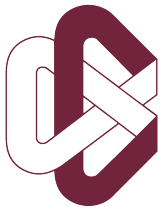


CIMAT

Centro de Investigación en Matemáticas
Unidad Monterrey

Análisis de Texto e Imágenes

Generación y Clasificación de Texto con Deep Learning

Gustavo Hernández Angeles

12 de octubre de 2025

Índice

1 Parte A: Generación de Texto

1.1 Datos

En esta sección se explica el proceso de recopilación y limpieza de datos para entrenar los modelos de generación de texto utilizando letras de canciones.

1.1.1 Recopilación de datos

Se utilizaron dos scripts de Python para scrapear las letras de las canciones de un artista específico desde la plataforma *Genius*. El primer script (`scrape_songs_alpaca.py`) realiza las siguientes tareas:

- Autenticación en la API de Genius.
- Búsqueda de canciones (URLs) del artista.
- Extracción de las letras de las canciones.
- Limpieza y preprocesamiento de las letras en formato Alpaca para Fine-Tuning.
- Almacenamiento de las letras en un archivo de texto.

El segundo script (`preprocess_scraped_songs.py`) se encarga de procesar la salida del primer script para generar un archivo .txt con el formato adecuado para el entrenamiento de los modelos RNN/LSTM/GRU.

La selección de artistas y el número de canciones a scrapear se encuentran configuradas directamente en el código del primer script. En este caso, elegí a *Kendrick Lamar*, *Kanye West* y *Jay-Z* debido al género compartido *Hip-Hop/Rap* y la riqueza lírica de sus canciones. El número de canciones por artista se estableció en 100, lo que da un total aproximado de 300 canciones.

La ejecución de ambos scripts se realiza desde la terminal con la serie de comandos:

```
python ./codigo/generative/scraping/scrape_songs_alpaca.py
python ./codigo/generative/scraping/preprocess_scraped_songs.py
```

Nota: Es importante asegurarse de tener la clave de API de Genius configurada en un archivo `.env` en el mismo directorio que el script. Ejemplo: `GENIUS_API_TOKEN="tu_token_aqui"`.

Al finalizar la ejecución, obtenemos distintos archivos para ambos formatos de salida particionados en conjuntos de entrenamiento y prueba. Además, también se generan los diccionarios necesarios para el preprocesamiento de texto en los modelos RNN/LSTM/GRU, y estadísticas del conjunto de datos. Los archivos generados son almacenados en el directorio `./data/text_gen/` y son los siguientes:

- `train_lyrics_alpaca.json` y
- `test_lyrics_alpaca.json` (formato Alpaca).
- `train_lyrics.txt` y
- `test_lyrics.txt` (formato para RNN/LSTM/GRU).
- `vocab_char.json` y

- `vocab_word.json` (diccionarios de caracteres y palabras).
- `dataset_info.json` (estadísticas del conjunto de datos).

1.1.2 Limpieza y preprocesamiento del texto

Genius provee las letras en formato HTML dentro de `lyrics containers`, los cuales son etiquetas con el atributo `data-lyrics-container="true"`. El script extrae el texto de estas etiquetas y realiza las siguientes operaciones de limpieza:

- Eliminación de etiquetas HTML.
- Eliminación de líneas en blanco y espacios innecesarios.
- Crea cada registro en formato Alpaca para el fine-tuning.

He decidido no eliminar las anotaciones de las canciones (como [Coro], [Verso 1], etc.) ya que pueden proporcionar contexto adicional al modelo durante el entrenamiento, además de que pueden ser útiles para la generación de texto (haciendo que el modelo también genere anotaciones). Además, se conservaron los metadatos como nombre de la canción y artista, ya que pueden ser útiles para futuras referencias o análisis.

Al realizar Fine-Tuning sobre el LLM, es conveniente proporcionar contexto adicional (y acorde) al modelo. Debido a que en este caso nos decidimos por utilizar un modelo instructivo, el formato Alpaca es adecuado para este propósito. Este formato contiene campos para la instrucción, entrada y salida, lo que permite al modelo aprender a generar texto basado en instrucciones específicas de un solo turno.

```
{
    "instruction": "<INSTRUCCIÓN>",
    "input": "<ENTRADA>",
    "output": "<SALIDA>"
}
```

donde:

- `<INSTRUCCIÓN>` es una cadena que indica la tarea a realizar, en este caso: “You are a hip-hop artist, helping people write lyrics.” o variantes.
- `<ENTRADA>` es una cadena que proporciona contexto adicional, en este caso: “Help me write a song in the style of `<ARTISTA>`.”
- `<SALIDA>` es la letra de la canción.

Por otro lado, al utilizar modelos RNN/LSTM/GRU basta con simplemente tener las letras en texto plano, ya que estos modelos no requieren el mismo nivel de contexto que los LLMs. Sin embargo, es importante asegurarse de que el texto esté limpio y bien formateado para evitar problemas durante el entrenamiento. El script de preprocesamiento convierte los textos de formato Alpaca a texto plano, asegurándose de que cada letra esté separada por los delimitadores de cada canción (En este caso `<|song_start|>` y `<song_end>`).

1.2 Transformers

1.2.1 Mistral 7B Instruct v0.3

En este trabajo se utilizó el modelo `unsloth/mistral-7B-instruct-v0.3-bnb-4bit` como base para el fine-tuning. Este modelo es una variante del modelo creado por Mistral: Mistral-7B [1], y optimizado por *Unsloth* para tareas de instrucción y cuantizado a 4 bits para reducir el tamaño del modelo.

El framework utilizado para el fine-tuning fue `transformers` de Hugging Face, junto con `peft` para la implementación de Low-Rank Adaptation (LoRA). La elección de LoRA se debe a su eficiencia en términos de memoria y tiempo de entrenamiento, permitiendo adaptar grandes modelos preentrenados con un costo computacional reducido [2].

1.2.2 Configuración del Fine-Tuning

Tokenizador y representación. El modelo base emplea un tokenizador sub-palabras basado en SentencePiece con un vocabulario de aproximadamente 32K tokens [3]. Este enfoque permite manejar adecuadamente la variabilidad léxica propia de letras de rap (slang, contracciones y variaciones estilísticas) sin inflar excesivamente el vocabulario. No se realizó entrenamiento de un tokenizador nuevo, ya que el objetivo fue un ajuste fino instructivo y no la adaptación desde cero a un dominio léxico extremadamente distinto.

Formato de datos. Cada ejemplo del conjunto de entrenamiento se construyó en formato tipo *Alpaca*: `(instruction, input, output)` y posteriormente convertido a un *chat template* estilo `alpaca` mediante `tokenizer.apply_chat_template`. Para cada canción se genera típicamente un par instrucción–respuesta, donde la instrucción solicita la generación de letras en el estilo de un artista específico y la salida contiene la letra correspondiente. Tras el mapeo, el campo final consumido por el entrenador es un único texto ya formateado.

Tamaño del conjunto. El corpus curado contiene 239 ejemplos de entrenamiento y 60 de prueba (correspondientes a canciones únicas). Con un tamaño de lote efectivo de 16 secuencias (lote por dispositivo = 2, acumulación de gradientes = 8), el número de pasos por época es:

$$\text{steps/epoch} = \lceil 239/16 \rceil = 15.$$

Se entrenaron 3 épocas, resultando en 45 pasos totales (se almacenaron checkpoints intermedios consistentes con este valor). Esto refleja un escenario de ajuste fino ligero, más orientado a estilo que a aprendizaje exhaustivo.

Longitud de secuencia. Se fijó `max_seq_length = 2048`. Dado que las letras individuales rara vez exceden este límite, se deshabilitó el *packing* (`packing=False`) para evitar mezclar canciones dentro de la misma secuencia y preservar coherencia estructural (secciones como [Verse], [Chorus], etc.).

Configuración LoRA. Se aplicó LoRA con `r = 32`, `lora_alpha = 64`, `lora_dropout = 0.0` sobre los módulos de proyección de atención y MLP: `q_proj, k_proj, v_proj, o_proj, gate_proj, up_proj, down_proj`. Suponiendo dimensiones del modelo base (d_model = 4096, tamaño intermedio = 14336, 32 capas), el número aproximado de parámetros entrenables añadidos por LoRA es:

$$\text{por capa} = 4 \times r(4096+4096) + 3 \times r(4096+14336) = 4(32 \cdot 8192) + 3(32 \cdot 18432) \approx 2{,}818{,}048,$$

$$\text{total} \approx 32 \times 2{,}818{,}048 \approx 90.2\text{M},$$

lo que representa 1.2 % de los =7.3B parámetros totales, pero concentra toda la capacidad de adaptación. Estos pesos LoRA se mantienen en precisión media (FP16/BF16) mientras los pesos base permanecen cuantizados a 4 bits [4].

