CSE556: Natural Language Processing
Assignment-2

Himanshu Raj (2022216) | Ishita (2022224) | Ritika Thakur (2022408)

March 16, 2025

1 Task 1

1.1 Preprocessing Steps

The preprocessing pipeline consists of the following steps:

- **Data Loading:** We load the raw JSON data (provided in `train.json` and `val.json`) containing sentences, annotated aspect terms, and sentiment polarities.
- **Tokenization:** We use NLTK's `word_tokenize` function for tokenization. This ensures that punctuation and special characters are handled correctly.
- **BIO Tagging:** For each sentence, we initialize all token labels as "O". Then, for each annotated aspect term, we determine the token span by comparing character offsets and assign a "B" label to the beginning token and "I" labels to subsequent tokens. This is done using the 'from' and 'to' indices given in the input files.
  Using this method instead of directly comparing tokens helps handling of misspelled or unknown tokens.
- **Output Format:** The preprocessed data is stored in a JSON file (e.g., `train_task_1.json` and `val_task_1.json`) in the following format:

```
{
  "sentence": "All the money went into the interior decoration, none of it went into the chef
  "tokens": ["All", "the", "money", ...],
  "labels": ["O", "O", "O", ..., "B"],
  "aspect_terms": ["interior decoration", "chef"]
}
```

```
{
  "sentence": "But the staff was so horrible to us.",
  "tokens": [
    "But",
    "the",
    "staff",
    "was",
    "so",
    "horrible",
    "to",
    "us",
    "."
  ],
  "labels": [
    "O",
    "O",
    "B",
    "O",
    "O",
    "O",
    "O",
    "O",
    "O"
  ],
  "aspect_terms": [
    "staff"
  ]
},
{
```


Figure 1: An instance from `train_task_1.json`

1.2 Model Architectures & Hyperparameters

We experiment with two types of recurrent models ('RNN' and 'GRU') using two pre-trained embedding sets ('GloVe' and 'fastText'). All models use pre-trained embeddings directly (i.e., no custom vocabulary is built). The main architecture is as follows:

- **Input:** Pre-trained word embeddings (300-dimensional) obtained from either GloVe or fastText.
- **Recurrent Layers:** A two-layer RNN/GRU is used. In our final implementation, the recurrent layers are non-bidirectional. (Bidirectional variants could further improve aspect span detection.)
- **Regularization:** A dropout layer with a probability of 0.3 is applied to the output of the recurrent layer to prevent overfitting.
- **Fully Connected Layer:** A linear layer maps the hidden representations to the output space corresponding to the BIO tags (with a label mapping: {`<PAD>:0, O:1, B:2, I:3`}).
- **Hyperparameters:**
  - **Embedding Dimension:** 300 (fixed by the pre-trained embeddings).
  - **Hidden Dimension:** 512.
  - **Number of Layers:** 2.
  - **Dropout:** 0.3.
  - **Optimizer:** AdamW with a learning rate of 0.0005 and weight decay of 1×10^{-4}.
  - **Scheduler:** We use a `ReduceLROnPlateau` scheduler that monitors the validation F1 score (mode='max') with a patience of 2 epochs.

1.3 Loss Plots

The following figures show example loss curves for the models:

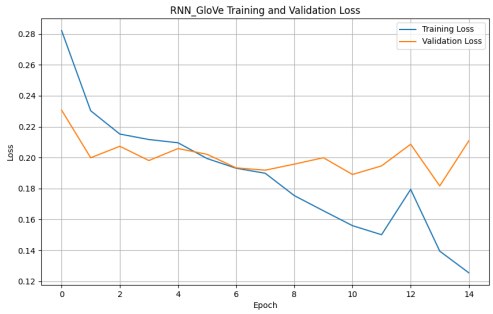


Figure 2: RNN with GloVe

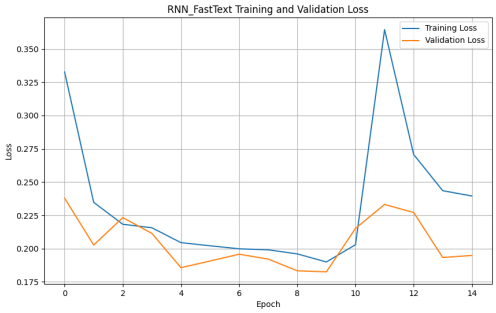


Figure 3: RNN with FastText

Figure 4: GRU with GloVe

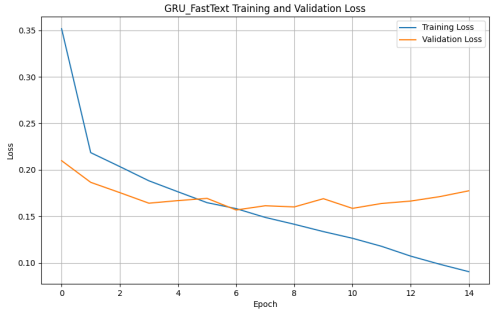


Figure 5: GRU with FastText

1.4 Performance Comparison

We trained four models:

- RNN with GloVe embeddings
- RNN with fastText embeddings
- GRU with GloVe embeddings
- GRU with fastText embeddings

Performance was measured at both the token level (tag-level F1) and at the span level (chunk-level F1) using the conlleval evaluation script. Our experimental results were as follows:

Model	Chunk-level F1	Tag-level F1
RNN-GloVe	60%	93.78%
RNN-fastText	60.56%	93.84%
GRU-GloVe	61.76%	94.52%
GRU-fastText	64.61%	94.58%

Table 1: Performance comparison on the validation set.

The high tag-level F1 indicates accurate token-level predictions; however, the chunk-level F1 is lower, reflecting challenges in correctly identifying entire aspect spans.

The models' performance is influenced by the quality of the embeddings and the type of recurrent unit:

- **Embeddings:** FastText embeddings outperform GloVe in our experiments, especially at the chunk level. This is likely because FastText learns representations at the subword level. It can capture morphological features and handle out-of-vocabulary or misspelled words better than GloVe, which treats each word as an atomic unit.
- **Recurrent Units:** The GRU-based models generally perform better than the RNN-based models. GRU cells include gating mechanisms (reset and update gates) that help mitigate the vanishing gradient problem, allowing them to capture long-range dependencies more effectively. This results in improved performance in identifying complete aspect spans.

Additionally, all models exhibit some degree of overfitting. Notably, the GRU model with GloVe embeddings overfits the most. This could be because:

- GloVe embeddings, while effective at capturing global co-occurrence statistics, do not incorporate subword information. As a result, the model may rely more on memorizing training examples.
- The GRU architecture, with its additional gating mechanisms and higher representational capacity, might be more prone to overfitting when combined with fixed embeddings.

There were also cases where `GRU_GloVe` performed slightly better than `GRU_FastText`. However, the difference was not huge and GRU with GloVe still experienced more overfitting than GRU with FastText.

1.5 Best Performing Model & Evaluation

The best-performing model, based on chunk-level F1, was **GRU-FastText** with a chunk F1 of 64.61% (and tag F1 of 94.58%).

1.6 Reasons for Performance

- **FastText Embeddings:** FastText embeddings improve performance because they capture subword-level information. This allows the model to better represent rare words and morphological variations, which is crucial for accurately identifying aspect term boundaries.
- **GRU Recurrent Unit:** GRU outperforms the standard RNN because its gating mechanisms help in effectively capturing long-term dependencies. This leads to more robust predictions of aspect spans, as the model can better utilize context from earlier and later parts of the sentence.
- **Overfitting:** The validation loss does not decrease much throughout as compared to the train loss. This could be improved upon through better feature engineering, bi-directional model and more complex architectures (like Conditional Random Field), embeddings and epochs.

2 Task 2

2.1 Preprocessing

2.1.1 tokenize_sentence(sentence)

The function `tokenize_sentence(sentence)` takes a sentence as input and breaks it down into individual words, known as tokens.

1. Remove all symbols: The function removes any character that is not a word character such as punctuation marks like commas, periods, exclamation points, etc.
2. Splitting on whitespace: After removing the symbols, the sentence is split into words at spaces and returns a list of individual words.

2.1.2 preprocess_data(data)

This function preprocesses a dataset containing sentences, their corresponding aspect terms, and sentiment polarity. It tokenizes sentences, identifies aspect terms, and determines their positions within the sentence. Then it outputs is a structured dataset as suggested in the assignment.

The function iterates over each entry in `data` and

1. Tokenizes sentence using `tokenize_sentence()`, which removes punctuation and splits the sentence into words.

2. Then it identifies Aspect Term Position in the sentence by iterating through the tokenized sentence and checking whether the aspect term matches a sequence of words in the sentence. If a match is found, the starting index of the aspect term in the tokenized sentence is recorded.

3. Then it constructes the Processed Entry of the following form and stores it -

    - `tokens`: The tokenized sentence.
    - `polarity`: The sentiment polarity of the aspect term.
    - `aspect_term`: The tokenized aspect term.
    - `index`: The position of the aspect term in the tokenized sentence.

2.1.3 ABSA Dataset (Aspect Based Sentiment Analysis Dataset)

The `ABSADataset` class extends the `Dataset` class from PyTorch and handles tokenization, encoding, and the preparation of necessary input features for the model.

1. Initializes -

    - `data`: A list of dictionary entries containing preprocessed text data.
    - `tokenizer`: A tokenizer (such as BERT or another transformer model tokenizer) to encode sentences.
    - `max_len`: The maximum sequence length for tokenized inputs (default: 128).

