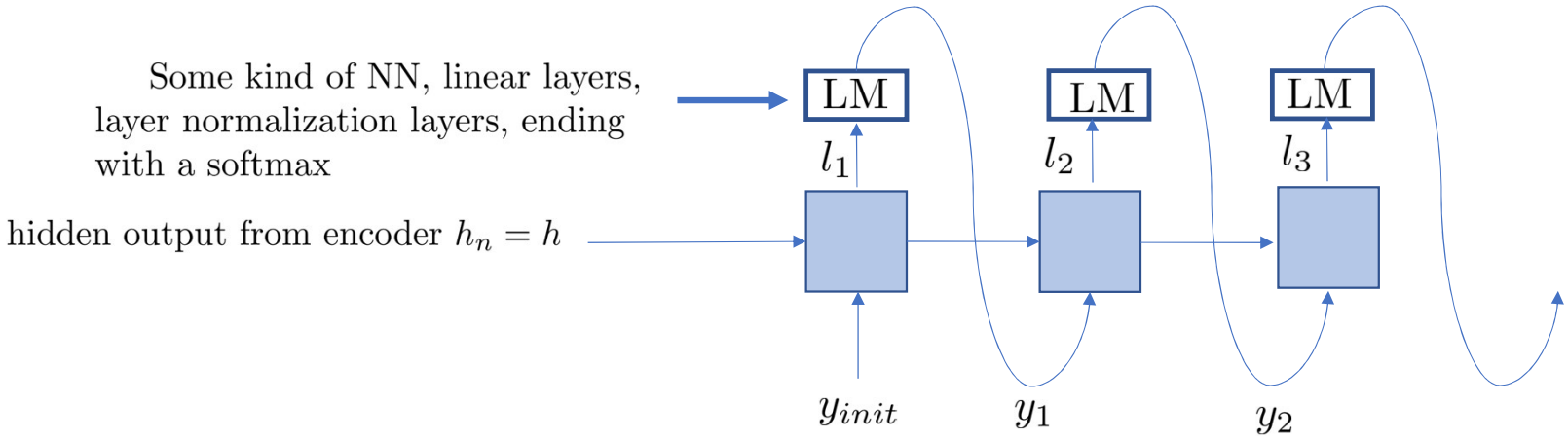
Generative Models

Lecture 7

Last time we coded a machine translation model English-French using a Seq2Seq model. It is an Encoder/Decoder model.

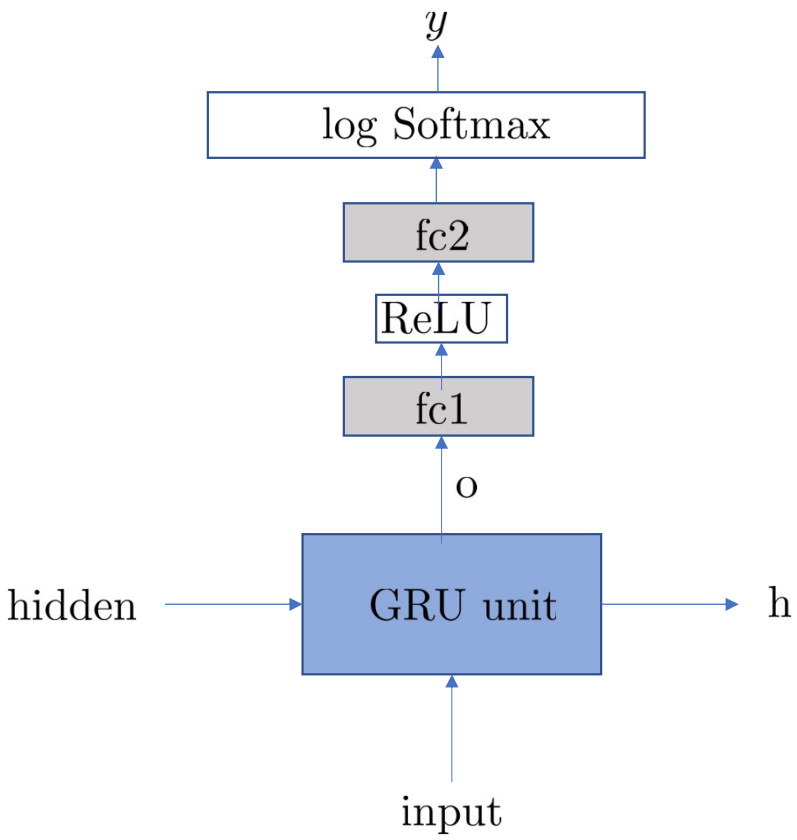
The Decoder is an AutoRegressive model



Each step in the Decoder looks like

```
class DecoderRNN(nn.Module):
    def __init__(self,vocab_size, hidden_size):
        super().__init__()
        self.hidden_size = hidden_size
        self.vocab_size = vocab_size

        self.embedding = nn.Embedding(vocab_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)
        self.fc1 = nn.Linear(hidden_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size,self.vocab_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        input_emb = self.embedding(input)
        o, h = self.gru(input_emb, hidden)
        l1 = F.relu(self.fc1(o))
        l2 = self.fc2(l1)
        y = self.softmax(l2)

        return y, h
```

y is a vector of dim $= len(fra - vocab)$
of the log probabilities of $Softmax(l_2)$



To train the network we need a *loss function*. The input is a `tensor_pair` consisting of a sentence in english (as a tensor of indices) and the translation into french.

The `rnn` network only uses the length of the translation and it outputs the 2-dim tensor of the $\log probs$, the i'th row is the output y_i corresponding to the i'th word in the translation `tensor_pair[1]`

The *loss* is the sum of the KL-divergencies. If the i'th word in the translation has index k_i (in the vocab) then the KL-divergence between the distribution over the vocab produced at the i'th step by the *decoder* and the distribution that has all its probability concentrated at k_i is precisely

$$-y_i[k_i] = -1 \cdot \log prob_i[k_i]$$

This is the `nn.NLLLoss()` (*Negative Log Likelihood Loss*) pytorch loss function.

