AMMI Bootcamp Paper Implementation

Paper Title: “Attention is all your need”

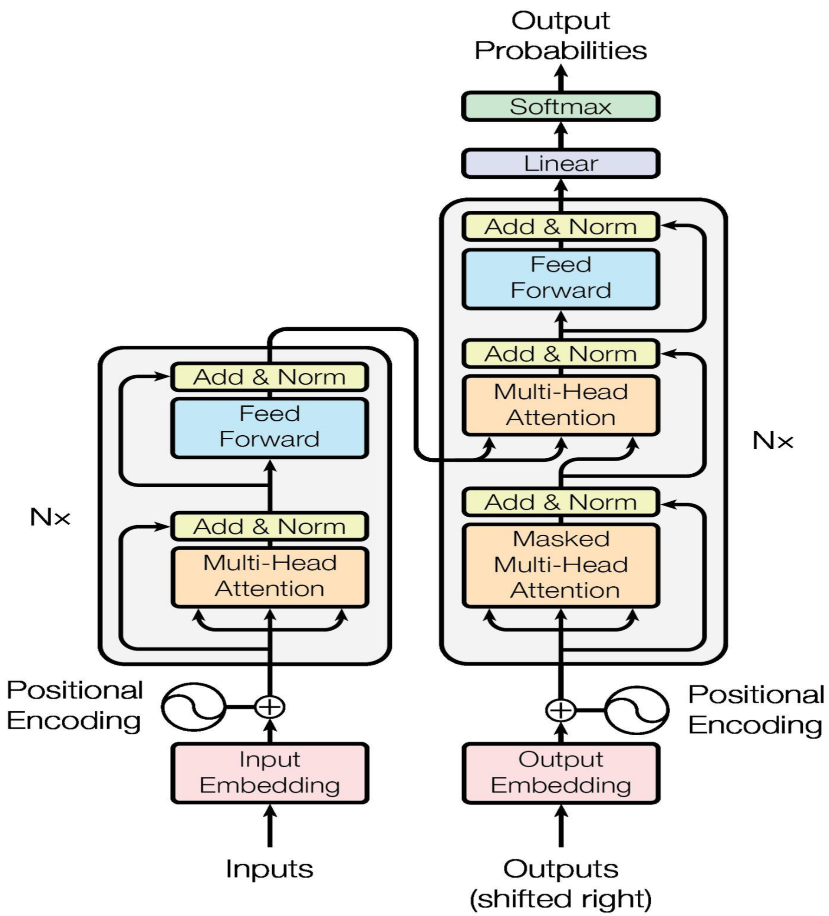
By: Amina Mardiyyah Rufai

African Institute for Mathematical Sciences
SENEGAL

Table of Contents

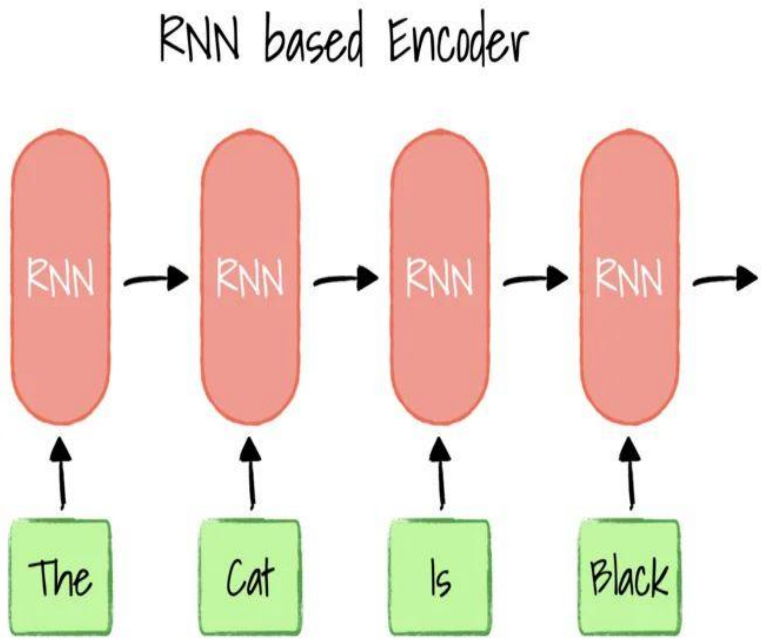
Core Idea of the Paper

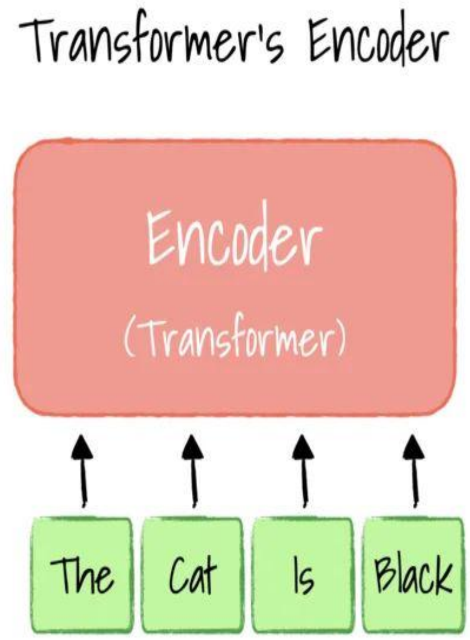
1. Introducing “Parallelization” such that both inputs and expected outputs can be passed into the model simultaneously



