Assignment 2: Vision Transformer Debugging Report

October 2, 2025

Introduction

This report documents the step-by-step debugging of a Vision Transformer (ViT) in PyCharm. Each snapshot shows the code location, the current tensor values, and shapes, with a brief description and an explanation connecting the tensor to the ViT pipeline.

Snapshots

1 Environment Setup (PyCharm + WSL)

1.1 Platform

- **Host OS:** Windows 10/11 with WSL2 (Ubuntu 22.04).
- **PyCharm:** Professional/Community, Python interpreter configured to use the WSL environment.
- **Python/PyTorch:** Python 3.10+, PyTorch, torchvision, numpy, pillow.
- **GPU (if available):** NVIDIA CUDA.

1.2 Interpreter Configuration

1. In PyCharm: `Settings > Project Interpreter > Add > WSL`.
2. Install deps: `pip install torch torchvision torchaudio pillow numpy`.
3. Confirm device: `torch.cuda.is_available()`.

1.3 Debugger Settings

- Place breakpoints at each pipeline step (# BREAKPOINT Sxx).
- Use the Debugger Variables panel to inspect shapes/values.
- Do not use print statements.

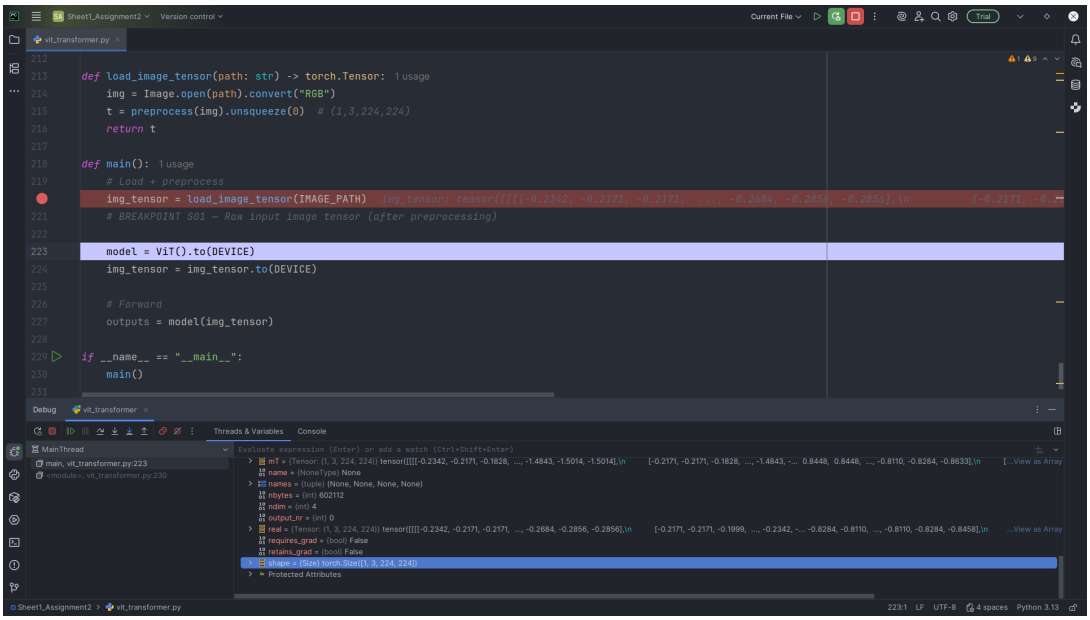
2 Unique Input Image & Preprocessing

2.1 Preprocessing Details

- Convert to RGB, resize to 224×224, normalize with ImageNet mean/std.
- Final tensor shape $(1, 3, 224, 224)$.

Figure 1: Original Input Image

S01 — Raw input tensor

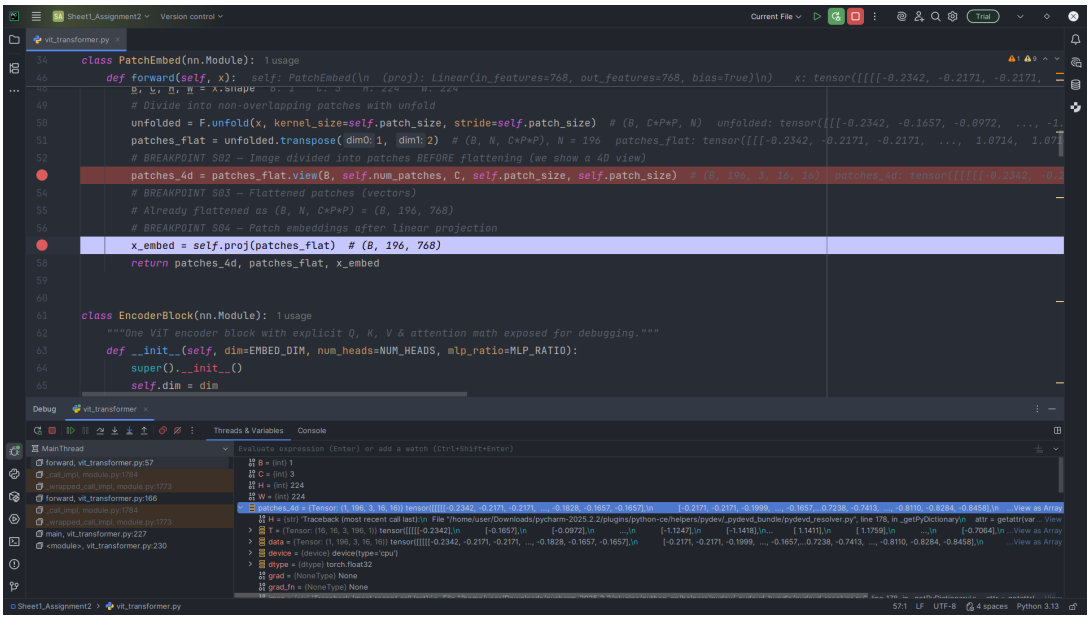


Image

Description: Preprocessed input image as a tensor of shape $1 \times 3 \times 224 \times 224$.

Explanation: Image loaded with PIL, normalized, and batched. Channel-first layout (`C`, `H`, `W`) with batch size 1 is expected by PyTorch modules.

S02 — Patches (4D view)

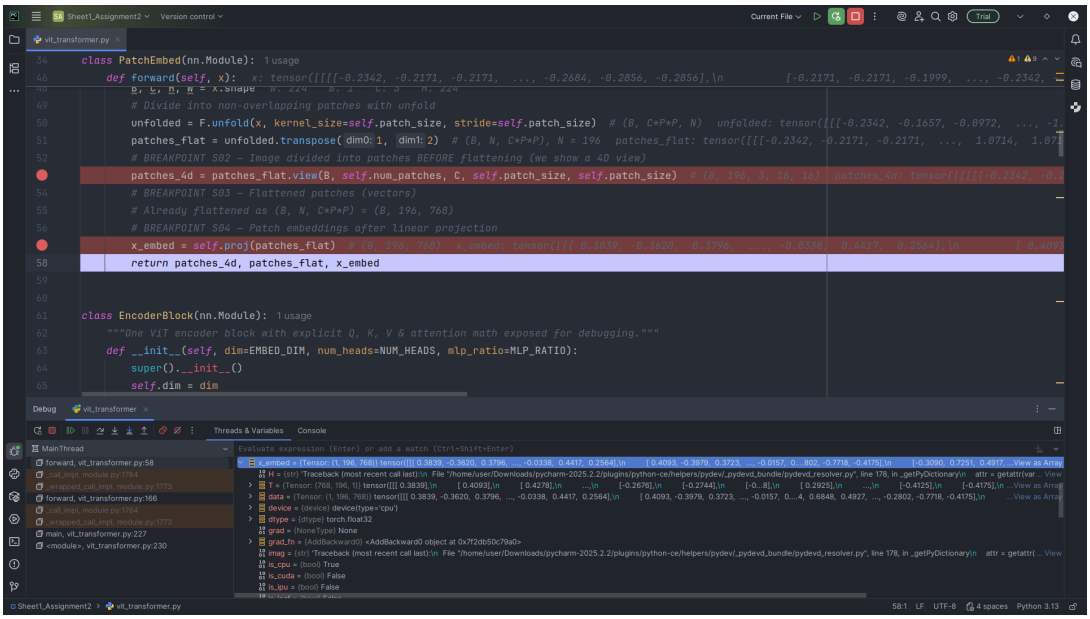


Image

Description: Image split into non-overlapping patches before flattening; shown as $(\mathtt{B}, \#\mathtt{patches}, \mathtt{C}, 16, 16)$ which corresponds to $1 \times 196 \times 3 \times 16 \times 16$.

