The Encoder

Hunter Phillips · Following
14 min read · May 10

This is the sixth article in The Implemented Transformer series. The Encoder is the first half of the transformer architecture, and it includes all the previous layers.

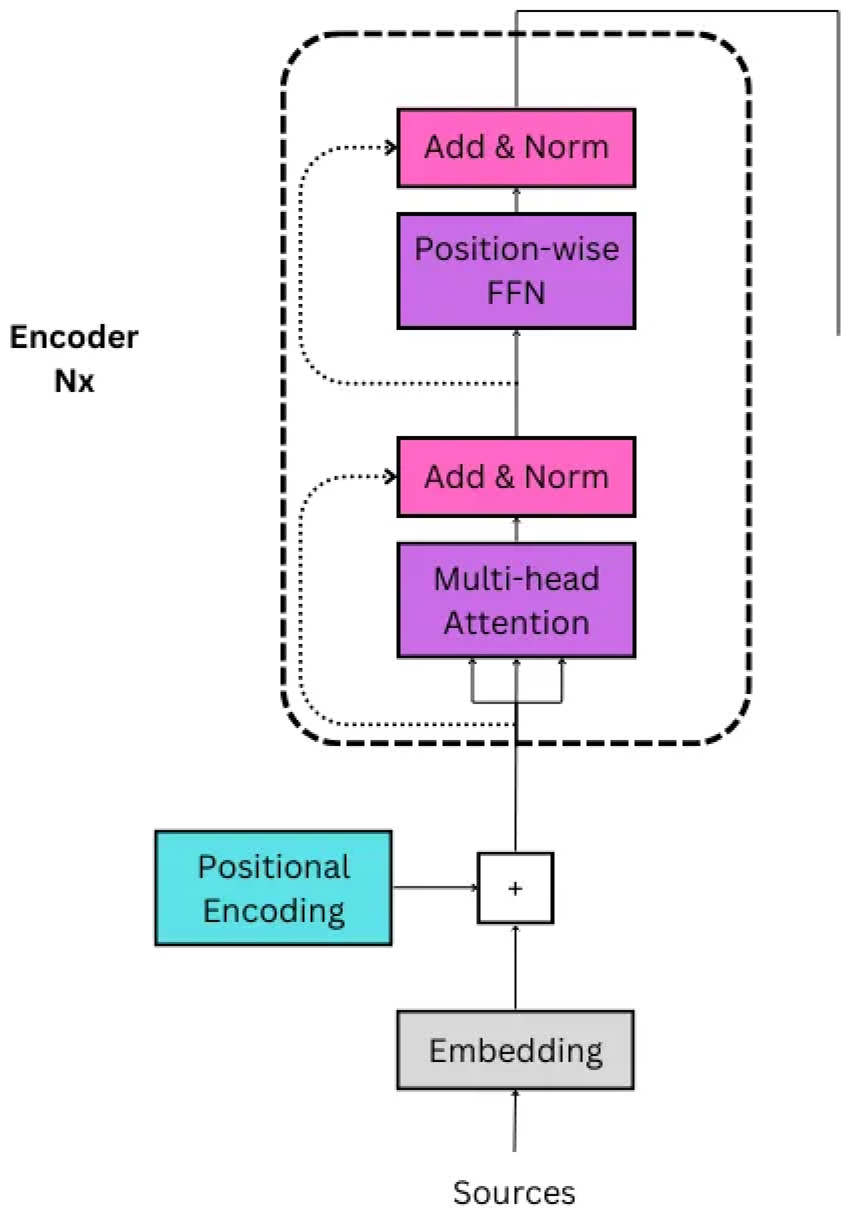


Image by Author

Background

The encoder layer is a wrapper for the sublayers mentioned in the previous articles. It takes the positionally embedded sequences and passes them through the multi-head attention mechanism and the position-wise FFN. After each of these sublayers, it performs residual addition and layer normalization. According to Jindřich on Stack Exchange:

> *The reason for having the residual connection in Transformer is more technical than motivated by the architecture design.*
>
> *Residual connections mainly help mitigate the vanishing gradient problem. During the back-propagation, the signal gets multiplied by the derivative of the activation function. In the case of ReLU, it means that in approximately half of the cases, the gradient is zero. Without the residual connections, a large part of the training signal would get lost during back-propagation. Residual connections reduce effect because summation is linear with respect to derivative, so each residual block also gets a signal that is not affected by the vanishing gradient. The summation operations of residual connections form a path in the computation graphs where the gradient does not get lost.*
>
> *Another effect of residual connections is that the information stays local in the Transformer layer stack. The self-attention mechanism allows an arbitrary information flow in the network and thus arbitrary permuting the input tokens. The residual connections, however, always "remind" the representation of what the original state was. To some extent, the residual connections give a guarantee that contextual representations of the input tokens really represent the tokens.*

Encoder Layer in Transformers

As mentioned above, the encoder layer is nothing more than a wrapper for the sublayers. It implements multi-head attention, a normalization layer with residual addition, a position-wise feed-forward network, and another layer normalization with residual addition.

Please note that ***nn.LayerNorm*** is used rather than the from-scratch implementation from the last article. Either is acceptable, but PyTorch's implementation is used for simplicity.

```
class EncoderLayer(nn.Module):
  def __init__(self, d_model: int, n_heads: int, d_ffn: int, dropout: float):
    """
    Args:
        d_model:      dimension of embeddings
        n_heads:      number of heads
        d_ffn:        dimension of feed-forward network
        dropout:      probability of dropout occurring
    """
    super().__init__()
    # multi-head attention sublayer
    self.attention = MultiHeadAttention(d_model, n_heads, dropout)
    # layer norm for multi-head attention
    self.attn_layer_norm = nn.LayerNorm(d_model)

    # position-wise feed-forward network
    self.positionwise_ffn = PositionwiseFeedForward(d_model, d_ffn, dropout)
    # layer norm for position-wise ffn
    self.ffn_layer_norm = nn.LayerNorm(d_model)

    self.dropout = nn.Dropout(dropout)

  def forward(self, src: Tensor, src_mask: Tensor):
    """
    Args:
        src:          positionally embedded sequences   (batch_size, seq_length,
        src_mask:     mask for the sequences            (batch_size, 1, 1, seq_l
    Returns:
        src:          sequences after self-attention    (batch_size, seq_length,
    """
    # pass embeddings through multi-head attention
    _src, attn_probs = self.attention(src, src, src, src_mask)

    # residual add and norm
    src = self.attn_layer_norm(src + self.dropout(_src))
```

```
        # position-wise feed-forward network
        _src = self.positionwise_ffn(src)

        # residual add and norm
        src = self.ffn_layer_norm(src + self.dropout(_src))

        return src, attn_probs
```

