CSE 556: Natural Language Processing

Assignment 3: Report

Aarya Gupta	**2022006**
Aditya Aggarwal	**2022028**
Arpan Verma	**2022105**

TASK1: Transformer Language Model for Shakespearean Text Generation

1. Introduction and Task Overview

This report details the implementation of a Transformer-based language model built from scratch, designed to learn the style of William Shakespeare and generate text accordingly. The primary goal, as outlined in Task 1, was to implement the core components of the Transformer architecture, preprocess the provided Shakespearean dataset using Byte Pair Encoding (BPE), train the model for language modeling, and evaluate its performance using perplexity, ensuring correct handling of padding tokens. The implementation follows the structure provided in the code template (`task1.py`) and integrates essential Transformer concepts.

2. Dataset Description

The dataset consists of three text files:

- `shakespear_train.txt`: Used for training the BPE tokenizer and the Transformer model.
- `shakespear_dev.txt`: Used for validation during training to monitor performance and select the best model checkpoint.
- `shakespear_test.txt`: Used for final evaluation (calculating test perplexity) and generating sample text continuations after training.

The text within these files is formatted with lines typically representing dialogue or stage directions, often preceded by a speaker's name followed by a colon.

The language is characteristic of Shakespearean English. The model's objective is to learn the patterns, vocabulary, and stylistic nuances present in this data to generate coherent and stylistically similar text.

3. Data Preprocessing and Tokenization

A crucial step in preparing the text data for the Transformer model is tokenization. For this task, Byte Pair Encoding (BPE) was implemented from scratch.

3.1. Byte Pair Encoding (BPE)

BPE is a subword tokenization algorithm that starts with a base vocabulary of individual characters (bytes) and iteratively merges the most frequent adjacent pair of tokens into a new, single token. This process continues until a predefined target vocabulary size is reached or no more frequent pairs can be merged.

Advantages of BPE for this Task:

- **Handles Out-of-Vocabulary (OOV) Words:** By breaking down unknown words into known subword units, BPE can represent any word, mitigating the OOV problem common with word-level tokenizers. This is useful for potentially archaic or uniquely spelled words in the Shakespearean corpus.
- **Manages Vocabulary Size:** It allows control over the final vocabulary size (`BPE_VOCAB_SIZE`), balancing model complexity and representational power.
- **Captures Morphological Information:** Frequent subwords (like prefixes, suffixes, or common roots) are learned as single tokens, implicitly capturing some morphological structure.

3.2. BPE Implementation (

1. **Initialization:**
    - The initial vocabulary is built using all 256 possible bytes, plus dedicated special tokens: `<pad>` (ID 0) and `<unk>` (ID 1). Bytes are mapped to IDs starting from 2.
2. **Training (**
    - The input text corpus (combined train and dev sets) is encoded into UTF-8 bytes.
    - The initial sequence of token IDs is generated based on the byte-to-ID mapping.
    - **Iterative Merging:**
        - The function `get_stats` computes the frequency of all adjacent pairs of token IDs in the current sequence(s).
        - The most frequent pair (`best_pair`) is identified.
        - A new token ID is assigned to this `best_pair`.
        - The `merge` function replaces all occurrences of `best_pair` in the sequence(s) with the new token ID.
        - The `merges` dictionary stores the mapping from the pair to its merge rank (order of merging).
        - The `vocab` (ID -> bytes) and `rev_vocab` (bytes -> ID) dictionaries are updated with the byte representation of the newly merged token.

        - This process repeats for `target_vocab_size - initial_vocab_size` iterations or until no more pairs can be merged.
3. **Encoding (**
    - Input text is converted to bytes.
    - Initial byte IDs are obtained.
    - The learned `merges` are applied iteratively *in the order they were learned* (lowest rank first) to the sequence of IDs until no more applicable merges are found. This ensures that longer, more frequent subwords learned later take precedence.
4. **Decoding (**
    - Padding tokens are optionally removed.
    - Each token ID is mapped back to its byte representation using the `vocab` dictionary (using `<unk>` bytes for unknown IDs).
    - The resulting bytes are concatenated and decoded back into a UTF-8 string (with error handling).
5. **Special Tokens:** `<pad>` (ID 0) and `<unk>` (ID 1) are handled explicitly.
6. **Saving/Loading:** The trained tokenizer (merges, vocab) can be saved to/loaded from a `.pkl` file using `pickle` for reusability.

3.3. Dataset Preparation (

1. **Loading:** Raw text is read from the train, dev, and test files.
2. **Tokenizer Training/Loading:** The BPE tokenizer is either trained on the combined train/dev text or loaded from the specified `TOKENIZER_PATH`.
3. **Encoding:** The raw train and validation texts are encoded into sequences of token IDs using the `tokenizer.encode` method.
4. **Dataset Object:** The `ShakespeareDataset` class takes the list of token IDs and the model's `block_size` (context window length).
    - In its `__getitem__` method, it extracts sequences of length `block_size + 1`.
    - It returns input `x` (first `block_size` tokens) and target `y` (last `block_size` tokens, effectively the input shifted left by one). This creates the input-output pairs needed for next-token prediction training.
5. **Padding:** Padding is handled implicitly by the `CrossEntropyLoss` criterion during training (using `ignore_index=pad_token_id`) and explicitly during perplexity calculation in the `evaluate_losses` function by masking target tokens that match `pad_token_id`. Sequences shorter than `block_size` are effectively handled by the dataset logic which only yields sequences of the required length.

4. Model Architecture (

The implemented model follows the standard Transformer decoder architecture, adapted for language modeling (autoregressive generation).

