```
from IPython.display import Image
import numpy as np
import torch
import torch.nn as nn
import math
import copy
import torch.nn.functional as F
from torch.nn.utils.rnn import pack_padded_sequence,
print("Torch version:", torch.__version__)
```

```
Torch version: 1.9.0+cu102
```

Attention Is All You Need (Transformer)

Core Idea of the Paper

Problem

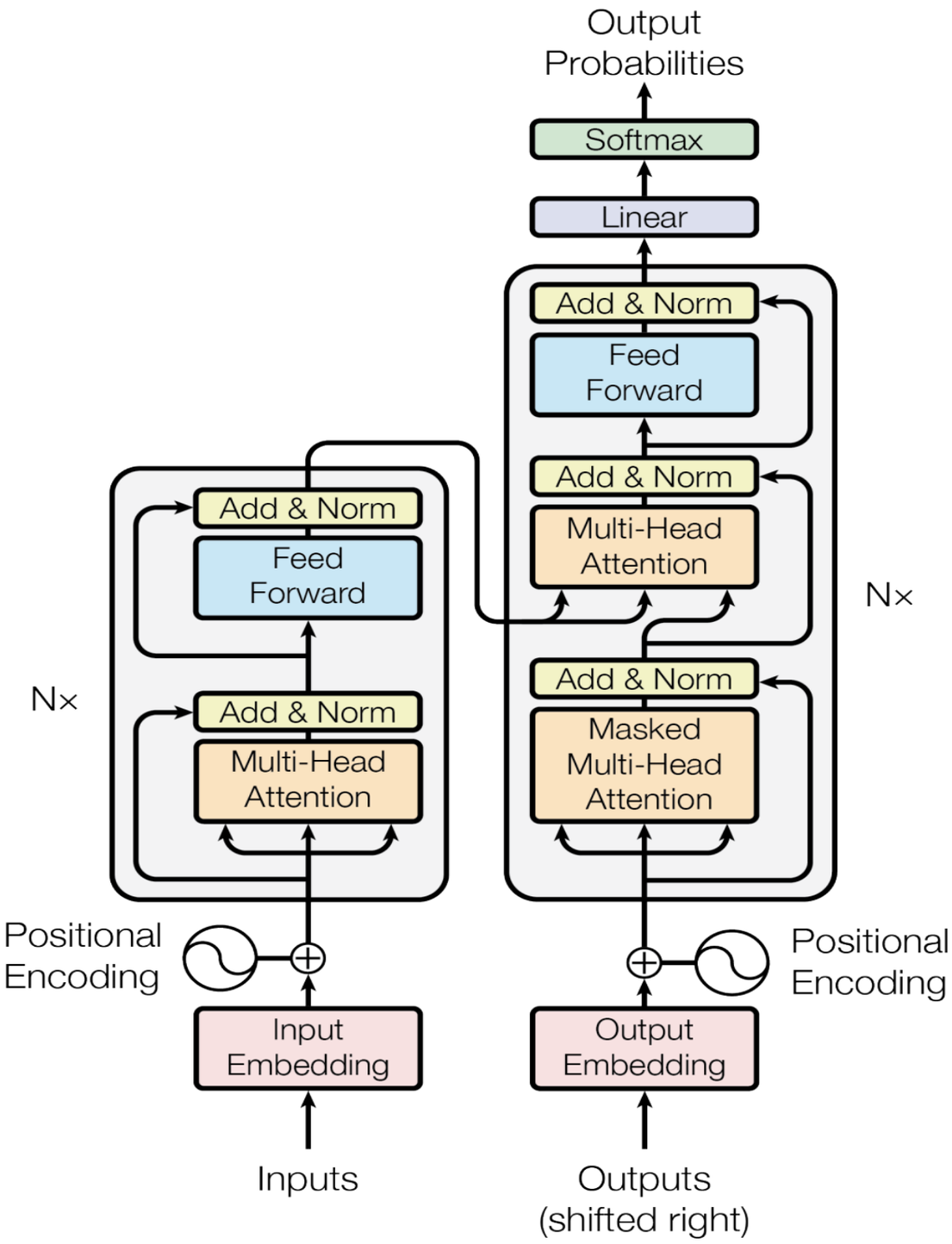
- In sequence-to-sequence problems such as the neural machine translation, the first proposals were based on the use of RNNs in an encoder-decoder architecture.
- The best performing models also connect the encoder and decoder through an attention mechanism.
- These architectures have a great limitation when working with long sequences
- In the encoder, the hidden state in every step is associated with a certain word in the input sentence, usually one of the most recent. Therefore, if the decoder only accesses the last hidden state of the decoder, it will lose relevant information about the first elements of the sequence.

