

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

```
212
213     def load_image_tensor(path: str) -> torch.Tensor: 1usage
214         img = Image.open(path).convert("RGB")
215         t = preprocess(img).unsqueeze(0) # (1,3,224,224)
216         return t
217
218     def main(): 1usage
219         # Load + preprocess
220         img_tensor = load_image_tensor(IMAGE_PATH) 1mg_tensor: tensor([([-0.2342, -0.2171, -0.2171, ..., -0.2684, -0.2856, -0.2856], [-0.2171, -0.2171, -0.1828, ..., -1.4843, -0.8448, 0.8448, ..., -0.8110, -0.8284, -0.8653], [-0.2171, -0.2171, -0.1999, ..., -0.2342, ..., -0.8284, -0.8110, ..., -0.8284, -0.8456])])
221         # BREAKPOINT S01 - Raw input image tensor (after preprocessing)
222
223         model = ViT().to(DEVICE)
224         img_tensor = img_tensor.to(DEVICE)
225
226         # Forward
227         outputs = model(img_tensor)
228
229     if __name__ == "__main__":
230         main()
```

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)

The screenshot shows the PyCharm IDE interface with the following details:

- Top Bar:** Shows the project name "Sheet1Assignment2", "Version control", "Current File", and a "Trial" button.
- Code Editor:** Displays two classes: `PatchEmbed` and `EncoderBlock`. The `PatchEmbed` class has several methods like `forward` and `proj`. The `EncoderBlock` class has an `__init__` method. A red circle highlights the line `x_embed = self.proj(patches_flat)`.
- Debug Bar:** Shows the current file as "vit_transformer.py" and includes buttons for "Run", "Stop", and "Step Into".
- Toolbars:** Includes "File", "Edit", "View", "Tools", "Run", "Debug", "Database", "Help", and "PyCharm".
- Bottom Status Bar:** Shows "J761 LF UTR-B 4 spaces Python 3.13".

Image

Description: Image split into non-overlapping patches before flattening; shown as $(B, \# \text{patches}, C, 16, 16)$ 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

```
class ViT(nn.Module): 1 usage
  def __init__(self,
...      # (Optional) init
  nn.init.trunc_normal_(self.pos_embed, std=0.02)
  nn.init.trunc_normal_(self.cls_token, std=0.02)

  def forward(self, x):  self: ViT(n (patch_embed): PatchEmbed(\n      proj): Linear(in_features=768, out_features=768, bias=True)\n    )\n    (blocks): Module
    # Snapshot 1 comes from the preprocessed input tensor
    patches_4d, patches_flat, x_embed = self.patch_embed(x) # S02, S03, S04 handled inside x_embed: tensor([[[-0.3839, -0.3620, 0.3796, ..., -0.0338
    # Class Token + pos embed
    B = x.size(0) B: 1
    cls Tok = self.cls_token.expand(B, -1, -1) # (B,1,C) cts Tok: tensor([[[-0.1252e-05, 2.5835e-02, 7.0004e-03, -2.1143e-02, 3.2950e-02, \n
    # BREAKPOINT S05 - Class token BEFORE concatenation
    x_tokens = torch.cat([cls Tok, x_embed], dim=1) # (B, 1+196, 768)
    # BREAKPOINT S06 - Embeddings AFTER adding the class token

    x_pos = x_tokens + self.pos_embed # (B, 197, 768)
    # BREAKPOINT S07 - Embeddings AFTER adding positional encoding

```

Image

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

Explanation: A linear layer maps raw patch vectors to the model dimension $d_{\text{model}} = 768$; 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]

The screenshot shows a Jupyter Notebook interface with several tabs at the top: 'Sheet1_Assignment2' (active), 'Version control', 'Current File', and 'Trial'. The main area displays Python code for a 'vit_transformer.py' module. The code includes various imports, a class definition for 'ViT(nn.Module)', and several sections labeled with '# BREAKPOINT S06', '# BREAKPOINT S07', '# BREAKPOINT S21', '# BREAKPOINT S22', and '# BREAKPOINT S23'. A red dot indicates the current execution point is at '# BREAKPOINT S23 - Final sequence output (including class token)'. Below the code, there's a 'Debug' toolbar with icons for step operations, a 'Threads & Variables' tab, and a 'Console' tab. A 'MainThread' section in the bottom left shows a stack trace starting with 'forward_vit_transformer.py:179'. A tooltip for the 'Evaluate expression' button says 'Evaluate expression (Enter) or set a watch (Ctrl+Shift+Enter)'. The bottom status bar shows 'Sheet1_Assignment2 2 vit_transformer.py' on the left, and '179 L F UTB-8 4 spaces Python 3.13' on the right.

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

```
SheetAssignment2 > vit_transformer.py > module.py
```

