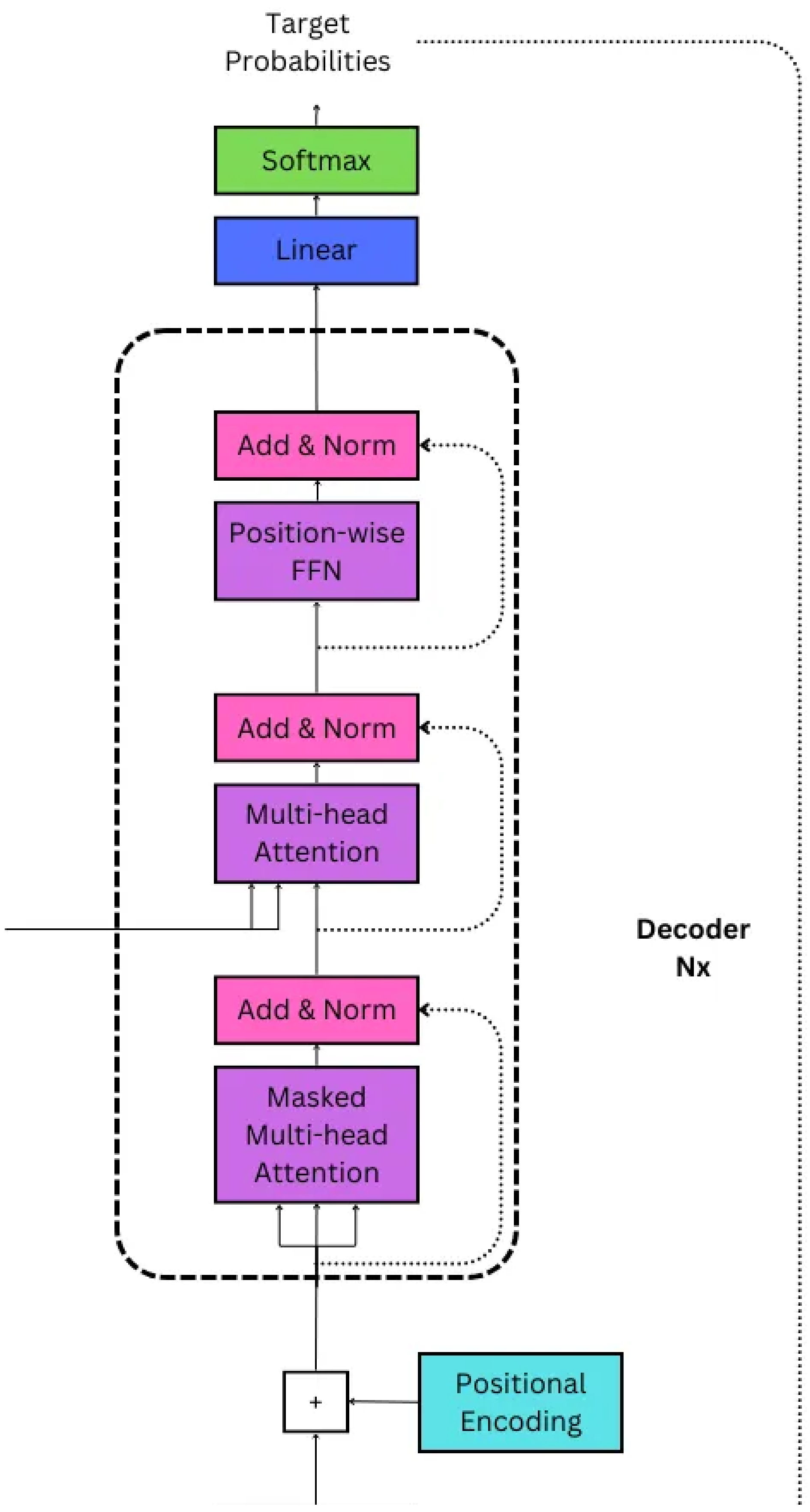
The Decoder

Hunter Phillips · Follow
16 min read · May 10, 2023

257

This is the seventh article in The Implemented Transformer series. The Decoder is the second half of the transformer architecture, and it includes all the previous layers.


Target Probabilities
Softmax
Linear
Add & Norm
Position-wise FFN
Add & Norm
Multi-head Attention
Add & Norm
Masked Multi-head Attention
Decoder Nx
+
Positional Encoding

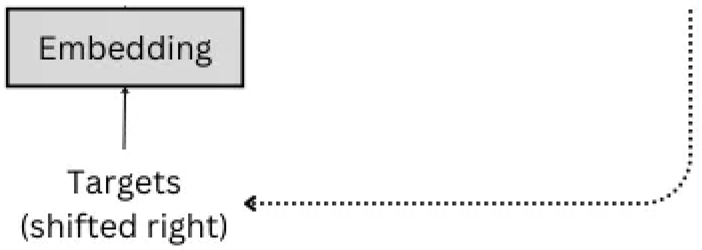


Image by Author

Background

The decoder layer is a wrapper for the sublayers mentioned in the previous articles. It takes the positionally embedded target sequences and passes them through a masked multi-head attention mechanism. Masking is used to prevent the decoder from viewing the next tokens in a sequence. It forces the model to predict the next token using only the previous tokens as context. Then, it is passed through another multi-head attention mechanism; it takes the output of the encoder layers as an additional input. Finally, it is passed through the position-wise FFN. After each of these sublayers, it performs residual addition and layer normalization.

Decoder Layer in Transformers

As mentioned above, the decoder layer is nothing more than a wrapper for the sublayers. It implements two multi-head attention sublayers and a position-wise feed-forward network, each followed by layer normalization and residual addition.

```
class DecoderLayer(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__()
    # masked multi-head attention sublayer
    self.masked_attention = MultiHeadAttention(d_model, n_heads, dropout)
    # layer norm for masked multi-head attention
    self.masked_attn_layer_norm = nn.LayerNorm(d_model)

    # 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, trg: Tensor, src: Tensor, trg_mask: Tensor, src_mask: Tensor
    """
    Args:
        trg:          embedded sequences                (batch_size, trg_seq_len
        src:          embedded sequences                (batch_size, src_seq_len
        trg_mask:     mask for the sequences            (batch_size, 1, trg_seq_
        src_mask:     mask for the sequences            (batch_size, 1, 1, src_s

    Returns:
        trg:          sequences after self-attention    (batch_size, trg_seq_len
        attn_probs:   attention softmax scores
    """
    # pass trg embeddings through masked multi-head attention
    _trg, masked_attn_probs = self.masked_attention(trg, trg, trg, trg_mask)

    # residual add and norm
    trg = self.masked_attn_layer_norm(trg + self.dropout(_trg))

    # pass trg and src embeddings through multi-head attention
    _trg, attn_probs = self.attention(trg, src, src, src_mask)
```

```
        # residual add and norm
        trg = self.attn_layer_norm(trg + self.dropout(_trg))

        # position-wise feed-forward network
        _trg = self.positionwise_ffn(trg)

        # residual add and norm
        trg = self.ffn_layer_norm(trg + self.dropout(_trg))

        return trg, masked_attn_probs, attn_probs
```

