



Universidade de Vigo

Trabajo Fin de Máster

Análisis comparativo de arquitecturas emergentes de aprendizaje profundo para el procesamiento del lenguaje natural

David Graña Vázquez

Máster en Técnicas Estadísticas

Curso 2025-2026

Propuesta de Trabajo Fin de Máster

Título en galego: Análise comparativa de arquitecturas emerxentes de aprendizaxe profunda para o procesamento da linguaxe natural
Título en español: Análisis comparativo de arquitecturas emergentes de aprendizaje profundo para el procesamiento del lenguaje natural
English title: Comparative Analysis of Emerging Deep Learning Architectures for Natural Language Processing
Modalidad: Modalidad A
Autor/a: David Graña Vázquez, USC
Director/a: Guillermo López Taboada, UDC
Breve resumen del trabajo: Desde la aparición del modelo Transformer en 2017, el Procesamiento del Lenguaje Natural (PLN) ha experimentado una transformación sin precedentes. Sin embargo, su elevado coste computacional y su complejidad creciente han impulsado el desarrollo de nuevas arquitecturas que buscan mantener su capacidad representacional mejorando la eficiencia y optimizando el uso de recursos. Este Trabajo Fin de Máster tiene como objetivo analizar y comparar distintas arquitecturas recientes aplicadas a tareas de PLN. Por un lado, se estudian variantes del Transformer que perfeccionan aspectos concretos de la arquitectura original, como el mecanismo de atención o el manejo de secuencias largas. Por otro, se exploran modelos emergentes como Mamba, que introducen una estructura fundamentalmente diferente y presentan ventajas notables en términos de escalabilidad y eficiencia. El estudio combina una revisión teórica de las principales innovaciones con un análisis empírico, entrenando modelos desde cero sobre tareas de lenguaje natural. Los resultados se evaluarán mediante métricas de rendimiento y eficiencia, con el objetivo de identificar las fortalezas y limitaciones de cada enfoque y ofrecer una visión comparativa clara del estado actual de estas arquitecturas.
Recomendaciones:
Otras observaciones:

Guillermo López Taboada, profesor de la UDC, informa que el Trabajo Fin de Máster titulado

Análisis comparativo de arquitecturas emergentes de aprendizaje profundo para el procesamiento del lenguaje natural

fue realizado bajo su dirección por David Graña Vázquez para el Máster en Técnicas Estadísticas. Estimando que el trabajo está terminado, da su conformidad para su presentación y defensa ante un tribunal.

En Santiago, a 17 de julio de 2026.

LOPEZ
TABOADA
GUILLERMO
-
44088774M

Firmado digitalmente por LOPEZ TABOADA GUILLERMO - 44088774M
Fecha: 2026.07.17 23:36:06 +02'00'

El director:
Guillermo López Taboada



El autor:
David Graña Vázquez

Declaración responsable. Para dar cumplimiento a la Ley 3/2022, de 24 de febrero, de convivencia universitaria, referente al plagio en el Trabajo Fin de Máster (Artículo 11, [Disposición 2978 del BOE núm. 48 de 2022](#)), **el/la autor/a declara** que el Trabajo Fin de Máster presentado es un documento original en el que se han tenido en cuenta las siguientes consideraciones relativas al uso de material de apoyo desarrollado por otros/as autores/as:

- Todas las fuentes usadas para la elaboración de este trabajo han sido citadas convenientemente (libros, artículos, apuntes de profesorado, páginas web, programas,...)
- Cualquier contenido copiado o traducido textualmente se ha puesto entre comillas, citando su procedencia.
- Se ha hecho constar explícitamente cuando un capítulo, sección, demostración,... sea una adaptación casi literal de alguna fuente existente.

Y, acepta que, si se demostrara lo contrario, se le apliquen las medidas disciplinarias que correspondan.

Índice general

Resumen	VII
1. Introducción	1
1.1. Contexto histórico	1
1.2. Estructura del Trabajo	2
1.3. Objetivos del Trabajo	3
2. Fundamentos del Aprendizaje Profundo en NLP	4
2.1. Red Neuronal de propagación directa	4
2.1.1. Retropropagación	6
2.1.2. Teorema de aproximación universal	7
2.2. Representación de palabras	7
2.2.1. Bolsa de palabras	7
2.2.2. Tokenización en subpalabras	9
2.2.3. Skip-Gram	9
2.2.4. Representación Dinámica (Embeddings)	11
2.2.5. Modelos de clasificación binaria	11
2.3. Redes Neuronales Recurrentes	14
2.3.1. Red neuronal recurrente simple	14
2.3.2. Modelado de lenguaje con RNNs	15
2.3.3. El problema del desvanecimiento del gradiente	15
2.3.4. LSTM	16
2.3.5. Encoder-Decoder	17
2.3.6. Mecanismo de atención	18
3. Arquitectura Transformer	19
3.1. Auto-atención	20
3.1.1. Queries, Keys y Values	21
3.1.2. Factor de escala	21
3.1.3. Formulación matricial y complejidad	21
3.1.4. Enmascaramiento causal	22
3.1.5. La atención como recuperación de información suave	22
3.2. Auto-atención multicabeza	23
3.2.1. Formulación	23
3.2.2. Dimensiones y coste	23
3.2.3. Interpretabilidad de las cabezas	23
3.3. Bloque Transformer	24
3.3.1. Capa de propagación directa posicional	24
3.3.2. Conexiones residuales	24
3.3.3. Normalización de capa	25

3.3.4.	Post-normalización vs Pre-normalización	25
3.4.	Codificación posicional	25
3.4.1.	Codificaciones sinusoidales	26
3.4.2.	Codificaciones aprendidas	26
3.5.	Encoder y Decoder del Transformer	26
3.5.1.	Encoder	26
3.5.2.	Decoder	27
3.5.3.	Entrenamiento con teacher forcing	27
3.6.	Variante decoder-only	27
3.7.	Modelos preentrenados	28
4.	Evaluación inicial de los modelos base	29
4.1.	Clasificación de textos	29
4.1.1.	Diseño experimental	29
4.1.2.	Resultados globales	30
4.1.3.	Efecto de la longitud de secuencia	30
4.1.4.	Discusión	32
4.2.	Modelado de lenguaje	34
4.2.1.	Diseño experimental	34
4.2.2.	Resultados	35
4.3.	Discusión: ¿cuándo importa la atención?	36
4.3.1.	Validación experimental de los fundamentos teóricos	36
4.3.2.	Hipótesis para los capítulos posteriores	37
5.	Mejoras en la arquitectura Transformer para secuencias largas	38
5.1.	Limitaciones del Transformer y motivación de nuevas arquitecturas	38
5.2.	Evolución de los componentes del Transformer	39
5.2.1.	RMSNorm	39
5.2.2.	SwiGLU	39
5.2.3.	FlashAttention	40
5.2.4.	Otras mejoras relevantes	40
5.3.	El problema de la codificación posicional	41
5.4.	Codificación posicional rotacional (RoPE)	41
5.4.1.	Intuición y formulación	41
5.4.2.	Formulación por pares de dimensiones	41
5.4.3.	Demostración de la propiedad de posición relativa	43
5.4.4.	Conexión con los números complejos e implementación eficiente	43
5.4.5.	Propiedades de decaimiento y extrapolación	44
5.4.6.	Adopción en modelos modernos	45
5.5.	ALiBi: Attention with Linear Biases	45
5.5.1.	Formulación	45
5.5.2.	Asignación de pendientes	45
5.5.3.	Propiedades de extrapolación	46
5.5.4.	Comparación entre RoPE y ALiBi	46
5.6.	Mixture of Experts (MoE)	47
5.6.1.	Principio de funcionamiento	47
5.6.2.	Capacidad vs. coste	48
5.6.3.	Balanceo de carga	48
5.6.4.	Relevancia para este trabajo	49
5.7.	Resumen	49

6. Modelos de espacio de estados (SSM)	50
6.1. Motivación: los tres modos de procesamiento de secuencias	50
6.2. Formulación continua	51
6.2.1. Papel de cada componente	51
6.2.2. Solución analítica	51
6.2.3. Conexión con otros modelos	51
6.3. Discretización	52
6.3.1. Método de Euler	52
6.3.2. Retención de orden cero (ZOH)	52
6.3.3. Transformada bilineal	52
6.3.4. La recurrencia discreta y el papel de Δ	52
6.4. Formulación convolucional	53
6.4.1. Derivación	53
6.4.2. Cálculo eficiente mediante FFT	53
6.4.3. La dualidad recurrencia-convolución	53
6.5. S4: Structured State Spaces for Sequence Modeling	54
6.5.1. El problema de la inicialización de $\mathbf{A}$	54
6.5.2. HiPPO: aproximación polinomial del historial	54
6.5.3. Parametrización DPLR y cálculo eficiente	55
6.5.4. Resultados y limitaciones de S4	55
6.6. Mamba: SSM selectivos	55
6.6.1. La limitación de la invariancia a la entrada	55
6.6.2. El mecanismo de selección	56
6.6.3. El algoritmo de escaneo paralelo selectivo	57
6.6.4. Arquitectura del bloque Mamba	57
6.6.5. El estado como cuello de botella de información	58
6.7. La dualidad SSM-Atención	58
6.8. Arquitecturas híbridas: Jamba	59
6.8.1. Motivación: limitaciones complementarias	59
6.8.2. Arquitectura	59
6.8.3. Ventajas en memoria e inferencia	60
6.8.4. Hallazgos del entrenamiento a escala	60
6.8.5. Relevancia para nuestro estudio	61
6.9. Resumen comparativo de arquitecturas	61
7. Evaluación experimental comparativa	63
7.1. Modelos evaluados	63
7.2. WikiText-103: modelado de lenguaje	65
7.2.1. Configuración experimental	65
7.2.2. Resultados de perplejidad	66
7.2.3. Eficiencia computacional	70
7.2.4. Síntesis: relación calidad-eficiencia	73
7.3. LRA ListOps: razonamiento estructural	74
7.3.1. Configuración experimental	75
7.3.2. Resultados de <i>accuracy</i>	75
7.3.3. Eficiencia computacional	78
7.3.4. Síntesis: relación calidad-eficiencia	81
7.4. Discusión de resultados	82

8. Conclusiones	84
8.1. Hallazgos principales	84
8.2. Contribuciones	85
8.3. Limitaciones del estudio	86
8.4. Líneas de trabajo futuro	87
8.5. Reflexión final	87
Apéndice	89
A. Conjuntos de datos	90
A.1. Visión general de los conjuntos de datos	90
A.2. Progreso	90
A.2.1. Progreso Tema	91
A.2.2. Progreso Género	93
A.3. 20newsgroups	93
A.4. WikiText-103	95
A.5. LRA ListOps	96
Bibliografía	98

Resumen

Resumen en español

Desde la aparición del Transformer en 2017, el procesamiento del lenguaje natural ha experimentado una transformación sin precedentes apoyada en el mecanismo de auto-atención, cuya capacidad de capturar dependencias entre tokens arbitrariamente distantes ha redefinido el estado del arte. Pero su coste cuadrático en la longitud de secuencia ha impulsado dos líneas de investigación paralelas: variantes del propio Transformer que mejoran la codificación posicional (RoPE, ALiBi) o escalan su capacidad de forma eficiente (Mixture of Experts), y arquitecturas alternativas como los SSMs (State Space Models), que ofrecen complejidad lineal, donde partimos de la variante estándar (Mamba), y lo extendemos a las arquitecturas híbridas (Jamba) que combinan ambos paradigmas. La pregunta de en qué tareas y bajo qué condiciones cada uno de estos enfoques aporta una ventaja real sigue abierta, y las comparaciones publicadas rara vez controlan a la vez el número de parámetros, los recursos computacionales y el protocolo de entrenamiento.

Este Trabajo Fin de Máster aborda dicha cuestión combinando una revisión teórica de las principales familias de arquitecturas con una evaluación empírica controlada de ocho modelos —FFN, LSTM, Transformer con embeddings aprendidos, Transformer con RoPE, Transformer con ALiBi, Transformer con RoPE y MoE, Mamba y Jamba— entrenados desde cero bajo condiciones homogéneas (idéntica dimensión del modelo, número de capas, optimizador y presupuesto efectivo de tokens por paso de optimización). Los modelos se evalúan en dos tareas deliberadamente contrastadas: modelado de lenguaje sobre WikiText-103, que prima la explotación de estadísticas distribucionales a múltiples escalas, y razonamiento composicional sobre LRA ListOps, que exige recuperación jerárquica basada en contenido. En ambas tareas se varía sistemáticamente la longitud de secuencia para aislar el efecto del contexto disponible sobre el rendimiento de cada arquitectura.

El trabajo deja, más allá de los resultados experimentales, cuatro contribuciones. La primera es la comparación homogénea de las ocho arquitecturas bajo un mismo protocolo, que permite atribuir las diferencias observadas a la estructura de cada modelo y no al ajuste de hiperparámetros. La segunda es la detección y corrección de varias asimetrías de implementación en los modelos de referencia, un paso previo imprescindible para que la comparación sea justa. La tercera es la evidencia empírica directa, a una escala accesible y reproducible, de las limitaciones de los SSMs para recuperar información por contenido, que complementa las pruebas formales recientes. La cuarta es la cuantificación de cuánta calidad y cuánta eficiencia aporta la hibridación SSM-atención a la escala estudiada.

Los resultados arrojan cinco hallazgos principales. **Primero**, la codificación posicional es determinante y dependiente de la tarea: ALiBi, el mejor Transformer puro a contexto largo en WikiText-103, se convierte en el peor en LRA ListOps, donde su sesgo lineal de localidad atenúa justo las puntuaciones de atención a posiciones distantes que el emparejamiento de paréntesis necesita amplificar. **Segundo**, la atención no es opcional cuando la tarea exige recuperación basada en contenido: Mamba alcanza una perplejidad comparable a los Transformers en WikiText-103 (63,0 a $L = 1024$) pero colapsa en ListOps a una *accuracy* escasamente superior al azar (18,45 % a $L = 2048$), pues su estado comprimido de dimensión fija ($d_{\text{state}} = 16$) actúa como un cuello de botella estructural que impide recuperar resultados parciales en posiciones anidadas arbitrarias. **Tercero**, el aumento de capacidad no sustituye al

mecanismo adecuado: el Transformer con MoE mejora la perplejidad en WikiText-103 un 21,6 % sobre el Transformer denso con RoPE, pero apenas afecta a la *accuracy* en ListOps, lo que confirma que añadir parámetros ayuda cuando el cuello de botella es estadístico y no cuando es composicional. **Cuarto**, la hibridación SSM-atención ofrece el mejor compromiso global: Jamba, con el patrón M-A-M-A, alcanza la segunda mejor perplejidad en WikiText-103 a $L = 1024$ con un *throughput* un 20 % superior al de los Transformers puros, y la mejor *accuracy* absoluta en LRA ListOps a $L = 2048$ (46,15 %), siendo además el único modelo que mejora con la longitud en esta tarea. **Quinto**, las predicciones teóricas de complejidad se confirman empíricamente con notable precisión: FFN y Mamba mantienen memoria y *throughput* esencialmente invariantes con L ; los Transformers puros muestran el crecimiento cuadrático de memoria predicho por la teoría; y Jamba ocupa el punto intermedio coherente con la mitad de sus capas lineales.

La conclusión general no es que una arquitectura sustituya a las demás, sino que cada una codifica una hipótesis distinta sobre la estructura de las dependencias del problema: a la escala estudiada y en régimen de entrenamiento desde cero, la hibridación resulta preferible en las dos tareas evaluadas.

English abstract

Since the introduction of the Transformer in 2017, natural language processing has undergone an unprecedented transformation built on the self-attention mechanism, whose ability to capture dependencies between arbitrarily distant tokens has redefined the state of the art. But its quadratic cost in sequence length has spurred two parallel research directions: variants of the Transformer itself that improve positional encoding (RoPE, ALiBi) or scale its capacity efficiently (Mixture of Experts), and alternative architectures such as SSMs (State Space Models), which offer linear complexity, where we start from the standard variant (Mamba) and extend the study to hybrid architectures (Jamba) that combine both paradigms. The question of on which tasks and under what conditions each of these approaches delivers a real advantage remains open, and published comparisons rarely control simultaneously for parameter count, computational resources, and training protocol.

This Master’s Thesis addresses this question by combining a theoretical review of the main families of architectures with a controlled empirical evaluation of eight models —FFN, LSTM, Transformer with learned positional embeddings, Transformer with RoPE, Transformer with ALiBi, Transformer with RoPE and MoE, Mamba, and Jamba— trained from scratch under homogeneous conditions (identical model dimension, number of layers, optimizer, and effective token budget per optimization step). The models are evaluated on two deliberately contrasting tasks: language modeling on WikiText-103, which rewards exploiting distributional statistics at multiple scales, and compositional reasoning on LRA ListOps, which demands hierarchical content-based retrieval. In both tasks the sequence length is varied systematically in order to isolate the effect of available context on the performance of each architecture.

Beyond the experimental results, the thesis makes four contributions. The first is a homogeneous comparison of the eight architectures under a single protocol, which lets observed differences be attributed to each model’s structure rather than to hyperparameter tuning. The second is the detection and correction of several implementation asymmetries in the baseline models, an essential prerequisite for a fair comparison. The third is direct empirical evidence, at an accessible and reproducible scale, of the limitations of SSMs in content-based retrieval, complementing recent formal proofs. The fourth is a quantification of how much quality and how much efficiency SSM-attention hybridization provides at the scale studied.

The results yield five main findings. **First**, positional encoding is decisive and task-dependent: ALiBi, the best pure Transformer at long context on WikiText-103, becomes the worst on LRA ListOps, where its linear locality bias attenuates the very attention scores at distant positions that bracket matching needs to amplify. **Second**, attention is not optional when the task requires content-based retrieval: Mamba reaches a perplexity comparable to that of Transformers on WikiText-103 (63,0 at $L = 1024$) but collapses on ListOps to an accuracy barely above chance (18,45 % at $L = 2048$), as

its fixed-size compressed state ($d_{\text{state}} = 16$) acts as a structural bottleneck that prevents recovering partial results at arbitrarily nested positions. **Third**, increased capacity does not substitute for the right mechanism: the Transformer with MoE improves WikiText-103 perplexity by 21,6 % over the dense Transformer with RoPE, but barely affects ListOps accuracy, confirming that adding parameters helps when the bottleneck is statistical and not when it is compositional. **Fourth**, SSM-attention hybridization offers the best overall trade-off: Jamba, with the M-A-M-A pattern, attains the second-best perplexity on WikiText-103 at $L = 1024$ with 20 % higher throughput than pure Transformers, and the best absolute accuracy on LRA ListOps at $L = 2048$ (46,15 %), being moreover the only model whose accuracy improves with length on this task. **Fifth**, theoretical complexity predictions are confirmed empirically with remarkable precision: FFN and Mamba keep memory and throughput essentially invariant with L ; pure Transformers exhibit the quadratic memory growth predicted by theory; and Jamba occupies the intermediate point consistent with half of its layers being linear.

The general conclusion is not that one architecture supplants the others, but that each encodes a different hypothesis about the structure of a problem’s dependencies: at the scale studied and under from-scratch training, hybridization proves preferable on both tasks evaluated.

Capítulo 1

Introducción

1.1. Contexto histórico

El estudio del lenguaje natural por parte de las máquinas tiene una larga historia que se remonta a los años 50 y 60 [23]. Inicialmente, los enfoques se basaban en reglas gramaticales y sintácticas, con el objetivo de traducir lenguajes de manera mecánica. Un ejemplo temprano de este enfoque es el programa de traducción automática Georgetown-IBM, que en 1954 logró traducir más de 60 oraciones del ruso al inglés. Este programa utilizaba un conjunto predefinido de reglas gramaticales y vocabulario específico para realizar traducciones. Sin embargo, estos sistemas eran limitados y no podían manejar la ambigüedad y la riqueza del lenguaje humano. Por ejemplo, no podían resolver ambigüedades como el significado múltiple de una palabra según el contexto, ni podían adaptarse a nuevas expresiones o situaciones no contempladas en sus reglas.

Durante estas décadas, se desarrollaron otros sistemas basados en reglas, incluyendo el sistema SHRDLU de Terry Winograd. Este programa podía entender y manipular bloques de colores en un entorno cerrado siguiendo instrucciones sencillas. El modelo utilizaba un conjunto de reglas sintácticas y semánticas para descomponer y entender las instrucciones, permitiéndole realizar tareas complejas en un dominio restringido. No obstante, SHRDLU también puso de manifiesto las limitaciones de los enfoques basados en reglas para comprender y generar lenguaje en contextos más amplios y variados. Aunque funcionaba bien dentro de su mundo limitado de bloques, las reglas que empleaba no podían generalizarse fácilmente a otros dominios más complejos del lenguaje humano.

Con el avance de la computación, se desarrollaron métodos estadísticos en los años 80 y 90. Estos enfoques se basaban en modelos probabilísticos para comprender y generar texto. A diferencia de los modelos basados en reglas, los modelos estadísticos podían aprender patrones del lenguaje a partir de grandes conjuntos de datos. Uno de los avances más significativos de esta época fue el desarrollo de los modelos de *n*-gramas, que permitían predecir la probabilidad de una palabra basada en las palabras precedentes. Estos modelos mejoraron considerablemente el rendimiento en tareas como la corrección ortográfica y la predicción de texto, pero aún carecían de la capacidad para capturar el contexto a largo plazo y la complejidad semántica del lenguaje.

La verdadera revolución en el procesamiento del lenguaje natural (NLP, *Natural Language Processing*) llegó con el desarrollo de redes neuronales y modelos de aprendizaje profundo en la década de 2010. Las redes neuronales recurrentes (RNN) comenzaron a mostrar resultados prometedores en una variedad de tareas de NLP. En particular, los modelos de secuencia a secuencia (seq2seq) con atención, introducidos por Bahdanau et al. en 2014 [3], representaron un avance importante al permitir que los modelos prestaran atención a diferentes partes de la entrada mientras generaban la salida. Sin embargo, a pesar de su éxito, las RNN enfrentaban dificultades con la paralelización y el manejo de dependencias a largo plazo.

En 2017, Vaswani et al. [39] presentaron la arquitectura Transformer, que marcó un hito significativo en el campo del NLP. A diferencia de las RNN, los Transformers utilizan un mecanismo de atención que

permite el procesamiento paralelo de secuencias, mejorando drásticamente la eficiencia y el rendimiento. Los Transformers no solo superaron a las RNN en tareas de traducción automática y resumen de texto, sino que también sentaron las bases para el desarrollo de modelos preentrenados de gran escala como BERT [11] y GPT [30], que han establecido nuevos estándares en el procesamiento del lenguaje.

Sin embargo, el mecanismo de atención del Transformer presenta un coste computacional cuadrático con respecto a la longitud de la secuencia, lo que limita su aplicación con textos largos. Esta restricción ha motivado dos líneas de investigación activas en los últimos años. Por un lado, se han desarrollado variantes eficientes del Transformer, como Longformer [4] o BigBird [42], que modifican el patrón de atención para reducir su complejidad. Por otro lado, han surgido arquitecturas alternativas basadas en modelos de espacio de estados (SSM, *State Space Models*), como S4 [16] y Mamba [17], que abandonan completamente el paradigma de la atención para proponer mecanismos con complejidad lineal capaces de modelar dependencias a larga distancia. Estas arquitecturas emergentes representan un cambio de paradigma que cuestiona si la atención es realmente el único camino hacia un procesamiento eficiente del lenguaje.

1.2. Estructura del Trabajo

Este trabajo está estructurado en ocho capítulos que recorren la evolución de las arquitecturas de aprendizaje profundo aplicadas al procesamiento del lenguaje natural, desde los fundamentos hasta las propuestas más recientes, culminando con un análisis empírico comparativo.

En el **segundo capítulo** se presentan las bases teóricas de las redes neuronales: cómo funcionan, qué ventajas tienen y cómo se entrenan. Esto se complementa con una introducción a los distintos métodos de representar el lenguaje como vectores, así como un caso práctico con una red neuronal simple. Para concluir el capítulo se explican las redes neuronales recurrentes, su forma más avanzada la LSTM y la arquitectura encoder-decoder con atención, que es el antecedente directo del Transformer.

En el **tercer capítulo** se entra en detalle en la arquitectura Transformer, explicando en profundidad su mecanismo de auto-atención y las ventajas que ofrece frente a los modelos recurrentes. Posteriormente se detalla la arquitectura completa, incluyendo los mecanismos complementarios a la atención (normalización, conexiones residuales, codificación posicional), los modelos preentrenados más relevantes y las innovaciones posteriores que han perfeccionado sus componentes internos. El capítulo concluye identificando las limitaciones del Transformer que motivan el desarrollo de nuevas arquitecturas.

El **cuarto capítulo** presenta una evaluación inicial de tres modelos base —FFN, LSTM y Transformer— en tareas de clasificación de textos y modelado de lenguaje. Los resultados revelan que cada tarea favorece a una arquitectura diferente, motivando la pregunta central de este trabajo: ¿cómo influye la longitud de secuencia en el rendimiento de cada modelo?

En el **quinto capítulo** se estudian las mejoras al Transformer diseñadas para abordar el problema de la longitud de secuencia. Se formalizan en detalle tres innovaciones que dan lugar a modelos experimentales: las codificaciones posicionales rotacionales (RoPE), la atención con sesgo lineal (ALiBi) y la técnica de Mixture of Experts (MoE).

El **sexto capítulo** se dedica íntegramente a los modelos de espacio de estados. Partiendo de la formulación continua de los sistemas dinámicos lineales, se desarrolla la discretización, la formulación convolucional dual y el modelo S4. Se culmina con la arquitectura Mamba, que introduce un mecanismo de selección dependiente de la entrada con complejidad lineal.

En el **séptimo capítulo** se lleva a cabo la evaluación comparativa principal del trabajo. Los ocho modelos —FFN, LSTM, Transformer con embeddings aprendidos, Transformer con RoPE, Transformer con ALiBi, Transformer con RoPE y MoE, Mamba y Jamba— se evalúan en dos tareas: modelado de lenguaje sobre WikiText-103 (variando la longitud de contexto) y razonamiento estructural sobre LRA ListOps. El capítulo concluye con un análisis transversal que sintetiza los patrones observados.

Finalmente, el **octavo capítulo** recoge las conclusiones del trabajo, sus principales contribuciones, las limitaciones del estudio y las líneas de trabajo futuro.

1.3. Objetivos del Trabajo

El objetivo principal de este trabajo es explorar, analizar y comparar distintas arquitecturas de aprendizaje profundo aplicadas al procesamiento del lenguaje natural, con especial énfasis en cómo la longitud de secuencia influye en el rendimiento de cada modelo. Para alcanzar este objetivo, se plantean los siguientes objetivos específicos:

- Revisar la literatura existente sobre la evolución del NLP, desde las redes recurrentes hasta las arquitecturas más recientes como Mamba.
- Explicar detalladamente la arquitectura Transformer, su mecanismo de atención y las innovaciones que han mejorado sus componentes.
- Presentar los modelos de espacio de estados (SSM) y la arquitectura Mamba, analizando sus fundamentos teóricos y las ventajas que ofrecen.
- Evaluar el rendimiento de ocho arquitecturas —desde modelos base hasta variantes avanzadas del Transformer y Mamba— en tareas de modelado de lenguaje y razonamiento estructural.
- Estudiar el efecto de la longitud de secuencia sobre el rendimiento y la eficiencia de cada arquitectura, identificando qué modelos escalan mejor.
- Proporcionar una visión comparativa clara del estado actual de las arquitecturas de aprendizaje profundo para NLP, identificando fortalezas y limitaciones de cada enfoque.

Capítulo 2

Fundamentos del Aprendizaje Profundo en NLP

El presente capítulo presenta los fundamentos del aprendizaje profundo aplicados al procesamiento del lenguaje natural. Estos conceptos son la base sobre la que se construyen todas las arquitecturas que se comparan en este trabajo, desde los modelos más sencillos hasta los Transformers y los modelos de espacio de estados. Comenzaremos con las redes neuronales de propagación directa, veremos cómo representar el lenguaje como vectores, y culminaremos con las redes recurrentes y el mecanismo de atención, que son los antecedentes directos de las arquitecturas modernas.

2.1. Red Neuronal de propagación directa

La base del aprendizaje profundo es la **red neuronal**. Una red neuronal consiste en un conjunto de neuronas artificiales distribuidas en distintas capas, de manera que el conjunto aproxima una determinada función f^* . Por ejemplo, en el contexto del procesamiento del lenguaje natural, podemos buscar una función para clasificar un texto x dentro de una temática determinada y , teniendo así la ecuación $y = f^*(\mathbf{x})$. Para llegar a aproximar f^* , la red neuronal define la igualdad $y = f(\mathbf{x}; \theta)$ y aprende los valores de θ que mejor aproximan la función.

Podemos entender las redes neuronales como la forma de superar las limitaciones de los modelos lineales. Los modelos lineales son muy útiles debido a su fácil comprensión y optimización, al presentar problemas de programación convexa. Sin embargo, están restringidos a aprovechar únicamente relaciones lineales, algo que no es suficiente para comprender las interacciones de un sistema tan complejo como el lenguaje humano. Para solucionar esto, podemos intentar aplicar el modelo lineal no a la entrada $\mathbf{x}$, sino a una transformación no lineal de la entrada $\phi(\mathbf{x})$. La estrategia del aprendizaje profundo es aprender ϕ , es decir dado el problema inicial, podemos desarrollarlo como

$$\mathbf{y} = f(\mathbf{x}; \theta, \mathbf{w}) = \phi(\mathbf{x}; \theta)^T \mathbf{w}$$

Así tenemos parámetros θ que nos permiten aprender ϕ de una amplia clase de funciones, y parámetros $\mathbf{w}$ que mapean $\phi(\mathbf{x})$ a la salida deseada. Esta tarea se consigue definiendo una función de coste, que compare la salida del modelo con el valor esperado, y resolviendo el problema de optimización asociado mediante descenso gradiente o alguna de sus variantes.

La unidad básica que nos permite construir una red neuronal es el perceptrón. Un **perceptrón** se puede representar como una función que recibe varias entradas $\mathbf{x} = [x_1, x_2, \dots, x_n]$, realiza una combinación lineal de estas con un vector de pesos $\mathbf{w} = [w_1, w_2, \dots, w_n]$ y un sesgo b , aplica una función de activación f y produce una salida y . Por lo tanto tenemos:

$$y = f(\mathbf{w}^T \mathbf{x} + b) \tag{2.1}$$

Si juntamos diferentes perceptrones y les otorgamos la misma entrada a todos, podemos construir $\mathbf{y} = [y_1, y_2, \dots, y_n]$ un vector de salidas, creando así una **capa** de neuronas. Este vector de salidas a su vez puede ser usado como entrada en nuevas neuronas, construyendo así una nueva capa. De esta forma utilizando n capas de neuronas conectadas, tenemos que $f(\mathbf{x}) = f^{(n)}(\dots f^{(2)}(f^{(1)}(\mathbf{x})))$, construyendo así una red neuronal, siendo $f^{(i)}$ con $i \in \{1, \dots, n\}$ la capa i -ésima. Según su función dentro de la red neuronal tenemos la **capa de entrada**, $f^{(1)}$ (la que procesa por primera vez el vector de entradas), la **capa de salida**, $f^{(n)}$ (la última de la red y que calcula el vector de salida) y las capas ocultas, $f^{(i)}$ con $i \in \{2, \dots, n-1\}$, que se encuentran entre la entrada y la salida. Esto es lo que denominamos una **red neuronal de propagación directa** (*feedforward neural network*) puesto que las neuronas solo envían información a las siguientes capas y nunca a las anteriores, tal y como se muestra en la Figura 2.1.

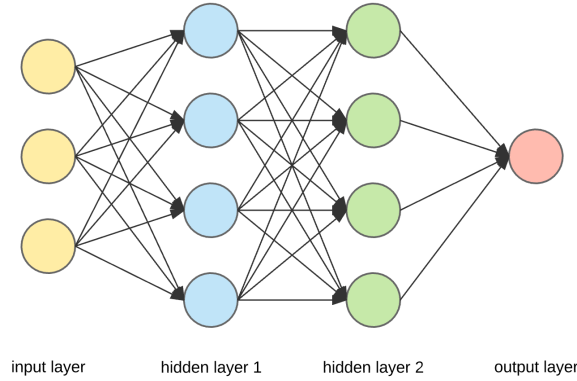


Figura 2.1: Ejemplo esquemático de una red neuronal de propagación directa

Tal y como mencionábamos al principio, una de las ventajas más importantes de las redes neuronales es su capacidad de tener en cuenta interacciones no lineales. Esto se consigue mediante las **funciones de activación** que se aplican en cada neurona, entre las que destacan:

- ReLU (*Rectified Linear Unit*): $g(z) = \max\{0, z\}$. Tiene la ventaja de ser muy similar a una función lineal, lo que facilita la optimización de los modelos. Su mayor inconveniente es no ser diferenciable en $z = 0$.
- Sigmoides: $\sigma(z) = \frac{1}{1+e^{-z}}$. Es una función diferenciable en todo su dominio, sin embargo se satura con facilidad, dificultando el descenso gradiente.
- Tangente hiperbólica: $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$. Al igual que la función sigmoide es diferenciable en todo su dominio.
- SiLU (*Sigmoid Linear Unit*): $\text{SiLU}(z) = z \cdot \sigma(z)$. Combina las propiedades de la ReLU y la sigmoide, siendo suave y no monótona. Es la función de activación utilizada en arquitecturas modernas como Mamba.

Por último, necesitamos definir el problema de optimización que resolveremos para entrenar nuestra red neuronal. Para ello el método más relevante es el de máxima verosimilitud. Es decir, buscamos encontrar el modelo que minimiza las diferencias entre $p_{\text{modelo}}(\mathbf{y}|\mathbf{x})$ (la distribución de probabilidad predecida por el modelo de los posibles valores de salida $\mathbf{y}$, dada la entrada $\mathbf{x}$) y $p_{\text{datos}}(\mathbf{y}|\mathbf{x})$ (la distribución de probabilidad real de y , representada por los datos de entrenamiento). Esto se consigue usando la entropía cruzada entre ambas distribuciones.

$$J(\theta) = -\mathbb{E}_{\mathbf{x}, \mathbf{y} \sim \hat{p}_{\text{datos}}} \log p_{\text{modelo}}(\mathbf{y}|\mathbf{x}) \quad (2.2)$$

De esta manera no hay que diseñar una función de coste específica para cada modelo. Al definir el modelo $p(\mathbf{y}|\mathbf{x})$, automáticamente queda determinada la función de coste. Por lo tanto una de las partes más influyentes sobre $J(\boldsymbol{\theta})$ es cómo escogemos representar la salida del modelo, es decir, cómo es nuestra **unidad de salida**. Entre las salidas principales destacamos:

- Unidad Lineal: $\hat{y} = \mathbf{W}^T \mathbf{h} + \mathbf{b}$ de forma que $p(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{y}; \hat{\mathbf{y}}, \mathbf{I})$
- Unidad Sigmoidal: Nuestra variable respuesta es binaria. Definimos una distribución de Bernoulli sobre y condicionada a $\mathbf{x}$, de forma que $\hat{y} = \sigma(\mathbf{w}^T \mathbf{h} + b)$, por lo tanto $P(y = 1|x) = \sigma(z)$, siendo $z = \mathbf{w}^T \mathbf{h} + b$
- Unidad Softmax: Nuestra variable respuesta es discreta con n posibles valores. Definiendo una distribución Multinoulli sobre $\mathbf{y}$ condicionada a $\mathbf{x}$ y partiendo de $\mathbf{z} = \mathbf{W}^T \mathbf{h} + \mathbf{b}$, tenemos que $\text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$. De esta forma buscamos maximizar $\log P(y = i, \mathbf{z}) = \log \text{softmax}(\mathbf{z})_i$

2.1.1. Retropropagación

Ahora para entrenar el modelo, debemos resolver un problema de optimización mediante descenso gradiente. Aunque no es tan sencillo, ya que evaluar numéricamente la expresión del gradiente puede ser muy costoso computacionalmente. Esta dificultad se resuelve gracias al algoritmo de **retropropagación** (*backpropagation*) que nos permite obtener de manera más eficiente el gradiente de la función de coste con respecto a los parámetros de la red neuronal $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. El algoritmo de retropropagación simplemente efectúa la regla de la cadena en un determinado orden para que sea más eficiente, almacenando en memoria los gradientes ya calculados, para así evitar cálculos redundantes.

Comprender bien este algoritmo es importante porque, como veremos, uno de los principales problemas de las redes recurrentes (el desvanecimiento del gradiente) surge precisamente de cómo se propagan los gradientes a través de muchas capas o pasos temporales. A continuación mostramos un ejemplo simplificado del algoritmo general de retropropagación; para una formulación más general consultar [14]:

Paso	Descripción
Requerimientos	Dada $J(\mathbf{y}, \hat{\mathbf{y}})$ donde $\mathbf{y}$ es el valor real y $\hat{\mathbf{y}}$ es la predicción de la red. Dados valores iniciales $\mathbf{W}^{(i)}$ y $\mathbf{b}^{(i)}, i \in \{1, \dots, l\}$ e inicializamos $\mathbf{a}^{(0)} = \mathbf{x}$
Propagación hacia adelante	Para cada $i = 1, 2, \dots, l-1, l$: $\mathbf{z}^{(i)} = \mathbf{W}^{(i)} \mathbf{a}^{(i-1)} + \mathbf{b}^{(i)}$ y $\mathbf{a}^{(i)} = f^{(i)}(\mathbf{z}^{(i)})$.
Error en la capa de salida	$\boldsymbol{\delta}^{(l)} = \nabla_{\hat{\mathbf{y}}} J \odot f'^{(l)}(\mathbf{z}^{(l)})$ siendo $\odot$ el producto de Hadamard (elemento a elemento).
Retropropagación del error	Para cada $i = l-1, l-2, \dots, 2, 1$: $\boldsymbol{\delta}^{(i)} = ((\mathbf{W}^{(i+1)})^\top \boldsymbol{\delta}^{(i+1)}) \odot f'^{(i)}(\mathbf{z}^{(i)})$.
Gradientes de pesos y sesgos	Para cada $i = l, l-1, \dots, 2, 1$: $\nabla_{\mathbf{W}^{(i)}} J = \boldsymbol{\delta}^{(i)} (\mathbf{a}^{(i-1)})^\top$ y $\nabla_{\mathbf{b}^{(i)}} J = \boldsymbol{\delta}^{(i)}$.
Actualización de pesos y sesgos	Para cada $i = l, l-1, \dots, 2, 1$: $\mathbf{W}^{(i)} = \mathbf{W}^{(i)} - \eta \nabla_{\mathbf{W}^{(i)}} J$ y $\mathbf{b}^{(i)} = \mathbf{b}^{(i)} - \eta \nabla_{\mathbf{b}^{(i)}} J$ siendo η el ratio de aprendizaje

2.1.2. Teorema de aproximación universal

Con todos estos elementos definidos ya podemos construir una red neuronal que consiga replicar una determinada función. Sin embargo, nada nos asegura que las redes neuronales sean capaces de aproximar cualquier función en la que estemos interesados. El **teorema de aproximación universal** (Cybenko, 1989) [6] establece que una red neuronal con una salida lineal y al menos una capa oculta con una función de activación de “compresión” (como por ejemplo la sigmoide), puede aproximar cualquier función Borel medible entre dos espacios de dimensión finita, dada una cantidad suficiente de unidades ocultas. Posteriormente este resultado ha sido ampliado para incluir otras funciones de activación como la ReLU (Hornik, K. 1991) [20]. Por un lado esto nos asegura que, sin importar la función que intentemos aprender, habrá una red neuronal de propagación directa que será capaz de representarla. El inconveniente es que nada nos asegura que el algoritmo de entrenamiento sea capaz de ‘aprender’ dicha función. Es por esto que muchas de las decisiones a la hora de construir una red neuronal buscan facilitar el proceso de descenso gradiente, como veremos a lo largo de este trabajo con las distintas arquitecturas.

2.2. Representación de palabras

Aunque ya hemos definido cómo crear una red neuronal sencilla, para conseguir que esta procese el lenguaje, debemos desarrollar una forma de representar el texto natural como vectores. Este problema lleva presente en el campo del estudio del lenguaje desde mucho antes de la aparición del aprendizaje profundo. Esta tarea consiste en cierta manera de ‘enseñar’ el significado de cada palabra de forma que nuestro modelo pueda entenderlas y por lo tanto extraer información útil de ellas. Trasladado a las personas, nosotros aprendemos el significado de muchas palabras al asociarlas con elementos físicos: ‘mesa’, ‘cielo’ o ‘río’ son todas palabras que representan algo concreto y es esta realidad física lo que les otorga significado. Sin embargo, nuestro modelo no va a tener ningún tipo de contacto con las entidades materiales que representan las palabras, sólo va a tener acceso a su comportamiento y uso al combinarse con otras palabras. Es de aquí de donde surge la idea principal a la hora de representar palabras, y es estudiarlas según su contexto.

La elección de la representación de las palabras es un factor determinante en el rendimiento de cualquier modelo de NLP, independientemente de la arquitectura utilizada. Tanto los Transformers como los modelos de espacio de estados que veremos en capítulos posteriores dependen de recibir como entrada representaciones vectoriales de las palabras. A continuación exploraremos las principales estrategias para construir estas representaciones.

2.2.1. Bolsa de palabras

Aunque nuestro modelo nunca haya experimentado lo que es un río, sí que puede aprender términos relacionados con la palabra ‘río’, como son ‘agua’, ‘naturaleza’, ‘pez’ etc. Por lo tanto nuestra representación vectorial debe tener como objetivo crear representaciones similares de palabras que suelen compartir contexto. Un primer acercamiento a esta tarea se denomina **bolsa de palabras**, que consiste en representar una palabra según el recuento de sus apariciones. Por un lado podemos contar el número de veces que aparecen por texto o documento de nuestros datos. Usando nuestro conjunto de datos PROGRESO¹, tenemos los siguientes ejemplos:

En la imagen 2.2, podemos ver el número de apariciones de algunas palabras a lo largo de 8 noticias. De esta forma podemos crear vectores de cada palabra que incluyen información de su significado. Por ejemplo ‘gol’ y ‘partido’ aparecen sobre todo en las noticias de deporte local, mientras que ‘trabajo’ o

¹El conjunto de datos PROGRESO fue construido específicamente para este trabajo a partir de noticias del diario *El Progreso* de Lugo, recopiladas entre diciembre de 2023 y marzo de 2024 mediante servicios de AWS (Lambda y S3). Contiene artículos en castellano y gallego clasificados por temática, género e idioma. Su construcción y estructura se describen en detalle en el Apéndice A.2.

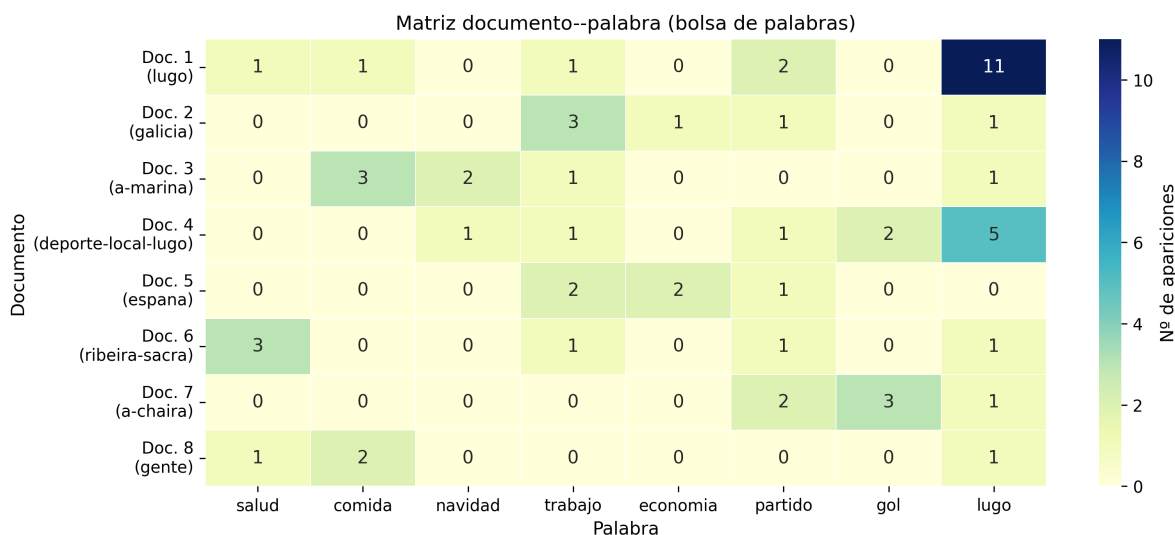


Figura 2.2: Bolsa de palabras: Documento-Palabra

‘economía’ se concentran en las de ámbito económico, social o político. La palabra ‘lugo’, en cambio, aparece de forma transversal en casi todas las noticias, al ser la ciudad de referencia del periódico. Por lo tanto, dado un conjunto de textos D , podemos construir un espacio vectorial de dimensión $|D|$ en el que poder representar palabras y conseguir que aquellas con significados parecidos se encuentren ‘cerca’ en dicho espacio.

