

Assignment: Text-to-Python Code Generation Using Seq2Seq Models (RNNs)

Objective

The objective of this assignment is to explore how different recurrent neural network architectures perform on a **text-to-code generation task**, where **natural language function descriptions (docstrings)** are translated into **Python source code**.

Implement and compare the following models:

- Vanilla RNN-based Seq2Seq
- LSTM-based Seq2Seq
- LSTM with Attention mechanism

Experiences form the tasks:

- Understand limitations of vanilla RNNs in modeling long sequences
- Observe how LSTM improves long-term dependency handling
- Learn how attention mechanisms overcome the fixed-length context bottleneck
- Analyze generated code using quantitative and qualitative metrics

Problem Description

Modern software repositories often contain **natural language documentation (docstrings)** paired with corresponding source code. Automatically generating code from text descriptions is a challenging task because it requires:

- Understanding semantic intent from text
- Preserving long-range dependencies
- Producing syntactically correct and structured output

This task serves as a practical application of **sequence-to-sequence learning** and demonstrates the effectiveness of **attention mechanisms**.

Task Definition

Input (Source Sequence)	Output (Target Sequence)
A natural language function description (docstring).	The corresponding Python function.
Example returns the maximum value in a list of integers	<pre>def max_value(nums): return max(nums)</pre>

Dataset

Dataset to Use

CodeSearchNet – Python CodeText Dataset

This dataset contains pairs of:

- English docstrings (text descriptions)
- Corresponding Python functions

Dataset Access

- Hugging Face dataset:
CodeSearchNet Python
<https://huggingface.co/datasets/Nan-Do/code-search-net-python>

Dataset Characteristics

- Language pair: English → Python
- Total size: ~250,000+ training pairs
- Validation and test splits provided
- Real-world GitHub code
- Each example consists of:
 - `docstring`
 - `code`

Dataset Usage for This Assignment

To keep training feasible:

- Use **5,000–10,000 training examples**
 - Limit maximum sequence length:
 - Docstring: 50 tokens
 - Code: 80 tokens
-

Models to Implement

Model 1: Vanilla RNN Seq2Seq

Architecture

- Encoder: RNN
- Decoder: RNN
- Fixed-length context vector
- No attention

Goal

- Establish a baseline
 - Observe performance degradation for longer docstrings
-

Model 2: LSTM Seq2Seq

Architecture

- Encoder: LSTM
- Decoder: LSTM
- Fixed-length context vector

Goal

- Improve modeling of long-range dependencies
 - Compare performance against vanilla RNN
-

Model 3: LSTM with Attention

Architecture

- Encoder: Bidirectional LSTM
- Decoder: LSTM
- Bahdanau (additive) attention mechanism

Goal

- Remove the fixed-context bottleneck
 - Improve code generation quality
 - Enable interpretability through attention visualization
-

Training Setup

Common Training Configuration

All models must use the same configuration for fair comparison:

- Tokenization: whitespace or provided tokenizer
 - Embedding dimension: 128–256
 - Hidden dimension: 256
 - Loss function: Cross-entropy loss
 - Optimizer: Adam
 - Teacher forcing during training
 - Same train/validation/test split
-

Evaluation Metrics

Metrics Used During Training

- **Training loss (cross-entropy)**
- **Validation loss**

Used to monitor convergence and overfitting.

Metrics Used for Evaluation

Metric	Description
Token-level Accuracy	Percentage of correctly predicted tokens
BLEU Score	N-gram overlap between generated and reference code
Exact Match Accuracy	Percentage of completely correct outputs (for short functions)

Exact match accuracy is especially meaningful for small code snippets.

Experiments and Analysis

Must analyze and compare all three models based on:

1. Training and validation loss curves
 2. BLEU score on the test set
 3. Types of errors:
 - Syntax errors
 - Missing indentation
 - Incorrect operators or variables
 4. Performance vs docstring length
-

Attention Analysis (Mandatory)

For the attention-based model:

- Visualize attention weights for **at least three test examples**
- Plot heatmaps showing alignment between docstring tokens and generated code tokens
- Interpret whether attention focuses on semantically relevant words

Example Question

Does the word *maximum* attend strongly to the `>` operator or `max()` function?

Deliverables

Item	Description
Source Code	Implementations of all three models
Trained Models	Saved checkpoints
Report (PDF)	Experimental results and discussion
Attention Visualizations	Heatmaps with explanations
README	Instructions to run the project

Grading Rubric

Component	Marks
Vanilla RNN implementation	15
LSTM implementation	20
Attention model	25
Experimental evaluation	15
Attention analysis	15
Code quality & reproducibility	10
Total	100

Bonus (Optional)

- Syntax validation using Python AST
 - Extend to longer docstrings
 - Compare with a Transformer-based model
-