Explanation: 'unfold' extracts 16×16 windows with stride 16. There are $14 \times 14 = 196$ patches.

S03 — Flattened patches

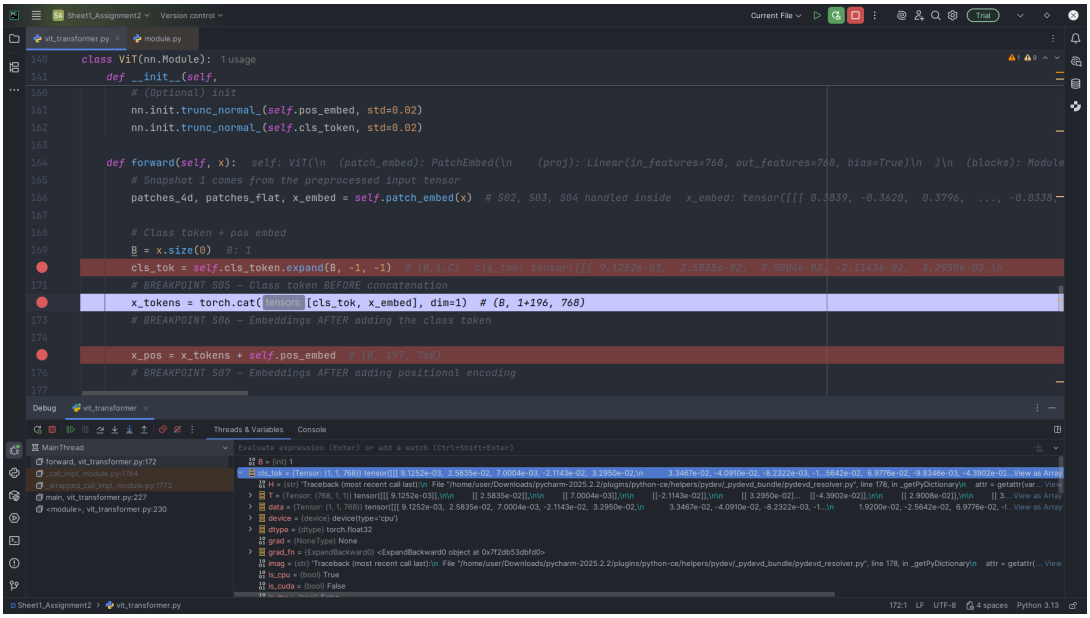


Image

Description: Flattened patch vectors of shape $1 \times 196 \times 768$.

Explanation: Each $16 \times 16 \times 3$ patch becomes a vector of length 768, ready for the linear projection.

S04 — Patch embeddings

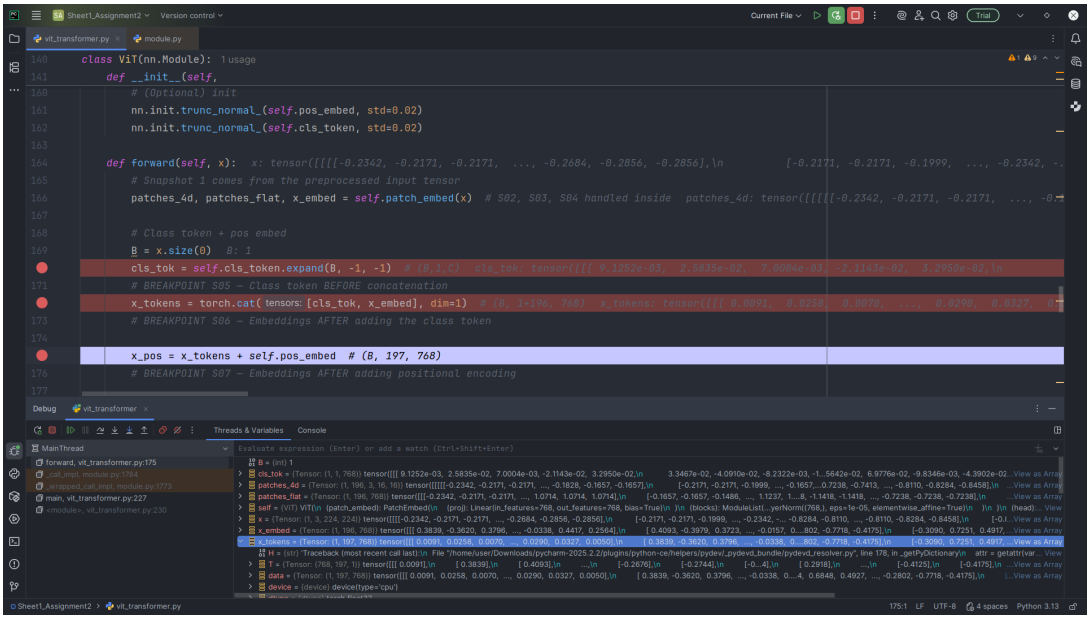


Image

Description: Projected patch embeddings $1 \times 196 \times 768$.

Explanation: A linear layer maps raw patch vectors to the model dimension $\texttt{d\_model} = 768$; this is the token embedding space used by the transformer.

S05 — Class token (before concat)

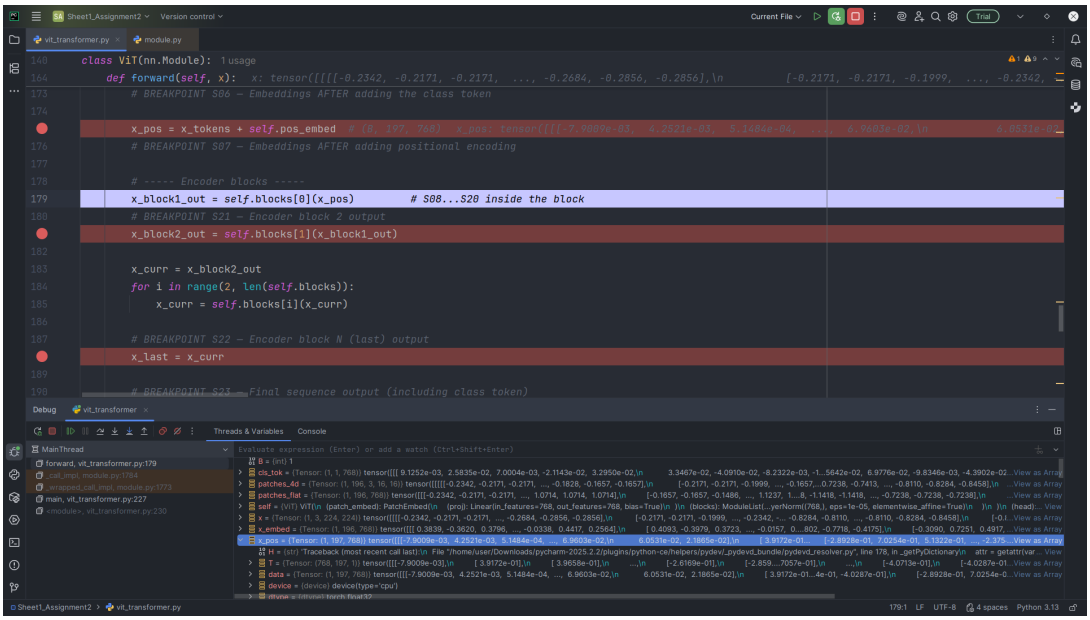


Image

Description: Learned [CLS] token expanded to batch: $1 \times 1 \times 768$.

Explanation: A trainable vector summarizes the entire sequence after attention; it will be used for classification.

