Exercises on Natural Language Processing with Transformers№ 5

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 **28,000 instances** for the training set and **14,000 instances** for the test set.

After downsampling, we further split the **test set** to create a **development set** using a **0.5** proportion of the test data. Since the **training set** was already predefined, no further modifications were needed. The final dataset sizes were as follows:

- **Training set size:** 28,000
- **Development set size:** 7,000
- **Test set size:** 7,000

- **Tokenizer:** We utilized the `distilroberta-base` tokenizer from Hugging Face to tokenize the dataset. This tokenizer is specifically designed for the DistilRoBERTa model, which is a smaller, more efficient version of the original RoBERTa model.
- **Truncation:** To ensure that the sequences do not exceed the maximum input size, we applied truncation, which limits the length of each sequence to **512 tokens**.
- **Padding:** Sequences were padded dynamically to match the longest sequence in each batch, rather than using a fixed length. This ensures efficient memory usage and proper batch processing.
- **Batch Processing:** We used the `map()` function from the Hugging Face `datasets` library to apply the tokenizer in batches, making the tokenization process efficient across the entire dataset.

We also implemented a custom training callback `CustomCallback(TrainerCallback)` for evaluation at the end of each epoch.

The callback performs the following actions:

- It copies the control state to prevent unwanted modifications.
- Evaluates on the training dataset, logging the training loss and relevant metrics.
- Evaluates on the validation dataset and ensures that the `eval_loss` metric is logged, even if it is missing from the results.

This custom callback is valuable for tracking both training and evaluation metrics, ensuring that performance on both datasets is consistently monitored throughout the training process.

During the training process across all models, the same optimizer and data collator were used to ensure consistency in the setup. Specifically, the **AdamW optimizer** was employed throughout.

The **DataCollatorWithPadding** was used as the data collator, ensuring that all input sequences in a batch are padded to the same length, which is crucial for efficient batch processing.

Additionally, the **EarlyStoppingCallback** was consistently used to monitor the validation loss and stop training early if no improvement was seen for a set number of steps, preventing overfitting and ensuring that the model performs optimally without wasting computational resources.

Following the tokenization process, we performed the following experiments to evaluate different model configurations:

1. **DistilRoBERTa Model for Sequence Classification:** We fine-tuned the pre-trained `distilroberta-base` model for sequence classification with 14 output labels. The model was adapted to the specific task by mapping the model's outputs to the corresponding labels.

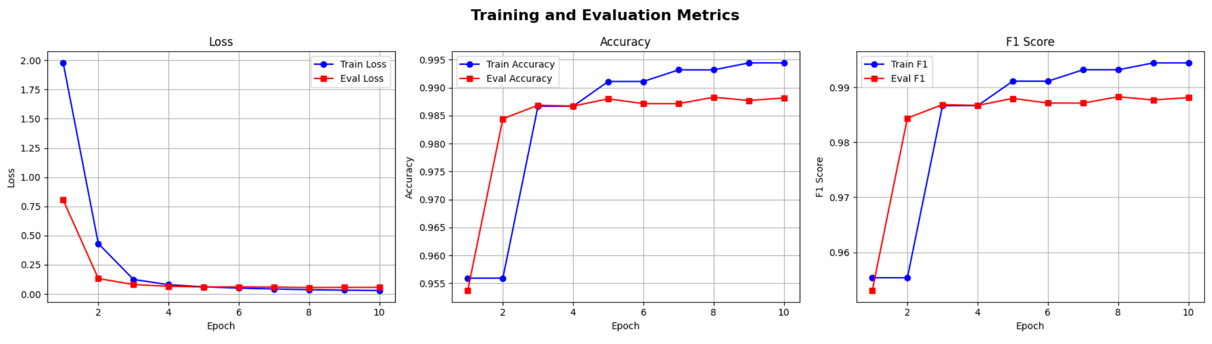

Figure 1: Training Parameters: Epochs=20, Batch Size=32, Gradient Accumulation=16, Learning Rate=1e-5, Weight Decay=0.1.