Decoder Stack

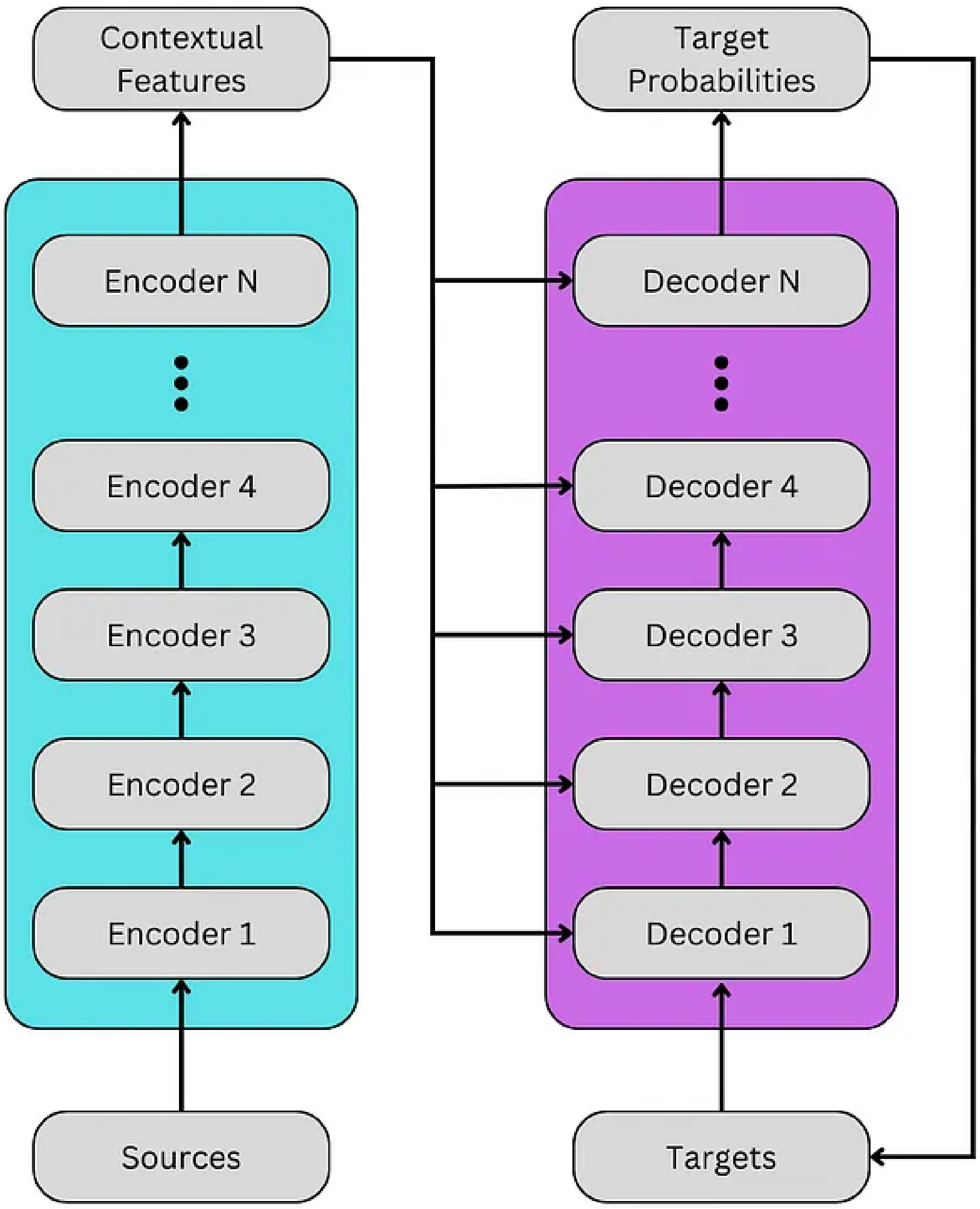


Image by Author

To exploit the benefits of the multi-head attention sublayers, input tokens are passed through a stack of decoder layers at a time, which can be seen in the image above. This is notated as *Nx* in the image at the beginning of the article.

The final linear layer is included in this module to create the logits. Logits are essentially a mock "count" of the frequency of each word in that position in a sequence given the previous words. These "counts" are passed through a softmax function to create a probability distribution that indicates the likelihood of each token in the sequence. The highest "count" will have the highest probability. This is done by projecting ***d_model*** to ***vocab_size***. The output will have a shape of ***(batch_size, seq_length, vocab_size)***. Like before, the linear layer will be broadcast across each sequence.

```
class Decoder(nn.Module):
  def __init__(self, vocab_size: int, d_model: int, n_layers: int,
               n_heads: int, d_ffn: int, dropout: float = 0.1):
    """
    Args:
        vocab_size:   size of the vocabulary
        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([DecoderLayer(d_model, n_heads, d_ffn, dropout)
                                 for layer in range(n_layers)])

    self.dropout = nn.Dropout(dropout)

    # set output layer
    self.Wo = nn.Linear(d_model, vocab_size)

  def forward(self, trg: Tensor, src: Tensor, trg_mask: Tensor, src_mask: Tensor
    """
    Args:
        trg:          embedded sequences                 (batch_size, trg_seq_len
        src:          encoded sequences from encoder     (batch_size, src_seq_len
        trg_mask:     mask for the sequences          (batch_size, 1, trg_seq_
        src_mask:     mask for the sequences          (batch_size, 1, 1, src_s

    Returns:
        output:             sequences after decoder         (batch_size, trg_s
        attn_probs:         attention softmax scores        (batch_size, n_hea
        masked_attn_probs:  masked attention softmax scores   (batch_size, n_hea
    """

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

    self.masked_attn_probs = masked_attn_probs
    self.attn_probs = attn_probs

    return self.Wo(trg)
```

Why Mask?

Target Mask

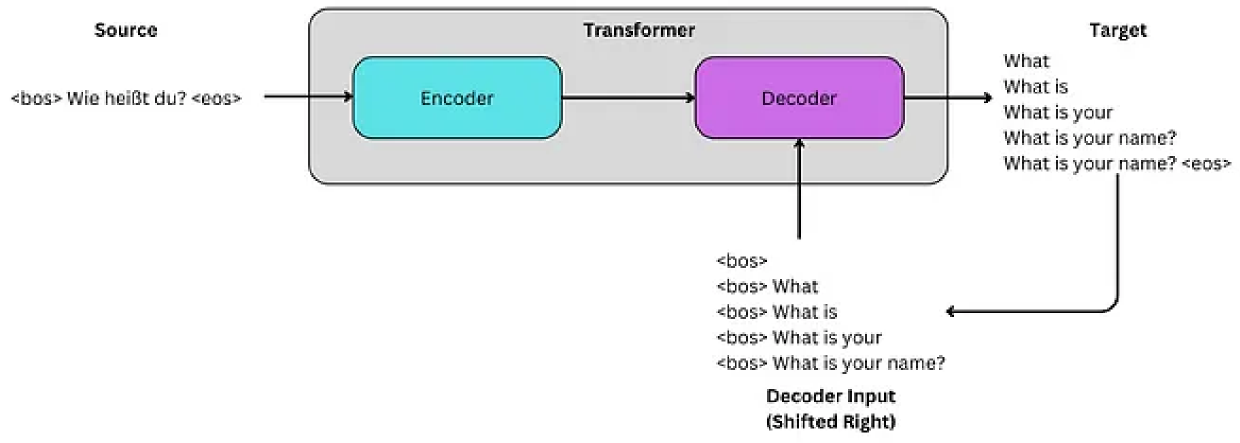


Image by Author

To understand the need for a target mask, it would be best to look at an example of the input and output of the decoder. The goal of the decoder is to predict the next token in the sequence given the encoded source sequence and part of the target sequence. For this to work, there must be a "start" token to prompt the model to predict the next token in the sequence. This is the use of the "*<bos>"* token in the above image. It's also important to note that the size of the input and output to the decoder must be the same.

If the goal is to have the model translate "***Wie heißt du?"*** to ***"What is your name?"***, the encoder would encode the meaning of the source sequence and pass it to the decoder. Given the "***<bos>"*** token and the encoded source, the decoder should predict "***What"***. Then, "***What***" is appended to "***<bos>"*** to create the new input, which is ***"<bos> What"***. This is why the inputs to the decoder are considered to be "shifted right." This can be passed to the decoder to predict `What is`. This token is appended to the previous input to create the new input, "***<bos> What is***". This is passed to the decoder to predict ***"What is your"***. This process repeats itself until the model predicts the ***"<eos>"*** token.

Given a target sequence of ***"<bos> What is your name? <eos>"***, the model can learn each iteration simultaneously by using the target mask:

```
<bos> -----------------------
<bos> what ------------------
<bos> what is ---------------
<bos> what is your ----------
<bos> what is your name -----
```

And the following would be the expected output for each sequence during inference:

```
what ------------------
what is ---------------
what is your ----------
what is your name -----
what is your name <eos>
```

Remember, the decoder's input and output must be the same length. Hence, each target sequence needs its last token removed before being passed to the decoder. If the target sequences are stored in *trg*, the input to the decoder would be *trg[:, :-1]* to select everything except the last token, which can be seen in the target input above. The expected output would be *trg[:, 1:]*, which is everything except the first token, which is the expected output seen above.