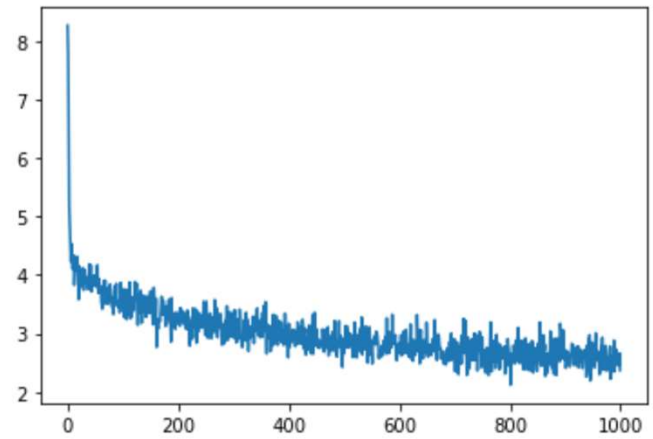
Training Loop

```
batch_losses = []
for k in range(1000):

    batch = random.sample(tensor_pairs,batch_sz)
    losses = []
    for tensor_pair in batch:

        optimizer.zero_grad()

        out = rnn(tensor_pair)

        loss = criterion(out,tensor_pair[1])

        loss.backward()

        optimizer.step()

        losses.append(loss.item())
    print(np.mean(losses))
    batch_losses.append(np.mean(losses))
```

```
1 optimizer = Adam(rnn.parameters(),lr=0.0001)
```

```
1 batch_sz = 32
```

```
1 plt.plot(batch_losses);
```



A major application of recurrent neural networks has been *Natural Language Processing (NLP)* where we try to teach the machine to understand human readable sentences, some applications are word completions, machine translation etc.

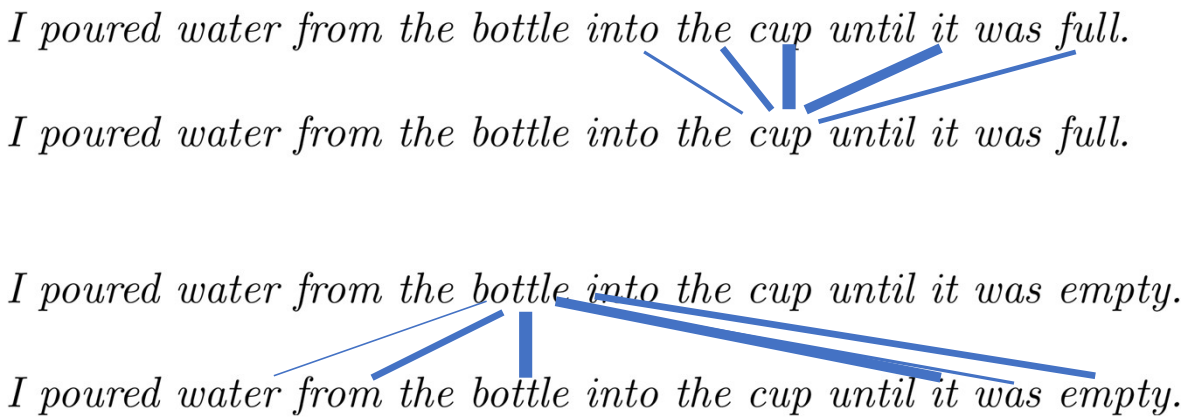
Human sentences can be hard for a machine to understand. For instance consider the two sentences

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* refers to *the cup* and in the second *it* refers to *the bottle.*

Viewing the sentences as purely sequences of words, a machine cannot determine what *it* refers to.

Attention is a mechanism to assign a measure of how much the each word relates to every other word in the sentence

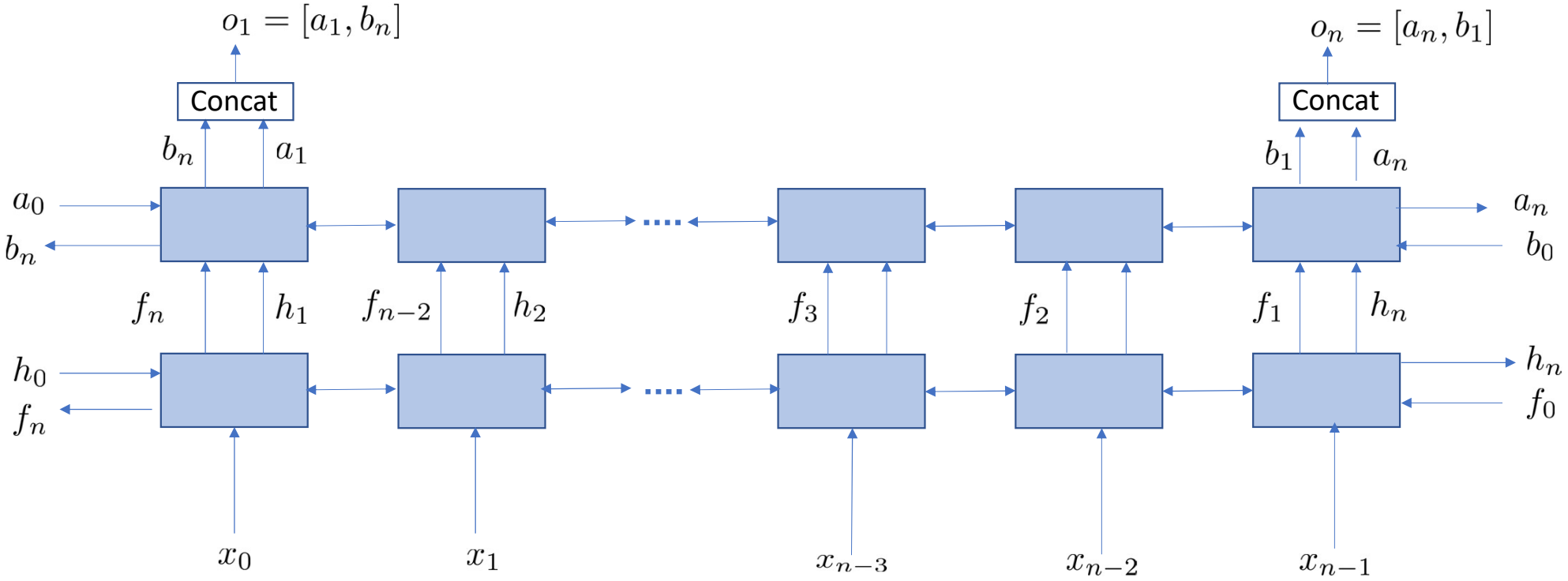


The weights of the lines indicate how strong the attention is between words (only some lines are drawn in reality there are lines from each word to every other word)

