Exercises on Natural Language Processing with Convolutional Neural Networks (CNN) № 4

Kagioglou Maria, Panourgia Evangelia 5/03/2025

Assignment 2

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

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

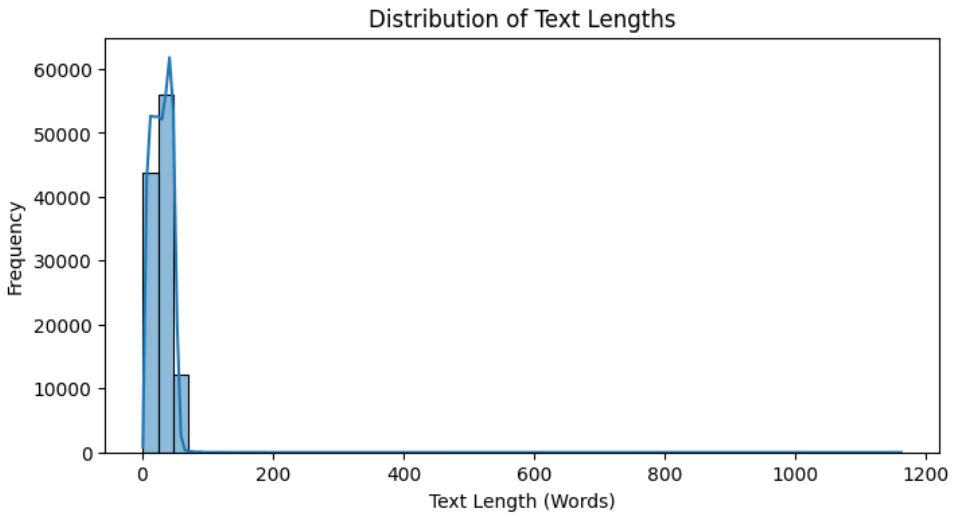

Figure 1: Dataset Statistics

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']
```

CNN Model

In order to bulid the CNN 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
```

3. `CNNModel` Class

Proceeding further, we define a custom **CNN model** that intergrates several components:

- Convolutional Layers: number of filters and the kernel size are configurable, allowing flexibility for capturing different n-gram patterns in the data. It uses one type of convolutional layer.
- **Attention Mechanism**
- **Layer Normalization**
- **Dropout**
- **Pooling Layer (max pooling)**

Moreover, we defined a **CNN with Multi-Filter Convolutions**:

- **Convolutional Layers:** Uses multiple convolutional layers with different kernel sizes (bi-gram, trigram, and fourgram), which allows the model to capture multiple levels of n-gram features simultaneously. Specifically, **it has three distinct convolution layers, one for each n gram (2 gram, 3 gram, 4 gram) and processes them in parallel.** The use of multi-filter convolutional layers enables the model to capture rich local features.
- Similar to the **CNN model**, it also supports **pooling** or **attention**, **layernorm** and **dropout**.

Evaluate Models

The next step was to try different models in order to evaluate their performance. Below, we can see the models ranked from worst to best:

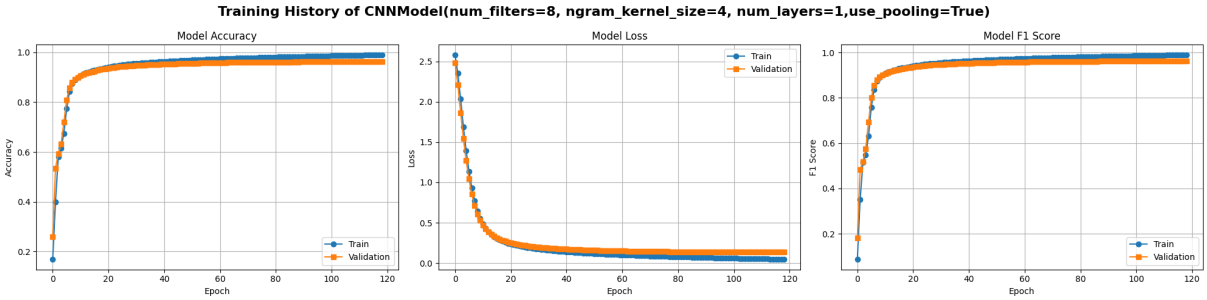

Figure 2: CNN fourgram model with pooling.