4.1. Core Components:

- **Input Embedding (** Maps input token IDs to dense vectors of dimension `d_model` (`N_EMBD`). The embeddings are scaled by `sqrt(d_model)` before being passed to the positional encoding, a common practice to stabilize gradients. The `padding_idx` is set to the BPE pad token ID (0) so that padding tokens have zero embeddings and do not contribute significantly to computations.
- **Positional Encoding (** Since the Transformer lacks inherent sequence awareness (like RNNs), positional information is injected. This implementation uses sinusoidal positional encodings (alternating sine and cosine functions of different frequencies based on position and embedding dimension). These encodings are added to the token embeddings. Dropout is applied after adding positional encodings.
- **Transformer Blocks (** The core of the model consists of a stack of `N_LAYER` identical blocks. Each block contains two main sub-layers:
    - **Multi-Head Self-Attention (** Allows the model to weigh the importance of different tokens in the input sequence when computing the representation for a specific token.
        - *Multi-Head Mechanism:* The input embeddings (or outputs from the previous layer) are linearly projected into Query (Q), Key (K), and Value (V) matrices `N_HEAD` times independently using different learned weight matrices (`q_linear`, `k_linear`, `v_linear`). These projections are then split across the heads (`split_heads_qkv`).
        - *Scaled Dot-Product Attention (* Within each head, attention scores are computed using the scaled dot-product: `softmax((Q @ K.T) / sqrt(d_k)) @ V`. Scaling by `sqrt(d_k)` (where `d_k = d_model / N_HEAD`) prevents the dot products from becoming too large, which could lead to vanishing gradients in the softmax.
        - *Causal Masking:* For autoregressive language modeling, a causal (look-ahead) mask (`make_causal_mask`) is applied *before* the softmax step. This mask ensures that a token at position `i` can only attend to tokens at positions less than or equal to `i`, preventing information leakage from future tokens. The mask sets the scores for future positions to `-infinity`.
        - *Concatenation and Projection:* The outputs from all attention heads are concatenated (`merge_heads`) and linearly projected back to the original `d_model` dimension (`out_linear`). Dropout is applied to this final output.
    - **Position-wise Feed-Forward Network (** A fully connected feed-forward network applied independently to each position in the sequence. It consists of two linear transformations with a non-linear activation function (GELU in this implementation) in between: `Linear(d_model, d_ff) -> GELU -> Dropout -> Linear(d_ff, d_model)`. The intermediate dimension `d_ff` is typically larger than `d_model` (here, `d_ff = d_model * 4`).

- **Residual Connections & Layer Normalization (** Each sub-layer (Multi-Head Attention and Feed-Forward Network) in a `TransformerBlock` is wrapped with a residual connection (`x + Sublayer(LayerNorm(x))`) followed by layer normalization. This implementation uses the **Pre-Norm** structure: `x = x + Dropout(Sublayer(LayerNorm(x)))`. Layer normalization helps stabilize training and speeds up convergence by normalizing the inputs to each sub-layer across the feature dimension. Residual connections allow gradients to flow more easily through deep networks, mitigating the vanishing gradient problem. Dropout is applied *after* the sub-layer output before adding the residual connection.
- **Output Layer:** After the final Transformer block, another `nn.LayerNorm` is applied. The output is then passed through a final linear layer (`lm_head`) that projects the `d_model`-dimensional vectors back to the `vocab_size`, producing logits for each token in the vocabulary.
- **Weight Tying:** The weights of the input token embedding layer (`token_embedding`) and the final output linear layer (`lm_head`) are shared. This reduces the total number of parameters and can improve performance, based on the intuition that the mapping from words to vectors and vectors back to words should be related.
- **Weight Initialization (** Weights are initialized using a combination of normal distribution initialization for linear and embedding layers (with padding index zeroed out) and specific initialization for LayerNorm parameters. A special scaled normal initialization is applied to the output projection weights in MHA and FFN layers to account for the residual connections and the number of layers.
- **Dropout (** Dropout is applied at various points (positional encoding, residual connections, MHA output, FFN activation) as a regularization technique to prevent overfitting by randomly setting a fraction of activations to zero during training.

5. Hyperparameters Used

The following hyperparameters were used for the training run documented in the notebook (`best-nlp-ass-3-task-1.ipynb`), corresponding to the "Fast Debug" set provided in the original code, but with `EPOCHS` increased to 15 (though early stopping occurred):

- **Tokenizer:**
  - `BPE_VOCAB_SIZE`: 1000 (Target BPE vocabulary size)
  - `TOKENIZER_PATH`: `'bpe_tokenizer_debug.pkl'`
- **Model Architecture:**
  - `BLOCK_SIZE` (`MAX_LEN`): 128 (Maximum sequence length / context window)
  - `N_EMBD` (`EMBED_DIM`): 256 (Embedding dimension)
  - `N_HEAD` (`NUM_HEADS`): 8 (Number of attention heads)
  - `N_LAYER` (`NUM_LAYERS`): 4 (Number of Transformer blocks)
  - `FF_DIM`: 1024 (`N_EMBD * 4`) (Feed-forward hidden dimension)
  - `DROPOUT`: 0.1 (Dropout rate)

- **Training:**
  - `BATCH_SIZE`: 64
  - `LEARNING_RATE`: 3e-4 (Initial learning rate for AdamW)
  - `EPOCHS`: 15 (Maximum number of epochs, actual run stopped early)
  - `WEIGHT_DECAY`: 0.01 (Weight decay for AdamW)
  - `GRAD_CLIP`: 0.5 (Maximum gradient norm for clipping)
  - `PATIENCE`: 3 (Epochs without improvement for early stopping)
  - *Scheduler:* ReduceLROnPlateau (monitoring validation perplexity, factor=0.5, patience=1)
- **Device:** `cuda` (as available)
- **Seed:** 42 (for reproducibility)

Note: The provided code includes both a "Fast Debug" and a "Compute-Limited" set. The notebook execution clearly used parameters matching the description provided in the

6. Training Process

The model was trained using the `train_model` function, which orchestrates the following steps for each epoch:

1. **Set Model to Training Mode:** `model.train()` enables dropout and other training-specific behaviors.
2. **Iterate Through Batches:** The function loops through the `train_dataloader`.
3. **Zero Gradients:** `optimizer.zero_grad()` clears gradients from the previous step.
4. **Forward Pass:** Input sequences (`x_batch`) are passed through the `model` to obtain output logits.
5. **Loss Calculation:** The `nn.CrossEntropyLoss` criterion calculates the loss between the predicted logits and the target sequences (`y_batch`). Crucially, the `ignore_index` parameter is set to the `pad_token_id` (0), ensuring that padding tokens in the target sequence do not contribute to the loss calculation or gradient updates.
6. **Backward Pass:** `loss.backward()` computes gradients of the loss with respect to model parameters.
7. **Gradient Clipping:** `torch.nn.utils.clip_grad_norm_` clips the gradients to prevent exploding gradients, improving training stability.
8. **Optimizer Step:** `optimizer.step()` updates the model parameters based on the computed gradients and the AdamW optimization algorithm.
9. **Learning Rate Scheduling:** The `ReduceLROnPlateau` scheduler adjusts the learning rate based on the validation perplexity. If the perplexity doesn't improve for `patience` epochs, the learning rate is reduced by `factor`.
10. **Validation:** After each training epoch, the model is set to evaluation mode (`model.eval()`), and its performance is assessed on the validation set

