A Transformer-Based Quantum Error Decoding Model: Detailed Explanation and Implementation

Saurabh Rai

March 24, 2025

Abstract

This document explains in detail a basic project that employs transformer architectures for quantum error decoding in a simple 3-qubit repetition code. We discuss the data generation process for syndrome measurements with independent and biased noise, describe the classical majority vote decoder, and then present an enhanced transformer-based decoder. Enhancements include separate embeddings for each syndrome bit, an extra classification head, and training strategies to mitigate overfitting such as weight decay and early stopping. The complete Python code is provided and explained section by section.

Contents

1 Introduction and Motivation

Quantum error correction (QEC) is essential for reliable quantum computing. In a noisy quantum environment, error decoding determines which qubit (if any) has undergone an error. In our project, we focus on a simple 3-qubit repetition (bit-flip) code. In this code, one of four error scenarios can occur:

- **Class 0**: No error — syndrome: $(0, 0)$.
- **Class 1**: Error on qubit 1 — syndrome: $(1, 0)$.
- **Class 2**: Error on qubit 2 — syndrome: $(1, 1)$.
- **Class 3**: Error on qubit 3 — syndrome: $(0, 1)$.

We simulate a sequence of syndrome measurements over several rounds, incorporating measurement noise. Two noise modes are considered: *independent noise* (each syndrome bit is flipped with the same probability) and *biased noise* (the two syndrome bits have different flip probabilities). The task is to decode the true error based on these noisy syndrome sequences.

2 Data Generation

2.1 Syndrome Mapping

Each syndrome is represented by two bits. In a traditional approach these two bits can be mapped to a single token (an integer between 0 and 3) using a predefined mapping:

$$(0, 0) \to 0, \quad (1, 0) \to 1, \quad (1, 1) \to 2, \quad (0, 1) \to 3.$$

This mapping simplifies the input representation. However, in our enhanced version, we use the two bits separately with distinct embedding layers so that the model can learn individual characteristics of each bit.

2.2 Noise Models

The code implements two noise models:

(a) **Independent Noise**: Each syndrome bit is flipped independently with a probability `noise_prob`.

(b) **Biased Noise**: The first bit is flipped with probability `noise_prob1` and the second with probability `noise_prob2`. This introduces a systematic bias that the model must learn to overcome.

2.3 Enhanced Input Representation

In the enhanced mode, rather than converting the two syndrome bits into one token, we retain them as a two-dimensional vector (i.e., `[bit1, bit2]`). This allows us to apply separate embedding layers to each bit later in the network.

3 Model Architecture

Our model is a transformer-based classifier with several enhancements:

- **Separate Bit Embeddings:** Instead of mapping the two bits into one token, we use a module (`SyndromeBitEmbedding`) that embeds each bit separately via two embedding layers and then combines the resulting vectors (by concatenation followed by a linear projection).
- **Transformer Encoder:** The embedded input (with positional encoding) is fed into multiple transformer encoder layers. The transformer architecture uses multi-head self-attention to capture dependencies over the sequence of syndrome measurements.
- **Extra Classification Head:** After mean pooling the output from the transformer, an extra fully connected layer with a ReLU activation and dropout is added before the final classification layer. This helps the network learn a complex non-linear mapping from the transformer output to the error class.

3.1 Positional Encoding

Transformers do not have inherent sequential order information, so a positional encoding is added to the input embeddings:

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right), \quad PE(pos, 2i{+}1) = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right).$$

This encoding is added to the embedded input to incorporate sequence order.

4 Training and Overfitting Mitigation

4.1 Overfitting Issues

Overfitting is a common challenge when training deep models. In our implementation, we address overfitting using:

- **Weight Decay:** L2 regularization is applied by setting the `weight_decay` parameter in the Adam optimizer. This penalizes large weights.
- **Early Stopping:** We monitor validation accuracy during training. If the validation accuracy does not improve for a specified number of epochs (patience), training stops early, and the best model parameters are restored.

4.2 Training Loop

The training loop involves:

1. Running a forward pass over the training data.
2. Computing the cross-entropy loss.
3. Backpropagating and updating the weights with the optimizer (which now includes weight decay).
4. Evaluating validation accuracy after each epoch.
5. Checking for early stopping if the validation accuracy does not improve for a preset number of epochs.

5 Experimental Setup

We perform experiments under two noise scenarios:

1. **Independent Noise:** where each bit is flipped with a probability of 0.4.
2. **Biased Noise:** where the first bit is flipped with a probability of 0.3 and the second bit with 0.1.

In both cases, we compare the performance of the transformer-based decoder with a classical majority vote decoder (which simply returns the most common token in the syndrome sequence).

6 Conclusion

This document presented a detailed explanation of a very basic transformer-based quantum error decoding model. We saw the data generation process for syndrome measurements under independent and biased noise, how separate bit embeddings and an enhanced classification head are integrated into a transformer architecture, and strategies to mitigate overfitting through

weight decay and early stopping. This is meant to be seen as a preliminary effort towards incorporating ideas from machine learning in quantum computing.