DSA4212 Assignment Report

Andrew Lyem
e0985912@u.nus.edu

Min Aung Oo
e0979790@u.nus.edu

Su Jia Ying, Joanne
e0968755@u.nus.edu

1. Introduction

1.1. Overview

Large language models (LLMs) have revolutionized natural language processing through their ability to generate coherent text, answer questions, and perform various language tasks. At the heart of these models lies the transformer architecture, introduced by Vaswani et al. (2017), which has become the foundation for state-of-the-art models like GPT, BERT, and their successors. This project focuses on implementing a transformer-based model from scratch to understand the fundamental mechanisms that power modern language models.

1.2. Problem Statement

The core task of this project is character-level language modeling using the text8 dataset. Character-level modeling operates at a finer granularity than word-level or subword-level approaches, predicting the next character in a sequence rather than the next word or token. This approach offers several advantages for learning: it eliminates the need for complex tokenization schemes, provides a smaller vocabulary size (27 characters: 26 lowercase letters plus space), and allows the model to learn morphology and word formation patterns directly from data.

Formally, given a text corpus $\mathcal{D} = \{x_1, x_2, ..., x_T\}$ where each x_t represents a character, we aim to learn a parametric model f_θ that captures the conditional probability distribution:

$$P_\theta(x_{t+1} \mid x_{t-L+1}, ..., x_t)$$

, where L is the sequence length (the number of previous characters the model observes to predict the next character). The model parameters θ are optimized to maximize the likelihood of the observed sequences in the training data, typically the cross-entropy loss function.

1.3. The text8 Dataset

The text8 dataset is a widely-used benchmark in language modeling research. It consists of the first 100 million characters extracted from a cleaned English Wikipedia dump (March 2006). The dataset has been preprocessed to include only lowercase letters (a-z) and spaces, with all punctuation, numbers, and special characters removed. This simplified character set makes it computationally tractable for experimentation while still preserving the essential linguistic structure of English text.

The dataset's characteristics makes it ideal for this project:

- **Fixed vocabulary**: Only 27 unique characters, simplifying the output layer.
- **Sufficient size**: 100 million characters provide enough data to train meaningful models.
- **Clean format**: Preprocessing eliminates noise and focuses learning on linguistic patterns.
- **Established benchmark**: Allows comparison with existing literature and baselines.

1.4. Objectives

This project has several interconnected goals that together provide comprehensive experience in training neural language models:

- **Architecture Implementation**: Build a transformer-based model capable of processing sequential character data and generating probability distributions over the next character. This involves implementing key components such as multi-head self-attention, position encodings, feed-forward networks, and layer normalization.
- **Model Training**: Develop a complete training pipeline including data preprocessing, batching, loss computation, and optimization. This requires careful consideration of hyperparameters such as learning rate, batch size, and regularization techniques.
- **Performance Evaluation**: Assess model quality using held-out test data, measuring accuracy as the percentage of correct next-character predictions. This metric provides an intuitive measure of how well the model has learned the underlying patterns in the data.
- **Hyperparameter Optimization**: Systematically explore the hyperparameter space, particularly the sequence length L, to identify configurations that yield optimal performance. This involves understanding the trade-off between model capacity, computational cost, and generalization ability.

Through this project, we gain practical insights into the challenges of training neural language models, including optimization dynamics, overfitting prevention, and the architectural choices that influence model performance. The experience gained here

translates directly to understanding and working with larger-scale language models in production settings.

2. Transformer Architecture

2.1. Background and Motivation

The transformer architecture was introduced by Vaswani et al. in their 2017 paper "Attention is All You Need," [1] which proposed a novel network architecture based entirely on attention mechanisms, eliminating the need for recurrence and convolutions. This foundational work has become one of the most cited papers of the 21st century, with over 173,000 citations as of 2025, and has become the cornerstone of modern large language models [2].