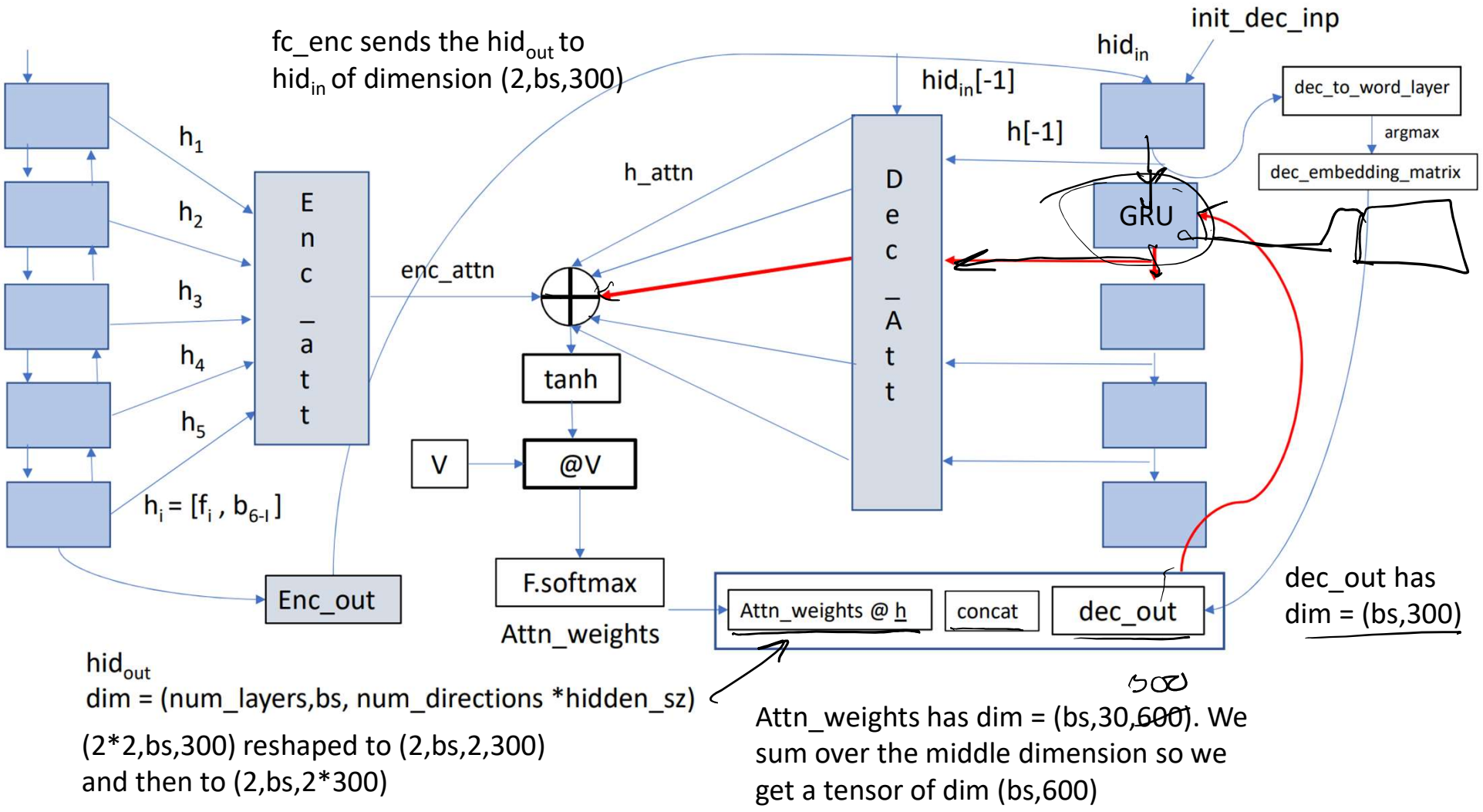
In the first sequence the attention of the word *it* to the word *cup* would be higher than the attention to the word *bottle* while in the second sentence it would be the other way round.

Attention has been combined with recurrent neural nets and the combination has been succesful for instance in machine translation and sentiment analysis.

To implement *Attention* in the *Seq2Seq* model we are going to use all the outputs from the *Encoder* and not just the final hidden output. Also in the *Encoder* part we will use a *bi-directional* RNN, where we use both the sequence and the reverse sequence as inputs and a 2-layer network.



The hidden output from a bidirectional layer is $[h_n, f_n]$


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
GRU
h_attn
h1
h2
h3
h4
h5
Enc_att
enc_attn
Dec_Att
tanh
V
@V
h_i = [f_i , b_6-i]
Enc_out
F.softmax
Attn_weights
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)

We now need the outputs from the *Encoder* rather that just the last hidden state

```
class Encoder(nn.Module):
    def __init__(self,vocab,hidden_size,n_layers=2,bidirectional=True,
                 dropout=0.5):
        super().__init__()
        self.vocab = vocab
        self.hidden_size = hidden_size
        self.n_layers = n_layers
        self.embed = nn.Embedding(len(vocab.itos),300)
        self.gru = nn.GRU(self.embed.embedding_dim,hidden_size,num_layer

    def forward(self,src,hidden=None):
        bs = src.size(0)
        hidden = torch.zeros(2*self.n_layers,bs,self.hidden_size).cuda()

        emb = self.embed(src)

        out,h = self.gru(emb,hidden)

        return out,h
```

`out` is the sequence of outputs from each step in the `Encoder`

$o_1, o_1, \ldots, o_n$

Now we return the sequence of outputs and the hidden state, rather than just the hidden state

