Assignment-2 Report

TASK-1

PREPROCESSING

1. Tokenization

The input sentences are split into individual words (tokens). Tokenization is essential as it helps in assigning labels to each word for Named Entity Recognition (NER) tasks.

Example Input Sentence:
"All the money went into the interior decoration, none of it went to the chefs."

Tokenized Output:
["All", "the", "money", "went", "into", "the", "interior", "decoration", ",", "none", "of", "it", "went", "to", "the", "chefs", "."]

2. BIO Encoding for Aspect Terms

The BIO (Beginning-Inside-Outside) tagging scheme is used to label each token. Aspect terms (words related to key concepts) are tagged using the B (Beginning) and I (Inside) tags, while other words are tagged as O (Outside).

Example Labels for the Above Sentence:
["O", "O", "O", "O", "O", "O", "B-ASPECT", "I-ASPECT", "O", "O", "O", "O", "O", "O", "B-ASPECT", "O"]

B-ASPECT: Marks the beginning of an aspect term (e.g., "interior").
I-ASPECT: Marks subsequent tokens in the aspect term (e.g., "decoration").
O: Marks words that are not part of any aspect term.

3. Extracting Aspect Terms

Aspect terms are extracted from the sentences based on BIO tagging. These terms help in aspect-based sentiment analysis.
Extracted Aspect Terms from the Example Sentence:
["interior decoration", "chefs"]

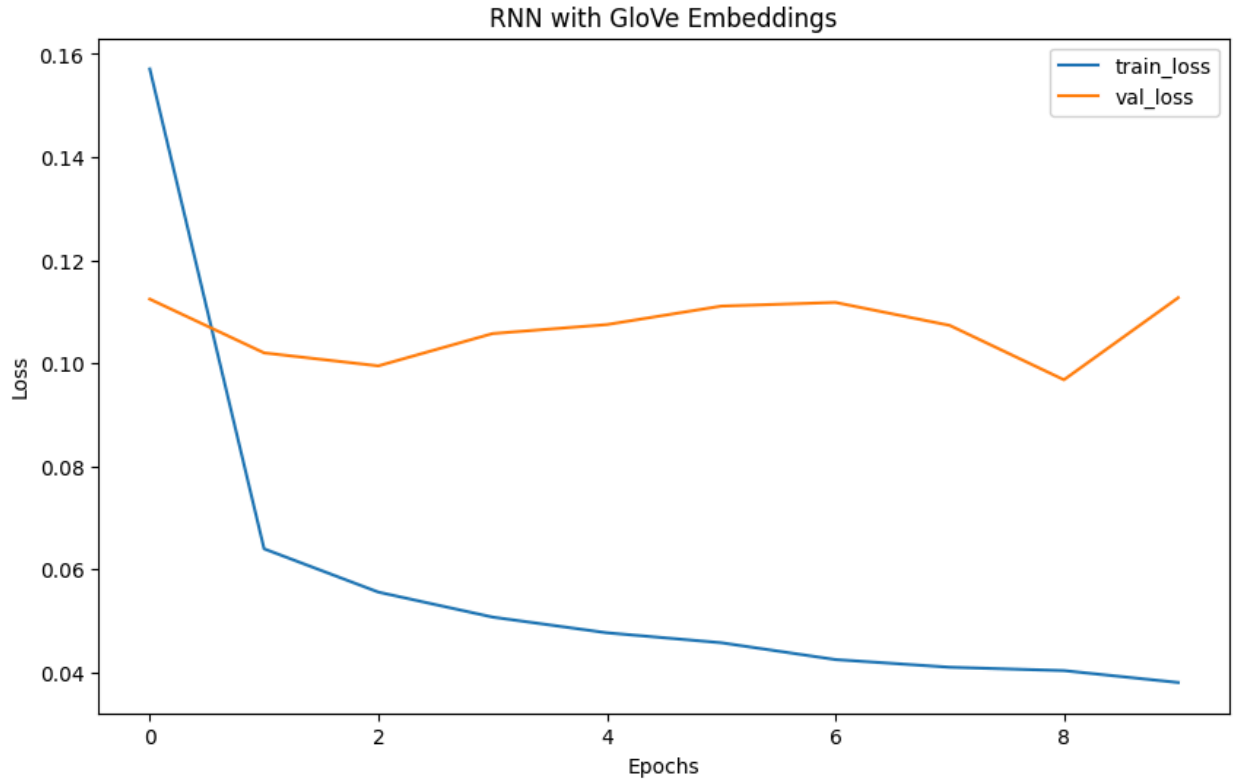
MODEL ARCHITECTURE AND HYPERPARAMETER USED

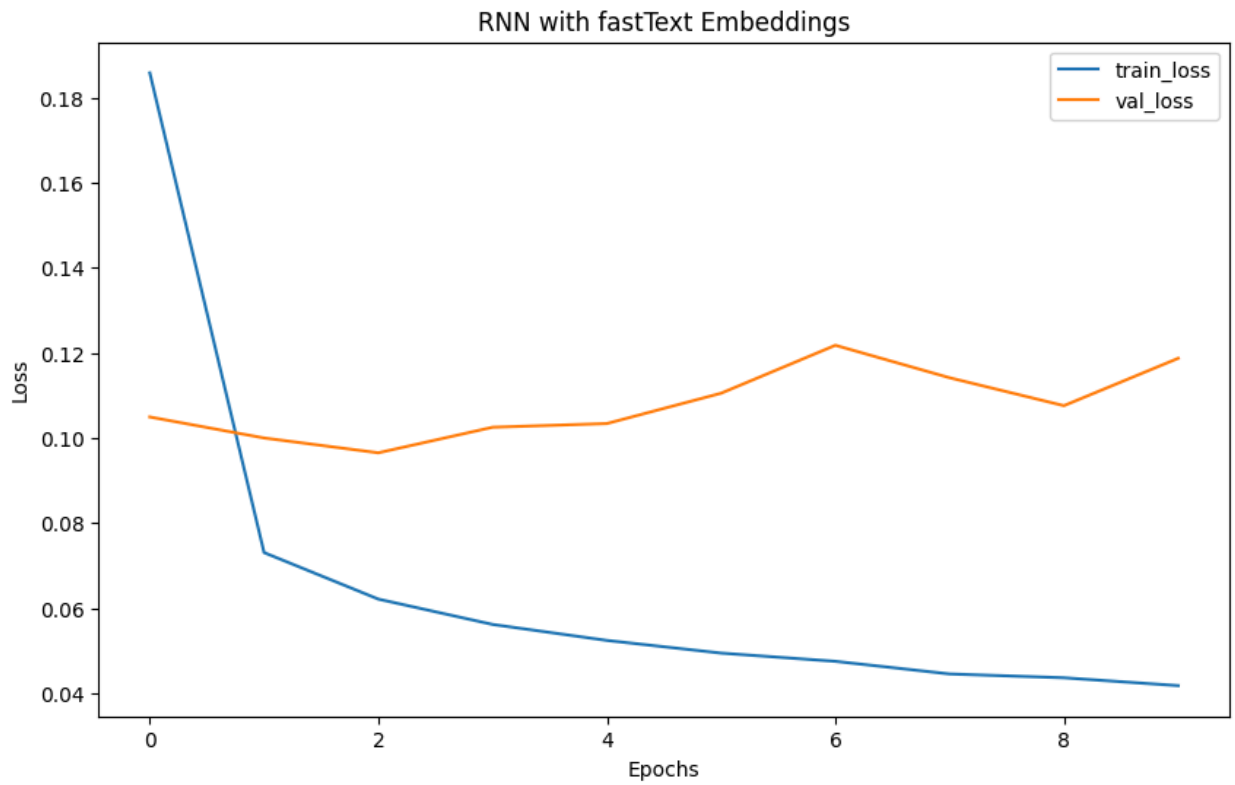
Since Task 1 only involves implementing a tokenizer, there is no deep learning model involved here. However, the tokenizer itself follows a structured approach:

