Exercises on Natural Language Processing with Recurrent Neural Networks (RNN) № 3

Kagioglou Maria, Panourgia Evangelia 26/02/2025

Assignment 9

We utilized the `DBpedia 14` dataset, available on Hugging Face, which comprises **560,000 Wikipedia articles** categorized into **14 distinct classes**. Each entry in the dataset includes the **title** of a Wikipedia article along with its **text content**. The classification categories encompass a diverse range of entities, including **Company, Educational Institution, Artist, Athlete, Office Holder**, and others.

Due to the large size of the initial dataset, which contained **560,000 instances** in the training set and **70,000 instances** in the test set, we encountered **memory constraints**. To address this issue, we **downsampled** the dataset to optimize memory usage. As a result, the final dataset sizes after downsampling were **140,000 instances** for the training set and **17,500 instances** for the test set.

After downsampling, we further split the training dataset to create a development set, using a **0.2** proportion of the training data. Since the test set was already predefined, no further modifications were needed. The final dataset sizes were as follows:

- **Training set size:** 112,000
- **Development set size:** 28,000
- **Test set size:** 17,500

The Table 1 depicts the dataset distribution across the train, test and validations sets.

	1	**4**	**12**	**6**	**7**	**0**	**13**	**5**	**11**	**3**	**9**	**8**	**10**	**2**
Train	8009	7959	8046	7992	7951	8026	8020	8018	8017	7930	7975	8046	8040	7971
Validation	1991	2041	1954	2008	2049	1974	1980	1982	1983	2070	2025	1954	1960	2029
Test	1250	1250	1250	1250	1250	1250	1250	1250	1250	1250	1250	1250	1250	1250

Table 1: Dataset Distribution Across Train, Validation, and Test Sets

Some basic statistics related to our dataset are the following ones : Before prepossessing:

- **Average Document Length (in words):** 46.104
- **Average Document Length (in characters):** 281.194

After prepossessing:

- **Average Document Length (in words:** 29.645

- **Average Document Length (in characters):** 217.337

In addition, we calculated the `Flesch-Kincaid Readability Score` evaluates the complexity of a text in terms of readability:

- **Higher score** = More difficult to read.
- **Lower score** = Easier to read.

The dataset statistics are as follows:

- **Number of training examples:** 112,000
- **Number of dev examples:** 28,000
- **Number of test examples:** 17,500
- Length of training set after preprocessing:
  - **Min length:** 1
  - **Max length:** 1,162
  - **Mean length:** 29
  - **Variance:** 231
  - **90th percentile of text length:** 48
  - **Coefficient of Variation:** 0.51

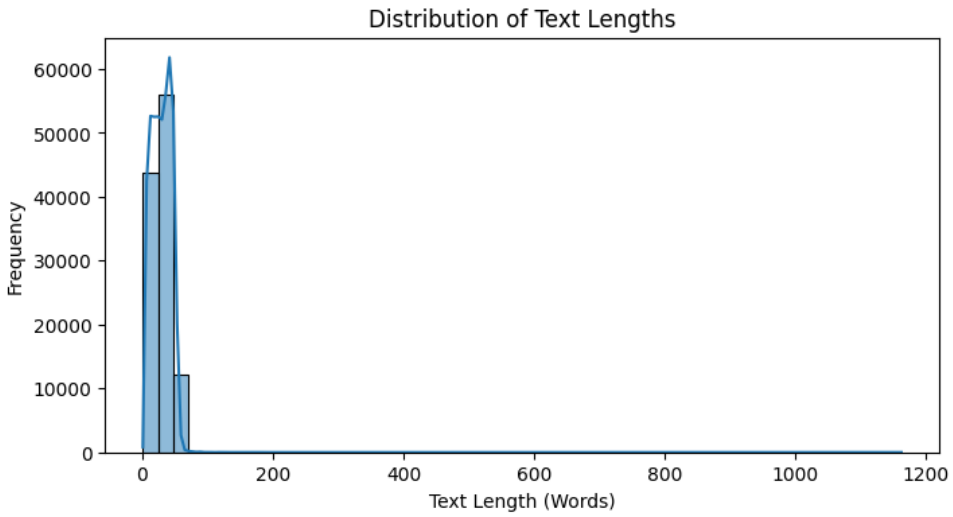

Figure 1: Dataset Statistics

Regarding nlp pre-process, we implemented `preprocess(data)`, is designed to preprocess textual data by tokenizing sentences, filtering stopwords, and returning a cleaned version of the input text.

Below is an example of the preprocessing:

Initial Text

Municipal Credit Union (MCU) is a state chartered credit union headquartered in New York City regulated under the authority of the National Credit Union Administration (NCUA). MCU is metro New York's largest credit union. As of 2008 MCU had $1.3 billion in assets, approximately 301,000 members, and 13 branches.

Processed Text

```
['municipal', 'credit', 'union', 'mcu', 'state', 'chartered', 'credit',
'union', 'headquartered', 'new', 'york', 'city', 'regulated', 'authority',
'national', 'credit', 'union', 'administration', 'ncua', 'mcu', 'metro',
'new', 'york', "'s", 'largest', 'credit', 'union', '2008', 'mcu',
'1.3', 'billion', 'assets', 'approximately', '301000', 'members', '13',
'branches']
```

RNN Model

In order to bulid the RNN model, we constructed the embedding matrix using pre-trained Word2Vec embeddings from the Google News dataset, loaded via:

```
word2vec = api.load('word2vec-google-news-300')
```

To **handle unknown words, we assigned them an index of 1 and used padding to standardize sentence lengths.** The process for creating the custom vocabulary and embedding matrix is outlined below:

- **Custom Vocabulary:** A vocabulary was generated from the training dataset using `CountVectorizer`, with a maximum of 10,000 features.
- **Embedding Initialization:** The Word2Vec embeddings were mapped to the vocabulary, with special tokens for padding (`PAD`) and unknown words (`UNK`). The average of all embeddings was used for unknown words.
- **Final Embedding Matrix:** This embedding matrix was then used as input to the RNN model.

