```python
def create_patches(self, x):
    """
    Convert input images to patches and flatten them for transformer input.
    """
    batch_size, channels, height, width = x.shape
    patches = x.unfold(2, self.patch_size, self.patch_size).unfold(3, self.patch_size, self.patch_size)
    patches = patches.contiguous().view(batch_size, channels, -1, self.patch_size * self.patch_size)  # Flatten patches
    patches = patches.permute(2, 0, 1, 3).contiguous().view(-1, batch_size, self.patch_size * self.patch_size * channels)  # Rearrange for transformer
    patches = self.patch_to_embedding(patches)
    return patches
```

Input Shape:

- `x.shape = [1, 3, 224, 224]`: This means you have a batch size of 1, with 3 channels (RGB), and each image is 224x224 pixels.

Line-by-Line Breakdown:

1. `batch_size, channels, height, width = x.shape`

- **Explanation**: This unpacks the shape of `x`.
    - `batch_size = 1`: Batch size of 1.
    - `channels = 3`: 3 channels (e.g., RGB image).
    - `height = 224`: Image height.
    - `width = 224`: Image width.

2. `patches = x.unfold(2, self.patch_size, self.patch_size).unfold(3, self.patch_size, self.patch_size)`

- **Explanation**: This splits the image into non-overlapping patches along the height and width dimensions.
    - `x.unfold(2, self.patch_size, self.patch_size)`: This splits the image along the height dimension (`dim=2`), where `self.patch_size` is the size of each patch (e.g., 16).
        - If `self.patch_size = 16`, this divides the 224x224 image into 14 patches along the height (224 // 16 = 14).
    - `unfold(3, self.patch_size, self.patch_size)`: This splits the image along the width dimension (`dim=3`), also into 14 patches along the width.
        - Final patch shape: `(batch_size, channels, num_patches_height, num_patches_width, patch_size, patch_size)`.
        - After this, `patches.shape = [1, 3, 14, 14, 16, 16]` because each 224x224 image is split into 14x14 patches of size 16x16.

torch.Tensor.unfold

Tensor.unfold(*dimension*, *size*, *step*) → Tensor

Returns a view of the original tensor which contains all slices of size `size` from `self` tensor in the dimension `dimension`.

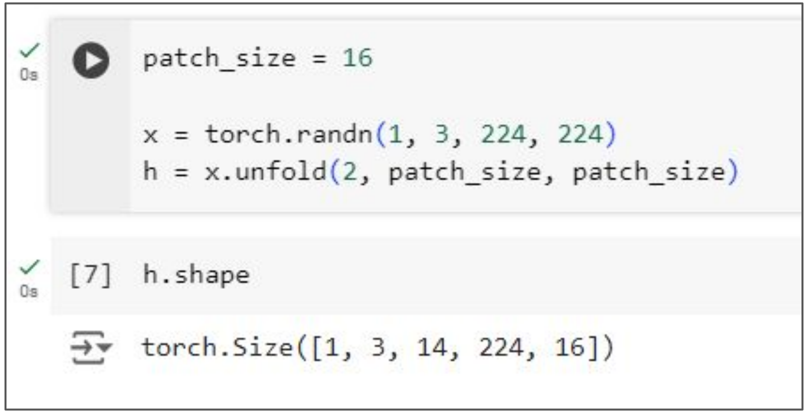
Step between two slices is given by `step`.

If *sizedim* is the size of dimension `dimension` for `self`, the size of dimension `dimension` in the returned tensor will be $(sizedim - size) / step + 1$.

An additional dimension of size `size` is appended in the returned tensor.

