NLP ASSIGNMENT -2

Part 1

1. Data Preprocessing

1.1. Tokenization and BIO encoding

1. `tokenize_with_offsets(sentence)->` Splits each sentence into tokens based on space using a simple python `split()` function. Also, this function identifies its character offsets within the original sentence(start and end indices).

2. `bio_encode(sentence, aspects)->` For each sentence we have aspect annotations containing from, to and term. We first tokenize the sentence (using the function above), then initialize a BIO label list (`labels = ["O"] * len(tokens)`). For each aspect: We determine which tokens overlap with the aspect's character span (`aspect['from']` to `aspect['to']`). Then mark the first overlapping token as **B** (begin), and any subsequent overlapping tokens as **I** (inside). Everything not covered by an aspect remains **O** (outside).

1.2. JSON Loading and Saving

`preprocess(input_file, output_file)->` reads a JSON file (like `train.json` or `val.json`), iterates through each entry, applies the `bio_encode` function, and then writes out a new JSON structure (`train_task_1.json`, `val_task_1.json`) that contains tokenized sentences with BIO labels.

2. Model Inputs and Embeddings

2.1. Vocabulary and Embedding Index

- We load GloVe embeddings (100-dimensional) from `glove.6B.100d.txt` and fastText embeddings (300-dimensional) from `cc.en.300.vec`.
- Each file is read into a dictionary (`embeddings_index`) mapping `word -> vector`.
- we then create a `word_index` that maps each word in the embedding vocabulary to an integer ID.

2.2. Embedding Matrix

- The function `create_embedding_matrix` populates a matrix of shape `[vocab_size, embedding_dim]`.
- We are up simply using `torch.tensor(np.array(list(glove.values())))` to build `embedding_matrix_glove` and similarly for fastText.
- These matrices are then used in the PyTorch `Embedding` layer, by calling `nn.Embedding.from_pretrained(embedding_matrix, freeze=False)`, allowing fine-tuning of embeddings.

3. Dataset and DataLoader

3.1. AspectDataset

- Expects a list of preprocessed data (each item has `tokens`, `labels`, etc.), a `word_index`, and a `max_len`.
- For each sample:
    1. Convert tokens into their integer IDs according to `word_index`.
    2. Convert labels **B**, **I**, **O** to numeric encodings (**B** → 1, **I** → 2, **O** → 0).
    3. Truncate/pad both the token IDs and label IDs to `max_len`.
    4. Return the PyTorch tensors plus the original token length.

4. Model Architectures

We trained four models. In each case, the input first goes to an embedding layer initialized with either GloVe or fastText vectors. The output then goes through an RNN or GRU, followed by a final linear layer for classification.

4.1. Common Elements

- **Output dimension:** 3 (the classes **B**, **I**, and **O**).
- **Hidden dimension:** 128.
- **Number of layers:** 2.
- **Dropout:** 0.3 (applied within RNN/GRU if `num_layers > 1`).
- **Loss function:** `CrossEntropyLoss`, applied to the final logits (reshaped to `[batch_size * max_len, output_dim]`).
- **Optimizer:** Adam with `learning_rate=0.001`.

5. Training Process and Loss Plots

We trained each model for 10 epochs. Below is a synthesis of the training/validation loss evolution. Although exact numbers vary slightly, each model generally shows:

- Training Loss starts relatively high and decreases steadily, indicating the network is learning.
- Validation Loss starts lower but does not always decrease at the same rate as training loss (sign of overfitting).

5.2. Plots

- **Blue line (Train Loss):** Decreases rapidly, showing the model is fitting the training set.
- **Orange line (Validation Loss):** Decreases early on but eventually levels out or starts to increase, a common overfitting pattern.
- **GRU_GloVe** achieves the highest F1 (64.44%).
- **RNN_GloVe** attains a slightly higher *non-O accuracy* (71.16%) but a slightly lower F1 (63.82%), largely because of a precision–recall trade-off.
- FastText-based models produce somewhat lower F1 scores (~62% and 60%).
- Overall, **GloVe-based embeddings** appear to help both RNN and GRU perform better on this specific dataset.

6. Best-Performing Model

Based on the F1 measure—which is often the key metric for BIO tagging tasks (due to the importance of handling both precision and recall)—the **GRU_GloVe** model (F1: ~64.44%) is the top performer among the four.

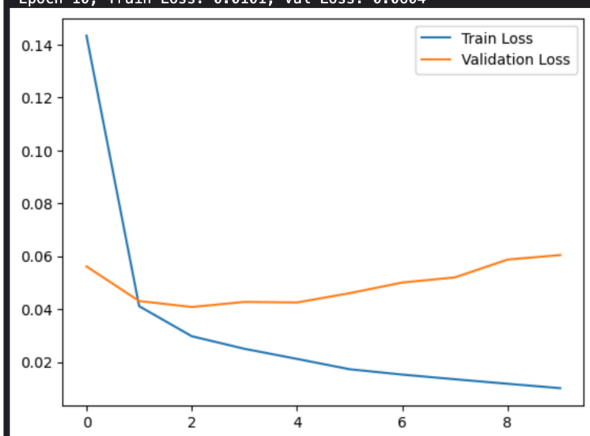
A quick summary of **GRU_GloVe**:

- Embedding: GloVe (100d), fine-tuned.
- Architecture: 2-layer GRU with hidden size 128 and 0.3 dropout.
- Optimizer: Adam (learning rate = 0.001).
- Loss: Cross-entropy across **B**, **I**, **O** classes.

TRAINING AND VALIDATION LOSS PLOTS

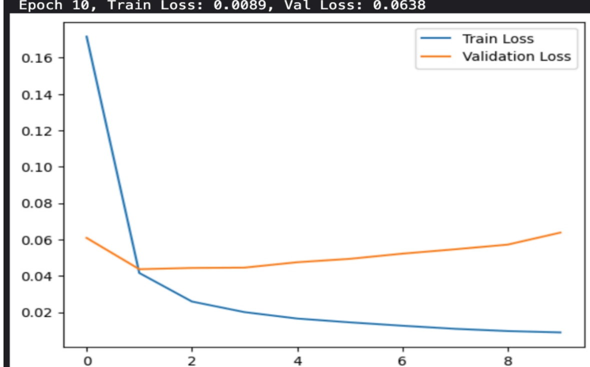
GRAPH 1

```
Training RNN_fastText...
Epoch 1, Train Loss: 0.1433, Val Loss: 0.0561
Epoch 2, Train Loss: 0.0411, Val Loss: 0.0430
Epoch 3, Train Loss: 0.0297, Val Loss: 0.0408
Epoch 4, Train Loss: 0.0250, Val Loss: 0.0427
Epoch 5, Train Loss: 0.0212, Val Loss: 0.0425
Epoch 6, Train Loss: 0.0173, Val Loss: 0.0460
Epoch 7, Train Loss: 0.0152, Val Loss: 0.0500
Epoch 8, Train Loss: 0.0135, Val Loss: 0.0520
Epoch 9, Train Loss: 0.0118, Val Loss: 0.0587
Epoch 10, Train Loss: 0.0101, Val Loss: 0.0604
```



