Exercise 2

Repeat Exercise 1 of Part 4 (NLP with RNNs), now using a stacked CNN with n-gram filters (e.g., $n = 2, 3, 4$), residual connections, and global max-pooling at the top layer, all implemented in Keras/TensorFlow or PyTorch.

Steps:

1. Tune the hyper-parameters (e.g., values of n, number of stacked convolutional layers) on the development subset of the dataset.

2. Monitor the performance of your models on the development subset during training to decide how many epochs to use. You may optionally add an extra CNN layer to produce word embeddings from characters, concatenating each resulting character-based word embedding with the corresponding pre-trained word embedding (e.g., obtained with Word2Vec).

3. Include experimental results of a baseline majority classifier, as well as experimental results of your best classifiers from exercise 15 of Part 2, exercise 9 of Part 3, and exercise 1 of Part 4. Otherwise, the contents of your report should be as in exercise 1 of Part 4, but now with information and results for the experiments of this exercise.

4. You may optionally wish to try ensembles (e.g., majority voting of the best checkpoints, temporal averaging of the weights of the best checkpoints, combining RNN and CNN classifiers).

Report:

- Curves showing the loss on training and development data as a function of epochs.
- Precision, recall, F1, precision-recall AUC scores for each class and classifier:
  - Separate for the training, development, and test subsets.
- Macro-averaged precision, recall, F1, precision-recall AUC scores:
  - Averaging the corresponding scores over the classes, separately for the training, development, and test subsets.

- Description of the methods and datasets used:
  - Include statistics like average document length, number of training/dev/test documents, and vocabulary size.
  - Describe preprocessing steps performed.
- Optionally, try ensemble methods (e.g., majority voting of the best checkpoints, temporal averaging of the weights of the best checkpoints).

Creating a Dataset

We will use the `Dataset` class from `PyTorch` to handle the text data. We will pad the text sequences with 0 to a pre-defined length (the average number of tokens in the training split).

In [ ]:
```
df_merge = pd.concat([df_fake, df_true], axis =0 )
df_merge.head(10)
```

Out[ ]:

	title	text	subject	date	label
0	Donald Trump Sends Out Embarrassing New Year'...	Donald Trump just couldn t wish all Americans ...	News	December 31, 2017	1
1	Drunk Bragging Trump Staffer Started Russian ...	House Intelligence Committee Chairman Devin Nu...	News	December 31, 2017	1
2	Sheriff David Clarke Becomes An Internet Joke...	On Friday, it was revealed that former Milwauk...	News	December 30, 2017	1
3	Trump Is So Obsessed He Even Has Obama's Name...	On Christmas day, Donald Trump announced that ...	News	December 29, 2017	1
4	Pope Francis Just Called Out Donald Trump Dur...	Pope Francis used his annual Christmas Day mes...	News	December 25, 2017	1
5	Racist Alabama Cops Brutalize Black Boy While...	The number of cases of cops brutalizing and ki...	News	December 25, 2017	1
6	Fresh Off The Golf Course, Trump Lashes Out A...	Donald Trump spent a good portion of his day a...	News	December 23, 2017	1
7	Trump Said Some INSANELY Racist Stuff Inside ...	In the wake of yet another court decision that...	News	December 23, 2017	1
8	Former CIA Director Slams Trump Over UN Bully...	Many people have raised the alarm regarding th...	News	December 22, 2017	1
9	WATCH: Brand-New Pro-Trump Ad Features So Muc...	Just when you might have thought we d get a br...	News	December 21, 2017	1

In [ ]:

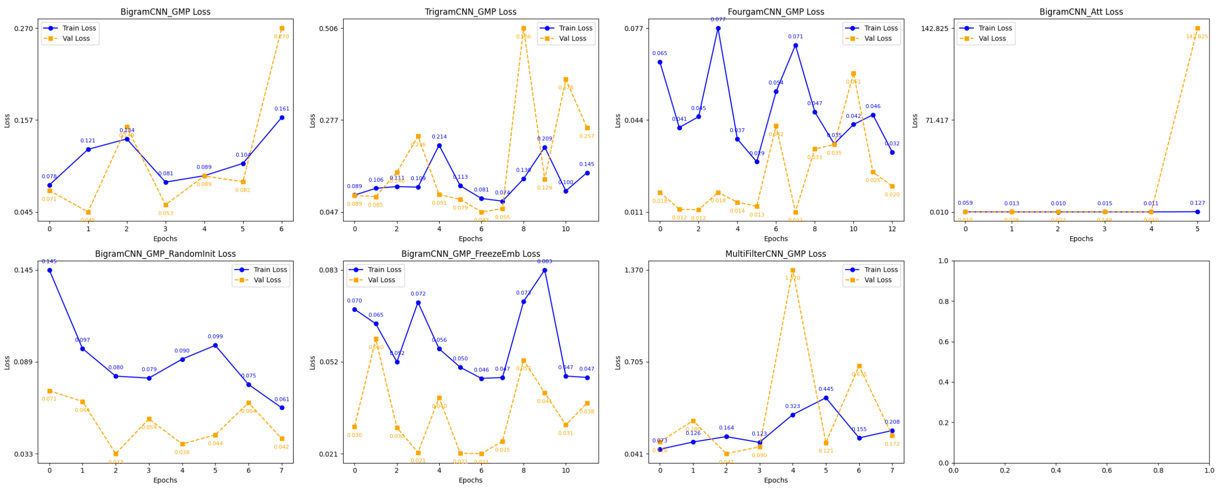
```
from sklearn.model_selection import train_test_split

# Split data into features (X) and target labels (y)
X = df['text'].values
y = df['label'].values

# Split into training, validation, and test sets with stratification
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.2, ran
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.

# Check the distribution of labels in each set
print("Training set class distribution:", np.bincount(y_train))
print("Validation set class distribution:", np.bincount(y_val))
print("Test set class distribution:", np.bincount(y_test))
```

```
Training set class distribution: [17133 18785]
Validation set class distribution: [2142 2348]
Test set class distribution: [2142 2348]
```

Define the model

We will create a model class and parameterize our neural network with several choices

In [ ]:
```
plot_loss_curves(results)
```

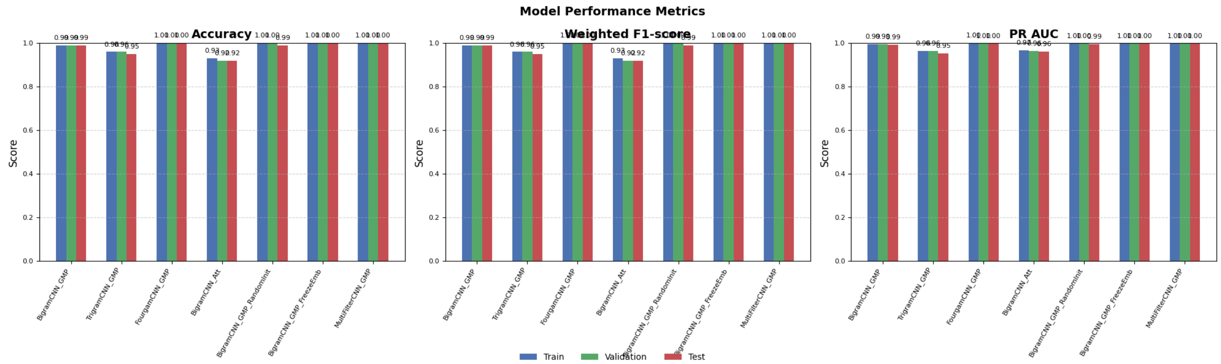


In [ ]:

	Model	Accuracy_train	Accuracy_val	Accuracy_test
0	BigramCNN_GMP	0.99	0.99	0.99
1	TrigramCNN_GMP	0.96	0.96	0.95
2	FourgamCNN_GMP	1.00	1.00	1.00
3	BigramCNN_Att	0.93	0.92	0.92
4	BigramCNN_GMP_RandomInit	1.00	1.00	0.99
5	BigramCNN_GMP_FreezeEmb	1.00	1.00	1.00
6	MultiFilterCNN_GMP	1.00	1.00	1.00

	Weighted_F1_train	Weighted_F1_val	Weighted_F1_test	PR_AUC_train
0	0.99	0.99	0.99	0.994890
1	0.96	0.96	0.95	0.963607
2	1.00	1.00	1.00	0.999716
3	0.93	0.92	0.92	0.967784
4	1.00	1.00	0.99	0.997904
5	1.00	1.00	1.00	0.998058
6	1.00	1.00	1.00	0.999606

	PR_AUC_val	PR_AUC_test
0	0.994841	0.992858
1	0.963684	0.953681
2	0.999075	0.998070
3	0.963834	0.962311
4	0.996964	0.994864
5	0.997704	0.997296
6	0.998946	0.998207



Baseline Dummy and Wide MLP

Model Performance Report

Bigram, Trigram, and Fourgram CNN Models:

1. BigramCNN_GMP

- **Accuracy (Train/Val/Test):** 0.99 / 0.99 / 0.99
- **Weighted F1 (Train/Val/Test):** 0.99 / 0.99 / 0.99
- **PR AUC (Train/Val/Test):** 0.994890 / 0.994841 / 0.992858

2. TrigramCNN_GMP

- **Accuracy (Train/Val/Test):** 0.96 / 0.96 / 0.95
- **Weighted F1 (Train/Val/Test):** 0.96 / 0.96 / 0.95
- **PR AUC (Train/Val/Test):** 0.963607 / 0.963684 / 0.953681

3. FourgramCNN_GMP

- **Accuracy (Train/Val/Test):** 1.00 / 1.00 / 1.00
- **Weighted F1 (Train/Val/Test):** 1.00 / 1.00 / 1.00
- **PR AUC (Train/Val/Test):** 0.999716 / 0.999075 / 0.998070

4. BigramCNN_Att

- **Accuracy (Train/Val/Test):** 0.93 / 0.92 / 0.92
- **Weighted F1 (Train/Val/Test):** 0.93 / 0.92 / 0.92
- **PR AUC (Train/Val/Test):** 0.967784 / 0.963834 / 0.962311

5. BigramCNN_GMP_RandomInit

- **Accuracy (Train/Val/Test):** 1.00 / 1.00 / 0.99

- **Weighted F1 (Train/Val/Test):** 1.00 / 1.00 / 0.99
- **PR AUC (Train/Val/Test):** 0.997904 / 0.996964 / 0.994864

6. **BigramCNN_GMP_FreezeEmb**

- **Accuracy (Train/Val/Test):** 1.00 / 1.00 / 1.00
- **Weighted F1 (Train/Val/Test):** 1.00 / 1.00 / 1.00
- **PR AUC (Train/Val/Test):** 0.998058 / 0.997704 / 0.997296

7. **MultiFilterCNN_GMP**

- **Accuracy (Train/Val/Test):** 1.00 / 1.00 / 1.00
- **Weighted F1 (Train/Val/Test):** 1.00 / 1.00 / 1.00
- **PR AUC (Train/Val/Test):** 0.999606 / 0.998946 / 0.998207

RNN, GRU, and LSTM Models:

1. **RNN**

- **Accuracy (Train/Val/Test):** 0.81 / 0.79 / 0.80
- **Weighted F1 (Train/Val/Test):** 0.81 / 0.79 / 0.80
- **PR AUC (Train/Val/Test):** 0.894212 / 0.878705 / 0.877910

2. **GRU**

- **Accuracy (Train/Val/Test):** 0.95 / 0.94 / 0.94
- **Weighted F1 (Train/Val/Test):** 0.95 / 0.94 / 0.94
- **PR AUC (Train/Val/Test):** 0.969749 / 0.963957 / 0.962466

3. **LSTM**

- **Accuracy (Train/Val/Test):** 0.91 / 0.90 / 0.90
- **Weighted F1 (Train/Val/Test):** 0.91 / 0.90 / 0.90
- **PR AUC (Train/Val/Test):** 0.946631 / 0.932909 / 0.933250

4. **LSTM_GMP**

- **Accuracy (Train/Val/Test):** 1.00 / 1.00 / 1.00
- **Weighted F1 (Train/Val/Test):** 1.00 / 1.00 / 1.00
- **PR AUC (Train/Val/Test):** 0.999973 / 0.999362 / 0.998946

5. **RandomInit**

- **Accuracy (Train/Val/Test):** 1.00 / 1.00 / 1.00
- **Weighted F1 (Train/Val/Test):** 1.00 / 1.00 / 1.00
- **PR AUC (Train/Val/Test):** 0.999277 / 0.998853 / 0.998003

Best Performing Models:

Top Model (Based on Accuracy, F1 Score, and PR AUC):

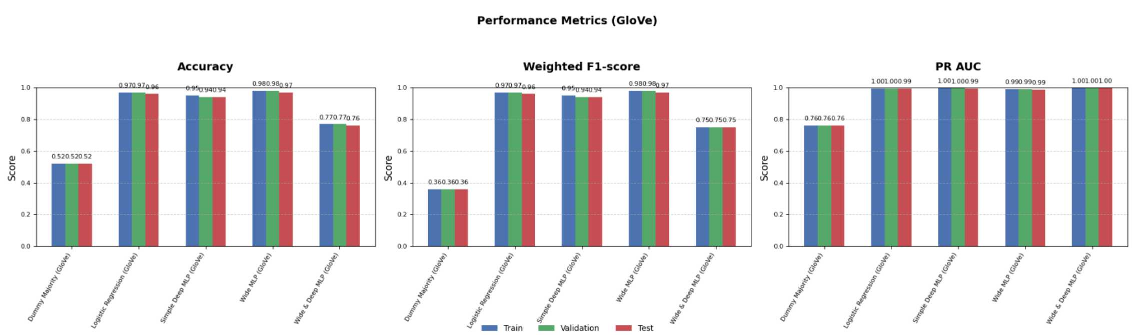
- **FourgramCNN_GMP** is the best-performing model with:
  - **Perfect accuracy** across all sets (Train/Val/Test = 1.00).
  - **Weighted F1 score of 1.00** for Train/Val/Test.
  - **PR AUC**: 0.999716 (Train), 0.999075 (Val), 0.998070 (Test).

