Text-to-Code Generation Using Seq2Seq Models

Experimental Results and Discussion

1. Introduction

This experiment evaluates four Sequence-to-Sequence (Seq2Seq) neural network architectures for automatic Python code generation from natural language docstrings. Models were trained on 10,000 samples from the CodeSearchNet Python dataset and evaluated on token accuracy, BLEU score, and syntax validity.

2. Methodology

2.1 Dataset & Configuration

Training: 10,000 samples | Validation: 1,500 | Test: 1,500
Max lengths: Docstring (100 tokens), Code (150 tokens)
Epochs: 15 | Batch Size: 64 | Optimizer: Adam (LR=0.001) | Hidden Dim: 256

2.2 Models Implemented

Model	Key Features
Vanilla RNN	Basic encoder-decoder; baseline for comparison
LSTM	2-layer LSTM with dropout (0.5); handles long sequences
LSTM + Attention	Bidirectional encoder + Bahdanau attention; dynamic focus
Transformer (Bonus)	2 layers, 8 heads; self-attention mechanism

3. Experimental Results

Models evaluated on test set (1,500 samples) using three metrics:

Model	Token Acc (%)	BLEU Score	Syntax Valid (%)
LSTM + Attention	**12.60**	**11.98**	**5.0**
LSTM	11.31	11.07	5.0
Vanilla RNN	10.05	8.98	0.0
Transformer	0.51	3.12	83.0*

4. Key Findings

4.1 LSTM + Attention: Best Performance

Achieved highest scores (BLEU: 11.98, Token Acc: 12.60%) due to:

- Bidirectional encoding captures full context
- Attention mechanism aligns keywords (e.g., "maximum" → "max()")
- No fixed-length bottleneck - maintains access to all encoder states

4.2 LSTM vs Vanilla RNN

LSTM outperformed RNN (11.07 vs 8.98 BLEU) because:

- Gating mechanisms prevent vanishing gradients
- Cell state preserves long-term dependencies (critical for 100+ token sequences)
- Can remember function signatures and variable names from docstring beginning

4.3 Transformer Underperformance

Despite being state-of-the-art, Transformer failed (3.12 BLEU) because:

- Requires 100K+ samples; only had 10K (data-hungry architecture)
- Sensitive to hyperparameters (warmup schedule, large batches needed)
- No inductive bias for sequences (must learn order from data)
- Lesson: Complex models ≠ better results on small datasets

4.4 Low Syntax Validity

Only 0-5% of generated code is syntactically valid because:

- Models learned lexical patterns but not Python grammar
- Token-level decoding ignores hierarchical structure (AST trees)
- Common errors: indentation, unbalanced parentheses, missing colons

5. Attention Mechanism Analysis

Attention heatmaps reveal successful semantic alignments:

- "maximum" in docstring → high attention on "max(" in code
- "sum" → "+" operator
- "returns" → "return" keyword
- Variable names copied directly from docstring mentions

Visualizations show the model learned to focus on relevant source tokens when generating each code token (see results/attention_viz/ folder).

6. Conclusion & Recommendations

For low-resource text-to-code generation (10K samples), LSTM with Attention provides the best performance. Key takeaway: appropriate inductive biases (sequential processing) outweigh architectural sophistication on small datasets.

References

[1] Sutskever et al. (2014). Sequence to Sequence Learning with Neural Networks. NeurIPS.
[2] Bahdanau et al. (2014). Neural Machine Translation by Jointly Learning to Align. ICLR.
[3] Vaswani et al. (2017). Attention Is All You Need. NeurIPS.
[4] Husain et al. (2019). CodeSearchNet Challenge. arXiv:1909.09436.