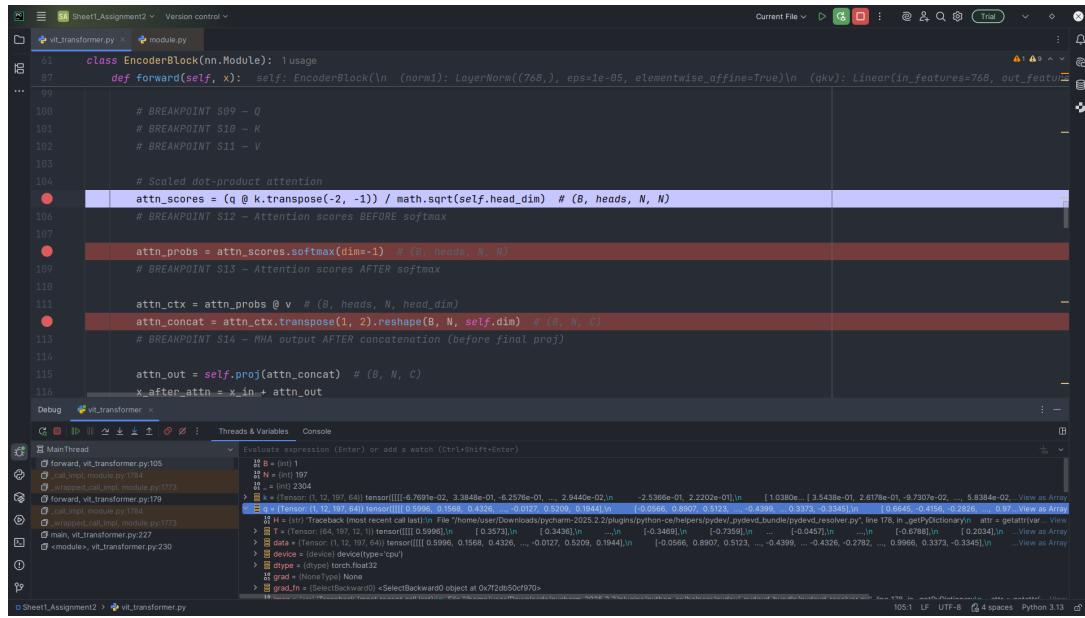
```
61     class EncoderBlock(nn.Module): 1 usage
62         def __init__(self, dim=EMBED_DIM, num_heads=NUM_HEADS, mlp_ratio=MLP_RATIO):
63             # Post-MLP norm (explicitly included to match the assignment's "post-MLP normalization" snapshot)
64             self.norms = nn.LayerNorm(dim)
65
66
67         def forward(self, x):  self: EncoderBlock(\n            norm1: LayerNorm(768),\n            eps=1e-05,\n            elementwise_affine=True)\n            (\n                qkv: Linear(in_features=768,\n                out_features=768,\n                bias=True))\n            (\n                x_in: x\n                x_in: tensor([[[-7.9909e-03,\n                    4.2521e-03,\n                    5.1484e-04,\n                    ...,\n                    6.9603e-02,\n                    6.8531e-02,\n                    2.1865e-02],\n                    ...,\n                    3.9172e-01,...\n                    ]])\n            )\n            # ----- Multi-Head Self Attention ----- (PreNorm)\n            x_norm1 = self.norm1(x_in)\n            qkv = self.qkv(x_norm1) # (B, N, 3*C)\n            B, N, _ = qkv.shape\n            qkv = qkv.reshape(B, N, 3, self.num_heads, self.head_dim).permute(2, 0, 3, 1, 4)\n            q, k, v = qkv[0], qkv[1], qkv[2] # each: (B, heads, N, head_dim)\n\n            # /BREAKPOINT S08 - Encoder block input tensor\n            # /BREAKPOINT S09 -\n            # /BREAKPOINT S10 - Q\n            # /BREAKPOINT S11 - K\n\n        Debug vit_transformer <=
```

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)



The screenshot shows a debugger interface with the following details:

- Code View:** The file `vit_transformer.py` is open, showing the `EncoderBlock` class definition and its `forward` method implementation. Breakpoints are set at several points in the code, indicated by red dots.
- Breakpoint Details:**
 - `# BREAKPOINT S09 - Q`
 - `# BREAKPOINT S10 - K`
 - `# BREAKPOINT S11 - V`
 - `# Scaled dot-product attention`
 - `attn_scores = (q @ k.transpose(-2, -1)) / math.sqrt(self.head_dim) # (B, heads, N, N)`
 - `# BREAKPOINT S12 - Attention scores BEFORE softmax`
 - `attn_probs = attn_scores.softmax(dim=-1) # (B, heads, N, N)`
 - `# BREAKPOINT S13 - Attention scores AFTER softmax`
 - `attn_ctx = attn_probs @ v # (B, heads, N, head_dim)`
 - `attn_concat = attn_ctx.transpose(1, 2).reshape(B, N, self.dim) # (B, N, C)`
 - `# BREAKPOINT S14 - MHA output AFTER concatenation (before final proj)`
 - `attn_out = self.proj(attn_concat) # (B, N, C)`
 - `x_after_attn = x_in + attn_out`
- Variables View:** The "Threads & Variables" tab is active, showing the current state of variables. A tooltip for the variable `x` indicates it is a tensor of shape `(1, 12, 197, 64)`.
- Call Stack:** The "Call Stack" tab shows the call history, starting from `forward` in `vit_transformer.py` and moving up through various `__wrapped__call_` methods and `_call_impl` methods.
- Registers:** The "Registers" tab is visible at the bottom left.

Image

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

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

S09 — Q (per head)

The screenshot shows the PyCharm IDE interface with the code editor open to `vit_transformer.py`. The code defines an `EncoderBlock` class with a `forward` method. A breakpoint is set on the line where `attn_scores` is calculated. The debugger's stack trace window is visible, showing the call stack for the most recent call to `attn_scores`. The stack trace includes frames from `forward`, `call_impl`, `wrapped_call_impl`, and `main` methods, along with internal PyTorch and PyDevolver frames. The `attn_scores` tensor is highlighted in the stack trace, showing its shape as `(B, 12, 197, 64)` and its numerical values.

Image

Description: Query tensor reshaped to heads: $(B, h, N, d_h) = 1 \times 12 \times 197 \times 64$.

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

S10 — K (per head)

This screenshot is identical to the one above, showing the PyCharm IDE with the `vit_transformer.py` code editor. A breakpoint is set on the line calculating `attn_scores`. The debugger's stack trace window is open, showing the same call stack and highlighting the `attn_scores` tensor.

