HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

PROJECT REPORT
Text Summerization

Course: Introduction to Artificial Intelligence

Supervisor: Assoc. Prof. Le Thanh Huong

Authors:	**Student ID:**
Nguyen Binh Anh	20235470
Chu Xuan Minh	20235527
Vu Hai An	20235468
Nguyen Tien Phat	20235544
Nguyễn Trung Hai	20235495

HA NOI, 11 - 2024

Table of Contents

ABSTRACT

This report examines the use of transformer-based models for abstractive text summarization, comparing their effectiveness and efficiency to traditional methods. The study focuses on a custom-built transformer model developed from scratch for summarization tasks, and compares its performance with fine-tuned transformer models and state-of-the-art (SOTA) pre-trained models. The research investigates how well the scratch-built model generates coherent, concise summaries and evaluates it in terms of summary quality, fluency, and content preservation. The findings highlight the trade-offs between training models from the ground up versus using pre-trained, fine-tuned transformers, offering insights into the best approaches for real-world text summarization challenges.

1 Introduction

Text summarization is a fundamental task in natural language processing (NLP) that aims to generate a concise representation of a longer text while retaining its essential meaning and key information. Abstractive summarization, unlike extractive methods, generates summaries by paraphrasing and rephrasing the original content rather than selecting fragments directly from the source text. This makes abstractive summarization a more challenging but valuable approach, as it requires the model to understand the underlying meaning and produce fluent, coherent summaries.

In recent years, transformer-based models have revolutionized NLP tasks, particularly with architectures such as the Transformer, BERT, and GPT. Among these, the T5 (Text-to-Text Transfer Transformer) model has gained significant attention for its versatility and effectiveness in various NLP tasks, including summarization. T5 treats all NLP tasks as text-to-text problems, allowing for a unified framework that can be adapted to a range of applications, including abstractive summarization.

This report focuses on the development and evaluation of an abstractive summarization model trained from scratch. We utilize the XSum dataset, which contains news articles paired with single-sentence summaries, making it an ideal resource for training and testing abstractive summarization models. To process the text data, we employ the T5 tokenizer, which efficiently converts raw text into a format suitable for transformer models. Additionally, we incorporate GloVe (Global Vectors for Word Representation) embeddings to improve the model's understanding of word semantics and relationships, enhancing its ability to generate coherent and contextually accurate summaries.

Our approach is evaluated against two key benchmarks: a fine-tuned version of a pre-trained transformer model and state-of-the-art (SOTA) abstractive summarization models. By comparing our model with these alternatives, we aim to assess the trade-offs between building a model from scratch and leveraging pre-trained, fine-tuned architectures, particularly in terms of summary quality, fluency, and relevance. Through this comparison, we seek to provide valuable insights into the effectiveness of transformer-based models for abstractive summarization tasks, highlighting the potential and limitations of different approaches in real-world applications.

2 Methodology

Our pipeline for the text summarization system is demonstrated in the figure below.

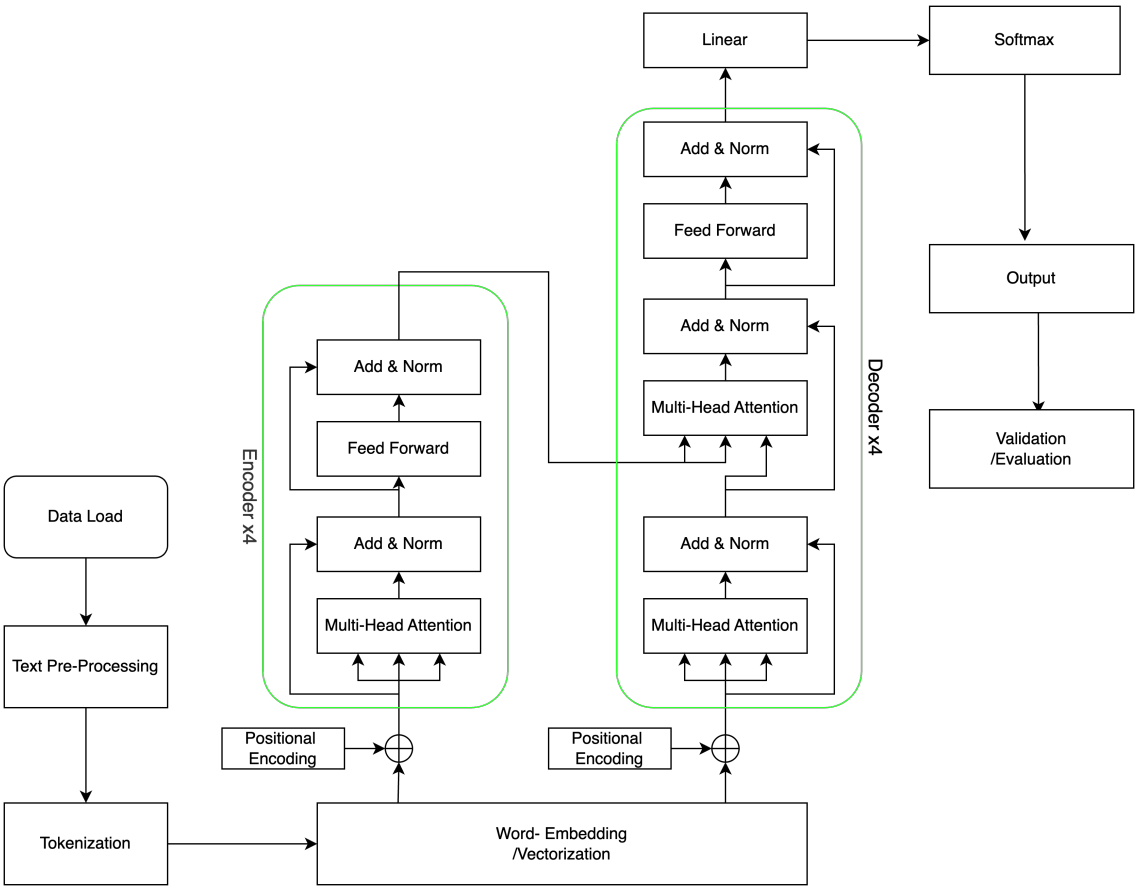


Figure 1: The pipeline of our text summarization system.

We will describe in detail three main parts: Preprocessing, Vector Representation, Transformer Architecture and Training Components.

2.1 Preprocessing

2.1.1 Dataset

2.1.2 Data cleaning

Data cleaning is a crucial step in preparing the dataset, ensuring that the input articles and highlights are consistent and usable for the model. Below is the following step for data cleaning:

- **Text Lowercasing**: All characters in the articles and highlights were converted to lowercase. Due to limited resources, this standardization will reduce the vocabulary size and ensure uniformity across the dataset, even though this might lead to a slight loss of meaning. This standardization helps the model focus on semantic meaning rather than being distracted by case variations.
- **Special Character Removal**: Non-alphanumeric characters, except for essential punctuation marks (.,!? % $ & @ # :) were removed from text. This step eliminates irrelevant symbols that do not contribute to the summarization task while retaining punctuation necessary for sentence structure.

