Transformer Code Clarifications (#1)

$S_K = \{ S_K^{(1)}, S_K^{(2)}, \dots, S_K^{n-2} \}$ Source Language Sentence

$T_K = \{ T_K^{(1)}, T_K^{(2)}, \dots, T_K^{n-2} \}$ Target Language Sentence

$1 \le K \le \text{batch\_size} = 20$

$n \equiv \text{max\_seq\_length} = 10$

$S_K^{(r)} \in \{ 1, \dots, \text{source\_vocab\_size} - 2 \}$ $\quad 1 \le K \le bs$, $1 \le r \le n$

$T_K^{(r)} \in \{ 1, \dots, \text{target\_vocab\_size} - 2 \}$

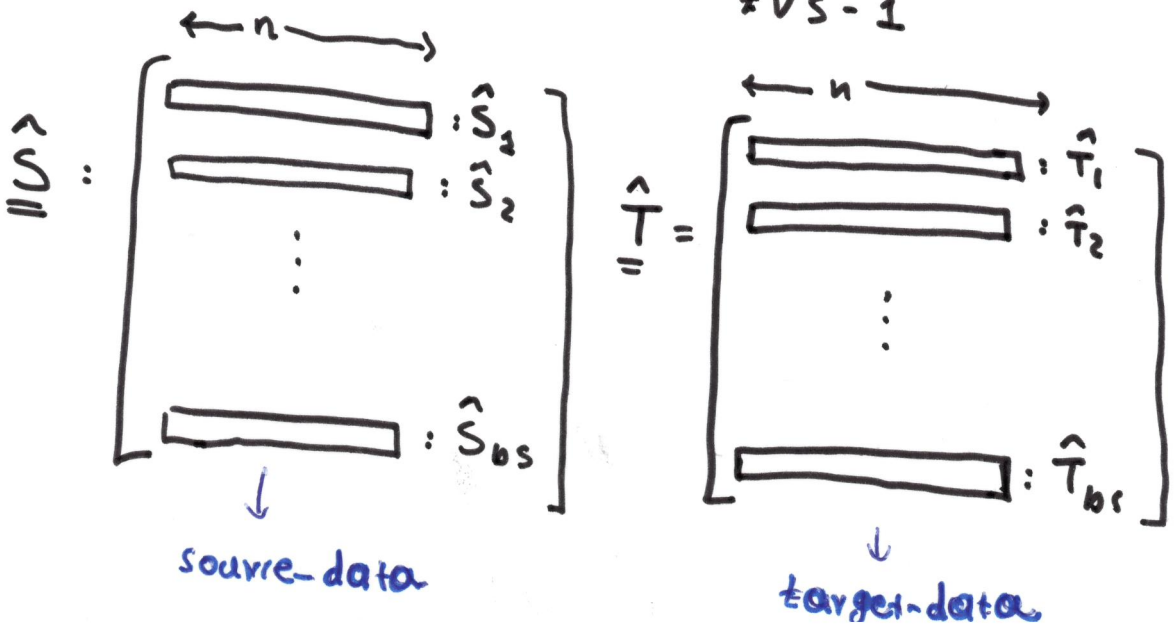
$$\begin{cases} \text{source\_vocab\_size} = 100 \\ \text{target\_vocab\_size} = 100 \end{cases}$$

$\hat{S}_K = \{ \underset{\downarrow}{<sos>}, S_K^{(1)}, S_K^{(2)}, \dots, S_K^{n-2}, \underset{\downarrow}{<eos>} \}$

0 $\qquad$ $svs-1$

$\hat{T}_K = \{ \underset{\downarrow}{<sos>}, T_K^{(1)}, T_K^{(2)}, \dots, T_K^{n-2}, \underset{\downarrow}{<eos>} \}$

0 $\qquad$ $tvs-1$



source_padding_idx ≡ source_vocab_size
target_padding_idx ≡ target_vocab.size

These indices could be used to indicate the required padding when source and target language sentences are not of the same length.