Solution

- This paper propose a new simple network architecture, the Transformer, based solely on attention mechanisms to draw global dependencies between input and output.
- Instead of paying attention to the last state of the encoder as is usually done with RNNs, in each step of the decoder we look at all the states of the encoder, being able to access information about all the elements of the input sequence.
- The total computational complexity per layer
- The amount of computation that can be parallelized, as measured by the minimum number of sequential operations required.
- The path length between long-range dependencies in the network. Learning long-range dependencies is a key challenge in many sequence transduction tasks.
- List item

Model Architecture

The Transformer - model architecture.



Input Embedding and Softmax

With this layer, we convert the input tokens and output tokens to vectors of dimension d_{model} using a learned embedding. We also use the usual learned linear transformation and softmax function to convert the decoder output to predicted next-token probabilities. In the embedding layers, we multiply those weights by $\sqrt{d_{\text{model}}}$.

```
1 class Embeddings(nn.Module):
2   def __init__(self, d_model, vocab):
```

```
    super(Embeddings, self).__init__()
    self.lut = nn.Embedding(vocab, d_model)
    self.d_model = d_model

  def forward(self, x):
    return self.lut(x) * math.sqrt(self.d_model)
```

Positional Encoding

In order for the model to make use of the order of the sequence, we must inject some information about the relative or absolute position of the tokens in the senquence. In this work, we use sine and cosine functions of different frequencies:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$

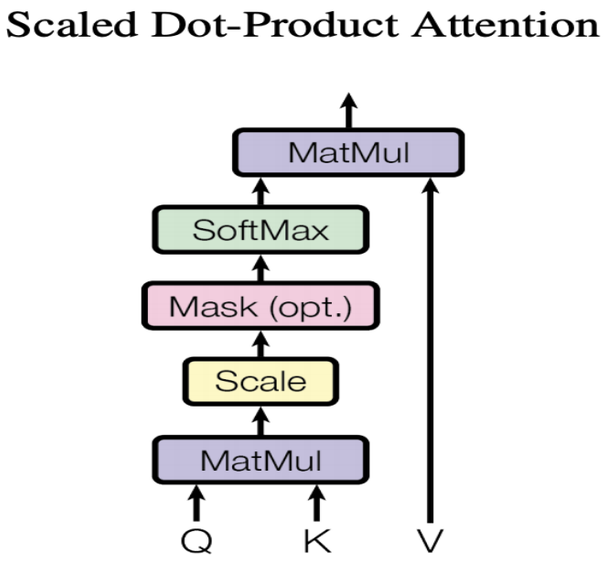
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

where pos is the position and i is the dimension. That is, each dimension of the positional encoding corresponds to a sinusoid. The wavelengths form a geometric progression from 2π to $10000 \cdot 2\pi$. We chose this function because we hypothesized **it would allow the model to easily learn to attend by relative positions, since for any fixed offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}.**

```
class PositionalEncoding(nn.Module):
    "Implement the PE function."
    def __init__(self, d_model, dropout, max_len=500
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)

        PE = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze
        div_term = torch.exp(torch.arange(0, d_model,
        PE[:, 0::2] = torch.sin(position * div_term)
        PE[:, 1::2] = torch.cos(position * div_term)
        PE = PE.unsqueeze(0)
        self.register_buffer('PE', PE)
```

```
14
15      def forward(self, x):
16          x = x + self.PE[:, :x.size(1)]
17          return self.dropout(x)
```

