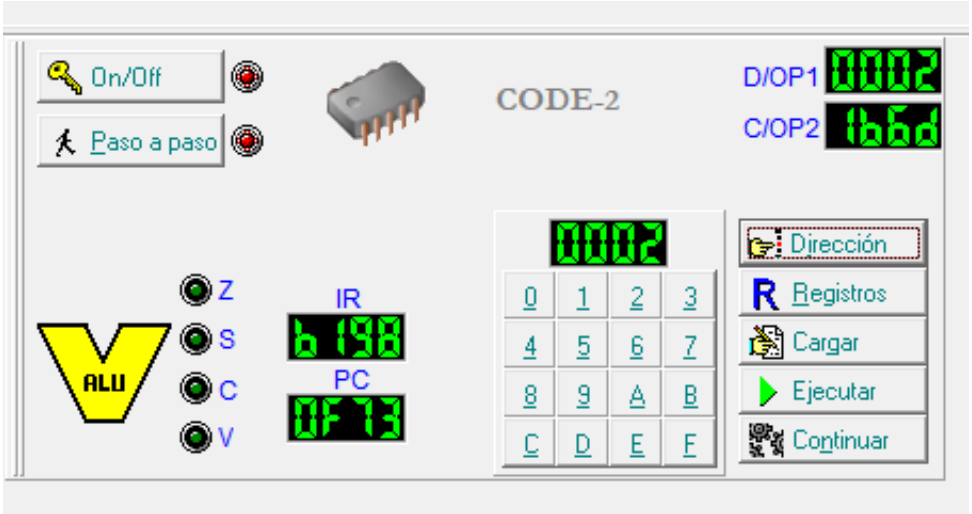

Figura 2.5: Interfaz del ensamblador y emulador de CODE-2. Permite ejecutar instrucciones paso a paso y observar el estado de los registros y la memoria.

- **Visualizador de memoria y registros:** facilita la depuración al mostrar el

estado interno de la CPU simulada en tiempo real.

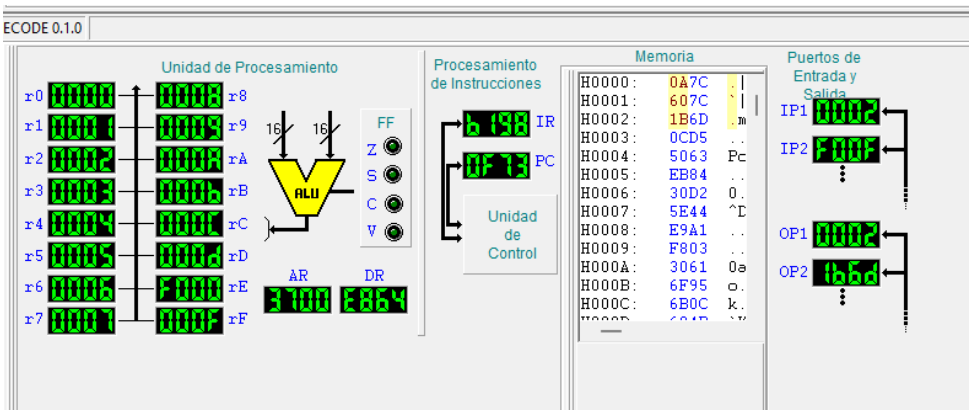

Figura 2.6: Visualizador de memoria y registros en CODE-2, mostrando el contenido de los registros y la memoria durante la ejecución del programa.

- **Desamblador** : convierte el código binario o pseudocódigo de vuelta a instrucciones ensamblador legibles, facilitando la comprensión del flujo de ejecución.

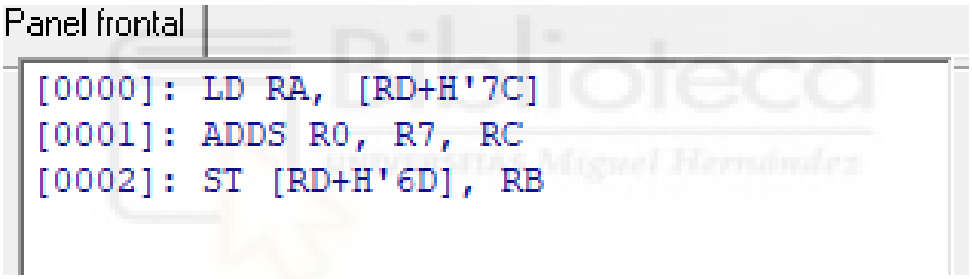

Figura 2.7: Desamblador CODE-2 en funcionamiento, mostrando la conversión de código binario a instrucciones ensamblador.

Estas herramientas son fundamentales para validar automáticamente los programas generados por el modelo de lenguaje, y forman parte del flujo de evaluación descrito en el Capítulo 4.

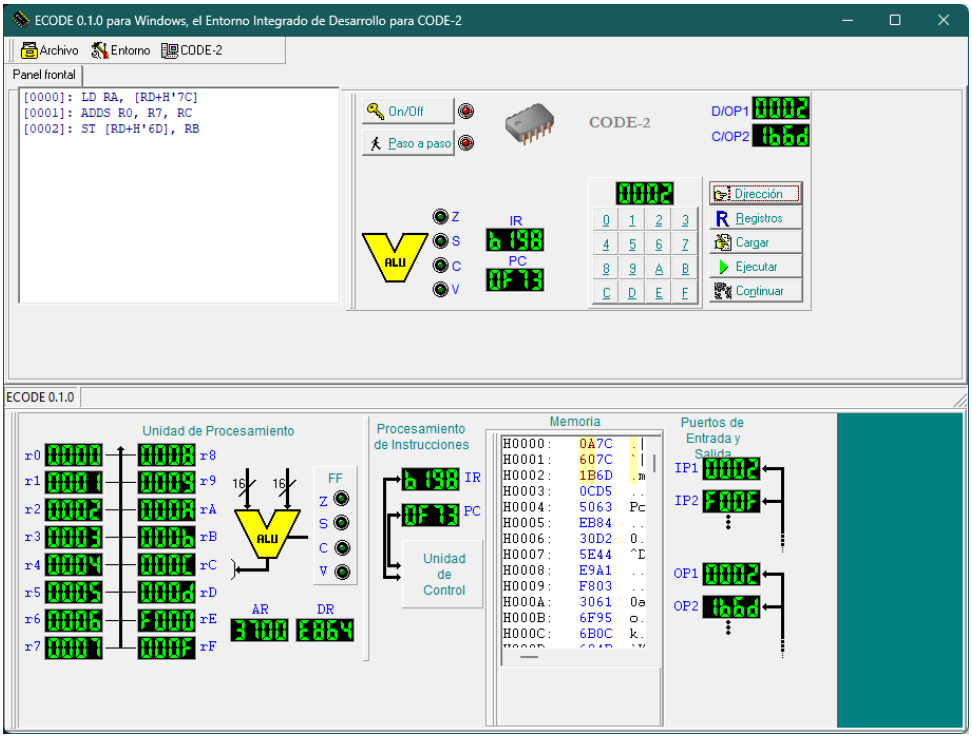

Figura 2.8: Emulador CODE-2 en funcionamiento, mostrando la ejecución de un programa ensamblador y el estado de los registros.

2.4. Técnicas de Fine-Tuning

El fine-tuning o ajuste fino consiste en adaptar un modelo de lenguaje previamente entrenado (preentrenado) a una tarea específica mediante una nueva fase de entrenamiento sobre un conjunto de datos personalizado. Esta estrategia permite aprovechar los conocimientos generales adquiridos durante el preentrenamiento, reduciendo el tiempo de entrenamiento y mejorando el rendimiento en tareas concretas como la generación de código en CODE-2.

Existen múltiples técnicas de fine-tuning, cada una con sus ventajas y compromisos en términos de eficiencia, rendimiento y consumo de recursos[3]. A continuación, se describen las principales técnicas evaluadas en este trabajo.

2.4.1. Prompt-Tuning

Prompt-Tuning[10] es una técnica liviana que consiste en añadir una secuencia de vectores entrenables (conocidos como *soft prompts*) al inicio de cada entrada del modelo, sin modificar los pesos del modelo preentrenado. Estos vectores actúan como un contexto adaptativo que guía la generación del modelo hacia la tarea específica.

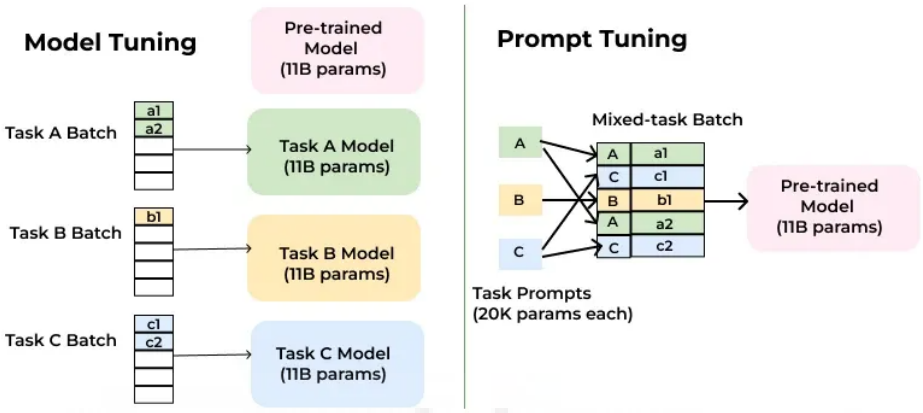

Figura 2.9: Componentes clave del prompt tuning.

- **Ventajas:** Muy eficiente en cuanto a parámetros; ideal para escenarios con recursos limitados.
- **Desventajas:** Menor capacidad de adaptación profunda; rendimiento inferior en tareas complejas si se compara con técnicas más invasivas.

Esta técnica resulta especialmente útil para modelos como T5-Small, donde el número de parámetros del prompt (por ejemplo, 100 tokens $\times$ 512 dimensiones) representa una fracción ínfima del modelo completo.[10]

2.4.2. Adapters

Los Adapters son módulos entrenables que se insertan entre las capas del Transformer. Durante el fine-tuning, sólo los parámetros de estos módulos se actualizan, manteniendo congelados los pesos originales del modelo.

Cada Adapter incluye una red de proyección que reduce la dimensionalidad (por ejemplo, de 512 a 64), aplica una no linealidad (ReLU) y la expande de nuevo, formando un cuello de botella controlado.[5]

- **Ventajas:** Balance entre eficiencia y expresividad; permite cargar múltiples Adapters para tareas distintas en el mismo modelo.
- **Desventajas:** Requiere modificar la arquitectura del modelo base.

En el presente trabajo, se han evaluado adaptadores con diferentes tamaños de cuello de botella y tasas de aprendizaje.

2.4.3. LoRA (Low-Rank Adaptation)

LoRA (Low-Rank Adaptation)[6] es una técnica de fine-tuning eficiente que permite adaptar modelos de lenguaje de gran tamaño introduciendo un número reducido de parámetros entrenables, manteniendo congelados los pesos originales del modelo. En lugar de modificar directamente las matrices completas de proyección de atención, LoRA descompone la actualización de los pesos en matrices de bajo rango.

Concretamente, para una matriz de pesos original $W \in \mathbb{R}^{d\times d}$, LoRA introduce dos matrices entrenables $A \in \mathbb{R}^{d\times r}$ y $B \in \mathbb{R}^{r\times d}$, con $r \ll d$, de forma que la actualización del peso queda definida como:

$$W' = W + \alpha AB$$

donde α es un factor de escalado que controla la magnitud de la adaptación. De este modo, el número de parámetros entrenables se reduce drásticamente, lo que disminuye el consumo de memoria y el coste computacional del entrenamiento.

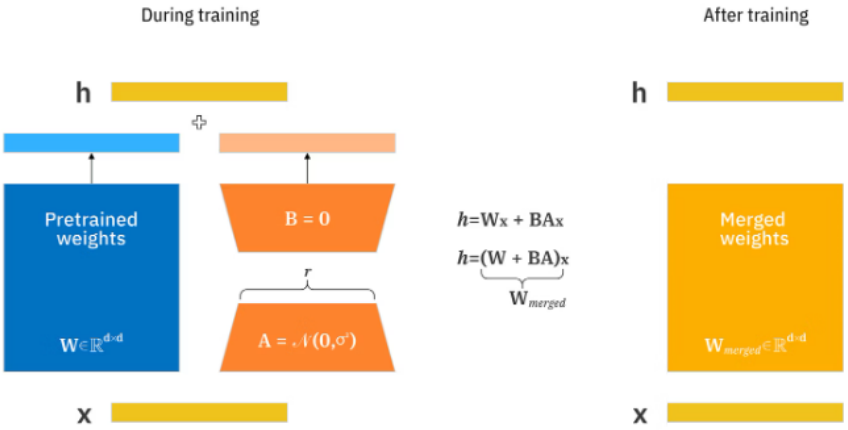

Figura 2.10: Diagrama de actualización de matrices mediante LoRA.

Desde un punto de vista conceptual, LoRA permite desacoplar el conocimiento general adquirido durante el preentrenamiento del modelo base de la adaptación a una tarea concreta. Esto resulta especialmente ventajoso en escenarios con recursos limitados o cuando se desea entrenar múltiples adaptaciones sobre un mismo modelo base.[6]

2.4.3.1 Modelos LoRA fusionados y no fusionados

Una característica relevante de LoRA es la posibilidad de mantener los adaptadores como componentes separados del modelo base o, alternativamente, fusionarlos con los pesos originales una vez finalizado el entrenamiento[6].

En el caso del modelo *no fusionado*, los adaptadores LoRA se almacenan de forma independiente y se combinan dinámicamente con el modelo base durante la inferencia. Este enfoque ofrece una mayor flexibilidad, ya que permite activar o desactivar adaptaciones específicas y continuar el fine-tuning sin modificar los pesos originales.

Por otro lado, el modelo *fusionado* integra permanentemente las matrices de LoRA en los pesos del modelo base, generando un único conjunto de parámetros. Esta fusión elimina la necesidad de utilizar librerías adicionales durante la inferencia y mejora el rendimiento en tiempo de ejecución, a costa de perder la posibilidad de separar nuevamente los adaptadores.

Desde un punto de vista teórico, la fusión de LoRA no altera la función representada por el modelo, ya que la operación $W' = W + \alpha AB$ se incorpora directamente en los pesos finales. La elección entre un modelo fusionado o no fusionado depende del caso de uso: mientras que el modelo no fusionado resulta adecuado durante fases de experimentación, el modelo fusionado es preferible para despliegue y evaluación final.

- **Ventajas:** Reducción significativa del número de parámetros entrenables, menor consumo de memoria, tiempos de entrenamiento más cortos y posibilidad de reutilizar el modelo base.
- **Limitaciones:** Requiere una selección cuidadosa del rango r y de los hiperparámetros de entrenamiento; la fusión de adaptadores es un proceso irreversible.

En conjunto, LoRA se ha consolidado como una de las técnicas más efectivas para el ajuste eficiente de modelos Transformer, siendo especialmente adecuada para tareas de generación de código y entornos con restricciones de hardware.

2.4.4. Full Fine-Tuning

Esta técnica consiste en reentrenar todos los parámetros del modelo preentrenado sobre el nuevo dataset. Es la forma más directa y efectiva de adaptación, pero también la más costosa.

- **Ventajas:** Máxima capacidad de adaptación a la tarea; ideal cuando se dispone de suficiente capacidad de cómputo.
- **Desventajas:** Requiere mucha memoria y tiempo de entrenamiento; riesgo elevado de sobreajuste.

En este TFG se ha realizado fine-tuning completo de T5-Small como línea base de comparación frente a técnicas eficientes como LoRA y Prompt-Tuning.

2.5. Cuantización

La cuantización[8], [13] es una técnica de compresión de modelos que permite reducir su tamaño y acelerar la inferencia al representar los parámetros con menor precisión numérica. En lugar de utilizar números en punto flotante de 32 bits (float32), se emplean representaciones más compactas como 16 bits (float16) o 8 bits (int8), lo que reduce el uso de memoria y mejora el rendimiento en hardware especializado (CPU, GPU o aceleradores como TPU).

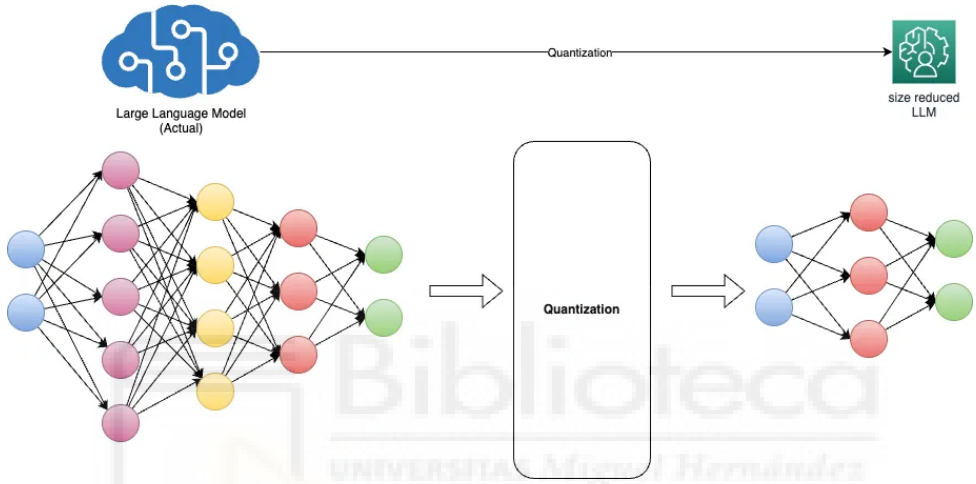

Figura 2.11: Reducción del tamaño del modelo mediante cuantización.

Esta técnica es especialmente relevante en entornos educativos y experimentales como CODE-2, donde el despliegue del modelo puede realizarse en equipos con recursos limitados.

2.5.1. ¿Por qué se necesita cuantizar?

Los modelos de lenguaje modernos, incluso en sus versiones reducidas como T5-Small, pueden tener decenas de millones de parámetros, lo que implica requerimientos elevados de memoria y cómputo para su ejecución. Esto limita su uso en:

- Entornos con recursos computacionales limitados (por ejemplo, dispositivos personales, sistemas educativos).
- Aplicaciones que requieren inferencia en tiempo real.

- Implementaciones embebidas o móviles.

La cuantización puede reducir el tamaño del modelo de forma significativa —en condiciones favorables hasta en un 75 %— con una pérdida de rendimiento controlada si se aplica correctamente[13]. También se combina de forma efectiva con otras técnicas como pruning o distillation[4].

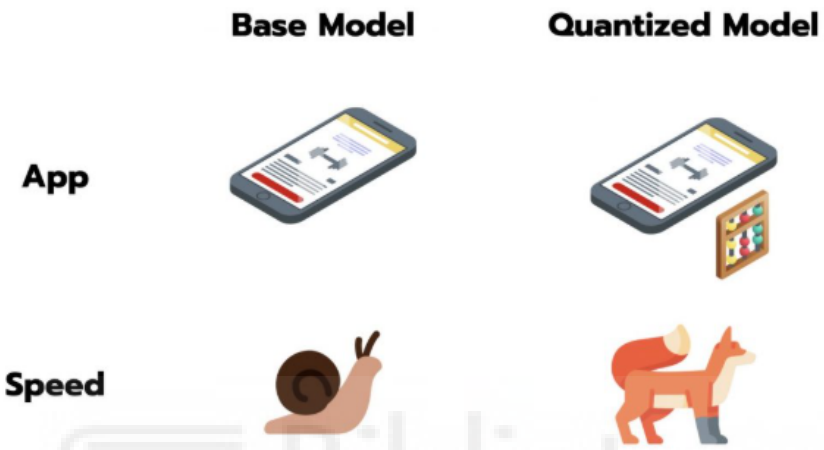

Figura 2.12: Mejora de la velocidad de inferencia mediante cuantización.

2.5.2. Post-Training Quantization (PTQ)

PTQ consiste en aplicar la cuantización al modelo ya entrenado, sin necesidad de volver a entrenarlo. Es una técnica rápida y sencilla de implementar.[8], [13]

- **Ventajas:** No requiere datos de entrenamiento; proceso rápido.
- **Desventajas:** Menor precisión si no se calibran correctamente los rangos de activación.

Normalmente se utilizan unos pocos datos de validación para ajustar las estadísticas (mínimos, máximos, histograma) necesarias para representar correctamente los pesos y activaciones en formato reducido (por ejemplo, `int8`).

2.5.2.1 Estrategias de PTQ utilizadas en este trabajo

En este proyecto se aplicaron dos estrategias complementarias de PTQ:

- **Cuantización dinámica INT8 en PyTorch:** Se comprimieron los pesos del modelo entrenado a 8 bits directamente en PyTorch, sin reentrenamiento. Esta técnica es rápida y permite evaluar el impacto inmediato de la compresión en el rendimiento del modelo.
- **Exportación a formato ONNX con cuantización INT8:** El modelo se convirtió a formato ONNX y se aplicó cuantización a 8 bits. Esta alternativa permite optimización adicional para CPU y mejora la portabilidad (véase subsección 2.5.7).

Ambas estrategias utilizaron datos del conjunto de validación para calibrar automáticamente cómo convertir los valores del modelo a representaciones de 8 bits sin perder demasiada calidad. El objetivo fue alcanzar un equilibrio entre tamaño del modelo (más pequeño) y precisión en la generación de código (manteniendo buena calidad).

2.5.3. Quantization-Aware Training (QAT)

QAT consiste en simular los efectos de la cuantización durante el entrenamiento, permitiendo al modelo adaptarse a la pérdida de precisión desde el inicio.

- **Ventajas:** Mayor precisión que PTQ; rendimiento óptimo tras la cuantización.
- **Desventajas:** Requiere reentrenamiento completo; mayor complejidad en la implementación.

Durante el QAT, se introducen operaciones falsas (*fake quantization*) en el grafo de entrenamiento, simulando el paso a enteros sin alterar los gradientes. [8]

2.5.3.1 Limitaciones de QAT en modelos complejos

A pesar de sus ventajas teóricas, QAT presenta limitaciones prácticas significativas cuando se aplica a modelos complejos como T5-Small:

- **Compatibilidad limitada en este entorno:** Las herramientas automáticas de QAT basadas en `torch.fx` disponibles en PyTorch 2.6 presentan incompatibilidades con modelos Transformer complejos como T5 en el entorno de desarrollo utilizado (Windows). El sistema no puede trazar correctamente el grafo de ejecución del modelo.
- **Limitación de plataforma:** Esta incompatibilidad se reproduce de forma consistente en Windows con la versión de PyTorch empleada. En otros entornos (Linux, versiones anteriores de PyTorch), QAT puede ser viable para modelos Transformer.
- **No es un error de uso:** El problema no es de programación incorrecta, sino una limitación de la cadena de herramientas utilizada en este trabajo.

Por estas razones, en este trabajo se utilizó una alternativa viable: la cuantización post-entrenamiento (PTQ), que comprime el modelo después de entrenarlo, sin los problemas de compatibilidad de QAT.

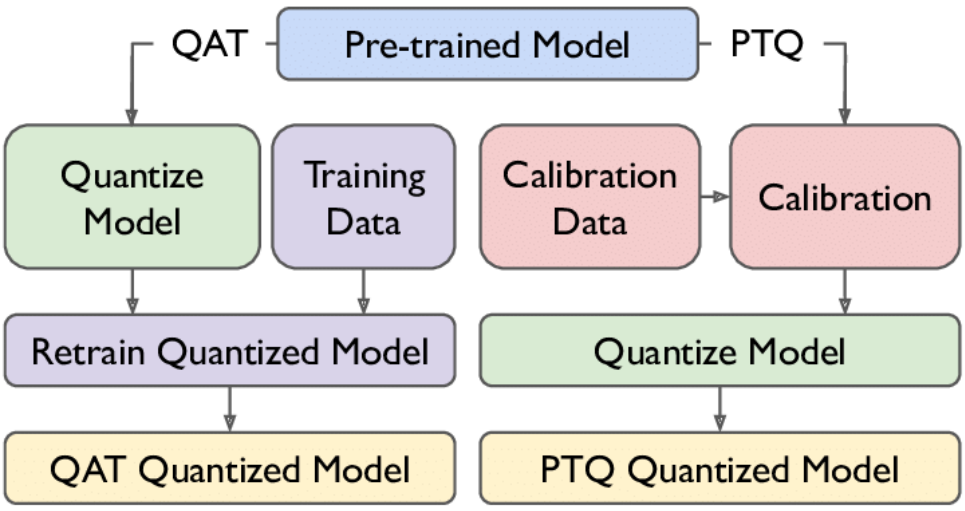

Figura 2.13: Comparativa entre compresión post-entrenamiento (PTQ) y compresión durante el entrenamiento (QAT).

