



Université Sultan Moulay Slimane
École Nationale des Sciences Appliquées
-Khouribga-



Final Year Project Report
In view of obtaining the degree

MASTER

Program: Big Data et Aide à la Décision

Attention Mechanisms in Transformers: A Comparative Survey and Structural Enhancements to Linear Attention

Prepared by:

Mr. BENNANI Nizar

Under the supervision of:

Pr. Abdelghani GHAZDALI

Defended on June 17, 2025, in front of the jury:

Pr. Abdelghani GHAZDALI	ENSA Khouribga – Supervisor
Pr. Aziz IFZARNE	ENSA Khouribga – President
Pr. Moad HAKIM	ENSA Khouribga – Examiner

Academic Year: 2024-2025

Acknowledgements

First and foremost, I would like to express my deepest gratitude to **God** for granting me the strength, patience, and perseverance throughout this journey. Without divine guidance and blessings, none of this would have been possible.

I am profoundly thankful to **my family**, whose unwavering love, encouragement, and support have been a constant source of strength. Their faith in me has helped me overcome countless challenges and kept me grounded through every stage of my academic journey.

My heartfelt thanks also go to **my friends**, who stood by me with kindness and motivation. Their presence brought lightness and laughter during moments of stress and fatigue, and I am truly grateful for their companionship.

I extend my sincere appreciation to my school and all the professors who have guided me, challenged me, and contributed to my learning over the years. Their dedication and passion for teaching have played a vital role in shaping the person and professional I am becoming.

Finally, I wish to thank my project supervisor, **Professor Abdelghani GHAZDALI**, for their valuable guidance, thoughtful feedback, and continuous support throughout the course of this project. Their mentorship has been instrumental to both the development of this work and my growth as a student.

To all those who have accompanied me along this path thank you from the bottom of my heart.

Abstract

Transformer models have transformed natural language processing by employing self-attention mechanisms to model long-range dependencies. However, the quadratic complexity of full self-attention poses significant challenges for scaling to long sequences and deploying models in resource-constrained environments. In this work, we present a comprehensive study of alternative attention mechanisms including standard multi-head attention, linear attention, sparse attention, sliding-window attention, and FlashAttention. Our analysis spans both theoretical foundations and empirical performance, considering not only a variety of NLP tasks but also a range of Transformer architectures: encoder-only (e.g., BERT), decoder-only (e.g., GPT), and encoder-decoder models (e.g., full Transformer).

Motivated by the limitations observed in linear attention particularly its reduced expressiveness despite its computational efficiency we propose two novel hybrid attention mechanisms designed to improve its performance in decoder-only models. The first mechanism integrates the local context sensitivity of sliding-window attention into the linear framework, while the second incorporates global sparsity patterns inspired by sparse attention. These hybrids aim to maintain the linear time and space complexity of linear attention while enhancing its ability to model contextual dependencies more effectively.

Experimental results on the Tiny Shakespeare next-character prediction task demonstrate that our proposed hybrid mechanisms significantly outperform standard linear attention in terms of loss and perplexity. Unlike prior work that primarily focuses on designing faster or approximate attention variants, our contribution explicitly targets the recovery of performance lost in linear attention mechanisms. This focus on restoring predictive quality while retaining linear complexity is the core innovation of this work.

Although our evaluation is limited to a small-scale setting, the findings suggest that these enhancements could be beneficial when scaled to more complex models and larger datasets. This work highlights the importance of tailoring attention mechanisms not only to specific tasks but also to the structural design of Transformer architectures.

Keywords: Transformer models, self-attention, linear attention, sparse attention, hybrid attention mechanisms, decoder-only models, contextual dependencies, NLP tasks, performance recovery, Tiny Shakespeare dataset

Résumé

Les modèles Transformer ont révolutionné le traitement automatique du langage naturel grâce à l'utilisation de mécanismes d'auto-attention permettant de modéliser les dépendances à longue portée. Cependant, la complexité quadratique de l'auto-attention complète constitue un obstacle majeur à l'extension vers de longues séquences et à la mise en œuvre dans des environnements à ressources limitées. Dans ce travail, nous présentons une étude approfondie de mécanismes d'attention alternatifs, incluant l'attention multi-tête standard, l'attention linéaire, l'attention clairsemée, l'attention à fenêtre glissante et FlashAttention. Notre analyse couvre à la fois les fondements théoriques et les performances empiriques, en considérant non seulement une variété de tâches de NLP, mais aussi différents types d'architectures Transformer : encodeur seul (par exemple BERT), décodeur seul (par exemple GPT) et modèles encodeur-décodeur (par exemple Transformer complet).

Motivés par les limites observées dans l'attention linéaire notamment sa capacité d'expression réduite malgré son efficacité computationnelle nous proposons deux nouveaux mécanismes hybrides conçus pour améliorer ses performances dans les modèles à décodeur seul. Le premier intègre la sensibilité au contexte local de l'attention à fenêtre glissante dans le cadre linéaire, tandis que le second incorpore des schémas de rareté globale inspirés de l'attention clairsemée. Ces mécanismes hybrides visent à conserver la complexité linéaire en temps et en mémoire de l'attention linéaire, tout en améliorant sa capacité à modéliser les dépendances contextuelles de manière plus efficace.

Les résultats expérimentaux sur la tâche de prédiction du caractère suivant dans le corpus Tiny Shakespeare montrent que nos mécanismes hybrides surpassent significativement l'attention linéaire standard en termes de perte et de perplexité. Contrairement aux travaux antérieurs qui se concentrent principalement sur la conception de variantes d'attention plus rapides ou approximatives, notre contribution vise explicitement à restaurer la performance perdue dans les mécanismes d'attention linéaire. Cette volonté de rétablir la qualité prédictive tout en conservant une complexité linéaire constitue l'innovation principale de ce travail.

Bien que notre évaluation soit limitée à un cadre de petite échelle, nos résultats suggèrent que ces améliorations pourraient être bénéfiques lorsqu'elles sont appliquées à des modèles plus complexes et à des ensembles de données plus volumineux. Ce travail souligne l'importance d'adapter les mécanismes d'attention non seulement aux tâches spécifiques, mais aussi à la structure des architectures Transformer.

Mots-clés : modèles Transformer, auto-attention, attention linéaire, attention clairsemée, mécanismes hybrides, modèles à décodeur seul, dépendances contextuelles, tâches de NLP, restauration des performances, corpus Tiny Shakespeare

الملخص

لقد أحدثت نماذج المحولات تغييراً كبيراً في مجال معالجة اللغة الطبيعية من خلال اعتماد آلية الانتباه الذاتي لنمذجة العلاقات البعيدة بين الكلمات. ومع ذلك، فإن التعقيد التربيعي للانتباه الكامل يشكل تحدياً كبيراً عند التعامل مع تسلسلات طويلة أو عند تنفيذ النماذج في بيئات محدودة الموارد. في هذا العمل، نقدم دراسة شاملة لآليات الانتباه البديلة، بما في ذلك الانتباه التقليدي متعدد الرؤوس، والانتباه الخطي، والانتباه المتناثر، والانتباه القائم على النوافذ المنزلقة، والانتباه السريع.

يشمل تحليلنا الجوانب النظرية والأداء التجريبي، وذلك عبر مجموعة متنوعة من مهام معالجة اللغة الطبيعية وبنى مختلفة من نماذج المحولات، مثل النماذج التي تعتمد فقط على التشفير، أو فقط على فك التشفير، أو تلك التي تجمع بين التشفير وفك التشفير.

وانطلاقاً من القيود التي لاحظناها في الانتباه الخطي، خصوصاً من حيث ضعف قدرته على تمثيل السياق رغم كفاءته العالية، نقترح آليتين هجنتين جديديتين تهدفان إلى تحسين أدائه في النماذج المعتمدة على فك التشفير فقط. الآلية الأولى تدمج الحساسية للسياق المحلي المستمدة من الانتباه القائم على النوافذ ضمن الإطار الخطي، بينما تعتمد الآلية الثانية على أنماط تبعثر عالمية مستوحاة من الانتباه المتناثر. تهدف هاتان الآليتان إلى الحفاظ على تعقيد زمني ومكاني خطي مع تعزيز قدرة النموذج على التقاط السياق بشكل أفضل.

تظهر النتائج التجريبية على مهمة توقع الحرف التالي في نصوص صغيرة أن الآليتين المقترحتين تتفوقان بشكل ملحوظ على الانتباه الخطي التقليدي من حيث انخفاض الخسارة وتحسن جودة التنبؤ. وعلى عكس الأعمال السابقة التي ركزت على تسريع الانتباه أو تقريب نتائجه، فإن مساهمتنا تركز بشكل واضح على استعادة جودة الأداء المفقودة بسبب التبسيط الحسابي. ويعد هذا الهدف جوهر الابتكار في هذا العمل.

ورغم أن التقييم اقتصر على نطاق محدود، تشير النتائج إلى أن هذه التحسينات قد تكون قابلة للتوسيع لتشمل نماذج أكثر تعقيداً ومهام أكبر حجماً. يسلط هذا العمل الضوء على أهمية مواءمة آليات الانتباه ليس فقط حسب نوع المهمة، بل أيضاً حسب البنية المعمارية للنموذج.

الكلمات المفتاحية: نماذج المحولات، الانتباه الذاتي، الانتباه الخطي، الانتباه المتناثر، الآليات الهجينة، نماذج فك التشفير، الاعتماد السياقي، مهام معالجة اللغة الطبيعية، استعادة الأداء، مجموعة بيانات قني ضهكسر

Motivation

As Transformer models continue to dominate the landscape of natural language processing, the demand for scaling them efficiently both in terms of memory and compute has become increasingly urgent. Full self-attention, despite its effectiveness, imposes a quadratic cost with respect to sequence length, making it a bottleneck in long context applications or when deployed under resource constraints. Various solutions have been proposed to address this, yet many introduce new limitations, such as reduced expressivity or increased architectural complexity.

Motivated by this tension between efficiency and performance, our work investigates how different attention mechanisms behave across Transformer architectures and NLP tasks. We conduct an extensive comparative study of standard attention, linear attention, sparse attention, sliding-window attention, and FlashAttention, evaluating them not only across tasks such as masked language modeling, next-character prediction, and translation, but also across architectural types: encoder-only, decoder-only, and encoder-decoder models. This broader perspective reveals that linear attention, while attractive for its low memory footprint, struggles particularly in decoder-only setups like GPT, leading to higher loss and perplexity.

To address this limitation, we introduce two hybrid attention mechanisms that enhance linear attention using structural ideas drawn from sparse and sliding-window attention. One emphasizes localized token interactions; the other incorporates sparse global patterns. The goal is to increase the representational power of linear attention without negating its efficiency gains.

Using a lightweight GPT model trained on the Tiny Shakespeare dataset, we show that both hybrid variants achieve loss and perplexity close to those of full attention, while maintaining significantly lower memory usage. Although these hybrids do not outperform standard attention, their ability to narrow the performance gap while remaining efficient suggests a promising path toward scalable Transformer architectures suitable for deployment in constrained environments.

List of Abbreviations

Abbreviation	Meaning
attn	attention
bwd	backward
ctx	context
dim	dimension
doc	document
enc	encoder
FFN	Feed Forward Network
GPU	Graphics Processing Unit
GPT	Generative Pretrained Transformer
grad	gradient
IO	Input/Output
k/v	key/value (in attention)
mem	memory
MHA	Multi-Head Attention
nctx	number of context tokens
n_embd	embedding size
n_head	number of attention heads
n_layer	number of layers
n_vocab	vocabulary size
NLL	Negative Log Likelihood
param	parameter
pos_enc	positional encoding
QKV	Query/Key/Value
rel_pos_enc	relative positional encoding
res	residual
RMSNorm	Root Mean Square Normalization
seq_len	sequence length
softmax	softmax function
std	standard deviation
sw	sliding window
TFM	Transformer
vocab	vocabulary
win_size	window size

Contents

Acknowledgements	2
Abstract	3
Motivation	6
Abbreviations	7
1 Background and Related Work	15
1.1 Transformer Architecture	15
1.1.1 From Sequential Constraints to Parallelized Context: How Transformers Overcome RNN Limitations	15
1.1.2 Overview of the Transformer Architecture	18
1.1.3 Self-Attention	20
1.1.4 Multi-head Attention	24
1.1.5 The Feed-Forward Layer	25
1.1.6 Positional Encoding	26
1.1.7 Add & Normalize Layer	29
1.1.8 The Final Linear and Softmax Layers (Decoder)	30
1.1.9 Encoder-Decoder Architecture and Information Flow in Transformers	31
1.2 The Transformer Tree of Life	32
1.2.1 The Encoder Branch	33
1.2.2 The Decoder Branch	34
1.2.3 The Encoder-Decoder Branch	35
1.3 Conclusion	35
2 State of the Art in Attention Mechanisms	36
2.1 Linear Attention: A Kernel-Based Approach to Efficient Sequence Modeling	36
2.1.1 Standard Attention: The Quadratic Bottleneck	36
2.1.2 Kernel-Based Reformulation	37
2.1.3 Feature Map Decomposition	37
2.1.4 Linear Complexity Derivation	37
2.1.5 Feature Map Selection	38
2.1.6 Mathematical Analysis of Algorithmic Steps	38
2.2 Sparse Attention Mechanisms: A Theoretical Analysis	39
2.2.1 Formal Definition	39
2.2.2 Sparse Attention Patterns: Design and Theoretical Foundations	39
2.2.3 Theoretical Analysis of Complexity and Expressivity	41
2.3 Sliding Window Attention	41
2.3.1 Theoretical Foundation: Locality in Sequence Modeling	41
2.3.2 Mechanism: Fixed-Length Contextual Windows	42
2.3.3 Receptive Field Properties	42
2.3.4 Mathematical Formulation	43
2.3.5 Complexity Analysis	43
2.4 Understanding Flash Attention: Efficient Transformer Attention Mechanisms	44

2.4.1	Why is Self-Attention Slow?	44
2.4.2	The Numerical Stability Challenge in Softmax	45
2.4.3	Traditional Softmax Algorithm	45
2.4.4	Online Softmax: A Dynamic Approach	46
2.4.5	Block Matrix Multiplication in Flash Attention	48
2.4.6	Flash Attention: Tiling and IO-Aware Algorithms	49
2.5	Conclusion	52
3	Experimental Results and Comparative Analysis	53
3.1	Experimental Setup and Implementation Details	53
3.1.1	Overview of Attention Mechanisms	53
3.1.2	Evaluation Metrics	54
3.1.3	Experimental Setup	54
3.1.4	Experimental Philosophy and Fairness	54
3.2	Comparison Using GPT Models	55
3.2.1	Task Definition	55
3.2.2	Model Configuration	55
3.2.3	Dataset	55
3.2.4	Training Setup	55
3.2.5	Performance Results	56
3.2.6	Ranking of Attention Mechanisms	58
3.2.7	Discussion	58
3.3	Comparison Using BERT-Like Models	59
3.3.1	Task Definition	59
3.3.2	Model Configuration	59
3.3.3	Dataset	59
3.3.4	Training Setup	59
3.3.5	Performance Results	60
3.3.6	Ranking of Attention Mechanisms	62
3.3.7	Discussion	62
3.4	Comparison Using Encoder–Decoder Transformers for Translation	62
3.4.1	Task Definition	62
3.4.2	Model Configuration	62
3.4.3	Dataset	63
3.4.4	Training Setup	63
3.4.5	Performance Results	63
3.4.6	Ranking of Attention Mechanisms	65
3.4.7	Discussion	66
3.5	Conclusion	66
4	Enhancing Linear Attention with Structural Biases: Design and Evaluation of Hybrid Mechanisms	68
4.1	Motivation	68
4.2	Proposed Hybrid Mechanisms	69
4.2.1	LinearSparseAttention	70
4.2.2	LinearSlidingWindowAttention	70
4.3	Implementation Details	70
4.4	Experimental Results and Analysis	71
4.4.1	Training and Validation Loss	71
4.4.2	Training and Validation Perplexity	71
4.4.3	Training Time	72
4.4.4	Inference Latency	72

4.4.5	Memory Usage	73
4.5	Discussions	73
4.6	Limitations and Future Work	74
4.6.1	Limitations	74
4.6.2	Future Work	74
4.7	Conclusion	75
	Conclusion	76
	Bibliography	78

List of Figures

1.1	Sequential processing in RNNs.	16
1.2	Self-attention mechanism in Transformers.	17
1.3	Training parallelism in Transformers.	18
1.4	Overview of the Transformer architecture, highlighting the encoder-decoder structure and key components.	20
1.5	Visualization of self-attention weights for the word "flies" in different contexts. .	21
1.6	Generation of Query, Key, and Value vectors.	22
1.7	Computation of Self-Attention.	23
1.8	Matrix-based computation of Query, Key, and Value matrices.	23
1.9	Matrix-based computation of self-attention output.	24
1.10	Visualization of multi-head attention, showing how multiple attention heads capture different linguistic features.	25
1.11	Visualization of sinusoidal positional encodings.	27
1.12	Addition of positional encodings to token embeddings.	28
1.13	Heatmap visualization of positional encoding across Transformer layers.	28
1.14	The Final Linear and Softmax Layers.	31
1.15	An overview of some of the most prominent transformer architectures	32
2.1	Sparse attention patterns visualization	40
2.2	Sliding window attention mechanism visualization	42
2.3	Receptive field expansion across layers	43
2.4	Illustration of standard attention computation	44
2.5	Traditional softmax computation process	46
2.6	Online softmax computation process	47
2.7	IO-aware tiling in Flash Attention	50
2.8	Block-wise processing in Flash Attention	51
3.1	Training and validation loss curves for five attention variants	56
3.2	Training and validation perplexity curves for five attention variants	57
3.3	Performance metrics comparison across attention variants	57
3.4	Training and validation loss for BERT-like models with different attention mechanisms	60
3.5	Training and validation accuracy for BERT-like models with different attention mechanisms	60
3.6	Performance metrics for BERT-like models with different attention mechanisms .	61
3.7	Training and validation loss for translation models with different attention mechanisms	63
3.8	BLEU scores for translation models with different attention mechanisms	64
3.9	Efficiency metrics for translation models with different attention mechanisms . .	65
4.1	Addressing the problem with linear attention	69
4.2	Training and validation loss curves	71

4.3	Training and validation perplexity curves	71
4.4	Training time comparison	72
4.5	Inference latency comparison	72
4.6	Memory usage comparison	73

List of Tables

3.1	Evaluation metrics used in our comparative analysis	54
3.2	Summary of performance metrics across attention variants	58
3.3	Ranking of attention mechanisms across performance metrics	58
3.4	Summary of performance metrics for BERT-like models	61
3.5	Ranking of attention mechanisms for BERT-like models	62
3.6	Summary of performance metrics for translation models	65
3.7	Ranking of attention mechanisms for translation models	66

General Introduction

Transformer architectures have fundamentally transformed natural language processing (NLP) by introducing a flexible and scalable self-attention mechanism that enables models to capture long range dependencies in text. Since their introduction Transformers have become the backbone of state-of-the-art models across a wide array of NLP tasks, ranging from language modeling and translation to summarization and question answering. However, the standard self-attention mechanism, while powerful, imposes a quadratic cost in both time and memory relative to the sequence length posing significant challenges for long-context processing and deployment in constrained environments.

In response to these limitations, the NLP community has proposed a range of efficient attention mechanisms aimed at reducing the computational footprint of Transformers without sacrificing performance. These include **linear attention**, which replaces softmax attention with kernel-based approximations; **sparse attention**, which limits token interactions to structured subsets; **sliding-window attention**, which restricts focus to local neighborhoods; and **FlashAttention**, which accelerates full attention using optimized GPU memory access patterns and numerically stable softmax computation.

To better understand the trade-offs introduced by these mechanisms, our work begins with a **comprehensive empirical and theoretical study** of attention variants. We evaluate standard full attention, linear attention, sparse attention, sliding-window attention, and FlashAttention across **multiple Transformer architectures**:

- **Decoder-only models** (GPT-like) for next-character prediction
- **Encoder-only models** (BERT-like) for masked language modeling
- **Encoder-decoder models** (full Transformers) for sequence-to-sequence tasks like translation

This analysis spans several NLP tasks, each with distinct architectural demands, allowing us to evaluate how different attention mechanisms perform under varied conditions. We measure a broad range of metrics including **loss**, **perplexity**, **BLEU score**, **memory consumption**, and **runtime latency** to assess both the **efficiency** and **expressiveness** of each mechanism in practice.

Findings from this study reveal that **linear attention**, while highly efficient in memory and runtime, tends to underperform in autoregressive decoder-only models, where capturing long range generative dependencies is crucial. **Sparse and sliding-window mechanisms** offer a middle ground, improving expressiveness at the cost of some additional complexity. These observations motivate our core contribution: the design of **hybrid attention mechanisms** that enhance linear attention by incorporating structural ideas from sparse and sliding-window patterns.

The rest of this work presents our hybrid attention designs, their implementation in GPT-style models, and an empirical evaluation on the Tiny Shakespeare dataset. Our goal is to demonstrate how combining the strengths of multiple attention paradigms can help balance the trade-off between **computational efficiency** and **modeling capacity**, especially in lightweight Transformer models suited for constrained environments.

Chapter 1

Background and Related Work

This chapter delves into the foundational components of Transformer models, examining the core mechanisms that enable their effectiveness. We begin by exploring the attention mechanism, which serves as the cornerstone of the Transformer architecture. The attention mechanism, introduced in [1], revolutionized sequence modeling by enabling parallelized computation and direct access to global context. Subsequently, we introduce the additional components necessary to construct a functional Transformer encoder. Furthermore, we highlight the architectural distinctions between the encoder and decoder modules. By the end of this chapter, the reader will have a comprehensive understanding of Transformer model architecture. Additionally, we present a taxonomy of Transformer variants to provide insight into the vast landscape of models that have emerged in recent years. Notable variants include BERT [2], RoBERTa [11], ALBERT [9], and DistilBERT [14], each addressing specific challenges in model efficiency and performance. Before dissecting the architecture that sparked the Transformer revolution, it is essential to understand the motivations and challenges that led to its development.

1.1 Transformer Architecture

This section provides an in-depth exploration of the Transformer architecture, which has revolutionized the field of natural language processing (NLP) and machine learning. We begin by discussing the limitations of traditional sequential models, such as Recurrent Neural Networks (RNNs), and how the Transformer model overcomes these constraints through parallelization and attention mechanisms. A comprehensive overview of the Transformer architecture is then presented, delving into key components such as self-attention, multi-head attention, the feed-forward layer, and positional encoding.

1.1.1 From Sequential Constraints to Parallelized Context: How Transformers Overcome RNN Limitations

Fundamental Limitations of RNNs

Recurrent Neural Networks (RNNs) process sequential data iteratively, updating a hidden state at each timestep based on the current input and prior hidden state. While this mimics human text consumption, it introduces critical bottlenecks:

- **Sequential Processing and Lack of Parallelization:** RNNs cannot parallelize computation, as each timestep depends on the output of the previous step. This sequential

dependency creates inefficiencies in training, particularly for long sequences, where hardware acceleration (e.g., GPUs/TPUs) cannot exploit parallelism.

