

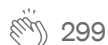


Positional Encoding



Hunter Phillips · Following

13 min read · May 9



This article is the second in The Implemented Transformer series. It introduces positional encoding from scratch. Then, it explains how PyTorch implements positional encoding. This is followed by the transformers implementation.

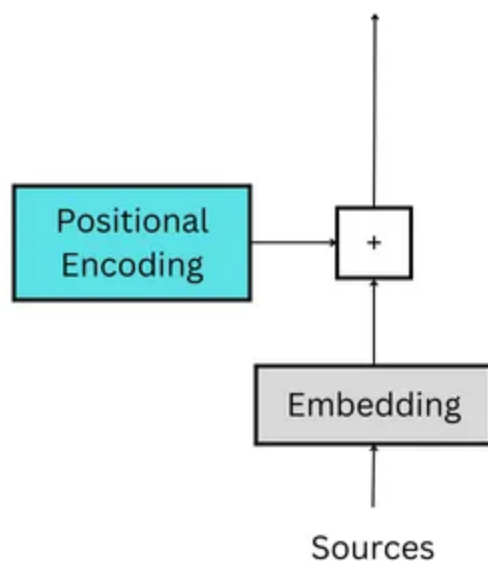


Image by Author

Background

Positional encoding is used to provide a relative position for each token or word in a sequence. When reading a sentence, each word is dependent on the words around it. For instance, some words have different meanings in different contexts, so a model should be able to understand these variations and the words that each relies on for context. An example is the word “trunk.” In one instance, it could be used to refer to an elephant using its trunk to drink water. In another, it could refer to a tree’s trunk being struck by lightning.

Since the model uses embedding vectors of length d_{model} to represent each word, any positional encoding has to be compatible. It may seem natural to use integers, with the first token receiving a 0, the second token receiving a 1,

and so on. However, this number quickly grows and cannot be easily added to an embedding matrix. Instead, a positional encoding vector is created for each position, meaning a positional encoding matrix can be created to represent all the possible positions a word can take.

To ensure each position has a unique representation, the authors of “Attention Is All You Need” used the sine and cosine functions to generate a unique vector for each position in the sequence. While this may seem strange, there are a few reasons why its useful. First of all, the output of sine and cosine is in $[-1, 1]$, which is normalized. It won't grow to an unmanageable size like integers would. Second, no additional training has to be done since unique representations are generated for each position.

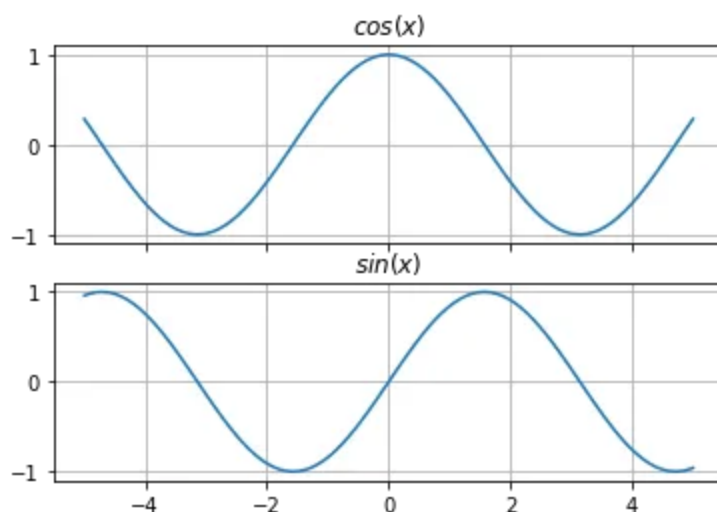


Image by Author

The equations used to generate the unique encodings appear daunting, but they are just modified versions of sine and cosine. The maximum number of positions, or vectors, in the embedding will be represented by L :

- For each $k = 0$ to $L - 1$:
 - For each $i = 0$ to $\frac{d_{model}}{2}$:
 - $PE_{(k,2i)} = \sin\left(\frac{k}{\frac{2i}{n^{d_{model}}}}\right)$
 - $PE_{(k,2i+1)} = \cos\left(\frac{k}{\frac{2i}{n^{d_{model}}}}\right)$

Image by Author

This essentially says that for each positional encoding vector, for every two elements, set the even element equal to $PE(k, 2i)$ and set the odd element equal to $PE(k, 2i+1)$. Then, repeat until there are d_{model} elements in the vector.

Each encoding vector has the same dimensionality, d_{model} , as an embedding vector. This allows them to be summed. k represents the position, starting with 0 and going to $L-1$. The highest number that i can be set to is d_{model} divided by 2 since the equations alternate for each element in the embedding. n can be set to any value, but the original paper recommends 10,000. In the image below, the following parameters are used to calculate the positional encoding vectors for a 6-token sequence:

- $n = 10,000$
- $L = 6$
- $d_{model} = 4$

$$\begin{aligned}
 PE_{(k,2i)} &= \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \\
 PE_{(k,2i+1)} &= \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)
 \end{aligned}
 \quad
 \begin{aligned}
 & \overbrace{d_{model} = 4} \\
 & k \left\{ \begin{aligned}
 P(0) &= \left[\sin\left(\frac{0}{10000^{\frac{0}{4}}}\right), \cos\left(\frac{0}{10000^{\frac{0}{4}}}\right), \sin\left(\frac{0}{10000^{\frac{1}{4}}}\right), \cos\left(\frac{0}{10000^{\frac{1}{4}}}\right) \right] \\
 P(1) &= \left[\sin\left(\frac{1}{10000^{\frac{0}{4}}}\right), \cos\left(\frac{1}{10000^{\frac{0}{4}}}\right), \sin\left(\frac{1}{10000^{\frac{1}{4}}}\right), \cos\left(\frac{1}{10000^{\frac{1}{4}}}\right) \right] \\
 P(2) &= \left[\sin\left(\frac{2}{10000^{\frac{0}{4}}}\right), \cos\left(\frac{2}{10000^{\frac{0}{4}}}\right), \sin\left(\frac{2}{10000^{\frac{1}{4}}}\right), \cos\left(\frac{2}{10000^{\frac{1}{4}}}\right) \right] \\
 P(3) &= \left[\sin\left(\frac{3}{10000^{\frac{0}{4}}}\right), \cos\left(\frac{3}{10000^{\frac{0}{4}}}\right), \sin\left(\frac{3}{10000^{\frac{1}{4}}}\right), \cos\left(\frac{3}{10000^{\frac{1}{4}}}\right) \right] \\
 P(4) &= \left[\sin\left(\frac{4}{10000^{\frac{0}{4}}}\right), \cos\left(\frac{4}{10000^{\frac{0}{4}}}\right), \sin\left(\frac{4}{10000^{\frac{1}{4}}}\right), \cos\left(\frac{4}{10000^{\frac{1}{4}}}\right) \right] \\
 P(5) &= \left[\sin\left(\frac{5}{10000^{\frac{0}{4}}}\right), \cos\left(\frac{5}{10000^{\frac{0}{4}}}\right), \sin\left(\frac{5}{10000^{\frac{1}{4}}}\right), \cos\left(\frac{5}{10000^{\frac{1}{4}}}\right) \right]
 \end{aligned} \right. \\
 & \quad \quad \quad \underbrace{\quad \quad \quad}_{i=0} \quad \underbrace{\quad \quad \quad}_{i=0} \quad \underbrace{\quad \quad \quad}_{i=1} \quad \underbrace{\quad \quad \quad}_{i=1}
 \end{aligned}$$

Image by Author

This image shows how the maximum value i is set to is 1, which is used by both sine and cosine. k changes with each row in the embedding matrix, starting with 0 and going to 5, which is the maximum length of 6. Each vector has $d_{model} = 4$ elements.

Basic Implementation

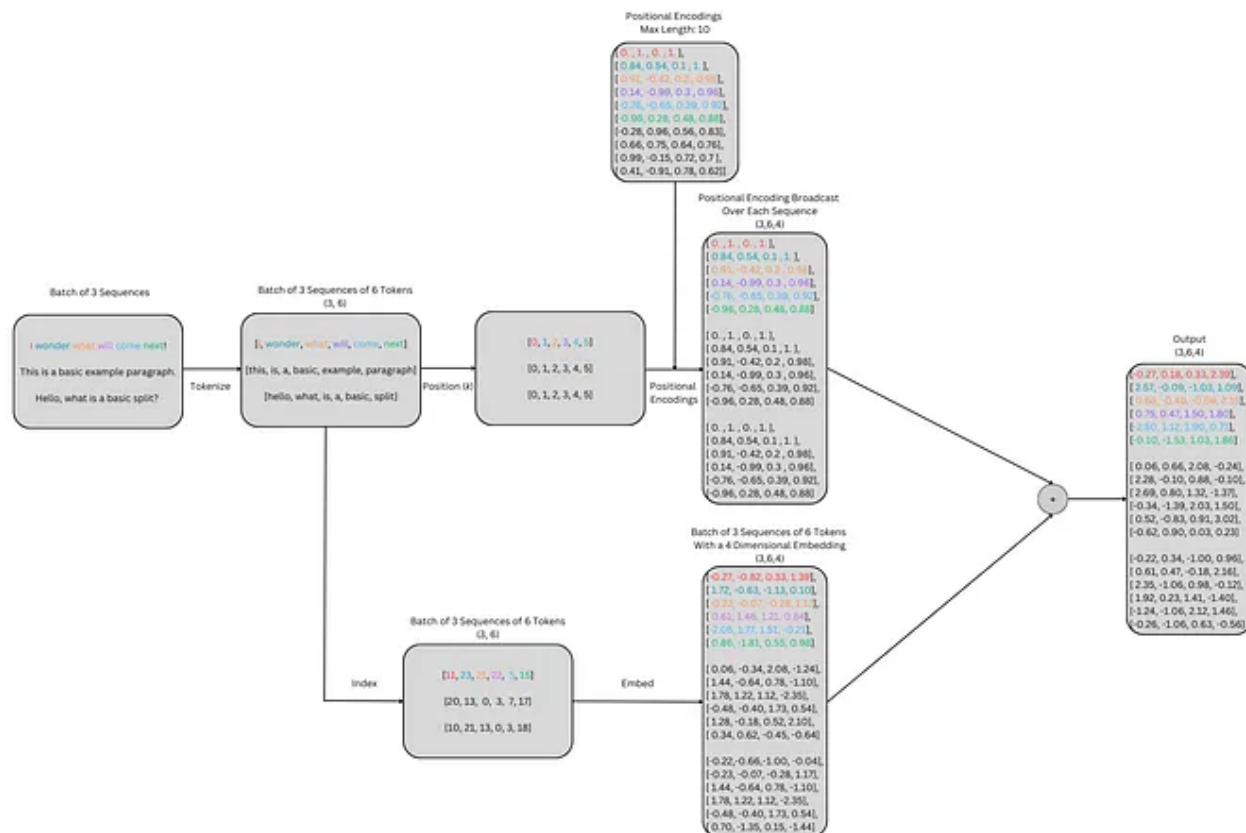


Image by Author

To see how these unique positional encoding vectors work with the embedding vectors, it would be best to work off of the examples from The Embedding Layer article.