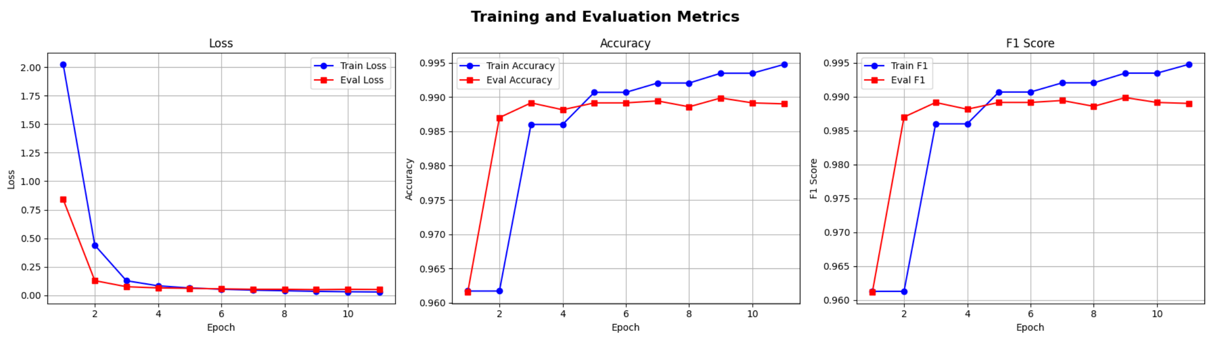

Figure 2: Training Parameters: Epochs=20, Batch Size=32, Gradient Accumulation=16, Learning Rate=1e-5, Weight Decay=0.0.

2. **Freezing the Base Model (DistilRoBERTa):** To reduce computational cost and focus the training on the classification head, we froze the parameters of the pre-trained DistilRoBERTa backbone, allowing only the classification head to be trained.

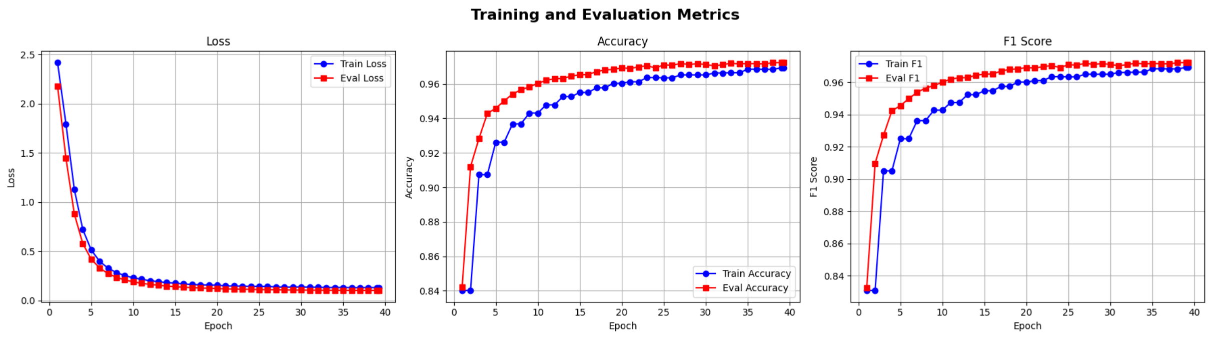

Figure 3: Training Parameters: Epochs=40, Batch Size=32, Gradient Accumulation=16, Learning Rate=2e-4, Weight Decay=0.0.

3. **Low-Rank Adaptation (LoRA) Model Configuration:** We applied LoRA to the pre-trained model, which is a parameter-efficient adaptation technique. The task type was set to `TaskType.SEQ_CLS` for sequence classification tasks, **the rank (`r`) was set to 8**, **the scaling factor `lora_alpha` was set to 32** to control the strength of the adaptation, and a **dropout rate of 0.1**.

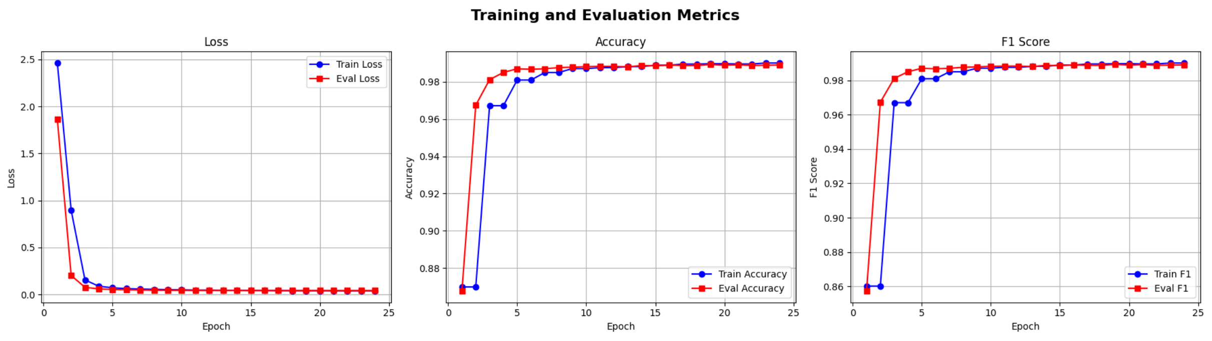

Figure 4: Training Parameters: Epochs=25, Batch Size=32, Gradient Accumulation=16, Learning Rate=2e-5, Weight Decay=0.0.

