Natural Language Processing with Deep Learning

Transformers and more Contextualized Word Embeddings

Navid Rekab-Saz

navid.rekabsaz@jku.at

Institute of Computational Perception

JOHANNES KEPLER UNIVERSITY LINZ

Institute of Computational Perception

Agenda

- Transformers
- seq2seq with Transformers
- BERT
- Model compression

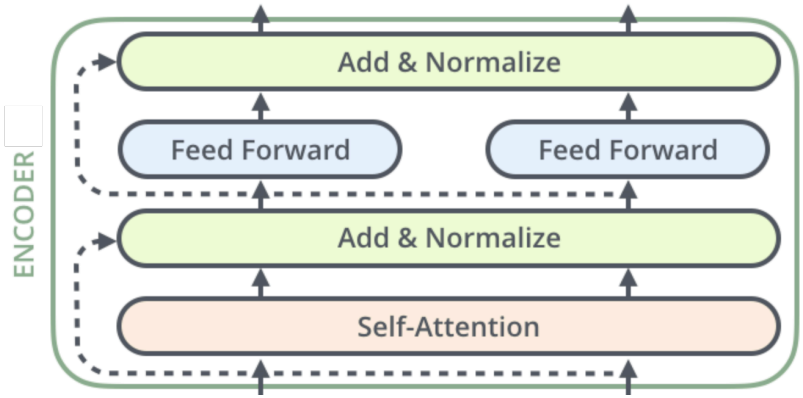
Agenda

- **Transformers**
- seq2seq with Transformers
- BERT
- Model compression

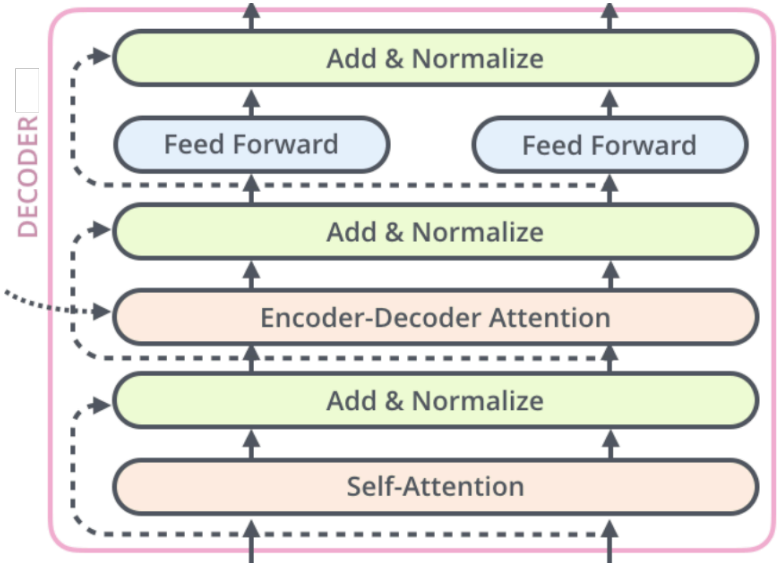
Transformers

- An attention model with DL best practices!
- Originally introduced for machine translation, and now widely adopted for non-recurrent sequence encoding and decoding

Transformer encoder



Transformer decoder



Attention is all you need. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Polosukhin, I. (2017). In *NeurIPS*.

Transformer encoder

Transformer encoder consists of two sub-layers:

- 1st : Multi-head scaled dot-product self-attention
- 2nd : Position-wise feed forward
- Each sub-layer is followed by layer normalization and residual networks … and drop-outs are applied after each computation

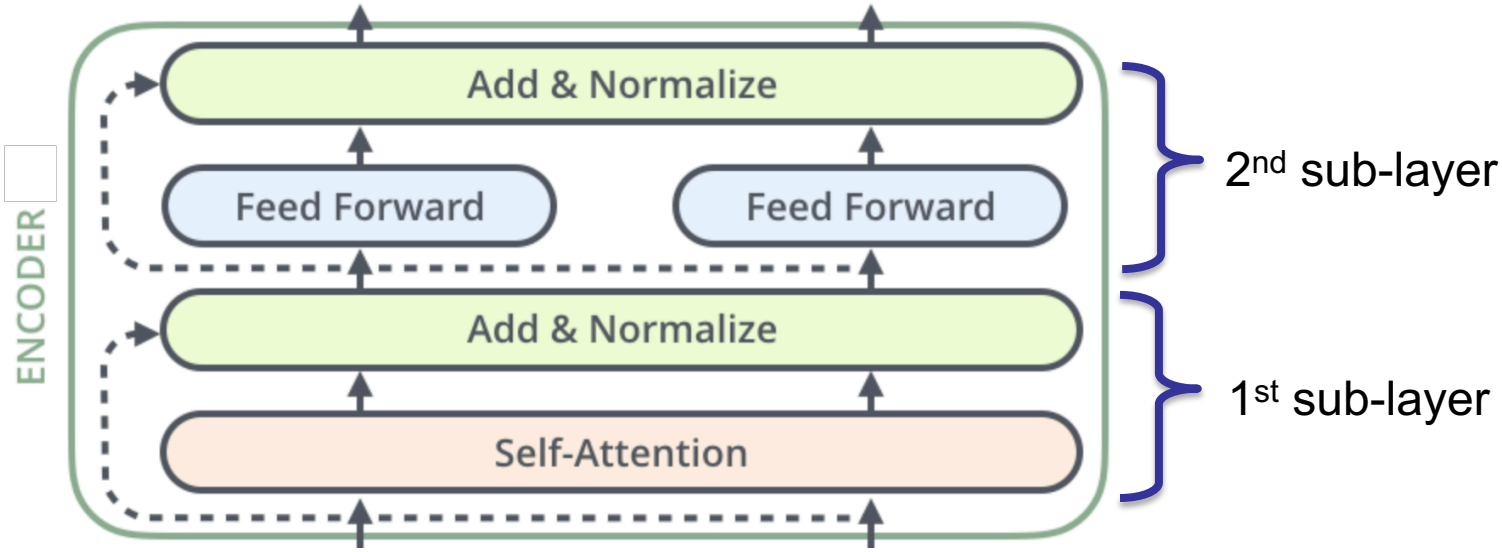


Figure source: http://jalammar.github.io/illustrated-transformer/

Transformer encoder

Let's start from multi-head scaled dot-product self-attention:

1. Scaled dot-product attention
2. Multi-head attention
3. self-attention (recap)

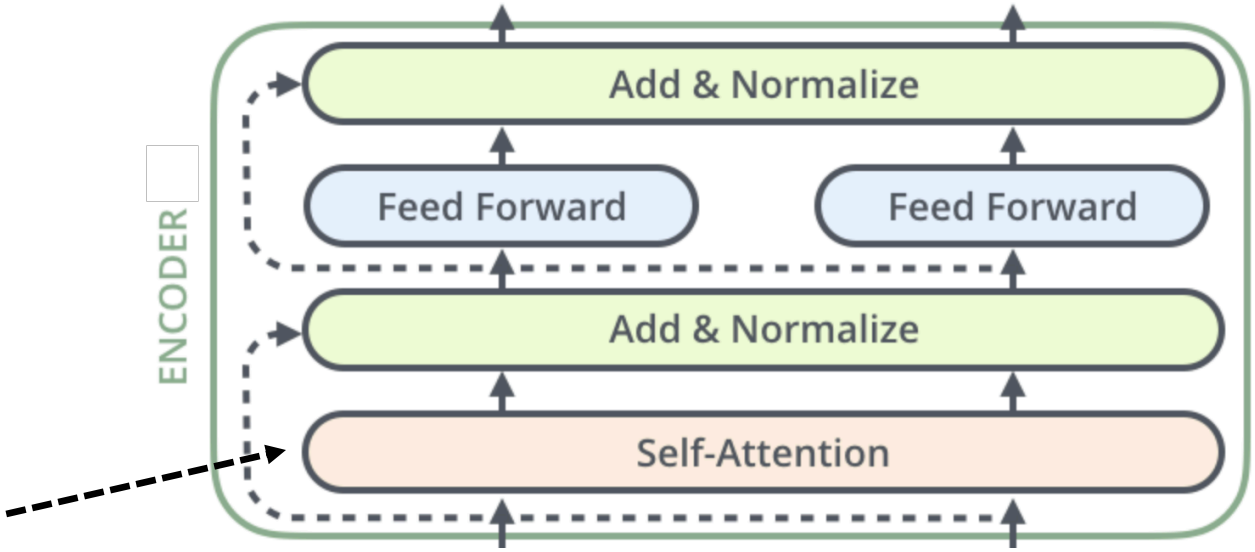


Figure source: http://jalammar.github.io/illustrated-transformer/

Recap: basic dot-product attention

- First, non-normalized attention scores:

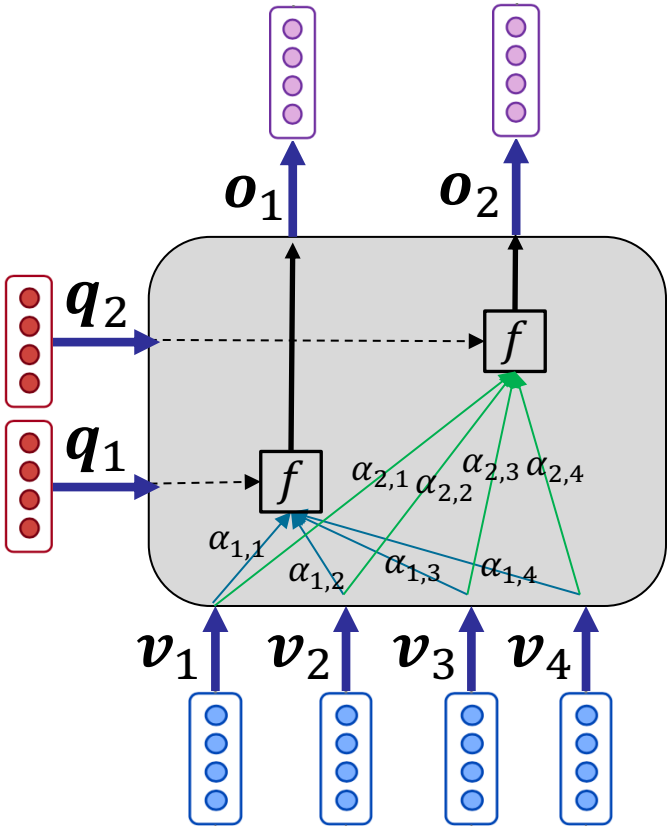
$$\tilde{\alpha}_{i,j} = \boldsymbol{q}_i \boldsymbol{v}_j^{\mathrm{T}}$$

  - $d = d_q = d_v$ dimension of vectors
  - has no parameter!

- Then, softmax over values:

$$\alpha_{i,j} = \mathrm{softmax}(\tilde{\boldsymbol{\alpha}}_i)_j$$

- Output (weighted sum): $\boldsymbol{o}_i = \sum_{j=1}^{|V|} \alpha_{i,j} \boldsymbol{v}_j$



Scaled dot-product attention

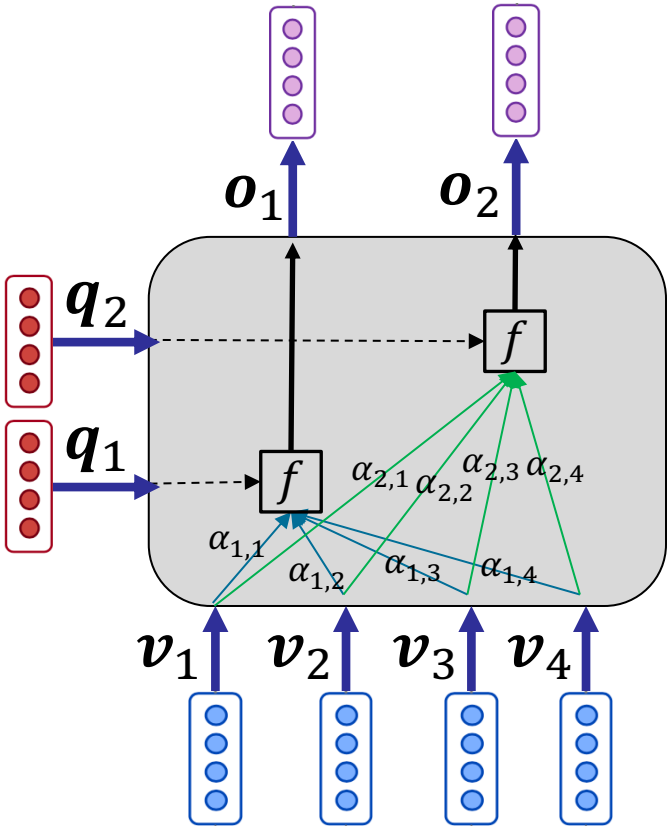
- Problem with basic doc-product attention:
  - As d gets large, the variance of $\tilde{\alpha}_{i,j}$ increases …
  - … this makes softmax very peaked for some values $\tilde{\boldsymbol{\alpha}}_i$ …
  - … and hence its gradient gets smaller
- Solution: normalize/scale $\tilde{\alpha}_{i,j}$ by size of d

Scaled dot-product attention

- Non-normalized attention scores:

$$\tilde{\alpha}_{i,j} = \frac{\boldsymbol{q}_i \boldsymbol{v}_j^{\mathrm{T}}}{\sqrt{d}}$$

- Softmax over values: $\alpha_{i,j} = \mathrm{softmax}(\tilde{\boldsymbol{\alpha}}_i)_j$
- Output (weighted sum): $\boldsymbol{o}_i = \sum_{j=1}^{|\boldsymbol{V}|} \alpha_{i,j} \boldsymbol{v}_j$

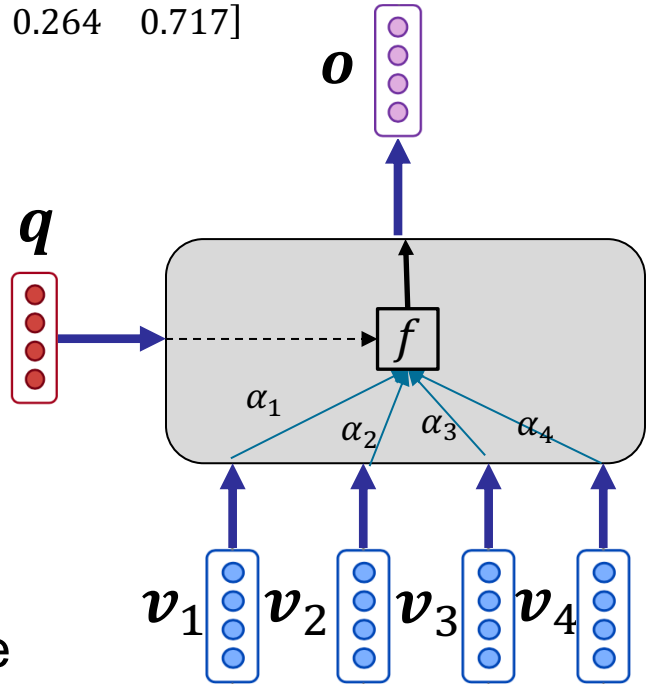


Problem with (single-head) attention

- In all attention networks so far, the final attention of query $\boldsymbol{q}$ on value vectors $\boldsymbol{V}$ are normalized with softmax
  - Recall that softmax makes the maximum value much higher than the other

$$\boldsymbol{z} = [1 \quad 2 \quad 5 \quad 6] \rightarrow \text{softmax}(\boldsymbol{z}) = [0.004 \quad 0.013 \quad 0.264 \quad 0.717]$$