This implementation is going to build directly off the one from the previous post. The output below embeds three sequences with a d_{model} of 4.

```
# set the output to 2 decimal places without scientific notation
torch.set_printoptions(precision=2, sci_mode=False)

# tokenize the sequences
tokenized_sequences = [tokenize(seq) for seq in sequences]

# index the sequences
indexed_sequences = [[stoi[word] for word in seq] for seq in tokenized_sequences]
```

```

# convert the sequences to a tensor
tensor_sequences = torch.tensor(indexed_sequences).long()

# vocab size
vocab_size = len(stoi)

# embedding dimensions
d_model = 4

# create the embeddings
lut = nn.Embedding(vocab_size, d_model) # look-up table (lut)

# embed the sequence
embeddings = lut(tensor_sequences)

embeddings

```

```

tensor([[[[-0.27, -0.82,  0.33,  1.39],
          [ 1.72, -0.63, -1.13,  0.10],
          [-0.23, -0.07, -0.28,  1.17],
          [ 0.61,  1.46,  1.21,  0.84],
          [-2.05,  1.77,  1.51, -0.21],
          [ 0.86, -1.81,  0.55,  0.98]],

        [[ 0.06, -0.34,  2.08, -1.24],
          [ 1.44, -0.64,  0.78, -1.10],
          [ 1.78,  1.22,  1.12, -2.35],
          [-0.48, -0.40,  1.73,  0.54],
          [ 1.28, -0.18,  0.52,  2.10],
          [ 0.34,  0.62, -0.45, -0.64]],

        [[-0.22, -0.66, -1.00, -0.04],
          [-0.23, -0.07, -0.28,  1.17],
          [ 1.44, -0.64,  0.78, -1.10],
          [ 1.78,  1.22,  1.12, -2.35],
          [-0.48, -0.40,  1.73,  0.54],
          [ 0.70, -1.35,  0.15, -1.44]]], grad_fn=<EmbeddingBackward0>)

```

The next step is to encode the position of each word in each sequence via positional encodings. The function below follows the definition above. The

only change worth mentioning is that L is notated as *max_length*. It is often set to an extremely large value in the thousands to ensure that almost every sequence can be encoded appropriately. This ensures that the same positional encoding matrix can be used for sequences of varying lengths. It can be sliced to the appropriate length before addition.

```
def gen_pe(max_length, d_model, n):

    # generate an empty matrix for the positional encodings (pe)
    pe = np.zeros(max_length*d_model).reshape(max_length, d_model)

    # for each position
    for k in np.arange(max_length):

        # for each dimension
        for i in np.arange(d_model//2):

            # calculate the internal value for sin and cos
            theta = k / (n ** ((2*i)/d_model))

            # even dims: sin
            pe[k, 2*i] = math.sin(theta)

            # odd dims: cos
            pe[k, 2*i+1] = math.cos(theta)

    return pe

# maximum sequence length
max_length = 10
n = 100
encodings = gen_pe(max_length, d_model, n)
```

The output of the encodings contains 10 position encoding vectors.

```
array([[ 0.      ,  1.      ,  0.      ,  1.      ],
       [ 0.8415,  0.5403,  0.0998,  0.995 ]],
```



```
[ 0.9093, -0.4161, 0.1987, 0.9801],
[ 0.1411, -0.99 , 0.2955, 0.9553],
[-0.7568, -0.6536, 0.3894, 0.9211],
[-0.9589, 0.2837, 0.4794, 0.8776],
[-0.2794, 0.9602, 0.5646, 0.8253],
[ 0.657 , 0.7539, 0.6442, 0.7648],
[ 0.9894, -0.1455, 0.7174, 0.6967],
[ 0.4121, -0.9111, 0.7833, 0.6216]])
```

As mentioned, the *max_length* is set to 10. While this is more than required, it ensures that if another sequence were to have a length of 7, 8, 9, or 10, the same positional encoding matrix could be used. It would just need to be sliced to the appropriate length. Below, the embeddings have a *seq_length* of six, so the encodings can be sliced accordingly.

```
# select the first six tokens
seq_length = embeddings.shape[1]
encodings[:seq_length]
```

```
tensor([[ 0.00,  1.00,  0.00,  1.00],
        [ 0.84,  0.54,  0.10,  1.00],
        [ 0.91, -0.42,  0.20,  0.98],
        [ 0.14, -0.99,  0.30,  0.96],
        [-0.76, -0.65,  0.39,  0.92],
        [-0.96,  0.28,  0.48,  0.88]])
```

Since the sequence lengths are the same for all three sequences, only one positional encoding matrix is needed, and it can be broadcast across all three using PyTorch. The embedded batch in this example has a shape of (3, 6, 4), and the positional encoding has a shape of (10, 4) before it is sliced to (6, 4). This matrix is then broadcast to create the (3, 6, 4) encoding matrix

seen in the image. For more information on broadcasting, read [A Simple Introduction to Broadcasting](#).