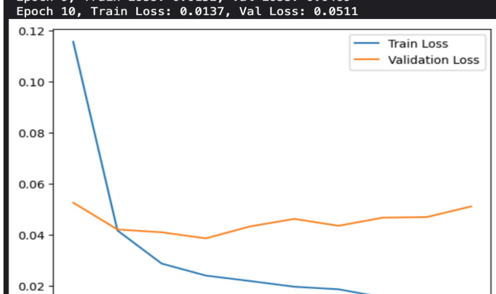
GRAPH 2

```
Training GRU_fastText...
Epoch 1, Train Loss: 0.1716, Val Loss: 0.0608
Epoch 2, Train Loss: 0.0416, Val Loss: 0.0436
Epoch 3, Train Loss: 0.0259, Val Loss: 0.0443
Epoch 4, Train Loss: 0.0200, Val Loss: 0.0445
Epoch 5, Train Loss: 0.0165, Val Loss: 0.0475
Epoch 6, Train Loss: 0.0144, Val Loss: 0.0493
Epoch 7, Train Loss: 0.0126, Val Loss: 0.0522
Epoch 8, Train Loss: 0.0109, Val Loss: 0.0546
Epoch 9, Train Loss: 0.0096, Val Loss: 0.0572
Epoch 10, Train Loss: 0.0089, Val Loss: 0.0638
```



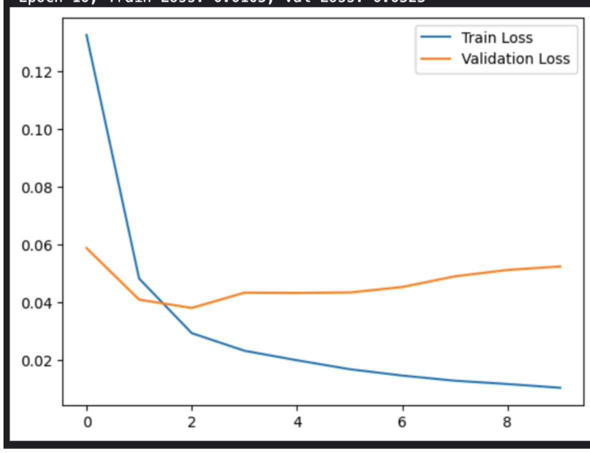
GRAPH 3

```
Training RNN_GloVe...
Epoch 1, Train Loss: 0.1155, Val Loss: 0.0525
Epoch 2, Train Loss: 0.0416, Val Loss: 0.0420
Epoch 3, Train Loss: 0.0287, Val Loss: 0.0409
Epoch 4, Train Loss: 0.0240, Val Loss: 0.0385
Epoch 5, Train Loss: 0.0218, Val Loss: 0.0432
Epoch 6, Train Loss: 0.0196, Val Loss: 0.0462
Epoch 7, Train Loss: 0.0186, Val Loss: 0.0435
Epoch 8, Train Loss: 0.0155, Val Loss: 0.0466
Epoch 9, Train Loss: 0.0152, Val Loss: 0.0469
Epoch 10, Train Loss: 0.0137, Val Loss: 0.0511
```



GRAPH 4

```
Training GRU_GloVe...
Epoch 1, Train Loss: 0.1327, Val Loss: 0.0589
Epoch 2, Train Loss: 0.0483, Val Loss: 0.0410
Epoch 3, Train Loss: 0.0294, Val Loss: 0.0382
Epoch 4, Train Loss: 0.0233, Val Loss: 0.0435
Epoch 5, Train Loss: 0.0200, Val Loss: 0.0434
Epoch 6, Train Loss: 0.0169, Val Loss: 0.0435
Epoch 7, Train Loss: 0.0147, Val Loss: 0.0454
Epoch 8, Train Loss: 0.0129, Val Loss: 0.0491
Epoch 9, Train Loss: 0.0118, Val Loss: 0.0513
Epoch 10, Train Loss: 0.0105, Val Loss: 0.0525
```

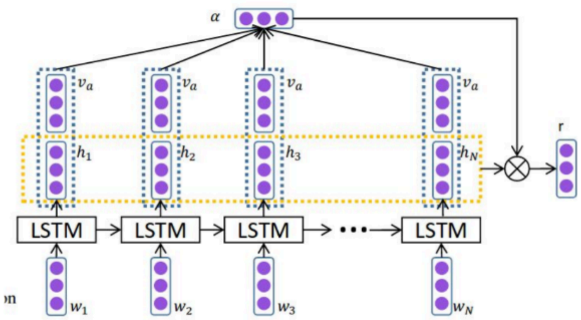



```
<ipython-input-13-e4477553aeb0>:3: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value), which uses the default pickle module implicitly. It is possible to construct malicious pickle data which will execute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more details). In a future release, the default value for `weights_only` will be flipped to `True`. This limits the functions that could be executed during unpickling. Arbitrary objects will no longer be allowed to be loaded via this mode unless they are explicitly allowlisted by the user via `torch.serialization.add_safe_globals`. We recommend you start setting `weights_only=True` for any use case where you don't have full control of the loaded file. Please open an issue on GitHub for any issues related to this experimental feature.
  model.load_state_dict(torch.load(model_path, map_location=device))
Evaluating GRU_GloVe.pth...
processed 4284 tokens with 371 phrases; found: 377 phrases; correct: 241.
accuracy:  69.50%; (non-O)
accuracy:  93.46%; precision:  63.93%; recall:  64.96%; FB1:  64.44
                X: precision:  63.93%; recall:  64.96%; FB1:  64.44  377
Evaluating GRU_fastText.pth...
processed 4284 tokens with 371 phrases; found: 361 phrases; correct: 228.
accuracy:  64.11%; (non-O)
accuracy:  93.07%; precision:  63.16%; recall:  61.46%; FB1:  62.30
                X: precision:  63.16%; recall:  61.46%; FB1:  62.30  361
```

GRAPH 6

```
<ipython-input-13-e4477553aeb0>:3: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value), which uses the default pickle module implicitly. It is possible to construct malicious pickle data which will execute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more details). In a future release, the default value for `weights_only` will be flipped to `True`. This limits the functions that could be executed during unpickling. Arbitrary objects will no longer be allowed to be loaded via this mode unless they are explicitly allowlisted by the user via `torch.serialization.add_safe_globals`. We recommend you start setting `weights_only=True` for any use case where you don't have full control of the loaded file. Please open an issue on GitHub for any issues related to this experimental feature.
  model.load_state_dict(torch.load(model_path, map_location=device))
Evaluating RNN_GloVe.pth...
processed 4284 tokens with 371 phrases; found: 403 phrases; correct: 247.
accuracy:  71.16%; (non-O)
accuracy:  93.30%; precision:  61.29%; recall:  66.58%; FB1:  63.82
                X: precision:  61.29%; recall:  66.58%; FB1:  63.82  403
Evaluating RNN_fastText.pth...
processed 4284 tokens with 371 phrases; found: 384 phrases; correct: 229.
accuracy:  64.32%; (non-O)
accuracy:  92.69%; precision:  59.64%; recall:  61.73%; FB1:  60.66
                X: precision:  59.64%; recall:  61.73%; FB1:  60.66  384
```

TASK-2



When switching from a unidirectional LSTM to a bidirectional LSTM (BiLSTM) in our Attention-Based LSTM model, the key difference is that now the model processes the input both forward and backward. Instead of just relying on past words, each word now has context from both previous and future words, making the representations more informative.

Since BiLSTM doubles the hidden size, adjustments are made in:

- **The final hidden state (`h_N`), where the last forward and backward hidden states are concatenated.**
- **The attention mechanism, which now attends over bidirectional hidden states.**
- **The projection layers, ensuring they handle the increased dimensionality.**

Overall, using BiLSTM improves the model's ability to capture dependencies in Aspect-Based Sentiment Analysis (ABSA), especially when context matters for determining sentiment.