Parameters

- **dimension** (*int*) – dimension in which unfolding happens
- **size** (*int*) – the size of each slice that is unfolded
- **step** (*int*) – the step between each slice



```
patch_size = 16

x = torch.randn(1, 3, 224, 224)
h = x.unfold(2, patch_size, patch_size)
```

```
[7] h.shape
```

```
torch.Size([1, 3, 14, 224, 16])
```

Example:

```
>>> x = torch.arange(1., 8)
>>> x
tensor([ 1.,  2.,  3.,  4.,  5.,  6.,  7.])
>>> x.unfold(0, 2, 1)
tensor([[ 1.,  2.],
        [ 2.,  3.],
        [ 3.,  4.],
        [ 4.,  5.],
        [ 5.,  6.],
        [ 6.,  7.]])
>>> x.unfold(0, 2, 2)
tensor([[ 1.,  2.],
        [ 3.,  4.],
        [ 5.,  6.]])
```

Let's take a smaller dimension, for example, a tensor of shape `[1, 3, 8, 8]` with `patch_size = 4`. This will make the unfolded patches easier to visualize.

1. Start with a tensor of shape `[1, 3, 8, 8]` (representing 1 image with 3 channels, and each channel is an 8x8 matrix).
2. Apply `unfold` with `patch_size = 4`.

The resulting shape will be `[1, 3, 2, 8, 4]`:

- `1`: Batch size.
- `3`: Number of channels (RGB).
- `2`: Number of patches along the height (8 pixels divided into two 4-pixel patches).
- `8`: The width remains unchanged since no unfolding happens along this dimension.
- `4`: The patch size, which is 4 rows of pixels from the height.

Here's how the process unfolds step by step for the smaller tensor of shape `[1, 3, 8, 8]` with `patch_size = 4`.

Original Tensor:

The original tensor `x` consists of 3 channels (RGB) with an 8x8 grid for each channel:

Channel 1:

```css
[[  1,   2,   3,   4,   5,   6,   7,   8],
 [  9,  10,  11,  12,  13,  14,  15,  16],
 [ 17,  18,  19,  20,  21,  22,  23,  24],
 [ 25,  26,  27,  28,  29,  30,  31,  32],
 [ 33,  34,  35,  36,  37,  38,  39,  40],
 [ 41,  42,  43,  44,  45,  46,  47,  48],
 [ 49,  50,  51,  52,  53,  54,  55,  56],
 [ 57,  58,  59,  60,  61,  62,  63,  64]]
```

Channel 2:

```css
[[ 65,  66,  67,  68,  69,  70,  71,  72],
 [ 73,  74,  75,  76,  77,  78,  79,  80],
 [ 81,  82,  83,  84,  85,  86,  87,  88],
 [ 89,  90,  91,  92,  93,  94,  95,  96],
 [ 97,  98,  99, 100, 101, 102, 103, 104],
 [105, 106, 107, 108, 109, 110, 111, 112],
 [113, 114, 115, 116, 117, 118, 119, 120],
 [121, 122, 123, 124, 125, 126, 127, 128]]
```

Channel 3:

```css
[[129, 130, 131, 132, 133, 134, 135, 136],
 [137, 138, 139, 140, 141, 142, 143, 144],
 [145, 146, 147, 148, 149, 150, 151, 152],
 [153, 154, 155, 156, 157, 158, 159, 160],
 [161, 162, 163, 164, 165, 166, 167, 168],
 [169, 170, 171, 172, 173, 174, 175, 176],
 [177, 178, 179, 180, 181, 182, 183, 184],
 [185, 186, 187, 188, 189, 190, 191, 192]]
```

Unfolded Patches:

The unfolding operation creates patches of size 4 from the height dimension. Each patch takes 4 consecutive rows. For example, the first unfolded patch for **Channel 1** looks like this:

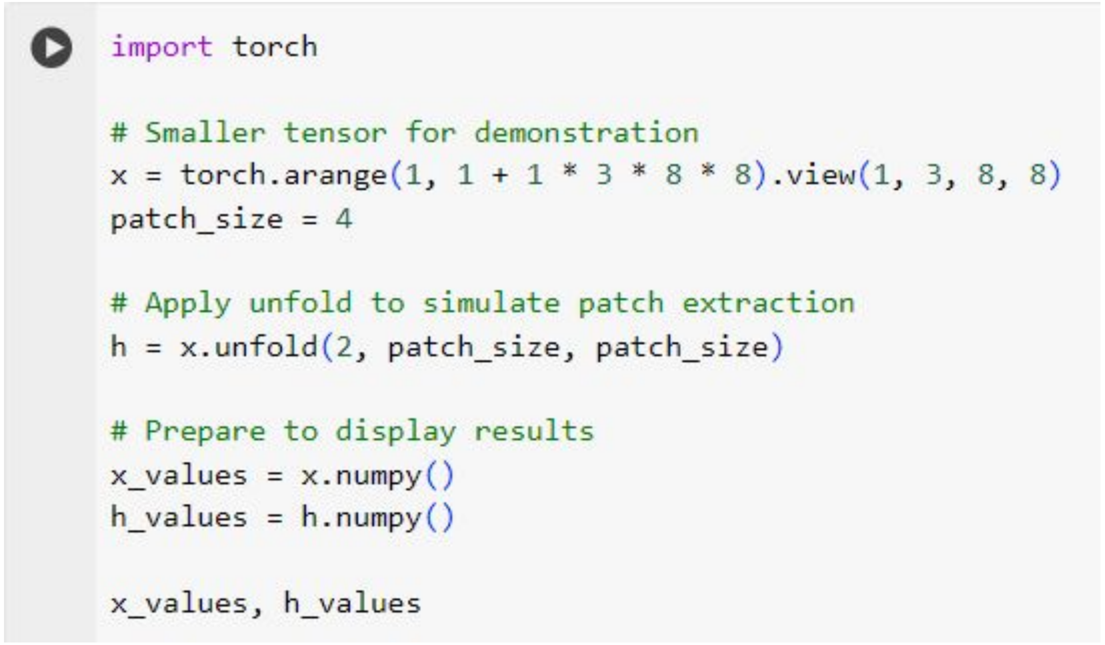
Patch 1:

```css
[[  1,   9,  17,  25],
 [  2,  10,  18,  26],
 [  3,  11,  19,  27],
 [  4,  12,  20,  28],
 [  5,  13,  21,  29],
 [  6,  14,  22,  30],
 [  7,  15,  23,  31],
 [  8,  16,  24,  32]]
```

Patch 2 for Channel 1 is:

```css
[[ 33,  41,  49,  57],
 [ 34,  42,  50,  58],
 [ 35,  43,  51,  59],
 [ 36,  44,  52,  60],
 [ 37,  45,  53,  61],
 [ 38,  46,  54,  62],
 [ 39,  47,  55,  63],
 [ 40,  48,  56,  64]]
```

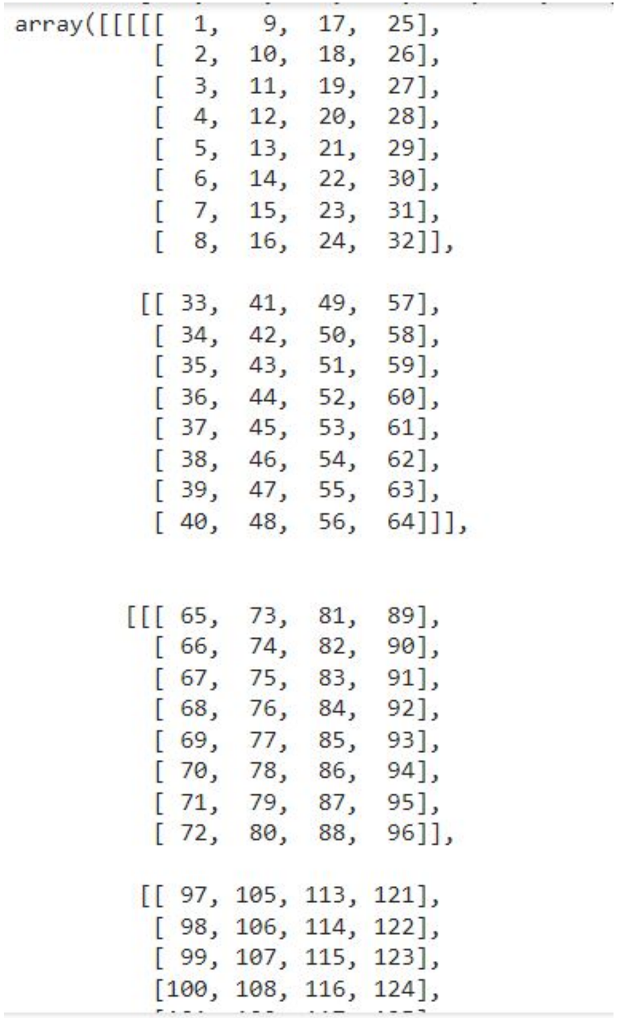
Each channel undergoes this unfolding process, and similar patches are created for the second and third channels. This makes the data more manageable for input into a transformer-based model.