The next step involved tokenizing the text and defining a function to handle padding dynamically. Instead of padding all sentences to a fixed maximum length (e.g., 1040), we applied **batch-wise dynamic padding.** This approach ensures that:

- Shorter sentences are padded only up to the length of the longest sentence in each batch.
- Unnecessary padding is avoided, leading to reduced memory usage.
- Computational efficiency is improved.

To achieve this, we implemented a **TextDataset class** for **tokenization** and a custom **collate_batch function** for **dynamic padding within the DataLoader**. The dataset class tokenizes text by mapping words to indices from a predefined vocabulary while handling unknown words with an UNK token. The collate_batch function processes each batch by:

- Extracting tokenized sequences and labels:

```
texts, labels = zip(*batch)  # Extract the text and labels from the batch
```

- Converting sequences into PyTorch tensors:

```
texts = [t.clone().detach() for t in texts]  # Convert sequences into PyTorch tensors
```

- Applying dynamic padding using `pad_sequence()`, ensuring all sequences in a batch are padded to match the longest one:

```
texts = torch.nn.utils.rnn.pad_sequence(
    texts, batch_first=True, padding_value=vocab.get('PAD', 0)
)  # Apply padding
```

- Converting labels into a single tensor:

```
labels = torch.tensor(labels, dtype=torch.long)  # Ensure labels are long tensors
return texts, labels
```

Proceeding further, we define a custom RNN model that incorporates an attention mechanism. This is done using two main classes: `CustomAttention` and `RNNModel`. Below, we describe the key features of the `CustomAttention` class and how it integrates with the `RNNModel`:

1. `CustomAttention` Class

The `CustomAttention` class defines a flexible attention mechanism that can apply two types of attention. The choice between the simpler or more complex attention mechanism can be controlled by the `use_second_attention` flag.

- **Attention Mechanism**: The class calculates attention scores to evaluate the importance of each token in the sequence. It uses an additive mechanism based on the learned parameters, where the attention scores are computed using a transformation of the RNN output.

- **Flexible Attention Layer**: The class offers two attention mechanisms:
  - `attention_layer_1`: A simple Multi-Layer Perceptron (MLP)-based attention mechanism that transforms the RNN output to compute attention scores.
  - `attention_layer_2`: A more complex hierarchical structure that introduces additional layers for more powerful attention mechanisms. This second layer provides an alternative strategy to compute attention scores, which can be toggled using the `use_second_attention` flag.
- **Padding Masking**: The attention mechanism ensures that padding tokens (e.g., tokens filled in to match the longest sequence) do not affect the attention scores. It does this by applying a padding mask, which assigns very low attention weights to padded tokens.
- **Output after Attention**: Once the attention mechanism computes the attention weights, it performs a weighted sum of the RNN hidden states, resulting in a context vector. This context vector is used to compute the final output.

2. Attention Formulae

The `CustomAttention` class computes the attention scores and context vector using the following steps:

Scoring Type	Formula
Unnormalized Attention Scores	$e_t = w^T \text{ReLU}(Wh_t)$
Normalized Attention Weights	$a_t = \text{softmax}(e_t)$
Context Vector (Weighted Sum)	$c = \sum_{t=1}^{T} a_t h_t$

Where:

- e_t: The unnormalized attention score, computed by applying a linear transformation to the hidden state.
- a_t: The normalized attention weight, obtained by applying the softmax function to the attention scores.
- c: The context vector, which is the weighted sum of the RNN hidden states, where each hidden state is weighted by the corresponding attention weight.

3. `RNNModel` Class

The `RNNModel` class defines the architecture of the RNN-based model, which can use either RNN, GRU, or LSTM cells. The model includes the following components:

- **Embedding Layer**: A word embedding layer that can either use pre-trained embeddings (such as Word2Vec) or random initialization. It also handles padding for sequences shorter than the longest sequence in the batch.

- **RNN Layer**: The main RNN layer, which can be an RNN, GRU, or LSTM. It processes the input sequence and outputs hidden states at each timestep.
- **Attention Integration**: If the `use_attention` flag is set, the output of the RNN layer is passed through the `CustomAttention` module to compute a context vector based on attention scores. This context vector is then passed to the fully connected output layer.
- **Pooling Option**: If `use_pooling` is set, pooling is applied to the output of the RNN (typically max pooling). Otherwise, attention is used to generate a weighted context vector.
- **Final Output Layer**: A fully connected layer that maps the RNN output (or attention output) to the final prediction.

