ASSIGNMENT - 3

CSE 556 - Natural Language Processing

Akshat Parmar

Rishi Pendyala

Vimal Jayant Subburaj

6.4.2025

Winter 2025

TASK 1

Lead : Akshat Parmar (2022050)

Report: Implement Transformer from Scratch

1. Preprocessing Steps

Tokenization:

- Each sentence is split into tokens based on spaces (i.e., whitespace tokenization).
- A special `<START>` token is added to the beginning, and `<STOP>` to the end of the token sequence.
- Unknown tokens are replaced with the `<UNK>` token.

Vocabulary Construction:

- A `Counter` is used to count the frequency of all tokens in the training data.
- A tokenizer dictionary is created by combining special tokens (`<PAD>`, `<START>`, `<STOP>`, `<UNK>`) with the most common tokens from the dataset.
- Each token is assigned a unique index, forming the basis for token-to-id conversion.
- An inverse tokenizer is also created for id-to-token mapping (used during decoding).

Padding:

- All token sequences are padded to a fixed maximum length (`MAX_LEN = 256`) using the `<PAD>` token.
- Padding is applied post-tokenization to ensure that all input sequences are of uniform length.

Embedding Lookup:

- Each token is mapped to its corresponding embedding vector using pre-trained embeddings (GloVe or FastText).
- If a token is not found in the embedding vocabulary, it is mapped to a zero vector.

Label Preparation for Language Modeling:

- For training, input sequences (X_tokens) are the original padded tokens excluding

the last token.
- Target sequences (y_tokens) are the same as the input, shifted one position to the left (i.e., predicting the next word).
- The loss function (CrossEntropyLoss) is set to ignore padding tokens during loss calculation (via ignore_index=tokenizer["<PAD>"]).

Dataset and DataLoader Preparation:

- TextDataset class wraps the text data and applies tokenization and padding during sample retrieval (__getitem__).
- PyTorch DataLoader is used to load batches of tokenized and padded sequences for training and validation.

Text Generation Preprocessing:

- During generation, the context string is tokenized without the <STOP> token.
- Tokens are sequentially appended based on model predictions until the <STOP> token is generated or the desired length is reached.
- The resulting token sequence is decoded back to text, excluding special tokens (<PAD>, <START>, <STOP>, <UNK>).

2. Transformer Language Model

TransformerLM Class

Embedding Layer:

- Embeds input tokens using a learnable embedding matrix of size `(vocab_size, d_model)`.
- Embeddings are scaled by √d_model.

Positional Encoding:

- Added to embeddings to inject sequence order information.
- Sinusoidal functions are used.
- Even indices: sin(position / (10000^(2i/d_model)))
- Odd indices: cos(position / (10000^(2i/d_model)))

Dropout Layer:

- Applied after positional encoding to prevent overfitting.

Transformer Blocks:

- A list of n_layers Transformer blocks (defined below), each consisting of self-attention and feedforward layers.

Output Layer:

- Linear layer projecting hidden states back to vocabulary size for token prediction.

Weight Initialization:

- Xavier uniform initialization for all weight matrices with more than one dimension.

Forward Pass:

- Input is embedded and combined with positional encoding.
- A causal mask is applied to ensure auto-regressive prediction (no access to future tokens).
- Output logits and self-attention weights from each block are returned.

TransformerBlock Class

Multi-Head Self-Attention:

- Attention is computed over the input sequence with a mask to preserve causality.

Layer Normalization:

- Applied after attention and feedforward sublayers using residual connections.

Feed Forward Network:

- Two linear layers with ReLU activation and dropout in between.

MultiHeadAttention Class

Linear Projections:

- Input is projected into query, key, and value representations using linear layers.

Head Splitting and Merging:

- Each head receives a slice of the projected input (`d_k = d_model / n_heads`).
- After attention calculation, heads are concatenated and passed through another

linear layer.

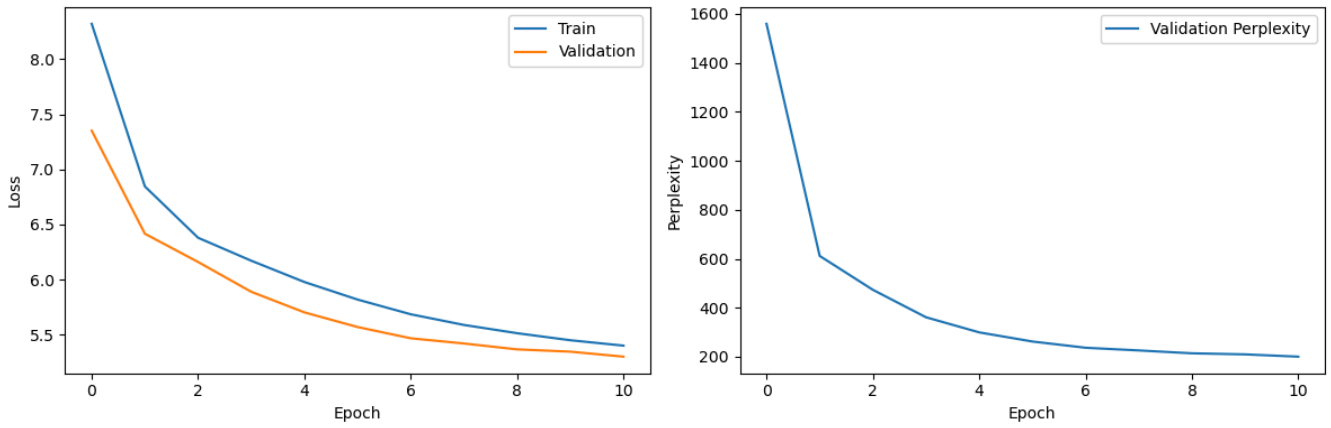
Attention Computation:

- Self-attention is computed using scaled dot-product attention with an optional mask.
- Attention weights and attended values are returned.