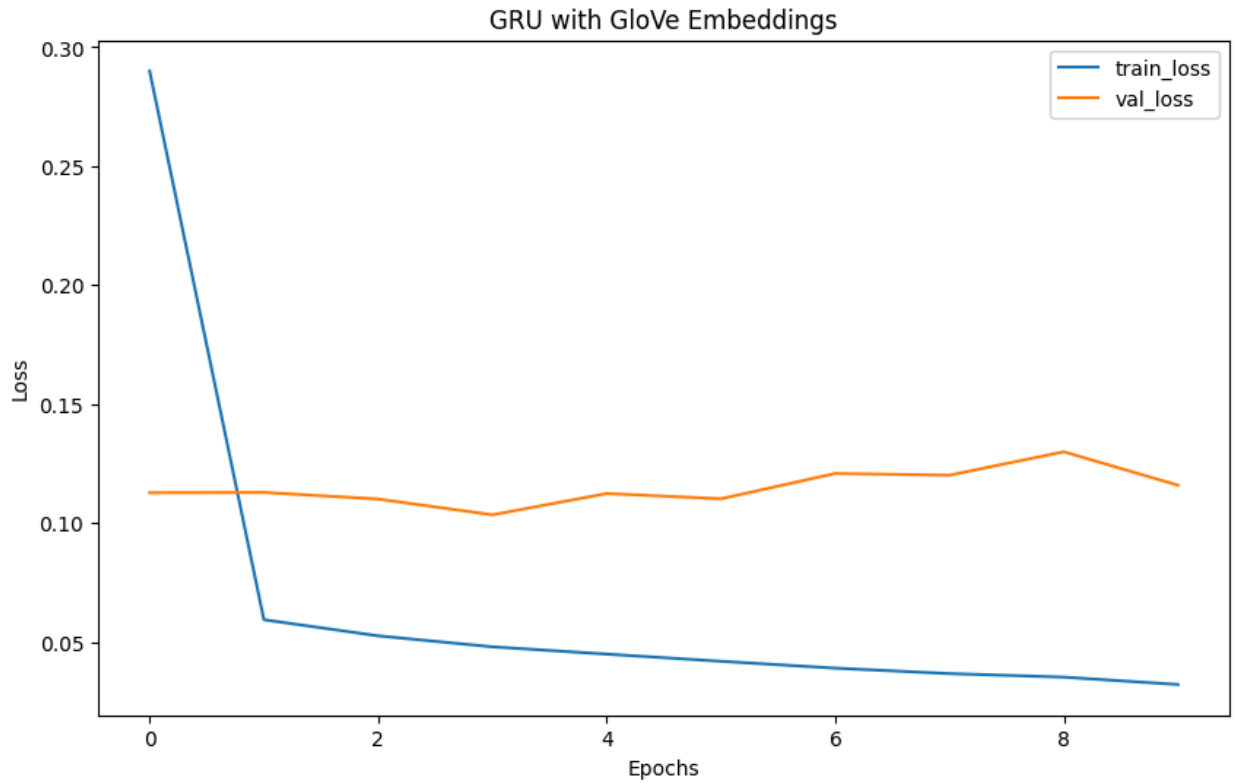
Byte Pair Encoding (BPE)-like merging strategy: The most frequent adjacent tokens are merged iteratively.

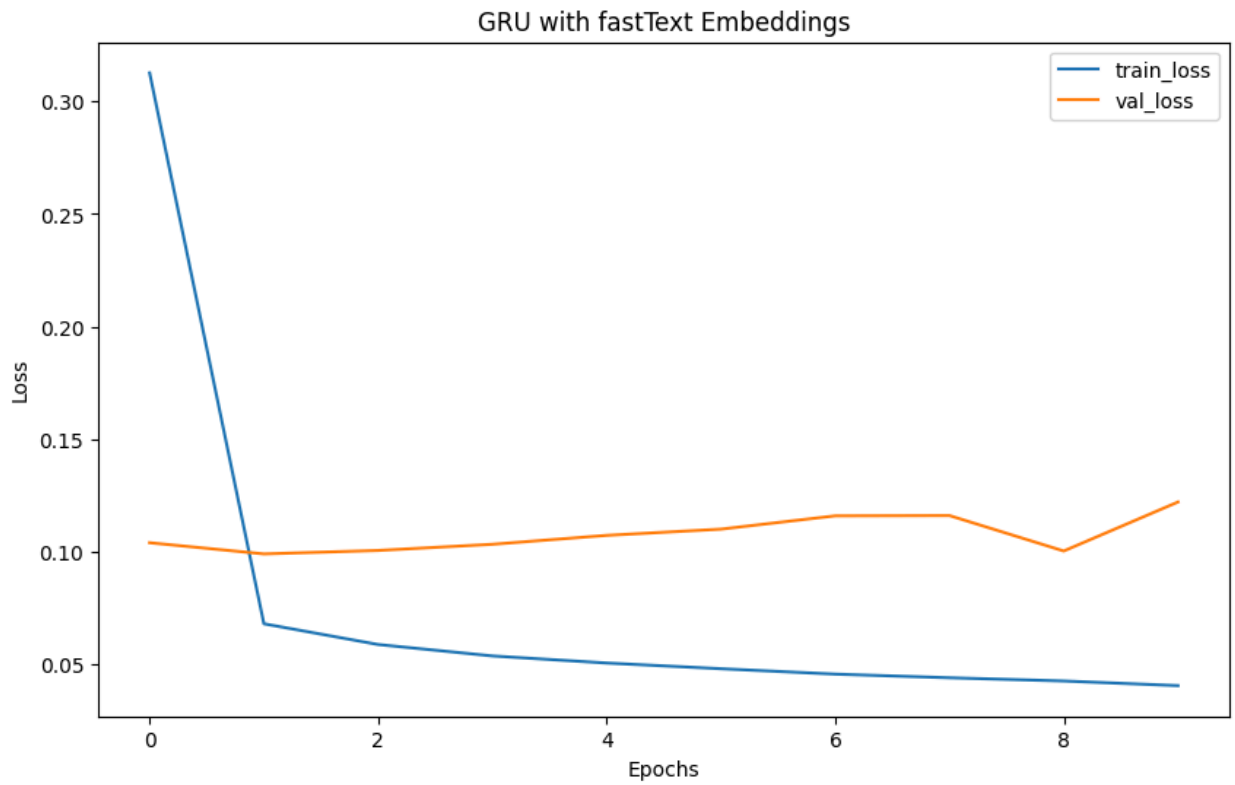
Vocabulary Size Limit: The number of tokens is constrained to optimize memory usage while ensuring meaningful tokenization.
Token Format: Special tokens like "##" are used to denote subwords.
Efficient String Processing: The implementation ensures fast tokenization using dictionary-based frequency lookups.

