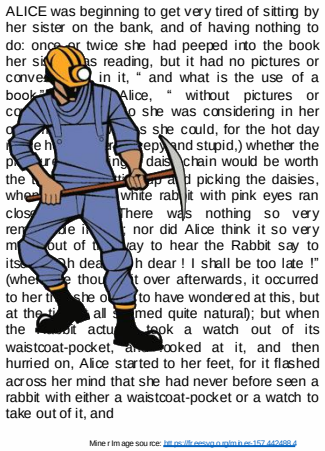
POLITECNICO DI MILANO

Sequence2Sequence Models & Transformers

Natural Language Processing

Mark Carman



Miner Image source: https://fr.vexels.com/png-svg/preview/157442488/

Lecture Contents:

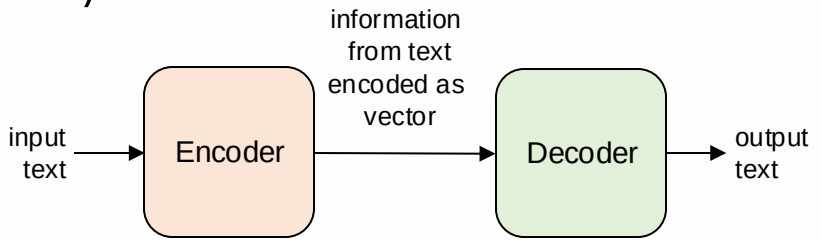
- Sequence-to-Sequence Models
- Attention Mechanisms
- Aside: Deep Learning
- Self-attention
- Transformer architecture
- Text Classification with BERT
- Text Generation with GPT

Sequence-to-sequence models

Sequence to sequence (seq2seq) models

RNNs / LSTMs turned were so powerful

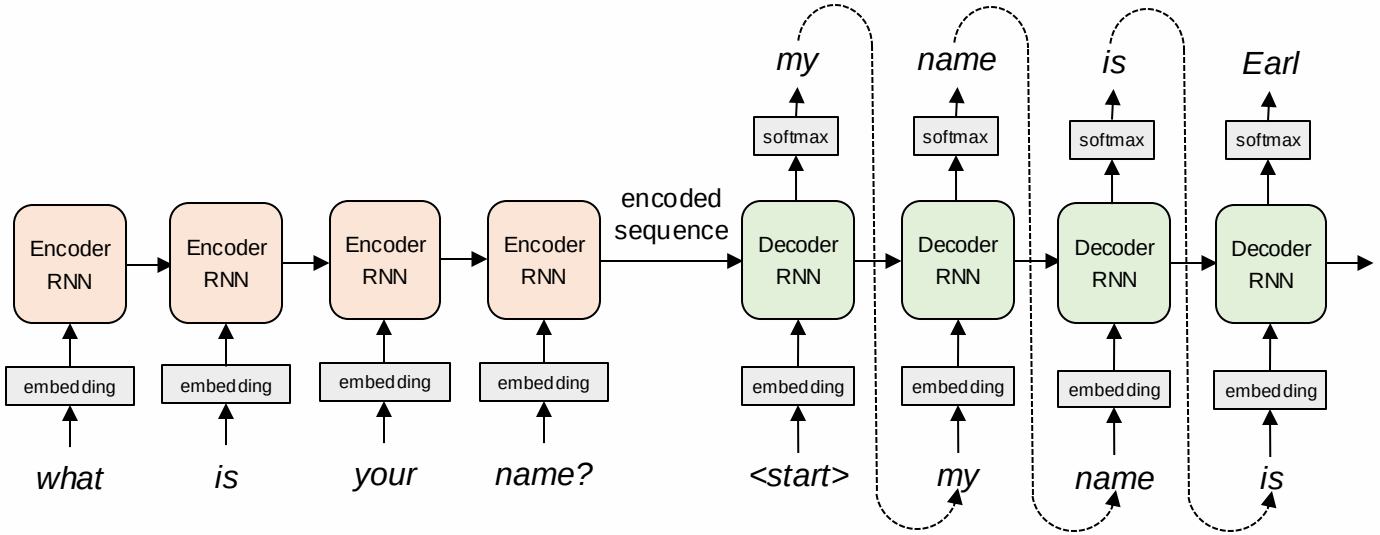
- that they were soon applied to **seq2seq** (text in and **text out**) tasks
- like **translation**, **summarization** and **dialog systems**



How can we learn translation models with LSTMs?

- by training **two different RNN** models: an **encoder** and a **decoder**

1. **Encoder**:
   - reads in the input text
   - and generates representation
   - for the sequence
2. **Decoder**:
   - takes output of encoder
   - and serializes it, by generating one word at a time



seq2seq models were Transformative

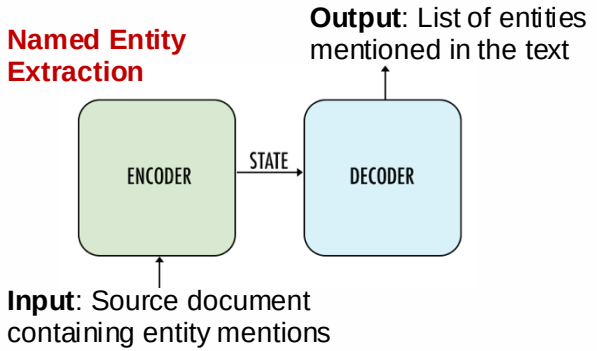
Upended state-of-the-art in wide variety of NLP tasks

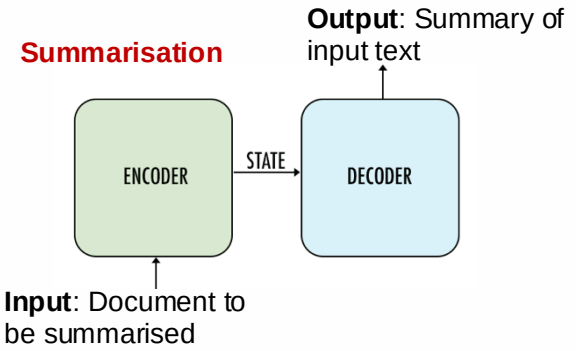
E.g. for question answering:

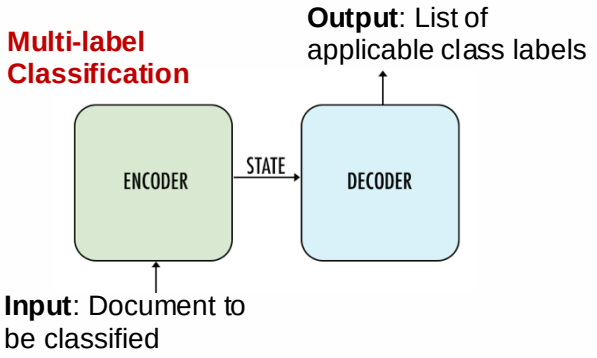
- instead of selecting best answer from fixed set of candidates
- learn to generate text containing any possible answer

MANY (if not all) tasks can be posed as seq2seq problems

- Named Entity Extraction
- Question answering
- Spelling correction
- Query reformulation
- Multi-label classification
- Translation
- Summarisation



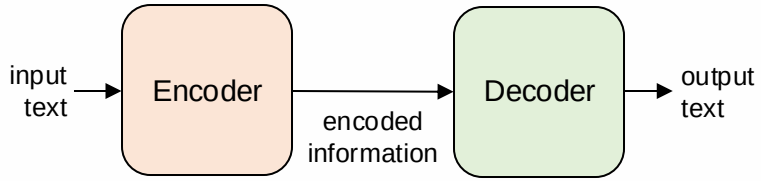




Problem of Translating Long Text

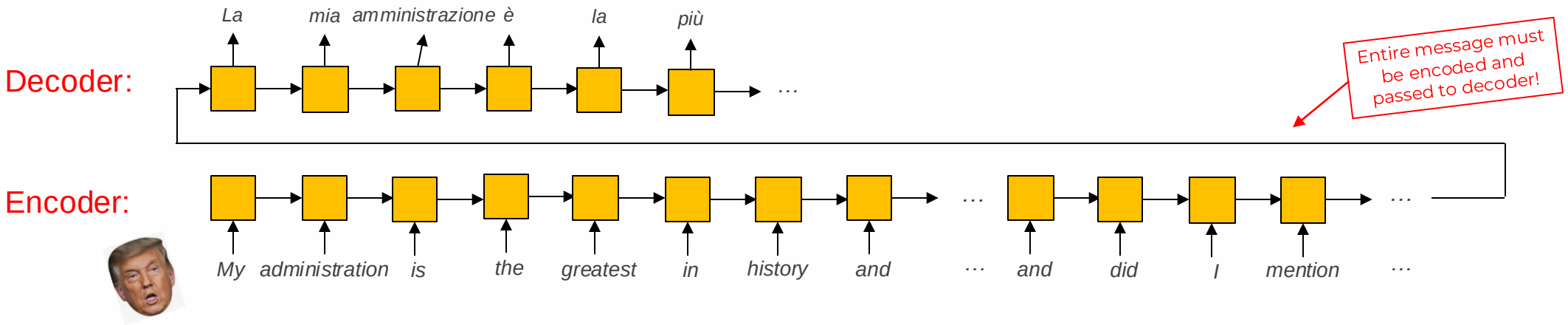
Ever thought about poor interpreters translating politicians during meetings?

- must wait for rambling politician to stop speaking
- before they can start translating…
- that's a lot of stuff to remember!



Same problem for encoder-decoder architecture

- **too much information** to pass through to the decoder
- easier to translate if decoder has access to **encoder's notes**



Attention mechanisms

Attention!

Attention is a critical building block for modern image and text processing

- what is attention?
- why is it implemented?
- how does it work?

Attention mechanisms in **computer vision** enable models to **concentrate processing** on **specific regions** of image when making predictions

- e.g. focus on certain pixels in image to determine road sign
- or to read text in images by moving from one character to the next

Attention mechanisms in **NLP** enable models to **concentrate processing** on **specific regions** of text when making predictiions

- e.g. focus on on sentiment-bearing words to determine sentiment
- or to extract named-entity that is answer to question

Motivating Attention for Translation

Attention models

- make encoded input available to decoder
- provides a direct route for the information to flow from input to output

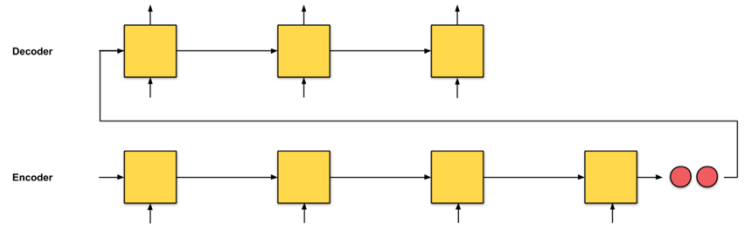
Why not directly map input words to output words?

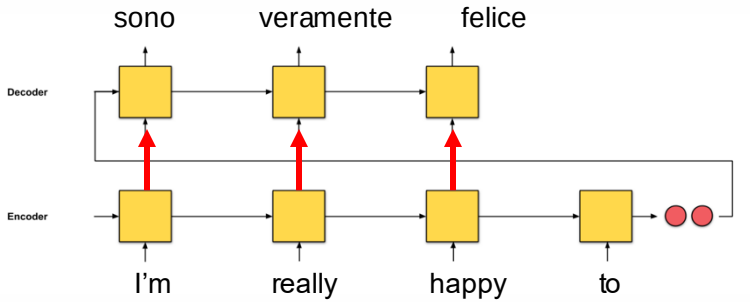
- because across languages **different number of tokens** required to describe same concept

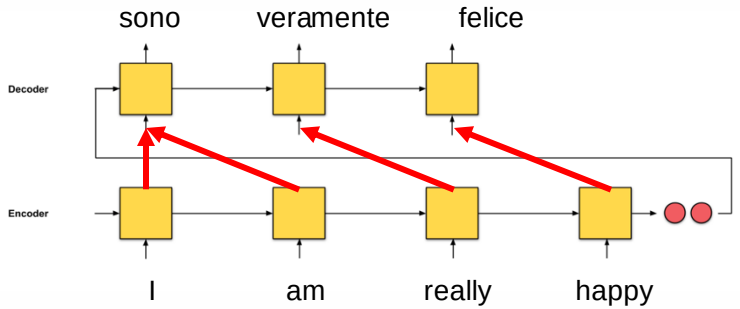
  “apple tree” => “Apfelbaum”

- and **different word order** is often used:

  “United Nations General Assembly”
  => “Assemblea Generale delle Nazioni Unite”





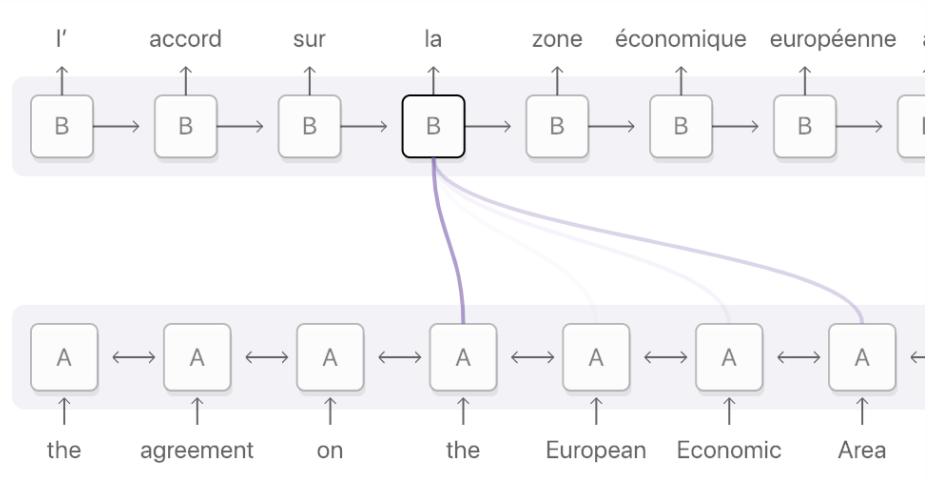


Adapted from: https://towardsdatascience.com/attn-illustrated-attention-5ec4ad276ee3

Towards attention (cont.)

