How to Implement Transformer in PyTorch

Germen to English Machine Translation

Jioh Lee

2/20/2024

INFONET LAB.

Index

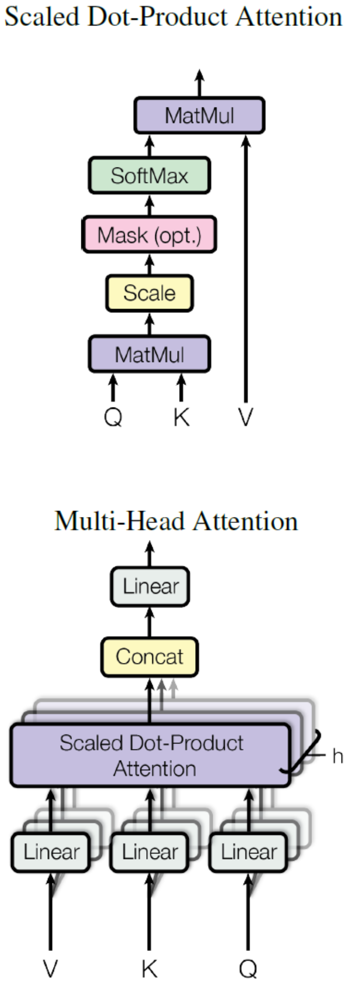
- Transformer Recap
- Details of implementation in PyTorch Library: nn.Transformer()
- Implementation from Scratch
- Full Code, Colab, Machine Translation: German to English

Transformer Recap

Transformer Architecture

Vaswani, Ashish, et al. "Attention is all you need." *Advances in neural information processing systems* 30 (2017).

- **Encoder**
  - Stack of Encoder Layers
    - In each Layers:
      - Multi-Head Attention (Self Attention)
      - Residual connection added to output and LayerNorm applied
      - Feed-Forward Network
      - Residual connection added to output and LayerNorm applied
- **Decoder**
  - Stack of Decoder Layers
    - In each Layers:
      - Multi-Head Attention (Self Attention)
      - Residual connection added to output LayerNorm applied
      - Multi-Head Attention (Cross Attention)
      - Residual connection added to output LayerNorm applied
      - Feed-Forward Network
      - Residual connection added to output LayerNorm applied



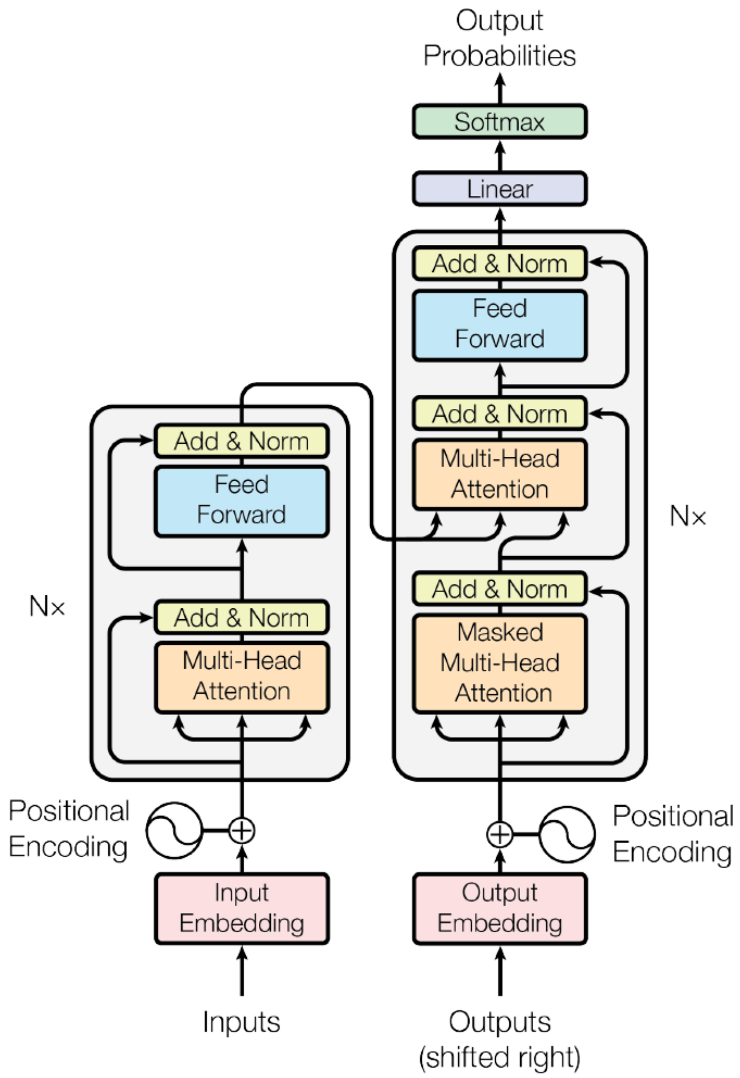


Figure 1: The Transformer - model architecture.

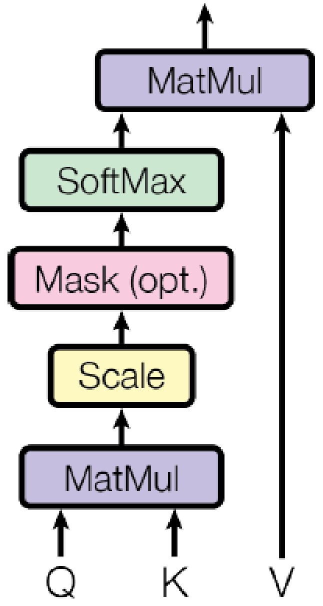
Transformer Architecture

Vaswani, Ashish, et al. "Attention is all you need." *Advances in neural information processing systems* 30 (2017).

▪ Scaled Dot-product Attention

- With inputs (Q: Query, K: Key, V: Value)
  - $Attention = softmax\left(\frac{Q \cdot K^T}{\sqrt{d_k}} + mask\right)V$
- Scale?
  - Re-scale attention score using value $\sqrt{d_k}$.
  - d_k is dimension of each vector in K.
- Why Scale?
  - Let's say we have $Q = [Q_1, Q_2, Q_3]^T, K = [K_1, K_2, K_3]^T, Q_i, K_j \in \mathbb{R}^{d_k}, i, j \in \{1, 2, 3\}$.
  - Assume elements of Q_i, K_j are independent random variables with mean 0 variance 1.
  - Its dot product $Q_i K_j^T$ has mean 0 and variance d_k.
  - Larger magnitude of softmax function input → Smaller gradient of softmax function → Less training efficiency
  - By scaling, reduce magnitude of softmax function input.

Scaled Dot-Product Attention

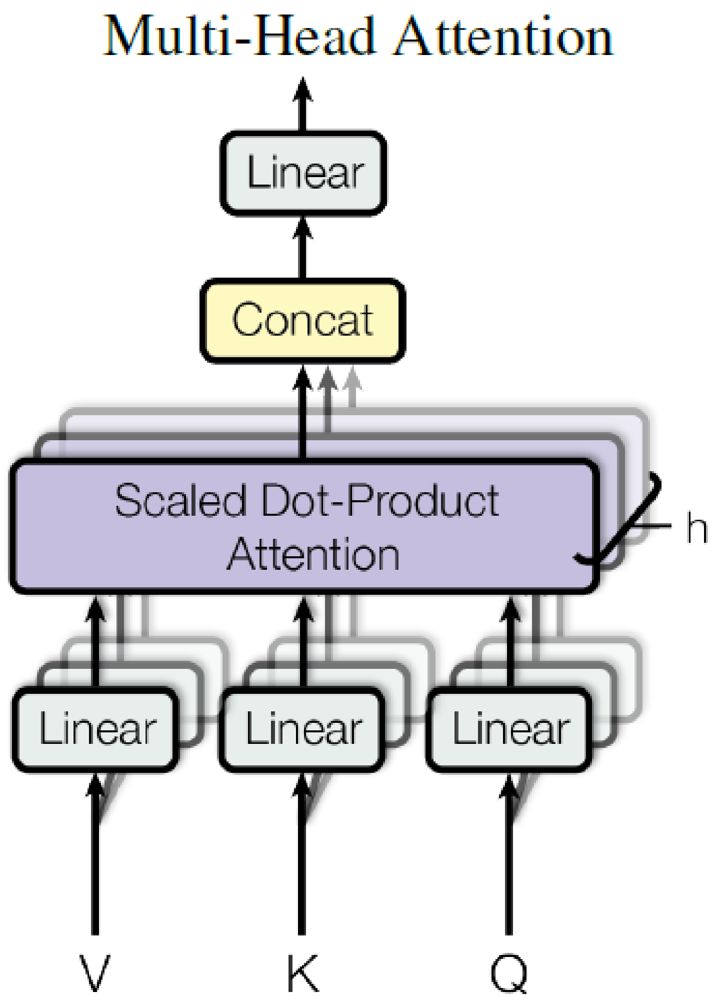


Transformer Architecture

Vaswani, Ashish, et al. "Attention is all you need." *Advances in neural information processing systems* 30 (2017).

▪ Multi-Head Attention

- Formula
  - $head_i = softmax\left(\frac{QK^T}{\sqrt{d_k}} + mask\right)V$
    - Each Query, Key, Value are projected into new Q, K, V by W
    - $W^Q, \mathrm{W}^{\mathrm{K}} \in \mathbb{R}^{d_{model} \times d_k}, \mathrm{W}^{\mathrm{V}} \in \mathbb{R}^{d_{model} \times d_v}$
  - $MultiHead(Q, K, V) = Concat(head_1, \ldots, head_h)W^O$
    - Each output heads are concatenated and projected by W
    - $W^O \in \mathbb{R}^{h * d_v \times d_{model}}$
  - $d_k = d_v = \frac{d_{model}}{h}$, h means number of heads
  - It can be calculated in parallel, without concatenation.



