QUORA QUESTION PAIR SIMILARITY

Natural Language Processing (DATS 6312)

<u>Final Project Individual Contribution Report</u>

Name: Richik Ghosh

Group: 5

Date: 12/11/2024

TABLE OF CONTENTS

1. **Introduction**
2. **Dataset & Outline of Shared Work**

3. **Description of Individual Work**

    3.1 Background and overview

    3.2 Siamese Transformer on SBERT with Dynamic Masking

        3.2.1 Data Preparation and Loading

        3.2.2 Model Architecture and Pipeline

        3.2.3 Results

    3.3 BERT & SBERT Embedding Alignment using Contrastive Loss

        3.3.1 SBERT Embedding Alignment using Contrastive Loss

            3.3.1.1 Data Preparation and Loading

            3.3.1.2 Model Architecture

        3.3.2 BERT Embedding Alignment using Contrastive Loss

        3.3.3 Results

    3.4 Cross-encoder BERT

        3.4.1 Data loading and preparation

        3.4.2 Model architecture

        3.4.3 Results

4. **Results**

5. **Summary and Conclusion**

6. **Code Source Analysis**

7. **References**

Introduction

Community-driven Q&A platforms like Reddit and Quora have experienced remarkable expansion in recent years, with Quora's monthly user base exceeding 100 million visitors. This growth has brought a significant challenge: an increasing number of semantically equivalent questions being posted across these platforms. This redundancy creates two main problems: users must spend more time searching through duplicate questions to find relevant answers, and content contributors face increased workload addressing multiple versions of essentially the same question. To address this challenge, our project utilized the Quora Question Pairs dataset to develop an advanced question similarity detection system using sophisticated deep learning approaches.

Through the implementation of cutting-edge architectures, we successfully enhanced our model's capability to identify and understand the underlying semantic connections between question pairs. The models demonstrated strong ability to comprehend both context and user intent in question formulation. Our extensive process of experimentation and hyperparameter optimization yielded impressive results, achieving a test F1 macro score of 93.28%. This performance validates the effectiveness of transformer-based Siamese networks in tackling complex natural language processing challenges, particularly in the domain of question pair similarity detection.

Dataset

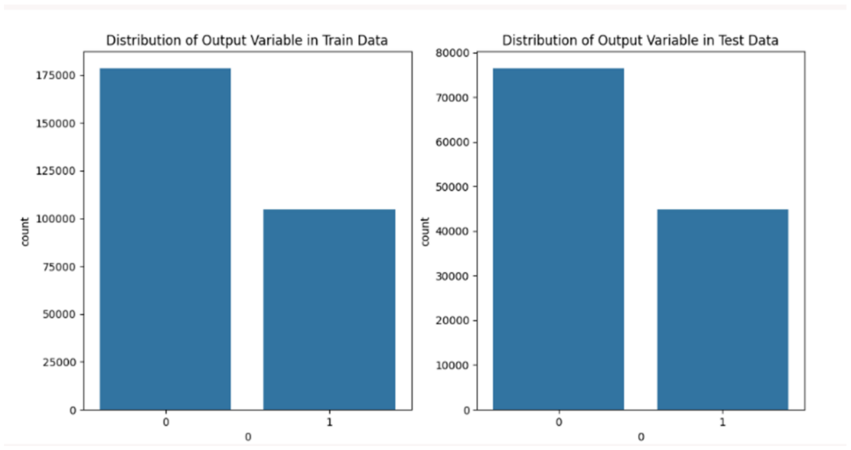
The dataset is sourced from Kaggle [1] and was introduced as part of a competition on the platform for the same challenge as our project.

The Quora Question Pairs dataset is composed of 404,290 question pairs, and has 6 total features: *id, qid1, qid2, question1, question2, is_duplicate*:

id	qid1	qid2	question1	question2	is_duplicate
38 38	61 61	62	What's one thing you would like to do better?	What's one thing you do despite knowing better?	0
180	361	362	How do you get deleted Instagram chats?	How can I view deleted Instagram dms?	1

Table 1: A snapshot of the Quora Question Pairs Dataset

The dataset is imbalanced – approximately 36.92% are duplicate questions while 63.08% are non-duplicate pairs:



A bar plot showcasing the imbalance in the data

Shared Work Outline:

1. Exploratory Data Analysis (EDA)

EDA: Conducted exploratory data analysis to understand the distribution of data, identify any patterns, and detect anomalies. This step included visualizations and statistical summaries.

2. Data Preprocessing

1. Data Normalization: Converting text to lowercase, expanding contractions, and standardizing symbols (e.g., currency, percentages).
2. Data Cleaning: Removing non-word characters and HTML tags, and processing large numbers into a readable format.
3. Stemming: Applied the PorterStemmer to reduce words to their root forms.

3. Feature Extraction

1. Tokenization: Tokenizing the questions using a word tokenizer.
2. Feature Categories: Extracted 37 features that fall into three main categories:
3. Fuzzy Features: Including fuzz_ratio, fuzz_partial_ratio, etc., to measure word-to-word fuzzy similarity.
4. Token Features: Analysis of stopwords and non-stopwords (e.g., common non-stopwords, common stopwords, word size difference).
5. Common Subsequence Features: Measuring similarity between parts of sentences (e.g., largest_common_subsequence, jaccard_similarity).

4. Vectorization

1. Weighted TF-IDF: Implemented weighted TF-IDF scores, leveraging the spaCy model for word embeddings. Combined the generated TF-IDF scores with word embeddings to enhance the feature representation.
2. BERT Embeddings: Created BERT embeddings

5. Model Training and Comparison

1. Classical Models with TF-IDF and BERT embeddings: Trained classical machine learning models (e.g. Logistic Regression, Random Forest, Naïve Bayes) using the weighted TF-IDF and the BERT embedding features.

2. DL Models: Trained and evaluated DL models: LSTM and GRU
3. Transformer Models: Applied different variations of the BERT architecture

6. Streamlit app

7. Final Model and Evaluation

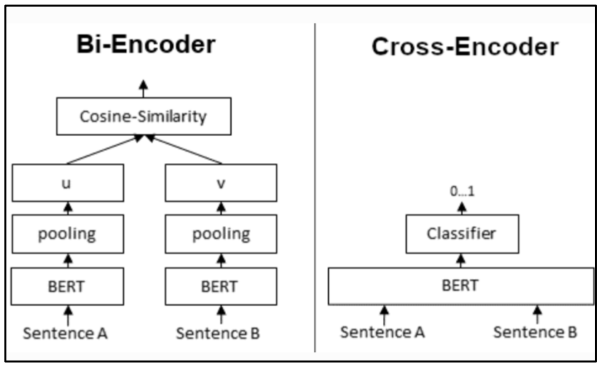
1. Compared and evaluated all the applied models

Evaluation Metrics: Used the F1 macro score for model evaluation.