Hiperparámetros de entrenamiento. Se utilizaron: tasa de aprendizaje `2e-4` (scheduler lineal con calentamiento de 5 pasos), `epochs = 3`, `per_device_train_batch_size = 2`, `gradient_accumulation_steps = 8`, `weight_decay = 0.01`, optimizador `AdamW 8-bit` [5], y semilla `3407`. El tamaño de lote efectivo resultante es 16 ejemplos. No se aplicó regularización adicional (`lora_dropout = 0`).

Gestión de memoria y hardware. El uso combinado de cuantización 4-bit + LoRA reduce el pico de VRAM necesario a un rango que típicamente cabe en una sola GPU de 12–16 GB manteniendo la ventana de contexto de 2048. El script registra memoria reservada y máxima por dispositivo antes y después del entrenamiento.

Hardware utilizado. El entrenamiento se realizó en un nodo con 2 GPUs NVIDIA RTX TITAN (24 GB VRAM cada una), 128 GB RAM y 24 núcleos de CPU Intel Xeon Silver 4214 (2.2 GHz). Equipo proporcionado por el Laboratorio de Supercómputo del Bajío (Lab-SB) del CIMAT.

1.2.3 Evaluación de la Generación de Texto

Perplejidad. La *perplejidad* (PPL) es una métrica estándar para evaluar modelos de lenguaje, que mide qué tan bien un modelo predice una muestra. Matemáticamente, la perplejidad se define como la exponencial de la entropía cruzada promedio por token:

$$\text{PPL} = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log P(w_i|w_{<i})\right),$$

donde N es el número total de tokens en el conjunto de evaluación, y $P(w_i|w_{<i})$ es la probabilidad condicional del token w_i dado el contexto previo $w_{<i}$. Una perplejidad más baja indica un mejor rendimiento del modelo, ya que sugiere que el modelo está menos "sorprendido"por los datos de evaluación.

Para evaluar el impacto del fine-tuning, se comparó la perplejidad del modelo ajustado con la del modelo base `mistral-7B-instruct-v0.3-bnb-4bit` sin ajuste fino. Esto permite cuantificar la mejora en la capacidad predictiva del modelo tras el entrenamiento con las letras de canciones. El código para calcular la perplejidad es proporcionado por la misma biblioteca `Unsloth`, se encuentra en `codigo/generative/mistral/utils/perplexity_unsloth.py`. La lógica fue adaptada en el script `calculate_perplexity.py` para evaluar ambos modelos (base y fine-tuned) sobre el conjunto de prueba.

El resultado de la evaluación de perplejidad en el conjunto de prueba es el siguiente:

- Perplejidad del modelo base (sin fine-tuning): 9.35.
- Perplejidad del modelo fine-tuned: 7.20.
- Mejora relativa: $(9.35 - 7.20)/9.35 \times 100\,\% \approx 22.93\,\%$.
- Interpretación: La reducción significativa en la perplejidad indica que el modelo fine-tuned ha mejorado su capacidad para predecir las letras de canciones, adaptándose

mejor al estilo y vocabulario específico del dominio. Esto fue logrado a pesar del tamaño relativamente pequeño del conjunto de datos de entrenamiento, demostrando la eficacia del enfoque de fine-tuning con LoRA en un modelo cuantizado.

Muestras generadas.

Se generaron varias muestras de texto utilizando el modelo fine-tuned para evaluar su capacidad de generación. Todas las muestras se generaron con los siguientes parámetros en la inferencia, por mostrar un mejor balance entre coherencia y diversidad:

- `max_new_tokens = 256`. Representa la longitud máxima de la secuencia generada.
- `temperature = 0.8` (0.7 para Kendrick Lamar). Controla la aleatoriedad de la generación; valores más bajos hacen que el modelo sea más conservador.
- `top_p = 0.9`. Utiliza muestreo de núcleo para limitar la selección de tokens a los más probables.
- `repetition_penalty = 1.3`. Penaliza la repetición de tokens para fomentar la diversidad en la generación.

A continuación se presentan algunos ejemplos representativos:

Ejemplo 1: Letra generada en el estilo de Jay-Z

En este ejemplo, se le pidió al modelo que generara una letra en el estilo de Jay-Z mediante el prompt:

```
You're a hip-hop artist, create song lyrics.
Write hip-hop lyrics in the style of Jay-Z.
```

Y se obtuvo la siguiente salida:

```
[Intro]
Uh huh uh (yeah)
Yeah yeah you know my name is Hov', right? Uhh
I got this little thing called ``The Blueprint''
And I just want to run through some shit for y'all tonight
Just have fun with it
(Hands up!)

[Verse 1]
Yo, yo, yo, yo, whattup! It's your boy Shawn
Back on that bullshit 'cause niggas ain't learn
Last time out I burned down the barn
Now I'm back and I might burn down another one
What do you wanna bet? You can put all ya chips in
If he don't win then we all gonna be broke again
But if he does win then we gon' get richer than rich men
So place your bets now before they stop taking them
See me as an investment like Berkley or Microsoft
Put ten dollars into me baby watch how far it goes
Watch how fast it grows
```

Podemos analizar la letra generada entre pilares del estilo de cada artista:

- **Persona y Autenticidad.** El intro es muy representativo del estilo de Jay-Z, utilizando *ad-libs* (Uh, uh, yeah) y directamente referenciando su nickname (Hov'). También hace referencia a su álbum "The Blueprint", lo que refuerza su identidad como artista. El uso de "Your boy Shawn" es también auténtico.
- **Temas y Contenido.** La letra realiza una metáfora sobre invertir en sí mismo como un artista, lo cual es un tema recurrente en la música de Jay-Z.
- **Rimas y Fluidez.** La letra presenta un esquema de rima consistente (AABB), aunque la estructura es predecible y carece de complejidad y expertiz de Jay-Z.

Ejemplo 2: Letra generada en el estilo de Kanye West

Nuevamente, se le pidió al modelo la generación de una letra, ahora en el estilo de Kanye West. Esto fue mediante el prompt:

```
You're a hip-hop artist, create song lyrics.
Write hip-hop lyrics in the style of Kanye West.
```

Y se obtuvo la siguiente salida:

```
[Intro]
I know He won't forsake me
He won't leave me
No matter what I do
And even if I go through hell tonight
He will still be God when it's all over with
(God is) (God is)

[Verse 1]
They say build your own
I said fuck that, we gon' buy our way out
We ain't finna wait 'til they give us something to celebrate
So let's get this bread
Fuck up school and just drop out
But you can't drop out without dropping some knowledge first
Let me teach y'all how to make yourself rich
How to stay fly while staying righteous
First thing you need is faith as big as Mount Rushmore
Then take it from here

[Chorus]
Jesus saved my soul
Now I drive Rollies and Bentleys
Bought myself a couple chains
Spendin' money like I don't owe Him nothin'
Everybody want heaven but nobody wanna die
Nobody wanna cry no more
```

```
That means everybody wanna live forever
Not ready to meet Your maker
```

- **Persona y Autenticidad.** La letra captura la dualidad central de la persona de Kanye West: la del predicador y el materialista. El intro con temática góspel y las referencias a Dios son muy representativas de su era en los finales de los 10's con el album *Jesus is King*. Esto se contrasta inmediatamente con un tono provocador y egocéntrico (Let me teach y'all", fuck that"), lo cual es sumamente auténtico a su estilo.
- **Temas y Contenido.** El modelo representa exitosamente la mezcla de temas de Kanye. Se abordan la religión y la fe ("Jesus saved my soul", "faith as big as Mount Rushmore") en conflicto directo con el materialismo extremo y el lujo ("drive Rollies and Bentleys"). Incluye también aforismos y comentarios sociales simplificados, como la línea "Everybody want heaven but nobody wanna die", un recurso muy común en sus letras.
- **Rimas y Fluidez.** Kanye a menudo prioriza el mensaje y la emoción sobre la complejidad de la rima. La letra generada refleja esto: el esquema de rima es inconsistente y conversacional. Se apoya más en la cadencia y en la fuerza de las declaraciones que en patrones de rima complejos, lo que resulta en un flujo que se asemeja a un sermón o una declaración apasionada, muy fiel a su estilo de entrega.

