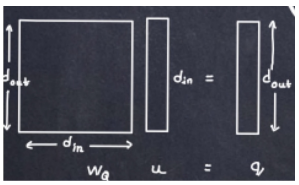
Multi-Head Attention

```python
import torch
import torch.nn as nn

class SelfAttention(nn.Module):
    def __init__(self, embed_size, heads):
        super(SelfAttention, self).__init__()
        self.embed_size = embed_size
        self.head_dim = embed_size // heads

        assert (self.head_dim * heads == embed_size),

        self.values = nn.Linear(self.head_dim, self.head_dim, bias=False)
        self.keys = nn.Linear(self.head_dim, self.head_dim, bias=False)
        self.queries = nn.Linear(self.head_dim, self.head_dim, bias=False)
        self.fc_out = nn.Linear(heads*self.head_dim, embed_size)

    def forward(self, values, keys, query, mask):
        N = query.shape[0]
        value_len, key_len, query_len = values.shape[1], keys.shape[1], query.shape[1]

        values = values.reshape(N, value_len, self.heads, self.head_dim)
        keys = keys.reshape(N, key_len, self.heads, self.head_dim)
        queries = query.reshape(N, query_len, self.heads, self.head_dim)

        energy = torch.enisum("nqhd, nkhd->nhqk", [queries, keys])

        if mask is not None:
            energy = energy.masked_fill(mask == 0, float("-1e20"))

        attention = torch.softmax(energy / (self.embed_size ** (1/2)), dim=3)

        out = torch.enisum("nhql, nlhd->nqhd", [attention, values]).reshape(N, query_len, self.heads*self.head_dim)

        out = self.fc_out()
```



A linear CNN block
$d_{in} = d_{out} = head\_dim$

attention = $\langle q_i, k_j \rangle$

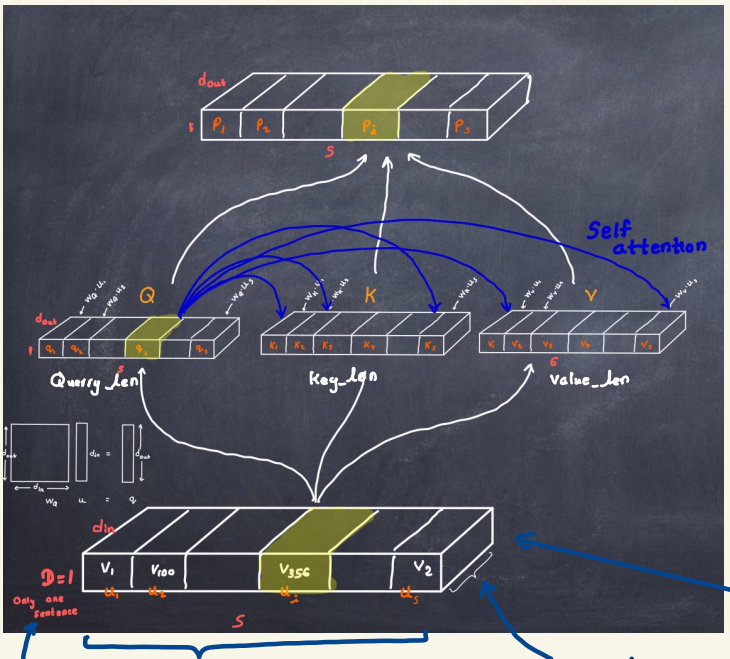
$\sqrt{d_k}$

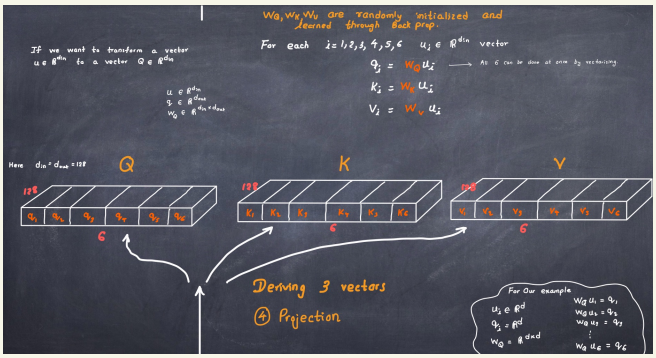
attention. values

$$P_1 = \langle q_1, K_1 \rangle V_1 + \langle q_1, K_2 \rangle V_2 + \langle q_1, K_3 \rangle V_3$$