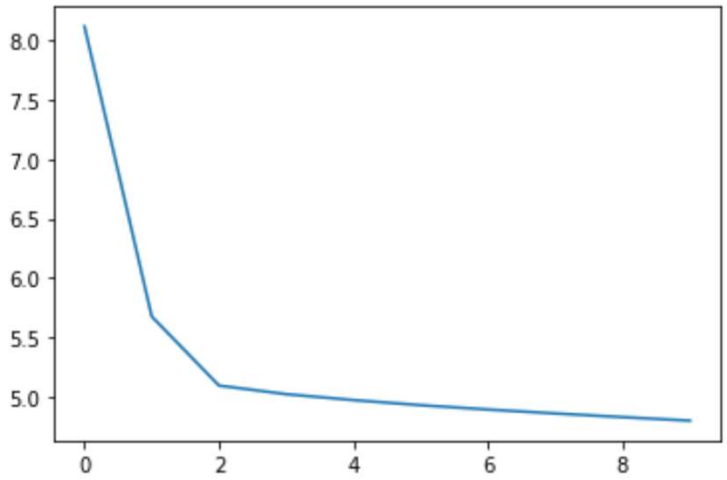
We can put all the attention stuff into a separate module

```
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)
```

Training loop (takes a very long time to train, if you don't have cuda, don't even try it)

```
epochs = range(10)
epoch_losses = []
for e in epochs:
    losses = []
    for i, (src,tgt) in enumerate(dl):

        out = model(src,tgt)

        loss = loss_fn(out,tgt)

        optim.zero_grad()

        loss.backward()

        optim.step()

        losses.append(loss.item())

    epoch_loss = np.mean(losses)
    epoch_losses.append(epoch_loss)
    print(epoch_loss)
```

```
1  plt.plot(epoch_losses);
```



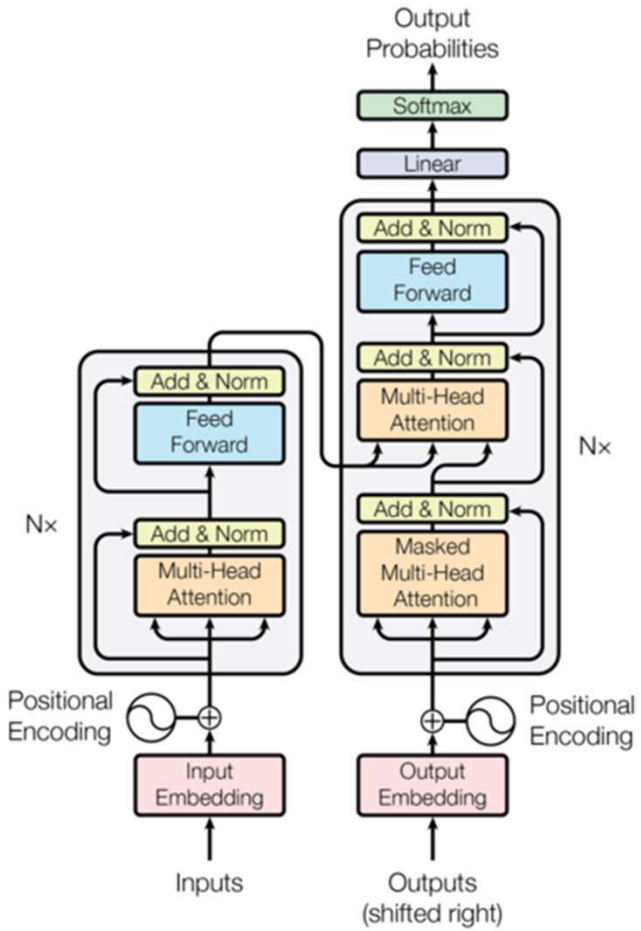
Transformers

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


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

Again we first have to encode each word into a tensor and each sentence into a sequence of tensors.

This is the purpose of the *Embedding Layer* in the diagram. This layer assigns to each word (enumerated in a dictionary which can have hundreds of thousands of words) and assigns to it a vector (1-dim tensor) of some fixed dimension. This works essentially the same way as in the previous model

We will return to the *Positional Encoding* in a later slide and go to the *Multi-head Attention Layer*

We have to define some measure of attention between the words (encoded as tensors) in the sentence.

We load the whole sentence (appropriately padded) in as a tensor of dimension $(seq\_length, embedding\_dim)$

The first layer consist of three linear layers `nn.Linear(embedding_dim,model_dim)` where the `model_dim` is an internal dimension, specified by the model. In many cases it is the same as the `embedding_dim`

```
emb_dim = 12
model_dim = 16
seq_len = 20

W_q = nn.Linear(emb_dim,model_dim)
W_k = nn.Linear(emb_dim,model_dim)
W_v = nn.Linear(emb_dim,model_dim)
```

```
print(q.shape)
print(k.shape)
print(v.shape)
```

```
torch.Size([20, 16])
torch.Size([20, 16])
torch.Size([20, 16])
```

These tensors are known as the *query, key* and *value* tensors.

Each of these have to be split equally among the h heads and so it is necessary that h divides *emb_dim*

```
h = 4
q_ = q.chunk(h,dim=-1)
k_ = k.chunk(h,dim=-1)
v_ = v.chunk(h,dim=-1)
```

```
q_[0].shape
```

```
torch.Size([20, 4])
```

In our case we assume we have 4 Scaled Dot-Product Attention units and the q,k,v tensors are each split into a 4 $(=h)$ tuple of tensors of dim $seq\_len, emb\_dim/h$.

We can split using the Pytorch command `chunk`

```
scaled_dot_products = []

for i in range(h):
    dot = (q_[i] @ k_[i].T)/np.sqrt(emb_dim/h)
    scaled_dot_products.append(dot)
```

```
scaled_dot_products[0].shape
```

```
torch.Size([20, 20])
```

We measure similarity of vectors using a scaled dot product so we dot each of the *seq_len* rows in each $q[i]$ with each of the *seq_len* rows in $k[i]$, $i = 1, \ldots, h$.