Top RNN Model:

- **LSTM_GMP** stands out with:
  - **Perfect accuracy** across all sets (Train/Val/Test = 1.00).
  - **Weighted F1 score of 1.00** for Train/Val/Test.
  - **PR AUC**: 0.999973 (Train), 0.999362 (Val), 0.998946 (Test).

Conclusion:

For this task, **FourgramCNN_GMP** and **LSTM_GMP** offer the best performance, with both models achieving perfect accuracy, weighted F1 scores, and strong PR AUC values across all datasets (train, validation, and test). The **FourgramCNN_GMP** model excels particularly in CNN-based approaches, while **LSTM_GMP** delivers impressive results with the RNN architecture.



Exercise 3

Repeat Exercise 2 of Part 4 (NLP with RNNs), now using a stacked CNN with n-gram filters (e.g., n = 2, 3, 4), residual connections, and a dense layer (the same at all word positions) with softmax at the top layer, implemented in PyTorch.

1. Tune the hyper-parameters (e.g., values of n, number of stacked convolutional layers) on the development subset of the dataset. Monitor the performance of your models on the development subset during training to decide how many epochs to use. You may optionally add a character-level CNN to produce word embeddings from characters, concatenating each resulting character-based word embedding with the corresponding pre-trained word embedding (e.g., obtained with Word2Vec).

2. Include experimental results of a baseline that tags each word with the most frequent tag it had in the training data; for words that were not encountered in the training data, the baseline should return the most frequent tag (over all words) of the training data. Also include experimental results of your best method from exercise 10 of Part 3 and exercise 2 of Part 4.

3. Otherwise, the contents of your report should be as in exercise 2 of Part 4, but now with information and results for the experiments of this exercise. You may optionally wish to try ensembles.

Data Download

In [ ]:

```
Downloading en_ewt-ud-train.conllu...
Downloaded en_ewt-ud-train.conllu
Downloading en_ewt-ud-dev.conllu...
Downloaded en_ewt-ud-dev.conllu
Downloading en_ewt-ud-test.conllu...
Downloaded en_ewt-ud-test.conllu
```

CNNs Architecture, Training and Evaluation

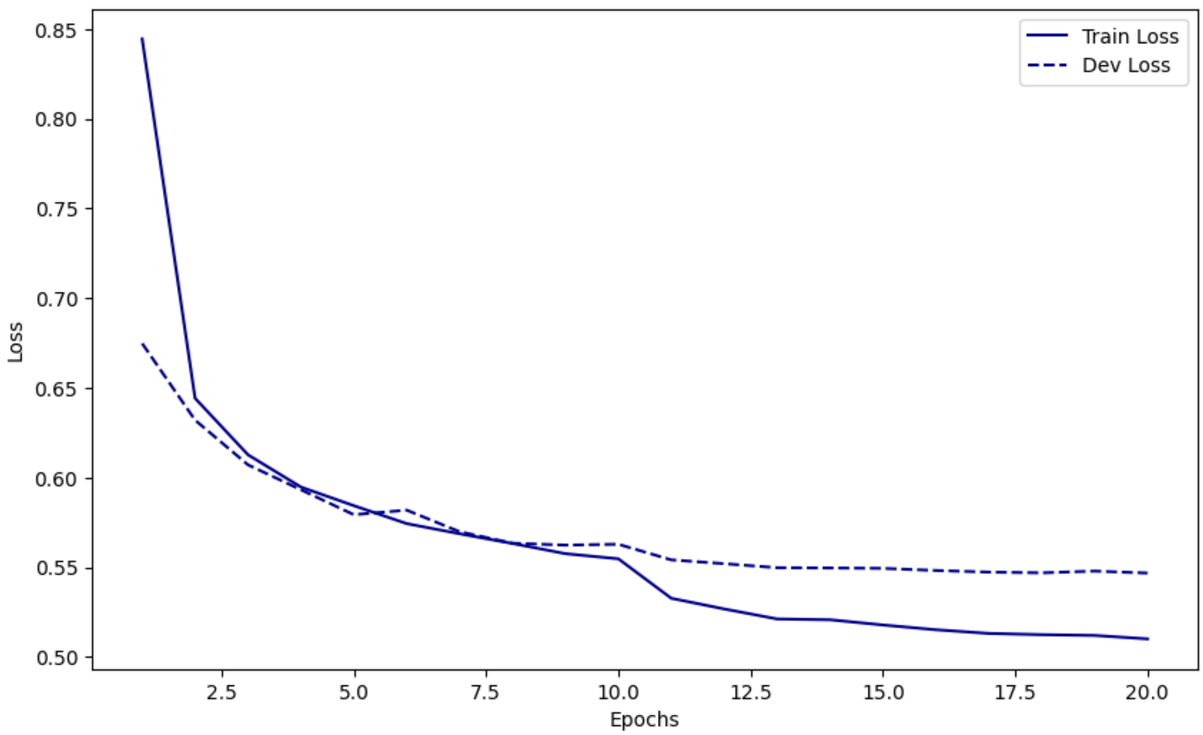
In [ ]:

```python
# Set device (you can change this to 'cuda' if you're using a GPU)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Define hyperparameters
input_dim = 300  # Example, change as needed
hidden_dim = 128  # Example, change as needed
output_dim = len(pos_tags)  # Number of POS tags
num_epochs = 20  # Number of epochs for training
```

In [ ]:

```python
# Shallow Model - Train
print("Training Shallow Model...")
shallow_model = train_and_evaluate_cnn(ShallowPOS_CNN, input_dim, hidden_dim
```

```
Training Shallow Model...
Epoch 1/20, Train Loss: 0.8446043420806205, Dev Loss: 0.6747861500820121
Epoch 2/20, Train Loss: 0.6442228303399625, Dev Loss: 0.6320350958888692
Epoch 3/20, Train Loss: 0.6127089372823973, Dev Loss: 0.6071083510952785
Epoch 4/20, Train Loss: 0.5947239903009476, Dev Loss: 0.5932203148092542
Epoch 5/20, Train Loss: 0.5843368412039764, Dev Loss: 0.5793033811009318
Epoch 6/20, Train Loss: 0.5743238648987612, Dev Loss: 0.5818347755380741
Epoch 7/20, Train Loss: 0.5686665016046418, Dev Loss: 0.569803922248066
Epoch 8/20, Train Loss: 0.5632487145138052, Dev Loss: 0.5633781988519176
Epoch 9/20, Train Loss: 0.557556352092021, Dev Loss: 0.5623411786063273
Epoch 10/20, Train Loss: 0.5547042744967067, Dev Loss: 0.5628197851933932
Epoch 11/20, Train Loss: 0.5326905837898052, Dev Loss: 0.5541053697429504
Epoch 12/20, Train Loss: 0.5267434754835153, Dev Loss: 0.5519880513873017
Epoch 13/20, Train Loss: 0.5211678501851328, Dev Loss: 0.5497303476608487
Epoch 14/20, Train Loss: 0.5207165053840255, Dev Loss: 0.549592876606119
Epoch 15/20, Train Loss: 0.517803917417488, Dev Loss: 0.549392014107011
Epoch 16/20, Train Loss: 0.5151517122312046, Dev Loss: 0.5481906846576466
Epoch 17/20, Train Loss: 0.5130948326593567, Dev Loss: 0.5472939990666277
Epoch 18/20, Train Loss: 0.5123970438579559, Dev Loss: 0.5468862764817431
Epoch 19/20, Train Loss: 0.5119764560935967, Dev Loss: 0.5478549765091492
Epoch 20/20, Train Loss: 0.5100530893705818, Dev Loss: 0.5467543665851865
```