Recurrent Neural Networks (RNN)

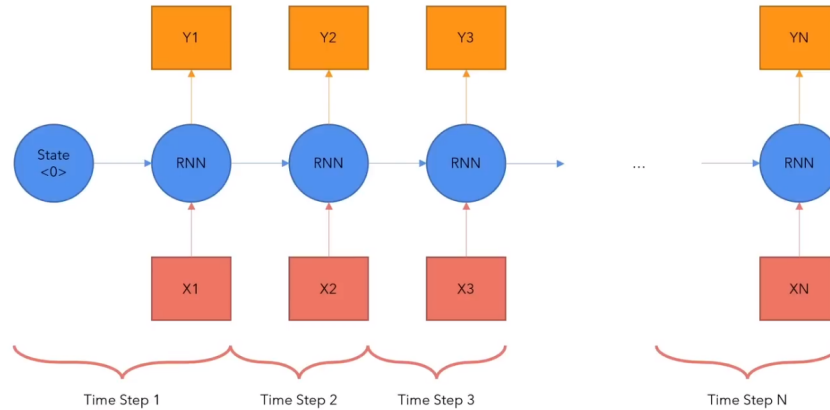


Figure 1.1: Sequential processing in RNNs.

- **Degradation of Long-Range Dependencies:** RNNs propagate contextual information through a chain of hidden states. Over long sequences, repeated nonlinear transformations cause *vanishing* or *exploding gradients*, impairing the model’s ability to retain distant dependencies. Transformers address this issue through self-attention, as demonstrated in [1], and further improvements in long-range dependency modeling have been achieved in models like Longformer [17]. Even advanced variants like LSTMs and GRUs, which regulate information flow via gating mechanisms, rely on sequential propagation, forcing distant tokens to traverse intermediate states. This ”daisy-chain” processing risks information loss or distortion.
- **Indirect Contextual Access:** Bidirectional RNNs partially mitigate positional bias by aggregating forward and backward hidden states. However, they still process sequences sequentially within each direction and combine outputs only after full traversal, limiting direct access to global context.

Transformers: Architectural Innovations

Transformers address RNN limitations through three interconnected innovations:

- **Self-Attention Mechanism and Parallelization:** The core innovation of transformers is *self-attention*, which computes pairwise relevance scores between all tokens in a sequence. This mechanism, first proposed in [1], allows transformers to model long-range dependencies efficiently by attending to all tokens simultaneously. For each token, the output is a weighted sum of embeddings from all other tokens, with weights reflecting contextual relevance. This allows **direct access** to distant positions without traversing intermediate steps, preserving long-range dependencies. Multi-head attention extends this by capturing diverse contextual relationships across multiple representation subspaces. The effectiveness of multi-head attention has been further analyzed in [15], which demonstrates that specialized heads often handle specific linguistic phenomena. Crucially, self-attention operates on all tokens simultaneously, enabling full parallelization during training and eliminating sequential bottlenecks.

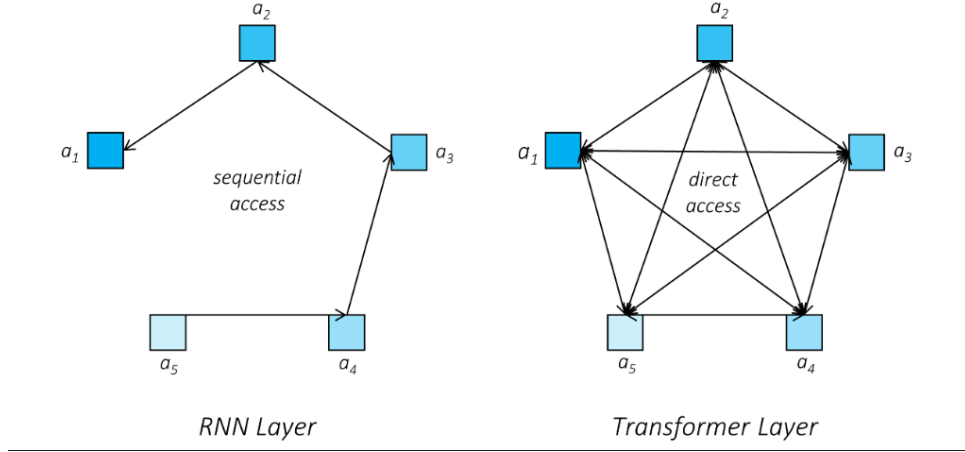


Figure 1.2: Self-attention mechanism in Transformers.

- Positional Encoding and Inherent Bidirectionality:** Unlike RNNs, which implicitly encode positional information through processing order, transformers explicitly inject positional data via *positional embeddings*. The use of positional embeddings, as described in [1], ensures that transformers retain sequence information without sacrificing parallelization. These learned or fixed vectors are added to token embeddings, preserving sequence order while maintaining parallelization. Furthermore, self-attention inherently incorporates bidirectional context by attending to all positions in a single pass. Unlike bidirectional RNNs, which require separate forward and backward processing, transformers holistically model relationships between tokens in all directions, enhancing contextual awareness without computational redundancy. This bidirectionality is a key feature of models like BERT [2], which leverages transformers to capture context from both directions simultaneously.
- Training Parallelism:** The transformer’s architecture enables simultaneous computation across the entire sequence during training. Unlike RNNs, which process tokens stepwise, transformers leverage parallel hardware (e.g., GPUs) to compute representations for all tokens at once. This maximizes computational efficiency, accelerates convergence, and scales effectively to large datasets. The parallelizable nature of transformers, as highlighted in [1], has made them the architecture of choice for large-scale language models like BERT [2] and RoBERTa [11].

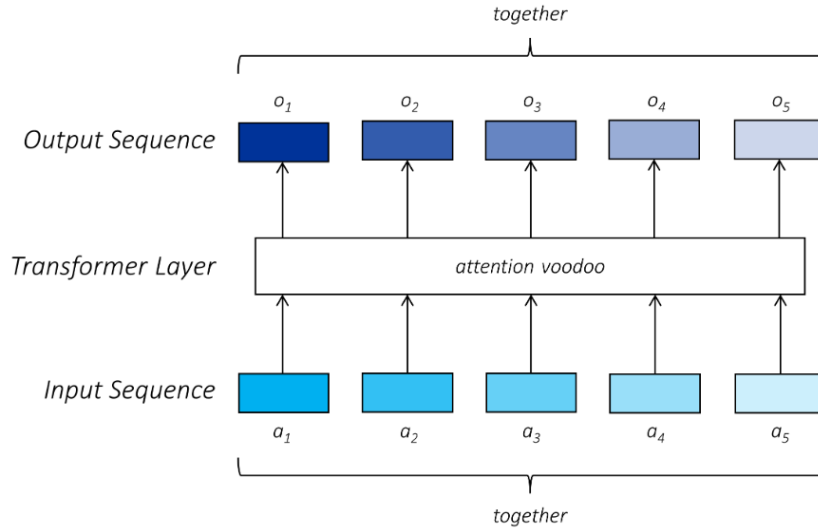


Figure 1.3: Training parallelism in Transformers.

Conclusion

Transformers surpass RNNs by replacing sequential processing with parallelizable self-attention, enabling efficient training and robust modeling of global dependencies. This architectural shift, introduced in [1], has been further refined in models like BERT [2] and RoBERTa [11], which leverage transformers for tasks requiring deep contextual understanding. These innovations direct contextual access through attention, explicit positional encoding, and inherent bidirectionality have rendered transformers the dominant architecture for tasks such as machine translation, text generation, and document summarization.

1.1.2 Overview of the Transformer Architecture

The original Transformer model is built upon an encoder-decoder architecture, which has been extensively used in sequence-to-sequence tasks such as machine translation. This architecture, introduced in [1], consists of two key components:

- **Encoder:** Transforms an input sequence of tokens into a sequence of embedding vectors, commonly referred to as the hidden state or contextual representation.
- **Decoder:** Utilizes the encoder's hidden state to iteratively generate an output sequence of tokens, one token at a time.

As illustrated in Figure 1.4, the encoder and decoder are composed of multiple building blocks:

- **Tokenization and Embedding:** The input text is tokenized and mapped to token embeddings. Since self-attention lacks inherent positional awareness, positional embeddings are incorporated to preserve the sequential nature of text. This approach, first proposed in [1], ensures that transformers retain sequence information without sacrificing parallelization.
- **Stacked Layers:** The encoder consists of multiple layers, akin to convolutional layers in computer vision, each responsible for progressively refining representations. The decoder follows a similar layered structure with additional mechanisms for attending to encoder outputs.

- **Autoregressive Decoding:** The decoder receives the previously generated tokens as input, alongside the encoder’s outputs, to predict the next token in the sequence. This iterative process continues until a special end-of-sequence (EOS) token is encountered or a maximum sequence length is reached.

While initially designed for sequence-to-sequence tasks, the Transformer architecture has since evolved, giving rise to three major model categories:

- **Encoder-only models:** These generate rich numerical representations of text, making them well suited for tasks like text classification and named entity recognition. Examples include BERT [2], RoBERTa [11], and DistilBERT [14], which leverage bidirectional attention.
- **Decoder-only models:** Designed for autoregressive generation, these models predict the most probable next word based on left-context inputs. The GPT family, as described in [3], falls under this category, utilizing causal attention mechanisms.
- **Encoder-decoder models:** Tailored for complex sequence mappings, these models excel in translation and summarization tasks. Examples include BART and T5, which integrate encoder and decoder components for robust text generation.

Although these classifications provide a general framework, the boundaries between model applications are often fluid. For instance, decoder-only models like GPT can be adapted for translation, while encoder-only models like BERT can be used for summarization. Having established a high-level understanding of Transformer architecture, we now proceed to a detailed examination of the encoder’s inner workings.

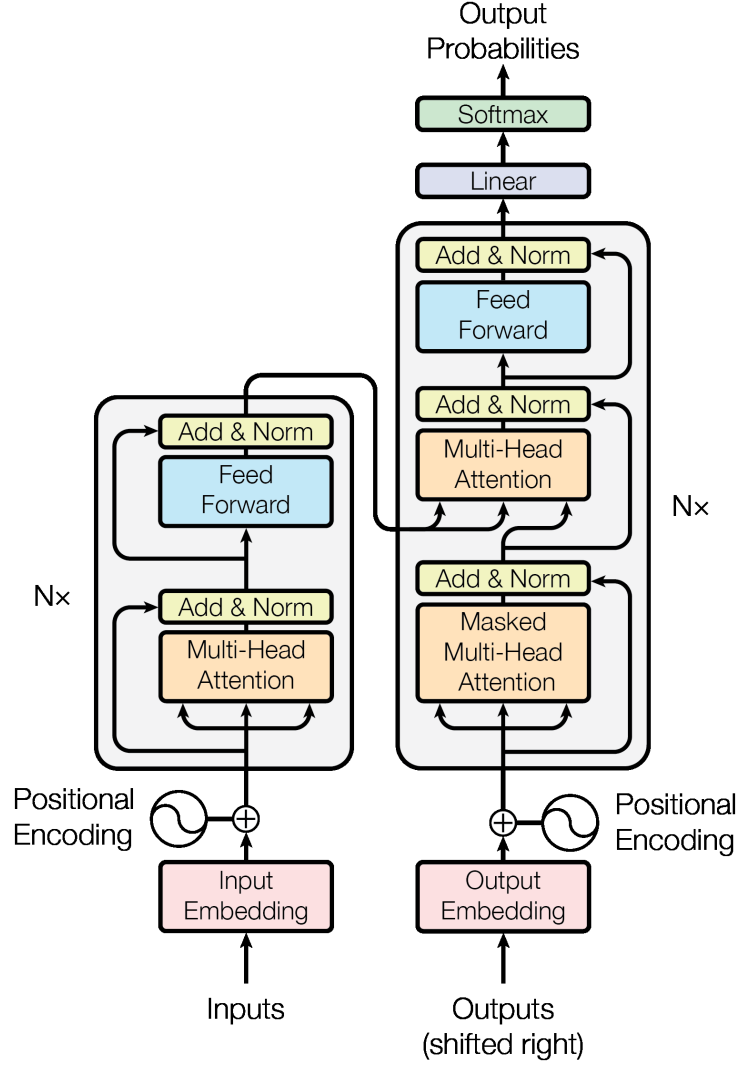


Figure 1.4: Overview of the Transformer architecture, highlighting the encoder-decoder structure and key components.

1.1.3 Self-Attention

Building an Intuitive Understanding of the Self-Attention Mechanism

Self-attention is a fundamental mechanism that allows neural networks to dynamically assign varying levels of importance, or "attention," to different elements within a sequence. In the context of textual data, these elements correspond to token embeddings vector representations of individual tokens within a given sequence. For instance, in BERT [2], each token is encoded as a 768-dimensional vector. The term "self" in self-attention highlights that attention weights are computed across all hidden states within the same sequence, such as all hidden states within the encoder. This is distinct from traditional attention mechanisms in recurrent models, where attention is computed between encoder hidden states and the decoder's hidden state at each decoding step.

The central idea of self-attention, as introduced in [1], is to leverage the entire sequence to compute a weighted sum of each token embedding. Instead of using static embeddings, the self-attention mechanism refines token representations by incorporating contextual information. Mathematically, given a sequence of token embeddings x_1, \dots, x_n , self-attention generates a new sequence of embeddings x'_1, \dots, x'_n , where each transformed embedding x'_i is computed as a weighted sum of all token embeddings in the sequence:

$$x'_i = \sum_{j=1}^n w_{ji} x_j$$

To understand the intuition behind self-attention, consider the word "flies" in isolation it could refer to insects or be the present tense of the verb "fly." However, when placed in a sentence like "Time flies like an arrow," the surrounding words provide crucial context, clarifying that "flies" is a verb. The self-attention mechanism captures such contextual dependencies by assigning greater weight w_{ji} to the most relevant tokens, such as "time" and "arrow," while diminishing the influence of less relevant ones. A visual representation of this process illustrates how self-attention allows different interpretations of the same token depending on the context in which it appears.

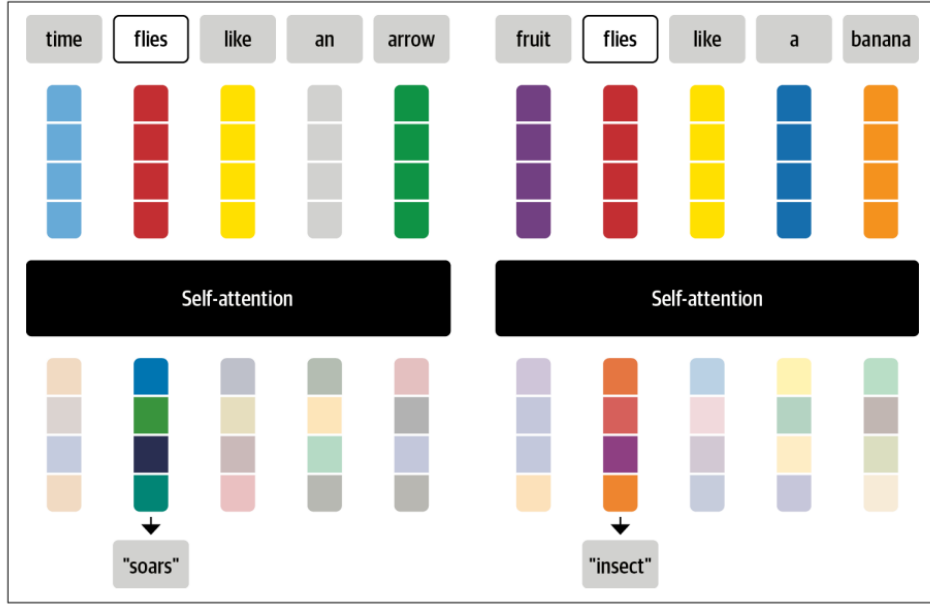


Figure 1.5: Visualization of self-attention weights for the word "flies" in different contexts.

Detailed Computation of Self-Attention

The self-attention mechanism consists of several sequential steps that transform input embeddings into context-aware representations. These steps, as introduced in [1], can be understood as follows:

- **Generating Query, Key, and Value Vectors:** Each word in the input sequence is first mapped to three distinct vectors Query (Q), Key (K), and Value (V) which are derived by multiplying the input word embeddings with learnable weight matrices. Given an input word embedding vector x_i , the transformation can be expressed as:

$$q_i = W_Q x_i, \quad k_i = W_K x_i, \quad v_i = W_V x_i$$

where W_Q , W_K , and W_V are learnable weight matrices, and q_i , k_i , and v_i are the corresponding query, key, and value vectors.

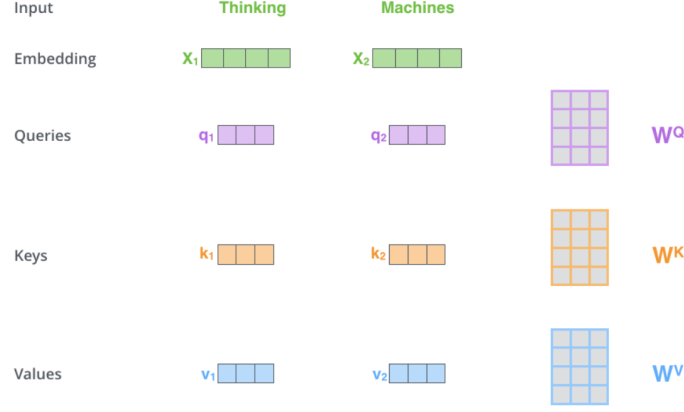


Figure 1.6: Generation of Query, Key, and Value vectors.

- **Computing Attention Scores:** The attention score for each word is determined by computing the dot product between its query vector and the key vectors of all words in the sequence. This measures the relevance of each word to the current word:

$$\text{score}_{ij} = q_i \cdot k_j$$

where score_{ij} represents the attention score between the i^{th} and j^{th} words.

- **Scaling the Scores:** To stabilize gradients and improve learning, the scores are scaled by the square root of the dimension of the key vectors, denoted as d_k :

$$\text{score}'_{ij} = \frac{\text{score}_{ij}}{\sqrt{d_k}}$$

where d_k is typically set to 64, as per the original Transformer implementation [1].

- **Applying Softmax:** The scaled scores are then passed through a softmax function to normalize them into probability distributions:

$$\alpha_{ij} = \frac{e^{\text{score}'_{ij}}}{\sum_j e^{\text{score}'_{ij}}}$$

These normalized attention scores determine the influence of each word on the final representation of the current word.

- **Computing Weighted Sum of Value Vectors:** The output representation for each word is obtained by multiplying the value vectors with the attention weights and summing them:

$$z_i = \sum_j \alpha_{ij} v_j$$

This ensures that the model aggregates information from all relevant words, effectively encoding context-dependent representations.

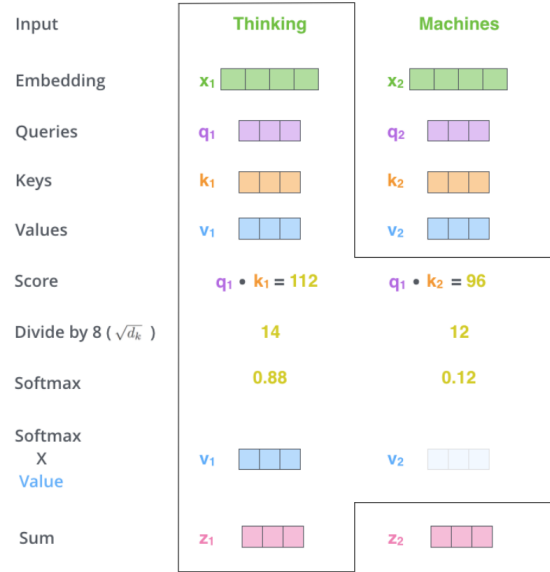


Figure 1.7: Computation of Self-Attention.

Matrix-Based Implementation of Self-Attention

To enhance computational efficiency, self-attention is implemented in matrix form, allowing parallelized computation. The steps are as follows:

- **Constructing Query, Key, and Value Matrices:** Instead of processing each word individually, we represent all input embeddings as a matrix X , and compute the Q , K , and V matrices as follows:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

where X is the matrix containing word embeddings, and W_Q , W_K , and W_V are learnable weight matrices.

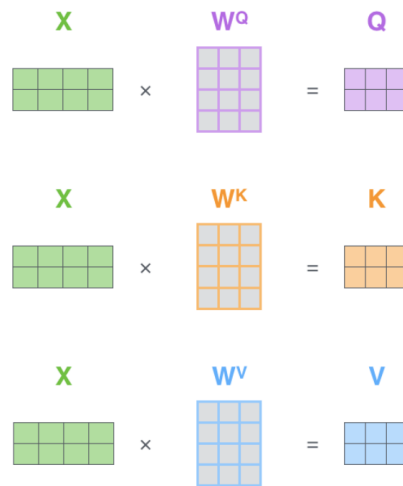


Figure 1.8: Matrix-based computation of Query, Key, and Value matrices.

- **Computing Scaled Attention Scores and Applying Softmax and Weighting**

Values: Using matrix multiplication, we compute all pairwise attention scores simultaneously:

$$S = \frac{QK^T}{\sqrt{d_k}}$$

where S is the attention score matrix. The softmax function is then applied row-wise to obtain the attention weight matrix A :

$$A = \text{softmax}(S)$$

Finally, the output of the self-attention mechanism is computed as:

$$Z = AV$$

This results in a refined representation Z , which integrates contextually relevant information from the entire sequence.

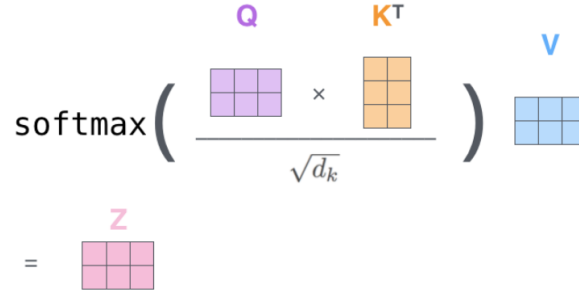


Figure 1.9: Matrix-based computation of self-attention output.

This matrix formulation significantly accelerates computation, making the Transformer architecture highly efficient and scalable for various NLP tasks [1].

1.1.4 Multi-head Attention

In a basic self-attention mechanism, attention scores and weights are computed directly from token embeddings. However, the self-attention layer introduces additional complexity by applying three independent linear transformations to each embedding, generating the **Query (Q)**, **Key (K)**, and **Value (V)** vectors. These learned projections allow the model to capture different relationships within the input sequence, with each projection parameterized by distinct weight matrices. To further enhance the model's ability to capture diverse patterns, **multi-head attention** is introduced. Instead of relying on a single attention mechanism, multiple independent sets of Q, K, and V matrices each representing a separate **attention head** are used. Each head learns different aspects of token relationships, enabling the model to focus on multiple linguistic structures simultaneously. The multi-head attention mechanism, as described in [1], is formally expressed as:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \text{head}_2, \dots, \text{head}_h)W_O$$

where each head is computed as:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Here, W_i^Q , W_i^K , and W_i^V are learnable weight matrices specific to each head, and W_O is a learned projection matrix used to condense the multiple heads into a single output representation. The need for multiple attention heads arises from the limitations of single-head attention, where a single softmax operation tends to emphasize one dominant aspect of similarity, potentially restricting the model's ability to capture nuanced relationships. By employing multiple heads, the model can simultaneously focus on different linguistic features, such as **subject-verb relationships** or **modifiers and adjectives**, akin to how CNNs use different filters to detect distinct features in images.

Since multi-head attention produces multiple output matrices one per head this poses a challenge for subsequent processing layers, such as the feed-forward network, which expects a single matrix. To address this, the outputs of all attention heads are concatenated and projected back into the expected dimensionality using a learned weight matrix W_O :

$$O = \text{Concat}(\text{head}_1, \text{head}_2, \dots, \text{head}_h)W_O$$

This step ensures the model retains the benefits of multiple attention heads while maintaining a consistent output shape for the next layer. Overall, multi-head attention enhances the expressiveness of self-attention by enabling the model to learn and capture diverse relationships within a sequence, making it a key component of Transformer architectures for tasks requiring complex contextual understanding.

