

Sheet 1 assignment 1

Team 6

Debugging Python (pytorch) transformer code based on **standard encoder–decoder Transformer** (Vaswani et al., 2017). It is a reduced model for debugging 2 encoder layers, 2 decoder layers, 4 attention heads, and 128 dimension embedding.

The input is `[1, 5, 6, 4, 3, 9, 5, 2, 0]`.

The target is `[1, 7, 4, 3, 5, 9, 2, 0]`.

Python version: 3.11.9

Pytorch version: 2.8.0

The code link:

https://github.com/Yahia-Ragab/Image-Processing-Projects/tree/main/sheet1_assignment1

Guiding Questions

1)What do each of the dimensions represent at embedding, attention, feed-forward, and output stages?

Embeddings: (batch, seq len, embedding dim), Attention: (batch, seq len, heads, heads dim), feed forward (batch, seq len, vocab size)

2)Why do Q, K, V tensors have the same shape, and why are they split into heads?

Same shape due to sharing the same linear projections and input. Split to let multiple attention to run in parallel.

3)What do the attention score matrices represent, and why must they be square?

Query key similarities, to use self-attention so each token can attend to all others in the sequence.

4)Why is masking necessary in the decoder, and how does the mask tensor enforce it?

To stop the decoder from attending to future or padded tokens by forcing softmax to zero out masked positions.

5)How do residual connections and layer normalization ensure consistency of shapes across blocks?

To keep shapes the same across layers

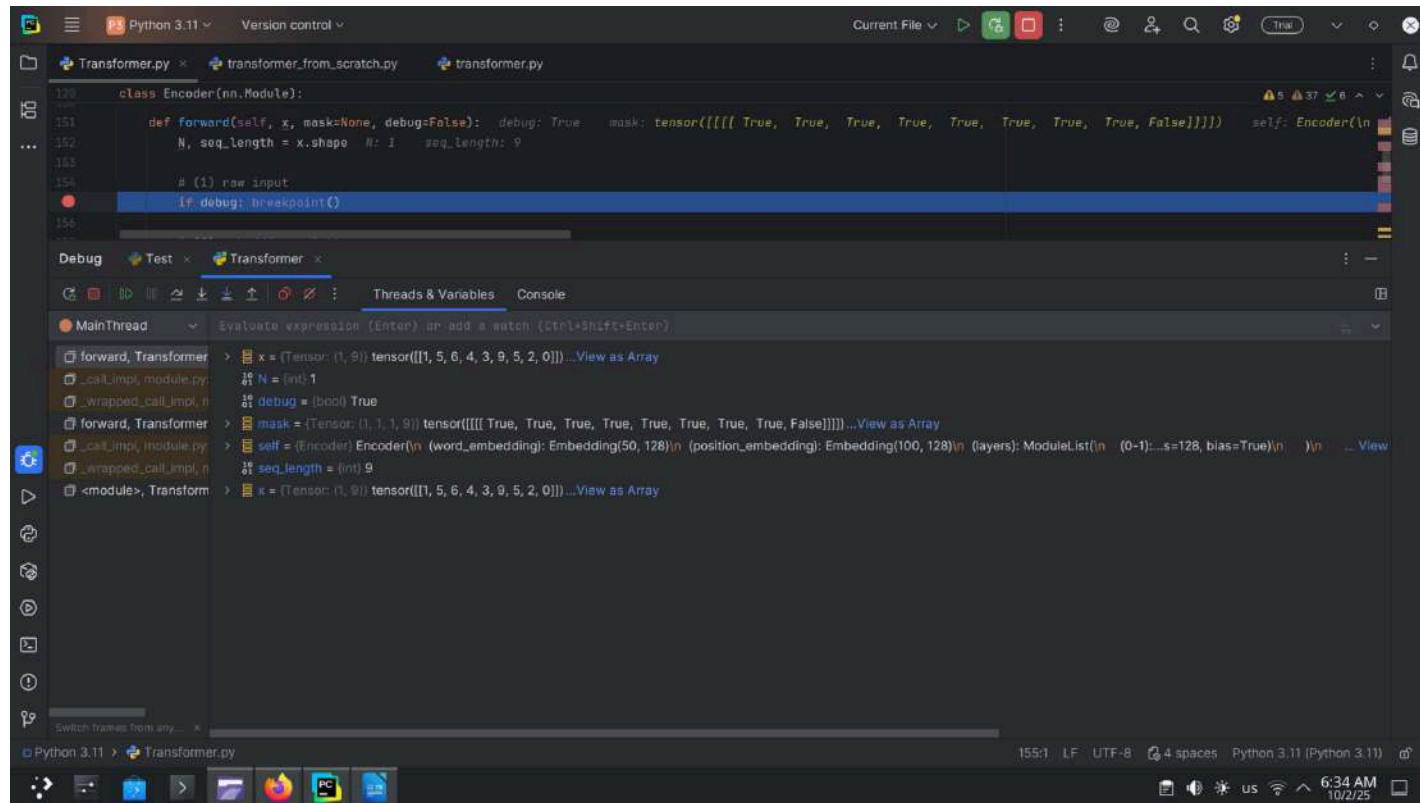
6)Why must the embedding dimension remain constant through all layers?

To ensure residuals, attention, ff blocks align without mismatch

7)How does the final projection connect decoder output to vocabulary logits?

Maps decoder output from embedding dim to vocab size

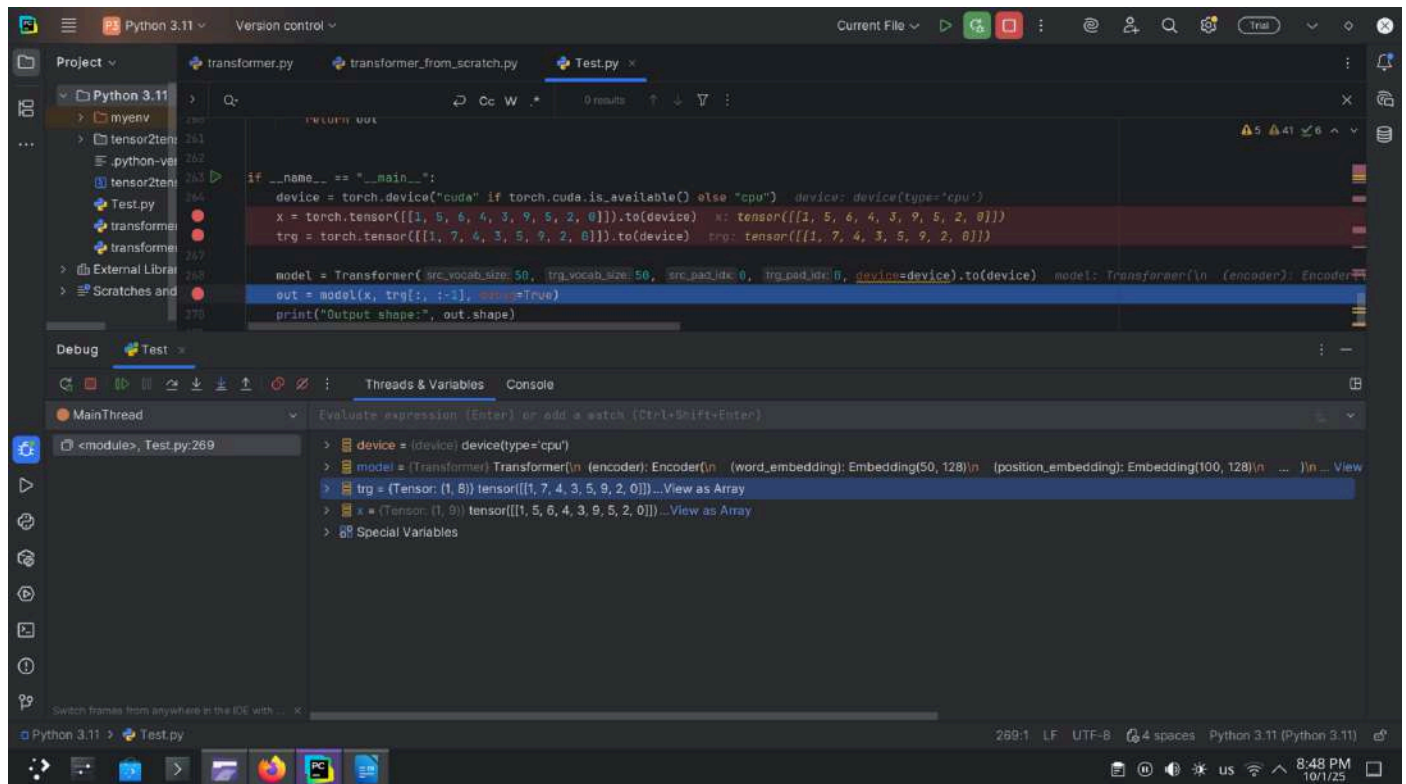
Snapshot 1: Raw input tokens(ID)



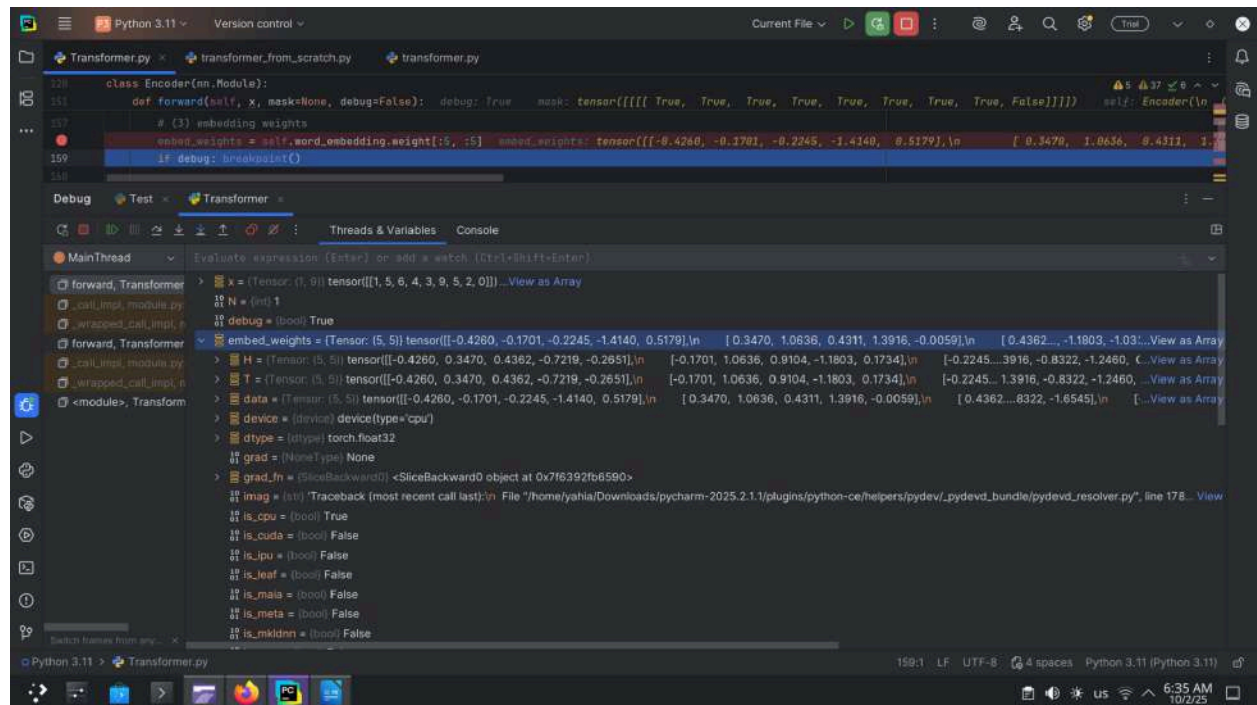
Tensor shape: (1, 9)

values: [1, 5, 6, 4, 3, 9, 5, 2, 0]

Snapshot 2 : Target tokens (ID)



Snapshot 3 : Embedding weight matrix (slice, e.g., 5x5)



Tensor shape: 5x5

Slice of values: [0.0490, -0.1884, -1.1856, 0.6156, -0.5038], [-1.8775, -0.6953, -2.5102, -0.3088, 0.1844]]

This is a small slice of the word embedding weight matrix ($n \times d$) each row corresponds to the learned vector representation of one token ID.

Snapshot 4: Input embeddings after lookup:

```
class Encoder(nn.Module):
    def forward(self, x, mask=None, debug=False):
        word_emb = self.word_embedding(x)
        if debug: breakpoint()

        # (5) embeddings + positional
        positions = torch.arange(0, seq_length).unsqueeze(0).expand(N, seq_length).to(self.device)
        pos_emb = self.position_embedding(positions)

        Debug
        Test
        Transformer
        Threads & Variables
        Console

MainThread
Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

forward, Transformer
> x = (Tensor(1, 9)) tensor([1, 5, 6, 4, 3, 9, 5, 2, 0]) ...View as Array
> N = (int) 1
> debug = (bool) True
forward, Transformer
> embed_weights = (Tensor(5, 5)) tensor([[-0.4260, -0.1701, -0.2245, -1.4140, 0.5179],
[ 0.3470, 1.0636, 0.4311, 1.3916, -0.0059],
[ 0.4362..., -1.1803, -1.031..., ...View as Array
> mask = (Tensor(1, 1, 9)) tensor([[[[ True, True, True, True, True, True, True, True, False]]]]) ...View as Array
> self = (Encoder) Encoder(in (word_embedding): Embedding(50, 128) (position_embedding): Embedding(100, 128) (layers): ModuleList(0-1) ...s=128, bias=True) ...View
> seq_length = (int) 9
> word_emb = (Tensor(1, 9, 128)) tensor([[[[ 0.3470, 1.0636, 0.4311, ..., 0.7354, 1.5248, -0.3489],
[-1.6929, 0.4121, -0.9850, ..., -0.3051, -0.12001, -1.1797],
[ 0.4362..., -1.1803, -1.031..., ...View as Array
> H = (str) 'Traceback (most recent call last):
File "/home/yahia/Downloads/pycharm-2025.2.1/plugins/python-ce/helpers/pydev/pydevd_bundle/pydevd_resolver.py", line 178, in ...View
> T = (Tensor(128, 9, 1)) tensor([[[[ 0.3470],
[-1.6929],
[ 1.1206],
...,
[-1.6929],
[ 0.4362],
[-0.9],
[ 0.5039],
..., ...View as Array
> data = (Tensor(1, 9, 128)) tensor([[[[ 0.3470, 1.0636, 0.4311, ..., 0.7354, 1.5248, -0.3489],
[-1.6929, 0.4121, -0.9850, ..., -0.3051, -0.12001, -1.1797],
[ 0.4362..., -1.1803, -1.031..., ...View as Array
> device = (device) device(type='cpu')
> dtype = (dtype) torch.float32
> grad = (NoneType) None
> grad_fn = (EmbeddingBackward0) <EmbeddingBackward0 object at 0x7f6392fb5b40>
> msg = (str) 'Traceback (most recent call last):
File "/home/yahia/Downloads/pycharm-2025.2.1/plugins/python-ce/helpers/pydev/pydevd_bundle/pydevd_resolver.py", line 178, in ...View
> is_cpu = (bool) True
```

Tensorshape (1,9,128)

Slice of values: [-1.8775, -0.6953, -2.5102, ..., 0.1594, 0.8897, 0.2964],
[0.8119, -0.6957, -0.3944, ..., 1.3909, 1.2429, 0.7979],
[0.0490, -0.1884, -1.1856, ..., -0.1613, 1.4124, 0.2325]

These are the word embeddings of the 9 input tokens, each mapped into a 128-dimensional dense vector space representing semantic features.

Snapshot 5: Embeddings after adding positional encoding:

```
class Encoder(nn.Module):
    def forward(self, x, mask=None, debug=False):
        # (5) embeddings + positional
        positions = torch.arange(0, seq_length).unsqueeze(0).expand(N, seq_length).to(self.device)
        pos_emb = self.position_embedding(positions)
        out = self.dropout(word_emb + pos_emb)
        if debug: breakpoint()
```

Debug Console:

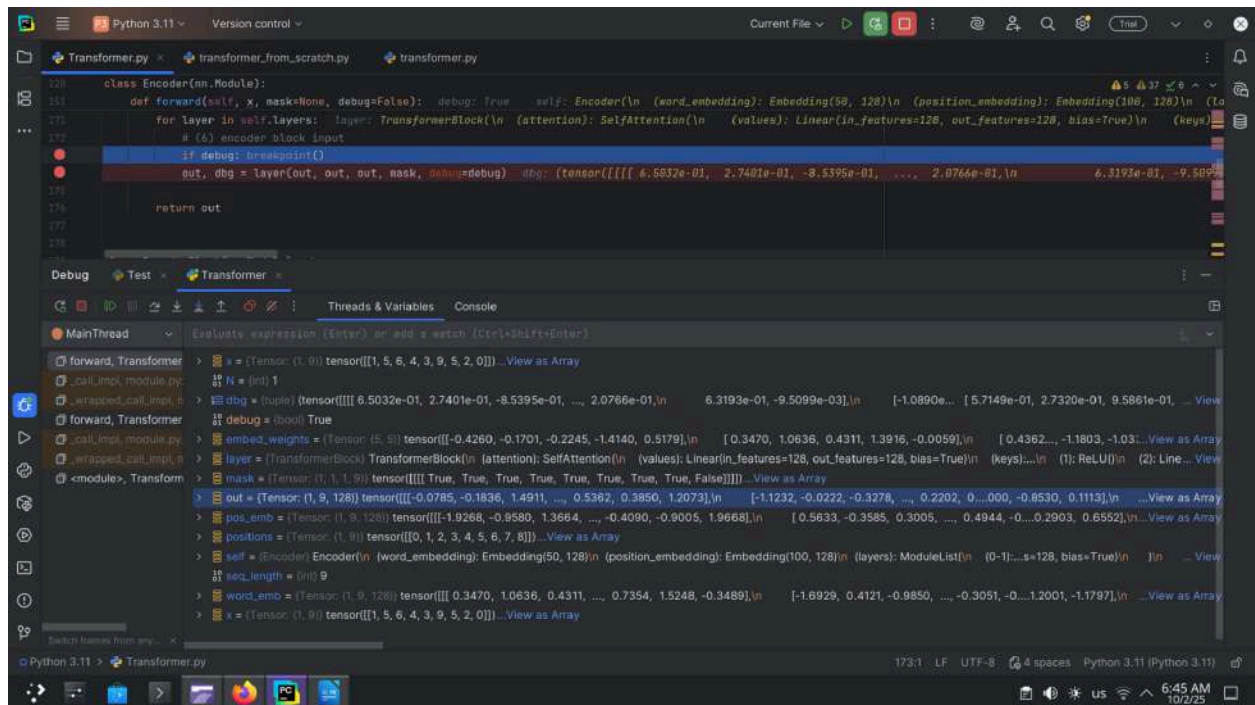
```
forward, Transformer: x = (Tensor(1, 9)) tensor([1, 5, 6, 4, 3, 9, 5, 2, 0])...View as Array
forward, Transformer: mask = (Tensor(1, 1, 9)) tensor([[[[ True, True, True, True, True, True, True, True, False]]]])...View as Array
forward, Transformer: pos_emb = (Tensor(1, 9, 128)) tensor([[-1.9268, -0.9580, 1.3664, ..., -0.4090, -0.9005, 1.9668],\n [ 0.5633, -0.3585, 0.3005, ..., 0.4944, -0.02903, 0.6552],\n [ 0.0000, 0.0000, 1.9973, ..., 0.3627, 0.6937, 1.7976],\n [-1.2550, 0.0596, -0.7605, ..., 0.2103, -0.347, -1.0108, -0.5827],\n [ 0.0000, -0.5701, -1.9230, ..., -0.5970, 1.1535, -0.7985],\n [ 0.0000, -0.5701, -1.9230, ..., -0.5970, 1.1535, -0.7985],\n [ 0.0000, -0.5701, -1.9230, ..., -0.5970, 1.1535, -0.7985],\n [ 0.0000, -0.5701, -1.9230, ..., -0.5970, 1.1535, -0.7985],\n [ 0.0000, -0.5701, -1.9230, ..., -0.5970, 1.1535, -0.7985]]])...View as Array
forward, Transformer: out = (Tensor(1, 9, 128)) tensor([[-0.0000, 0.0000, 1.9973, ..., 0.3627, 0.6937, 1.7976],\n [-1.2550, 0.0596, -0.7605, ..., 0.2103, -0.347, -1.0108, -0.5827],\n [ 0.0000, 0.0000, 1.9973, ..., 0.3627, 0.6937, 1.7976],\n [-1.2550, 0.0596, -0.7605, ..., 0.2103, -0.347, -1.0108, -0.5827],\n [ 0.0000, 0.0000, 1.9973, ..., 0.3627, 0.6937, 1.7976],\n [-1.2550, 0.0596, -0.7605, ..., 0.2103, -0.347, -1.0108, -0.5827],\n [ 0.0000, 0.0000, 1.9973, ..., 0.3627, 0.6937, 1.7976],\n [-1.2550, 0.0596, -0.7605, ..., 0.2103, -0.347, -1.0108, -0.5827],\n [ 0.0000, 0.0000, 1.9973, ..., 0.3627, 0.6937, 1.7976],\n [-1.2550, 0.0596, -0.7605, ..., 0.2103, -0.347, -1.0108, -0.5827]]])...View as Array
```