Generating the right output word

- often requires knowing **more than just the current word** in input
- indeed it can require knowledge of a **future word** in the sentence
  - e.g. for determining the gender of the determinant:



Source: https://towardsdatascience.com/attn-illustrated-attention-5ec4ad276ee3

Need for attention

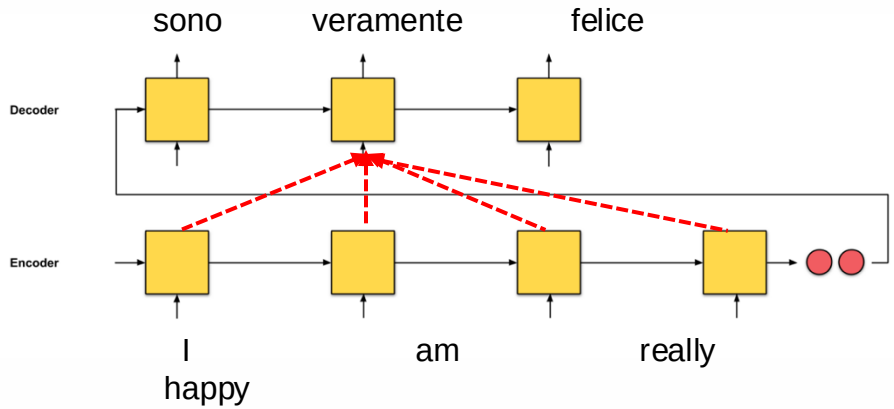
Need mechanism that can:

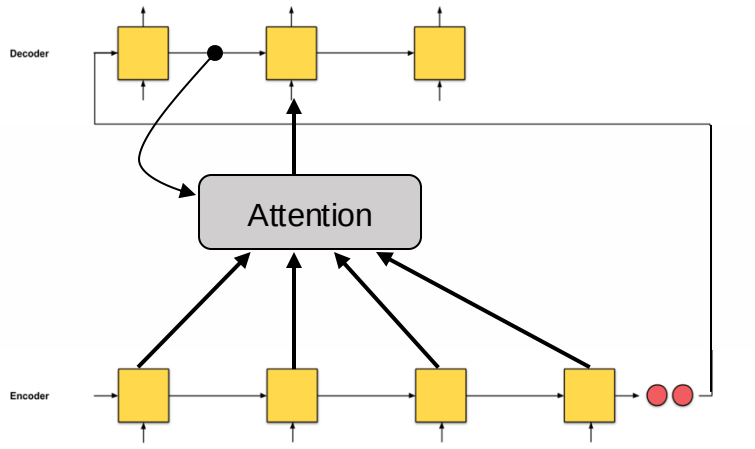
- pass information from embeddings of input word to corresponding output word

Attention:

- provides **direct route** for information to flow from input to output

What information flows into the decoder is controlled by **previous state of decoder**





Adapted from:: https://towardsdatascience.com/attn-illustrated-attention-5ec4ad276ee3

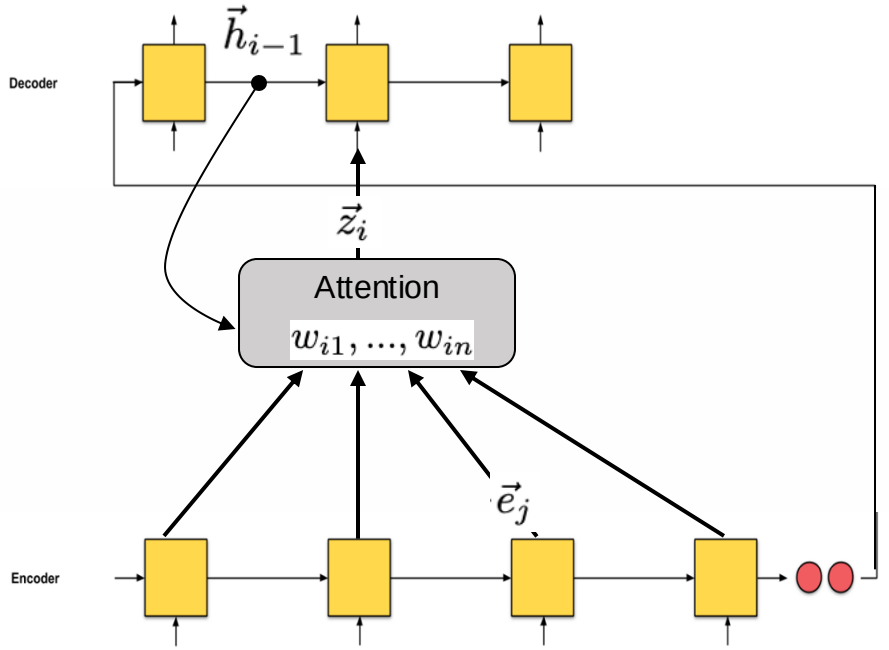
How does attention work?

Similarity is computed between **state of decoder** and **output embedding** for each term

$$w_{ij} = P(j|i) = \text{softmax}(similarity(\vec{h}_{i-1}, \vec{e}_j))$$

Soft-attention produces **weighted average** over input embeddings

$$\vec{z}_i = \sum_j w_{ij} \vec{e}_j$$



Source: 2015 paper by Bahdanau et al., https://arxiv.org/pdf/1409.0473.pdf

Soft-attention allows word reordering and more …

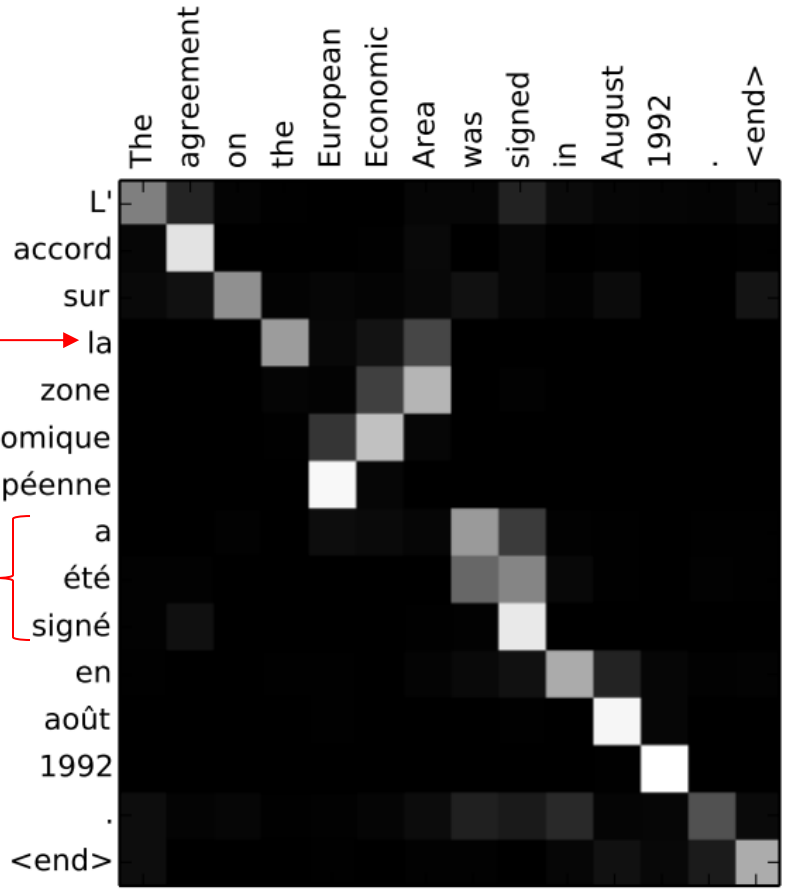
Soft-attention computes **weighted average**

- can **visualise weights** used to generate each output token
- e.g. when translating from English to French:

When translating the article “the”, need to know whether noun it refers to is masculine or feminine, hence attention placed also on word “Area”

When translating “European Economic Area”, word order needs to be reversed as shown by reverse diagonal attention

Verb in past simple passive form “was signed” is 2 words in English but becomes 3 words in French



From 2015 paper by Dzmitry Bahdanau, KyungHyun Cho and Yoshua Bengio “NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE” https://arxiv.org/pdf/1409.0473.pdf

Calculating similarity

How is the **similarity** between the state and embedding computed?

- In **additive attention**:
  compute similarity by **concatenating** decoder state and encoder embedding in a feedforward network

$$similarity(\vec{h}_{i-1}, \vec{e}_j) = \text{FFNN}(\vec{h}_{i-1}, \vec{e}_j)$$

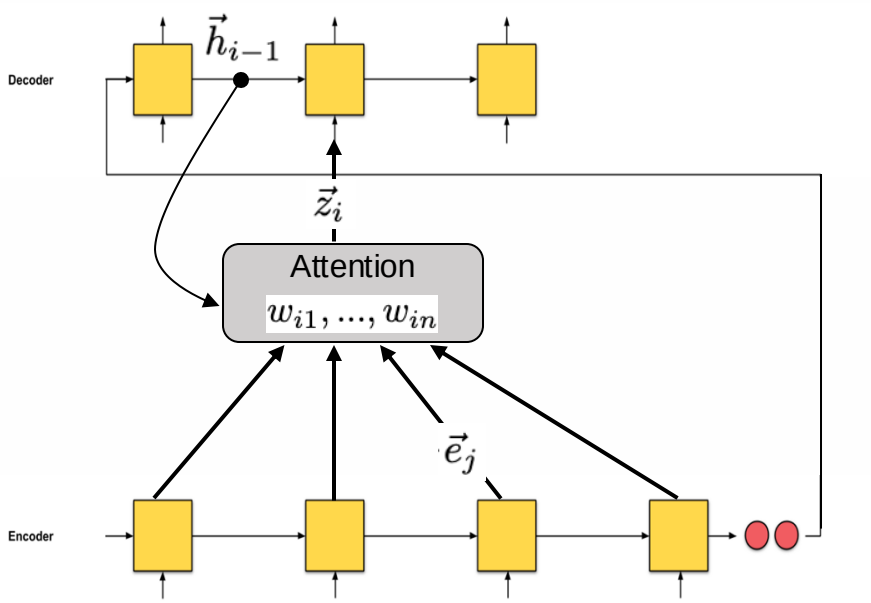
- In more common **multiplicative attention**:
  compute **dot product** between decoder state and encoder embedding

$$similarity(\vec{h}_{i-1}, \vec{e}_j) = \frac{\vec{h}_{i-1} \cdot \vec{e}_j}{\sqrt{d}}$$

  - Note: we divide by square root of embedding size d so that standard deviation remains 1:

$$if\ a_i, b_i \sim \mathcal{N}(0,1) => VAR[a_i b_i] = 1$$

$$=> VAR[(a_1, ..., a_d) \cdot (b_1, ..., b_d)] = \sum^{d} VAR[a_i b_i] = d$$



Source 2015 paper by Bahdanau et al.; https://arxiv.org/pdf/1409.0473.pdf

How does attention work?

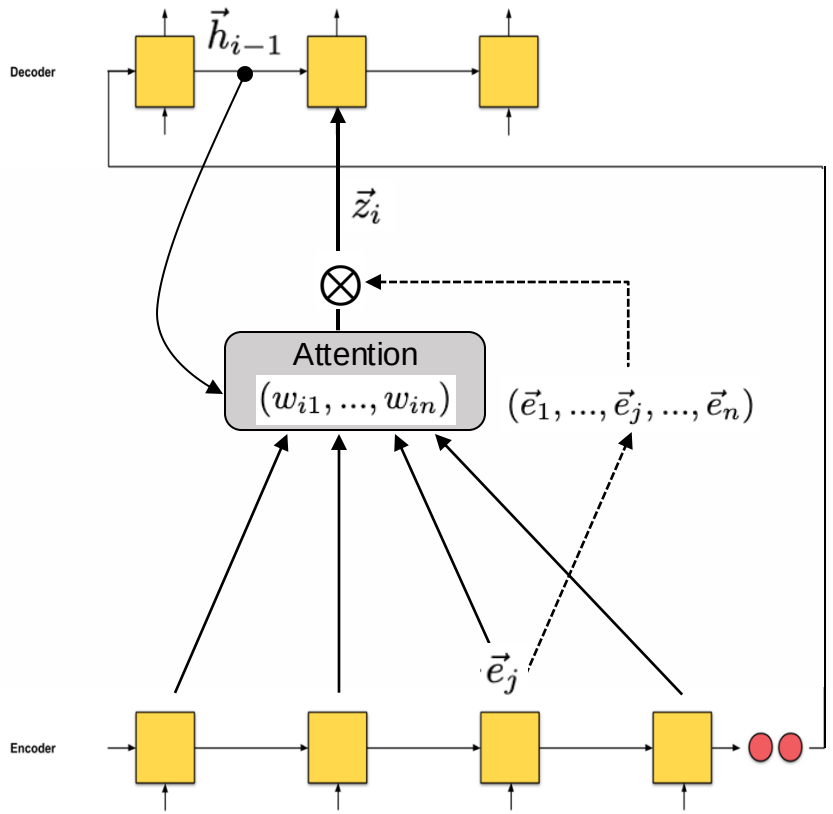
Once we have calculated the **similarity weights** at position i

- they are used to weight the **embeddings** from the encoder

$$\vec{z}_i = \sum_j w_{ij} \vec{e}_j$$

- in the case of multiplicative attention we have:

$$\vec{z}_i = \sum_j \text{softmax}(\frac{\vec{h}_{i-1} \cdot \vec{e}_j}{\sqrt{d}}) \vec{e}_j$$



Source: 2015 paper by Bahdanau et al., https://arxiv.org/pdf/1409.0473.pdf

Generalizing the notation

Multiplicative attention computes:

$$\vec{z}_i = \sum_j w_{ij}\vec{e}_j = \sum_j \text{softmax}(\frac{\vec{h}_{i-1} \cdot \vec{e}_j}{\sqrt{d}})\vec{e}_j$$

But we can then generalise the notation, to speak of:

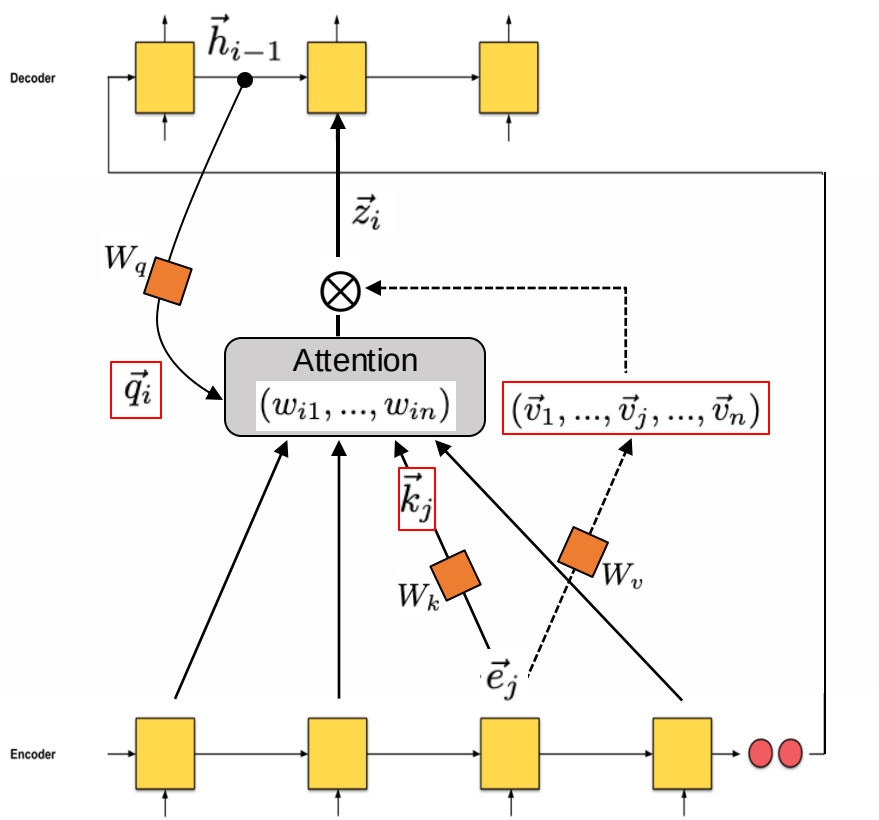
- **queries** (what is being looked up),
- **keys** (index of what to find), and
- **values** (stored information at key).

$$\vec{z}_i = \sum_j w_{ij}\vec{v}_j = \sum_j \text{softmax}(\frac{\vec{q}_i \cdot \vec{k}_j}{\sqrt{d}})\vec{v}_j$$