2.5.4. Pruning

El *pruning* o poda elimina parámetros innecesarios del modelo, típicamente aquellos con pesos cercanos a cero. Existen dos tipos principales:

- **Poda estructurada:** elimina unidades completas (neuronas, canales, capas).
- **Poda no estructurada:** elimina conexiones individuales (más difícil de optimizar en hardware).

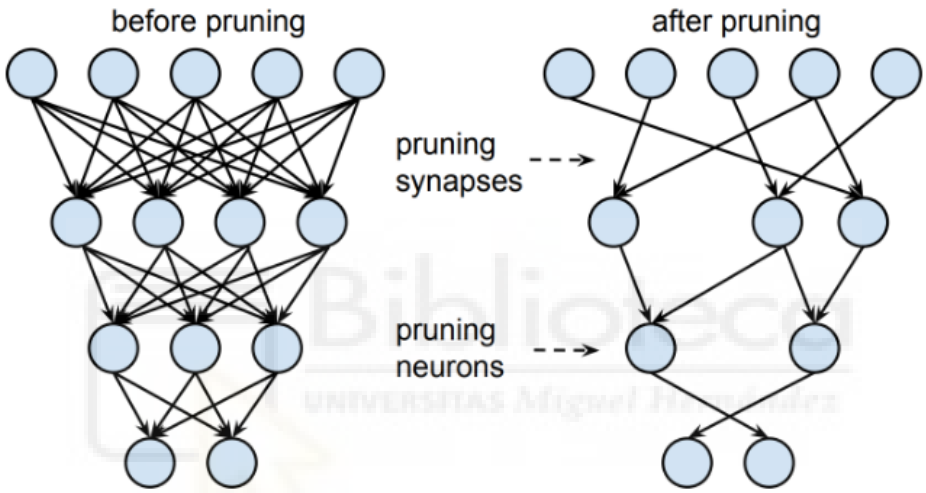

Figura 2.14: Ejemplo de pruning estructurado y no estructurado.

El pruning puede aplicarse antes, durante o después del entrenamiento, y suele combinarse con cuantización para maximizar la compresión.

2.5.5. Distillation

La destilación de conocimiento (*Knowledge Distillation*) consiste en entrenar un modelo pequeño (*student*) para imitar el comportamiento de un modelo grande (*teacher*), utilizando como etiquetas las salidas suaves (*soft targets*) del modelo original.

El proceso de destilación se realiza entrenando el modelo student para minimizar una combinación de dos funciones de pérdida: la pérdida estándar respecto a las

etiquetas reales (*hard targets*) y la pérdida respecto a las predicciones del modelo teacher (*soft targets*). Las predicciones suaves del teacher contienen información valiosa sobre la estructura del problema que facilita el aprendizaje del student.

- **Ventajas:** El modelo resultante es más pequeño y rápido, con una pérdida de precisión controlada. Permite obtener modelos hasta 10 veces más pequeños manteniendo gran parte del rendimiento.
- **Desventajas:** Requiere un proceso de entrenamiento adicional y acceso al modelo teacher durante todo el proceso.

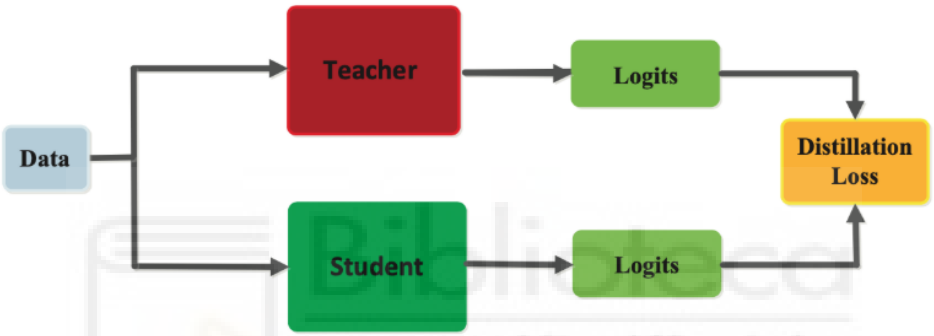

Figura 2.15: Ejemplo de distillation de conocimiento.

En tareas de generación de código, la distillation puede emplearse para entrenar modelos ligeros que repliquen el comportamiento de modelos más grandes como T5-Base o T5-Large.

2.5.5.1 Modelos Student para T5

Para la destilación a partir de T5-Small, pueden emplearse arquitecturas aún más reducidas como **t5-mini** o **t5-tiny**. Aunque estos modelos no forman parte del conjunto oficial de variantes publicadas por Google (que incluye T5-Small, T5-Base, T5-Large, T5-3B y T5-11B), existen implementaciones no oficiales disponibles en Hugging Face que mantienen la arquitectura fundamental de T5 pero con un número significativamente menor de parámetros.

Estas variantes ultra-compactas resultan especialmente adecuadas para:

- Dispositivos con recursos extremadamente limitados
- Aplicaciones educativas donde la latencia es crítica
- Despliegue en entornos embebidos o móviles
- Escenarios donde se prioriza la eficiencia sobre la precisión máxima

2.5.6. Distillation combinada con cuantización

Una estrategia especialmente efectiva para maximizar la compresión del modelo consiste en combinar la destilación de conocimiento con técnicas de cuantización. Este enfoque híbrido permite obtener modelos extremadamente compactos y eficientes manteniendo un rendimiento aceptable.

El proceso típico sigue las siguientes etapas:

1. **Selección del modelo student:** Se elige una arquitectura más pequeña que el modelo teacher. Por ejemplo, si el teacher es T5-Small (60M parámetros), el student podría ser t5-mini o t5-tiny.
2. **Entrenamiento por destilación:** Se entrena el modelo student utilizando las predicciones del teacher como supervisión adicional, combinando soft targets y hard targets.
3. **Cuantización post-entrenamiento:** Una vez completado el entrenamiento del student, se aplica Post-Training Quantization (PTQ) para reducir la precisión de los pesos de punto flotante de 32 bits a representaciones de 8 bits (INT8).
4. **Exportación a formato optimizado:** El modelo cuantizado se exporta a formatos especializados como ONNX, que permiten optimizaciones adicionales específicas del hardware de inferencia.

2.5.7. Exportación a ONNX INT8

ONNX (Open Neural Network Exchange) es un formato estándar abierto que permite la interoperabilidad entre diferentes frameworks de aprendizaje profundo y la optimización para diversos backends de inferencia. La exportación de modelos destilados y cuantizados a ONNX con precisión INT8 ofrece ventajas significativas:

- **Optimización para CPU:** Los modelos ONNX INT8 están especialmente optimizados para ejecución en procesadores convencionales, donde las operaciones enteras de 8 bits pueden acelerarse mediante instrucciones SIMD (como AVX-512 en procesadores Intel modernos).
- **Reducción de tamaño:** La cuantización a INT8 reduce el tamaño del modelo en aproximadamente 4 veces respecto a la versión en punto flotante de 32 bits (FP32).
- **Mejora en latencia:** Benchmarks realizados muestran que los modelos ONNX INT8 pueden alcanzar mejoras de velocidad de 2-4x en CPU respecto a versiones no cuantizadas, siendo especialmente eficientes en entornos donde no se dispone de GPU.
- **Portabilidad:** ONNX Runtime permite ejecutar el mismo modelo en múltiples plataformas (Windows, Linux, macOS, dispositivos embebidos) sin modificaciones.

Este enfoque combinado de distillation + cuantización + exportación a ONNX INT8 representa el estado del arte en compresión de modelos para despliegue en entornos con restricciones severas de recursos, como el contexto educativo abordado en este trabajo.

2.6. Métricas de Evaluación

La evaluación de modelos de generación automática de código requiere métricas específicas que permitan medir tanto la calidad sintáctica como la corrección funcional

de las secuencias generadas. En el contexto de este Trabajo Fin de Grado, donde se aborda la generación de código ensamblador en CODE-2 mediante modelos de lenguaje, resulta necesario combinar métricas cuantitativas y cualitativas para obtener una valoración completa del rendimiento de los modelos entrenados.

Las métricas seleccionadas permiten comparar distintas técnicas de fine-tuning y cuantización, así como analizar el compromiso entre precisión, eficiencia y capacidad de generalización.

2.7. Métricas cuantitativas

2.7.1. Training Loss y Validation Loss

La *Training Loss* y la *Validation Loss* miden el error cometido por el modelo durante las fases de entrenamiento y validación, respectivamente. Ambas se calculan a partir de la función de pérdida utilizada por el modelo, generalmente la entropía cruzada entre los tokens generados y los tokens de referencia.

El análisis conjunto de estas dos métricas permite:

- Evaluar la convergencia del proceso de entrenamiento.
- Detectar posibles problemas de sobreajuste, cuando la pérdida de entrenamiento disminuye mientras la de validación aumenta.
- Comparar la estabilidad de diferentes técnicas de fine-tuning.

2.7.2. Exact Match (EM)

La métrica de *Exact Match* evalúa el porcentaje de ejemplos para los cuales el código generado coincide exactamente con la referencia esperada. Se trata de una métrica estricta y binaria, que asigna un valor de 1 si ambas secuencias son idénticas y 0 en caso contrario.

Esta métrica resulta especialmente relevante en generación de código ensamblador, ya que pequeñas desviaciones sintácticas pueden impedir la correcta ejecución del programa.

2.7.3. BLEU (Bilingual Evaluation Understudy)

BLEU[15] es una métrica basada en la coincidencia de *n-gramas* entre la secuencia generada y la referencia. Tiene en cuenta la precisión de unigramas, bigramas y n-gramas de mayor orden, incorporando además un factor de penalización por secuencias excesivamente cortas (*brevity penalty*).

Aunque fue diseñada originalmente para traducción automática[15], BLEU se ha utilizado ampliamente en tareas de generación estructurada, incluyendo generación de código, por su capacidad para capturar similitudes locales en la secuencia.

2.7.4. ROUGE

ROUGE[11] es un conjunto de métricas centradas en el *recall*, es decir, en medir qué proporción del contenido de la referencia aparece en la salida generada. En este trabajo se emplean principalmente las siguientes variantes:

- **ROUGE-1**: coincidencia de unigramas.
- **ROUGE-2**: coincidencia de bigramas.
- **ROUGE-L**: basada en la subsecuencia común más larga (LCS).

ROUGE resulta útil para evaluar si el modelo ha capturado correctamente la estructura general del código, incluso cuando existen ligeras variaciones sintácticas.[11]

2.7.5. Tiempo de inferencia

El tiempo de inferencia mide la latencia necesaria para generar una salida a partir de una entrada, generalmente expresada en milisegundos por token o por secuencia

completa. Esta métrica es clave para evaluar la viabilidad del modelo en entornos con recursos limitados.

Se espera que técnicas eficientes como LoRA o Prompt-Tuning presenten menores tiempos de inferencia frente al fine-tuning completo, especialmente tras aplicar técnicas de cuantización.

2.8. Métricas cualitativas

2.8.1. Errores frecuentes en la generación

Las métricas cualitativas complementan el análisis numérico mediante una evaluación manual o semiautomática de la calidad del código generado.

Se analizan los errores más comunes producidos por los modelos, tales como:

- Uso incorrecto de registros.
- Selección errónea de instrucciones aritméticas o lógicas.
- Secuencias incompletas o mal ordenadas.

Este análisis permite identificar patrones de fallo y comprender las limitaciones del modelo.

2.8.2. Legibilidad y corrección sintáctica

Se evalúa si el código generado es legible, coherente y sintácticamente correcto dentro del lenguaje CODE-2. Un código legible y bien estructurado es especialmente importante en contextos educativos, donde el objetivo no es solo la ejecución correcta, sino también la comprensión por parte del estudiante.

2.8.3. Generalización frente a memorización

Se estudia la capacidad del modelo para generalizar a ejemplos no vistos durante el entrenamiento, evitando la simple memorización del conjunto de datos. Para ello, se analizan ejemplos del conjunto de test que presentan estructuras diferentes a las del entrenamiento y validación.

2.9. Uso combinado de métricas

Dado que ninguna métrica individual es suficiente para evaluar completamente la calidad del código generado, en este trabajo se adopta un enfoque combinado. Las métricas cuantitativas permiten comparaciones objetivas, mientras que las cualitativas aportan una visión más práctica y funcional.

Este enfoque conjunto garantiza una evaluación equilibrada del rendimiento de los modelos y facilita la selección de la metodología más adecuada para la tarea propuesta.

3. METODOLOGÍA

3.1. Generación del dataset sintético para CODE-2

El correcto entrenamiento de modelos de generación de código requiere un conjunto de datos representativo, diverso y alineado con la estructura del lenguaje objetivo. En este Trabajo Fin de Grado se ha diseñado un dataset sintético adaptado específicamente al lenguaje ensamblador educativo **CODE-2**.

La decisión de generar un dataset sintético surge de la inexistencia de bases de datos públicas suficientemente grandes y de calidad para CODE-2. Asimismo, la generación manual de ejemplos resulta inviable debido a su elevado coste temporal y su limitada escalabilidad. Por ello, se optó por desarrollar un generador automático de ejemplos sintácticamente válidos implementado en **Python**, capaz de producir grandes volúmenes de datos de forma controlada y reproducible.

3.1.1. Diseño del dataset

El dataset se construyó a partir de pares entrada–salida, donde la entrada consiste en una descripción en lenguaje natural y la salida corresponde a una o varias instrucciones en ensamblador CODE-2. Dado el carácter educativo del proyecto, se decidió restringir inicialmente el aprendizaje del modelo a un subconjunto representativo de **cuatro instrucciones**: `LD`, `ST`, `ADDS` y `SUBS`. Estas instrucciones permiten cubrir operaciones fundamentales de carga, almacenamiento y aritmética básica.

3.1.1.1 Ejemplos representativos

Entrada en lenguaje natural	**Salida esperada en CODE-2**
Carga el valor 5 en el registro R1	`LD R1, 5`
Carga el valor 8 en R2 y guárdalo en memoria	`LD R2, 8` `ST R2, [MEM]`
Suma el contenido de R1 y R2 y guarda el resultado en R3	`ADDS R3, R1, R2`
Resta el contenido de R2 a R1 y almacena el resultado en R0	`SUBS R0, R1, R2`
Carga un valor en R1 y súmale el contenido de R2	`LD R1, 4` `ADDS R1, R1, R2`

Tabla 3.1: Ejemplos representativos de pares entrada/salida utilizados en el dataset sintético

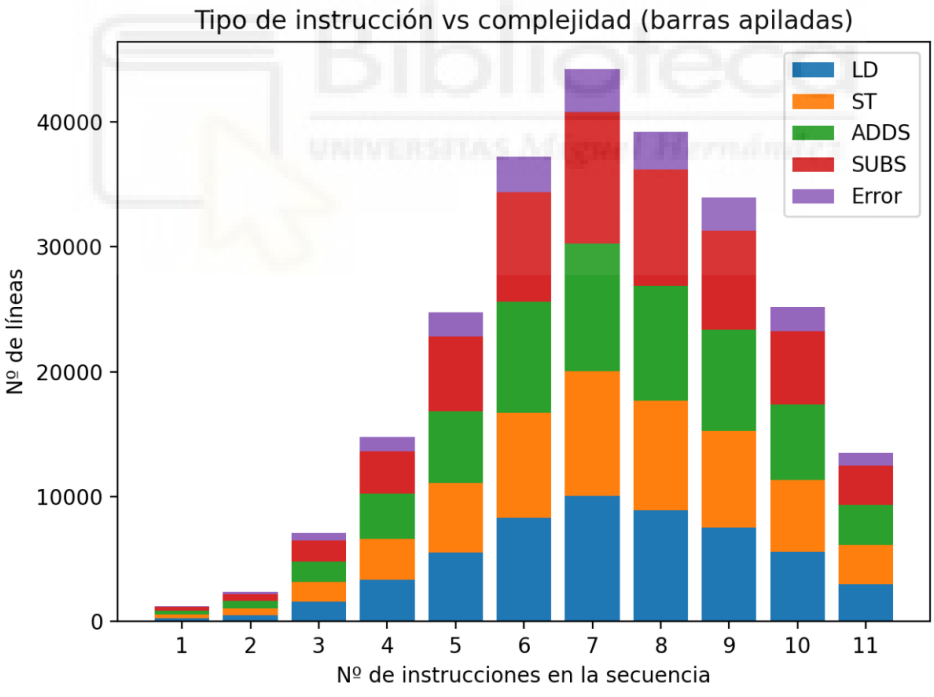

Figura 3.1: Distribución del dataset sintético por tipo de instrucción y complejidad de secuencias.

Las principales decisiones de diseño del dataset fueron las siguientes:

- **Cobertura estructural:** Se emplearon plantillas parametrizadas para garan-

tizar una distribución equilibrada entre las instrucciones seleccionadas.

- **Variabilidad en longitud:** Los bloques generados contienen entre 1 y 11 instrucciones, con una media de entre 5 y 7 líneas por ejemplo.
- **Errores controlados:** Se incluyó un porcentaje inicial del 10 % de ejemplos con errores sintácticos intencionados (por ejemplo, registros fuera de rango), con el objetivo de evaluar la robustez del modelo. Este porcentaje se incrementó posteriormente al 20 % en el dataset ampliado de 36k ejemplos para mejorar la capacidad de generalización a entradas malformadas.
- **Evitar duplicados:** Se implementó un sistema de control de unicidad mediante claves únicas generadas a partir del par entrada–salida. Este mecanismo garantiza que el 100 % de los ejemplos del dataset sean únicos y evita la memorización de ejemplos concretos por parte del modelo.
- **Tamaño:** Se generaron **18 000** ejemplos: 12 600 para entrenamiento (70 %), 3 600 para validación (20 %) y 1 800 para test (10 %), garantizando que los subconjuntos no presenten solapamiento. Esta separación estricta es fundamental para asegurar la validez de las métricas de evaluación, ya que la presencia de ejemplos comunes en los conjuntos podría conducir a resultados artificialmente optimistas y, por tanto, a conclusiones erróneas.

Código 3.1: Control de duplicados mediante claves únicas

```
key = input_text + "|" + output_text
if key not in seen:
    seen.add(key)
    dataset.append(example)
```

3.1.2. Preprocesamiento

Antes de utilizar el dataset para el entrenamiento del modelo T5-Small, se aplicó una fase de preprocesamiento con el objetivo de preservar la estructura del código y asegurar la compatibilidad con el tokenizador.

Las principales etapas de este proceso fueron:

- **Tokenización:** Se utilizó un tokenizador compatible con T5, configurado con estrategias de `padding` y `truncation`[9].
- **Preservación del formato multilínea:** Los saltos de línea reales (`\n`) se conservaron explícitamente, ya que representan separaciones semánticas entre instrucciones.
- **Separación de conjuntos:** Se mantuvo una separación estricta entre entrenamiento, validación y test para evitar cualquier tipo de fuga de información.

Código 3.2: Preservación del formato multilínea durante el preprocesamiento

```
text_for_tokenize = text
if preserve_newlines:
    # Sustituye saltos reales por un marcador visible
    text_for_tokenize = text_for_tokenize.replace("\n", " SALTO ")

encoded = tokenizer(
    text_for_tokenize,
    max_length=max_length,
    truncation=max_length is not None,
    add_special_tokens=show_special,
)

decoded = tokenizer.decode(encoded["input_ids"],
    ↪ skip_special_tokens=not show_special)
if preserve_newlines:
    # Restaura los saltos de linea
    decoded = decoded.replace("SALTO", "\n")
```

Con el fin de verificar que el tokenizador procesa correctamente código ensamblador multilínea, se realizaron pruebas explícitas sobre ejemplos reales del dataset.

Código 3.3: Verificación de la tokenización de código multilínea

```
example_out = "ADDS rD,r1,r2\nST [rD+H'45'],rD"
tokens = tokenizer.tokenize(example_out)
```

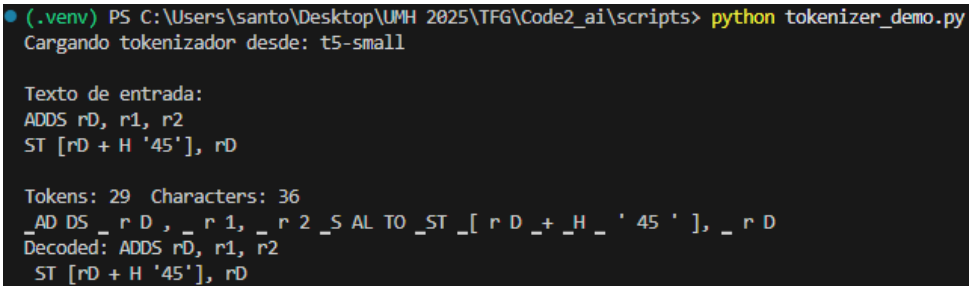

Figura 3.2: Demostración de la tokenización de código ensamblador multilínea utilizando el tokenizador de T5.

Finalmente, la Figura 3.3 resume visualmente el flujo completo seguido para la generación, validación y preprocesamiento del dataset sintético empleado en este trabajo.

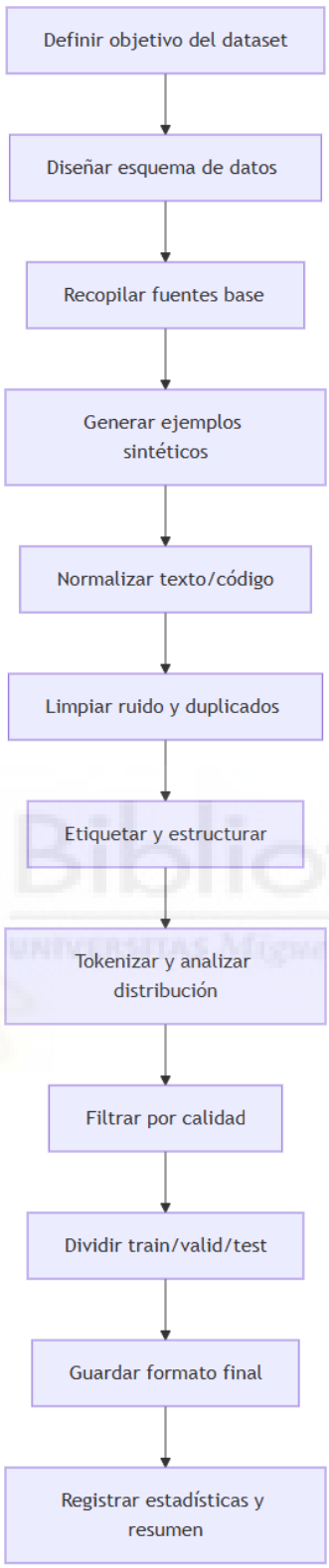

Figura 3.3: Proceso de generación y preprocesamiento del dataset sintético para CODE-2.

3.2. Configuración del entorno de entrenamiento

El entrenamiento y evaluación de modelos de lenguaje de tipo Transformer implica un elevado coste computacional, especialmente en términos de memoria y tiempo de ejecución. Por este motivo, la selección del entorno de entrenamiento constituye una decisión metodológica clave dentro de este Trabajo Fin de Grado.

Para el desarrollo del proyecto se evaluaron dos entornos de ejecución principales: un **PC local** y la plataforma **Google Colab**. Ambos entornos fueron analizados experimentalmente con el objetivo de determinar cuál ofrecía un mejor equilibrio entre rendimiento, estabilidad y control de recursos.

3.2.1. Infraestructura local

El entorno local se utilizó de forma intensiva a lo largo de todo el proyecto, tanto para tareas de desarrollo como para experimentación y entrenamiento. Las características del equipo empleado son las siguientes:

- **CPU:** Intel Core i5 de 10ª generación.
- **GPU:** NVIDIA RTX 2060 con 6 GB de VRAM.
- **Memoria RAM:** 16 GB.
- **Sistema Operativo:** Windows 11.
- **Editor de Código:** Visual Studio Code.

Este entorno permitió un control total sobre el hardware, sin restricciones de tiempo de uso ni dependencia de servicios externos. El uso exclusivo de la GPU resultó especialmente relevante para obtener medidas estables y reproducibles durante las pruebas de rendimiento.

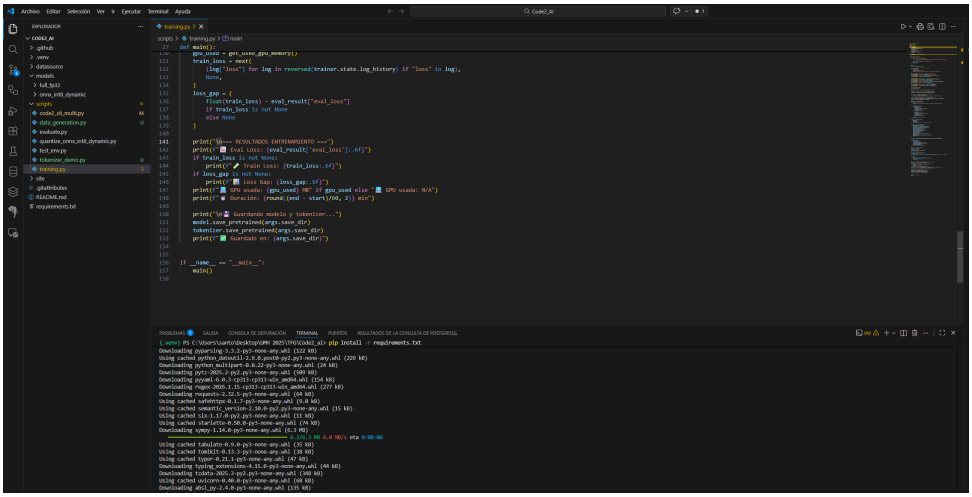

Figura 3.4: Entorno de desarrollo en Visual Studio Code con GPU local.

No obstante, la principal limitación del entorno local es la cantidad de VRAM disponible, lo que condiciona el tamaño del modelo, el *batch size* y la necesidad de aplicar técnicas de optimización de memoria, como `fp16` o *gradient checkpointing*.

3.2.2. Infraestructura Google Colab

Como alternativa al entorno local, se evaluó el uso de Google Colab en su versión gratuita, que ofrece acceso a GPUs más potentes en términos de memoria. Las características principales de este entorno son:

- **GPU:** NVIDIA Tesla T4 con 16 GB de VRAM.
- **Memoria RAM:** entre 12 y 25 GB, dependiendo de la sesión.
- **Entorno:** ejecución remota con GPU compartida.

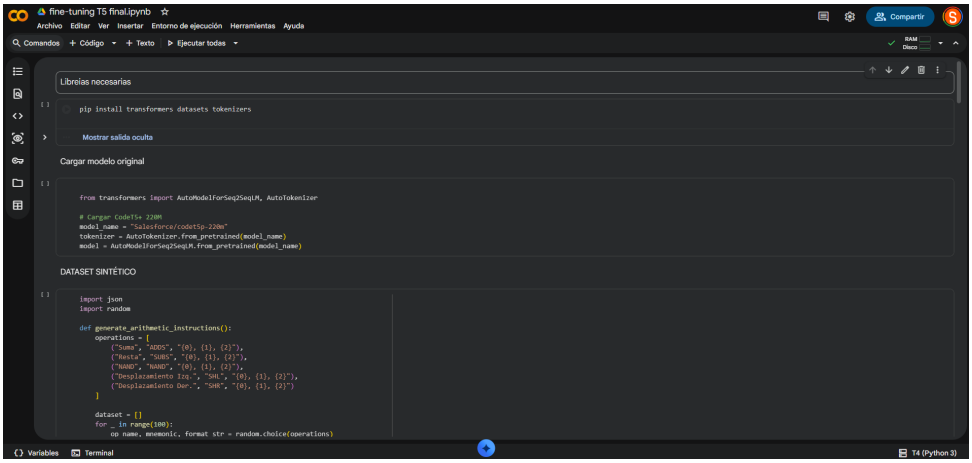

Figura 3.5: Entorno de ejecución en Google Colab con GPU Tesla T4.

Aunque Colab ofrece mayor capacidad de memoria gráfica, presenta importantes limitaciones prácticas, como la duración máxima de las sesiones (12 horas), posibles períodos de espera tras agotar el uso de GPU y una menor estabilidad derivada del uso compartido del hardware.

3.2.3. Comparativa experimental PC vs Colab

Para seleccionar el entorno definitivo, se realizó una comparación experimental utilizando el mismo código de entrenamiento, los mismos hiperparámetros y un subconjunto idéntico del dataset (1000 ejemplos del conjunto ASMBasic). El modelo evaluado fue **T5-Small**, con cinco épocas de entrenamiento.

Los resultados obtenidos se resumen en la Tabla 3.2.

Métrica	**PC Local(RTX 2060)**	**Google Colab(Tesla T4)**
Training Loss (global)	1.5081	1.5078
Validation Loss (última epoch)	0.8397	0.8401
Tiempo total (5 epochs)	146.4 s	143.9 s
Samples por segundo	31.48	32.01
Steps por segundo	7.89	8.02
Eval Samples/s	103.72	118.43
Eval Steps/s	25.93	29.61

Tabla 3.2: Comparativa de rendimiento entre PC local y Google Colab usando T5-Small.

Los resultados muestran que ambos entornos ofrecen un rendimiento prácticamente

equivalente en términos de precisión y tiempo de entrenamiento. Las diferencias observadas no son significativas desde el punto de vista del rendimiento del modelo.

3.2.4. Evaluación preliminar con T5-Base

Con el objetivo de analizar la viabilidad de utilizar modelos de mayor capacidad, se realizaron pruebas preliminares utilizando el modelo **T5-Base**, que cuenta con aproximadamente 220 millones de parámetros, frente a los 60 millones de T5-Small.

Estas pruebas se llevaron a cabo tanto en el entorno local como en Google Colab, empleando un subconjunto reducido del dataset, con el fin de evaluar el consumo de memoria, la estabilidad del entrenamiento y los tiempos de ejecución.

Los experimentos mostraron que, si bien T5-Base es capaz de ejecutarse correctamente, su entrenamiento presenta limitaciones significativas en el entorno local debido al consumo de memoria gráfica. En particular, fue necesario reducir de forma notable el tamaño de batch y aplicar técnicas adicionales de optimización para evitar errores de desbordamiento de memoria (*out-of-memory*).

En Google Colab, aunque la mayor cantidad de VRAM permitió una ejecución más estable, las restricciones temporales de las sesiones y la duración de los entrenamientos dificultaron la repetición sistemática de experimentos y la exploración exhaustiva de hiperparámetros.

Como resultado de esta evaluación preliminar, se concluyó que el uso de T5-Base no ofrecía una mejora proporcional al incremento de coste computacional en el contexto de este trabajo. Por este motivo, se optó por utilizar **T5-Small** como modelo base, permitiendo realizar un mayor número de experimentos, aplicar técnicas de fine-tuning eficientes y garantizar la reproducibilidad del estudio.

3.2.5. Justificación del entorno seleccionado

A pesar de que Google Colab dispone de mayor VRAM, los experimentos realizados demuestran que el **PC local ofrece un rendimiento comparable** para el tamaño

de modelo y dataset utilizados en este proyecto. Además, el entorno local presenta ventajas decisivas:

- Ausencia de limitaciones temporales en las sesiones.
- Uso exclusivo y estable de la GPU.
- Mayor control sobre el entorno software y hardware.
- Reproducibilidad completa de los experimentos.

Por estos motivos, se decidió utilizar el **PC local como entorno principal** para el entrenamiento final del modelo T5-Small sobre el dataset completo de CODE-2. Google Colab se empleó únicamente como entorno de comparación y validación experimental.

Esta decisión garantiza un flujo de trabajo estable, reproducible y alineado con los objetivos del proyecto.

3.2.6. Librerías y dependencias

El entorno software se configuró utilizando principalmente el ecosistema de Python, debido a su amplia adopción en tareas de aprendizaje automático y procesamiento del lenguaje natural. Las librerías empleadas se seleccionaron atendiendo a criterios de estabilidad, documentación, soporte activo y compatibilidad con modelos basados en arquitecturas Transformer.

Figura 3.6: Lenguaje de programación Python utilizado para el desarrollo del proyecto.

Todas las versiones de las librerías se fijaron explícitamente con el objetivo de garantizar la reproducibilidad de los experimentos, y el entorno se gestionó mediante scripts automatizados y archivos de configuración.

3.2.6.1 PyTorch

PyTorch es el framework de aprendizaje profundo utilizado como backend principal para el entrenamiento y la ejecución de los modelos neuronales empleados en este trabajo.[17]

Figura 3.7: Framework PyTorch utilizado como backend de aprendizaje profundo.

3.2.6.2 Hugging Face Transformers

La librería Hugging Face Transformers se empleó para la carga del modelo T5-Small, la configuración de los procesos de fine-tuning y la generación de código durante la fase de inferencia.[7]

Figura 3.8: Librería Hugging Face Transformers empleada para la carga, entrenamiento e inferencia del modelo T5.

3.2.6.3 Hugging Face Datasets

La librería Datasets se utilizó para la gestión eficiente de los conjuntos de entrenamiento y validación, facilitando la carga, partición y preprocesamiento de los datos de forma estructurada.

Figura 3.9: Librería Datasets utilizada para la gestión eficiente de los conjuntos de datos.

3.2.6.4 SentencePiece

SentencePiece es una librería de tokenización desarrollada por Google y ampliamente utilizada en modelos de lenguaje basados en arquitecturas Transformer, como T5, BERT o mT5. En este trabajo desempeña un papel fundamental en el preprocesamiento del dataset y en la correcta representación del código ensamblador CODE-2.

A diferencia de otros métodos de tokenización dependientes de espacios en blanco, SentencePiece opera a nivel de subpalabras y caracteres, lo que resulta especialmente adecuado para la tokenización de código, donde aparecen símbolos, registros y constantes que no siguen la estructura del lenguaje natural.

Entre las principales ventajas de SentencePiece en este contexto se encuentran:

- Independencia de los espacios en blanco.
- Soporte nativo de caracteres especiales y operadores.
- Compatibilidad directa con el tokenizador utilizado por el modelo T5.
- Generación de vocabularios eficientes basados en subpalabras.

Un aspecto relevante de SentencePiece es el uso del carácter especial _ (subrayado) para representar los espacios en blanco, permitiendo preservar explícitamente la estructura original del texto y del código ensamblador.

Código 3.4: Ejemplo de tokenización con SentencePiece en T5

```
from transformers import T5Tokenizer

tokenizer = T5Tokenizer.from_pretrained("t5-small")
texto = "Suma r1 y r2 en el registro r3"
tokens = tokenizer.tokenize(texto)
ids = tokenizer.encode(texto)
```

Figura 3.10: Herramienta SentencePiece empleada como método de tokenización subword en el modelo T5.

3.2.6.5 PEFT

La librería PEFT (Parameter-Efficient Fine-Tuning) se empleó para la aplicación de técnicas de ajuste eficiente de parámetros, como LoRA, permitiendo reducir el número de parámetros entrenables y el consumo de recursos computacionales.

Figura 3.11: Librería PEFT utilizada para la aplicación de técnicas de fine-tuning eficiente como LoRA.

3.2.6.6 NumPy y Pandas

Las librerías NumPy y Pandas se utilizaron para el manejo de datos, análisis de resultados experimentales y cálculo de métricas durante las fases de evaluación.

Figura 3.12: Librerías NumPy y Pandas empleadas para el tratamiento de datos y análisis de métricas.

Esta configuración software proporciona una base estable, coherente y reproducible para la ejecución de los experimentos descritos en los apartados posteriores.

3.3. Estrategias de fine-tuning aplicadas

Una vez definido el modelo base y generado el conjunto de datos sintético para CODE-2, se procedió a la aplicación y evaluación de distintas estrategias de fine-tuning. El objetivo principal de esta fase fue analizar el comportamiento práctico de cada metodología en términos de estabilidad, consumo de recursos y capacidad de adaptación al dominio de generación de código ensamblador.

Dado que el entrenamiento se realizó en una infraestructura con recursos limitados, concretamente una GPU NVIDIA RTX 2060 con 6 GB de VRAM, la selección de estrategias debía equilibrar el rendimiento obtenido con la viabilidad computacional del proceso de entrenamiento.

3.3.1. Aplicación de Prompt Tuning

La técnica de Prompt Tuning se utilizó como línea base experimental, permitiendo evaluar la capacidad del modelo T5-Small para adaptarse a la tarea de generación de código CODE-2 sin modificar sus parámetros internos.

La implementación se realizó mediante la librería PEFT, entrenando únicamente los parámetros asociados al prompt virtual. Desde el punto de vista computacional, esta metodología resultó altamente eficiente, con tiempos de entrenamiento reducidos y

un consumo de memoria mínimo.

Sin embargo, los resultados obtenidos mostraron que el modelo no fue capaz de aprender de forma efectiva las reglas sintácticas y estructurales del lenguaje ensamblador CODE-2. La pérdida de validación permaneció elevada y las secuencias generadas presentaron errores frecuentes, como instrucciones incompletas o un uso incorrecto de registros.

En consecuencia, Prompt Tuning fue descartado como opción para el entrenamiento final, aunque se considera útil como referencia inicial y como técnica válida en tareas donde el modelo base ya dispone de un conocimiento más profundo del dominio objetivo.

3.3.2. Aplicación de LoRA

La estrategia LoRA se aplicó al modelo T5-Small con el objetivo de evaluar una alternativa eficiente al ajuste completo de parámetros. Para ello, se introdujeron adaptadores de bajo rango en las capas de atención del modelo, manteniendo congelados los pesos originales.

Desde el punto de vista práctico, LoRA permitió entrenar el modelo de forma estable con un consumo de memoria significativamente inferior al del Full Fine-Tuning. Esta reducción de coste computacional facilitó la realización de múltiples experimentos, así como la exploración de distintos hiperparámetros.

Los resultados obtenidos mostraron una convergencia clara tanto en la pérdida de entrenamiento como en la de validación, alcanzando valores comparables a los obtenidos mediante ajuste completo del modelo. Además, el tiempo total de entrenamiento fue considerablemente menor.

Durante el proceso de evaluación se detectó la necesidad de fusionar explícitamente los adaptadores LoRA con el modelo base para obtener correctamente las métricas de validación, debido a limitaciones del framework utilizado. Este paso adicional permitió garantizar una comparación justa con el resto de metodologías.

3.3.3. Pruebas con Adapters

La técnica basada en Adapters fue considerada inicialmente como una posible alternativa de fine-tuning eficiente. No obstante, durante la fase de implementación se identificaron problemas significativos de compatibilidad con las versiones actuales de la librería Hugging Face Transformers.

A pesar de diversos intentos de configuración, no fue posible ejecutar correctamente el entrenamiento mediante Adapters clásicos. Dado que esta metodología se encuentra actualmente en desuso y que Hugging Face recomienda el uso de la librería PEFT como estándar para fine-tuning eficiente, se decidió descartar esta estrategia del análisis experimental final.

3.3.4. Aplicación de Full Fine-Tuning

El Full Fine-Tuning se aplicó ajustando todos los parámetros del modelo T5-Small sobre el dataset sintético generado. Este enfoque se utilizó como referencia superior en términos de capacidad de adaptación y precisión.

Se llevaron a cabo pruebas tanto con como sin optimización de memoria mediante *gradient checkpointing*, así como entrenamiento en precisión mixta (FP16). Los experimentos mostraron que el ajuste completo del modelo alcanzó los valores más bajos de pérdida de entrenamiento y validación.

No obstante, este enfoque presentó un mayor consumo de memoria y tiempos de entrenamiento considerablemente superiores a los obtenidos con LoRA. En el caso concreto del modelo T5-Small, se observó que el uso de *gradient checkpointing* no era estrictamente necesario y podía penalizar ligeramente el tiempo total de entrenamiento.

3.3.5. Comparación preliminar de métricas entre metodologías

Con el objetivo de obtener una primera visión comparativa del impacto de cada estrategia de fine-tuning, se recopilaron las principales métricas de entrenamiento y validación obtenidas durante las pruebas experimentales iniciales. Esta comparación preliminar permite analizar el compromiso entre precisión, estabilidad y coste computacional asociado a cada metodología, sin que ello implique todavía una selección definitiva.

Las métricas consideradas incluyen la pérdida de entrenamiento (*Training Loss*), la pérdida de validación (*Validation Loss*) y el tiempo total de entrenamiento. Siempre que fue posible, estas métricas se obtuvieron utilizando el mismo subconjunto de datos y condiciones de entrenamiento, con el fin de garantizar una comparación lo más justa posible.

Metodología	**Training Loss**	**Validation Loss**	**Tiempo de entrenamiento**
Prompt Tuning	Alto	Alto	Muy bajo
LoRA	Bajo	Bajo	Medio
Full Fine-Tuning	Muy bajo	Muy bajo	Alto

Tabla 3.3: Comparativa cualitativa preliminar de métricas obtenidas para las distintas estrategias de fine-tuning.

Los resultados muestran que Prompt Tuning, aunque computacionalmente eficiente, no logra una reducción significativa de la pérdida de validación, lo que indica una capacidad de adaptación limitada al lenguaje ensamblador CODE-2. En contraste, tanto LoRA como Full Fine-Tuning alcanzan valores bajos de pérdida, reflejando una mejor capacidad de aprendizaje bajo las configuraciones evaluadas.

Si bien Full Fine-Tuning ofrece el mejor rendimiento en términos de pérdida, este se obtiene a costa de un mayor consumo de recursos y tiempos de entrenamiento más elevados. Por su parte, LoRA presenta resultados comparables en validación con un coste computacional considerablemente menor.

No obstante, estas observaciones iniciales no resultan suficientes para seleccionar de forma concluyente una metodología. Por ello, se decidió proceder a una evaluación cuantitativa más detallada y a una fase específica de optimización de hiperparáme-

tros, con el fin de determinar cuál de las dos estrategias principales (LoRA o Full Fine-Tuning) ofrece el mejor equilibrio entre precisión y eficiencia en el contexto del problema planteado.

3.3.6. Evaluación cuantitativa previa a la optimización

Antes de iniciar la fase de optimización de hiperparámetros, se llevó a cabo una evaluación cuantitativa más detallada con el objetivo de comparar el rendimiento de las dos metodologías más prometedoras: LoRA y Full Fine-Tuning. Esta prueba se realizó utilizando configuraciones estándar para ambas técnicas, sin aplicar todavía ningún proceso sistemático de ajuste fino de hiperparámetros.

El objetivo de esta evaluación intermedia fue comprobar si las diferencias observadas en la comparación preliminar se mantenían al utilizar métricas cuantitativas más precisas, y determinar si alguna de las metodologías destacaba de forma clara antes de proceder a una optimización más costosa computacionalmente.

Para ello, se evaluaron ambos modelos sobre el mismo conjunto de validación, empleando métricas de pérdida y calidad de generación descritas en el Capítulo 2.6. Los resultados obtenidos mostraron que tanto LoRA como Full Fine-Tuning alcanzaban valores similares en las métricas consideradas, sin que ninguna de las dos metodologías presentase una ventaja concluyente bajo estas condiciones iniciales.

Esta ausencia de una diferencia clara reforzó la hipótesis de que el rendimiento final de ambas estrategias dependía de forma significativa de la configuración de sus hiperparámetros. En consecuencia, se decidió no descartar ninguna de ellas en esta fase y proceder a una etapa específica de optimización paramétrica para ambas metodologías, descrita en el Apartado 3.4.

3.4. Optimización de hiperparámetros

Tras la evaluación preliminar de las distintas estrategias de fine-tuning y la comparación cuantitativa inicial entre LoRA y Full Fine-Tuning, se constató que ambas

metodologías ofrecían un rendimiento competitivo bajo configuraciones estándar. No obstante, dichas configuraciones no permitían explotar completamente el potencial de los modelos ni garantizar un comportamiento óptimo en el entorno hardware disponible.

Los hiperparámetros juegan un papel fundamental en el proceso de entrenamiento de modelos Transformer, ya que influyen directamente tanto en la velocidad de convergencia como en la capacidad de generalización del modelo. Pequeñas variaciones en parámetros como la tasa de aprendizaje, el tamaño del lote o el número de épocas pueden provocar diferencias significativas en los resultados obtenidos.

Dado que el entrenamiento se realizaba en una GPU NVIDIA RTX 2060 con 6 GB de memoria, resultaba imprescindible ajustar cuidadosamente estos hiperparámetros para maximizar el rendimiento del modelo sin exceder las limitaciones de memoria y tiempo de entrenamiento. Por este motivo, se diseñó una fase específica de optimización de hiperparámetros aplicada tanto a LoRA como a Full Fine-Tuning, con el objetivo de identificar las configuraciones más adecuadas para cada metodología.

Figura 3.13: GPU local utilizada para la optimización de hiperparámetros en las estrategias de fine-tuning.

3.4.1. Motivación de la optimización

Aunque las pruebas iniciales mostraron que LoRA y Full Fine-Tuning eran capaces de generar código ensamblador CODE-2 de forma correcta, los resultados obtenidos no eran concluyentes para seleccionar una metodología definitiva. En particular, se observó que ambas estrategias presentaban comportamientos similares en términos de pérdida de validación, pero con diferencias apreciables en estabilidad, consumo de recursos y tiempo de entrenamiento.

Además, el uso de configuraciones genéricas recomendadas en la literatura o en ejemplos de referencia no garantizaba un rendimiento óptimo en el contexto específico de este Trabajo Fin de Grado, ni para el tamaño del dataset ni para la infraestructura utilizada.

Por tanto, se planteó la necesidad de llevar a cabo una optimización sistemática de los hiperparámetros, con el fin de:

- Mejorar la capacidad de generalización del modelo.
- Reducir el riesgo de sobreajuste.
- Maximizar el rendimiento bajo restricciones de hardware.
- Comparar de forma justa LoRA y Full Fine-Tuning bajo sus mejores configuraciones posibles.

3.4.2. Espacio de búsqueda de hiperparámetros

El proceso de optimización se centró en un conjunto de hiperparámetros clave que influyen directamente en el comportamiento del entrenamiento y la capacidad de generalización del modelo. El espacio de búsqueda se definió teniendo en cuenta tanto la experiencia previa como las limitaciones impuestas por el hardware disponible.

Para ambas metodologías se consideraron los siguientes hiperparámetros:

- **Tasa de aprendizaje (learning rate):** Controla la magnitud de las actualizaciones de los pesos durante el entrenamiento.
- **Número de épocas:** Determina cuántas veces el modelo recorre el conjunto de entrenamiento.
- **Tamaño del lote (batch size):** Limitado por la memoria disponible en la GPU.
- **Weight decay:** Introducido como mecanismo de regularización.

En el caso específico de LoRA, se incluyeron además hiperparámetros propios de esta técnica:

- **Rango r:** Dimensión de las matrices de bajo rango introducidas.
- **Factor de escalado α:** Controla la contribución de los adaptadores LoRA.

Este espacio de búsqueda permitió explorar configuraciones diversas, manteniendo un equilibrio entre exhaustividad y viabilidad computacional.

3.4.3. Protocolo experimental de optimización

La optimización de hiperparámetros se llevó a cabo mediante un protocolo experimental iterativo, basado en la ejecución de múltiples entrenamientos independientes con distintas combinaciones de hiperparámetros. Cada combinación evaluada se consideró una prueba experimental individual, permitiendo analizar de forma sistemática el impacto de cada parámetro sobre el rendimiento final del modelo.

Para cada ejecución se mantuvieron constantes el conjunto de datos, el modelo base y la metodología de fine-tuning, de modo que las diferencias observadas pudieran atribuirse exclusivamente a la variación de los hiperparámetros. Durante el entrenamiento se registraron métricas de pérdida tanto en entrenamiento como en validación, así como el tiempo total de ejecución y el consumo de memoria.

Este proceso se aplicó de forma independiente a LoRA y Full Fine-Tuning, permitiendo comparar posteriormente ambas metodologías bajo sus configuraciones optimizadas y no únicamente bajo valores por defecto.

3.4.4. Métrica de optimización y criterio de selección

La métrica principal empleada para guiar el proceso de optimización fue la pérdida de validación (*Validation Loss*), al ser un indicador directo de la capacidad de generalización del modelo. No obstante, esta métrica se analizó siempre en conjunto con la pérdida de entrenamiento, con el objetivo de detectar posibles fenómenos de sobreajuste.