We scale by dividing by $\sqrt{emb\_dim/h}$. We can compute all these dot products by taking the matrix product $\frac{1}{\sqrt{emb\_dim/h}} q[i] \cdot k[i]^T$

```
softmax = nn.Softmax(1)
```

```
attn_wgts = []
for m in scaled_dot_products:
    attn_wgts.append(softmax(m))
```

Next we apply the *Softmax* function to each of the rows in these $seq\_len \times seq\_len$ matrices.

The entries in each row are positive and sum to 1

```
weighted_values = []
for i in range(h):
    weighted_values.append(attn_wgts[i] @ v_[i])
```

```
weighted_values[0].shape
```

```
torch.Size([20, 4])
```

The value matrices $v[i]$, $i = 1, \ldots, h$ have dimensions $seq\_len \times (model\_dim/h)$ and so we can multiply each matrix $attn\_wgts[i]$ with the corresponding $v[i]$ to get tensors of dimensions $seq\_len \times (model\_dim/h)$

```
out = torch.cat(weighted_values,dim=1)
```

```
out.shape
```

```
torch.Size([20, 16])
```

Finally we can concatenate these tensors along the last dim (this means we are concatenating the rows which each has dimension emb_dim/h)

```
W_o = nn.Linear(emb_dim,emb_dim)
o = W_o(out)
```

```
o.shape
```

```
torch.Size([20, 16])
```

We add a linear layer on top

```
q_[2].size()
```

```
torch.Size([20, 4])
```

```
k_[2].T.size()
```

```
torch.Size([4, 20])
```

```
scaled_dot_products = []

for i in range(h):
    dot = (q_[i] @ k_[i].T)/np.sqrt(emb_dim/h)
    scaled_dot_products.append(dot)
```

```
scaled_dot_products[0].size()
```

```
torch.Size([20, 20])
```

```
scaled_dot_products[0].shape
```

```
torch.Size([20, 20])
```

Remark that we get h different attention matrices, one for each head. This gives $h * (20 * 20)$ parameters

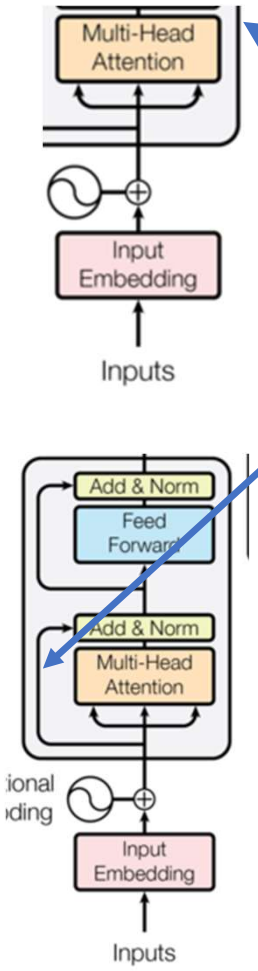
```
softmax = nn.Softmax(1)
```

```
attn_wgts = []
for m in scaled_dot_products:
    attn_wgts.append(softmax(m))
```

```
attn_wgts[0].sum(1)
```

```
tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
        1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
        1.0000, 1.0000], grad_fn=<SumBackward1>)
```

```
weighted_values = []
for i in range(h):
    weighted_values.append(attn_wgts[i] @ v_[i])
```



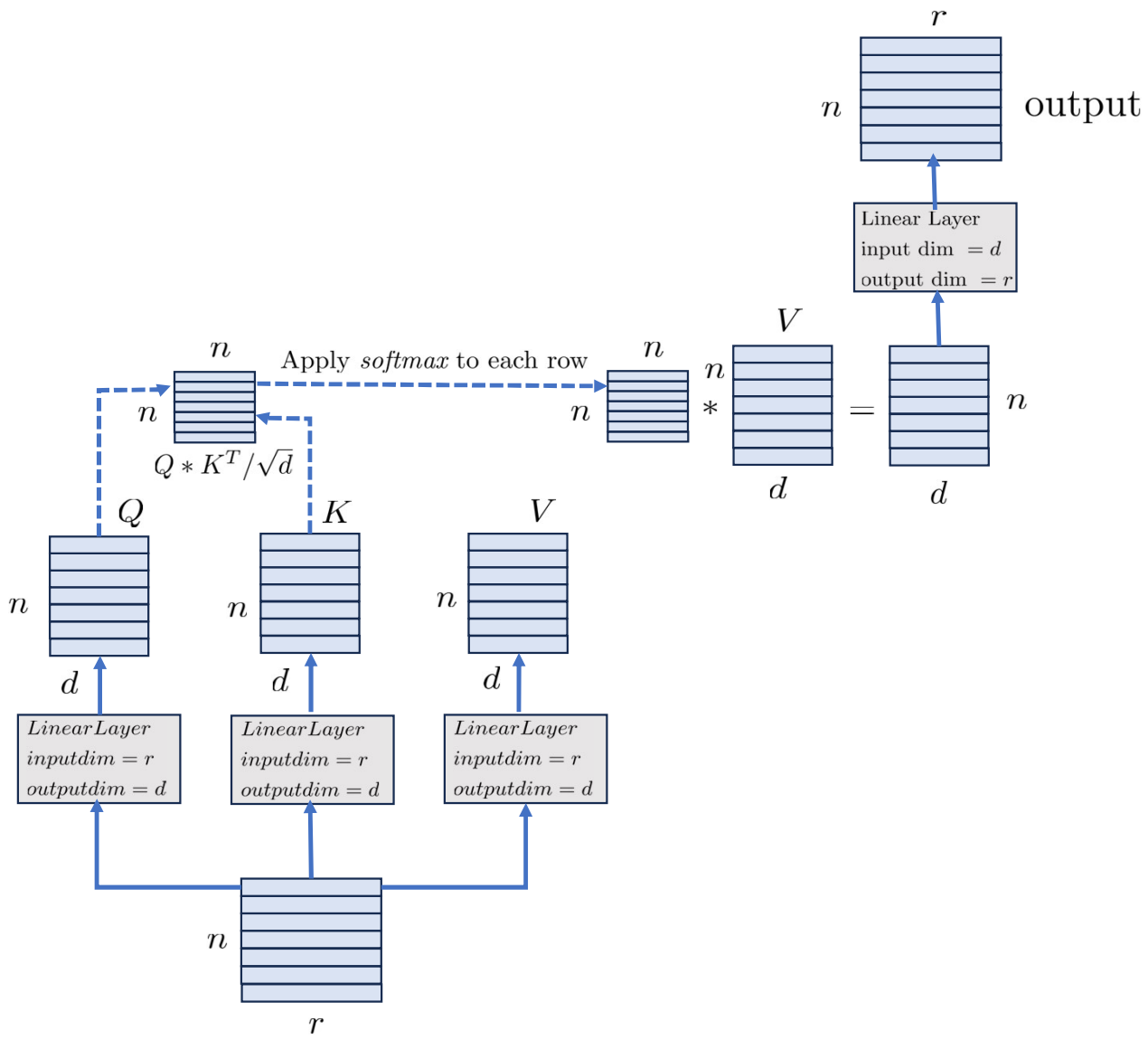
The only trainable parameters are the parameters of the linear layers. The Scaled Dot Product attention is a straight computation without any gradients