Evaluate Models

The next step was to try different models in order to evaluate their perfornamce:

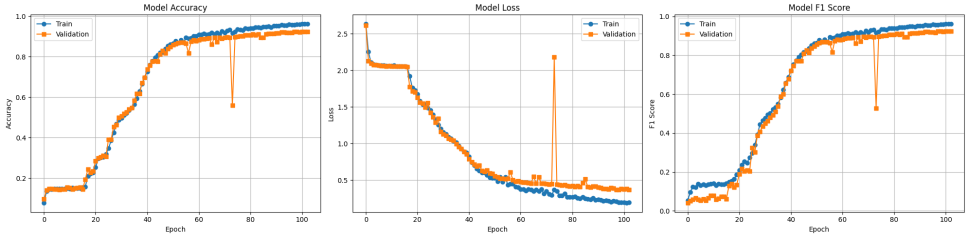

Figure 2: A simple RNN model with hidden_dim = 256, epochs = 180, batch_size = 32, $l_r = 10^{-5}$, and num_layers = 1.

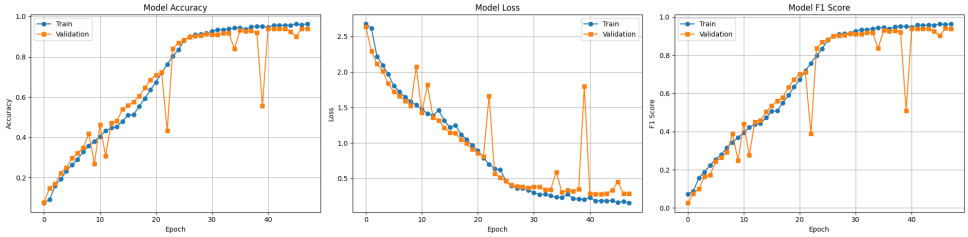

Figure 3: RNN model with hidden_dim = 256, epochs = 180, batch_size = 32, $l_r = 10^{-5}$,num_layers = 1, bidirectional=True and use_layer_norm=True.

Poor Behavior of the RNN Model

- **Weight Sharing & Gradient Issues:**
  - RNNs apply the same weight matrices at each time step. During backpropagation, **gradients can either vanish (become extremely small) or explode (become excessively large)**:
    - This makes it difficult for the model to learn long-range dependencies, leading to unstable training and poor generalization.

        * RNNs struggle with sequences that require remembering information from far back in the input.

- **Weight Update Sensitivity:**
  - The weight update process in RNNs is highly sensitive to initialization and learning rate:
    * Poor initialization or an unsuitable learning rate can cause oscillating or diverging loss, making training unstable.

Oscillation Issue in Bidirectional RNNs

As shown in Figures 2 and 3, the oscillation problem worsens due to:

- **Increased Parameter Count:**
  - A Bidirectional RNN **doubles the number of parameters**, as it maintains two hidden states per timestep (one forward and one backward).
  - This makes training more difficult, especially if the dataset is small or the learning rate is too high.
- **Exacerbation of Vanishing/Exploding Gradients:**
  - Since standard RNNs already suffer from vanishing/exploding gradients, **doubling the hidden states amplifies the issue**.
- **Layer Normalization Instability:**
  - **LayerNorm normalizes activations per sample**, dynamically adjusting hidden state distributions at every timestep.
  - This can create unexpected interactions with RNN activations, leading to further instability during training.

GRU & LSTM: Solutions to RNN Problems

GRU and LSTM architectures mitigate the above **RNN limitations** by incorporating **gating mechanisms**:

- They use **forget, input, and output gates** (LSTM) or **reset and update gates** (GRU) to regulate gradient flow.
- These mechanisms prevent **vanishing/exploding gradients**, allowing for more stable training and better long-term memory retention.
- They improve weight updates, smoothing out learning and enabling stable convergence.

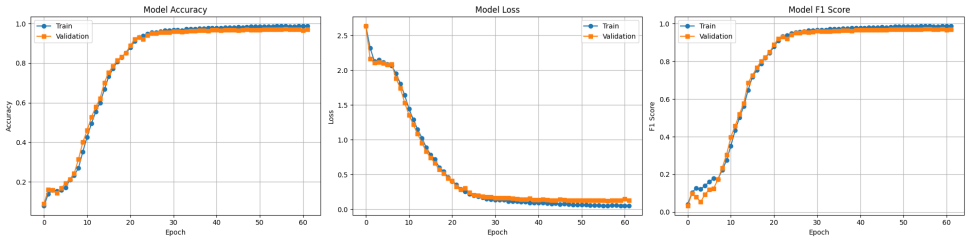

Figure 4: A simple LSTM model with hidden_dim = 256, epochs = 180, batch_size = 32, $l_r = 10^{-5}$, and num_layers = 1.

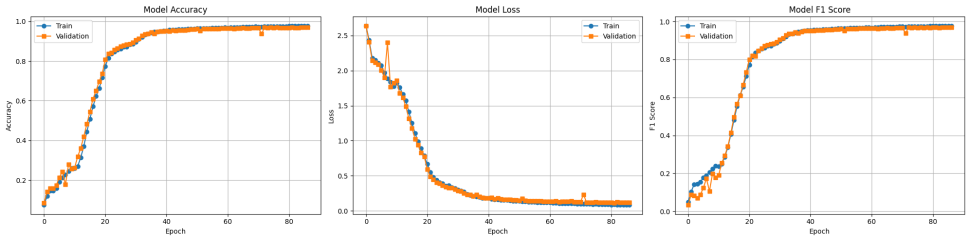

Figure 5: LSTM model with hidden_dim = 256, epochs = 180, batch_size = 32, $l_r = 10^{-5}$, num_layers = 1, dropout_rnn=02̇, freeze=True.