2.2. Transformer Architecture

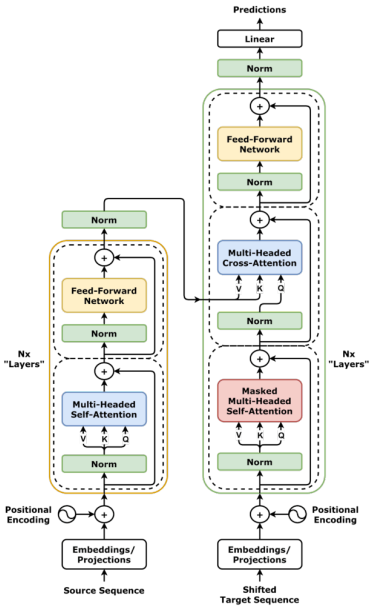


Figure 1: Transformer Architecture

While the original transformer as shown in Figure 1 consisted of both encoder and decoder components designed for sequence-to-sequence tasks like machine translation, modern language models predominantly use a decoder-only architecture. Despite significant innovation, decoder-only transformers remain the cornerstone of generative large language models, with most modern LLMs using architectures that largely match the original GPT model [3].

The decoder-only architecture, as implemented in GPT models, is trained to predict the next token in a sequence based on preceding tokens through a process known as language modeling [4]. Unlike the original transformer decoder, GPT-style decoders remove the cross-attention component that was used to incorporate encoder information, since there is no separate encoder in these models [4].

For our character-level language modeling task, we adopt this decoder-only architecture, which is well-suited for autoregressive generation where each character is predicted based solely on the preceding context.

2.3. Core Components

- **Token Embeddings**
  The first step in processing input is converting discrete characters into continuous vector representations. Each character in our vocabulary (26 lowercase letters plus space) is mapped to a learned embedding vector of dimension d_{model}. These embeddings are trainable parameters that the model learns during training to capture semantic relationships between characters.

  Mathematically, for an input sequence of character indices $[c_1, c_2, ..., c_L]$, the embedding layer produces:

  $$\mathcal{E} = \left[\mathbf{e}_{c_1}, \mathbf{e}_{c_2}, ..., \mathbf{e}_{c_L}\right] \in \mathbb{R}^{L \times d_{\text{model}}}$$

  , where each $\mathbf{e}_{c_i}$ is the embedding vector for character c_i.

- **Positional Encoding**
  Unlike recurrent networks that process sequences in order, transformers treat input sequences as sets and lack inherent understanding of token positions, requiring an explicit method to encode positional information [5]. The original transformer paper introduced positional encoding to provide models with information about the order of elements in a sequence [1].

  There are several approaches to positional encoding [6], however in this project two positional encoding will be used, in particular learned positional encoding and rotary positional encoding (RoPE).

  1. **Learned Positional Encoding**
     In the original transformer paper, both learnable vectors and sinusoidal functions were introduced as positional encoding methods and performed nearly identically [7]. Many modern pretrained language models utilize learnable positional embeddings for their flexibility and task-specific adaptability [3].

     The positional encodings are added element-wise to the token embeddings:

     $$\mathbf{X} = \mathbf{X} + \mathbf{PE}$$

     where $\mathbf{X} \in \mathbb{R}^{L \times d_{\text{model}}}$ represents the combined input to the transformer blocks.

  2. **Rotary Positional Embedding (RoPE)**
     Rotary Position Embedding (RoPE) is a novel

positional encoding method that encodes absolute position with a rotation matrix while simultaneously incorporating explicit relative position dependency in the self-attention formulation [8]. Developed by Jianlin Su and introduced in the RoFormer paper, RoPE has garnered widespread adoption in modern large language models due to its elegant design that unifies absolute and relative positional encoding approaches [9].

