NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

SC4002 Group Project

Group number: **Group 25**

Group member	Contribution
Loh Xian Jun, Daryl	Part 1, Part 3
Liau Zheng Wei	Part 2, Part 1
Yip Jun Kai	Part 3, Part 2
Ethan Yeo	Part 1, Part 3
Christopher Angelo	Part 2, Part3

Part 1: Preparing Word Embeddings

This part focuses on preparing word embeddings using pre-trained vectors (GloVe) for the TREC question classification task.

(1a) Vocabulary Size:

- The vocabulary size formed from the training data using spaCy tokenization (en_core_web_sm) is **8106**.

(1b) OOV Words:

- **Total OOV words:** There are **3106** words in the training data vocabulary that are not present in the pre-trained GloVe (glove-wiki-gigaword-100) dictionary.
- **OOV words per topic category (unique tokens):**

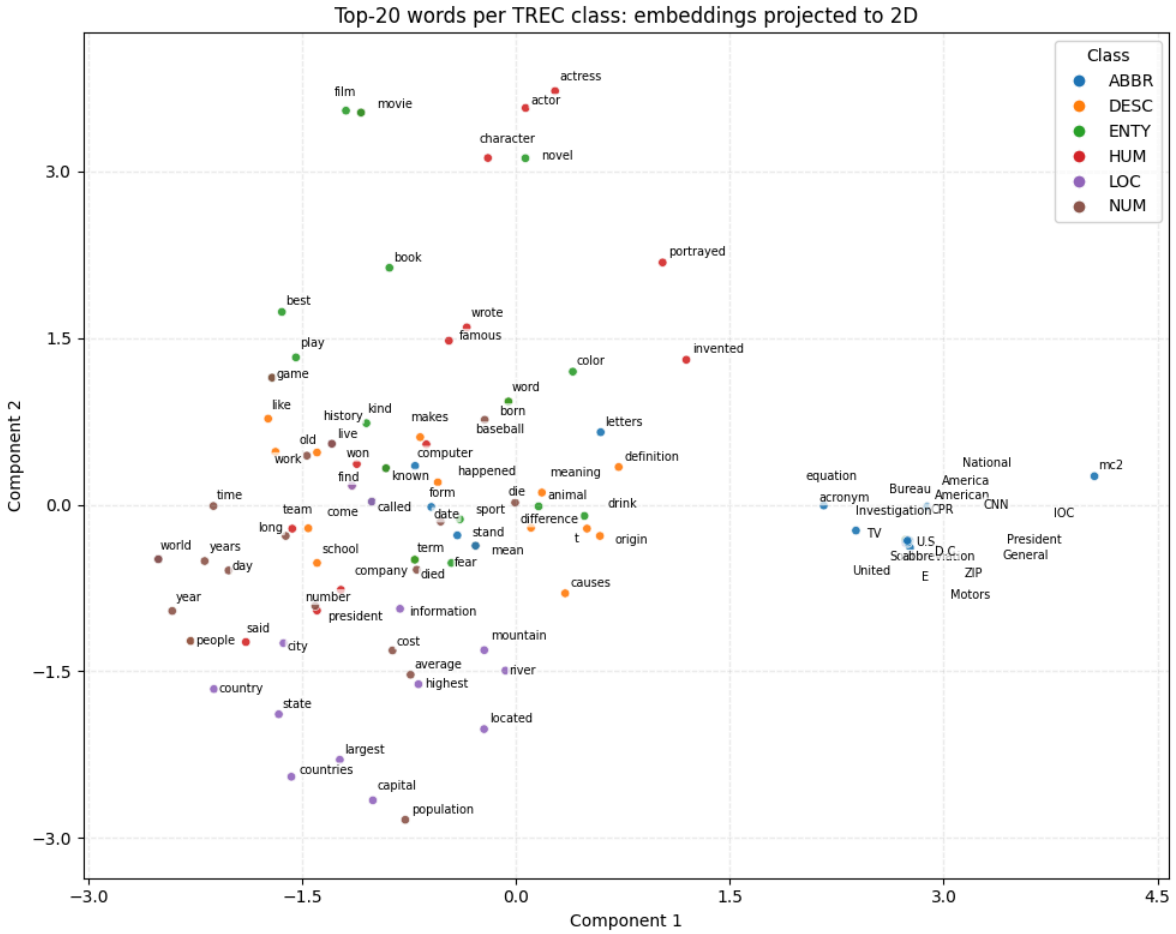
	HUM	DESC	ENTY	LOC	NUM	ABBR
Count	1302	631	888	728	623	71

(1c) OOV Mitigation Strategy (Appendix A):