The preprocessing pipeline for Aspect-Based Sentiment Analysis (ABSA) follows these steps:

1. **Setup Paths**
    - **Defines directories for dataset storage and processed output.**
2. **Load Tokenizer & Model**
    - **Loads the BERT tokenizer and model to process input text.**

3. Text Cleaning & Tokenization
    - Removes punctuation from sentences.
    - Tokenizes sentences and aspect terms.
4. Data Formatting
    - Reads raw JSON data.
    - Converts each sentence to lowercase.
    - Extracts aspect terms, polarity, and computes aspect positions.
    - Stores formatted data in a structured JSON format.
5. Preprocessing Execution
    - Runs the above steps for training and validation datasets.
6. Dataset Creation
    - Constructs a PyTorch `Dataset` class to:
        - Tokenize and encode sentences/aspects.
        - Assign polarity labels.
        - Apply padding and truncation.
7. DataLoader Creation
    - Wraps the dataset into a PyTorch `DataLoader` for efficient batch processing.

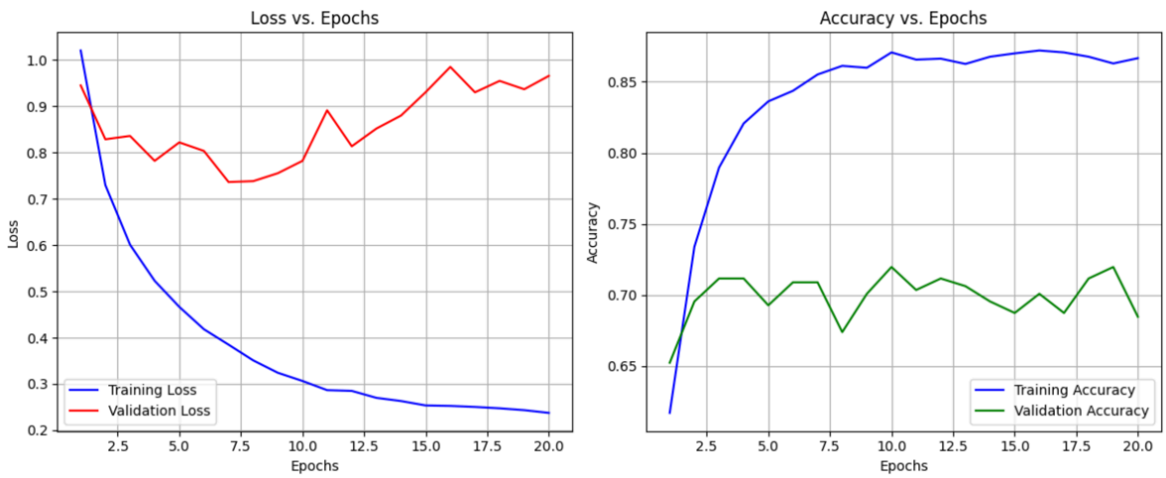
Model Parameters

- Embedding Model: BERT (`bert-base-uncased`)
- Hidden Dimension: `128` (LSTM hidden state size)
- Number of Layers: `2` (stacked LSTMs)
- Bidirectional LSTM: `True` (captures both forward and backward context)
- Dropout: `0.4` (for regularization)

