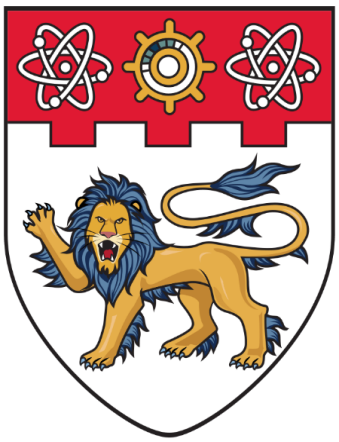


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

AY 2025/2026 Semester 1
SC4002 Natural Language Processing

Group Assignment

Group Number: 35

Name	Matriculation No.	Contributions
Tay Wei Yang	U2221303F	Part 1
Lim Xin Wei	U2240093B	Part 2
Pearlina Tan Qinlin	U2221690F	Part 2
Lim Jia Jie, Isaac	U2222066A	Part 3
Jefferson Ho Qi Yong	U2210058H	Part 3

Part 0: Dataset Preparation

We utilized the **TREC Question Classification** dataset, which contains a collection of open-domain questions labeled into six broad categories: **HUM** (Human), **ENTY** (Entity), **DESC** (Description), **NUM** (Numerical), **LOC** (Location), and **ABBR** (Abbreviation).

Tokenization & Normalization

We tokenize with spaCy en_core_web_sm. Lowercasing standardizes surface forms and improves coverage of GloVe 6B vectors, with punctuation preserved for question syntax which can be predictive for TREC classes.

```
TEXT = data.Field(tokenize='spacy',
                  tokenizer_language='en_core_web_sm',
                  include_lengths=True,
                  lower=True)

LABEL = data.LabelField()

train_data, test_data = datasets.TREC.splits(TEXT, LABEL, fine_grained=False)

print("Example:", vars(train_data.examples[0]))
print(f"Training samples: {len(train_data)}")
print(f"Test samples: {len(test_data)}")
```

```
Example: {'text': ['how', 'did', 'serfdom', 'develop', 'in', 'and', 'then', 'leave', 'russia', '?'], 'label': 'DESC'}
Training samples: 5452
Test samples: 500
```

Train/Val/Test Split

After loading, we split the data into **training**, **validation**, and **test** sets following an 80 : 20 training-validation split on the original training data to allow hyperparameter tuning and early-stopping based on validation performance.

```
train_data, valid_data = train_data.split(split_ratio=0.8, random_state=random.seed(SEED))

print(f"Training: {len(train_data)}")
print(f"Validation: {len(valid_data)}")
print(f"Test: {len(test_data)}")
```

```
Training: 4362
Validation: 1090
Test: 500
```

Part 1. Preparing Word Embeddings

Embedding Source

We use GloVe 6B 300-dim vectors (glove-wiki-gigaword-300) for strong general-domain coverage and stable performance on short queries. Higher dims (300d) help when classes hinge on sparse lexical cues such as ABBR.

PAD and Trainability

<pad> is initialized to the zero vector, while all other tokens load pre-trained vectors when available. Embeddings remain trainable to allow domain adaptation on TREC while keeping GloVe as initialization.

OOV Construction

For tokens not found in GloVe we build a fallback vector:

1. Try lowercase form,
2. Optionally average any available sub-candidates,
3. Fall back to a small Gaussian N(0, 0.1) if none resolve.

This preserves coverage without collapsing everything to a single <unk> class, improving robustness on rare proper nouns. A single <unk> erases potentially useful distinctions between different unseen words, hence our per-token fallback preserves variance and empirically stabilizes validation accuracy.