Tensorshape: (1,9,128)

Slice of values: $\begin{bmatrix} -2.5388, -1.0521, -4.0549, \dots, -0.1709, 1.2688, -1.8986, \\ 1.3252, -1.1009, -0.2238, \dots, 2.0256, 1.3895, 0.2673, \\ 0.0000, -0.5701, -1.9230, \dots, -0.5970, 1.1535, -0.7985 \end{bmatrix}$

This is the sum of word embeddings and positional embeddings, giving each token both semantic meaning and positional context in the sequence.

Snapshot 6: Encoder block input tensor:



```
class Encoder(nn.Module):
    def forward(self, x, mask=None, debug=False):
        for layer in self.layers:
            # (6) encoder block input
            if debug: breakpoint()
            out, dbg = layer(out, out, out, mask, debug=debug)
            dbg: (tensor([[[[ 6.5032e-01, 2.7401e-01, -8.5395e-01, ..., 2.0766e-01, 6.3193e-01, -9.5099e-03],
            return out

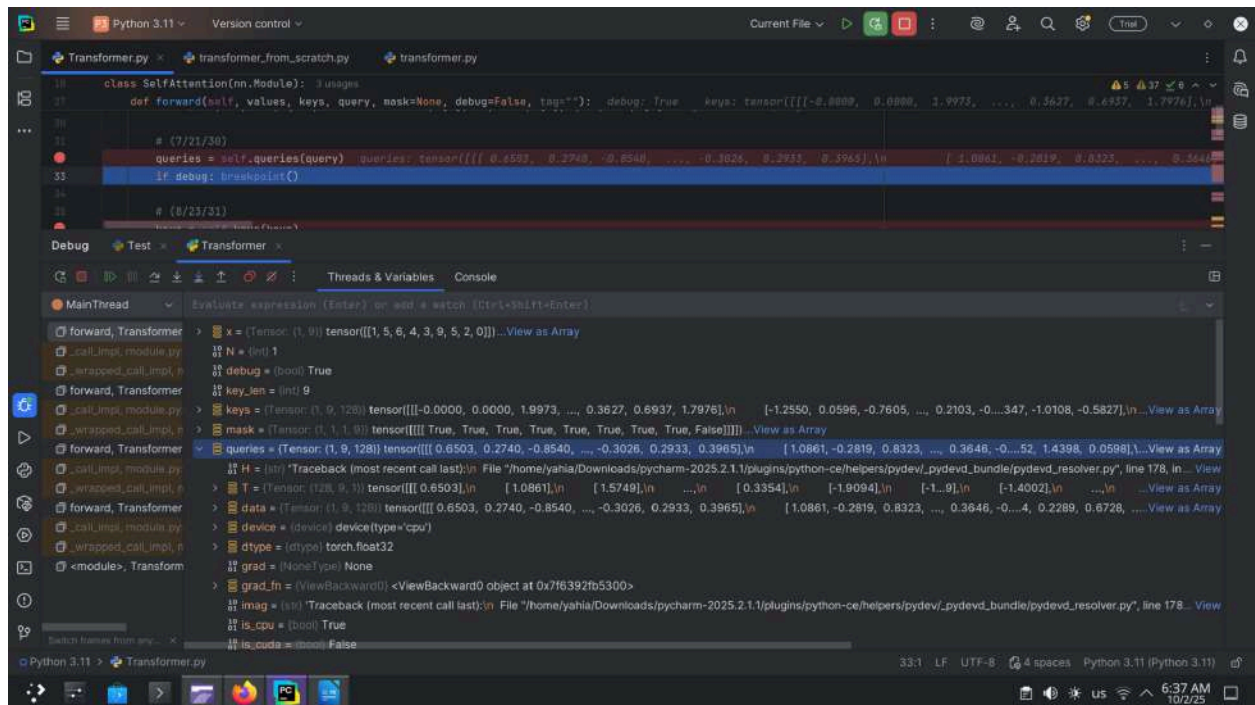
Debug
Test
Transformer
Main Thread
Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)
forward, Transformer
    x = (Tensor of 1, 9) tensor([[[[ 1, 5, 6, 4, 3, 9, 5, 2, 0]]]]) View as Array
    N = (int) 1
    dbg = (tuple) (tensor([[[[ 6.5032e-01, 2.7401e-01, -8.5395e-01, ..., 2.0766e-01, 6.3193e-01, -9.5099e-03],
    debug = (bool) True
    embed_weights = (Tensor of 5, 9) tensor([[[[ -0.4260, -0.1701, -0.2245, -1.4140, 0.5179],
    layer = (TransformerBlock) TransformerBlock((attention): SelfAttention((values): Linear((in_features=128, out_features=128, bias=True))
    mask = (Tensor of 1, 1, 9) tensor([[[[ True, True, True, True, True, True, True, True, False]]]]) View as Array
    pos_emb = (Tensor of 1, 9, 128) tensor([[[[ -0.0785, -0.1836, 1.4911, ..., 0.5382, 0.3850, 1.2073],
    positions = (Tensor of 1, 9) tensor([[[[ 0, 1, 2, 3, 4, 5, 6, 7, 8]]]]) View as Array
    self = (Encoder) Encoder((word_embedding): Embedding(50, 128) (position_embedding): Embedding(100, 128) (layers): ModuleList(0-1)
    seq_length = (int) 9
    word_emb = (Tensor of 1, 9, 128) tensor([[[[ 0.3470, 1.0636, 0.4311, ..., 0.7354, 1.5248, -0.3489],
    x = (Tensor of 1, 9) tensor([[[[ 1, 5, 6, 4, 3, 9, 5, 2, 0]]]]) View as Array
```

Tensorshape: (1,9,128)

Slice of values: $\begin{bmatrix} -2.5388, -1.0521, -4.0549, \dots, -0.1709, 1.2688, -1.8986, \\ 1.3252, -1.1009, -0.2238, \dots, 2.0256, 1.3895, 0.2673, \\ 0.0000, -0.5701, -1.9230, \dots, -0.5970, 1.1535, -0.7985 \end{bmatrix}$

This is the input to the first encoder block, combining word and positional embeddings, ready for multi-head self-attention processing.

Snapshot 7: self-attention Q

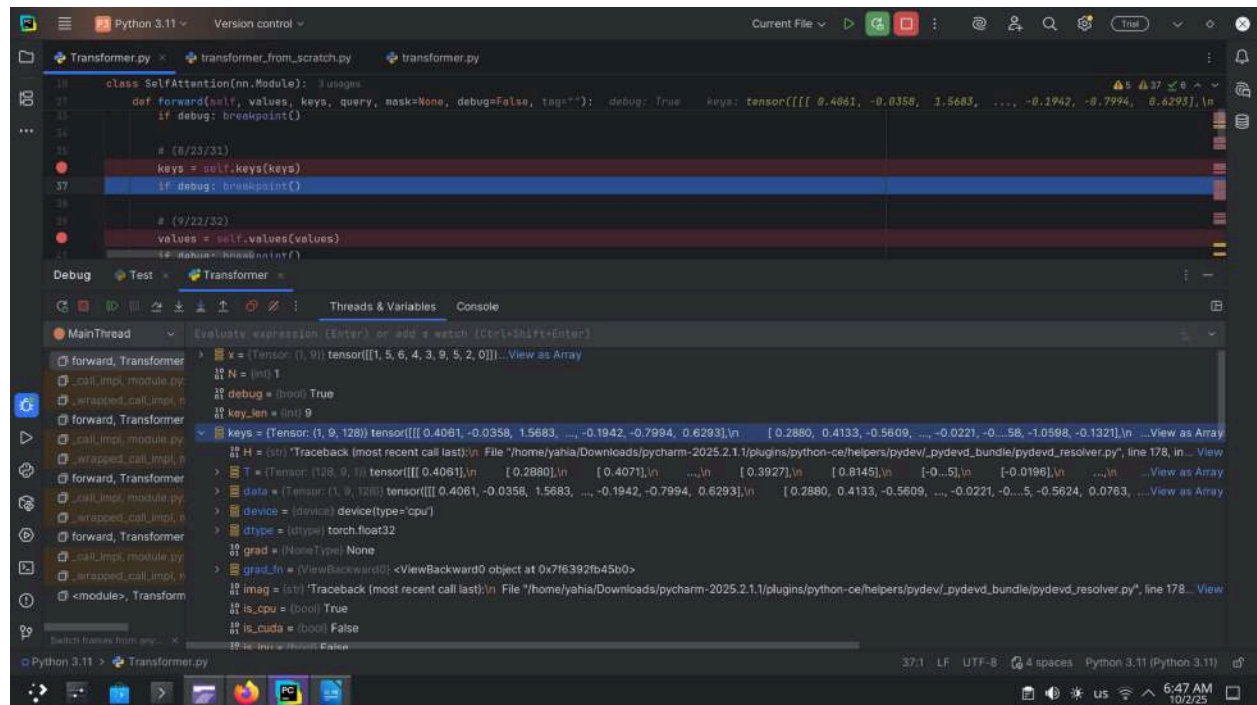


Tensorshape: (1,9,128)

Slice of values: $[-1.8223, -0.8279, -1.8595, \dots, 0.5733, -0.2205, 1.2118],$
 $[-1.3279, -0.8813, -0.3898, \dots, 0.3625, 0.4639, -0.0891],$
 $[-0.2024, -0.8526, -0.1024, \dots, 0.0668, 0.1907, 0.8454]$

these are the transformed queries obtained by projecting the input sequence through a linear layer, preparing them for scaled dot-product attention with keys and values.

Snapshot8: Self attention K:

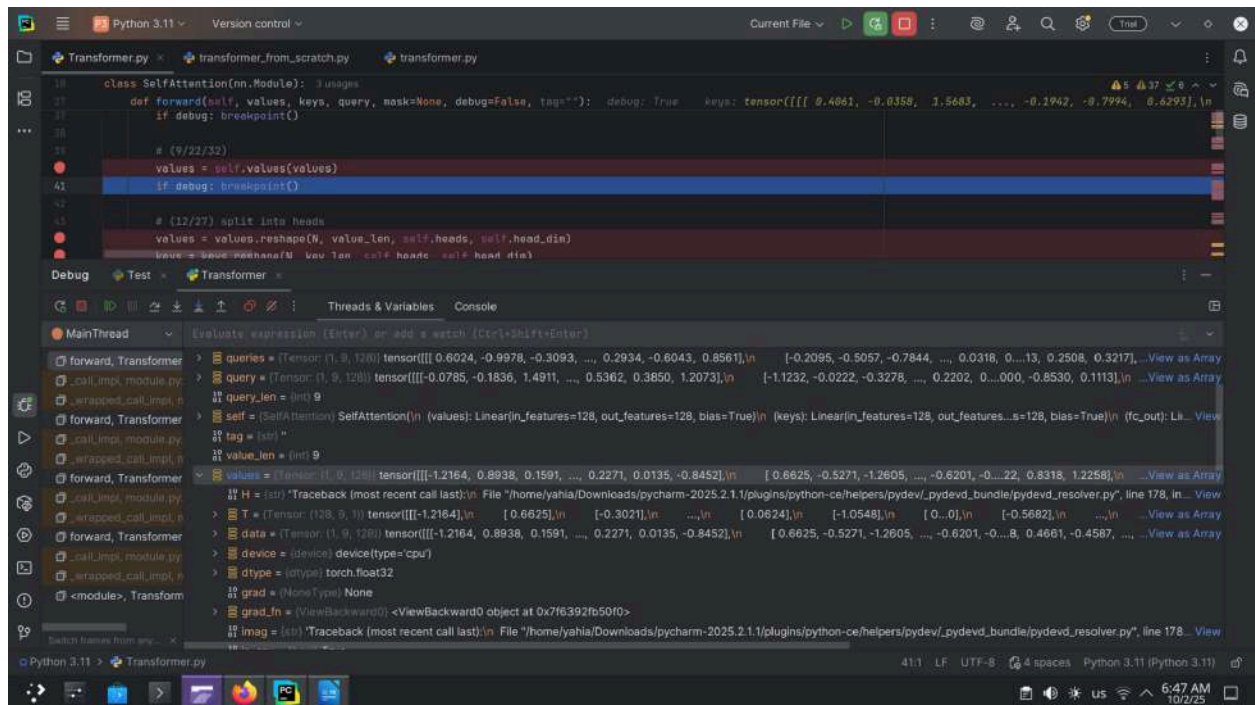


Tensorshape: (1, 9, 128)

Slice of values: `[[0.0354, -0.9729, 0.0476, ..., -0.0135, -0.6007, -0.4278],
[-0.3524, -0.8507, -1.6053, ..., 0.8506, -0.4977, -1.1236],
[-1.1195, 0.2592, -1.1742, ..., -0.5030, -1.2827, -0.1696]]`

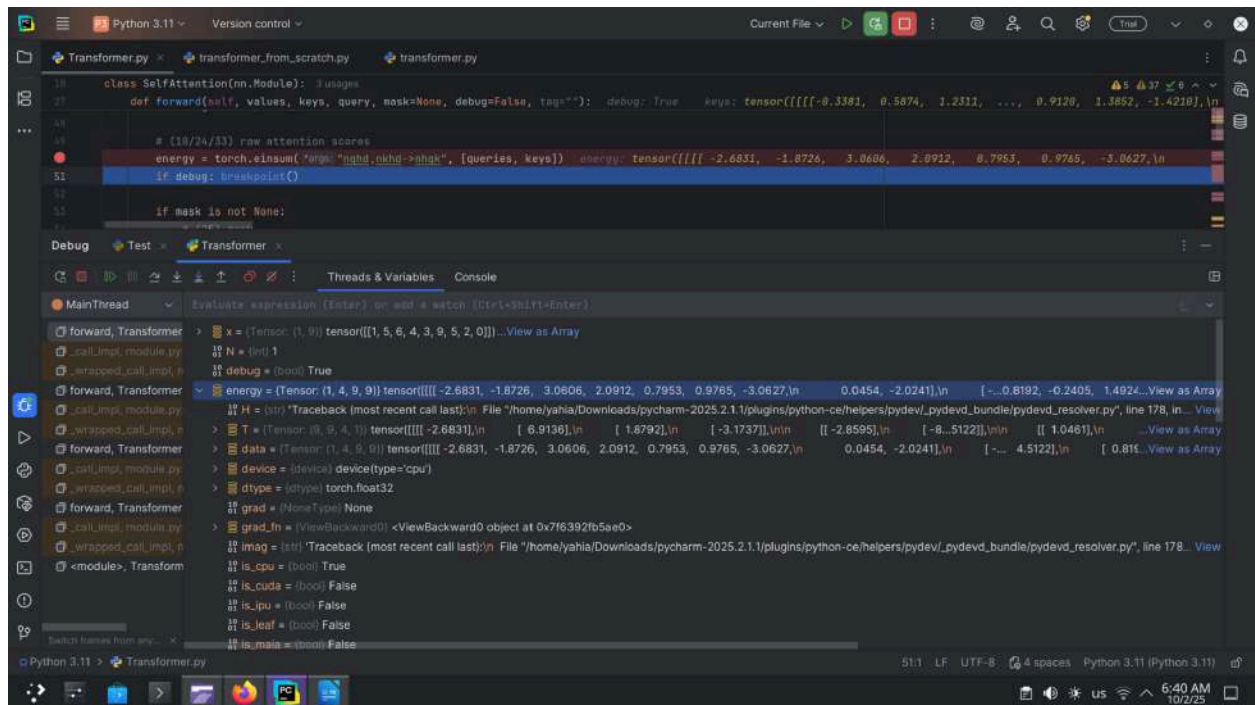
These are the projected keys, obtained by applying a linear transformation to the input sequence, which will be compared with queries during the attention score calculation.

Snapshot 9: Self-attention values (V).



Tensorshape (1,9,128)Slice of the values:[[-0.4912, 1.2013, 0.3175, ..., 0.5066, -1.1901, -0.6701],[2.7700, -0.0197, 1.9663, ..., 0.4106, 1.1850, 0.1731],[0.0386, -0.0689, 1.2187, ..., 2.3305, 1.1628, -0.1199]]These are the projected values, obtained by applying a linear transformation to the input sequence, which will be combined with attention weights to produce context-aware representations.