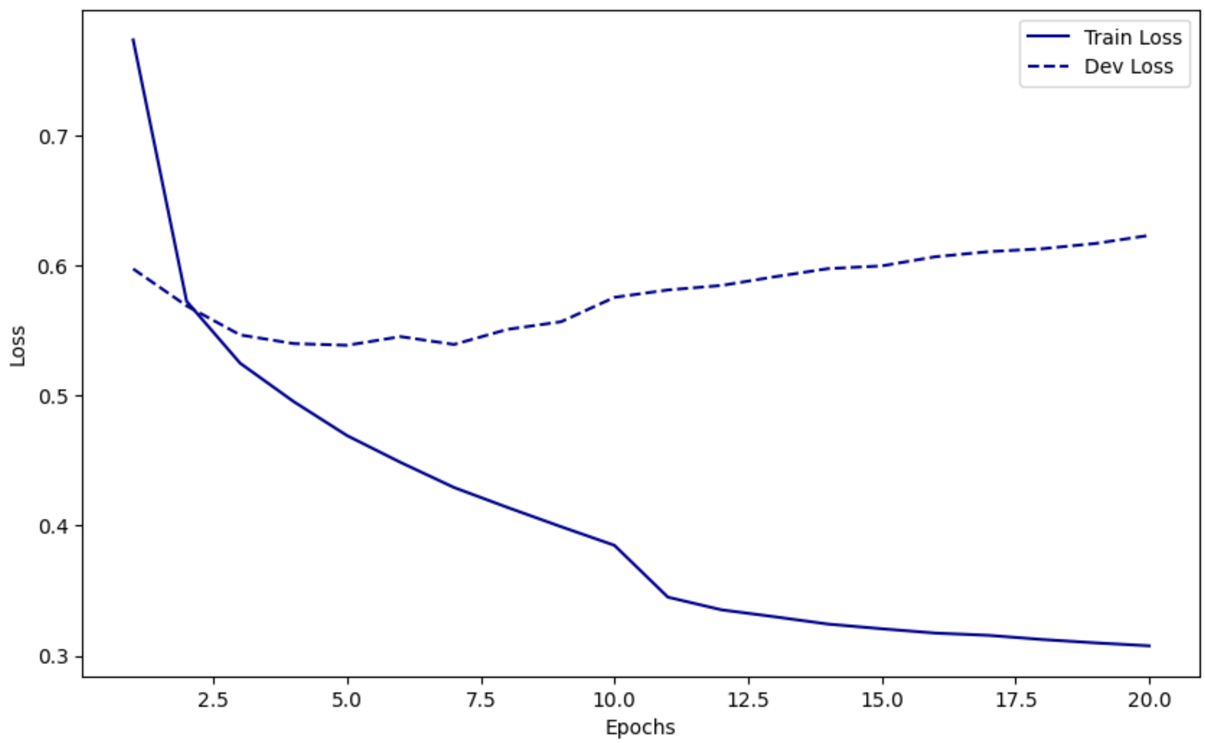


```
Classification Report for ShallowPOS_CNN:
              precision    recall  f1-score   support

         ADJ       0.90      0.90      0.90      1794
         ADP       0.85      0.73      0.78      2030
         ADV       0.93      0.84      0.89      1183
         AUX       0.97      0.95      0.96      1543
       CCONJ       0.98      0.27      0.43       736
         DET       0.94      0.74      0.83      1896
        INTJ       0.94      0.70      0.81       121
        NOUN       0.89      0.90      0.89      4123
         NUM       0.76      0.36      0.49       542
        PART       0.79      0.50      0.62       649
        PRON       0.97      0.97      0.97      2166
       PROPN       0.90      0.78      0.84      2076
       PUNCT       0.54      0.98      0.70      3096
       SCONJ       0.82      0.57      0.68       384
         SYM       0.98      0.59      0.74       109
        VERB       0.93      0.90      0.91      2606
           X       0.00      0.00      0.00        42
           _       0.97      0.76      0.85       354

    accuracy                           0.83     25450
   macro avg       0.84      0.69      0.74     25450
weighted avg       0.86      0.83      0.83     25450


AUC Scores for Each Class:
ADJ: 0.9885
ADP: 0.9793
ADV: 0.9927
AUX: 0.9982
CCONJ: 0.9545
DET: 0.9870
INTJ: 0.9768
NOUN: 0.9859
NUM: 0.9721
PART: 0.9811
PRON: 0.9991
PROPN: 0.9804
PUNCT: 0.9650
SCONJ: 0.9854
SYM: 0.9724
VERB: 0.9938
X: 0.8714
_: 0.9817

Macro-Averaged Precision: 0.8371382663093933
Macro-Averaged Recall: 0.6911115070293722
Macro-Averaged F1: 0.736742833415145
Macro-Averaged Precision-Recall AUC: 0.9758623703945345
```

In [ ]:

```
# Deep Model - Train
print("\nTraining Deep Model...")
deep_model = train_and_evaluate_cnn(DeepPOS_CNN, input_dim, hidden_dim, outp
```

```
Training Deep Model...
Epoch 1/20, Train Loss: 0.7736386407610809, Dev Loss: 0.597489666445811
Epoch 2/20, Train Loss: 0.5725546782448603, Dev Loss: 0.5690608834264272
Epoch 3/20, Train Loss: 0.5249782080236889, Dev Loss: 0.5467145647246736
Epoch 4/20, Train Loss: 0.49549732145573194, Dev Loss: 0.5400558091270595
Epoch 5/20, Train Loss: 0.4692995683131872, Dev Loss: 0.5387196674905624
Epoch 6/20, Train Loss: 0.4486944978530942, Dev Loss: 0.5454042985251075
Epoch 7/20, Train Loss: 0.42938716755633444, Dev Loss: 0.5393703370763544
Epoch 8/20, Train Loss: 0.4140540090302688, Dev Loss: 0.5509671523308096
Epoch 9/20, Train Loss: 0.3992549410668548, Dev Loss: 0.5568088631059293
Epoch 10/20, Train Loss: 0.3849147849516636, Dev Loss: 0.5755845048000341
Epoch 11/20, Train Loss: 0.3450388986435549, Dev Loss: 0.5812750504653257
Epoch 12/20, Train Loss: 0.33531700747249815, Dev Loss: 0.5847260800369999
Epoch 13/20, Train Loss: 0.3299435027371666, Dev Loss: 0.5914308030653119
Epoch 14/20, Train Loss: 0.32431398323375255, Dev Loss: 0.597701545459286
Epoch 15/20, Train Loss: 0.32072734135335945, Dev Loss: 0.5997743144220576
Epoch 16/20, Train Loss: 0.3174297206697455, Dev Loss: 0.6068145512488851
Epoch 17/20, Train Loss: 0.3156589540003478, Dev Loss: 0.6107434376812818
Epoch 18/20, Train Loss: 0.31248465366086303, Dev Loss: 0.6129652914173322
Epoch 19/20, Train Loss: 0.30989390276280393, Dev Loss: 0.6170460566467509
Epoch 20/20, Train Loss: 0.3076193625391579, Dev Loss: 0.6232517334378154
```