La forma más habitual de medir la similitud entre vectores es usar la función coseno:

$$\text{similitud}(\mathbf{v}_1, \mathbf{v}_2) = \cos(\theta) = \frac{\mathbf{v}_1 \cdot \mathbf{v}_2}{\|\mathbf{v}_1\| \|\mathbf{v}_2\|}$$

donde $\mathbf{v}_1 \cdot \mathbf{v}_2$ representa el producto punto entre los vectores y $\|\mathbf{v}_1\|$, $\|\mathbf{v}_2\|$ son sus normas. Nuestro objetivo es que palabras similares formen un ángulo pequeño θ entre sus respectivos vectores, lo que implica que $\cos(\theta)$ se acerque a 1. Para palabras sin relación, buscamos que $\cos(\theta)$ se acerque a 0, y un valor cercano a -1 indica una relación de oposición o anti-similitud.

Una alternativa a la representación por documento consiste en contar coocurrencias de palabras, es decir, cuando coinciden en un intervalo determinado (ventana de contexto). En la figura 2.3 mostramos esta representación, contando aquellos casos en que las palabras coinciden a menos de 4 posiciones.

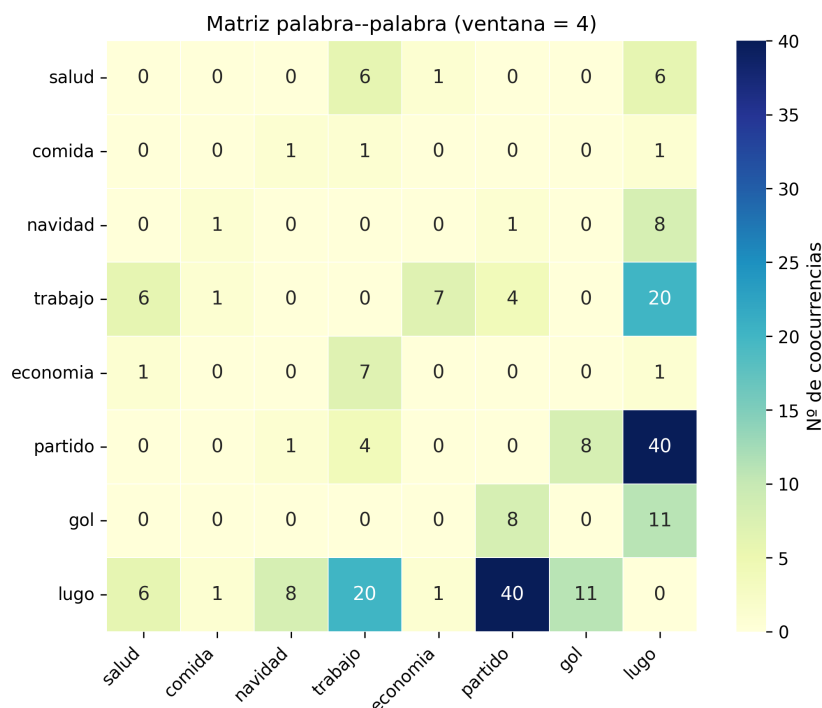


Figura 2.3: Bolsa de palabras: Palabra-Palabra

También aquí vemos cómo los vectores nos aportan información sobre las palabras, pues aquellas relacionadas suelen aparecer juntas, como es el caso de ‘economía’ y ‘trabajo’ o de ‘partido’ y ‘gol’. Con este método estaríamos construyendo un espacio vectorial de dimensión $|V|$ siendo V el conjunto de palabras usado en nuestros datos, comúnmente llamado **vocabulario**.

2.2.2. Tokenización en subpalabras

Las representaciones anteriores asumen que la unidad mínima de texto es la palabra, lo que plantea dos problemas: el vocabulario crece sin límite con el tamaño del corpus, y cualquier palabra no vista en entrenamiento se convierte en un token desconocido (`<unk>`) que destruye información. La **tokenización en subpalabras** resuelve ambos descomponiendo las palabras en unidades más pequeñas y frecuentes. El algoritmo más extendido es **Byte-Pair Encoding** (BPE) [32]: partiendo de un vocabulario de caracteres individuales, fusiona iterativamente el par de tokens adyacentes más frecuente hasta alcanzar un tamaño de vocabulario objetivo. Así, las palabras comunes se representan con un solo token, mientras que las raras se descomponen en fragmentos conocidos (por ejemplo, `tokenización` → `token`, `iza`, `ción`). El resultado es un vocabulario de tamaño fijo y sin palabras fuera de vocabulario, lo que lo convierte en el estándar de los modelos modernos. El tokenizador de GPT-2 [31], que emplearemos en los experimentos de modelado de lenguaje y en el estudio comparativo, usa una variante de BPE a nivel de bytes con un vocabulario de unas 50 000 subpalabras.

2.2.3. Skip-Gram

Aunque ambas aproximaciones cumplen nuestro objetivo de dar representaciones vectoriales a las palabras que contengan información sobre su significado, presentan varios problemas. El más importante es el tamaño de los vectores, pues ya solo con nuestros datos tenemos $|D| = 5430$ y $|V| = 74083$. Además en ambos casos a medida que aumenta nuestro conjunto de datos aumentan las dimensiones de los espacios vectoriales, haciendo que sean muy poco prácticos y cada vez sean más dispersos, con

vectores que contienen muchos valores igual a 0. Esto hace que tengan un rendimiento bastante pobre al ser usados en redes neuronales, pues provocan fácilmente problemas de sobreajuste.

Para solucionar este problema, buscamos crear una representación del lenguaje con vectores mucho más densos y cortos. Un modelo que nos ofrece esto es el **Skip-Gram** [27]. Este modelo en vez de contar cuántas veces la palabra ‘trabajo’ aparece cerca de ‘economía’, busca entrenar un clasificador que resuelva el problema de predicción binaria: “¿Es probable que la palabra ‘trabajo’ aparezca cerca de ‘economía’?”. Una vez resuelto este problema, podemos usar los pesos de clasificación como representaciones de las palabras. Para ello primero establecemos d , el tamaño del espacio vectorial que queremos definir. Después cada palabra w en el vocabulario V se asigna a dos vectores aleatorios: el vector de entrada $v(w)$ y el vector de salida $u(w)$, donde ambos vectores están en $\mathbb{R}^d$. Para una palabra objetivo w_t y una palabra de contexto w_c dentro de una ventana predefinida, el modelo Skip-Gram intenta maximizar la probabilidad de w_c dado w_t usando la función softmax:

$$P(w_c|w_t) = \frac{\exp(u(w_c)^T v(w_t))}{\sum_{w \in V} \exp(u(w)^T v(w_t))}$$

Los vectores $v(w_t)$ y $u(w_c)$ se actualizan para maximizar esta probabilidad mediante descenso de gradiente. Después del entrenamiento, los vectores de palabra capturan relaciones semánticas y sintácticas, con palabras similares teniendo vectores cercanos. Aplicando esta representación al conjunto de datos PROGRESO con $d = 100$, podemos observar las relaciones entre palabras en la figura 2.4:

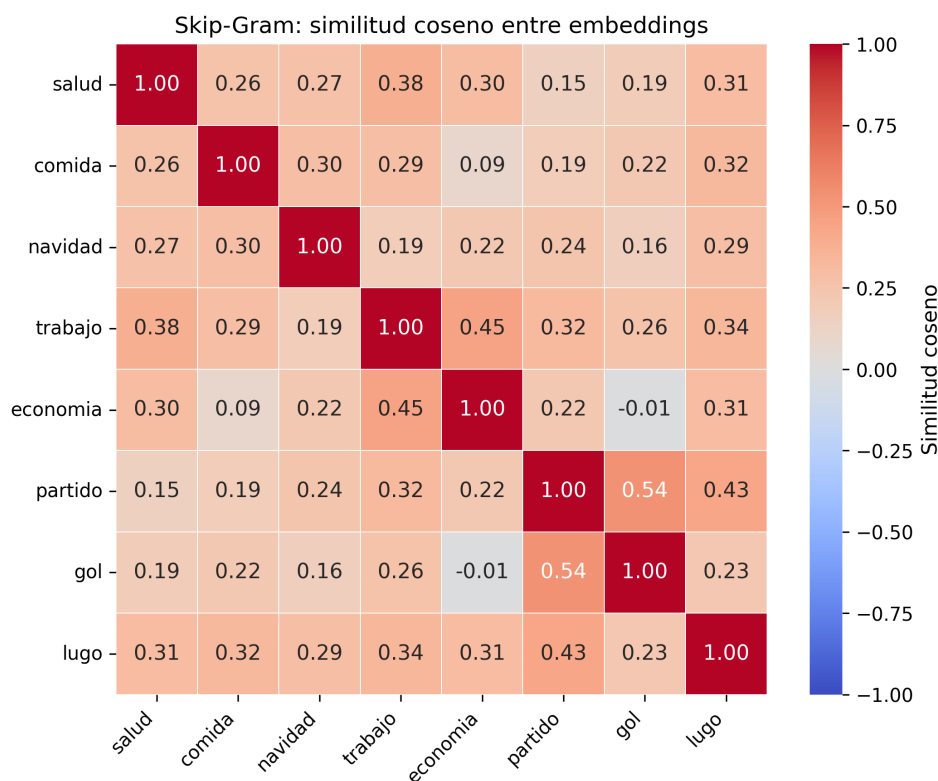


Figura 2.4: Skip-Gram: Mapa de calor de los cosenos

Tal y como observamos, palabras que consideraríamos similares presentan los cosenos más altos, mientras que aquellas muy distintas tienen un valor pequeño o incluso negativo. Algunos ejemplos son ‘partido’ y ‘gol’, o ‘trabajo’ y ‘economía’, que son palabras que van de la mano en la mayoría de contextos. Por otro lado, ‘gol’ y ‘economía’ tienen un coseno prácticamente nulo (ligeramente negativo),

reflejando que aparecen en contextos muy distintos. En la figura 2.5 podemos ver las 5 palabras más cercanas a las anteriores según su representación vectorial.



Figura 2.5: Skip-Gram: Top 5 palabras más similares

Estos vecinos confirman que el modelo ha aprendido relaciones de significado coherentes: las palabras más próximas a ‘gol’ son términos futbolísticos como ‘goles’ o ‘empató’, las más cercanas a ‘salud’ giran en torno al ámbito médico (‘mental’, ‘tabaquismo’) y las de ‘lugo’ son otros topónimos como ‘lucense’, ‘provincia’ u ‘ourense’.

2.2.4. Representación Dinámica (Embeddings)

Como última opción, tenemos la representación dinámica. Este método consiste en usar el proceso de entrenamiento de nuestra red neuronal para que aprenda las mejores representaciones de cada una de las palabras. Dada una red neuronal, en vez de usar como entrada una de las representaciones anteriores, usaríamos *one hot encodings*: vectores de dimensión $|V|$ con 1 en la posición de la palabra y 0 en el resto. A partir de esto se añade una capa de *embedding* que transforma el vector disperso de dimensión $|V|$ a un vector denso de dimensión menor d . Esta capa consiste simplemente en una matriz $\mathbf{E} \in \mathbb{R}^{|V| \times d}$ que multiplicamos por la entrada, cuyos pesos son calculados al mismo tiempo que se entrena el modelo.

La importancia de esta técnica va mucho más allá de un simple preprocesado. Una vez entrenada, la capa de *embedding* captura relaciones semánticas similares a las del modelo Skip-Gram, pero optimizadas para la tarea concreta que se está resolviendo. Además, los pesos obtenidos se pueden reutilizar en otros modelos sin tener que volver a entrenarlos. Este concepto de **aprendizaje de transferencia** (*transfer learning*) es fundamental en las arquitecturas modernas: los modelos preentrenados proporcionan representaciones dinámicas extremadamente ricas, entrenadas sobre cantidades masivas de texto. De la misma manera, las capas de *embedding* son un componente esencial tanto en los Transformers como en los modelos de espacio de estados como Mamba.

2.2.5. Modelos de clasificación binaria

Para ilustrar el impacto de las representaciones de palabras, usaremos un ejemplo sencillo de clasificación de textos: clasificar artículos según si son de opinión o informativos (opinión: 0, información: 1).

1). En la gráfica 2.6 podemos ver la distribución de los artículos según su tipología:

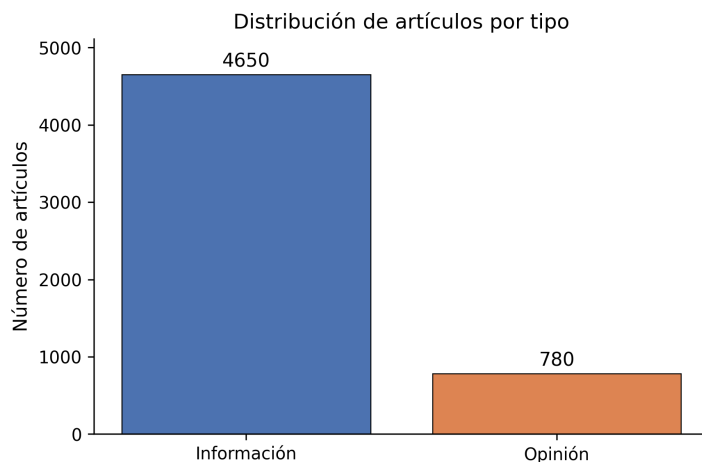


Figura 2.6: Gráfico de barras de los artículos según su tipo

Probamos con tres representaciones de los artículos: una bolsa de palabras, el modelo *Skip-Gram* con $d = 100$, y una representación dinámica. Todas usan la misma arquitectura de red neuronal de propagación directa (capa de entrada con 256 neuronas, capa oculta de 128, ambas con activación ReLU, y salida con función sigmoide), tomando como representación de cada texto la media de los vectores de sus palabras. Obtenemos los siguientes resultados:

Métrica	Valor (%)
Precisión (Entrenamiento)	100.00
Precisión (Test)	96.32

Tabla 2.1: Precisión con bolsa de palabras

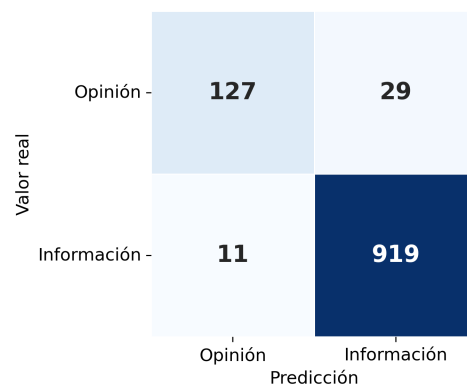


Figura 2.7: Matriz de confusión con bolsa de palabras

Métrica	Valor (%)
Precisión (Entrenamiento)	99.84
Precisión (Test)	94.75

Tabla 2.2: Precisión con Skip-Gram

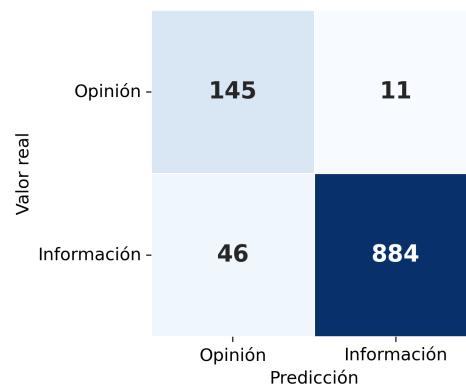


Figura 2.8: Matriz de confusión con Skip-Gram

Métrica	Valor (%)
Precisión (Entrenamiento)	100.00
Precisión (Test)	94.11

Tabla 2.3: Precisión con representación dinámica

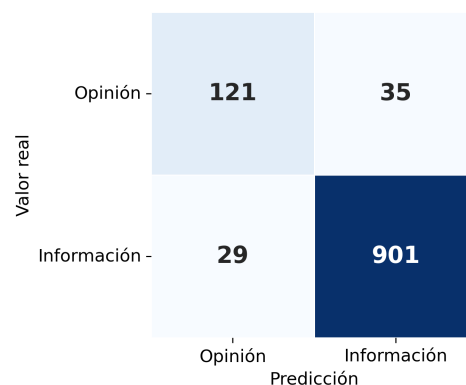


Figura 2.9: Matriz de confusión con representación dinámica

Como el conjunto está desbalanceado (cerca de seis artículos informativos por cada artículo de opinión), un clasificador que predijese siempre la clase mayoritaria alcanzaría ya un 85,64 % de precisión en test, por lo que la métrica realmente informativa es el acierto sobre la clase minoritaria. En este aspecto el modelo Skip-Gram es el mejor con diferencia, clasificando correctamente un 92,95 % de los artículos de opinión, frente al 81,41 % de la bolsa de palabras y el 77,56 % de la representación dinámica. Además es el único que no memoriza por completo el conjunto de entrenamiento (99,84 % frente al 100 % de los otros dos modelos), lo que se traduce en una mejor generalización. La representación dinámica, pese a su elevada precisión global, es la que peor identifica los artículos de opinión y la que más sobreajusta: esto se debe a que la red neuronal, al ser sencilla, tiene que realizar dos tareas simultáneamente, aprender las representaciones de las palabras y clasificar los textos, con un número de ejemplos limitado. Estos resultados ilustran la idea central de la sección: las representaciones semánticas densas como Skip-Gram capturan relaciones de significado que una bolsa de palabras dispersa no recoge, y resultan especialmente valiosas cuando una de las clases dispone de pocos ejemplos.

2.3. Redes Neuronales Recurrentes

Las redes neuronales de propagación directa nos permiten resolver problemas sencillos de NLP. Sin embargo, presentan una limitación fundamental: no tienen en cuenta el **carácter secuencial del lenguaje**. Analizan todas las partes del texto de manera simultánea, sin considerar el orden de las palabras. Este problema es especialmente relevante para comprender por qué se desarrollaron los distintos mecanismos de procesamiento de secuencias que estudiaremos: las redes recurrentes procesan las secuencias paso a paso manteniendo un estado, los Transformers las procesan en paralelo usando atención, y los modelos de espacio de estados mantienen un estado comprimido que se actualiza con cada nuevo elemento. Todos comparten el objetivo de capturar las dependencias entre las palabras de una secuencia, pero difieren radicalmente en cómo lo consiguen.

2.3.1. Red neuronal recurrente simple

Una **red neuronal recurrente** o **RNN** (*Recurrent Neural Network*) es aquella que contiene un ciclo entre sus conexiones, es decir que el valor de alguna neurona depende directa o indirectamente de una de sus salidas anteriores. El caso más sencillo se puede observar en la Figura 2.10.

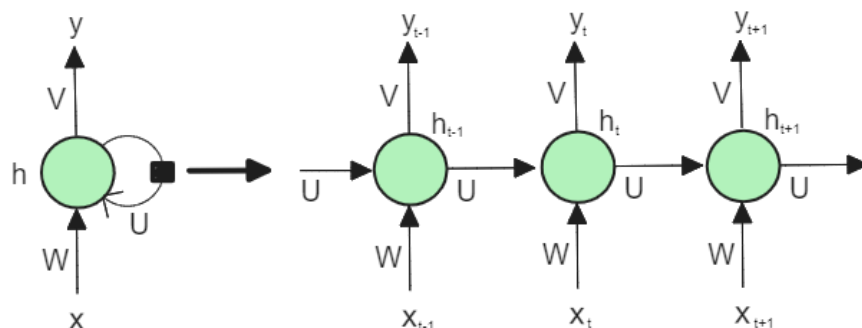


Figura 2.10: Esquema de una red neuronal recurrente sencilla

De esta forma, con una RNN de longitud n y $\forall t \in \{1, \dots, n\}$, tendríamos que para obtener la salida $\mathbf{y}_t$ de una entrada $\mathbf{x}_t$, se calcula el valor de la capa oculta $\mathbf{h}_t$ a partir de $\mathbf{x}_t$ y de su valor justo anterior $\mathbf{h}_{t-1}$:

$$\mathbf{h}_t = g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{x}_t) \quad (2.3)$$

$$\mathbf{y}_t = f(\mathbf{V}\mathbf{h}_t) \quad (2.4)$$

siendo $\mathbf{U}$, $\mathbf{W}$ y $\mathbf{V}$ matrices de pesos con dimensiones correspondientes al tamaño de las capas de entrada, oculta y de salida; g la función de activación y f la de salida, tratándose habitualmente de la función *softmax* cuando se resuelven problemas de modelado de lenguaje. A la hora de entrenar el modelo también se usa el algoritmo de retropropagación, teniendo ahora en cuenta que para evaluar el gradiente del coste con respecto a $\mathbf{h}_t$ necesitamos conocer su influencia sobre la salida t y todas aquellas posteriores.

Es importante destacar la estructura de la ecuación 2.3, pues contiene los elementos fundamentales que reaparecerán en las arquitecturas posteriores: una transformación del estado anterior ($\mathbf{U}\mathbf{h}_{t-1}$), una incorporación de la nueva entrada ($\mathbf{W}\mathbf{x}_t$), y una no linealidad (g). Como veremos, los modelos de espacio de estados de los que trataremos en el capítulo 6 comparten esta estructura recurrente, pero prescindiendo de la no linealidad en la dinámica del estado, lo que les permite una reformulación eficiente como convolución.

2.3.2. Modelado de lenguaje con RNNs

Una vez presentada la arquitectura de una RNN, podemos estudiar su uso para modelar el lenguaje. **Modelar el lenguaje** consiste en calcular la probabilidad de una palabra o secuencia de palabras subsiguientes dada una secuencia de palabras anteriores. De esta forma no buscamos solo predecir la siguiente palabra, sino entender el lenguaje y generar textos que sean coherentes y correctos. Por ejemplo, dado el contexto ‘En Lugo se come’, si queremos saber cómo de probable es la palabra ‘bien’, calcularíamos:

$$P(\text{bien} | \text{En Lugo se come}) \quad (2.5)$$

Lo mismo se puede hacer para calcular la probabilidad de una secuencia entera:

$$P(w_{1:n}) = \prod_{i=1}^n P(w_i | w_{<i}) \quad (2.6)$$

Las RNN permiten procesar la entrada palabra a palabra, manteniendo la información de la anterior capa oculta, permitiendo conservar el contexto de todas las palabras anteriores. Para entrenarlas usamos un algoritmo de auto-supervisión: las propias secuencias de palabras son su propia supervisión. Para cada paso t usamos como entrada la secuencia hasta el elemento t y buscamos predecir la verdadera palabra en $t + 1$, usando la entropía cruzada entre la distribución de probabilidad predicha y la real.

El modelado de lenguaje es la tarea fundamental sobre la que se preentrenan la gran mayoría de los modelos modernos, desde GPT (que predice la siguiente palabra de izquierda a derecha) hasta BERT (que predice palabras enmascaradas en cualquier posición). Comprender esta tarea es esencial para entender el posterior proceso de preentrenamiento.

2.3.3. El problema del desvanecimiento del gradiente

Aunque presentan una mejora con respecto a las redes de propagación directa, las RNNs sufren un problema fundamental a la hora de capturar dependencias entre palabras muy alejadas en la secuencia. Este problema, conocido como **desvanecimiento del gradiente** (*vanishing gradient*), es una de las principales motivaciones detrás de todas las arquitecturas que estudiaremos en este trabajo.

Para entender por qué ocurre, consideremos el gradiente de la función de coste J con respecto a los parámetros en un paso temporal temprano t . Dado que $\mathbf{h}_t$ influye en las salidas de todos los pasos posteriores, por la regla de la cadena tenemos:

$$\frac{\partial J}{\partial \mathbf{h}_t} = \sum_{k=t}^n \frac{\partial J_k}{\partial \mathbf{h}_k} \prod_{j=t+1}^k \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}} \quad (2.7)$$

El término problemático es el producto $\prod_{j=t+1}^k \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}$, que implica multiplicar repetidamente matrices jacobianas. Si los valores propios de estas matrices son menores que 1, el producto tiende exponencialmente a cero a medida que $k - t$ crece. Esto significa que la influencia de un token sobre otro muy lejano se diluye completamente durante el entrenamiento, impidiendo que el modelo aprenda dependencias a larga distancia.

A mayores, los pesos que determinan los valores de la capa oculta tienen que optimizar dos tareas a la vez: producir el mejor valor posible para el momento actual y además transmitir información relevante para los momentos futuros. Esta doble tarea hace que la optimización sea especialmente difícil.

Este problema ha sido abordado de maneras fundamentalmente distintas por las diferentes familias de arquitecturas: las LSTM introducen puertas para controlar el flujo de información; los Transformers eliminan la recurrencia y permiten conexiones directas entre cualquier par de tokens mediante la atención; y los modelos de espacio de estados como S4 utilizan inicializaciones especiales (como la matriz HiPPO) que preservan matemáticamente la información del historial.

2.3.4. LSTM

Una de las primeras arquitecturas diseñadas específicamente para mitigar el desvanecimiento del gradiente es **LSTM** (*Long Short-Term Memory*) [19]. La idea central de LSTM es separar el almacenamiento de información a largo plazo del procesamiento a corto plazo, mediante la introducción de un **estado de celda** $\mathbf{c}_t$ (independiente del estado oculto $\mathbf{h}_t$) y un sistema de **puertas** (*gates*) que controlan el flujo de información.

Las puertas son capas de propagación directa con función sigmoide, seguidas de una multiplicación elemento a elemento con la capa sobre la que se aplican. De esta forma, valores en la capa alineados con los de la puerta se mantienen, mientras que el resto son eliminados. Dados $\mathbf{h}_t \forall t \in \{1, \dots, n\}$ los estados ocultos, $\mathbf{c}_t \forall t \in \{1, \dots, n\}$ los estados de celda, y $\mathbf{U}_g, \mathbf{W}_g \forall g \in \{f, i, r, o\}$ las matrices de pesos, tenemos:

- **Forget Gate:** Decide qué información del estado de celda se descarta.

$$\mathbf{f}_t = \sigma(\mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{W}_f \mathbf{x}_t) \quad (2.8)$$

$$\mathbf{k}_t = \mathbf{c}_{t-1} \odot \mathbf{f}_t \quad (2.9)$$

- **Input Gate:** Selecciona qué nueva información se incorpora al estado de celda.

$$\mathbf{i}_t = \sigma(\mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{W}_i \mathbf{x}_t) \quad (2.10)$$

$$\tilde{\mathbf{c}}_t = \tanh(\mathbf{U}_r \mathbf{h}_{t-1} + \mathbf{W}_r \mathbf{x}_t) \quad (2.11)$$

$$\mathbf{j}_t = \tilde{\mathbf{c}}_t \odot \mathbf{i}_t \quad (2.12)$$

$$\mathbf{c}_t = \mathbf{j}_t + \mathbf{k}_t \quad (2.13)$$

- **Output Gate:** Determina qué parte del estado de celda se usa como salida.

$$\mathbf{o}_t = \sigma(\mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{W}_o \mathbf{x}_t) \quad (2.14)$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \quad (2.15)$$

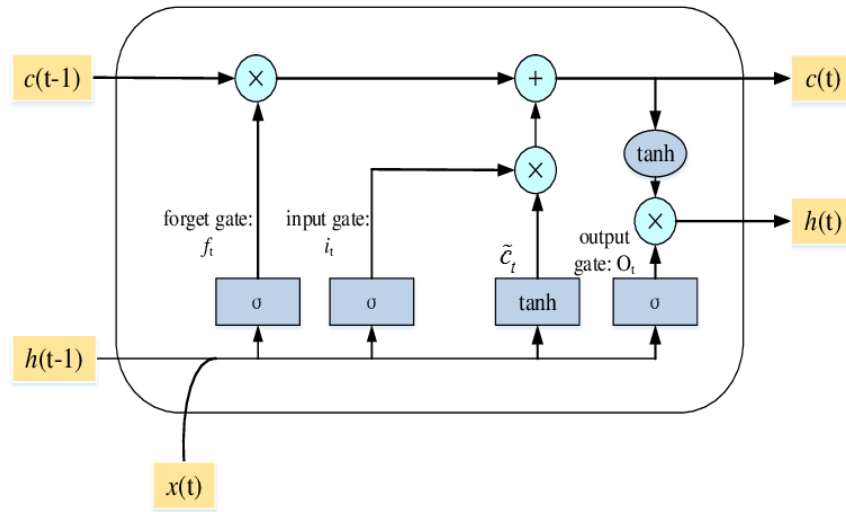


Figura 2.11: Esquema de una unidad LSTM

La clave del diseño LSTM es que el estado de celda $\mathbf{c}_t$ se actualiza de manera aditiva ($\mathbf{c}_t = \mathbf{j}_t + \mathbf{k}_t$), en lugar de multiplicativa como en la RNN simple. Esto crea un camino directo para el flujo del gradiente a través de los pasos temporales, mitigando (aunque no eliminando completamente) el problema del desvanecimiento. El mecanismo de puertas de la LSTM es un concepto que reaparece, con distintas formas, en las arquitecturas posteriores. Por ejemplo, el mecanismo de selección de Mamba puede entenderse como una generalización del sistema de puertas, donde el paso de discretización Δ_t controla simultáneamente cuánta información del estado anterior se preserva y cuánta nueva información se incorpora.

En todo tipo de redes neuronales recurrentes, incluidas las LSTM, al procesar los datos de manera secuencial, tienden a obtener más información sobre las últimas palabras de la secuencia. Es por este motivo que en determinadas tareas, como la clasificación de textos, son interesantes las arquitecturas **bidireccionales**. En estas se procesan las secuencias de izquierda a derecha y de derecha a izquierda, concatenando después los resultados. De esta manera se obtiene información de cómo interactúan las palabras en ambos sentidos, mejorando la comprensión general del texto. Este concepto de bidireccionalidad también será central en modelos como BERT, que lo implementa de una forma diferente mediante el enmascaramiento de palabras.

2.3.5. Encoder-Decoder

Hasta ahora, solo hemos tratado problemas que implican clasificar el texto o generar nuevas palabras a partir del texto. Sin embargo no nos sirven si queremos generar otra secuencia de distinto tamaño a la entrada, como puede ser un problema de traducción o de resumen de un texto. La manera más habitual de afrontar este tipo de tareas es con una arquitectura **encoder-decoder** (codificador-descifrador).

La idea detrás de esta arquitectura es usar dos redes neuronales: un encoder que tome una secuencia y cree una representación comprimida de ella, llamada contexto; y un decoder que interprete el contexto y genere la secuencia de salida. Pongamos el ejemplo de un problema de traducción de español a gallego. Nuestro objetivo sería que, dada la secuencia en español al encoder, este la interprete como cualquier red neuronal recurrente y finalmente genere un contexto. Después el decoder sería el encargado de, recibiendo como entrada un separador (habitualmente se usa ' $\langle s \rangle$ '), generar de manera **autorregresiva** las palabras de la traducción: para generar cada palabra toma como entrada la anterior palabra generada y usa tanto la información de la anterior capa oculta como la del contexto. Podemos ver una simplificación de la arquitectura en la Figura 2.12.

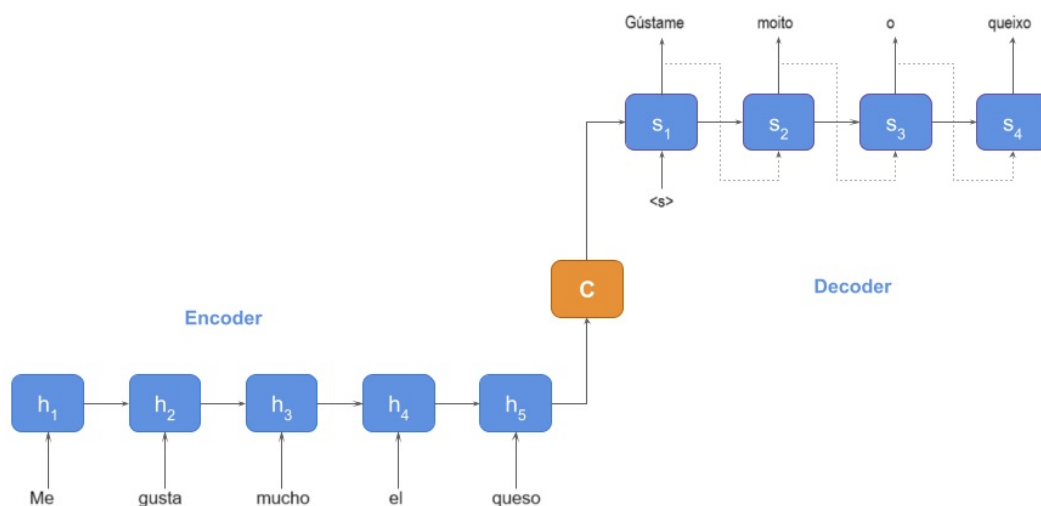


Figura 2.12: Esquema simplificado de arquitectura Encoder-Decoder

La estructura encoder-decoder es un patrón recurrente en las arquitecturas de aprendizaje profundo que trasciende a las redes recurrentes. El Transformer original mantiene esta estructura, y muchos modelos modernos pueden entenderse como variaciones de este paradigma. Incluso los modelos que prescinden del decoder (como BERT, que solo usa el encoder) o del encoder (como GPT, que solo usa el decoder) heredan conceptos fundamentales de esta arquitectura.

2.3.6. Mecanismo de atención

Una vez tenemos el modelo encoder-decoder, nos falta decidir cómo construir el contexto. Nuestra primera opción puede ser simplemente usar el último valor de la capa oculta del encoder, sin embargo esto presenta problemas. El decoder, a partir del contexto, debe inferir el significado del texto de entrada, por lo que tomando sólo el último valor la información del principio de la secuencia se irá diluyendo. Esto no es más que otra manifestación del problema del desvanecimiento del gradiente descrito en la sección 2.3.3.

Para solucionar esto, Bahdanau et al. (2014) [3] proponen el mecanismo de **atención**, que busca conservar la información de todas las capas ocultas del encoder, no solo de la última. En vez de construir un vector de contexto fijo, tendremos un vector dinámico que otorgue al decoder la información más relevante dependiendo de qué palabra esté generando.

La forma más sencilla de conseguir esto es con la **atención de producto escalar**. Se busca puntuar la similitud entre el estado oculto anterior del decoder y todos los estados ocultos del encoder calculando el producto escalar entre ellos. Con estas puntuaciones se aplica un *softmax* para crear vectores de peso, y finalmente se hace una media ponderada de los estados ocultos del encoder. Dadas $\mathbf{h}_j$ los estados ocultos del encoder y $\mathbf{s}_i$ los del decoder:

$$\alpha_{ij} = \text{softmax}(\mathbf{s}_{i-1} \cdot \mathbf{h}_j) = \frac{\exp(\mathbf{s}_{i-1} \cdot \mathbf{h}_j)}{\sum_k \exp(\mathbf{s}_{i-1} \cdot \mathbf{h}_k)} \quad (2.16)$$

$$\mathbf{c}_i = \sum_j \alpha_{ij} \mathbf{h}_j \quad (2.17)$$

Los pesos α_{ij} son interpretables: nos indican cuánta ‘atención’ está prestando el decoder en el paso i a cada palabra j del texto de entrada. Por ejemplo, en una tarea de traducción, cuando el decoder está generando un sustantivo en el idioma objetivo, los pesos de atención tenderán a concentrarse en el sustantivo correspondiente del idioma de entrada.

Este mecanismo de atención es el antecesor directo de la auto-atención del Transformer. La diferencia fundamental, como veremos en el siguiente capítulo, es que en la auto-atención cada palabra atiende al *resto de palabras de su propia secuencia*, en lugar de a los estados ocultos de otra red. Esta evolución es lo que permite prescindir completamente de la recurrencia.

A la hora de entrenar el encoder-decoder hay que prestar especial atención al decoder. Si solo entrenásemos proporcionando la secuencia de entrada, el decoder se entrenaría a partir de sus propias predicciones. Esto puede provocar que el decoder se desvíe cada vez más al generar palabras erróneas. Por este motivo en el entrenamiento se suele dar como entrada del decoder la palabra correcta en vez de su predicción, técnica conocida como *teacher forcing*.

Con la arquitectura encoder-decoder y el mecanismo de atención, las redes recurrentes alcanzaron resultados destacados en tareas como la traducción automática. Sin embargo, su naturaleza secuencial seguía limitando tanto la velocidad de entrenamiento como la capacidad de capturar dependencias entre posiciones muy alejadas en la secuencia. Estas limitaciones motivaron la búsqueda de arquitecturas que pudieran procesar toda la secuencia en paralelo, dando lugar a la arquitectura Transformer que se presenta en el siguiente capítulo.

Capítulo 3

Arquitectura Transformer

Las arquitecturas recurrentes descritas en el capítulo anterior comparten una restricción fundamental: su naturaleza intrínsecamente secuencial. El cálculo del estado oculto en el instante t no puede llevarse a cabo hasta que no se ha completado el del instante $t - 1$, lo que establece una dependencia causal estricta a lo largo de toda la secuencia. Esta característica tiene dos consecuencias importantes que limitan su aplicabilidad a largo plazo. En primer lugar, impide la paralelización del proceso de entrenamiento: dado que los estados ocultos deben computarse de forma estrictamente ordenada, no es posible distribuir el cálculo de una misma secuencia entre múltiples unidades de procesamiento, lo que reduce drásticamente el aprovechamiento del hardware moderno basado en GPUs y limita el procesamiento en lotes (*batching*) de los ejemplos. En segundo lugar, a pesar de las mejoras introducidas por la LSTM mediante sus mecanismos de puertas, el problema del desvanecimiento del gradiente sigue presente en la práctica para secuencias suficientemente largas: el gradiente debe propagarse a través de cientos o miles de pasos temporales, y aunque las puertas atenúan el problema, no lo eliminan por completo. Como consecuencia, el contexto efectivo que estos modelos son capaces de capturar entre posiciones muy alejadas dentro de una misma secuencia queda severamente limitado.

Es por estos motivos que en 2017, Vaswani et al. presentan la arquitectura **Transformer** en el artículo *Attention is All You Need* [39], que ha revolucionado el campo del aprendizaje profundo y, en particular, el del procesamiento del lenguaje natural. La propuesta supone una ruptura radical con el paradigma recurrente: en lugar de procesar la secuencia paso a paso, el Transformer opera sobre todos los elementos de la secuencia de forma simultánea, lo que permite una paralelización completa durante el entrenamiento. La arquitectura Transformer, que podemos ver en la Figura 3.1, mantiene la estructura encoder-decoder propia de los modelos secuencia a secuencia, pero introduce dos novedades fundamentales. La primera es la eliminación completa de la recurrencia, sustituyendo las conexiones temporales por operaciones que actúan globalmente sobre la representación de la secuencia. La segunda, y más importante, es el mecanismo de **auto-atención** (*self-attention*), que constituye el núcleo computacional de la arquitectura. Este mecanismo, que generaliza la atención de Bahdanau descrita en la sección 2.3.6, permite que cada posición de la secuencia atienda directamente a cualquier otra posición con un coste computacional constante, independientemente de la distancia que las separe. De este modo, el modelo es capaz de capturar dependencias de largo alcance de forma directa y eficiente, sin depender de conexiones recurrentes intermedias que puedan degradar la señal del gradiente.

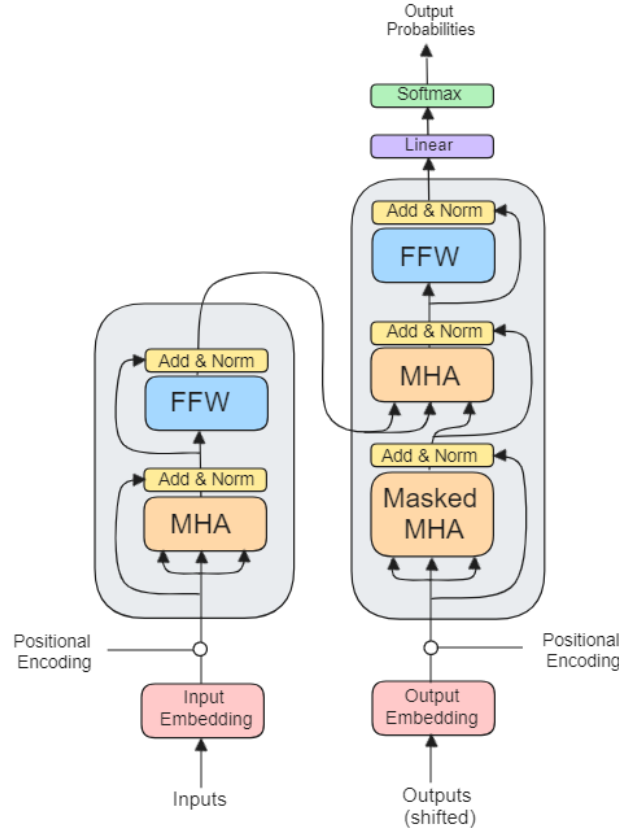


Figura 3.1: Esquema de la arquitectura Transformer (Vaswani et al., 2017)

3.1. Auto-atención

Tal y como vimos en la sección 2.3.6, el mecanismo de atención se basa en evaluar la relación entre un elemento y un conjunto de otros elementos para comprender su relevancia en el contexto actual. En la atención de Bahdanau, esta comparación se realizaba entre los estados del decoder y los del encoder, es decir entre dos secuencias distintas. La **auto-atención** generaliza esta idea: el conjunto con el que se compara cada palabra son el *resto de palabras de la propia secuencia*. Esto permite que cada palabra acceda directamente a la información de todas las demás, sin necesidad de intermediarios recurrentes.

Para llevar a cabo este mecanismo recurrimos de nuevo al producto escalar y la función softmax. De esta forma, si tuviéramos un modelo con una única capa de atención, tendríamos:

$$\mathbf{y}_i = \sum_{j \leq i} \alpha_{ij} \mathbf{x}_j = \sum_{j \leq i} \text{softmax}(\mathbf{x}_i \cdot \mathbf{x}_j) \mathbf{x}_j \quad \forall j \leq i \quad (3.1)$$

Como se puede observar, a diferencia de en una RNN, aquí no necesitamos calcular las salidas anteriores para seguir calculando el resto. Esto permite la paralelización del cálculo, lo que reduce sustancialmente los tiempos de entrenamiento. Sin embargo, este mecanismo de atención sigue siendo demasiado sencillo como para mejorar a las RNNs en el modelado de lenguaje, pues cada entrada desempeña simultáneamente tres roles distintos que conviene separar.

3.1.1. Queries, Keys y Values

Si observamos la ecuación 3.1, veremos que la entrada $\mathbf{x}$ desempeña tres funciones diferentes:

- **Query (q)**: Es la palabra que hace de foco de la atención al ser comparada con el resto. Responde a la pregunta: “¿qué información necesito?”
- **Key (k)**: Es la palabra con la que se compara el foco de atención. Responde a: “¿qué información tengo para ofrecer?”
- **Value (v)**: Es la palabra a partir de la que se calcula la salida. Contiene la información que se extrae cuando la query y la key son compatibles.

