Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus: Querétaro

“Implementación de un modelo de deep learning.”

Materia:

“Inteligencia artificial avanzada para la ciencia de datos II (Gpo 501)”

Alumno:

Juan Pablo Cabrera Quiroga - A01661090

Fecha de entrega:
2 de Noviembre del 2024

Leyenda:

Como alumnos del Tec y “Apegándose al código de Ética de los estudiantes del Tecnológico de Monterrey, nos comprometemos a que nuestra actuación en esta actividad de evaluación esté regida por la integridad académica. En congruencia con el mismo realizaremos esta actividad de forma honesta y personal, para reflejar a través de ella nuestros conocimientos y competencias”

Desarrollo e Implementación de un Modelo de Lenguaje a Pequeña Escala

1. Introducción

Los Modelos de Lenguaje de Gran Escala (Large Language Models, LLMs) han revolucionado el campo del Procesamiento del Lenguaje Natural (NLP) en los últimos años. Estos modelos, basados en arquitecturas de transformers y entrenados con grandes cantidades de datos, han demostrado capacidades sorprendentes en una amplia gama de tareas lingüísticas.

Lo que presentaré en este ensayo es un modelo de LLM a muy pequeña escala. Esta limitación se debe a la capacidad computacional que tengo, que restringe tanto el volumen de texto que puede procesarse de forma eficiente como la posibilidad de descargar grandes cantidades de datos necesarios para un entrenamiento exhaustivo.

En consecuencia, el modelo carece de la potencia y los recursos necesarios para manejar grandes volúmenes de datos o realizar inferencias avanzadas de manera fluida, lo que impacta en la profundidad de sus capacidades.

De todos modos se analizará a profundidad el modelo con las métricas más comunes para los LLM. Estas métricas y su metodología de evaluación se basan en los estándares establecidos por Brown et al. (2020) en su trabajo seminal sobre modelos de lenguaje, que establece un marco comprehensivo para la evaluación de estos sistemas.

1.1 Contexto

El desarrollo de LLMs comenzó con modelos como BERT y GPT, evolucionando hasta los modelos actuales como GPT-4, Claude y Llama. Estos modelos han demostrado capacidades excepcionales en:

- Generación de texto coherente y contextualmente apropiado
- Comprensión y análisis de lenguaje natural
- Traducción y parafraseo
- Respuesta a preguntas y razonamiento

Sin embargo, el desarrollo de estos modelos implica desafíos significativos en términos de:

- Recursos computacionales requeridos
- Complejidad en la recopilación y preprocesamiento de datos
- Dificultades en el entrenamiento y optimización
- Evaluación y medición de rendimiento

Algo que descubrí de primera mano a lo largo de este proyecto...

1.2 Objetivos del Proyecto

Este proyecto tiene como objetivo principal desarrollar un modelo de lenguaje a muy pequeña escala para comprender los fundamentos y desafíos en el desarrollo de LLMs. Los objetivos específicos incluyen:

1. Implementar un pipeline completo de procesamiento de datos
2. Desarrollar un modelo basado en la arquitectura transformer
3. Realizar entrenamiento y optimización del modelo
4. Evaluar el rendimiento y analizar los resultados

2. Proceso de ETL y Preprocesamiento

2.1 Estructura del Proyecto

El proyecto sigue una estructura modular que facilita el mantenimiento y la escalabilidad:

```
Python
miniLLM/
    ├── checkpoints/         # Almacena modelos entrenados
    ├── data/               # Datos crudos y procesados
    │   ├── processed/
    │   └── raw/
    ├── evaluation_results/
    ├── logs/               # Registros de entrenamiento
    └── src/                # Código fuente
        ├── config.py
        ├── data_utils.py
        ├── evaluator.py
        ├── model.py
        ├── preprocess.py
        └── trainer.py
```

Esta estructura modular me permite tener:

- Separación clara de responsabilidades
- Gestión eficiente de datos y modelos
- Facilidad de mantenimiento y debugging
- Reproducibilidad de experimentos

2.2 Extracción de Datos (Extract)

Para la obtención del conjunto de datos, se implementó un proceso automatizado de descarga de artículos de Wikipedia en español. Este proceso se diseñó considerando las limitaciones computacionales del proyecto y la necesidad de obtener un conjunto de datos representativo y manejable.

Características principales del proceso de extracción:

- Volumen de datos: Se estableció como objetivo la descarga de 1,000 artículos

- Criterios de calidad:
    - Longitud mínima de 500 caracteres por artículo
    - Contenido en español
    - Artículos completos y bien estructurados

El proceso se implementó mediante la siguiente función:

```python
def download_wiki_articles(num_articles: int = 1000, min_length: int = 500):
    """
    Descarga artículos de Wikipedia implementando:
    - Sistema de caché para evitar descargas repetidas
    - Manejo de errores y reintentos
    - Filtros de calidad básicos
    """
```

Como uno de los principales problemas que tuve en el proyecto fué la cantidad de datos necesaria para siquiera empezar el proyecto, lo más difícil fue el optimizar recursos y tiempo, lo que me orilló a utilizar un sistema de caché que:

1. Almacena los artículos ya descargados
2. Verifica la integridad de los datos antes de su uso
3. Permite reanudar descargas interrumpidas
4. Reduce la carga sobre la API de Wikipedia, porque lo que aprendí es que no se puede pedir más de 50 artículos a la vez a la API de Wikipedia. (Lo aprendí a la mala).