Snapshot 10: Attention score matrix before softmax:

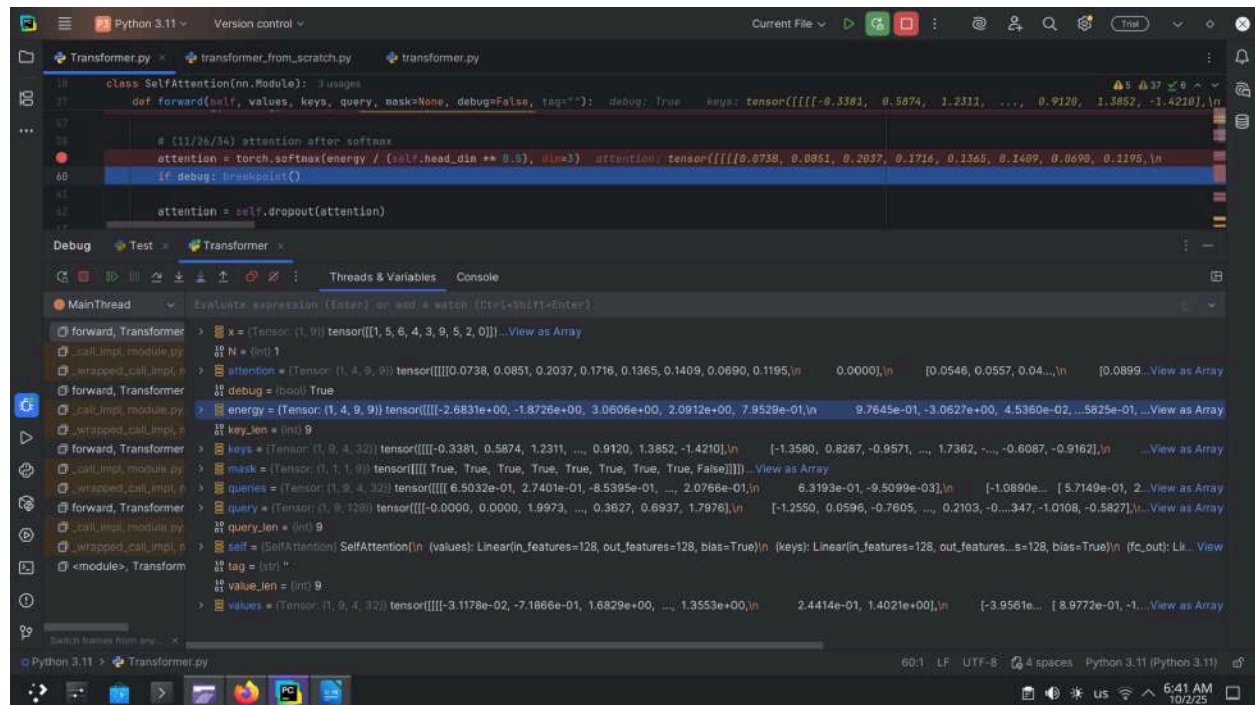


Attention scores (energy) $\rightarrow (1, 4, 9, 9)$

Slice example: `[[[[5.3990, -8.9257, -0.4634, ... , -2.7132], [...]]]]`

This tensor represents raw dot-product similarity between each query and key for all 9 tokens across 4 attention heads.

Snapshot11: Attention score matrix after softmax.



Attention after softmax : (1, 4, 9, 9)

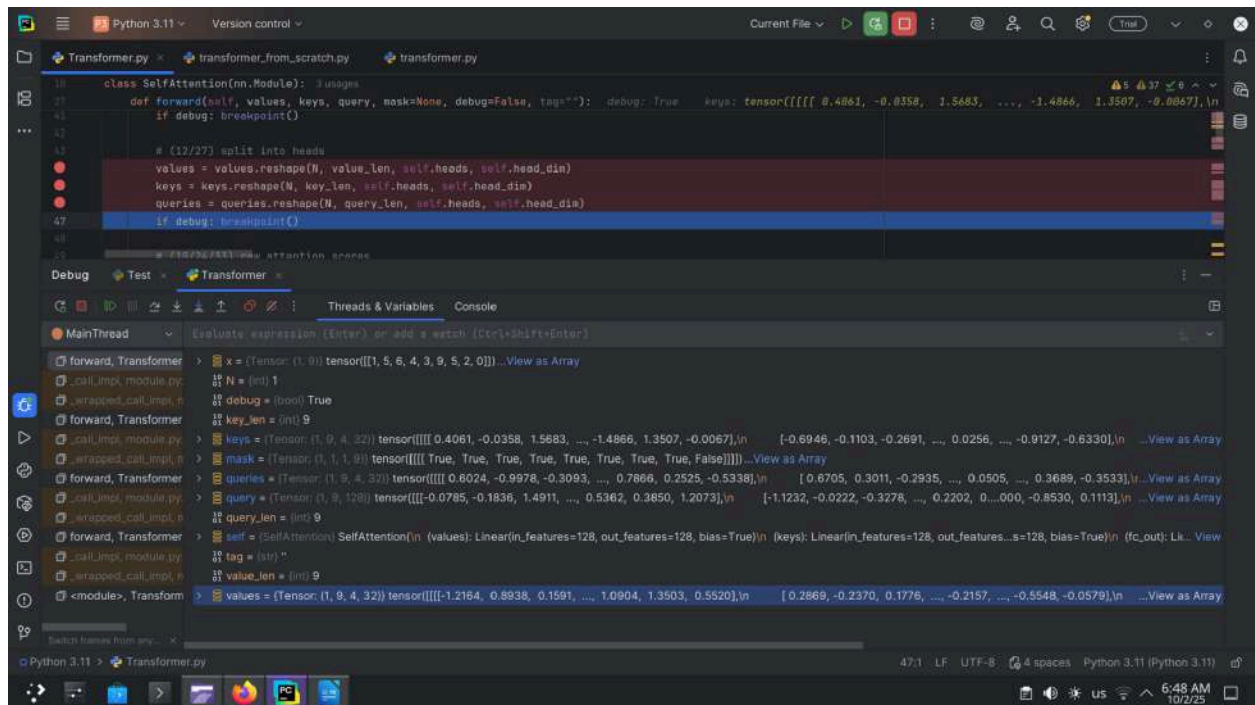
```
[[[0.4200, 0.0334, 0.1490, 0.1093, 0.0270, 0.0641, 0.0155, 0.1818,
0.0000],
```

```
[0.1005, 0.1898, 0.0370, ... , 0.0000]
```

```
[0.0776, 0.1671, 0.2629, 0.0681, 0.0811, 0.1180, 0.1636, 0.0616,
0.0000]]]
```

This is the **normalized attention distribution**, where each row sums to 1 across keys, showing how much each token attends to others (mask ensures padding = 0).

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



Tensor shape: (1, 9, 4, 32)

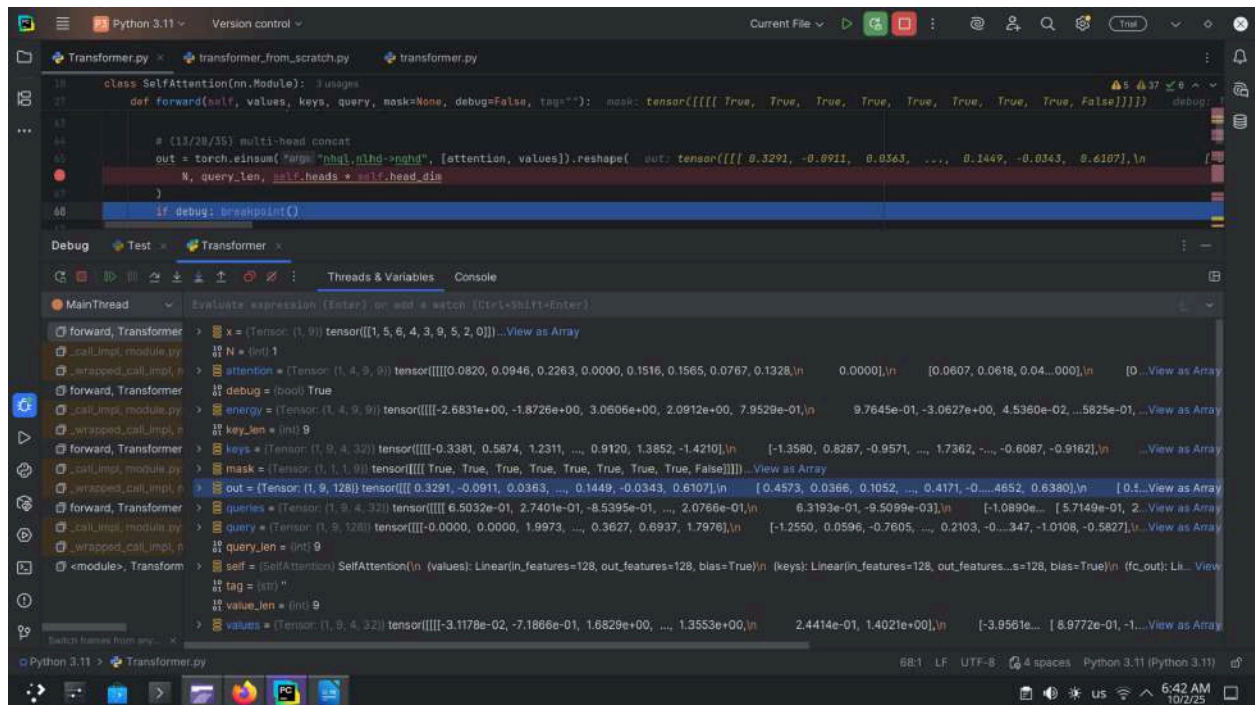
Slice of values:[[[-0.4912, 1.2013, 0.3175, ..., 0.6415, 0.4274, -1.2214],

[-0.2831, -0.6020, 0.7960, ..., -1.3493, -0.3679, 0.0341],

[0.0548, 0.7233, -1.0724, ..., 2.3305, 1.1628, -0.1199]]]

The tensors are reshaped to split the 128-dimensional embeddings into 4 attention heads of size 32, allowing each head to learn different attention patterns in parallel.

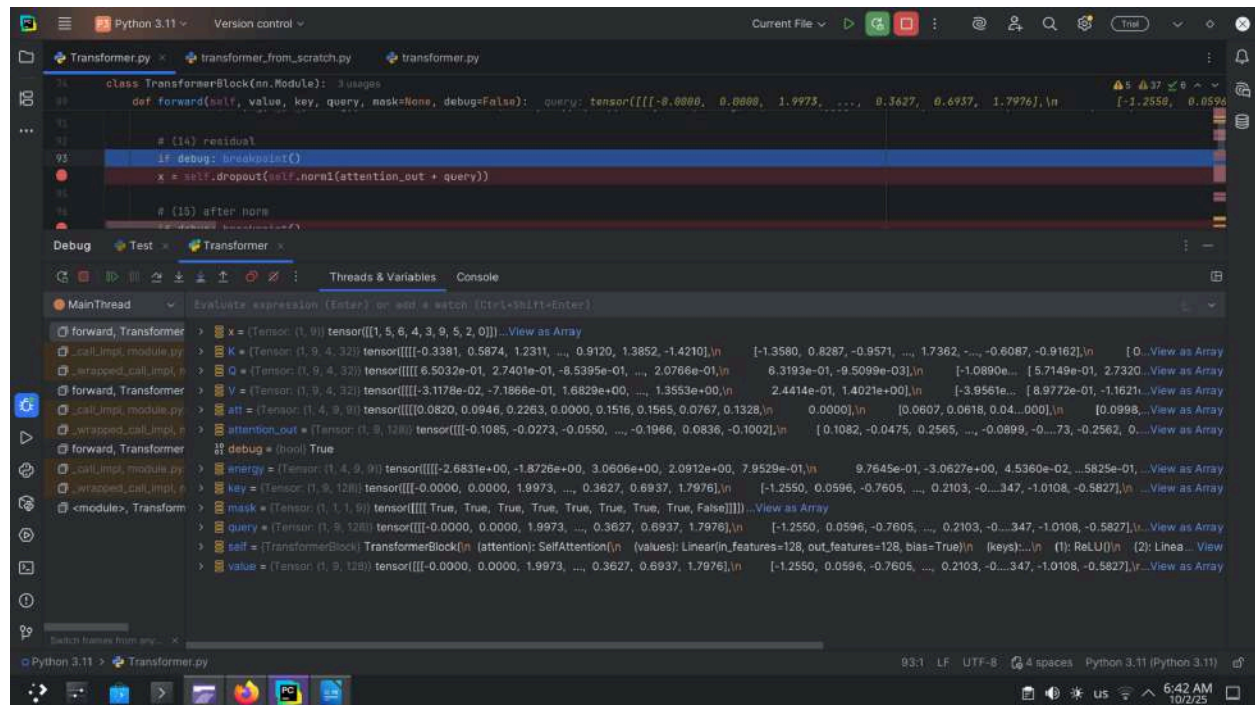
Snapshot 13: Multi-head attention output after concatenation.



Multi-head concat output → (1, 9, 128)

This is the **final combined context representation** where all 4 attention heads' weighted values (32 dims each) are concatenated back into the original hidden size of 128 for every token across the 9 sequence positions.

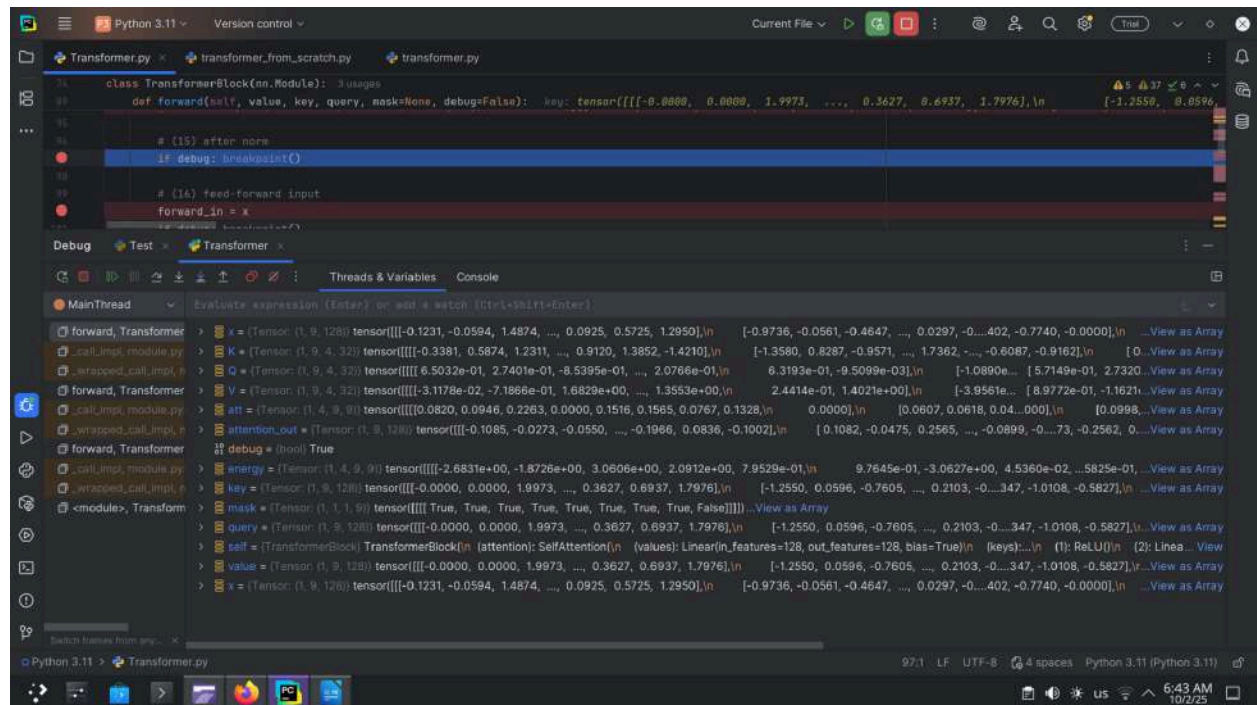
Snapshot 14: Residual connection tensors.



Tensorshape: (1, 9, 128)

the model adds the **original query embedding (1, 9, 128)** back to the **attention output (1, 9, 128)**, applies layer normalization and dropout, keeping the same hidden size per token (128) across the 9 tokens. This stabilizes training and preserves original information while integrating contextual attention.

Snapshot15: Layer normalization output:

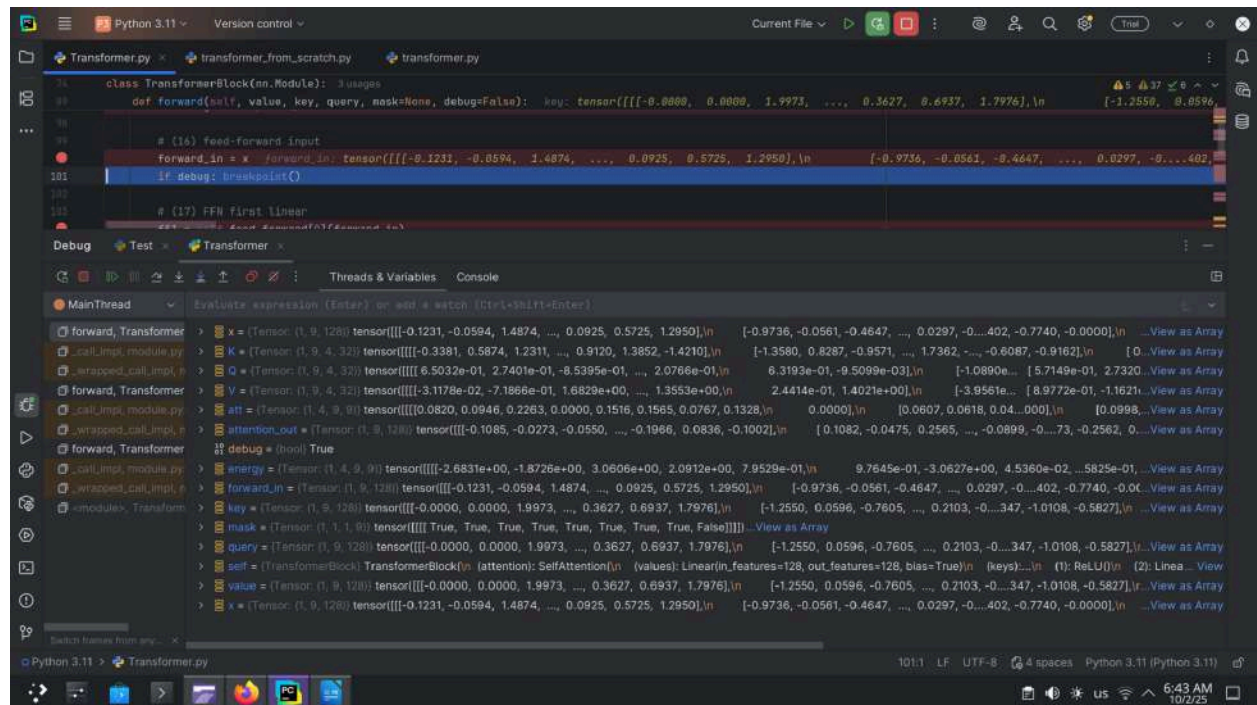


Tensor shape: (1, 9, 128)

Slice: tensor([[[[-1.9255, -0.6748, -2.6549, ..., 0.0373, 1.0216,
-0.0000], [0.5753, -0.6659, -0.3649, ..., 0.0921, 0....]]]])

This is the normalized output after the residual connection, keeping the same (`batch=1`, `seq_len=9`, `hidden_dim=128`) shape while ensuring stabilized feature distributions for each token.

Snapshot16: Feed-forward input



Tensor shape: (1, 9, 128)

Slice of values: tensor([[[[-1.9255, -0.6748, -2.6549, ..., 0.0373, 1.0216, -0.0000], [0.5753, -0.6659, -0.3649, ..., 0.0921, 0.8343, -0.1246]]]])

Explanation: This is the normalized token representation after the residual connection, with 9 tokens each mapped to a 128-dim hidden vector, ready for the feed-forward network.

Snapshot17: Feed-forward first linear layer output

The screenshot displays a Jupyter Notebook environment with a Python 3.11 interpreter. The code defines a `TransformerBlock` class with a `forward` method. The method takes `half`, `value`, `key`, `query`, and `mask` as inputs. It performs a feed-forward network (FFN) operation on the `value` tensor, which is a 1D tensor of size 128. The output of the FFN is then added to the original `value` tensor using a residual connection. The final output is a 1D tensor of size 128.

The execution output shows the internal state of the block, including the input, key, value, and mask tensors, and the output of the feed-forward network. The output is a 1D tensor of size 128, with values ranging from approximately -1.2550 to 1.2550.

Tensor shape: (1, 9, 512)

Slice of values: `tensor([[[[0.1018, -0.6590, -0.3004, ..., -0.6805, -0.8444, 0.5074], [0.4767, 0.8947, -0.4337, ..., -0.0662, 0.2190, 0.4403]]]])`

This is the output of the first feed-forward linear layer, where each of the 9 tokens is projected from 128 dimensions up to 512 before applying ReLU.

Snapshot18: Feed-forward second linear layer output.

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

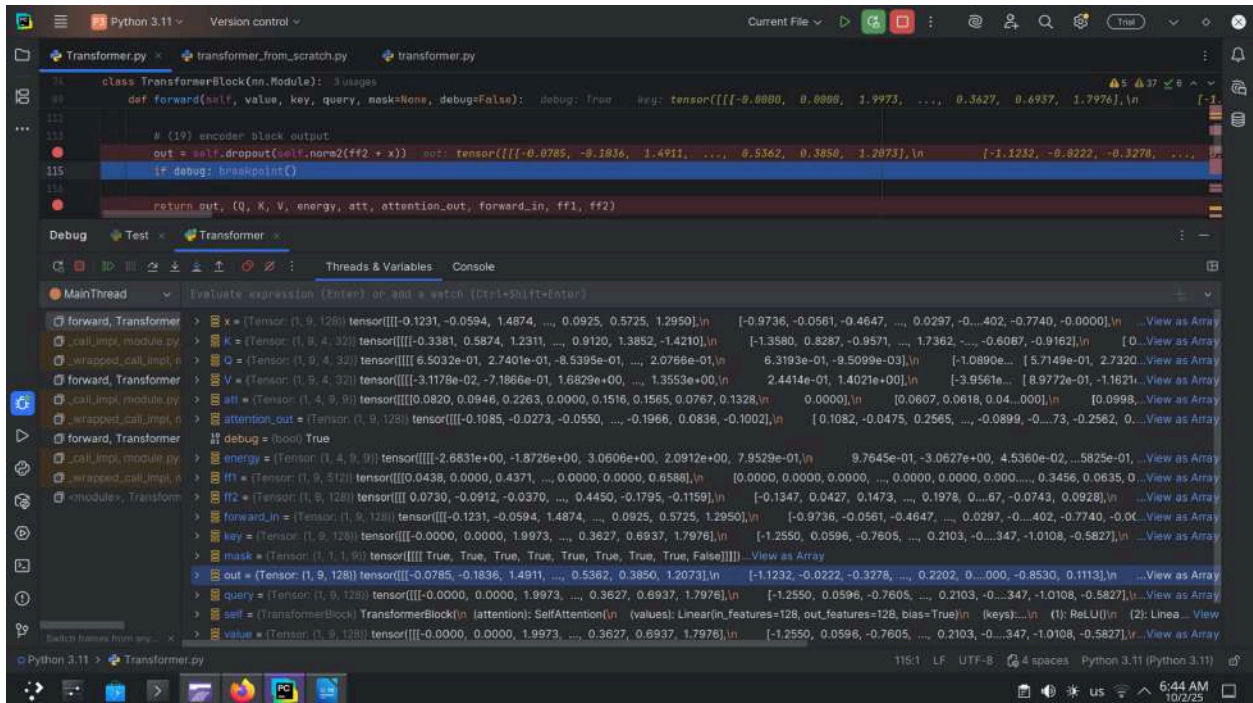
- Code Cell:** Contains the definition of the `TransformerBlock` class and its `forward` method. The `forward` method takes `half, value, key, query, mask=None, debug=False` as arguments. It includes a feed-forward network (FFN) and a multi-head attention mechanism. The output of the block is shown as a tensor with dimensions `[1, 9, 128]`.
- Output:** The output of the `forward` method is displayed as a tensor with dimensions `[1, 9, 128]`. The tensor values are shown in a grid format, with some values highlighted in red.
- Variable Explorer:** Located at the bottom of the notebook, it shows the variables defined in the code cell. The variables are listed in a table with their names and types.

Tensor shape: (1, 9, 128)

Slice of values: `tensor([[-0.1795, -0.0159, -0.3711, ..., -0.2412, 0.5005, 0.3594], [-0.0432, 0.0264, 0.0500, ..., 0.2240, 0.1852, 0.7200]])`

This is the output of the second feed-forward linear layer, projecting back from $512 \rightarrow 128$ dimensions to match the model size for residual connection.

Snapshot 19: Encoder block final output tensor.



```
class TransformerBlock(nn.Module):
    def forward(self, value, key, query, mask=None, debug=False):
        # (19) encoder block output
        out = self.dropout(self.norm2(ff2 + x))
        if debug: breakpoint()
        return out, (Q, K, V, energy, att, attention_out, forward_in, ff1, ff2)
```

Debug Console:

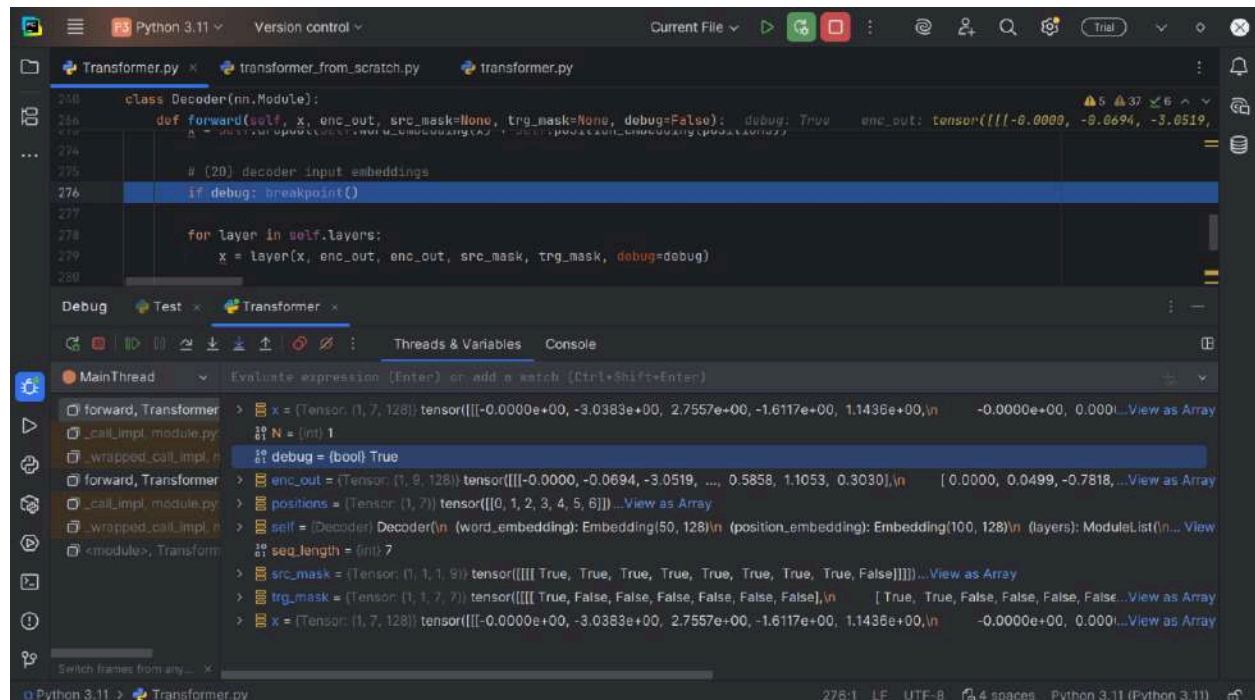
```
> x = (Tensor(1, 9, 128)) tensor([[-0.1231, -0.0594, 1.4874, ..., 0.0925, 0.5725, 1.2950], ..., [-0.9736, -0.0561, -0.4647, ..., 0.0297, -0.402, -0.7740, -0.0000], ..., [-1.3580, 0.8287, -0.9571, ..., 1.7362, ..., -0.6087, -0.9162]],) ... View as Array
> k = (Tensor(1, 9, 32)) tensor([[-0.3381, 0.5874, 1.2311, ..., 0.9120, 1.3852, -1.4210], ..., [-1.3580, 0.8287, -0.9571, ..., 1.7362, ..., -0.6087, -0.9162]],) ... View as Array
> q = (Tensor(1, 9, 32)) tensor([[-0.0000, 0.0000, 0.0000, ..., 0.0000, 0.0000, 0.0000], ..., [-0.1347, 0.0427, 0.1473, ..., 0.1978, 0.0067, -0.0743, 0.0928]],) ... View as Array
> ff1 = (Tensor(1, 9, 128)) tensor([[-0.1231, -0.0594, 1.4874, ..., 0.0925, 0.5725, 1.2950], ..., [-0.9736, -0.0561, -0.4647, ..., 0.0297, -0.402, -0.7740, -0.0000]],) ... View as Array
> ff2 = (Tensor(1, 9, 128)) tensor([[-0.0000, 0.0000, 1.9973, ..., 0.3627, 0.6937, 1.7976], ..., [-1.2550, 0.0596, -0.7605, ..., 0.2103, -0.347, -1.0108, -0.5827]],) ... View as Array
> mask = (Tensor(1, 1, 1, 9)) tensor([[[[ True, True, True, True, True, True, True, True, False]]]]) ... View as Array
> out = (Tensor(1, 9, 128)) tensor([[-0.0785, -0.1836, 1.4911, ..., 0.5362, 0.3850, 1.2073], ..., [-1.1232, -0.0222, -0.3278, ..., 0.2202, 0.0000, -0.8530, 0.1113]],) ... View as Array
> query = (Tensor(1, 9, 128)) tensor([[-0.0000, 0.0000, 1.9973, ..., 0.3627, 0.6937, 1.7976], ..., [-1.2550, 0.0596, -0.7605, ..., 0.2103, -0.347, -1.0108, -0.5827]],) ... View as Array
> self = (TransformerBlock) TransformerBlock(in_attention: SelfAttention(in_features=128, out_features=128, bias=True) (keys):... (l): ReLU(1) (2): Linear... View
> value = (Tensor(1, 9, 128)) tensor([[-0.0000, 0.0000, 1.9973, ..., 0.3627, 0.6937, 1.7976], ..., [-1.2550, 0.0596, -0.7605, ..., 0.2103, -0.347, -1.0108, -0.5827]],) ... View as Array
```

Tensor shape: (1, 9, 128)

Slice of values: tensor([[[[-2.1399, -0.6979, -3.0788, ..., -0.0000, 1.5579, 0.3727], [0.6075, -0.0000, -0.2510, ..., 0.0000, 0.9430, 0.4812]]]])

Explanation: This is the **encoder block output** after applying feed-forward + residual + layer normalization, ready to be passed to the next encoder layer or decoder.

Snapshot20: Decoder block input tensor.

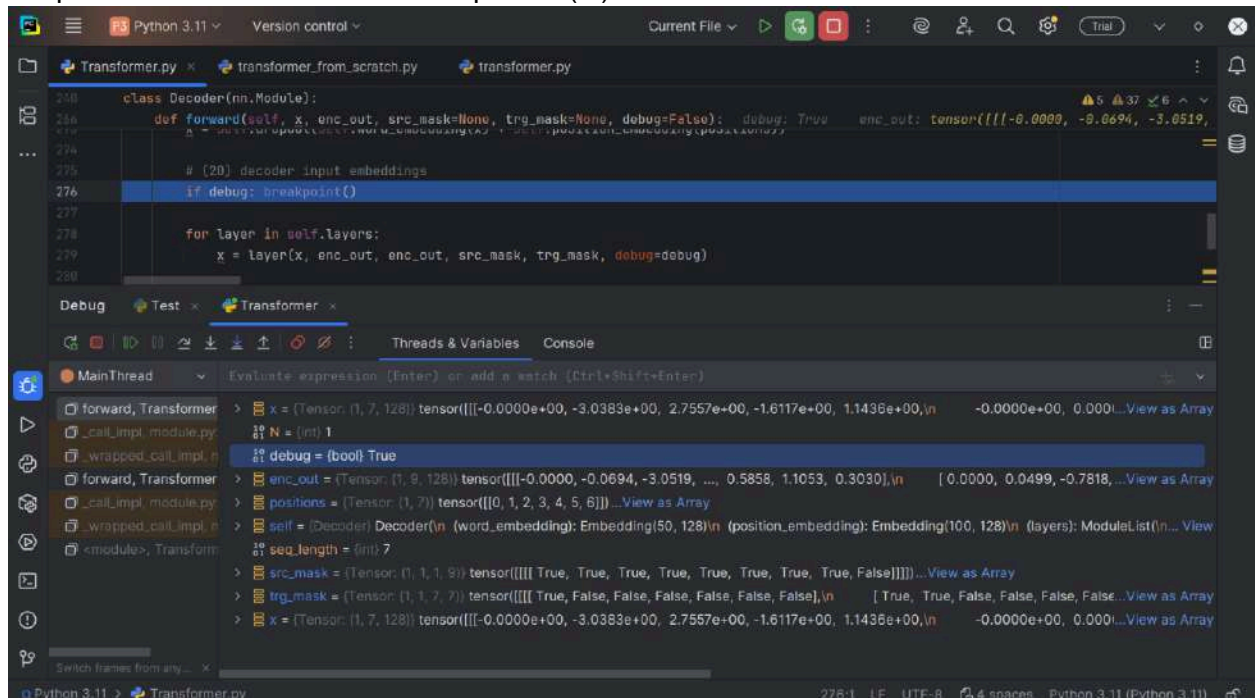


Tensor shape: `(1, 7, 128)`

Slice of values: `tensor([[[-0.0000, -3.0383, 2.7557, -1.6117, 1.1436],`
`[0.8359, -0.2832, 0.0587, -0.8705, -1.2216], ...]])`

This tensor represents the decoder input embeddings (token + positional), giving each of the 7 target tokens a 128-dimensional embedding vector.

Snapshot21: Masked self-attention queries (Q).

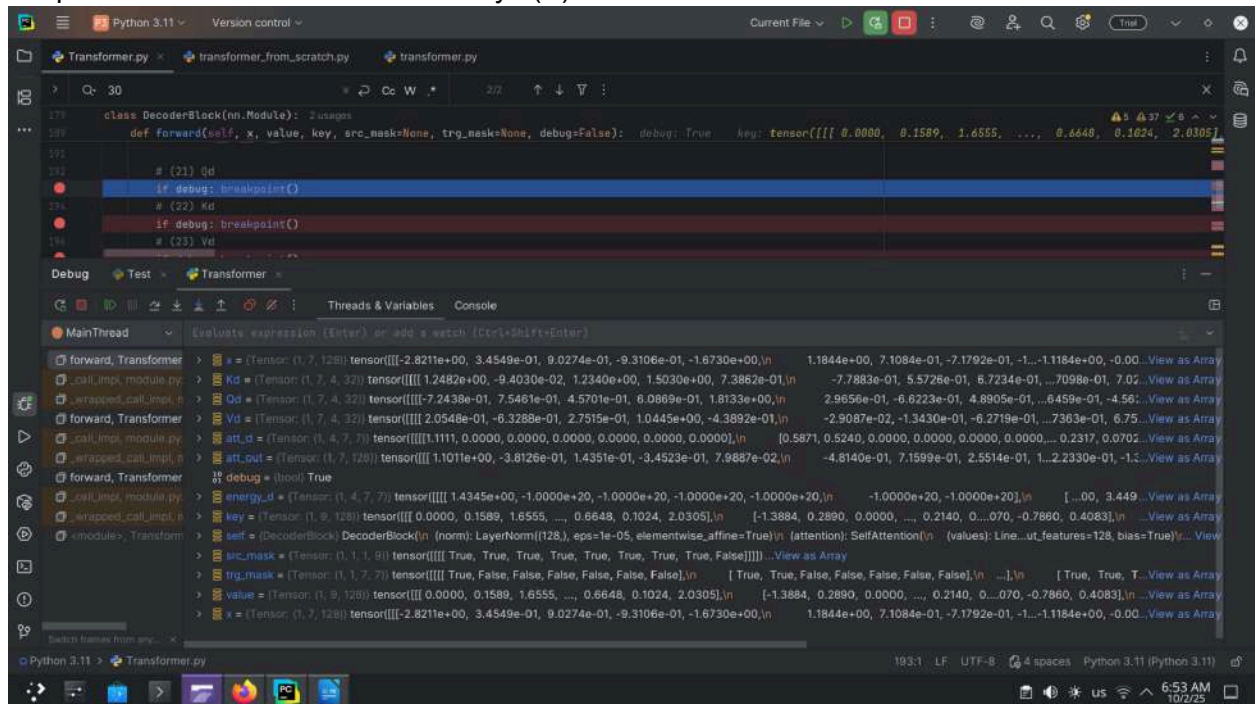


Tensor shape: (1, 7, 128)

Slice of values: `tensor([[-0.0000, -3.0383, 2.7557, -1.6117, 1.1436],
[0.8359, -0.2832, 0.0587, -0.8705, -1.2216], ...])`

This tensor represents the decoder input embeddings (token + positional), giving each of the 7 target tokens a 128-dimensional embedding vector.

Snapshot22: Masked self-attention keys (K):

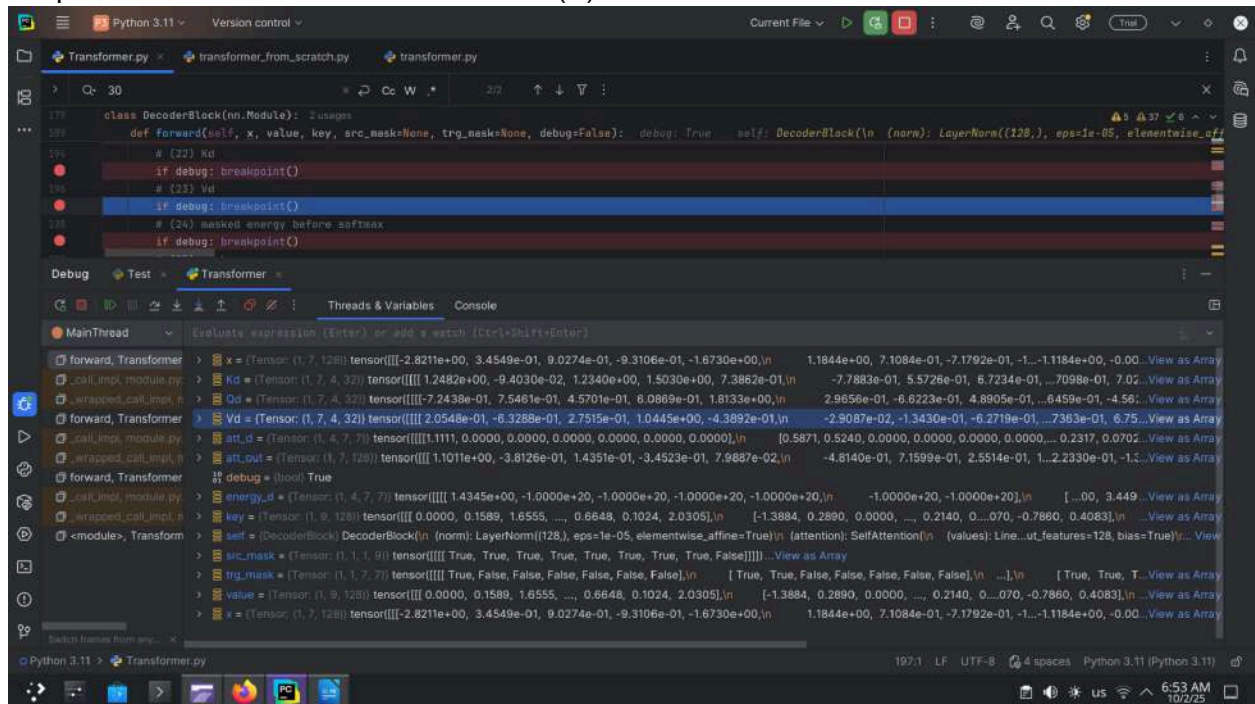


Tensor shape: (1, 7, 4, 32)

Slice of values: `[[[1.4765, -0.8994, 0.1106, -0.4785, ...], ...]]`

This is the **key tensor for decoder self-attention**, where the 7 target tokens are projected into 4 attention heads, each with 32 features.

Snapshot23: Masked self-attention values (V).

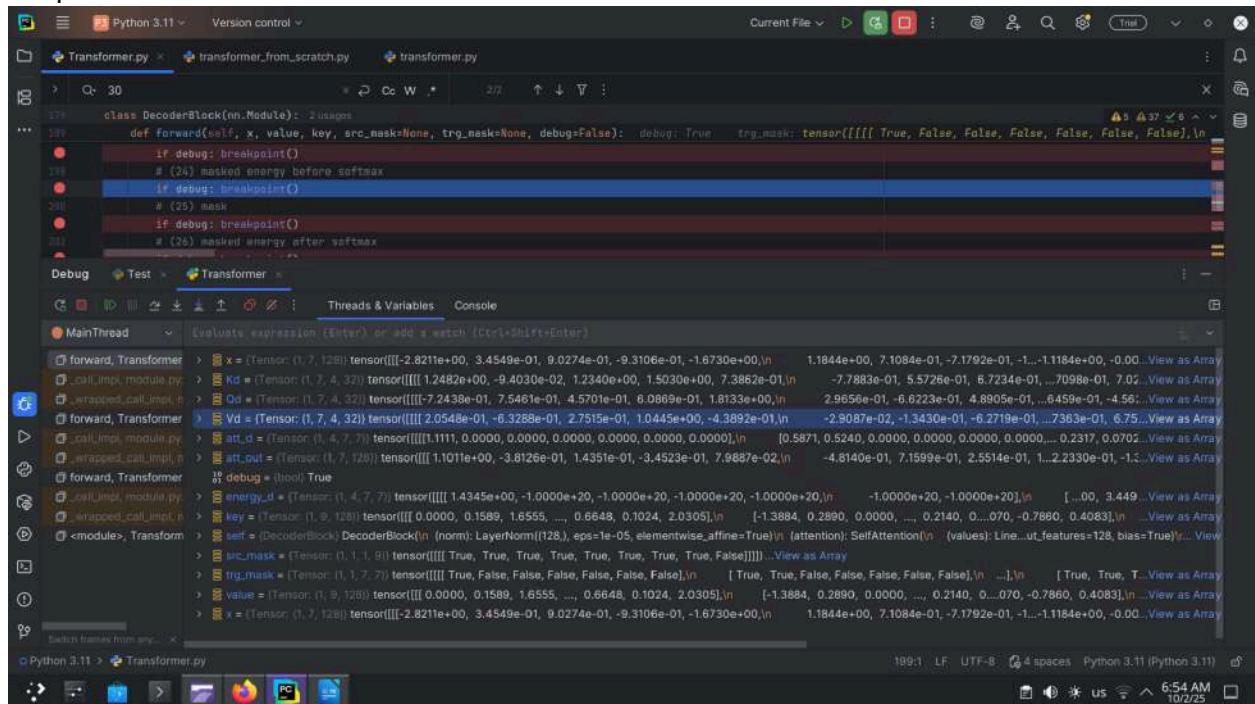


Tensor shape: (1, 7, 4, 32)

Slice of values: `[[[[1.1681, -0.4677, 0.0840, 0.3390, -0.6092, 1.7005, -1.4352, 0.2627, ...]]]]`

each of the 9 tokens is projected into 4 heads with 32 features, carrying the contextual information to be aggregated by the attention mechanism.

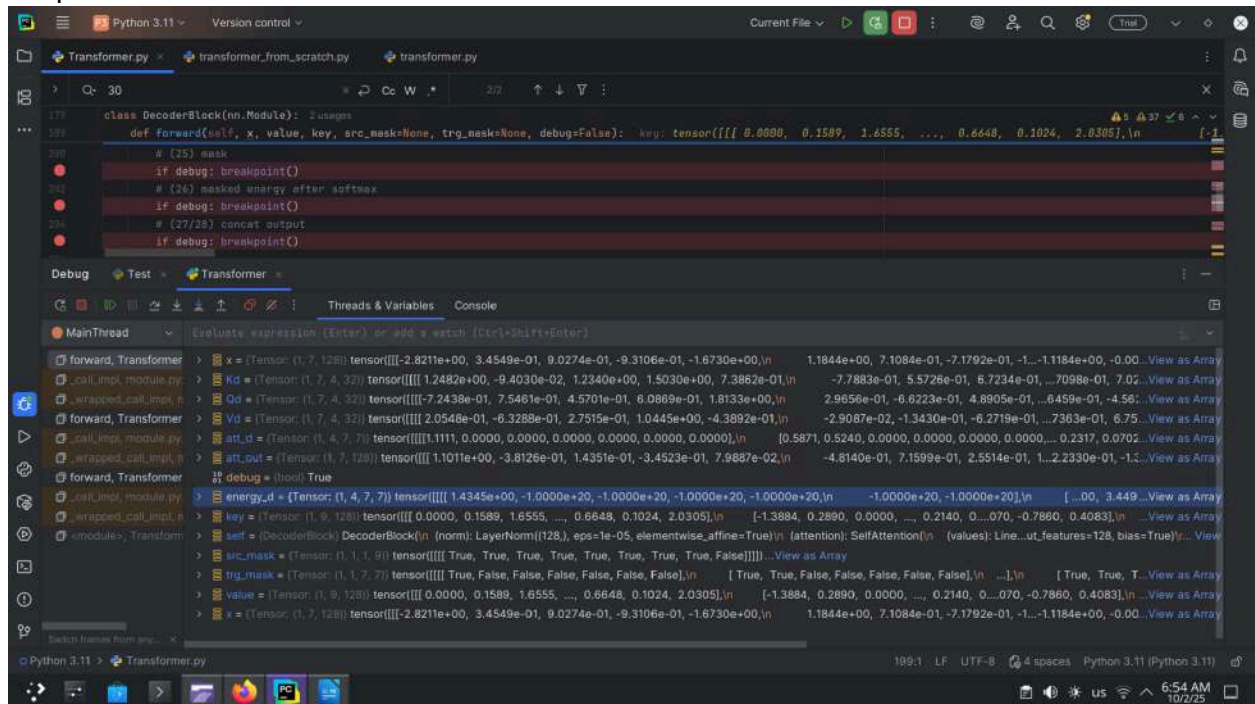
Snapshot24: Masked attention scores before mask.



Tensor shape: (1, 4, 7, 7)

Slice of values: `[[[3.0325, -1.0000e+20, -1.0000e+20, -1.0000e+20, -1.0000e+20, -1.0000e+20, -1.0000e+20], [0.2433, -0.2223, -3.2639, 1.2004, 2.8968, 7.8033, 4.6633], ...]]` This is the **scaled dot-product attention score matrix with mask applied**, where `-1e20` enforces causal masking to prevent attention to future tokens.

Snapshot25 Mask tensor.

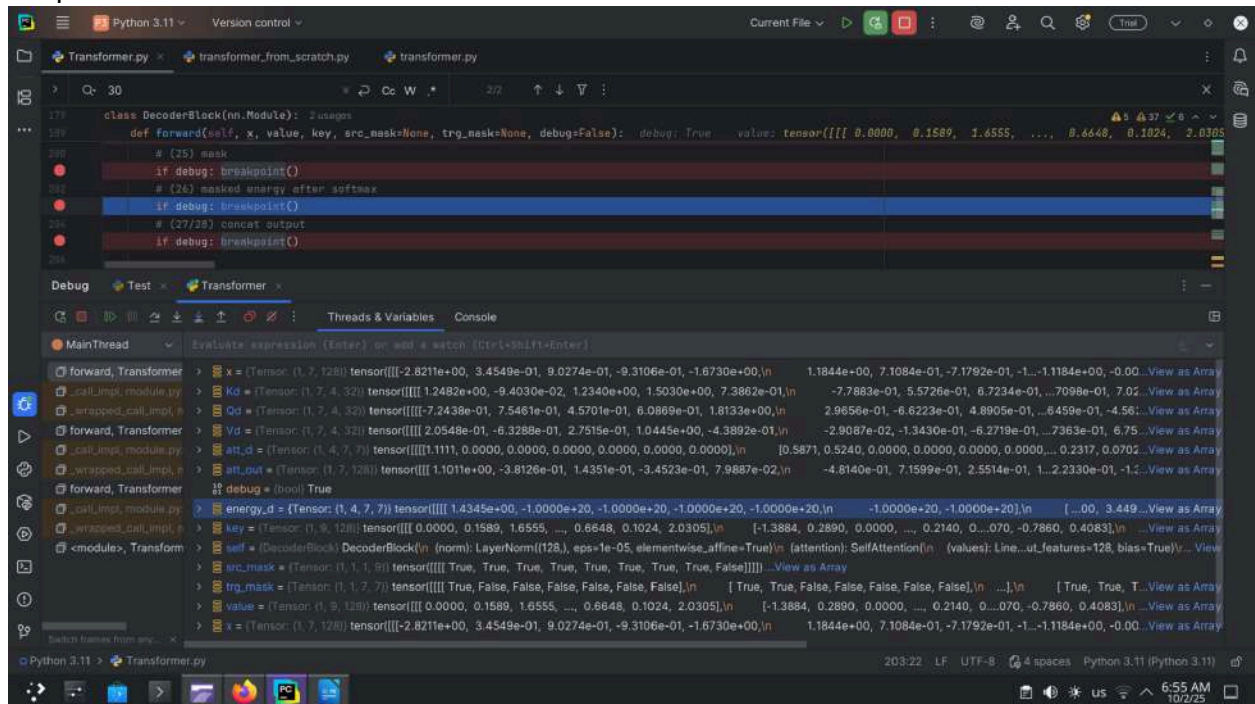


Tensor shape: (1, 1, 7, 7)

Slice of values: `[[[True, False, False, False, False, False, False], [True, True, False, False, False, False, False], ...]]]`

causal mask that prevents each decoder position from attending to future tokens (upper-triangular False values).

Snapshot26: Masked attention scores after mask + softmax.

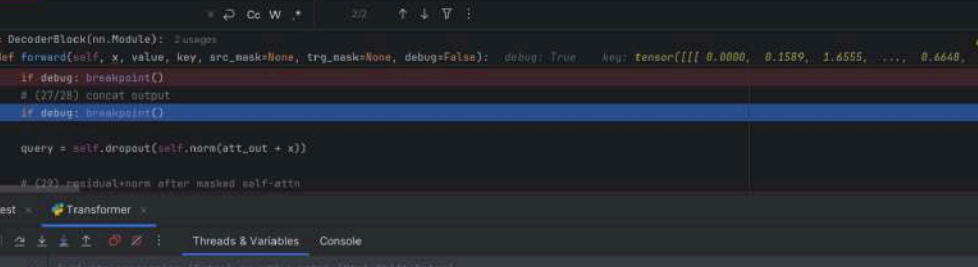


Tensor shape: (1, 4, 7, 7)

Slice of values: `[[[1.1111, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000], [0.2433, 0.8678, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000], ...]]`

attention probability matrix after applying the mask and softmax; rows sum to 1 over valid tokens, zeros remain where masked.

Snapshot27: Masked self-attention multi-head split.



The screenshot shows a Jupyter Notebook with a Python 3.11 environment. The code defines a `DecoderBlock` class and calls its `forward` method. The output is displayed in a table format, showing the internal state of the model, including the key, value, and output tensors, and the attention weights.

The code defines a `DecoderBlock` class with the following methods:

- `forward(self, x, value, key, src_mask=None, trg_mask=None, debug=False)`: The main forward method that takes an input `x` and a value `value`, and returns the output `output`.
- `__init__(self, nn.Module)`: The constructor that initializes the `DecoderBlock` with a `nn.Module` object.

The output of the `forward` method is displayed in a table format, showing the internal state of the model, including the key, value, and output tensors, and the attention weights.

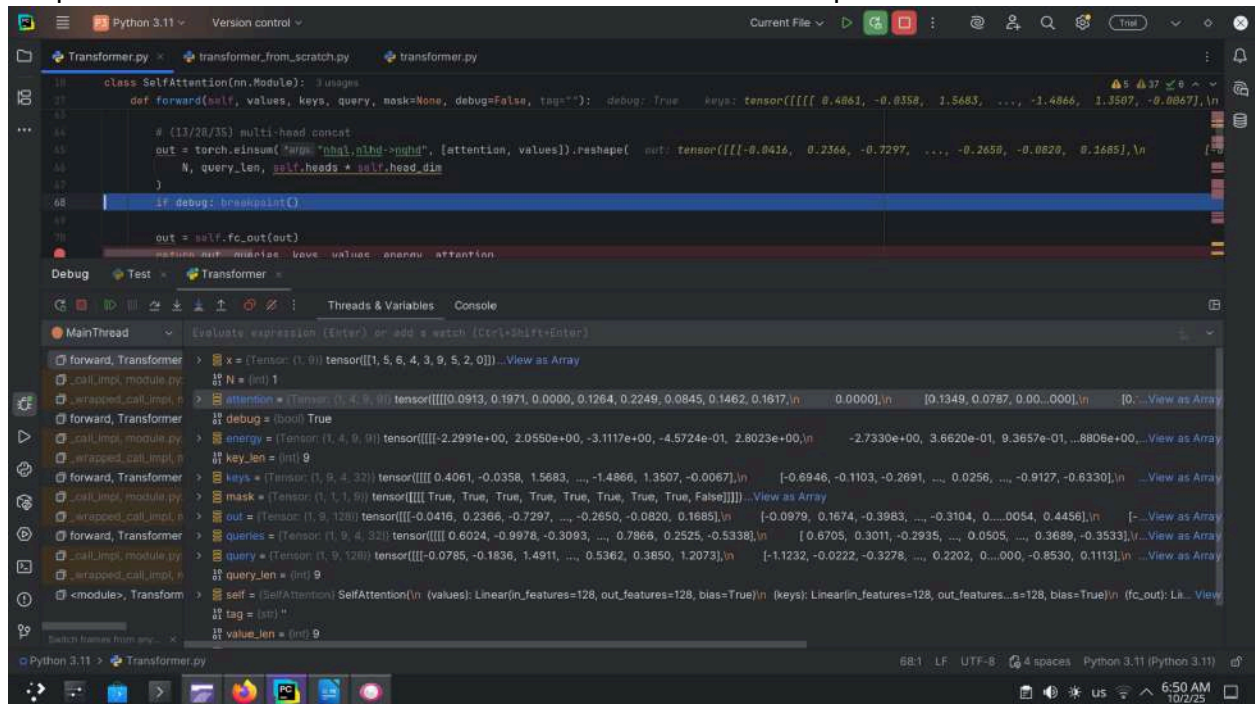
Variable	Value
<code>x</code>	<code>(Tensor: (1, 7, 128)) tensor([[-2.8211e+00, 3.4549e-01, 9.0274e-01, -9.3106e-01, -1.6730e+00, ..., 1.8444e+00, 7.1084e-01, -7.1792e-01, -1.1184e+00, -0.00...]])</code>
<code>key</code>	<code>(Tensor: (1, 7, 128)) tensor([[-2.8211e+00, 3.4549e-01, 9.0274e-01, -9.3106e-01, -1.6730e+00, ..., 1.8444e+00, 7.1084e-01, -7.1792e-01, -1.1184e+00, -0.00...]])</code>
<code>value</code>	<code>(Tensor: (1, 7, 128)) tensor([[-2.8211e+00, 3.4549e-01, 9.0274e-01, -9.3106e-01, -1.6730e+00, ..., 1.8444e+00, 7.1084e-01, -7.1792e-01, -1.1184e+00, -0.00...]])</code>
<code>output</code>	<code>(Tensor: (1, 7, 128)) tensor([[-2.8211e+00, 3.4549e-01, 9.0274e-01, -9.3106e-01, -1.6730e+00, ..., 1.8444e+00, 7.1084e-01, -7.1792e-01, -1.1184e+00, -0.00...]])</code>

Tensor shape: (1, 7, 128)

Slice of values: `[[[0.2468, 0.4039, -0.9672, 0.6223, -0.4919, -0.0373, 0.3309, -0.1236, ...]]]`

final attended representation, obtained by weighting `Vd` with `att_d`, concatenating all heads, and projecting back to the model dimension.