To summarize, like the encoder layer, the decoder requires its inputs to be masked. While padding masks are necessary for the input, a look-ahead, or subsequent, mask is also necessary for the target sequences. At inference, the model will only be provided with a start token and must predict the next token based on it. Then, given two tokens, it must predict the third token. This process is repeated until the end-of-sequence token is predicted. This is the autoregressive behavior of the transformer. In other words, future tokens are predicted based only on past tokens and the embeddings from the encoder.

To mimic this behavior, the model learns all of these iterations simultaneously using the subsequent mask.

PyTorch's *torch.tril* can be used to create the subsequent mask. It will have the shape of *(trg_seq_length, trg_seq_length)*.

```
trg_seq_length = 10

subsequent_mask = torch.tril(torch.ones((seq_length, seq_length))).int()
```

```
tensor([[1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [1, 1, 0, 0, 0, 0, 0, 0, 0, 0],
        [1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
        [1, 1, 1, 1, 0, 0, 0, 0, 0, 0],
        [1, 1, 1, 1, 1, 0, 0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]],
        dtype=torch.int32)
```

For each token in the sequence, the probability distribution will only be able to consider the previous tokens. However, since the target sequences must also be padded, the padding mask and subsequent mask have to be combined.

```
pad_mask = torch.Tensor([[1,1,1,1,1,1,1,0,0,0]]).unsqueeze(1).unsqueeze(2).int()
pad_mask
```

```
tensor([[[[1, 1, 1, 1, 1, 1, 1, 0, 0, 0]]]], dtype=torch.int32)
```

This can be easily accompished using the *&* operator, which returns a 1 only when both masks have a 1.

```
subsequent_mask & pad_mask
```

```
tensor([[[[1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
          [1, 1, 0, 0, 0, 0, 0, 0, 0, 0],
          [1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
          [1, 1, 1, 1, 0, 0, 0, 0, 0, 0],
          [1, 1, 1, 1, 1, 0, 0, 0, 0, 0],
          [1, 1, 1, 1, 1, 1, 0, 0, 0, 0],
          [1, 1, 1, 1, 1, 1, 1, 0, 0, 0],
          [1, 1, 1, 1, 1, 1, 1, 0, 0, 0],
          [1, 1, 1, 1, 1, 1, 1, 0, 0, 0],
          [1, 1, 1, 1, 1, 1, 1, 0, 0, 0]]]], dtype=torch.int32)
```

This final target mask has to be created for every sequence in a batch, which means it will take of a shape of ***(batch_size, 1, trg_seq_length, trg_seq_length)***. This mask will be broadcast across each head.

When to Use the Source and Target Masks

Since there are two multi-head attention mechanisms used in the decoder, the target mask will be used in the first one, and the source mask will be used in the second when the encoder's embeddings are provided to the decoder.

In the first mechanism, the target sequences are multiplied against each other. As mentioned, the probability distributions for each token will only consider the previous tokens. This reflects the model's behavior during inference and can be seen in the target mask below, where each token only relies on the tokens before it:

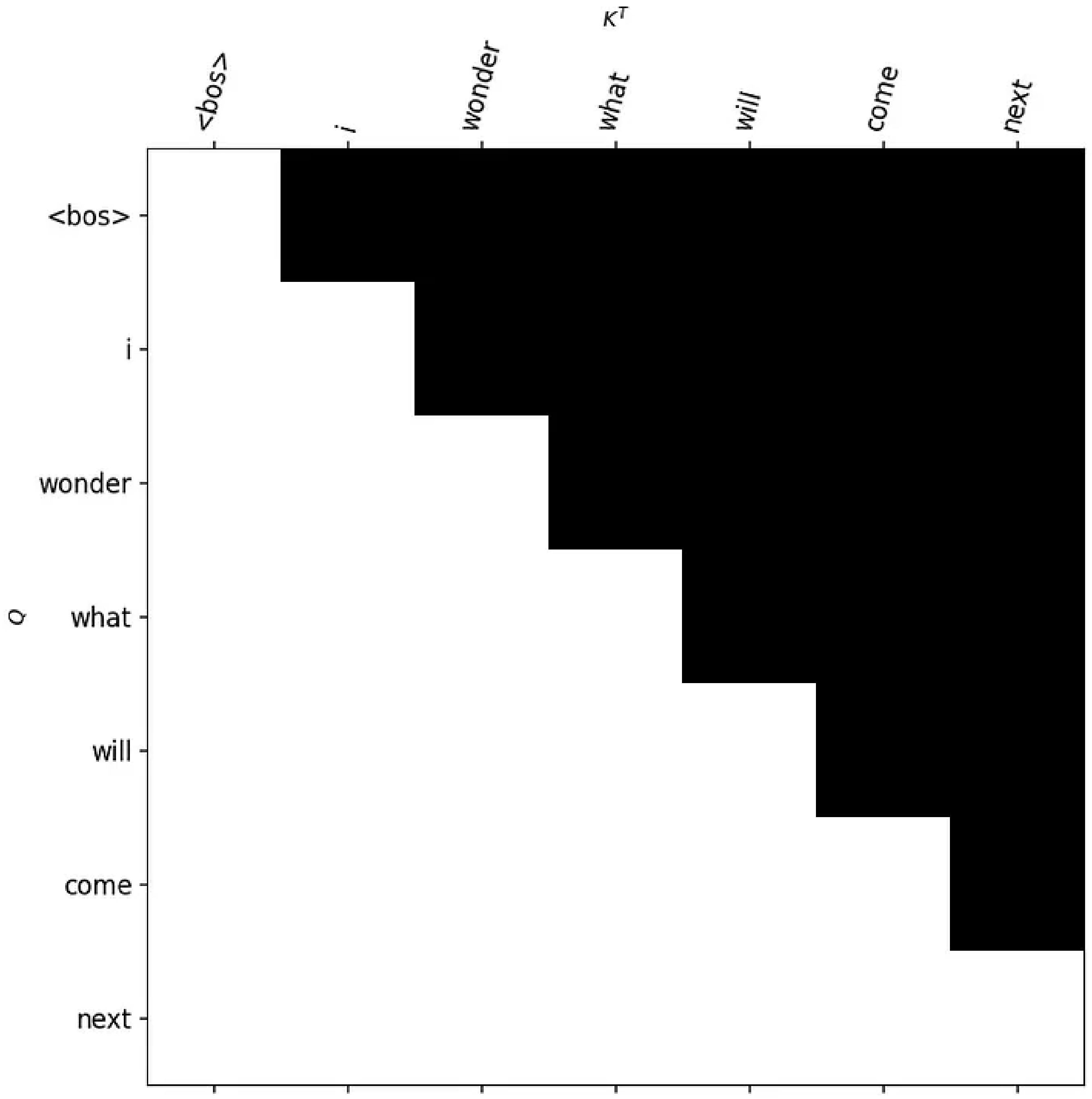
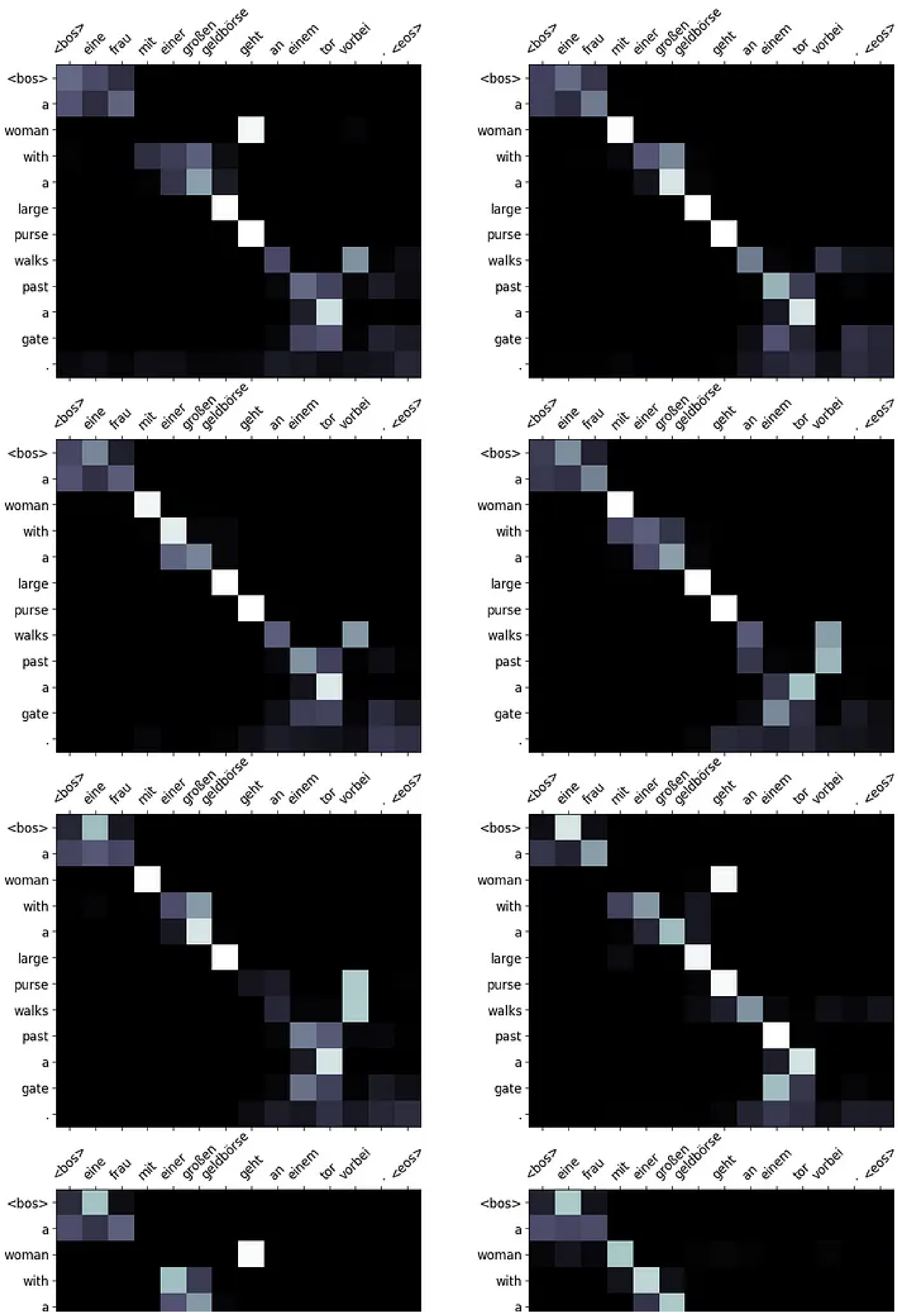