This allows the two matrices to be added without any issues.

```
embedded_sequence + encodings[:seq_length] # encodings[:6]
```

When the positional encodings are added to the embeddings, the output is the same as the image at the beginning of the section.

```
tensor([[[[-0.27,  0.18,  0.33,  2.39],
          [ 2.57, -0.09, -1.03,  1.09],
          [ 0.68, -0.49, -0.08,  2.15],
          [ 0.75,  0.47,  1.50,  1.80],
          [-2.80,  1.12,  1.90,  0.71],
          [-0.10, -1.53,  1.03,  1.86]],

        [[ 0.06,  0.66,  2.08, -0.24],
          [ 2.28, -0.10,  0.88, -0.10],
          [ 2.69,  0.80,  1.32, -1.37],
          [-0.34, -1.39,  2.03,  1.50],
          [ 0.52, -0.83,  0.91,  3.02],
          [-0.62,  0.90,  0.03,  0.23]],

        [[-0.22,  0.34, -1.00,  0.96],
          [ 0.61,  0.47, -0.18,  2.16],
          [ 2.35, -1.06,  0.98, -0.12],
          [ 1.92,  0.23,  1.41, -1.40],
          [-1.24, -1.06,  2.12,  1.46],
          [-0.26, -1.06,  0.63, -0.56]]], grad_fn=<AddBackward0>)
```

This output is what would be passed to the next layer of the model, which is the Multi-head Attention that will be covered in the next article.

However, this basic implementation is not efficient due to its use of nested loops, especially if larger values of *d_model* and *max_length* are used. Instead, a more PyTorch-centric approach can be used.

Modifying the Positional Encoding Formula for PyTorch

$$\text{Rule 1: } \log_b (M \cdot N) = \log_b M + \log_b N$$

$$\text{Rule 2: } \log_b \left(\frac{M}{N} \right) = \log_b M - \log_b N$$

$$\text{Rule 3: } \log_b (M^k) = k \cdot \log_b M$$

$$\text{Rule 4: } \log_b (1) = 0$$

$$\text{Rule 5: } \log_b (b) = 1$$

$$\text{Rule 6: } \log_b (b^k) = k$$

$$\text{Rule 7: } b^{\log_b (k)} = k$$

Where:

$b > 0$ but $b \neq 1$, and M , N , and k are real numbers but M and N must be positive!

©chilimath.com

Image by Chili Math

To exploit the capabilities of PyTorch, the original equations, specifically the divisor, need to be modified using logarithmic rules.

The divisor is:

$$\frac{1}{n^{\frac{2i}{d_{model}}}}$$

To modify the divisor, n is brought into the numerator by negating its exponent. Then, rule 7 is used to raise the entire equation to be an exponent of e . Then, rule 3 is used to pull the exponent outside of the \log . This is then simplified to get the result.

$$\frac{1}{n^{\frac{2i}{d_{model}}}} = n^{-\frac{2i}{d_{model}}} = e^{\log(n^{-\frac{2i}{d_{model}}})} = e^{-\frac{2i}{d_{model}} \log(n)} = e^{-\frac{2i \log(n)}{d_{model}}}$$

This is significant because it can be used to generate all the divisors for the positional encodings at once. Below it is apparent that only two divisors are necessary for a 4-dimensional embedding since the divisor only changes every $2i$, where i is the dimension. This repeats across each position:

$$\begin{aligned} P(0) &= \left[\sin\left(\frac{0}{n^{\frac{0}{4}}}\right), \cos\left(\frac{0}{n^{\frac{0}{4}}}\right), \sin\left(\frac{0}{n^{\frac{2}{4}}}\right), \cos\left(\frac{0}{n^{\frac{2}{4}}}\right) \right] \\ P(1) &= \left[\sin\left(\frac{1}{n^{\frac{0}{4}}}\right), \cos\left(\frac{1}{n^{\frac{0}{4}}}\right), \sin\left(\frac{1}{n^{\frac{2}{4}}}\right), \cos\left(\frac{1}{n^{\frac{2}{4}}}\right) \right] \\ P(2) &= \left[\sin\left(\frac{2}{n^{\frac{0}{4}}}\right), \cos\left(\frac{2}{n^{\frac{0}{4}}}\right), \sin\left(\frac{2}{n^{\frac{2}{4}}}\right), \cos\left(\frac{2}{n^{\frac{2}{4}}}\right) \right] \end{aligned}$$

Since only the highest number that i can be set to is d_model divided by 2, the terms can be calculated once:

```
d_model = 4
n = 100

div_term = torch.exp(torch.arange(0, d_model, 2) * -(math.log(n) / d_model))
```

This short snippet of code can be used to generate all of the divisors needed. For this example, d_model is set to 4, and n is set to 100. The output is two divisors:

```
tensor([1.0000, 0.1000])
```

From here, it is possible to take advantage of PyTorch's indexing capabilities to create the entire positional encoding matrix with a few lines of code. The next step is to generate each position from k to $L-1$.

```
max_length = 10

# generate the positions into a column matrix
k = torch.arange(0, max_length).unsqueeze(1)
```

```
tensor([[0],
        [1],
        [2],
        [3],
```

```
[4],
[5],
[6],
[7],
[8],
[9]])
```

With the positions and divisors, the inside of the sin and cosine functions can be easily calculated.