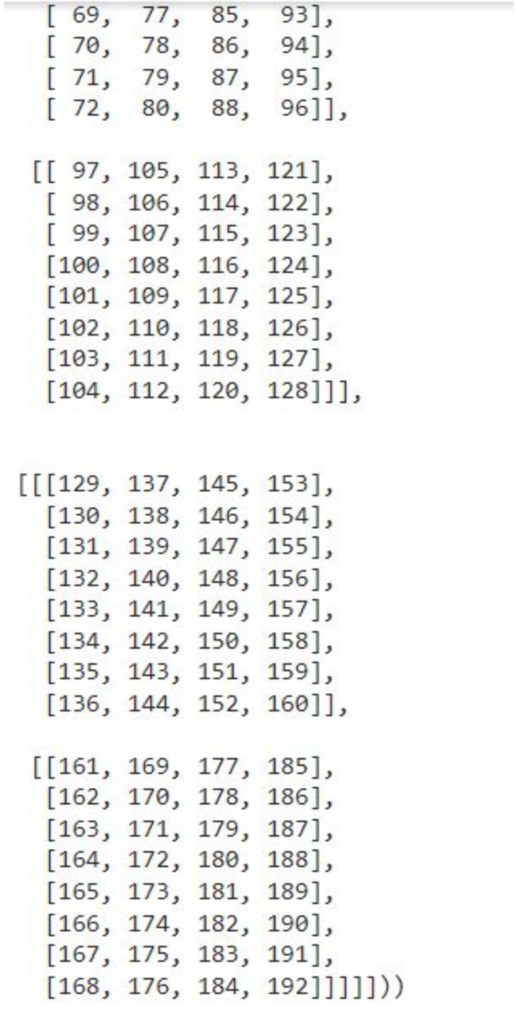
```
import torch

# Smaller tensor for demonstration
x = torch.arange(1, 1 + 1 * 3 * 8 * 8).view(1, 3, 8, 8)
patch_size = 4

# Apply unfold to simulate patch extraction
h = x.unfold(2, patch_size, patch_size)

# Prepare to display results
x_values = x.numpy()
h_values = h.numpy()

x_values, h_values
```