Description of Individual Work

3.1 Background and overview

In my part of the project, I have essentially worked with two architectures – bi-encoder and cross-encoder:



Encoder types and their architecture

The bi-encoders are the first three (i.e. the Siamese BERT architecture) where both the embeddings are updated together in parallel, while the other is the cross-encoder, used at the end which ultimately gave the best results.

The bi-encoder approach leveraged advancements in transformer-based models, specifically Sentence-BERT (SBERT), to generate contextualized embeddings that capture the nuanced semantic relationships between paired inputs.

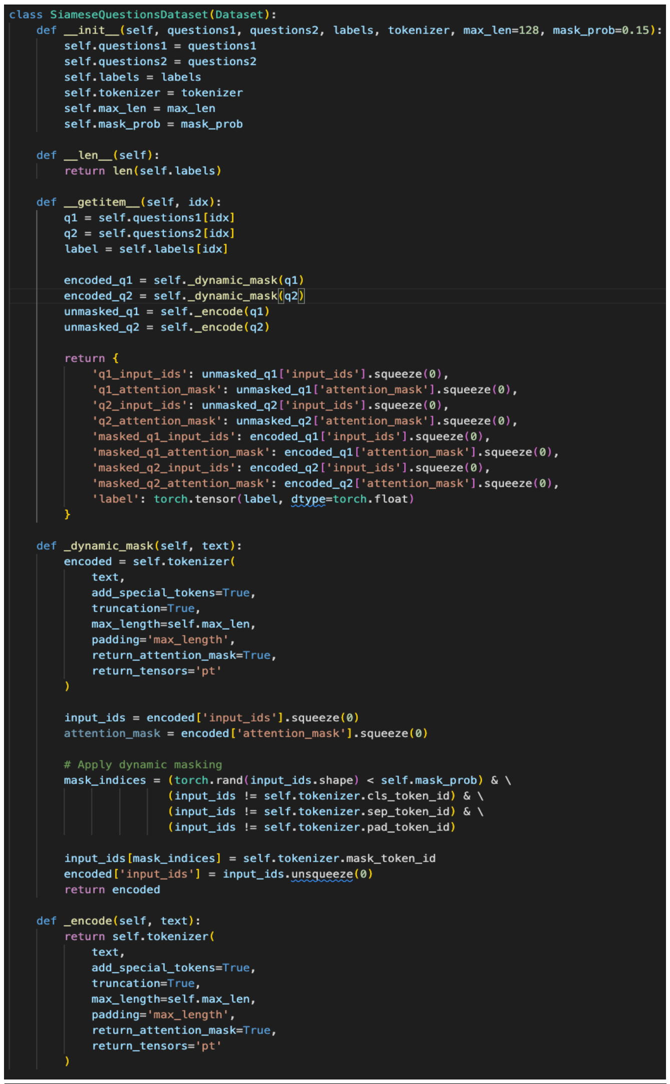
The system incorporated a unique processing mechanism, combining individual embeddings of paired questions with their absolute difference to represent both shared and contrasting features. To enhance the representation learning process, multiple pooling strategies were implemented and evaluated, enabling an empirical assessment of their impact on model performance. Additionally, dynamic masking inspired by self-supervised learning methods was introduced as an auxiliary task. This involved masking random tokens during training, which improved the model's ability to handle noise and variations in the input data. I also employed contrastive loss to efficiently align the embeddings according to their true label values, in addition to implementing cross-encoder technique to pass the question pairs simultaneously to the BERT model to generate the class labels.

This design choice capitalized on SBERT's ability to efficiently encode sentence-level semantics while minimizing task-specific data requirements. The architecture was designed to balance computational efficiency and performance, making it scalable and generalizable. The combination of innovative strategies like fine-tuning, pooling experimentation, and auxiliary learning enabled the model to achieve robust performance and adapt effectively to real-world semantic similarity challenges.