3. Hyperparameters

```
#config
d_model =64 #embedding dimension
n_heads = 4 #number of heads
n_layers = 6 #number of layers
dropout = 0.1 #dropout rate
num_epochs = 11
lr = 1e-4
```

4. Training and Validation Loss Plots



4. Performance

```
Model Perplexity: 434.70

Generated Text Samples:
First RICHARD : The king , he thy city .
HENRY RICHARD : Why , and the preserved thy king
KING II : IV : And , I the day
KING YORK RICHARD : I the father in he ,
GLOUCESTER : I , I thou , I more me
BRUTUS : A father , your father , my lord
First RICHARD : I not , thou their cousin ,
Thou crown , I this , Should thine , I
SICINIUS : I not thee .
KING RICHARD WARWICK : The world out , thy heart
GLOUCESTER : O , what , I that the lord
First RICHARD : Why , you have a king is
GLOUCESTER : What , you 's 'll , I ,
And he is the neck , that the grace ,
KING VI : O , your father , I :
First RICHARD : The lord , my lord , my
QUEEN OF : OF RICHARD : I honour 's this
KING HENRY YORK : : But I lives , the
MENENIUS : Therefore , be , I your lord ,
```

Key Observations

The model output shows that it has learned the structural format of Shakespearean dialogue, with character names followed by lines of speech using appropriate vocabulary like "thy" and "lord." However, the generated sentences often lack grammatical correctness and coherence, with noticeable repetition and awkward phrasing. While character consistency appears, the high perplexity (434.70) suggests the model is still struggling to make confident predictions, indicating it may be undertrained or has limited capacity.

TASK 2

Lead : Vimal Jayant Subburaju (2022571)

Claim Normalization

Preprocessing:

The preprocessing is done in the following steps:

- **Type Check:** **If text is not a string, it returns the input as is.**
- **Expand Contractions:** **It uses contractions.fix(text) to replace contractions (e.g., "can't" → "cannot").**
- **Lowercasing:** **Converts all text to lowercase.**
- **Abbreviation Expansion:** **Uses abb_dict, a predefined dictionary, to replace abbreviations with their full forms.**
- **Remove URLs:** **Eliminates web links (http://, https://, or www.).**
- **Remove Special Characters:** **Deletes non-alphanumeric characters except spaces.**
- **Remove Extra Whitespace:** **Ensures single spaces between words and trims leading/trailing spaces.**

From kaggle I had found out a dataset(csv) file that contains the most common 150 abbreviations and its full form all in lower case. I had used that csv file and converted it to abb_dict to expand abbreviations.

Model Architecture:

I had tried out both Bart-Base and T5-Base using the following Hyperparameters and trained them separately and finally evaluated to find the best model and finally Fine Tuned the best model. The following is the hyperparameters used for both the models:

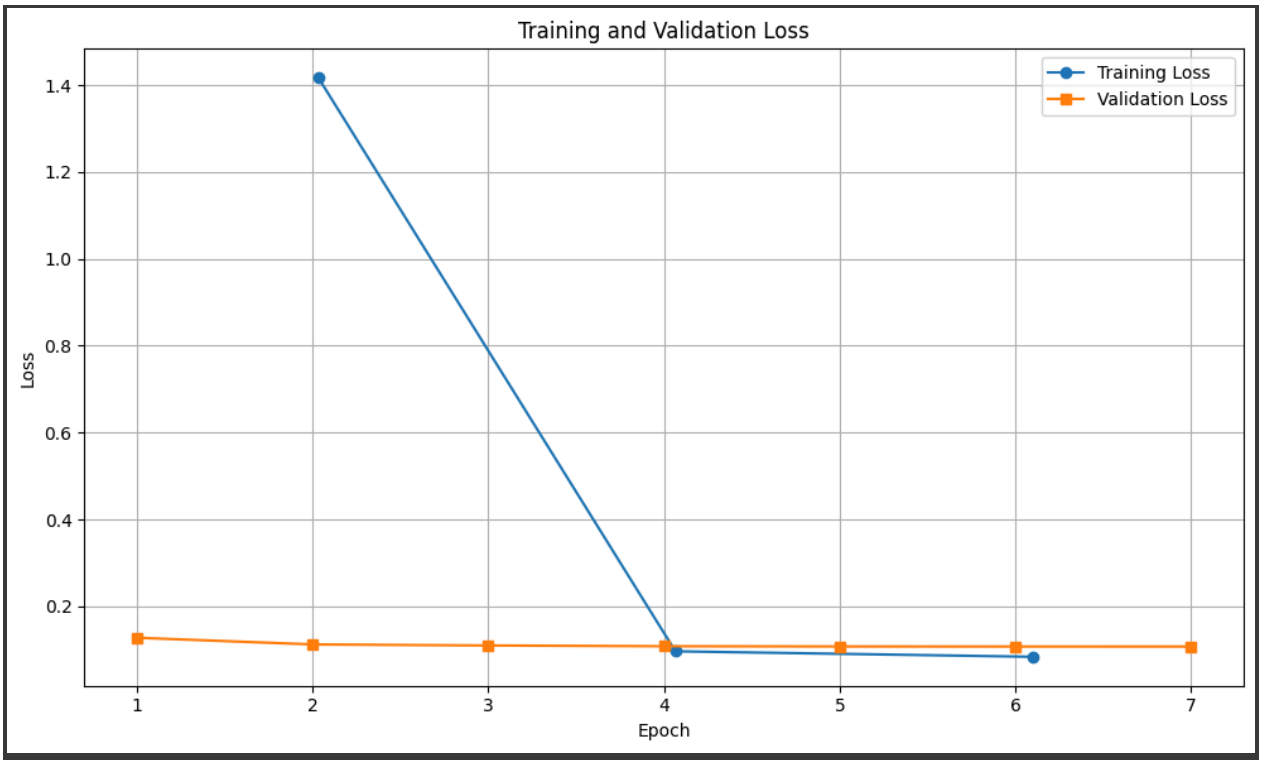
	BART	T5
Learning Rate	1e-4	1e-4
Training Batch	8	4

