Transformer-Based Sequence-to-Sequence Model for Text Reconstruction

Galileo Steinberg

November 14, 2025

1 Introduction

This project tackles the task of reconstructing original sentences from preprocessed text. The preprocessing removes determiners, conjunctions, and prepositions, and lemmatizes remaining words. For example, "with scarcely semi - colon after his hearty thanks , little man begin his recital :" must be reconstructed as "with scarcely a semi - colon after his hearty thanks , the little man began his recital :". The model must identify missing function words, restore proper verb conjugations, and maintain grammatical coherence.

This task is motivated by practical applications in text processing and understanding how neural models learn linguistic structure. Text reconstruction requires the model to internalize grammatical rules, lexical relationships, and syntactic patterns without explicit linguistic annotations. Success on this task demonstrates that neural architectures can learn these patterns from data alone.

I implement a Transformer encoder-decoder model and conduct three systematic experiments: (1) comparing sinusoidal versus learnable positional encodings to understand how position information affects performance, (2) evaluating greedy decoding versus beam search with varying beam widths to assess the cost-benefit trade-off of different generation strategies, and (3) exploring architectural variants by varying attention heads and model depth to identify which design choices most impact performance.

2 Code Structure and Execution

My codebase consists of `model.py` (Transformer architecture with both positional encoding variants), `dataset.py` (data loading with teacher forcing), `main.py` (training and evaluation), and `evaluate_decoding.py` (decoding strategy comparison).

To run experiments, execute: `./run_experiment1.sh` for positional encoding comparison, `./run_experiment2.` for decoding algorithms, and `./run_experiment3.sh` for architecture variants. See `EXPERIMENTS.md` and `colab_setup.ipynb` for detailed instructions. All experiments used Google Colab with GPU acceleration and random seed 42.

3 Methods

3.1 Transformer Architecture

The Transformer uses an encoder-decoder architecture with attention mechanisms instead of recurrence. The encoder processes the source sequence (preprocessed text) through multiple layers, each containing multi-head self-attention and feedforward networks. Self-attention allows each position to attend to all positions in the input, capturing dependencies regardless of distance. The decoder generates the target sequence (original text) autoregressively, using masked self-attention to prevent attending to future positions and cross-attention to incorporate information from the encoder outputs.

Multi-head attention is the core mechanism. It computes attention by projecting queries, keys, and values through learned linear transformations, computing attention scores as the dot product of queries and keys scaled by the square root of the dimension, applying softmax to get attention weights, and taking a weighted sum of values. Multiple heads run this process in parallel with different learned projections, allowing the model to capture different types of relationships simultaneously.

Masking is essential for correct behavior. Padding masks prevent attention to padding tokens in variable-length sequences. Causal masks in the decoder prevent positions from attending to future tokens, which is necessary for autoregressive generation where each token can only depend on previous tokens. During training with teacher forcing, causal masking ensures the model learns to predict each token using only previous context.

Positional encoding adds position information since Transformers lack inherent sequential structure. Sinusoidal encoding uses fixed sine and cosine functions at different frequencies for each dimension, providing smooth patterns that capture relative positions. Learnable encoding treats positions as trainable embeddings similar to word embeddings.

3.2 Implementation

I implemented `SeqPairDataset` which loads JSON data, tokenizes using a shared vocabulary (minimum frequency 2), adds special tokens (`<bos>`, `<eos>`, `<pad>`, `<unk>`), and creates teacher forcing pairs. The encoder input is the full source sequence, decoder input is the target shifted right (last token removed), and labels are shifted left (first token removed). This shifting enables the model to learn next-token prediction.

For Experiment 1, I added `LearnablePosEncoding` to `model.py`, implementing positional embeddings as trainable parameters. Both encoder and decoder accept a `pos_encoding_type` parameter to switch between sinusoidal and learnable variants.

Training uses Adam optimizer (learning rate 0.001) with cross-entropy loss that ignores padding tokens. Models train for 15-20 epochs with early stopping based on development loss. Hyperparameters: batch size 64, model dimension 128, feedforward dimension 512, dropout 0.1, maximum sequence length 50.

4 Results

4.1 Experiment 1: Positional Encoding Strategies

I compared sinusoidal versus learnable positional encoding with identical hyperparameters (15 epochs, 4 heads, 2 layers, seed 42).

Table 1: Experiment 1: Positional Encoding Comparison

Method	**Parameters**	**Test BLEU**	**Best Dev Loss**	**Dev BLEU**
Sinusoidal	14,765,033	0.8591	0.6597	0.8694
Learnable	14,790,633	0.8582	0.6443	0.8694
Difference	+25,600	-0.0009	-0.0154	0.0000

Sinusoidal encoding achieves slightly higher test BLEU (0.8591 vs 0.8582) despite learnable encoding having 25,600 additional parameters and lower development loss (0.6443 vs 0.6597). Both achieve identical development BLEU (0.8694). The 0.09% difference is marginal, suggesting both work comparably.

4.2 Experiment 2: Decoding Algorithms

I trained one model to convergence (20 epochs) then evaluated with greedy decoding and beam search at widths 3, 5, and 10.

Table 2: Experiment 2: Decoding Strategy Comparison

Strategy	**BLEU**	**Time/Sample (s)**	**Speedup**	**Avg Length**
Greedy	0.8656	0.0984	1.0×	16.52
Beam (width=3)	0.8667	0.5020	0.20×	16.53
Beam (width=5)	0.8666	0.5377	0.18×	16.53
Beam (width=10)	0.8660	1.0435	0.09×	16.53

Greedy decoding achieves 0.8656 BLEU with fastest inference (0.0984s/sample). Beam width 3 provides only 0.11% improvement while being 5.1× slower. Wider beams don't help: width 10 actually performs worse (0.8660) at 10.6× slower.

