Assignment 1: Debugging Transformer Models with PyCharm and WSL

43 Snapshots with Shape & Role Explanations (Encoder–Decoder Transformer)

Objective

Debugger-based walkthrough of a standard encoder–decoder Transformer using **PyCharm + WSL**. We include **43 numbered snapshots**, each with the exact variable, the **shape**, and a short explanation of **what it represents and why the shape is correct**.

Environment (PyCharm + WSL) — Optional but Recommended

If available, include:

- **Env-1:** PyCharm interpreter set to WSL (screenshot: `env1.png`).
- **Env-2:** Debugger hit with Variables pane visible (screenshot: `env2.png`).

Model & Data Compliance

- **Transformer:** Encoder–decoder (Vaswani et al., 2017), **2** encoder layers, **2** decoder layers, **4** heads, $d_{\text{model}} = 128$, $d_{\text{ff}} = 256$.
- **Source (6 tokens):** `hello from cairo egypt team project`
- **Target (5 tokens):** `we are debugging transformer today`
- **Shapes used throughout:** batch = 1; src_len = 6; tgt_len = 5; heads = 4; $d_{\text{head}} = d_{\text{model}}/4 = 32$.

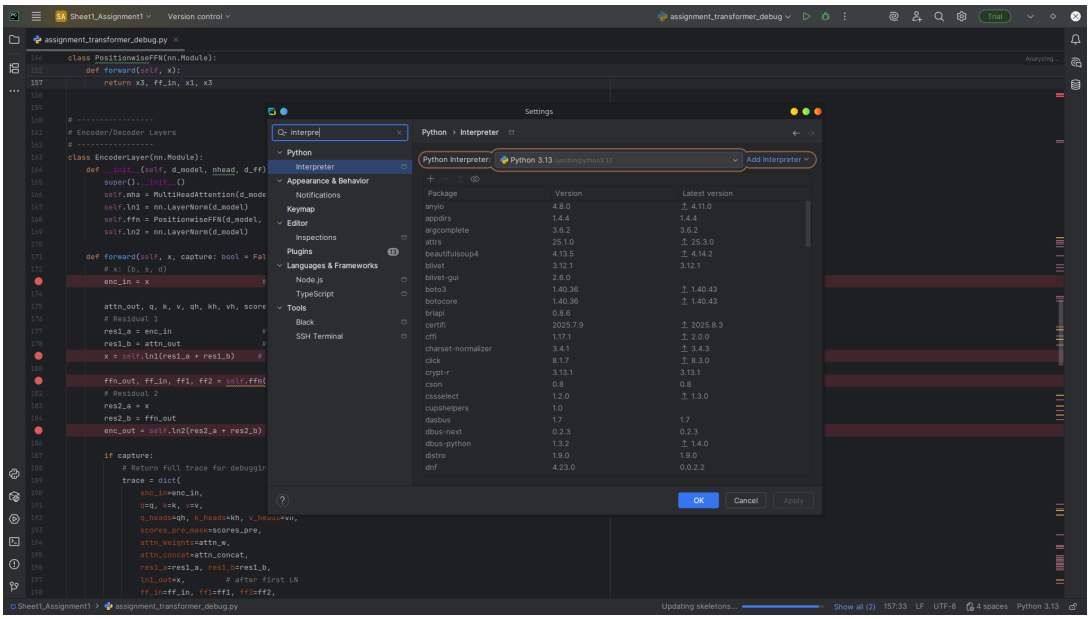
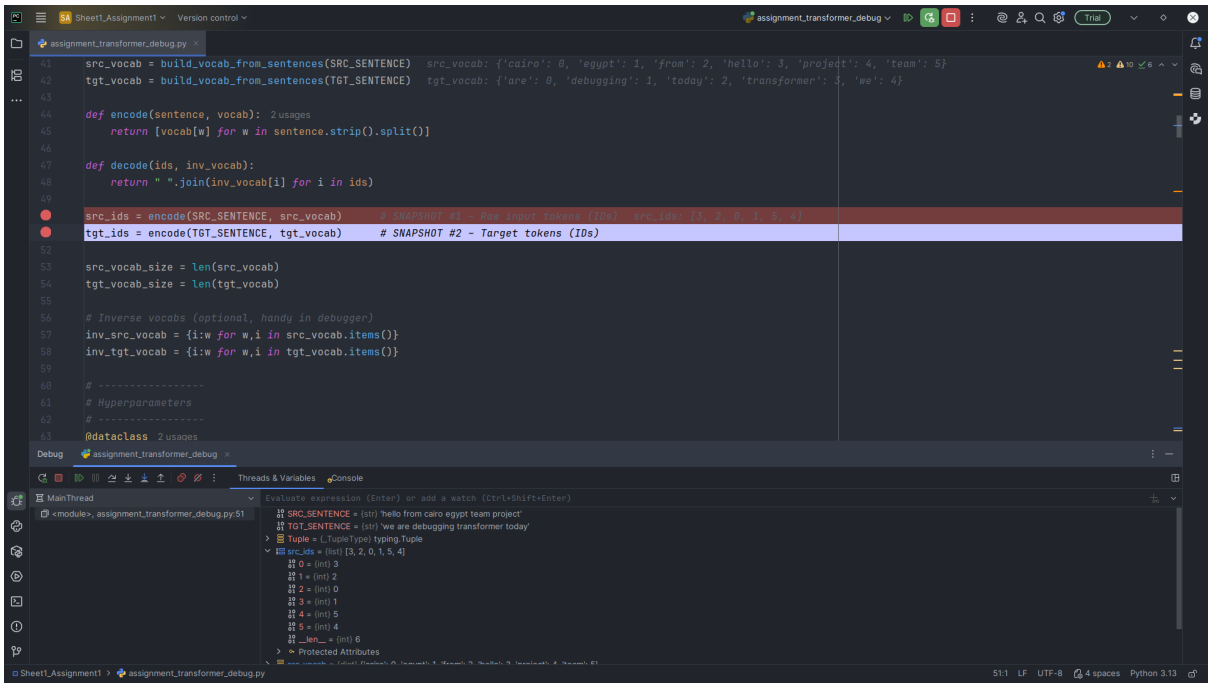


Figure 1: PyCharm interpreter configuration on Linux (instead of WSL).

Input & Embedding (Snapshots 1–5)

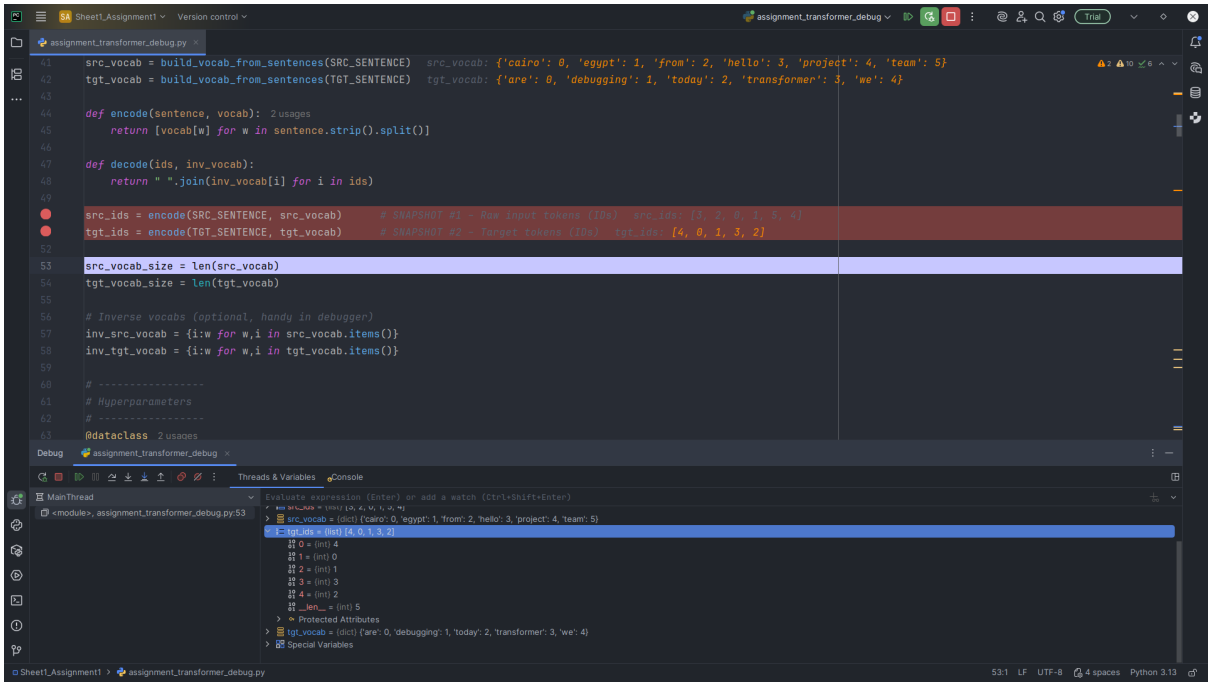


Snapshot #1 — Raw input tokens (IDs)

Explanation & Shape

Variable (as shown in PyCharm): `src_ids`
Observed shape: (6,)

Integer IDs for each source token; length equals the source sequence length (6). This is the discrete input to the embedding lookup.

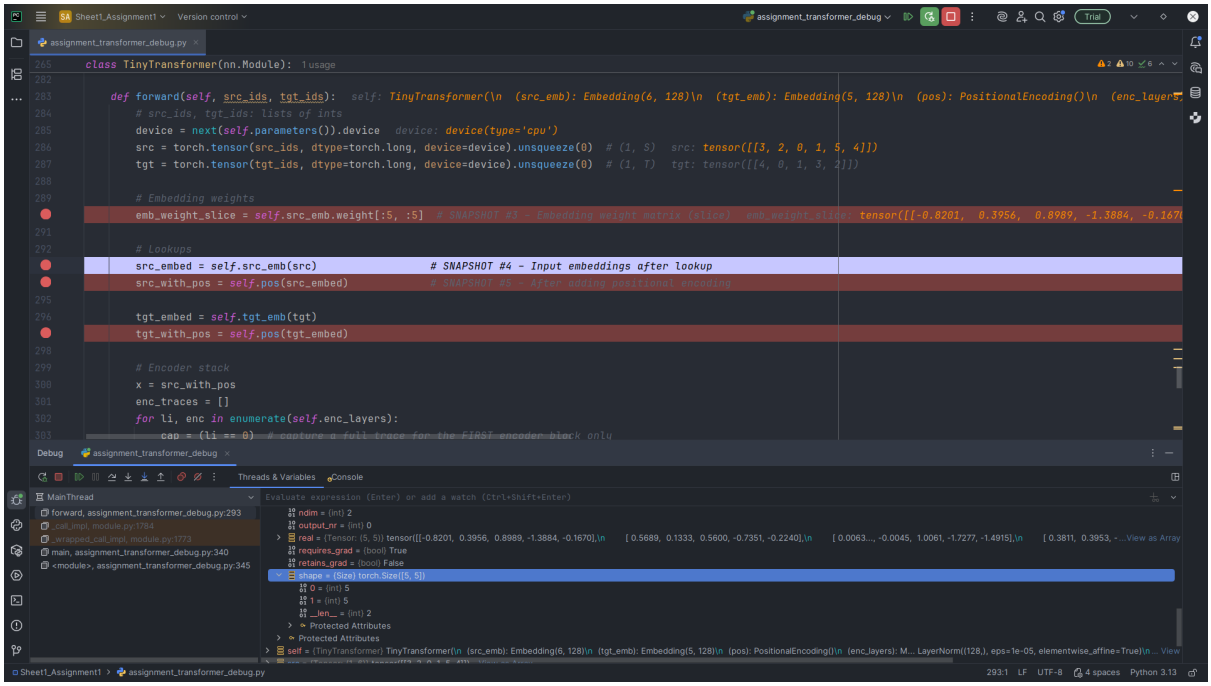


Snapshot #2 — Target tokens (IDs)

Explanation & Shape

Variable (as shown in PyCharm): `tgt_ids`
Observed shape: (5,)

Integer IDs for each target token; length equals the target sequence length (5) used for teacher forcing.

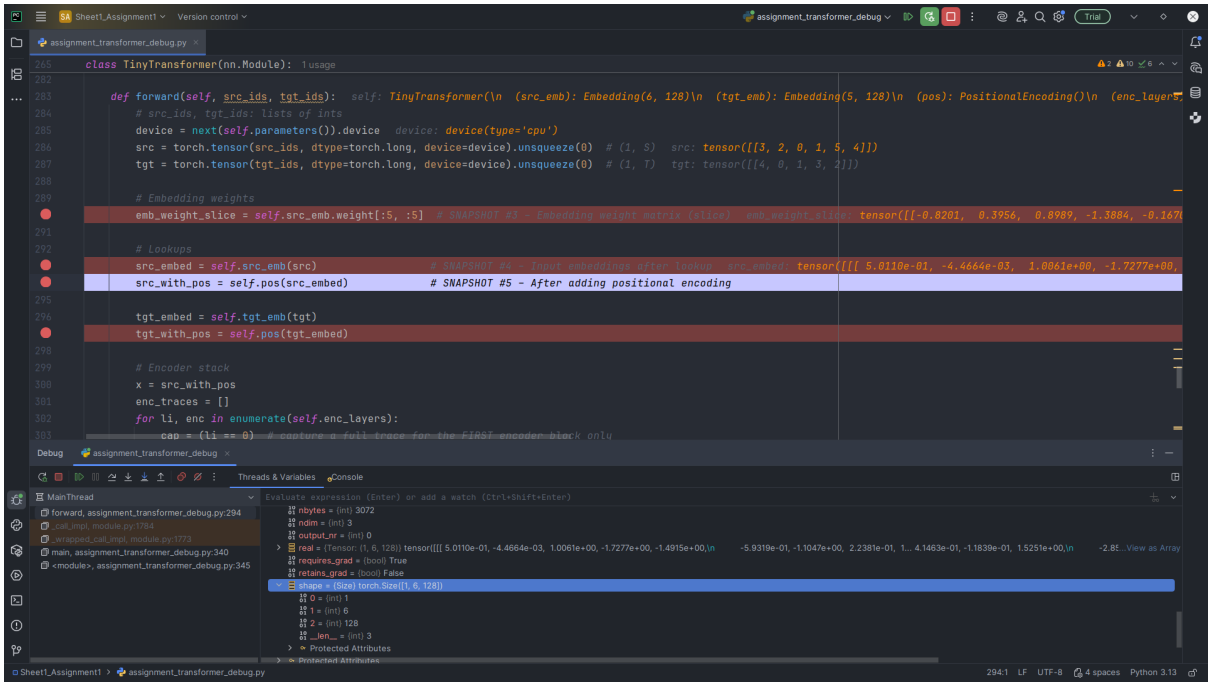


Snapshot #3 — Embedding weight matrix (slice)

Explanation & Shape

Variable (as shown in PyCharm): `refs["emb_weight_slice"]`
Observed shape: (5,5) slice of full (src_vocab_size, 128)

A small slice of the learnable embedding table. Rows = vocabulary items; columns = 128 embedding features. The full table is (vocab, 128).

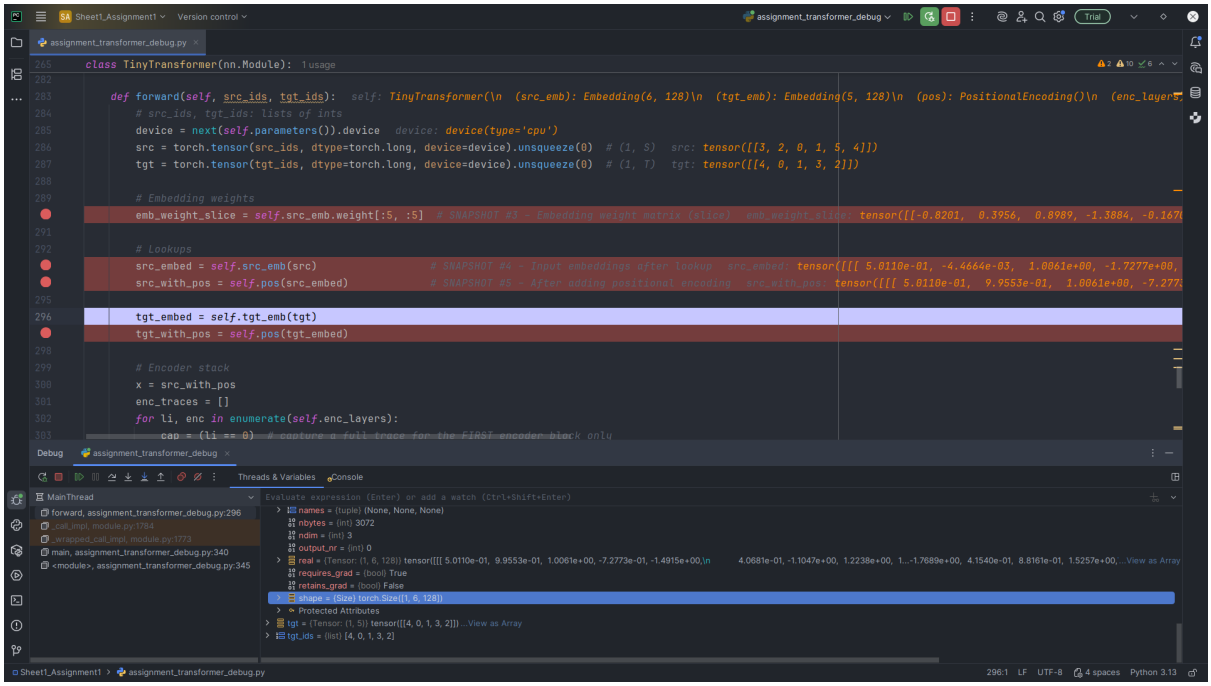


Snapshot #4 — Input embeddings after lookup

Explanation & Shape

Variable (as shown in PyCharm): `refs["src_embed"]`
Observed shape: (1,6,128)

Each source token ID is mapped to a 128-d vector. Batch size is 1, sequence length is 6, and $d_{\text{model}} = 128$.