RoPE organizes the d features as $d/2$ pairs, treating each pair as coordinates in a 2D plane that are rotated by an angle depending on the token's position [10]. For a token at position m, the rotation is applied as:

$$\text{RoPE}\big(x_m^{(i)}, x_m^{(i+d/2)}, m\big) = \begin{pmatrix} \cos(m\theta_i) & -\sin(m\theta_i) \\ \sin(m\theta_i) & \cos(m\theta_i) \end{pmatrix} \begin{pmatrix} x_m^{(i)} \\ x_m^{(i+d/2)} \end{pmatrix}$$

where $\theta_i = 10000^{-2i/d}$. A key property is that the dot-product attention between positions m and n becomes a function of only their relative distance $m - n$, naturally encoding relative positional information [10].

RoPE is parameter-free, inherently relative, and scales gracefully from short sequences to book-length contexts [11]. Unlike methods like T5′s relative positional bias, RoPE works with efficient transformer variants including kernelized attention mechanisms, since it does not require constructing the full $N \times N$ attention matrix [9]. In billion-parameter models, RoPE demonstrates approximately 30% faster convergence compared to learned absolute positional embeddings and 10-20% improvement over T5′s relative position encoding [9].

For character-level modeling on text8, RoPE offers memory efficiency (no additional parameters), natural decay of attention with distance for modeling both local and long-range patterns, and the ability to handle sequences longer than those seen during training.

- **Multi-Head Self-Attention**
  Self-attention is the core mechanism that allows the model to weigh the importance of different positions in the sequence when processing each token. Self-attention transforms the representation of each token based on its relationship to other tokens in the sequence [3].

  For a given input $\mathbf{X}$, we compute three matrices through learned linear transformations:
  - Query: $\mathbf{Q} = \mathbf{X}\mathbf{W}^Q$
  - Key: $\mathbf{K} = \mathbf{X}\mathbf{W}^K$
  - Value: $\mathbf{V} = \mathbf{X}\mathbf{W}^V$

  , then the attention mechanism is computed as follows

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

  , where d_k is the dimension of the key vectors, and the scaling factor $\sqrt{d_k}$ prevents the dot product from growing too large.

  In decoder-only models, masked self-attention is critical as it ensures that each position can only attend to earlier positions by masking future positions and setting them to negative infinity before the softmax operation [12]. This left-to-right processing is essential for autoregressive text generation, where each token is predicted based only on preceding token [3].

  Rather than performing a single attention operation, multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions [1]. The model uses h parallel attention heads, each with its own learned projection matrices:

  $$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = (\text{head}_1 \oplus ... \oplus \text{head}_h)\mathbf{W}^O$$

  where $\oplus$ is the concatenation operation and $\text{head}_i = \text{Attention}\big(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V\big)$.

  The original transformer paper found that using 8 attention heads provided good performance, though they noted that quality degrades with too many heads [1].

- **Feed-Forward Neural Network**
  After the attention mechanism, each position passes through a position-wise feed-forward network. This consists of two linear transformations with a ReLU or other activation function in between, applied identically to each position:

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

  While the linear transformations are the same across different positions, they use different parameters from layer to layer NeurIPS. The intermediate dimension (also called filter size or feedforward size) is typically 4 times the model dimension.

- **Layer Normalization & Residual Connections**
  Layer normalization and residual connections, while conceptually not essential to the transformer's operation, are necessary for numer-

ical stability and successful training. The residual connection can be expressed as:

$$\mathbf{y} = F(\mathbf{x}) + \mathbf{x}$$

where $F(\mathbf{x})$ represents the sub-layer function (either attention or feed-forward). These residual connections help avoid vanishing gradient problems and stabilize the training process.

Layer normalization is typically applied either before (pre-norm) or after (post-norm) each sublayer, normalizing the activations across the feature dimension.

- **Decoder Block Structure**
  A decoder-only transformer block consists of two sublayers: masked multi-head self-attention and a position-wise feed-forward network Wikipedia. The complete block can be described as:

  Multiple decoder blocks are stacked sequentially. GPT-1 used 12 decoder blocks, while larger models scale up to hundreds of layers [3].