- **Whitespace Normalization**: Extra whitespaces, tabs, and newline characters were removed, and consecutive spaces were replaced with a single space. This ensures consistent formatting and avoids unnecessary token padding during tokenization.

2.1.3 Tokenization

In text summarization, tokenization—the process of breaking a text or sentence into smaller units called tokens—plays a vital role in understanding the content and structure of the text, making it easier to analyze and process. These tokens can be words, subwords, or characters. In this project, we use the T5Tokenizer, which employs a subword-based tokenization approach. With a maximum sequence length of 512 tokens, this tokenizer is well-suited for handling long text inputs, ensuring efficient and accurate preparation for the summarization model.

2.2 Vector Representation

To enable computation in machine learning models, raw text data must be transformed into numerical representations. In this project, text vectorization is achieved through two key steps: constructing Embedding and Positional Encoding.

2.2.1 Text Embedding

In this project, we utilize two methods for text embedding to represent words as dense numerical vectors in a continuous vector space, enabling the model to capture semantic and syntactic relationships between words:

1. **GloVe Embeddings:** We leverage pre-trained GloVe (Global Vectors for Word Representation) embeddings, which were trained on large corpora such as Wikipedia and Common Crawl. These 300-dimensional vectors already encapsulate meaningful word relationships, providing the model with prior linguistic knowledge without requiring additional training on word semantics. GloVe embeddings are used to map each word in the text to a fixed-length 300-dimensional vector, enabling efficient context understanding.

2. **Randomly Initialized 512-Dimensional Vectors:** As an alternative, we also experiment with randomly initialized vectors of 512 dimensions. The higher dimensionality is intended to provide greater capacity for capturing nuanced relationships and context in the text. Although these vectors are not pre-trained, their increased dimensionality offers the potential to learn more complex features during the model training process.

By combining these approaches, we aim to evaluate the trade-off between pre-trained linguistic knowledge and the flexibility offered by higher-dimensional embeddings initialized from scratch.

2.2.2 Positional Encoding

A key aspect of processing text is accounting for the order of words in the input sequence. To address this, a positional encoding vector is added to each input embedding. In our model, we use sinusoidal encoding for this purpose. These vectors follow a specific, predefined pattern that helps the model determine the position of each word and the relative distance between words in the sequence. The intuition behind this approach is that by adding these positional values to the word embeddings, the model gains meaningful information about the order of the words, enhancing its ability to understand the structure of the text. The positional encoding is calculated as:

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

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

Where:

- pos is the position of the word in the sequence.
- i is the index of the dimension in the embedding.
- d_{model} is the embedding size.

If we assumed the embedding has a dimensionality of 4 and a sentence of 3 words, the actual positional encodings would look like this:

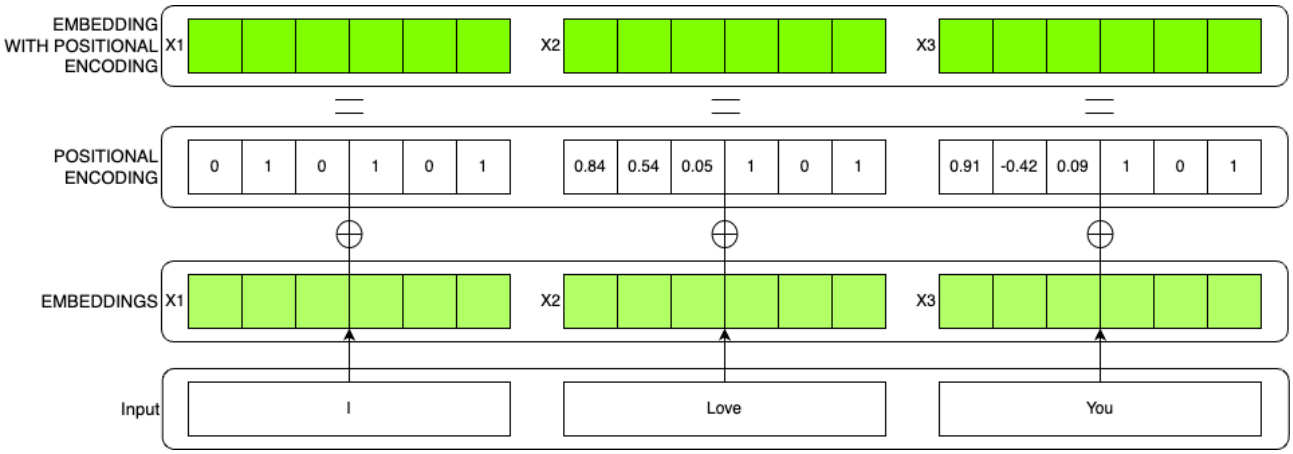


Figure 2: Enter Caption

2.3 Transformer Architecture

Most competitive neural sequence transduction models have an encoder-decoder structure. Here, the encoder maps an input sequence of symbol representations $(x_1,...,x_n)$ to a sequence of continuous representations z = $(z_1,...,z_n)$. Given z, the decoder then generates an output sequence $(y_1, ...,y_m)$ of symbols one element at a time. This process is inherently auto-regressive: at each step, the decoder relies on the symbols it has already generated as additional context to predict the next symbol. Let's delve deeper into the roles of the encoder and decoder and explore how they achieve outstanding performance in sequence transduction tasks like text summarization.

2.3.1 Self-Attention

Consider the sentence:

"The student did not finish the test because it was too difficult."

Here, the word "it" raises a question: does it refer to "test" or "student"? For humans, the answer is obvious—"it" refers to the "test." However, for a machine, determining this requires understanding the context of the sentence.

This is where self-attention comes into play. Self-attention allows the model to examine all the other words in the input sequence to gather clues about the relationships between them. By evaluating which words have stronger or weaker connections, the model can better understand the role of "it" in this context. This mechanism enables the model to assign greater weight

to "test" when encoding the meaning of "it," leading to a more accurate understanding of the sentence. Self-attention enables models to capture intricate relationships between words, significantly enhancing their ability to understand and process natural language.