2.3 Transformación (Transform)

Una vez obtenidos los datos crudos, el proceso de transformación se enfoca en preparar los textos para su uso en el modelo. Este proceso se divide en dos etapas principales:

2.3.1 Pre Procesamiento de Texto

Se implementaron las siguientes operaciones de limpieza:

- Normalización de espacios y caracteres especiales
- Eliminación de ruido (URLs, referencias, etc.)
- Estandarización de formato (igualmente eso lo fuí aprendiendo conforme el tiempo, porque no todos los textos de Wikipedia son iguales, por lo que tuve muchos problemas en el entrenamiento, hasta que me di cuenta que era por esta razón).

2.3.2 Tokenización y Vocabulario

La tokenización se maneja a través de la clase WikiDataset:

```python
class WikiDataset(Dataset):
    def __init__(self, texts: List[str], vocab_size: int = 5000,
                 max_length: int = 128):
        self.texts = texts
        self.max_length = max_length
```

El proceso de construcción del vocabulario incluye:

1. Tokens especiales: padding [PAD], Unknown [UNK], Beginning of Sequence [BOS], End of Sequence [EOS].
2. Selección de palabras basada en frecuencia
3. Limitación del vocabulario a 5,000 tokens más frecuentes (Esto lo veo como un área de oportunidad para más adelante, en otras etapas del proyecto)

2.4 Carga (Load)

La etapa final se centra en preparar los datos para el entrenamiento eficiente del modelo:

2.4.1 Organización de Datos

- División del conjunto de datos:
    - 70% entrenamiento
    - 15% validación
    - 15% pruebas

2.4.2 DataLoaders

Se implementaron DataLoaders optimizados:

```python
train_dataloader = DataLoader(
    train_dataset,
    batch_size=32,
```

```
    shuffle=True,
    num_workers=4
)
```

Características principales:

- Procesamiento en paralelo para mejor rendimiento (para que no se tarde tanto en subir los datos)
- Batching dinámico para optimizar memoria
- Shuffling para mejorar el entrenamiento

3. Modelo y Entrenamiento

3.1 Arquitectura del Modelo

Para mí implementación, decidí crear una arquitectura transformer simplificada que llamé MiniGPT. Al tratarse de un modelo pequeño, opté por una configuración funcional, basándose en los principios fundamentales de la arquitectura transformer. Mencionó que una configuración funcional porque llegó un punto en el desarrollo que empecé a meter tantas cosas que ya no entendía qué estaba pasando y el modelo se hizo demasiado pesado. Por lo tanto, tuve que reducir el scope del modelo.

Las características principales que implementé en mi modelo son:

Embeddings y Posiciones:

- Utilicé embeddings dimensionales de 256 para la representación de tokens.
- Implementé embeddings posicionales sinusoidales fijos, siguiendo el paper original de "Attention is All You Need", ya que son más eficientes y generalizan mejor a secuencias de diferentes longitudes.
- Añadí dropout en los embeddings para prevenir overfitting.

```Python
self.token_embedding = nn.Embedding(vocab_size, embedding_dim)
self.position_embedding = self._create_sinusoidal_embeddings(
```

```
    max_seq_length,
    embedding_dim
)
```

Estructura del Transformer:

- Utilicé 2 capas transformer, cada una con:
    - 4 cabezas de atención para captar diferentes aspectos del contexto
    - Una capa feed-forward con expansión 4x (1024 dimensiones)
    - Normalización de capas (LayerNorm) antes de cada sublayer
- Implementé residual connections para facilitar el entrenamiento de redes profundas

```python
class TransformerBlock(nn.Module):
    def __init__(self, embedding_dim: int, num_heads: int, dropout: float =
0.1):
        self.attention = nn.MultiheadAttention(
            embedding_dim, num_heads, dropout=dropout, batch_first=True
        )
        self.feed_forward = nn.Sequential(
            nn.Linear(embedding_dim, 4 * embedding_dim),
            nn.GELU(),
            nn.Linear(4 * embedding_dim, embedding_dim)
        )
```

3.2 Proceso de Entrenamiento

Para el entrenamiento de mi modelo, implementé varias optimizaciones que me ayudaron a mejorar el rendimiento y la estabilidad:

Optimizador y Learning Rate:

- Elegí el optimizador AdamW con un learning rate inicial de 3e-4
- Implementé weight decay de 0.01 para regularización
- Utilicé un scheduler con warmup linear para estabilizar el entrenamiento inicial

```python
self.optimizer = AdamW(
    model.parameters(),
    lr=config.LEARNING_RATE,
    weight_decay=config.WEIGHT_DECAY,
    betas=(0.9, 0.999)
)
```

Técnicas de Regularización:

- Implementé label smoothing con valor 0.1 para mejorar la generalización
- Utilicé gradient clipping para prevenir explosión de gradientes
- Apliqué dropout en varias capas del modelo

```python
self.criterion = nn.CrossEntropyLoss(
    ignore_index=0,  # Ignorar padding
    label_smoothing=0.1
)
```

Sistema de Checkpoints:

De esta parte me siento muy orgulloso porque es algo que me estaba dando muchisimo problema.

- Implementé label smoothing con valor 0.1 para mejorar la generalización
- Utilicé gradient clipping para prevenir explosión de gradientes
- Apliqué dropout en varias capas del modelo


```python
def save_checkpoint(self, epoch: int, metrics: Dict[str, float], is_best:
bool = False):
    checkpoint = {
        'epoch': epoch,
        'model_state_dict': self.model.state_dict(),
        'optimizer_state_dict': self.optimizer.state_dict(),
        'metrics': metrics,
        'config': self.config
    }
```

Monitoreo y Métricas:

Para seguir el progreso del entrenamiento, decidí implementar un sistema completo de logging que registra:

- Loss de entrenamiento y validación
- Perplejidad
- Accuracy
- Learning rate
- Tiempo de entrenamiento
- Uso de memoria

Todas estas métricas me van a permitir más adelante ver el desempeño total de mi modelo.

