Recurrent Neural Networks
Homework #1

1. Introduction

This assignment focuses on classifying short texts into two categories: AI-generated and human-written.

In this project, I started by implementing a basic LSTM classification model, then extended it with Auto-LSTM (encoder-decoder) and CNN-LSTM architectures.

I used the BERT tokenizer to preprocess the input texts, and finally compared the performance across different models. The main goals of this project are:

- Implement a basic LSTM-based classification model.
- Extend the architecture with Auto-LSTM (Encoder-Decoder) and CNN-LSTM.
- Apply BERT tokenizer for input text preprocessing.
- Analyze and compare the performance of different models.

2. Models and Implementation

2-0. Data Preprocessing

1. Read AI_Human.csv dataset
2. Clean invalid or empty content
3. Sample 50,000 examples from both AI and Human categories
4. Use bert-base-uncased tokenizer for subword tokenization
5. Analyze token length distribution and set max_len = 128
6. Spilt the dataset into validation, test and train

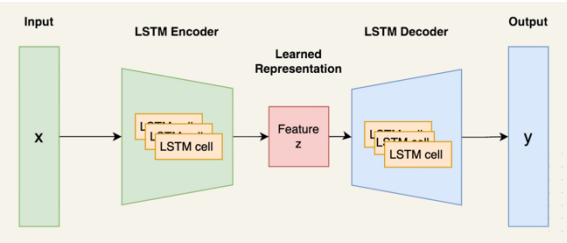
2-1. LSTM Classifier

- Embedding input
- LSTM layer (hidden dim = 256)
- Fully connected output layer for classification

LSTM Classifier is a classic recurrent neural network model that processes the input sequence step by step and learns long-range dependencies through hidden states. It is simple and effective for basic text classification tasks.

2-2. Auto-LSTM Classifier

- Embedding input
- Encoder LSTM to generate hidden state



- Decoder LSTM to reconstruct the embedded vector (reconstruction branch)
- Classifier head based on the encoder's hidden state

Auto-LSTM Classifier extends the LSTM by adding an encoder-decoder structure. The encoder summarizes the input sequence into a hidden representation, and the decoder attempts to reconstruct the original embedding. Although reconstruction is not used during training here, the architecture enhances representational learning.

2-3. CNN-LSTM Classifier

- Embedding input
- 1D Convolution (Conv1D) for extracting local n-gram features
- LSTM processes the CNN output sequence
- Dropout and classifier layer

CNN-LSTM Classifier combines the strength of convolutional layers and LSTM. A 1D convolution is first applied over the embedded input to capture local n-gram features, followed by an LSTM to capture the temporal relationships. This hybrid model is effective when both local patterns and sequential context matter.

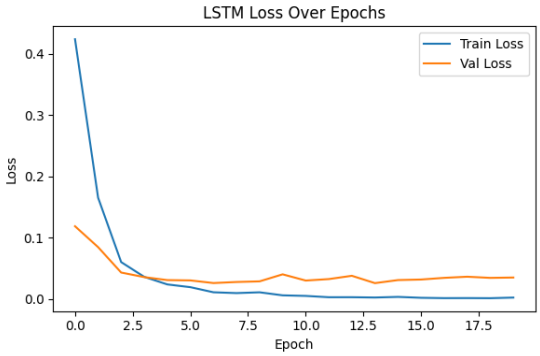
4. Training and Evaluation Strategy

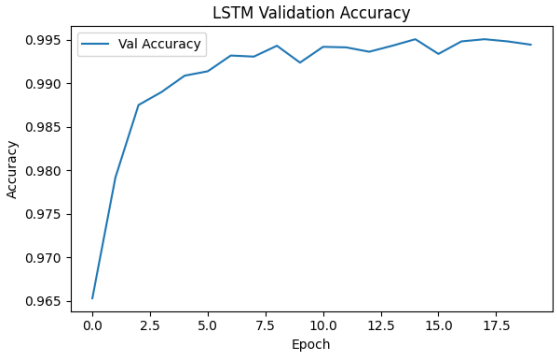
All models share the same training pipeline for a fair comparison. I used the following strategy:

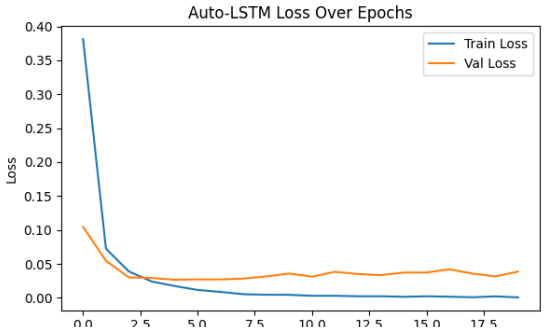
- **Loss Function:** BCELoss was used for binary classification
- **Optimizer:** All models used the Adam optimizer with default settings.
- **Epochs:** Each model was trained for up to 20 epochs.
- **Tokenizer:** bert-base-uncased tokenizer was used for subword tokenization.
- **Input Length:** The maximum sequence length was set to 512 tokens, and inputs were truncated accordingly.
- **Batching:** A custom collate_fn was implemented to handle padding and length sorting. Inputs were batched using PyTorch's DataLoader with a batch size of 32.