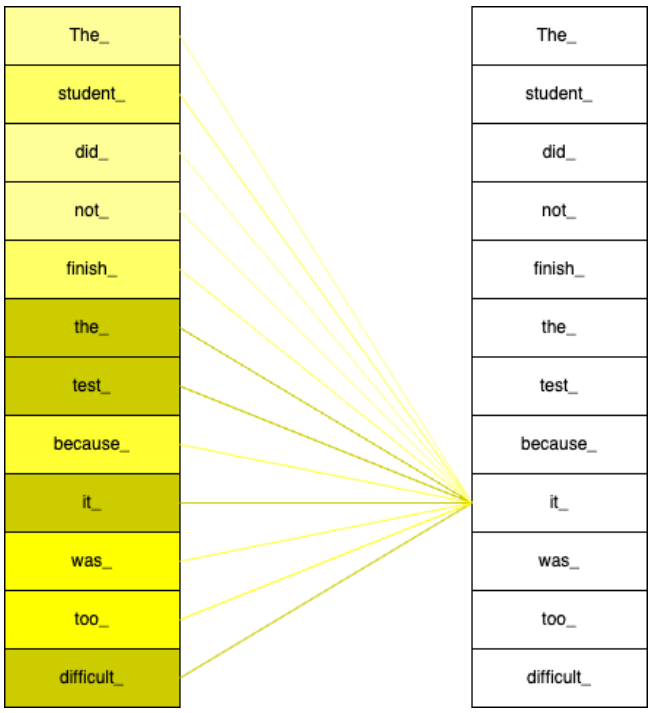


Figure 3: The word "it" is more closely related to "test" than to "student." Furthermore, "it" has a strong connection to the attribute "difficult," which describes the test. As a result, the attention weight for "test" and "difficult" is higher compared to "student"

Attention is a mechanism that takes a query and softly retrieves relevant information from a key-value store by focusing on the values associated with the keys most similar to the query. This "retrieval" process involves calculating a weighted average of the values and assigning higher weights to those linked to keys that closely match the query. In self-attention, the same set of elements serves as the source for the queries, keys, and values, allowing the model to assess word relationships within the same context dynamically.

Consider a token $\mathbf{x}_i$ in the sequence $\mathbf{x}_{1:n}$. From it, we define a query $\mathbf{q}_i = Q\mathbf{x}_i$, for matrix $Q \in \mathbb{R}^{d\times d}$. Then, for each token in the sequence $\mathbf{x}_j \in \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, we define both a key and a value similarly, with two other weight matrices: $\mathbf{k}_j = K\mathbf{x}_j$, and $\mathbf{v}_j = V\mathbf{x}_j$ for $K, V \in \mathbb{R}^{d\times d}$.

Our contextual representation $\mathbf{h}_i$ of $\mathbf{x}_i$ is a linear combination (that is, a weighted sum) of the values of the sequence,

$$\mathbf{h}_i = \sum_{j=1}^{n} \alpha_{ij}\mathbf{v}_j,$$

where the weights, α_{ij}, control the strength of contribution of each $\mathbf{v}_j$. Going back to our key-value store analogy, the α_{ij} softly selects what data to look up. We define these weights by computing the affinities between the keys and the query, $\mathbf{q}_i^\top \mathbf{k}_j$, and then computing the softmax over the sequence:

$$\alpha_{ij} = \frac{\exp(\mathbf{q}_i^\top \mathbf{k}_j)}{\sum_{j'=1}^{n} \exp(\mathbf{q}_i^\top \mathbf{k}_{j'})}.$$

Intuitively, what we've done by this operation is take our element $\mathbf{x}_i$ and look in its own sequence $\mathbf{x}_{1:n}$ to figure out what information (in an informal sense), from what other tokens, should be used in representing $\mathbf{x}_i$ in context. The use of matrices K, Q, V intuitively allows us

to use different views of the $\mathbf{x}_i$ for the different roles of key, query, and value. We perform this operation to build $\mathbf{h}_i$ for all $i \in \{1, \ldots, n\}$.

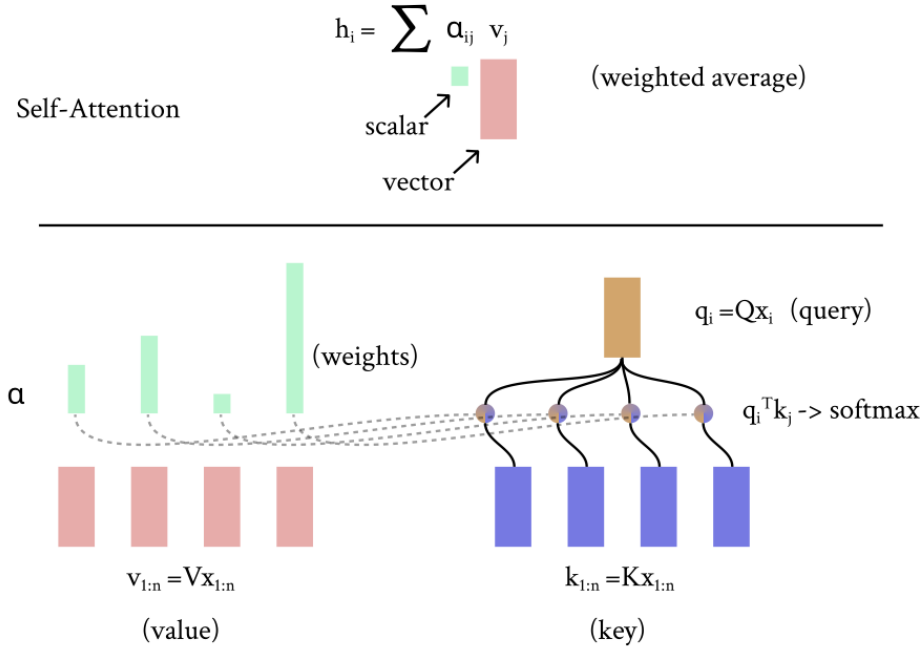


Figure 4: Minimal self-attention architecture

2.3.2 Attention logit scaling

Consider the dot products $\mathbf{q}_i^\top \mathbf{k}_j$. The intuition behind scaling is that, as the dimensionality d of the vectors being dotted grows large, the magnitude of the dot product of even random vectors (e.g., at initialization) grows roughly as $\sqrt{d}$. To counteract this effect, the dot products are normalized by dividing by $\sqrt{d}$. Formally, the scaled dot-product attention is expressed as:

$$\boldsymbol{\alpha} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right) \in \mathbb{R}^{n \times n}.$$

This normalization ensures that the attention mechanism remains stable during training by preventing excessively large values from dominating the softmax operation.

2.3.3 Multi-Head-Attention

Intuitively, a single call of self-attention is best at picking out a single value (on average) from the input value set. It does so softly, by averaging over all of the values, but it requires a balancing game in the key-query dot products in order to carefully average two or more things. In What we'll present now, *multi-head self-attention*, intuitively applies self-attention multiple times at once, each with different key, query, and value transformations of the same input, and then combines the outputs.

