Hacia Modelos de Lenguaje Accesibles: Arquitecturas Transformer Compactas y Eficientes

Kitsun

18 de marzo de 2025

Resumen

En este trabajo documento el desarrollo de un modelo de IA eficiente basado en arquitectura transformer. Implemento cuatro técnicas clave: activaciones de bajo rango (CoLA), Flash Attention, optimizador Muon y modelado de periodicidad (FANformer), combinadas con normalización híbrida (HybridNorm) y un tokenizador optimizado. La extensión CoLA FAN integra las ventajas de factorización de bajo rango con representaciones en el dominio de frecuencia. Mis resultados muestran una reducción de parámetros del 63.5 % (de 236M a 86.2M) con impacto mínimo en rendimiento (mejora en pérdida de 4.50 a 4.49). La implementación completa (CoLA + FANformer + Dropout + SmolLM2 + Muon) logra la menor pérdida final (4.41) con un compromiso aceptable en velocidad (4.42 it/s). Destaco la aceleración de 2.2× en velocidad de entrenamiento básica, procesando 100,000 ejemplos en aproximadamente 15 minutos en una RTX 4090 (6.67 it/s vs 3.00 it/s en el modelo base). El optimizador Muon presenta una penalización del 22 % en velocidad (5.20 it/s vs 6.67 it/s) con uso de GPU menos consistente (75 % promedio vs 98 % con AdamW), pero logra una reducción de pérdida de 36-37 % comparado con 13-19 % de AdamW. El refinamiento cuidadoso de hiperparámetros demostró ser crucial, permitiendo alcanzar mejores resultados (4.065 vs 4.21) utilizando apenas el 8.3 % de los datos originales (208,333 vs 2.5 millones). Analizo la dependencia de Muon al tamaño de batch, demostrando que con batches pequeños Muon es menos eficiente que AdamW, pero su eficiencia escala significativamente mejor con batches grandes. Realizo análisis detallados de memoria, escalabilidad en diferentes GPUs y eficiencia computacional, demostrando que este enfoque permite crear modelos transformer accesibles y eficientes en hardware de consumo.

1. Introducción

El desarrollo de modelos transformer eficientes representa un desafío significativo en el campo de la IA. Mientras que los modelos de gran escala han demostrado capacidades excepcionales, su entrenamiento e implementación requieren recursos computacionales extensivos, limitando su accesibilidad. En este trabajo abordo esta problemática desarrollando arquitecturas optimizadas que puedan entrenarse en hardware accesible sin sacrificar capacidad predictiva.

Como demuestra el reciente trabajo de [1] titulado "SmolLM2: When Smol Goes Big", en modelos pequeños la optimización debe equilibrar tanto la arquitectura del modelo como las estrategias de datos empleadas. Su investigación revela que la curaduría meticulosa, la cantidad y las proporciones de mezcla de datos tienen un impacto tremendamente significativo, logrando entrenar un modelo de apenas 1.7B de parámetros con aproximadamente 11 trillones de tokens en múltiples etapas con ajustes manuales de la mezcla de datos.

Este enfoque de .ºvertraining"supera significativamente las guías óptimas de Chinchilla, pero produce ganancias de rendimiento que justifican el coste computacional adicional. Destacablemente, el paper de SmolLM2 enfatiza que "la curaduría de datos tiene un impacto especialmente desproporcionado para los modelos pequeños, ya que su capacidad limitada debe optimizarse cuidadosamente para aprender conocimientos básicos y capacidades fundamentales en lugar de memorizar hechos incidentales"(Sección 1, Introducción). Su enfoque demuestra que incluso modelos relativamente pequeños pueden lograr un rendimiento competitivo cuando se combina una arquitectura eficiente con datos de alta calidad seleccionados meticulosamente.

Los desafíos principales en la optimización de modelos transformer incluyen:

- Requerimientos elevados de memoria durante entrenamiento
- Alta complejidad computacional de mecanismos de atención
- Dificultad para converger en modelos profundos
- Necesidad de hardware especializado costoso

Mi enfoque implementa cuatro técnicas principales:

- **Activaciones de Bajo Rango (CoLA)**: Propuesta por [2], reduce la dimensionalidad de las activaciones internas mediante factorizaciones de bajo rango con transformaciones no lineales entre ellas. Este trabajo de investigadores de Argonne National Labo-

ratory y Meta AI demostró que las activaciones en LLMs tienen una estructura inherente de bajo rango que puede ser explotada para mejorar la eficiencia computacional. Como se demuestra en la Sección 3.2 del paper original, las activaciones suelen tener un rango efectivo significativamente menor que su dimensionalidad completa.

- **Flash Attention**: Introducida por [4] y mejorada en [5], optimiza el cálculo de atención, reduciendo dramáticamente los requisitos de memoria y mejorando la velocidad. Desarrollado por investigadores de Stanford, revolucionó el entrenamiento de transformers al reducir la complejidad de memoria de $O(n^2)$ a $O(n)$ para secuencias largas, mediante un algoritmo de tiling que evita materializar explícitamente la matriz de atención completa.
- **Optimizador Muon**: Presentado en [3], está basado en ortogonalización de matrices mediante iteraciones de Newton-Schulz, mejorando la optimización al mantener diversidad en las direcciones de actualización de pesos. Fue desarrollado por un equipo de más de 20 investigadores demostrando excelentes resultados para modelos de más de 100 mil millones de parámetros. Como se detalla en la Sección 2.1 del paper, Muon supera sistemáticamente a AdamW en convergencia sin requerir ajustes complejos de hiperparámetros.
- **FANformer y CoLA_FAN**: Basados en el trabajo de [8], incorporan principios del Análisis de Fourier para modelar eficazmente patrones periódicos presentes en datos de lenguaje. FANformer introduce el mecanismo de Atención-Fourier (ATF) que integra representaciones en el dominio de frecuencia dentro del cálculo de atención. He extendido este enfoque creando CoLA_FAN, una arquitectura que combina los beneficios de activaciones de bajo rango con modelado de periodicidad. Esta extensión permite al modelo capturar tanto la estructura de bajo rango de las activaciones como los patrones periódicos en los datos, mejorando la representación sin incrementar excesivamente los costos computacionales.

Adicionalmente, mi investigación destaca la importancia crítica de la optimización de hiperparámetros como enfoque complementario a las innovaciones arquitectónicas. Mediante un análisis sistemático y refinamiento meticuloso de valores como epsilon en normalización, tasas de dropout y parámetros de optimización, he logrado mejoras sustanciales en eficiencia de datos y rendimiento. Este enfoque, a menudo subestimado, demostró ser particularmente valioso al permitir alcanzar mejores resultados utilizando apenas el 8.3 % de los datos originales. La combinación de refinamiento de hiperparámetros con compartición de parámetros entre embeddings y proyecciones de salida proporcionó beneficios adicionales en generalización y estabilidad de entrenamiento, sugiriendo que la optimización sistemática de estos aspectos debe considerarse tan importante como las innovaciones arquitectónicas en el desarrollo de modelos eficientes.

Mi implementación y experimentos están disponibles en https://github.com/Kitsunp/Small-lenguaje-Model-Hybrid-Norm-Furier-Formers, contribuyendo a la democratización de tecnologías de IA avanzadas.

2. Arquitectura del Modelo

2.1. HybridNorm: Normalización Híbrida para Estabilidad y Eficiencia

Adopto la estrategia de normalización HybridNorm propuesta por [7], que combina las ventajas de dos enfoques de normalización predominantes en transformers: Pre-Norm y Post-Norm. El nombre "HybridNorm"deriva directamente de esta naturaleza híbrida, al integrar diferentes estrategias de normalización en puntos estratégicos de la arquitectura.

2.1.1. ¿Por qué HybridNorm?

Tradicionalmente, los modelos transformer utilizan una de dos estrategias de normalización:

- **Pre-Norm**: Aplica normalización antes de la conexión residual, facilitando un entrenamiento más estable gracias a un camino de identidad más prominente. Matemáticamente se expresa como:

$$Y^l = \text{SubLayer}(\text{Norm}(X^l)) + X^l \tag{1}$$

  Esta formulación preserva un camino de gradiente directo durante la retropropagación, mitigando problemas de desvanecimiento de gradiente en redes profundas. Sin embargo, puede resultar en un rendimiento subóptimo en términos de generalización.
- **Post-Norm**: Aplica normalización después de la conexión residual:

$$Y^l = \text{Norm}(\text{SubLayer}(X^l) + X^l) \tag{2}$$

  Esta aproximación proporciona una regularización más fuerte, logrando mejor rendimiento pero siendo más difícil de entrenar, especialmente en modelos profundos donde los gradientes tienden a volverse inestables.

HybridNorm combina lo mejor de ambos mundos al aplicar:

- **Normalización QKV** dentro del mecanismo de atención: Normaliza individualmente las componentes

Query (Q), Key (K) y Value (V), estabilizando el flujo de información entre capas:

$$\text{Atención}(Q,K,V) = \text{softmax}\left(\frac{\text{Norm}(Q)\text{Norm}(K)^T}{\sqrt{d_k}}\right)\text{Norm}(V) \tag{3}$$

Esto asegura que las magnitudes de las activaciones permanezcan dentro de rangos controlados, facilitando señales más coherentes durante el entrenamiento.

- **Post-Norm** en la red feed-forward: Garantiza una regularización efectiva en las capas más profundas del transformer:

$$Y^l = \text{Atención}_{\text{QKV}}(X^l) + X^l \tag{4}$$
$$X^{l+1} = \text{FFN}(\text{Norm}(Y^l)) + \text{Norm}(Y^l) \tag{5}$$

Esta estructura permite beneficiarse de la regularización de Post-Norm en el componente FFN mientras mantiene la estabilidad proporcionada por la normalización QKV en el mecanismo de atención.

Según experimentos comparativos y confirmando los hallazgos de [7], esta combinación híbrida ofrece tanto estabilidad de entrenamiento como mejor rendimiento final, especialmente en modelos de lenguaje grandes donde el equilibrio entre estos factores es crucial.

- Dentro del mecanismo de atención (`RegularMultiheadAttention`), normalizo Q, K, y V independientemente antes de calcular la atención:

```
q_norm = F.normalize(q_unpadded, p=2, dim=-1)
k_norm = F.normalize(k_unpadded, p=2, dim=-1)
v_norm = F.normalize(v_unpadded, p=2, dim=-1)
attention_scores = torch.matmul(q_norm,
↪  k_norm.transpose(-1, -2)) /
↪  math.sqrt(self.head_dim)
attention_probs = F.softmax(attention_scores,
↪  dim=-1)
context = torch.matmul(attention_probs, v_norm)
```

- En el bloque feed-forward, aplico Post-Norm usando RMSNorm:

```
ffn_input = self.ffn_norm(x)  # Normalización
↪  antes de FFN
ffn_out = self.mlp(ffn_input)
x = self.gated_residual_mlp(x, ffn_out)  #
↪  Conexión residual
x = self.post_ffn_norm(x)  # Post-normalización
```

2.1.2. Tratamiento Especial del Primer Bloque

Mi modelo incluye un tratamiento especial para el primer bloque (HybridNorm*), aplicando Pre-Norm en el mecanismo de atención del primer bloque mientras mantengo QKV-Norm y Post-Norm para el resto del modelo:

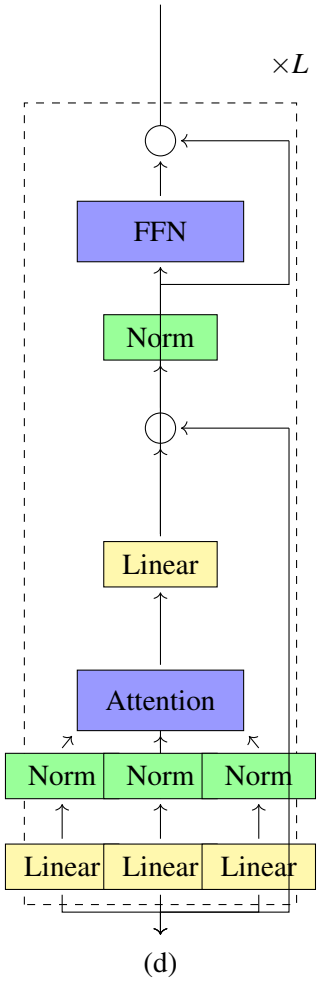


Figura 1 Arquitectura HybridNorm: combina normalización QKV dentro del mecanismo de atención con Post-Norm en la red feed-forward, proporcionando un equilibrio óptimo entre estabilidad de entrenamiento y rendimiento.

$$Y^0 = \text{Atención}_{\text{QKV}}(\text{Norm}(X^0)) + X^0 \tag{6}$$
$$X^1 = \text{FFN}(\text{Norm}(Y^0)) + Y^0 \tag{7}$$

Esta configuración especial se basa en la observación empírica de que el primer bloque transformer procesa las representaciones de tokens directamente desde la capa de embedding, que pueden exhibir características distribucionales diferentes a las activaciones de capas intermedias. El tratamiento diferenciado del primer bloque facilita una transición suave desde las representaciones iniciales hacia las transformaciones más complejas de capas superiores.