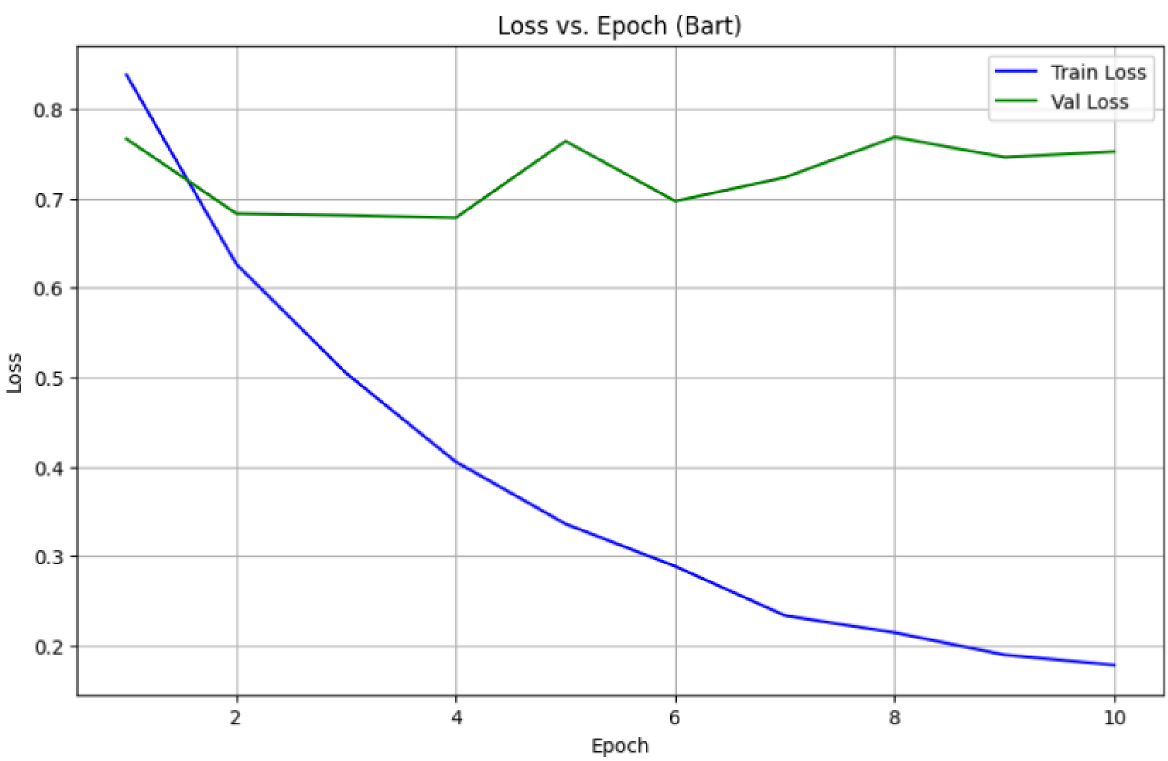
Training Parameters

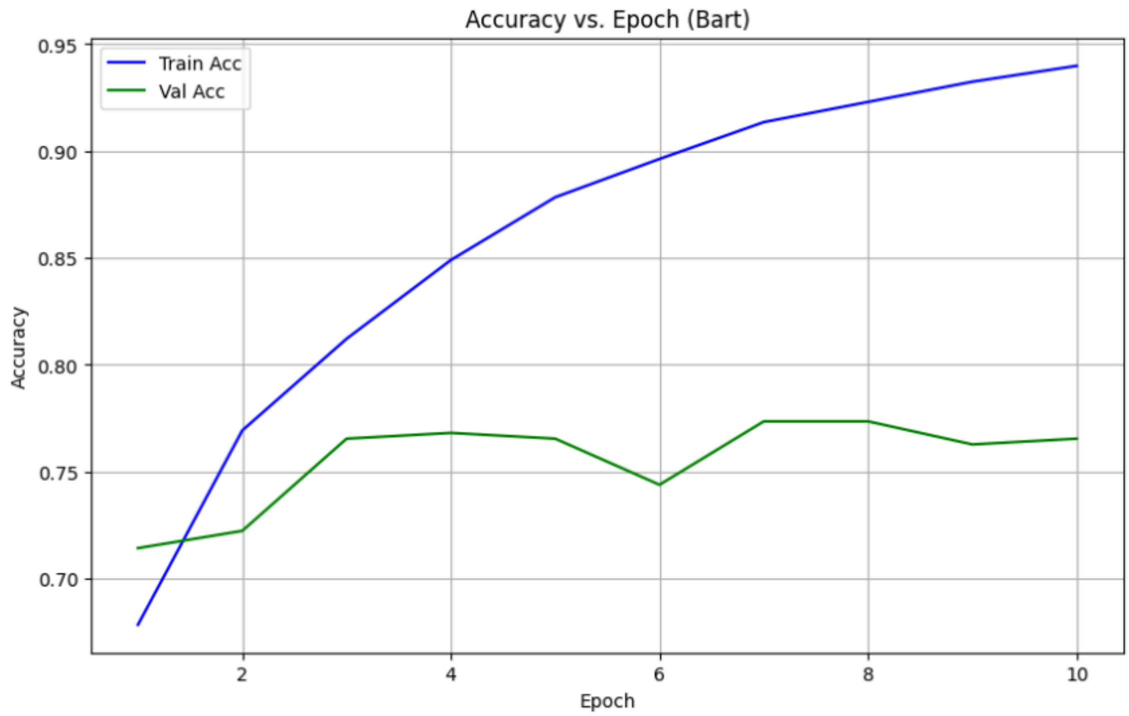
- Batch Size: `32` (for both training and validation)
- Optimizer: `AdamW` (adaptive learning rate optimization)
- Learning Rate: `2e-5` (fine-tuned for transformer-based models)
- Loss Function: `CrossEntropyLoss` (for multi-class sentiment classification)
- Number of Epochs: `20` (sufficient for convergence without overfitting)

For custom-architecture:

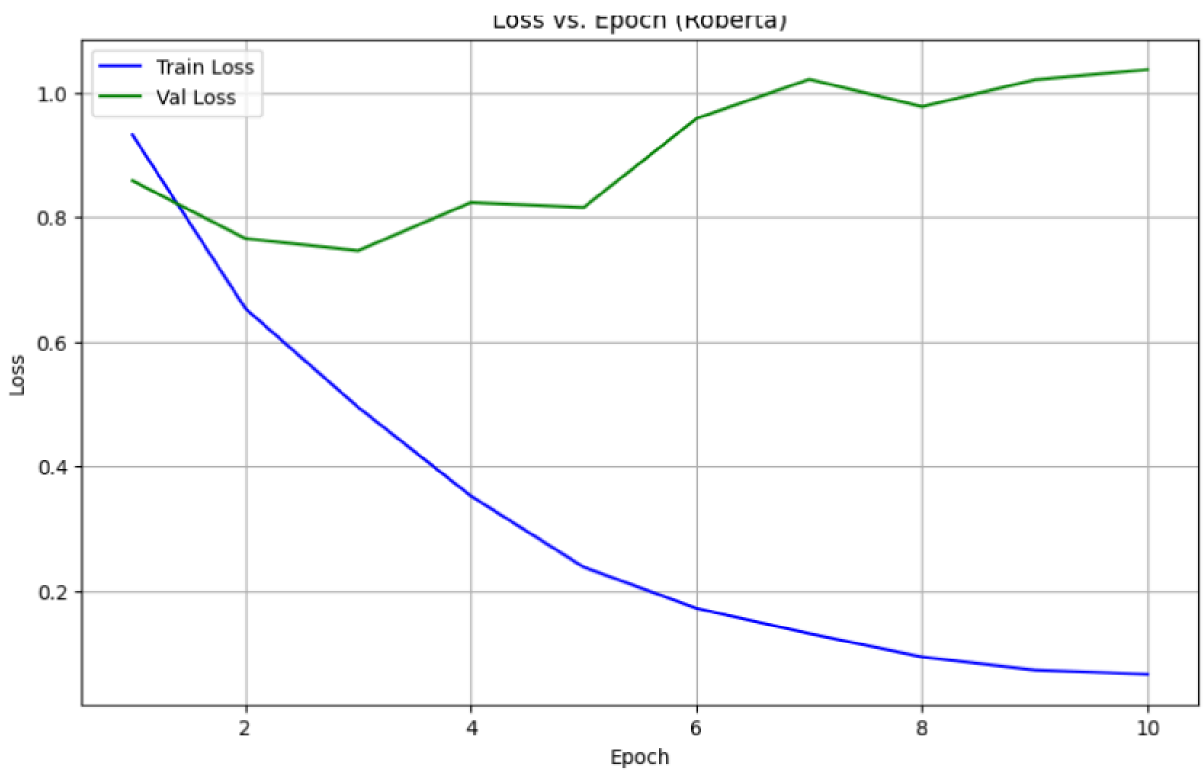


For BERT , BART , ROBERTA:

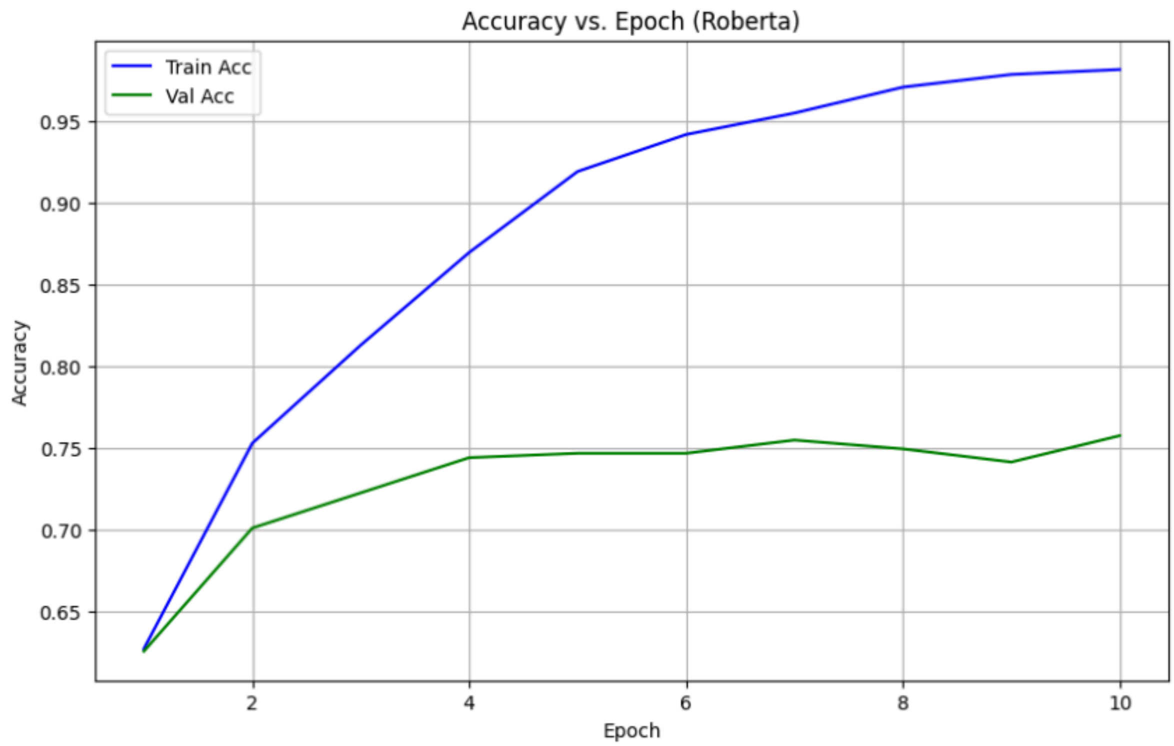


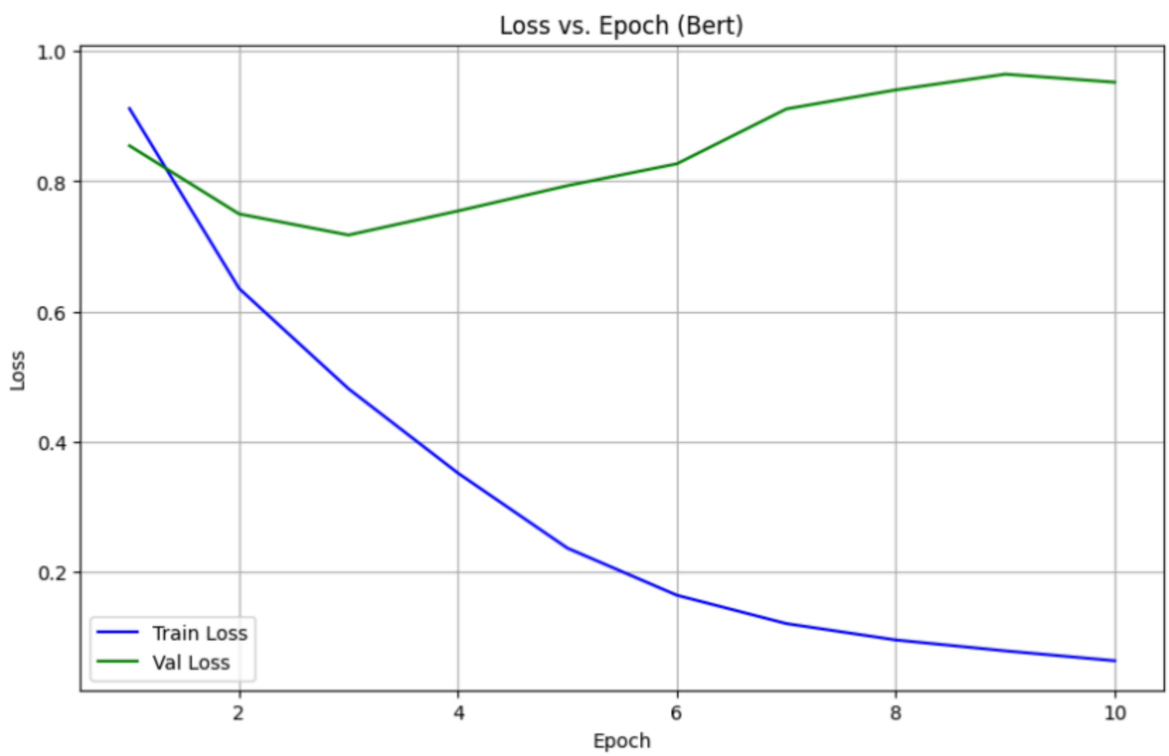


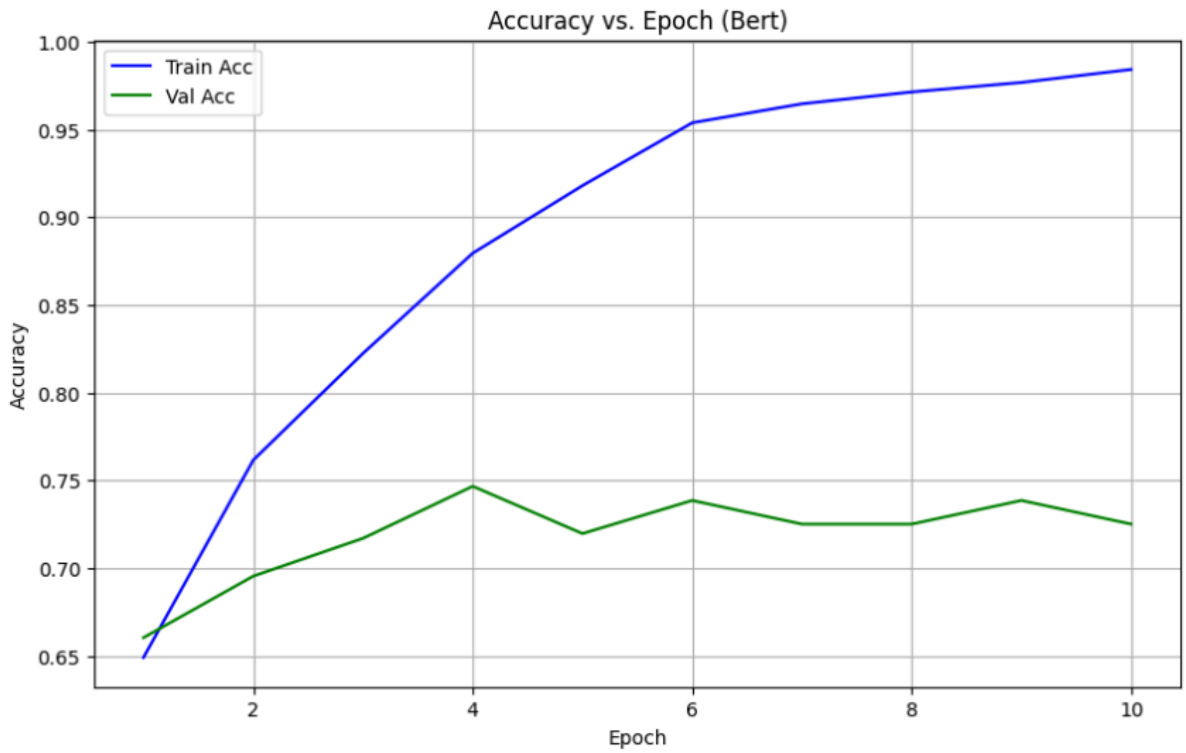
Accuracy vs. Epoch (Bart)
Train Acc
Val Acc
Accuracy
Epoch
0.95
0.90
0.85
0.80
0.75
0.70
2
4
6
8
10



Loss vs. Epoch (Roberta)
Train Loss
Val Loss
Loss
Epoch
1.0
0.8
0.6
0.4
0.2
2
4
6
8
10


Accuracy vs. Epoch (Roberta)
Train Acc
Val Acc
Accuracy
0.95
0.90
0.85
0.80
0.75
0.70
0.65
2
4
6
8
10
Epoch


Loss vs. Epoch (Bert)
Train Loss
Val Loss
Loss
1.0
0.8
0.6
0.4
0.2
2
4
6
8
10
Epoch