Ejemplo 3: Letra generada en el estilo de Kendrick Lamar

Esta vez se le pidió al modelo generar una letra para una canción de Hip-Hop al estilo de Kendrick Lamar, mediante el prompt:

```
You're a hip-hop artist, create song lyrics.
Write hip-hop lyrics in the style of Kendrick Lamar.
```

Y se obtuvo la siguiente salida:

```
[Intro]
I don't know what they want from me! I gave them my life
They wanna see some type of drama? Here it is
(Bitch)

[Verse 1]
What up homie? What you writing about today? (Today?)
Writing 'bout these niggas that be hating on me (Huh?)
And all this fake love that surrounds me (Uh)
All this bullshit that goes around me (Yeah)
It gets tiring sometimes when you trying to stay positive
But negative energy always surrounding you
So how can we live with no stress and anxiety?
When everybody got something bad to say bout you
Everybody wants something but ain't willing to give nothin' back
Nobody really cares if you win or lose at shit
If your name isn't mentioned then nobody gon' mention you
Even though you did more than most people ever done
That'll make you feel like you worthless
```

```
Like you never won
Never been number one
Always second place even when you first
How do you deal with that kind of pressure? How do you cope?
Knowin' every move you make will affect your whole family tree
Your friends, your enemies, your lovers, haters, neighbors
The world, society, everyone expect perfection
From an imperfect person
```

- **Persona y Autenticidad.** Captura a la perfección la persona introspectiva, ansiosa y abrumada que define gran parte de la obra reciente de Kendrick Lamar (DAMN y MM&BS). La letra se lee como un monólogo interno o una sesión de terapia, explorando la presión de la fama de una manera vulnerable y conflictiva, lo cual es una característica central de su estilo.
- **Temas y Contenido.** Los temas abordados son extremadamente fieles al universo lírico de Kendrick. Se centra en la salud mental ("stress and anxiety"), el peso de las expectativas ("everyone expect perfection"), la falsedad en la industria ("fake love that surrounds me"), y la responsabilidad generacional ("every move you make will affect your whole family tree"). Este enfoque en el conflicto interno y la carga del liderazgo es mucho más representativo que la simple bravuconería.
- **Rimas y Fluidez.** El modelo imita exitosamente la fluidez conversacional y casi de "spoken word"de Kendrick. En lugar de adherirse a un esquema de rima estricto, prioriza el flujo de conciencia y la carga emocional del mensaje. Las líneas son largas y discursivas, construyendo una tensión que culmina en la lista final ("Your friends, your enemies..."). Esta estructura, que favorece el ritmo narrativo sobre la rima perfecta, es una técnica clave en el repertorio de Kendrick.

En los tres ejemplos, el modelo fine-tuned demuestra una capacidad notable para capturar los estilos únicos de cada artista, tanto en términos de contenido temático como la estructura lírica. Aunque no alcanza la complejidad y profundidad de los artistas originales, especialmente en términos de rima y metáforas, el modelo genera letras coherentes y estilísticamente apropiadas para los artistas. Además, el modelo respetó en su mayoría la estructura de las letras basandose en los datos de entrenamiento (por ejemplo, siempre inicio con el [Intro], [Verse 1], etc.). Esto sugiere que el enfoque de fine-tuning con un conjunto de datos relativamente pequeño pero bien curado puede ser efectivo para adaptar un modelo de lenguaje grande a tareas creativas específicas como la generación de letras de canciones.

1.3 Modelos RNN/LSTM/GRU

1.3.1 RNN Simple

Las RNN simples constituyen la forma canónica de modelar dependencias secuenciales mediante un estado oculto recurrente. Cada paso de tiempo aplica una transformación no lineal sobre la combinación del embedding del token actual y el estado oculto previo. Sin embargo, su profundidad temporal efectiva se ve limitada por el *desvanecimiento* y la *explosión* del gradiente, lo cual dificulta capturar dependencias de largo alcance en secuencias lingüísticas (estructuras repetitivas, coherencia temática, progresión narrativa, etc.).

Tokenización y representación. Se entrenaron dos variantes: a nivel carácter y a nivel palabra. El vocabulario carácter contiene 157 símbolos (incluyendo saltos de línea, puntuación y caracteres Unicode presentes en las letras). El vocabulario palabra contiene 19,107 tokens (palabras, formas con apóstrofos y algunos préstamos multilingües). Cada token se proyecta a un embedding de dimensión 128.

Formato de datos. El corpus en texto plano se segmenta en ventanas deslizantes de longitud fija `seq_length = 100`. Para cada ventana de 100 tokens (o caracteres) se predice el siguiente token. No se mezcla información entre canciones más allá de la concatenación lineal del corpus, pero se preservan delimitadores estructurales (`<|song_start|>` / `<song_end>`), lo que provee señales débiles de segmentación.

Tamaño del conjunto. El conjunto de entrenamiento contiene 820,993 caracteres (nivel de carácter) y 157,910 palabras (nivel de palabra). El conjunto de prueba contiene 214,957 caracteres y 41,889 palabras, respectivamente. El número de secuencias efectivas por nivel se calculan como (longitud−100). Con `batch_size = 64`, esto produce aproximadamente 12.8K batches por época (carácter) y 2.5K batches por época (palabra), lo que impacta la relación señal–ruido de la estimación del gradiente (más estable en la variante palabra, más estocástica en la variante carácter).

Arquitectura. La celda recurrente implementa la actualización clásica:

$$h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b), \quad y_t = W_{hy}h_t + c,$$

donde $x_t \in \mathbb{R}^{128}$, $h_t \in \mathbb{R}^{256}$. Se emplean 2 capas apiladas (`num_layers = 2`) con `dropout = 0.3` entre capas (no entre pasos temporales internos de cada capa).

Capacidad y parámetros. Aproximadamente:

- Variante palabra: $\sim$ 7.59M parámetros entrenables (embeddings ≈2.45M, núcleo recurrente ≈0.23M, proyección de salida ≈4.91M).
- Variante carácter: $\sim$ 0.29M parámetros (la reducción proviene del vocabulario compacto).

Esto ilustra la fuerte asimetría de capacidad inducida por el tamaño del vocabulario y la capa de salida softmax.

Hiperparámetros de entrenamiento. Se usó el optimizador Adam (`lr = 2e-3`), 20 épocas, `batch_size = 64`, `seq_length = 100`, `gradient clipping = 5.0`. La función de pérdida es la entropía cruzada. Un scheduler `ReduceLROnPlateau` reduce la tasa de aprendizaje con factor 0.5 tras 2 épocas sin mejora significativa (`patience = 2`). La pérdida se reporta en escala log-perplexity y puede re–expresarse como PPL $= e^{\mathcal{L}}$. Se habilitó opción de *early stopping* configurable.

Regularización y estabilidad. El *dropout* inter–capas y el *gradient clipping* controlan, respectivamente, sobre–ajuste y explosión de gradientes. Aún así, las limitaciones estructurales (sin mecanismos de compuertas ni memoria explícita) reducen su capacidad para retener contexto semántico más allá de decenas de pasos.

Hardware utilizado. El mismo entorno descrito en la subsección de Transformers: 2× NVIDIA RTX TITAN (24 GB VRAM), 128 GB RAM y 24 núcleos CPU Intel Xeon Silver 4214, utilizando `PyTorch` para la implementación y entrenamiento.

1.3.2 LSTM

Las LSTM introducen compuertas que mitigan el desvanecimiento del gradiente y permiten retener información relevante a través de pasos temporales largos. La celda mantiene un estado de memoria c_t separado del estado oculto h_t, modulando flujos de información mediante puertas sigmoides.

Tokenización y representación. Se emplean las mismas dos granularidades (carácter y palabra) con embeddings de dimensión 128 inicializados aleatoriamente y aprendidos de forma conjunta. No se reutilizó el tokenizador sub-palabras del modelo Transformer dado que aquí el foco es analizar arquitecturas recurrentes "clásicas".

Formato de datos. Idéntico al de la RNN simple: ventanas deslizantes de longitud 100, objetivo el token siguiente. La preservación de delimitadores de canción actúa como débil frontera semántica, y se espera que la LSTM capture patrones de estructura (introducciones, coros) con mayor robustez que la RNN simple.

Tamaño del conjunto. Igual a la variante anterior; esto permite comparaciones controladas de arquitectura.

Arquitectura. Dos capas LSTM (`hidden_dim = 256`, `num_layers = 2`) con `dropout = 0.3` entre capas. Ecuaciones de la celda (omitiendo subíndices de capa):

$$
\begin{aligned}
i_t &= \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i),\\
f_t &= \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f),\\
g_t &= \tanh(W_{xg}x_t + W_{hg}h_{t-1} + b_g),\\
o_t &= \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o),\\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t,\\
h_t &= o_t \odot \tanh(c_t).
\end{aligned}
$$

Capacidad y parámetros. Aprox.:

- Variante palabra: $\sim$ 8.28M parámetros (el incremento sobre RNN simple proviene del factor de 4 en matrices de compuertas).
- Variante carácter: $\sim$ 0.98M parámetros.

El salto de capacidad frente a la RNN simple se concentra en las matrices de compuertas, incrementando la expresividad temporal.

Hiperparámetros de entrenamiento. Se mantuvieron los mismos valores globales para favorecer comparabilidad: `Adam`, 20 épocas de entrenamiento, `batch_size = 64`, `seq_length = 100`, `clip = 5.0`, scheduler adaptativo sobre la pérdida de validación y potencial *early*

stopping. La métrica primaria interna es la entropía cruzada; la perplexity derivada sirve para contrastes posteriores (no reportada aquí).

Regularización y estabilidad. El mecanismo de compuertas (en particular f_t) estabiliza el flujo de gradientes y atenúa el desvanecimiento típico. El *dropout* entre capas actúa como regularizador estructural, y el *gradient clipping* asegura control frente a ráfagas de magnitud.

Ventajas observadas conceptualmente. Mayor retención de contexto largo (estados temáticos, continuidad de narrador) y mejor manejo de secciones donde la progresión semántica se apoya en repeticiones moduladas (p. ej. estribillos).

1.3.3 GRU

Las GRU (Gated Recurrent Units) simplifican la estructura de la LSTM fusionando algunas compuertas y eliminando el estado de memoria separado. Mantienen capacidad para mitigar el desvanecimiento del gradiente con menor costo parametrizable, lo que puede traducirse en entrenamientos más rápidos y menor riesgo de sobre–ajuste en conjuntos moderados.

Tokenización y representación. Se reutiliza idéntica configuración (carácter y palabra, embeddings de 128). La comparación GRU vs LSTM bajo el mismo espacio de embeddings permite aislar el impacto de la diferencia arquitectónica.

Formato de datos. Igual que en las dos arquitecturas previas (ventanas de 100 tokens $\rightarrow$ predicción del siguiente). Esto garantiza que cualquier diferencia empírica futura (perplejidad, estabilidad) se atribuya al diseño de la celda y no a variaciones de entrada.

Arquitectura. Dos capas GRU de 256 unidades con `dropout = 0.3` entre capas. Las ecuaciones de la celda:

$$
\begin{aligned}
z_t &= \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z) && \text{(puerta de actualización)} \\
r_t &= \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r) && \text{(puerta de reinicio)} \\
ildeh_t &= \tanh(W_{xh}x_t + W_{hh}(r_t \odot h_{t-1}) + b_h) && \text{(candidato)} \\
h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t.
\end{aligned}
$$

Capacidad y parámetros. Aproximadamente:

- Variante palabra: $\sim$ 8.05M parámetros (intermedio entre RNN simple y LSTM; 3 grupos de matrices en lugar de 1 o 4).
- Variante carácter: $\sim$ 0.75M parámetros.

La reducción frente a LSTM puede favorecer menor sobre–ajuste en dominios con variabilidad estilística pero tamaño moderado.

Hiperparámetros de entrenamiento. Se conservaron los mismos (`Adam`, `lr = 2e-3`, `epochs = 20`, `batch_size = 64`, `seq_length = 100`, clipping 5.0, scheduler adaptativo). Esta homogeneidad facilita comparaciones directas posteriores de eficiencia y convergencia.

Regularización y estabilidad. El diseño de la puerta de actualización z_t actúa como interpolador adaptativo entre memoria previa y nuevo contenido, reduciendo la necesidad de un estado separado tipo c_t. El *dropout* inter–capas y el clipping complementan el control de generalización y estabilidad numérica.

Consideraciones comparativas. Conceptualmente, la GRU ofrece un compromiso: menos parámetros que LSTM, más expresión que la RNN simple. En escenarios con ventana de

contexto moderada (100 tokens) y un dominio con repeticiones estilísticas claras (estructuras líricas), puede captar patrones de transición (Intro → Verso, Verso → Chorus) sin incurrir en el costo completo de las LSTM.

1.3.4 Perplejidad de los Modelos Recurrentes

La evaluación de perplejidad para RNN, LSTM y GRU se realizó mediante un esquema *streaming* continuo: el conjunto de prueba completo se recorre secuencialmente en bloques de tamaño fijo (`block_size = 100`), preservando el estado oculto entre bloques y calculando la pérdida en cada desplazamiento (teacher forcing con desfase 1). Este enfoque evita dos sesgos frecuentes: (i) usar solo la última posición de cada ventana y (ii) sub-muestrear tokens con ventanas disjuntas. No se reinició el estado en delimitadores de canción, lo cual puede reducir ligeramente la pérdida al permitir continuidad artificial; la decisión se mantuvo consistente entre arquitecturas y niveles.

Resultados. Los valores agregados de PPL obtenidos mediante el esquema streaming se resumen en la Table 1.1. Cada cifra corresponde a la evaluación completa sobre el conjunto de prueba, preservando estado oculto entre bloques.

Modelo	PPL (carácter)	PPL (palabra)
RNN	6.83	2,283.42
LSTM	**4.61**	**1,046.79**
GRU	6.90	8,970.11

Cuadro 1.1: PPL de modelos recurrentes a nivel carácter y palabra.

Granularidad y escala. Las perplejidades a nivel carácter son órdenes de magnitud menores que a nivel palabra por razones estructurales: (a) el vocabulario carácter (157 símbolos) induce un espacio de decisión logarítmicamente mucho más pequeño que el vocabulario palabra (19,107 tokens); (b) la distribución de frecuencias a nivel palabra sigue una cola larga (Zipf) que incrementa la entropía empírica y dificulta estimar probabilidades fiables con un corpus moderado; (c) la perplejidad no es comparable entre niveles, pues una palabra concentra la información de varios caracteres (≈4–5 en promedio).

Comparación arquitecturas (carácter). Como se aprecia en la Table 1.1, LSTM obtiene la menor perplejidad (4.61) seguido de RNN (6.83) y GRU (6.90). La ventaja de LSTM concuerda con su mayor capacidad para retener dependencias temporales. El valor de GRU probablemente pueda descender con más épocas.

Comparación arquitecturas (palabra). La LSTM también lidera a nivel palabra (1,046.79), seguida de la RNN (2,283.42). La GRU presenta una perplejidad muy elevada (8,970.11), cercana al orden de magnitud del vocabulario, lo que sugiere sub-entrenamiento más que limitación intrínseca de la celda.

Limitaciones. La perplejidad (i) es sensible a la tokenización y no captura aspectos estilísticos (rima, coherencia, estructura lírica), (ii) puede estar ligeramente optimista al no reiniciar estados entre canciones, (iii) penaliza con fuerza tokens raros elevando cifras a nivel palabra y (iv) no mide repetición ni diversidad semántica. Se requiere complementarla con análisis cualitativo y métricas adicionales orientadas a estilo.

1.3.5 Muestras generadas por Modelos Recurrentes

En esta sección se presenta un análisis cualitativo enfocado en: (i) coherencia local y global, (ii) estructura lírica (uso de etiquetas y segmentos), (iii) repetición y degradación, (iv) rasgos estilísticos (temas, rimas aproximadas), y (v) diferencias atribuibles al nivel de tokenización. Se omite mostrar la salida completa por razones de espacio, pero se incluyen fragmentos representativos y observaciones clave.

LSTM

- **Carácter:** Muestra fragmentos con continuidad sintáctica moderada: *"I want you to the ... I said I was ... I think I was the days..."*. Aunque hay inserciones redundantes ("the"), consigue mantener sujeto implícito y verbos en secuencia plausible. La granularidad carácter permite interpolar transiciones suaves y ocasionales coincidencias fonéticas finales (pseudo-rimas débiles en terminaciones "-ing" / "-ion" cuando se generan). No se observa un bucle corto estable hasta después de varias oraciones truncadas, lo que sugiere dinámica de estado más rica.
- **Palabra:** Presenta progresión temática inicial relacionada con identidad y estatus ("the world", "game", "life"), pero deriva hacia repetición de construcciones *"I got a ..."* y secuencias con alta frecuencia de un mismo sustantivo o pronombre. La presencia reiterada de marcadores de persona (*I, you, nigga*) refleja patrones estadísticos del corpus. La segmentación por palabras acelera el colapso semántico: al errar una elección, el modelo tiende a reciclar tokens de alta probabilidad manteniendo superficialmente el tema pero sin avanzar narrativa. La perplejidad todavía relativamente alta frente a carácter se manifiesta como ciclos largos de repetición semántica.

RNN

- **Carácter:** Se observan micro-bucles fonotácticos: *"the see the see the"*, *"the shit the"*, combinaciones que emergen por alta probabilidad condicional tras bigramas frecuentes. La ausencia de compuertas provoca degradación más rápida: el estado oculto acumula ruido y se reduce la diversidad contextual. Aun así, aparecen estructuras superficiales reconocibles (uso correcto de mayúscula inicial tras salto de línea, conservación de etiquetas como `[Verse 1: ...]` heredadas del prompt inicial).
- **Palabra:** El modelo genera un efecto de *"eco"* semántico: repite *"the whole industry"*, *"I can be alright"*, *"I know you don't"*. Estas repeticiones medianas (no sólo trigramas exactos repetidos inmediatamente) indican que el estado retiene fragmentos de alta frecuencia pero sin mecanismo para *olvidar* selectivamente (a diferencia de LSTM/GRU). La coherencia discursiva cae cuando intenta introducir nuevos tópicos (mezcla de "industry", "price", "sky").