$$P_2 = \langle q_2, K_1 \rangle V_1 + \langle q_2, K_2 \rangle V_2 + \langle q_2, K_3 \rangle V_3$$

$$P_i = \sum_{j=1}^{S} \text{Softmax}\left(\frac{\langle q_i, K_j \rangle V_j}{\sqrt{d_k}}\right) \quad \cdots \quad S = 6 \text{ in our example}$$

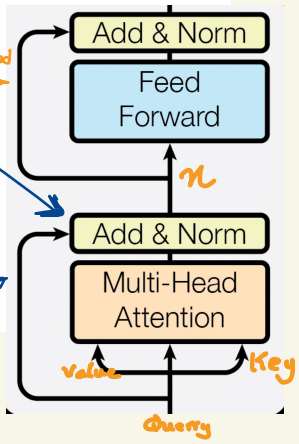




Embed Size is the total no. of values in this matrix

[0] dimension in Values, Keys and Queries

```python
class TrasformerBlock(nn.Module):
    def __init__(self, embed_size, heads, dropout, forward_expansion):
        super(TrasformerBlock, self).__init__()
        self.attention = SelfAttention(embed_size, heads)
        self.norm1 = nn.LayerNorm(embed_size)       #BatchNorm - Average accross the batch, LayerNorm - Average across all examples.
        self.norm2 = nn.LayerNorm(embed_size)

        self.feed_forward = nn.Sequential(
            nn.Linear(embed_size, forward_expansion*embed_size),
            nn.Relu(),
            nn.Linear(forward_expansion,*embed_size, embed_size)
        )

        self.droupout = nn.Dropout(dropout)


    def forward(self, value, key, query, mask):
        attention = self.attention(value, key, query, mask)
        x = self.droupout(self.norm1(attention + query))
        forward = self.feed_forward(x)
        out = self.dropout(self.norm2(forward + x))
        return out
```

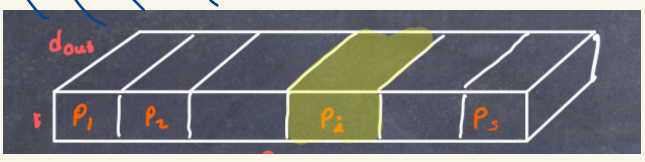


x is added again

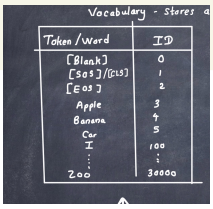
Query is added again? Why?

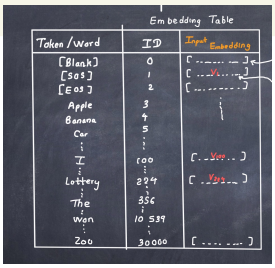
embed size - total values in embedding matrix

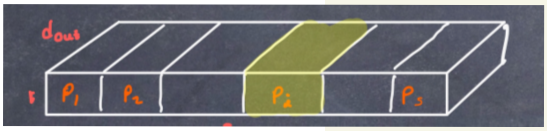
forward expansion * embed size



embed size








```python
class Encoder(nn.Module):
    def __init__(
        self,
        src_vocab_size,
        embed_size,
        num_layers,
        heads,
        device, forward_expansion,
        dropout,
        max_length,
    ):
        super(Encoder, self).__init__()
        self.embed_size = embed_size
        self.device = device
        self.word_embedding = nn.Embedding(src_vocab_size, embed_size)
        self.position_embedding = nn.Embedding(max_length, embed_size)

        self.layers == nn.ModuleList(
            [
                TrasformerBlock(
                    embed_size,
                    heads,
                    dropout=dropout,
                    forward_expansion=forward_expansion,
                )
            ]
        )

        self.droupout == nn.Dropout(dropout)


    def forward(self, x, mask):
        N, seq_length = x.shape
        positions = torch.arange(0, seq_length).expand(N, seq_length).to(self.device)

        out = self.dropout(self.word_embedding(x) + self.position_embedding(positions))

        for layer in self.layers:
            out = layer(out, out, out, mask)

        return out
```