Evaluating Batch	8	4
Weight Decay	0.01	0.01

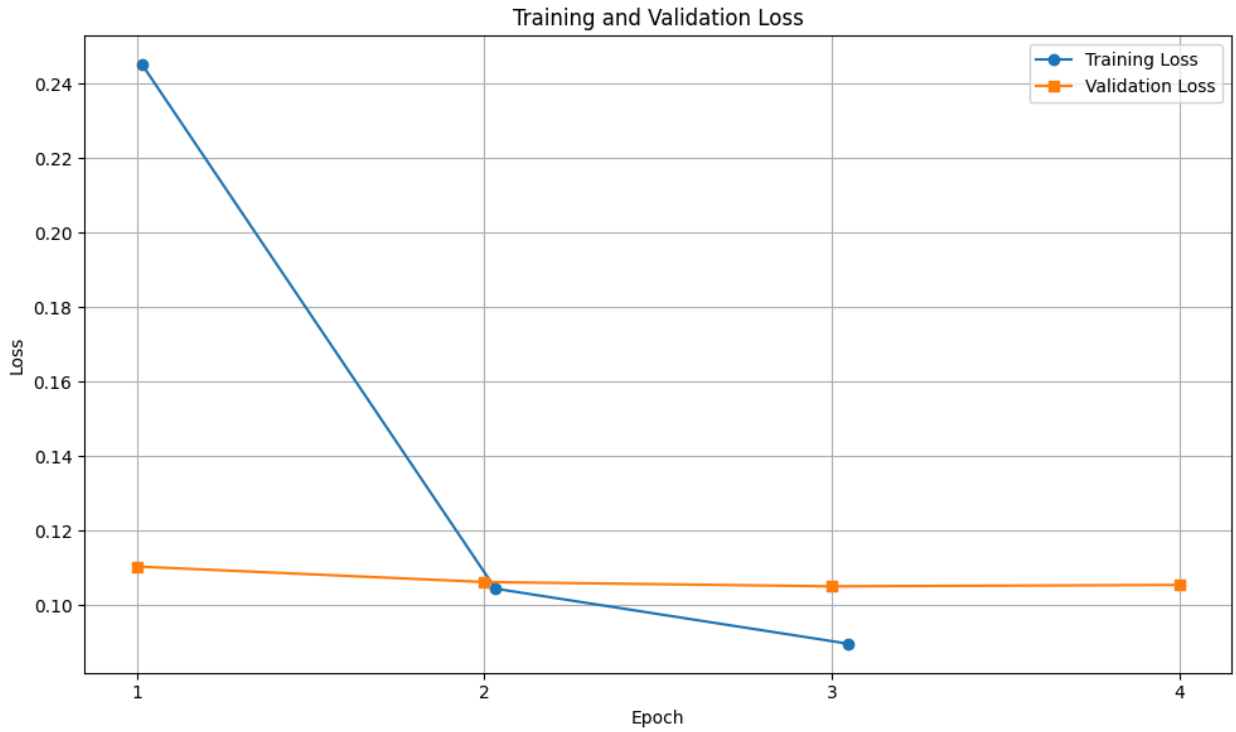
PLOTS:

The following are the Training and Validation plots:

BART:



T5:



The following are the Evaluation Metrics for both the models:

BART:

```
Evaluation Results:
Average ROUGE-L F1 Score: 0.3835
Average ROUGE-L Recall: 0.4052
Average ROUGE-L Precision: 0.4186
Average BLEU-4 Score: 0.2114
Average BERTScore F1: 0.8792
```

T5:

```
Evaluation Results:
Average ROUGE-L F1 Score: 0.3325
Average ROUGE-L Recall: 0.3595
Average ROUGE-L Precision: 0.3574
Average BLEU-4 Score: 0.1613
Average BERTScore F1: 0.8696
```

Analysis:

From the evaluation results, BART outperforms T5 across all metrics:

ROUGE-L Scores:

- **F1 Score: BART (0.3835) > T5 (0.3325)**
- **Recall: BART (0.4052) > T5 (0.3595)**
- **Precision: BART (0.4186) > T5 (0.3574)**
- **BART exhibits better text overlap with the reference, indicating it generates more relevant and coherent summaries.**

BLEU-4 Score:

- **BART (0.2114) > T5 (0.1613)**
- **BLEU measures exact n-gram match precision, meaning BART produces more lexically accurate sequences.**

BERTScore F1:

- **BART (0.8792) > T5 (0.8696)**
- **BART achieves a higher semantic similarity with the reference, showing better contextual understanding.**

NOTE: BART WAS RUN FOR 7 EPOCHS WHILE T5 WAS TRAINED FOR 4 EPOCHS ONLY. DUE TO COMPUTATION CONSTRAINT. THE RESULT WOULDVE BEEN DIFFERENT IF THERE WAS ACCESS TO MORE RESOURCE FOR COMPUTATION.