This is where we are now

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.

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.


r
n
output
Linear Layer
input dim = d
output dim = r
V
n
n
Apply softmax to each row
n
n
n
*
=
n
d
d
$Q * K^T/\sqrt{d}$
Q
K
V
n
n
n
d
d
d
LinearLayer
inputdim = r
outputdim = d
LinearLayer
inputdim = r
outputdim = d
LinearLayer
inputdim = r
outputdim = d
n
r

```
weighted_values = []
for i in range(h):
    weighted_values.append(attn_wgts[i] @ v_[i])
```

```
weighted_values[0].shape
```

```
torch.Size([20, 4])
```

```
out_h = torch.cat(weighted_values,dim=1)
```

```
out_h.shape
```

```
torch.Size([20, 16])
```

```
W_o = nn.Linear(model_dim,emb_dim)
o = W_o(out_h)
```

```
o.shape
```

```
torch.Size([20, 12])
```

```
o = input_seq + o
```

Since o has the same size as the input sequence we can form the sum (the skip connection)

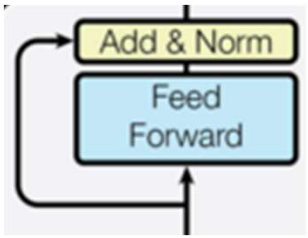
```
ln_1 = nn.LayerNorm(12)
```

```
o = ln_1(o)
```

Applying Layer Norm

```
o.shape
```

```
torch.Size([20, 12])
```



The final layer in the module is a simple feed forward net with two linear layers, followed by a skip and a layer norm

```
ff_layer = nn.Sequential(nn.Linear(12,2024),
                         nn.ReLU(),
                         nn.Linear(2024,12))
```

```
ln_2 = nn.LayerNorm(12)
```

```
output = ln_2(o + ff_layer(o))
```

```
output.shape
```

```
torch.Size([20, 12])
```

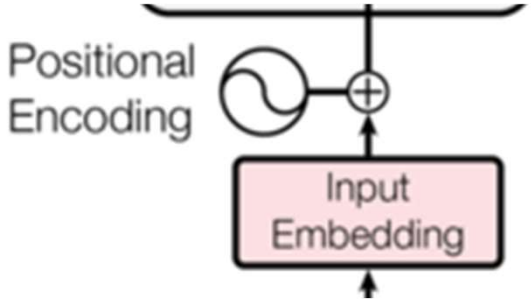
Since the `output` has the same shape as the input we can stack as many of these modules on top of each other, at each step, the `output` becomes the `input` to the next module

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 in a sentence is very important so we need some way of encoding the relative position in the sequence.

This encoding is done through the *Positional Encoding*



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 word in the sentence we position encode by simply adding the position vector to the word embedding vector $X_i + p_i$

```
class PositionalEncoder(nn.Module):
    def __init__(self,emb_dim,dropout=0,max_sent_len=5000):
        super().__init__()

        self.emb_dim = emb_dim
        self.dropout = nn.Dropout(dropout)
        self.max_sent_len = max_sen_length

        self.div_term = torch.tensor(torch.exp(-2*i/self.emb_dim * torch.log(10000)) for i in torch.arange(0,self.emb_dim

        self.position = torch.arange(self.max_sent_len)

        self.pe = torch.zeros(self.max_sent_len,self.emb_dim)

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

    def forward(self,encoded_seq):

        x =  encoded_seq + self.pe[:,:x.size(1)]
        return self.dropout(x)

```

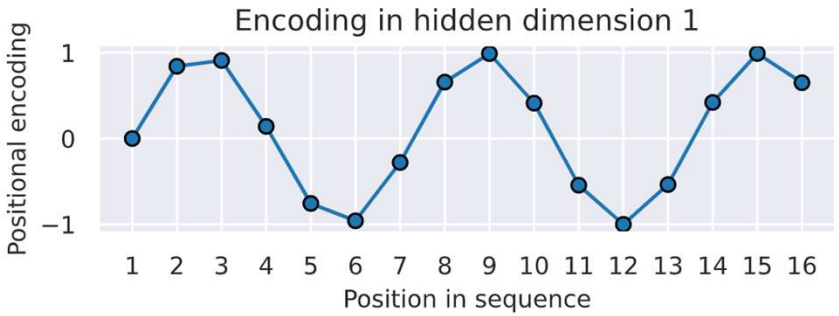
Each position encoding vector is generated as a vector of values of a sin or cos function