- **Output Layer**
  The final decoder block's output is passed through a linear projection layer followed by a softmax function to produce a probability distribution over the vocabulary:

  $$P[c_{t+1}|c_1, ..., c_t] = \text{softmax}(\mathbf{W}_{\text{out}}\mathbf{h}_t + \mathbf{b}_{\text{out}})$$

  where $\mathbf{h}_t$ is the hidden state at position t from the last decoder block. Some models share the weight matrix between the embedding layers and the pre-softmax linear transformation, multiplying the embedding weights by $\sqrt{d_{\text{model}}}$ in the embedding layers.

3. Experiments & Tuning

3.1. Implementation Details

All models were implemented using JAX and Flax, leveraging their efficient automatic differentiation and compilation capabilities. Training was performed on Google TPU T4 and NVIDIA RTX 3070 Ti.

For our character-level language modelling task on text8, the implemented base decoder-only transformer was with the following hyperparameters and specification:

- **Vocabulary Size**: 27 (26 lowercase letters + space character).
- **Model Dimension** (d_{model}): 64.
- **Number of Decoder Blocks**: 6 decoder blocks.
- **Number of Attention Layers**: 8 heads.
- **Max Length**: 128 characters.
- **MLP Ratio**: 4

Next, other considered hyperparameters that affect the training process are

- **Learning Rate**: With an initial value of 0.001.
- **Batch Size**: 128
- **Sequence Length**: 32

The loss function used is the cross-entropy loss to maximize the likelihood of the correct next character and the optimizer used was Adam optimizer.

The result from this base model and first configuration is 64.3% accuracy and minimum test loss of 1.29. The loss curve is shown as follows:

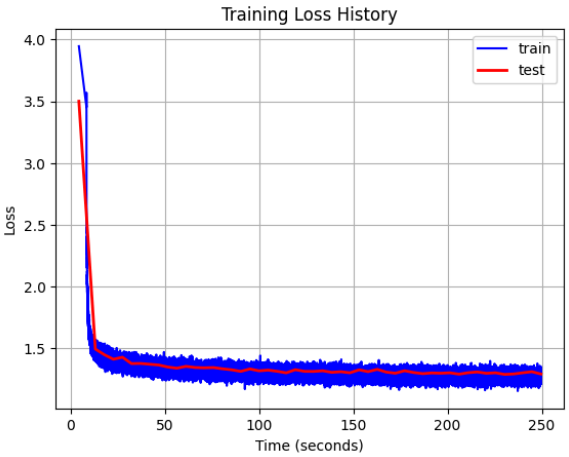


Figure 2: Base Decoder-Only-Transformer Loss Curve

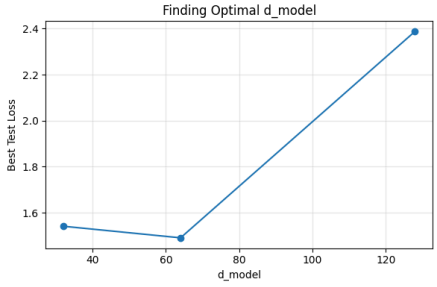
3.2. Hyperparameter Tuning

Now from the base model, the existing hyperparameters are further fine-tuned and new hyperparameters are also introduced.

The tuning process is done by testing several candidates by their test loss, then a small subset of the training data was used to tune the hyperparameters, in particular the first $500,000$ characters of the text8 training dataset.

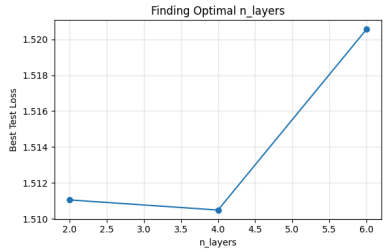
The fine-tuned hyperparameters are

- **Model Dimension**
  The candidates that were chosen are $32, 64, 128$, then the result is as follows:



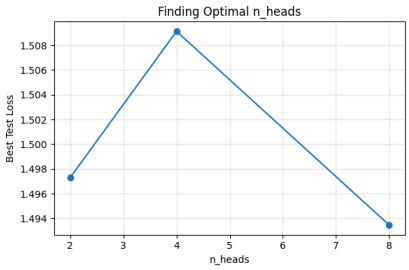
From Figure 3, the best observed hidden model dimension is 64.