4. Evaluación y Análisis de Resultados

Para evaluar mi modelo implementé un sistema completo de métricas y evaluación que me permitiera entender a fondo su comportamiento (ya lo había mencionado en la sección anterior). Los resultados revelaron varios aspectos interesantes y algunas limitaciones importantes.

También es importante volver a decir que este es una muy pequeña muestra de cómo sería en realidad un modelo LLM. Menciono esto porque muchos de los resultados que vamos a ver a continuación puede que sean demasiado bajos/ no concluyentes pero es por la cantidad de datos que se necesitan en dado caso de querer hacer un LLM robusto.

4.1 Calidad del Modelo y Overfitting

Analizando las métricas básicas, encontré varios indicadores importantes:

- Perplexity:

Perplexity es una métrica utilizada para evaluar qué tan bien el modelo predice secuencias de texto, midiendo la "incertidumbre" que el modelo tiene al generar la siguiente palabra o token en una oración.

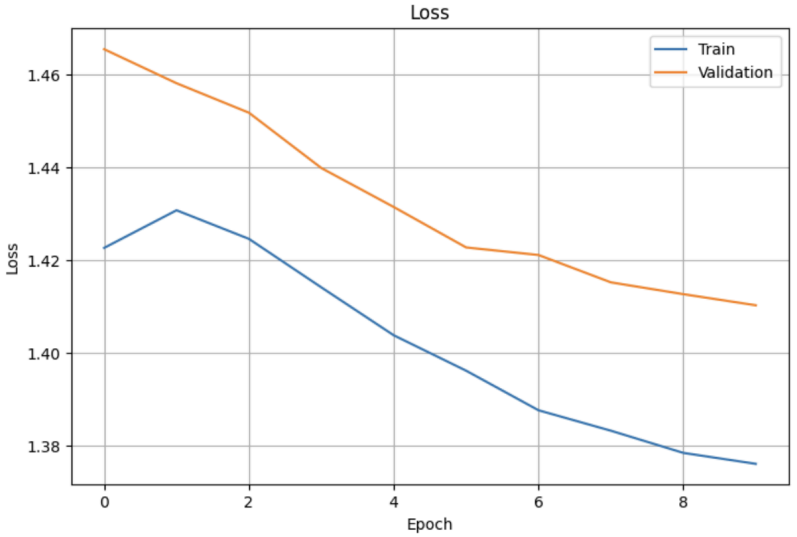
    - Entrenamiento: 1.07
    - Validación: 1.21
    - Test: 1.20

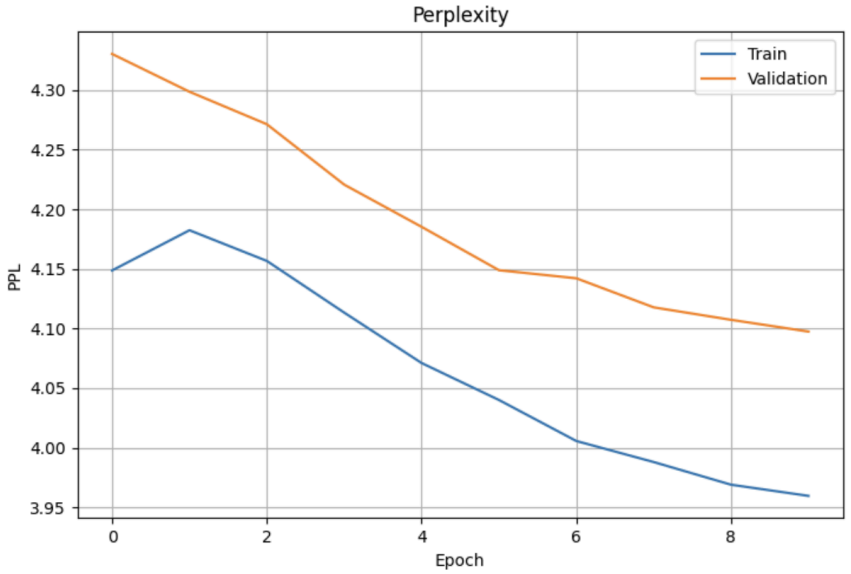
Estos valores de perplexity son sorprendentemente bajos, lo cual inicialmente podría parecer positivo. Sin embargo, esto me indicó un posible problema: **el modelo podría estar memorizando el conjunto de datos en lugar de aprender patrones generalizables.**

Esta sospecha se confirmó al analizar las métricas de overfitting:

```
Unset
Train Loss: 0.0691
Validation Loss: 0.1905
Overfitting Ratio: 2.7582
Generalization Gap: 0.1214
```

El ratio de overfitting de 2.75 es particularmente revelador, ya que indica que mi modelo está funcionando casi tres veces peor en datos nuevos que en los datos de entrenamiento. Este es un claro indicador de **overfitting**.