- Common in language, a word may be related to several other words in sequence, each through a specific concept
  - Like the relations of a verb to its subject and to its object
- However in a (single-head) attention network, all concepts are aggregated in one attention set
- Due to softmax, value vectors must compete for the attention of query vector → softmax bottleneck



Multi-head attention

- Multi-head attention approaches this by calculating multiple sets of attentions between queries and values

Multi-head attention:

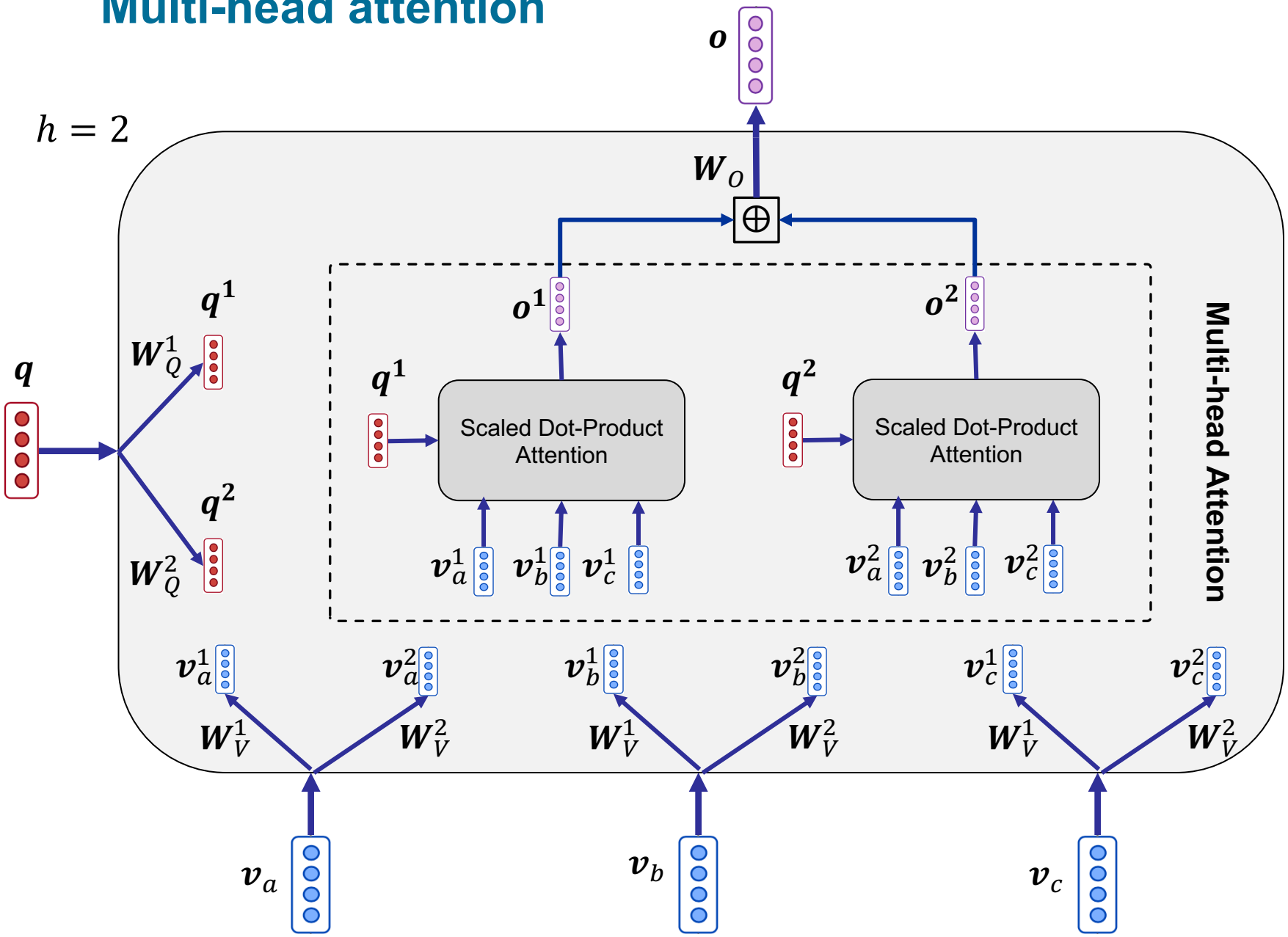
1. Down-project query and value vectors to h subspaces (heads)

2. In each subspace, calculate a simple attention network using the queries and values projected in the subspace, resulting in output vectors of the subspace

3. Concatenate the output vectors of all subspaces regarding each query, resulting in the final output of each query

- In multi-head attention, each head (and each subspace) can specialize on capturing a specific kind of relations

Multi-head attention



Multi-head attention – formulation

- Down-project every query $\boldsymbol{q}_i$ to h vectors, each with size $^d/_h$:

size: $^d/_h$ ← $\boldsymbol{q}_i^1 = \boldsymbol{q}_i \boldsymbol{W}_Q^1 \quad \dots \quad \boldsymbol{q}_i^h = \boldsymbol{q}_i \boldsymbol{W}_Q^h$ → Matrix size: $d \times {}^d/_h$

- Down-project every value $\boldsymbol{v}_j$ to h vectors, each with size $^d/_h$:

size: $^d/_h$ ← $\boldsymbol{v}_j^1 = \boldsymbol{v}_j \boldsymbol{W}_V^1 \quad \dots \quad \boldsymbol{v}_j^h = \boldsymbol{v}_j \boldsymbol{W}_V^h$ → Matrix size: $d \times {}^d/_h$

- Calculate outputs of subspaces corresponding to $\boldsymbol{q}_i$:

size: $^d/_h$ ← $\boldsymbol{o}_i^1 = \mathrm{ATT}(\boldsymbol{q}_i^1, \boldsymbol{V}^1) \quad \dots \quad \boldsymbol{o}_i^h = \mathrm{ATT}(\boldsymbol{q}_i^h, \boldsymbol{V}^h)$

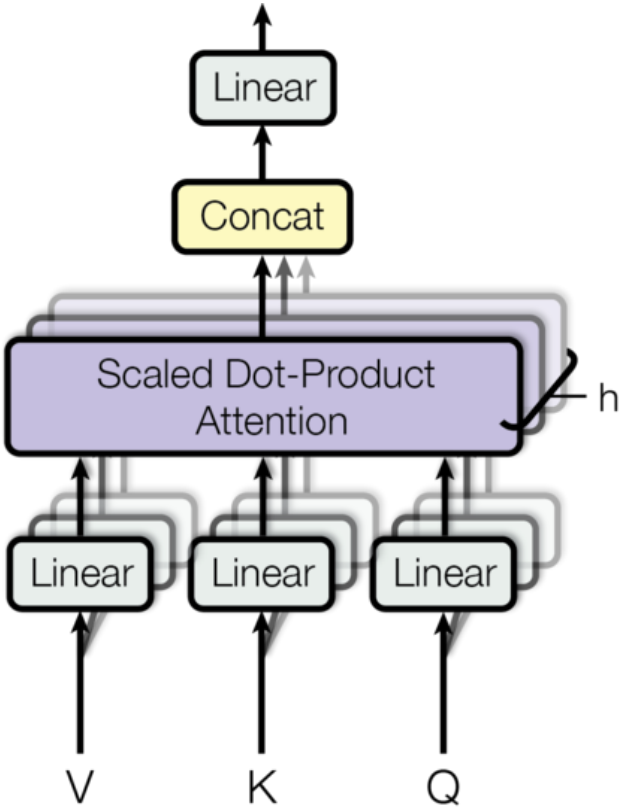
- Concatenate outputs of subspaces for $\boldsymbol{q}_i$ as its final output:

size: d ← $\boldsymbol{o}_i = \boldsymbol{W}_O \, [\boldsymbol{o}_i^1; \dots; \boldsymbol{o}_i^h]$