2. The method `__len__` returns the total number of samples in the dataset.

3. The `__getitem__` method retrieves and processes a sample at a given index:

    - Extracts sentence tokens, aspect terms, aspect indices, and sentiment polarity.
    - Ensures `aspect_terms` and `aspect_indices` are stored as lists.
    - Converts sentiment polarity from textual labels (`negative, neutral, positive, conflict`) to numerical values using a mapping:

        ```
        polarity_mapping = {'negative': 0, 'neutral': 1, 'positive': 2, 'conflict': 3}
        ```

    - Tokenizes the sentence using the provided tokenizer, adding special tokens and ensuring the sequence is padded or truncated to the specified `max_len`.
    - Creates an `aspect_mask`, a tensor marking the position of aspect terms in the sentence, accounting for the presence of special tokens - CLS at the beginning and SEP at the end.

```
    warnings.warn(
Sample 0: {'input_ids': tensor([ 101, 2021, 1996, 3095, 2001, 2061, 9202, 2000, 2149,  102,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0]), 'attention_mask': tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0]), 'token_type_ids': tensor([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0]), 'aspect_mask': tensor([0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0.]), 'polarity': tensor(0)}
```

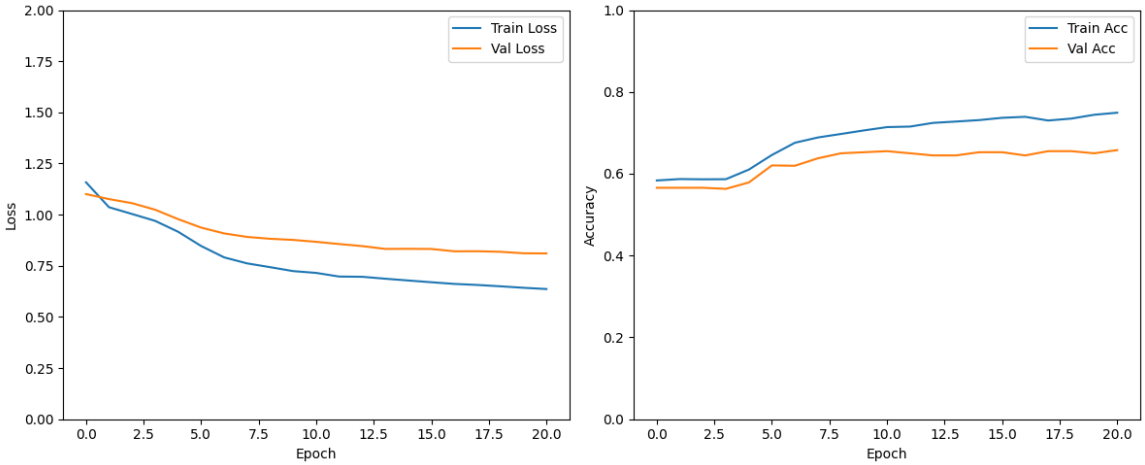

2.2 Architectures

2.2.1 Model 1 - Interactive Multi Head Attention Network

The first model that I tried used a combination of **BERT, Bidirectional GRU, Multi-Head Attention, and Aspect-Aware Position Embeddings**.

- BERT generated contextualized word embeddings, which are further refined using a **Bidirectional GRU (BiGRU)** to capture sequential dependencies.
- Aspect-aware **position embeddings** were added to enhance the model's focus on aspect terms within a sentence.
- The model employs **two multi-head attention mechanisms**: (1) **Aspect-to-Context Attention**, where the aspect term learns from the context, and (2) **Context-to-Aspect Attention**, where the sentence learns relevant information from the aspect term. Then these attention outputs were concatenated and passed through a **fusion layer**.
- The output layer then classified sentiment into different categories (e.g., positive, neutral, negative).

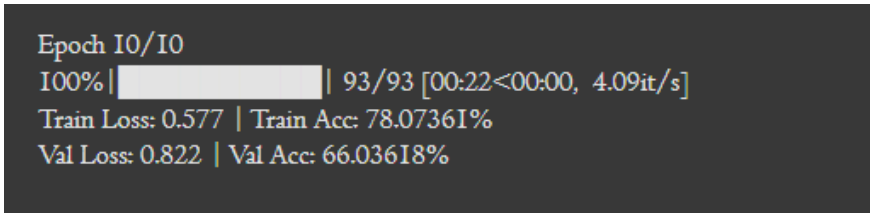
```
Epoch 21/21
100%|██████████| 93/93 [00:23<00:00,  3.98it/s]
Train Loss: 0.636 | Train Acc: 74.91501%
Val Loss: 0.810 | Val Acc: 65.77577%
Best model saved!
```

