Empathetic Response Generation Using a Transformer-Based Seq2Seq Model

AIM 829 - Natural Language Processing

A Project Report Submitted
in Partial Fulfillment of the Requirements
for the Award of the Degree

MASTER OF TECHNOLOGY

in

Computer Science and Engineering

Submitted by

Mohit Sharma, Parv Gatecha, Himanshu Shivhare, Mohit Gupta

(MT2024091, MT2024108, MT2024058, MT2024049)

Submitted to

Department of Computer Science and Engineering
International Institute of Information Technology
Bangalore - 560100, India

April 2025

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

With the growing influence of Artificial Intelligence in healthcare, Natural Language Processing (NLP) has shown promise in analyzing mental health-related text. Transformer-based models, known for their self-attention mechanisms and parallel processing, have significantly improved performance in sequence modeling tasks.

1.2 Project Motivation

This project aims to build a Transformer-based Seq2Seq model tailored for a mental health context. By training the model on conversational or condition-specific data, we aim to enable the system to understand and generate contextually appropriate responses.

1.3 Objectives

- To implement a Transformer-based sequence-to-sequence model.
- To apply the model to a dataset involving mental health-related language.
- To evaluate its performance in generating context-aware sequences.

1.4 Report Overview

The subsequent chapters detail the related work, model architecture, implementation, results, and future directions.

Chapter 2

Dataset Description

2.1 Overview

The *EmpatheticDialogues* dataset [1], released by Facebook AI, consists of over 24,000 open-domain conversations grounded in emotional contexts. Each dialogue simulates an interaction where one speaker describes an emotional situation and the other responds empathetically.

2.2 Structure

Each entry includes:

- An emotion label (from 32 predefined emotions)
- A situation description
- A multi-turn conversation between a speaker and a listener

2.3 Dataset Splits

The dataset is divided into:

- **Train**: 19,533 conversations
- **Validation**: 2,770 conversations
- **Test**: 2,547 conversations

2.4 Relevance to This Work

This dataset is used to train and evaluate a sequence-to-sequence Transformer model aimed at generating emotionally appropriate and empathetic responses in dialogue systems.

Chapter 3

DATA SET PREPROCESSING

The preprocessing phase included several key steps to prepare the dialogue data and ensure reproducibility:

- Dataset subset size was set using `DATASET_SUBSET_SIZE = None`, allowing full usage or controlled sampling.
- Maximum sequence length: `MAX_LEN = 60`.
- Vocabulary size for WordPiece tokenizer: `VOCAB_SIZE = 20000`.
- Tokenizer was saved to `wp_tokenizer.json` for reuse.
- Batch size: `BATCH_SIZE = 32`; learning rate: `LEARNING_RATE = 0.0005`.
- Early stopping configured with `PATIENCE = 5`, max epochs: `N_EPOCHS = 50`.
- Model configuration: `D_MODEL = 256`, `N_HEADS = 8`, `ENC_LAYERS = 3`, `DEC_LAYERS = 3`, `PF_DIM = 512`, `DROPOUT = 0.2`.
- Reproducibility was ensured by setting all relevant seeds including `random`, `torch`, and `PYTHONHASHSEED`.
- Special tokens defined: `<PAD>`, `<SOS>`, `<EOS>`, `<UNK>`.

3.1 Dialogue Pair Extraction

To train the model, input-output pairs were extracted using the following logic:

- Converted HuggingFace dataset to a Pandas DataFrame.
- Optionally sampled a subset of conversation IDs.
- Grouped the utterances by conversation ID and sorted them by their indices.
- Generated input-output pairs using adjacent utterances.
- Skipped conversations with fewer than two utterances.

This ensured that the model could learn coherent dialogue patterns effectively.

3.2 Tokenizer Loading

To handle vocabulary and tokenization, a WordPiece tokenizer [2] was either trained from scratch or loaded from a pre-existing configuration. This enabled flexible usage based on whether the tokenizer file already existed or not.

Tokenizer Training (if not already available)

- A **WordPiece tokenizer** was initialized with the `<UNK>` token.
- The pre-tokenization strategy used was simple whitespace-based tokenization.
- Special tokens included in the tokenizer were: `<PAD>`, `<SOS>`, `<EOS>`, `<UNK>`.
- Training was performed using a generator that yielded alternating batches from both input and target utterances.
- A `WordPieceTrainer` was configured with a target vocabulary size of 20,000 and special tokens.
- Once training completed, the tokenizer was saved to a file for reuse across runs.

Tokenizer Loading (if already trained)

- If the tokenizer file existed at the predefined path, it was directly loaded using HuggingFace's `Tokenizer.from_file()` method [3].
- This step significantly reduced startup time by reusing the trained tokenizer and avoiding retraining.

Vocabulary and Special Token Indices

- After loading or training, the vocabulary size was recorded and used as both `INPUT_DIM` and `OUTPUT_DIM`.
- Token IDs were retrieved for each special token:
  - `<PAD>` → `PAD_IDX`
  - `<SOS>` → `SOS_IDX`
  - `<EOS>` → `EOS_IDX`
  - `<UNK>` → `UNK_IDX`
- A check ensured all special tokens were present. If missing, a manual addition process could be triggered.

3.3 Numericalization, Padding, and DataLoader Creation

After tokenization, we performed numericalization and padding operations to transform raw dialogue utterances into fixed-length tensor inputs suitable for model training.

Text to Sequence Encoding

To convert each utterance into a sequence of token IDs, the following function was used:

```
def text_to_sequence_tokenizer(text, tokenizer):
    """Encodes text to a list of token IDs, adding SOS/EOS."""
    encoded = tokenizer.encode(text)
    return [SOS_IDX] + encoded.ids + [EOS_IDX]
```

- This function utilizes the tokenizer's `encode()` method for tokenization and token ID conversion.
- Special tokens `<SOS>` and `<EOS>` are manually added to each sequence.

Sequence to Text Decoding

To convert model predictions back into human-readable text:

```
def sequence_to_text_tokenizer(sequence, tokenizer):
    """Decodes a list of token IDs back to text, removing special tokens."""
    ids = [idx for idx in sequence if idx not in [PAD_IDX, SOS_IDX, EOS_IDX]]
    return tokenizer.decode(ids)
```

- Special token IDs are removed before decoding.
- The function uses `decode()` to reconstruct text from the token ID sequence.