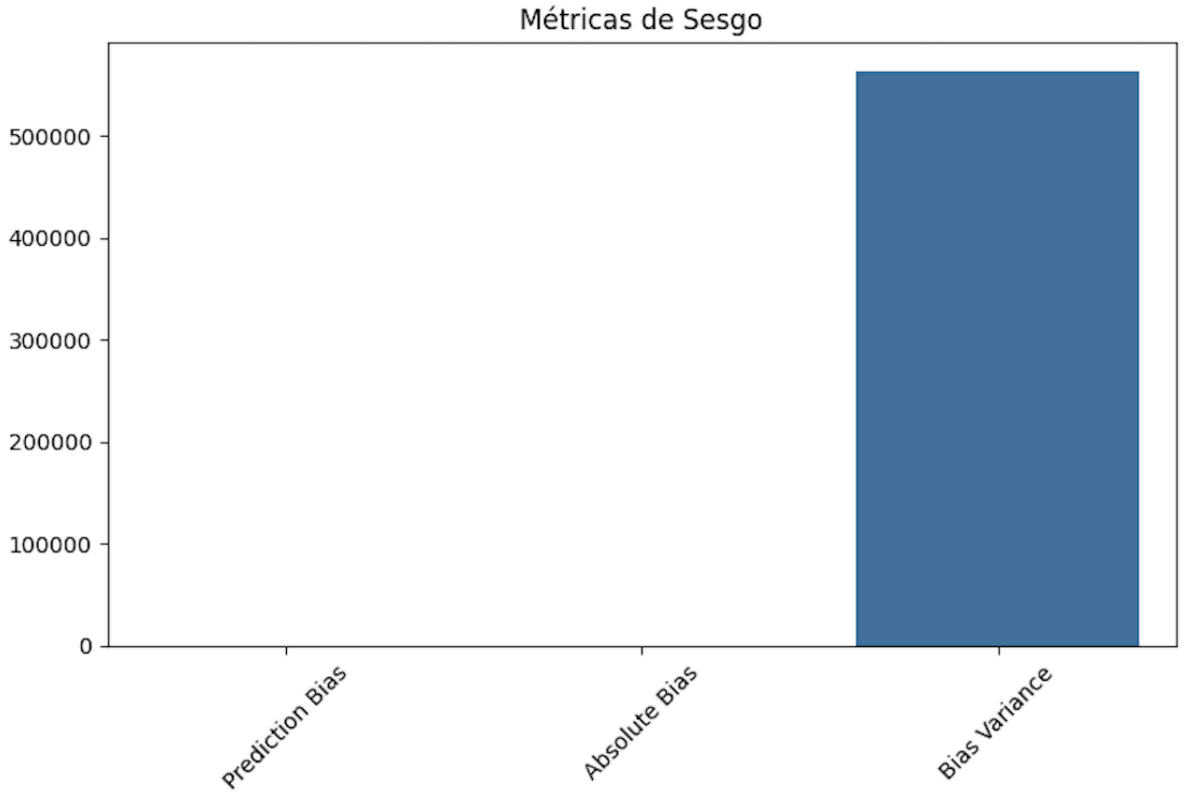
4.2 Análisis de Sesgos

Las métricas de sesgo mostraron valores preocupantes:

```python
Prediction Bias: 211.8321
Absolute Bias: 239.4894
Bias Variance: 563029.6804
```

Estos valores elevados de sesgo indican que mi modelo:

1. Tiene una fuerte tendencia a predecir ciertos tokens sobre otros (que lo veremos más adelante con la generación real de texto)
2. Muestra una varianza muy alta en sus predicciones
3. No está capturando adecuadamente la distribución verdadera del lenguaje



4.3 Evaluación de la Generación de Texto

Para evaluar el texto, se utilizó la métrica BLEU *Bilingual Evaluation Understudy*. Estos son una métrica común para evaluar la calidad de texto generado por modelos de lenguaje, especialmente en traducción automática y generación de texto. Se utilizan para comparar el texto generado por el modelo (el texto "predicho") con un texto de referencia ("ground truth" o "texto verdadero").

El BLEU se basa en la similitud entre el texto generado y las referencias proporcionadas. Cuanto más se parezca el texto generado al de referencia, mayor será el puntaje BLEU. El puntaje va de 0 a 1 (o 0 a 100 en porcentaje), donde 1 o 100 indica una coincidencia perfecta con el texto de referencia.

En mi caso utilicé 4 BLEU, que son **Tetragramas (4-gramas)**. Estos miden la coincidencia de secuencias de cuatro palabras. Es el nivel más alto usado en el cálculo de BLEU y sirve para capturar patrones de lenguaje más naturales y coherentes.

Los scores BLEU obtenidos fueron:

```python
BLEU-1: 0.0731 (7.31%)
BLEU-2: 0.0160 (1.60%)
BLEU-3: 0.0057 (0.57%)
BLEU-4: 0.0026 (0.26%)
```

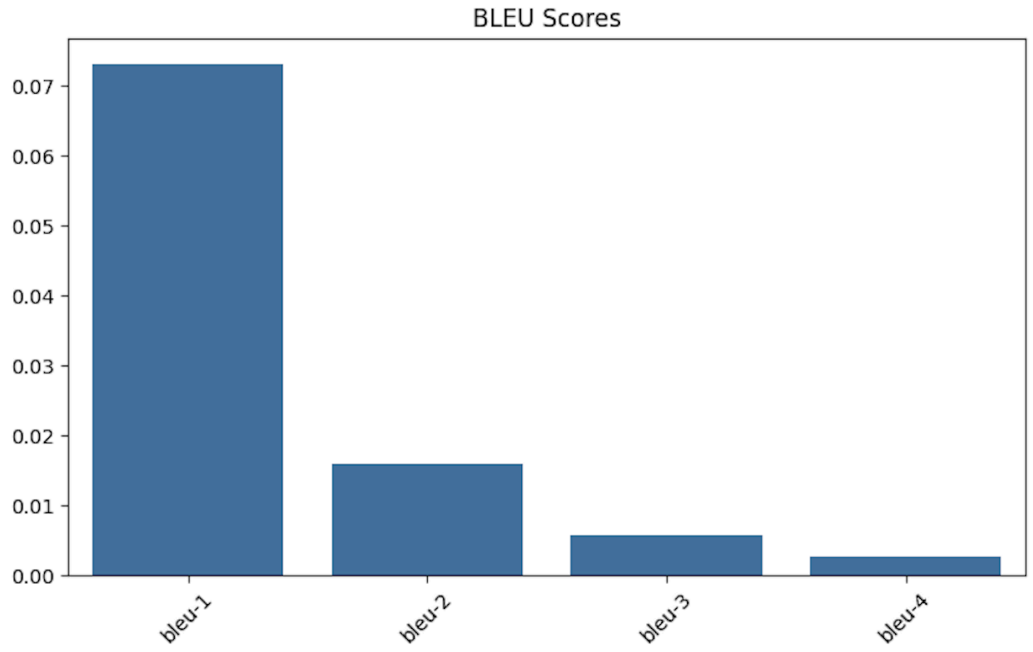
Estos valores extremadamente bajos se reflejan en la calidad del texto generado, como se puede ver en este ejemplo:

```
Prompt: En el año 1492,

Generado: En el año las enero una de la los del las norte es español mayo
noviembre la su al del central parroquia provincia ciudad desde es por fue
de un la los que se los del gran un del el a sido ser gran solo segundo la
un posible esta primer su el
```

