Journal **XX** (XXXX) XXXXXX https://doi.org/XXXX/XXXX

WatGPT: A Quant-Based LLM

Abstract

In this paper, we introduce WatGPT, a specialized Large Language Model (LLM) designed for handling a range of quantitative finance tasks. Leveraging the latest advancements in model architectures such as Structured State Spaces (SSMs), Receptance Weighted Key Value (RWKV) RNNs, and traditional Transformers, WatGPT aims to provide robust solutions for tasks including portfolio optimization, risk management, and algorithmic trading. Unlike general-purpose models, WatGPT integrates domain-specific knowledge, which significantly enhances its performance on specialized tasks.

Keywords: WatGPT, Large Language Model (LLM), Quantitative Finance, NLP (Natural Language Processing), Financial Analysis, Risk Management, Algorithmic Trading, Structured State Spaces (SSMs), Receptance Weighted Key Value (RWKV) RNNs, Transformers, HiPPO

1 Introduction

The integration of NLP technologies in financial applications has seen exponential growth, driven by the capabilities of LLMs to understand and generate human-like text. However, the unique challenges of the financial domain—such as the need for precise risk assessment and real-time decision-making—demand specialized solutions. WatGPT is designed to address these challenges by incorporating financial domain expertise directly into its architecture and training process, offering improved accuracy and efficiency on finance-specific tasks.

1.1 Project Scope and Goals

The development of WatGPT was guided by the need to create a more focused and effective tool for financial analysis and decision-making. The primary objectives for WatGPT include code adaptation—adjusting the code of in-house models to better suit specified tasks, either through direct modifications by user command or by autonomously adapting to fulfill task requirements, and finance advisory—providing expert financial advice through a conversational model that leverages in-house models and data, focusing on user queries and contextual understanding.

Given the broad potential applications and the importance of maintaining manageable scope, the project emphasizes the creation of a model that integrates seamlessly with existing financial analysis tools and datasets. It focuses on a large-language model that plays the role of an expert financial advisor and helper. Code adaptation was neglected as core functionality of this model for several reasons—the LLM would require new architecture for code-style corpus, the LLM would not be as strong in the financial domain, and a code synthesizer model could be well achieved through a separate, better tailored model for our needs. As a result, separate projects exist for code adaption and data augmentation.

2 Dataset

2.1 Data Criterion

WatGPT should be trained on solely finance and quantitative text and numerical data, as we want to constraint the pre-context of the LLM to these two realms.

Textual data includes any of the following—regulatory filings and disclosures (annual and quarterly reports, SEC filings), financial news and analysis (articles and news updates, analyst reports), corporate communications (earnings call transcripts, press releases), and legal documents (contracts and agreements, legal proceedings).

Numerical data includes—market data (stock prices, indices, exchange rates, commodities prices), financial statements (balance sheets, income statements, cash flow statements), economic data (GDP figures, employment statistics, consumer and producer prices), trading data (transaction data, order book data), derivative data (options and futures), and credit data (credit scores and ratings, loan and mortgage data).

2.2 Datasets

The Pile (184B tokens)—The pile is favored for its extensive data cleaning and preprocessing. It covers numerous domains, so we can pick and choose. It contains deliberate duplications to reflect content quality so de-duplication will significantly reduces size. We can train the tokenizer on this dataset.

C4 (138B tokens)—Supports different processing compared to The Pile. Noted for high-quality natural language content and a diverse web domain distribution, though it includes a substantial amount of patent-related data.

Wikipedia (24B tokens)—Can include up-to-date English Wikipedia pages (as of July 1, 2022) to ensure the factuality of the model. The Wikipedia data, characterized by a higher-than-average markup reflected in its inefficient tokenization, suggests potential areas for improvement in data cleaning for future training.

Polygon API—A robust financial data platform that furnishes a comprehensive array of real-time and historical data across various asset classes, including stocks, Forex, and cryptocurrencies, catering to the needs of developers, financial analysts, and traders. It provides crucial real-time market data such as price updates, trade details, and quote data, alongside historical market data essential for backtesting trading strategies and conducting thorough financial analyses. The API encompasses a wide range of data points from stock prices to corporate earnings, and detailed transaction data for cryptocurrencies, ensuring depth and breadth in market coverage. Additionally, it aggregates data from multiple sources to deliver accurate market insights and supports real-time data needs through WebSocket technology for high-frequency trading applications. Key data provided includes transaction-level ticks, aggregated bars, fundamental corporate data, comprehensive reference data, and potentially, sentiment analysis derived from various media sources.