Padding and Truncating Sequences

For efficient batching and uniform input size, each sequence was padded to a fixed maximum length using:

```
def pad_sequence(seq, max_len, pad_idx):
    """Pads or truncates a sequence to max_len."""
    seq = seq[:max_len]
    padded = seq + [pad_idx] * (max_len - len(seq))
    return padded
```

- Sequences longer than `max_len` are truncated.
- Shorter sequences are padded with the `<PAD>` token to reach the desired length.

DataLoader Preparation

- The above functions serve as key components in preparing input-output pairs for PyTorch's `DataLoader`.
- Numericalized and padded sequences are batched together for parallel training of the sequence-to-sequence model.

Chapter 4

Architecture of Transformer

Transformers [4] are a type of neural network used for processing sequences. Unlike recurrent neural networks, they do not have recurrent connections, which means they lack explicit memory of previous states. However, this allows the model to process the entire sequence at once rather than element-by-element, and it overcomes this lack of memory by perceiving the entire sequence simultaneously.
Transformers require more memory during training.

4.1 Positional Encoding in Transformer Models

- **Objective**
  - To inject information about the relative and absolute position of tokens in a sequence.
  - Enhances the Transformer model's ability to recognize the order of elements in input data.
- **Motivation**
  - The Transformer model processes all tokens simultaneously and doesn't inherently know their order.
  - Positional encoding adds contextual sequence information, enabling the model to capture dependencies between words based on their positions.
- **Methodology**
  - Computes a positional encoding matrix using sine and cosine functions at different frequencies.
  - For even indices:

$$\mathrm{PE}_{(\mathrm{pos},2i)} = \sin\left(\frac{\mathrm{pos}}{10000^{2i/d_{\mathrm{model}}}}\right)$$

  - For odd indices:

$$\mathrm{PE}_{(\mathrm{pos},2i+1)} = \cos\left(\frac{\mathrm{pos}}{10000^{2i/d_{\mathrm{model}}}}\right)$$

  - These position vectors are added to the input embeddings.
  - The `register_buffer` method is used to store the positional encoding in the model without treating it as a trainable parameter.

- **Key Parameters**
  - d_{model}: The dimension of the embedding vector (e.g., 512). Must match the model's internal representation size.
  - Dropout: Dropout probability (default = 0.1) applied after encoding, helping to prevent overfitting.
  - max_len: Maximum sequence length supported by the positional encoding (default = 5000).

- **Input Specification**
  - Accepts a tensor of shape [seq_len, batch_size, embedding_dim], which contains input embeddings for each token.

- **Output Specification**
  - Returns a tensor of the same shape, but with added positional encoding, followed by dropout for regularization.

- **Significance**
  - Allows the model to understand and leverage word order, which is essential for accurate predictions in NLP tasks.
  - Enables the Transformer to generalize well to different sequence lengths.
  - Maintains high computational efficiency by supporting full parallel processing during training and inference.

4.2 Multi-Head Attention

- **Objective**
  - To allow the model to jointly attend to information from different representation subspaces at different positions.
  - Enables the model to capture complex dependencies between words across different parts of the sequence.

- **Motivation**
  - Single attention heads can only capture certain relationships between tokens in the sequence.
  - Multi-head attention allows for multiple perspectives on the data by using different heads to focus on different aspects of the sequence.

- **Methodology**
  - Splits the input into multiple smaller "heads" for parallel computation.

  - Each head independently computes an attention score and contributes to a weighted sum of the input tokens.
  - The outputs from all heads are concatenated and linearly transformed to obtain the final result.

- **Key Parameters**
  - d_{model}: The size of the model's input and output vectors.
  - n_{heads}: The number of attention heads used in the multi-head attention mechanism.
  - Dropout: Dropout probability applied after attention for regularization.

- **Input Specification**
  - Accepts input tensors of shape [seq_len, batch_size, d_model].

- **Output Specification**
  - Returns a tensor of the same shape [seq_len, batch_size, d_model] after applying attention.

- **Significance**
  - Helps the model to attend to different parts of the sequence simultaneously, improving its ability to learn complex patterns and dependencies.
  - Enables parallelization during training and inference, resulting in faster computation.