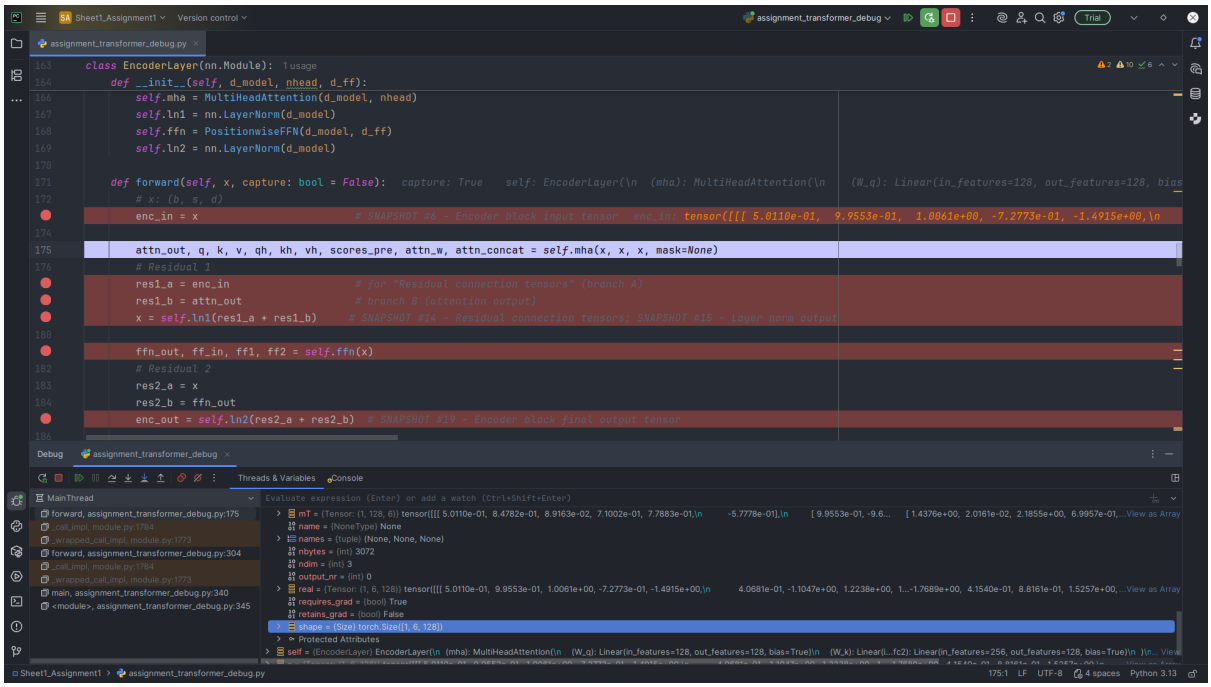
Snapshot #5 — Embeddings after adding positional encoding

Explanation & Shape

Variable (as shown in PyCharm): `refs["src_with_pos"]`
Observed shape: (1,6,128)

Sinusoidal positional encodings are added element-wise to embeddings to encode order; shape remains $(1, 6, 128)$.

Encoder (One Block, Full Trace) — Snapshots 6–19

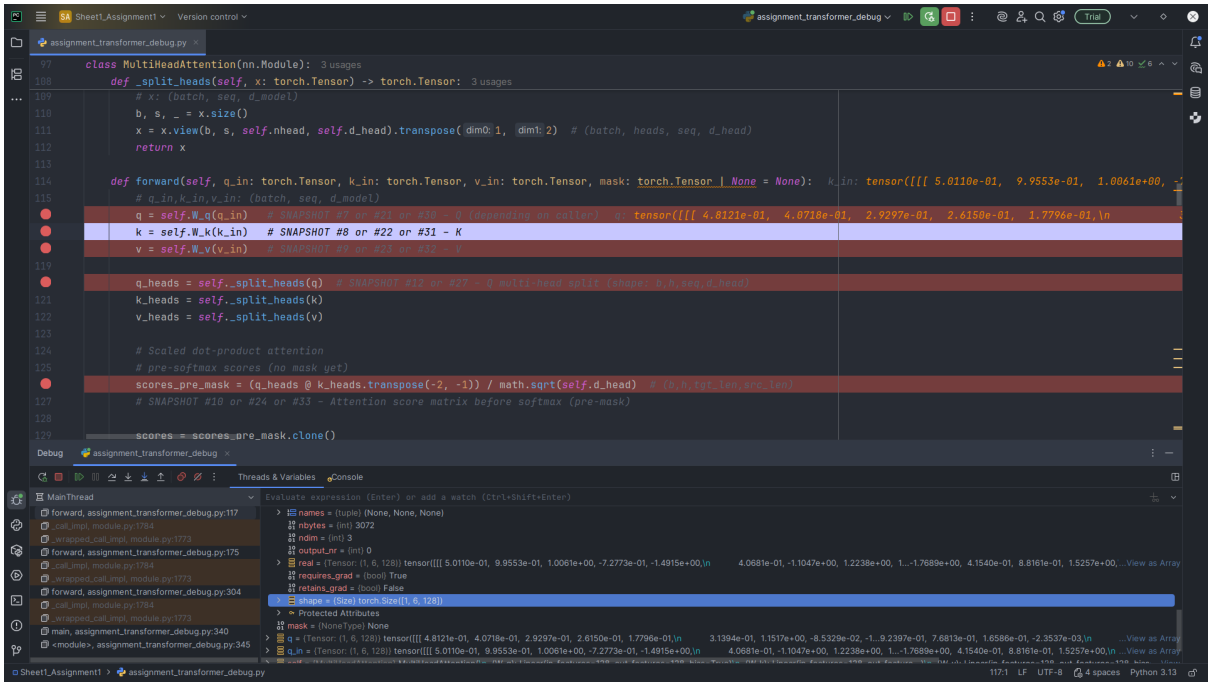


Snapshot #6 — Encoder block input tensor

Explanation & Shape

Variable (as shown in PyCharm): `refs["enc_traces"][0]["enc_in"]`
Observed shape: (1,6,128)

Tensor entering the first encoder layer (embeddings + position). Per-token dimension remains 128.

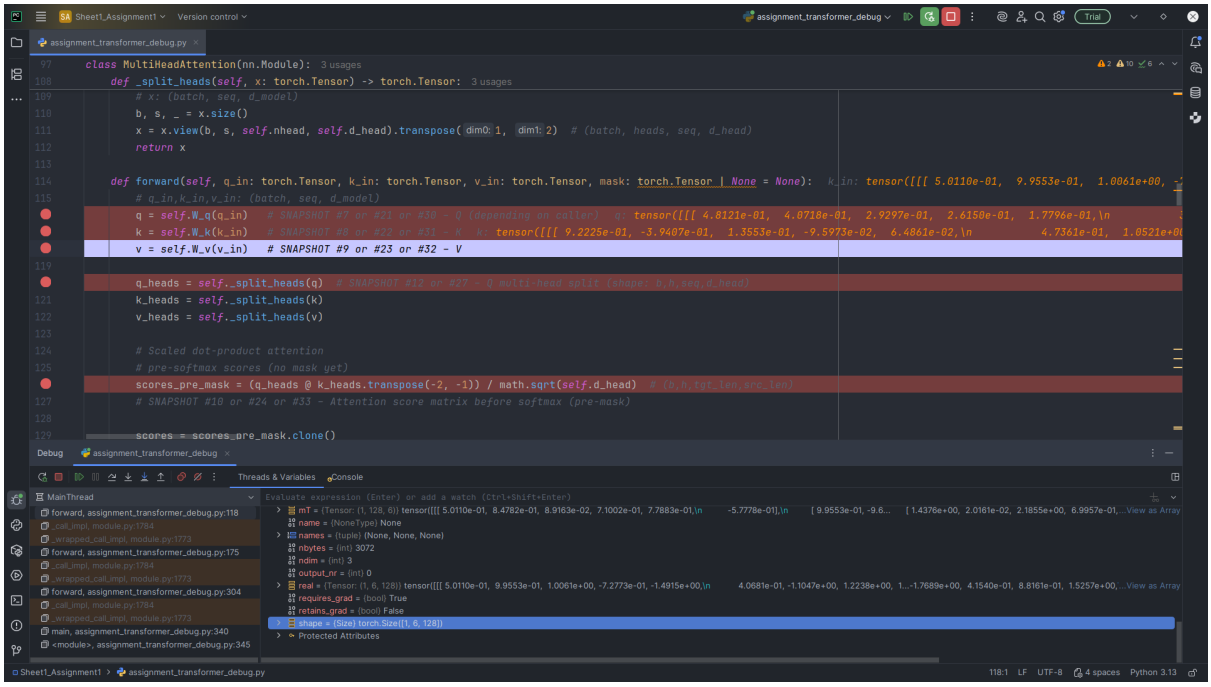


Snapshot #7 — Self-attention queries (Q)

Explanation & Shape

Variable (as shown in PyCharm): `refs["enc_traces"][0]["q"]`
Observed shape: (1,6,128)

Q is a linear projection of the encoder input. Q, K, and V share the same shape so dot-product attention is valid in the same representation space.

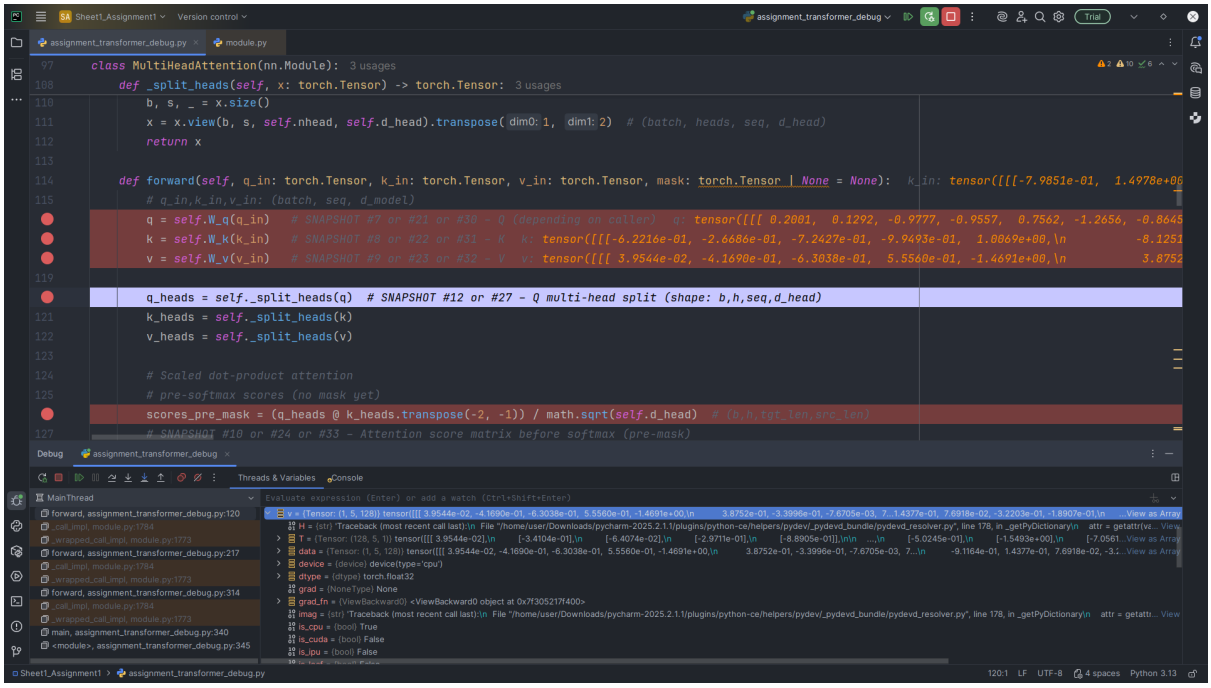


Snapshot #8 — Self-attention keys (K)

Explanation & Shape

Variable (as shown in PyCharm): `refs["enc_traces"][0]["k"]`
Observed shape: (1,6,128)

K is another linear projection representing addresses/features compared against Q. Same shape ensures $QK^\top$ is defined.

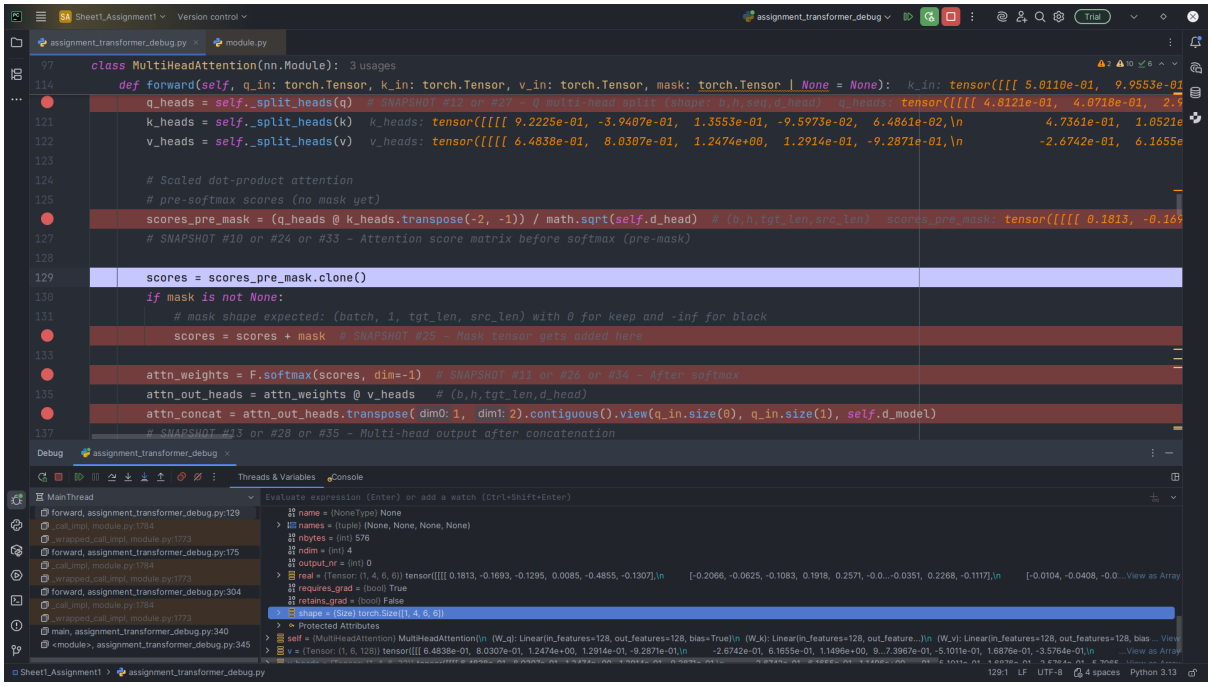


Snapshot #9 — Self-attention values (V)

Explanation & Shape

Variable (as shown in PyCharm): `refs["enc_traces"][0]["v"]`
Observed shape: (1,6,128)

V carries the features to aggregate via attention weights. Same shape keeps residuals compatible.