Snapshot28: Masked self-attention multi-head concatenated output.



Tensor shape: (1, 7, 128)

Slice of values:

```
[[[ 0.2468, 0.4039, -0.9672, 0.6223, -0.4919, -0.0373, 0.3309,
-0.1236, ...]]]
```

decoder masked self-attention output after all 4 heads have produced their weighted sums of values (Vd), concatenated together, and projected back into the 128-dim model space.

Snapshot29: Residual + normalization after masked self-attention.

```
class DecoderBlock(nn.Module):
    def forward(self, x, value, key, src_mask=None, trg_mask=None, debug=False):
        # (20) residual+norm after masked self-atten
        if debug: breakpoint()
        out, dbg = self.transformer_block(value, key, query, src_mask, debug=debug)
        # (21) cross-atten queries
```

Debug Console:

- `forward, Transformer`: `x = (Tensor: (1, 7, 128)) tensor([[-2.8211e+00, 3.4549e-01, 9.0274e-01, -9.3106e-01, -1.6730e+00, ..., 1.1844e+00, 7.1084e-01, -7.1792e-01, -1.1184e+00, -0.00... View as Array`
- `forward, Transformer`: `key = (Tensor: (1, 9, 128)) tensor([[[0.0000, 0.1589, 1.6555, ..., 0.6648, 0.1024, 2.0305], ..., [-1.3884, 0.2890, 0.0000, ..., 0.2140, 0.070, -0.7860, 0.4083], ..., [-1.2000e+00, 1.2553e-02, 7.9158e-01, -8.8058e-01, -1.1087e+00, ..., 5.4448e-01, 1.0656e+00, -2.9488e-01, 7.1... View as Array`
- `forward, Transformer`: `energy_d = (Tensor: (1, 4, 7, 7)) tensor([[[[1.4345e+00, -1.0000e+20, -1.0000e+20, -1.0000e+20, -1.0000e+20, ..., -1.0000e+20, -1.0000e+20], ..., [..., 3.449... View as Array`

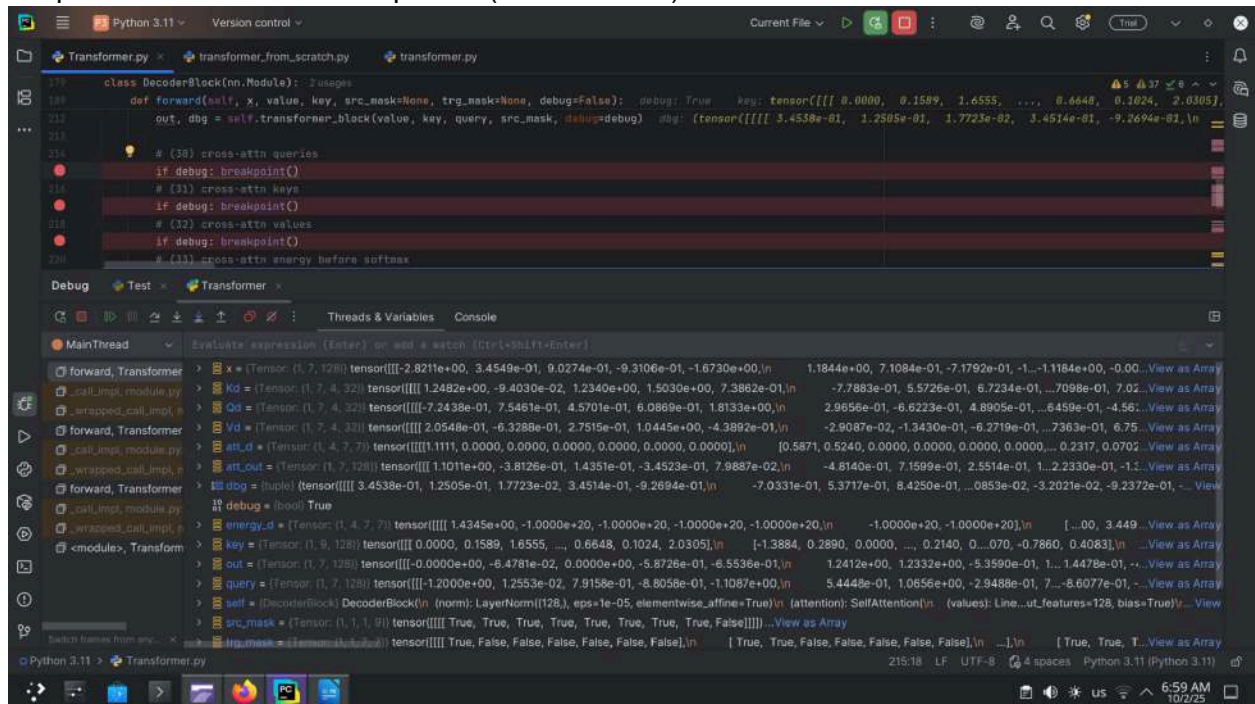
Tensor shape: (1, 7, 128)

Slice of values:

```
[[[-0.0000, -3.0383, 2.7557, -1.6117, 1.1436, 0.0000, 0.0000, 1.1074, ...]]]
```

decoder hidden state after adding the residual connection (`x + att_out`) and applying layer normalization, ensuring stable gradients and preserving sequence information.

Snapshot30: Cross-attention queries (from decoder).



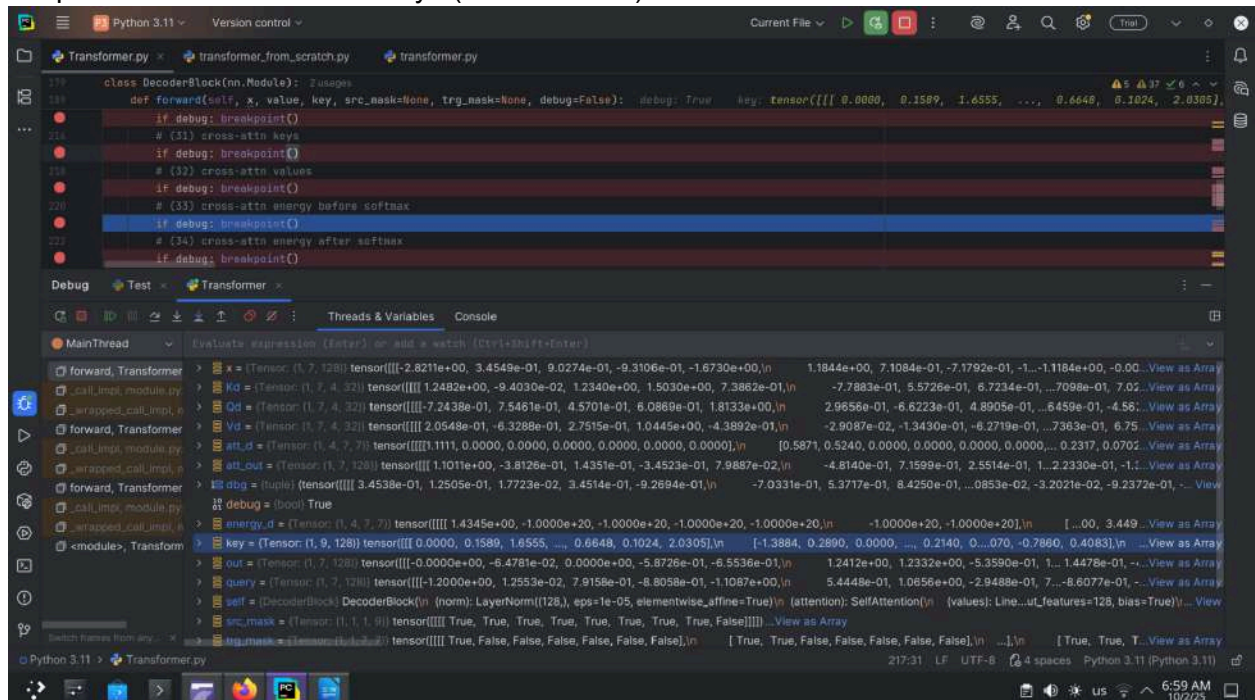
The screenshot shows a Python IDE with a file named `transformer.py` open. The code defines a `DecoderBlock` class with a `forward` method. Several debug breakpoints are set in the `forward` method, specifically around the cross-attention queries, keys, values, and energy calculations. The `Debug` window is open, showing the state of the program at a breakpoint. The `Threads & Variables` pane shows the `MainThread` with various variables inspected, including `k`, `kd`, `qd`, `vd`, `att_d`, `energy_d`, `key`, `out`, `query`, `self`, `src_mask`, and `tgt_mask`. The `query` variable is highlighted, showing its shape as `(1, 7, 128)` and its values as `[[[0.1580, -0.0000, 1.3742, -0.8172, 0.4774, ...]]]`.

Tensor shape: (1, 7, 128)

Slice of values: `[[[0.1580, -0.0000, 1.3742, -0.8172, 0.4774, ...]]]`

projected queries from the decoder sequence, used to attend over encoder outputs in cross-attention.

Snapshot31: Cross-attention keys (from encoder).

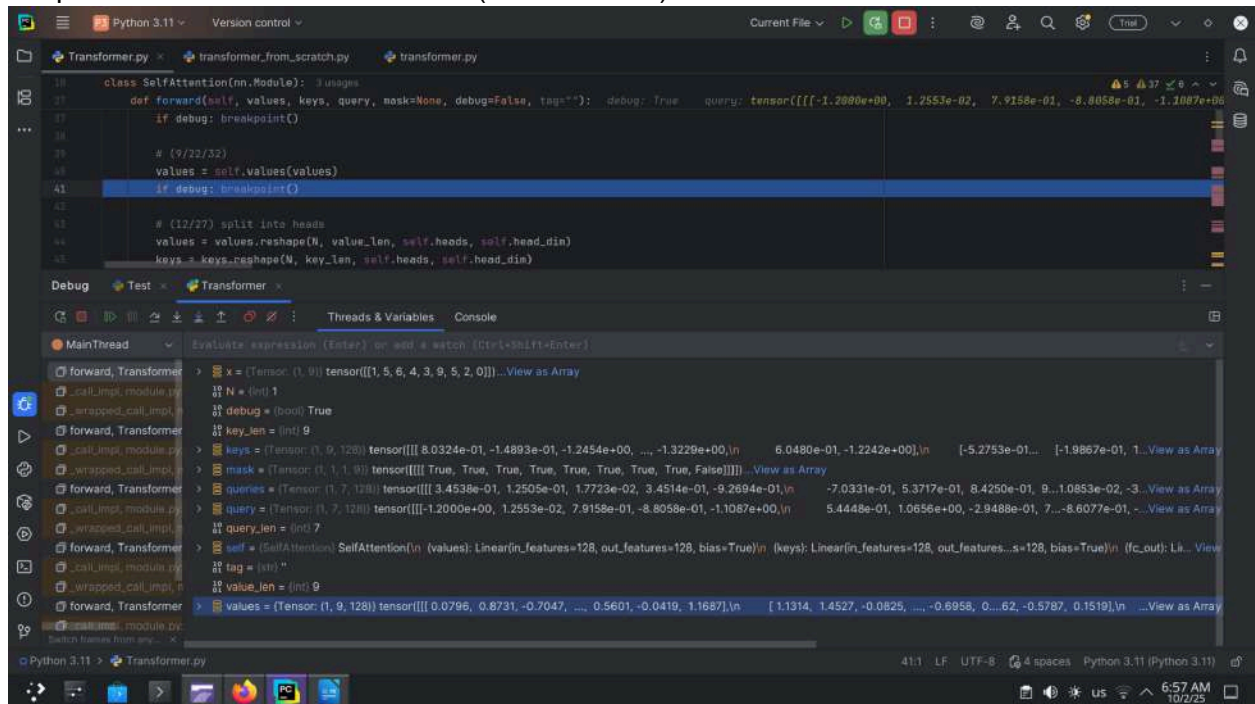


Tensor shape: (1, 9, 128)

Slice of values: `[[[-0.0000, -0.0694, -3.0519, ..., 0.5858, 1.1053]]]`

projected keys from the encoder output, representing source-side context.

Snapshot32: Cross-attention values (from encoder).

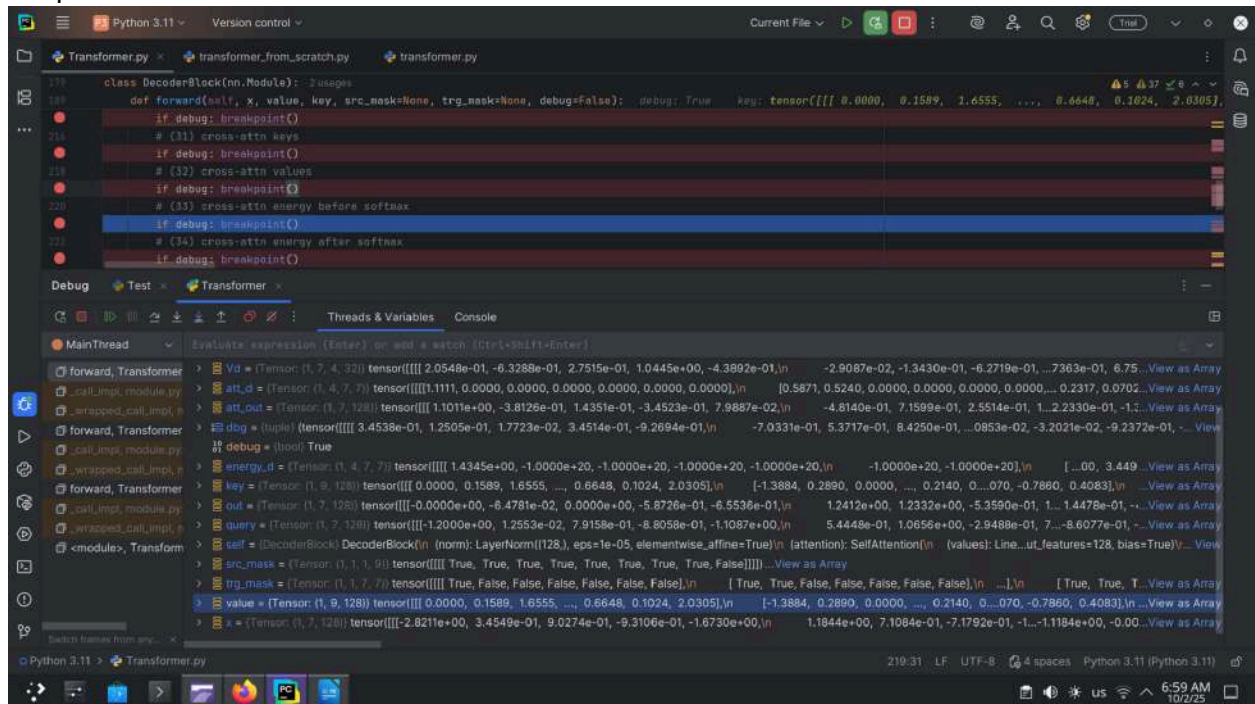


Tensor shape: (1, 9, 128)

Slice of values: `[[[-0.0000, -0.0694, -3.0519, ..., 0.3030, 0.3145]]]`

projected values from encoder outputs, providing the actual source information to aggregate via attention.

Snapshot33: Cross-attention score matrix before softmax.

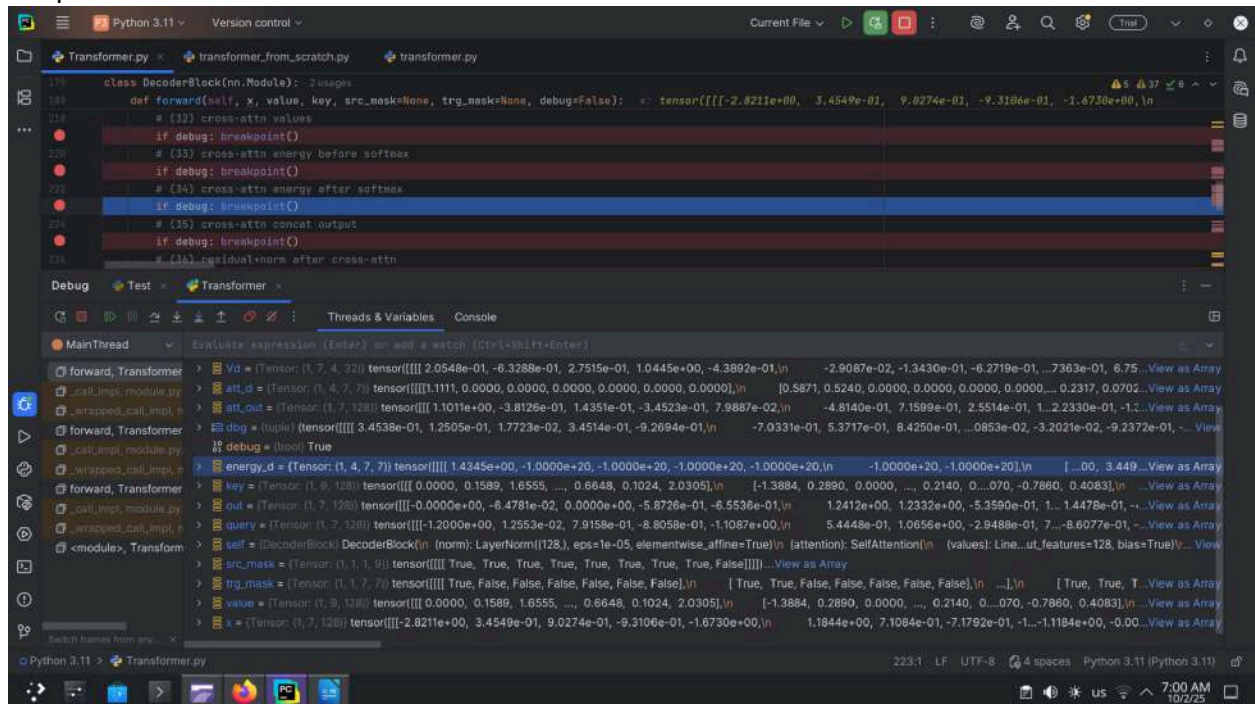


Tensor shape: (1, 4, 7, 9)

Slice of values: `[[[4.7009e-01, 1.8903e-01, -3.4222e+00, ..., 3.0993e+00, -1.0000e+20]]]`

Raw attention scores computed as dot-product of queries and keys before applying masking and softmax.