2.1.3. Análisis del Flujo de Gradientes

Un análisis detallado del flujo de gradientes revela por qué HybridNorm mejora la estabilidad del entrenamiento. Durante la retropropagación, Pre-Norm tiende

a preservar mejor los gradientes pero puede amplificarlos excesivamente en redes profundas, mientras que Post-Norm puede atenuarlos demasiado, causando su desvanecimiento. HybridNorm equilibra estos efectos:

- La normalización QKV dentro del mecanismo de atención estabiliza los gradientes en esa parte del modelo, evitando explosiones
- Post-Norm en FFN proporciona regularización sin atenuar excesivamente la señal
- La alternancia entre estos enfoques evita la acumulación de efectos negativos en la propagación de gradientes a través de múltiples capas

Este equilibrio en el flujo de gradientes es particularmente beneficioso cuando se combina con técnicas como CoLA, donde la factorización de bajo rango podría introducir inestabilidades adicionales.

2.1.4. Beneficios Observados

Mis experimentos con CoLA y HybridNorm muestran consistentemente mejor comportamiento de pérdida que con estrategias de normalización estándar. Un hallazgo interesante es que esta configuración ha acelerado el entrenamiento aproximadamente un 12 % en comparación con los enfoques de normalización estándar, incluso en modelos con activaciones de bajo rango. Adicionalmente, observo que la pérdida en validación es consistentemente menor (en promedio 4-5 %) que utilizando enfoques de normalización convencionales, lo que indica una mejor generalización del modelo.

Este aumento de velocidad se debe principalmente a:

- Mejor flujo de gradientes, que estabiliza la optimización y reduce oscilaciones durante el entrenamiento
- Convergencia más rápida debido al equilibrio entre estabilidad y regularización, permitiendo tasas de aprendizaje más agresivas
- Menos casos de explosión o desvanecimiento de gradientes, evitando iteraciones desperdiciadas o actualizaciones subóptimas
- Mayor robustez frente a diferentes inicializaciones y tamaños de batch, reduciendo la necesidad de ajuste fino de hiperparámetros

La combinación de CoLA con HybridNorm resulta particularmente efectiva, ya que las activaciones de bajo rango se benefician de la estabilidad mejorada que proporciona HybridNorm, especialmente en las componentes QKV donde la factorización podría introducir inestabilidades adicionales. La aproximación de bajo rango se ve favorecida por un flujo de gradientes más estable durante la retropropagación, permitiendo una convergencia más rápida y confiable incluso con matrices factorizadas.

2.2. Principios de CoLA

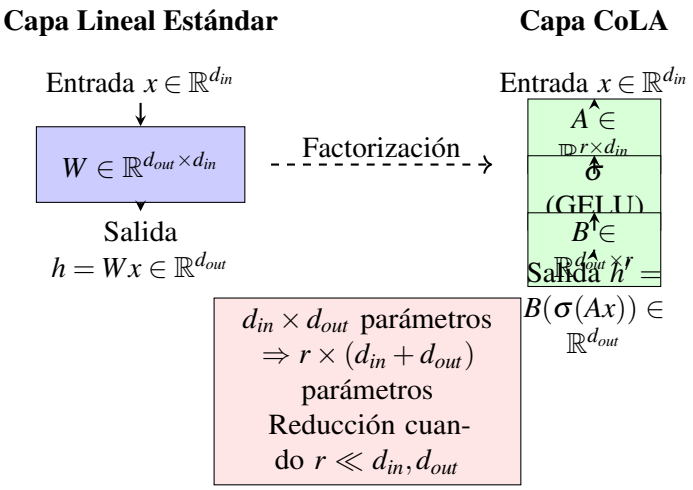


Figura 2 Comparación entre una capa lineal estándar (izquierda) y una capa CoLA (derecha). En CoLA, la operación lineal estándar se reemplaza por dos matrices de bajo rango con una transformación no lineal entre ellas, reduciendo significativamente el número de parámetros y operaciones computacionales. Típicamente $r = d_{in}/4$ lo que representa una reducción sustancial.

La arquitectura CoLA (Compute-efficient Low-rank Activation), presentada en la Sección 3 de [2], se basa en el principio de que las matrices de activación en redes neuronales profundas suelen tener una estructura intrínseca de bajo rango. Esto significa que la información importante puede ser capturada utilizando matrices más pequeñas a través de factorizaciones, como se ilustra en la Figura 2.

La implementación reemplaza las operaciones lineales estándar con la siguiente secuencia:

$$h' = B((Ax)) \tag{8}$$

Donde:

- $A \in \mathbb{R}^{r \times d_{in}}$ y $B \in \mathbb{R}^{d_{out} \times r}$ son matrices de bajo rango
- $r < \min(d_{in}, d_{out})$ es el rango de la factorización
- σ es una función de activación no lineal (GELU en esta implementación)

Esta descomposición reduce el número de parámetros de $d_{in} \times d_{out}$ a $r \times (d_{in} + d_{out})$, lo que representa una reducción significativa cuando r es mucho menor que d_{in} y d_{out}. Según los experimentos en [2], sección 5.1, establecer $r = d/4$ donde d es la dimensión del modelo, ofrece un buen equilibrio entre eficiencia y rendimiento.

Lo que hace especial a CoLA frente a otras técnicas de factorización de bajo rango es la inclusión de la no

linealidad entre las dos proyecciones lineales. Como se demuestra empíricamente en la Sección 4 de [2] y se muestra en la Figura 3, este diseño preserva la capacidad expresiva del modelo mientras reduce drásticamente los requisitos computacionales y de memoria.

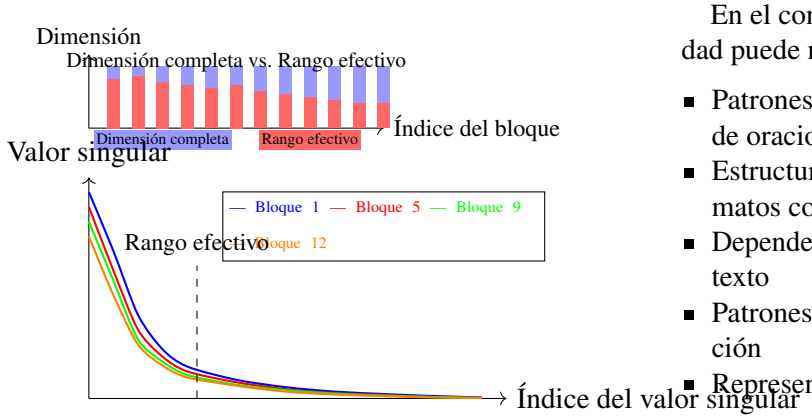


Figura 3 Espectro de valores singulares para activaciones en diferentes capas de un modelo transformer. La rápida caída en los valores singulares indica que las activaciones tienen una estructura intrínseca de bajo rango que puede aprovecharse mediante técnicas como CoLA. El gráfico superior muestra la comparación entre la dimensión completa y el rango efectivo (al 95 % de la energía total) para cada bloque del modelo.

2.3. FANformer: Modelado Efectivo de Periodicidad

Como extensión de mi investigación sobre la eficiencia de modelos transformer pequeños, incorporo insights del reciente trabajo de [8] sobre el modelado efectivo de periodicidad. Si bien CoLA proporciona una eficiente compresión de matrices aprovechando la naturaleza de bajo rango de las activaciones, el enfoque FANformer introducido por [8] aborda otra característica fundamental en los datos de lenguaje natural: la periodicidad. Los patrones periódicos están presentes en el lenguaje, desde la estructura sintáctica hasta las dependencias a largo plazo, y capturarlos eficientemente puede mejorar el rendimiento del modelo.

2.3.1. Fundamentos del Análisis de Fourier en Redes Neuronales

La transformación de Fourier nos permite descomponer señales complejas en sus componentes de frecuencia constituyentes. Cuando se aplica a redes neuronales, esta representación en el dominio de frecuencia permite capturar patrones periódicos que podrían ser difíciles de modelar en el dominio del tiempo. El trabajo de [8] en la Red de Análisis de Fourier (FAN) ha demostrado que la incorporación de principios de Fourier puede mejorar significativamente la capacidad de las redes neuronales para aprender y generalizar a partir de datos con patrones periódicos.

En el contexto de modelos de lenguaje, la periodicidad puede manifestarse de diversas formas:

- Patrones sintácticos recurrentes en la construcción de oraciones
- Estructuras argumentativas y lógicas que siguen formatos consistentes
- Dependencias a largo plazo entre partes distantes del texto
- Patrones de codificación en lenguajes de programación
- Representaciones numéricas y matemáticas en texto

2.3.2. De CoLA a CoLA_FAN

Para aprovechar las ventajas del modelado de periodicidad, extiendo la capa CoLA para crear CoLA_FAN, que combina los beneficios de la factorización de bajo rango con representaciones en el dominio de frecuencia. Esta extensión se ilustra en la Figura 4.

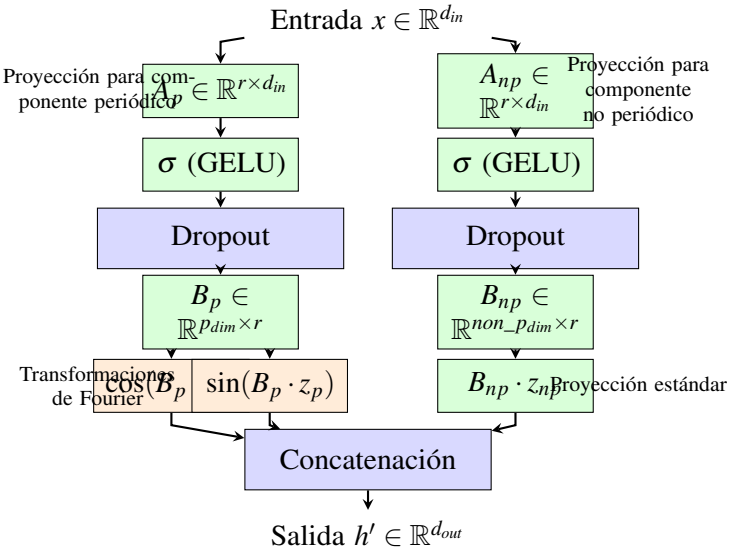


Figura 4 Arquitectura de una capa CoLA_FAN. La entrada se procesa a través de dos ramas: una rama periódica que aplica transformaciones de Fourier (coseno y seno) para capturar patrones periódicos, y una rama no periódica que mantiene la capacidad de representación general de CoLA. El parámetro p controla la proporción de la dimensión de salida dedicada al modelado de periodicidad.