TRAINING AND VALIDATION LOSS PLOTS




RNN with fastText Embeddings
train_loss
val_loss
Loss
Epochs
0.18
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0
2
4
6
8


GRU with GloVe Embeddings
train_loss
val_loss
Loss
Epochs
0.30
0.25
0.20
0.15
0.10
0.05
0
2
4
6
8



PERFORMANCE COMPARISON OF ALL MODELS

MODEL	TAG LEVEL F1	CHUNK LEVEL F1
RNN with GloVe	94.19	92.69
RNN with fastText	94.01	92.56
GRU with GloVe	93.68	92.19
GRU with fastText	93.75	92.29

BEST PERFORMING MODEL

RNN models outperform GRU models in both Tag-level and Chunk-level F1 scores.
The RNN with GloVe model achieves the highest F1 scores (Tag: 94.19, Chunk: 92.69), making it the best-performing model.

```
Best Model: RNN with GloVe
  Tag-level F1: 94.1921
  Chunk-level F1: 92.6930

Best model (RNN with GloVe) saved to best_model.h5
Model information saved to model_info.pkl
```

TASK-2

PREPROCESSING
1. Tokenization
Each sentence is tokenized by splitting it into individual words. Tokenization helps in identifying aspect terms and their positions within the sentence.
Example:
Original Sentence:
"The food is uniformly exceptional with a very capable kitchen."

Tokenized Output:
["The", "food", "is", "uniformly", "exceptional", "with", "a", "very", "capable", "kitchen", "."]

2. Extracting Aspect Terms
Each sentence contains one or more aspect terms, each associated with a sentiment polarity. The preprocessing step extracts these terms along with their sentiment.

Example:
For the sentence above, if the dataset contains:

Aspect term: "food" with polarity positive
Aspect term: "kitchen" with polarity positive
We extract these terms separately.

3. Identifying Aspect Term Index
Each aspect term's position is determined in the tokenized sentence. The algorithm scans the sentence and locates the starting position of the aspect term.

Example:

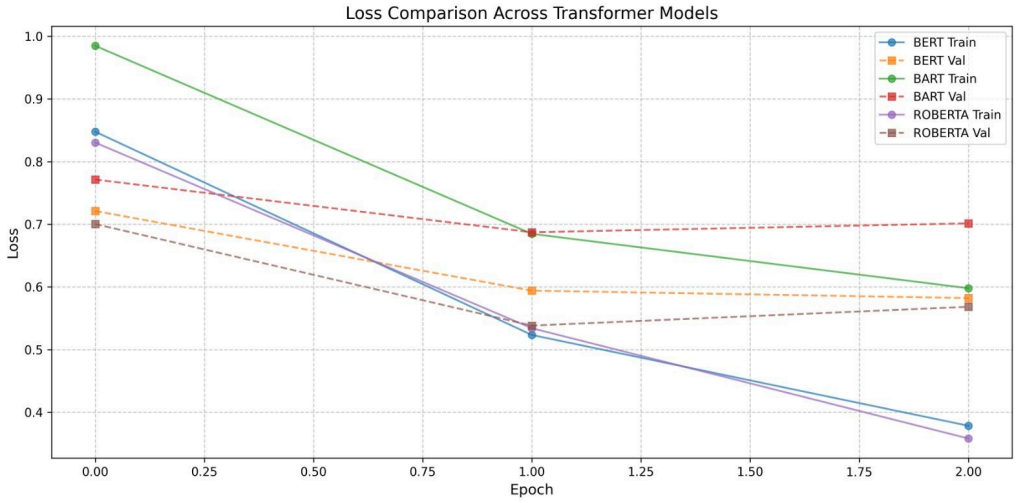
Aspect Term	Index in Tokens
food	1
kitchen	9

If direct matching fails, a fallback approach checks for substring matches within tokens.

Model Architectures and Hyperparameters
SimpleRNN
Embedding: Word embedding + position embedding
RNN: Single-layer RNN
Hidden Dim: Variable (set by experiment)
Output Dim: Number of classes
Activation: Linear

SimpleGRU
Embedding: Word embedding + position embedding
GRU: Single-layer GRU
Hidden Dim: Variable
Output Dim: Number of classes
Activation: Linear

AttentionLSTM
Embedding: Word embedding + position embedding
LSTM: Bidirectional, multi-layer (default: 2 layers)
Hidden Dim: Variable
Attention: Fully connected attention mechanism
Dropout: 0.3 (if n_layers > 1)
Output Dim: Number of classes
Activation: ReLU + Linear

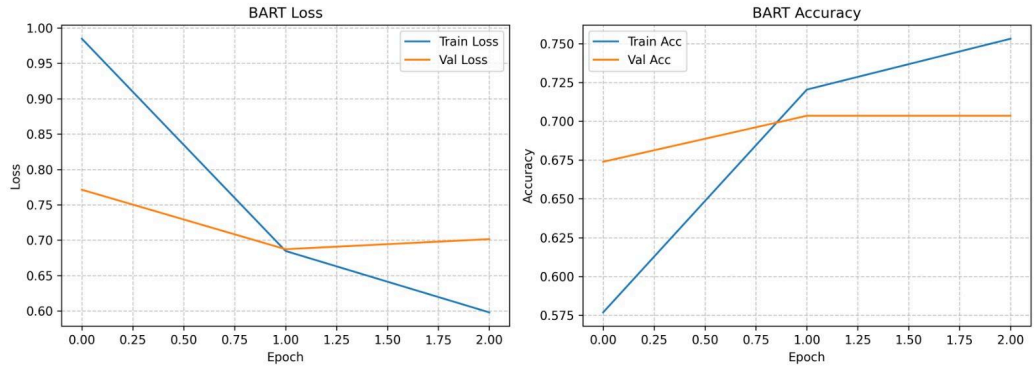


RoBERTa has the steepest decline in training loss, indicating strong learning progress.
BERT also shows a steady decrease in loss, closely following RoBERTa.
BART has a higher validation loss compared to the other models, suggesting it may not generalize as well.

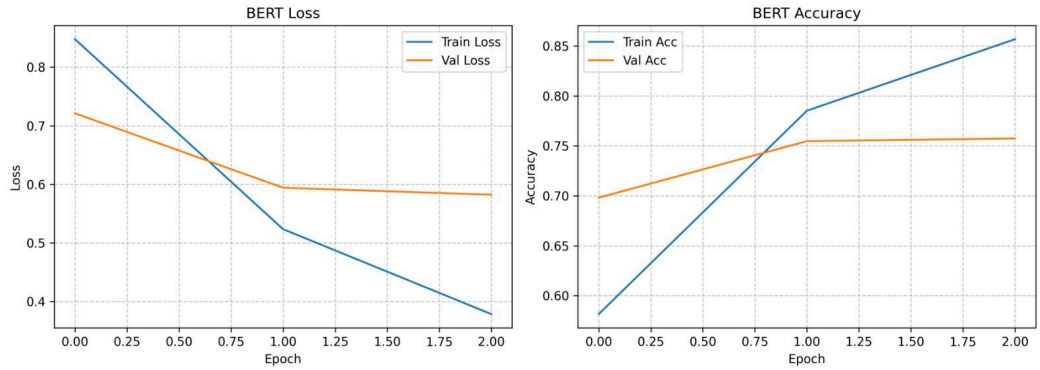
Conclusion:

RoBERTa exhibits the best training performance in terms of minimizing loss over epochs.