4. **Custom Model with Dropout and Max Pooling:** This model incorporated a dropout layer and max pooling over the sequence's hidden states. The final pooled representation was passed through a classification head for label prediction. The dropout layer helped in preventing overfitting.

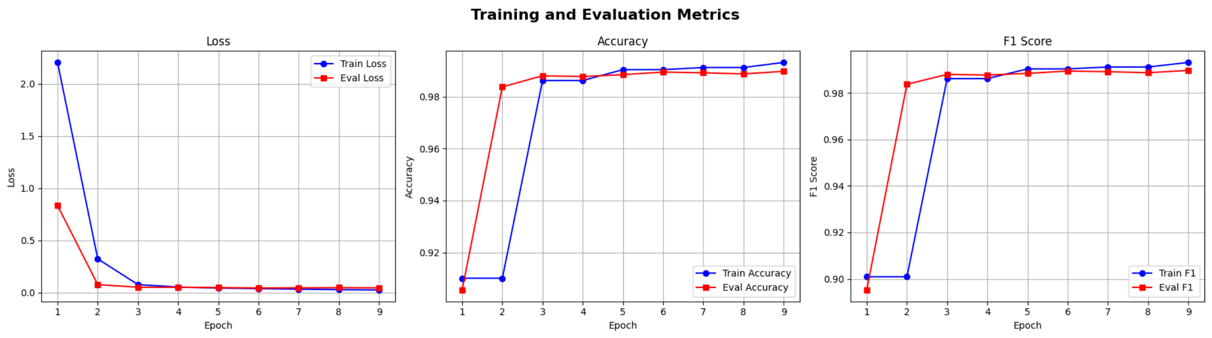

Figure 5: Training Parameters: Epochs=25, Batch Size=32, Gradient Accumulation=16, Learning Rate=1e-5, Weight Decay=0.0, Dropout=0.5

5. **Custom Model with Attention:** In this model, a self-attention mechanism was introduced to weight each token's contribution to the final representation. The model computed attention scores for each token, applying softmax to the scores, and then aggregated the weighted sequence outputs before passing them through a classification head.

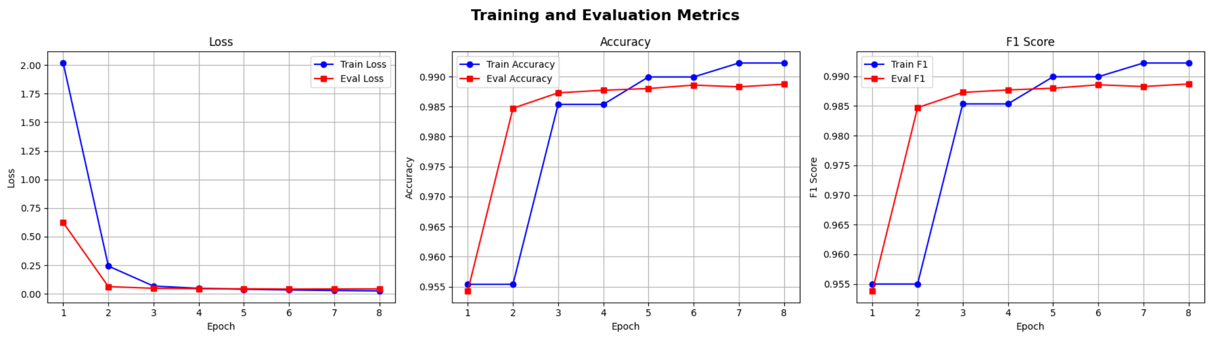

Figure 6: Training Parameters: Epochs=25, Batch Size=32, Gradient Accumulation=16, Learning Rate=1e-5, Weight Decay=0.0, Dropout=0.5