Encoder Stack

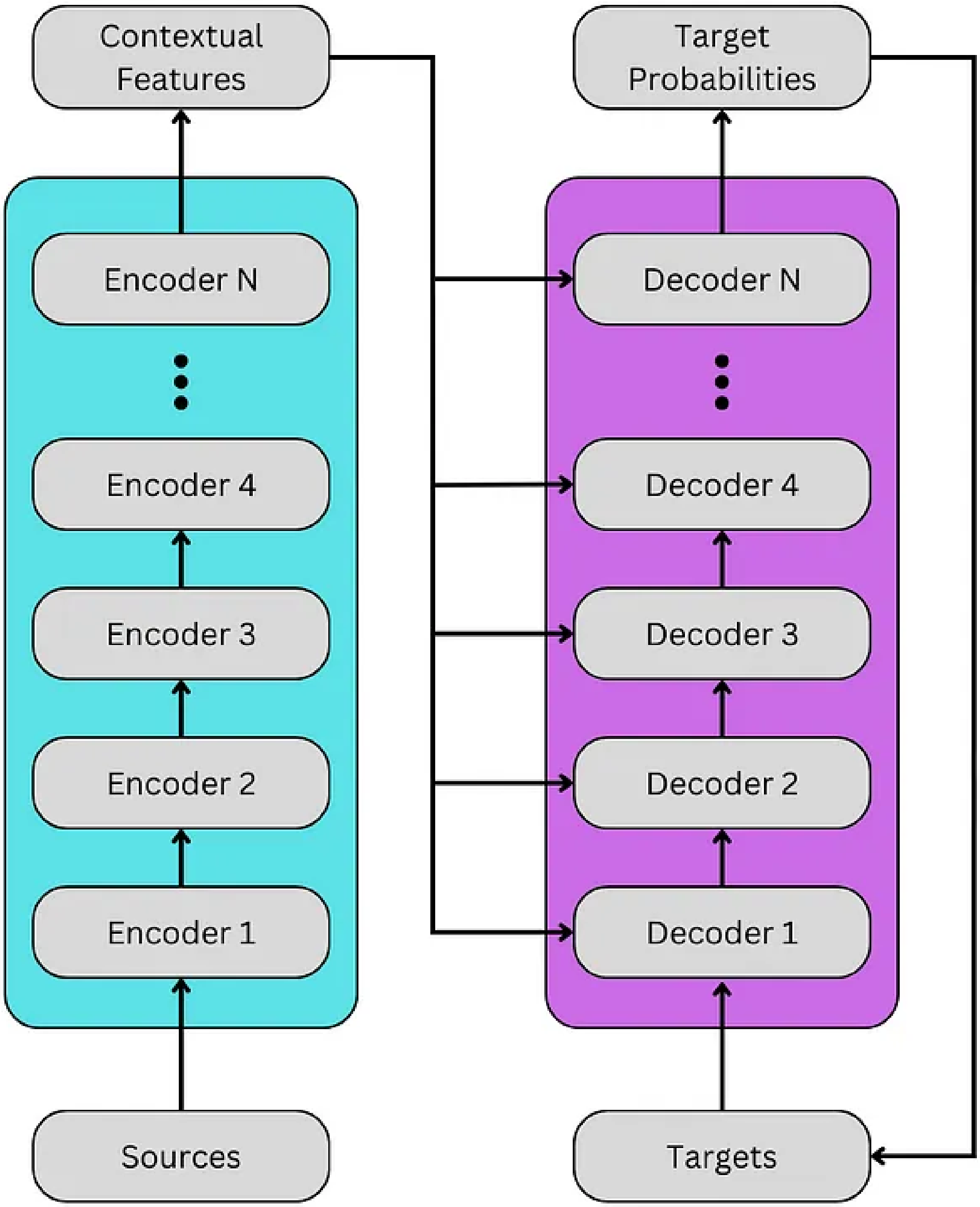


Image by Author

To exploit the benefits of the multi-head attention sublayer, input tokens are passed through a stack of encoder layers at a time before being passed to the decoder. This is notated as *Nx* in the image at the beginning of the article, and the above image shows how these stacked encoders pass their output to the decoder layers, which will be discussed in the next article.

The attention probabilities can be accessed via ***encoder.attn_probs*** after a forward pass.

```
class Encoder(nn.Module):
  def __init__(self, d_model: int, n_layers: int,
               n_heads: int, d_ffn: int, dropout: float = 0.1):
    """
    Args:
        d_model:      dimension of embeddings
        n_layers:     number of encoder layers
        n_heads:      number of heads
        d_ffn:        dimension of feed-forward network
        dropout:      probability of dropout occurring
    """
    super().__init__()

    # create n_layers encoders
    self.layers = nn.ModuleList([EncoderLayer(d_model, n_heads, d_ffn, dropout)
                                 for layer in range(n_layers)])

    self.dropout = nn.Dropout(dropout)

  def forward(self, src: Tensor, src_mask: Tensor):
    """
    Args:
        src:          embedded sequences                (batch_size, seq_length,
        src_mask:     mask for the sequences            (batch_size, 1, 1, seq_l

    Returns:
        src:          sequences after self-attention    (batch_size, seq_length,
    """

    # pass the sequences through each encoder
    for layer in self.layers:
      src, attn_probs = layer(src, src_mask)

    self.attn_probs = attn_probs
```

```
    return src
```

Forward Pass

The example below shows a forward pass using three sequences of equivalent length and no source mask. These sequences are embedded, positionally encoded, and then passed through the multi-head attention mechanism and FFN as well as their residual addition and layer norms. They depend on the functions from the previous articles.

The input to the encoder has a shape of ***(batch_size, seq_length)***, and the output has a shape of ***(batch_size, seq_length, d_model)***.

```
torch.set_printoptions(precision=2, sci_mode=False)

# convert the sequences to integers
sequences = ["I wonder what will come next!",
             "This is a basic example paragraph.",
             "Hello, what is a basic split?"]

# 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()

# parameters
vocab_size = len(stoi)
d_model = 8
d_ffn = d_model*4 # 32
```

```
n_heads = 4
n_layers = 4
dropout = 0.1

# create the embeddings
lut = Embeddings(vocab_size, d_model) # look-up table (lut)

# create the positional encodings
pe = PositionalEncoding(d_model=d_model, dropout=0.1, max_length=10)

# embed the sequence
embeddings = lut(tensor_sequences)

# positionally encode the sequences
X = pe(embeddings)

# initialize encoder
encoder = Encoder(d_model, n_layers, n_heads,
                  d_ffn, dropout)

# pass through encoder
encoder(src=X, src_mask=None)
```

```
tensor([[[-0.61, -0.22,  1.25,  1.36,  0.33, -0.29, -1.99,  0.18],
         [-0.85,  0.20, -0.11,  1.91, -0.71, -1.51,  0.88,  0.18],
         [-0.15, -0.13,  1.31,  0.65,  0.92, -1.56, -1.53,  0.49],
         [-0.73, -1.20,  1.11,  1.77, -0.12,  0.05, -1.22,  0.36],
         [ 0.19, -0.93,  0.67,  0.14, -1.34, -0.26, -0.57,  2.11],
         [-1.19,  0.73,  0.65,  1.37,  1.04, -1.38, -0.69, -0.53]],

        [[-0.67,  0.09,  0.02,  2.36, -0.82, -1.14,  0.12,  0.04],
         [-0.98,  0.36,  1.26,  1.80, -0.37, -0.23, -1.23, -0.60],
         [-0.95,  0.91,  0.83,  1.32, -0.35, -1.92,  0.08,  0.08],
         [-0.71, -1.79,  1.49,  0.89, -0.21, -0.70,  0.16,  0.86],
         [-0.98,  0.65,  0.94,  1.89, -0.43, -0.89, -1.08, -0.09],
         [-2.00, -0.69,  0.66,  1.74, -0.02,  0.01,  0.10,  0.21]],

        [[ 1.32, -0.05,  0.47,  0.81, -1.26, -0.56, -1.63,  0.90],
         [-0.25, -0.60, -0.14,  2.03,  0.34, -1.37, -0.85,  0.84],
         [-1.28, -0.44,  0.49,  1.99, -0.25,  0.55, -1.25,  0.18],
         [-1.43,  1.06,  1.09,  1.41, -0.42, -0.92, -0.79,  0.01],
         [-0.48, -1.37,  0.91, -0.11,  0.73, -1.30, -0.06,  1.68],
         [ 0.13, -1.64,  1.13,  1.33, -0.97, -0.78, -0.05,  0.84]]],
       grad_fn=<NativeLayerNormBackward0>)
```

The attention probabilities can be viewed as well; they have a shape of ***(batch_size, n_heads, Q_length, K_length)***. These values can be easily viewed using the ***display_attention*** function, which can be seen in the appendix; this function was updated from the previous article. Darker colors are lower probabilities, with black being 0, and lighter colors are higher probabilities.

```
# sequence 0
display_attention(tokenized_sequences[0], tokenized_sequences[0], encoder.attn_p
```