Transformer Architecture

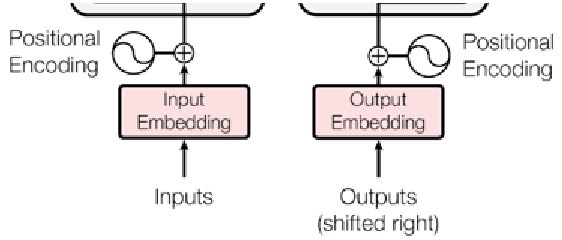
Vaswani, Ashish, et al. "Attention is all you need." *Advances in neural information processing systems* 30 (2017).

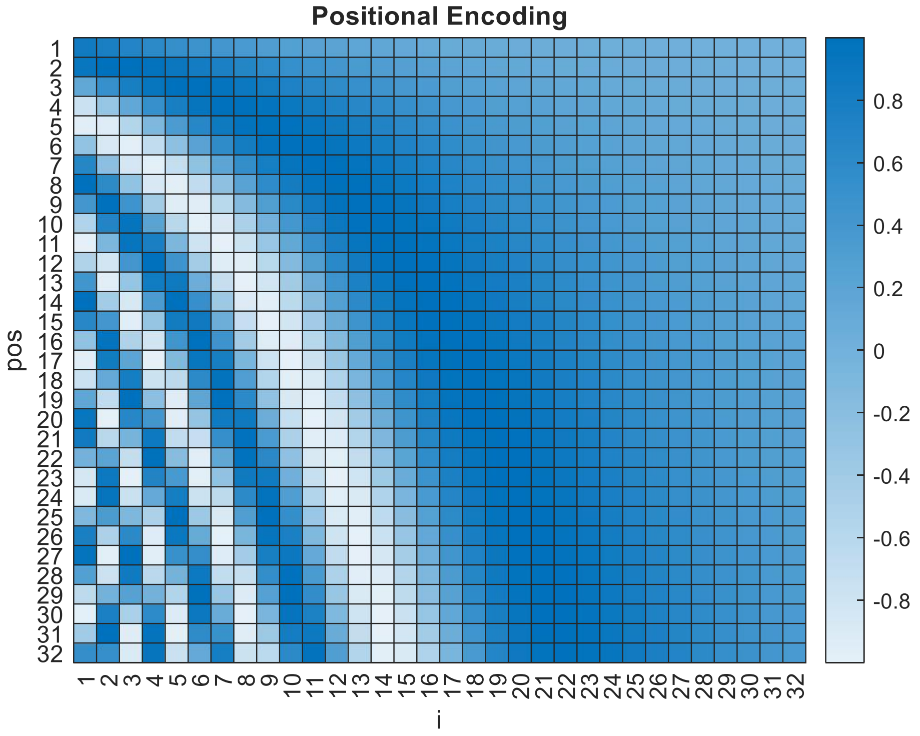
▪ Position-wise Feed-Forward Network

- Same parameters across all positions in a sentence.
  - Forward operation to [batch_size, sequence_length, embedding_dimension]
- Different parameters across all layers.
- $FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2$
  - $W_1 \in \mathbb{R}^{d_{model} \times d_{ff}}, W_2 \in \mathbb{R}^{d_{ff} \times d_{model}}$

▪ Positional Encoding

- Inject some information about the position of the tokens in the sequence.
- In the original paper, they use:
  - $PE(pos, 2i) = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right), PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$
  - pos means position of tokens, i means dimension index.
- In practice, we can use learnable embedding. → nn.Embbeding().





Transformer Architecture

Vaswani, Ashish, et al. "Attention is all you need." *Advances in neural information processing systems* 30 (2017).

- **Masking**
  - (1) Encoder Self Attention
  - We have
    - Q, K, V: output of previous encoder layer
  - Then we get attention score as follow:
    - $QK^T = \begin{bmatrix} Q_1K_1 & Q_1K_2 & Q_1K_3 \\ Q_2K_1 & Q_2K_2 & Q_2K_3 \\ Q_3K_1 & Q_3K_2 & Q_3K_3 \end{bmatrix}$
  - Assume Q_3 and K_3 is from padding token, should be masked, then we get:
    - $\mathrm{softmax}\left(\mathrm{QK^T} + \begin{bmatrix} 0 & 0 & -\infty \\ 0 & 0 & -\infty \\ 0 & 0 & -\infty \end{bmatrix}\right) = \begin{bmatrix} e_{11} & e_{12} & 0 \\ e_{21} & e_{22} & 0 \\ e_{31} & e_{32} & 0 \end{bmatrix}$

→ In nn.Transformer().forward() "src_key_padding_mask" argument do this.

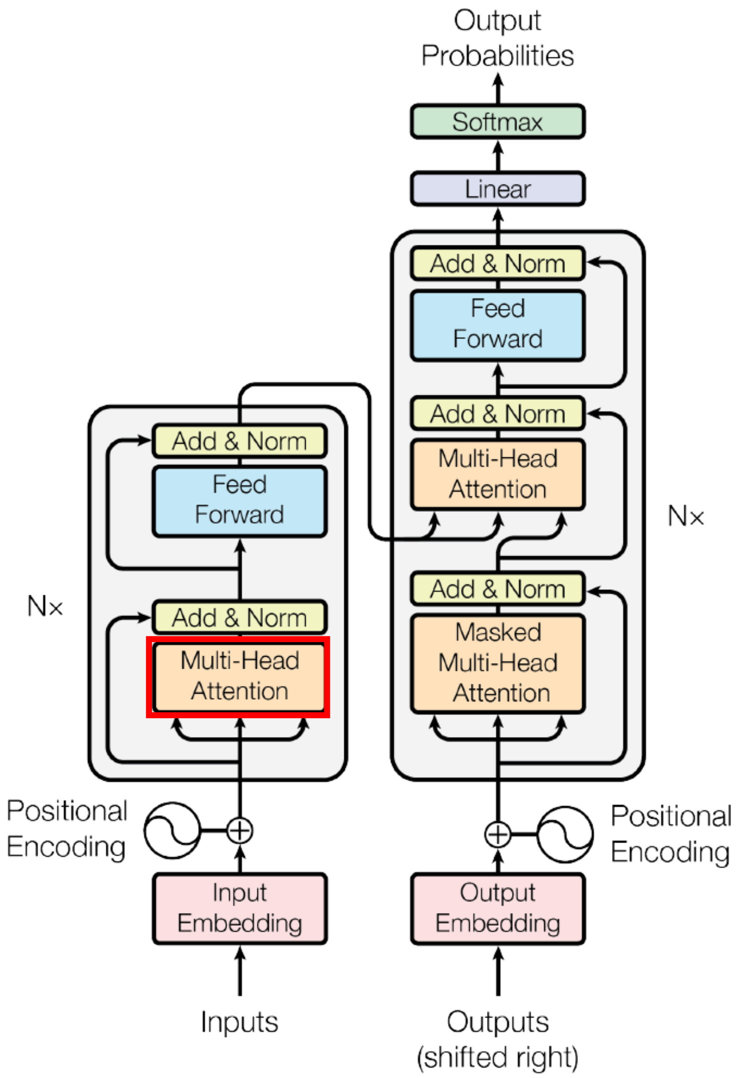


Figure 1: The Transformer - model architecture.

Transformer Architecture

Vaswani, Ashish, et al. "Attention is all you need." *Advances in neural information processing systems* 30 (2017).

▪ Masking

- (2) Decoder Self Attention
- We have
  - Q, K, V: output of previous decoder layer
- Then we get attention score as follow:
  - $QK^T = \begin{bmatrix} Q_1K_1 & Q_1K_2 & Q_1K_3 \\ Q_2K_1 & Q_2K_2 & Q_2K_3 \\ Q_3K_1 & Q_3K_2 & Q_3K_3 \end{bmatrix}$
- Assume Q_3 and K_3 is from padding token, should be masked
- Q_i should pay attention to K_j, $j \le i$, then indices of $j > i$ should be mask
  - $\mathrm{softmax}\left(\mathrm{QK}^{\mathrm{T}} + \begin{bmatrix} 0 & 0 & -\infty \\ 0 & 0 & -\infty \\ 0 & 0 & -\infty \end{bmatrix} + \begin{bmatrix} 0 & -\infty & -\infty \\ 0 & 0 & -\infty \\ 0 & 0 & 0 \end{bmatrix}\right) = \begin{bmatrix} 1 & 0 & 0 \\ e_{21} & e_{22} & 0 \\ e_{31} & e_{32} & 0 \end{bmatrix}$

  → In nn.Transformer().forward(), ”tgt_key_padding_mask” and “tgt_mask” do this.

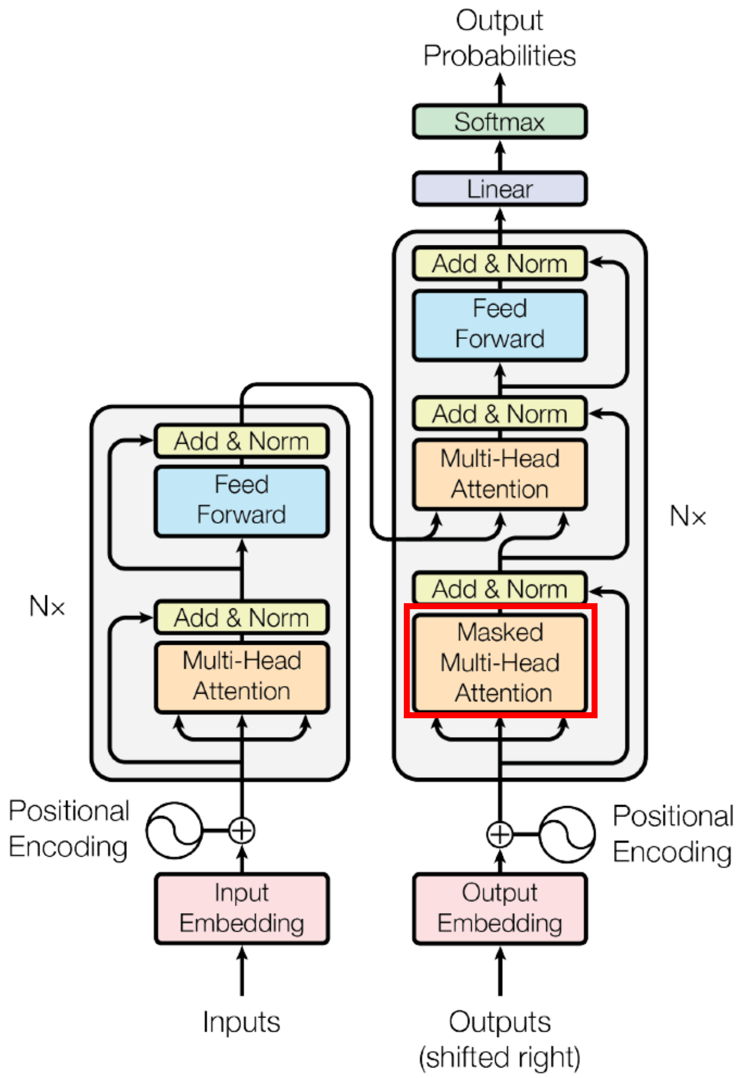


Figure 1: The Transformer - model architecture.