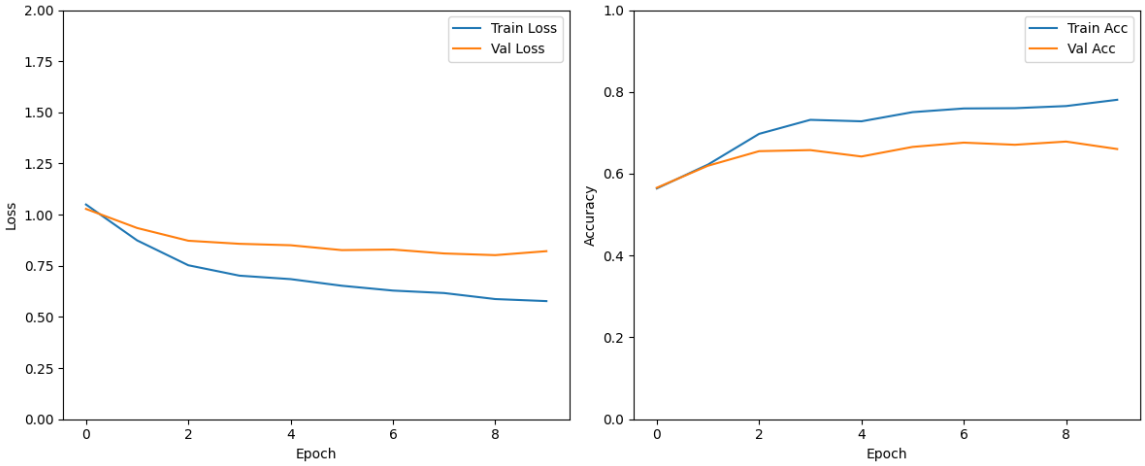


2.2.2 Model 2 - Graph-Based Sentiment Model

The second model I tried was **Graph-Based Sentiment Model**. It first extracts **BERT embeddings** for contextualized word representations. These embeddings are refined using **Graph Convolutional Networks (GCN)**, which capture syntactic and semantic relationships through dependency-based adjacency matrices.

Next, a **Bidirectional LSTM (BiLSTM)** captures sequential dependencies, and an **attention mechanism** highlights the most relevant tokens. The outputs from **GCN and BiLSTM** are then fused via a **linear transformation layer**, combining syntactic and sequential knowledge. Finally, a **fully connected output layer** performs sentiment classification, effectively leveraging both structural and contextual information.





2.2.3 Model 3 - Target Dependent LSTM and GCN

OVERVIEW The **TD_LSTM_GCN** model is designed for **aspect-based sentiment analysis (ABSA)** and incorporates three main components:

- **BERT Encoder**: Extracts contextual embeddings from input text.
- **Target-Dependent LSTM (TD-LSTM)**: Captures the left and right contexts separately along with the aspect term.
- **Graph Convolutional Network (GCN)**: Uses dependency information to capture syntactic relationships.
- **Attention Fusion Mechanism**: Integrates TD-LSTM and GCN representations to enhance sentiment classification.

MODEL COMPONENTS

1. BERT Encoder

- The model initializes a **BERT-base** model (`bert-base-uncased`) to generate deep contextualized word embeddings.
- The **last hidden states** of BERT are used as input embeddings for subsequent modules.

2. Target-Dependent LSTM (TD-LSTM) The TD-LSTM module consists of three LSTMs:

1. **Left LSTM** ($lstm_{left}$): Processes words **before the aspect term**.
2. **Right LSTM** ($lstm_{right}$): Processes words **after the aspect term**.
3. **Target LSTM** ($lstm_{target}$): Processes the **aspect term itself**.

Each LSTM is unidirectional, ensuring that the model processes left and right contexts separately to capture sentiment dependencies.

3. Graph Convolutional Layer (GCN Implementation)

- **Adjacency Matrix Processing:**Self-loops are added to the adjacency matrix and the matrix is normalized ($D^{-1/2}AD^{-1/2}$) for stability.
- **Feature Transformation:** The input is multiplied with a **learnable weight matrix**. The adjacency matrix is used to aggregate neighboring word features.

4. Graph Convolutional Network (GCN)

- The model uses **multiple GCN layers** ($num\_gcn\_layers = 2$) to capture syntactic dependencies between words.
- The input to GCN layers is the BERT embedding matrix.
- The adjacency matrix ($dependency\_matrix$) defines word relationships.
- Each GCN layer applies **graph convolution** followed by a ReLU activation.