Best Model-Comparison with Baselines

The best model using the **Transformers** framework was selected based on the **lowest validation loss**, the highest **F1-score** and **accuracy**, as well as its ability to effectively retain **training performance over multiple epochs**. According to this criteria the best model is: **Low-Rank Adaptation (LoRA) Model Configuration**, displaying the following results in test set:

Metric	**Value**
test_loss	0.0398
test_accuracy	0.9906
test_f1	0.9906
test_runtime	18.0398 seconds
test_samples_per_second	388.03
test_steps_per_second	48.504

Table 1: Test Results

Class	**Precision**	**Recall**	**F1-score**
0	0.99	0.97	0.98
1	0.98	0.99	0.99
2	0.98	0.98	0.98
3	0.99	1.00	1.00
4	0.99	0.98	0.98
5	0.99	1.00	1.00
6	0.98	0.98	0.98
7	0.99	1.00	1.00
8	1.00	1.00	1.00
9	1.00	0.99	0.99
10	0.99	0.99	0.99
11	1.00	0.99	0.99
12	0.99	0.99	0.99
13	0.99	0.99	0.99
Accuracy		0.99	
Macro Avg	0.99	0.99	0.99
Weighted Avg	0.99	0.99	0.99

Table 2: Classification Report for Model in Test Set.

In the following table, the results across all models (Transformers and baselines) are presented. As we can see, the Transformers outperform the baselines:

Metric	**Transformer**	**CNN**	**RNN**	**MLP**	**Logistic Regression**
Macro Precision	0.9906	0.9773	0.9770	0.9379	0.9373
Macro Recall	0.9906	0.9774	0.9770	0.9378	0.9373
Macro F1-score	0.9906	0.9773	0.9771	0.9378	0.9372
Best Val.loss	0.04176	0.07291	0.07262	0.19186	-
Test Accuracy	0.9906	0.9906	0.9906	0.9906	0.9906
Test F1	0.9906	0.9906	0.9906	0.9906	0.9906

Table 3: Comparison of different models with metrics and test results.

Assignment 2

Transformers (POS) Tagger

We worked on downloading and processing CoNLL-U formatted datasets. Our goal was to extract words (tokens) and Part-of-Speech (POS) tags from each sentence and organize them into structured Pandas DataFrames. We implemented the function `download_and_read_conllu(url)`, which fetched a CoNLL-U formatted dataset from a given URL, processed it into a structured Pandas DataFrame, extracted words (tokens) and their corresponding POS tags, and assigned a unique sentence ID to each sentence. We handled three different datasets: training, development (validation), and test datasets. Each dataset was processed separately and stored in Pandas DataFrames: `train_df` for training, `dev_df` for development, and `test_df` for testing. This structured data facilitated further analysis such as POS tagging and syntactic parsing. By processing the CoNLL-U datasets, we created structured data that could be used for various natural language processing tasks.

We loaded a total of 12,544 sentences for training, 2,001 sentences for validation, and 2,077 sentences for testing. This structured data facilitated further analysis such as POS tagging and syntactic parsing. By processing the CoNLL-U datasets, we created structured data that could be used for various natural language processing tasks. This assignment helped us understand data handling, token extraction, and POS tagging in computational linguistics.

To further process the POS tags, we created a dictionary `pos_mapping_dict` that assigned a unique numerical ID to each POS tag. We then implemented the function `map_pos_to_category(pos_tags)`, which took a list of POS tags and replaced each tag with its corresponding numeric value from the dictionary. This step ensured that the POS tags were converted into a numerical format suitable for computational analysis and machine learning applications.

Additionally, the tokens and POS columns in `train_df`, `test_df`, and `dev_df` were converted from space-separated strings to lists using the `.str.split()` function. This conversion made it easier to manipulate and analyze the individual tokens and their respective POS tags. The POS tags were then further transformed into numerical categories using the `map_pos_to_category` function, ensuring consistency and compatibility with machine learning models.

By processing the CoNLL-U datasets, converting text data into structured lists, and mapping POS tags to numerical values, we created a dataset that could be effectively used for various natural language processing tasks. This assignment helped us understand data handling, token extraction, POS tagging, and feature engineering in computational linguistics.

Then, we converted Pandas DataFrames into Hugging Face Dataset objects. We stored them in a `DatasetDict` for structured management. This approach provided an efficient and standardized format for handling NLP datasets.