Image by Author

In the second mechanism, the target sequences are the queries, and the sources are the keys. This creates a probability distribution between each target token and source token. During inference, this helps the model identify which target tokens are the best fit for the given source tokens. An example of a trained distribution can be seen below:


<bos>
eine
frau
mit
einer
großen
geldbörse
geht
an
einem
tor
vorbei
.
<eos>
<bos>
a
woman
with
a
large
purse
walks
past
a
gate
.

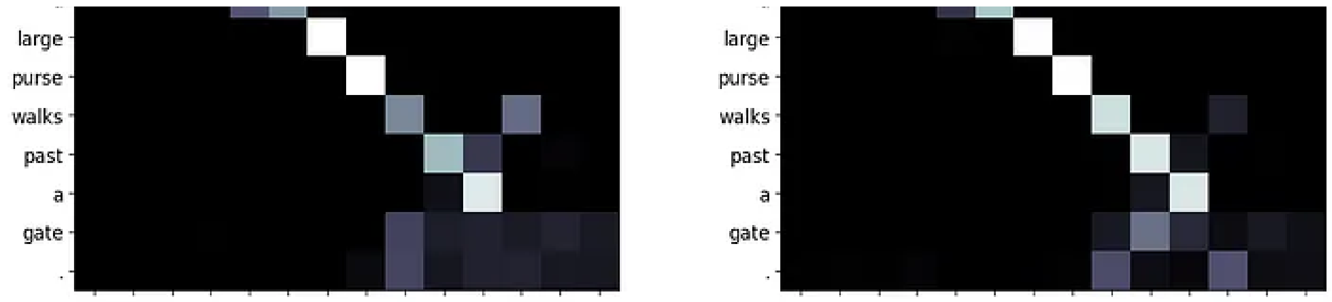


Image by Author

In this visualization, the relationship between each query and its key can be seen. For instance, "*<bos>*" has a strong relationship with "*eine*". "*a*" has its strongest relationship with "*frau*". "*woman*" has its with "*mit*". "*with*" has its with "*einer*". This shows how each of these query tokens relates to the key, or German equivalent, of the English token that should be predicted next.

To recreate this, it is time to combine the encoder and decoder to create a model that can be trained to translate German to English.

Training a Simple Model

Before building the model, German and English vocabularies and sequences must be created. The functions in the appendix are based on those from the encoder article, but they are generalized for English and German.

This model uses the same English example as the previous articles, and a German equivalent was generated using Google Translate.

```
de_example = "Hallo! Dies ist ein Beispiel für einen Absatz, der in seine Grundk
en_example = "Hello! This is an example of a paragraph that has been split into

# build the vocab
```

```
de_stoi = build_vocab(de_example)
en_stoi = build_vocab(en_example)

# build integer-to-string decoder for the vocab
de_itos = {v:k for k,v in de_stoi.items()}
en_itos = {v:k for k,v in en_stoi.items()}
```

Three German-English pairs can be created to facilitate the forward pass. These have to be tokenized, indexed based on the vocabulary, and padded.

```
de_sequences = ["Ich frage mich, was als nächstes kommt!",
                "Dies ist ein Beispiel für einen Absatz.",
                "Hallo, was ist ein Grundkomponenten?"]

en_sequences = ["I wonder what will come next!",
                "This is a basic example paragraph.",
                "Hello, what is a basic split?"]

# pad the sequences
max_length = 9
pad_idx = de_stoi['<pad>']

de_padded_seqs = []
en_padded_seqs = []

# pad each sequence
for de_seq, en_seq in zip(de_indexed_sequences, en_indexed_sequences):
  de_padded_seqs.append(pad_seq(torch.Tensor(de_seq), max_length, pad_idx))
  en_padded_seqs.append(pad_seq(torch.Tensor(en_seq), max_length, pad_idx))

# create a tensor from the padded sequences
de_tensor_sequences = torch.stack(de_padded_seqs).long()
en_tensor_sequences = torch.stack(en_padded_seqs).long()
```

Now, the target sequences can be prepared for training. Each of the target sequences has nine tokens: six from the original sentence, a start token, an end token, and a pad token. The first sequence's tokenized representation can be seen below:

```
['<bos>', 'i', 'wonder', 'what', 'will', 'come', 'next', '<eos>', '<pad>']
```

As mentioned before, the input to the decoder will be a subset of this sequence. The last token has to be removed from the target sequences to allow the decoder to predict the next token in the sequence:

```
['<bos>', 'i', 'wonder', 'what', 'will', 'come', 'next', '<eos>']
```

Similarly, the expected output of the sequence is another subset of the sequence. The first token has to be removed to create the expected output:

```
['i', 'wonder', 'what', 'will', 'come', 'next', '<eos>', '<pad>']
```

The code for these subsets can be seen below, and the source and target masks can be generated as well.

```
# remove last token
trg = en_tensor_sequences[:,:-1]

# remove the first token
expected_output = en_tensor_sequences[:,1:]
```

```
# generate masks
src_mask = make_src_mask(de_tensor_sequences, pad_idx)
trg_mask = make_trg_mask(trg, pad_idx)
```

The target mask for the first sequence can be viewed:

```
display_mask(trg[0].int().tolist(), trg_mask[0])
```

