1 Transformers

A very common problem is classification of *sequences*. For instance sentences are sequences of words, time series are sequences of numbers, episodes in games are sequences of actions etc.

Sequences have until recently been processed in machine learning, using Recurrent Neural Nets, where each item in the sequence is fed into a layer together with the output of the layer from the previous item (so a Recurrent Layer takes two inputs)

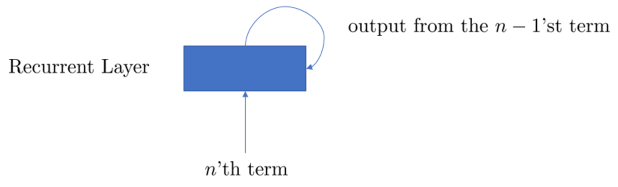


Figure 1: Recurrent Layer

After the last item in the sequence the output of the layer is the output of the sequence. The output (a tensor) is then a representation of the input sequence that can then be processed further to classify the sequence.

One of the problems with this architecture is that for longer sequences, inputs from the beginning of the sequence, tend to be forgotten by the layer by the time we get to the end of the sequence. To remedy this problem some clever recurrent layers have been invented, *Long Short Term Memory (LSTM)* and *Gated Recurrent Unit (GRU)*. These layers contain a number of *gates* that help in recalling earlier inputs in the sequence.

To visualize a recurrent layer it is useful to *unroll* the layer. This is just a way to visualize the recurrent layer, it is still just one layer in the NN

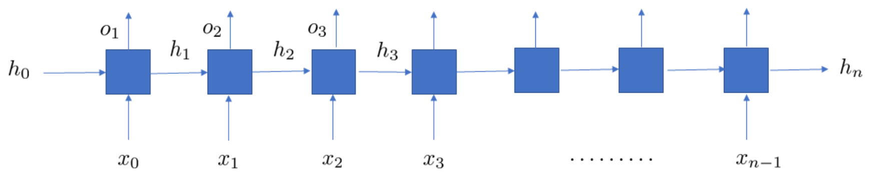


Figure 2: Unrolled Recurrent Layer

Here $X_{0:n-1} = \{x_0, x_1, x_2, \ldots, x_{n-1}$ is the input sequence and the $h_0, h_1, \ldots, h_{n-1}, h_n$ are the *hidden vectors*. The input hidden vector can be anything (for instance the 0-vector).

We have already seen examples on the *Encoder-Decoder* architecture. An example of this using Recurrent layers is the *Seq2Seq* network.

A Seq2Seq network takes a sequence $x_{0:n}$ as input and outputs another sequence $y_{0:m}$

The encoder is a usual RNN (with LSTM or GRU cells). The RNN can have multiple layers and be single or bi-directional

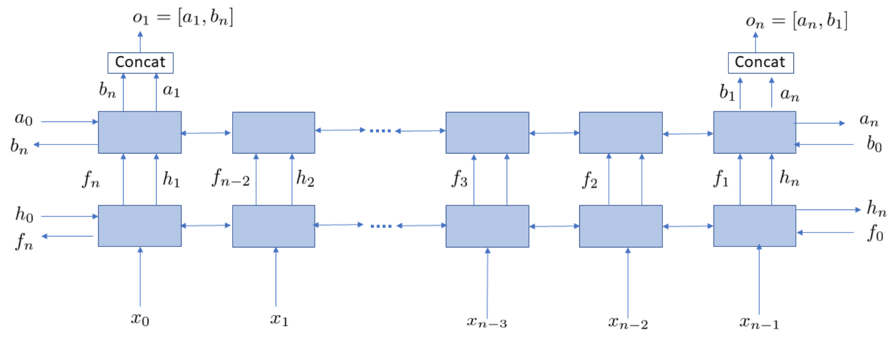


Figure 3: Double Layer Bi-Directional RNN

In fig. 34 the output from the RNN is the concatenation $[a_n, b_n]$

The Decoder is an *Autoregressive* network i.e. at each step the input is the output of the previous step

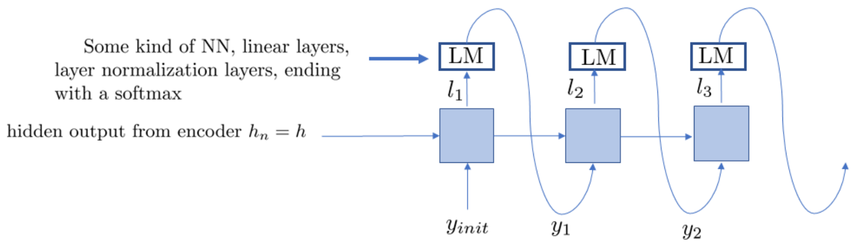

Some kind of NN, linear layers, layer normalization layers, ending with a softmax
hidden output from encoder $h_n = h$
LM
LM
LM
l_1
l_2
l_3
y_{init}
y_1
y_2

Figure 4: Autoregressive Decoder

The output is the sequence of y's, $Y_{1:m} = \{y_1, y_2, \ldots, y_m\}$, the initial y_{init} is usually some token indicating the start of a sentence. The output is thus generated sequentially word for word.

If we are working with a *Natural Language Processing (NLP)* model, the tensors (vectors) in the input sequence will be words and the sequence will be a sentence (or several sentences). The words will come from a *Vocabulary* i.e. a list of possible words. Of course the network does not understand words, so each word is encoded into a vector.

Before the RNN gets the sequence, the sentence is first *Tokenized*, i.e. basically split into words, though sometimes it can be split into smaller tokens. The position of the token in the vocabulary is determined and the encoding vector is computed by the *Encoding* layer, which is basically just a matrix E of dim vocabulary length $\times$ embedding dimension

For instance *Machine Translation* works this way. Say we are translating from English to French. We then have both English and French vocabularies.

An English sentence is tokenized and encoded.

The Encoder RNN (not to be confused with the Encoding layer) outputs a vector which is sent to the Decoder RNN (h in fig.35)

The first step takes the output from the Encoder together with an initial token (y_{init} in fig.35). The *LM* network takes the output from the RNN cell (ℓ_1) and computes a probability distribution over the French vocabulary and y_1 is then a sample from this distribution i.e. a token. Before the next step it is encoded with the French encoder i.e. turned into a vector.