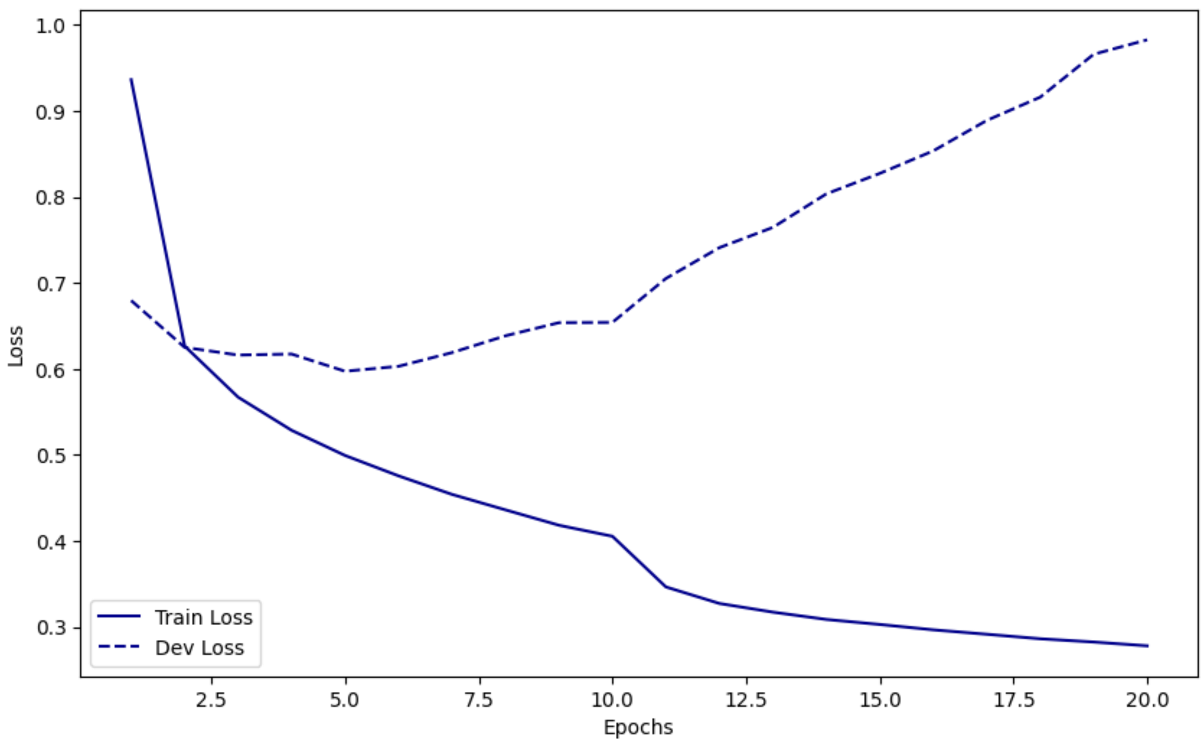


```
Classification Report for DeepPOS_CNN:
              precision    recall  f1-score   support

         ADJ       0.91      0.88      0.90      1794
         ADP       0.81      0.77      0.79      2030
         ADV       0.90      0.87      0.88      1183
         AUX       0.96      0.97      0.96      1543
       CCONJ       0.68      0.36      0.47       736
         DET       0.86      0.80      0.83      1896
        INTJ       0.93      0.76      0.84       121
        NOUN       0.90      0.89      0.90      4123
         NUM       0.62      0.53      0.57       542
        PART       0.61      0.66      0.64       649
        PRON       0.96      0.97      0.97      2166
       PROPN       0.89      0.80      0.84      2076
       PUNCT       0.61      0.85      0.71      3096
       SCONJ       0.77      0.71      0.74       384
         SYM       0.96      0.61      0.74       109
        VERB       0.92      0.91      0.92      2606
           X       0.00      0.00      0.00        42
           _       0.95      0.77      0.85       354

    accuracy                           0.84     25450
   macro avg       0.79      0.73      0.75     25450
weighted avg       0.84      0.84      0.84     25450


AUC Scores for Each Class:
ADJ: 0.9850
ADP: 0.9807
ADV: 0.9926
AUX: 0.9985
CCONJ: 0.9551
DET: 0.9881
INTJ: 0.9789
NOUN: 0.9830
NUM: 0.9705
PART: 0.9812
PRON: 0.9993
PROPN: 0.9732
PUNCT: 0.9652
SCONJ: 0.9873
SYM: 0.9727
VERB: 0.9923
X: 0.8404
_: 0.9836

Macro-Averaged Precision: 0.7919515829976682
Macro-Averaged Recall: 0.727741052539293
Macro-Averaged F1: 0.752212080440213
Macro-Averaged Precision-Recall AUC: 0.9737568122028613
```

In [ ]:

```
# Very Deep Model - Train
print("\nTraining Very Deep Model...")
very_deep_model = train_and_evaluate_cnn(VeryDeepPOS_CNN, input_dim, hidden_
```

```
Training Very Deep Model...
Epoch 1/20, Train Loss: 0.9364304318701948, Dev Loss: 0.6795115483583962
Epoch 2/20, Train Loss: 0.6265663460789822, Dev Loss: 0.6250142339701042
Epoch 3/20, Train Loss: 0.5670794998194937, Dev Loss: 0.6160516358333123
Epoch 4/20, Train Loss: 0.5285368002882116, Dev Loss: 0.6172162003086922
Epoch 5/20, Train Loss: 0.4992443193313956, Dev Loss: 0.597215403590286
Epoch 6/20, Train Loss: 0.4755320267048863, Dev Loss: 0.6029499950713681
Epoch 7/20, Train Loss: 0.45397160375068557, Dev Loss: 0.618910239677979
Epoch 8/20, Train Loss: 0.43597622876359843, Dev Loss: 0.6384463466349102
Epoch 9/20, Train Loss: 0.4180797252060824, Dev Loss: 0.6537254436273026
Epoch 10/20, Train Loss: 0.40524341181205925, Dev Loss: 0.6540592374807611
Epoch 11/20, Train Loss: 0.34633021131894187, Dev Loss: 0.7051294672683367
Epoch 12/20, Train Loss: 0.32709902633094284, Dev Loss: 0.7409660087566925
Epoch 13/20, Train Loss: 0.3170138965507508, Dev Loss: 0.7645145130710196
Epoch 14/20, Train Loss: 0.30852521546392103, Dev Loss: 0.8035311177186201
Epoch 15/20, Train Loss: 0.30266435934464236, Dev Loss: 0.8273660158901884
Epoch 16/20, Train Loss: 0.29644534039328946, Dev Loss: 0.8535530160094861
Epoch 17/20, Train Loss: 0.29128456215585286, Dev Loss: 0.888957850952495
Epoch 18/20, Train Loss: 0.2860324644254103, Dev Loss: 0.9160384632142863
Epoch 19/20, Train Loss: 0.2822558688560134, Dev Loss: 0.965959765073052
Epoch 20/20, Train Loss: 0.27782923753883565, Dev Loss: 0.982735942778432
```