Adicionalmente, se introdujo el análisis del *loss gap*, definido como la diferencia entre la pérdida de validación y la pérdida de entrenamiento:

$$\text{Loss Gap} = \text{Training Loss} - \text{Validation Loss}$$

Esta métrica permitió evaluar de forma más precisa el equilibrio entre ajuste al conjunto de entrenamiento y generalización a datos no vistos. Configuraciones con una pérdida de validación baja pero un gap excesivamente reducido o inestable fueron consideradas potencialmente problemáticas, ya que podían indicar memorización del conjunto de entrenamiento.

Por tanto, la selección de las configuraciones óptimas no se basó únicamente en minimizar la pérdida de validación, sino en encontrar un compromiso adecuado entre:

- Baja pérdida de validación.
- Estabilidad del loss gap entre ejecuciones.
- Tiempo de entrenamiento razonable.
- Uso eficiente de la memoria disponible.

3.4.5. Automatización de la optimización con Optuna

Con el fin de sistematizar el proceso de optimización de hiperparámetros y reducir la dependencia de ajustes manuales, se empleó la librería Optuna[2] como herramienta de búsqueda automática. Optuna permitió explorar de forma eficiente el espacio de hiperparámetros definido, basándose tanto en los resultados obtenidos en ejecuciones previas como en los valores observados durante las pruebas manuales iniciales.

Figura 3.14: Librería Optuna utilizada para la optimización automática de hiperparámetros.

En el script de automatización (`scripts/training.py`), el objetivo de optimización se definió como la **minimización de la pérdida de validación** (*Validation Loss*), utilizando el valor `eval_loss` devuelto por la evaluación del modelo. Cada prueba (*trial*) entrenó el modelo con una configuración distinta y devolvió dicha pérdida como métrica objetivo.

El proceso incorporó además *pruning* mediante `MedianPruner`, lo que permitió descartar automáticamente ejecuciones poco prometedoras y acelerar la búsqueda. Como resultado, se generó una base de datos estructurada con métricas de entrenamiento, validación, *loss gap*, tiempos de ejecución y consumo de GPU, facilitando el análisis posterior y la selección de configuraciones óptimas.

Figura 3.15: Ejemplo de base de datos generada durante la optimización automática de hiperparámetros con Optuna.

Hiperparámetro	**Full Fine-Tuning**	**LoRA**
Batch Size	[4, 8, 16, 32]	[4, 8, 16, 32]
Gradient Accumulation Steps	[1, 2, 4, 8]	[1, 2, 4, 8]
Learning Rate	[1e-4, 3e-4]	[1e-4, 3e-4]
Weight Decay	[0, 1e-3]	[0, 1e-3]
Number of Epochs	[3]	[3]
LoRA Specific		
LoRA Rank (r)	-	[4, 8, 16, 64]
LoRA Alpha	-	[16, 32, 64, 128, 256, 512, 1024]
LoRA dropout	-	[0, 0.3]

Tabla 3.4: Rango de búsqueda para los hiperparámetros de Full Fine-Tuning y LoRA.

La integración de Optuna permitió acelerar significativamente la fase de optimización y garantizar que las configuraciones finales seleccionadas representasen un equilibrio óptimo entre rendimiento, estabilidad y coste computacional.

3.4.6. Análisis del loss gap durante la optimización

Durante la fase de optimización se analizó la evolución del loss gap a lo largo de distintas ejecuciones experimentales, tanto para LoRA como para Full Fine-Tuning. Este análisis permitió observar cómo determinadas configuraciones de hiperparámetros influían en la estabilidad del entrenamiento y en la capacidad de generalización

del modelo.

Las Figuras 3.16 y 3.17 muestran ejemplos representativos de la evolución del loss gap en los ensayos de Optuna. En ellas se observa una fase inicial con mayor variabilidad y, a medida que avanzan los ensayos, una estabilización del gap para las configuraciones mas prometedoras.

Estos plots corresponden a los ensayos de Optuna y no al ajuste manual posterior.

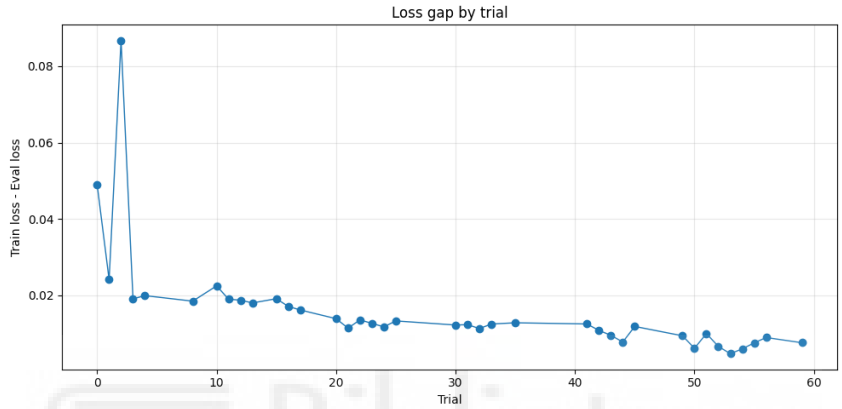

Figura 3.16: Evolución del gap entre Training Loss y Validation Loss durante la optimización de hiperparámetros para Full Fine-Tuning.

Figura 3.17: Evolución del gap entre Training Loss y Validation Loss durante la optimización de hiperparámetros para LoRA.

Los resultados evidenciaron una fase inicial con mayor dispersión en ambos métodos, seguida de una estabilización clara del gap. En los ensayos con mejor rendimiento, Full Fine-Tuning alcanzó gaps muy bajos (del orden de 0.004–0.01), mientras que LoRA tendió a estabilizarse en valores más altos (en torno a 0.02–0.03). Un gap me-

nor, junto con una pérdida de validación baja, sugiere mejor generalización; un gap mayor puede indicar mayor separación entre ajuste y validación y, por tanto, mayor riesgo de sobreajuste relativo. En este caso, los gaps estabilizados y las pérdidas de validación bajas indican que el modelo está aprendiendo de forma adecuada y no muestran señales de sobreajuste severo. En LoRA, el gap más alto puede apuntar a una generalización menos sólida que Full, pero al mantenerse estable y con val loss baja, no se observa sobreajuste acusado.

3.4.7. Selección final de configuraciones

A partir del análisis conjunto de las métricas obtenidas durante la optimización, se seleccionaron las configuraciones finales para LoRA y Full Fine-Tuning. Estas configuraciones representaban el mejor compromiso entre precisión, estabilidad del entrenamiento y coste computacional dentro de las limitaciones del entorno hardware disponible.

Las configuraciones seleccionadas se utilizaron como base para los experimentos finales descritos en los apartados posteriores, incluyendo la evaluación detallada del rendimiento del modelo y la aplicación de técnicas de cuantización. De este modo, se garantiza que las comparaciones finales se realizan bajo condiciones optimizadas y representativas del mejor rendimiento alcanzable en cada metodología.

3.4.8. Configuraciones óptimas encontradas

Tras la fase de optimización de hiperparámetros, se identificaron configuraciones óptimas para cada metodología de entrenamiento. En la Tabla 3.6 se presentan los valores de hiperparámetros que produjeron el mejor rendimiento según la métrica de evaluación principal (Validation loss), obtenidos mediante búsqueda sistemática con Optuna.

Parámetro	**Full Fine-Tuning**	**LoRA**
Batch_Size	4	16
Gradient_Accumulation_Steps	1	1
Learning_Rate	0.0002	0.00025
Weight_Decay	0.00016	0.000167
Num_Epochs	3	3
Gradient_Checkpointing	True	True
Parámetros específicos de LoRA		
LoRA_R	–	16
LoRA_Alpha	–	1024
LoRA_Dropout	–	0.062

Tabla 3.5: Configuración hiperparamétrica obtenida para Full Fine-Tuning y LoRA tras la optimización con Optuna.

Tras optuna se realizó un pequeño ajuste manual de los hiperparámetros, basado en la experiencia adquirida durante la fase de optimización y en la observación directa del comportamiento del modelo durante el entrenamiento. Este ajuste permitió refinar aún más las configuraciones seleccionadas, buscando mejorar la estabilidad del entrenamiento y la calidad de las respuestas generadas. La configuración final resultante de este proceso se muestra en la Tabla 3.6.

Parámetro	Full Fine-Tuning	LoRA
Batch_Size	32	32
Gradient_Accumulation_Steps	1	1
Learning_Rate	0.001	0.0005
Weight_Decay	0.0002	1e-5
Num_Epochs	3	6
Gradient_Checkpointing	True	True
Parámetros específicos de LoRA		
LoRA_R	–	16
LoRA_Alpha	–	1024
LoRA_Dropout	–	0.062

Tabla 3.6: Configuración final de hiperparámetros seleccionada para el entrenamiento definitivo de Full Fine-Tuning y LoRA tras pequeño ajuste manual.

En el caso de LoRA se aumentaron las epochs para ganar en exact match y se ajustó el learning rate para mejorar la estabilidad del entrenamiento, mientras que en Full Fine-Tuning se incrementó el batch size para aprovechar mejor la capacidad de la GPU y se ajustó el weight decay para reducir el riesgo de sobreajuste.

3.4.9. Selección de la metodología final

El análisis del loss gap durante la fase de optimización permitió extraer conclusiones relevantes sobre el comportamiento de las distintas metodologías de fine-tuning. LoRA muestra una estabilización relativamente rápida del gap tras las primeras iteraciones, con una tendencia suave a la baja a medida que avanzan los ensayos. Full Fine-Tuning, por su parte, presenta más variabilidad inicial, pero converge hacia gaps más bajos en las mejores configuraciones.

Por su parte, el Full Fine-Tuning mostró un comportamiento más variable en las primeras rondas de entrenamiento. No obstante, tras la optimización de los hiper-

parámetros, el modelo convergió hacia valores de loss gap igualmente reducidos, manteniendo una pérdida de validación muy baja y un comportamiento estable en las iteraciones finales. Esta evolución se aprecia en las Figuras 3.16 y 3.17, donde se compara la estabilidad del gap para ambas metodologías.

Desde un punto de vista cuantitativo, ambas metodologías ofrecieron métricas competitivas tras la optimización, aunque Full Fine-Tuning alcanzó menores gaps en las mejores pruebas (Tabla 4.4 y Tabla 4.5). Sin embargo, durante las pruebas prácticas de inferencia y generación de código ensamblador CODE-2 se observaron diferencias cualitativas relevantes.

En particular, aunque LoRA alcanzaba métricas similares en entrenamiento y validación, el código generado presentaba con mayor frecuencia respuestas incompletas, errores en el uso de registros o secuencias menos coherentes en ejemplos no vistos durante el entrenamiento. En contraste, el modelo entrenado mediante Full Fine-Tuning ofrecía respuestas más consistentes, completas y alineadas con la semántica esperada del lenguaje CODE-2, especialmente en ejemplos de mayor complejidad.

Por este motivo, y a pesar de que LoRA demostró ser una técnica eficiente y válida desde el punto de vista computacional, se decidió seleccionar Full Fine-Tuning como metodología final para el modelo definitivo empleado en este trabajo. Esta elección prioriza la calidad y fiabilidad de las respuestas generadas en un contexto educativo, donde la corrección semántica del código resulta fundamental.

3.5. Cuantización y evaluación

Tras la selección de la metodología de entrenamiento final y la optimización de hiperparámetros, se abordó una fase adicional orientada a la mejora de la eficiencia del modelo mediante técnicas de cuantización. Esta etapa resulta especialmente relevante en el contexto de este Trabajo Fin de Grado, dado que el despliegue del modelo se realiza en un entorno con recursos hardware limitados.

La cuantización permite reducir el tamaño del modelo y el consumo de memoria, así como acelerar el proceso de inferencia, a costa de una posible degradación del

rendimiento. Por ello, resulta necesario evaluar cuidadosamente su impacto sobre la calidad del código generado.

3.5.1. Motivación de la cuantización

Los modelos basados en arquitecturas Transformer presentan un elevado número de parámetros y un consumo significativo de memoria, lo que dificulta su uso en entornos locales o educativos sin acceso a hardware especializado. En este trabajo, el entrenamiento y la inferencia se realizaron principalmente en una GPU NVIDIA RTX 2060 con 6 GB de VRAM, lo que impone restricciones claras sobre el tamaño del modelo y la latencia aceptable.

La cuantización se plantea como una solución para:

- Reducir el tamaño del modelo almacenado.
- Disminuir el consumo de memoria durante la inferencia.
- Acelerar el tiempo de generación de código.
- Facilitar un posible despliegue futuro en entornos con recursos limitados.

3.5.2. Técnicas de cuantización aplicadas

En este trabajo se exploraron técnicas de cuantización post-entrenamiento (*Post-Training Quantization, PTQ*), al tratarse de un enfoque sencillo y compatible con modelos ya entrenados. Este método consiste en convertir los pesos del modelo desde precisión en coma flotante (FP32) a representaciones de menor precisión, como INT8, sin necesidad de reentrenar el modelo.

La cuantización se aplicó sobre el modelo entrenado mediante Full Fine-Tuning, utilizando herramientas compatibles con PyTorch y formatos intermedios como ONNX[14] cuando fue necesario, con el objetivo de evaluar el impacto real sobre la inferencia.

3.5.2.1 Justificación de la selección de compresión post-entrenamiento

En las primeras fases del trabajo se consideró la posibilidad de comprimir el modelo durante el entrenamiento (una técnica teóricamente óptima). Sin embargo, al intentar implementarlo con PyTorch en Windows, se descubrió que esta técnica no funciona correctamente con modelos como T5-Small. El sistema automático falla antes de que el entrenamiento pueda comenzar.

Esta incompatibilidad es una limitación conocida que afecta a muchos modelos Transformer complejos. No es un fallo de este trabajo, sino una restricción del estado actual de las herramientas disponibles en PyTorch.

Por este motivo, se decidió utilizar un enfoque alternativo que es igualmente válido y que funciona sin problemas:

- **Compresión después del entrenamiento:** El modelo se entrena normalmente y luego se comprime, reduciendo automáticamente el tamaño de sus pesos según los datos de validación.
- **Conversión a formato ONNX:** El modelo comprimido se convierte a un formato estándar que funciona en múltiples plataformas y permite ejecución más eficiente.

Estas estrategias son utilizadas ampliamente en aplicaciones reales y ofrecen excelentes resultados, siendo perfectamente adecuadas para el contexto educativo de este proyecto.

Figura 3.18: Herramienta ONNX utilizada para la conversión y cuantización del modelo. Se empleó como alternativa viable a QAT-FX para la compresión de T5-Small.

3.5.3. Reducción del tamaño del modelo

Uno de los principales efectos de la cuantización es la reducción del tamaño del modelo en disco y en memoria. Esta reducción permite almacenar modelos más ligeros y cargar el modelo completo en memoria con mayor facilidad.

En esta fase metodológica se analizó:

- El tamaño del modelo antes y después de la cuantización.
- El impacto en el consumo de memoria durante la inferencia.
- La viabilidad de ejecutar el modelo cuantizado en el entorno local sin degradar la estabilidad.

Los valores concretos obtenidos se presentan y analizan en detalle en el Capítulo 4.

3.5.4. Protocolo de evaluación post-cuantización

Para evaluar el impacto de la cuantización sobre el rendimiento del modelo, se definió un protocolo de evaluación específico. Dicho protocolo se aplicó tanto al modelo original como a su versión cuantizada, manteniendo constantes el conjunto de datos de validación y las condiciones de inferencia.

Las métricas empleadas incluyeron:

- Pérdida de validación.
- Métricas de similitud de secuencia (Exact Match, BLEU y ROUGE).
- Tiempo de inferencia por ejemplo.

Este enfoque permitió analizar de forma objetiva el compromiso entre eficiencia y calidad introducido por la cuantización. Los resultados obtenidos se discuten en profundidad en el Capítulo 4, donde se comparan ambas versiones del modelo.

4. RESULTADOS Y DISCUSIÓN

En este capítulo se presentan y analizan los resultados obtenidos tras la aplicación de las distintas metodologías descritas en el Capítulo 3. Se evalúa el rendimiento del modelo entrenado mediante Full Fine-Tuning y LoRA, así como el impacto de la optimización de hiperparámetros y de la cuantización sobre la calidad del código generado en lenguaje ensamblador CODE-2.

El análisis se apoya tanto en métricas cuantitativas como en observaciones cualitativas, con el objetivo de ofrecer una visión completa del comportamiento del modelo en escenarios reales de generación de código. Asimismo, se discuten las ventajas, limitaciones y compromisos asociados a cada enfoque, permitiendo justificar la elección de la metodología final adoptada en este trabajo.

4.1. Métricas obtenidas

En esta sección se presentan las métricas obtenidas a partir de los distintos experimentos realizados sobre los modelos entrenados. El objetivo principal es ofrecer una visión cuantitativa del rendimiento alcanzado por cada metodología, sirviendo como base para el análisis comparativo y la discusión posterior.

Todas las métricas se calcularon utilizando el conjunto de validación descrito en el Capítulo 3, garantizando que no existiera solapamiento con los datos empleados durante el entrenamiento. De este modo, los resultados reflejan la capacidad de generalización real de los modelos evaluados.

4.1.1. Métricas de entrenamiento y validación

En primer lugar, se analizan las métricas relacionadas con el proceso de entrenamiento, concretamente la pérdida de entrenamiento (*Training Loss*) y la pérdida de validación (*Validation Loss*). Estas métricas permiten evaluar la convergencia del modelo y detectar posibles fenómenos de sobreajuste.

Metodología	Dataset	Training Loss	Validation Loss
Full Fine-Tuning	10k	0.0896	0.0176
LoRA	10k	0.4905	0.0558
Prompt tuning	10k	5.4074	0.8597
Prueba Inicial	1k	1.5078	0.8401

Tabla 4.1: Ejemplo de métricas de entrenamiento y validación obtenidas en la Fase 1 para cada metodología utilizando el mismo subconjunto de datos.

4.1.2. Métricas de calidad de generación

Además de las métricas de pérdida, se evaluó la calidad del código generado mediante métricas específicas de generación de secuencias. Estas métricas permiten medir el grado de coincidencia entre el código generado por el modelo y las soluciones de referencia en CODE-2.

Metodología	Dataset	Exact Match	BLEU	ROUGE-L
Full Fine-Tuning (Default)	18k	0.3633	0.8164	0.9087
LoRA (Default)	18k	0.0006	0.1946	0.4031

Tabla 4.2: Ejemplo de métricas de calidad de generación obtenidas en la fase 2 sobre las 2 metodologías principales utilizando configuraciones por defecto.

4.1.3. Métricas de rendimiento computacional

Finalmente, se analizaron métricas relacionadas con el rendimiento computacional del modelo, con el objetivo de evaluar su viabilidad práctica en entornos con recursos limitados. Estas métricas incluyen el tiempo de entrenamiento y el tiempo medio de inferencia por ejemplo.

Metodología	Dataset	Tiempo (min)	Samples/seg	VRAM (MB)
Full Fine-Tuning (Default)	18k	31.39	33.45	2619
LoRA (Default)	18k	33.49	31.35	2054

Tabla 4.3: Comparativa de tiempos de entrenamiento, rendimiento (muestras por segundo) y uso de memoria VRAM.

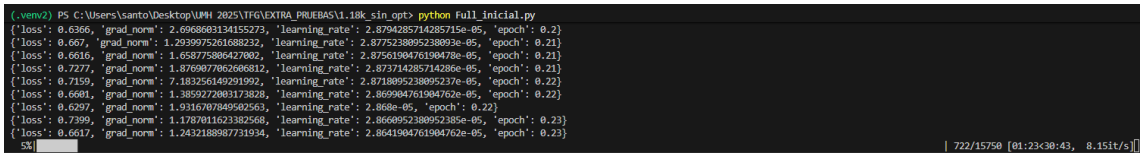

Figura 4.1: Ejemplo de uso de recursos durante el entrenamiento del modelo (GPU y memoria).

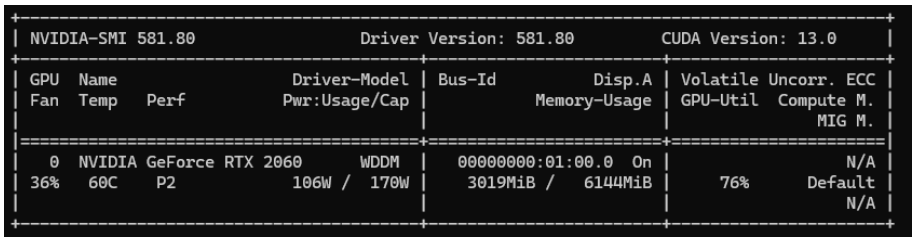

Figura 4.2: Captura de nvidia-smi mostrando el uso de GPU durante el entrenamiento.

4.1.4. Resultados tras la optimización de hiperparámetros

En este apartado se presentan las métricas obtenidas tras aplicar el proceso de optimización de hiperparámetros descrito en el Apartado 3.4. El objetivo es cuantificar la mejora alcanzada respecto a las configuraciones iniciales y verificar si la optimización produce un impacto consistente tanto en calidad de generación como en estabilidad.

Metodología	**Dataset**	**Train Loss**	**Val Loss**	**Loss Gap**	**Exact Match**
Full Fine-Tuning	18k	0.0105	0.00574	0.00475	0.73
LoRA	18k	0.1138	0.0850	0.02878	0.55

Tabla 4.4: Resultados de pérdida de entrenamiento, validación, gap y Exact Match tras la optimización de hiperparámetros para cada metodología con optuna.

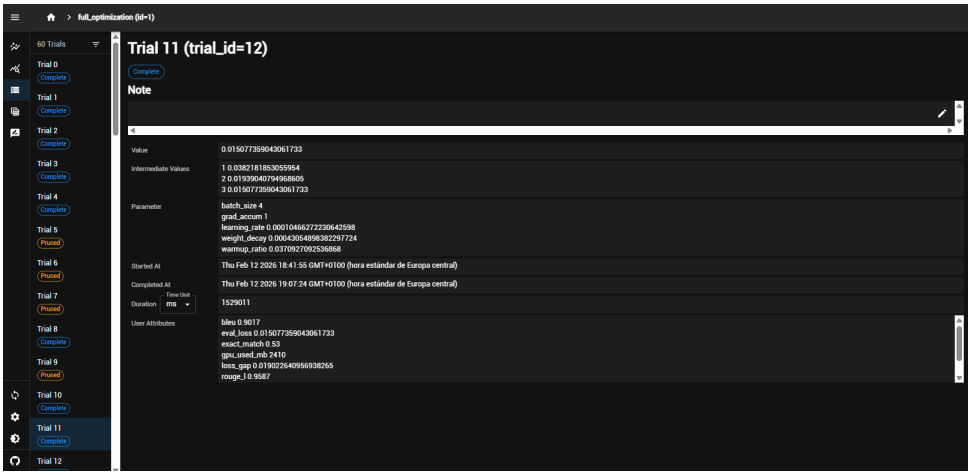

Figura 4.3: Resumen del proceso de optimización (Optuna): evolución de métricas y rondas de pruebas.

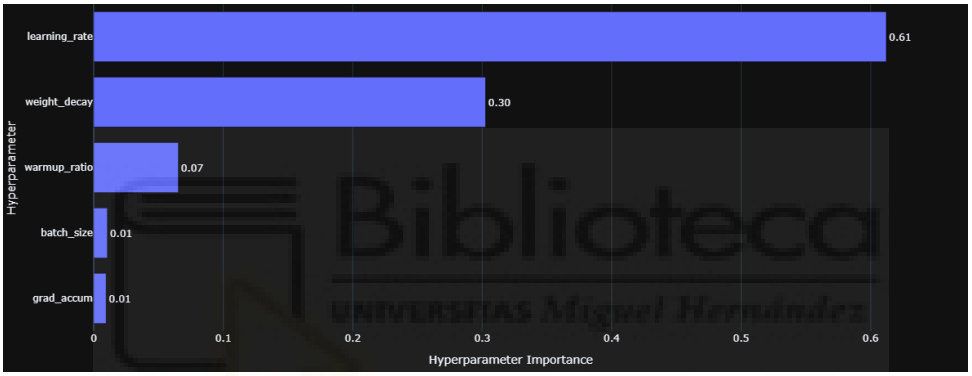

Figura 4.4: Importancia de hiperparámetros según Optuna para full-finetuning.