$d_m \equiv d_{model} = 200$

$H \equiv num\_heads = 4$

$\Rightarrow d_h = \frac{d_m}{H} = 50$

$d_{ff} : d_{feed\text{-}forward} = 800$

I) Source and Target Data Construction:

source_data.shape = torch.Size([20, 10])
target_data.shape = torch.Size([20, 10])

General shape of source and target data:

[batch_size × max_seq_length]
= 20 × 10

II) Source and Target Masks Construction:

Transformer.py > generate_mask(source, target)

source : [batch_size x max_seq_length]
target : [batch_size x max_seq_length]

Breakdown computation of source and target masks:

```
SM = (source != self.source_padding_idx)
SM.shape = torch.Size([20, 10])
SM = SM.unsqueeze(1)
SM.shape = torch.Size([20, 1, 10])
SM = SM.unsqueeze(2)
SM.shape = torch.Size([20, 1, 1, 10])
```

```
TM = (target != self.target_padding_idx)
TM.shape = torch.Size([20, 10])
TM = TM.unsqueeze(1)
TM.shape = torch.Size([20, 1, 10])
TM = TM.unsqueeze(2)
TM.shape = torch.Size([20, 1, 1, 10])
```

SM : [batch_size x 1 x 1 x max_seq_length]

TM : [batch_size x 1 x max_seq_length x max_seq_length]

```
SL = target.size(1)

NPM = (1- torch.triu(torch.ones(1, SL, SL,
                                device = self.device),
                     diagonal=1)).bool()
```

NPM.shape = torch.Size([1,10,10])

↳ Elements above the main diagonal are False, while elements below the main diagonal are True.

TM = TM &NPM

TM.shape = torch.Size([20,1,10,10])

(*) When processing the source sequence all words are taken into consideration when extracting the MultiHeadAttention vector representation of each word. That is, each word can look at any point of the sequence.

(*) When processing the target sequence each word can look only at the past instances of the sentence that is at the words that precede it.

III) Encoder and Decoder Output Computation:

Transformer.py > forward(source, target)

source_embedded.shape = torch.Size([20,10,200])

target_embedded.shape = torch.Size([20,10,200])

General shape of embedded source and target sequences:

[batch_size × max_seq_length × d_model]
= 20 × 10 × 200

encoder_output.shape = torch.Size([20,10,200])

Decoder output before the final fully connected layer:

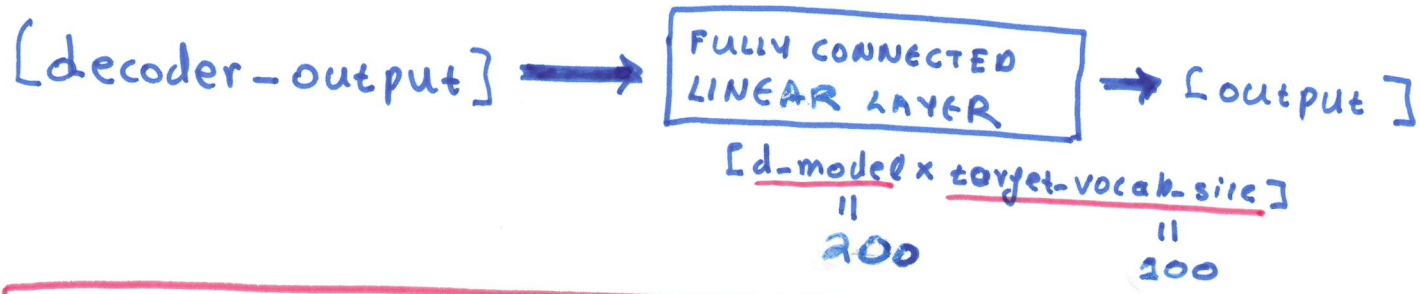
decoder_output.shape = torch.Size([20,10,200])

General shape of the encoder and decoder outputs:

[batch_size × max_seq_length × d_model]
= 20 × 10 × 200

IV) Transformer Output Computation:

Transformer.py > forward(source, target)



output.shape = torch.Size([20, 10, 100])

General shape of transformer output:

[batch_size x max_seq_length x target_vocab_size]

V) Loss Computation:

```
out = output.contiguous().view(-1, target_vocab_size)
```

out.shape = torch.Size([200, 10])

(*) Performing this operation ensures that the target_vocab_size output of the fully connected layer of the transformer for each sequence and for each instance of the sequence is row-wise stored in the out tensor.