3.2 Siamese Transformer on SBERT with Dynamic Masking

3.2.1 Data Preparation and Loading

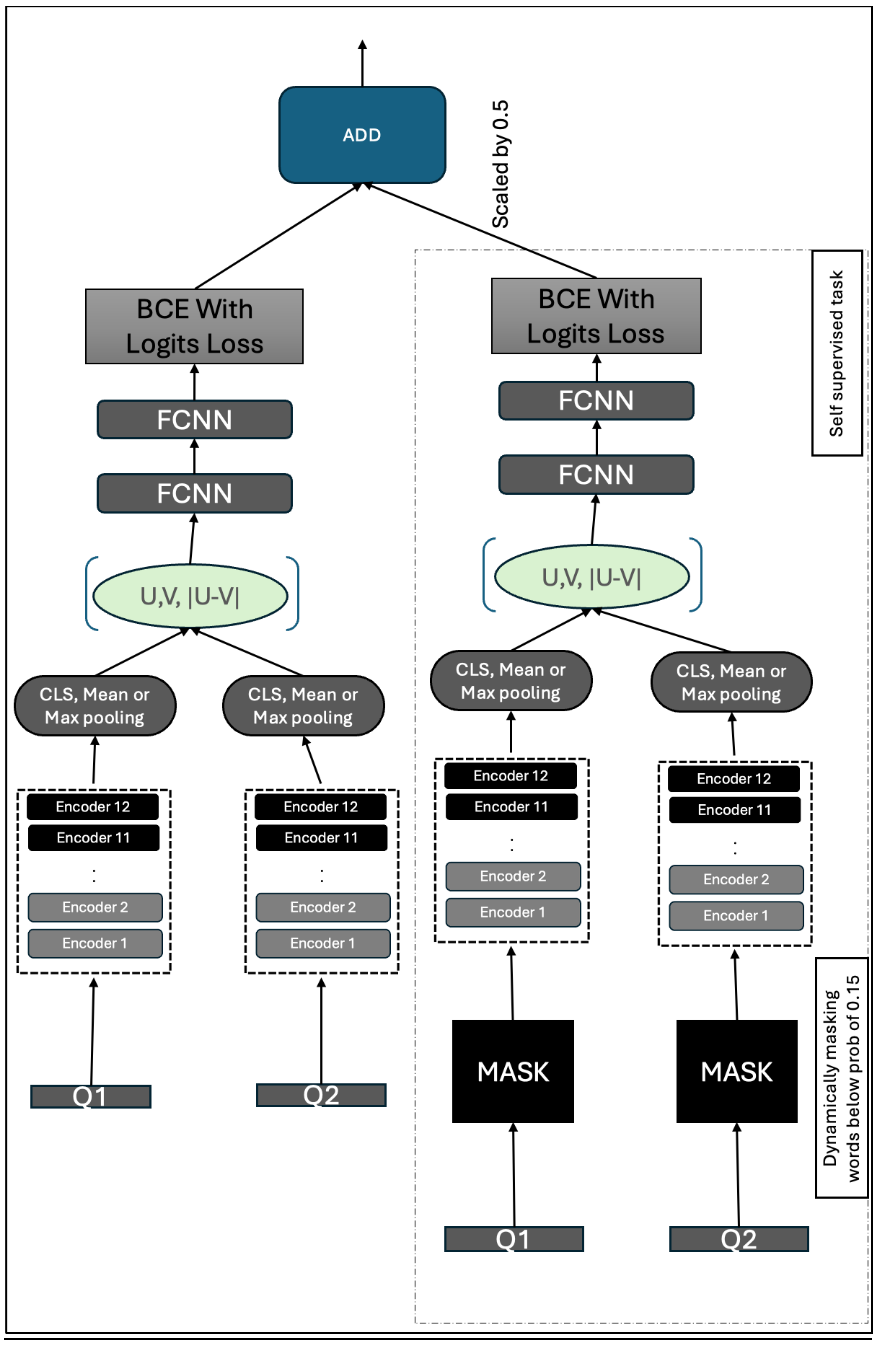
I implemented a robust data preparation pipeline to handle question pairs for training the Siamese BERT [2] model. The dataset was split into training (90%), validation (5%), and test (5%) sets using stratified sampling to maintain label distribution. I applied dynamic masking with a 15% probability during tokenization to enhance model robustness, ensuring [CLS], [SEP], and padding tokens remained unaffected. Each question was tokenized using the sentence-transformers/paraphrase-MiniLM-L6-v2 tokenizer, with truncation or padding applied to a maximum length of 128 tokens. Both unmasked and masked representations of the questions were generated for the main and auxiliary tasks, respectively. Using PyTorch's DataLoader, I loaded the data in batches of size 32, ensuring efficient training with shuffled training data and fixed validation ordering.



```python
class SiameseQuestionsDataset(Dataset):
    def __init__(self, questions1, questions2, labels, tokenizer, max_len=128, mask_prob=0.15):
        self.questions1 = questions1
        self.questions2 = questions2
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_len = max_len
        self.mask_prob = mask_prob

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, idx):
        q1 = self.questions1[idx]
        q2 = self.questions2[idx]
        label = self.labels[idx]

        encoded_q1 = self._dynamic_mask(q1)
        encoded_q2 = self._dynamic_mask(q2)
        unmasked_q1 = self._encode(q1)
        unmasked_q2 = self._encode(q2)

        return {
            'q1_input_ids': unmasked_q1['input_ids'].squeeze(0),
            'q1_attention_mask': unmasked_q1['attention_mask'].squeeze(0),
            'q2_input_ids': unmasked_q2['input_ids'].squeeze(0),
            'q2_attention_mask': unmasked_q2['attention_mask'].squeeze(0),
            'masked_q1_input_ids': encoded_q1['input_ids'].squeeze(0),
            'masked_q1_attention_mask': encoded_q1['attention_mask'].squeeze(0),
            'masked_q2_input_ids': encoded_q2['input_ids'].squeeze(0),
            'masked_q2_attention_mask': encoded_q2['attention_mask'].squeeze(0),
            'label': torch.tensor(label, dtype=torch.float)
        }

    def _dynamic_mask(self, text):
        encoded = self.tokenizer(
            text,
            add_special_tokens=True,
            truncation=True,
            max_length=self.max_len,
            padding='max_length',
            return_attention_mask=True,
            return_tensors='pt'
        )

        input_ids = encoded['input_ids'].squeeze(0)
        attention_mask = encoded['attention_mask'].squeeze(0)

        # Apply dynamic masking
        mask_indices = (torch.rand(input_ids.shape) < self.mask_prob) & \
                       (input_ids != self.tokenizer.cls_token_id) & \
                       (input_ids != self.tokenizer.sep_token_id) & \
                       (input_ids != self.tokenizer.pad_token_id)

        input_ids[mask_indices] = self.tokenizer.mask_token_id
        encoded['input_ids'] = input_ids.unsqueeze(0)
        return encoded

    def _encode(self, text):
        return self.tokenizer(
            text,
            add_special_tokens=True,
            truncation=True,
            max_length=self.max_len,
            padding='max_length',
            return_attention_mask=True,
            return_tensors='pt'
        )
```

Data Loading Pipeline

3.2.2 Model Architecture and Pipeline



Model Architecture

The architecture relied on a shared pretrained BERT encoder for both questions. To optimize computational efficiency, I froze all but the last two encoder layers for fine-tuning. The pooled embeddings from the encoder were generated using one of three strategies: CLS pooling (using the [CLS] token embedding), mean pooling (averaging embeddings weighted by attention masks), and max pooling (extracting the maximum value for each embedding dimension).

The embeddings of Q1 and Q2 were combined into a single vector using three components: the embeddings of Q1 (U), the embeddings of Q2 (V), and the element-wise absolute difference (|U - V|). This combined representation was passed through a feed-forward neural network (FCNN) for classification. As the group member previously tested in their implementation that taking the absolute difference of the embeddings, helped produced better predictions, I followed the same strategy. The main task of the FCNN was to predict whether the two questions were duplicates, with Binary Cross-Entropy with Logits Loss (BCEWithLogitsLoss) used to compute the loss. This functionality was implemented in the forward method of the SBERTSiamese class.

To improve the robustness of the embeddings, I introduced an auxiliary self-supervised task that used dynamically masked question pairs. Masked embeddings were processed similarly to the main task, and the auxiliary loss was also computed using BCEWithLogitsLoss. The forward_with_masked method handled this parallel computation. The final loss combined the main and auxiliary losses using a weighted sum:

$$\text{Total Loss} = \text{Min. Loss} + 0.5 * \text{Auxiliary Loss}$$

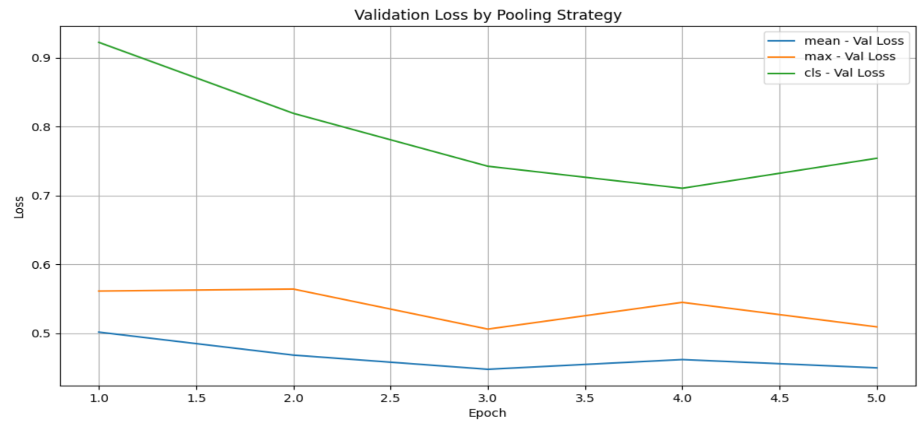
This combination, implemented in the train function, ensured that the auxiliary task complemented the main task without dominating it by masking, as I am scaling it by a factor of 0.5.