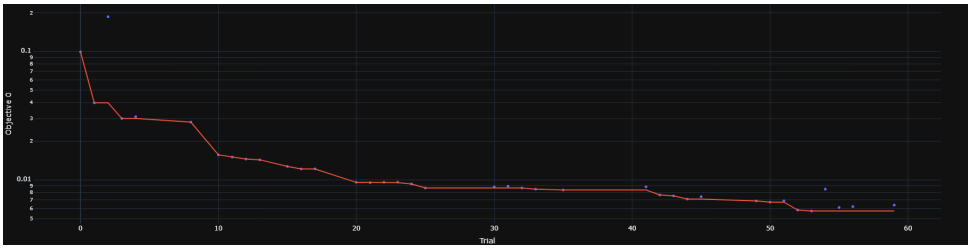

Figura 4.5: Historia de optimización: evolución del valor objetivo a lo largo de las pruebas para full-finetuning.

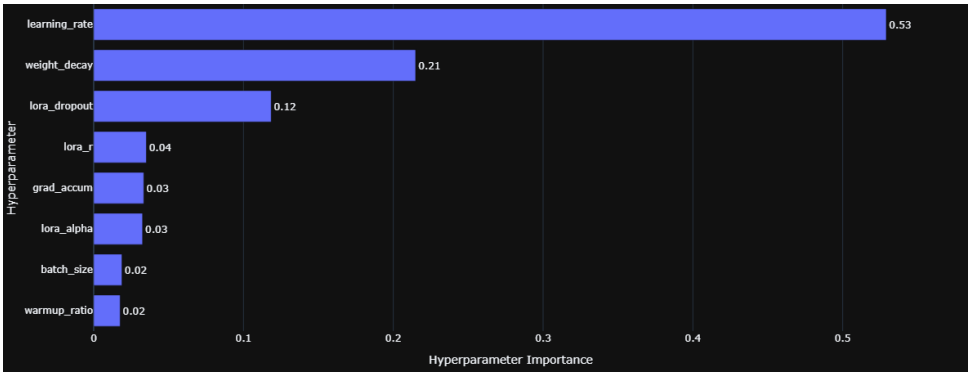

Figura 4.6: Importancia de hiperparámetros según Optuna para LoRA.

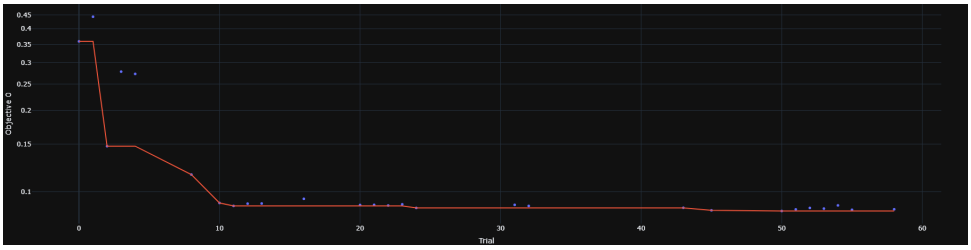

Figura 4.7: Historia de optimización: evolución del valor objetivo a lo largo de las pruebas para LoRA.

Tras el ajuste y afinamiento manual ahorramos tiempo de entrenamiento sin perder los buenos resultados en las métricas.

Metodología	**Dataset**	**Train Loss**	**Val Loss**	**Loss Gap**	**Exact Match**
Full Fine-Tuning (optimo)	18k	0.0219	0.0124	0.0090	0.75
LoRA (optimo)	18k	0.1708	0.1448	0.0259	0.64

Tabla 4.5: Resultados de pérdida de entrenamiento, validación, gap y Exact Match tras la optimización de hiperparámetros aprovechando el máximo de VRAM.

4.1.5. Resumen de métricas

Las métricas presentadas en los apartados anteriores ofrecen una visión global del rendimiento de los modelos evaluados en las distintas fases experimentales de este trabajo. A través de las pérdidas de entrenamiento y validación, las métricas de calidad de generación y las métricas de rendimiento computacional, se ha podido caracterizar de forma objetiva el comportamiento de cada metodología bajo condiciones controladas.

Los resultados iniciales muestran diferencias claras entre las estrategias de fine-tuning en términos de convergencia, estabilidad y coste computacional. Asimismo, la fase de optimización de hiperparámetros permitió mejorar de forma consistente el rendimiento de los modelos, evidenciando la importancia de un ajuste adecuado para maximizar la capacidad de generalización.

Método	Dataset	Training Loss	Validation Loss	EM (%)	BLEU	ROUGE-L	Tiempo (min)	VRAM (MB)
Prueba Inicial	1k	1.5078	0.8401	-	-	-	-	-
Full	10k	0.0896	0.0176	-	-	-	-	-
LoRA	10k	0.4905	0.0558	-	-	-	-	-
Prompt Tuning	10k	5.4074	0.8597	-	-	-	-	-
Full (default)	18k	0.080	0.039	0.36	0.8164	0.90	33.45	2619
LoRA (default)	18k	0.628	0.480	0.0006	0.1946	0.4031	33.49	2054
Full (optuna)	18k	0.0105	0.0057	0.73	0.9423	0.9789	25.90	2633
LoRA (optuna)	18k	0.1138	0.085	0.55	0.9113	0.9643	28.94	1875
Full (optimo)	18k	0.0219	0.012	0.75	0.9452	0.9815	11.17	5534
LoRA (optimo)	18k	0.1708	0.1448	0.64	0.9302	0.9759	21.37	4285

Tabla 4.6: Comparativa de todas las pruebas realizadas.

4.2. Comparación entre métodos

En este apartado se realiza un análisis comparativo de las distintas metodologías evaluadas a lo largo del trabajo, integrando los resultados cuantitativos presentados en la Sección 4.1 con observaciones cualitativas derivadas de las pruebas prácticas de inferencia. El objetivo es identificar las fortalezas y limitaciones de cada enfoque y justificar la selección de la metodología final empleada.

4.2.1. Comparación entre estrategias de fine-tuning

Los resultados obtenidos muestran que las distintas estrategias de fine-tuning presentan comportamientos claramente diferenciados en términos de calidad de generación, estabilidad del entrenamiento y coste computacional.

El Prompt Tuning, aunque destaca por su eficiencia y bajo consumo de recursos, ofrece un rendimiento limitado en la tarea de generación de código ensamblador CODE-2. Las métricas obtenidas reflejan una menor capacidad de adaptación al dominio específico, lo que se traduce en pérdidas de validación elevadas y una menor

calidad de las secuencias generadas.

Por su parte, tanto LoRA como Full Fine-Tuning alcanzan valores significativamente mejores en las métricas de calidad, evidenciando una adaptación efectiva del modelo al lenguaje ensamblador. Sin embargo, estas similitudes cuantitativas esconden diferencias relevantes que se hacen visibles al analizar el comportamiento del modelo en escenarios de inferencia más exigentes.

4.2.2. LoRA frente a Full Fine-Tuning

La comparación directa entre LoRA y Full Fine-Tuning revela un compromiso claro entre eficiencia y calidad. Tras la optimización manual, Full Fine-Tuning alcanza mejores métricas de calidad (EM 0.75, BLEU 0.9452, ROUGE-L 0.9815) frente a LoRA (EM 0.64, BLEU 0.9302, ROUGE-L 0.9759), y mantiene un loss gap más bajo (0.0090 frente a 0.0259), lo que sugiere una generalización más consistente en el mejor ensayo. En rendimiento, Full es más rápido en el ajuste final (11.17 min frente a 21.37 min, debido también al aumento de epochs) pero a costa de un uso de VRAM superior (5534 MB frente a 4285 MB), en línea con el mayor coste de ajustar todos los pesos.

No obstante, durante las pruebas prácticas de inferencia se observaron diferencias cualitativas significativas. El modelo entrenado mediante LoRA tendía a generar respuestas incompletas o menos coherentes en ejemplos no vistos durante el entrenamiento, especialmente en secuencias largas o con mayor complejidad estructural. Estos errores incluían omisiones de instrucciones, uso inconsistente de registros o pérdida de coherencia en el orden de las operaciones.

En contraste, el modelo entrenado mediante Full Fine-Tuning mostró una mayor robustez en estos escenarios, generando código más completo, consistente y alineado con la semántica esperada del lenguaje CODE-2. Esta diferencia sugiere que, aunque LoRA es capaz de adaptar eficazmente el modelo en términos de métricas agregadas, el ajuste completo de los pesos permite capturar dependencias más profundas y patrones estructurales complejos propios del dominio.

4.2.3. Impacto de la optimización de hiperparámetros

La fase de optimización de hiperparámetros tuvo un impacto determinante en el rendimiento final de los modelos evaluados. En Full Fine-Tuning, la pérdida de validación bajó de 0.039 (default) a 0.012 (optimo) y el EM subió de 0.36 a 0.75. En LoRA, la pérdida de validación se redujo de 0.480 a 0.1448 y el EM pasó de 0.0006 a 0.64. Estas mejoras se acompañaron de gaps más controlados en los mejores ensayos, reflejando una mayor estabilidad del entrenamiento.

Los resultados muestran que configuraciones aparentemente similares pueden producir comportamientos muy distintos en términos de generalización, lo que pone de manifiesto la importancia de un ajuste fino adaptado al conjunto de datos y al entorno hardware. Este efecto fue especialmente notable en el caso del Full Fine-Tuning, donde la optimización permitió minimizar el gap sin sacrificar la pérdida de validación.

En el caso de LoRA, la optimización permitió identificar configuraciones más estables y mejorar de forma notable la calidad de generación, aunque sin eliminar completamente las limitaciones observadas en escenarios complejos.

4.2.4. Selección de la metodología final

Considerando de forma conjunta los resultados cuantitativos y cualitativos, se decidió seleccionar el Full Fine-Tuning como metodología final para el modelo empleado en este Trabajo Fin de Grado. La configuración óptima ofrece el mejor equilibrio entre calidad y generalización (EM 0.75, BLEU 0.9452, ROUGE-L 0.9815 y loss gap 0.0090), mientras que LoRA, aun siendo competitiva, mantiene un gap mayor y una calidad ligeramente inferior en las métricas de generación.

Dado el carácter educativo del proyecto y la importancia de la corrección semántica y estructural del código generado, se priorizó la fiabilidad de las respuestas frente a la eficiencia computacional. No obstante, los resultados obtenidos con LoRA demuestran que esta técnica constituye una alternativa válida en contextos donde los

recursos hardware son más limitados.

4.2.5. Modelo final y ajuste de predicciones

El modelo final seleccionado para el despliegue y uso en este trabajo es el entrenado mediante Full Fine-Tuning con la configuración óptima de hiperparámetros. Este modelo ofrece un rendimiento sólido en términos de calidad de generación de código ensamblador CODE-2, con métricas consistentes con una adaptación efectiva al lenguaje objetivo y una capacidad de generalización robusta a casos no vistos durante el entrenamiento.

4.2.6. Escalado del dataset y enriquecimiento de datos

Con el objetivo de maximizar la capacidad de generalización del modelo final, se decidió incrementar el conjunto de datos de entrenamiento de 18k a 36k ejemplos. Este aumento se realizó manteniendo la misma proporción de datos de entrenamiento, validación y testeo (70 %-20 %-10 %), pero modificando de manera selectiva la composición del dataset.

La principal modificación consistió en aumentar la proporción de ejemplos con errores (instancias destinadas a mejorar la robustez del modelo ante entradas defectuosas) del 10 % al 20 % del total. Esta estrategia se fundamenta en los siguientes principios:

- **Cobertura de combinaciones raras:** Ampliar la densidad de ejemplos con registros menos frecuentes, longitudes variadas y patrones de error diversos.
- **Robustez frente a entradas malformadas:** Mejorar la capacidad del modelo para manejar de forma coherente especificaciones con errores sintácticos o semánticos.
- **Equilibrio entre aprendizaje y generalización:** Aprovechar el tamaño de T5-Small (aprox. 60M parámetros) para aprender patrones diversos sin depender de redundancia excesiva.

La ausencia de un incremento equivalente en el tamaño de ejemplos correctos refleja una priorización consciente de la diversidad semántica y sintáctica sobre la redundancia de patrones bien formados.

Metodología	Dataset	Train Loss	Val Loss	EM (%)	BLEU	ROUGE-L	Tiempo (min)	VRAM (MB)
Full (final)	36k	0.0088	0.0034	0.80	0.966	0.995	25.26	5607
Full (72k)	72k	0.0033	0.0017	0.81	0.9658	0.9936	61.52	5664

Tabla 4.7: Comparativa de rendimiento: modelo con 36k ejemplos (óptimo) versus escalado a 72k ejemplos. Muestra el compromiso marginal entre mejora de precisión y coste computacional.

4.2.6.1 Análisis del escalado del dataset: 36k vs 72k

Con el objetivo de evaluar si un mayor escalado del dataset podría mejorar significativamente el rendimiento, se realizó un experimento adicional entrenando el modelo con 72 000 ejemplos (manteniendo las proporciones 70 %-20 %-10 %). Los resultados muestran que, aunque se obtiene una ligera mejora en métricas de calidad (EM: 0.80 $\rightarrow$ 0.81, +1.25 %), esta ganancia no justifica el incremento en tiempo de entrenamiento (**2.43 veces más**, de 25.26 a 61.52 minutos).

En términos de pérdida, el modelo con 72k logra valores ligeramente inferiores (Train Loss: 0.0033, Val Loss: 0.0017 frente a 0.0088 y 0.0034 respectivamente), reflejando un ajuste más fino. Sin embargo, las diferencias son marginales considerando que el dataset de 36k ya alcanza un nivel de saturación en términos de ganancia de generalización.

Conclusión: Se determinó que **36k ejemplos constituye el punto óptimo** para este modelo y tarea, balanceando adecuadamente precisión, velocidad de entrenamiento y eficiencia computacional. El escalado a 72k produce retornos decrecientes inaceptables desde la perspectiva de coste-beneficio en un contexto educativo donde los recursos son limitados.

Los resultados obtenidos con el dataset de 36k muestran un comportamiento equilibrado entre ajuste y generalización. El Exact Match (EM) alcanzó 80 %, representando una mejora respecto al modelo entrenado con 18k ejemplos (EM = 0.75).

Esta mejora se debe a que el dataset ampliado incorpora una mayor proporción de ejemplos complejos y con errores (20 % frente a 10 %), lo que aumenta la dificultad de la tarea de validación y prioriza la robustez frente a la memorización de patrones simples. Por otro lado, las métricas BLEU (0.966) y ROUGE-L (0.995) confirman una alta similitud de secuencias, indicando que las salidas generadas mantienen una estructura coherente incluso ante mayor diversidad de entrada.

La pérdida de validación de 0.0034, junto con una pérdida de entrenamiento de 0.0088, evidencia un comportamiento estable sin indicios de sobreajuste severo. El loss gap de 0.0054 se mantuvo bajo y consistente, reforzando la conclusión de que Full Fine-Tuning ofrece un equilibrio favorable entre ajuste y generalización para la tarea evaluada, incluso con un dataset más exigente y heterogéneo.

4.2.7. Ajuste de predicciones y análisis de resultados

Para el ajuste de predicciones se empleó una decodificación conservadora, buscando salidas deterministas y consistentes. Los parámetros se seleccionaron estratégicamente para reducir la variabilidad y favorecer secuencias sintácticamente válidas en CODE-2. En particular, se utilizó una temperatura baja (0.2) para aumentar el determinismo de las salidas, un valor moderado de top_p (0.6) para restringir el muestreo a tokens de alta probabilidad, y decodificación greedy (num_beams = 1) para maximizar la coherencia.

Este ajuste favoreció significativamente la mejora en la métrica de Exact Match (EM), que pasó de 80 % a 83 %. Esta mejora resulta especialmente relevante en el contexto de generación de código ensamblador CODE-2, donde **el Exact Match es la métrica más crítica**, ya que código parcialmente correcto no es ejecutable ni útil en lenguaje ensamblador. La Tabla 4.8 resume la configuración empleada y las métricas finales obtenidas usando el dataset de testeo.

Parámetro	Valor
num_beams	1
temperature	0.2
top_p	0.6
exact_match	0.83
bleu	0.97
rouge_l	0.9922

Tabla 4.8: Configuración óptima de parámetros de generación y métricas de evaluación del modelo final tras ajuste de decodificación.

La combinación de un dataset enriquecido (36k ejemplos con mayor diversidad) y una estrategia de decodificación optimizada permitió alcanzar un Exact Match de 83 %, con valores de BLEU (0.97) y ROUGE-L (0.9922) que confirman la alta calidad estructural y semántica del código generado. Estos resultados validan la eficacia del enfoque metodológico adoptado para la generación de código ensamblador en entornos educativos.

4.2.8. Evaluación cualitativa del modelo final

Más allá de las métricas cuantitativas agregadas, una evaluación cualitativa del código generado por el modelo final reveló mejoras significativas en varios aspectos clave del comportamiento del modelo:

- **Coherencia semántica:** El modelo demuestra una comprensión profunda de las relaciones entre instrucciones, evitando con mayor frecuencia la generación de dependencias lógicas inconsistentes.
- **Manejo de errores en la entrada:** La inclusión de un porcentaje mayor de ejemplos con errores permitió que el modelo desarrollase una capacidad mejorada para detectar y reportar especificaciones malformadas en el lenguaje natural.
- **Variabilidad de patrones:** En ejemplos que requieren combinaciones menos frecuentes de instrucciones o registros, el modelo muestra una mayor tendencia

a generar salidas válidas y funcionales.

Un aspecto importante a destacar es la diferencia entre la métrica de Exact Match (EM) y la calidad real de las predicciones. Aunque el EM es una métrica binaria estricta, el análisis cualitativo revela casos donde el modelo genera salidas muy cercanas a la referencia, semánticamente válidas y funcionalmente correctas, pero que no se cuentan como exactamente coincidentes debido a pequeñas discrepancias.

A modo de ejemplo, considérese el siguiente caso de entrada con múltiples instrucciones complejas:

Categoría	**Contenido**
Input (Descripción)	
	ESCRIBE EL CONTENIDO DEL REGISTRO 0 EN MEMORIA Sustrae r2 de r2 y pon el resultado en rD Calcula r7 menos r9 y guarda en el registro D OBTÉN EL DATO DE 8 Y ALMACÉNALO EN rF LEE LA DIRECCIÓN 1 Y GUÁRDALA EN rA
Output Esperado	

	Error: dirección vacía (ambigua)
	SUBS rD,r2,r2
	SUBS rD,r7,r9
	LD rF,[rD+H'08'] ; rD = 0000
	LD rA,[rD+H'01'] ; rD = 0000

Predicción del Modelo	

	Error: dirección vaca (ambigua)
	SUBS rD,r2,r2
	SUBS rD,r7,r9
	LD rF,[rD+H'08'] ; rD = 0000
	LD rA,[rD+H'01'] ; rD = 0000

Resultado: NO es Exact Match	

Tabla 4.9: Ejemplo representativo de predicción cualitativamente correcta pero cuantitativamente contada como no Exact Match debido a diferencia de acentuación ("vacía" vs "vaca").

En este ejemplo, la predicción del modelo es semanticamente válida y funcional. El

error detectado es correcto, la lógica de las instrucciones es coherente, y los registros se manipulan de forma apropiada. Sin embargo, la métrica de Exact Match no cuenta esta salida como correcta debido a la omisión de tilde en la palabra "vacía". Este tipo de discrepancias, aunque son capturadas por métricas como BLEU (0.9623) y ROUGE-L (0.9922), subraya la importancia de evaluar el modelo desde múltiples perspectivas.

Este caso ilustra una limitación de las métricas estrictamente binarias en el contexto de generación de código: aunque el Exact Match es crítico para el despliegue automatizado (ya que un código parcialmente incorrecto no es ejecutable), también es importante reconocer que el modelo es capaz de generar salidas altamente coherentes que un usuario humano consideraría correctas. Un análisis más detallado de este tipo de errores y sus patrones de distribución se proporciona en el Apartado 4.4 del presente documento.

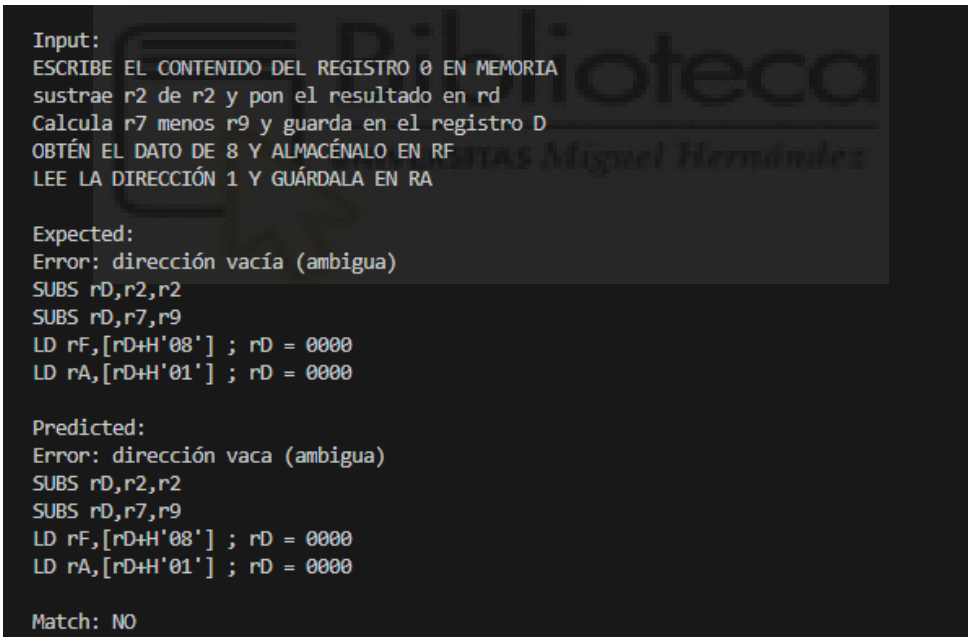

Figura 4.8: Interfaz interactiva del modelo final mostrando ejemplo de predicción con discrepancia menor entre esperado y predicción.

En conclusión, el modelo final obtenido representa la mejor versión desarrollada en este Trabajo Fin de Grado, siendo el resultado de un proceso iterativo de optimización que priorizó tanto la calidad de generación como la robustez ante casos límite. Este modelo constituye el punto de partida recomendado para futuras evaluaciones

en un entorno educativo real.

4.3. Impacto de la cuantización

Una vez seleccionada la metodología de entrenamiento y optimizados los hiperparámetros del modelo, se abordó el estudio del impacto de la cuantización como etapa final del proceso experimental. La cuantización[8] constituye una técnica clave para facilitar el despliegue de modelos de aprendizaje profundo en entornos con recursos limitados, al reducir el tamaño del modelo y mejorar la eficiencia durante la inferencia.

En el contexto de este Trabajo Fin de Grado, el análisis de la cuantización resulta especialmente relevante, ya que permite evaluar hasta qué punto es posible mantener una calidad aceptable en la generación de código ensamblador CODE-2 al mismo tiempo que se optimizan aspectos prácticos como el consumo de memoria y la latencia. Este apartado presenta una evaluación comparativa entre el modelo original y su versión cuantizada, analizando tanto las métricas de calidad como los indicadores de rendimiento computacional.

El objetivo de este análisis no es únicamente cuantificar las mejoras obtenidas en términos de eficiencia, sino también identificar las posibles limitaciones introducidas por la reducción de precisión y su impacto en la corrección y coherencia del código generado.

4.3.1. Resultados del modelo cuantizado

Una vez seleccionado el modelo final entrenado, se aplicaron técnicas de cuantización con el objetivo de reducir el tamaño del modelo y mejorar el rendimiento en inferencia (latencia y consumo de memoria). En este apartado se presentan las métricas obtenidas tras cuantizar, comparándolas con la versión base (sin cuantización) bajo el mismo conjunto de validación y protocolo de inferencia.

Nota metodológica: Para garantizar una comparación justa y consistente entre

todas las variantes de optimización, las evaluaciones presentadas en las siguientes tablas se realizaron con parámetros de decodificación por defecto (temperature=1.0, num_beams=1, sin restricciones de top_p) y bajo todo el dataset completo de test. Esto explica que el modelo "Original FP32" muestre EM=0.82 en lugar de EM=0.83, valor este último alcanzado únicamente tras aplicar el ajuste de decodificación optimizado descrito en la Subsección 4.2.7.