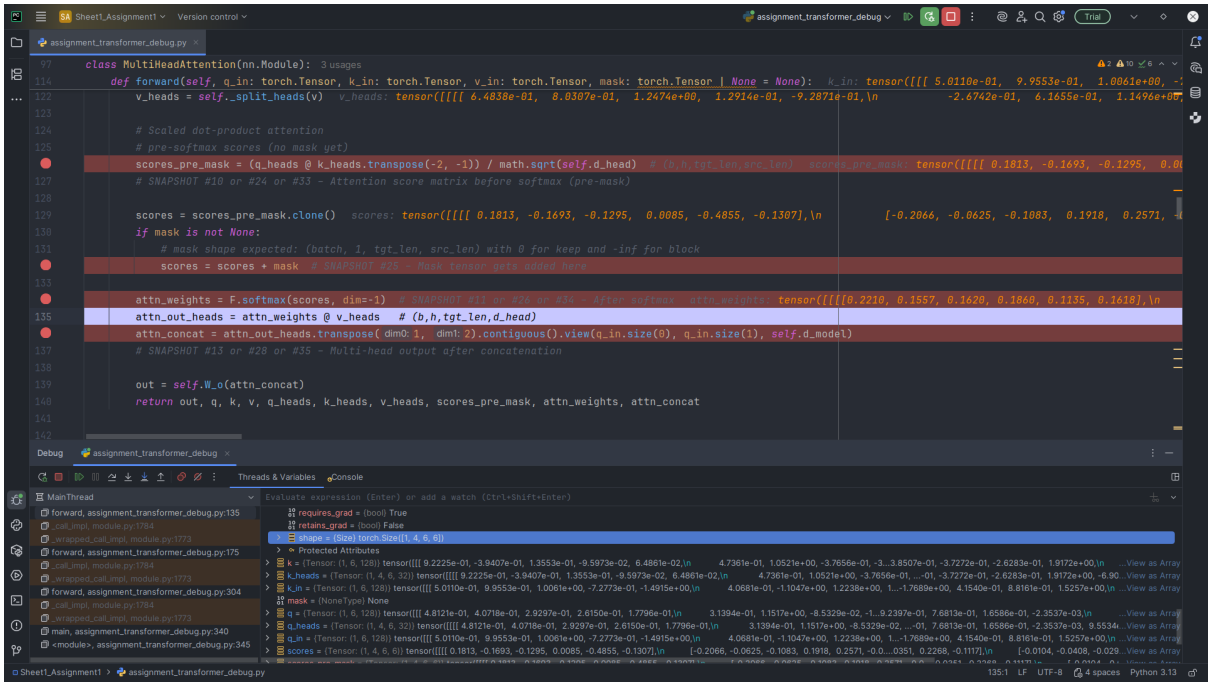


Snapshot #10 — Attention scores before softmax

Explanation & Shape

Variable (as shown in PyCharm): `refs["enc_traces"][0]["scores_pre_mask"]`
Observed shape: (1,4,6,6)

Per-head scaled dot products $QK^\top/\sqrt{d_{\text{head}}}$. Square 6×6 because encoder self-attends over the same sequence.

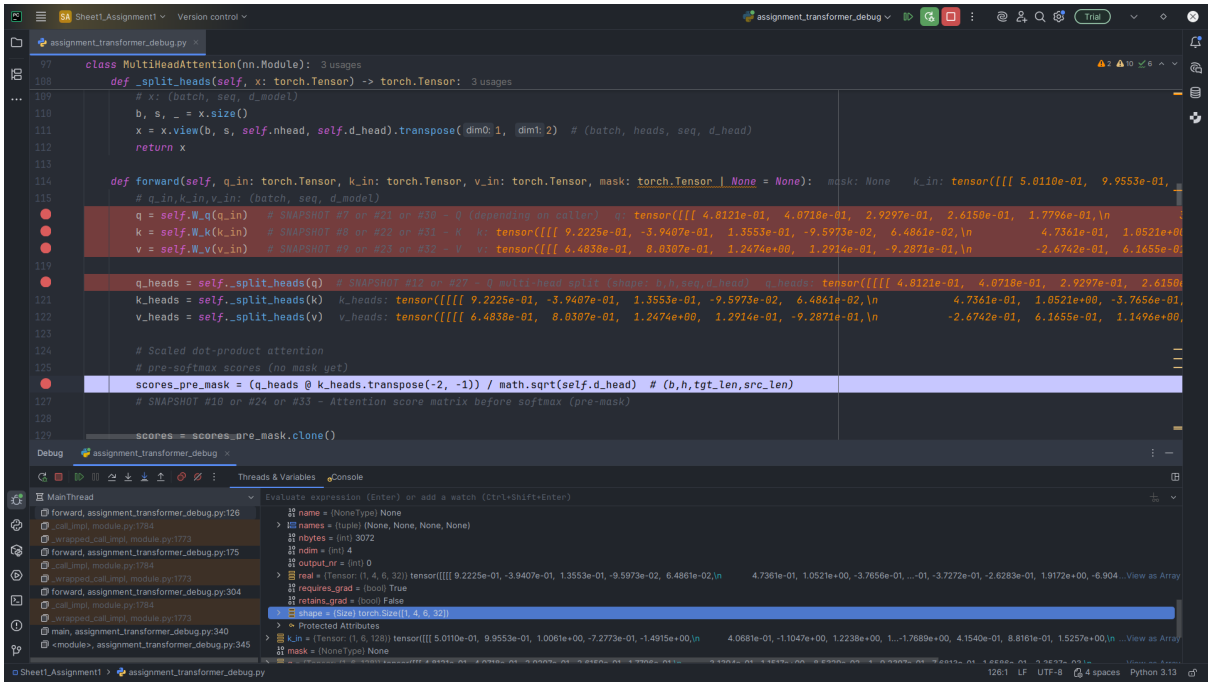


Snapshot #11 — Attention scores after softmax

Explanation & Shape

Variable (as shown in PyCharm): `refs["enc_traces"][0]["attn_weights"]`
Observed shape: (1,4,6,6)

Row-wise probabilities per head; each row sums to ≈ 1. These weights mix V.

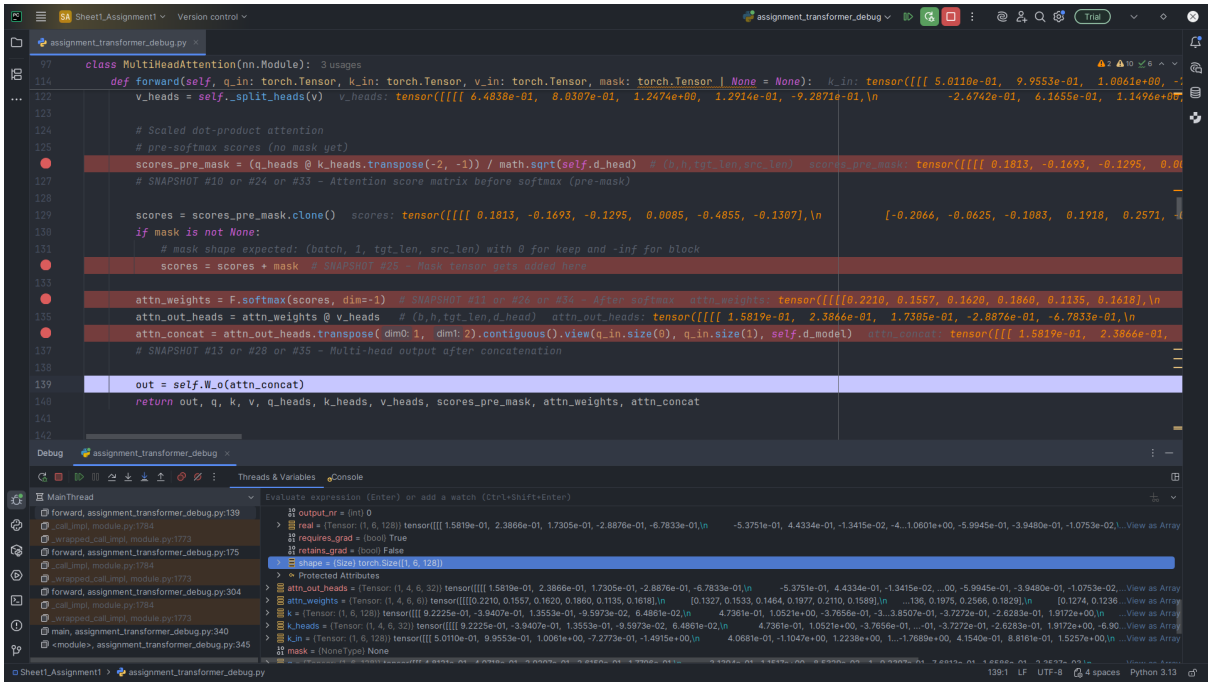


Snapshot #12 — Multi-head split (Q/K/V)

Explanation & Shape

Variable (as shown in PyCharm): `refs["enc_traces"][0]["q_heads"]` (& k_heads, v_heads)
Observed shape: (1,4,6,32) each

Split $d_{\text{model}} = 128$ into 4 heads of 32 to capture diverse relations in parallel.

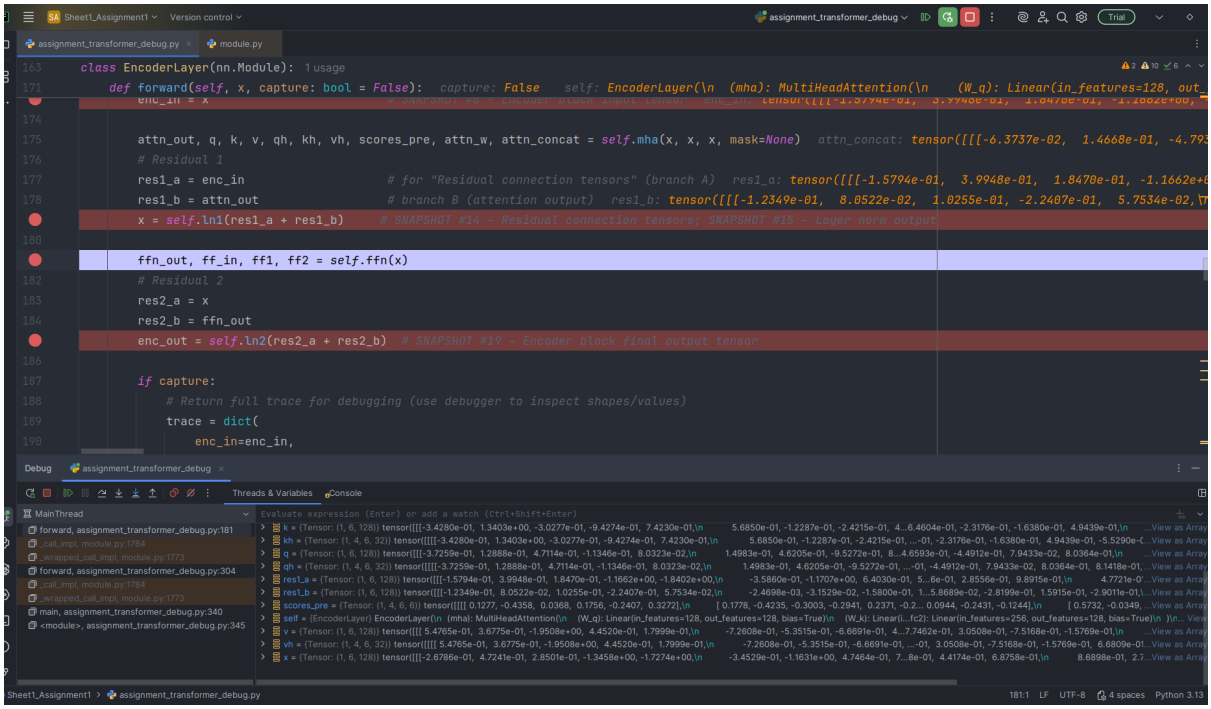


Snapshot #13 — Multi-head attention output after concatenation

Explanation & Shape

Variable (as shown in PyCharm): `refs["enc_traces"][0]["attn_concat"]`
Observed shape: (1,6,128)

Per-head outputs $(1, 4, 6, 32)$ are concatenated to $(1, 6, 128)$, restoring model dimension for residual addition.

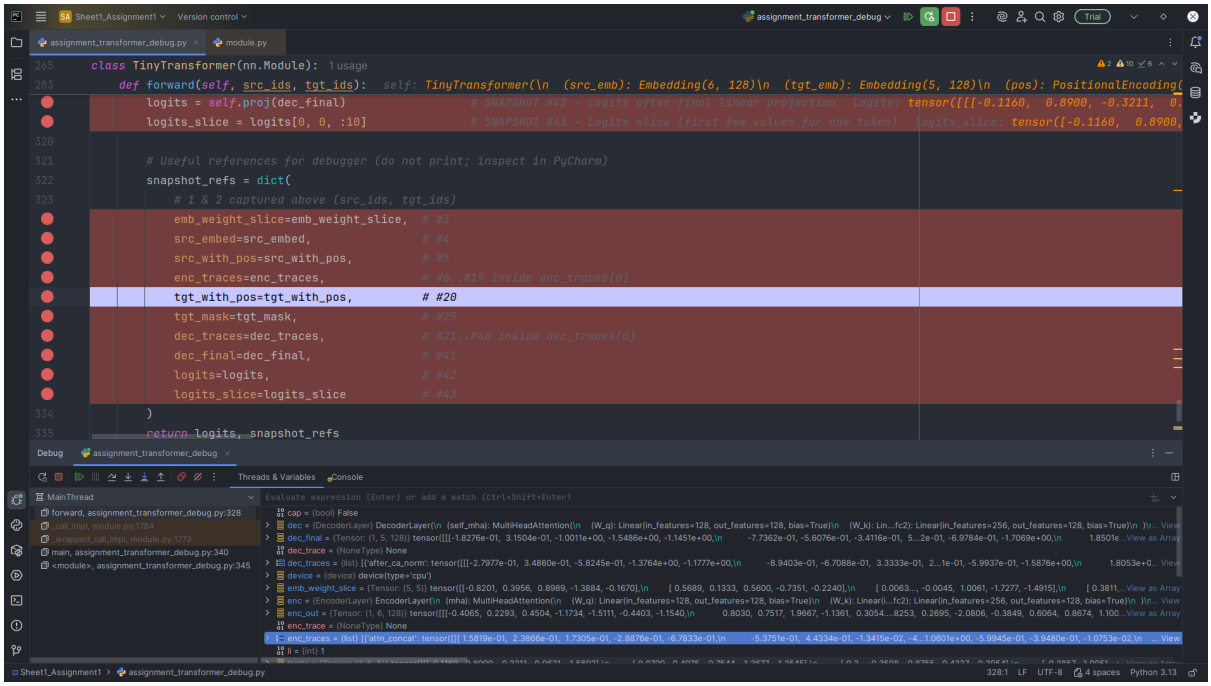


Snapshot #14 — Residual connection tensors

Explanation & Shape

Variable (as shown in PyCharm): `refs["enc_traces"][0]["res1_a"]`, `refs["enc_traces"][0]["res1_b"]`
Observed shape: (1,6,128) each

Residual adds attention output to the input tensor element-wise; identical shapes ensure valid addition and stable gradients.

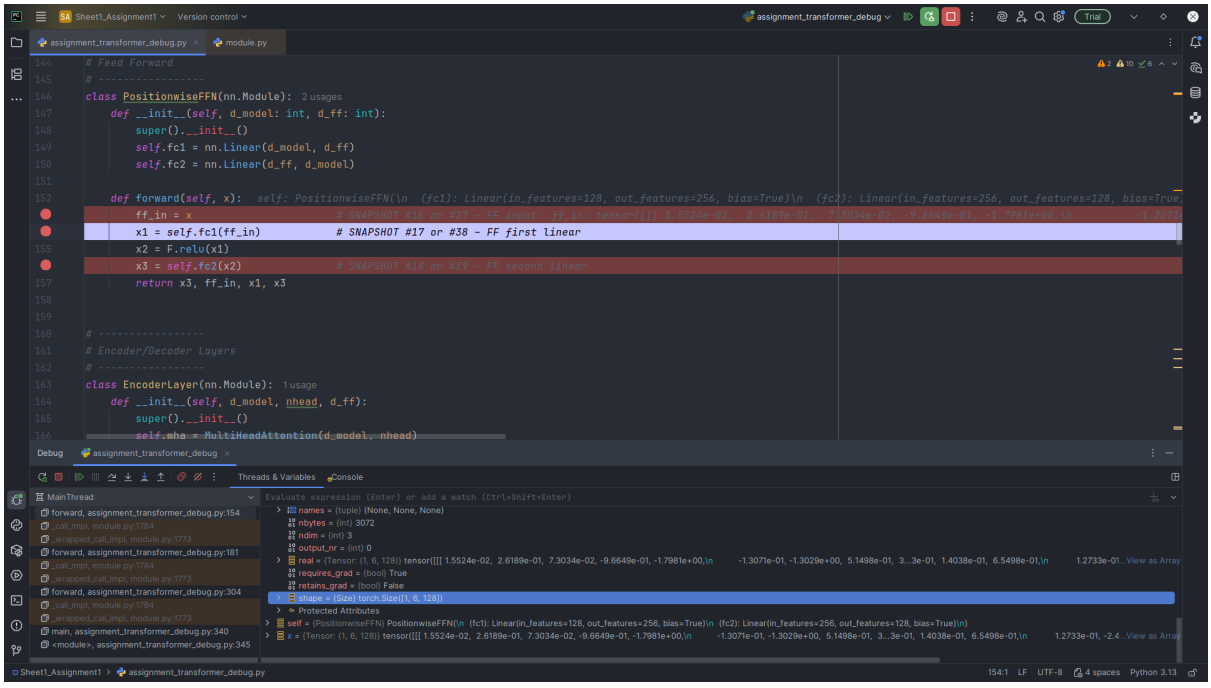


Snapshot #15 — Layer normalization output

Explanation & Shape

Variable (as shown in PyCharm): `refs["enc_traces"][0]["ln1_out"]`
Observed shape: (1,6,128)

LayerNorm re-scales the residual sum per feature without changing shape, stabilizing training.