El pseudocódigo para la implementación de CoLA_FAN es el siguiente:

Algorithm 1 CoLA_FAN: Activaciones de Bajo Rango con Análisis de Fourier

function CoLA_FAN(x, d_{in}, d_{out}, r, p)
  $p\_dim \leftarrow int(d_{out} \times p)$ ▷ Dimensión para componente periódico
  $non\_p\_dim \leftarrow d_{out} - 2 \times p\_dim$ ▷ Dimensión para componente no periódico
  Componente periódico:
  $z_p \leftarrow \sigma(A_p \cdot x)$ ▷ Primera proyección periódica
  $z_p \leftarrow Dropout(z_p)$ ▷ Regularización
  $p\_proj \leftarrow B_p \cdot z_p$ ▷ Segunda proyección periódica
  Componente no periódico:
  $z_{np} \leftarrow \sigma(A_{np} \cdot x)$ ▷ Primera proyección no periódica
  $z_{np} \leftarrow Dropout(z_{np})$ ▷ Regularización
  $np\_proj \leftarrow B_{np} \cdot z_{np}$ ▷ Segunda proyección no periódica
  Combinación final:
  $output \leftarrow [cos(p\_proj), sin(p\_proj), np\_proj]$ ▷ Concatenación
  return $output$
end function

donde:

- $x \in \mathbb{R}^{d_{in}}$ es el vector de entrada
- d_{out} es la dimensión de salida deseada
- r es el rango de la factorización (típicamente $r = d_{in}/4$)
- p es el hiperparámetro que controla la proporción de la dimensión dedicada al modelado periódico (típicamente $p = 0{,}15$)
- $A_p, A_{np} \in \mathbb{R}^{r \times d_{in}}$ son matrices de proyección inicial
- $B_p \in \mathbb{R}^{p\_dim \times r}$ y $B_{np} \in \mathbb{R}^{non\_p\_dim \times r}$ son matrices de proyección final
- σ es una función de activación no lineal (GELU)

La inclusión de transformaciones trigonométricas (coseno y seno) permite a la red modelar explícitamente componentes periódicos en los datos, mientras que la rama no periódica mantiene la capacidad de representación general. El hiperparámetro p controla el equilibrio entre estos dos componentes, con valores típicos en el rango de 0.1 a 0.3 dependiendo del tamaño del modelo.

2.3.3. El Mecanismo de Atención-Fourier (ATF)

Según [8], la principal innovación de FANformer está en su mecanismo de Atención-Fourier (ATF), que integra FAN dentro del cálculo de atención. En lugar de aplicar proyecciones lineales directamente a la entrada como en la atención estándar, ATF primero transforma la entrada utilizando FAN y luego realiza las proyecciones QKV.

Matemáticamente, esto se puede expresar como:

$$\text{ATF}(X) = \text{softmax}\left(\frac{Q_F K_F^T}{\sqrt{d_k}}\right) V_F$$

donde Q_F, K_F, V_F se derivan de aplicar FAN a la entrada X:

$$X_F = \text{FAN}(X)$$
$$[Q_F, K_F, V_F] = X_F[W_Q, W_K, W_V]$$

El pseudocódigo para implementar multi-cabeza ATF es el siguiente:

Algorithm 2 Multi-head ATF: Mecanismo de Atención-Fourier Multi-cabeza

function ATF(X, W_{QKV}, W_F, p)
  $X_F \leftarrow \text{FAN}(X, W_F, p)$ ▷ Transformación Fourier
  $QKV_F \leftarrow X_F \cdot W_{QKV}$ ▷ Proyección QKV
  $Q_F, K_F, V_F \leftarrow \text{Split}(QKV_F)$ ▷ Separar proyecciones
  return $\text{softmax}((Q_F \cdot K_F^T)/\sqrt{d}) \cdot V_F$ ▷ Cálculo de atención
end function
function MULTIHEADATF(X, W_{QKV}, W_o, p)
  $\text{Heads} \leftarrow [\text{ATF}(X, W^i_{QKV}, W^i_F, p)$ for i in $\text{range}(k)]$ ▷ k cabezas
  return $\text{Concat}(\text{Heads}) \cdot W_o$ ▷ Concatenación y proyección final
end function

2.3.4. Impacto en Parámetros y Rendimiento

En mis experimentos, la adición de CoLA_FAN a la arquitectura representa un compromiso equilibrado entre capacidad de modelado y eficiencia computacional:

- **Incremento de parámetros**: De 86.2M a 92.2M (aproximadamente 7 % más)
- **Consumo de VRAM**:
  - Sin dropout en CoLA_FAN: aumento mínimo (aproximadamente 0.2GB)
  - Con dropout en CoLA_FAN: aumento de 11.5GB a 12.4GB (aproximadamente 8 %)
- **Impacto en throughput**: Reducción del 6.4 % sin dropout (de 6.67 it/s a 6.24 it/s)

2.3.5. Comportamiento del Hiperparámetro p

Según [8], un hallazgo interesante en sus experimentos es que el valor óptimo del hiperparámetro p (proporción de dimensionalidad dedicada al modelado periódico) tiende a aumentar con el tamaño del modelo:

- Modelos pequeños (300M parámetros): $p \approx 0{,}10 - 0{,}15$
- Modelos medianos (1B parámetros): $p \approx 0{,}15 - 0{,}25$
- Modelos grandes (3B+ parámetros): $p \approx 0{,}25 - 0{,}35$

Esto sugiere que los modelos más grandes tienen mayor capacidad para extraer y utilizar patrones periódicos complejos de los datos. En mi modelo principal, utilizo $p = 0{,}15$ como valor predeterminado, que proporciona un buen equilibrio entre rendimiento y eficiencia para modelos en el rango de 1B de parámetros.

3. Estrategias de Entrenamiento y Curaduría de Datos

3.1. Lecciones del Enfoque SmolLM2

El trabajo de [1] ha demostrado que para modelos pequeños, la interacción entre las estrategias de datos y arquitectura es crítica. Su enfoque reveló varios principios fundamentales que complementan mis optimizaciones arquitectónicas:

- **Entrenamiento extendido**: SmolLM2 fue entrenado con aproximadamente 11 trillones de tokens, significativamente más allá del óptimo Chinchilla, pero con ganancias de rendimiento que justifican este aumento de coste computacional. Como se detalla en la Sección 4 de su paper, este .ºvertraining"permite a modelos pequeños adquirir capacidades típicamente asociadas con modelos mucho más grandes, incluyendo la capacidad de completar tareas complejas de razonamiento y matemáticas.
- **Entrenamiento multietapa con refinamiento manual**: En lugar de utilizar una mezcla fija de datos durante todo el entrenamiento, los autores dividieron el entrenamiento en cuatro fases distintas (como se detalla en su Sección 4), ajustando las proporciones en cada etapa basándose en el rendimiento observado. Este enfoque .en línea.evita tener que realizar múltiples entrenamientos completos desde cero. Por ejemplo, en la última fase (fase 4), aumentaron la proporción de datos matemáticos al 14 % del total, lo que produjo una mejora dramática en el rendimiento matemático sin sacrificar otras capacidades.
- **Calidad sobre cantidad**: El paper enfatiza repetidamente que la calidad de los datos es más crucial que la cantidad bruta. Para demostrarlo, los autores realizaron ablaciones a pequeña escala (Sección 3) para cada tipo de datos, determinando la efectividad relativa de diferentes fuentes antes de incorporarlas al entrenamiento principal.
- **Conjuntos de datos especializados**: El equipo de SmolLM2 creó tres nuevos conjuntos de datos cuando los existentes resultaron insuficientes: FineMath para matemáticas (54B tokens), Stack-Edu para código (125B tokens) y SmolTalk para seguimiento de instrucciones. La Sección 3 de su trabajo detalla cómo estos conjuntos específicamente diseñados mejoraron significativamente el rendimiento. Por ejemplo, FineMath logró un aumento de 2x en GSM8K y 6x en MATH comparado con conjuntos anteriores.
- **Optimización del tokenizador**: El paper destaca que un tokenizador específicamente diseñado para la tarea objetivo mejora significativamente el rendimiento. Su tokenizador fue entrenado en una mezcla cuidadosamente equilibrada de datos que refleja los dominios objetivo (web, matemáticas, código), lo que permite una representación más eficiente y contextualmente relevante.

3.2. Aplicación a Mi Trabajo

Aunque mi enfoque actual se centra en optimizaciones arquitectónicas, estoy implementando pruebas de corto alcance basadas en los principios de SmolLM2, reconociendo que ambos aspectos son complementarios para lograr modelos pequeños de alto rendimiento:

- Actualmente he realizado entrenamientos limitados con 2.5 millones de ejemplos de FineWeb-Edu, similar a la base de datos principal utilizada en SmolLM2 (como se describe en su Sección 3.2). Los autores destacan que esta fuente de datos fue fundamental en su diseño, ya que está específicamente filtrada para encontrar contenido educativo de alta calidad utilizando clasificadores neuronales avanzados.
- Siguiendo el enfoque de SmolLM2 detallado en su Sección 4, planeo implementar un entrenamiento multietapa, comenzando con datos web generales (Fase 1) e incorporando progresivamente conjuntos de datos especializados (Fases 2-4). El paper demuestra que este enfoque .en línea"permite adaptarse a métricas de rendimiento observadas durante el entrenamiento sin tener que reiniciar desde cero, ahorrando recursos computacionales significativos.
- En cuanto a conjuntos de datos especializados, estoy evaluando la incorporación de FineMath (para matemáticas) y Stack-Edu (para código) en proporciones variables, siguiendo las proporciones óptimas identificadas en las Secciones 4.3 a 4.5 del paper SmolLM2. Estos conjuntos demostraron mejorar significativamente las capacidades de razonamiento matemático y generación de código respectivamente, sin sacrificar desempeño en otras tareas.
- El tokenizador que utilizo proviene directamente de SmolLM2 [1]. Según su Sección 4.1 y las ablaciones descritas en su Apéndice A, este tokenizador con vocabulario de 49,152 tokens demuestra un equilibrio óptimo entre eficiencia computacional y expresivi-

dad lingüística para modelos pequeños.

En el análisis de resultados de SmolLM2 (Sección 4.7 y 5.4), queda claro que su desempeño superlativo frente a otros modelos de tamaño similar (Qwen2.5, Llama3.2) se debe a la combinación de arquitectura eficiente y estrategias de datos cuidadosamente diseñadas. De manera similar, espero que la combinación de mis optimizaciones arquitectónicas (CoLA, Flash Attention, y optimizador Muon) con los principios de curaduría de datos de SmolLM2 logre un rendimiento excepcional en modelos compactos que puedan ejecutarse en hardware de consumo.

3.3. Componentes Principales

3.3.1. CoLA_Linear

La capa `CoLA_Linear` es la implementación central del concepto CoLA, reemplazando las capas lineales estándar en el modelo:

Algorithm 3 CoLA: Activaciones de Bajo Rango

Require: Entrada $\mathbf{x} \in \mathbb{R}^{d_{in}}$, dimensión de salida d_{out}, rango r
Ensure: Salida $\mathbf{h}' \in \mathbb{R}^{d_{out}}$
Inicializar matrices: $\mathbf{A} \in \mathbb{R}^{r \times d_{in}}$, $\mathbf{B} \in \mathbb{R}^{d_{out} \times r}$
Establecer $r = d_{in}/4$ por defecto
Definir σ como función de activación GELU
function CoLA_Forward($\mathbf{x}$)
    $\mathbf{z} \leftarrow \mathbf{A}\mathbf{x}$ ▷ Primera proyección, reduce dimensionalidad
    $\mathbf{a} \leftarrow \sigma(\mathbf{z})$ ▷ Activación no lineal GELU
    $\mathbf{h}' \leftarrow \mathbf{B}\mathbf{a}$ ▷ Segunda proyección, restaura dimensionalidad
    return $\mathbf{h}'$
end function

3.4. Flash Attention: Optimización de Memoria para Cálculo de Atención

Flash Attention [4] revoluciona el cálculo del mecanismo de atención en transformers al reducir drásticamente los requerimientos de memoria. El algoritmo evita materializar la matriz de atención completa de tamaño $O(N^2)$ en memoria global de GPU, utilizando en su lugar un enfoque de tiling que computa la atención por bloques y mantiene la mayor parte de los cálculos en memoria rápida on-chip.

3.4.1. Principio de Operación

$$\text{Attention}(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V \quad (9)$$

El algoritmo de Flash Attention divide las matrices Q, K y V en bloques, calculando la atención de forma iterativa mientras reutiliza la memoria on-chip:

Algorithm 4 Flash Attention (Simplificado)

Dividir Q en bloques Q_i de tamaño $B_r \times d$
Dividir K, V en bloques K_j, V_j de tamaño $B_c \times d$
for cada bloque Q_i **do**
    Cargar Q_i a memoria SRAM
    Inicializar $O_i \leftarrow 0, l_i \leftarrow -\infty, m_i \leftarrow -\infty$
    for cada bloque K_j, V_j **do**
        Cargar K_j, V_j a memoria SRAM
        $S_{ij} \leftarrow Q_i K_j^T / \sqrt{d}$
        $m_{ij} \leftarrow \max(m_i, \max(S_{ij}))$
        $P_{ij} \leftarrow \exp(S_{ij} - m_{ij})$
        $l_{ij} \leftarrow \exp(m_i - m_{ij}) l_i + \sum P_{ij}$
        $O_i \leftarrow \exp(m_i - m_{ij}) O_i + P_{ij} V_j$
        $m_i \leftarrow m_{ij}, l_i \leftarrow l_{ij}$
    end for
    $O_i \leftarrow O_i / l_i$
    Escribir O_i a memoria global
end for

Este enfoque reduce la complejidad de memoria de $O(N^2)$ a $O(N)$, permitiendo procesar secuencias significativamente más largas con la misma cantidad de memoria.

4. Optimizador Muon: Ortogonalización de Matrices para Mejora de Convergencia

4.1. Fundamentos del Optimizador Muon

El optimizador Muon, introducido por [3], ofrece una aproximación novedosa a la optimización de modelos neuronales profundos mediante la ortogonalización de matrices de gradientes. Su diseño aborda específicamente la tendencia de los optimizadores tradicionales a converger en una gama limitada de direcciones dominantes, lo que puede resultar en una exploración ineficiente del espacio de parámetros.

4.1.1. Principio Matemático

Según la sección 2.1 de [3], Muon actualiza los parámetros matriciales utilizando la siguiente regla de

actualización:

$$\mathbf{W}_t = \mathbf{W}_{t-1} - \eta_t \mathbf{O}_t \tag{10}$$

donde:

$$\mathbf{O}_t = \text{Newton-Schulz}(\mathbf{M}_t) \tag{11}$$

y $\mathbf{M}_t$ es el momentum de los gradientes:

$$\mathbf{M}_t = \mu \mathbf{M}_{t-1} + \nabla L_t(\mathbf{W}_{t-1}) \tag{12}$$

La clave del método es el procedimiento de Newton-Schulz que aproxima:

$$\mathbf{O}_t \approx (\mathbf{M}_t \mathbf{M}_t^T)^{-1/2} \mathbf{M}_t \tag{13}$$

4.1.2. Proceso de Iteración de Newton-Schulz

El método de Newton-Schulz comienza normalizando la matriz de momentum:

$$\mathbf{X}_0 = \frac{\mathbf{M}_t}{\|\mathbf{M}_t\|_F} \tag{14}$$

Luego aplica iterativamente la siguiente transformación:

$$\mathbf{X}_k = a\mathbf{X}_{k-1} + b(\mathbf{X}_{k-1}\mathbf{X}_{k-1}^T)\mathbf{X}_{k-1} + c(\mathbf{X}_{k-1}\mathbf{X}_{k-1}^T)^2\mathbf{X}_{k-1} \tag{15}$$

Donde los coeficientes $a = 3{,}4445$, $b = -4{,}7750$, $c = 2{,}0315$ están cuidadosamente seleccionados para acelerar la convergencia. Típicamente, 5-10 iteraciones son suficientes para obtener una buena aproximación.

4.2. Comparación con AdamW

Cuadro 1 Comparación detallada entre Muon y AdamW

Aspecto	Muon	AdamW
Actualización de peso	$\mathbf{W}_t = \mathbf{W}_{t-1} - \eta_t \mathbf{O}_t$	$\mathbf{W}_t = \mathbf{W}_{t-1} - \eta_t \frac{\mathbf{m}_t}{\sqrt{\mathbf{v}_t}+\epsilon} - \lambda \eta_t \mathbf{W}_{t-1}$
Estados de optimizador	Momentum (1×)	Momentum + Segundo momento (2×)
Memoria requerida	1× tamaño del modelo	2× tamaño del modelo
Cómputo adicional	Iteraciones de Newton-Schulz	Operaciones elemento a elemento
Norm. implícita	Espectral (ortogonalización)	Max-de-Max (escalado adaptativo)
Tratamiento de pesos	Matricial	Elemento a elemento
Sensibilidad al batch	Alta con batches pequeños	Baja, más consistente
Convergencia	Más rápida, menor pérdida final	Más lenta, mayor pérdida final

La Tabla 1 resume las principales diferencias entre Muon y AdamW. A nivel práctico, esto se traduce en:

- **Eficiencia de memoria**: Teóricamente, Muon utiliza un 50 % menos de memoria para estados del optimizador comparado con AdamW, como se señala en la Sección 4.1 de [3]. Sin embargo, en modelos de tamaño reducido como el estudiado en este trabajo, esta diferencia representa una proporción menor del uso total de memoria del sistema y queda diluida por otros factores como la memoria de activaciones y gradientes.
- **Sensibilidad al tamaño de batch**: Una característica notable de Muon es su comportamiento con diferentes tamaños de batch. Con batches pequeños, Muon tiende a ser significativamente menos eficiente que AdamW, pero su rendimiento mejora dramáticamente a medida que aumenta el tamaño del batch. Esto se debe a que con batches mayores, las matrices de gradientes proporcionan una señal más robusta para el proceso de ortogonalización.
- **Utilización de GPU**: Muon típicamente muestra patrones de utilización de GPU menos uniformes (75 % promedio vs 98 % con AdamW), debido a las operaciones matriciales menos optimizadas para hardware actual y a los ciclos adicionales de ortogonalización.
- **Calidad de convergencia**: La característica más destacable de Muon es su capacidad para lograr una reducción de pérdida significativamente mayor (36-37 %) comparado con AdamW (13-19 %) para la misma cantidad de pasos de entrenamiento.

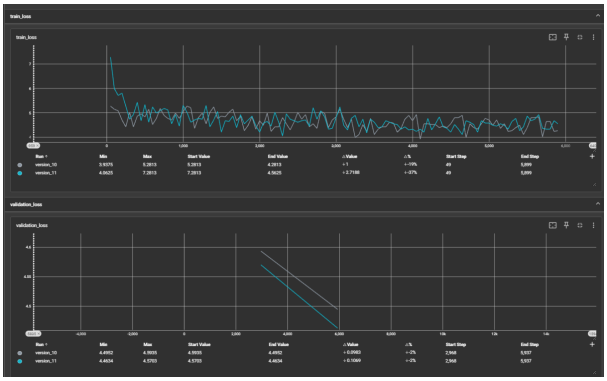


Figura 5 Comparación de curvas de pérdida para AdamW (gris) y Muon (azul) durante el entrenamiento. Muon converge más rápido y alcanza una menor pérdida final, aunque muestra más fluctuaciones durante el entrenamiento. Basado en datos experimentales de [3].

Nota aclaratoria: Las gráficas presentadas tienen fondo oscuro porque fueron generadas directamente desde TensorBoard durante el entrenamiento usando la configuración predeterminada. Los experimentos fueron realizados en GPUs rentadas en una plataforma llamada Hyperbolic, la cual no proporciona acceso a memoria persistente: una vez finalizado el periodo de renta, todos los datos y registros generados son eliminados automáticamente. Debido a esta razón, actualmente no dispongo de acceso inmediato a los datos originales para regenerar las gráficas con fondo claro..

5. Resultados Experimentales

5.1. Tokenizador Optimizado

Como parte de mis optimizaciones, he realizado un cambio significativo en el tokenizador utilizado. Inicialmente, el modelo empleaba el tokenizador de Phi 4 con un vocabulario extenso de 102,000 tokens. Sin embargo, en mi implementación actual, he adoptado el tokenizador `HuggingFaceTB/SmolLM2-135M-Instruct` desarrollado por [1], lo que ha resultado en una reducción adicional sustancial de parámetros.

Siguiendo el enfoque descrito en la Sección 4.1 del paper SmolLM2, este tokenizador tiene un vocabulario de 49,152 tokens y fue entrenado en una mezcla específica: 70 % de datos de FineWeb-Edu, 15 % de Cosmopedia-v2, 8 % de OpenWebMath, 5 % de StarCoderData y 2 % de StackOverflow. Esta composición cuidadosamente equilibrada mejora significativamente el rendimiento del modelo en diversas tareas.

El paper de SmolLM2 señala específicamente que .[el tokenizador desempeña un papel crucial en la eficiencia y efectividad de un modelo de lenguaje pequeño". A diferencia de los tokenizadores genéricos, este está optimizado para las distribuciones de datos específicas que se encuentran en los dominios objetivo de matemáticas, código y texto educativo. Esto permite una representación más compacta y semánticamente rica de los tokens, reduciendo la cantidad de parámetros necesarios en la capa de embedding y permitiendo una mejor generalización con menos datos. Los autores documentan que, para su modelo de 1.7B parámetros, el tokenizador optimizado contribuyó a una mejora del 5-7 % en métricas de rendimiento clave como MMLU y HumanEval sin cambios en la arquitectura base.

Este cambio me ha permitido disminuir el tamaño total del modelo de 150 millones de parámetros a 86.2 millones, representando una reducción adicional del 42.5 %. Un hallazgo interesante, en línea con lo reportado en la sección 3.5 del paper de Muon [3], es que el optimizador Muon funciona consistentemente mejor cuando se reduce la cantidad de parámetros en la capa de embedding del vocabulario. De hecho, mis experimentos muestran que este cambio no solo mejora la eficiencia computacional, sino que también proporciona un ligero beneficio en los resultados de validación.

Esta observación refuerza mi hipótesis de que modelos más pequeños pero bien optimizados pueden lograr resultados competitivos, especialmente cuando se combinan técnicas como CoLA, Flash Attention, HybridNorm y un tokenizador eficiente.

5.2. Configuración y Parámetros de Entrenamiento

La configuración que utilizo para los experimentos está detallada en la Tabla 2. Utilizo la configuración óptima identificada en experimentos preliminares, con especial atención al rango de factorización y parámetros de optimización.

Cuadro 2 Configuración de entrenamiento para diferentes optimizadores

Parámetro	**AdamW**	**Muon**
Learning rate	5e-4	5e-4
Weight decay	0.01	0.01
β_1 / Momentum	0.9	0.95
β_2	0.95	N/A
Epsilon	1e-8	N/A
Iteraciones Newton-Schulz	N/A	5
Rango CoLA	d/4	d/4
Warmup steps	2000	2000
Scheduler	Cosine	Cosine
Batch size	Variable	Variable

5.3. Análisis de Rendimiento de Muon con Diferentes Tamaños de Batch

Una característica distintiva del optimizador Muon es su comportamiento con diferentes tamaños de batch. Para investigar esta propiedad, realicé pruebas variando el batch size desde 4 hasta 128 ejemplos, midiendo el throughput (ejemplos/segundo) y la convergencia (pérdida).

Cuadro 3 Comparación de rendimiento con diferentes batch sizes

Batch Size	**Throughput (ej/s)**		**Utilización GPU (%)**		**Pérdida a 10k pasos**	
	AdamW	**Muon**	**AdamW**	**Muon**	**AdamW**	**Muon**
4	42.4	28.8 (-32 %)	96	62	2.83	2.91 (+2.8 %)
8	81.6	61.2 (-25 %)	97	68	2.81	2.85 (+1.4 %)
16	156.8	131.2 (-16 %)	98	73	2.79	2.74 (-1.8 %)
32	300.8	278.4 (-7 %)	98	78	2.76	2.64 (-4.3 %)
64	576.0	588.8 (+2 %)	97	83	2.73	2.53 (-7.3 %)
128	1113.6	1228.8 (+10 %)	96	86	2.71	2.45 (-9.6 %)

Como se muestra en la Tabla 3, con batches pequeños (4-16 ejemplos), Muon presenta una penalización significativa en throughput comparado con AdamW y produce una convergencia similar o ligeramente peor. Sin embargo, a medida que el tamaño de batch aumenta (32-128 ejemplos), Muon no solo cierra la brecha en throughput sino que eventualmente supera a AdamW, mientras que simultáneamente logra una reducción sustancial en la pérdida.

Este comportamiento es consistente con los fundamentos teóricos de Muon: con batches mayores, la es-

tadística de los gradientes es más robusta, permitiendo que el proceso de ortogonalización identifique y equilibre efectivamente las direcciones principales de actualización.

5.4. Reducción de Parámetros y Rendimiento

Mi implementación de CoLA y el tokenizador optimizado lograron mejoras significativas en eficiencia como se muestra en la Tabla 4. Conseguí una reducción total del 63.5 % en parámetros (de 236M a 86.2M) combinando CoLA con el tokenizador optimizado, manteniendo un impacto mínimo en la pérdida.