Core Idea of the Paper

2. Relying entirely on self-attention to compute representations of its input and output, without using RNN or CNN.





Core Idea of the Paper

The two Key features of the Architecture:

- Multihead Attention (Implemented both in the Encoder and Decoder(Layer)
- The Positional Encoder

The Embedding Class
(similar to word2vec)

Two key things:

- Using the learned linear transformation and softmax function to convert the decoder output to predicted next-token probabilities(implemented in the decoder class)

- In the embedding layer, the learned weights are multiplied by √dmodel.

```
# The Input Embeddings:

class Embeddings(nn.module):
  def __init__(self, vocab_size, dmodel):
    """
    Vocab_Size = Length os entire sequence fed into the network
    """
      super(Embeddings, self).__init__()
      self.embed = nn.Embedding(vocab_size, dmodel)
      self.dmodel = dmodel
  def forward(self, x):
      return self.embed(x) * math.sqrt(self.dmodel)
```

The Positional Encoder Class
(Implemented in both the encoder and decoder layer)

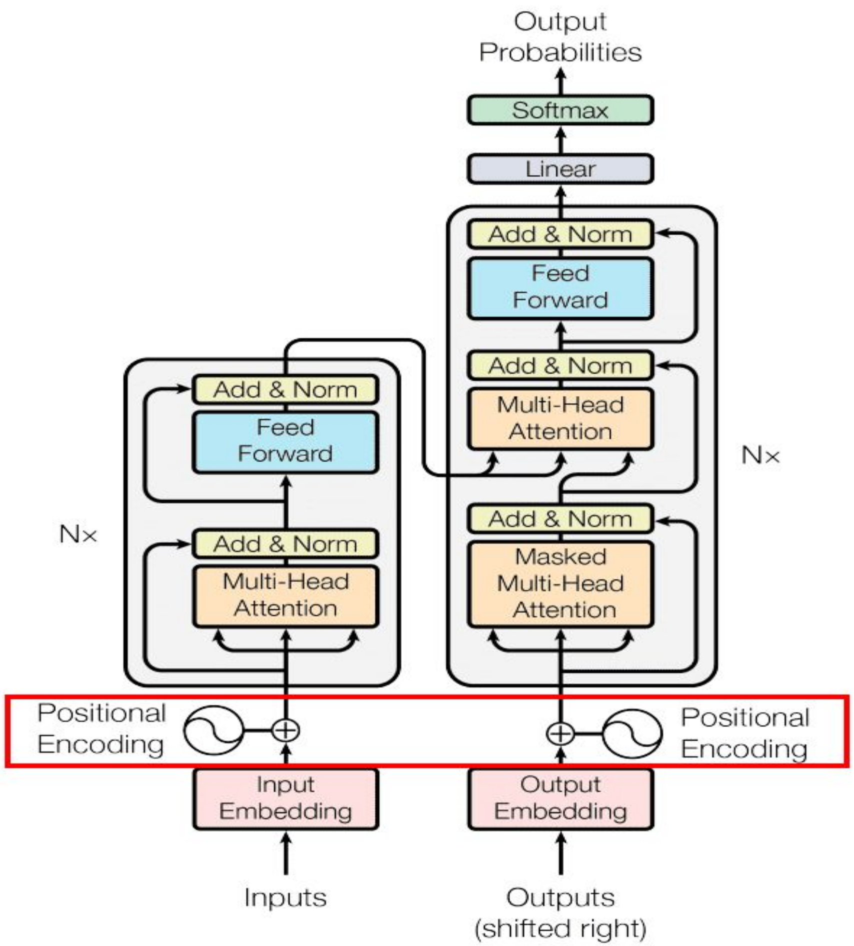
The Idea:

- No CNN or RNN, how is the seq2seq modelling done?