$$\begin{aligned}
 P(0) &= \left[\sin\left(\frac{0}{100^{\frac{0}{4}}}\right), \cos\left(\frac{0}{100^{\frac{0}{4}}}\right), \sin\left(\frac{0}{100^{\frac{2}{4}}}\right), \cos\left(\frac{0}{100^{\frac{2}{4}}}\right) \right] \\
 P(1) &= \left[\sin\left(\frac{1}{100^{\frac{0}{4}}}\right), \cos\left(\frac{1}{100^{\frac{0}{4}}}\right), \sin\left(\frac{1}{100^{\frac{2}{4}}}\right), \cos\left(\frac{1}{100^{\frac{2}{4}}}\right) \right] \\
 P(2) &= \left[\sin\left(\frac{2}{100^{\frac{0}{4}}}\right), \cos\left(\frac{2}{100^{\frac{0}{4}}}\right), \sin\left(\frac{2}{100^{\frac{2}{4}}}\right), \cos\left(\frac{2}{100^{\frac{2}{4}}}\right) \right] \\
 &\vdots \\
 P(9) &= \left[\sin\left(\frac{9}{100^{\frac{0}{4}}}\right), \cos\left(\frac{9}{100^{\frac{0}{4}}}\right), \sin\left(\frac{9}{100^{\frac{2}{4}}}\right), \cos\left(\frac{9}{100^{\frac{2}{4}}}\right) \right]
 \end{aligned}$$

By multiplying k and *div_term*, the inputs can be calculated for every position. PyTorch will automatically broadcast the matrices to allow for multiplication. Note that this is the Hadamard product and not matrix multiplication because the corresponding elements will be multiplied by each other:

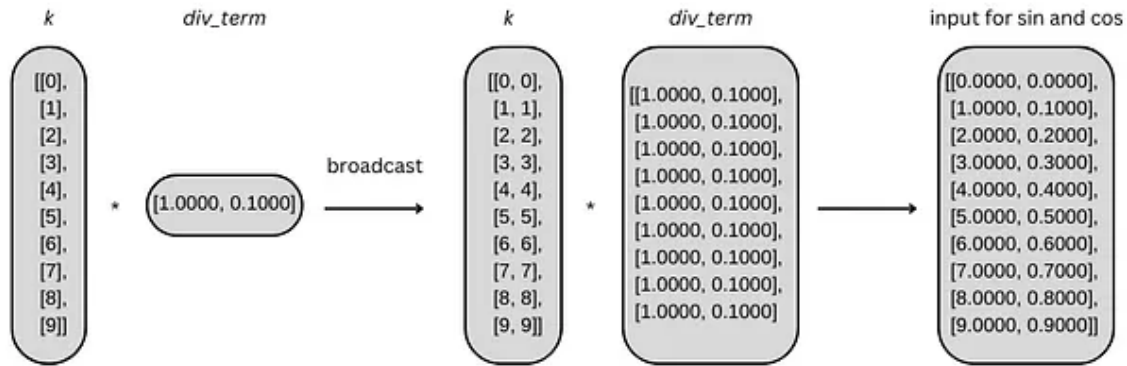


Image by Author

```
k*div_term
```

The output for this calculation can be seen in the image above. All that is left to do is plug the input into the cos and sin functions and to save these in a matrix appropriately.

This can be started by creating an empty matrix of the appropriate size:

```
# generate an empty tensor
pe = torch.zeros(max_length, d_model)
```

```
tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]])
```



```
[0., 0., 0., 0.],
[0., 0., 0., 0.],
[0., 0., 0., 0.],
[0., 0., 0., 0.]])
```

Now, the even columns, which are sin, can be selected with `pe[:, 0::2]`. This tells PyTorch to select every row and each even column. The same can be done for the odd columns, which are cos: `pe[:, 1::2]`. Once again, this tells PyTorch to select every row and each odd column. Since the result of `k*div_term` has all the necessary inputs stored in it, it can be used to calculate each odd and even column.

```
# set the odd values (columns 1 and 3)
pe[:, 0::2] = torch.sin(k * div_term)

# set the even values (columns 2 and 4)
pe[:, 1::2] = torch.cos(k * div_term)

# add a dimension for broadcasting across sequences: optional
pe = pe.unsqueeze(0)
```

```
tensor([[[[ 0.00,  1.00,  0.00,  1.00],
           [ 0.84,  0.54,  0.10,  1.00],
           [ 0.91, -0.42,  0.20,  0.98],
           [ 0.14, -0.99,  0.30,  0.96],
           [-0.76, -0.65,  0.39,  0.92],
           [-0.96,  0.28,  0.48,  0.88],
           [-0.28,  0.96,  0.56,  0.83],
           [ 0.66,  0.75,  0.64,  0.76],
           [ 0.99, -0.15,  0.72,  0.70],
           [ 0.41, -0.91,  0.78,  0.62]]]])
```

These are identical to the values acquired with the nested for-loop. To recap, here is all of the code together:

```
max_length = 10
d_model = 4
n = 100

def gen_pe(max_length, d_model, n):
    # calculate the div_term
    div_term = torch.exp(torch.arange(0, d_model, 2) * -(math.log(n) / d_model))

    # generate the positions into a column matrix
    k = torch.arange(0, max_length).unsqueeze(1)

    # generate an empty tensor
    pe = torch.zeros(max_length, d_model)

    # set the even values
    pe[:, 0::2] = torch.sin(k * div_term)

    # set the odd values
    pe[:, 1::2] = torch.cos(k * div_term)

    # add a dimension
    pe = pe.unsqueeze(0)

    # the output has a shape of (1, max_length, d_model)
    return pe

gen_pe(max_length, d_model, n)
```

Although its more complex, this is the implementation that PyTorch uses due to its enhanced performance for machine learning.