Forzar a un mismo vector a cumplir simultáneamente estos tres roles limita la expresividad del modelo. La solución es crear **proyecciones lineales independientes** para cada rol. Dada una entrada $\mathbf{x}_i \in \mathbb{R}^d$, definimos las matrices de pesos $\mathbf{W}^Q, \mathbf{W}^K \in \mathbb{R}^{d \times d_k}$ y $\mathbf{W}^V \in \mathbb{R}^{d \times d_v}$ (donde d_k y d_v son las dimensiones de las queries/keys y values respectivamente) de manera que:

$$\mathbf{q}_i = \mathbf{W}^Q \mathbf{x}_i; \quad \mathbf{k}_i = \mathbf{W}^K \mathbf{x}_i; \quad \mathbf{v}_i = \mathbf{W}^V \mathbf{x}_i \quad (3.2)$$

Sustituyendo en la ecuación 3.1:

$$\mathbf{y}_i = \sum_{j \leq i} \alpha_{ij} \mathbf{v}_j = \sum_{j \leq i} \text{softmax} \left(\frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}} \right) \mathbf{v}_j \quad \forall j \leq i \quad (3.3)$$

3.1.2. Factor de escala

El término $\sqrt{d_k}$ que aparece en el denominador de la ecuación 3.3 merece una explicación más detallada, ya que su presencia es crucial para la estabilidad del entrenamiento. Cuando las componentes de $\mathbf{q}$ y $\mathbf{k}$ son variables aleatorias independientes con media 0 y varianza 1, el producto escalar $\mathbf{q} \cdot \mathbf{k} = \sum_{i=1}^{d_k} q_i k_i$ es una suma de d_k variables aleatorias, cada una con media 0 y varianza 1. Por lo tanto:

$$\mathbb{E}[\mathbf{q} \cdot \mathbf{k}] = 0, \quad \text{Var}(\mathbf{q} \cdot \mathbf{k}) = d_k \quad (3.4)$$

Esto implica que, a medida que d_k crece, los valores del producto escalar se dispersan cada vez más, tomando valores con una magnitud del orden de $\sqrt{d_k}$. Valores muy grandes o muy pequeños en la entrada de la función softmax producen distribuciones de atención cercanas a vectores *one-hot* (toda la atención concentrada en una sola posición), lo que satura los gradientes y dificulta el entrenamiento. Al dividir por $\sqrt{d_k}$, normalizamos la varianza del producto escalar a 1, manteniendo los valores en un rango donde el softmax produce gradientes útiles. Este tipo de atención se denomina **atención de producto escalar escalado** (*scaled dot-product attention*).

3.1.3. Formulación matricial y complejidad

Como no es necesario calcular $\mathbf{y}_{i-1}$ para obtener $\mathbf{y}_i$, podemos procesar secuencias enteras almacenándolas en una única matriz $\mathbf{X} \in \mathbb{R}^{L \times d}$, donde L es la longitud de la secuencia y cada fila es la representación de un token. Al multiplicar por las matrices de pesos obtenemos:

$$\mathbf{Q} = \mathbf{XW}^Q \in \mathbb{R}^{L \times d_k}; \quad \mathbf{K} = \mathbf{XW}^K \in \mathbb{R}^{L \times d_k}; \quad \mathbf{V} = \mathbf{XW}^V \in \mathbb{R}^{L \times d_v} \quad (3.5)$$

Con estas matrices, el mecanismo de auto-atención se expresa de forma compacta como:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax} \left(\frac{\mathbf{QK}^T}{\sqrt{d_k}} \right) \mathbf{V} \quad (3.6)$$

Es fundamental analizar la complejidad computacional de esta operación, puesto que es la principal limitación del Transformer. El cálculo se descompone en tres pasos:

1. **Cálculo de la matriz de atención $\mathbf{S} = \mathbf{QK}^T \in \mathbb{R}^{L \times L}$:** requiere $\mathcal{O}(L^2 \cdot d_k)$ operaciones y $\mathcal{O}(L^2)$ de memoria para almacenar la matriz completa.
2. **Softmax por filas:** se aplica sobre cada una de las L filas de $\mathbf{S}$, con coste $\mathcal{O}(L^2)$.
3. **Multiplicación por $\mathbf{V}$:** $\text{softmax}(\mathbf{S}) \cdot \mathbf{V} \in \mathbb{R}^{L \times d_v}$, con coste $\mathcal{O}(L^2 \cdot d_v)$.

El coste total es $\mathcal{O}(L^2 \cdot d)$ en cómputo y $\mathcal{O}(L^2)$ en memoria (dominado por la materialización de la matriz de atención). Para $L = 512$ y $d = 64$, la matriz de atención tiene 262,144 entradas; para $L = 4096$ tiene más de 16 millones, y para $L = 16384$ supera los 268 millones. Esto hace que el procesamiento de secuencias largas con atención completa sea muy costoso, y motiva las variantes eficientes de los capítulos siguientes.

3.1.4. Enmascaramiento causal

Al considerar el conjunto de la secuencia en $\mathbf{QK}^T$, se tiene en cuenta también los tokens posteriores al actual. Esto puede ser un problema dependiendo de la tarea. Si el objetivo es aprender relaciones entre palabras (como en un encoder), no hay inconveniente. Pero si queremos predecir la siguiente palabra (como en un decoder o en modelado de lenguaje), dar acceso a los tokens futuros haría la tarea trivial.

Para solucionar esto se aplica un **enmascaramiento causal** (*causal masking*). Definimos una matriz de máscara $\mathbf{M} \in \mathbb{R}^{L \times L}$ como:

$$M_{ij} = \begin{cases} 0 & \text{si } j \leq i \\ -\infty & \text{si } j > i \end{cases} \quad (3.7)$$

que se suma a la matriz de puntuaciones antes de aplicar el softmax:

$$\text{MaskedAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax} \left(\frac{\mathbf{QK}^T + \mathbf{M}}{\sqrt{d_k}} \right) \mathbf{V} \quad (3.8)$$

Al aplicar softmax, los valores $-\infty$ se convierten en 0, garantizando que cada posición i solo atienda a las posiciones $j \leq i$. La máscara convierte la matriz de atención (que era densa) en triangular inferior, eliminando toda la información del futuro.

Cabe notar que este enmascaramiento no reduce la complejidad asintótica: aunque la mitad superior de la matriz es cero, el cálculo completo de $\mathbf{QK}^T$ sigue siendo necesario en la implementación estándar.

3.1.5. La atención como recuperación de información suave

Es útil interpretar la auto-atención desde la perspectiva de la **recuperación de información**. En un sistema clásico de bases de datos, dada una consulta (*query*), buscamos en un índice de claves (*keys*) y devolvemos el valor (*value*) asociado a la clave que coincide exactamente. La auto-atención funciona de manera análoga, pero con una *coincidencia suave*: en vez de seleccionar un único valor, calcula una media ponderada de todos los valores, donde los pesos reflejan la compatibilidad entre la query y cada key.

Esta interpretación también ayuda a entender por qué los Transformers son más expresivos que las RNN para ciertas tareas. En una RNN, para que el token en la posición t acceda a la información del token en la posición 1, la información debe fluir a través de $t - 1$ pasos intermedios, degradándose en cada uno. En un Transformer, la query del token t puede acceder directamente a la key y al value del token 1 en una sola operación, con un “camino” de longitud constante. Esto explica la capacidad superior de los Transformers para capturar dependencias a larga distancia, aunque tenga un coste de complejidad cuadrático.

3.2. Auto-atención multicabeza

A la hora de estudiar el lenguaje, nos damos cuenta de que las palabras se relacionan entre sí de múltiples formas simultáneamente. Por ejemplo en la oración:

‘Mientras iba andando por el parque, me distraje mirando un pájaro y me tropecé con un arbusto’

encontramos relaciones temporales (los hechos son simultáneos), espaciales (se referencia un lugar y elementos presentes en él) y causales (un hecho da lugar a otro). Una única cabeza de atención comprime todas estas relaciones en un único conjunto de pesos α_{ij} , obligándola a encontrar un compromiso que inevitablemente pierde información.

La arquitectura Transformer soluciona esto mediante la **auto-atención multicabeza** (*multihead self-attention*), que consiste en ejecutar r capas de auto-atención en paralelo, cada una con sus propios parámetros, de manera que cada “cabeza” pueda especializarse en un tipo diferente de relación.

3.2.1. Formulación

Para cada cabeza $i \in \{1, \dots, r\}$, definimos matrices de proyección independientes:

$$\mathbf{W}_i^Q \in \mathbb{R}^{d \times d_k}, \quad \mathbf{W}_i^K \in \mathbb{R}^{d \times d_k}, \quad \mathbf{W}_i^V \in \mathbb{R}^{d \times d_v} \quad (3.9)$$

y calculamos:

$$\mathbf{head}_i = \text{Attention}(\mathbf{XW}_i^Q, \mathbf{XW}_i^K, \mathbf{XW}_i^V) \in \mathbb{R}^{L \times d_v} \quad (3.10)$$

Las r salidas se concatenan y se proyectan de vuelta a la dimensión original d mediante una matriz de salida $\mathbf{W}^O \in \mathbb{R}^{r d_v \times d}$:

$$\text{MultiHead}(\mathbf{X}) = (\mathbf{head}_1 \oplus \mathbf{head}_2 \oplus \dots \oplus \mathbf{head}_r) \mathbf{W}^O \quad (3.11)$$

3.2.2. Dimensiones y coste

Un aspecto crucial del diseño multicabeza es la elección de las dimensiones. Si tomásemos $d_k = d_v = d$ para cada cabeza, el coste total sería r veces mayor que el de una única cabeza de atención de dimensión d . Para mantener el coste computacional constante, se reduce la dimensión de cada cabeza proporcionalmente al número de cabezas: $d_k = d_v = d/r$.

En el Transformer original, se usan $r = 8$ cabezas con $d = 512$, resultando en $d_k = d_v = 64$ por cabeza. Así, el coste total de la atención multicabeza es:

$$\mathcal{O} \left(r \cdot L^2 \cdot \frac{d}{r} + L \cdot r \cdot \frac{d}{r} \cdot d \right) = \mathcal{O}(L^2 \cdot d + L \cdot d^2) \quad (3.12)$$

que es equivalente al de una única cabeza de dimensión d , más el coste de la proyección de salida. El primer término domina cuando $L > d$ (secuencias largas) y el segundo cuando $d > L$ (modelos grandes con secuencias cortas).

3.2.3. Interpretabilidad de las cabezas

Estudios empíricos han demostrado que las distintas cabezas de atención se especializan efectivamente en capturar diferentes tipos de relaciones lingüísticas. Se han identificado cabezas que se focalizan en relaciones sintácticas (como la dependencia sujeto-verbo), cabezas que capturan correferencias (qué pronombre se refiere a qué sustantivo), cabezas que atienden a posiciones relativas fijas (el token inmediatamente anterior o posterior) y cabezas que distribuyen su atención de forma más uniforme, capturando contexto global. Esta especialización emergente es uno de los factores que hacen a la atención multicabeza tan efectiva.

Sin embargo, también se ha observado que algunas cabezas son redundantes o contribuyen poco al rendimiento final. Esto ha motivado investigaciones sobre la poda de cabezas (*head pruning*) y también diseños más eficientes que reducen el número de parámetros sin sacrificar rendimiento, como la *Multi-Query Attention* y la *Grouped-Query Attention*.

3.3. Bloque Transformer

Una vez definido el mecanismo de atención multicabeza, podemos describir el bloque completo de la arquitectura Transformer. Cada bloque combina la capa de atención con una red de propagación directa y dos mecanismos de estabilización del entrenamiento: las conexiones residuales y la normalización de capa.

3.3.1. Capa de propagación directa posicional

Después de la capa de atención multicabeza, se aplica una **red de propagación directa posicional** (*position-wise feedforward network*, FFN). Esta red se aplica de forma independiente e idéntica a cada posición de la secuencia. En el Transformer original, consiste en dos transformaciones lineales con una activación ReLU intermedia:

$$\text{FFN}(\mathbf{z}) = \text{ReLU}(\mathbf{z}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2 \quad (3.13)$$

donde $\mathbf{W}_1 \in \mathbb{R}^{d \times d_{ff}}$ y $\mathbf{W}_2 \in \mathbb{R}^{d_{ff} \times d}$, siendo d_{ff} la dimensión intermedia (en el Transformer original, $d_{ff} = 2048 = 4d$).

La función de esta capa es doble. En primer lugar, introduce **no linealidad** en el modelo. Aunque la atención computa pesos de forma no lineal (a través del softmax), la operación sobre los valores es una combinación lineal ponderada. Sin la FFN, apilar múltiples bloques de atención sería equivalente a una sola capa de atención con pesos diferentes, sin ganar expresividad. En segundo lugar, la expansión a dimensión $d_{ff} = 4d$ seguida de una contracción de vuelta a d actúa como un **cuello de botella informativo** (*bottleneck*), permitiendo al modelo procesar la información en un espacio de mayor dimensión antes de comprimirla de nuevo.

Es importante notar que esta capa es “posicional” (*position-wise*) porque se aplica independientemente a cada token. A diferencia de la capa de atención, que mezcla información entre posiciones, la FFN solo transforma la representación de cada posición individualmente. Se puede entender como una capa que procesa “en profundidad” lo que la atención ha mezclado “en anchura”.

3.3.2. Conexiones residuales

Las **conexiones residuales** (*residual connections* o *skip connections*), introducidas por He et al. (2015) [18] en el contexto de redes convolucionales profundas, consisten en sumar la entrada de una subcapa directamente a su salida:

$$\mathbf{z} = \mathbf{x} + \text{Subcapa}(\mathbf{x}) \quad (3.14)$$

Su impacto en el entrenamiento es profundo. Consideremos la retropropagación del gradiente a través de N_{capas} bloques residuales. Si denotamos la salida del bloque l como $\mathbf{x}_l = \mathbf{x}_{l-1} + F_l(\mathbf{x}_{l-1})$, donde F_l representa la transformación del bloque, podemos expandir:

$$\mathbf{x}_{N_{\text{capas}}} = \mathbf{x}_0 + \sum_{l=1}^{N_{\text{capas}}} F_l(\mathbf{x}_{l-1}) \quad (3.15)$$

El gradiente con respecto a $\mathbf{x}_0$ contiene siempre el término $\frac{\partial \mathbf{x}_{N_{\text{capas}}}}{\partial \mathbf{x}_0} = \mathbf{I} + \dots$, donde $\mathbf{I}$ es la matriz identidad. Este término garantiza que el gradiente nunca desaparezca completamente, independientemente de la profundidad de la red. Es conceptualmente análogo a la actualización aditiva del estado de celda en la LSTM, y cumple la misma función: crear un “atajo” para el flujo del gradiente.

3.3.3. Normalización de capa

La **normalización de capa** (*layer normalization*) estandariza las activaciones dentro de cada capa, evitando que los valores se dispersen y manteniendo el entrenamiento estable. Para un vector $\mathbf{z} \in \mathbb{R}^d$, se calcula:

$$\text{LayerNorm}(\mathbf{z}) = \gamma \odot \frac{\mathbf{z} - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \quad (3.16)$$

donde $\mu = \frac{1}{d} \sum_{i=1}^d z_i$ y $\sigma^2 = \frac{1}{d} \sum_{i=1}^d (z_i - \mu)^2$ son la media y varianza calculadas sobre las d dimensiones del vector, ϵ es una constante pequeña para evitar la división por cero, y $\gamma, \beta \in \mathbb{R}^d$ son parámetros aprendibles de escala y desplazamiento.

A diferencia de la *batch normalization* (que normaliza a lo largo de la dimensión del lote), la normalización de capa opera de forma independiente sobre cada ejemplo, lo que la hace más adecuada para secuencias de longitud variable y tamaños de lote pequeños, ambas situaciones habituales en NLP.

3.3.4. Post-normalización vs Pre-normalización

En el Transformer original, la normalización se aplica *después* de la conexión residual (**Post-LN**):

$$\mathbf{z} = \text{LayerNorm}(\mathbf{x} + \text{MultiHead}(\mathbf{x})) \quad (3.17)$$

$$\mathbf{y} = \text{LayerNorm}(\mathbf{z} + \text{FFN}(\mathbf{z})) \quad (3.18)$$

Sin embargo, se ha demostrado [41] que esta configuración puede generar inestabilidades durante el entrenamiento, especialmente en modelos muy profundos, puesto que los gradientes deben fluir a través de la normalización antes de alcanzar la conexión residual. La alternativa, adoptada por la gran mayoría de modelos modernos, es la **pre-normalización (Pre-LN)**:

$$\mathbf{z} = \mathbf{x} + \text{MultiHead}(\text{LayerNorm}(\mathbf{x})) \quad (3.19)$$

$$\mathbf{y} = \mathbf{z} + \text{FFN}(\text{LayerNorm}(\mathbf{z})) \quad (3.20)$$

En esta configuración, la conexión residual actúa directamente sobre las activaciones sin normalizar, creando un camino limpio para el gradiente. La normalización se aplica solo a la entrada de cada subcapa, controlando la escala de las activaciones que entran en la atención y la FFN. Esta disposición permite entrenar modelos más profundos de forma estable sin necesidad de esquemas de calentamiento (*warmup*) tan agresivos del ratio de aprendizaje.

3.4. Codificación posicional

Una de las ventajas del Transformer es su capacidad de paralelización al no ser un modelo secuencial. Sin embargo, esto conlleva la pérdida de la información posicional. Si introducimos una secuencia en el mecanismo de atención, obtenemos el mismo resultado que con esa secuencia permutada, ya que la operación de atención es **invariante a permutaciones**: la función softmax y la media ponderada no dependen del orden de las entradas. Para comprobarlo formalmente, si π es una permutación de $\{1, \dots, L\}$ y $\mathbf{X}_\pi$ es la matriz $\mathbf{X}$ con sus filas permutadas según π , entonces $\text{Attention}(\mathbf{X}_\pi) = \text{Attention}(\mathbf{X})_\pi$. Es decir, la atención permuta su salida de la misma manera que su entrada, pero no distingue entre diferentes ordenamientos.

Esto es claramente indeseable para el lenguaje, donde el orden es fundamental: *El gato persigue al ratón* y *El ratón persigue al gato* contienen las mismas palabras pero tienen significados opuestos. La solución es añadir **codificaciones posicionales** (*positional encodings*) a las representaciones de las palabras.

3.4.1. Codificaciones sinusoidales

En el artículo original, se proponen codificaciones basadas en funciones sinusoidales con distintas frecuencias:

$$PE_{(pos, 2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right) \quad (3.21)$$

$$PE_{(pos, 2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right) \quad (3.22)$$

siendo $pos \in \{0, \dots, L-1\}$ la posición del token en la secuencia e $i \in \{0, \dots, d/2-1\}$ el índice de la dimensión. Cada dimensión del vector posicional corresponde a una senoide con una frecuencia diferente: las primeras dimensiones oscilan rápidamente (frecuencias altas, capturando posiciones cercanas) y las últimas oscilan lentamente (frecuencias bajas, capturando la posición global).

Una propiedad fundamental de esta codificación es que la diferencia entre las codificaciones de dos posiciones cualesquiera es una transformación lineal de la distancia entre ellas. Formalmente, para cualquier desplazamiento fijo k , existe una matriz $\mathbf{T}_k$ tal que $PE_{pos+k} = \mathbf{T}_k \cdot PE_{pos}$. Esto permite al modelo aprender relaciones posicionales relativas a partir de operaciones lineales sobre las codificaciones absolutas.

La representación final de cada token se construye sumando su embedding semántico y su codificación posicional:

$$\mathbf{x}_{pos} = \text{Embedding}(\text{token}_{pos}) + PE_{pos} \quad (3.23)$$

3.4.2. Codificaciones aprendidas

Una alternativa utilizada en modelos como BERT y GPT consiste en reemplazar las funciones sinusoidales por una **matriz de embeddings posicionales** $\mathbf{P} \in \mathbb{R}^{L_{max} \times d}$ cuyos pesos se aprenden durante el entrenamiento. Cada fila $\mathbf{P}_{pos}$ es un vector aprendible que se suma al embedding del token de la misma forma que las codificaciones sinusoidales.

Las codificaciones aprendidas tienen la ventaja de poder adaptarse a los patrones posicionales específicos de los datos de entrenamiento. Su principal inconveniente es que no pueden generalizar a secuencias más largas que la longitud máxima L_{max} vista durante el entrenamiento, ya que no existen embeddings para posiciones superiores. Las codificaciones sinusoidales, al ser funciones continuas, pueden en principio extenderse a cualquier longitud, aunque en la práctica su rendimiento también degrada para posiciones no vistas durante el entrenamiento.

Como veremos en el capítulo de innovaciones, investigaciones posteriores han propuesto métodos más sofisticados de codificación posicional, como RoPE (*Rotary Position Embeddings*), que integran la información posicional directamente en el cálculo de la atención y permiten una mejor generalización a secuencias más largas.

3.5. Encoder y Decoder del Transformer

El Transformer original utiliza una estructura encoder-decoder, sustituyendo las capas recurrentes por bloques de auto-atención.

3.5.1. Encoder

El **encoder** consiste en una pila de N_{capas} bloques idénticos (en el artículo original, $N_{\text{capas}} = 6$). Cada bloque contiene una capa de atención multicabeza y una capa FFN, ambas con conexiones residuales y normalización. El encoder procesa toda la secuencia de entrada simultáneamente, produciendo una representación contextualizada $\mathbf{H}^{enc} \in \mathbb{R}^{L \times d}$ de cada token.

La entrada al encoder es la suma de los embeddings de los tokens y las codificaciones posicionales. Dado que el encoder no utiliza enmascaramiento causal, cada token puede atender a todos los demás tokens de la secuencia, tanto anteriores como posteriores. Esto permite crear representaciones bidireccionales, capturando el contexto completo de cada palabra.

3.5.2. Decoder

El **decoder** tiene una estructura similar al encoder, pero con dos diferencias importantes que reflejan su papel generativo.

En primer lugar, su capa de auto-atención utiliza **enmascaramiento causal** para garantizar que la generación sea autorregresiva: el token en la posición i solo puede atender a los tokens en las posiciones $\{1, \dots, i\}$ de la secuencia de salida.

En segundo lugar, incluye una capa adicional de **atención cruzada** (*cross-attention*) entre la auto-atención y la FFN. En esta capa, las queries provienen de la salida de la auto-atención del decoder, mientras que las keys y values provienen de la salida del encoder $\mathbf{H}^{enc}$:

$$\text{CrossAttention}(\mathbf{X}_{dec}) = \text{Attention}(\mathbf{X}_{dec}\mathbf{W}^Q, \mathbf{H}^{enc}\mathbf{W}^K, \mathbf{H}^{enc}\mathbf{W}^V) \quad (3.24)$$

Esto permite al decoder acceder a la información de toda la secuencia de entrada en cada paso de generación, de forma análoga a la atención de Bahdanau pero usando el mecanismo de producto escalar escalado. Nótese que la longitud de las queries (L_{dec}) puede diferir de la de las keys/values (L_{enc}), lo que hace que la matriz de atención cruzada sea de tamaño $L_{dec} \times L_{enc}$ en vez de cuadrada.

La salida del último bloque del decoder se pasa por una capa lineal $\mathbf{W}_{vocab} \in \mathbb{R}^{d \times |V|}$ (donde $|V|$ es el tamaño del vocabulario) seguida de un softmax para obtener las probabilidades sobre el vocabulario:

$$P(w_{t+1} | \mathbf{w}_{1:t}, \mathbf{x}) = \text{softmax}(\mathbf{h}_t^{dec} \mathbf{W}_{vocab}) \quad (3.25)$$

3.5.3. Entrenamiento con teacher forcing

Durante el entrenamiento, al igual que en el encoder-decoder recurrente, se utiliza *teacher forcing*: en vez de alimentar al decoder con sus propias predicciones (lo que podría acumular errores), se le proporciona la secuencia de salida correcta desplazada una posición hacia la derecha. La función de coste es la entropía cruzada entre las probabilidades predichas (ecuación 3.25) y los tokens reales en cada posición.

Una ventaja crucial del Transformer sobre las RNN en este aspecto es que, gracias al enmascaramiento causal, todas las posiciones del decoder se pueden entrenar simultáneamente en un solo paso hacia adelante. En una RNN, se necesitarían L pasos secuenciales. Esta paralelización del entrenamiento, combinada con la capacidad de capturar dependencias largas, es lo que ha permitido a los Transformers escalar a tamaños y conjuntos de datos sin precedentes.

3.6. Variante decoder-only

La arquitectura Transformer original fue diseñada con una estructura encoder-decoder para tareas de secuencia a secuencia, como la traducción automática. Sin embargo, para tareas de generación de texto, como el modelado de lenguaje, se ha popularizado una variante simplificada que utiliza únicamente el decoder, conocida como **Transformer decoder-only**.

En esta variante se elimina el encoder y la capa de atención cruzada (cross-attention), conservando únicamente la autoatención enmascarada y la capa de propagación directa. La clave es la **máscara causal**: una matriz triangular inferior que se aplica a las puntuaciones de atención antes del softmax, asignando $-\infty$ a todas las posiciones futuras. De esta forma, el token en la posición t solo puede atender a los tokens en posiciones $1, \dots, t$, garantizando que la predicción de cada token dependa exclusivamente del contexto anterior:

$$\text{CausalAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax} \left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}} + \mathbf{M} \right) \mathbf{V} \quad (3.26)$$

donde $\mathbf{M}$ es la máscara causal con $M_{ij} = 0$ si $i \geq j$ y $M_{ij} = -\infty$ si $i < j$.

Esta arquitectura es la base de los modelos generativos de lenguaje más utilizados en la actualidad, como la familia GPT [30] y LLaMA [38]. Su simplicidad y eficacia la convierten en el estándar para

el modelado de lenguaje autorregresivo, y es la variante que utilizaremos en la parte experimental de este trabajo.

3.7. Modelos preentrenados

Una de las consecuencias más importantes de la arquitectura Transformer es que su capacidad de paralelización permite entrenar modelos con cientos de millones o miles de millones de parámetros sobre grandes volúmenes de texto. Estos modelos, una vez entrenados en tareas generales de lenguaje, pueden reutilizarse como punto de partida para tareas específicas, proceso conocido como **aprendizaje de transferencia** (*transfer learning*).

Esta idea ha dado lugar a dos grandes familias de modelos preentrenados. Por un lado, los modelos de tipo encoder como BERT [11] y su variante optimizada RoBERTa [25], que se entrenan con objetivos bidireccionales (predicción de palabras enmascaradas) y destacan en tareas de comprensión como clasificación y extracción de información. Por otro, los modelos de tipo decoder-only como GPT [30], que se entrenan con el objetivo autorregresivo de predecir el siguiente token y destacan en tareas de generación. Modelos más recientes como LLaMA [38] combinan la arquitectura decoder-only con mejoras como las codificaciones posicionales rotatorias y normalizaciones más estables, permitiendo alcanzar rendimientos competitivos con modelos mucho mayores.

El éxito de estos modelos ha transformado el campo del NLP: en lugar de diseñar y entrenar arquitecturas específicas para cada tarea, el paradigma actual consiste en adaptar modelos preentrenados de propósito general. No obstante, el crecimiento en tamaño de estos modelos agrava el problema de la complejidad cuadrática de la atención, motivando la búsqueda de arquitecturas más eficientes.

En el siguiente capítulo evaluaremos el Transformer original —tal como se ha descrito en este capítulo— junto con la FFN y el LSTM, para cuantificar su rendimiento base antes de introducir las mejoras y alternativas arquitectónicas que se formalizan en los capítulos 5 y 6.

Capítulo 4

Evaluación inicial de los modelos base

El objetivo de este capítulo es comparar el rendimiento de los tres modelos base —FFN, LSTM y Transformer— en dos tareas de procesamiento del lenguaje natural: clasificación multiclase de textos y modelado de lenguaje. Los resultados revelarán que cada tarea favorece a una arquitectura diferente, motivando la pregunta central de este trabajo sobre el papel de la longitud de secuencia. Este capítulo cumple así una función de motivación empírica para el resto del trabajo: identifica la autoatención como el mecanismo pivote, muestra que no siempre es necesaria y expone el problema de longitud y optimización que las arquitecturas de los capítulos siguientes tratan de resolver.

4.1. Clasificación de textos

4.1.1. Diseño experimental

Para la tarea de clasificación evaluamos los tres modelos sobre dos conjuntos de datos: **20news-groups**, un corpus estándar en inglés con 18 846 artículos distribuidos en 20 categorías temáticas, y **progreso_tema**, un conjunto de datos propio con 4 339 noticias del diario El Progreso clasificadas en 15 temáticas, cuyos textos son considerablemente más largos (media de 450 palabras frente a las aproximadamente 200 de 20newsgroups). Ambos conjuntos se describen en detalle en el Apéndice A.

Los textos se tokenizan a nivel de palabra extrayendo secuencias alfanuméricas en minúsculas. A partir de los textos de entrenamiento se construye un vocabulario de 20 000 tokens, incluyendo tokens especiales de padding, clasificación (`<cls>`) y desconocido (`<unk>`). Las secuencias se truncan o rellenan a una longitud fija `max_len`, y se genera una máscara de atención binaria que indica qué posiciones corresponden a tokens reales.

Los tres modelos comparten la misma estructura base: una capa de embedding de dimensión 128, una capa específica de la arquitectura (media de embeddings en la FFN, LSTM bidireccional de una capa, o un bloque Transformer decoder-only con 4 cabezas de atención), y una cabeza de clasificación formada por dos capas lineales con ReLU y dropout de 0.2. El número de parámetros es comparable entre modelos, y todos se entrenan con el optimizador Adam durante un máximo de 15 épocas con parada temprana basada en la pérdida de validación.

Para evaluar la influencia de los hiperparámetros y garantizar la robustez de los resultados, cada modelo se ejecuta con dos ratios de aprendizaje (0.001 y 0.0002), tres longitudes máximas de secuencia (256, 512 y 1024 tokens) y tres semillas aleatorias (42, 1234, 13), resultando en 18 ejecuciones por modelo y conjunto de datos. Todos los entrenamientos se realizan en GPU V100 con precisión mixta. Se reportan dos métricas principales, promediadas sobre las tres semillas de la mejor configuración de cada modelo. La **precisión** (*accuracy*) es la fracción de textos clasificados correctamente. El **F1-score macro** es la media *no ponderada* del F1 de cada clase —donde el F1 de una clase es la media armónica de su precisión y su exhaustividad (*recall*)—, de modo que todas las categorías pesen por igual con independencia de su frecuencia. La diferencia entre ambas métricas es informativa: cuando

la *accuracy* es alta pero el F1 macro es bajo, el modelo acierta en las clases mayoritarias pero falla en las minoritarias, un sesgo que la *accuracy* por sí sola no revela.

4.1.2. Resultados globales

La Tabla 4.1 muestra los resultados de la mejor configuración de cada modelo, calculada como la media sobre las tres semillas del par (lr, max_len) que maximiza la precisión en test.

Tabla 4.1: Mejor configuración por modelo en clasificación (media $\pm$ desviación sobre 3 seeds)

Dataset	Modelo	lr	L	Accuracy	F1 macro
20news	FFN	0.001	1024	0,613 $\pm$ 0,006	0,604 $\pm$ 0,007
	LSTM	0.001	1024	0,537 $\pm$ 0,002	0,530 $\pm$ 0,003
	Transformer	0.001	512	0,582 $\pm$ 0,003	0,571 $\pm$ 0,005
Progreso	FFN	0.001	1024	0,721 $\pm$ 0,011	0,600 $\pm$ 0,009
	LSTM	0.001	1024	0,604 $\pm$ 0,019	0,472 $\pm$ 0,008
	Transformer	0.001	512	0,686 $\pm$ 0,002	0,517 $\pm$ 0,006

El resultado más notable es que la FFN obtiene la mejor precisión en ambos conjuntos de datos, superando al Transformer por más de 3 puntos en 20newsgroups y por más de 3.5 puntos en Progreso. El LSTM queda en última posición en ambos casos. Cabe destacar además que la mejor configuración del Transformer usa una longitud máxima de 512, no de 1024, lo que ya anticipa un problema con secuencias largas que analizamos a continuación.

Más allá de la precisión global, la brecha entre *accuracy* y F1 macro revela diferencias cualitativas importantes. En 20newsgroups, donde las 20 categorías están distribuidas de forma relativamente equilibrada, ambas métricas son cercanas para los tres modelos (1–2 puntos de diferencia). En Progreso, sin embargo, la distribución de las 15 temáticas es marcadamente desigual —categorías dominantes como *lugo* o *galicia* concentran un porcentaje mucho mayor de ejemplos—, y la brecha se amplía de forma notable: la FFN alcanza un 72.1 % de *accuracy* pero solo un 60.0 % de F1 macro (12 puntos de diferencia), mientras que el Transformer presenta la brecha más extrema (68.6 % vs. 51.7 %, casi 17 puntos). El LSTM muestra un patrón similar (60.4 % vs. 47.2 %). Esto indica que todos los modelos tienden a clasificar correctamente las clases mayoritarias pero fallan sistemáticamente en las minoritarias, un problema que la arquitectura por sí sola no resuelve y que subraya la importancia de evaluar con métricas que penalicen este sesgo.

4.1.3. Efecto de la longitud de secuencia

Para estudiar cómo afecta la longitud de la secuencia al rendimiento de cada modelo, comparamos las tres longitudes evaluadas para ambos ratios de aprendizaje. La Tabla 4.2 muestra los resultados completos.

Tabla 4.2: Precisión en test según longitud de secuencia y ratio de aprendizaje (media $\pm$ desviación sobre 3 semillas)

Dataset	Modelo	lr = 0.001			lr = 0.0002		
		$L=256$	$L=512$	$L=1024$	$L=256$	$L=512$	$L=1024$
20news	FFN	,574 $\pm$,006	,597 $\pm$,003	,613 $\pm$,006	,337 $\pm$,010	,453 $\pm$,005	,540 $\pm$,005
	LSTM	,444 $\pm$,003	,489 $\pm$,003	,537 $\pm$,002	,296 $\pm$,003	,351 $\pm$,010	,408 $\pm$,008
	Transformer	,525 $\pm$,005	,582 $\pm$,003	,471 $\pm$,087	,327 $\pm$,012	,460 $\pm$,001	,513 $\pm$,005
Progreso	FFN	,603 $\pm$,016	,704 $\pm$,003	,721 $\pm$,011	,317 $\pm$,006	,516 $\pm$,013	,610 $\pm$,006
	LSTM	,498 $\pm$,016	,532 $\pm$,009	,604 $\pm$,019	,341 $\pm$,014	,460 $\pm$,030	,500 $\pm$,006
	Transformer	,615 $\pm$,011	,686 $\pm$,002	,513 $\pm$,172	,448 $\pm$,010	,576 $\pm$,011	,629 $\pm$,010

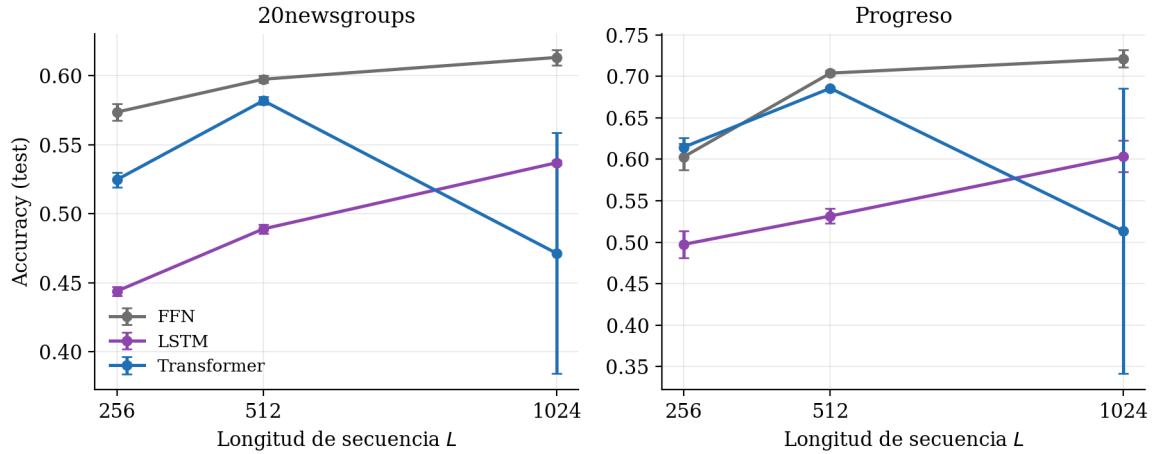


Figura 4.1: Precisión en test frente a la longitud de secuencia en clasificación (lr = 0,001, media $\pm$ desviación sobre 3 semillas). La FFN y el LSTM mejoran de forma monótona; el Transformer, en cambio, se degrada bruscamente a $L=1024$, con una varianza entre semillas muy elevada (barras de error).

La FFN y el LSTM muestran un comportamiento consistente con ambos ratios de aprendizaje: su rendimiento mejora de forma monótona al aumentar la longitud de secuencia, ya que disponen de más información para construir la representación del texto. La FFN con lr = 0,001 pasa de 0.574 a 0.613 en 20newsgroups y de 0.603 a 0.721 en Progreso; el LSTM sigue la misma tendencia, aunque partiendo de niveles inferiores.

El Transformer, sin embargo, presenta un patrón que depende críticamente del ratio de aprendizaje. Con lr = 0,001, mejora de $L=256$ a $L=512$ pero su rendimiento **cae drásticamente al pasar a $L=1024$** : en 20newsgroups, la precisión baja de 0.582 a 0.471, perdiendo más de 11 puntos; en Progreso, la caída es de 0.686 a 0.513, una pérdida de 17 puntos (Figura 4.1). Sin embargo, con lr = 0,0002 el

Transformer **mejora de forma monótona con la longitud**, pasando de 0.327 a 0.513 en 20newsgroups y de 0.448 a 0.629 en Progreso. Este contraste es revelador: la degradación a $L=1024$ no es una limitación arquitectónica intrínseca del Transformer, sino una **patología de optimización** que se manifiesta cuando la combinación de un ratio de aprendizaje alto y una superficie de pérdida más compleja (con la matriz de atención de 1024×1024 entradas por cabeza) impide la convergencia.

Dos observaciones refuerzan esta interpretación. En primer lugar, la varianza entre semillas del Transformer a $L=1024$ con $\text{lr} = 0,001$ es extraordinariamente alta (± 0.087 en 20newsgroups, ± 0.172 en Progreso), más de un orden de magnitud mayor que la de la FFN, lo que indica una fuerte sensibilidad a la inicialización propia de un proceso de optimización inestable. En segundo lugar, como veremos en la siguiente subsección, el Transformer con $\text{lr} = 0,001$ a $L=1024$ ni siquiera consigue minimizar la pérdida de entrenamiento, lo que descarta el sobreajuste como causa del problema.

En cuanto a la eficiencia computacional, el coste refleja las diferencias teóricas entre arquitecturas. La FFN completa el entrenamiento en menos de 2 minutos en todos los casos. El LSTM, al ser secuencial, escala linealmente con la longitud, pasando de 1.3 a 12 minutos en 20newsgroups al cuadruplicar la secuencia. El Transformer se sitúa entre ambos, alcanzando los 9 minutos con $L=1024$.

4.1.4. Discusión

Los resultados de clasificación revelan que la autoatención no proporciona una ventaja inherente en esta tarea, pero el análisis del grid completo de hiperparámetros permite ir más allá de esta conclusión superficial y diagnosticar los modos de fallo de cada arquitectura.

La Tabla 4.3 presenta la pérdida de entrenamiento y validación para la configuración con $\text{lr} = 0,001$ y $L=1024$, que es la que exhibe el contraste más marcado entre arquitecturas.

Tabla 4.3: Pérdida de entrenamiento y validación en clasificación ($\text{lr} = 0,001$, media sobre 3 semillas). Se incluye la pérdida de entrenamiento a $L=512$ y a $L=1024$: a diferencia de FFN y LSTM, la del Transformer *empeora* al aumentar la longitud.

Dataset	Modelo	$\mathcal{L}_{\text{train}}^{L=512}$	$\mathcal{L}_{\text{train}}^{L=1024}$	$\mathcal{L}_{\text{val}}^{L=1024}$	Diagnóstico
20news	FFN	0.136	0.089	1.931	Sobreajuste
	LSTM	0.114	0.111	4.012	Sobreajuste severo
	Transformer	0.584	1.295	2.044	Optimización inestable
Progreso	FFN	0.133	0.008	1.123	Sobreajuste
	LSTM	0.082	0.029	3.824	Sobreajuste severo
	Transformer	0.283	1.226	2.165	Optimización inestable

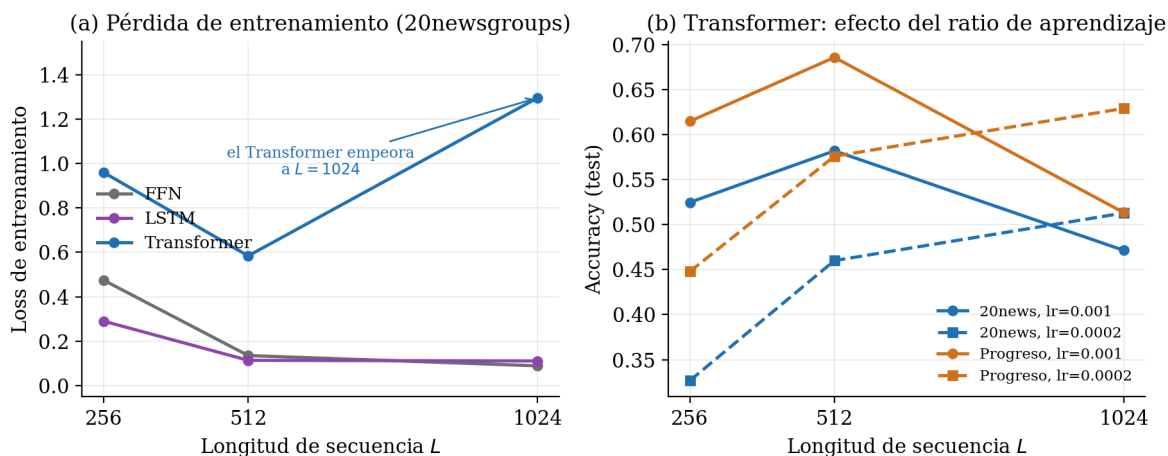


Figura 4.2: (a) Pérdida de entrenamiento frente a la longitud ($lr = 0,001$): la de FFN y LSTM disminuye, pero la del Transformer *empeora* a $L=1024$, lo que descarta tanto el sobreajuste como la falta de capacidad. (b) Precisión del Transformer con $lr = 0,001$ frente a $lr = 0,0002$: con el ratio menor desaparece el colapso a $L=1024$, lo que confirma un fallo de optimización y no una limitación arquitectónica.

Los tres modelos presentan patologías de entrenamiento distintas:

La **FFN** alcanza un loss de entrenamiento prácticamente nulo (0.089 en 20newsgroups, 0.008 en Progreso), lo que indica que memoriza los datos de entrenamiento con facilidad. Sin embargo, su loss de validación permanece elevado (1.93, 1.12), revelando un sobreajuste considerable. A pesar de ello, obtiene la mejor precisión en test. Esto se explica por la naturaleza de bolsa de palabras de su representación: al promediar los embeddings de los tokens, la FFN aprende una función robusta que mapea la distribución de vocabulario de cada texto a su categoría. Este resultado es coherente con el teorema de aproximación universal presentado en el capítulo 2: la red aproxima eficazmente la función objetivo, aunque con datos limitados tiende a sobreajustar los detalles del conjunto de entrenamiento.

El **LSTM** también reduce el loss de entrenamiento a valores cercanos a cero (0.111, 0.029), pero su loss de validación es el más alto de los tres modelos (4.01, 3.82), indicando un sobreajuste aún más severo. Esto sugiere que la información secuencial que captura el LSTM —el orden de las palabras— no solo no ayuda en clasificación, sino que contribuye a memorizar patrones secuenciales espurios del entrenamiento que no generalizan. Las limitaciones del procesamiento recurrente descritas en la sección 2.3 —en particular, la compresión forzosa de toda la secuencia en un vector de estado de dimensión fija— se manifiestan aquí como una propensión a memorizar en lugar de abstraer.

El **Transformer** exige una lectura más cuidadosa. A $L=512$ ajusta los datos de entrenamiento sin dificultad (pérdida de 0.584 en 20newsgroups y 0.283 en Progreso), lo que demuestra que dispone de **capacidad de sobra** para la tarea. Sin embargo, al pasar a $L=1024$ su pérdida de entrenamiento no disminuye, sino que **empeora** (1.295 y 1.226): el modelo deja de ajustar unos datos que sí ajustaba con secuencias más cortas. Esto descarta a la vez el sobreajuste —la brecha train-val es moderada, no como la de la FFN o el LSTM— y la falta de capacidad —ajusta bien a $L=512$ y, como veremos, domina el modelado de lenguaje—. El diagnóstico correcto es un **fallo de convergencia**: en el régimen $lr = 0,001$ con $L=1024$ la optimización se vuelve inestable y el modelo no alcanza un buen mínimo. Lo corrobora la enorme dispersión entre semillas en esa configuración (± 0.087 y ± 0.172 de *accuracy*, frente a ± 0.004 a $L=512$). La causa probable combina la complejidad cuadrática de la atención (una matriz de 1024×1024 puntuaciones por cabeza, formalizada en el capítulo 3) con la codificación posicional absoluta aprendida, cuyo espacio de representación se vuelve más difícil de optimizar a medida que crece L (Figura 4.2a).