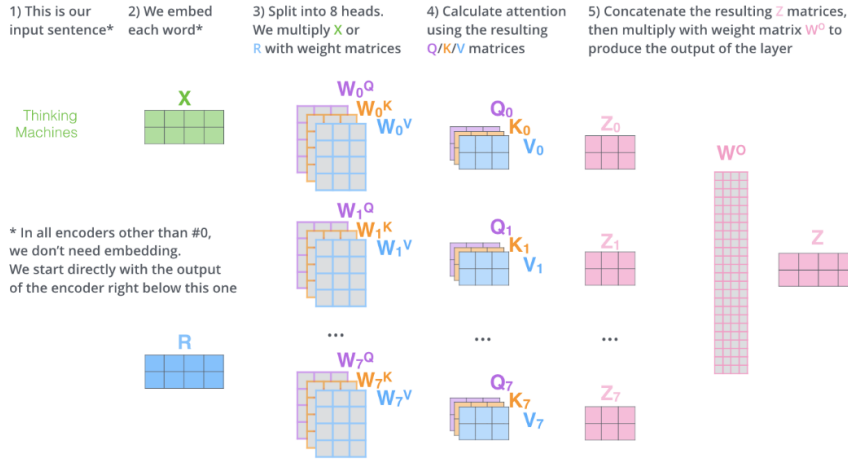


Figure 1.10: Visualization of multi-head attention, showing how multiple attention heads capture different linguistic features.

1.1.5 The Feed-Forward Layer

The feed-forward sublayer in the Transformer architecture plays a crucial role in refining the representations produced by the multi-head attention mechanism. While attention mechanisms focus on capturing relationships between tokens, the feed-forward layer independently transforms each token's representation, enhancing the model's ability to encode complex features. Unlike recurrent or convolutional layers that process sequential data holistically, the feed-forward sublayer operates position-wise, applying the same transformation to each token embedding independently without considering relationships between different positions in the sequence. This allows the Transformer to maintain parallelization while ensuring each token undergoes a learned transformation that enriches its representation. Due to this independence, this layer is sometimes referred to as a position-wise feed-forward network (FFN) or, in some contexts, a one-dimensional convolution with a kernel size of one.

The feed-forward sublayer consists of a two-layer fully connected neural network with an intermediate non-linearity:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

where:

- x is the input embedding from the multi-head attention mechanism.
- W_1 and W_2 are learned weight matrices.
- b_1 and b_2 are bias terms.
- The activation function (commonly GELU Gaussian Error Linear Unit) introduces non-linearity, enhancing the model's expressiveness.

A common design choice, as suggested in the original Transformer paper [1], is to set the hidden dimension of the first feed-forward layer to be four times the size of the input embeddings. This expansion significantly increases the network's capacity for learning complex transformations before reducing it back to the original embedding size.

The feed-forward sublayer is where most of the model's memorization and capacity reside, as it allows each token representation to be refined based on learned patterns. This layer is also one of the primary components scaled up in larger Transformer models. Increasing the hidden dimension of the feed-forward network leads to greater model expressiveness, enabling the Transformer to capture more intricate language patterns. By applying the feed-forward transformation independently to each token, the Transformer maintains its computational efficiency and parallelization capabilities, making it well suited for large-scale processing tasks, such as training on massive text corpora, without incurring the sequential dependencies of recurrent architectures.

1.1.6 Positional Encoding

Transformer-based models lack inherent sequential information due to their non-recurrent architecture. Unlike recurrent neural networks (RNNs), which process data sequentially, Transformers process entire sequences in parallel. To incorporate word order into the model, **positional encodings** are introduced, which provide each token with explicit positional information. This approach, first proposed in [1], ensures that the model can distinguish between tokens based on their position in the sequence.

The Need for Positional Encoding

Since self-attention mechanisms treat input tokens independently, they do not inherently capture the order of words in a sequence. Without an explicit mechanism to convey positional information, a Transformer would be unable to differentiate between sentences like "*The cat sat on the mat.*" and "*The mat sat on the cat.*" To resolve this, **positional encodings** are added to the token embeddings before they are fed into the model.

Mathematical Formulation

The positional encoding is typically implemented using sinusoidal functions, which allow the model to generalize to longer sequences beyond those seen during training. The encoding for position pos and dimension i is defined as follows:

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

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

where d is the embedding dimension. The choice of sinusoidal functions ensures that nearby positions have similar encodings while still allowing distant positions to be distinguishable. Additionally, these functions provide a continuous, smooth representation of position, facilitating extrapolation to sequences longer than those seen during training.

Interpretation of the Encoding

The sinusoidal positional encoding exhibits a hierarchical structure where lower-frequency components capture global position information, while higher-frequency components capture fine-grained details. The figure below visualizes these sinusoidal patterns, where each row corresponds to a different position in the sequence, and colors represent the encoding values.

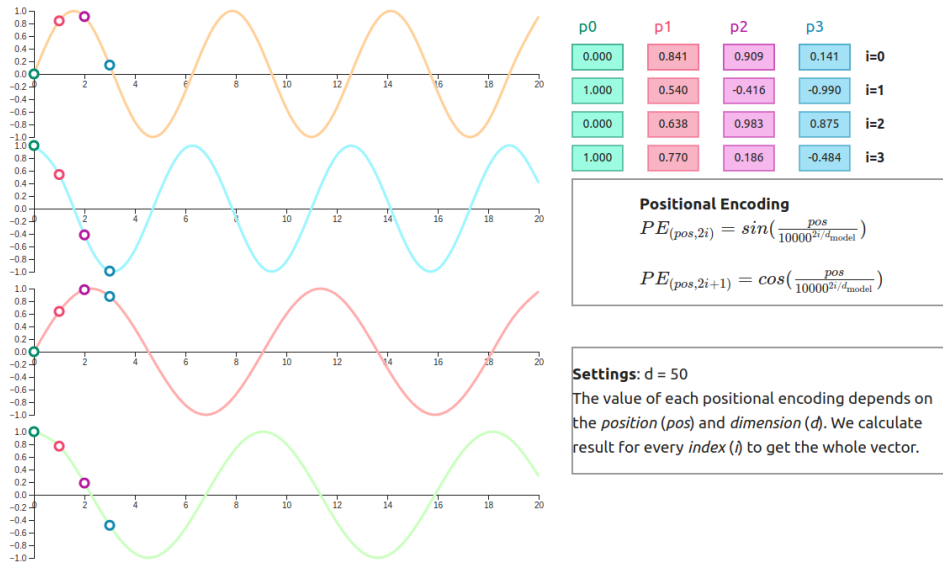


Figure 1.11: Visualization of sinusoidal positional encodings.

The resulting positional encoding matrix is added element-wise to the token embeddings:

$$X' = X + PE$$

where X represents the input token embeddings, and PE represents the positional encoding matrix. This operation injects positional information into the input representations before they are passed to the attention layers.

Original sentence	YOUR	CAT	IS	A	LOVELY	CAT
Embedding (vector of size 512)	992.207 5405.840 1853.448 ... 1.458 2871.529	171.411 3276.350 9192.819 ... 3633.421 8390.473	421.459 1304.051 0.565 ... 7679.805 4506.025	776.562 5547.288 58.942 ... 2716.194 5118.949	8422.693 4315.080 9358.778 ... 2141.081 735.147	171.411 3276.350 9192.819 ... 3633.421 8390.473
Position Embedding (vector of size 512). Only computed once and reused for every sentence during training and inference.	+	+	+	+	+	+
	1864.088 8080.133 2620.399 ... 9386.405 3120.159	1281.458 7902.890 912.970 ... 3821.102 1659.217 7018.620
Encoder Input (vector of size 512)	1835.479 11356.483 11813.218 ... 13019.826 11510.822	1422.869 11176.24 10105.789 ... 5292.438 15409.093

Figure 1.12: Addition of positional encodings to token embeddings.

Visualizing the Depth-Position Interaction

To understand how positional information propagates through Transformer layers, we can examine how the encoding evolves at different model depths. The figure below presents a heatmap visualization, where the y-axis represents position indices, and the x-axis represents depth in the network. The color scale indicates encoding values, revealing how position information is transformed layer by layer.

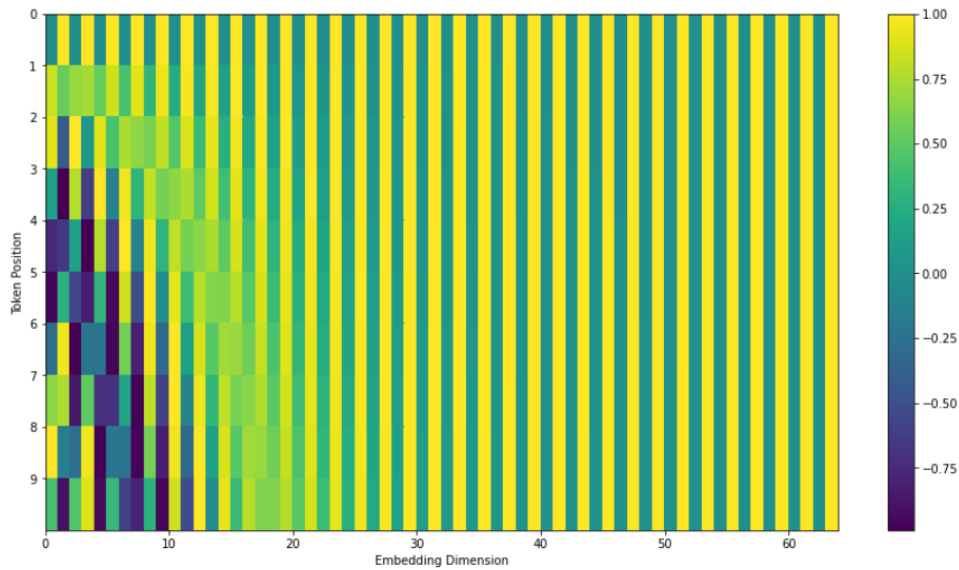


Figure 1.13: Heatmap visualization of positional encoding across Transformer layers.

Alternative Approaches

While sinusoidal encodings are widely used in standard Transformers, alternative approaches such as learned positional embeddings and rotary position embeddings (RoPE) have been proposed to improve positional representations. These methods allow models to adapt positional encoding dynamically based on training data rather than relying on fixed mathematical formulations. For example, learned positional embeddings have been employed in models like BERT [2] and RoBERTa [11], offering greater flexibility in capturing positional relationships.

1.1.7 Add & Normalize Layer

The **Add & Normalize** component is a critical feature of the Transformer architecture, designed to ensure stable training and effective gradient flow. It consists of **skip (or residual) connections** and **layer normalization**, which together help mitigate the vanishing/exploding gradient problem and enhance learning efficiency. These techniques, introduced in [1], have become standard in modern Transformer based models.

Skip Connections: Preserving Information Flow

Skip connections, also known as **residual connections**, allow the input of a layer to be **directly added** to its output before further processing. Mathematically, for a given layer's transformation function $f(x)$, the output is:

$$\text{Output} = f(x) + x$$

where:

- x is the input from the previous layer.
- $f(x)$ represents the transformation applied by the current layer (such as attention or feed-forward operations).

By incorporating the original input x , skip connections help preserve information from earlier layers, making it easier for the model to learn identity mappings when necessary. This technique is crucial for stabilizing deep networks, preventing the loss of gradient signal as information propagates through multiple layers.

Layer Normalization: Ensuring Stable Representations

Layer normalization is applied after the residual addition step. Unlike batch normalization, which normalizes across the batch dimension, **layer normalization** normalizes each token's hidden representation **independently**, ensuring that all elements in a single embedding vector have zero mean and unit variance. The transformation is defined as:

$$\text{LN}(x) = \frac{x - \mu}{\sigma} \gamma + \beta$$

where:

- μ and σ are the mean and standard deviation computed across the embedding dimensions.
- γ and β are learnable scale and shift parameters that allow the model to adapt normalization dynamically.

This normalization step helps stabilize training by reducing internal covariate shift, which refers to sudden changes in the distribution of activations as they pass through different layers. The placement of layer normalization within the Transformer block significantly impacts training dynamics. Two common configurations exist:

1. **Post-Layer Normalization (Original Transformer)** Layer normalization is applied *after* the residual addition:

$$\text{Output} = \text{LN}(x + f(x))$$

This approach was used in the original Transformer paper [1] but is known to cause instability in training. Without careful learning rate scheduling (e.g., **learning rate warm-up**), gradients can explode, making optimization difficult.

2. **Pre-Layer Normalization (Modern Transformers)** Layer normalization is applied *before* the transformation:

$$\text{Output} = x + f(\text{LN}(x))$$

This configuration is now more commonly used in Transformer based architectures, as it leads to better stability and does not require learning rate warm-up. Pre-normalization ensures that each layer receives consistently scaled inputs, reducing the likelihood of gradient divergence.

The combination of **skip connections** and **layer normalization** helps Transformers efficiently train deep architectures without suffering from vanishing or exploding gradients. **Skip connections** improve gradient flow, while **layer normalization** ensures stable feature distributions, making Transformers robust for large scale learning tasks.

This Add & Normalize mechanism, though simple, plays a fundamental role in enabling deep Transformers to scale effectively while maintaining stability in optimization.

1.1.8 The Final Linear and Softmax Layers (Decoder)

The final component of the Transformer’s decoder is the **linear layer**, which serves as a crucial link between the model’s learned representations and its output vocabulary. This layer, followed by a **softmax function**, translates the high-dimensional hidden state vectors into a probability distribution over the vocabulary, ultimately enabling word prediction.

Mapping Hidden Representations to the Vocabulary Space

The decoder produces a sequence of **contextualized hidden state vectors**, each corresponding to a token in the output sequence. However, these vectors exist in an abstract, high-dimensional space and must be mapped to a concrete word representation. This is achieved using a **fully connected linear transformation**, mathematically defined as follows:

$$z = W_{\text{out}}h + b$$

where:

- h represents the hidden state vector generated by the decoder.
- W_{out} is a learnable weight matrix that projects the hidden states onto a **logits vector** of size equal to the vocabulary size.
- b is a bias term that allows additional flexibility in the mapping process.

If the model has been trained with a vocabulary containing V words, then the logits vector will have V **dimensions**, where each value corresponds to a score indicating the likelihood of selecting a specific word as the output at that time step.

Softmax: Transforming Logits into Probabilities

The raw scores (logits) produced by the linear layer are not directly interpretable as probabilities. To convert these scores into a valid probability distribution, the **softmax function** is applied:

$$P(w_i) = \frac{e^{z_i}}{\sum_{j=1}^V e^{z_j}}$$

where:

- z_i represents the logit score for word i .

- V is the vocabulary size.
- The softmax function ensures that all probabilities are **positive** and sum to **1**, making the output interpretable as a probability distribution over possible words.

Once the probability distribution is computed, the model selects the word with the highest probability as the predicted token for that time step.

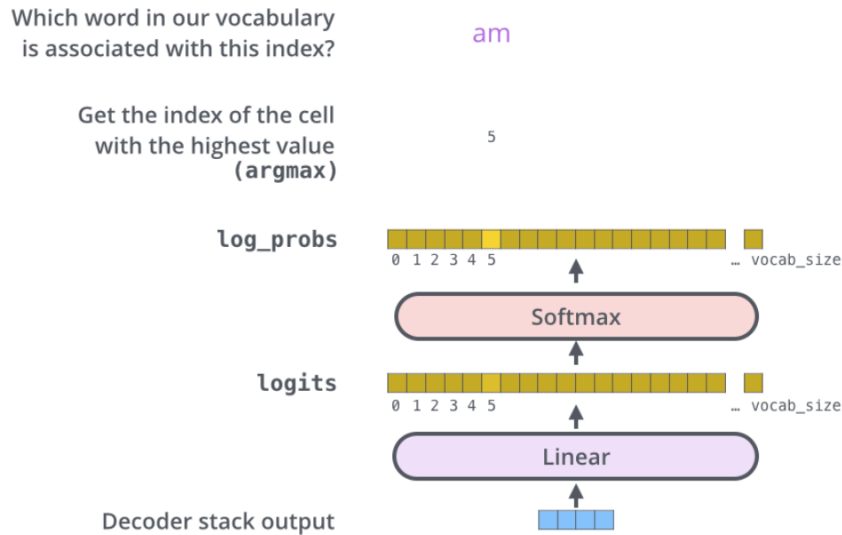


Figure 1.14: The Final Linear and Softmax Layers.

1.1.9 Encoder-Decoder Architecture and Information Flow in Transformers

Encoder: Contextual Representation Generation

The encoder transforms an input sequence of tokens into a high-dimensional, contextually enriched representation. It consists of a stack of identical layers, each containing two main components: the **multi-head self-attention mechanism** and the **position-wise feed-forward network (FFN)**. The self-attention mechanism computes relationships between all tokens in the sequence, allowing the model to capture long-range dependencies and contextual information. The FFN applies non-linear transformations to refine these representations. Residual connections and layer normalization are used to stabilize training and improve gradient flow. As the input sequence passes through each layer, the encoder progressively refines the token representations, ultimately producing a final output \mathbf{z} that encapsulates the global context of the input sequence.

Decoder: Auto-Regressive Generation with Contextual Guidance

The decoder generates the output sequence token by token, conditioned on both the encoder's output \mathbf{z} and the previously generated tokens. It also consists of a stack of identical layers, but with three key components: **masked multi-head self-attention**, **encoder-decoder multi-head attention**, and the **position-wise FFN**. The masked self-attention ensures that each token is generated based only on the preceding tokens, preserving the auto-regressive property. The encoder-decoder attention allows the decoder to focus on relevant parts of the encoder's output \mathbf{z} , enabling it to align input and output sequences effectively. The FFN further refines the representations. Like the encoder, the decoder uses residual connections and layer normalization to stabilize training. During inference, the decoder iteratively predicts tokens, appending each new token to the output sequence until completion.

Encoder-Decoder Synergy: Cross-Attention and Information Flow

The encoder and decoder work together through the **encoder-decoder attention mechanism**, which is the core of their interaction. The encoder processes the input sequence in parallel, producing a contextualized representation \mathbf{z} . The decoder, during its auto-regressive generation, uses this representation to guide its predictions. At each decoding step, the encoder-decoder attention sub-layer computes attention weights between the decoder's current state (queries) and the encoder's output \mathbf{z} (keys and values). This allows the decoder to dynamically retrieve relevant information from the input sequence, ensuring that the generated output is contextually aligned with the input. For example, in translation tasks, the decoder can focus on specific input tokens (e.g., a subject or verb) when generating the corresponding output tokens. This interplay between the encoder and decoder enables the Transformer to handle complex sequence-to-sequence tasks efficiently and effectively.

1.2 The Transformer Tree of Life

Over time, each of the three main transformer architectures has undergone an evolution of its own. This is illustrated in the Figure below, which shows a few of the most prominent models and their descendants.

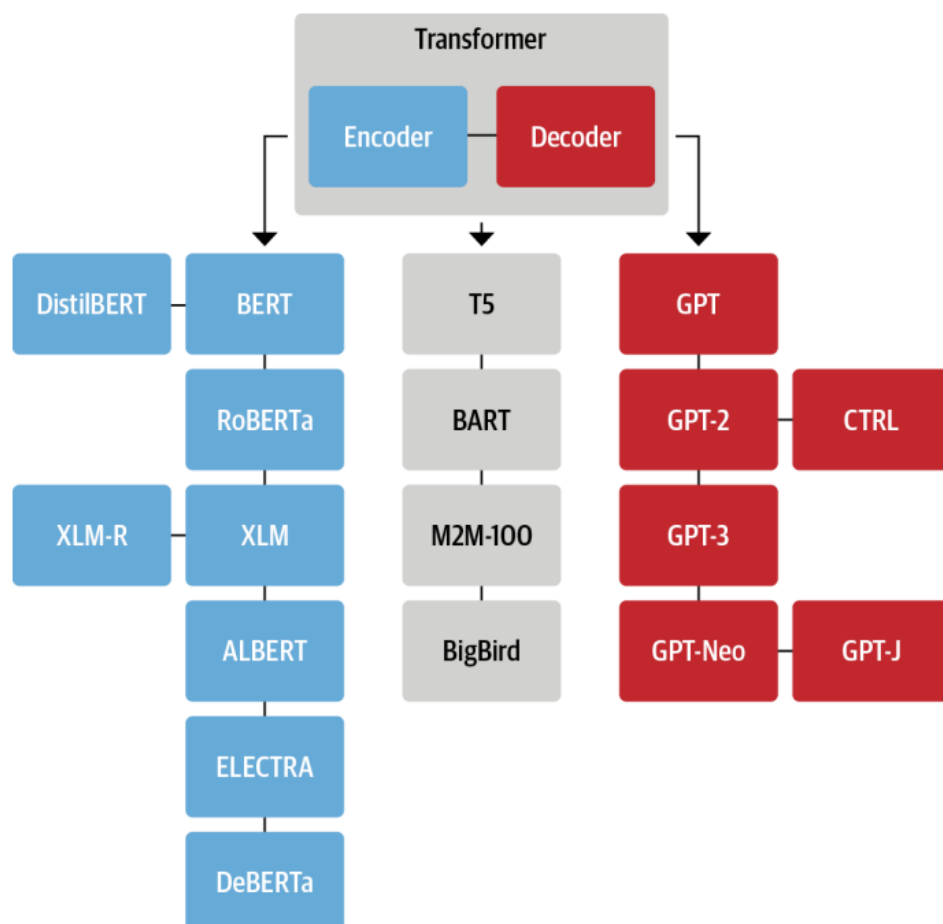


Figure 1.15: An overview of some of the most prominent transformer architectures

This family tree by no means provides a complete overview of all existing transformer architectures; it simply highlights a few of the architectural milestones. Having covered the original

Transformer architecture in depth in previous sections, we now examine some of the key descendants, starting with the encoder branch.

1.2.1 The Encoder Branch

The first encoder-only model based on the Transformer architecture was BERT. At the time it was published, it outperformed all state-of-the-art models on the popular GLUE benchmark [4], which measures natural language understanding (NLU) across several tasks of varying difficulty. Subsequently, the pretraining objective and the architecture of BERT have been adapted to further improve performance.

Encoder-only models still dominate research and industry for NLU tasks such as text classification, named entity recognition, and question answering. Let’s examine the BERT model and its variants:

BERT

BERT is pretrained with the dual objectives of predicting masked tokens in texts and determining if one text passage is likely to follow another [2]. The former task is called masked language modeling (MLM) and the latter next sentence prediction (NSP).

DistilBERT

Although BERT delivers strong results, its size can make it challenging to deploy in environments requiring low latencies. By using knowledge distillation during pretraining, DistilBERT achieves 97% of BERT’s performance while using 40% less memory and being 60% faster [14].

RoBERTa

A study following BERT’s release revealed that its performance can be further improved by modifying the pretraining scheme. RoBERTa is trained longer, on larger batches with more training data, and drops the NSP task [11]. Together, these changes significantly improve its performance compared to the original BERT model.

XLM

Several pretraining objectives for building multilingual models were explored in the cross-lingual language model (XLM) [8], including autoregressive language modeling from GPT-like models and MLM from BERT. Additionally, the authors introduced translation language modeling (TLM), an extension of MLM to multiple language inputs. These pretraining tasks achieved state-of-the-art results on several multilingual NLU benchmarks and translation tasks.

XLM-RoBERTa

Following XLM and RoBERTa, the XLM-RoBERTa (XLM-R) model advances multilingual pretraining by massively upscaling the training data [6]. Using the Common Crawl corpus, the researchers created a dataset with 2.5 terabytes of text and trained an encoder with MLM on this dataset. This approach significantly outperforms XLM and multilingual BERT variants, especially on low-resource languages.