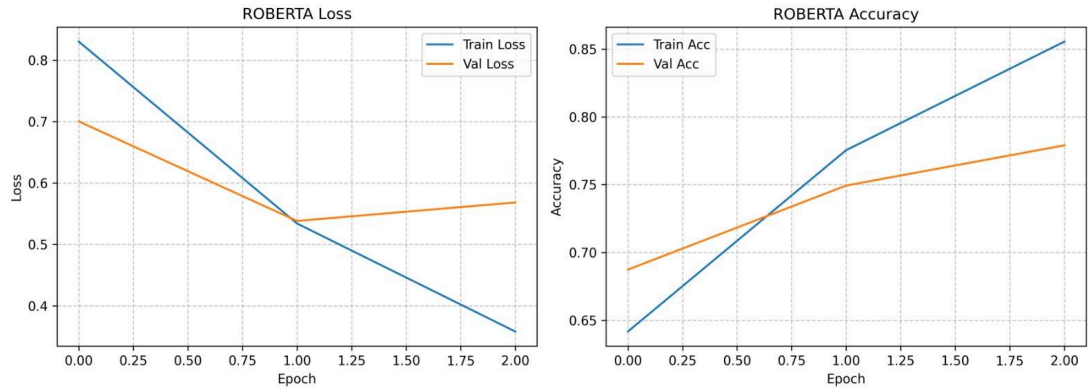
BART - Best Val Acc: 0.7035

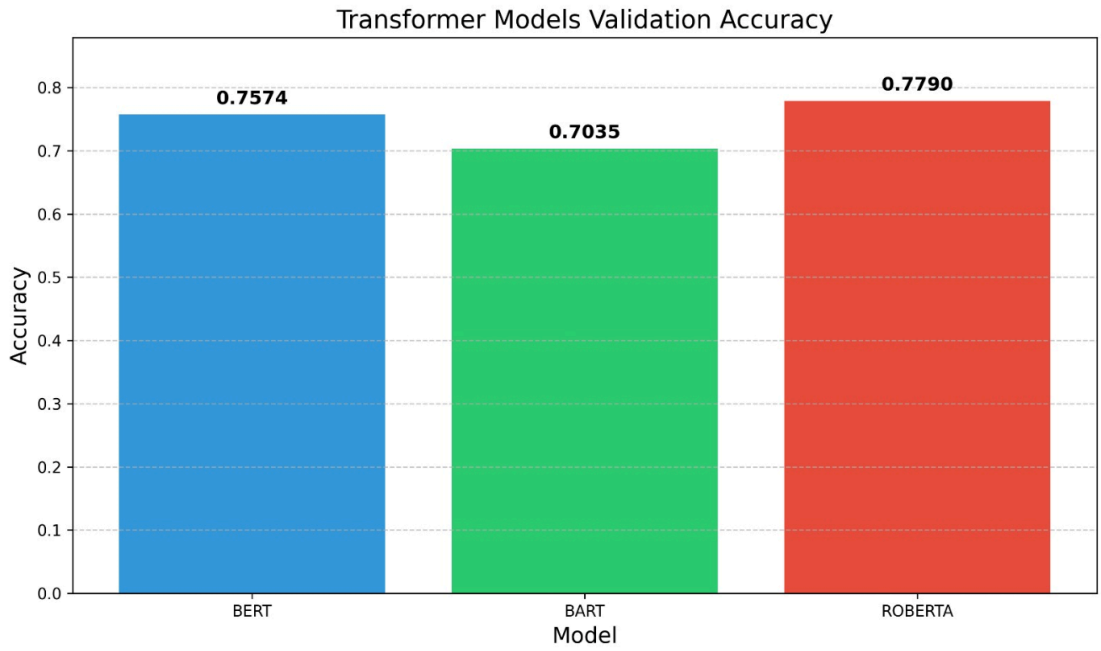


BERT - Best Val Acc: 0.7574



ROBERTA - Best Val Acc: 0.7790





BART Performance

- Best Validation Accuracy: 0.7035

BART shows an improving trend in training accuracy but lags behind in validation accuracy.
The training loss decreases significantly, but validation loss remains relatively high, indicating possible overfitting.

BERT Performance

- Best Validation Accuracy: 0.7574

BERT maintains a balance between training and validation loss.
The accuracy improvement is consistent, making it a strong performer.

RoBERTa Performance

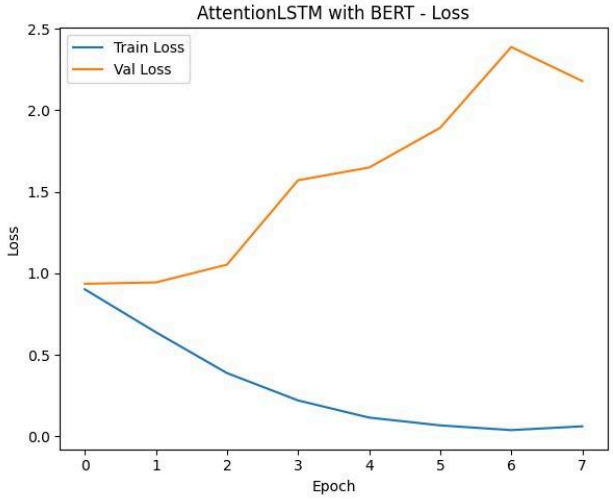
- Best Validation Accuracy: 0.7790

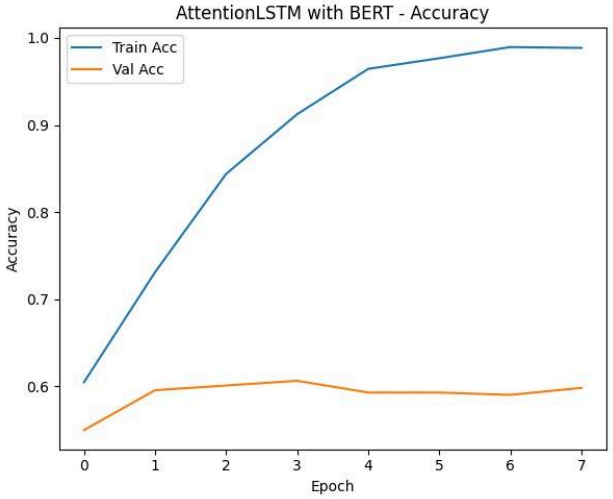
RoBERTa achieves the highest validation accuracy.
The validation loss is more stable compared to BART.
The accuracy curve shows consistent improvement, indicating better generalization.

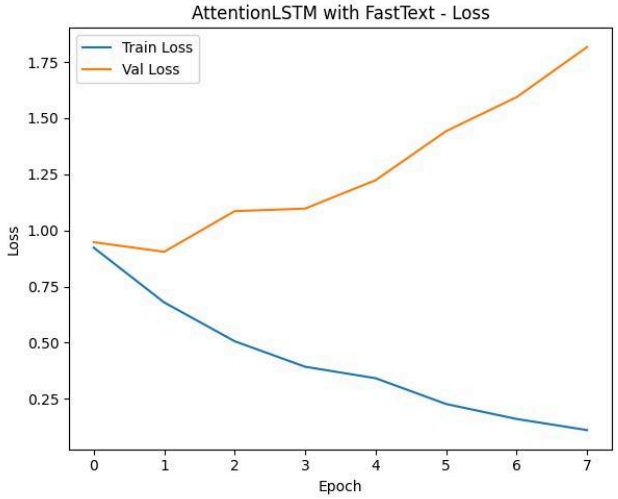
EVALUATION METRIC ON VALIDATION OF ALL 3 MODELS