Size: $d \times d$
This matrix linearly combines the dimensions of the concatenated vectors

Parameters are shown in red

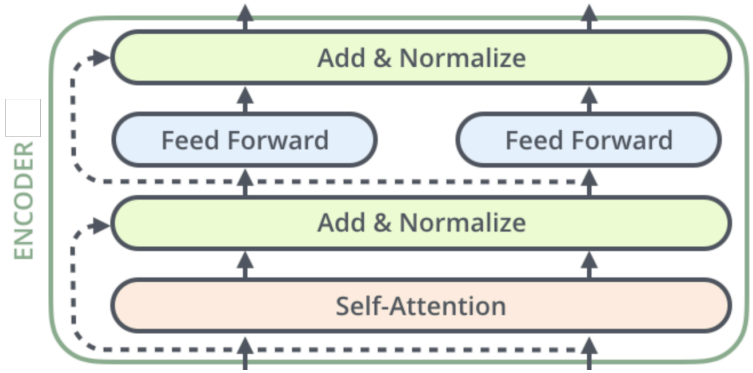
Multi-head attention – in original paper

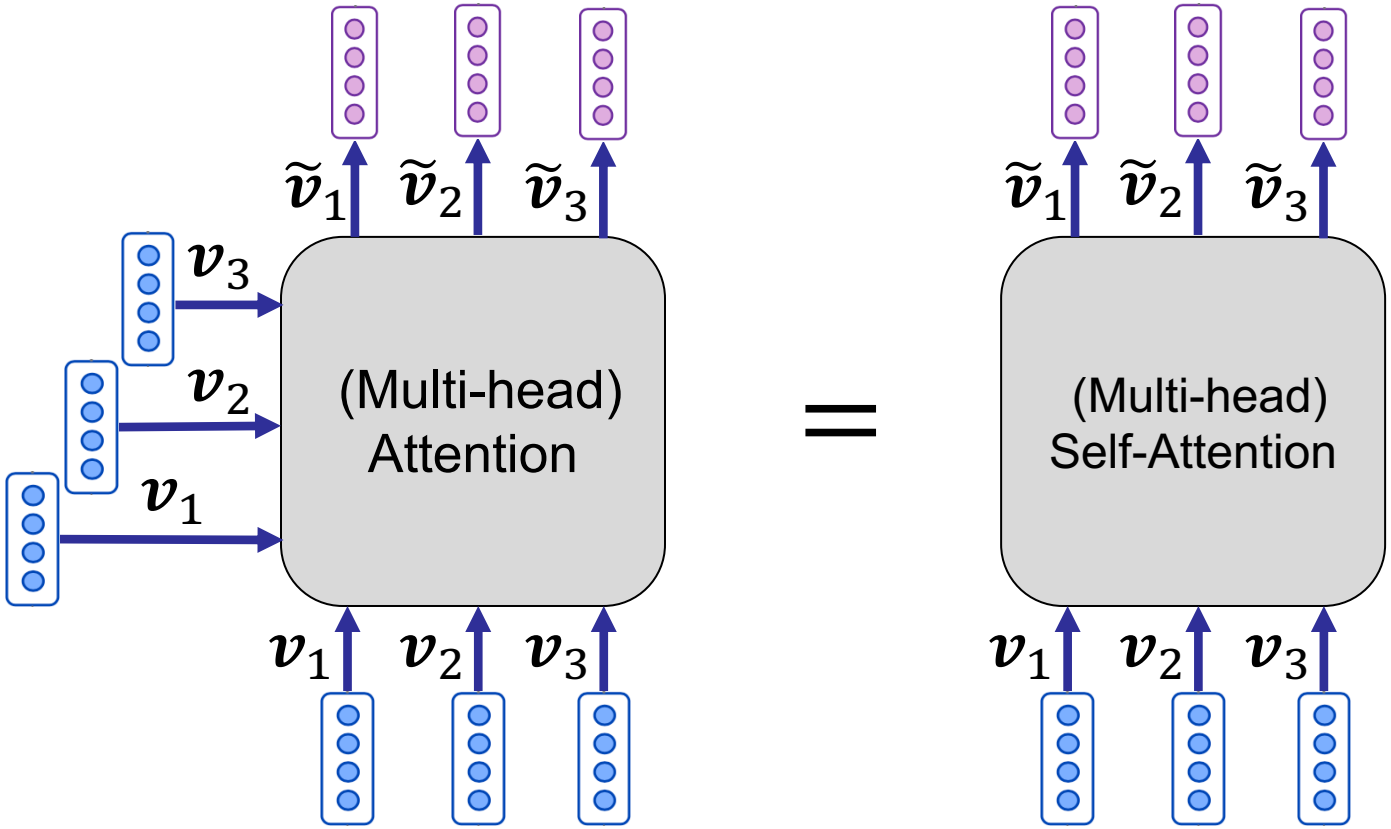


- Default number of heads in Transformers: $h = 8$
- Recall: Attentions (and Transformers) in fact have three inputs (not two), namely queries, keys, and values.
  - Keys are used to calculate attentions
  - Values are used to produce outputs

Self-attention – recap

- Values are the same as queries
- Each encoded vector is the contextual embedding of the corresponding input vector
  - $\tilde{\boldsymbol{v}}_i$ is the contextual embedding of $\boldsymbol{v}_i$



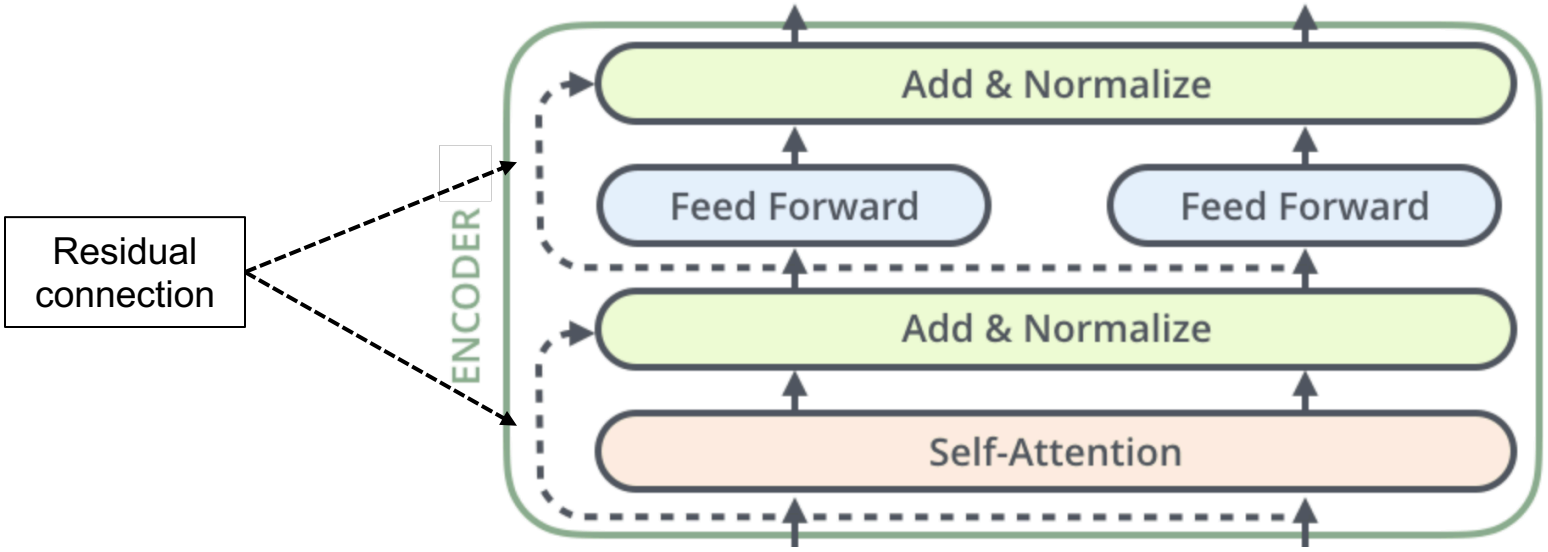


Residuals

- Residual (short-cut) connection:

$$\text{output} = f(x) + x$$

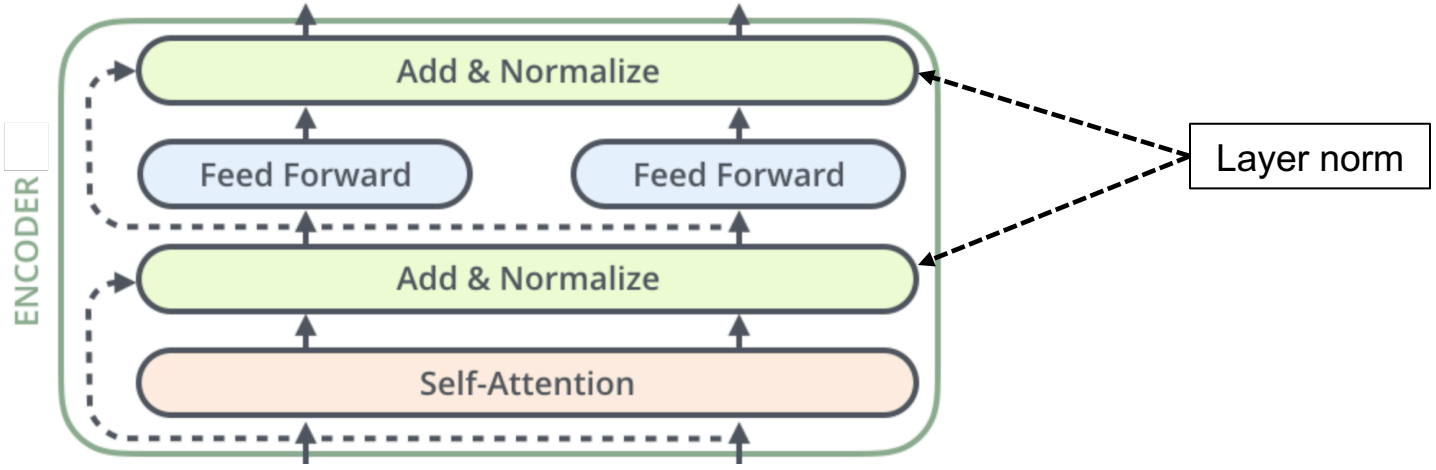
- Learn in detail:
  - He, Kaiming; Zhang, Xiangyu; Ren, Shaoqing; Sun, Jian (2016). "Deep Residual Learning for Image Recognition" . In proc. of CVPR
  - Srivastava, Rupesh Kumar; Greff, Klaus; Schmidhuber, Jürgen (2015). "Highway Networks". https://arxiv.org/pdf/1505.00387.pdf



Layer normalization

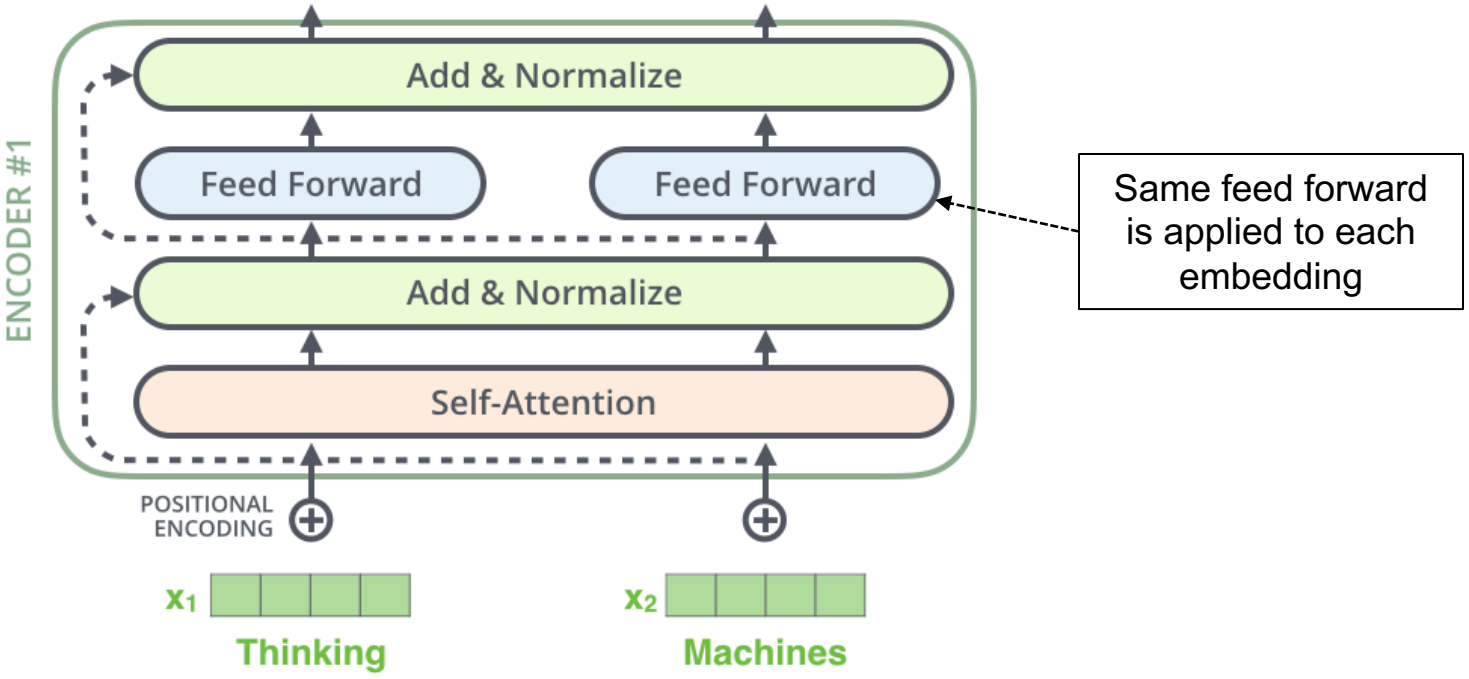
- Layer normalization changes each vector to have mean 0 and variance 1 …
  - … and learns two more parameter vectors per layer that set new means and variances for each dimension of the vectors

- Learn in detail:
  - Batch Normalization: Deep Learning book section 8.7.1 http://www.deeplearningbook.org/contents/optimization.html
  - Talk by Goodfellow https://www.youtube.com/watch?v=Xogn6veSyxA&feature=youtu.be
  - Paper: https://arxiv.org/pdf/1607.06450.pdf



Feed Forward on embedding

- In Transformers, a two-layer feed forward neural network (with ReLU) is applied to each embedding
  - With the feed forward network, the Transformers gain the capacity to learn non-linear transformations over each (contextualized) embedding



Transformer encoder – summary

- Multi-head self-attention model followed by a feed-forward layer

Benefits (as in attentions)

- No locality bias
  - A long-distance context has "*equal opportunity*"
- Single computation per layer (non-autoregressive)
  - Friendly with high parallel computations of GPU

- Look here for self-teaching and the PyTorch implementation:
  - http://nlp.seas.harvard.edu/2018/04/03/attention.html
  - also available on Google Colab

Position embeddings

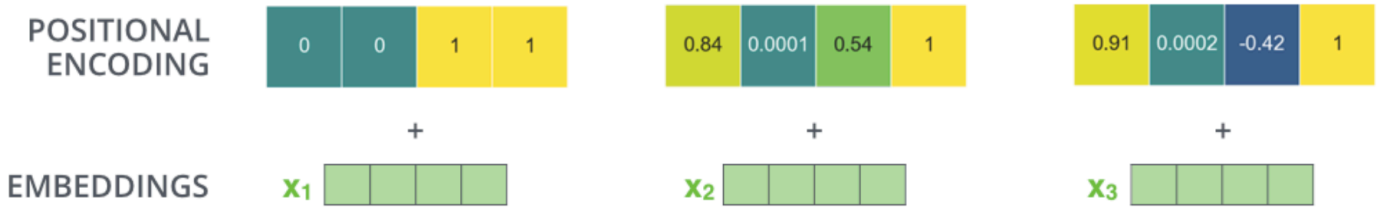
- Transformers are agnostic regarding the position of tokens (no locality bias)
  - A context token in long-distance has the same effect as one in short-distance
- However, the positions of tokens can still bring useful information