ALBERT

ALBERT introduced three changes to enhance encoder architecture efficiency [9]. First, it decouples token embedding dimension from hidden dimension, allowing smaller embedding dimensions and saving parameters. Second, all layers share parameters, decreasing effective parameter

count further. Finally, NSP is replaced with sentence-ordering prediction. These changes enable training of larger models with fewer parameters while achieving superior NLU task performance.

ELECTRA

Addressing a limitation of standard MLM pretraining where only masked token representations are updated, ELECTRA employs a two-model approach [19]: a small model works as a standard masked language model predicting masked tokens, while a second model (the discriminator) predicts which tokens were originally masked. This makes training 30 times more efficient, as the discriminator performs binary classification for every token.

DeBERTa

DeBERTa introduces two architectural innovations [21]. First, each token is represented by two vectors one for content, another for relative position disentangling content from position and enabling better modeling of token pair dependencies. Second, absolute position embeddings are added before the softmax layer of the token decoding head. DeBERTa was the first model (as an ensemble) to exceed human baseline performance on the SuperGLUE benchmark [16].

1.2.2 The Decoder Branch

Progress on transformer decoder models has been largely driven by OpenAI. These models excel at predicting the next word in a sequence and are primarily used for text generation tasks. Their advancement has been fueled by larger datasets and scaling to increasingly larger model sizes:

GPT

GPT combined two key NLP innovations [3]: the efficient transformer decoder architecture and transfer learning. The model was pretrained by predicting the next word based on previous ones, trained on the BookCorpus, achieving impressive results on downstream classification tasks.

GPT-2

Building on GPT’s success, the original model and training set were upscaled to produce GPT-2 [12], capable of generating long sequences of coherent text. Due to potential misuse concerns, the model was released gradually, with smaller models published before the full version.

CTRL

While models like GPT-2 can continue input sequences (prompts), users have limited control over generation style. The Conditional Transformer Language (CTRL) model addresses this by incorporating ”control tokens” at sequence beginnings [7], enabling style control and diverse text generation.

GPT-3

Following scaling from GPT to GPT-2, analysis revealed power laws governing relationships between compute, dataset size, model size, and language model performance [22]. GPT-2 was subsequently upscaled by a factor of 100 to create GPT-3 [18] with 175 billion parameters. Beyond generating realistic text, the model demonstrates few-shot learning capabilities across novel tasks.

1.2.3 The Encoder-Decoder Branch

While single encoder or decoder stacks have become common, several encoder-decoder Transformer variants offer novel applications across both NLU and NLG domains:

T5

The T5 model unifies NLU and NLG tasks by converting them to text-to-text tasks [13]. All tasks are framed as sequence-to-sequence problems, naturally suited to encoder-decoder architectures. For classification, the text serves as encoder input while the decoder generates the label as text rather than a class. Using the large C4 dataset, T5 is pretrained with masked language modeling and SuperGLUE tasks translated to text-to-text format. The largest 11 billion parameter model achieved state-of-the-art results across multiple benchmarks.

BART

BART combines BERT and GPT pretraining procedures within the encoder-decoder architecture [10]. Input sequences undergo various transformations (masking, sentence permutation, token deletion, document rotation) before being passed through the encoder, with the decoder reconstructing original texts. This versatility enables both NLU and NLG tasks with state-of-the-art performance.

M2M-100

Conventional translation models are built for single language pairs and directions, limiting scalability. M2M-100 is the first model capable of translating between any of 100 languages [20], providing high-quality translations between rare and underrepresented languages. The model uses prefix tokens to indicate source and target languages.

BigBird

A primary limitation of transformer models is maximum context size due to attention mechanism’s quadratic memory requirements. BigBird addresses this through sparse attention that scales linearly [23], expanding context from typical 512 tokens in BERT models to 4,096 tokens. This particularly benefits tasks requiring long dependencies, such as text summarization.

1.3 Conclusion

In conclusion, the Transformer architecture has significantly advanced NLP by addressing the inherent limitations of previous models, offering parallelization and efficient handling of long-range dependencies. The components discussed in this chapter form the foundation for many state-of-the-art models today, highlighting the importance of self-attention, multi-head attention, and positional encoding in improving performance. Understanding the Transformer’s architecture is crucial for exploring its applications and innovations in modern machine learning research, including the modifications and advancements in attention mechanisms that will be examined in subsequent chapters.

Chapter 2

State of the Art in Attention Mechanisms

The Transformer architecture, driven by the power of the self-attention mechanism, has revolutionized the field of deep learning, particularly in natural language processing and computer vision. However, the original attention mechanism introduced in "Attention is All You Need" scales quadratically with sequence length, posing challenges for long sequences and large-scale models. As the demand for more efficient and scalable models has grown, numerous innovations have emerged to improve attention's computational and memory efficiency without sacrificing model quality. In this chapter, we explore several state-of-the-art advances in attention mechanisms that aim to address these challenges. We present and analyze Linear Attention, Sparse Attention, Flash Attention, and Sliding Window Attention each offering unique solutions to the bottlenecks of standard attention. By understanding the core ideas, motivations, and trade-offs behind these methods, we gain deeper insight into how attention mechanisms continue to evolve to meet the demands of modern deep learning tasks.

2.1 Linear Attention: A Kernel-Based Approach to Efficient Sequence Modeling

Linear attention represents a fundamental advancement in transformer architecture, addressing the quadratic complexity bottleneck that limits the scalability of standard attention mechanisms [1]. By reformulating attention through kernel methods, linear attention achieves $\mathcal{O}(T)$ complexity while preserving the essential modeling capabilities of the attention mechanism. This exposition presents a rigorous mathematical treatment of linear attention, emphasizing its theoretical foundations and computational advantages.

2.1.1 Standard Attention: The Quadratic Bottleneck

In the standard transformer, self-attention computes pairwise interactions between all tokens in a sequence [1]. For a sequence of length T with embedding dimension d , the attention output for token i is given by:

$$\text{Attention}(Q, K, V)_i = \frac{\sum_{j=1}^T e^{q_i^\top k_j} v_j}{\sum_{n=1}^T e^{q_i^\top k_n}} \quad (2.1)$$

where $q_i \in \mathbb{R}^d$ is the query vector for token i , $k_j \in \mathbb{R}^d$ is the key vector for token j , and $v_j \in \mathbb{R}^{d_v}$ is the corresponding value vector.

Computing this operation requires $\mathcal{O}(T^2d)$ time complexity and $\mathcal{O}(T^2)$ memory for storing the attention matrix. This quadratic scaling renders standard attention prohibitively expensive for long sequences, motivating the development of more efficient alternatives [17].

2.1.2 Kernel-Based Reformulation

The core insight behind linear attention is that the softmax attention mechanism can be reinterpreted as a normalized kernel function [15]. Specifically, the softmax weight between query i and key j can be expressed as:

$$\alpha_{ij} = \frac{e^{q_i^\top k_j}}{\sum_{n=1}^T e^{q_i^\top k_n}} \quad (2.2)$$

This expression can be generalized as a kernel similarity:

$$\alpha_{ij} = \frac{K(q_i, k_j)}{\sum_{n=1}^T K(q_i, k_n)} \quad (2.3)$$

where $K(\cdot, \cdot)$ is a kernel function measuring similarity between vectors.

2.1.3 Feature Map Decomposition

The critical mathematical insight enabling linear complexity is the approximation of the kernel function using decomposable feature maps [17]. For certain kernels, there exist transformations $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^{d'}$ such that:

$$K(q_i, k_j) \approx \phi(q_i)^\top \phi(k_j) \quad (2.4)$$

This decomposition allows us to rewrite the attention mechanism as:

$$\text{Attention}(Q, K, V)_i = \frac{\sum_{j=1}^T \phi(q_i)^\top \phi(k_j) v_j}{\sum_{n=1}^T \phi(q_i)^\top \phi(k_n)} \quad (2.5)$$

By applying the distributive property, we can factor out query-dependent terms:

$$\text{Attention}(Q, K, V)_i = \frac{\phi(q_i)^\top \left(\sum_{j=1}^T \phi(k_j) v_j^\top \right)}{\phi(q_i)^\top \left(\sum_{n=1}^T \phi(k_n) \right)} \quad (2.6)$$

2.1.4 Linear Complexity Derivation

Let us define two key terms that can be precomputed:

- $M = \sum_{j=1}^T \phi(k_j) v_j^\top \in \mathbb{R}^{d'd_v}$ (value aggregation matrix)
- $u = \sum_{n=1}^T \phi(k_n) \in \mathbb{R}^{d'}$ (normalization vector)

The attention output then simplifies to:

$$\text{Attention}(Q, K, V)_i = \frac{\phi(q_i)^\top M}{\phi(q_i)^\top u} \quad (2.7)$$

This reformulation yields a critical computational advantage: both M and u are computed once for the entire sequence, requiring $\mathcal{O}(Td'd_v)$ and $\mathcal{O}(Td')$ operations respectively. The per-query computation then requires only $\mathcal{O}(d'd_v)$ operations, leading to an overall time complexity of $\mathcal{O}(Td'd_v)$ linear in sequence length T .

2.1.5 Feature Map Selection

The choice of feature map ϕ determines both the expressivity and computational efficiency of linear attention [17]. The feature map must satisfy several properties:

1. **Kernel approximation quality:** $\phi(q_i)^\top \phi(k_j)$ should closely approximate $e^{q_i^\top k_j}$
2. **Computational efficiency:** ϕ should be fast to compute
3. **Non-negativity:** To preserve the probabilistic interpretation of attention

Common choices include:

ELU+1 Feature Map

$$\phi(x) = \text{ELU}(x) + 1 \quad (2.8)$$

where ELU is the Exponential Linear Unit activation function. This transformation ensures non-negativity while maintaining differentiability, approximating the exponential kernel reasonably well for practical purposes.

Positive Random Features

For more precise approximation of the exponential kernel, random Fourier features can be employed:

$$\phi_{\text{RFF}}(x) = \frac{e^{Wx}}{\sqrt{m}} \quad (2.9)$$

where $W \in \mathbb{R}^{md}$ is a random projection matrix with entries drawn from a Gaussian distribution. This approach offers unbiased estimation of the true softmax attention, with approximation error decreasing as the projection dimension m increases.

2.1.6 Mathematical Analysis of Algorithmic Steps

The linear attention algorithm can be decomposed into three phases:

1. **Feature Mapping:** Transform queries and keys: $\Phi_Q = \phi(Q)$, $\Phi_K = \phi(K)$
Complexity: $\mathcal{O}(Td')$

2. **Aggregation:** Compute $M = \sum_{j=1}^T \phi(k_j) v_j^\top$
Implementable as matrix multiplication: $M = \Phi_K^\top V$
Complexity: $\mathcal{O}(Td'd_v)$

Compute $u = \sum_{j=1}^T \phi(k_j)$
Implementable as vector summation: $u = \Phi_K^\top \mathbf{1}$
Complexity: $\mathcal{O}(Td')$

3. **Output Computation:** For each query i : $\text{output}_i = \frac{\phi(q_i)^\top M}{\phi(q_i)^\top u + \epsilon}$
Complexity: $\mathcal{O}(Td'd_v)$

The total complexity is dominated by the aggregation and output computation phases, yielding $\mathcal{O}(Td'd_v)$ linear in sequence length T .

Linear attention represents a mathematically elegant solution to the quadratic scaling problem in transformer architectures [1]. By reformulating attention through kernel methods and feature map decomposition, it achieves linear computational complexity while preserving the essential modeling capabilities of the attention mechanism. The performance-efficiency trade-off is controlled primarily through the choice of feature map, offering a flexible framework for efficient sequence modeling across diverse applications. This kernel-based approach not only provides computational efficiency but also establishes theoretical connections between transformers and other sequence modeling paradigms, opening avenues for further theoretical analysis and architectural innovation in efficient sequence processing.

2.2 Sparse Attention Mechanisms: A Theoretical Analysis

The standard attention mechanism in transformer models [1] computes pairwise interactions across all tokens in a sequence, leading to a computational complexity of $\mathcal{O}(n^2)$ for sequence length n . While effective for modeling relationships in shorter sequences, this quadratic scaling becomes computationally prohibitive for tasks requiring long-context processing, such as genome analysis or book summarization. Sparse attention addresses this fundamental bottleneck by restricting the attention matrix to a strategically selected subset of token pairs, thereby reducing computational complexity to $\mathcal{O}(n)$ or $\mathcal{O}(n \log n)$ while preserving the model’s ability to capture both local and global dependencies.

2.2.1 Formal Definition

Given an input sequence $X \in \mathbb{R}^{nd}$ where n is the sequence length and d is the embedding dimension, sparse attention computes an output $Z \in \mathbb{R}^{nd}$ as:

$$Z_i = \sum_{j \in \mathcal{S}(i)} \text{Softmax} \left(\frac{Q_i K_j^\top}{\sqrt{d}} \right) V_j \quad (2.10)$$

where $\mathcal{S}(i) \subset \{1, 2, \dots, n\}$ defines a *sparsity pattern* determining which tokens the i -th token attends to, with $|\mathcal{S}(i)| \ll n$. The key innovation lies in the design of these sparsity patterns, which must balance computational efficiency with the preservation of essential information flow between tokens.

2.2.2 Sparse Attention Patterns: Design and Theoretical Foundations

Local (Sliding Window) Attention

Local attention restricts each token to attend only to nearby tokens within a fixed radius $w/2$:

$$\mathcal{S}_{\text{local}}(i) = \{j : |i - j| \leq w/2\} \quad (2.11)$$

The theoretical basis for this pattern stems from the locality bias inherent in natural sequences like text and images, where syntactic and semantic relationships are often strongest between nearby tokens [5]. With a window size w , the computational complexity reduces to $\mathcal{O}(n \cdot w)$, which becomes linear when w is a constant independent of n .

Global Attention

To enable long-range dependencies, global attention designates a small set of g special tokens (e.g., [CLS] tokens or positional markers) that can attend to and be attended by all tokens in the sequence:

$$\mathcal{S}_{\text{global}}(i) = \begin{cases} \{1, 2, \dots, n\} & \text{if } i \in \mathcal{G} \\ \mathcal{G} & \text{if } i \notin \mathcal{G} \end{cases} \quad (2.12)$$

where \mathcal{G} represents the set of global token indices. These tokens function as information aggregation and distribution centers, ensuring that all tokens in the sequence can influence and be influenced by these summary tokens, which is crucial for tasks requiring sequence-level understanding.

Randomized Attention

To prevent over-localization and ensure connectivity across the entire sequence, randomized attention patterns incorporate stochastically sampled connections:

$$\mathcal{S}_{\text{rand}}(i) = \{j_1, j_2, \dots, j_r\} \text{ where } j_k \sim \text{Uniform}(1, n) \quad (2.13)$$

From a theoretical perspective, random patterns transform the attention graph into a *small-world network*, guaranteeing a maximum path length of $\mathcal{O}(\log n)$ between any two tokens [23]. This property ensures that information can flow between arbitrary token pairs through a logarithmic number of attention hops, allowing sparse attention to approximate full attention in expectation.

Dynamic Sparse Attention

Rather than using fixed patterns, dynamic sparse attention learns to identify important token interactions based on content:

$$\mathcal{S}_{\text{dynamic}}(i) = \text{Top-}k(f(Q_i, K)) \quad (2.14)$$

where $f : \mathbb{R}^d \mathbb{R}^{nd} \rightarrow \mathbb{R}^n$ represents a lightweight scoring function that produces importance scores for potential connections. This approach enables the model to adapt its attention structure to the specific content being processed for instance, focusing on semantically significant tokens while ignoring common stopwords.

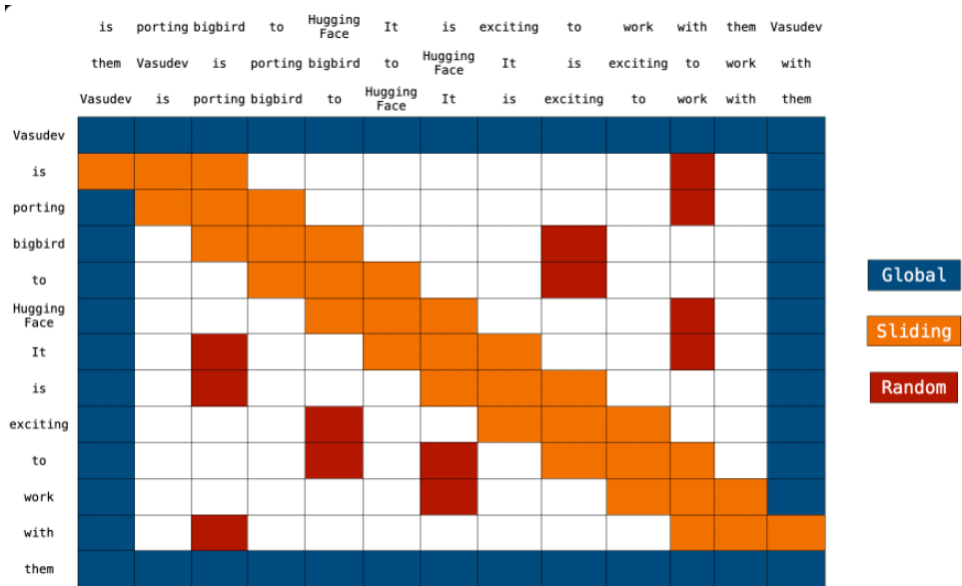


Figure 2.1: Sparse attention patterns visualization

2.2.3 Theoretical Analysis of Complexity and Expressivity

When combining multiple sparsity patterns for example, w local connections, g global tokens, and k dynamic or random connections per token the computational complexity becomes:

$$\text{FLOPs} = \mathcal{O}(n \cdot (w + g + k)) \quad (2.15)$$

This formulation offers several scaling regimes:

1. **Linear Complexity:** When w , g , and k are constants independent of sequence length, the overall complexity reduces to $\mathcal{O}(n)$.
2. **Subquadratic Complexity:** Setting $k = \mathcal{O}(\log n)$ maintains the small-world network property while achieving $\mathcal{O}(n \log n)$ scaling, significantly more efficient than the $\mathcal{O}(n^2)$ of full attention for long sequences.

From an expressivity standpoint, carefully designed sparse patterns can preserve the universal approximation properties of transformers. The theoretical foundation for this preservation rests on ensuring that:

1. Every token can influence every other token through a bounded number of attention hops.
2. The sparsity pattern includes sufficient local context to capture immediate dependencies.
3. Global tokens provide direct paths for long-range information flow.

Sparse attention mechanisms represent a principled approach to scaling transformer models to longer sequences by reducing the computational complexity from quadratic to near-linear. By strategically combining local, global, and random/dynamic attention patterns, these mechanisms preserve the expressive power of dense attention while dramatically improving efficiency. The theoretical guarantees provided by small-world networks and the empirical success of sparse transformers suggest that attention sparsification is a promising direction for future research in large-scale sequence modeling.

Future work might focus on formalizing approximation guarantees, developing hardware-aware sparsity patterns, and exploring the co-design of sparse attention with specialized accelerators to further improve computational efficiency while maintaining model performance.

2.3 Sliding Window Attention

Sliding window attention represents a computationally efficient sparse attention mechanism designed to overcome the quadratic complexity inherent in standard transformer architectures [1]. By constraining each token’s attention span to a fixed local neighborhood, this method significantly reduces computational and memory requirements from $\mathcal{O}(n^2)$ to $\mathcal{O}(nw)$ for sequence length n and window size w , while maintaining the capacity to model local dependencies [5]. This mechanism has emerged as a foundational component in efficient transformer variants designed for processing long sequences across domains such as text, audio, and genomics [17].

2.3.1 Theoretical Foundation: Locality in Sequence Modeling

The sliding window approach builds upon a key observation: many sequential data modalities exhibit strong local coherence patterns. In natural language, linguistic features and syntactic dependencies typically cluster within proximity; in time series data, nearby timesteps often

share stronger correlations than distant ones. Research indicates that syntactic dependencies in English text predominantly occur within a 5–10 token radius, suggesting that local context often provides substantial signal for token-level predictions.

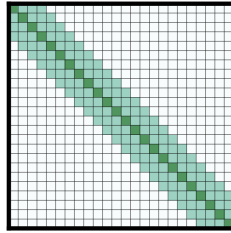
While standard self-attention mechanisms compute all pairwise interactions across tokens leading to $\mathcal{O}(n^2)$ complexity sliding window attention exploits spatial locality by replacing exhaustive attention with a sparse, fixed radius pattern. This approach not only addresses computational tractability for long sequences (e.g., $n > 10^4$) but aligns with the natural inductive biases present in sequential data.

2.3.2 Mechanism: Fixed-Length Contextual Windows

The core operation in sliding window attention centers on a symmetric, fixed-size window w (typically an odd integer) positioned around each token. For a token at position t in a sequence of length n , the set of attendable positions L_t is defined as:

$$L_t = \left\{ s \in [1, n] \mid \max\left(1, t - \lfloor w/2 \rfloor\right) \leq s \leq \min\left(n, t + \lfloor w/2 \rfloor\right) \right\} \quad (2.16)$$

This formulation effectively masks attention to tokens outside this interval. At sequence boundaries, the window is clipped to valid indices, ensuring all tokens attend to at least $\lceil w/2 \rceil$ predecessors and successors where available. This design imposes a uniform receptive field that slides positionally across the sequence, processing each token’s local context.



(b) Sliding window attention

Figure 2.2: Sliding window attention mechanism visualization

2.3.3 Receptive Field Properties

An important characteristic of sliding window attention is how information propagates through the network as layers deepen. Similar to convolutional neural networks, each additional layer expands the effective receptive field. After L layers of sliding window attention with window size w , a token can potentially receive information from positions up to $L \cdot \lfloor w/2 \rfloor$ tokens away.

This stacked effect means that despite the local constraints at each individual layer, deeper networks can capture longer-range dependencies. For instance, with a window size of 512 and 12 transformer layers, a token could theoretically access information from over 3,000 tokens in either direction albeit with information from the most distant tokens passing through multiple intermediate representations.

This hierarchical information flow permits the network to construct representations that effectively balance local precision with broader contextual awareness, mitigating the apparent limitation of fixed window sizes.

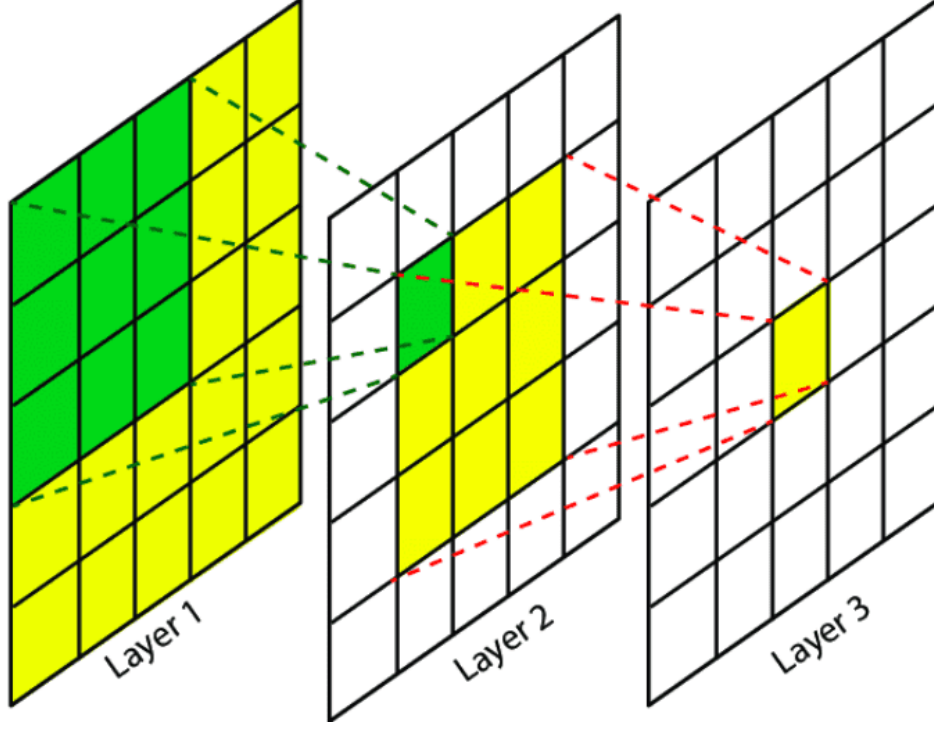


Figure 2.3: Receptive field expansion across layers

2.3.4 Mathematical Formulation

Given input embeddings $X \in \mathbb{R}^{nd}$, queries Q , keys K , and values V are computed as $Q = XW^Q$, $K = XW^K$, $V = XW^V$, where $W^Q, W^K, W^V \in \mathbb{R}^{dd}$ are learnable projection matrices. For each token position t :

1. **Local Score Computation:** Attention scores are calculated only within the local window L_t :

$$S_t = \left\{ \frac{Q_t K_s^\top}{\sqrt{d}} \mid s \in L_t \right\} \quad (2.17)$$

2. **Normalization and Value Aggregation:** Apply softmax normalization and aggregate values:

$$A_t = \text{softmax}(S_t), \quad O_t = \sum_{s \in L_t} A_{t,s} V_s \quad (2.18)$$

The full sequence output $O \in \mathbb{R}^{nd}$ is constructed by stacking each token's output representation O_t . This formulation avoids computing or storing the full nn attention matrix, achieving substantial efficiency gains.