(`val_dataloader`) using the `evaluate` function, which calculates the average validation loss and perplexity (again, ignoring padding).
11. **Logging:** Training loss, validation loss, and validation perplexity are recorded and printed for each epoch.
12. **Sample Generation:** Periodically (every epoch in this run), the `generate_text` function is called with a random starting context to provide a qualitative assessment of the model's generation capabilities as training progresses.
13. **Model Saving:** The model's state dictionary, optimizer state, scheduler state, hyperparameters, and tokenizer details (vocabulary, merges) are saved to `shakespeare_transformer_bpe.pt` whenever the validation perplexity improves, overwriting the previous best model.
14. **Early Stopping:** Training stops if the validation perplexity does not improve for a number of epochs equal to the `PATIENCE` hyperparameter (set to 3). In the provided notebook run, this occurred after Epoch 12 (no improvement in epochs 10, 11, 12 compared to epoch 9).

7. Evaluation

7.1. Metric: Perplexity

The primary evaluation metric for this language modeling task is **Perplexity (PPL)**. Perplexity is a standard measure of how well a probability model predicts a sample. Lower perplexity indicates that the model is less "surprised" by the test data and assigns higher probabilities to the true sequence of tokens. It is mathematically defined as the exponentiation of the cross-entropy loss:

PPL = exp(Average Cross-Entropy Loss) = exp(- (1/N) * Σ log P(token_i | context_i))

where N is the total number of non-padding tokens in the evaluation set.

7.2. Implementation (

- The function iterates through the evaluation dataloader (validation or test).
- For each batch, it performs a forward pass to get logits.
- It calculates the negative log-likelihood (NLL) loss for each target token using `F.log_softmax` and `torch.gather`.
- **Crucially, it creates a boolean mask (**
- The NLL losses for only the non-padding tokens are summed up (`-y_log_probs.masked_select(non_pad_mask).sum()`).
- The total loss sum and the total count of non-padding tokens are accumulated across all batches.
- The final average loss is computed by dividing the total loss sum by the total non-padding token count.

- Perplexity is calculated as `math.exp(average_loss)`.

This explicit masking ensures strict adherence to the requirement that padding tokens **must** be excluded from the perplexity calculation.