For custom architecture:

```
Epoch 1/20 | Train Loss: 1.0202, Train Acc: 0.6170 | Val Loss: 0.9447, Val Acc: 0.6523
Epoch 2/20 | Train Loss: 0.7295, Train Acc: 0.7339 | Val Loss: 0.8284, Val Acc: 0.6954
Epoch 3/20 | Train Loss: 0.6008, Train Acc: 0.7896 | Val Loss: 0.8356, Val Acc: 0.7116
Epoch 4/20 | Train Loss: 0.5228, Train Acc: 0.8207 | Val Loss: 0.7818, Val Acc: 0.7116
Epoch 5/20 | Train Loss: 0.4665, Train Acc: 0.8362 | Val Loss: 0.8217, Val Acc: 0.6927
Epoch 6/20 | Train Loss: 0.4181, Train Acc: 0.8436 | Val Loss: 0.8033, Val Acc: 0.7089
Epoch 7/20 | Train Loss: 0.3848, Train Acc: 0.8551 | Val Loss: 0.7361, Val Acc: 0.7089
Epoch 8/20 | Train Loss: 0.3509, Train Acc: 0.8612 | Val Loss: 0.7380, Val Acc: 0.6739
Epoch 9/20 | Train Loss: 0.3241, Train Acc: 0.8598 | Val Loss: 0.7552, Val Acc: 0.7008
Epoch 10/20 | Train Loss: 0.3064, Train Acc: 0.8707 | Val Loss: 0.7818, Val Acc: 0.7197
Epoch 11/20 | Train Loss: 0.2866, Train Acc: 0.8656 | Val Loss: 0.8910, Val Acc: 0.7035
Epoch 12/20 | Train Loss: 0.2850, Train Acc: 0.8663 | Val Loss: 0.8133, Val Acc: 0.7116
Epoch 13/20 | Train Loss: 0.2700, Train Acc: 0.8625 | Val Loss: 0.8516, Val Acc: 0.7062
Epoch 14/20 | Train Loss: 0.2631, Train Acc: 0.8676 | Val Loss: 0.8797, Val Acc: 0.6954
Epoch 15/20 | Train Loss: 0.2536, Train Acc: 0.8700 | Val Loss: 0.9302, Val Acc: 0.6873
Epoch 16/20 | Train Loss: 0.2526, Train Acc: 0.8720 | Val Loss: 0.9847, Val Acc: 0.7008
Epoch 17/20 | Train Loss: 0.2503, Train Acc: 0.8707 | Val Loss: 0.9299, Val Acc: 0.6873
Epoch 18/20 | Train Loss: 0.2474, Train Acc: 0.8676 | Val Loss: 0.9545, Val Acc: 0.7116
Epoch 19/20 | Train Loss: 0.2434, Train Acc: 0.8629 | Val Loss: 0.9365, Val Acc: 0.7197
Epoch 20/20 | Train Loss: 0.2375, Train Acc: 0.8666 | Val Loss: 0.9654, Val Acc: 0.6846
```

For BART:

```
Epoch 1: Train Loss=0.8380, Train Acc=0.6785 | Val Loss=0.7664, Val Acc=0.7143
Epoch 2: Train Loss=0.6267, Train Acc=0.7693 | Val Loss=0.6830, Val Acc=0.7224
Epoch 3: Train Loss=0.5047, Train Acc=0.8122 | Val Loss=0.6810, Val Acc=0.7655
Epoch 4: Train Loss=0.4056, Train Acc=0.8490 | Val Loss=0.6782, Val Acc=0.7682
Epoch 5: Train Loss=0.3361, Train Acc=0.8784 | Val Loss=0.7638, Val Acc=0.7655
Epoch 6: Train Loss=0.2888, Train Acc=0.8963 | Val Loss=0.6968, Val Acc=0.7439
Epoch 7: Train Loss=0.2339, Train Acc=0.9135 | Val Loss=0.7233, Val Acc=0.7736
Epoch 8: Train Loss=0.2147, Train Acc=0.9230 | Val Loss=0.7683, Val Acc=0.7736
Epoch 9: Train Loss=0.1899, Train Acc=0.9325 | Val Loss=0.7458, Val Acc=0.7628
Epoch 10: Train Loss=0.1785, Train Acc=0.9399 | Val Loss=0.7522, Val Acc=0.7655
```

For ROBERTA:

```
Epoch 1: Train Loss=0.9322, Train Acc=0.6268 | Val Loss=0.8586, Val Acc=0.6253
Epoch 2: Train Loss=0.6534, Train Acc=0.7528 | Val Loss=0.7658, Val Acc=0.7008
Epoch 3: Train Loss=0.4946, Train Acc=0.8129 | Val Loss=0.7463, Val Acc=0.7224
Epoch 4: Train Loss=0.3520, Train Acc=0.8696 | Val Loss=0.8238, Val Acc=0.7439
Epoch 5: Train Loss=0.2375, Train Acc=0.9193 | Val Loss=0.8159, Val Acc=0.7466
Epoch 6: Train Loss=0.1714, Train Acc=0.9419 | Val Loss=0.9587, Val Acc=0.7466
Epoch 7: Train Loss=0.1308, Train Acc=0.9551 | Val Loss=1.0216, Val Acc=0.7547
Epoch 8: Train Loss=0.0933, Train Acc=0.9710 | Val Loss=0.9777, Val Acc=0.7493
Epoch 9: Train Loss=0.0722, Train Acc=0.9787 | Val Loss=1.0209, Val Acc=0.7412
Epoch 10: Train Loss=0.0653, Train Acc=0.9818 | Val Loss=1.0373, Val Acc=0.7574
```

For BERT:

```
Epoch 1: Train Loss=0.9117, Train Acc=0.6491 | Val Loss=0.8546, Val Acc=0.6604
Epoch 2: Train Loss=0.6354, Train Acc=0.7616 | Val Loss=0.7499, Val Acc=0.6954
Epoch 3: Train Loss=0.4813, Train Acc=0.8224 | Val Loss=0.7174, Val Acc=0.7170
Epoch 4: Train Loss=0.3518, Train Acc=0.8794 | Val Loss=0.7544, Val Acc=0.7466
Epoch 5: Train Loss=0.2365, Train Acc=0.9179 | Val Loss=0.7933, Val Acc=0.7197
Epoch 6: Train Loss=0.1642, Train Acc=0.9537 | Val Loss=0.8269, Val Acc=0.7385
Epoch 7: Train Loss=0.1206, Train Acc=0.9645 | Val Loss=0.9111, Val Acc=0.7251
Epoch 8: Train Loss=0.0955, Train Acc=0.9713 | Val Loss=0.9403, Val Acc=0.7251
Epoch 9: Train Loss=0.0788, Train Acc=0.9767 | Val Loss=0.9645, Val Acc=0.7385
Epoch 10: Train Loss=0.0636, Train Acc=0.9841 | Val Loss=0.9521, Val Acc=0.7251
```

TASK 3

Fine tuning spanbert and spanbert-crf

1. **Dataset description**

   - The squad v2 data is a question answering dataset , typical examples from squad v2 comprises contexts , questions and answers.

- The data set comprises squad 1 dataset along with the questions about (50k) that don't have any answers(is impossible flag is set true for this and the answer[text] is an empty list).
- The answer in squad v2 is a dictionary with key as **text** (that contains the answer) and **answer_start**(with value the starting of the answer in the text.)
- Since the training data set comprises around 130k question training and fine tuning would be very difficult due to lack of gpu .
- In this assignment I have used only 15k train samples and the whole validation set around (12k) samples.

2. Preprocessing steps

For spanbert

The dataset is loaded using the load_dataset function from the datasets library.

The preprocessing function is applied to the training and validation datasets using the map method that ensures that the entire dataset is processed in batches, and unnecessary columns are removed to save memory.

The preprocessing of the data involves many process ;

1. **Cleaning Questions**
    - This involves removing any leading whitespace from the questions to ensure consistency in the input data.this also helps to remove unnecessary white space which can add noise and affect models performance.