Where the query key and values are usually transformed by a linear transformation:

$$\vec{q}_i = W_q\vec{h}_{i-1} \qquad \vec{k}_j = W_k\vec{e}_j \qquad \vec{v}_j = W_v\vec{e}_j$$

$$W_q, W_k, W_v \in \mathbf{R}^{d \times d}$$

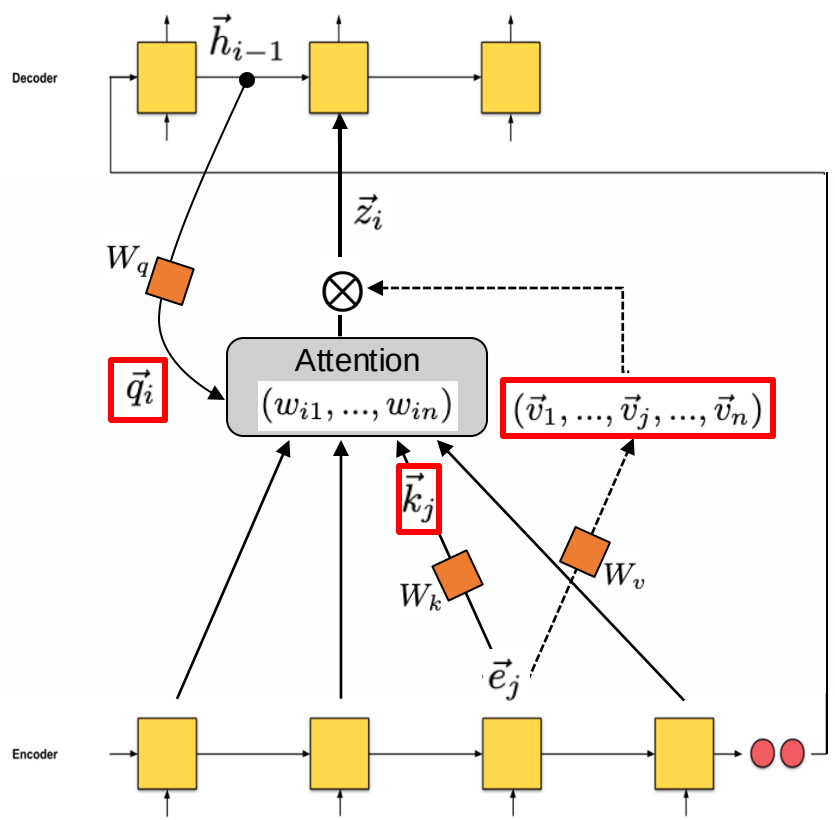


Source: 2015 paper by Bahdanau et al., https://arxiv.org/pdf/1409.0473.pdf

Interpreting queries, keys, values:

Hypothetical example:

- **query**: what is being looked up
  need *adjective that describes a person*
- **key**: index of what to find
  have *adjective that describes people & animals*
- ***value***: stored information at key
  word is *"friendly"*, which in Italian could be translated to *"simpatico"*



Source: 2015 paper by Bahdanau et al., https://arxiv.org/pdf/1409.0473.pdf

Aside: Deep Learning

What is deep learning?

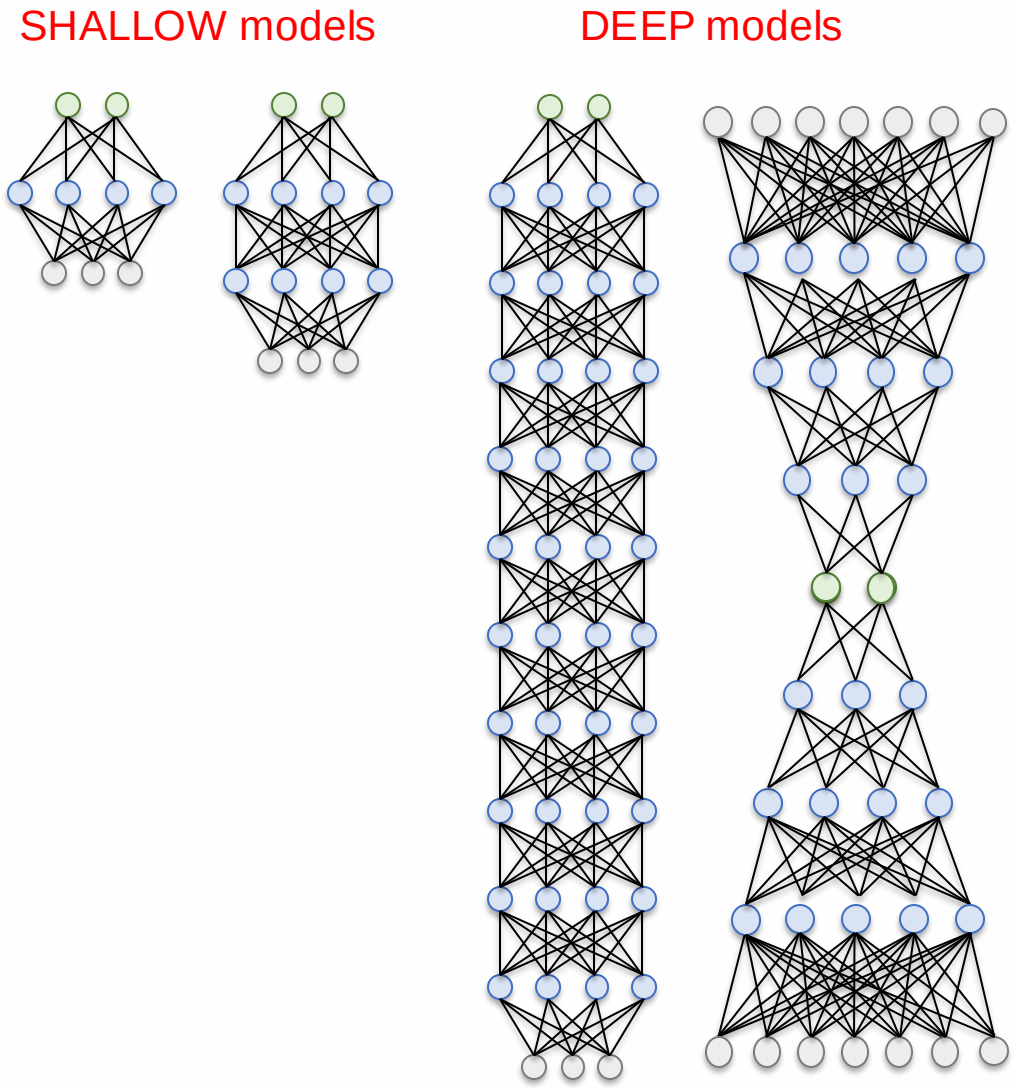
Deep Neural Networks are just NNs with **MANY** layers:

Why do we want many layers?

- massively **improves performance**
- by allowing network to learn **hierarchy of useful features** automatically

What are the downsides?

- need **lots of training** data!
- and large **computing resources** (GPUs)
- can be unstable & **harder** to train



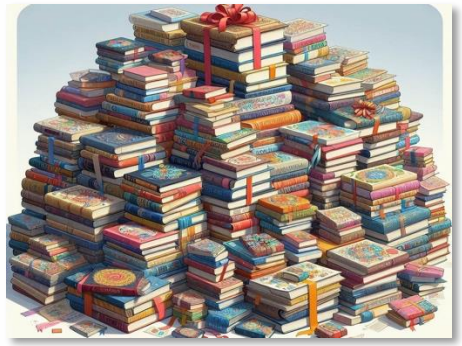
Why is Deep Learning so popular?

Massive popularity due to:

- **amazing performance** on many tasks, especially with **complicated raw data** types like images and **text**
- greatly **simplifies training pipeline**: little or no need for pre-processing
- allows for **transfer learning** where we **reuse knowledge** from previously trained systems

Possible due to:

- **hardware advancements**:
  graphics processing units (GPUs) for fast matrix multiplication
- **huge amounts of data**
  available to learn features for complex tasks like image recognition
- **clever architectures**
  e.g. convolutional, recurrent and self-attention networks
- **improved training procedures**
  fast gradient descent routines, batch normalisation, dropout, etc.
- **toolkits**
  like PyTorch/TensorFlow that perform **automated differentiation**



"massive pile of books" image source:

$$\nabla Loss(\theta) = (\frac{\partial Loss(\theta)}{\partial \theta_1}, ..., \frac{\partial Loss(\theta)}{\partial \theta_m})$$

Why is deep learning important for text?

State-of-the-art performance for most **text processing** tasks

- including **classification, summarisation, generation, translation,** etc.

Until 2017, deep architectures involved stacking many layers of LSTMs on top of each other (e.g. ELMO)

In 2017, a new architecture, called a Transformer, emerged

- in the paper: "*Attention Is All You Need*" https://arxiv.org/abs/1706.03762
- makes use of a **stack** of **self-attention** networks
- more on that in a minute …

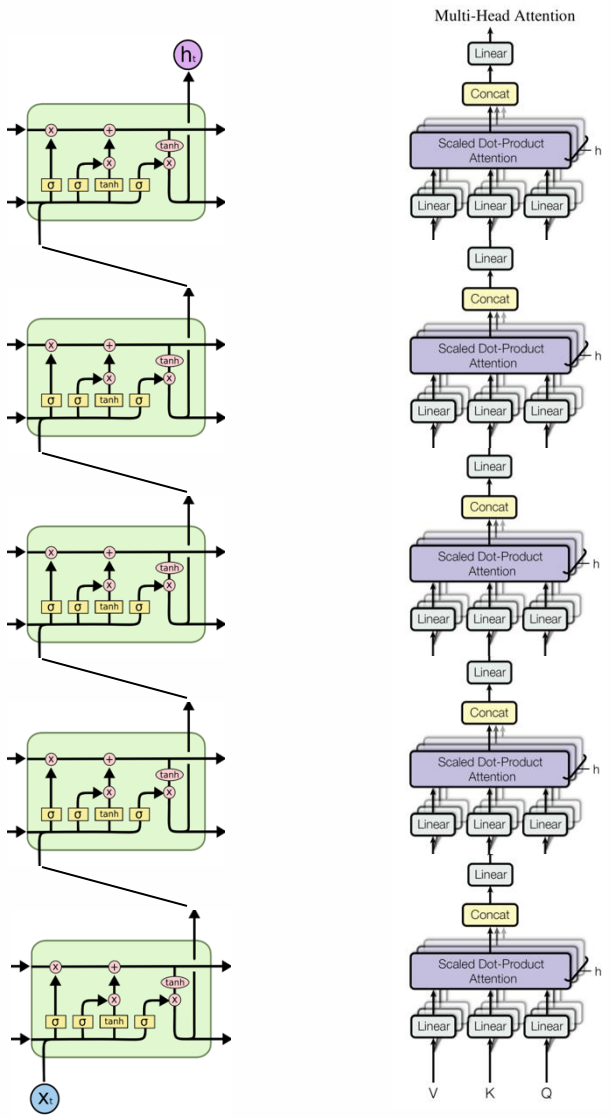


Image Source: "Attention Is All You Need" by Vaswani et al.
Images source: http://jrmeyer.github.io/nlp/2017/08/Understanding-LSTMs/

Motivating self-attention:
slow training of RNNs

Training costs of RNNs

Lessons from deep learning

- **deeper models work better** than shallow ones
- since each layer builds on simpler features from layer below

Problem with training RNNs:

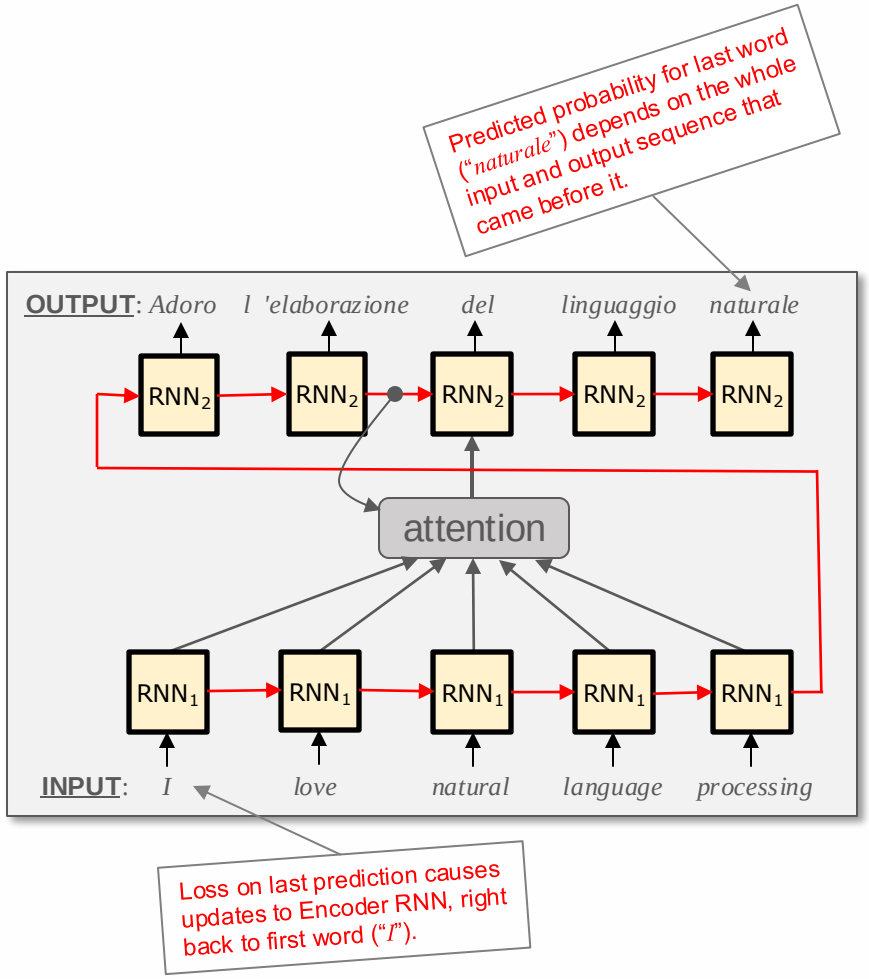
- when generating, info must propagate from **first position of encoder** to **last position of decoder**
- & gradient information must pass back along entire sequence to update parameters

Calculation must be done sequentially

- cannot be parallelised
- training time linear in length of text, O(*n*)
- difficult to learn **deeper networks** with many layers

But during training:

- already know desired output and have attention mechanism passing information directly from input to output
- so can't something be done to **speed up the training**?



