Lecture 16:

Deep Learning & Transformers

- Deep Learning Principles
- Transformer Overview
- Embedding and Encoding
- Attention!
- Encoders and Decoders
- More Playtime with PyTorch…

Deep Learning

The term ‘Deep Learning’ is not precisely defined, but generally involves the use of an enormous number of learned parameters to solve problems, often starting from relatively ‘raw’ input data.

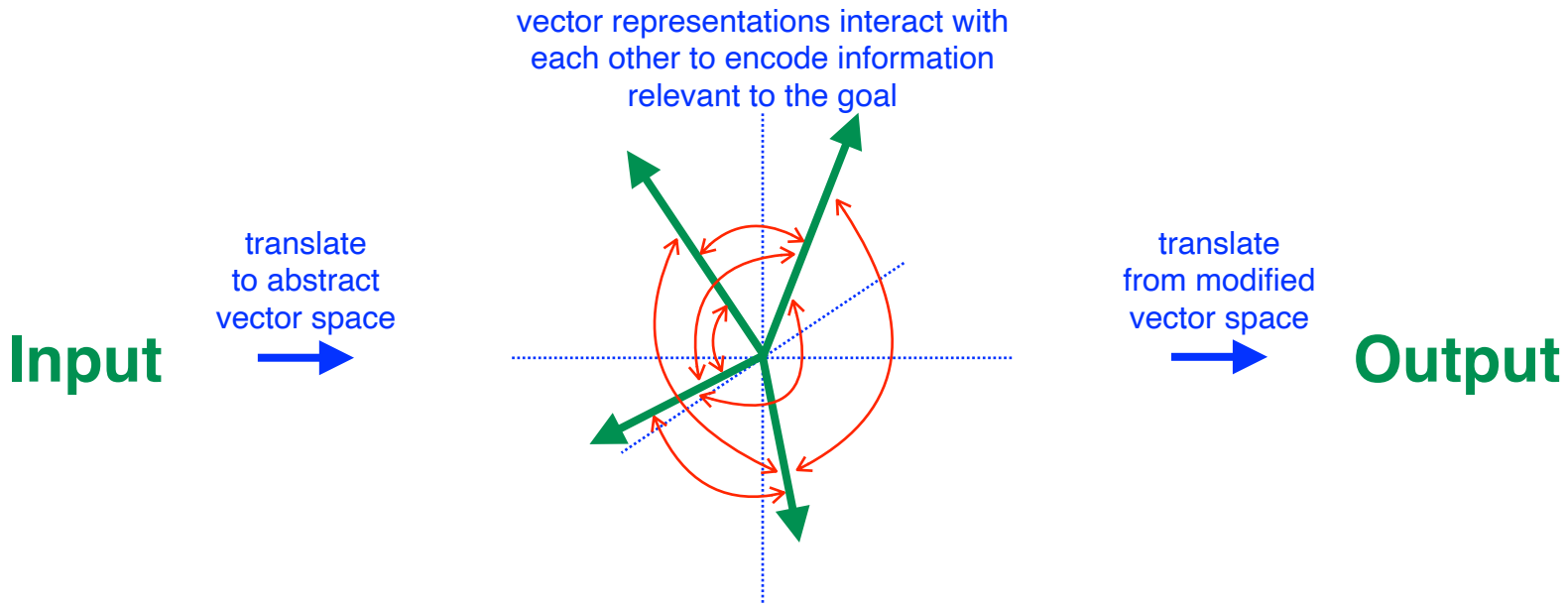
The trick to keep this from descending into total chaos is to **define a structure**, with notional logical units that have functional purposes, **around which the learning must take place.**

The actual functions of different parts of the network are then **emergent from the learning process**, driven entirely by the loss function that defines the goal through back-propagation.

It is, nonetheless, remarkable that just this is enough to prevent complete anarchy from millions of free parameters, and that simple back-propagation turns out to be so robust and scalable!

Transformers

Transformers* were initially invented for Language Models, but have multiple uses. They allow for non-sequential input of variable length and, thus, *do not have a fixed network architecture for interactions*. Instead, interaction between elements happens via the 'Attention Mechanism,' which evaluates associations and propagates their impacts. They are also very good at handling/processing large input strings.



The first step begins with the input, which could be a sentence or a recorded detector event. This is then broken up into separate items, such as words (or word fragments), or detector elements or individual pieces of information from an event. Each **discreet** item is mapped to a unique mathematical object (*e.g.* a vector) that acts as a 'token' for the item (we will talk about continuous parameter inputs shortly). This process is called ***Tokenization***

* "Attention is All You Need," Vashwani et al., arXiv:1706.03762

‘Big Picture’ Snapshot:

Here’s the basic idea behind the attention mechanism:

1) Produce a space in which to generically characterise individual tokens
2) Add further pertinent information relating to any particular token
3) Using this, produce another space for associations between tokens
4) Also produce a set of proposed modifications to token characterisations due to these associations
5) Implement modifications in accordance with the strengths of these associations

Ok, let’s dive in!

Embedding

Take each token to be represented by a vector in some large, d-dimensional space of characteristics. The exact nature of this space is something that will be learned by the network from examples, so we will just define a working area for this with a large dimensional vector for the "embedding" space.

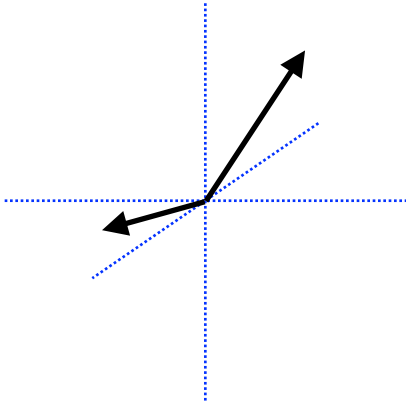
Once learned, these embeddings represent the basic descriptions for each token, independent of its use.

All relevant characteristics and operations will need to be couched in the language of this space!

We can form a list of embedding tokens, $\mathbf{E}$, by pasting together the individual vectors corresponding to the **vocabulary** of possible tokens (*e.g.* words or event characteristics) of size S

Any given input (*e.g.* a sentence or event) of length L can then be translated into an initial table of tokens, $\mathbf{T}$, (sometimes called the 'context window') drawn from this vocabulary

embedding 'characterisation space' for individual tokens



d-dimensional 'Working Space' for Embedded Characteristics

Token Label

	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}	...	C_d
1	15	50	-43	58	-13	-79	-96	22	73	-44	-51	48
2	20	-65	65	-67	-50	-61	-86	-71	68	80	-69	60
3	34	46	65	48	38	24	-39	92	98	-18	70	-69
4	73	-4	49	-26	-64	-63	54	-14	-33	-89	-51	98
5	-56	46	68	-93	4	-69	-66	-66	33	3	14	-83
6	-59	78	[illegible]									34
7	-80	47	[illegible]									62
8	24	-11	-12	59	-14	-13	48	-36	38	12	16	77
9	-36	-59	62	2	55	15	-19	7	11	87	-88	-36
10	-26	49	10	2	55	35	89	55	-59	-28	-61	-59
11	89	-18	-29	44	58	-43	-2	4	-81	-18	29	95
12	82	81	-9	-48	67	-61	-58	-54	-49	55	-63	50
13	67	11	-98	-95	-15	63	-70	-87	-4	38	99	-56
14	32	-96	-25	-44	-88	86	-80	-97	92	-69	-45	-64
15	4	16	-9	69	-54	83	19	68	-21	37	75	-62
...	-40	82	-18	-8	-22	62	-88	30	-64	-45	-33	-51
S	86	50	98	28	-23	-38	-37	-46	-84	50	-16	-64

learned weights

Embedded Characteristics

Token

	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}	...	C_d
4	73	-4	49	-26	-64	-63	54	-14	-33	-89	-51	98
11	89	-18	-29	44	58	-43	-2	4	-81	-18	29	95
5	-56	46	68	-93	4	-69	-66	-66	33	3	14	-83
5	-56	46	68	-93	4	-69	-66	-66	33	3	14	-83
8	24	-11	-12	59	-14	-13	48	-36	38	12	16	77
15	4	16	-9	69	-54	83	19	68	-21	37	75	-62
3	34	46	65	48	38	24	-39	92	98	-18	70	-69
8	24	-11	-12	59	-14	-13	48	-36	38	12	16	77
1	15	50	-43	58	-13	-79	-96	22	73	-44	-51	48

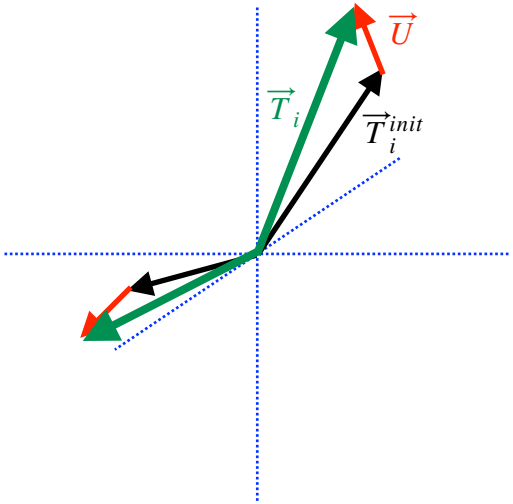
Encoding

We also want to account for additional information about the specific token, such as the order of a given word in a sentence, or the position of a given detector element, **or a continuous property** like time, temperature or energy that some sensor registered.