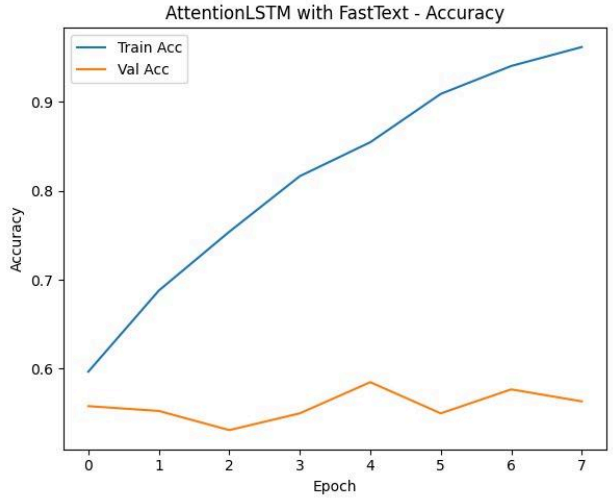
MODEL	BEST VALIDATION ACCURACY
BART	0.7035

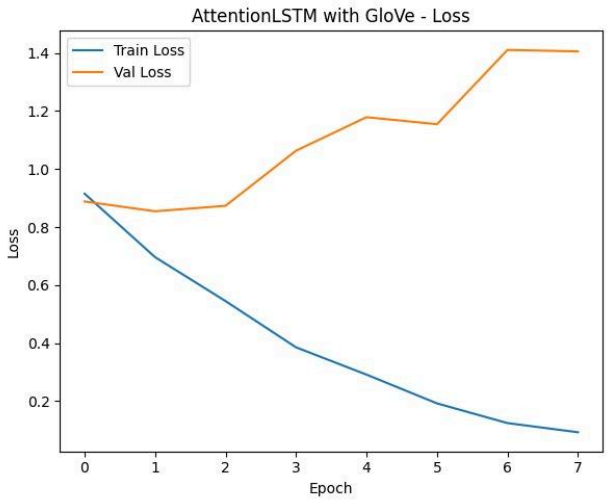
BERT	0.7574
RoBERTa	0.7790

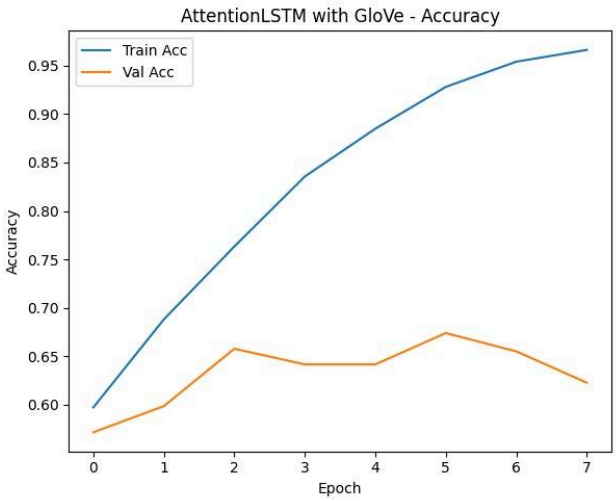


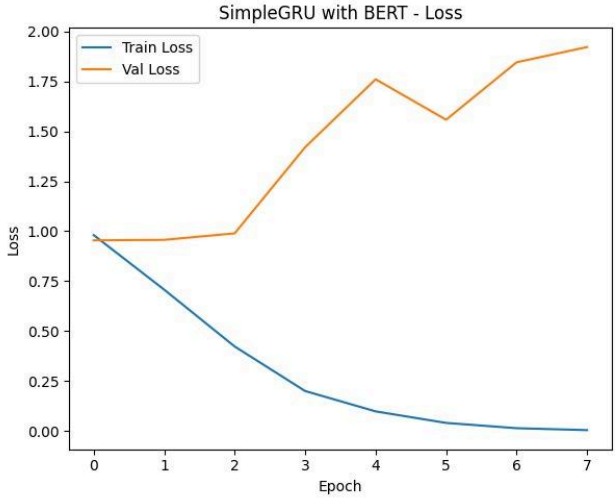


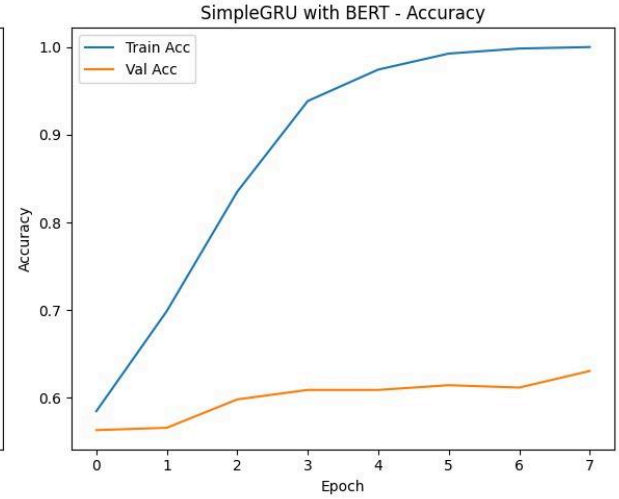


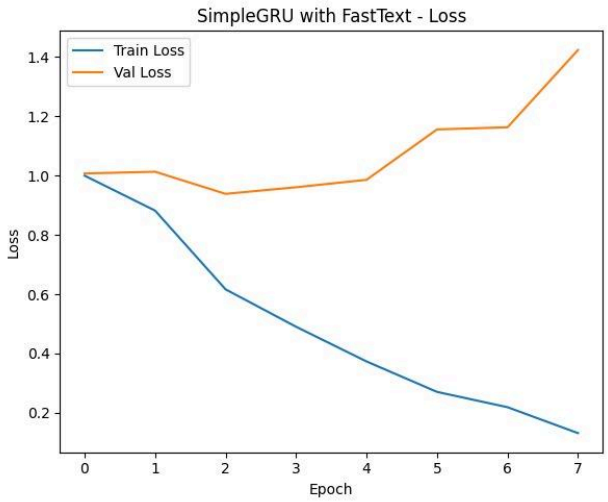


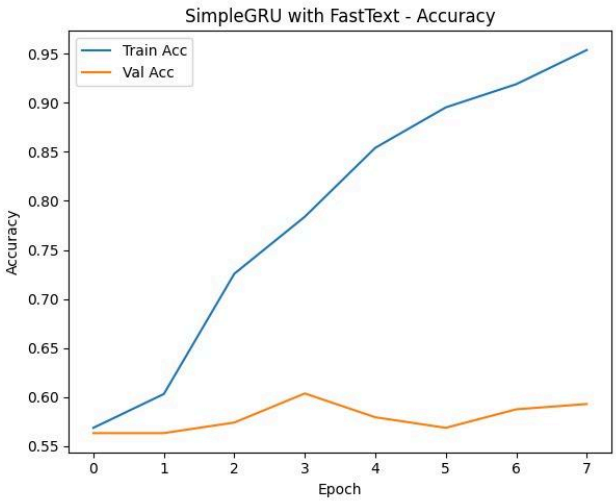

AttentionLSTM with GloVe - Loss
Train Loss
Val Loss
Loss
Epoch

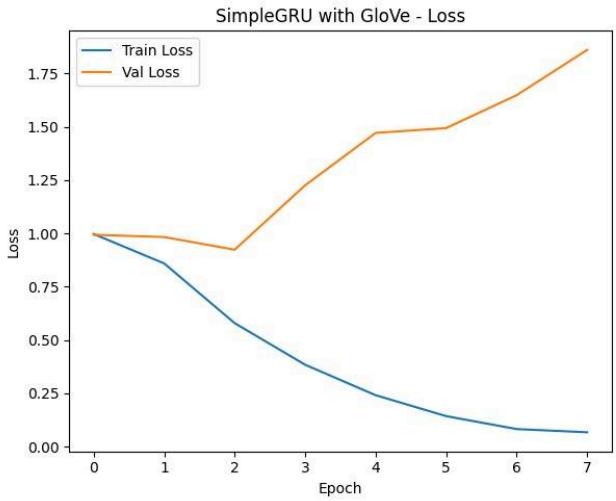

AttentionLSTM with GloVe - Accuracy
Train Acc
Val Acc
Accuracy
Epoch

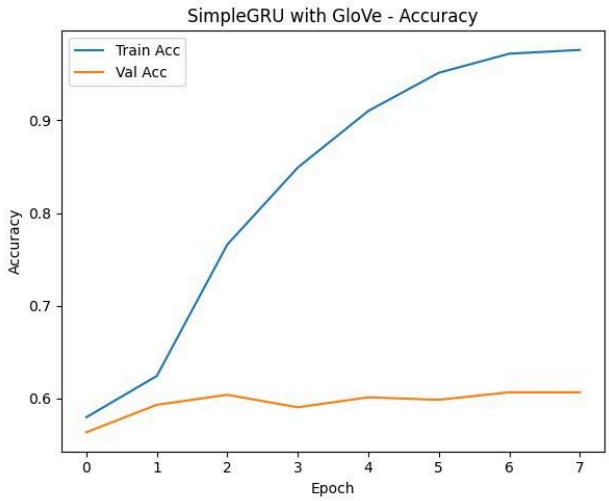

SimpleGRU with BERT - Loss
Train Loss
Val Loss
Loss
Epoch

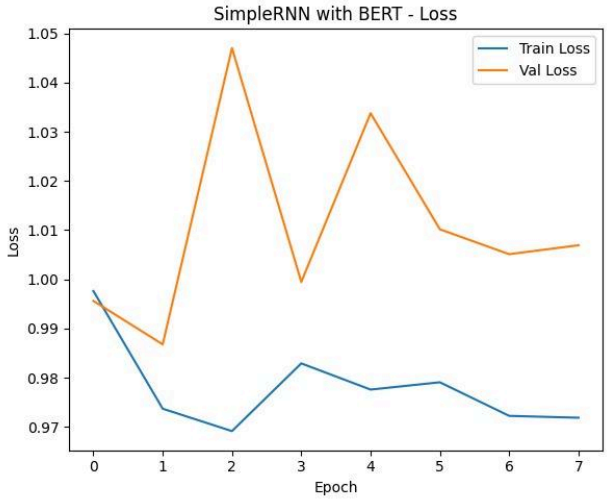

SimpleGRU with BERT - Accuracy
Train Acc
Val Acc
Accuracy
Epoch

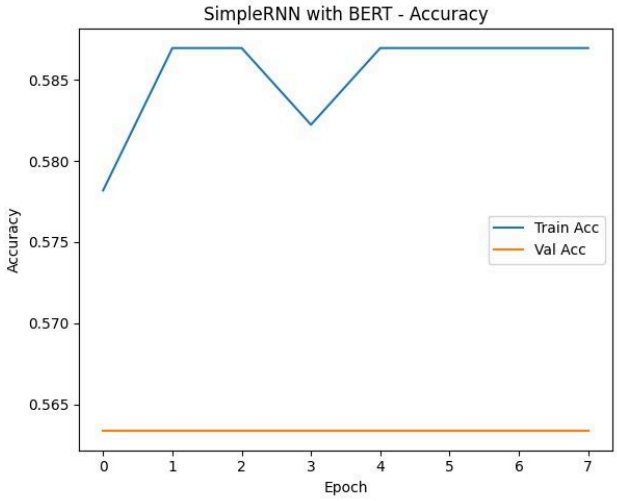

SimpleGRU with FastText - Loss
Train Loss
Val Loss
Loss
Epoch

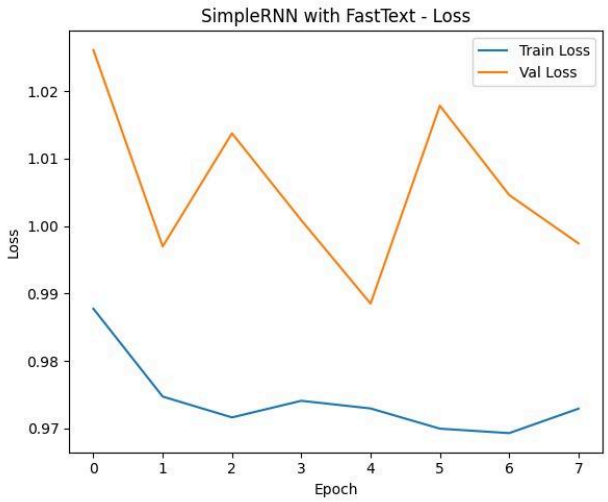

SimpleGRU with FastText - Accuracy
Train Acc
Val Acc
Accuracy
Epoch