Position embeddings – a common solution in Transformers:

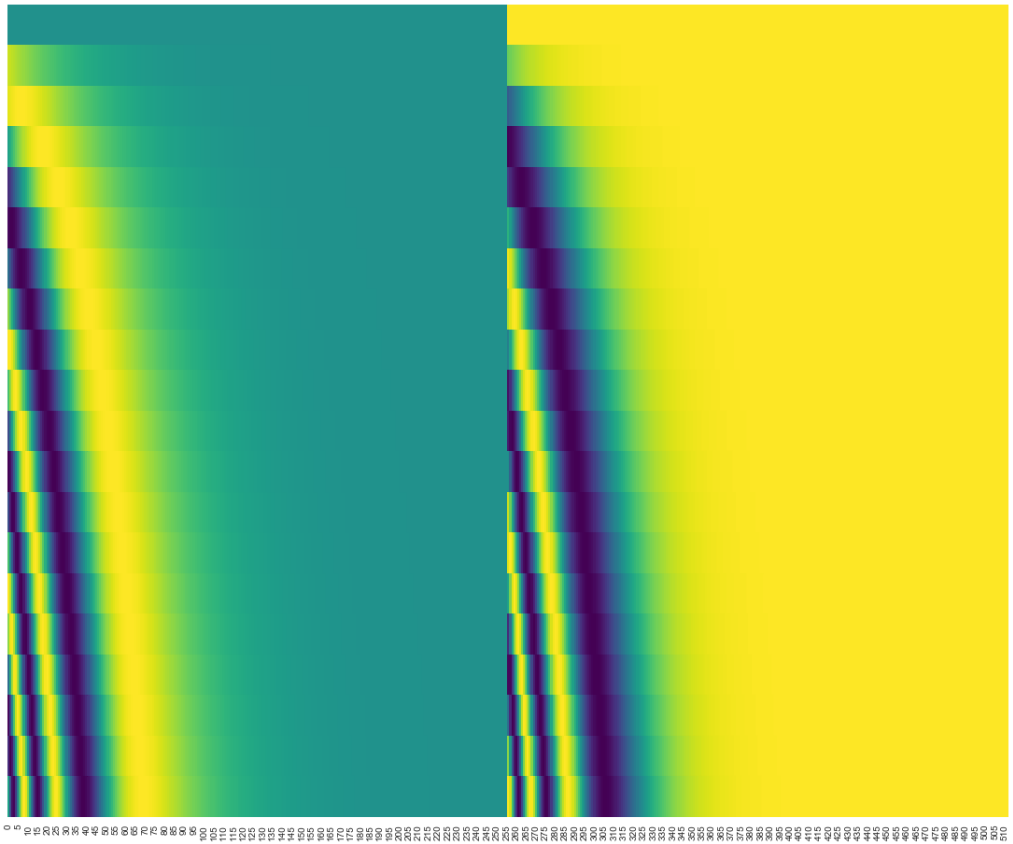
- Consider an embedding for each position, and add its values to the token embedding at that position
  - Position embedding is usually created using a sine/cosine function, or learned end-to-end with the model
  - Using position embeddings, the same word at different locations will have different overall representations

Position embeddings – examples

An example of embeddings with four dimensions:



Position embedding for location 0



Position embeddings

Position embedding for location 20

Values from -1 (dark) to +1 (light)

Dimensions (512)