S06 — Tokens after adding [CLS]

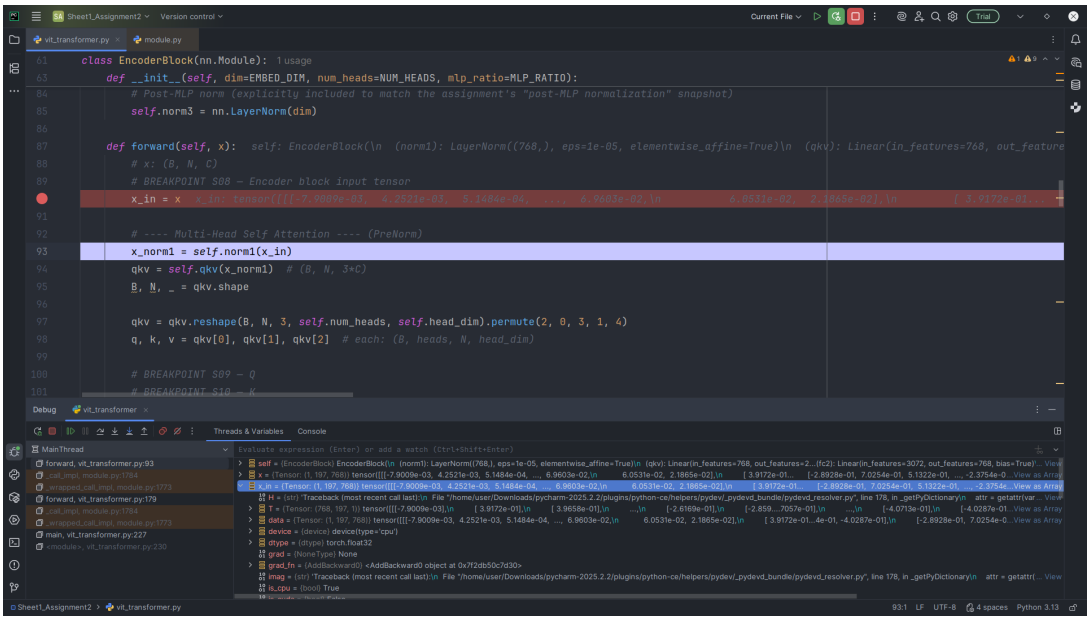


Image

Description: Sequence with class token prepended: $1 \times 197 \times 768$.

Explanation: Concatenate [CLS] (length 1) with the 196 patch tokens to form the transformer input sequence.

S07 — + Positional encoding

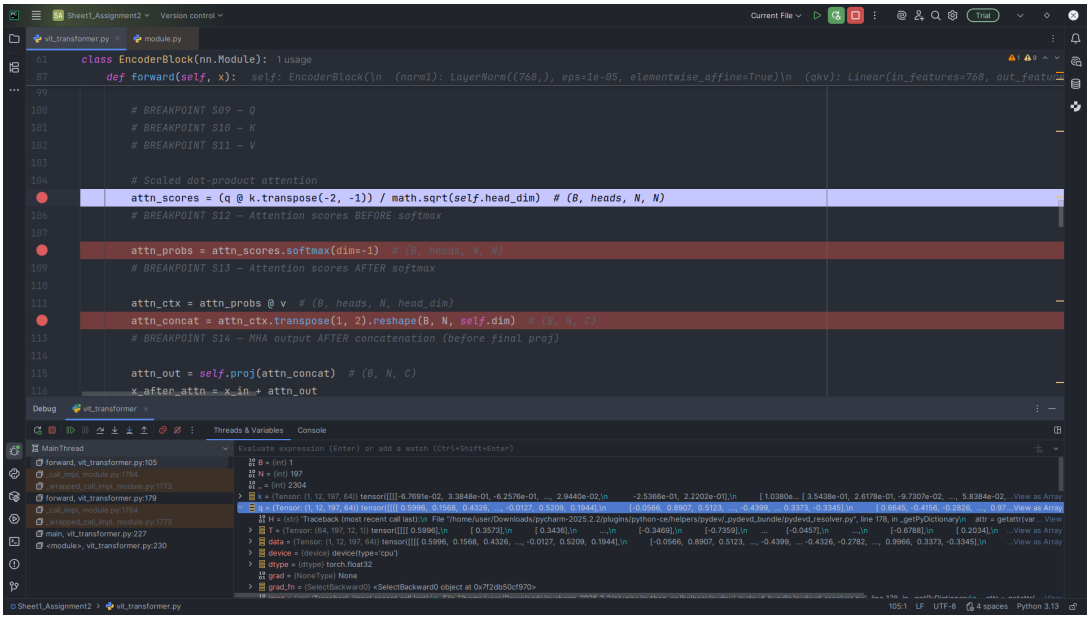


Image

Description: Token embeddings after adding positional encodings: $1 \times 197 \times 768$.

Explanation: Sinusoidal/learned positional vectors inject order information so attention can exploit spatial positions.

S08 — Block input (pre-norm)

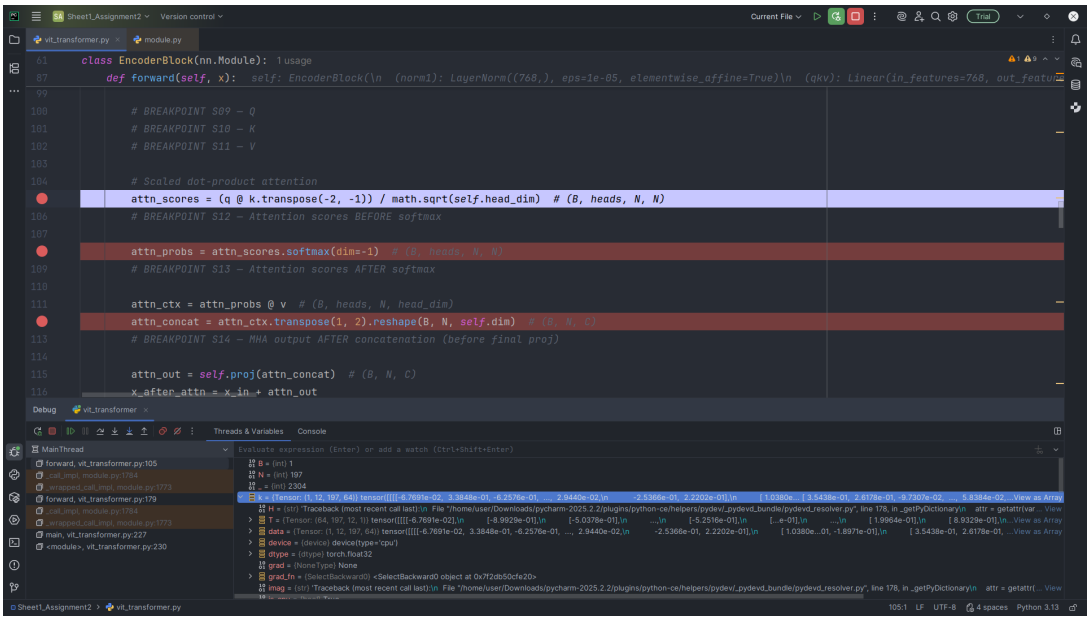


Image

Description: Input to the first encoder block: $1 \times 197 \times 768$.

Explanation: Each block applies LayerNorm (Pre-LN), Multi-Head Self-Attention (MHSA), residual, then MLP + residual.

S09 — Q (per head)

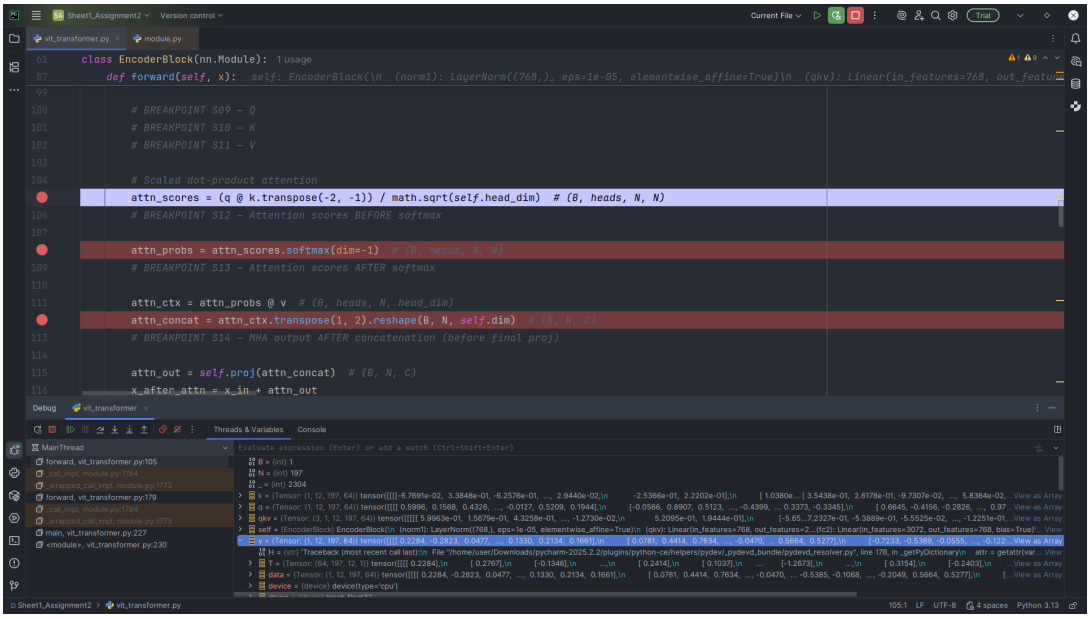


Image

Description: Query tensor reshaped to heads: $(\mathtt{B}, \mathtt{h}, \mathtt{N}, \mathtt{d\_h}) = 1 \times 12 \times 197 \times 64$.

