Transformer-RNN Hybrid Model Using Sequential Reformulation of Attention Mechanism for Natural Language Processing (NLP)

Justine M. Favia

John Mark A. Bermudez

Notre Dame of Marbel University

Bachelor of Science in Computer Science

Merch Jay P. Dollaga, MIT

December 2024

Approval Sheet

This Thesis entitled

Transformer-RNN Hybrid Model Using Sequential Reformulation of Attention Mechanism for Natural Language Processing (NLP), prepared, and submitted by **Justine M. Favia,** and **John Mark A. Bermudez**, in partial fulfillment of the requirement for the degree **Bachelor of Science in Computer Science** is hereby accepted.

Merch Jay P. Dollaga, MIT
Thesis Adviser

Brenda M. Balala, MIT

Panel Chair

Hajah T. Sueno, DIT
Panel Member

Engr. Victorino R. Tobias, MEP-ECE
Panel Member

Accepted and approved for the conferral of the degree

Bachelor of Science in Computer Science

Engr.Victorino R. Tobias, Jr., MEP-ECE
Dean, College of Engineering and Technology

Table of Contents

List of Figures

Introduction

Background of the Study

The introduction of Transformers represented a major advancement in sequence modeling, offering a highly efficient architecture that takes full advantage of GPU parallelism. However, Transformers are computationally intensive during inference and requires massive amount of dataset which diminishes the effectiveness of the model especially in resource-constrained settings. (Feng et al., 2024) According to Duman K. et al. (2022), There have also been a slew of high-tech progressions since the debut of transformer models. This architecture has been effective and versatile in its use. As impressive as these accomplishments are, there are caveats. State of the art transformer architectures revolves around the self-attention layer and its variants which can be very computationally expensive. This is, the resource cost, both time and space, of this mechanism is proportional to the square of the length of the input sequence. This quadratic situation is a striking challenge on the one hand when one has to deal with long input or need computation in a hurry, on the other.

The computational complexity of the self-attention mechanism increases exponentially by a factor of 2 with increasing number of tokens, leading to a significant rise in computational load as image resolution increases. To mitigate this, some researchers explore techniques such as dynamic token selection or token pruning to reduce token redundancy and alleviate the computational demands of attention calculations. (Zhang et al., 2024) According to Walton et al. (2022) this challenge has recently garnered a substantial amount of recognition with many researchers seeking to simplify the attention mechanism. Various approaches have been proposed such as sparse attention, linearized attention and kernel-based attention. Nevertheless, a majority of these approaches involve some kind of compromise either in terms of efficiency or the model's performance. Transformers' ascent to prominence as the industry standard for language processing and their subsequent advances in computer vision have resulted in a commensurate increase in parameter dimensions and quantity of training data. Many now think that as a result, transformers are inappropriate for few data sets. This pattern gives rise to worries like: data accessibility issues in some scientific fields and the rejection of research applications by those with low resources in the working environment. Our goal in this work is to provide a method for transformers to enable small-scale learning.

The concept of Small Language models has emerged as a promising direction in this research landscape. It involves redesigning the transformer architecture to achieve similar or better performance with reduced computational requirements. This approach not only addresses the complexity issue but also potentially leads to more deployable models for resource-constrained environments and without the demand of big data requirement. In this study we propose sequential reformulations of the attention mechanism to modify the standard attention mechanism architecture into a sequential computation with linear complexity. According to Dolga et al. (2024), "The time complexity of the standard attention mechanism in a transformer scales quadratically in direct proportion to the length of the input sequence" , this means that the amount of parameters scale exponentially under the amount of floating point operations per token. We will present a modification to reduce this to linear time scaling. To reduced memory and energy consumption, making models more suitable for edge devices and mobile applications. This efficiency opens up new possibilities for deploying NLP models in resource-constrained environments.

Objectives of the Study

The main objective of this study is to engineer a small language model that achieves linear inference complexity while maintaining high performance across various NLP tasks.

Specifically, this study aims to attain the following:

1. Develop a hybrid transformer model by integrating RNN architecture with a linear reformulated attention mechanism:

a. Reconstructing the attention mechanism dot product computation into a linear element-directed attention.

b. Design a model training architecture that enables the RNN hidden layers to update weights concurrently and parallelized during training.

2. Training and performance assessment of the hybrid RNN-Transformer model from the original Transformer model using quantitative evaluation of the following:

a. Cosine Similarity

b. Perplexity

c. ROUGE (Recall-Oriented Understudy for Gisting Evaluation) Score

3. Model deployment

a. Deploy the hybrid RNN-transformer model as an executable program to a PC.

Review of Related Literature

This section reviews related literature and systems that will help researchers gather basic information and references for the current study.

Transformer Architecture

The Transformer model has achieved great success since its introduction and has become the dominant architecture in NLP. It was originally proposed by (Vaswani, 2017) as a sequence-to-sequence model for machine translation. Unlike recurrent neural networks, the transformer relies entirely on attention mechanisms rather than recurrence. According to Karita et al. (2019), transformer are trained to sequential information through a self-attention layer instead of linear recurrent connection which can be found in RNNs. The primary components of the Transformer include multi-head self-attention, which captures dependencies across different positions in a sequence, and feed-forward networks for processing the information. Self-attention, or intra-attention, calculates the output at a given position in the sequence by considering all other positions within the same sequence. This mechanism facilitates the flow of information across the entire sequence.

Attention Mechanism

Attention is the mechanism which provides computation for the Transformer to communicate different positions from the input and output sequences to compute contextualized representations. It passes the array of query and a set of key-value pairs to the target indexes of the output results in providing higher scores to the most related tokens from the input when predicting each output element. According to Soydaner (2022), the concept of incorporating an algorithm in reference to how humans perceive shapes on the nervous system inspired the invention of neural networks. This concept is called the attention mechanism, and it has been the subject of extensive research and development throughout the last decade. Self-attention mechanism from transformers enables contextualization of a position in a sequence by communicating to all positions within the input. It operates on “queries”, “keys” and “values” packed into matrices. Where Q(query)K(key)^T (sequence length) gives the similarity between each query and all keys, and the scaling is processed to the attention score dot product by dividing it with the square root of the dimensions of the queries and the keys. This is then calculated through a SoftMax function to generate attention weights, which represent a probability distribution of the attention output which results in the computation of the weighted sum of the scores. According to an expert in the field of model architecture, they replicate the dataset retrieval process by issuing a query for a specific word and retrieving the corresponding keys for all words (Wang, n.d.). This means that the standard transformer model is able to capture contextual relationships by attending to all positions in the sequence, unlike RNNs, which rely on sequential updates of the hidden state. The attention mechanism in Transformers enables parallel processing, weighing the importance of all tokens simultaneously and effectively capturing long distant token relationship without the limitations of vanishing gradient problem from RNN sequential propagation.

Training and Inference Complexity

Self-attention enables context modeling regardless of sequence length, it incurs significant computational costs scaling quadratically with the length of the input/output sequences. During training, this complexity can be reduced by parallelizing attention computations across all heads and tokens. According to Aloise & Montréal (2021), the Transformer has been an innovation to sequence modelling operations, but its effectiveness comes with a cost of a quadratic computational and memory complexity with respect to the sequence length, making its adoption challenging. The original Transformer model has a self-attention component with O (n ^2) time and memory complexity" (Beltagy et al., 2020) This means that for a sequence of length n with h attention heads, this requires O(n^2*h) operations to be computed in parallel. However, during inference the attention computations must be performed sequentially for each token. This amounts to O(n^2) time complexity per layer, becoming prohibitively expensive for long sequences on resource constrained devices. Additionally, storing the full self-attention matrix requires O(n^2) space. This makes it challenging to deploy large Transformer models for tasks requiring processing of lengthy inputs, such as long document classification.

Attention Free Transformers

Some researchers have sought to develop Transformer variants without attention to address its quadratic complexity bottleneck. The Performer (Choromanski et al., 2021) replaces attention with a Fourier transform approach, reducing complexity to O (n log n) by leveraging fast Fourier transforms (FFTs). However, FFTs still involve matrix operations with cubic runtime. The Linear Transformer (Katharopoulos et al., 2020), removes attention entirely, using simple linear transformations inspired by convolutional networks. It achieves competitive performance to attention-based models while being 10-100x faster to train. The Universal Transformer (Dehghani et al., 2019) uses dynamic convolution to relate inputs and outputs. It achieves O(n) training time and O (n log n) inference time per layer. However, attention remains crucial to the model's success in practice. It can be said that attention-free models improve efficiency, attention mechanisms have generally proven crucial to the Transformer's strong empirical performance. Reformulating attention in more efficient ways remains a promising research direction.

RNN Architecture

The sequential computations also do not leverage parallel hardware. As a result, RNN inference does not scale efficiently compared to its training. Recurrent Neural Networks are commonly used for modelling data with sequential dependencies prior to Transformers. Their recurrent connections allow modeling of sequential dependencies. According to Indrajitbarat (2023) Recurrent Neural Networks (RNNs) face limitations in processing long sequences due to the vanishing and exploding gradient problems, where the gradients of the loss function either diminish to zero or grow uncontrollably as they propagate through time. This hinders the model's ability of retrieving distant unit dependencies. To address these challenges, several advanced techniques have been suggested, such as the use of GRUs or “Gated Recurrent Units” and LSTMs (Long Short-Term Memory) cells. These architectures are specifically designed to mitigate the issues of gradient instability, and empirical results demonstrate their superior performance in training RNNs on extended sequence lengths. However, standard RNNs have difficulty to obtain long-range dependencies because of the vanishing gradients problem. More sophisticated gated RNN variants like LSTMs and GRUs were developed to address this. According to Al-selwi et al. (2024) “LSTM” (Long Short and Term Memory) is a variation of Recurrent Neural Network that gained recognition because of the performance in processing sequential data with distant contextual relationships. Although its reputation is commendable, challenges related to proper initialization and optimization of LSTM models are more challenging than traditional RNNs due to their more complex architecture, which includes a larger number of parameters. LSTMs introduce additional gating mechanisms which includes “input gating”, “forget gating”, and “output gating”. Compared to the single hidden state update mechanism in standard RNNs, each gate has its own set of weights, which significantly increasing the total parameter count. This added complexity requires more computation and memory for optimization, and the growing hidden state, along with the increasing number of tokens, further complicates training by amplifying the difficulty in maintaining and updating long-term dependencies. As a result, LSTM models can be harder to optimize, limiting their performance and accuracy, especially on longer sequences. Transformers replaced recurrence with attention to enable modeling of relationships between all positional elements. They overcome the limitations of RNNs by leveraging positional encodings and attention across the full context. While RNNs update their hidden state sequentially, transformers' self-attention allows relating each position to every other position in constant time irrespective of sequence length. (Karita et al., 2019) Overall, attention mechanisms have proven more effective than recurrence for capturing dependencies in sequential data like text. However, RNNs remain relevant in processes demanding the modeling of time series and temporal data like speech and time-series.

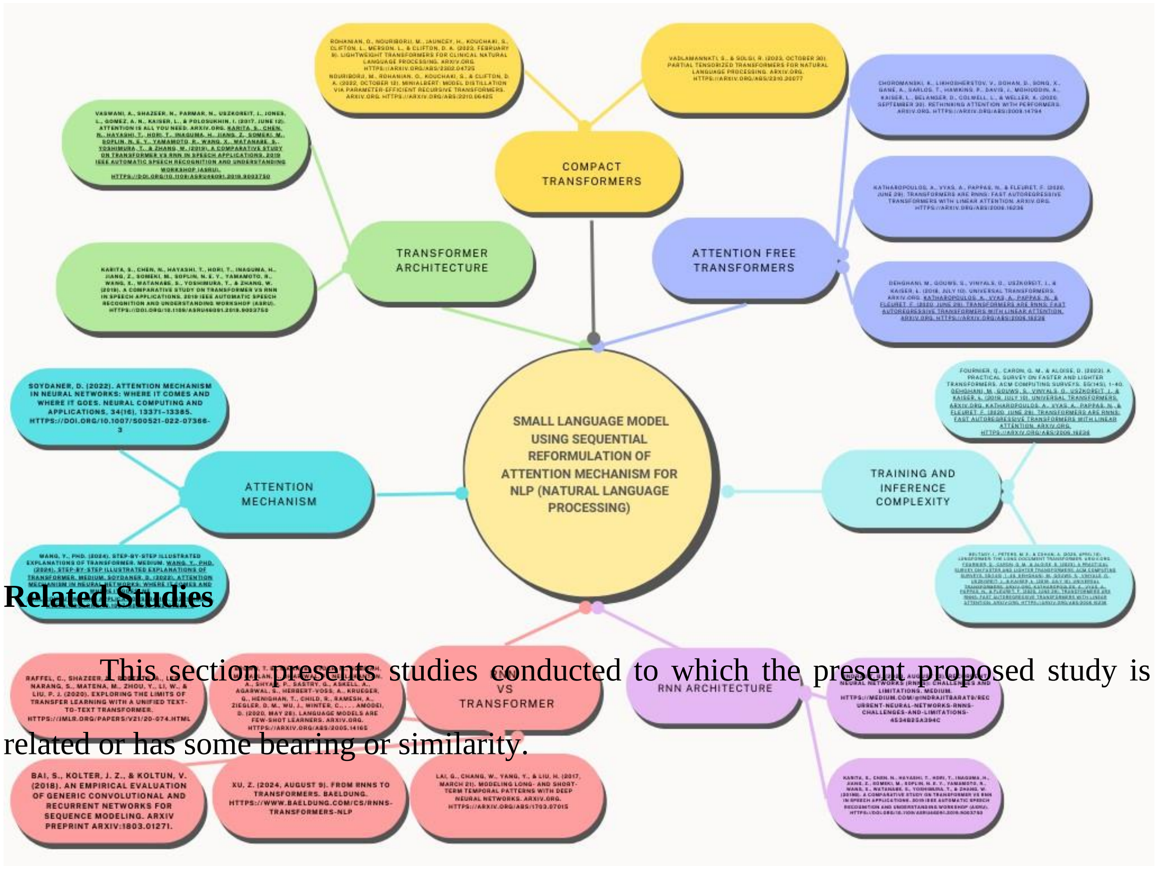
Small Language models

Small language models have acquired huge recognition in Natural Language Processing (NLP) due to their potential for achieving performance comparable to larger models while being more resource-efficient. According to the study of Lu et al. (2024) small language models have a parameter count ranging from 100M(million) to 5B(billion) parameters. Techniques like knowledge distillation and parameter-efficient architectures have been essential in creating these compact models. For instance, the development of lightweight clinical transformers demonstrates that smaller models, with parameters ranging from 15 to 65 million, can effectively handle tasks like Named Entity Recognition and Sequence Classification, often matching or surpassing larger models such as BioBERT and ClinicalBioBERT (Rohanian et al., 2020). Other strategies, like cross-layer parameter sharing, have been explored to reduce model size while maintaining performance, as seen in MiniALBERT, which combines distillation and recursive parameter sharing to create a compact model without significant accuracy loss across various NLP tasks (Nouriborji et al., 2021). The continued refinement of these compact transformer architectures offers promising avenues for reducing computational costs without sacrificing performance.