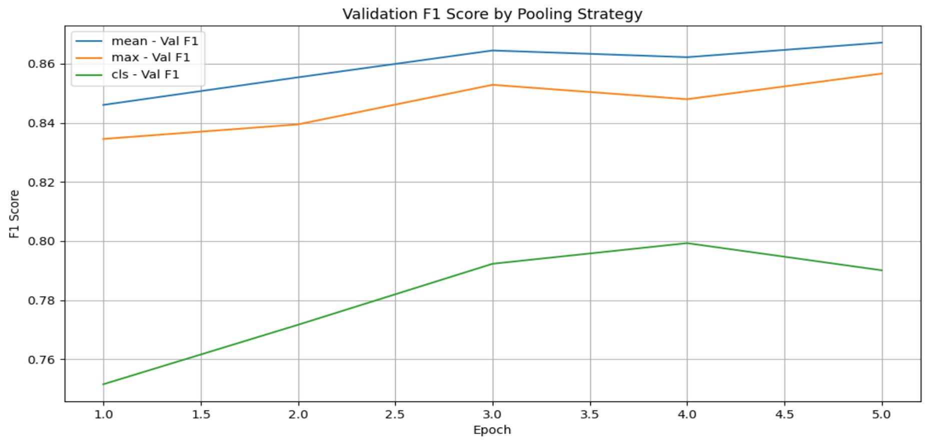
During training, the combined loss was used to optimize the model. Both unmasked and masked inputs were used for the main training and auxiliary training respectively, while only the unmasked inputs were used for validation and testing. I experimented with both high and low learning rates, but finally stuck to 3e-5 as it gave the best convergence trend. Checkpoints were saved based on the validation loss for each pooling strategy, ensuring the best-performing models were retained. After training, I evaluated the model on the test set and visualized trends in validation loss, F1 score, and accuracy for each pooling strategy.

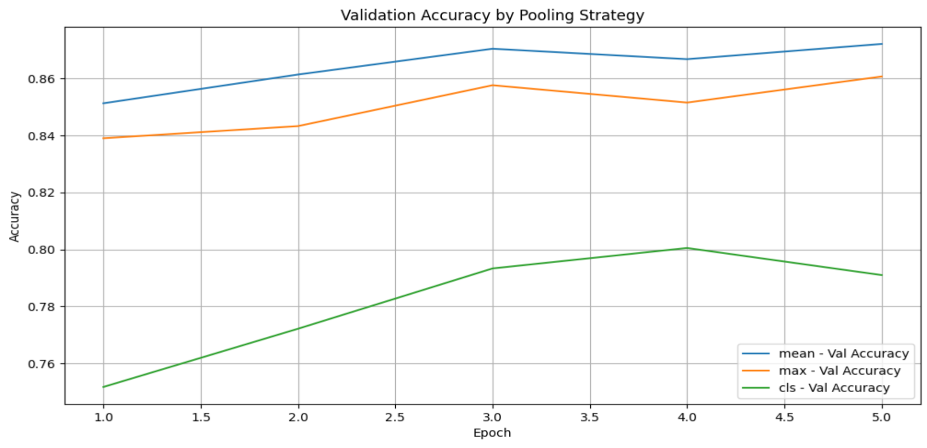
This architecture leverages shared encoding, feature combination, and an auxiliary self-supervised task to enhance semantic understanding and model robustness. By combining main and auxiliary losses, the model achieved better generalization and improved its ability to capture meaningful relationships between question pairs.

3.2.3 Results

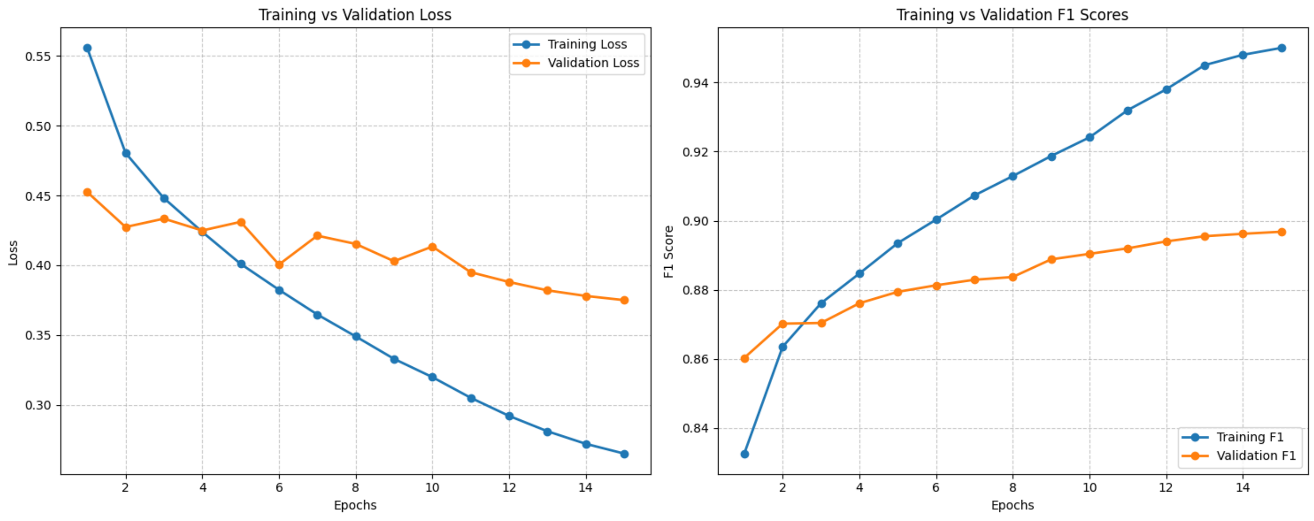
I trained the model for 5 epochs first for all pooling strategies. Here are the results







Since the mean pooling technique produced the best results, I stuck with that and finally trained my model for 15 epochs.



It is evident from the above plots that the model started overfitting around the 8th epoch. The best validation F1 macro I obtained is 89.68%.