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)

The screenshot shows a PyCharm interface with the following details:

- File Structure:** The left sidebar shows the project structure with files like `vit_transformer.py`, `modis.py`, and `vit_transformer.py`.
- Code Editor:** The main area displays the `vit_transformer.py` file. The code implements an EncoderBlock class using PyTorch's nn.Module. It includes various layers like LayerNorm, Linear, and Softmax, along with attention mechanisms. Several lines of code are annotated with red dots and arrows, indicating specific points of interest or analysis.
- Debugging:** A "Debug" tab is open at the bottom left, showing the current stack trace and variable values. The stack trace starts with `forward_vit_transformer` and includes frames for `call_impl`, `wrapped_call_impl`, and `__call__`. Variable values are shown for `B` (int 1), `N` (int 197), and `C` (int 2304).
- Tool Window:** The bottom right contains a "Threads & Variables" tool window, which is currently empty.

Image

Description: Scaled dot-product scores (B, h, N, 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

The screenshot shows the PyCharm debugger interface during the execution of `vit_transformer.py`. The code being debugged is the `EncoderBlock` class. A breakpoint is set at line 110, where the attention context tensor is calculated. The variable `attn_ctx` is highlighted in red, showing its shape as $(B, heads, N, head_dim)$. The tensor contains numerical values ranging from -0.1995 to 0.3045. The code then performs a transpose operation and reshapes it to (B, N, C) . This is followed by a residual connection and normalization. The next step involves an MLP, represented by the `ff_in` variable. The final output is the `ff_hidden` variable, which is the result of applying the `self.act` activation function to the feed-forward layer's input. The code is annotated with several `# BREAKPOINT` comments indicating specific points of interest in the process.

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

```

61     class EncoderBlock(nn.Module): 1 usage
...
87         def forward(self, x):  self: EncoderBlock( (norm1): LayerNorm((768,), eps=1e-05, elementwise_affine=True) (qkv): Linear(in_features=768, out_features=768, bias=False)
...
109         # BREAKPOINT S13 - Attention scores AFTER softmax
...
110
111         attn_ctx = attn_probs @ v  # (B, heads, N, head_dim)  attn_ctx: tensor([[[-0.0902, 0.3167, 0.2022, ..., -0.0293, 0.2382, 0.2193], ...
112         attn_concat = attn_ctx.transpose(1, 2).reshape(B, N, self.dim)  # (B, N, C)  attn_concat: tensor([[[-0.0902, 0.3167, 0.2022, ..., -0.1395, 0.3045,
...
113         # BREAKPOINT S14 - MHA output AFTER concatenation (before final proj)
...
114
115         attn_out = self.proj(attn_concat)  # (B, N, C)  attn_out: tensor([[[-0.0264, 0.1278, -0.1207, ..., -0.0287, -0.1059, 0.0908], ...
116         x_after_attn = x_in + attn_out  x_after_attn: tensor([[0.0185, 0.1321, -0.1202, ..., 0.0409, -0.0454, 0.1118], ...
117         x_post_attn_norm = self.norm(x_after_attn)  # (B, N, C)  x_post_attn_norm: tensor([[[-0.0756, 0.8789, -0.9845, ..., 0.2341, -0.3760, 0.7358], ...
...
118         # BREAKPOINT S15 - Residual connection + normalization (post-attention)
...
119
120         # ---- MLP ----
121         ff_in = x_post_attn_norm
...
122         # BREAKPOINT S16 - Feed-forward input
...
123
124         ff_hidden = self.act(self.fc1(ff_in))  # (B, N, hidden)
...
125         # BREAKPOINT S17 - Feed-forward hidden layer output
...
126

```

Image

Description: Head-wise context $\text{attn_probs} \cdot V$ 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)

```

61     class EncoderBlock(nn.Module): 1 usage
...
87         def forward(self, x):  self: EncoderBlock( (norm1): LayerNorm((768,), eps=1e-05, elementwise_affine=True) (qkv): Linear(in_features=768, out_features=768, bias=False)
...
109         # BREAKPOINT S13 - Attention scores AFTER softmax
...
110
111         attn_ctx = attn_probs @ v  # (B, heads, N, head_dim)  attn_ctx: tensor([[[-0.0902, 0.3167, 0.2022, ..., -0.0293, 0.2382, 0.2193], ...
112         attn_concat = attn_ctx.transpose(1, 2).reshape(B, N, self.dim)  # (B, N, C)  attn_concat: tensor([[[-0.0902, 0.3167, 0.2022, ..., -0.1395, 0.3045,
...
113         # BREAKPOINT S14 - MHA output AFTER concatenation (before final proj)
...
114
115         attn_out = self.proj(attn_concat)  # (B, N, C)  attn_out: tensor([[[-0.0264, 0.1278, -0.1207, ..., -0.0287, -0.1059, 0.0908], ...
116         x_after_attn = x_in + attn_out  x_after_attn: tensor([[0.0185, 0.1321, -0.1202, ..., 0.0409, -0.0454, 0.1118], ...
117         x_post_attn_norm = self.norm(x_after_attn)  # (B, N, C)  x_post_attn_norm: tensor([[[-0.0756, 0.8789, -0.9845, ..., 0.2341, -0.3760, 0.7358], ...
...
118         # BREAKPOINT S15 - Residual connection + normalization (post-attention)
...
119
120         # ---- MLP ----
121         ff_in = x_post_attn_norm
...
122         # BREAKPOINT S16 - Feed-forward input
...
123
124         ff_hidden = self.act(self.fc1(ff_in))  # (B, N, hidden)
...
125         # BREAKPOINT S17 - Feed-forward hidden layer output
...
126

```

Image

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

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)

The screenshot shows the PyCharm IDE interface during debugging. The code editor displays the `vit_transformer.py` file, specifically the `EncoderBlock` class. The cursor is at line 137, which contains the line `return block_out`. Several lines of code above it are highlighted in red, indicating they are part of the current breakpoint or step. The stack trace in the bottom left corner shows the call stack from the main thread up to the current line, with each frame having a small icon and some text describing the frame.

```

class EncoderBlock(nn.Module):
    def forward(self, x):
        ff_in = x.post_attn_norm
        ff_hidden = self.act(self.fc1(ff_in)) # (B, N, hidden)
        ff_out = self.fc2(ff_hidden) # (B, N, C)
        x_after_mlp = ff_in + ff_out
        x_post_mlp_norm = self.norm3(x_after_mlp) # (B, N, C)
        block_out = x_post_mlp_norm
        return block_out

```

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 Layer-Norm, producing the MLP input.