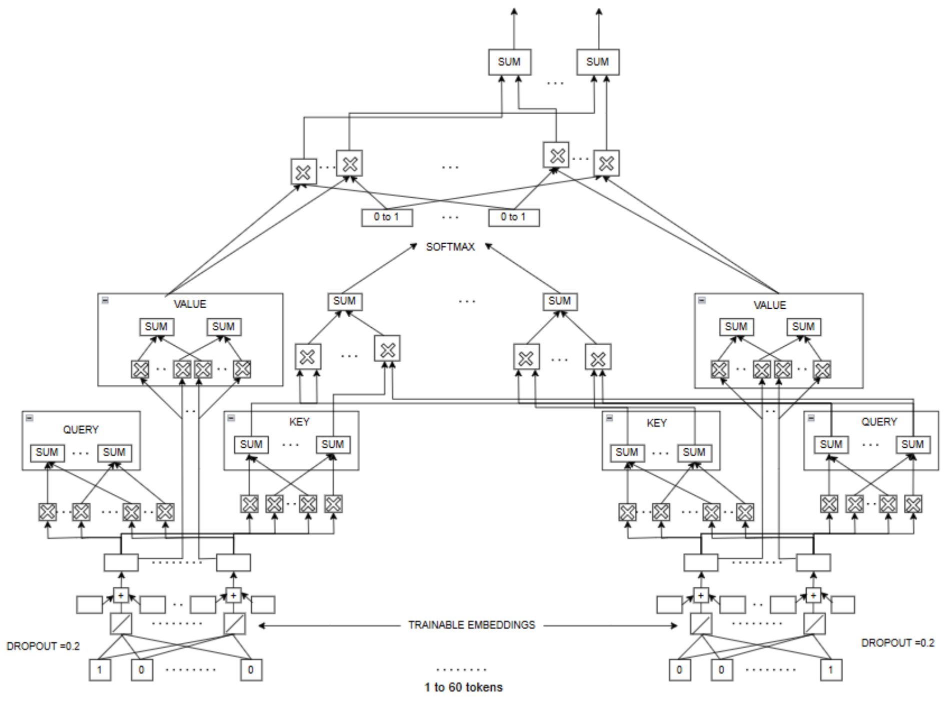

Figure 4.1: **Architecture of SELF-ATTENTION LAYER**

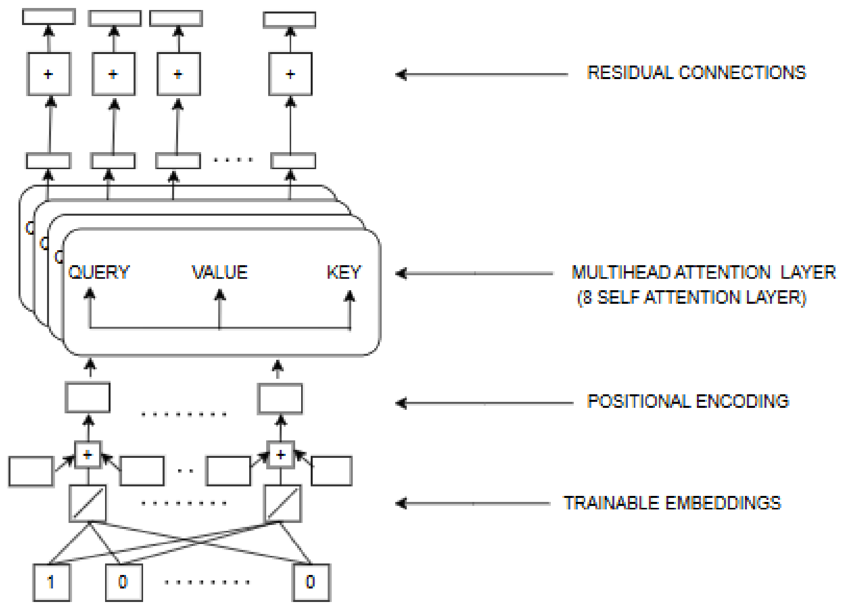

Figure 4.2: **COMPACT SELF-ATTENTION LAYER**

4.3 Position-wise Feedforward Network

- **Objective**
  - To apply a non-linear transformation to each token's representation independently.
  - Helps the model to learn complex mappings from input to output space.
- **Motivation**
  - Each token in the sequence should be processed independently to apply non-linear transformations.
  - A feedforward network adds complexity and flexibility to the model by providing a layer of non-linearities after the attention mechanism.
- **Methodology**
  - Applies two linear transformations with a ReLU activation function in between.
  - The first linear transformation projects the input into a higher-dimensional space, while the second brings it back to the original dimensionality.
  - Dropout is applied between the layers for regularization.
- **Key Parameters**
  - d_{model}: The size of the model's input and output vectors.
  - d_{ff}: The dimension of the feedforward network's hidden layer.
  - Dropout: Dropout probability applied after the ReLU activation to prevent overfitting.
- **Input Specification**
  - Accepts a tensor of shape [seq_len, batch_size, d_model].
- **Output Specification**
  - Returns a tensor of the same shape [seq_len, batch_size, d_model].
- **Significance**
  - Provides additional non-linear transformations to enrich token representations.
  - Helps the model learn more expressive features and relationships between tokens.

4.4 Encoder Layer

- **Objective**
  - To transform input sequences into a continuous representation that captures both semantic and positional information.
  - Consists of a self-attention mechanism followed by a position-wise feedforward network.
- **Motivation**
  - Each token in the input sequence needs to be processed while considering other tokens and their relative positions.
  - The encoder layer's self-attention mechanism allows each token to "attend" to all other tokens in the sequence.
- **Methodology**
  - The input sequence is passed through a multi-head self-attention mechanism.
  - A position-wise feedforward network follows the attention mechanism.
  - Layer normalization and residual connections are used to stabilize training.
- **Key Parameters**
  - d_{model}: The size of the model's input and output vectors.
  - n_{heads}: The number of attention heads.
  - d_{ff}: The dimension of the feedforward network.
  - Dropout: Dropout probability for regularization.
- **Input Specification**
  - Accepts a tensor of shape [seq_len, batch_size, d_model].
- **Output Specification**
  - Returns a tensor of the same shape [seq_len, batch_size, d_model].
- **Significance**
  - The encoder layer helps the model capture and process input sequences more effectively.
  - It allows the model to build contextual representations of the input data.

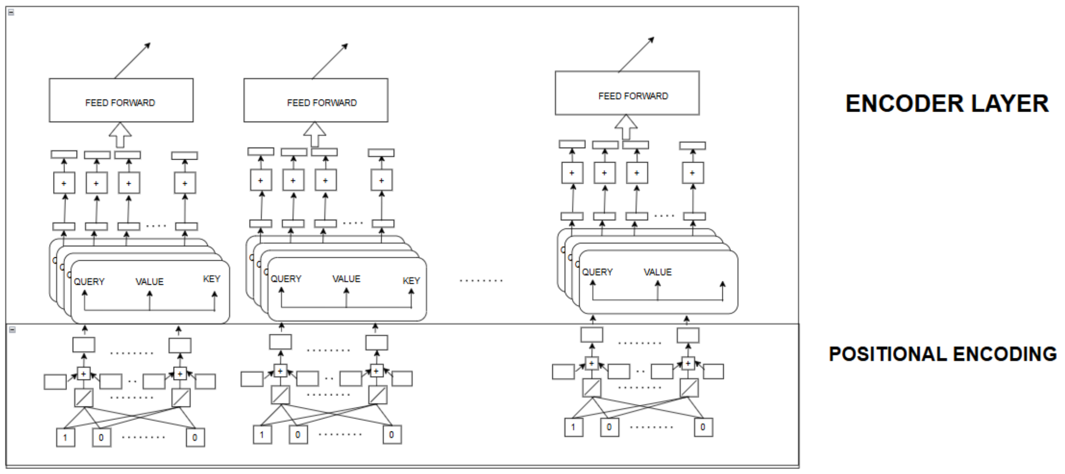

Figure 4.3: **ARCHITECTURE OF ENCODER LAYER**

4.5 Decoder Layer

- **Objective**
  - To generate output sequences from encoded input sequences, utilizing both self-attention and encoder-decoder attention mechanisms.
  - The decoder layer enables the model to produce output while attending to both the previously generated tokens and the input sequence.
- **Motivation**
  - The decoder needs to attend to previously generated tokens (masked self-attention) and to the encoder's output sequence.
  - This allows the decoder to generate contextually appropriate sequences.
- **Methodology**
  - The decoder uses a masked self-attention mechanism to prevent attending to future tokens.
  - A second multi-head attention layer attends to the encoder's output (cross-attention).
  - A position-wise feedforward network processes the output.
  - Layer normalization and residual connections help stabilize training.
- **Key Parameters**
  - d_{model}: The size of the model's input and output vectors.
  - n_{heads}: The number of attention heads.
  - d_{ff}: The dimension of the feedforward network.
  - Dropout: Dropout probability applied for regularization.
- **Input Specification**

  - Accepts the target sequence tensor of shape [trg_len, batch_size, d_model], the encoder output of shape [src_len, batch_size, d_model], and masks for both.