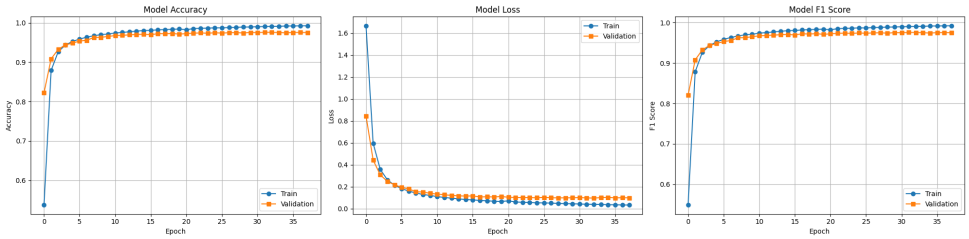

Figure 6: LSTM model with hidden_dim = 256, epochs = 180, batch_size = 32, $l_r = 10^{-5}$, num_layers = 1, use_attention=True(simple architecture), dropout_attention=0.1, attention_dim=256.

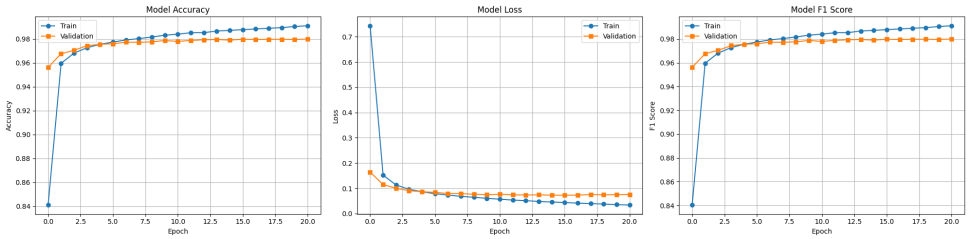

Figure 7: LSTM model with hidden_dim = 256, epochs = 180, batch_size = 32, $l_r = 10^{-5}$, num_layers = 2, use_attention=True(simple architecture), dropout_attention=0.1, attention_dim=256, use_layernorm=True, dropout_rnn=0.3.

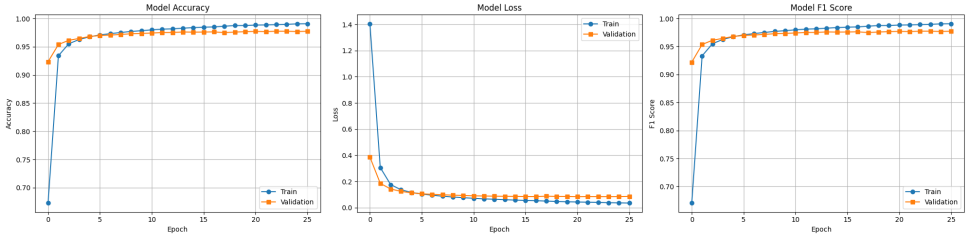

Figure 8: LSTM model with hidden_dim = 128, epochs = 180, batch_size = 32, $l_r = 10^{-5}$, num_layers = 2, use_attention=True(simple architecture), dropout_attention=0.1, attention_dim=256, use_layernorm=True, dropout_rnn=0.5.

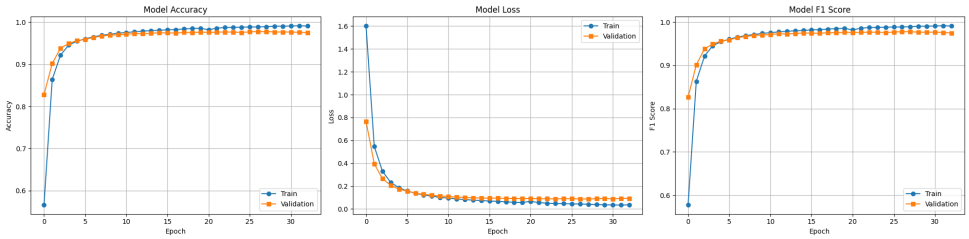

Figure 9: LSTM model with hidden_dim = 256, epochs = 180, batch_size = 32, $l_r = 10^{-5}$, num_layers = 1, use_attention=True(simple architecture), dropout_attention=0.1, attention_dim=256, bidirectional=True.

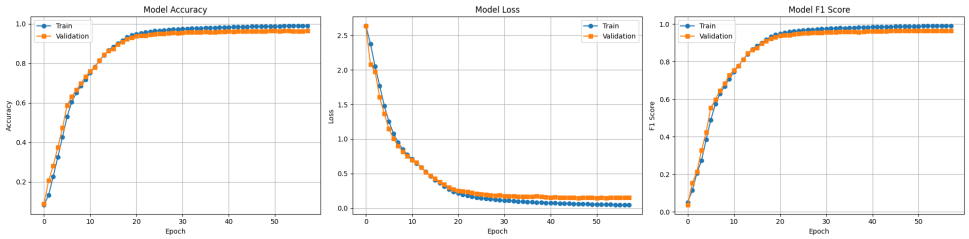

Figure 10: GRU model with hidden_dim = 256, epochs = 180, batch_size = 32, $l_r = 10^{-5}$, num_layers = 1.