5. Attention Mechanism Two attention scores are computed:

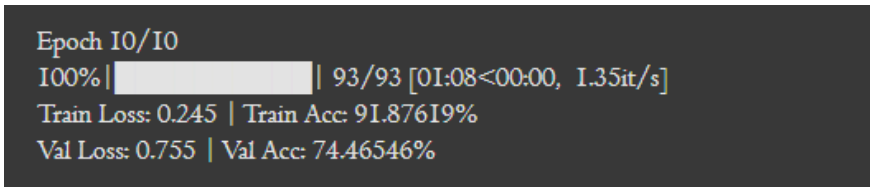
1. **LSTM-based attention**: Determines the importance of TD-LSTM features.
2. **GCN-based attention**: Determines the importance of GCN features.

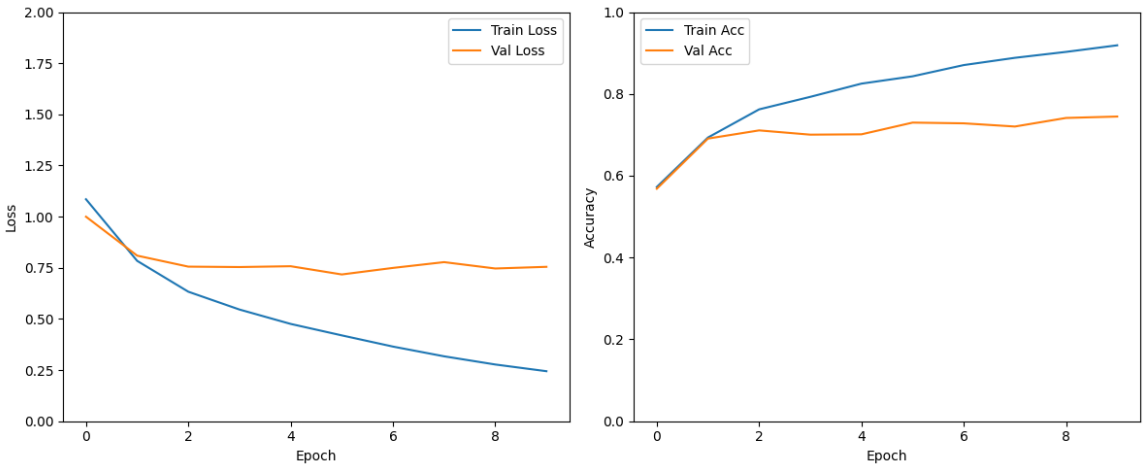
The final representation is a **weighted sum** of TD-LSTM and GCN outputs, ensuring optimal feature integration.

6. Fully Connected Layer (Classification Head) : The final representation is concatenated ($hidden\_dim \times 3$) and passed through a **fully connected layer** (fc). It produces sentiment classification output ($output\_dim = 4$).

FORWARD PASS

1. **Input Processing:***input_ids*, *attention_mask*, and *token_type_ids* are fed into **BERT** to obtain contextual embeddings.
2. **Dependency Matrix:** If no dependency matrix is provided, a simple heuristic-based approximation is created.
3. **Aspect Term Identification:**The model identifies the start and end positions of aspect terms using *aspect_mask*.
4. **TD-LSTM Processing:**The **left**, **right**, and **target** representations are extracted using LSTMs.
5. **Graph Convolution Processing:** The word embeddings pass through **GCN layers** to refine representations using syntactic dependencies. Aspect-specific representations are extracted from the GCN outputs.
6. **Attention-Based Fusion:** Computes **importance scores** for TD-LSTM and GCN representations. Produces a **final hidden representation** using a weighted sum.
7. **Classification:** The final representation is passed through a **fully connected layer** for sentiment prediction.





2.2.4 Hyperparameters Used

- Batch Size = 32
- Output Dimension = 4
- Maximum Sequence Lenght = 128

- Hidden Dimension 256
- Number of GCN Layers = 3
- Learning Rate = 4e-6
- Number of epochs = 10

Batch Size = 32 Output Dimension == 4 Maximum Sequence Lenght = 128 Hidden Dimension 256 Number of GCN Layers = 3 Learning Rate = 4e-6 Number of epochs = 10

2.2.5 Justification of the Best Model - TD LSTM and GCN

- **Aspect-Specific Context Modeling**: Separately captures left, right, and aspect-specific dependencies for improved ABSA.
- **Enhanced Structural & Contextual Understanding**: Combines sequential (TD-LSTM) and syntactic (GCN) modeling for richer sentiment representation.
- **Target-Specific Representations**: Prevents irrelevant aspect mixing by processing left and right contexts separately.
- **GCN for Dependency Parsing**: Incorporates syntactic relationships, making the model more robust to complex sentence structures.
- **Attention Fusion**: Selectively integrates crucial features from TD-LSTM and GCN for optimal classification.
- **Aspect-Specific Focus**: Explicitly processes aspect terms, ensuring sentiment analysis remains target-focused.