S17 — FFN hidden (after fc1 + GELU)

```
61     class EncoderBlock(nn.Module): 1 usage
62         def forward(self, x): self: EncoderBlock(n_ (norm1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)\n63             ff_in = x.post_attn_norm  ff_in: tensor([(0.0756, 0.8789, -0.9045, ..., 0.2341, 0.3760, 0.7358),\n64                 [-1.1643, -0.3995, 0.9296, ...\n65             # BREAKPOINT S16 - Feed-forward input\n66\n67             ff_hidden = self.act(self.fel(ff_in)) # (B, N, hidden)  ff_hidden: tensor([([-0.0403, -0.1405, -0.1583, ..., -0.1003, 1.0271, -0.1086],\n68                 [0.1003, 0.1405, 0.1583, ..., 0.0403, -0.1271, -0.1086],\n69             # BREAKPOINT S17 - Feed-forward hidden layer output\n70\n71             ff_out = self.fc2(ff_hidden) # (B, N, C)  ff_out: tensor([(3.0884e-02, 1.9893e-02, 3.9001e-02, ..., -7.8870e-02, 1.6322e-01, 5.3875e-02,\n72                 [0.1003, 0.1405, 0.1583, ..., 0.0403, -0.1271, -0.1086],\n73             # BREAKPOINT S18 - Feed-forward output after second linear\n74\n75             x_after_mlp = ff_in + ff_out\n76             x_post_mlp_norm = self.norm3(x_after_mlp) # (B, N, C)\n77             # BREAKPOINT S19 - Residual connection + normalization (post-MLP)\n78\n79             block_out = x_post_mlp_norm\n80             # BREAKPOINT S20 - Encoder block final output\n81\n82             return block_out\n83\n84\n85\n86\n87\n88\n89\n90\n91\n92\n93\n94\n95\n96\n97\n98\n99\n100\n101\n102\n103\n104\n105\n106\n107\n108\n109\n110\n111\n112\n113\n114\n115\n116\n117\n118\n119\n120\n121\n122\n123\n124\n125\n126\n127\n128\n129\n130\n131\n132\n133\n134\n135\n136\n137\n138\n139\n140\n141\n142\n143\n144\n145\n146\n147\n148\n149\n150\n151\n152\n153\n154\n155\n156\n157\n158\n159\n160\n161\n162\n163\n164\n165\n166\n167\n168\n169\n170\n171\n172\n173\n174\n175\n176\n177\n178\n179\n180\n181\n182\n183\n184\n185\n186\n187\n188\n189\n190\n191\n192\n193\n194\n195\n196\n197\n198\n199\n200\n201\n202\n203\n204\n205\n206\n207\n208\n209\n210\n211\n212\n213\n214\n215\n216\n217\n218\n219\n220\n221\n222\n223\n224\n225\n226\n227\n228\n229\n230\n231\n232\n233\n234\n235\n236\n237\n238\n239\n240\n241\n242\n243\n244\n245\n246\n247\n248\n249\n250\n251\n252\n253\n254\n255\n256\n257\n258\n259\n260\n261\n262\n263\n264\n265\n266\n267\n268\n269\n270\n271\n272\n273\n274\n275\n276\n277\n278\n279\n280\n281\n282\n283\n284\n285\n286\n287\n288\n289\n290\n291\n292\n293\n294\n295\n296\n297\n298\n299\n300\n301\n302\n303\n304\n305\n306\n307\n308\n309\n310\n311\n312\n313\n314\n315\n316\n317\n318\n319\n320\n321\n322\n323\n324\n325\n326\n327\n328\n329\n330\n331\n332\n333\n334\n335\n336\n337\n338\n339\n340\n341\n342\n343\n344\n345\n346\n347\n348\n349\n350\n351\n352\n353\n354\n355\n356\n357\n358\n359\n360\n361\n362\n363\n364\n365\n366\n367\n368\n369\n370\n371\n372\n373\n374\n375\n376\n377\n378\n379\n380\n381\n382\n383\n384\n385\n386\n387\n388\n389\n390\n391\n392\n393\n394\n395\n396\n397\n398\n399\n399\n400\n401\n402\n403\n404\n405\n406\n407\n408\n409\n410\n411\n412\n413\n414\n415\n416\n417\n418\n419\n420\n421\n422\n423\n424\n425\n426\n427\n428\n429\n430\n431\n432\n433\n434\n435\n436\n437\n438\n439\n440\n441\n442\n443\n444\n445\n446\n447\n448\n449\n450\n451\n452\n453\n454\n455\n456\n457\n458\n459\n459\n460\n461\n462\n463\n464\n465\n466\n467\n468\n469\n470\n471\n472\n473\n474\n475\n476\n477\n478\n479\n480\n481\n482\n483\n484\n485\n486\n487\n488\n489\n489\n490\n491\n492\n493\n494\n495\n496\n497\n498\n499\n499\n500\n501\n502\n503\n504\n505\n506\n507\n508\n509\n509\n510\n511\n512\n513\n514\n515\n516\n517\n517\n518\n519\n519\n520\n521\n522\n523\n524\n525\n526\n527\n527\n528\n529\n529\n530\n531\n532\n533\n533\n534\n535\n535\n536\n536\n537\n537\n538\n538\n539\n539\n540\n540\n541\n541\n542\n542\n543\n543\n544\n544\n545\n545\n546\n546\n547\n547\n548\n548\n549\n549\n550\n550\n551\n551\n552\n552\n553\n553\n554\n554\n555\n555\n556\n556\n557\n557\n558\n558\n559\n559\n560\n560\n561\n561\n562\n562\n563\n563\n564\n564\n565\n565\n566