This vector together with the unprocessed output (ℓ_1 in fig 35) is sent to the next step and the process is repeated. This continues until the token y_n for some n is certain token that tells the RNN cell to stop.

Remark that there is an element of randomness in the output from the Decoder: each output token is sampled from a distribution.

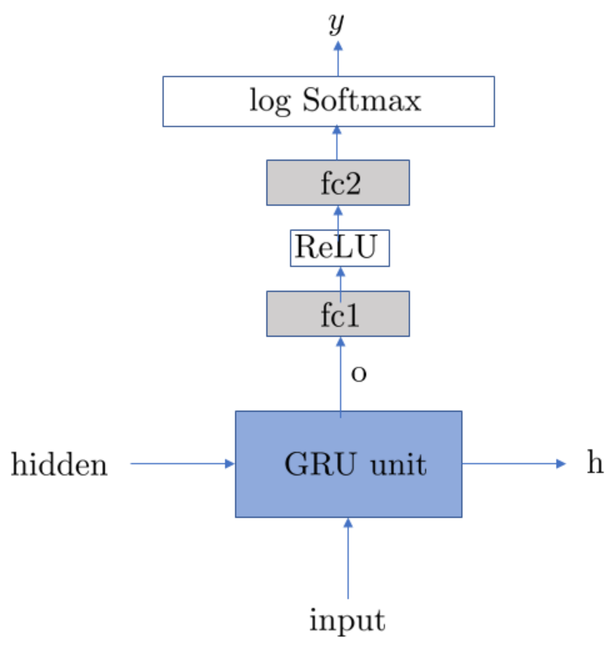


Figure 5: Decoder Step

The hidden vector output from the last step of the Encoder has to contain all the information of the input sequence and we are basically ignoring the output from the previous step.

The *Attention* mechanism is designed to incorporate this data into the inputs to the Decoder.

Consider the following two sentences

I poured water from the bottle into the cup until it was full.

I poured water from the bottle into the cup until it was empty.

In the first sentence *it* clearly refers to *cup* and in the second *it* refers to *bottle*

It is easy for a person to see this but not so for the computer, which only sees two sequences of vectors that only differ on the last position.

The notion of *Attention* is to quantify the the connections between the individual tokens. So in the first sentence we should have strong attention between *it* and *cup* and in the second, strong attention between *it* and *bottle*

The attention mechanism in a *Seq2Seq* network is quite complicated to implement

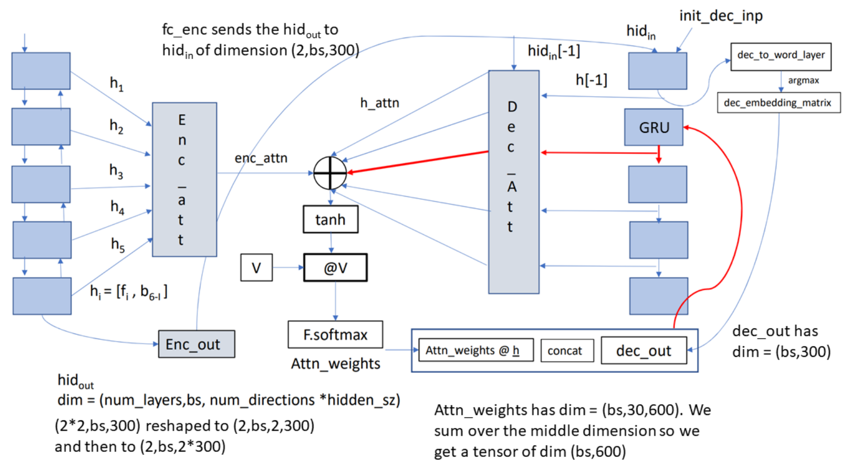
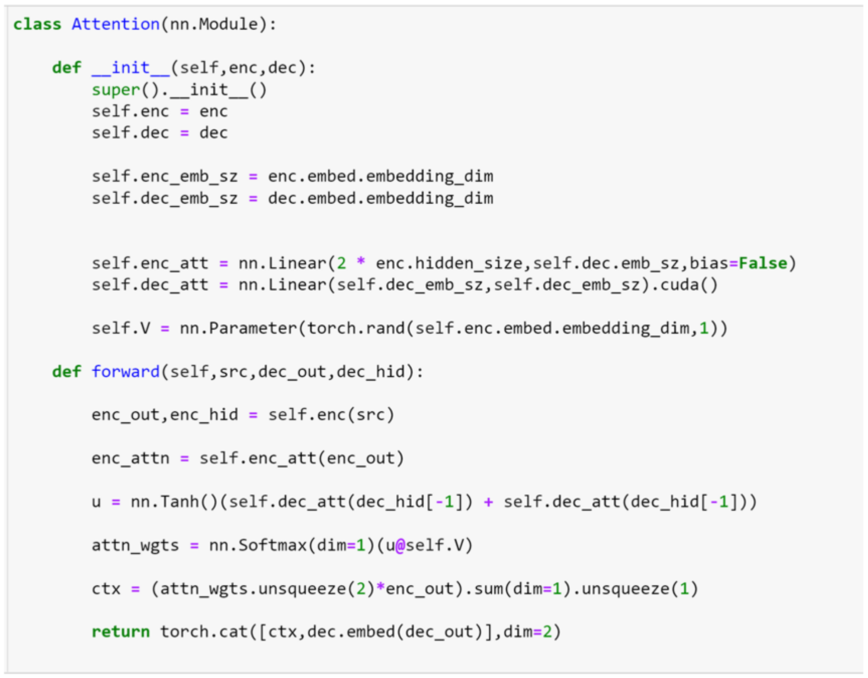

fc_enc sends the hid_out to hid_in of dimension (2,bs,300)
init_dec_inp
hid_in
hid_in[-1]
h[-1]
dec_to_word_layer
argmax
dec_embedding_matrix
h_1
h_2
h_3
h_4
h_5
h_i = [f_i , b_6-i]
Enc_att
enc_attn
h_attn
Dec_Att
GRU
tanh
V
@V
F.softmax
Attn_weights
Enc_out
Attn_weights @ h
concat
dec_out
dec_out has dim = (bs,300)
hid_out dim = (num_layers,bs, num_directions *hidden_sz)
(2*2,bs,300) reshaped to (2,bs,2,300) and then to (2,bs,2*300)
Attn_weights has dim = (bs,30,600). We sum over the middle dimension so we get a tensor of dim (bs,600)