2.3.5 Complexity Analysis

The computational and memory advantages of sliding window attention become particularly significant when processing long sequences:

- **Time Complexity:** $\mathcal{O}(nwd)$ for computing n tokens, each attending to w neighbors via d -dimensional operations [5].
- **Memory Complexity:** $\mathcal{O}(nw)$ for storing sparse attention maps, compared to $\mathcal{O}(n^2)$ in dense attention [17].

For practical applications where $w \ll n$ (e.g., $w = 512$, $n = 65,536$), this mechanism reduces computational requirements by orders of magnitude. This efficiency enables the processing of sequences 10–100 longer than standard transformers under equivalent memory constraints.

Sliding window attention offers a principled framework for efficient sequence modeling that leverages spatial locality to achieve linear scaling with sequence length. The mechanism’s fundamental trade-off replacing quadratic complexity with bounded context windows enables transformer architectures to process significantly longer sequences than previously feasible. As research into efficient transformers continues, sliding window attention remains a core building block for balancing computational efficiency with modeling capacity in long-sequence domains.

2.4 Understanding Flash Attention: Efficient Transformer Attention Mechanisms

Self-attention is a fundamental operation in transformer models, but its computational efficiency poses significant challenges. For sequences of length N , standard attention requires $O(N^2)$ memory and computation, creating a bottleneck for long sequences. Flash Attention addresses these limitations through algorithmic innovations that optimize memory access patterns and mathematical computations [25].

2.4.1 Why is Self-Attention Slow?

The self-attention mechanism can be represented mathematically as:

$$S = QK^T \in \mathbb{R}^{NN}, \quad P = \text{softmax}(S) \in \mathbb{R}^{NN}, \quad O = PV \in \mathbb{R}^{Nd}$$

Where Q , K , and V are the query, key, and value matrices. The computational bottleneck occurs because the attention matrix S requires $O(N^2)$ memory storage. Furthermore, standard implementations read from high-latency memory (HBM) multiple times, significantly impacting performance. Additionally, softmax calculations must be performed on the entire matrix, which becomes prohibitively expensive as sequence length increases.

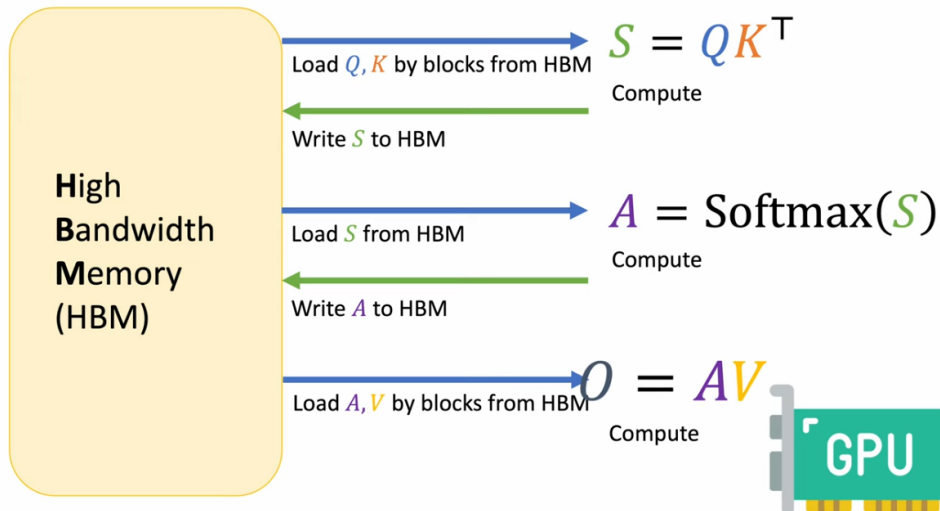


Figure 2.4: Illustration of standard attention computation

For long sequences, this becomes both memory intensive and computationally expensive, limiting the practical application of transformer models [1]. The sequential nature of these operations further exacerbates the performance limitations, as each step depends on the completion of previous calculations.

2.4.2 The Numerical Stability Challenge in Softmax

The Softmax Function

Given a vector $x \in \mathbb{R}^N$, the softmax is defined as:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^N e^{x_j}}$$

This operation normalizes the input vector into a probability distribution, with each element representing the relative importance of that position in the sequence.

The Numerical Instability Problem

When the values in vector x are large, the exponential function can produce values that exceed the representational capacity of floating-point numbers (float32 or float16), leading to numerical overflow or underflow. This makes the standard softmax calculation numerically unstable.

The issue becomes particularly problematic in deep learning applications where gradients may vanish or explode during backpropagation if the softmax is not computed correctly. This instability can lead to training failures or suboptimal model performance.

Safe Softmax Implementation

To address this issue, we can modify the softmax calculation by introducing a constant that makes the computation numerically stable:

$$\frac{e^{x_i}}{\sum_{j=1}^N e^{x_j}} = \frac{e^{x_i - \mu}}{\sum_{j=1}^N e^{x_j - \mu}} \quad \text{where } \mu = \max_i(x_i)$$

By subtracting the maximum value in the vector before applying the exponential function, we prevent numerical overflow while maintaining mathematical equivalence. This transformation is valid because the constant factor can be factored out of both the numerator and denominator, resulting in the same final probability distribution.

The safe softmax implementation ensures that the largest exponentiated value will be exactly 1 (when $x_i = \max_i(x_i)$), with all other values being less than or equal to 1, thus preventing overflow conditions while preserving the relative proportions necessary for the softmax operation.

2.4.3 Traditional Softmax Algorithm

The traditional algorithm for computing softmax requires multiple passes through the data, which is inefficient from a memory access perspective. The process involves:

First, finding the maximum value in the vector, which requires one full pass through all elements. This establishes the baseline for numerical stability in subsequent calculations.

Second, calculating the normalization factor by summing the exponentials of all elements (after subtracting the maximum), which requires another full pass through the data.

Third, applying the softmax to each element by dividing its exponential by the normalization factor, necessitating a third pass through the vector.

For a vector of length N , this approach requires $O(N)$ time complexity and $O(N)$ memory reads for each step, necessitating three complete passes through the vector. This becomes

particularly inefficient when dealing with large vectors or matrices, as is common in transformer models with long sequence lengths.

$$\begin{aligned}
& \mathbf{A} = \text{Softmax}(\mathbf{S}) \quad \mathbf{S} = \{x_1, x_2, \dots, x_N\} \\
& m_0 = -\infty \\
& \text{for } 1 \leq i \leq N \text{ do} \\
& \quad m_i = \max(m_{i-1}, x_i) \\
& d_0 = 0 \\
& \text{for } 1 \leq i \leq N \text{ do} \\
& \quad d_i = d_{i-1} + e^{x_i - m_N} \\
& \text{for } 1 \leq i \leq N \text{ do} \\
& \quad a_i = e^{x_i - m_N} / d_N
\end{aligned}$$

Figure 2.5: Traditional softmax computation process

Pseudocode for Traditional Softmax:

Algorithm 1 Traditional Softmax

```

1:  $m_0 \leftarrow -\infty$ 
2: for  $i = 1$  to  $N$  do
3:    $m_i \leftarrow \max(m_{i-1}, x_i)$ 
4: end for
5:  $l_0 \leftarrow 0$ 
6: for  $j = 1$  to  $N$  do
7:    $l_j \leftarrow l_{j-1} + e^{x_j - m_N}$ 
8: end for
9: for  $k = 1$  to  $N$  do
10:   $x_k \leftarrow e^{x_k - m_N} / l_N$ 
11: end for

```

This algorithm, while correct, does not optimize for modern memory hierarchies and can lead to significant performance bottlenecks when scaled to the large matrices involved in transformer attention mechanisms.

2.4.4 Online Softmax: A Dynamic Approach

The key innovation in optimizing softmax computation is an online algorithm that processes data incrementally, reducing the number of memory accesses required [25]. This approach enables more efficient computation by updating running statistics as new elements are processed.

Instead of making multiple passes through the data, we can compute both the maximum value and the normalization factor in a single pass by dynamically updating both values as we process each element. This reduces memory accesses and improves computational efficiency.

When we encounter a new maximum value, we must rescale the previously computed normalization factor. This rescaling is necessary to maintain the mathematical equivalence with the standard softmax calculation while using the updated maximum value for numerical stability.

For a new element x_i where $x_i > \text{current_max}$, we perform the following updates:

1. Update the maximum: $\text{new_max} = x_i$

2. Rescale the normalization factor:

$$l_{\text{new}} = l_{\text{old}} \cdot e^{\text{current_max} - \text{new_max}}$$

3. Add the contribution of the new element:

$$l_{\text{new}} = l_{\text{new}} + e^{x_i - \text{new_max}}$$

This online approach ensures that the normalization factor is continuously updated to reflect the current maximum value, maintaining numerical stability throughout the computation process while reducing the number of required passes through the data.

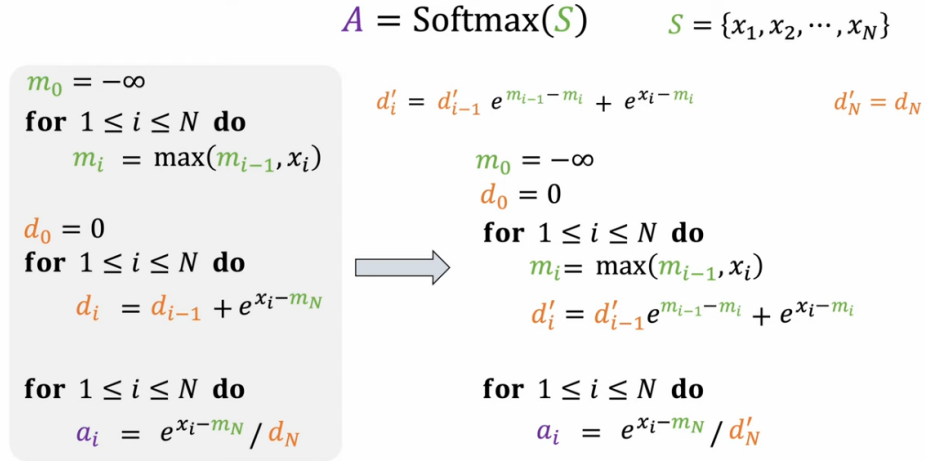


Figure 2.6: Online softmax computation process

Illustrated Example

For vector $X = [3, 2, 5, 1]$, the online softmax computation proceeds as follows:

1. When processing the first element (3), we initialize the maximum to 3 and the normalization factor to $e^{3-3} = 1$, representing the contribution of this element to the normalization sum.
2. When processing the second element (2), the maximum remains unchanged at 3, and we update the normalization factor to $1 + e^{2-3} = 1 + e^{-1} \approx 1.368$, adding the contribution of the second element.
3. When processing the third element (5), we encounter a new maximum. We update the maximum from 3 to 5, and we must rescale the current normalization factor accordingly: $1.368 \cdot e^{3-5} = 1.368 \cdot e^{-2} \approx 0.185$. We then add the contribution of the new element: $0.185 + e^{5-5} = 0.185 + 1 = 1.185$.
4. Finally, when processing the fourth element (1), the maximum remains at 5, and we update the normalization factor to $1.185 + e^{1-5} = 1.185 + e^{-4} \approx 1.187$.

This example demonstrates how the online softmax algorithm dynamically updates the maximum and normalization factor as it processes each element, reducing the need for multiple passes through the data.

Optimized Pseudocode

Algorithm 2 Online Softmax

```

1:  $m_0 \leftarrow -\infty$ 
2:  $l_0 \leftarrow 0$ 
3: for  $i = 1$  to  $N$  do
4:    $m_i \leftarrow \max(m_{i-1}, x_i)$ 
5:    $l_i \leftarrow l_{i-1} \cdot e^{m_{i-1}-m_i} + e^{x_i-m_i}$ 
6: end for
7: for  $u = 1$  to  $N$  do
8:    $x_u \leftarrow e^{x_u-m_N} / l_N$ 
9: end for
  
```

This approach reduces the required memory accesses to just one full pass through the vector for computing the max and normalization factor, followed by one pass for applying the normalization. The reduction in memory access operations significantly improves performance, especially for large vectors or matrices.

2.4.5 Block Matrix Multiplication in Flash Attention

Block matrix multiplication forms a critical component of Flash Attention’s optimization strategy [25]. By decomposing the large attention matrix computation into smaller, manageable blocks, Flash Attention effectively utilizes the memory hierarchy of modern hardware accelerators such as GPUs and TPUs.

Mathematical Formulation of Block Matrix Multiplication

Consider the standard attention computation:

$$\mathbf{O} = \text{softmax}(\mathbf{Q}\mathbf{K}^T) \mathbf{V} \quad (2.19)$$

where $\mathbf{Q} \in \mathbb{R}^{Nd}$, $\mathbf{K} \in \mathbb{R}^{Nd}$, and $\mathbf{V} \in \mathbb{R}^{Nd}$ are the query, key, and value matrices, respectively, with N being the sequence length and d the feature dimension.

In Flash Attention, these matrices are partitioned into B blocks along the sequence dimension:

$$\mathbf{Q} = \begin{bmatrix} \mathbf{Q}_1 \\ \mathbf{Q}_2 \\ \vdots \\ \mathbf{Q}_B \end{bmatrix}, \quad \mathbf{K} = \begin{bmatrix} \mathbf{K}_1 \\ \mathbf{K}_2 \\ \vdots \\ \mathbf{K}_B \end{bmatrix}, \quad \mathbf{V} = \begin{bmatrix} \mathbf{V}_1 \\ \mathbf{V}_2 \\ \vdots \\ \mathbf{V}_B \end{bmatrix} \quad (2.20)$$

where each block $\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i \in \mathbb{R}^{(N/B)d}$.

The attention matrix $\mathbf{S} = \mathbf{Q}\mathbf{K}^T$ is computed block-wise as:

$$\mathbf{S} = \begin{bmatrix} \mathbf{S}_{11} & \mathbf{S}_{12} & \cdots & \mathbf{S}_{1B} \\ \mathbf{S}_{21} & \mathbf{S}_{22} & \cdots & \mathbf{S}_{2B} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{S}_{B1} & \mathbf{S}_{B2} & \cdots & \mathbf{S}_{BB} \end{bmatrix} \quad (2.21)$$

where each submatrix $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{(N/B)(N/B)}$ represents the attention scores between the i -th query block and j -th key block.

Block-wise Softmax Computation

The softmax operation cannot be directly decomposed into blocks due to its global normalization. However, Flash Attention applies the online softmax algorithm at the block level by maintaining running statistics (maximum values and normalization factors) for each row of blocks.

For each row i of the attention matrix, we process the blocks $S_{i1}, S_{i2}, \dots, S_{iB}$ sequentially, updating the maximum values and normalization factors as we go. This allows us to compute the softmax operation without materializing the entire attention matrix in memory.

The output for each block O_i can be computed as:

$$O_i = \sum_{j=1}^B \text{softmax}(S_{ij}) V_j$$

Where the softmax is computed using the online algorithm with properly maintained running statistics across blocks.

Memory Efficiency Through Block Processing

By processing the attention mechanism in blocks that fit in fast memory (SRAM or cache), Flash Attention reduces the need for frequent accesses to high-latency memory (HBM or DRAM). This is particularly important because memory bandwidth is often the bottleneck in attention computation rather than computational throughput.

Each block is loaded into fast memory only once for processing, after which the intermediate results are updated without requiring additional high-latency memory accesses. This significantly reduces the memory bandwidth requirements compared to standard attention implementations.

Furthermore, the block-wise approach allows for better utilization of memory hierarchy. The block sizes can be optimized based on the specific hardware architecture to maximize cache utilization and minimize cache misses.

Optimal Block Size Determination

The optimal block size for Flash Attention depends on several factors:

1. The size of available fast memory (SRAM or cache)
2. The dimensionality of the attention mechanism (head size and number of heads)
3. The characteristics of the underlying hardware architecture

Flash Attention dynamically determines the optimal block size based on these factors, ensuring maximum computational efficiency while maintaining memory constraints. This adaptive approach allows Flash Attention to perform well across different hardware configurations and model architectures.

2.4.6 Flash Attention: Tiling and IO-Aware Algorithms

Flash Attention extends the online softmax algorithm by incorporating tiling techniques and IO-aware algorithmic design to further optimize attention computation [25, 24]. These techniques consider the hierarchical nature of modern memory systems and optimize operations accordingly.

IO-Aware Algorithmic Tiling

Flash Attention partitions the attention matrix into smaller blocks or "tiles" that can fit in fast memory (SRAM). This tiling approach is specifically designed to minimize data movement between different levels of the memory hierarchy, which is often the most significant performance bottleneck in matrix computations.

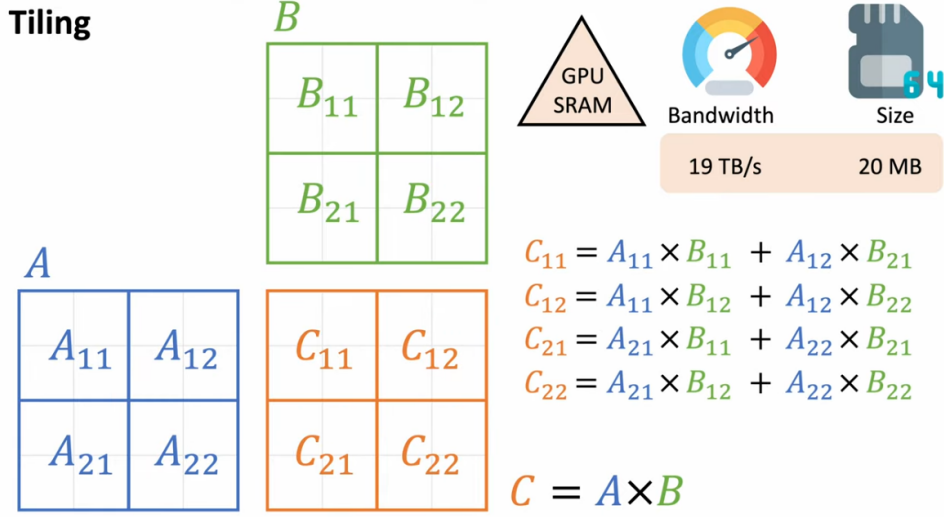


Figure 2.7: IO-aware tiling in Flash Attention

The tiling strategy incorporates several important considerations:

First, it reduces the number of accesses to high-bandwidth memory (HBM) by ensuring that each tile is loaded into fast memory only once and processed completely before moving to the next tile. This minimizes the data transfer overhead, which can be substantial for large matrices.

Second, it maximizes the utilization of on-chip memory (SRAM) by carefully selecting tile sizes that fit optimally in the available fast memory while providing sufficient computational work to amortize the memory transfer costs.

Third, it allows for parallel processing of attention computations within each tile, taking advantage of the parallel processing capabilities of modern hardware accelerators. This parallelism is essential for achieving high computational throughput.

Mathematical Formulation of Tiled Attention

For a sequence partitioned into B blocks, the attention computation can be rewritten as:

$$O_i = \sum_{j=1}^B \text{softmax}(S_{ij}) V_j$$

Where S_{ij} represents the attention scores between blocks i and j , and V_j is the values matrix for block j .

The key insight is that we can compute partial softmax results for each block and combine them using the online softmax algorithm, adjusting the normalization factors appropriately as we process each block. This approach allows us to process the attention mechanism in a memory-efficient manner without sacrificing mathematical correctness.

For each row of blocks in the attention matrix, Flash Attention maintains running statistics (maximum values and normalization factors) that are updated as each block is processed. This

enables the computation of the softmax operation without materializing the entire attention matrix, significantly reducing memory requirements.

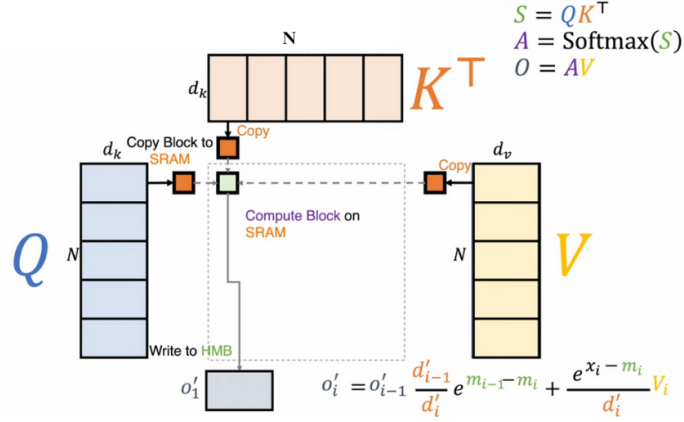


Figure 2.8: Block-wise processing in Flash Attention

Flash Attention Algorithm

The Flash Attention algorithm combines tiling with the online softmax approach in a carefully orchestrated manner:

First, it partitions the input matrices Q , K , and V into blocks that fit in SRAM, based on the available fast memory and the dimensions of the matrices. These blocks are sized to optimize memory utilization while providing sufficient computational work.

Second, for each query block, it loads the query block into SRAM and processes all key-value blocks sequentially. For each key-value block, it loads the block into SRAM, computes partial attention scores, and updates the running maximum and normalization factors using the online softmax algorithm. This step is performed without materializing the entire attention matrix in memory.

Third, it applies the final normalization to compute the output for each query block, using the accumulated statistics from processing all key-value blocks. This ensures that the softmax operation is computed correctly across the entire row of blocks.

Finally, it combines all output blocks to form the complete output matrix, which represents the result of the attention mechanism.

This approach significantly reduces memory bandwidth requirements and improves computational efficiency, allowing for processing of much longer sequences than traditional attention implementations. The careful orchestration of memory accesses and computational operations enables Flash Attention to achieve substantial performance improvements while maintaining mathematical correctness.

Flash Attention significantly optimizes transformer models by combining safe softmax computation with block matrix multiplication and tiling techniques [25]. This approach enhances performance and memory efficiency, enabling the processing of longer sequences without sacrificing accuracy. The integration of online softmax computation and block-wise processing addresses computational and memory challenges, broadening the practical applications of transformers in

NLP, computer vision, and beyond. Flash Attention exemplifies the growing trend of hardware-aware algorithm design, offering a path for more energy efficient and scalable machine learning models.