- **Output Specification**
  - Returns a tensor of shape [trg_len, batch_size, d_model] and the attention weights of shape [batch_size, n_heads, trg_len, src_len].

- **Significance**
  - The decoder layer enables the model to generate output sequences by attending to both previously generated tokens and the input sequence.
  - It plays a crucial role in tasks like translation, summarization, and generative text modeling.

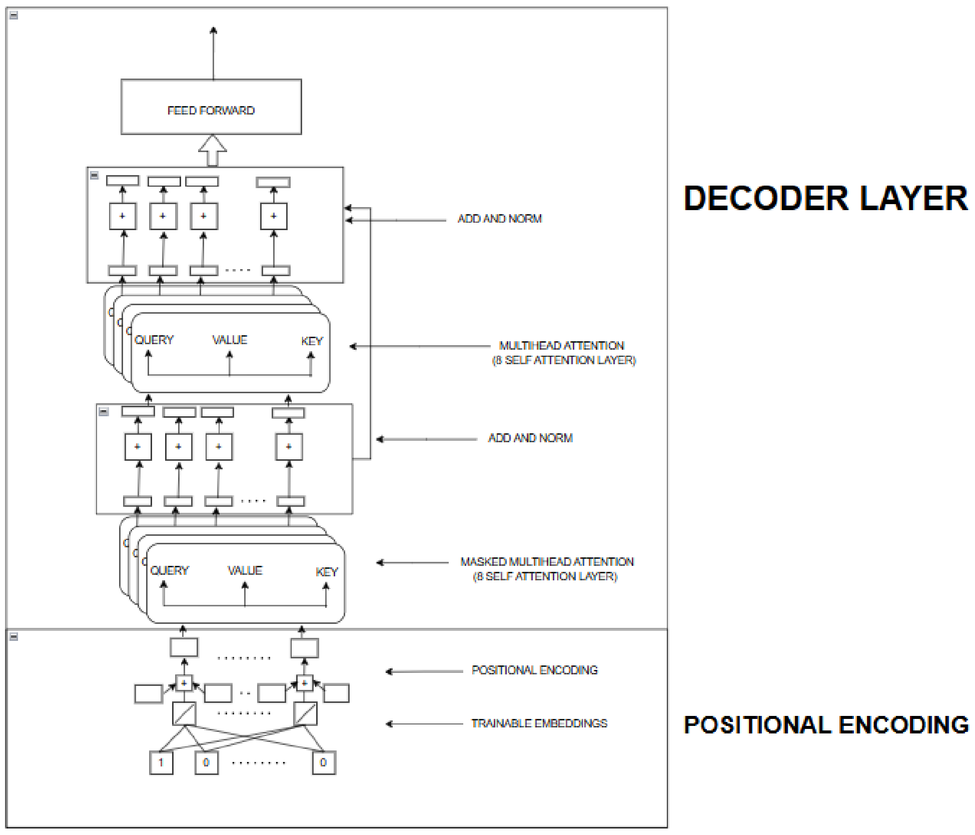

Figure 4.4: **ARCHITECTURE OF DECODER LAYER**

4.6 Encoder Module

Class: `Encoder(nn.Module)`
The Encoder is the stack of multiple Transformer Encoder layers responsible for encoding the input sequence into contextualized representations using self-attention and feedforward networks.

Arguments

- `input_dim`: The vocabulary size (number of unique tokens in source language).
- `d_model`: Dimensionality of token embeddings and model hidden size.
- `n_layers`: Number of encoder layers to be stacked (e.g., 3 in this implementation).
- `n_heads`: Number of attention heads in multi-head self-attention.
- `pf_dim`: Dimensionality of the feedforward network within each encoder layer.
- `dropout`: Dropout probability used across layers to prevent overfitting.
- `max_len`: Maximum length of input sequences. Used for creating positional encodings.

Layers Used

- `Embedding Layer`: Maps each input token to a dense vector of size `d_model`.
- `PositionalEncoding`: Adds positional information to the input embeddings.
- `EncoderLayer (stacked)`: A stack of Transformer Encoder layers that include multi-head attention and feedforward sublayers.
- `Dropout`: Applied after the embedding + positional encoding.

Forward Pass

1. Input token IDs of shape $[batch\_size, src\_len]$ are mapped to embeddings of size $[batch\_size, src\_len, d\_model]$.
2. The embeddings are scaled by $\sqrt{d_{\text{model}}}$ for stabilizing training.
3. The embeddings are then permuted to shape $[src\_len, batch\_size, d\_model]$ for compatibility with positional encoding and encoder layers.
4. Positional encoding is added to incorporate order information.
5. The result is passed through a stack of encoder layers (e.g., 3 layers), each applying self-attention and feedforward transformations.
6. The final output is returned in shape $[src\_len, batch\_size, d\_model]$.

Output

- Tensor of shape $[src\_len, batch\_size, d\_model]$ representing the encoded input sequence with context-aware representations.