Classification Report for VeryDeepPOS_CNN:

	precision	recall	f1-score	support
ADJ	0.89	0.88	0.89	1794
ADP	0.80	0.74	0.77	2030
ADV	0.89	0.87	0.88	1183
AUX	0.96	0.96	0.96	1543
CCONJ	0.59	0.36	0.45	736
DET	0.82	0.80	0.81	1896
INTJ	0.89	0.69	0.78	121
NOUN	0.89	0.88	0.89	4123
NUM	0.55	0.44	0.49	542
PART	0.59	0.63	0.61	649
PRON	0.97	0.97	0.97	2166
PROPN	0.89	0.79	0.84	2076
PUNCT	0.59	0.79	0.68	3096
SCONJ	0.78	0.67	0.72	384
SYM	0.93	0.61	0.74	109
VERB	0.90	0.91	0.91	2606
X	0.00	0.00	0.00	42
_	0.91	0.77	0.83	354
accuracy			0.82	25450
macro avg	0.77	0.71	0.73	25450
weighted avg	0.83	0.82	0.82	25450

```
AUC Scores for Each Class:
ADJ: 0.9730
ADP: 0.9756
ADV: 0.9867
AUX: 0.9975
CCONJ: 0.9435
DET: 0.9847
INTJ: 0.9577
NOUN: 0.9705
NUM: 0.9643
PART: 0.9754
PRON: 0.9986
PROPN: 0.9483
PUNCT: 0.9587
SCONJ: 0.9819
SYM: 0.9632
VERB: 0.9830
X: 0.8466
_: 0.9776

Macro-Averaged Precision: 0.768889222869066
Macro-Averaged Recall: 0.7088301159808167
Macro-Averaged F1: 0.7327654697761675
Macro-Averaged Precision-Recall AUC: 0.9659298061725151
```

Choice of Architectures for Shallow, Somewhat Deep, and Deep CNNs

1. Shallow CNN (ShallowPOS_CNN)

The **Shallow CNN** is designed to be a relatively simple model with only two convolutional layers. This architecture is useful when the input data does not require too much feature extraction complexity, or when computational efficiency is a priority.

Architecture Details:

- **Convolutional Layers**:
  - **Conv1**: The first convolutional layer applies 64 filters of size 3, with zero-padding to preserve sequence length. This helps capture local features from the input sequences.
  - **Conv2**: The second convolutional layer applies the same number of filters (64) with the same kernel size (3).
- **Max Pooling**:
  - A **max pooling** layer with a kernel size of 2 is applied after the convolutional layers. Pooling reduces the dimensionality of the feature maps, which helps prevent overfitting and reduces the computational load.
- **Dropout**:
  - A dropout layer with a probability of 0.3 is used after each convolutional layer to regularize the model and prevent overfitting.
- **Fully Connected Layer**:
  - The final output of the model is produced by a fully connected layer, which maps the learned features to the output space (POS tags).

Why this architecture?

- **Shallow**: The model only uses two convolutional layers, making it lightweight and computationally efficient.
- **Local Features**: It focuses on learning local features via small kernels.
- **Regularization**: The use of dropout helps to generalize the model despite its simplicity.

---

2. Somewhat Deep CNN (DeepPOS_CNN)

The **Somewhat Deep CNN** is a deeper model that uses two convolutional layers, followed by **global average pooling (GAP)** instead of max pooling. GAP helps reduce overfitting by summarizing feature maps in a compact way.

Architecture Details:

- **Convolutional Layers**:
  - **Conv1**: The first convolutional layer applies `hidden_dim` filters with a kernel size of 3, which allows the model to learn low-level features from the input data.
  - **Conv2**: The second convolutional layer uses the same number of filters (`hidden_dim`), deepening the feature extraction process.
- **Global Average Pooling**:
  - Instead of using max pooling, this model applies **global average pooling (GAP)** after each convolutional layer. GAP reduces each feature map to a single value, which encourages the model to focus on global patterns rather than local ones.
- **Dropout**:
  - Dropout (0.3) is applied after the GAP layers and before the fully connected layer to further reduce overfitting.
- **Fully Connected Layer**:
  - A fully connected layer is used to produce the final predictions, reducing the feature maps to the target output dimension.

Why this architecture?

- **Deeper Learning**: The addition of global average pooling allows the model to capture more abstract features and learn global patterns from the data.
- **GAP Regularization**: GAP prevents the model from overfitting by focusing on the global structure of the feature maps instead of local patterns.

---

3. Very Deep CNN (VeryDeepPOS_CNN)

The **Very Deep CNN** is the most complex model with five convolutional layers, followed by **global average pooling**. This architecture aims to extract hierarchical and abstract features from the input sequence by passing it through multiple layers of convolutions and pooling.

Architecture Details:

- **Convolutional Layers**:
  - This model uses **five convolutional layers** with a kernel size of 3 and padding of 1, which enables the model to learn increasingly abstract features at different levels of depth.
  - Each convolutional layer uses `hidden_dim` filters, which gradually capture more complex patterns in the data.
- **Global Average Pooling**:

- After each convolutional layer, **global average pooling (GAP)** is applied. This operation reduces the sequence length to 1 for each feature map, effectively condensing the learned features into a more compact representation.

- **Dropout**:

  - Dropout (0.3) is applied after the final pooling operation to help reduce overfitting and improve generalization.

- **Fully Connected Layer**:

  - A fully connected output layer is used to map the learned features to the target output (POS tags).

Why this architecture?

- **Very Deep Learning**: With five convolutional layers, this model is able to extract highly abstract and hierarchical features from the input data.
- **Hierarchical Feature Extraction**: By stacking several layers, the model can detect progressively complex patterns from local to global representations.
- **Robust Generalization**: The use of GAP and dropout ensures that the model generalizes well despite its depth, preventing overfitting.

Conclusion:

- **Shallow CNN**: Simple and efficient, focuses on local features with fewer convolutional layers and max pooling.
- **Somewhat Deep CNN**: Deeper than the shallow model, using global average pooling and dropout for regularization, capable of learning more abstract features.
- **Very Deep CNN**: A highly expressive model with five convolutional layers, ideal for capturing complex hierarchical features, balanced with global average pooling and dropout for regularization.

Comparison Board

Model	Accuracy	Macro avg F1-score	Weighted avg F1-score	Strengths	Weaknesses
ShallowPOS_MLP	0.83	0.74	0.83	High performance on **PRON (0.97 precision)**, **AUX (0.96 recall)**, and **VERB (0.91**	Struggles with **X (f1 = 0.00)** and **CCONJ (0.45 f1)**.

