Z-GED Quick Read Guide

What the Code Does, in Plain Language

Student + Faculty Friendly Summary

February 13, 2026

1) The Big Picture (30-second version)

This project learns to generate circuit topologies (which nodes connect to which, and with what component types) from a compact 8-number latent code.

The flow is:

1. Turn each circuit into graph numbers (nodes, edges, poles/zeros).
2. Encode that graph into 8 latent numbers with a VAE encoder.
3. Decode those 8 numbers back into a circuit topology.
4. Train with losses that reward correct topology and connectivity.

2) What Inputs Look Like in Code

Node features (4 numbers per node)

$$[\text{is\_GND},\ \text{is\_VIN},\ \text{is\_VOUT},\ \text{is\_INTERNAL}]$$

Example: VIN node is $[0, 1, 0, 0]$.

Edge features (3 numbers per edge)

$$[\log_{10}(R),\ \log_{10}(C),\ \log_{10}(L)]$$

If a component is absent on that edge, its slot is set to 0.

Example edge with R=1kΩ, C=100nF, and no inductor:

$$[3.0,\ -7.0,\ 0.0]$$

because $\log_{10}(1000) = 3$, $\log_{10}(10^{-7}) = -7$.

Poles and zeros These are variable-length lists of complex values stored as $[\text{real}, \text{imag}]$ pairs.

3) Encoder: How the Model Compresses a Circuit into 8 Numbers

The encoder has two main stages.

Stage A: Graph message passing

The GNN updates each node by reading neighboring nodes and edge values. For each edge, it uses separate transforms for R, C, and L, and derives presence masks from nonzero values:

$$\text{is\_R} = |\log_{10}(R)| > 0.01, \quad \text{is\_C} = |\log_{10}(C)| > 0.01, \quad \text{is\_L} = |\log_{10}(L)| > 0.01.$$

So the model can treat resistor, capacitor, and inductor effects differently.

Stage B: 3 latent branches

After GNN processing, the encoder builds 3 branches:

- **Topology branch (2D):** global graph structure.
- **Structure branch (2D):** uses GND/VIN/VOUT embeddings.
- **Pole-zero branch (4D):** uses DeepSets over poles/zeros.

Together:

$$z = [z_{\text{topology}}(2) \mid z_{\text{structure}}(2) \mid z_{\text{pz}}(4)] \in \mathbb{R}^8.$$

Because this is a VAE, code samples from a Gaussian using μ and $\log \sigma^2$:

$$z = \mu + \sigma \odot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I).$$

4) Decoder: How 8 Numbers Become a Circuit

The decoder is autoregressive (step-by-step generation):

1. Predict total node count (3 to max_nodes).
2. Generate node types one at a time.
3. Generate edge/component classes one edge at a time with causal attention.

Edge class is an 8-way choice:

Class	Meaning
0	No edge
1	R
2	C
3	L
4	RC
5	RL
6	CL
7	RCL

This is why the decoder can output both connectivity and component type together.

What Causal Attention Is Doing (Context + K/Q/V)

The decoder is a sequence model. At each step, it predicts the *next* graph decision while looking at past decisions.

Context used at each decoding step

- Global context: latent code z (the whole-circuit intent).
- Position context: where we are in the sequence (position embedding).
- History context: previously generated nodes/edges.
- Local context: current node position or current node pair (i, j).

K/Q/V in plain language

- **Q (Query):** "What information do I need *right now* to make this prediction?"
- **K (Key):** "What kind of information is stored in each past token?"
- **V (Value):** "What actual content should be read from each past token?"

Attention scores are computed from Query–Key similarity, then used to mix Values.

Node decoder (Transformer decoder)

- Query comes from current step token (latent + position + length context).
- Keys/Values come from previously generated node embeddings.
- Output predicts the next node type.

Edge decoder (causal self-attention)

- Each edge-step token includes edge-pair embedding + latent projection + position + previous-edge-token embedding.
- The causal mask blocks access to future edge steps.
- So edge decision t can depend on edge decisions $1 \dots t-1$, but not $t+1 \dots$.

Training vs inference

- Training: teacher forcing provides true previous edge classes.
- Inference: model feeds back its own previous predictions.

5) How Numbers Are Manipulated During Training

At each batch:

1. Encoder outputs $z, \mu, \log \sigma^2$.
2. Targets are converted to dense matrices (node types, edge existence, component classes).
3. Decoder predicts logits (unnormalized scores) for:

- node type,
- node count,
- edge-component class.

4. Losses compare predicted logits to target labels.

Total loss in code:

$$\mathcal{L} = 1.0\,\mathcal{L}_{\text{node-type}} + 5.0\,\mathcal{L}_{\text{node-count}} + 2.0\,\mathcal{L}_{\text{edge-comp}} + 5.0\,\mathcal{L}_{\text{connectivity}} + 0.01\,\mathcal{L}_{\text{KL}}.$$

Interpretation:

- Bigger weight = optimizer cares more about that objective.
- Connectivity loss strongly pushes VIN/VOUT to remain connected.
- KL loss keeps latent space smooth for sampling/interpolation.

6) A Tiny End-to-End Example

Suppose one circuit edge is represented as:

$$[3.0,\ -7.0,\ 0.0].$$

The model interprets this as:

- resistor present (1kΩ),
- capacitor present (100nF),
- inductor absent.

After full encoding, assume latent code:

$$z = [1.2, -0.8,\ 0.5, -1.1,\ 0.2, 0.1, -0.3, 0.4].$$

Decoder then predicts:

- node count (for example 4 nodes),
- node sequence (GND, VIN, VOUT, INTERNAL),
- each edge class (for example R, C, no-edge, ...).

7) Why This Design Is Easy to Explain

- Graph encoder captures local circuit structure.
- Latent code compresses that structure into 8 interpretable dimensions.
- Autoregressive decoder builds a valid graph step-by-step.
- Joint edge-component class avoids inconsistent edge/type predictions.

8) Current Practical Numbers

Using current production code:

- Encoder parameters: 83,411
- Decoder parameters: 7,698,901
- Dataset size: 360 circuits (288 train / 72 validation)

9) Where to Look in the Repository

- `ml/data/dataset.py`
- `ml/models/gnn_layers.py`
- `ml/models/encoder.py`
- `ml/models/decoder.py`
- `ml/losses/circuit_loss.py`
- `scripts/training/train.py`

One-line takeaway

This code learns a compact 8-number representation of circuits, then uses that representation to generate circuit topology step-by-step with graph-aware deep learning.