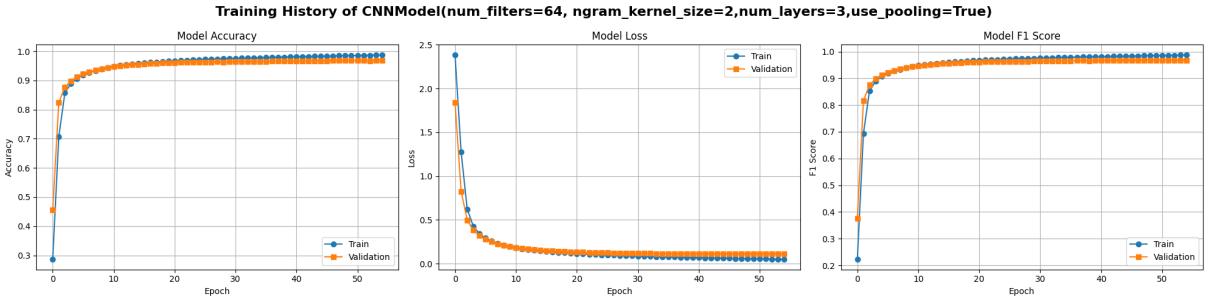

Figure 3: CNN bigram model with pooling.

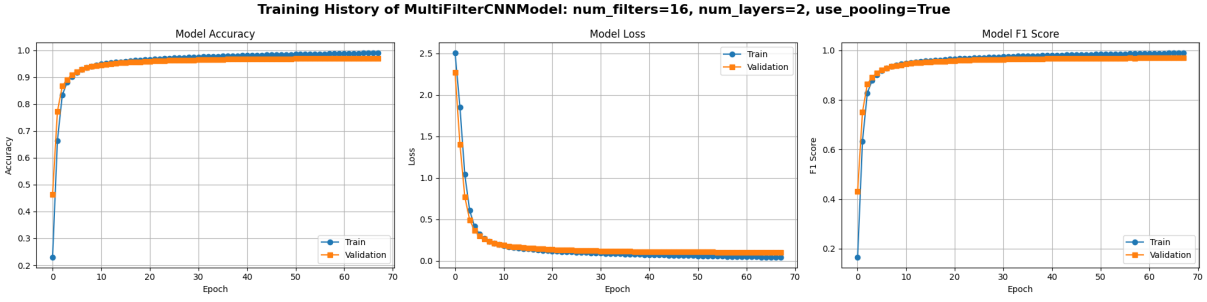

Figure 4: MultiFilterCNN model with pooling and 2 layers.

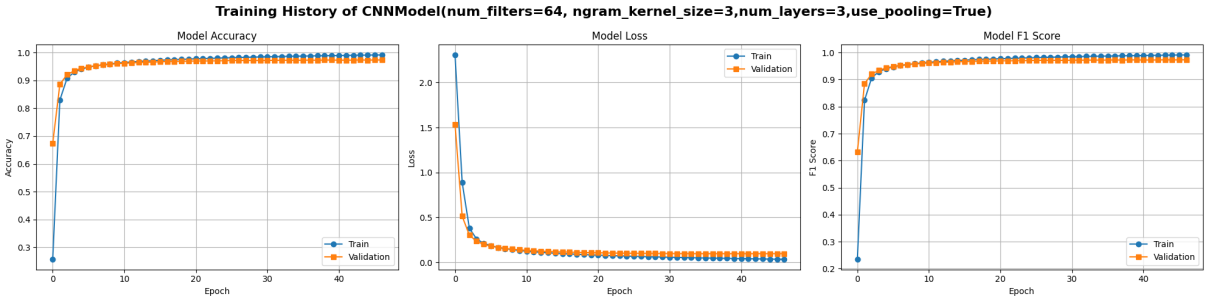

Figure 5: CNN trigram model with pooling.

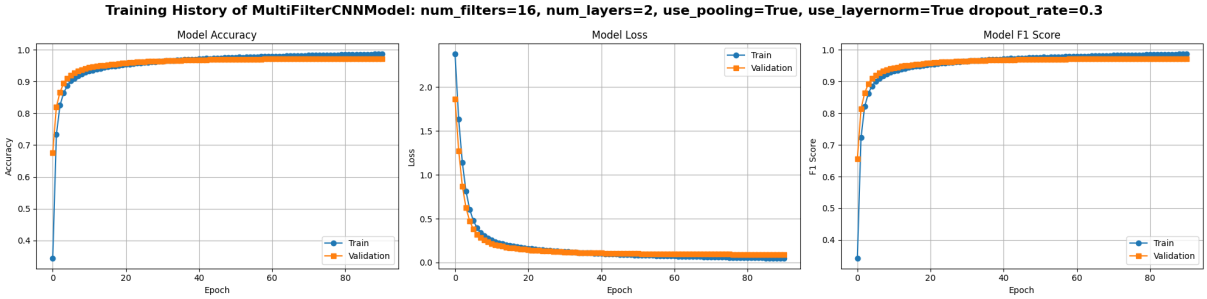

Figure 6: MultiFilterCNN model with 2 layers, pooling, layernorm and dropout.