Lo más revelador es que el problema **desaparece con un ratio de aprendizaje más bajo**: con

$\text{lr} = 0,0002$, el Transformer mejora de forma monótona con la longitud (Tabla 4.2), alcanzando 0.513 en 20newsgroups y 0.629 en Progreso a $L=1024$ (Figura 4.2b). Esto confirma que la degradación observada con $\text{lr} = 0,001$ es una patología de optimización —un mínimo local o una zona de inestabilidad numérica— y no una limitación inherente de la autoatención para secuencias largas. Esta distinción es relevante para el resto del trabajo: sugiere que innovaciones arquitectónicas que faciliten la optimización con secuencias largas (como las codificaciones posicionales relativas del capítulo 5) podrían mitigar este tipo de fallo.

En resumen, cada arquitectura falla de una manera distinta: la FFN y el LSTM sobreajustan —su pérdida de entrenamiento tiende a cero al crecer L — pero generalizan razonablemente gracias a la robustez de su representación; el Transformer, en cambio, no llega a converger en el régimen de secuencias largas y ratio de aprendizaje alto —no por falta de capacidad, sino por una optimización inestable que un ratio menor corrige—. Ningún modelo resuelve satisfactoriamente la tarea, lo que plantea la pregunta: ¿en qué tipo de tareas sí aporta una ventaja clara la autoatención?

4.2. Modelado de lenguaje

4.2.1. Diseño experimental

La segunda tarea es el modelado de lenguaje autorregresivo sobre WikiText-103 (descrito en el apéndice): dado un contexto de tokens previos, el modelo debe predecir el siguiente token. A diferencia de la clasificación, esta tarea exige capturar las dependencias secuenciales entre palabras para generar predicciones coherentes en cada posición.

Para que esta evaluación sea directamente comparable con el estudio principal, los tres modelos base se entrenan con **el mismo protocolo y las mismas implementaciones que se emplean en el capítulo 7**: tokenizador de GPT-2 (vocabulario de unas 50 000 subpalabras), $d_{\text{model}} = 256$, una FFN, un LSTM de 2 capas y un Transformer *decoder-only* de 4 bloques con 8 cabezas y codificación posicional absoluta aprendida; optimizador AdamW con ratio de aprendizaje 3×10^{-4} , 5 épocas y precisión completa, sobre el conjunto completo de WikiText-103. La métrica es la **perplejidad** (*perplexity*), definida como la exponencial de la entropía cruzada media por token (sección 2.1):

$$\text{PPL} = \exp(\mathcal{L}), \quad \mathcal{L} = -\frac{1}{L} \sum_{t=1}^L \log p_{\text{modelo}}(x_t \mid x_{<t}). \quad (4.1)$$

Intuitivamente, la perplejidad mide el grado de *incertidumbre* del modelo al predecir el siguiente token: un valor de k indica que, en promedio, el modelo está tan indeciso como si tuviera que elegir uniformemente entre k alternativas igualmente probables. Una perplejidad de 1 correspondería a un modelo perfecto que asigna probabilidad 1 al token correcto, mientras que un modelo que repartiera la probabilidad de forma uniforme sobre todo el vocabulario ($\sim 50\,000$ subpalabras) alcanzaría una perplejidad cercana a ese tamaño. Valores más bajos indican, por tanto, un mejor modelo del lenguaje.

4.2.2. Resultados

Tabla 4.4: Perplejidad en test en WikiText-103 según la longitud de contexto (mismo protocolo que el capítulo 7). En **negrita**, el mejor valor de cada modelo.

Modelo	$L=128$	$L=256$	$L=512$	$L=1024$	Parámetros
FFN	768,6	855,4	927,3	1046,9	~25.8M
LSTM	162,3	149,0	140,6	144,3	~26.8M
Transformer	66,6	66,8	71,0	79,4	~29.0M

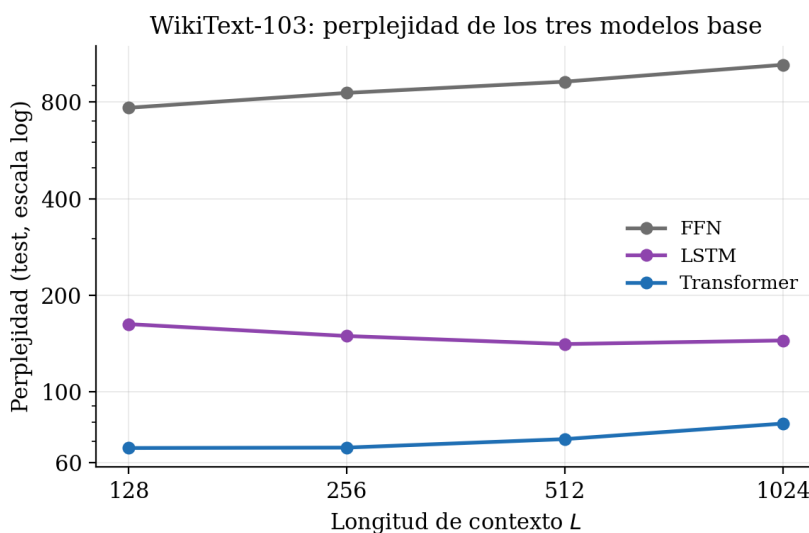


Figura 4.3: Perplejidad en test en WikiText-103 (escala logarítmica) frente a la longitud de contexto. El Transformer domina, pero se degrada al aumentar la longitud; la FFN es la peor y empeora de forma monótona; el LSTM satura en torno a 140–160.

La Tabla 4.4 y la Figura 4.3 muestran una jerarquía clara e *inversa* a la observada en clasificación, y revelan que cada arquitectura responde de forma distinta a la longitud del contexto.

El **Transformer** obtiene con diferencia la menor perplejidad (66,6 a $L=128$, frente a los 162,3 del LSTM y los 768,6 de la FFN): el acceso directo a cualquier posición del contexto mediante autoatención es decisivo en una tarea que depende del orden de las palabras. Ahora bien —y este es el matiz relevante para el resto del trabajo— su perplejidad **empeora al aumentar la longitud**, pasando de 66,6 a $L=128$ a 79,4 a $L=1024$ (+19%). El Transformer con codificación posicional absoluta aprendida no solo no aprovecha el contexto adicional, sino que se degrada con él: el mismo patrón que ya observamos en clasificación. A diferencia de aquel caso, aquí no realizamos un barrido del ratio de aprendizaje, por lo que no podemos aislar la causa con la misma rotundidad; pero el hecho de que la degradación aparezca en *ambas* tareas apunta de forma consistente a las limitaciones de la codificación posicional absoluta, que el capítulo 5 aborda mediante codificaciones relativas.

La **FFN** queda en último lugar, con una perplejidad altísima (768–1047) que además **crece monótonamente con la longitud**. Su representación de media acumulada de embeddings no puede capturar dependencias de orden y, cuanto más largo es el contexto, más se diluye en el promedio la información reciente. Es el peor de los tres modelos precisamente por carecer de cualquier mecanismo posicional, lo que valida en la práctica las limitaciones teóricas discutidas en el capítulo 2.

El **LSTM** se sitúa en una posición intermedia y exhibe un tercer patrón: su perplejidad apenas varía con la longitud (~ 140 – 162), con un mínimo en torno a $L=512$ (140,6). Su procesamiento recurrente le permite explotar el contexto mucho mejor que la FFN, pero el cuello de botella del estado de dimensión fija le impide aprovechar secuencias muy largas, de modo que satura en lugar de seguir mejorando.

En conjunto, los tres modelos base ya muestran dos de las formas cualitativas de respuesta a la longitud que reaparecerán en el estudio completo del capítulo 7 —degradación (FFN y Transformer con codificación absoluta) y saturación (LSTM)—, mientras que la tercera, la mejora monótona, solo aparecerá allí con las arquitecturas modernas.

4.3. Discusión: ¿cuándo importa la atención?

Los resultados de las secciones anteriores revelan un contraste llamativo. En clasificación de textos, la FFN supera al Transformer; en modelado de lenguaje, el Transformer es claramente superior. Este patrón no es casual y se explica por la naturaleza de cada tarea.

En clasificación, el objetivo es asignar una etiqueta a todo el documento. Para ello basta con capturar qué palabras aparecen y con qué frecuencia, información que la FFN obtiene eficazmente mediante el promedio de embeddings. Las relaciones posicionales entre tokens son secundarias: un artículo sobre deportes contiene palabras como *gol*, *equipo* o *partido* independientemente de su orden. En este contexto, la autoatención no solo no aporta ventaja, sino que añade complejidad innecesaria que dificulta el entrenamiento con datos limitados.

En modelado de lenguaje, por el contrario, el modelo debe predecir la siguiente palabra en cada posición. Esta tarea exige comprender las dependencias secuenciales: qué palabras tienden a seguir a otras, cómo la aparición de un sujeto condiciona el verbo posterior, o cómo un conector al inicio de una frase condiciona su estructura. El mecanismo de autoatención está diseñado precisamente para capturar estas relaciones entre posiciones, lo que explica que su perplejidad (~ 67) se sitúe muy por debajo de la del LSTM (~ 150) y de la de la FFN (~ 800), como recoge la Tabla 4.4.

4.3.1. Validación experimental de los fundamentos teóricos

Los resultados de este capítulo validan experimentalmente las propiedades teóricas descritas en los capítulos 2 y 3, cerrando el arco entre la teoría y la práctica.

La **FFN como aproximador universal sin noción de orden** (capítulo 2): en clasificación, la FFN demuestra que una red de propagación directa es capaz de aprender una representación eficaz cuando la distribución de palabras es suficiente para discriminar categorías. Sin embargo, su perplejidad, del orden de 800–1000 en modelado de lenguaje, confirma que la ausencia de cualquier mecanismo posicional la hace incapaz de capturar dependencias secuenciales. El promedio de embeddings destruye toda información de orden, exactamente como predice la teoría: la FFN es un poderoso aproximador de funciones, pero la función “siguiente token dado el contexto ordenado” no puede expresarse mediante una media aritmética de los embeddings de entrada.

El **LSTM y las limitaciones de la recurrencia** (capítulo 2): el LSTM mejora sustancialmente a la FFN en modelado de lenguaje (perplejidad en torno a 150 frente a los ~ 800 de la FFN), confirmando que la memoria secuencial y las puertas de olvido permiten capturar dependencias temporales. No obstante, su perplejidad duplica la del Transformer, ilustrando el cuello de botella recurrente: toda la información del contexto debe comprimirse en un vector de estado de dimensión fija, lo que produce una pérdida progresiva de información cuanto mayor es la distancia entre posiciones dependientes. En clasificación, esta misma propiedad opera en su contra: el LSTM no solo no aporta información

útil sobre el orden (irrelevante para la tarea) sino que memoriza patrones secuenciales espurios del entrenamiento que no generalizan, como evidencia su severo sobreajuste (Tabla 4.3).

El **Transformer y el poder de la autoatención** (capítulo 3): con acceso directo a cualquier posición del contexto en $\mathcal{O}(1)$ saltos, el Transformer obtiene la menor perplejidad (~ 67), confirmando que el acceso global es decisivo para el modelado de lenguaje. Sin embargo, su degradación con secuencias largas —visible tanto en clasificación como en modelado de lenguaje— anticipa un problema central formalizado en el capítulo 3: la complejidad cuadrática $\mathcal{O}(L^2)$ y la codificación posicional fija imponen un escenario de optimización difícil que, combinado con datos limitados, puede impedir la convergencia. El hallazgo de que esta degradación es sensible al ratio de aprendizaje refuerza la hipótesis de que es un problema de optimización, no de capacidad representacional.

4.3.2. Hipótesis para los capítulos posteriores

Las observaciones de este capítulo definen la agenda del resto del trabajo y se resumen en tres ideas. La autoatención resulta decisiva cuando importan las dependencias de orden (modelado de lenguaje) y prescindible cuando no (clasificación). En su forma original, además, es *frágil* con secuencias largas, y no por falta de capacidad: lo que falla es la optimización, lastrada por la codificación posicional absoluta y por el coste cuadrático de la atención. Y esa fragilidad resulta corregible, porque desaparece sin más que bajar el ratio de aprendizaje. Esto conduce a la pregunta central del trabajo: **¿qué ocurre cuando aumentamos sistemáticamente la longitud de la secuencia en una tarea donde el orden sí importa?** Los tres hechos anteriores motivan directamente las hipótesis que los capítulos 5 y 6 formalizan y que el capítulo 7 contrasta variando la longitud en igualdad de condiciones:

1. **Codificaciones posicionales relativas** (capítulo 5): si la degradación del Transformer con secuencias largas —en clasificación y en modelado de lenguaje— se debe en parte a las limitaciones de la codificación posicional aprendida —que asigna un vector fijo a cada posición sin codificar explícitamente distancias relativas—, las codificaciones rotacionales (RoPE) y los sesgos lineales de atención (ALiBi) deberían facilitar la generalización a secuencias más largas, tanto en clasificación como en modelado de lenguaje.
2. **Escalado de capacidad con Mixture of Experts** (capítulo 5): al margen del problema de optimización, cabe preguntarse si aumentar la capacidad del modelo —más parámetros sin multiplicar proporcionalmente el coste por token— mejora el modelado cuando el cuello de botella es *estadístico* (la riqueza de patrones del lenguaje) y no de optimización ni de mecanismo. MoE permite aislar este factor: si la ganancia aparece en modelado de lenguaje pero no en razonamiento estructural, confirmaría que añadir capacidad ayuda solo cuando el cuello de botella lo es de capacidad.
3. **Modelos de espacio de estados como alternativa lineal** (capítulo 6): si el LSTM está limitado por el cuello de botella del estado fijo y la recurrencia secuencial, un modelo de espacio de estados como Mamba —que combina complejidad lineal en la longitud con un mecanismo de selección dependiente de la entrada— debería superar al LSTM y potencialmente competir con el Transformer, especialmente en secuencias muy largas donde la complejidad cuadrática de la atención se convierte en un factor limitante.
4. **Arquitecturas híbridas** (capítulo 6): la combinación de capas de atención y capas de SSM en una arquitectura híbrida como Jamba debería capturar lo mejor de ambos mundos: el acceso global del Transformer y la eficiencia lineal de Mamba. Si esta hipótesis se confirma, indicaría que la atención y la recurrencia selectiva son mecanismos complementarios, no competidores.

La evaluación comparativa del capítulo 7 contrastará estas hipótesis mediante un estudio sistemático con ocho arquitecturas y cuatro longitudes de secuencia sobre WikiText-103, utilizando las condiciones homogéneas de entrenamiento que se describirán en dicho capítulo.

Capítulo 5

Mejoras en la arquitectura Transformer para secuencias largas

Como vimos en el capítulo 4, el Transformer original supera con claridad a la FFN y al LSTM en modelado de lenguaje, pero presenta limitaciones fundamentales —complejidad cuadrática en la longitud de secuencia y codificación posicional fija— que restringen su escalabilidad. Este capítulo aborda estas limitaciones en tres niveles. En primer lugar, se analizan las limitaciones del Transformer original y se revisan las mejoras que la comunidad ha adoptado progresivamente en sus componentes internos. A continuación, se formalizan tres innovaciones concretas que dan lugar a tres de los modelos experimentales de este trabajo: las codificaciones posicionales relativas RoPE y ALiBi, que mejoran la generalización a secuencias más largas que las vistas en entrenamiento, y la técnica de Mixture of Experts (MoE), que escala la capacidad del modelo sin escalar proporcionalmente su coste computacional.

5.1. Limitaciones del Transformer y motivación de nuevas arquitecturas

A pesar de su éxito, la arquitectura Transformer presenta limitaciones fundamentales que motivan el desarrollo de nuevas soluciones.

La limitación más evidente es la **complejidad cuadrática** $\mathcal{O}(L^2)$ del mecanismo de auto-atención. La materialización de la matriz de atención completa $\mathbf{QK}^T \in \mathbb{R}^{L \times L}$ requiere tanto $\mathcal{O}(L^2)$ operaciones como memoria. Para $L = 512$, esto supone 262,144 entradas por cabeza; para $L = 4096$, más de 16 millones; y para $L = 16384$, más de 268 millones. FlashAttention reduce la memoria adicional pero no elimina el coste cuadrático en cómputo.

La segunda limitación es el **coste de inferencia autorregresiva**. Incluso con KV-cache, el coste por token generado crece linealmente con la longitud del contexto, ya que cada nuevo query debe compararse con todas las keys anteriores. Esto contrasta con los modelos recurrentes, donde el coste por token es constante.

Estas limitaciones definen dos ejes de mejora. Por un lado, las secciones siguientes de este capítulo estudian mejoras que operan dentro del paradigma de la atención: codificaciones posicionales relativas (RoPE, ALiBi) que mejoran la generalización a secuencias largas, y Mixture of Experts que escala la capacidad sin escalar proporcionalmente el coste. Por otro, el capítulo 6 presenta los modelos de espacio de estados, que reemplazan completamente la atención por un mecanismo de complejidad lineal.

Tabla 5.1: Comparativa entre las arquitecturas estudiadas

	FFN	RNN/LSTM	Transformer
Procesamiento	Paralelo	Secuencial	Paralelo
Orden de secuencia	No	Implícito	Requiere PE
Depend. largas	No	Limitadas	Sí ($\mathcal{O}(1)$ camino)
Complejidad	$\mathcal{O}(1)$	$\mathcal{O}(L)$	$\mathcal{O}(L^2)$
Preentrenamiento	Inviabile	Inviabile	Altamente efectivo
Inferencia/token	—	$\mathcal{O}(1)$	$\mathcal{O}(L)$

5.2. Evolución de los componentes del Transformer

Desde su publicación en 2017, la estructura global del Transformer se ha mantenido sorprendentemente estable —pilas de bloques con atención y FFN—, pero los detalles de implementación han evolucionado de forma significativa. A continuación resumimos brevemente las mejoras más relevantes adoptadas en los modelos de estado del arte actuales. Aquellas innovaciones que dan lugar a modelos experimentales en este trabajo (RoPE, ALiBi y MoE) se formalizan en detalle en las secciones siguientes.

5.2.1. RMSNorm

La normalización de capa (ecuación 3.16) requiere calcular tanto la media como la varianza. **RMS-Norm** (*Root Mean Square Layer Normalization*) [43] simplifica este proceso eliminando el centrado por la media:

$$\text{RMSNorm}(\mathbf{z}) = \gamma \odot \frac{\mathbf{z}}{\text{RMS}(\mathbf{z})}, \quad \text{RMS}(\mathbf{z}) = \sqrt{\frac{1}{d} \sum_{i=1}^d z_i^2} \quad (5.1)$$

donde $\gamma \in \mathbb{R}^d$ es el único parámetro aprendible. La intuición es que el efecto más importante de la normalización es el **reescalado** de las activaciones, no el centrado. RMSNorm produce resultados comparables a LayerNorm con menor coste computacional al requerir una sola reducción (la suma de cuadrados) frente a las dos de LayerNorm. Es utilizada en LLaMA [38], Mamba [17] y la gran mayoría de modelos recientes.

5.2.2. SwiGLU

La capa FFN original (ecuación 3.13) utiliza una activación ReLU entre dos transformaciones lineales. Las **Gated Linear Units** (GLU) [10] mejoran este diseño introduciendo un mecanismo de puerta multiplicativo, donde una rama computa la activación y otra controla cuánta información fluye, de forma análoga a las puertas de la LSTM. **SwiGLU** [35] es la variante que se ha convertido en

estándar, reemplazando la sigmoide de la GLU por la función SiLU:

$$\text{SwiGLU}(\mathbf{z}) = (\mathbf{z}\mathbf{W}_1) \odot \text{SiLU}(\mathbf{z}\mathbf{W}_2) \quad (5.2)$$

Para compensar la proyección adicional, la dimensión intermedia se reduce típicamente de $4d$ a $\frac{8}{3}d$, manteniendo un número similar de parámetros. SwiGLU es utilizada en LLaMA, PaLM y en la capa de *gating* del bloque Mamba.

5.2.3. FlashAttention

FlashAttention [7] no modifica el mecanismo de atención matemáticamente, sino que optimiza su implementación a nivel de hardware. El cuello de botella de la atención estándar no es el cálculo aritmético, sino la transferencia de la matriz $L \times L$ entre la memoria principal de la GPU (HBM) y la memoria rápida (SRAM). FlashAttention resuelve esto dividiendo las matrices $\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$ en bloques que caben en SRAM y calculando la atención por bloques sin materializar nunca la matriz completa. El resultado es una reducción significativa del uso de memoria (de $\mathcal{O}(L^2)$ a $\mathcal{O}(L)$ en memoria adicional) y una aceleración práctica de 2-4× respecto a la implementación estándar. FlashAttention es compatible con todas las variantes de atención (causal, sliding window, etc.) y ha sido adoptada universalmente. Una versión posterior, FlashAttention-2 [8], optimiza adicionalmente el reparto del trabajo entre los *warps* de la GPU y reduce aún más el tiempo de cómputo.

5.2.4. Otras mejoras relevantes

La **KV-cache** almacena las matrices de keys y values de los tokens ya procesados durante la generación autorregresiva, evitando su recálculo y reduciendo el coste por token de $\mathcal{O}(L^2d)$ a $\mathcal{O}(Ld)$. Su principal inconveniente es el consumo de memoria, que crece linealmente con la longitud de la secuencia y el número de capas.

Para reducir el tamaño de la KV-cache, se han propuesto variantes de la atención multicabeza que comparten keys y values entre cabezas. **Multi-Query Attention** (MQA) [34] utiliza un único par $(\mathbf{K}, \mathbf{V})$ compartido por todas las cabezas, mientras que **Grouped-Query Attention** (GQA) [1] agrupa las cabezas en G grupos, cada uno con su propio par, ofreciendo un compromiso entre eficiencia y expresividad.

Estas innovaciones se combinan de forma prácticamente universal en los modelos modernos. La Tabla 5.2 resume su adopción en los modelos más representativos.

Tabla 5.2: Adopción de innovaciones en modelos representativos

Modelo	Pre/Post-LN	Norm.	FFN	Pos. Enc.	Atención	MoE
Transformer (2017)	Post-LN	LayerNorm	ReLU	Sinusoidal	MHA	No
BERT (2018)	Post-LN	LayerNorm	GELU	Aprendida	MHA	No
GPT-2 (2019)	Pre-LN	LayerNorm	GELU	Aprendida	MHA	No
LLaMA (2023)	Pre-LN	RMSNorm	SwiGLU	RoPE	MHA	No
LLaMA-2 (2023)	Pre-LN	RMSNorm	SwiGLU	RoPE	GQA	No
Mistral (2023)	Pre-LN	RMSNorm	SwiGLU	RoPE	GQA + SW	No
Mixtral (2024)	Pre-LN	RMSNorm	SwiGLU	RoPE	GQA + SW	Si

Se observa una clara convergencia: los modelos recientes utilizan de forma prácticamente universal pre-normalización con RMSNorm, activaciones SwiGLU, codificación posicional RoPE y alguna variante de atención con keys y values compartidas. Estas mejoras, combinadas con FlashAttention a nivel de implementación, constituyen la “receta” actual para construir modelos de lenguaje de estado del arte.

5.3. El problema de la codificación posicional

Antes de presentar las soluciones, conviene comprender por qué la codificación posicional del Transformer original es una limitación. Las codificaciones sinusoidales y aprendidas que vimos en el capítulo 3 son **absolutas**: asignan un vector fijo a cada posición $t \in \{0, 1, \dots, L_{max} - 1\}$ que se suma al embedding del token. Esto tiene dos consecuencias problemáticas.

En primer lugar, el modelo no puede procesar secuencias más largas que L_{max} , la longitud máxima vista en entrenamiento. Para las codificaciones aprendidas, las posiciones $t \geq L_{max}$ simplemente no tienen embedding definido. Para las sinusoidales, aunque las funciones sin y cos están definidas para todo t , el modelo nunca ha visto los patrones correspondientes durante el entrenamiento, por lo que no sabe cómo interpretar esas posiciones.

En segundo lugar, las posiciones absolutas no codifican de forma explícita la **distancia relativa** entre tokens. Para que el Transformer aprenda que la relación entre los tokens en las posiciones 5 y 8 es similar a la relación entre los 100 y 103 (ambas separadas por 3 posiciones), debe aprender esta invariancia de forma implícita, lo que requiere más datos y más capacidad.

Las dos innovaciones que presentamos a continuación abordan este problema desde perspectivas complementarias: RoPE codifica posiciones relativas en el espacio de los embeddings mediante rotaciones, mientras que ALiBi las codifica directamente en las puntuaciones de atención mediante un sesgo lineal.

5.4. Codificación posicional rotacional (RoPE)

5.4.1. Intuición y formulación

Las **Rotary Position Embeddings** (RoPE), propuestas por Su et al. [36], codifican la información posicional mediante rotaciones en el espacio de los embeddings. En lugar de sumar un vector de posición al embedding como en las codificaciones absolutas, RoPE aplica una rotación dependiente de la posición a los vectores de query y key *antes* de calcular el producto escalar de atención.

La intuición es geométrica: si pensamos en los vectores de query y key como puntos en un espacio multidimensional, rotar cada uno según su posición hace que el ángulo entre ellos (y por tanto su producto escalar) dependa únicamente de la diferencia entre posiciones, no de sus valores absolutos. Formalmente, buscamos una función $f(\mathbf{x}, m)$ que, aplicada al embedding $\mathbf{x}$ en la posición m , satisfaga:

$$\langle f(\mathbf{q}, m), f(\mathbf{k}, n) \rangle = g(\mathbf{q}, \mathbf{k}, m - n) \quad (5.3)$$

es decir, que el producto escalar entre el query en la posición m y el key en la posición n dependa únicamente de los vectores originales y de la diferencia $m - n$. La solución que propone RoPE es una rotación: $f(\mathbf{x}, m) = \mathbf{R}_m \mathbf{x}$, donde $\mathbf{R}_m$ es una matriz de rotación que depende de la posición.

5.4.2. Formulación por pares de dimensiones

Puesto que no existe una noción única de rotación en espacios de dimensión arbitraria, RoPE descompone el vector en pares de dimensiones consecutivas $(2i, 2i + 1)$ y aplica una rotación 2D independiente en cada par, cada una a una frecuencia diferente. Para un vector de dimensión d , la

rotación en la posición m para el par i -ésimo se define como:

$$\begin{pmatrix} x_{2i} \\ x_{2i+1} \end{pmatrix} \mapsto \begin{pmatrix} \cos(m\theta_i) & -\sin(m\theta_i) \\ \sin(m\theta_i) & \cos(m\theta_i) \end{pmatrix} \begin{pmatrix} x_{2i} \\ x_{2i+1} \end{pmatrix} \quad (5.4)$$

donde $\theta_i = 10000^{-2i/d}$ es la frecuencia base para el par i -ésimo, análoga a las frecuencias de las codificaciones sinusoidales del Transformer original que vimos en la sección 3.4.1. Las dimensiones bajas (i pequeño) rotan a frecuencias altas, capturando relaciones posicionales de corto alcance, mientras que las dimensiones altas (i grande) rotan a frecuencias bajas, capturando relaciones de largo alcance. Esta estructura multifrecuencia es la que permite a RoPE codificar simultáneamente información posicional a diferentes escalas.

La matriz de rotación completa para un vector de dimensión d en la posición m es por tanto una matriz bloque-diagonal, donde cada bloque 2×2 corresponde a un par de dimensiones:

$$\mathbf{R}_m = \begin{pmatrix} \mathbf{R}_m^{(0)} & & \\ & \ddots & \\ & & \mathbf{R}_m^{(d/2-1)} \end{pmatrix}, \quad \mathbf{R}_m^{(i)} = \begin{pmatrix} \cos(m\theta_i) & -\sin(m\theta_i) \\ \sin(m\theta_i) & \cos(m\theta_i) \end{pmatrix} \quad (5.5)$$

Para ver cómo actúa esta estructura en la práctica, consideremos un ejemplo concreto.

Ejemplo 5.1 (Rotación RoPE con $d = 4$). Consideremos un vector de dimensión $d = 4$, que se descompone en $d/2 = 2$ pares. Las frecuencias base son:

$$\theta_0 = 10000^{-0/4} = 1 \quad (\text{alta frecuencia}), \quad \theta_1 = 10000^{-2/4} = \frac{1}{100} \quad (\text{baja frecuencia})$$

Para un token en la posición $m = 3$, los ángulos de rotación de cada par son $3\theta_0 = 3 \text{ rad} \approx 172$ y $3\theta_1 = 0,03 \text{ rad} \approx 1,7$. La diferencia es dramática: el primer par rota casi media vuelta, mientras que el segundo apenas se mueve. La matriz de rotación completa es:

$$\mathbf{R}_3 = \begin{pmatrix} \cos(3) & -\sin(3) & 0 & 0 \\ \sin(3) & \cos(3) & 0 & 0 \\ 0 & 0 & \cos(0,03) & -\sin(0,03) \\ 0 & 0 & \sin(0,03) & \cos(0,03) \end{pmatrix} \approx \begin{pmatrix} -0,990 & -0,141 & 0 & 0 \\ 0,141 & -0,990 & 0 & 0 \\ 0 & 0 & 1,000 & -0,030 \\ 0 & 0 & 0,030 & 1,000 \end{pmatrix}$$

Al aplicarla al vector $\mathbf{q} = (1, 0, 1, 0)^T$:

$$\mathbf{R}_3 \mathbf{q} = \begin{pmatrix} -0,990 \\ 0,141 \\ 1,000 \\ 0,030 \end{pmatrix}$$

El primer par $(1, 0) \mapsto (-0,990, 0,141)$ ha sido prácticamente invertido, mientras que el segundo par $(1, 0) \mapsto (1,000, 0,030)$ apenas se ha modificado. Esto ilustra cómo las dimensiones bajas codifican variaciones posicionales rápidas (distinguen tokens cercanos) y las altas codifican variaciones lentas (son sensibles solo a diferencias de posición grandes).

Podemos verificar numéricamente la propiedad de posición relativa. Tomando $\mathbf{k} = (0,5, 0,8, 0,3, 0,6)^T$ con posiciones $m = 3$ y $n = 5$:

$$\langle \mathbf{R}_3 \mathbf{q}, \mathbf{R}_5 \mathbf{k} \rangle = -0,6476 = \langle \mathbf{q}, \mathbf{R}_2 \mathbf{k} \rangle$$

confirmando que la puntuación de atención depende únicamente de la distancia relativa $n - m = 2$.

La Figura 5.1 ilustra geométricamente el efecto descrito en el ejemplo anterior, mostrando cómo la rotación RoPE actúa de forma radicalmente distinta en cada par de dimensiones.

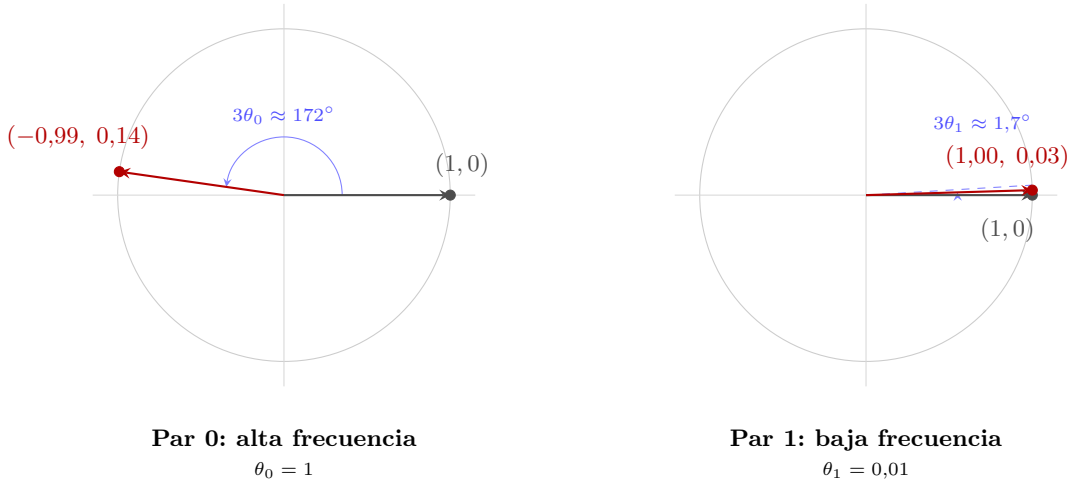


Figura 5.1: Efecto de la rotación RoPE sobre el vector $(1, 0)$ en la posición $m = 3$ para cada par de dimensiones del Ejemplo 5.1. El par de alta frecuencia (izquierda) rota casi media vuelta, codificando variaciones posicionales rápidas. El par de baja frecuencia (derecha) apenas se mueve, siendo sensible solo a diferencias de posición grandes.

5.4.3. Demostración de la propiedad de posición relativa

Como acabamos de verificar numéricamente en el ejemplo anterior, la propiedad fundamental de RoPE es que la puntuación de atención entre dos posiciones depende solo de la distancia relativa. Esto se demuestra de forma directa utilizando que las matrices de rotación 2D son ortogonales, es decir que satisfacen $\mathbf{R}_m^{(i)T} = \mathbf{R}_{-m}^{(i)}$, y la propiedad de composición $\mathbf{R}_a^{(i)} \mathbf{R}_b^{(i)} = \mathbf{R}_{a+b}^{(i)}$:

$$\langle \mathbf{R}_m \mathbf{q}, \mathbf{R}_n \mathbf{k} \rangle = (\mathbf{R}_m \mathbf{q})^T (\mathbf{R}_n \mathbf{k}) = \mathbf{q}^T \mathbf{R}_m^T \mathbf{R}_n \mathbf{k} = \mathbf{q}^T \mathbf{R}_{n-m} \mathbf{k} = \langle \mathbf{q}, \mathbf{R}_{n-m} \mathbf{k} \rangle \quad (5.6)$$

donde en el tercer paso usamos que $\mathbf{R}_m^T = \mathbf{R}_{-m}$ y la propiedad de composición $\mathbf{R}_{-m} \mathbf{R}_n = \mathbf{R}_{n-m}$. Por lo tanto, la puntuación de atención entre los tokens en las posiciones m y n depende únicamente de $n - m$, que es precisamente la propiedad que buscábamos al inicio de la sección.

5.4.4. Conexión con los números complejos e implementación eficiente

Existe una interpretación elegante de RoPE utilizando números complejos que, además de proporcionar una visión algebraica más limpia, es la base de su implementación eficiente. Cada par de

dimensiones (x_{2i}, x_{2i+1}) se puede representar como un número complejo $z_i = x_{2i} + i x_{2i+1}$. La rotación de la ecuación (5.4) es entonces simplemente la multiplicación por un fasor $e^{im\theta_i}$:

$$z_i \mapsto z_i \cdot e^{im\theta_i} = (x_{2i} + i x_{2i+1})(\cos(m\theta_i) + i \sin(m\theta_i)) \quad (5.7)$$

y el producto escalar en cada par de dimensiones es la parte real del producto conjugado: $\text{Re}[(\mathbf{R}_m z_i^q)^* (\mathbf{R}_n z_i^k)] = \text{Re}[z_i^{q*} \cdot z_i^k \cdot e^{i(n-m)\theta_i}]$, que efectivamente solo depende de $n - m$.

Esta formulación no es solo una curiosidad algebraica; es lo que hace que RoPE sea eficiente en la práctica. En lugar de materializar y multiplicar la matriz de rotación $\mathbf{R}_m \in \mathbb{R}^{d \times d}$ (una operación $\mathcal{O}(d^2)$ por posición), la implementación representa cada par (q_{2i}, q_{2i+1}) como un número complejo y lo multiplica por el fasor $e^{im\theta_i}$, que se precomputa una sola vez para todas las posiciones y frecuencias. En la práctica, esto se reduce a una multiplicación elemento a elemento entre el tensor de queries o keys (reinterpretado como tensor de complejos) y un tensor de fasores precomputado:

$$\mathbf{q_rotated} = \mathbf{q_complex} \odot \mathbf{freqs_cis}$$

donde $\mathbf{freqs_cis}[m, i] = e^{im\theta_i}$ y $\odot$ denota el producto de Hadamard en el campo complejo. El coste resultante es $\mathcal{O}(d)$ por posición, idéntico al de las codificaciones posicionales absolutas. Además, dado que RoPE solo modifica queries y keys (no los valores), es compatible con FlashAttention [7] sin ninguna modificación: basta aplicar la rotación antes de invocar el kernel de atención eficiente.

5.4.5. Propiedades de decaimiento y extrapolación

Una propiedad importante de RoPE es el **decaimiento con la distancia**. Se puede demostrar [36] que el producto escalar $\langle \mathbf{R}_m \mathbf{q}, \mathbf{R}_n \mathbf{k} \rangle$ tiende a decrecer a medida que $|m - n|$ aumenta, debido a las oscilaciones de las funciones trigonométricas a diferentes frecuencias. Este decaimiento implícito actúa como un sesgo inductivo que favorece la atención a tokens cercanos, similar al efecto de ALiBi (que veremos en la siguiente sección) pero emergiendo de forma natural de la geometría rotacional en lugar de ser impuesto explícitamente.

Sin embargo, RoPE no ofrece garantías teóricas de extrapolación perfecta a secuencias mucho más largas que las de entrenamiento. En la práctica, el rendimiento degrada gradualmente cuando la longitud de inferencia supera en más de 2-4× la longitud de entrenamiento. El origen de esta degradación se entiende bien si volvemos a la estructura multifrecuencia que vimos en la subsección anterior: las componentes de baja frecuencia (índices i altos) apenas completan una fracción de un ciclo durante el entrenamiento, y el modelo aprende a interpretar solo ese rango limitado de ángulos. Al extrapolar a posiciones mucho mayores, estas componentes alcanzan ángulos que nunca aparecieron durante el entrenamiento, produciendo representaciones fuera de distribución.

Para abordar este problema se han propuesto varias técnicas de escalado de frecuencias. La más directa es la **interpolación posicional** (PI) [5]: en lugar de usar la posición m directamente, se escala por un factor L/L' donde L es la longitud de entrenamiento y $L' > L$ la longitud deseada, de modo que las posiciones $0, \dots, L' - 1$ se comprimen al rango $[0, L)$:

$$\theta_i^{\text{PI}} = \theta_i, \quad m \mapsto m \cdot \frac{L}{L'} \quad (5.8)$$

Esto garantiza que todos los ángulos caigan en rangos vistos durante el entrenamiento, pero al comprimir las posiciones se pierde resolución posicional: tokens cercanos reciben rotaciones más similares, dificultando su distinción.

El **NTK-aware scaling** [28] propone una solución más sofisticada que evita este compromiso: en lugar de escalar las posiciones, se reescalan las frecuencias base θ_i de forma diferenciada según su índice. La idea clave es que las componentes de alta frecuencia (índices bajos) ya cubren múltiples ciclos completos durante el entrenamiento y no necesitan ajuste, mientras que las de baja frecuencia (índices altos) son precisamente las que sufren la extrapolación. El escalado modifica la base de la siguiente forma:

$$\theta_i^{\text{NTK}} = (\alpha \cdot 10000)^{-2i/d} \quad (5.9)$$

donde $\alpha > 1$ es un factor que depende del ratio de extensión L'/L . Al aumentar la base de 10000 a $\alpha \cdot 10000$, todas las frecuencias se reducen, pero el efecto es proporcionalmente mayor en las componentes de baja frecuencia (que ya tenían θ_i pequeño), mientras que las de alta frecuencia apenas se ven afectadas. Esto preserva la resolución posicional local mientras extiende el rango efectivo de las componentes lentas.

Esta técnica y sus variantes (como *YaRN* [28], que combina el escalado NTK con interpolación parcial y un factor de atenuación por frecuencia) son las que han permitido extender la ventana de contexto de modelos como LLaMA 2 de 4096 a 32768 tokens, y de Code Llama hasta 100000 tokens, sin necesidad de reentrenar desde cero. La conexión con el tema central de este trabajo es directa: el comportamiento de RoPE frente a la longitud de secuencia determina en gran medida la capacidad del Transformer para procesar documentos largos, que es precisamente lo que evaluamos experimentalmente en el capítulo 7.

5.4.6. Adopción en modelos modernos

RoPE ha sido adoptada como codificación posicional estándar en la mayoría de los grandes modelos de lenguaje recientes, incluyendo LLaMA [38], Mistral, Qwen y PaLM. Frente a las codificaciones posicionales absolutas, su principal ventaja es la codificación de distancias relativas sin parámetros adicionales aprendibles. Frente a ALiBi, que veremos a continuación, su ventaja es que la información posicional se integra directamente en el espacio de representación de queries y keys, permitiendo que la posición interactúe con el contenido semántico de los tokens. Esta diferencia tiene implicaciones tanto teóricas como prácticas que analizaremos tras presentar ALiBi.

5.5. ALiBi: Attention with Linear Biases

5.5.1. Formulación

ALiBi (*Attention with Linear Biases*), propuesto por Press et al. [29], adopta un enfoque radicalmente distinto: en lugar de codificar posiciones mediante rotaciones o vectores sumados al embedding, ALiBi **no utiliza ninguna codificación posicional en los embeddings**. Toda la información posicional se inyecta directamente en las puntuaciones de atención mediante un sesgo lineal que penaliza las distancias:

$$\text{ALiBi}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}} + s \cdot \mathbf{D}\right) \mathbf{V} \quad (5.10)$$

donde $\mathbf{D}$ es una matriz de distancias con $D_{ij} = -|i - j|$ (que en el caso causal se simplifica a $D_{ij} = j - i$ para $j \leq i$, donde todos los valores son no positivos) y $s > 0$ es una pendiente (*slope*) fija que varía por cabeza de atención.

La intuición es directa: para cada par de posiciones (i, j) , se suma a la puntuación de atención un término negativo proporcional a la distancia entre ellas. Cuanto más lejos esté un token, más se penaliza su puntuación de atención, haciendo menos probable que el modelo le preste atención. La magnitud de esta penalización está controlada por la pendiente s .

5.5.2. Asignación de pendientes

Las pendientes se asignan como una serie geométrica fija (no aprendida): para un modelo con h cabezas, las pendientes son:

$$s_j = 2^{-8j/h}, \quad j = 1, 2, \dots, h \quad (5.11)$$

Por ejemplo, para un modelo con $h = 8$ cabezas, las pendientes son $\{2^{-1}, 2^{-2}, 2^{-3}, \dots, 2^{-8}\} = \{0,5, 0,25, 0,125, \dots, 0,0039\}$. La cabeza con $s = 0,5$ penaliza fuertemente las posiciones lejanas, concentrándose en el contexto inmediato. La cabeza con $s = 0,0039$ apenas penaliza, pudiendo “mirar”

miles de tokens hacia atrás. Esta diversidad permite al modelo capturar simultáneamente dependencias a diferentes escalas sin parámetros adicionales.

Nótese que cuando h no es una potencia de 2, se utiliza una interpolación entre las series para h potencias de 2 inmediatamente inferior y superior. Esta asignación fija se eligió empíricamente tras comparar múltiples esquemas, y su eficacia sugiere que una distribución geométrica de “alcances” de atención es un sesgo inductivo apropiado para el procesamiento del lenguaje.

La Figura 5.2 visualiza el efecto de esta asignación sobre la matriz de atención. Cada submatriz muestra el sesgo $s \cdot D_{ij}$ para una pendiente diferente, haciendo evidente cómo las cabezas con pendientes altas se concentran en el contexto inmediato mientras que las de pendientes bajas mantienen una visión prácticamente uniforme de toda la secuencia.