- **Number of Decoder Blocks**
  The chosen candidates were 2, 4, and 6. The result is shown as follows



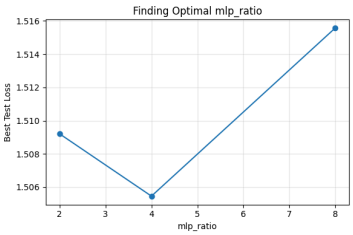
  From Figure 4, the best observed number of layers is 4.

- **Number of Attention Layers**
  The chosen values were 2, 4, and 8 and the results are shown as follows



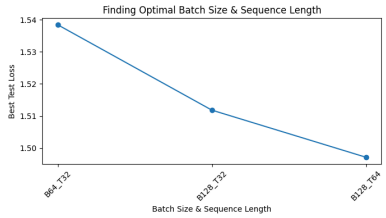
  From Figure 5, the best observed number of attention heads were 8.

- **MLP Ratio**
  The chosen values were 2, 4, and 8 and the results are shown as follows



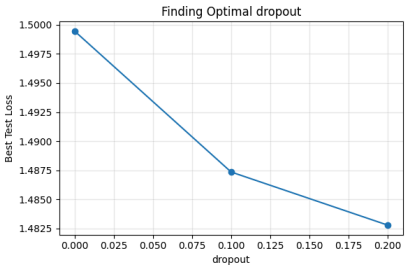
  From Figure 6, the best MLP ratio is 4.

- **Batch Size & Sequence Length**
  The candidates that were chosen are $(B, T) = ((64, 32), (128, 32), (128, 64))$, then the result is as follows:



  From Figure 7, the best observed batch size and sequence length are 128 and 32 respectively.

- **Dropout**
  The chosen values were 0, 0.1, and 0.2 and the results are shown as follows



  From Figure 8, the best dropout is 0.2.

3.3. Experimentation

3.3.1. Transformer-I

Based on the chosen best hyperparameters, trained with the Adam optimizer and learning rate 0.01 with the full dataset in $80,000$ iterations, the model reached 59.62% accuracy.

3.3.2. Transformer-II

Afterwards, a different configuration was tested with the following specifications

- **Hidden Model Dimensions**: 256
- **Number of Decoder Blocks**: 6 blocks
- **Number of Attention Layers**: 8 heads
- **Model Dimension**: 256
- **MLP Ratio**: 8

and trained with the AdamW optimizer with an exponential decay scheduler with the following specifications:

Initial Learning Rate	6×10^{-4}
Transition Steps	$1,000$
Decay Rate	0.96
End Value	10^{-5}

Table 1: Exponential Decay Scheduler Parameters

where the model reached an accuracy of 68.31%.

3.3.3. Transformer-III

Another experiment was run with the same configuration as the previous experiment with a linear learning rate warmup cosine decay scheduler with the following specifications:

Initial Learning Rate	0
Peak Learning Rate	6×10^{-4}
Warmup Steps	$2,000$
Decay Steps	$100,000$

Table 2: Warmup Cosine Decay Scheduler Parameters

Hyperparameter	Learning Rate	Batch Size	Sequence Length	Model Size	Val. Loss	Acc. (All)	Acc. (Next Char.)
Learning Rate	$\mathbf{3 \times 10^{-4}}$	128	64	Small	1.4215	56.79%	58.06%
Batch Size	3×10^{-4}	**128**	64	Small	1.4215	56.81%	58.08%
Sequence Length	3×10^{-4}	128	**128**	Small	1.4112	57.87%	58.57%
Model Size	3×10^{-4}	128	128	**Small**	1.4112	57.87%	58.57%