The `DatasetDict` provides a structured way to store and manage the datasets, making it easier to handle different subsets during model training and evaluation. Each subset maintains a consistent format, ensuring compatibility with NLP pipelines. By organizing the data into a standardized structure, we facilitate efficient preprocessing, model input preparation, and performance evaluation. This dataset structure enables seamless integration with deep learning frameworks and allows for streamlined experimentation in POS tagging and other token-based analyses.

The model selection is a crucial step for NLP tasks. We use `model_name = "distilbert-base-cased"`, which specifies **DistilBERT** (`https://huggingface.co/distilbert/distilbert-base-cased`), a lightweight version of BERT. The `AutoTokenizer` from Hugging Face is used to load the tokenizer for this model.

The tokenization process is performed using the pretrained tokenizer. First, the tokenizer is initialized with `tokenizer = AutoTokenizer.from_pretrained(model_name)`, which loads the pretrained tokenizer. Then, the function `tokenizer(pos_dataset["train"][0]["tokens"], is_split_into_words=True)` is used to tokenize the first sentence from the training dataset. The argument `is_split_into_words=True` ensures that the input is treated as a **list of words** rather than a single string. Finally, `tokenized_input["input_ids"]` provides the **token IDs**, representing the sentence in numerical form.

To inspect the tokenization output, we convert the token IDs back into readable tokens. The function `tokenizer.convert_ids_to_tokens(tokenized_input["input_ids"])` converts the token IDs back into **actual tokens**. The script then prints both the **original tokens** (`pos_dataset["train"][0]["tokens"]`) and the **tokenized version** (`tokens`).This step is essential because **BERT-based models use subword tokenization (WordPiece)**, meaning some words may be split into smaller units, affecting the alignment of POS tags.

The function `align_ner_labels_with_tokens(examples)` tokenizes input text while aligning Part-of-Speech (POS) labels with the tokenized words. It applies `tokenizer(examples["tokens"], truncation=True, is_split_into_words=True)` to ensure words are treated as lists and truncated if exceeding the model's token limit. The function retrieves word indices using `word_mapping = tokenized_data.word_ids(batch_index=index)` and assigns `-100` to special tokens (`[CLS]`, `[SEP]`), assigns POS labels to the first subword of each word, and marks subsequent subwords with `-100` to maintain label integrity. The function returns a tokenized dictionary where the `"labels"` field contains aligned POS tags. Finally, we apply the function across the dataset using `tokenized_POS = pos_dataset.map(align_ner_labels_with_tokens, batched=True)` to ensure correct label alignment.

The `DataCollatorForTokenClassification` dynamically pads input sequences, ensuring that variable-length sentences are processed correctly during training. This padding mechanism allows the model to handle different sentence lengths efficiently without unnecessary truncation or loss of information.

We use `AutoModelForTokenClassification` to load a transformer model like `distilbert-base-cased`, adding a classification head to predict POS tags for each token. The number of output labels is set using `num_labels=len(unique_pos_tags)`, leveraging transfer learning for efficient and accurate POS tagging. The model consists of several key components: an embeddings layer, transformer layers, and a classification head. The embeddings layer includes `word_embeddings` for converting tokens into 768-dimensional vectors, `position_embeddings` for encoding positional information, `LayerNorm` for stability, and `dropout` to prevent overfitting. The transformer consists of six `TransformerBlock` layers (compared to 12 in BERT), each containing a self-attention mechanism (`DistilBertSdpaAttention`) with linear layers for query, key, and value transformations (`q_lin`, `k_lin`, `v_lin`), followed by an output transformation layer (`out_lin`). The feed-forward network (`FFN`) includes `lin1` and `lin2` for expanding and compressing hidden states, with `GELUActivation` providing non-linearity. `LayerNorm` ensures stability, while dropout prevents overfitting. The classification head consists of a final `dropout` layer and a linear classifier (`Linear(768 → 17)`) that maps token embeddings to 17 possible POS tags, predicting one label per token.

Hyperparameter tuning is performed using **Optuna**, which searches for the best combination of learning rate, batch size, and weight decay while disabling Weights Biases logging to prevent external tracking. The model is trained using `Trainer` with training arguments set to run for **5 epochs**, evaluating at specific steps, and implementing **early stopping** if no improvement is observed within 3 steps to prevent overfitting. The optimization process executes **5 trials** (`n_trials=5`), aiming to minimize evaluation loss, and stores the best hyperparameters for fine-tuning. The hyperparameter optimization process identified the best values for training. The optimal **learning rate** (`2.86e-05`) balances fast convergence while avoiding instability. The **batch size per device** (`48`) ensures efficient training without excessive memory usage. A **weight decay** of `0.0071` helps prevent overfitting while maintaining generalization, leading to improved model performance.