Explanation: Linear projections split `d_model` into $h = 12$ heads with head dimension `d_h` $= 64$.

S10 — K (per head)

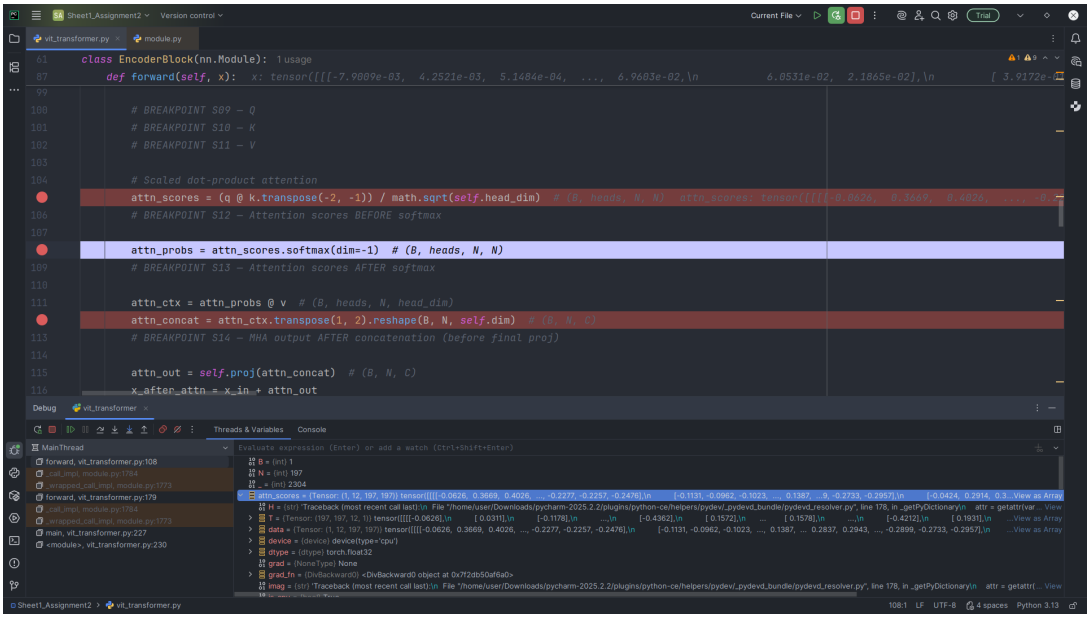


Image

Description: Key tensor $1 \times 12 \times 197 \times 64$.

Explanation: Keys are used with queries to compute attention compatibility scores.

S11 — V (per head)

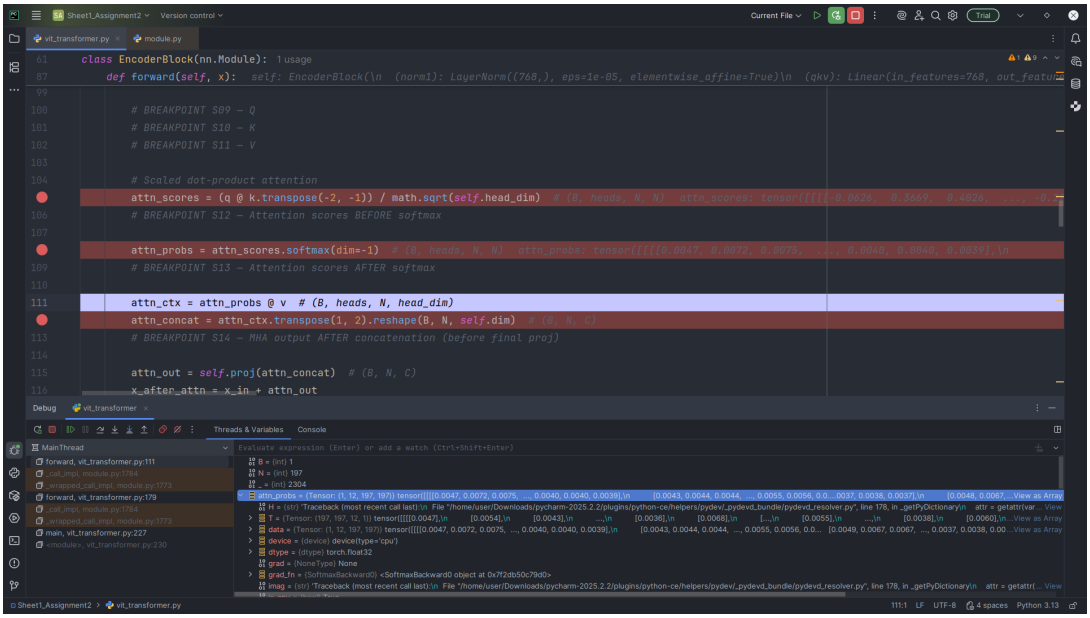


Image

Description: Value tensor $1 \times 12 \times 197 \times 64$.

Explanation: Values are the information that will be aggregated based on the attention probabilities.

S12 — Attention scores (pre-softmax)

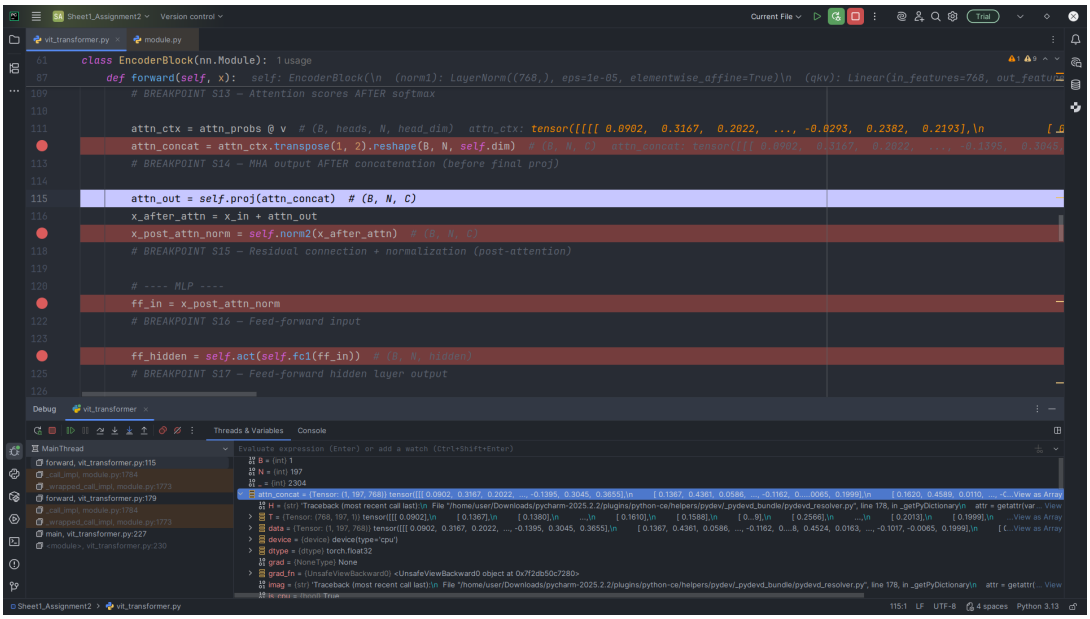


Image

Description: Scaled dot-product scores $(\mathtt{B}, \mathtt{h}, \mathtt{N}, \mathtt{N}) = 1 \times 12 \times 197 \times 197$.

Explanation: Scores are computed as $qk^\top/\sqrt{d\_h}$. These are raw compatibilities between all token pairs per head.