4.3 Experiment 3: Architecture Variants

I tested attention heads (2, 4, 8) with 2 layers fixed, and model depth (1, 2, 4 layers) with 4 heads fixed.

Table 3: Experiment 3a: Effect of Attention Heads (2 layers)

Heads	**Parameters**	**Test BLEU**	**Best Dev Loss**	**Dev BLEU**
2	14,765,033	0.8582	0.6702	0.8679
4	14,765,033	0.8591	0.6597	0.8694
8	14,765,033	0.8634	0.6541	0.8740

Table 4: Experiment 3b: Effect of Model Depth (4 heads)

Layers	**Parameters**	**Test BLEU**	**Best Dev Loss**	**Dev BLEU**
1	14,302,185	0.8471	0.8013	0.8544
2	14,765,033	0.8591	0.6597	0.8694
4	15,690,729	0.8728	0.5902	0.8827

More attention heads provide modest monotonic improvement (2 heads: 0.8582, 8 heads: 0.8634, +0.52 points). Model depth has much stronger impact (1 layer: 0.8471, 4 layers: 0.8728, +2.57 points) with only 9.7% more parameters. The 4-layer model achieves best overall performance.

4.4 Example Outputs

Here are example reconstructions from the best model (4 layers, 4 heads, sinusoidal encoding, greedy decoding):

Example 1:

- Source: *with scarcely semi - colon after his hearty thanks , little man begin his recital :*
- Target: *with scarcely a semi - colon after his hearty thanks , the little man began his recital :*
- Prediction: *with scarcely a semi - colon after his hearty thanks , the little man began his recital :*

Example 2:

- Source: *he be always go lose something*
- Target: *he was always going to lose something*
- Prediction: *he was always going to lose something*

Example 3:

- Source: *i have report on case development*
- Target: *i have a report on the case development*
- Prediction: *i have a report on the case development*

Example 4 (error case):

- Source: *he say he be not aware any plan*
- Target: *he said he was not aware of any plan*
- Prediction: *he said he was not aware of any plans*

Example 5 (error case):

- Source: *market be expect fall today*
- Target: *the market was expected to fall today*
- Prediction: *the markets were expected to fall today*

Examples 1-3 show perfect reconstruction. Example 4 has a minor error ("plans" vs "plan"). Example 5 demonstrates typical errors: number mismatch ("markets" vs "market") and verb form ("were" vs "was").

5 Analysis

Several patterns emerge across experiments. Simpler approaches often match complex ones: sinusoidal encoding outperforms learnable despite no trainable parameters, and greedy decoding achieves 99.87% of beam search performance while being 5× faster. For well-defined tasks with clear patterns, complexity doesn't guarantee improvement.

The minimal beam search benefit contrasts with tasks like machine translation where it helps significantly. Text reconstruction is highly constrained by grammatical rules and source content, limiting valid outputs. The model learns these constraints, assigning high confidence to correct sequences, making greedy selections often optimal. Beam width 10 hurting performance reveals that wider beams retain lower-quality hypotheses that accumulate errors. The scoring function (cumulative log-probability) doesn't perfectly align with BLEU.

Architecture comparison shows depth beats width. Deeper networks enable hierarchical processing: early layers capture local patterns (word relationships, grammar), middle layers handle medium-range dependencies (phrase structure, agreement), and deep layers model long-range coherence. Attention heads operate in parallel at the same abstraction level, providing diversity but not hierarchy. The 4-layer model achieves 0.8728 BLEU with only 9.7% more parameters than 1 layer, showing architectural organization matters more than raw parameter count.

The learnable encoding paradox (lower development loss, worse test BLEU) suggests overfitting or metric mismatch. Cross-entropy loss measures token-level accuracy while BLEU measures sequence-level n-gram overlap. Learnable embeddings may fit validation distribution better without improving generation quality. Sinusoidal patterns' smooth mathematical structure generalizes more robustly.

Common failure patterns include article selection ("a" vs "an" vs "the"), verb conjugation (lemma to correct tense), preposition choice, number agreement, and long sequences (30+ tokens). However, high BLEU scores (0.85-0.87) indicate these are rare. The model successfully handles most cases.

6 Conclusion

This project demonstrates that Transformer encoder-decoder models achieve strong performance on text reconstruction (0.85-0.87 BLEU). The best configuration uses 4 layers, 4 attention heads, sinusoidal positional encoding, and greedy decoding (0.8728 BLEU).

Key findings: (1) Simpler methods often match complex alternatives for well-defined tasks. (2) Model depth is more important than width for hierarchical feature learning. (3) Beam search provides minimal benefit when models are confident and tasks are constrained.

For practical deployment, the 2-layer model with 4 heads (0.8591 BLEU) offers 98.4% of best performance with faster training. Beam search is not recommended given its 5-10× computational cost for only 0.11% BLEU improvement.

Limitations include testing only two architectural dimensions (heads and depth), using a single random seed, relying solely on BLEU, and computational constraints preventing exploration of very deep/wide models. Future work could explore relative positional encodings, layer-wise learning rates, alternative attention mechanisms, ensemble methods, and data augmentation.

The key takeaway is that thoughtful design beats complexity for specialized tasks. Understanding text reconstruction's characteristics (constrained outputs, grammatical rules) guided effective choices. Matching model design to task properties is more valuable than applying general-purpose solutions.