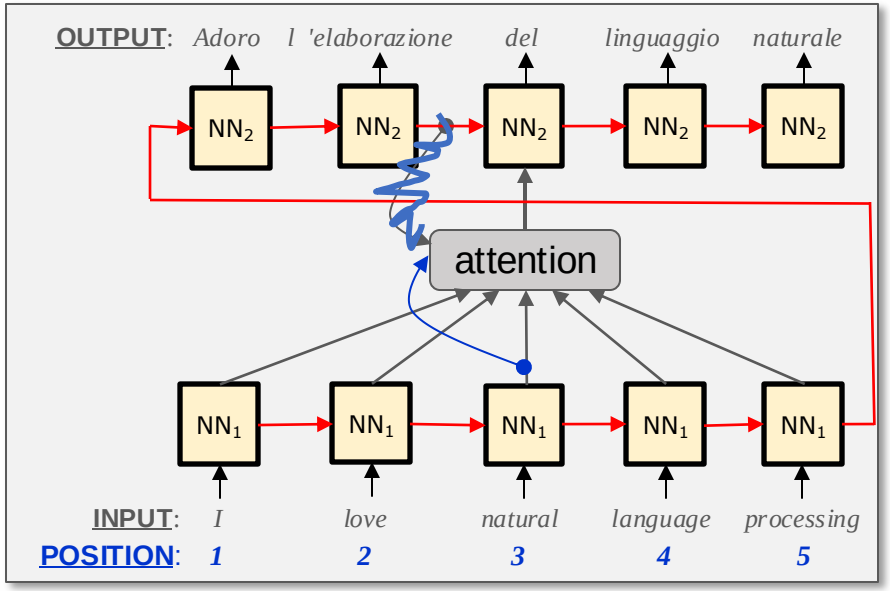
Removing the RNN

During **training**:

- already know **entire output** want to see, so no need to generate it one term at a time
- have attention mechanism that **passes information directly** from input to output

CRAZY IDEA:

- What happens if we **remove recurrent links** from the encoder and decoder?
- just have a NN encoder/decoder without the recurrent part …



Creates two problems:

1. what should use for the query?
2. lose all information about order of words

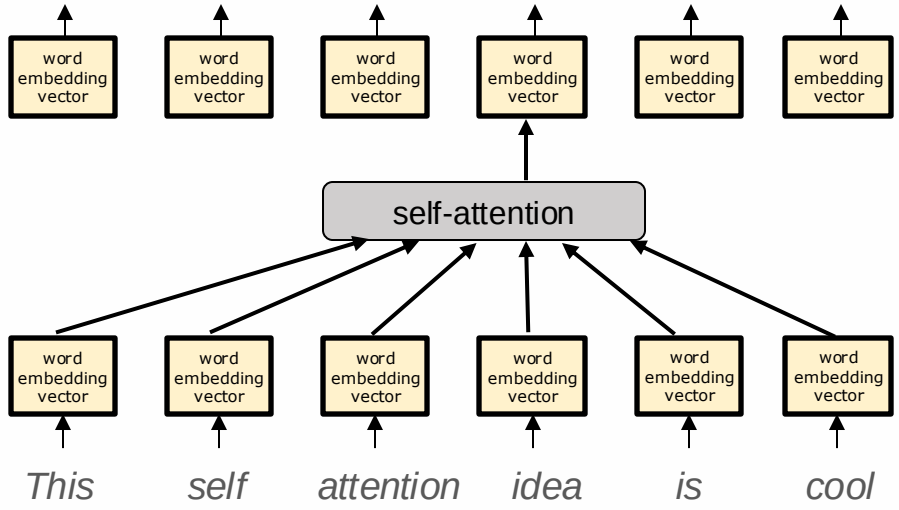
Possible solutions:

1. use **current output of encoder as query** rather than context of decoder
   - but what would that even mean?
2. add **position information directly to the data** provided of the encoder
   - e.g. let the model learn an embedding to represent position 1, 2, etc.

So what is self-attention?

Self attention is a mechanism for:

- **combining** word **embedding vectors** to **produce new** word **embedding vectors**
- each high-level embedding is **weighted average** of word embeddings below it



Weights are computed:

- based on **similarity between embeddings** in respective positions
- model parameters control the process, learning how best to compute the weights

So what is self-attention - exactly?

Self-attention is a mechanism for:

- **updating** a sequence of **embedding vectors** based on the **weighted average** of incoming embedding vectors

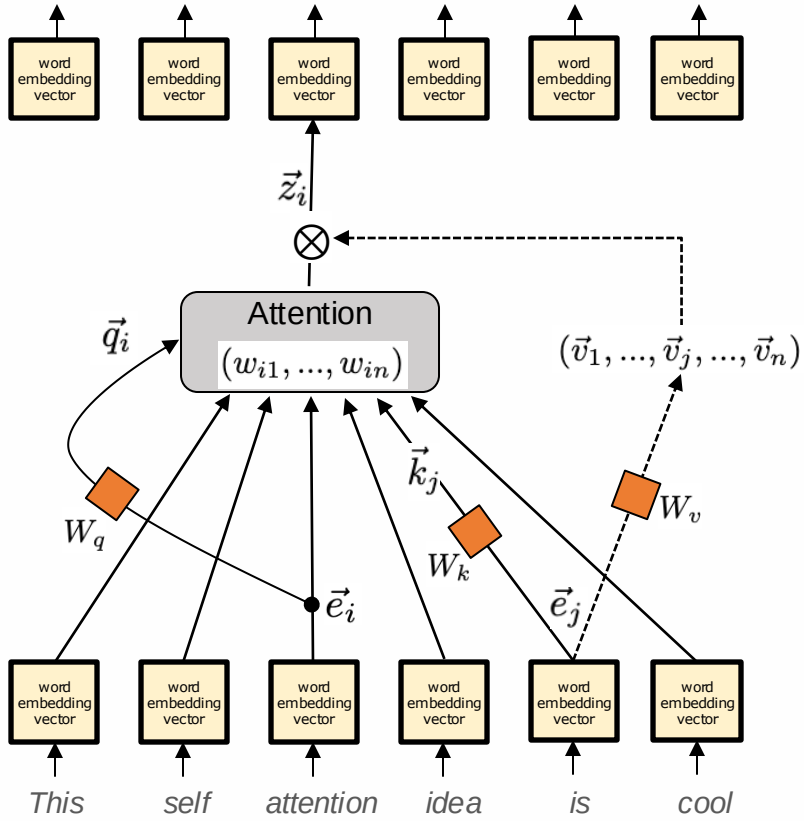
$$\vec{z}_i = \sum_j w_{ij}\vec{v}_j = \sum_j \text{softmax}(\frac{\vec{q}_i \cdot \vec{k}_j}{\sqrt{d}})\vec{v}_j$$

- where the **weights** depend on the **similarity between embeddings** in respective positions

More specifically, at each position i, compute:

- **query:** linear transform of *embedding at position i*
- **key:** linear transform of *embedding at position j*
- **value:** linear transform of *embedding at position j*

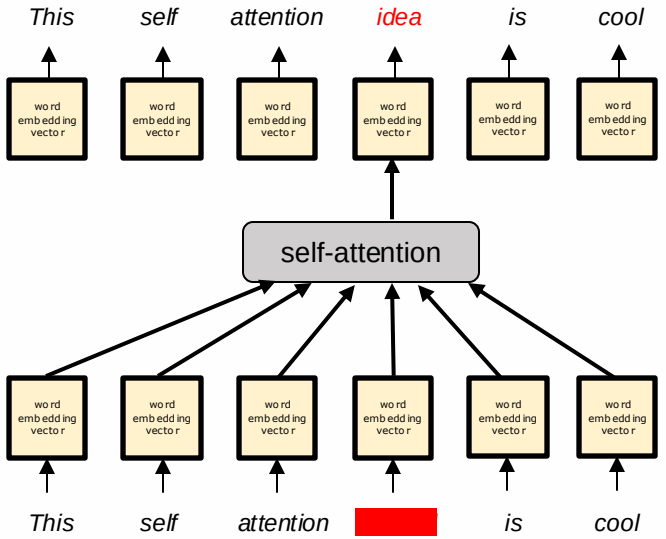
$$\vec{q}_i = W_q\vec{e}_i$$
$$\vec{k}_j = W_k\vec{e}_j$$
$$\vec{v}_j = W_v\vec{e}_j$$
$$W_q, W_k, W_v \in \mathbf{R}^{d\times d}$$



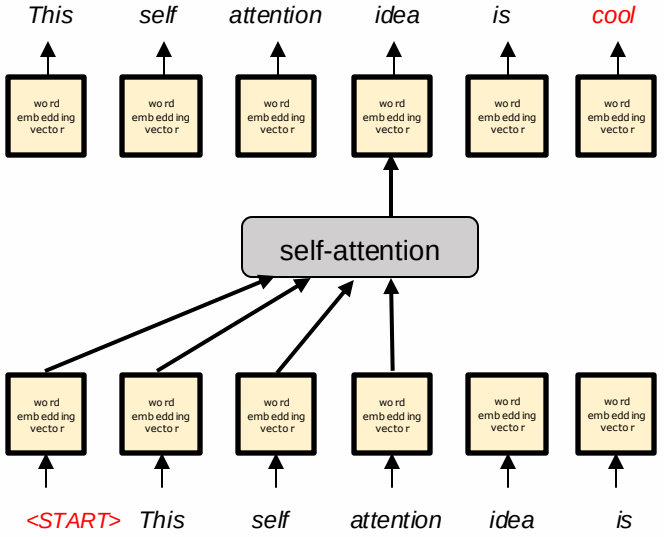
What is it trained to do?

Self-attention models are trained to either:

1. **recover missing words** from the input text based on the surrounding context
   - i.e. to perform the Masked Language Modelling (MLM) task,
   - where input text has been corrupted by masking randomly certain tokens



2. **predict** the **next word** from the **previous words** in text
   - i.e. generated text



Transformer

Original Transformer

Original Transformer from 2017 (“Attention is all you need”) paper

- was brilliant
- looked rather complicated because it contained both an **encoder** and a **decoder**

Internal architecture is actually surprisingly simple:

- basic self-attention module contains:
  - **multiple attention heads** working in parallel
  - **feedforward network**, with **residual connections** and **normalisation**
- architecture is word position agnostic
  - so **positional encoding** provided as additional input to bottom layer

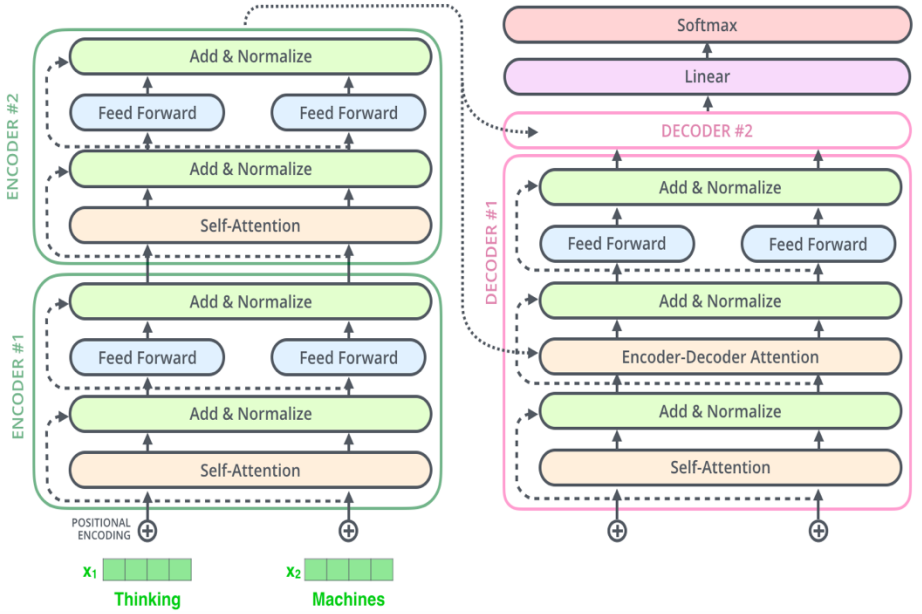


Image Source: http://jalammar.github.io/illustrated-transformer/

Inside Transformer

Basic self-attention module is stacked on itself **many** times

- allows **semantics of each token** to build up over multiple steps

Note: transformers are MUCH **faster to train** than stacks of RNNs

- in RNNs gradient must be iterated back along sequence

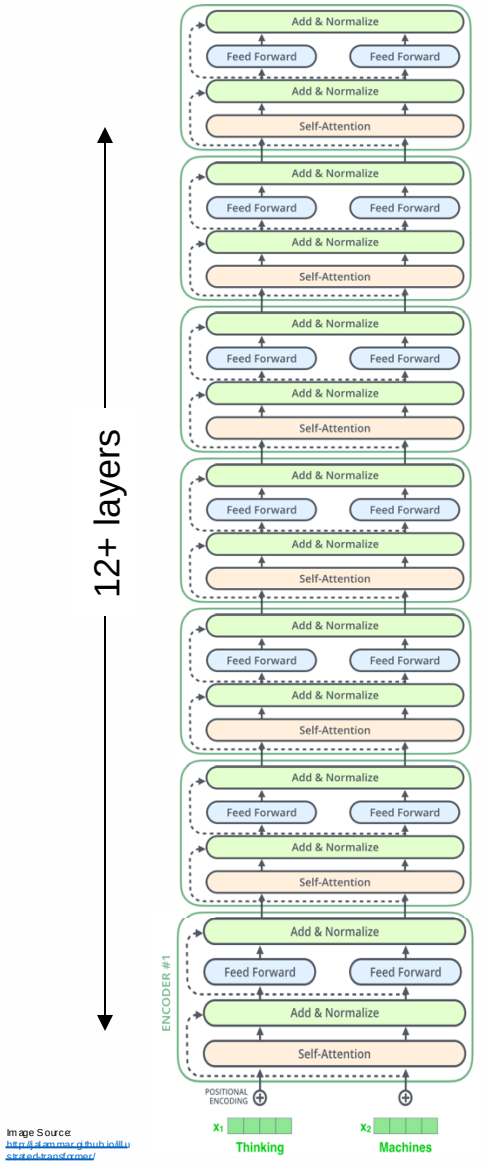


Image Source:
http://jalammar.github.io/illustrated-transformer/

Why does self-attention make sense?

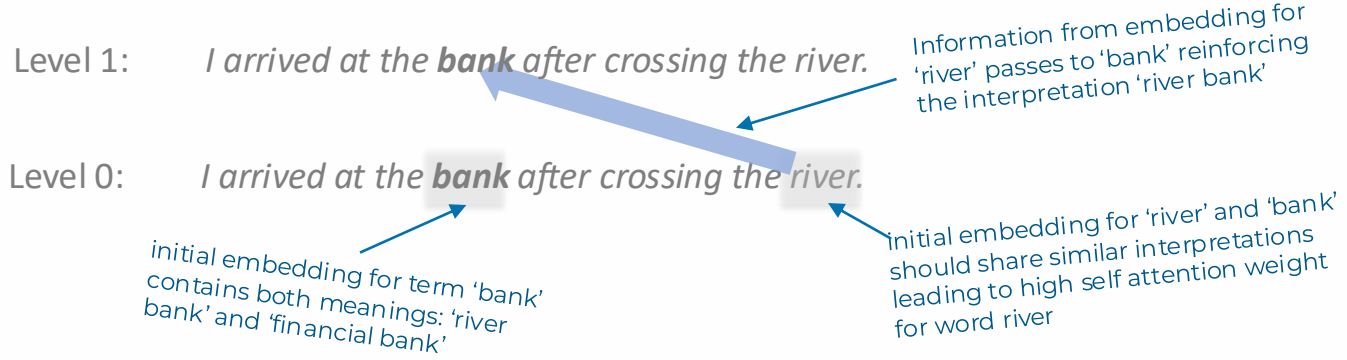
Motivating self-attention

Sentence 1: *I arrived at the **bank** after crossing the street.*
Sentence 2: *I arrived at the **bank** after crossing the river.*

What type of bank are we talking about here, a river bank or a financial bank?