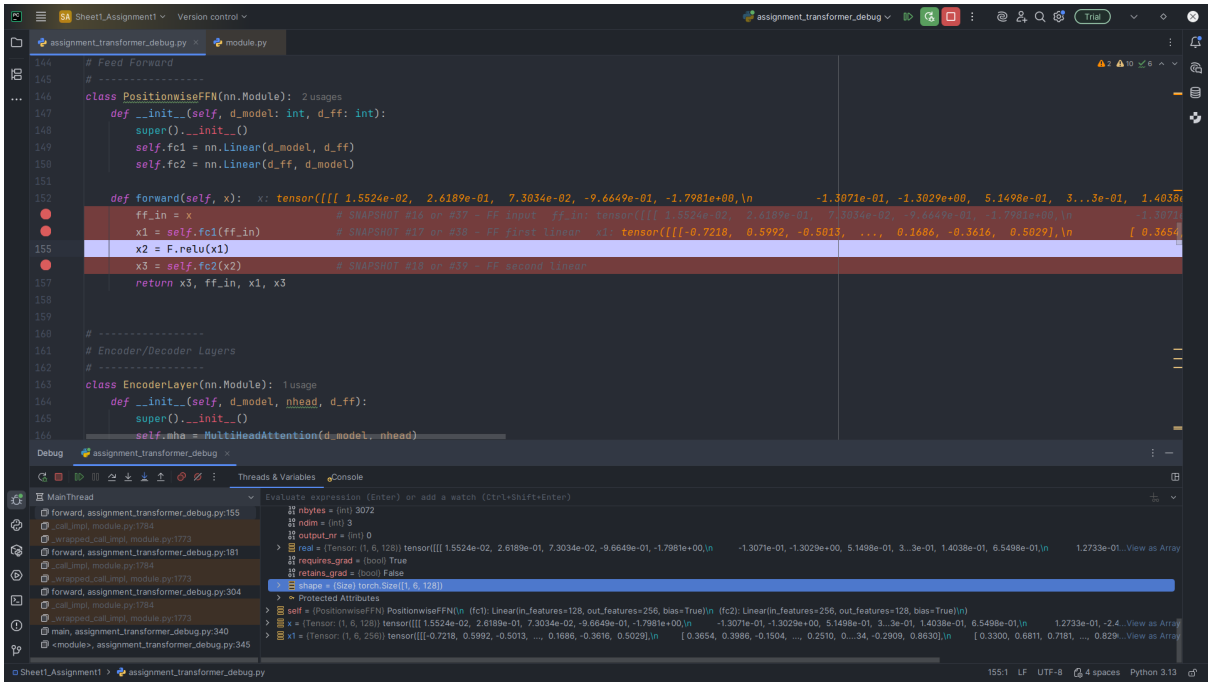


Snapshot #16 — Feed-forward input

Explanation & Shape

Variable (as shown in PyCharm): `refs["enc_traces"][0]["ff_in"]`
Observed shape: (1,6,128)

Normalized token states feed the position-wise FFN; $d_{\text{model}} = 128$ is preserved for residuals.

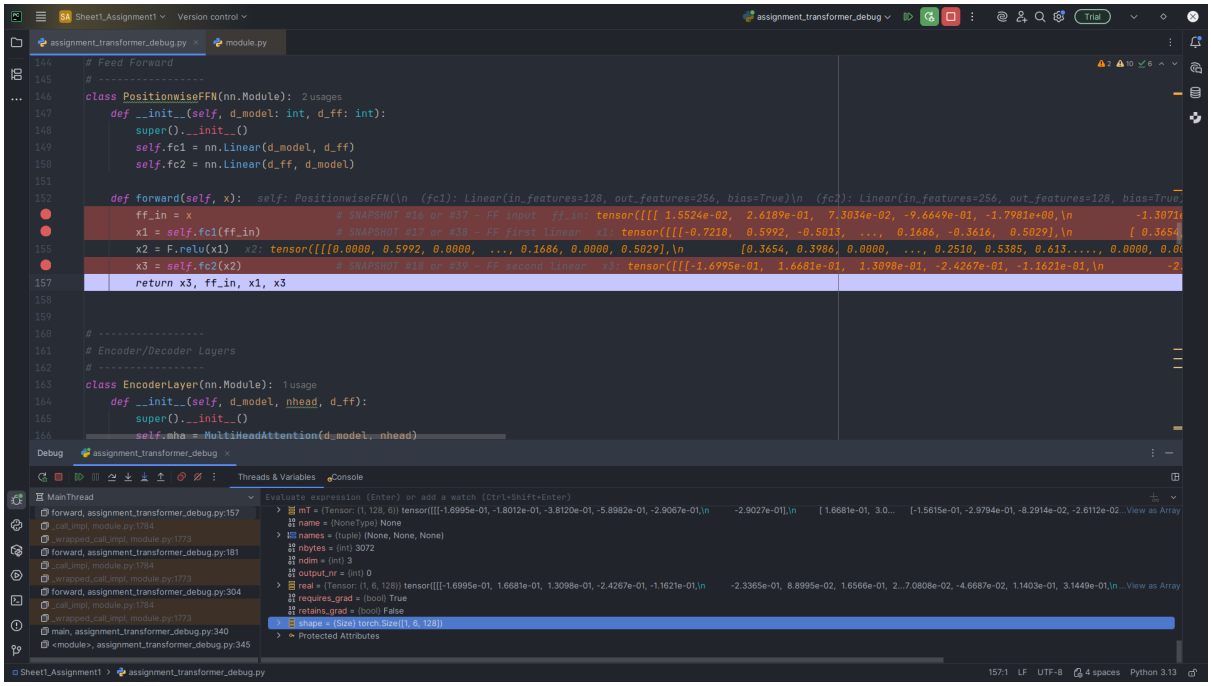


Snapshot #17 — Feed-forward first linear layer output

Explanation & Shape

Variable (as shown in PyCharm): `refs["enc_traces"][0]["ff1"]`
Observed shape: (1,6,256)

FFN expands to $d_{\text{ff}} = 256$, increasing per-token capacity before projecting back.

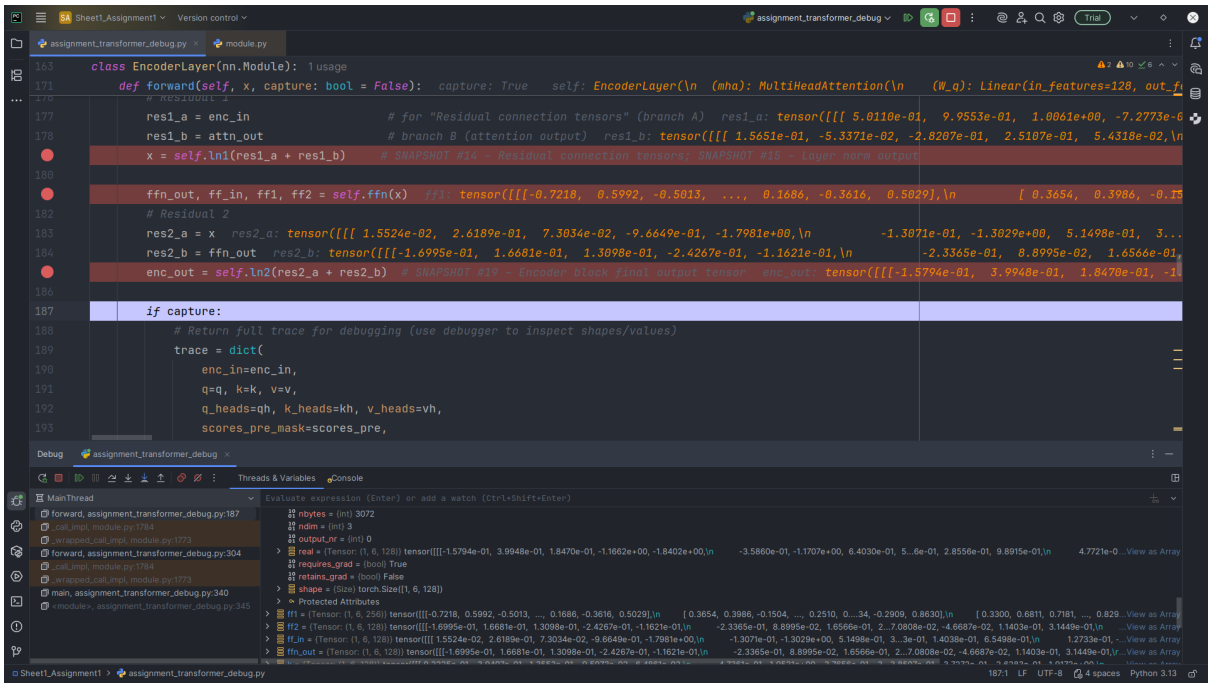


Snapshot #18 — Feed-forward second linear layer output

Explanation & Shape

Variable (as shown in PyCharm): `refs["enc_traces"][0]["ff2"]`
Observed shape: (1,6,128)

Projects back to 128 so it can be added (residual) to the FFN input.



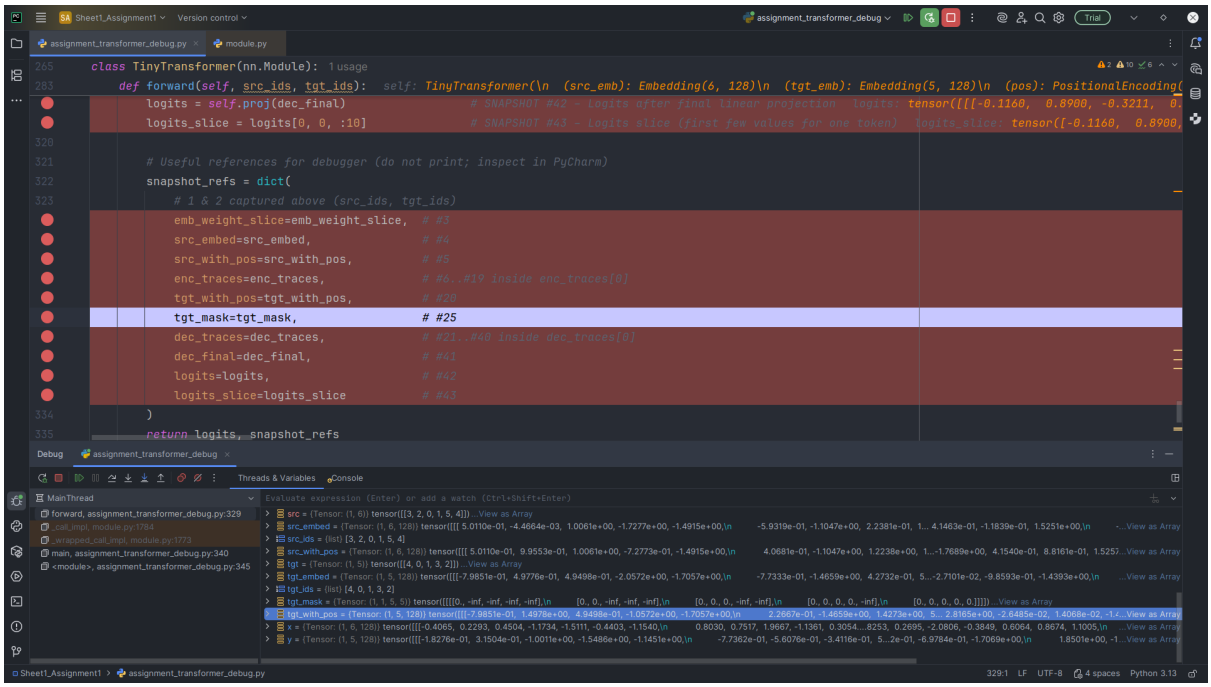
Snapshot #19 — Encoder block final output tensor

Explanation & Shape

Variable (as shown in PyCharm): `refs["enc_traces"][0]["enc_out"]`
Observed shape: (1,6,128)

After FFN residual + LayerNorm, this output feeds later encoder layers and decoder cross-attention.

Decoder (One Block, Full Trace) — Snapshots 20–40

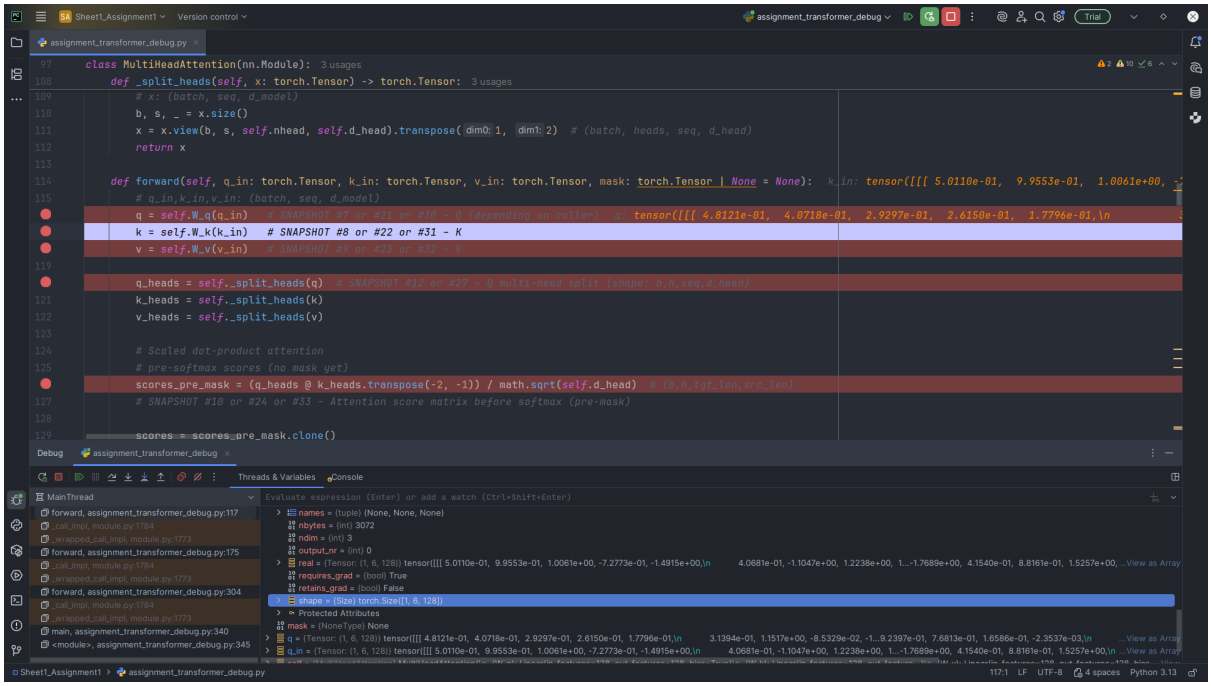


Snapshot #20 — Decoder block input tensor

Explanation & Shape

Variable (as shown in PyCharm): `refs["tgt_with_pos"]`
Observed shape: (1,5,128)

Target embeddings + positional encodings (teacher forcing). Target sequence length is 5.