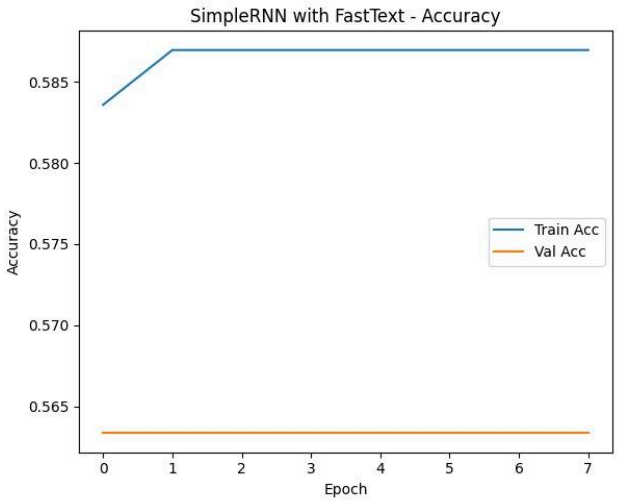

SimpleGRU with GloVe - Loss
Train Loss
Val Loss
Loss
Epoch

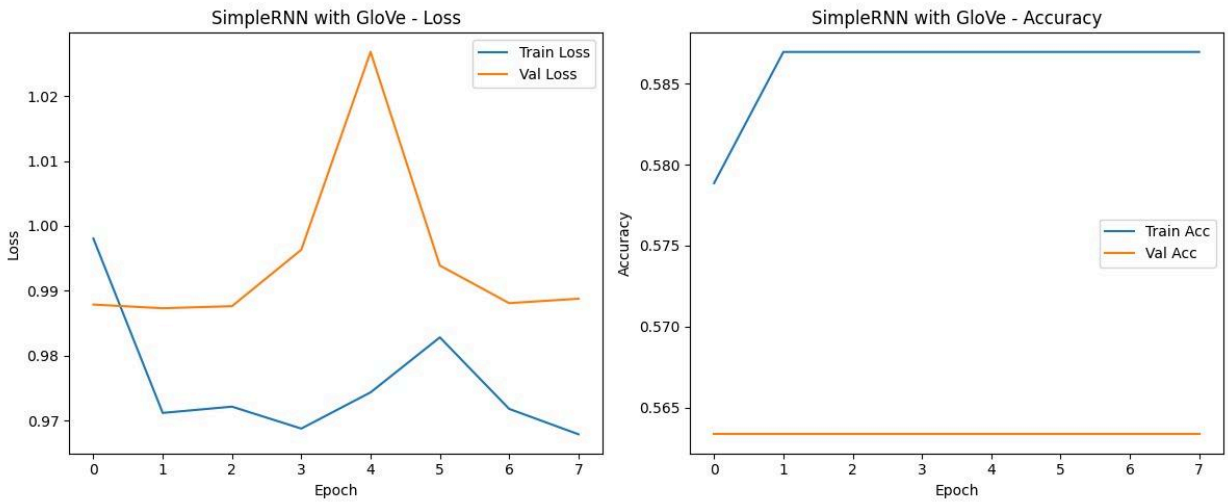

SimpleGRU with GloVe - Accuracy
Train Acc
Val Acc
Accuracy
Epoch


SimpleRNN with BERT - Loss
Train Loss
Val Loss
Loss
Epoch


SimpleRNN with BERT - Accuracy
Train Acc
Val Acc
Accuracy
Epoch


SimpleRNN with FastText - Loss
Train Loss
Val Loss
Loss
Epoch


SimpleRNN with FastText - Accuracy
Train Acc
Val Acc
Accuracy
Epoch



AttentionLSTM Models

Training loss decreases, but validation loss rises early → Overfitting.
BERT embeddings perform best, while FastText and GloVe struggle.
High gap between training & validation accuracy → Poor generalization.

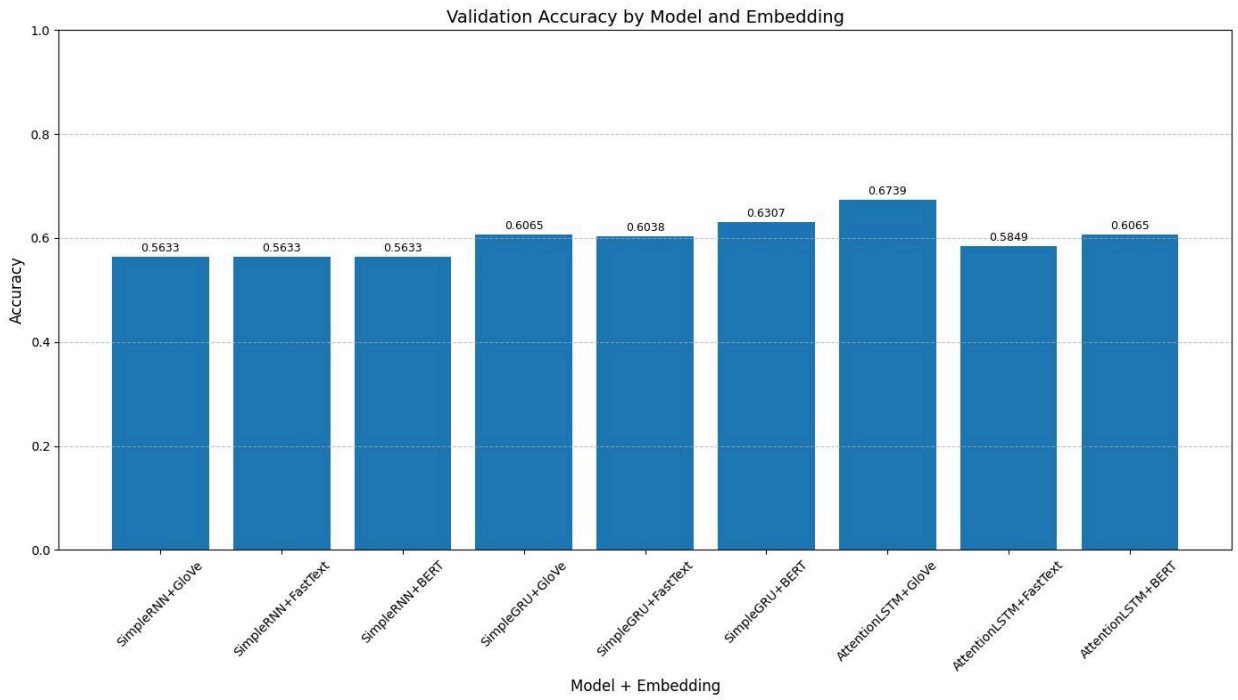
SimpleGRU Models

More stable than LSTM, but still overfits.
BERT outperforms FastText & GloVe, but overall validation accuracy is low.
GRU captures dependencies better than SimpleRNN.

SimpleRNN Models