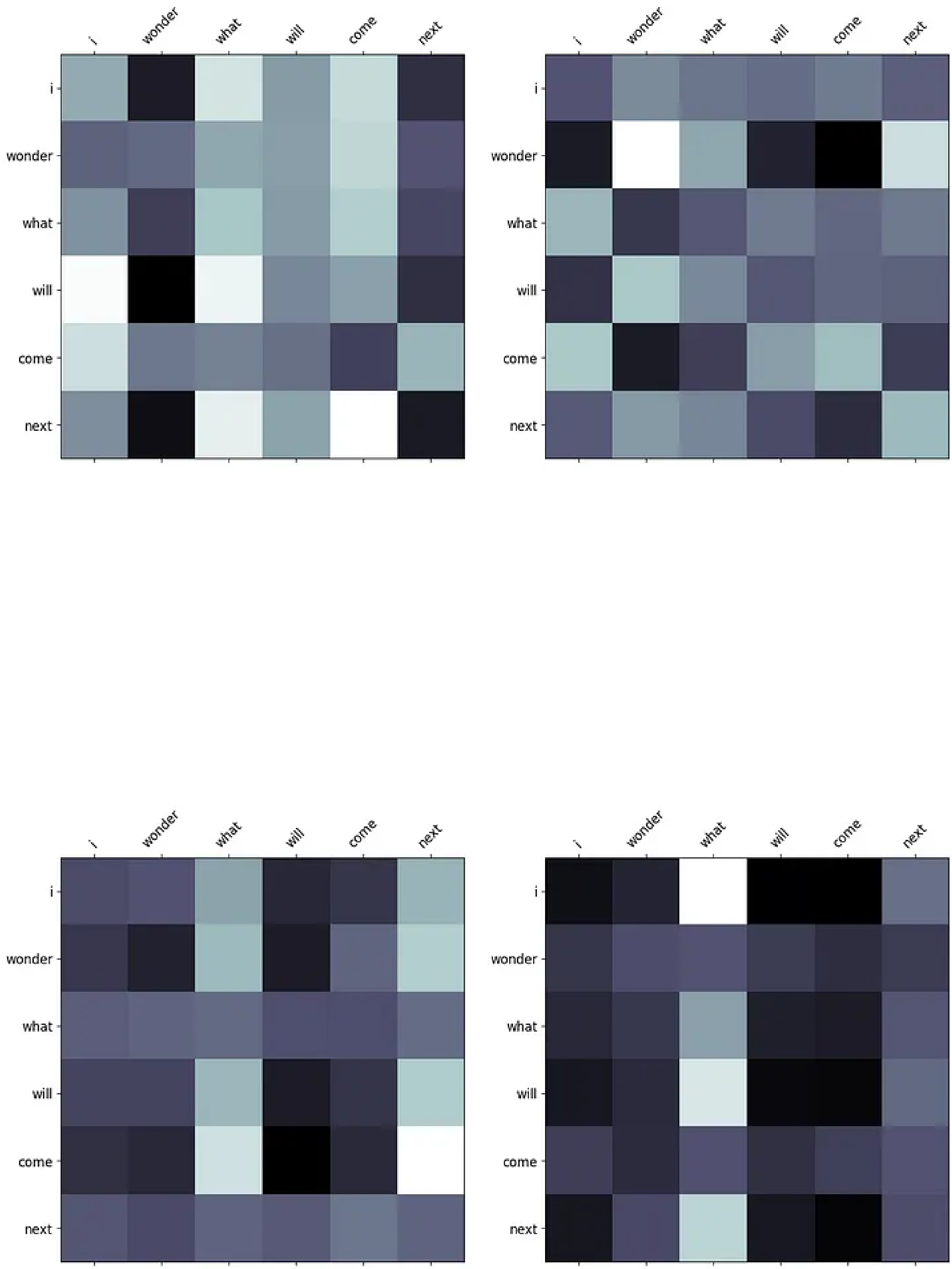


Image by Author

Why Mask?

Padding

In the example above, the ***src_mask*** is set to ***None***. As mentioned in the third article, an optional mask can be passed through the multi-head attention layer. For the encoder, this mask is often created based on the padding of the sequences. The three sequences used in this article are all of length 6. However, it is more likely that sequences of varying lengths will be in a batch. Padding is added to sequences to ensure all the sequences in a batch have the same length. When this occurs, the model does not need to pay attention to padding tokens. A mask vector is created for each sequence to reflect the values that should be attended to.

This mask has a shape of ***(batch_size, 1, 1, seq_length)***. This is broadcast across each head's representation of a sequence.

For example, the three sequences below have different lengths:

- "What will come next?" → ***[21, 22, 5, 15]***
- "This is a basic paragraph." → ***[20, 13, 0, 3, 17]***
- "A basic split will come next!" → ***[0, 3, 18, 22, 5, 15]***

To be used in a tensor, they must be the same length, so padding must be added. This can be done using a simple function with ***pad*** from ***torch.nn.functional***. It allows every input to be padded to the same length. It requires:

- *seq*: a sequence
- *(0, pad_to_add)*: a tuple indicating the dimension to pad and by how much
- *value=pad_idx*: a value to use for the padding, which is normally an integer

This function can be used in a for-loop to pad a batch of sequences to the same length, which can be seen below. These padded sequences can be used as input to the encoder.

```
from torch.nn.functional import pad

def pad_seq(seq: Tensor, max_length: int = 10, pad_idx: int = 0):
  pad_to_add = max_length - len(seq) # amount of padding to add

  return pad(seq,(0, pad_to_add), value=pad_idx,)

sequences = ['What will come next?',
             'This is a basic paragraph.',
             'A basic split will come next!']

# 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

max_length = 8
pad_idx = len(stoi) # 24

padded_seqs = []

for seq in seqs:
  # pad each sequence
  padded_seqs.append(pad_seq(torch.Tensor(seq), max_length, pad_idx))

# create a tensor from the padded sequences
```

```
tensor_sequences = torch.stack(padded_seqs).long()

tensor_sequences
```

```
tensor([[21, 22,  5, 15, 24, 24, 24, 24],
        [20, 13,  0,  3, 17, 24, 24, 24],
        [ 0,  3, 18, 22,  5, 15, 24, 24]])
```

The padding token is appended to each sequence as many times as necessary to acquire the maximum length. Since the first sequence has a length a four, four padding tokens have to be added. The second sequence requires three padding tokens, and the last requires two.

When these sequences are passed through the encoder without a source mask, the attention layers consider each padded token as part of the probability distribution, which can be seen below.

```
torch.set_printoptions(precision=2, sci_mode=False)

# parameters
vocab_size = len(stoi) + 1 # add one for the padding token
d_model = 8
d_ffn = d_model*4 # 32
n_heads = 4
n_layers = 4
dropout = 0.1

# create the embeddings
lut = Embeddings(vocab_size, d_model) # look-up table (lut)

# create the positional encodings
pe = PositionalEncoding(d_model=d_model, dropout=0.1, max_length=10)

# embed the sequence
embeddings = lut(tensor_sequences)
```

```
# positionally encode the sequences
X = pe(embeddings)

# initialize encoder
encoder = Encoder(d_model, n_layers, n_heads,
                  d_ffn, dropout)

# pass through encoder
encoder(src=X, src_mask=None)

# probabilities for sequence 0
encoder.attn_probs[0]
```

```
tensor([[[[0.16, 0.12, 0.14, 0.12, 0.10, 0.08, 0.12, 0.16],
          [0.15, 0.12, 0.12, 0.13, 0.11, 0.10, 0.13, 0.15],
          [0.17, 0.12, 0.10, 0.13, 0.12, 0.09, 0.13, 0.15],
          [0.15, 0.12, 0.09, 0.13, 0.13, 0.11, 0.13, 0.14],
          [0.16, 0.12, 0.11, 0.13, 0.12, 0.10, 0.13, 0.15],
          [0.15, 0.12, 0.10, 0.13, 0.13, 0.11, 0.13, 0.14],
          [0.16, 0.12, 0.11, 0.13, 0.11, 0.09, 0.13, 0.15],
          [0.16, 0.12, 0.11, 0.13, 0.12, 0.09, 0.13, 0.15]],

         [[0.19, 0.10, 0.08, 0.13, 0.13, 0.10, 0.13, 0.15],
          [0.15, 0.12, 0.09, 0.13, 0.13, 0.12, 0.13, 0.14],
          [0.19, 0.10, 0.07, 0.13, 0.13, 0.10, 0.13, 0.15],
          [0.22, 0.09, 0.08, 0.13, 0.13, 0.09, 0.12, 0.15],
          [0.17, 0.10, 0.10, 0.13, 0.13, 0.10, 0.13, 0.14],
          [0.14, 0.12, 0.10, 0.13, 0.13, 0.12, 0.13, 0.13],
          [0.17, 0.10, 0.09, 0.13, 0.13, 0.10, 0.13, 0.14],
          [0.23, 0.08, 0.07, 0.13, 0.13, 0.08, 0.12, 0.15]],

         [[0.08, 0.09, 0.15, 0.18, 0.13, 0.13, 0.11, 0.14],
          [0.07, 0.05, 0.25, 0.22, 0.11, 0.10, 0.09, 0.12],
          [0.16, 0.16, 0.11, 0.09, 0.12, 0.12, 0.13, 0.11],
          [0.05, 0.05, 0.15, 0.26, 0.12, 0.13, 0.09, 0.15],
          [0.04, 0.03, 0.21, 0.29, 0.11, 0.11, 0.08, 0.13],
          [0.04, 0.03, 0.28, 0.28, 0.10, 0.09, 0.07, 0.11],
          [0.04, 0.04, 0.21, 0.28, 0.11, 0.11, 0.08, 0.13],
          [0.04, 0.05, 0.17, 0.28, 0.11, 0.12, 0.09, 0.15]],

         [[0.11, 0.12, 0.16, 0.13, 0.11, 0.14, 0.10, 0.12],
          [0.13, 0.12, 0.16, 0.13, 0.11, 0.13, 0.10, 0.12],
          [0.08, 0.14, 0.13, 0.13, 0.12, 0.16, 0.13, 0.12],
          [0.10, 0.13, 0.13, 0.13, 0.12, 0.14, 0.12, 0.12],
          [0.13, 0.12, 0.12, 0.12, 0.13, 0.12, 0.13, 0.13],
          [0.14, 0.12, 0.13, 0.12, 0.12, 0.12, 0.12, 0.12],
```

```
          [0.12, 0.13, 0.12, 0.13, 0.13, 0.13, 0.13, 0.13],
          [0.09, 0.13, 0.16, 0.13, 0.11, 0.15, 0.10, 0.12]]],
        grad_fn=<SelectBackward0>)
```