To understand why self-attention is useful for language models, consider that

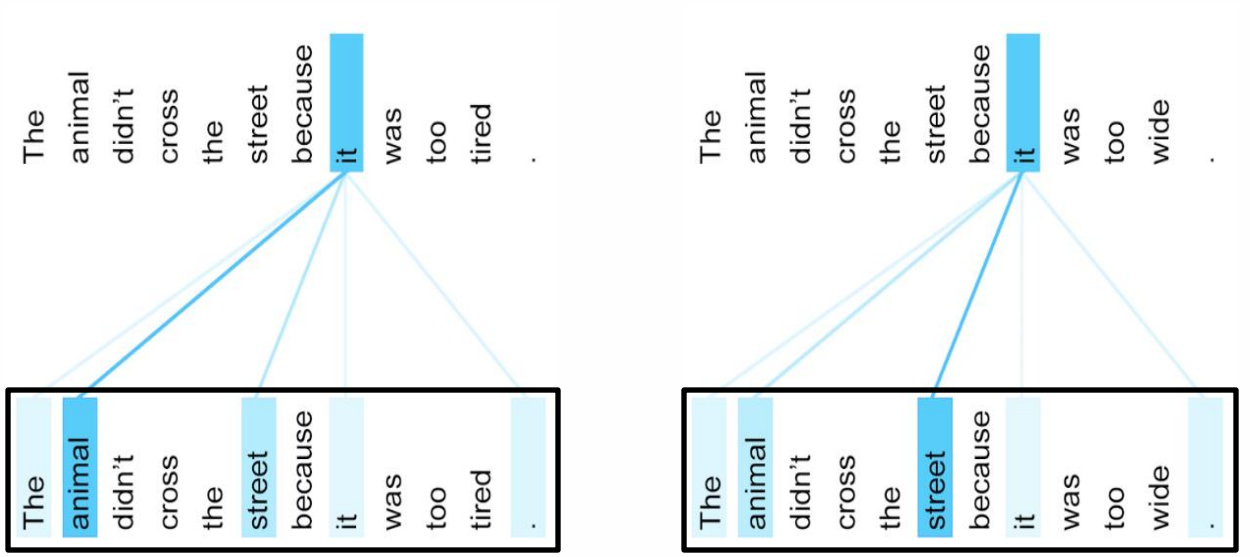
- words take on **different meanings** depending on their context
- attention mechanism allows representation to **depend on context**
- learns **weighting function** over lower-level embeddings of context terms



Motivating self-attention (cont.)

Complicated tasks like **coreference resolution** can be handled quite effectively with multiple layers of self-attention

- Consider what “it” refers to in the following two sentences:



- Note that more attention is placed on “animal” in first sentence and “street” in second
- How is this possible?
  - assume that lower embedding for “it” on left already contains information from “tired”
  - then similarity with word “animal” should be higher than “street”

Image source: “Transformer: A Novel Neural Network Architecture for Language Understanding”, by Uszkoreit et al. https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html

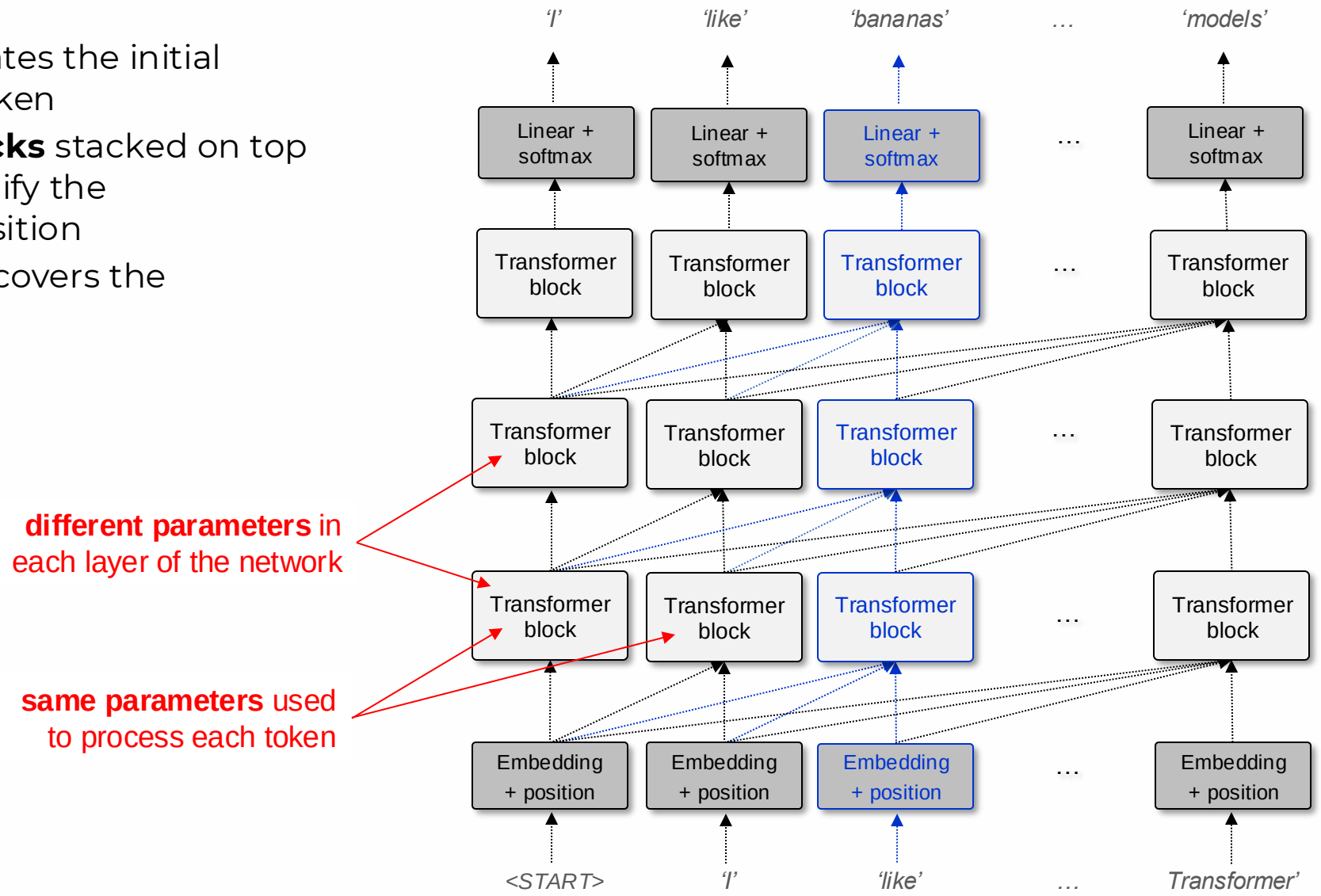
Transformer architecture

- structure of model
- implementation via matrix multiplication

The Transformer

Transformer consists of:

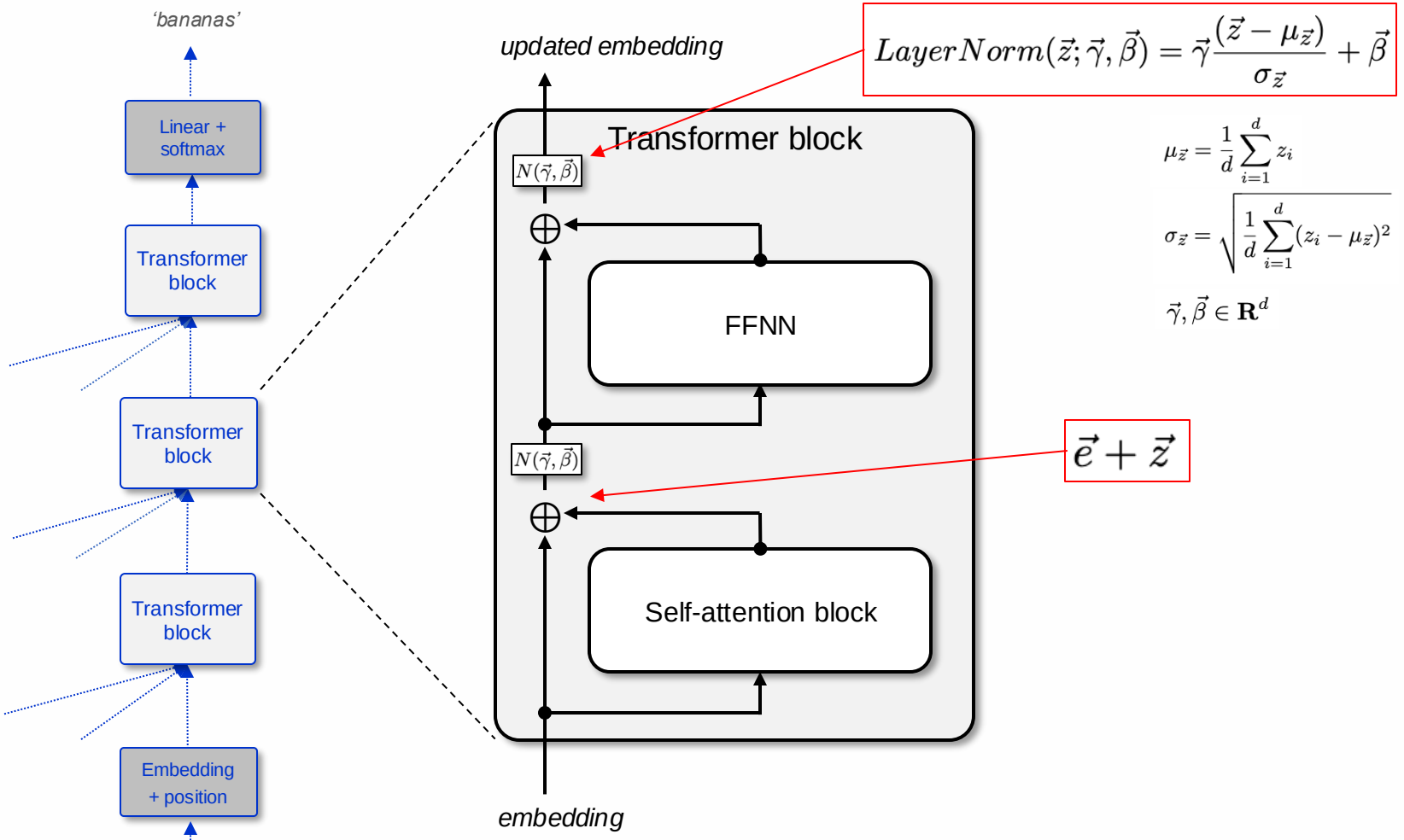
- **input module** that creates the initial embedding for each token
- many **transformer blocks** stacked on top of each other, that modify the embedding at each position
- **output module** that recovers the predicted word



The Transformer block

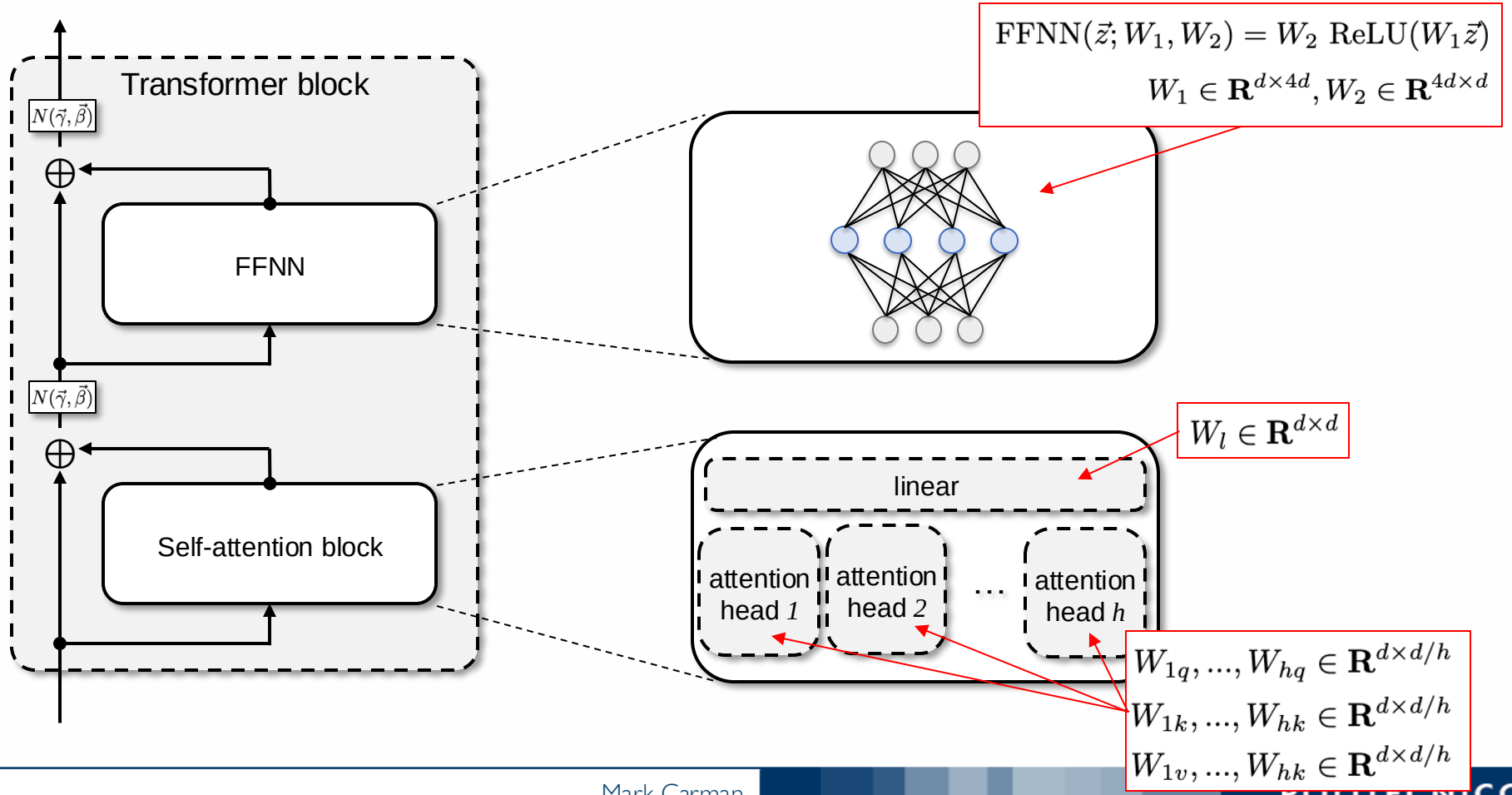
Each Transformer block modifies the incoming embedding

- by **adding** information from a **self-attention block** and a **Feed-Forward Neural Network** (FFNN)



Inside the Transformer block

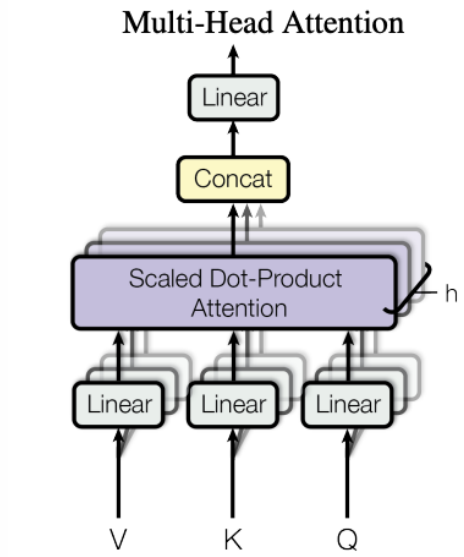
- Inside the **self-attention block** are multiple self-attention heads
  - Each working with embedding of size d/h
- Inside the **Feed-Forward Neural Network** (FFNN)
  - is just a single layer with fan-out of 4



Scaled dot-product attention

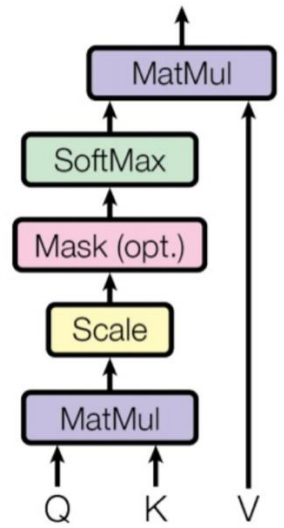
Three components of self-attention:

- **query** → what we are looking up
  - embedding of the word in the current position
- **key** → what we are looking for
  - other word in the series after a linear transformation
- **value** → what will be updated
  - the embedding of the word that will be used to update



Each of these components produced by a linear mapping (reduction) on the original embedding

- $d_k = d$ / *#number of parallel attention heads*



Q, K and V are all matrices, allowing attention weights for all query positions to be calculated with a single matrix product

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

If vectors Q and K both contain standardised normally distributed values (with zero mean and unit variance), then the dot product will be zero mean, but with a standard deviation of $\sqrt{d_k}$, so divide by this value to normalise the similarity

Self-attention via Matrix Multiplications

$$Q = \begin{pmatrix} - & \vec{q}_0 & - \\ - & \vec{q}_1 & - \\ - & \vdots & - \\ - & \vec{q}_m & - \end{pmatrix} \qquad K^T = \begin{pmatrix} | & | & & | \\ \vec{k}_0 & \vec{k}_1 & \cdots & \vec{k}_n \\ | & | & & | \end{pmatrix}$$

Calculations for a single layer of self-attention

- is just a series of matrix multiplications/manipulations
- in which **all token positions** are **updated in parallel**
- and all attention heads are computed at same time

$$QK^T = \begin{pmatrix} \vec{q}_0 \cdot \vec{k}_0 & \vec{q}_0 \cdot \vec{k}_1 & \cdots & \vec{q}_0 \cdot \vec{k}_n \\ \vec{q}_1 \cdot \vec{k}_0 & \vec{q}_1 \cdot \vec{k}_1 & \cdots & \vec{q}_1 \cdot \vec{k}_n \\ \vdots & \vdots & \ddots & \vdots \\ \vec{q}_m \cdot \vec{k}_0 & \vec{q}_m \cdot \vec{k}_1 & \cdots & \vec{q}_m \cdot \vec{k}_n \end{pmatrix}$$

Steps:

1. first compute **Query, Key** and **Value matrices**
2. **multiply Query and Key** matrices
   for autoregressive model, multiply also by mask matrix so future tokens have similarity of zero
3. **apply softmax** to rows of resulting matrix
4. **multiply attention weights** by Value matrix
5. for multi-head attention, **concatenate** each head output **and multiply by output matrix**

$$\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) = \begin{pmatrix} \text{softmax}(\frac{1}{\sqrt{d_k}}\langle \vec{q}_0 \cdot \vec{k}_0, \vec{q}_0 \cdot \vec{k}_1, \ldots, \vec{q}_0 \cdot \vec{k}_n \rangle) \\ \text{softmax}(\frac{1}{\sqrt{d_k}}\langle \vec{q}_1 \cdot \vec{k}_0, \vec{q}_1 \cdot \vec{k}_1, \ldots, \vec{q}_1 \cdot \vec{k}_n \rangle) \\ \vdots \\ \text{softmax}(\frac{1}{\sqrt{d_k}}\langle \vec{q}_m \cdot \vec{k}_0, \vec{q}_m \cdot \vec{k}_1, \ldots, \vec{q}_m \cdot \vec{k}_n \rangle) \end{pmatrix} = \begin{pmatrix} s_{0,0} & s_{0,1} & \cdots & s_{0,n} \\ s_{1,0} & s_{1,1} & \cdots & s_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ s_{m,0} & s_{m,1} & \cdots & s_{m,n} \end{pmatrix}$$

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V = \begin{pmatrix} s_{0,0} & s_{0,1} & \cdots & s_{0,n} \\ s_{1,0} & s_{1,1} & \cdots & s_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ s_{m,0} & s_{m,1} & \cdots & s_{m,n} \end{pmatrix} \begin{pmatrix} - & \vec{v}_0 & - \\ - & \vec{v}_1 & - \\ - & \vdots & - \\ - & \vec{v}_n & - \end{pmatrix}$$

For more information see:
http://www.columbia.edu/~jsl2239/transformers.html

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

Video

Excellent video showing the computations being performed inside the transformer

- **Attention in transformers, visually explained | Chapter 6, Deep Learning (Grant Sanderson)**
  https://www.youtube.com/watch?v=eMlx5fFNoYc

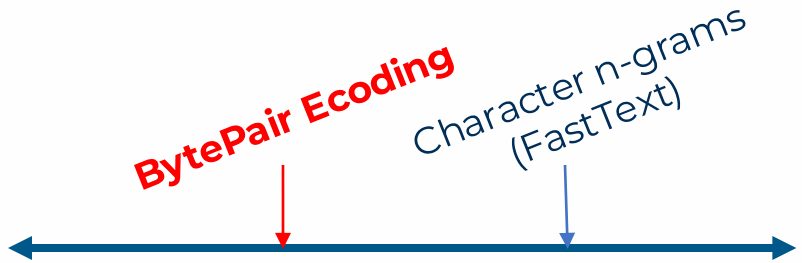
Generating Input for Transformer

- Tokenization
- Positional embedding

Tokenization: word or character-level?

Is it better to use **word-level** or **character-level representations** in the model?

- choice depends on computational trade-offs:
  - expressivity of sequence vs sequence length (inference complexity)
- and also what the language is:
  - Chinese has logograms at syllable rather than character level
  - DNA/Protein sequences have few characters and no spaces



Words as tokens

- Much **larger number of tokens**:
  e.g. 1 million for English Twitter, more for multi-lingual text
- **No use** or explicit modelling **of morphology**:
  $dist^{(0)}$('apple','apples') = $dist^{(0)}$('apple','elephant')

Characters as tokens

- **Longer sequence** length
- Burdon on model to learn word structure, with embeddings containing lots of disjunctive information
- Much **less interpretable**

Transformers use sub-word tokens

How should we break up words into sub-word tokens? Use the data!

- find **frequent character sequences** by performing a **byte-pair encoding**
- iteratively replace most frequent consecutive characters by new characters

```
though they think that the thesis is thorough enough
th → θ:
        θough θey θink θat θe θesis is θorough enough
ou+g+h → ə:
        θə θey θink θat θe θesis is θorə enə
θe → ψ:
        θə ψy θink θat θe ψsis is θorə enə
```

- in this way, common prefixes/suffixes become vocabulary elements:

Input sentence:	*'I like playing football.'*
Word level tokenization:	*'I', 'like', 'playing', 'football', '.'*
Sub-word level tokenization:	*'I', 'like', 'play', '#ing', 'foot', '#ball', '.'*

Embedding Positional Information

Need for positional information

Transformer architecture is symmetric with respect to the input features

- and thus agnostic with respect to their ordering
- like training classifier on table with **reordered features** => learns **same model**

$min_\vartheta\ loss($

Feature 1: body-mass-index	Feature 2: heart-rate	Feature 3: sys. blood pressure	...	Class: risk
21.2	111	132	...	low
31.7	84	112	...	high
...	...	...		...
27.8	79	145	...	high

$;\vartheta) \quad = \quad min_\vartheta\ loss($

Feature 1: sys. blood pressure	Feature 2: body-mass-index	Feature 3: heart-rate	...	Class: risk
132	21.2	111	...	low
112	31.7	84	...	high
...	...	...		...
145	27.8	79	...	high

$;\vartheta)$

- but the **meaning of text** depends on the **word order**!

"there's a rat in the white house" ≠ "there's a white rat in the house"

- so we need to inform the model about the order of words in the text:

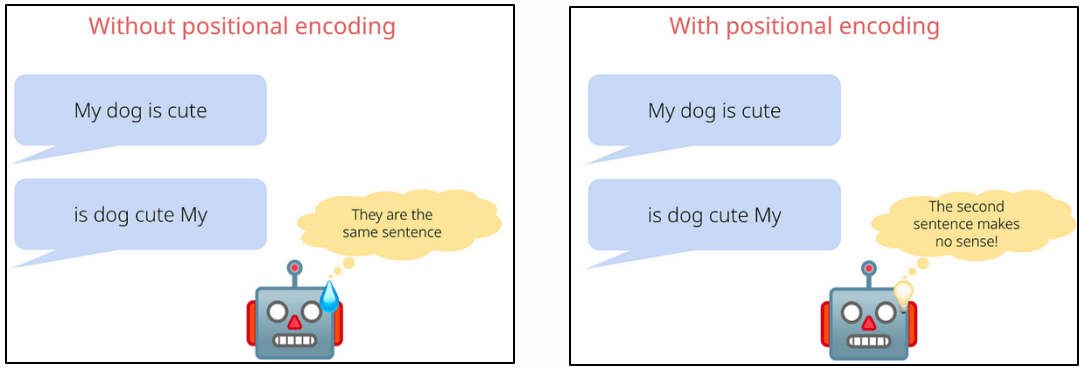


Image source:
https://pub.towardsai.net/the-quest-to-have-endless-conversations-with-llama-and-chatgpt-%EF%B8%8F-81360b9b34b2

Positional Encoding using

We need to inform the model about the order of words in the text

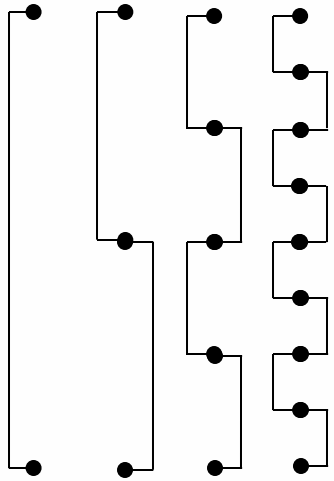
- "I want my mommy" => ("I", 1) ("want", 2), ("my", 3), ("mommy", 4)
- Embedding["my",3] = embedding["my"] + embedding[3]

Simplest way to do that would be to use a binary encoding of the position

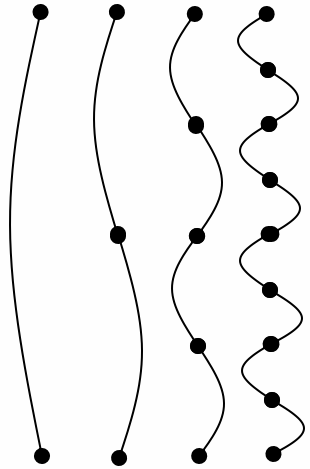
- Since the embedding vector is made of **floating point** values, makes more sense to encode positions using sinusoids …

```
E(0) = 00000
E(1) = 00001
E(2) = 00010
E(3) = 00011
E(4) = 00100
E(5) = 00101
E(6) = 00110
E(7) = 00111
```

=



=>



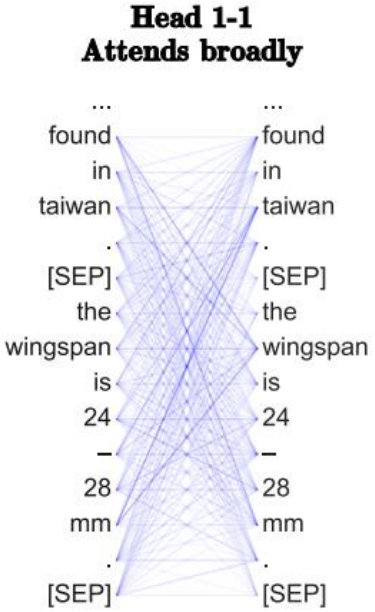
- Embedding[3] = (0, 0, 0, 1, 1) => (0.098, 0.195, 0.382, 0.707, 1)

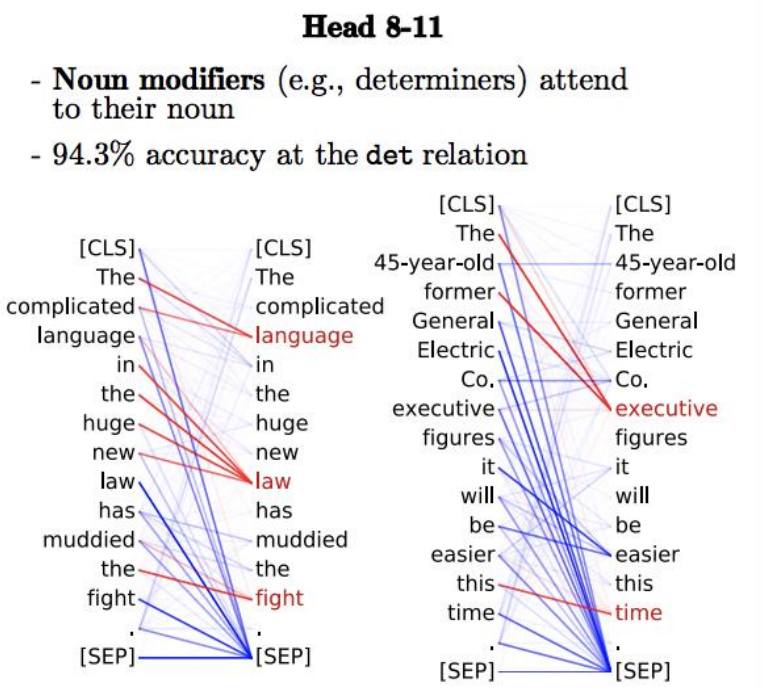
Why does it work so well?

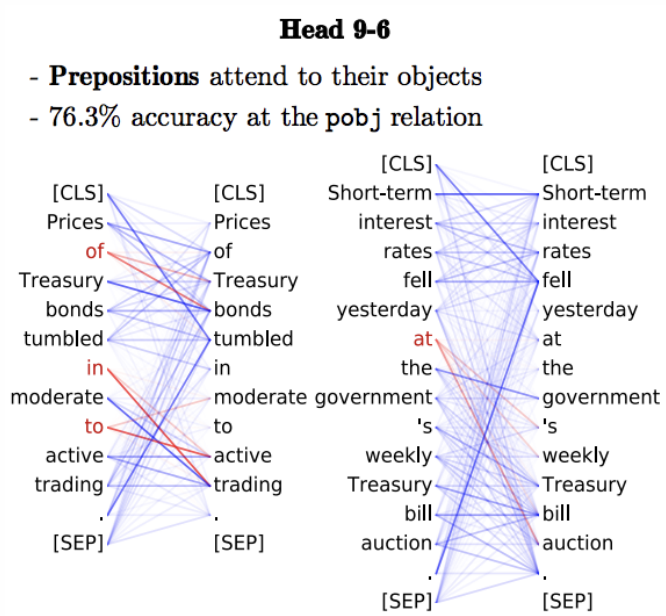
What is stacked self-attention learning?

Lots of visualisation is going on trying to interpret what is being learnt.

- some heads simply *aggregate information* or *attend to a previous token*
- others *learn language relationships* (2019 paper by Clark et al.)
- see demo: https://colab.research.google.com/drive/1PEHWRHrvxQvYr9NFRC-E_fr3xDq1htCj







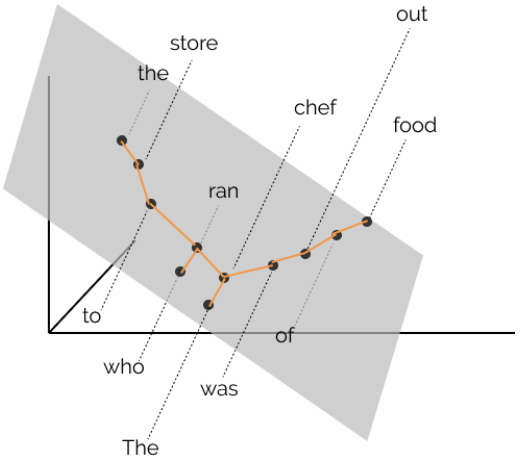
Source: https://arxiv.org/pdf/1906.04341.pdf

Why is stacked self-attention so useful?