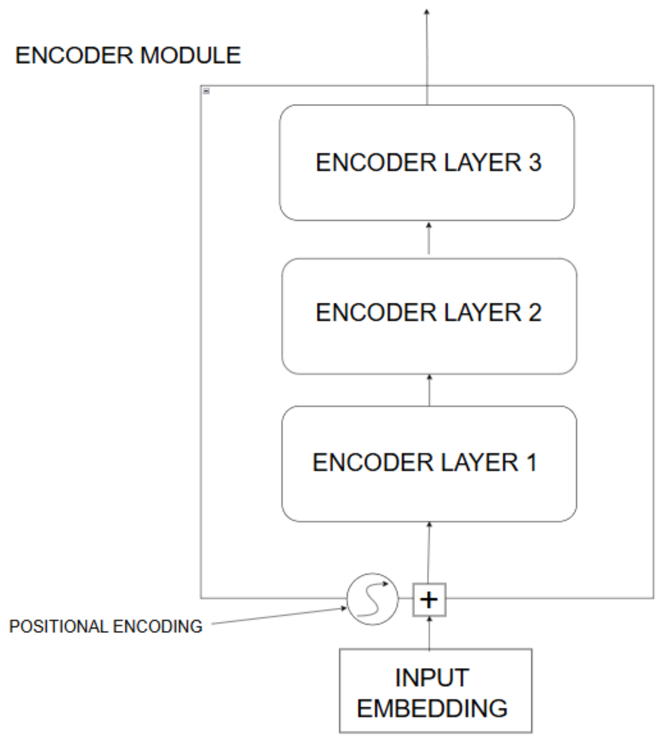

Figure 4.5: **ARCHITECTURE OF ENCODER MODULE**

4.7 Decoder Module

Class: `Decoder(nn.Module)`
The Decoder is the Transformer stack that generates the output sequence based on the encoder output and previously generated tokens.

Arguments

- `output_dim`: Size of the target vocabulary.
- `d_model`: Dimensionality of token embeddings and model hidden size.
- `n_layers`: Number of decoder layers to be stacked.
- `n_heads`: Number of attention heads in multi-head attention.
- `pf_dim`: Dimensionality of the position-wise feedforward network.
- `dropout`: Dropout probability for regularization.
- `max_len`: Maximum target sequence length for positional encoding.

Layers Used

- `Embedding Layer`: Converts target token IDs to dense vectors.
- `PositionalEncoding`: Adds positional information to target embeddings.
- `DecoderLayer (stacked)`: Applies masked self-attention, encoder-decoder attention, and feedforward operations.
- `Linear Layer`: Projects decoder outputs to vocabulary logits.
- `Dropout`: Applied after token embedding + positional encoding.

Forward Pass

1. The target token IDs of shape $[batch\_size, trg\_len]$ are mapped to embeddings of shape $[batch\_size, trg\_len, d\_model]$.
2. Embeddings are scaled by $\sqrt{d_{\text{model}}}$.
3. Shape is permuted to $[trg\_len, batch\_size, d\_model]$ for compatibility.
4. Positional encoding is added to the target embeddings.
5. The result is passed through a stack of decoder layers. Each layer:
    - Applies masked multi-head self-attention on the target.
    - Applies encoder-decoder attention using encoder outputs.
    - Passes through a feedforward network.
6. The final output is passed through a linear layer to get logits over the target vocabulary.

Output

- `output`: Tensor of shape $[trg\_len, batch\_size, output\_dim]$ containing vocabulary logits.
- `attention`: Attention weights from the last encoder-decoder attention layer.

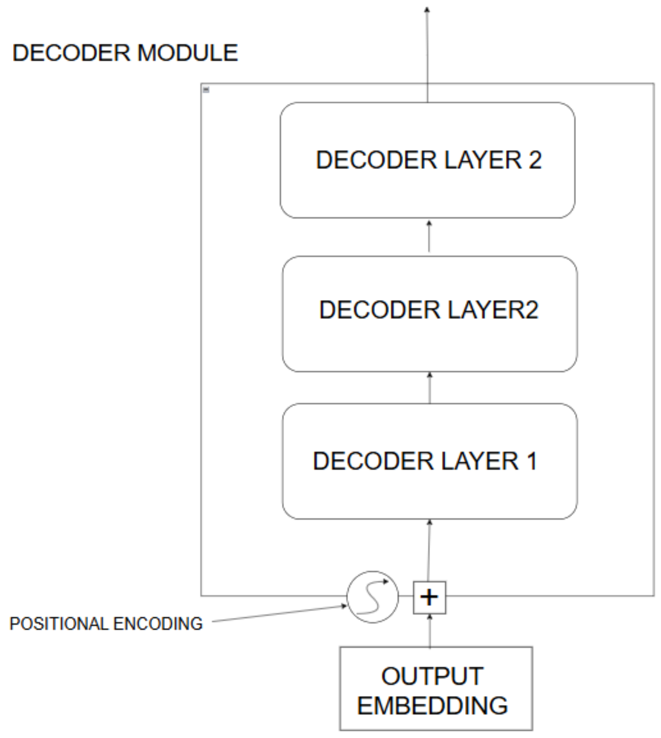

Figure 4.6: **ARCHITECTURE OF DECODER MODULE**

4.8 Seq2Seq Transformer

Class: `Seq2SeqTransformer(nn.Module)`
This class encapsulates the complete Transformer-based sequence-to-sequence model, integrating both encoder and decoder modules. It also handles the creation of attention masks required for the Transformer mechanism.

Arguments

- `encoder`: The Transformer encoder module.
- `decoder`: The Transformer decoder module.
- `src_pad_idx`: Padding token index for the source sequence.
- `trg_pad_idx`: Padding token index for the target sequence.
- `device`: Computation device (CPU/GPU).

Key Components

- `make_src_mask`: Creates a mask for source tokens to ignore padding during attention.
- `make_trg_mask`: Creates a mask for target tokens that masks padding and future tokens (to maintain auto-regression).

Source Masking

- Padding tokens in the source are masked using a boolean mask.
- Output shape: $[batch\_size, 1, 1, src\_len]$ for broadcasting in multi-head attention.

Target Masking

1. **Padding Mask:** Prevents attention to padded tokens. Shape: $[batch\_size, 1, 1, trg\_len]$.
2. **Subsequent Mask:** A lower triangular matrix allowing attention only to previous and current positions. Shape: $[trg\_len, trg\_len]$.
3. **Combined:** Logical AND of both masks. Shape: $[batch\_size, 1, trg\_len, trg\_len]$.

Forward Pass

1. Source and target token IDs are moved to the specified device.
2. Source and target masks are computed.
3. Source tokens are passed through the encoder.
4. Target tokens and encoder output are passed through the decoder.

Output

- `output`: Tensor of shape $[trg\_len, batch\_size, output\_dim]$, representing vocabulary logits for each position.
- `attention`: Tensor of shape $[batch\_size, n\_heads, trg\_len, src\_len]$, containing encoder-decoder attention weights.