Output in next two slides

```
(array([[[[  1,   2,   3,   4,   5,   6,   7,   8],
          [  9,  10,  11,  12,  13,  14,  15,  16],
          [ 17,  18,  19,  20,  21,  22,  23,  24],
          [ 25,  26,  27,  28,  29,  30,  31,  32],
          [ 33,  34,  35,  36,  37,  38,  39,  40],
          [ 41,  42,  43,  44,  45,  46,  47,  48],
          [ 49,  50,  51,  52,  53,  54,  55,  56],
          [ 57,  58,  59,  60,  61,  62,  63,  64]],

         [[ 65,  66,  67,  68,  69,  70,  71,  72],
          [ 73,  74,  75,  76,  77,  78,  79,  80],
          [ 81,  82,  83,  84,  85,  86,  87,  88],
          [ 89,  90,  91,  92,  93,  94,  95,  96],
          [ 97,  98,  99, 100, 101, 102, 103, 104],
          [105, 106, 107, 108, 109, 110, 111, 112],
          [113, 114, 115, 116, 117, 118, 119, 120],
          [121, 122, 123, 124, 125, 126, 127, 128]],

         [[129, 130, 131, 132, 133, 134, 135, 136],
          [137, 138, 139, 140, 141, 142, 143, 144],
          [145, 146, 147, 148, 149, 150, 151, 152],
          [153, 154, 155, 156, 157, 158, 159, 160],
          [161, 162, 163, 164, 165, 166, 167, 168],
          [169, 170, 171, 172, 173, 174, 175, 176],
          [177, 178, 179, 180, 181, 182, 183, 184],
          [185, 186, 187, 188, 189, 190, 191, 192]]]]),
```



```
array([[[[[  1,   9,  17,  25],
          [  2,  10,  18,  26],
          [  3,  11,  19,  27],
          [  4,  12,  20,  28],
          [  5,  13,  21,  29],
          [  6,  14,  22,  30],
          [  7,  15,  23,  31],
          [  8,  16,  24,  32]],

         [[ 33,  41,  49,  57],
          [ 34,  42,  50,  58],
          [ 35,  43,  51,  59],
          [ 36,  44,  52,  60],
          [ 37,  45,  53,  61],
          [ 38,  46,  54,  62],
          [ 39,  47,  55,  63],
          [ 40,  48,  56,  64]]],


        [[[ 65,  73,  81,  89],
          [ 66,  74,  82,  90],
          [ 67,  75,  83,  91],
          [ 68,  76,  84,  92],
          [ 69,  77,  85,  93],
          [ 70,  78,  86,  94],
          [ 71,  79,  87,  95],
          [ 72,  80,  88,  96]],

         [[ 97, 105, 113, 121],
          [ 98, 106, 114, 122],
          [ 99, 107, 115, 123],
          [100, 108, 116, 124],
```



```
          [ 69,  77,  85,  93],
          [ 70,  78,  86,  94],
          [ 71,  79,  87,  95],
          [ 72,  80,  88,  96]],

         [[ 97, 105, 113, 121],
          [ 98, 106, 114, 122],
          [ 99, 107, 115, 123],
          [100, 108, 116, 124],
          [101, 109, 117, 125],
          [102, 110, 118, 126],
          [103, 111, 119, 127],
          [104, 112, 120, 128]]],


        [[[129, 137, 145, 153],
          [130, 138, 146, 154],
          [131, 139, 147, 155],
          [132, 140, 148, 156],
          [133, 141, 149, 157],
          [134, 142, 150, 158],
          [135, 143, 151, 159],
          [136, 144, 152, 160]],

         [[161, 169, 177, 185],
          [162, 170, 178, 186],
          [163, 171, 179, 187],
          [164, 172, 180, 188],
          [165, 173, 181, 189],
          [166, 174, 182, 190],
          [167, 175, 183, 191],
          [168, 176, 184, 192]]]]])
```

224x224 image is split into `(224 // 16) ^ 2 = 196` patches.

Let's walk through a smaller example with a tensor of shape `[1, 3, 4, 4]` and `patch_size = 2`. This makes it easier to visualize what's happening in each step of the unfolding process.