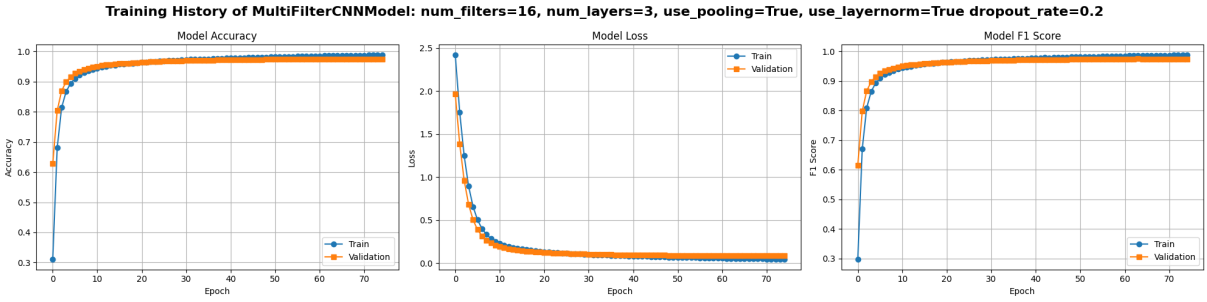

Figure 7: MultiFilterCNN model with 3 layers, pooling, layernorm and dropout.

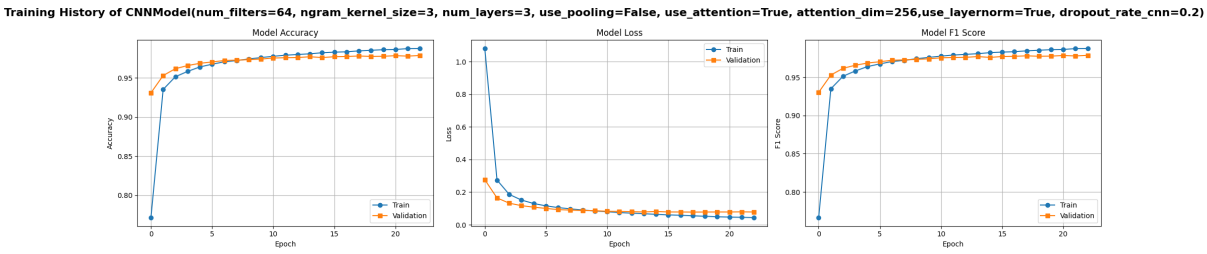

Figure 8: CNN trigram model with pooling, layernorm and dropout.

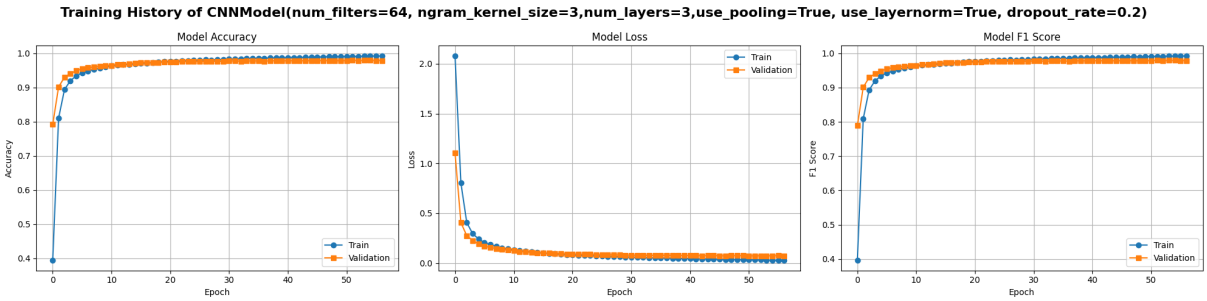

Figure 9: CNN trigram model with pooling, layernorm and dropout **(Best CNN model)**.