One way to do this is to define a modification vector to represent this information **in terms of the embedding space**, and then define a structure of

$$\overrightarrow{T}_i = \overrightarrow{T}_i^{init} + \overrightarrow{U}$$

where a given token vector is the sum of the initial vector and a modification vector, $\overrightarrow{U}$, containing the additional information.



The modifications can either be a fixed property of the token, or something that can vary from case to case. At the end of the day, the network will learn to interpolate from examples of the combined vector and, for a large enough embedding space, the chance of ambiguities arising between different initial vectors becomes vanishingly small.

The form of the modification vectors can be chosen at the start to convey useful structural properties (*e.g.* similarity, periodicity etc.), and the network will then 'learn around' these with the remaining freedom of vector definitions.

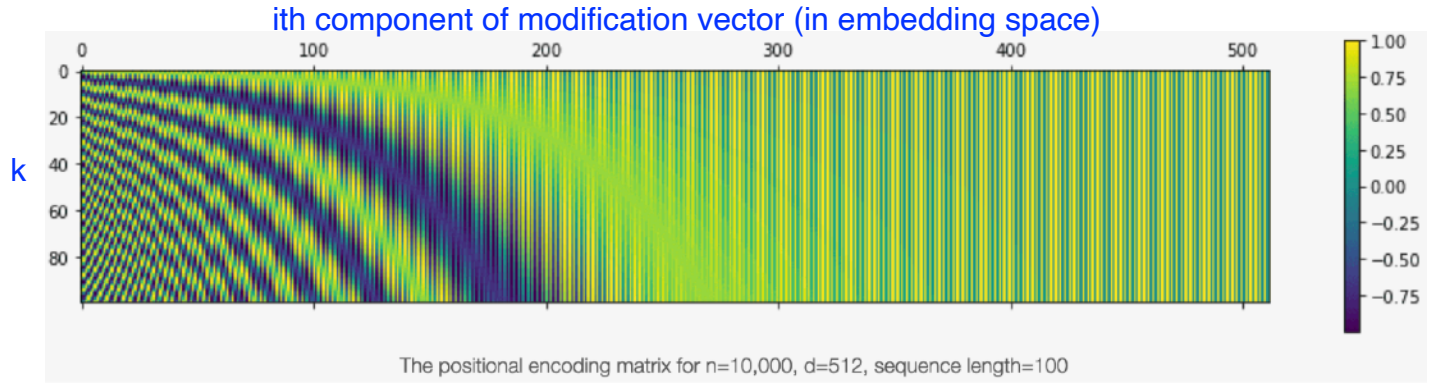
Example of specified encoding:

Say we want to encode the position of a token from a vocabulary of length L using a d-dimensional embedding space. One suggested approach* is to first divide the embedding space into even and odd components and then use a harmonic sequence of sin and cos functions to define a modification vector for a position k of a given token as:

(even) (odd)

$$U_{2n}(k) = \sin\left(\frac{k}{\alpha^{2n/d}}\right) \qquad U_{2n+1}(k) = \cos\left(\frac{k}{\alpha^{2n/d}}\right)$$

where n is an index running from 0 to L/2 and α is a large constant (typically chosen to be ~10000 for LLMs)

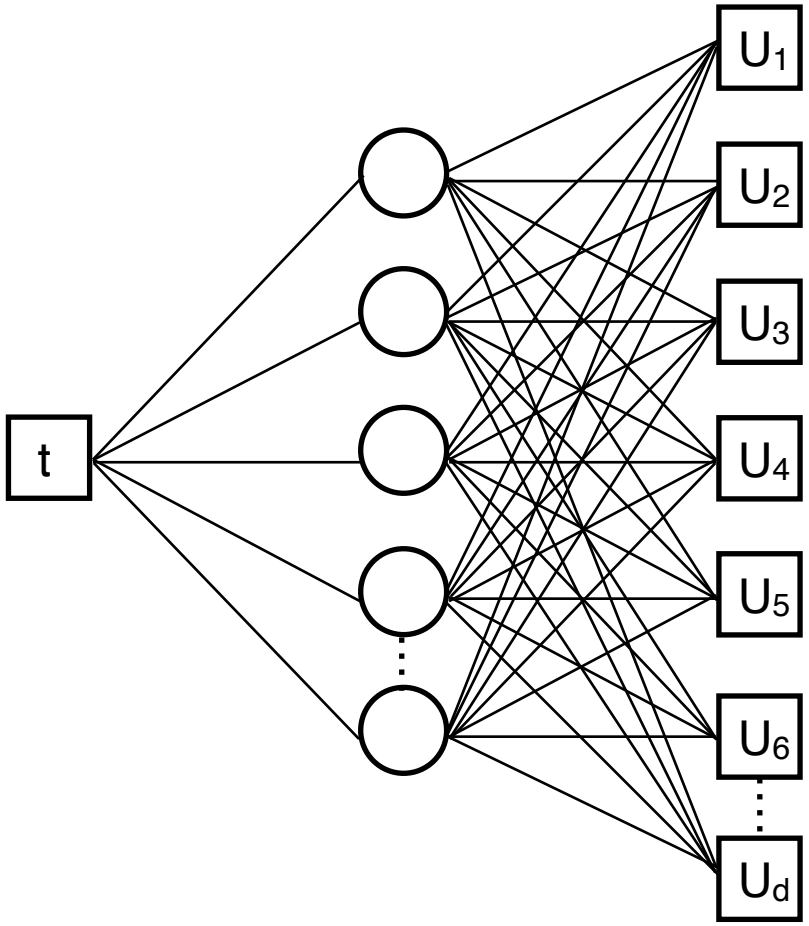


- vector is normalised between -1 and 1 (good for stability)
- unique encoding for each position
- nearby positions have more similar looking vectors (relative distance)

* "Attention is All You Need," Vashwani et al., arXiv:1706.03762

An encoding can also be learned using a simple ANN!

Say we have some time measurement from a particular detector element:

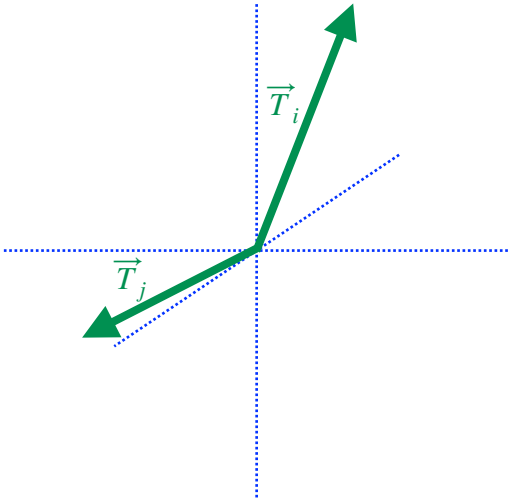


$$\overrightarrow{T}_i = \overrightarrow{T}_i^{init} + \overrightarrow{U}$$

The encoding is emergent from the learning process!

Feeding the value through some activation functions followed by a fully connected layer can be used to produce a modification vector in the embedding space, which can be added as an encoding to the token vector

Ok, we now have our token vectors,



let’s move on to assessing associations…

How Learned Interactions Take Place

The learned associations between tokens are characterised by tuned weights in two matrices (called "**Query**" and "**Key**") that create vectors in an "association space," where the strength of associations are represented by the alignment of these vectors.

The impact of these associations are then defined by another learned matrix of weights (called "**Value**"), which provide modification vectors to be added to the tokens.

"**Attention**" is the application of these modifications, weighted by the strength of the associations.

Query

For the ith token vector, we now want to allow the presence of other tokens in the sequence (*e.g.* words in the sentence, measurements from other detector sensors etc.) to provide context so as to modify the interpretation.

The first step is to produce a new vector, in a **new space**, that represents what the token under consideration is 'looking for.' We will assume that this 'query vector' can be produced from some linear transformation of the token vector, and need not necessarily have the same dimensions as the embedding space. Again, the production of this will be learned from examples, we just need to provide the structure:

$$\overrightarrow{Q}_i \equiv \mathbf{W_Q}\overrightarrow{T}_i$$

where $\mathbf{W}_Q$ is a matrix of weights to be learned for the transformation and $\overrightarrow{\mathbf{Q}}_i$ is the produced 'query' vector corresponding to the the ith token.

characteristics

queries

	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}	...	C_d
Q_1	39	-55	95	-66	-64	-30	64	-57	5	-71	-41	78
Q_2	-79	24	-55	-10	-85	-71	32	-94	-88	-79	94	21
Q_3	67	-49	-2	-44	24	-43	-32	-61	-22	-14	28	3
Q_4	-47	-94	-27	[illegible]			99	12	-64	14	50	80
Q_5	-88	86	13	-66			46	-43	75	-5	11	-19
Q_6	7	-91	-34	-69				8	-57	-65	88	
Q_7	-75	-31	-87	29	18	-92		96	26	-28	-64	
Q_8	3	[illegible]								54	-27	
Q_9	-45	72	-2	-65	-75	81	60	-93	26	54	72	-39
Q_{10}	-35	-11	-9	-56	51	-59	47	-5	-65	41	-75	-94
Q_{11}	79	-71	-1	-91	-67	-2	71	84	11	71	26	69
Q_{12}	-49	-48	-27	78	64	-14	51	84	-32	-3	-63	-61
:	:	:	:	:	:	:	:	:	:	:	:	:
Q_k	:	:	:	:	:	:	:	:	:	:	:	:

$\mathbf{W_Q}$

(learned conversion weights)

We can thus produce a list of query vectors:

$$\mathbf{Q} \equiv \mathbf{W_Q}\mathbf{T}^T$$

Query

Token

	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	...	Q_k
4	38	-68	-99	10	73	53	60	84	-58	67
11	67	21	-85	70	-29	42	56	9	-62	43
5	-30	74	-16	-69	-95	-36	46	-24	87	-46
5	-93	-98	52	-47	3	67	27	35	0	21
8	73	50	97	-36	-49	39	18	96	-77	17
15	58	69	25	-87	87	92	97	18	95	11
3	27	-85	26	1	-29	86	57	13	-51	-62
8	-40	74	-47	72	15	-38	31	59	4	-37
1	-32	-4	62	-44	35	44	-61	42	20	92

Key

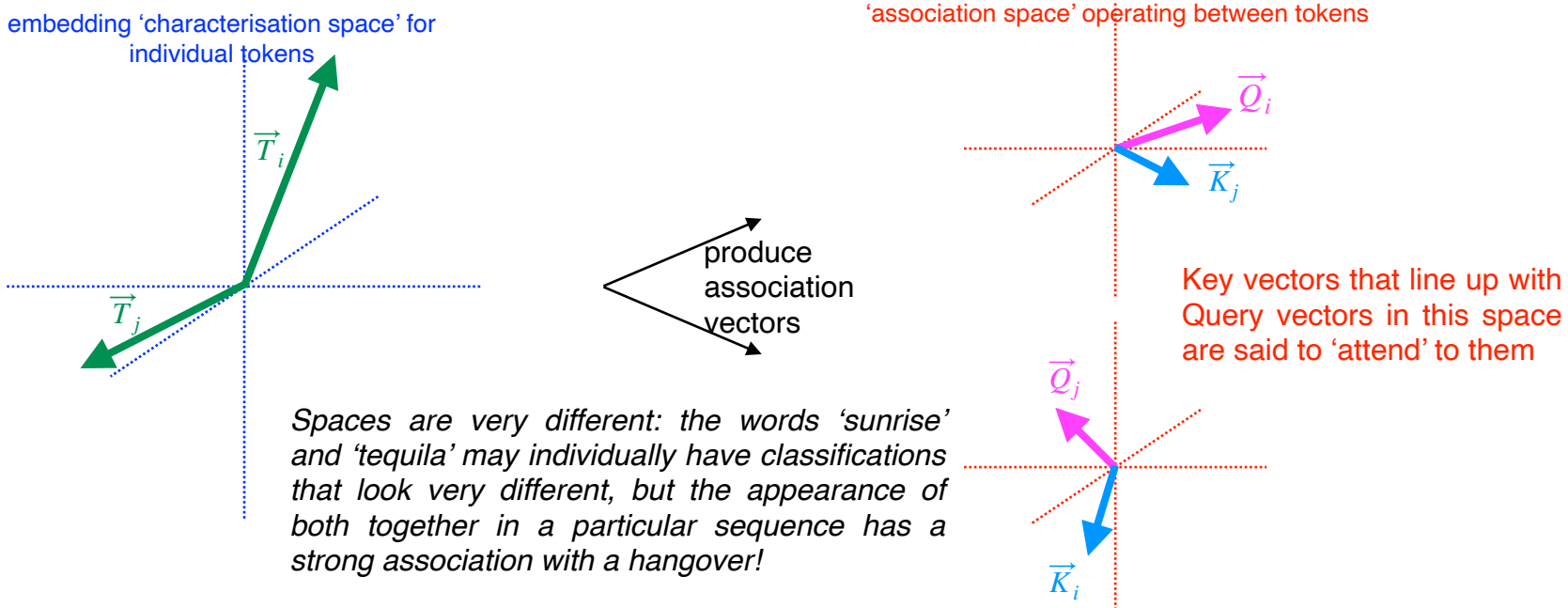
Similarly, we'll also have each of the tokens produce a new vector, in the same space as the query, that represents what relevance it can offer. We will similarly assume that this 'key vector' can be produced from some linear transformation of the other token vectors. Again, this will be learned, we just need to provide the structure:

$$\mathbf{K} \equiv \mathbf{W}_K \mathbf{T}^T$$

where $\mathbf{W}_K$ is a matrix of weights to be learned for the transformation and $\mathbf{K}$ is the resulting list of key vectors, transformed from the list of token characteristics.

Key

Token	K_1	K_2	K_3	K_4	K_5	K_6	K_7	K_8	...	K_k
4	-96	-69	-48	-34	84	-62	86	-67	28	54
11	35	66	17	-35	-86	-16	-49	-18	76	-90
5	18	-68	56	-31	87	-74	-47	-24	-83	-43
5	69	69	-66	-14	-36	-56	17	78	16	-91
8	-31	-6	77	-26	79	-56	70	-18	54	-49
15	-40	6	-65	-1	27	85	-1	8	92	60
3	-33	-81	63	43	58	-27	-14	19	16	-75
8	23	-31	-75	-22	-71	-60	-46	-16	-61	-79
1	88	12	-35	-45	-94	-78	60	-8	60	81



Spaces are very different: the words 'sunrise' and 'tequila' may individually have classifications that look very different, but the appearance of both together in a particular sequence has a strong association with a hangover!

Key vectors that line up with Query vectors in this space are said to 'attend' to them

So, the dot product between Key and Query indicates the strength of association, which can be fed to softmax to give a comparative relevance on a scale of 0-1

Value and Attention

We now want the characterisation for each token vector to be modified by the presence of each of the other tokens according to the computed strength of their associations. *But how should these modifications be done?*

This, too, is learned! Tuned weights define a series of vectors, back in the original embedding space*, that are added to the original token vectors to modify their directions. These are the 'value' vectors, which thus provides a list of proposed modifications due to associations with a each token.