RNN VS Transformers

Transformers and Recurrent Neural Networks (RNNs) differ fundamentally in their architectural approach to processing sequential data in NLP tasks. RNNs process information in a sequential manner, updating the hidden layer that encodes past information, whereas Transformers takes advantage of simultaneous sequence processing by processing the inputs using self-attention mechanisms. This architectural distinction leads to notable performance differences. Transformer models are proficient at capturing distant contextual relationships and are highly parallelizable (can be trained simultaneously across multiple batches of dataset), leading to faster training and inference on large datasets. The self-attention mechanism allows the model to assign varying weights on emphasis to different input tokens when making predictions, giving it an ability to understand long distant dependencies of input tokens without the requirement of sequential processing that of RNNs. Transformers in a conventional implementation constitutes both an encoder and decoder layer integrated with multi-head self-attention mechanisms, cross attention and feed-forward neural network for each forward pass of attention layer. (Xu, 2024) They have shown superior performance in tasks requiring understanding of broader context, such as machine translation and text summarization (Raffel et al., 2020). RNNs, however, can be more memory-efficient for very long sequences and perform well on tasks that require strict sequential processing. (Bai et al., 2018) The performance gap is particularly noticeable in complex language understanding tasks, where Transformer-based models like BERT and GPT have set new performance standards in the field. (Brown et al., 2020) However, RNNs still maintain an edge in certain time-series prediction tasks and in scenarios with limited computational resources. (Lai et al., 2018)

Figure 1 Literature Map


SMALL LANGUAGE MODEL USING SEQUENTIAL REFORMULATION OF ATTENTION MECHANISM FOR NLP (NATURAL LANGUAGE PROCESSING)
COMPACT TRANSFORMERS
TRANSFORMER ARCHITECTURE
ATTENTION FREE TRANSFORMERS
ATTENTION MECHANISM
TRAINING AND INFERENCE COMPLEXITY
RNN VS TRANSFORMER
RNN ARCHITECTURE

Related Studies

This section presents studies conducted to which the present proposed study is related or has some bearing or similarity.

Do You Even Need Attention?

Presented in the study of (Melas-Kyriazi, 2021)titled "Do You Even Need Attention? A Stack of Feed-Forward Layers Does Surprisingly Well on ImageNet" provides valuable insights that inform my thesis on compact transformers for NLP. The authors demonstrate that by replacing the attention layers in Vision Transformers (ViTs) with simple feed-forward layers, they can achieve competitive performance on ImageNet, suggesting that the efficacy of transformer models may stem more from architectural choices, such as patch embeddings, than from attention itself. This finding aligns with my investigation into sequential reformulations of attention mechanisms in NLP, where it aims to optimize efficiency without compromising performance. By examining the fundamental elements that contribute to model success, this research underscores the potential of alternative architectures in natural language processing, encouraging a deeper understanding of how compact designs can be effectively implemented.

An Attention Free Transformer

The research presented by (Zhai et al., 2021) in "An Attention Free Transformer" offers a groundbreaking perspective on transformer architecture by addressing the computational inefficiencies associated with traditional attention mechanisms. By eliminating the need for explicit attention calculations, this model retains direct interaction between sequence elements while achieving linear time and space complexity with respect to both input and model sizes. The study introduces two variants—AFT-local and AFT conv—that leverage learned position biases to enhance parameter efficiency and performance. Empirical results demonstrate that the attention-free approach provides competitive performance across various tasks, including image classification and language modeling, while significantly reducing computational overhead. This aligns with my thesis on compact transformers in NLP, as it reinforces the potential for optimizing transformer architectures by rethinking foundational mechanisms, thereby paving the way for more efficient and scalable models in natural language processing.

Linformer: Self-Attention with Linear Complexity

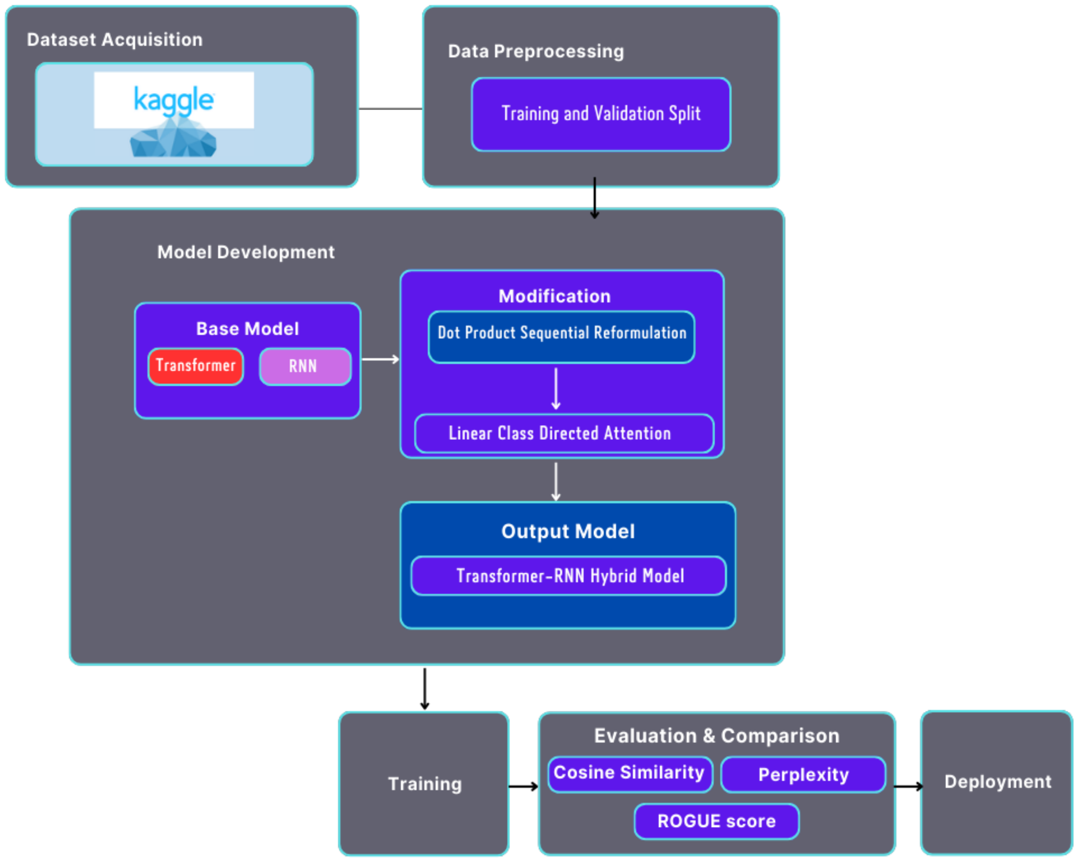
According to the Study of Wang et al. (2020) "Linformer": Self-Attention with Linear Complexity," the authors address the computational bottleneck inherent in traditional Transformer architectures by leveraging the low-rank properties of self-attention. They propose a novel mechanism that reformulates self-attention into an operation with linear complexity, both in space and time, by decomposing the scaled dot-product attention into smaller, more manageable components. This low-rank factorization allows for efficient approximations, maintaining performance while significantly reducing resource requirements. Through empirical evaluations, the Linformer demonstrates competitive results in pretraining and fine-tuning on various NLP tasks, showcasing its potential to enhance the efficiency of Transformer models without sacrificing accuracy. This work serves as a foundational reference for exploring compact Transformer implementations in the context of sequential reformulations of attention mechanisms.

Synthesis

The studies on transformer architecture, including Melas-Kyriazi (2021), (Zhai et al., 2021), and (Wang et al., 2020), collectively highlight the posibility for optimizing efficiency in NLP through innovative design choices. Melas-Kyriazi's research challenges the necessity of attention layers, suggesting that simpler feed-forward architectures can perform competitively, indicating that core structural elements may be more influential than attention itself. Zhai et al. further this notion by introducing attention-free transformers, which maintain direct interactions between sequence elements while achieving linear complexity, thus reducing computational demands. Similarly, Wang et al. tackle the inefficiencies of traditional self-attention by proposing the Linformer, which reformulates attention into a low-rank, linear complexity model. Together, these works encourage a reevaluation of transformer foundations, underscoring the viability of alternative architectures and compact designs in enhancing the performance and scalability of NLP models. This synthesis aligns closely with my thesis objective of developing more efficient transformer implementations through sequential reformulations of attention mechanisms.

Concept of the Study

Figure *2*. Conceptual Framework of the Study



Dataset Acquisition

The data will be a collection of text corpora downloaded from Kaggle, such as the Kaggle's 3k Conversation dataset that will represent various linguistic patterns and structures suitable for natural language processing tasks. The collected dataset is subsequently divided, with 80% allocated for training and 20% for validation, ensuring a balanced approach for model development and assessment.

Data Preprocessing

The preparation of data will involve tasks such as removing noise, handling missing values, converting characters into indices using a character-to-index mapping, and ensuring consistent tokenization. The text is then processed into sequences of fixed length to be used as input for training the model, with the dataset being split into training and validation sets. This preprocessing ensures that the model receives clean and structured input data.

Model Development

The model development consists of three main stages: the base model, modification, and output model. The base model includes two components: Transformer and RNN, which form the foundational architecture for sequence processing. The modification stage, involves the reformulation of standard dot product attention mechanism into a linear element directed attention.

Base Model (Transformer and RNN)

Figure *3*. Base Model References Transformer (left side) and RNN (right side) architecture.

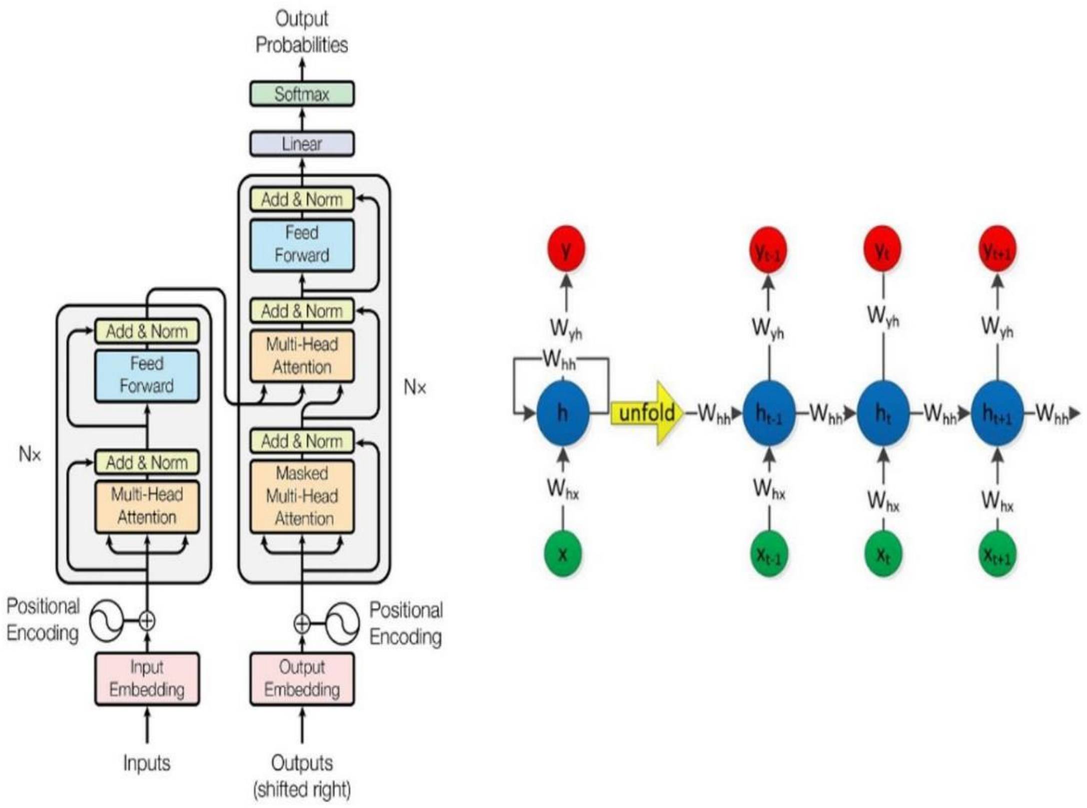


Figure 3 visualizes the architectures of Transformer and Recurrent Neural Network model, emphasizing their key components and structural differences. RNNs process sequential data by updating hidden states at each time step, with each hidden state containing information from previous time steps. This sequential structure allows RNNs to model temporal dependencies effectively. However, their ability to capture long distant contextual relationship is bounded by the vanishing gradient problem, where gradients diminish as they are propagated through many time steps. This hampers the model's learning from distant data points, impeding its ability for tasks that demands understanding long-range text relationships. In contrast, Transformers operates with self-attention mechanisms, which allow it to calculate for the attention scores in parallelized procedure while weighing the importance of each token relative to others. This ability enables Transformers to capture token dependencies despite large relative distances in the embedding space and contextual relationships much more effectively, overcoming the limitations of sequential processing. Despite their strength in handling long-range dependencies, Transformers have a higher computational cost compared to RNNs' algorithmic time complexity and can be very heavy to train because of its "data-hungry" nature that comes from its self-attention layer that is made to capture long-range dependencies which means it requires a massive dataset for the transformer to learn meaningful representation. By combining the strengths of both models into a hybrid Transformer-RNN architecture, we can potentially overcome the individual weaknesses of each. The RNN's sequential nature allows it to capture fine-grained temporal dependencies (short term), while the Transformer's attention mechanism ensures the model can handle long-range dependencies more effectively. The hybrid model leverages the RNN's ability to handle small, sequential data points in conjunction with the Transformer's capacity to model global context**.** The Transformer processes the input sequence with batch parallelized computation, making it highly efficient for long-range dependencies.

Key components of the Transformer model include:

- **Self-Attention Mechanism**: This algorithm calculates the weighted sum of all input tokens for each position, allowing the model to capture token relevance without considering the relationship of distance from each input in the sequence.

The attention is computed using queries, keys, and values, typically through a dot product attention mechanism.

- **Multi Headed Attention**: Enhances the capability of transformers to recognize disinclination between the connection of input tokens, the computation is parallel across multiple heads. Each head performs independent attention and the outputs are concatenated and passed through a linear layer. This occurs by splitting the heads N number of times before passing the input vector into the attention block for initializing QKV vectors for attention processing.
- **Position-wise Feedforward Networks**: After attention, the outputs pass through position-wise feedforward networks consisting of two linear layers and a ReLU activation. These networks are responsible for further processing the result of the attention layer.
- **Positional Encoding**: Since the Transformer operates in parallel across all tokens, it does not inherently account for token order. To overcome this, positional encodings are combined to the token embeddings using addition to represent vectors for the order of tokens.
- **Residual Connections & Layer Normalization**: Training stabilizer to balance gradient flow from exploding/vanishing by preventing extreme values. This layer is employed after each attention and feedforward block.

RNN Model

These are sequential models that computes tokens in order, maintaining a hidden state to capture temporal dependencies. Despite their effectiveness in handling sequential data, RNNs often struggle with long-range dependencies due to issues like vanishing gradients. To mitigate this, Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) variants were developed, which improve the model's ability to capture long-range dependencies.