Transformer Architecture

Vaswani, Ashish, et al. "Attention is all you need." *Advances in neural information processing systems* 30 (2017).

▪ Masking

- (3) Decoder Cross Attention
- We have
  - ○ Q: output of previous decoder layer
  - ○ K, V: output of Encoder
- Then we get attention score as follow:
  - ○ $QK^T = \begin{bmatrix} Q_1K_1 & Q_1K_2 & Q_1K_3 \\ Q_2K_1 & Q_2K_2 & Q_2K_3 \\ Q_3K_1 & Q_3K_2 & Q_3K_3 \end{bmatrix}$
- Assume K_3 is from padding token (in input sentence), should be masked
  - ○ $\mathrm{softmax}\left(\mathrm{QK^T} + \begin{bmatrix} 0 & 0 & -\infty \\ 0 & 0 & -\infty \\ 0 & 0 & -\infty \end{bmatrix}\right) = \begin{bmatrix} e_{11} & e_{12} & 0 \\ e_{21} & e_{22} & 0 \\ e_{31} & e_{32} & 0 \end{bmatrix}$

  → In nn.Transformer().forward(), “src_key_padding_mask” do this.

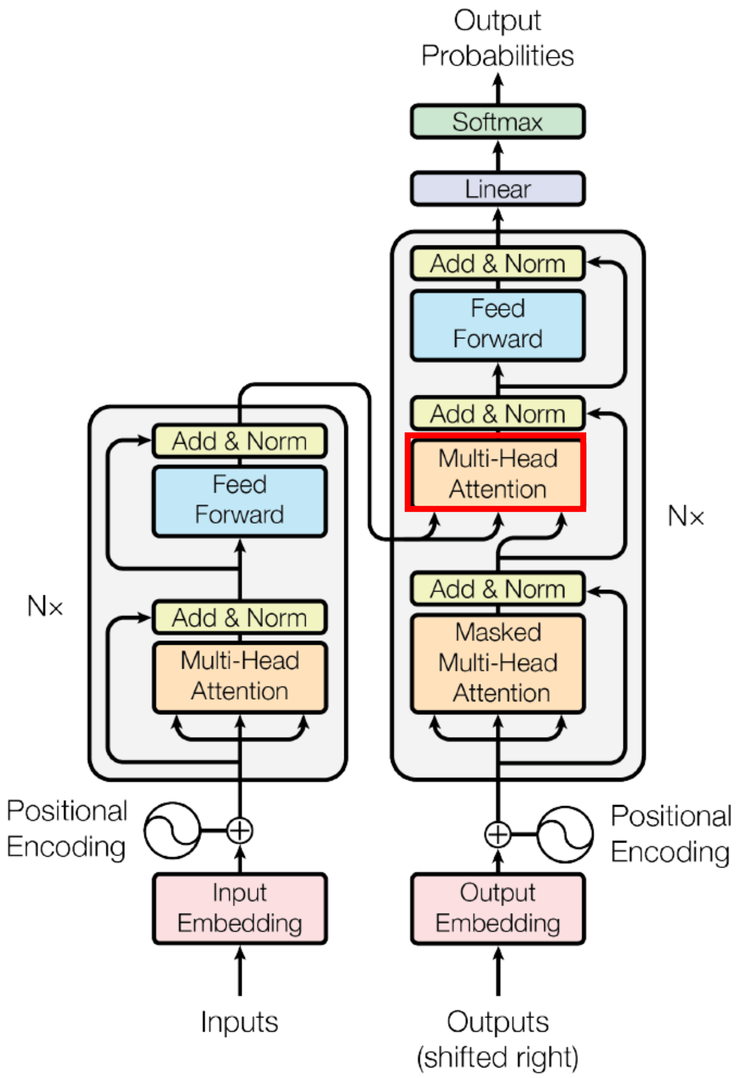


Figure 1: The Transformer - model architecture.

Transformer Implementation

nn.Transformer()

nn.Transformer()

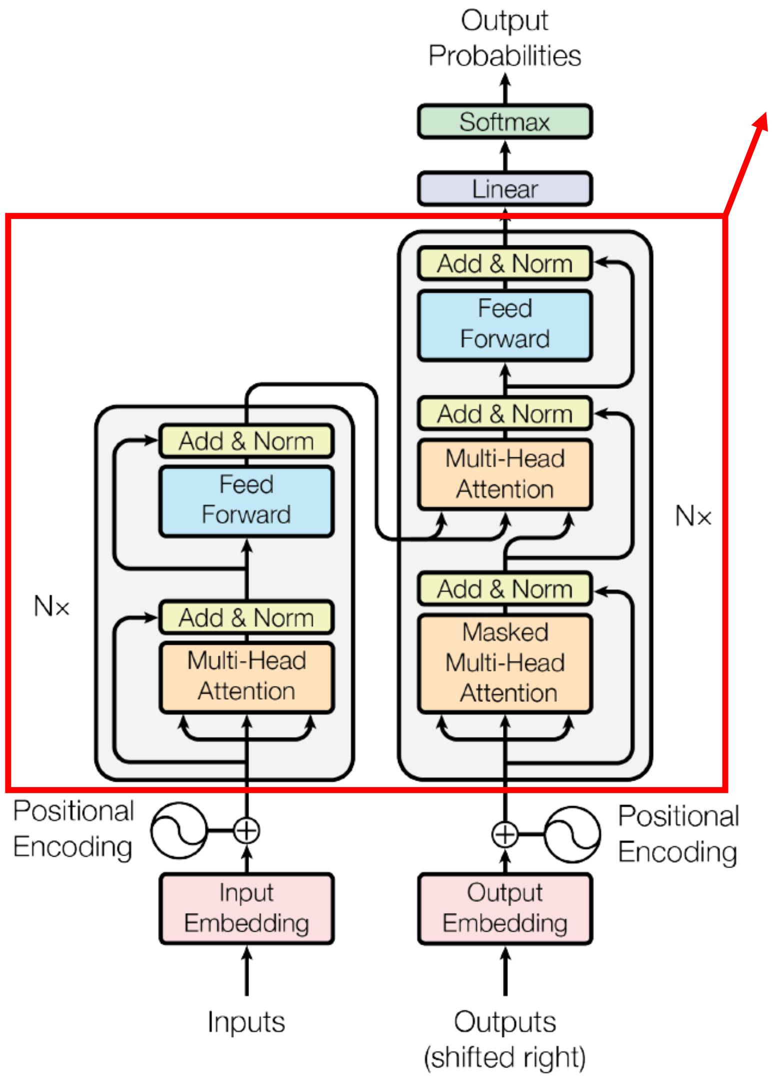


Figure 1: The Transformer - model architecture.

```
nn.Transformer()
    nn.TransformerEncoder()
        nn.ModuleList(): [
                nn.TransformerEncoderLayer()
                ...
                ]
        nn.LayerNorm() ?
    nn.TransformerDecoder()
        nn.ModuleList(): [
                nn.TransformerDecoderLayer()
                ...
                ]
        nn.LayerNorm() ?
```

nn.TransformerEncoderLayer():
- nn.MultiHeadAttention() + dropout
- nn.LayerNorm()
- 2 * nn.Linear + activation + dropout
- nn.LayerNorm()

nn.TransformerDecoderLayer():
- nn.MultiHeadAttention() + dropout
- nn.LayerNorm()
- nn.MultiHeadAttention() + dropout
- nn.LayerNorm()
- 2 * nn.Linear + activation + dropout
- nn.LayerNorm()

- Model Parameters:
  - Dimension of Model (word embedding dimension)
  - Hidden dimension of FFN
  - # of Heads
  - # of (Encoder/Decoder)layers
  - Dropout ratio

- You need more:
  - Input Embedding
  - Positional Encoding
  - Output Linear Layer

nn.Transformer()

- **Print(nn.Transformer())**

```
Transformer(
  (encoder): TransformerEncoder(
    (layers): ModuleList(
      (0-5): 6 x TransformerEncoderLayer(
        (self_attn): MultiheadAttention(
          (out_proj): NonDynamicallyQuantizableLinear(in_features=512, out_features=512, 
bias=True)
        )
        (linear1): Linear(in_features=512, out_features=2048, bias=True)
        (dropout): Dropout(p=0.1, inplace=False)
        (linear2): Linear(in_features=2048, out_features=512, bias=True)
        (norm1): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
        (norm2): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
        (dropout1): Dropout(p=0.1, inplace=False)
        (dropout2): Dropout(p=0.1, inplace=False)
      )
    )
    (norm): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
  )
…
)
```

Why additional LayerNorm?

```
Transformer(
…
  (decoder): TransformerDecoder(
    (layers): ModuleList(
      (0-5): 6 x TransformerDecoderLayer(
        (self_attn): MultiheadAttention(
          (out_proj): NonDynamicallyQuantizableLinear(in_features=512, 
out_features=512, bias=True)
        )
        (multihead_attn): MultiheadAttention(
          (out_proj): NonDynamicallyQuantizableLinear(in_features=512, 
out_features=512, bias=True)
        )
        (linear1): Linear(in_features=512, out_features=2048, bias=True)
        (dropout): Dropout(p=0.1, inplace=False)
        (linear2): Linear(in_features=2048, out_features=512, bias=True)
        (norm1): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
        (norm2): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
        (norm3): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
        (dropout1): Dropout(p=0.1, inplace=False)
        (dropout2): Dropout(p=0.1, inplace=False)
        (dropout3): Dropout(p=0.1, inplace=False)
      )
    )
    (norm): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
  )
)
```

nn.Transformer()

▪ Why Additional LayerNorm in nn.Transformer(Encoder/Decoder)?