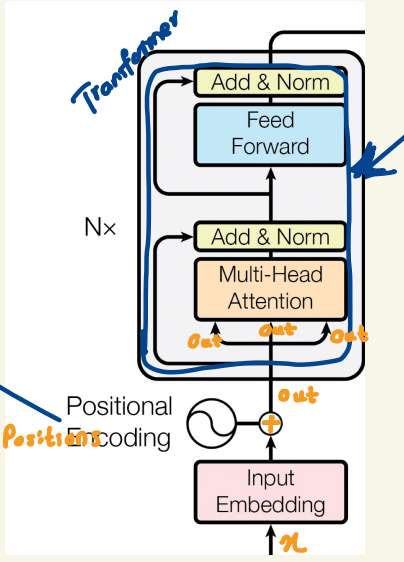
- positional encoding needs an idea about the length of longest sequence

only one transforme layer is in self.layers

Transformer
- out
- out
- out
- mask

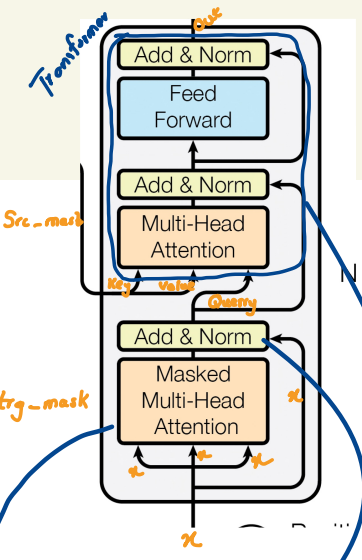
Getting it to cuda if available

- All key, query, value in transforme is having same out, dimension

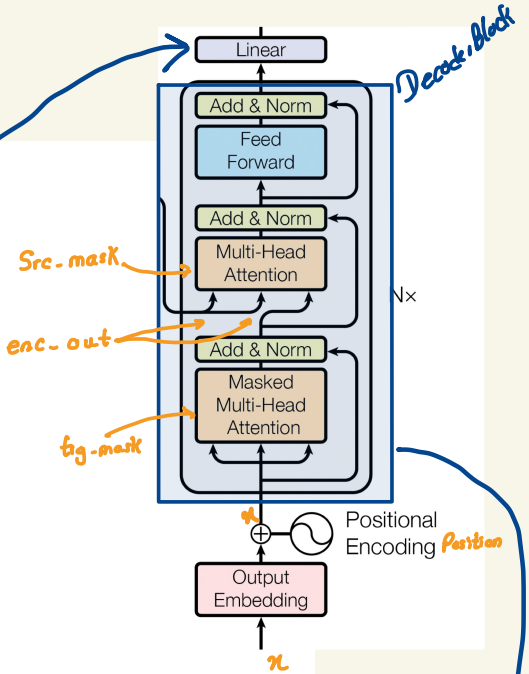


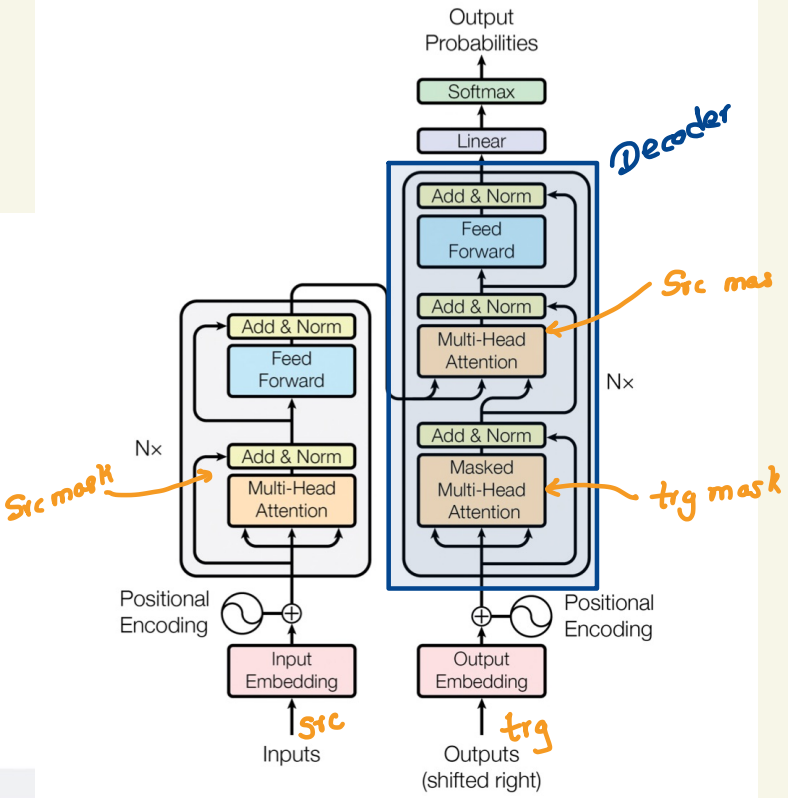
- Encoder

```python
class DecoderBlock(nn.Module):
    def __init__(
        self,
        embed_size,
        heads,
        device,
        forward_expansion,
        dropout
    ):

        super(DecoderBlock, self).__init__
        self.attention = SelfAttention(embed_size, heads)
        self.norm = nn.LayerNorm(embed_size)
        self.trasformer_block = TrasformerBlock(embed_size, heads, dropout, forward_expansion)
        self.dropout = nn.Dropout(dropout)


    def forward(self, x, value, key, src_mask, trg_mask):
        attention = self.attention(x, x, x, trg_mask)
        query = self.dropout(self.norm(attention + x))
        out = self.trasformer_block(value, key, query, src_mask)
        return out
```




```python
class Decoder(nn.Module):
    def __init__(self,
                 trg_vocab_size,
                 embed_size,
                 num_layers,
                 heads,
                 forward_expansion,
                 dropout,
                 device,
                 max_length,
    ):
        super(Decoder, self).__init__
        self.device = device
        self.word_embedding = nn.Embedding(trg_vocab_size, embed_size)
        self.position_embedding = nn.Embedding(max_length, embed_size)

        self.layers = nn.ModuleList(
            [DecoderBlock(embed_size, heads, forward_expansion, dropout, device) for _ in range(num_layers)]