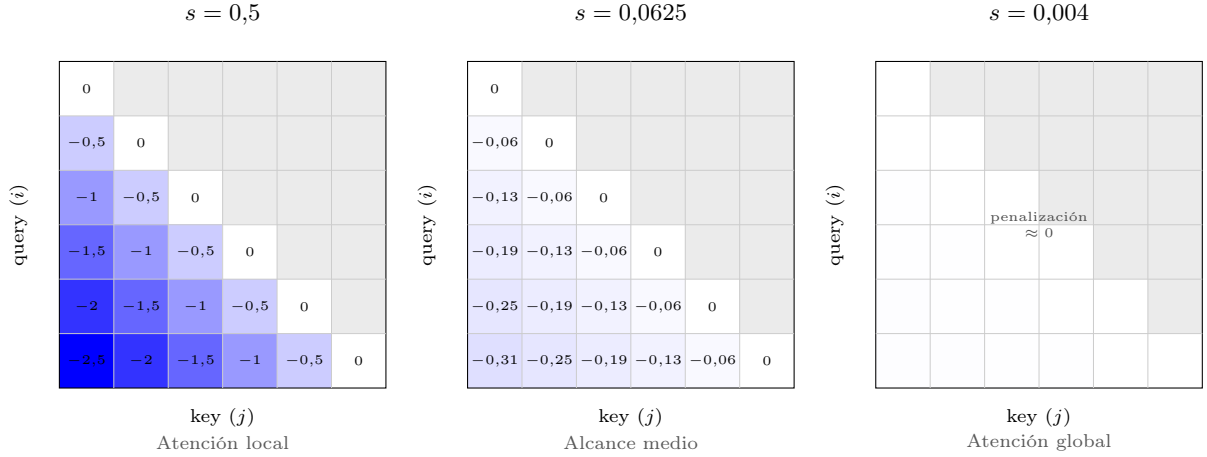


Figura 5.2: Sesgo ALiBi $s \cdot D_{ij}$ para tres cabezas de atención con pendientes decrecientes en una secuencia de 6 tokens. La escala de color es compartida entre las tres matrices para reflejar la magnitud absoluta de la penalización. Las celdas en gris claro corresponden a posiciones futuras (enmascaradas en el caso causal). La cabeza con $s = 0,5$ penaliza fuertemente las posiciones lejanas, concentrándose en el contexto inmediato, mientras que la cabeza con $s = 0,004$ apenas penaliza, pudiendo atender a todo el contexto disponible.

5.5.3. Propiedades de extrapolación

La propiedad más destacada de ALiBi es su capacidad de **extrapolación a secuencias largas**. Press et al. [29] demuestran que un modelo entrenado con ALiBi en secuencias de longitud L mantiene su rendimiento (o lo mejora) cuando se evalúa en secuencias de longitud $2L$, $4L$ o incluso $8L$, un comportamiento que no se observa con codificaciones sinusoidales ni aprendidas.

La razón intuitiva es que ALiBi no aprende representaciones posicionales absolutas: la única señal posicional es el sesgo lineal, y este está definido para cualquier distancia. Un token que aparece en la posición 1000 durante la inferencia simplemente recibe un sesgo proporcional a su distancia, exactamente como lo haría un token en la posición 100 durante el entrenamiento (pero con un valor absoluto 10 veces mayor). El softmax se encarga de reescalar las puntuaciones, por lo que el modelo no necesita haber “visto” esas distancias absolutas para manejarlas correctamente.

5.5.4. Comparación entre RoPE y ALiBi

Ambas estrategias logran que la atención dependa de la distancia relativa entre tokens, pero difieren en dónde y cómo se inyecta la información posicional. RoPE opera *en el espacio de los embeddings*, modificando queries y keys mediante rotaciones antes del cálculo de atención. ALiBi opera *en el*

espacio de las puntuaciones, sumando un sesgo después de calcular $\mathbf{QK}^T$. Esto tiene implicaciones prácticas: RoPE permite que la codificación posicional interactúe con el contenido semántico de los tokens (la rotación modifica la representación), mientras que ALiBi mantiene una separación estricta entre contenido y posición (los embeddings no contienen información posicional alguna).

En cuanto a la extrapolación, ALiBi ofrece garantías más robustas gracias a la simplicidad de su mecanismo. RoPE requiere técnicas adicionales de escalado para extrapolar. Sin embargo, RoPE ha demostrado un rendimiento ligeramente superior en tareas que requieren razonamiento posicional fino, posiblemente porque la integración de posición en el espacio de representación permite interacciones más ricas. En nuestros experimentos evaluaremos empíricamente cuál de estos mecanismos produce mejores resultados en nuestras tareas y cómo se comportan al variar la longitud de la secuencia.

5.6. Mixture of Experts (MoE)

Las innovaciones anteriores mejoran la codificación posicional del Transformer, pero no abordan un problema diferente: cómo **escalar la capacidad del modelo** (su número de parámetros y, por tanto, el “conocimiento” que puede almacenar) sin escalar proporcionalmente el coste computacional. La técnica de **Mixture of Experts** (MoE) [33, 12] propone una solución elegante a este problema, motivada por una observación simple: en un Transformer denso, todos los parámetros se activan para cada token, pero la mayoría de tokens probablemente solo necesitan una fracción de la capacidad total del modelo.

5.6.1. Principio de funcionamiento

La idea central es reemplazar la capa FFN de cada bloque Transformer por un conjunto de E redes FFN independientes (los “expertos”), junto con una red de enrutamiento (*router*) que, para cada token, selecciona qué expertos deben procesarlo. Formalmente:

$$\text{MoE}(\mathbf{z}) = \sum_{i=1}^E g_i(\mathbf{z}) \cdot \text{FFN}_i(\mathbf{z}) \quad (5.12)$$

donde $g_i(\mathbf{z})$ son los pesos de enrutamiento. En la práctica, para cada token solo se activan los k expertos con mayores pesos (típicamente $k = 1$ o $k = 2$), mientras que el resto tienen peso cero:

$$g_i(\mathbf{z}) = \begin{cases} \text{softmax}(\mathbf{z}\mathbf{W}_{\text{router}})_i & \text{si } i \in \text{TopK}(\mathbf{z}\mathbf{W}_{\text{router}}) \\ 0 & \text{en caso contrario} \end{cases} \quad (5.13)$$

donde $\mathbf{W}_{\text{router}} \in \mathbb{R}^{d \times E}$ es la matriz de pesos del router y TopK selecciona los k expertos con mayor puntuación. Es importante notar que el router es una simple capa lineal seguida de softmax, por lo que su coste computacional es negligible comparado con el de los expertos. La Figura 5.3 ilustra este mecanismo para un ejemplo con $E = 4$ expertos y $k = 2$.

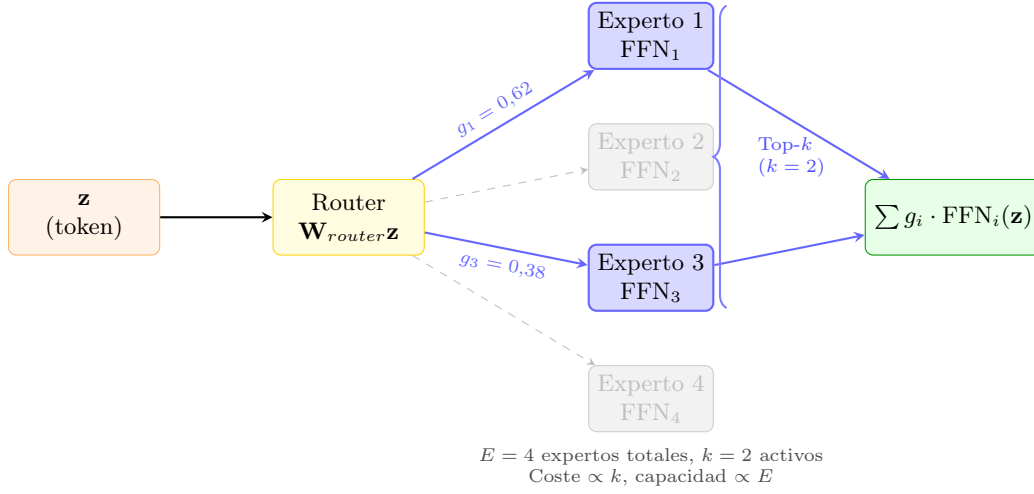


Figura 5.3: Diagrama del mecanismo Mixture of Experts (MoE). El router asigna pesos g_i a cada experto basándose en la representación del token $\mathbf{z}$, y solo los k expertos con mayor peso (en azul) procesan el token. Los expertos inactivos (en gris) no consumen computación. La salida es la combinación ponderada de los expertos activos.

La variante más extrema es el **Switch Transformer** [12], que utiliza $k = 1$ (un solo experto por token). Esto simplifica el enrutamiento y maximiza la eficiencia, ya que cada token solo necesita pasar por una FFN. Fedus et al. demuestran que esta simplificación no degrada el rendimiento y de hecho puede mejorarlo al reducir la complejidad de la señal de enrutamiento.

5.6.2. Capacidad vs. coste

La consecuencia más importante de MoE es que el **número total de parámetros** crece con E (hay E copias de la FFN), pero el **coste computacional por token** crece solo con k (solo se activan k expertos). Concretamente, si la FFN original tiene $2 \cdot d \cdot d_{ff}$ parámetros, una capa MoE con E expertos tiene $E \cdot 2 \cdot d \cdot d_{ff} + d \cdot E$ parámetros (los expertos más el router), pero el coste por token es solo $k \cdot 2 \cdot d \cdot d_{ff} + d \cdot E \approx k \cdot 2 \cdot d \cdot d_{ff}$ (el router es negligible).

Por ejemplo, Mixtral-8x7B [22] tiene $E = 8$ expertos con $k = 2$, alcanzando 47B de parámetros totales pero con un coste computacional por token equivalente a un modelo denso de ~ 13 B. Esto permite al modelo almacenar mucho más “conocimiento” (especializado en diferentes tipos de tokens o contextos) sin un coste proporcional.

5.6.3. Balanceo de carga

Un desafío fundamental de los modelos MoE es el **balanceo de carga** entre expertos. Sin restricciones, el router tiende a enviar la mayoría de tokens a unos pocos expertos, un fenómeno conocido como *winner-take-all* o colapso del router. Esto ocurre porque los expertos que reciben más tokens al inicio del entrenamiento mejoran más rápido, lo que refuerza la preferencia del router por ellos, creando un ciclo de retroalimentación positiva. Los expertos infrautilizados no reciben suficientes gradientes para mejorar y quedan efectivamente “muertos”.

Para evitar este colapso, se añade una **pérdida auxiliar de balanceo** que penaliza distribuciones desiguales de tokens entre expertos:

$$\mathcal{L}_{balance} = \alpha \cdot E \cdot \sum_{i=1}^E f_i \cdot p_i \quad (5.14)$$

donde f_i es la fracción de tokens asignados al experto i en el lote actual, p_i es la probabilidad media asignada al experto i por el router (antes de TopK), y α es un hiperparámetro que controla la intensidad de la regularización (típicamente $\alpha = 0,01$). Esta pérdida se minimiza cuando $f_i = p_i = 1/E$ para todo i , es decir, cuando todos los expertos reciben la misma proporción de tokens. El producto $f_i \cdot p_i$ (en lugar de, por ejemplo, la varianza de f_i) hace que la pérdida sea diferenciable respecto a los pesos del router y pueda optimizarse conjuntamente con la pérdida principal.

Otra técnica complementaria es la de **factor de capacidad**: se limita el número máximo de tokens que cada experto puede procesar a $C = c \cdot T/E$, donde $c \geq 1$ es el factor de capacidad (típicamente $c = 1,25$) y T es el número de tokens en el lote. Los tokens que sobrepasan la capacidad de su experto asignado se procesan con un camino residual directo (sin pasar por ningún experto), lo que penaliza implícitamente al router por asignar distribuciones desiguales.

5.6.4. Relevancia para este trabajo

En nuestros experimentos, incorporamos una variante MoE sobre el Transformer con embeddings posicionales aprendidos. Los expertos comparten la misma arquitectura FFN pero con pesos independientes, y el router utiliza selección top- k . Esto nos permite evaluar si la mayor capacidad del modelo (más parámetros totales, pero coste similar por token) se traduce en mejor rendimiento en modelado de lenguaje y razonamiento estructural. Al compararlo con los Transformers densos (vanilla, RoPE, ALiBi) podremos aislar el efecto de la capacidad del modelo frente al efecto de la codificación posicional.

5.7. Resumen

Las tres innovaciones presentadas en este capítulo abordan limitaciones diferentes del Transformer original. RoPE y ALiBi mejoran la codificación posicional para permitir una mejor generalización a secuencias largas: RoPE lo hace mediante rotaciones en el espacio de los embeddings (integrando la posición con el contenido), ALiBi mediante un sesgo lineal en las puntuaciones de atención (separando posición y contenido). MoE, por su parte, escala la capacidad del modelo sin escalar proporcionalmente su coste, permitiendo almacenar más “conocimiento” activando solo una fracción de los parámetros para cada token. Conviene matizar un punto que se retomará en el capítulo experimental: aunque la motivación teórica de RoPE y ALiBi incluye la *extrapolación* a secuencias más largas que las vistas en entrenamiento, los experimentos de este trabajo entrenan y evalúan cada modelo a la *misma* longitud. Lo que medimos, por tanto, no es la extrapolación posicional propiamente dicha, sino el efecto de la longitud de entrenamiento sobre la calidad y la optimización; la evaluación *train-short/test-long* se plantea como línea de trabajo futuro.

Capítulo 6

Modelos de espacio de estados (SSM)

Las arquitecturas presentadas hasta ahora —desde la FFN hasta las variantes mejoradas del Transformer— operan dentro de dos paradigmas: el procesamiento independiente por posición (FFN) o el mecanismo de atención (Transformer y variantes). En este capítulo presentamos una tercera familia de modelos que propone un mecanismo fundamentalmente distinto para procesar secuencias: los **modelos de espacio de estados** (SSM, *State Space Models*). Inspirados en la teoría de sistemas dinámicos lineales, los SSM ofrecen complejidad lineal o casi-lineal en la longitud de la secuencia, manteniendo la capacidad de capturar dependencias a larga distancia. Culminaremos con la arquitectura Mamba, que introduce un mecanismo de selección dependiente de la entrada que supera las limitaciones de los SSM clásicos.

6.1. Motivación: los tres modos de procesamiento de secuencias

A lo largo de este trabajo hemos visto tres maneras fundamentales de procesar una secuencia de tokens. La primera es la **recurrencia** (RNN, LSTM): procesar la secuencia token a token, manteniendo un estado oculto que se actualiza en cada paso. Esto ofrece complejidad lineal y coste constante por token en inferencia, pero impide la paralelización del entrenamiento. La segunda es la **atención** (Transformer): calcular interacciones directas entre todos los pares de tokens. Esto permite paralelización completa y captura de dependencias a cualquier distancia, pero con coste cuadrático. La tercera, que introducimos ahora, es la **convolución**: aplicar un filtro sobre la secuencia completa, permitiendo paralelización con coste lineal o casi-lineal.

Los modelos de espacio de estados son notables porque unifican estos tres modos. Admiten una formulación recurrente (eficiente en inferencia), una formulación convolucional (eficiente en entrenamiento) y, como veremos, una conexión teórica con la atención. Esta versatilidad es lo que les confiere su potencial como alternativa y complemento al Transformer.

Los SSM se inspiran además en la observación de que muchos sistemas físicos pueden describirse mediante ecuaciones diferenciales lineales con un estado oculto continuo. Un termostato, un circuito eléctrico o la dinámica de un péndulo se modelan naturalmente como sistemas que reciben una señal de entrada, mantienen un estado interno y producen una salida. Trasladado al procesamiento de secuencias, la señal de entrada es la secuencia de tokens, el estado oculto codifica el contexto acumulado y la salida es la representación procesada.

6.2. Formulación continua

Un modelo de espacio de estados continuo se define mediante el siguiente sistema de ecuaciones diferenciales ordinarias (EDO) lineales:

$$\mathbf{h}'(t) = \mathbf{A}\mathbf{h}(t) + \mathbf{B}u(t) \quad (6.1)$$

$$y(t) = \mathbf{C}\mathbf{h}(t) + Du(t) \quad (6.2)$$

donde $u(t) \in \mathbb{R}$ es la señal de entrada escalar en el instante t , $\mathbf{h}(t) \in \mathbb{R}^N$ es el vector de estado oculto de dimensión N , $y(t) \in \mathbb{R}$ es la salida, y las matrices $\mathbf{A} \in \mathbb{R}^{N \times N}$, $\mathbf{B} \in \mathbb{R}^{N \times 1}$, $\mathbf{C} \in \mathbb{R}^{1 \times N}$ y $D \in \mathbb{R}$ son los parámetros del sistema (no confundir N , la dimensión del estado, con la longitud de la secuencia L).

6.2.1. Papel de cada componente

Cada matriz del sistema cumple una función específica. La matriz $\mathbf{A}$ gobierna la **dinámica interna** del estado: determina cómo decae, se preserva o se transforma la información almacenada en $\mathbf{h}$ a lo largo del tiempo. Sus autovalores determinan el comportamiento cualitativo: autovalores con parte real negativa producen decaimiento (el estado “olvida”), autovalores puramente imaginarios producen oscilaciones (el estado “recuerda” de forma periódica), y autovalores con parte real positiva producen crecimiento inestable (no deseable en la práctica).

La matriz $\mathbf{B}$ controla **cómo la entrada influye en el estado**: qué componentes del estado se actualizan cuando llega una nueva entrada y con qué magnitud. La matriz $\mathbf{C}$ realiza la **lectura**: extrae la salida del estado, combinando sus componentes. El escalar D es un término de **paso directo** (*feedthrough*) que conecta la entrada directamente con la salida sin pasar por el estado; en la práctica suele omitirse o fijarse a cero, ya que las conexiones residuales del modelo cumplen esta función.

6.2.2. Solución analítica

La EDO lineal (6.1) tiene solución analítica cerrada. Dada una condición inicial $\mathbf{h}(0) = \mathbf{h}_0$:

$$\mathbf{h}(t) = e^{t\mathbf{A}}\mathbf{h}_0 + \int_0^t e^{(t-\tau)\mathbf{A}}\mathbf{B}u(\tau) d\tau \quad (6.3)$$

donde $e^{t\mathbf{A}} = \sum_{k=0}^{\infty} \frac{(t\mathbf{A})^k}{k!}$ es la **exponencial matricial**. El primer término describe la evolución del estado inicial bajo la dinámica de $\mathbf{A}$, y el segundo es una integral de convolución que acumula la influencia de toda la entrada pasada, ponderada por la respuesta al impulso del sistema $e^{t\mathbf{A}}\mathbf{B}$.

Esta solución analítica es clave para la discretización, ya que nos permite derivar fórmulas exactas (no aproximadas) para el sistema discreto.

6.2.3. Conexión con otros modelos

Esta formulación es general y engloba numerosos sistemas conocidos en estadística y aprendizaje automático. Las **RNN** pueden verse como un caso particular donde las matrices dependen de la entrada (no lineales) y la discretización es implícita. Los **Modelos Ocultos de Markov** (HMM) son sistemas de espacio de estados con variables discretas y transiciones estocásticas, mientras que los SSM que usamos aquí operan con estados continuos y transiciones deterministas. El **filtro de Kalman** constituye el análogo gaussiano continuo del HMM: un SSM lineal con ruido gaussiano donde el estado se estima de forma óptima.

La diferencia fundamental con estos modelos clásicos es que en los SSM para aprendizaje profundo, las matrices ($\mathbf{A}, \mathbf{B}, \mathbf{C}$) se **aprenden** durante el entrenamiento, típicamente parametrizadas de forma que garanticen estabilidad (autovalores de $\mathbf{A}$ con parte real negativa) y eficiencia computacional.

6.3. Discretización

Para aplicar el modelo continuo a secuencias discretas (como tokens de texto), necesitamos discretizarlo. Dado un paso temporal $\Delta > 0$, se transforma el sistema continuo $(\mathbf{A}, \mathbf{B}, \mathbf{C})$ en un sistema discreto $(\bar{\mathbf{A}}, \bar{\mathbf{B}}, \mathbf{C})$.

6.3.1. Método de Euler

La aproximación más simple es el **método de Euler** (hacia adelante), que reemplaza la derivada por un cociente de diferencias:

$$\frac{\mathbf{h}_{k+1} - \mathbf{h}_k}{\Delta} \approx \mathbf{A}\mathbf{h}_k + \mathbf{B}u_k \quad \Rightarrow \quad \mathbf{h}_{k+1} = (\mathbf{I} + \Delta\mathbf{A})\mathbf{h}_k + \Delta\mathbf{B}u_k \quad (6.4)$$

dando $\bar{\mathbf{A}}_{Euler} = \mathbf{I} + \Delta\mathbf{A}$ y $\bar{\mathbf{B}}_{Euler} = \Delta\mathbf{B}$. Este método es simple pero puede ser numéricamente inestable para Δ grande.

6.3.2. Retención de orden cero (ZOH)

El método preferido en la práctica es la **retención de orden cero** (*Zero-Order Hold*, ZOH), que asume que la entrada se mantiene constante en cada intervalo $[k\Delta, (k+1)\Delta)$ y utiliza la solución exacta (6.3):

$$\bar{\mathbf{A}} = \exp(\Delta\mathbf{A}) \quad (6.5)$$

$$\bar{\mathbf{B}} = (\Delta\mathbf{A})^{-1}(\exp(\Delta\mathbf{A}) - \mathbf{I}) \cdot \Delta\mathbf{B} \quad (6.6)$$

Este método es exacto bajo la asunción de entrada constante entre pasos, y produce un sistema discreto estable siempre que el sistema continuo lo sea (los autovalores de $\bar{\mathbf{A}} = \exp(\Delta\mathbf{A})$ tienen módulo < 1 si y solo si los de $\mathbf{A}$ tienen parte real < 0).

6.3.3. Transformada bilineal

Una tercera opción es la **transformada bilineal** (o método de Tustin), que aproxima $e^{\Delta\mathbf{A}}$ mediante un cociente de Padé de orden (1, 1):

$$\bar{\mathbf{A}}_{bilinear} = \left(\mathbf{I} - \frac{\Delta}{2}\mathbf{A}\right)^{-1} \left(\mathbf{I} + \frac{\Delta}{2}\mathbf{A}\right), \quad \bar{\mathbf{B}}_{bilinear} = \left(\mathbf{I} - \frac{\Delta}{2}\mathbf{A}\right)^{-1} \Delta\mathbf{B} \quad (6.7)$$

Este método preserva la estabilidad del sistema continuo (es incondicionalmente estable) y fue el utilizado en los primeros modelos SSM [16]. Sin embargo, en Mamba se prefiere la ZOH por su interpretación más directa y porque el paso Δ dependiente de la entrada tiene un significado más claro bajo ZOH.

6.3.4. La recurrencia discreta y el papel de Δ

Independientemente del método de discretización, el sistema discreto opera como una recurrencia:

$$\mathbf{h}_k = \bar{\mathbf{A}}\mathbf{h}_{k-1} + \bar{\mathbf{B}}u_k \quad (6.8)$$

$$y_k = \mathbf{C}\mathbf{h}_k \quad (6.9)$$

El parámetro Δ controla la **resolución temporal** del modelo, y su efecto sobre $\bar{\mathbf{A}}$ es crucial para entender los SSM. Considerando una entrada diagonal $A_{ii} = -\lambda_i$ con $\lambda_i > 0$, la discretización ZOH da $\bar{A}_{ii} = e^{-\Delta\lambda_i}$. Para Δ pequeño, $\bar{A}_{ii} \approx 1 - \Delta\lambda_i \approx 1$: el estado decae lentamente, **preservando información a largo plazo**. Para Δ grande, $\bar{A}_{ii} \approx 0$: el estado se renueva rápidamente, **priorizando la entrada reciente**. Esta interpretación es análoga a la puerta de olvido de la LSTM, con la diferencia clave de que en los SSM clásicos Δ es un parámetro global fijo, mientras que en Mamba será dependiente de la entrada.

6.4. Formulación convolucional

La formulación recurrente (6.8)-(6.9) procesa la secuencia de forma secuencial. Sin embargo, existe una propiedad fundamental de los sistemas lineales e invariantes en el tiempo (LTI) que permite una alternativa paralela: cuando las matrices $(\bar{\mathbf{A}}, \bar{\mathbf{B}}, \mathbf{C})$ **no dependen de la entrada**, la salida es una convolución de la entrada con un kernel fijo.

6.4.1. Derivación

Expandiendo la recurrencia (6.8) paso a paso (asumiendo $\mathbf{h}_{-1} = \mathbf{0}$):

$$\mathbf{h}_0 = \bar{\mathbf{B}} u_0 \quad (6.10)$$

$$\mathbf{h}_1 = \bar{\mathbf{A}}\bar{\mathbf{B}} u_0 + \bar{\mathbf{B}} u_1 \quad (6.11)$$

$$\mathbf{h}_2 = \bar{\mathbf{A}}^2\bar{\mathbf{B}} u_0 + \bar{\mathbf{A}}\bar{\mathbf{B}} u_1 + \bar{\mathbf{B}} u_2 \quad (6.12)$$

$$\vdots$$

$$\mathbf{h}_k = \sum_{j=0}^k \bar{\mathbf{A}}^j \bar{\mathbf{B}} u_{k-j} \quad (6.13)$$

Aplicando $\mathbf{C}$ para obtener la salida:

$$y_k = \mathbf{C}\mathbf{h}_k = \sum_{j=0}^k \mathbf{C}\bar{\mathbf{A}}^j \bar{\mathbf{B}} u_{k-j} \quad (6.14)$$

Esto tiene exactamente la forma de una convolución discreta causal. Definiendo el **kernel de convolución** $\bar{\mathbf{K}} \in \mathbb{R}^L$ como:

$$\bar{\mathbf{K}} = (\mathbf{C}\bar{\mathbf{B}}, \mathbf{C}\bar{\mathbf{A}}\bar{\mathbf{B}}, \dots, \mathbf{C}\bar{\mathbf{A}}^{L-1}\bar{\mathbf{B}}) \quad (6.15)$$

la salida completa se puede calcular como $\mathbf{y} = \bar{\mathbf{K}} * \mathbf{u}$, donde $*$ denota la convolución y $\mathbf{u} = (u_0, \dots, u_{L-1})$ es la secuencia de entrada.

6.4.2. Cálculo eficiente mediante FFT

Usando el **teorema de convolución**, la convolución en el dominio temporal equivale a una multiplicación punto a punto en el dominio frecuencial:

$$\mathbf{y} = \bar{\mathbf{K}} * \mathbf{u} = \text{IFFT}(\text{FFT}(\bar{\mathbf{K}}) \odot \text{FFT}(\mathbf{u})) \quad (6.16)$$

donde FFT e IFFT son la transformada rápida de Fourier y su inversa. Dado que la FFT tiene complejidad $\mathcal{O}(L \log L)$, toda la operación para procesar la secuencia completa se realiza en $\mathcal{O}(L \log L)$, frente al $\mathcal{O}(L^2)$ de la atención y el $\mathcal{O}(L)$ secuencial de la recurrencia (nótese que estas complejidades se refieren al procesamiento de toda la secuencia de longitud L , no al coste por token). La diferencia con la recurrencia es que la FFT es completamente **paralelizable**: no hay dependencia secuencial entre pasos.

6.4.3. La dualidad recurrencia-convolución

Esta **dualidad** es una propiedad clave de los SSM y no tiene análogo directo en las RNN ni en los Transformers. Durante el **entrenamiento**, donde se procesan secuencias completas, se usa la formulación convolucional (eficiente y paralelizable, $\mathcal{O}(L \log L)$). Durante la **inferencia autorregresiva**, donde se genera un token a la vez, se usa la formulación recurrente (coste $\mathcal{O}(1)$ por token, ya que basta actualizar $\mathbf{h}_k$ con la ecuación (6.8)).

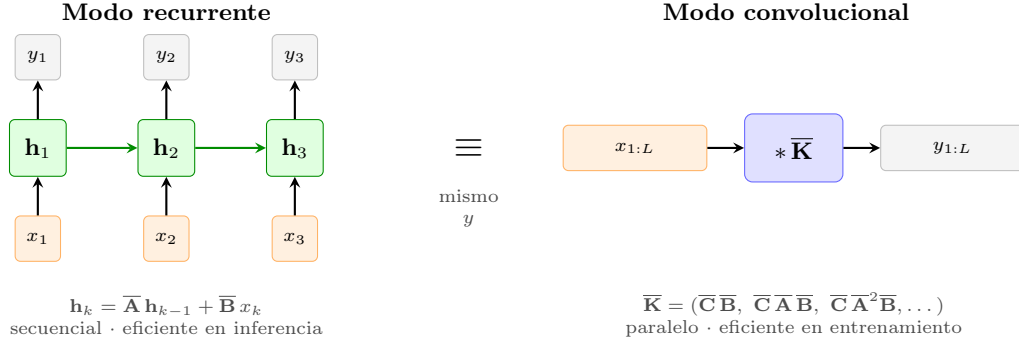


Figura 6.1: Las dos formas de computar un mismo SSM. El modo recurrente actualiza el estado paso a paso y es eficiente en inferencia (coste constante por token); el modo convolucional aplica un único kernel global $\bar{K}$ sobre toda la secuencia y es paralelizable en entrenamiento. Ambos producen idéntica salida, lo que permite usar cada modo en la fase que más conviene.

Esta dualidad contrasta con el Transformer, que tiene el mismo coste cuadrático en entrenamiento e inferencia (parcialmente mitigado por la KV-cache), y con las RNN, que tienen coste lineal en ambos modos pero no se benefician de la paralelización del entrenamiento. Los SSM obtienen lo mejor de ambos mundos: paralelización en entrenamiento y coste constante por token en inferencia.

La condición necesaria para esta dualidad es que el sistema sea **lineal e invariante en el tiempo** (LTI): las matrices $(\bar{A}, \bar{B}, \bar{C})$ deben ser las mismas para todos los tokens. Como veremos, esta condición será la que Mamba rompa deliberadamente.

6.5. S4: Structured State Spaces for Sequence Modeling

6.5.1. El problema de la inicialización de A

Un desafío fundamental de los SSM es la inicialización de la matriz A . Una inicialización aleatoria produce modelos incapaces de capturar dependencias a larga distancia, ya que las potencias $\bar{A}^k$ decaen rápidamente con k (los autovalores de $\bar{A}$ tienen módulo estrictamente menor que 1, y elevarlos a potencias grandes los lleva rápidamente a cero). Esto significa que la contribución de la entrada u_0 a la salida y_k (dada por $\bar{C}\bar{A}^k\bar{B}$) se desvanece exponencialmente, un problema análogo al desvanecimiento del gradiente en las RNN.

6.5.2. HiPPO: aproximación polinomial del historial

El modelo **S4** (*Structured State Spaces for Sequence Modeling*) [16] resuelve este problema utilizando el framework **HiPPO** (*High-order Polynomial Projection Operator*) [15], que inicializa A de forma que el estado oculto mantenga una aproximación comprimida de todo el historial de la entrada.

La idea de HiPPO es formalmente elegante: en cada instante t , el estado $h(t) \in \mathbb{R}^N$ almacena los N primeros coeficientes de la proyección de la función de entrada $u(\tau)$ (para $\tau \leq t$) sobre una base de polinomios ortogonales. Es decir, si $\{p_n\}_{n=0}^{N-1}$ son polinomios ortogonales respecto a una medida μ_t sobre $[0, t]$, entonces:

$$h_n(t) \approx \int_0^t p_n(\tau) u(\tau) d\mu_t(\tau), \quad n = 0, \dots, N-1 \quad (6.17)$$

El estado almacena una representación comprimida pero rica del historial completo: los coeficientes de bajo orden capturan tendencias globales y los de alto orden capturan variaciones de alta frecuencia. La clave está en que existe una EDO $h'(t) = Ah(t) + Bu(t)$ que mantiene esta aproximación actualizada en tiempo real conforme llega nueva entrada.

La variante **HiPPO-LegS** (*Scaled Legendre*) utiliza los polinomios de Legendre con una medida uniforme sobre una ventana deslizante $[0, t]$ que se reescala continuamente. La matriz $\mathbf{A}$ resultante tiene la estructura:

$$A_{nk} = \begin{cases} -(2n+1)^{1/2}(2k+1)^{1/2} & \text{si } n > k \\ -(n+1) & \text{si } n = k \\ 0 & \text{si } n < k \end{cases} \quad (6.18)$$

Esta es una matriz triangular inferior con una estructura muy específica que garantiza dos propiedades: que los autovalores tengan parte real negativa (estabilidad) y que el sistema preserve información del historial completo de forma óptima en el sentido de mínimos cuadrados. La estructura triangular inferior asegura además que el coeficiente n -ésimo solo dependa de los coeficientes de orden inferior, creando una jerarquía natural de resoluciones.

6.5.3. Parametrización DPLR y cálculo eficiente

Un problema práctico con la matriz HiPPO es que calcular el kernel $\overline{\mathbf{K}}$ (ecuación 6.15) requiere computar las potencias $\overline{\mathbf{A}}^k$, lo cual es costoso para matrices densas de tamaño $N \times N$. S4 resuelve esto demostrando que la matriz HiPPO puede descomponerse como la suma de una matriz **diagonal** y una de **bajo rango** (**DPLR**: *Diagonal Plus Low-Rank*):

$$\mathbf{A} = \mathbf{\Lambda} - \mathbf{p}\mathbf{q}^T \quad (6.19)$$

donde $\mathbf{\Lambda}$ es diagonal y $\mathbf{p}, \mathbf{q} \in \mathbb{R}^N$. Esta estructura permite calcular el kernel $\overline{\mathbf{K}}$ en el dominio frecuencial utilizando el **kernel de Cauchy**:

$$\hat{K}(\omega) = \mathbf{C}(e^{i\omega}\mathbf{I} - \overline{\mathbf{A}})^{-1}\overline{\mathbf{B}} \quad (6.20)$$

que, gracias a la estructura DPLR y al lema de inversión de matrices (Woodbury), puede evaluarse en $\mathcal{O}(N)$ para cada frecuencia ω , dando un coste total de $\mathcal{O}(NL)$ para calcular el kernel en todas las frecuencias.

6.5.4. Resultados y limitaciones de S4

S4 logró resultados sobresalientes en el benchmark *Long Range Arena* (LRA), una suite de tareas diseñadas específicamente para evaluar la capacidad de los modelos para capturar dependencias en secuencias de hasta 16.000 tokens. En particular, superó a los Transformers eficientes (como Performer y Linear Transformer) en la mayoría de las tareas, demostrando que los SSM son una alternativa viable.

Sin embargo, S4 tiene una limitación fundamental: las matrices $(\mathbf{A}, \mathbf{B}, \mathbf{C})$ son **independientes de la entrada**. Esto significa que el modelo aplica la misma transformación a todos los tokens, sin capacidad de “atender” selectivamente a partes relevantes de la secuencia. Esta invariancia es precisamente lo que permite la formulación convolucional, pero limita la expresividad del modelo en tareas que requieren procesamiento sensible al contexto.

6.6. Mamba: SSM selectivos

6.6.1. La limitación de la invariancia a la entrada

En los SSM lineales como S4, las matrices $(\mathbf{A}, \mathbf{B}, \mathbf{C}, \Delta)$ son parámetros fijos del modelo: no dependen del token que se está procesando. Esto implica que la transformación aplicada a la secuencia es la misma independientemente de su contenido, lo que constituye una limitación grave para tareas que requieren procesamiento sensible al contexto.

Consideremos la frase “El gato, que según varias fuentes fidedignas duerme la siesta, persigue al ratón”. Un SSM lineal no puede “decidir” ignorar la cláusula subordinada para conectar “gato” con

“persigue”: aplica la misma transformación a cada token, sin capacidad de selección. Del mismo modo, en modelado de lenguaje, cuando el modelo encuentra un delimitador como un punto y seguido, debería poder “reiniciar” parcialmente su estado para indicar que empieza un nuevo contexto, pero un SSM con Δ fijo no puede hacer esto.

Esta limitación también explica por qué la formulación convolucional es posible: al ser $(\bar{\mathbf{A}}, \bar{\mathbf{B}}, \mathbf{C})$ independientes de la entrada, el kernel $\bar{\mathbf{K}}$ se puede precalcular. Hacer los parámetros dependientes de la entrada rompe esta propiedad, eliminando la posibilidad de usar FFT, pero es necesario para dotar al modelo de verdadera expresividad.

6.6.2. El mecanismo de selección

Mamba [17] resuelve esta limitación haciendo que los parámetros $\mathbf{B}$, $\mathbf{C}$ y Δ sean **funciones de la entrada**:

$$\mathbf{B}_k = \text{Linear}_B(\mathbf{x}_k) \in \mathbb{R}^N \quad (6.21)$$

$$\mathbf{C}_k = \text{Linear}_C(\mathbf{x}_k) \in \mathbb{R}^N \quad (6.22)$$

$$\Delta_k = \text{softplus}(\text{Linear}_\Delta(\mathbf{x}_k)) \in \mathbb{R}^+ \quad (6.23)$$

donde $\mathbf{x}_k$ es la representación del token en la posición k y $\text{softplus}(x) = \log(1 + e^x)$ garantiza que $\Delta_k > 0$. La matriz $\mathbf{A}$ se mantiene como parámetro fijo (diagonal, inicializada con HiPPO), ya que hacer $\mathbf{A}$ dependiente de la entrada complicaría la discretización sin beneficio claro.

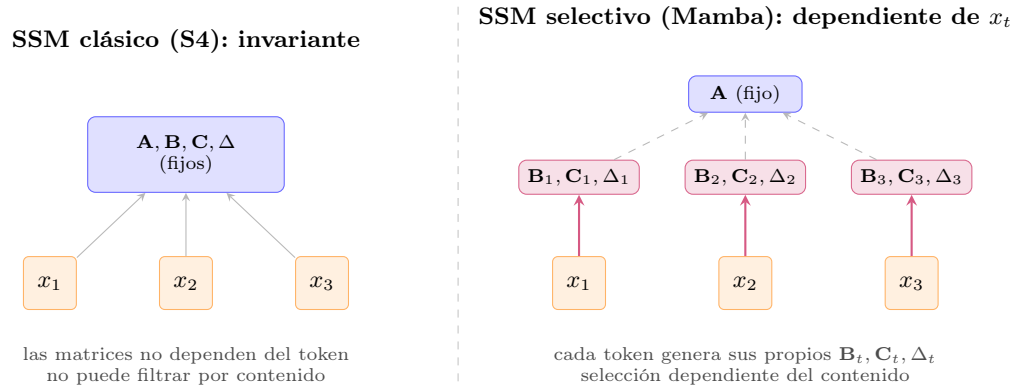


Figura 6.2: Invariancia frente a selección. En los SSM clásicos (S4), las matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$ y el paso Δ son fijos y se aplican por igual a todos los tokens. En Mamba, $\mathbf{B}_t, \mathbf{C}_t, \Delta_t$ se proyectan a partir de cada token x_t —mientras $\mathbf{A}$ permanece fijo y es modulado por Δ_t —, lo que permite al modelo decidir qué fracción de cada entrada se incorpora al estado en función de su contenido.

El parámetro Δ_k es especialmente importante y admite una interpretación precisa. Bajo la discretización ZOH (ecuación 6.5), Δ_k controla la escala temporal del modelo en cada paso:

- Cuando Δ_k es **pequeño**: $\bar{\mathbf{A}}_k = \exp(\Delta_k \mathbf{A}) \approx \mathbf{I}$ y $\bar{\mathbf{B}}_k \approx \Delta_k \mathbf{B}_k \approx \mathbf{0}$. El estado se **preserva** casi intacto y la entrada actual tiene poca influencia. El modelo “ignora” el token actual, manteniendo su memoria del contexto anterior.
- Cuando Δ_k es **grande**: $\bar{\mathbf{A}}_k \rightarrow \mathbf{0}$ y $\bar{\mathbf{B}}_k \rightarrow -\mathbf{A}^{-1} \mathbf{B}_k$. El estado anterior se “olvida” y se **reinicia** con la nueva entrada. El modelo “atiende” al token actual, incorporándolo fuertemente al estado.

Este mecanismo es funcionalmente análogo a la puerta de olvido de la LSTM, pero con una diferencia importante: en la LSTM, la puerta depende tanto del estado anterior como de la entrada

actual ($f_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}, \mathbf{x}_t])$), mientras que en Mamba Δ_k depende **únicamente de la entrada** ($\Delta_k = \text{softplus}(\mathbf{w}^T \mathbf{x}_k)$). Esta simplificación no es accidental, sino un requisito para la eficiencia computacional: dado que Δ_k , $\mathbf{B}_k$ y $\mathbf{C}_k$ dependen solo de $\mathbf{x}_k$, las matrices discretizadas $\bar{\mathbf{A}}_k$ y $\bar{\mathbf{B}}_k$ pueden precalcularse en paralelo para todas las posiciones de la secuencia *antes* de evaluar la recurrencia. Si Δ dependiera del estado $\mathbf{h}_{k-1}$ (como en la LSTM), la discretización en la posición k requeriría conocer el resultado de la posición $k-1$, creando una dependencia secuencial completa que impediría toda paralelización.

6.6.3. El algoritmo de escaneo paralelo selectivo

Al depender los parámetros de la entrada, la formulación convolucional deja de ser aplicable (el kernel cambia en cada posición, por lo que no existe un kernel $\bar{\mathbf{K}}$ fijo). Mamba necesita calcular la recurrencia (6.8) pero sin perder la eficiencia de la paralelización.

La solución es el **escaneo paralelo** (*parallel scan* o *prefix sum*), un algoritmo clásico de computación paralela que puede evaluar recurrencias asociativas en $\mathcal{O}(\log L)$ pasos paralelos en lugar de L pasos secuenciales. La observación clave es que la recurrencia $\mathbf{h}_k = \bar{\mathbf{A}}_k \mathbf{h}_{k-1} + \bar{\mathbf{B}}_k u_k$ se puede reescribir como una operación asociativa sobre pares $(\bar{\mathbf{A}}_k, \bar{\mathbf{B}}_k u_k)$, y el escaneo paralelo se puede aplicar a cualquier operación asociativa.

Mamba implementa este escaneo con **kernels CUDA optimizados**¹ que explotan la jerarquía de memoria de la GPU (SRAM vs. HBM), de forma análoga a FlashAttention. En lugar de materializar las matrices $\bar{\mathbf{A}}_k$ y $\bar{\mathbf{B}}_k$ de tamaño $N \times N$ en memoria global, las calcula en la memoria rápida de la GPU (SRAM) procesando la secuencia por bloques. Esto reduce el cuello de botella de memoria y permite que la implementación sea prácticamente tan rápida como una convolución, a pesar de que el sistema ya no es LTI.

6.6.4. Arquitectura del bloque Mamba

El bloque Mamba sigue una estructura inspirada en el bloque Transformer, pero reemplazando la atención por el SSM selectivo y el par FFN+atención por un diseño más integrado:

1. **Proyección de entrada:** Se aplica una proyección lineal que expande la dimensión por un factor E (típicamente $E = 2$), creando dos ramas de dimensión Ed .
2. **Rama SSM:** Una rama pasa por una convolución 1D causal (con kernel de tamaño d_{conv} , típicamente 4), seguida de una activación SiLU y el módulo SSM selectivo descrito anteriormente. La convolución causal actúa como un contexto local previo al SSM, similar al procesamiento n-grama.
3. **Rama de puerta:** La otra rama pasa directamente por una activación SiLU y actúa como puerta multiplicativa (análoga a la puerta de la GLU vista en la sección 5.2.2).
4. **Fusión:** Ambas ramas se multiplican elemento a elemento: $\mathbf{y} = \text{SSM}(\sigma(\text{Conv1D}(\mathbf{x}_1))) \odot \sigma(\mathbf{x}_2)$.
5. **Proyección de salida:** Una proyección lineal reduce la dimensión de Ed al tamaño original d .

Al igual que en el Transformer, se apilan múltiples bloques con conexiones residuales y normalización (típicamente RMSNorm con pre-normalización):

$$\mathbf{x}_{l+1} = \mathbf{x}_l + \text{MambaBlock}(\text{RMSNorm}(\mathbf{x}_l)) \quad (6.24)$$

El resultado es una arquitectura más simple que el Transformer (no hay mecanismo de atención, cabezas múltiples, ni capa FFN separada) pero con una expresividad comparable en muchas tareas.

¹Estos kernels se distribuyen en los paquetes `mamba-ssm` y `causal-conv1d`, que requieren versiones específicas de CUDA, PyTorch y del ABI de C++. A diferencia del Transformer, cuya implementación es autocontenida en PyTorch, la dependencia de Mamba en kernels compilados externos introduce restricciones de reproducibilidad que deben tenerse en cuenta en la práctica.

Los parámetros totales de un bloque Mamba son aproximadamente $3Ed^2 + 2EdN + Ed \cdot d_{conv}$, que para valores típicos ($E = 2$, $N = 16$, $d_{conv} = 4$) resulta en $\sim 6d^2$, comparable a los $\sim 12d^2$ de un bloque Transformer (atención + FFN).

6.6.5. El estado como cuello de botella de información

Un aspecto fundamental de la arquitectura Mamba —compartido con todos los modelos recurrentes— es que toda la información sobre la secuencia pasada se comprime en un vector de estado $\mathbf{h}_k \in \mathbb{R}^N$ de dimensión fija N (el parámetro `d_state`, típicamente $N = 16$). A diferencia de la atención, que preserva el acceso directo a cada token anterior mediante la caché KV de tamaño proporcional a L , Mamba debe codificar el historial completo de la secuencia en estos N grados de libertad por canal.