- nn.Transformer has “norm_first” argument

CLASS torch.nn.Transformer(*d_model=512*, *nhead=8*, *num_encoder_layers=6*, *num_decoder_layers=6*, *dim_feedforward=2048*, *dropout=0.1*, *activation=<function relu>*, *custom_encoder=None*, *custom_decoder=None*, *layer_norm_eps=1e-05*, *batch_first=False*, *norm_first=False*, *bias=True*, *device=None*, *dtype=None*) [SOURCE]

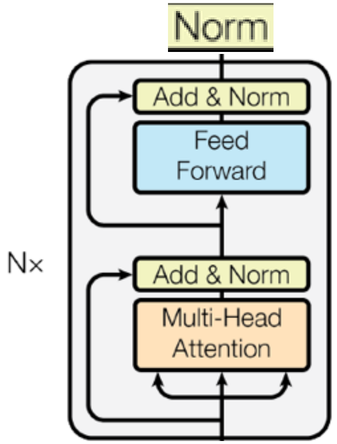
- In each nn.Transformer(Encoder/Decoder)Layer
  - ○ You can perform LayerNorm operation before its sublayers (MHA, FFN)
    - ➢ nn.TransformerEncoderLayer, same as nn.TransformerDecoderLayer

```
if self.norm_first:
    x = x + self._sa_block(self.norm1(x), src_mask, src_key_padding_mask, is_causal=is_causal)
    x = x + self._ff_block(self.norm2(x))
else:
    x = self.norm1(x + self._sa_block(x, src_mask, src_key_padding_mask, is_causal=is_causal))
    x = self.norm2(x + self._ff_block(x))

return x
```

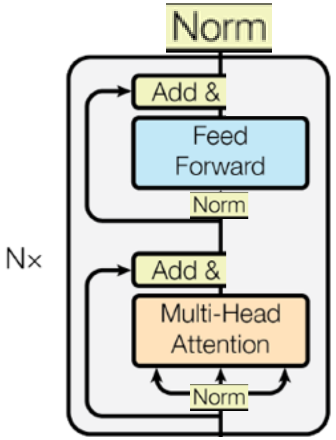
    - ➢ nn.Transformer(Encoder/Decoder)

```
for mod in self.layers:
    output = mod(output, src_mask=ma

if convert_to_nested:
    output = output.to_padded_tensor

if self.norm is not None:
    output = self.norm(output)
```

self.norm always be initialized.



self.norm_first == False



self.norm_first == True

- To ensure the output values to be normalized and maintain same number of parameters in both case

nn.Transformer()

Batch First?

- nn.Transformer has “batch_first” argument.

```
CLASS torch.nn.Transformer(d_model=512, nhead=8, num_encoder_layers=6,
    num_decoder_layers=6, dim_feedforward=2048, dropout=0.1, activation=<function
    relu>, custom_encoder=None, custom_decoder=None, layer_norm_eps=1e-05,
    batch_first=False, norm_first=False, bias=True, device=None, dtype=None) [SOURCE]
```

- Usually when we handle torch.Tensor, the shape would be like
  - [batch_size, channels, height, width]
- Sometimes when we handle sequence data (variable lengths) using torch.Tensor, the shape would be like
  - [sequence_length, batch_size, dimension_embedding].
  - batch size dimension is not in the first place of shape.
  - This type of shape can be considered when we use RNN based model.
    - [1, batch_size, dim_embedding] shaped input at one forward operation in RNN based model.
- nn.RNN() and nn.LSTM() provided this argument.
- Which one to use? → Up to you!
- I prefer “batch_first=True”, because it would be easier to think about operation of MHA.

TransformerModel() using nn.Transformer

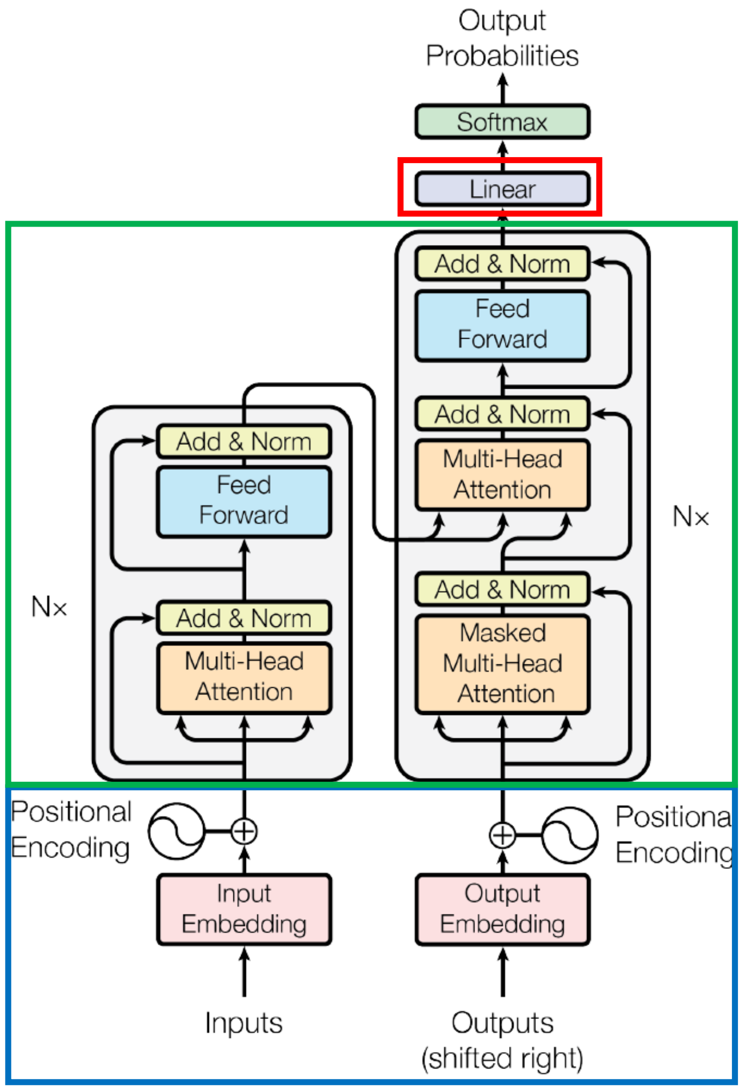


Figure 1: The Transformer - model architecture.

TransformerModel()
- TransformerEmbedding()
  - nn.Embedding()
  - nn.Embedding()
  - nn.Dropout()
- TransformerEmbedding()
  - nn.Embedding()
  - nn.Embedding()
  - nn.Dropout()
- nn.Transformer()
- nn.Linear()

```python
class TransformerModel(nn.Module):
    def __init__(self, vocab_size_src, vocab_size_tgt, d_model=256,
        n_heads=8, n_enc_layers=3, n_dec_layers=3, d_feedforward=512,
        dropout=0.1, src_max_len=100, tgt_max_len=100,
    ) -> None:
        super().__init__()

        self.embed_src = TransformerEmbedding(d_model=d_model,
            n_embeddings=vocab_size_src, max_len=src_max_len,
        )

        self.embed_tgt = TransformerEmbedding(d_model=d_model,
            n_embeddings=vocab_size_tgt, max_len=tgt_max_len,
        )

        self.transformer = nn.Transformer(
            d_model,
            n_heads,
            n_enc_layers,
            n_dec_layers,
            d_feedforward,
            dropout,
            batch_first=True,
        )

        self.fc = nn.Linear(d_model, vocab_size_tgt)
```

TransformerModel() using nn.Transformer

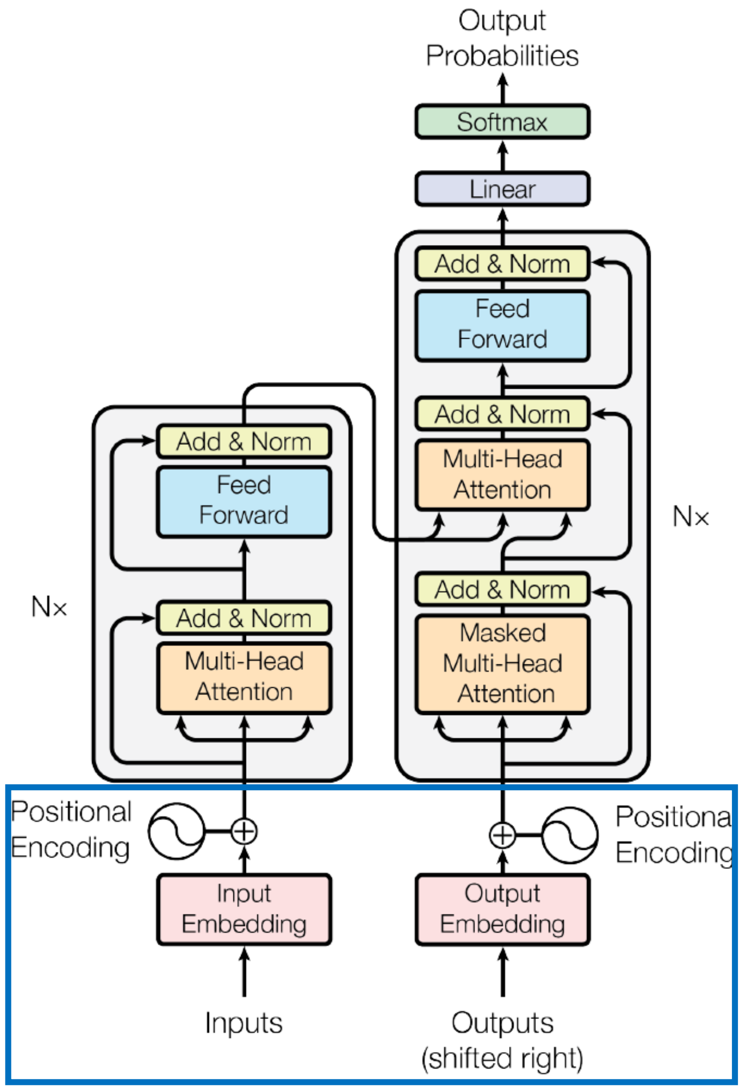


Figure 1: The Transformer - model architecture.