- **Strategy:** The chosen strategy combines **Subword Hashing (Character n-grams)** with **fine-tuning the UNK embedding**.
  - For OOV tokens encountered during runtime, an embedding is composed by hashing character n-grams (range 3-5) using MD5 into a fixed number of buckets (20000). This subword vector is averaged with the trainable UNK vector.
  - The base embedding layer, initialized with GloVe vectors, is **unfrozen** (freeze=False) during training to allow adaptation to the specific domain vocabulary of the TREC dataset.
- **Rationale:** This approach preserves some semantic information for rare words or morphological variations not present in GloVe and provides a learnable signal even for OOV tokens. Fine-tuning allows the model to adjust known embeddings for better task performance.
- **Effectiveness:** Training logs show successful convergence with this strategy, achieving a test accuracy of approximately 91.0% with a BiLSTM model after 10 epochs (best validation accuracy at epoch 8).

(1d) Embedding Visualization:

- **Method:** The top 20 most frequent words (excluding stopwords and punctuation) from each topic category were selected. Their GloVe embeddings (or subword+UNK blend for OOV) were retrieved and projected into 2D space using **PCA**.



- **Analysis:**
  - The PCA plot shows noticeable clustering based on topic semantics. For instance, location-related terms (LOC) form a relatively distinct group, as do numeric terms (NUM) and descriptive terms (DESC).
  - There is some overlap between categories, particularly ENTY (Entity) and HUM (Human), likely due to shared contexts (e.g., questions about people who are entities) and lexical ambiguity.
  - The clusters are not perfectly linearly separable in 2D, suggesting that a classifier would need to capture more complex, potentially non-linear relationships or leverage sequential information (like an RNN) to achieve high accuracy.

Part 2: Model Training & Evaluation - RNN

This part involves training and evaluating a BiLSTM model using the prepared embeddings.

(2a) Best RNN Configuration:

Based on the grid search performed, the best configuration identified using early stopping on the validation set was:

- **Model Type:** BiLSTM (implicitly, as only BiLSTM was used in grid search)
- **Sentence Pooling:** final
- **Hidden Dimension:** 128
- **Dropout:** 0.3
- **Weight Decay (L2 Regularization):** 0.0001
- **Optimizer:** Adam
- **Learning Rate:** 0.001
- **Batch Size:** 64
- **Best Epoch (selected based on validation accuracy):** 7

(2b) Regularization Strategies Comparison:

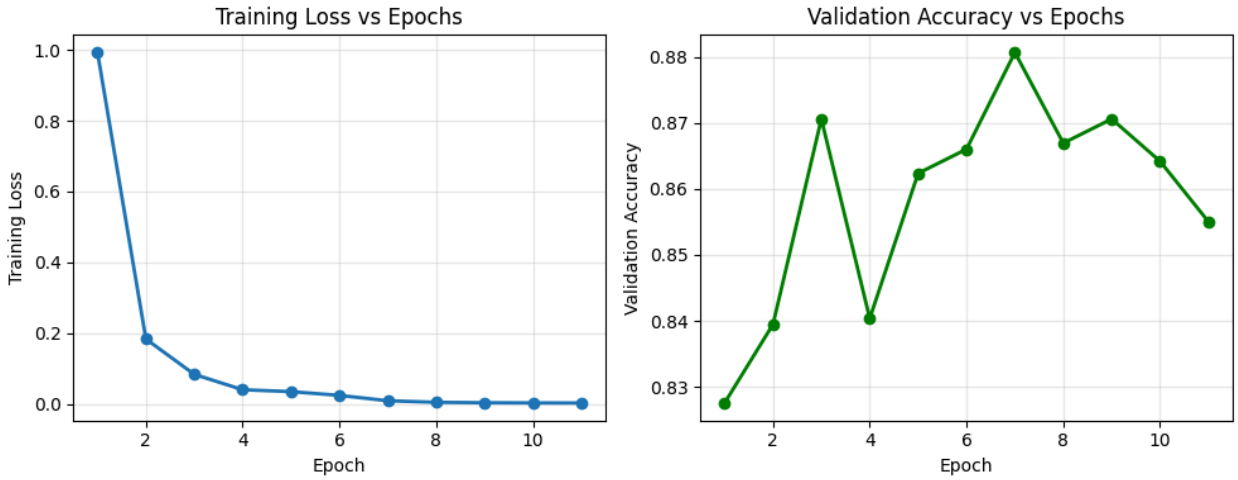
The following regularization strategies were compared within the grid search, primarily varying dropout and weight decay (L2):

Test Accuracy	Dropout = 0	Dropout = 0.3	Dropout = 0.5
Weight Decay = 0	0.866	0.882	0.878
Weight Decay = 0.0001	0.906	0.874	0.884