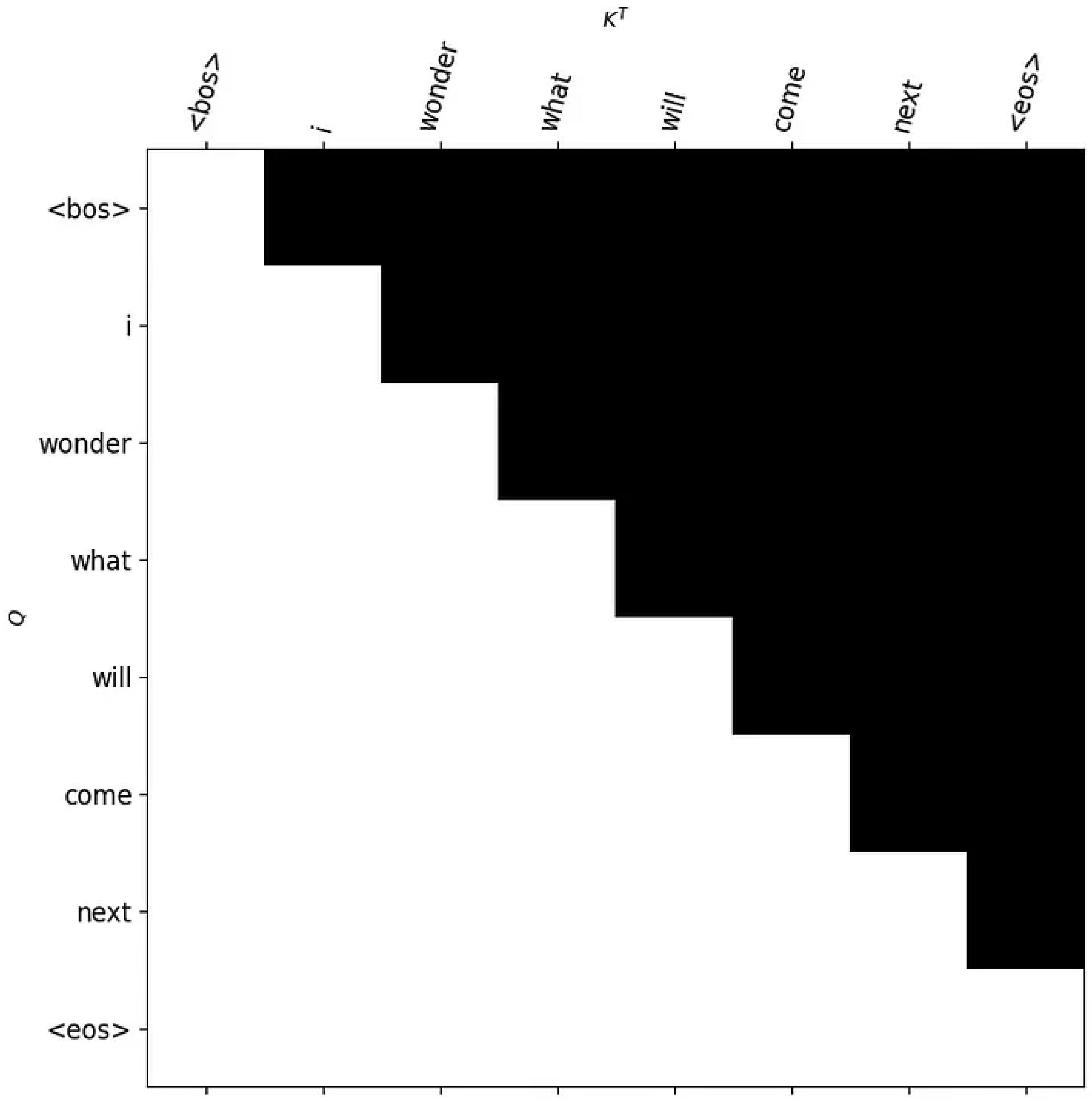


Image by Author

From here, the model can be created. The source embeddings, target embeddings, positional encodings, encoder, and decoder have to be initialized. ***nn.Sequential*** can be used with the source and target embeddings and the positional encodings to create a forward pass through both.

```
# parameters
de_vocab_size = len(de_stoi)
en_vocab_size = len(en_stoi)
d_model = 32
d_ffn = d_model*4 # 32
n_heads = 4
n_layers = 3
dropout = 0.1
max_pe_length = 10

# create the embeddings
de_lut = Embeddings(de_vocab_size, d_model) # look-up table (lut)
en_lut = Embeddings(en_vocab_size, d_model)

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

# embed and encode
de_embed = nn.Sequential(de_lut, pe)
en_embed = nn.Sequential(en_lut, pe)

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

# initialize the decoder
decoder = Decoder(en_vocab_size, d_model, n_layers, n_heads, d_ffn, dropout)
```

With the layers created, the model can be initialized in ***nn.ModuleList,*** which stores all the components in a list that can be accessed by ***Module*** methods, like ***parameters()***. The original paper uses Xavier/Glorot initialization for the model's parameters. All the biases will be 0, and all the weights will fall in the range of:

$$[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}]$$

```
# initialize the model
model = nn.ModuleList([de_embed, en_embed, encoder, decoder])

# normalize the weights
for p in model.parameters():
  if p.dim() > 1:
    nn.init.xavier_uniform_(p)
```

The total number of parameters can be previewed with a simple function.

```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f'The model has {count_parameters(model):,} trainable parameters.')
```

```
The model has 91,675 trainable parameters.
```

Now, a simple forward pass can be completed on the model, and the predictions can be previewed by taking the *argmax* of the logits.

```
# pass through encoder
encoded_embeddings = encoder(src=de_embed(de_tensor_sequences),
                              src_mask=src_mask)

# logits for each output
logits = decoder(trg=en_embed(trg), src=encoded_embeddings,
                 trg_mask=trg_mask,
                 src_mask=src_mask)

predictions = [[en_itos[tok] for tok in seq] for seq in logits.argmax(-1).tolist
```

```
[['a', '<eos>', 'basic', 'a', 'this', 'a', 'a', 'an'],
 ['wonder', 'into', 'any', 'wonder', 'i', 'wonder', 'wonder', 'an'],
 ['that', 'any', 'has', 'basic', 'split', 'wonder', 'example', 'wonder']]
```

Without training, the output is useless, but this illustrates a basic forward pass. Now, the model can be trained to generate the expected outputs. The hyperparameters, optimizer, and loss function must be chosen. Adam will be the optimizer, and Cross Entropy Loss will be used to assess the loss of the model. The loss function takes the logits, converts them to probabilities with softmax, and compares the *argmax* of them to the expected output.

```
# hyperparameters
LEARNING_RATE = 0.005
EPOCHS = 50

# adam optimizer
optimizer = torch.optim.Adam(model.parameters(), lr = LEARNING_RATE)

# loss function
criterion = nn.CrossEntropyLoss(ignore_index = en_stoi["<pad>"])
```

A training loop can be created to update the parameters, and the predictions can be previewed on each iteration since only three sequences are used. Note that *torch.nn.utils.clip_grad_norm_(model.parameters(), 1)* is used to prevent exploding gradients.

```
# set the model to training mode
model.train()

# loop through each epoch
```

```
for i in range(EPOCHS):
  epoch_loss = 0

  # zero the gradients
  optimizer.zero_grad()

  # pass through encoder
  encoded_embeddings = encoder(src=de_embed(de_tensor_sequences),
                               src_mask=src_mask)

  # logits for each output
  logits = decoder(trg=en_embed(trg), src=encoded_embeddings,
                   trg_mask=trg_mask,
                   src_mask=src_mask)

  # calculate the loss
  loss = criterion(logits.contiguous().view(-1, logits.shape[-1]),
                   expected_output.contiguous().view(-1))

  # backpropagation
  loss.backward()

  # clip the weights
  torch.nn.utils.clip_grad_norm_(model.parameters(), 1)

  # update the weights
  optimizer.step()

  # preview the predictions
  predictions = [[en_itos[tok] for tok in seq] for seq in logits.argmax(-1).toli

  if i % 7 == 0:
    print("="*25)
    print(f"epoch: {i}")
    print(f"loss: {loss.item()}")
    print(f"predictions: {predictions}")
```