Fails to learn meaningful patterns → training loss barely decreases.
Validation accuracy remains low (~0.58), showing poor generalization.
No significant difference among embeddings → vanishing gradient issue.

Conclusion:
BERT consistently performs better than FastText and GloVe across all architectures, but still suffers from overfitting.
AttentionLSTM and GRU show better training performance than SimpleRNN, but both struggle with validation performance.
SimpleRNN is ineffective, as it does not learn meaningful patterns from the data.



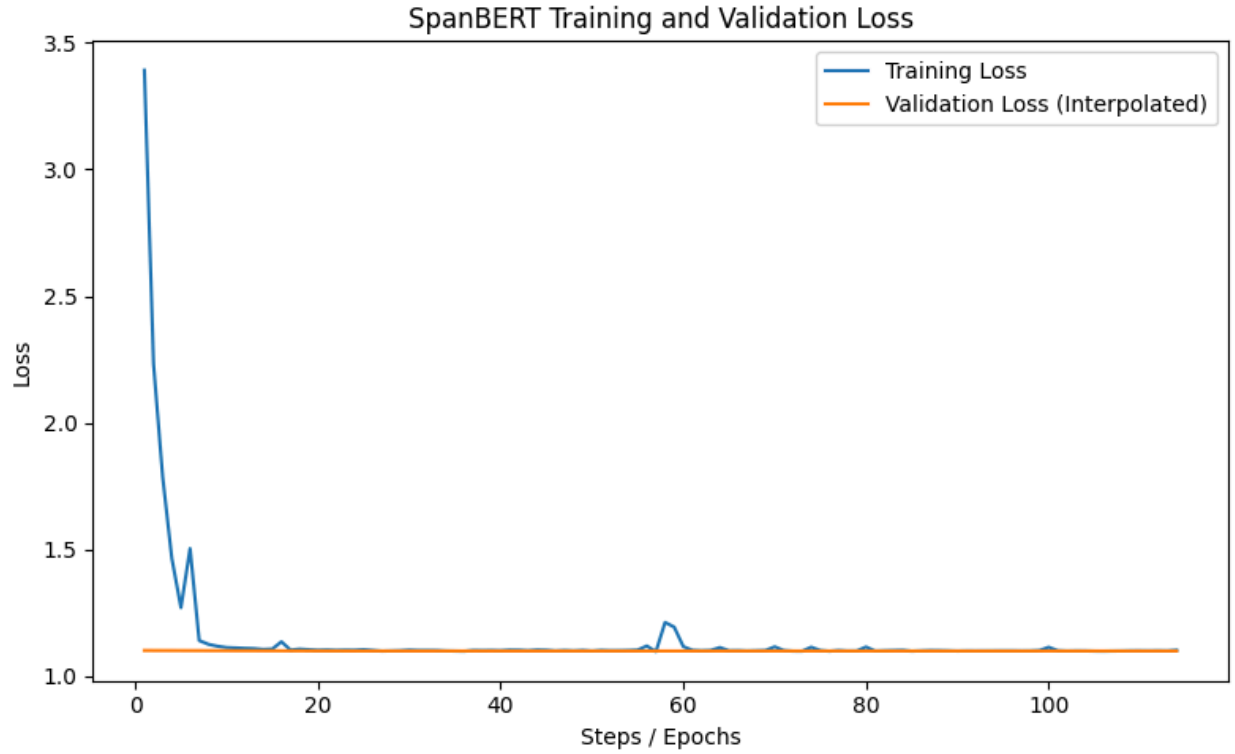
TASK-3

Dataset Description
For this task, we use the Stanford Question Answering Dataset v2 (SQuAD v2), a widely used benchmark dataset for extractive question answering. SQuAD v2 consists of question-answer pairs derived from Wikipedia articles, where some questions have no valid answer in the given context. This makes the task more challenging as the model must also determine when a question is unanswerable.