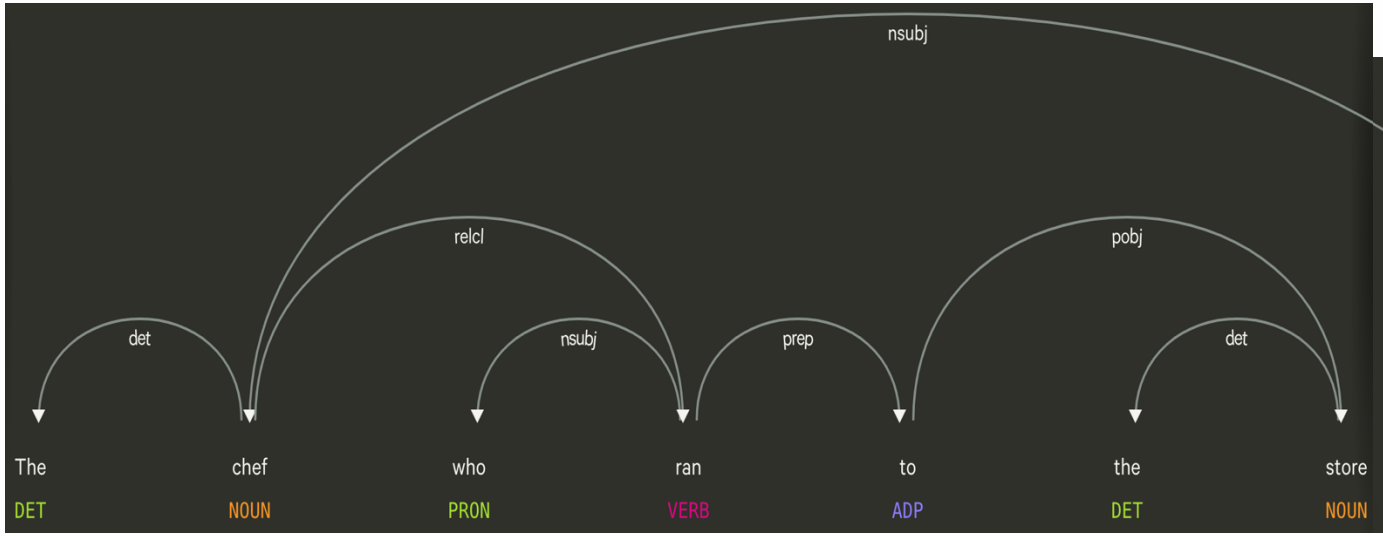
It's been argued that Transformers **effectively learn** how to build a **dependency parse tree** over concepts in the text

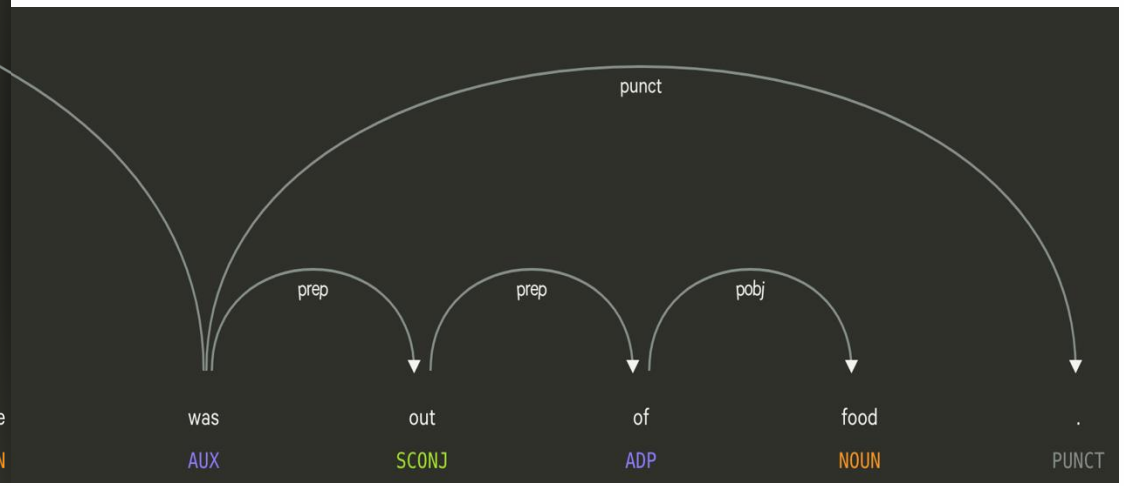
Hewitt et al. 2019 https://nlp.stanford.edu/pubs/hewitt2019structural.pdf

- Consider the examples:
  - The store was out of food. ___
  - The chef who ran to the store was out of food. ___
- To predict next sentence, need to who is out of food
  - was it the store or the chef?



Source: https://nlp.stanford.edu//~johnhew//structural-probe.html





Source: https://explosion.ai/demos/displacy?text=The%20chef%20who%20ran%20to%20the%20store%20was%20out%20of%20food.&model=en_core_web_sm&cpu=0&cph=0

But what does the FFNN do?

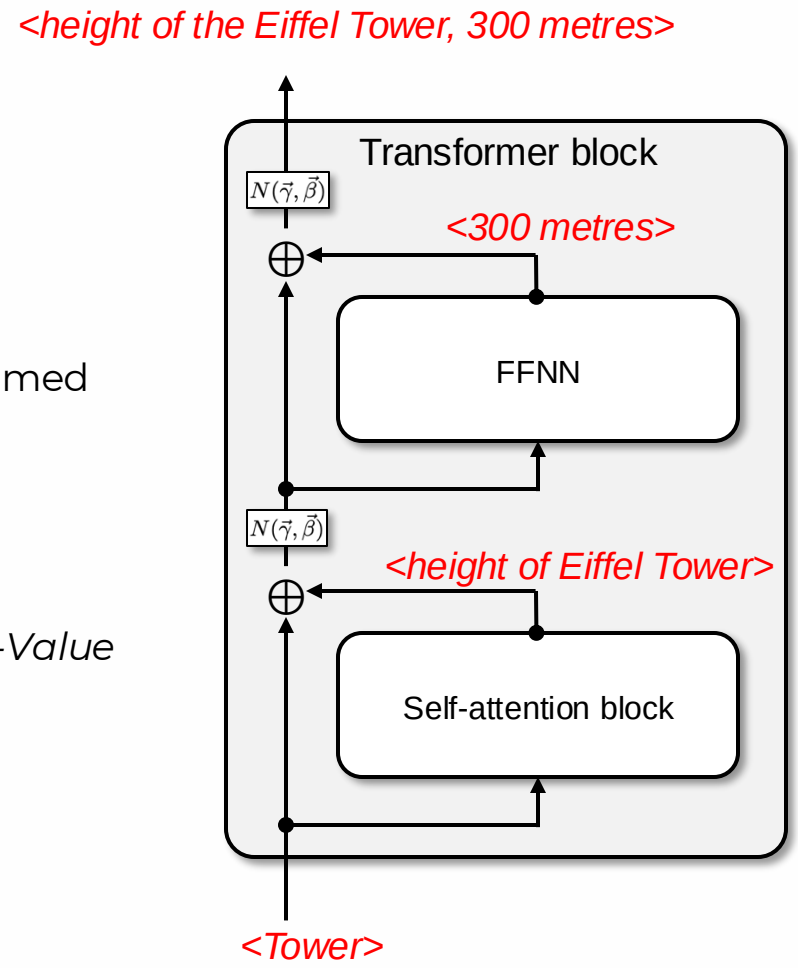
The two blocks within the Transformer provide complimentary functionality:

Self-attention **builds patterns** from input text

- E.g. identifies noun phrases that may correspond to named entities:
- "The Eiffel Tower", "fastest animal on earth", ...

Feed-forward layer **looks up facts** for entities

- See paper *"Transformer Feed-Forward Layers Are Key-Value Memories"* https://arxiv.org/abs/2012.14913



BERT vs GPT

Story of two architectures:

Original Transformer from 2017 paper "Attention is all you need"

- was **designed for translation** tasks
- brilliant but complicated, containing both **encoder** and **decoder**
- encoder and decoder are independently useful, so subsequent models mainly implemented only one or other:

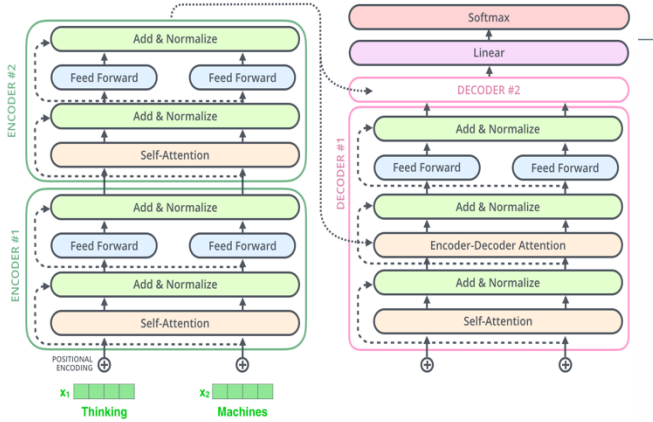
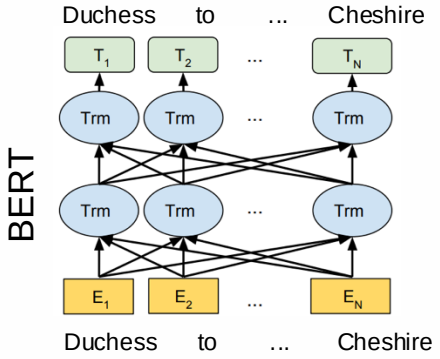


Image Source: http://jalammar.github.io/illustrated-transformer/

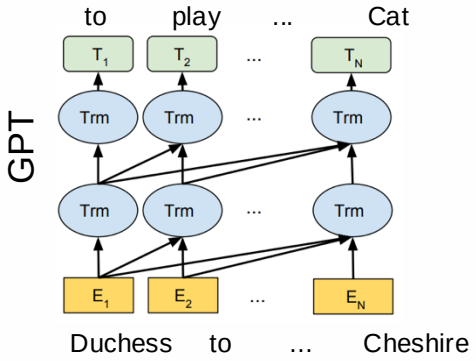
BERT = **B**idirectional **E**ncoder **R**epresentations from **T**ransformers

- 2019 paper by Devlin et al. (Google)
- **autoencoder**: recovers (potentially corrected) input at top of each column
- learnt by **masking** random tokens and recovering them on output
- great for **representing text** (e.g. for building classifiers)



GPT = **G**enerative **P**retrained **T**ransformer

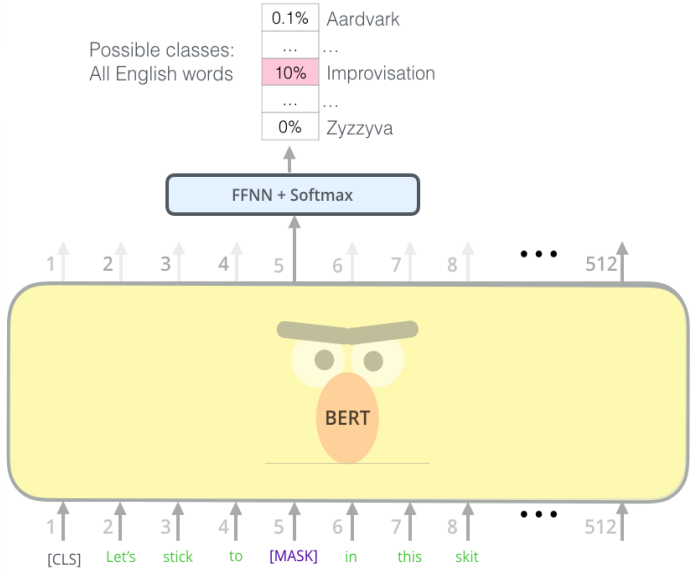
- GPT-2, 2019 paper by Radford et al. (OpenAI)
- **autoregressive**: predicts next token at top of each column
- learnt by **masking** future tokens
- great for **generating text**



Pretraining of BERT and GPT

BERT:

- by **masking out** random words in the input using a special [MASK] token
- model must recover all words including the masked ones
- **trained on** Wikipedia and a corpus of books



Source: http://jalammar.github.io/illustrated-bert/

WIKIPEDIA

Image source: https://en.wikipedia.org/wiki/File:Wikipedia-logo-v2.svg

GPT-2

- by simply **masking future words** in the sequence and at each point predicting the next word
- **trained on** 40GB of web text that Reddit users rated highly
  - note: model will produce similar text to that which it was trained on (so garbage in => garbage out)
  - so important to pre-train it on good quality text

reddit

Image source: https://en.wikipedia.org/wiki/Reddit

Transformer Sizes & # of Parameters

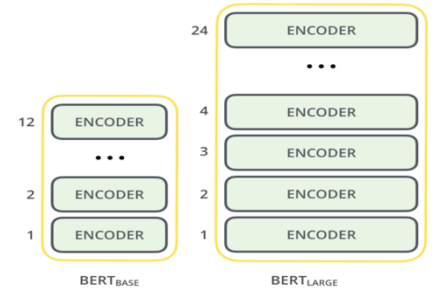
Transformers come in multiple sizes, depending on:

- number of self-attention layers
- size of the embedding used at each layer
- number of parallel attention heads
- typical sizing for BERT models:
  - base model has 110M parameters, while large model has 340M
- typical sizing for the GPT-2 models:
  - largest has **1.5 billion** parameters and a vocabulary of 50,257

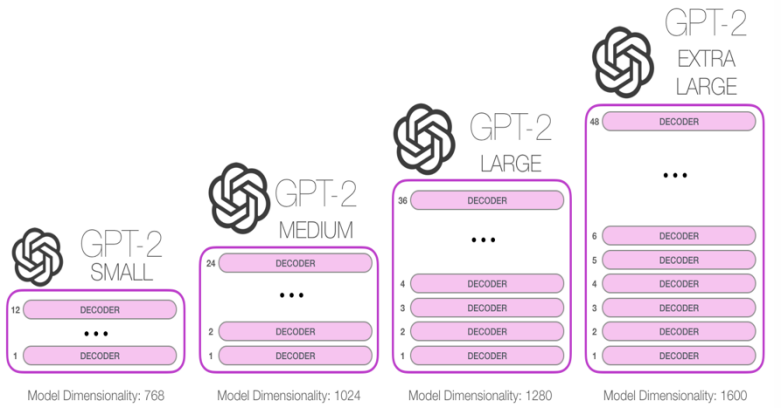
More parameters results in:

- better performance
- but longer training times
- and larger memory requirements

BERT/GPT used a **context size** of **500** or **1000 tokens**



Source: http://jalammar.github.io/illustrated-bert/



Source: http://jalammar.github.io/illustrated-gpt2/

Parameters	Layers	d_{model}
117M	12	768
345M	24	1024
762M	36	1280
1542M	48	1600

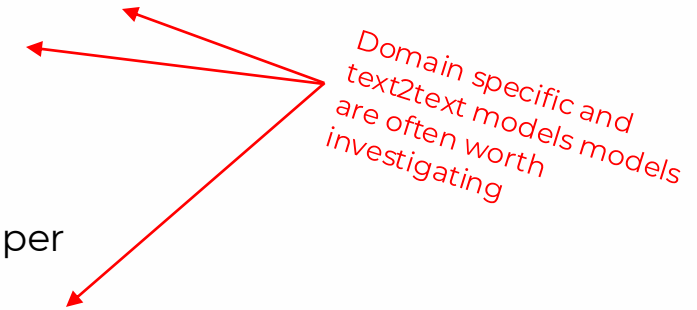
Table 2. Architecture hyperparameters for the 4 model sizes.