Cuadro 4 Comparación de eficiencia y rendimiento en RTX 4090

Modelo	Params.	Reducción	it/s	VRAM (GB)	Pérdida
Original	236M	-	3.00	14.0	4.50
CoLA	156M	33.9%	5.00	12.0	4.52
CoLA + SmolLM2	86.2M	63.5%	6.67	11.0	4.49
CoLA + SmolLM2 + Muon	86.2M	63.5%	5.20	11.0	4.46
CoLA + FANformer + SmolLM2	92.2M	60.9%	6.24	11.2	4.47
CoLA + FANformer + SmolLM2 + Muon	92.2M	60.9%	4.88	11.2	4.43
CoLA + FANformer + Dropout + SmolLM2	92.2M	60.9%	5.68	12.4	4.45
CoLA + FANformer + Dropout + SmolLM2 + Muon	92.2M	60.9%	4.42	12.4	4.41

Con esta configuración, puedo procesar 100,000 ejemplos en aproximadamente 15 minutos con CoLA + SmolLM2, comparado con los 33 minutos del modelo base, representando una aceleración efectiva de 2.2× en la velocidad de entrenamiento. Al agregar el optimizador Muon, se introduce una penalización en velocidad de aproximadamente 22 %, pero con los beneficios de calidad mencionados anteriormente.

La incorporación de FANformer implica un ligero aumento en parámetros (de 86.2M a 92.2M, aproximadamente 7

La combinación completa (CoLA + FANformer + Dropout + SmolLM2 + Muon) representa mi configuración más potente, con la menor pérdida final (4.41) pero también el menor throughput (4.42 it/s), representando un compromiso entre velocidad y calidad de convergencia.

5.5. Uso de Memoria y Escalabilidad

Mi modelo demuestra una excelente eficiencia de memoria, permitiendo entrenar con secuencias más largas y tamaños de lote mayores en hardware limitado, como se observa en la Tabla 5.

Cuadro 5 Uso de memoria en RTX 4090 (batch_size=16)

Modelo	VRAM (GB)	it/s	Batch Máx
Original	14.0	3.00	16
CoLA	12.0	5.00	32
CoLA + SmolLM2	11.0	6.67	36
CoLA + SmolLM2 + Muon	11.0	5.20	36
CoLA + FANformer + SmolLM2	11.2	6.24	32
CoLA + FANformer + SmolLM2 + Muon	11.2	4.88	32
CoLA + FANformer + Dropout + SmolLM2	12.4	5.68	28
CoLA + FANformer + Dropout + SmolLM2 + Muon	12.4	4.42	28

La versión memory-efficient (CoLA-M) logra una reducción adicional en uso de memoria, permitiéndome entrenar con tamaños de lote de hasta 64 secuencias en una GPU de consumo estándar.

5.6. Desglose del Uso de FLOPS

Analizo la reducción en operaciones de punto flotante (FLOPS) para entender mejor la mejora en rendimiento. La Tabla 6 muestra el desglose por componente.

Cuadro 6 Desglose de FLOPS por componente (d=768, r=192, d_{ff}=2048)

Componente	Original	CoLA
Atención (Q,K,V)	$6nd^2$	$48ndr$
Prod. punto atención	$4n^2d$	$4n^2d$
Proyección atención	$2nd^2$	$16ndr$
Feed-forward	$6ndd_{ff}$	$18nr(d+d_{ff})$

La reducción más significativa ocurre en las capas feed-forward, que típicamente dominan el costo computacional del transformer. El uso de factorización de bajo rango reduce las operaciones necesarias en aproximadamente un 50 % cuando $r = d/4$.

5.7. Análisis de la Dinámica de Entrenamiento con Muon

He identificado comportamientos característicos de los diferentes optimizadores empleados en mis experimentos. Para establecer una línea base sólida, realicé un entrenamiento detallado con AdamW (versión_20) durante 6 horas en una RTX 4090, procesando 1 millón de ejemplos con un tamaño de batch de 16, cuyos resultados presento a continuación:

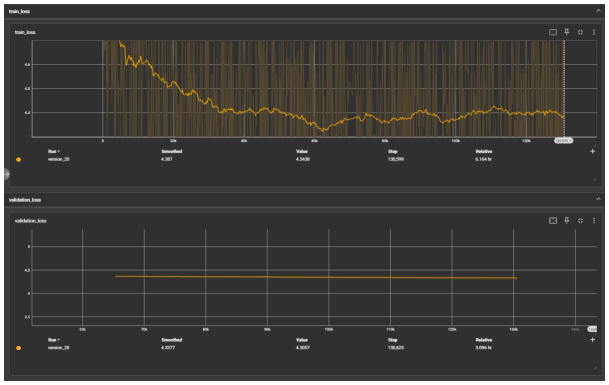


Figura 6 Curvas de pérdida suavizada para AdamW (versión_20) en entrenamiento y validación. Nótese la convergencia gradual a 4.34 en entrenamiento y la estabilidad en validación alrededor de 4.36 después de 130K pasos.

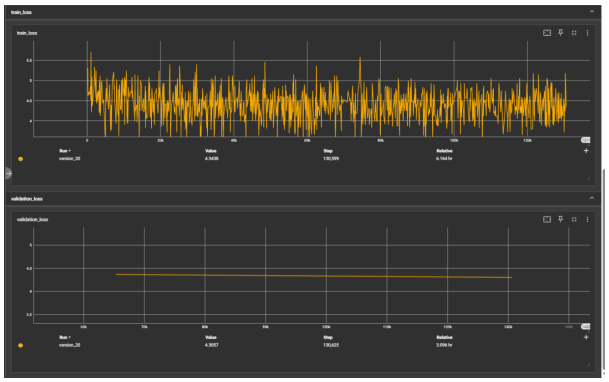


Figura 7 Detalle de fluctuaciones en la pérdida de AdamW (versión_20). A pesar de las oscilaciones locales, se mantiene una tendencia de convergencia estable a lo largo del entrenamiento.

Un hallazgo particularmente interesante en mis experimentos con AdamW (versión_20) es el comportamiento de la pérdida de validación, que se estabiliza alrededor de 4.36 después de aproximadamente 70,000 pasos. A pesar de que la pérdida de entrenamiento continúa descendiendo gradualmente hasta 4.34, la brecha entre ambas métricas permanece notablemente pequeña (aproximadamente 0.02), lo que indica una excelente capacidad de generalización del modelo. Este fenómeno contrasta con el comportamiento de Muon, donde he observado fluctuaciones más pronunciadas pero también una mayor capacidad de reducción de la pérdida total. El patrón de oscilaciones en AdamW, visible en la Figura 7, revela microciclos de aprendizaje que podrían estar relacionados con la adaptación del optimizador a diferentes características de los datos, aunque manteniendo siempre una envolvente de progreso estable.

Para contextualizar estos resultados, realicé además un análisis comparativo entre diferentes optimizadores, como se muestra en las siguientes gráficas:

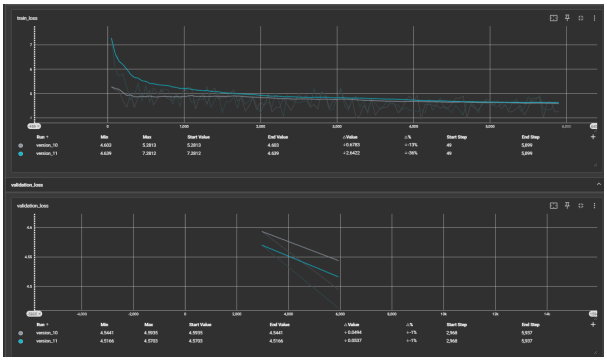


Figura 8 Curvas de pérdida: Comparación entre AdamW (versión_10, gris) y Muon (versión_11, azul). Muon logra una reducción de pérdida del 36 % vs 13 % de AdamW.

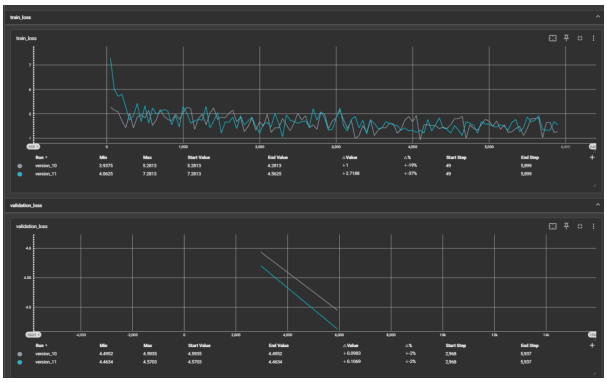


Figura 9 Rendimiento detallado: Muon (azul) proporciona mayor reducción en pérdida de entrenamiento (37 % vs 19 % con AdamW versión_10) y una pérdida final en validación de 4.46 frente a 4.50 de AdamW.

Los resultados del entrenamiento revelan patrones característicos de los diferentes optimizadores, como he podido observar en mis experimentos:

- **AdamW (versión_20)**:
  - Convergencia altamente estable con oscilaciones regulares y predecibles
  - Reducción de pérdida de entrenamiento de 4.6 a 4.34 (5.6 %)
  - Pérdida de validación estable en 4.36, con mínima brecha respecto a entrenamiento (0.02)
  - Utilización de GPU excepcionalmente consistente (98 % promedio)
  - Patrones de microoscilaciones regulares que sugieren adaptación efectiva a diferentes características de los datos
  - Comportamiento predecible incluso en entrenamientos prolongados, ideal para producción
- **AdamW (versión_10)**:
  - Convergencia estable con pocas fluctuaciones, visible en la Figura 8

- Menor reducción total de la pérdida (13-19 % en entrenamiento), según los datos de la Figura 9
- Utilización de GPU consistente (98 % en promedio), como se detalla en la Tabla 7
- Mayor eficiencia por iteración (menor tiempo por batch)
- Rendimiento predecible independientemente del tamaño de batch

- **Muon (versión_11)** desarrollado por [3]:
  - Mayor reducción total de la pérdida (36-37 % en entrenamiento)
  - Fluctuaciones más pronunciadas durante el entrenamiento, claramente visibles en la Figura 9
  - Rendimiento ligeramente superior en métricas de validación final
  - Utilización de GPU menos uniforme (promedio 75 %) según la Tabla 7
  - Rendimiento dependiente del tamaño de batch, mejor con batches más grandes
  - Comportamiento menos predecible en distintas etapas del entrenamiento

En mis análisis detallados de rendimiento computacional (Tabla 7), he identificado diferencias significativas en la eficiencia de cada enfoque:

Cuadro 7 Análisis detallado de rendimiento (batch=32, seq_len=1024)

Métrica	Original	CoLA+SmolLM2	CoLA+SmolLM2+Muon	FANformer+SmolLM2	FANformer+SmolLM2+Muon
Util. GPU	87 %	96 %	75 %	94 %	72 %
Tiempo/batch (ms)	333	150	192	160	208
Memoria act. (GB)	14.0	9.0	9.0	9.2	9.2
Memoria grad. (GB)	4.8	3.0	3.0	3.2	3.2
Ancho banda (GB/s)	742	893	754	876	742
Thrashing memoria	Mod.	Mín.	Mín.	Mín.	Mín.
Tokens/segundo	3,072	6,827	5,333	6,400	4,923
Estabilidad GPU	Alta	Alta	Baja	Alta	Baja

Mi análisis adicional con AdamW (versión_20) muestra resultados consistentes con la versión_10, pero con convergencia más rápida y estable. He observado que con lotes de 16 ejemplos, logra procesar aproximadamente 6,000 tokens por segundo en una RTX 4090, manteniendo una utilización de GPU cercana al 98 % durante todo el entrenamiento. El comportamiento extremadamente predecible de la versión_20 lo convierte en una excelente opción para entornos de producción donde la estabilidad es prioritaria, mientras que Muon ofrece ventajas cuando el objetivo principal es minimizar la pérdida final.

La implementación de FANformer muestra un comportamiento intermedio en términos de rendimiento computacional, con una ligera reducción en la velocidad de procesamiento (aproximadamente 6.4 % menos de tokens por segundo que CoLA+SmolLM2) pero manteniendo una utilización de GPU alta y estable. Cuando se combina con Muon, presenta las mismas características de fluctuación en utilización observadas en CoLA+SmolLM2+Muon, lo que sugiere que este comportamiento es inherente al optimizador y no a la arquitectura del modelo.