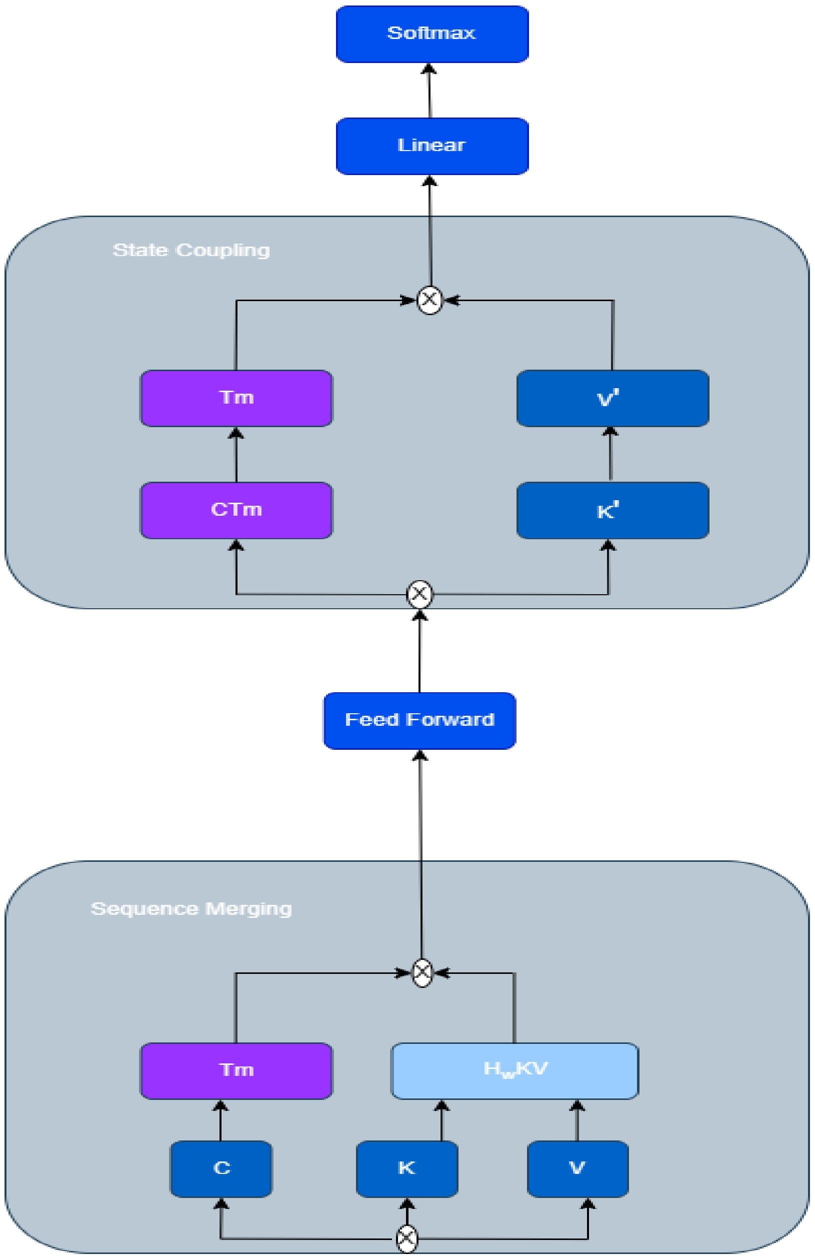
In our approach, we employ a custom Shift layer as a lightweight strategy for sequential processing The shift layer processes token embeddings by blending the current embedding with the previous one using a single learnable parameter The primary advantage of the Shift layer lies in its simplicity and efficiency: it captures a basic form of temporal dependency with significantly fewer parameters compared to the numerous weights in an RNN’s recurrent and input matrices. This reduction makes it computationally lighter and faster, avoiding the sequential bottlenecks of RNNs, especially for long sequences. While it sacrifices some of the expressive power of RNNs in modeling complex, long-range dependencies, its integration within our AttentionStreamBlock alongside SequenceMergingSeq and StateCoupling, complements this limitation by providing additional sequence-processing capabilities, making it a suitable choice for tasks where token order is critical yet computational efficiency is paramount.

Modification (Dot Product Sequential Reformulation Modified to Become a Linear Element-Directed Attention Mechanism)

In a standard Transformer model, the attention mechanism relies on dot product attention, it is when the Q (query), K(key), and V(value) are projected for each token. The computation of the weights uses dot product of queries and keys, followed by a SoftMax operation. This enables the model to assign different attention weights to different tokens, depending on their relevance to the current position. While highly effective, this approach has a few limitations, especially in sequence models where maintaining temporal dependencies is crucial. The standard dot product attention mechanism tends to lose the fine-grained sequential relationships between tokens, which are essential for tasks such as language modeling or sequence prediction.

Sequential Reformulation of Attention

Figure *4*. Linear Element Directed Attention



According to fig.4, it focuses on the Sequential Reformulation of Attention (SRA) block as a modification of the standard attention mechanism designed to capture both the sequential nature of token processing and the global dependencies within the sequence. This approach avoids the complexity of the traditional dot-product attention used in Transformers and instead leverages a series of linear transformations, time modulation, and careful integration of token representations. Below, we explain the core components and procedures of this modified attention mechanism in reference to the architecture. Attention process is partitioned into distinct steps that focus on different aspects of token interaction:

- **Sequence Merging Block**
- **Time Modulation (Tm)**
- **HwKV (Shift layer weight Key-Value Transformation)**
- **Feedforward Layer**
- **State Coupling Block**

The overall flow of information in this mechanism involves transforming hidden states through linear projections, applying time-modulated attention, and carefully merging key components to ensure that both sequential and global dependencies are captured efficiently.

Sequence Merging Block

The first step in the process begins with the output of the Shift layer (which processes tokens sequentially). The output from the Shift layer is passed to the sequence merging block for creation of attention components. The process begins on the weights from Shift Layer (representing token-wise information) is processed and split into three distinct vectors: C, K, and V. These vectors represent the core components required for the attention mechanism, where:

- **C** (Terminal Vector) is used to encode the token's current state.
- **K** (Key Vector) encodes contextual information for attention computation.
- **V** (Value Vector) holds the actual information content that will be referenced based on attention scores.

Time Modulation (Tm)

After obtaining the **C** (Terminal Vector), it is passed through a Time Modulation (Tm) step. Time modulation that adjusts the attention scores dynamically through time based on sequential dependencies. This allows the model to regulate how strongly each token’s attention should be influenced by previous tokens in the sequence, helping to retain sequential information.

- **C** is modulated through **Tm** to adjust its relevance in the attention mechanism. This process modifies the token representation based on its position in the sequence and its relationship to preceding tokens.
- **Tm** is used to dynamically adjust the relevance of each token based on its relative position and sequence context.

Shift Layer Weight Key-Value Transformation (HwKV)

While the **C** vector undergoes time modulation, the **K** (Key Vector) and **V** (Value Vector) are processed separately through a hidden weight transformation, called **HwKV**. The transformation combines these components with other information to refine the attention mechanism and improve the token-context relationship.

- **K** and **V** are transformed via learned linear projections into new forms (**K'** and **V'**).
- These transformed **K'** and **V'** are designed to focus attention based on both the local context (current token) and global context (previous tokens) to effectively capture dependencies within the sequence.

The goal of **HwKV** is to ensure that the model is able to adaptively learn the contextual relevance of each token (through **K'**) and adjust its response (via **V'**) to generate precise attention scores.

Feedforward Layer

Once the vectors are processed through the sequence merging, time modulation, and hybrid transformations, they are passed through a **Feedforward Layer**. This feedforward block processes the combined attention vectors. This layer is a conventional approach standardized in transformer architecture to create nonlinear transformations of the attention scores. The feedforward layer typically consists of two linear layers where the first is before ReLU activation function which is positioned in between and the second is linear layer is placed after the activation function, providing nonlinear transformations of the token representations. This layer allows for additional processing of token-wise information, enabling the model to build higher-level abstractions.

State Coupling Block

After the feedforward layer processes the attention vectors, the output is passed to the **State Coupling Block**. This block ensures that the information is properly consolidated and mapped into a form that can be used for final token predictions or classification. The output of the feedforward layer is split into two parts:

- Ctm: The modulated **C** (Terminal Vector) from the previous block.
- K': The transformed Key Vector, which was passed through time modulation in the previous step.

The Ctm vector is time-modulated again through Tm, adjusting its relevance based on the current token's sequence position. Finally, K' and Ctm are concatenated by addition and passed through the final transformation, which applies the last time modulation step to merge the information from both vectors.

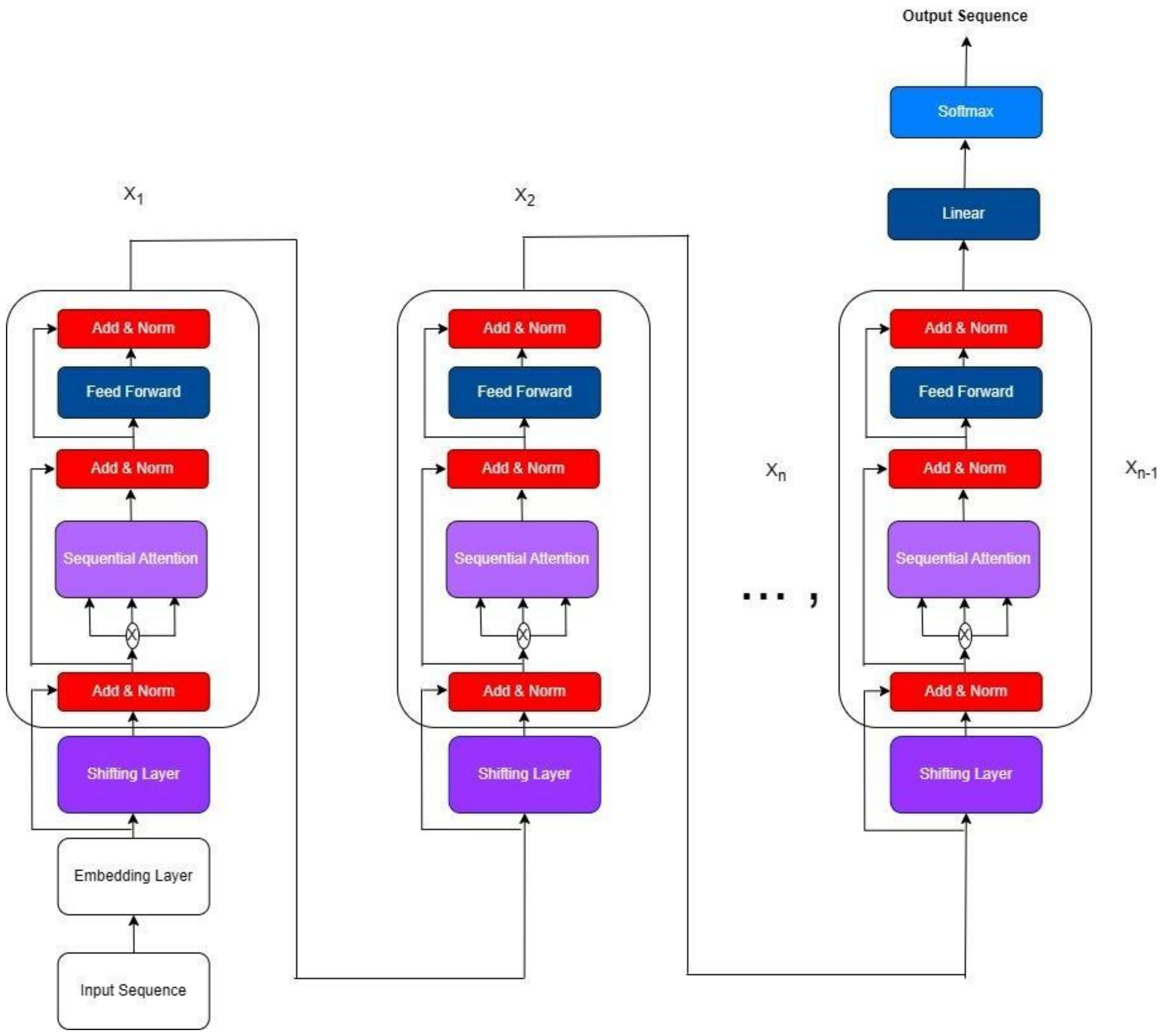
Final Output

After the **State Coupling** step, the result is computed by passing the merged vectors through a Linear Layer. This linear transformation serves as a crucial step in converting the high-dimensional representations into a usable form for prediction, classification, or sequence generation. The linear layer applies a learned weight matrix to the output vectors, mapping them to the appropriate output space for the downstream task.

Token-level Output: The linear layer transforms the token-level representations into a probability distribution over the vocabulary, which is used to predict the next token (or sequence of tokens). This is calculated by passing linear layer's classifier output to a SoftMax activation to obtain a probability distribution, where each token in the vocabulary has an associated probability. The model then samples from a pool of output probabilities for the highest score as the next token prediction.

Output Model (Transformer-RNN Hybrid Model)

Figure 5. Transformer-RNN Hybrid model.



The **Transformer-RNN Hybrid Model** combines the strengths of RNN to Transformers to capture both sequential dependencies and global context within a sequence. Below is a detailed breakdown of how the model processes an input sequence, step by step, to generate output probabilities using a sequence of layers and operations.

Input Sequence and Embedding Layer

The model begins by receiving an input sequence of tokens, where each token in the sequence is represented as a discrete value (e.g., indices of words or characters). The first token, denoted as **X1**, is passed to the embedding layer. This layer transforms the token into a continuous vector representation, which captures semantic information about the token in a high-dimensional space.

Shift Layer (First Pass)

After the token embedding for is obtained from the nn.Embedding layer, it is passed through a Shift layer within the first AttentionStreamBlock. The Shift layer processes the embedded sequence by blending the current token's embedding with the embedding of the previous token (if available), using a single learnable parameter μ\muμ. For the first token, where no prior token exists, the output is simply the embedding of X1.

Add & Norm Layer

After processing through the Shift layer, the result is passed to an Add & Norm layer. This layer applies a residual connection, combining the original and modified embeddings to ease gradient flow, and layer normalization to stabilize training and optimize performance.

Add: The layer normalized result of the Shift Layer is added with the previous output of the Shift layer (the token's embedding) through a residual connection.

- **Norm**: The sum is then normalized across the feature dimension, ensuring the output distribution remains stable.

Sequential Reformulated Attention Block

Next, the result from computation of Add & Norm layer is directed towards the Sequential Reformulated Attention Block. Here, the hidden state is split into three components: **C** (Terminal vector), K (Key vector), and **V** (Value vector).

- C is processed through time modulation (Tm), which adjusts the relevance of the token based on its sequential position and previous tokens.
- K and V are transformed through the HwKV (Hybrid Key-Value Transformation) to refine their contextual encoding.

The weights of the attention score are calculated by comparing C, K, and V, where the model calculates how much attention each token should pay to the others in the sequence based on their relevance. This attention is then used to generate new token representations.

Attention Score Computation (Sequence Merge and State Coupling)

Within the attention block, two key operations take place:

- **Sequence Merging:** The attention score is computed sequentially, ensuring that each token attends to the previous tokens (this preserves the RNN-like sequential processing).

- **State Coupling:** After computing the attention score, the token representations are combined (via addition or concatenation) and passed through a state coupling block, which further refines the output for classification or prediction tasks.

Add & Norm Residual Layer (After Attention Block)

Once the attention scores are computed and the resulting representations have been processed, they are passed through another **Add & Norm** residual layer. This ensures that the output remains stable to make the model less sensitive to changes in input distribution.