S13 — Attention probabilities

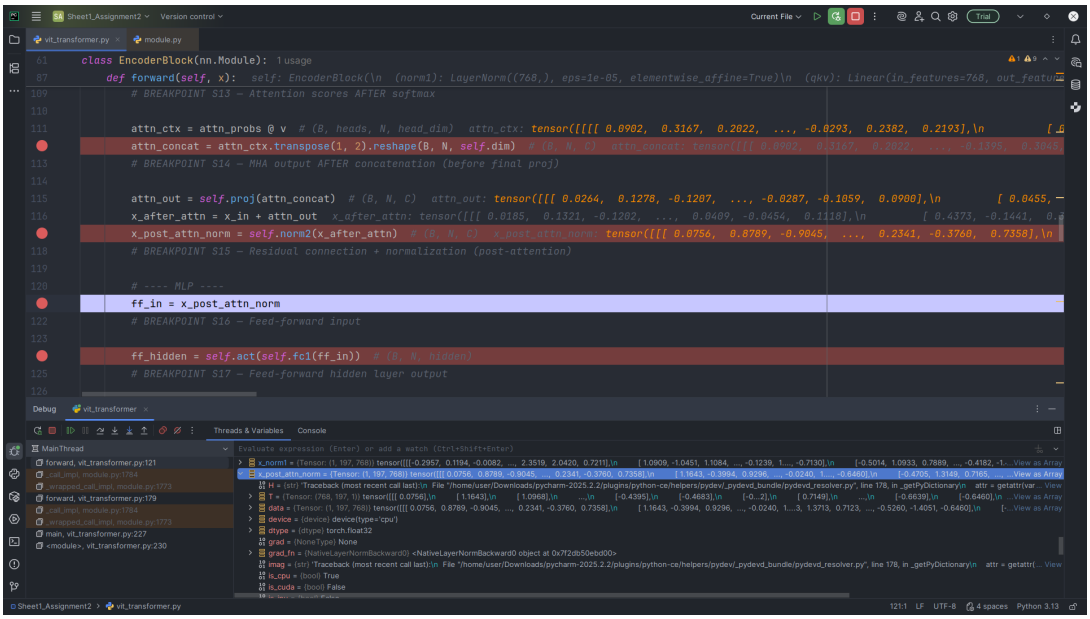


Image

Description: Softmax over the last dimension of scores: $1 \times 12 \times 197 \times 197$.

Explanation: Applying softmax along keys turns scores into probabilities; each row sums to 1 and indicates how much each token attends to others.

S14 — Context per head

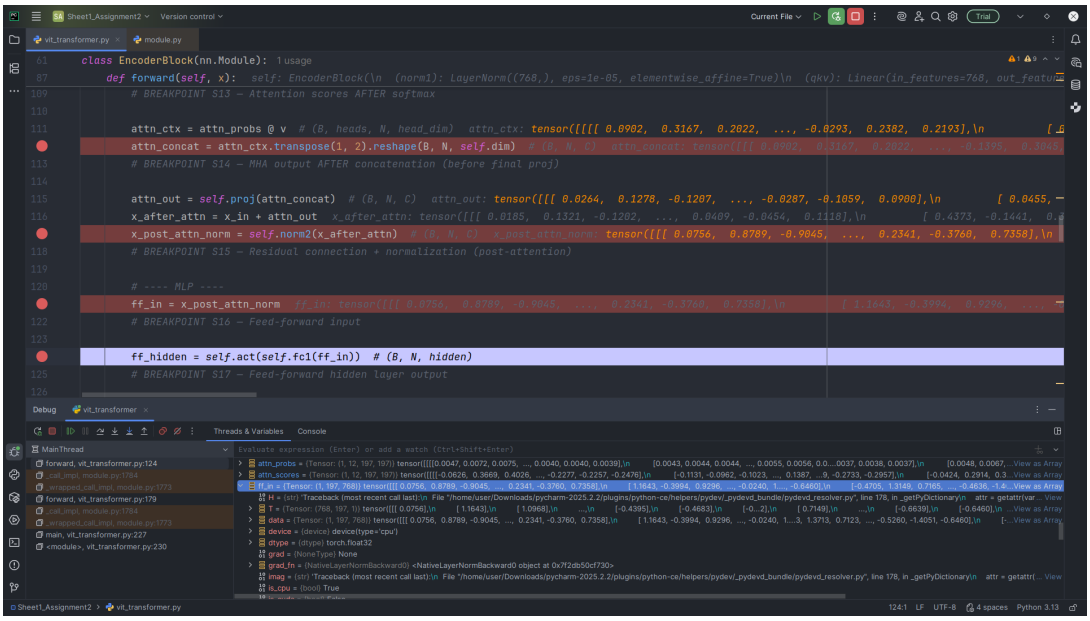


Image

Description: Head-wise context `attn_probs` · `V` with shape $1 \times 12 \times 197 \times 64$.

Explanation: Probabilities weight the values to aggregate information from relevant tokens for each head.

S15 — MHA output (concat + proj)

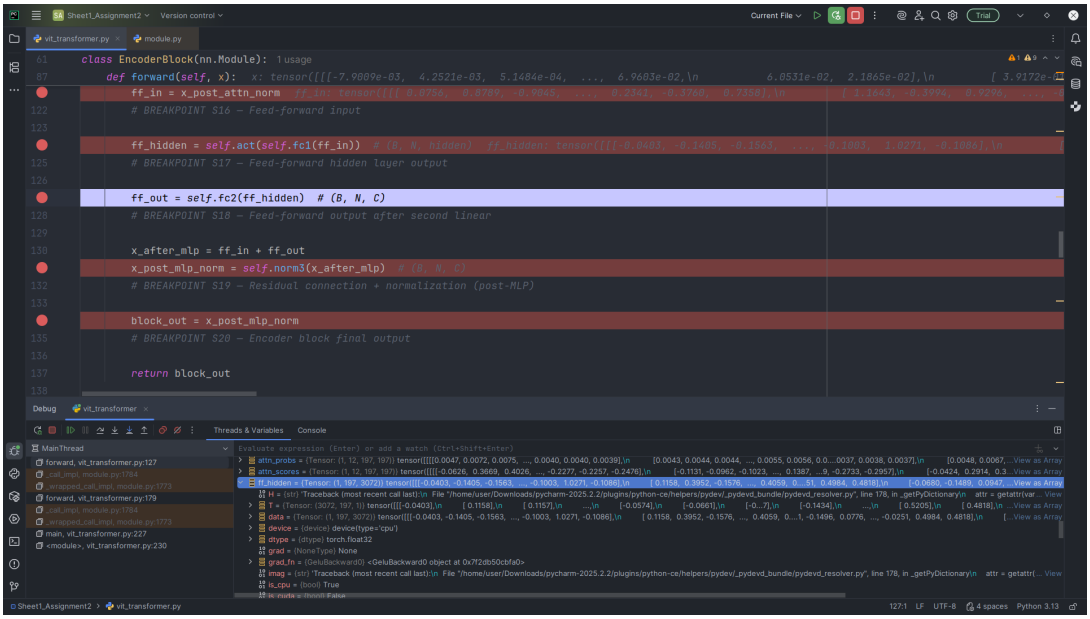


Image

Description: Concatenated head contexts then projected: `attn_out` $\in 1 \times 197 \times 768$.

Explanation: Per-head contexts are concatenated along d_h to $197 \times (12 \cdot 64)$ and passed through a linear layer to return to `d_model`.

S16 — Residual + norm (post-attention)

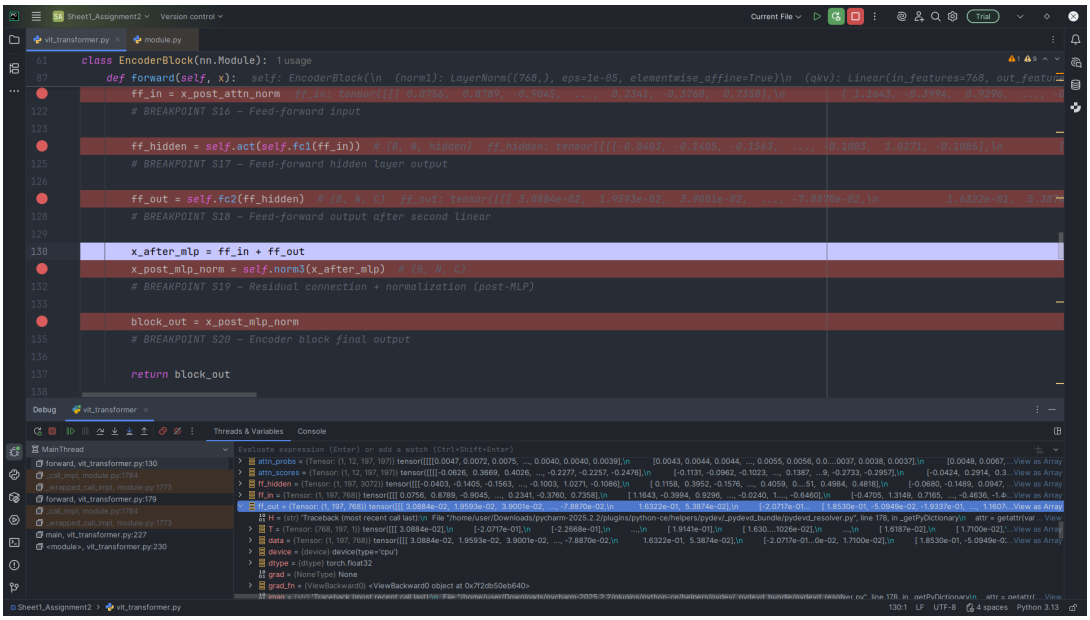


Image

Description: Feed-forward input `ff_in` with shape $1 \times 197 \times 768$.

Explanation: Add the attention output to the block input (residual) and apply LayerNorm, producing the MLP input.

S17 — FFN hidden (after fc1 + GELU)

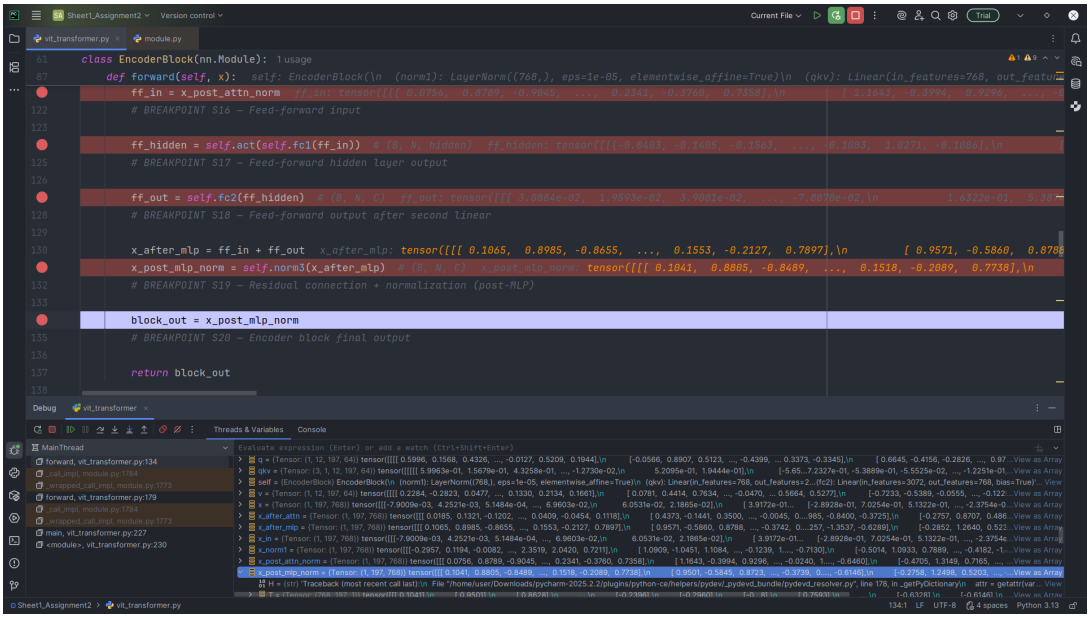


Image

Description: Nonlinear hidden activations `ff_hidden` (often $\approx 4 \times d\_model$ width).

Explanation: The first linear expands the channel dimension; GELU adds nonlinearity before the second linear brings it back.

S18 — FFN output (after fc2)

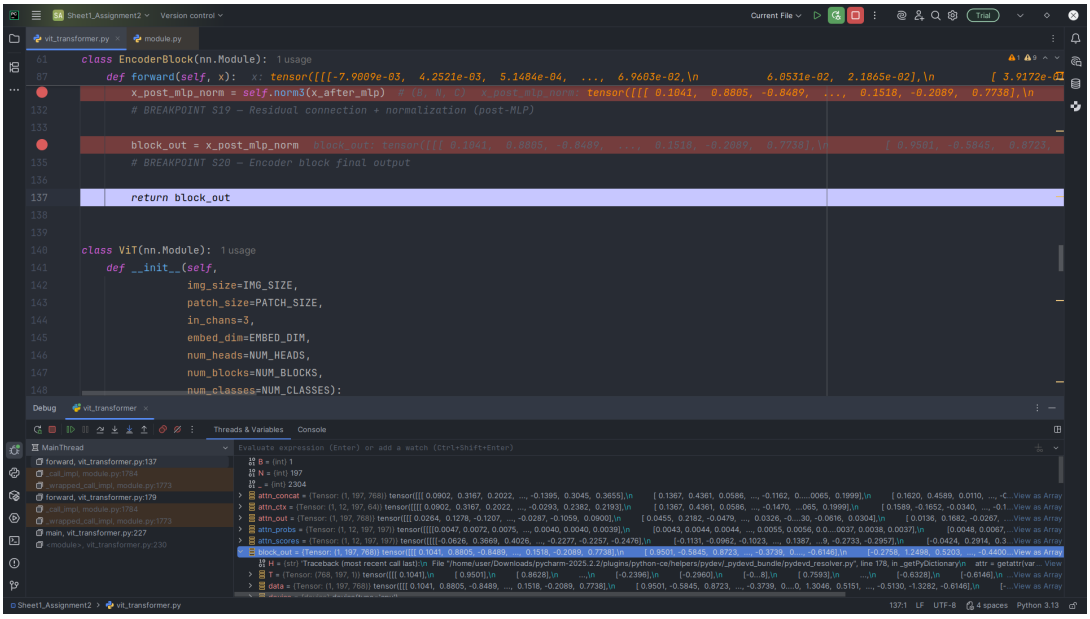


Image

Description: Feed-forward output `ff_out` $\in 1 \times 197 \times 768$.

Explanation: Second linear projects back to `d_model`.

S19 — Residual + norm (post-MLP)

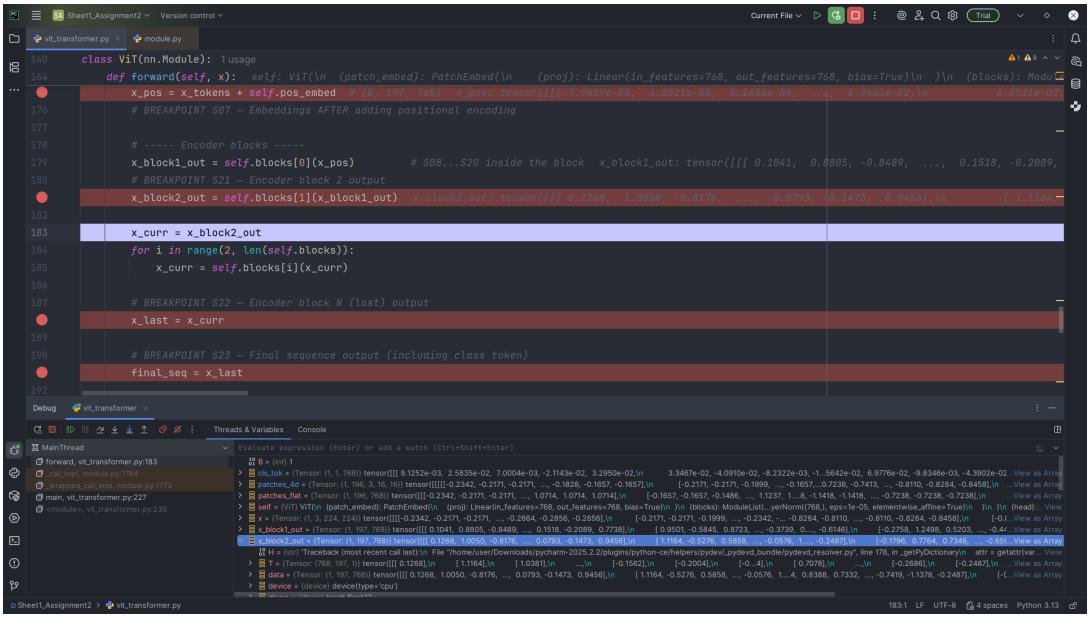


Image

Description: Post-MLP normalized tensor `x_post_mlp_norm` $\in 1 \times 197 \times 768$.

Explanation: Add `ff_out` to `ff_in` (residual) and apply LayerNorm. This is the block output.

S20 — Encoder block output

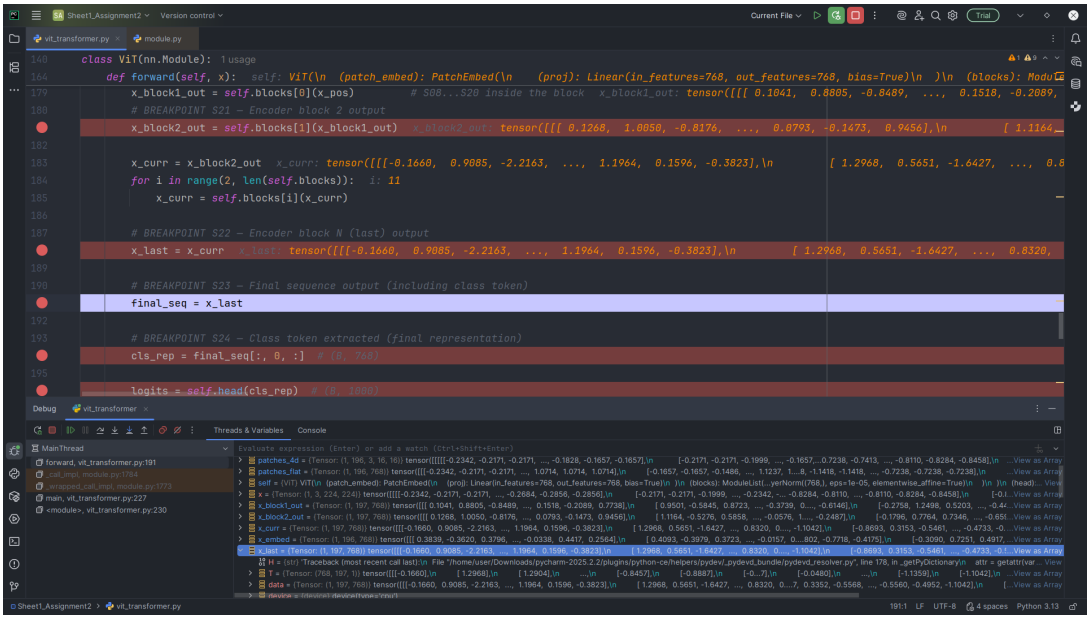


Image

Description: Final output of the shown encoder block: $1 \times 197 \times 768$.

Explanation: This tensor is fed to subsequent encoder blocks (or heads to classification if it is the last block).

S21 — Encoder block 2 output

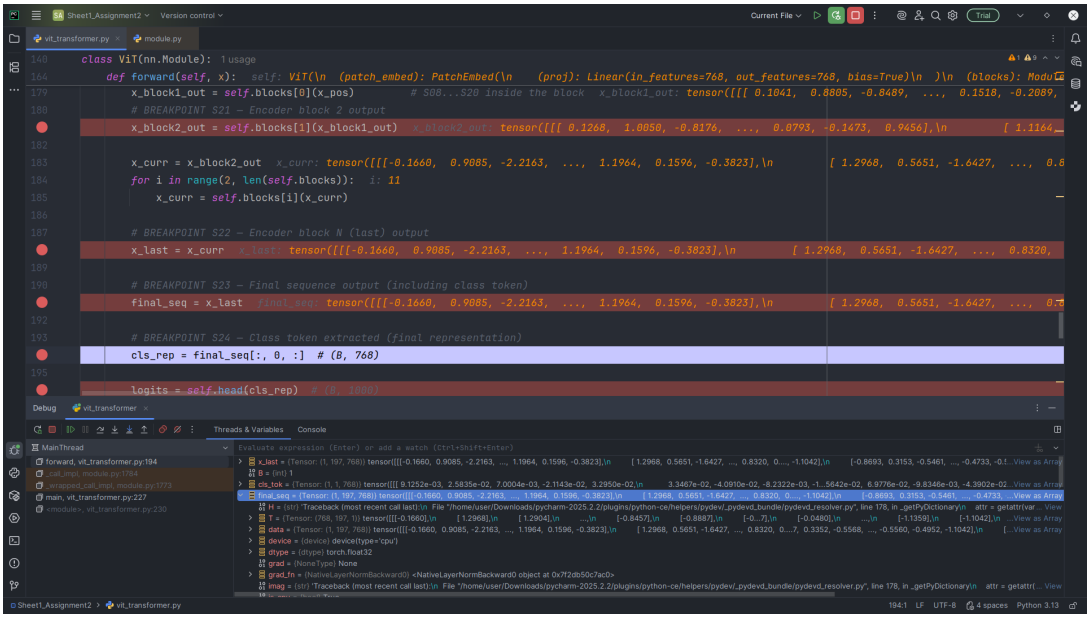


Image

Description: Output after the second encoder block: $1 \times 197 \times 768$.

Explanation: Demonstrates propagation of features through deeper layers.

S22 — Encoder block N (last) output

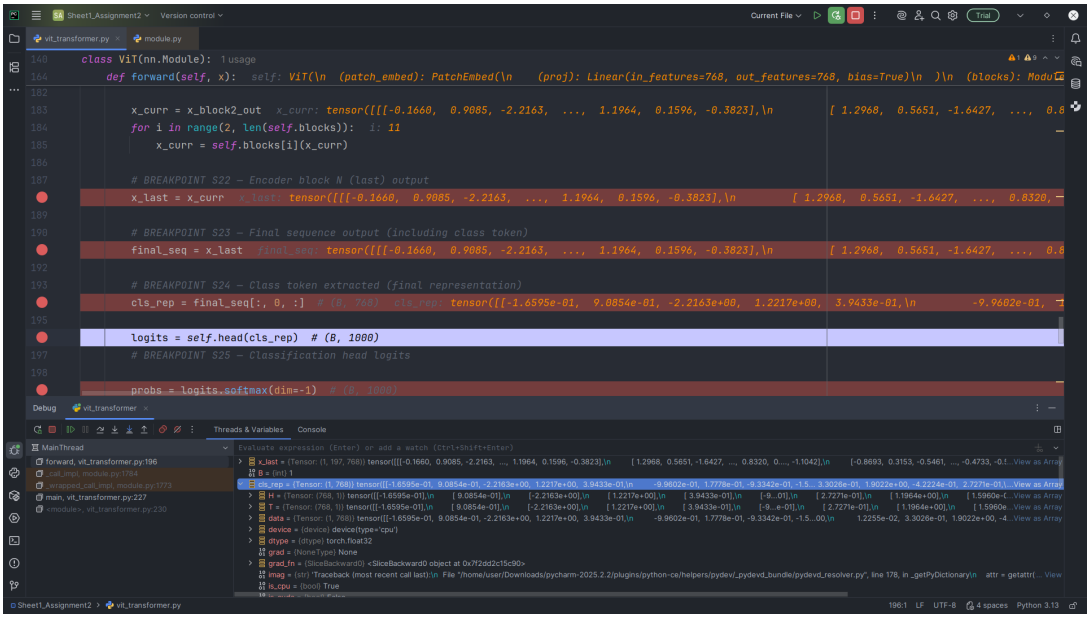


Image

Description: Last encoder output $1 \times 197 \times 768$.

Explanation: This sequence holds the final token features used to compute the class representation.

S23 — Final sequence (incl. [CLS])

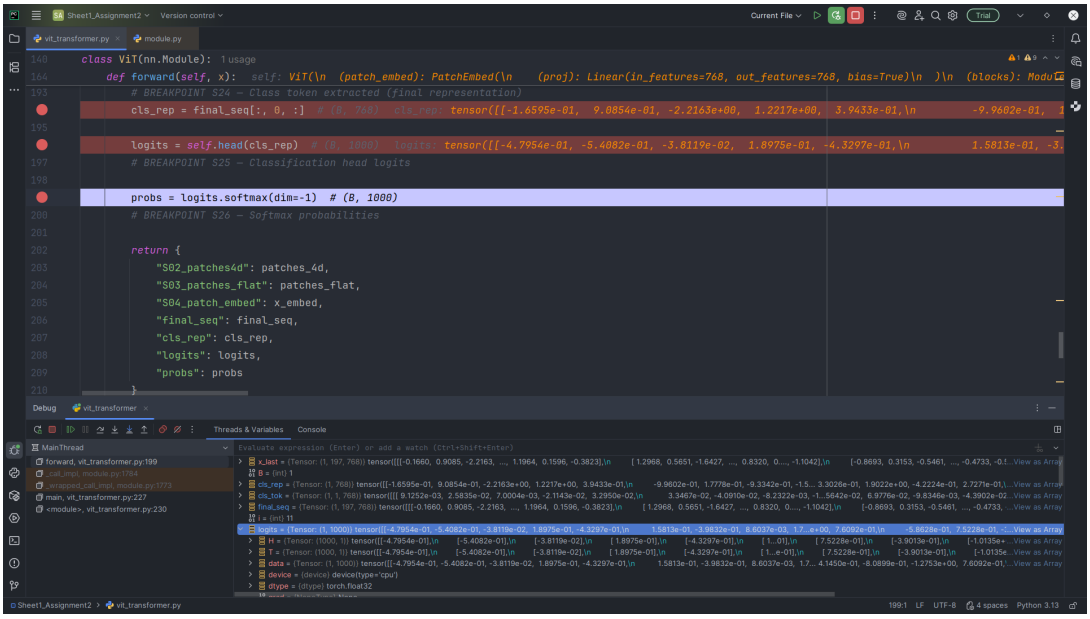


Image

Description: Final sequence $1 \times 197 \times 768$ with the first token being [CLS].

Explanation: We will slice index 0 to obtain the class representation.

S24 — Class token (final rep)

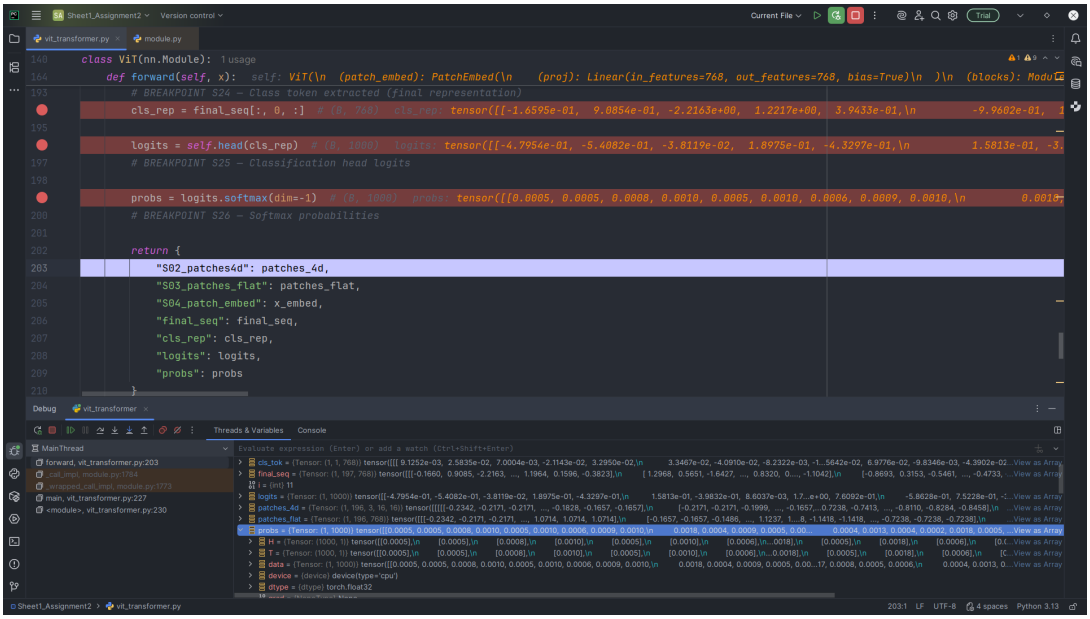


Image

Description: Class token representation $\texttt{cls\_rep} \in 1 \times 768$.

Explanation: This vector summarizes the image; it goes into the classification head.

S25 — Logits

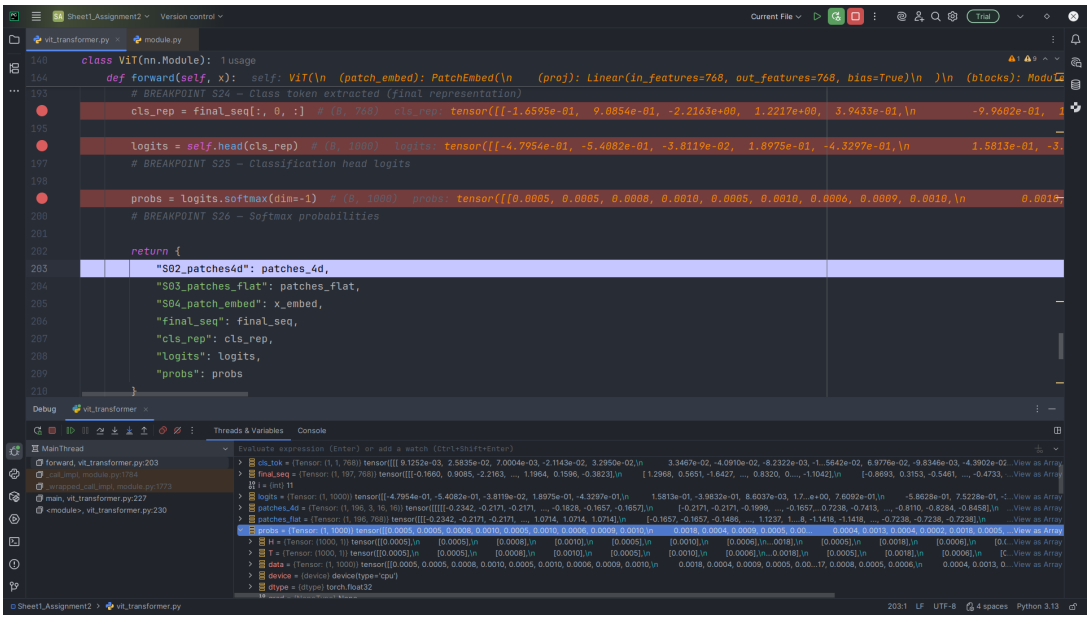


Image

Description: Classification logits 1×1000.

Explanation: Linear head maps `cls_rep` to 1000 ImageNet classes. Values are unnormalized scores.

S26 — Softmax probabilities



Image

Description: Class probabilities 1×1000.

Explanation: Softmax converts logits to probabilities; the argmax gives the predicted class while the values reflect model confidence.

3 Guiding Questions and Answers

1. **Why must the image be split into patches before embedding?**
   Vision Transformers operate on sequences of tokens, not on raw pixels. Splitting an image into 16×16 patches converts the 2D grid into a sequence of 196 patch tokens. Each patch is flattened into a vector and later projected to the model dimension. This allows the transformer to treat image patches like words in a sentence, enabling self-attention across the entire image.

2. **Why is a class token added, and how does it affect the shape?**
   The `[CLS]` token is a special learned vector prepended to the patch sequence. During training, it learns to aggregate information from all patches so it can represent the entire image. Adding it increases the sequence length from 196 to 197 tokens, changing the shape from $(1, 196, 768)$ to $(1, 197, 768)$.

3. **Why are positional encodings needed in ViT?**
   Self-attention alone is permutation-invariant, meaning it does not know the order or spatial arrangement of tokens. Positional encodings (sinusoidal or learned vectors) inject spatial order into token embeddings so the model can understand where each patch came from in the original image.

4. **Why do Q, K, V have the same dimensions, and how do attention weights scale with patch count?**

Queries (Q), Keys (K), and Values (V) are all derived from the same embedding dimension d_{model} and projected to the same head dimension $d_h = 64$ for stability in the dot-product attention calculation. This ensures the scaled dot-product $QK^\top/\sqrt{d_h}$ is well-defined. Attention weights form matrices of shape $(N \times N)$ per head, so the cost scales quadratically $\mathcal{O}(N^2)$ with the number of tokens $N = 197$.

5. **How do residual connections preserve shape consistency across encoder blocks?**
   Each encoder block uses residual (skip) connections that add the input back to the output of the sub-layer (MHA or MLP). Since both input and output share the same dimensionality $(1, 197, 768)$, the residual ensures consistent shapes across all layers. LayerNorm also preserves the same shape, so the data flows smoothly through the network depth.

6. **Why is only the class token used for the final classification?**
   The `[CLS]` token attends to all patch tokens and is optimized to summarize the entire image. Using only this single token reduces the classification head to a simple linear layer ($768 \to 1000$) instead of requiring pooling over all patches, making it efficient and accurate for image-level predictions.

4 Reflection

Debugging the Vision Transformer step by step in PyCharm gave me a much clearer understanding of how the model works internally. By stopping at each breakpoint and checking tensor shapes, I saw how the raw image becomes patches, how the `[CLS]` token is added, and how positional encodings give the model a sense of spatial order. Watching the Q, K, V tensors and attention scores in real time helped me understand how self-attention redistributes information across all tokens. I also realized the importance of residual connections for keeping shapes consistent throughout the network.

This process did not only confirm the theoretical structure of ViTs but also improved my debugging skills in PyCharm, such as setting effective breakpoints and extracting value slices directly from the debugger instead of relying on print statements. Overall, this assignment deepened my intuition for transformers and gave me practical habits I can reuse when working with other deep learning architectures.