For an integer number of heads k, we define matrices $K^{(\ell)}, Q^{(\ell)}, V^{(\ell)} \in \mathbb{R}^{d \times d/k}$ for $\ell \in \{1, \ldots, k\}$. (We'll see why we have the dimensionality reduction to d/k soon.) These are our key, query, and value matrices for each head. Correspondingly, we get keys, queries, and values $k_{1:n}^{(\ell)}, q_{1:n}^{(\ell)}, v_{1:n}^{(\ell)}$ as in single-head self-attention. We then perform self-attention with each head:

$$h_i^{(\ell)} = \sum_{j=1}^{n} \alpha_{ij}^{(\ell)} v_j^{(\ell)}$$

$$\alpha_{ij}^{(\ell)} = \frac{\exp\left(q_i^{(\ell)} \cdot k_j^{(\ell)}\right)}{\sum_{j'=1}^{n} \exp\left(q_i^{(\ell)} \cdot k_{j'}^{(\ell)}\right)} \tag{23}$$

Note that the output $h_i^{(\ell)}$ of each head is in reduced dimension d/k. Finally, we define the output of multi-head self-attention as a linear transformation of the concatenation of the head outputs, letting $O \in \mathbb{R}^{d \times d}$:

$$h_i = O\left[v_i^{(1)}, \ldots, v_i^{(k)}\right] \tag{24}$$

where we concatenate the head outputs, each of dimensionality $d \times d/k$, at their second axis, such that their concatenation has dimension $d \times d$.

Sequence-tensor form. To understand why we have the reduced dimension of each head output, it's instructive to get a bit closer to how multi-head self-attention is implemented in code. In practice, multi-head self-attention is no more expensive than single-head due to the low-rankness of the transformations we apply.

For a single head, recall that $x_{1:n}$ is a matrix in $\mathbb{R}^{n \times d}$. Then we can compute our value vectors as a matrix as $x_{1:n}V$, and likewise our keys and queries $x_{1:n}K$ and $x_{1:n}Q$, all matrices in $\mathbb{R}^{n \times d}$. To compute self-attention, we can compute our weights in matrix operations:

$$\alpha = \text{softmax}\left(x_{1:n}QK^\top x_{1:n}^\top\right) \in \mathbb{R}^{n \times n} \tag{25}$$

and then compute the self-attention operation for all $x_{1:n}$ via:

$$h_{1:n} = \text{softmax}\left(x_{1:n}QK^\top x_{1:n}^\top\right) x_{1:n}V \in \mathbb{R}^{n \times d} \tag{26}$$

2.3.4 Feedforward Layer

The feedforward layer is a critical component of the transformer model, designed to address the limitations of the self-attention mechanism. Self-attention alone lacks element-wise non-linearities, resulting in the simple re-averaging of value vectors. This restricts the model's ability to learn complex relationships. The feedforward layer introduces non-linear activation and additional transformations, enabling the model to capture richer and more expressive features.

- **Linear Transformation:**
  The input tensor is passed through a linear layer that applies a linear transformation:

  $$\mathbf{x}' = \mathbf{x}\mathbf{W} + \mathbf{b}$$

  Here, $\mathbf{W}$ and $\mathbf{b}$ are learnable parameters. This operation projects the input tensor from its original size (e.g., 256) to a higher dimensional space (e.g., 512), allowing the model to process more complex patterns.

- **ReLU Activation Function:**
  After the linear transformation, the tensor is passed through a *Rectified Linear Unit (ReLU)* activation function, defined as:

  $$\text{ReLU}(x) = \max(0, x)$$

  This operation introduces non-linearity into the model by zeroing out negative values and leaving positive values unchanged. ReLU enables the model to learn more complex patterns by allowing certain features to activate only when they contribute positively to the representation.

- **Dropout for Regularization:**
  The output of the ReLU activation function is passed through a dropout layer. Dropout randomly sets a fraction of the tensor elements to zero during training. This process simulates "dropping out" some weights, forcing the model to focus on other, stronger features to compensate. By doing so, the model avoids over-relying on specific features, encouraging it to learn a more robust and general representation that performs well on unseen data.

2.3.5 Residual Connections

Residual connections are a fundamental component of Transformer architectures, facilitating the stable and efficient training of deep neural networks. These connections directly combine the input of a layer with its output, addressing challenges such as vanishing gradients and improving the optimization process.

Mathematically, a residual connection is defined as:

$$f_{\text{residual}}(\mathbf{h}) = f(\mathbf{h}) + \mathbf{h}, \tag{29}$$

where:

- $\mathbf{h}$: Input hidden states provided to the layer.
- $f(\mathbf{h})$: The transformation applied to $\mathbf{h}$ by a sub-layer, such as self-attention or feedforward operations.

Purpose and Benefits

- **Gradient Flow:** Residual connections allow gradients to propagate through the network *without interruption* during backpropagation. Because the identity function contributes a gradient of 1, the overall gradient is preserved, reducing the risk of vanishing gradients in deeper networks.
- **Simplified Learning:** By introducing residuals, the network learns the difference between the transformation $f(\mathbf{h})$ and the identity mapping. This simplifies optimization and reduces the complexity of the learning task for each layer.
- **Stability in Training:** Residual connections enhance the stability of training deep networks, making it feasible to stack numerous layers—an essential feature of Transformer architectures.

Add & Norm Block : In Transformers, residual connections are used in conjunction with *Layer Normalization* to form an *Add & Norm* block. This block ensures smoother training dynamics and consistent activation scaling. The *post-normalization* approach is employed, where Layer Normalization (LN) is applied *after* the residual connection:

$$\mathbf{h}_{\text{output}} = \text{LN}(f(\mathbf{h}) + \mathbf{h}). \tag{30}$$

2.3.6 Layer Norm

One important learning aid in Transformers is *layer normalization*. The intuition of layer norm is to reduce uninformative variation in the activations at a layer, providing a more stable input to the next layer. Further work shows that this may be most useful not in normalizing the forward pass, but actually in improving gradients in the backward pass.

To do this, layer norm computes statistics across the activations at a layer to estimate the mean and variance of the activations, and normalizes the activations with respect to those estimates, while optionally learning (as parameters) an elementwise additive bias and multiplicative gain by which to sort of de-normalize the activations in a predictable way. The third part seems not to be crucial, and may even be harmful, so we omit it in our presentation.

One question to ask when understanding how layer norm affects a network is, *"computing statistics over **what?**"* That is, what constitutes a layer? In Transformers, the answer is always that statistics are computed independently for a single index into the sequence length (and a single example in the batch) and shared across the d hidden dimensions. Put another way, the statistics for the token at index i won't affect the token at index $j \neq i$.