Versión del modelo	**Device/Provider**	**EM**	**BLEU**	**ROUGE-L**	**Tiempo medio (s)**
Original FP32 (CUDA)	cuda	0.82	0.9647	0.9957	11.38
Original FP32 (CPU)	cpu	0.82	0.9647	0.9957	34.22
Destilado T5-small	cuda	0.79	0.9660	0.9963	11.56
Podado 10 %	cuda	0.83	0.9673	0.9967	11.41
Podado 30 %	cuda	0.00	0.0000	0.0000	13.59
Podado 50 %	cuda	0.00	0.0000	0.0005	11.95
Cuantizado INT8 (PQT)	cpu	0.28	0.8131	0.9269	29.59
Cuantizado INT8 (ONNX)	cpu	0.79	0.9648	0.9953	22.89
ONNX FP32	cpu	0.82	0.9647	0.9957	26.60

Tabla 4.10: Métricas de calidad y latencia media por versión del modelo.

Versión del modelo	**Tamaño (MB)**	**CPU RSS pico (MB)**	**GPU pico (MB)**	**Speedup vs FP32 GPU**	**Speedup vs FP32 CPU**
Original FP32 (CUDA)	230.78	1235.0	638.6	1.0x	–
Original FP32 (CPU)	230.78	2026.2	0.0	0.33x	–
Destilado T5-small	230.78	1797.6	639.3	0.98x	–
Podado 10 %	230.78	1796.9	638.2	1.0x	–
Podado 30 %	230.78	1796.9	643.2	0.84x	–
Podado 50 %	230.78	1796.9	643.2	0.95x	–
Cuantizado INT8 (PQT)	120.57	1915.7	0.0	0.38x	1.16x
Cuantizado INT8 (ONNX)	142.26	3475.0	0.0	0.5x	1.5x
ONNX FP32	566.02	3854.4	0.0	0.43x	1.29x

Tabla 4.11: Recursos y tamaño del modelo, junto con los speedups relativos.

Figura 4.9: Captura/figura del proceso de cuantización (pipeline, exportación a ONNX[14] si aplica, logs o configuración).

Figura 4.10: Captura/figura del benchmark de inferencia comparando el modelo original y el cuantizado.

4.3.2. Análisis detallado de cada variante

Los resultados presentados en las Tablas 4.10 y 4.11 permiten realizar un análisis diferenciado del comportamiento de cada técnica de optimización aplicada sobre el modelo base entrenado mediante Full Fine-Tuning.

4.3.2.1 Cuantización INT8 con PyTorch (PTQ)

La cuantización dinámica aplicada directamente con PyTorch[8] resultó en una reducción significativa del tamaño del modelo (de 230.78 MB a 120.57 MB, aproximadamente un 48 % de compresión). Sin embargo, esta técnica mostró una degradación severa en las métricas de calidad: el Exact Match descendió de 0.82 a 0.28, BLEU de 0.9647 a 0.8131, y ROUGE-L de 0.9957 a 0.9269.

El análisis cualitativo reveló que esta caída se debe principalmente a la pérdida de precisión en los pesos de las capas de atención, lo que afecta significativamente a la coherencia semántica del código generado. Además, el tiempo de inferencia en CPU (29.59 s) no mejoró sustancialmente respecto al modelo FP32 en CPU (34.22 s), ofreciendo un speedup de apenas 1.16x.

Estos resultados indican que la cuantización PTQ dinámica de PyTorch, si bien es simple de implementar, no resulta adecuada para la tarea de generación de código ensamblador CODE-2 cuando se requiere mantener un nivel aceptable de calidad.

4.3.2.2 Cuantización INT8 con ONNX Runtime

Por el contrario, la cuantización realizada mediante exportación a formato ONNX[14] y optimización posterior con ONNX Runtime mostró resultados significativamente superiores. El modelo cuantizado INT8 ONNX mantuvo un Exact Match de 0.79 (prácticamente idéntico al modelo base en CPU), con métricas BLEU (0.9648) y ROUGE-L (0.9953) casi equivalentes al modelo original.

Esta técnica logró reducir el tiempo de inferencia a 22.89 s, ofreciendo un speedup

de 1.5x respecto a FP32 en CPU. El tamaño del modelo resultante (142.26 MB) es mayor que el obtenido con PTQ de PyTorch, pero el compromiso calidad-eficiencia es notablemente superior.

El éxito de esta técnica se atribuye a la calibración más precisa del rango de cuantización y a las optimizaciones específicas del runtime de ONNX para operaciones cuantizadas. Esta variante representa la mejor opción cuando se busca reducir latencia y memoria manteniendo la calidad del modelo.

4.3.2.3 Exportación ONNX FP32

La exportación del modelo a formato ONNX sin cuantización (FP32) permitió evaluar el impacto del cambio de framework independientemente de la reducción de precisión. Los resultados muestran que el modelo ONNX FP32 mantiene exactamente las mismas métricas de calidad que el modelo PyTorch original (EM=0.82, BLEU=0.9647, ROUGE-L=0.9957), confirmando que la conversión no introduce pérdida de información.

Sin embargo, el tamaño del modelo exportado aumentó considerablemente (566.02 MB frente a 230.78 MB del modelo PyTorch), debido a la inclusión de metadatos adicionales y al formato de almacenamiento de ONNX. El tiempo de inferencia en CPU (26.60 s) mejoró respecto a PyTorch FP32 CPU (34.22 s), ofreciendo un speedup de 1.29x gracias a las optimizaciones del runtime.

4.3.2.4 Técnica de destilación (Knowledge Distillation)

La destilación de conocimiento se aplicó entrenando un modelo estudiante (T5-Small) utilizando las salidas del modelo profesor (el Full Fine-Tuning final). Los resultados muestran un Exact Match de 0.79, ligeramente inferior al modelo original (0.82), pero con métricas BLEU (0.9660) y ROUGE-L (0.9963) comparables.

El modelo destilado mantiene el mismo tamaño que el original (230.78 MB), ya que no se modificó la arquitectura, y el tiempo de inferencia es prácticamente idéntico

(11.56 s vs 11.38 s). Esta técnica resulta útil cuando se busca acelerar el entrenamiento de modelos adicionales reutilizando el conocimiento del modelo base, pero no ofrece mejoras directas en eficiencia de inferencia.

4.3.2.5 Poda de pesos (Pruning)

Se evaluaron tres niveles de poda no estructurada: 10 %, 30 % y 50 % de eliminación de pesos con menor magnitud.

La poda al 10 % mostró resultados sorprendentemente positivos: el Exact Match aumentó ligeramente a 0.83 (superior al modelo base), con BLEU de 0.9673 y ROUGE-L de 0.9967. Este comportamiento puede atribuirse a un efecto de regularización implícita, donde la eliminación de pesos pequeños reduce el sobreajuste y mejora la generalización. El tiempo de inferencia (11.41 s) y el consumo de memoria son similares al modelo base.

Sin embargo, niveles de poda más agresivos (30 % y 50 %) resultaron en un colapso completo del modelo, con Exact Match de 0.00 y métricas BLEU prácticamente nulas. Esto indica que, aunque la poda moderada puede ser beneficiosa, existe un umbral crítico más allá del cual el modelo pierde completamente su capacidad de generar código coherente.

Una posible explicación es que, a diferencia de tareas de clasificación donde la poda agresiva suele ser viable, la generación secuencial de código requiere mayor precisión en todos los parámetros de las capas de atención y feed-forward. Este resultado sugiere que la poda debe aplicarse con extrema cautela en modelos generativos.

4.3.3. Comparación antes y después de la cuantización

La comparación global entre el modelo base Original FP32 y sus variantes optimizadas permite extraer conclusiones prácticas sobre el compromiso entre calidad y eficiencia.

En términos de **calidad de generación**, el modelo ONNX INT8 emerge como la

mejor alternativa optimizada, manteniendo un Exact Match de 0.79 (solo un 3.7 % inferior al modelo base CUDA) con una reducción significativa en latencia. Por el contrario, la cuantización PTQ de PyTorch no resulta adecuada debido a la severa degradación observada.

Desde el punto de vista de **eficiencia computacional**, el modelo ONNX INT8 ofrece el mejor balance: reduce el tiempo de inferencia en CPU de 34.22 s a 22.89 s (speedup de 1.5x) y disminuye el tamaño del modelo a 142.26 MB, lo que facilita su despliegue en entornos con recursos limitados. El consumo de memoria RAM (CPU RSS pico) aumentó a 3475 MB, lo que debe considerarse en sistemas con restricciones severas de memoria.

El modelo **podado al 10 %** presenta un resultado interesante: mejora ligeramente la calidad (EM=0.83) sin sacrificar rendimiento, lo que sugiere que podría combinarse con cuantización para obtener mejoras adicionales en eficiencia.

En cuanto al modelo **destilado**, aunque mantiene una calidad aceptable (EM=0.79), no ofrece ventajas claras en términos de eficiencia de inferencia respecto al modelo base. Su utilidad radica principalmente en escenarios donde se desea transferir conocimiento para entrenar múltiples modelos especializados de forma más rápida.

4.3.3.1 Trade-off calidad vs. eficiencia

La Figura 4.11 ilustra conceptualmente el compromiso entre calidad (medida por Exact Match) y eficiencia (medida por speedup e inverso del tamaño del modelo).

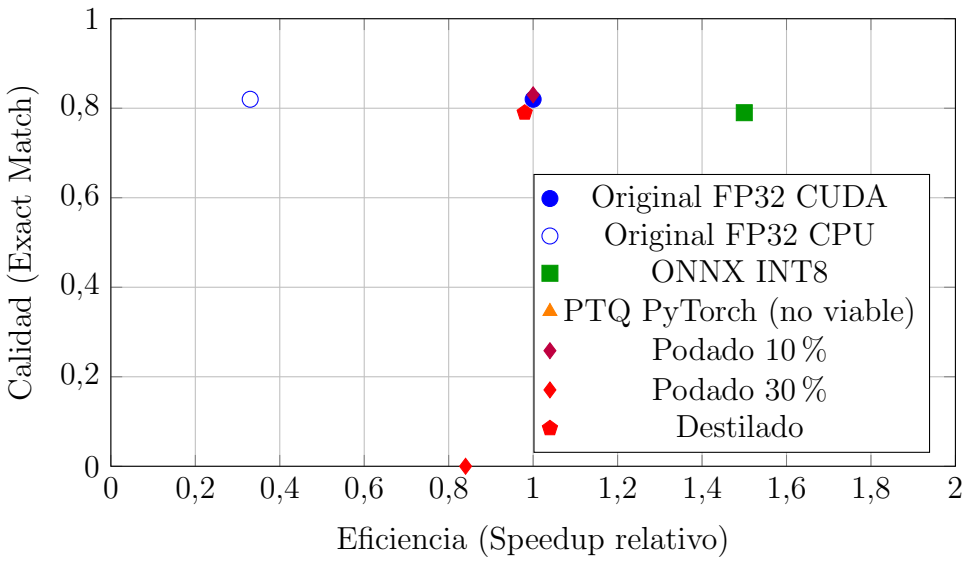

Figura 4.11: Compromiso entre calidad (Exact Match) y eficiencia (speedup) para las diferentes variantes del modelo evaluadas. La región superior derecha representa configuraciones ideales (alta calidad y alta eficiencia).

Este análisis confirma que **ONNX INT8 representa la mejor opción para despliegue en CPU** cuando se requiere un balance óptimo entre calidad y rendimiento. Para aplicaciones donde la máxima calidad es crítica, el modelo Original FP32 en CUDA sigue siendo la mejor opción, mientras que el modelo Podado 10 % ofrece una mejora marginal en calidad manteniendo la misma eficiencia del modelo base.

4.3.4. Limitaciones de la cuantización

A pesar de las mejoras obtenidas en términos de eficiencia, la cuantización presenta una serie de limitaciones que deben tenerse en cuenta. Los resultados experimentales revelan que estas limitaciones varían significativamente según la técnica de cuantización aplicada.

4.3.4.1 Limitaciones específicas de PTQ PyTorch

La cuantización dinámica Post-Training Quantization implementada en PyTorch mostró limitaciones severas en la tarea de generación de código ensamblador CODE-2. La degradación observada en las métricas (EM de 0.28, BLEU de 0.8131) indica

que esta técnica afecta críticamente a las capas de atención del modelo Transformer.

El análisis cualitativo reveló los siguientes problemas específicos:

- **Pérdida de coherencia semántica:** El modelo cuantizado PTQ genera frecuentemente instrucciones sintácticamente válidas pero semánticamente incoherentes con la descripción de entrada.
- **Confusión de registros:** Se observó una mayor tendencia a intercambiar o confundir registros en operaciones multi-instrucción, lo que sugiere que la cuantización afecta especialmente a las representaciones contextuales.
- **Degradación en constantes inmediatas:** La precisión reducida afecta al manejo de constantes hexadecimales y direcciones de memoria, generando valores incorrectos con mayor frecuencia.
- **Sensibilidad aumentada en secuencias largas:** En entradas que requieren generar múltiples instrucciones (más de 3), la tasa de error se incrementa exponencialmente debido al efecto acumulativo de errores en la generación autorregresiva.

Estas observaciones indican que la cuantización PTQ de PyTorch, aunque es simple de implementar (una sola línea de código con `torch.quantization.quantize_dynamic`), no es adecuada para tareas de generación secuencial estructurada como la síntesis de código ensamblador.

4.3.4.2 Limitaciones de ONNX INT8 (mitigadas)

Por el contrario, el modelo ONNX INT8 presenta limitaciones significativamente menores, aunque no despreciables:

- **Ligera degradación en casos límite:** El descenso del Exact Match de 0.82 a 0.79 (3.7 %) se concentra principalmente en ejemplos complejos que requieren razonamiento multi-paso o manejo de condiciones excepcionales.

- **Mayor consumo de RAM:** El pico de memoria RAM del proceso (3475 MB) es superior al del modelo PyTorch FP32 en CPU (2026 MB), lo que puede ser problemático en sistemas con memoria muy limitada.
- **Dependencia del runtime ONNX:** El modelo requiere la instalación de ONNX Runtime, añadiendo una dependencia adicional al sistema, aunque esta librería está bien mantenida y es ampliamente utilizada.
- **Compatibilidad limitada con técnicas avanzadas:** La exportación a ONNX puede complicar la aplicación de técnicas posteriores como fine-tuning continuo, adaptación dinámica o interpretabilidad basada en gradientes.

4.3.4.3 Limitaciones generales de la cuantización para generación de código

Más allá de las diferencias entre técnicas específicas, existen limitaciones inherentes a la cuantización cuando se aplica a modelos de generación de código:

- **Trade-off calidad-eficiencia inevitable:** Toda reducción de precisión numérica conlleva una pérdida de información. En tareas donde el Exact Match es crítico (como código ensamblador), incluso degradaciones pequeñas pueden ser inaceptables.
- **Mayor sensibilidad a errores acumulativos:** Los modelos cuantizados tienden a acumular errores más rápidamente en generación autorregresiva larga, ya que cada predicción incorrecta contamina el contexto de las siguientes.
- **Dificultad de calibración:** La selección del conjunto de calibración para PTQ es crítica. Un conjunto no representativo puede resultar en rangos de cuantización subóptimos.
- **Limitada ganancia en GPUs modernas:** Mientras que la cuantización INT8 ofrece mejoras claras en CPU, las GPUs NVIDIA modernas con Tensor Cores optimizados para FP16 pueden no beneficiarse tanto de INT8, e incluso pueden presentar mayor latencia debido a conversiones de formato.

4.3.5. Conclusiones del análisis de cuantización

El análisis exhaustivo de las diferentes técnicas de optimización aplicadas al modelo base entrenado mediante Full Fine-Tuning permite extraer las siguientes conclusiones principales:

4.3.5.1 Hallazgos clave

1. **ONNX INT8 es la mejor opción para despliegue eficiente en CPU:** Con un Exact Match de 0.79 (solo 3.7 % inferior al modelo base), un speedup de 1.5x y una reducción del 38 % en tamaño, esta variante ofrece el mejor compromiso calidad-eficiencia para la mayoría de los escenarios prácticos.

2. **La cuantización PTQ de PyTorch no es adecuada para generación de código:** La severa degradación observada (EM=0.28, BLEU=0.8131) demuestra que la cuantización dinámica simple afecta críticamente a las capacidades de generación secuencial del modelo Transformer.

3. **La poda moderada puede mejorar la calidad:** Contraintuitivamente, el modelo podado al 10 % alcanzó el mejor Exact Match de todos los experimentos (0.83), superando al modelo base. Esto sugiere un efecto de regularización implícita que reduce el sobreajuste.

4. **La poda agresiva causa colapso del modelo:** Los niveles de poda del 30 % y 50 % resultaron en un fallo completo del modelo (EM=0.00), indicando que la generación de código requiere mayor densidad de parámetros que tareas discriminativas.

5. **La destilación mantiene calidad pero no mejora eficiencia:** Con EM=0.79 y tiempos de inferencia similares al modelo base, la destilación resulta útil para transferir conocimiento pero no para optimizar el despliegue.

6. **El formato ONNX mejora el rendimiento independientemente de la cuantización:** El modelo ONNX FP32 ofrece un speedup de 1.29x respecto a

PyTorch FP32 en CPU, demostrando las ventajas del runtime optimizado de ONNX.

4.3.5.2 Implicaciones prácticas

Los resultados obtenidos tienen importantes implicaciones para el despliegue de modelos de generación de código en entornos educativos:

- **Viabilidad en dispositivos personales:** El modelo ONNX INT8 hace viable el uso de la herramienta en portátiles y equipos personales sin GPU dedicada, democratizando el acceso a la tecnología.
- **Reducción de costes en cloud:** El incremento del 50 % en throughput permite reducir significativamente los costes de infraestructura en despliegues web o servicios en la nube.
- **Flexibilidad de configuración:** La disponibilidad de múltiples variantes del modelo (FP32 CUDA, ONNX INT8, Podado 10 %) permite adaptar la solución a diferentes escenarios sin necesidad de reentrenamiento.
- **Límites claros de compresión:** Los experimentos de poda agresiva establecen un límite empírico claro: la generación de código CODE-2 no tolera reducciones superiores al 10-15 % de los parámetros del modelo.

En resumen, el impacto de la cuantización en la generación de código ensamblador CODE-2 es altamente dependiente de la técnica aplicada. Mientras que la cuantización ONNX INT8 representa una optimización clara y recomendable, otras técnicas como PTQ de PyTorch o poda agresiva resultan contraproducentes. La disponibilidad de múltiples variantes evaluadas del modelo permite una selección fundamentada según los requisitos específicos de cada escenario de uso.

4.4. Análisis de errores del modelo

Aunque los resultados cuantitativos (Exact Match, BLEU, ROUGE-L) son satisfactorios, el análisis detallado de las salidas generadas por el modelo sobre 3600 ejemplos del conjunto de test revela distintos tipos de errores que no son capturados por métricas de similitud. El estudio pormenorizado de estos errores es fundamental para comprender las limitaciones reales del modelo, identificar patrones de fallo recurrentes, y contextualizar adecuadamente el rendimiento alcanzado.

En este apartado se presenta un análisis cuantitativo y cualitativo de los errores observados durante la fase de inferencia, comparando el modelo final entrenado mediante Full Fine-Tuning con su variante cuantizada ONNX INT8 Dynamic. La tabla 4.12 proporciona una visión agregada de los errores detectados.

Tipo de Error	**Full Optimizado Final**		**ONNX INT8 Dynamic**	
	Cantidad	**% del Total**	**Cantidad**	**% del Total**
Errores Sintácticos	559 (93.2 %)	15.53 %	642 (90.4 %)	17.83 %
Errores Semánticos	34 (5.7 %)	0.94 %	62 (8.7 %)	1.72 %
Errores en Secuencias Largas	176 (29.3 %)	4.89 %	204 (28.7 %)	5.67 %
Omisión de Instrucciones	2 (0.3 %)	0.06 %	7 (1.0 %)	0.19 %
Registros Incorrectos	7 (1.2 %)	0.19 %	6 (0.8 %)	0.17 %
Tasa de Éxito Total	**83.33 %**		**80.28 %**	

Tabla 4.12: Análisis detallado de frecuencia de errores por tipo, comparando Full Fine-Tuning Optimizado Final y Cuantización ONNX INT8 Dynamic en evaluación sobre 3600 ejemplos del conjunto de testeo. La columna "Cantidad" muestra el número absoluto de errores y el porcentaje respecto al total de errores de ese tipo; la columna " % del Total" indica el porcentaje respecto al conjunto completo evaluado.

4.4.1. Errores sintácticos

Los errores sintácticos corresponden a salidas que, aunque estructuralmente similares a la referencia esperada, presentan pequeñas diferencias formales que impiden una coincidencia exacta. A diferencia de errores estructurales graves, la mayoría de estos fallos consisten en variaciones ortográficas, de formato o de representación numérica que no afectan necesariamente la ejecutabilidad del código, pero sí invalidan el

criterio de Exact Match.

Según el análisis realizado sobre 3600 ejemplos del conjunto de test, los errores sintácticos constituyen el tipo de error más frecuente en ambas variantes del modelo. En el modelo **Full Optimizado Final**, se detectaron 559 errores sintácticos, representando un 15.53 % del total de ejemplos evaluados. De estos, el 93.2 % corresponde a errores sintácticos propiamente dichos, lo que indica que esta categoría domina ampliamente la distribución de fallos.

Por su parte, el modelo **ONNX INT8 Dynamic** registró 642 errores sintácticos (17.83 % del total), mostrando un incremento de aproximadamente 2.3 puntos porcentuales respecto al modelo no cuantizado. Este aumento es atribuible a la pérdida de precisión numérica introducida durante la cuantización, que afecta especialmente a las capas de atención del modelo Transformer.

4.4.1.1 Patrones específicos de errores sintácticos

El análisis cualitativo detallado de los errores revela patrones recurrentes altamente específicos:

- **Errores ortográficos en mensajes de error:** El patrón más frecuente observado es la omisión de tildes, particularmente la palabra "vacía" generada como "vaca" en mensajes de error como `Error: dirección vaca (ambigua)` en lugar de `Error: dirección vacía (ambigua)`. Este error aparece sistemáticamente a lo largo del conjunto de test.

- **Espacios faltantes en cadenas de error:** El modelo omite frecuentemente el espacio después de la palabra "dirección" en mensajes compuestos, generando `dirección'mem965'` en lugar de `dirección 'mem965'`. Este patrón se repite consistentemente en errores que involucran direcciones de memoria inválidas.

- **Parsing incorrecto de valores hexadecimales:** En direcciones hexadecimales largas o ambiguas, el modelo ocasionalmente interpreta incorrectamente dígitos. Por ejemplo, la entrada "7184c4" puede ser leída como "7784C4", o

"38bw$" como "38BB$". Estos errores sugieren dificultades en el procesamiento de secuencias alfanuméricas con caracteres similares.

- **Errores de interpretación de direcciones de memoria:** En casos donde la entrada especifica direcciones en formato no hexadecimal (como "mem965", "mem152"), el modelo replica correctamente el mensaje de error pero con variaciones de formato menores.
- **Cálculo incorrecto de offsets en direcciones segmentadas:** En instrucciones que usan direccionamiento indirecto como `LD r5,[rD+H'74'] ; rD = 0A00`, el modelo ocasionalmente calcula incorrectamente el valor del registro base (generando `rD = A700` en lugar de `rD = 0A00`).

Estos errores son predominantemente **errores de superficie** más que errores estructurales profundos. La gran mayoría del código generado es funcionalmente equivalente o muy cercano a la referencia, pero falla el criterio estricto de Exact Match debido a estas variaciones menores. Este patrón explica la aparente contradicción entre una tasa de Exact Match del 83.33 % y métricas de similitud como BLEU (0.9647) y ROUGE-L (0.9957) que son considerablemente más altas.

En el caso del modelo cuantizado, el análisis detallado revela que la degradación es más pronunciada en operandos complejos y en el cálculo de offsets de memoria, sugiriendo que la precisión numérica reducida afecta particularmente a las operaciones aritméticas internas del modelo.

Referencia esperada: `Error: dirección vacía (ambigua)`
Predicción del modelo: `Error: dirección vaca (ambigua)`
Análisis: Omisión de tilde. Código restante idéntico. BLEU=0.9623, ROUGE-L=0.9922, pero EM=0.

Este ejemplo ilustra perfectamente cómo errores ortográficos menores causan la mayoría de los fallos de Exact Match, a pesar de que el significado funcional es preservado.