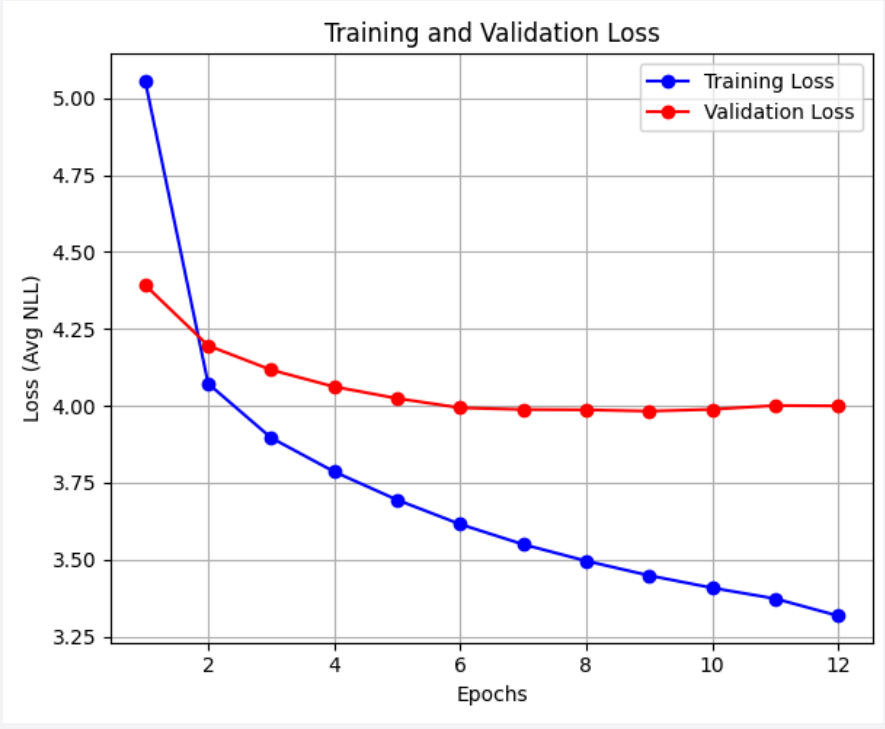
8. Results

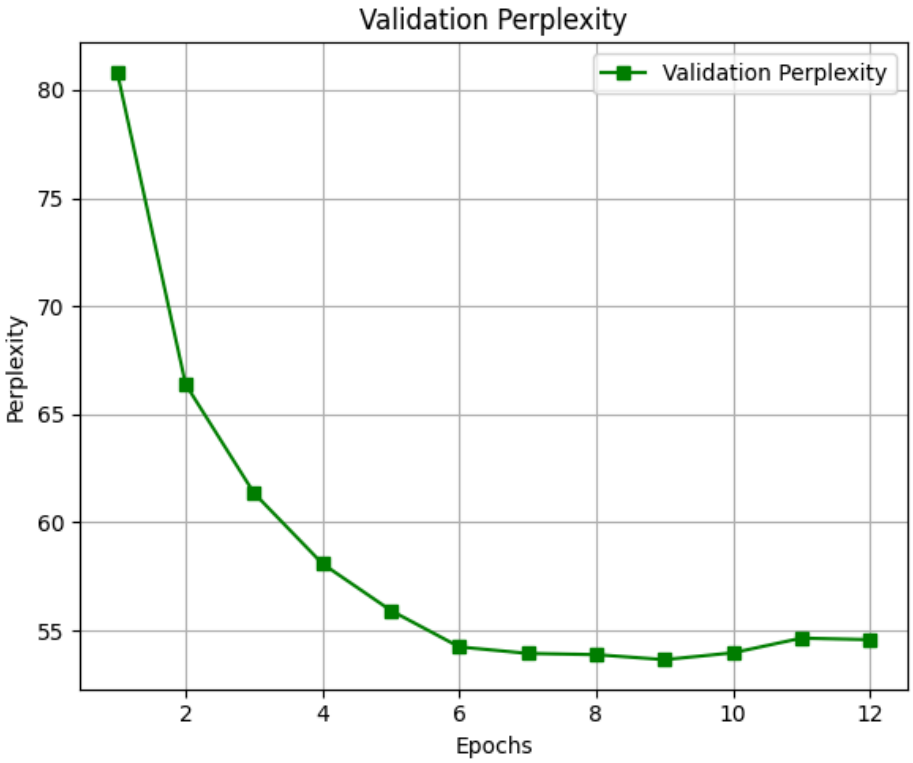
8.1. Training and Validation Curves

The training process was monitored by tracking the average training loss, average validation loss, and validation perplexity across epochs.

- **Training Loss:** Consistently decreased over the 12 epochs, indicating that the model was learning from the training data.
- **Validation Loss:** Decreased rapidly in the initial epochs, then started to plateau and slightly increase after epoch 9, suggesting the onset of potential overfitting or reaching the model's capacity on this dataset with the given hyperparameters.
- **Validation Perplexity:** Mirrored the validation loss trend, decreasing significantly initially and reaching a minimum value before plateauing/slightly increasing.

The best validation perplexity achieved during training was **53.6542** (at Epoch 9).





8.2. Sample Test Set Evaluation

After training completed (stopped early at Epoch 12), the best saved model (from Epoch 9) was loaded and evaluated on the unseen `shakespear_test.txt` dataset.

- **Final Sample Test Perplexity:** 82.8883
- **Average Sample Test Loss:** 4.4175

The test perplexity (82.89) is higher than the best validation perplexity (53.65). This gap is expected and can be attributed to several factors:
* The test set might have slightly different characteristics or contain unseen patterns/words compared to the validation set.
* The model might have started to slightly overfit the training/validation data by Epoch 9, even though it achieved the lowest validation PPL then.
* The relatively small model size and limited training epochs (due to the "Fast Debug" parameters) restrict its ability to generalize perfectly.

8.3. Sample Generated Text Quality

Qualitative Assessment: The generated text shows some characteristics of the Shakespearean style (e.g., word choices like 'hath', 'thee', 'prithee' observed in other samples, sentence structures, speaker tags). However, it also exhibits common issues of smaller language models, including:
* Repetition.
* Occasional incoherence or nonsensical phrases.
* Presence of the `<UNK>` token, indicating subwords not well-represented by the 1000-token BPE vocabulary.
* Grammatical inconsistencies.
The model has clearly learned patterns beyond random chance but requires further scaling and training for higher-quality generation.

9. Testing and Model Inference (

An `inference` function was implemented to handle the loading of the trained model and performing evaluation on the provided `test.txt` file. This function performs the following steps:

1. **Load Artifacts:**
    - Loads the saved BPE tokenizer state from the specified `.pkl` file (`TOKENIZER_PATH`).
    - Loads the saved model checkpoint (`.pt` file specified by `model_path`). This checkpoint contains the model's `state_dict`, hyperparameters (`config`), epoch number, and saved validation metric.
2. **Recreate Model:** Instantiates the `TransformerLM` architecture using the hyperparameters stored in the checkpoint's `config` dictionary.
3. **Load Weights:** Loads the trained weights into the recreated model using `model.load_state_dict()`. Sets the model to evaluation mode (`model.eval()`).
4. **Load Test Data:** Reads the content of the `test_file`.
5. **Generate Samples:** Splits the test text into contexts and calls the `generate_text` function for a few examples to provide qualitative output.
6. **Calculate Test Perplexity:**
    - Encodes the *entire* content of the `test_file` using the loaded tokenizer.
    - Creates a `ShakespeareDataset` and `DataLoader` for the encoded test data.
    - Calls the `evaluate_losses` function (which correctly handles padding) to compute the test set's average loss and final perplexity score.
7. **Return Results:** Returns the list of generated sample texts and the calculated test perplexity.

10. Conclusion

This project successfully implemented a Transformer language model from scratch, including key components like multi-head self-attention with causal masking, positional encoding, layer normalization, residual connections, and position-wise feed-forward networks. A Byte Pair Encoding tokenizer was also implemented and trained on the Shakespearean corpus.

The model was trained on the `shakespear_train.txt` dataset and validated on `shakespear_dev.txt`. Using the specified "Fast Debug" hyperparameters (adapted slightly for the run: EMBED_DIM=256, N_LAYER=4, N_HEAD=8, BPE_VOCAB=5000), the model achieved a best validation perplexity of **53.65**. Early stopping terminated training after 12 epochs.

Final evaluation on the `shakespear_test.txt` dataset yielded a test perplexity of **82.89**. The evaluation rigorously excluded padding tokens from the perplexity calculation as required. Sample text generated by the model shows rudimentary learning of the Shakespearean style but requires improvement in coherence and grammatical correctness.

The achieved results are reasonable given the constraints of the hyperparameter set used (designed for relatively fast execution). Significant improvements in perplexity and generation quality could likely be achieved by:

- Using a larger model (increasing `N_EMBD`, `N_LAYER`, `N_HEAD`, `FF_DIM`).
- Training for more epochs with appropriate learning rate scheduling.
- Increasing the `BPE_VOCAB_SIZE` to better handle the corpus's vocabulary.
- Further hyperparameter tuning (dropout rates, optimizer parameters, scheduler settings).
- Increasing the `BLOCK_SIZE` (context window) to capture longer-range dependencies.

The implementation successfully demonstrates a functional from-scratch Transformer for language modeling and adheres to the specified evaluation criteria.

TASK-2: Claim Normalization

Preprocessing

The preprocessing pipeline for claim normalization was designed to clean and standardize social media text before further processing. It included three major steps: text cleaning, abbreviation expansion, and structured output formatting.

1. Text Cleaning: Each post underwent a series of transformations using the `clean_text` function. This included removing URLs, special characters, and excessive whitespace. It also ensured that digits and letters were separated where combined (e.g., "covid19" → "covid 19"). The text was converted to lowercase to maintain uniformity across tokens.