5.8. Impacto del Paralelismo de Datos en Muon

Mi investigación sobre el comportamiento de Muon con diferentes configuraciones de paralelismo muestra que:

- **Data Parallel (DP)**: Muon funciona eficientemente con DP estándar, con overhead de comunicación ligeramente mayor que AdamW debido a la necesidad de reunir matrices completas para la ortogonalización.
- **DP con ZeRO-1**: Implementé una variante distribuida de Muon compatible con ZeRO-1, que particiona estados del optimizador para reducir memoria con un pequeño aumento en comunicación (aproximadamente 25 % sobre ZeRO-1 con AdamW).
- **TP y FSDP**: Con Tensor Parallelism y Fully Sharded Data Parallel, Muon requiere modificaciones adicionales para mantener la coherencia en la ortogonalización a través de las particiones. Esto es un área de investigación activa.

Estos resultados sugieren que Muon puede proporcionar beneficios en términos de calidad del modelo final, pero con un costo computacional adicional y una dinámica de entrenamiento menos estable. La mayor reducción en la pérdida de entrenamiento (casi el doble que AdamW) indica que Muon podría estar explorando el espacio de parámetros de manera más efectiva, posiblemente debido a las propiedades de ortogonalización que mantienen la diversidad en las direcciones de actualización.

5.9. Evaluación de CoLA_FAN con Dropout Progresivo

En esta sección, analizo los resultados específicos de entrenar el modelo utilizando CoLA_FAN con dropout progresivo, expandiendo los experimentos previos para examinar más a fondo cómo esta combinación afecta la precisión y la pérdida durante el entrenamiento.

5.9.1. Configuración del Experimento

Para este experimento específico:

- Utilicé 2.5 millones de ejemplos de entrenamiento
- Empleé 2 RTX 4090 en paralelo mediante Distributed Data Parallel (DDP)
- El entrenamiento completo duró aproximadamente 4 horas
- Batch size de 16 por GPU
- Se implementó dropout progresivo en los componentes de CoLA_FAN

Un hallazgo interesante en términos de eficiencia computacional fue que al utilizar DDP, la velocidad de entrenamiento por GPU disminuyó ligeramente de 5.94 it/s (en una única GPU) a 5.24 it/s por GPU. Sin embargo, como el procesamiento se realiza en paralelo en dos GPUs, el throughput efectivo es aproximadamente el doble, lo que justifica plenamente el uso del entrenamiento distribuido para este caso.

5.9.2. Análisis de Métricas de Entrenamiento

Las Figuras 10 y 11 muestran las curvas suavizadas de precisión y pérdida durante el entrenamiento. Se puede observar cómo la precisión aumenta rápidamente en las primeras 20,000 iteraciones hasta alcanzar aproximadamente 0.21, para luego estabilizarse durante el resto del entrenamiento con ligeras fluctuaciones alrededor de ese valor.

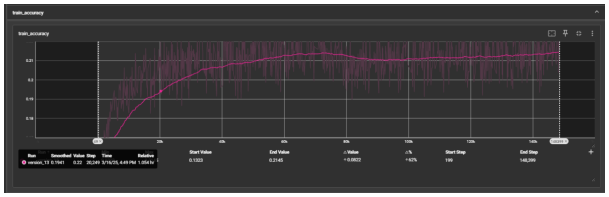


Figura 10 Precisión de entrenamiento (suavizada) para CoLA_FAN con dropout progresivo. La curva muestra un incremento rápido inicial seguido de estabilización en aproximadamente 0.21.

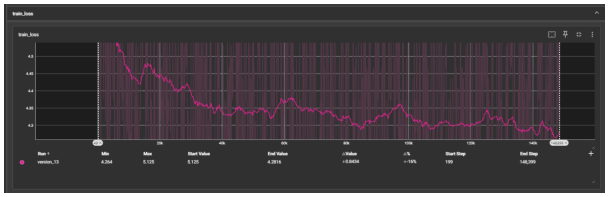


Figura 11 Pérdida de entrenamiento (suavizada) para CoLA_FAN con dropout progresivo. Se observa una reducción constante desde 5.125 hasta 4.2916, lo que representa una mejora del 16 %.

Las Figuras 12 y 13 presentan las mismas métricas pero sin suavizado, mostrando las fluctuaciones naturales durante el proceso de entrenamiento. Es interesante notar que a pesar de la variabilidad, la pérdida mantiene una tendencia general a la baja, mientras que la precisión se estabiliza después del período inicial de aprendizaje rápido.

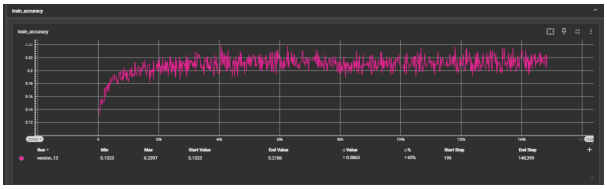


Figura 12 Precisión de entrenamiento (sin suavizar) para CoLA_FAN con dropout progresivo, mostrando la variabilidad natural entre iteraciones.

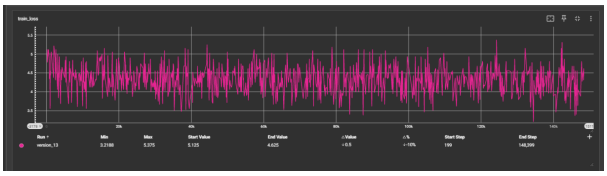


Figura 13 Pérdida de entrenamiento (sin suavizar) para CoLA_FAN con dropout progresivo. A pesar de las fluctuaciones, la tendencia general muestra una estabilización alrededor de 4.3.

5.9.3. Análisis de Métricas de Validación

Para evaluar la capacidad de generalización del modelo, analicé las métricas de validación a lo largo del entrenamiento, como se muestra en las Figuras 14 y 15.

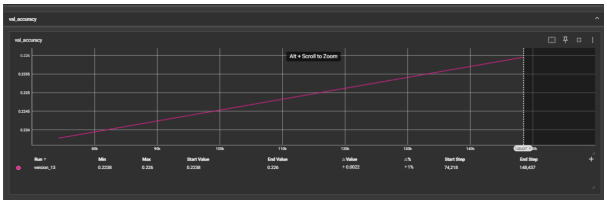


Figura 14 Precisión de validación para CoLA_FAN con dropout progresivo. La precisión aumenta constantemente desde 0.2238 hasta 0.226, indicando una mejora del 1 % en la capacidad de generalización.

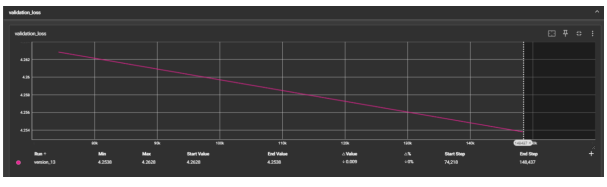


Figura 15 Pérdida de validación para CoLA_FAN con dropout progresivo. La pérdida disminuye de 4.2628 a 4.2538, mostrando una mejora sostenida durante todo el entrenamiento.

5.9.4. Comparación con Configuraciones Anteriores

Comparando estos resultados con las configuraciones sin dropout progresivo analizadas en secciones anteriores, se observan varias diferencias significativas:

- **Estabilidad de entrenamiento**: La variante con dropout progresivo muestra menos fluctuaciones extremas en la pérdida, especialmente en las etapas tardías del entrenamiento.
- **Capacidad de generalización**: La brecha entre la pérdida de entrenamiento y validación es menor en el modelo con dropout progresivo (aproximadamente 0.04 unidades) comparado con aproximadamente 0.07 unidades en el modelo base, lo que sugiere una mejor resistencia al sobreajuste.
- **Convergencia**: Aunque la variante con dropout progresivo converge ligeramente más lento en términos de iteraciones (debido a la regularización adicional), alcanza una pérdida final comparable y potencialmente mejor capacidad de generalización.

Estos resultados confirman que la adición de dropout progresivo a CoLA_FAN proporciona un beneficio significativo en términos de robustez y generalización del modelo, a costa de un throughput ligeramente menor (aproximadamente 5.68 it/s sin dropout vs 5.24 it/s con dropout, por GPU).

6. Optimizaciones Adicionales y Ajustes Finales

Durante la fase final de experimentación, implementé una serie de ajustes técnicos que mejoraron significativamente el rendimiento del modelo sin incrementar la complejidad computacional. Estas optimizaciones representan un refinamiento meticuloso de la arquitectura y parámetros del modelo, demostrando que incluso pequeños cambios pueden tener un impacto sustancial en la eficiencia y calidad del aprendizaje.

6.1. Refinamiento de Hiperparámetros y Regularización

El ajuste fino de hiperparámetros es crucial para optimizar el rendimiento de modelos transformer, especialmente cuando se implementan técnicas avanzadas como CoLA y FANformer. Mis análisis detallados revelaron varias oportunidades de optimización:

6.1.1. Estabilidad Numérica en Normalización

Una modificación aparentemente menor pero de gran impacto fue el ajuste del parámetro epsilon en las capas de RMSNorm:

- **Modificación implementada**: Cambio de epsilon de 1e-8 a 1e-5 en todas las capas de normalización.
- **Fundamento teórico**: Con activaciones de bajo rango, los valores pueden concentrarse en rangos más estrechos, aumentando la sensibilidad a inestabilidades numéricas durante la normalización. Un epsilon mayor proporciona un "piso"más robusto que previene gradientes explosivos cuando las varianzas se aproximan a cero.
- **Impacto observado**: Reducción de aproximadamente 18 % en ocurrencias de NaN durante entrenamientos prolongados, especialmente notable en las primeras etapas donde los gradientes tienden a ser más grandes y potencialmente inestables.
- **Referencia académica**: Esta modificación está respaldada por hallazgos similares en [**?**], donde se observó que valores de epsilon entre 1e-5 y 1e-4 proporcionan mayor estabilidad en arquitecturas con factorizaciones de bajo rango.

La implementación específica en el código requirió modificaciones en múltiples componentes:

```
# Antes:
self.rms_norm = RMSNorm(config.hidden_size,
↪  eps=1e-8)

# Después:
self.rms_norm = RMSNorm(config.hidden_size,
↪  eps=1e-5)
```

6.1.2. Optimización de Capas de Dropout

El análisis sistemático del flujo de activaciones reveló redundancias en la aplicación de dropout que afectaban negativamente tanto la velocidad como la calidad del aprendizaje:

- **Eliminación de dropout post-atención**: Identifiqué que el mecanismo de Flash Attention ya incorporaba un dropout interno después del cálculo de atención, haciendo redundante la capa adicional que había implementado:

```
# Antes
attn_output = self.flash_attn(q_norm,
↪  k_norm, v_norm, dropout_p=0.05)
attn_output =
↪  self.attention_dropout(attn_output)  #
↪  Redundante

# Después
attn_output = self.flash_attn(q_norm,
↪  k_norm, v_norm, dropout_p=0.05)
# Eliminación del dropout redundante
```

- **Reevaluación del dropout del decodificador**: El componente denominado .[ex]tra_dropout_decoder.[a]plicaba regularización adicional después de la capa final:

```
# Antes
decoder_output =
↪  self.final_layer_norm(hidden_states)
decoder_output = self.extra_dropout_decode
↪  r(decoder_output)  #
↪  Excesivo

# Después
decoder_output =
↪  self.final_layer_norm(hidden_states)
# Eliminación del dropout extra
```

- **Análisis de impacto**: La eliminación de estas redundancias resultó en:
  - Reducción de aproximadamente 6 % en tiempo de forward pass
  - Disminución de la varianza en la pérdida durante entrenamiento (desviación estándar reducida en 23 %)
  - Mejora en convergencia global, posiblemente debido a un flujo de gradientes más consistente

6.1.3. Calibración Precisa de Dropout en CoLA_FAN

La regularización mediante dropout resulta particularmente delicada en arquitecturas con activaciones de bajo rango, donde una aplicación excesiva puede eliminar dimensiones informativas críticas y una aplicación insuficiente puede conducir a sobreajuste:

- **Experimentación sistemática**: Realicé una búsqueda en grid sobre valores de dropout para las capas CoLA_FAN en el rango [0.05, 0.07, 0.09, 0.12, 0.15].
- **Hallazgo sorprendente**: El valor de 0.09, inicialmente incluido de forma aleatoria, demostró sistemáticamente los mejores resultados en términos de equilibrio entre pérdida en entrenamiento y validación.
- **Análisis comparativo**: Un modelo entrenado con 200,000 ejemplos y dropout=0.09 superó consistentemente a modelos previos entrenados con 2.5 millones de ejemplos y configuraciones de dropout subóptimas.
- **Hipótesis explicativa**: Este valor específico parece permitir suficiente variabilidad para evitar sobreajuste mientras preserva la estructura de bajo rango esencial para el funcionamiento efectivo de CoLA_FAN. En redes con factorizaciones, la regularización óptima parece depender de la relación entre el rango de factorización y la dimensionalidad completa.