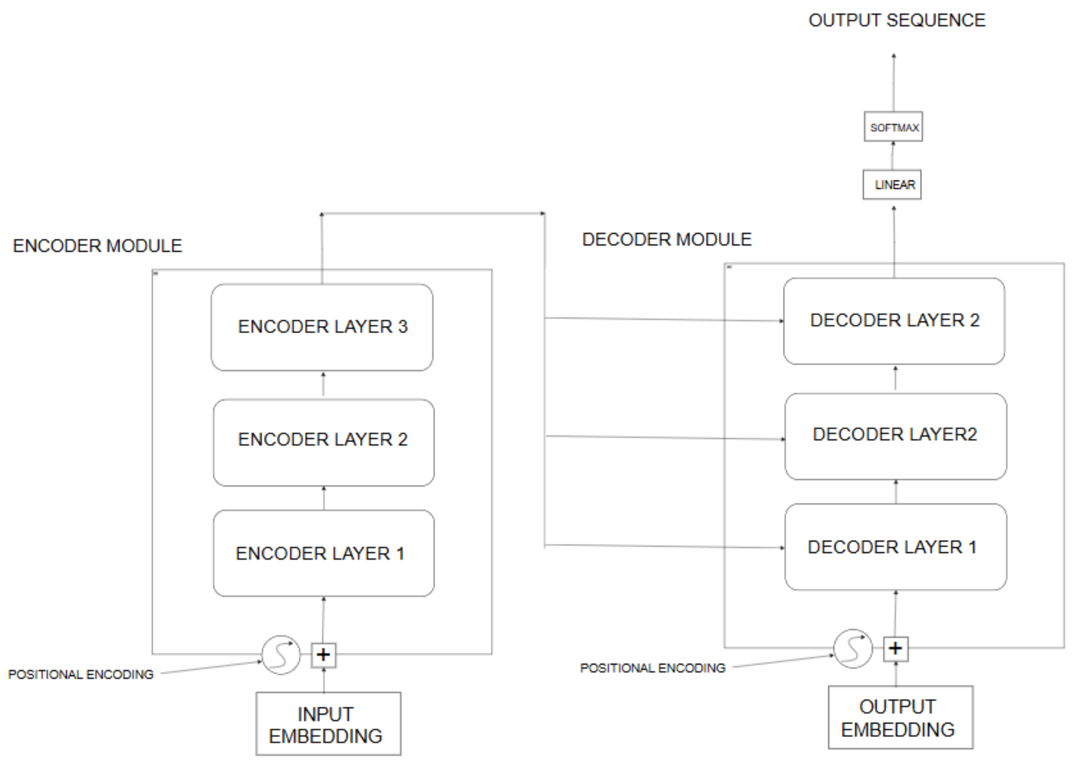

Figure 4.7: **ARCHITECTURE OF SEQ TO SEQ TRANSFORMER**

Chapter 5

Word Embedding Techniques Used

This project explores three different approaches to initialize and utilize word embeddings within the Transformer-based encoder architecture: Trainable Embeddings, GloVe, and FastText. These methods influence how textual input is represented and encoded before being processed by the encoder layers.

1. Trainable Embeddings

- Embedding weights are initialized randomly and updated during training.
- Implemented using the `nn.Embedding` module in PyTorch.
- Does not rely on any pre-trained external vectors.
- Suitable when task-specific word representations are desired.
- Simplifies integration as no additional loading or preprocessing is needed.

2. GloVe Embeddings

GloVe (Global Vectors for Word Representation) [5] utilizes pre-trained word vectors trained on large-scale corpora.

- Embedding weights are loaded externally and assigned to the model.
- Helps capture general semantic and syntactic relationships between words.
- Can be used in two ways:
  - **Frozen**: Embeddings remain fixed during training.
  - **Trainable**: Embeddings are fine-tuned with the model.
- Requires preprocessing to align the vocabulary and embedding dimensions.

3. FastText Embeddings

FastText [6] is similar to GloVe but includes subword information via character n-grams.

- Improves handling of rare and out-of-vocabulary words.

- Integrated in the same way as GloVe embeddings.
- Useful for morphologically rich languages or noisy text.

Integration and Implementation Differences

- The key variation occurs in how the embedding layer is initialized.
- For trainable embeddings:
  - No external data is required.
  - Embeddings are initialized using:

    ```
    self.tok_embedding = nn.Embedding(input_dim, d_model)
    ```

- For pre-trained embeddings (GloVe/FastText):
  - The embedding matrix is passed as a tensor to the encoder.
  - Initialized using:

    ```
    self.tok_embedding.weight.data.copy_(pretrained_embeddings)
    ```

  - Optional control over training behavior using:

    ```
    self.tok_embedding.weight.requires_grad = False
    ```

- These differences are confined to the encoder's initialization phase.
- The downstream processing remains unchanged for all three types.

Chapter 6

Training Procedure and Hyperparameter

The training of the Transformer-based Seq2Seq model was performed using a carefully monitored loop with early stopping, model checkpointing, and performance tracking. The process was designed to ensure efficient convergence while avoiding overfitting.

Training Strategy

- The model is trained using the `train()` function, which performs one full pass over the training dataset per epoch.
- Evaluation is conducted at the end of each epoch using the `evaluate()` function on the validation set.
- The total training time, as well as the time per epoch, is logged for transparency and reproducibility.
- Training and validation loss values are recorded and plotted to monitor the learning progress.

Key Hyperparameters

- **Number of Epochs (N_EPOCHS)**: Defines the maximum number of passes over the entire training data. Early stopping may terminate the process before this limit is reached.
- **Clip (CLIP)**: Gradient clipping value to avoid exploding gradients during backpropagation.
- **Criterion**: The loss function used for training, typically CrossEntropyLoss for language modeling or sequence prediction tasks.
- **Optimizer**: An optimization algorithm such as Adam [7] is used to update the model parameters based on gradients.
- **Scheduler**: A learning rate scheduler, e.g., `ReduceLROnPlateau`, adjusts the learning rate based on validation loss to improve convergence.

- **Patience (PATIENCE)**: The number of consecutive epochs without improvement in validation loss after which early stopping is triggered.

Model Checkpointing and Early Stopping

- The best model is saved using `torch.save()` [8] whenever the validation loss improves.
- Early stopping is implemented to halt training if validation loss does not improve for a pre-defined number of epochs (`PATIENCE`), avoiding unnecessary computation and overfitting.

Performance Metrics

- **Loss**: Tracks the negative log-likelihood or cross-entropy loss, depending on the criterion.
- **Perplexity (PPL)**: Computed as the exponential of the loss, perplexity serves as a widely used metric in sequence modeling to evaluate the model's ability to predict the next token.