Value

Token	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8	V_9	V_{10}	...	V_d
4	14	-91	-26	-53	80	-10	96	87	-33	-48	-89	-76
11	-52	17	83	25	-60	20	35	63	-84	10	32	-80
5	-63	-10	56	18	-85	-38	53	30	-56	-71	-63	11
5	33	87	47	-39	-47	14	-20	-96	-34	55	-92	32
8	-22	61	-82	-79	-66	89	-36	87	53	-70	-97	42
15	-49	-47	-22	55	34	-41	88	-13	8	9	97	-73
3	23	62	-49	-16	69	-59	-67	-6	35	21	87	37
8	36	-41	-33	-85	9	-96	-54	-3	-37	15	-81	33
1	58	80	-22	-44	-85	-20	89	-11	97	-17	-2	95

Attention** is then defined as the matrix of proposed modifications multiplied by the computed strength of the association:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_K}}\right)V$$

dimensionality of Q and K matrices, added to keep the product of 'raw' Q and K values in a reasonable range for softmax

Attention

Token	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	...	A_d
4	19	8	-1	-1	-34	-4	-3	56	77	28	-32	16
11	-28	-56	23	13	-63	20	-4	47	-78	7	7	19
5	49	23	-6	-22	-9	-51	-1	-14	8	34	2	20
5	51	-8	-26	-77	35	-24	-24	-26	-40	9	-42	-13
8	85	-24	-2	-29	47	46	-29	-8	-13	-1	63	19
15	0	-3	-35	18	-33	1	-2	-2	-15	15	20	27
3	30	11	-2	53	-25	-20	35	-27	-5	-1	13	-9
8	-34	-37	0	-23	-11	-1	3	30	-8	11	-36	-3
1	38	0	-48	-49	-1	-3	-4	26	-4	35	69	19

*In practise, this is done in 2 steps: a learned value matrix is first computed in the same space as the key and query, and a second learned matrix then upscales this to the dimensionality of the embedding space.

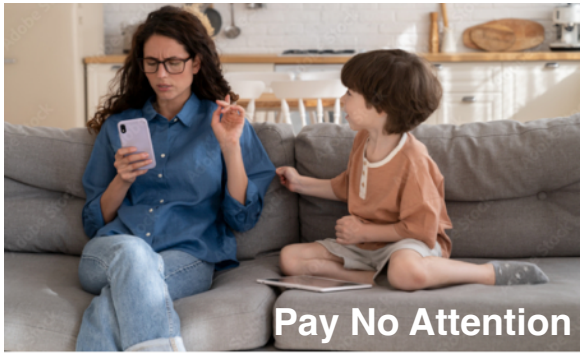
** Bahdanau et al., arXiv:1409.0473

Masking

There are times when we may want to remove the influence of certain tokens. This can, for example, be for 'padded' inputs that don't exist (sometimes more computationally efficient than removing them altogether), or as a diagnostic to separate the impact of certain information, or guiding the training process to only look at certain bits of the provided information.

Embedded Characteristics

Token	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}	...	C_d
4	73	-4	49	-26	-64	-63	54	-14	-33	-89	-51	98
11	89	-18	-29	44	58	-43	-2	4	-81	-18	29	95
5	-56	46	68	-93	4	-69	-66	-66	33	3	14	-83
5	-56	46	68	-93	4	-69	-66	-66	33	3	14	-83
8	24	-11	-12	59	-14	-13	48	-36	38	12	16	77
15	4	16	-9	69	-54	83	19	68	-21	37	75	-62
3	34	46	65	48	38	24	-39	92	98	-18	70	-69
8	24	-11	-12	59	-14	-13	48	-36	38	12	16	77
1	15	50	-43	58	-13	-79	-96	22	73	-44	-51	48



A natural way to do this is to set the corresponding attentions to zero, so that they cannot make modifications.

But rather than directly setting the attentions themselves to zero, the corresponding arguments of the softmax in attention are instead set to $-\infty$, which does the job and also keeps the calculations of relevance normalised to the range between 0-1 for the tokens that are present:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\text{mask} + \frac{QK^T}{\sqrt{d_K}}\right)V$$

mask elements set to 0 for present tokens, and to $-\infty$ for absent tokens

Multi-Headed Attention

Typically, the key and query space is broken into multiple pieces and processed simultaneously as 'multi-headed attention.' This takes advantage of parallel processing to efficiently provide **different** independent assessments of Attention

W_Q

characteristics

queries	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	...	Cd
Q1	39	-55	95	-66	-64	-30	64	-57	5	-71	-41	78
Q2	-79	24	-55	-10	-85	-71	32	-94	-88	-79	94	21
Q3	67	-49	-2	-44	24	-43	-32	-61	-22	-14	28	3
Q4	-47	-94	-27	13	21	-47	-99	12	-64	14	50	80
Q5	-88	86	13	-66	10	-39	46	-43	75	-5	11	-19
Q6	7	-91	-34	-69	-4	38	-54	-18	8	-57	-65	88
Q7	-75	-31	-87	29	18	-92	35	91	96	26	-28	-64
Q8	3	-26	-8	-97	-77	-43	-5	-38	-1	-4	-54	-27
Q9	-45	72	-2	-65	-75	81	60	-93	26	54	72	-39
Q10	-35	-11	-9	-56	51	-59	47	-5	-65	41	-75	-94
Q11	79	-71	-1	-91	-67	-2	71	84	11	71	26	69
Q12	-49	-48	-27	78	64	-14	51	84	-32	-3	-63	-61
:	:	:	:	:	:	:	:	:	:	:	:	:
Qk	:	:	:	:	:	:	:	:	:	:	:	:

W_K

characteristics

keys	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	...	Cd
K1	-80	83	20	54	-90	-18	57	-39	64	-76	-5	2
K2	25	42	87	24	-13	83	33	85	-74	-45	7	43
K3	72	-35	-90	69	83	-64	-15	-45	66	-44	-97	21
K4	-40	63	52	-60	7	-74	31	-36	54	-67	-51	-91
K5	-34	-62	-81	81	-5	81	98	74	82	36	50	69
K6	-9	-76	-36	-77	-64	-48	-78	-64	32	-98	80	-59
K7	-32	83	-24	56	-70	-50	-38	-88	49	-3	92	-36
K8	62	9	92	-67	12	3	-59	-12	78	93	-66	85
K9	73	-67	77	-49	-61	54	-14	-98	97	9	13	-38
K10	85	-59	-5	43	-29	-74	12	61	-8	-15	-81	-55
K11	-27	31	66	-45	-66	22	-39	-76	-67	63	29	64
K12	68	68	20	5	60	-20	-80	-11	-75	-81	40	87
:	:	:	:	:	:	:	:	:	:	:	:	:
Kk	:	:	:	:	:	:	:	:	:	:	:	:

Attention

Token	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	...	Ad
4	19	8	-1	-1	-34	-4	-3	56	77	28	-32	16
11	-28	-56	23	13	-63	20	-4	47	-78	7	7	19
5	49	23	-6	-22	-9	-51	-1	-14	8	34	2	20
5	51	-8	-26	-77	35	-24	-24	-26	-40	9	-42	-13
8	85	-24	-2	-29	47	46	-29	-8	-13	-1	63	19
15	0	-3	-35	18	-33	1	-2	-2	-15	15	20	27
3	30	11	-2	53	-25	-20	35	-27	-5	-1	13	-9
8	-34	-37	0	-23	-11	-1	3	30	-8	11	-36	-3
1	38	0	-48	-49	-1	-3	-4	26	-4	35	69	19

Attention Head 1

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_K}}\right)V$$

Attention Head 2

$$\text{Attention}(Q', K', V') = \text{softmax}\left(\frac{Q'K'^T}{\sqrt{d'_K}}\right)V'$$

etc.

$$\text{Attention}(Q'', K'', V'') = \text{softmax}\left(\frac{Q''K''^T}{\sqrt{d''_K}}\right)V''$$

$$\mathbf{T}^* = \mathbf{T} + \text{Attention}$$

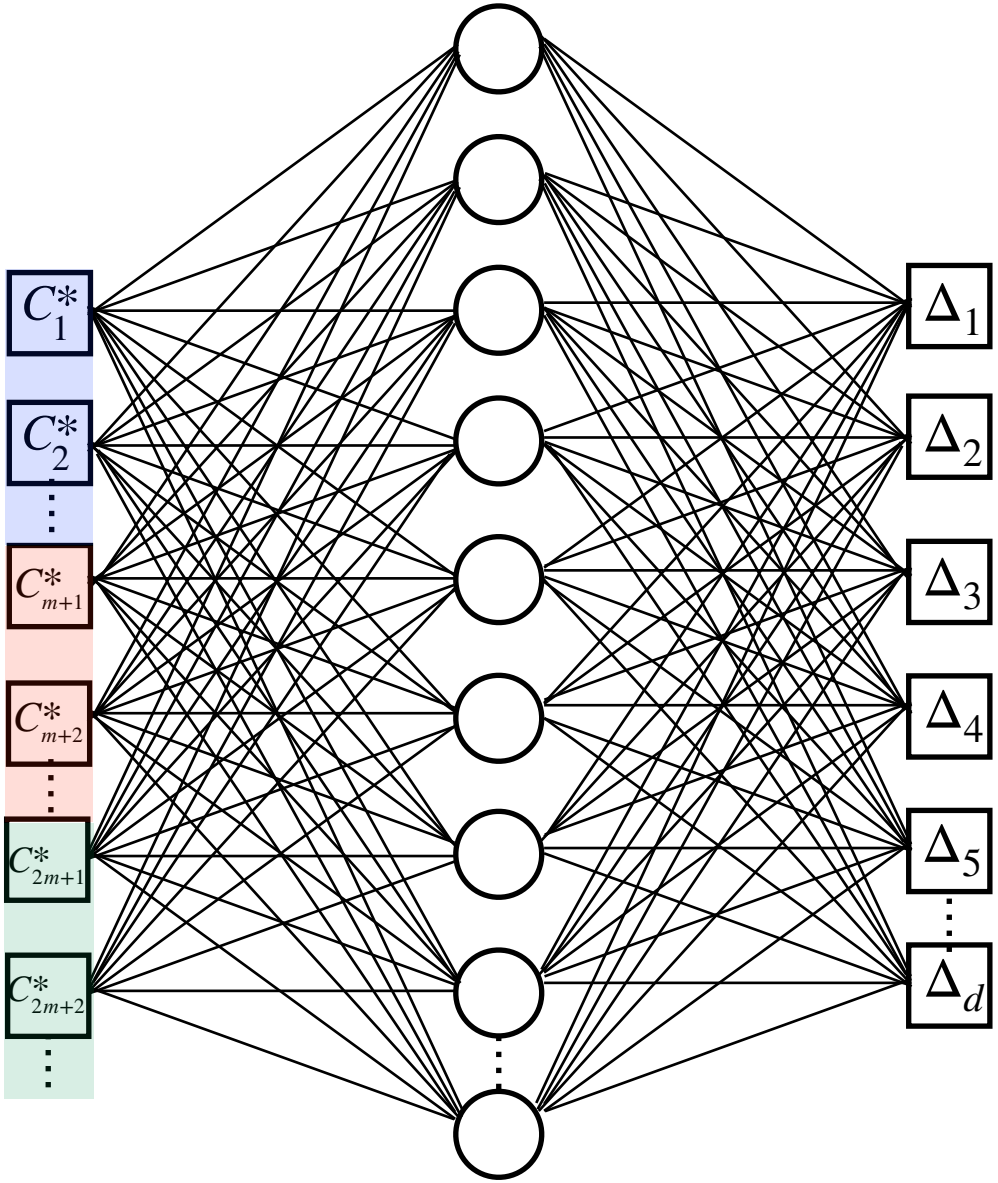
(for stability, re-normalise $\mathbf{T}^*$ to have mean 0 and unit variance)

The new set of token vectors are a dynamic, adaptive modification of the initial set

modified characteristics

Token	C*1	C*2	C*3	C*4	C*5	C*6	C*7	C*8	C*9	C*10	C*11	C*12	...
4	1.5	0.3	0	-0.7	-0.1	-2.8	-0.5	-0.8	-0.9	0.3	-2.1	-0.6	...
11	-0.1	0.9	2.9	1.1	-1.1	0	0	-0.3	1.3	2.3	-0.9	0.4	...
5	0.6	-0.7	0	0.1	1.3	-1.1	-1.5	0	-1.8	0.9	-0.9	1	...
5	0.4	-0.1	0	1	1	0.5	1.9	0	-0.9	1.9	0	0.3	...
8	0.2	-0.5	0.7	0	-0.5	2.9	1.4	-0.9	-1.1	0.5	0.3	2.6	...
15	1.9	-1.9	0	0	-2.4	-1.9	-2.6	-1.6	0.1	0	1	-2.9	...
3	0.1	1.9	0.6	-0	-2.8	0.2	0.2	0	-1.9	0	1.3	0	...
8	2.5	0.7	-0.1	1.5	-1.4	2.2	0.2	0.4	0	-1	0.3	-1.4	...
1	0.4	0	1.3	1.9	0	1	0.7	1.8	1.1	-2.8	0.5	-3	...

Finally, **for each of the vectors in the modified Token matrix**, elements from different attention heads 'talk' to each other via a simple feed-forward network, typically via a single hidden layer with larger dimension (this layer often contains most of the weights!).



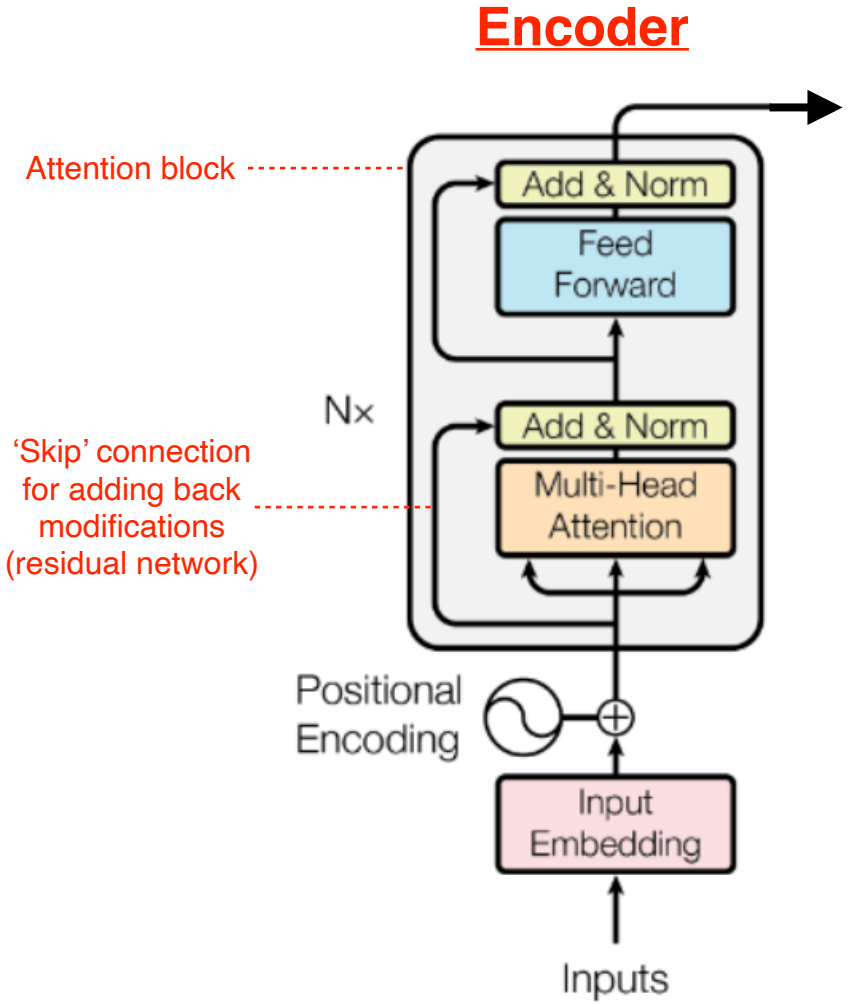
The same learned network is used to consolidate information across the different attention heads and permit a (piece-wise) non-linear mixing to yield more nuanced tweaks to the modified token matrix

$$\mathbf{T}^{**} = \mathbf{T}^{*} + \boldsymbol{\Delta}$$

(and normalise again)

Together, this whole process constitutes one '**Attention Block.**' One can then string multiple such blocks together for even deeper learning

The process described so far constitutes an 'encoder,' for providing context to input data



To make use of this, it's convenient to have a single vector ($\vec{h}_{enc}$) that captures relevant information from the final 'hidden state' of the encoder.

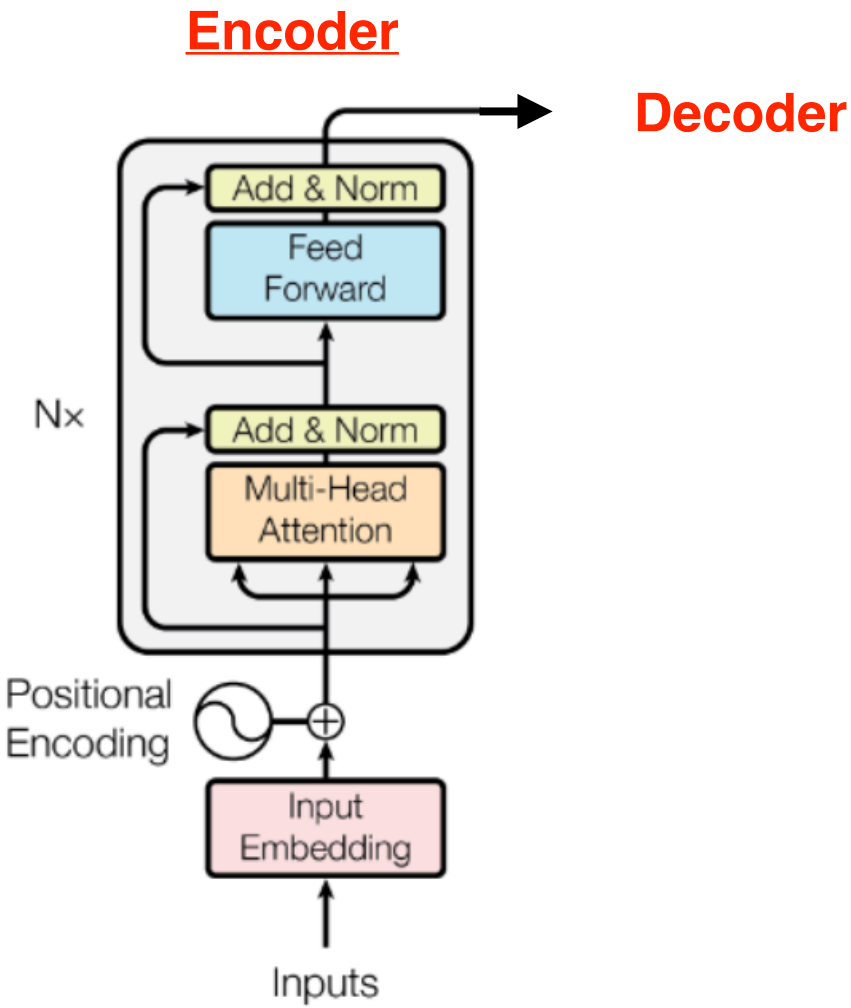
There are a number of ways to do this, including:

1) Sum or linearly combine all the individual modified token vectors
2) Explicitly add another 'dummy' vector to the token matrix to gather this information
3) Just take the last vector in the token matrix to represent this

In any case, all token vectors interact with each other via Attention. If you treat a given vector as having the relevant information, the learning process will adjust weights to make this the case, because that's what will work best to minimise the loss function!

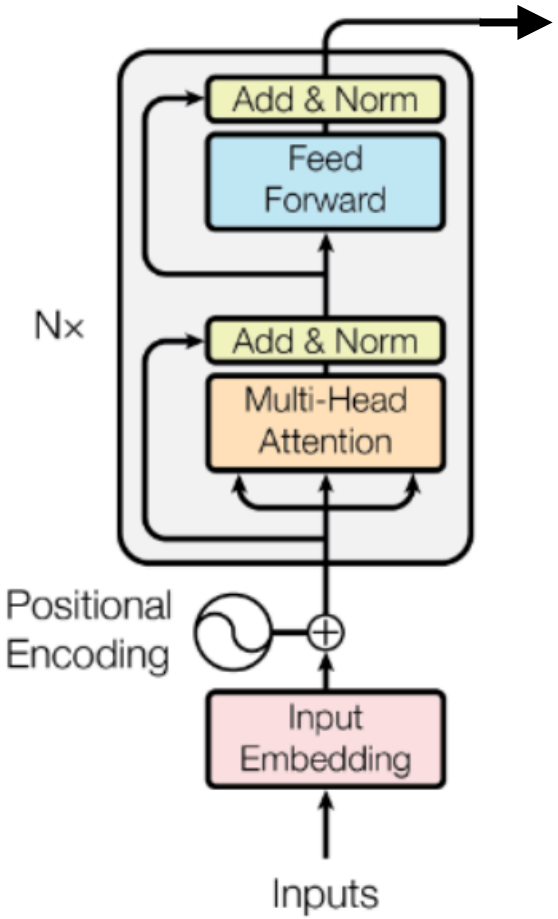
(Deep learning magic!)

In a language translation task, the encoder output (source language) then needs to be fed into a **decoder** to generate a translated sentence in the target language word-by-word



In a language translation task, the encoder output (source language) then needs to be fed into a **decoder** to generate a translated sentence in the target language word-by-word

Encoder



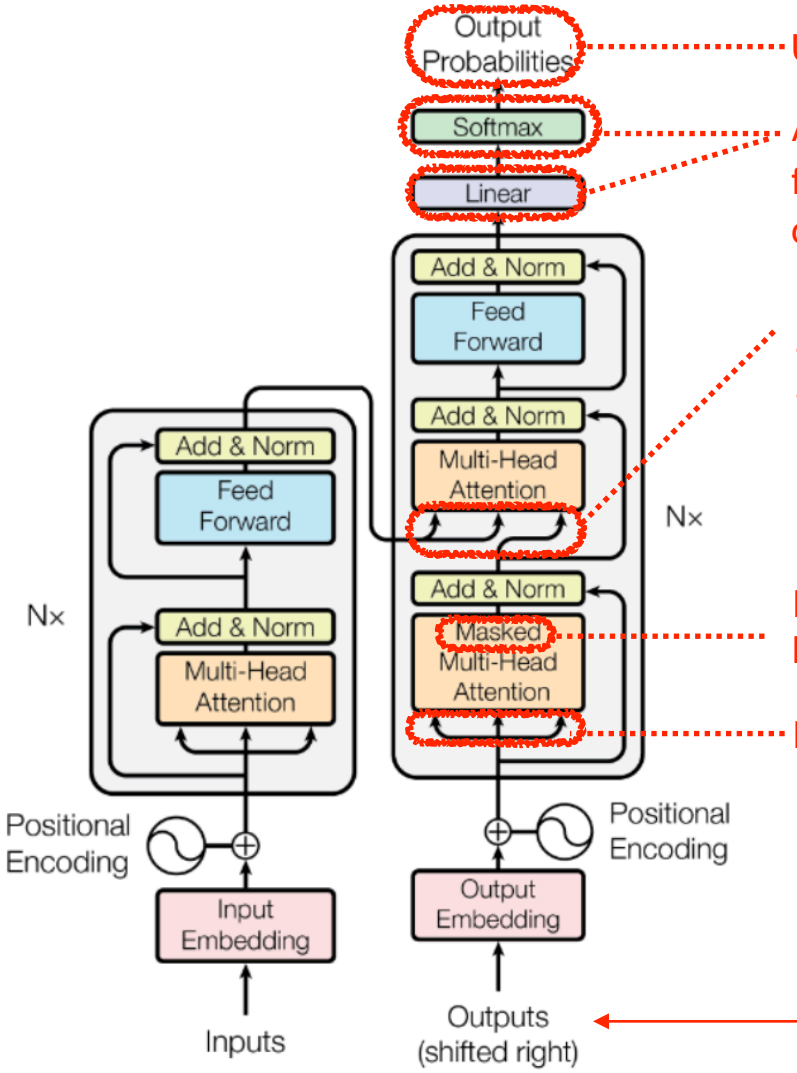
Decoder

We will also need a 'hidden state vector' for the decoder to represent information about the next word to be generated, in the same embedding space of tokens in the target language. This can be initially generated, for example, by applying a 'decoder initialisation' matrix (to be learned) to the hidden state vector from the encoder:

$$\vec{h}_{dec} = \mathbf{D}_{init}\vec{h}_{enc}$$

The decoder uses the same sorts of elements in a modified configuration. We'll run through this **very briefly**, and then move on to a practical example of transformers applied to a problem in experimental physics…

Encoder-Decoder Pair Used for Translation*



Used to generate next word in translated sentence

A simple, fully connected layer (matrix transformation) from $\vec{h}_{dec}$ to a vocabulary vector, providing weights that can be turned into probabilities for the next word choice

Cross-attention: Q from decoder (looking for relevance of the next word to be generated in the target language), K from encoder (relevance provided by the input source) and V also from the encoder (this is how basic characteristics should be adjusted). Attention is then applied to $\vec{h}_{dec}$

During training, don't look at answers for examples provided beyond the words already generated (*i.e.* don't cheat!)