(a) What is the size of the vocabulary formed from your training data according to your tokenization method?

```
vocab_size = len(TEXT.vocab)
print(f"Answer: Vocabulary size = {vocab_size}")
```

```
Answer: Vocabulary size = 7435
```

(b) We use OOV (out-of-vocabulary) to refer to those words that appear in the training data but not in the Word2vec (or Glove) dictionary. How many OOV words exist in your training data? What is the number of OOV words for each topic category?

```
oov_words = []
for word in TEXT.vocab.itos:
    if word not in glove:
        oov_words.append(word)

print(f"Total OOV words: {len(oov_words)}")
print(f"Percentage: {len(oov_words)/vocab_size*100:.2f}%")
print(f"Sample OOV words: {oov_words[:20]}")
```

```
Total OOV words: 192
```

```
oov_per_topic = defaultdict(set)

for example in train_data.examples:
    label = example.label
    for word in example.text:
        if word not in glove:
            oov_per_topic[label].add(word)

print("OOV words per topic:")
for topic, words in sorted(oov_per_topic.items()):
    print(f"{topic}: {len(words)} unique OOV words")
```

```
OOV words per topic:
ABBR: 4 unique OOV words
DESC: 68 unique OOV words
ENTY: 47 unique OOV words
HUM: 39 unique OOV words
LOC: 15 unique OOV words
NUM: 21 unique OOV words
```

(c) The existence of the OOV words is one of the well-known limitations of Word2vec (or Glove). Without using any transformer-based language models (e.g., BERT, GPT, T5), what do you think is the best strategy to mitigate such limitations? Implement your solution in your source code. Show the corresponding code snippet.

```
def create_embedding_matrix_with_oov_handling(vocab, glove_model, embedding_dim):
    vocab_size = len(vocab)
    embedding_matrix = np.zeros((vocab_size, embedding_dim))

    oov_count = 0

    for idx, word in enumerate(vocab.itos):
        if word == '<pad>':
            embedding_matrix[idx] = np.zeros(embedding_dim)
        elif word in glove_model:
            embedding_matrix[idx] = glove_model[word]
        else:
            oov_count += 1
            similar_embeddings = []

            # Character n-grams
            if len(word) > 3:
                for i in range(len(word) - 2):
                    trigram = word[i:i+3]
                    if trigram in glove_model:
                        similar_embeddings.append(glove_model[trigram])

            # Remove special characters
            cleaned_word = ''.join(c for c in word if c.isalnum())
            if cleaned_word and cleaned_word in glove_model:
                similar_embeddings.append(glove_model[cleaned_word])

            # Lowercase
            if word.lower() in glove_model:
                similar_embeddings.append(glove_model[word.lower()])

            if similar_embeddings:
                embedding_matrix[idx] = np.mean(similar_embeddings, axis=0)
            else:
                embedding_matrix[idx] = np.random.normal(0, 0.1, embedding_dim)

    print(f"OOV words handled: {oov_count}")
    return embedding_matrix

embedding_matrix = create_embedding_matrix_with_oov_handling(TEXT.vocab, glove, embedding_dim)
print(f"Embedding matrix shape: {embedding_matrix.shape}")
```

```
OOV words handled: 191
Embedding matrix shape: (7435, 300)
```

OOV is a known limitation, without mitigating it, many tokens map to UNK which leads to loss of information. We combine multiple techniques into a single flow whereby it stacks layers of strategy to address this issue.

1. Use pretrained embeddings for known tokens
2. For unseen tokens, compute the average character ngram embedding
3. Remove special character to handle noisy variations
4. Convert to a lowercase to handle mismatches
5. Assign a random vector so no word is left unpresented.

This ensures that every token in the vocab has a usable embedding even if it's unseen in a pretrained GloVe set.

(d) Select the 20 most frequent words from each topic category in the training set (removing stopwords if necessary). Retrieve their pretrained embeddings (from Word2Vec or GloVe). Project these embeddings into 2D space (using e.g., t-SNE or Principal Component Analysis).

```
# Get top words per topic
nlp = spacy.load('en_core_web_sm')
stopwords = nlp.Defaults.stop_words

topic_word_counts = defaultdict(Counter)

for example in train_data.examples:
    label = example.label
    for word in example.text:
        if word not in stopwords and word in glove:
            topic_word_counts[label][word] += 1

all_words = []
all_labels = []
all_embeddings = []

for topic, word_counts in topic_word_counts.items():
    top_words = [word for word, count in word_counts.most_common(20)]
    for word in top_words:
        all_words.append(word)
        all_labels.append(topic)
        all_embeddings.append(glove[word])

print(f"Total words for visualization: {len(all_words)}")
```