2.3 Fine Tuning BERT, BART, RoBERTa - Additional Task

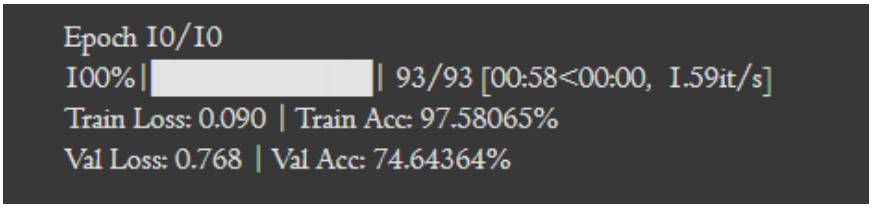
2.3.1 BERT for ABSA

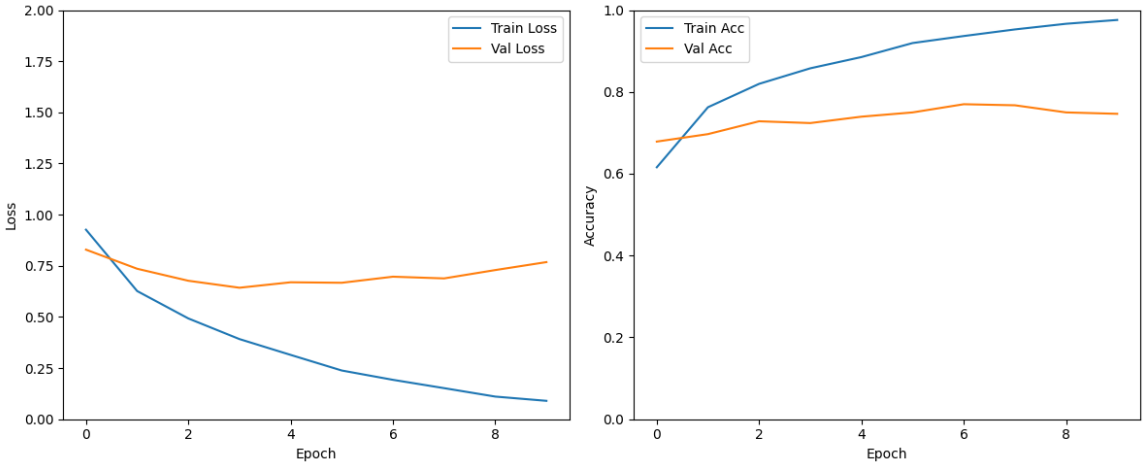
The `BERTForABSA` class fine-tunes BERT for ABSA by leveraging both the aspect representation and the sentence representation. The architecture consists of:

- A BERT model pre-trained on general language understanding tasks.
- A dropout layer with a probability of 0.1 to prevent overfitting.
- A linear classifier that takes a concatenation of the aspect representation and the sentence representation.

The forward pass includes:

1. Extracting contextualized token representations using BERT.
2. Computing the aspect representation as the weighted sum of token embeddings, using an aspect mask.
3. Extracting the sentence representation from the `[CLS]` token.
4. Concatenating both representations and passing them through a classifier.





2.3.2 BART for ABSA

The `BARTForABSA` class is similar to the BERT model but uses BART, a denoising autoencoder-based model. Key characteristics include:

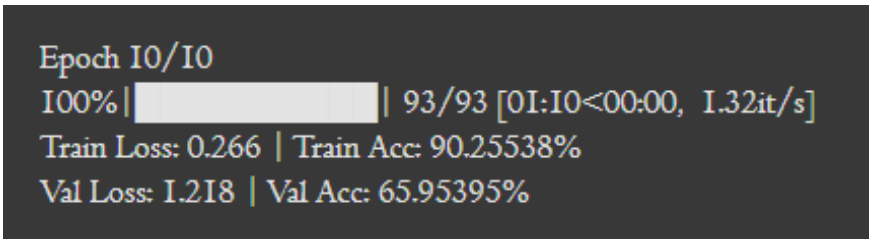
- BART model as the feature extractor.
- A dropout layer with a probability of 0.1.
- A classifier that combines the aspect representation and sentence representation.