\n566\n567\n567\n568\n568\n569\n569\n570\n570\n571\n571\n572\n572\n573\n573\n574\n574\n575\n575\n576\n576\n577\n577\n578\n578\n579\n579\n580\n580\n581\n581\n582\n582\n583\n583\n584\n584\n585\n585\n586\n586\n587\n587\n588\n588\n589\n589\n590\n590\n591\n591\n592\n592\n593\n593\n594\n594\n595\n595\n596\n596\n597\n597\n598\n598\n599\n599\n600\n600\n601\n601\n602\n602\n603\n603\n604\n604\n605\n605\n606\n606\n607\n607\n608\n608\n609\n609\n610\n610\n611\n611\n612\n612\n613\n613\n614\n614\n615\n615\n616\n616\n617\n617\n618\n618\n619\n619\n620\n620\n621\n621\n622\n622\n623\n623\n624\n624\n625\n625\n626\n626\n627\n627\n628\n628\n629\n629\n630\n630\n631\n631\n632\n632\n633\n633\n634\n634\n635\n635\n636\n636\n637\n637\n638\n638\n639\n639\n640\n640\n641\n641\n642\n642\n643\n643\n644\n644\n645\n645\n646\n646\n647\n647\n648\n648\n649\n649\n650\n650\n651\n651\n652\n652\n653\n653\n654\n654\n655\n655\n656\n656\n657\n657\n658\n658\n659\n659\n660\n660\n661\n661\n662\n662\n663\n663\n664\n664\n665\n665\n666\n666\n667\n667\n668\n668\n669\n669\n670\n670\n671\n671\n672\n672\n673\n673\n674\n674\n675\n675\n676\n676\n677\n677\n678\n678\n679\n679\n680\n680\n681\n681\n682\n682\n683\n683\n684\n684\n685\n685\n686\n686\n687\n687\n688\n688\n689\n689\n690\n690\n691\n691\n692\n692\n693\n693\n694\n694\n695\n695\n696\n696\n697\n697\n698\n698\n699\n699\n700\n700\n701\n701\n702\n702\n703\n703\n704\n704\n705\n705\n706\n706\n707\n707\n708\n708\n709\n709\n710\n710\n711\n711\n712\n712\n713\n713\n714\n714\n715\n715\n716\n716\n717\n717\n718\n718\n719\n719\n720\n720\n721\n721\n722\n722\n723\n723\n724\n724\n725\n725\n726\n726\n727\n727\n728\n728\n729\n729\n730\n730\n731\n731\n732\n732\n733\n733\n734\n734\n735\n735\n736\n736\n737\n737\n738\n738\n739\n739\n740\n740\n741\n741\n742\n742\n743\n743\n744\n744\n745\n745\n746\n746\n747\n747\n748\n748\n749\n749\n750\n750\n751\n751\n752\n752\n753\n753\n754\n754\n755\n755\n756\n756\n757\n757\n758\n758\n759\n759\n760\n760\n761\n761\n762\n762\n763\n763\n764\n764\n765\n765\n766\n766\n767\n767\n768\n768\n769\n769\n770\n770\n771\n771\n772\n772\n773\n773\n774\n774\n775\n775\n776\n776\n777\n777\n778\n778\n779\n779\n780\n780\n781\n781\n782\n782\n783\n783\n784\n784\n785\n785\n786\n786\n787\n787\n788\n788\n789\n789\n790\n790\n791\n791\n792\n792\n793\n793\n794\n794\n795\n795\n796\n796\n797\n797\n798\n798\n799\n799\n800\n800\n801\n801\n802\n802\n803\n803\n804\n804\n805\n805\n806\n806\n807\n807\n808\n808\n809\n809\n810\n810\n811\n811\n812\n812\n813\n813\n814\n814\n815\n815\n816\n816\n817\n817\n818\n818\n819\n819\n820\n820\n821\n821\n822\n822\n823\n823\n824\n824\n825\n825\n826\n826\n827\n827\n828\n828\n829\n829\n830\n830\n831\n831\n832\n832\n833\n833\n834\n834\n835\n835\n836\n836\n837\n837\n838\n838\n839\n839\n840\n840\n841\n841\n842\n842\n843\n843\n844\n844\n845\n845\n846\n846\n847\n847\n848\n848\n849\n849\n850\n850\n851\n851\n852\n852\n853\n853\n854\n854\n855\n855\n856\n856\n857\n857\n858\n858\n859\n859\n860\n860\n861\n861\n862\n862\n863\n863\n864\n864\n865\n865\n866\n866\n867\n867\n868\n868\n869\n869\n870\n870\n871\n871\n872\n872\n873\n873\n874\n874\n875\n875\n876\n876\n877\n877\n878\n878\n879\n879\n880\n880\n881\n881\n882\n882\n883\n883\n884\n884\n885\n885\n886\n886\n887\n887\n888\n888\n889\n889\n890\n890\n891\n891\n892\n892\n893\n893\n894\n894\n895\n895\n896\n896\n897\n897\n898\n898\n899\n899\n900\n900\n901\n901\n902\n902\n903\n903\n904\n904\n905\n905\n906\n906\n907\n907\n908\n908\n909\n909\n910\n910\n911\n911\n912\n912\n913\n913\n914\n914\n915\n915\n916\n916\n917\n917\n918\n918\n919\n919\n920\n920\n921\n921\n922\n922\n923\n923\n924\n924\n925\n925\n926\n926\n927\n927\n928\n928\n929\n929\n930\n930\n931\n931\n932\n932\n933\n933\n934\n934\n935\n935\n936\n936\n937\n937\n938\n938\n939\n939\n940\n940\n941\n941\n942\n942\n943\n943\n944\n944\n945\n945\n946\n946\n947\n947\n948\n948\n949\n949\n950\n950\n951\n951\n952\n952\n953\n953\n954\n954\n955\n955\n956\n956\n957\n957\n958\n958\n959\n959\n960\n960\n961\n961\n962\n962\n963\n963\n964\