Figure 6: Attention in a Seq2Seq Network

Below is the code for an *Attention* class



```
class Attention(nn.Module):

    def __init__(self,enc,dec):
        super().__init__()
        self.enc = enc
        self.dec = dec

        self.enc_emb_sz = enc.embed.embedding_dim
        self.dec_emb_sz = dec.embed.embedding_dim


        self.enc_att = nn.Linear(2 * enc.hidden_size,self.dec.emb_sz,bias=False)
        self.dec_att = nn.Linear(self.dec_emb_sz,self.dec_emb_sz).cuda()

        self.V = nn.Parameter(torch.rand(self.enc.embed.embedding_dim,1))

    def forward(self,src,dec_out,dec_hid):

        enc_out,enc_hid = self.enc(src)

        enc_attn = self.enc_att(enc_out)

        u = nn.Tanh()(self.dec_att(dec_hid[-1]) + self.dec_att(dec_hid[-1]))

        attn_wgts = nn.Softmax(dim=1)(u@self.V)

        ctx = (attn_wgts.unsqueeze(2)*enc_out).sum(dim=1).unsqueeze(1)

        return torch.cat([ctx,dec.embed(dec_out)],dim=2)
```

Figure 7: Attention Class

The *Transformer* architecture arose out of the realization that Attention was really the essential thing in understanding the meaning of a sentence.

The Transformer was first described in the 2017 paper
"Attention Is All You Need" by Ashish Vaswani and several others from Google Brain.

Like the Seq2Seq, a Transformer based machine translator consists of an Encoder and a Decoder. The Encoder/Decoder architecture has been used for several years.
Both the Encoder and the Decoder consist of several *Multi-head Attention* layers

We begin by describing how a single *Attention Head* processes an input sequence.

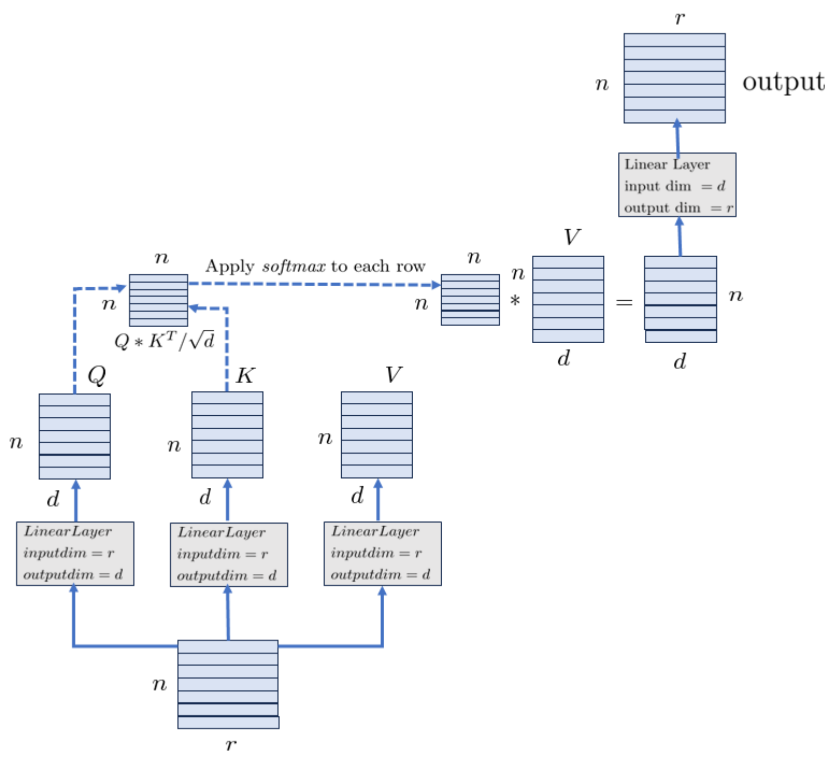


Figure 8: Attention Head

Assume we have an input sequence of length n, tokenized and encoded into vectors of dimension r, so the sequence is a tensor of dimensions $n \times r$.

There are three linear layers of input dimension r and output dimension d, d is called the *model dimension*.

Applying each of these three layers to the input we get three tensors Q, K and V each of dimension $n \times d$

They are called the *Query*, the *Key* and the *Value*.

Since the Query and the Key, both have dimension $n \times d$, we can matrix multiply $Q * K^T$ to get a matrix of dim $n \times n$. We normalize by dividing by $\sqrt{d}$ and then apply the *softmax function* to each of the rows. This produces an $n \times n$ matrix, with rows with positive entries, summing to 1.

Finally we multiply this matrix with the Value matrix to produce an output of dimension $n \times d$

The rationale behind this construction is that we measure distance between vectors by their *dot* product (this is called *cosine similarity*). Two vectors in the same direction has an angle of 0 and so $\cos = 1$, if they have opposite directions i.e. are very different the angle is π so $\cos = -1$.

The matrix product $Q * K^T$ has as entries the dot product between the rows if Q and the rows of K. Taking $softmax$ of the rows and multiplying by V produces a matrix where the entries are weighted sums of the columns of V

In the real model we want to take advantage of *parallelization* so we split the input vectors into h chunks, each of dimension r/h and we apply h attention heads (this requires that r is a multiple of h). The processing by the h attention heads can now be done in parallel. After processing we concatenate all the outputs.

