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

Read this in the exact order the encoder runs in code.

Step E1: Start from node and edge tensors

- Nodes: one-hot vectors (size 4 each).
- Edges: $[\log_{10}(R), \log_{10}(C), \log_{10}(L)]$ (size 3 each).

Step E2: Run 3 GNN layers (ImpedanceGNN)

Each layer applies an `ImpedanceConv`, then normalization/activation/residual in the wrapper.

Inside one ImpedanceConv layer, per directed edge $j \to i$:

1. Read edge values:

$$v_R = \log_{10}(R), \quad v_C = \log_{10}(C), \quad v_L = \log_{10}(L).$$

2. Build component masks:

$$m_R = \mathbf{1}(|v_R| > 0.01), \quad m_C = \mathbf{1}(|v_C| > 0.01), \quad m_L = \mathbf{1}(|v_L| > 0.01).$$

3. Compute component-specific messages:

$$\text{msg}_R = f_R([h_j, v_R]), \quad \text{msg}_C = f_C([h_j, v_C]), \quad \text{msg}_L = f_L([h_j, v_L]).$$

4. Combine by masks (this is the key update you asked about):

$$\text{msg}_{j\to i} = m_R \,\text{msg}_R + m_C \,\text{msg}_C + m_L \,\text{msg}_L.$$

5. Compute attention weight a_{ij} from $[h_i, h_j]$, then scale message:

$$\tilde{\text{msg}}_{j\to i} = a_{ij} \,\text{msg}_{j\to i}.$$

Then per node i:

1. Aggregate incoming messages (sum over neighbors).
2. Add bias and dropout (inside `ImpedanceConv`).
3. In `ImpedanceGNN`: LayerNorm, ReLU (except last layer), and residual add.

So each layer produces a new node state:

$$h_i^{(\ell+1)} = \text{ResidualProj}\left(h_i^{(\ell)}\right) + \text{PostProcess}\left(\sum_{j\in\mathcal{N}(i)} \tilde{\text{msg}}_{j\to i}\right).$$

Step E3: Build 3 latent branches from final node states

- **Topology branch (2D):** mean+max graph pooling $\to$ MLP $\to (\mu_{\text{topo}}, \log\sigma^2_{\text{topo}})$.
- **Structure branch (2D):** extract final embeddings of GND, VIN, VOUT, concatenate, MLP $\to (\mu_{\text{struct}}, \log\sigma^2_{\text{struct}})$.
- **Pole-zero branch (4D):** DeepSets(poles), DeepSets(zeros), concatenate, MLP $\to (\mu_{\text{pz}}, \log\sigma^2_{\text{pz}})$.

Step E4: Concatenate and sample latent code

$$\mu = [\mu_{\text{topo}} | \mu_{\text{struct}} | \mu_{\text{pz}}], \quad \log \sigma^2 = [\log \sigma^2_{\text{topo}} | \log \sigma^2_{\text{struct}} | \log \sigma^2_{\text{pz}}].$$

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

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

4) Decoder: How 8 Numbers Become a Circuit

Read this section as the exact generation order in code:

Step 1: Build global decoder context from latent z

- Input is one 8D latent vector.
- A context MLP turns z into a hidden-state vector used by later steps.

Step 2: Predict node count first

- A node-count head predicts how many nodes to generate (from 3 to max_nodes).
- This happens before any node/edge sequence decoding.

Step 3: Node Transformer runs next (one node at a time)

For node position i:

1. Build current node token from: latent context + position embedding + total-length embedding.
2. Attend to previously generated node embeddings $[0, \ldots, i-1]$.
3. Predict node-type logits for position i.
4. Convert predicted type to node embedding and append to history.

Node Transformer K/Q/V

- Q: from the current node token ("what do I need now?").
- K: from previous-node memory ("what past info is available?").
- V: from previous-node memory ("what content do I read?").

Step 4: Edge Transformer runs after nodes (one edge-pair step at a time)

Edge pairs are ordered as (i, j) with $j < i$. For edge step t:

1. Build edge-step token from:
   - node-pair embedding (from node i, j),
   - latent projection,
   - edge position embedding,

- previous-edge-token embedding.

2. Run Transformer self-attention with a causal mask.
3. Predict 8-class edge/component logits for this step.
4. Feed predicted edge class to later edge steps.

Edge Transformer K/Q/V

- Q_t: from current edge-step token x_t.
- $K_{1..t}, V_{1..t}$: from current and earlier edge-step tokens.
- Causal mask blocks future steps, so step t cannot see $t+1, t+2, \ldots$.

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

Step 5: Training vs inference

- Training (teacher forcing): previous edge tokens come from ground truth.
- Inference (generation): previous edge tokens come from model 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.