- **Observations:**
  - Applying some form of regularization generally yielded improvements or comparable results to no regularization, preventing significant overfitting.
  - Weight decay alone (wd=0.0001) with no dropout performed very well, achieving the highest test accuracy for the final pooling, hidden=128 configuration.
  - Combining high dropout (0.5) with weight decay also achieved a high test accuracy (0.884), suggesting this combination effectively regularized the larger model.
  - The optimal regularization strategy depends on other hyperparameters (like hidden size and pooling method). Across all configurations tested, strategies involving weight decay (either alone or combined with dropout) frequently appeared among the top performers.

(2c) Training Dynamics (Best Configuration):

- **Best Configuration:** As reported in (2a)



- **Discussion:**
  - **Training Loss:** The training loss curve typically shows a rapid decrease in the initial epochs, indicating the model is quickly learning patterns from the data. It continues to decrease, potentially at a slower rate, as training progresses.
  - **Validation Accuracy:** The validation accuracy curve generally increases sharply early on and then plateaus or starts to slightly decrease if overfitting occurs. Early stopping selects the model state corresponding to the peak validation accuracy (epoch 7 in this case).
  - **Dynamics:** The curves suggest that the model learned effectively in the first few epochs. The use of early stopping prevented overfitting by stopping training when validation performance ceased to improve, selecting the model from epoch 4 which generalized best to unseen validation data. The combination of Adam optimization, appropriate learning rate, and mild L2 regularization contributed to stable convergence.

(2d) Sentence Representation Strategies:

The following pooling strategies were implemented to aggregate hidden states from the BiLSTM into a single sentence representation vector:

1. **final:** Concatenating the final hidden states of the forward and backward LSTM passes.
2. **mean:** Mean-pooling the BiLSTM output hidden states across the sequence length (considering padding).
3. **max:** Max-pooling the BiLSTM output hidden states across the sequence length (considering padding).
4. **meanmax:** Concatenating the results of mean-pooling and max-pooling.

Test Accuracy Scores (Averaged across different regularization/hidden size settings):

- final: **0.8780** (Highest average)
- mean: 0.8677
- max: 0.8688
- meanmax: 0.8712

Discussion: Final hidden state pooling (final) achieved the highest average test accuracy across the configurations tested. This suggests that for the TREC question classification task, the last hidden state of the BiLSTM captures the most relevant information for classifying question types. This indicates that the sequential processing of the BiLSTM effectively accumulates and preserves the most important

contextual information in its concluding representations. Mean pooling and max pooling performed reasonably well but they were outperformed by the final state approach, suggesting that averaging across all time steps or taking maximum activations may introduce noise from less relevant words in the question context. The concatenation of mean and max pooling (meanmax) showed intermediate performance, better than either method alone but still below the final state approach, indicating that while combining different pooling strategies can be beneficial but not beneficial enough to surpass the effectiveness of the carefully accumulated final hidden state representation for this specific question classification task.

(2e) Topic-wise Accuracy (Best Model):

- **Best Model:** Configuration from (2a).

Topic-wise Test Accuracies:

  - HUM (Human): 0.9077
  - ENTY (Entity): 0.7553
  - DESC (Description): 0.9638
  - NUM (Numeric): 0.8230
  - LOC (Location): 0.8395
  - ABBR (Abbreviation): 0.7778

Discussion on Differences:

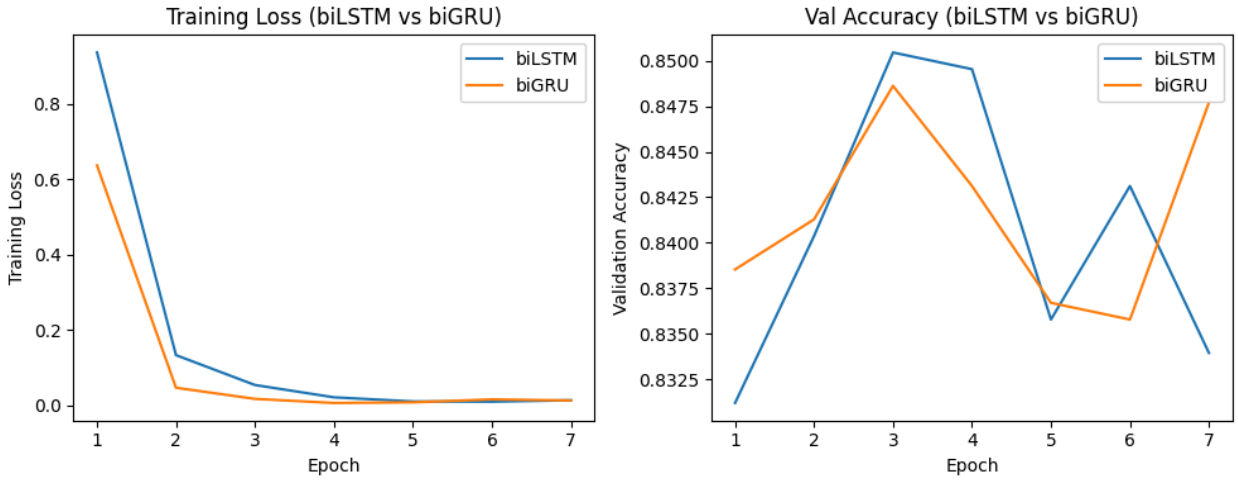
- **High Accuracy (DESC):** Description questions likely have very distinct linguistic patterns (e.g., "What is...", "Define...") making them easier to classify, resulting in near-perfect accuracy.
- **Moderate Accuracy (HUM, NUM, LOC):** These categories seem reasonably distinguishable, possibly using characteristic question words (Who, How many/much, Where) and named entities.
- **Lower Accuracy (ENTY, ABBR):**
  - **ENTY:** Entity questions might overlap significantly with HUM (people are entities) or DESC (definitions of entities), leading to confusion. They might also cover a very broad range of topics. The relatively lower OOV coverage noted in Part 1 for ENTY compared to HUM might also play a role if crucial entity names are missed.
  - **ABBR:** Abbreviation questions are likely the least frequent category (as suggested by the low OOV count in Part 1) and often very short, providing fewer contextual clues for the BiLSTM to capture, making them inherently harder.
- **Other Factors:** Differences in the number of training examples per category (class imbalance) and the inherent ambiguity or subtlety of questions within certain topics can also contribute to accuracy variations.

Part 3: Enhancement

This part explores improvements over the basic BiLSTM model.

(3a) BiLSTM vs BiGRU:

- **Models Compared:**
  - BiLSTM (2 layers, hidden=128, dropout=0.5, meanmax pooling)
  - BiGRU (2 layers, hidden=128, dropout=0.5, meanmax pooling)

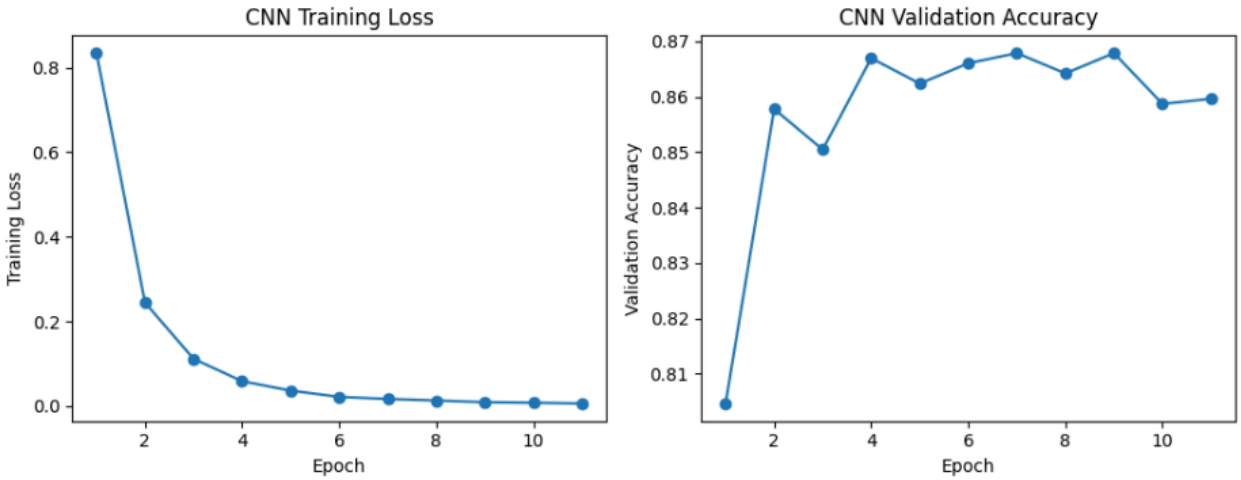


Test Accuracies:

- BiLSTM: **0.860** (Best validation accuracy at epoch 3)
- BiGRU: **0.868** (Best validation accuracy at epoch 3)
- **Discussion:** In this specific comparison, the BiGRU slightly outperformed the BiLSTM on the test set, although both achieved comparable validation accuracies. The training dynamics show both models learning quickly, with BiLSTM perhaps converging slightly faster in terms of validation performance in this run. BiGRUs are often computationally slightly lighter, but BiLSTMs sometimes capture longer-range dependencies more effectively, which might explain the minor performance difference here.