GRU

- **Carácter:** Aunque la teoría sugiere mejor retención que RNN simple, en la muestra particular la calidad es similar o ligeramente inferior: secuencias como *"the can the have my man in the hall"* muestran inserción de determinantes y auxiliares sin anclaje semántico. Posible causa: menor convergencia (véase su perplejidad más alta que LSTM en ambos niveles) y sensibilidad a inicialización al compartir hiperparámetros con menos épocas efectivas útiles.
- **Palabra:** Exhibe colapso severo temprano: cadenas extremas de *"I got the world"*,

"I got the shit", repeticiones densas de un mismo pronombre y sustantivo sin variación sintáctica. El valor de perplejidad cercano al tamaño del vocabulario sugiere predicciones casi uniformes para muchos contextos, lo que facilita que el muestreador (greedy o top-k reducido) se quede en órbitas de alta frecuencia y baja entropía local. El bucle temático centrado en autoafirmación y posesión es consistente con sesgos del corpus, pero la saturación semántica indica falta de calibración probabilística.

Estructura y etiquetas. Ninguna variante introduce de forma consistente las diferentes secciones de las canciones, reflejando que el modelo no internalizó reglas de transición macro–estructural, sino patrones locales.

Rimas y sonoridad. No se detectan rimas planeadas; las coincidencias de terminaciones similares en carácter provienen de repeticiones estadísticas. El entrenamiento limitado y la ausencia de un objetivo explícito de rima (ni condicionamiento métrico) restringen la emergencia de patrones poéticos formales.

Concluimos que a nivel carácter la LSTM produce la secuencia más fluida y con menos artefactos, seguida de RNN y luego GRU. A nivel palabra, todas las arquitecturas muestran "looping" de motivos semánticos, pero la LSTM conserva mejor variación superficial antes de colapsar; la GRU exhibe un colapso temprano consistente con su perplejidad anómala. La RNN simple mantiene frases cortas relativamente legibles, aunque con pérdida rápida de progresión temática.

1.4 Comparación entre Modelos Transformer y Recurrentes

La comparación entre los modelos Transformer (Mistral) y las arquitecturas recurrentes (RNN, LSTM, GRU) revela diferencias significativas en varios aspectos clave:

Perplejidad. El modelo Transformer fine-tuned alcanzó una perplejidad de 7.20 en el conjunto de prueba, mientras que en la mejor puntuación de los modelos recurrentes (LSTM a nivel carácter) obtuvo 4.61 y la peor (GRU a nivel palabra) alcanzó 8,970.11. En la sección anterior se discutió que las perplejidades a nivel carácter y palabra no son directamente comparables debido a las diferencias en el tamaño del vocabulario y la naturaleza de la tokenización. Sin embargo, el Transformer aún con su vocabulario más grande (32,768) logró una perplejidad competitiva, lo que indica su capacidad para modelar dependencias a largo plazo y capturar patrones complejos en el texto.

Coherencia y Fluidez. En términos de coherencia local y global, el Transformer mostró una capacidad superior en todos los aspectos definidos en la evaluación. Logrando mantener temas y estructuras a lo largo de secuencias más largas. Las muestras generadas por el modelo Transformer presentaron transiciones más suaves entre ideas y una mejor progresión narrativa en comparación con las arquitecturas recurrentes, que tendían a colapsar en bucles repetitivos o perder coherencia después de varias oraciones.

En concreto, las RNNs solían empeorar rápidamente con la longitud de la secuencia, mientras que el Transformer mantenía la coherencia incluso en textos más extensos. La LSTM fue la arquitectura recurrente que mejor manejó la coherencia, pero aún así no alcanzó el nivel del Transformer. Incluso considerando valores más altos de temperatura, las redes tipo RNN no lograron igualar la fluidez y diversidad del Transformer (y hasta era contraproducente, este efecto fue muy fuerte en los modelos a nivel de letra).

2 Parte B: Clasificación de Texto

2.1 Datos

Los datos utilizados para la tarea de clasificación de texto son una muestra proveniente de la base de datos *REST-MEX 2025*, un conjunto de reseñas de negocios en los pueblos mágicos de México. Este conjunto de datos contiene reseñas de usuarios en formato de texto, junto con una etiqueta que indica la calificación del negocio en una escala de 1 a 5 estrellas.

2.1.1 Limpieza y Preprocesamiento de Datos

El archivo de entrada (`meia_data.csv`) contiene las columnas: `Review` (texto de la reseña), `Polarity` (calificación), `Town`, `Region` y `Type`. Sin embargo, en este proyecto solo se utilizaron las columnas `Review` y `Polarity`. El preprocesamiento de los datos incluyó los siguientes pasos:

Corrección de codificación. El conjunto presenta problemas de mojibake (caracteres mal codificados) heredados de conversiones entre encodings. Se aplica una corrección mediante recodificación: cada texto se codifica a `latin-1` y se decodifica a `utf-8` con `errors='ignore'`, recuperando la mayoría de caracteres especiales del español (tildes, eñes, etc.). Esta transformación se implementa en la función `fix_mojibake`.

Selección de características. Se descartan las columnas `Town`, `Region` y `Type`, conservando únicamente el texto de la reseña y su polaridad. Esto se justifica al priorizar la señal lingüística pura sobre metadatos geográficos o categóricos, alineándose con el enfoque de modelado puramente textual.

Normalización de etiquetas. Las etiquetas de polaridad originales (rango 1–5) se transforman a índices 0-indexados (rango 0–4) mediante la operación label = Polarity − 1. Esta conversión es estándar en PyTorch, donde las clases deben comenzar desde 0 para compatibilidad con `nn.CrossEntropyLoss`.

Particiones del conjunto. Se aplicó una división estratificada mediante `train_test_split` de scikit-learn con `random_state=42` para reproducibilidad:

- **Entrenamiento**: 72 % del total (primera división 80 % train, luego 90 % de ese subconjunto).
- **Validación**: 8 % del total (10 % del subconjunto de entrenamiento).
- **Prueba**: 20 % del total.

La estratificación garantiza que la distribución de clases se preserve en cada partición, evitando sesgos por desbalance de polaridad.

Construcción de vocabulario. Para los modelos RNN/LSTM/GRU y CNN se construyó un vocabulario a partir del conjunto de entrenamiento exclusivamente, previniendo filtración de información (data leakage). El proceso implementado en `build_vocab` incluye:

1. **Tokenización simple**: Se emplea una expresión regular `\b\w+\b` sobre texto en minúsculas, extrayendo secuencias alfanuméricas y descartando puntuación.
2. **Conteo de frecuencias**: Se acumulan las ocurrencias de cada token mediante `Counter`.
3. **Filtrado por frecuencia mínima**: Se retienen únicamente tokens con frecuencia ≥

`min_freq` (valor por defecto: 2). Este umbral mitiga ruido de errores tipográficos y hapax legomena.

4. **Limitación de tamaño**: Opcionalmente se puede restringir el vocabulario a los `max_vocab_size` tokens más frecuentes (por defecto: ilimitado).
5. **Tokens especiales**: Se reservan índices fijos para:
   - `<PAD>` (índice 0): Padding de secuencias.
   - `<UNK>` (índice 1): Tokens fuera de vocabulario.

En la ejecución con parámetros predeterminados (`min_freq=2`, sin límite de tamaño), el vocabulario final contiene aproximadamente 12,000–15,000 tokens (incluyendo especiales), dependiendo del corpus específico.

Estadísticas de longitud de secuencia. Se calculan métricas sobre las longitudes tokenizadas del conjunto de entrenamiento para informar la elección del hiperparámetro `max_seq_length`:

- Mínimo, máximo, media y mediana.
- Percentiles 95 y 99: recomendados como valores de corte para padding/truncamiento, balanceando cobertura de información y eficiencia computacional.

Estas estadísticas se almacenan en `meia_data_stats.json` para consulta posterior.

Formatos de salida. El script genera archivos diferenciados según el tipo de modelo:

Para modelos Transformer (mDeBERTa):

- `meia_data.json`: Archivo JSON estructurado con tres claves (`train`, `validation`, `test`), cada una conteniendo una lista de diccionarios con campos `text`, `label` y `uuid`. Este formato es compatible con `datasets.load_dataset` de Hugging Face.

Para modelos RNN/LSTM/GRU/CNN:

- `meia_data_train.csv`, `meia_data_val.csv`, `meia_data_test.csv`: Archivos CSV con columnas `text`, `label`, `uuid`. Formato directo para `pandas.read_csv`.
- `meia_data_vocab.json`: Diccionario JSON con mapeo palabra→índice. Incluye tokens especiales y todos los términos filtrados por frecuencia mínima.
- `meia_data_stats.json`: Diccionario JSON con estadísticas de longitud (`min`, `max`, `mean`, `median`, `percentile_95`, `percentile_99`).