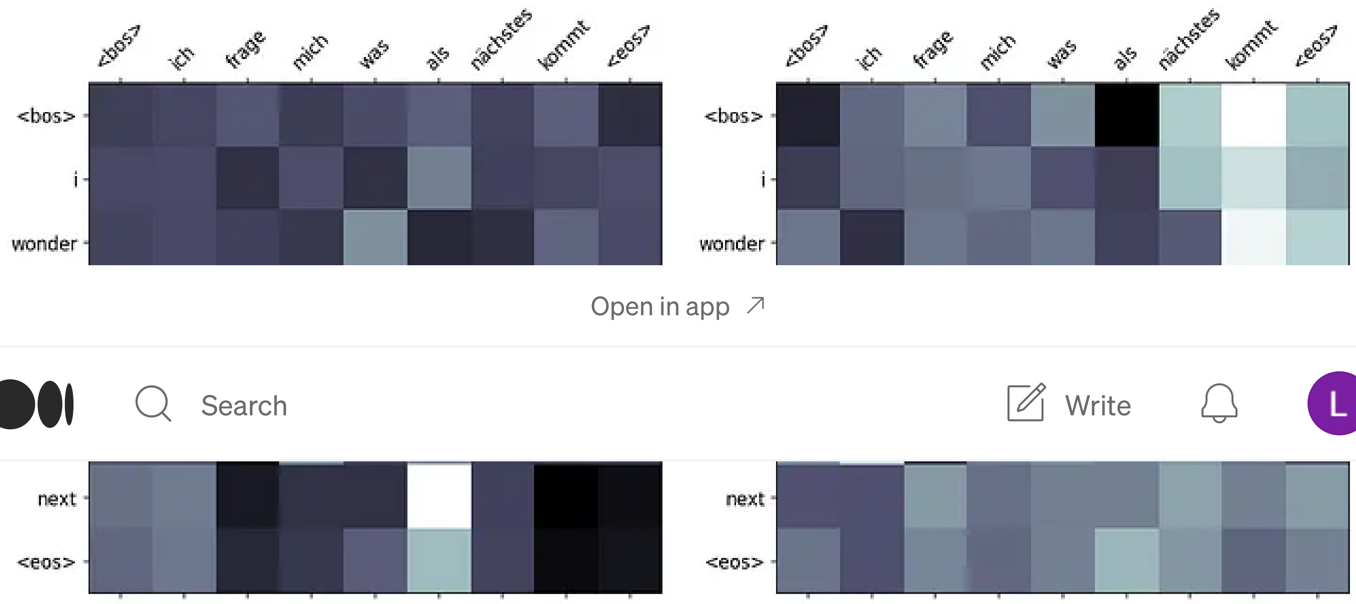
```
=========================
epoch: 0
loss: 3.8633525371551514
predictions: [['an', 'an', 'an', 'an', 'an', 'an', 'an', 'an'],
              ['of', 'an', 'an', 'an', 'an', 'an', 'an', 'an'],
              ['an', 'an', 'an', 'an', 'an', 'an', 'an', 'been']]
=========================
epoch: 7
loss: 2.7589643001556396
```

```
predictions: [['i', 'i', 'i', 'i', 'i', 'i', 'i', 'i'],
              ['is', 'is', 'is', 'is', 'is', 'paragraph', 'is', 'is'],
              ['is', 'is', 'is', 'a', 'is', 'basic', 'basic', 'basic']]
=========================
epoch: 14
loss: 1.7105616331100464
predictions: [['i', 'i', 'i', 'a', 'will', '<eos>', '<eos>', '<eos>'],
              ['hello', 'is', 'this', 'is', 'paragraph', 'paragraph', '<eos>', '
              ['hello', 'example', 'is', 'is', 'basic', '<eos>', '<eos>', '<eos>
=========================
epoch: 21
loss: 1.2171827554702759
predictions: [['i', 'what', 'what', 'next', 'next', 'next', '<eos>', '<eos>'],
              ['this', 'is', 'a', 'basic', 'paragraph', 'a', 'paragraph', '<eos>
              ['this', 'basic', 'is', 'a', 'basic', 'basic', '<eos>', '<eos>']]
=========================
epoch: 28
loss: 0.8726108074188232
predictions: [['i', 'what', 'what', 'will', 'come', '<eos>', '<eos>', '<eos>'],
              ['this', 'is', 'a', 'basic', 'paragraph', 'example', '<eos>', 'par
              ['hello', 'what', 'is', 'a', 'basic', 'split', '<eos>', '<eos>']]
=========================
epoch: 35
loss: 0.6604534387588501
predictions: [['i', 'wonder', 'next', 'will', 'come', 'next', '<eos>', 'next'],
              ['this', 'is', 'a', 'basic', 'example', 'paragraph', '<eos>', 'par
              ['hello', 'what', 'is', 'a', 'basic', 'basic', '<eos>', '<eos>']]
=========================
epoch: 42
loss: 0.3311622142791748
predictions: [['i', 'wonder', 'what', 'will', 'come', 'next', '<eos>', '<eos>'],
              ['this', 'is', 'a', 'basic', 'paragraph', 'paragraph', '<eos>', '<
              ['hello', 'what', 'is', 'a', 'basic', 'split', '<eos>', '<eos>']]
=========================
epoch: 49
loss: 0.19808804988861084
predictions: [['i', 'wonder', 'what', 'will', 'come', 'next', '<eos>', '<eos>'],
              ['this', 'is', 'a', 'basic', 'example', 'paragraph', '<eos>', 'par
              ['hello', 'what', 'is', 'a', 'basic', 'split', '<eos>', 'split']]
```

In 50 epochs, all three sequences were successfully predicted by the model. The decoder attention between the source and the target for the first sequence can be previewed.

```
# convert the indices to strings
decoder_input = [en_itos[i] for i in trg[0].tolist()]

display_attention(de_tokenized_sequences[0], decoder_input, decoder.attn_probs[0
```



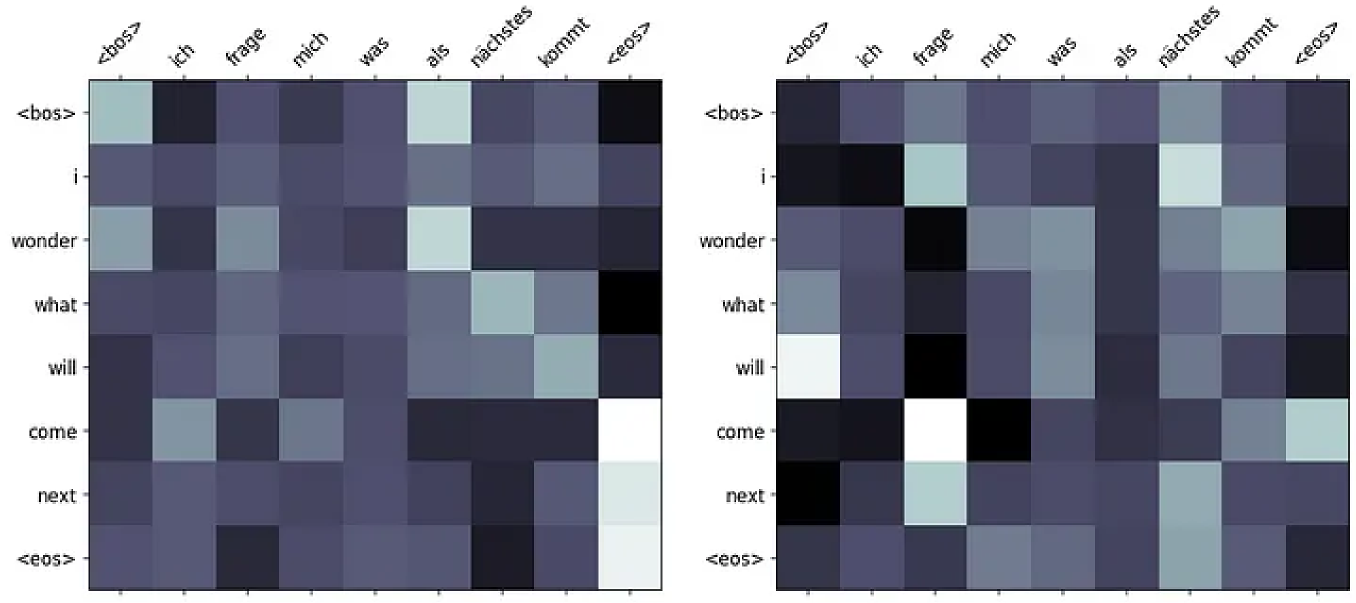


Image by Author

The masked attention can be viewed as well.

```
display_attention(decoder_input, decoder_input, decoder.masked_attn_probs[0],n_h
```