Esta compresión tiene una consecuencia directa sobre el escalado en longitud: a medida que la secuencia crece, el ratio N/L decrece y la pérdida de información por compresión se agrava. Para $N = 16$ y $L = 128$, cada componente del estado resume en promedio 8 tokens; para $L = 1024$, resume 64. El mecanismo de selección (Δ_k dependiente de la entrada) permite priorizar qué información retener, pero no puede superar la capacidad intrínseca del estado. Esta limitación es análoga a la del estado oculto del LSTM ($\mathbf{h}_t \in \mathbb{R}^d$), con la diferencia de que en Mamba cada uno de los d_{model} canales mantiene su propio estado de dimensión N , resultando en un estado total de $d_{\text{model}} \times N$ escalares, lo que proporciona mayor capacidad agregada que un LSTM de la misma dimensión.

En el capítulo experimental verificaremos si esta limitación teórica se manifiesta en la práctica como una saturación del rendimiento de Mamba al aumentar la longitud de secuencia más allá de cierto umbral.

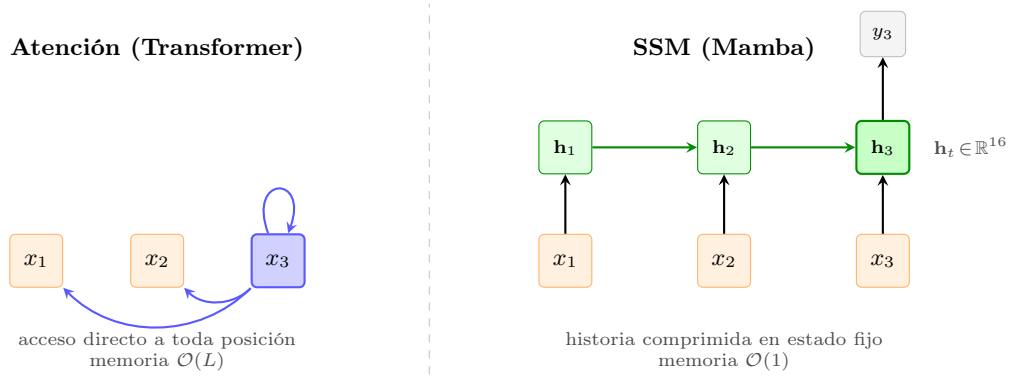


Figura 6.3: Acceso a la información en atención frente a SSM. En la atención, cada posición accede directamente a todas las demás, a costa de una memoria que crece con la longitud. En un SSM, toda la historia se comprime en un estado de dimensión fija $\mathbf{h}_t \in \mathbb{R}^{d_{\text{state}}}$ ($d_{\text{state}} = 16$ en nuestros experimentos): la salida en cada posición solo puede leer ese estado, no los tokens originales. Este cuello de botella es la causa estructural del colapso de Mamba en tareas de recuperación basada en contenido (Capítulo 7).

6.7. La dualidad SSM-Atención

Un resultado teórico reciente que conecta ambos paradigmas es la **dualidad entre SSM y atención**. Mamba-2 [9] demuestra que existe una equivalencia formal entre ciertos SSM estructurados y una forma restringida de atención lineal.

La observación parte de escribir la salida del SSM de forma matricial. Definiendo la **matriz de mezcla** $\mathbf{M} \in \mathbb{R}^{L \times L}$ como:

$$M_{ij} = \begin{cases} \mathbf{C}_i^T \left(\prod_{s=j+1}^i \overline{\mathbf{A}}_s \right) \overline{\mathbf{B}}_j & \text{si } i \geq j \\ 0 & \text{si } i < j \end{cases} \quad (6.25)$$

la salida del SSM se puede escribir como $\mathbf{y} = \mathbf{M}\mathbf{u}$, donde $\mathbf{M}$ es una matriz triangular inferior (causal). Cuando $\bar{\mathbf{A}}_k$ es un escalar (o una matriz diagonal), esta expresión se asemeja a una forma de atención con una estructura particular.

Concretamente, Dao y Gu [9] demuestran que bajo ciertas condiciones sobre la estructura de $\mathbf{A}$ (matrices **semiseparables**), la matriz $\mathbf{M}$ puede descomponerse y calcularse de formas equivalentes: como recurrencia ($\mathcal{O}(LN)$ secuencial), como convolución ($\mathcal{O}(L \log L)$ paralelo, solo para SSM lineales), o como un producto matricial similar a la atención ($\mathcal{O}(L^2N)$ paralelo). Este resultado, denominado *State Space Duality* (SSD), permite elegir el algoritmo más eficiente según el hardware y la longitud de la secuencia.

Arquitectónicamente, el resultado SSD da lugar a **Mamba-2**, que introduce tres cambios respecto a Mamba-1. El primero restringe $\mathbf{A}_k$ a una estructura escalar (compartida por todas las dimensiones del estado), lo que convierte el producto $\prod_s \bar{\mathbf{A}}_s$ en un simple producto de escalares y habilita la formulación como atención por bloques. A ello se suman las **cabezas de estado** múltiples, análogas a las cabezas de atención del Transformer: cada cabeza mantiene su propio estado SSM de dimensión N , y la salida se concatena, lo que aumenta la capacidad representacional sin escalar N individualmente. Por último, reemplaza el escaneo paralelo selectivo por un **algoritmo por bloques** (*chunk-wise*): la secuencia se divide en bloques de tamaño B , el procesamiento intra-bloque se realiza mediante un producto matricial de tamaño $B \times B$ (la “atención” implícita del SSM), y el procesamiento inter-bloque se realiza mediante la recurrencia del estado. Este diseño explota mejor las jerarquías de memoria de las GPUs modernas y permite un *throughput* significativamente mayor que Mamba-1 en la práctica.

Esta conexión teórica sugiere que la atención y los SSM no son paradigmas completamente disjuntos, sino extremos de un espectro. La atención completa ofrece máxima expresividad (cualquier par de posiciones puede interactuar directamente) pero con coste cuadrático. Los SSM comprimen toda la información en un estado de tamaño fijo, logrando coste lineal pero con capacidad limitada de recuperación precisa de información (el estado actúa como un “cuello de botella” de información). Arquitecturas futuras podrían explotar esta dualidad para combinar las ventajas de ambos enfoques según las necesidades de cada capa o tarea.

6.8. Arquitecturas híbridas: Jamba

6.8.1. Motivación: limitaciones complementarias

Aunque los modelos de espacio de estados como Mamba han demostrado ventajas significativas en eficiencia, presentan limitaciones en tareas que requieren una recuperación precisa de información a partir de posiciones específicas del contexto. El mecanismo de atención, por su parte, permite acceder directamente a cualquier posición de la secuencia mediante las puntuaciones de atención, pero con un coste cuadrático que limita su escalabilidad.

Esta observación sugiere que ambos mecanismos son complementarios. Los SSMs comprimen eficientemente el contexto en un estado latente de tamaño fijo, lo que los hace idóneos para capturar tendencias globales y dependencias difusas a lo largo de la secuencia. Sin embargo, esta compresión implica inevitablemente una pérdida de información: el estado latente no puede retener cada token individual de una secuencia larga. La atención, en cambio, preserva el acceso directo a todas las posiciones, a costa de materializar una matriz de tamaño $L \times L$. La pregunta natural es si es posible combinar ambos paradigmas en una arquitectura que herede las fortalezas de cada uno.

6.8.2. Arquitectura

Jamba, propuesto por AI21 Labs [24], materializa esta idea mediante una arquitectura que intercala capas Mamba con capas de autoatención estándar. Cada capa se complementa con una red de propagación directa (MLP), normalización RMSNorm y conexiones residuales, siguiendo el esquema pre-norm habitual en arquitecturas modernas. La unidad básica es, por tanto, un bloque residual que

contiene bien un SSM selectivo (capa Mamba) o bien un módulo de autoatención multicabeza (capa de atención), seguido de una FFN.

El aspecto más distintivo de Jamba es la flexibilidad en la proporción de capas de cada tipo. En la configuración original a gran escala, los autores usan un ratio de 1:7 (una capa de atención por cada siete capas Mamba), lo que reduce drásticamente el coste computacional y de memoria respecto a un Transformer puro. Sin embargo, esta proporción es un hiperparámetro que puede ajustarse según los recursos disponibles y los objetivos de la tarea. En nuestra implementación experimental, con solo 4 capas, utilizamos un ratio de 1:1 con la disposición M–A–M–A (una capa Mamba, una de atención, alternadas), lo que permite evaluar la hibridación en condiciones donde ambos mecanismos contribuyen de forma equilibrada.

Un hallazgo relevante del trabajo original es que las capas Mamba proporcionan implícitamente información posicional suficiente para el modelo, haciendo innecesario el uso de codificaciones posicionales explícitas como embeddings aprendidos, RoPE o ALiBi. Esto resulta notable porque significa que las capas de atención en Jamba operan sin ninguna inyección de información posicional propia: la noción de orden proviene exclusivamente de las capas SSM que las preceden. En nuestra implementación respetamos esta decisión de diseño.

6.8.3. Ventajas en memoria e inferencia

Una de las principales ventajas prácticas de Jamba se manifiesta durante la inferencia autoregresiva. En un Transformer estándar, cada capa de atención mantiene una caché de keys y values (KV cache) que crece linealmente con la longitud de la secuencia generada. Para modelos profundos con contextos largos, esta caché puede dominar el consumo de memoria. En Jamba, solo las capas de atención requieren KV cache; las capas Mamba mantienen un estado latente de tamaño fijo independiente de la longitud de la secuencia. En la configuración original con ratio 1:7, esto supone una reducción de hasta 8 veces en la memoria de KV cache respecto a un Transformer del mismo número total de capas.

Durante el entrenamiento, las ventajas son más matizadas. Las capas Mamba se benefician de su complejidad lineal y no materializan ninguna matriz de atención, pero las capas de atención sí incurrir en el coste cuadrático habitual. El resultado neto depende de la proporción: con más capas Mamba, el modelo se acerca al perfil de eficiencia de un SSM puro; con más capas de atención, se acerca al de un Transformer. Esta modularidad permite diseñar configuraciones adaptadas al hardware disponible.

6.8.4. Hallazgos del entrenamiento a escala

El entrenamiento de Jamba a escala (~ 52 B parámetros en la configuración original) reveló varias propiedades interesantes para la comunidad de investigación.

En primer lugar, los autores encontraron que la combinación de capas Mamba-1 con capas de atención funciona mejor que la combinación con Mamba-2, a pesar de que Mamba-2 supera a Mamba-1 cuando se usa como arquitectura pura. Este resultado sugiere que la dualidad teórica SSM-atención establecida por Mamba-2 (Sección 6.7) no se traduce directamente en una ventaja cuando ambos mecanismos coexisten en la misma arquitectura. La hipótesis propuesta es que, al ser Mamba-2 más similar a la atención que Mamba-1, la hibridación con capas de atención explícitas aporta menor diversidad de sesgos inductivos.

En segundo lugar, la estabilización del entrenamiento requirió la incorporación de RMSNorm dentro de las capas Mamba, una modificación no necesaria cuando Mamba se entrena de forma aislada a escalas menores. Esto apunta a que la interacción entre capas de naturaleza distinta introduce dinámicas de gradiente que deben gestionarse con cuidado.

Es importante señalar que estos hallazgos se obtuvieron a una escala (~ 52 B parámetros, decenas de capas) muy diferente a la de nuestros experimentos (~ 28 M parámetros, 4 capas). Fenómenos como la necesidad de RMSNorm interna o la superioridad de Mamba-1 sobre Mamba-2 en hibridación podrían depender de la escala: a menor profundidad, las dinámicas de gradiente son más estables y la diversidad de sesgos inductivos podría importar menos. Nuestros experimentos no pueden confirmar ni refutar

directamente estos hallazgos, pero sí pueden verificar si las ventajas cualitativas de la hibridación — mejor perplejidad que Mamba puro, mejor eficiencia que un Transformer puro— se manifiestan también a escala reducida.

6.8.5. Relevancia para nuestro estudio

Desde la perspectiva de nuestro análisis comparativo, Jamba ocupa un papel central porque permite responder a una pregunta que ni los Transformers puros ni los SSMs puros pueden abordar: ¿aporta la hibridación un beneficio real frente a cada paradigma por separado?

Si Mamba puro ofrece eficiencia lineal pero perplejidad limitada, y el Transformer con codificación posicional relativa ofrece buena perplejidad pero con coste cuadrático, Jamba debería idealmente combinar la calidad del segundo con la eficiencia del primero. Nuestro diseño experimental, que evalúa todos estos modelos bajo las mismas condiciones y con variación sistemática de la longitud de secuencia, permite verificar empíricamente esta hipótesis y cuantificar hasta qué punto la hibridación justifica la complejidad adicional de la arquitectura.

6.9. Resumen comparativo de arquitecturas

Para concluir la parte teórica, la Tabla 6.1 resume las principales características de todas las arquitecturas estudiadas en este trabajo.

Tabla 6.1: Comparativa de arquitecturas para el procesamiento de secuencias

Arquitectura	Complejidad (entrenamiento) ²	Complejidad (inferencia/token)	Paralelizable	Dependencias largas	Depend. de la entrada
FFN	$\mathcal{O}(L)$	$\mathcal{O}(1)$	Sí	No	No
RNN/LSTM	$\mathcal{O}(L)$	$\mathcal{O}(1)$	No	Limitadas	Sí
Transformer	$\mathcal{O}(L^2)$	$\mathcal{O}(L)$	Sí	Sí	Sí
Transformer + RoPE	$\mathcal{O}(L^2)$	$\mathcal{O}(L)$	Sí	Sí (relativa)	Sí
Transformer + ALiBi	$\mathcal{O}(L^2)$	$\mathcal{O}(L)$	Sí	Sí (relativa)	Sí
Transformer + MoE	$\mathcal{O}(L^2)$	$\mathcal{O}(L)$	Sí	Sí	Sí
SSM (S4)	$\mathcal{O}(L \log L)$	$\mathcal{O}(1)$	Sí (FFT)	Sí (HiPPO)	No
Mamba	$\mathcal{O}(L)$	$\mathcal{O}(1)$	Sí (scan)	Sí (selectivo)	Sí

La tabla revela una evolución clara. La FFN no captura relaciones secuenciales. Las RNN/LSTM ofrecen complejidad lineal y dependencia de la entrada, pero no son paralelizables. El Transformer resuelve la paralelización e introduce atención global, pero con coste cuadrático. Las variantes con codificación posicional relativa (RoPE, ALiBi) mejoran la generalización a secuencias largas sin reducir la complejidad. MoE escala la capacidad manteniendo el coste por token. Los SSM lineales (S4) logran complejidad casi-lineal y paralelización, pero pierden la dependencia de la entrada. Finalmente, Mamba

combina las mejores propiedades: complejidad lineal, paralelización durante el entrenamiento, coste constante por token en inferencia y dependencia selectiva de la entrada.

Es importante matizar que esta comparación teórica no implica que Mamba sea superior en todas las tareas. Los Transformers, gracias a su atención global, poseen una capacidad intrínseca para recuperar información de cualquier posición con precisión, capacidad que Mamba sacrifica al comprimir la información en un estado de tamaño fijo. El rendimiento relativo depende de la naturaleza de la tarea, la longitud de las secuencias y los recursos disponibles. En el siguiente capítulo exploraremos empíricamente estas diferencias.

Capítulo 7

Evaluación experimental comparativa

En los capítulos anteriores hemos presentado los fundamentos teóricos de las principales familias de arquitecturas para modelado de secuencias: redes de propagación directa, redes recurrentes (LSTM), Transformers con distintas codificaciones posicionales, modelos de espacio de estados (Mamba) y arquitecturas híbridas (Jamba). También hemos identificado el problema central que motiva este trabajo: la relación entre la longitud de secuencia y el rendimiento de cada arquitectura.

Esta relación no es trivial. A nivel teórico, los Transformers pueden atender a cualquier posición dentro de la ventana de contexto gracias a su mecanismo de atención global, pero el coste cuadrático en memoria y cómputo limita en la práctica las longitudes alcanzables. Las arquitecturas recurrentes procesan la secuencia de forma lineal, lo que las hace eficientes en inferencia, pero las expone al problema del desvanecimiento del gradiente cuando las dependencias relevantes se encuentran a gran distancia. Los modelos de espacio de estados, por su parte, proponen una vía intermedia: mantienen un estado comprimido de tamaño fijo que se actualiza en cada paso temporal, lo que les confiere complejidad lineal sin renunciar —al menos en principio— a la capacidad de capturar dependencias a largo alcance. Las arquitecturas híbridas combinan estos mecanismos con la hipótesis de que distintas capas pueden especializarse en capturar patrones a diferentes escalas temporales. Sin embargo, la pregunta de cuándo y en qué medida cada familia de arquitecturas se beneficia realmente de secuencias más largas permanece abierta, y las comparaciones publicadas rara vez controlan simultáneamente el número de parámetros, los recursos computacionales y el protocolo de entrenamiento.

En este capítulo abordamos esta cuestión de forma empírica. Comparamos ocho modelos entrenados desde cero bajo condiciones controladas en dos tareas complementarias: modelado de lenguaje (WikiText-103) y razonamiento estructural (LRA ListOps). Las dos tareas se han elegido porque plantean exigencias distintas sobre la capacidad de modelado de secuencias: WikiText-103 requiere capturar coherencia estadística local y dependencias gramaticales a media distancia, mientras que ListOps demanda un razonamiento composicional estrictamente jerárquico sobre secuencias de longitud variable. El objetivo no es alcanzar el estado del arte en cada tarea, sino aislar el efecto de la longitud de secuencia sobre el rendimiento de cada arquitectura en igualdad de condiciones.

7.1. Modelos evaluados

Los ocho modelos seleccionados cubren los principales paradigmas de modelado de secuencias. La Tabla 7.1 resume sus características.

Modelo	Paradigma	Complejidad	Parámetros
FFN	Propagación directa	$\mathcal{O}(L)$	25.8M
LSTM	Recurrente	$\mathcal{O}(L)$	26.8M
Transformer	Atención (PE aprendidos)	$\mathcal{O}(L^2)$	29.0M
Transformer + RoPE	Atención (PE relativas)	$\mathcal{O}(L^2)$	28.9M
Transformer + ALiBi	Atención (sesgo lineal)	$\mathcal{O}(L^2)$	28.9M
Transformer + RoPE + MoE	Atención + MoE	$\mathcal{O}(L^2)$	35.2M
Mamba	Espacio de estados (SSM)	$\mathcal{O}(L)$	27.5M
Jamba	Híbrido SSM + Atención	$\mathcal{O}(L^2)$	28.2M

Tabla 7.1: Resumen de los ocho modelos evaluados. La complejidad se refiere a la longitud de secuencia L . Los parámetros corresponden a la configuración con $d_{\text{model}} = 256$ y $L = 256$.

La **FFN** (red de propagación directa) sirve como línea base inferior: carece de todo mecanismo de dependencia secuencial. En modelado de lenguaje, genera una representación del contexto mediante la media acumulada de los embeddings anteriores, lo que la convierte en un modelo puramente estadístico sin noción de orden.

El **LSTM** [19] representa el paradigma recurrente clásico. Su estado oculto se actualiza secuencialmente mediante puertas que regulan el flujo de información, lo que le otorga complejidad lineal en la longitud de la secuencia pero limita el paralelismo durante el entrenamiento.

El **Transformer** con embeddings posicionales aprendidos constituye nuestro modelo de referencia para la familia de atención. Utiliza embeddings de posición entrenables, lo que implica que la representación de cada posición se aprende durante el entrenamiento sin ninguna estructura geométrica impuesta. Su complejidad es cuadrática en la capa de autoatención.

El **Transformer con RoPE** reemplaza los embeddings posicionales aprendidos por Rotary Position Embeddings [36], que codifican las posiciones relativas mediante rotaciones en el espacio de los embeddings. Mantiene la misma complejidad cuadrática pero con mejor capacidad de extrapolación a secuencias largas.

El **Transformer con ALiBi** sustituye toda codificación posicional explícita por sesgos lineales añadidos directamente a las puntuaciones de atención [29], que penalizan las posiciones lejanas de forma proporcional a su distancia sin añadir parámetros entrenables.

El **Transformer con RoPE y MoE** combina la codificación posicional RoPE con una capa de Mixture of Experts [33] que reemplaza la FFN del bloque Transformer. En cada paso, un router selecciona los 2 expertos más relevantes de un total de 4, lo que incrementa la capacidad del modelo sin multiplicar proporcionalmente el coste computacional por token. Este modelo tiene un número de parámetros significativamente mayor ($\sim 35\text{M}$ frente a $\sim 29\text{M}$ de los demás Transformers) debido a los expertos adicionales.

Mamba representa el paradigma de los modelos de espacio de estados con selectividad dependiente de la entrada [17]. Su complejidad es lineal y no materializa ninguna matriz de atención, lo que le otorga ventajas teóricas en memoria y *throughput* para secuencias largas.

Jamba es una arquitectura híbrida que alterna capas Mamba con capas de autoatención [24], buscando combinar la eficiencia lineal de los SSMs con la expresividad del mecanismo de atención. En

nuestra implementación, la disposición sigue el patrón M–A–M–A (4 capas en total, con atención cada 2 capas).

Para que la comparación sea lo más justa posible, todos los modelos comparten la misma dimensión del modelo ($d_{\text{modelo}} = 256$), el mismo número de capas ($N_{\text{layers}} = 4$), el mismo optimizador (AdamW con $\text{lr} = 3 \times 10^{-4}$), y se entrenan sobre los mismos datos durante el mismo número de épocas. Como muestra la Tabla 7.1, el número total de parámetros se mantiene en un rango comparable (25.8–29.0M) para todos los modelos excepto el MoE (35.2M), cuyo exceso se justifica por la naturaleza misma de la arquitectura de expertos.

Antes de presentar los resultados, es necesario señalar una limitación metodológica importante. Debido a las restricciones de cómputo, cada configuración (modelo $\times$ longitud) se ha entrenado con una única semilla aleatoria. Esto implica que las diferencias de perplejidad entre modelos del grupo competitivo —por ejemplo, RoPE (55.8) frente a ALiBi (53.4) a $L = 1024$ — podrían encontrarse dentro del margen de variabilidad entre ejecuciones, y deben interpretarse con la cautela correspondiente. Las conclusiones que siguen se basan en tendencias consistentes a lo largo de las cuatro longitudes evaluadas, no en comparaciones puntuales entre pares de modelos.

7.2. WikiText-103: modelado de lenguaje

La primera tarea evalúa la capacidad de modelado de lenguaje mediante predicción del siguiente token (*next-token prediction*) sobre el corpus WikiText-103 [26]. Este corpus contiene aproximadamente 103 millones de tokens extraídos de artículos verificados de Wikipedia y constituye un benchmark estándar para evaluar modelos de lenguaje a escala moderada; sus características y estadísticas se describen en detalle en el Apéndice A. La elección de esta tarea como punto de partida se debe a que el modelado de lenguaje expone de forma directa la capacidad de cada arquitectura para explotar el contexto disponible: un modelo que aprovecha secuencias más largas producirá predicciones más precisas y, por tanto, una perplejidad menor.

7.2.1. Configuración experimental

Cada modelo recibe secuencias de longitud fija L (denominada *block size*) y predice el token en cada posición dado el contexto anterior. La métrica principal es la perplejidad (*perplexity*) sobre el conjunto de test, definida como $\text{PPL} = \exp(\mathcal{L})$ donde $\mathcal{L}$ es la entropía cruzada media por token. Valores menores indican mejor capacidad predictiva.

El aspecto central de nuestro diseño experimental es la variación sistemática de la longitud de secuencia. Entrenamos cada modelo con cuatro longitudes: $L \in \{128, 256, 512, 1024\}$. Para garantizar una comparación justa, fijamos el número total de tokens procesados por GPU en cada paso de entrenamiento a 8192, ajustando el tamaño de lote inversamente a la longitud de secuencia (por ejemplo, *batch size* de 64 para $L = 128$ y de 8 para $L = 1024$). De este modo, la cantidad total de tokens vistos durante el entrenamiento es prácticamente idéntica para todas las longitudes ($\sim 567\text{M}$ tokens por modelo), aislando el efecto de la longitud de contexto disponible.

Conviene precisar el alcance de este diseño. Las longitudes evaluadas ($L \leq 1024$ en WikiText-103 y $L \leq 2048$ en LRA ListOps) se sitúan en un régimen accesible y reproducible, no en el de contexto extremo ($\geq 10^4$ tokens) donde el coste cuadrático del Transformer se vuelve prohibitivo. El objetivo no es, por tanto, caracterizar el punto en que la atención deja de ser viable, sino cómo *comienza* a manifestarse la sensibilidad a la longitud y qué propiedad arquitectónica la gobierna en cada tarea. Los patrones cualitativos que identificamos —degradación, saturación y mejora— emergen ya en este rango, y su extrapolación a contextos mayores se discute como limitación en el Capítulo 8.

La Tabla 7.2 resume los hiperparámetros compartidos por todos los modelos. El entrenamiento se realizó en un clúster con 8 GPUs NVIDIA A10G utilizando PyTorch DDP (*Distributed Data Parallel*), sin precisión mixta (AMP desactivada por incompatibilidades con los kernels CUDA de Mamba). Se utilizó gradient checkpointing para reducir el consumo de memoria.

Hiperparámetro	Valor
d_{model}	256
d_{ffn}	1024
Capas (N_{layers})	4
Cabezas de atención	8
Dropout	0.1
Optimizador	AdamW
Learning rate	3×10^{-4}
Weight decay	0.01
Gradient clipping	1.0
Épocas	5
Tokens por GPU por step	8192
GPUs (DDP)	$8 \times \text{NVIDIA A10G}$
Precisión mixta (AMP)	No
Gradient checkpointing	Sí
Semilla	42

Tabla 7.2: Hiperparámetros compartidos en los experimentos de WikiText-103.

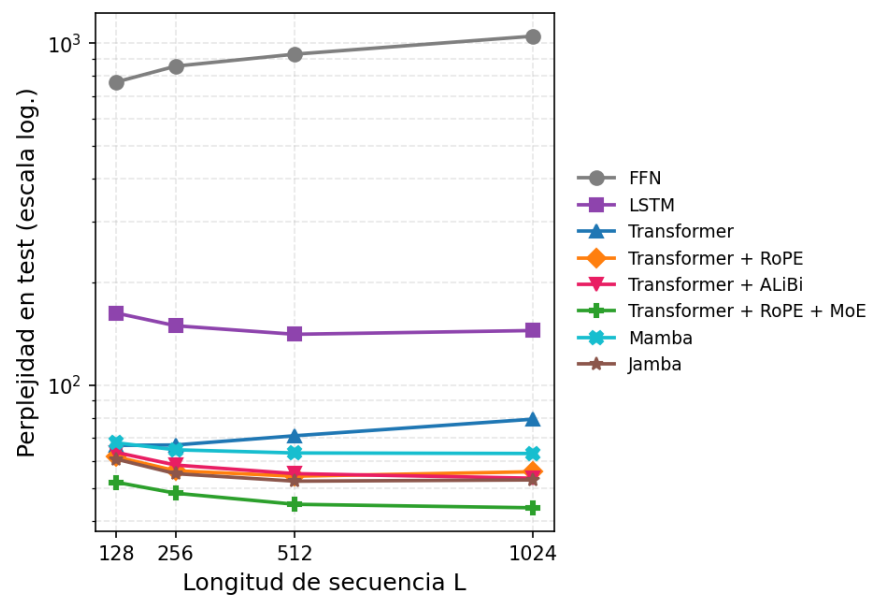
7.2.2. Resultados de perplejidad

La Tabla 7.3 presenta la perplejidad en test para los ocho modelos en las cuatro longitudes evaluadas. La columna Δ indica el cambio relativo en perplejidad al pasar de $L = 128$ a $L = 1024$: valores negativos indican que el modelo aprovecha secuencias más largas; valores positivos, degradación.

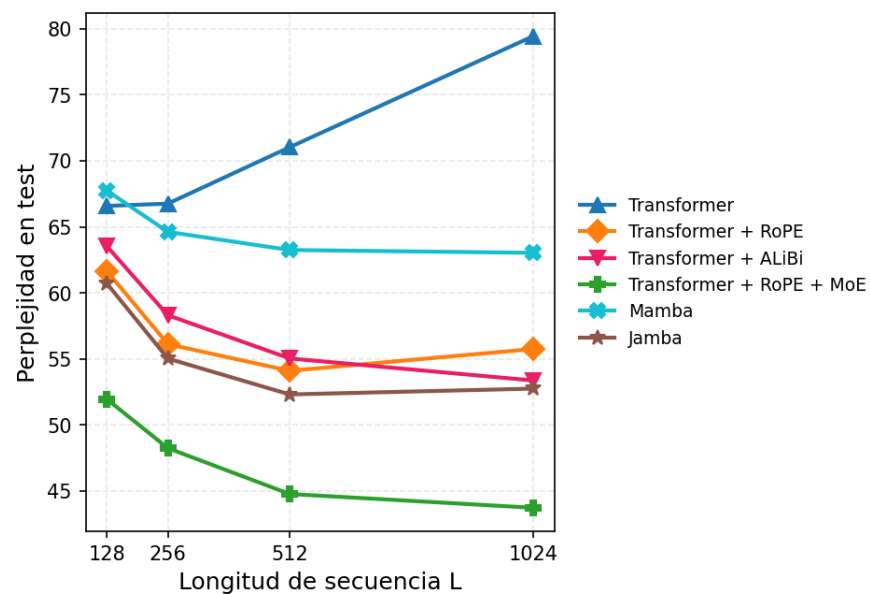
Modelo	$L=128$	$L=256$	$L=512$	$L=1024$	$\Delta_{128 \rightarrow 1024}$
FFN	768.6	855.4	927.3	1046.9	+36,2 %
LSTM	162.3	149.0	140.6	144.3	-11,1 %
Transformer	66.6	66.8	71.0	79.4	+19,3 %
Transformer + RoPE	61.6	56.1	54.1	55.8	-9,6 %
Transformer + ALiBi	63.5	58.3	55.0	53.4	-16,0 %
Transformer + RoPE + MoE	51.9	48.2	44.8	43.7	-15,8 %
Mamba	67.7	64.6	63.2	63.0	-7,0 %
Jamba	60.7	55.0	52.3	52.8	-13,1 %

Tabla 7.3: Perplejidad en test (menor es mejor) de los ocho modelos en WikiText-103 para cuatro longitudes de secuencia. $\Delta_{128 \rightarrow 1024}$ indica el cambio porcentual relativo.

La Figura 7.1 visualiza estos resultados. La vista panorámica (a) muestra la separación entre las tres familias de rendimiento: la FFN con perplejidades del orden de 10^3 , el LSTM en torno a 140–162, y el grupo de Transformers y SSMS por debajo de 80. La vista de detalle (b) amplía este último grupo, donde se aprecian con mayor claridad las diferencias entre arquitecturas y su evolución con la longitud.



(a) Vista panorámica (escala logarítmica).



(b) Detalle de los modelos competitivos.

Figura 7.1: Perplejidad en test en función de la longitud de secuencia en WikiText-103.

Los resultados revelan tres patrones de escalado claramente diferenciados, que la Figura 7.2 resume de forma visual.

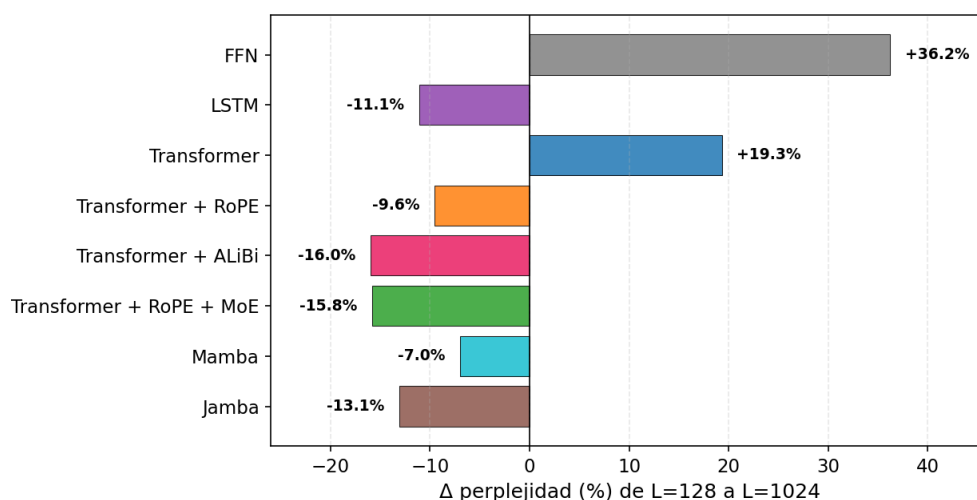


Figura 7.2: Cambio relativo en perplexidad al pasar de $L = 128$ a $L = 1024$. Valores negativos (hacia la izquierda) indican que el modelo aprovecha secuencias más largas; valores positivos indican degradación.

Degradación con la longitud. La FFN empeora monótonamente (+36,2%) al aumentar L , como cabe esperar de un modelo sin capacidad secuencial: bloques más largos implican más contexto que el modelo no puede explotar, y el promedio de embeddings sobre ventanas más amplias produce representaciones progresivamente más uniformes y menos discriminativas. Más revelador resulta el caso del Transformer con embeddings posicionales aprendidos, que se degrada globalmente un +19,3%. Aunque alcanza su mejor perplexidad a $L = 128$ (66.6), a partir de $L = 256$ empeora de forma sostenida hasta 79.4 a $L = 1024$. Es importante precisar que este deterioro no se debe a un problema de extrapolación, ya que cada modelo se entrena y se evalúa a la misma longitud. La causa es más bien que los embeddings posicionales absolutos, al asignar un vector independiente a cada posición, definen un espacio de representación que crece linealmente con L y resulta progresivamente más difícil de optimizar. En contraste, las codificaciones relativas (RoPE, ALiBi) codifican relaciones entre posiciones, lo que constituye un espacio de representación más compacto cuya dificultad de optimización no escala con la longitud de la secuencia.

Mejora con la longitud. Los Transformers con codificación posicional relativa y el modelo híbrido Jamba se benefician del aumento de contexto, aunque con matices importantes. ALiBi es el único Transformer puro con mejora estrictamente monótona en todo el rango (-16,0%), coherente con su diseño basado en un sesgo lineal que favorece la localidad sin fijar una longitud máxima. El modelo con MoE, basado en RoPE, muestra igualmente una mejora monótona (-15,8%) y alcanza la mejor perplexidad absoluta en todas las longitudes (43.7 a $L = 1024$), ventaja que se explica en parte por su mayor número de parámetros (~35M frente a ~29M). Comparando RoPE con y sin MoE a $L = 1024$ (55.8 frente a 43.7), la ganancia de -21,6% es atribuible íntegramente a la mayor capacidad del modelo, ya que ambos comparten la misma codificación posicional y el mismo régimen de entrenamiento.

RoPE y Jamba mejoran de forma consistente hasta $L = 512$ pero experimentan una ligera reversión a $L = 1024$: RoPE pasa de 54.1 a 55.8 (+3,0%) y Jamba de 52.3 a 52.8 (+0,8%). En el caso de RoPE, la reversión puede indicar una interacción desfavorable entre la base de frecuencia utilizada ($\theta = 10000$) y la longitud máxima: las frecuencias rotacionales más altas completan ciclos completos a $L = 1024$, lo que puede introducir ambigüedades posicionales. En Jamba, la reversión es marginal y consistente con ruido de una única semilla. Es notable que a $L = 128$ RoPE supera a ALiBi (61.6 frente a 63.5), pero a $L = 1024$ la relación se invierte (55.8 frente a 53.4): el sesgo de localidad implícito de ALiBi

se convierte en ventaja precisamente cuando el contexto es largo y el modelo debe discriminar entre dependencias a múltiples distancias.

Saturación. El LSTM mejora significativamente de $L = 128$ a $L = 512$ (de 162.3 a 140.6) pero invierte ligeramente su tendencia a $L = 1024$ (144.3). Este comportamiento es consistente con la capacidad limitada de su estado oculto de dimensión fija: una vez que el estado captura las dependencias que puede representar, el contexto adicional no aporta información explotable y la perplejidad se satura. La leve degradación a $L = 1024$ puede reflejar que el entrenamiento con secuencias más largas expone al LSTM a dependencias que su estado no puede codificar, introduciendo ruido en la optimización.

Mamba exhibe un patrón de saturación cualitativamente similar: mejora de forma consistente hasta $L = 512$ (63.2) pero apenas progresa al duplicar la longitud (63.0 a $L = 1024$), con una ganancia neta modesta de -7.0% . A diferencia de los resultados previos a las correcciones arquitectónicas¹, Mamba ya no empeora a $L = 1024$ sino que se estabiliza. No obstante, la saturación temprana sigue siendo notable: mientras que ALiBi o MoE continúan mejorando de $L = 512$ a $L = 1024$, Mamba ha agotado prácticamente su capacidad de mejora. La causa más probable es la dimensión del estado latente ($d_{\text{state}} = 16$), que actúa como cuello de botella de información: más allá de cierta longitud, el estado comprimido no puede retener las dependencias adicionales que el contexto más largo ofrece. Este mecanismo se ha analizado en detalle en la Sección 6.6.5.

Cabe destacar, finalmente, el rendimiento de Jamba, que alcanza la segunda mejor perplejidad en todas las longitudes a partir de $L = 256$ con un número de parámetros comparable al de los Transformers puros ($\sim 28\text{M}$). Su ventaja sobre Mamba puro (52.8 frente a 63.0 a $L = 1024$) sugiere que la combinación de capas SSM con capas de atención es sustancialmente más eficaz que cualquiera de los dos mecanismos por separado: las capas de atención aportan la expresividad global que las capas Mamba por sí solas no pueden ofrecer con un estado latente limitado.

7.2.3. Eficiencia computacional

Más allá de la calidad del modelado, la viabilidad práctica de cada arquitectura depende de su eficiencia cuando se escala a secuencias largas. Evaluamos tres métricas: el tiempo total de entrenamiento, el *throughput* (tokens procesados por segundo en las 8 GPUs) y el pico de memoria GPU. La Tabla 7.4 presenta las dos primeras para las longitudes extremas, y la Tabla 7.5 el pico de memoria para las cuatro longitudes.

¹Antes de añadir las conexiones residuales y la normalización pre-bloque ($\mathbf{x} = \mathbf{x} + \text{blk}(\text{norm}(\mathbf{x}))$), Mamba mostraba una degradación activa a $L = 1024$ (perplejidad de 81.8). Las correcciones eliminaron esta anomalía, confirmando que el patrón observado originalmente era un artefacto de implementación, no una limitación arquitectónica intrínseca.

Modelo	Tiempo de entrenamiento (s)			Throughput (Ktok/s)		
	$L=128$	$L=1024$	Δ	$L=128$	$L=1024$	Δ
FFN	686	679	-1,0 %	854	863	+1,1 %
LSTM	891	1394	+56,5 %	657	420	-36,1 %
Transformer	924	1275	+38,0 %	634	460	-27,5 %
Transformer + RoPE	931	1279	+37,4 %	629	459	-27,0 %
Transformer + ALiBi	930	1325	+42,5 %	630	443	-29,7 %
Transformer + RoPE + MoE	1178	1531	+30,0 %	495	382	-22,8 %
Mamba	861	850	-1,3 %	681	690	+1,3 %
Jamba	885	1054	+19,1 %	660	557	-15,6 %

Tabla 7.4: Tiempo de entrenamiento y *throughput* para $L = 128$ y $L = 1024$. Δ indica el cambio relativo.

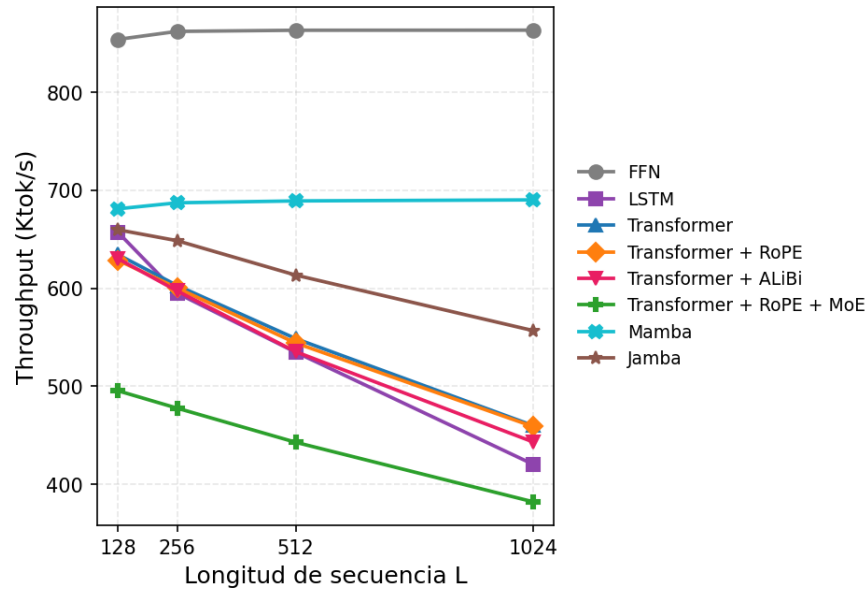
Modelo	Pico de memoria GPU (MB)			
	$L=128$	$L=256$	$L=512$	$L=1024$
FFN	6721	6721	6721	6721
LSTM	7124	7124	7124	7124
Transformer	7911	8200	8778	9935
Transformer + RoPE	7933	8223	8798	9955
Transformer + ALiBi	7934	8225	8802	9957
Transformer + RoPE + MoE	8471	8762	9338	10493
Mamba	7750	7742	7736	7735
Jamba	7789	7929	8215	8792

Tabla 7.5: Pico de memoria GPU (MB por GPU) en función de la longitud de secuencia.

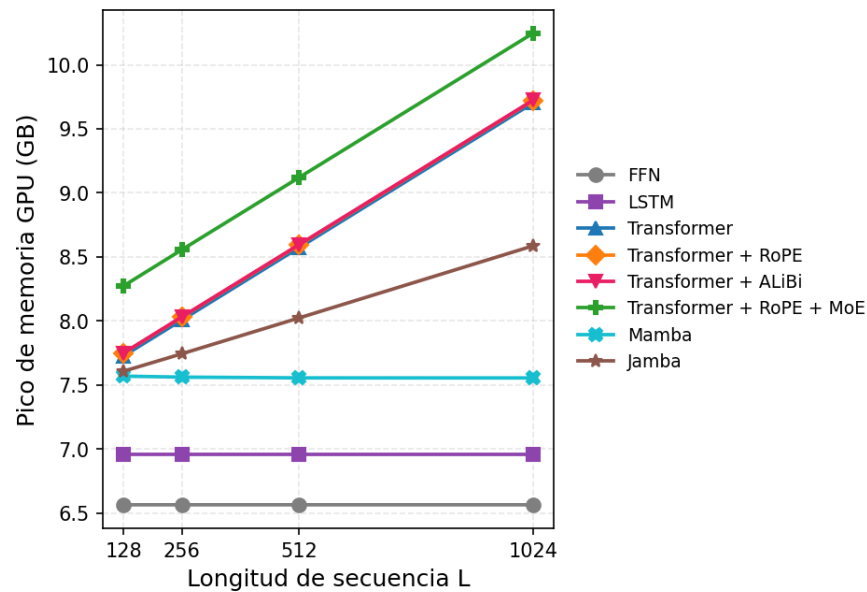
Como referencia, cada GPU A10G dispone de 24 GB de memoria, lo que supone un máximo agregado de $8 \times 24 = 192$ GB en el clúster. Los valores de la Tabla 7.5 corresponden al pico de

memoria *por GPU*: incluso en la configuración más exigente (Transformer + RoPE + MoE a $L=1024$) el pico se sitúa en torno a 10,5 GB de los 24 GB disponibles. La memoria no fue, por tanto, un factor limitante en ninguno de los experimentos; el interés de este dato es caracterizar el patrón de crecimiento del consumo con la longitud de secuencia de cada arquitectura.

La Figura 7.3 muestra la evolución de ambas métricas, donde se aprecia la dicotomía fundamental entre modelos con y sin atención cuadrática.



(a) Throughput de entrenamiento.



(b) Pico de memoria GPU.

Figura 7.3: Métricas de eficiencia computacional en función de la longitud de secuencia en WikiText-103.