Positional Encoding in Transformers

Now that all the hard work is out of the way, the implementation is straightforward. It is derived from [The Annotated Transformer](#) and [PyTorch](#). Note that the default value for n is 10,000, and the default *max_length* is 5,000.

This implementation also incorporates dropout, which randomly zeroes out some of the elements of its input with the given probability, p . This helps with regularization and prevents neurons from co-adapting (overrelying on each other). The outputs are also scaled by a factor of $\frac{1}{(1-p)}$. Rather than going in-depth on it in this article. Please see the article on the [Dropout Layer](#) for more information. It would be best to become comfortable with it now before moving to the rest of the transformer model because it is in almost every other layer.

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model: int, dropout: float = 0.1, max_length: int = 5000)
        """
        Args:
            d_model:        dimension of embeddings
            dropout:        randomly zeroes-out some of the input
            max_length:     max sequence length
        """
        # inherit from Module
        super().__init__()

        # initialize dropout
        self.dropout = nn.Dropout(p=dropout)

        # create tensor of 0s
        pe = torch.zeros(max_length, d_model)

        # create position column
        k = torch.arange(0, max_length).unsqueeze(1)

        # calc divisor for positional encoding
        div_term = torch.exp(
            torch.arange(0, d_model, 2) * -(math.log(10000.0) / d_model)
        )
```

```

# calc sine on even indices
pe[:, 0::2] = torch.sin(k * div_term)

# calc cosine on odd indices
pe[:, 1::2] = torch.cos(k * div_term)

# add dimension
pe = pe.unsqueeze(0)

# buffers are saved in state_dict but not trained by the optimizer
self.register_buffer("pe", pe)

def forward(self, x: Tensor):
    """
    Args:
        x:          embeddings (batch_size, seq_length, d_model)

    Returns:
        embeddings + positional encodings (batch_size, seq_length, d_model)
    """
    # add positional encoding to the embeddings
    x = x + self.pe[:, : x.size(1)].requires_grad_(False)

    # perform dropout
    return self.dropout(x)

```

Forward Pass

To perform the forward pass, the same embedded sequences from earlier can be used.

embeddings

```

tensor([[[[-0.27, -0.82,  0.33,  1.39],
          [ 1.72, -0.63, -1.13,  0.10],
          [-0.23, -0.07, -0.28,  1.17],
          [ 0.61,  1.46,  1.21,  0.84],

```

```

[-2.05,  1.77,  1.51, -0.21],
[ 0.86, -1.81,  0.55,  0.98]],

[[ 0.06, -0.34,  2.08, -1.24],
[ 1.44, -0.64,  0.78, -1.10],
[ 1.78,  1.22,  1.12, -2.35],
[-0.48, -0.40,  1.73,  0.54],
[ 1.28, -0.18,  0.52,  2.10],
[ 0.34,  0.62, -0.45, -0.64]],

[[-0.22, -0.66, -1.00, -0.04],
[-0.23, -0.07, -0.28,  1.17],
[ 1.44, -0.64,  0.78, -1.10],
[ 1.78,  1.22,  1.12, -2.35],
[-0.48, -0.40,  1.73,  0.54],
[ 0.70, -1.35,  0.15, -1.44]]], grad_fn=<EmbeddingBackward0>)

```

With the sequences embedded, the positional encoding matrix can be created. The dropout is set to 0.0 to easily see the addition between the embeddings and the positional encodings. The values are different than the from-scratch implementations because n has a default value of 10,000 instead of 100.

```

d_model = 4
max_length = 10
dropout = 0.0

# create the positional encoding matrix
pe = PositionalEncoding(d_model, dropout, max_length)

# preview the values
pe.state_dict()

```

```

OrderedDict([('pe',
              tensor([[ 0.00,  1.00,  0.00,  1.00],
                      [ 0.84,  0.54,  0.01,  1.00],
                      [ 0.91, -0.42,  0.02,  1.00],
                      [ 0.14, -0.99,  0.03,  1.00],

```

```
[ -0.76, -0.65,  0.04,  1.00],
[ -0.96,  0.28,  0.05,  1.00],
[ -0.28,  0.96,  0.06,  1.00],
[  0.66,  0.75,  0.07,  1.00],
[  0.99, -0.15,  0.08,  1.00],
[  0.41, -0.91,  0.09,  1.00]]]))))
```

Before adding them, the sequences have a shape of $(batch_size, seq_length, d_model)$, which is $(3, 6, 4)$. The positional encodings are of the same size once they are sliced and broadcast, so the output of the forward pass has a size of $(batch_size, seq_length, d_model)$, which is still $(3, 6, 4)$. This represents 3 sequences of 6 tokens embedded in a 4-dimensional space with positional encodings to indicate their location in the sequence.