So, we compute the statistics for a single index $i \in \{1, \ldots, n\}$ as

$$\hat{\mu}_i = \frac{1}{d}\sum_{j=1}^{d} h_{ij}, \quad \hat{\sigma}_i = \sqrt{\frac{1}{d}\sum_{j=1}^{d}(h_{ij} - \hat{\mu}_i)^2},$$

where (as a reminder), $\hat{\mu}_i$ and $\hat{\sigma}_i$ are scalars, and we compute the layer norm as

$$\mathrm{LN}(h_i) = \frac{h_i - \hat{\mu}_i}{\hat{\sigma}_i},$$

where we've broadcasted the $\hat{\mu}_i$ and $\hat{\sigma}_i$ across the d dimensions of h_i. Layer normalization is a great tool to have in your deep learning toolbox more generally.

2.3.7 Future Masking

Future masking is a mechanism used to ensure causality in sequence-based models by preventing each token in a sequence from attending to future tokens during the attention computation. This guarantees that the model processes the input in a left-to-right manner, crucial for tasks like autoregressive language modeling or text generation.

Future masking is implemented using a binary mask matrix that identifies future tokens in the sequence. The mask is an $n \times n$ matrix (where n is the sequence length) with values:

$$M[i,j] = \begin{cases} 1 & \text{if } j > i \text{ (future token)}, \\ 0 & \text{otherwise.} \end{cases}$$

Instead of adding the mask to the attention scores, the attention weights for future tokens are directly set to $-\infty$, ensuring that they are ignored during the softmax computation.

Future masking ensures that each token only attends to itself and preceding tokens in the sequence, preserving the causal structure of the data. For example, when processing the sequence *"Henry likes eating apple,"* the token *"likes"* will only attend to *"Henry"* and itself, ignoring *"eating"* and *"apple."*

The raw attention scores are computed as the dot product of Query (Q) and Key (K) vectors:

$$\text{Scores} = \frac{QK^\top}{\sqrt{d_k}}$$

The mask is applied by replacing scores corresponding to future tokens with $-\infty$, which results in zero probability for those tokens after the softmax step:

$$\text{Scores}[i,j] = \begin{cases} -\infty & \text{if } M[i,j] = 1, \\ \text{Scores}[i,j] & \text{otherwise.} \end{cases}$$

After applying the mask, the modified attention scores are passed through the softmax function:

$$\text{Attention Weights}[i, j] = \text{softmax}(\text{Scores}[i, j])$$

The $-\infty$ scores for future tokens are effectively ignored, as they result in zero probability after the softmax step. This ensures that each token only attends to itself and preceding tokens.

The attention weights are then used to compute a weighted sum of the Value (V) matrix, aggregating information only from allowable tokens. This operation ensures that each token's final representation reflects the context from itself and all preceding tokens in the sequence.

Consider the input sequence *"Henry likes eating apple."* The corresponding binary mask matrix M for this sequence is:

$$M = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Applying this mask to the attention scores ensures that:

- *"Henry"* only attends to itself.
- *"likes"* attends to both *"Henry"* and itself, ignoring *"eating"* and *"apple."*
- *"eating"* attends to *"Henry,"* *"likes,"* and itself, ignoring *"apple."*
- *"apple"* attends to all tokens.

This approach guarantees causality in the model, ensuring that the left-to-right processing structure required for autoregressive tasks is preserved.

2.3.8 Cross-Attention.

Cross-attention allows the decoder to attend to different parts of the input document, determining how sentences or phrases in the input sequence (e.g., "The company reported significant growth this year, driven by increased sales and new product launches") relate to phrases in the output summary (e.g., "Company sees strong growth due to sales and new products"). Cross-attention uses one sequence to define the keys and values of self-attention, and another sequence to define the queries. You might think, hey wait, isn't that just what attention always was before we got into this self-attention business? Yeah, pretty much. So if

$$\mathbf{h}^{(x)} = \text{TransformerEncoder}(\mathbf{w}),$$

and we have some intermediate representation $\mathbf{h}^{(y)}$ of sequence $\mathbf{y}$, then we let the queries come from the decoder (the $\mathbf{h}^{(y)}$ sequence) while the keys and values come from the encoder:

$$\mathbf{q}_i = \mathbf{Q}\mathbf{h}_i^{(y)}, \quad i \in \{1, \ldots, m\},$$

$$\mathbf{k}_j = \mathbf{K}\mathbf{h}_j^{(x)}, \quad j \in \{1, \ldots, n\},$$

$$\mathbf{v}_j = \mathbf{V}\mathbf{h}_j^{(x)}, \quad j \in \{1, \ldots, n\},$$

and compute the attention on $\mathbf{q}, \mathbf{k}, \mathbf{v}$ as we defined for self-attention.

2.3.9 The Final Linear and Softmax Layer

The decoder stack outputs a vector of floats. How do we turn that into a word? That's the job of the final Linear layer which is followed by a Softmax Layer.

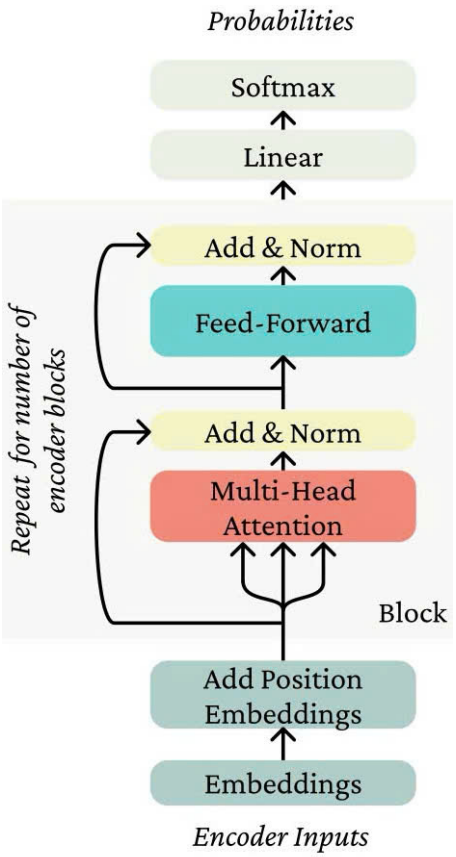
The Linear layer is a simple fully connected neural network that projects the vector produced by the stack of decoders, into a much, much larger vector called a logits vector.

Let's assume that our model knows 10,000 unique English words (our model's "output vocabulary") that it's learned from its training dataset. This would make the logits vector 10,000 cells wide – each cell corresponding to the score of a unique word. That is how we interpret the output of the model followed by the Linear layer.

The softmax layer then turns those scores into probabilities (all positive, all add up to 1.0). The cell with the highest probability is chosen, and the word associated with it is produced as the output for this time step.