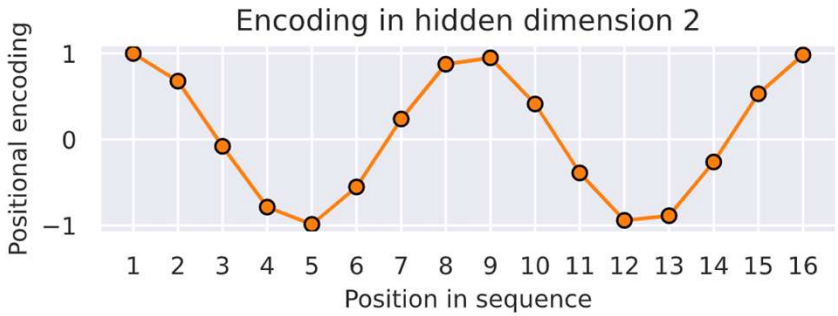
$$p_k(j) = \begin{cases} \sin(k \exp(-2j/emb\_dim \log(10000)) \text{ if } k \text{ is even} \\ \cos(k \exp(-2j/emb\_dim \log(10000)) \text{ if } k \text{ is odd} \end{cases}$$

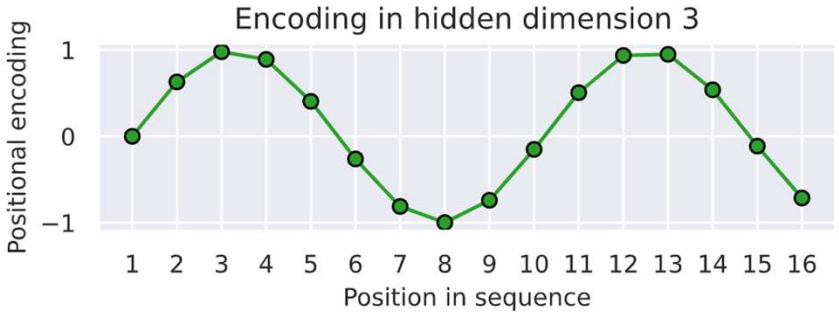
Here k is the position in the sentence and j is the coordinate in the k'th position encoding vector

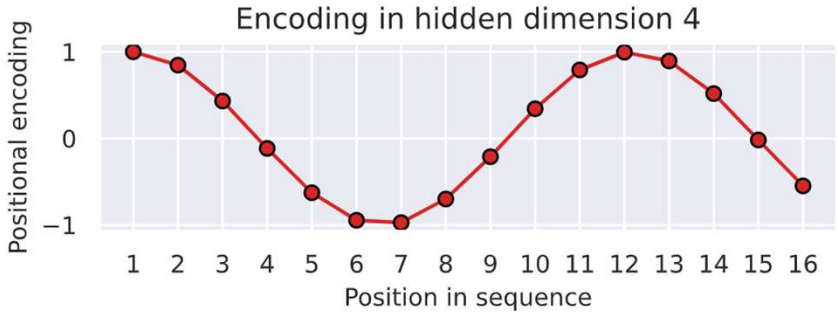
```
p_5 = [np.cos(- 5 * np.exp((-2*j/emb_dim)*np.log(10000))) for j in range(emb_dim)]
p_5
```

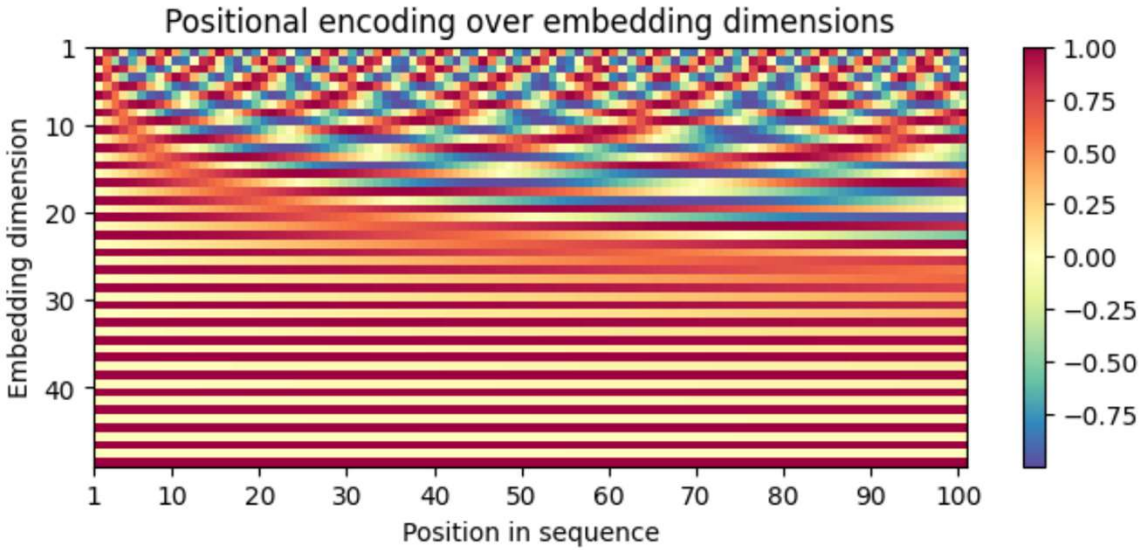
```
[0.2836218546322625,
 0.47378072050724973,
 0.9731902242785206,
 0.9987502603949663,
 0.9999419807006283,
 0.9999973069578462,
 0.9999998750000026,
 0.999999994198014,
 0.9999999997306956,
 0.9999999999875,
 0.9999999999994198,
 0.999999999999973]
```


Encoding in hidden dimension 1
Positional encoding
1
0
−1
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Position in sequence


Encoding in hidden dimension 2
Positional encoding
1
0
−1
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Position in sequence


Encoding in hidden dimension 3
Positional encoding
1
0
−1
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Position in sequence

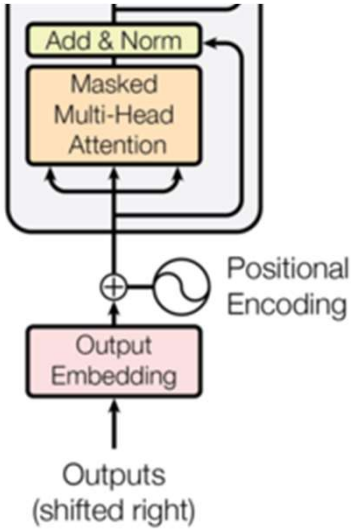

Encoding in hidden dimension 4
Positional encoding
1
0
−1
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Position in sequence


Positional encoding over embedding dimensions
Embedding dimension
1
10
20
30
40
Position in sequence
1
10
20
30
40
50
60
70
80
90
100
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75

The *Decoder* basically has the same structure as the *Encoder* though with some important differences.