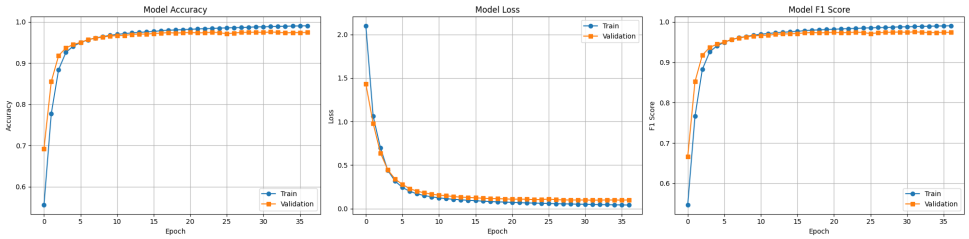

Figure 11: GRU model with hidden_dim = 256, epochs = 180, batch_size = 32, $l_r = 10^{-5}$, num_layers = 1, use_pooling=True.

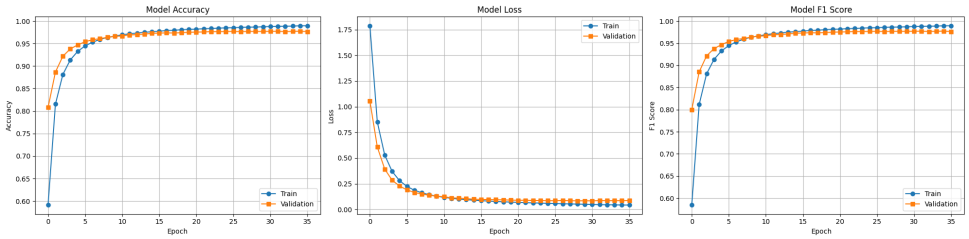

Figure 12: GRU model with hidden_dim = 256, epochs = 180, batch_size = 32, $l_r = 10^{-5}$, num_layers = 1, use_attention=True, dropout_rate_at=0.5, use_second_attention=True(complex architecture).

The models that achieve the best results (in terms of training and validation losses, accuracy, and F1-score) are shown in Figures 8 9 11 and 12 with the model shown in Figure 8 displaying the best results:

Test Set

```
=== LSTM-Model-9 (128H, 2L, SA-Simple, LN, Dropout=0.5) Test Classification Report ===
              precision    recall  f1-score   support

           0     0.9368    0.9488    0.9428      1250
           1     0.9800    0.9776    0.9788      1250
           2     0.9504    0.9648    0.9575      1250
           3     0.9896    0.9888    0.9892      1250
           4     0.9761    0.9800    0.9780      1250
```

```
              5     0.9762    0.9856    0.9809      1250
              6     0.9740    0.9608    0.9674      1250
              7     0.9911    0.9848    0.9880      1250
              8     0.9952    0.9968    0.9960      1250
              9     0.9762    0.9864    0.9813      1250
             10     0.9839    0.9752    0.9795      1250
             11     0.9943    0.9776    0.9859      1250
             12     0.9840    0.9824    0.9832      1250
             13     0.9719    0.9688    0.9704      1250

    accuracy                           0.9770     17500
   macro avg     0.9771    0.9770    0.9771     17500
weighted avg     0.9771    0.9770    0.9771     17500

Class 0 Precision-Recall AUC: 0.9834
Class 1 Precision-Recall AUC: 0.9976
Class 2 Precision-Recall AUC: 0.9878
Class 3 Precision-Recall AUC: 0.9977
Class 4 Precision-Recall AUC: 0.9947
Class 5 Precision-Recall AUC: 0.9968
Class 6 Precision-Recall AUC: 0.9933
Class 7 Precision-Recall AUC: 0.9987
Class 8 Precision-Recall AUC: 0.9993
Class 9 Precision-Recall AUC: 0.9984
Class 10 Precision-Recall AUC: 0.9953
Class 11 Precision-Recall AUC: 0.9977
Class 12 Precision-Recall AUC: 0.9979
Class 13 Precision-Recall AUC: 0.9916
Macro Precision-Recall AUC: 0.9938
```

Baseline Models

MLP-Logistic Regressiion-Majority & Random Classifier

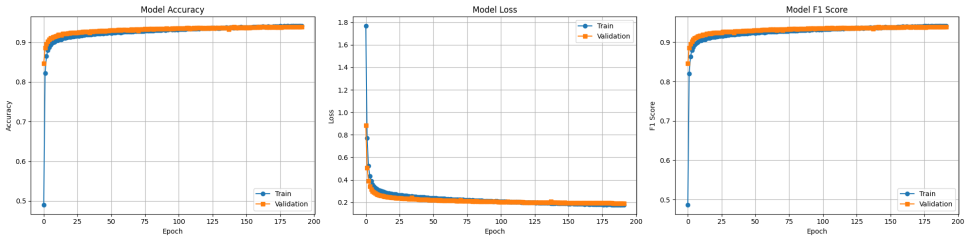

Figure 13: MLP, word2vec representation, hidden_layers:[512], dropout=0.5, batch_norm=True, layer_norm=True.

Below, we can see the macro metrics for the MLP, Logistic Regression, Majority & Random Classifier:

Metric	RNN	MLP	Logistic Regression	Majority	Random
Macro Precision	0.9770	0.9379	0.9373	0.93	0.0731
Macro Recall	0.9770	0.9378	0.9373	0.07	0.0731
Macro F1-score	0.9771	0.9378	0.9372	0.01	0.0731
Macro AUC	0.9938	0.9787	0.9782	-	0.5357

Table 2: Macro Metrics for the Test Set (RNN, MLP, Logistic Regression, Majority Classifier, and Random Classifier)

In the following plots (Figure 14), we can observe whether the models correctly predict the class.