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

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

- The positional encoder is made permutation invariant



The Positional Encoder Class
(Implemented in both the encoder and decoder layer)

Implementation: Things to consider;

- Maximum Length of the sequence
- How each formula works.
- The dimensions of output of the encoder
-

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

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

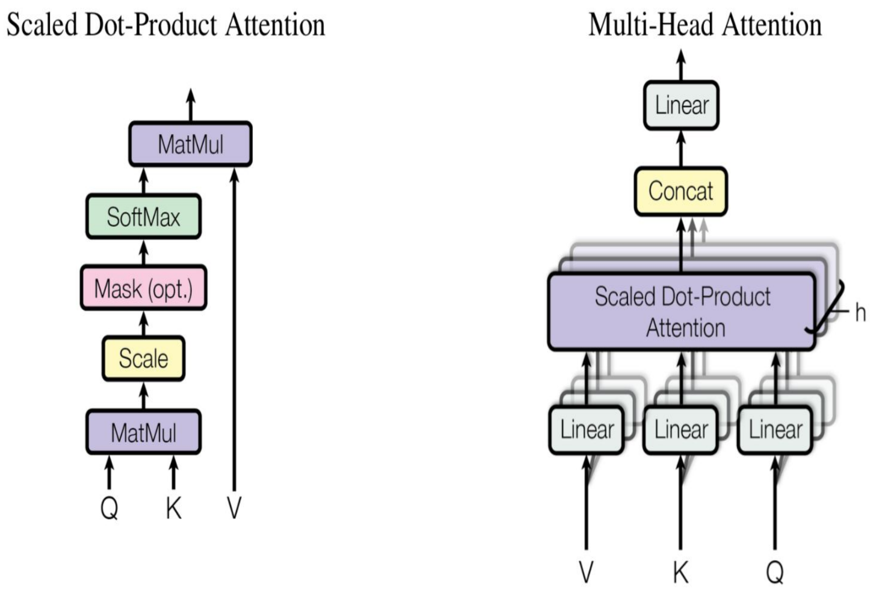
```python
class PositionalEncoder(nn.Module):
    """Implement the PE function.

    PE(pos,2i) = sin(pos/100002i/dmodel)
    PE(pos,2i+1) = cos(pos/100002i/dmodel)
    dim_PE == dmodel
    pos: position of words
    i: dimension

    """
    class PositionalEncoder(nn.Module):
    "Implement the PE function."
    def __init__(self, dmodel, max_len=5000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=0.1)

        # Compute the positional encodings
        pe = torch.zeros(max_len, dmodel)
        pos = torch.arange(0, max_len).unsqueeze(1)
        div_exp_term = torch.exp(torch.arange(0, dmodel, 2) *
                             -(math.log(10000.0) / dmodel))
        pe[:, 0::2] = torch.sin(pos * div_exp_term)
        pe[:, 1::2] = torch.cos(pos * div_exp_term)
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = x + Variable(self.pe[:, :x.size(1)],
                         requires_grad=False)
        return self.dropout(x)
```

The Attention Mechanism

(Scaled Dot Product and Multihead Attention)

1. Scaled Dot Product Attention(the self-attention Mechanism used)

- Compute a dot product
- Scale : dividing by the dimension of the embedding(dmodel)



The Attention Mechanism

(Scaled Dot Product and Multihead Attention)

1. Scaled Dot Product Attention(the self-attention Mechanism used)
- Compute a dot product
- Scale my dividing by the dimension

```python
#SCaled Dot Product Attention
def attention(q, k, v, dk,  dropout=None, mask=None):
    """' Compute 'Scaled Dot Product Attention'
    q-query: bs, n, dmodel
    k-key: bs, n, dmodel
    v-value: bs, n, dmodel
    att_scores = Attention
    Mask optional: As mentioned in the Paper
    Attention(Q, K, V ) = softmax(Q*K.T/√dk)*V

    """
    mat_mult = torch.matmul(q, k.transpose(-2, -1)) / math.sqrt(d_k) #matmul + scaling
    #Optional Mask
    # if mask != None:
    #     mask = mask.unsqueeze(1)
    #     att_scores = scores.masked_fill(mask == 0, -1e9)

    #Softmax Computation
    scores = f.softmax(scores, dim = -1) #softmax

    #Optional Dropout Implementation
    if dropout != None:
        scores = dropout(scores)
    output = torch.matmul(scores, v)    #final matmul with v
    return output
```

The Attention Mechanism

(Scaled Dot Product and Multihead Attention)

1. Multi-Head Attention(the self-attention Mechanism used)
 - Compute a dot product
 - Scale my dividing by the dimension

```python
#SCaled Dot Product Attention
def attention(q, k, v, dk,  dropout=None, mask=None):
    """' Compute 'Scaled Dot Product Attention'
    q-query: bs, n, dmodel
    k-key: bs, n, dmodel
    v-value: bs, n, dmodel
    att_scores = Attention
    Mask optional: As mentioned in the Paper
    Attention(Q, K, V ) = softmax(Q*K.T/√dk)*V

    """
    mat_mult = torch.matmul(q, k.transpose(-2, -1)) / math.sqrt(d_k) #matmul + scaling
    #Optional Mask
    # if mask != None:
    #     mask = mask.unsqueeze(1)
    #     att_scores = scores.masked_fill(mask == 0, -1e9)

    #Softmax Computation
    scores = f.softmax(scores, dim = -1) #softmax

    #Optional Dropout Implementation
    if dropout != None:
        scores = dropout(scores)
    output = torch.matmul(scores, v)    #final matmul with v
    return output
```

AIMS African Institute for Mathematical Sciences SENEGAL

Thank You!