(3b) CNN Model:

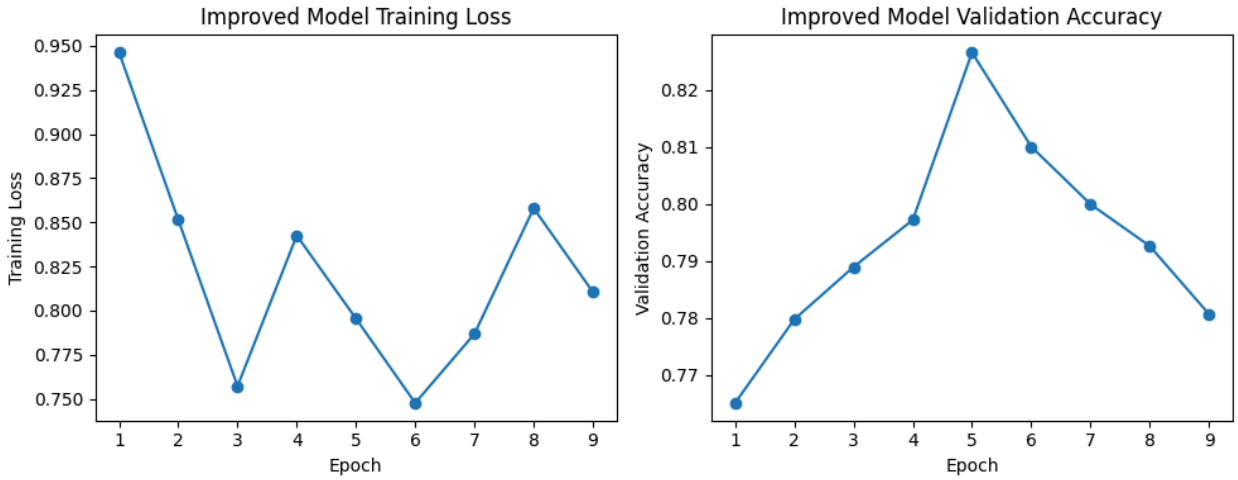
- **Model:** Kim-style TextCNN with multiple filter widths (3, 4, 5) and 128 filters per width, using max-pooling over time for each filter map, followed by dropout (0.5) and a fully connected layer. Embeddings were the same fine-tuned GloVe + subword hashing.



- **Test Accuracy: 0.892** (Best validation accuracy at epoch 5)
- **Discussion:** The TextCNN model achieved a strong test accuracy, outperforming the BiGRU and the BiLSTM results from earlier parts in this specific run. The training loss decreased steadily, and validation accuracy peaked around epoch 7 before plateauing. This indicates that CNNs, by capturing local n-gram features through different filter sizes, are also highly effective for this sentence classification task, potentially being faster to train than RNNs.

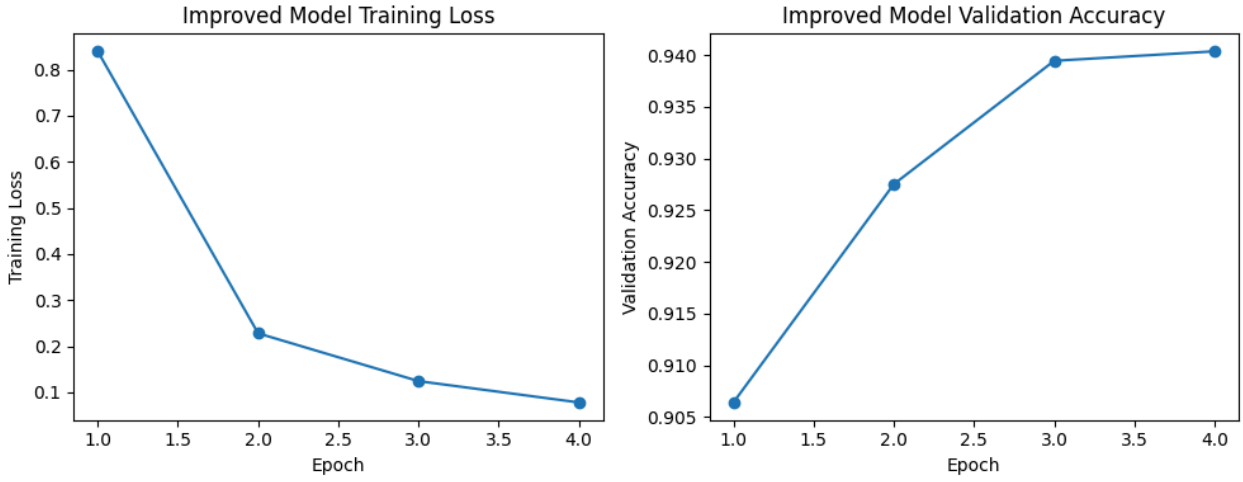
(3c) Improvement Strategies:

- **Strategy 1:** Combined several techniques:
  1. **Self-Attentive Pooling:** Replaced standard pooling (mean, max, etc.) with a self-attention mechanism over the BiGRU hidden states. This allows the model to learn the importance of different words/hidden states when forming the sentence representation. (Implemented in AttnBiGRU class).
  2. **Focal Loss:** Replaced standard Cross-Entropy loss with Focal Loss (gamma=2.0). This helps focus training on harder-to-classify examples and addresses potential class imbalance by down-weighting easy examples. Class weights (alpha parameter, based on inverse frequency) were also included.
  3. **Mixup:** Applied Mixup regularization in the embedding space (alpha_mix=0.2) during training. This involves creating synthetic training examples by taking convex combinations of pairs of examples and their labels, which can improve generalization and model calibration.
- **Model Architecture:** Used a 2-layer BiGRU with hidden size 128 and dropout 0.5, incorporating the self-attention mechanism (AttnBiGRU).



- **Test Accuracy: 0.868** (Best validation accuracy at epoch 5)
- **Discussion:** While self-attention, focal loss, and mixup are advanced techniques often leading to improvements, in this specific experiment run, the result was lower than the simpler TextCNN or some BiGRU configurations. The training curves show reasonable convergence, but validation accuracy plateaued at a lower level compared to the CNN. This could be due to hyperparameter interactions, the specific implementation details, or the dataset characteristics potentially not benefiting as much from this particular combination compared to simpler models. Further tuning might be needed to optimize this combined approach.

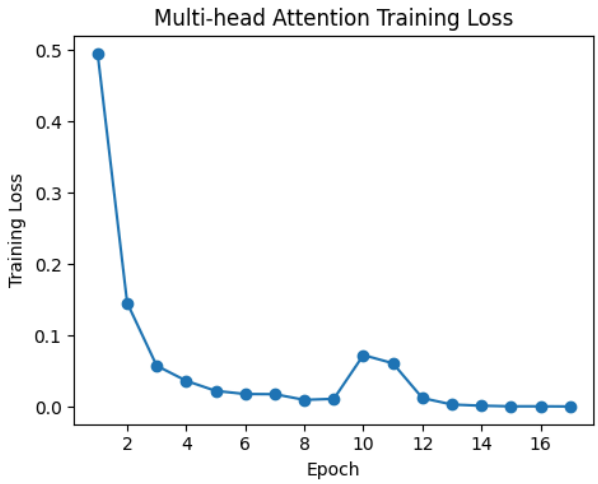
- **Strategy 2:** Fine tuning a pretrained transformer
- **Model Architecture:** Used Distilled version of BERT consisting of 6 transformer layers with attention mechanisms, hidden size of 768, learning rate of 2e-5, and weight decay of 0.01.

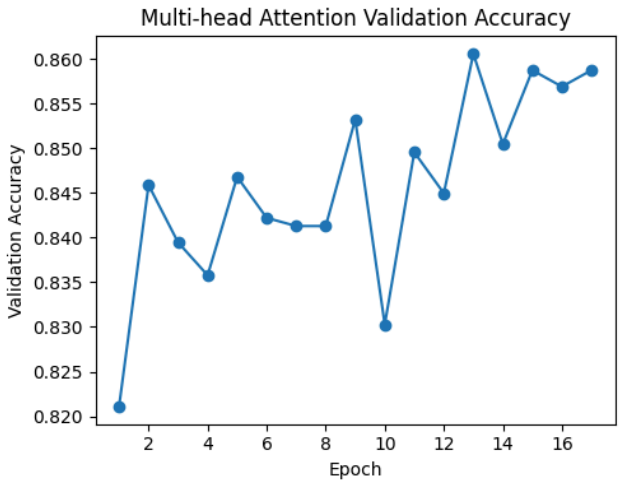


- **Test Accuracy: 0.972** (Best validation accuracy at epoch 4)
- **Discussion:** Despite the use of a max epoch of 4, significant improvements can be seen in the validation accuracy. Through the use of a pretraining transformer, large-scale language understanding acquired during pretraining on massive text corpora (e.g., Wikipedia, books) can be leveraged. Transformers like BERT already encode rich contextual word representations through

self-attention, which captures dependencies between all words in a sentence simultaneously, instead of pretrained word embeddings (Word2Vec or Glove) which usually only captures the meaning of the words themselves with no context. Fine-tuning then simply adapts this general knowledge to the specific classification task with minimal additional training, resulting in faster convergence, stronger generalization, and substantially higher accuracy, especially when labeled data is small like in this scenario.