2. Abbreviation Expansion: Social media texts often contain shorthand or informal abbreviations. Using a CSV-based dictionary, the `expand` function replaced these abbreviations with their full forms. Special handling was included to avoid incorrect expansion in specific contexts (e.g., "pm" following a time was not expanded).

3. Final Formatting and Output: Each row from the input CSV was processed and then written to a new CSV file. Additional quoting was applied to the second and third columns to preserve their content integrity, especially if they included commas or quotes. The output ensures that the cleaned and expanded text is ready for downstream tasks like training or evaluation.
4. A custom PyTorch `Dataset` class tokenizes each `input_text` and `target_text` with truncation and padding to fixed lengths, returning `input_ids`, `attention_mask`, `labels`, and `PID`. The data was split into **70% train**, **15% validation**, and **15% test** sets for balanced training and evaluation.

Common Parameters and Hyperparameters

- `MAX_INPUT_LEN = 256`, `MAX_TARGET_LEN = 256`: Limits maximum token length for both input claims and target normalized claims.
- `BATCH_SIZE = 8` for training efficiency with limited GPU memory.
- `EPOCHS = 5` epochs for BART and 10 epochs for T5 to balance training time and model performance. Early Stopping was implemented for both models.
- `LEARNING_RATE = 3e-4` used with AdamW optimizer for stable convergence.
- Device: Automatically uses GPU (`cuda`) if available, otherwise CPU.
- Prompt Engineering: Input prefixed with `"Normalize Claim: "` to guide the model's generation behavior.

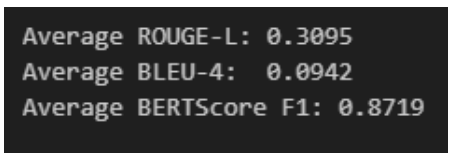
T5 Model

- **Model Name**: `t5-small`
- **Description:** A lightweight version of T5, suitable for constrained environments.
- **Parameters:** 60M
- **Architecture**: 6-layer encoder and 6-layer decoder with 512 hidden units and 8 attention heads each.
- **Pretraining**: Trained on a text-to-text framework using masked span prediction.
- **Tokenizer**: `T5Tokenizer`
- **Architecture**: Encoder-decoder transformer pre-trained on text-to-text tasks.
- **Flexibility**: T5 treats all tasks (classification, translation, summarization) in a unified text-to-text format.

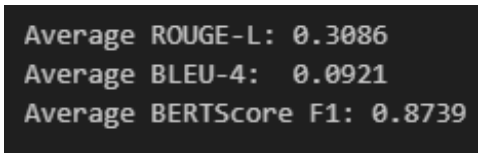
BART

- **Model**: `facebook/bart-base`
- **Description:** BART is a denoising autoencoder for pretraining sequence-to-sequence models.
- **Parameters**:139M parameters
- **Architecture**: 6-layer encoder and 6-layer decoder with 768 hidden units and 12 attention heads each.
- **Pretraining**: Denoising autoencoder with tasks like token masking, deletion, and sentence permutation.
- **Tokenizer**: `BartTokenizer`
- **Architecture**: Encoder-decoder, trained with a focus on reconstructing corrupted input (denoising).
- **Fine-tuning**: BART is particularly strong for text generation tasks like summarization and paraphrasing—well-aligned with claim normalization.