Resource Constraint:

T5-Base, as we know, is a bigger model than BART primarily due to its larger number of parameters and deeper transformer layers, which increase its computational and memory requirements. Maybe if I had used T5-Small instead of T5-Base and compared it with BART, I would have achieved a more balanced comparison in terms of resource usage. However, the difference in metrics would likely be very minute, as both models follow similar transformer-based architectures optimized for text generation.

The primary advantage of T5-Base lies in its ability to generalize better due to its

larger capacity, but in a constrained training environment like Colab, the additional complexity may not always translate to significantly better performance within a limited number of epochs.

T5-Base was run for fewer epochs than BART primarily due to GPU memory constraints and Colab's free-tier limitations. Since both models were trained on a T4 GPU, running T5 for extended epochs (e.g., 10) while keeping BART at 5 would cause the session to crash due to CUDA memory overflow or exceed the available runtime. To maximize performance within these constraints, the training was adjusted strategically. BART was run for 7 epochs, as it showed consistent improvement until it plateaued, while T5 was run for 4 epochs, as going beyond that would have led to crashes while offering diminishing returns.

The decision was also influenced by the need to balance training across both models. Running T5 for too long would have prevented BART from being trained adequately. Instead, by allocating epochs efficiently, both models could be evaluated fairly without exceeding Colab's limitations. Additionally, performance trends were observed during training, and since BART's performance stabilized at 7 epochs while T5 remained competitive even at 4 epochs, extending training further was unnecessary given the constraints. This approach ensured optimal use of resources while maximizing the results for both models.

TASK 3

Lead : **Rishi Pendyala**

Dataset Description and Preprocessing:

Dataset:

The SQuAD v2 dataset (Stanford Question Answering Dataset v2) is an NLP benchmark dataset designed for evaluating machine reading comprehension and question answering systems. It builds on the original SQuAD dataset by including both answerable and unanswerable questions for a given context, making it more challenging and realistic for real-world applications.

Below is the code for downloading the dataset from huggingface.

```
'''Reference: https://huggingface.co/datasets/rajpurkar/squad_v2'''
from datasets import load_dataset

ds = load_dataset("rajpurkar/squad_v2")
ds.save_to_disk("squad_v2")
```
✓ 0.0s

The DatasetDict is an object from the Hugging Face datasets library that organizes datasets into splits (e.g., train, validation, test). It serves as a container for one or more Dataset objects, each representing a specific subset of the data.

There are 130319 rows in the training dataset and 11873 rows in the validation dataset, out of which we are using 15000 for training.

DatasetDict({ train: Dataset({ features: ['id', 'title', 'context', 'question', 'answers'], num_rows: 130319 }) validation: Dataset({ features: ['id', 'title', 'context', 'question', 'answers'], num_rows: 11873 }) })

Example of one data row:

{'id': '56be85543aeaaa14008c9063', 'title': 'Beyoncé', 'context': 'Beyoncé Giselle Knowles-Carter (/biːˈjɒnseɪ/ bee-YON-say) (born September 4, 1981) is an American singer, songwriter, record producer and actress. Born and raised in Houston, Texas, she performed in various singing and dancing competitions as a child, and rose to fame in the late 1990s as lead singer of R&B girl-group Destiny\'s Child. Managed by her father, Mathew Knowles, the group became one of the world\'s best-selling girl groups of all time. Their hiatus saw the release of Beyoncé\'s debut album, Dangerously in Love (2003), which established her as a solo artist worldwide, earned five Grammy Awards and featured the Billboard Hot 100 number-one singles "Crazy in Love" and "Baby Boy".', 'question': 'When did Beyonce start becoming popular?', 'answers': {'text': ['in the late 1990s'], 'answer_start': [269]}