Scaled Dot-Product Attention



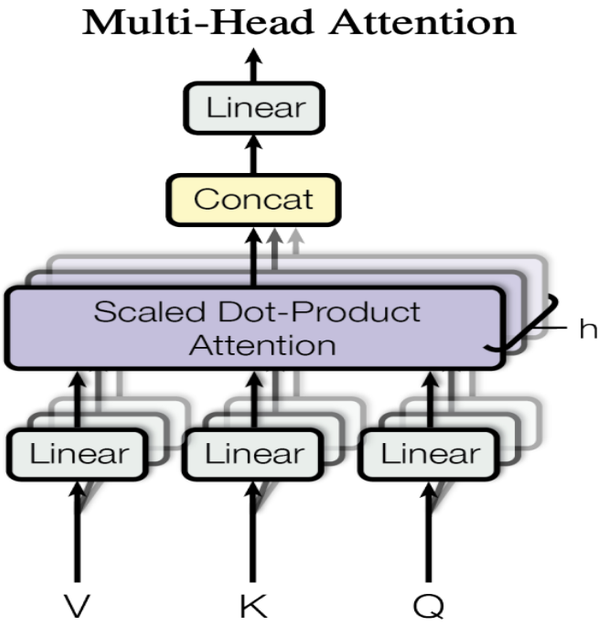
An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix Q. The keys and values are also packed together into matrices K and V. We compute the matrix of outputs as:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

```
class ScaledDotProductAttention(nn.Module):
  ''' Scaled Dot-Product Attention '''

  def __init__(self, d_k, dropout=0.1):
    super().__init__()
    self.d_k = d_k
    self.dropout = nn.Dropout(dropout)

  def forward(self, q, k, v, mask=None):

    attn = torch.matmul(q / math.sqrt(self.d_k), k.t

    if mask is not None:
        attn = attn.masked_fill(mask == 0, -1e9)

    attn = self.dropout(F.softmax(attn, dim=-1))
    output = torch.matmul(attn, v)

    return output, attn
```

Multi-Head Attention



Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_\text{h})W^O$$

$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_\text{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_\text{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_\text{model} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_\text{model}}$.

```
class MultiHeadAttention(nn.Module):
  ''' Multi-Head Attention module '''

  def __init__(self, n_head, d_model, d_k, d_v, dropo
    super().__init__()

    self.n_head = n_head
    self.d_k = d_k
    self.d_v = d_v

    self.w_q = nn.Linear(d_model, n_head * d_k)
    self.w_k = nn.Linear(d_model, n_head * d_k)
    self.w_v = nn.Linear(d_model, n_head * d_v)
    self.FC = nn.Linear(n_head * d_v, d_model)

    self.attention = ScaledDotProductAttention(d_k)

    self.dropout = nn.Dropout(dropout)
    self.layerNorm = nn.LayerNorm(d_model, eps=1e-6)


  def forward(self, Q, K, V, mask=None):

    residual = Q

    # Pass through the pre-attention projection: b x
    # Separate different heads: b x lq x n x dv
    Q = self.w_q(Q).view(Q.size(0), Q.size(1), self.n
    K = self.w_k(K).view(K.size(0), K.size(1), self.n
```

```
30      V = self.w_v(V).view(V.size(0), V.size(1), self.n
31
32      # Transpose for attention dot product: b x n x lq
33      Q, K, V = Q.transpose(1, 2), K.transpose(1, 2), V
34
35      if mask is not None:
36          mask = mask.unsqueeze(1)   # For head axis br
37
38      Q, attn = self.attention(Q, K, V, mask=mask)
39
40      # Transpose to move the head dimension back: b x
41      # Combine the last two dimensions to concatenate
42      Q = Q.transpose(1, 2).contiguous().view(Q.size(0
43      Q = self.FC(Q)
44      Q = self.dropout(Q)
45
46      return Q, attn
```

Position-wise Feed-Forward Networks

In addition to attention sub-layers, each of the layers in our encoder and decoder contains a fully connected feed-forward network, which is applied to each position separately and identically. This consists of two linear transformations with a ReLU activation in between.

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

```
1 class PositionwiseFeedForward(nn.Module):
2   ''' A two-feed-forward-layer module '''
3
4   def __init__(self, d_model, d_hid, dropout=0.1):
5     super().__init__()
6     self.W_1 = nn.Linear(d_model, d_hid)
7     self.W_2 = nn.Linear(d_hid, d_model)
8     self.dropout = nn.Dropout(dropout)
9
10  def forward(self, x):
11    x = self.W_1(x)
12    x = F.relu(x)
```

```
    x = self.W_2(x)
    x = self.dropout(x)

    return x
```

