Department of Computing

Imperial College of Science, Technology and Medicine

Transformers

Author: Anton Zhitomirsky

Contents

1 Transformers **2**

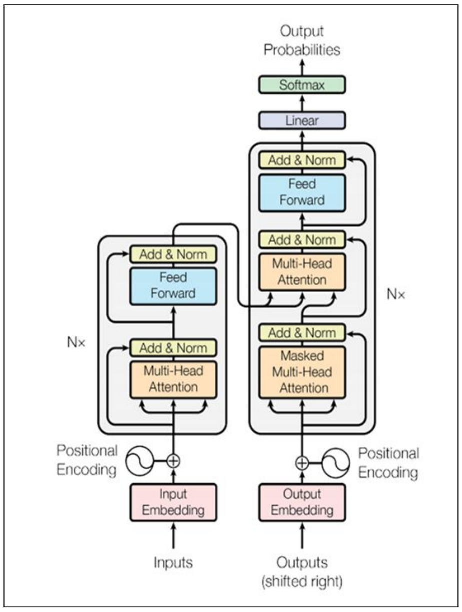
2 Transformer — Encoder **3**
2.1 Self-attention 4
2.1.1 Vector form 4
2.1.2 Matrix Form 7
2.2 Multi-head attention 8
2.3 Normalization 8
2.3.1 Layer normalization 8
2.3.2 Residual Connections 9
2.4 Position-wise Feed forward Network 9
2.5 Positional Encoding 10
2.5.1 Pseudo-code 11

3 Transformer — Decoder **11**
3.1 Pipeline 12
3.1.1 Testing 12
3.1.2 Training 12
3.2 Masked Multi-head Self attention 13
3.3 Cross Attention 13
3.4 Conclusion 13

4 Tricks **13**
4.1 Decaying learning rate 13

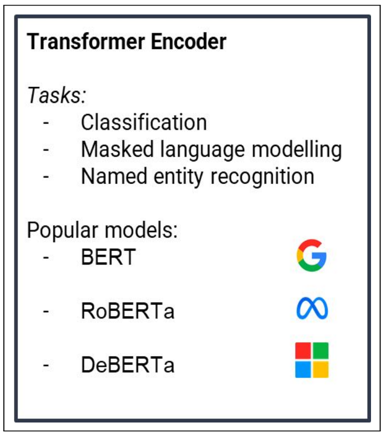
5 Code **13**
5.1 Self Attention 13
5.2 MHA 15

1 Transformers

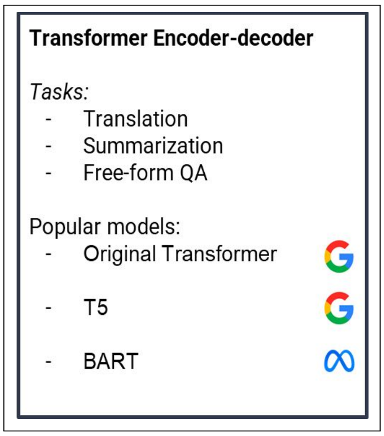


- There are n amounts of encoders and n amounts of decoders.
- Within one layer (encoder or decoders) we have multiple sub-layers
- In the encoder, we have 2 sub-layers, which processes muilti-head attention and the add and norm, the second sub-layer has a feed forward network and add & norm.
- In the decoder we have 3 sublayers, which is the masked multi-head attention, sublayer 2 (which is also known as cross attention) and sublayer three which is a feed-forward network.

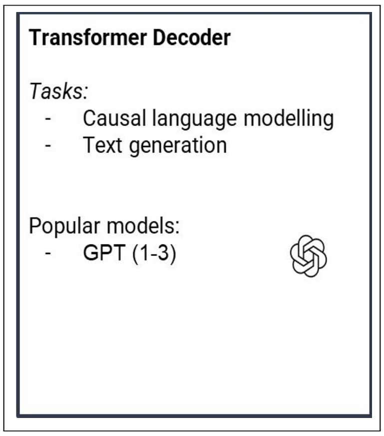
The original paper utilizes this as an encoder decoder network.



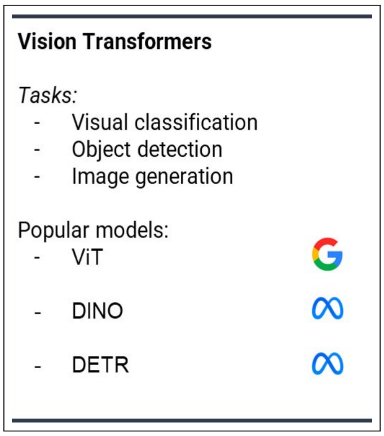
(a) Classificaiton tasks, feeding in a sequence and get a prediction over full input (sentiment classification) or a subset (entity recognition).



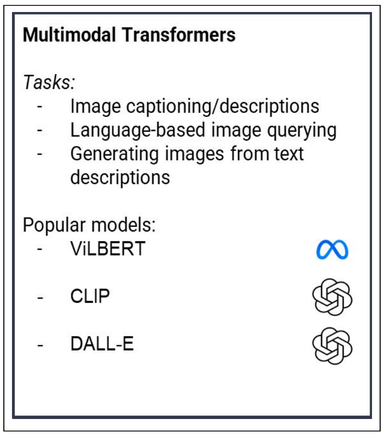
(b) Sequence to sequence tasks



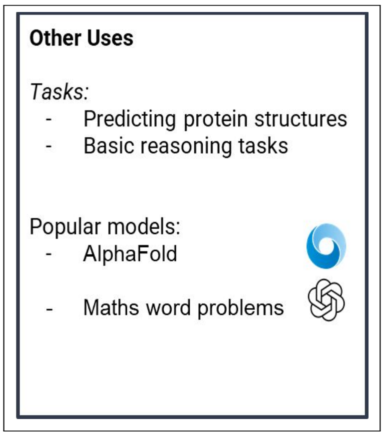
(c) For language specific task such as text generation or language modelling



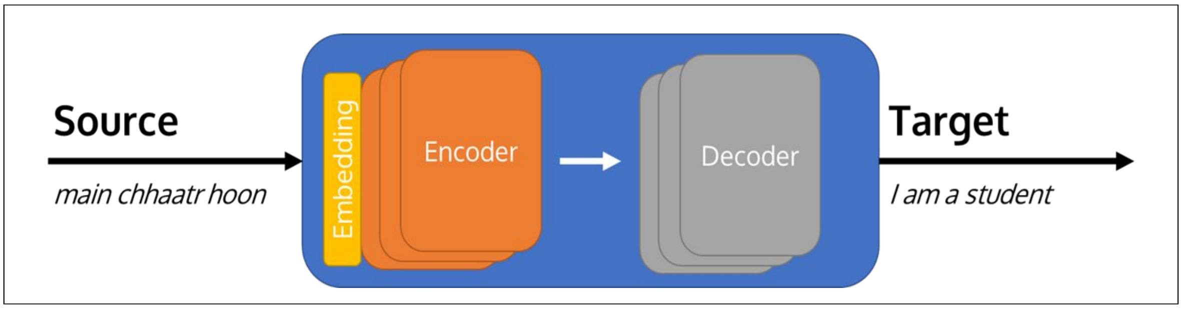
(d) Use in place of CNNs, object detection or generation



(e) Incorparating more than one modality (e.g. text, image, sound). So, generating images from text, or image captioning from an image.

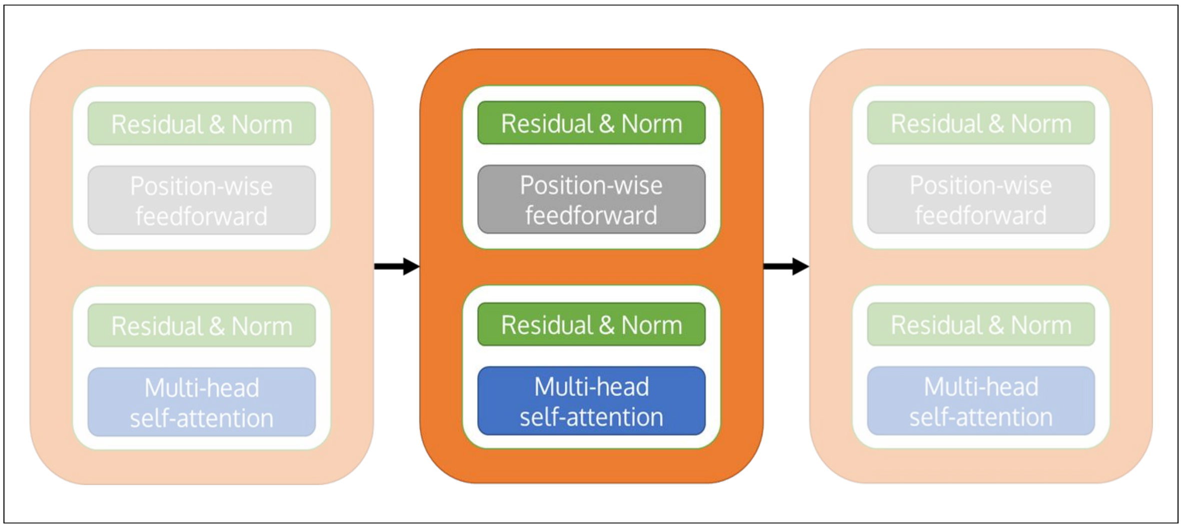


(f)

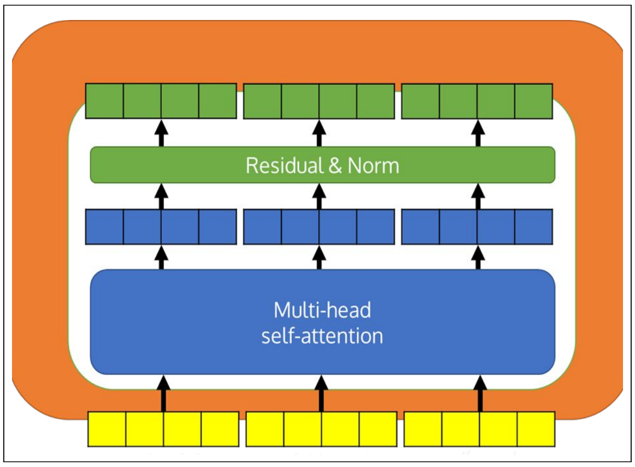


Like with other translation, the Transformer takes in a source sentence and generates a target. Inside, the trasnformer, we have a stack of encoder layers and a stack of decoder layers. At some point, we use the encoded output to help us generate the target. Note that the output of one encoder layer is sent as input to the next encoder layer. Similarly, the output of one decoder layer is sent to the next decoder layer as input

2 Transformer — Encoder



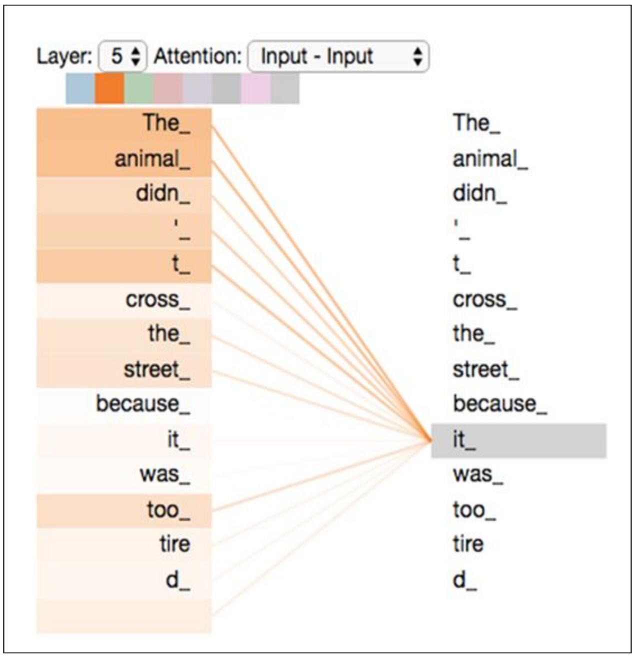
The output of the first encoder serves as the input for the second encoder. There are two sublayers. The first sublayer contains a multi-head self-attention module, and the second contains a position-wise feed forward network. A common theme in Transformers is that each sublayer will have a residual connection and layer normalization



- The multi-head self-attention receives some input, the yellow, which are some word representations (here, with 4 dimensions each). Generally, we have S words, each with D dimensions.
- The multi-head self-attention layer processes the input and outputs another set of $S \times D$ encodings.
- These ecnodings get sent to the Residual & Norm, which outputs another set of $S \times D$ encodings
- Conceptually this should make transformers easy to work with -mostly everything in the encoder stays as $S \times D$.

2.1 Self-attention

The self-attention mechanism allows each input in a sequence to look at the whole sequnce to compute a representation of the sequence. Each word therefore gets a representation based on all the other words in the input sequence. Each S in an $S \times D$ input has looked at every other word to compute its D-dimensional representation.

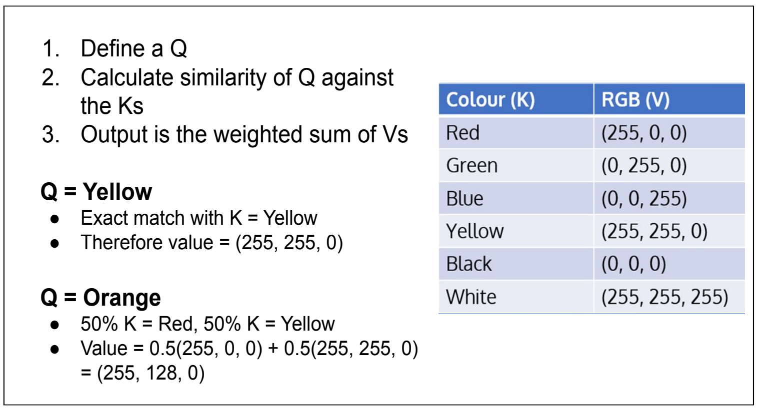


- As an example: “the animal didn’t cross the street because it was too tired’
- If we were processing this sequence, we would have some hidden state representation for the word “it”.
- In an RNN, the word “it” hasn’t been explicitly forced to look at other words in the sequence to align itself within them; the importance of words would be equal where we don’t want them to be
- Transformers are designed such that the hidden state representation of the word “it” would be influenced more by “animal” than “because”.
- For self-attention, each input (e.g. “it”) looks at the whole sequence and then computes a representation of that word, based on how important each of the other words are to it.s

The attention function consists of 3 variables

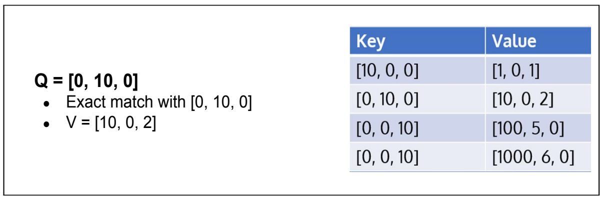
$$\text{Attention}(Q, K, V) = \sigma(\frac{QK^T}{\sqrt{d_h}})V, \quad \textbf{Q}\text{eries}, \textbf{K}\text{eys}, \textbf{V}\text{alues}, \quad \sigma : \text{softmax} \tag{2.1.1}$$

2.1.1 Vector form



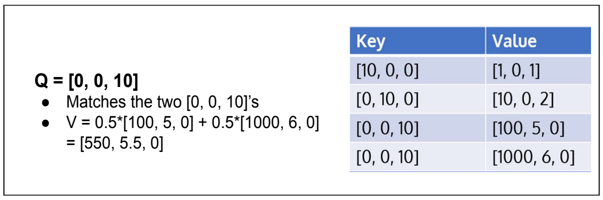
Think of it as a look-up in a dictionary (color to RGB value)

1. The query is the colour we want to find.
2. We would index the dictionary and receive a direct match
3. If the query is “orange” we want 50% red and 50% yellow and combine them



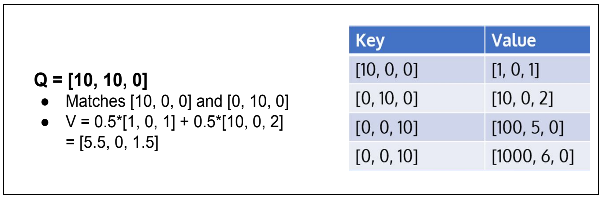
Key	Value
[10, 0, 0]	[1, 0, 1]
[0, 10, 0]	[10, 0, 2]
[0, 0, 10]	[100, 5, 0]
[0, 0, 10]	[1000, 6, 0]

(g) direct match



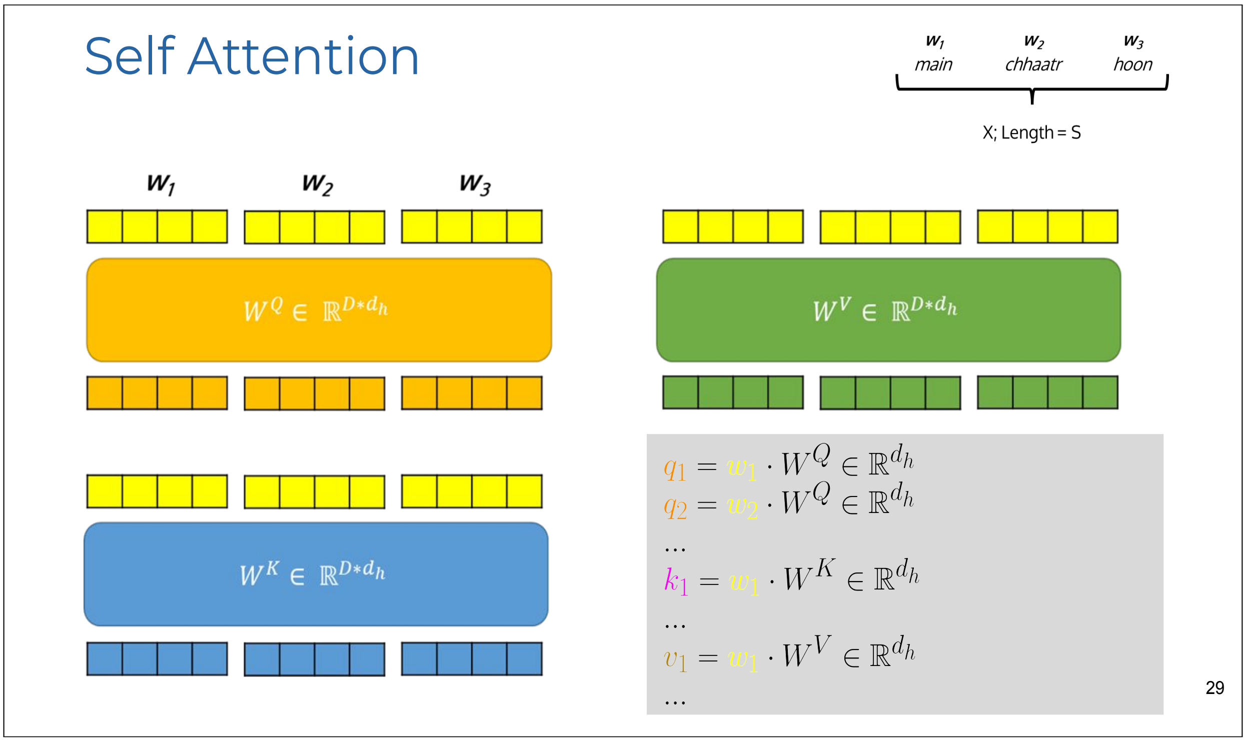
Key	Value
[10, 0, 0]	[1, 0, 1]
[0, 10, 0]	[10, 0, 2]
[0, 0, 10]	[100, 5, 0]
[0, 0, 10]	[1000, 6, 0]

(h) matches two keys



Key	Value
[10, 0, 0]	[1, 0, 1]
[0, 10, 0]	[10, 0, 2]
[0, 0, 10]	[100, 5, 0]
[0, 0, 10]	[1000, 6, 0]

(i) matches two keys, but in a combination



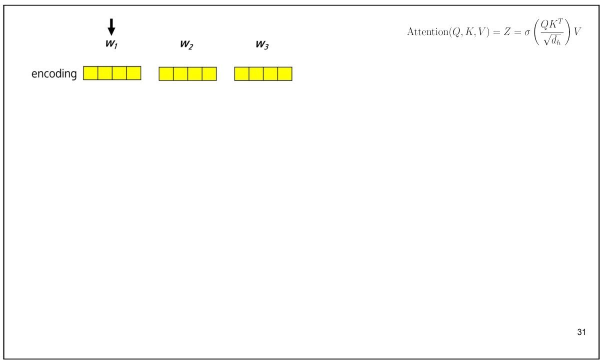
First, we take our word embeddings and project them through a learnt weight matrix, to a representation $D \times d_h$. Firstly, we set d_h equal to D. We do the same for our keys and values.

We can simplify the above to do everything in matrix form:

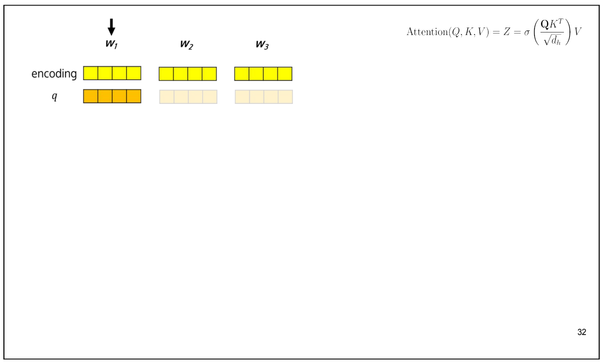
$$Q = W \cdot W^Q \in \mathbb{R}^{S \times d_h} \tag{2.1.2}$$

$$K = W \cdot W^K \in \mathbb{R}^{S \times d_h} \tag{2.1.3}$$

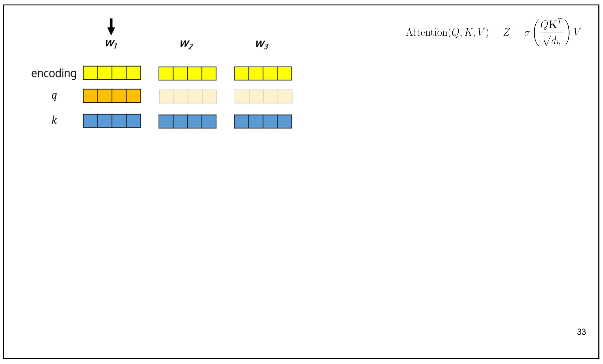
$$V = W \cdot W^V \in \mathbb{R}^{S \times d_h} \tag{2.1.4}$$



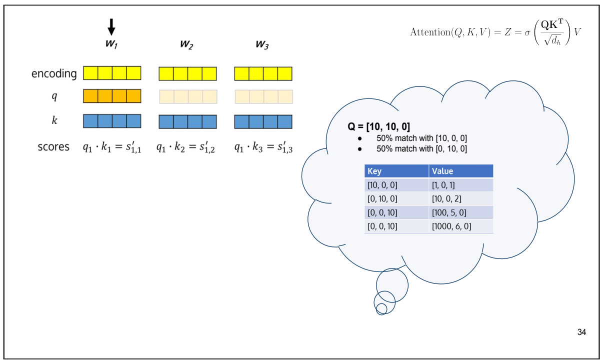
(j) Looking at attention in vector form first, we're going to work out our attention-ized representation for each word. Let's start with w_1



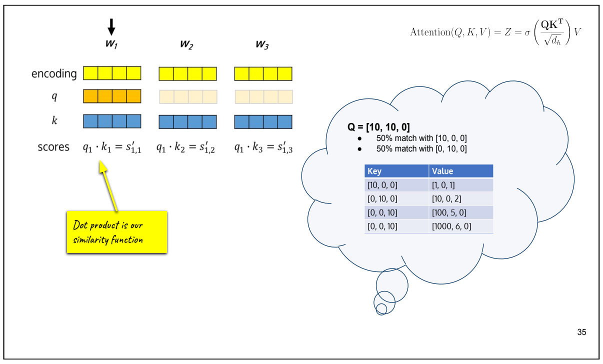
(k) Due to the projections above, we have query vectors for all words. Right now, we only need the query vector for w_1



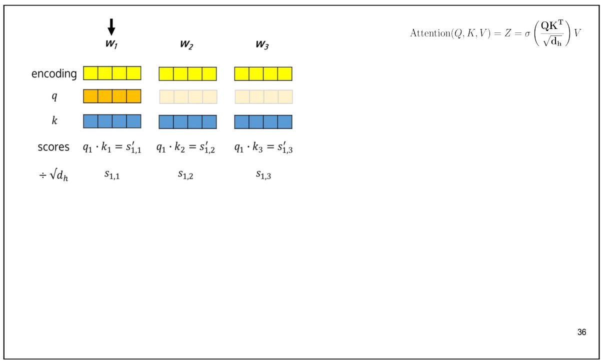
(l) Thinking back to the intuition, we want to compare how similar the query is to all keys we have. This is done by some similarity function. So here, we want to get a representation for w1. We will get this representation by comparing our query vector to all our keys.



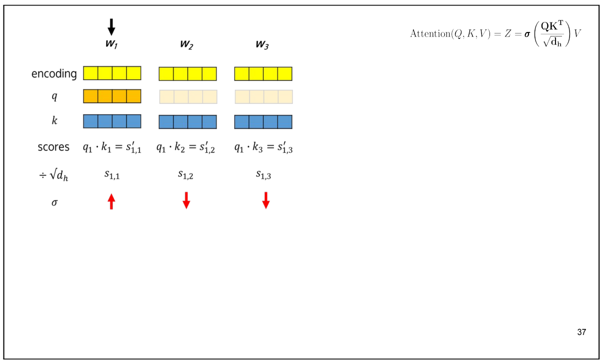
(m) Now we're going to work out the similarity between the query vector we have, and ALL the key vectors. We then obtained a weighting for how similar each of the key vectors were to the query vector. The score here is NOT the weighting, whoeve,r we need it in our calculation of the weighting



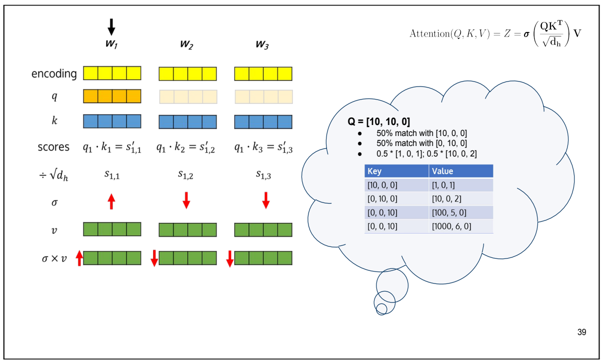
(n) Lets say the similarity is the dot product, so we get an un-normalized similarity between the given query vector and key vectors.



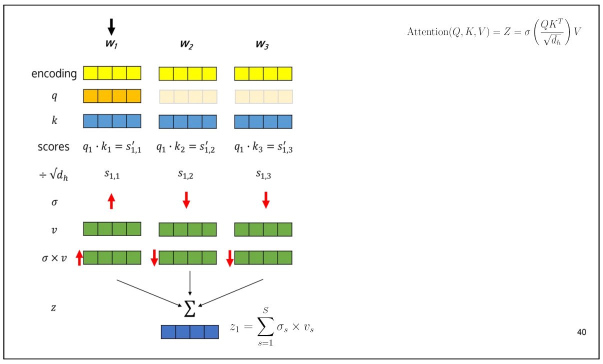
(o) In the next step, apply softmax, but before we do this, divide by $\sqrt{d_h}$. Post-division values are now closer to 0. This makes the softmax operation less peaky. That means that the outputs of the softmax operation will ideally not have an individual value which dominates the weightings



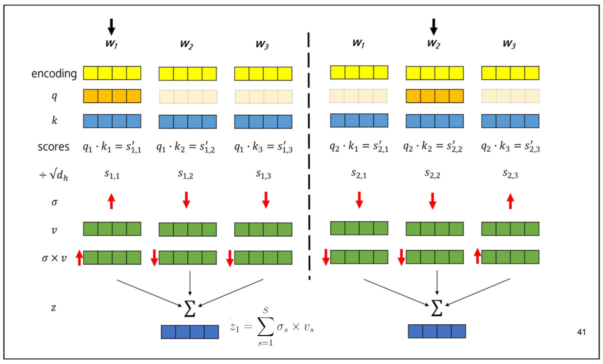
(p) Arrows indicate strong and weak weightings.



(q) The first step is simply retrieving the value vectors we obtained previously. The second step is actually performing the weighting on each of our value vectors. So our value vectors retain more information if the softmax value for that index is high, and retain less information if the softmax value for that index is low. The softmax here is the way we normalize similarity scores



(r) Finally we obtain our contextual representation for the first word. This is the sum over our weighted value vectors.

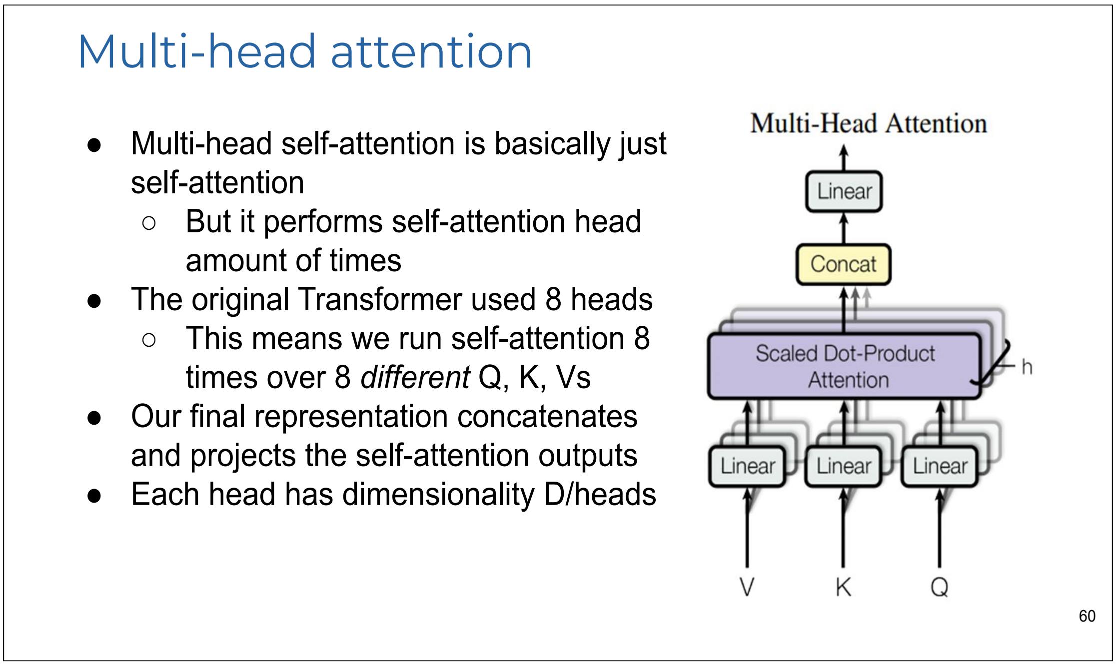


(s) We repeat the process for every word in our input sequence. Our overall representation for our input sequence would then be all of our z vectors stsacked togheter. Each z-vector is D-dimensional, and obviously we have S amounts of them.

2.1.2 Matrix Form

$$\text{Attention}(Q,K,V) = Z = \underbrace{\sigma\Big(\frac{\overbrace{Q}^{\in\mathbb{R}^{S\times d_h}}\ \overbrace{K^T}^{\in\mathbb{R}^{d_h\times S}}}{\sqrt{d_h}}\Big)}_{\text{attention matrix}\in\mathbb{R}^{S\times S}} V, \quad S = \text{source sequence length}$$

2.2 Multi-head attention



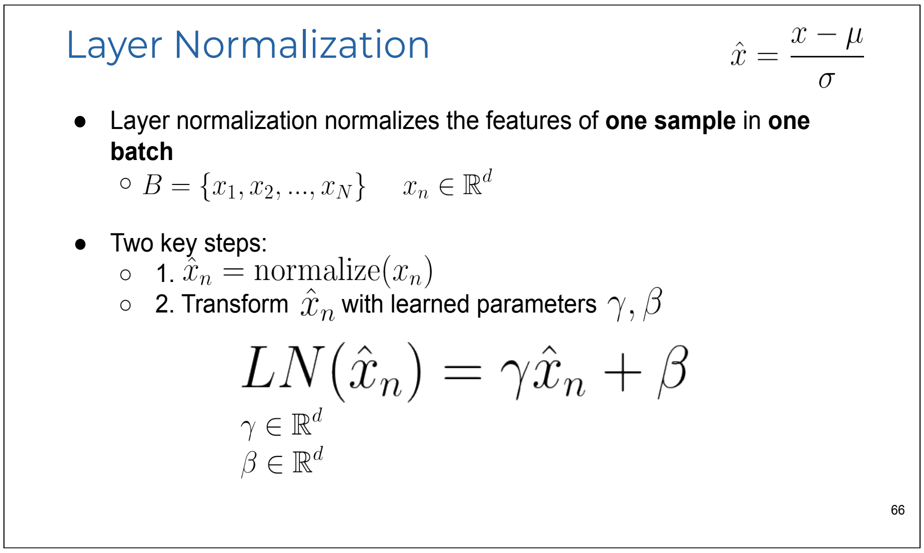
We perform self-attention *head* amount of times. Each head may refer to different parts of the sequence. The number of heads needs to be equally divisible by the model dimensionality

d_h therefore represents the model dimensionality divided by the number of heads.

2.3 Normalization

2.3.1 Layer normalization

Data fed into a neural network should be normalized e.g. $\hat{x} = \frac{x-\mu}{\sigma}$. There are also normalization techniques within the activations or layers of a network



It normalizes the feature of one sample in one batch. The first step is that we normalize each one of the items in our batch respective to that item itself. Then transform the variable with learnable parameters (γ, β)

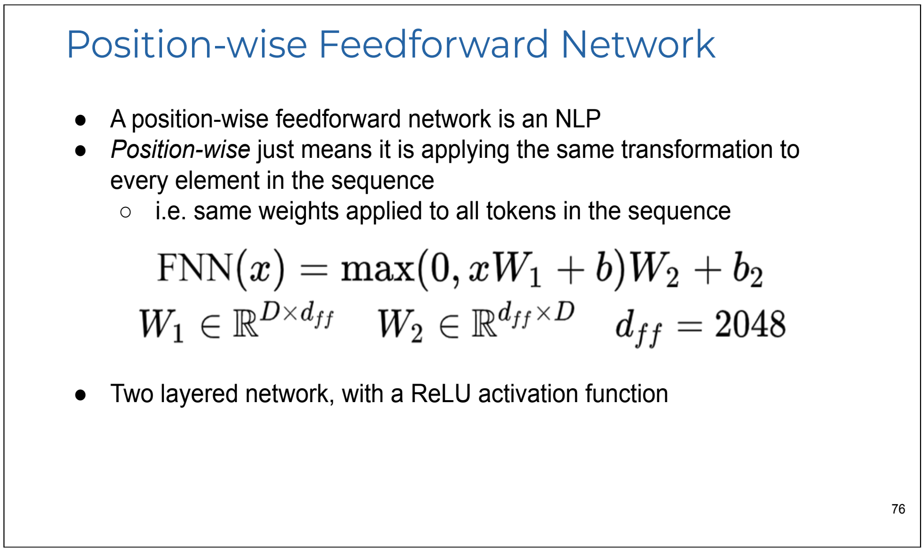
2.3.2 Residual Connections

You would assume that if you want to stack layers, we should get a lower training error. In practice this doesn't happen; as you increase the number of layers, what you see is an increase in the training error that you will see.

Residual connections help mitigate the vanishing gradient problem, where Vanishing gradients = tiny weight changes. Residual connections provide a shortcut for information to flow to layer layers of the network, where the output of an earlier layer is added directly to the output of a layer layer. (see ResNet in Deep Learning lectures).

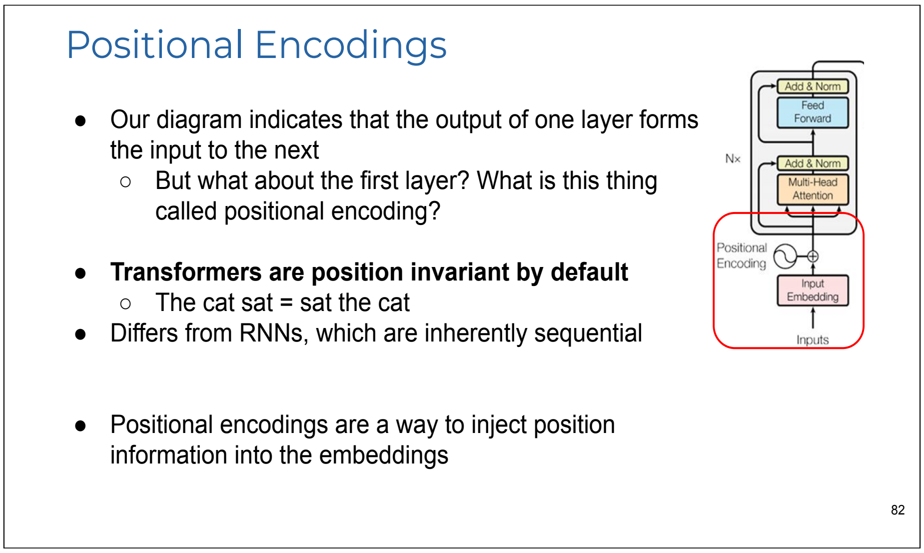
The reason this works so well: it lets each weight matrix to focus on only learning the difference between the previous layer and the current layer instead of learning an entire transformation as what happens in the traditional case.

2.4 Position-wise Feed forward Network

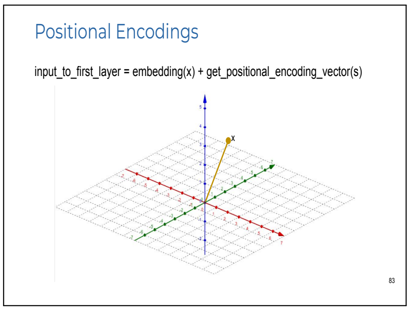


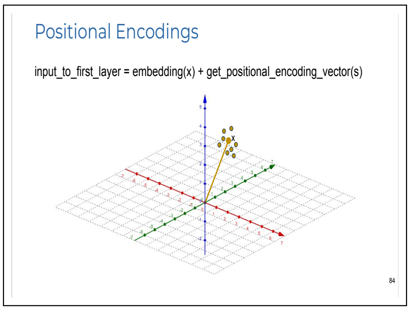
network is an NLP $\rightarrow$ **network is an MLP**. Transformation is applied to all weights in a sequence. In the above example, there are 2 layers, with a ReLU activation function. We project x from d_{ff} into D. Then W_2 projects it down again.

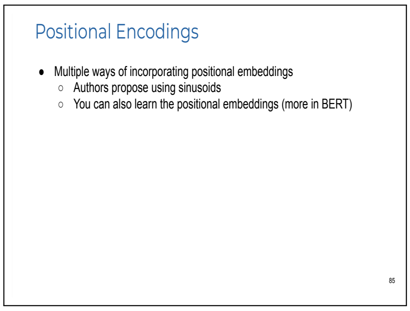
2.5 Positional Encoding



This is inherently different to RNN which processes things in the order the arrive in.







The positional encoding vector is independent of the word, but only to the position in the sequence. The addition, permute the vector by small amounts. In vector space, the offset remains the same for any embedding given the same position.

There are multiple ways of incorporating positional embeddings, such as sinusoids or learning the positional embeddings. Applying positional encoding depends ONLY on the index the word appears in.

$$PE_{pos,2i} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

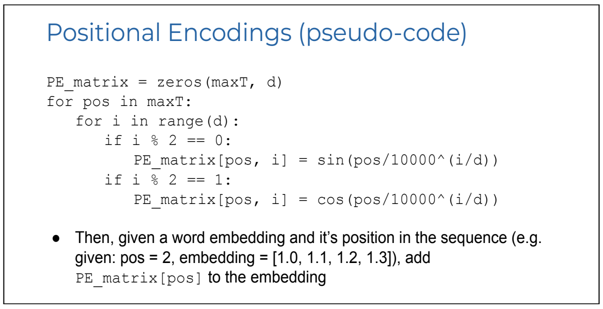
$$PE_{pos,2i+1} = \cos\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

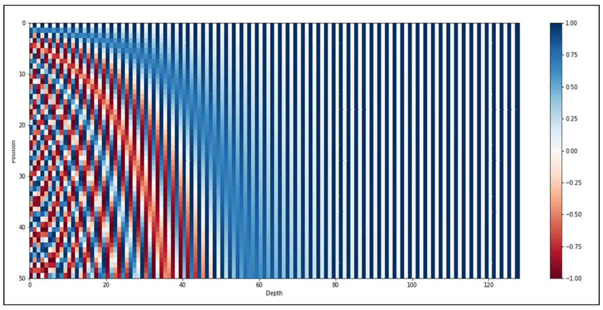
pos : position of the word, i : index of d

The $2i$ and $2i+1$ impl that we're going to be looping over the indexes in range of d. E.g. consider we had a vector of 512 dimension. We would be looping over it. At an even index (e..g 0) we'd apply the $\sin$ variant of PE. At an odd index (e.g. 1), we would apply the $\cos$ variant.

The only difference between the functions is whether we use a sin or cosine. Note that when we're early on in looping over d, i will be small. The resulting exponent would be therefore be small. This makes the denominator small. Thus the overall value of $\frac{pos}{10000^{(2i/d)}}$ would be larger than when we're later in our loop over d

2.5.1 Pseudo-code

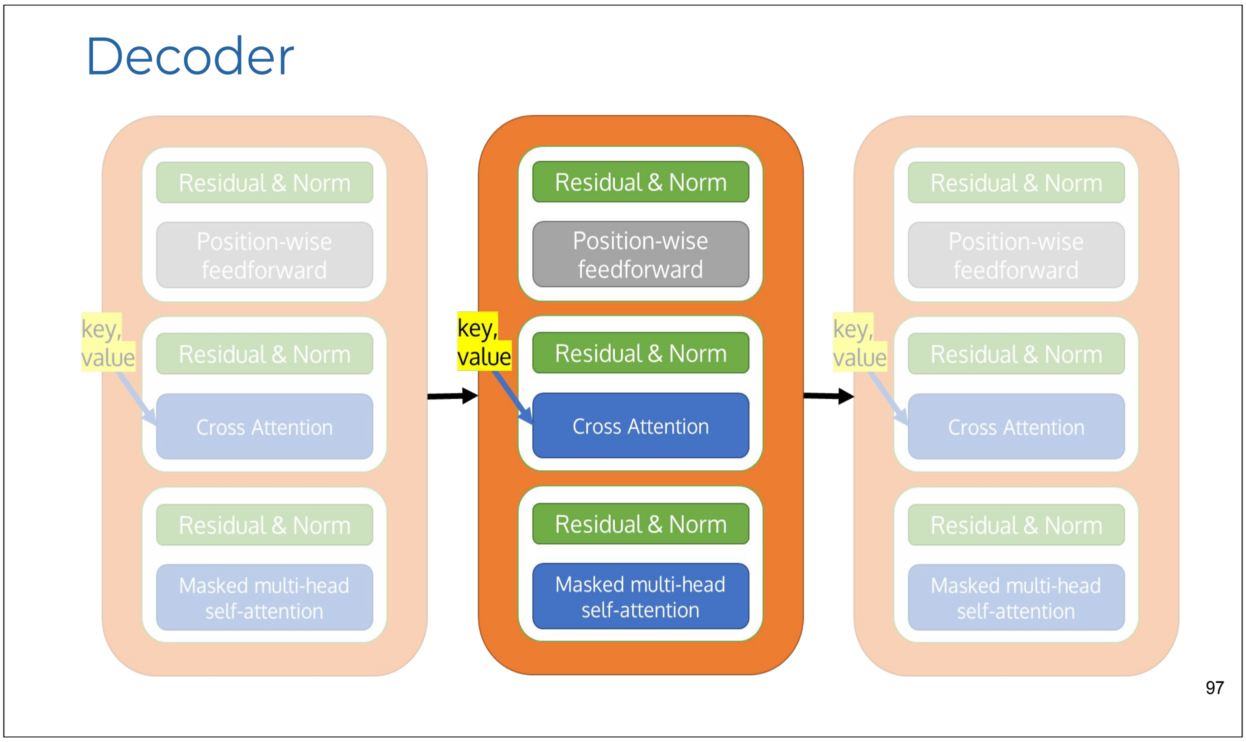




Consider position 0, i.e. the first word in the sequence, positional encoding function means no manipulation is done to the vector. At position 1, the positional encoding function shows that indexes up to 10 have values close to 1 added to them. At position 22 the encoding function shows that the first few indexes are affected heavily by positional encoding. some indexes have values close to -1 added to them, followed shortly by a value +1 added to them.

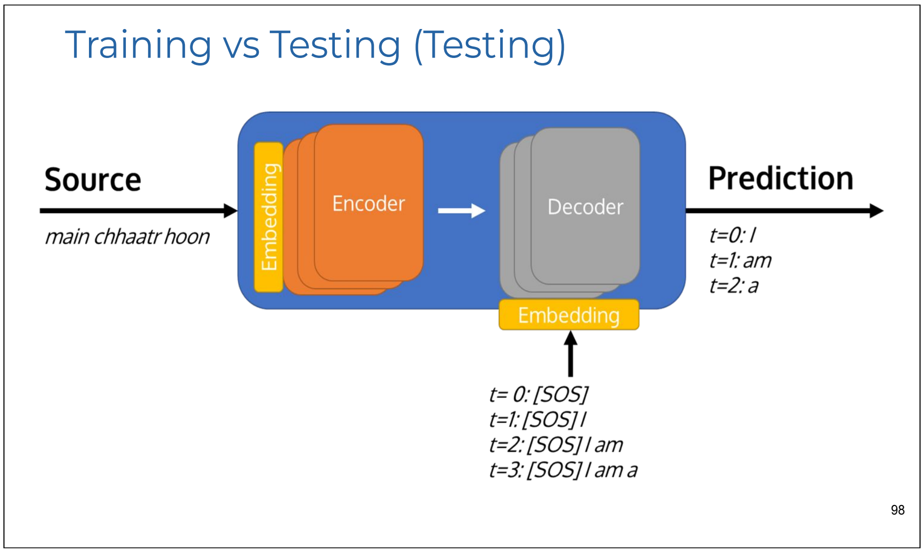
We don't just add another dimension to indicate the position because that draws the samples away form the 0 mean. Also, there is less generability because if during training, the model hasn't seen that word in that position then during testing it won't be very good.

3 Transformer — Decoder



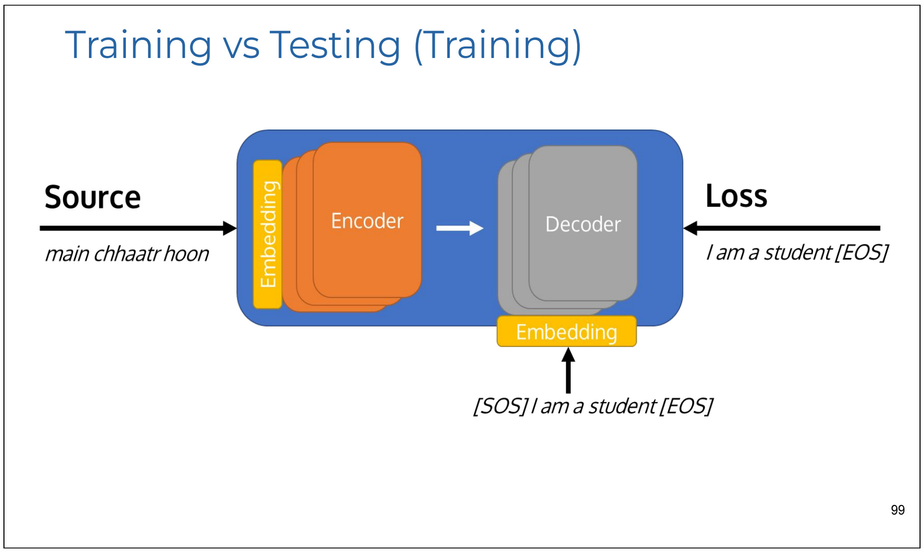
3.1 Pipeline

3.1.1 Testing



Our source sentence get encoded via our encoder. We feed an SOS token to our decoder, which then predicts the first word. We append the prediction to our SOS token, and then use this to predict the next word. At some point our decoder will predict na EOS token and we'll know that this is the end of the generation

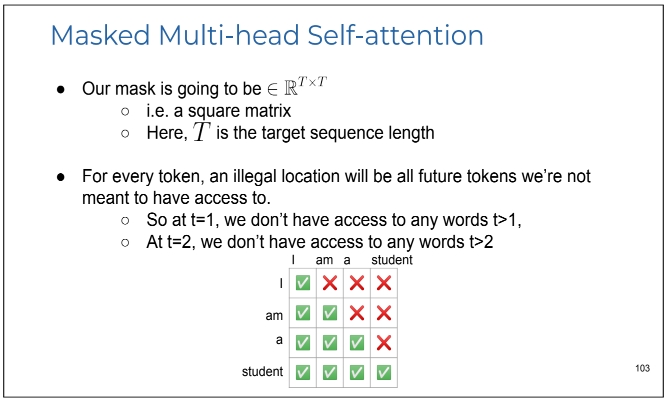
3.1.2 Training



However, during training, we won't get the model to generate auto-regressively. We're going to feed the whole target sequence as input to the decoder and decode the sentence in one go. This means we cna run the decoder stack once. As opposed to running it for as many tokens as we have in the target sequence (Which is what we do when performing inference). If we feed in the whole sequence ot the decoder, we need a way to tell the decoder not to look at future tokens when computing attention for one of the tokens. E.g. when trying to compute teh attention-based representation for "am" we should not be allowed to look at "a" or "student", since these tokens are in the future. We enforce this uni-directionality with masking.

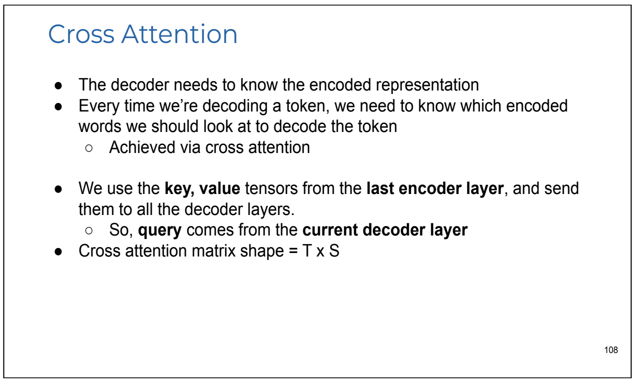
3.2 Masked Multi-head Self attention

When we looked at attention in the encoder, the bidirectionality meant that each word looked at every other word, including future tokens for the word which we are trying to calculate the attention for. Masked multi-head means that we're not looking at future tokens.



We enforce uni-directionality with a mask. We set the values in the upper triangle to negative infinity. The mask is a $T \times T$ matrix.

3.3 Cross Attention



Thinking back to the dictionary analogy, we have a query vector (in target-language embedding space). We're then looking to find the similar key vectors from the encoder. In other words, which encoded words are most similar to this current decoding word. Then, using the softmax weighted similarities, we obtain some aggregated value indicating which words are most important.
$T \times S$ is the similarity between target words and source words.

3.4 Conclusion

Our overall decoder is similar to the encoder layer in construction. We create a decoder and decoder layer class, the decoder layer class contains one decoder layer, and the decoder class contains embeddings + PE and a for loop over the desired number of decoder layers.

4 Tricks

4.1 Decaying learning rate

$$lr = \sqrt{\frac{1}{d}} \times \min\left(\sqrt{\frac{1}{d}}, i \times warmup^{-1.5}\right), \quad \begin{cases} i: & \text{current global step} \\ warmup: & \text{hyperparameter (default: 4000)} \\ d: & \text{model dimensionality} \end{cases} \tag{4.1.1}$$

5 Code

5.1 Self Attention

```python
# %%
import torch
import torch.nn.functional as F
import math
import numpy as np

# %%


def scaled_dot_product_attention(Q, K, V, dk=4):
    ## matmul Q and K
    QK = Q @ K.T

    ## scale QK by the sqrt of dk
    matmul_scaled = QK / math.sqrt(dk)

    attention_weights = F.softmax(matmul_scaled, dim=-1)

    ## matmul attention_weights by V
    output = attention_weights @ V

    return output, attention_weights

# %%


def print_attention(Q, K, V, n_digits=3):
    temp_out, temp_attn = scaled_dot_product_attention(Q, K, V)
    temp_out, temp_attn = temp_out.numpy(), temp_attn.numpy()
    print('Attention weights are:')
    print(np.round(temp_attn, n_digits))
    print()
    print('Output is:')
    print(np.around(temp_out, n_digits))


# %%
temp_k = torch.Tensor([[10, 0, 0],
                      [0, 10, 0],
                      [0, 0, 10],
                      [0, 0, 10]])  # (4, 3)

temp_v = torch.Tensor([[1, 0, 1],
                      [10, 0, 2],
                      [100, 5, 0],
                      [1000, 6, 0]])  # (4, 3)

# %%
# This `query` aligns with the second `key`,
# so the second `value` is returned.
temp_q = torch.Tensor([[0, 10, 0]])  # (1, 3)
print_attention(temp_q, temp_k, temp_v)

# %%
# This query aligns with a repeated key (third and fourth),
# so all associated values get averaged.
temp_q = torch.Tensor([[0, 0, 10]])  # (1, 3)
print_attention(temp_q, temp_k, temp_v)

# %%
# This query aligns equally with the first and second key,
# so their values get averaged.
temp_q = torch.Tensor([[10, 10, 0]])  # (1, 3)
print_attention(temp_q, temp_k, temp_v)
```

```python

# %%
temp_q = torch.Tensor([[0, 10, 0], [0, 0, 10], [10, 10, 0]])  # (3, 3)
print_attention(temp_q, temp_k, temp_v)
```

code/self_attention.py

5.2 MHA

```python
# %%
import torch
import torch.nn as nn
import math as m
import torch.nn.functional as F

# %%


class MultiHeadAttention(nn.Module):
    def __init__(self, d_model=4, num_heads=2, dropout=0.3):
        super().__init__()

        # d_q, d_k, d_v
        self.d = d_model//num_heads

        self.d_model = d_model
        self.num_heads = num_heads

        self.dropout = nn.Dropout(dropout)


        self.linear_Qs = nn.ModuleList([nn.Linear(d_model, self.d)
                                        for _ in range(num_heads)])
        ## create a list of layers for K, and a list of layers for V
        self.linear_Ks = nn.ModuleList([nn.Linear(d_model, self.d)
                                        for _ in range(num_heads)])
        self.linear_Vs = nn.ModuleList([nn.Linear(d_model, self.d)
                                        for _ in range(num_heads)])

        self.mha_linear = nn.Linear(d_model, d_model)

    def scaled_dot_product_attention(self, Q, K, V):
        # shape(Q, K, V) = [B x seq_len x D/num_heads]

        # q => [b x seq_len x d_h]
        # k => [b x seq_len x d_h]
        # => [b x d_h x seq_len]
        # q matmul k => [b x seq_len x seq_len]

        Q_K_matmul = torch.matmul(Q, K.permute(0, 2, 1))
        scores = Q_K_matmul/m.sqrt(self.d)
        # shape(scores) = [B x seq_len x seq_len]

        attention_weights = F.softmax(scores, dim=-1)
        # shape(attention_weights) = [B x seq_len x seq_len]

        output = torch.matmul(attention_weights, V)
        # shape(output) = [B x seq_len x D/num_heads]

        return output, attention_weights

    def forward(self, x):
        # shape(x) = [B x seq_len x D]
```

```
        Q = [linear_Q(x) for linear_Q in self.linear_Qs]
        K = [linear_K(x) for linear_K in self.linear_Ks]
        V = [linear_V(x) for linear_V in self.linear_Vs]
        # shape(Q, K, V) = [B x seq_len x D/num_heads] * num_heads

        output_per_head = []
        attn_weights_per_head = []
        # shape(output_per_head) = [B x seq_len x D/num_heads] * num_heads
        # shape(attn_weights_per_head) = [B x seq_len x seq_len] * num_heads
        for Q_, K_, V_ in zip(Q, K, V):

            ## run scaled_dot_product_attention
            output, attn_weight = self.scaled_dot_product_attention(Q_, K_, V_)

            # shape(output) = [B x seq_len x D/num_heads]
            # shape(attn_weights_per_head) = [B x seq_len x seq_len]
            output_per_head.append(output)
            attn_weights_per_head.append(attn_weight)

        # example output_per_head = [
        #   [
        #       [0.00, 0.01],
        #       [0.10, 0.11],
        #       [0.20, 0.21]
        #   ], (tensor)
        #   [
        #       [1.00, 1.01],
        #       [1.10, 1.11],
        #       [1.20, 1.21]
        #   ], (tensor)
        #   [
        #       [2.00, 2.01],
        #       [2.10, 2.11],
        #       [2.20, 2.21]
        #
        #   ] (tensor)
        # ]

        # example output = [
        #   [ 0.00, 0.01, 1.00, 1.01, 2.00, 2.01 ],
        #   [ 0.10, 0.11, 1.10, 1.11, 2.10, 2.11 ]
        #   [ 0.20, 0.21, 1.20, 1.21, 2.20, 2.21]
        # ] # [3 x 6]

        # [3 x 3]
        # [
        # [0.0, 0.1, 0.2]
        # [1.0, 1.1, 1.2]
        # []
        # ]

        # [2 x 2 x 3 x 3]
        # [
        #   [
        #       [0.0, 0.1, 0.2],
        #       [1.0, 1.1, 1.2],
        #       [2.0, 2.1, 2.2]
        #   ],
        #   []
        #
        # ]

        output = torch.cat(output_per_head, -1)
        attn_weights = torch.stack(attn_weights_per_head).permute(1, 0, 2, 3)
```

```
        # shape(output) = [B x seq_len x D]
        # shape(attn_weights) = [B x num_heads x seq_len x seq_len]

        projection = self.dropout(self.mha_linear(output))

        return projection, attn_weights

# %%
```

code/mha.py