Table 3: LSTM Configurations & Accuracies

Hyperparameter	Learning Rate	Batch Size	Sequence Length	Acc. (All)	Acc. (Next Char.)
Transformer-I	0.01	128	32	57.89%	59.62%
Transformer-II	[Refer to Table 1]	256	64	65.35%	68.31%
Transformer-III	[Refer to Table 2]	64	256	65.96%	69.08%
RNN-LSTM	3×10^{-4}	128	128	57.87%	58.57%

Table 4: Model Summaries

the model also uses rotary positional encoding (RoPE) instead of learned positional encoding. The result for this configuration was 69.08% accuracy.

3.3.4. RNN-LSTM

LSTMs are a type of recurrent neural network that process sequences one step at a time, using a hidden state and gating mechanisms to remember or forget information. This makes them good at modelling short- to medium-range dependencies but limits parallelism and long-range modeling. Transformers remove recurrence entirely and use self-attention, allowing every position to directly access all others in parallel. As a result, transformers capture long-range patterns more effectively and train much faster, especially on large datasets.

Finally, an RNN-LSTM model was tested using PyTorch with epochs, where one epoch corresponds to a complete pass through the dataset using a PyTorch dataloader. This is simply the standard training setup in PyTorch, and it provides a convenient way to monitor training and validation performance after each full sweep of the data.

The specifications and results shown on Table 3, where each row represents one experiment for one hyperparameter testing at different values. The hyperparameters to be tuned are in the order of the rows in the table below. In each experiment, the best value is found and then used for the next test. All tests were performed with a smaller subset of the full data. For better generalization, the smallest validation loss is used as the point of reference.

The final row of Table 3 is the optimal hyperparameter configuration for the LSTM model, where the models have an embedding dimension of 128 and hidden dimension of 256.

4. Discussion

From the trained models, the configurations and hyperparameter are summarized in Table 4.

Notably, the hyperparameter values that was obtained from the hyperparameter tuning were sub-optimal when trained on the full dataset. This may be attributed to the subset of the training dataset not being able to represent the whole dataset. Another possible cause is due to sequential dependence, as the hyperparameter tuning process only uses the first 500,000 characters, then the earlier portions of the text might have different characteristics than the later portions. The difference of the character frequency distribution between the subset and the full dataset may also play a role in the lower accuracy of the hyperparameter tuned model.

Based on Table 4, the optimal model is **Transformer-III**:

- **Sequence Length**: 256 characters.
- **Architecture**: 6 decoder layers, $d_{\text{model}} = 256$, & 8 attention heads.
- **Positional Encoding**: RoPE
- **Learning Rate**: Warmup Cosine Decay Scheuler (Refer to Table 2)
- **Dropout**: 0.2
- **Batch Size**: 64

with achieved accuracy of 69.08%.

Since only **Transformer-III** used RoPE, therefore positional encoding represents a significant reduction for next-character predictions. This might be caused by several factors, such as:

- **Explicit Relative Position Encoding**
  RoPE encodes absolute position with a rotation matrix while simultaneously incorporating explicit relative position dependency in the self-attention formulation [8]. Unlike sinusoidal encodings where relative position information must be learned implicitly, RoPE mathematically guarantees that attention scores depend only on relative distances: $\text{Attention}(m, n) \propto f(|m - n|)$. For character-level modeling, this is particularly valuable because the most predictive information typically comes from nearby characters (e.g., completing common words like "the", "and", "tion").
- **Natural Distance Decay Property**
  RoPE exhibits naturally decaying inter-token dependency with increasing relative distances [8], which aligns well with linguistic intuition for character-level modeling. The rotation angle $\theta_i = 10000^{-2i/d}$ creates a hierarchy where the similarity between positions decreases as $\text{sim}(m, n) = \cos((m - n)\theta_i)$, meaning immediate neighbors (distance 1-3) receive high attention for capturing character sequences within words, word-level context (distance 4-10) receives moderate attention for word boundaries, and distant context (distance 10+) receives lower attention to reduce noise.