      For example : **“ how are you, my teaching assistant?”** after cleaning would be like **“how are you, my teaching assistant?”**
2. **tokenize_inputs**
    - This function converts the context and the correspon ding questions into tokens .In this assignment i have used Spanbert-base-model tokenizer to tokenise.
    - It handles **truncation** :If the combined length of the question and context exceeds the max_length (384 tokens), then the context is truncated. The truncation strategy is set to **"only_second"** that means the context (second part of the input) is truncated, not the question.
    - It handles **stride** : doc_stride(128) for long contexts ,this allows to split the context into overlapping chunks so that no information is lost.

- In order to have a uniform sequence length **padding(max length 384)** is also done.
- It also generates an **offset mapping** that records the start and the end position of each token in the original text for locating the answer in the context.

3. **modify_offset_mapping:**
    - Since the offset mapping is a list of tuple indicating the start and the end position of each character in the original text . since the there are token such as **CLS and SEP** that not the p[art of the context neither for questions . To focus only on the context tokens, the offset mapping is modified. Tokens that are not part of the context (i.e., tokens with sequence_id != 1 special tokens) are assigned a (0, 0) offset, effectively ignoring them.
4. **Find_answer_positions:**
    - This function locates the start and the ending token position of the answer in the tokenized input.
    - **Context Span:** The start and end indices of the context in the tokenized sequence are identified. This ensures that the answer lies within the context span.
    - **Answer Validation:** If the answer's character positions fall outside the context span, the answer is considered invalid, and the start and end positions are set to (0, 0).
    - **Token Position Calculation**: If the answer is valid, the exact start and end token positions are calculated by iterating through the offset mapping and comparing it with the answer's character positions.
5. **process _example:** this function stores the processed data , including the example ids ,start and ending position of the answer and add it to the tokenized inputs .

For spanbert-crf model

The preprocessing steps are same as spanbert except the labels are added using the bio encoding on the answer span in the context ,that are used for crf layer .B represents beginning of the answer , I is used for the other token for answer and O is used for other tokens than answer span.

Justification of the model choices and hyperparameters

- For this assignment I have used spanBert-base-cased model as the pretrained model as it has predefined architecture for question answering task.Unlike traditional BERT, which is trained on random token masking.
- SpanBERT introduces span masking and span boundary objective during pre-training i.e. the model is explicitly trained to predict contiguous spans of text, making it highly effective for tasks like extracting answers from passages in SQuAD v2.
- For choosing the best hyperparameters i have performed grid search .the screenshot for the same ;

```
# Define hyperparameter search space
learning_rates = [2e-7,3e-5, 5e-5]
batch_sizes = [8, 16, 32]
weight_decays = [0.01, 0.1, 0.005]
warmup_steps = [0, 100, 200]
optimizers = ["AdamW", "SGD", "ADAM"]

# Generate all possible hyperparameter combinations
hyperparameter_combinations = list(product(learning_rates, batch_sizes, weight_decays, warmup_steps, optimizers))
print(f"Total Hyperparameter Combinations: {len(hyperparameter_combinations)}")
```

For finding the hyper parameter i took the subset of 5k training example and 2k validation example and after 3 epochs I looked at the training and validation loss along with the em score . After the grid search for the spanbert, the learning rate = 2e-7 , optimizer =”adamW” , warmup_steps=200 and the weight decay =0.01 comes to the best parameters. I have fine tuned the model on this .

The model choices for spanbert and spanbert-crf are:

```
training_args = TrainingArguments(
    output_dir="./spanbert_qa_gpu",
    evaluation_strategy="epoch",
    learning_rate=2e-7,
    num_train_epochs=6,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    weight_decay=0.01,
    logging_steps=200,
    save_strategy="epoch",

    fp16=True,
)
```

evaluation_strategy="epoch": This ensures that the model is evaluated on the validation set at the end of each epoch to helps in identifying overfitting or underfitting early in the training process.

learning_rate=2e-7: The learning rate is set to a small value (2e-7) because fine-tuning a large pretrained model like SpanBERT requires careful adjustment of weights,ensuring stable convergence and avoiding overshooting the optimal weights which are particularly important for QA tasks.the model learns if high l_r is chosen the model does not learn and underfits.

Epoch : i have chosen 6 epoch for monitoring the models performance as fine tuning of transformers requires a balance between underfitting and overfitting. And these models usually overfits after 6 -7 epochs or even before 6 .so 6 is the optimal choice for having the idea of models performance.

per_device_train_batch_size=8: A batch size of 8 is chosen to balance memory usage and training stability. Larger batch sizes can speed up training but require more GPU memory. i have used the same for the evaluation also to avoids out-of-memory errors during evaluation.

weight_decay=0.01

Weight decay is a form of regularization that prevents overfitting by penalizing large weights. A value of 0.01 is used in fine-tuning tasks to ensure the model generalizes well to unseen data without overfitting.

logging_steps=200

I have Logged at every 200 steps to provides frequent updates on the training progress without overwhelming the logs and to see the convergence and divergence.

save_strategy="epoch"

Saved the model at the end of each epoch to ensures that checkpoints are available for later use, such as resuming training or selecting the best-performing model.

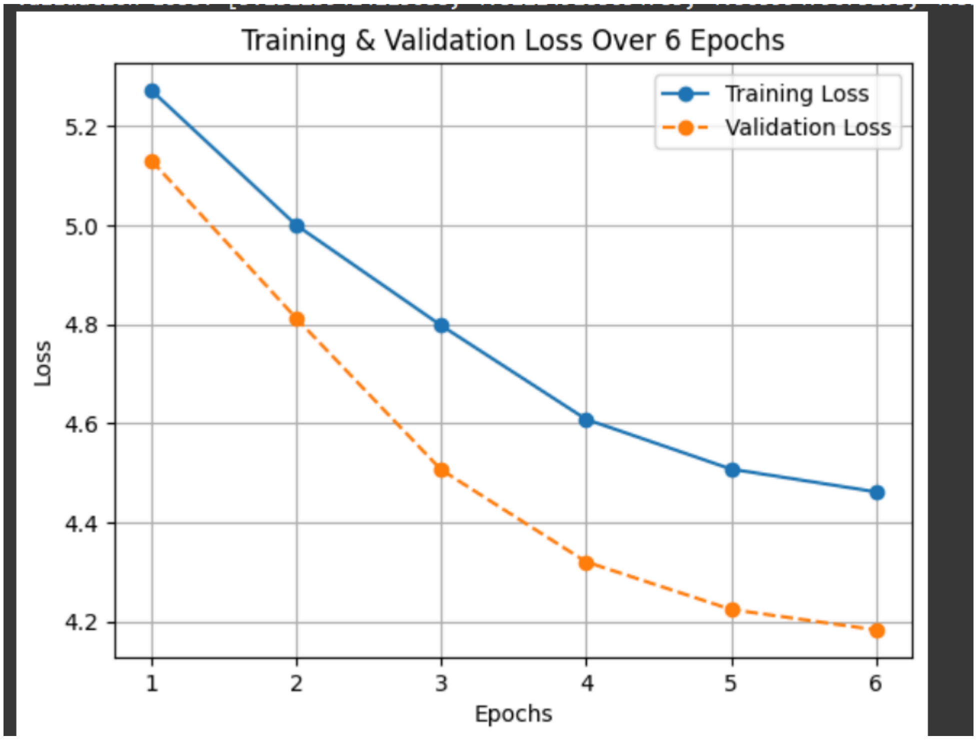
fp16=True

Enabled mixed-precision training (fp16) to reduces memory usage and speeds up training by using 16-bit floating-point numbers instead of 32-bit on gpu.