n964\n965\n965\n966\n966\n967\n967\n968\n968\n969\n969\n970\n970\n971\n971\n972\n972\n973\n973\n974\n974\n975\n975\n976\n976\n977\n977\n978\n978\n979\n979\n980\n980\n981\n981\n982\n982\n983\n983\n984\n984\n985\n985\n986\n986\n987\n987\n988\n988\n989\n989\n990\n990\n991\n991\n992\n992\n993\n993\n994\n994\n995\n995\n996\n996\n997\n997\n998\n998\n999\n999\n1000\n1000\n1001\n1001\n1002\n1002\n1003\n1003\n1004\n1004\n1005\n1005\n1006\n1006\n1007\n1007\n1008\n1008\n1009\n1009\n1010\n1010\n1011\n1011\n1012\n1012\n1013\n1013\n1014\n1014\n1015\n1015\n1016\n1016\n1017\n1017\n1018\n1018\n1019\n1019\n1020\n1020\n1021\n1021\n1022\n1022\n1023\n1023\n1024\n1024\n1025\n1025\n1026\n1026\n1027\n1027\n1028\n1028\n1029\n1029\n1030\n1030\n1031\n1031\n1032\n1032\n1033\n1033\n1034\n1034\n1035\n1035\n1036\n1036\n1037\n1037\n1038\n1038\n1039\n1039\n1040\n1040\n1041\n1041\n1042\n1042\n1043\n1043\n1044\n1044\n1045\n1045\n1046\n1046\n1047\n1047\n1048\n1048\n1049\n1049\n1050\n1050\n1051\n1051\n1052\n1052\n1053\n1053\n1054\n1054\n1055\n1055\n1056\n1056\n1057\n1057\n1058\n1058\n1059\n1059\n1060\n1060\n1061\n1061\n1062\n1062\n1063\n1063\n1064\n1064\n1065\n1065\n1066\n1066\n1067\n1067\n1068\n1068\n1069\n1069\n1070\n1070\n1071\n1071\n1072\n1072\n1073\n1073\n1074\n1074\n1075\n1075\n1076\n1076\n1077\n1077\n1078\n1078\n1079\n1079\n1080\n1080\n1081\n1081\n1082\n1082\n1083\n1083\n1084\n1084\n1085\n1085\n1086\n1086\n1087\n1087\n1088\n1088\n1089\n1089\n1090\n1090\n1091\n1091\n1092\n1092\n1093\n1093\n1094\n1094\n1095\n1095\n1096\n1096\n1097\n1097\n1098\n1098\n1099\n1099\n1100\n1100\n1101\n1101\n1102\n1102\n1103\n1103\n1104\n1104\n1105\n1105\n1106\n1106\n1107\n1107\n1108\n1108\n1109\n1109\n1110\n1110\n1111\n1111\n1112\n1112\n1113\n1113\n1114\n1114\n1115\n1115\n1116\n1116\n1117\n1117\n1118\n1118\n1119\n1119\n1120\n1120\n1121\n1121\n1122\n1122\n1123\n1123\n1124\n1124\n1125\n1125\n1126\n1126\n1127\n1127\n1128\n1128\n1129\n1129\n1130\n1130\n1131\n1131\n1132\n1132\n1133\n1133\n1134\n1134\n1135\n1135\n1136\n1136\n1137\n1137\n1138\n1138\n1139\n1139\n1140\n1140\n1141\n1141\n1142\n1142\n1143\n1143\n1144\n1144\n1145\n1145\n1146\n1146\n1147\n1147\n1148\n1148\n1149\n1149\n1150\n1150\n1151\n1151\n1152\n1152\n1153\n1153\n1154\n1154\n1155\n1155\n1156\n1156\n1157\n1157\n1158\n1158\n1159\n1159\n1160\n1160\n1161\n1161\n1162\n1162\n1163\n1163\n1164\n1164\n1165\n1165\n1166\n1166\n1167\n1167\n1168\n1168\n1169\n1169\n1170\n1170\n1171\n1171\n1172\n1172\n1173\n1173\n1174\n1174\n1175\n1175\n1176\n1176\n1177\n1177\n1178\n1178\n1179\n1179\n1180\n1180\n1181\n1181\n1182\n1182\n1183\n1183\n1184\n1184\n1185\n1185\n1186\n1186\n1187\n1187\n1188\n1188\n1189\n1189\n1190\n1190\n1191\n1191\n1192\n1192\n1193\n1193\n1194\n1194\n1195\n1195\n1196\n1196\n1197\n1197\n1198\n1198\n1199\n1199\n1200\n1200\n1201\n1201\n1202\n1202\n1203\n1203\n1204\n1204\n1205\n1205\n1206\n1206\n1207\n1207\n1208\n1208\n1209\n1209\n1210\n1210\n1211\n1211\n1212\n1212\n1213\n1213\n1214\n1214\n1215\n1215\n1216\n1216\n1217\n1217\n1218\n1218\n1219\n1219\n1220\n1220\n1221\n1221\n1222\n1222\n1223\n1223\n1224\n1224\n1225\n1225\n1226\n1226\n1227\n1227\n1228\n1228\n1229\n1229\n1230\n1230\n1231\n1231\n1232\n1232\n1233\n1233\n1234\n1234\n1235\n1235\n1236\n1236\n1237\n1237\n1238\n1238\n1239\n1239\n1240\n1240\n1241\n1241\n1242\n1242\n1243\n1243\n1244\n1244\n1245\n1245\n1246\n1246\n1247\n1247\n1248\n1248\n1249\n1249\n1250\n1250\n1251\n1251\n1252\n1252\n1253\n1253\n1254\n1254\n1255\n1255\n1256\n1256\n1257\n1257\n1258\n1258\n1259\n1259\n1260\n1260\n1261\n1261\n1262\n1262\n1263\n1263\n1264\n1264\n1265\n1265\n1266\n1266\n1267\n1267\n1268\n1268\n1269\n1269\n1270\n1270\n1271\n1271\n1272\n1272\n1273\n1273\n1274\n1274\n1275\n1275\n1276\n1276\n1277\n1277\n1278\n1278\n1279\n1279\n1280\n1280\n1281\n1281\n1282\n1282\n1283\n1283\n1284\n1284\n1285\n1285\n1286\n1286\n1287\n1287\n1288\n1288\n1289\n1289\n1290\n1290\n1291\n1291\n1292\n1292\n1293\n1293\n1294\n1294\n1295\n1295\n1296\n1296\n1297\n1297\n1298\n1298\n1299\n1299\n1300\n1300\n1301\n1301
```