Look at context of words already generated (self-attention)

sentence to be translated

translated words generated so far

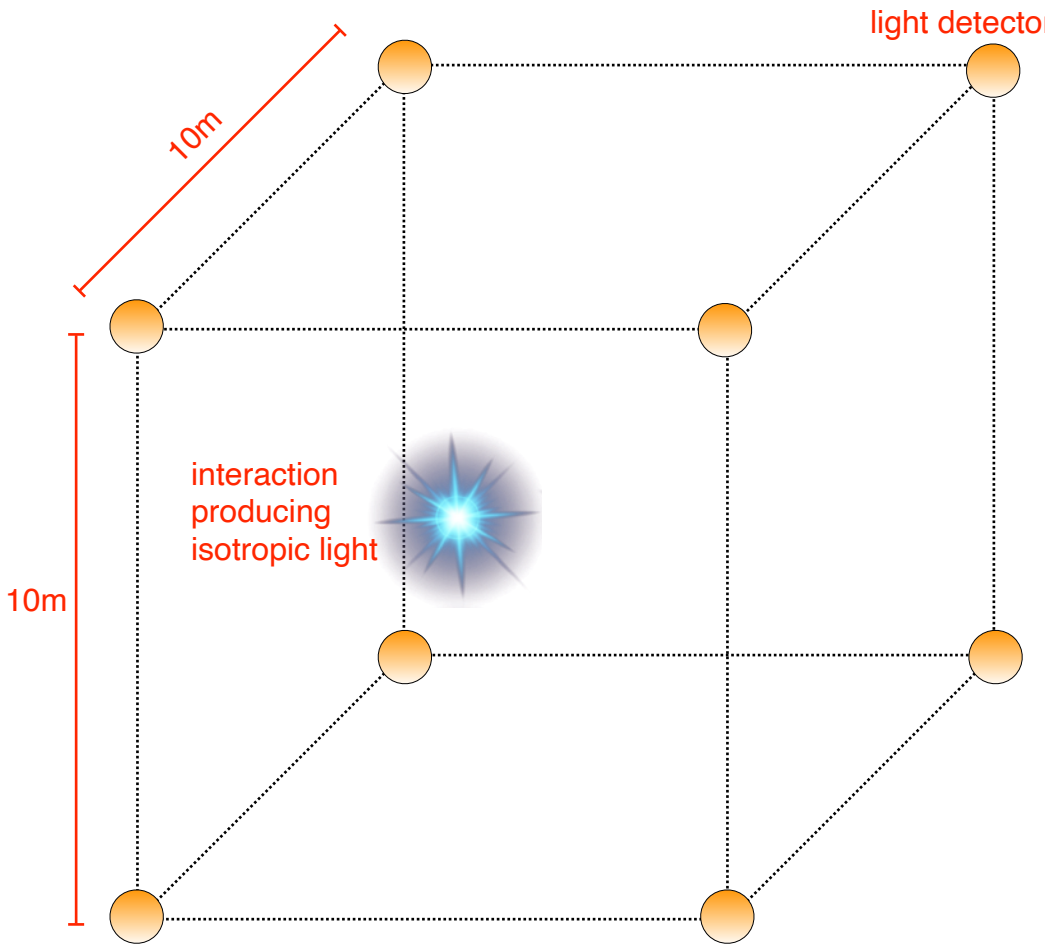
PLUS the hidden state vector $\vec{h}_{dec}$

* "Attention is All You Need," Vashwani et al., arXiv:1706.03762


Let's
Play

Example of an encoder-based reconstruction algorithm for position using timing

(Thanks to Cal Hewitt for help with this!)



Simplified Idealised Scenario

- Detection volume is a 10m cube with a fully contained non-dispersive medium of refractive index 1.5
- Light from the interaction is point-like, instantaneous and isotropic
- PMTs have uniform 4π efficiency and an instantaneous response function
- Each PMT has a trigger efficiency of 0.8 when the event is at the centre of the detector, scaling with the inverse square of distance to the event
- A valid event trigger requires at least 4 PMTs to register hits

Generate Training Data

```python
class Event_Generator:

    def __init__(self):
        # Define PMT positions by permuting the PMT separations
        self.pmt_x, self.pmt_y, self.pmt_z = zip(*itertools.product([-PMT_Sep, PMT_Sep], repeat=3))

    def generate(self):
        # Generate event randomly within the cubic volume
        x, y, z = (PMT_Sep*np.random.uniform(-1, 1) for _ in range(3))

        pmt_ids, hit_times = [], []
        nhits = 0

        for pmt_id in range(8):
            distance_to_pmt = np.linalg.norm((x - self.pmt_x[pmt_id], y - self.pmt_y[pmt_id], z - self.pmt_z[pmt_id]))
            hit_time = (distance_to_pmt / Phase_Velocity)
            # Scale PMT trigger efficiency by 1/distance^2 relative to the cube centre
            # and then see if it triggers when Poisson fluctuated
            if np.random.poisson(lam=Eff_Mid*(3*(PMT_Sep**2)/4) / (distance_to_pmt ** 2)) > 0:
                nhits += 1
                hit_times.append(hit_time)
                pmt_ids.append(pmt_id + 1)
            else:
            # In this example, we will record ALL PMT entries, but will pad the PMTs that are not hit
                hit_times.append(0)
                pmt_ids.append(0)

        if nhits >= 4: # Trigger the overall event on at least 4 hits
            hit_times = np.array(hit_times)

            # Make a 2-D array by stack the array of PMT ID's on top of the array of hit times
            # (i.e. column-wise stacking for axis=1, as opposed to along the same dimension for axis=0)
            # and return this along with the truth array (net output will need to be in the same
            # form as the truth information for MSELoss to recognise)
            return np.stack((pmt_ids, hit_times), axis=1), np.array((x, y, z)).astype(np.float32)
        else:
            return self.generate()  # Try again if the event hasn't triggered
```

```python
class Dataset(Dataset):

    # Generates data set of n_events examples with truth information on initialisation
    def __init__(self, n_events):
        self.n_events = n_events
        self.generator = Event_Generator()
        self.data = [self.generator.generate() for _ in range(n_events)]

    def __len__(self):
        return self.n_events

    def __getitem__(self, idx):
        return self.data[idx]
```

Define the Network

```
# Create the model class: 32 embedding dimensions, 4 atten heads, 64 nodes for the feed-forward, 2 atten blocks
class TransformerNN(nn.Module):
    def __init__(self, dim_embed=32, nhead=4, dim_feedforward=64, num_layers=5):
        super().__init__()
        self.d_model = dim_embed

# Fully-learned embeddings for each PMT ID + 1 'dummy' token for hidden state summary vector
        self.pmt_id_embedding = nn.Embedding(9, dim_embed)

# Single linear with simple activation y=x for hit time encoding onto token space
        self.ht_encoding = nn.Linear(1, dim_embed)

# Single linear with simple activation y=x for PMT position encoding onto token space
        self.pmt_pos_embedding = nn.Linear(3, dim_embed)

# Set transformer components. Define the ordering sequence for indexing the batch number
        encoder_layer = nn.TransformerEncoderLayer(d_model=dim_embed, nhead=nhead,
            dim_feedforward=dim_feedforward, batch_first=True)
        self.transformer_encoder = nn.TransformerEncoder(encoder_layer, num_layers=num_layers)
        self.prediction = nn.Linear(dim_embed, 3) # Three predicted values: x,y,z of event
```

```
# Define the sequence of forward flow through the network
    def forward(self, x):

# Create a torch tensor for the number 0 (which will be the location in the token stack for the dummey vector)
# to be associated with the device/processor dealing with the input x, and repeat this for every event
# keeping a dimension for the token number and a dimension for the token entry
        dummy_tokens = self.pmt_id_embedding(torch.tensor(0).to(x.device)).repeat(x.shape[0], 1, 1)

# Create a torch tensor for the value 'FALSE' (default mask value) to be associated with the device/processor
# dealing with the input x, and repeat this for every event, keeping a dimension for the token number
        dummy_tokens_padding_mask = torch.tensor(False).to(x.device).repeat(x.shape[0], 1)

# For each event and PMT, set the mask flag true if the PMT ID (0 index in 3rd entry) is set to zero
        padding_mask = x[:,:,0] == 0
        padding_mask = torch.cat((dummy_tokens_padding_mask, padding_mask), axis=1)

# For each event and PMT, create the embedding space (pass each PMT ID as long integer for torch to identify embedding)
        pmt_emb = self.pmt_id_embedding(x[:, :, 0].long())

# For each event and PMT, encode hit time (1 index in 3rd entry). An exta dimension is added (unsqueeze(2)) for torch
# to recognise generic format for layer-to-layer linear map, though we are just mapping a single point to a full layer
        ht_enc = self.ht_encoding(x[:, :, 1].float().unsqueeze(2))

# Re-use memory space of x for tokens: combine dummy tokens & sum of embeddings & encodings in list-wise way (axis=1)
        x = torch.cat((dummy_tokens, pmt_emb + ht_enc), axis=1)

        x = self.transformer_encoder(x, src_key_padding_mask=padding_mask) # Modify tokens: Go Transformer!!

# Re-use x again to produce predictions from hidden state vector (all batches, 0th pos. token, all token-space values)
        x = self.prediction(x[:, 0, :])

        return x
```