Snapshot34: Cross-attention score matrix after softmax.



```
class DecoderBlock(nn.Module):
    def forward(self, x, value, key, src_mask=None, trg_mask=None, debug=False):
        # (32) cross-attn values
        if debug: breakpoint()
        # (33) cross-attn energy before softmax
        if debug: breakpoint()
        # (34) cross-attn energy after softmax
        if debug: breakpoint()
        # (35) cross-attn concat output
        if debug: breakpoint()
        # (36) residual+norm after cross-attn
```

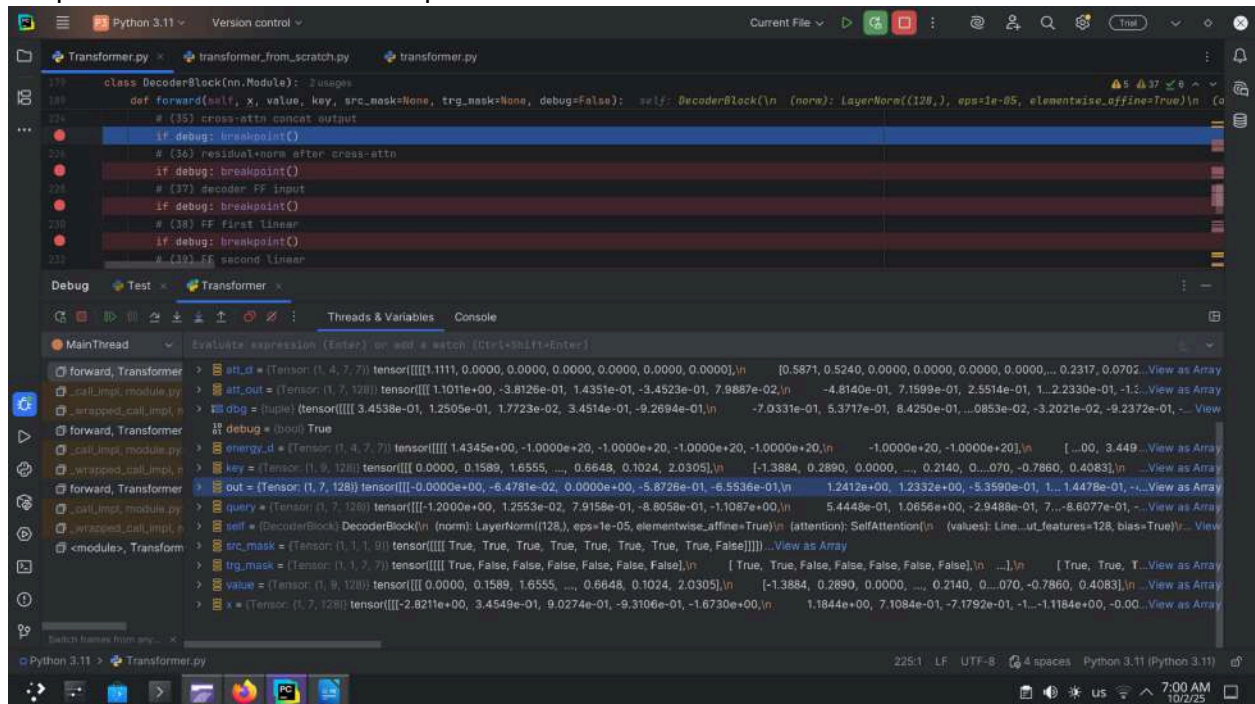
Debug Console:

```
forward, Transformer: Vd = (Tensor: (1, 7, 4, 32)) tensor([[[[ 2.0548e-01, -6.3288e-01, 2.7515e-01, 1.0445e+00, -4.3892e-01, ..., -2.9087e-02, -1.3430e-01, -6.2719e-01, ..., 7.363e-01, 6.75...
att_d = (Tensor: (1, 4, 7, 7)) tensor([[[[ 1.1111, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000], ..., [ 0.5871, 0.5240, 0.0000, 0.0000, 0.0000, 0.0000, ..., 0.2317, 0.0702...
att_out = (Tensor: (1, 7, 128)) tensor([[[[ 1.1011e+00, -3.8126e-01, 1.4351e-01, -3.4523e-01, 7.9887e-02, ..., -4.8140e-01, 7.1599e-01, 2.5514e-01, ..., 1.22330e-01, -1.1...
dog = (tuple) (tensor([[[[ 3.4538e-01, 1.2505e-01, 1.7723e-02, 3.4514e-01, -9.2694e-01, ..., -7.0331e-01, 5.3717e-01, 8.4250e-01, ..., 0.0853e-02, -3.2021e-02, -9.2372e-01, ...
energy_d = (Tensor: (1, 4, 7, 7)) tensor([[[[ 1.4345e+00, -1.0000e+20, -1.0000e+20, -1.0000e+20, -1.0000e+20, ..., -1.0000e+20, -1.0000e+20], ..., [ ..., 3.449...
key = (Tensor: (1, 9, 128)) tensor([[[[ 0.0000, 0.1589, 1.6555, ..., 0.6648, 0.1024, 2.0305], ..., [-1.3884, 0.2890, 0.0000, ..., 0.2140, 0.0070, -0.7860, 0.4083], ...,
out = (Tensor: (1, 7, 128)) tensor([[[[ -0.0000e+00, -6.4781e-02, 0.0000e+00, -5.8726e-01, -6.5536e-01, ..., 1.2412e+00, 1.2332e+00, -5.3590e-01, ..., 1.4478e-01, ...
query = (Tensor: (1, 7, 128)) tensor([[[[ -1.2000e+00, 1.2553e-02, 7.9158e-01, -8.8058e-01, -1.1087e+00, ..., 5.4448e-01, 1.0656e+00, -2.9488e-01, ..., 7.1...-8.6077e-01, ...
self = [DecoderBlock] DecoderBlock(in (norm): LayerNorm(128), eps=1e-05, elementwise_affine=True)(attention: SelfAttention(in (values): Linear(...), features=128, bias=True))...
src_mask = (Tensor: (1, 1, 1, 9)) tensor([[[[ True, True, True, True, True, True, True, True, True, False]]], ..., [ True, True, True, True, True, True, True, True, False]]], ...
trg_mask = (Tensor: (1, 1, 1, 7)) tensor([[[[ True, False, False, False, False, False, False], ..., [ True, True, False, False, False, False, False], ..., [ True, True, T...
value = (Tensor: (1, 9, 128)) tensor([[[[ 0.0000, 0.1589, 1.6555, ..., 0.6648, 0.1024, 2.0305], ..., [-1.3884, 0.2890, 0.0000, ..., 0.2140, 0.0070, -0.7860, 0.4083], ...,
v = (Tensor: (1, 7, 128)) tensor([[[[ -2.8211e+00, 3.4549e-01, 9.0274e-01, -9.3106e-01, -1.6730e+00, ..., 1.1844e+00, 7.1084e-01, -7.1792e-01, ..., -1.1184e+00, -0.00...]
```

Tensor shape: (1, 4, 7, 9) Slice of values: [[[[0.2955, -0.8131, -0.1729, ..., -0.2260, 0.2122]]]]

Normalized attention weights after softmax, showing how much each decoder token attends to encoder tokens.

Snapshot35: Cross-attention output after concatenation.

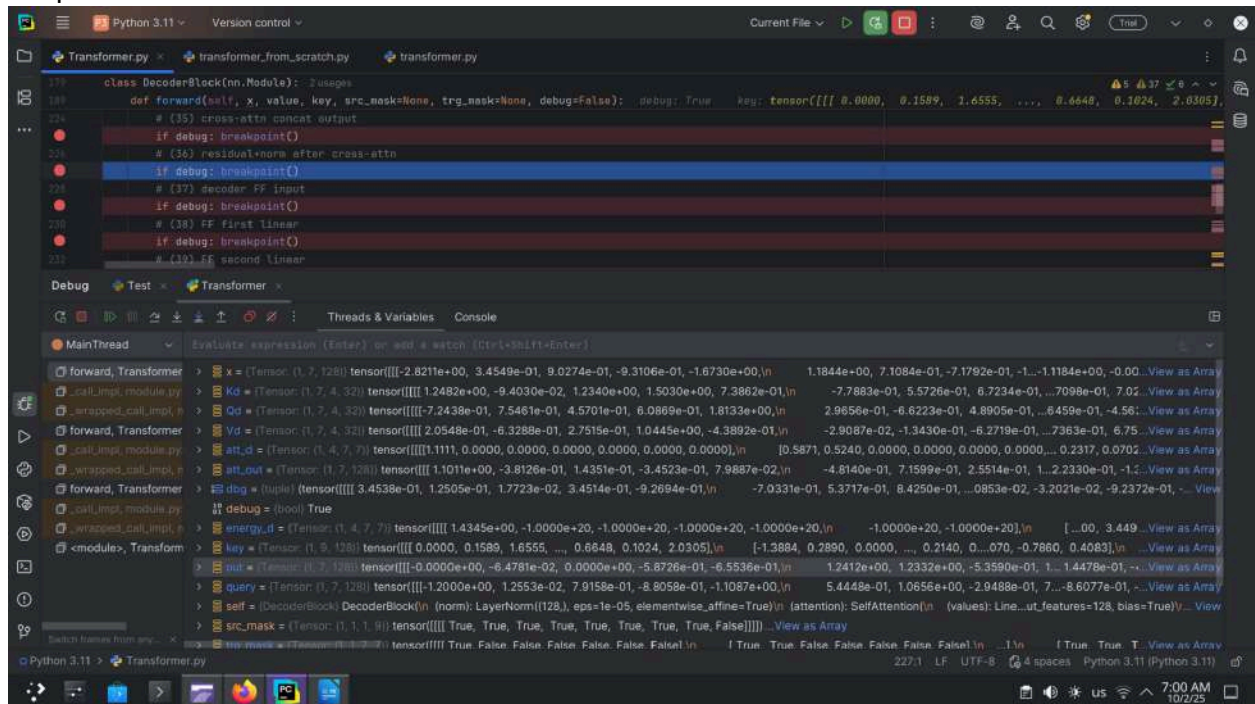


Tensor shape: (1, 7, 128)

Slice of values: `[[[0.2468, -0.0093, 0.0000, ..., -0.3323, -0.1044]]]`

The attended values from all heads concatenated and linearly projected, forming the cross-attention output.

Snapshot36: Residual + normalization after cross-attention.

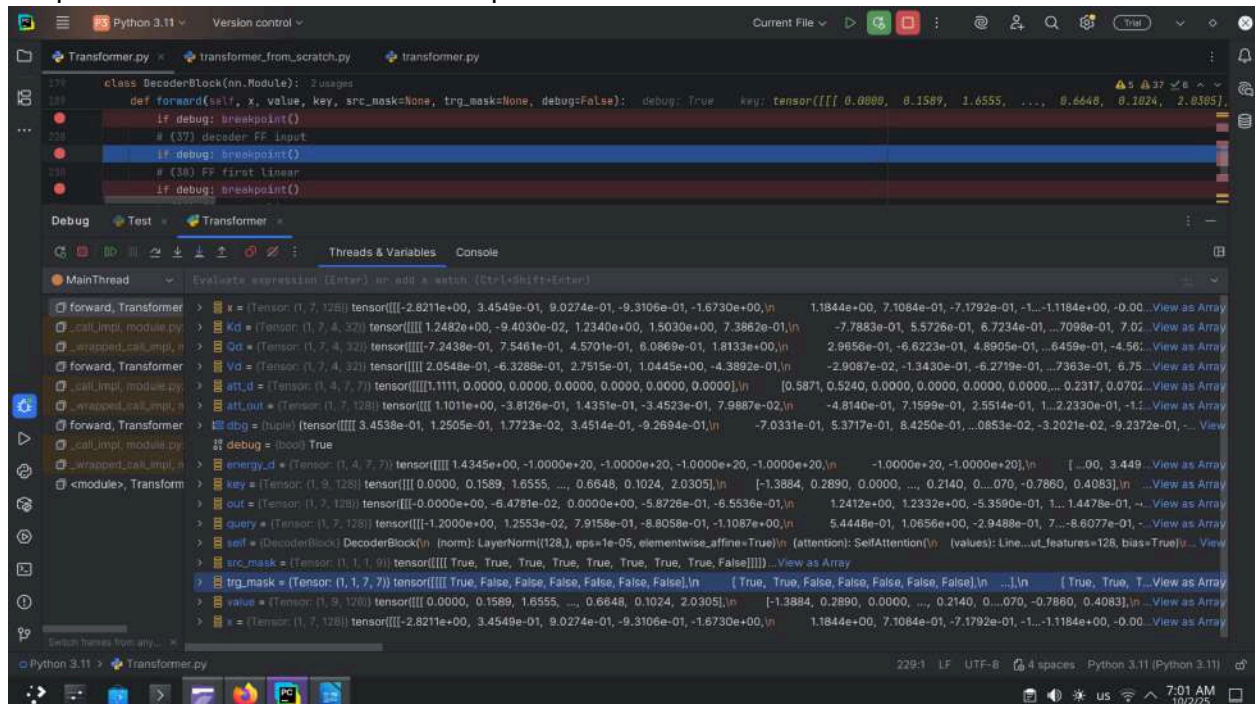


Tensor shape: (1, 7, 128)

Slice of values: `[[0.2489, -0.3606, 1.4464, -0.8921, 0.1097, ...]]`

The cross-attention output added to the residual input and normalized for stability.

Snapshot37: Decoder feed-forward input.

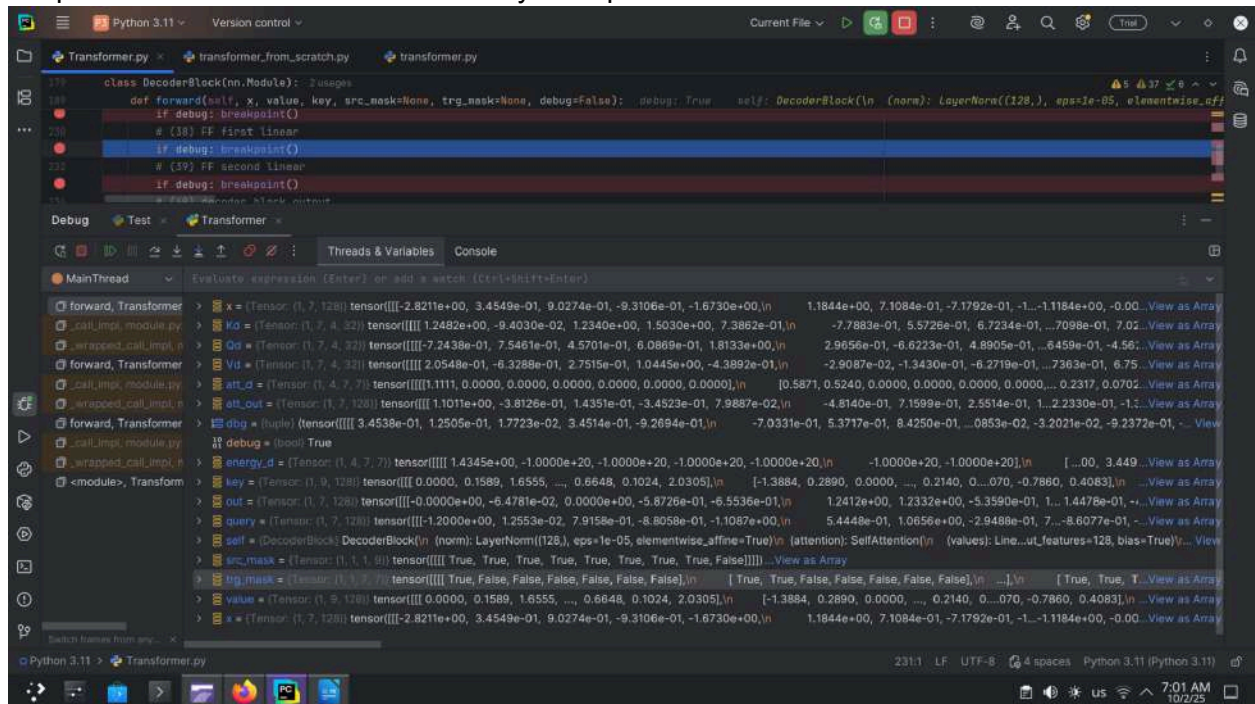


Tensor shape: (1, 7, 128)

Slice of values: `[[[0.2489, -0.3606, 1.4464, -0.8921, 0.1097, ...]]]`

Explanation: The normalized hidden states entering the feed-forward network.