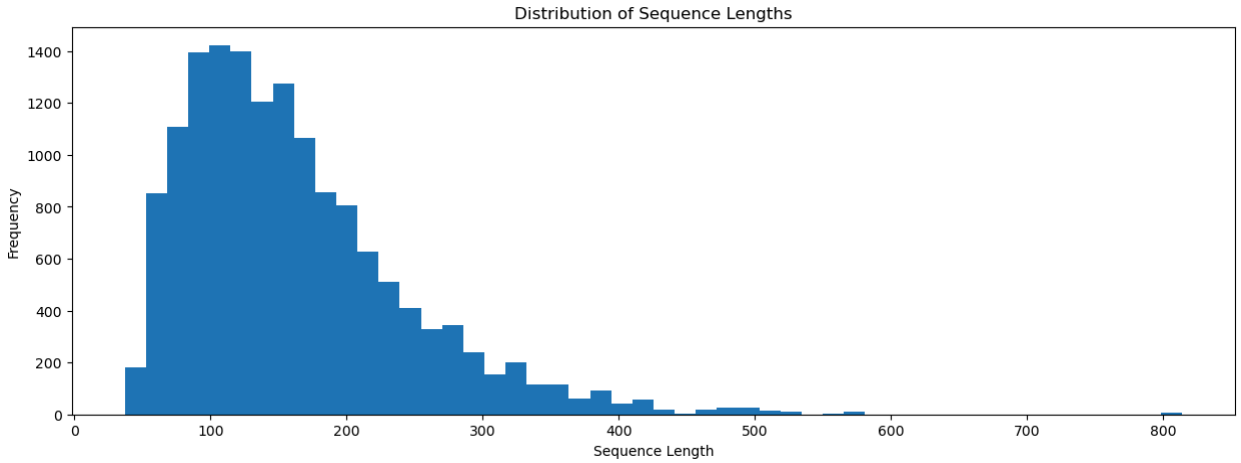
Preprocessing:

The preprocess_data function prepares the SQuAD v2 dataset for fine-tuning a question-answering model using the following:

1. **Text Cleaning:** Strips whitespace from questions and contexts to ensure clean input for tokenization.

2. **Tokenization:** Combines questions and contexts into a single sequence, truncates contexts to fit within the `max_length`, and uses a sliding window (`stride=128`) to handle long contexts. This ensures all relevant context is processed and generates overlapping tokens for better answer coverage.

   In order to choose max length I did a statistical analysis of the distribution of the sequence lengths as follows:



   We can see that a length of 300 covers about 90-95% therefore I have used it as the max_lenght required, and the stride as 128 for getting the context better.

3. **Offset Mapping:** Maps token positions back to their character positions in the original text, allowing alignment between tokenized sequences and the answer spans.
4. **Sample Mapping:** Tracks which tokenized sequence corresponds to each original example, critical for handling multiple overlapping tokenized outputs per context.
5. **Answer Span Mapping:** Calculates start_positions and end_positions by aligning answer character positions with token offsets in the context. If no answer exists, positions are set to 0, ensuring compatibility with unanswerable questions.

This preprocessing ensures that the data is tokenized, aligned, and appropriately labeled for training, helping the model learn to locate answer spans or identify unanswerable questions.

SpanBERTModel

1. **Base Model**:

Utilizes `AutoModelForQuestionAnswering` with pre-trained weights from `SpanBERT/spanbert-base-cased`. This directly outputs logits for `start_positions` and `end_positions`, which are crucial for extractive QA tasks like SQuAD v2.

2. **Weight Initialization**:

The `qa_outputs` layer's weights are initialized using a normal distribution (`mean=0.0`, `std=0.02`), and biases are set to zero. This ensures stable and unbiased starting conditions for fine-tuning.

3. **Forward Pass**:

The model takes tokenized `input_ids` and `attention_mask` and outputs loss (if `start_positions` and `end_positions` are provided) or logits during inference.

SpanBERT-CRF Model

1. **Base Model**:

SpanBERT Pretraining: Utilizes SpanBERT/spanbert-base-cased as the base model. This is ideal for span-based tasks (e.g., QA) due to its pre-training on contiguous spans, enabling strong contextual representations for span prediction.

CRF Integration: Instead of directly predicting start/end positions (standard SpanBERT), the model treats QA as a sequence tagging problem with a Conditional Random Field (CRF) layer. This enforces structural constraints on output spans (e.g., ensuring valid transitions like O → B → I instead of invalid transitions like I → O).

2. **QA Transformation Layer**

Architecture: The qa_transforms module (Linear → GELU → Dropout → Linear) projects SpanBERT's hidden states into emission scores for two tags (B and I).

Purpose: Converts contextual embeddings into tag-specific logits, which the CRF uses to model transition dependencies.

Weight Initialization: Linear layers are initialized with mean=0.0, std=0.02 (matching BERT's initialization) for stable fine-tuning, while biases are zero-initialized to avoid premature skews.

3. **CRF Layer**

Role: The CRF learns transition rules between tags (e.g., B must follow O, not I), ensuring predictions form valid, contiguous spans.

Training: Maximizes the likelihood of valid tag sequences (derived from ground-truth spans) using the Viterbi algorithm.

Inference: Decodes the most probable tag sequence (e.g., O, B, I, I, O) via dynamic programming, avoiding invalid sequences.

4. **Span Construction**

Tag-to-Span Conversion: In get_span_predictions, the first and last positions of B/I tags define the answer span. This assumes a single contiguous span (suitable for SQuAD v2).

Handling No-Answer: Returns (0, 0) if no B/I tags exist, aligning with SQuAD v2's unanswerable questions.

5. **Loss Function**

CRF Negative Log-Likelihood: The loss optimizes the CRF's joint probability of the correct tag sequence, considering both emissions and transitions. This contrasts with standard SpanBERT's cross-entropy loss on start/end logits.

Hyperparameters Used in Training

Learning Rate (`2e-5`)

After testing with many other learning rate values, I found this to be the most optimum as for higher learning rates, both models start overfitting and the EM scores go down a lot. For 15 epochs this is a good learning rate as the model can train optimally.

Number of Epochs (`15 for SpanBERT and 10 for SpanBERT-CRF`)

Chosen to allow both models sufficient time to converge. For the base model, earlier epochs may yield competitive performance due to its simpler architecture.