Final Output

- Total training time is displayed in minutes and seconds.
- The best validation loss achieved is reported.
- The model with the best validation performance is saved to disk in the file: `empathetic-transformer-basic-best.pt`.

Chapter 7

Inference and Sampling Strategy

To generate a response from the trained Transformer model, we employ a word-level autoregressive decoding function that supports multiple sampling strategies: greedy decoding, top-k, and top-p (nucleus) sampling. The function is designed to simulate human-like dialogue by controlling randomness through the `temperature` hyperparameter.

Input Preprocessing

Given an input sentence, we tokenize it using a word-level tokenizer and truncate the sequence if it exceeds a predefined maximum length. The tokenized input is then converted into a tensor and passed to the encoder:

$$\text{src\_tensor} = \text{word\_to\_id}(\text{tokens}) \in \mathbb{Z}^{1\times L}, \quad L \leq \texttt{MAX\_LEN}$$

$$\text{enc\_src} = \text{Encoder}(\text{src\_tensor}, \text{src\_mask})$$

Autoregressive Decoding

The decoding begins with the `SOS` (Start-of-Sequence) token and generates one token at a time until either the `EOS` (End-of-Sequence) token is generated or the maximum length is reached.

At each timestep:

1. The target sequence so far is passed through the decoder.
2. Logits for the next token are extracted and scaled using the temperature $\tau > 0$ [9]:
$$\mathbf{z}_{\text{scaled}} = \frac{\mathbf{z}}{\tau}$$
3. A softmax is applied to obtain the probability distribution over the vocabulary:
$$\mathbf{p} = \text{softmax}(\mathbf{z}_{\text{scaled}})$$

Sampling Strategies

The function supports three decoding strategies:

- **Greedy decoding:** Selects the token with the highest probability:

$$\hat{y}_t = \arg\max_j p_j$$

- **Top-k sampling** [10]: Retains only the top-k tokens with the highest probabilities, normalizes over them, and samples:

$$\text{topk\_indices} = \text{argsort}(p)[-k:]$$

$$\hat{y}_t \sim \text{Categorical}(p_{\text{topk}})$$

- **Top-p (nucleus) sampling** [11]: Selects the smallest set of tokens whose cumulative probability exceeds p:

$$\text{sorted\_probs}, \text{sorted\_indices} = \text{sort}(p, \text{descending=True})$$

$$\text{cumsum} = \text{cumsum}(\text{sorted\_probs}) > p$$
$$\hat{y}_t \sim \text{Categorical}(p_{\text{filtered}})$$

Termination

The process continues until the end-of-sequence token is predicted or the maximum generation length is reached. The predicted token IDs are converted back into words to form the final response.

Hyperparameters

- `MAX_LEN`: Maximum length of the generated sequence.
- `temperature` $\in (0, \infty)$: Controls randomness; lower values lead to more deterministic outputs.
- `strategy`: Sampling strategy – `greedy`, `topk`, or `topp`.
- `k`: Number of candidates for top-k sampling.
- `p`: Cumulative probability threshold for top-p sampling.

Example

Given the input: "`I'm feeling lonely today`", the model generates a response based on the learned weights and the specified sampling strategy. This flexible decoding function enables diverse response generation suitable for empathetic dialogue systems.

Chapter 8

RESULTS

The following tables summarize key performance metrics for each embedding type. Table 8.1 shows the epoch with the best validation loss and convergence time, while Table 8.2 presents the resulting BLEU score[12] on the test set.

Embedding Type	**Best Val. Epoch**	**Conv. Epochs**
Trainable Embeddings	12	50
GloVe	15	40
FastText	20	30

Table 8.1: Model Convergence Metrics Across Different Embedding Methods

Embedding Type	**BLEU Score**
Trainable Embeddings	0.84
GloVe	0.18
FastText	0.82

Table 8.2: Model BLEU Scores Across Different Embedding Methods

8.1 Observations

- **Trainable Embeddings**: While this approach took the longest to converge (50 epochs, see Table 8.1), it achieved a high BLEU score of 0.84 (Table 8.2), indicating strong performance after sufficient task-specific tuning. It provides flexibility in learning word representations suitable for customization.

- **GloVe**: This embedding method converged moderately fast (40 epochs, Table 8.1) but yielded a significantly lower BLEU score (0.18, Table 8.2). While leveraging general pre-tned knowledge, it may not have adapted as well to the specific empathetic dialogue task compared to the other methods in this experiment.

- **FastText**: FastText converged the quickest (30 epochs, Table 8.1) and achieved a high BLEU score of 0.82 (Table 8.2), comparable to trainable embeddings but with faster training. Its subword information likely contributed to effective handling of the vocabulary, balancing speed and performance.