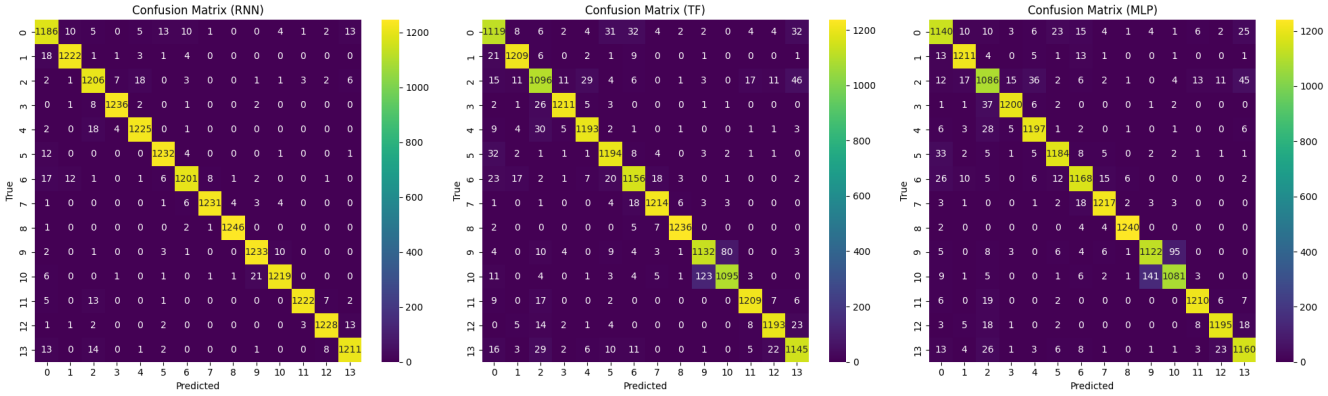

Figure 14: Confusion Matric for RNN, MLP and Logistic Regression model.

As we can see from the above results, the RNN model outperforms the baseline models, achieving better metric results and lower training and validation losses.

Assignment 2

RNN (POS) Tagger

The preprocessing stage for POS tagging involves multiple steps, including data acquisition, analysis, vocabulary creation, embedding initialization, and dataset preparation. We retrieved CoNLL-U formatted data from the Universal Dependencies dataset being available at: `https://github.com/UniversalDependencies/UD_English-EWT` (provides three '$.conllu$' files: `train`, `dev`, and `test`), where we extract sentences and their corresponding POS tags and store them in a structured format for further processing. We analyzed the distribution of POS tags and sentence lengths to gain insights into the dataset, helping us understand the frequency of different POS categories and determining an appropriate sentence length threshold for model training. Sentences longer than the 95th percentile are truncated to ensure uniform processing. We created a vocabulary for words and POS tags, assigning unique indices to each, while pre-trained FastText embeddings are used to initialize word representations, ensuring robust contextual understanding. We then converted sentences and POS tags into numerical representations using the constructed vocabularies and created a dataset class to efficiently handle data loading and batching. This structured preprocessing ensures that raw text data is transformed into a suitable format for training a POS tagging leveraging the Pytorch Python library.

Our dataset consists of 12,544 training sentences, 2,001 validation sentences, and 2,077 test sentences. Additionally, Figure 15 illustrates the distribution of POS tags, highlighting the imbalance across different categories. The figure also presents the sentence length distribution, demonstrating that the majority of sentences falling within the 95th percentile have an average length of 39 tokens.

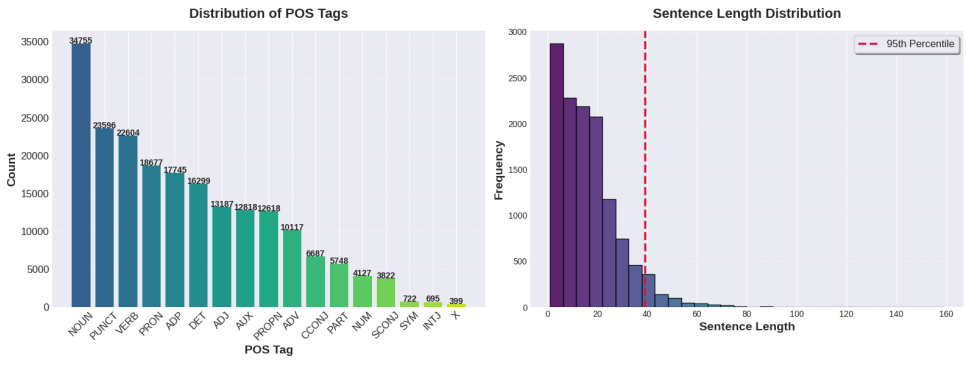

Figure 15: Setence and POS Distribution

We also implemented an RNN-based POS tagging model that supports multiple functionalities. Our model incorporates word embeddings, allowing the use of pre-trained embeddings while providing the flexibility to freeze or fine-tune them. The architecture supports different recurrent units, including RNN, GRU, and LSTM, with options for bidirectionality and multiple stacked layers. Dropout regularization is applied at various stages, including embeddings, re-

current layers, and fully connected layers, to prevent overfitting. Additionally, an optional attention mechanism is integrated to enhance sequence representation, with dropout applied to the attention output for further regularization. The final classification layer maps the processed output to POS tag indices, ensuring effective sequence labeling.