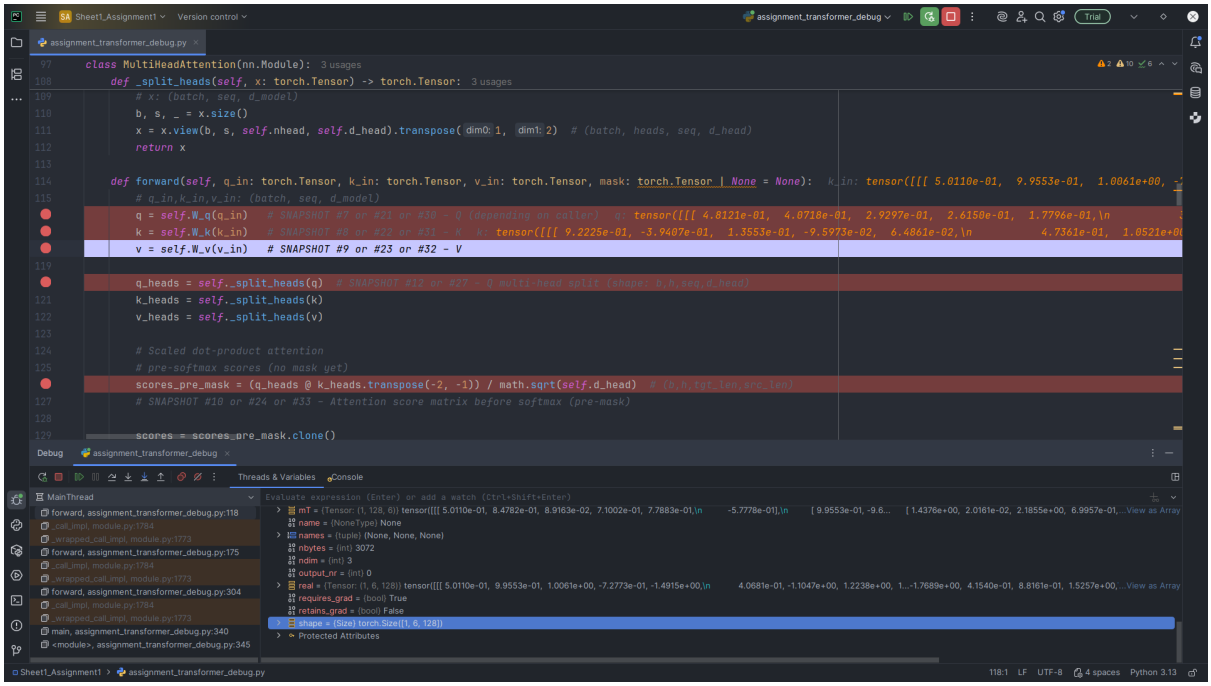


Snapshot #21 — Masked self-attention queries (Q)

Explanation & Shape

Variable (as shown in PyCharm): `refs["dec_traces"][0]["q_m"]`
Observed shape: (1,5,128)

Decoder Q is a linear projection of decoder inputs; same shape as K,V to enable attention.

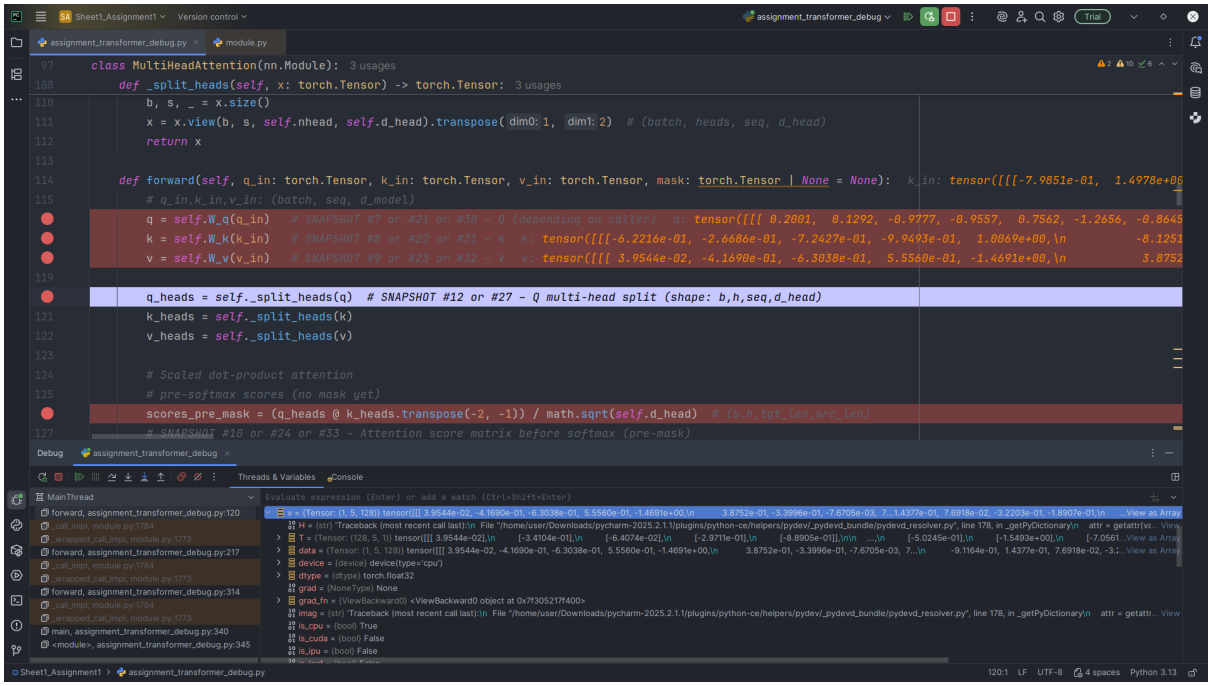


Snapshot #22 — Masked self-attention keys (K)

Explanation & Shape

Variable (as shown in PyCharm): `refs["dec_traces"][0]["k_m"]`
Observed shape: (1,5,128)

Decoder K shares the same shape so $QK^\top$ is valid across time steps.

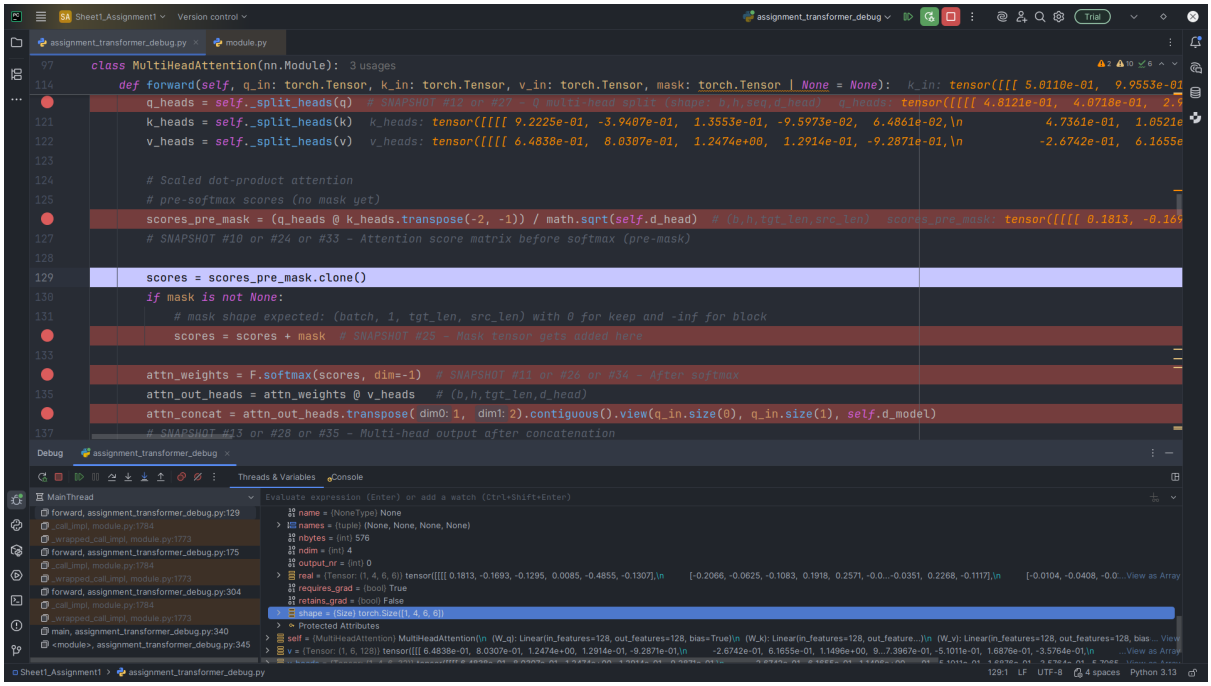


Snapshot #23 — Masked self-attention values (V)

Explanation & Shape

Variable (as shown in PyCharm): `refs["dec_traces"][0]["v_m"]`
Observed shape: (1,5,128)

V provides features to be aggregated via masked attention weights.

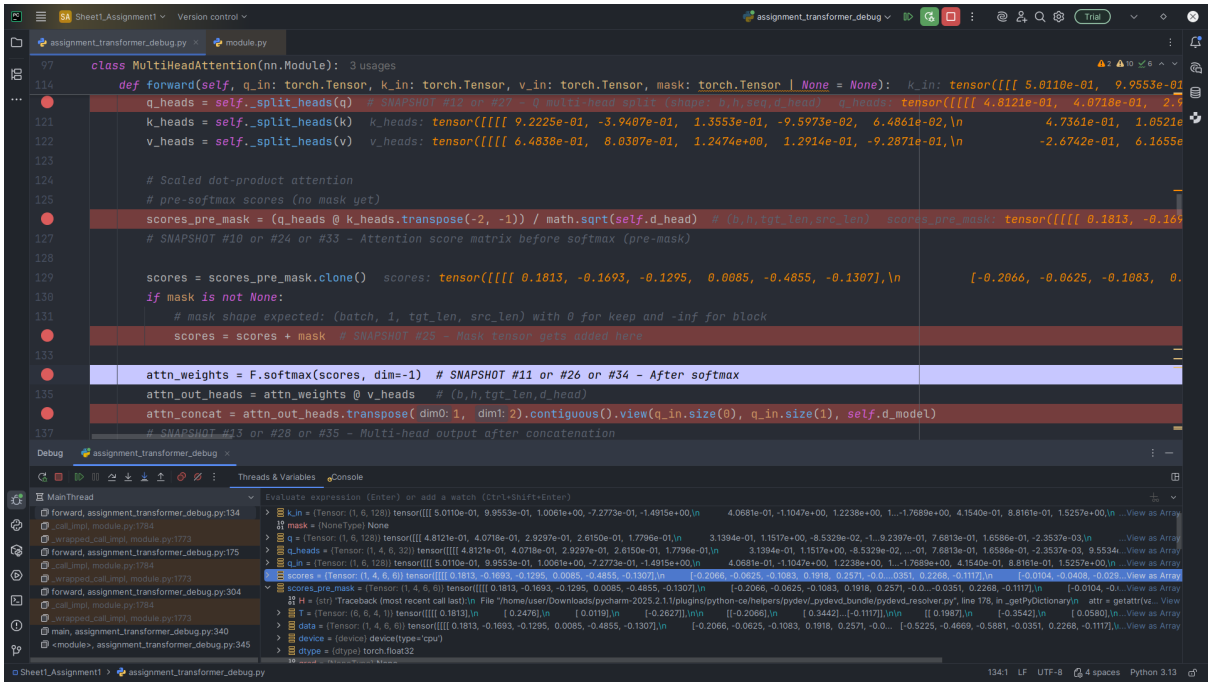


Snapshot #24 — Masked attention scores before mask

Explanation & Shape

Variable (as shown in PyCharm): `refs["dec_traces"][0]["scores_pre_mask_m"]`
Observed shape: (1,4,5,5)

Raw per-head scores before masking; square because masked self-attention is over the target sequence.

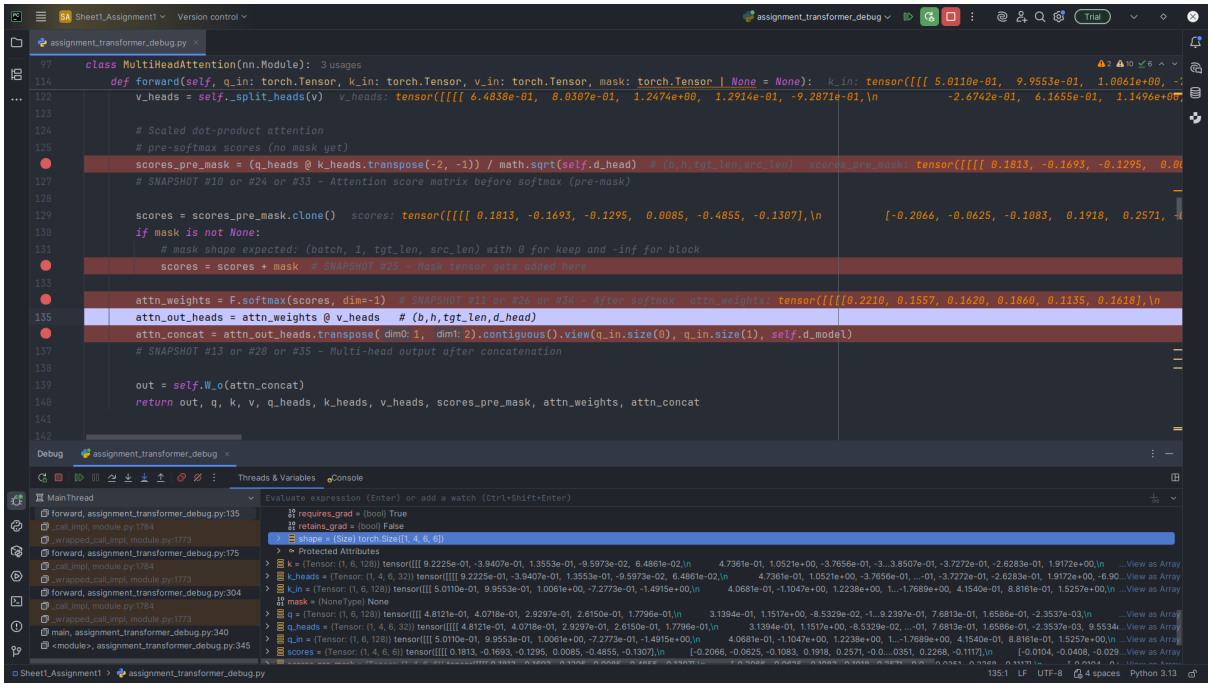


Snapshot #25 — Mask tensor

Explanation & Shape

Variable (as shown in PyCharm): `refs["tgt_mask"]`
Observed shape: (1,1,5,5) with 0 and $-\infty$

Upper-triangular $-\infty$ above the diagonal prevents attention to future positions, enforcing autoregressive decoding.

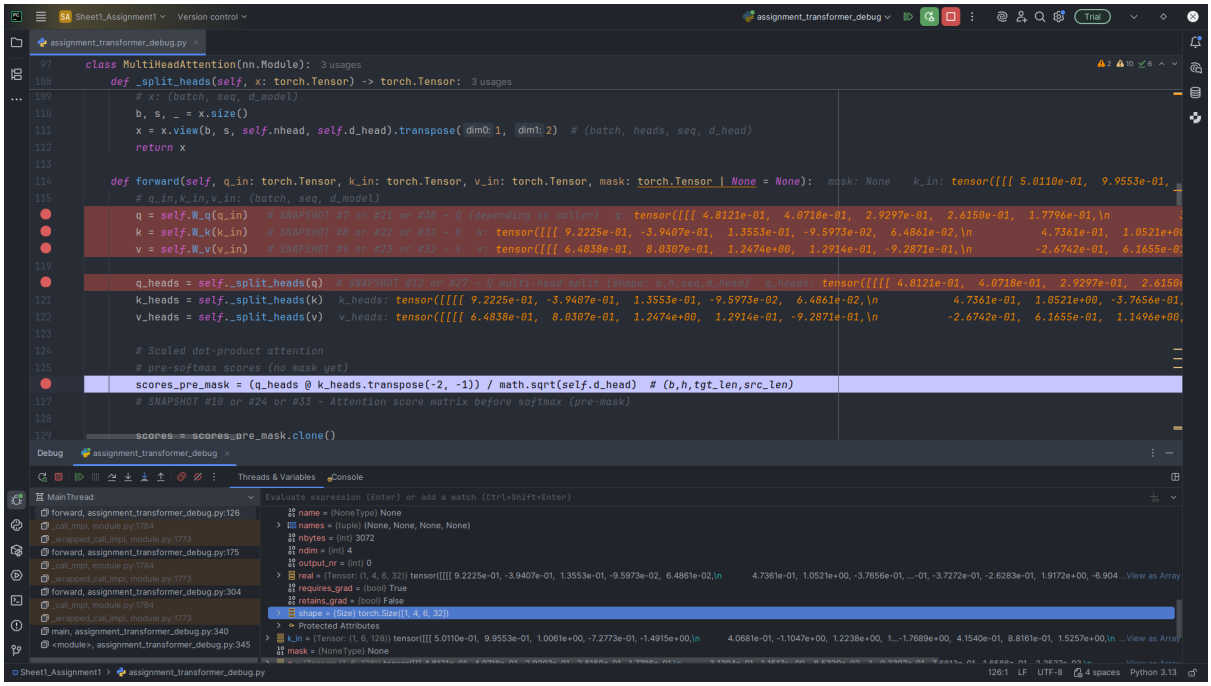


Snapshot #26 — Masked attention scores after mask + softmax

Explanation & Shape

Variable (as shown in PyCharm): `refs["dec_traces"][0]["attn_weights_m"]`
Observed shape: (1,4,5,5)

After mask + softmax, probabilities at masked positions are ≈ 0; rows sum to ≈ 1.

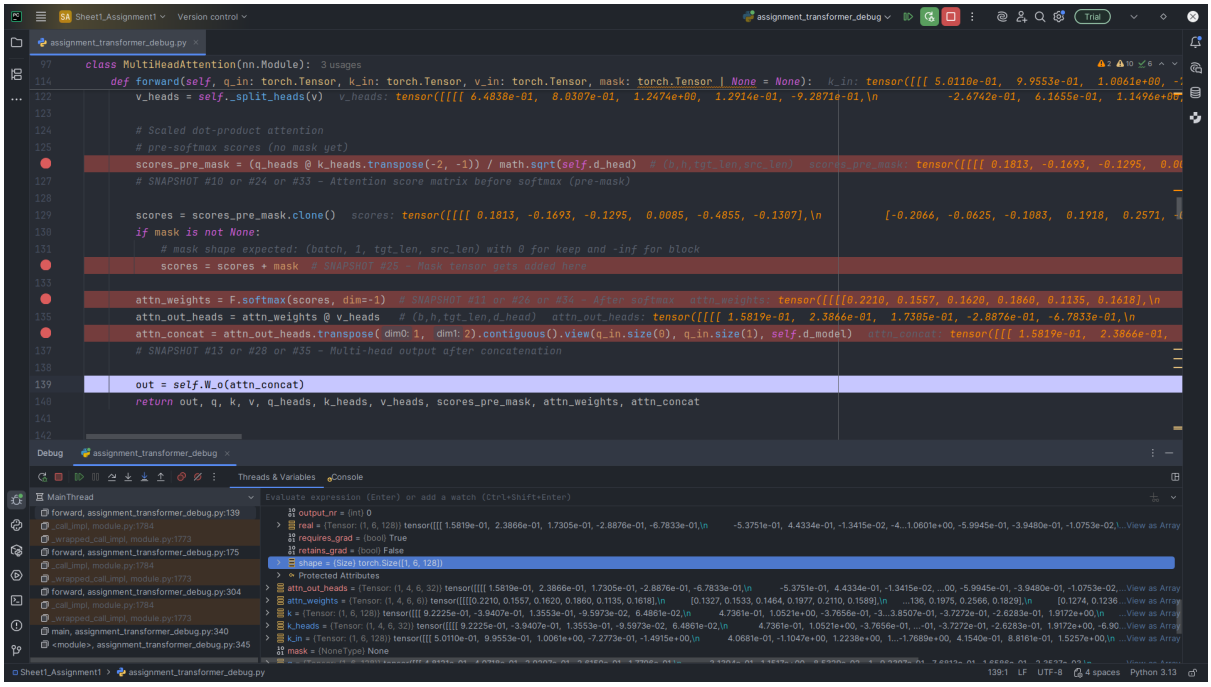


Snapshot #27 — Masked self-attention multi-head split

Explanation & Shape

Variable (as shown in PyCharm): `refs["dec_traces"][0]["q_heads_m"]` (& `k_heads_m`, `v_heads_m`)
Observed shape: (1,4,5,32) each

Splitting into 4 heads of 32 captures multiple relations per token in parallel.

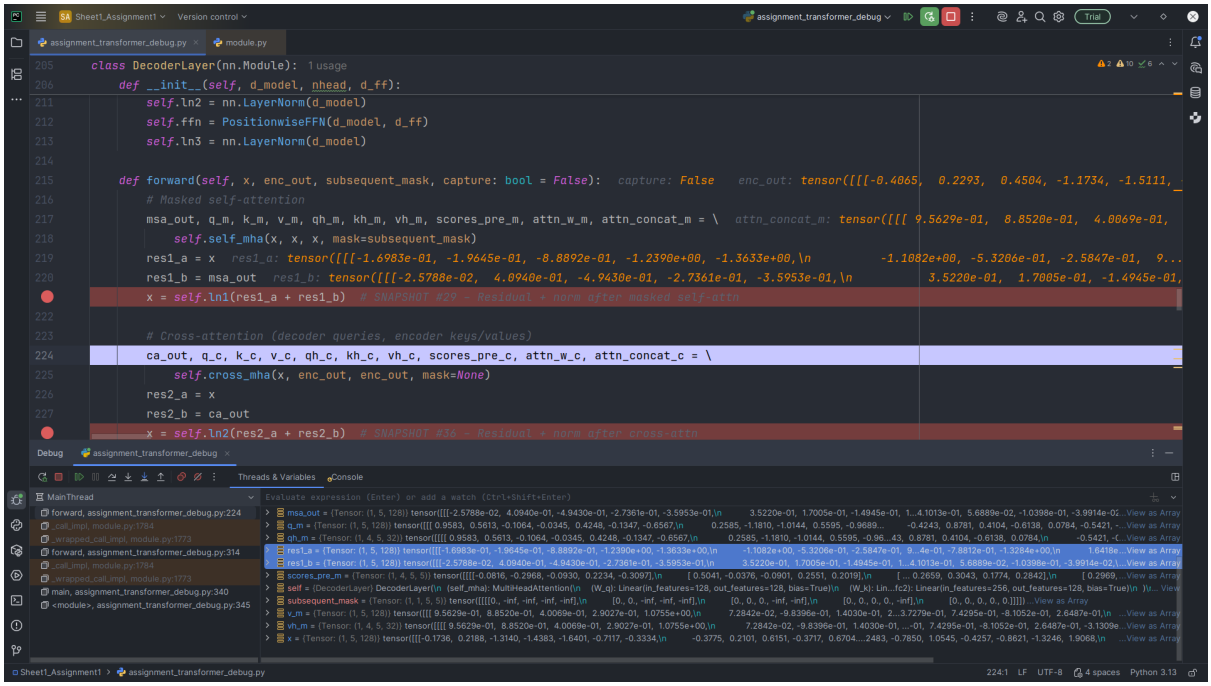


Snapshot #28 — Masked self-attention multi-head concatenated output

Explanation & Shape

Variable (as shown in PyCharm): `refs["dec_traces"][0]["attn_concat_m"]`
Observed shape: (1,5,128)

Per-head outputs are concatenated to 128 to preserve residual compatibility.

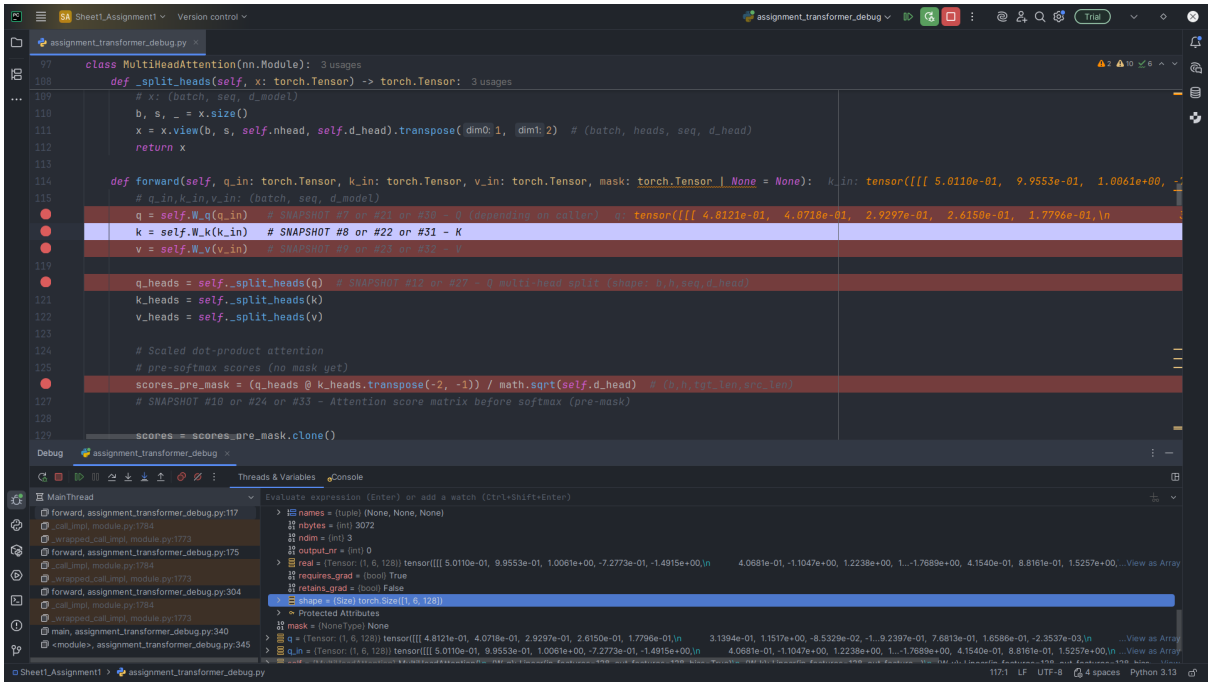


Snapshot #29 — Residual + normalization after masked self-attention

Explanation & Shape

Variable (as shown in PyCharm): `refs["dec_traces"][0]["after_msa_norm"]`
Observed shape: (1,5,128)

Residual (decoder input + masked self-attn output) followed by LayerNorm; shape unchanged.

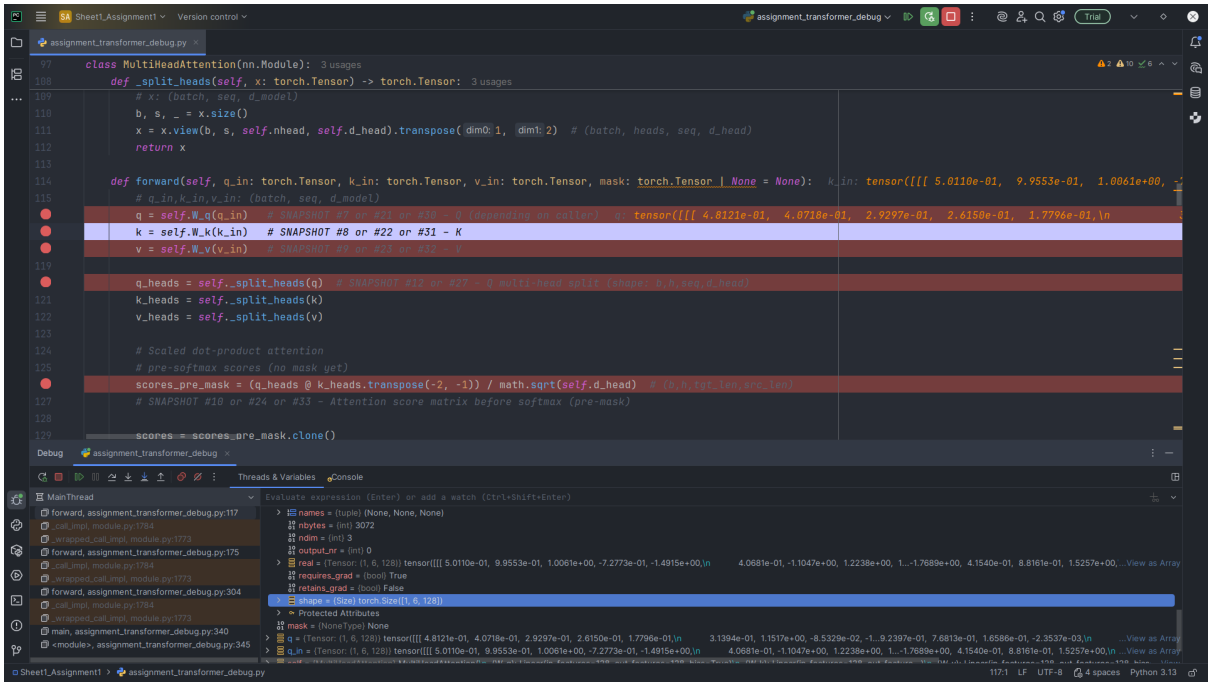


Snapshot #30 — Cross-attention queries (from decoder)

Explanation & Shape

Variable (as shown in PyCharm): `refs["dec_traces"][0]["q_c"]`
Observed shape: (1,5,128)

Queries derived from decoder states to attend over encoder outputs.

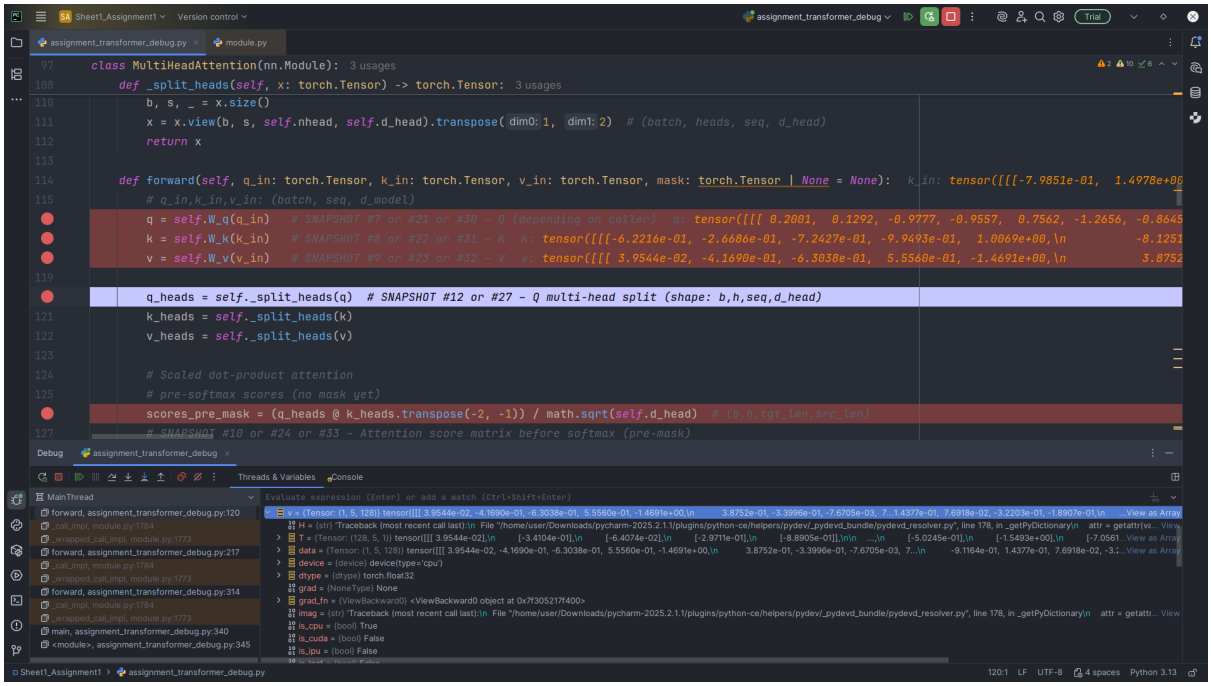


Snapshot #31 — Cross-attention keys (from encoder)

Explanation & Shape

Variable (as shown in PyCharm): `refs["dec_traces"][0]["k_c"]`
Observed shape: (1,6,128)

Keys originate from encoder outputs (src length = 6).

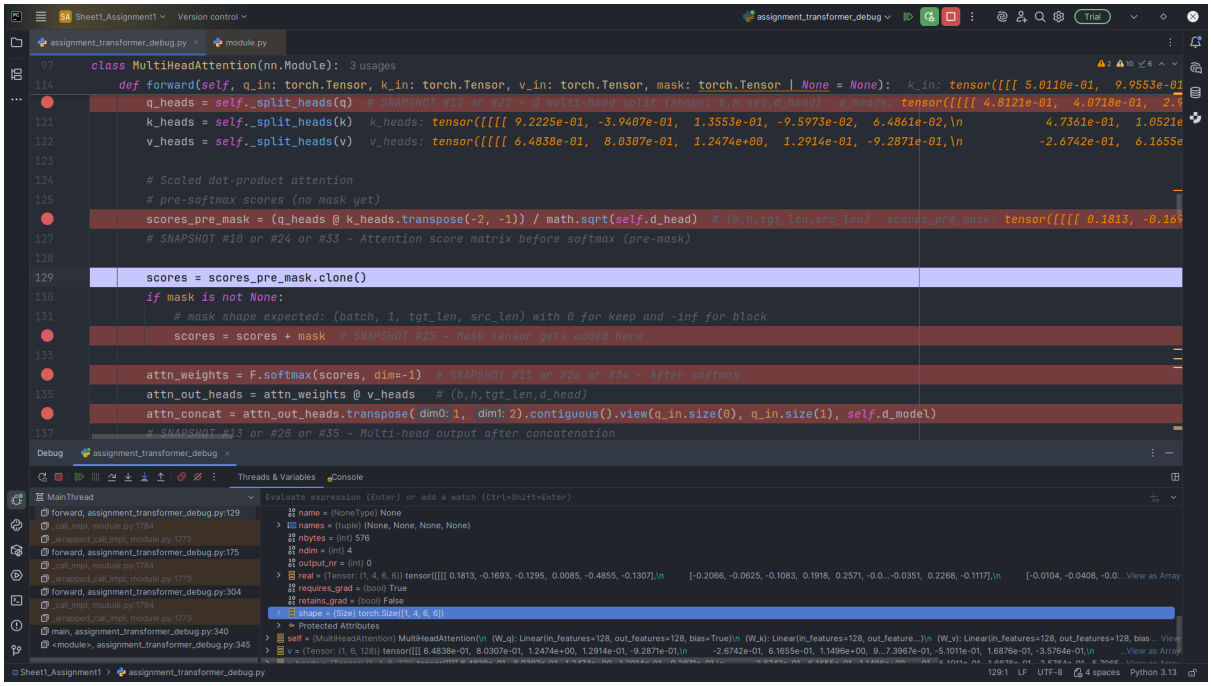


Snapshot #32 — Cross-attention values (from encoder)

Explanation & Shape

Variable (as shown in PyCharm): `refs["dec_traces"][0]["v_c"]`
Observed shape: (1,6,128)

Values also come from encoder outputs and will be aggregated into decoder states.