Figure	Epoch	Train Loss	Val. Loss	Train Accuracy	Val. Accuracy	Train F1	Val. F1
Figure 2	113	0.05050	0.13538	0.9845	0.9617	0.9884	0.9618
Figure 3	51	0.05475	0.11103	0.9856	0.9678	0.9856	0.9678
Figure 4	63	0.04590	0.10309	0.9881	0.9691	0.9881	0.9691
Figure 5	39	0.04490	0.09686	0.9885	0.9729	0.9885	0.9729
Figure 6	91	0.04832	0.09235	0.9872	0.9725	0.9872	0.9725
Figure 7	69	0.04848	0.08695	0.9870	0.9746	0.9870	0.9746
Figure 8	19	0.05185	0.07791	0.9854	0.9775	0.9854	0.9775
Figure 9	55	0.02964	0.07291	0.9915	0.9790	0.9915	0.9790

Table 2: Performance metrics for Figures 2–9.

Below, it is displayed the classification report of the best CNN model (Figure 9)

Test Set

Class	Precision	Recall	F1-score	Support	PR AUC
0	0.9651	0.9304	0.9475	1250	0.9849
1	0.9707	0.9816	0.9761	1250	0.9937
2	0.9550	0.9512	0.9531	1250	0.9909
3	0.9912	0.9888	0.9900	1250	0.9988
4	0.9743	0.9696	0.9719	1250	0.9954
5	0.9825	0.9872	0.9848	1250	0.9971
6	0.9656	0.9648	0.9652	1250	0.9926
7	0.9849	0.9936	0.9892	1250	0.9995
8	0.9992	0.9960	0.9976	1250	0.9998
9	0.9755	0.9864	0.9809	1250	0.9986
10	0.9864	0.9840	0.9852	1250	0.9983
11	0.9833	0.9920	0.9877	1250	0.9982
12	0.9809	0.9880	0.9845	1250	0.9986
13	0.9681	0.9696	0.9688	1250	0.9939
Accuracy	0.9774				
Macro Avg	0.9773	0.9774	0.9773	17500	0.9945
Weighted Avg	0.9773	0.9774	0.9773	17500	

Table 3: Test Set Classification Report with Precision-Recall AUC

Baseline Models

RNN-MLP-Logistic Regression-Majority & Random Classifier

In (Figure 10), we can see the results for the **best rnn model**:

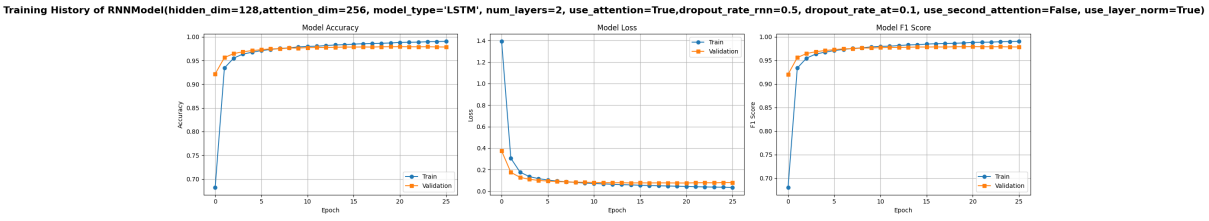

Figure 10: RNN model

In (Figure 11), we can see the results for the **best mlp model**:

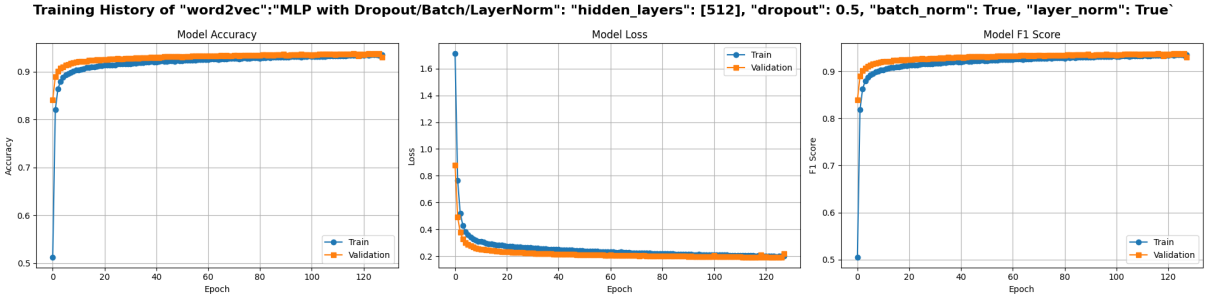

Figure 11: MLP model

Comparison

Below, we present the macro metrics for test set for the CNN, RNN, MLP, Logistic Regression, Majority & Random Classifier:

Metric	CNN	RNN	MLP	Logistic Regression	Majority	Random
Macro Precision	0.9773	0.9770	0.9379	0.9373	0.93	0.0731
Macro Recall	0.9774	0.9770	0.9378	0.9373	0.07	0.0731
Macro F1-score	0.9773	0.9771	0.9378	0.9372	0.01	0.0731
Macro AUC	0.9945	0.9938	0.9787	0.9782	-	0.5357
Best Val. loss	0.07291	0.07262	0.19186	-	-	-

Table 4: Macro Metrics for the Test Set (CNN, RNN, MLP, Logistic Regression, Majority Classifier, and Random Classifier)

As we can see from the above results, the RNN & CNN models outperform the baseline models, achieving better metric results and lower training and validation losses, while RNN & CNN achieve quite similar results.

Assignment 3

CNN (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 12 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.

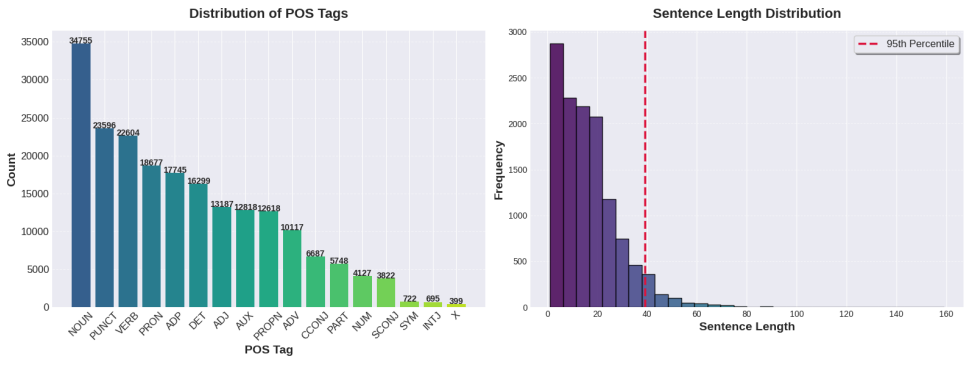

Figure 12: Setence and POS Distribution

We also implemented two CNN-based POS tagging model that support multiple functionalities. The POSCNNModel class is a neural network implemented in PyTorch for Part-of-Speech (POS) tagging, utilizing convolutional layers to extract hierarchical features from input word embeddings. the model allows for optional use of pretrained word embeddings, multiple convolutional layers, dropout regularization, and and attention-based output computations, making it

adaptable to various NLP tasks. Regarding the second class, the MultiFilterCNN_POS class supports POS tagging by leveraging multiple convolutional filters (bigram, trigram, and four-gram) to capture varying n-gram features, enhancing contextual representation for each token. It provides dropout, pre-trained embeddings, freeze embeddings, attention mechanism for learning token-specific importance, making it adaptable to different NLP tasks.

At this point worth mentioning that in POS tagging, using pooling is not valid for per token classification so we did not implemented in this task.

We optimized the model's performance through a systematic manually hyperparameter tuning process based on train and dev sets for both classes that we have already describe above. 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 selected this score compared to accuracy as from Figure 12 there is imbalances in y being POS tags). We utilized early stopping (with patient 5) 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 for each of the two classes. We experimented in number of layers, number of filters, dropout, learning rate etc. (all configurations checked are presenting in the corresponding jupyter notebook).

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

Analyzing the results from the loss and F1-macro score plots in Figures 13, 14, we observe that the optimal model is **from the class MultiFilterCNN_POS** having the name **More Regularization**. It achieves one of the lowest loss with minimal gap between training and validation losses, indicating reduced overfitting. Additionally, it attains one of the highest macro F1 score, demonstrating superior predictive performance. It also has a logic behaviour across epochs. At this point it is worth mentioning that we did not selected the model Faster Convergence having quite close scores with the pre-selected one as it trained only for 3 epochs, the other model presented better behaviour across epochs. In the Jupyter Notebook, we also print the values of both loss functions and the F1 macro score analytically. Additionally, we create separate dataframes for each class, containing the corresponding metric values. This approach ensures a more accurate and convenient comparison of the results. Quite close was the performance of model Larger Filters & Lower LR.

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 5 (metrics for training set), 6 (metrics for dev set), 7 (metrics for test set).