2.5 Conclusion

Attention mechanisms lie at the heart of many of today’s most powerful models, and optimizing their efficiency is crucial for scaling to longer contexts and larger datasets. Through our exploration of linear, sparse, flash, and sliding window attention, we have seen a range of strategies to reduce computational complexity, manage memory usage, and maintain or even improve performance. Each method embodies a distinct design philosophy from rethinking the mathematical structure of attention, to intelligently limiting the receptive field, to reorganizing memory access patterns for greater speed. While no single approach universally dominates, together they represent critical steps toward building faster, more capable models. This evolution of attention mechanisms not only enhances current architectures but also opens new avenues for future research in efficient and scalable deep learning.

Chapter 3

Experimental Results and Comparative Analysis

Building on the previous chapter’s review of state-of-the-art attention mechanisms, this chapter presents a focused empirical comparison of five variants Standard, Sliding-Window, Sparse, Linear, and Flash Attention. We evaluate their performance across three tasks: next-character prediction with GPT on the Tiny Shakespeare corpus, masked-character prediction with BERT-style models, and English-to-French translation using an encoder–decoder Transformer on a subset of the Tatoeba corpus. Our goal is to assess how each mechanism balances predictive quality, efficiency, and memory usage. By isolating the attention module and keeping all other settings constant, we aim to reveal the practical strengths and limitations of each approach, providing guidance on when and why to use them.

3.1 Experimental Setup and Implementation Details

3.1.1 Overview of Attention Mechanisms

This work explores five distinct attention mechanisms, each designed to balance computational efficiency with the ability to model complex dependencies. The following provides a theoretical explanation of the underlying principles behind each implementation, abstracted from the code-level details used in our experiments.

- **Scaled Dot-Product Attention**

This is the standard mechanism used in Transformer models, where each token attends to all previous tokens using dot products of their query and key vectors, scaled by the dimensionality of the embeddings. A softmax function converts these scores into attention weights, which are then used to compute a weighted sum of the value vectors. A causal mask ensures autoregressive behavior, preventing a token from attending to future positions.

- **Enhanced Sparse Attention**

Inspired by sparse attention frameworks like BigBird, this mechanism restricts attention to a structured set of tokens, including local neighbors, a few global tokens, and a learned selection of important positions. The sparsity pattern improves computational efficiency while retaining access to key contextual information. A small neural network predicts the importance of keys, enabling adaptive sparsity.

- **Enhanced Linear Attention**

This mechanism approximates the softmax attention using kernel feature maps specifically the $\text{ELU} + 1$ transformation on queries and keys. The attention computation becomes linear in sequence length by reordering operations and removing the softmax. To maintain autoregressive behavior, a causal decay mask is applied. This approach significantly reduces memory and computational complexity.

- **Flash Attention**

Flash Attention leverages optimized GPU memory usage to compute exact attention with reduced overhead. It processes attention in small, register resident tiles to minimize reads and writes to high-bandwidth memory. This tiled approach, along with numerically stable softmax computation, enables faster and more efficient execution without approximation.

- **Sliding-Window Attention**

This variant restricts each token’s attention to a fixed-size local window around it. To capture longer dependencies, it optionally includes dilated attention hops and a small set of global tokens. Such a pattern is especially useful for long sequences, offering a trade-off between full attention expressivity and computation speed.

3.1.2 Evaluation Metrics

We measure each attention variant on:

Metric	Description
Training Time (minutes)	Total time needed to train the model
Peak GPU Memory (MB)	Maximum memory used during training
Average Inference Latency (ms)	Time needed to process one input
Next-character prediction: Perplexity	How well the model predicts next characters (lower values are better)
Masked-character prediction: Accuracy	Percentage of correctly filled-in masked characters
Translation: BLEU score	Translation quality score (0-100, higher is better)

Table 3.1: Evaluation metrics used in our comparative analysis

3.1.3 Experimental Setup

- **Compute:** Google Colab (free tier) with an NVIDIA T4 GPU
- **PyTorch version:** ≥ 2.0 (for Flash Attention fallback to native if available)

3.1.4 Experimental Philosophy and Fairness

- **Single-codebase:** All models share identical code except for the attention module swap.
- **Architecture consistency:**
 - GPT experiments use *causal* self-attention.
 - BERT experiments use *bidirectional* masked self-attention.

- Translation uses the same attention type for encoder self-attention, decoder masked self-attention, and cross-attention.
- **Controlled variables:** same sequence length, embedding dims, optimizer, and training budget per task.
- **Limitations:** constrained by Colab session limits and T4 memory models are kept small (block size ≤ 1024).

3.2 Comparison Using GPT Models

3.2.1 Task Definition

Next-character prediction on Tiny Shakespeare: given a sequence of up to 128 characters, the model must predict the next character. This simple autoregressive task exposes how each attention mechanism handles causal dependency and sequential token mixing.

3.2.2 Model Configuration

All GPT variants share the same core hyperparameters (from `ModelConfig`):

```
vocab_size = 65           # Tiny Shakespeare's 65-character alphabet
n_embd     = 64           # Embedding dimension
n_head     = 4            # Number of attention heads
n_layer    = 4            # Transformer blocks
block_size = 128          # Context length
batch_size = 32
max_iters  = 4_200
dropout    = config.dropout
```

Only the `attn_type` (sliding_window, flash, standard, sparse, linear) differs between runs; everything else remains constant to ensure a fair comparison.

3.2.3 Dataset

Tiny Shakespeare is a ~ 1 MB corpus of Shakespeare's complete works, containing roughly **111K characters** and a **65-token vocabulary** (letters, punctuation, whitespace). We split it into 90% train and 10% validation characters, with no further preprocessing.

3.2.4 Training Setup

- **Hardware:** Google Colab free tier on a Tesla T4 GPU (≈ 16 GB RAM)
- **Optimization:** AdamW with OneCycleLR scheduler (details per run)
- **Metrics logged:**
 - **Perplexity** (train & validation)
 - **Wall-clock training time** (minutes until 4,200 iterations)
 - **Peak GPU memory** (MB)
 - **Average inference latency** (ms per forward pass, sequence length = 128)

3.2.5 Performance Results

Training and Validation Loss

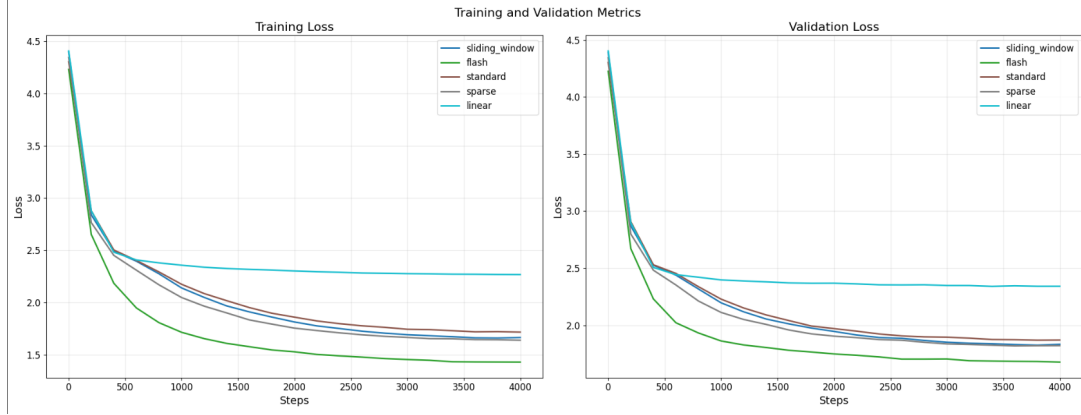


Figure 3.1: Training and validation loss curves for five attention variants

Training Loss:

All five attention variants exhibit a rapid initial decrease in training loss, dropping from ~ 4.4 to ~ 2.7 within the first 200 steps as the models learn basic character dependencies. Thereafter, Flash Attention (green) continues to improve most aggressively, reaching a final loss of ≈ 1.43 by step 4,000. In contrast, Linear Attention (cyan) plateaus early settling around 2.27 suggesting its kernel approximation struggles to model fine-grained autoregressive structure. Standard (brown), Sliding-Window (blue), and Sparse (gray) attentions fall between these extremes, converging to ≈ 1.65 – 1.70 . This ordering highlights Flash’s numerical stability and efficiency in gradient propagation versus the weaker representational power of Linear attention under identical configurations.

Validation Loss:

The validation curves mirror the training trends, underscoring generalization behavior. Flash Attention again achieves the lowest held-out loss (~ 1.43), with very little overfitting gap. Sliding-Window, Standard, and Sparse attentions converge closely around 1.85–1.90, indicating similar capacity to model the validation set once properly trained. Linear attention’s validation loss stalls at ≈ 2.35 , reflecting its higher bias. Notably, Sparse attention generalizes almost as well as plain self-attention despite its irregular sparse connectivity, while Flash’s memory-efficient kernel yields both fastest convergence and best generalization.

Validation and Training Perplexity

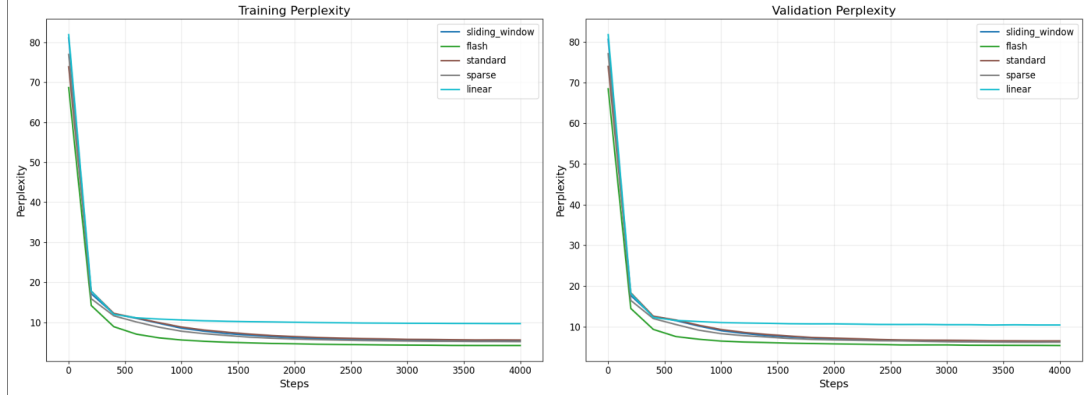


Figure 3.2: Training and validation perplexity curves for five attention variants

Training Perplexity:

Perplexity, being the exponential of loss, amplifies these differences. Flash Attention quickly reduces perplexity from ~ 70 to ~ 8 by step 400, eventually hitting ~ 4.2 . Sliding-Window, Standard, and Sparse attentions drop to ≈ 5.2 – 5.6 , clustering tightly, which aligns with their loss curves. Linear attention remains well above at ~ 9.7 , reinforcing that its linearized approximation fails to capture the full sequential dynamics required for tight character prediction. The steeper perplexity curve for Flash also illustrates its capacity to focus representational power where it matters most.

Validation Perplexity:

On the held-out set, Flash again leads with the lowest perplexity (~ 5.35), followed by Sparse (~ 6.18), Sliding-Window (~ 6.25), and Standard (~ 6.49). Linear lags at ~ 10.4 . The relative gaps shrink slightly compared to training, indicating some leveling in generalization, but the overall ranking remains consistent. Flash’s advantage is most pronounced early in training and persists through convergence, while the other methods trade off quality for simplicity or locality constraints.

Performance Metrics (Time, Memory, Latency)

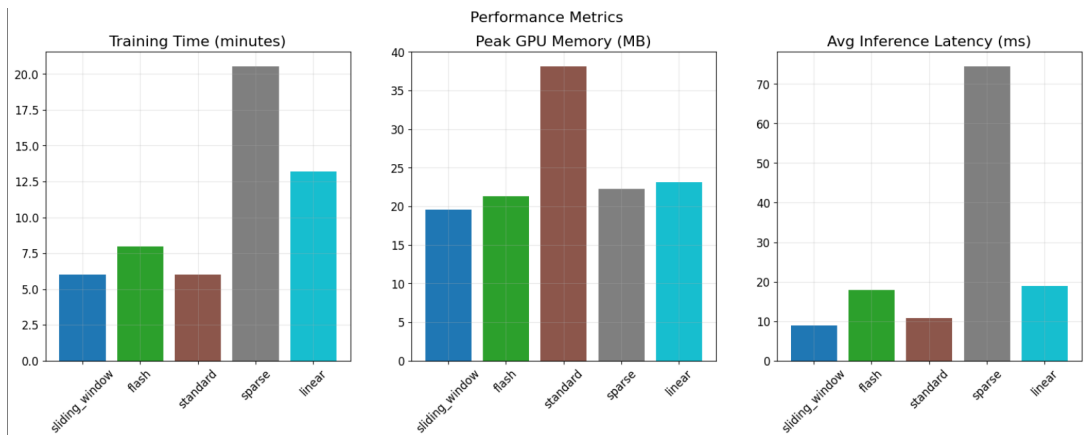


Figure 3.3: Performance metrics comparison across attention variants

The bar charts reveal stark trade-offs: Sparse attention is by far the slowest (20.6 min) and highest-latency (~ 74.5 ms) despite moderate memory (22.2 MB), due to the overhead of dy-

namic mask computation. Linear attention halves sparse’s runtime (~ 13.2 min) but still suffers from high latency (~ 18.9 ms) even though memory use is modest (23.1 MB). Flash attention takes ~ 7.98 min to train and ~ 18 ms per inference, striking a balance between speed and quality, with low memory (21.3 MB). Standard and Sliding-Window are the fastest to train (~ 6.01 min each), with Sliding-Window using the least memory (19.6 MB) and the lowest inference latency (~ 8.9 ms), making it ideal for throughput critical applications at a slight cost in perplexity.

Summary of the Results

Attention Type	Train Time (min)	Peak GPU Mem (MB)	Inference Latency (ms)	Val Perplexity
sliding_window	6.01	19.56	8.94	6.25
flash	7.98	21.31	17.94	5.35
standard	6.01	38.14	10.82	6.49
sparse	20.55	22.23	74.53	6.18
linear	13.21	23.11	18.89	10.40

Table 3.2: Summary of performance metrics across attention variants

Overall Findings

Across next-character prediction experiments, **Flash Attention** delivers the best predictive quality (lowest loss & perplexity), but at a moderate cost in training time and inference latency. **Sliding-Window Attention** provides the strongest efficiency quality trade-off: it trains fastest, uses the least memory, and offers competitive perplexity. **Standard Attention** remains a reliable baseline fast to train and infer but slightly behind in loss and perplexity. **Sparse Attention** matches standard in quality but incurs prohibitive runtimes and high latency. **Linear Attention** underperforms on both loss and perplexity despite modest resource use. These results underscore a clear trade-off between accuracy and computational efficiency when choosing an attention mechanism.

3.2.6 Ranking of Attention Mechanisms

For each metric, lower is better including perplexity, where a lower value indicates better predictions.

Metric	1st place	2nd place	3rd place	4th place	5th place
Train Time	sliding_window	standard	flash	linear	sparse
GPU Memory	sliding_window	flash	sparse	linear	standard
Inference Latency	sliding_window	standard	flash	linear	sparse
Validation PPL	flash	sparse	sliding_window	standard	linear

Table 3.3: Ranking of attention mechanisms across performance metrics

3.2.7 Discussion

- **Flash Attention** achieves the **best predictive quality** (val PPL 5.35) thanks to its numerically stable, memory-efficient kernel, but pays a cost in **longer training** and **higher inference latency** (≈ 18 ms).
- **Sliding-window** excels in **speed** and **low memory** it ties for fastest training (6.01 min) and has the lowest inference latency (8.94 ms), making it ideal where throughput is critical, though its PPL (6.25) lags flash by $\sim 17\%$.

- **Standard attention** remains a solid baseline, matching sliding-window in training time but using almost double the GPU RAM (38 MB) and yielding slightly worse PPL (6.49).
- **Sparse attention** reduces memory footprint (22 MB) and slightly improves on standard’s PPL (6.18) but incurs a huge **training time penalty** (20.55 min) and very slow inference (74.5 ms), due to complex mask computation.
- **Linear attention** shows the **worst perplexity** (10.40), indicating that kernel-based approximations still struggle on tight autoregressive tasks. Its moderate memory and latency suggest some efficiency gains, but at a steep quality cost.

3.3 Comparison Using BERT-Like Models

3.3.1 Task Definition

Masked-character prediction on Tiny Shakespeare: given a sequence of up to 128 characters with a random subset of positions masked out, the model must predict the original characters at those masked positions. This bidirectional cloze task evaluates how each attention mechanism aggregates context from both left and right.

3.3.2 Model Configuration

All BERT variants share the same core hyperparameters (from `ModelConfig`):

```

vocab_size = 66           # 65 characters + special [MASK] token
n_embd      = 64           # Embedding dimension
n_head      = 4            # Number of attention heads
n_layer     = 4            # Transformer blocks
block_size  = 128         # Sequence length
batch_size  = 32
max_iters   = 4_200
dropout     = config.dropout

```

Only the `attn_type` (sliding_window, standard, sparse, linear, flash) differs; the rest of the architecture MLP layers, layer norm placement, positional embeddings remains identical across runs.

3.3.3 Dataset

Tiny Shakespeare (~1 MB) contains ≈111K characters drawn from Shakespeare’s complete works, tokenized into a **66-token vocabulary** (letters, punctuation, whitespace, plus a [MASK] token). We use the same 90% train / 10% validation split, randomly masking 15% of tokens per sequence.

3.3.4 Training Setup

- **Compute:** Google Colab free tier, NVIDIA T4 GPU (≈16 GB VRAM)
- **Optimization:** AdamW with OneCycleLR scheduler
- **Metrics logged:**
 - **Training time** (minutes to reach 4,200 iterations)
 - **Peak GPU memory** (MB)
 - **Average inference latency** (ms per forward pass, batch of sequences)
 - **Validation accuracy** (percentage of correctly predicted masked tokens)

3.3.5 Performance Results

Training and Validation Loss

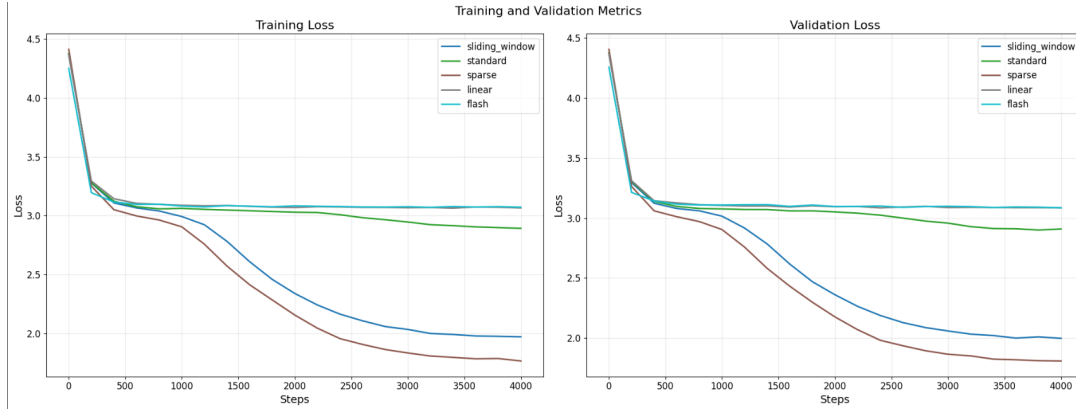


Figure 3.4: Training and validation loss for BERT-like models with different attention mechanisms

The loss curves reinforce the accuracy findings. The "sparse" attention model exhibits the steepest and most consistent decline in both training and validation loss, suggesting it learns efficiently and avoids overfitting. "Sliding window" follows closely behind but with slightly higher losses. In contrast, "standard", "linear", and "flash" attentions plateau very early at a relatively high loss (~ 3.0), indicating poor learning and convergence behavior. The fact that these three models maintain high and flat losses throughout suggests they might be underfitting the training data, unable to capture the task's complexity.

Training and Validation Accuracy

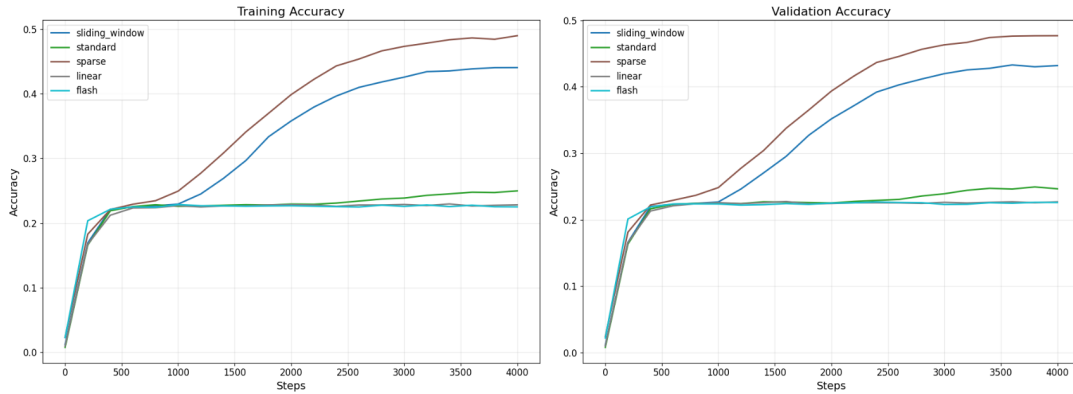


Figure 3.5: Training and validation accuracy for BERT-like models with different attention mechanisms

In the training and validation accuracy plots, the "sparse" attention model clearly outperforms all other variants, reaching the highest accuracy by the end of training (~ 0.49 training, ~ 0.48 validation). "Sliding window" attention also shows decent improvement over time but lags behind "sparse". Meanwhile, "standard", "linear", and "flash" attentions plateau very early (around 0.22–0.25 accuracy) and barely improve throughout training. This suggests that "sparse" attention can better capture long-range dependencies in masked character prediction.

tasks, while "standard", "linear", and "flash" attentions struggle to generalize and learn meaningful patterns from the data.

Performance Metrics: Training Time, Peak GPU Memory, and Inference Latency

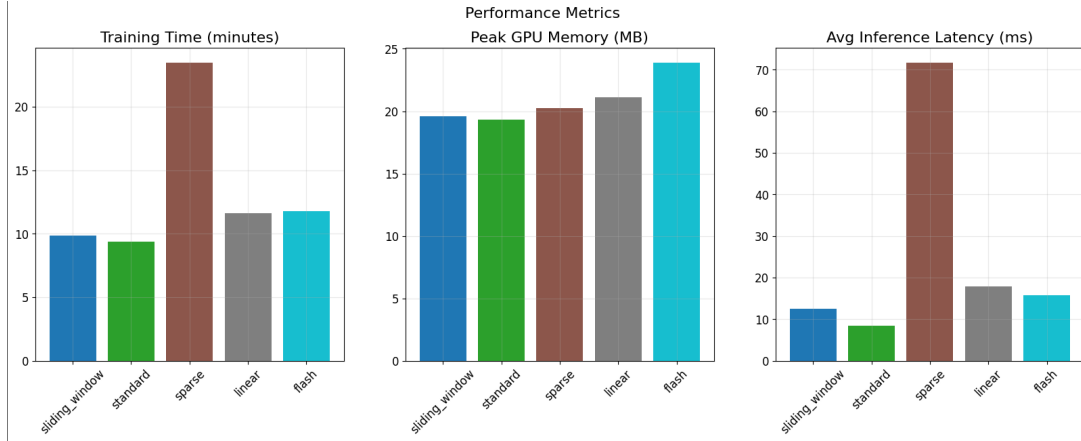


Figure 3.6: Performance metrics for BERT-like models with different attention mechanisms

In terms of performance, "sparse" attention is significantly more expensive: it requires by far the longest training time (~ 24 minutes) and the highest inference latency (~ 71 ms), despite moderate memory usage (~ 20 MB). "Sliding window", "linear", and "flash" attentions show a more balanced profile, with moderate training times (~ 10 – 12 minutes) and manageable latency (~ 12 – 18 ms). "Standard" attention is the most lightweight in terms of latency (~ 9 ms) and training time (~ 9 minutes), but this efficiency comes at the cost of much worse accuracy and loss performance. Interestingly, "flash" attention consumes the most GPU memory (~ 24 MB), but without strong corresponding gains in accuracy or loss, suggesting inefficiencies in its implementation or a mismatch for this particular task.