A diagram of a complete *Transformer* model with Encoder and Decoder is shown in fig. 40

The *Multi-Head Attention Layer* is shown in fig. 41

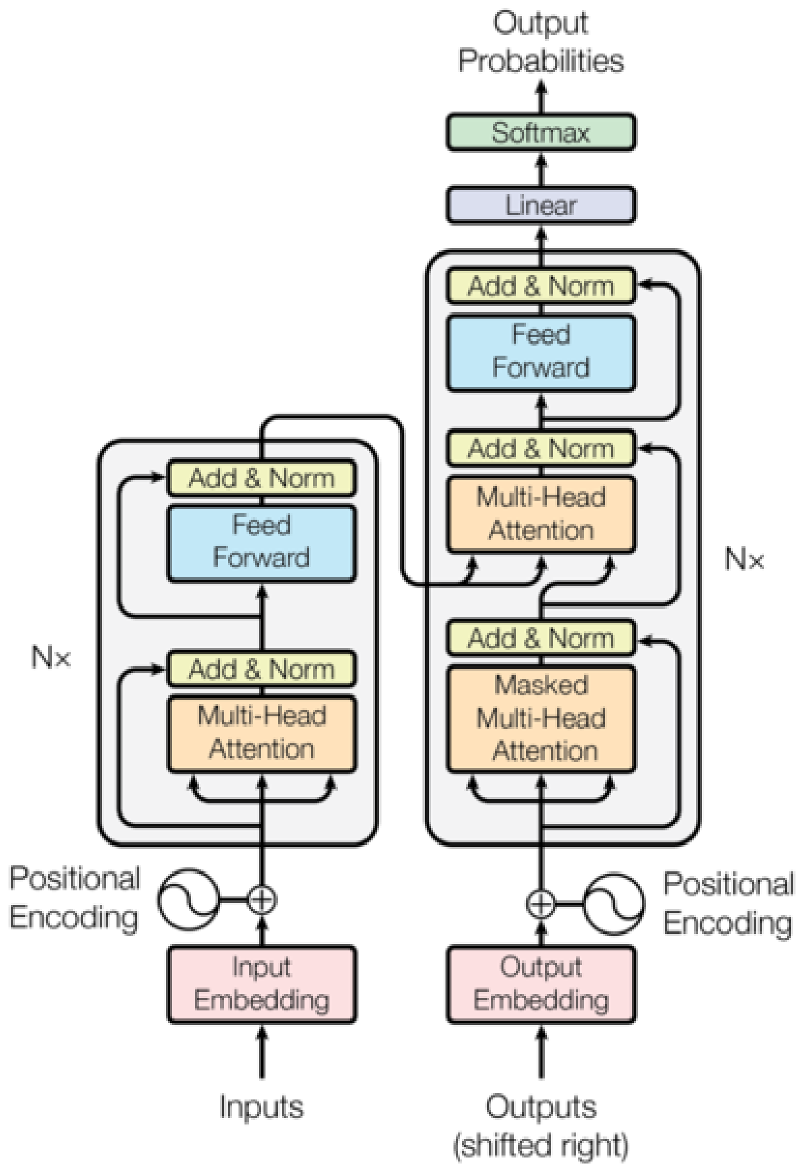


Figure 9: Transformer Encoder and Decoder

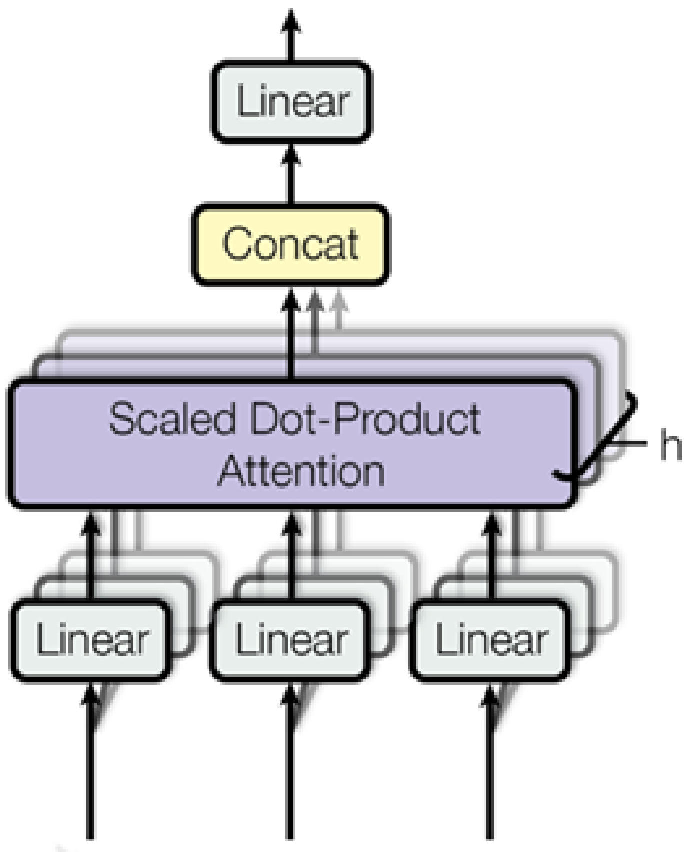


Figure 10: Multi-Head Attention Layer

The only trainable parameters are the parameters of the linear layers. The Scaled Dot Product attention is a straight computation without any gradients

Since the output from the scaled dot product has the same dimension the same as the input, we can add the input tensor and the output tensor. This is the arrow that goes around the block.

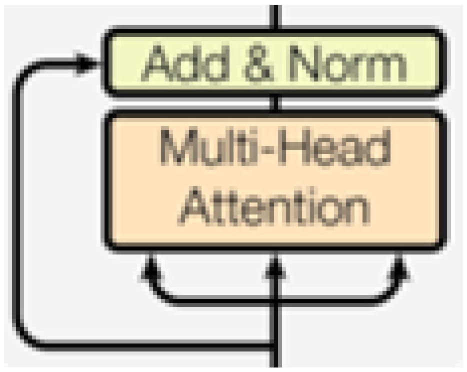


Figure 11: Multi-Head Attention Layer with Residual Connection

The total transformation is then of the form $id+\phi$ where ϕ is all the attention stuff. We are using gradient descent to train the model and the great advantage of this construction is that the gradient is of the form $I+\nabla\phi$ and so the gradient is not going to vanish.

This construction is called a *skip connection* or *residual connection* and greatly improves convergence of the gradient descent algorithm.