We optimized the model's performance through a systematic manually hyperparameter tuning process based on train and dev sets. The training process involves minimizing the loss function, observing the gap of train and val loss in order to avoid over fit while monitoring the F1-macro score to assess model performance. We utilized early stopping to prevent overfitting by halting training when validation loss ceases to improve for a specified number of epochs. The optimizer is adjusted to ensure efficient gradient updates, and dropout is applied at multiple levels to regularize the model. The training and validation losses, along with F1-macro scores, are plotted to analyze the model's learning progress, helping us identify the optimal configuration for achieving the best POS tagging results.

We evaluated different configurations to optimize model performance. The following combinations were tested:

- **Model 1:** Hidden dimension of 128, single-layer bidirectional LSTM, attention enabled, dropout rate of 0.3, it is presented the loss and Macro F1 score in Figure]16. Early stopping was triggered in epoch 36.
- **Model 2:** Hidden dimension of 128, two-layer bidirectional LSTM, attention enabled, dropout rate increased to 0.4 to mitigate overfitting, it is presented the loss and Macro F1 score in Figure]17.Early stopping was triggered in epoch 29.
- **Model 3:** Hidden dimension increased to 256, two-layer bidirectional LSTM, attention enabled, dropout rate maintained at 0.4, it is presented the loss and Macro F1 score in Figure]18. Early stopping was triggered in epoch 20.

Each configuration was manually analyzed to determine the best balance between model complexity and generalization ability.

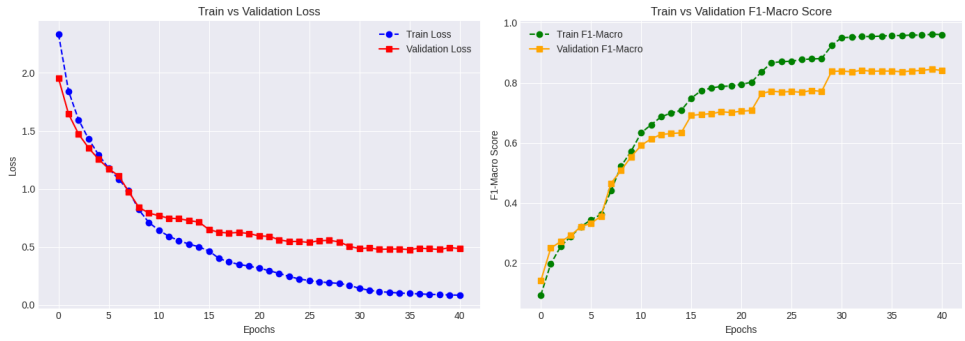

Figure 16: Combination of Hyper parameters for Model 1

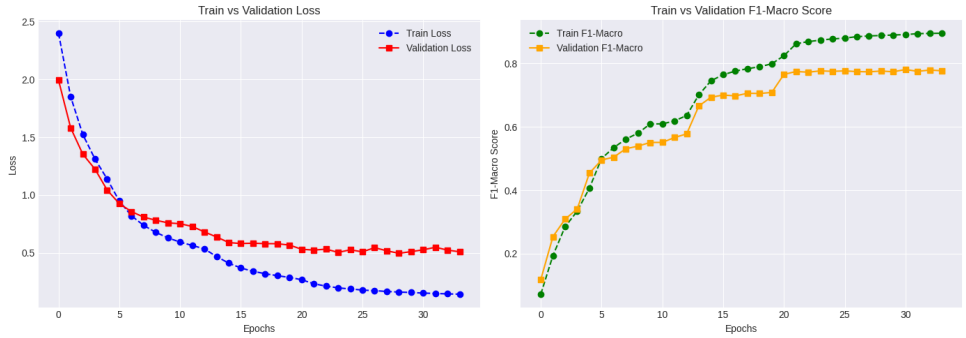

Figure 17: Combination of Hyper parameters for Model 2

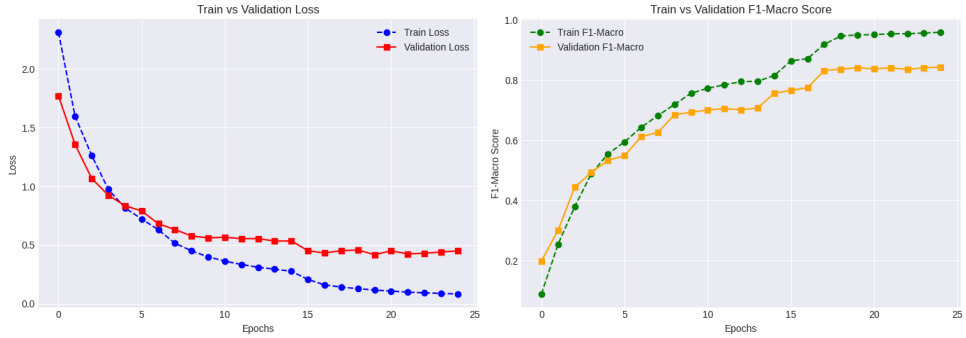

Figure 18: Combination of Hyper parameters for Model 3

Analyzing the results from the loss and F1-macro score plots, we observe that the optimal model is **Model 3**. It achieves the lowest loss with minimal gap between training and validation losses, indicating reduced overfitting. Additionally, it attains the highest macro F1 score, demonstrating superior predictive performance.

Leveraging the optimal hyperparameter combination, we trained the final model using the training and test datasets. We calculated performance metrics, including precision, recall, F1-score, macro average, and PR-AUC per class, for all dataset splits Tables 3 (metrics for training set), 4 (metrics for dev set), 5 (metrics for test set). Additionally, we replotted the loss curves in Figure 19 for the training and validation sets to further analyze model convergence and performance stability .