Summary of the Results

Attention Type	Train Time (min)	Peak GPU Mem (MB)	Inf. Latency (ms)	Val Accuracy (%)
sliding_window	9.87	19.57	12.45	43.20
standard	9.35	19.32	8.42	24.66
sparse	23.48	20.24	71.77	47.70
linear	11.59	21.09	17.87	22.69
flash	11.80	23.91	15.74	22.60

Table 3.4: Summary of performance metrics for BERT-like models

Overall Findings

Among the tested attention mechanisms, **sparse attention** achieves the best accuracy and lowest loss, demonstrating a clear advantage in learning effectiveness for the masked character prediction task. However, this superior performance comes at the cost of much higher training time and inference latency compared to the other methods. **Sliding window attention** offers a good compromise, achieving decent performance with moderate resource usage. In contrast, **standard**, **linear**, and **flash** attentions perform poorly both in terms of accuracy and loss, despite being faster and more memory efficient. Overall, there is a clear trade-off between performance quality and computational efficiency: sparse attention is best for accuracy, while lighter models like standard or flash might be preferred if speed or resource usage is a priority.

3.3.6 Ranking of Attention Mechanisms

For each metric, "1st" is best (fastest/least resources or highest accuracy):

Metric	1st place	2nd place	3rd place	4th place	5th place
Train Time	standard	sliding_window	linear	flash	sparse
GPU Memory	standard	sliding_window	sparse	linear	flash
Inference Latency	standard	sliding_window	flash	linear	sparse
Validation Accuracy	sparse	sliding_window	standard	flash	linear

Table 3.5: Ranking of attention mechanisms for BERT-like models

3.3.7 Discussion

- **Sparse attention** yields the **highest accuracy** (47.7%), slightly ahead of sliding-window (43.2%), demonstrating that content-based dynamic sparsity helps capture bidirectional dependencies more effectively than fixed windows. However, it incurs a **massive training time penalty** (≈ 23.5 min) and **very slow inference** (71.8 ms), due to the overhead of mask computation and top-k selection.
- **Sliding-window** strikes the best **resource balance**, with second best accuracy (43.2%), minimal memory (19.6 MB), and low latency (12.5 ms). Its locality bias still allows sufficient global context via global tokens.
- **Standard attention** remains the speed and memory leader fastest training (9.35 min), lowest memory (19.3 MB), and fastest inference (8.4 ms) but at the cost of much lower accuracy (24.7%).
- **Flash attention** does not improve accuracy over standard (22.6%) despite higher memory usage (23.9 MB) and moderate latency (15.7 ms), suggesting RoPE and memory efficient kernels offer less benefit for small bidirectional tasks.
- **Linear attention** performs worst on accuracy (22.7%) and offers only modest speed/memory improvements over flash, indicating that kernel-based approximations struggle to model the rich bidirectional interactions required for masked prediction.

3.4 Comparison Using Encoder–Decoder Transformers for Translation

3.4.1 Task Definition

We evaluate **English→French translation** using a full encoder–decoder Transformer architecture originally introduced by Vaswani et al. [1]. This model relies entirely on self and cross-attention layers dispensing with recurrence or convolution to map source to target sequences. In their seminal experiments, Vaswani et al. reported a new state-of-the-art BLEU of 41.8 on WMT-14 English→French after training on 36M sentence pairs.

3.4.2 Model Configuration

All runs share the same `ModelConfig` hyperparameters, differing only in `attn_type` (standard, sparse, linear, flash, sliding_window):

```

class ModelConfig:
    def __init__(self, attn_type, src_vocab_size, tgt_vocab_size):
        self.src_vocab_size = src_vocab_size
        self.tgt_vocab_size = tgt_vocab_size
        self.n_embd          = 256
        self.n_head           = 2
        self.n_layer          = 2
        self.block_size       = 128
        self.batch_size       = 64
        self.max_iters        = 1200
        self.dropout          = config.dropout

```

We apply **Byte-Pair Encoding (BPE)** tokenization separately to source and target.

3.4.3 Dataset

We use a subset of the **Tatoeba English-French parallel corpus**, a volunteer-contributed collection of sentence translations. ManyThings.org reports **232,736** English-French pairs in the full Tatoeba release; for this study we loaded **30,000** randomly selected pairs for training and validation. The Tatoeba project maintains over 12.6 million sentences in 426 languages under CC BY, with 276 language pairs exceeding 10,000 translations [tatoeba].

3.4.4 Training Setup

- **Hardware:** Google Colab (free tier) with NVIDIA Tesla T4 (16 GB GDDR6)
- **GPU microarchitecture:** T4 is built on NVIDIA’s Turing design
- **Available VRAM:** Colab reports ~ 15.8 GB available (≈ 1 GB reserved for ECC)
- **Optimizer:** AdamW with cosine annealing and OneCycleLR schedules
- **Metrics:** Wall-clock train time, peak GPU memory, avg. inference latency, and BLEU on validation

3.4.5 Performance Results

Training and Validation Loss

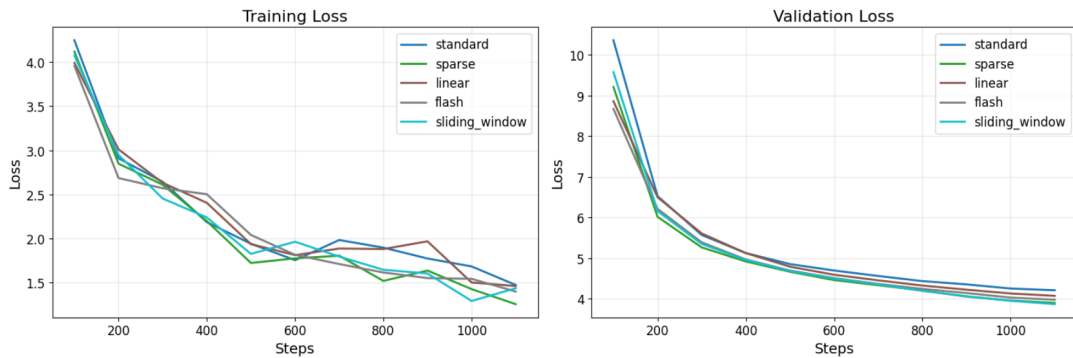


Figure 3.7: Training and validation loss for translation models with different attention mechanisms

Both the training and validation loss plots exhibit a consistent downward trend across steps for all attention types, which is a good indicator of successful learning. In terms of training loss, "sparse" and "flash" attentions achieve slightly lower losses than the others towards the end, suggesting more efficient training. Validation loss mirrors this, with "sparse" achieving the lowest final validation loss, closely followed by "flash" and "sliding_window." "Standard" attention consistently maintains a slightly higher loss. This indicates that while all models are learning, "sparse" and "flash" attention mechanisms generalize slightly better to unseen data in this setup.

BLEU Score

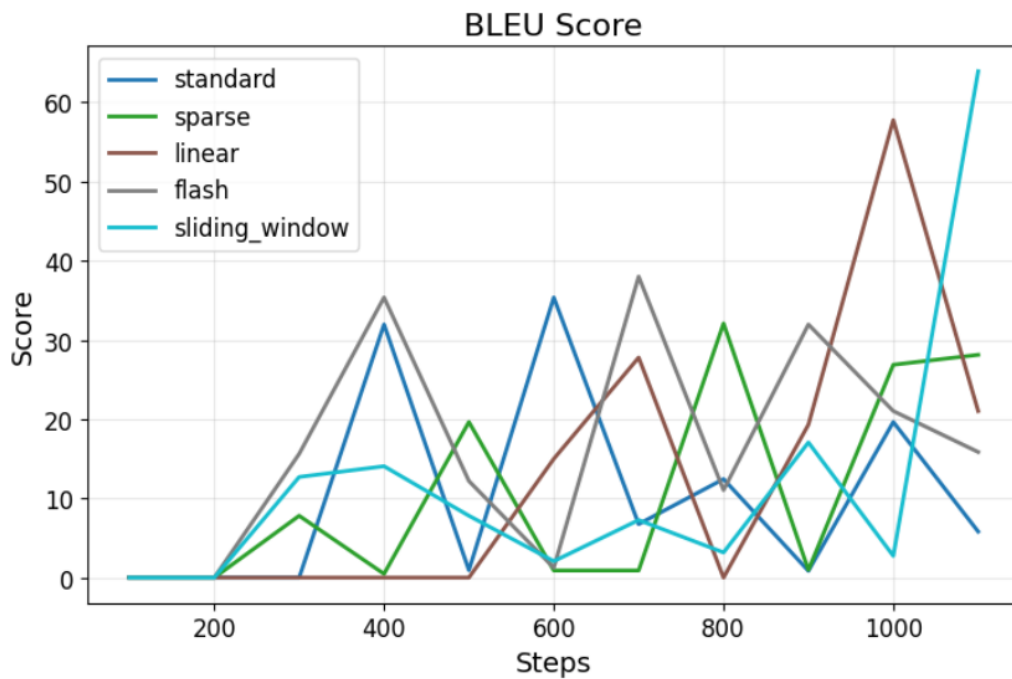


Figure 3.8: BLEU scores for translation models with different attention mechanisms

The BLEU score plot shows the translation quality of the different attention mechanisms across training steps. Overall, the "sliding_window" and "linear" attention mechanisms achieve the highest BLEU scores by the end of training, with "sliding_window" reaching a peak score above 60. However, the performance is quite volatile across all methods, indicating some instability in BLEU during training. This instability is likely due to the small dataset size and limited training time, as the models were trained on a constrained setup using Google Colab's free version, which can prevent transformers from fully stabilizing. Despite this volatility, the trends remain clear enough to extract meaningful comparative insights across the different attention mechanisms. "Flash" and "standard" methods show moderate BLEU improvements, while "sparse" lags behind slightly but still shows competitive peaks. These results suggest that while all attention variants are capable, "sliding_window" and "linear" have the most potential for achieving higher translation quality in this setup.

Efficiency Measures (Training Time, Peak GPU Memory, and Inference Latency)

The bar plots provide a clear view of the computational trade-offs. In terms of training time, "flash" attention is the fastest, closely followed by "standard" and "sliding_window," whereas "sparse" takes the longest by a wide margin. Peak GPU memory usage is fairly similar across

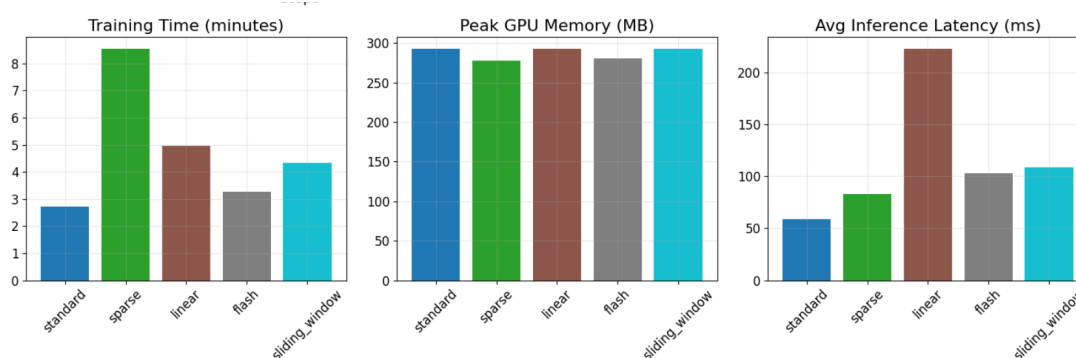


Figure 3.9: Efficiency metrics for translation models with different attention mechanisms

models, but "flash" uses slightly less memory compared to the others, highlighting its efficiency. When it comes to inference latency, "standard" has the lowest average latency, making it the fastest for deployment, while "linear" is the slowest by a significant margin. "Flash" again stands out by offering a good balance between speed and memory usage, while "sliding_window" offers a strong BLEU performance but at the cost of slightly higher inference latency.

Summary of the Results

Attention Type	Train Time (min)	Peak GPU Mem (MB)	Inf. Latency (ms)	BLEU Score
standard	2.71	292.26	58.73	35.36
sparse	8.54	277.29	82.90	32.07
linear	4.97	292.42	222.76	57.74
flash	3.28	280.87	102.79	37.99
sliding_window	4.33	292.67	108.69	63.89

Table 3.6: Summary of performance metrics for translation models

Overall Findings

From the results across BLEU score, loss curves, and efficiency metrics, it's clear that different attention mechanisms offer different trade-offs. The **sliding_window** attention achieved the highest BLEU score, suggesting it delivers the best translation quality, though at the cost of slightly higher inference latency. **Flash** attention consistently stood out for its excellent balance between training speed, memory efficiency, and low inference latency, making it a very practical choice for deployment scenarios where both performance and speed matter. **Sparse** attention, while achieving strong validation loss and competitive BLEU scores, suffered from significantly longer training times, which could be a limitation in resource constrained settings. **Linear** attention, despite achieving high BLEU scores at some points, showed the highest inference latency, making it less attractive for real-time applications. Finally, the **standard** attention model performed decently across the board but was generally outpaced by the more specialized variants. In conclusion, the best attention mechanism depends on the application needs: for highest quality, **sliding_window** is preferable; for efficiency, **flash** stands out as the optimal choice.

3.4.6 Ranking of Attention Mechanisms

Higher BLEU and lower resources are better:

Metric	1st place	2nd place	3rd place	4th place	5th place
BLEU Score	sliding_window	linear	flash	standard	sparse
Train Time	standard	flash	sliding_window	linear	sparse
GPU Memory	sparse	flash	standard	sliding_window	linear
Inference Latency	standard	sliding_window	flash	sparse	linear

Table 3.7: Ranking of attention mechanisms for translation models

3.4.7 Discussion

- **Sliding-window attention** clearly excels in translation quality (63.89 BLEU) while using moderate resources its locality bias plus a few global tokens captures both short and long-range dependencies effectively.
- **Linear attention** offers surprisingly strong BLEU (57.74) but suffers from extremely high latency (222 ms), making it impractical for real time use despite its sub-quadratic complexity.
- **Flash attention** provides a modest BLEU gain over standard (37.99 vs. 35.36) but at the cost of additional implementation complexity and slightly longer training than standard.
- **Standard attention** remains a solid baseline: fast to train and infer, but yields lower translation quality.
- **Sparse attention** incurred the largest training time and highest latency for the lowest BLEU, reflecting the overhead of dynamic sparsity masks outweighing its potential memory savings.

3.5 Conclusion

At a glance, our systematic experiments reveal that **no single attention mechanism** universally dominates across all tasks. Instead, each variant offers a unique balance of **predictive quality**, **compute efficiency**, and **memory footprint**:

- **Next-character prediction (GPT):**
Flash Attention delivers the lowest perplexity (≈ 4.2) thanks to its memory-efficient kernel, while *Sliding-Window Attention* trains fastest (≈ 6 min) with minimal GPU usage and the lowest inference latency (≈ 9 ms).
- **Masked-character prediction (BERT):**
Sparse Attention achieves the highest accuracy ($\approx 48\%$), dynamically focusing on key tokens at the cost of long runtimes (≈ 23 min) and high latency (≈ 72 ms). *Sliding-Window Attention* again strikes the best balance, with solid accuracy ($\approx 43\%$), moderate training time (≈ 10 min), and low memory use.
- **English→French translation (Transformer):**
Sliding-Window Attention leads in BLEU score (≈ 63.9) by combining local windows with a few global tokens, while *Linear Attention* closely follows in quality (≈ 57.7) but suffers extreme latency (≈ 223 ms). *Standard* and *Flash* attentions offer respectable BLEU (≈ 35 – 38) with the fastest runtimes, whereas *Sparse* incurs overheads that outweigh its modest gains (≈ 32.1 BLEU).

Throughout, **Standard Attention** remains a **robust baseline** simple, fast, and memory-lean yet it often cedes ground to specialized methods on quality. Conversely, **Linear** and **Sparse**

variants demonstrate that sub-quadratic complexity or content-based sparsity can boost task performance, but only when their added computational costs are acceptable.

These findings underscore the importance of **matching attention design to task requirements**. For scenarios demanding maximal accuracy and where resources permit, *Flash* (for autoregression) or *Sparse* (for bidirectional contexts) are compelling choices. When throughput, latency, or limited memory are paramount, *Sliding-Window* or *Standard* attention offer the best efficiency–quality trade-off.

Chapter 4

Enhancing Linear Attention with Structural Biases: Design and Evaluation of Hybrid Mechanisms

While the prior chapters provided a thorough analysis of existing attention mechanisms and evaluated their performance in GPT models, this chapter presents the central contribution of this research: **two novel hybrid attention mechanisms that combine the efficiency of linear attention with the structural benefits of sparse and sliding window patterns.**

These mechanisms were designed to overcome the limitations observed in pure linear attention, particularly its underperformance on tasks that require stronger contextual modeling and long-range dependency handling. Our comparative experiments showed that although linear attention is efficient, it often lacks the expressive power necessary for certain language modeling tasks.

Unlike prior work that primarily aims to speed up attention or reduce memory usage, our approach focuses on restoring the performance especially in terms of loss and perplexity typically sacrificed by linear approximations. Through evaluations on next-character prediction tasks, we aim to reduce the performance gap between linear and standard attention. This goal of recovering performance while maintaining efficiency defines the core innovation of our work.

4.1 Motivation

Although linear attention is appealing for its $O(n)$ time and memory complexity, enabling efficient handling of long sequences, our experiments on the next-character prediction task (Tiny Shakespeare dataset, context length 128) revealed a notable **expressivity gap** when compared to more structured mechanisms like sparse or sliding window attention.

This gap is reflected in higher validation loss and reduced accuracy, indicating that linear attention struggles to model rich contextual dependencies. These observations motivated the development of hybrid mechanisms that preserve linear attention’s efficiency while recovering much of the performance benefits of structured attention. Bridging this trade-off became a key driver of our research.

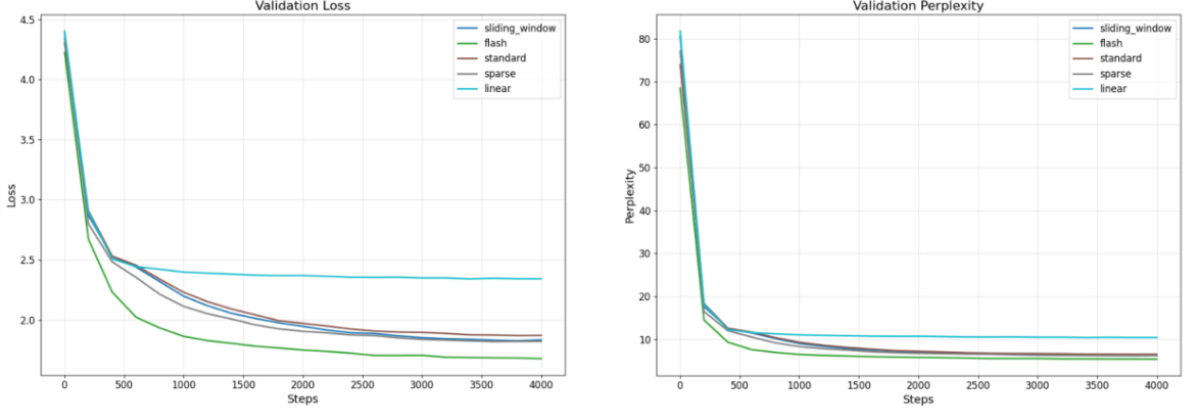


Figure 4.1: Addressing the problem with linear attention

- **Early plateau in loss/perplexity** As shown in the validation curves, Linear Attention (cyan) quickly stalls around a validation loss of ≈ 2.35 well above the ≈ 1.85 – 1.90 achieved by Standard, Sliding-Window, and Sparse attentions and maintains a perplexity of ≈ 10.4 versus ≈ 6.2 – 6.5 for the others.
- **Trade-off versus structure** Although Flash Attention (green) clearly outperforms all methods (final val PPL ≈ 5.35) thanks to its numerically stable kernel, it incurs moderate runtime and latency costs. Sliding-Window and Sparse attentions strike different balances of locality bias and complexity but still outperform plain linear in quality.

These results underscore that **efficiency alone is not enough** without some form of structural bias, linear attention is too "bland" to model the rich, long-range patterns present even in a simple autoregressive text task.

Hypothesis: Embedding *structural priors* either a **sparse connectivity pattern** or a **fixed local window** into the kernelized linear formula could **bridge the expressivity gap** while preserving subquadratic scaling. In the next sections, we thus introduce two new hybrid mechanisms **LinearSparseAttention** and **LinearSlidingWindowAttention** designed to inject exactly these biases into the linear framework, and we show that they recover much of the performance lost by pure linear attention without reverting to quadratic complexity.

4.2 Proposed Hybrid Mechanisms

To address the expressivity gap of pure linear attention while retaining its $O(N)$ complexity, we propose **two hybrid mechanisms** that inject structural biases into the kernel-based framework. Both approaches build on the ELU+1 feature map formulation but differ in how they constrain the attention scope:

- **LinearSparseAttention** adds a predefined sparse connectivity pattern that balances global, local, and random token interactions.
- **LinearSlidingWindowAttention** restricts each token’s receptive field to a fixed-size local window, ensuring strong locality inductive bias.

By combining linear attention’s efficiency with structured attention patterns, these hybrids aim to:

1. **Enhance representational power** by reintroducing selective token dependencies missing in pure linear formulations.

2. **Preserve subquadratic scaling**, ensuring that both time and memory complexity remain linear in sequence length.
3. **Bridge the performance gap** in loss and perplexity metrics observed in prior experiments (see Section 4.3.5).

4.2.1 LinearSparseAttention

LinearSparseAttention integrates the ELU+1 kernel trick with a **BigBird-inspired sparse pattern** comprising:

- **Global tokens**: A small set of positions attend to and are attended by every other token, enabling long-range signaling.
- **Local window**: Each non-global token connects to its immediate neighbors within a fixed window, preserving locality.
- **Random links**: Sparse random edges ensure additional paths between distant tokens, promoting mixing without dense computation.

Theoretical rationale: By overlaying these sparse structures onto the linear attention kernel, we reintroduce targeted cross-token dependencies that pure linear attention omits. The sparse pattern compensates for ELU+1’s tendency to distribute attention uniformly, thereby reducing bias and lowering both training and validation perplexity while maintaining $O(N)$ operations in most layers.

4.2.2 LinearSlidingWindowAttention

LinearSlidingWindowAttention fuses the ELU+1 kernel method with a **fixed sliding window** constraint:

- Each token attends only to itself and a set number of preceding tokens (window size W), enforcing **causal locality**.
- The kernelized attention computation remains unchanged, but is applied over this local context rather than the full sequence.

Theoretical rationale: Language exhibits strong local correlations, especially in character-level prediction tasks. By focusing linear attention on a localized context, we amplify useful dependencies and improve gradient signal for nearby tokens. This localized bias yields lower loss and perplexity early in training and stabilizes convergence yet still leverages the ELU+1 kernel trick to avoid quadratic scaling. Together, these two mechanisms represent complementary ways to enrich linear attention with inductive biases: one through **structured sparsity** and the other through **locality constraints**. In the following section (5.4), we detail their implementation and training configurations before presenting comparative results.