The optimal model was achieved at Epoch 20, with a validation loss of approximately 0.32. The best configuration, **"More Regularization"** (stemming from the class **from the class MultiFilterCNN_POS**), is given by num_filters $= 128$, dropout_rate $= 0.5$, learning_rate $= 3 \times 10^{-4}$, weight_decay $= 10^{-4}$, step_size $= 6$, $\gamma = 0.75$, and num_layers $= 2$ with attention, optimizing both generalization and performance.

To evaluate the performance of our model, we compare it against the **Majority Baseline** and the **Optimal MLP** classifier from Task 10. Additionally, we analyze its performance relative to the **Optimal RNN** from the previous assignment, where a window size of 3 was used for the POS tagging task. In terms of **accuracy (F1 score)**, the Majority Baseline achieves 0.86, which is outperformed by the Optimal MLP with 0.91, while the Optimal RNN achieves the highest score of 0.92. Similarly, for the **macro-average F1 score**, the Majority Baseline reaches 0.80, whereas the Optimal MLP improves upon it with 0.83, and the Optimal RNN further outperforms both with a score of 0.86. Based on these results, it is evident that our RNN surpasses both the Majority Baseline and the Optimal MLP classifier.

Having the aforementioned ones in mind, **further comparing the Optimal RNN and the Optimal CNN**, we observe that the **CNN achieves an accuracy (F1 score) of** 0.92, which is identical to that of the RNN and higher than the Optimal MLP (0.91) and the Majority Baseline (0.86). However, when examining the **macro-average F1 score, the CNN achieves a score of** 0.84**, which is slightly lower than that of the RNN**, which stands at 0.86, but regarding the corresponding values of majority and optimal MLP our Optimal CNN outperforms. Regarding CNN and RNN comparison, this suggests that while both models demonstrate strong overall performance, the **RNN has a slight advantage in terms of handling class balance and generalization**, particularly when considering macro-average metrics.