3 Implementation

3.1 Architecture

WatGPT is built upon a hybrid architecture that incorporates elements of SSMs, RWKV RNNs, and Transformers. This design allows it to effectively process and analyze long sequences of financial data (SSM), manage high-frequency updates (RWKV RNN), and understand complex financial reports (Transformer).

3.1.1 Structured State Space (SSM) Models:

SSMs are utilized for their efficiency in modeling long-term dependencies and handling continuous-time data, crucial for tasks such as long-term market prediction and risk assessment. The focus with SSMs is primarily on risk assessment, portfolio optimization, and long-term market predictions due to their ability to excel with large temporal relationships over extended periods.

The model components include a state vector for representing the internal state of the system at some arbitrary time, an optional input vector which includes external inputs or control signals applied to the system, influencing state transitions, an output vector for observable outputs of the system, and a state transition function, which defines the evolution of the state vector, denoded commonly as the following:

$$x_t + 1 = \mathrm{A}x_t + \mathrm{B}u_t + w_t$$

where A is the state transition matrix, B maps the control inputs to the state space, and w_t is process noise.

The output function can generally be described as:

$$y_t = \mathrm{C}x_t + \mathrm{D}u_t + v_t$$

where C is the output matrix, D maps the control inputs directly to the outputs, and v_t is observation noise.

HiPPO Integration: We can leverage HiPPO in our SSM framework to improve on the state vector component.

Polynomial State Representation—HiPPO transforms the state representation into a system where the state vector represents coefficients of a polynomial basis. This polynomial representation captures the dynamics of the system over continuous time, which is discretized for practical computation.

State Transition Dynamics—The HiPPO approach modifies the traditional state transition matrix to implement an operator that projects the polynomial representation forward in time. This operator can handle rapid changes in input signals by dynamically adjusting the polynomial coefficients, thereby capturing both high-frequency components and long-range dependencies effectively.

Memory and Computation—HiPPO provides a mathematical framework for efficiently computing the impact of historical data on the current state through its polynomial projection method. This aspect is crucial for efficiently managing memory and computation in models dealing with long input sequences.

Learning and Adaptation—In a HiPPO-based SSM, learning involves optimizing the parameters that define the polynomial projection, which can be done through gradient-based optimization techniques commonly used in machine learning.

3.1.2 Receptance Weighted Key Value (RWKV) RNNs:

RWKV RNNs enhance the model's ability to focus on relevant portions of data, improving computational efficiency and response accuracy, particularly beneficial for real-time trading algorithms.

Input Layer: The input layer defines how the input data will be represented (e.g., for sequential data like text, use embeddings to convert input tokens into dense vectors).

RNN Layer: The RNN layer defines the type of neural network cells used for computation and learning in the recurrent layer (e.g., LSTM, GRU). These cells should be capable of maintaining long-term dependencies.

Attention Module & Receptance: Each input token (or feature vector) at a position is transformed into three vectors: a key vector K, a query vector Q, and a value vector V, which are typically achieved through linear projection (multiplication by a trained weight matrix).

The attention score between a query and all keys is computed, often using the dot product, followed by a softmax operation to normalize the scores. The equation is generally given by:

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

where d_k is the dimensionality of the key vectors, used to scale the dot products to prevent extremely large values due to high dimensionality.

If we conceptualize "receptance" as a measure of relevance or responsiveness of the keys to the query, this could imply an additional weighting mechanism in the attention calculation.

We can define a "receptance" weight for each key, which quantifies its importance or relevance in the context being modeled, and integrate the receptance weights directly into the attention scores computation:

$$\mathrm{Attention}(\mathrm{Q},\mathrm{K},\mathrm{V},\mathrm{R}) = \mathrm{softmax}\left(\frac{\mathrm{Q}K^T + R}{\sqrt{d_k}}\right)\mathrm{V}$$

where R represents the matrix of receptance weights added to the dot product of the queries and keys.

Output Layer: Define the output layer based on the task (e.g., a softmax layer for classification).

3.1.3 Transformers:

The Transformer architecture provides robust performance in understanding and generating natural language, aiding in tasks like sentiment analysis and regulatory compliance.