The last four tokens of each representation of the sequence should not be accounted for in the probability distribution, but they clearly are as the visualization below shows.

```
# sequence 0
display_attention(tensor_sequences[0].int().tolist(), tensor_sequences[0].int().
```

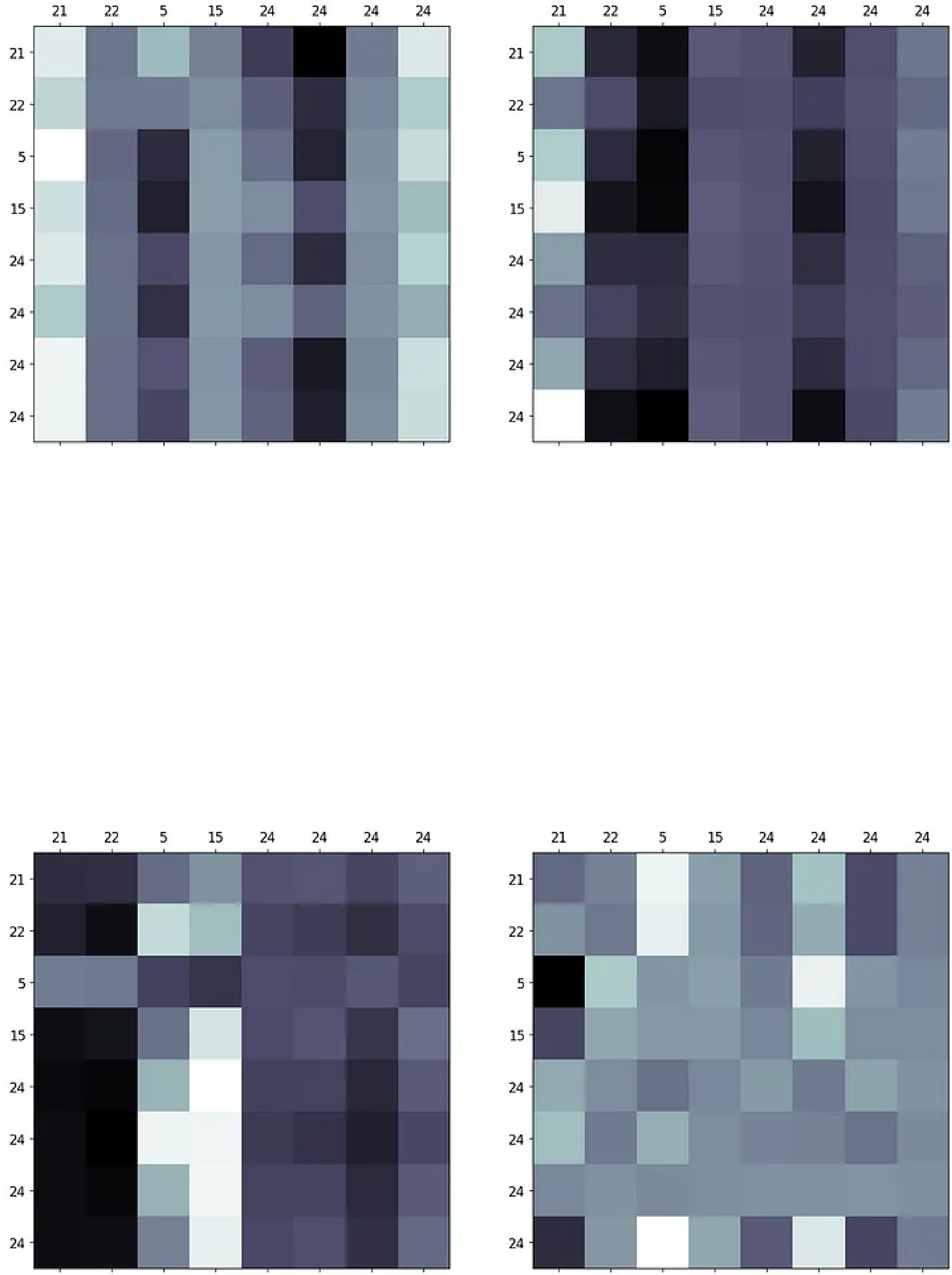


Image by Author

The model should not learn the relation of each token to the padding tokens. It should focus on the tokens from the original sequence and their relation to each other while ignoring the padding tokens. These padding tokens need to be shut off. This is done by creating a source mask that indicates which tokens need to be accounted for.

The Source Mask

A source mask can be created by comparing the tokens in the padded sequences, ***tensor_sequences***, to the padding index. Only the padding token should not be accounted for. When this is passed into the encoder, each padding token's value needs to be replaced with an extremely large negative value, such as -∞ or *-1e10*. Since this is an insignificant value when it is exponentiated (*e-∞ = 0*), it will not have a significant impact on the softmax output. This means only the appropriate tokens will be considered in the probability distribution, and the padding tokens will have a value of 0.

```
tensor([[21, 22,  5, 15, 24, 24, 24, 24],
        [20, 13,  0,  3, 17, 24, 24, 24],
        [ 0,  3, 18, 22,  5, 15, 24, 24]])
```

For reference, the tensor sequences can be seen above.

```
tensor_sequences != pad_idx # pad_idx is 24 in this example
```

```
tensor([[ True,  True,  True,  True, False, False, False, False],
        [ True,  True,  True,  True,  True, False, False, False],
```

```
          [ True,  True,  True,  True,  True,  True, False, False]])
```

Since the padding token is 24, the corresponding values are set to ***False***, and all other tokens remain ***True***. As of now, it has a shape of ***(batch_size, seq_length)***. In order to broadcast this across the attention probabilities, which have a shape of ***(batch_size, n_heads, Q_length, K_length)***, it needs a shape of ***(batch_size, 1, 1, seq_length)***. Remember that ***seq_length, Q_length,*** and ***K_length*** are the same length in the encoder, which is 8 in this context. Essentially, the tensor will contain 3 sequences of 1 matrix that has 1 row, which is the source mask for the sequence. This mask is broadcast across the keys and prevent the queries from calculating their context in the sentence. Remember that the queries are on the left side (rows), and the keys are across the top (columns) in the visualizations.

```
src_mask = (tensor_sequences != pad_idx).unsqueeze(1).unsqueeze(2)
src_mask
```

```
tensor([[[[ True,  True,  True,  True, False, False, False, False]]],
        [[[ True,  True,  True,  True,  True, False, False, False]]],
        [[[ True,  True,  True,  True,  True,  True, False, False]]]])
```