- **Strategy 3:** Multi-head self attention
- **Model Architecture:** Multi-head Self-Attention BiGRU with bidirectional GRU layers (2 layers, hidden size 128), multi-head self-attention (4 heads), layer normalization, dropout 0.5, learning rate 1e-3, and weight decay 1e-4.





- **Test Accuracy: 0.856** (Best validation accuracy at epoch 13)
- **Discussion:** Multi-head self-attention improves representation by attending to multiple aspects of the input simultaneously. Each head captures different relationships, helping identify patterns in challenging topics like ENTY, ABBR, and NUM. Layer normalization stabilizes training by normalizing activations, reducing internal covariate shift, and improving gradient flow. Combined with bidirectional GRU, this captures sequential dependencies and complex token interactions. The model learns richer sentence representations than simple pooling methods, leading to better generalization, especially for weak topics that benefit from multiple attention perspectives.

(3d) Weak Topic Improvement Strategy:

Strategy	**Model**
Weighted Cross-Entropy loss	BiGRU with meanmax pooling (similar to the best baseline from Part 2, but using GRU and potentially different hyperparameters for this run).
The strategy of applying a weighted cross-entropy loss to the Bi-GRU model yielded mixed results when compared to the baseline. It achieved significant performance gains for the HUM (0.9692) and NUM (0.8761) categories. However, this came at the cost of reduced accuracy for ENTY (0.7234), DESC (0.9493), and LOC (0.8272). Performance for the ABBR category remains unchanged (0.7778).	

Fine tuning a pretrained transformer	Distilled version of BERT consisting of 6 transformer layers with attention mechanisms, hidden size of 768, learning rate of 2e-5, and weight decay of 0.01.
Fine-tuning a pretrained transformer proved to be the most effective strategy, delivering superior performance across all categories without exception. This model established new best scores for HUM (0.9846), ENTY (0.9149), DESC (0.9855), NUM (0.9823), and LOC (0.9877). Most impressively, it achieved a perfect score of 1.0000 for the ABBR category, demonstrating its comprehensive ability to master the classification task.	
Multi-head Self-Attention with Layer Normalization	Multi-head Self-Attention BiGRU with layer normalization
The multi-head self-attention approach produced largely unfavorable results compared to the baseline, with only one category showing meaningful improvement. While performance for DESC improved to 0.9710 and NUM saw a marginal increase to 0.8319, there was a notable drop in accuracy for the HUM (0.8769) and LOC (0.8025) categories. Performance for both ENTY (0.7553) and ABBR (0.7778) remained identical to the baseline.	

Topic	Part 2e - Best	Bi-GRU	Transformer	Multi-head
HUM	0.9077	0.9692	**0.9846**	0.8769
ENTY	0.7553	0.7234	**0.9149**	0.7553
DESC	0.9638	0.9493	**0.9855**	0.9710
NUM	0.8230	0.8761	**0.9823**	0.8319
LOC	0.8395	0.8272	**0.9877**	0.8025
ABBR	0.7778	0.7778	**1.0000**	0.7778