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

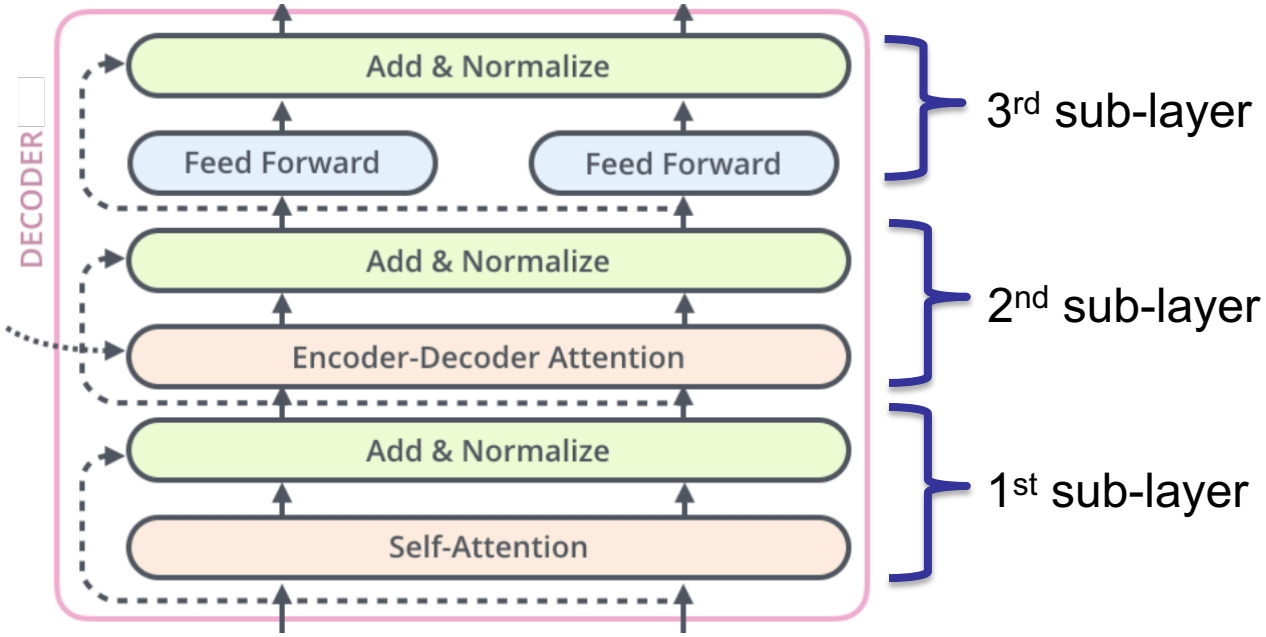
Agenda

- Transformers
- **seq2seq with Transformers**
- BERT
- Model compression

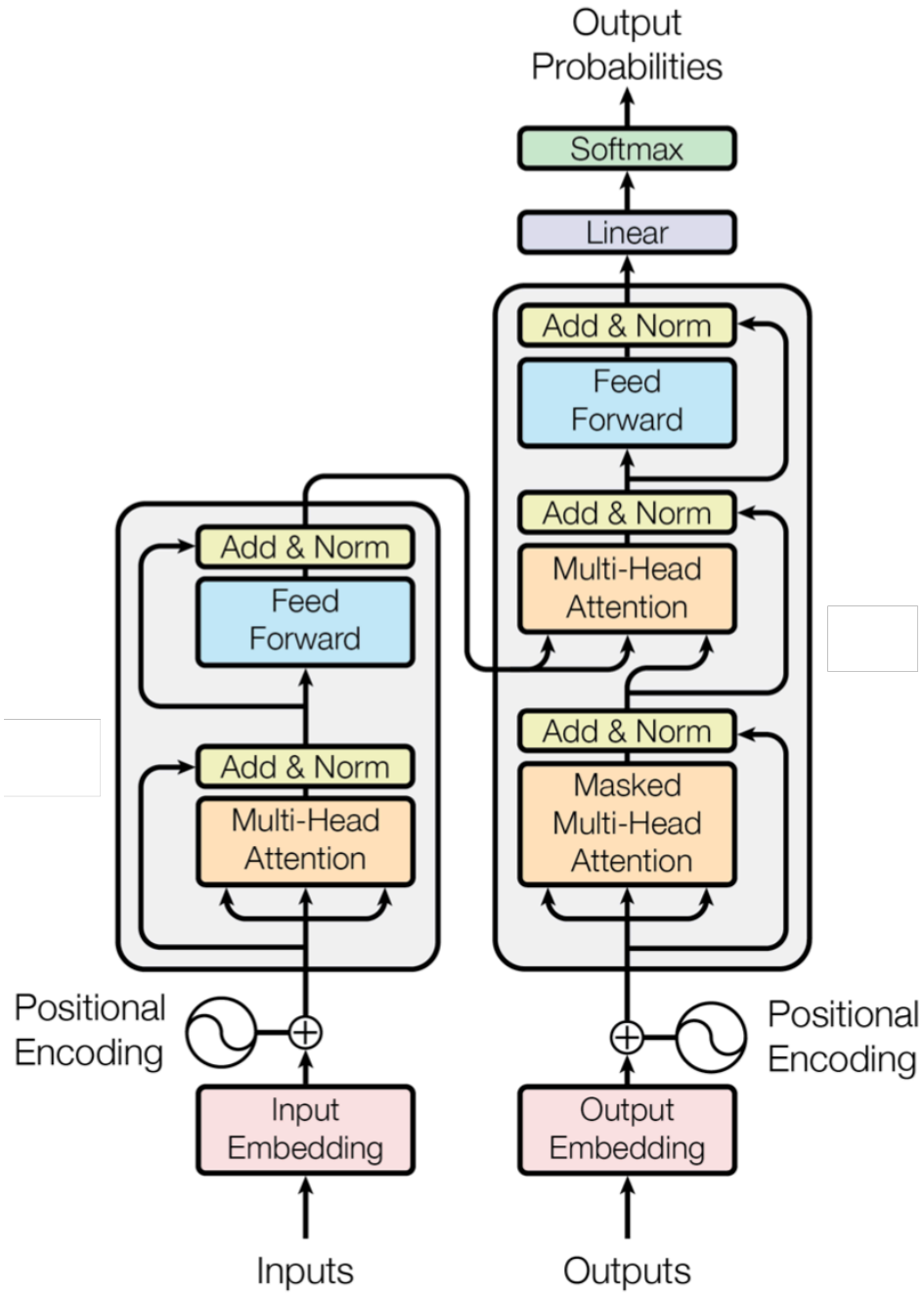
Transformer decoder

Transformer encoder consists of three sub-layers:

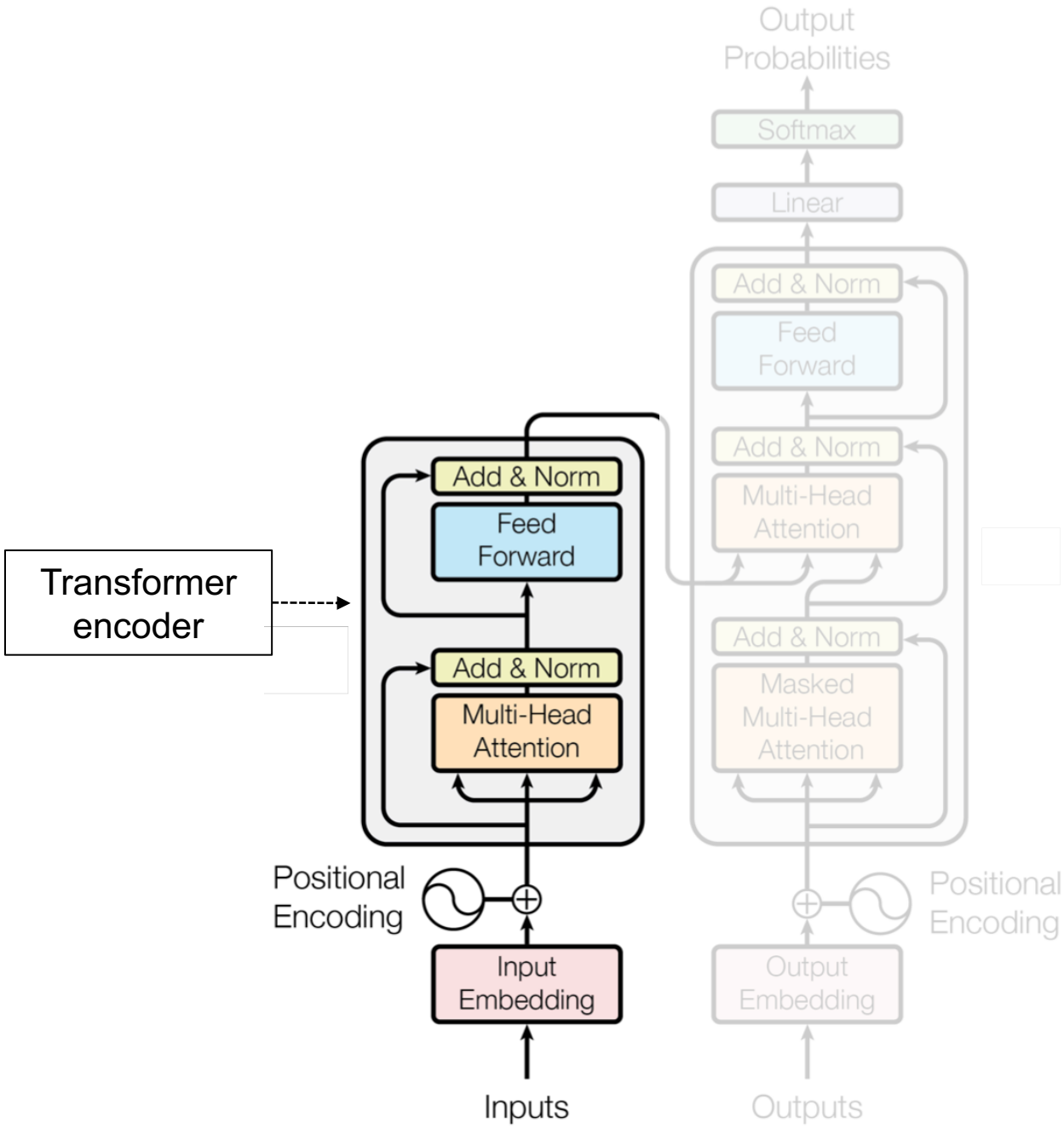
- 1st : Masked multi-head self-attention
- 2nd : Multi-head encoder-decoder attention
- 3rd : Position-wise feed forward