When the same padded sequences are passed through the encoder, the attention probabilities now reflect the expected outcome. The queries, which are the tokens on the left, no longer have an association with the padding tokens in the key.

```
torch.set_printoptions(precision=2, sci_mode=False)

# parameters
vocab_size = len(stoi) + 1 # add one for the padding token
d_model = 8
d_ffn = d_model*4 # 32
n_heads = 4
n_layers = 4
dropout = 0.1

# create the embeddings
lut = Embeddings(vocab_size, d_model) # look-up table (lut)

# create the positional encodings
pe = PositionalEncoding(d_model=d_model, dropout=0.1, max_length=10)

# embed the sequence
embeddings = lut(tensor_sequences)

# positionally encode the sequences
X = pe(embeddings)

# initialize encoder
encoder = Encoder(d_model, n_layers, n_heads,
                  d_ffn, dropout)

# pass through encoder
encoder(src=X, src_mask=src_mask)

# probabilities for sequence 0
encoder.attn_probs[0]
```

```
tensor([[[[0.26, 0.26, 0.25, 0.24, 0.00, 0.00, 0.00, 0.00],
          [0.28, 0.29, 0.24, 0.19, 0.00, 0.00, 0.00, 0.00],
          [0.24, 0.21, 0.24, 0.30, 0.00, 0.00, 0.00, 0.00],
          [0.24, 0.25, 0.25, 0.25, 0.00, 0.00, 0.00, 0.00],
          [0.29, 0.23, 0.21, 0.27, 0.00, 0.00, 0.00, 0.00],
          [0.32, 0.21, 0.18, 0.29, 0.00, 0.00, 0.00, 0.00],
          [0.29, 0.21, 0.20, 0.29, 0.00, 0.00, 0.00, 0.00],
          [0.29, 0.22, 0.21, 0.28, 0.00, 0.00, 0.00, 0.00]],

         [[0.16, 0.24, 0.37, 0.22, 0.00, 0.00, 0.00, 0.00],
          [0.20, 0.25, 0.31, 0.24, 0.00, 0.00, 0.00, 0.00],
          [0.33, 0.23, 0.18, 0.26, 0.00, 0.00, 0.00, 0.00],
          [0.17, 0.22, 0.37, 0.23, 0.00, 0.00, 0.00, 0.00],
```

```
          [0.26, 0.25, 0.24, 0.25, 0.00, 0.00, 0.00, 0.00],
          [0.19, 0.24, 0.33, 0.24, 0.00, 0.00, 0.00, 0.00],
          [0.28, 0.25, 0.21, 0.25, 0.00, 0.00, 0.00, 0.00],
          [0.27, 0.24, 0.23, 0.25, 0.00, 0.00, 0.00, 0.00]],

         [[0.22, 0.33, 0.30, 0.15, 0.00, 0.00, 0.00, 0.00],
          [0.22, 0.41, 0.20, 0.17, 0.00, 0.00, 0.00, 0.00],
          [0.26, 0.18, 0.19, 0.37, 0.00, 0.00, 0.00, 0.00],
          [0.25, 0.21, 0.28, 0.26, 0.00, 0.00, 0.00, 0.00],
          [0.26, 0.19, 0.19, 0.35, 0.00, 0.00, 0.00, 0.00],
          [0.26, 0.15, 0.26, 0.33, 0.00, 0.00, 0.00, 0.00],
          [0.26, 0.16, 0.17, 0.41, 0.00, 0.00, 0.00, 0.00],
          [0.26, 0.13, 0.16, 0.45, 0.00, 0.00, 0.00, 0.00]],

         [[0.21, 0.32, 0.21, 0.26, 0.00, 0.00, 0.00, 0.00],
          [0.27, 0.23, 0.24, 0.26, 0.00, 0.00, 0.00, 0.00],
          [0.18, 0.31, 0.29, 0.23, 0.00, 0.00, 0.00, 0.00],
          [0.24, 0.28, 0.22, 0.26, 0.00, 0.00, 0.00, 0.00],
          [0.25, 0.22, 0.30, 0.23, 0.00, 0.00, 0.00, 0.00],
          [0.24, 0.22, 0.31, 0.23, 0.00, 0.00, 0.00, 0.00],
          [0.24, 0.22, 0.32, 0.22, 0.00, 0.00, 0.00, 0.00],
          [0.26, 0.20, 0.31, 0.23, 0.00, 0.00, 0.00, 0.00]]],
        grad_fn=<SelectBackward0>)
```

The first sequence can be seen in the visualization below.

```
# sequence 0
display_attention(tensor_sequences[0].int().tolist(), tensor_sequences[0].int().
```