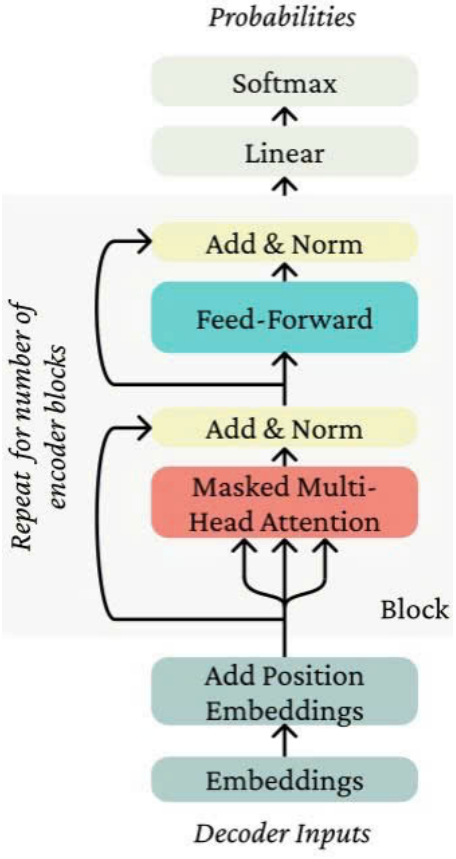
2.3.10 Transformer Encoder

A Transformer Encoder takes a single sequence $\mathbf{w}_{1:n}$ and performs no future masking. It embeds the sequence with E to make $\mathbf{x}_{1:n}$, adds the position representation, and then applies a stack of independently parameterized *Encoder Blocks*, each of which consists of (1) multi-head attention and Add & Norm, and (2) feed-forward and Add & Norm. So, the output of each Block is the input to the next. *Uses of the Transformer Encoder* A Transformer Encoder is great in contexts where you aren't trying to generate text autoregressively (there's no masking in the encoder so each position index can see the whole sequence) and want strong representations for the whole sequence (again, possible because even the first token can see the whole future of the sequence when building its representation).



Transformer Encoder

Figure 5



Transformer Decoder

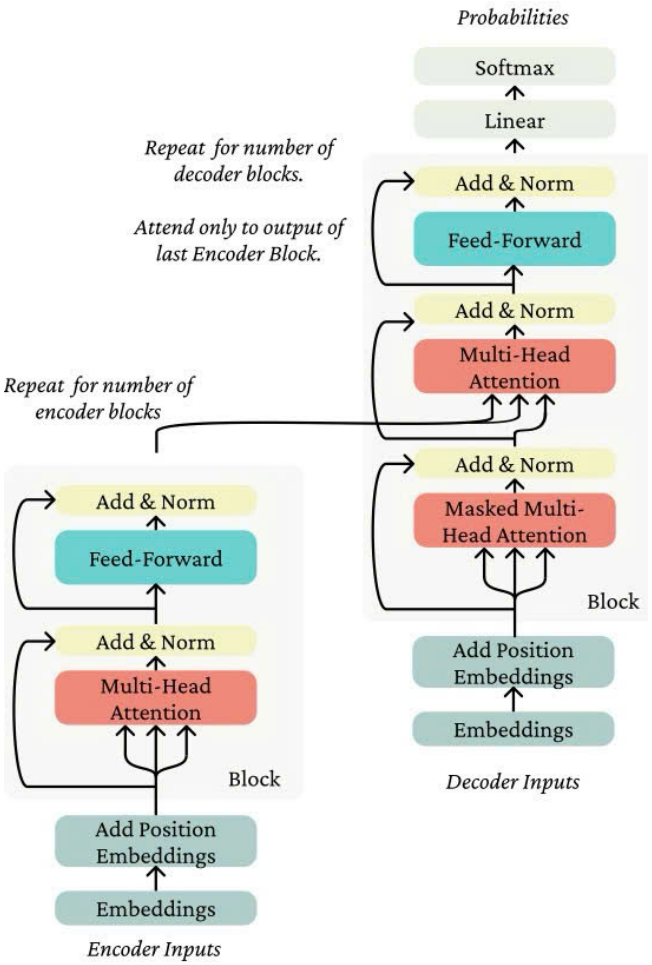
Figure 6

2.3.11 Transformer Decoder

These differ from Transformer Encoders simply by using future masking at each application of self-attention. This ensures that the informational constraint (no cheating by looking at the future!) holds throughout the architecture.

2.3.12 Transformer Encoder-Decoder

A Transformer encoder-decoder takes as input two sequences. Figure 6 shows the whole encoder-decoder structure. The first sequence $x_{1:n}$ is passed through a Transformer Encoder to build contextual representations. The second sequence $y_{1:m}$ is encoded through a modified Transformer Decoder architecture in which cross-attention is applied from the encoded representation of $y_{1:m}$ to the output of the Encoder.



Transformer Encoder-Decoder

Figure 7

3 Experimentals

3.1 Experimental Setup

3.1.1 Dataset and Framework

The dataset used in this project is the Extreme Summarization (XSum) dataset, which is specifically designed for evaluating abstractive single-document summarization systems. The primary goal of using XSum is to generate a concise, one-sentence summary that effectively answers the question, "What is the article about?"

The XSum dataset comprises 226,711 BBC news articles published between 2010 and 2017, each paired with a single-sentence abstractive summary. These articles span a wide range of domains, including News, Politics, Sports, Weather, Business, Technology, Science, Health, Family, Education, Entertainment, and Arts. This diversity makes XSum a robust and challenging benchmark for text summarization tasks.

For this project, we use the official random splits of the dataset:

- **Training set:** 204,045 articles (90%)
- **Validation set:** 11,332 articles (5%)
- **Test set:** 11,334 articles (5%)

The XSum dataset was chosen for this project due to its suitability for training models with computational limitations. With single-sentence summaries and relatively concise input articles, XSum offers a balance between task complexity and efficiency, allowing for faster training times. This makes it an ideal choice for models that require a dataset that is both challenging and computationally feasible, ensuring effective training within resource constraints.

Framework: The experiments were conducted using the PyTorch deep learning framework. We chose PyTorch for its flexibility, ease of use, and strong community support, making it well-suited for implementing and training complex Transformer-based models.

3.1.2 Evaluation metrics

ROUGE-N Impact on Content Coverage:

- **ROUGE-1 (Unigram):** Measures the overlap of individual words between the generated summary and the reference. A higher ROUGE-1 score indicates that the model is effectively capturing the essential keywords and main topics of the source text.
- **ROUGE-2 (Bigram):** Assesses the overlap of two-word sequences. Improved ROUGE-2 scores suggest that the model maintains the contextual relationships between words, enhancing the coherence and relevance of the summary.

Model Evaluation:

- High ROUGE-N scores generally reflect that the model is good at retaining important information.
- However, solely relying on ROUGE-N might overlook the structural and grammatical quality of the summary.