Class	Precision	Recall	F1-score	AUC-PR
DET	0.9901	0.9942	0.9922	0.9924
SYM	0.9537	0.8987	0.9254	0.9264
PRON	0.9920	0.9942	0.9931	0.9934
INTJ	0.9743	0.8834	0.9266	0.9290
PROPN	0.9802	0.9745	0.9774	0.9782
ADP	0.9740	0.9858	0.9798	0.9805
NOUN	0.9857	0.9869	0.9863	0.9874
CCONJ	0.9879	0.9950	0.9915	0.9916
AUX	0.9929	0.9936	0.9933	0.9935
ADJ	0.9725	0.9775	0.9750	0.9757
PART	0.9860	0.9890	0.9875	0.9876
ADV	0.9620	0.9570	0.9595	0.9605
X	0.9580	0.6884	0.8012	0.8236
PUNCT	0.9965	0.9980	0.9973	0.9974
NUM	0.9848	0.9872	0.9860	0.9861
VERB	0.9902	0.9896	0.9899	0.9905
SCONJ	0.9509	0.9044	0.9271	0.9285
Macro AUC-PR		0.9660		

Table 3: Train Set Performance Metrics

Class	Precision	Recall	F1-score	AUC-PR
DET	0.9783	0.9872	0.9827	0.9833
SYM	0.8615	0.6914	0.7671	0.7770
PRON	0.9832	0.9854	0.9843	0.9849
INTJ	0.8795	0.6404	0.7411	0.7608
PROPN	0.7864	0.6991	0.7402	0.7540
ADP	0.9199	0.9719	0.9452	0.9470
NOUN	0.8404	0.9120	0.8747	0.8836
CCONJ	0.9845	0.9948	0.9896	0.9897
AUX	0.9787	0.9831	0.9809	0.9815
ADJ	0.8606	0.8821	0.8712	0.8758
PART	0.9608	0.9623	0.9615	0.9620
ADV	0.9163	0.8364	0.8745	0.8803
X	0.0000	0.0000	0.0000	0.0012
PUNCT	0.9930	0.9893	0.9911	0.9918
NUM	0.9547	0.7825	0.8601	0.8703
VERB	0.9233	0.9202	0.9217	0.9260
SCONJ	0.8921	0.7806	0.8327	0.8381
Macro AUC-PR		0.8475		

Table 4: Validation Set Performance Metrics

Class	Precision	Recall	F1-score	AUC-PR
DET	0.9851	0.9946	0.9898	0.9901
SYM	0.9053	0.7963	0.8473	0.8512
PRON	0.9905	0.9826	0.9866	0.9873
INTJ	0.9394	0.7686	0.8455	0.8546
PROPN	0.7499	0.7806	0.7649	0.7743
ADP	0.9334	0.9651	0.9490	0.9506
NOUN	0.8790	0.8825	0.8807	0.8903
CCONJ	0.9944	0.9930	0.9937	0.9938
AUX	0.9809	0.9881	0.9845	0.9849
ADJ	0.8603	0.8960	0.8778	0.8819
PART	0.9540	0.9857	0.9696	0.9700
ADV	0.9035	0.8839	0.8936	0.8964
X	0.1333	0.0488	0.0714	0.0919
PUNCT	0.9917	0.9897	0.9907	0.9913
NUM	0.9349	0.6761	0.7847	0.8090
VERB	0.9256	0.9317	0.9286	0.9322
SCONJ	0.8943	0.7789	0.8326	0.8383
Macro AUC-PR		0.8640		

Table 5: Test Set Performance Metrics

The best model was obtained at **Epoch 21**, with a **Validation Loss of 0.40560**.

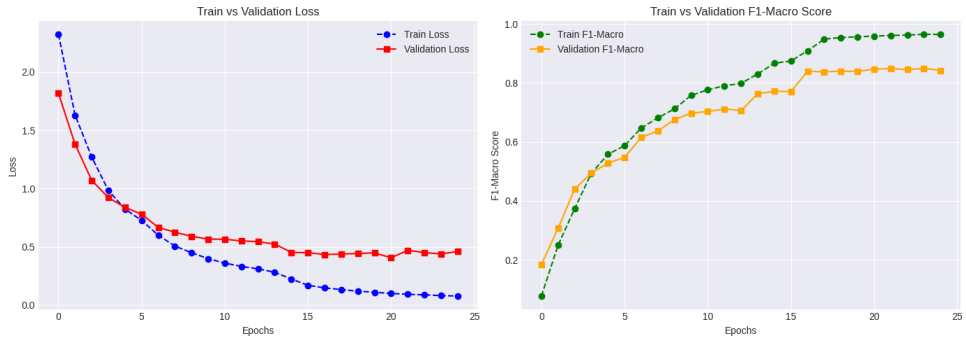

Figure 19: Optimal Model Plots

The RNN model demonstrates superior performance compared to both the Majority Baseline and the Optimal MLP classifier from Task 10. In terms of accuracy (F1-score) (in test set), the RNN achieves 0.92, outperforming the Optimal MLP (0.91) and the Majority Baseline (0.86). Similarly, for the macro average F1-score (in test set), the RNN attains 0.86, which is higher than the Optimal MLP (0.83) and the Majority Baseline (0.80). These results indicate that the RNN effectively captures sequential dependencies in the data, leading to better overall generalization compared to the simpler baselines.