3.3 BERT & SBERT Embedding Alignment using Contrastive Loss

In this segment I have tried using both BERT (bert-base-uncased; the uncased version was chosen since the text had already been lowered in the preprocessing phase) and

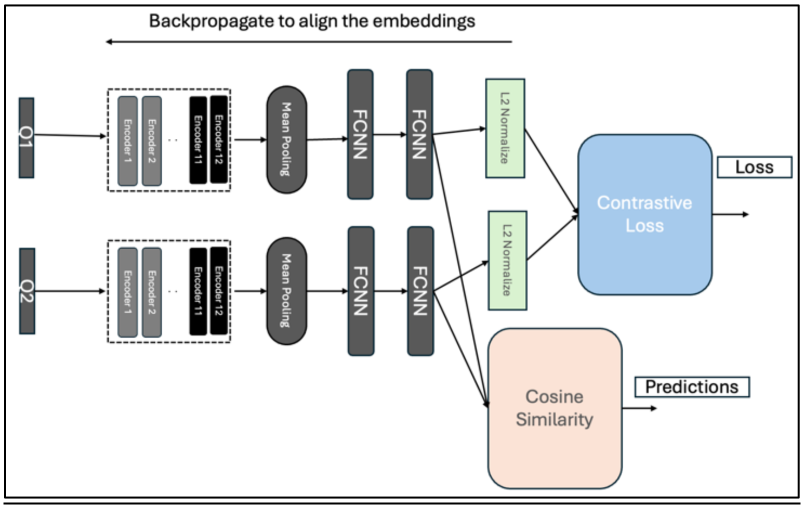
SBERT (sentence-transformers/paraphrase-MiniLM-L6-v2; consisting of 6 encoders).

3.3.1 SBERT Embedding Alignment using Contrastive Loss

3.3.1.1 Data Preparation and Loading

I used the encode_plus method from Hugging Face's AutoTokenizer for tokenization, ensuring each question was tokenized, padded to 128 tokens, and enriched with attention masks and special tokens like [CLS] and [SEP]. I later tested the direct tokenizer() call but found no difference in results. The tokenized outputs were integrated into a custom PyTorch dataset class (SiameseQuestionsDataset), producing question pairs and binary labels as PyTorch tensors for efficient batching and GPU processing during training.

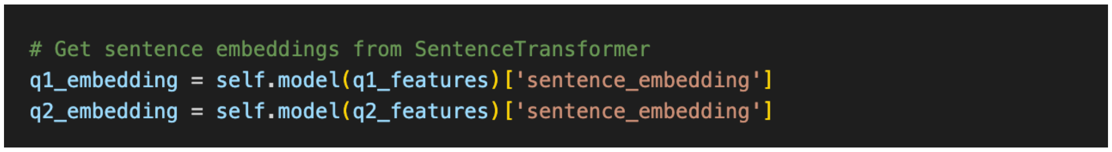
3.3.1.2 Model Architecture



Model Architecture

The Siamese network architecture leveraged Sentence-BERT (SBERT) to effectively handle the task of question similarity detection. The architecture consisted of two identical branches, each processing one question from a pair independently. Both branches utilized a pre-trained SBERT encoder, designed to generate semantically rich embeddings. This pre-trained model provided a robust starting point, as it was already optimized for semantic similarity tasks.

The mean embeddings for the SBERT captured here by this line of code. These embeddings are used in our alignment.



```
# Get sentence embeddings from SentenceTransformer
q1_embedding = self.model(q1_features)['sentence_embedding']
q2_embedding = self.model(q2_features)['sentence_embedding']
```

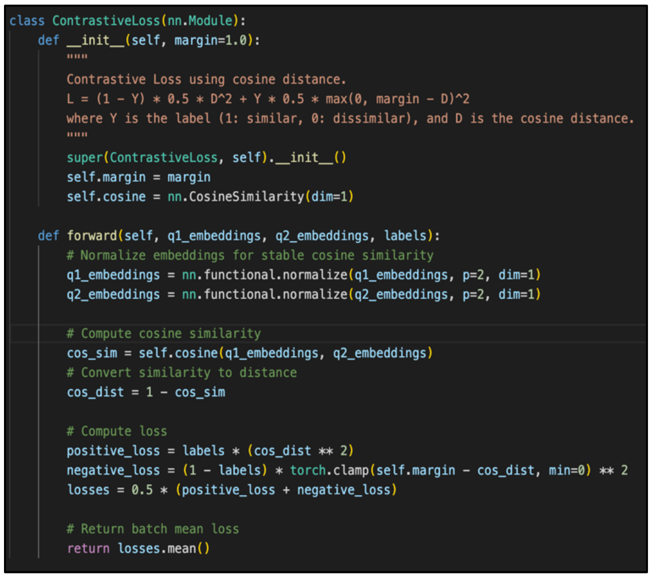
Mean pooling in SBERT

To tailor SBERT to the specific requirements of this task, I selectively unfroze the last two layers of the transformer encoder. This allowed the model to refine its higher-level representations during fine-tuning while preserving the general semantic understanding encoded in the frozen layers. This selective unfreezing approach struck a balance between avoiding overfitting and enabling task-specific learning. Each SBERT encoder processed its respective question to produce contextual embeddings, which were then passed through fully connected neural networks (FCNNs). These FCNNs reduced the dimensionality of the embeddings, incorporated non-linear transformations through ReLU activation, and applied dropout for regularization to prevent overfitting. The final representations were compared using cosine similarity, which quantified the semantic relationship between the questions. A decision threshold of 0.5 was then applied to classify the question pairs as duplicates or non-duplicates.

I used **contrastive loss [3]** to train and align the embeddings of the Siamese network. The goal of contrastive loss is to optimize the model's embeddings such that similar question pairs are brought closer together in the embedding space, while dissimilar pairs are pushed apart by at least a specified margin. This loss function is particularly suitable for tasks like question similarity, as it enforces alignment of semantically related embeddings while ensuring separation for unrelated ones.

$$\boxed{L = Y \cdot 0.5 \cdot D^2 + (1 - Y) \cdot 0.5 \cdot \max(0, \text{margin} - D)^2}$$

Contrastive loss equation



```python
class ContrastiveLoss(nn.Module):
    def __init__(self, margin=1.0):
        """
        Contrastive Loss using cosine distance.
        L = (1 - Y) * 0.5 * D^2 + Y * 0.5 * max(0, margin - D)^2
        where Y is the label (1: similar, 0: dissimilar), and D is the cosine distance.
        """
        super(ContrastiveLoss, self).__init__()
        self.margin = margin
        self.cosine = nn.CosineSimilarity(dim=1)

    def forward(self, q1_embeddings, q2_embeddings, labels):
        # Normalize embeddings for stable cosine similarity
        q1_embeddings = nn.functional.normalize(q1_embeddings, p=2, dim=1)
        q2_embeddings = nn.functional.normalize(q2_embeddings, p=2, dim=1)

        # Compute cosine similarity
        cos_sim = self.cosine(q1_embeddings, q2_embeddings)
        # Convert similarity to distance
        cos_dist = 1 - cos_sim

        # Compute loss
        positive_loss = labels * (cos_dist ** 2)
        negative_loss = (1 - labels) * torch.clamp(self.margin - cos_dist, min=0) ** 2
        losses = 0.5 * (positive_loss + negative_loss)

        # Return batch mean loss
        return losses.mean()
```