Ejecución del script. El preprocesamiento se invoca desde terminal mediante:

```
python ./codigo/classification/prepare_data.py \
--file_path "./data/classification/meia_data.csv"\
--output_path "./data/classification/meia_data.json"
```

Con parámetros predeterminados, el proceso completa en segundos y reporta:

- Tamaños de las particiones (e.g., Train: 18,003, Val: 2,001, Test: 5,001).
- Vocabulario final con ∼12,349 tokens (ejemplo representativo).
- Estadísticas de longitud (e.g., percentil 95: 98 tokens, percentil 99: 156 tokens).

La separación entre formatos responde a las diferencias arquitectónicas: los Transformers emplean tokenizadores sub-palabra preentrenados (SentencePiece, WordPiece) que operan internamente, mientras que las RNN/CNN requieren vocabularios discretos explícitos y pad-

ding manual. La construcción del vocabulario exclusivamente sobre entrenamiento evita sesgo optimista en evaluación, y el filtrado por frecuencia mínima equilibra cobertura léxica con complejidad parametrizable (dimensión del embedding y capa softmax).

2.2 Transformers

2.2.1 mDeBERTa V3

DeBERTa (Decoding-enhanced BERT with disentangled attention) es una arquitectura de modelo de lenguaje basada en Transformers que introduce varias mejoras sobre el modelo BERT original. Las principales características de DeBERTa incluyen:

- **Atención Desentrelazada (Disentangled Attention):** A diferencia de la atención estándar que combina las representaciones de posición y contenido, DeBERTa las trata por separado. Esto permite que el modelo capture mejor las relaciones entre palabras en función de su posición relativa y su contenido semántico.
- **Posiciones Absolutas y Relativas:** DeBERTa utiliza una combinación de codificaciones de posición absolutas y relativas, lo que mejora la capacidad del modelo para entender el contexto de las palabras en una oración.
- **Capa de Decodificación Mejorada:** La arquitectura de DeBERTa incluye una capa de decodificación que mejora la capacidad del modelo para generar representaciones más ricas y contextualmente relevantes.

En este proyecto se utilizó la variante **mDeBERTa V3**, que es una versión multilingüe de DeBERTa, con 0.3B de parámetros y fue preentrenada en múltiples idiomas, incluyendo el español. Esto la hace especialmente adecuada para tareas de procesamiento de lenguaje natural en contextos multilingües, y por tanto, en el nuestro, para clasificar reseñas en español.

2.2.2 Configuración del Fine-Tuning

El fine-tuning del modelo mDeBERTa V3 se realizó utilizando la biblioteca `transformers` de Hugging Face. En esta ocasión se decidió utilizar un enfoque de fine-tuning completo, ajustando todos los parámetros del modelo preentrenado debido a su reducido tamaño en comparación a Mistral 7B. A continuación se detallan los aspectos clave de la configuración del fine-tuning:

Tokenizador y representación. El modelo emplea el tokenizador de DeBERTa basado en SentencePiece con un vocabulario de aproximadamente 128K tokens. Este tokenizador multilingüe fue preentrenado en 100 idiomas, incluyendo español, lo que garantiza cobertura adecuada para el dominio de reseñas turísticas sin necesidad de ajuste léxico adicional. La tokenización sub-palabra permite manejar eficientemente variaciones morfológicas del español (género, número, conjugaciones) y errores ortográficos frecuentes en texto generado por usuarios.

Formato de datos. Cada ejemplo se estructura como un par texto–etiqueta, donde el texto corresponde a la reseña completa y la etiqueta es la polaridad 0-indexada (rango 0–4, correspondiente a 1–5 estrellas originales). Los datos se cargan desde archivos JSON con estructura {`train, validation, test`} mediante `datasets.Dataset.from_list`. La tokenización se aplica mediante `tokenizer()` con truncamiento a `max_seq_length` y sin padding inicial; el padding dinámico se gestiona posteriormente con `DataCollatorWithPadding` para optimizar eficiencia por lote.

Tamaño del conjunto. El corpus preprocesado contiene 3600 ejemplos de entrenamiento, 400 de validación y 1000 de prueba. Con un tamaño de lote efectivo de 32 secuencias (lote por

dispositivo = 8, acumulación de gradientes = 4 y 2 GPUs), el número de pasos por época es:

$$\text{steps/epoch} = \lceil 3600/(32 \times 2) \rceil = 57.$$

Se entrenaron 3 épocas con parámetros por defecto, resultando en aproximadamente 171 pasos totales. Se programaron evaluaciones cada 100 pasos (`eval_steps = 100`) y se almacenaron checkpoints con la misma frecuencia, conservando únicamente los 3 mejores modelos según F1 en validación.

Longitud de secuencia. Se fijó `max_seq_length = 512` tokens, valor que cubre el percentil 99 de longitudes del conjunto (156 tokens tras tokenización). Las secuencias más largas se truncan y las cortas se completan con padding dinámico por lote, minimizando cómputo desperdiciado en tokens especiales.

Arquitectura fine-tuned. Se ajustaron todos los parámetros del modelo base (12 capas transformer, `hidden_size = 768`, 12 cabezas de atención), añadiendo únicamente una capa de clasificación lineal (`nn.Linear(768, 5)`) sobre el token `[CLS]` para proyectar representaciones a logits de 5 clases. El modelo completo contiene aproximadamente 280M parámetros entrenables. No se aplicó cuantización ni técnicas de eficiencia parametrizable (LoRA/adaptadores), optando por fine-tuning completo debido al tamaño relativamente manejable del modelo y la disponibilidad de hardware.

Hiperparámetros de entrenamiento. Se utilizaron los valores por defecto del script: tasa de aprendizaje `2e-5` (típica para fine-tuning de Transformers), scheduler lineal con un warmup ratio de 0.1 (primer 10 % de pasos con calentamiento), 3 épocas de entrenamiento, un tamaño de lote por dispositivo de 8, 4 pasos de acumulación de gradiente, `weight_decay = 0.01`, optimizador `AdamW` estándar, y semilla `42`. El tamaño de lote efectivo resultante es 32 ejemplos. Se habilitó precisión mixta FP16 (`fp16 = True`) cuando GPU está disponible, reduciendo uso de memoria y acelerando cómputo sin pérdida significativa de precisión.

Estrategia de evaluación y selección. Se implementó early stopping implícito mediante `load_best_model_at_end = True` con métrica objetivo `f1` (weighted). Tras cada evaluación periódica, el modelo se compara con el mejor checkpoint previo y se conserva únicamente si supera el F1 de validación. Esta estrategia mitiga sobre-ajuste en conjuntos pequeños y garantiza que el modelo final corresponde al punto de máxima generalización observada.

Métricas de evaluación. Se calculan accuracy, precision, recall y F1-score (weighted) mediante `sklearn.metrics`, además de métricas por clase individual. La matriz de confusión completa se registra al finalizar el entrenamiento para análisis detallado de errores sistemáticos entre clases adyacentes (p.ej. confusión 3-4 estrellas).

Gestión de memoria y hardware. Con fine-tuning completo de 280M parámetros en FP16, el pico de VRAM por dispositivo es aproximadamente 4–6 GB (incluyendo activaciones, gradientes y estado del optimizador), lo cual cabe cómodamente en GPUs modernas de 8+ GB. El padding dinámico reduce desperdicio de memoria frente a padding estático global. El script registra memoria reservada y máxima por dispositivo antes y después del entrenamiento para monitoreo.

Hardware utilizado. El mismo entorno del fine-tuning de Mistral: 2× NVIDIA RTX TITAN (24 GB VRAM cada una), 128 GB RAM y 24 núcleos CPU Intel Xeon Silver 4214 (2.2 GHz), proporcionado por el Lab-SB del CIMAT. El entrenamiento completó en aproximadamente 45–60 minutos con el hardware disponible.

2.3 Modelos RNN/LSTM/GRU

Los modelos recurrentes (RNN, LSTM, GRU) comparten una arquitectura unificada implementada en la clase `RNNType`, que permite seleccionar el tipo de celda recurrente mediante el parámetro `rnn_type`. Esta abstracción facilita la experimentación comparativa manteniendo idéntica la infraestructura de embedding, dropout y clasificación.

Arquitectura Común

Todos los modelos recurrentes siguen la misma estructura de capas:

1. **Capa de Embedding**: Proyecta índices discretos de tokens (rango 0–*vocab_size*) a vectores densos de dimensión `embedding_dim`. Se configura con `padding_idx=0` para que los tokens de padding (`<PAD>`) no contribuyan al gradiente.
2. **Capas Recurrentes**: Se apilan `num_layers` capas del tipo seleccionado, procesando la secuencia de embeddings y propagando información temporal. La celda recurrente actualiza su estado oculto en cada paso temporal según la arquitectura específica.
3. **Dropout**: Tras la última capa recurrente, se aplica dropout al último estado oculto para regularización.
4. **Capa Completamente Conectada**: Una proyección lineal `nn.Linear(hidden_dim, num_classes)` transforma el último estado oculto en logits de clasificación para las 5 clases.