Total dataset size:
Training samples: 130,319
Validation samples: 11,873
Subset used: Due to computational constraints, a subset of 15,000 samples is used for training.
Preprocessing Steps
Before fine-tuning the SpanBERT model, the dataset undergoes the following preprocessing steps:

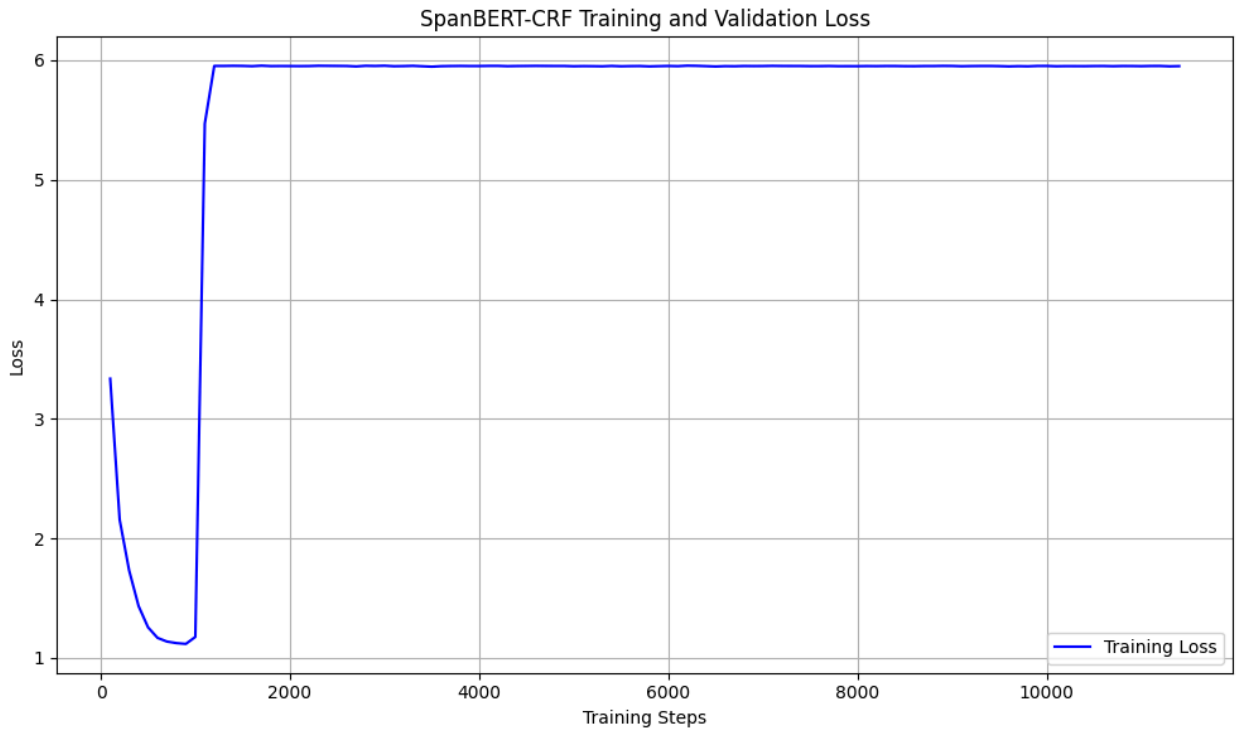
Tokenization: The dataset is tokenized using the SpanBERT tokenizer (spanbert-base-cased). This involves converting text into numerical token IDs, truncating longer sequences, and padding shorter ones to a fixed length.

Handling Overflow: Since long contexts might exceed the model's input limit, a stride-based sliding window approach is used to split text into overlapping chunks (max length: 384 tokens, stride: 128).
Offset Mapping: Each token is mapped to its corresponding character position in the original text to correctly align answer spans.
Answer Alignment:
If the answer exists, its start and end positions in the tokenized text are computed.
If no valid answer exists, the [CLS] token is used as the answer position, training the model to recognize unanswerable questions.




```
Evaluating with exact match on the validation set...

Exact Match (EM) Score: 49.85%
Final EM score on validation set: 49.85%
```




```
Exact Match (EM) Score: 53.78%
```

Justification of model choices and hyperparameters:

Pretrained Transformer-Based Model:

SpanBERT is specifically designed for extractive question answering (QA) tasks, making it suitable for this problem.
It improves upon BERT by enhancing span-level representations, which is beneficial for tasks like SQuAD.

Tokenization Approach:

The Hugging Face tokenizer is used to process the data by applying truncation and padding. The "only_second" truncation strategy ensures that the context (passage) is truncated while keeping the full question.

Hyperparameters:
Preprocessing & Tokenization
max_length=384 & stride=128

- Ensures that long contexts are split into multiple overlapping chunks (to prevent loss of important information).
- Stride of 128 ensures overlap between chunks, reducing information loss.

Training Hyperparameters:

- num_train_epochs=6
  At least 6 epochs are chosen since large transformer models require more training cycles for QA tasks.
  Empirical studies suggest SpanBERT converges well around 4-6 epochs.

- learning_rate=3e-5
  A low learning rate (3e-5) is used because fine-tuning a large transformer model requires small adjustments to pre-trained weights.
  Too high a learning rate can cause instability, while too low can slow down learning.

- per_device_train_batch_size=8
  A batch size of 8 balances memory efficiency and training speed.
  Large transformers like SpanBERT require lower batch sizes due to high memory consumption.

- per_device_eval_batch_size=8
  Keeping the evaluation batch size the same as the training batch size ensures consistency.

- weight_decay=0.01
  L2 regularization to prevent overfitting.

- save_total_limit=2 & load_best_model_at_end=True
  Ensures only the best two models are saved (prevents excessive storage usage).
  Automatically loads the best-performing model at the end.

- evaluation_strategy="epoch" & save_strategy="epoch"
  The model is evaluated and saved at the end of every epoch, ensuring regular performance checks.

- logging_steps=100
  Logs training progress every 100 steps, providing frequent updates.

Comparative analysis of SpanBERT-CRF and SpanBERT.
Performance (Exact Match Score)

- SpanBERT: 49.85%
- SpanBERT-CRF: 53.78% (higher accuracy, better span prediction)

Losses

- SpanBERT: Loss decreases smoothly and stabilizes.
- SpanBERT-CRF: Loss initially decreases but then spikes and plateaus, indicating possible training instability.

Conclusion

- SpanBERT-CRF outperforms SpanBERT in exact match but has training stability challenges. If fine-tuned properly, it can be a better choice for span-based tasks.