```
pe(embeddings)
```

```
tensor([[[[-0.27,  0.18,  0.33,  2.39],
          [ 2.57, -0.09, -1.12,  1.10],
          [ 0.68, -0.49, -0.26,  2.17],
          [ 0.75,  0.47,  1.24,  1.84],
          [-2.80,  1.12,  1.55,  0.79],
          [-0.10, -1.53,  0.60,  1.98]],

         [[ 0.06,  0.66,  2.08, -0.24],
          [ 2.28, -0.10,  0.79, -0.10],
          [ 2.69,  0.80,  1.14, -1.35],
          [-0.34, -1.39,  1.76,  1.54],
          [ 0.52, -0.83,  0.56,  3.10],
          [-0.62,  0.90, -0.40,  0.35]],

         [[-0.22,  0.34, -1.00,  0.96],
          [ 0.61,  0.47, -0.27,  2.17],
          [ 2.35, -1.06,  0.80, -0.10],
          [ 1.92,  0.23,  1.15, -1.35],
          [-1.24, -1.06,  1.77,  1.54],
          [-0.26, -1.06,  0.20, -0.44]]], grad_fn=<AddBackward0>)
```

The next article in the series is the [Multi-Head Attention Layer](#).

Please don't forget to like and follow for more! :)

References

1. [Dropout Layer](#)
2. [Positional Encoding Overview](#)
3. [PyTorch's Positional Encoding Implementation](#)
4. [The Annotated Transformer](#)
5. [Transformer's Positional Encoding](#)

Appendix

Visualizing the Uniqueness of Positional Encodings

Before concluding, it would be beneficial to verify the uniqueness of the positional encodings and see how they work with larger sequences.

Using matplotlib, the vectors can be easily compared to each other.

```
def visualize_pe(max_length, d_model, n):  
    plt.imshow(gen_pe(max_length, d_model, n), aspect="auto")  
    plt.title("Positional Encoding")  
    plt.xlabel("Encoding Dimension")  
    plt.ylabel("Position Index")
```

```

# set the tick marks for the axes
if d_model < 10:
    plt.xticks(torch.arange(0, d_model))
if max_length < 20:
    plt.yticks(torch.arange(max_length-1, -1, -1))

plt.colorbar()
plt.show()

# plot the encodings
max_length = 10
d_model = 4
n = 100

visualize_pe(max_length, d_model, n)

```

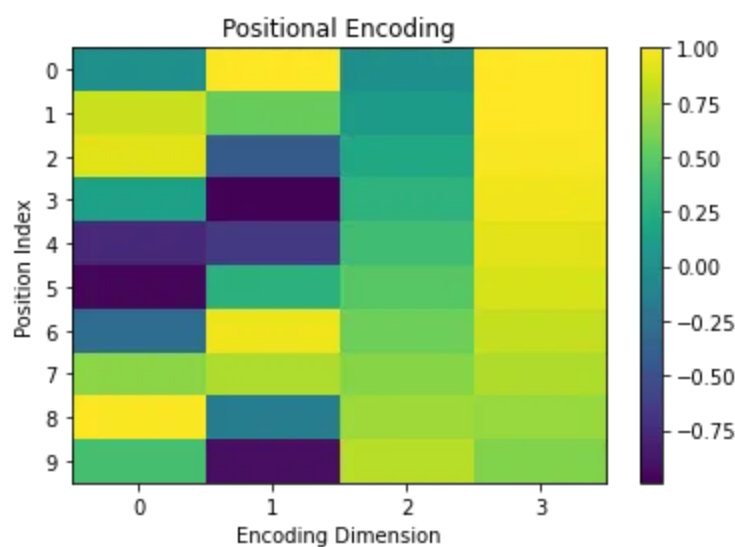


Image by Author

Each row represents a positional encoding vector, and each column represents the corresponding dimension it will be added to when it is combined with the embeddings.

This can also be seen with larger values of n , d_model , and max_length :