Tokenization: The tokenizer is a common component found in systems where text pre-processing is essential (LLMs, compilers, etc) and is required for us as we preferablyw ant to pre-process the text from user prior to sending it into the transformer.

There are a few production-ready algorithms used today. Some examples include WordPiece and SentencePiece.

WordPiece—A tokenization algorithm that breaks text down into a set of known words, and for out-of-vocabulary words, it further breaks them down into subwords or word pieces. The algorithm starts with a base vocabulary of individual characters and iteratively adds the most beneficial token (combination of characters or character sequences) to the vocabulary based on certain criteria, such as maximizing the likelihood of the training data. WordPiece is notably used in models like BERT (Bidirectional Encoder Representations from Transformers) and others in the Transformer family.

SentencePiece—A tokenization library that treats the input text as a raw stream of Unicode characters and directly learns subword units (tokens) from this text. Unlike WordPiece, SentencePiece does not require pre-tokenization of the text into words. SentencePiece supports two subword algorithms: Byte Pair Encoding (BPE) and Unigram Language Model. It learns subword tokens directly from the raw text, considering the entire text sequence, which allows it to capture information beyond word boundaries. SentencePiece allows for a more flexible tokenization that is not biased by pre-existing notions of "words" and is effective across languages with different characteristics. It also simplifies the preprocessing pipeline by eliminating the need for language-specific tokenization rules.

There is also prevalence of succesful tokenizer implementations that do not use WordPiece or SentencePiece and instead use a Unigram tokenizer, which is followed by the demonstrated efficacy in studies such as Kudo and Richardson (2018) and Bostrom and Durrett (2020). Mirroring the approach used by GPT-2 (Radford et al., 2019), the model processes data at the byte level, recognizing each of the 256 possible byte values as distinct tokens. This methodology enhances the handling of diverse data formats, from plain text to binary data, by treating them uniformly as sequences of byte-level tokens.

The tokenization process involves a pretokenization step where byte sequences are segmented according to intricate regular expressions, some examples being:

$$[A - Za - z] + |[0 - 9]|[\hat{}A - Za - z0 - 9] +$$

$$'(?:[sdmt]|ll|ve|re)|\ ?\backslash p\{L\} + |\ ?\backslash p\{N\} + |\ ?\ [\hat{}\backslash s\backslash p\{L\}\backslash p\{N\}] + |\backslash s + (?!\backslash S)|\backslash s +$$

$$\begin{aligned}&'(?\,i\!:[sdmt]|ll|ve|re)|[\text{\textasciicircum}\backslash r\backslash n\backslash p\{L\}\backslash p\{N\}]?+\backslash p\{L\}\\ &\quad +|\backslash p\{N\}\{1,3\}|\,?\,[\text{\textasciicircum}\backslash s\backslash p\{L\}\backslash p\{N\}]++[\backslash r\backslash n]\\ &\quad *|\backslash s*[\backslash r\backslash n]|\backslash s+(?!\backslash S)|\backslash s+\end{aligned}$$

which separates different character classes and includes spaces within alphabetic chunks to allow the formation of multi-word tokens.

Decoder-Only Design: A decoder-only architecture is sufficient because there is no need to encode data during pre-processing.

Due to this design, the attention mechanism is referred to as "causal" or "masked" self-attention because it prevents positions from attending to subsequent positions in the sequence, ensuring that the generation of an output element can only depend on known outputs.

Embeddings: The input text is converted into embeddings, which are dense vector representations that capture the semantic properties of the words. Although positional encodings can be added to embeddings to provide the model with information about the order of words in the sequence,

positional embeddings are not used for WatGPT for several reasons—the nature of the task does not require positional encodings, namely, due to the lack of order significance in text, it implies that it is simpler without them.

Formally—the initial representations H^0 are derived using an embedding matrix W_{em} and a standard basis vector e_{xt} for each token index xt, followed by LayerNorm (LNem) application:

$$\begin{aligned}\overline{h_0^t} &= W_{\text{em}} e_{x_t} \quad \forall t\\ h_0^t &= \text{LN}_{\text{em}}\left(\overline{h_0^t}\right) \quad \forall t\end{aligned}$$

No positional embeddings are applied, considering the functionality of ALiBi.