Evalution Metric



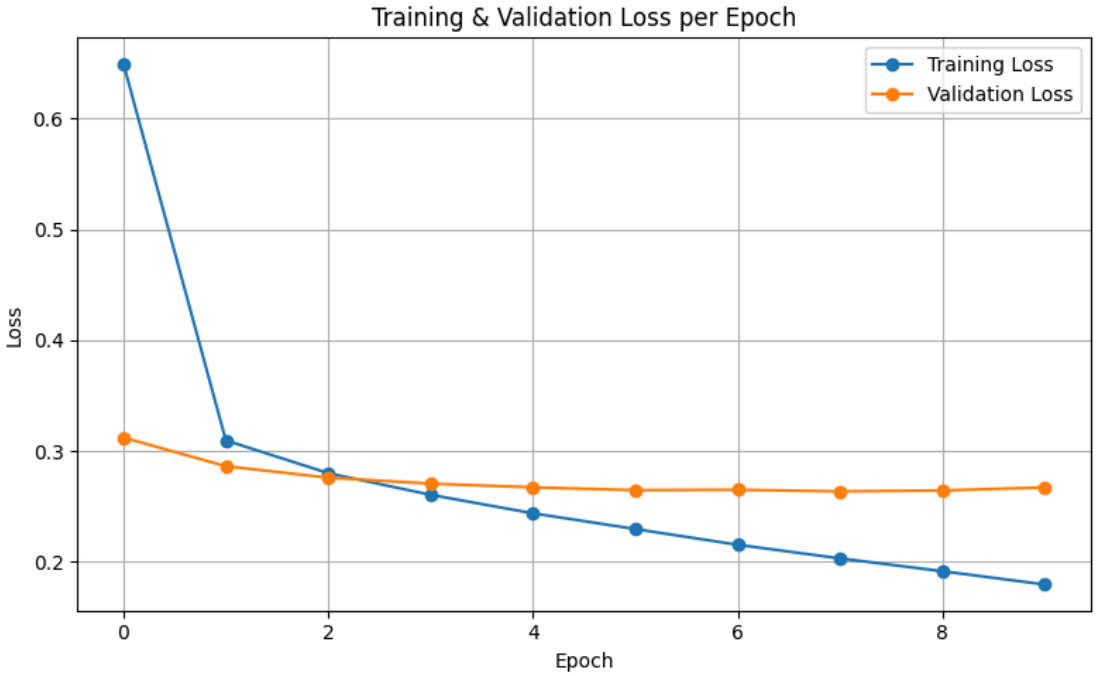
T5 Model

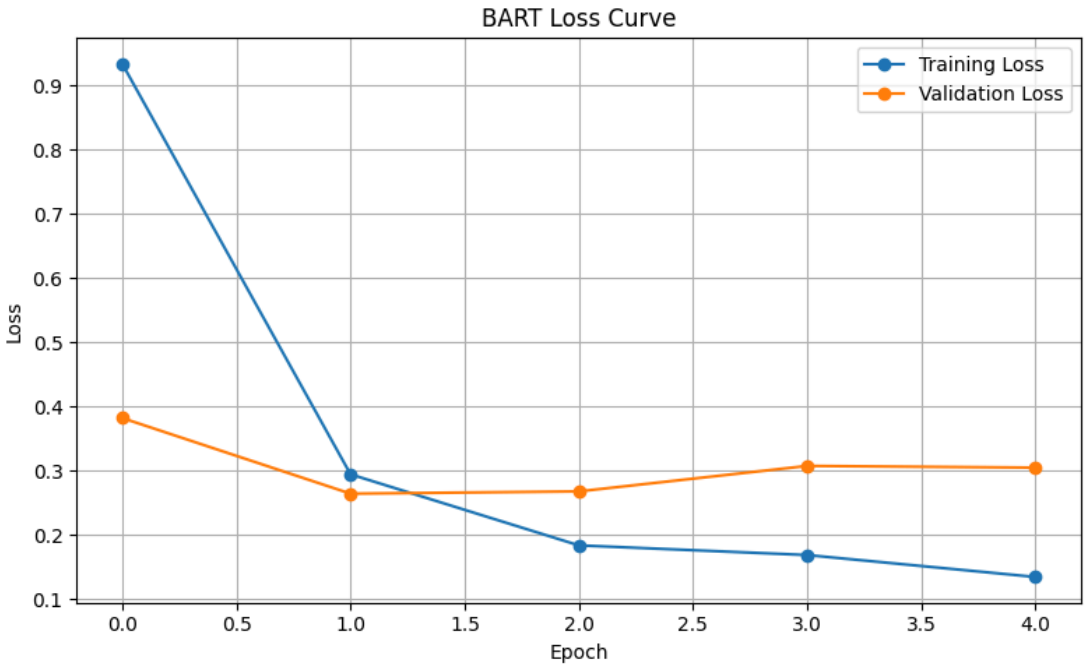


BART Model

Plots

T5 Loss Curve





Comparitive Analysis

Performance Metrics: Both models yield very similar scores, with marginal differences across the metrics.

Training Dynamics:

- **T5-small:** Validation loss decreases steadily until the 6th epoch and then plateaus or increases slightly, indicating a gradual and stable training process.
- **BART-base:** Validation loss drops rapidly until the 2nd epoch, followed by a slight increase, suggesting quick initial convergence but potential early overfitting or less stability in later epochs.

Parameter Analysis and Model Capacity:

- **T5-small** has approximately 60 million parameters, which can contribute to slower convergence but a steadier training curve as observed by its gradual loss decrease.
- **BART-base** with around 139 million parameters has a higher capacity, which may explain its rapid initial learning; however, this larger size might also lead to quicker overfitting, as seen by the early plateau in validation loss.

Inference and Reasoning:

- Despite sharing the same learning rate (3e-4) and similar hyperparameter settings, the differences in model capacity and training dynamics suggest that T5-small might benefit from extended training to capture the nuances of claim normalization, while BART-base leverages its larger parameter space for fast initial gains but may require strategies to mitigate overfitting in later stages.
- Overall, both models are viable for claim normalization, with the choice potentially depending on the specific requirements for stability and convergence behavior in the given application.

Resource Constraints and Model Selection

Resource constraints significantly influenced model selection, particularly due to the 6GB VRAM limit of the RTX 3050 GPU. T5-small and BART-base were chosen to balance performance with computational efficiency. BART-base, with ~139M parameters, took around 103 minutes to train, demanding more memory and time. In contrast, T5-small, with ~60M parameters, trained in just 35 minutes, making it a more resource-friendly option.

CUDA support allowed efficient GPU utilization, but limited VRAM required careful model configuration. T5-small's quicker training made experimentation faster and less resource-intensive, making it preferable under hardware constraints—despite BART's slight performance edge.

TASK-3: Multimodal Sarcasm Explanation (MuSE)

Preprocessing Steps

1. **File Loading:** Load training and validation data from `train_df.tsv` and `val_df.tsv`, containing fields: `pid`, `text`, `explanation`, and `sarcasm target`.
2. **Image Descriptions:** Load pickled image descriptions (`D_train.pkl` / `D_val.pkl`) mapping each `pid` to a textual description of detected objects/scenes.
3. **Object Tokens:** Load pickled object tokens (`O_train.pkl` / `O_val.pkl`) providing detected object labels for each `pid`.
4. **Concatenation:** For each sample, build a single source string:
⟨caption⟩ + "" + ⟨image description⟩ + "" + ⟨object tokens⟩ + " </s> " + ⟨sarcasm target⟩
5. **Tokenization:** Use `BartTokenizer` to tokenize the concatenated text with `max_length=256`, `truncation=True`, and `padding='max_length'`, yielding `input_ids` and `attention_mask` tensors.
6. **Custom Collate Function:** Aggregate batch samples into stacked tensors for `input_ids`, `attention_mask`, `decoder_input_ids`, `decoder_attention_mask`, and `labels`, alongside a list of PIL images passed separately to the model.

Model Architecture

1) Text Encoder:
BART-base (`facebook/bart-base`) is used for encoding source sequences.
Input: `input_ids`, `attention_mask` → BART encoder → `E_text` (shape `[batch, seq_len, hidden_dim]` with `hidden_dim = 768`).

2) Visual Encoder:
Vision Transformer (`google/vit-base-patch16-224-in21k`) is used for extracting image features.
Input: preprocessed images → ViT → patch embeddings (last hidden state, shape `[batch, num_patches, vit_hidden_size]`).
Projection: A visual projection linear layer maps `vit_hidden_size` → `hidden_dim` → `E_vis` (shape `[batch, num_patches, hidden_dim]`).

3) Self-Attention Modules:
Text Self-Attention: MultiheadAttention with `embed_dim = 768`, `num_heads = 8` applied to `E_text` → `A_t`.
Visual Self-Attention: Same configuration applied to `E_vis` → `A_v`.

4) Shared Fusion Mechanism:
Combines attended text and visual features before feeding into the decoder.