Feedforward Layer

The layer normalized state coupling scores proceeds to a Feedforward network. This has a ReLU placed at the center of two feedforward networks. The feedforward network provides an additional layer of abstraction, allowing the model to learn more complex representations of the token information.

Add & Norm Residual Layer (After Feedforward)

After the feedforward layer, the output undergoes another Add & Norm operation, ensuring that the output is properly normalized and stabilized before the next token is processed.

Shift Layer (Nth Step)

After processing X1, the next token X2 is handled. The output from the prior Add & Norm layer is fed into the Shift layer, blending it with the current embedding using μ\muμ. This step processes tokens sequentially, lightly incorporating prior context.

Repeat the Process for All Tokens

This process repeats for each token (X2, X3, etc.), passing through the Shift layer, Add & Norm, SequenceMergingSeq, StateCoupling, and residual connections. Each token leverages prior context efficiently, combining lightweight shifting with attention-like mechanisms.

Final Output (After Last Token)

Once the last token is processed, the transformed sequence is encoded with both shifting and attention features. The final output goes through a linear layer, reducing its dimensionality for tasks like token prediction or classification.

SoftMax and Output Probabilities

Finally, the calculated result from the linear layer proceeds for an activation using a SoftMax activation function. This converts the logits (raw predictions) into probabilities, allowing the model to make a final decision.

- **For Token Prediction**: The SoftMax output represents the output probabilities of the entire vocabulary for the next token. The calculation consists of sampling high probability token from the distribution.

Significance of the Study

This study addresses the pressing challenges associated with traditional transformer architectures, particularly their computational inefficiencies and high resource demands, which limit their applicability in low-resource environments. By focusing on the development of a hybrid transformer-RNN with linear inference complexity, this research contributes significantly to the ongoing discourse on optimizing transformer models for natural language processing (NLP). The introduction of sequential reformulations of the attention mechanism not only enhances computational efficiency but also maintains model performance, making sophisticated NLP tools more accessible across various fields, including those with limited data availability. Ultimately, this work aims to democratize access to advanced language technologies, facilitating small-scale learning and enabling the deployment of effective models in resource-constrained settings. By demonstrating that compact transformers can outperform traditional architectures even on minimal datasets, this research paves the direction for further innovations in the design of smarter and lighter neural networks, addressing both practical and theoretical aspects of NLP.

Scope and Limitations

This study focuses on the design and evaluation of compact transformer architectures specifically tailored for natural language processing operations. The primary objective is to achieve linear inference complexity through novel sequential reformulations of the attention mechanism, allowing for efficient model performance even in resource-constrained environments. The research will investigate various NLP applications, including text generation and classification, while assessing the proposed models' capabilities in handling longer sequences and maintaining semantic integrity.

However, the study focuses on the proposed methods that will involve trade-offs in terms of model complexity and performance, and these compromises will be analyzed but not exhaustively explored across all possible scenarios. Additionally, while the study aims to demonstrate the applicability of compact transformers, its findings may not generalize to all transformer-based models or to other domains such as computer vision, where different constraints and challenges exist.

Operational Definition of Terms

The following terms are defined according to how they are used in the study.

Transformer - A state of the art architecture made for sequence modeling, notable for utilizing self-attention mechanisms that enable parallel processing of input data, improving efficiency in natural language processing.

Self-Attention Mechanism - A component of Transformers that enables the model to quantify the emphasis to the scores of individual tokens in a sequence when inferencing representations, with quadratic computational complexity growing in direct proportion with the quantity of tokens.

Computational Complexity - A measure of the resources (time and space) required by an algorithm to process input data, often described in terms of big O notation, indicating how resource requirements scale with input size.

Sequential Reformulations - Is a proposed method aimed at altering the standard attention mechanism to achieve linear time complexity while enabling data parallelization similar to standard transformer attention layer, thereby improving efficiency and scalability of transformer models.

Tokenization – It is a procedure of partitioning text from a sequence into smaller elements, which can consist of characters, words, or sub words. This step is crucial for transforming the dataset into a format that the model can read, allowing for the vectorization of characters into numerical representations.

Embeddings - These are numerical vector representations of data elements, such as words, characters, or tokens, within a continuous, high-dimensional space. The objective of embeddings is to capture and encode the semantic meaning or contextual relationships between these elements, providing a neural network to process and comprehend patterns in dataset. In modeling language, embeddings enable the representation of similar words or characters to be located near each other in the vector space, enhancing learning and generalization for sequence generation, classification, and translation tasks.

SoftMax - SoftMax is a function that transforms a vector of real numbers into a probability distribution, where each value corresponds to the likelihood of the element that a certain token is a constituent of a particular class.

ReLU - Also known as "Rectified Linear Unit", is a kind of activation function used in neural networks that orients non-linearity to the model by displaying the input if it's positive and 0 if false.

Recurrence - Is a concept where the output of a system at a given time step is influenced by its previous outputs, enabling recurrent neural networks to process sequential data by "remembering" past inputs.

Computational Cost - Refers to the amount of resources, such as time and memory, required to execute an algorithm or process, which is influenced by factors like the number of operations, data size, hardware limitations, and algorithm complexity.

Time Modulation - Is a technique used to adjust the speed or duration of a signal or process over time, with applications in audio processing, signal processing, and machine learning.

State Coupling- Is a technique in machine learning where two or more classes with similar characteristics are combined into a single channel to improve model performance, handle imbalanced datasets, and increase robustness.

Sequence Merging - Sequence merging is the process of combining two or more sequences into a single, longer sequence.

Linear Projection - Linear projection is a transformation that maps data from a higher-dimensional space to a lower-dimensional space while preserving linear relationships, used for feature extraction alongside visualization.

Adam (Adaptive Moment Estimation) **Optimizer** - A neural network training algorithm that effectively updates the weights during training process. This optimization algorithm dynamically adjusts the learning rate for training each individual parameters of the model.

Forward Pass - The forward pass is the process of moving an input to the network's output layers to generate predictions.

Hidden Layer – This defines as an intermediate that is positioned in between input layer and output layer. This is where computation and feature extraction occur.

Element-Directed Attention – It is a reformulated attention mechanism for neural networks where the focus is directed towards specific tokens of the input based on the target class, improving model accuracy and interpretability.

Cross-Entropy Loss – Defines as a computation of the loss function to determine the divergence between the predicted distribution of labels and the actual distribution, it is used to assess the model’s accuracy performance in predicting tokens.

Residual Layer - A residual layer, a key component of residual neural networks (ResNets), allows the model to learn an "identity funcmtion" as a shortcut, enabling it to easily learn the original input as the output and helping to alleviate the vanishing/exploding gradient problem.

Layer Normalization – It is a method implemented to stabilize and speed up the training of deep neural networks by normalizing the activations within a layer, helping to resolve problems such as internal covariate shift.

Positional Encoding - Positional encoding is a component of transformers that provide information regarding the positional context of input sequences, thereby allowing the model to retain spatial relationships position of each word or element in a sequence, which is crucial because transformer models lack inherent order information.

Linear Layer - It defines the fundamental foundation of neural networks by processing the transformation of the input data by calculating the product of the weight matrix then added with a bias vector.

Method

Materials

Hardware

The hardware specification for this study includes a laptop equipped with Intel(R) Core (TM) i5-13420H Processor, NVIDIA RTX 4050 GPU, 8GB RAM, and 64-bit Windows 11 English operating system.

Software

The software specifications for this study includes PyTorch framework and Visual Studio Code

- **Programming Language**: Python with PyTorch was chosen for its rich ecosystem and seamless CUDA support, enabling efficient model development and training. Python offers a wide range of libraries like NumPy, pandas, and Matplotlib for data manipulation, preprocessing, and visualization, while PyTorch provides GPU acceleration via CUDA, speeding up training by leveraging parallel computation.
- **Libraries/Frameworks**
    - **PyTorch**: The primary deep learning framework used to build, train, and optimize the model. It handles tensor operations, model layers, and provides support for GPU acceleration through CUDA.
    - **torch.utils.data:** Used to define and manage datasets through the Dataset class, and to efficiently load data in batches using the DataLoader class, which is essential for training large datasets like text.
    - **torch.nn:** Contains essential neural network modules like Embedding, forward pass layers, and Linear layers, which are used to build the architecture of the model like embedding characters, and applying linear transformations.
    - **torch.optim**: Provides optimization algorithms, like Adam (optim.Adam), for updating model parameters based on the loss function during training.

  - **OS**: Used for file handling, specifically loading the text data from files and saving the trained model's state.
  - **NumPy**: Used for its support for array manipulations and numerical operations required by PyTorch.

Data

- **Source and Composition**: The dataset is sourced from [Kaggle’s 3K Conversation Dataset]. It contains 3724 rows with 2 features corresponding to input and response pair.
- **Key Variables**: Input and Response variables.

Procedures

Data Gathering

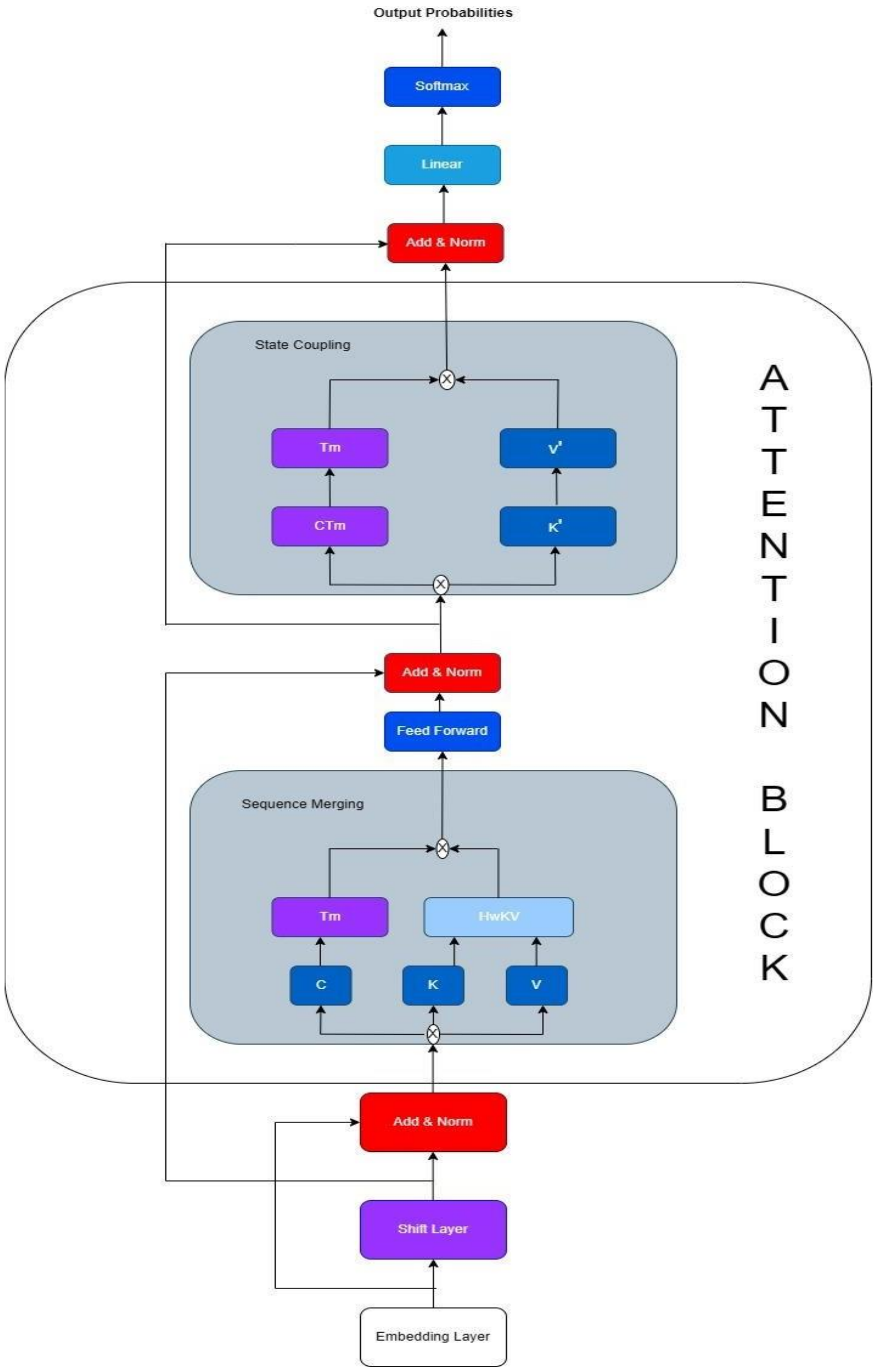
The data gathering process begins with collecting text data from a designated folder named 'text-data'. This folder contains a collection of text files that serve as the primary corpus for the project. All files in this folder are systematically processed to ensure compatibility with various encoding formats, including UTF-8, ISO-8859-1, and cp1252. This step is crucial for accommodating diverse text file types and minimizing the risk of data loss due to encoding errors. The contents of successfully processed files are compiled into a single dataset, forming the foundation for subsequent analysis and model development. Files that cannot be read with the specified encodings are excluded from the dataset to maintain data integrity. By implementing this thorough and inclusive approach, the process ensures a rich and diverse corpus, which is vital for building a robust and effective model.

Pre-processing

Pre-processing is an essential step in preparing raw text for model training. It involves transforming the text into a format that can be efficiently processed by the model. This process begins with constructing a vocabulary from the provided text, where each unique character is assigned a numerical index. The text is then processed into an ordinal array of indices, creating a numerical representation that facilitates efficient processing. The dataset is further structured to generate input sequences of a specified length along with their corresponding target sequences. To guarantee optimal model training, the dataset is split in two, with one part dedicated to training the model and the other for evaluation. with uneven proportion making 80% of the data allocated for training, while the minority 20% is reserved for validation testing. This pre-processing approach ensures the text data is optimally prepared for training and evaluating the model.

Building the Model (Application of the Algorithm)/Simulation Process

Fig 6. Hybrid Transformer-RNN Model Instance Architecture.

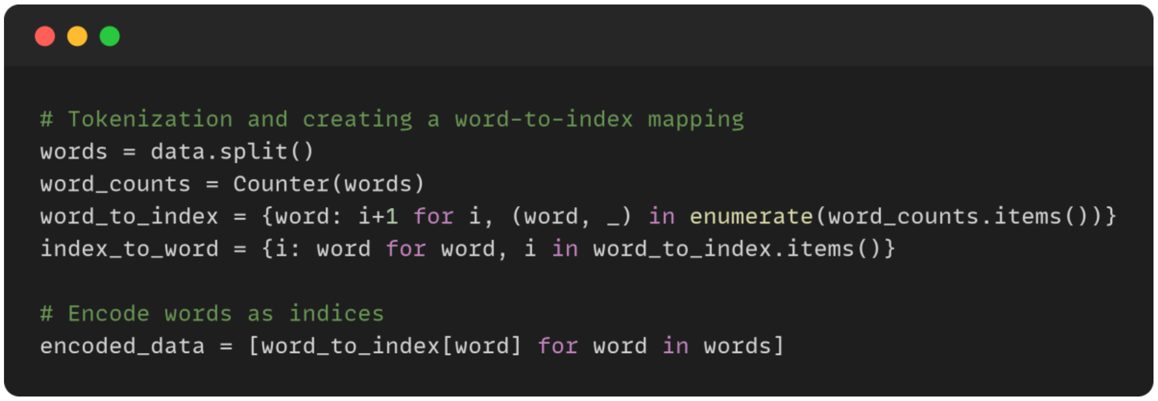


According to Figure 6, the "AttentionStream" Model architecture blends lightweight Shift-based sequential processing with attention-like mechanisms. Designed for sequential data, it efficiently transforms context to predict outputs for tasks like token generation or classification. Below is a concise breakdown of the architecture and data flow through each layer.