Model	Accuracy	Macro avg F1-score	Weighted avg F1-score	Strengths	Weaknesses
				f1-score).	
DeepPOS_MLP	0.83	0.74	0.83	Improved recall for **AUX (0.94)** and **PRON (0.97)**, maintains strong performance across categories.	Struggles with rare categories like **X**.
VeryDeepPOS_MLP	0.82	0.74	0.82	Strong performance on **AUX** and **PRON**, but slightly lower overall performance compared to other MLP models.	Some decrease in performance, especially in **CCONJ (0.43 f1)** and **NUM (0.56 f1)**.
ShallowPOS_BiGRU	0.84	0.75	0.84	High performance on **PRON (0.97 precision)**, **AUX (0.97 recall)**, and **VERB (0.92 f1-score)**.	Struggles with **X (f1 = 0.04)** and **CCONJ (0.45 f1)**.
DeepPOS_BiGRU	0.84	0.75	0.84	High performance on **AUX (0.97)** and **PRON (0.97)**.	Struggles with **X** and **CCONJ (0.45 f1)**.
VeryDeepPOS_BiGRU	0.84	0.76	0.84	Strong performance on **AUX (0.97)**, **PRON (0.98)**, and **VERB (0.92)** with a good balance across categories.	Some degradation in performance for **CCONJ (0.46 f1)** and **NUM (0.57 f1)**, but more consistent than previous models.
ShallowPOS_CNN	0.83	0.74	0.83	High performance on **AUX (0.96 recall)** and **PRON (0.97 precision)**,	Struggles with **X (f1 = 0.00)** and **CCONJ (0.43 f1)**.

Model	Accuracy	Macro avg F1-score	Weighted avg F1-score	Strengths	Weaknesses
				PUNCT (0.99 recall).	
DeepPOS_CNN	0.84	0.75	0.84	Good performance across most categories, especially **AUX (0.97)** and **VERB (0.92)**.	Struggles with **X (f1 = 0.00)** and **CCONJ (0.47 f1)**.
VeryDeepPOS_CNN	0.82	0.73	0.82	Strong performance on **AUX (0.96)** and **PRON (0.97)**.	Some struggles with **CCONJ (0.45 f1)** and **NUM (0.49 f1)**.
Baseline Tagger	0.86	0.80	0.86	High accuracy and strong performance on **CCONJ (0.99 precision)** and **PUNCT (0.99 precision)**.	Struggles with **X (f1 = 0.00)** and **PROPN (0.66 f1)**.

Exercise 3 (Corrected)

Data Parsing and Preprocessing (Corrected)

Now instead of context windows we will use sentences.

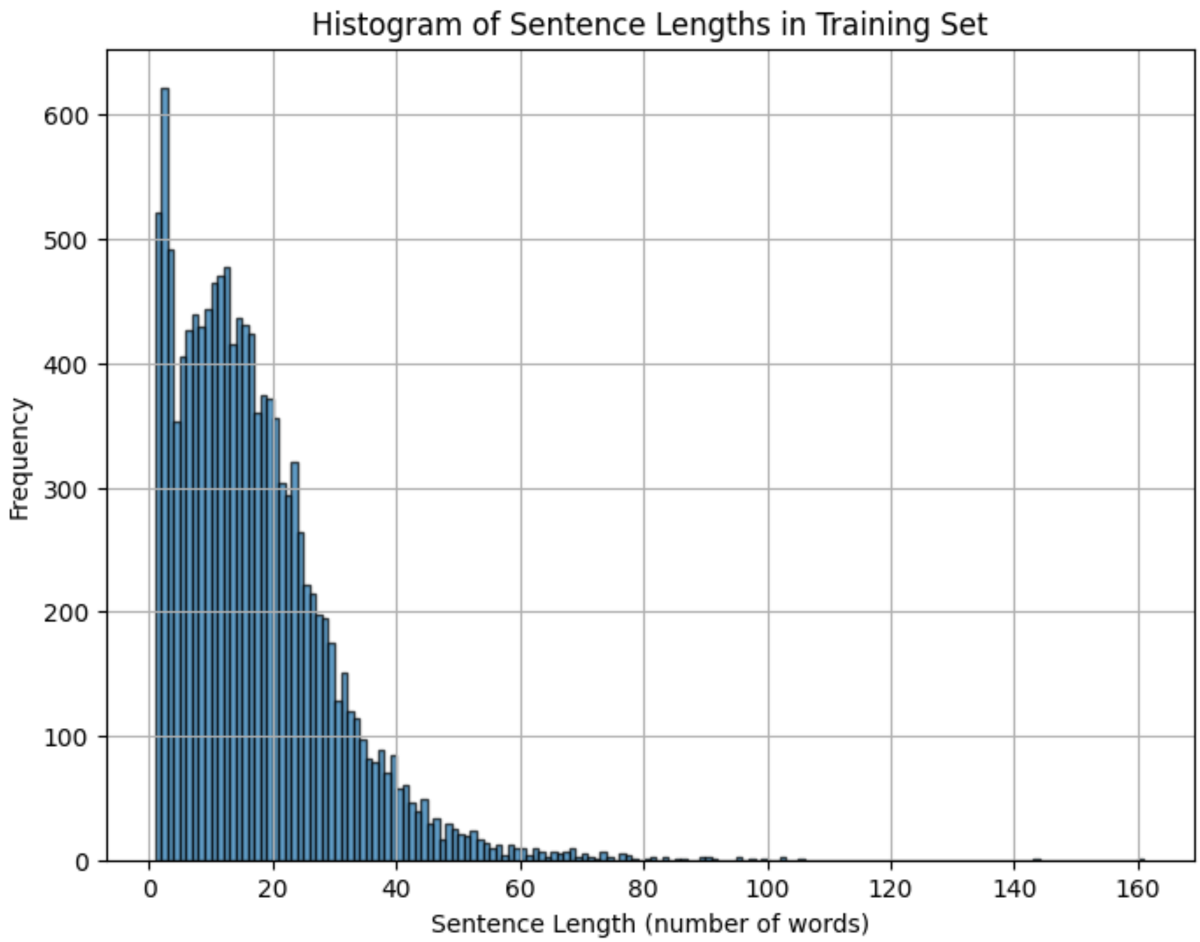
In [ ]:
```
train_sentences[5]
```

Out[ ]:
```
[('The', 'DET'),
 ('third', 'ADJ'),
 ('was', 'AUX'),
 ('being', 'AUX'),
 ('run', 'VERB'),
 ('by', 'ADP'),
 ('the', 'DET'),
 ('head', 'NOUN'),
 ('of', 'ADP'),
 ('an', 'DET'),
 ('investment', 'NOUN'),
 ('firm', 'NOUN'),
 ('.', 'PUNCT')]
```

In [ ]:
```
# Calculate the length of each sentence (in terms of the number of words)
```

```
sentence_lengths = [len(sentence) for sentence in train_sentences]

print("Mean train sentence length: ", np.mean(sentence_lengths))

plt.figure(figsize=(8, 6))
plt.hist(sentence_lengths, bins=range(1, max(sentence_lengths) + 2), edgecol
plt.title("Histogram of Sentence Lengths in Training Set")
plt.xlabel("Sentence Length (number of words)")
plt.ylabel("Frequency")
plt.grid(True)
plt.show()
```

```
Mean train sentence length:  16.520248724489797
```



After extracting the embeddings from the sentences and padding or truncating them to the desired `max_length`, we will convert them to `torch.tensors` and later transform them into `TensorDatasets` and `DataLoaders`

In [ ]:
```
# Iterate through the first batch of the train_loader to check the dimension
for inputs, labels in train_loader:
    print("Inputs shape:", inputs.shape)  # Check the input (embeddings) sha
    print("Labels shape:", labels.shape)  # Check the labels shape
    break  # Stop after the first batch
```

```
Inputs shape: torch.Size([64, 50, 300])
Labels shape: torch.Size([64, 50])
```