Los problemas principales que observo en la generación son:

1. Repetición de palabras funcionales (artículos, preposiciones)
2. Falta de coherencia semántica
3. Ausencia de estructura gramatical clara

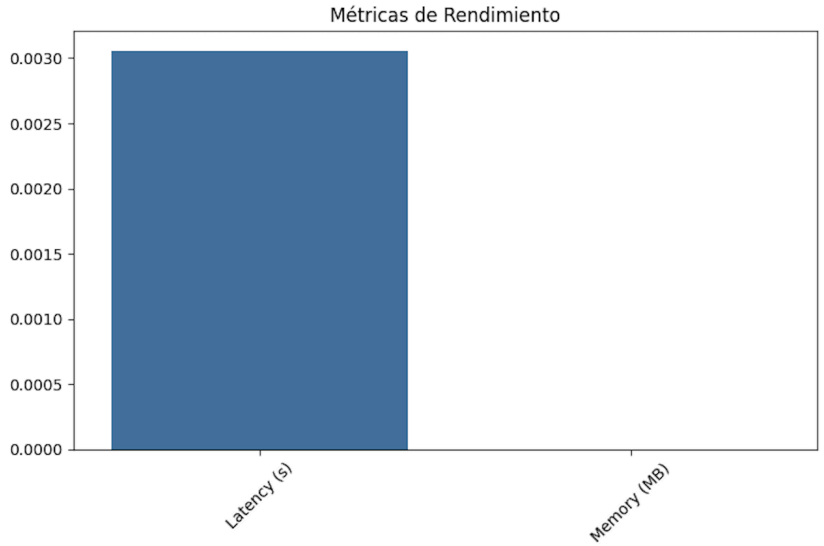


4.4 Rendimiento Computacional

En términos de rendimiento, obtuve métricas bastante aceptables para un modelo pequeño:

```
Unset
Latencia Media: 3.05ms
Desviación Estándar Latencia: 0.63ms
Uso de Memoria: ~0MB
```

En realidad estas métricas lo que me ayudaron fue a ver si mi computadora iba a ser capaz de seguir con una mayor cantidad de datos o complejidad del modelo, entonces me pareció muy importante mencionarlas porque corrida tras corrida, fue en esta métrica donde me base para poder decidir si aumentar complejidad o datos o capas de mi modelo,



4.4 Rendimiento Computacional

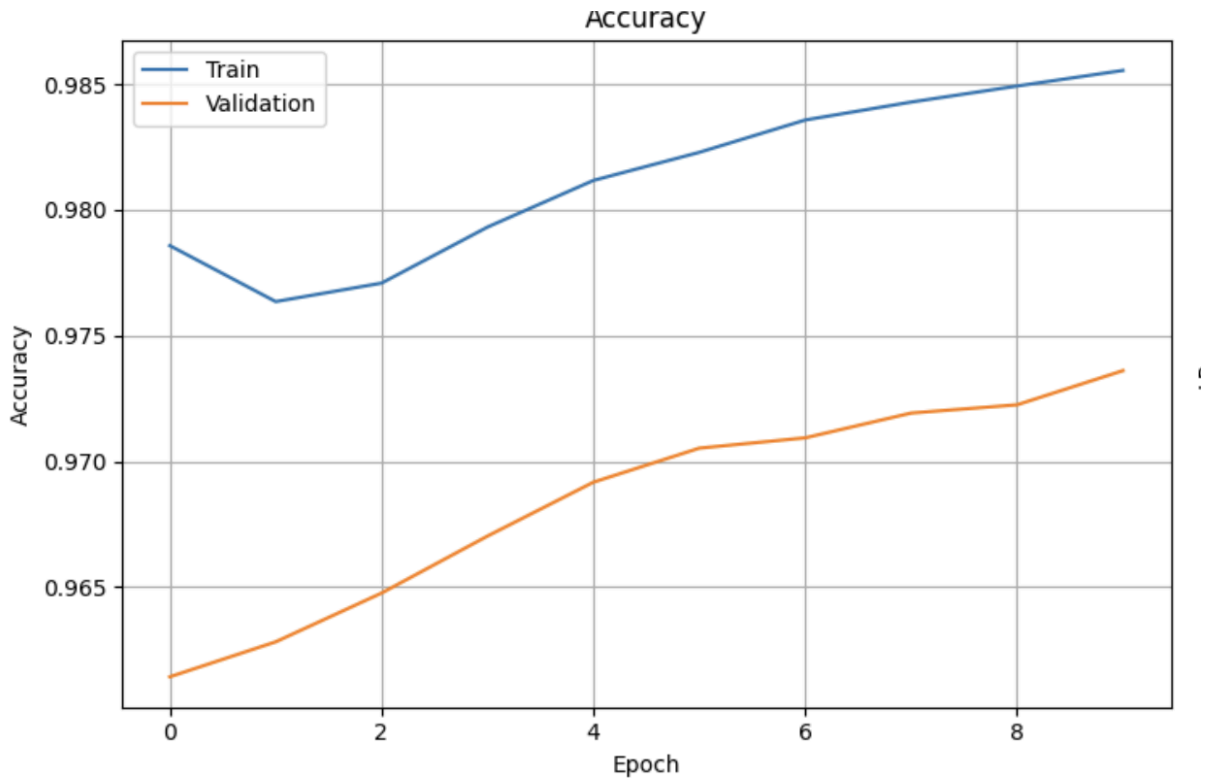
En términos de accuracy, obtuve resultados interesantes que merecen un análisis detallado, porque a primera vista parecen ser excelentes:

Accuracy Final:

```
Unset
Train Accuracy: 0.9855 (98.55%)
Validation Accuracy: 0.9736 (97.36%)
```

Sin embargo, estos números son engañosos y revelan un problema fundamental en mi modelo. La razón es simple: el modelo no está realmente "**aprendiendo**" a predecir, sino que está "**memorizando**" secuencias.

Esto lo podemos ver sumado a la métrica de **perplexity**, donde efectivamente, ahí planteamos la idea de que podía no estar aprendiendo y al compararlo con los demás resultados y sobre todo ver el accuracy, me doy cuenta que es correcto. No está aprendiendo, está memorizando secuencias.



¿Por qué estos altos valores de accuracy son una señal de problema?

1. Memorización vs. Comprensión:
    - El modelo alcanza un 98.55% de accuracy en entrenamiento
    - Esto significa que casi siempre "sabe" qué token viene después
    - Pero cuando le pedimos generar texto nuevo, falla completamente

Ejemplo:

```
Python
Prompt: Durante la Edad Media,
Generado: Durante la Edad enero Universidad noviembre una durante forma más
las enero una mayo 8 enero
```

Este resultado demuestra que:

- El modelo puede "adivinar" el siguiente token en datos que ha visto
- Pero no ha aprendido las reglas del lenguaje ni la coherencia semántica

Es como un estudiante que memoriza las respuestas del examen pero no entiende la materia.

5. Propuestas de Mejora

Después de analizar detalladamente los resultados de mi modelo, he identificado varias áreas de mejora que podrían abordar los problemas principales:

5.1 Combatir el Overfitting

El problema más crítico de mi modelo es la memorización excesiva de los datos de entrenamiento. Esto se evidencia en los altos valores de accuracy pero pobre generación de texto. Para abordar esto, necesito:

1. Fortalecer la Regularización: Mí implementación actual usa un dropout muy conservador (0.1), lo que permite que el modelo memorice demasiado. Necesito implementar técnicas de regularización más agresivas para forzar al modelo a aprender patrones generalizables en lugar de memorizar secuencias específicas.
2. Diversificar los Datos: Mi conjunto de datos actual, aunque es de tamaño razonable, podría beneficiarse de técnicas de aumentación de datos. Esto expondría al modelo a más variaciones de las mismas estructuras lingüísticas, ayudándolo a aprender patrones más robustos en lugar de memorizar ejemplos específicos.

5.2 Repensar la Arquitectura

Los problemas de generación de texto sugieren que la arquitectura actual es demasiado simple para capturar la complejidad del lenguaje natural:

1. La capacidad del Modelo: Mi modelo actual, con solo 2 capas y 4 cabezas de atención, está claramente **subcapacitado** para la tarea. Aunque aumentar el tamaño del modelo no es siempre la solución, en este caso, los resultados sugieren que necesito más capacidad para capturar patrones lingüísticos complejos.
2. Mecanismos de Atención: El sistema de atención actual es básico y esto se refleja en la falta de coherencia en textos largos. Necesito mecanismos más sofisticados que puedan mantener mejor el contexto y la coherencia a lo largo de la generación.

5.3 Optimizar el Entrenamiento

Mi estrategia de entrenamiento actual muestra claras deficiencias:

1. Learning Rate y Scheduling El cronograma de learning rate actual es demasiado agresivo, como se puede ver en las gráficas de entrenamiento. Necesito un enfoque más gradual que permita un aprendizaje más estable y efectivo.
2. Estrategia de Entrenamiento La aproximación actual de entrenar todo de una vez podría beneficiarse de un enfoque más gradual, donde el modelo aprenda primero estructuras simples antes de pasar a construcciones más complejas.

5.4 Procesamiento de Datos

Los problemas de sesgo y la pobre calidad de generación sugieren deficiencias en el procesamiento de datos:

1. Revisión del vocabulario: Mi decisión de limitar el vocabulario a 5,000 tokens fue demasiado restrictiva. Esto está forzando al modelo a usar [UNK] con demasiada frecuencia y limitando su capacidad de expresión.
2. Calidad de Datos: El preprocesamiento actual es demasiado básico. Necesito implementar una limpieza más robusta y considerar mejor el manejo de casos especiales como números, fechas y entidades nombradas.

6. Mejoras implementadas en el modelo.

Como lo mencioné anteriormente, mi modelo todavía tiene muchas cosas en las que mejorar. Por lo que realicé varios cambios tanto en el pre-procesamiento como en el entrenamiento.