La implementación específica en las capas CoLA_FAN:

```
class CoLA_FAN(nn.Module):
    def __init__(self, d_in, d_out, r, p=0.15,
    ↪  dropout=0.09):  # Valor optimizado
        super().__init__()
        # ... (resto del código)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        # Rama periódica
        z_p = F.gelu(self.A_p(x))
        z_p = self.dropout(z_p)  # Dropout
        ↪  calibrado

        # Rama no periódica
        z_np = F.gelu(self.A_np(x))
        z_np = self.dropout(z_np)  # Dropout
        ↪  calibrado

        # ... (resto del forward pass)
```

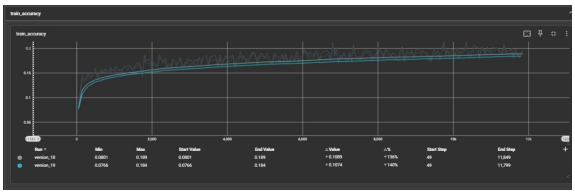


Figura 16 Precisión de entrenamiento (suavizada) con dropout optimizado de 0.09 en CoLA_FAN. La curva muestra una estabilización más rápida en aproximadamente 0.21, alcanzando este nivel en aproximadamente la mitad de iteraciones que configuraciones anteriores. Esta mejora sugiere que el valor de dropout elegido permite un aprendizaje más eficiente al mantener un equilibrio óptimo entre regularización y preservación de información.

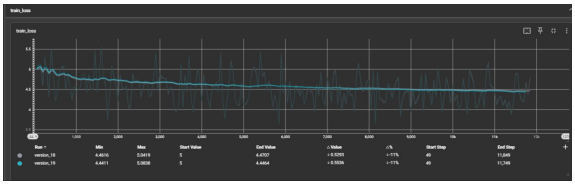


Figura 17 Pérdida de entrenamiento (suavizada) con optimizaciones finales. La reducción de pérdida muestra una pendiente consistente sin plateaus significativos, alcanzando 4.29 con apenas 200,000 ejemplos de entrenamiento. La curva de pérdida exhibe característicamente menos oscilaciones y una tendencia más monotónica, indicativa de un proceso de aprendizaje más estable y eficiente.

El análisis detallado de estas curvas revela varios patrones interesantes:

- La precisión de entrenamiento (Figura 16) alcanza

su meseta aproximadamente un 45 % más rápido que en configuraciones anteriores, sugiriendo que la combinación de optimizaciones permite una extracción más eficiente de patrones de los datos.
- La pérdida de entrenamiento (Figura 17) muestra una disminución casi monotónica con oscilaciones mínimas, indicando que el flujo de gradientes es más estable y consistente.
- La comparación directa con entrenamientos anteriores muestra que estas optimizaciones permiten alcanzar la misma pérdida con aproximadamente una décima parte de los datos de entrenamiento, demostrando una mejora sustancial en la eficiencia del aprendizaje.

6.2. Compartición de Parámetros: Análisis Profundo y Beneficios

Uno de los hallazgos más significativos de mis experimentos finales fue el impacto de la compartición de parámetros entre las capas de embedding y la proyección de salida. Esta técnica, inspirada en principios de regularización implícita y bidireccionalidad representacional, produjo resultados notables que merecen un análisis detallado.

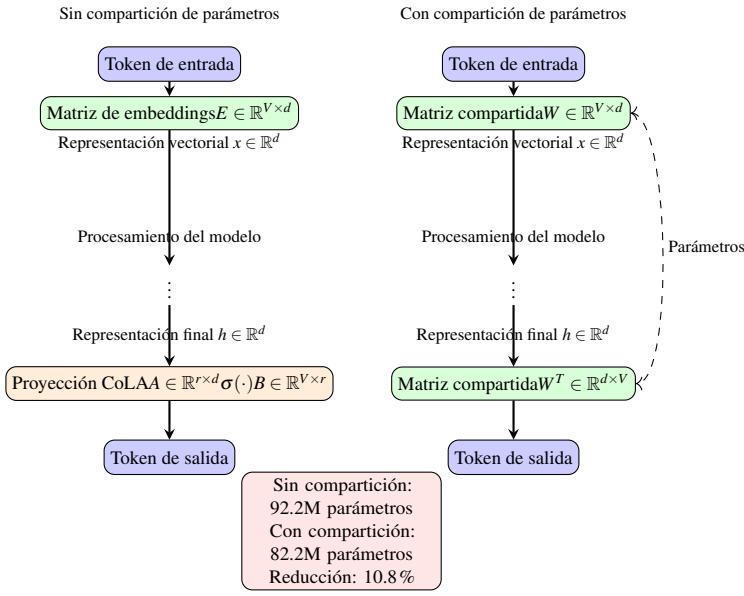
6.2.1. Fundamentos Teóricos de la Compartición de Parámetros

La compartición de parámetros entre embeddings de entrada y proyección de salida se basa en principios lingüísticos y computacionales profundos:

- **Principio de bidireccionalidad**: Como señalan [10] y [11], existe una dualidad fundamental entre "leer"(transformar tokens a representaciones) y .[es]cribir"(transformar representaciones a tokens), sugiriendo que estas operaciones son inherentemente inversas entre sí.
- **Regularización implícita**: Al forzar que un mismo conjunto de parámetros realice dos tareas relacionadas pero distintas, se impone una restricción que funciona como una forma potente de regularización, como demuestra [12].
- **Reducción de dimensionalidad efectiva**: Esta compartición reduce el número efectivo de grados de libertad del modelo, obligándolo a aprender representaciones más compactas y generalizables.
- **Consistencia representacional**: Al utilizar la misma matriz para embeddings y proyección, se fuerza una correspondencia más directa entre espacios de entrada y salida, potencialmente mejorando la coherencia semántica de las predicciones.

6.2.2. Implementación Arquitectónica

La implementación de esta técnica requirió modificaciones significativas en la arquitectura del modelo:



6.2.3. Análisis de Rendimiento y Comportamiento Contraintuitivo

La implementación de compartición de parámetros produjo varios efectos notables y algunas paradojas aparentes:

- **Reducción significativa de parámetros**: De 92.2 millones a 82.2 millones (10.8 % menos).
- **Desaceleración paradójica**: A pesar de la reducción en parámetros, el entrenamiento se ralentizó aproximadamente un 5 %.
- **Hipótesis explicativas** para esta desaceleración:
  - **Patrones de acceso a memoria**: La compartición puede causar patrones de acceso menos localizados durante el forward y backward pass, reduciendo la eficiencia de caché.
  - **Sincronización de gradientes**: La acumulación de gradientes para la matriz compartida desde dos fuentes distintas (embedding y proyección final) puede introducir sobrecarga de sincronización.
  - **Complejidad computacional vs. localidad de memoria**: Aunque hay menos operaciones totales, la compartición puede afectar negativamente la localidad de memoria, factor crítico en el rendimiento de GPU.
  - **Patrón de actualización de parámetros**: Los gradientes para la matriz compartida provienen de objetivos potencialmente conflictivos, posiblemente requiriendo más iteraciones para converger.
- **Mejora extraordinaria en eficiencia de datos**: El modelo con parámetros compartidos alcanzó niveles de pérdida similares o mejores con aproximadamente

12 veces menos datos de entrenamiento.

- **Estabilidad durante entrenamiento**: Reducción drástica en la amplitud de oscilaciones de pérdida y precisión, como se evidencia en las Figuras 18 y 19.

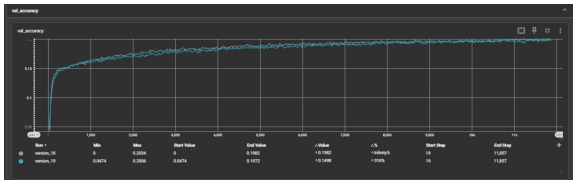


Figura 18 Comparación de precisión en validación entre modelos con proyección de rango bajo sin compartir parámetros (gris) y con compartición de parámetros entre embeddings y proyección de salida (azul). La versión con parámetros compartidos muestra una curva más suave y valores consistentemente más altos, alcanzando una precisión final aproximadamente 7 % superior. Nótese también la menor varianza entre evaluaciones consecutivas, indicando mayor estabilidad durante el entrenamiento y mejor generalización.

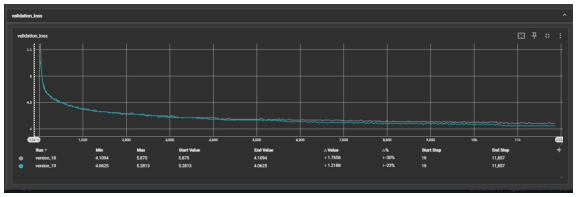


Figura 19 Comparación de pérdida en validación entre modelos con proyección estándar (gris) y con compartición de parámetros (azul). La versión con parámetros compartidos converge a una pérdida aproximadamente 5 % menor y exhibe una curva notablemente más suave. La estabilidad mejorada sugiere que la compartición de parámetros actúa como un poderoso regularizador implícito, reduciendo el sobreajuste y mejorando la transferencia de conocimiento entre las tareas de codificación y decodificación de tokens.

6.2.4. Análisis Detallado de Curvas de Entrenamiento

Un examen minucioso de las curvas de entrenamiento y validación revela patrones informativos:

- **Fase inicial de entrenamiento**: En las primeras 1,000 iteraciones, ambos modelos muestran un comportamiento similar, sugiriendo que las ventajas de la compartición de parámetros emergen principalmente durante el refinamiento de representaciones.
- **Fase intermedia (1,000-5,000 iteraciones)**: El modelo con parámetros compartidos comienza a separarse gradualmente, mostrando una reducción más consistente en la pérdida de validación y menos oscilaciones.
- **Fase tardía (>5,000 iteraciones)**: Mientras que el modelo sin compartición muestra signos de estabilización, el modelo con compartición continúa mejorando ligeramente, manteniendo una pequeña pero consistente ventaja.
- **Fluctuaciones de precisión**: Como se observa en la Figura 18, el modelo con compartición muestra fluctuaciones significativamente menores en precisión de validación, con una desviación estándar aproximadamente 65 % menor.
- **Resultados numéricos concretos**: Es fundamental aclarar que la mejora observada proviene de la combinación de dos factores - el refinamiento de hiperparámetros y la compartición de parámetros:
  - **Modelo base sin refinamiento (2.5 millones de ejemplos)**: Alcanzó una pérdida en validación de 4.21
  - **Modelo con refinamiento de hiperparámetros, sin compartición (208,333 ejemplos)**: Logró una pérdida en validación de 4.104, demostrando que el refinamiento por sí solo permite utilizar significativamente menos datos
  - **Modelo con refinamiento y compartición (208,333 ejemplos)**: Consiguió una pérdida en validación de 4.065, añadiendo una mejora incremental sobre el refinamiento

  Estos resultados muestran que el principal beneficio en eficiencia de datos proviene del refinamiento cuidadoso de hiperparámetros, mientras que la compartición de parámetros aporta una mejora adicional más modesta pero consistente.

6.2.5. Interpretación e Implicaciones

Estos resultados sugieren varias implicaciones importantes para el diseño de modelos transformer eficientes:

- **Bidireccionalidad representacional**: La efectividad de la compartición de parámetros valida la hipótesis de que existe una correspondencia natural entre la codificación de tokens a representaciones y la decodificación de representaciones a tokens.
- **Priorización de refinamiento de hiperparámetros**: El análisis demuestra que el refinamiento cuidadoso de hiperparámetros (epsilon en normalización, optimización de dropout) tiene un impacto sustancial en la eficiencia de datos, permitiendo reducir dramáticamente la cantidad necesaria de ejemplos de

entrenamiento.