1. Input Sequence and Embedding Layer

The model begins by accepting an input sequence of tokens. Each token in the sequence is passed through an Embedding Layer that converts each token (usually represented as discrete indices) into dense vectors. The embedding layer encodes each token into a high-dimensional continuous space that captures semantic connections of words to create a continuous vector representation for all input tokens.

Fig 7. Tokenization and encoding

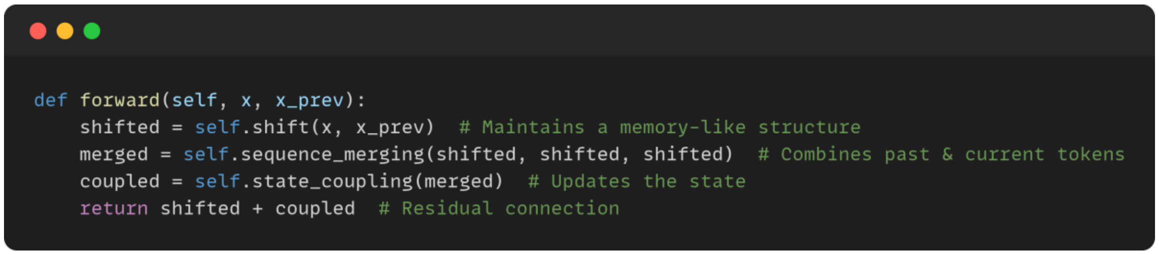


```
# Tokenization and creating a word-to-index mapping
words = data.split()
word_counts = Counter(words)
word_to_index = {word: i+1 for i, (word, _) in enumerate(word_counts.items())}
index_to_word = {i: word for word, i in word_to_index.items()}

# Encode words as indices
encoded_data = [word_to_index[word] for word in words]
```

2. Shift Layer (Sequential Processing)

Once the tokens have been embedded, the sequence is passed into a Shift Layer. processes the embedded sequence by blending the current token's embedding with information from the previous token's embedding (if available), using a learnable parameter mu. it computes a weighted combination of the current embedding x and the previous embedding x_prev (set to None in the initial pass if there is no prior token the embedding output is the current token x. The result is a shifted embedding vector that prepares the sequence for subsequent processing while introducing a controlled dependency on prior context.

Fig 8. Shift layer implementation



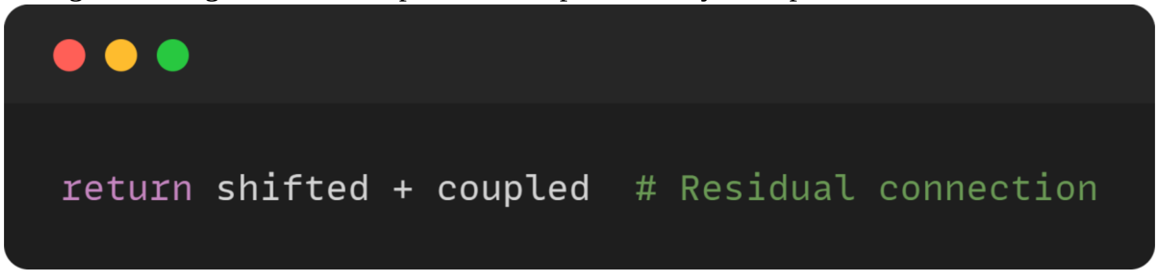
```
def forward(self, x, x_prev):
    shifted = self.shift(x, x_prev)  # Maintains a memory-like structure
    merged = self.sequence_merging(shifted, shifted, shifted)  # Combines past & current tokens
    coupled = self.state_coupling(merged)  # Updates the state
    return shifted + coupled  # Residual connection
```

3. Add & Norm (Residual Connection + Normalization)

After the RNN processes the sequence, the output hidden states are passed through an Add & Norm layer. This layer involves the following steps:

- **Residual Connection**: The Shift output is added to its input (token embedding), preserving the original representation and aiding gradient flow for efficient learning.

Fig 9. Adding of shifted output with the previous layer output



```
return shifted + coupled  # Residual connection
```

- **Layer Normalization**: This normalizes the output across the feature dimension, improving training stability and helping to avoid issues like exploding gradients.
- **Input**: RNN output
- **Output**: Residual connection followed by layer-normalized output

4. Sequential Reformulated Attention Block

After the Add & Norm operation, the model enters the Sequential Reformulated Attention Block, which combines the benefits of both attention mechanisms and sequential processing.

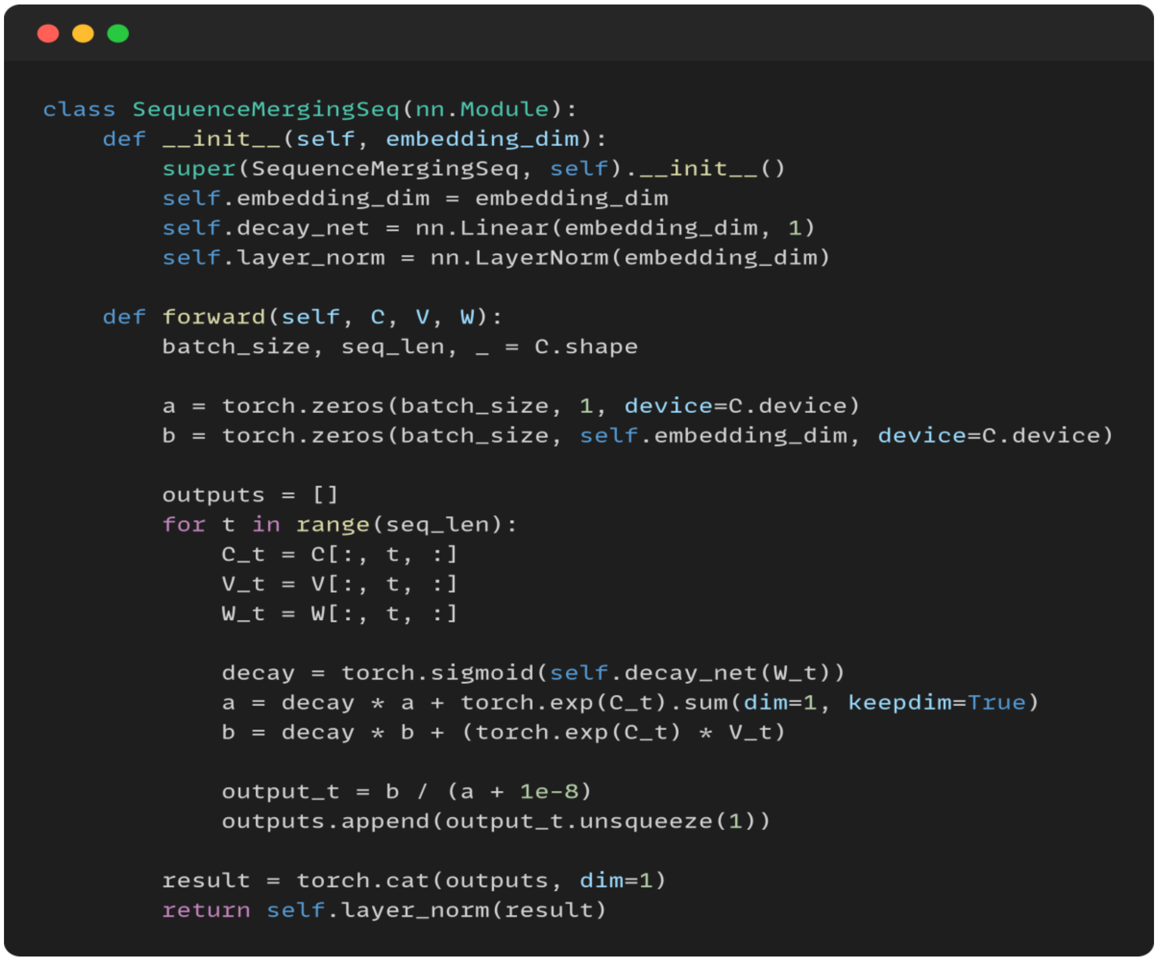
- **Sequence Merging**: The hidden state vector from the RNN is split into three components:
    - **C (Terminal vector)**: This vector represents the current token's information and is used to compute attention.
    - **K (Key vector)**: This vector represents the token's key information, which is used for comparison in the attention mechanism.
    - **V (Value vector)**: This vector represents the token's value, which is used to produce the weighted output after computing attention scores.

The Shift layer output is split into these components to ensure that each part of the token's information is handled in a specialized way.

- **Time Modulation (Tm)**: The **C** vector (Terminal vector) undergoes **Time Modulation (Tm)**, which adjusts the importance of each token's representation based on its sequential position, adding an additional dynamic aspect to the model.

- **HwKV (Hidden weight Key-Value Transformation)**: The **K** and **V** vectors are processed by the **HwKV** transformation using the shift layer output, This helps the model adapt the attention mechanism to the specific characteristics of the token's context.
- **Concatenation**: The time-modulated **C** vector (C_tm) and the processed **K** and **V** vectors are concatenated together to form the final input to the feedforward layer.
- **Input**: Shift layer output (split into C, K, and V)
- **Output**: Concatenated output of time-modulated C and HwKV processed K and V

Fig 9. Sequence Merging block



```
class SequenceMergingSeq(nn.Module):
    def __init__(self, embedding_dim):
        super(SequenceMergingSeq, self).__init__()
        self.embedding_dim = embedding_dim
        self.decay_net = nn.Linear(embedding_dim, 1)
        self.layer_norm = nn.LayerNorm(embedding_dim)

    def forward(self, C, V, W):
        batch_size, seq_len, _ = C.shape

        a = torch.zeros(batch_size, 1, device=C.device)
        b = torch.zeros(batch_size, self.embedding_dim, device=C.device)

        outputs = []
        for t in range(seq_len):
            C_t = C[:, t, :]
            V_t = V[:, t, :]
            W_t = W[:, t, :]

            decay = torch.sigmoid(self.decay_net(W_t))
            a = decay * a + torch.exp(C_t).sum(dim=1, keepdim=True)
            b = decay * b + (torch.exp(C_t) * V_t)

            output_t = b / (a + 1e-8)
            outputs.append(output_t.unsqueeze(1))

        result = torch.cat(outputs, dim=1)
        return self.layer_norm(result)
```

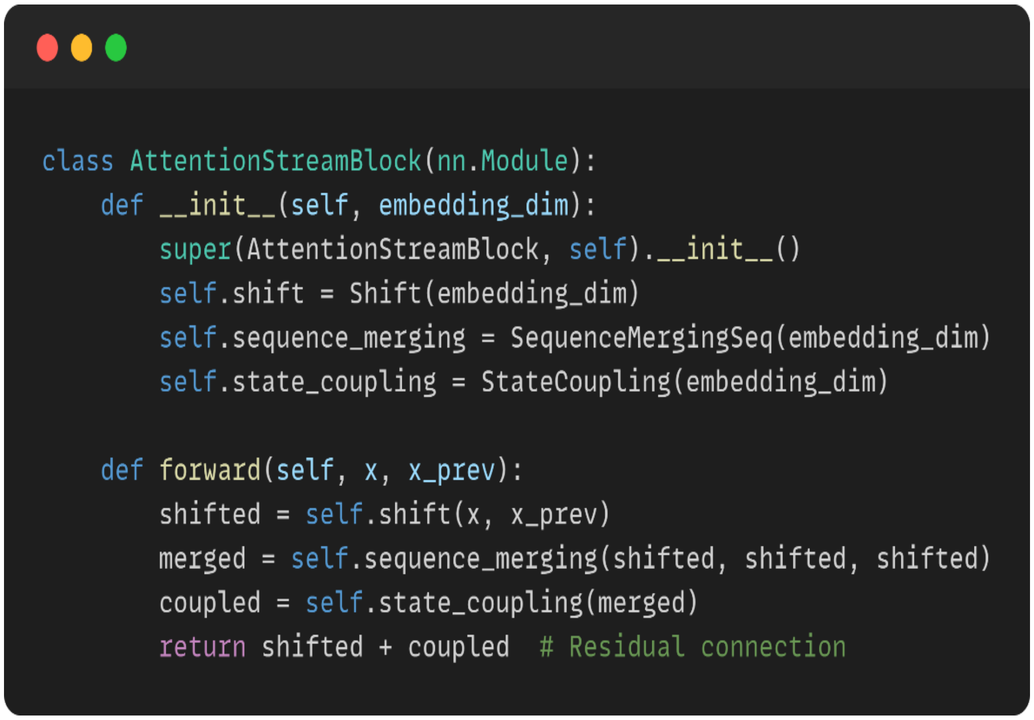
5. Feedforward Layer

Once the attention components have been merged and processed, the concatenated vector from the attention layer proceeds for processing in the feedforward network. This layer consists of a fully connected neural network, typically involving a two-layer structure with an activation function (such as ReLU) in between. The feedforward layer allows the model to perform additional non-linear transformations of the token representation, improving the model's capacity to capture complex relationships in the data. The result would be the transformed representation of the token after feedforward processing

6. Add & Norm (Residual Connection + Normalization)

The processing result from the feedforward layer proceeds once more to an Add & Norm layer, similar to previous steps. This residual connection ensures that the model can learn both the transformed and original representations of the data. The normalization helps stabilize the output and improves convergence during training.