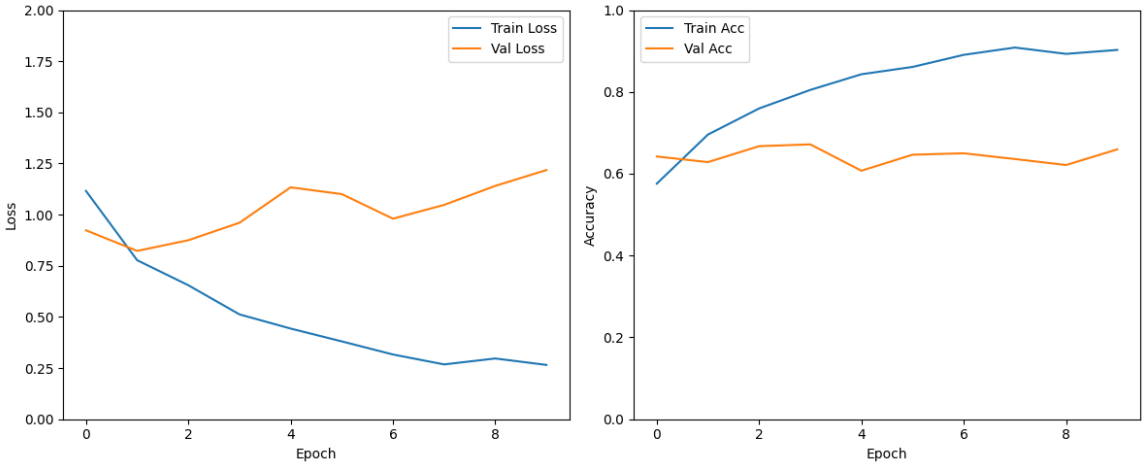
The architecture follows the same approach as BERT and RoBERTa, where:

1. The model extracts contextual embeddings.
2. Aspect and sentence representations are computed.
3. These representations are concatenated and passed through a classifier.

Differences from BERT:

- BART does not use token type embeddings.
- Uses the first token's embedding for the sentence representation.

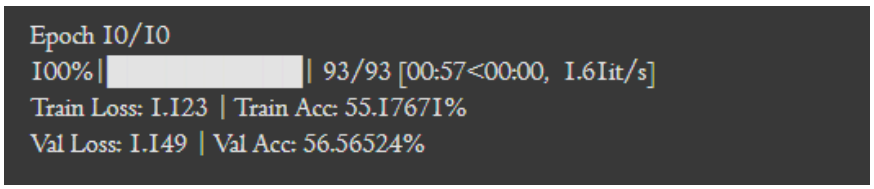


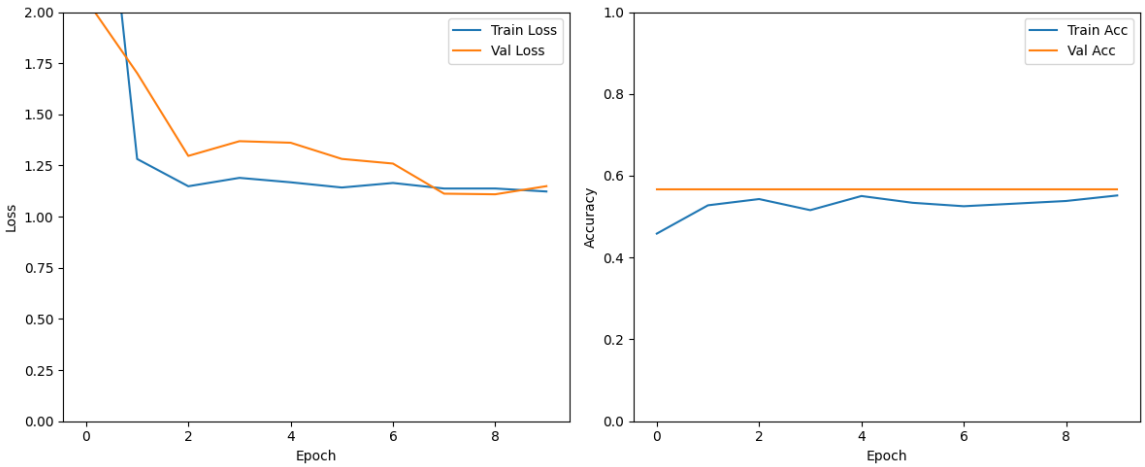


2.3.3 RoBERTa for ABSA

The `RoBERTaForABSA` class fine-tunes RoBERTa for ABSA. The architecture follows the same approach as BERT and BART, where:

1. The RoBERTa model extracts contextual embeddings.
2. Aspect and sentence representations are computed.
3. These representations are concatenated and passed through a classifier.





2.3.4 COMPREHENSIVE ANALYSIS

Why Does BERT Perform Best?

- **Bidirectional Context Encoding**: BERT's Masked Language Model (MLM) + Next Sentence Prediction (NSP) training helps capture aspect-sentiment relationships effectively.

- **Token Type Embeddings**: Helps differentiate aspect terms from the sentence, improving classification.
- **Strong Sentence Representation**: Uses the `[CLS]` token, leading to better sentiment predictions.

Why Does BART Underperform?

- Uses a **denoising autoencoder** for pretraining, which is less effective for fine-grained ABSA.
- **No token type embeddings**, making aspect-sentiment differentiation harder.
- **First token used for sentence representation**, which is weaker than BERT's `[CLS]` token.

Why Does RoBERTa Perform the Worst?