The last layer is a *Layer Normalization Layer* which basically normalizes the input to approximately normally distributed.

The final layer in the block is a linear layer followed by a residual and a layer norm layer

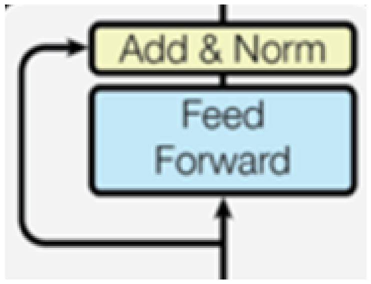


Figure 12: Top Layer of the Block

Since the output dimension is equal to the input dimension these blocks can be stacked (this is the $\times N$) in the diagram in fig. 40.

The model does not make use of the order of the vectors in the sequence.

Think of linear regression over data vectors $X_1, X_2, \ldots, X_n$ and targets y. The order of the data vectors does not matter. But if we for instance had a time-series, the order would be very important.

Clearly the order of the tokens in a sentence and their encodings is very important so we need some way of encoding the relative position in the sequence.

This encoding is done through the *Positional Encoding*

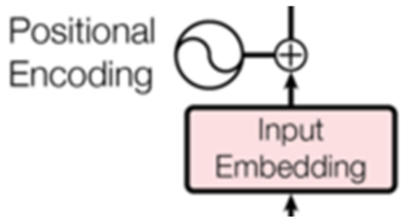


Figure 13: Positional Encoding Layer

The idea is that at each position i in the sentence we have a vector p_i of dim $= emb\_dim$, so p_i only depends on the position.

If X_i is the embedding of the i'th token in the sentence we position encode by simply adding the position vector to the word embedding vector $X_i + p_i$.

So we need to construct vectors of dim $= r$.

Say we are in position i. We define the j'th coordinate

$$p_i(j) = \begin{cases} \sin\left(\dfrac{i}{10000^{j/r}}\right) & \text{if } j \text{ is even} \\ \cos\left(\dfrac{i}{10000^{j-1/r}}\right) & \text{if } j \text{ is odd} \end{cases}$$

for $j = 0, 1, \ldots, r-1$ and $i = 0, 1, 2, \ldots, \text{max sentence length} - 1$ (we can set the *max sentence length* to some suitable number e.g. 1000)

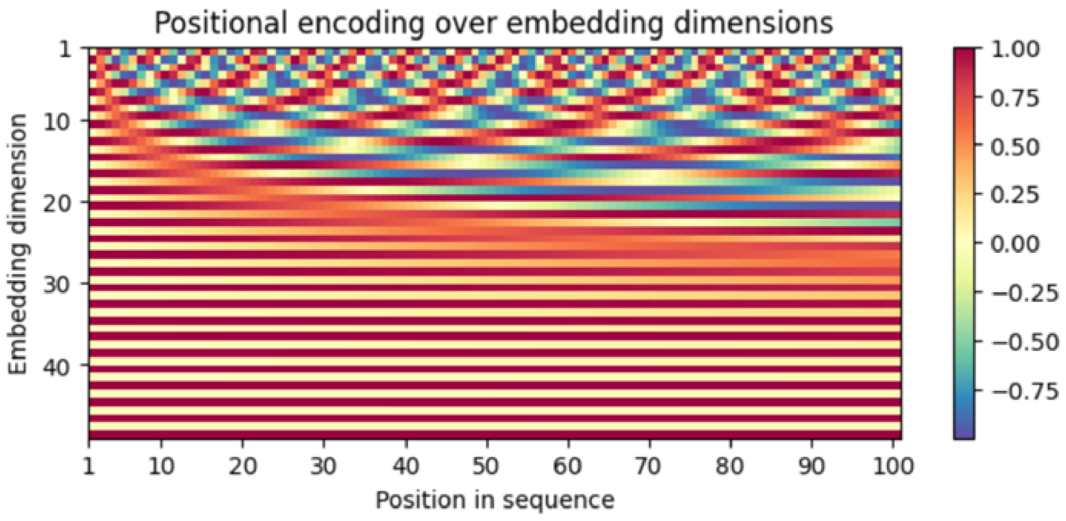


Figure 14: Positional Encoding Vectors

The i'th columns is the positional encoding vector p_i which is added to the encoding vector of the i'th token in the sentence

The Decoder is quite similar to the Encoder. But the job of the Decoder is to generate words. The significant difference is the addition of a *Mask* to the Multi-Head Attention layer.

In the training phase the Decoder inputs a sentence and the Decoder is trained to reproduce this sentence by looking at the attentions between tokens.

A mask is given by an $n \times n$ matrix (where n is the length of the tokenized sentence) with just 0's and 1's as entries.

Like in the Seq2Seq RNN model we want the Decoder the generate sentences by sequentially guess the next token in the sentence conditioned on the previous tokens. Thus the model should only be able to compute attentions to all the previous tokens.

In an attention head, the $n \times n$ matrix $Q * K^T/\sqrt{d}$ is masked out by replacing the i, j'th entry with $-\infty$ if the i, j'th entry in the mask is 0 and left unchanged if the entry in the mask is a 1.

There is a ready made command in Pytorch to achieve this, `masked_fill`

```
weights = Q @ K.transpose
weights = weights.masked_fill(mask[...,:n,:n] == 0,float('-inf'))
```

Assume the i, j'th entry in the masked matrix is $-\infty$. When we take $softmax$ this entry will be 0 so the attention between token i and token j are set to 0.

Multiplying by the value matrix V, the entries in this matrix product are the weighted sums of the rows of V. If the i, j'th entry in $W = softmax(Q*K^T/\sqrt{d})$ is 0 then the i'th row in $W * V$ is

$$\sum_k w_{ik} v_{k\cdot}$$

and $w_{ij} = 0$ so the i'th token in the output has attention 0 with the j'th token.