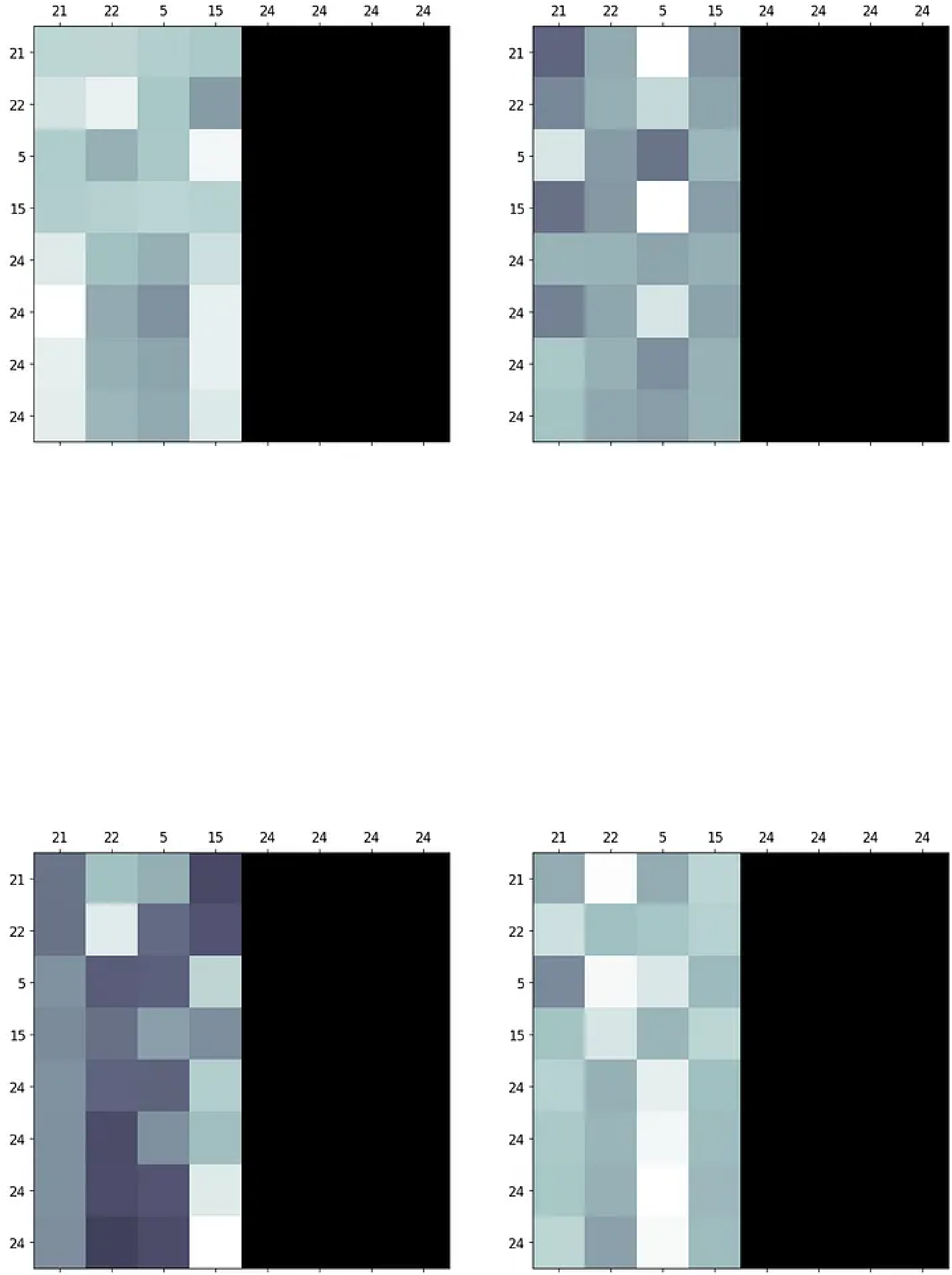


Image by Author

Putting it Together

All of these components can be put together in a function:

```
def make_src_mask(src: Tensor, pad_idx: int = 0):
  """
  Args:
      src:          raw sequences with padding        (batch_size, seq_length)

  Returns:
      src_mask:     mask for each sequence            (batch_size, 1, 1, seq_len
  """
  # assign 1 to tokens that need attended to and 0 to padding tokens, then add 2
  src_mask = (src != pad_idx).unsqueeze(1).unsqueeze(2)

  return src_mask

def pad_seq(seq: Tensor, max_length: int = 10, pad_idx: int = 0):
  """
  Args:
      seq:          raw sequence (batch_size, seq_length)
      max_length:   maximum length of a sequence
      pad_idx:      index for padding tokens

  Returns:
      padded seq:   padded sequence (batch_size, max_length)
  """
  pad_to_add = max_length - len(seq) # amount of padding to add

  return pad(seq,(0, pad_to_add), value=pad_idx,)

sequences = ['What will come next?',
             'This is a basic paragraph.',
             'A basic split will come next!']

# 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

max_length = 8
```

```
pad_idx = len(stoi)

padded_seqs = []

for seq in indexed_sequences:
  # pad each sequence
  padded_seqs.append(pad_seq(torch.Tensor(seq), max_length, pad_idx))

# create a tensor from the padded sequences
tensor_sequences = torch.stack(padded_seqs).long()

# create the source masks for the sequences
src_mask = make_src_mask(tensor_sequences, pad_idx)

torch.set_printoptions(precision=2, sci_mode=False)

# parameters
vocab_size = len(stoi) + 1 # add one for the padding token
d_model = 8
d_ffn = d_model*4 # 32
n_heads = 4
n_layers = 4
dropout = 0.1

# create the embeddings
lut = Embeddings(vocab_size, d_model) # look-up table (lut)

# create the positional encodings
pe = PositionalEncoding(d_model=d_model, dropout=0.1, max_length=10)

# embed the sequence
embeddings = lut(tensor_sequences)

# positionally encode the sequences
X = pe(embeddings)

# initialize encoder
encoder = Encoder(d_model, n_layers, n_heads,
                  d_ffn, dropout)

# pass through encoder
encoder(src=X, src_mask=src_mask)

# preview each sequence
for i in range(0,3):
  display_attention(tensor_sequences[i].int().tolist(), tensor_sequences[i].int(
```

Below is the first sequence:

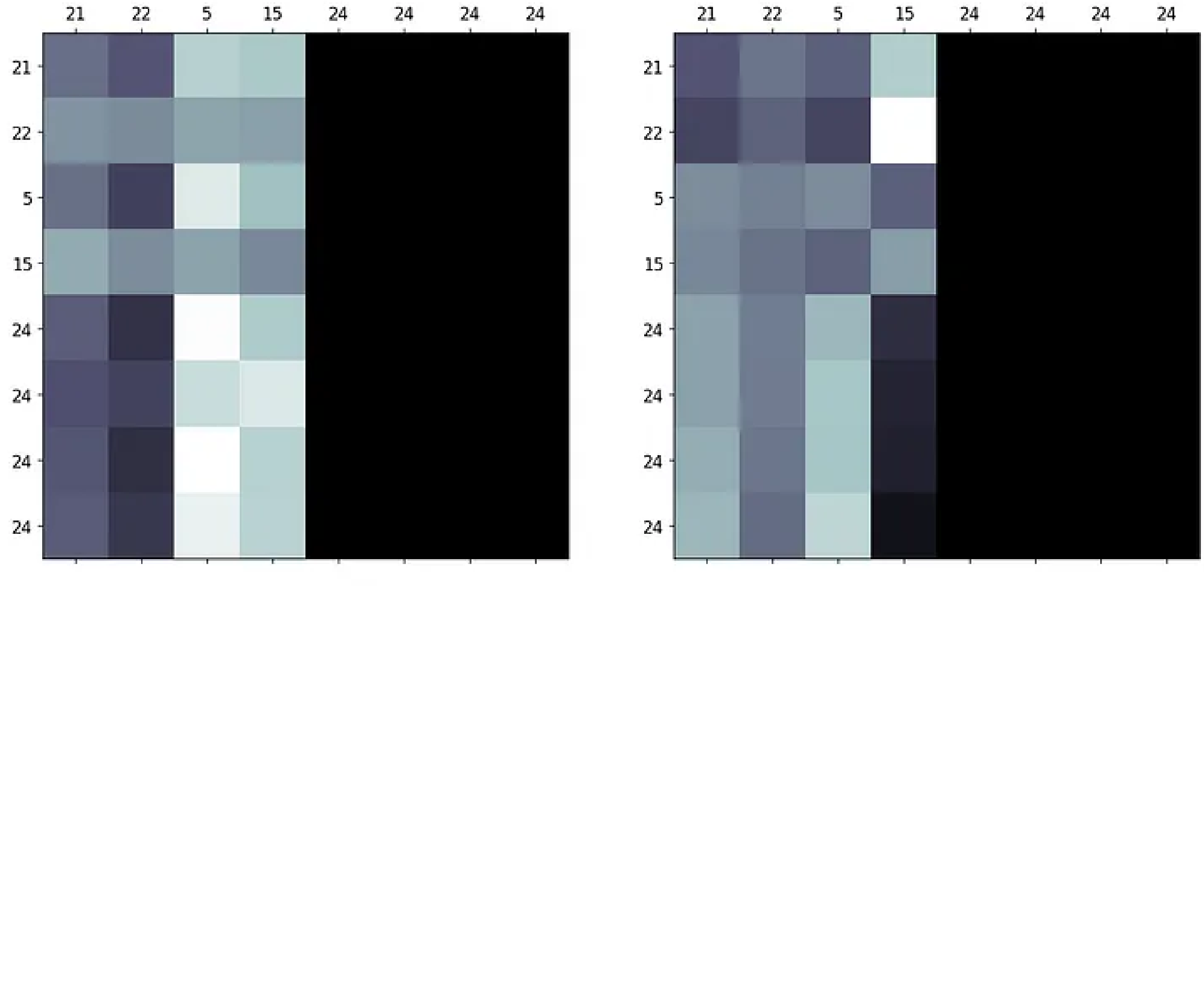


Image by Author

Below is the second sequence:

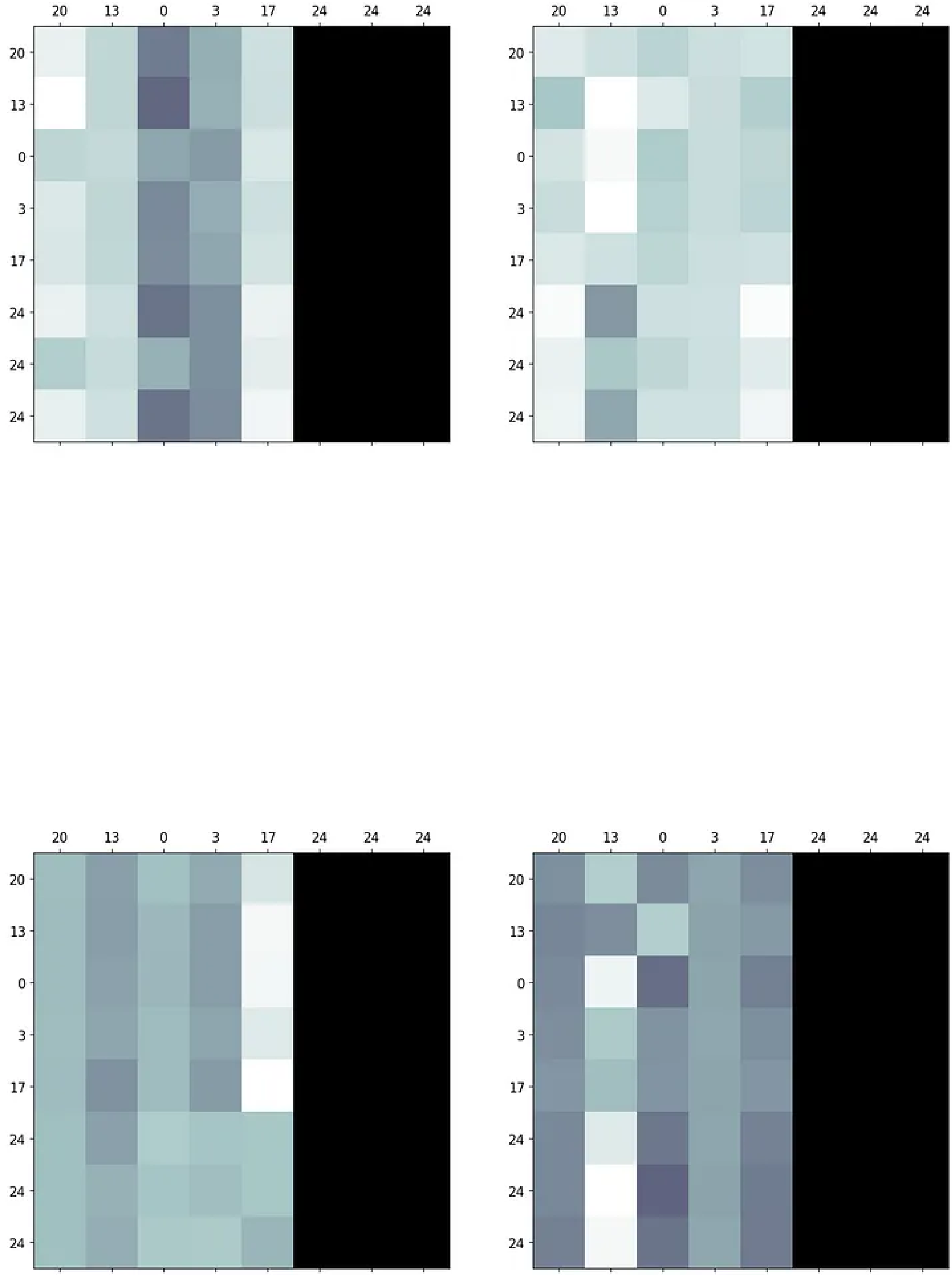


Image by Author

Below is the third sequence:

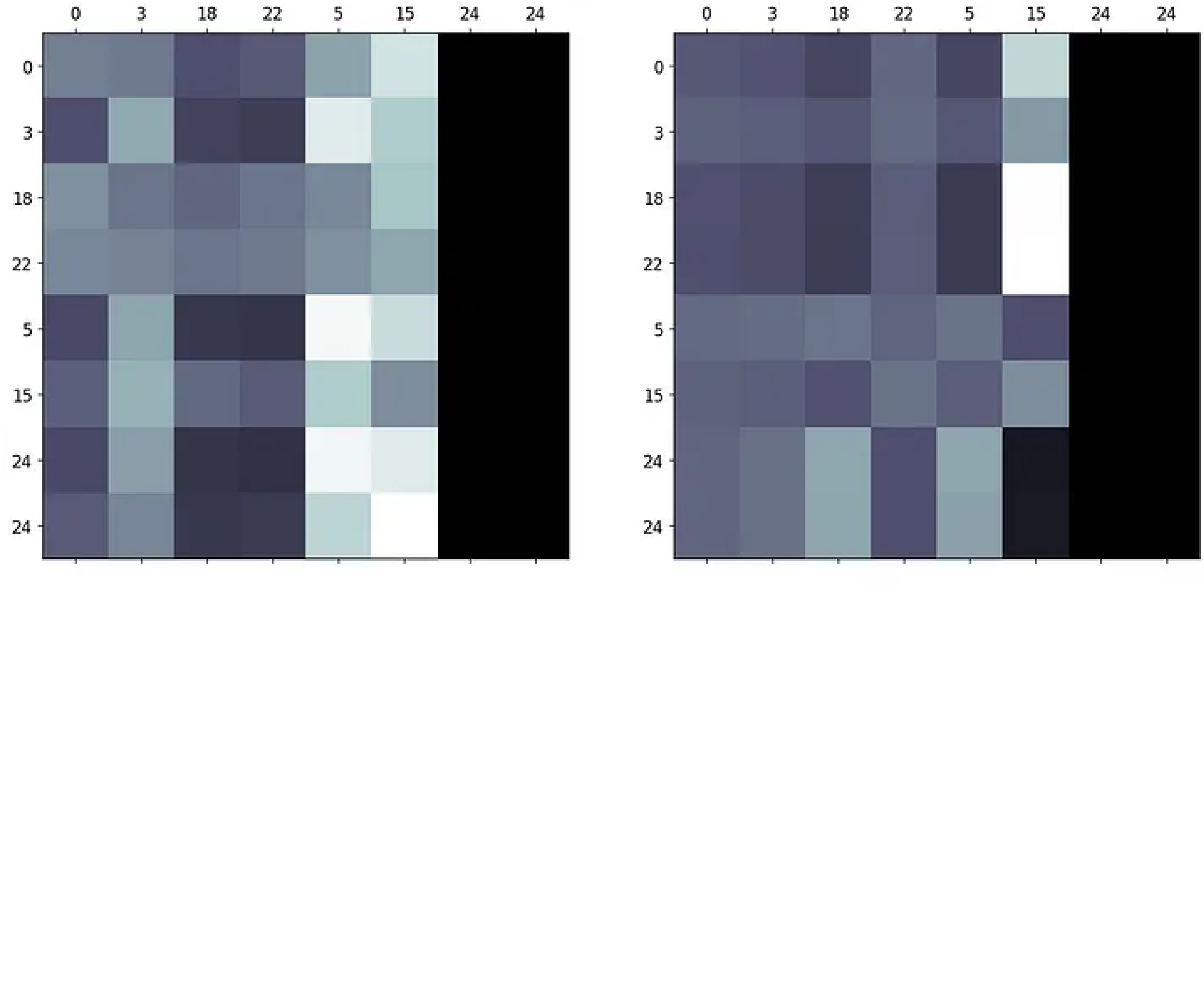


Image by Author

The next article covers The Decoder, which is similar to the encoder.

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

References

1. Deepak Saini's Transformer Implementation

2. Harvard's The Annotated Transformer

Appendix

Display Attention

This function is used to display the attention matrices.

```
def display_attention(sentence: list, translation: list, attention: Tensor,
                      n_heads: int = 8, n_rows: int = 4, n_cols: int = 2):
  """
    Display the attention matrix for each head of a sequence.

    Args:
        sentence:     German sentence to be translated to English; list
        translation:  English sentence predicted by the model
        attention:    attention scores for the heads
        n_heads:      number of heads
        n_rows:       number of rows
        n_cols:       number of columns
  """
  # ensure the number of rows and columns are equal to the number of heads
  assert n_rows * n_cols == n_heads

  # figure size
```

```
    fig = plt.figure(figsize=(15,25))

    # visualize each head
    for i in range(n_heads):

      # create a plot
      ax = fig.add_subplot(n_rows, n_cols, i+1)

      # select the respective head and make it a numpy array for plotting
      _attention = attention.squeeze(0)[i,:,:].cpu().detach().numpy()

      # plot the matrix
      cax = ax.matshow(_attention, cmap='bone')

      # set the size of the labels
      ax.tick_params(labelsize=12)

      # set the indices for the tick marks
      ax.set_xticks(range(len(sentence)))
      ax.set_yticks(range(len(translation)))

      # if the provided sequences are sentences or indices
      if isinstance(sentence[0], str):
        ax.set_xticklabels([t.lower() for t in sentence], rotation=45)
        ax.set_yticklabels(translation)
      elif isinstance(sentence[0], int):
        ax.set_xticklabels(sentence)
        ax.set_yticklabels(translation)

    plt.show()
```