Original Tensor `x`:

Imagine we have a tensor `x` of shape `[1, 3, 4, 4]`, where:

- 1: batch size
- 3: number of channels (e.g., RGB)
- 4: height and width of the image (a 4x4 grid)

For simplicity, let's assume each channel has the following values:

Channel 1:

```lua
[[1, 2, 3, 4],
 [5, 6, 7, 8],
 [9, 10, 11, 12],
 [13, 14, 15, 16]]
```

Channel 2:

```lua
[[17, 18, 19, 20],
 [21, 22, 23, 24],
 [25, 26, 27, 28],
 [29, 30, 31, 32]]
```

Channel 3:

```lua
[[33, 34, 35, 36],
 [37, 38, 39, 40],
 [41, 42, 43, 44],
 [45, 46, 47, 48]]
```

Step 1: Unfold the Tensor into Patches

We want to split this 4x4 grid into patches of size 2x2. Unfolding means creating patches of this size across both the height and width dimensions.

For **Channel 1**, the patches would look like this:

- **Patch 1 (top-left):**

```lua
[[1, 2],
 [5, 6]]
```

- **Patch 2 (top-right):**

```lua
[[3, 4],
 [7, 8]]
```

- **Patch 3 (bottom-left):**

```lua
[[9, 10],
 [13, 14]]
```

- **Patch 4 (bottom-right):**

```lua
[[11, 12],
 [15, 16]]
```

Similarly, patches are created for **Channel 2** and **Channel 3**.

Step 2: Flatten the Patches

Each patch (which is a 2x2 grid) is flattened into a 1D vector of length 4. So each 2x2 patch becomes:

- **Patch 1 (top-left) for Channel 1:**

```csharp
[1, 2, 5, 6]
```

- **Patch 2 (top-right) for Channel 1:**

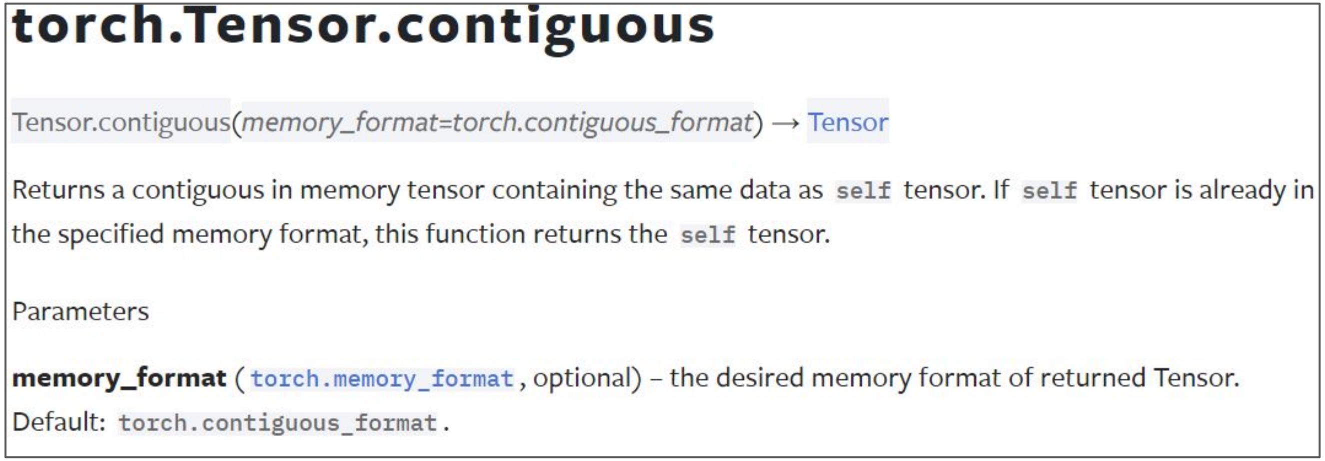
```csharp
[3, 4, 7, 8]
```

This flattening happens for all patches and channels.

Step 3: Reshape for Transformer Input

After unfolding and flattening, the patches are rearranged into a shape suitable for transformer input. The result is a tensor of shape `[1, 3, 4, 4]` (4 patches per channel, with 4 elements per patch).