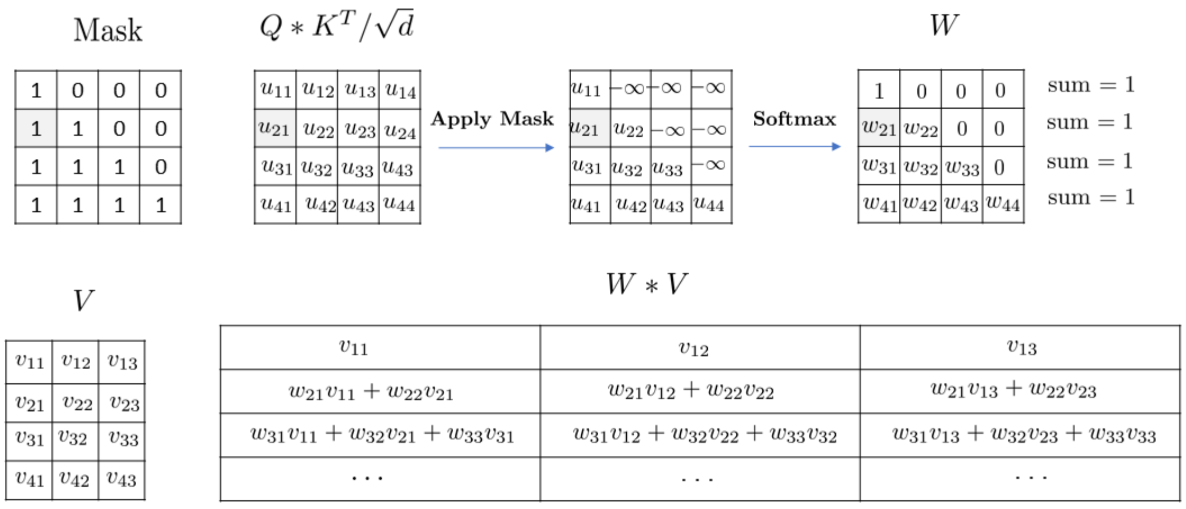


Figure 15: Masking the Attentions

From fig. 46 we can see that for the lower triangular mask (also known as the *auto-regressive* mask) that each of the rows in $W * V$ don't depend on any of the subsequent rows. This means that for the model to predict a token it can only condition of the previous tokens

Both the Encoder and the Decoder can be used as independent models.

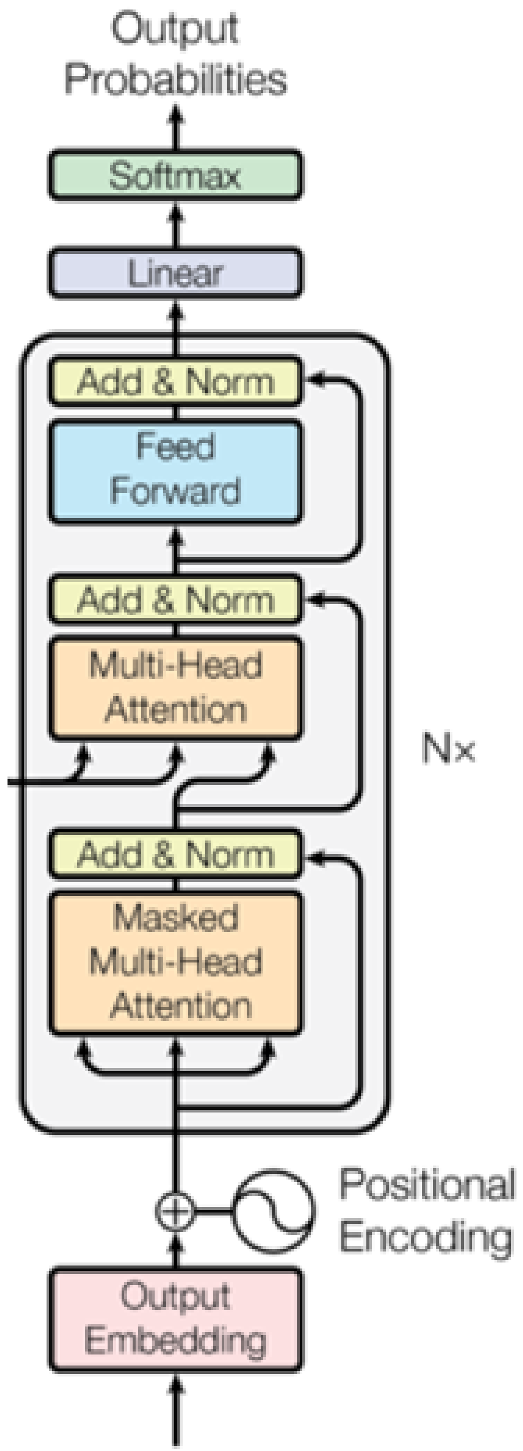


Figure 16: Decoder

If the input to the decoder is the sequence of encoded tokens, positional encoded,

$$a_0, a_1, a_2, \ldots, a_n$$

then the encoder outputs a sequence of conditional probabilities (over the vocabulary)

$$\begin{aligned}
&p(x|a_0)\\
&p(x|a_0, a_1)\\
&p(x|a_0, a_1, a_2)\\
&\vdots\\
&p(x|a_0, a_1 \ldots, a_{n-1})
\end{aligned}$$

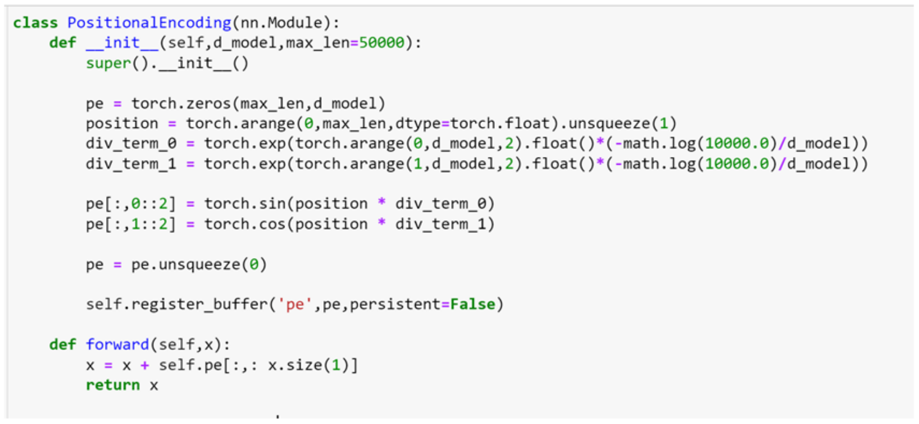
When trained on many sentences (100's of millions), the model is able to predict, for any sequence of tokens, the distribution over the vocabulary conditional on the sequence.