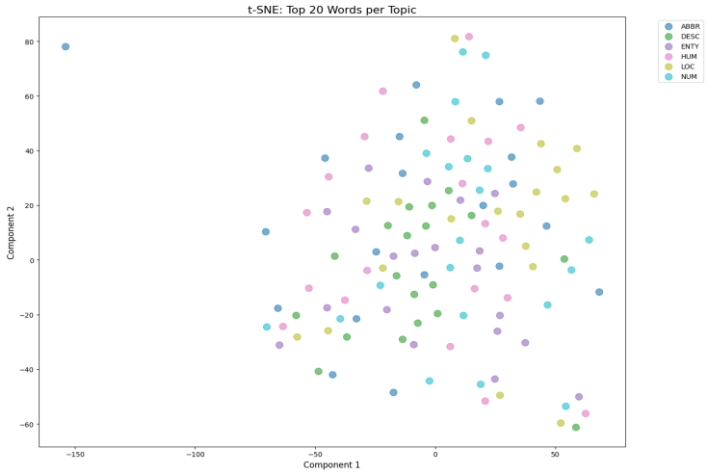
```
Total words for visualization: 120
```

We first filter out the common stopwords and ensure that only words existing in the GloVe vocab are considered. For every training example, it counts the frequency of each remaining word under its respective topic. After accumulating all the counts, we select the top20 most frequent words per topic.

For each top word, we retrieve the glove embedding vector and store them along with the labels.

We use t-SNE (perplexity=30, random_state=42) on the GloVe vectors of the top-20 most frequent non-stop-words per class. t-SNE preserves local neighborhoods, which is appropriate for visualizing topic clusters in sparse, short questions.

(e) Plot the points in a scatter plot, color-coded by their topic category. Attach your plot here. Analyze your findings.



The words from 6 topics are spread across the space with some overlapping clusters. This suggests that semantic content shared by the most frequent words is greater than the unique content that differentiates the topics. Some topic clusters like LOC appear more tightly grouped, indicating that location related words share stronger semantic similarity within the Glove embedding space. In contrast, topics like DESC and ENTY are more dispersed, implying a broader range of meanings. In summary, this visualization indicates partial clustering by topic, but substantial overlap across categories, reflecting that GloVe embeddings encode general semantics rather than distinct topic boundaries.

Part 2. Model Training & Evaluation - RNN

Following the requirements, we built a recurrent neural network (RNN) classifier on top of the pretrained GloVe embeddings and TREC question classification dataset.

Input Representation

- Tokenization and vocabulary follows the same setup as Part 1.
- Pretrained GloVe vectors (300d) are used to initialize the embedding matrix.
- For OOV words, we apply the mitigation strategy from Part 1:
    - All unseen tokens are mapped to a shared <unk> embedding, initialized as the average of pretrained embeddings of rare in-vocabulary words, and updated during training.
- All embeddings (including <unk> and <pad>) are set to be **trainable**, allowing task-specific adaptation.

RNN architecture (base model)

- Embedding dimension: **300**
- Recurrent encoder: **2-layer bidirectional RNN**
- Hidden size: **256**
- Classifier: fully connected layer from the aggregated sentence vector to **6-way softmax.**
- Loss: cross-entropy.

Data split & evaluation

- Training set split into **train / validation = 0.8 / 0.2**.
- **Accuracy** is used for validation and test, as required.
- Early stopping based on validation accuracy to prevent overfitting.

Sequence packing

TorchText yields text as [seq_len, batch]. We keep batch_first=False and use pack_padded_sequence(... , enforce_sorted=False) so the RNN ignores pads and learns from true token length only.

Sentence representation (aggregation)
We ablate three readouts: last hidden state (captures tail-biased context), mean pooling (smooths over the full span), and max pooling (selects strongest features). For short questions, max often surfaces decisive n-grams; last can underperform if crucial tokens appear early. We report test accuracy for all three and select the best for downstream comparisons.

Regularization plan
We test (i) none, (ii) dropout 0.5, (iii) L2 (1e-4), (iv) dropout+L2. Dropout combats co-adaptation in hidden units; L2 dampens over-confident weights; the combo is our default if it yields the highest validation accuracy.

Optimization & schedule
Adam (lr=1e-3) converges reliably on small text tasks; we cap at 20 epochs with early stopping patience=3 on validation accuracy. We reload the best checkpoint for test evaluation to avoid reporting post-overfit performance.