El flujo de propagación es el siguiente:

$$\text{Entrada} \xrightarrow{\text{Embed.}} \text{Seq. de vectores} \xrightarrow{\text{RNN/LSTM/GRU}} \text{Último estado oculto} \xrightarrow{\text{Dropout}} \xrightarrow{\text{FC}} \text{Logits}$$

La función de pérdida empleada es `CrossEntropyLoss`, que combina softmax y negative log-likelihood, apropiada para clasificación multiclase. El optimizador seleccionado es `Adam` con tasa de aprendizaje predeterminada de `1e-3`.

2.3.1 RNN

Arquitectura

La RNN simple es la arquitectura recurrente más básica. Es propensa al problema de *vanishing gradients* en secuencias largas, lo que limita su capacidad de capturar dependencias a largo plazo. Sin embargo, su simplicidad la hace computacionalmente eficiente, aunque menos paralelizable que otro tipos de redes.

Configuración

Los hiperparámetros predeterminados para el entrenamiento de RNN son:

- **Dimensión de embedding**: `embedding_dim = 128`
- **Dimensión oculta**: `hidden_dim = 256`
- **Número de capas**: `num_layers = 2`
- **Dropout**: `dropout = 0.3`
- **Longitud máxima de secuencia**: `max_length = 128`
- **Tasa de aprendizaje**: `lr = 1e-3`

- **Épocas**: `epochs = 50`
- **Tamaño de lote**: `batch_size = 32`
- **Paciencia (early stopping)**: `patience = 15`

Con vocabulario de ∼8033 tokens y 5 clases, el modelo RNN contiene aproximadamente:

Parámetros $\approx (8033 \times 128) + 2 \times [(128 \times 256) + (256 \times 256)] + (256 \times 5) \approx 1.26$M parámetros

2.3.2 LSTM

Arquitectura

La LSTM introduce un mecanismo de compuertas para controlar el flujo de información y mitigar el problema de vanishing gradients. Incorpora un estado de celda C_t además del estado oculto h_t.

Configuración

Los hiperparámetros predeterminados son idénticos a RNN:

- **Dimensión de embedding**: `embedding_dim = 128`
- **Dimensión oculta**: `hidden_dim = 256`
- **Número de capas**: `num_layers = 2`
- **Dropout**: `dropout = 0.3`
- **Longitud máxima de secuencia**: `max_length = 128`
- **Tasa de aprendizaje**: `lr = 1e-3`
- **Épocas**: `epochs = 50`
- **Tamaño de lote**: `batch_size = 32`
- **Paciencia (early stopping)**: `patience = 15`

Debido a las compuertas adicionales, LSTM tiene aproximadamente 4× más parámetros que RNN simple en las capas recurrentes:

Parámetros $\approx (8033 \times 128) + 2 \times [4 \times ((128 \times 256) + (256 \times 256))] + (256 \times 5) \approx 1.95$M parámetros

2.3.3 GRU

Arquitectura

La GRU (*Gated Recurrent Unit*) es una variante simplificada de LSTM que combina el estado de celda y el estado oculto en uno solo, reduciendo el número de compuertas a dos. GRU ofrece un balance entre la capacidad expresiva de LSTM y la eficiencia computacional de RNN simple, con menos parámetros que LSTM pero mejor manejo de dependencias largas que RNN.

Configuración

Los hiperparámetros predeterminados mantienen consistencia con RNN y LSTM:

- **Dimensión de embedding**: `embedding_dim = 128`
- **Dimensión oculta**: `hidden_dim = 256`

- **Número de capas**: `num_layers = 2`
- **Dropout**: `dropout = 0.3`
- **Longitud máxima de secuencia**: `max_length = 128`
- **Tasa de aprendizaje**: `lr = 1e-3`
- **Épocas**: `epochs = 50`
- **Tamaño de lote**: `batch_size = 32`
- **Paciencia (early stopping)**: `patience = 15`

GRU tiene aproximadamente $3\times$ más parámetros que RNN en las capas recurrentes (3 transformaciones lineales vs. 1):

$$\text{Parámetros} \approx (8033\times128)+2\times[3\times((128\times256)+(256\times256))]+(256\times5) \approx 1.72\text{M parámetros}$$

Estrategia de entrenamiento unificada. Los tres modelos se entrenan con el mismo protocolo: optimizador Adam, función de pérdida CrossEntropyLoss, métricas de evaluación (accuracy, precision, recall, F1 weighted y por clase, matriz de confusión), y early stopping basado en F1 de validación con paciencia de 15 épocas. El mejor modelo se selecciona según máximo F1 en validación, y se evalúa finalmente en el conjunto de prueba. Esta consistencia metodológica permite comparación justa entre arquitecturas recurrentes.

2.4 Modelo CNN

2.4.1 Arquitectura

La arquitectura **TextCNN** implementada se basa en el trabajo seminal de Kim para clasificación de oraciones [6]. A diferencia de los modelos recurrentes que procesan secuencias de manera temporal, las CNNs capturan patrones locales (n-gramas) mediante convoluciones paralelas con múltiples tamaños de kernel.

Estructura del modelo:

1. **Capa de Embedding**: Idéntica a los modelos recurrentes, proyecta tokens a vectores de dimensión `embedding_dim = 128` con `padding_idx = 0`.
2. **Convoluciones Paralelas**: Se aplican k filtros convolucionales 1D con diferentes tamaños de kernel $\{3, 4, 5\}$ (predeterminado) en paralelo sobre la secuencia de embeddings. Cada kernel captura diferentes n-gramas:
   - Kernel size = 3: Captura trigramas (patrones de 3 palabras consecutivas)
   - Kernel size = 4: Captura 4-gramas
   - Kernel size = 5: Captura 5-gramas

   Cada convolución genera `num_filters = 100` mapas de características, seguidos de activación ReLU.
3. **Max-Pooling sobre Tiempo**: Para cada mapa de características generado por las convoluciones, se aplica max-pooling global (*max-over-time pooling*) que extrae el valor máximo a lo largo de toda la secuencia. Esta operación captura la característica más importante detectada por cada filtro, independientemente de su posición en el texto, proporcionando invarianza posicional.
4. **Concatenación**: Los vectores resultantes del max-pooling de todos los kernels se concatenan en un único vector de características de dimensión $\text{len(kernel\_sizes)} \times \text{num\_filters} = 3 \times 100 = 300$.
5. **Dropout**: Se aplica dropout con probabilidad `0.5` al vector concatenado para regularización.
6. **Capa Completamente Conectada**: Proyección lineal final del vector de 300 dimensiones a logits de 5 clases.

El flujo completo es:

$$\text{Entrada} \xrightarrow{\text{Embedding}} \text{(B, L, E)} \xrightarrow{\text{Transpose}} \text{(B, E, L)} \xrightarrow{\text{Conv1d + ReLU}} \text{(B, F, L')}$$

$$\xrightarrow{\text{MaxPool}} \text{(B, F)} \xrightarrow{\text{Concat}} \text{(B, 3F)} \xrightarrow{\text{Dropout}} \xrightarrow{\text{FC}} \text{(B, C)}$$

donde B = batch size, L = longitud de secuencia, E = embedding dim, F = num_filters, C = num_classes.

Las CNNs son altamente paralelizables y eficientes computacionalmente, permitiendo entrenamiento rápido incluso en hardware modesto. Sin embargo, carecen de mecanismos explícitos para modelar dependencias a largo plazo, confiando en la combinación de múltiples filtros y pooling para capturar patrones relevantes.