Code representation of contrastive loss

After normalizing the embeddings using L2 normalization, I applied the contrastive loss function, where Y is the binary label (Y = 1 for similar pairs, Y= 0 for dissimilar pairs). The cosine distance D=(1−cosine similarity) measures the angular difference between vectors, with smaller D values indicating higher semantic similarity. This approach is particularly suited for high-dimensional data, as it focuses on the direction of vectors rather than their magnitude. Unlike geometric measures such as Euclidean distance, which can become unreliable in high-dimensional spaces due to the curse of dimensionality (where data points appear equidistant), cosine distance provides a more robust measure of similarity by capturing the relative orientation of embeddings. This ensures that semantic relationships between vectors are effectively preserved. The hyperparameter **margin** (set to 0.8 after experimenting with numerous values) specifies the minimum required separation for dissimilar pairs, ensuring embeddings are aligned for similar pairs and sufficiently apart for dissimilar ones, promoting robust representation learning.

The contrastive loss function has two components:

1. **Positive Loss** (Y=1): Minimizes the squared cosine distance (D^2) for similar pairs.

2. **Negative Loss** (Y=0): When the label Y=0 (indicating dissimilar pairs), the loss function ensures that the embeddings of the two inputs are separated by at least a defined margin. If the cosine distance D between the embeddings is smaller than the margin, the term (margin - D)^2 penalizes the model. This encourages the embeddings to move further apart in the vector space. Conversely, if D>=margin, no penalty is applied, as the embeddings are already sufficiently separated. This mechanism ensures that dissimilar pairs maintain a meaningful separation, improving the model's ability to distinguish between unrelated inputs.

The training process employed the AdamW optimizer with a learning rate of 3e-5, which was selected for its ability to handle sparse gradients and provide stable convergence. A batch size of 32 was chosen to balance computational efficiency and memory constraints. I monitored metrics like validation loss and F1 scores. Checkpointing was implemented to save the model with the best validation loss, ensuring robustness during evaluation and enabling recovery if needed.

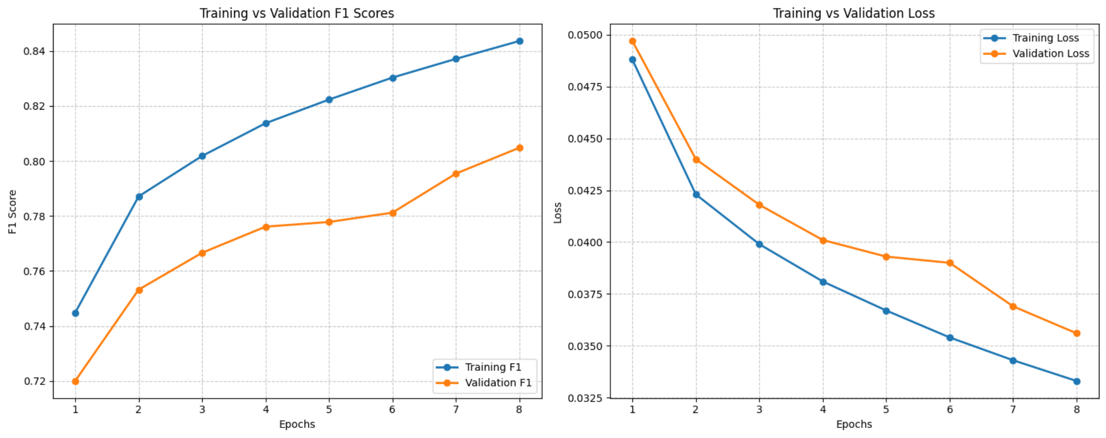
The implemented Siamese SBERT model effectively captured semantic relationships between question pairs through selective fine-tuning, FCNNs for dimensionality reduction, and contrastive loss for optimizing embeddings. With careful tuning of parameters and robust training practices, the architecture achieved reliable performance, demonstrating its suitability for question similarity detection tasks.

3.3.2 BERT Embedding Alignment using Contrastive Loss

In this implementation, I used a BERT-based Siamese architecture (*bert-base-uncased*) to align question embeddings with contrastive loss. Similar to SBERT, two identical BERT encoders processed question pairs independently to generate contextualized embeddings. While the results were comparable to SBERT, the BERT implementation required significantly more training time due to its larger size (12 encoder layers versus SBERT's 6 layers in the MiniLM variant). This increased computational cost arose from longer forward passes and backpropagation times. Although BERT demonstrated its capability to handle the task effectively, SBERT proved more efficient, offering similar performance with reduced resource requirements. This highlighted the trade-off between computational efficiency and model complexity when selecting architectures for embedding alignment tasks.

3.3.3 Results

I tried running this alignment architecture using both BERT and SBERT. The results I got are similar indicating that smaller model like SBERT can perform similarly like this. So, choosing SBERT over BERT for this architecture is the best as it removes the computational overhead.



Results from SBERT

The best val F1 macro from SBERT is around 80.48% and for the BERT model its 79.48%. The training f1 macros were almost the same of around 84%. But looking at the validation trend, SBERT performed better.