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)

The screenshot shows a PyCharm IDE interface with several windows open. The main window displays a Python script named `vit_transformer.py` containing code for an EncoderBlock. The code includes various operations like LayerNorm, Linear, and Feed-forward layers, with multiple breakpoints set across the file. A call stack window is visible at the bottom, showing the execution path from the main function down through various helper methods and tensor operations. The status bar at the bottom right indicates the current file is `vit_transformer.py`, and the Python version is 3.13.

Image

Description: Feed-forward output $\text{ff_out} \in 1 \times 197 \times 768$.

Explanation: Second linear projects back to d_{model} .

S19 — Residual + norm (post-MLP)

Image

Description: Post-MLP normalized tensor $x_{\text{post_mlp_norm}} \in 1 \times 197 \times 768$.

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

S20 — Encoder block output

```
class ViT(nn.Module): 1 usage
  def forward(self, x):
    ...  
    x_pos = x.tokens + self.pos_embed # (6, 197, 768) x_pos: tensor([[ 7.9809e-03, 4.2521e-03, 5.1668e-04, ..., 8.7603e-02], [ 0.0531e-02, 0.0531e-02, 0.0531e-02, ..., 0.0531e-02], ...])  
    # BREAKPOINT S07 - Embeddings AFTER adding positional encoding  
    ...  
    # ----- Encoder blocks -----  
    x_block1_out = self.blocks[0](x_pos) # S08...S20 inside the block x_block1_out: tensor([[[-0.1041, 0.8805, -0.8489, ..., 0.1518, -0.2089, ..., 0.1184], [-0.1041, 0.8805, -0.8489, ..., 0.1518, -0.2089, ..., 0.1184], ...]])  
    # BREAKPOINT S21 - Encoder block 2 output  
    x_block2_out = self.blocks[1](x_block1_out) # x_block2_out: tensor([[[-0.1268, 1.0050, -0.8176, ..., 0.0793, -0.1473, 0.9456], [-0.1268, 1.0050, -0.8176, ..., 0.0793, -0.1473, 0.9456], ...]])  
    ...  
    x_curr = x_block2_out  
    for i in range(2, len(self.blocks)):  
        x_curr = self.blocks[i](x_curr)  
    ...  
    # BREAKPOINT S22 - Encoder block N (last) output  
    x_last = x_curr  
    ...  
    # BREAKPOINT S23 - Final sequence output (including class token)  
    final_seq = x_last  
    ...
```

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

The screenshot shows a debugger interface with the following details:

- Code View:** The code for the `ViT` class is displayed, specifically the `forward` method. A red box highlights the line where `x_block2_out` is assigned, which is the output of the second encoder block.
- Breakpoints:** Several breakpoints are set across the code, indicated by red dots.
- Call Stack:** The call stack shows the execution flow from the main thread down to the `forward` method of `ViT_transformer.py`. It includes frames for `patch_embed`, `PatchEmbed`, `ViT`, and `ViT_transformer`.
- Registers:** A register dump is visible at the bottom of the stack.
- Registers:** The registers section shows various memory addresses and their values.
- Registers:** The registers section shows various memory addresses and their values.

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

The screenshot shows a PyCharm interface with several annotations and breakpoints applied to the code. The code is a class definition for `ViT(nn.Module)`, which includes methods for forward passes through different blocks and a final sequence output.

Annotations include:

- # BREAKPOINT S21 - Encoder block 1 output
- # BREAKPOINT S22 - Encoder block 2 output
- # BREAKPOINT S23 - Final sequence output (including class token)
- # BREAKPOINT S24 - Class token extracted (final representation)

Breakpoints are marked with red dots on specific lines, such as `x_block1_out`, `x_block2_out`, `x_last`, `final_seq`, `cls_rep`, and `logits`.

Callouts and tool tips provide detailed information about the annotated lines, such as:

- Annotations for `x_block1_out` and `x_block2_out` mention "S08...S20 inside the block".
- An annotation for `x_last` mentions "Tensor([[-0.1660, 0.9085, -2.2163, ..., 1.1964, 0.1596, -0.3823], [1.2968, 0.5651, -1.6427, ..., 0.8320, 0.7551, -0.1042]]".
- An annotation for `final_seq` mentions "Tensor([[-0.1660, 0.9085, -2.2163, ..., 1.1964, 0.1596, -0.3823], [1.2968, 0.5651, -1.6427, ..., 0.8320, 0.7551, -0.1042]]".
- An annotation for `cls_rep` mentions "Tensor([[-0.1660, 0.9085, -2.2163, ..., 1.1964, 0.1596, -0.3823], [1.2968, 0.5651, -1.6427, ..., 0.8320, 0.7551, -0.1042]]".
- An annotation for `logits` mentions "Tensor([[-0.1660, 0.9085, -2.2163, ..., 1.1964, 0.1596, -0.3823], [1.2968, 0.5651, -1.6427, ..., 0.8320, 0.7551, -0.1042]]").

The bottom status bar indicates the current file is `vit_transformer.py`, and the Python version is 3.13.