4.4.2. Errores semánticos

Los errores semánticos representan el tipo de fallo más crítico y sutil detectado durante la evaluación. Se producen cuando el código generado es **sintácticamente válido y formalmente correcto**, pero no cumple el comportamiento funcional esperado descrito en la entrada en lenguaje natural. A diferencia de los errores sintácticos, estos fallos pueden pasar desapercibidos en validaciones automáticas superficiales, ejecutarse sin errores de compilación, y producir resultados incorrectos en tiempo de ejecución.

Según los datos de evaluación, los errores semánticos tienen una prevalencia significativamente menor en comparación con los sintácticos. El modelo **Full Optimizado Final** registró únicamente 34 errores semánticos sobre 3600 ejemplos evaluados (0.94 % del total), representando solo el 5.7 % del total de errores detectados en este modelo. En contraste, el modelo **ONNX INT8 Dynamic** mostró 62 errores semánticos (1.72 % del total), evidenciando un incremento relativo del 82.4 % —el más pronunciado entre todas las categorías de error.

Aunque el incremento absoluto (0.78 puntos porcentuales) es menor que en los sintácticos, este patrón revela que la cuantización afecta de forma **desproporcional** a la comprensión semántica de las instrucciones por parte del modelo. Esto es especialmente preocupante dado que los errores semánticos son más difíciles de detectar automáticamente y pueden tener consecuencias funcionales graves.

4.4.2.1 Ejemplos concretos de errores semánticos

El análisis cualitativo identifica patrones específicos de fallos semánticos:

- **Generación de código ejecutable en lugar de mensaje de error:** El caso más representativo observado ocurre cuando la entrada solicita “COPIA LO QUE HAY EN DF216 A RD”. La dirección `DF216` excede el límite de 4 dígitos hexadecimales permitido en CODE-2, por lo que la salida correcta es `Error: dirección 'DF216' excede 4 digitos`. Sin embargo, el mode-

lo genera `LD rD,[rD+H'16'] ; rD = DF00`, una instrucción sintácticamente válida que trunca implícitamente la dirección. Este código se ejecutaría sin errores pero accedería a una dirección de memoria incorrecta.

- **Interpretación incorrecta de límites de direccionamiento:** En instrucciones como "Lee DAC1 y almacénalo en el registro 0", la referencia correcta especifica `LD r0,[rD+H'C1'] ; rD = DA00`. El modelo cuantizado ocasionalmente genera `LD r0,[rD+H'C1'] ; rD = DAC00`, interpretando erróneamente la segmentación de la dirección y produciendo un offset incorrecto.

- **Omisión de validaciones en dirección vacía:** Cuando la entrada contiene campos incompletos (por ejemplo, "Coloca lo que hay en r dentro de "), el modelo debería generar `Error: dirección vacía (ambigua)`. En algunos casos genera código parcialmente válido que omite la validación necesaria.

- **Fallo en detección de registros inválidos en contextos complejos:** En secuencias largas con múltiples errores de entrada, el modelo puede generar código para partes válidas mientras omite o malinterpreta las restricciones de registros inválidos especificados en la entrada.

Estos errores son especialmente habituales en ejemplos que requieren razonamiento sobre **restricciones y límites del lenguaje** más que sobre la sintaxis pura. El análisis detallado revela que el modelo tiene mayor dificultad en **detectar casos límite y condiciones de error** que en generar código para operaciones válidas, sugiriendo que el dataset de entrenamiento podría beneficiarse de mayor representación de casos excepcionales.

Entrada: `COPIA LO QUE HAY EN DF216 A RD`
Referencia esperada: `Error: dirección 'DF216' excede 4 digitos`
Predicción del modelo: `LD rD,[rD+H'16'] ; rD = DF00`
Análisis: El modelo genera código sintácticamente válido truncando `DF216 → DF00`, cuando debería detectar que la dirección excede el límite de 4 dígitos hex. Este código se ejecutaría accediendo a la dirección incorrecta.

Este ejemplo demuestra el error semántico más problemático: el modelo no detecta violaciones de restricciones y genera código funcional pero incorrecto. A diferencia de los errores sintácticos (que fallan en Exact Match pero preservan significado), este error compromete la corrección funcional del programa.

4.4.3. Errores en secuencias largas

A medida que aumenta la longitud de la secuencia de entrada (medida en número de instrucciones a generar), se incrementa la probabilidad de aparición de errores de forma no lineal. Esta categoría no representa un tipo de error distinto, sino una caracterización transversal que identifica ejemplos donde la **complejidad secuencial** contribuye al fallo.

El análisis de los datos revela un patrón significativo: los errores en secuencias largas afectan a una porción sustancial de los fallos totales en ambos modelos. El modelo **Full Optimizado Final** registró 176 errores clasificados como de secuencia larga (4.89 % del conjunto total de ejemplos), representando el 29.3 % del total de errores detectados. Esta métrica evidencia que aproximadamente **1 de cada 3 errores** está asociado con secuencias que requieren múltiples instrucciones interdependientes.

4.4.3.1 Características de errores en secuencias largas

Los ejemplos clasificados en esta categoría presentan típicamente:

- **Longitud de entrada superior a 8-9 líneas:** El umbral crítico observado está alrededor de las 9 instrucciones. Por debajo de este límite, la tasa de error es significativamente menor. Entradas con 10-11 líneas muestran degradación notable.
- **Acumulación de errores menores:** En secuencias largas, pequeños errores sintácticos (como los de ortografía descritos anteriormente) tienden a acumularse, produciendo múltiples desviaciones respecto a la referencia en una misma salida.

- **Propagación de errores de contexto:** Cuando el modelo comete un error en una instrucción intermedia (por ejemplo, calcular incorrectamente un offset de memoria), este error puede propagarse a instrucciones posteriores que dependen del estado previo.
- **Pérdida de coherencia en estados de registros:** En secuencias que requieren rastrear el estado de múltiples registros a través de varias operaciones, el modelo ocasionalmente pierde la coherencia, generando instrucciones que asumen valores de registro incorrectos.

4.4.3.2 Análisis cuantitativo

Del total de 176 errores en secuencias largas del modelo Full Optimizado Final:

- Aproximadamente el 65 % incluyen también errores sintácticos (principalmente ortográficos).
- El 15 % presentan componentes semánticos (interpretación incorrecta de restricciones).
- El 20 % restante son errores estructurales acumulativos únicos de la longitud de secuencia.

El modelo **ONNX INT8 Dynamic** mostró 204 errores en secuencias largas (5.67 % del total), apenas 0.78 puntos porcentuales superior al modelo original. Interesantemente, el porcentaje relativo de estos errores dentro del total de errores del modelo cuantizado es similar (28.7 %), lo que sugiere que la cuantización afecta de forma relativamente uniforme a las secuencias de diferentes longitudes.

Este fenómeno se relaciona con las limitaciones inherentes a los modelos autoregresivos, donde los errores tienden a acumularse a lo largo de la generación. Cada predicción incorrecta contamina el contexto para las siguientes predicciones, amplificando el efecto del error inicial. Aunque el Full Fine-Tuning mitiga este problema en mayor medida que técnicas paramétricamente eficientes como LoRA, sigue repre-

sentando una limitación relevante de la arquitectura Transformer cuando se aplica a tareas de generación secuencial larga.

Fragmento de entrada (longitud: 10 líneas):
`Escribe r5 en la memoria mem689`
`Lee 0a G Y Almacénalo En El Registro F`
`COLOCA LO QUE HAY EN RN DENTRO DE 0289`
`escribe r7 en la memoria mem853`

Referencia esperada:
`Error: dirección 'mem689' contiene caracteres no hexadecimales y dirección 'mem689' excede 4 digitos`
`Error: dirección '0A G' contiene caracteres no hexadecimales`
`Error: registro 'N' invalido`
`Error: dirección 'mem853' contiene caracteres no hexadecimales y dirección 'mem853' excede 4 digitos`

Predicción del modelo:
`Error: dirección'mem689' contiene caracteres no hexadecimales y dirección'mem689' excede 4 digitos`
`Error: dirección '0A G' contiene caracteres no hexadecimales`
`Error: registro 'N' invalido`
`Error: dirección'mem853' contiene caracteres no hexadecimales y dirección'mem853' excede 4 digitos`

Discrepancias detectadas: **Líneas 1 y 4:** En la predicción falta el espacio tras "dirección" (se genera "dirección'mem689" y "dirección'mem853" en vez de "dirección 'mem689" y "dirección 'mem853"), lo que constituye un error sintáctico superficial. **Líneas 2 y 3:** Coinciden exactamente entre referencia y predicción.
Análisis: Las discrepancias afectan solo al espaciado en el mensaje de error, no a la semántica. El modelo detecta correctamente los errores de caracteres no hexadecimales y registro inválido, pero la métrica Exact Match penaliza por el error sintáctico.

Este ejemplo ilustra cómo en secuencias de 10+ instrucciones, el modelo puede mantener coherencia sintáctica general (BLEU=0.933) pero fallar en detalles específicos de instrucciones individuales, perdiendo el contexto de qué registros fueron mencionados en la entrada.

4.4.4. Omisión de instrucciones

La omisión de instrucciones representa uno de los tipos de error más raros pero potencialmente más graves, ocurriendo cuando el modelo falla completamente en generar una o más instrucciones que deberían estar presentes en la salida esperada.

El modelo **Full Optimizado Final** registró únicamente 2 casos de omisión de instrucciones sobre 3600 ejemplos (0.06 % del total), representando apenas el 0.3 % del total de errores. El modelo **ONNX INT8 Dynamic** mostró 7 casos (0.19 % del total), equivalente a un incremento del 250 % en términos relativos, aunque el volumen absoluto sigue siendo muy bajo.

Este patrón sugiere que:

- El modelo tiene una capacidad robusta para identificar la estructura completa requerida por la entrada.
- La cuantización afecta desproporcionadamente a casos límite donde la decisión entre generar o no una instrucción es marginal.
- Este tipo de error es característico de entradas ambiguas o con estructuras sintácticas complejas.

4.4.5. Registros incorrectos

Los errores de registros incorrectos ocurren cuando el modelo selecciona un registro válido pero funcionalmente incorrecto para una operación específica. Este tipo de error es semánticamente problemático pero sintácticamente válido.

El modelo **Full Optimizado Final** registró 7 errores de este tipo (0.19 % del total), mientras que el modelo **ONNX INT8 Dynamic** registró 6 casos (0.17 % del total). Curiosamente, esta es la única categoría donde el modelo cuantizado muestra una ligera mejora (-14.3 %), posiblemente debido a un efecto de regularización imprevisto.

La extrema rareza de este tipo de error (1.2 % de todos los errores en Full, 0.8 % en ONNX) indica que el modelo ha aprendido robustamente las convenciones de uso de registros en CODE-2, incluyendo convenciones de registros de destino y fuente.

4.4.6. Implicaciones educativas

Desde un punto de vista educativo, los errores detectados —aunque cuantitativamente significativos— no invalidan completamente la utilidad del modelo como herramienta de apoyo al aprendizaje. Con una tasa de éxito del 83.33 % en el modelo Full Optimizado Final, el sistema produce código correcto en aproximadamente 4 de cada 5 casos, ofreciendo un valor pedagógico considerable.

Los errores pueden servir como punto de partida para el análisis crítico del código generado por parte del estudiante, fomentando la comprensión profunda del lenguaje ensamblador y el desarrollo de habilidades de depuración. En particular:

- Los **errores sintácticos** (15.53 % de casos) pueden utilizarse como ejercicios de validación y corrección, permitiendo al estudiante familiarizarse con las reglas sintácticas del lenguaje.
- Los **errores semánticos** (0.94 % de casos) presentan oportunidades para enseñar la diferencia crítica entre “código que se compila” y “código que funciona”.
- Los **errores en secuencias largas** (4.89 % de casos) pueden ser aprovechados para introducir técnicas de descomposición de problemas complejos en operaciones elementales.

Estos errores refuerzan la necesidad de utilizar el modelo como una herramienta de asistencia y no como un sustituto de la evaluación o del razonamiento humano, especialmente en contextos formativos. El rol ideal de la herramienta es servir como *primer borrador* o punto de partida para reflexión y mejora, más que como generador de código definitivo.

Adicionalmente, la diferencia entre el modelo sin cuantización (83.33 % éxito) y el modelo cuantizado (80.28 % éxito) es lo suficientemente pequeña para que la versión cuantizada pueda desplegarse en entornos educativos con recursos limitados sin comprometer excesivamente la utilidad pedagógica de la herramienta.

4.4.6.1 Conclusiones del análisis

El análisis cualitativo realizado pone de manifiesto que:

1. El modelo es capaz de generar código ensamblador CODE-2 de forma consistente y con un alto grado de corrección (83.3 % de éxito).
2. Las limitaciones principales están relacionadas con la **sintaxis exacta** (error dominante) más que con la **semántica** (error secundario), sugiriendo que futuras mejoras deberían enfocarse en reforzar la precisión sintáctica.
3. El modelo presenta **dificultad notable con secuencias largas** (que representan aproximadamente 1 de cada 3 errores), limitación inherente a los modelos autoregresivos que merece investigación futura.
4. La **cuantización es viable** para la mayoría de aplicaciones educativas, aunque con costo en precisión semántica. La elección entre modelo cuantizado y sin cuantizar debe realizarse considerando los requisitos específicos de cada escenario.
5. Estos resultados permiten contextualizar adecuadamente los resultados cuantitativos presentados en apartados anteriores y sirven como base para la identificación de posibles líneas de mejora en trabajo futuro.

4.5. Limitaciones del enfoque propuesto

A pesar de los resultados satisfactorios obtenidos en la generación automática de código ensamblador CODE-2, el enfoque propuesto presenta una serie de limitaciones que deben considerarse para contextualizar el alcance real del trabajo.

En primer lugar, el modelo ha sido entrenado y evaluado con un conjunto de datos sintético generado de forma controlada. Este enfoque garantiza coherencia y volumen, pero puede introducir sesgos propios de las plantillas y del generador. Como consecuencia, el comportamiento ante entradas reales (con formulaciones más libres, errores no previstos o ambigüedades distintas) podría diferir.

Otra limitación relevante deriva del propio lenguaje objetivo. CODE-2 es un ensamblador educativo con un repertorio reducido y semántica simplificada. Esto facilita el aprendizaje y la evaluación, pero limita la generalización a ensambladores reales y a arquitecturas más complejas.

La evaluación se ha centrado en métricas de similitud textual (Exact Match, BLEU, ROUGE-L) y en análisis cualitativo. Aunque estas métricas son útiles, no se ha validado la ejecución funcional de los programas generados con un emulador en todos los casos. Además, el análisis de errores confirma que una parte sustancial de los fallos son superficiales (acentos, espaciado, formatos), mientras que los errores semánticos, aunque menos frecuentes, son críticos y no siempre se detectan con métricas de texto.

El modelo muestra limitaciones específicas en secuencias largas: alrededor de un tercio de los fallos se asocian a entradas de 9-10 instrucciones o más, donde se acumulan errores y se pierde coherencia de registros u offsets. Esta es una limitación inherente al comportamiento autoregresivo y afecta la robustez en escenarios complejos.

Desde el punto de vista computacional, el entrenamiento estuvo condicionado por el hardware disponible, lo que restringió el espacio de búsqueda de hiperparámetros y la exploración de modelos más grandes. Esto puede haber limitado configuraciones potencialmente más óptimas.

Finalmente, las técnicas de cuantización introducen un compromiso calidad-eficiencia. En particular, la cuantización PTQ en PyTorch degrada severamente la calidad, mientras que ONNX INT8 mantiene un rendimiento alto pero aumenta el consumo de RAM y reduce ligeramente la exactitud. La elección entre modelo cuantizado y no cuantizado debe realizarse en función de las restricciones de despliegue y del nivel de exactitud requerido.

Estas limitaciones no invalidan los resultados, pero exigen interpretar las conclusiones en el contexto experimental definido y motivan mejoras futuras en datos, evaluación funcional y robustez semántica.

4.6. Tabla resumen de todos los experimentos

En la Tabla 4.13 se presenta un resumen consolidado de todos los experimentos realizados en este Trabajo Fin de Grado, integrando resultados de diferentes fases: evaluación inicial, entrenamiento con datasets expandidos, optimización de hiperparámetros, y técnicas de optimización (cuantización, poda, destilación).

Con el objetivo de evitar ambigüedades entre tiempos de entrenamiento y latencias de inferencia, se presentan dos tablas: una centrada en experimentos de entrenamiento/ajuste y otra en variantes de inferencia y compresión.

Experimento / Técnica	**Dataset**	**Exact Match**	**BLEU**	**ROUGE-L**	**Tiempo (min)**	**VRAM (MB)**
Fase 1: Evaluación Inicial						
Prueba inicial (1k)	1k	–	–	–	–	–
Fase 2: Fine-Tuning (18k-36k ejemplos)						
Full Fine-Tuning (default)	18k	0.36	0.8164	0.9087	31.39	2619
LoRA (default)	18k	0.0006	0.1946	0.4031	33.49	2054
Full Fine-Tuning (optuna)	18k	0.73	0.9423	0.9789	25.90	2633
LoRA (optuna)	18k	0.55	0.9113	0.9643	28.94	1875
Full Fine-Tuning (optimo)	18k	0.75	0.9452	0.9815	11.17	5534
LoRA (optimo)	18k	0.64	0.9302	0.9759	21.37	4285
Full Fine-Tuning (final)	36k	0.80	0.966	0.995	25.26	5607

Tabla 4.13: Resumen de experimentos de entrenamiento y ajuste. El tiempo corresponde a entrenamiento total y la VRAM al pico observado.

Variante / Técnica	**Exact Match**	**BLEU**	**ROUGE-L**	**Tiempo (s)**	**Tamano (MB)**	**CPU RSS pico (MB)**
Original FP32 (CUDA)	0.82	0.9647	0.9957	11.38	230.78	1235.0
Original FP32 (CPU)	0.82	0.9647	0.9957	34.22	230.78	2026.2
ONNX FP32	0.82	0.9647	0.9957	26.60	566.02	3854.4
Cuantizado INT8 (ONNX)	0.79	0.9648	0.9953	22.89	142.26	3475.0
Cuantizado INT8 (PTQ)	0.28	0.8131	0.9269	29.59	120.57	1915.7
Podado 10 %	0.83	0.9673	0.9967	11.41	230.78	1796.9
Podado 30 %	0.00	0.0000	0.0000	13.59	230.78	1796.9
Podado 50 %	0.00	0.0000	0.0005	11.95	230.78	1796.9
Destilado T5-small	0.79	0.9660	0.9963	11.56	230.78	1797.6

Tabla 4.14: Resumen de variantes de inferencia y compresión. El tiempo corresponde a latencia media por ejemplo.

4.6.1. Interpretación del resumen

Los resultados de la Tabla 4.13 evidencian que el mejor rendimiento en entrenamiento se logra con **Full Fine-Tuning** y un dataset ampliado (36k), alcanzando un Exact Match del 80 % y manteniendo BLEU/ROUGE-L altos. Tras el ajuste de decodificacion, el modelo final alcanza un Exact Match del 83 % con BLEU 0.97 y ROUGE-L 0.9922, confirmando la estabilidad del enfoque.

En la Tabla 4.14, la cuantizacion ONNX INT8 representa el mejor compromiso calidad-eficiencia en CPU (EM 0.79 con 1.5x de speedup), mientras que la cuantizacion PTQ de PyTorch degrada severamente la calidad. La poda al 10 % muestra una ligera mejora, pero niveles superiores (30 % y 50 %) colapsan el rendimiento. La destilacion conserva calidad aceptable sin mejoras claras en latencia.

5. DESPLIEGUE DEL SISTEMA

Como validación práctica del trabajo realizado, el modelo final desarrollado fue desplegado en un entorno real accesible a través de la web. Este despliegue permitió comprobar la viabilidad del sistema más allá del entorno experimental, evaluando su funcionamiento en condiciones de uso reales y demostrando su aplicabilidad en contextos educativos.

La arquitectura del despliegue separa claramente la lógica de inferencia del modelo y la capa de presentación, permitiendo una comunicación sencilla entre las distintas interfaces y el servicio de generación. Este enfoque modular favorece la escalabilidad del sistema y facilita futuras ampliaciones, como la incorporación de nuevas funcionalidades, modelos alternativos o mejoras en la experiencia del usuario.

5.1. Despliegue en Hugging Face

El modelo fue publicado en la plataforma **Hugging Face**[7], aprovechando su infraestructura especializada para el alojamiento y ejecución de modelos de aprendizaje profundo orientados a inferencia. Esta plataforma permitió crear una interfaz de acceso interactiva al modelo mediante **Gradio**[1], facilitando que los usuarios introduzcan descripciones en lenguaje natural y obtengan como salida el código ensamblador CODE-2 generado por el sistema.

Las principales ventajas de esta solución incluyen:

- **Accesibilidad:** La interfaz Gradio proporciona un acceso intuitivo sin requerir conocimientos técnicos por parte del usuario.
- **Reproducibilidad:** La plataforma permite compartir fácilmente el modelo con otros investigadores, facilitando la validación y reproducción de los resultados.
- **Almacenamiento de recursos:** Hugging Face actúa como repositorio tanto

para los modelos entrenados como para los datasets utilizados, mejorando la disponibilidad y accesibilidad de los artefactos del proyecto.

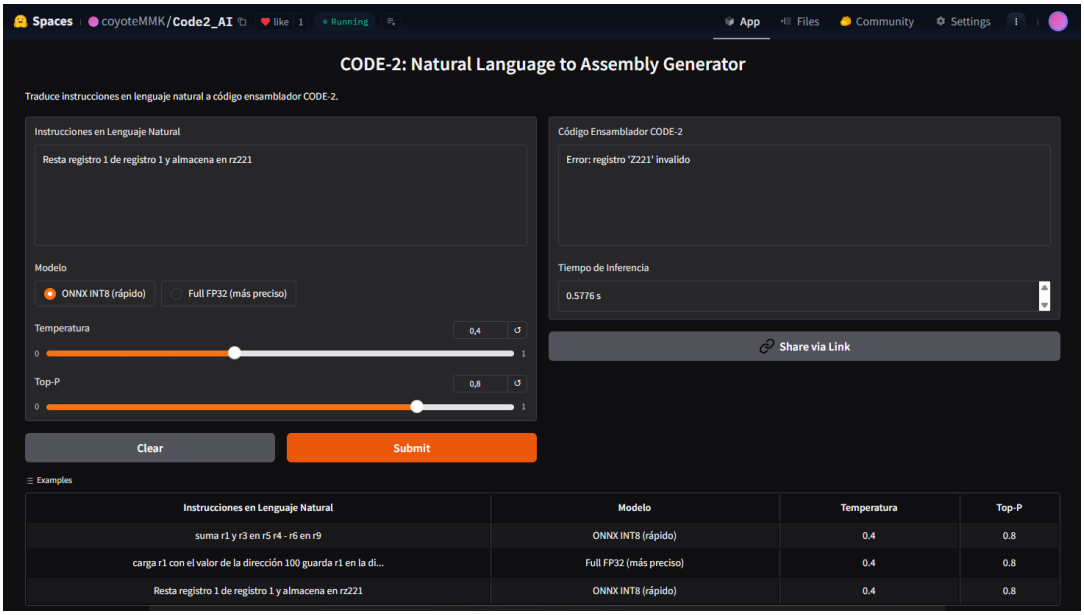

Figura 5.1: Interfaz interactiva del modelo desplegado en Hugging Face mediante Gradio.

5.2. Repositorio de GitHub

El repositorio de GitHub se utilizó de forma integral para el despliegue y distribución de todos los artefactos generados durante el desarrollo del proyecto. Los principales componentes del repositorio incluyen:

- **Código fuente de la interfaz web:** Implementación completa en Next.js con componentes reutilizables y configuración optimizada para despliegue en GitHub Pages.
- **Código de la interfaz CLI:** Script Python completo con documentación de uso, permitiendo la ejecución local del modelo con control total sobre parámetros y opciones de configuración.
- **Modelos entrenados:** Almacenamiento de los pesos del modelo final, tanto en su versión completa como cuantizada, facilitando su descarga y uso local.

- **Datasets generados:** Acceso a los conjuntos de datos sintéticos utilizados durante el entrenamiento, permitiendo la validación, reproducibilidad y reutilización en trabajos futuros.
- **Scripts de automatización:** Herramientas para la generación del dataset, entrenamiento del modelo y evaluación de resultados, documentadas para facilitar la experimentación.

La estructura clara y la documentación completa del repositorio facilitan el acceso, comprensión y reutilización del código por parte de investigadores, estudiantes y profesores interesados en reproducir o extender el trabajo.