General shape of the intermediate out tensor after the view operation:

```
out : [ batch_size * max_seq_length x target_vocab_size ]
```

```
tgt = target_data.contiguous().view(-1)
```

```
tgt.shape = torch.Size([200])
```

General shape of the intermediate tgt tensor after the view operation:

```
tgt: [ batch_size * max_seq_length ]
```

VI) Compute MultiHeadAttention Output within EncoderLayer and the first stage of the DecoderLayer { SelfAttention Mechanism }:

EncoderLayer.py > forward (x , mask) >

attn_output = self.self_attn (x , x , x , mask)

where x ≡ source_embedded ≡ encoder_output

↓

From previous encoder layers

and mask ≡ source_mask

DecoderLayer.py > forward (x , enc_output, source_mask, target_mask) >

attn_output = self.self_attn (x , x , x , target_mask)

where x ≡ target_embedded ≡ decoder_output

↓

From previous decoder layers

EncoderLayer.py > forward(x, mask) >

attn_output = self.self_attn(x, x, x, mask)

x ≡ source_embedded ≡ encoder_output

↓

From previous encoder layers

mask ≡ source_mask

x.shape = torch.Size([20, 10, 200])

X : [batch_size × max_seq_length × d_model]

mask.shape = torch.Size([20, 1, 1, 10])

mask : [batch_size × 1 × 1 × max_seq_length]

MultiHeadAttention.py > forward(Q, K, V, mask)

W_q : FULLY CONNECTED LINEAR LAYER [d_model × d_model]

W_k : FULLY CONNECTED LINEAR LAYER [d_model × d_model]

W_v : FULLY CONNECTED LINEAR LAYER [d_model × d_model]

W_o : FULLY CONNECTED LINEAR LAYER [d_model × d_model]

Q.shape = torch.Size([20, 10, 100])
K.shape = torch.Size([20, 10, 100])
V.shape = torch.Size([20, 10, 100])

Q = K = V = x (Initial State)

Q, K, V : [batch_size × max_seq_length × d_model]

Q_o = self.W_q(Q)
K_o = self.W_k(K)
V_o = self.W_v(V)

Q_o.shape = torch.Size([20, 10, 200])
K_o.shape = torch.Size([20, 10, 200])
V_o.shape = torch.Size([20, 10, 200])

Q_o, K_o, V_o : [batch_size × max_seq_length × d_model]

Q ≡ x → W_q : [200×200] → Q_o : [20 × 10 × 200]

K ≡ x → W_k : [200×200] → K_o : [20 × 10 × 200]

V ≡ x → W_v : [200×200] → V_o : [20 × 10 × 200]

MultiHeadAttention.py > forward(Q, K, V, mask)

> split_heads(x)

- $x \equiv Q_0$: [20 x 10 x 100]
- $x \equiv K_0$: [20 x 10 x 100]
- $x \equiv V_0$: [20 x 10 x 100]

[batch_size x max_seq_length x d_model]

batch_size, seq_len, d_model = x.size()
20 10 200

Q_s = Q_0.view(batch_size, seq_len, self.num_head, self.d_k
K_s = K_0.view(batch_size, seq_len, self.num_heads, self.d_k
V_s = V_0.view(batch_size, seq_len, self.num_heads, self.d_k

Q_s.shape = torch.Size([20, 10, 4, 50])
K_s.shape = torch.Size([20, 10, 4, 50])
V_s.shape = torch.Size([20, 10, 4, 50])

Q_s, K_s, V_s : [batch_size x max_seq_length x num_heads x dk]

```
Qt = Qs.transpose(1,2)
Kt = Ks.transpose(1,2)
Vt = Vs.transpose(1,2)
```

```
Qt.shape = torch.Size([20,4,10,50])
Kt.shape = torch.Size([20,4,10,50])
Vt.shape = torch.Size([20,4,10,50])
```

Qt, Kt, Vt: [batch_size x num_heads x max_seq_length x d_k]

MultiHeadAttention.py > forward(Q,K,V,mask)

Q ≡ Qt

K ≡ Kt

V ≡ Vt

> scaled_dot_product_attention(Q,K,V,mask)

Ktrans = Kt.transpose(-2,1)

```
Ktraus.shape = torch.Size([20,4,50,10])
```

General shape of Ktrans:

[batch_size x num_heads x d_k x max_seq_length]

Compute Intermediate Attention Scores :

```
AT = torch.matmul(Qt, Kraus) / self.scale
                  ↓          ↘
          [20×4×10×50]   [20×4×50×10]
```

```
AT.shape = torch.Size([20, 4, 10, 10])
```

This matrix multiplication is possible since the first two dimensions of tensors Qt and Kraus are of equal size. Moreover, sizes of the last two dimensions match [:, :, max-seq-length, d-k] × [:, :, d-k, max-seq-length] so that ordinary matrix multiplication between two 2-dimensional matrices can be performed.

```
[batch-size × num-heads × max-seq-length × d-k]

                  *
[batch-size × num-heads × dk × max-seq-length]
                  ||
[batch-size × num-heads × max-seq-length × max-seq-length]
```

Source Masking Application:

```
AT = AT.masked_fill(mask == 0, -1e9)
```

[20 × 4 × 10 × 10] [20 × 1 × 1 × 10]

AT.shape = torch.Size([20, 4, 10, 10])

General shape of AT at this stage:

[batch-size × num-heads × max-seq-length × max-seq-length]

The previous operation is possible since the source-mask tensor (mask of size: [20 × 1 × 1 × 10]) can be broadcast to a shape [20 × 4 × 10 × 10] by expanding the dimensions with size 1.