This project successfully implemented and trained a decoder-only transformer architecture for character-level language modeling on the text8 dataset, achieving a final test accuracy of 69.08% on next-character prediction. Through systematic experimentation, several critical design choices were identified that significantly impact the model's performance: sequence length of 256 characters provided optimal balance between capturing local patterns and long-range dependencies, while Rotary Position Embedding (RoPE) demonstrated clear advantages over traditional learned encodings through its explicit relative position information and natural distance decay properties. The experimental process revealed important methodological insights, such as hyperparameters optimized on training subsets did not necessarily transfer optimally to the full dataset, highlighting the importance of validating configurations on representative data samples and understanding potential distributional differences between subset and complete corpus.

Comparative analysis demonstrated the transformer's superior performance over baseline models, including RNN-LSTM architectures, validating the effectiveness of self-attention mechanisms for character-level prediction tasks. The ablation studies confirmed that architectural components such as RoPE positional encoding, appropriate context length, and careful regularization each contribute meaningfully to the final performance. These findings demonstrate that while transformers were originally designed for word-level and subword-level modeling, their architectural principles translate effectively to fine-grained character-level sequential prediction.

5. References

[1] A. Vaswani *et al.*, "Attention Is All You Need." [Online]. Available: https://arxiv.org/abs/1706.03762

[2] P. Olson, "Ex-Google Scientists Kickstarted the Generative AI Era of ChatGPT, Midjourney." Accessed: Nov. 16, 2025. [Online]. Available: https://www.bloomberg.com/opinion/features/2023-07-13/ex-google-scientists-kickstarted-the-generative-ai-era-of-chatgpt-midjourney?embedded-checkout=true

[3] C. R. W. Ph.D, "Decoder-Only Transformers: The Workhorse of Generative LLMs." [Online]. Available: https://cameronrwolfe.substack.com/p/decoder-only-transformers-the-workhorse

[4] M. Ardi, "Meet GPT, The Decoder-Only Transformer | Towards Data Science." [Online]. Available: https://towardsdatascience.com/meet-gpt-the-decoder-only-transformer-12f4a7918b36/

[5] M. Saeed, "A Gentle Introduction to Positional Encoding In Transformer Models, Part 1." [Online]. Available: https://machinelearningmastery.com/a-gentle-introduction-to-positional-encoding-in-transformer-models-part-1/

[6] A. Tam, "Positional Encodings in Transformer Models - MachineLearningMastery.com." [Online]. Available: https://machinelearningmastery.com/positional-encodings-in-transformer-models/

[7] G. Wang, Y. Lu, L. Cui, T. Lv, D. Florencio, and C. Zhang, "A Simple yet Effective Learnable Positional Encoding Method for Improving Document Transformer Model." [Online]. Available: https://aclanthology.org/2022.findings-aacl.42.pdf

[8] J. Su, Y. Lu, S.-F. Pan, B. Wen, and Y. Liu, "RoFormer: Enhanced Transformer with Rotary Position Embedding." 2021. doi: 10.48550/arxiv.2104.09864.

[9] S. Biderman *et al.*, "Rotary Embeddings: A Relative Revolution." [Online]. Available: https://blog.eleuther.ai/rotary-embeddings/

[10] "Rotary Positional Embeddings (RoPE)." Accessed: Nov. 20, 2025. [Online]. Available: https://nn.labml.ai/transformers/rope/index.html

[11] Shubham, "Inside RoPE: Rotary Magic into Position Embeddings." Accessed: Nov. 20, 2025. [Online]. Available: https://learnopencv.com/rope-position-embeddings/

[12] prashun javeri, "GPT Architecture." Accessed: Nov. 20, 2025. [Online]. Available: https://medium.com/@prashunjaveri/gpt-architecture-0415e7a5796d