(a) Report the final configuration of your best model, namely the number of training epochs, learning rate, optimizer, batch size and hidden dimension.
In the 4 strategies (No Regularization, Dropout, L2 Regularization and Dropout + L2 Regularization) we employed, the best-performing RNN configuration is Dropout + L2 Regularization:

Parameters	Value
Hidden Dimension	256
Hidden Layers	2
Patience	5
Batch Size	64
Optimizer	Adam
Learning Rate	0.001
Dropout	0.5
L2 Weight Decay	1e-4
Epoch	20
Early Stopping	10

This configuration (RNN + Dropout + L2) achieves our best test accuracy of 0.9153.

(b) Report all the regularization strategies you have tried. Compare the accuracy on the test set among all strategies and the one without any regularization.
To improve the generalization ability of the RNN classifier and prevent overfitting we employed three different regularisation techniques: Dropout, L2 Regularization and Dropout + L2.

Each of the models were trained under identical configurations except for the regularisation technique. We then compared the accuracy on the test set among all the techniques against the baseline model without any regularisation.

Comparison Summary

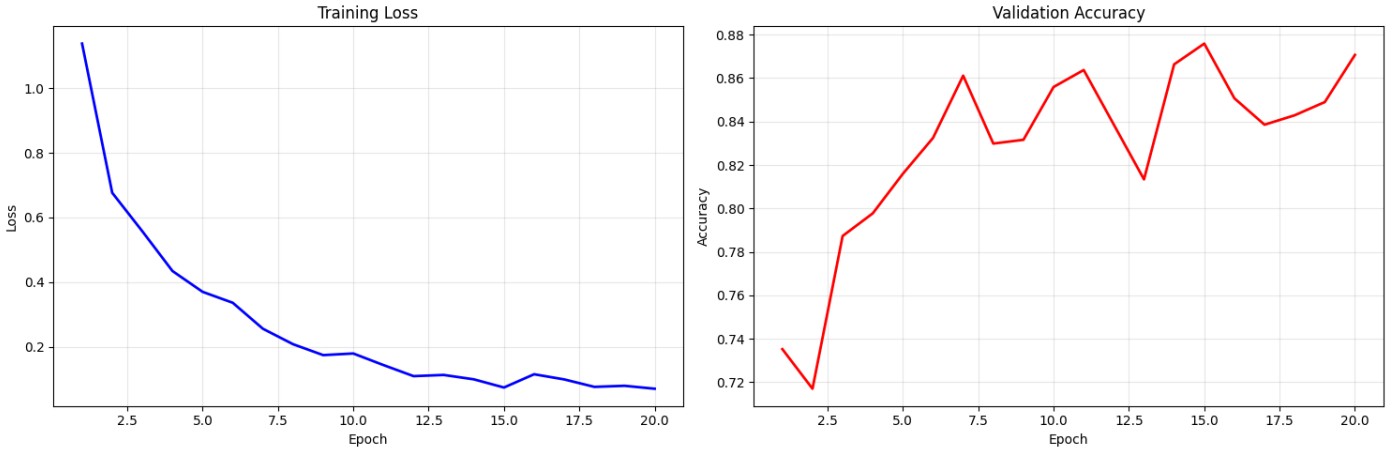
Regularization Strategy	Description	Test Accuracy
No Regularization	Baseline model without any regularization	0.8899
Dropout (0.5)	Randomly disables neurons to reduce overfitting	0.8948

L2 Regularization	Penalizes large weight values (weight decay = 1e-4)	0.8992
Dropout + L2	Combines dropout and L2 weight decay for stronger regularization	0.9153

(c) For the best configuration and regularization strategy in your experiments, plot the training loss curve and validation accuracy curve during training with x-axis being the number of training epochs. Discuss what the curves inform about the training dynamics.

Key observations:

- **Monotonic loss decrease:** Training loss decreases over the epochs, indicating stable optimization.
- **Validation accuracy rises then decreases**: Validation accuracy increases rapidly in early epochs and stabilizes around the 8–10 epoch range. After 10 epochs, gains are marginal and small fluctuations appear, suggesting the model has essentially converged. Continuing training risks overfitting.