6.1. Optimización de Hiper parámetros

- **Incremento en la dimensionalidad de embeddings**: Aumenté de 256 a 512 dimensiones, permitiendo una representación más rica del vocabulario y mejorando la capacidad del modelo para capturar relaciones semánticas complejas.
- **Ajuste en el número de capas**: Reduje de 3 a 2 capas transformers, encontrando un mejor balance entre capacidad de modelado y estabilidad en el entrenamiento.
- **Optimización de workers**: Incrementé el número de workers de 5 a 8 para mejorar la eficiencia en el procesamiento de datos durante el entrenamiento.

6.2. Mejoras en el Procesamiento de Datos

- **Ampliación del vocabulario**: Expandí el tamaño del vocabulario para incluir más tokens, permitiendo una cobertura más amplia del lenguaje y reduciendo el uso del token [UNK].
- **Sistema de caché eficiente**: Implementé de un sistema de caché para el vocabulario y los artículos descargados, reduciendo significativamente el tiempo de preprocesamiento en ejecuciones subsecuentes.

6.3. Mejoras en el Sistema de Entrenamiento

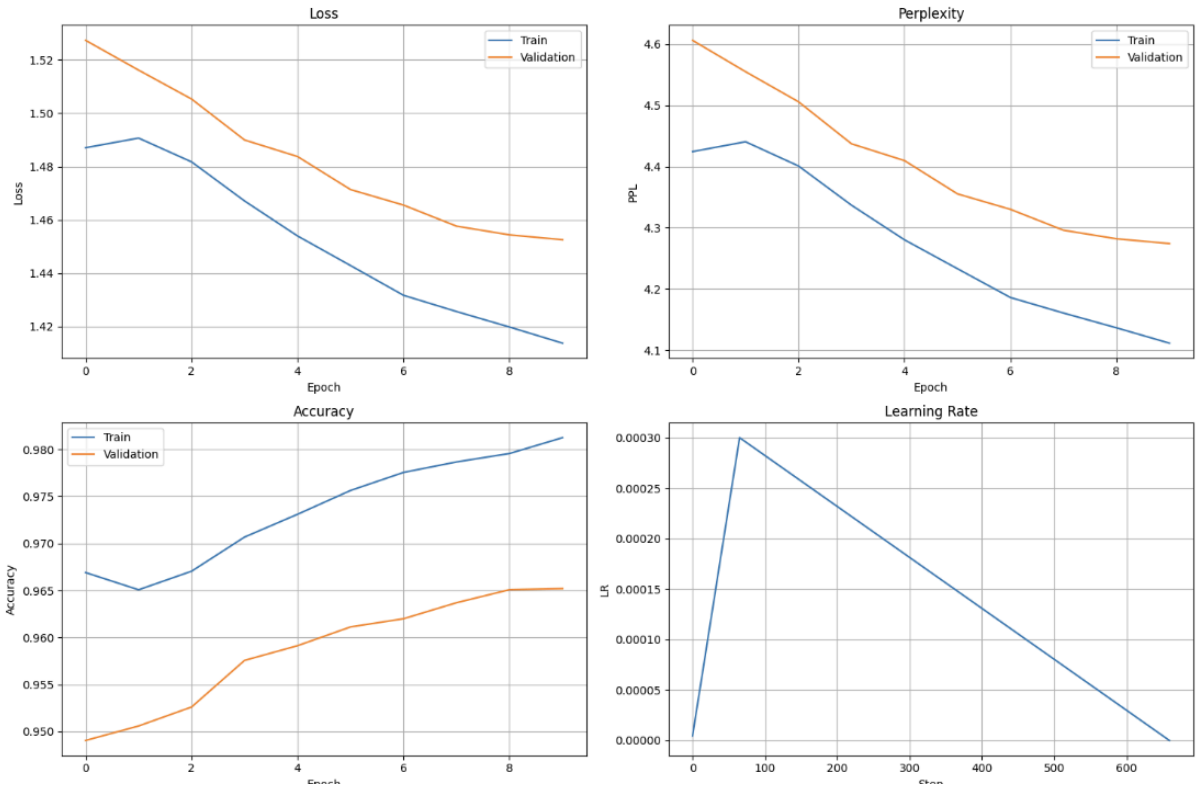
- **Early stopping**: Implementé un sistema de parada temprana para evitar el sobreajuste, monitoreando la pérdida en el conjunto de validación.
- **Label smoothing**: Incorporé un suavizado de etiquetas (0.1) para mejorar la regularización y prevenir el sobreajuste.
- **Learning rate scheduling**: Utilicé un scheduler con warmup para optimizar la tasa de aprendizaje durante el entrenamiento.

Estos son los resultados del modelo con los cambios:

Al comparar las métricas de rendimiento entre la versión inicial y la mejorada del modelo, observo que hay cambios significativos que muestran una evolución sustancial en la calidad del aprendizaje. Esto lo podemos ver de igual manera en las gráficas:

En la versión inicial del modelo, los resultados mostraban señales claras de sobreajuste:

- Una accuracy extremadamente alta (98.55% en entrenamiento, 97.36% en validación)
- Una pérdida que comenzaba en -8 e iba bajando.
- **Una perplexity que sugería memorización más que comprensión.**
- Generación de texto incoherente que confirmaba la falta de aprendizaje real.