- **No token type embeddings** (like BART).
- **No NSP training**, reducing its ability to learn aspect-sentiment separation.
- **Aggressive pretraining on large data** helps general NLP tasks but not ABSA.

3 Task 3

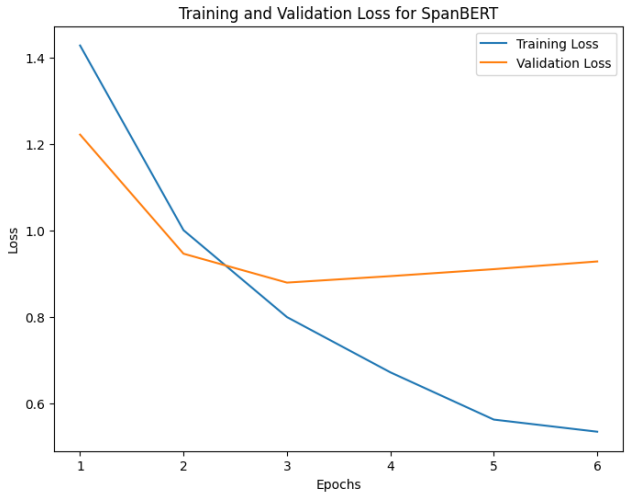
3.1 Dataset description and preprocessing

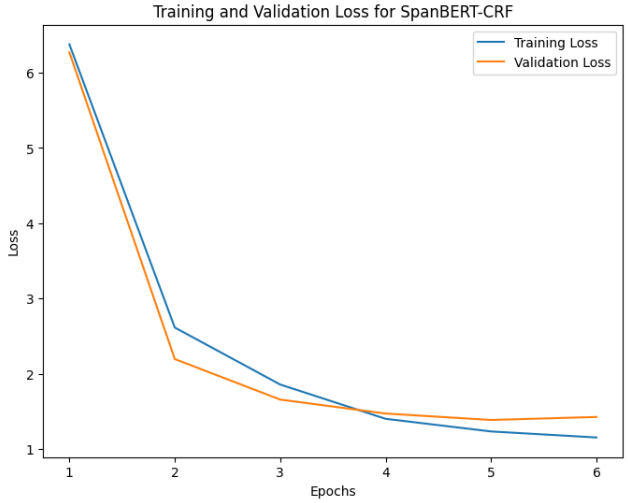
The SQuADv2 dataset is used for the Question Answering task and has 130k samples in the training set and 12k samples in the validation set. The columns are id, title, context, question and answer. It has over 50k unanswerable questions, and it expects the model to abstain from answering if the question can not be answered from the given context.
First, we convert the text input into tokens using the BERT tokenizer. We pass the question and context and get the tokenized output, where they are separated by [SEP] token and the it begins with [CLS] token. Since SpanBERT is an encoder-only model, it cannot give the answer in text format directly, so we predict the span of the answer and for that we need to find the answer in terms of token indices for fine-tuning and prediction tasks. Therefore, we store the start position and end positions in the preprocessing.

3.2 Justification of model choices and hyperparameters

I am using a subset of 60k samples for fine-tuning the models.
SpanBERT: There were no changes with the architecture of SpanBERT. I tried 4 different configurations and the model was usually overfitting after the 2nd epoch. The best EM score was achieved with this configurations, so the hyperparameters are chosen empirically.
SpanBERT-CRF: I have added two CRF layers, one for each start position and end position. I have also added extra hidden layers between the CRF layers and the final hidden layer of SpanBERT, so it can learn better representations. For this model as well, the hyperparameters were chosen empirically when the best EM score was achieved.

3.3 Training and validation plots





3.4 Comparative analysis of SpanBERT-CRF and SpanBERT

The SpanBERT model tends to overfit after 3rd epoch whereas the SpanBERT-CRF model slightly overfits after 5th epoch. SpanBERT model has lower values for loss on both training and validation sets. Even though SpanBERT overfits earlier, it has a better exact match score on the validation set than SpanBERT-CRF.

3.5 Exact-match scores on the validation set

```
70.27951606174385
SpanBERT Exact Match Score: 70.28 %
```

```
66.03254067584481
SpanBERT-CRF Exact Match Score: 66.03 %
```

4 Individual Contribution

- Himanshu Raj: Task 3 implementation and report
- Ishita: Task 2 implementation and report
- Ritika Thakur: Task 1 implementation and report

5 References

- https://github.com/sighsmile/conlleval
- https://nlp.stanford.edu/projects/glove/
- https://fasttext.cc/docs/en/english-vectors.html
- https://github.com/TarikBugraAy/BERT_Fine_Tune_For_ABSA
- https://huggingface.co/learn/nlp-course/en/chapter7/7
- https://github.com/huggingface/transformers/blob/main/src/transformers/models/bert/modeling$_bert.py$