Encoder Transformers Machine Learning NLP

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 Word2ve

· 15 min rea

View list

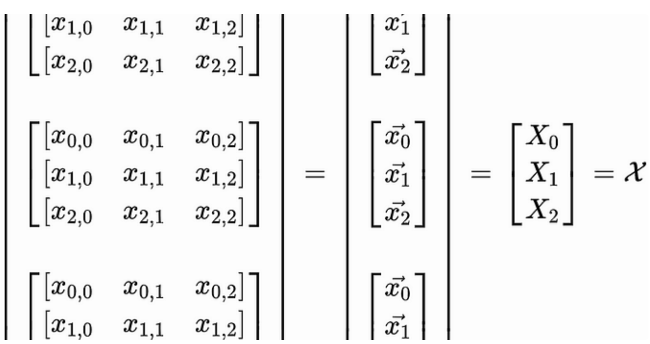
Written by Hunter Phillips

Following

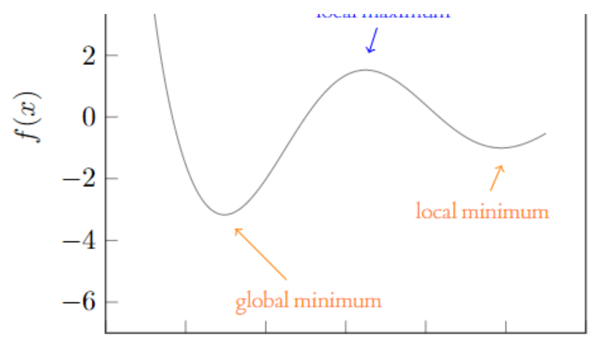
219 Followers

Machine Learning Engineer and Data Scientist

More from Hunter Phillips



Hunter Phillips



Hunter Phillips

A Simple Introduction to Tensors

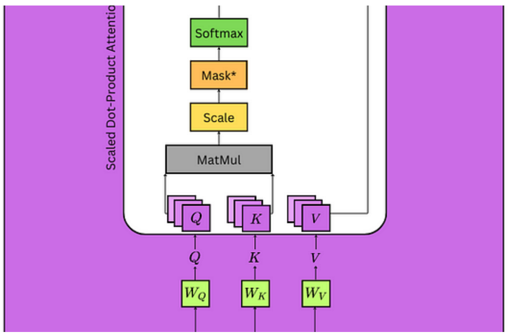
A tensor is a generalization of vectors and matrices to n dimensions. Understanding ho...

11 min read · May 10

A Simple Introduction to Gradient Descent

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

10 min read · May 18

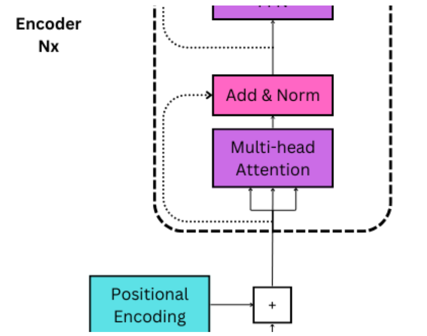


Hunter Phillips

Multi-Head Attention

This article is the third in The Implemented Transformer series. It introduces the multi-...

25 min read · May 9



Hunter Phillips

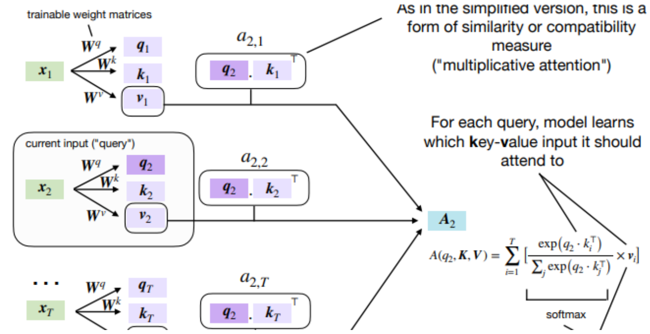
Position-Wise Feed-Forward Network (FFN)

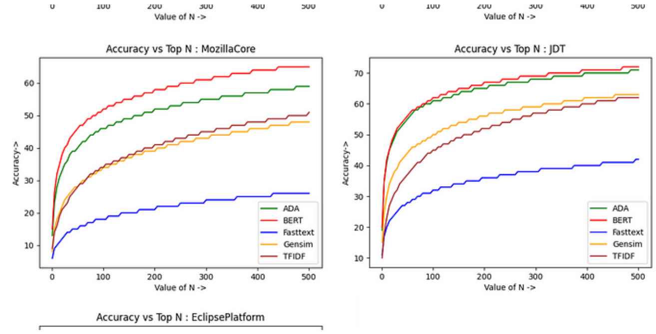
This is the fourth article in The Implemented Transformer series. The Position-wise Feed-...

9 min read · May 9

See all from Hunter Phillips

Recommended from Medium





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

Avinash Patil

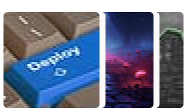
Embeddings: BERT better than ChatGPT4?

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

4 min read · Sep 19

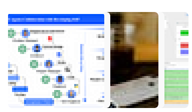
3 1

Lists



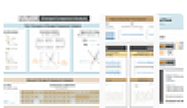
Predictive Modeling w/ Python

20 stories · 452 saves



Natural Language Processing

669 stories · 283 saves

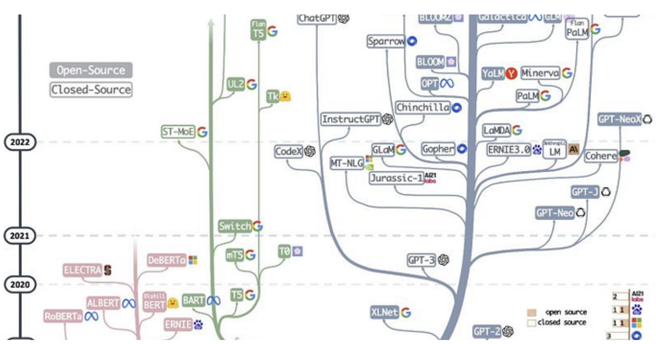


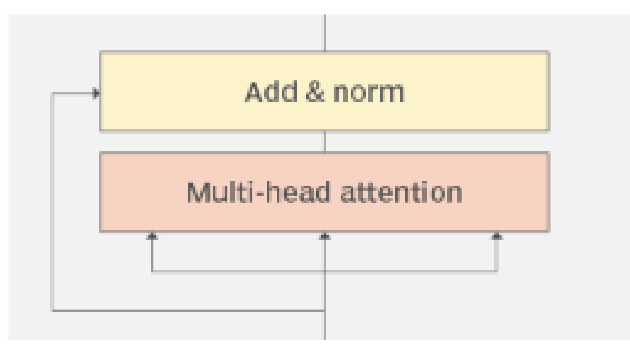
Practical Guides to Machine Learning

10 stories · 519 saves

The New Chatbots: ChatGPT, Bard, and Beyond

13 stories · 133 saves





Haifeng Li

A Tutorial on LLM

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

15 min read · Sep 14

372

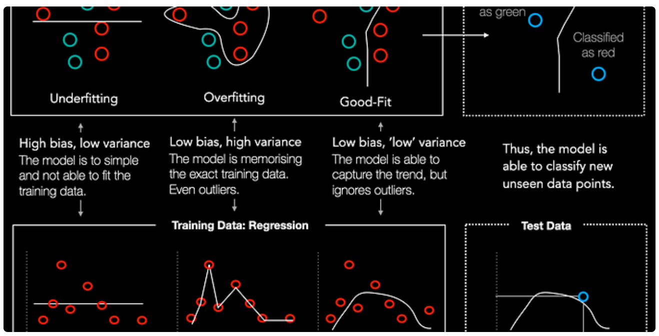
Eugene Ku

Transformer Architecture (Part 3— Scaling Self-Attention)

In part 2, we talked about how Self-Attention (Multi-Head Attention) takes attention...

8 min read · Aug 23

10



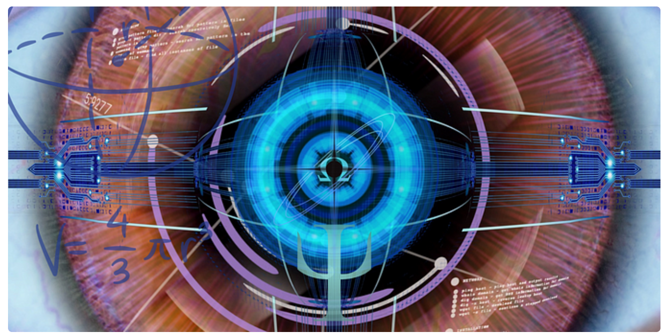
Frederik vl in Advanced Deep Learning

Understanding Bias and Variance in Machine Learning

The terms bias and variance describe how well the model fits the actual unknown data...

3 min read · Sep 15

44



Nabeel Khan in Artificialis

Navigating Math for Computer Vision: Your Ultimate Roadmap

I got myself occupied with developing an understanding of Convolutional Neural...

5 min read · May 15

20

See more recommendations