NLP Assignment 2 Report

Building ‘mini-Copilot’: Sequential code generation with recurrent architectures

Name: Diganta Mandal
Roll: 22CS30062
Course: Natural Language Processing

Introduction

This report presents the design and implementation of a token-level code completion system “*mini-Copilot”* built using recurrent neural architectures.The report discusses the **model architectures**, **design decisions**, and **training configurations**, followed by a detailed analysis of the **evaluation metrics** (loss, perplexity, and top-5 accuracy) and **training curves**.

Model Architectures

The project implements two sequential models : **RNNModel** and **LSTMModel**, for token-level code generation. Both models follow a similar three-stage architecture:

1. **Embedding Layer:**
   Converts discrete token indices into continuous vector representations of dimension *embedding_dim (256)*. The embedding layer uses a “`padding_idx = 0`” to ensure that padding tokens do not contribute to learning.

2. **Recurrent Layer:**

   ○ In the **RNNModel**, a multi-layer vanilla RNN (`nn.RNN`) is used with *hidden_dim = 512*, *num_layers = 2*, and *tanh* as the non-linearity.

   ○ In the **LSTMModel**, the recurrent backbone is replaced with a multi-layer LSTM (`nn.LSTM`) having identical hidden size, number of layers, and dropout configuration.
   Both recurrent layers process sequences in `(seq_len, batch, feature)` format and output contextualized hidden representations for each token.

3. **Dropout and Output Layer:**
   A dropout layer with a rate of *0.3* is applied to regularize the model and prevent overfitting. The final linear layer projects the hidden representations back to the vocabulary space, generating logits for each token position.

The overall forward pass can be summarized as:

Input Tokens → Embedding → RNN/LSTM → Dropout → Linear (Vocab Projection)

Design Choices

Embedding Layer
We used an embedding dimension of **256**, balancing representational power with computational efficiency. Instead of using pre-trained embeddings like Word2Vec or GloVe, we chose to t**rain embeddings from scratch**, as our dataset consists of code tokens rather than natural language words. Code tokens often differ syntactically and semantically from standard language tokens, so domain-specific embeddings learned during training are more likely to capture relevant structure (e.g., variable patterns, syntax, and operator co-occurrence). Random initialization allows the model to **learn representations** best suited for the code completion task.

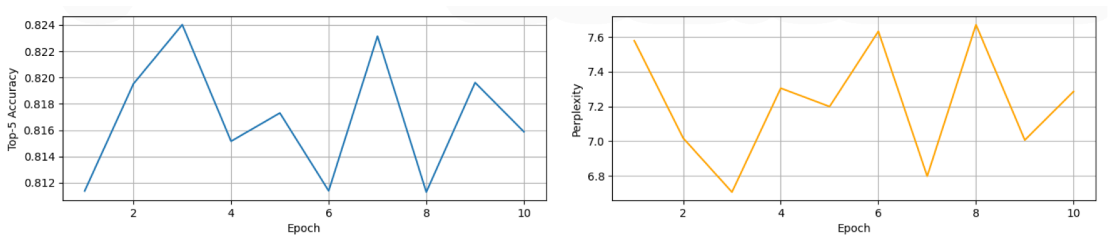
Recurrent Stack
We implemented a **2-layer** recurrent architecture with **512** hidden units per layer. This depth enables the network to capture both short and long-range dependencies between tokens in code sequences.
For activation, we used **tanh**, which is the standard nonlinearity for RNN and LSTM cells. It effectively handles both **positive and negative activations** while **preventing gradient explosion** to some extent. The choice of LSTM over a simple RNN helps mitigate the vanishing gradient problem and preserves long-term context crucial for modeling nested code structures and indentation patterns in code completion.

Regularization
To prevent overfitting, we applied a Dropout layer between **recurrent layers** and before the **output layer** with a dropout probability of **0.3**. This value provides a good balance, strong enough to regularize without causing underfitting. Dropout ensures that the model generalizes better to unseen code by preventing it from **memorizing token sequences** in the training data.