Finally, after permuting and combining, these patches are passed through the `patch_to_embedding` layer, which maps them to the input space of the transformer.



torch.Tensor.contiguous

Tensor.contiguous(*memory_format=torch.contiguous_format*) → Tensor

Returns a contiguous in memory tensor containing the same data as `self` tensor. If `self` tensor is already in the specified memory format, this function returns the `self` tensor.

Parameters

memory_format (`torch.memory_format`, optional) – the desired memory format of returned Tensor. Default: `torch.contiguous_format`.

Example of Positional Encoding:

Let's consider a simple example to illustrate the concept of positional encoding in the context of a Transformer model.

Suppose we have a Transformer model tasked with translating English sentences into French. One of the sentences in English is:

"The cat sat on the mat."

Before the sentence is fed into the Transformer model, it undergoes tokenization, where each word is converted into a token. Let's assume the tokens for this sentence are:

["The", "cat" , "sat", "on", "the" ,"mat"]

Next, each token is mapped to a high-dimensional vector representation through an embedding layer. These embeddings encode semantic information about the words in the sentence. However, they lack information about the order of the words.

Embeddings={**E1**,**E2**,**E3**,**E4**,**E5**,**E6**}

where each **Ei** is a 4-dimensional vector.

This is where positional encoding comes into play. To ensure that the model understands the order of the words in the sequence, positional encodings are added to the word embeddings. These encodings provide each token with a unique positional representation.

Calculating Positional Encodings

- Let's say the embedding dimensionality is 4 for simplicity.
- We'll use sine and cosine functions to generate positional encodings. Consider the following positional encodings for the tokens in our example sentence:

$$\text{PE}(1) = [\sin(\tfrac{1}{10000^{2\times 0/4}}), \cos(\tfrac{1}{10000^{2\times 0/4}}), \sin(\tfrac{1}{10000^{2\times 1/4}}), \cos(\tfrac{1}{10000^{2\times 1/4}})]$$
$$\text{PE}(2) = [\sin(\tfrac{2}{10000^{2\times 0/4}}), \cos(\tfrac{2}{10000^{2\times 0/4}}), \sin(\tfrac{2}{10000^{2\times 1/4}}), \cos(\tfrac{2}{10000^{2\times 1/4}})]$$
$$\text{PE}(3) = [\sin(\tfrac{3}{10000^{2\times 0/4}}), \cos(\tfrac{3}{10000^{2\times 0/4}}), \sin(\tfrac{3}{10000^{2\times 1/4}}), \cos(\tfrac{3}{10000^{2\times 1/4}})]$$
$$\text{PE}(4) = [\sin(\tfrac{4}{10000^{2\times 0/4}}), \cos(\tfrac{4}{10000^{2\times 0/4}}), \sin(\tfrac{4}{10000^{2\times 1/4}}), \cos(\tfrac{4}{10000^{2\times 1/4}})]$$
$$\text{PE}(5) = [\sin(\tfrac{5}{10000^{2\times 0/4}}), \cos(\tfrac{5}{10000^{2\times 0/4}}), \sin(\tfrac{5}{10000^{2\times 1/4}}), \cos(\tfrac{5}{10000^{2\times 1/4}})]$$
$$\text{PE}(6) = [\sin(\tfrac{6}{10000^{2\times 0/4}}), \cos(\tfrac{6}{10000^{2\times 0/4}}), \sin(\tfrac{6}{10000^{2\times 1/4}}), \cos(\tfrac{6}{10000^{2\times 1/4}})]$$

- These positional encodings are added element-wise to the word embeddings. The resulting vectors contain both semantic and positional information, allowing the Transformer model to understand not only the meaning of each word but also its position in the sequence.

The line `x_patches = torch.cat((cls_tokens, x_patches), dim=0)`. Here, you are concatenating the class token to the beginning of the patch embeddings along dimension 0, which represents the patch sequence.
After concatenation, the shape of `x_patches` should be `( n_patches + 1,batch_size, embedding_dim)`, where `n_patches + 1` accounts for the extra class token.