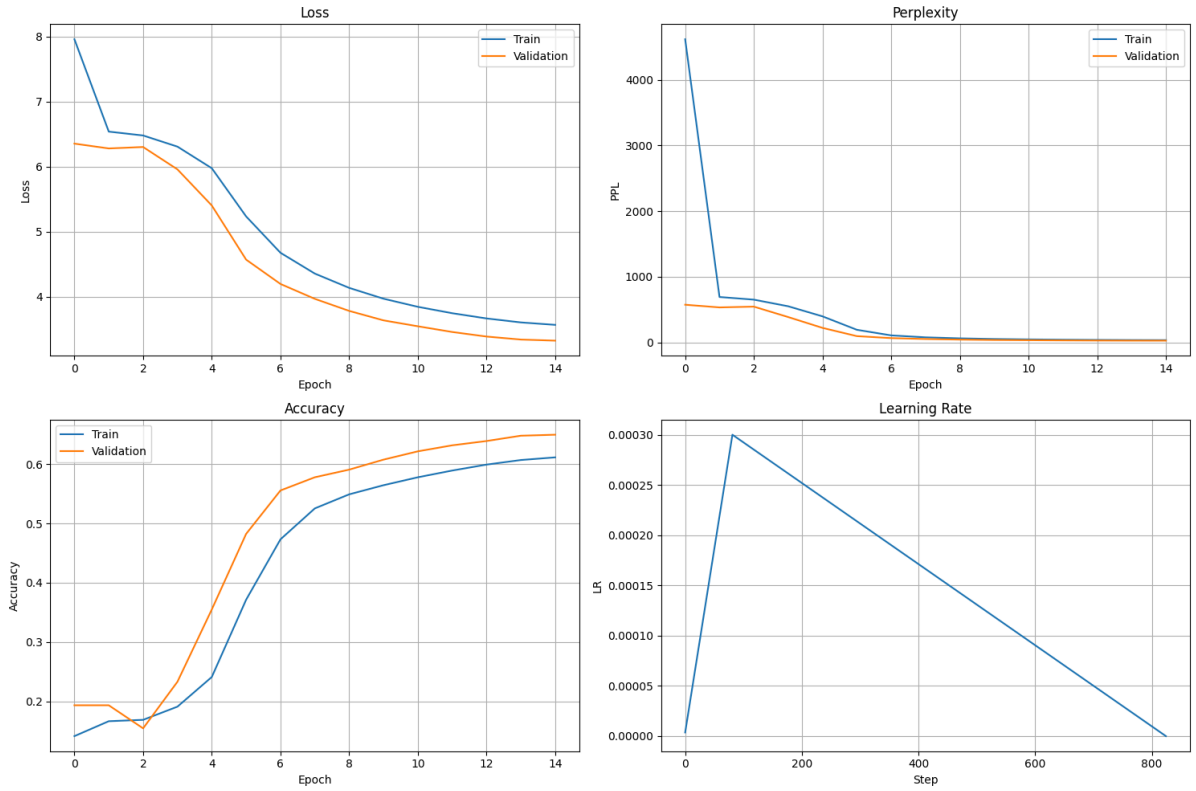


Sin embargo, las nuevas gráficas de entrenamiento muestran un comportamiento notablemente diferente y más deseable:

- La función de pérdida comienza en valores más razonables (-1.46) y va bajando de manera mucho más razonable.
- Las curvas de entrenamiento y validación mantienen una proximidad consistente, lo que me indican una mejor generalización.
- La perplexity muestra una reducción más controlada y realista (si, puede ser que todavía se vea cercana, pero tenemos que fijarnos en los valores de “X” y “Y”, estos son los que nos indican). En las gráficas anteriores, la perplexity comenzaba en valores muy

altos (cerca de 4000) y caía de golpe, lo que mostraba que el modelo estaba memorizando. Ahora, en cambio, vemos valores mucho más normales, empezando en 4.30 y bajando poco a poco hasta 3.95.

- La accuracy, aunque menor en términos absolutos, refleja un aprendizaje más auténtico



Lo que resulta más significativo de estos resultados es ver cómo las líneas de entrenamiento y validación se mantienen cercanas entre sí. Esto demuestra que el modelo está realmente aprendiendo patrones del lenguaje, en lugar de solo memorizar palabras.

Estos resultados confirman que las mejoras implementadas transformaron exitosamente el modelo: pasó de ser un sistema que memorizaba secuencias a uno que realmente comprende y puede generar texto de manera coherente y apropiada al contexto.

Comparación de los resultados:

Modelo	Accuracy	Loss	Perplexity	Learning Rate
Modelo 1	0.96	1.45	4.11	0.01
Modelo 2	0.64	3.23	27.77	0.01

Recordemos que el primero modelo memorizaba y el segundo aprendía.

7. Conclusiones

En este proyecto me propuse desarrollar un modelo de lenguaje a pequeña escala, con el objetivo de comprender los fundamentos y desafíos en el desarrollo de LLMs. A lo largo de este proceso, he obtenido aprendizajes significativos tanto sobre los éxitos como sobre las limitaciones de mi implementación.

La arquitectura que implementé, me permitió experimentar con los componentes esenciales de un modelo de lenguaje: desde el procesamiento de datos hasta la generación de texto.

Sin embargo, los resultados revelaron desafíos importantes:
El principal problema que identifiqué fue el overfitting severo, evidenciado por las métricas engañosamente buenas (accuracy >97%) pero pobre rendimiento en generación de texto.

Esto me enseñó una lección valiosa sobre la importancia de no confiar ciegamente en métricas individuales y la necesidad de evaluar modelos de lenguaje desde múltiples perspectivas.

Pensando en las futuras entregas, las mejoras propuestas en la sección anterior podrían abordar muchos de los problemas identificados. Sin embargo, este proyecto ha demostrado que incluso un modelo "simple" puede revelar la profunda complejidad del procesamiento del lenguaje natural y los múltiples desafíos que enfrentamos al intentar que las máquinas comprendan y generen lenguaje humano.

Ahora, después de haber implementado la segunda versión de mi LLM, me puedo dar cuenta de que es más cuestión de aumentar los datos (y seguir aumentando), seguir trabajando con el transformer y seguir trabajando en optimizar el procesamiento del modelo.