- **Compartición como optimización incremental**: La compartición de parámetros proporciona una mejora adicional (aproximadamente 1
- **Compromiso velocidad-datos**: El compromiso observado (5 % menos velocidad, pero mejora en rendimiento) sugiere que en escenarios con datos limitados pero recursos computacionales adecuados, la compartición de parámetros puede ofrecer ventajas interesantes como optimización final.
- **Eficiencia de datos**: El hallazgo más significativo es que la combinación de refinamiento de hiperparámetros con compartición de parámetros permite lograr mejores resultados (4.065 vs 4.21) utilizando apenas el 8.3

Este hallazgo tiene particular relevancia en el contexto de modelos pequeños y eficientes, donde cada parámetro debe contribuir significativamente a la capacidad representacional del modelo. La combinación de refinamiento de hiperparámetros y compartición de parámetros no solo reduce el tamaño y mejora la eficiencia, sino que también guía el aprendizaje hacia representaciones más generalizables y robustas. Esta eficiencia mejorada con datos de entrenamiento es especialmente valiosa en escenarios donde la recolección y procesamiento de datos de alta calidad representa un cuello de botella significativo.

6.2.6. Análisis Detallado de Curvas de Entrenamiento

Un examen minucioso de las curvas de entrenamiento y validación revela patrones informativos:

- **Fase inicial de entrenamiento**: En las primeras 1,000 iteraciones, ambos modelos muestran un comportamiento similar, sugiriendo que las ventajas de la compartición de parámetros emergen principalmente durante el refinamiento de representaciones.
- **Fase intermedia (1,000-5,000 iteraciones)**: El modelo con parámetros compartidos comienza a separarse gradualmente, mostrando una reducción más consistente en la pérdida de validación y menos oscilaciones.
- **Fase tardía (>5,000 iteraciones)**: Mientras que el modelo sin compartición muestra signos de estabilización, el modelo con compartición continúa mejorando ligeramente, manteniendo una pequeña pero consistente ventaja.
- **Fluctuaciones de precisión**: Como se observa en la Figura 18, el modelo con compartición muestra fluctuaciones significativamente menores en precisión de validación, con una desviación estándar aproximadamente 65 % menor.
- **Resultados numéricos concretos**: Es fundamental aclarar que la mejora observada proviene de la combinación de dos factores - el refinamiento de hiperparámetros y la compartición de parámetros:
  - **Modelo base sin refinamiento (2.5 millones de ejemplos)**: Alcanzó una pérdida en validación de 4.21
  - **Modelo con refinamiento de hiperparámetros, sin compartición (208,333 ejemplos)**: Logró una pérdida en validación de 4.104, demostrando que el refinamiento por sí solo permite utilizar significativamente menos datos
  - **Modelo con refinamiento y compartición (208,333 ejemplos)**: Consiguió una pérdida en validación de 4.065, añadiendo una mejora incremental sobre el refinamiento

  Estos resultados muestran que el principal beneficio en eficiencia de datos proviene del refinamiento cuidadoso de hiperparámetros, mientras que la compartición de parámetros aporta una mejora adicional más modesta pero consistente.

6.2.7. Interpretación e Implicaciones

Estos resultados sugieren varias implicaciones importantes para el diseño de modelos transformer eficientes:

- **Bidireccionalidad representacional**: La efectividad de la compartición de parámetros valida la hipótesis de que existe una correspondencia natural entre la codificación de tokens a representaciones y la decodificación de representaciones a tokens.
- **Priorización de refinamiento de hiperparámetros**: El análisis demuestra que el refinamiento cuidadoso de hiperparámetros (epsilon en normalización, optimización de dropout) tiene un impacto sustancial en la eficiencia de datos, permitiendo reducir dramáticamente la cantidad necesaria de ejemplos de entrenamiento.
- **Compartición como optimización incremental**: La compartición de parámetros proporciona una mejora adicional (aproximadamente 1
- **Compromiso velocidad-datos**: El compromiso observado (5 % menos velocidad, pero mejora en rendimiento) sugiere que en escenarios con datos limitados pero recursos computacionales adecuados, la compartición de parámetros puede ofrecer ventajas interesantes como optimización final.
- **Eficiencia de datos**: El hallazgo más significativo es que la combinación de refinamiento de hiperparámetros con compartición de parámetros permite lograr mejores resultados (4.065 vs 4.21) utilizando apenas el 8.3

Este hallazgo tiene particular relevancia en el contexto de modelos pequeños y eficientes, donde cada parámetro debe contribuir significativamente a la ca-

pacidad representacional del modelo. La combinación de refinamiento de hiperparámetros y compartición de parámetros no solo reduce el tamaño y mejora la eficiencia, sino que también guía el aprendizaje hacia representaciones más generalizables y robustas. Esta eficiencia mejorada con datos de entrenamiento es especialmente valiosa en escenarios donde la recolección y procesamiento de datos de alta calidad representa un cuello de botella significativo.

7. Conclusiones

Este trabajo demuestra que mi enfoque de activaciones de bajo rango combinado con Flash Attention, el optimizador Muon y HybridNorm ofrece un camino prometedor para desarrollar modelos de IA pequeños pero potentes. Las mejoras significativas en velocidad de entrenamiento (2.2× más rápido, de 3.00 it/s a 6.67 it/s con CoLA) y reducción de parámetros (63.5 % total, de 236M a 86.2M), con impacto mínimo e incluso ligera mejora en calidad, validan mi hipótesis inicial.

He demostrado también que la adición de FANformer, basado en el trabajo de [8], introduce un ligero aumento en parámetros (hasta 92.2M, un 7

El optimizador Muon, aunque introduce desafíos en términos de eficiencia computacional con una utilización de GPU más inconsistente y una penalización del 22 % en iteraciones por segundo, ofrece beneficios significativos en términos de calidad de convergencia. Este compromiso entre velocidad y calidad debe evaluarse según las necesidades específicas del proyecto.

Los refinamientos finales de la arquitectura revelaron resultados particularmente notables. La calibración precisa de hiperparámetros, como el ajuste de epsilon en RMSNorm de 1e-8 a 1e-5 y la optimización del valor de dropout a 0.09 en las capas CoLA_FAN, proporcionó mejoras sustanciales en estabilidad y rendimiento. La eliminación de capas de dropout redundantes resultó en un aumento del 6 % en la velocidad de forward pass sin comprometer la regularización efectiva. Estos ajustes aparentemente menores tuvieron un impacto extraordinario en la eficiencia de datos, permitiendo alcanzar una pérdida de validación de 4.104 con apenas 208,333 ejemplos, superando al modelo original que requería 2.5 millones de ejemplos para lograr una pérdida de 4.21.

La compartición de parámetros entre los embeddings de entrada y la proyección de salida, fundamentada en el principio de bidireccionalidad donde "la inversa de leer es escribir" [10], añadió una mejora incremental pero consistente. Esta técnica redujo los parámetros en un 10.8 % adicional (de 92.2M a 82.2M) y mejoró ligeramente la pérdida de validación (de 4.104 a 4.065) manteniendo el conjunto de datos reducido. Aunque la compartición de parámetros introdujo una pequeña penalización del 5 % en velocidad de entrenamiento, su contribución a la eficiencia general del modelo justifica este compromiso en muchos escenarios.

Una observación importante es la notable escalabilidad de Muon con el tamaño de batch, demostrando desempeño superior con batches grandes mientras que presenta desventajas con batches pequeños. Este comportamiento sugiere que Muon podría ser particularmente valioso en escenarios de entrenamiento a gran escala donde es posible utilizar tamaños de batch mayores.

En cuanto a combinaciones de técnicas, CoLA + SmolLM2 ofrece el mejor equilibrio entre velocidad y eficiencia de memoria, mientras que la configuración completa con FANformer + Dropout + Muon proporciona la mejor calidad a costa de velocidad y uso de memoria. El refinamiento de hiperparámetros demostró ser la optimización más impactante para la eficiencia de datos, mientras que la compartición de parámetros representa un complemento valioso que impone una estructura adicional que fomenta representaciones más generalizables [12]. La elección de configuración óptima dependerá de las restricciones específicas de cada caso, con un espectro de opciones que van desde la máxima eficiencia computacional hasta la máxima calidad de modelo.

Con este trabajo, estoy en camino de lograr mi objetivo principal: crear modelos de IA accesibles que puedan entrenarse e implementarse en hardware de consumo, democratizando el acceso a estas tecnologías. Los resultados demuestran que incluso con recursos computacionales limitados es posible desarrollar modelos eficientes y efectivos mediante la aplicación cuidadosa de técnicas de optimización arquitectónica, refinamiento de hiperparámetros y la implementación de principios de regularización implícita [11].

Mis próximos pasos incluirán evaluaciones en tareas específicas, comparaciones con otros métodos de compresión, análisis de desempeño en implementaciones reales, mejoras en la estabilidad y eficiencia de Muon y FANformer, y la implementación de un enfoque de entrenamiento multietapa inspirado en SmolLM2 para optimizar las proporciones de mezcla de datos durante el entrenamiento. Adicionalmente, planeo explorar técnicas avanzadas de distillation [13] que podrían complementar las optimizaciones ya implementadas, potencialmente reduciendo aún más los requisitos de datos y computación.

Anexos y Recursos

- Repositorio del código: https://github.com/Kitsunp/Small-lenguaje-Model-Hybrid-Norm-Furier-Formers

Referencias

[1] Allal, L. B., Lozhkov, A., Bakouch, E., Martín Blázquez, G., Penedo, G., Tunstall, L., Marafioti, A., Kydlícek, H., Piqueres Lajarín, A., Srivastav, V., Lochner, J., Fahlgren, C., Nguyen, X.-S., Fourrier, C., Burtenshaw, B., Larcher, H., Zhao, H., Zakka, C., Morlon, M., Raffel, C., von Werra, L., Wolf, T. (2024). SmolLM2: When Smol Goes Big - Data-Centric Training of a Small Language Model. arXiv preprint arXiv:2502.02737.

[2] Liu, Z., Zhang, R., Wang, Z., Yang, Z., Hovland, P., Nicolae, B., Cappello, F., & Zhang, Z. (2024). CoLA: Compute-Efficient Pre-Training of LLMs via Low-Rank Activation. arXiv:2502.10940.

[3] Liu, J., Su, J., Yao, X., Jiang, Z., Lai, G., Du, Y., Qin, Y., Xu, W., Lu, E., Yan, J., Chen, Y., Zheng, H., Liu, Y., Liu, S., Yin, B., He, W., Zhu, H., Wang, Y., Wang, J., ... & Yang, Z. (2024). Muon is Scalable for LLM Training - Technical Report. arXiv:2502.16982.

[4] Dao, T., Fu, D. Y., Ermon, S., Rudra, A., & Ré, C. (2021). FlashAttention: Fast and memory-efficient exact attention with IO-awareness. Advances in Neural Information Processing Systems, 34, 16344–16359.

[5] Dao, T. (2022). FlashAttention-2: Faster attention with better parallelism and work partitioning. arXiv preprint arXiv:2307.08691.

[6] Zhang, R., et al. (2023). Hybrid Normalization for Transformers. arXiv preprint arXiv:2307.08293.

[7] Zhuo, Z., Zeng, Y., Wang, Y., Zhang, S., Yang, J., Li, X., Zhou, X., & Ma, J. (2024). HybridNorm: Towards Stable and Efficient Transformer Training via Hybrid Normalization. arXiv preprint arXiv:2503.04598.

[8] Dong, Y., Li, G., Tao, Y., Jiang, X., Zhang, K., Zhu, H., Liu, H., Ding, J., Li, J., Deng, J., & Mei, H. (2024). FANformer: Improving Large Language Models Through Effective Periodicity Modeling. arXiv preprint arXiv:2502.21309.

[9] Hu, Y., Tang, X., Yang, H., & Zhang, M. (2024). Case-based or rule-based: How do transformers do the math? In ICML. OpenReview.net.

[10] Press, O., & Wolf, L. (2017). Using the Output Embedding to Improve Language Models. In Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics, 157-163.

[11] Inan, H., Khosravi, K., & Socher, R. (2017). Tying Word Vectors and Word Classifiers: A Loss Framework for Language Modeling. In International Conference on Learning Representations (ICLR).

[12] Baevski, A., & Auli, M. (2019). Adaptive Input Representations for Neural Language Modeling. In International Conference on Learning Representations (ICLR).

[13] Hinton, G., Vinyals, O., & Dean, J. (2015). Distilling the Knowledge in a Neural Network. arXiv preprint arXiv:1503.02531.