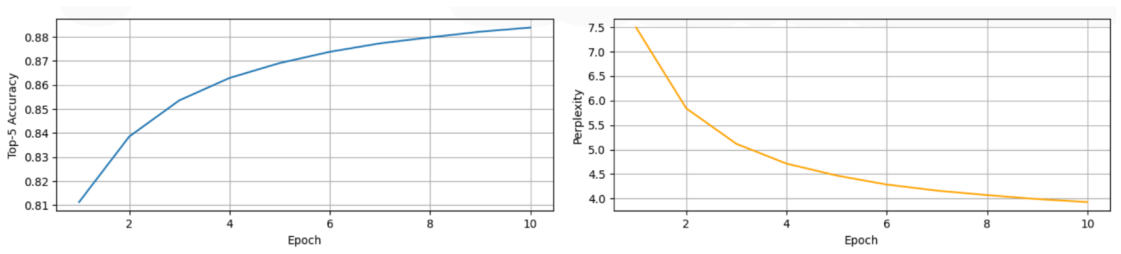
Training Duration
We trained both models for **10 epochs** instead of waiting for full convergence. This decision was made as a trade-off between **computational cost** and **performance**. Each epoch required approximately **15–20 minutes** for the RNN and around **30 minutes** for the LSTM, making full convergence training (typically requiring 30–50 epochs) computationally expensive and time-intensive.
Given that the primary goal of this assignment was to **compare** model architectures rather than achieve maximum possible accuracy, training for 10 epochs was sufficient to observe meaningful trends in loss reduction, learning stability, and relative performance between RNN and LSTM models. Additionally, early stopping helps avoid overfitting and provides a realistic assessment of each model's learning efficiency within practical runtime constraints.

Results

Baseline Comparison



RNN



LSTM

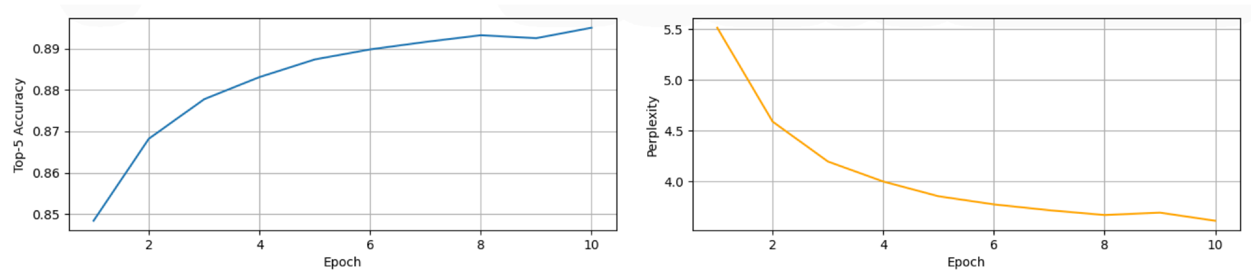
Model	Test Top-5 Accuracy	Perplexity	Parameter Counts	Training Time
RNN	82.49%	6.63	8,611,090	2h 10m
LSTM	88.48%	3.89	11,369,746	3h 4m

Best Performing Model:
The LSTM outperforms the RNN in both metrics: it achieves a higher Top-5 accuracy (**88.48%** vs. **82.49%**) and a **lower perplexity** (**3.89** vs. **6.63**). Lower perplexity indicates the model predicts the next word more confidently, and higher Top-5 accuracy shows better predictive performance across the vocabulary.

Analysis of Parameter Count vs. Performance:
While the LSTM has more parameters (≈11.37M vs. ≈8.61M), the improved performance is **not solely due to the number of parameters**. The architectural advantages of LSTMs, like **gating mechanisms** that capture **long-term dependencies**, allow them to model sequences more effectively than standard RNNs, even for the same task.

Conclusion:
The LSTM's superior performance arises from its architecture rather than just the increase in parameters, demonstrating that **model design is as important, if not more, than parameter count** for sequential prediction tasks.

Residual Connection Results



<u>Residual LSTM</u>

Since LSTM outperformed RNN, we designed a LSTM with skip connections between its first and second recurrent layer. The results compared with vanilla LSTM are as follows:

Model	Test Top-5 Accuracy	Perplexity	Parameter Counts	Training Time
Vanilla LSTM	88.48%	3.89	11,369,746	3h 4m
Residual LSTM	89.59%	3.58	11,369,746	3h 8m

Percentage Improvement compared to Vanilla LSTM = (89.59 - 88.48)/88.48 * 100
= **1.254%**

Accuracy Improvement:
The **Residual LSTM** achieved a Test Top-5 Accuracy of **89.59%**, compared to **88.48%** for the Vanilla LSTM, resulting in a **1.25%** relative improvement. This indicates that residual connections help the model capture deeper temporal dependencies and mitigate the vanishing gradient problem commonly encountered in standard LSTMs.

Reduction in Perplexity:
The perplexity decreased from **3.89** to **3.58**, reflecting a **7.97% improvement**. A lower perplexity signifies that the Residual LSTM makes more confident and consistent predictions

Model Complexity:
Both models have the same number of parameters (**11,369,746**), demonstrating that the performance gains were achieved **without increasing model size**. The improvement is purely due to architectural enhancement rather than additional capacity.

Best Model URL: https://www.kaggle.com/models/digsm003/best_model_22cs30062