Layers: The decoder architecture comprises multiple layers, where each layer includes a self attention mechanism and feed forward network—for weighing the importance of different words in the input sequence relative to each other, for introducing non-linearity, and for enabling the model to learn complex patterns and relationships beyond the attention mechanism.

Formally—Each layer l in the model updates the sequence representation H^l using SelfAttention (SA) and FeedForward Network (FFN), combined with residual connections and LayerNorm applications:

$$\begin{aligned}\overline{H^\ell} &= H^{\ell-1} + SA^\ell\left(\text{LN}_{\text{in}}^\ell\left(H^{\ell-1}\right)\right) \quad \forall \ell\\ H^\ell &= \overline{H^\ell} + \text{FFN}^\ell\left(\text{LN}_{\text{at}}^\ell\left(\overline{H^\ell}\right)\right) \quad \forall \ell\end{aligned}$$

Self-Attention: ALiBi (Attention with Linear Biases) will be used in the self-attention component. It modifies traditional self-attention by introducing a linear bias based on the distance between token positions, reducing attention to distant tokens and enhancing focus on nearby tokens. This is achieved through a bias term

$$\mathrm{b(i,j)} = -\gamma \mid \mathrm{i} - \mathrm{j} \mid$$

where γ is a scaling factor.

This modification simplifies the implementation as the bias is directly incorporated into the attention scores before the softmax operation, without the need for learning additional parameters.

More formally—SelfAttention is computed for each attention head n, where queries Q, keys K, and values V are derived from the input, adjusted by learned biases, and combined with a distance-based ALiBi matrix A_n to focus attention based on token proximity:

$$\begin{aligned}Q_n &= W_{n,q}^\ell X + b_{n,q}^\ell \quad \forall n\\ K_n &= W_{n,k}^\ell X + b_{n,k}^\ell \quad \forall n\\ V_n &= W_{n,v}^\ell X + b_{n,v}^\ell \quad \forall n\\ \overline{S_n} &= A_n + \frac{K_n^\top Q_n}{\sqrt{D_n}} \quad \forall n\\ S_n &= drop_{p_{at}}\left(\text{softmax}\left(\overline{S_n} \odot M\right)\right) \quad \forall n\\ \overline{Y_n} &= V_n S_n \quad \forall n\\ Y &= drop_{p_h}\left(\sum_{n=1}^{N} U_n^\ell \overline{Y_n} + c^\ell\right)\end{aligned}$$

Feed Forward Network: The Feed-Forward Network (FFN), structured as a multi-layer perceptron (MLP), consists of two fully connected layers with a non-linear activation function, commonly ReLU or GELU. This FFN processes input features independently at each sequence position, enabling the transformation of self-attention outputs into more complex patterns. Each FFN is uniquely parameterized for every position, ensuring tailored and detailed feature processing without weight sharing across positions.

Formally—The FFN component applies a GELU activation to the linearly transformed input, followed by dropout and another linear transformation:

$$h = \text{gelu}\left(W_f^\ell x + b_f^\ell\right)$$

$$y = drop_{p_f}\left(U_f^{\ell} h + c_f^{\ell}\right)$$

Normalization: Layer Normalization (LayerNorm) significantly enhances training by normalizing input features independently across each sample, reducing internal covariate shift, and mitigating issues related to initial weight settings. This normalization facilitates the use of higher learning rates, supports the training of deeper networks, and reduces batch size dependency, promoting faster convergence and more stable learning. LayerNorm is implemented by computing the mean μ and standard deviation σ for each layer's inputs, then normalizing and scaling the outputs as follows:

$$y = \frac{x - \mu}{\sigma} \gamma - \beta$$

Here, γ and β are learnable parameters that adjust the scale and shift of the normalized data, ensuring flexibility in the model's feature representation. This mechanism is applied before each sub-layer within the Transformer architecture.

More formally—LayerNorm normalizes the layer outputs using mean $\mu(x)$ and variance $\sigma^2(x)$, scaled and shifted by trainable parameters γ and β:

$$y = \mathrm{LN}_\theta(x) = \frac{x - \mu(x)}{\sqrt{\sigma^2(x) + \epsilon}} \odot \gamma_\theta + \beta_\theta$$

3.2 Model Size

3.2.1 How do we determine model size?

We can use the Chinchilla scaling laws to help us determine the most optimal model size (for the models this applies to) depending on the anount of data we have and the training bandwidth (time and computational resources) we have.