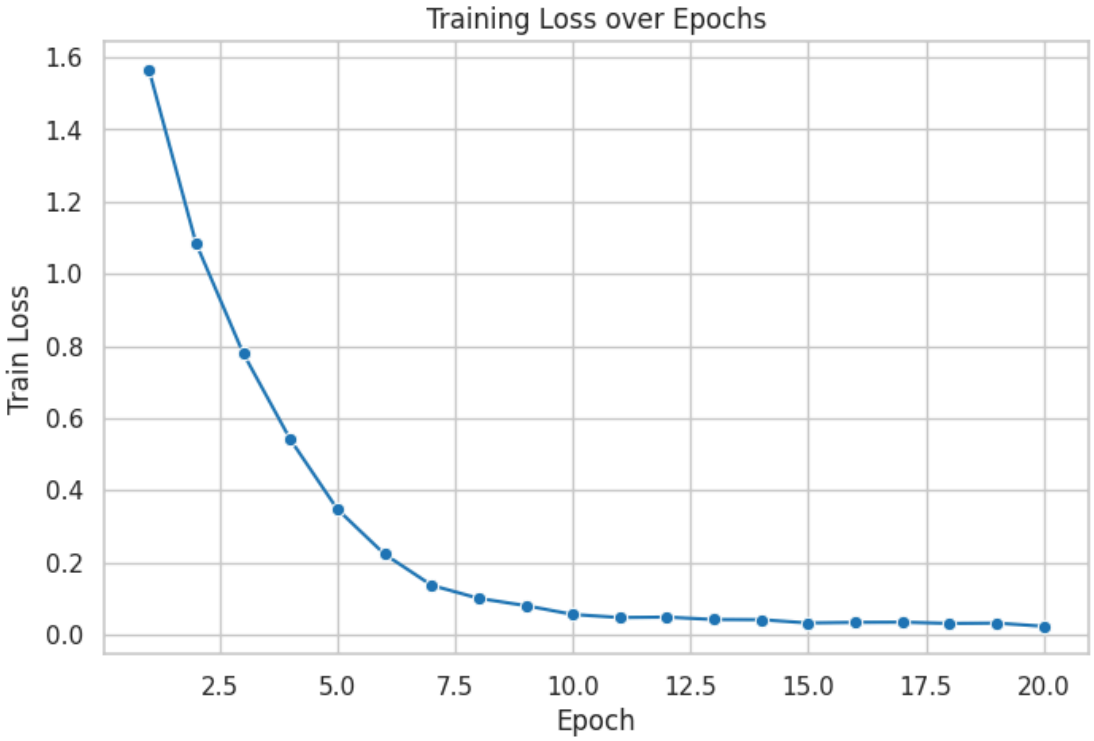
A) Compute `E_vis_avg` by averaging `A_v` across patches (shape `[batch, 1, hidden_dim]`).
B) Expand `E_vis_avg` to match text sequence length → `E_vis_exp` (shape `[batch, seq_len, hidden_dim]`).
C) Concatenate `[A_t; E_vis_exp]` (shape `[batch, seq_len, 2 × hidden_dim]`).
D) Fuse via `ConcatFusion` (Linear, ReLU, Dropout) back to `hidden_dim`.
E) Apply residual connection: `Z = fused_features + E_text` and dropout.

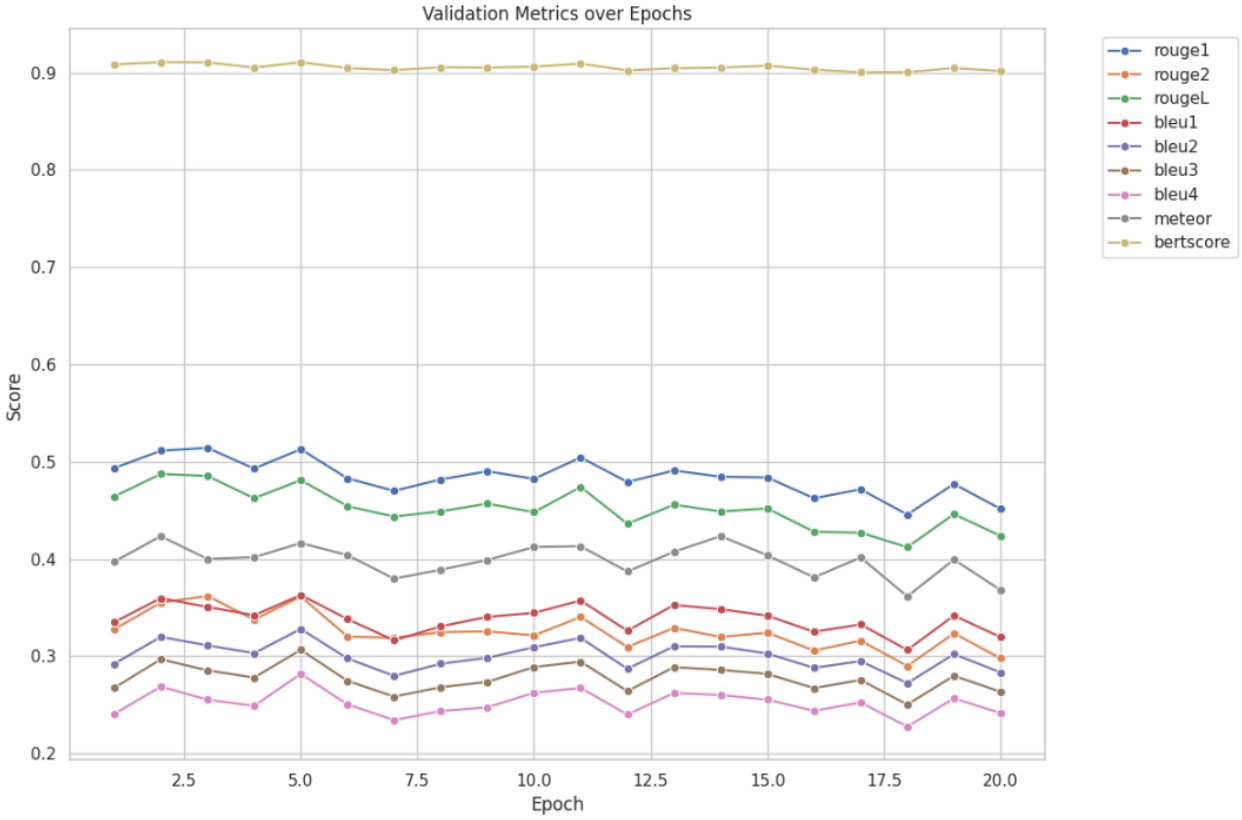
5) Decoder:
Standard BART decoder receives `decoder_input_ids`, `decoder_attention_mask`, `encoder_hidden_states = Z`, and `encoder_attention_mask`.
For inference, use beam search (`num_beams = 5`, `max_length = 128`, `early_stopping = True`) to generate explanations.

