Project Report: Text-to-Python Code Generation Using Seq2Sec Models

Submitted By:

Amit Kumar Roy

Roll: BSSE 1314

IIT, DU

Submitted to:

Mridha Md. Nafis Fuad

Lecturer

Institute of Information Technology

University of Dhaka

Submission Date: 13 February, 2026

Executive Summary

This project implements and evaluates four sequence-to-sequence (Seq2Seq) architectures for automatic Python code generation from natural language descriptions. Using the CodeSearchNet Python dataset, I trained and compared Vanilla RNN, LSTM, LSTM with Attention, and Transformer models.

[Source Code link: https://www.kaggle.com/code/amitroy13/text-to-code]

1. Introduction

1.1 Background

Automatic code generation from natural language descriptions is a challenging task in Natural Language Processing (NLP) and Software Engineering. The ability to translate human-readable descriptions into executable code can significantly improve developer productivity and make programming more accessible.

1.2 Project Objectives

The primary objectives of this project are:

1. **Implement Four Seq2Seq Architectures:** Vanilla RNN Seq2Seq (Baseline), LSTM Seq2Seq (Improved memory), LSTM with Bahdanau Attention (Enhanced alignment), Transformer (State-of-the-art)
2. **Comprehensive Evaluation:** Compare models using BLEU score, token accuracy, and exact match metrics, analyze performance across different sequence lengths, visualize training and validation loss curves.

1.3 Dataset

- **Source:** Hugging Face Datasets (**https://huggingface.co/datasets/Nan-Do/code-search-net-python**)
- **Task:** Docstring → Python Code Generation
- **Sample Size:** 10,000 carefully filtered examples
- **Filtering Criteria:** Docstring length ≤ 50 tokens and code length ≤ 80 tokens
- **Train/Val/Test Split:** 60% / 20% / 20% (6,000 / 2,000 / 2,000)

2 Model Architectures

2.1 Model 1: Vanilla RNN Seq2Seq

Architecture: This baseline uses a single-layer RNN for both the encoder and decoder. It features a 128-dimension embedding and a 256-dimension hidden state. The decoder utilizes a 0.5 teacher forcing ratio to assist with convergence during training.

Limitations: The model suffers from the vanishing gradient problem, making it difficult to learn long sequences..

2.2 Model 2: LSTM Seq2Seq

Architecture: This model replaces the standard RNN with a single-layer LSTM, maintaining the 128/256 dimension split but adding a 0.5 dropout for regularization. The decoder includes both cell state and hidden state memory.

Improvements: The use of gating mechanisms (input, forget, and output gates) allows for better long-term dependency handling. The dedicated cell state provides the additional memory needed to outperform the vanilla RNN.

2.3 Model 3: LSTM with Bahdanau Attention

Architecture: The encoder is upgraded to a Bidirectional LSTM to capture context from both directions. It integrates Bahdanau (Additive) Attention, which uses an alignment scoring function to compute a dynamic context vector for the decoder.

Key Innovation: Instead of a fixed bottleneck, this model allows for selective focus on specific source tokens. This creates a dynamic alignment between the docstring and the code based on weighted attention scores.

2.4 Model 4: Transformer

Architecture: This approach ditches recurrence for a 3-layer Transformer with 8-head multi-head attention and a 512-dimension feedforward layer. It uses sinusoidal positional encodings and masked self-attention in the decoder.

Advantages: The model enables parallel processing, removing the sequential bottleneck of RNNs. It scales much better to long sequences by using full self-attention.

3 Training Configuration

Setup: Here set the embedding dimension to 128 and the hidden/cell state dimension to 256. The models are trained over 10 epochs with a batch size of 64 samples. To prevent exploding gradients, a gradient clipping threshold of 1.0 is applied.

Process: The learning rate is set to 0.001 for the RNN, LSTM, and Attention models, while a lower rate of 0.0001 is used for the Transformer to ensure stability. Training is hardware-accelerated using CUDA where available.

Methodology: A Teacher Forcing Ratio of 0.5 is implemented, meaning the model uses the actual ground-truth token as the next input 50% of the time to keep training focused. We perform validation at the end of every epoch to monitor for overfitting. The final model selection for testing is based on the version that achieves the best (lowest) validation loss.

4. Results and Analysis

4.1 Quantitative Results

Model	BLEU Score	Token Accuracy (%)	Exact Match (%)	Train Loss	Val Loss
Vanilla RNN	0.07	17.07	0.0	~4.5	~4.8
LSTM	0.05	17.05	0.0	~4.3	~4.6
LSTM + Attention	0.03	**18.02**	0.0	~4.1	~4.4
Transformer	**0.23**	**19.89**	0.0	**3.8**	**4.1**

Transformer: Achieved 3.3x higher BLEU score than best RNN-based model.

Token Accuracy Progression: Each model improves: RNN (17.07%) → LSTM (17.05%) → Attention (18.02%) → Transformer (19.89%). Here approximately 2.1% token accuracy improvement (LSTM → LSTM+Attention)

Zero Exact Match: None achieved perfect generation (expected due to vocabulary diversity) **BLEU Anomaly:** Observed anomaly: LSTM+Attention has lower BLEU but higher token accuracy than simpler models. That means BLEU alone is insufficient for code generation evaluation

4.2 Performance by Sequence Length

Average Loss by Docstring Length Category

Model	Short (1-15)	Medium (16-35)	Long (36-50)	Best For
Vanilla RNN	3.8	4.5	5.2	Baseline only
LSTM	3.5	4.2	4.9	Short sequences only
LSTM + Attention	3.2	3.9	4.5	Medium sequences
Transformer	**2.9**	**3.6**	**4.2**	**All lengths**

Insights:

- **Long sequences (>35 tokens):** Use Transformer ONLY
- **Medium sequences (16-35 tokens):** Transformer or LSTM+Attention
- **Short sequences (<15 tokens):** Any model works, but Transformer still best

5. Conclusions

This project successfully implemented and evaluated four Seq2Seq architectures for automatic Python code generation from natural language descriptions. Through comprehensive experiments on the CodeSearchNet dataset, the results demonstrated: