Putting it All Together: The Implemented Transformer

Hunter Phillips · Following

28 min read · May 10

This is the eighth and final article in The Implemented Transformer series. The encoder and decoder are combined to create a model capable of easily translating German to English.

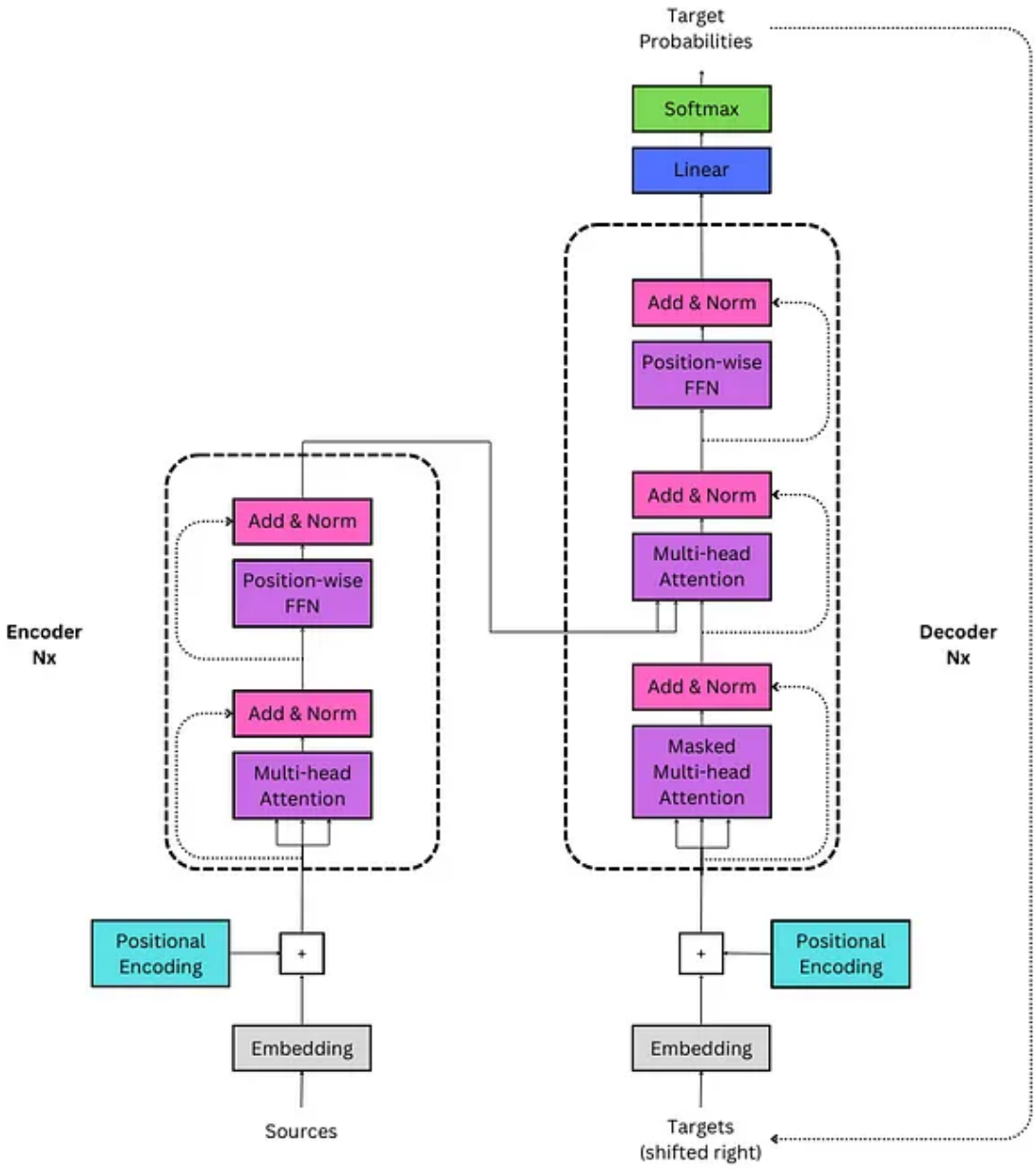


Image by Author

The seven previous articles in this series examined the transformer's components in detail:

1. The Embedding Layer

2. Positional Encoding
3. Multi-Head Attention
4. Position-Wise Feed-Forward Network
5. Layer Normalization
6. The Encoder
7. The Decoder

A quick overview of each is below, and German to English translation follows.

The Embedding Layer

The embedding layer provides each token in a corpus with a corresponding vector representation. This is the first layer that each sequence must be passed through. Each token in each sequence has to be embedded in a vector with a length of ***d_model***. The input into this layer is ***(batch_size, seq_length)***. The output is ***(batch_size, seq_length, d_model)***.

```
class Embeddings(nn.Module):
  def __init__(self, vocab_size: int, d_model: int):
    """
    Args:
      vocab_size:     size of vocabulary
      d_model:        dimension of embeddings
    """
    # inherit from nn.Module
    super().__init__()

    # embedding look-up table (lut)
    self.lut = nn.Embedding(vocab_size, d_model)
```

```
    # dimension of embeddings
    self.d_model = d_model

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

    Returns:
                      embedding vector
    """
    # embeddings by constant sqrt(d_model)
    return self.lut(x) * math.sqrt(self.d_model)
```

Positional Encoding

These embedded sequences are then positionally encoded to provide additional context to each word. This also allows for a single word to have varying meanings depending on its placement in the sentence. The input to the layer is *(batch_size, seq_length, d_model)*. The positional encoding matrix, with a size of *(max_length, d_model)*, must be sliced to the same length as each sequence in the batch, giving it a size of *(seq_length, d_model)*. This same matrix is broadcast and added to each sequence in the batch to ensure consistency. The final output is *(batch_size, seq_length, d_model)*.

```
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_mod
        """
        # add positional encoding to the embeddings
        x = x + self.pe[:, : x.size(1)].requires_grad_(False)

        # perform dropout
        return self.dropout(x)
```

Multi-Head Attention

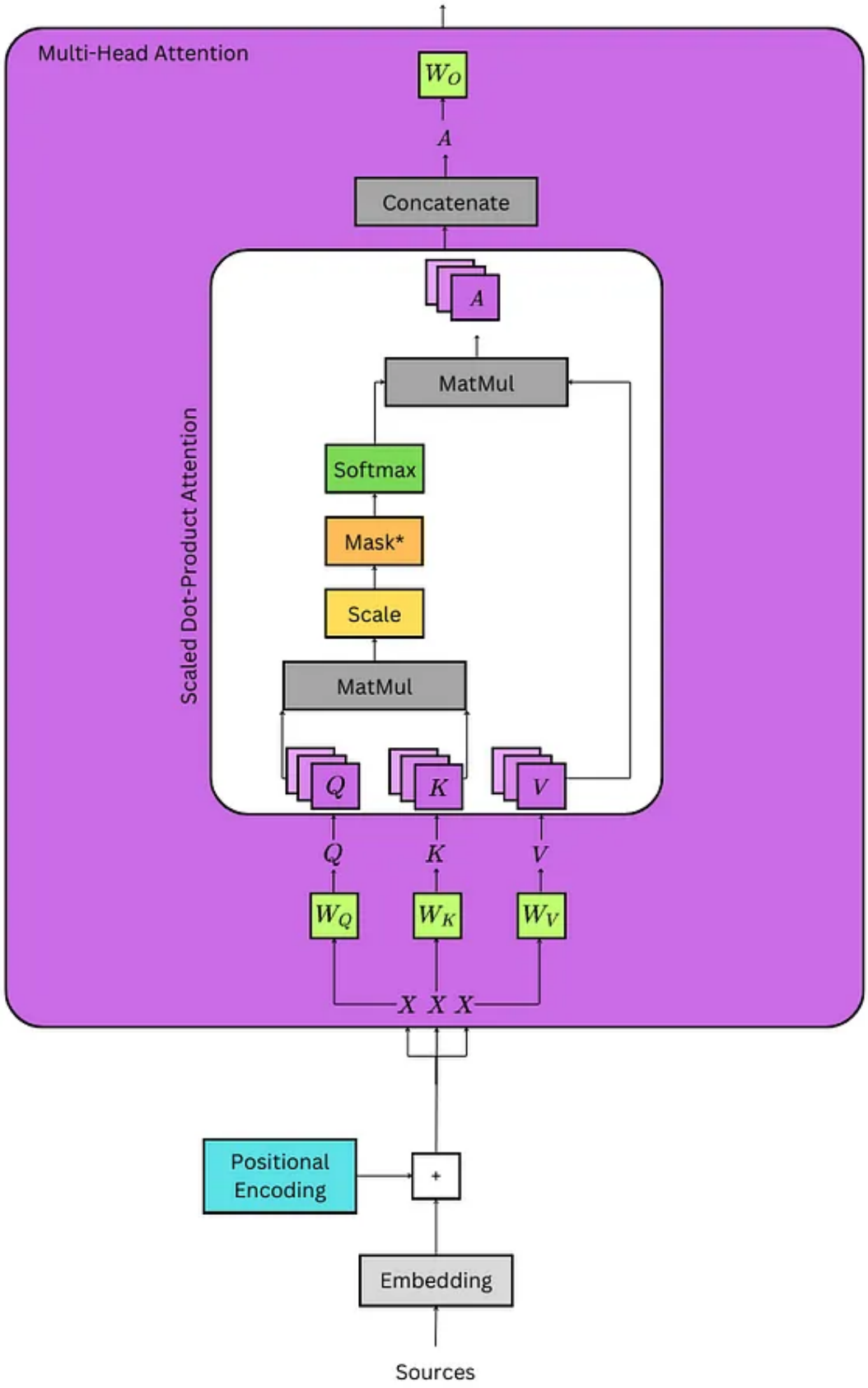

Multi-Head Attention
W_O
A
Concatenate
A
MatMul
Softmax
Mask*
Scale
MatMul
Scaled Dot-Product Attention
Q
K
V
Q
K
V
W_Q
W_K
W_V
X X X
Positional Encoding
+
Embedding
Sources

Image by Author

Three identical versions of these embedded and encoded sequences are passed to the multi-head attention layer to create unique query, key, and value tensors that are transformed by linear layers. They all have a size of *(batch_size, seq_length, d_model)*, where *seq_length* varies based on the respective length of each sequence. These tensors are split into their respective number of heads, taking a size of *(batch_size, n_heads, seq_length, d_key)*, where *d_key = (d_model / n_heads)*. Each sequence now has *n_heads* representations that can attend to different aspects of the sequence during training.

The query and key tensors are multiplied by each other to generate a probability distribution, and they are divided by *√(d_key)*. The key tensor has to be transposed. The output of the multiplication represents each sequence's relationship to itself, and it represents the target sequence's relationship to the source sequence in the second attention mechanism of the decoder. These distributions have a size of *(batch_size, n_heads, Q_length, K_length)*. They are masked depending on the padding of the sequences, or if they are in the first attention-mechanism of the decoder, they are also masked to allow the sequence to only attend to previous tokens, which is the autoregressive property of the decoder.

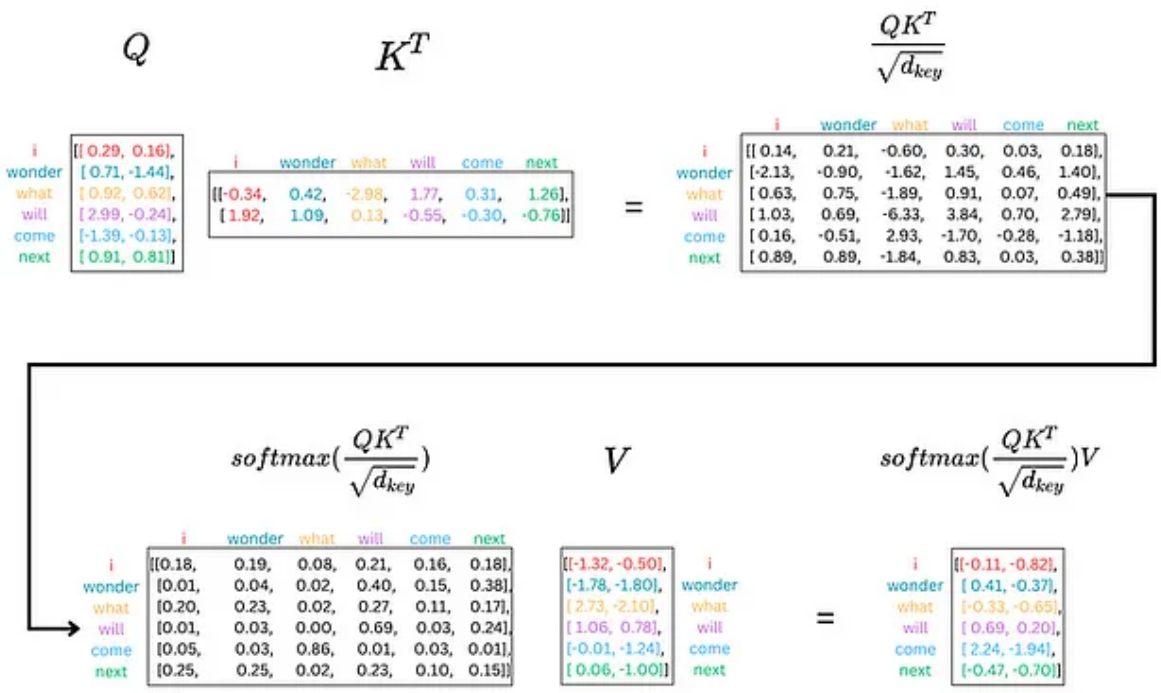


Image by Author

These probabilities are multiplied by another representation of the sequence, which is the values tensor. In the decoder's second attention mechanism, it is the source sequence again. The values tensor has a shape of *(batch_size, n_heads, V_length, d_key)*. The output of the multiplication is *(batch_size, n_heads, Q_length, d_key)*. The two tensors are multiplied together to reweight the values tensor by calculating a summary of the most important contexts for each token in each head or subspace.

This output from the attention mechanism is concatenated back to its original shape, *(batch_size, seq_length, d_model)*, where *seq_length* = *Q_length*. Finally, this tensor is passed through a linear layer with a shape of *(d_model, d_model)* that is broadcast across each sequence. The final output is *(batch_size, seq_length, d_model)*.

```
class MultiHeadAttention(nn.Module):
  def __init__(self, d_model: int = 512, n_heads: int = 8, dropout: float = 0.1)
    """
    Args:
        d_model:      dimension of embeddings
        n_heads:      number of self attention heads
        dropout:      probability of dropout occurring
    """
    super().__init__()
    assert d_model % n_heads == 0            # ensure an even num of heads
    self.d_model = d_model                   # 512 dim
    self.n_heads = n_heads                   # 8 heads
    self.d_key = d_model // n_heads          # assume d_value equals d_key | 512

    self.Wq = nn.Linear(d_model, d_model)    # query weights
    self.Wk = nn.Linear(d_model, d_model)    # key weights
    self.Wv = nn.Linear(d_model, d_model)    # value weights
    self.Wo = nn.Linear(d_model, d_model)    # output weights

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

  def forward(self, query: Tensor, key: Tensor, value: Tensor, mask: Tensor = No
    """
    Args:
       query:         query vector         (batch_size, q_length, d_model)
       key:           key vector           (batch_size, k_length, d_model)
       value:         value vector         (batch_size, s_length, d_model)
       mask:          mask for decoder

    Returns:
       output:        attention values     (batch_size, q_length, d_model)
       attn_probs:    softmax scores       (batch_size, n_heads, q_length, k_len
    """
    batch_size = key.size(0)

    # calculate query, key, and value tensors
    Q = self.Wq(query)                       # (32, 10, 512) x (512, 512) = (32,
    K = self.Wk(key)                         # (32, 10, 512) x (512, 512) = (32,
    V = self.Wv(value)                       # (32, 10, 512) x (512, 512) = (32,

    # split each tensor into n-heads to compute attention

    # query tensor
    Q = Q.view(batch_size,                   # (32, 10, 512) -> (32, 10, 8, 64)
               -1,                           # -1 = q_length
               self.n_heads,
               self.d_key
               ).permute(0, 2, 1, 3)         # (32, 10, 8, 64) -> (32, 8, 10, 64
```

```
        # key tensor
        K = K.view(batch_size,                    # (32, 10, 512) -> (32, 10, 8, 64)
                   -1,                            # -1 = k_length
                   self.n_heads,
                   self.d_key
                   ).permute(0, 2, 1, 3)          # (32, 10, 8, 64) -> (32, 8, 10, 64
        # value tensor
        V = V.view(batch_size,                    # (32, 10, 512) -> (32, 10, 8, 64)
                   -1,                            # -1 = v_length
                   self.n_heads,
                   self.d_key
                   ).permute(0, 2, 1, 3)          # (32, 10, 8, 64) -> (32, 8, 10, 64

        # computes attention
        # scaled dot product -> QK^{T}
        scaled_dot_prod = torch.matmul(Q,         # (32, 8, 10, 64) x (32, 8, 64, 10)
                                       K.permute(0, 1, 3, 2)
                                       ) / math.sqrt(self.d_key)      # sqrt(64)

        # fill those positions of product as (-1e10) where mask positions are 0
        if mask is not None:
          scaled_dot_prod = scaled_dot_prod.masked_fill(mask == 0, -1e10)

        # apply softmax
        attn_probs = torch.softmax(scaled_dot_prod, dim=-1)

        # multiply by values to get attention
        A = torch.matmul(self.dropout(attn_probs), V)       # (32, 8, 10, 10) x (32,
                                                            # (batch_size, n_heads,

        # reshape attention back to (32, 10, 512)
        A = A.permute(0, 2, 1, 3).contiguous()              # (32, 8, 10, 64) -> (32
        A = A.view(batch_size, -1, self.n_heads*self.d_key) # (32, 10, 8, 64) -> (32

        # push through the final weight layer
        output = self.Wo(A)                                 # (32, 10, 512) x (512,

        return output, attn_probs                           # return attn_probs for
```

Position-Wise Feed Forward Network (FFN)

After being passed through layer normalization and undergoing residual addition, the output from the attention mechanism is passed to the FFN. The FFN consists of two linear layers with a ReLU activation function. The first layer has a shape of *(d_model, d_ffn)*. This is broadcast across each sequence of the *(batch_size, seq_length, d_model)* tensor, and it allows the model to learn more about each sequence. The tensor has a shape of *(batch_size, seq_length, d_ffn)* at this point, and it is passed through ReLU. Then, it is passed through the second layer, which has a shape of *(d_ffn, d_model)*. This contracts the tensor to its original size, *(batch_size, seq_length, d_model)*. The outputs are passed through layer normalization and undergo residual addition.

```
class PositionwiseFeedForward(nn.Module):
  def __init__(self, d_model: int, d_ffn: int, dropout: float = 0.1):
    """
    Args:
        d_model:      dimension of embeddings
        d_ffn:        dimension of feed-forward network
        dropout:      probability of dropout occurring
    """
    super().__init__()

    self.w_1 = nn.Linear(d_model, d_ffn)
    self.w_2 = nn.Linear(d_ffn, d_model)
    self.dropout = nn.Dropout(dropout)

  def forward(self, x):
    """
    Args:
        x:            output from attention (batch_size, seq_length, d_model)

    Returns:
        expanded-and-contracted representation (batch_size, seq_length, d_model)
    """
    # w_1(x).relu(): (batch_size, seq_length, d_model) x (d_model,d_ffn) -> (bat
    # w_2(w_1(x).relu()): (batch_size, seq_length, d_ffn) x (d_ffn, d_model) ->
    return self.w_2(self.dropout(self.w_1(x).relu()))
```

Layer Normalization and Residual Addition

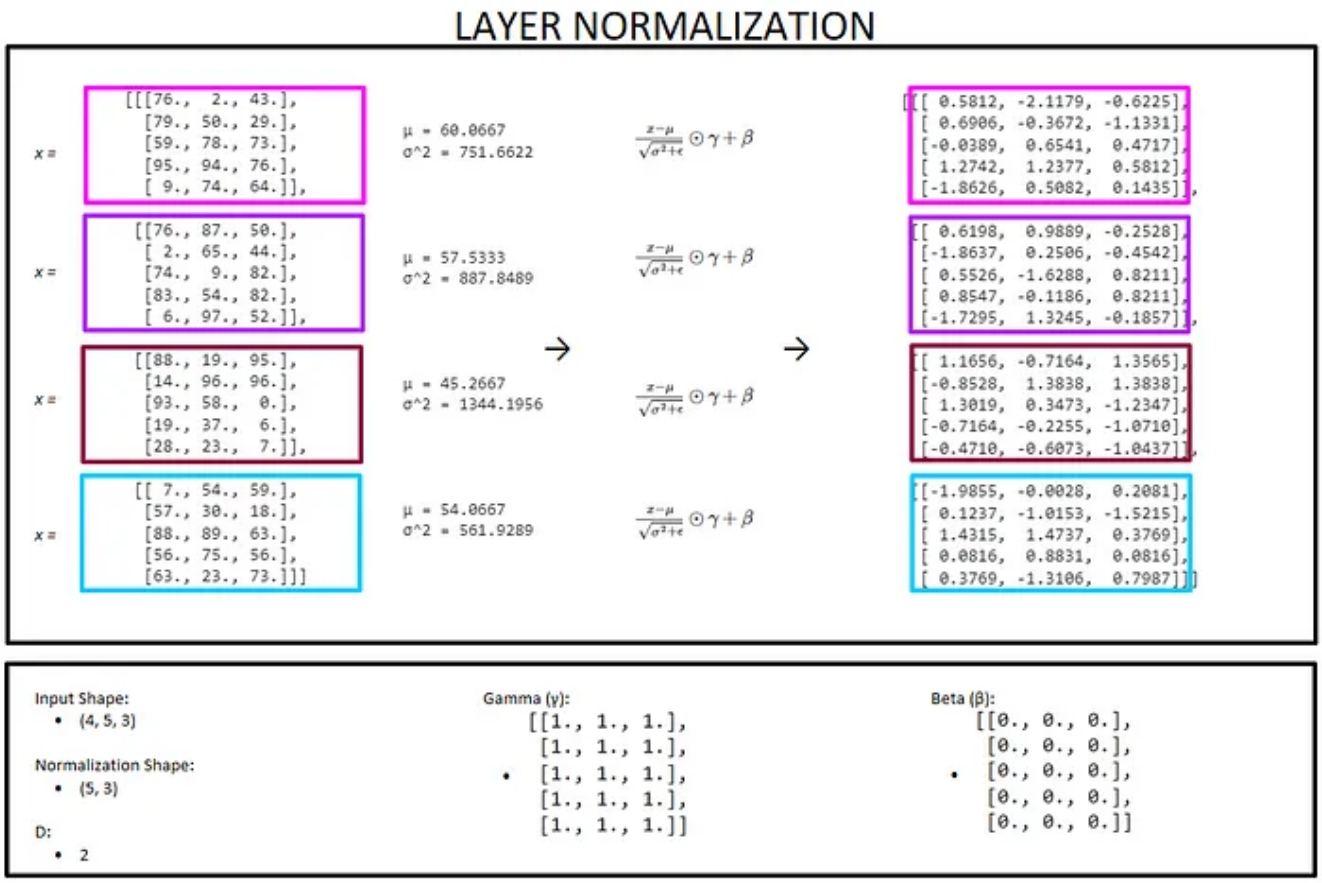


Image by Author

With a shape of *(batch_size, seq_length, d_model)*, layer normalization is performed across each *d_model* vector. The values are standardized using a modified z-score equation to maintain the mean and standard deviation of each embedding vector; this prevents issues with gradient descent.

Residual addition takes the embeddings before they were passed into the layer and adds them to the output. This enriches the embedding vectors with the information obtained from the multi-head attention and FFN.

Neither layer normalization or residual addition impact the shape of their input. These are implemented in the Encoder and Decoder modules, and *nn.LayerNorm* is used for simplicity rather than the custom module created in the article.

The Encoder

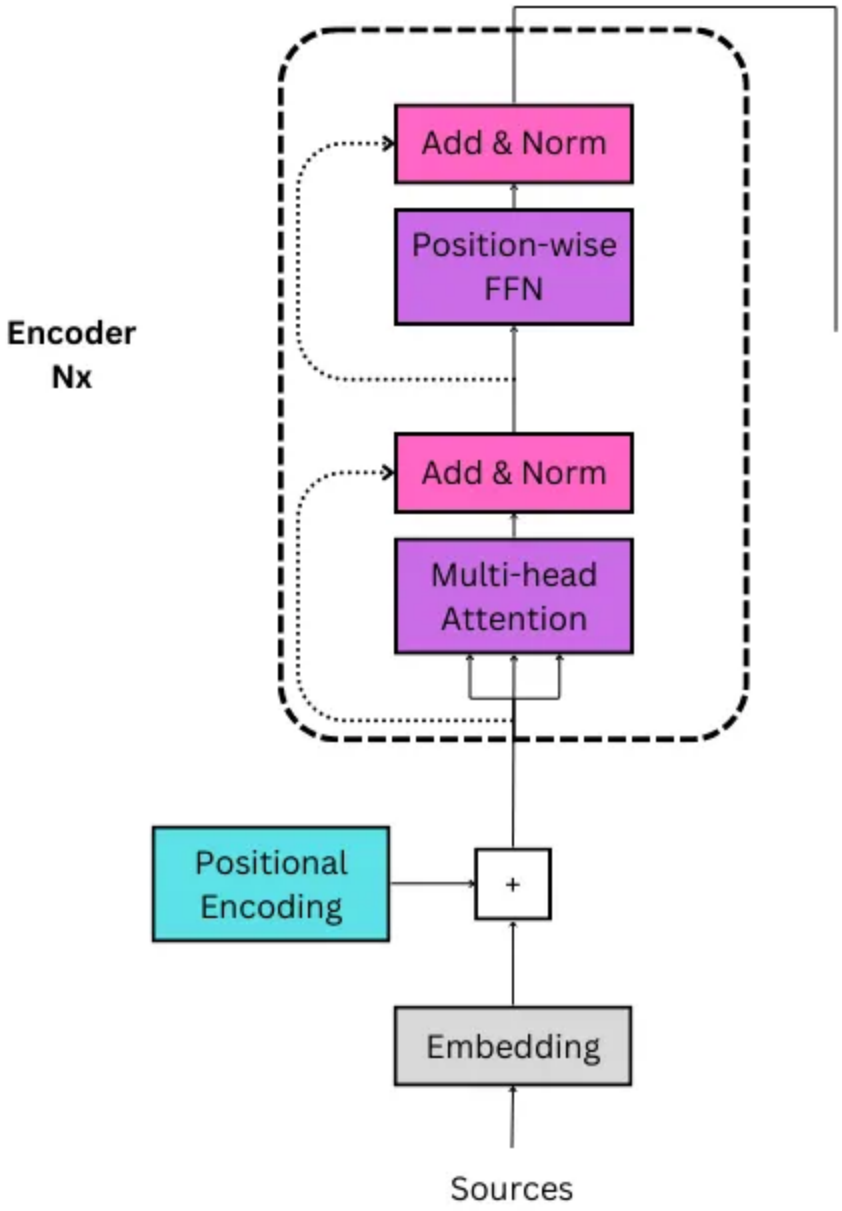


Image by Author

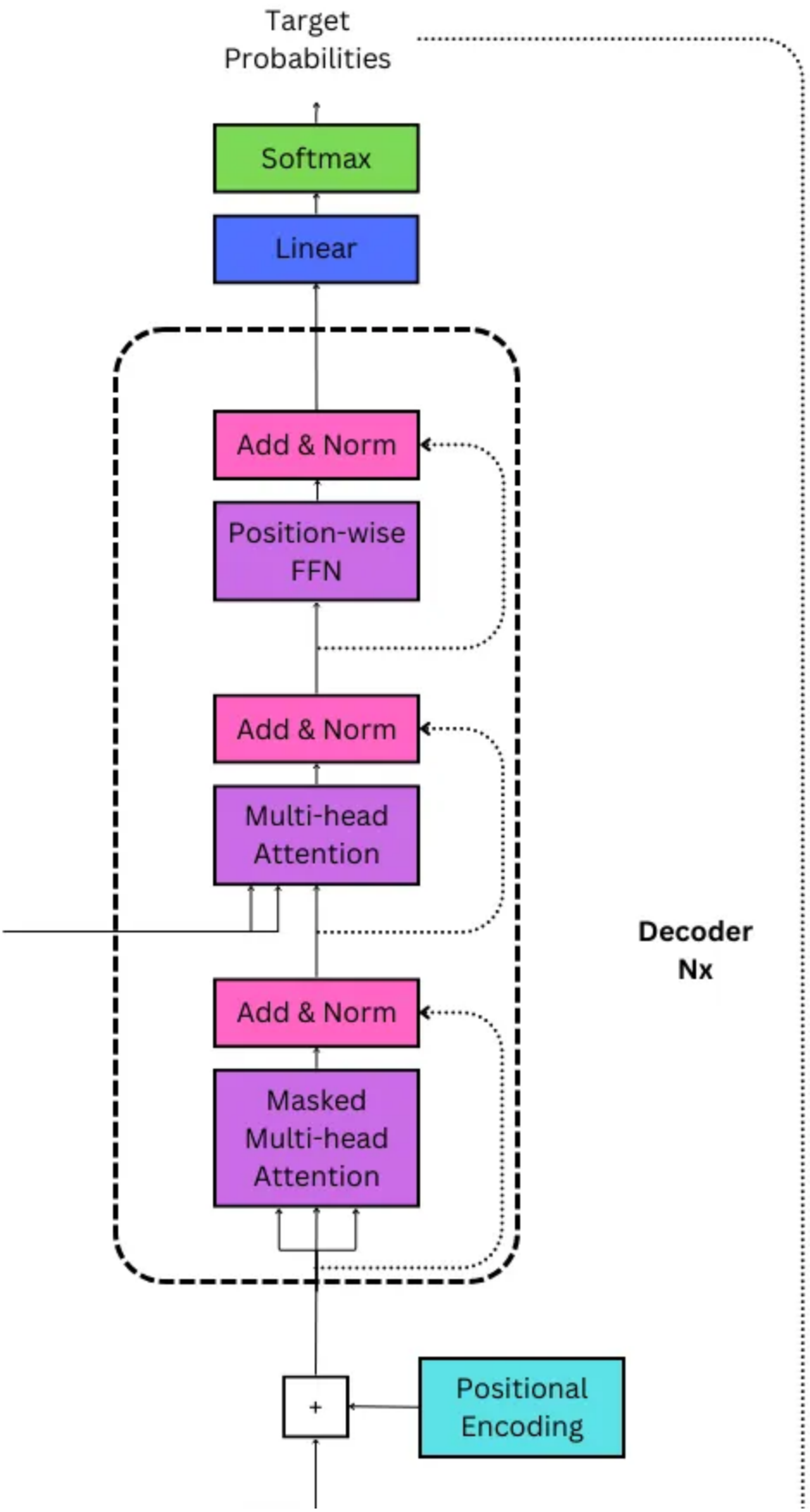
Each encoder layer includes all of the aforementioned layers. It is responsible for enriching the embeddings of the source sequences. The input has a size of ***(batch_size, seq_length, d_model).*** The embedded

sequences are passed directly to the multi-head attention mechanism. After being passed through *Nx* layers in the Encoder stack, the output is an enriched representation of each sequence that contains as much context as possible. It has a size of *(batch_size, seq_length, d_model).*

```
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

    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
```

The Decoder


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

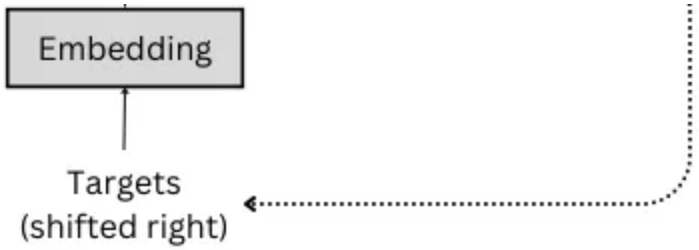


Image by Author

Each decoder layer has two responsibilities: (1) to learn the autoregressive representation of the shifted target sequence and (2) to learn how the target sequence relates to the enriched embeddings from the Encoder. Like the Encoder, a Decoder stack has ***Nx*** decoder layers. As mentioned before, the Encoder output is passed to each decoder layer.

The input to the first decoder layer is shifted right, and it is embedded and encoded. It has a shape of ***(batch_size, seq_length, d_model)***. It is passed through the first attention mechanism, where the model learns an autoregressive representation of the sequence with itself. The output of this mechanism retains its shape, and it is passed to the second attention mechanism. It is multiplied against the encoder's enriched embeddings, and the output once again retains its original shape.

After being passed through the FFN, the tensor is passed through a final linear layer that has a shape of ***(d_model, vocab_size)***. This creates a tensor with a size of ***(batch_size, seq_length, vocab_size)***. These are the logits for the sequence. These logits can be passed through a softmax function, and the highest probability is the prediction for each token.

```
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:   self-attention softmax scores    (batch_size, n_heads, tr
    """
    # pass trg embeddings through masked multi-head attention
    _trg, 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, attn_probs

    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 target 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_seq_len
            attn_probs:   self-attention softmax scores     (batch_size, n_heads, tr
        """

        # pass the sequences through each decoder
        for layer in self.layers:
```

```
            trg, attn_probs = layer(trg, src, trg_mask, src_mask)

        self.attn_probs = attn_probs

        return self.Wo(trg)
```

The Transformer

The Encoder and Decoder can be combined in a module to create the Transformer model. The module can be initialized with an Encoder, Decoder, and the target and source embeddings.

The forward pass requires the source sequences and shifted target sequences. The sources are embedded and passed through the Encoder. The output and embedded target sequences are passed through the Decoder. The functions to create the source and target masks are also part of the module.

The logits are the output of the model. The tensor has a size of *(batch_size, seq_length, vocab_size)*.

```
class Transformer(nn.Module):
    def __init__(self, encoder: Encoder, decoder: Decoder,
                 src_embed: Embeddings, trg_embed: Embeddings,
                 src_pad_idx: int, trg_pad_idx: int, device):
        """
        Args:
            encoder:      encoder stack
            decoder:      decoder stack
            src_embed:    source embeddings and encodings
            trg_embed:    target embeddings and encodings
            src_pad_idx:  padding index
            trg_pad_idx:  padding index
            device:       cuda or cpu
```

```
        Returns:
            output:       sequences after decoder           (batch_size, trg_seq_len
        """
        super().__init__()

        self.encoder = encoder
        self.decoder = decoder
        self.src_embed = src_embed
        self.trg_embed = trg_embed
        self.device = device
        self.src_pad_idx = src_pad_idx
        self.trg_pad_idx = trg_pad_idx

    def make_src_mask(self, src: Tensor):
        """
        Args:
            src:          raw sequences with padding        (batch_size, seq_length)

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

        return src_mask

    def make_trg_mask(self, trg: Tensor):
        """
        Args:
            trg:          raw sequences with padding        (batch_size, seq_length)

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

        seq_length = trg.shape[1]

        # assign True to tokens that need attended to and False to padding tokens, t
        trg_mask = (trg != self.trg_pad_idx).unsqueeze(1).unsqueeze(2) # (batch_size

        # generate subsequent mask
        trg_sub_mask = torch.tril(torch.ones((seq_length, seq_length), device=self.d

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

        return trg_mask

    def forward(self, src: Tensor, trg: Tensor):
```

```
        """
        Args:
            trg:          raw target sequences              (batch_size, trg_seq_len
            src:          raw src sequences                 (batch_size, src_seq_len

        Returns:
            output:       sequences after decoder           (batch_size, trg_seq_len
        """

        # create source and target masks
        src_mask = self.make_src_mask(src) # (batch_size, 1, 1, src_seq_length)
        trg_mask = self.make_trg_mask(trg) # (batch_size, 1, trg_seq_length, trg_seq

        # push the src through the encoder layers
        src = self.encoder(self.src_embed(src), src_mask)  # (batch_size, src_seq_le

        # decoder output and attention probabilities
        output = self.decoder(self.trg_embed(trg), src, trg_mask, src_mask)

        return output
```

Generating a Model

The simple function below initializes the Encoder, Decoder, positional encodings, and embeddings. Then, it passes these into the Transformer module to create a model that can be trained. In the last article, these steps were performed on their own, which is an acceptable alternative.

```
def make_model(device, src_vocab, trg_vocab, n_layers: int = 3, d_model: int = 5
               d_ffn: int = 2048, n_heads: int = 8, dropout: float = 0.1,
               max_length: int = 5000):
  """
    Construct a model when provided parameters.

    Args:
        src_vocab:    source vocabulary
        trg_vocab:    target vocabulary
```

```
        n_layers:     Number of Encoder and Decoders
        d_model:      dimension of embeddings
        d_ffn:        dimension of feed-forward network
        n_heads:      number of heads
        dropout:      probability of dropout occurring
        max_length:   maximum sequence length for positional encodings

    Returns:
        Transformer model based on hyperparameters
    """

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

    # create the decoder
    decoder = Decoder(len(trg_vocab), d_model, n_layers, n_heads, d_ffn, dropout)

    # create source embedding matrix
    src_embed = Embeddings(len(src_vocab), d_model)

    # create target embedding matrix
    trg_embed = Embeddings(len(trg_vocab), d_model)

    # create a positional encoding matrix
    pos_enc = PositionalEncoding(d_model, dropout, max_length)

    # create the Transformer model
    model = Transformer(encoder, decoder, nn.Sequential(src_embed, pos_enc),
                        nn.Sequential(trg_embed, pos_enc),
                        src_pad_idx=src_vocab.get_stoi()["<pad>"],
                        trg_pad_idx=trg_vocab.get_stoi()["<pad>"],
                        device=device)

    # initialize parameters with Xavier/Glorot
    for p in model.parameters():
      if p.dim() > 1:
        nn.init.xavier_uniform_(p)

    return model
```

Translating German to English

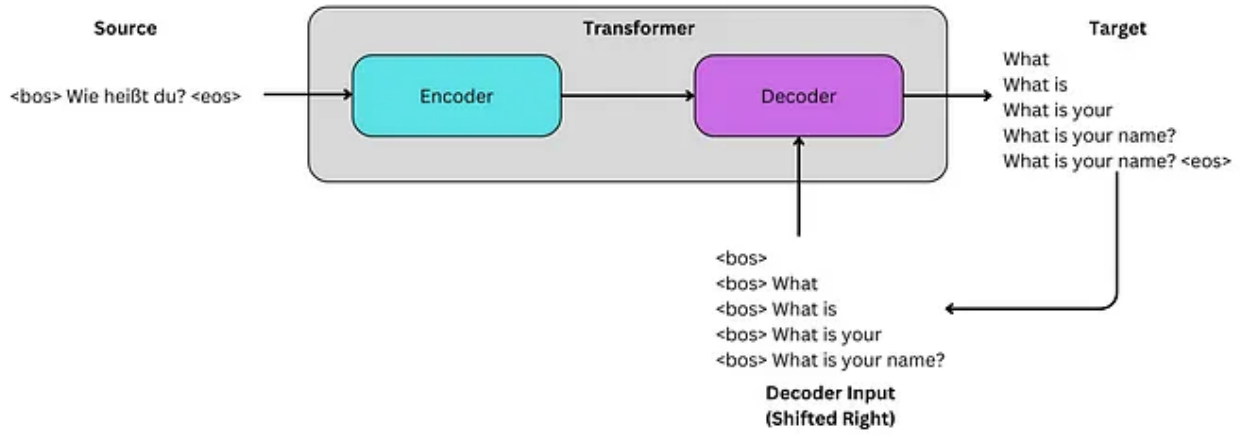


Image by Author

Preprocessing the Data

The previous article trained a transformer model to translate from German to English using a small dataset. This article will use the *Multi30k* dataset from *torchtext.datasets*. It contains a train, validation, and test set. All the custom functions to load the tokenizers, generate the vocabulary, process the data, and generate batches can be found in the appendix.

The first step is load each language's tokenizer from spaCy and to create the vocabulary for both languages using *load_vocab*. It calls on *build_vocabary*, a custom function that uses the *build_vocab_from_iterator* function from *torchtext.vocab*. The minimum frequency for a word to appear in the vocabulary is 2, and each word in the vocabulary is lowercase. The *build_vocabulary* function loads the *Multi30k* dataset to generate the vocabulary.

```
# global variables used later in the script
spacy_de, spacy_en = load_tokenizers()
```

```
vocab_src, vocab_trg = load_vocab(spacy_de, spacy_en)
```

```
Loaded English and German tokenizers.

Building German Vocabulary...
Building English Vocabulary...

Vocabulary sizes:
 Source: 8147
 Target: 6082
```

With the vocabulary generated, some of the global variables, which will be represented with capital letters, can be set. The variables below are for the indices of "*<bos>*", "*<eos>*", and "*<pad>*", which are the same for the source and target vocabularies.

```
BOS_IDX = vocab_trg['<bos>']
EOS_IDX = vocab_trg['<eos>']
PAD_IDX = vocab_trg['<pad>']
```

The dataset can be loaded for processing.

```
# raw data
train_data_raw, val_data_raw, test_data_raw = datasets.Multi30k(language_pair=("
```

Each set is a data iterator, which can be thought of as a list of tuples. Each tuple contains a German-English pair, like *("Wie heißt du?", "What is your*

name?"). This data can be tokenized and converted to the appropriate index based on the vocabulary. These actions are performed in the custom function *data_process*.

```
# processed data
train_data = data_process(train_data_raw)
val_data = data_process(val_data_raw)
test_data = data_process(test_data_raw)
```

These data iterators can now be passed to a *DataLoader* from *torch.utils.data* that can be used to generate batches during training. The *DataLoader* requires a data iterator, the batch size, and a collate function for customizing the batches. It also allows for the batches to be shuffled and for the last batch to be dropped if it is not a full batch. As a reminder, the batch size is the number of sequences used during each optimization step.

In the code below, *MAX_PADDING* represents the maximum number of tokens a sequence can have. The *pad* function from *torch.nn.functional* truncates any sequences longer than it and adds padding otherwise. This is used by the *generate_batch* function, which adds "<*bos*>", "<*eos*>", and "<*pad*>" tokens to the sequences and generates batches for training. When creating each *DataLoader*, the data iterator is converted to a map-style dataset because they can be easily shuffled and provide their size on demand.

```
MAX_PADDING = 20
BATCH_SIZE = 128

train_iter = DataLoader(to_map_style_dataset(train_data), batch_size=BATCH_SIZE,
                        shuffle=True, drop_last=True, collate_fn=generate_batch)
```

```
valid_iter = DataLoader(to_map_style_dataset(val_data), batch_size=BATCH_SIZE,
                        shuffle=True, drop_last=True, collate_fn=generate_batch)

test_iter = DataLoader(to_map_style_dataset(test_data), batch_size=BATCH_SIZE,
                       shuffle=True, drop_last=True, collate_fn=generate_batch)
```

Creating the Model

The next step is to create the model to train the data. The *make_model* function can be passed parameters to create a model, and *model.cuda()* can be used to ensure the model will train on the GPU if it is available. These values were chosen empirically.

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model = make_model(device, vocab_src, vocab_trg,
                   n_layers=3, n_heads=8, d_model=256,
                   d_ffn=512, max_length=50)
model.cuda()
```

The model's total trainable parameters can also be previewed to assess its size.

```
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 9,159,362 trainable parameters.
```

Functions for Training

To train the model, the Adam optimizer can be used with a learning rate of 0.0005, and Cross Entropy Loss can be used for the loss function. Cross Entropy Loss accepts the logits from the model as an input, converts them with a softmax function, takes the ***argmax*** of each token, and compares them to the expected target output.

```
LEARNING_RATE = 0.0005

optimizer = torch.optim.Adam(model.parameters(), lr = LEARNING_RATE)
criterion = nn.CrossEntropyLoss(ignore_index = PAD_IDX)
```

The model can be trained using the function below, which are the steps to be performed during each epoch. The model calculates the logits and updates the parameters based on the loss. At the end, the function returns the average loss for the batches in the epoch. Note that the logits and expected output are reshaped to be a single sequence rather than separate sequences. For the logits, given *(3, 10, 27)*, three sequences of ten tokens represented by a 27-element vector, the new shape would be *(30, 27)*, one large sequence. When argmax is performed, the output is a 30-element vector. The expected output, which would have a shape of *(3,10)* can also be reshaped to be a 30-element vector, and the two can be easily compared to each other.

```
def train(model, iterator, optimizer, criterion, clip):
  """
    Train the model on the given data.

    Args:
        model:        Transformer model to be trained
        iterator:     data to be trained on
        optimizer:    optimizer for updating parameters
        criterion:    loss function for updating parameters
        clip:         value to help prevent exploding gradients

    Returns:
        loss for the epoch
  """

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

  epoch_loss = 0

  # loop through each batch in the iterator
  for i, batch in enumerate(iterator):

    # set the source and target batches
    src,trg = batch

    # zero the gradients
    optimizer.zero_grad()

    # logits for each output
    logits = model(src, trg[:,:-1])

    # expected output
    expected_output = trg[:,1:]

    # 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(), clip)

    # update the weights
    optimizer.step()
```

```
      # update the loss
      epoch_loss += loss.item()

    # return the average loss for the epoch
    return epoch_loss / len(iterator)
```

The *evaluate* function below performs the same processes as the *train* function, but it does not update the weights. This will be used with the test and validation sets to see how the model generalizes.

```
def evaluate(model, iterator, criterion):
  """
    Evaluate the model on the given data.

    Args:
        model:        Transformer model to be trained
        iterator:     data to be evaluated
        criterion:    loss function for assessing outputs

    Returns:
        loss for the data
  """

  # set the model to evaluation mode
  model.eval()

  epoch_loss = 0

  # evaluate without updating gradients
  with torch.no_grad():

    # loop through each batch in the iterator
    for i, batch in enumerate(iterator):

      # set the source and target batches
      src, trg = batch


      # logits for each output
      logits = model(src, trg[:,:-1])
```

```
        # expected output
        expected_output = trg[:,1:]

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

        # update the loss
        epoch_loss += loss.item()

    # return the average loss for the epoch
    return epoch_loss / len(iterator)
```

Finally, one last function can be created to calculate how long each epoch takes.

```
def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

Training the Model

The training loop can now be created to train the model and evaluate its performance on the validation set.

```
N_EPOCHS = 10
CLIP = 1

best_valid_loss = float('inf')

# loop through each epoch
for epoch in range(N_EPOCHS):
```

```
  start_time = time.time()

  # calculate the train loss and update the parameters
  train_loss = train(model, train_iter, optimizer, criterion, CLIP)

  # calculate the loss on the validation set
  valid_loss = evaluate(model, valid_iter, criterion)

  end_time = time.time()

  # calculate how long the epoch took
  epoch_mins, epoch_secs = epoch_time(start_time, end_time)

  # save the model when it performs better than the previous run
  if valid_loss < best_valid_loss:
    best_valid_loss = valid_loss
    torch.save(model.state_dict(), 'transformer-model.pt')

  print(f'Epoch: {epoch+1:02} | Time: {epoch_mins}m {epoch_secs}s')
  print(f'\tTrain Loss: {train_loss:.3f} | Train PPL: {math.exp(train_loss):7.3f
  print(f'\t Val. Loss: {valid_loss:.3f} |  Val. PPL: {math.exp(valid_loss):7.3f
```

```
Epoch: 01 | Time: 0m 21s
 Train Loss: 4.534 | Train PPL:  93.169
  Val. Loss: 3.474 |  Val. PPL:  32.280
Epoch: 02 | Time: 0m 13s
 Train Loss: 3.219 | Train PPL:  24.992
  Val. Loss: 2.735 |  Val. PPL:  15.403
Epoch: 03 | Time: 0m 13s
 Train Loss: 2.544 | Train PPL:  12.733
  Val. Loss: 2.225 |  Val. PPL:   9.250
Epoch: 04 | Time: 0m 14s
 Train Loss: 2.096 | Train PPL:   8.131
  Val. Loss: 1.980 |  Val. PPL:   7.246
Epoch: 05 | Time: 0m 13s
 Train Loss: 1.801 | Train PPL:   6.055
  Val. Loss: 1.829 |  Val. PPL:   6.229
Epoch: 06 | Time: 0m 14s
 Train Loss: 1.588 | Train PPL:   4.896
  Val. Loss: 1.743 |  Val. PPL:   5.717
Epoch: 07 | Time: 0m 13s
 Train Loss: 1.427 | Train PPL:   4.166
  Val. Loss: 1.700 |  Val. PPL:   5.476
Epoch: 08 | Time: 0m 13s
 Train Loss: 1.295 | Train PPL:   3.650
  Val. Loss: 1.679 |  Val. PPL:   5.358
```

```
Epoch: 09 | Time: 0m 13s
 Train Loss: 1.184 | Train PPL:   3.268
  Val. Loss: 1.677 |  Val. PPL:   5.349
Epoch: 10 | Time: 0m 13s
 Train Loss: 1.093 | Train PPL:   2.984
  Val. Loss: 1.677 |  Val. PPL:   5.351
```

The accuracy can also be assessed on the test set using the *evaluate* function before assessing the results.

```
# load the weights
model.load_state_dict(torch.load('transformer-model.pt'))

# calculate the loss on the test set
test_loss = evaluate(model, test_iter, criterion)

print(f'Test Loss: {test_loss:.3f} | Test PPL: {math.exp(test_loss):7.3f}')
```

```
Test Loss: 1.692 | Test PPL:   5.430
```

While the loss has decreased significantly, there is no indication of how successful the model is at translating from German to English. This can be assessed in two ways. The first is to provide it with a sentence and to preview its translation during inference. The second is to compute its accuracy via another metric, like BLEU, which is a standard for translation tasks.

Inference

Real-time translation can be performed by passing a sentence to the function below. It will be tokenized and passed through the model,

generating one token at a time. Once the "*<eos>*" token occurs, the output will be returned.

```
def translate_sentence(sentence, model, device, max_length = 50):
  """
    Translate a German sentence to its English equivalent.

    Args:
        sentence:     German sentence to be translated to English; list or str
        model:        Transformer model used for translation
        device:       device to perform translation on
        max_length:   maximum token length for translation

    Returns:
        src:                  return the tokenized input
        trg_input:            return the input to the decoder before the final o
        trg_output:           return the final translation, shifted right
        attn_probs:           return the attention scores for the decoder heads
        masked_attn_probs:    return the masked attention scores for the decoder
  """

  model.eval()

  # tokenize and index the provided string
  if isinstance(sentence, str):
    src = ['<bos>'] + [token.text.lower() for token in spacy_de(sentence)] +  ['
  else:
    src = ['<bos>'] + sentence + ['<eos>']

  # convert to integers
  src_indexes = [vocab_src[token] for token in src]

  # convert list to tensor
  src_tensor = torch.tensor(src_indexes).int().unsqueeze(0).to(device)

  # set <bos> token for target generation
  trg_indexes = [vocab_trg.get_stoi()['<bos>']]

  # generate new tokens
  for i in range(max_length):

    # convert the list to a tensor
    trg_tensor = torch.tensor(trg_indexes).int().unsqueeze(0).to(device)

    # generate the next token
```

```
    with torch.no_grad():

      # generate the logits
      logits = model.forward(src_tensor, trg_tensor)

      # select the newly predicted token
      pred_token = logits.argmax(2)[:,-1].item()

      # if <eos> token or max length, stop generating
      if pred_token == vocab_trg.get_stoi()['<eos>'] or i == (max_length-1):

        # decoder input
        trg_input = vocab_trg.lookup_tokens(trg_indexes)

        # decoder output
        trg_output = vocab_trg.lookup_tokens(logits.argmax(2).squeeze(0).tolist(

        return src, trg_input, trg_output, model.decoder.attn_probs, model.decod

      # else, continue generating
      else:
        # add the token
        trg_indexes.append(pred_token)
```

An example from the training set can be used to ensure the resulting visualizations demonstrate how attention works.

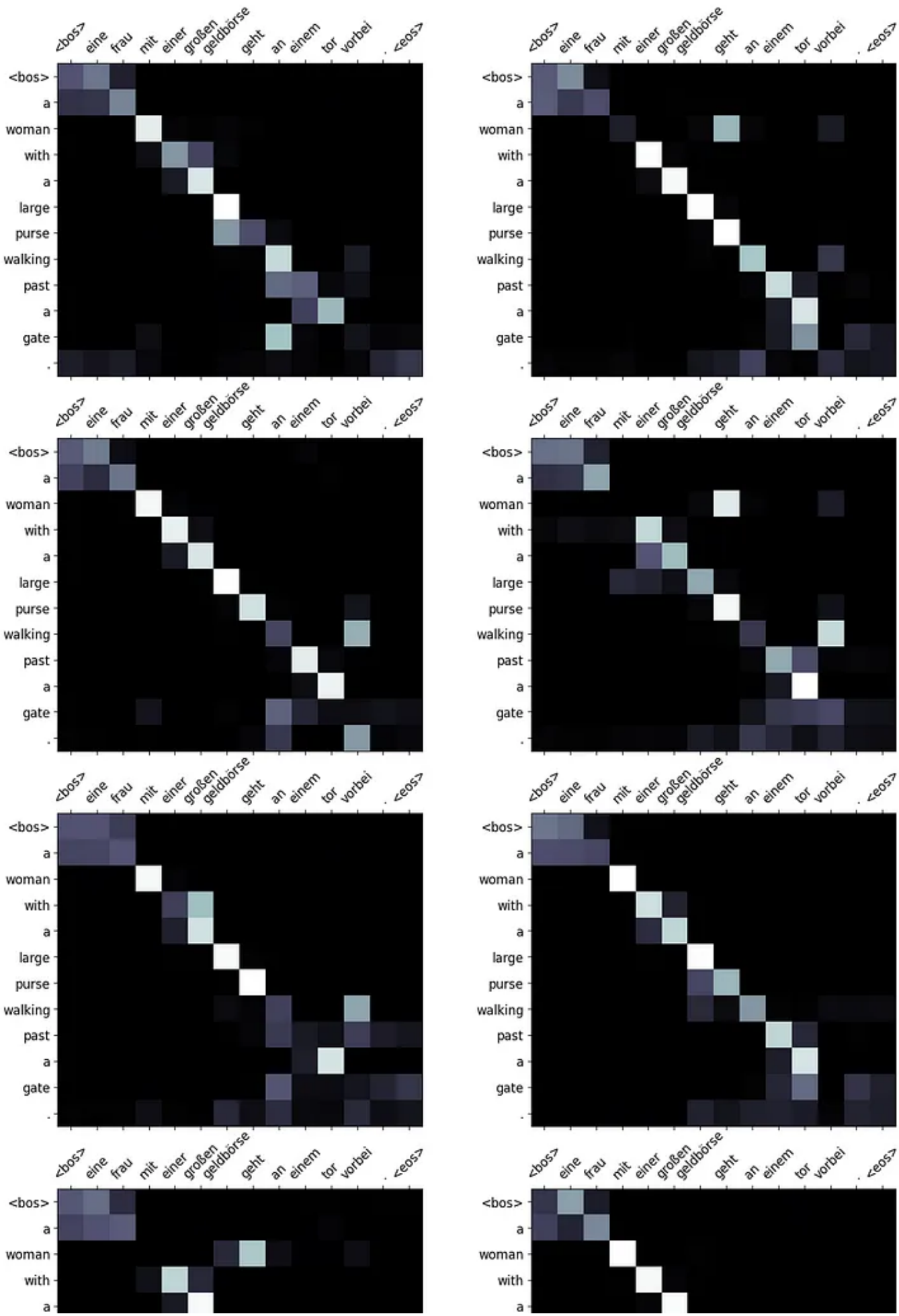
```
# 'a woman with a large purse is walking by a gate'
src = ['eine', 'frau', 'mit', 'einer', 'großen', 'geldbörse', 'geht', 'an', 'ein

src, trg_input, trg_output, attn_probs, masked_attn_probs = translate_sentence(s

print(f'source = {src}')
print(f'target input = {trg_input}')
print(f'target output = {trg_output}')
```

```
source = ['<bos>', 'eine', 'frau', 'mit', 'einer', 'großen', 'geldbörse', 'geht'
target input = ['<bos>', 'a', 'woman', 'with', 'a', 'large', 'purse', 'walking',
```

```
target output = ['a', 'woman', 'with', 'a', 'large', 'purse', 'walking', 'past',
```

The target output is the model's prediction for the source sequence, and the target input is the final input to the decoder before the end-of-sequence token is generated. This is what is visualized with the source sequence in the attention matrix.

```
display_attention(src, trg_input, attn_probs)
```


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

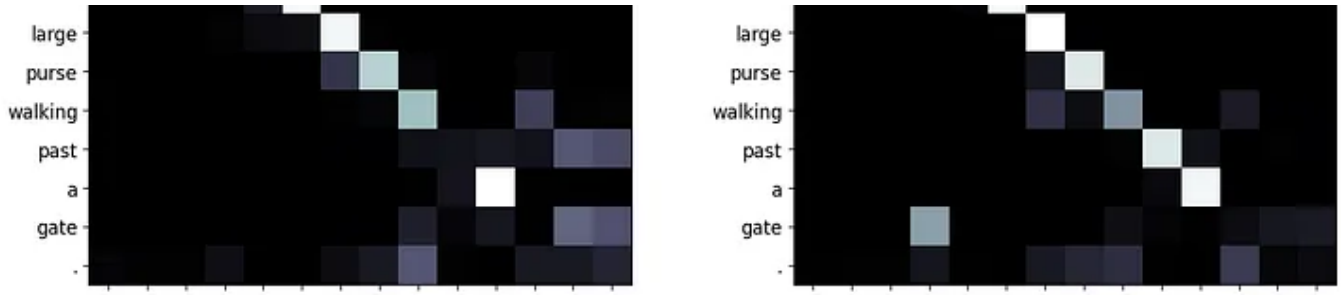
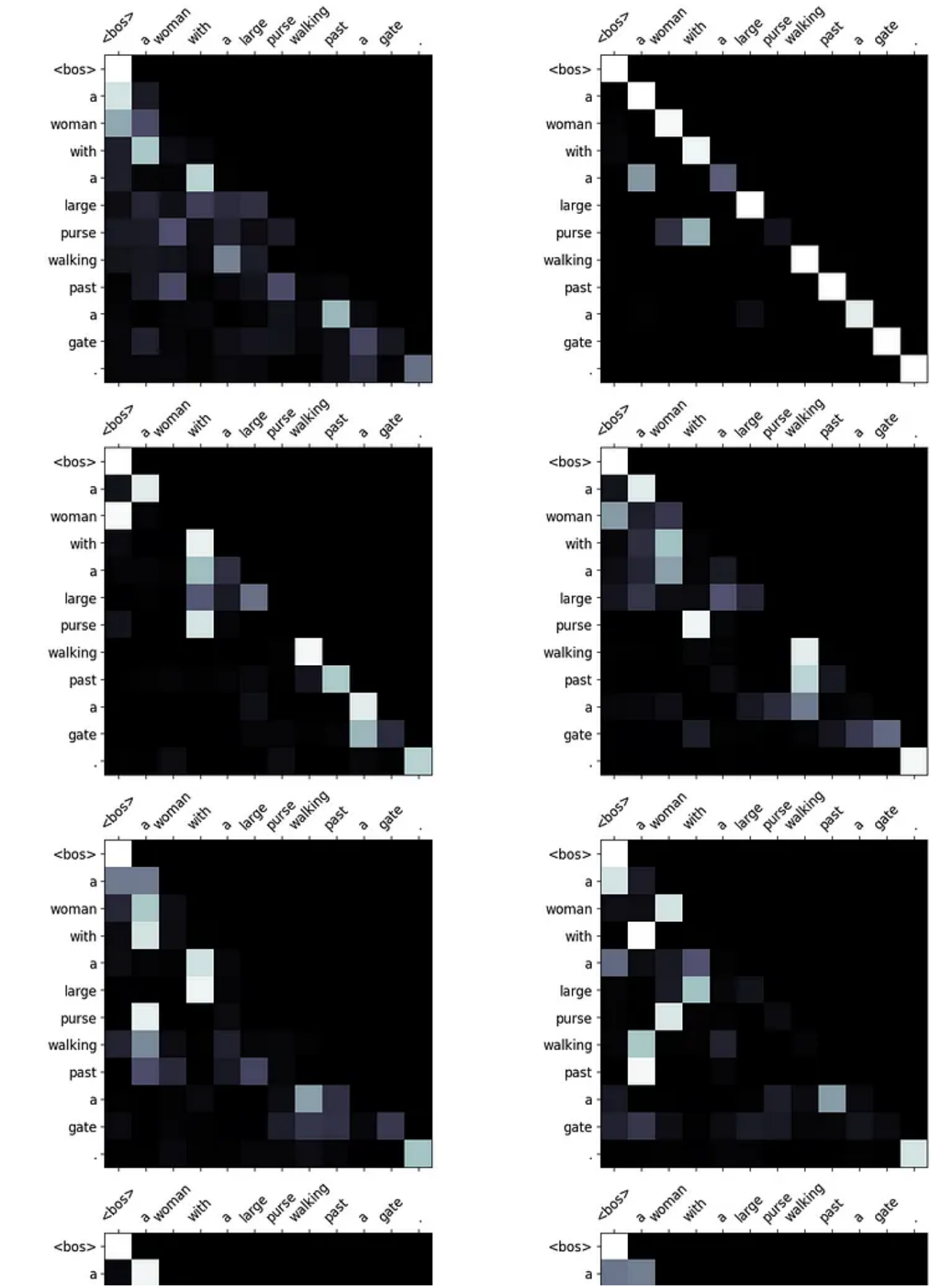


Image by Author

The masked attention matrix can also be viewed with the target input.

```
display_attention(trg_input, trg_input, masked_attn_probs)
```


<bos>
a
woman
with
a
large
purse
walking
past
a
gate
.

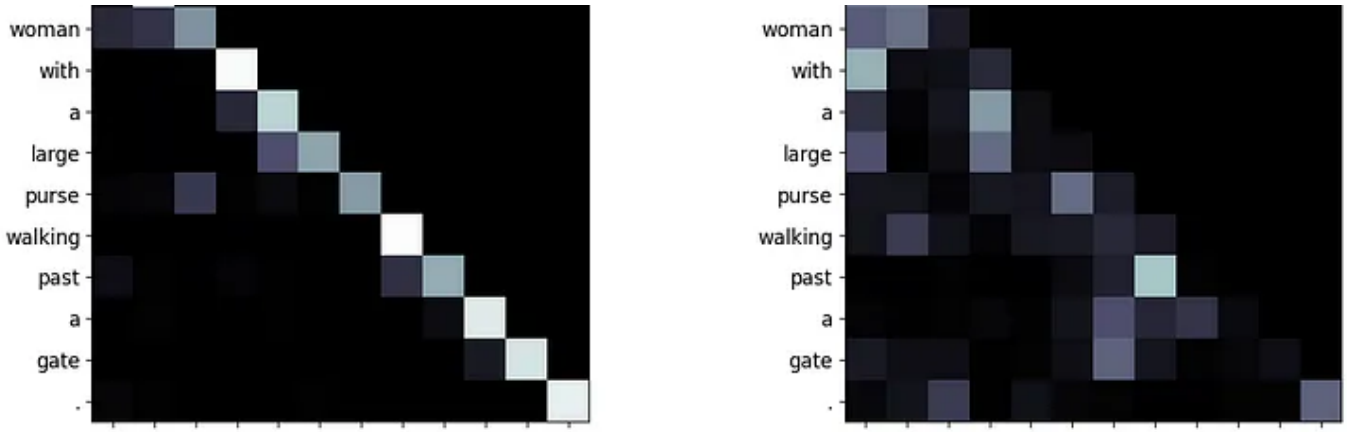


Image by Author

Although these are useful visualizations, a sentence that isn't in the training set can be used to determine the model's usefulness for actual translations. The following two examples are from the test set.

```
# A guy works on a building
src = 'Ein Typ arbeitet an einem Gebäude.'

src, trg_input, trg_output, attn_probs, masked_attn_probs = translate_sentence(s

print(f'source = {src}')
print(f'target input = {trg_input}')
print(f'target output = {trg_output}')
```

```
source = ['<bos>', 'ein', 'typ', 'arbeitet', 'an', 'einem', 'gebäude', '.', '<eo
target input = ['<bos>', 'a', 'guy', 'working', 'on', 'a', 'building', '.']
target output = ['a', 'guy', 'working', 'on', 'a', 'building', '.', '<eos>']
```

The first example is a valid translation, but the second example is not.

```
#  A mother teaches her two young boys to fish off of a rocky coast into very bl
src = 'Eine Mutter bringt ihren zwei kleinen Söhnen an einer felsigen Küste mit

src, trg_input, trg_output, attn_probs, masked_attn_probs = translate_sentence(s

print(f'source = {src}')
print(f'target input = {trg_input}')
print(f'target output = {trg_output}')
```

```
source = ['<bos>', 'eine', 'mutter', 'bringt', 'ihren', 'zwei', 'kleinen', 'söhn
target input = ['<bos>', 'a', 'mother', 'is', 'training', 'her', 'two', 'small',
target output = ['a', 'mother', 'is', 'training', 'her', 'two', 'small', 'sons',
```

To assess how accurate the model is on the entirety of the test set, the BLEU score can now be calculated.

BLEU Score

Bilingual evaluation understudy (BLEU) is a commonly used metric to evaluate machine translation models. The score ranges between 0 and 1, with a 1 meaning the prediction and expected translation are identical.

According to Google's AutoML documentation, a BLEU score's value can have the following meanings (in terms of percentage):

- < 10: almost useless
- 10-19: hard to understand
- 20-29: understandable but significant grammatical errors
- 30-39: understandable to good

- 40-49: high quality
- 50-59: high quality, adequate, and fluent
- > 60: better than human quality

To calculate the BLEU score, the model's predictions and their expected values need to be generated. This can be completed with the function below, which utilizes the ***translate_sentence*** function.

```
def compute_metrics(model, iterator):
  """
    Generate predictions for the provided iterator.

    Args:
        model:        Transformer model to be trained
        iterator:     data to be evaluated

    Returns:
        predictions:  list of predictions, which are tokenized strings
        labels:       list of expected output, which are tokenized strings
  """

  # set the model to evaluation mode
  model.eval()

  predictions = []
  labels = []

  # evaluate without updating gradients
  with torch.no_grad():

    # loop through each batch in the iterator
    for i, batch in enumerate(iterator):

      # set the source and target batches
      src, trg = batch

      # predict the output
      src_out, trg_input, trg_output, attn_probs, masked_attn_probs = translate_

      # prediction | remove <eos> token
```

```
        predictions.append(trg_output[:-1])

        # expected output | add extra dim for calculation
        labels.append([vocab_trg.lookup_tokens(trg.tolist())])

    # return the average loss for the epoch
    return predictions, labels
```

The *test_data* generated earlier, which contains tokenized sequences, can be passed to the *compute_metrics* function. The predictions and labels can then be passed to *bleu_score* from *torchtext.data.metrics* to calculate the BLEU score.

```
from torchtext.data.metrics import bleu_score
bleu_score(predictions, labels)
```

```
0.3588869571685791
```

This output indicates the translations are understandable to good, which is an acceptable outcome for this tutorial. With this example complete, the Implemented Transformer series is at its end.

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

References

1. Deepak Saini's Transformer Implementation

2. Harvard's The Annotated Transformer

Appendix

Packages

```
!pip install -q portalocker

# importing required libraries
import math
import copy
import time
import random
import spacy
import numpy as np
import os

# torch packages
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch import Tensor
import torch.optim as optim

# load and build datasets
import torchtext
from torchtext.data.functional import to_map_style_dataset
from torch.nn.functional import pad
from torch.utils.data import DataLoader
from torchtext.vocab import build_vocab_from_iterator
import torchtext.datasets as datasets
import portalocker

# visualization packages
from mpl_toolkits import mplot3d
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
```

Loading the Tokenizers

This function downloads the German and English tokenizers provided by spaCy.

```
def load_tokenizers():
  """
    Load the German and English tokenizers provided by spaCy.

    Returns:
        spacy_de:     German tokenizer
        spacy_en:     English tokenizer
  """
  try:
    spacy_de = spacy.load("de_core_news_sm")
  except OSError:
    os.system("python -m spacy download de_core_news_sm")
    spacy_de = spacy.load("de_core_news_sm")

  try:
    spacy_en = spacy.load("en_core_web_sm")
  except OSError:
    os.system("python -m spacy download en_core_web_sm")
    spacy_en = spacy.load("en_core_web_sm")

  print("Loaded English and German tokenizers.")
  return spacy_de, spacy_en
```

Tokenize the Sequences

This function uses a spaCy tokenizer to tokenize a provided sequence.

```
def tokenize(text: str, tokenizer):
  """
    Split a string into its tokens using the provided tokenizer.

    Args:
```

```
            text:         string
            tokenizer:    tokenizer for the language

        Returns:
            tokenized list of strings
    """
    return [tok.text.lower() for tok in tokenizer.tokenizer(text)]
```

Yield Tokens

This function calls on the provided tokenizer to yield the tokens for the correct language. If ***index = 0***, then German is tokenized. If ***index = 1***, English is tokenized. Each tuple from the data iterator contains a German-English pair, like ***(“Wie heißt du?”, “What is your name?”)***.

```
def yield_tokens(data_iter, tokenizer, index: int):
    """
        Return the tokens for the appropriate language.

        Args:
            data_iter:    text here
            tokenizer:    tokenizer for the language
            index:        index of the language in the tuple | (de=0, en=1)

        Yields:
            sequences based on index
    """
    for from_tuple in data_iter:
        yield tokenizer(from_tuple[index])
```

Building the Vocabulary

This function accepts the German and English spaCy tokenizers as parameters, and it accepts the minimum frequency required for a word to be

included in the vocabulary. The *tokenize_de* and *tokenize_en* functions call on *tokenize* and pass the respective tokenizer for each language.

The German-English dataset is loaded using *datasets.Multi30k(language_pair = ("de", "en"))*. This returns train, validation, and test sets that can be iterated over to generate the vocabulary.

The *build_vocab_from_iterator* function from *torchtext.vocab* is used to build the vocabulary with all these components. It uses *yield_tokens* to generate the tokens for each sequence. *yield_tokens* takes *train* + *val* + *test*, which creates a single data iterator with all the sources, the tokenization function for the respective language (*tokenize_de* or *tokenize_en*), and the appropriate index for the language in the iterator (0 for German and 1 for English). It also requires the minimum frequency and the special tokens. The special tokens are

- "<*bos*>" for the start of sequences
- "<*eos*>" for the end of sequences
- "<*pad*>" for the padding
- "<*unk*>" for tokens that are not present in the vocabulary

```
def build_vocabulary(spacy_de, spacy_en, min_freq: int = 2):

  def tokenize_de(text: str):
    """
      Call the German tokenizer.

      Args:
          text:         string
          min_freq:     minimum frequency needed to include a word in the vocabu

      Returns:
```

```
            tokenized list of strings
    """
    return tokenize(text, spacy_de)

def tokenize_en(text: str):
    """
      Call the English tokenizer.

      Args:
          text:         string

      Returns:
          tokenized list of strings
    """
    return tokenize(text, spacy_en)

print("Building German Vocabulary...")

# load train, val, and test data pipelines
train, val, test = datasets.Multi30k(language_pair=("de", "en"))

# generate source vocabulary
vocab_src = build_vocab_from_iterator(
      yield_tokens(train + val + test, tokenize_de, index=0), # tokens for eac
      min_freq=min_freq,
      specials=["<bos>", "<eos>", "<pad>", "<unk>"],
)

print("Building English Vocabulary...")

# generate target vocabulary
vocab_trg = build_vocab_from_iterator(
      yield_tokens(train + val + test, tokenize_en, index=1), # tokens for eac
      min_freq=2, #
      specials=["<bos>", "<eos>", "<pad>", "<unk>"],
)

# set default token for out-of-vocabulary words (OOV)
vocab_src.set_default_index(vocab_src["<unk>"])
vocab_trg.set_default_index(vocab_trg["<unk>"])

return vocab_src, vocab_trg
```