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])

The screenshot shows a Jupyter Notebook interface with several cells of Python code for a ViT transformer. The code includes forward passes through the patch embed layer, blocks, and classification head, along with softmax and logit calculations. A 'Breakpoint' is set at the softmax line. The 'Threads & Variables' pane is open, showing the current thread and variables like `cls_rep`, `logits`, and `probs`. The 'Evaluate expression' dropdown is active, showing the value of `cls_rep` as a tensor of shape (768, 1) containing [1.6595e-01, 9.0854e-01]. The status bar at the bottom indicates the file is 'Sheet1_Assignment2.ipynb'.

```
class ViT(nn.Module): 1 usage
  def forward(self, x):  self: ViT\n    (patch_embed): PatchEmbed(\n        (proj): Linear(in_features=768, out_features=768, bias=True)\n    )\n    (blocks): ModuleList(\n        (x_block2_out): x_curr: tensor([[-0.1660, 0.9085, -2.2163, ..., 1.1964, 0.1596, -0.3823],\n        for i in range(2, len(self.blocks)): i: 11\n            x_curr = self.blocks[i](x_curr)\n\n        # BREAKPOINT S22 - Encoder block N (last) output\n        x_last = x_curr  x.last: tensor([[[-0.1660, 0.9085, -2.2163, ..., 1.1964, 0.1596, -0.3823],\n\n        # BREAKPOINT S23 - Final sequence output (including class token)\n        final_seq = x_last  final_seq: tensor([[[-0.1660, 0.9085, -2.2163, ..., 1.1964, 0.1596, -0.3823],\n\n        # BREAKPOINT S24 - Class token extracted (final representation)\n        cls_rep = final_seq[:, 0, :] # (B, 768)  cls.rep: tensor([[1.6595e-01, 9.0854e-01, -2.2163e+00, 1.2217e+00,\n\n        logits = self.head(cls_rep) # (B, 1000)\n\n        # Breakpoint S25 - Classification head logits\n\n        probs = logits.softmax(dim=-1) # (B, 1000)
```

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)

The screenshot shows a Jupyter Notebook environment with the following details:

- File Bar:** Sheet1_Assignment2, Version control, Current File, Trial.
- Toolbar:** Run, Kernel, Help, etc.
- Code Cell:** A Python class definition for `ViT` (ViT transformer) with several methods like `forward`, `logits`, and `probs`.
- Breakpoints:** Breakpoints are set at lines 164, 193, 195, 197, 198, and 200.
- Call Stack:** Shows the call history of the current cell, starting from `forward` down to `__init__`.
- Variables:** A list of variables in the current scope, including `cls`, `cls_head`, `cls_logit`, `cls_norm`, `cls_token`, `cls_type`, `cls_value`, `final_seq`, `logits`, `patch_embed`, `prob`, `prob_fn`, `proj`, `proj_fn`, `tokens`, and `tokens_fn`.
- Console:** Displays the output of the last cell, which includes tensor operations and numerical values.

Image

Description: Class token representation $\text{cls_rep} \in 1 \times 768$.

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

S25 — Logits

The screenshot shows a PyCharm IDE window with the following details:

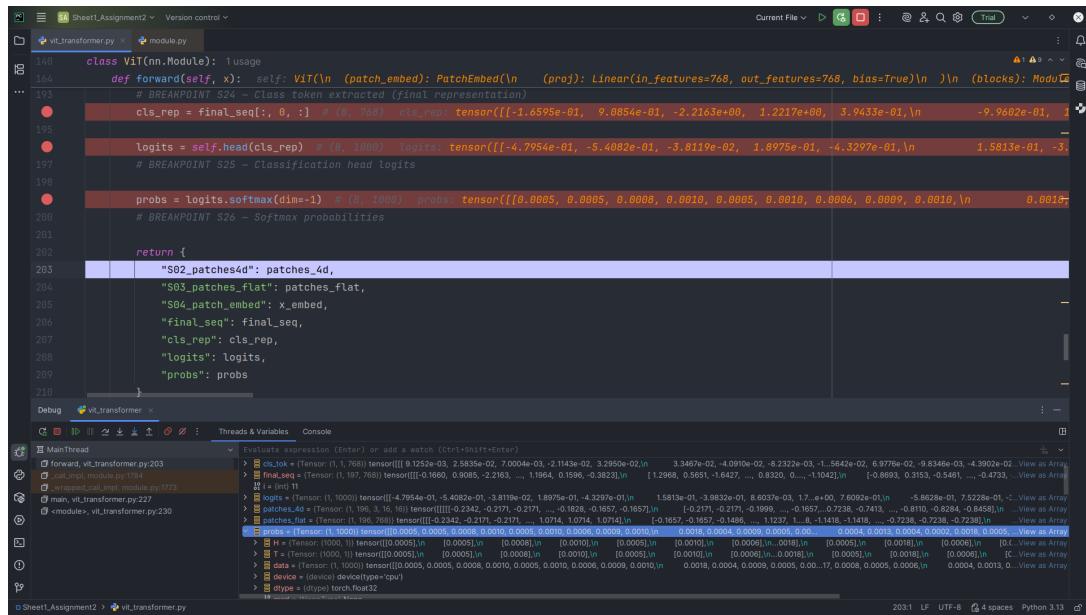
- File:** vit_transformer.py
- Toolbars:** Standard toolbar with icons for file operations.
- Code Editor:** Displays the `ViT(nn.Module)` class definition. The code includes several annotations:
 - # BREAKPOINT S24 - Class token extracted (final representation) at line 193.
 - # BREAKPOINT S25 - Classification head logits at line 195.
 - # BREAKPOINT S26 - Softmax probabilities at line 198.
- Annotations:** Numerous red circular markers are placed along the code, primarily between lines 193 and 198, indicating specific points of interest or errors.
- Bottom Status Bar:** Shows the current file as "vit_transformer.py", the line number as 202, and the column number as 1.

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 ($768 \rightarrow 1000$) 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.