The base model would benefit from the full 15 epochs since the CRF layer requires additional training to optimize the structured dependencies but due to the huge amount of time that it was taking, I restricted it to 10 epochs.

Batch Size (8)

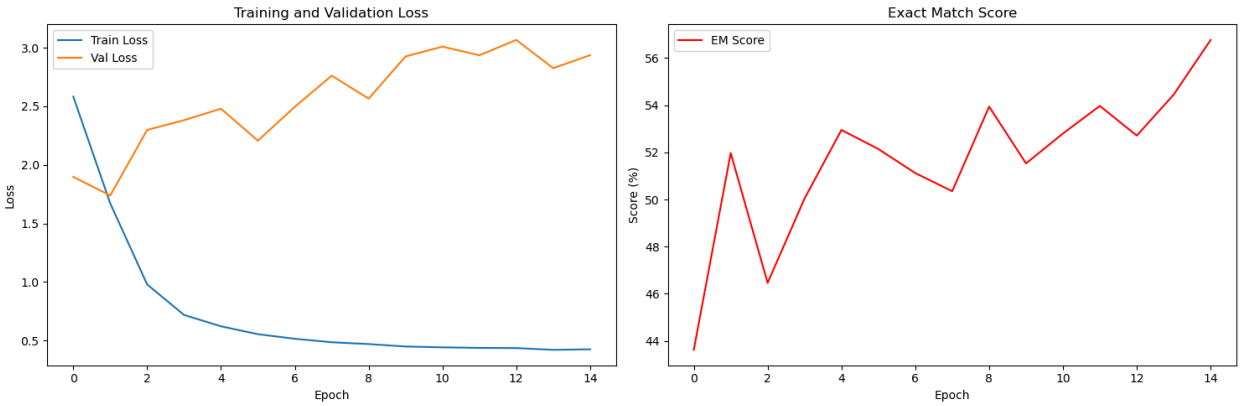
Smaller batch size ensures stable gradient updates while being memory-efficient for SpanBERT's architecture.

Padding Length (300)

Explanation given above

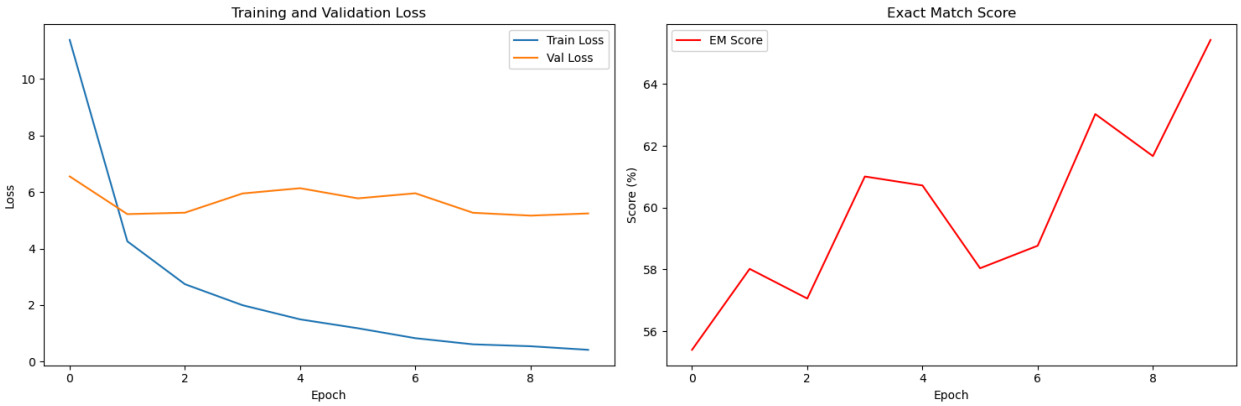
Train and Validation Loss Plots:

For SpanBERT:




```
Epoch 15:
Average training loss: 0.425
Average validation loss: 2.936
Exact match score: 56.77
```

For SpanBERT-CRF:



```
Epoch 10:
Average training loss: 0.4157
Average validation loss: 5.2437
Exact match score: 65.43%
```

Comparative analysis of SpanBERT-CRF and SpanBERT:

SpanBERT Performance Analysis:

SpanBERT exhibits faster convergence in training loss, with a steep decline during the initial epochs. This suggests efficient learning of start and end boundary predictions due to its simpler architecture. However, the lack of structured modeling causes the model to plateau earlier and become more prone to overfitting. This is evident in the validation loss, where SpanBERT shows initially lower values but fluctuates significantly and eventually increases. These fluctuations indicate that the model struggles to generalize effectively, particularly in handling noisy or ambiguous validation data. Furthermore, the Exact Match (EM) score for SpanBERT is relatively lower, with slower improvements across epochs. This reflects the model's difficulty in capturing nuanced span boundaries and resolving ambiguous cases, limiting its overall effectiveness.

SpanBERTCRF Performance Analysis:

SpanBERTCRF, on the other hand, converges more slowly in training loss compared to SpanBERT. The slower convergence is due to the additional complexity introduced by the CRF layer, which optimizes sequence-level consistency. Despite this slower convergence, the validation loss for SpanBERTCRF remains stable across epochs, suggesting better generalization. The CRF layer enforces structural constraints, reducing the risk of overfitting and ensuring valid span predictions. In terms of the Exact Match (EM) score, SpanBERTCRF consistently outperforms SpanBERT, showing higher scores and steady improvement over epochs. This reflects the CRF's ability to handle nuanced and edge-case predictions by modeling dependencies between tags and enforcing structural consistency, making SpanBERTCRF a more robust and effective solution for the task.