(d) RNNs produce a hidden vector for each word, instead of the entire sentence. Which methods have you tried in deriving the final sentence representation to perform sentiment classification? Describe all the strategies you have implemented, together with their accuracy scores on the test set.

Summary of Results

Aggregation Strategy	Description	Test Accuracy
Last Hidden State	Concatenates last forward and backward hidden states.	0.9142
Mean Pooling	Averages all hidden states across the sequence.	0.8875
Max Pooling	Takes element-wise maximum across all hidden states	0.9114

(e) Report topic-wise accuracy (accuracy for each topic) on the test set for the best model you have. Discuss what may cause the difference in accuracies across different topic categories.

Topic	Topic-Wise Accuracy
ABBR	0.7778
ENTY	0.8298
HUM	0.8519
NUM	0.8850
LOC	0.9077

DESC	0.9710

Analysis of differences:

- **DESC, LOC, NUM** - These categories often contain distinctive lexical and syntactic patterns, which are well captured by pretrained embeddings plus the RNN encoder.
- **HUM, ENTY** - more semantically diverse, leading to overlaps with other categories
- **ABBR** - Very few training samples, which might result in strong class imbalance and notably weaker accuracy.

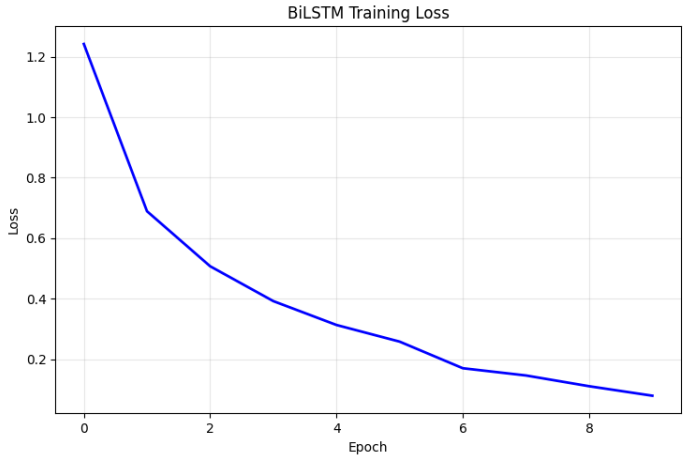
Part 3. Enhancement

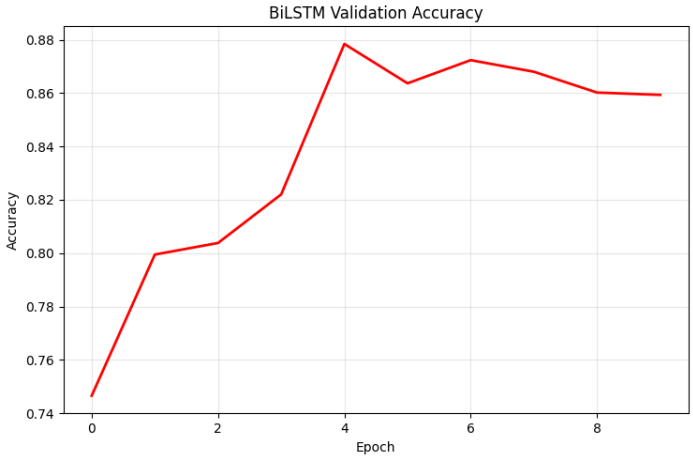
(a) Plot the training loss curve and validation accuracy curve of biLSTM and biGRU. Report their accuracies on the test set

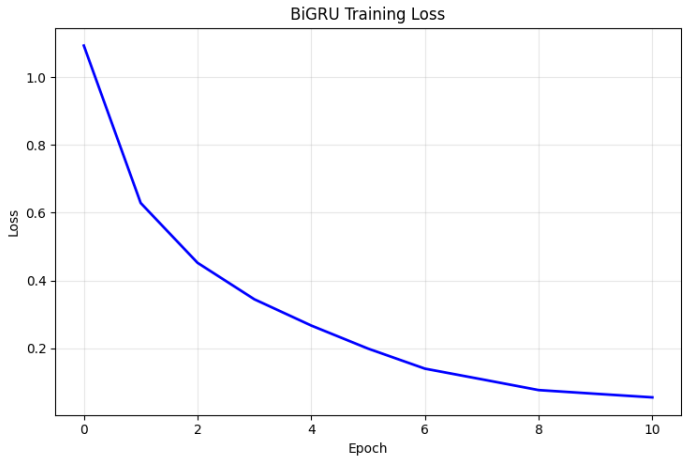
biLSTM (2 layers, hidden=256 per direction) models longe-range dependencies and benefits from both left and right context in short queries (e.g. “What river runs through X?”). Two layers balances the capacity vs overfit on TREC scale, and we keep dropout on the recurrent stack when n_layers > 1.