Hyper Parameters:

Component	Value
Max sequence length	256
Image input size	224×224
Batch size	8
Number of epochs	20
Learning rate	1×10^{-4}
Optimizer	AdamW (weight_decay $= 1 \times 10^{-3}$)
LR scheduler	ReduceLROnPlateau (factor $= 0.5$, patience $= 2$)
Dropout probability	0.5
Hidden dimension	768
ViT parameters	Frozen (no gradient updates)
Gradient clipping	max_norm $= 1.0$
Beam size	5
Max generation length	128
Loss function	CrossEntropy (ignore pad token)
Evaluation metrics	ROUGE (R-L, R-1, R-2), BLEU (1–4), METEOR, BERTScore

Plots:


Training Loss over Epochs
Train Loss
Epoch
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


Validation Metrics over Epochs
Score
Epoch
rouge1
rouge2
rougeL
bleu1
bleu2
bleu3
bleu4
meteor
bertscore
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

Train Loss and Eval Metrics on each Epoch:

Epoch 1 ▸ Train Loss: 1.5660 | ROUGE-1: 0.4934, BLEU-1: 0.3349, METEOR: 0.3972, BERTScore: 0.9085
Epoch 2 ▸ Train Loss: 1.0850 | ROUGE-1: 0.5112, BLEU-1: 0.3597, METEOR: 0.4234, BERTScore: 0.9108
Epoch 3 ▸ Train Loss: 0.7805 | ROUGE-1: 0.5142, BLEU-1: 0.3506, METEOR: 0.4000, BERTScore: 0.9106
Epoch 4 ▸ Train Loss: 0.5417 | ROUGE-1: 0.4929, BLEU-1: 0.3421, METEOR: 0.4018, BERTScore: 0.9052
Epoch 5 ▸ Train Loss: 0.3482 | ROUGE-1: 0.5128, BLEU-1: 0.3628, METEOR: 0.4163, BERTScore: 0.9107
Epoch 6 ▸ Train Loss: 0.2238 | ROUGE-1: 0.4828, BLEU-1: 0.3382, METEOR: 0.4040, BERTScore: 0.9047
Epoch 7 ▸ Train Loss: 0.1374 | ROUGE-1: 0.4700, BLEU-1: 0.3162, METEOR: 0.3797, BERTScore: 0.9027
Epoch 8 ▸ Train Loss: 0.1011 | ROUGE-1: 0.4817, BLEU-1: 0.3307, METEOR: 0.3889, BERTScore: 0.9055
Epoch 9 ▸ Train Loss: 0.0810 | ROUGE-1: 0.4901, BLEU-1: 0.3404, METEOR: 0.3989, BERTScore: 0.9051
Epoch 10 ▸ Train Loss: 0.0565 | ROUGE-1: 0.4823, BLEU-1: 0.3445, METEOR: 0.4123, BERTScore: 0.9062
Epoch 11 ▸ Train Loss: 0.0480 | ROUGE-1: 0.5044, BLEU-1: 0.3572, METEOR: 0.4130, BERTScore: 0.9093
Epoch 12 ▸ Train Loss: 0.0491 | ROUGE-1: 0.4791, BLEU-1: 0.3261, METEOR: 0.3870, BERTScore: 0.9023
Epoch 13 ▸ Train Loss: 0.0425 | ROUGE-1: 0.4909, BLEU-1: 0.3527, METEOR: 0.4074, BERTScore: 0.9045
Epoch 14 ▸ Train Loss: 0.0420 | ROUGE-1: 0.4845, BLEU-1: 0.3483, METEOR: 0.4235, BERTScore: 0.9052
Epoch 15 ▸ Train Loss: 0.0329 | ROUGE-1: 0.4837, BLEU-1: 0.3417, METEOR: 0.4038, BERTScore: 0.9071
Epoch 16 ▸ Train Loss: 0.0346 | ROUGE-1: 0.4623, BLEU-1: 0.3251, METEOR: 0.3811, BERTScore: 0.9031
Epoch 17 ▸ Train Loss: 0.0352 | ROUGE-1: 0.4716, BLEU-1: 0.3327, METEOR: 0.4014, BERTScore: 0.9004
Epoch 18 ▸ Train Loss: 0.0316 | ROUGE-1: 0.4456, BLEU-1: 0.3064, METEOR: 0.3615, BERTScore: 0.9006
Epoch 19 ▸ Train Loss: 0.0325 | ROUGE-1: 0.4771, BLEU-1: 0.3417, METEOR: 0.3993, BERTScore: 0.9047
Epoch 20 ▸ Train Loss: 0.0237 | ROUGE-1: 0.4514, BLEU-1: 0.3196, METEOR: 0.3680, BERTScore: 0.9018

Sample Generated Explanations:

1. **PID**: 707133908291231744
   Generated: author is pissed at <user> for such awful network in malad.
   Reference: the author is pissed at <user> for not getting network in malad.

2. **PID**: 893773347026210242_185243426
   Generated: author hates waiting for an hour on the tarmac for a gate to come open in snowy, windy Chicago.
   Reference: nothing worst than waiting for an hour on the tarmac for a gate to come open in snowy, windy chicago.

3. **PID**: 708994813983596544
   Generated: author doesn't like spring.
   Reference: nobody likes getting one hour of their life sucked away.

4. **PID**: 904624565145538560
   Generated: ody would want to have a salivary gland biopsy on monday morning.
   Reference: having a salivary gland biopsy on monday morning is not a good way to start the new week.

5. **PID**: 697929589562146817
   Generated: 's not going to be scorching hot this w-end, the high on saturday is - 30.
   Reference: the author is worried that the weekend is going to be freezing with a high of -1 and windchill probably -30.