Fig 10. Attention Stream model



```
class AttentionStreamBlock(nn.Module):
    def __init__(self, embedding_dim):
        super(AttentionStreamBlock, self).__init__()
        self.shift = Shift(embedding_dim)
        self.sequence_merging = SequenceMergingSeq(embedding_dim)
        self.state_coupling = StateCoupling(embedding_dim)

    def forward(self, x, x_prev):
        shifted = self.shift(x, x_prev)
        merged = self.sequence_merging(shifted, shifted, shifted)
        coupled = self.state_coupling(merged)
        return shifted + coupled  # Residual connection
```

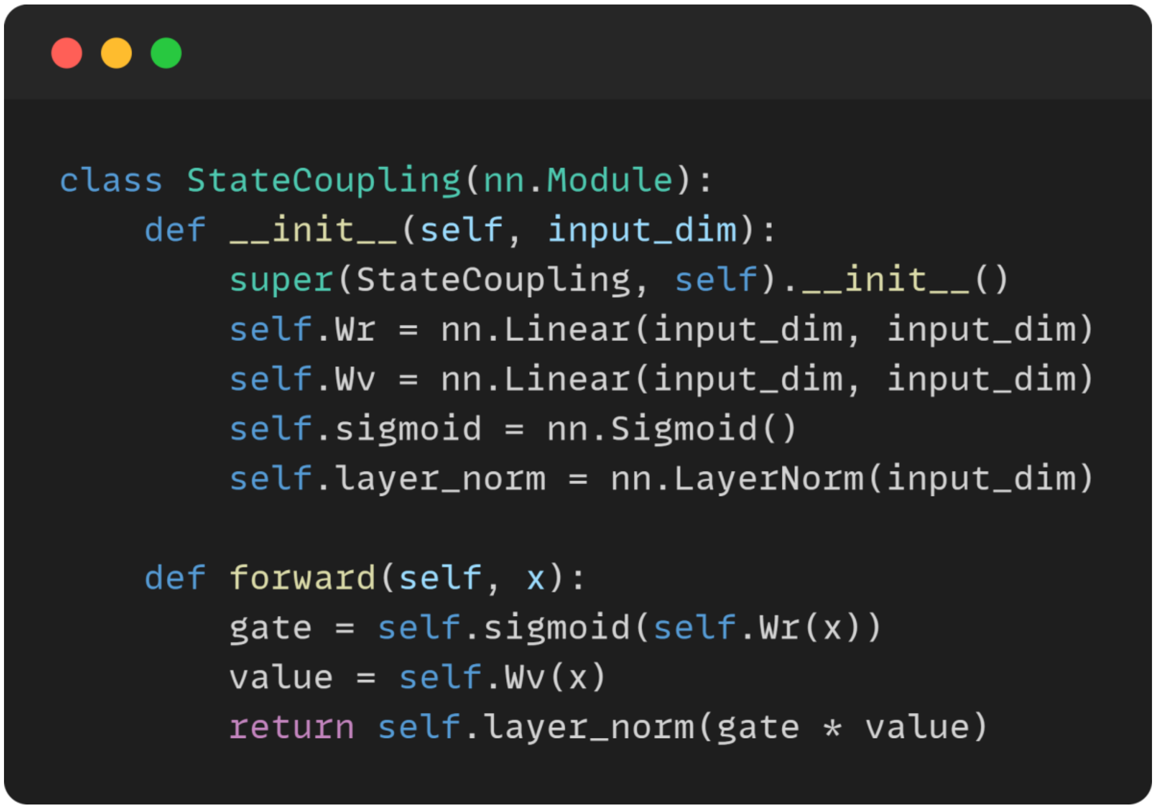
7. State Coupling Block

The normalized scores from the Add & Norm layer are then directed to the state coupling block, which further refines the token representations. This block involves the following steps:

- **Splitting the Output**: The output is split into two components:
    - **C_tm**: The time-modulated output from the sequence merging step, which contains the adjusted token representation.
    - **K'**: A new key representation, computed from the previous block's output.
- **Time Modulation (Tm)**: The C_tm vector undergoes Time Modulation (Tm) again, adjusting the token's relevance based on its position in the sequence and its context from the previous layers.
- **Passing to** V': The K' vector is passed to the V' transformation, which refines the value associated with the token.
- **Concatenation**: The time-modulated C_tm and the processed V' vectors (derived from K') are concatenated to create the final output of the state coupling block.
- **Input**: Output from the Add & Norm layer

The output of this step will be the merged output from time-modulated C_tm and processed V'.

Fig 11. State Coupling block



```python
class StateCoupling(nn.Module):
    def __init__(self, input_dim):
        super(StateCoupling, self).__init__()
        self.Wr = nn.Linear(input_dim, input_dim)
        self.Wv = nn.Linear(input_dim, input_dim)
        self.sigmoid = nn.Sigmoid()
        self.layer_norm = nn.LayerNorm(input_dim)

    def forward(self, x):
        gate = self.sigmoid(self.Wr(x))
        value = self.Wv(x)
        return self.layer_norm(gate * value)
```

8. Add & Norm Residual Layer (Final Step)

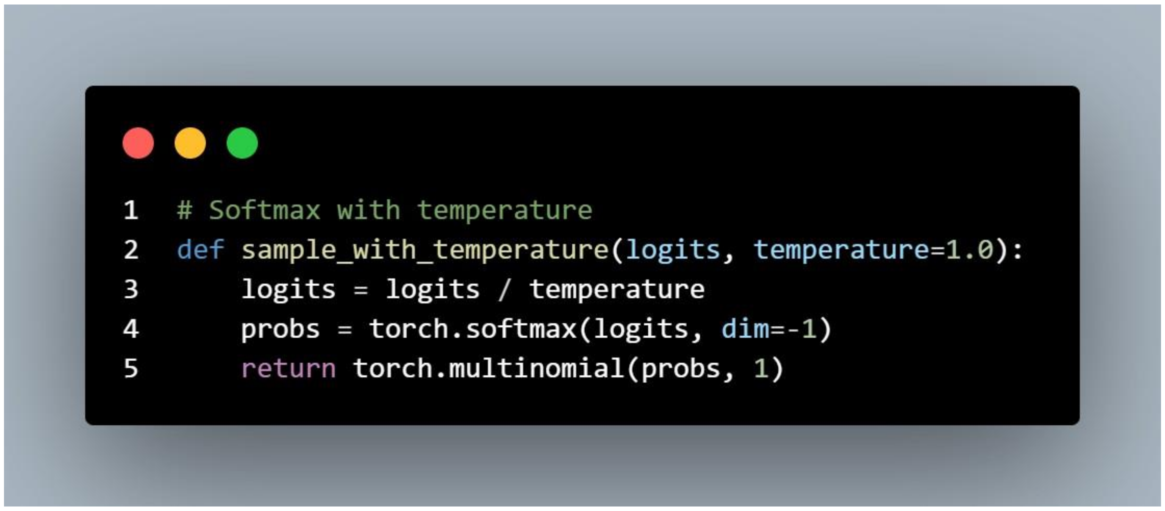
Once the final merged output is computed, the state coupling Block output is passed through another Add & Norm layer. This ensures that the model maintains stable training and optimizes the flow of information through residual connections. The result will be the layer normalized final output from the residual connection.

9. Linear Layer (Projection to Output Space)

The last computation of the Add & Norm layer is forwarded to a Linear Layer. This component reduces the dimensionality of the output to match the desired output space, which could represent either the number of possible token predictions (in a language modeling task) or the number of classes or distinct categories that a model has to identify (in a classification task) influences the model's output, which will be the 'logits' representing the unscaled prediction values for each possible token or class

10. SoftMax Function

Fig 12. SoftMax used in sampling logits with temperature



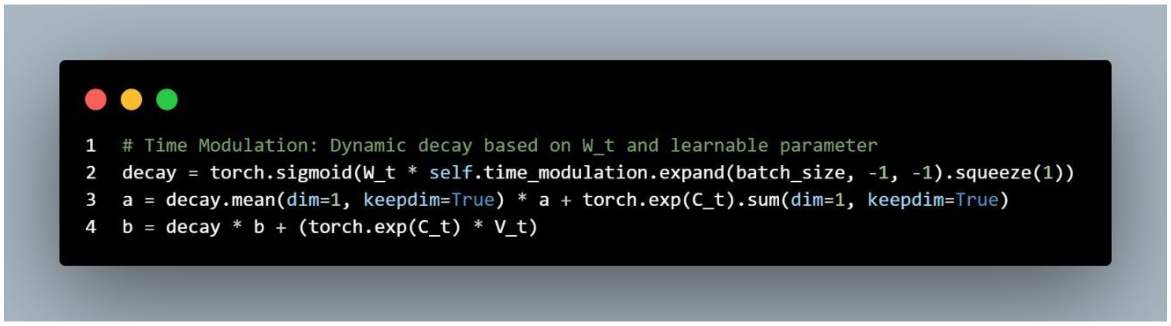
```python
# Softmax with temperature
def sample_with_temperature(logits, temperature=1.0):
    logits = logits / temperature
    probs = torch.softmax(logits, dim=-1)
    return torch.multinomial(probs, 1)
```

To convert the raw logits into interpretable probabilities, the logits from the linear layer is passed through a SoftMax function. The SoftMax converts the logits into a pool of output probabilities, ensuring that the sum of all probabilities for a token or class is equal to 1. The model can then select the most likely token as its prediction, or sample from the pool scores to come up with the next token prediction. The result will be the weighted sum of the tokens in a probability distribution.

9.) **Time Modulation**

Time Modulation in the SequenceMergingSeq layer, replacing a static decay network with a learnable parameter vector. This mechanism dynamically adjusts the influence of past tokens on the current output.

Fig 13. Time Modulation



```python
# Time Modulation: Dynamic decay based on W_t and learnable parameter
decay = torch.sigmoid(W_t * self.time_modulation.expand(batch_size, -1, -1).squeeze(1))
a = decay.mean(dim=1, keepdim=True) * a + torch.exp(C_t).sum(dim=1, keepdim=True)
b = decay * b + (torch.exp(C_t) * V_t)
```

Time Modulation acts like a dynamic attention mechanism, weighting the contribution of past information (a,b) based on the current token and the learned time modulation. It emphasizes or downplays tokens in a single pass, mimicking attention's focus without multi-head complexity used by transformers.

Evaluating the Model

The process begins with text generation, where an initial input string is provided, and the model predicts subsequent characters iteratively. This prediction process utilizes the SoftMax function to sample from the output probability distribution, enabling diverse and creative text generation. The generated outputs are assessed for their coherence, relevance, and overall quality in real-time scenarios. To quantitatively evaluate the model, several metrics are employed: Cosine Similarity, Perplexity, and ROUGE score. After text generation, both the model's output and reference texts are converted into fixed-size embeddings, either by averaging the embeddings or using the last hidden state. Cosine similarity is then calculated to measure the semantic alignment between the generated and reference texts. A high cosine similarity score indicates closer alignment in meaning.

Perplexity, a metric of uncertainty in the output predictions, is analyzed based on the probabilities assigned to each predicted word by the SoftMax function. Lower perplexity values signify the model's effectiveness in generating coherent and fluent text. Additionally, ROUGE score is applied to assess the overlap between the generated text and reference texts by examining n-gram matches, enabling an evaluation of textual similarity and relevance. The insights gained through these evaluations guide future refinements and enhancements to the model's architecture, ensuring continual improvements in text generation quality.

Perplexity

Perplexity is a metric used to evaluate how well a probabilistic model, like a language model, predicts a sequence of words. It measures the model's uncertainty by calculating the average number of choices it considers for each word, expressed as 2 entropy. A lower perplexity indicates the model is more confident and accurate in its predictions, while a higher value suggests confusion or poor fit to the data. For example, a perplexity of 10 means the model is as uncertain as if it were choosing randomly from 10 equally likely options per word.

$$H(P) = -\frac{1}{N}\sum_{i=1}^{N} \quad P(X_i)$$

Cosine Similarity

Cosine Similarity is a measure of similarity between two vectors, often used in text analysis to compare documents or embeddings. It calculates the cosine of the angle between the vectors, given by ∥A∥∥B∥A·B, where a value of 1 means identical direction (highly similar), 0 means perpendicular (unrelated), and -1 means opposite. In practice, it’s great for capturing semantic likeness in high-dimensional spaces, like word embeddings, regardless of their magnitude, making it ideal for tasks like document comparison or clustering.

$$Cosine\ Similarity = \frac{A}{||A||} \circ \frac{B}{||B||}$$

Rouge

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) is a set of metrics used to assess the quality of text summaries or translations by comparing a generated text to one or more reference texts. Variants like ROUGE-N (N-gram overlap) and ROUGE-L (Longest Common Subsequence) compute an F1-score, 2×Precision+Recall / Precision×Recall, based on matching words or sequences. It’s widely used in natural language processing to quantify how much of the reference content is captured, it focuses on surface-level overlap rather than semantics.

Rouge-N

ROUGE-N (where "N" stands for N-grams) measures the overlap of N-grams — sequences of N consecutive words—between a generated (candidate) text and a reference text. It’s a way to check how many chunks of words (e.g., unigrams for N=1, bigrams for N=2) match, focusing on exact, consecutive word sequences. The score is typically computed as an F1-score, balancing precision (how much of the generated text matches the reference) and recall (how much of the reference is captured in the generated text).

$$Rouge\ 1 = 2x\,\frac{\left(\frac{Matching\ Ngrams}{Generating\ Words}\right) \times \left(\frac{Matching\ Ngrams}{Reference\ Words}\right)}{\left(\frac{Matching\ Ngrams}{Generating\ Words}\right) + \left(\frac{Matching\ Ngrams}{Reference\ Words}\right)}$$

Rouge-L

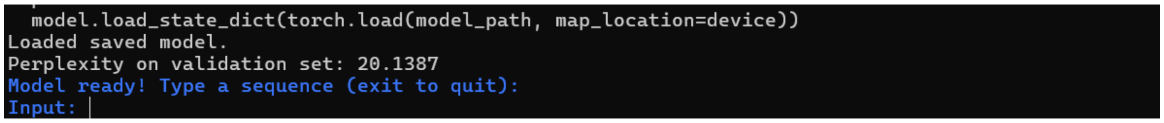
ROUGE-L (Longest Common Subsequence) evaluates similarity based on the longest sequence of words that appear in both the generated and reference texts, in the same order, but not necessarily consecutively. Unlike ROUGE-N, it doesn't require contiguous matches, making it more flexible for capturing structural similarity. It also uses an F1-score based on the length of this LCS.

$$Rouge\ L = 2x\,\frac{\left(\frac{LCS\ Length}{Generated\ Bigrams}\right) \times \left(\frac{LCS\ Length}{Reference\ Words}\right)}{\left(\frac{LCS\ Length}{Generated\ Words}\right) + \left(\frac{Matching\ Ngrams}{Generated\ Words}\right)}$$

Results and Discussion

The hyperparameters in the model, including embedding dimension, batch size, learning rate, number of layers, and epochs, play a significant role in its performance. The embedding dimension is set to 50, balancing model expressiveness and computational efficiency, while the batch size of 256 ensures efficient training without memory overload. The learning rate of 0.001 is a typical value for Adam optimization, providing a good trade-off between fast convergence and stability. With a single layer (num_layers = 1), the model remains simple, appropriate for this task, but could be adjusted for more complex problems. Training for 180 epochs allows sufficient time for the model to learn, although the optimal number could vary depending on the dataset. The use of Adam and CrossEntropyLoss ensures efficient optimization and appropriate loss calculation for the task, though further tuning of these hyperparameters might improve performance depending on validation results.

Fig 14. Model Perplexity

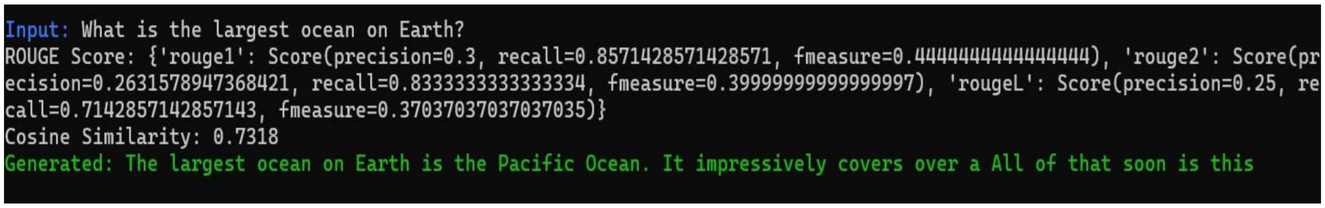


```
  model.load_state_dict(torch.load(model_path, map_location=device))
Loaded saved model.
Perplexity on validation set: 20.1387
Model ready! Type a sequence (exit to quit):
Input: 
```

The perplexity of the model on the validation set is 20.1387, indicating how well the model predicts the next word in a sequence. Lower perplexity values typically suggest better predictive performance, though the value here implies there is room for improvement. This metric reflects the model's ability to generate coherent text, with a lower perplexity indicating more accurate predictions

The model’s performance was evaluated using ROUGE scores, perplexity, and cosine similarity metrics. The model utilizes a text corpus as its dataset, which is partitioned into an 80% training and 20% validation split to train and assess its capabilities. Designed with a lightweight Shift layer and sequentially reformulated attention mechanism within the “SequenceMerging” component, the Attention Stream Model was tested on a text file containing kaggles 3K dataset and some factual question and answering texts. The discussion will examine the effectiveness of the "AttentionStream" model in focusing on relevant tokens in a single pass, and potential areas for improvement.