From: https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

Related models

Variants on BERT:

- **RoBERTa** (Facebook's version of BERT)
  - modifies slightly training objective
  - trained on more data with larger batches
- **XLNet** (BERT with some GPT-2)
  - introduces autoregressive modelling (GPT-2) into BERT training
  - was quite hyped for a while
- **DistilBERT** (a distilled version of BERT)
  - designed to be smaller (40%) and faster (60%) to fine-tune, while retaining 97% of accuracy
- Huge number of **other variations** out there
  - Many **pretrained** for **specific text domains**, e.g. medical literature
  - Many **finetuned** for **specific tasks**, e.g. question answering



Encoder-Decoder model:

- **T5** (Text-To-Text Transfer Transformer)
  - uses encoder+decoder model, same as the original transformer paper
  - uses clever **relative** positional encoding
  - so **particularly useful** for **translation** or other **text2text problems**

What can Transformers be used for?

Text classification with BERT

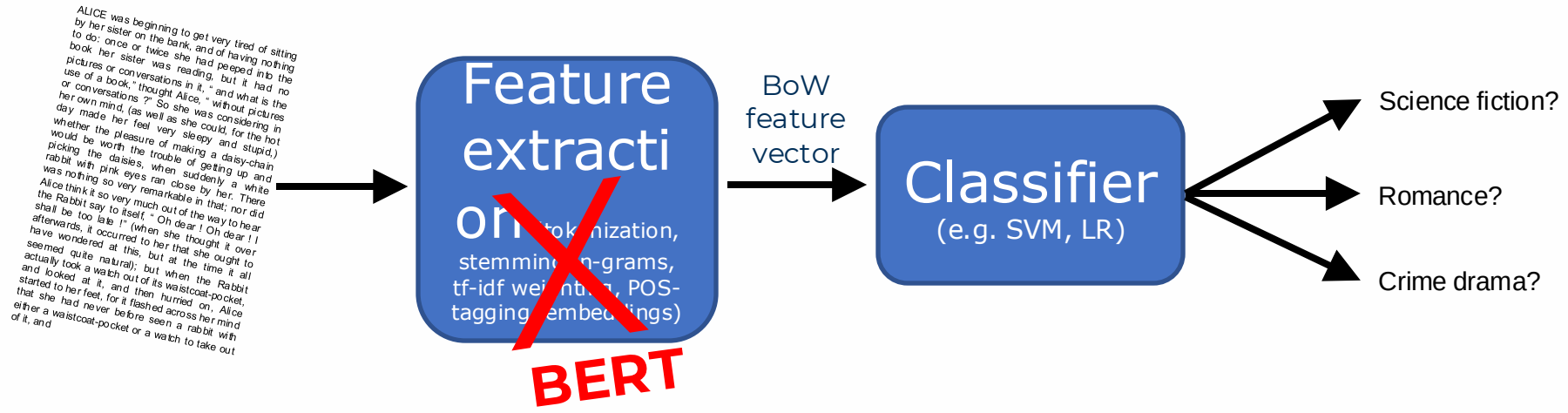
Traditionally, to build a text classifier:

- first decide on types of features to extract from text (e.g. n-gram counts)
- and how to process them (e.g. stemming, idf-weighting, add PoS tags, etc.)

BERT removes the feature extraction step

Moreover, performance improvements likely over count-based features

- since BERT leverages unsupervised pre-training (language modelling)
- and doesn't discard word order



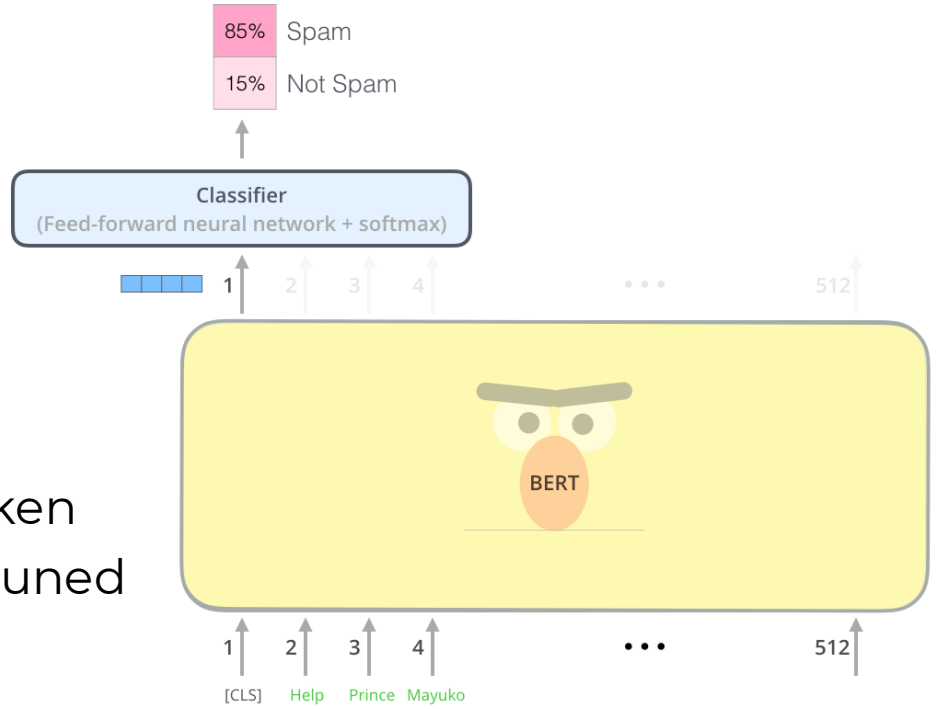
How is BERT fine-tuned?

During **pre-training**:

- a special **[CLS]** token was added to start of text of all text,
- but wasn't used for anything (loss function ignored it)

During **fine-tuning**:

- teach the model to produce **class label** in place of [CLS] token
- i.e. instead of predicting a missing word, the model is fine-tuned to predict the missing class label



Source: http://jalammar.github.io/illustrated-bert/

What else can we do with them?

Transformers allow for **transfer learning** with text

- model **comes pre-trained** on enormous quantities of data
- **fine-tune** model on specific task for which little data is usually available

Transfer learning for text can lead to:

- big **improvements in performance**
  - better performance from **small training corpus** since model leverages unsupervised pretraining to learn language model
- **multi-linguality:**
  - Multilingual BERT has been pretrained on **104 languages**!
  - can **train** classification model **on English documents**, use it on Italian ones!
    - transfer learning across languages was possible but practically unheard of before
    - amazing possibilities for resource poor languages

How does using BERT for classification work in practice?

Download large **pre-trained language model**

- choose type of model: lowercase? multilingual? size?
- choose what layers of the model you want to fine-tune on your labelled training data

Advantage: **no need to do any feature engineering**!

- e.g. choose whether to run stemming, whether to use n-grams, etc.

Any cons compared to training simple linear classifier e.g. an SVM?

- hard **limit on length of text**
  - usually to be less than 1000 tokens due all pairwise comparisons being performed
  - may need to break text into smaller chunks
- takes **much longer** and requires **more effort** to train model
  - **need fast hardware** (GPUs) to train model
  - may need to spend time **tuning the learning rate** to make sure model learns but doesn't collapse
- model will be big and require more memory
  - may be **slower** when making predictions
- predictions are **less interpretable**
  - although techniques exist to try to explain the predictions (e.g. LIME)

Text Generation with GPT type models

GPT-2 is a fantastic text generator

Context (WebText test)

Corporal Michael P. Goeldin was an unskilled laborer from Ireland when he enlisted in Company A in November 1860. Goldein survived the war. Corporal Patrick O'Neal, also from Ireland, first enlisted in 1854 and served with Company L, 3d U.S. Artillery, in Oregon. He returned to the East Coast and enlisted in the company in 1860. O'Neal served until 1874, when he was named superintendent of the National Cemetery at Willets Point, New York. Corporal Benjamin Browne was a shoemaker from Orange County, New York. In August 1862, he enlisted in the newly formed 124th New York Volunteers, and was one of sixty-one men who transferred into Company A that October. Browne reenlisted in the company in February 1864 while it was camped at Brandy Station. He returned to civilian life after completing his enlistment in 1867.

On 10 June, Artificer William Collins was promoted to corporal, probably to fill a combat leadership void for the crossing of the James River. Collins's service record does not reflect the qualities he demonstrated to earn this promotion, but he had obviously overcome some serious problems. Born in Sacketts Harbor, New York, Collins enlisted in the company in December 1853 at the age of twenty-two, and reenlisted in December 1858. Just a month before the war began in April 1861, Collins went "over the hill" and was not caught until three years later. Returned to the company on 22 March 1864, he was tried

Image source: "Language Models are Unsupervised Multitask Learners" by Radford et al. https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

GPT-2 completion

in June of that year. Sentenced to be shot for desertion, he remained on active duty.

Captain James J. Dall was born in Virginia in 1829 during the Revolution and joined his father in the Army in 1836. He served with the 2d Virginia Infantry until 1845, when he went to the Western Reserve Regiment; and, on his discharge, he reenlisted with Company I, 25th New York Volunteers (which in 1845 numbered about 4,500 men). One of his brothers, James, served with his company in the same capacity in the same brigade. While the regiment remained in Connecticut through 1846, Captain Dall served with Company I at various times before his discharge. A veteran of the Mexican War, he had served two years on the staff of Brigadier General John J. Sullivan at San Antonio. During the Peninsula Campaign of 1863, he commanded a company of twenty-three and was in charge of the battery of fifty-four heavy mortars and the gunboat Pinta of the Battery, a gunboat and several other boats. Captain Dall was on active duty with Company I in the fall of 1865. Two years later, he was

Conditioned on **large** piece of text

- GPT-2 (2019) model can generate new content
- that **sounds HIGHLY believable**
- and **VERY creative**
- but **can be nonsensical**
- or **disagree** with **common sense** or reliable sources of true facts

- Note: newer/bigger models, like GPT-3/4 don't make nearly as many nonsensical statements,
  - but do still invent untruths (as do humans),
  - which depending on task might be (in)appropriate

GPT-2 is a fantastic text generator

Context (WebText test)

Corporal Michael P. Goeldin was an unskilled laborer from Ireland when he enlisted in Company A in November 1860. Goldein survived the war. Corporal Patrick O'Neal, also from Ireland, first enlisted in 1854 and served with Company L, 3d U.S. Artillery, in Oregon. He returned to the East Coast and enlisted in the company in 1860. O'Neal served until 1874, when he was named superintendent of the National Cemetery at Willets Point, New York. Corporal Benjamin Browne was a shoemaker from Orange County, New York. In August 1862, he enlisted in the newly formed 124th New York Volunteers, and was one of sixty-one men who transferred into Company A that October. Browne reenlisted in the company in February 1864 while it was camped at Brandy Station. He returned to civilian life after completing his enlistment in 1867.

On 10 June, Artificer William Collins was promoted to corporal, probably to fill a combat leadership void for the crossing of the James River. Collins's service record does not reflect the qualities he demonstrated to earn this promotion, but he had obviously overcome some serious problems. Born in Sacketts Harbor, New York, Collins enlisted in the company in December 1853 at the age of twenty-two, and reenlisted in December 1858. Just a month before the war began in April 1861, Collins went "over the hill" and was not caught until three years later. Returned to the company on 22 March 1864, he was tried

GPT-2 completion

in June of that year. Sentenced to be shot for desertion, he remained on active duty.

Captain James J. Dall was born in Virginia in 1829 during the Revolution and joined his father in the Army in 1836. He served with the 2d Virginia Infantry until 1845, when he went to the Western Reserve Regiment; and, on his discharge, he reenlisted with Company I, 25th New York Volunteers (which in 1845 numbered about 4,500 men). One of his brothers, James, served with his company in the same capacity in the same brigade. While the regiment remained in Connecticut through 1846, Captain Dall served with Company I at various times before his discharge. A veteran of the Mexican War, he had served two years on the staff of Brigadier General John J. Sullivan at San Antonio. During the Peninsula Campaign of 1863, he commanded a company of twenty-three and was in charge of the battery of fifty-four heavy mortars and the gunboat Pinta of the Battery, a gunboat and several other boats. Captain Dall was on active duty with Company I in the fall of 1865. Two years later, he was

Image source: "Language Models are Unsupervised Multitask Learners" by Radford et al. https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

Text generation is building block for **many** applications

- from **question answering** to **chatbots**
- but is now being applied to **ever-more tasks**
  - from classification to data integration
  - more on this in a moment
- so models providing massive improvements in text generation performance are a big deal

GPT-2 is a fantastic text generator

Context (WebText test)

Corporal Michael P. Goeldin was an unskilled laborer from Ireland when he enlisted in Company A in November 1860. Goldein survived the war. Corporal Patrick O'Neal, also from Ireland, first enlisted in 1854 and served with Company L, 3d U.S. Artillery, in Oregon. He returned to the East Coast and enlisted in the company in 1860. O'Neal served until 1874, when he was named superintendent of the National Cemetery at Willets Point, New York. Corporal Benjamin Browne was a shoemaker from Orange County, New York. In August 1862, he enlisted in the newly formed 124th New York Volunteers, and was one of sixty-one men who transferred into Company A that October. Browne reenlisted in the company in February 1864 while it was camped at Brandy Station. He returned to civilian life after completing his enlistment in 1867.

On 10 June, Artificer William Collins was promoted to corporal, probably to fill a combat leadership void for the crossing of the James River. Collins's service record does not reflect the qualities he demonstrated to earn this promotion, but he had obviously overcome some serious problems. Born in Sacketts Harbor, New York, Collins enlisted in the company in December 1853 at the age of twenty-two, and reenlisted in December 1858. Just a month before the war began in April 1861, Collins went "over the hill" and was not caught until three years later. Returned to the company on 22 March 1864, he was tried

GPT-2 completion

in June of that year. Sentenced to be shot for desertion, he remained on active duty.

Captain James J. Dall was born in Virginia in 1829 during the Revolution and joined his father in the Army in 1836. He served with the 2d Virginia Infantry until 1845, when he went to the Western Reserve Regiment; and, on his discharge, he reenlisted with Company I, 25th New York Volunteers (which in 1845 numbered about 4,500 men). One of his brothers, James, served with his company in the same capacity in the same brigade. While the regiment remained in Connecticut through 1846, Captain Dall served with Company I at various times before his discharge. A veteran of the Mexican War, he had served two years on the staff of Brigadier General John J. Sullivan at San Antonio. During the Peninsula Campaign of 1863, he commanded a company of twenty-three and was in charge of the battery of fifty-four heavy mortars and the gunboat Pinta of the Battery, a gunboat and several other boats. Captain Dall was on active duty with

Image source: "Language Models are Unsupervised Multitask Learners" by Radford et al. https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

Text generation is building block for **many** applications

- from **question answering** to **chatbots**
- but is now being applied to **ever-more tasks**
  - from classification to data integration
  - more on this in a moment
- so models providing massive improvements in text generation performance are a big deal

Conclusions

Conclusions

Sequence2sequence models:

- make use of Recurrent Neural Network and encoder-decoder framework
- improved state-of-art for many NLP tasks, like translation, question answering, summarization, etc.
- made use of attention architecture

Transformer models:

- removed recurrent link in RNN with attention and replaced with self-attention
- stack many self-attention and feedforward blocks on top of each other in deep architecture
- provide state-of-the-art performance for text classification (BERT) and text generation (GPT*)