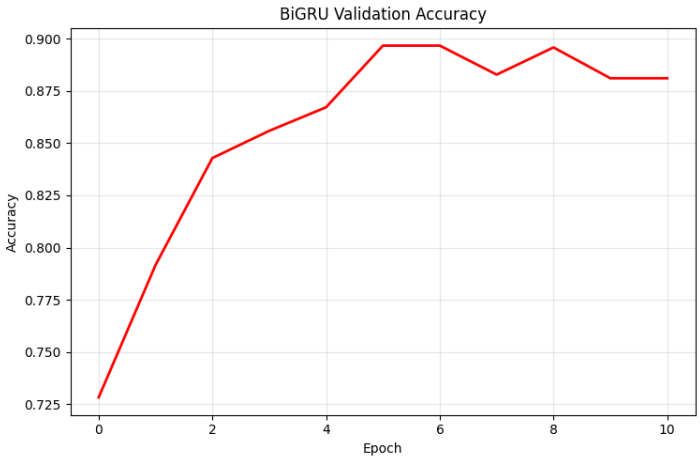
The biGRU model uses fewer gates than biLSTM, reducing parameters and training time while retaining gating benefits. On short sequences, biGRU often matches or exceed biLSTM due to simpler dynamics.

Both bidirectional models outperformed the simple RNN. BiGRU has the best overall performance, suggesting GRU’s simple gating works well for this dataset size and question length distribution.



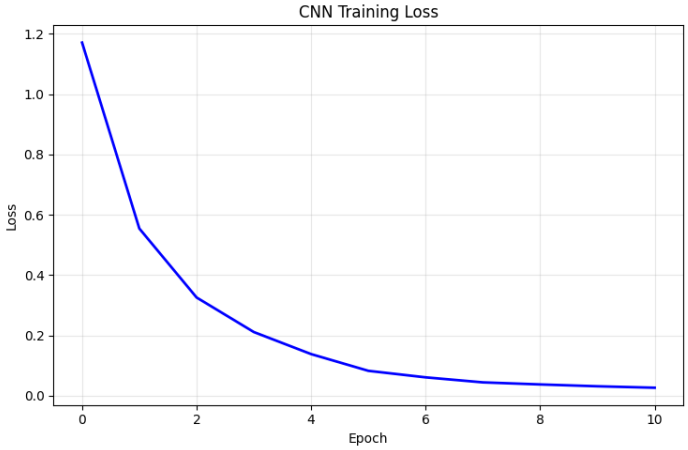


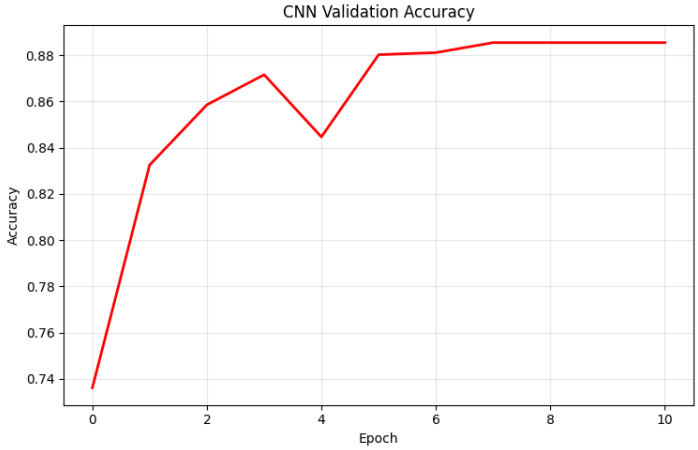




Model	Validation Accuracy	Test Accuracy
BiLSTM	0.878	0.8894
BiGRU	0.881	0.9055

(b) Plot the training loss curve and validation accuracy curve of CNN. Report its accuracy score on the test set
Intuitively, we observe that local phrase patterns (captured by CNN) are informative for TREC questions, while long-range dependencies are relatively limited, so both CNN and BiGRU perform strongly.