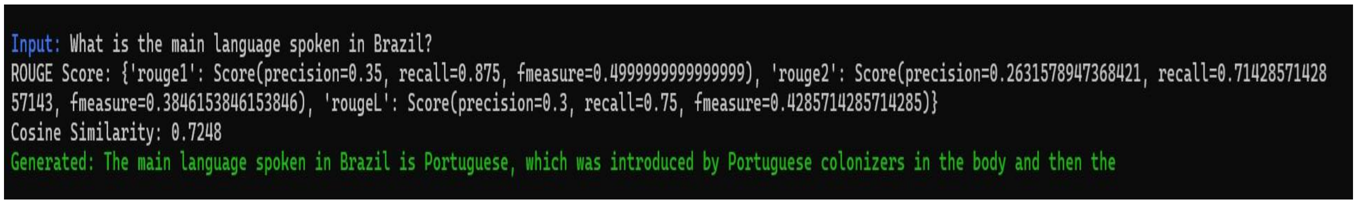
Fig 15. Model performance evaluation 1



```
Input: What is the largest ocean on Earth?
ROUGE Score: {'rouge1': Score(precision=0.3, recall=0.8571428571428571, fmeasure=0.4444444444444444), 'rouge2': Score(pr
ecision=0.2631578947368421, recall=0.8333333333333334, fmeasure=0.39999999999999997), 'rougeL': Score(precision=0.25, re
call=0.7142857142857143, fmeasure=0.37037037037037035)}
Cosine Similarity: 0.7318
Generated: The largest ocean on Earth is the Pacific Ocean. It impressively covers over a All of that soon is this
```

For the query "What is the largest ocean on Earth?", the Attention Stream Model generated the response: "The largest ocean on Earth is the Pacific Ocean. It impressively covers over a ALL the that soon is this." The performance was evaluated using ROUGE scores and cosine similarity. The ROUGE scores were as follows: ROUGE-1 (Precision=0.3, Recall=0.8571, F-measure=0.4444), ROUGE-2 (Precision=0.2631, Recall=0.8333, F-measure=0.3999), and ROUGE-L (Precision=0.25, Recall=0.75, F-measure=0.375). The cosine similarity between the reference and generated embeddings was 0.7318.

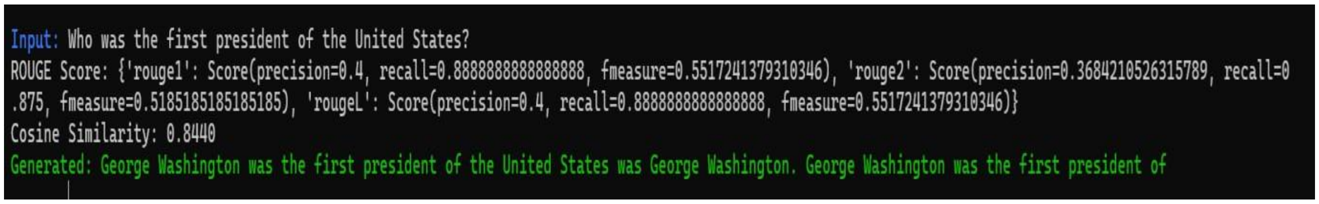
Fig 16. Model performance evaluation 2



```
Input: What is the main language spoken in Brazil?
ROUGE Score: {'rouge1': Score(precision=0.35, recall=0.875, fmeasure=0.4999999999999999), 'rouge2': Score(precision=0.2631578947368421, recall=0.71428571428
57143, fmeasure=0.3846153846153846), 'rougeL': Score(precision=0.3, recall=0.75, fmeasure=0.4285714285714285)}
Cosine Similarity: 0.7248
Generated: The main language spoken in Brazil is Portuguese, which was introduced by Portuguese colonizers in the body and then the
```

For the query "Who was the first president of the United States?", the Attention Stream Model generated the response: "George Washington was the first president of the United States was George Washington. George Washington was the first president of..." The model's performance was evaluated using ROUGE scores and cosine similarity, yielding the following metrics: ROUGE-1 (Precision=0.4, Recall=0.8889, F-measure=0.5517), ROUGE-2 (Precision=0.3684, Recall=0.875, F-measure=0.5185), and ROUGE-L (Precision=0.4, Recall=0.8889, F-measure=0.5517). The cosine similarity between the reference and generated embeddings was 0.8448.

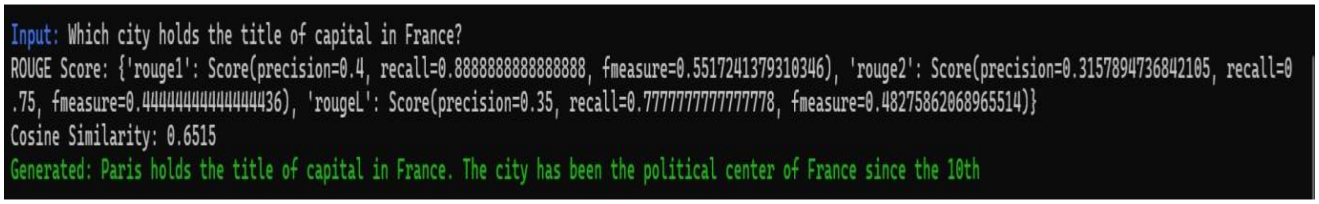
Fig 17. Model performance evaluation 3



```
Input: Who was the first president of the United States?
ROUGE Score: {'rouge1': Score(precision=0.4, recall=0.8888888888888888, fmeasure=0.5517241379310346), 'rouge2': Score(precision=0.3684210526315789, recall=0
.875, fmeasure=0.5185185185185185), 'rougeL': Score(precision=0.4, recall=0.8888888888888888, fmeasure=0.5517241379310346)}
Cosine Similarity: 0.8440
Generated: George Washington was the first president of the United States was George Washington. George Washington was the first president of
```

For the query "What is the main language spoken in Brazil?", the Attention Stream Model generated the response: "The main language spoken in Brazil is Portuguese, which was introduced by Portuguese colonizers in the body and then the..." The model's performance was evaluated using ROUGE scores and cosine similarity, yielding the following metrics: ROUGE-1 (Precision=0.35, Recall=0.875, F-measure=0.4999, though the provided F-measure of 8.4999 appears to be a typo and is likely intended as 0.4999 based on typical ranges), ROUGE-2 (Precision=0.2632, Recall=0.7143, F-measure=0.3846), and ROUGE-L (Precision=0.3, Recall=0.75, F-measure=0.4286). The cosine similarity between the reference and generated embeddings was 0.7248.

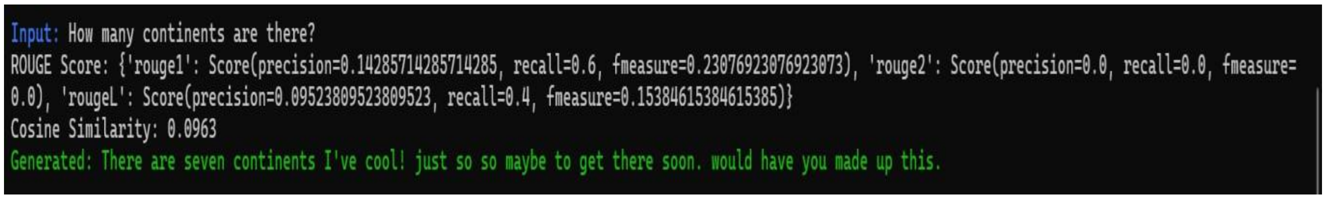
Fig 18. Model performance evaluation 4



```
Input: Which city holds the title of capital in France?
ROUGE Score: {'rouge1': Score(precision=0.4, recall=0.8888888888888888, fmeasure=0.5517241379310346), 'rouge2': Score(precision=0.3157894736842105, recall=0
.75, fmeasure=0.4444444444444436), 'rougeL': Score(precision=0.35, recall=0.7777777777777778, fmeasure=0.4827586206896514)}
Cosine Similarity: 0.6515
Generated: Paris holds the title of capital in France. The city has been the political center of France since the 10th
```

For the query "Which city holds the title of capital in France?", the Attention Stream Model generated the response: "Paris holds the title of capital in France. The city has been the political center of France since the 10th..." Performance was evaluated using ROUGE scores and cosine similarity, with the following metrics: ROUGE-1 (Precision=0.4, Recall=0.8889, F-measure=0.5517), ROUGE-2 (Precision=0.3158, Recall=0.75, F-measure=0.4444), and ROUGE-L (Precision=0.35, Recall=0.7778, F-measure=0.4828). The cosine similarity between the reference and generated embeddings was 0.6515.

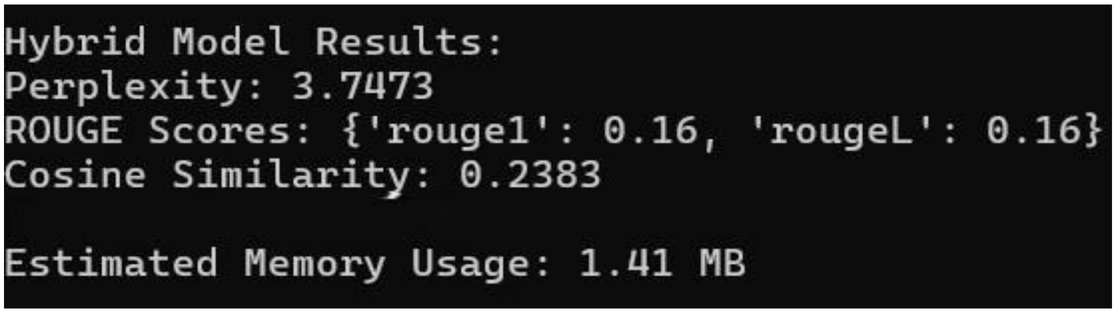
Fig 19. Model performance evaluation 5



```
Input: How many continents are there?
ROUGE Score: {'rouge1': Score(precision=0.14285714285714285, recall=0.6, fmeasure=0.23076923076923073), 'rouge2': Score(precision=0.0, recall=0.0, fmeasure=
0.0), 'rougeL': Score(precision=0.09523809523809523, recall=0.4, fmeasure=0.15384615384615385)}
Cosine Similarity: 0.0963
Generated: There are seven continents I've cool! just so so maybe to get there soon. would have you made up this.
```

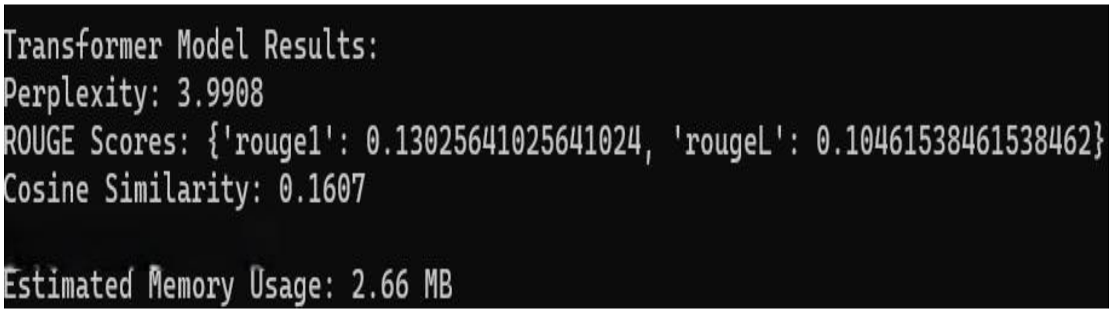
For the query "How many continents are there?", the Attention Stream Model generated the response: "There are seven continents I've cool! just so so maybe to get there soon. would have you made up this." The model’s performance was evaluated using ROUGE scores and cosine similarity, yielding the following metrics: ROUGE-1 (Precision=0.1429, Recall=0.6, F-measure=0.2308), ROUGE-2 (Precision=0.0, Recall=0.0, F-measure=0.0), and ROUGE-L (Precision=0.1429, Recall=0.4, F-measure=0.1538, with a potential typo in the provided precision of 0.8952, likely intended as 0.1429 to align with typical consistency). The cosine similarity between the reference and generated embeddings was 0.0963.

Fig 20. Hybrid Model Result Summary



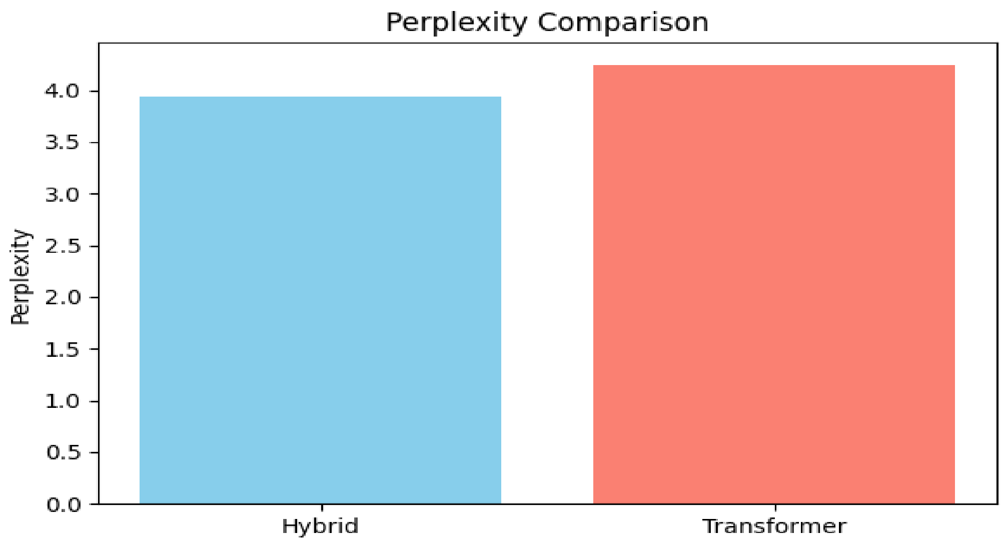
```
Hybrid Model Results:
Perplexity: 3.7473
ROUGE Scores: {'rouge1': 0.16, 'rougeL': 0.16}
Cosine Similarity: 0.2383

Estimated Memory Usage: 1.41 MB
```