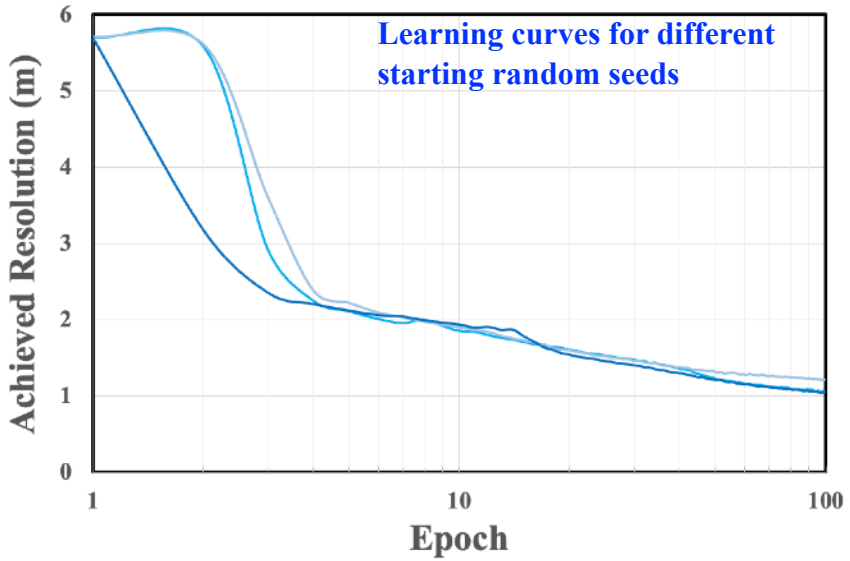
Training Loop

```
N_TRAIN_EXAMPLES = 32000
BATCH_SIZE = 32
N_EPOCHS = 100

print("Generating Training Data...")
train_dataset = Dataset(n_events=N_TRAIN_EXAMPLES)
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE)
model = TransformerNN()
criterion = torch.nn.MSELoss()
opt = torch.optim.Adam(model.parameters(), lr=0.001)
resolutions = []

print("Start Training...")
print("Epoch  Resolution (m)")
for i_epoch in range(N_EPOCHS):
    if(i_epoch>0): print(f" {i_epoch}       {sumloss/nn:.3f}")
    sumloss=0
    nn=0
    for i_step, (batch, labels) in enumerate(train_loader):
        output = model(batch)
        loss = criterion(output, labels)
        loss.backward()
        sumloss=sumloss+np.sqrt(loss.detach())
        nn=nn+1
        opt.step()
        opt.zero_grad()

    resolutions.append(sumloss/nn)
```



Note that the transformer is fitting the event position using the PMT timing information… even though we have not given it the positions of the PMTs!

There is not a physical model here - the transformer is simply looking at individual associations (*e.g.* when these tubes are hit with these times, the position is here) and using a large number of parameters to do a numerical interpolation

But would it help to provide the PMT position information?

Let's try adding an encoding for these by modifying the following lines:

In Event_Generator, provide the PMT positions as part of the output:

```
return np.stack((pmt_ids, hit_times, self.pmt_x, self.pmt_y, self.pmt_z), axis=1), np.array((x, y, z)).astype(np.float32)
```

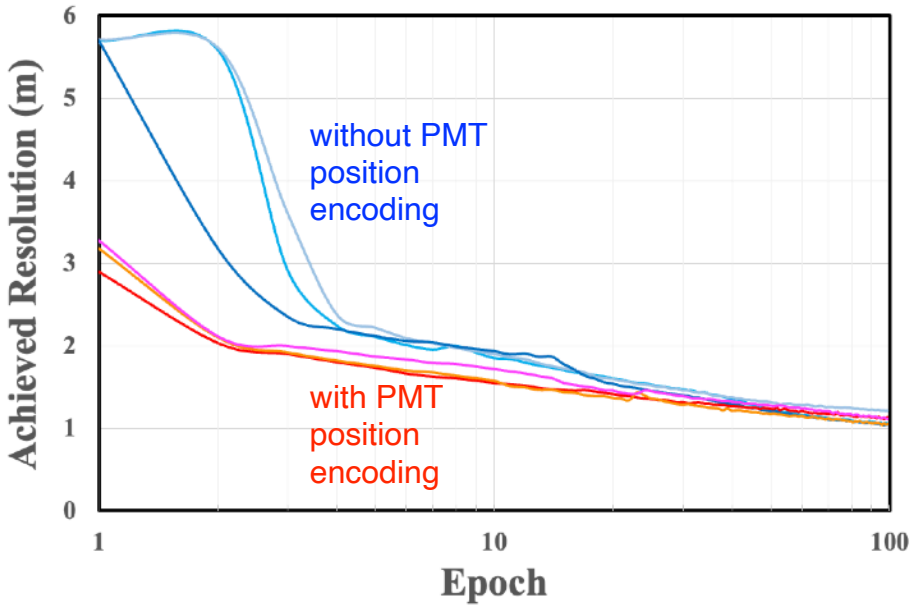
In TransformerNN, define the linear encoding of the PMT positions:

```
self.pmt_pos_embedding = nn.Linear(3, dim_embed)
```

In the forward, encode the PMT positions, adding this to the embedding along with the hit time encoding:

```
pmt_pos_enc = self.pmt_pos_embedding(x[:,:,2:5].float())
```

```
x = torch.cat((dummy_tokens, pmt_emb + ht_enc + pmt_pos_enc), axis=1)
```

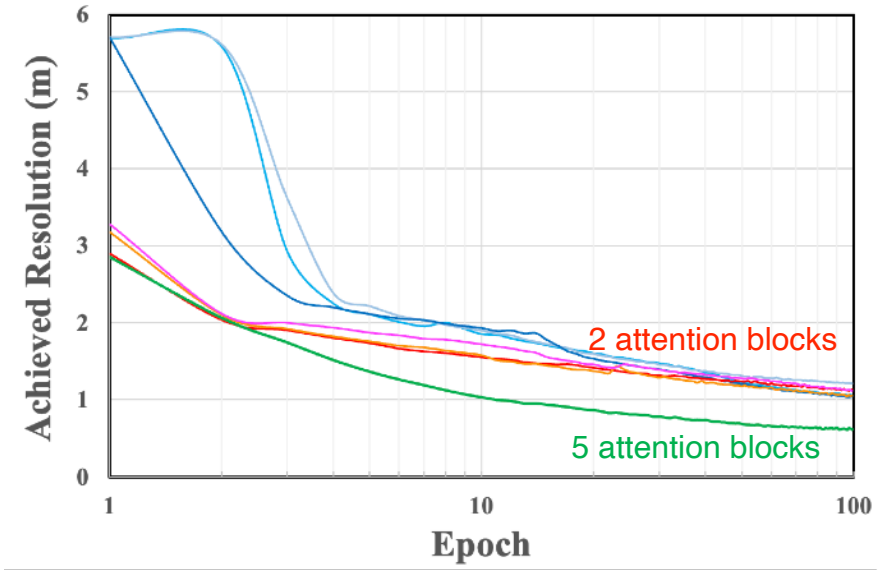


Yes, learning is faster… at least initially, before settling down to a very similar curve

We can also try adding more attention blocks:

```
class TransformerNN(nn.Module):
    def __init__(self, dim_embed=32, nhead=4, dim_feedforward=64, num_layers=2):
```

5



But, for this particular problem, it is possible to construct an analytical solution that would give the event position **exactly**, whereas here we are limited to a gradual power-law improvement!

This is because there is still no physical model, so the resolution is limited by the extent of arbitrarily tuned parameters!

Physical models **are** better, but ML approaches like this are useful when the complexity of a problem is such that analytic modelling is not tractable or too time consuming to implement