The Decoder essentially is trying to compute the next word in the translated sentence given the previous words.

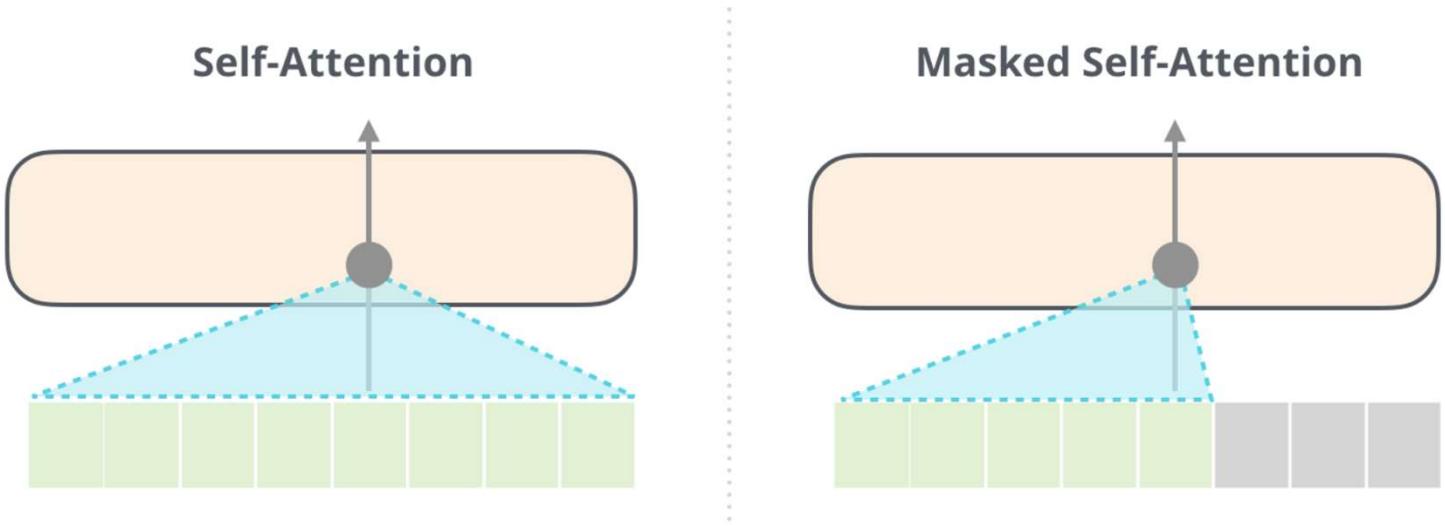
This means that we can't use information about the next word and in this case it means we can't use attentions with the next word. We are using a *mask* to 0 out the attentions. We also insert a `<bos>` token as the first word so the actual translated sentences are shifted one position to the right



Masking

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 to generate sentences by sequentially guessing 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.


Self-Attention
Masked Self-Attention

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

$$\sum_k w_{ik} v_k.$$

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

Mask

1	0	0	0
1	1	0	0
1	1	1	0
1	1	1	1

$Q * K^T / \sqrt{d}$

u_{11}	u_{12}	u_{13}	u_{14}
u_{21}	u_{22}	u_{23}	u_{24}
u_{31}	u_{32}	u_{33}	u_{43}
u_{41}	u_{42}	u_{43}	u_{44}

Apply Mask →

u_{11}	$-\infty$	$-\infty$	$-\infty$
u_{21}	u_{22}	$-\infty$	$-\infty$
u_{31}	u_{32}	u_{33}	$-\infty$
u_{41}	u_{42}	u_{43}	u_{44}

Softmax →

W

1	0	0	0	sum $= 1$
w_{21}	w_{22}	0	0	sum $= 1$
w_{31}	w_{32}	w_{33}	0	sum $= 1$
w_{41}	w_{42}	w_{43}	w_{44}	sum $= 1$

V

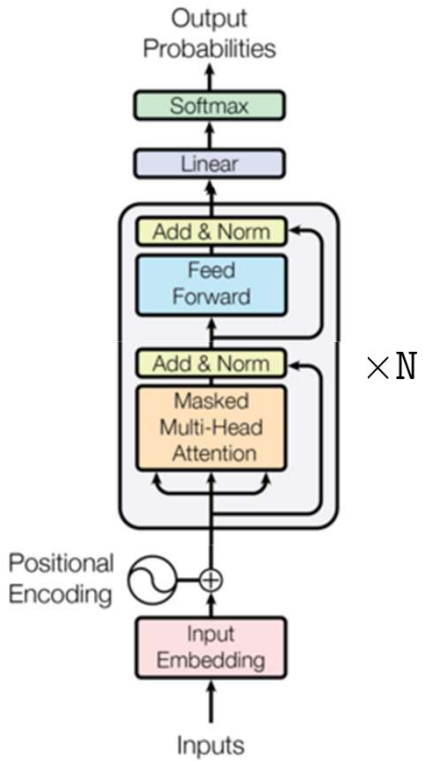
v_{11}	v_{12}	v_{13}
v_{21}	v_{22}	v_{23}
v_{31}	v_{32}	v_{33}
v_{41}	v_{42}	v_{43}

$W * V$

v_{11}	v_{12}	v_{13}
$w_{21}v_{11} + w_{22}v_{21}$	$w_{21}v_{12} + w_{22}v_{22}$	$w_{21}v_{13} + w_{22}v_{23}$
$w_{31}v_{11} + w_{32}v_{21} + w_{33}v_{31}$	$w_{31}v_{12} + w_{32}v_{22} + w_{33}v_{32}$	$w_{31}v_{13} + w_{32}v_{23} + w_{33}v_{33}$
$\cdots$	$\cdots$	$\cdots$

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

Masked Decoder Network



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.

GPT-2