The Chinchilla laws refer to research findings that optimize the trade-off between the size of a language model and the amount of training data used, particularly focusing on the effectiveness of model training relative to computational efficiency. This approach was developed based on empirical evidence showing that larger models do not always yield proportionally better performance unless paired with enough training data.

These laws suggest that beyond a certain model size, the benefits of adding more parameters diminish unless significantly more training data is used. Furthermore, the goal is to maximize performance per unit of computational resource (e.g., FLOPs) by adjusting this ratio.

3.2.2 How do we determine tokenizer vocabulary size?

The tokenizer vocabulary size will depend on the dataset for training (The Pile, C4, etc), the tokenization algorithm used (BPE, Unigram, etc), and several other constraints. More outlined below.

Tokenizer Implementation and Scaling: Depending on the tokenizer design and dataset, different approaches can be adopted. There are no conclusions on the tokenizer training dataset and tokenizer design yet, but most likely a split and merge approach will be adopted where the training dataset is divided into N chunks for each of the D domains, resulting in T chunks in total. We can deduce the vocabulary size after determining N, D and T.

Tokenizer Hierarchy: Due to the split and merge approach, individual tokenizers trained on each chunk will be hierarchically merged to form a comprehensive tokenizer.

Merging Strategy and Vocabulary Adjustment: The merging of tokenizers is based on a weighted average of the probability distributions of tokens from each tokenizer, where weights are determined by the relative sizes of the data chunks. After merging, the tokenizer has a vocabulary of V tokens (V should be deduceable from the parameters specified earlier). To manage this size and optimize performance, tokens with the smallest probabilities are dropped to reduce the vocabulary to V'. Additionally, we may also add bytes that don't occur in the Pile and an <|endoftext|> token to ensure no out-of-vocabulary tokens.

Vocabulary Size Considerations: A larger vocabulary allows more information to be captured within the context window of the model, enhancing the detail and specificity of tokenization. However, it also increases the overhead due to a larger proportion of model parameters dedicated to token embeddings. The optimal vocabulary size is determined through experiments where the total encoded size (in bytes) of other datasets is minimized based on the vocabulary size V'.

3.3 Scaling Techniques

3.3.1 Scaling the SSM:

Scaling Structured State Space (SSM) models to handle large workloads effectively involves several key strategies that focus on computational efficiency, parallel processing, and optimization of memory usage. More below.

Fast Fourier Transform (FFT):
Utilize FFT-based algorithms for operations that involve polynomials or sequences, as FFT can efficiently compute convolutions used in the state transitions of SSMs.

Low Rank Approximations: Use low-rank approximations for matrices involved in the state space transformations to reduce computation and storage costs.

3.3.2 Scaling the RWKV RNN:

Scaling the RWKV RNN model to handle large workloads involves strategies revolving around computation and attention. More below.

Sparse Computations: RWKV involves weighted attention mechanisms that can be made sparse. Focus on optimizing these attention computations by leveraging sparsity for efficiency.

Attention Optimization: Optimize the attention mechanism in RWKV by reducing the complexity of key-value computations or by using techniques like low-rank approximations to make the computation of attention scores less resource-intensive.

3.3.3 Scaling the Tokenizer:

Scaling a tokenizer to handle large workloads, especially when used in conjunction with Transformer models, involves optimizing the tokenizer's efficiency, capacity, and throughput to manage extensive and complex datasets efficiently. More below.

Efficient Tokenization Logic: Choose tokenization algorithms that are inherently efficient and quick to execute, such as Unigram, Byte-Pair Encoding (BPE), or WordPiece. Optimize these algorithms by refining their implementation to reduce computational overhead.

3.3.4 Scaling the Transformer:

Scaling a Transformer model to handle large workloads effectively involves a combination of architectural decisions, efficient computation practices, and infrastructure optimization. More below.

Model Pruning: Reduce the size of the Transformer by pruning less important weights or attention heads. This can significantly reduce computational requirements without a substantial loss in performance.

Model Quantization: Apply quantization techniques to reduce the precision of the weights from floating points to integers, which decreases model size and speeds up computation.

Knowledge Distillation: Train a smaller, more efficient "student" model that learns to mimic a larger "teacher" model. This approach can retain much of the performance of the larger model but with fewer resources.

Sparse Attention Mechanisms: Implement variants of the attention mechanism that reduce complexity, such as Longformer’s windowed or global attention, or BigBird’s block-sparse attention, which can handle longer sequences more efficiently.