TransformerModel()
- **TransformerEmbedding()**
  - nn.Embedding()
  - nn.Embedding()
  - nn.Dropout()
- **TransformerEmbedding()**
  - nn.Embedding()
  - nn.Embedding()
  - nn.Dropout()
- nn.Transformer()
- nn.Linear()

```
class TransformerEmbedding(nn.Module):
    def __init__(self, d_model, n_embeddings,
        max_len=100, dropout=0.1) -> None:
        super().__init__()

        self.max_len = max_len

        self.embed_tok = nn.Embedding(n_embeddings, d_model)
        self.embed_pos = nn.Embedding(max_len, d_model)

        self.do = nn.Dropout(dropout)
        self.scale = d_model ** (1 / 2)

    def forward(self, x):
        batch_size, seq_len = x.shape
        pos = torch.arange(0, seq_len).repeat(batch_size, 1).to(x.device)
        pos = torch.where(x != 0, pos, self.max_len - 1)
        return self.do((self.embed_tok(x) * self.scale) + self.embed_pos(pos))
```

TransformerModel() using nn.Transformer

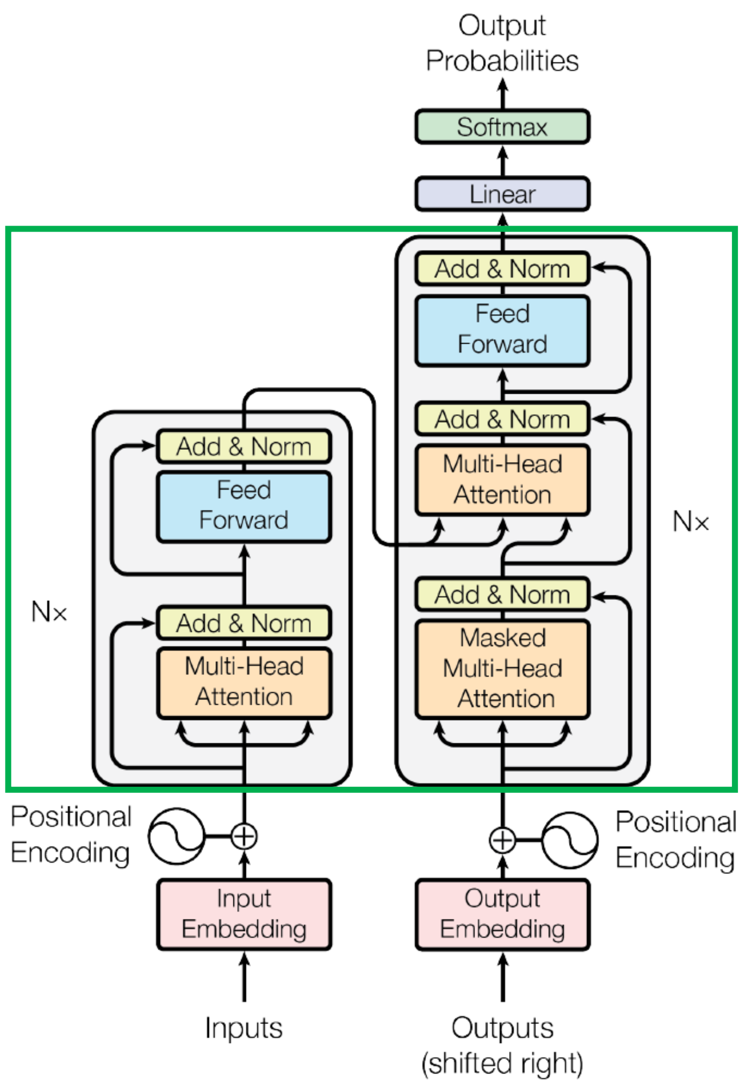


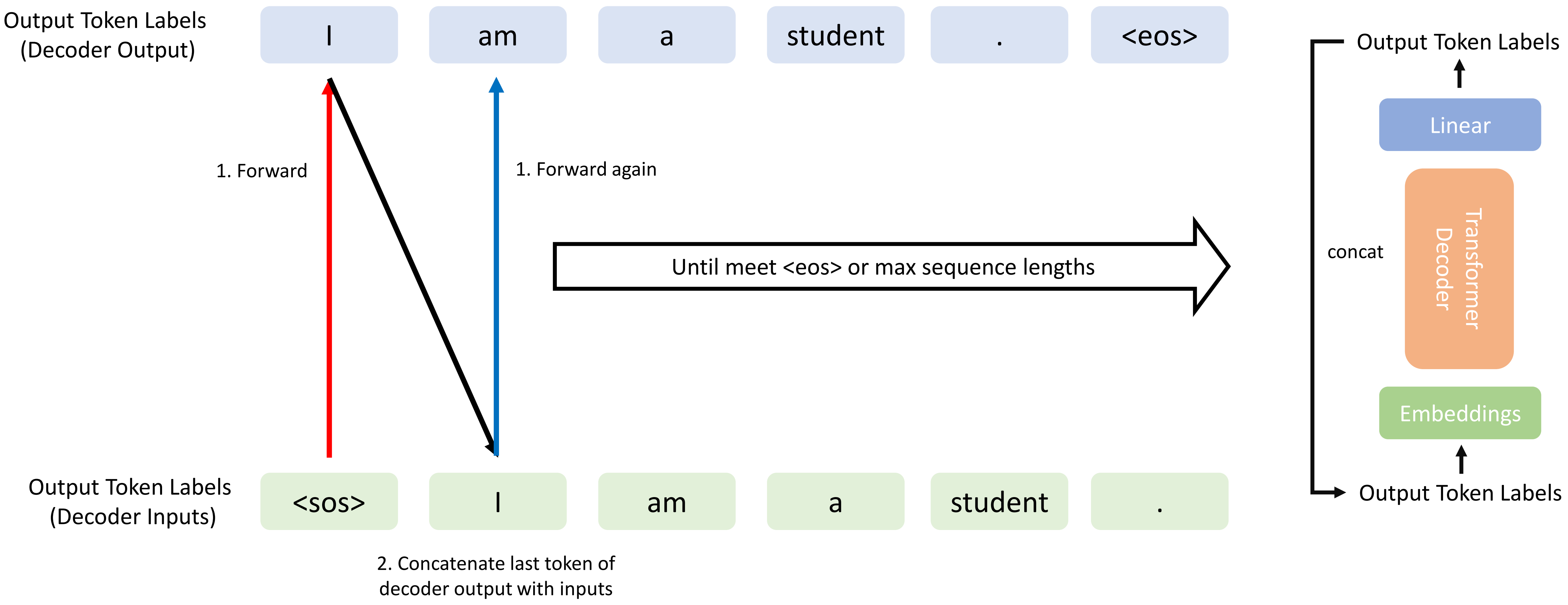
Figure 1: The Transformer - model architecture.

TransformerModel()
.forward()

```python
def make_padding_mask(self, x):
    return torch.where(x == 0, True, False)  # (N, L)

def make_causal_mask(self, sz):
    return torch.ones([sz, sz]).tril() == 0  # (L, L)

def forward(self, src, tgt):
    src_key_padding_mask = self.make_padding_mask(src).to(src.device)
    tgt_key_padding_mask = self.make_padding_mask(tgt).to(tgt.device)
    tgt_mask = self.make_causal_mask(tgt.shape[-1]).to(tgt.device)

    enc_in = self.embed_src(src)
    dec_in = self.embed_tgt(tgt)

    out = self.transformer(
        enc_in,
        dec_in,
        tgt_mask=tgt_mask,
        src_key_padding_mask=src_key_padding_mask,
        tgt_key_padding_mask=tgt_key_padding_mask,
    )

    return self.fc(out)
```

TransformerModel() using nn.Transformer()

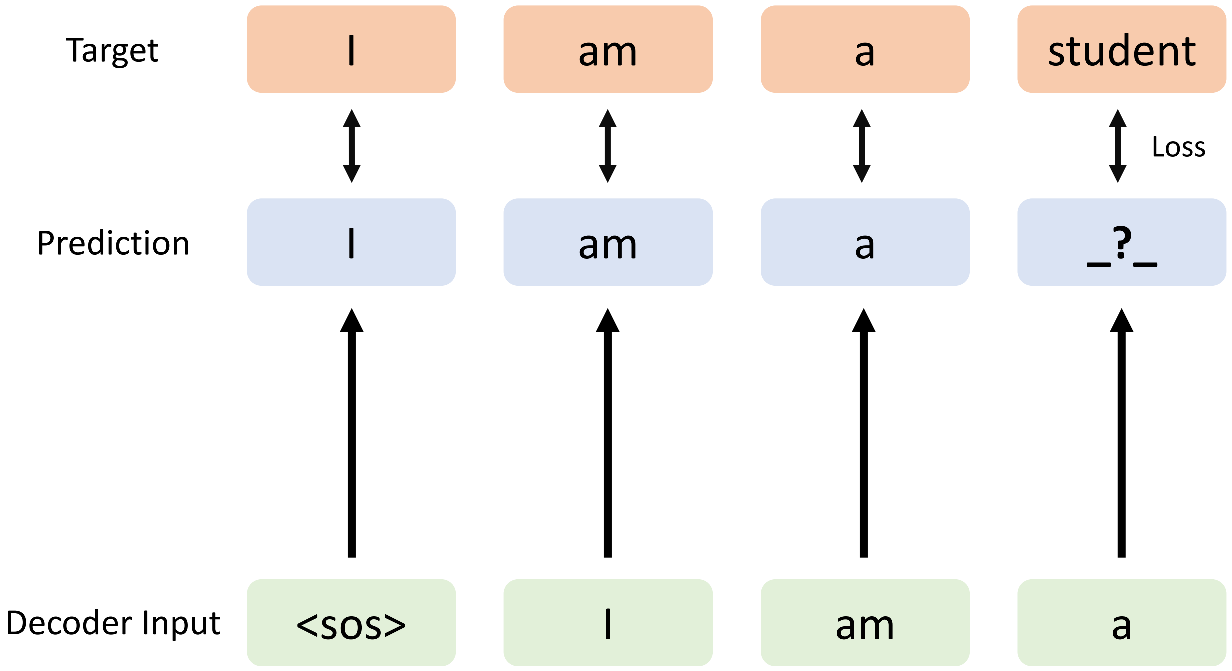
- **Autoregressive Inference**



TransformerModel() using nn.Transformer()

▪ Autoregressive Inference

- Training procedure would be like this…
- Minimize cross entropy between token labels of target sentence and predicted token labels.