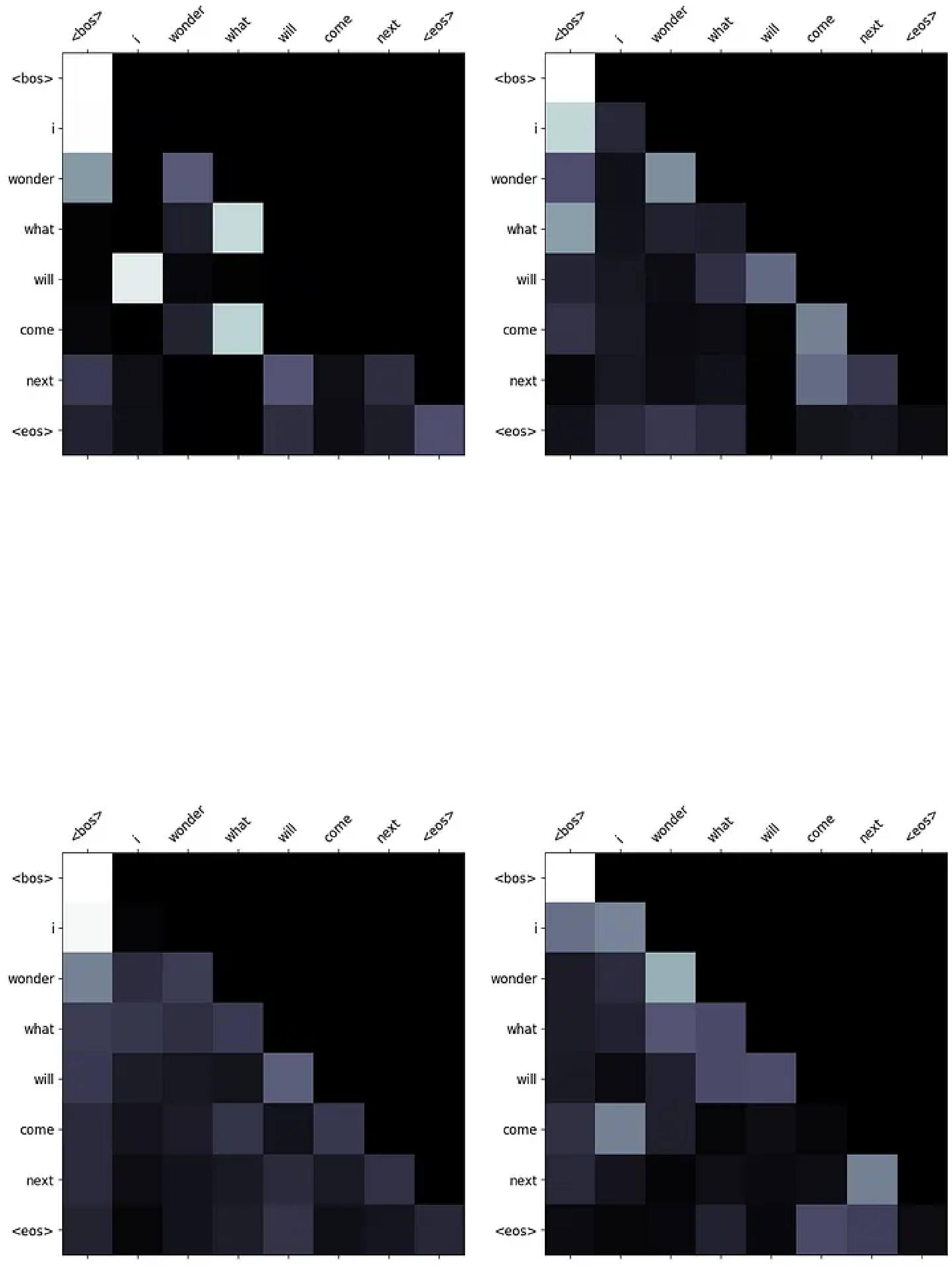


Image by Author

These do not look like the example from earlier in the article, and this is due to the small number of samples that the model was trained on. However, this approach can easily be scaled to thousands of examples, which will be the goal of the last article in the series, Putting it All Together: The Implemented Transformer.

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

References

1. Deepak Saini's Transformer Implementation
2. Harvard's The Annotated Transformer

Appendix

Tokenization

This function is used to tokenize a sentence into its words, and beginning and end of sequence tokens are appended to the list.

```
def tokenize(sequence, special_toks=True):
  # remove punctuation
  for punc in ["!", ".", "?", ","]:
    sequence = sequence.replace(punc, "")

  # split the sequence on spaces and lowercase each token
  sequence = [token.lower() for token in sequence.split(" ")]

  # add beginning and end tokens
```

```
    if special_toks:
      sequence = ['<bos>'] + sequence + ['<eos>']

    return sequence
```

Build Vocabulary

This function is used to generate a vocabulary for a provided corpus. The tokenizer is used to split each corpus into its tokens, and the unique tokens are mapped to an integer and stored in a dictionary.

```
def build_vocab(data):
  # tokenize the data and remove duplicates
  vocab = list(set(tokenize(data, special_toks=False)))

  # sort the vocabulary
  vocab.sort()

  # add special tokens
  vocab = ['<pad>', '<bos>', '<eos>'] + vocab

  # assign an integer to each word
  stoi = {word:i for i, word in enumerate(vocab)}

  return stoi
```

Padding

This function pads a sequence to its maximum length, and it truncates any sequences that are too long.

```
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,)
```

Source Mask

This function is used to create a source mask for a padded sequence.

```
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
```

Target Mask

This function is used to create a target mask for a padded sequence. The target mask allows a token to only attend to itself and any token before it.

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

  Returns:
      trg_mask:     mask for each sequence            (batch_size, 1, seq_length
  """

  seq_length = trg.shape[1]

  # assign True to tokens that need attended to and False to padding tokens, the
  trg_mask = (trg != pad_idx).unsqueeze(1).unsqueeze(2) # (batch_size, 1, 1, seq

  # generate subsequent mask
  trg_sub_mask = torch.tril(torch.ones((seq_length, seq_length))).bool() # (batc

  # bitwise "and" operator | 0 & 0 = 0, 1 & 1 = 1, 1 & 0 = 0
  trg_mask = trg_mask & trg_sub_mask

  return trg_mask
```

Display Mask

This function is used to display the target mask.

```
def display_mask(sentence: list, mask: Tensor):
  """
    Display the target mask for each sequence.

    Args:
        sequence:     sequence to be masked
        mask:         target mask for the heads
  """
  # figure size
  fig = plt.figure(figsize=(8,8))

  # create a plot
  ax = fig.add_subplot(mask.shape[0], 1, 1)

  # select the respective head and make it a numpy array for plotting
  mask = mask.squeeze(0).cpu().detach().numpy()

  # plot the matrix
  cax = ax.matshow(mask, 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(sentence)))

  # set labels
  ax.xaxis.set_label_position('top')
  ax.set_ylabel("$Q$")
  ax.set_xlabel("$K^T$")

  if isinstance(sentence[0], int):
    # convert indices to German/English
    sentence = [en_itos[tok] for tok in sentence]

  ax.set_xticklabels(sentence, rotation=75)
  ax.set_yticklabels(sentence)

  plt.show()
```

Display Attention

This function is used to display the attention matrices of the encoder and decoder.