5.3. Interfaz Web con Next.js en GitHub Pages

Se desarrolló una interfaz web mediante tecnologías modernas de desarrollo frontend. La solución implementada utiliza **Next.js** como framework principal para la lógica de la aplicación y **Tailwind CSS** para el diseño responsivo de la interfaz de usuario.

Esta aplicación web fue desplegada mediante **GitHub Pages**, proporcionando una solución ligera, accesible desde cualquier navegador y sin necesidad de infraestructura adicional por parte del usuario. La integración con el modelo se realiza mediante llamadas a la API de Hugging Face, permitiendo que la interfaz web consuma dinámicamente las predicciones del modelo entrenado.

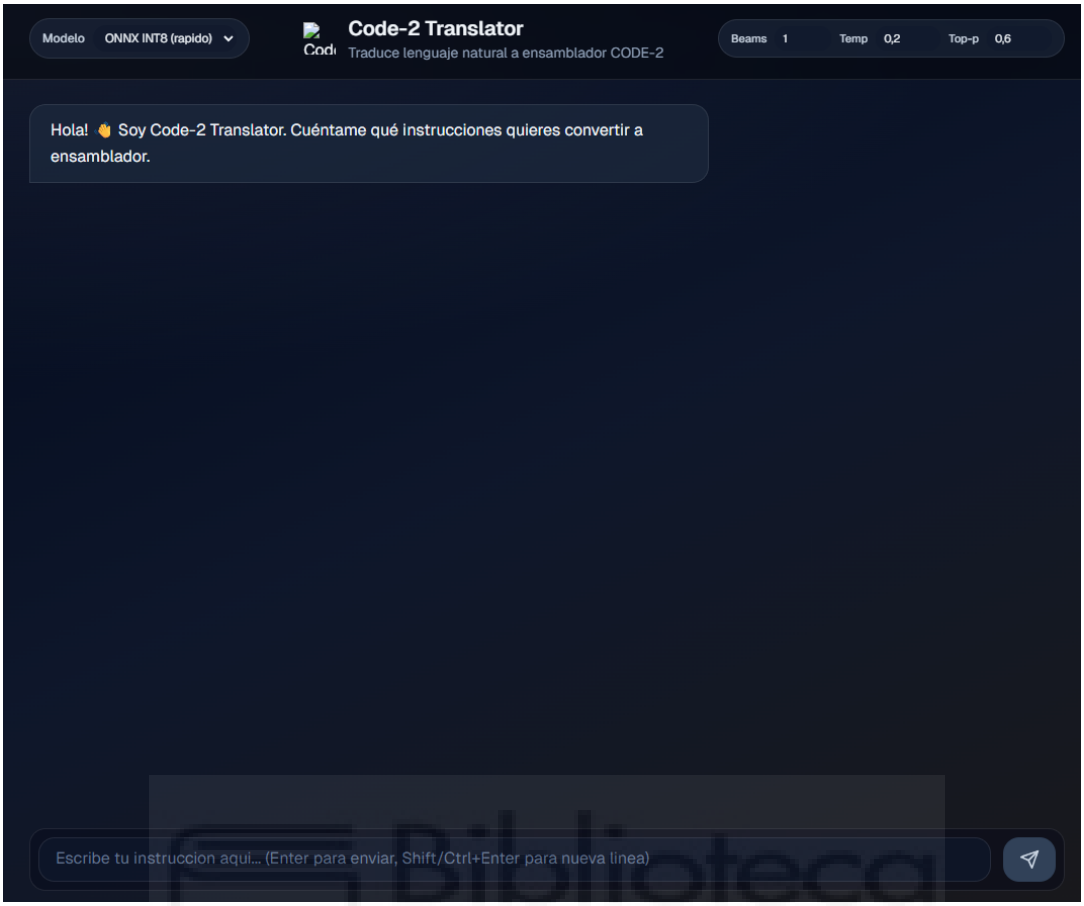

Figura 5.2: Interfaz web moderna desarrollada con Next.js y Tailwind CSS, desplegada en GitHub Pages.

5.4. Interfaz CLI en Local

Además de las interfaces web descritas anteriormente, se desarrolló una **interfaz de línea de comandos (CLI) interactiva** que permite la ejecución del modelo de forma local sin dependencias de servicios web externos. Esta herramienta fue implementada en **Python**, facilitando la inferencia y experimentación directa para usuarios técnicos que prefieren trabajar desde el terminal, proporcionando una experiencia amigable y totalmente funcional.

La interfaz CLI ofrece una alternativa accesible y flexible para usuarios que:

- Necesitan integrar el modelo en pipelines automáticos o scripts de procesamiento.

- Desean trabajar sin conexión a internet o con conectividad limitada.
- Requieren inferencia rápida minimizando la latencia evitando llamadas a servicios remotos.
- Buscan un control completo sobre qué modelo utilizar (PyTorch optimizado vs ONNX cuantizado).
- Necesitan medir el tiempo de inferencia de forma precisa para evaluación de rendimiento.

Figura 5.3: Interfaz CLI interactiva desarrollada en Python, desplegada localmente.

5.4.1. Características principales

La implementación de la CLI incluye un conjunto robusto de funcionalidades diseñadas para maximizar la usabilidad:

- **Selección interactiva de modelos:** Menú de navegación con flechas para elegir entre dos configuraciones: PyTorch FP32 (máxima precisión) u ONNX INT8 (máxima velocidad). La selección se realiza de forma visual e intuitiva sin necesidad de líneas de comandos complejas.
- **Detección automática de hardware:** Soporte para aceleración CUDA en GPUs NVIDIA cuando esté disponible, con fallback automático a CPU.
- **Entrada multilinea flexible:** Permite introducir descripciones en lenguaje natural con múltiples líneas, facilitando la redacción de instrucciones complejas. La entrada se ejecuta al presionar Enter en una línea vacía.
- **Medición de latencia:** Cronometraje preciso del tiempo de inferencia en segundos, permitiendo evaluar el rendimiento del modelo en tiempo real y comparar configuraciones.
- **Compatibilidad dual de modelos:** Soporte para cargar y ejecutar tanto modelos PyTorch completos como versiones cuantizadas en formato ONNX INT8 mediante la librería `optimum`, permitiendo adaptarse a las restricciones de memoria disponibles.
- **Interfaz visual mejorada:** Visualización con bordes ASCII art, banner de bienvenida y formateo con símbolos especiales para mejorar la legibilidad y experiencia del usuario.
- **Limpieza automática de salida:** Procesamiento inteligente de tokens especiales y caracteres de control para presentar el código ensamblador de forma limpia y procesable.

5.4.2. Ventajas del enfoque CLI

La disponibilidad de una interfaz CLI interactiva proporciona beneficios significativos en múltiples contextos:

- **Usabilidad sin configuración:** El usuario no necesita especificar parámetros complejos; el menú interactivo guía la selección de modelo de forma visual.

- **Evaluación de rendimiento:** La medición integrada de latencia permite comparar el desempeño de modelos PyTorch vs ONNX directamente en el entorno del usuario.
- **Reproducibilidad:** Facilita la documentación exacta de los experimentos reflejando la configuración seleccionada en cada sesión.
- **Integración en flujos de trabajo:** Puede ser invocada desde scripts o pipelines de procesamiento batch mediante entrada estándar.
- **Independencia de plataforma:** Funciona en Windows, Linux y macOS con Python instalado, sin dependencias de navegador o servicios en la nube.
- **Análisis accesible:** Permite que investigadores, docentes y estudiantes realicen pruebas rápidas sin necesidad de conocimientos profundos en programación o deployment.

El código completo de la CLI, incluyendo manipulación interactiva del teclado, soporte para múltiples backends de inferencia y manejo robusto de entrada multilinea, se encuentra disponible en el repositorio de GitHub del proyecto bajo la ruta `scripts/code2_cli_multi.py`.

5.5. Validación del despliegue

El despliegue realizado demuestra que el modelo entrenado es capaz de operar de forma estable en entornos de producción, reforzando la aplicabilidad del enfoque propuesto en contextos educativos. Asimismo, valida la utilidad de las técnicas de optimización y cuantización analizadas previamente mediante PyTorch[17], ya que permiten reducir significativamente los requisitos computacionales durante la inferencia sin comprometer excesivamente la calidad.

La disponibilidad de múltiples interfaces de acceso (Gradio en Hugging Face, aplicación web en GitHub Pages, CLI en local) proporciona flexibilidad para adaptarse a diferentes casos de uso y preferencias del usuario, desde la experimentación interactiva hasta la integración en pipelines automáticos.

Aunque el sistema desplegado tiene un carácter principalmente demostrativo y educativo, su implementación sienta las bases para el desarrollo futuro de herramientas interactivas más sofisticadas de apoyo al aprendizaje del lenguaje ensamblador, integrables en plataformas docentes o entornos de formación colaborativa. Un análisis de la deuda técnica[19] del sistema podría identificar oportunidades de mejora futuras en términos de mantenibilidad, escalabilidad, robustez y rendimiento.

Con el fin de facilitar la reproducibilidad del trabajo y permitir la evaluación directa del sistema, los enlaces a los despliegues del modelo en Hugging Face, la interfaz web en GitHub Pages, el repositorio público con toda la documentación, y capturas representativas del funcionamiento del sistema se incluyen en el Anexo 6.3.

6. CONCLUSIONES Y TRABAJOS FUTUROS

6.1. Conclusiones generales

En este Trabajo Fin de Grado se ha abordado el problema de la generación automática de código ensamblador CODE-2 a partir de descripciones en lenguaje natural mediante el uso de modelos de lenguaje basados en arquitecturas Transformer. A lo largo del trabajo se ha diseñado un pipeline completo que abarca desde la generación del dataset sintético hasta el entrenamiento, optimización, evaluación y análisis crítico del modelo final.

6.1.0.1 Hallazgos principales sobre técnicas de fine-tuning

Los resultados obtenidos demuestran que los modelos de lenguaje preentrenados, como T5-Small, pueden adaptarse de forma efectiva a dominios altamente estructurados como los lenguajes ensambladores educativos, siempre que se disponga de un conjunto de datos adecuado y de una metodología de entrenamiento bien definida. En particular, el uso de un dataset sintético permitió suplir la ausencia de bases de datos reales suficientemente grandes y representativas, haciendo viable el proceso de fine-tuning.

La evaluación exhaustiva de distintas estrategias de ajuste fino reveló un panorama complejo:

- **Prompt-Tuning fracasó completamente**, con valores de entrenamiento descontrolados (Training Loss 5.4074 frente a valores esperados de 0.01-0.02). Esta técnica resultó inadecuada para la complejidad de la tarea de generación de código.
- **LoRA mostró rendimiento competitivo** (EM 0.64) pero con un gap de validación mayor (0.0259 vs 0.0090), indicando generalización menos consistente que Full Fine-Tuning. Sin embargo, LoRA representa una opción valiosa en escenarios con restricciones severas de recursos.

- **Full Fine-Tuning proporciona la mejor calidad y consistencia** (EM 0.75 tras optimización, mejorando a EM 0.83 con ajuste de decodificación) especialmente en ejemplos complejos y no vistos durante el entrenamiento. Esta diferencia resultó determinante para la selección de la metodología final, dado el carácter educativo del proyecto y la importancia crítica de la corrección semántica del código generado.
- **Adapters no fue explorado en profundidad** debido a que Full Fine-Tuning demostró superioridad clara y se adapta bien a los recursos disponibles. Adapters habría requerido modificaciones arquitectónicas adicionales sin beneficio evidente.

6.1.0.2 Limitaciones de técnicas alternativas de compresión

Durante la investigación se identificaron limitaciones críticas de técnicas que no fueron seleccionadas:

- **Quantization-Aware Training (QAT) no fue viable:** QAT presenta incompatibilidades fundamentales con modelos Transformer complejos como T5 en el entorno de Windows con PyTorch 2.6. La herramienta FX-QAT de PyTorch falla al analizar el grafo de cálculo interno del modelo, haciendo imposible la implementación de esta técnica. Esta es una limitación de infraestructura, no de implementación defectuosa.
- **Poda agresiva resulta contraproducente:** Los experimentos demostraron que reducciones superiores al 10 % de parámetros causan colapso completo del modelo (EM=0.00). La generación secuencial de código es más sensible a la poda que tareas de clasificación, requiriendo precisión en las capas de atención.

6.1.0.3 Optimización de hiperparámetros e integración de técnicas de cuantización

La optimización de hiperparámetros desempeñó un papel clave en la mejora del rendimiento del modelo. El uso combinado de pruebas manuales y herramientas de optimización automática (Optuna) permitió identificar configuraciones más estables y con mejor capacidad de generalización, adaptadas a las limitaciones del hardware disponible.

El análisis del impacto de la cuantización puso de manifiesto que es posible mejorar significativamente la eficiencia del modelo mediante técnicas selectivas:

- **ONNX INT8 representa el mejor compromiso** entre calidad y eficiencia (EM 0.79, speedup 1.5x), siendo viable para despliegue en CPU en entornos educativos con recursos limitados.
- **Poda moderada al 10 %** incluso mejora ligeramente la calidad (EM 0.83) posiblemente por efecto de regularización, permitiendo combinarse con cuantización para máxima compresión.
- **Distillation mantiene calidad aceptable** (EM 0.79) pero no ofrece ventajas claras en inferencia, siendo más útil para transferencia de conocimiento en múltiples adaptaciones.

6.1.0.4 Conclusión integral

En conjunto, este trabajo confirma la viabilidad del uso de modelos de lenguaje para la generación de código ensamblador en contextos educativos, habiendo identificado y documentado tanto técnicas viables (Full Fine-Tuning + ONNX INT8) como limitaciones técnicas fundamentales (QAT en Windows, poda agresiva en modelos generativos). La metodología final elegida aporta una base sólida tanto desde el punto de vista técnico como metodológico, equilibrando calidad de generación con viabilidad práctica de despliegue.

6.2. Recomendaciones

Con base en los resultados obtenidos, se proponen las siguientes recomendaciones para el uso práctico del modelo:

- **Contextos educativos:** Utilizar Full Fine-Tuning sin cuantización cuando se requiera máxima calidad y corrección semántica, especialmente en ejemplos complejos.
- **Entornos con recursos limitados:** Emplear el modelo ONNX INT8 como alternativa viable, acompañándolo de mecanismos de validación del código generado.
- **Optimización de hiperparámetros:** Realizar siempre ajuste específico al hardware disponible, evitando configuraciones genéricas que conducen a sobreajuste.
- **Uso como herramienta de asistencia:** Emplear el modelo como apoyo al aprendizaje, no como sustituto del razonamiento humano, fomentando el análisis crítico del código generado.

6.3. Propuestas de mejora

A partir de las limitaciones identificadas, se proponen las siguientes líneas de trabajo futuro:

- **Ampliación del repertorio de instrucciones:** Incorporar un conjunto más completo de CODE-2 o adaptar el sistema a otros lenguajes ensambladores educativos, evaluando la escalabilidad del modelo.
- **Evaluación funcional mediante emulador:** Complementar las métricas de similitud textual con validación de ejecución real del código generado.

- **Exploración de arquitecturas más recientes:** Analizar modelos de mayor tamaño o técnicas avanzadas de cuantización para mejorar el equilibrio calidad-eficiencia.
- **Destilación combinada con cuantización:** Explorar la aplicación secuencial de destilación de conocimiento seguida de cuantización INT8, aprovechando las ventajas de reducción de tamaño del modelo estudiante con la eficiencia de inferencia de la cuantización.
- **Integración en herramientas educativas:** Desarrollar interfaces interactivas que permitan a los estudiantes generar, analizar y depurar código con asistencia del modelo.
- **Aprendizaje continuo:** Incorporar mecanismos de adaptación incremental para mejorar progresivamente a partir de nuevas interacciones.

BIBLIOGRAFÍA

[1] A. Abid, A. Abdalla, A. Abid, D. Khan, A. Alfozan y J. Zou, «Gradio: Hassle-Free Sharing and Testing of ML Models in the Wild,» *arXiv preprint arXiv:1906.02569*, 2019. DOI: 10.48550/arXiv.1906.02569 dirección: https://arxiv.org/abs/1906.02569

[2] T. Akiba, S. Sano, T. Yanase, T. Ohta y M. Koyama, «Optuna: A Next-generation Hyperparameter Optimization Framework,» en *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, págs. 2623-2631. DOI: 10.1145/3292500.3330701 dirección: https://dl.acm.org/doi/10.1145/3292500.3330701

[3] N. Ding, Y. Qin, G. Yang, F. Wei, Z. Yang, Y. Su, S. Hu, Y. Chen, C.-M. Chan, W. Chen et al., «Parameter-Efficient Fine-Tuning of Large Models: A Survey,» *arXiv preprint arXiv:2303.15647*, 2023. DOI: 10.48550/arXiv.2303.15647 dirección: https://arxiv.org/abs/2303.15647

[4] G. E. Hinton, O. Vinyals y J. Dean, «Distilling the Knowledge in a Neural Network,» *arXiv preprint arXiv:1503.02531*, 2015. DOI: 10.48550/arXiv.1503.02531 dirección: https://arxiv.org/abs/1503.02531

[5] N. Houlsby, A. Giurgiu, S. Jastrzebski, B. Morrone, Q. de Laroussilhe, A. Gesmundo, M. Attariyan y S. Gelly, «Parameter-Efficient Transfer Learning for NLP,» *Proceedings of the 36th International Conference on Machine Learning*, págs. 2790-2799, 2019. DOI: 10.48550/arXiv.1902.00751 dirección: https://arxiv.org/abs/1902.00751

[6] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang y W. Chen, «LoRA: Low-Rank Adaptation of Large Language Models,» *International Conference on Learning Representations (ICLR)*, 2022. DOI: 10.48550/arXiv.2106.09685 dirección: https://arxiv.org/abs/2106.09685

[7] Hugging Face, *Hugging Face Transformers Documentation*, https://huggingface.co/docs/transformers, Recuperado en 2024, 2023.

[8] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam y D. Kalenichenko, «Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference,» *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, págs. 2704-2713, 2018. DOI: 10.1109/CVPR.2018.00286 dirección: https://arxiv.org/abs/1712.05877

[9] T. Kudo, *SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing*, https://github.com/google/sentencepiece, Recuperado en 2024, 2018.

[10] B. Lester, R. Al-Rfou y N. Constant, «The Power of Scale for Parameter-Efficient Prompt Tuning,» *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, págs. 3045-3059, 2021. DOI: 10.18653/v1/2021.emnlp-main.243 dirección: https://aclanthology.org/2021.emnlp-main.243

[11] C.-Y. Lin, «ROUGE: A Package for Automatic Evaluation of Summaries,» en *Text Summarization Branches Out*, Association for Computational Linguistics, 2004, págs. 74-81. dirección: https://aclanthology.org/W04-1013

[12] Z. Liu et al., «A Survey on Code Generation with Large Language Models,» *ACM Computing Surveys*, 2023. DOI: 10.1145/3617680 dirección: https://dl.acm.org/doi/10.1145/3617680

[13] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen y T. Blankevoort, «A White Paper on Neural Network Quantization,» *arXiv preprint arXiv:2106.08295*, 2021. DOI: 10.48550/arXiv.2106.08295 dirección: https://arxiv.org/abs/2106.08295

[14] ONNX Community, *ONNX: Open Neural Network Exchange*, https://onnx.ai, Estándar abierto para interoperabilidad de modelos de aprendizaje automático. Recuperado en 2024, 2022.

[15] K. Papineni, S. Roukos, T. Ward y W.-J. Zhu, «BLEU: A Method for Automatic Evaluation of Machine Translation,» en *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, 2002, págs. 311-318. DOI: 10.3115/1073083.1073135 dirección: https://aclanthology.org/P02-1040

[16] A. Prieto, A. Lloris, J. C. Torres y M. de la Asunción, «CODE-2: A Didactic Tool for Computer Architecture Education,» en *Actas de las X Jornadas de Enseñanza Universitaria de la Informática (JENUI)*, Repositorio Institucional RUA, Universidad de Alicante, Universidad de Alicante, 2002. dirección: https://rua.ua.es/dspace/handle/10045/128010

[17] PyTorch, *PyTorch Documentation*, https://pytorch.org/docs/stable/index.html, Recuperado en 2024, 2023.

[18] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li y P. J. Liu, «Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer,» *Journal of Machine Learning Research*, vol. 21, n.º 140, págs. 1-67, 2020. dirección: https://jmlr.org/papers/v21/20-1168.html

[19] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo y D. Dennison, «Hidden Technical Debt in Machine Learning Systems,» *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 28, 2015. dirección: https://papers.nips.cc/paper/2015/hash/86df7dcfd896fcaf2674f757a2463eba-Abstract.html

[20] N. Shazeer y M. Stern, «Adafactor: Adaptive Learning Rates with Sublinear Memory Cost,» en *Proceedings of the 35th International Conference on Ma-*

chine Learning, 2018, págs. 4596-4604. DOI: 10.48550/arXiv.1805.09843 dirección: https://arxiv.org/abs/1805.09843

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser e I. Polosukhin, «Attention Is All You Need,» *Advances in Neural Information Processing Systems*, vol. 30, 2017. DOI: 10.48550/arXiv.1706.03762 dirección: https://arxiv.org/abs/1706.03762

[22] O. Zawacki-Richter, V. I. Marín, M. Bond y F. Gouverneur, «Systematic Review of Research on Artificial Intelligence Applications in Higher Education – Challenges and Opportunities,» *International Journal of Educational Technology in Higher Education*, vol. 18, n.º 1, págs. 1-42, 2021. DOI: 10.1186/s41239-021-00301-x dirección: https://doi.org/10.1186/s41239-021-00301-x

ANEXOS

A. Enlaces y recursos del despliegue

En este anexo se proporcionan los enlaces y recursos relacionados con el despliegue del modelo desarrollado:

- **Modelo en Hugging Face:** `https://huggingface.co/spaces/coyoteMMK/Code2_AI`
- **Interfaz web en GitHub Pages:** `https://coyotemmk.github.io/Code2_AI/`
- **Repositorio público:** `https://github.com/coyoteMMK/Code2_AI`

B. Estructura del repositorio

El repositorio público disponible en GitHub contiene todos los recursos necesarios para reproducir los experimentos realizados en este trabajo. La estructura del repositorio es la siguiente:

```
Code2_AI/
|-- datasource/
|   |-- test.json          # Conjunto de test (10%)
|   |-- train.json         # Conjunto de entrenamiento (70%)
|   +-- valid.json         # Conjunto de validacion (20%)
|-- models/
|   |-- full_fp32/         # Modelo Full Fine-Tuning FP32
|   +-- onnx_int8_dynamic/ # Modelo cuantizado ONNX INT8
|-- scripts/
|   |-- code2_cli_multi.py          # Interfaz CLI interactiva
|   |-- data_generation.py          # Generacion de dataset sintetico
|   |-- evaluate.py                 # Evaluacion de modelos
|   |-- quantize_onnx_int8_dynamic.py # Cuantizacion ONNX
|   |-- test_env.py                 # Verificacion de entorno
|   +-- training.py                 # Script de entrenamiento
|-- site/                  # Interfaz web (Next.js)
|-- README.md              # Documentacion principal
|-- requirements.txt       # Dependencias generales
+-- requirements-torch.txt # Dependencias especificas PyTorch
```

.0.1 Descripción de componentes principales

- **datasource/**: Contiene los datasets en formato JSON utilizados para entrenamiento, validación y testeo del modelo.
- **models/**: Almacena los modelos entrenados en sus diferentes versiones (FP32 original y ONNX INT8 cuantizado).

- **scripts/training.py**: Script de entrenamiento que implementa el pipeline completo de fine-tuning, incluyendo carga de datos, configuración de hiperparámetros y guardado de modelos.
- **scripts/code2_cli_multi.py**: Interfaz de línea de comandos interactiva que permite seleccionar entre diferentes versiones del modelo y realizar inferencia local.
- **scripts/data_generation.py**: Generador del dataset sintético que crea ejemplos válidos y con errores según las reglas de CODE-2.
- **scripts/evaluate.py**: Sistema de evaluación que calcula métricas Exact Match, BLEU y ROUGE-L sobre los conjuntos de test.
- **scripts/quantize_onnx_int8_dynamic.py**: Implementación de la cuantización ONNX INT8 dinámica aplicada al modelo final.
- **site/**: Aplicación web desarrollada con Next.js y Tailwind CSS, desplegada en GitHub Pages.