A Large Language Model (LLM) is basically a very large Transformer Decoder, trained on many millions of sentences.

We shall look at the code for a simple implementation of $GPT-2$, one of the earlier $LLMs$. We will take advantage of the predefined layers in Pytorch.

The LLM itself is just a `nn.TransformerEncoder`. We have to specify the configuration of the individual `nn.TransformerEncoderLayer`.
We also have to specify an *Embedding Layer* and a *Positional Encoding*.

Here is the code which implements the algorithm above



```
class PositionalEncoding(nn.Module):
    def __init__(self,d_model,max_len=50000):
        super().__init__()

        pe = torch.zeros(max_len,d_model)
        position = torch.arange(0,max_len,dtype=torch.float).unsqueeze(1)
        div_term_0 = torch.exp(torch.arange(0,d_model,2).float()*(-math.log(10000.0)/d_model))
        div_term_1 = torch.exp(torch.arange(1,d_model,2).float()*(-math.log(10000.0)/d_model))

        pe[:,0::2] = torch.sin(position * div_term_0)
        pe[:,1::2] = torch.cos(position * div_term_1)

        pe = pe.unsqueeze(0)

        self.register_buffer('pe',pe,persistent=False)

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

Figure 17: Positional Encoding Code

The layers in the $GPT-2$ model are masked encoder layers

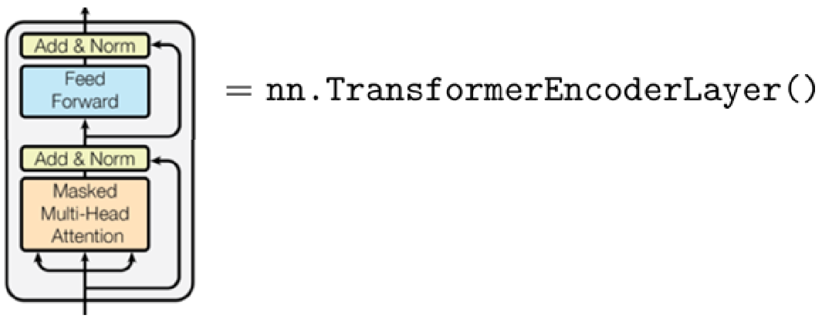


Figure 18: Encoder Layer in GPT-2

The full decoder part of the $GPT-2$ model can be described by the figure below

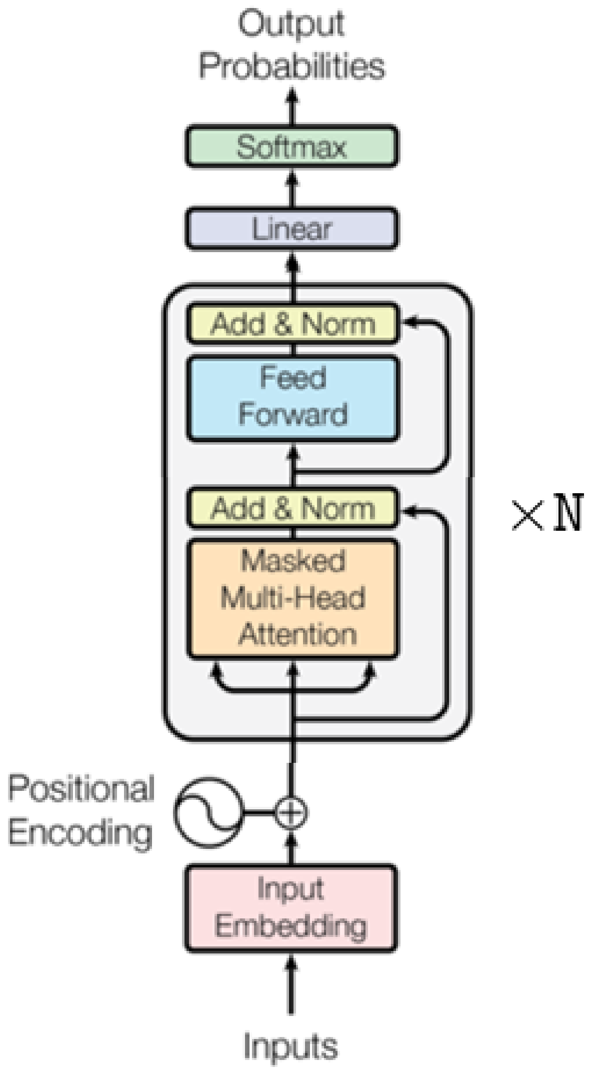
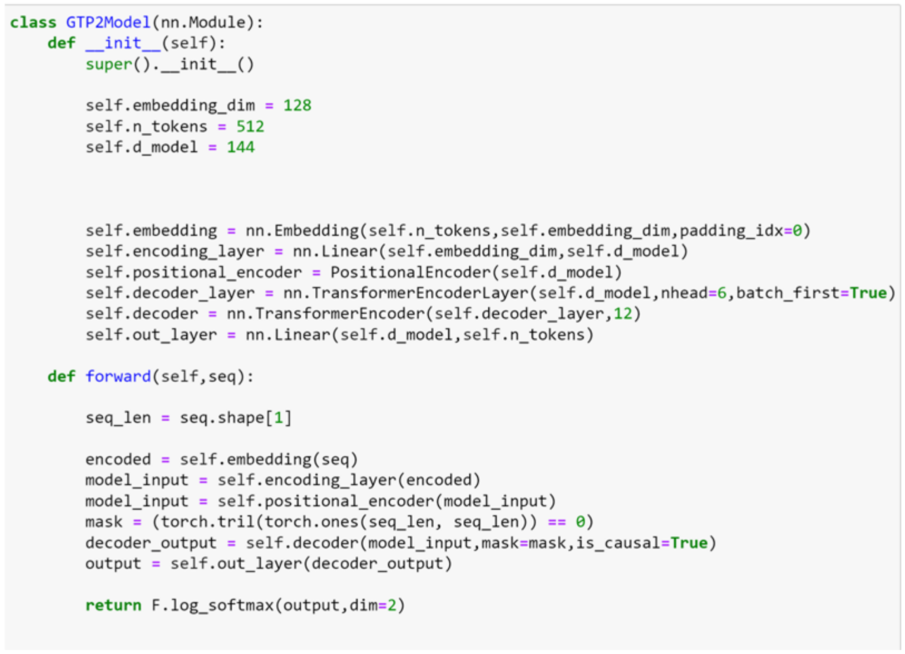


Figure 19: GPT-2