Los perfiles de eficiencia se agrupan en tres categorías nítidas: En primer lugar, los modelos **sin atención cuadrática** —FFN y Mamba— mantienen un *throughput* y un consumo de memoria esencialmente constantes con la longitud de secuencia. FFN es el modelo más rápido (~ 860 Ktok/s), como cabría esperar de una arquitectura sin mecanismo secuencial alguno. Mamba alcanza ~ 690 Ktok/s independientemente de L , confirmando empíricamente su complejidad lineal: al no materializar ninguna matriz de atención, el coste por token es invariante a la longitud del contexto. El LSTM, a pesar de su complejidad teórica lineal, pierde un 36,1 % de *throughput* al pasar de $L = 128$ a $L = 1024$ (de 657 a 420 Ktok/s) debido a su naturaleza intrínsecamente secuencial, que limita el paralelismo durante el entrenamiento con secuencias largas.

En segundo lugar, los **Transformers puros** (Transformer, RoPE, ALiBi) presentan un crecimiento uniforme de memoria y una caída de *throughput* del 27–30 % al pasar de $L = 128$ a $L = 1024$. Todos los Transformers utilizan bloques de atención manual que materializan explícitamente la matriz de puntuaciones $L \times L$, lo que produce un crecimiento de memoria de ~ 7.9 GB a ~ 10 GB (+25 %) ². El modelo con MoE es consistentemente el más lento (~ 382 – 495 Ktok/s), penalizado por el routing y la evaluación secuencial de los expertos seleccionados en cada token ³.

En tercer lugar, **Jamba** se sitúa en un punto intermedio al alternar capas Mamba con capas de atención. Su *throughput* cae un 15,6 % de $L = 128$ a $L = 1024$ (de 660 a 557 Ktok/s), aproximadamente la mitad de la caída de los Transformers puros, y su pico de memoria crece de 7.8 GB a 8.8 GB (+12,9 %), también significativamente menos que los Transformers puros (+25 %). Ambos resultados son coherentes con el hecho de que solo la mitad de sus capas incurren en coste cuadrático.

7.2.4. Síntesis: relación calidad-eficiencia

La Figura 7.4 sintetiza la relación entre calidad de modelado y eficiencia computacional a $L = 1024$, la longitud más exigente de nuestros experimentos.

²Nótese que una implementación basada en kernels de atención fusionados (FlashAttention [7]) evitaría este crecimiento cuadrático de memoria al calcular la atención por bloques sin materializar la matriz completa. Hemos optado por implementaciones manuales para garantizar un control homogéneo sobre el cómputo de atención en las tres variantes posicionales, eliminando la posibilidad de que diferencias de rendimiento se deban a optimizaciones de bajo nivel asimétricas entre modelos.

³La implementación utilizada itera secuencialmente sobre los expertos activos. En sistemas de producción, kernels de tipo *grouped GEMM* [13] paralelizarían esta operación, reduciendo significativamente la penalización.

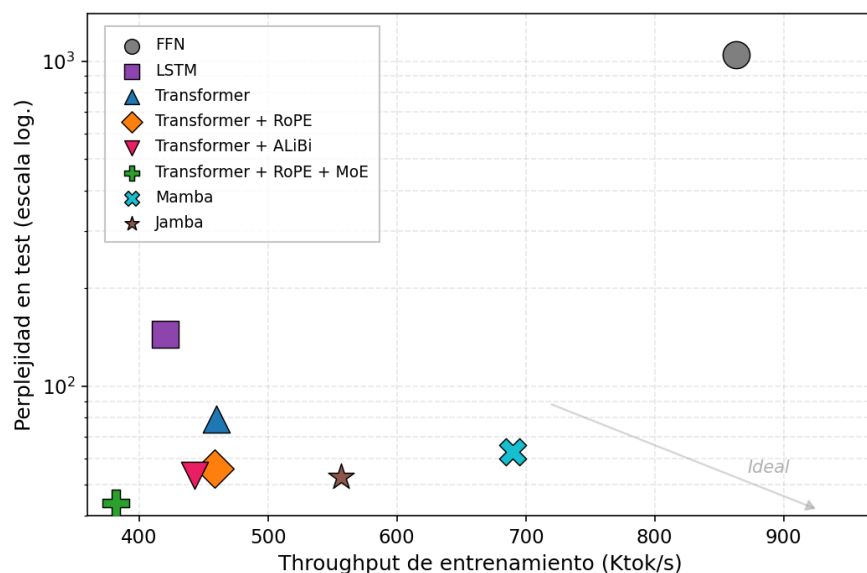


Figura 7.4: Perplejidad en test frente a *throughput* de entrenamiento a $L = 1024$. La esquina inferior derecha (baja perplejidad, alto *throughput*) representa el punto ideal.

Mamba ocupa el extremo de máxima eficiencia (690 Ktok/s) pero con una perplejidad que se sitúa por debajo de su potencial teórico (63.0), limitada por la saturación del estado latente. En el extremo opuesto, el modelo con MoE alcanza la mejor perplejidad (43.7) pero al menor *throughput* (382 Ktok/s). Jamba emerge como el modelo con mejor equilibrio global: logra la segunda mejor perplejidad (52.8) con un *throughput* de 557 Ktok/s, posicionándose claramente en la zona ideal del gráfico. Los Transformers con RoPE y ALiBi se agrupan en una zona de perplejidad similar (~54–56) pero con *throughput* inferior a Jamba (~443–459 Ktok/s), penalizados por tener todas sus capas basadas en atención.

En conjunto, los resultados de WikiText-103 establecen tres conclusiones preliminares. Primera, la codificación posicional es determinante: el Transformer con embeddings aprendidos se degrada al aumentar la longitud, mientras que RoPE y ALiBi mejoran de forma consistente hasta $L = 512$ y ALiBi continúa mejorando hasta $L = 1024$. Segunda, la hibridación SSM-atención (Jamba) ofrece el mejor compromiso entre calidad de modelado y eficiencia computacional, superando tanto a Mamba puro como a los Transformers puros con codificación relativa. Tercera, la corrección de la implementación de Mamba —añadiendo conexiones residuales y normalización pre-bloque— revela que este modelo es sustancialmente más capaz de lo que una implementación incompleta sugería, aunque su estado latente de dimensión fija sigue imponiendo un techo a la mejora con secuencias largas.

7.3. LRA ListOps: razonamiento estructural

La segunda tarea evalúa la capacidad de razonamiento composicional jerárquico mediante LRA ListOps [37]. Cada instancia es una expresión anidada construida con operadores sobre listas de enteros del 0 al 9 —MAX, MIN, MEDIAN y SUM-MODULO-10— y la etiqueta es el resultado entero (10 clases) de evaluar la expresión completa. Un ejemplo de longitud reducida ilustra la estructura: [MAX 3 [MIN 4 2 5] [SUM 1 7]] se evalúa a $\max(3, 2, 8) = 8$. A diferencia de WikiText-103, donde el éxito depende de capturar coherencia estadística local, ListOps exige resolver simultáneamente el emparejamiento de corchetes arbitrariamente anidados y la propagación de resultados parciales a través de la jerarquía. Es por tanto una prueba directa de *recuperación basada en contenido* y de paralelismo estructural a larga distancia.

7.3.1. Configuración experimental

Se utiliza la versión estándar de ListOps con división canónica train/validation/test. A diferencia de WikiText-103, ListOps no requiere un vocabulario de lenguaje natural: el conjunto de símbolos se reduce a los operadores (MAX, MIN, MEDIAN y SUM-MODULO-10), los diez dígitos, los delimitadores de paréntesis y los tokens especiales de relleno y desconocido. Empleamos por tanto un **vocabulario reducido** específico de la tarea (del orden de una quincena de tokens), compartido junto con la capa de embedding entre todas las arquitecturas. Este vocabulario compacto explica que los modelos de ListOps tengan muchos menos parámetros que sus equivalentes de WikiText-103 —por ejemplo, $\sim 3.2\text{M}$ frente a $\sim 29\text{M}$ en el Transformer denso—, ya que la matriz de embedding deja de dominar el recuento total. Cada instancia se trunca o se rellena hasta una longitud fija L , y entrenamos cada modelo con tres longitudes: $L \in \{512, 1024, 2048\}$. Para los modelos causales (Transformer y variantes, Mamba, Jamba) se utiliza *last-token pooling*: el modelo procesa la secuencia entera de forma autorregresiva y la predicción se obtiene del estado oculto en la última posición no-padding, de modo que la información debe fluir desde cualquier punto del árbol sintáctico hasta el final de la secuencia. Para FFN, que no posee mecanismo secuencial, se utiliza *mean pooling*.

Como en WikiText-103, fijamos el presupuesto de tokens por GPU y por paso de optimización en 8192 y ajustamos el tamaño de lote inversamente a L , lo que mantiene el número total de tokens vistos durante el entrenamiento aproximadamente constante entre longitudes y aísla el efecto del contexto disponible. El resto del régimen de entrenamiento —optimizador AdamW con $\text{lr} = 3 \times 10^{-4}$, weight decay 0,01, gradient clipping 1,0, AMP desactivada— es idéntico al de la sección anterior. La métrica principal es la *accuracy* en test; la referencia de azar de la tarea no es el 10 % de una distribución uniforme de 10 clases, sino el $\sim 17\%$ que alcanza un clasificador que siempre predice la clase mayoritaria, ya que las etiquetas de ListOps están desbalanceadas (Apéndice A.5).

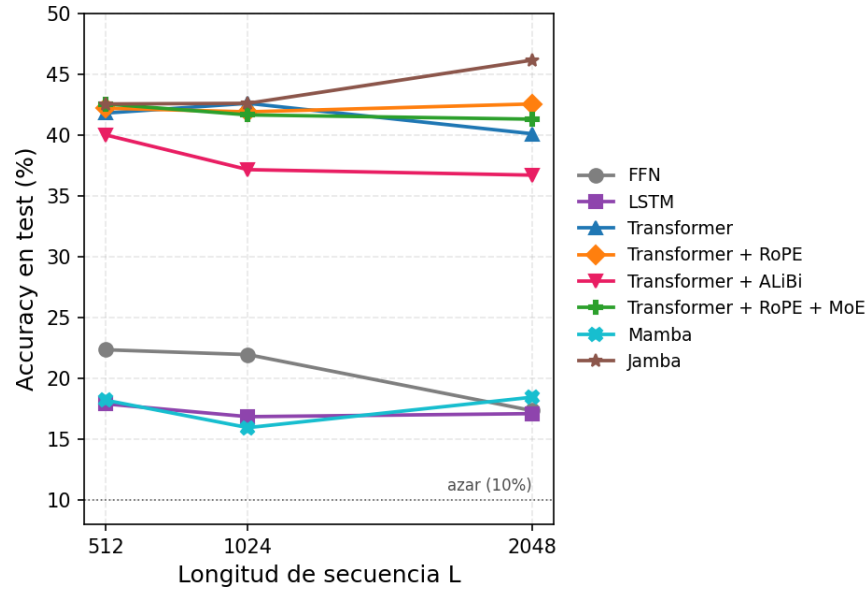
7.3.2. Resultados de *accuracy*

La Tabla 7.6 presenta la *accuracy* en test para los ocho modelos en las tres longitudes evaluadas. La columna Δ indica el cambio absoluto (en puntos porcentuales) al pasar de $L = 512$ a $L = 2048$: valores positivos indican que el modelo aprovecha el contexto adicional; valores negativos, degradación.

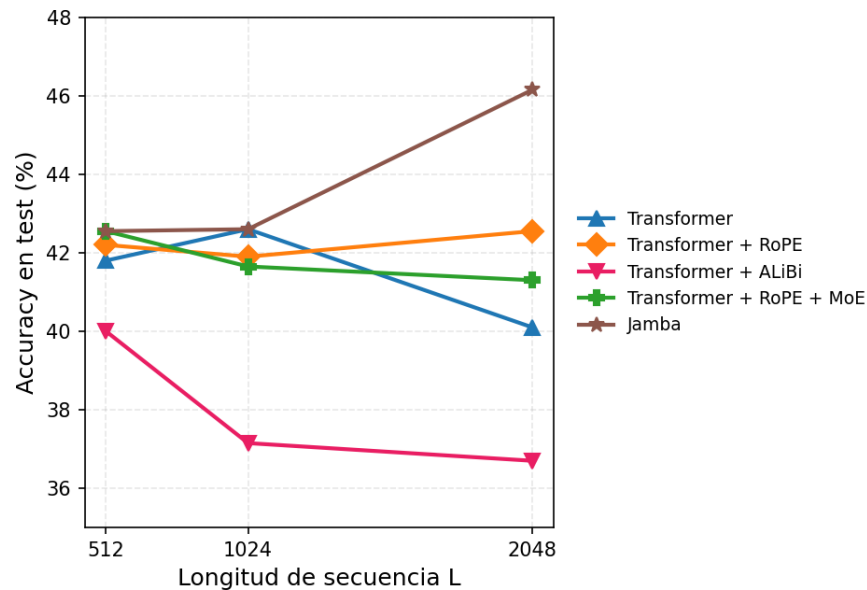
Modelo	$L=512$	$L=1024$	$L=2048$	$\Delta_{512 \rightarrow 2048}$
FFN	22.35	21.95	17.35	−5,00
LSTM	17.90	16.85	17.10	−0,80
Transformer	41.80	42.60	40.10	−1,70
Transformer + RoPE	42.20	41.90	42.55	+0,35
Transformer + ALiBi	40.00	37.15	36.70	−3,30
Transformer + RoPE + MoE	42.55	41.65	41.30	−1,25
Mamba	18.20	15.95	18.45	+0,25
Jamba	42.55	42.60	46.15	+3,60

Tabla 7.6: Accuracy en test (%) en LRA ListOps para las tres longitudes evaluadas. $\Delta_{512 \rightarrow 2048}$ es el cambio absoluto en puntos porcentuales. La línea de azar (clase mayoritaria) es $\sim 17\%$.

La Figura 7.5 visualiza estos resultados. La vista panorámica (a) revela una separación neta entre dos grupos: por un lado FFN, LSTM y Mamba, todos confinados en la franja 16–22 %, escasamente por encima del azar; por otro, todos los modelos con atención más Jamba, agrupados entre el 37 % y el 46 %. La vista de detalle (b) amplía este segundo grupo, donde se aprecia con claridad que solo Jamba mejora de forma sostenida al aumentar L , mientras que el resto de modelos o se mantienen planos (RoPE) o se degradan (Transformer, MoE y, sobre todo, ALiBi).



(a) Vista panorámica.



(b) Detalle del grupo competitivo.

Figura 7.5: Accuracy en test en función de la longitud de secuencia en LRA ListOps.

La Figura 7.6 resume el escalado con la longitud mediante el cambio absoluto de *accuracy* entre $L = 512$ y $L = 2048$.

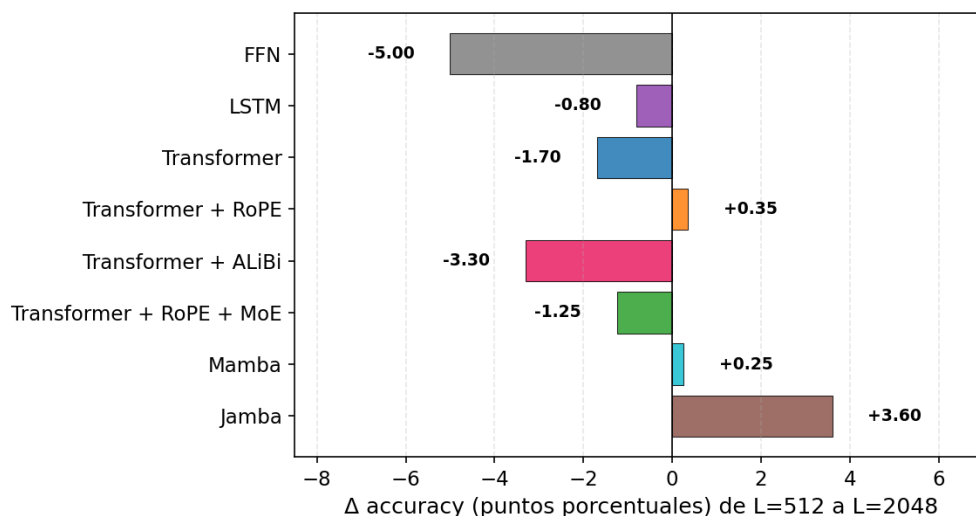


Figura 7.6: Cambio absoluto de *accuracy* (en puntos porcentuales) al pasar de $L = 512$ a $L = 2048$. Solo Jamba muestra una mejora claramente positiva; el resto permanece plano o se degrada.

Los resultados admiten una lectura en tres bloques.

Modelos que fracasan en la tarea. La FFN, el LSTM y Mamba se quedan en accuracies del orden del 16–22 % en las tres longitudes, manteniéndose cerca de la línea de azar de la tarea (~ 17 %, la clase mayoritaria) y es compatible con que estos modelos estén capturando solo el sesgo marginal de clases sin aprender la operación composicional. La *accuracy* de entrenamiento se mantiene en el mismo rango que la de test (FFN: 21–23 %; LSTM: 18–21 %; Mamba: 19–21 %), lo que descarta sobreajuste e indica una limitación de *capacidad inductiva*, no de capacidad de memorización.

La FFN fracasa por razones triviales: al promediar los embeddings antes de la cabeza de clasificación, destruye el orden y la estructura de paréntesis sin los cuales la expresión no puede evaluarse. El LSTM tampoco aprende la tarea, lo que es coherente con los resultados originales del benchmark LRA [37], donde las arquitecturas recurrentes obtenían accuracies cercanas al azar: la propagación puramente secuencial del estado oculto dificulta el seguimiento de niveles de anidamiento arbitrarios en una sola pasada.

El caso de Mamba es el resultado más informativo de toda la sección. A pesar de mostrar una complejidad comparable a la de los Transformers en WikiText-103 (63–67 frente a 54–80), Mamba colapsa en ListOps. La explicación está en la naturaleza del mecanismo selectivo descrito en la Sección 6.6: el filtro de selección de Mamba modula *cuánto* de la entrada se incorpora al estado en cada paso, pero el estado mismo es un vector comprimido de dimensión fija ($d_{\text{state}} = 16$) que se actualiza de forma estrictamente secuencial. No existe un mecanismo de recuperación por contenido que permita, en la posición del corchete de cierre, “ir a buscar” el corchete de apertura correspondiente para recuperar el resultado parcial almacenado allí. Trabajos recientes sobre las limitaciones de los SSMs [21, 2] demuestran formalmente que tareas de *copia* y *recall* asociativo basadas en contenido son intrínsecamente difíciles para SSMs de estado fijo, precisamente porque el ancho de banda del estado pone un techo al número de pares clave-valor recuperables. ListOps cae de lleno en esta categoría: para evaluar correctamente `[MAX 3 [MIN ...] [SUM ...]]` hace falta recuperar los resultados de los subárboles en el momento exacto en que se cierra el corchete que los engloba, una operación que la atención resuelve trivialmente y los SSMs no.

Modelos que resuelven la tarea pero no escalan. Los cuatro Transformers puros y el Transformer+MoE alcanzan accuracies entre el 37 % y el 42,5 % y, en contraste con la mejora monótona que muchos de ellos exhibían en WikiText-103, en ListOps muestran un comportamiento plano o ligeramente decreciente al aumentar L . El Transformer con embeddings aprendidos pasa de 41,80 % a 40,10 % (−1,7 pp); RoPE se mantiene esencialmente constante (+0,35 pp); MoE pierde 1,25 pp. La ausencia de mejora con el contexto adicional es coherente con la naturaleza de la tarea: una expresión ListOps tiene una longitud efectiva fija independiente del *padding*; aumentar L no añade información nueva, solo más tokens de relleno que el modelo debe aprender a ignorar. En este escenario, los Transformers con codificación posicional adecuada simplemente no se benefician de un contexto mayor.

El caso del MoE merece comentario separado. Con 9,5M de parámetros totales (frente a 3,2M de los Transformers densos), el modelo dispone de aproximadamente tres veces más capacidad y aun así no supera al Transformer+RoPE en ninguna de las tres longitudes (41,30 % frente a 42,55 % a $L = 2048$). Esto sugiere que en ListOps la capacidad por parámetro no es el factor limitante: el cuello de botella es la *precisión composicional* con la que el modelo encadena operaciones de recuperación basadas en contenido, una propiedad estructural del mecanismo de atención que añadir expertos no mejora.

ALiBi: cuando la inducción se vuelve perjudicial. Transformer+ALiBi es el único modelo del grupo competitivo que se degrada de forma sostenida con la longitud, pasando de 40,00 % a 36,70 % (−3,3 pp) y siendo claramente el peor de los Transformers a $L = 2048$. Este resultado contrasta de forma muy marcada con WikiText-103, donde ALiBi era el único Transformer puro con mejora estrictamente monótona en todo el rango y superaba a RoPE a $L = 1024$. La explicación reside precisamente en lo que constituye la fortaleza de ALiBi en modelado de lenguaje: el sesgo lineal decreciente con la distancia, que codifica una preferencia por el contexto local. En lenguaje natural esta preferencia es una buena *prior*, porque las dependencias relevantes para predecir el siguiente token son mayoritariamente locales o medias. En ListOps, en cambio, el corchete de apertura que un MAX debe recuperar puede encontrarse a cientos de tokens de distancia, y la penalización lineal que aplica ALiBi atenúa precisamente las puntuaciones de atención que el modelo necesitaría amplificar. La situación se agrava con L porque las distancias máximas implicadas crecen con la longitud, y por tanto la fracción de la masa atencional que el modelo *a priori* suprime es mayor. Esta observación es una instancia local de un principio más general que el resto del capítulo confirma transversalmente: las codificaciones posicionales con sesgos inductivos fuertes ayudan cuando coinciden con la estructura de dependencias del problema y perjudican cuando no lo hacen.

Jamba: el único modelo que escala. Jamba es el único modelo cuya *accuracy* mejora claramente con la longitud, ganando +3,6 pp entre $L = 512$ y $L = 2048$ y alcanzando el mejor resultado absoluto de la tabla (46,15 % a $L = 2048$, 3,6 pp por encima del segundo clasificado, Transformer+RoPE). Lo notable es que este modelo logra mejorar con el contexto en una tarea donde ni siquiera los Transformers puros lo consiguen, y lo hace con solo 2,5M de parámetros, 25 % menos que un Transformer denso. La interpretación es coherente con la hipótesis arquitectónica que motiva la hibridación: las capas Mamba aportan un procesamiento secuencial eficiente que comprime de forma adaptativa el contexto irrelevante (el *padding* y los tokens estructurales sin contenido numérico), mientras que las dos capas de atención intercaladas (patrón M–A–M–A) ejecutan, a partir de esa representación pre-procesada, las operaciones de recuperación basadas en contenido que la tarea exige. Ninguno de los dos mecanismos por separado es suficiente —Mamba puro colapsa, el Transformer puro no escala— pero la combinación los complementa.

7.3.3. Eficiencia computacional

El perfil de eficiencia en ListOps reproduce las tres categorías ya identificadas en WikiText-103, con magnitudes proporcionalmente mayores por trabajar con secuencias hasta el doble de largas. La Tabla 7.7 recoge el *throughput* y el tiempo total de entrenamiento, y la Tabla 7.8 el pico de memoria por GPU.

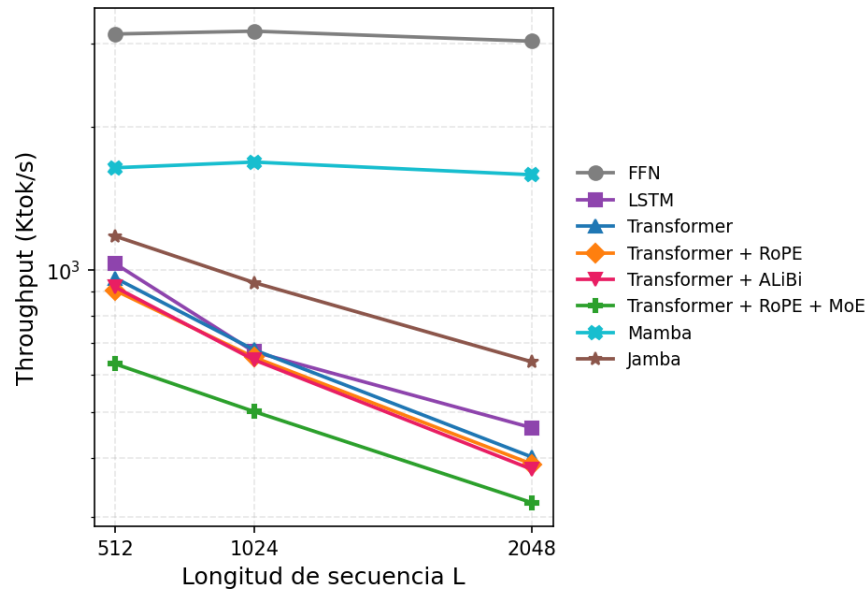
Modelo	Tiempo de entrenamiento (s)			Throughput (Ktok/s)		
	$L=512$	$L=2048$	Δ	$L=512$	$L=2048$	Δ
FFN	166	647	+289 %	3147	3038	-3,5 %
LSTM	487	4277	+778 %	1029	463	-55,0 %
Transformer	526	4814	+815 %	960	402	-58,2 %
Transformer + RoPE	549	4939	+800 %	904	388	-57,1 %
Transformer + ALiBi	544	5118	+841 %	919	378	-58,9 %
Transformer + RoPE + MoE	796	5911	+643 %	632	322	-49,1 %
Mamba	301	1183	+293 %	1642	1586	-3,4 %
Jamba	428	3054	+613 %	1177	638	-45,8 %

Tabla 7.7: Tiempo total de entrenamiento y *throughput* para $L = 512$ y $L = 2048$ en LRA ListOps. Δ es el cambio relativo.

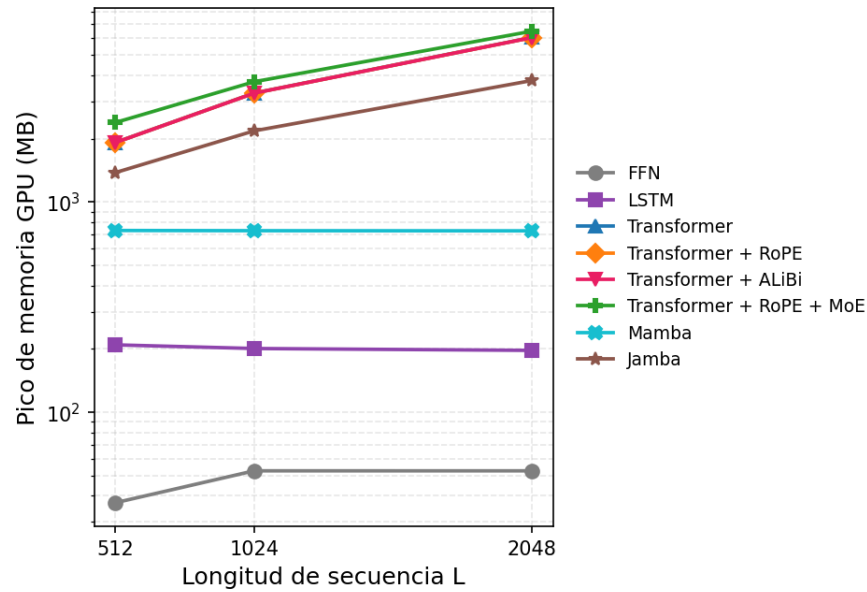
Modelo	Pico de memoria GPU (MB)		
	$L=512$	$L=1024$	$L=2048$
FFN	37	53	53
LSTM	209	201	197
Transformer	1916	3296	6065
Transformer + RoPE	1914	3293	6058
Transformer + ALiBi	1914	3292	6057
Transformer + RoPE + MoE	2384	3730	6496
Mamba	731	730	729
Jamba	1378	2179	3784

Tabla 7.8: Pico de memoria GPU (MB por GPU) en función de la longitud de secuencia en LRA ListOps.

La Figura 7.7 muestra ambas métricas en escala logarítmica.



(a) Throughput de entrenamiento.



(b) Pico de memoria GPU.

Figura 7.7: Métricas de eficiencia computacional en función de la longitud de secuencia en LRA ListOps (ambos ejes verticales en escala logarítmica).

Los **modelos sin atención cuadrática** (FFN y Mamba) mantienen un *throughput* y un consumo de memoria prácticamente constantes con L , confirmando una vez más su complejidad lineal. Mamba sostiene ~ 1.6 Mtok/s en las tres longitudes y un pico de memoria estable en torno a 730 MB; FFN duplica ese *throughput* y consume el orden de 50 MB. El LSTM, pese a su complejidad teórica lineal, pierde un 55% de *throughput* al pasar de $L = 512$ a $L = 2048$ debido a su naturaleza intrínsecamente secuencial, que limita el paralelismo durante el entrenamiento.

Los **Transformers puros** muestran una caída de *throughput* del 57–59% y un crecimiento de memoria de 1,9 GB a 6,1 GB (+217%, cuadrático en L como predice la teoría) al pasar de $L = 512$ a $L = 2048$. La penalización es significativamente mayor que en WikiText-103 porque las longitudes son hasta el doble. El MoE añade el coste constante del routing y la evaluación secuencial de expertos sobre todo lo anterior, resultando consistentemente el modelo más lento del grupo (322 Ktok/s a $L = 2048$).

Jamba ocupa de nuevo un punto intermedio: su *throughput* cae un 45,8% (de 1177 a 638 Ktok/s) y su pico de memoria crece de 1,4 a 3,8 GB (+175%). En las dos métricas, Jamba es aproximadamente la mitad de costoso que un Transformer puro y, como hemos visto, alcanza una *accuracy* claramente superior.

7.3.4. Síntesis: relación calidad-eficiencia

La Figura 7.8 sitúa cada modelo en el plano *accuracy* vs. *throughput* a $L = 2048$, la longitud más exigente de la tarea.

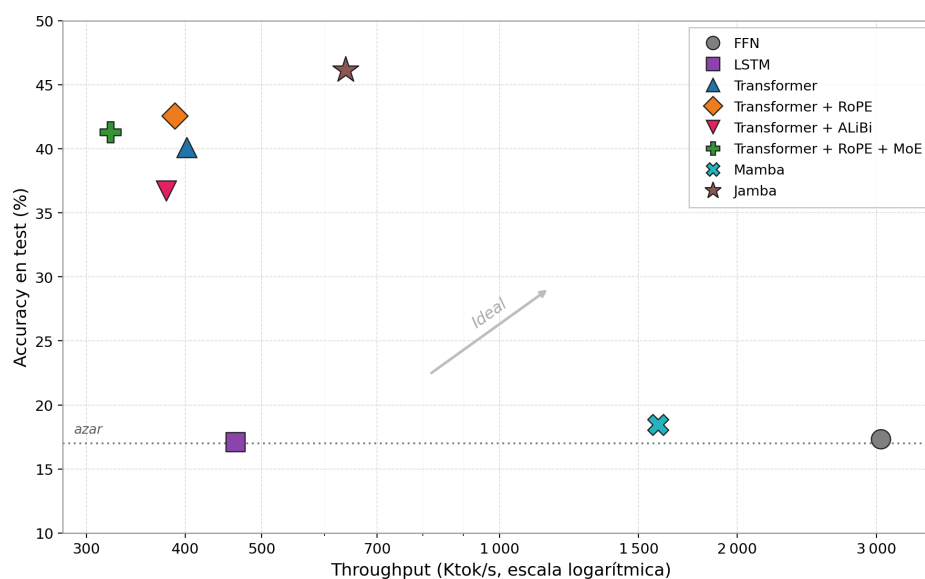


Figura 7.8: Accuracy en test frente a *throughput* de entrenamiento a $L = 2048$ en LRA ListOps. La esquina superior derecha (alta *accuracy*, alto *throughput*) es el punto ideal.

El gráfico hace evidente que Jamba ocupa por sí solo la esquina superior derecha del plano: alcanza la *accuracy* máxima (46,15%) con un *throughput* de 638 Ktok/s, casi el doble que cualquier Transformer puro. El grupo Transformer/RoPE/MoE/ALiBi se agrupa en una región de *accuracy* intermedia (37–43%) y *throughput* bajo (~320–400 Ktok/s). En el cuadrante inferior derecho, los modelos lineales (Mamba, FFN) ofrecen *throughput* muy alto pero *accuracy* cercana al azar, lo que en esta tarea los convierte en líneas base que cuantifican el coste de carecer de atención.

En conjunto, ListOps deja tres conclusiones que complementan las de WikiText-103. La primera es que, cuando la tarea obliga a localizar información por su contenido, la atención deja de ser opcional: los modelos que carecen de ella (FFN, LSTM, Mamba) fracasan al margen de lo eficientes que sean, de cuántos parámetros tengan o de la longitud de contexto que manejen. La segunda matiza el papel de los sesgos posicionales, que no son buenos por sistema: ALiBi, el mejor Transformer a contexto largo en WikiText-103, cae aquí al último puesto justo por el sesgo de localidad que en lenguaje natural lo favorecería. Y la tercera vuelve a situar a Jamba como la opción más equilibrada: es el único que mejora con la longitud en ListOps, el único que junta a la vez bajo coste y alta calidad, y lo consigue con menos parámetros que los Transformers densos.

7.4. Discusión de resultados

Las dos tareas evaluadas iluminan facetas distintas y en parte complementarias de las arquitecturas estudiadas. WikiText-103 prima la capacidad de explotar estadísticas distribucionales a múltiples escalas; LRA ListOps prima la capacidad de ejecutar operaciones de recuperación composicional entre tokens distantes. Que un modelo destaque en una de ellas no implica que destaque en la otra, y los puntos de divergencia entre ambos benchmarks son tanto o más informativos que las coincidencias.

Lo que ambas tareas confirman. Hay tres cosas que se mantienen al pasar de una tarea a otra. La **FFN** hace de suelo en las dos, lo que descarta que cualquier resultado posterior venga de un artefacto del protocolo. El **LSTM** se queda claramente por detrás de los modelos con atención en ambas, en línea con la transición histórica de las RNNs al Transformer [39]. Y **Jamba** vuelve a ser el modelo más equilibrado entre calidad y coste: segunda mejor perplejidad en WikiText-103 a $L = 1024$ y mejor *accuracy* en ListOps a $L = 2048$, en los dos casos con más *throughput* que los Transformers puros y con menos parámetros que casi todos ellos.

Lo que cada tarea reordena. Las divergencias entre tareas son las que mejor delimitan las fortalezas y debilidades de cada arquitectura. La Tabla 7.9 resume el ranking de los cinco modelos competitivos en cada tarea a la longitud máxima evaluada.

WikiText-103 ($L = 1024$)			LRA ListOps ($L = 2048$)		
Pos.	Modelo	PPL ↓	Pos.	Modelo	Acc. (%) ↑
1	RoPE + MoE	43.7	1	Jamba	46.15
2	Jamba	52.8	2	Transformer + RoPE	42.55
3	ALiBi	53.4	3	RoPE + MoE	41.30
4	RoPE	55.8	4	Transformer	40.10
5	Transformer	79.4	5	ALiBi	36.70

Tabla 7.9: Ranking de los cinco modelos competitivos a la longitud máxima evaluada en cada tarea. Nótese el descenso de ALiBi de la tercera a la última posición y el ascenso de Jamba al primer puesto.

Sobresalen dos cambios de orden. El primero es que **ALiBi cae de la tercera a la última posición**. Su sesgo lineal de localidad, que en lenguaje natural codifica una *prior* útil sobre dónde suelen estar las dependencias relevantes, se vuelve un lastre en cuanto la tarea exige mirar a posiciones lejanas y arbitrarias. Es, a nuestro juicio, lo que más dice el estudio sobre el papel de la posición: ninguna codificación es mejor o peor en abstracto; lo es según encaje con la estructura de dependencias que el problema impone. RoPE, que codifica relaciones angulares sin privilegiar la cercanía, aguanta mejor el cambio de tarea; ALiBi solo rinde cuando su prejuicio de localidad acierta con lo que el problema de verdad necesita.

El segundo es que el **Transformer + MoE pierde su ventaja** al pasar de WikiText a ListOps. Con 9,5M de parámetros totales frente a los 3,2M de los Transformers densos, cabría esperar que rindiese más en las dos tareas. Y así ocurre en WikiText-103 (43,7 frente a 55,8 PPL de RoPE, un $-21,6\%$), pero en ListOps la ventaja se evapora (41,30% frente al 42,55% de RoPE, incluso algo *por debajo*). Encaja con la idea de que MoE da más capacidad por token pero no toca la capacidad

estructural de la atención: los expertos de más rinden mientras el problema sea estadístico (lenguaje) y no aportan nada cuando es composicional (razonamiento jerárquico).

Mamba y el coste de no tener atención. La diferencia más brusca entre tareas es la de Mamba. En WikiText-103 llega a una perplejidad de 63,0 a $L = 1024$, comparable a la de los Transformers puros, porque el modelado de lenguaje no pide recuperar pares clave-valor lejanos de forma exacta: basta con aproximar la distribución del siguiente token. En ListOps, en cambio, se desploma hasta el 18,45 %, apenas por encima del azar. Detrás de las dos cosas está la misma propiedad: el estado comprimido de dimensión fija ($d_{\text{state}} = 16$, visto en la Sección 6.6.5) funciona como una memoria asociativa de muy poco ancho de banda. Sobra para retener la estadística reciente de un texto y falta para guardar los resultados parciales de una expresión anidada y profunda. Es, sobre nuestra propia infraestructura, una comprobación directa de los límites de acceso por contenido que la literatura ha probado de forma formal [21, 2].

Por qué Jamba gana en ambas tareas. La hibridación M–A–M–A combina las dos propiedades que las tareas demandan por separado: capas Mamba que comprimen el contexto de forma eficiente para acelerar el cómputo y atenuar el ruido posicional, y capas de atención intercaladas que dotan al modelo del mecanismo de recuperación basado en contenido del que Mamba puro carece. En WikiText-103 esa mezcla da la segunda mejor perplejidad con menos coste que cualquier Transformer puro. En ListOps, donde el acceso por contenido es imprescindible, pasa directamente a ser la mejor *accuracy* del estudio, y encima la única que crece con el contexto. Visto así, elegir entre SSM puro, Transformer puro o híbrido no es decidir entre eficiencia y calidad —Jamba gana en las dos—, sino anticipar qué pide el problema: en cuanto haya el menor indicio de recuperación por contenido, las capas de atención dejan de ser prescindibles.

Limitaciones de la evaluación. Conviene leer todo lo anterior con dos cautelas, válidas para las dos tareas. La primera es que cada configuración se ha entrenado con una sola semilla, así que las diferencias pequeñas dentro del grupo competitivo hay que tomarlas con pinzas; lo que sostenemos se apoya en tendencias que se repiten a lo largo de las longitudes evaluadas, no en comparaciones sueltas entre modelos parecidos. La segunda es la escala, que es moderada ($\sim 3\text{--}35$ M parámetros, $L \leq 2048$): algunos de estos resultados —en particular la ventaja del patrón M–A–M–A sobre Mamba puro— podrían no trasladarse tal cual a varios miles de millones de parámetros y contextos de $\geq 100\text{k}$ tokens, donde entran en juego cosas como el coste de la KV-cache en inferencia o la proporción óptima de capas [24, 40].

Capítulo 8

Conclusiones

Este trabajo se planteó como un estudio comparativo de las principales familias de arquitecturas de aprendizaje profundo para procesamiento del lenguaje natural, con énfasis en cómo la longitud de secuencia condiciona el rendimiento y la eficiencia de cada una de ellas. El estudio combina una revisión teórica —redes de propagación directa, recurrentes, Transformer y sus variantes modernas, modelos de espacio de estados y arquitecturas híbridas— con una evaluación empírica controlada de ocho arquitecturas en dos tareas complementarias: modelado de lenguaje en WikiText-103 y razonamiento estructural en LRA ListOps. En este capítulo recogemos los hallazgos principales, identificamos las contribuciones del trabajo, discutimos sus limitaciones y proponemos líneas para extenderlo.

8.1. Hallazgos principales

De la evaluación experimental extraemos cinco hallazgos, que recogemos a continuación y que tocan algunas de las preguntas que hoy vertebran el modelado de secuencias.

La codificación posicional pesa, y depende de la tarea. El Transformer con embeddings aprendidos empeora al alargar la secuencia en WikiText-103 (+19,3 % de perplejidad de $L = 128$ a $L = 1024$), mientras que RoPE y ALiBi consiguen lo contrario. Hasta ahí, nada que no diga ya la literatura. La sorpresa llega al cruzar las dos tareas: ALiBi, el mejor Transformer a contexto largo en WikiText-103, es el peor en ListOps, donde su sesgo de localidad rebaja justo las puntuaciones de atención hacia las posiciones lejanas que el emparejamiento de corchetes necesitaría reforzar. La moraleja es simple: una codificación posicional no vale más ni menos por sí misma, sino según su *prior* encaje con las dependencias reales del problema. RoPE, que mide relaciones angulares sin una dependencia directa de la cercanía, es la que mejor aguanta el salto de una tarea a otra.

Sin atención no se resuelve lo que exige buscar por contenido. El contraste más claro de todo el trabajo es el de Mamba consigo mismo: a la altura de los Transformers en WikiText-103 (perplejidad 63,0 a $L = 1024$, dentro del rango 54–80 de los Transformers puros) y hundido en ListOps (*accuracy* 18,45 % a $L = 2048$, casi el azar). La razón es estructural. El mecanismo selectivo de Mamba decide cuánto de cada entrada entra en el estado, pero no tiene forma de *volver* a por algo que guardó pasos atrás. En lenguaje, donde basta con aproximar la distribución del siguiente token, eso casi no se nota; en razonamiento composicional, donde hay que recobrar resultados parciales exactos metidos en cualquier nivel de anidamiento, es letal. El estado de dimensión fija $d_{\text{state}} = 16$ que analizábamos en el Capítulo 6 deja de ser una abstracción y se convierte en el estrangulamiento que explica el colapso observado.

Más capacidad no supe al mecanismo que falta. El Transformer con MoE añade $\sim 6.3\text{M}$ de parámetros sobre el denso con RoPE en forma de expertos. En WikiText-103, donde ese extra sube el recuento total de 28,9M a 35,2M (+22 %), la perplejidad mejora un 21,6 % a $L=1024$ (43,7 frente a 55,8). En ListOps, donde el vocabulario reducido hace que ese mismo extra triplique el recuento ($3,2\text{M} \rightarrow 9,5\text{M}$ totales, 5,3M activos), la *accuracy* se queda igual (41,30 % frente al 42,55 % del RoPE denso, incluso algo *por debajo*). Los parámetros de más rinden mientras al modelo le falte haber visto datos, pero no compran nada cuando lo que falta es el mecanismo. Aquí, el “más parámetros, mejor” de manual no se cumple.

La hibridación SSM–atención es la que mejor equilibra todo. Jamba —patrón M–A–M–A, 2,5M de parámetros— es el modelo que mejor concilia calidad y coste en las dos tareas: segunda mejor perplejidad en WikiText-103 a $L = 1024$ con un 20 % más de *throughput* que los Transformers puros, y la mejor *accuracy* absoluta en ListOps a $L = 2048$, además de ser el único que mejora con la longitud. Funciona porque las dos carencias que lo motivan se compensan: las capas Mamba aportan el procesamiento secuencial barato y descartan el contexto que no importa, y las capas de atención intercaladas ponen el acceso por contenido que a Mamba puro le falta. Por separado, ninguno de los dos basta; juntos, superan a ambos.

La teoría de la complejidad se cumple casi al pie de la letra. La medida empírica de la eficiencia casi calca lo que predice la teoría: FFN y Mamba mantienen *throughput* y memoria planos con L ; los Transformers puros enseñan el crecimiento cuadrático de memoria y la caída de *throughput* proporcional a L ; y Jamba se queda en el punto intermedio que cabe esperar de tener la mitad de las capas lineales. El único que se sale del guion es el LSTM: su complejidad lineal sobre el papel se traduce en una caída de *throughput* del 36–55 % con la longitud, porque su carácter secuencial impide paralelizar el entrenamiento.

8.2. Contribuciones

El trabajo deja cuatro contribuciones.

Comparación experimental homogénea de ocho arquitecturas. La literatura sobre arquitecturas eficientes para secuencias largas compara a menudo modelos entrenados bajo condiciones distintas: diferentes optimizadores, diferentes recursos computacionales, diferentes datos de preentrenamiento. En este trabajo todas las arquitecturas se entrenan desde cero con el mismo optimizador, el mismo número de capas, la misma dimensión del modelo, la misma disciplina de entrenamiento (sin precisión mixta, sin scheduler, sin warmup) y el mismo presupuesto efectivo de tokens por paso. Esto permite atribuir las diferencias observadas a las propiedades estructurales de cada arquitectura y no a un mejor ajuste de hiperparámetros particulares.