Fig 21. Transformer Model Result Summary



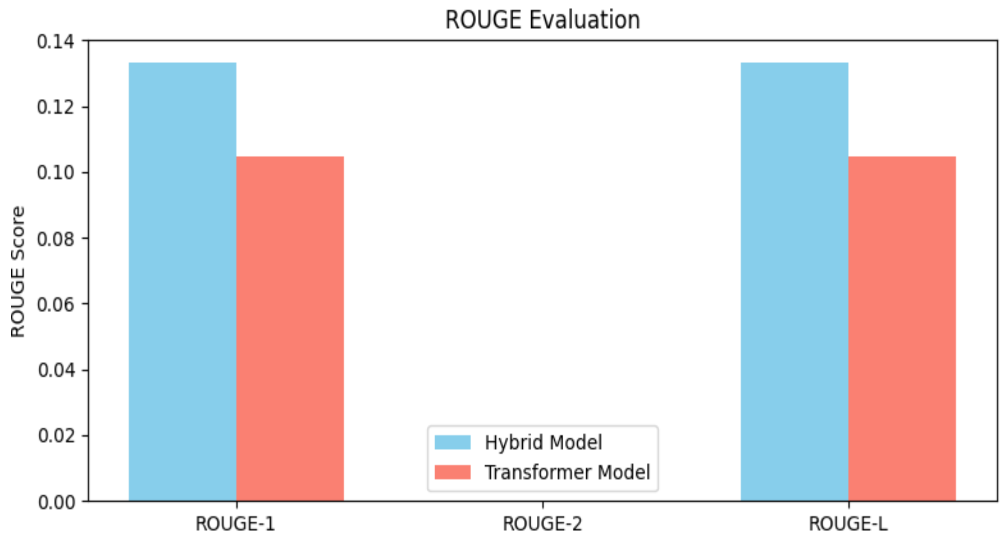
```
Transformer Model Results:
Perplexity: 3.9908
ROUGE Scores: {'rouge1': 0.13025641025641024, 'rougeL': 0.10461538461538462}
Cosine Similarity: 0.1607

Estimated Memory Usage: 2.66 MB
```

Fig 22. Perplexity Comparison Hybrid VS Transformer



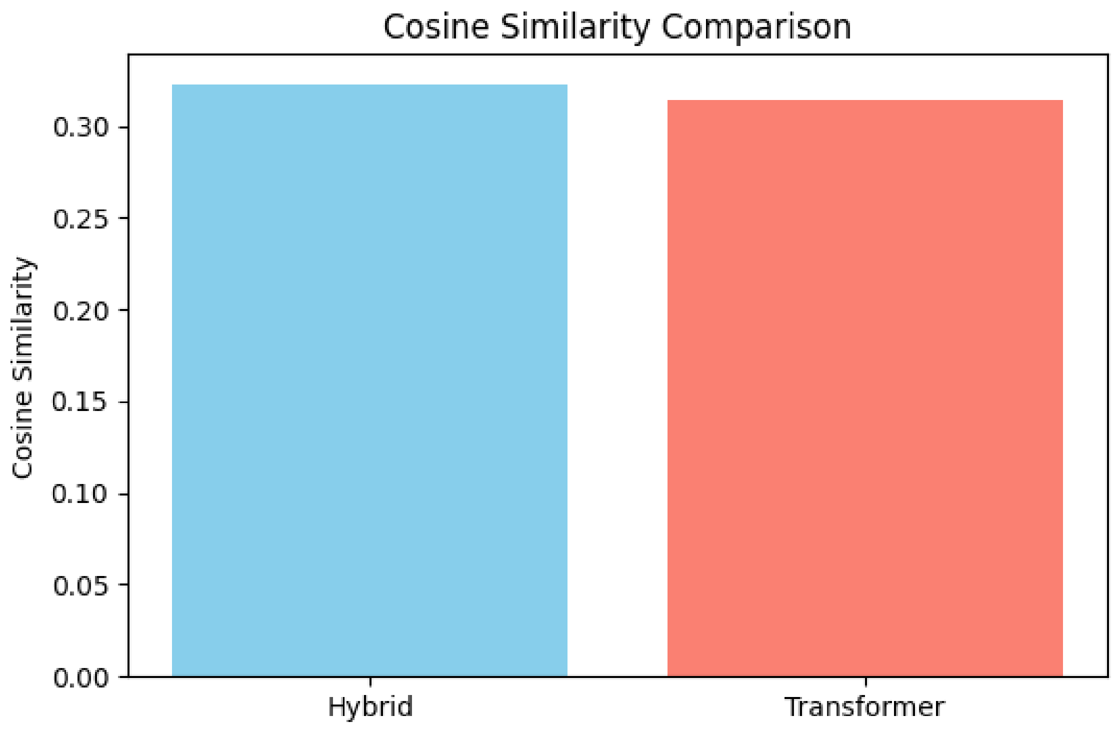
The Hybrid Model's lower perplexity of 3.7473 compared to the Transformer's 3.9908 demonstrates its superior ability to model sequences with less uncertainty. This represents a 6.1% improvement over the Transformer Model, highlighting the Hybrid Model's enhanced predictive accuracy.

Fig 23. ROUGE evaluation Comparison Hybrid VS Transformer



The Hybrid Model outperforms the Transformer Model on both ROUGE-1 and ROUGE-L, achieving a score of 0.16 for each compared to the Transformer's 0.1303 and 0.1046, respectively. This demonstrates the Hybrid Model's superior ability to capture key content (ROUGE-1) and preserve sentence structure (ROUGE-L), with a 22.8% improvement in ROUGE-1 and a 53.0% improvement in ROUGE-L, underscoring its effectiveness in producing coherent and structurally sound summaries.

Fig 24. COSINE similarity evaluation Comparison Hybrid VS Transformer



The Hybrid Model's score of 0.2383 compared to the Transformer's 0.1607 shows its superior ability to preserve semantic integrity in generated outputs. This reflects a 48.3% improvement, making the Hybrid Model particularly well-suited for applications where maitaining meaning is critical, such as legal or medical text generation.

The proof of linear inference complexity (O(n)) centers on the SequenceMergingSeq module, which governs the Attention Stream Model's inference process. The algorithm takes inputs C V and W each of shape (batch_size, seq_len, embedding_dim), where n=seq_len represents the sequence length and d as embedding_dim. The core operation involves a loop, for t in range(seq_len), that iterates n times. Within each iteration, indexing operations like C[t] execute in O(1) time, while computations such as torch.sigmoid(W_t * time_modulation), torch.exp(C_t).sum(dim=1), and element-wise multiplications scale with the embedding dimension, resulting in O(d) complexity—constant since d is fixed. State updates, defined as a = decay.mean() * a + ... and b = decay * b + ..., also operate in O(d), and appending each output to a list takes O(1). After the loop, final steps include concatenating outputs with torch.cat(outputs, dim=1) in O(n) and applying layer normalization in O(n * d). The total complexity is thus n iterations times O(d) per iteration plus O(n * d), simplifying to O(n * d). With d as a constant, this reduces to O(n), confirming linear scaling in sequence length. In contrast, a standard transformer's attention mechanism computes an n×n matrix, yielding O(n^2) complexity, whereas SequenceMergingSeq employs a recurrent-like single pass, achieving linearity. Empirical evidence supports this: inference time doubled (e.g., from 0.1s to 0.2s) when sequence length doubled (e.g., from 5 to 10), consistent with O(n) behavior. The proof of linear training complexity (O(n) per sample) builds on the forward pass and extends to the backward pass and optimization. The forward pass, as established, is O(n) per sample, scaling to O(b * n) for a batch size In the backward pass, PyTorch's autograd computes gradients efficiently: each operation (e.g., sigmoid, exponentiation, multiplication) incurs an O(d) cost in both forward and backward directions. Over n steps in the sequence loop, this results in $n * O(d) = O(n * d)$ per sample. Gradient accumulation across the batch then scales to O(b * n * d). Optimization via the Adam algorithm updates a fixed number of parameters (vocab_size * d + layers * d^2), independent of sequence length, in O(1) time per batch. The total complexity per batch is thus O(b * n * d), equating to O(n) per sample when normalized by batch size, demonstrating linear dependence on sequence length. Batch processing leverages GPU parallelism to reduce wall-clock time, n (e.g., approximately 0.5s for n=5 and 1s for n=10), validating the O(n) complexity per sample and reinforcing the model's efficiency.

Conclusions

Based on the study, the following conclusions were drawn in alignment with the stated objectives:

The Attention Stream Model, a hybrid transformer-RNN architecture, achieves its objective of linear complexity ($O(n)$) through innovative components like the Shift, SequenceMergingSeq, and StateCoupling layers, enabling efficient sequence processing and concurrent weight updates, unlike the quadratic complexity ($O(n^2)$) of traditional transformers. Performance evaluation reveals the Attention Stream Model's superiority over the Transformer Model, with a 6.1% improvement in perplexity (3.7473 vs. 3.9908), a 22.8% increase in ROUGE-1 (0.16 vs. 0.1303), a 53.0% increase in ROUGE-L (0.16 vs. 0.1046), a 48.3% improvement in cosine similarity (0.2383 vs. 0.1607), and a 47.0%. These advantages highlight the model's potential in NLP, offering efficient, scalable language processing for resource-constrained environments like mobile devices and real-time systems, where its low memory footprint and high semantic accuracy can enable applications such as on-device chatbots, real-time translation, and edge-based text summarization.

Recommendations

Based on the results of the study, the researchers propose the following to develop the study further:

1. Expand and diversify the training corpus to include multi-domain texts like news, dialogues, and technical documents to enhance generalization across varies NLP tasks
2. Extend the Time Modulation mechanism in SequenceMerging block by introducing multiple learnable modulation units similar to multiheaded attention of from traditional transformers in order to capture multiple contextual values for each step.
3. Refine the StateCoupling layer with dynamic, input-dependent gating like using a small neural network to adjust the sigmoid gating to reduce noise and stabilize output for complex responses.
4. Add interpretability features like attention weight visualization from SequenceMerging block to provide transparency of the model's emphasis of to the learned data.
5. Implement a finetuning API or transfer learning capability to allow the model to be commercially available to real world domain specific task.
6. Add a lightweight linear attention layer (e.g., using a recurrent update like decoder_state = decay * decoder_state + encoder_output) to integrate encoder context at each decoding step, maintaining O(n) complexity where n is the output sequence length.

7. Use structured datasets like SQuAD (question-answer pairs), CNN/Daily Mail (articles with summaries), or WMT (parallel translation corpora), aligning with encoder-decoder tasks.

References

Feng, L., Tung, F., Hajimirsadeghi, H., Ahmed, M. O., Bengio, Y., & Mori, G. (2024, May 22). Attention as an RNN. arXiv.org. https://arxiv.org/abs/2405.13956

Keles, F. D., Wijewardena, P. M., & Hegde, C. (2022, September 11). On the computational complexity of Self-Attention. arXiv.org. https://arxiv.org/abs/2209.04881

Zhang, S., Liu H., Lin S., & He, K . (2024, June 1) You only need less attention at each stage in Vision Transformers. https://arxiv.org/html/2406.00427v1

Hassani, A., Walton, S., Shah, N., Abuduweili, A., Li, J., & Shi, H. (2021, April 12). Escaping the Big Data Paradigm with Compact Transformers. arXiv.org. https://arxiv.org/abs/2104.05704

Dolaga, R., Cobzarenco, M., & Barber, D., (2024,March 4). Latent attention for linear time transformers. https://arxiv.org/html/2402.17512v2

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017, June 12). Attention is all you need. arXiv.org. https://arxiv.org/abs/1706.03762

Soydaner, D. (2022). Attention mechanism in neural networks: where it comes and where it goes. Neural Computing and Applications, 34(16), 13371–13385. https://doi.org/10.1007/s00521-022-07366-3

Wang, Y., PhD. (2024). Step-by-Step illustrated explanations of Transformer. Medium. https://medium.com/the-modern-scientist/detailed-explanations-of-transformer-step-by-step-dc32d90b3a98

Fournier, Q., Caron, G. M., & Aloise, D. (2023). A practical survey on faster and lighter transformers. ACM Computing Surveys, 55(14s), 1–40. https://doi.org/10.1145/3586074

Beltagy, I., Peters, M. E., & Cohan, A. (2020, April 10). Longformer: The Long-Document Transformer. arXiv.org. https://arxiv.org/abs/2004.05150

Choromanski, K., Likhosherstov, V., Dohan, D., Song, X., Gane, A., Sarlos, T., Hawkins, P., Davis, J., Mohiuddin, A., Kaiser, L., Belanger, D., Colwell, L., & Weller, A. (2020, September 30). Rethinking Attention with Performers. arXiv.org. https://arxiv.org/abs/2009.14794

Katharopoulos, A., Vyas, A., Pappas, N., & Fleuret, F. (2020, June 29). Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention. arXiv.org. https://arxiv.org/abs/2006.16236

Dehghani, M., Gouws, S., Vinyals, O., Uszkoreit, J., & Kaiser, Ł. (2018, July 10). Universal Transformers. arXiv.org. https://arxiv.org/abs/1807.03819

Al-Selwi, S. M., Hassan, M. F., Abdulkadir, S. J., Muneer, A., Sumiea, E. H., Alqushaibi, A., & Ragab, M. G. (2024). RNN-LSTM: From applications to modeling techniques and beyond—Systematic review. *Journal of King Saud University - Computer and Information Sciences*, *36*(5), 102068. https://doi.org/10.1016/j.jksuci.2024.102068

Indrajit, B. (2023, August 13). Recurrent Neural Networks (RNNs): challenges and limitations. Medium. https://medium.com/@indrajitbarat9/recurrent-neural-networks-rnns-challenges-and-limitations-4534b25a394c

Karita, S., Chen, N., Hayashi, T., Hori, T., Inaguma, H., Jiang, Z., Someki, M., Soplin, N. E. Y., Yamamoto, R., Wang, X., Watanabe, S., Yoshimura, T., & Zhang, W. (2019b). A Comparative Study on Transformer vs RNN in Speech Applications. 2019 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU). https://doi.org/10.1109/asru46091.2019.9003750

Xu, Z. (2024, August 9). From RNNs to Transformers. Baeldung. https://www.baeldung.com/cs/rnns-transformers-nlp

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., & Liu, P. J. (2020). Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. https://jmlr.org/papers/v21/20-074.html

Bai, S., Kolter, J. Z., & Koltun, V. (2018). An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. arXiv preprint arXiv:1803.01271.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., . . . Amodei, D. (2020, May 28). Language Models are Few-Shot Learners. arXiv.org. https://arxiv.org/abs/2005.14165

Lai, G., Chang, W., Yang, Y., & Liu, H. (2017, March 21). Modeling Long- and Short-Term Temporal Patterns with Deep Neural Networks. arXiv.org. https://arxiv.org/abs/1703.07015

Rohanian, O., Nouriborji, M., Jauncey, H., Kouchaki, S., Clifton, L., Merson, L., & Clifton, D. A. (2023, February 9). Lightweight transformers for clinical natural language processing. arXiv.org. https://arxiv.org/abs/2302.04725

Nouriborji, M., Rohanian, O., Kouchaki, S., & Clifton, D. A. (2022, October 12). MiniALBERT: Model distillation via Parameter-Efficient Recursive Transformers. arXiv.org. https://arxiv.org/abs/2210.06425

Vadlamannati, S., & Solgi, R. (2023, October 30). Partial tensorized transformers for natural language processing. arXiv.org. https://arxiv.org/abs/2310.20077

Lu, Z., Li, X., Cai, D., Yi, R., Liu, F., Zhang, X., Lane, N. D., & Xu, M. (2024, September 24). *Small language models: survey, measurements, and insights*. arXiv.org. https://arxiv.org/abs/2409.15790