TASK 3
Rishi Pendyala

Preprocessing is handled by the class MuSEDataset

MuSEDataset explanation:

- Loads multimodal sarcasm data from a TSV file (text, explanation, image ID, target).
- Retrieves image descriptions and object detections from pickle files.
- Loads and preprocesses images (resize, normalize).
- Constructs a combined input: "Text: ... Image Description: ... Objects: ..."
- Tokenizes multimodal input, explanation, and sarcasm target using a tokenizer
- Handles missing images with blank tensors and missing descriptions/objects with defaults.
- Returns a dictionary with tokenized inputs, attention masks, preprocessed image tensor, and raw text fields.

Model Architecture:

High level image features are extracted using a Vision Transformer (vit_b_16)

Text features along with the sarcasm target are obtained from bart encoder

Cross modal attention:

Given text features Et and image features Ev

Calculate attention using $A = softmax((QK^T)/sqrt(dk))V$ for both vision and text separately

Then do element wise multiplication for cross modal flow of information

$Ftv = At \odot Ev$

$Fvt = Av \odot Et$

Gated Fusion:

Using

- Ftv = text-aware vision
- Fvt = vision-aware text

Gate Computation:
$g = \sigma(W_1 \cdot [Ftv; Fvt] + b_1)$

Fused Output:
$F = g \odot Ftv + (1 - g) \odot Fvt$

(σ = sigmoid, [;] = concatenation, $\odot$ = element wise multiplication)

Return F1, F2, Fv, Ft

Shared Fusion:

Merges all fused features (F1, F2, Fv, Ft) using learnable scalars → FSF.

Z is a linear projection of FSF for the required dimensions

Then the model can either be in training mode or inference mode

Training Mode (target_ids is not None):

- Pass the truncated Z as encoder output into the BART decoder.
- Compute the loss by supplying target_ids as decoder labels.
- Return the decoder outputs including the loss.

Inference Mode (target_ids is None):

1. Wrap Z in a BaseModelOutput to simulate standard encoder output.
2. Generate sequences using BART's beam search with predefined parameters.

Hyperparameters:

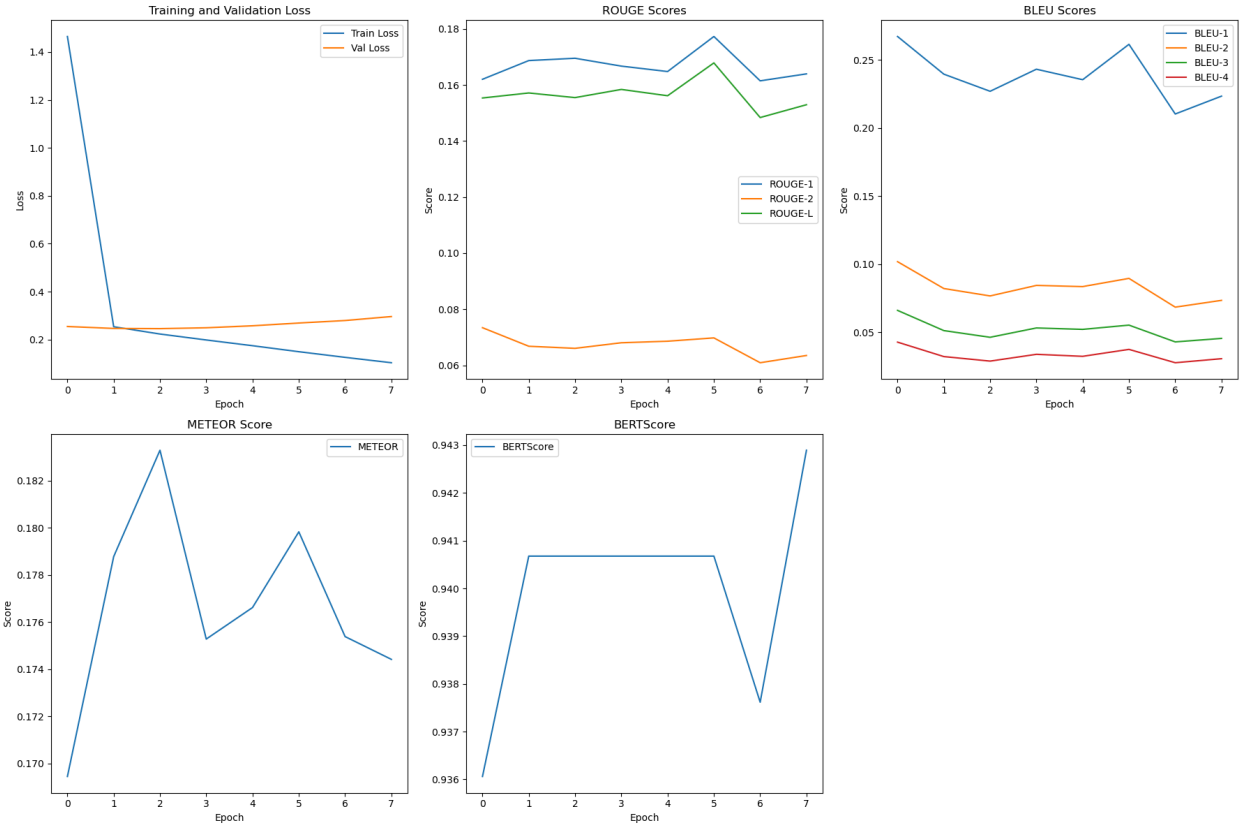
Batch size = 16

Learning rate = 0.5e-5

Epochs = 8

Tokenizer - BART Base

Training:



Epoch 1/8

Training Loss: 1.4646

Validation Loss: 0.2547

ROUGE Scores:

ROUGE-1: 0.1620

ROUGE-2: 0.0734

ROUGE-L: 0.1553

BLEU Scores:

BLEU-1: 0.2672

BLEU-2: 0.1019

BLEU-3: 0.0662

BLEU-4: 0.0429

METEOR: 0.1694

BERTScore: 0.9361

Epoch 2/8

Training Loss: 0.2537

Validation Loss: 0.2467

ROUGE Scores:

ROUGE-1: 0.1687

ROUGE-2: 0.0668

ROUGE-L: 0.1571

BLEU Scores:

BLEU-1: 0.2396

BLEU-2: 0.0822

BLEU-3: 0.0513

BLEU-4: 0.0323

METEOR: 0.1788

BERTScore: 0.9407

Epoch 3/8

Training Loss: 0.2233

Validation Loss: 0.2459

ROUGE Scores:

ROUGE-1: 0.1695

ROUGE-2: 0.0661

ROUGE-L: 0.1555

BLEU Scores:

BLEU-1: 0.2270

BLEU-2: 0.0768

BLEU-3: 0.0464

BLEU-4: 0.0289

METEOR: 0.1833

BERTScore: 0.9407

Epoch 4/8

Training Loss: 0.1985

Validation Loss: 0.2494

ROUGE Scores:

ROUGE-1: 0.1667

ROUGE-2: 0.0681

ROUGE-L: 0.1584

BLEU Scores:

BLEU-1: 0.2432

BLEU-2: 0.0845

BLEU-3: 0.0533

BLEU-4: 0.0339

METEOR: 0.1753

BERTScore: 0.9407

Epoch 5/8

Training Loss: 0.1746

Validation Loss: 0.2577

ROUGE Scores:

ROUGE-1: 0.1648

ROUGE-2: 0.0686

ROUGE-L: 0.1561

BLEU Scores:

BLEU-1: 0.2355

BLEU-2: 0.0836

BLEU-3: 0.0522

BLEU-4: 0.0324

METEOR: 0.1766

BERTScore: 0.9407

Epoch 6/8

Training Loss: 0.1496

Validation Loss: 0.2692

ROUGE Scores:

ROUGE-1: 0.1773

ROUGE-2: 0.0698

ROUGE-L: 0.1679

BLEU Scores:

BLEU-1: 0.2615

BLEU-2: 0.0897

BLEU-3: 0.0554

BLEU-4: 0.0375

METEOR: 0.1798

BERTScore: 0.9407

Epoch 7/8

Epoch complete. Average loss: 0.1260

Training Loss: 0.1260

Validation Loss: 0.2796

ROUGE Scores:

ROUGE-1: 0.1615

ROUGE-2: 0.0609

ROUGE-L: 0.1484

BLEU Scores:

BLEU-1: 0.2103

BLEU-2: 0.0685

BLEU-3: 0.0430

BLEU-4: 0.0277

METEOR: 0.1754

BERTScore: 0.9376

Checkpoint saved to checkpoints\muse_model_epoch_7.pt

Epoch 8/8

Training on epoch 8...

Total number of batches: 187

Training Loss: 0.1036

Validation Loss: 0.2963

ROUGE Scores:

ROUGE-1: 0.1639

ROUGE-2: 0.0635

ROUGE-L: 0.1529

BLEU Scores:

BLEU-1: 0.2235

BLEU-2: 0.0735

BLEU-3: 0.0456

BLEU-4: 0.0307

METEOR: 0.1744

BERTScore: 0.9429

Sample Results:

Example 1:

Text: '<user> thank u for this awesome network in malad (see pic) . # patheticcs'

Generated: the author is pissed at <user> for getting rid of some ice.

Reference: the author is pissed at <user> for not getting network in malad.

Example 2:

Text: Nothing like waiting for an hour on the tarmac for a gate to come open in snowy, windy Chicago!

Generated: the author isn't excited for a buttload of work.

Reference: nothing worst than waiting for an hour on the tarmac for a gate to come open in snowy, windy chicago.

Example 3:

Text: 'ahhh ! my favorite thing about spring ! # dst # springforward '

Generated: when you're upset, posting more to facebook doesn't eliminate any drama.

Reference: nobody likes getting one hour of their life sucked away.

Example 4:

Text: 'can anyone of you imagine a better way to start the new week than having a salivary gland biopsy on monday morning ? me neither ... emoji_300'

Generated: the author is sad to have to work in such a cold weather.

Reference: having a salivary gland biopsy on monday morning is not a good way to start the new week.

Example 5:

Text: phew ! it 's going to be scorching hot this w-end ! the high on saturday is - 1 ! windchill will prob be - 30 . emoji_619 emoji_300

Generated: the author isn't excited for school tomorrow because it's too hot.

Reference: the author is worried that the weekend is going to be freezing with a high of -1 and windchill probably -30.