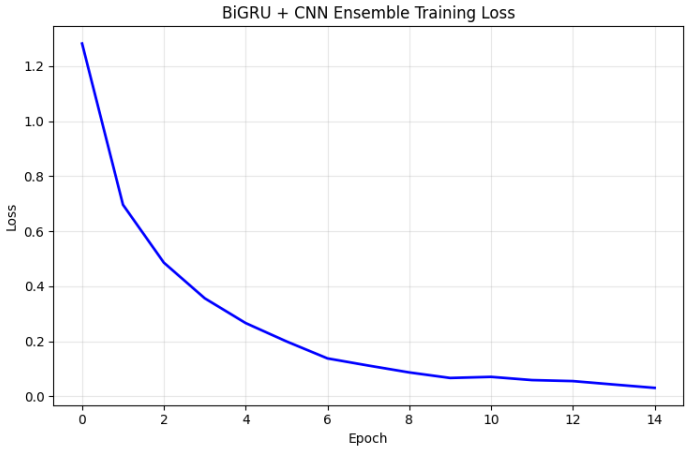


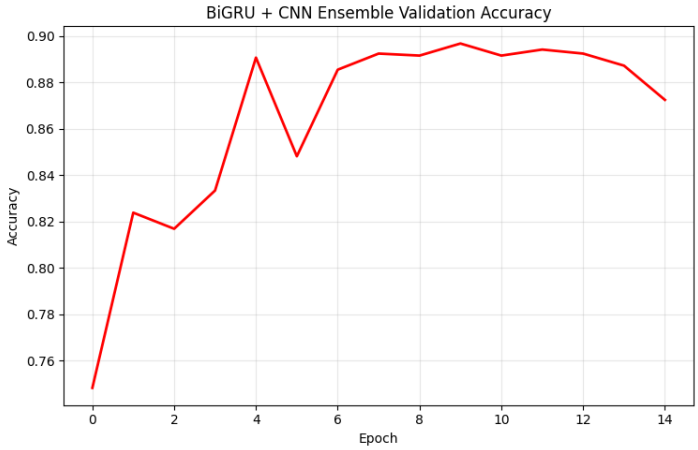


Model	Validation Accuracy	Test Accuracy
CNN	0.889	0.9050

(c) Describe your final improvement strategy in Part 3.3. Plot the corresponding training loss curve and validation accuracy curve. Report the accuracy on the test set.

We combined BiGRU with CNN ensemble to leverage the strengths of both sequential and local feature extraction. The BiGRU encodes long-range dependencies and overall structure, while the CNN focuses on extracting the most important local features from the BiGRU's contextual output. The model was trained using Adam optimizer, cross entropy loss, weight decay and early stopping based on validation accuracy to prevent overfitting. A patience counter halts training when validation performance stops improving.





Model	Validation Accuracy	Test Accuracy
BiLSTM	0.878	0.8894
BiGRU	0.881	0.9055
CNN	0.889	0.9050
Ensemble (BiGRU + CNN)	0.897	0.9114

(d) Describe your strategy for improvement of weak topics in Part 3.4. Report the topic-wise accuracy on the test set after applying the strategy and compare with the results in Part 2(e).

In the baseline RNN model from part 2(e), certain topics such as *ABBR* (abbreviations) and *ENTY* (entity) exhibited noticeably lower accuracy compared to others. This performance gap was primarily due to class imbalance where these topics had significantly fewer training samples, resulting in weaker gradient signals during training.

To address this issue, a class-weighted loss function was implemented. Specifically, the CrossEntropyLoss function was modified to include class weights inversely proportional to class frequency in the training set. This means that misclassifications in minority classes were penalized more heavily, thereby encouraging the model to learn more balanced decision boundaries.

$$w_c = \frac{N}{K \cdot n_c}$$

Where W_c is the weight for class c, N is the total number of samples, K is the number of classes, and n_c is the number of samples in class c.