ROUGE-L Impact on Structural Similarity:

- **Longest Common Subsequence (LCS):** ROUGE-L evaluates the longest sequence of words that appear in both the generated and reference summaries in the same order. This metric emphasizes the preservation of the logical flow and sentence structure.

Model Evaluation:

- A higher ROUGE-L score indicates that the summarizer maintains the sequence of key information, leading to more logically coherent and structured summaries.
- It helps ensure that the summary follows a natural progression similar to human-written summaries.

Enhancing Summary Coherence:

- By focusing on the longest common subsequence, ROUGE-L encourages the model to produce summaries that are not only factually accurate but also narratively coherent, enhancing readability and comprehension.

ROUGE-S Impact on Semantic Similarity:

- **Skip-Bigrams:** ROUGE-S measures the overlap of word pairs allowing for gaps between them. This metric captures the semantic relationships and broader contextual information beyond adjacent word pairs.

Model Evaluation:

- A higher ROUGE-S score indicates that the model effectively captures the underlying semantics and meaning of the source text, even when words are not directly adjacent.
- This leads to summaries that better reflect the intended message and nuances of the original content.

Flexibility in Summary Generation:

- ROUGE-S allows the model to recognize and reproduce important word associations and concepts that may be separated by other words, resulting in more flexible and semantically rich summaries.

3.1.3 Model Configuraions

We evaluated three model configuration in our experiments:

Model Variant	Embedding Type	Number of Parameters
GloVe-based	Pre-trained GloVe	39M
Non-GloVe (Random)	Randomly Initialized	93.5M
Pre-trained (T5-small)	Pre-trained (HuggingFace)	77M
Pre-trained (Pegasus)	Pre-trained (HuggingFace)	568M

Table 1: Different model configurations

The GloVe-based model utilizes 300-dimensional pre-trained embeddings, resulting in a smaller parameter count (39 million). In contrast, the non-GloVe model, with its randomly initialized 512-dimensional embeddings, has a significantly larger parameter space (93.5 million). The pre-trained models have 77 and 568 million parameters. This difference in the number of parameters can affect both the computational resources required for training and the model's ability to generalize, with larger models potentially being more prone to overfitting if not trained carefully.

3.2 Training

The training process of a Transformer-based text summarizer involves optimizing the model to predict the summary tokens for a given input sequence accurately. This section provides an overview of the training setup, including the loss function, optimization strategy, and learning rate schedule.

3.2.1 Loss Function: Cross-Entropy

Cross-entropy loss is used as the objective function during training. It measures the difference between the predicted probability distribution over the vocabulary and the actual target tokens in the summary. The formula for the cross-entropy loss is as follows:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{V} y_{ij}\log(\hat{y}_{ij})$$

Where:

- N: The number of tokens in the summary.
- V: Vocabulary size.

- y_{ij}: Ground truth token probability (1 for the correct token, 0 otherwise).
- $\hat{y}_{ij}$: Predicted probability for the j-th token in the vocabulary.

During training, padding tokens are ignored to prevent them from contributing to the loss. This is achieved using a **padding mask** that excludes such tokens from the loss computation.

3.2.2 Optimizer: AdamW

The **AdamW optimizer** is used to update the model's weights during training. It is an improved version of the Adam optimizer that decouples weight decay from the gradient updates, resulting in better generalization. The key update rule for AdamW is:

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{m_t}{\sqrt{v_t} + \epsilon} - \eta \cdot \lambda \cdot \theta_t$$

Where:

- θ_t: Model parameters at step t.
- m_t: Exponential moving average of gradients (momentum term).
- v_t: Exponential moving average of squared gradients (variance term).
- η: Learning rate.
- λ: Weight decay coefficient.
- ϵ: A small constant to prevent division by zero.

AdamW improves over traditional Adam by separately applying the weight decay term $\lambda \cdot \theta_t$ directly to the parameters, which ensures better handling of overfitting.

3.2.3 Learning Rate Scheduler

A **learning rate scheduler** is applied to improve training convergence and stability. A **warm-up strategy** is used, where the learning rate gradually increases in the initial training steps (the warm-up phase) and then decays according to a predefined schedule. This prevents large gradient updates early on, which could destabilize the model, and ensures smoother convergence as training progresses.

3.2.4 Training Pipeline

The training process follows these steps:

1. **Data Preparation**: Input text and target summaries are tokenized, converted into sequences of integers, and padded to a fixed length.
2. **Model Input**: The tokenized input sequence is fed into the Transformer model, producing predicted probabilities for each token in the summary.
3. **Loss Calculation**: Cross-entropy loss is computed between the predicted and target tokens, ignoring padding tokens.
4. **Backpropagation**: Gradients of the loss function with respect to model parameters are computed using backpropagation.

5. **Parameter Update**: The AdamW optimizer updates the model parameters using the computed gradients.

6. **Evaluation**: After each epoch, the model's performance is evaluated on a validation set using metrics like BLEU, ROUGE, or METEOR.

3.2.5 Training Result

This training setup ensures the Transformer-based text summarizer learns effectively and generalizes well to unseen data, producing high-quality summaries.]