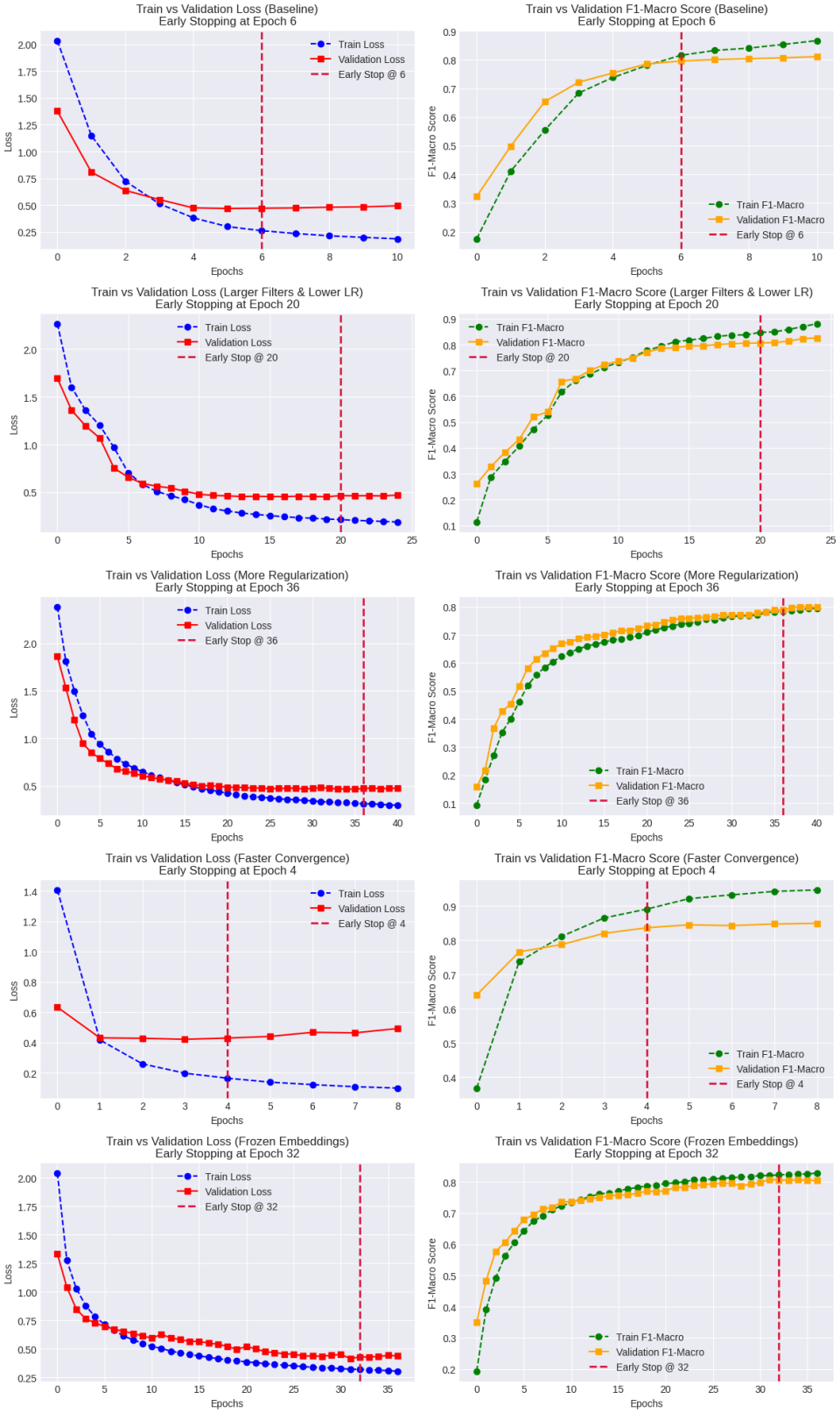

Figure 13: Losses Plots for the Class **POSCNNModel**

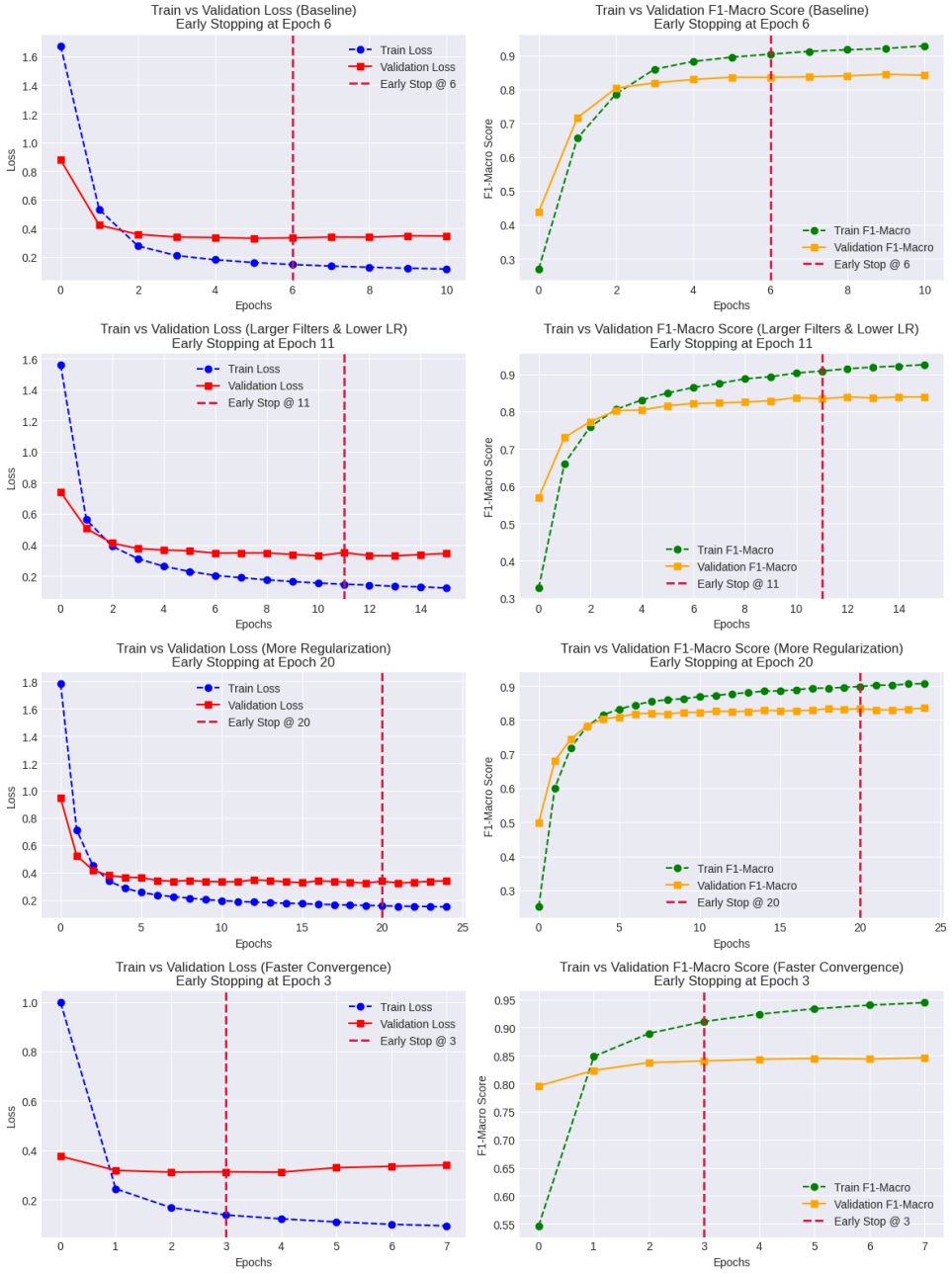

Figure 14: Losses Plots for the Class **MultiFilterCNN_POS**

Class	Precision	Recall	F1 Score	AUC-PR
CCONJ	0.9846	0.9877	0.9861	0.9864
NUM	0.9583	0.9791	0.9686	0.9689
ADP	0.9281	0.9669	0.9471	0.9490
PROPN	0.9390	0.8942	0.9160	0.9199
ADV	0.9076	0.8772	0.8922	0.8955
SYM	0.8963	0.7539	0.8189	0.8255
AUX	0.9764	0.9823	0.9793	0.9799
PUNCT	0.9904	0.9966	0.9935	0.9937
PRON	0.9796	0.9772	0.9784	0.9795
X	0.9147	0.2965	0.4478	0.6063
SCONJ	0.8526	0.7675	0.8078	0.8122
VERB	0.9662	0.9624	0.9643	0.9664
INTJ	0.9543	0.7303	0.8274	0.8428
DET	0.9810	0.9858	0.9834	0.9840
ADJ	0.9305	0.9421	0.9363	0.9382
NOUN	0.9475	0.9651	0.9562	0.9593
PART	0.9649	0.9760	0.9704	0.9708
Macro AUC-PR		0.9164		

Table 5: Per-Class Metrics on the Training Set

Class	Precision	Recall	F1 Score	AUC-PR
CCONJ	0.9921	0.9869	0.9895	0.9897
NUM	0.9640	0.7109	0.8183	0.8397
ADP	0.9089	0.9694	0.9382	0.9404
PROPN	0.7662	0.6670	0.7132	0.7290
ADV	0.9298	0.8256	0.8746	0.8820
SYM	0.8182	0.6667	0.7347	0.7430