Snapshot38: Feed-forward first linear layer output.



```
class DecoderBlock(nn.Module):
    def forward(self, x, value, key, src_mask=None, trg_mask=None, debug=False):
        # (38) FF first linear
        if debug: breakpoint()
        # (39) FF second linear
        if debug: breakpoint()
        # (40) Decoder block output
        return self.decoder_block_output

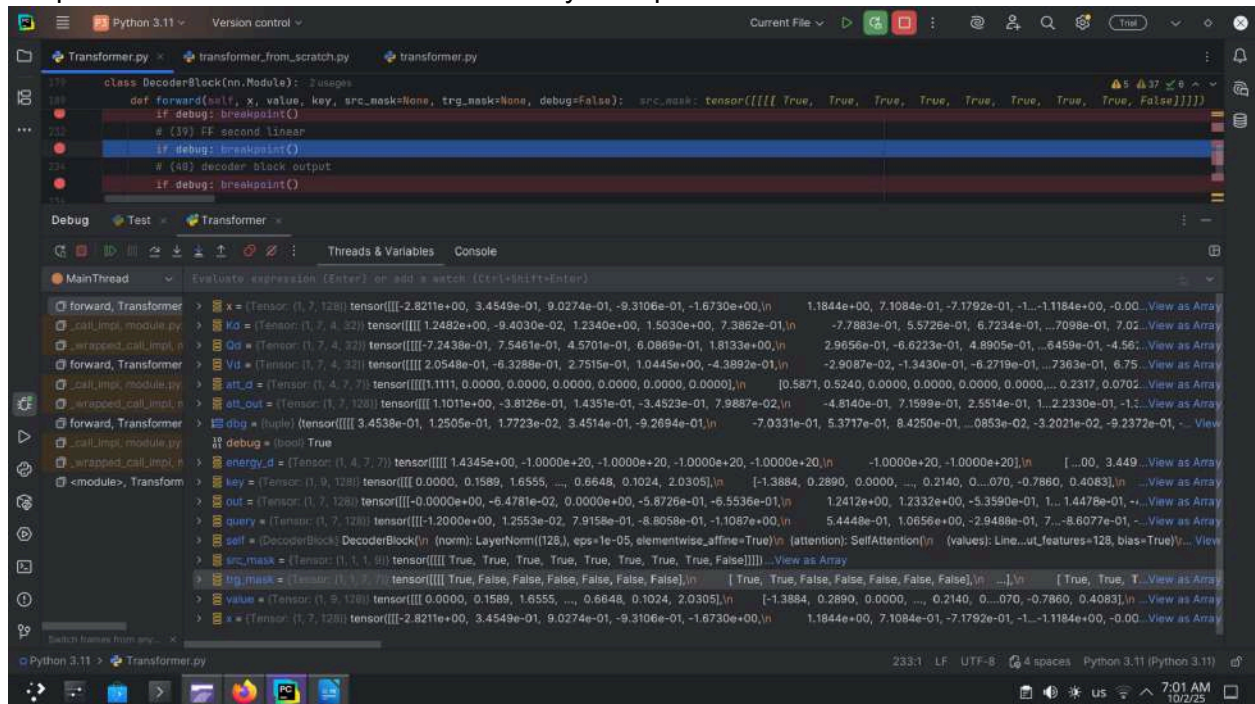
# MainThread
# Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)
# forward, Transformer
x = (Tensor: (1, 7, 128)) tensor([[-2.8211e+00, 3.4549e-01, 9.0274e-01, -9.3106e-01, -1.6730e+00, 1.1844e+00, 7.1084e-01, -7.1792e-01, -1.1184e+00, -0.00... View as Array
k_d = (Tensor: (1, 7, 4, 32)) tensor([[[[ 1.2482e+00, -9.4030e-02, 1.2340e+00, 1.5030e+00, 7.3862e-01, -7.7883e-01, 5.5726e-01, 6.7234e-01, ...7098e-01, 7.02... View as Array
q_d = (Tensor: (1, 7, 4, 32)) tensor([[[[ -7.2438e-01, 7.5461e-01, 4.5701e-01, 6.0869e-01, 1.8133e+00, 2.9656e-01, -6.6223e-01, 4.8905e-01, ...6459e-01, -4.56... View as Array
v_d = (Tensor: (1, 7, 4, 32)) tensor([[[[ 2.0548e-01, -6.3288e-01, 2.7515e-01, 1.0445e+00, -4.3892e-01, -2.9087e-02, -1.3430e-01, -6.2719e-01, ...7363e-01, 6.75... View as Array
att_d = (Tensor: (1, 4, 7, 7)) tensor([[[[ 1.1111, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, ...0.5871, 0.5240, 0.0000, 0.0000, 0.0000, 0.0000, ...0.2317, 0.0702... View as Array
att_out = (Tensor: (1, 7, 128)) tensor([[[[ 1.1011e+00, -3.8126e-01, 1.4351e-01, -3.4523e-01, 7.9887e-02, -4.8140e-01, 7.1599e-01, 2.5514e-01, ...2.2330e-01, -1.1... View as Array
dbg = (tuple) (tensor([[[[ 3.4538e-01, 1.2505e-01, 1.7723e-02, 3.4514e-01, -9.2694e-01, -7.0331e-01, 5.3717e-01, 8.4250e-01, ...0853e-02, -3.2021e-02, -9.2372e-01, -... View as Array
debug = (bool) True
energy_d = (Tensor: (1, 4, 7, 7)) tensor([[[[ 1.4345e+00, -1.0000e+20, -1.0000e+20, -1.0000e+20, -1.0000e+20, -1.0000e+20, -1.0000e+20, -1.0000e+20, ...0.00, 3.449... View as Array
key = (Tensor: (1, 9, 128)) tensor([[[[ 0.0000, 0.1589, 1.6555, ..., 0.6648, 0.1024, 2.0305], ..., [-1.3884, 0.2890, 0.0000, ..., 0.2140, 0.070, -0.7860, 0.4083], ... View as Array
out = (Tensor: (1, 7, 128)) tensor([[[[ -0.0000e+00, -6.4781e-02, 0.0000e+00, -5.8726e-01, -6.5536e-01, 1.2412e+00, 1.2332e+00, -5.3590e-01, ...1.4478e-01, -... View as Array
query = (Tensor: (1, 7, 128)) tensor([[[[ -1.2000e+00, 1.2553e-02, 7.9158e-01, -8.8058e-01, -1.1087e+00, 5.4448e-01, 1.0656e+00, -2.9488e-01, ...-8.6077e-01, -... View as Array
self = (DecoderBlock) DecoderBlock((norm): LayerNorm(128), eps=1e-05, elementwise_affine=True) (attention): SelfAttention((values): Linear(...features=128, bias=True)...) View as Array
src_mask = (Tensor: (1, 1, 1, 9)) tensor([[[[ True, True, True, True, True, True, True, True, False]]]] ... View as Array
trg_mask = (Tensor: (1, 1, 9, 7)) tensor([[[[ True, False, False, False, False, False, False], ..., [ True, True, False, False, False, False, False], ...], ..., [ True, True, T... View as Array
value = (Tensor: (1, 9, 128)) tensor([[[[ 0.0000, 0.1589, 1.6555, ..., 0.6648, 0.1024, 2.0305], ..., [-1.3884, 0.2890, 0.0000, ..., 0.2140, 0.070, -0.7860, 0.4083], ... View as Array
x = (Tensor: (1, 7, 128)) tensor([[[[ -2.8211e+00, 3.4549e-01, 9.0274e-01, -9.3106e-01, -1.6730e+00, 1.1844e+00, 7.1084e-01, -7.1792e-01, -1.1184e+00, -0.00... View as Array
```

Tensor shape: (1, 7, 512)

Slice of values: `[[[0.1859, 0.5946, 0.0402, ..., -0.9247, -0.6089]]]`

The output after applying the first linear transformation in the feed-forward network, expanding dimensionality.

Snapshot39: Feed-forward second linear layer output.

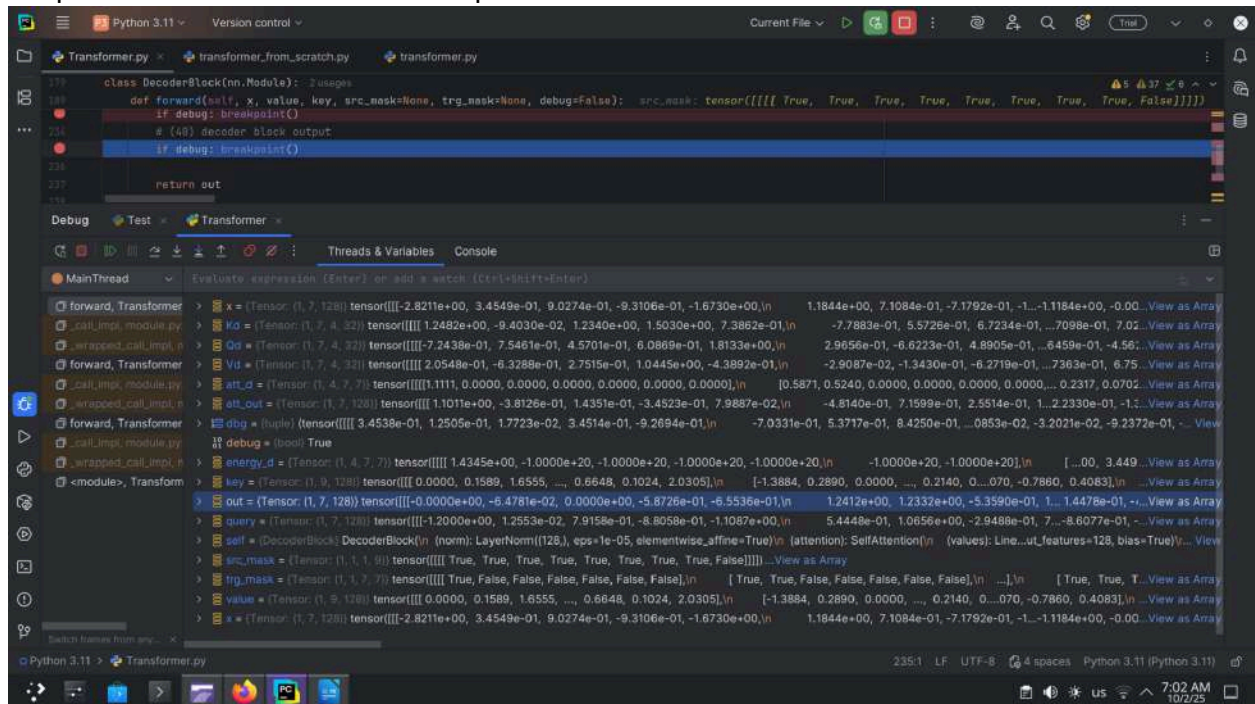


Tensor shape: (1, 7, 128)

Slice of values: `[[[1.1881, 0.1314, -0.8549, ..., 0.2898, -0.0338]]]`

The output after projecting back to model dimension, preparing for residual connection.

Snapshot40: Decoder block final output tensor.




Tensor shape: (1, 7, 128)

Slice of values: `[[[0.2489, -0.3606, 1.4464, -0.8921, 0.1097, ...]]]`

Final hidden states of the decoder block after cross-attention and feed-forward, passed to the next decoder block or output layer.

Snapshot 41: Decoder final sequence output (before projection).



```
class Decoder(nn.Module):
    def forward(self, x, enc_out, src_mask=None, trg_mask=None, debug=False):
        # (42) decoder output before projection
        if debug: breakpoint()

        out = self.fc_out(x)
        # (42) logits
        if debug: breakpoint()

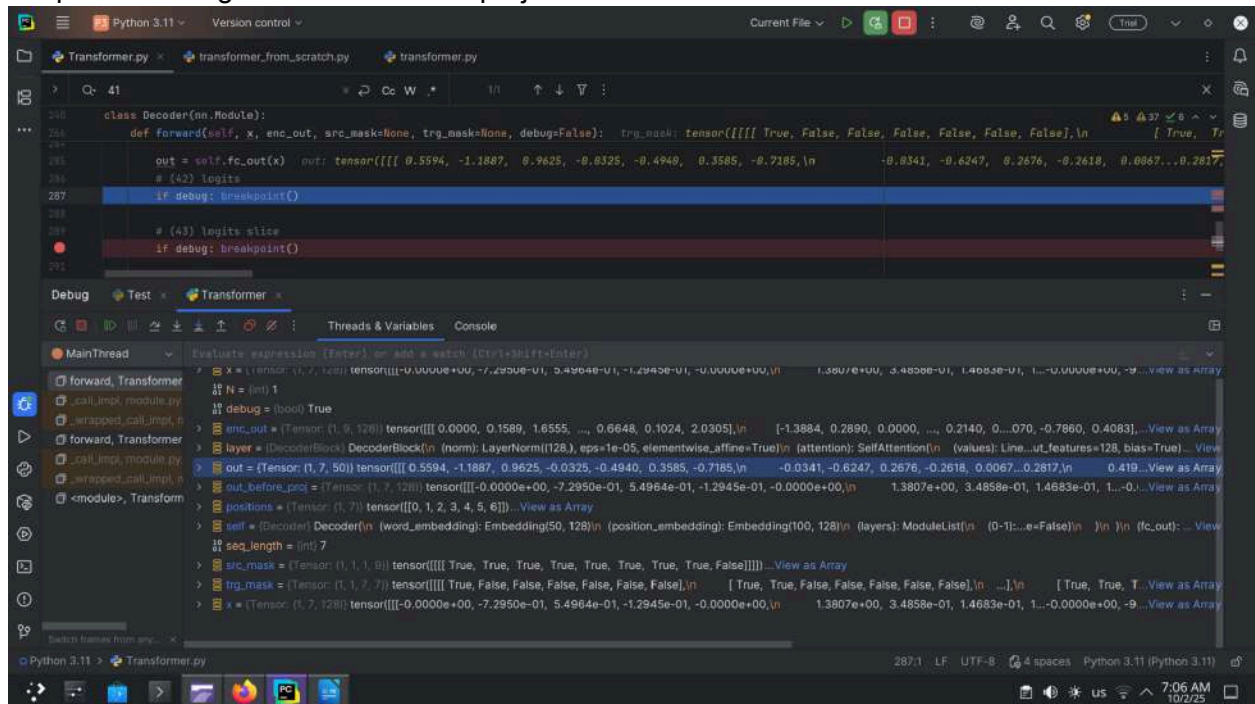
        x = (Tensor(1, 7, 128) tensor[[[-0.0000e+00, -7.2950e-01, 5.4964e-01, -1.2945e-01, -0.0000e+00, ..., 1.3807e+00, 3.4858e-01, 1.4683e-01, ..., -0.0000e+00, -9.1388e-01, 0.2890, 0.0000, ..., 0.2140, -0.070, -0.7860, 0.4083], ...])
        debug = (bool) True
        enc_out = (Tensor(1, 9, 128) tensor[[[0.0000, 0.1589, 1.6555, ..., 0.6648, 0.1024, 2.0305], ...])
        layer = (DecoderBlock DecoderBlock(in_norm: LayerNorm(128), eps=1e-05, elementwise_affine=True)(in_attention: SelfAttention(in_values: Line...ut_features=128, bias=True)...))
        out_before_proj = (Tensor(1, 7, 128) tensor[[[-0.0000e+00, -7.2950e-01, 5.4964e-01, -1.2945e-01, -0.0000e+00, ..., 1.3807e+00, 3.4858e-01, 1.4683e-01, ..., -0.0000e+00, -9.1388e-01, 0.2890, 0.0000, ..., 0.2140, -0.070, -0.7860, 0.4083], ...])
        positions = (Tensor(1, 7) tensor[[[0, 1, 2, 3, 4, 5, 6], ...])
        self = (Decoder) Decoder(in_word_embedding: Embedding(50, 128)(in_position_embedding: Embedding(100, 128)(in_layers: ModuleList(in_0-1)...e=False)(in_2)...in_fc_out: ...))
        seq_length = (int) 7
        src_mask = (Tensor(1, 1, 9) tensor[[[True, True, True, True, True, True, True, True, False], ...])
        trg_mask = (Tensor(1, 7, 7) tensor[[[True, False, False, False, False, False, False], ...])
        x = (Tensor(1, 7, 128) tensor[[[-0.0000e+00, -7.2950e-01, 5.4964e-01, -1.2945e-01, -0.0000e+00, ..., 1.3807e+00, 3.4858e-01, 1.4683e-01, ..., -0.0000e+00, -9.1388e-01, 0.2890, 0.0000, ..., 0.2140, -0.070, -0.7860, 0.4083], ...])
```

Tensor shape: (1, 7, 128)

Slice of values: `[[[0.2614, 0.0205, 1.9414, 0.5245, -0.2197, -1.0555, 0.0000, 0.2613]]]`

decoder hidden representation before the final linear layer, capturing contextualized token embeddings of size 128.

Snapshot 42: Logits after final linear projection.



```
class Decoder(nn.Module):
    def forward(self, x, enc_out, src_mask=None, trg_mask=None, debug=False):
        # (42) logits
        # (43) logits slice
        if debug: breakpoint()

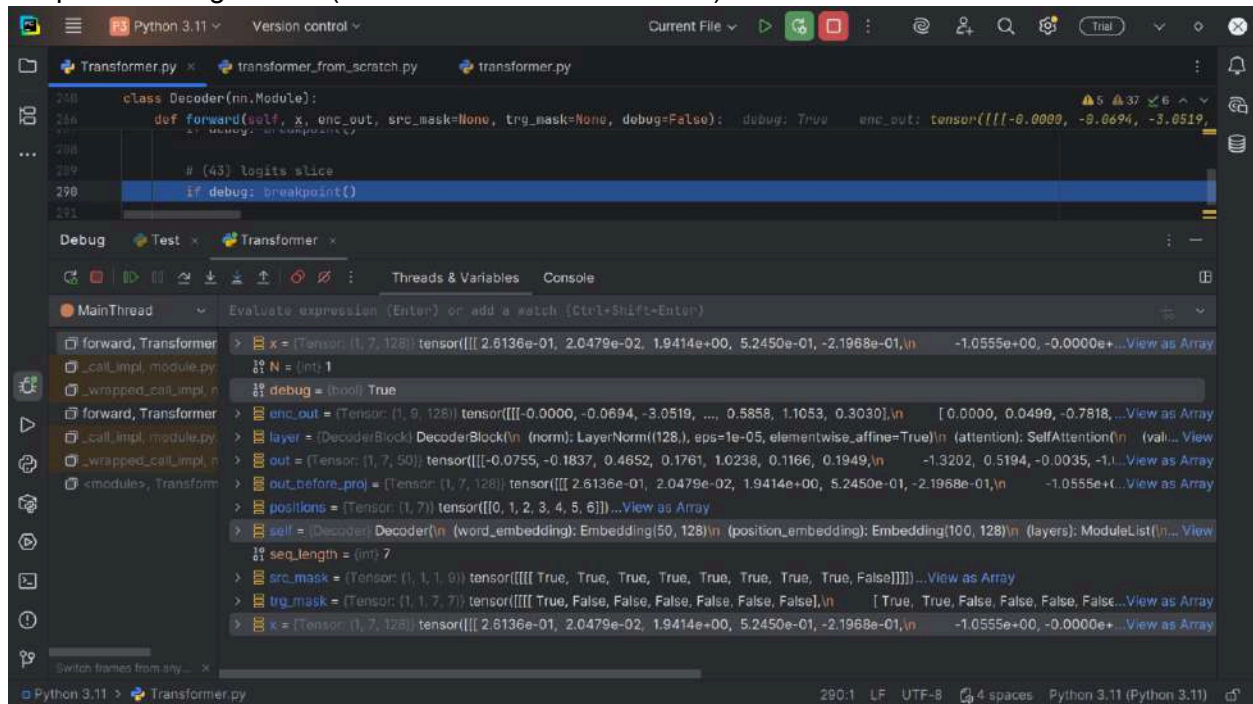
out = (Tensor(1, 7, 50)) tensor([[-0.0755, -0.1837, 0.4652, 0.1761, 1.0238, 0.1166, 0.1949, -1.3202], ...])
```

Tensor shape: (1, 7, 50)

Slice of values: `[[-0.0755, -0.1837, 0.4652, 0.1761, 1.0238, 0.1166, 0.1949, -1.3202]]`

These are the unnormalized prediction scores over the vocabulary (size 50) for each of the 7 target positions.

Snapshot43: Logits slice (first few values for one token).



Tensor shape: (1, 7, 50)

Slice of values: `[[[-0.0755, -0.1837, 0.4652, 0.1761, 1.0238]]]`

A narrowed view of logits showing the first few vocabulary scores, used to inspect how likely each token is before softmax.