3.4. Cross-Encoder BERT

3.4.1. Data Loading and Preparation

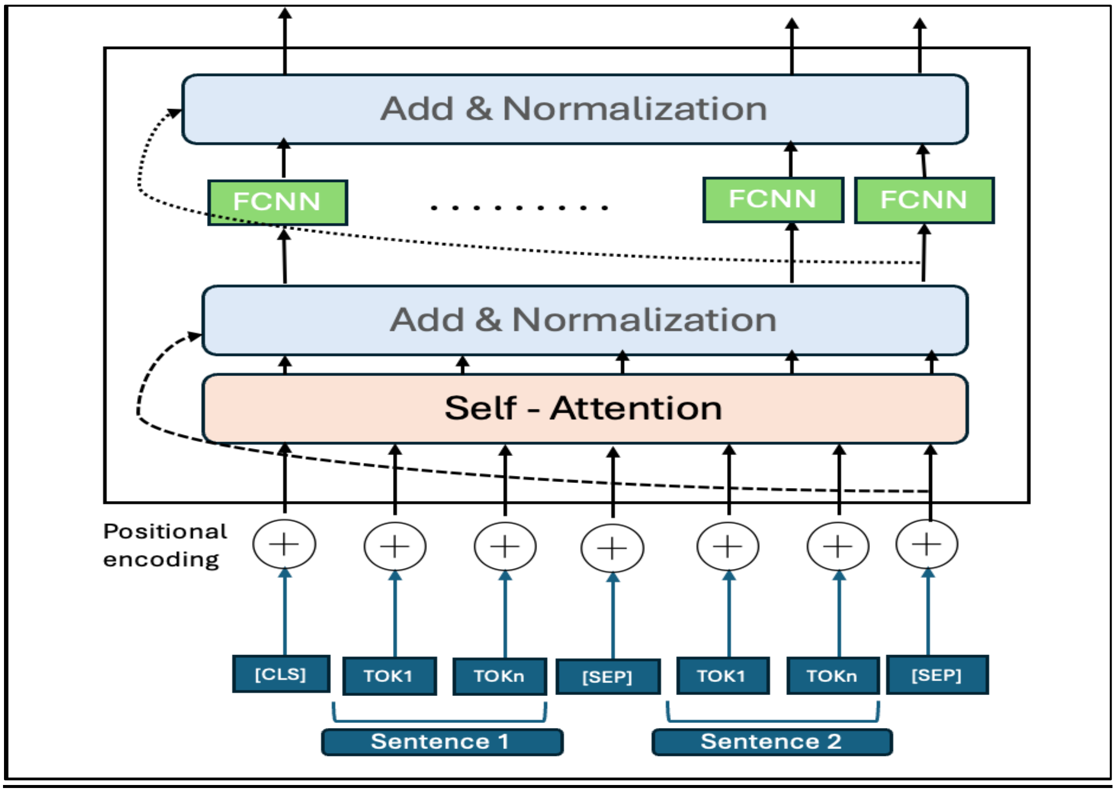
For this algorithm [4], I implemented a Dataset class to handle paired text data for the question similarity task. I used BertTokenizerFast to tokenize question pairs (question1 and question2) with padding, truncation, and a maximum token length of 256 (128 each from both the questions). The tokenized outputs included input_ids, attention_mask, and corresponding integer labels.

I used the tokenizer to format the two questions into a single sequence, allowing BERT to process them jointly and capture their interactions directly. Unlike the bi-encoder in Siamese networks, where each question is encoded independently, the cross-encoder combines the questions into the format:

[CLS] question1 [SEP] question2 [SEP]

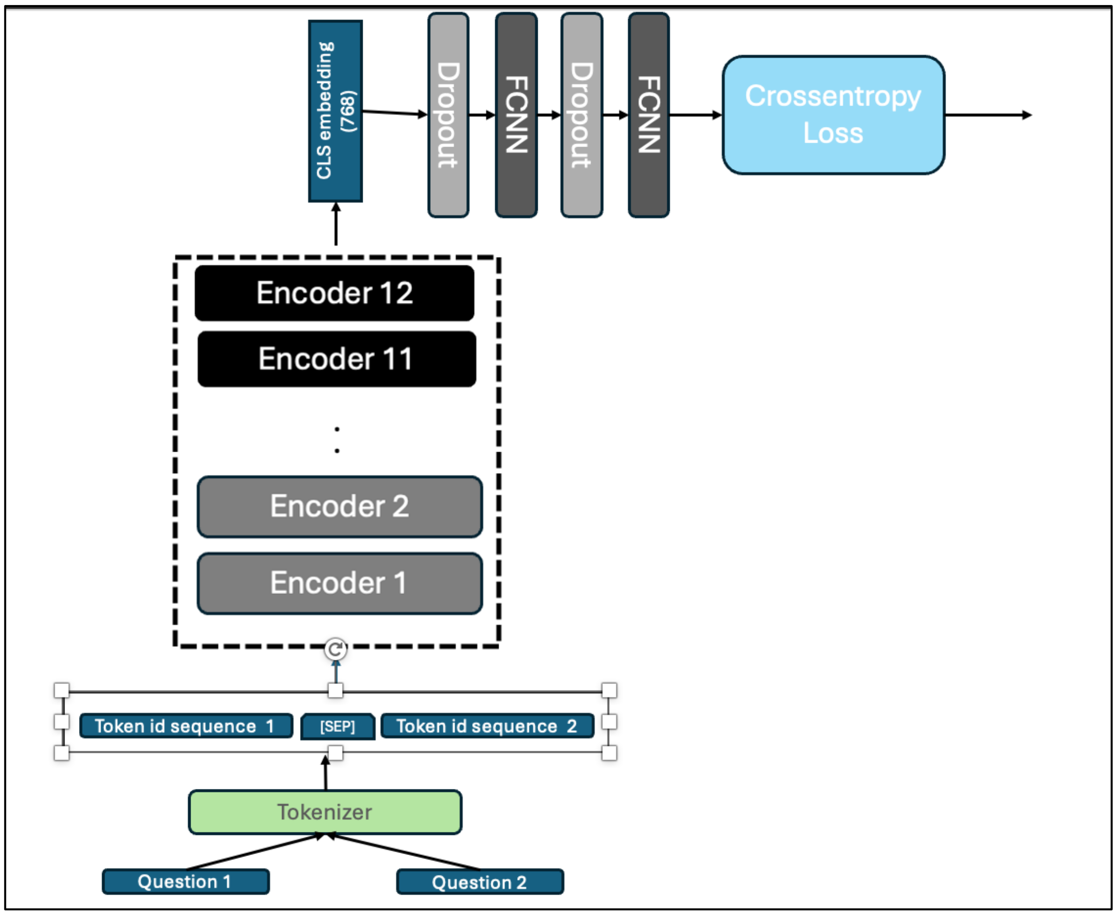
The tokenizer added the necessary special tokens ([CLS] and [SEP]), tokenized the sequence, and returned input_ids and attention_mask. Although token_type_ids were not explicitly returned, they were implicitly generated by the tokenizer to distinguish question1 (segment 0) from question2 (segment 1), ensuring BERT could differentiate the two inputs. I passed this formatted sequence through BERT, using the output embedding of the [CLS] token to represent the overall relationship between the two questions, enabling effective similarity classification.

3.4.2. Model Architecture



BERT Sequence Pair Architecture

The above is one encoder, showcasing how a tokenizer is adding a [SEP] between the question pairs. The outputs of each encoder are passed to the next encoder (there are a total of 12 encoders as we are using 'bert-base-uncased'). The entire architecture is shown below:

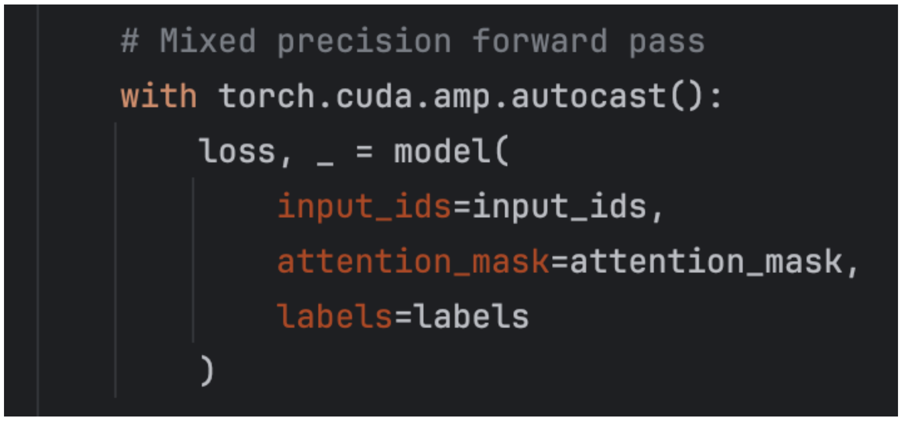


Cross-encoder BERT for Sequence Pair Classification task

I followed the above architecture, where the tokenizer handles the task of concatenation of the sentences using the [SEP] token embedding. As the token_id types were implicitly sent by the tokenizer to the encoder, it can understand that they are two separate sentences.

Of the 12 total encoders, 10 were frozen and two were kept un-frozen for selective fine-tuning. The output of the BERT encoder is the CLS embedding, which is again passed on to two Fully Dense Layers, which help to capture more non-linear relationships. At the end, two logit columns are obtained as outputs, which are then passed to the CrossEntropyLoss function for computation of the loss. Gradients are then updated through back-propagation.

I have employed two dropout layers to regularize my model, and I have chosen a learning rate of 3e-1, which was experimentally found to be the best for converging. The model was trained for 14 epochs and the validation F1 score obtained was 94.38%. Since the model is heavy computationally (the BERT is processing 256 tokens at a time) I used a special technique called mixed precision:

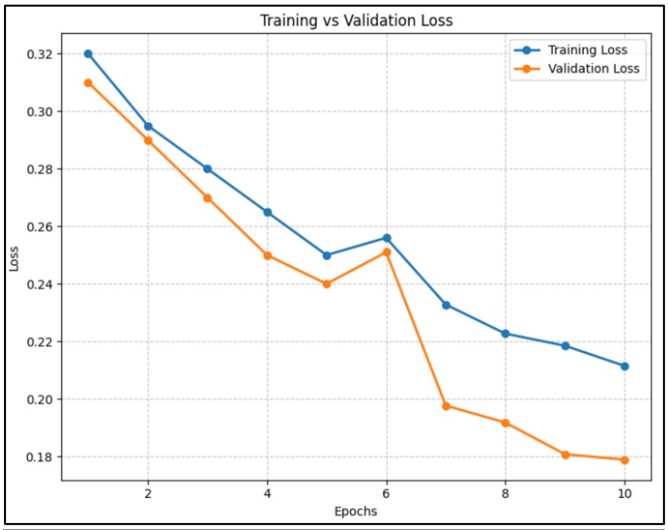


```
# Mixed precision forward pass
with torch.cuda.amp.autocast():
    loss, _ = model(
        input_ids=input_ids,
        attention_mask=attention_mask,
        labels=labels
    )
```

Mixed Precision

The 'torch.cuda.amp.autocast()'. It allows for the automatic management of mixed precision computations during the forward and backward passes of a neural network; it converts between FP16 (half-precision) and FP32 (single-precision) data types. By leveraging mixed precision, the model can take advantage of the performance and memory benefits of FP16 data types while maintaining the numerical stability of FP32 for critical operations.

<u>3.4.3 Results</u>



By training this model, I observed that the results were satisfactory and the model performed the best amongst all the models; the validation F1 macro score achieved was 93.28%.

RESULTS

Algorithm	F1-macro
SBERT+ dynamic masking	89.68%
SBERT+ contrastive loss	80.48%
BERT + contrastive loss	79.48%
Cross-Encoder BERT	**93.28%**

End-to-end performance table

By looking at the overall results it can be understood that the Cross-encoder BERT performed the best with a test F1 macro score of 93.28%.

SUMMARY AND CONCLUSION

In this project, I explored various transformer-based architectures to solve the Quora Question Pair Similarity challenge. My work primarily focused on implementing and comparing three main approaches: bi-encoder (Siamese) architectures with BERT/SBERT and a cross-encoder BERT architecture. Through these implementations, I gained valuable insights into the strengths and limitations of different transformer-based approaches for semantic similarity tasks.

My initial implementation of Siamese SBERT with dynamic masking achieved an F1-macro score of 89.68%. This architecture incorporated dynamic masking with a 15% probability as an auxiliary task, which notably improved the model's robustness. Through experimentation with different pooling strategies, I found that mean pooling consistently outperformed max and CLS pooling. The combination of main and auxiliary losses proved particularly effective. However, the model showed signs of overfitting around the eighth epoch, highlighting the importance of careful monitoring during training.

The embedding alignment approach using contrastive loss provided interesting insights into the relative performance of BERT and SBERT architectures. SBERT achieved an F1-macro score of 80.48%, slightly outperforming BERT's 79.48%. A key learning from this implementation was that the smaller SBERT model could achieve comparable results with significantly lower computational overhead. The contrastive loss implementation effectively aligned embeddings, while L2 normalization and cosine similarity proved crucial for comparing high-dimensional embeddings.

The cross-encoder BERT architecture emerged as the most effective approach, achieving the highest F1-macro score of 93.28%. This architecture's superior performance can be attributed to its ability to process question pairs simultaneously, enabling better capture of cross-question interactions. The implementation of mixed precision training proved crucial for managing the computational demands of this larger model. Selective fine-tuning of only two encoder layers helped balance performance and computational efficiency.

Through this project, I learned that while bi-encoder architectures offer computational efficiency advantages, the cross-encoder's ability to process question pairs simultaneously leads to superior performance in semantic similarity tasks. The comparable performance of simpler architectures like SBERT to larger models in specific tasks highlighted the importance of choosing architectures based on both performance metrics and computational requirements. The success of the cross-encoder BERT model, achieving a 93.28% F1-macro score, validates the effectiveness of transformer-based architectures in capturing complex semantic relationships in text similarity tasks.

Code Source Analysis

Toral lines of code: 1480

Total lines of code copied: 745

Total non copied lines: 49.66%

References

[1] https://quoradata.quora.com/First-Quora-Dataset-Release-Question-Pairs

[2] Reimers, N. (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. *arXiv preprint arXiv:1908.10084*.

[3] Wang, F., & Liu, H. (2021). Understanding the behavior of contrastive loss. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition* (pp. 2495-2504).

[4] https://www.sbert.net/examples/applications/cross-encoder/README.html