AUX	0.9674	0.9812	0.9743	0.9749
PUNCT	0.9923	0.9893	0.9908	0.9914
PRON	0.9853	0.9799	0.9826	0.9835
X	1.0000	0.0345	0.0667	0.5184
SCONJ	0.8580	0.7551	0.8033	0.8085
VERB	0.9096	0.9164	0.9130	0.9175
INTJ	0.9429	0.5789	0.7174	0.7619
DET	0.9763	0.9888	0.9825	0.9830
ADJ	0.9100	0.8680	0.8885	0.8939
NOUN	0.8196	0.9241	0.8687	0.8782
PART	0.9343	0.9607	0.9473	0.9480
Macro AUC-PR		0.8696		

Table 6: Per-Class Metrics on the Development Set

Class	Precision	Recall	F1 Score	AUC-PR
CCONJ	0.9930	0.9944	0.9937	0.9938
NUM	0.9271	0.5989	0.7277	0.7673
ADP	0.9258	0.9661	0.9455	0.9473
PROPN	0.7790	0.7090	0.7424	0.7560
ADV	0.9220	0.8605	0.8902	0.8945
SYM	0.8208	0.8056	0.8131	0.8136
AUX	0.9757	0.9828	0.9792	0.9798
PUNCT	0.9930	0.9864	0.9897	0.9905
PRON	0.9822	0.9840	0.9831	0.9838
X	0.0000	0.0000	0.0000	0.0008
SCONJ	0.8889	0.7579	0.8182	0.8253
VERB	0.9282	0.9333	0.9307	0.9342
INTJ	0.9639	0.6612	0.7843	0.8133
DET	0.9829	0.9876	0.9853	0.9857
ADJ	0.9131	0.8835	0.8981	0.9025
NOUN	0.8208	0.9257	0.8701	0.8794
PART	0.9448	0.9762	0.9602	0.9608
Macro AUC-PR		0.8487		

Table 7: Per-Class Metrics on the Test Set