Due to the broadcasting, each slice of the source-mask along the dimensions 2 and 3 will be applied across the corresponding dimensions of the tensor attn-scores.

```
For i in range(batch-size):
    for j in range(max-seq-length):
        # The (i,j)-th slice of the source-mask: source-mask[i,:,:,j]
        # will determine whether the (i,j)-th slice of the attn-scores:
        #                          attn-scores[i,:,:,j] gets replaced
        # with -1e9.
```

Compute Attention Probabilities:

```
AP = torch.softmax(AT, -1)
```

```
AP.shape = torch.Size([20, 4, 10, 10])
```

General shape of AP at this stage:

[batch_size x num_heads x max_seq_len x max_seq_len]

(*) We can verify that the sum of all elements across the last dimension of tensor AP are equal to 1 by invoking:

```
torch.sum(AP, -1)
```

Compute Final Output for ScaledDotProductAttention (within the context of the Encoder):

[20x4x10x10] [20x4x10x50]

```
A = torch.matmul(AP, Vt)
```

(*) Tensor shapes are compatible in order for the multiplication to be executable!!!

```
A.shape = torch.Size([20, 4, 10, 50])
```

General shape of A at this stage:

[batch_size x num_heads x max_seq_length x d_k]

MultiHeadAttention.py > forward(Q, K, V, mask)

> combine_heads(x)

output = self.W_o(self.combine_heads(attn_output))

x ≡ attn_output ≡ A

A.shape = torch.Size([20, 4, 10, 50])
0 1 2 3

General shape of x at this stage: {x ≡ A}

[batch_size × num_heads × max_seq_length × d_k]

Atrans = A.transpose(1,2)

Atrans.shape = torch.Size([20, 10, 4, 50])

Atrans.is_contiguous == False

Atrans : [batch_size × max_seq_length × num_heads × d_k]

Acomb = Atrans.view(batch_size, seq_len, self.d_model)

Acomb.shape = torch.Size([20, 10, 200])

General shape of Acomb at this stage:

[batch_size × max_seq_length × d_model]

Compute the final output of the MultiHeadAttention Mechanism within the context of the Encoder:

Acomb → W-O: [200×200] → output: [20 × 10 × 200]

General shape of output:

[batch_size × max_seq_length × d_model]

Decoder.py > forward(x, enc_output, source_mask, target_mask)

attn_output = self.self_attn(x, x, x, target_mask)

x ≡ target_embedded ≡ decoder_output

x.shape = torch.Size([20, 10, 200])

x : [batch_size × max_seq_length × d_model]

target_mask.shape = torch.Size([20, 1, 10, 10])

target_mask : [batch_size × 1 × max_seq_length × max_seq_length]

Remaining steps are essentially the same as in the computation of the MultiHeadAttention Mechanism within the context of the Encoder.

The only difference is on the application of the target mask.

Target Masking Application:

AT = AT.masked_fill(mask == 0, -1e9)

mask ≡ target_mask

[20 × 4 × 10 × 10] [20 × 1 × 10 × 10]

General shape of AT at this stage:

[batch_size × num_heads × max_seq_length × max_seq_length]

General shape of mask ≡ target_mask at this stage:

[batch_size × 1 × max_seq_length × max_seq_length]

AT.shape = torch.Size([20, 4, 10, 10])

The previous operation is possible since the target_mask tensor (mask of size: [20 × 1 × 10 × 10]) can be broadcast to a shape [20 × 4 × 10 × 10] by expanding the dimension with size 1.

```
for i in range(batch_size):
    # The i-th slice of the target_mask:  target_mask[i,:,:,:]
    # will determine whether the i-th slice of the attn_scores:
    # attn_scores[i,:,:,:] gets replaced with -1e9.
```

Compute Cross Attention:

Decoder.py > forward(x, enc_output, source_mask, target_mask)

attn_output = self.cross_attn(x, enc_output, enc_output, source_mask)

x ≡ target_embedded ≡ decoder_output

x.shape = torch.Size([20, 10, 200])

enc_output.shape = torch.Size([20, 10, 200])

source_mask.shape = torch.Size([20, 1, 1, 10])

target_mask.shape = torch.Size([20, 1, 10, 10])

x : [batch_size × max_seq_length × d_model]

enc_output : [batch_size × max_seq_length × d_model]

source_mask: [batch_size × 1 × 1 × max_seq_length]

target_mask: [batch_size × 1 ×

MultiHeadAttention.py > forward(Q, K, V, mask)

Q ≡ x ≡ target_embedded ≡ decoder_output

K ≡ enc_output

V ≡ enc_output

mask ≡ source_mask

Based on the tensor assignments during calling the MultiHeadAttention mechanism in the context of the cross attention computation it is easy to deduce that:

$$\underline{\underline{S}} = \underline{\underline{Q}} \cdot \underline{\underline{K}}^T \quad \text{(per attention head)}$$

$\underline{\underline{Q}}$ ↓ decoder_output

$\underline{\underline{K}}^T$ ↓ encoder_output

$$\underline{\underline{\hat{S}}} = \text{softmax}\left(\sqrt{d_K}^{-1} \cdot \underline{\underline{S}}\right)$$

$\underline{\underline{\hat{S}}}$ ↓ decoder-encoder outputs similarity values

$$\underline{\underline{Z}} = \underline{\underline{\hat{S}}} \cdot \underline{\underline{V}}$$

$\underline{\underline{V}}$ ↓ encoder_output

$\underline{\underline{\hat{S}}}$ ↓ decoder-encoder similarities

$\underline{\underline{Z}}$ ↓ Provides an alternative representation of the source sequence taking into consideration its similarities with the target sequence.

Positional Encoding Computation:

Transformer.py > forward(source, target)

```
source_embedded = self.dropout(self.positional_encoding(
                  self.encoder_embedding(source)))
```

```
target_embedded = self.dropout(self.positional_encoding(
                  self.decoder_embedding(target)))
```

```
source.shape = torch.Size([20,10])
target.shape = torch.Size([20,10])
```

General shape of source and target tensors at this stage: [batch_size x max_seq_length].

Acquire the embedding representation of the source and target tensors:

```
S = self.encoder_embedding(source)
T = self.decoder_embedding(target)
```

```
S.shape = torch.Size([20,10,100])
T.shape = torch.Size([20,10,100])
```

S, T : [batch_size x max_seq_length x d_model]

PositionalEncoding.py > forward(x)

x ≡ S or x ≡ T

```
batch_size, max_seq_length, d_model = x.size()
device = x.device
PE = torch.zeros(max_seq_length, d_model).to(device)
PE.shape = torch.size([10, 200])
```

PE : [max_seq_length x d_model]

```
position = torch.arange(0, max_seq_length, device=device,
                        dtype=torch.float)
position.shape = torch.Size([10])
position = position.unsqueeze(1)
position.shape = torch.Size([10, 1])
```

position : [max_seq_length x 1]

```
numerator = torch.arange(0, d_model, 2, device=device,
                                    ↑ STEP  dtype=torch.float)
numerator.shape = torch.Size([100])
```

numerator : $\left[\frac{\text{d\_model}}{2}\right]$

```
denominator = (torch.log(torch.FloatTensor([10000])) / d_model).
                                                    to(device)
denominator.shape = torch.Size([1])
```

denominator : [1]

```
div_term = torch.exp(numerator * denominator)
div_term.shape = torch.Size([100])
```

div_term : $\left[\frac{d\_model}{2}\right]$

Replace the even and odd positions of the positional encoding:

```
PE[:, 0::2] = torch.sin(position * div_term)
PE[:, 1::2] = torch.cos(position * div_term)
```

This operation could be further analyzed as:

```
even_positions = torch.sin(position * div_term)   # [10x1] * [100]
even_positions.shape = torch.Size([10, 100])

odd_positions = torch.cos(position * div_term)    # [10x1] * [100]
odd_positions.shape = torch.Size([10, 100])
```

even_positions, odd_positions : $\left[max\_seq\_length \times \frac{d\_model}{2}\right]$

```
PE = PE.unsqueeze(0)
PE.shape = torch.Size([1, 10, 200])
```

PE : [1 x max_seq_length x d_model]

```
PE0 = PE[:, :x.size(1)]
PE0.shape = torch.size([1, 10, 200])
```

PE0 : [1 x max_seq_length x d_model]

```
X_pos = x + PE0
X_pos.shape = torch.Size([20, 10, 200])
```

x_pos : [batch_size x max_seq_length x d_model]