What is the correct word to fill in the blank space?
→ "student" should be in blank.

<sos> "__" → <sos> "I"
<sos> I "__" → <sos> I "am"
<sos> I am "__" → <sos> I am "a"
<sos> I am a "__" → <sos> I am a "student"
<sos> I am a student "__" → <sos> I am a student "."
<sos> I am a student . "__" → <sos> I am a student . "<eos>"

TransformerModel() using nn.Transformer

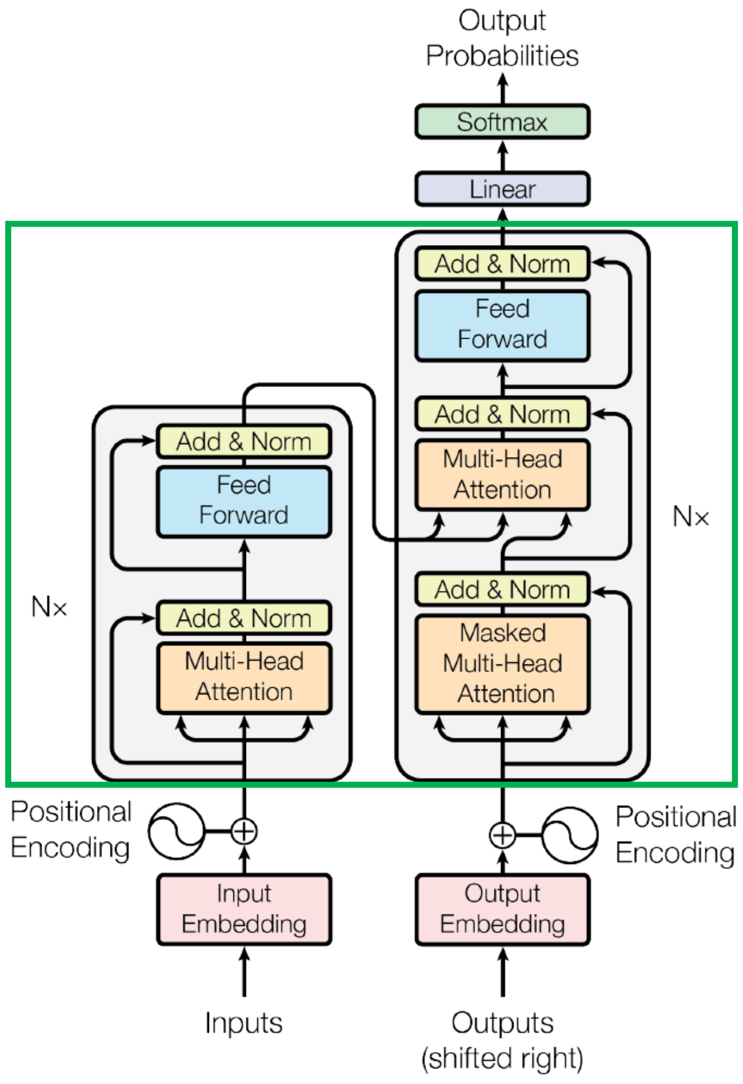


Figure 1: The Transformer - model architecture.

TransformerModel()
.translate()

→ Autoregressive Inference

```
def translate(self, src, start_id, end_id, max_len=100):
    self.eval()

    src_p_mask = self.make_padding_mask(src).to(src.device)
    enc_in = self.embed_src(src)

    # variable length batch translation
    preds = [torch.ones_like(src[:, 0].unsqueeze(1)) * start_id]
    complete = torch.zeros_like(preds[-1]) != 0
    lengths = torch.ones_like(preds[-1]) * max_len
    for idx in range(max_len):
        tgt = torch.cat(preds, dim=-1)
        tgt_p_mask = self.make_padding_mask(tgt).to(src.device)
        tgt_c_mask = self.make_causal_mask(tgt.shape[-1]).to(src.device)
        dec_in = self.embed_tgt(tgt)

        output = self.transformer(
            enc_in,
            dec_in,
            tgt_mask=tgt_c_mask,
            src_key_padding_mask=src_p_mask,
            tgt_key_padding_mask=tgt_p_mask,
        )
        output = self.fc(output)

        pred = output.argmax(-1)[:, -1].unsqueeze(1)
        preds.append(pred)

        # False -> True, then record its length
        lengths[(pred == end_id) & (complete == False)] = idx
        complete[pred == end_id] = True
        if torch.all(complete):
            break

    return torch.cat(preds[1:], dim=-1), lengths.view(-1).tolist()
```

Get attention map from nn.Transformer()

- **nn.MultiHeadAttention()**

forward(*query, key, value, key_padding_mask=None, need_weights=True, attn_mask=None, average_attn_weights=True, is_causal=False*) [SOURCE]

need_weights (*bool*) – If specified, returns `attn_output_weights` in addition to `attn_outputs`. Set `need_weights=False` to use the optimized `scaled_dot_product_attention` and achieve the best performance for MHA. Default: `True`.

average_attn_weights (*bool*) – If true, indicates that the returned `attn_weights` should be averaged across heads. Otherwise, `attn_weights` are provided separately per head. Note that this flag only has an effect when `need_weights=True`. Default: `True` (i.e. average weights across heads)

Outputs:

- **attn_output** - Attention outputs of shape (L, E) when input is unbatched, (L, N, E) when `batch_first=False` or (N, L, E) when `batch_first=True`, where L is the target sequence length, N is the batch size, and E is the embedding dimension `embed_dim`.
- **attn_output_weights** - Only returned when `need_weights=True`. If `average_attn_weights=True`, returns attention weights averaged across heads of shape (L, S) when input is unbatched or (N, L, S), where N is the batch size, L is the target sequence length, and S is the source sequence length. If `average_attn_weights=False`, returns attention weights per head of shape $(\text{num\_heads}, L, S)$ when input is unbatched or $(N, \text{num\_heads}, L, S)$.

```python
# self-attention block
def _sa_block(self, x: Tensor,
              attn_mask: Optional[Tensor], key_padding_mask: Option
    x = self.self_attn(x, x, x,
                       attn_mask=attn_mask,
                       key_padding_mask=key_padding_mask,
                       need_weights=False, is_causal=is_causal)[0]
    return self.dropout1(x)
```

```python
# self-attention block
def _sa_block(self, x: Tensor,
              attn_mask: Optional[Tensor], key_padd
    x = self.self_attn(x, x, x,
                       attn_mask=attn_mask,
                       key_padding_mask=key_padding
                       is_causal=is_causal,
                       need_weights=False)[0]
    return self.dropout1(x)

# multihead attention block
def _mha_block(self, x: Tensor, mem: Tensor,
               attn_mask: Optional[Tensor], key_pad
    x = self.multihead_attn(x, mem, mem,
                            attn_mask=attn_mask,
                            key_padding_mask=key_pa
                            is_causal=is_causal,
                            need_weights=False)[0]
    return self.dropout2(x)
```

nn.MultiHeadAttention() gives us attention weights but nn.Transformer() doesn't use it.

Do we have to modify PyTorch code to get attention weights? → No

Get attention map from nn.Transformer()

▪ nn.MultiHeadAttention()

forward(*query, key, value, key_padding_mask=None, need_weights=True, attn_mask=None, average_attn_weights=True, is_causal=False*) [SOURCE]

need_weights (*bool*) – If specified, returns `attn_output_weights` in addition to `attn_outputs`. Set `need_weights=False` to use the optimized `scaled_dot_product_attention` and achieve the best performance for MHA. Default: `True`.

average_attn_weights (*bool*) – If true, indicates that the returned `attn_weights` should be averaged across heads. Otherwise, `attn_weights` are provided separately per head. Note that this flag only has an effect when `need_weights=True`. Default: `True` (i.e. average weights across heads)

Outputs:

- **attn_output** - Attention outputs of shape (L, E) when input is unbatched, (L, N, E) when `batch_first=False` or (N, L, E) when `batch_first=True`, where L is the target sequence length, N is the batch size, and E is the embedding dimension `embed_dim`.
- **attn_output_weights** - Only returned when `need_weights=True`. If `average_attn_weights=True`, returns attention weights averaged across heads of shape (L, S) when input is unbatched or (N, L, S), where N is the batch size, L is the target sequence length, and S is the source sequence length. If `average_attn_weights=False`, returns attention weights per head of shape $(\text{num\_heads}, L, S)$ when input is unbatched or $(N, \text{num\_heads}, L, S)$.