Identificación y corrección de problemas de implementación en modelos de referencia. Durante la fase de validación se detectaron tres asimetrías en las implementaciones iniciales que invalidaban la comparabilidad de los resultados: el Transformer basado en `nn.TransformerEncoder` despachaba implícitamente al kernel FlashAttention, mientras que las variantes con RoPE y ALiBi (implementadas manualmente) no lo hacían; el modelo ALiBi carecía de la normalización final `ln_f` antes de la cabeza de salida; y la implementación de Mamba omitía las conexiones residuales y la normalización pre-bloque, lo que producía una degradación activa con la longitud que no era arquitectónica sino de implementación. Las tres correcciones se documentan explícitamente en el capítulo experimental, y los resultados pre y post corrección de Mamba muestran la magnitud del efecto: una perplejidad de 81,8 a $L = 1024$ pasa a 63,0 tras añadir las residuales y la pre-normalización. Esta

observación tiene un valor metodológico más general: en comparaciones entre arquitecturas heterogéneas, la verificación de detalles de implementación frente a referencias canónicas es un paso previo no negociable antes de extraer conclusiones.

Evidencia empírica directa de las limitaciones de los SSMs en recuperación basada en contenido. El contraste entre Mamba en WikiText-103 (funciona) y Mamba en LRA ListOps (colapsa) constituye una validación empírica, sobre una infraestructura reproducible y a escala accesible, de las predicciones teóricas formales sobre la incapacidad de los SSMs de estado fijo para realizar recuperación asociativa exacta. Este resultado complementa las pruebas formales [21, 2] con una demostración en condiciones de entrenamiento realistas y aporta una intuición útil para profesionales que deban elegir arquitectura: si la tarea contiene cualquier componente de recuperación basada en contenido, prescindir de la atención no es una opción.

Cuantificación del compromiso calidad-eficiencia de la hibridación. Los datos recogidos permiten cuantificar de forma precisa la ventaja del patrón M–A–M–A de Jamba sobre las arquitecturas puras a la escala estudiada: con menor número de parámetros que cualquier Transformer puro, Jamba alcanza calidad de modelado superior en una de las tareas (LRA ListOps) y muy próxima al mejor en la otra (WikiText-103), con *throughput* intermedio y un pico de memoria entre un 12 y un 16 % inferior a $L = 1024$. Para tareas en las que ninguna de las dos dimensiones del compromiso —calidad o eficiencia— puede sacrificarse, la hibridación es preferible a ambos extremos.

8.3. Limitaciones del estudio

Todo lo anterior hay que leerlo teniendo presentes cuatro límites del diseño experimental.

Una sola semilla aleatoria por configuración. Por restricciones de cómputo, cada par (modelo, longitud) se ha entrenado una sola vez. Esto implica que las diferencias pequeñas entre modelos próximos del grupo competitivo —por ejemplo, los ~ 1.5 pp que separan a Transformer+RoPE y Transformer+MoE en LRA ListOps a $L = 2048$ — pueden encontrarse dentro del margen de variabilidad entre semillas. Las conclusiones que extraemos se apoyan en tendencias consistentes a lo largo de tres o cuatro longitudes evaluadas, no en comparaciones puntuales entre modelos cercanos. Los contrastes cualitativos del estudio —el colapso de Mamba en ListOps, la inversión de ALiBi entre tareas, la mejora de Jamba con la longitud— son de magnitud suficiente para ser robustos a esta variabilidad.

Escala moderada. Los modelos evaluados tienen entre 3 y 35 millones de parámetros y se entrenan con secuencias de hasta 2048 tokens. Algunos de los hallazgos a esta escala podrían no transferirse de forma directa a las escalas actuales del estado del arte ($\geq 10^{10}$ parámetros, contextos de 10^5 tokens o más), donde aparecen consideraciones adicionales que no se manifiestan a escala pequeña: el coste del KV-cache en inferencia, los regímenes específicos de aprendizaje a escala (emergencia de capacidades, leyes de escalado), o las decisiones sobre proporción óptima de capas atención/SSM en arquitecturas híbridas [24, 40]. En particular, la ventaja relativa de Jamba sobre Mamba puro observada en este trabajo puede sobreestimar o subestimar la diferencia que se manifestaría a varios miles de millones de parámetros.

Dos tareas, dos paradigmas. La selección de WikiText-103 y LRA ListOps cubre dos extremos cualitativos del espectro de tareas de modelado de secuencias —estadística distribucional y razonamiento composicional— pero no agota el espacio de problemas relevantes. Tareas como recuperación de información, traducción, resumen abstractivo o razonamiento matemático plantean exigencias intermedias o de naturaleza distinta que el estudio actual no caracteriza. En particular, el resto de subtareas del benchmark LRA [37] (Text, Retrieval, Image, Pathfinder, Path-X) probaría en qué medida los patrones identificados son específicos de ListOps o generales.

Entrenamiento desde cero. Todas las arquitecturas se entrenan sin preentrenamiento, lo que sitúa la comparación en un régimen experimentalmente limpio pero alejado de la práctica habitual. El preentrenamiento podría alterar el ranking relativo entre arquitecturas, ya que algunas familias se benefician más que otras del aprendizaje de representaciones generales antes de la tarea final.

8.4. Líneas de trabajo futuro

De esos límites salen, casi solas, cinco líneas de continuación, que ordenamos de menor a mayor coste.

Replicación con múltiples semillas. La extensión más inmediata y de menor coste es repetir cada experimento con tres a cinco semillas y reportar medias e intervalos de confianza. Esto permitiría establecer si las diferencias pequeñas dentro del grupo competitivo son estadísticamente significativas, y reforzaría las conclusiones cualitativas que el estudio ya extrae.

Expansión del benchmark LRA. Evaluar las ocho arquitecturas en las cinco sub tareas restantes de LRA (Text, Retrieval, Image, Pathfinder y, muy especialmente, Path-X, que prueba dependencias de hasta 16k tokens) permitiría comprobar si la inversión de ALiBi observada entre WikiText y ListOps se reproduce en otras combinaciones tarea-modelo, y caracterizar con más precisión qué propiedades estructurales de las tareas favorecen a cada arquitectura.

Exploración sistemática de variantes de hibridación. Jamba admite múltiples configuraciones del patrón de capas (MAMA, MAAA, MAAA, MMMA. . .) y de la proporción global atención/SSM. El estudio actual fija el patrón M-A-M-A por consistencia con la propuesta original [24], pero un barrido sistemático permitiría caracterizar el compromiso óptimo entre eficiencia y capacidad de recuperación. La intuición que se desprende del estudio actual sugiere que incrementar la proporción de capas de atención debería ayudar más en tareas composicionales y menos en modelado de lenguaje, pero esta hipótesis requiere verificación experimental.

Comparación con preentrenamiento. Reproducir el estudio preentrenando primero todos los modelos sobre un corpus general (por ejemplo C4 o The Pile) y ajustando después a cada tarea permitiría medir hasta qué punto las diferencias observadas dependen del régimen *from-scratch* y cuáles son robustas al cambio de protocolo. Algunos resultados sobre LRA sugieren que el preentrenamiento puede reducir significativamente las diferencias entre arquitecturas, lo que tendría implicaciones para la interpretación del ranking obtenido.

Escalado controlado. Finalmente, replicar el estudio a tres escalas (por ejemplo 30M, 300M y 3000M parámetros) y con longitudes crecientes (hasta 32k tokens o más) permitiría caracterizar empíricamente si las propiedades identificadas a escala pequeña —en particular el beneficio de la hibridación, el techo del estado de Mamba y la inversión de ALiBi entre tareas— se mantienen, se atenúan o se invierten al escalar. Este sería, con diferencia, el experimento más costoso pero también el de mayor valor para conectar los hallazgos del trabajo con las decisiones de diseño que se toman en sistemas de producción.

8.5. Reflexión final

El objetivo de este trabajo no ha sido decidir cuál es la mejor arquitectura, sino entender qué supone cada una sobre la estructura del problema y en qué condiciones ese supuesto se sostiene. Visto así, el resultado más informativo del estudio es el de Mamba. Alcanza una perplejidad comparable a la de los Transformers en WikiText-103 y, aun así, no aprende ListOps. La diferencia no es de implementación ni

de escala: su estado comprimido de dimensión fija supone que la información relevante del pasado cabe en unos pocos grados de libertad, y ese supuesto resulta razonable para el modelado de lenguaje pero incompatible con una tarea que exige recuperar resultados parciales exactos en posiciones anidadas arbitrarias.

Cada familia comparada responde, en realidad, a un supuesto distinto, y cada supuesto tiene un coste. La atención global asume que cualquier posición puede ser relevante para cualquier otra, y lo paga con un coste cuadrático en cómputo y memoria. Los modelos de espacio de estados asumen que el pasado útil puede resumirse en un estado de tamaño fijo, y lo pagan con la imposibilidad de recuperar de forma exacta lo que ese estado no conserva. La hibridación parte de que ninguno de los dos supuestos vale para todo, y por eso alterna capas de atención y de espacio de estados: así cada mecanismo cubre lo que el otro deja fuera. La codificación posicional ilustra el mismo principio con especial claridad: ALiBi, el mejor Transformer a contexto largo en WikiText-103, es el peor en ListOps, porque su sesgo de localidad penaliza justamente las dependencias distantes que el emparejamiento de paréntesis necesita. Ninguna decisión de diseño es buena o mala en sí misma; lo es en relación con la estructura de la tarea.

Esta es, en el fondo, la utilidad de un estudio comparativo controlado. Su aportación no es un veredicto sobre qué modelo conviene usar hoy —una respuesta que cambia con cada nueva publicación— sino una explicación de por qué cada diseño funciona donde lo hace, y esa explicación dura más que los modelos concretos. Las arquitecturas se suceden, pero los mecanismos que cada una aísla —la representación distribuida, la atención, la selectividad dependiente de la entrada— se incorporan al repertorio común del que parte el diseño siguiente. Es muy probable que los modelos relevantes dentro de unos años sigan evolucionando e incorporando nuevos mecanismos; sin embargo los principios que esta comparación ha permitido aislar —que la codificación posicional es determinante y depende de la tarea, que el razonamiento composicional necesita recuperación basada en contenido, que combinar varios paradigmas puede rendir más que apostar por uno solo— seguirán siendo útiles para tomar esas decisiones de diseño. Aportar a esa base de evidencia es lo que da sentido a este trabajo.

Apéndice

Apéndice A

Conjuntos de datos

A.1. Visión general de los conjuntos de datos

A lo largo del trabajo se emplean cinco conjuntos de datos que cubren tareas y dominios deliberadamente distintos: dos corpus propios contruidos a partir del diario *El Progreso* (**Progreso Tema** y **Progreso Género**), el corpus estándar de clasificación de texto **20newsgroups**, el corpus de modelado de lenguaje **WikiText-103** y el *benchmark* de razonamiento estructural **LRA ListOps**. La Tabla A.1 resume sus características principales; las secciones siguientes describen cada uno con el mismo nivel de detalle, acompañando la descripción de dos gráficas homogéneas: la distribución de clases o etiquetas y la distribución de la longitud de los textos.

Tabla A.1: Resumen comparativo de los conjuntos de datos utilizados.

Conjunto	Tarea	Idioma	Ejemplos	Clases	Long. media
Progreso Tema	Clasif. multiclase	Cast./Gal.	4 339	15	450 pal.
Progreso Género	Clasif. binaria	Cast./Gal.	5 430	2	454 pal.
20newsgroups	Clasif. multiclase	Inglés	18 846	20	200 pal.
WikiText-103	Modelado de leng.	Inglés	103.2M tok.	–	3 600 tok.
LRA ListOps	Razon. estructural	Sintético	~100 000	10	variable

A.2. Progreso

Los conjuntos Progreso se han construido de manera incremental a lo largo de varios meses (desde el 06/12/2023 hasta el 30/03/2024) a partir de noticias del diario *El Progreso* de Lugo. Para la recopilación se utilizó AWS (Amazon Web Services) y sus servicios Lambda y S3, que permiten automatizar el procesado y la extracción de los datos sin necesidad de aprovisionar un servidor propio:

- **Lambda**: servicio que permite ejecutar código sin aprovisionar ni administrar servidores.
- **S3**: servicio de almacenamiento de objetos en la nube.

En Lambda se desplegó un programa de Python que se ejecutaba diariamente, accedía al fichero "<https://www.elprogreso.es/sitemap.news.xml.gz>" que publica El Progreso y lo guardaba en formato

CSV en un *bucket* de S3. Dicho fichero contiene la URL de cada noticia disponible ese día, junto con información relevante (fecha, título, palabras clave, etc.). A partir de las URL se extrae el texto de cada noticia, descartando las que no son útiles (noticias en vídeo u otros formatos, o aquellas en las que la extracción falló), y se consolidan todos los datos en un único CSV.

A continuación se realiza, de forma local, una fase de limpieza y preparación. En primer lugar se añaden tres columnas:

- **TEMA**: temática de la noticia, extraída de la URL.
- **IDIOMA**: indicador del idioma del texto, creado manualmente mediante detección de términos y revisión, ya que el indicador original presentaba numerosos errores.
- **NOTICIAID**: identificador de cada noticia, que permite localizar sus versiones en castellano y gallego.

Tras eliminar noticias excesivamente cortas o con errores de formato, se obtienen los campos finales que recoge la Tabla A.2. A partir de este conjunto global se derivan los subconjuntos dedicados a cada tarea de procesamiento del lenguaje: clasificación multiclase (**Progreso Tema**), clasificación binaria (**Progreso Género**) y traducción (**progreso_trad**).

Tabla A.2: Columnas del conjunto de datos Progreso.

Columna	Tipo	Descripción
ID	Entero	Identificador de la fila
NOTICIAID	Entero	Identificador de la noticia
IDIOMA	Categorico	Idioma del texto
PUBLICATIONDATE	Fecha	Fecha de publicación
TITLE	Texto	Título de la noticia
LINK	Texto	URL de la noticia
GENRES	Categorico	Tipo de noticia (opinión o prensa)
KEYWORDS	Texto	Términos relevantes del texto
TEMA	Categorico	Temática de la noticia
TEXT	Texto	Contenido de la noticia

A.2.1. Progreso Tema

El conjunto **Progreso Tema** contiene 5 431 artículos clasificados según su temática. Aunque la URL distingue hasta 29 temáticas, muchas de ellas aparecen de forma marginal, por lo que solo se conservan las 15 más frecuentes, que reúnen 4 339 artículos. La Figura A.1 muestra a la izquierda esta distribución y, a la derecha, la distribución de la longitud de los textos (en palabras).

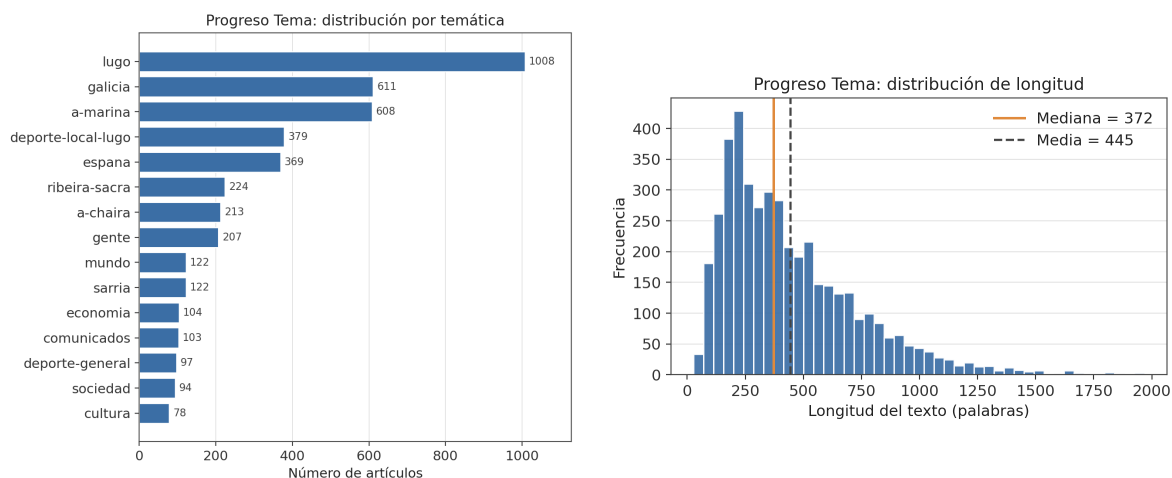


Figura A.1: Progreso Tema: distribución por temática (izquierda) y distribución de longitud de los artículos (derecha).

Las temáticas más comunes son las de carácter geográfico: *lugo* (ciudad donde se redacta el periódico), *galicia* (la comunidad autónoma) o comarcas de la provincia como *a-marina*, *ribeira-sacra* o *a-chaira*; junto a ellas aparecen categorías por tipo de contenido como *deporte-local-lugo*, *gente*, *cultura* o *economía*. La distribución es marcadamente desigual: *lugo* concentra por sí sola más artículos que las ocho temáticas menos frecuentes juntas, un desequilibrio que justifica el uso del F1 macro como métrica complementaria a la *accuracy* en los experimentos de clasificación.

La longitud de los textos presenta una asimetría positiva, con una mediana en torno a 370 palabras y una media de ~ 450 , incrementada de forma significativa por una cola de artículos extensos. La Figura A.2 desagrega esta longitud por temática mediante diagramas de caja ordenados por mediana: la mayoría de los temas se sitúan entre 300 y 600 palabras, destacando *comunicados* y *deporte-general* por su mayor extensión y *a-marina* por la menor.

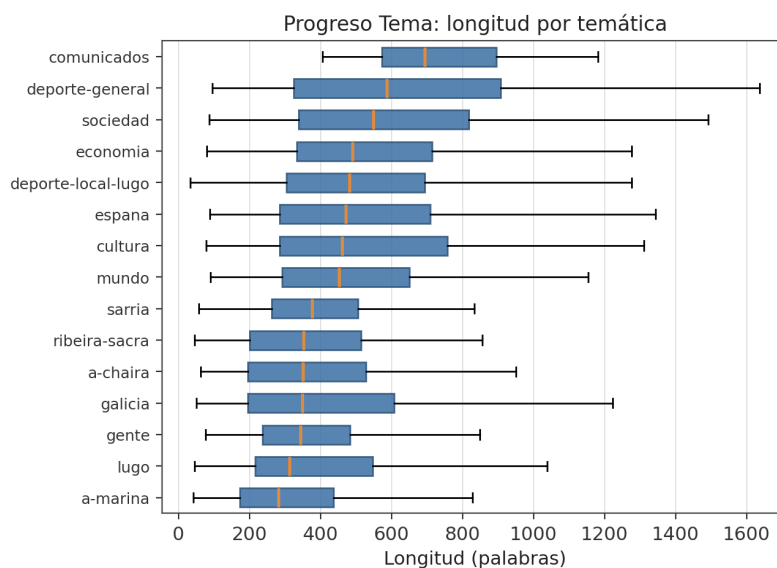


Figura A.2: Progreso Tema: distribución de la longitud (palabras) por temática.

Antes de entrenar los modelos, los textos se limpian y preprocesan. Primero se eliminan signos de puntuación y caracteres especiales y se convierte todo a minúsculas; a continuación se eliminan las *stopwords* (preposiciones, artículos, conjunciones y otras palabras muy frecuentes con poco contenido semántico) en los modelos entrenados desde cero, mientras que se mantienen en los modelos preentrenados. Después se construye un índice de palabras por frecuencia, limitado a las 10 000 más frecuentes, y se truncan los textos a 200 tokens para acelerar y estabilizar el entrenamiento. Por último, las etiquetas se codifican en vectores *one-hot* de dimensión igual al número de temáticas y se divide el conjunto en entrenamiento y test (80 % y 20 %).

A.2.2. Progreso Género

El conjunto **Progreso Género** reutiliza los mismos textos pero plantea una tarea de **clasificación binaria**: distinguir entre artículos de *prensa* (noticia informativa) y de *opinión*. Comprende 5 430 artículos, con un fuerte desequilibrio entre clases (4 650 de prensa frente a 780 de opinión, esto es, un 85,6 % frente a un 14,4 %), tal y como muestra la Figura A.3.

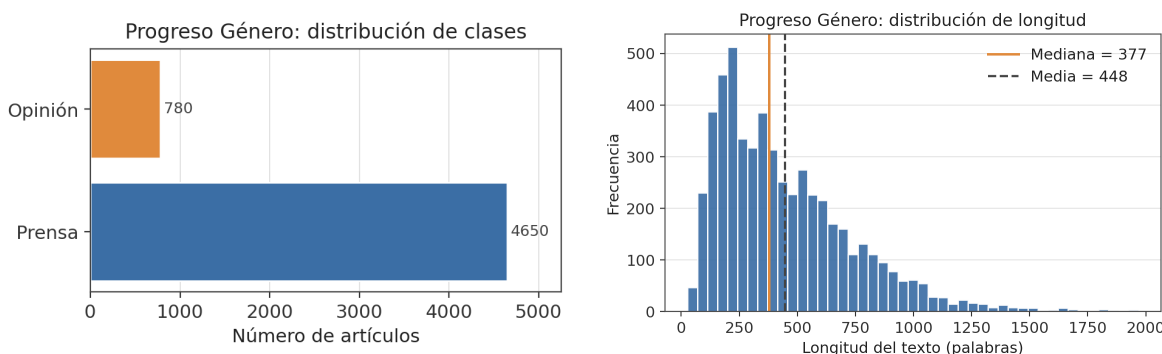


Figura A.3: Progreso Género: distribución de clases (izquierda) y distribución de longitud de los artículos (derecha).

Al compartir el corpus con Progreso Tema, la distribución de longitud es prácticamente idéntica (mediana ~ 370 palabras, media ~ 454). El desequilibrio de clases es, en este caso, aún más pronunciado que en Progreso Tema, lo que convierte a este subconjunto en un escenario especialmente exigente para métricas sensibles al balanceo de clases. El preprocesado es el mismo descrito para Progreso Tema, con la única diferencia de que la etiqueta es binaria.

A.3. 20newsgroups

El conjunto de datos 20newsgroups es un corpus estándar ampliamente utilizado en la literatura de clasificación de textos. Fue recopilado originalmente por Ken Lang y contiene artículos publicados en 20 grupos de noticias de Usenet, cada uno correspondiente a una temática diferente. Las categorías abarcan áreas como informática (*comp.**), ciencia (*sci.**), deportes y ocio (*rec.**), política (*talk.politics.**) y religión (*soc.religion.**, *alt.atheism*), entre otras.

En este trabajo se utiliza la versión proporcionada por scikit-learn, eliminando cabeceras, firmas y citas (*headers*, *footers*, *quotes*) para evitar que los modelos se apoyen en metadatos en lugar de en el contenido real de los textos. Tras este filtrado, el conjunto contiene 11 314 artículos de entrenamiento y 7 532 de test, distribuidos de forma aproximadamente equilibrada entre las 20 categorías, tal y como se muestra en la Figura A.4. Las categorías con menos representación son *talk.religion.misc* (377) y *talk.politics.misc* (465), mientras que la mayoría se sitúan en torno a los 580–600 artículos.

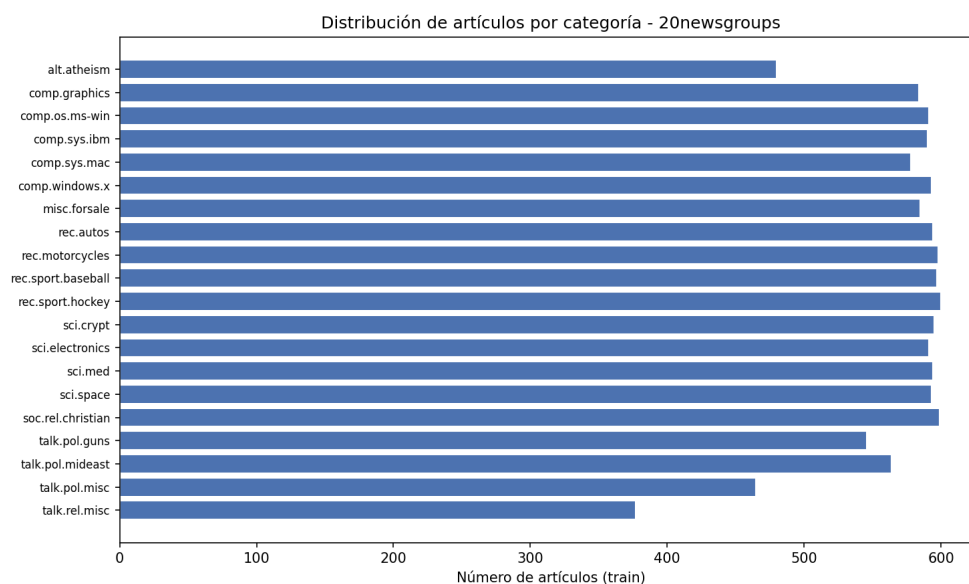


Figura A.4: Distribución de artículos por categoría en 20newsgroups (partición de entrenamiento)

En cuanto a la longitud de los textos, la distribución presenta una marcada asimetría positiva (Figura A.5): la mayoría de los artículos son relativamente cortos, con una mediana en torno a 120 palabras, pero existe una cola larga de artículos con varios miles de palabras. La media, cercana a 200 palabras, se ve incrementada de forma significativa por estos artículos extensos. Esta variabilidad en la longitud es relevante para nuestro estudio, ya que los modelos deben manejar tanto textos muy cortos como textos que superan con creces las longitudes máximas de secuencia evaluadas (256, 512 y 1024 tokens).

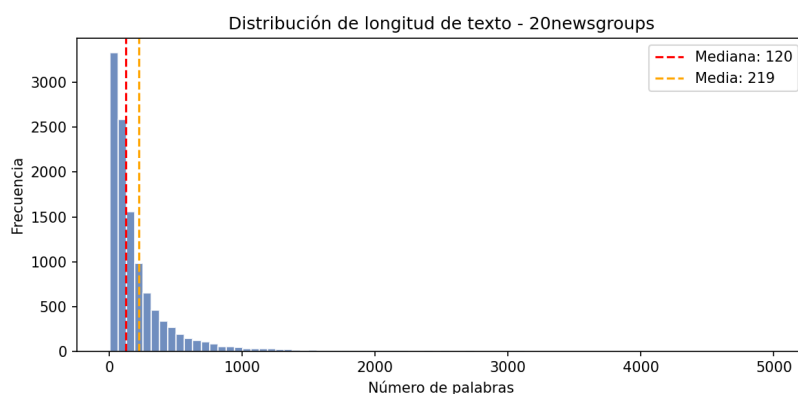


Figura A.5: Distribución de la longitud de texto (en palabras) en 20newsgroups

Una característica importante de este conjunto de datos es que, a diferencia de Progreso, los textos están en inglés. Esto implica que no hay comparación directa entre los rendimientos absolutos de los modelos en ambos conjuntos, pero sí permite verificar que los patrones observados (qué arquitectura destaca en cada tarea) son consistentes independientemente del idioma y del dominio.

A.4. WikiText-103

WikiText-103 es un corpus de modelado de lenguaje a nivel de palabra, creado por Merity et al. (2017) a partir de artículos verificados y destacados de la Wikipedia en inglés. A diferencia de otros corpus de lenguaje como Penn Treebank, WikiText-103 preserva la puntuación original, las mayúsculas y los números, proporcionando un texto más natural y representativo del lenguaje real.

El corpus contiene un total de aproximadamente 103 millones de tokens, distribuidos en unos 28 000 artículos con una longitud media de 3 600 tokens por artículo. La Tabla A.3 muestra la distribución por particiones.

Tabla A.3: Estadísticas de WikiText-103 por partición

Partición	Tokens	Vocabulario
Train	103.2M	267 735
Validación	218K	—
Test	246K	—

La Figura A.6 muestra la distribución de tokens por partición y la distribución de longitudes de los artículos. La distribución de longitudes es asimétrica, con una mediana de aproximadamente 1 800 tokens y una media de 3 600, indicando la presencia de artículos muy extensos que elevan la media. Esta variabilidad es especialmente relevante para nuestro análisis: los artículos más largos proporcionan contexto suficiente para evaluar cómo se comportan los modelos con secuencias de miles de tokens, mientras que los más cortos permiten comprobar que no se pierde rendimiento en secuencias breves.

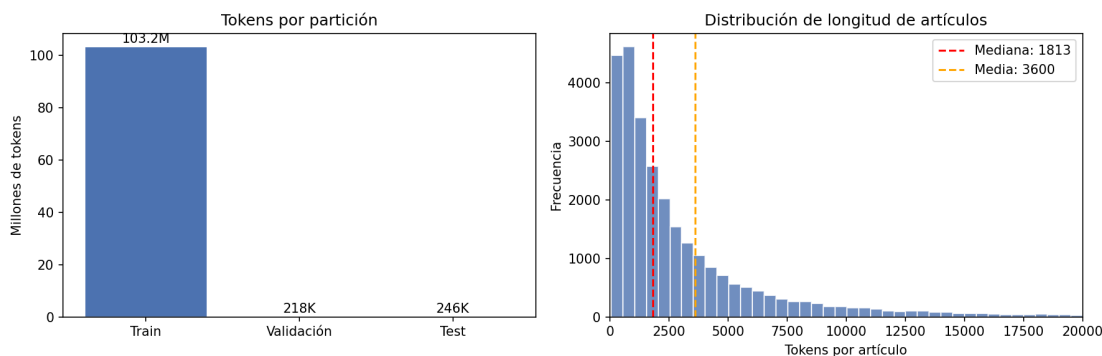


Figura A.6: WikiText-103: tokens por partición (izquierda) y distribución de longitud de artículos (derecha)

En nuestros experimentos, WikiText-103 se utiliza para la tarea de modelado de lenguaje autorregresivo: dado un contexto de tokens anteriores, el modelo debe predecir el siguiente token. El texto se divide en bloques de longitud fija (`BLOCK_SIZE`), y se utiliza el tokenizador de GPT-2 (vocabulario de aproximadamente 50 000 subpalabras). La métrica de evaluación es la perplejidad, que mide la calidad del modelo como predictor del siguiente token: valores más bajos indican un mejor ajuste al lenguaje. Variando el tamaño de bloque (128, 256, 512, 1024) podemos evaluar directamente cómo la longitud del contexto disponible afecta al rendimiento de cada arquitectura.

A.5. LRA ListOps

Antes de analizar los resultados, conviene caracterizar la estructura del conjunto de datos empleado en la tarea de razonamiento estructural. Utilizamos la versión `listops-1000` del benchmark *Long Range Arena* (LRA), accesible desde Hugging Face, que reproduce la tarea *ListOps* original con secuencias largas. Cada ejemplo es una expresión anidada formada por los operadores `MIN`, `MAX`, `MED` y `SUM_MOD` aplicados sobre listas de dígitos 0–9, y la etiqueta es el resultado de evaluar la expresión completa, es decir, un problema de clasificación en 10 clases. El conjunto de entrenamiento contiene alrededor de 96 000 ejemplos, con particiones de validación y test de 2 000 ejemplos cada una. Las cuatro figuras siguientes describen, respectivamente, la distribución de longitudes, la profundidad de anidamiento, el balanceo de clases y la frecuencia de operadores; todas se han calculado sobre el conjunto de entrenamiento.

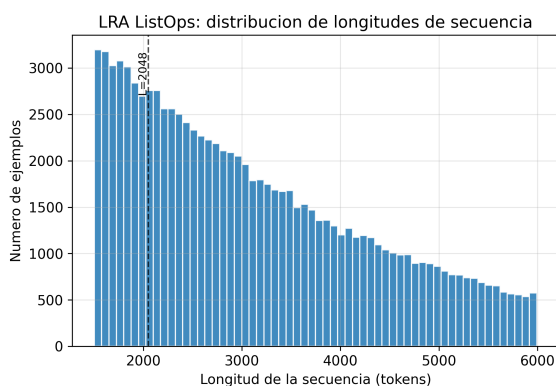


Figura A.7: Distribución de las longitudes de secuencia (en tokens) del conjunto de entrenamiento de LRA ListOps. La única línea vertical visible corresponde a $L = 2048$; los umbrales $L = 512$ y $L = 1024$ no aparecen porque quedan por debajo de la longitud mínima del conjunto.

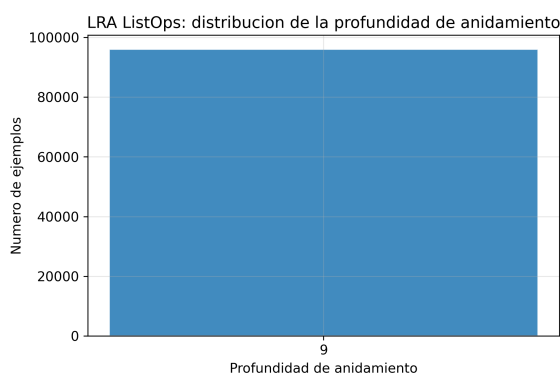


Figura A.8: Distribución de la profundidad máxima de anidamiento. Todos los ejemplos presentan exactamente profundidad 9, de modo que la dificultad jerárquica es homogénea en todo el conjunto.

Si pensamos en lo que estas dos primeras figuras implican, encontramos el rasgo más determinante del conjunto. La Figura A.7 muestra que las secuencias son uniformemente largas: la más corta supera los 1 500 tokens y la cola se extiende hasta cerca de 6 000. En consecuencia, las dos longitudes máximas más pequeñas que evaluamos, $L = 512$ y $L = 1024$, *truncan la totalidad* de los ejemplos — por eso sus líneas ni siquiera aparecen en el histograma — y solo $L = 2048$ deja sin truncar una minoría de las secuencias (en torno al 20 %). Esto matiza la interpretación del barrido de longitudes: en ListOps no estamos comparando secuencias cortas frente a secuencias completas, sino la *fracción* de una expresión intrínsecamente larga que cada modelo llega a observar. Dado que la profundidad de anidamiento es constante e igual a 9 (Figura A.8), resolver la expresión exige propagar información desde el operador más externo hasta sus operandos más profundos, de modo que un truncamiento severo elimina precisamente la estructura necesaria para calcular la respuesta.

La Figura A.9 revela un desequilibrio relevante: aunque la tarea tiene 10 clases, las salidas 0 y 9 aparecen muy por encima de la frecuencia uniforme (alrededor de 16 000 ejemplos cada una, frente a los 7 000–8 700 de las clases intermedias). Este sesgo es esperable: el operador `MIN` tiende a producir valores bajos (con el 0 como resultado más frecuente) y `MAX` valores altos (con el 9 como resultado más frecuente), de modo que las clases extremas acumulan una frecuencia muy superior a la del resto. Esto tiene una consecuencia práctica sobre la referencia con la que comparar los modelos: un clasificador

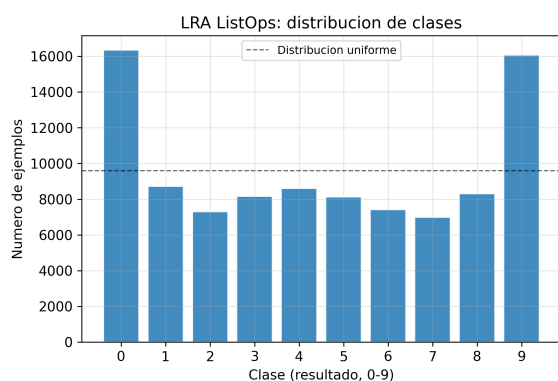


Figura A.9: Distribución de las clases de salida (0–9). La línea discontinua indica la distribución uniforme; las clases 0 y 9 están claramente sobre-representadas.

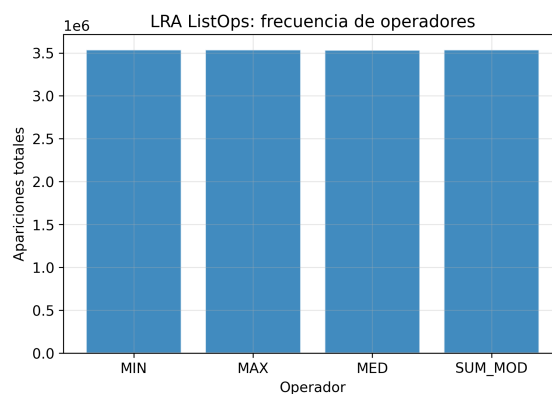


Figura A.10: Frecuencia total de aparición de cada operador. Los cuatro operadores aparecen de forma prácticamente equilibrada, con del orden de 150 operadores por ejemplo de media.

que predijese al azar entre las 10 clases obtendría un 10 % de acierto, pero un clasificador trivial que predijese siempre la clase más frecuente (0) alcanzaría ya un $\sim 17\%$. Esta última cifra es, por tanto, la línea de base realmente exigente: un modelo que no la supere no está aprendiendo la estructura de la tarea. Podemos verificar que este valor coincide con el comportamiento de los modelos que no aprenden la estructura de la tarea: como veremos en el Capítulo 7, FFN, LSTM y Mamba se estancan en torno al 16–22 %, es decir, esencialmente en la base de la clase mayoritaria, mientras que las variantes con atención superan claramente ese umbral. Finalmente, la Figura A.10 confirma que los cuatro operadores están equilibrados, por lo que la dificultad no proviene de la rareza de ningún operador concreto, sino del razonamiento jerárquico de largo alcance sobre secuencias largas.

Bibliografía

- [1] Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebrón, F., & Sanghai, S. (2023). GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP 2023)*.
- [2] Arora, S., Eyuboglu, S., Timalisina, A., Johnson, I., Poli, M., Zou, J., Rudra, A., & Ré, C. (2024). Zoology: Measuring and Improving Recall in Efficient Language Models. In *Proceedings of the International Conference on Learning Representations (ICLR 2024)*.
- [3] Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural Machine Translation by Jointly Learning to Align and Translate. *arXiv preprint arXiv:1409.0473*.
- [4] Beltagy, I., Peters, M. E., & Cohan, A. (2020). Longformer: The Long-Document Transformer. *arXiv preprint arXiv:2004.05150*.
- [5] Chen, S., Wong, S., Chen, L., & Tian, Y. (2023). Extending Context Window of Large Language Models via Positional Interpolation. *arXiv preprint arXiv:2306.15595*.
- [6] Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2(4), 303–314.
- [7] Dao, T., Fu, D. Y., Ermon, S., Rudra, A., & Ré, C. (2022). FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems (NeurIPS 2022)*.
- [8] Dao, T. (2023). FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. *arXiv preprint arXiv:2307.08691*.
- [9] Dao, T., & Gu, A. (2024). Transformers are SSMS: Generalized Models and Efficient Algorithms Through Structured State Space Duality. In *Proceedings of the 41st International Conference on Machine Learning (ICML 2024)*.
- [10] Dauphin, Y. N., Fan, A., Auli, M., & Grangier, D. (2017). Language Modeling with Gated Convolutional Networks. In *Proceedings of the 34th International Conference on Machine Learning (ICML 2017)*.
- [11] Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2018). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805*.
- [12] Fedus, W., Zoph, B., & Shazeer, N. (2022). Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *Journal of Machine Learning Research*, 23(120), 1–39.
- [13] Gale, T., Narayanan, D., Young, C., & Zaharia, M. (2023). MegaBlocks: Efficient Sparse Training with Mixture-of-Experts. In *Proceedings of Machine Learning and Systems (MLSys 2023)*.
- [14] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.

- [15] Gu, A., Dao, T., Ermon, S., Rudra, A., & Ré, C. (2020). HiPPO: Recurrent Memory with Optimal Polynomial Projections. In *Advances in Neural Information Processing Systems (NeurIPS 2020)*.
- [16] Gu, A., Goel, K., & Ré, C. (2022). Efficiently Modeling Long Sequences with Structured State Spaces. In *Proceedings of the International Conference on Learning Representations (ICLR 2022)*.
- [17] Gu, A., & Dao, T. (2024). Mamba: Linear-Time Sequence Modeling with Selective State Spaces. In *Proceedings of the 41st International Conference on Machine Learning (ICML 2024)*.
- [18] He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep Residual Learning for Image Recognition. *arXiv preprint arXiv:1512.03385*.
- [19] Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9(8), 1735–1780.
- [20] Hornik, K. (1991). Approximation Capabilities of Multilayer Feedforward Networks. *Neural Networks*, 4(2), 251–257.
- [21] Jelassi, S., Brandfonbrener, D., Kakade, S. M., & Malach, E. (2024). Repeat After Me: Transformers are Better than State Space Models at Copying. In *Proceedings of the 41st International Conference on Machine Learning (ICML 2024)*.
- [22] Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., ... & Sayed, W. E. (2024). Mixtral of Experts. *arXiv preprint arXiv:2401.04088*.
- [23] Jurafsky, D., & Martin, J. H. (2025). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. 3rd ed. draft, Stanford University.
- [24] Lieber, O., Lenz, B., Bata, H., Cohen, G., Osin, J., Dalmedigos, I., ... & Shoham, Y. (2024). Jamba: A Hybrid Transformer-Mamba Language Model. *arXiv preprint arXiv:2403.19887*.
- [25] Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., & Stoyanov, V. (2019). *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. *arXiv preprint arXiv:1907.11692*.
- [26] Merity, S., Xiong, C., Bradbury, J., & Socher, R. (2017). Pointer Sentinel Mixture Models. In *Proceedings of the 5th International Conference on Learning Representations (ICLR 2017)*.
- [27] Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. In *Proceedings of the International Conference on Learning Representations (ICLR 2013)*.
- [28] Peng, B., Quesnelle, J., Fan, H., & Shippole, E. (2023). YaRN: Efficient Context Window Extension of Large Language Models. *arXiv preprint arXiv:2309.00071*.
- [29] Press, O., Smith, N. A., & Lewis, M. (2022). Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation. In *Proceedings of the International Conference on Learning Representations (ICLR 2022)*.
- [30] Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). Improving Language Understanding by Generative Pre-Training. *OpenAI Blog*.
- [31] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language Models are Unsupervised Multitask Learners. *OpenAI Technical Report*.
- [32] Sennrich, R., Haddow, B., & Birch, A. (2016). Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL 2016)*.

- [33] Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., & Dean, J. (2017). Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. In *Proceedings of the International Conference on Learning Representations (ICLR 2017)*.
- [34] Shazeer, N. (2019). Fast Transformer Decoding: One Write-Head is All You Need. *arXiv preprint arXiv:1911.02150*.
- [35] Shazeer, N. (2020). GLU Variants Improve Transformer. *arXiv preprint arXiv:2002.05202*.
- [36] Su, J., Lu, Y., Pan, S., Murtadha, A., Wen, B., & Liu, Y. (2021). RoFormer: Enhanced Transformer with Rotary Position Embedding. *arXiv preprint arXiv:2104.09864*.
- [37] Tay, Y., Dehghani, M., Abnar, S., Shen, Y., Bahri, D., Pham, P., Rao, J., Yang, L., Ruder, S., & Metzler, D. (2021). Long Range Arena: A Benchmark for Efficient Transformers. In *Proceedings of the International Conference on Learning Representations (ICLR 2021)*.
- [38] Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., ... & Lample, G. (2023). LLaMA: Open and Efficient Foundation Language Models. *arXiv preprint arXiv:2302.13971*.
- [39] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention Is All You Need. In *Proceedings of the 31st Conference on Neural Information Processing Systems (NIPS 2017)*, Long Beach, CA, USA.
- [40] Waleffe, R., Byeon, W., Riach, D., Norick, B., Korthikanti, V., Dao, T., Gu, A., Hatamizadeh, A., Singh, S., Narayanan, D., Kulshreshtha, G., Singh, V., Casper, J., Kautz, J., Shoeybi, M., & Catanzaro, B. (2024). An Empirical Study of Mamba-based Language Models. *arXiv preprint arXiv:2406.07887*.
- [41] Xiong, R., Yang, Y., He, D., Zheng, K., Zheng, S., Xing, C., Zhang, H., Lan, Y., Wang, L., & Liu, T.-Y. (2020). On Layer Normalization in the Transformer Architecture. In *Proceedings of the 37th International Conference on Machine Learning (ICML 2020)*.
- [42] Zaheer, M., Guruganesh, G., Dubey, K. A., Ainslie, J., Alberti, C., Ontañón, S., Pham, P., Ravula, A., Wang, Q., Yang, L., & Ahmed, A. (2020). Big Bird: Transformers for Longer Sequences. In *Advances in Neural Information Processing Systems (NeurIPS 2020)*.
- [43] Zhang, B., & Sennrich, R. (2019). Root Mean Square Layer Normalization. In *Advances in Neural Information Processing Systems (NeurIPS 2019)*.