```
# plot the encodings
max_length = 1000
d_model = 512
n = 10000

visualize_pe(max_length, d_model, n)
```

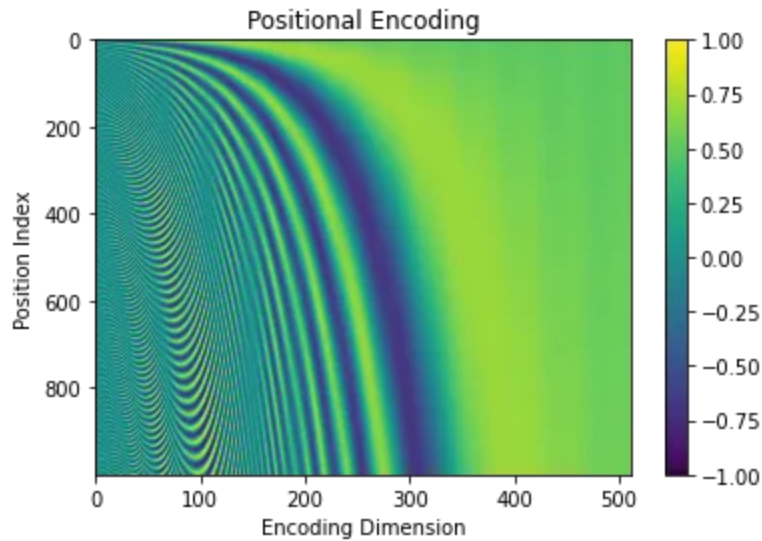


Image by Author

Positional Encoding

Transformers

Machine Learning

NLP

Artificial Intelligence

More from the list: "NLP"

Curated by Himanshu Birla



Jon Gi... in Towards Data ...

Characteristics of Word Embeddings

★ · 11 min read · Sep 4, 2021



Jon Gi... in Towards Data ...

The Word2vec Hyperparameters

★ · 6 min read · Sep 3, 2021



Jon Gi... in

The Word2vec

★ · 15 min read

[View list](#)**Written by Hunter Phillips**

219 Followers

Machine Learning Engineer and Data Scientist

Following

**More from Hunter Phillips**

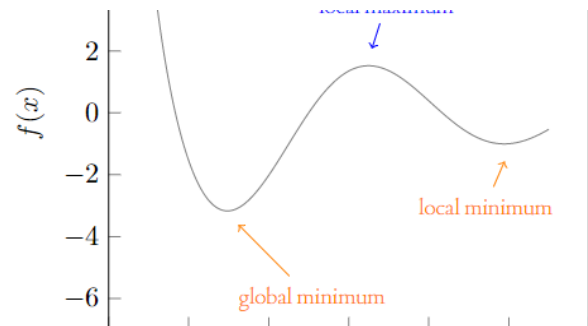
$$\begin{bmatrix} \begin{bmatrix} x_{1,0} & x_{1,1} & x_{1,2} \end{bmatrix} \\ \begin{bmatrix} x_{2,0} & x_{2,1} & x_{2,2} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \vec{x}_1 \\ \vec{x}_2 \end{bmatrix} = \begin{bmatrix} X_0 \\ X_1 \\ X_2 \end{bmatrix} = \mathcal{X}$$



Hunter Phillips

A Simple Introduction to Tensors

A tensor is a generalization of vectors and matrices to n dimensions. Understanding ho...



Hunter Phillips

A Simple Introduction to Gradient Descent


Gradient descent is one of the most common optimization algorithms in machine learning....

11 min read · May 10

 273  5




 Hunter Phillips

What is an RDD in PySpark?

This article covers the basic uses of resilient distributed datasets in PySpark. It includes...

7 min read · Jun 10

 100 


 

10 min read · May 18

 108 



 Hunter Phillips

An Introduction to Machine Learning in Python: The Normal...

The Normal Equation is a closed-form solution for minimizing a cost function and...

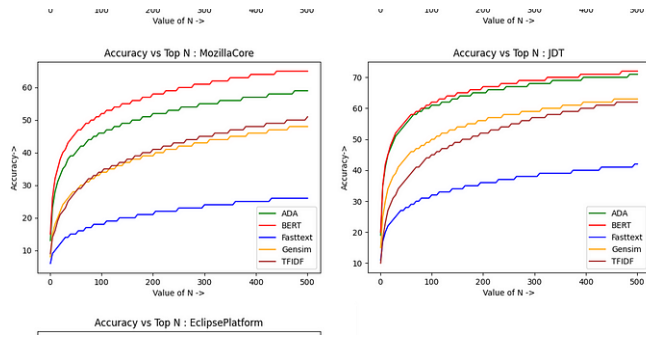
4 min read · May 22

 175 

See all from Hunter Phillips

Recommended from Medium



Avinash Patil

Embeddings: BERT better than ChatGPT4?

In this study, we compared the effectiveness of semantic textual similarity methods for...

4 min read · Sep 19



3



1



Zain ul Abideen

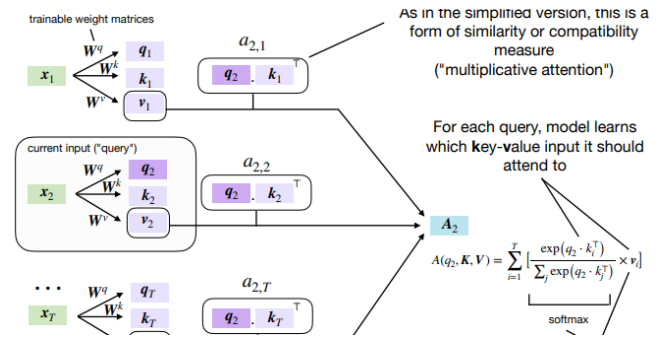
Attention Is All You Need: The Core Idea of the Transformer

An overview of the Transformer model and its key components.

6 min read · Jun 26



144

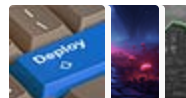


Lists



Natural Language Processing

669 stories · 283 saves



Predictive Modeling w/ Python

20 stories · 452 saves



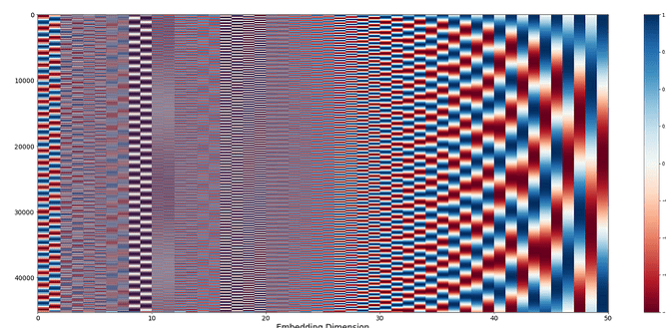
AI Regulation

6 stories · 138 saves



ChatGPT prompts

24 stories · 459 saves





Attention X

Unlocking the Magic of Self-Attention with Math & PyTorch

Welcome to the wondrous world of Self-Attention, a pivotal concept within the realm...

4 min read · Jun 19



7



...



Eugene Ku

Transformer Architecture (Part 1—Positional Encoding)

Nowadays, arguably the most popular and influential model behind the hypes of deep...

4 min read · Aug 22



15



...



Haifeng Li

A Tutorial on LLM

Generative artificial intelligence (GenAI), especially ChatGPT, captures everyone's...

15 min read · Sep 14



372



...



David Shapiro

A Pro's Guide to Finetuning LLMs

Large language models (LLMs) like GPT-3 and Llama have shown immense promise for...

12 min read · Sep 23



283



6



...

See more recommendations