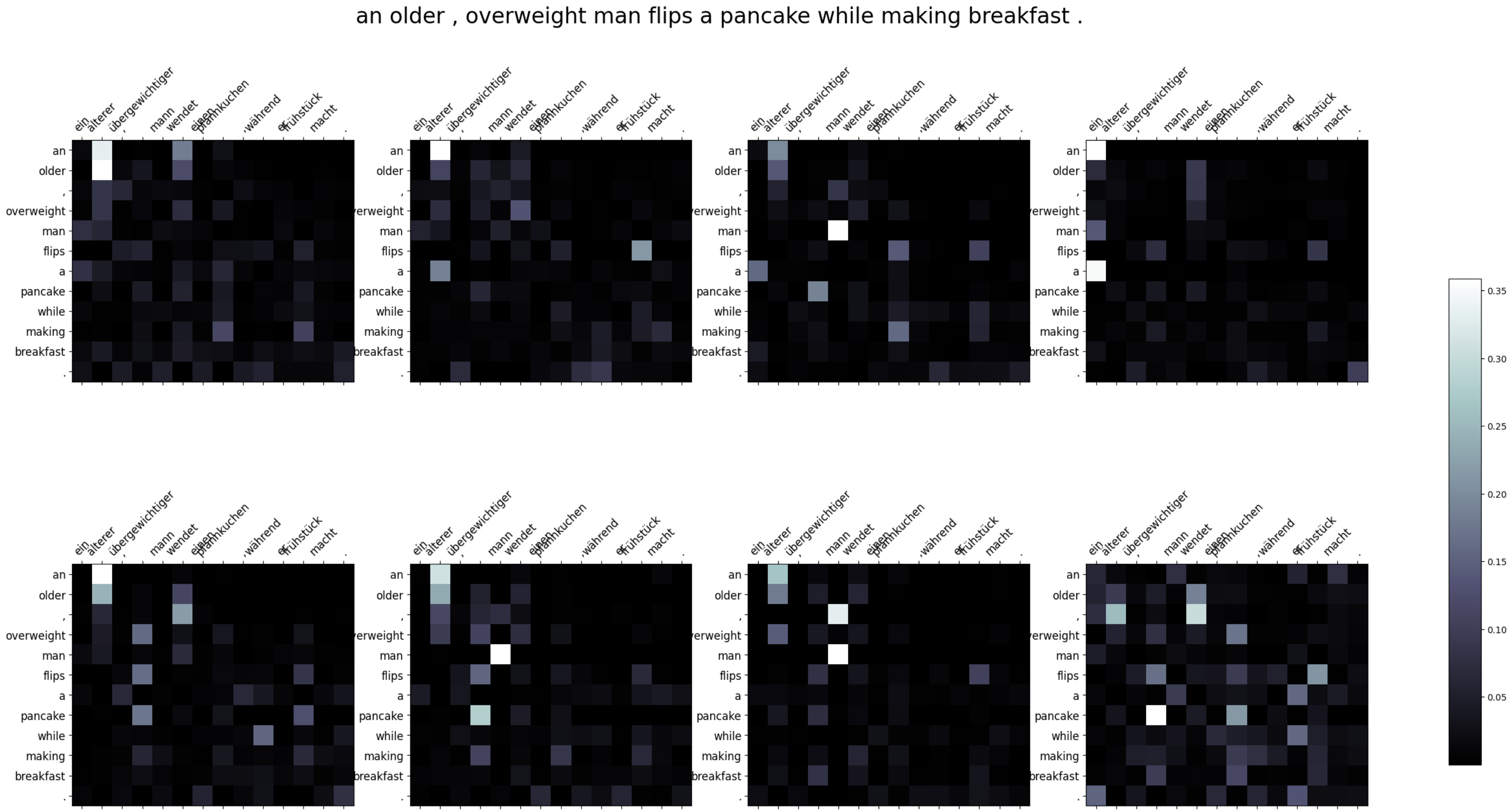
```python
class MultiheadAttentionHook:
    def __init__(self, mha_module: nn.Module) -> None:
        self.data = 0

        forward_org = mha_module.forward

        def wrap_forward(*args, **kwargs):
            kwargs["need_weights"] = True
            kwargs["average_attn_weights"] = False

            return forward_org(*args, **kwargs)

        mha_module.forward = wrap_forward

        def hook(module, x, y):
            self.data = y[1]

        mha_module.register_forward_hook(hook)
```

➢ Example

```python
# set hook
attn_hook = MultiheadAttentionHook(model.transformer.decoder.layers[-1].multihead_attn)
with torch.no_grad():
    # inference
    trans, lengths = model.translate(src, 2, 3, 50)
# get attention map
attn_maps = attn_hook.data
```

Get attention map from nn.Transformer()

- **Attention map from nn.MultiHeadAttention**
  - German-English Translation, number of heads is 8.



Transformer Implementation

from Scratch

nn.Transformer() Recap

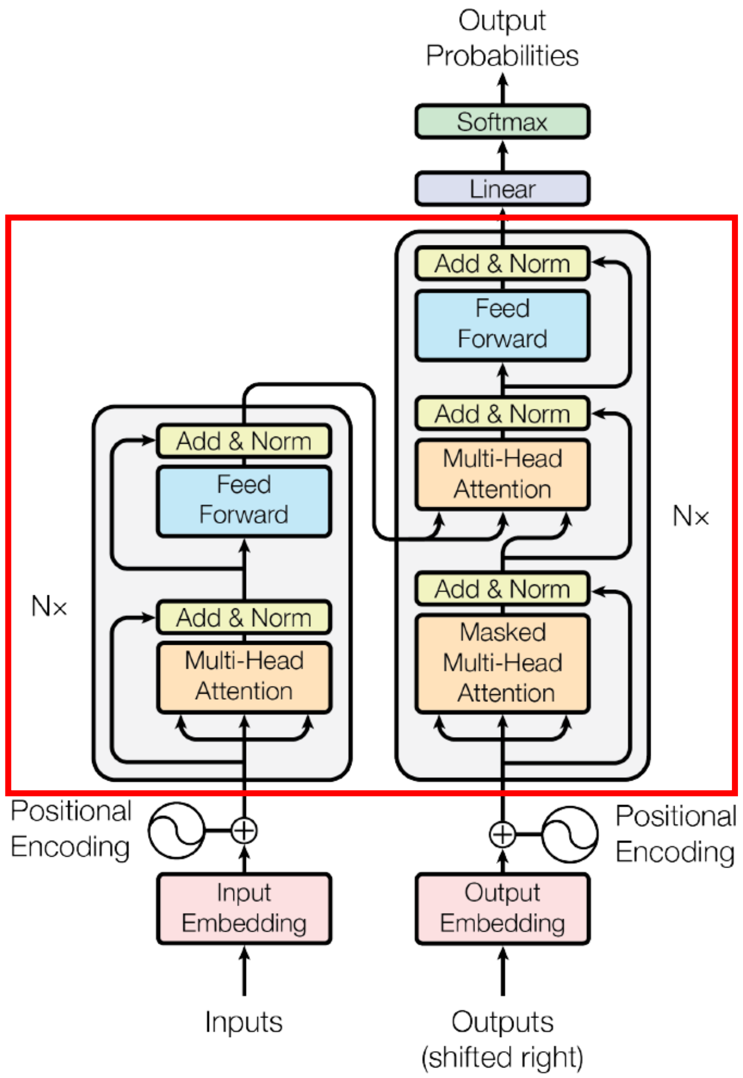


Figure 1: The Transformer - model architecture.

nn.Transformer()
    nn.TransformerEncoder()
        nn.ModuleList(): [
            nn.TransformerEncoderLayer()
            …
            ]
        ~~nn.LayerNorm()~~
    nn.TransformerDecoder()
        nn.ModuleList(): [
            nn.TransformerDecoderLayer()
            …
            ]
        ~~nn.LayerNorm()~~

nn.TransformerEncoderLayer():
- nn.MultiHeadAttention() + dropout
- nn.LayerNorm()
- 2 * nn.Linear + activation + dropout
- nn.LayerNorm()

nn.TransformerDecoderLayer():
- nn.MultiHeadAttention() + dropout
- nn.LayerNorm()
- nn.MultiHeadAttention() + dropout
- nn.LayerNorm()
- 2 * nn.Linear + activation + dropout
- nn.LayerNorm()

Transformer Implementation from Scratch

- **Going to implement…**
  - To reuse the former TransformerModel Class …
  - Reference Link (REF)

```
Transformer()
    TransformerEncoder()
        nn.ModuleList(): [
            TransformerEncoderLayer()    { MultiHeadAttention() + dropout
            …                              nn.LayerNorm()
            ]                              FFN() + dropout
                                           nn.LayerNorm()
    TransformerDecoder()
        nn.ModuleList(): [
            TransformerDecoderLayer()    { …
            …
            ]
```

Transformer Implementation from Scratch

…
- **MultiHeadAttention()** + dropout
- nn.LayerNorm()
- FFN() + dropout
- nn.LayerNorm()

```python
class MultiheadAttention(nn.Module):
    def __init__(self, d_model, n_heads, dropout=0.1) -> None:
        super().__init__()

        self.d_model = d_model
        self.n_heads = n_heads
        self.head_dim = d_model // n_heads

        self.fc_query = nn.Linear(d_model, d_model)
        self.fc_key = nn.Linear(d_model, d_model)
        self.fc_value = nn.Linear(d_model, d_model)

        self.fc_out = nn.Linear(d_model, d_model)

        self.do = nn.Dropout(p=dropout)
        self.attn_scale = 1 / self.head_dim ** (1 / 2)
```

Transformer Implementation from Scratch

- **Forward in Parallel**

Divide into h heads: $h \times d_k = d_{model}$

$$\begin{bmatrix} x_1 \\ x_2 \\ x_N \end{bmatrix} * W_Q = \begin{bmatrix} q_1 \\ q_2 \\ q_N \end{bmatrix} \Rightarrow \begin{array}{|c|c|c|} \hline & q_1 & \\ \hline & q_2 & \\ \hline & q_N & \\ \hline \end{array}$$

$[N, L, d_{model}] * [d_{model}, d_{model}] = [N, L, d_{model}]$ $\quad$ $[N, L, h, d_k]$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_N \end{bmatrix} * W_K = \begin{bmatrix} k_1 \\ k_2 \\ k_N \end{bmatrix} \Rightarrow \begin{array}{|c|c|c|} \hline & k_1 & \\ \hline & k_2 & \\ \hline & k_N & \\ \hline \end{array}$$

$[N, S, d_{model}] * [d_{model}, d_{model}] = [N, S, d_{model}]$ $\quad$ $[N, S, h, d_k]$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_N \end{bmatrix} * W_V = \begin{bmatrix} v_1 \\ v_2 \\ v_N \end{bmatrix} \Rightarrow \begin{array}{|c|c|c|} \hline & v_1 & \\ \hline & v_2 & \\ \hline & v_N & \\ \hline \end{array}$$

$[N, S, d_{model}] * [d_{model}, d_{model}] = [N, S, d_{model}]$ $\quad$ $[N, S, h, d_v]$

1. Permute Q and K
→ $[N, S, h, d_k] \rightarrow [N, h, S, d_k]$

2. Attention score $= \frac{Q \cdot K^T}{\sqrt{d_k}}$
→ $[N, h, L, S] = [N, h, L, d_k] * [N, h, S, d_k]^T$

3. Attention value $= \mathrm{softmax}\left(\frac{Q \cdot K^T}{\sqrt{d_k}} + mask\right) * V$
→ $[N, h, L, d_k] = [N, h, L, S] * [N, h, S, d_k]$

4. Permute and concatenate attention values
→ $[N, h, \mathrm{L}, d_k] \rightarrow [N, S, L, d_k] \rightarrow [N, \mathrm{L}, d_{model}]$

5. Multiply W_O for output linear operation.
→ $[N, L, d_{model}] = [N, L, d_{model}] * [d_{model}, d_{model}]$

You can calculate multiple attention heads in parallel.