Evaluation metrics for POS tagging include **accuracy** and **macro F1-score**. The accuracy is computed using Hugging Face's `evaluate` library, while the macro F1-score is calculated using Scikit-learn's `f1_score`, ensuring all POS tags are treated equally. The `compute_metrics(p)` function follows several steps: (1) extracting predictions and labels by converting logits to label indices, (2) mapping numerical labels to POS tags using `pos_mapping_dict`, (3) ignoring special tokens (`-100`) to focus only on actual words, and (4) flattening predictions and labels to compute accuracy and macro F1-score effectively.

The model's performance on the development set demonstrates strong predictive accuracy and efficiency. The evaluation loss is **0.1281**, indicating good model performance. The accuracy is **96.42%**, meaning the model correctly predicts **96.4%** of POS tags, while the macro F1-score of **0.8966** suggests balanced performance across all POS tags. In terms of efficiency, evaluation completes in **3.69 seconds**, processing **542.1 samples per second** and evaluating approximately **5.7 steps per second**. Additionally, with **epoch = 0.69**, these metrics are reported before completing the first epoch, highlighting the model's early effectiveness in POS tagging.

Regarding the prediction in **test set** as it is depicted in the Table 5, the model demonstrates strong overall performance in POS tagging, achieving an **accuracy of 96%**, correctly predicting most tokens. The **weighted F1-score of 96%** reflects balanced performance across all tags, while the **macro F1-score of 90%** suggests slight imbalances across classes. High-performing POS tags include **Punctuation (PUNCT)**, with **99% precision and 100% recall**, and **Determiners (DET)**, **Pronouns (PRON)**, and **Auxiliary Verbs (AUX)**, each with an **F1-score of 99%**, demonstrating high accuracy. Additionally, **Verbs (VERB)** and **Adpositions (ADP)** maintain strong classification performance with **97-98% F1-scores**. However, some tags remain challenging, such as **X (unknown words)**, which has an **F1-score of 0%**, indicating struggles with rare or ambiguous tokens. **Symbols (SYM)** and **Interjections (INTJ)** show lower recall (**80-86%**), and **Proper Nouns (PROPN)** have **88% precision and 91% recall**, possibly affected by capitalization variations.

The following table compares different POS tagging models based on accuracy and macro F1-score:

Model	Accuracy (F1 Score)	Macro AVG (F1 Score)
Majority Baseline	**0.86**	**0.80**
Optimal MLP	**0.91**	**0.83**
Optimal RNN	**0.92**	**0.86**
Optimal CNN	**0.92**	**0.84**
Transformers Model	**0.96**	**0.90**

Table 4: Comparison of POS Tagging Models on the Test Set

The **Transformer Model achieves the highest accuracy (96%)**, surpassing the **CNN, RNN, MLP, and Majority Baseline** approaches. Its **macro F1-score (0.90)** is also the highest, indicating robust performance across all POS tags. Additionally, it demonstrates **fast evaluation runtime (3.69s)** with high efficiency, processing **542 samples per second**. While **RNN and CNN models remain competitive**, the Transformers model clearly outperforms them, showcasing its effectiveness for POS tagging tasks.

POS Tag	**Precision**	**Recall**	**F1-score**	**Support**
CCONJ	0.99	0.99	0.99	736
ADV	0.94	0.93	0.94	1183
VERB	0.97	0.98	0.97	2606
ADP	0.97	0.98	0.98	2030
DET	1.00	0.99	0.99	1896
PRON	0.99	0.99	0.99	2166
AUX	0.99	0.99	0.99	1543
PART	0.99	0.99	0.99	649
PUNCT	0.99	1.00	0.99	3096
X	0.00	0.00	0.00	42
SCONJ	0.98	0.94	0.96	384
NOUN	0.94	0.94	0.94	4123
SYM	0.93	0.80	0.86	109
ADJ	0.93	0.93	0.93	1794
NUM	0.94	0.96	0.95	542
PROPN	0.88	0.91	0.90	2076
INTJ	0.96	0.76	0.85	121
Accuracy			0.96	25096
Macro Avg	0.91	0.89	0.90	25096
Weighted Avg	0.96	0.96	0.96	25096

Table 5: Classification Report for POS Tagging Model on Test Set.