In CRF loss: reduction='mean' is used for normalizing the loss as crf loss would vary significantly depending on the batch size, making it harder to compare results or tune hyperparameters.this insures that the loss is normalized with respect to the batch size and the sequence length, gradient updates are proportional to the average error across the batch. This leads to more stable training dynamics compared to summing the loss, which could result in excessively large gradients for larger batches.

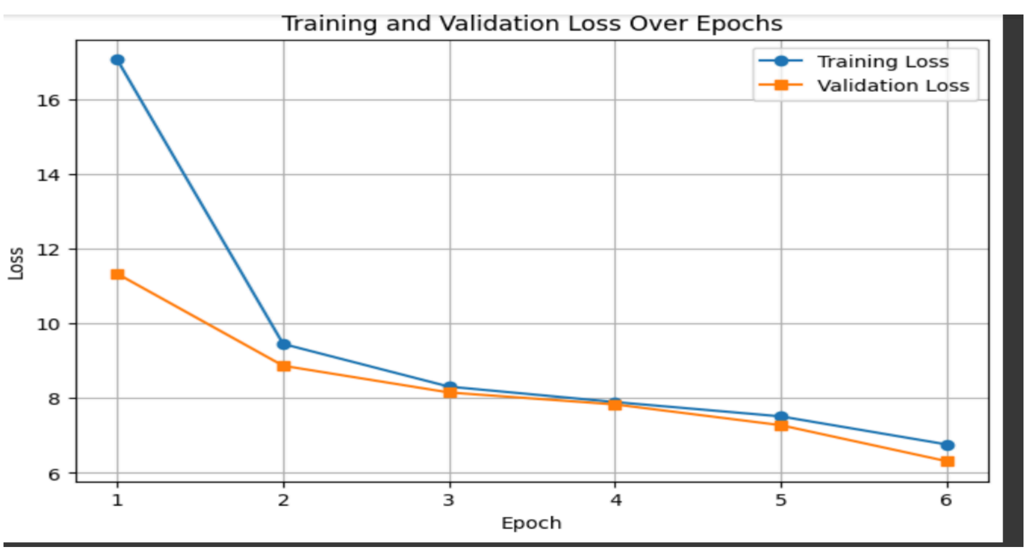
TRAINING AND THE VALIDATION PLOTS FOR SPANBERT AND THE SPANBERT -CRF

For spanbert



The model generalizes well as the training and the validation loss decreases with epoch. Suggesting the model works pretty well on the inseen data.A small gap between training and validation loss suggests good generalization.

For spanbert -crf



Decreasing Trend: Both training and validation loss are consistently decreasing, which suggests that the model is learning effectively.and generalizes on the unseen data.
Initially, the training loss is much higher than the validation loss, but they converge as training progresses, with each epoch the loss decreases gradually, indicating that the model is stabilizing.

Comparative analysis on spanbert and spanbert-crf

1. Performance Comparison

- **Loss Reduction:**
    - Both models show a steady decline in training and validation loss over epochs.
    - However, **SpanBERT-CRF starts with a higher loss** but **drops more significantly**, indicating better learning capacity.
- **Validation Loss & Overfitting:**
    - In SpanBERT, training and validation loss decrease in sync, but there's still a small gap.
    - In SpanBERT-CRF, the gap between training and validation loss is narrower, suggesting **better generalization** and less overfitting.
- **Exact Match (EM) Score:**
    - SpanBERT-CRF achieves a higher EM score (66.23%) compared to SpanBERT (47.99%).
    - This suggests that the **CRF layer helps in capturing structured dependencies** in the answer spans, leading to better extraction accuracy.

2. Impact of BIO Encoding in SpanBERT-CRF

- The **BIO tagging scheme helps in better structuring answer spans**:
    - **'B' (Beginning)** ensures the start of an answer is explicitly marked.
    - **'I' (Inside)** maintains continuity in multi-token answers.
    - **'O' (Outside)** avoids irrelevant text being classified as part of the answer.
- This structured labeling helps **CRF learn dependencies across tokens**, whereas vanilla SpanBERT relies on independent token predictions.

3. Training Efficiency

- The training time for SpanBERT-CRF is slightly higher due to the additional CRF computations.
- However, the improved generalization and **higher exact match accuracy** justify the added complexity.

Conclusion

- SpanBERT-CRF outperforms standard SpanBERT in question answering tasks.
- The CRF layer enhances the span predictions by accounting the modeling token dependencies,eventually leading to a better alignment with true answer spans.
- For structured tasks like answer span extraction, adding a CRF with suitable hyperparameters significantly improves accuracy without overfitting.

Screenshots for EM scores.

For spanbert

```
Using device: cuda
/usr/local/lib/python3.11/dist-packages/transformers/training_args.py:1575: FutureWarning: `evaluation_strategy` is d
  warnings.warn(
Some weights of BertForQuestionAnswering were not initialized from the model checkpoint at SpanBERT/spanbert-base-cas
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
<ipython-input-3-a91f926569d7>:168: FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0 for
  trainer = Trainer(
```

[11484/11484 44:20, Epoch 6/6]

Epoch	Training Loss	Validation Loss	Exact Match
1	5.273000	5.131239	47.805946
2	5.000500	4.812249	47.713299
3	4.797600	4.505995	47.881749
4	4.608000	4.319988	47.831214
5	4.507400	4.223758	47.999663
6	4.461400	4.182481	47.848059

For spanbert-crf

```
training the model
Epoch 1: 100%|██████████| 1912/1912 [20:26<00:00,  1.56it/s]
Validation: 100%|██████████| 1538/1538 [06:21<00:00,  4.03it/s]
Epoch 1
Train Loss: 17.0643 | Val Loss: 11.3318 | EM Score: 49.05%
Epoch 2: 100%|██████████| 1912/1912 [20:06<00:00,  1.58it/s]
Validation: 100%|██████████| 1538/1538 [06:15<00:00,  4.10it/s]
Epoch 2
Train Loss: 9.4521 | Val Loss: 8.8681 | EM Score: 54.57%
Epoch 3: 100%|██████████| 1912/1912 [20:01<00:00,  1.59it/s]
Validation: 100%|██████████| 1538/1538 [06:17<00:00,  4.07it/s]
Epoch 3
Train Loss: 8.3085 | Val Loss: 8.1521 | EM Score: 59.08%
Epoch 4: 100%|██████████| 1912/1912 [23:32<00:00,  1.35it/s]
Validation: 100%|██████████| 1538/1538 [07:25<00:00,  3.45it/s]
Epoch 4
Train Loss: 7.8959 | Val Loss: 7.8336 | EM Score: 61.49%
Epoch 5: 100%|██████████| 1912/1912 [20:18<00:00,  1.57it/s]
Validation: 100%|██████████| 1538/1538 [06:27<00:00,  3.97it/s]
Epoch 5
Train Loss: 7.5123 | Val Loss: 7.2766 | EM Score: 65.17%
Epoch 6: 100%|██████████| 1912/1912 [20:14<00:00,  1.57it/s]
Validation: 100%|██████████| 1538/1538 [06:23<00:00,  4.01it/s]
Epoch 6
Train Loss: 6.7611 | Val Loss: 6.3127 | EM Score: 66.23%

Training complete!.
```

Contribution

Task 1 : Ram dabas 2021275

Task2 : kartik prasad 2022240

Task3 : ayaan hasan 2022121