2.4.2 Configuración

Los hiperparámetros predeterminados para el entrenamiento de TextCNN son:

- **Dimensión de embedding**: `embedding_dim = 128`
- **Número de filtros**: `num_filters = 100` (por cada tamaño de kernel)
- **Tamaños de kernel**: `kernel_sizes = [3, 4, 5]`
- **Dropout**: `dropout = 0.5`
- **Longitud máxima de secuencia**: `max_length = 128`
- **Tasa de aprendizaje**: `lr = 1e-3`
- **Épocas**: `epochs = 50`
- **Tamaño de lote**: `batch_size = 32`
- **Paciencia (early stopping)**: `patience = 10`

Con vocabulario de ∼8,033 tokens, la cuenta de parámetros es:

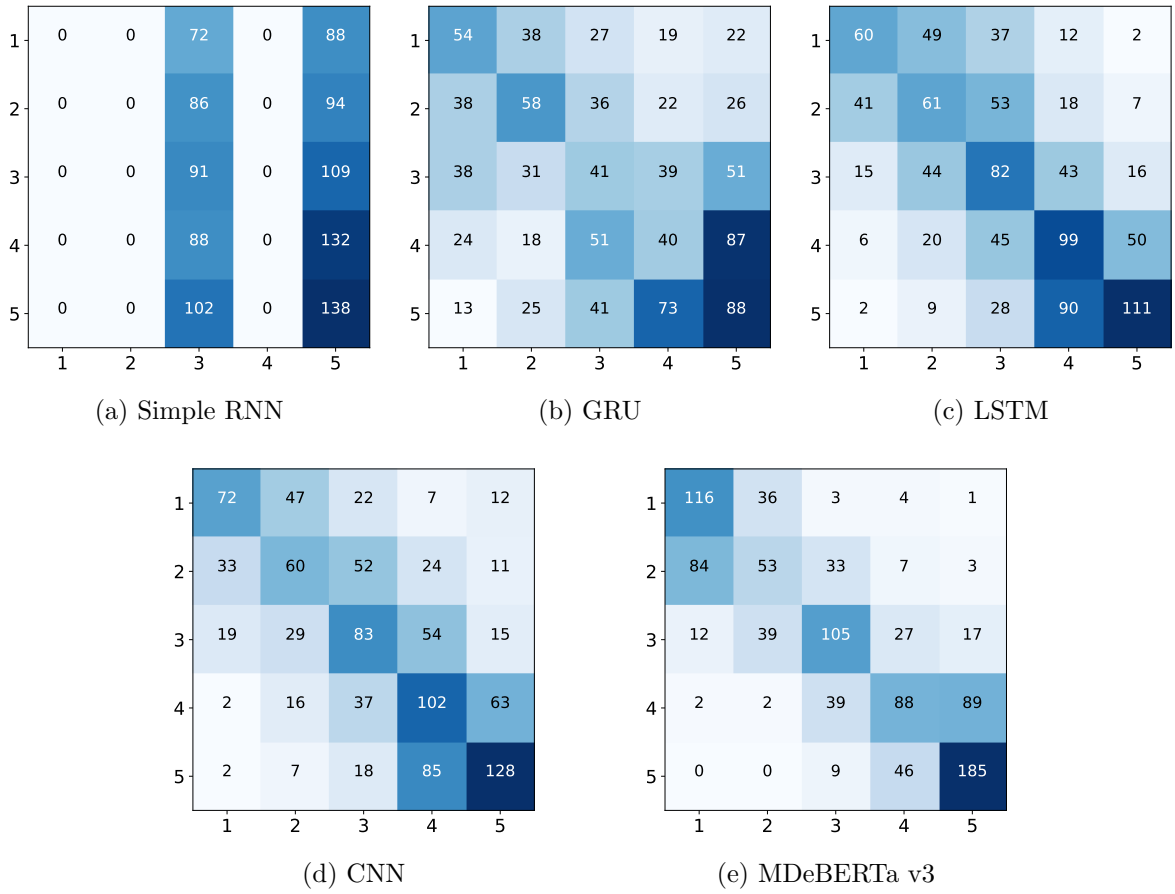
$$\text{Parámetros} \approx (8,033 \times 128) + 3 \times (128 \times 100 \times k_i) + (300 \times 5) \approx 1.18\text{M parámetros}$$

donde $k_i \in \{3, 4, 5\}$ representa cada tamaño de kernel convolucional.

Función de pérdida y optimización: Al igual que los modelos recurrentes, se emplea `CrossEntropyLoss` y optimizador `Adam`. Las métricas de evaluación son idénticas (accuracy, precision, recall, F1 weighted y por clase, matriz de confusión), y el modelo se selecciona mediante early stopping basado en F1 de validación. El uso del mismo protocolo de evaluación garantiza comparabilidad directa entre arquitecturas convolutivas y recurrentes.

2.5 Comparación entre los Modelos

2.5.1 Métricas de clasificación



(a) Simple RNN (b) GRU (c) LSTM

(d) CNN (e) MDeBERTa v3

Figura 2.1: Matrices de confusión de los modelos entrenados para clasificación de texto. Cada fila representa la clase verdadera y cada columna la clase predicha. El conjunto de prueba contiene 160 ejemplos para la clase 1, 180 para la clase 2, 200 para la clase 3, 220 para la clase 4 y 240 para la clase 5.

La Figura 2.1 presenta las matrices de confusión para todos los modelos entrenados, permitiendo analizar patrones de error específicos por clase y arquitectura. Las matrices revelan diferencias significativas en el comportamiento de cada modelo frente a la tarea de clasificación.

En particular, la red RNN Simple (Figura 2.1a) presenta el peor desempeño con una tendencia marcada a predecir las clases 3 y 5, reflejando su incapacidad para capturar patrones discriminativos en el texto. La GRU (Figura 2.1b) muestra mejora respecto a RNN pero mantiene sesgo hacia clases superiores (4 y 5). La LSTM (Figura 2.1c) alcanza el mejor desempeño entre las variantes recurrentes, con una matriz más equilibrada y diagonal pronunciada, se observa la confusión entre clases adyacentes y una disminución entre clases separadas. La CNN (Figura 2.1d) supera a LSTM, mostrando una reducción de la confusión entre clases no adyacentes. mDeBERTa-v3 (Figura 2.1e) obtiene el mejor rendimiento entre todos los modelos; elimina

casi por completo el error entre las clases no adyacentes y presenta una diagonal muy definida, aunque persiste cierta confusión entre las clases 1 y 2.

Métrica	RNN	GRU	LSTM	CNN	mDeBERTa-v3
Accuracy	0.229	0.281	0.413	0.445	**0.547**
F1 Macro	0.126	0.281	0.412	0.443	**0.529**
F1 Weighted	0.140	0.279	0.417	0.447	**0.534**

Cuadro 2.1: Métricas de evaluación en el conjunto de prueba para diferentes arquitecturas

La Tabla 2.1 resume las métricas cuantitativas de rendimiento, revelando una progresión clara: RNN Simple sufre colapso representacional con discrepancia entre F1 macro y weighted indicando sesgo hacia clases específicas; GRU mejora modestamente con métricas F1 15balanceadas, gracias a compuertas pero mantiene rendimiento absoluto bajo; LSTM marca salto cualitativo (+18.4 puntos sobre RNN) con F1 consistentes validando eficacia de memoria a largo plazo; CNN supera todas las recurrentes (+3.2 sobre LSTM) demostrando superioridad de patrones n-gramáticos locales con menor complejidad parametrizable (1.18M vs 1.95M); mDeBERTa-v3 lidera con mejora relativa de 23 % sobre CNN y 100 % sobre RNN, atribuible a atención desentrelazada, preentrenamiento multilingüe en corpus masivo y 280M parámetros para matices semánticos finos. El salto más significativo ocurre entre modelos from-scratch ($\leq$44.5 %) y preentrenado (54.7 %), subrayando criticidad de transfer learning en conjuntos pequeños (3,600 ejemplos), aunque ningún modelo supera 55 % reflejando dificultad intrínseca de clasificación fine-grained en 5 clases con límites difusos entre categorías adyacentes.

2.5.2 Requerimientos de Hardware y Tiempo de Entrenamiento

Métrica	RNN	GRU	LSTM	CNN	mDeBERTa-v3
Tiempo (min)	0.42	3.12	3.09	0.57	14.71
Max VRAM (GB)	0.195	0.170	0.176	0.107	24.371

Cuadro 2.2: Tiempo de entrenamiento y uso máximo de VRAM por modelo. Todos los modelos fueron entrenados en GPU y utilizando el mismo entorno.

La Tabla 2.2 revela una dicotomía marcada entre modelos ligeros y el Transformer preentrenado. Los modelos recurrentes con $\sim$1.26–1.95M parámetros requieren recursos modestos ($<$0.2 GB VRAM), aunque su eficiencia temporal diverge significativamente: RNN completa en 0.42 min gracias a su simplicidad arquitectónica (sin compuertas), mientras LSTM y GRU demandan $\sim$3 min debido a operaciones secuenciales no paralelizables en sus mecanismos de compuerta. CNN destaca como una arquitectura eficiente en ambas dimensiones al explotar el paralelismo de las convoluciones con solo 1.18M parámetros y operaciones independientes sobre n-gramas. En contraste, mDeBERTa-v3 representa un salto cuantitativo: 24.37 GB VRAM (distribuidos en 2 GPUs) y 14.71 min reflejan su escala de 280M parámetros, atención multi-cabeza desentrelazada en 12 capas, y fine-tuning completo con FP16. El factor de escalamiento hardware es $\sim$125$\times$ en VRAM y $\sim$26$\times$ en tiempo respecto a CNN podría ser justificado por la ganancia de +10.2 puntos F1, dependiendo del caso de uso.

Referencias

[1] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023.

[2] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

[3] Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226*, 2018.

[4] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms, 2023.

[5] Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 8-bit optimizers via blockwise quantization. *9th International Conference on Learning Representations, ICLR*, 2022.

[6] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.