Low-Rank Factorization: Use matrix factorization techniques to approximate attention layers, reducing the number of computations needed.

3.3.5 Common Machine Learning Scaling Techniques:

Below we will describe some common scaling techniques that can be used for 1 or more of the model architectures.

Parameter Sharing: Implement parameter sharing across different parts of the model to reduce the number of unique parameters that need to be updated and stored during training.

Batch processing and normalization: Implement mini-batch training to process multiple instances of data simultaneously, which can optimize the use of computational resources and reduce training time. Also incorporate batch normalization techniques to maintain numerical stability and improve the convergence rate during training.

Memory management: Use checkpointing strategies during training to save and restore model states periodically, thus managing memory usage more effectively and allowing for recovery in case of interruptions.

Memory optimization: Implement techniques such as gradient checkpointing and activation recomputation during backpropagation to reduce memory overhead at the cost of additional computations.

Data Access: Implement an efficient data loading and preprocessing pipeline to ensure that data feeding does not become a bottleneck. Cache frequently accessed data in memory to speed up data retrieval operations during model training.

Asynchronous Updates: Implement asynchronous gradient updates where possible, which can speed up training by allowing different parts of the model to be updated independently without waiting for a global synchronization, useful in distributed settings.

Hardware: Leverage GPU/TPU clusters which are optimized for parallel processing of large-scale matrix operations typical in any of the models above. In synergy, utilize distributed computing strategies to train the model across multiple machines, effectively partitioning the workload. **(NEED TO DISCUSS WITH LEADS ON HARDWARE)**

4 Training

To be determined.

5 Evaluations

To be determined.

6 Future Work

Future work will focus on expanding the model's capabilities to additional financial sectors, if possible, and further refining its ability to interact with and enhance in-house models.

7 Conclusion

WatGPT epitomizes a concerted effort to harness the transformative potential of large language models in quantitative finance. By delineating clear project goals, aligning model architectures with specific tasks, and fostering a nuanced understanding of model efficiency, this endeavor lays the groundwork for a paradigm shift in algorithmic trading, risk management, portfolio optimization, and several other finance tasks.

References

[1] BloombergGPT: A Large Language Model for Finance. (2024). Retrieved from Bloomberg

[2] Gu, A., Goel, K., & Ré, C. (2022). Efficiently Modeling Long Sequences with Structured State Spaces. Retrieved from arXiv:2111.00396

[3] Peng, B., Alcaide, E., Anthony, Q., et al. (2023). RWKV: Reinventing RNNs for the Transformer Era. Findings of the Association for Computational Linguistics: EMNLP 2023, pages 14048–14077. Retrieved from RWKV-LM GitHub

[4] O. Press, N. A. Smith, and M. Lewis, "Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2022. Available: https://arxiv.org/abs/2108.12409

[5] A. Gu, T. Dao, S. Ermon, A. Rudra, and C. Ré, "HiPPO: Recurrent Memory with Optimal Polynomial Projections," *arXiv preprint arXiv:2008.07669*, October 2020. Available: https://arxiv.org/abs/2008.07669

[6] L. Xue, A. Barua, N. Constant, R. Al-Rfou, S. Narang, M. Kale, A. Roberts, and C. Raffel, "ByT5: Towards a Token-Free Future with Pre-trained Byte-to-Byte Models," 2022. Retrieved from https://github.com/google-research/byt5

[7] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, J. Dean, "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer," *arXiv preprint arXiv:1701.06538*, 2017. Available: https://arxiv.org/abs/1701.06538

[8] A. Gu* and T. Dao*, "Mamba: Linear-Time Sequence Modeling with Selective State Spaces," Machine Learning Department, Carnegie Mellon University, and Department of Computer Science, Princeton University, 2023. Email: agu@cs.cmu.edu, tri@tridao.me. Available online: https://github.com/state-spaces/mamba

[9] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. de Las Casas, L. A. Hendricks, J. Welbl, A. Clark, T. Hennigan, E. Noland, K. Millican, G. van den Driessche, B. Damoc, A. Guy, S. Osindero, K. Simonyan, E. Elsen, J. W. Rae, O. Vinyals, and L. Sifre, "Training Compute-Optimal Large Language Models," arXiv preprint arXiv:2203.15556, Mar. 2022.