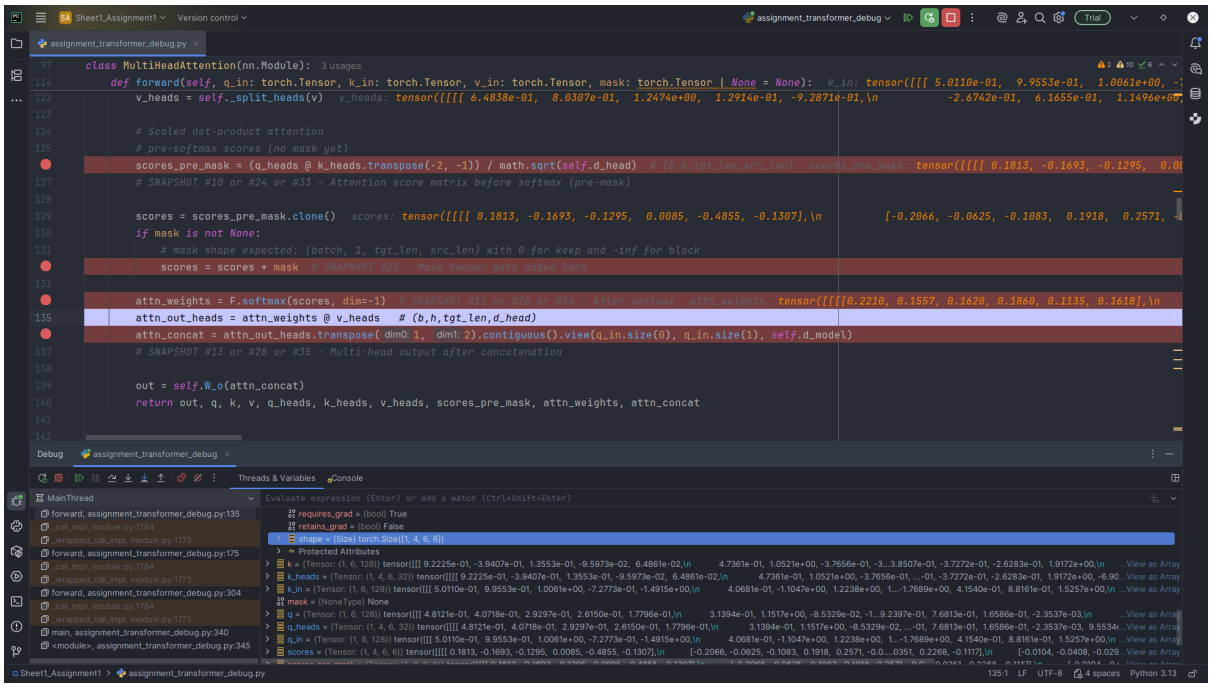


Snapshot #33 — Cross-attention score matrix before softmax

Explanation & Shape

Variable (as shown in PyCharm): `refs["dec_traces"][0]["scores_pre_mask_c"]`
Observed shape: (1,4,5,6)

Per-head scores are rectangular (tgt_len = 5 queries over src_len = 6 keys).

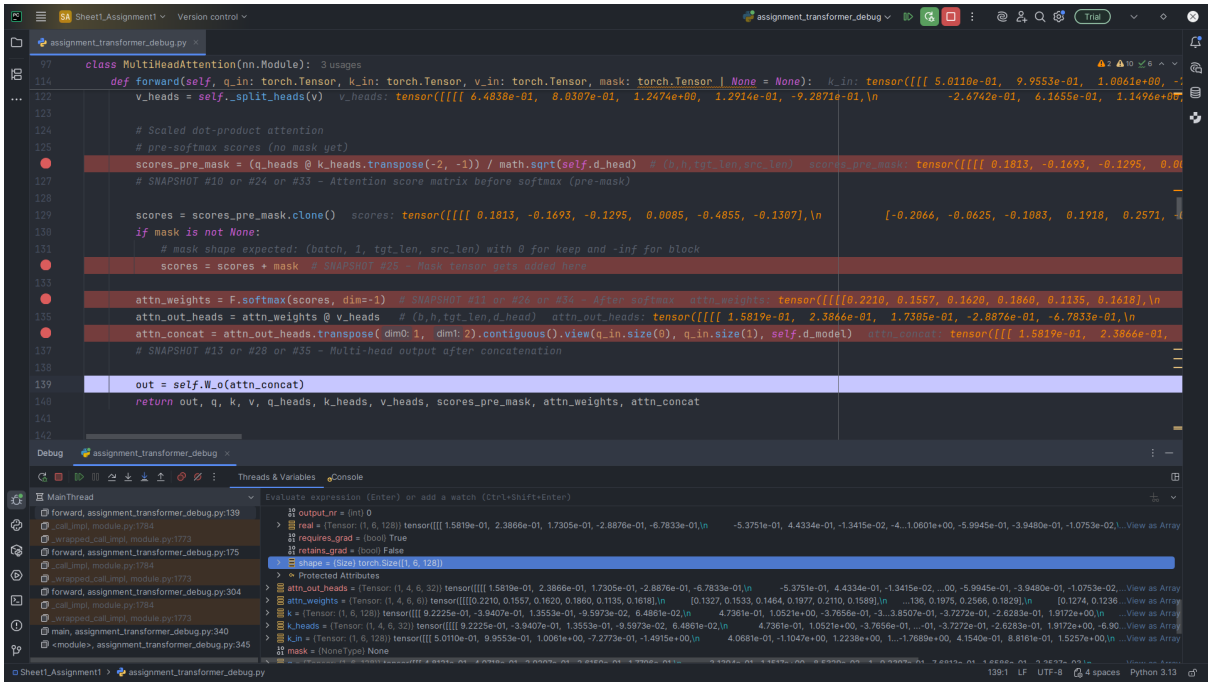


Snapshot #34 — Cross-attention score matrix after softmax

Explanation & Shape

Variable (as shown in PyCharm): `refs["dec_traces"][0]["attn_weights_c"]`
Observed shape: (1,4,5,6)

Row-wise probabilities over encoder positions; each row sums ≈ 1.

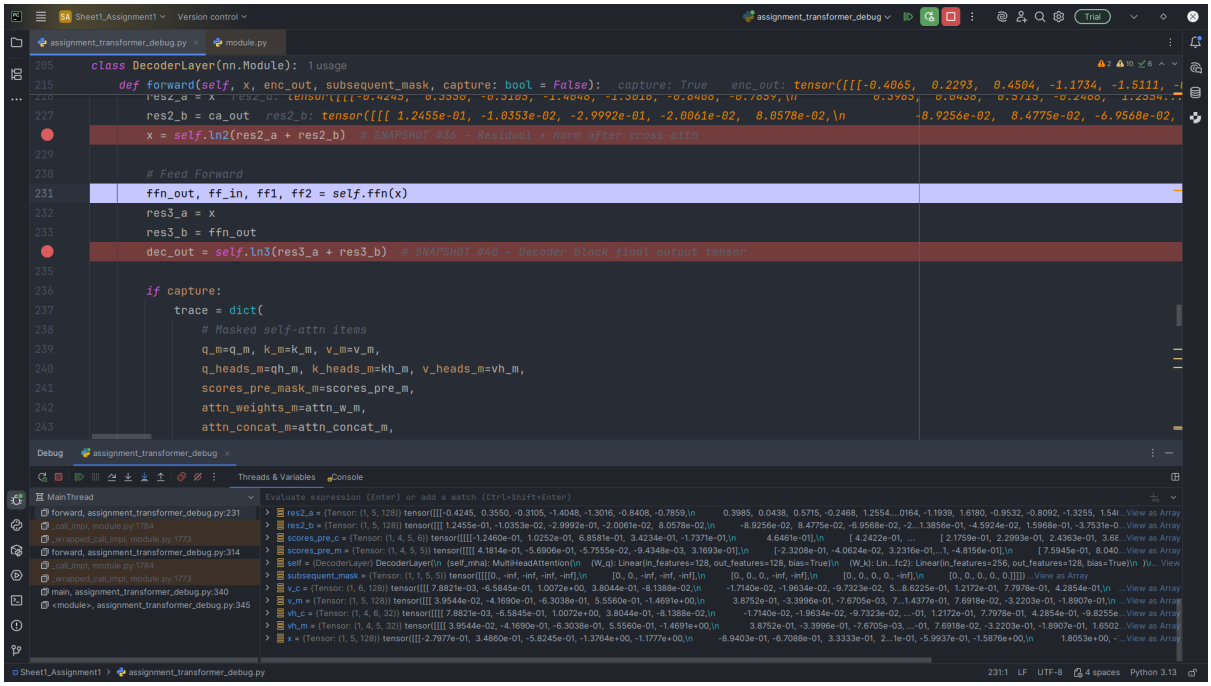


Snapshot #35 — Cross-attention output after concatenation

Explanation & Shape

Variable (as shown in PyCharm): `refs["dec_traces"][0]["attn_concat_c"]`
Observed shape: (1,5,128)

Concatenated per-head outputs, restoring $d_{\text{model}} = 128$ for residual addition.

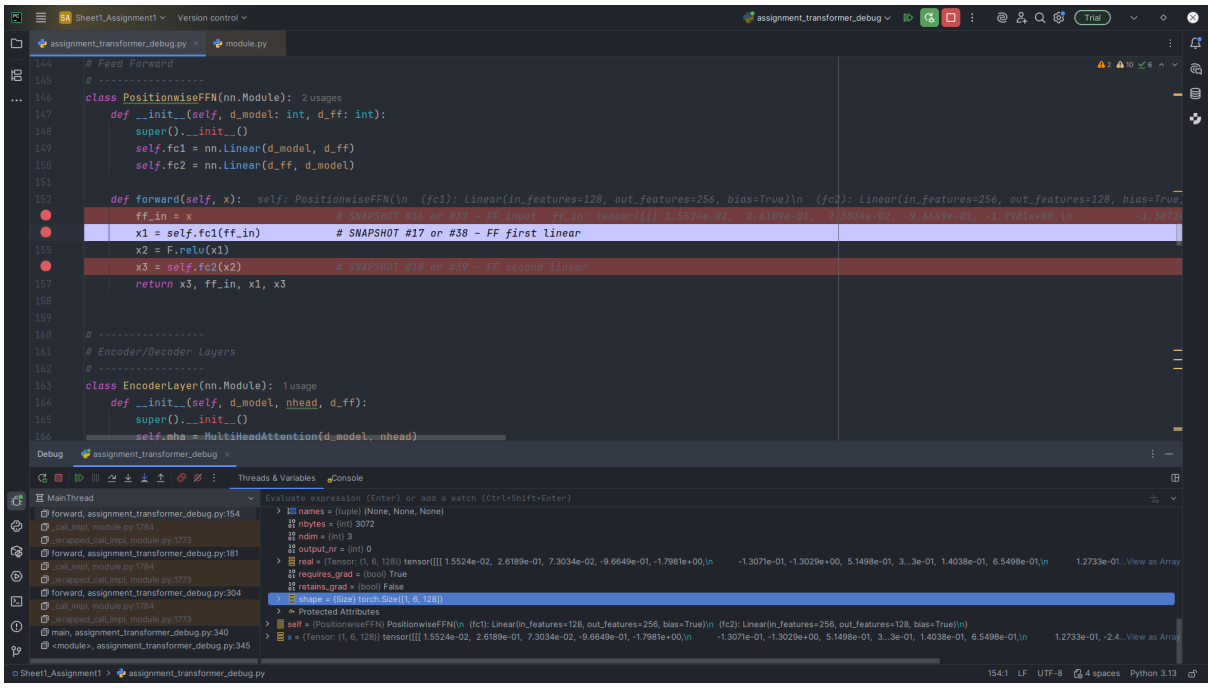


Snapshot #36 — Residual + normalization after cross-attention

Explanation & Shape

Variable (as shown in PyCharm): `refs["dec_traces"][0]["after_ca_norm"]`
Observed shape: (1,5,128)

Residual (decoder state + cross-attn output) then LayerNorm; shape stays $(1, 5, 128)$.

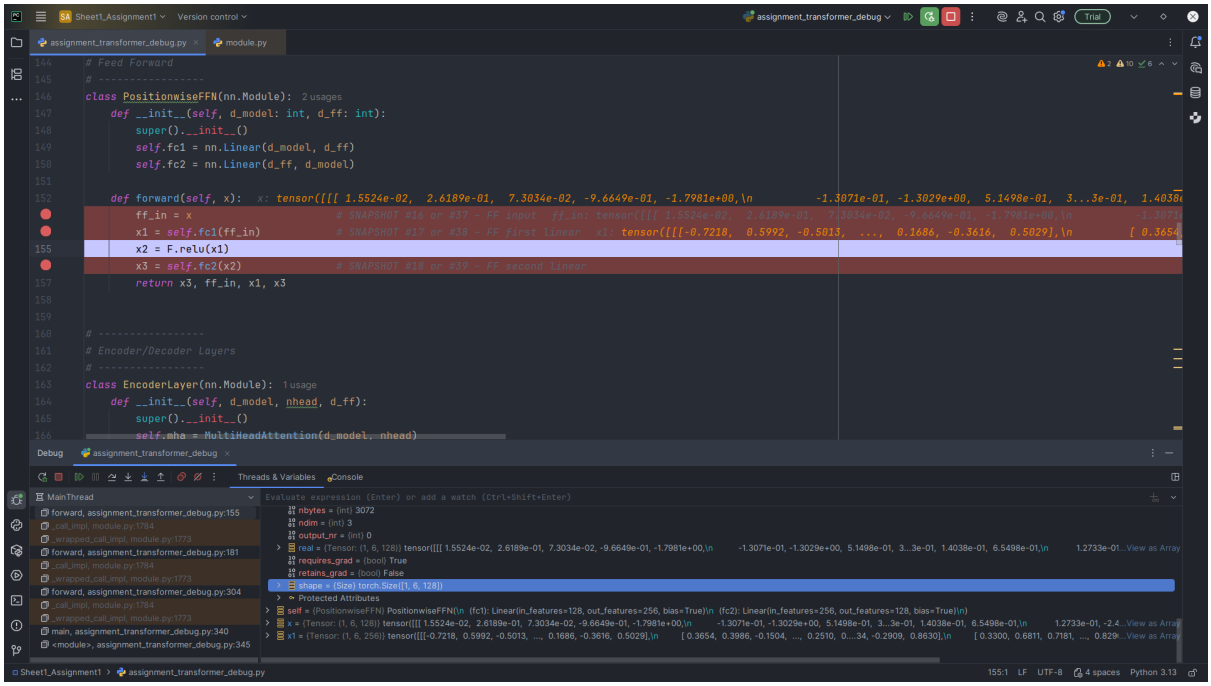


Snapshot #37 — Decoder feed-forward input

Explanation & Shape

Variable (as shown in PyCharm): `refs["dec_traces"][0]["ff_in"]`
Observed shape: (1,5,128)

Input to the decoder FFN; d_{model} stays constant across blocks to support residuals.