```
class GTP2Model(nn.Module):
    def __init__(self):
        super().__init__()

        self.embedding_dim = 128
        self.n_tokens = 512
        self.d_model = 144



        self.embedding = nn.Embedding(self.n_tokens,self.embedding_dim,padding_idx=0)
        self.encoding_layer = nn.Linear(self.embedding_dim,self.d_model)
        self.positional_encoder = PositionalEncoder(self.d_model)
        self.decoder_layer = nn.TransformerEncoderLayer(self.d_model,nhead=6,batch_first=True)
        self.decoder = nn.TransformerEncoder(self.decoder_layer,12)
        self.out_layer = nn.Linear(self.d_model,self.n_tokens)

    def forward(self,seq):

        seq_len = seq.shape[1]

        encoded = self.embedding(seq)
        model_input = self.encoding_layer(encoded)
        model_input = self.positional_encoder(model_input)
        mask = (torch.tril(torch.ones(seq_len, seq_len)) == 0)
        decoder_output = self.decoder(model_input,mask=mask,is_causal=True)
        output = self.out_layer(decoder_output)

        return F.log_softmax(output,dim=2)
```

Figure 20: GPT-2 Decoder Code

There is of course more to the model.
We have to have a *Vocabulary* and a *Tokenizer* and we have to write a training loop

The model is trained using a large collection of sentences.
Each sentence is tokenized and is encoded as a sequence of the indices in the vocabulary of the tokens.

The sequence of indices is encoded as vectors by indexing into the rows of the Embedding matrix, `nn.Embedding(n_tokens,embedding_dim)`

Let $s_0, s_1, \ldots, s_n$ be this sequence of vectors, each s_i has dimension = `embedding_dim`.

There may be a linear layer, `nn.Linear(embedding_dim,d_model)` but we will ignore this and assume `embedding_dim = d_model`

The `PositionalEncoding` object generates positional vectors $p_0, p_1, \ldots, p_n$ and the sequence of vectors is encoded by $s_i \to s_i + p_i = v_i$

We skip the last vectors and run the sequence $v_0, v_1, \ldots, v_{n-1}$ through the GPT2Model. This produces a matrix of $n-1$ probability distributions over the rows of the embedding matrix.

The loss is the `NLLLoss`, the negative log-likelihood

As an example assume the batch dimension is 1, the sequence length is 32 and the sequence of indices with the last entry skipped is the sequence of integers of length 31

```
tensor([[ 55, 349, 344, 340, 248, 185, 385, 295, 235, 288, 460,
69, 506, 76, 263, 484, 107, 475, 183, 136, 165, 399, 49, 60, 365, 508,
409, 266, 327, 455, 360]]
```

The sequence of 128-dimensional embedding vectors is

```
tensor([[[ 0.2332, 0.3238, 0.3962, ..., 0.3544, -0.5660, 0.0352],
[-1.0464, -0.3580, 0.6031, ..., -0.8632, 0.3152, -0.8056], [ 0.0434,
0.9078, 0.4252, ..., -0.8134, 0.7593, -0.3334], ..., [ 0.0856, 0.4225,
-0.7702, ..., 0.0386, -1.1915, -0.3743], [ 0.3835, -0.1794, 0.0523,
..., 0.7691, -0.5199, -0.0756], [-0.9753, 0.0934, 0.1699, ..., -0.1787,
0.0256, 0.0690]]], grad_fn=<EmbeddingBackward0>)
```

and the positional encoded vectors

```
tensor([[[ 0.2332, 0.3238, 0.3962, ..., 0.3544, -0.5660, 0.0352],
[-1.0464, -0.3580, 0.6031, ..., -0.8632, 0.3152, -0.8056], [ 0.0434,
0.9078, 0.4252, ..., -0.8134, 0.7593, -0.3334], ..., [ 0.3835, -0.1794,
0.0523, ..., 0.7691, -0.5199, -0.0756], [-0.9753, 0.0934, 0.1699, ...,
-0.1787, 0.0256, 0.0690], [-1.0479, -0.4880, 0.9205, ..., 0.6270, 0.2469,
-0.1215]]], grad_fn=<ViewBackward0>)
```

The output from the `gpt` model is

```
tensor([[[-6.5559, -6.9626, -6.6778, ..., -6.7882, -7.0207, -6.3044],
[-7.0115, -7.4990, -6.2573, ..., -6.3470, -6.5157, -6.0783], [-6.6251,
-7.6515, -6.4477, ..., -6.6678, -6.8381, -6.3857], ..., [-6.0543, -7.3394,
-6.2361, ..., -6.8638, -5.9808, -6.8644], [-6.0439, -7.4653, -6.4409,
..., -6.6636, -6.1591, -6.9721], [-6.0961, -7.5900, -6.4112, ..., -6.6708,
-6.0003, -6.8073]]], grad_fn=<LogSoftmaxBackward0>)
```

The target is the original sequence with the *first* entry skipped.

```
tensor([[349, 344, 340, 248, 185, 385, 295, 235, 288, 460, 69, 506,
76, 263, 484, 107, 475, 183, 136, 165, 399, 49, 60, 365, 508, 409,
266, 327, 455, 360, 357]])
```

and we get the loss

```
nn.NLLLoss()(output.view(output.shape[0]*output.shape[1],-1),inputs.reshape(-1))
= tensor(6.5316, grad_fn=<NllLossBackward0>)
```