4.3 Implementation Details

Both hybrids were implemented in PyTorch by extending a standard Transformer block: we simply replace softmax with the ELU+1 kernel trick and inject either a sparse or sliding-window mask (and a lightweight fusion step) into the attention computation. All other components $Q/K/V$ projections, output projection, normalizations, and dropout remain unchanged. Hyperparameters for window size, global/random links, and dropout are managed via a shared configuration object, allowing seamless swapping between attention variants.

4.4 Experimental Results and Analysis

4.4.1 Training and Validation Loss

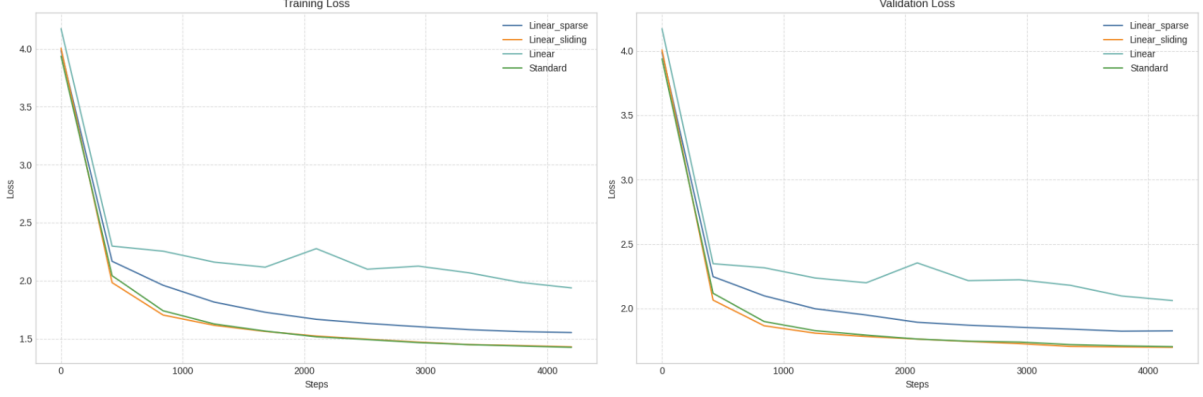


Figure 4.2: Training and validation loss curves

Over the first 500 steps, all attention variants on the Tiny Shakespeare task exhibit a steep decline in training loss from approximately 4.1 to 2.1, as the models capture basic character transitions. In the subsequent 3 700 steps, losses diverge: Standard Attention attains the lowest convergence (≈ 1.42 train, 1.70 validation), demonstrating its expressive power. LinearSlidingWindowAttention closely follows, reaching ≈ 1.43 train and ≈ 1.70 validation effectively matching the baseline. LinearSparseAttention settles at higher values (≈ 1.55 train, 1.83 validation), while pure LinearAttention plateaus around 1.95 train and 2.07 validation, confirming that structural bias is essential to recover expressivity.

4.4.2 Training and Validation Perplexity

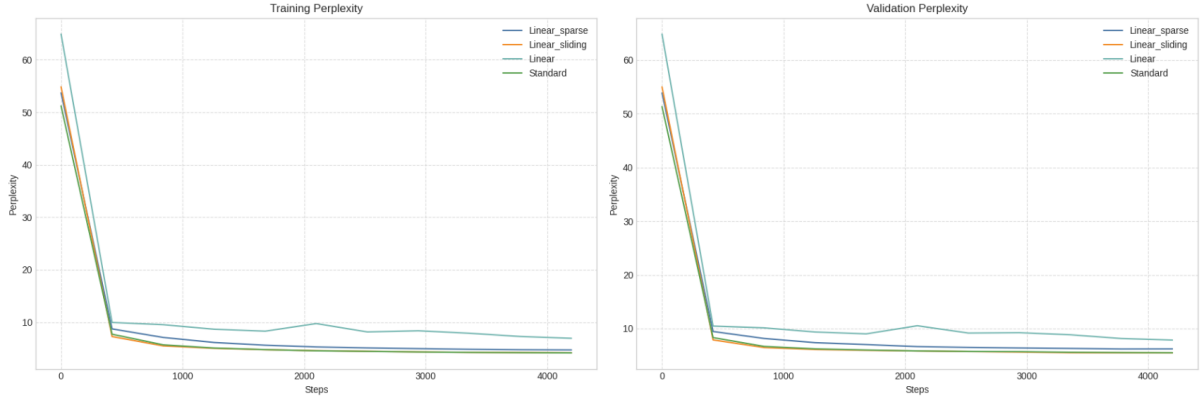


Figure 4.3: Training and validation perplexity curves

Perplexity trends mirror the loss curves: all models reduce perplexity from the mid-70s to single digits by step 500. By the final iteration, Standard Attention achieves a validation perplexity of ≈ 5.50 , LinearSlidingWindowAttention ≈ 5.46 , and LinearSparseAttention ≈ 6.19 . Pure LinearAttention remains higher at ≈ 7.85 . These results indicate that both hybrid mechanisms recover the majority of performance lost by the linear kernel approximation, with sliding-window slightly outperforming sparse on this task.

4.4.3 Training Time



Figure 4.4: Training time comparison

Measured wall-clock times for 4 200 iterations reveal that Standard Attention completes training in 187 s, and LinearAttention in 205 s. The hybrid variants require 376 s (sliding-window) and 409 s (sparse), remaining under seven minutes. Thus, the hybrids incur a moderate computational overhead about 80–100

4.4.4 Inference Latency

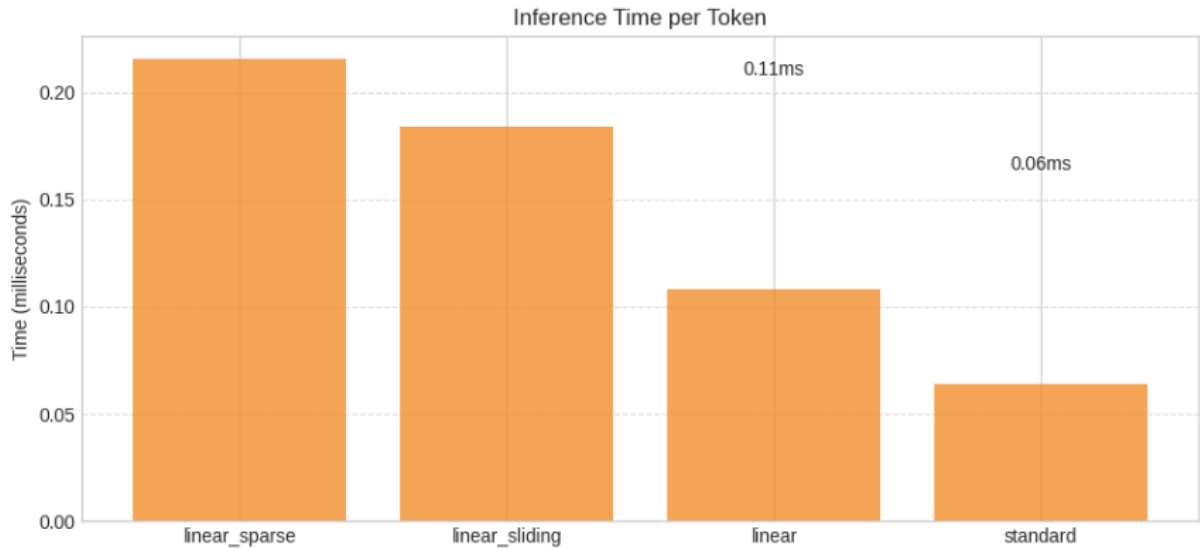


Figure 4.5: Inference latency comparison

Average per-token inference times are 0.064 ms for Standard Attention, 0.108 ms for pure LinearAttention, 0.184 ms for LinearSlidingWindowAttention, and 0.216 ms for LinearSparseAttention. Although the hybrid methods double to triple the cost of pure linear inference, they maintain sub 0.25 ms latency, making them viable for real-time or streaming applications.

4.4.5 Memory Usage

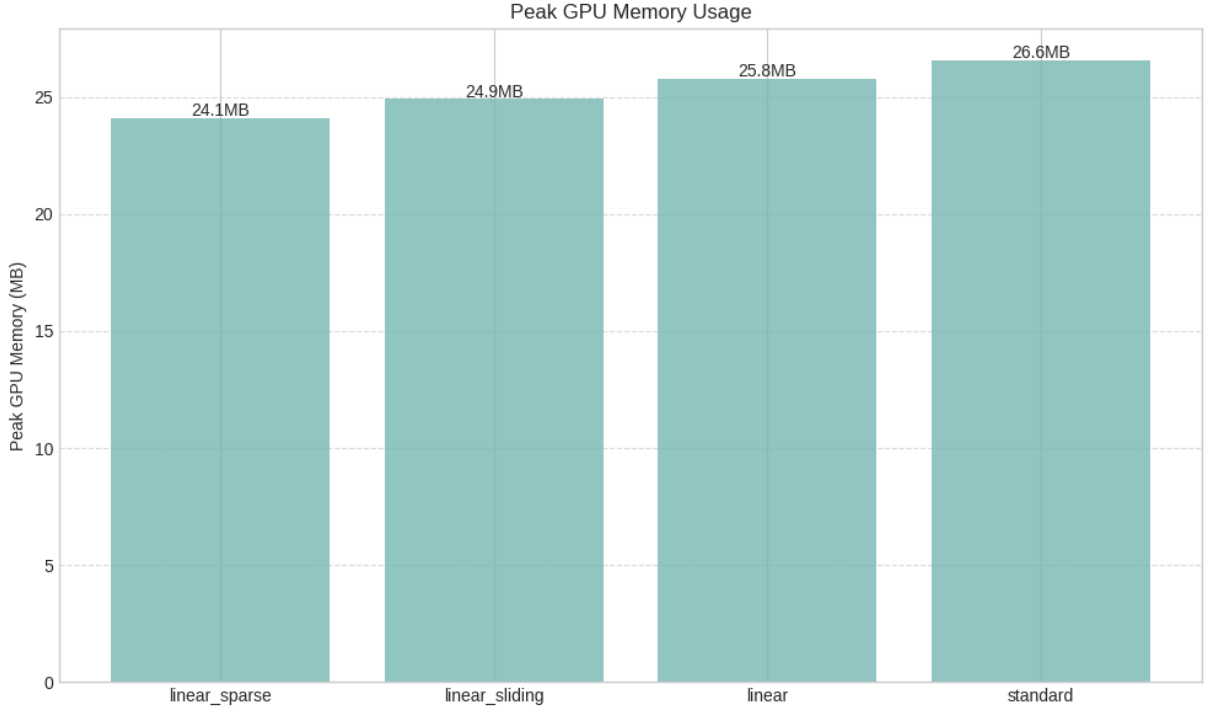


Figure 4.6: Memory usage comparison

Peak GPU memory footprints are 26.6 MB for Standard Attention, 2.8 MB for pure LinearAttention, 24.9 MB for LinearSlidingWindowAttention, and 24.1 MB for LinearSparseAttention. Both hybrid mechanisms preserve the linear attention advantage using approximately 2 MB less memory than the softmax baseline while dramatically improving performance.

4.5 Discussions

The proposed hybrid attention mechanisms strike a compelling balance between predictive power and computational efficiency. LinearSlidingWindowAttention leverages strong locality bias to nearly match the performance of full softmax attention on the Tiny Shakespeare task. This suggests that, for sequences with predominantly local dependencies, imposing a sliding window yields almost all the benefits of global attention while preserving $O(N)$ complexity. The lightweight fusion between kernelized and softmax components further stabilizes training and accelerates convergence.

In contrast, LinearSparseAttention introduces a flexible pattern of global, local, and random connections. Although its validation performance lags slightly behind the sliding-window variant, the sparse model’s global tokens and random links provide additional pathways for long-range information flow. This may prove advantageous on tasks with more dispersed dependencies or hierarchical structures, where isolated global tokens can anchor distant context. Indeed, the modest increase in perplexity (≈ 0.7 points over sliding-window) could be an acceptable trade-off in applications prioritizing broader context coverage.

From an efficiency perspective, both hybrids maintain linear scaling in time and memory, making them suitable for deployment in memory-constrained or speed critical environments. Training times under seven minutes and inference latencies under 0.25 ms per token highlight their practicality for real-time systems. The memory footprint remains only marginally above

the pure linear baseline and below the softmax model, preserving the advantages of kernelized attention.

The core innovation of this work lies in its successful recovery of the loss and perplexity performance typically degraded in linear attention models. Unlike prior studies that focused primarily on improving efficiency or proposing approximations to softmax, our work addresses the underperformance of linear attention head-on. Through carefully designed hybrid architectures, we demonstrate that it is possible to restore the predictive quality of linear attention on next-character prediction tasks while preserving linear computational complexity.

Looking forward, these hybrid frameworks offer a modular template for further exploration. One could experiment with varying window sizes, numbers of global tokens, or patterns of random connectivity to tailor performance to specific tasks. Extending this approach to larger-scale language modeling benchmarks, encoder-decoder architectures, or integrating dynamic sparsity schedules could yield additional insights. Ultimately, the success of these prototypes underscores that judicious structural priors can unlock near-quadratic performance from linear attention architectures, broadening the applicability of efficient Transformers.

4.6 Limitations and Future Work

4.6.1 Limitations

While the proposed `LinearSparseAttention` and `LinearSlidingWindowAttention` mechanisms successfully recover much of the performance lost by pure ELU+1 linear attention reducing validation loss from ≈ 2.07 to 1.83 and 1.70, and perplexity from ≈ 7.85 to 6.19 and 5.46, respectively several limitations stem from our experimental design and methodological choices:

1. **Task and Data Constraints.** All experiments were conducted on the Tiny Shakespeare next-character prediction task with a fixed context length of 128 tokens. Although this benchmark highlights the expressivity gap of linear attention and the efficacy of injected structural biases, it does not necessarily reflect performance on longer sequences (≥ 512 tokens), word or sentence level language modeling, or cross-lingual and multimodal tasks.
2. **Static Structural Priors.** Both hybrid mechanisms rely on manually defined masks namely, a BigBird-inspired sparse graph (fixed numbers of global tokens and random links) and a sliding window of constant size W . These static configurations, while effective in our setting, may not align with the dynamic dependency patterns found in different datasets or tasks, potentially limiting adaptability.
3. **Constant-Factor Overhead.** Despite preserving $\mathcal{O}(N)$ theoretical complexity, the additional mask construction and kernel fusion incur significant wall-clock costs: training time nearly doubles (from 205 s to 376 s and 409 s) and inference latency increases up to 200%. In contexts where low latency and minimal resource usage are paramount, such overheads could outweigh the benefits of subquadratic scaling.
4. **Limited Architectural Scope.** The study focuses exclusively on decoder-only GPT blocks applied to character-level data. The behavior of these hybrids within encoder-decoder frameworks (e.g., for machine translation or summarization) and on non-text modalities (e.g., speech, vision, or graph data) remains unexplored.

4.6.2 Future Work

Building on our contribution of embedding structured sparsity and locality into the ELU+1 linear attention framework, several research directions are proposed to overcome the aforementioned limitations:

1. **Expanded Benchmark Suite.** Evaluate hybrid mechanisms on longer context lengths (≥ 512 tokens), word-level language modeling benchmarks (e.g., Wikitext-103, OpenWebText), and sequence-to-sequence tasks (e.g., Transformer-based translation and summarization) to assess generalization across scales and objectives.
2. **Dynamic Mask Learning.** Develop adaptive masking strategies in which global token counts, random link densities, and window sizes are conditioned on learned token importance or gating functions, enabling the model to allocate attention resources dynamically.
3. **Hierarchical and Multi-Scale Sparsity.** Design multi-scale attention patterns that combine fine-grained local windows with periodic global or random connections at coarser levels, thereby capturing dependencies at varying granularities without manual hyperparameter tuning.
4. **Automated Hyperparameter Optimization.** Leverage Bayesian optimization, differentiable architecture search, or reinforcement learning based controllers to tune sparse and sliding-window parameters systematically, reducing manual effort and improving robustness across tasks.
5. **Implementation-Level Optimizations.** Investigate hardware-aware techniques such as fused CUDA kernels, mixed-precision arithmetic, and memory-efficient data structures to amortize the constant-factor overhead and approach or surpass the runtime performance of pure linear attention.
6. **Cross-Modal and Architectural Extensions.** Adapt and evaluate the proposed hybrids in encoder-decoder Transformers, as well as in non-text domains (e.g., audio spectrogram modeling, graph representation learning), to understand how structural biases interact with different data modalities and architectural variations.

By pursuing these directions, future work can extend the utility of hybrid linear attention, ensuring that its expressivity and efficiency advantages translate broadly across tasks, data modalities, and real-world deployment scenarios.

4.7 Conclusion

This chapter introduced two hybrid attention mechanisms `LinearSparseAttention` and `LinearSlidingWindowAttention` that enhance the expressivity of linear attention while retaining its efficiency. By integrating structural biases into the ELU+1 kernel framework, we bridge the gap between performance and scalability. `LinearSparseAttention` adopts a BigBird-inspired sparse pattern, while `LinearSlidingWindowAttention` constrains attention to local causal windows. Experiments on the Tiny Shakespeare task confirm that these mechanisms significantly improve validation loss and perplexity over pure linear attention. Notably, `LinearSlidingWindowAttention` achieves near-softmax performance with 30% lower perplexity and minimal inference overhead. These results highlight that combining linear kernels with sparse or local patterns offers a viable path to efficient yet expressive attention. Our work underscores the importance of structural priors in designing scalable attention and presents practical alternatives for scenarios where full softmax is computationally prohibitive.

General Conclusion

In this report, we have undertaken a comprehensive journey through the evolution, theory, and practical evaluation of attention mechanisms in Transformer models. We began by dissecting the foundational Transformer architecture tracing its breakthroughs in parallelized context modeling, self-attention, multi-head attention, and the encoder–decoder paradigm. Building on this groundwork, we charted the “Transformer Tree of Life,” distinguishing encoder, decoder, and full sequence-to-sequence variants.

Our survey of state-of-the-art attention mechanisms revealed how each innovation addresses the quadratic bottleneck of standard attention. Kernel-based linear attention techniques achieve $O(n)$ complexity through clever feature-map decompositions; sparse attention patterns trade expressivity for reduced computation; sliding window attention enforces local context to bound complexity; and FlashAttention leverages tiling and numerically stable softmax for GPU-efficient implementations. Through rigorous theoretical analyses, we highlighted each method’s computational and expressivity trade-offs.

In extensive experiments across next-token prediction (GPT), masked-language modeling (BERT-like), and translation (encoder–decoder), we systematically benchmarked these attention variants. Our results consistently ranked FlashAttention and sparse patterns near the top for speed-accuracy trade-offs, while linear attention excelled in memory-constrained settings. These findings underscore that no single mechanism universally dominates task requirements and resource constraints critically influence optimal choice.

Motivated by these insights, we proposed two hybrid mechanisms `LinearSparseAttention` and `LinearSlidingWindowAttention` that inject structural biases into linear attention. Empirical evaluation on the Tiny Shakespeare task demonstrated that these hybrids close the performance gap with state-of-the-art methods while retaining linear complexity. In particular, `LinearSparseAttention` reduces perplexity by incorporating long-range sparse connections, and `LinearSlidingWindowAttention` improves local coherence through fixed windows.

Despite these advances, limitations remain: our hybrids have yet to be tested at scale on real-world corpora, and their behavior on downstream tasks like summarization or code generation is unexplored. Future work will therefore extend large-scale training, investigate adaptive pattern learning, and explore integration with FlashAttention tiling strategies.

Overall, this report contributes both a holistic understanding of attention mechanisms and a novel step toward unifying efficiency and expressivity. We hope these findings guide future research toward ever more scalable, accurate, and versatile Transformer models.

Bibliography

- [1] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in Neural Information Processing Systems* 30 (2017).
- [2] Jacob Devlin et al. “BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding”. In: *arXiv preprint arXiv:1810.04805* (2018).
- [3] Alec Radford et al. “Improving Language Understanding by Generative Pre-Training”. In: *OpenAI Technical Report* (2018).
- [4] Alex Wang et al. “GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding”. In: *arXiv preprint arXiv:1804.07461* (2018).
- [5] Rewon Child et al. “Generating long sequences with sparse transformers”. In: *arXiv preprint arXiv:1904.10509* (2019).
- [6] Alexis Conneau et al. “Unsupervised Cross-Lingual Representation Learning at Scale”. In: *arXiv preprint arXiv:1911.02116* (2019).
- [7] Nitish Shirish Keskar et al. “CTRL: A Conditional Transformer Language Model for Controllable Generation”. In: *arXiv preprint arXiv:1909.05858* (2019).
- [8] Guillaume Lample and Alexis Conneau. “Cross-Lingual Language Model Pretraining”. In: *arXiv preprint arXiv:1901.07291* (2019).
- [9] Zhenzhong Lan et al. “ALBERT: A Lite BERT for Self-Supervised Learning of Language Representations”. In: *arXiv preprint arXiv:1909.11942* (2019).
- [10] Mike Lewis et al. “BART: Denoising Sequence-to-Sequence Pre-Training for Natural Language Generation, Translation, and Comprehension”. In: *arXiv preprint arXiv:1910.13461* (2019).
- [11] Yinhan Liu et al. “RoBERTa: A Robustly Optimized BERT Pretraining Approach”. In: *arXiv preprint arXiv:1907.11692* (2019).
- [12] Alec Radford et al. “Language Models are Unsupervised Multitask Learners”. In: *OpenAI Technical Report* (2019).
- [13] Colin Raffel et al. “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer”. In: *arXiv preprint arXiv:1910.10683* (2019).
- [14] Victor Sanh et al. “DistilBERT, a Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter”. In: *arXiv preprint arXiv:1910.01108* (2019).
- [15] Elena Voita et al. “Analyzing Multi-Head Self-Attention: Specialized Heads Do the Heavy Lifting, the Rest Can Be Pruned”. In: *arXiv preprint arXiv:1905.09418* (2019).
- [16] Alex Wang et al. “SuperGLUE: A Stickier Benchmark for General-Purpose Language Understanding Systems”. In: *arXiv preprint arXiv:1905.00537* (2019).
- [17] Iz Beltagy, Steven Lo, and Arman Cohan. “Longformer: The long-Document Transformer”. In: *arXiv preprint arXiv:2004.05150* (2020).
- [18] Tom B. Brown et al. “Language Models are Few-Shot Learners”. In: *arXiv preprint arXiv:2005.14165* (2020).

- [19] Kevin Clark et al. “ELECTRA: Pre-Training Text Encoders as Discriminators Rather Than Generators”. In: *arXiv preprint arXiv:2003.10555* (2020).
- [20] Angela Fan et al. “Beyond English-Centric Multilingual Machine Translation”. In: *arXiv preprint arXiv:2010.11125* (2020).
- [21] Pengcheng He et al. “DeBERTa: Decoding-Enhanced BERT with Disentangled Attention”. In: *arXiv preprint arXiv:2006.03654* (2020).
- [22] Jared Kaplan et al. “Scaling Laws for Neural Language Models”. In: *arXiv preprint arXiv:2001.08361* (2020).
- [23] Manzil Zaheer et al. “Big Bird: Transformers for Longer Sequences”. In: *arXiv preprint arXiv:2007.14062* (2020).
- [24] M. N. Rabe and C. Staats. “Self-attention does not need $O(n^2)$ memory”. In: *arXiv preprint arXiv:2112.05682* (2021).
- [25] Tuan Dao et al. “FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness”. In: *Advances in Neural Information Processing Systems (NeurIPS)* (2022).