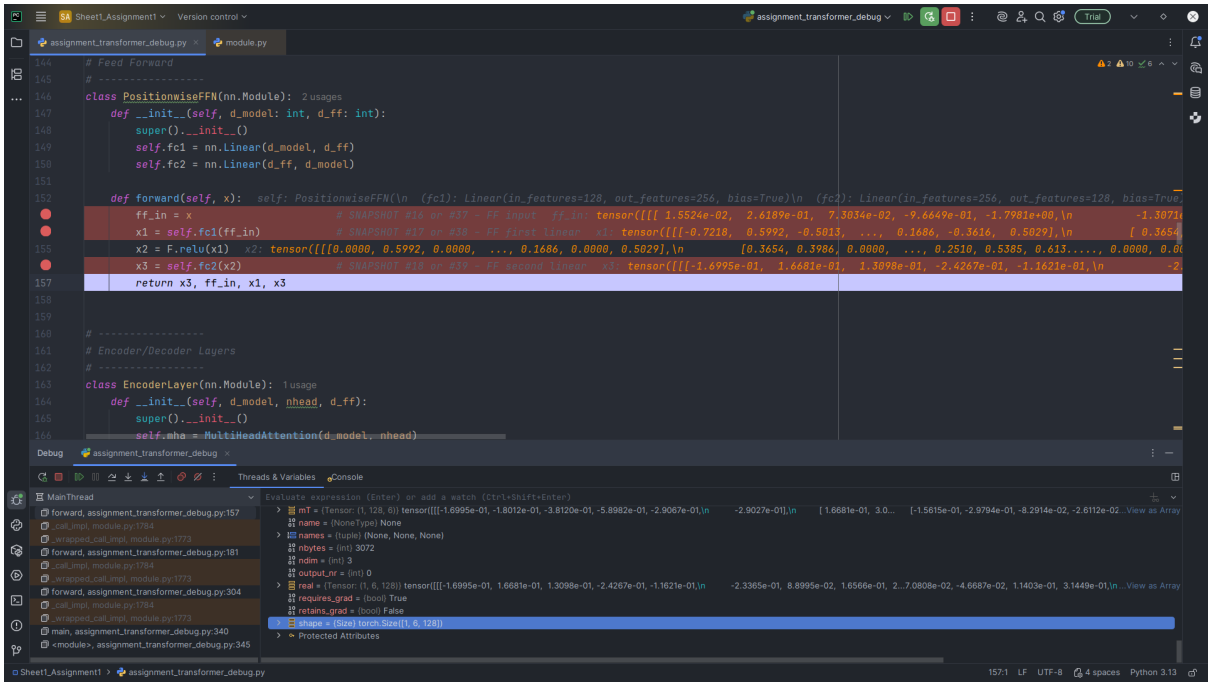


Snapshot #38 — Feed-forward first linear layer output

Explanation & Shape

Variable (as shown in PyCharm): `refs["dec_traces"][0]["ff1"]`
Observed shape: (1,5,256)

Expanded to $d_{\text{ff}} = 256$ for more capacity and non-linearity.

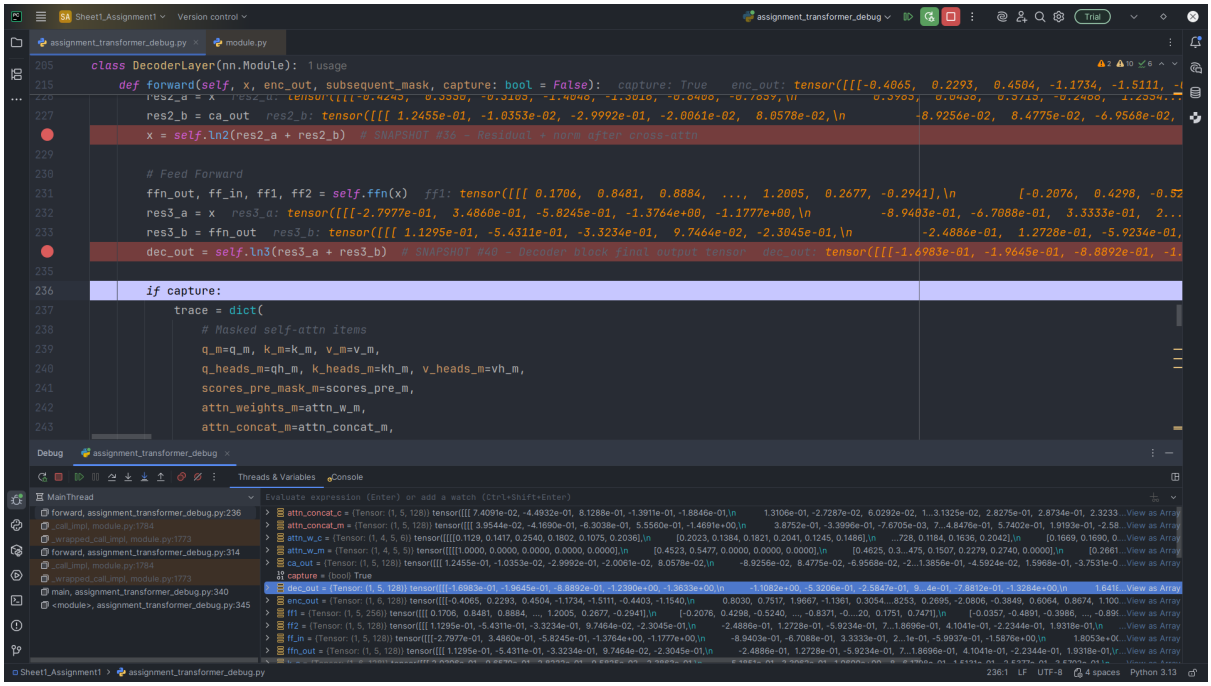


Snapshot #39 — Feed-forward second linear layer output

Explanation & Shape

Variable (as shown in PyCharm): `refs["dec_traces"][0]["ff2"]`
Observed shape: (1,5,128)

Projected back to 128 to match the residual branch.



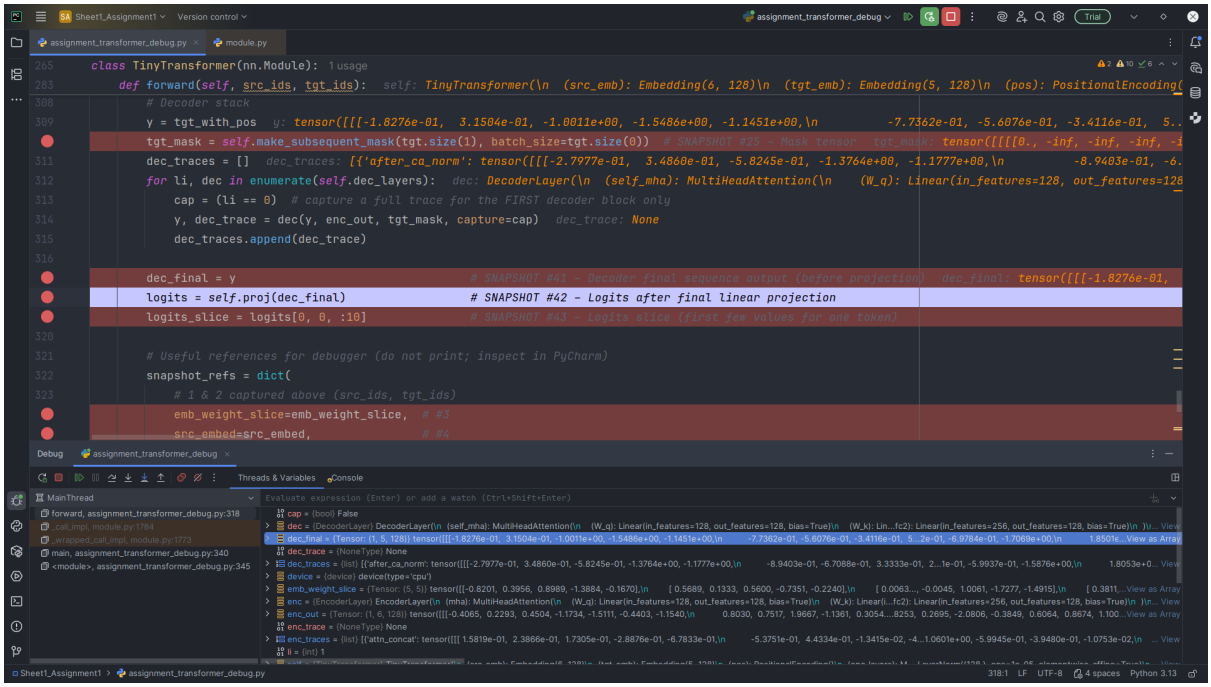
Snapshot #40 — Decoder block final output tensor

Explanation & Shape

Variable (as shown in PyCharm): `refs["dec_traces"][0]["dec_out"]`
Observed shape: (1,5,128)

Output of the decoder block after FFN residual + LayerNorm; feeds the next decoder layer or final projection.

Final Output (Snapshots 41–43)

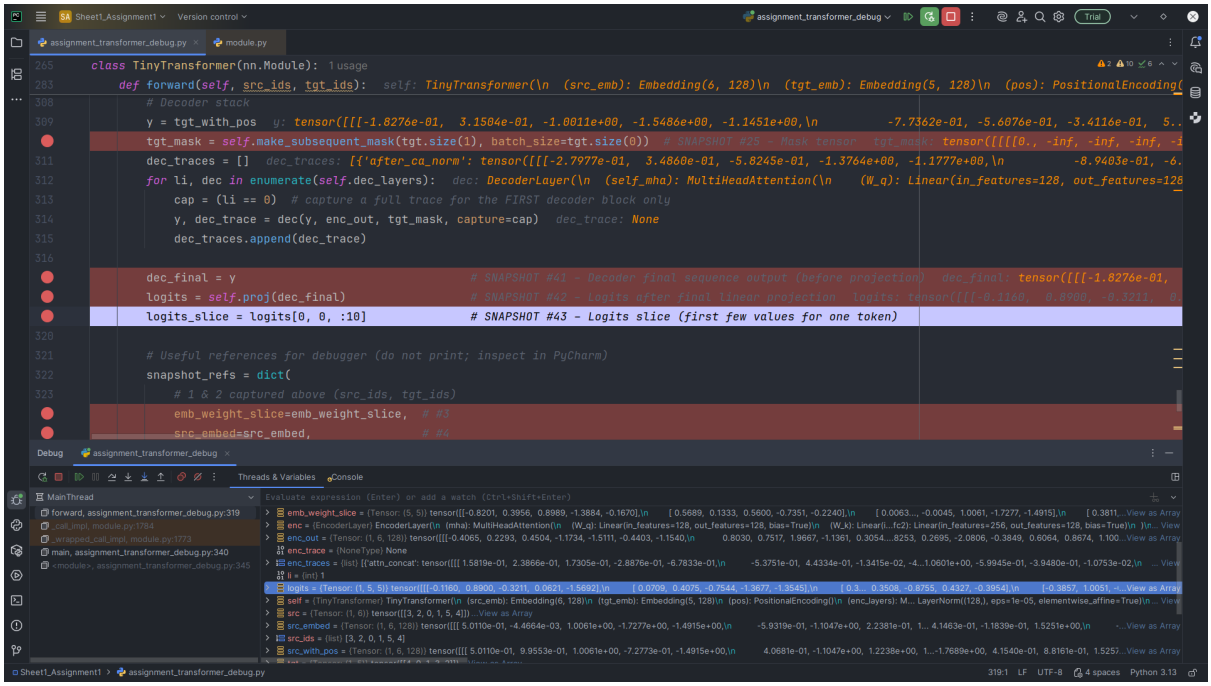


Snapshot #41 — Decoder final sequence output (before projection)

Explanation & Shape

Variable (as shown in PyCharm): `refs["dec_final"]`
Observed shape: (1,5,128)

Final decoder hidden states after all decoder layers; still in model space 128.

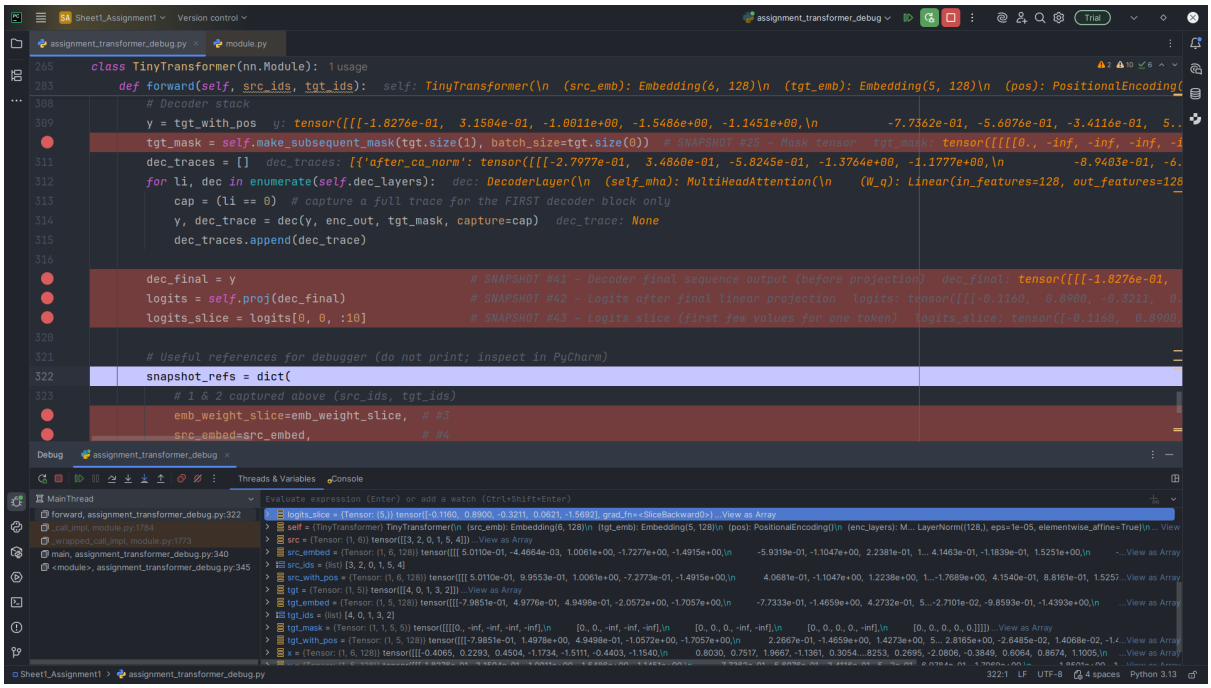


Snapshot #42 — Logits after final linear projection

Explanation & Shape

Variable (as shown in PyCharm): `refs["logits"]`
Observed shape: $(1, 5, tgt\_vocab\_size)$

A linear map converts $128 \rightarrow vocab\_size$ per time step, producing unnormalized log-probabilities (logits) over the target vocabulary.



Snapshot #43 — Logits slice (first few values for one token)

Explanation & Shape

Variable (as shown in PyCharm): `refs["logits_slice"]`
Observed shape: (≤ 10 elements)

A small slice of a token's logits for quick sanity-check: confirms projection and vocab dimension are correct.

Guiding Questions — Answers

1) What do the dimensions represent at embedding, attention, FFN, and output stages?
Embeddings use $(\text{batch}, \text{seq\_len}, d_{\text{model}})$. Attention uses $(\text{batch}, \text{heads}, \text{tgt\_len}, \text{src\_len})$ for scores/weights and $(\text{batch}, \text{heads}, \text{seq\_len}, d_{\text{head}})$ for per-head Q/K/V and outputs, with $d_{\text{head}} = d_{\text{model}}/\text{heads}$. The FFN preserves $(\cdot, \cdot, d_{\text{model}})$ externally while expanding internally to d_{ff} then projecting back. Logits are $(\text{batch}, \text{seq\_len}, \text{vocab\_size})$.

2) Why do Q, K, V have the same shape, and why split heads?
They're linear projections of the same d_{model}-dimensional token states so dot products $QK^\top$ and weighted sums on V are compatible. Splitting into heads lets the model attend to different subspaces in parallel.

3) What do the attention score matrices represent, and why square?
They are scaled dot-product affinities between queries and keys per head. In self-attention, target and source are the same sequence, so the matrix is square (seq $\times$ seq). In cross-attention, it's rectangular (tgt_len $\times$ src_len).

4) Why is masking necessary in the decoder, and how does the mask enforce it?
Masking prevents access to future positions during training. Adding $-\infty$ above the diagonal before softmax forces those probabilities to 0, ensuring autoregressive behavior.

5) How do residuals and LayerNorm ensure consistency of shapes across blocks?
Residuals require identical shapes so sublayers return d_{model}; LayerNorm normalizes features without changing dimensions, stabilizing depth.

6) Why keep the embedding dimension constant through layers?
A fixed d_{model} keeps interfaces consistent and enables valid residual additions across stacked blocks.

7) How does the final projection connect decoder output to vocabulary logits?
A learned linear map applies at each time step: $W \in \mathbb{R}^{d_{\text{model}} \times \text{vocab\_size}}$, producing logits over the target vocabulary.