APPENDIX A

```
import torch.nn as nn
import torch.nn.functional as F
import hashlib

class SubwordHashingEmbedder(nn.Module):
    def __init__(self, base_vectors, vocab, emb_dim=100,
num_buckets=20000, ngram_range=(3,5), unk_idx=None):
        super().__init__()
        self.emb_dim = emb_dim
        self.vocab = vocab
        self.unk_idx = unk_idx if unk_idx is not None else
vocab.stoi.get(vocab.default_token, 0)
        # Trainable base embedding initialized with pretrained vectors
        self.base = nn.Embedding.from_pretrained(base_vectors,
freeze=False)  # unfreeze to adapt domain
        # Trainable subword buckets
        self.ngram_min, self.ngram_max = ngram_range
        self.num_buckets = num_buckets
        self.subword_table = nn.Embedding(num_buckets, emb_dim)
        nn.init.uniform_(self.subword_table.weight, a=-0.05, b=0.05)

    def _hash(self, s):
        return int(hashlib.md5(s.encode('utf-8')).hexdigest(), 16) %
self.num_buckets

    def subword_embed(self, token):
        token = f"<{token}>"
        vecs = []
        for n in range(self.ngram_min, self.ngram_max+1):
            for i in range(len(token)-n+1):
                ngram = token[i:i+n]
                h = self._hash(ngram)
                vecs.append(self.subword_table.weight[h])
        if len(vecs) == 0:
            return self.subword_table.weight[self._hash(token)]
        return torch.stack(vecs, dim=0).mean(dim=0)

    def forward_token_ids(self, ids):
        # ids: LongTensor [seq_len] or [batch, seq_len]
        base_emb = self.base(ids)
        return base_emb

    def forward_tokens(self, tokens):
```

```
        # tokens: list[str], return [len, emb_dim]
        device = self.base.weight.device
        embs = []
        for tok in tokens:
            idx = self.vocab.stoi.get(tok, self.unk_idx)
            if idx < len(self.base.weight):
                base_vec = self.base.weight[idx]
                if torch.all(base_vec == 0):  # treat zero row as OOV
in gensim path
                    sw = self.subword_embed(tok).to(device)
                    unk_vec = self.base.weight[self.unk_idx]
                    embs.append(0.5*sw + 0.5*unk_vec)
                else:
                    embs.append(base_vec)
            else:
                sw = self.subword_embed(tok).to(device)
                unk_vec = self.base.weight[self.unk_idx]
                embs.append(0.5*sw + 0.5*unk_vec)
        return torch.stack(embs, dim=0)

# Ensure UNK and PAD exist
if '<unk>' not in TEXT.vocab.stoi:
    TEXT.vocab.itos.insert(0, '<unk>')
    TEXT.vocab.stoi['<unk>'] = 0
if '<pad>' not in TEXT.vocab.stoi:
    TEXT.vocab.itos.insert(1, '<pad>')
    TEXT.vocab.stoi['<pad>'] = 1

unk_idx = TEXT.vocab.stoi['<unk>']
pad_idx = TEXT.vocab.stoi['<pad>']

# If vectors row for UNK is zero, init it small random
if vectors.shape[0] <= unk_idx or torch.all(vectors[unk_idx] == 0):
    with torch.no_grad():
        vectors = torch.cat([vectors, torch.zeros(2, emb_dim)], dim=0)
if vectors.shape[0] <= 1 else vectors
        if vectors.shape[0] <= unk_idx:
            pad_rows = unk_idx - vectors.shape[0] + 1
            vectors = torch.cat([vectors, torch.zeros(pad_rows,
emb_dim)], dim=0)
        nn.init.uniform_(vectors[unk_idx], a=-0.05, b=0.05)

embedder = SubwordHashingEmbedder(vectors, TEXT.vocab,
emb_dim=emb_dim, num_buckets=20000, ngram_range=(3,5),
unk_idx=unk_idx).to(device)


# Iterators with lengths
```

```
BATCH_SIZE = 64
train_iter, valid_iter, test_iter = BucketIterator.splits(
    (train_data, valid_data, test_data),
    batch_size=BATCH_SIZE,
    sort_within_batch=True,
    sort_key=lambda x: len(x.text),
    device=device
)  # classic torchtext iterators


# Simple classifier (BiLSTM) using our embedding layer

class BiLSTMClassifier(nn.Module):
    def __init__(self, embedder, hidden_size=128, num_layers=1,
num_classes=len(LABEL.vocab), dropout=0.3):
        super().__init__()
        self.embedder = embedder
        self.embedding_dim = embedder.emb_dim
        self.lstm = nn.LSTM(self.embedding_dim, hidden_size,
num_layers=num_layers, batch_first=True, bidirectional=True)
        self.dropout = nn.Dropout(dropout)
        self.fc = nn.Linear(2*hidden_size, num_classes)

    def forward(self, text, lengths):
        # text: [seq_len, batch]; convert to [batch, seq_len] tokens
ids → embeddings
        text = text.transpose(0,1)  # [batch, seq_len]
        # Map token ids to embeddings via base table directly (fast
path)
        emb = self.embedder.forward_token_ids(text)  # [batch,
seq_len, emb_dim]
        # Pack and LSTM
        lengths_cpu = lengths.cpu()
        packed = nn.utils.rnn.pack_padded_sequence(emb, lengths_cpu,
batch_first=True, enforce_sorted=False)
        outputs, (h, c) = self.lstm(packed)
        # Concatenate final forward/backward hidden
        h_cat = torch.cat([h[-2], h[-1]], dim=1)  # [batch, 2*hidden]
        out = self.dropout(h_cat)
        logits = self.fc(out)
        return logits

num_classes = len(LABEL.vocab)
model = BiLSTMClassifier(embedder, hidden_size=128, num_layers=1,
num_classes=num_classes, dropout=0.3).to(device)
```