Transformer Implementation from Scratch

…
- **MultiHeadAttention()** + dropout
- nn.LayerNorm()
- FFN() + dropout
- nn.LayerNorm()

```python
def forward(self, Q, K, V, mask=None):
    q = self.fc_key(Q)
    k = self.fc_key(K)
    v = self.fc_key(V)
    # Q: (N, L, d_model), q: (N, L, d_model)
    # K: (N, S, d_model), k: (N, S, d_model)
    # V: (N, S, d_model), v: (N, S, d_model)

    q = q.view(*Q.shape[:2], self.n_heads, self.head_dim).permute(0, 2, 1, 3)
    k = k.view(*K.shape[:2], self.n_heads, self.head_dim).permute(0, 2, 1, 3)
    v = v.view(*V.shape[:2], self.n_heads, self.head_dim).permute(0, 2, 1, 3)
    # q: (N, L, d_model) -> (N, n_heads, L, head_dim)
    # k: (N, S, d_model) -> (N, n_heads, S, head_dim)
    # v: (N, S, d_model) -> (N, n_heads, S, head_dim)

    attn_score = torch.matmul(q, k.permute(0, 1, 3, 2)) * self.attn_scale
    # attn_score: (N, n_heads, L, S)
    if mask is not None:
        attn_score = attn_score.masked_fill(mask, float("-inf"))
        # False -> "-inf"

    attn_weight = torch.softmax(attn_score, dim=-1)
    # attn_weight: (N, n_heads, L, S)

    attn_value = torch.matmul(self.do(attn_weight), v)
    # attn_value: (N, n_heads, L, head_dim)

    attn_value = attn_value.permute(0, 2, 1, 3).contiguous()
    attn_value = attn_value.view(*attn_value.shape[:2], -1)
    attn_value = self.fc_out(attn_value)
    # attn_value: (N, L, d_model)

    return attn_value, attn_weight
```

Transformer Implementation from Scratch

nn.Transformer()
    nn.TransformerEncoder()
        nn.ModuleList(): [
            nn.TransformerEncoderLayer()
            …
            ]
    …

```python
class TransformerEncoderLayer(nn.Module):
    def __init__(self, d_model, n_heads, d_feedforward, dropout=0.1) -> None:
        super().__init__()

        self.mha = MultiheadAttention(d_model=d_model, n_heads=n_heads,
dropout=dropout)
        self.ln1 = nn.LayerNorm(d_model)

        self.ff = FeedForward(
            d_model=d_model, d_feedforward=d_feedforward, dropout=dropout
        )
        self.ln2 = nn.LayerNorm(d_model)

        self.do = nn.Dropout(dropout)

    def forward(self, src, self_mask=None):
        attn_value = self.mha(src, src, src, self_mask)[0]
        src = self.ln1(self.do(attn_value) + src)
        src = self.ln2(self.do(self.ff(src)) + src)

        return src
```

Transformer Implementation from Scratch

nn.Transformer()
    nn.TransformerEncoder()
        nn.ModuleList(): [
            nn.TransformerEncoderLayer()
            …
            ]
    …

```python
class TransformerEncoder(nn.Module):
    def __init__(
        self,
        d_model,
        n_heads,
        n_layers,
        d_feedforward,
        dropout=0.1,
    ) -> None:
        super().__init__()

        self.layers = nn.ModuleList(
            [
                TransformerEncoderLayer(d_model, n_heads, d_feedforward, dropout)
                for _ in range(n_layers)
            ]
        )

    def forward(self, src, self_mask):
        for l in self.layers:
            src = l(src, self_mask)

        return src
```

Transformer Implementation from Scratch

nn.Transformer()
    …
    nn.TransformerDecoder()
        nn.ModuleList(): [
            nn.TransformerDecoderLayer()
            …
            ]

```
class TransformerDecoderLayer(nn.Module):
    def __init__(self, d_model, n_heads, d_feedforward, dropout=0.1) -> None:
        super().__init__()

        self.mha1 = MultiheadAttention(
            d_model=d_model, n_heads=n_heads, dropout=dropout
        )
        self.ln1 = nn.LayerNorm(d_model)

        self.mha2 = MultiheadAttention(
            d_model=d_model, n_heads=n_heads, dropout=dropout
        )
        self.ln2 = nn.LayerNorm(d_model)

        self.ff = FeedForward(
            d_model=d_model, d_feedforward=d_feedforward, dropout=dropout
        )
        self.ln3 = nn.LayerNorm(d_model)

        self.do = nn.Dropout(dropout)

    def forward(self, src, tgt, self_mask=None, cross_mask=None):
        attn_self = self.mha1(tgt, tgt, tgt, self_mask)[0]
        tgt = self.ln1(self.do(attn_self) + tgt)

        attn_cross = self.mha2(tgt, src, src, cross_mask)[0]
        tgt = self.ln2(self.do(attn_cross) + tgt)
        tgt = self.ln3(self.do(self.ff(tgt)) + tgt)

        return tgt
```

Transformer Implementation from Scratch

nn.Transformer()
…
nn.TransformerDecoder()
nn.ModuleList(): [
nn.TransformerDecoderLayer()
…
]

```python
class TransformerDecoder(nn.Module):
    def __init__(
        self,
        d_model,
        n_heads,
        n_layers,
        d_feedforward,
        dropout=0.1,
    ) -> None:
        super().__init__()

        self.layers = nn.ModuleList(
            [
                TransformerDecoderLayer(d_model, n_heads, d_feedforward, dropout)
                for _ in range(n_layers)
            ]
        )

    def forward(self, src, tgt, self_mask, cross_mask):
        for l in self.layers:
            tgt = l(src, tgt, self_mask, cross_mask)

        return tgt
```

Transformer Implementation from Scratch

- **Replace nn.Transformer() with own implemented Transformer()**

```python
class TransformerModel(nn.Module):
    def __init__(self, vocab_size_src, vocab_size_tgt, d_model=256,
        n_heads=8, n_enc_layers=3, n_dec_layers=3, d_feedforward=512,
        dropout=0.1, src_max_len=100, tgt_max_len=100,
    ) -> None:
        super().__init__()

        self.embed_src = TransformerEmbedding(d_model=d_model,
            n_embeddings=vocab_size_src, max_len=src_max_len,
        )

        self.embed_tgt = TransformerEmbedding(d_model=d_model,
            n_embeddings=vocab_size_tgt, max_len=tgt_max_len,
        )

        self.transformer = nn.Transformer(
            d_model,
            n_heads,
            n_enc_layers,
            n_dec_layers,
            d_feedforward,
            dropout,
            batch_first=True,
        )

        self.fc = nn.Linear(d_model, vocab_size_tgt)
```

```python
class TransformerModel(nn.Module):
    def __init__(self, vocab_size_src, vocab_size_tgt, d_model=256,
        n_heads=8, n_enc_layers=3, n_dec_layers=3, d_feedforward=512,
        dropout=0.1, src_max_len=100, tgt_max_len=100,
    ) -> None:
        super().__init__()

        self.embed_src = TransformerEmbedding(d_model=d_model,
            n_embeddings=vocab_size_src, max_len=src_max_len,
        )

        self.embed_tgt = TransformerEmbedding(d_model=d_model,
            n_embeddings=vocab_size_tgt, max_len=tgt_max_len,
        )

        self.transformer = Transformer(
            d_model,
            n_heads,
            n_enc_layers,
            n_dec_layers,
            d_feedforward,
            dropout,
        )

        self.fc = nn.Linear(d_model, vocab_size_tgt)
```

Machine Translation: German to English

Using PyTorch nn.Transformer and from Scratch

Dataset

Elliott, Desmond, et al. "Multi30k: Multilingual English-German image descriptions." *arXiv preprint arXiv:1605.00459* (2016).

- **Multi30K: Multilingual English-German Image Descriptions.**
  - Included in Torchtext
  - Train set: 29,000 of German-English sentence pairs
  - Validation set: 1014 sentence pairs
  - Test set: 1000 sentence pairs →Failed to load from PyTorch. → use Validation set instead
- **Test set sentence pair examples**

German	English
Eine gruppe von männern lädt baumwolle auf einen lastwagen.	A group of men are loading cotton onto a truck.
Ein mann schläft in einem grünen raum auf einem sofa.	A man sleeping in a green room on a couch.
Ein junge mit kopfhörern sitzt auf den schultern einer frau.	A boy wearing headphones sits on a woman 's shoulders .
Zwei männer bauen eine blaue eisfischerhütte auf einem zugefrorenen see auf.	Two men setting up a blue ice fishing hut on an iced over lake.
Ein mann mit beginnender glatze , der eine rote rettungsweste trägt , sitzt in einem kleinen boot.	A balding man wearing a red life jacket is sitting in a small boat.

Colab

- **nn.Transformer**
  - LINK
- **Transformer from Scratch**
  - LINK