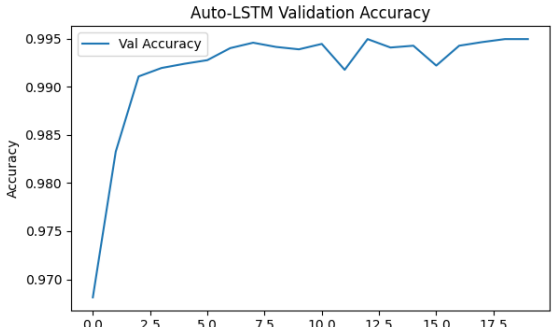
- **Packed Sequences:** pack_padded_sequence() was used before feeding input into the LSTM to improve computational efficiency by skipping padding tokens.
- **Embedding:** All models used nn.Embedding with padding index set based on the BERT tokenizer.
- **Validation Loop:** After each training epoch, validation loss and accuracy were calculated using the held-out validation set.
- **Evaluation:** After training, the evaluate_model() function was used to compute test loss, test accuracy, and the confusion matrix. Classification reports were printed to analyze precision, recall, and F1-score.
- **Visualization:** Training/validation loss and accuracy curves were plotted for each model. Confusion matrices were visualized using Seaborn's heatmap.

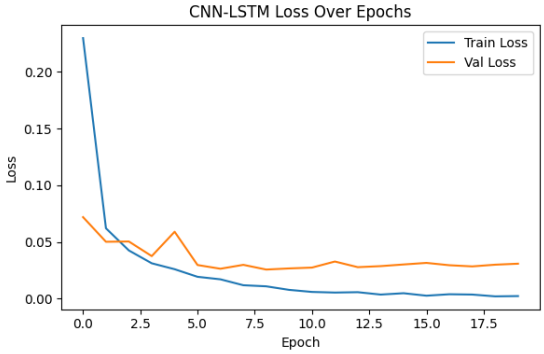
5. Model Performance Comparison

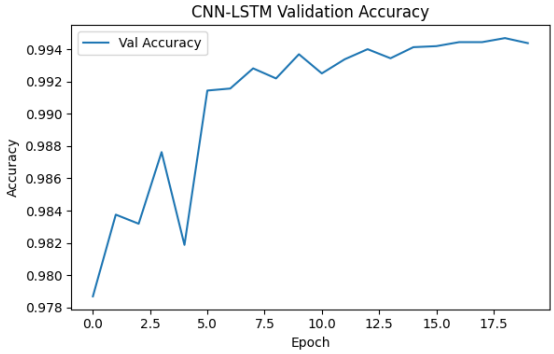












Model	Test Accuracy	Test Loss
LSTM	99.42	0.0368
Auto-LSTM	**99.48**	0.0333
CNN-LSTM	99.41	**0.0300**

- I think all three models achieved strong performance. LSTM and Auto-LSTM showed slightly higher accuracy, while CNN-LSTM achieved the lowest test loss.
- **BERT Tokenizer** effectively leveraged pre-trained subword segmentation.
- The training curves show that all three models converged well:
  - LSTM converged quickly and maintained stable validation accuracy above 99.5%.
  - Auto-LSTM also reached high accuracy, but with slight fluctuations during validation.
  - CNN-LSTM showed more variation in validation loss, especially in early epochs, but eventually achieved similar accuracy.
  - Overall, LSTM was the most stable, while Auto-LSTM and CNN-LSTM learned effectively with different dynamics. Among them, Auto-LSTM achieved the best performance in terms of test accuracy.

6. Conclusion

Through this assignment, I not only implemented a basic LSTM model, but also explored more advanced architectures like Auto-LSTM and CNN-LSTM. I learned how different structures affect training stability and performance, and how to apply tools like BERT tokenizer and PackedSequence to handle variable-length input efficiently.

Overall, I found that all three models performed very well, but they showed different learning behaviors. Auto-LSTM achieved the highest test accuracy, making it the best-performing model overall. LSTM was the most stable during training, while CNN-LSTM captured local patterns effectively but was more sensitive in early epochs.

This project helped me understand the strengths and the different in LSTM-based models, and I gained a lot of hands-on experience in model design, training, and evaluation.