Input shape explained:

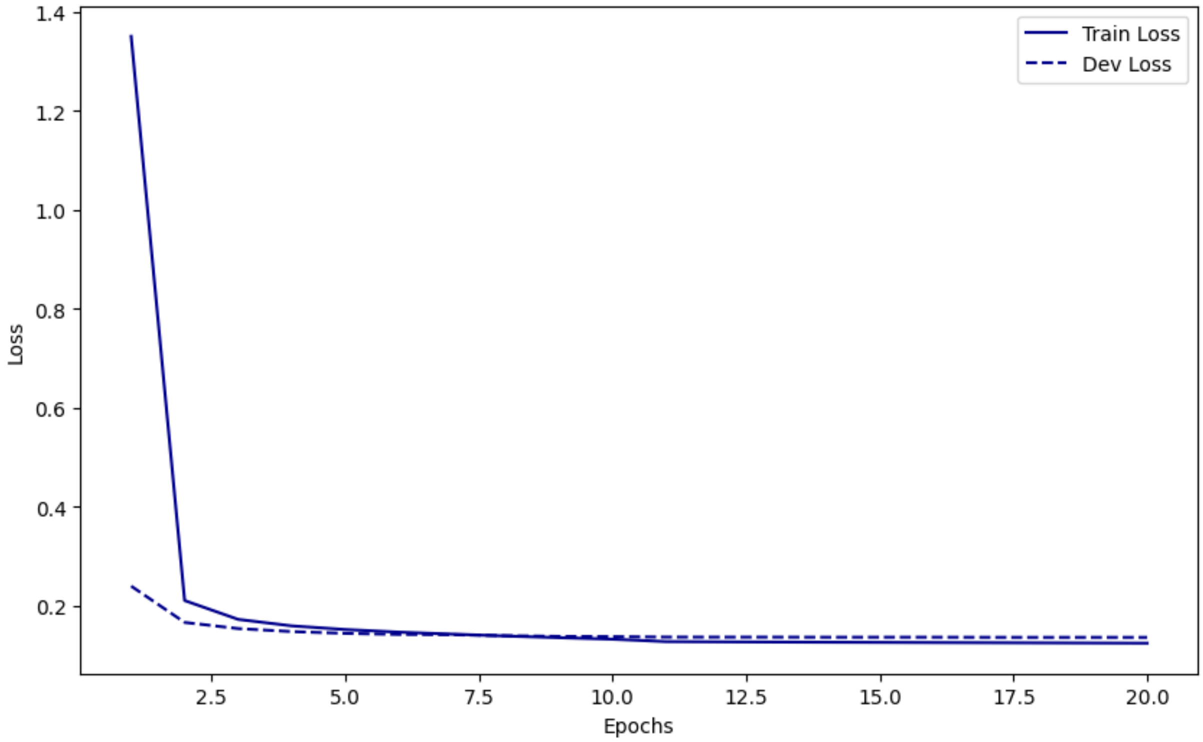
- 64: Batch size
- 50: sequence length
- 300: embedding dimension

Labels shape explained:

- 64: Batch size
- 50: sequence length

In [ ]:
```
trained_model = train_and_evaluate_cnn(
    model_class=POSCNN,
    embedding_dim=300,        # Word2Vec embeddings are 300-dimensional
    num_classes=len(pos_tags),  # Number of unique POS tags
    num_epochs=20             # Number of epochs for training
)
```

```
Epoch 1/20, Train Loss: 1.3503, Dev Loss: 0.2396
Epoch 2/20, Train Loss: 0.2105, Dev Loss: 0.1663
Epoch 3/20, Train Loss: 0.1724, Dev Loss: 0.1538
Epoch 4/20, Train Loss: 0.1593, Dev Loss: 0.1478
Epoch 5/20, Train Loss: 0.1519, Dev Loss: 0.1442
Epoch 6/20, Train Loss: 0.1466, Dev Loss: 0.1420
Epoch 7/20, Train Loss: 0.1426, Dev Loss: 0.1413
Epoch 8/20, Train Loss: 0.1388, Dev Loss: 0.1394
Epoch 9/20, Train Loss: 0.1354, Dev Loss: 0.1381
Epoch 10/20, Train Loss: 0.1324, Dev Loss: 0.1376
Epoch 11/20, Train Loss: 0.1275, Dev Loss: 0.1367
Epoch 12/20, Train Loss: 0.1270, Dev Loss: 0.1366
Epoch 13/20, Train Loss: 0.1266, Dev Loss: 0.1366
Epoch 14/20, Train Loss: 0.1263, Dev Loss: 0.1363
Epoch 15/20, Train Loss: 0.1259, Dev Loss: 0.1364
Epoch 16/20, Train Loss: 0.1256, Dev Loss: 0.1364
Epoch 17/20, Train Loss: 0.1253, Dev Loss: 0.1361
Epoch 18/20, Train Loss: 0.1250, Dev Loss: 0.1360
Epoch 19/20, Train Loss: 0.1246, Dev Loss: 0.1361
Epoch 20/20, Train Loss: 0.1243, Dev Loss: 0.1360
```




```
Classification Report for POSCNN:
              precision    recall  f1-score   support

          -1       0.00      0.00      0.00         0
         ADJ       0.91      0.89      0.90      1781
         ADP       0.85      0.85      0.85      2009
         ADV       0.88      0.89      0.88      1174
         AUX       0.97      0.97      0.97      1531
       CCONJ       0.64      0.48      0.55       734
         DET       0.93      0.93      0.93      1881
        INTJ       0.91      0.79      0.85       121
        NOUN       0.91      0.90      0.90      4095
         NUM       0.75      0.49      0.59       538
        PART       0.86      0.86      0.86       642
        PRON       0.98      0.98      0.98      2145
       PROPN       0.90      0.80      0.84      2066
       PUNCT       0.74      0.80      0.77      3081
       SCONJ       0.91      0.78      0.84       378
         SYM       0.89      0.61      0.73       108
        VERB       0.93      0.93      0.93      2586
           X       0.17      0.02      0.04        42
           _       0.85      0.82      0.83       347

    accuracy                           0.86     25259
   macro avg       0.79      0.73      0.75     25259
weighted avg       0.88      0.86      0.87     25259

/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_ranking.py:379: Und
efinedMetricWarning: Only one class is present in y_true. ROC AUC score is n
ot defined in that case.
  warnings.warn(
```

```
AUC Scores for each class:
ADJ: 0.9905
ADP: 0.9905
ADV: 0.9937
AUX: 0.9984
CCONJ: 0.9777
DET: 0.9969
INTJ: 0.9908
NOUN: 0.9875
NUM: 0.9480
PART: 0.9965
PRON: 0.9995
PROPN: 0.9756
PUNCT: 0.9766
SCONJ: 0.9943
SYM: 0.9531
VERB: 0.9957
X: 0.8609
_: 0.9910

Macro-Averaged Precision: 0.7887945702424318
Macro-Averaged Recall: 0.7791115085565101
Macro-Averaged F1: 0.7507063245541478
Macro-Averaged Precision-Recall AUC: 0.9787210696753399
```