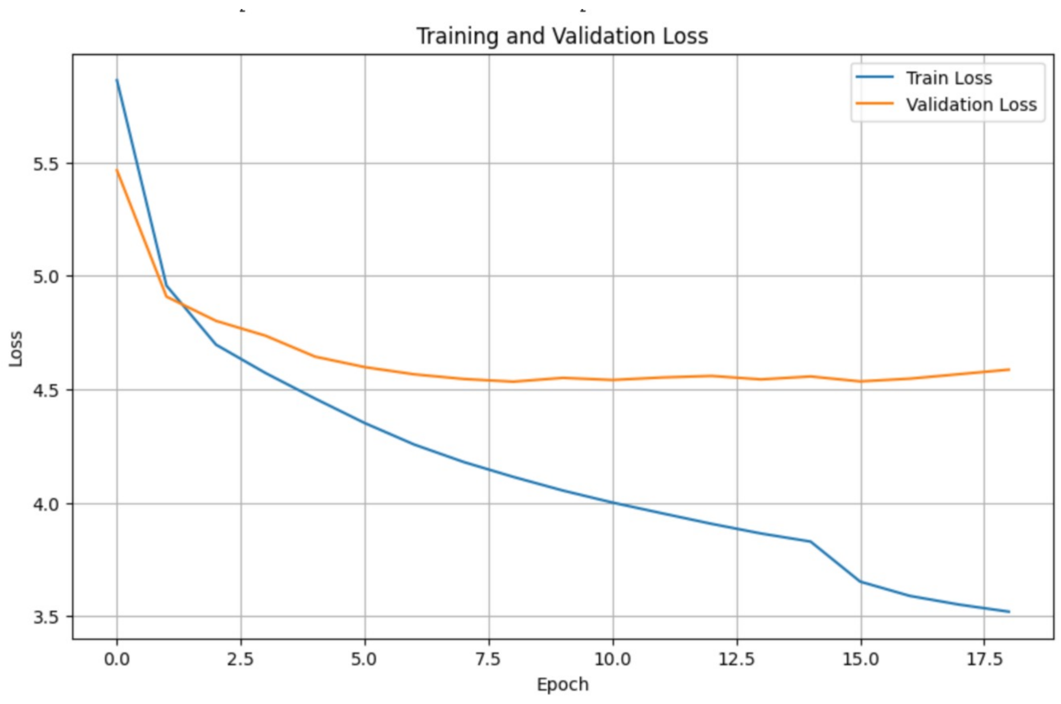

Figure 8.1: **Training Loss for trainable embeddings (with Validation and Early Stopping)**

Chapter 9

CONCLUSION

Based on the observed results from this specific experiment:

- **Trainable Embeddings** offer great flexibility for task-specific learning and achieved high performance (BLEU 0.84), although they required the longest training time (50 epochs).
- **GloVe**, while offering moderate convergence time (40 epochs), yielded significantly lower performance (BLEU 0.18) compared to the other methods in this empathetic dialogue context.
- **FastText** proved highly efficient, providing the quickest convergence (30 epochs) combined with strong overall performance (BLEU 0.82), likely benefiting from its handling of the vocabulary through subword information.

9.1 Improvements

The current model implementation includes several key improvements:

- Uses pre-trained **FastText word embeddings**, which provide strong out-of-vocabulary (OOV) word handling.
- Employs a **word-level tokenizer** to enhance representation granularity.
- Incorporates a **validation loop with early stopping** to prevent overfitting and ensure better generalization.
- **Saves the best model** checkpoint based on validation performance.
- Applies **sampling strategies** such as Top-k and Top-p (nucleus sampling) for improved inference quality.

9.2 Limitations and Future Work

- **Tokenizer:** Using WordPiece is a significant improvement over basic splitting, handling unknown words and morphology better.

- **Training:** Training on the full dataset with validation and early stopping is crucial for better generalization and preventing overfitting. Expect much longer training times.
- **Model Quality:** Even with these improvements, the quality depends heavily on sufficient training time, data quality, and potentially further hyperparameter tuning or model scaling. The responses should be more coherent now, but may still lack deep understanding or perfect empathy.
- **Safety & Evaluation:** Still lacks safety layers and proper evaluation metrics (BLEU [12], ROUGE [13], perplexity, human evaluation). This is critical for any real deployment.
- **Next Steps:** Consider experimenting with learning rate schedulers, different optimizers (AdamW [14]), larger model sizes (if resources allow), or more advanced attention mechanisms. For production-level quality, fine-tuning large pre-trained models is the standard approach.

Author Contributions

Authors

Mohit Sharma (MT2024091)
Parv Gatecha (MT2024108)
Himanshu Shivhare (MT2024058)
Mohit Gupta (MT2024049)

“All authors listed above contributed equally to the conceptualization, design, implementation, experimentation, analysis, and writing of this project report. The work was carried out jointly, and all authors worked together at every stage as part of a fully collaborative effort.”

Bibliography

[1] H. Rashkin, E. M. Smith, M. Li, and Y.-L. Boureau, "Towards Empathetic Open-domain Conversation Models: A New Benchmark and Dataset," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*, Florence, Italy, Jul. 2019, pp. 5370–5380. doi: 10.18653/v1/P19-1534.

[2] Y. Wu et al., "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation," *arXiv preprint arXiv:1609.08144*, 2016. [Online]. Available: https://arxiv.org/abs/1609.08144

[3] T. Wolf et al., "Transformers: State-of-the-Art Natural Language Processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations (EMNLP Demos)*, Online, Oct. 2020, pp. 38–45. doi: 10.18653/v1/2020.emnlp-demos.6.

[4] A. Vaswani et al., "Attention Is All You Need," in *Advances in Neural Information Processing Systems 30 (NIPS 2017)*, I. Guyon et al., Eds. Curran Associates, Inc., 2017, pp. 5998–6008. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[5] J. Pennington, R. Socher, and C. D. Manning, "GloVe: Global Vectors for Word Representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Doha, Qatar, Oct. 2014, pp. 1532–1543. doi: 10.3115/v1/D14-1162.

[6] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching Word Vectors with Subword Information," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, Dec. 2017. doi: 10.1162/tacl_a_00051.

[7] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *arXiv preprint arXiv:1412.6980*, 2014. [Online]. Available: https://arxiv.org/abs/1412.6980 (Conference version: ICLR 2015)

[8] A. Paszke et al., "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32 (NIPS 2019)*, H. Wallach et al., Eds. Curran Associates, Inc., 2019, pp. 8026–8037. [Online]. Available: https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[9] G. Hinton, O. Vinyals, and J. Dean, "Distilling the Knowledge in a Neural Network," *arXiv preprint arXiv:1503.02531*, 2015. [Online]. Available: https://arxiv.org/abs/1503.02531

[10] A. Fan, M. Lewis, and Y. Dauphin, "Hierarchical Neural Story Generation," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Melbourne, Australia, Jul. 2018, pp. 889–898. doi: 10.18653/v1/P18-1082.

[11] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The Curious Case of Neural Text Degeneration," in *International Conference on Learning Representations (ICLR)*, 2020. [Online]. Available: https://openreview.net/forum?id=rygGZiA5Ye

[12] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: a Method for Automatic Evaluation of Machine Translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, Philadelphia, PA, USA, Jul. 2002, pp. 311–318. doi: 10.3115/1073083.1073135.

[13] C.-Y. Lin, "ROUGE: A Package for Automatic Evaluation of Summaries," in *Text Summarization Branches Out: Proceedings of the ACL-04 Workshop*, Barcelona, Spain, Jul. 2004, pp. 74–81. [Online]. Available: https://aclanthology.org/W04-1013/

[14] I. Loshchilov and F. Hutter, "Decoupled Weight Decay Regularization," *arXiv preprint arXiv:1711.05101*, 2017. [Online]. Available: https://arxiv.org/abs/1711.05101 (Conference version: ICLR 2019)