Worked example: CircuitHistoryEncoder and Positional Encoding

This document walks through a tiny, concrete example of how the `CircuitHistoryEncoder` and the sinusoidal `PositionalEncoding` (as implemented in `transformer/PositionalEncoding.py`) transform a short sequence of circuit actions into position-aware embeddings suitable for a transformer.

Setup and conventions

We use the following conventions consistent with the code:

- Embedding dimension (model width): $d_{model} = 8$.
- Token types: 0=pad, 1=input, 2=constant, 3=add, 4=multiply.
- Node index embedding vocabulary size: 100 (indices 0..99).
- The transformer expects input shaped as $(\text{seq\_len}, \text{batch}, d_{model})$.

Actions and tokens

Consider a tiny circuit with three actions (three nodes):

$$\text{actions} = [\ (\text{"input"}, -, -),\ (\text{"input"}, -, -),\ (\text{"add"}, 0, 1)\ ]$$

Using `encode_circuit_actions`, these become the token list (sequence length $S = 3$):

$$\begin{aligned}
\tau_0 &= \{\text{type} : 1,\ \text{value} : 0,\ \text{node_idx} : 0\},\\
\tau_1 &= \{\text{type} : 1,\ \text{value} : 1,\ \text{node_idx} : 1\},\\
\tau_2 &= \{\text{type} : 3,\ \text{value} : 0,\ \text{node_idx} : 2,\ \text{input1} : 0,\ \text{input2} : 1\}.
\end{aligned}$$

Embedding step

Each token yields three embeddings which are summed elementwise to produce the final token embedding:

- $E_{type}(\tau)$: from a learnable `nn.Embedding(5, d_model)` using the token `type`.
- $E_{value}(\tau)$: from a learnable linear layer `nn.Linear(1, d_model)` applied to the scalar `value`.
- $E_{node}(\tau)$: from a learnable `nn.Embedding(100, d_model)` using `node_idx`.

Concrete computed embeddings

The encoder output (before batching) is the time-first matrix X with shape (S, d_{model}) where $S = 3$. We truncate values to two decimals for readability:

$$X \approx \begin{bmatrix} 1.07 & -0.16 & -0.86 & -3.53 & -0.68 & -1.20 & 0.58 & -0.14 \\ -0.46 & -0.15 & 0.94 & -2.41 & -3.41 & 0.41 & 1.36 & -0.30 \\ -0.56 & -2.16 & 0.28 & 3.31 & 2.04 & -0.51 & -1.87 & 0.05 \end{bmatrix}$$

Batch dimension and positional encoding

We add a batch dimension ($B = 1$):

$$X_{batched} \in \mathbb{R}^{3\times 1\times 8}, \quad X_{batched} = \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix}.$$

The sinusoidal positional encoding matrix P is defined by:

$$P_{pos,2i} = \sin\Big(\frac{pos}{10000^{2i/d_{model}}}\Big), \qquad P_{pos,2i+1} = \cos\Big(\frac{pos}{10000^{2i/d_{model}}}\Big).$$

For $d_{model} = 8$ and positions $0, 1, 2$, rounded values are:

$$\begin{aligned} P_{0,:} &\approx [0, 1, 0, 1, 0, 1, 0, 1], \\ P_{1,:} &\approx [0.84, 0.54, 0.10, 0.99, 0.01, 1.00, 0.00, 1.00], \\ P_{2,:} &\approx [0.91, -0.42, 0.20, 0.98, 0.02, 1.00, 0.00, 1.00]. \end{aligned}$$

Applying $Y = X_{batched} + P_{0:3,:}$ gives:

$$Y \approx \begin{bmatrix} 1.07 & 0.84 & -0.86 & -2.53 & -0.68 & -0.20 & 0.58 & 0.86 \\ 0.38 & 0.39 & 1.04 & -1.42 & -3.40 & 1.41 & 1.36 & 0.70 \\ 0.35 & -2.57 & 0.48 & 4.29 & 2.06 & 0.49 & -1.87 & 1.05 \end{bmatrix}$$

Remarks

- The positional encoding is deterministic and fixed (not learned).
- The encoder sums three information sources: type, value, and node identity.

- For batching circuits of varying lengths, pad to a common S and use attention masks.
- Avoid fixing device tensors at construction; use `module.to(device)`.

Short example: encoding $x_0 + x_1 + 1$

Below is a concise walkthrough of how the expression $x_0 + x_1 + 1$ is turned into the network input:

1. Linearize the computation as actions (one action per node), e.g.:

$$\begin{aligned}
&(\text{"input"}, -, -) \quad (\text{node } 0),\\
&(\text{"input"}, -, -) \quad (\text{node } 1),\\
&(\text{"constant"}, -, -) \quad (\text{node } 2),\\
&(\text{"add"}, 0, 1) \quad (\text{node } 3 \ = x_0 + x_1),\\
&(\text{"add"}, 3, 2) \quad (\text{node } 4 \ = (x_0 + x_1) + 1).
\end{aligned}$$

2. Tokens: each action becomes a token dictionary with fields `type`, `value`, and `node_idx` (and inputs for ops). Example token sequence:

$$\tau_0, \tau_1, \tau_2, \tau_3, \tau_4$$

.

3. Embedding: for each token we compute

$$x_i = E_{type}(\tau_i) + E_{value}(\tau_i) + E_{node}(\tau_i)$$

resulting in a time-first matrix $X \in \mathbb{R}^{S \times d_{model}}$ (here $S = 5$).

4. Batch and add positional encodings: convert to $X_{batched} \in \mathbb{R}^{S \times 1 \times d_{model}}$ and compute

$$Y = X_{batched} + P_{0:S,:}$$

which is the final input to the transformer.

What the transformer does

After receiving the positional-encoded input Y, the transformer applies stacked layers of multi-head self-attention and positionwise feed-forward networks. The self-attention layers let each position (token) attend to other positions in the sequence (so the model can combine input tokens and constants according to the computation graph), and the feed-forward layers mix and transform these combined features.

Typical outputs depend on the task: for generation or prediction you may decode from the final sequence representations (e.g., project the last token or all tokens to logits over actions or values); for classification/regression you might pool across positions (mean, attention-pool, or take a special token) and apply a final linear head to produce the desired scalar or categorical outputs.

Beginner-friendly: switching to a one-hot encoder

We will replace the current sum-of-embeddings input with a simple one-hot style encoder. Here's a plain-language explanation a newcomer can follow:

- A "one-hot" vector is just a long row of zeros with a single 1 in the position that represents some choice. For example, if we have 5 token types and the token is an "input", the type one-hot looks like [0,1,0,0,0].
- For each token we build a few small one-hot vectors (one for the token type, one for the node index, and one for the value bucket). We glue those small rows together into one longer row. That long row is the token's feature vector.
- A learned linear layer then squashes this long one-hot row down to the transformer's working size (the d_{model} that the transformer expects). This is similar to how the current embeddings work, but more explicit about each field.
- After this projection we add the same positional encoding and pass the result into the transformer unchanged.

Why this is helpful for beginners:

- It's very explicit — every bit of the input corresponds to a clear, human-readable field (type, node id, value bucket).
- It's easy to convert model outputs back into readable predictions (look at which position in the one-hot is most likely).
- It's straightforward to implement and reason about: build one-hot, project, add position, run transformer.

Practical note: if the node index or value spaces are very large, the one-hot vector gets large. In that case we can still use the same idea but implement it efficiently with small embedding tables for node indices and types (those are equivalent but more memory-friendly).