```
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()
```

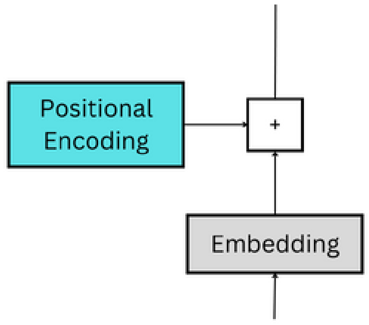
Decoder NLP Transformers Machine Learning Translation

Written by Hunter Phillips

415 Followers

Follow

More from Hunter Phillips

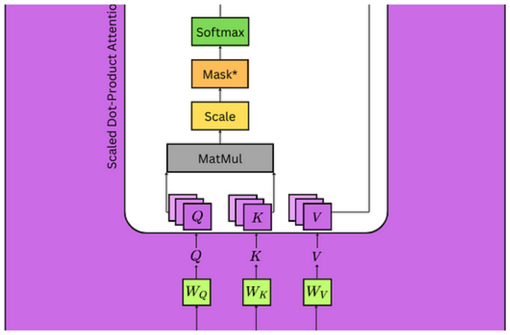


Hunter Phillips

Positional Encoding

This article is the second in The Implemented Transformer series. It introduces positional...

13 min read · May 9, 2023



Hunter Phillips

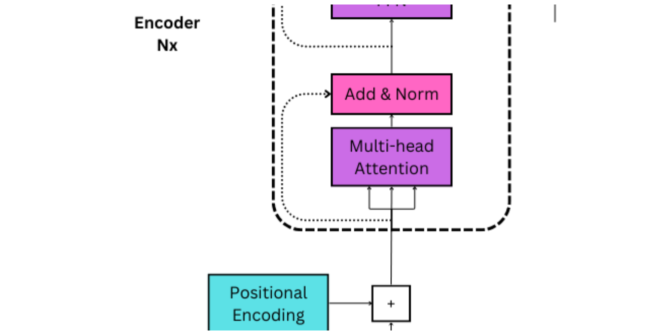
Multi-Head Attention

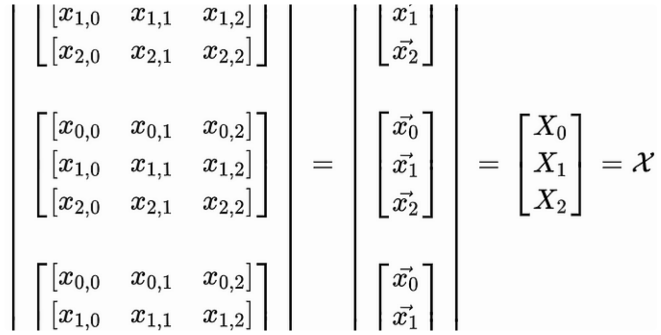
This article is the third in The Implemented Transformer series. It introduces the multi-...

25 min read · May 9, 2023

347 2

197





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, 2023

178

Hunter Phillips

A Simple Introduction to Tensors

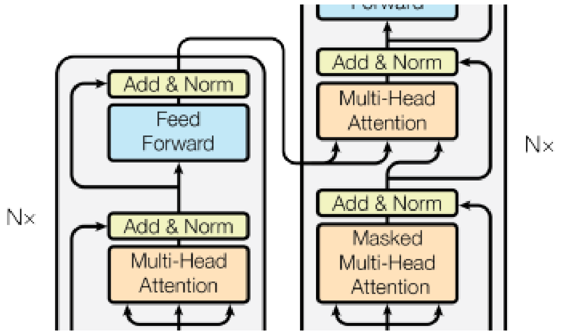
A tensor is a generalization of vectors and matrices to n dimensions. Understanding ho...

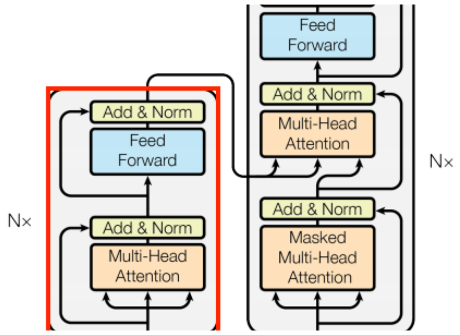
11 min read · May 10, 2023

773 11

See all from Hunter Phillips

Recommended from Medium





Dr. Ernesto Lee

An Intuitive Explanation of 'Attention Is All You Need': The...

This is the technology that makes ChatGPT works!

9 min read · Oct 14, 2023

28

Rokas Liuberskis in Python in Plain English

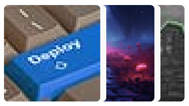
Building a Transformer model with Encoder and Decoder layers in...

In this tutorial, we continue implementing the complete Transformer model in TensorFlow....

· 9 min read · Aug 23, 2023

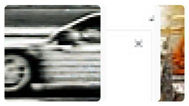
5

Lists



Predictive Modeling w/ Python

20 stories · 904 saves



Natural Language Processing

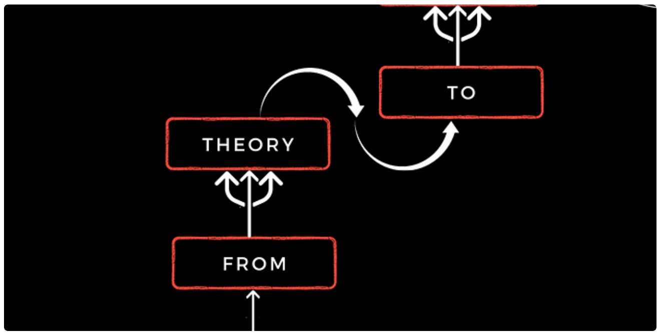
1200 stories · 670 saves

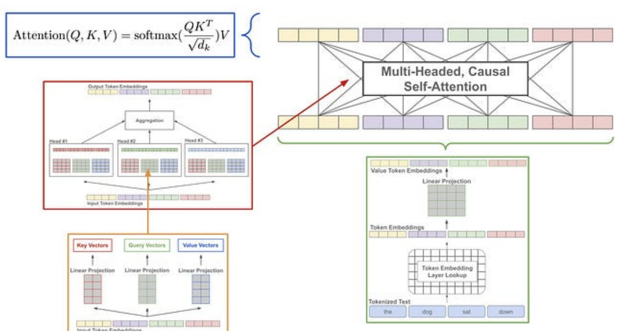
Practical Guides to Machine Learning

10 stories · 1059 saves

The New Chatbots: ChatGPT, Bard, and Beyond

12 stories · 306 saves




Attention(Q, K, V) = softmax($\frac{QK^T}{\sqrt{d_k}}$)V
Multi-Headed, Causal Self-Attention

Awadelrahman M. A. Ahmed

From Theory to Code: Make Sense of Transformers in Machine...

Intro

19 min read · Oct 12, 2023

68

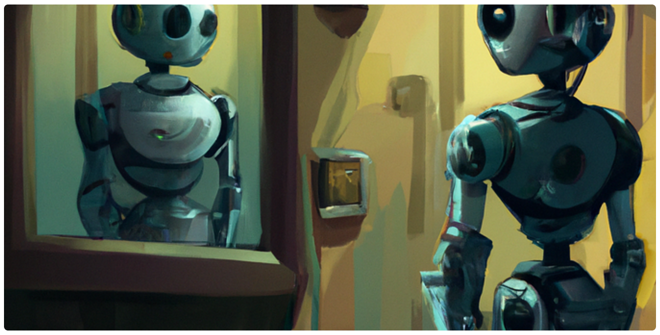
Akash Kesrwani

Multi-Head Self Attention: Short Understanding

Each “block” of a large language model (LLM) is comprised of self-attention and a feed-...

3 min read · Sep 8, 2023

51 1



Thomas van Dongen in Towards Data Science

Demystifying efficient self-attention

A practical overview

20 min read · Nov 7, 2022

623 2

Tech & Tales

Linear Transformations in Machine Learning: A Fundamental Guide

Linear transformations are a foundational concept in machine learning and data...

5 min read · Sep 3, 2023

278 4

See more recommendations