        )

        self.fc_out = nn.Linear(embed_size, trg_vocab_size)
        self.dropout = nn.Dropout(dropout)


    def forward(self, x, enc_out, src_mask, trg_mask):
        N, seq_length = x.shape
        positions = torch.arange(0, seq_length).expand(N, seq_length).to(self.device)
        x = self.dropout((self.word_embedding(x) + self.position_embedding(positions)))

        for layer in self.layers:
            x = layer(x, enc_out, enc_out, trg_mask)

        out = self.fc_out(x)
```






```python
class Trasformer(nn.Module):
    def __init__(self,
                 src_vocab_size,
                 trg_vocab_size,
                 src_pad_idx,
                 trg_pad_idx,
                 embed_size=256,
                 num_layers=6,
                 forward_expansion=4,
                 heads=8,
                 dropout=0,
                 device='cuda',
                 max_length=100,
    ):
        super(Trasformer, self).__init__()

        self.encoder = Encoder(…

        self.decoder = Decoder(…

        self.src_pad_idx = src_pad_idx
        self.trg_pad_idx = trg_pad_idx
        self.device = device

    def make_src_mask(self, src):
        src_mask = (src != self.src_pad_idx).unsqueeze(1).unsqueeze(2)
        return src_mask.to(self.device)

    def make_trg_mask(self, trg):
        N, trg_len = trg.shape
        trg_mask = torch.tril(torch.ones((trg_len, trg_len))).expand(N, 1, trg_len, trg_len)
        return trg_mask.to(self.device)

    def forward(self, src, trg):
        src_mask = self.make_src_masc(src)
        trg_mask = self.make_trg_mask(trg)
        enc_src = self.encoder(src, src_mask)
        out = self.decoder(trg, enc_src, src_mask, trg_mask)
        return out
```