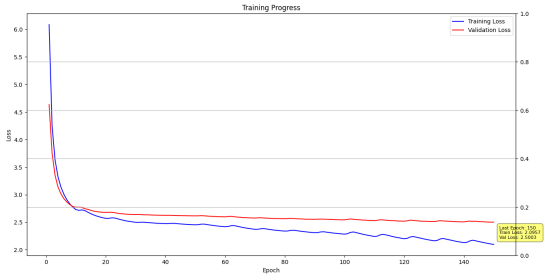


Figure 8: GloVe-Embedding Model

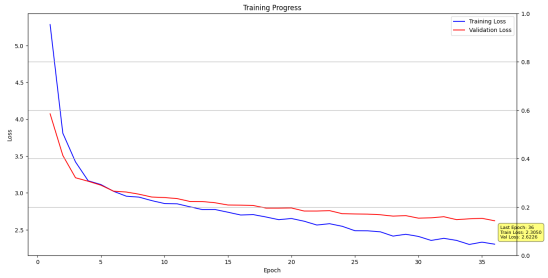


Figure 9: Random Embedding Model

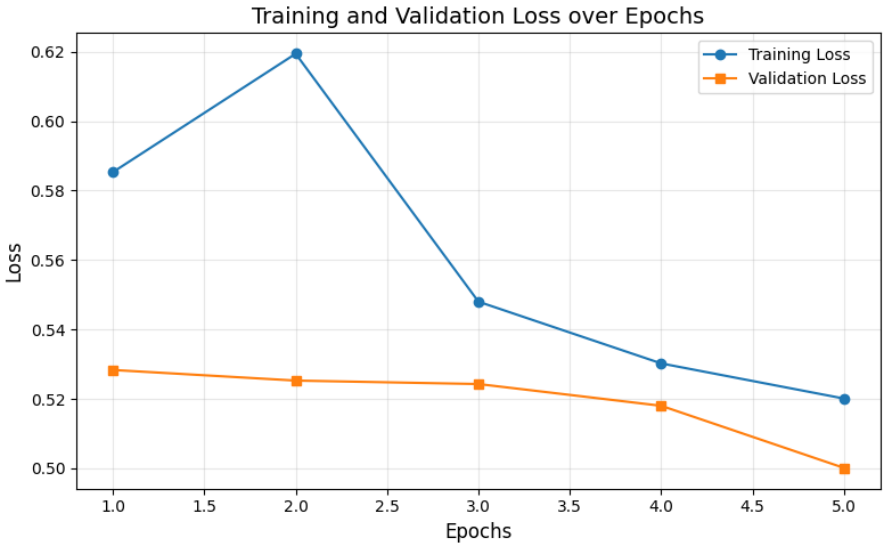


Figure 10: Pre-Trained Model

The pretrained model performs the best, demonstrating faster convergence and strong generalization, making it the most suitable choice overall. The GloVe embedding model is stable with steady loss reduction, indicating good generalization, though it slightly underfits and has room for improvement. In contrast, the random higher-dimensional model struggles to converge and shows higher overall losses, suggesting it is less effective without further optimization or significantly more training epochs.

3.2.6 Training Details

The models were trained on a Tesla T4 GPU provided by Kaggle, which offers a balance of performance and accessibility for research purposes. The total training time for the models was approximately **140 hours**.

Due to differences in convergence rates and resource limitations, the number of training epochs varied for each model variant:

Model Variant	Epochs
GloVe-based	150
Non-GloVe (Random)	36
Pre-trained (T5-small)	5

Table 2: The number of training epochs per model

The GloVe-based model, leveraging pre-trained embeddings, required more epochs (**150**) to fine-tune its parameters and achieve optimal performance. The non-GloVe model, with randomly initialized embeddings, converged faster but showed signs of potential overfitting beyond **36 epochs**. The pre-trained model, T5-small, was trained for **5 epochs** only to adapt to specific tasks.

3.3 Experimental results

Model	ROUGE-1	ROUGE-2	ROUGE-L
Pretrained google/pegasus-xsum	0.4743	0.2473	0.3954
Pretrained google/flan-t5-small	0.2772	0.0845	0.2192
Finetuned google/flan-t5-small	0.3311	0.1131	0.2622
GloVe	0.2822	0.0807	0.2219
Non-GloVe	0.2139	0.0493	0.1686

Table 3: The result of different models

The results of the GloVe and non-GloVe models show that the GloVe-based approach outperforms the non-GloVe model across all ROUGE metrics:

- **ROUGE-1**: GloVe achieves 0.2822 compared to non-GloVe's 0.2139.
- **ROUGE-2**: GloVe reaches 0.0807, while non-GloVe lags at 0.0493.
- **ROUGE-L**: GloVe scores 0.2219 versus 0.1686 for non-GloVe.

This performance gap highlights the advantage of leveraging pre-trained embeddings like GloVe for text generation or summarization tasks. GloVe provides richer word representations that likely helped the model capture better context and semantics.

Although the non-GloVe model uses higher-dimensional vectors (512 dimensions compared to GloVe's 300), which theoretically allows it to capture more content, the increased number of parameters makes it more challenging to train effectively without sufficient resources and training time. This likely led to underfitting, as the model was unable to fully leverage its higher-dimensional capacity.

However, both models developed by my team are significantly behind pretrained models like Pegasus-XSum or Flan-T5. With further optimization, additional training epochs, and better resource allocation, the non-GloVe model may have the potential to outperform GloVe and narrow the gap with pretrained models.

4 Discussion

Pros

- **Modern Techniques:** The model leverages a Transformer-based architecture with multi-head attention and robust design principles, which are highly effective for sequence-to-sequence tasks like text summarization.
- **Effective Performance Under Constraints:** Despite limited resources, reduced training dataset size, and the constraints of GPU, the model successfully generates meaningful summaries. This demonstrates the robustness of the architecture and its adaptability to constrained environments.
- **Modular and Flexible Design:** The model's structure allows for easy customization of parameters, such as the number of layers, attention heads, and embedding dimensions, making it versatile for different task requirements.
- **Foundation for Future Enhancements:** The architecture provides a strong foundation for iterative improvements, allowing it to scale with better resources and datasets.

Cons

- **High Memory and Computational Requirements:** The Transformer model requires significant memory and computational power, particularly with a large number of layers, attention heads, and embedding dimensions. This poses a bottleneck on resource-limited hardware.
- **Limited Training Dataset:** The training dataset was reduced to 3200 examples to meet resource constraints. While this sped up training, it limited the model's capacity for generalization on larger or unseen datasets.
- **Suboptimal Training Due to Resource Constraints:** Resource limitations prevented the model from being trained to its full potential, which may have restricted its performance and refinement of outputs.

Future Work

- **Enhanced Computational Resources:** Training the model on robust hardware with higher computational power would allow for the use of larger batch sizes, additional layers, and increased embedding dimensions, significantly boosting performance.
- **Expanded Dataset Utilization:** Leveraging the full dataset or incorporating additional datasets would improve the model's generalization and ability to handle diverse inputs effectively.
- **Optimizing Training Strategy:** Employing advanced optimization techniques and further tuning hyperparameters could improve efficiency and performance even in resource-constrained settings.
- **Embedding Dimensionality and Quality:** Increasing the embedding dimensions or exploring contextual embeddings (e.g., BERT or T5) could further enhance the semantic understanding and quality of generated summaries.

- **Scalability:** The model's modular design makes it well-suited for iterative scaling and experimentation, enabling future researchers to build upon this work for more refined results.

5 Conclusion

This project demonstrates the effectiveness and adaptability of Transformer-based architectures for text summarization tasks, even under constrained resources. The model's ability to generate meaningful summaries with a reduced dataset and limited computational power highlights the robustness and versatility of its design. Despite facing challenges such as high memory requirements, a limited dataset, and resource constraints, the project establishes a strong foundation for future work. By enhancing computational resources, expanding dataset utilization, optimizing training strategies, and exploring advanced embedding techniques, this model can be further refined and scaled for improved performance. The modular nature of the architecture ensures that it remains a flexible and promising platform for ongoing research and development in text summarization.

References

[1] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. In Advances in Neural Information Processing Systems (NeurIPS), 30.

[2] CS224N: Natural Language Processing with Deep Learning Stanford / Winter 2025.

[3] Jalammar, J. (2018). The Illustrated Transformer.

[4] Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. "Layer Normalization." arXiv.