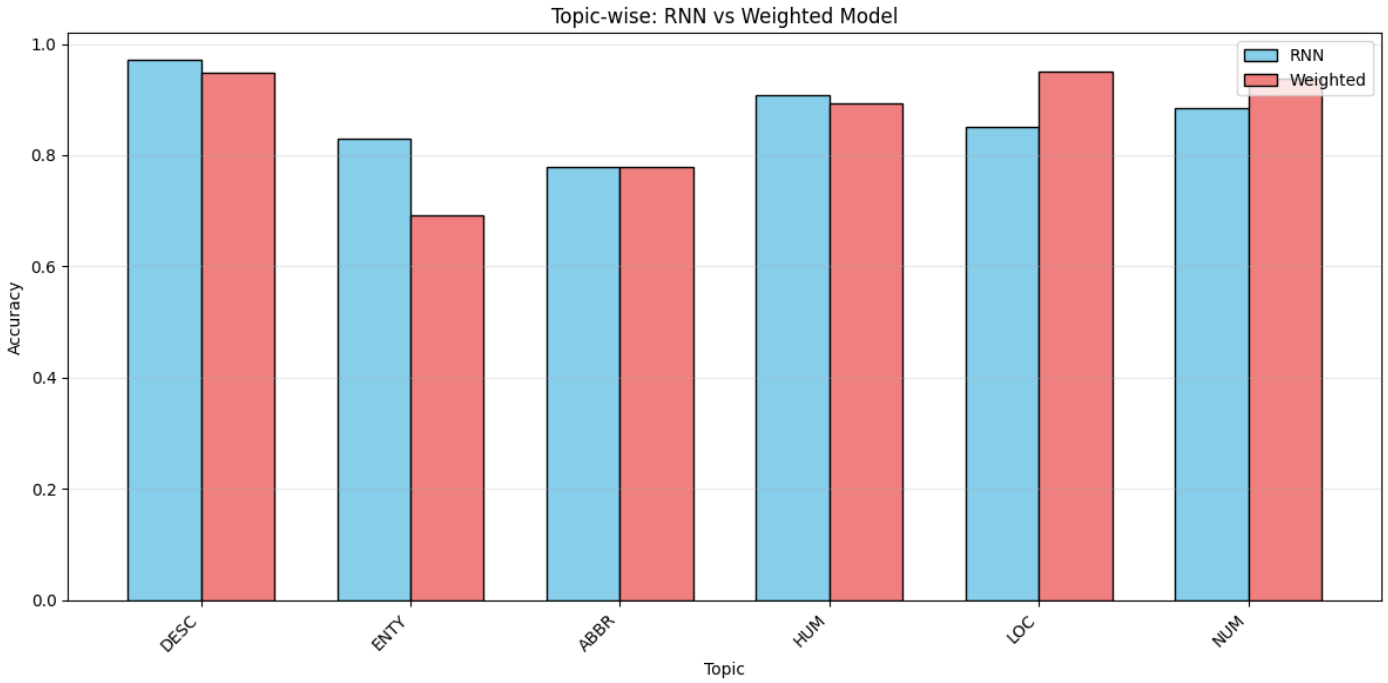
The improvement strategy was applied using a **BiLSTM** model trained with:

- Weighted cross-entropy loss
- Dropout = 0.5
- Weight decay = 1e-4
- Learning rate = 0.001

This model was compared against the baseline RNN from Part 2(e) to assess topic-level improvements.

Topic-wise comparison

Topic	RNN (Accuracy)	Weighted	Improvement
DESC	0.9710	0.9493	- 0.0217
ENTY	0.8298	0.6915	- 0.1383
ABBR	0.7778	0.7778	+ 0.0000
HUM	0.9077	0.8923	- 0.0154
LOC	0.8519	0.9506	+ 0.0988
NUM	0.8850	0.9381	+ 0.0531


Topic-wise: RNN vs Weighted Model
RNN
Weighted
Accuracy
1.0
0.8
0.6
0.4
0.2
0.0
DESC
ENTY
ABBR
HUM
LOC
NUM
Topic