Built our Encoder

Since the core encoder contains N=6 encoder sublayer, we will define an EncoderLayer class for each sublayer.

```
class EncoderLayer(nn.Module):
  ''' Compose with two layers '''

  def __init__(self, d_model, d_hid, n_head, d_k, d_
    super(EncoderLayer, self).__init__()
    self.MHA = MultiHeadAttention(n_head, d_model, d_
    self.layerNorm = nn.LayerNorm(d_model, eps=1e-6)
    self.FFN = PositionwiseFeedForward(d_model, d_hi

  def forward(self, encoder_input, mask=None):
    res1 = encoder_input
    output1, MHA = self.MHA(encoder_input, encoder_in
    output1 = res1 + self.layerNorm(output1)
    res2 = output1
    output2 = self.FFN(output1)
    output2 = res2 + self.layerNorm(output2)
    encoder_output = output2
    return encoder_output, MHA
```

```
class Encoder(nn.Module):
  "Core encoder is a stack of N layers"
  def __init__(self, layer, N):
    super(Encoder, self).__init__()
    self.layers = nn.ModuleList([copy.deepcopy(layer
    self.layerNorm = nn.LayerNorm(d_model, eps=1e-6)

  def forward(self, x, mask):
    "Pass the input (and mask) through each layer in
    for layer in self.layers:
      x = layer(x, mask)
```

```
        x = layer(x, mask)
    return self.layerNorm(x)
```

Built our Decoder

Since the core decoder contains N=6 decoder sublayer, we will define an DecoderLayer class for each sublayer.

```
class DecoderLayer(nn.Module):
  ''' Compose with three layers '''

  def __init__(self, d_model, d_hid, n_head, d_k, d_
    super(DecoderLayer, self).__init__()
    self.MMHA = MultiHeadAttention(n_head, d_model, 
    self.MHA = MultiHeadAttention(n_head, d_model, d_
    self.layerNorm = nn.LayerNorm(d_model, eps=1e-6)
    self.FFN = PositionwiseFeedForward(d_model, d_hi

  def forward(self, decoder_input, encoder_output, se
    res1 = decoder_input
    output1, MMHA = self.MMHA(decoder_input, decoder_
    output1 = res1 + self.layerNorm(output1)
    res2 = output1
    output2, MHA = self.MHA(output1, encoder_output, 
    output2 = res2 + self.layerNorm(output2)
    res3 = output2
    output3 = self.FFN(output2)
    decoder_output = res3 + output3

    return decoder_output, MMHA, MHA
```

```
class Decoder(nn.Module):
    "Generic N layer decoder with masking."
    def __init__(self, layer, N):
        super(Decoder, self).__init__()
        self.layers = nn.ModuleList([copy.deepcopy(la

    def forward(self, x, memory, src_mask, tgt_mask)
        for layer in self.layers:
            x = layer(x, memory, src_mask, tgt_mask)
```

```
        return x
```

Transformer

```
def get_pad_mask(seq, pad_idx):
  return (seq != pad_idx).unsqueeze(-2)


def get_subsequent_mask(seq):
    ''' For masking out the subsequent info. '''
  sz_b, len_s = seq.size()
  subsequent_mask = (1 - torch.triu(
      torch.ones((1, len_s, len_s), device=seq.device
  return subsequent_mask
```

```
class Transformer(nn.Module):
    ''' A sequence to sequence model with attention m

    def __init__(
            self, n_src_vocab, n_trg_vocab, src_pad_i
            d_word_vec=512, d_model=512, d_inner=2048
            n_layers=6, n_head=8, d_k=64, d_v=64, dro

        super().__init__()

        self.src_pad_idx, self.trg_pad_idx = src_pad_
        self.d_model = d_model

        self.encoder = Encoder(
            n_src_vocab=n_src_vocab, n_position=n_pos
            d_word_vec=d_word_vec, d_model=d_model, d
            n_layers=n_layers, n_head=n_head, d_k=d_k
            pad_idx=src_pad_idx, dropout=dropout, sca

        self.decoder = Decoder(
            n_trg_vocab=n_trg_vocab, n_position=n_pos
            d_word_vec=d_word_vec, d_model=d_model, d
            n_layers=n_layers, n_head=n_head, d_k=d_k
            pad_idx=trg_pad_idx, dropout=dropout, sca

```

```
26          self.trg_word_prj = nn.Linear(d_model, n_trg_
27
28      def forward(self, src_seq, trg_seq):
29
30          src_mask = get_pad_mask(src_seq, self.src_pad
31          trg_mask = get_pad_mask(trg_seq, self.trg_pad
32
33          enc_output, *_ = self.encoder(src_seq, src_ma
34          dec_output, *_ = self.decoder(trg_seq, trg_ma
35          seq_logit = self.trg_word_prj(dec_output)
36          if self.scale_prj:
37              seq_logit *= self.d_model ** -0.5
38
39          return seq_logit.view(-1, seq_logit.size(2))
```

Train model

> Todo

Evaluation

> Todo