Load the Vocabulary

This function generates and saves the vocabulary if it has not been created yet. Otherwise, it loads the vocabulary. It requires the spaCy tokenizers and minimum frequency as input.

```
def load_vocab(spacy_de, spacy_en, min_freq: int = 2):
  """
    Args:
        spacy_de:     German tokenizer
        spacy_en:     English tokenizer
        min_freq:     minimum frequency needed to include a word in the vocabula

    Returns:
        vocab_src:    German vocabulary
        vocab_trg:     English vocabulary
  """

  if not os.path.exists("vocab.pt"):
    # build the German/English vocabulary if it does not exist
    vocab_src, vocab_trg = build_vocabulary(spacy_de, spacy_en, min_freq)
    # save it to a file
    torch.save((vocab_src, vocab_trg), "vocab.pt")
  else:
    # load the vocab if it exists
    vocab_src, vocab_trg = torch.load("vocab.pt")

  print("Finished.\nVocabulary sizes:")
  print("\tSource:", len(vocab_src))
  print("\tTarget:", len(vocab_trg))
  return vocab_src, vocab_trg
```

Indexing Sequences

This function accepts the raw German-English tuples, tokenizes them, converts them to tensors, and returns a list of tuples.

```
def data_process(raw_data):
  """
    Process raw sentences by tokenizing and converting to integers based on
    the vocabulary.

    Args:
        raw_data:     German-English sentence pairs
    Returns:
        data:         tokenized data converted to index based on vocabulary
  """

  data = []
  # loop through each sentence pair
  for (raw_de, raw_en) in raw_data:
    # tokenize the sentence and convert each word to an integers
    de_tensor_ = torch.tensor([vocab_src[token.text.lower()] for token in spacy_
    en_tensor_ = torch.tensor([vocab_trg[token.text.lower()] for token in spacy_

    # append tensor representations
    data.append((de_tensor_, en_tensor_))
  return data
```

Generating Batches

This function is used to add start, end, and pad tokens to the indexed sequences.

```
def generate_batch(data_batch):
  """
    Process indexed-sequences by adding <bos>, <eos>, and <pad> tokens.

    Args:
        data_batch:     German-English indexed-sentence pairs

    Returns:
        two batches:    one for German and one for English
  """
  de_batch, en_batch = [], []
```

```
    # for each sentence
    for (de_item, en_item) in data_batch:
      # add <bos> and <eos> indices before and after the sentence
      de_temp = torch.cat([torch.tensor([BOS_IDX]), de_item, torch.tensor([EOS_IDX
      en_temp = torch.cat([torch.tensor([BOS_IDX]), en_item, torch.tensor([EOS_IDX

      # add padding
      de_batch.append(pad(de_temp,(0, # dimension to pad
                                MAX_PADDING - len(de_temp), # amount of padding to a
                               ),value=PAD_IDX,))

      # add padding
      en_batch.append(pad(en_temp,(0, # dimension to pad
                                MAX_PADDING - len(en_temp), # amount of padding to a
                               ),
                               value=PAD_IDX,))

    return torch.stack(de_batch), torch.stack(en_batch)
```

Displaying Attention

This function can display self-attention, masked attention, and source-target attention.

```
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()
```

Transformers NLP Translation Machine Learning 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 Word2ve

· 15 min rea

View list

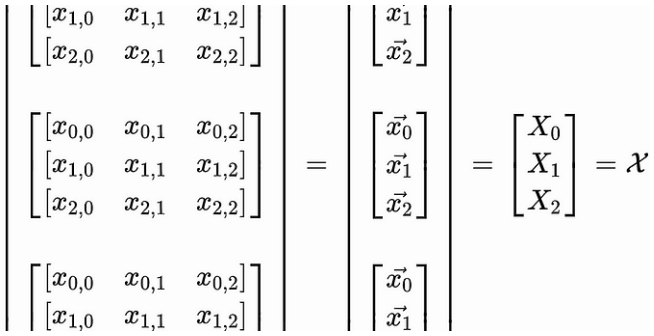
Written by Hunter Phillips

Following

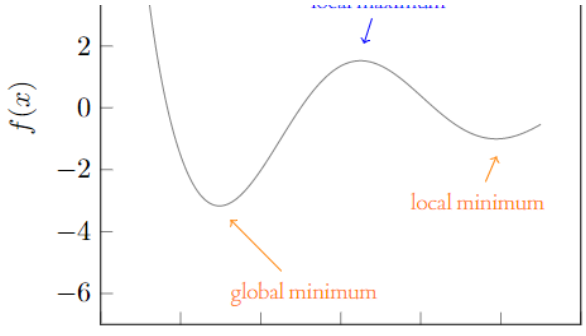
219 Followers

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

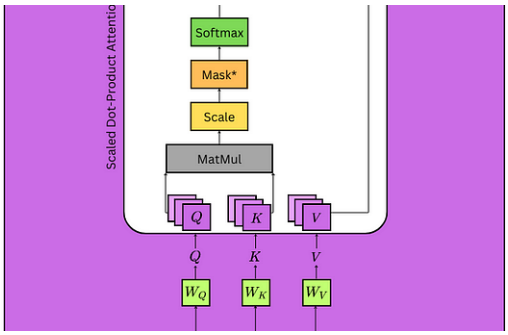
273 5

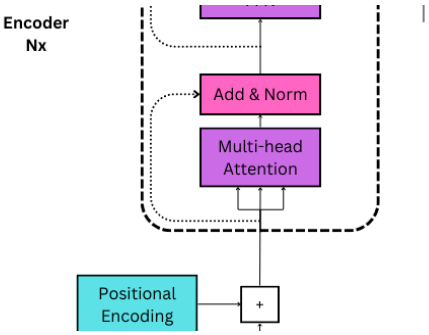
A Simple Introduction to Gradient Descent

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

10 min read · May 18

108





Hunter Phillips

Multi-Head Attention

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

25 min read · May 9

167

Hunter Phillips

Position-Wise Feed-Forward Network (FFN)

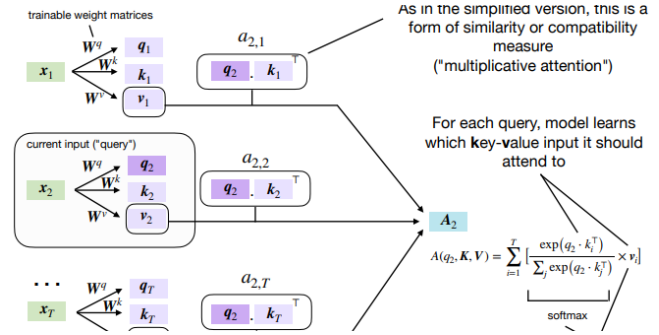
This is the fourth article in The Implemented Transformer series. The Position-wise Feed-...

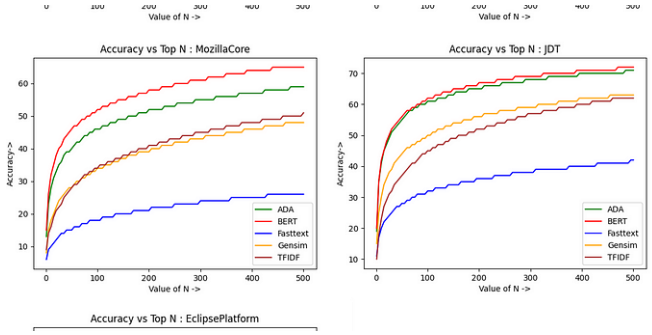
9 min read · May 9

155

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...

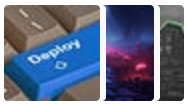
4 min read · Sep 19

3 1

Lists

Natural Language Processing

669 stories · 283 saves



Predictive Modeling w/ Python

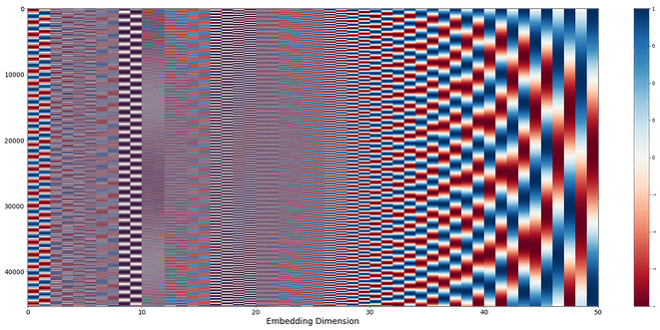
20 stories · 452 saves

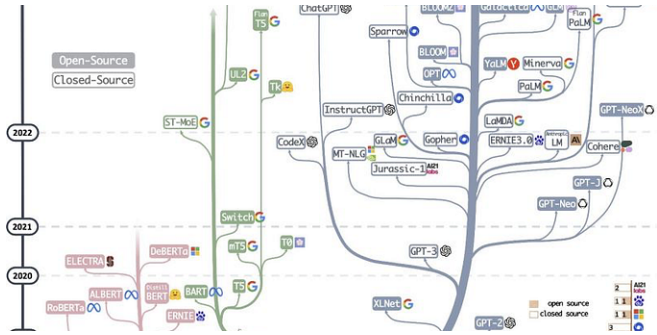
AI Regulation

6 stories · 138 saves

ChatGPT prompts

24 stories · 459 saves





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

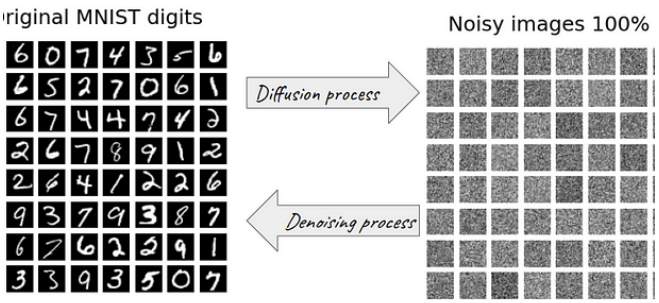
David Shapiro

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



Antony M. Gitau

A friendly Introduction to Denoising Diffusion Probabilistic...

I recently attended a Nordic probabilistic AI school, ProbAI 2023, which inspired my...

9 min read · Jul 9

53

See more recommendations