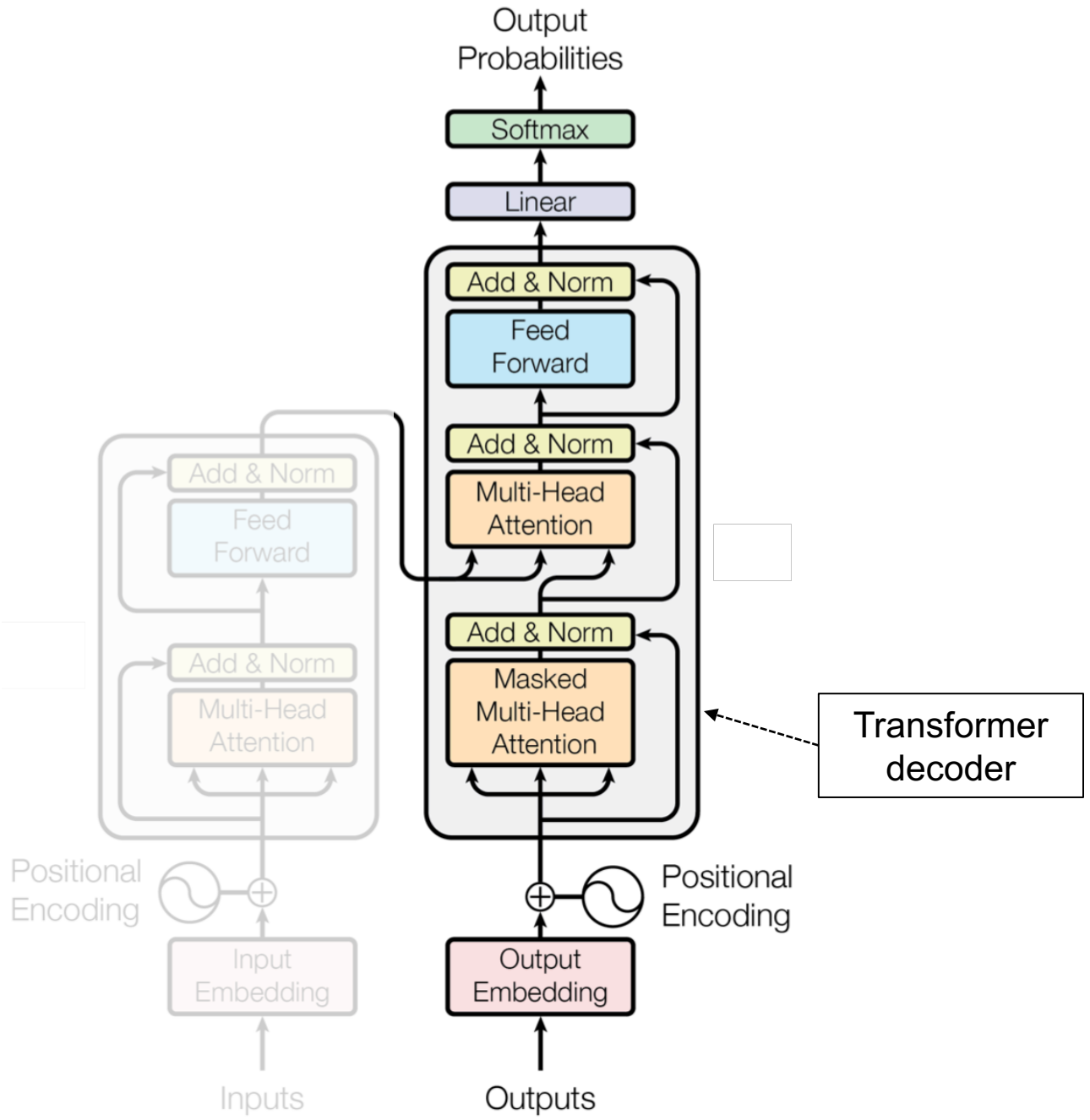
Seq2seq with Transformers



Seq2seq with Transformers



Seq2seq with Transformers



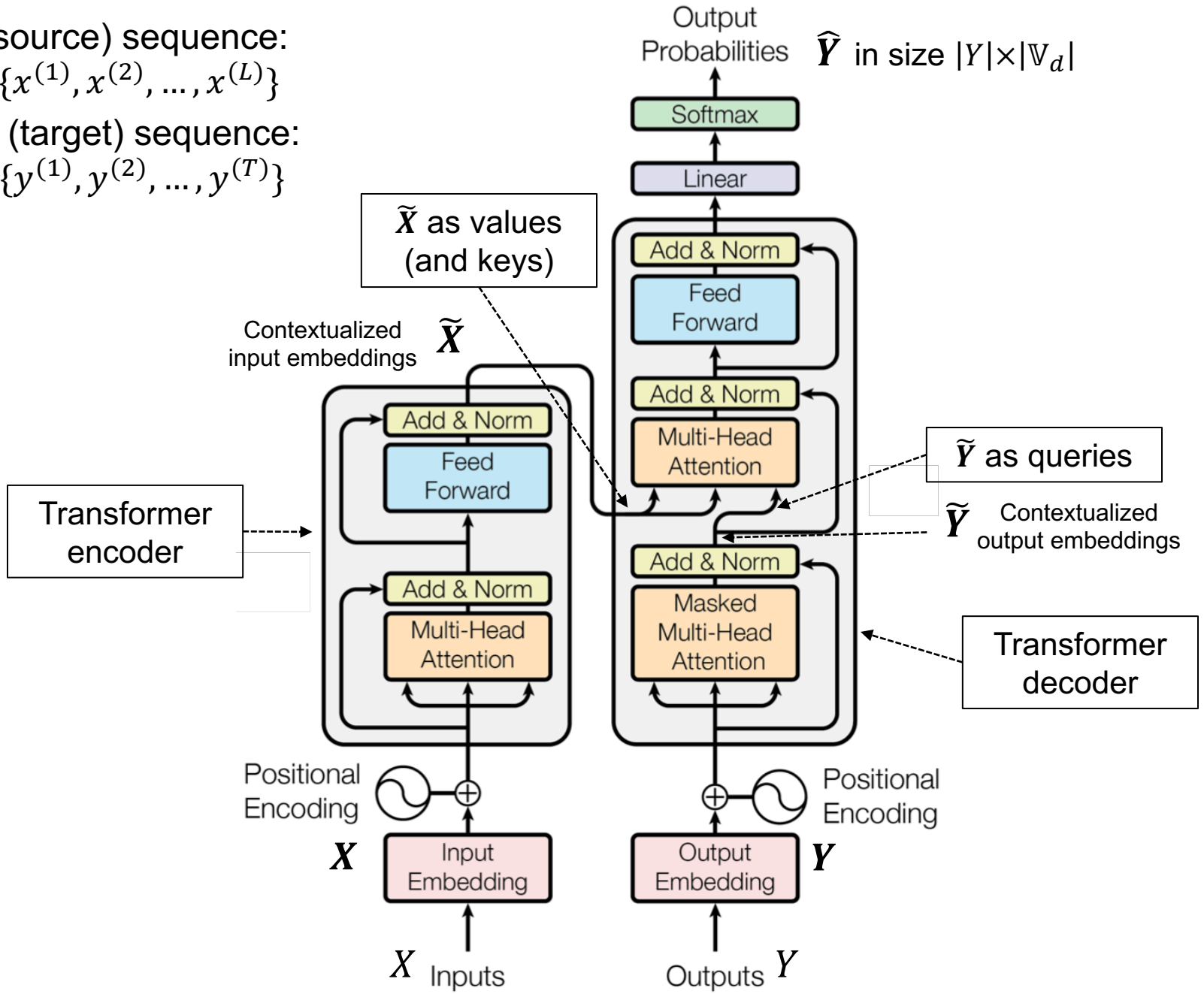
Seq2seq with Transformers

Input (source) sequence:

$X = \{x^{(1)}, x^{(2)}, \ldots, x^{(L)}\}$

Output (target) sequence:

$Y = \{y^{(1)}, y^{(2)}, \ldots, y^{(T)}\}$



Seq2seq with Transformers – training

1. Data preparation: Input (source) sequence X and output (target) sequence Y are both started with $< \text{sos} >$ and ended with $< \text{eos} >$
2. Transformer encoder: outputs contextualized embeddings $\widetilde{\boldsymbol{X}}$
   - $\widetilde{\boldsymbol{X}}$ is in size $|X| \times d$, where d is the embedding vector dimensions
3. Decoder self-attention: create contextualized embeddings $\widetilde{\boldsymbol{Y}}$
   - $\widetilde{\boldsymbol{Y}}$ is in size $|Y| \times d$
4. Decoder attention: $\widetilde{\boldsymbol{Y}}$ vectors attend to $\widetilde{\boldsymbol{X}}$ vectors ($\widetilde{\boldsymbol{Y}}$ queries, $\widetilde{\boldsymbol{X}}$ values)
   - Result matrix is in size $|Y| \times d$
5. Prediction: Transformer decoder results are used to calculate $\widehat{\boldsymbol{Y}}$, the probability distributions of next tokens
   - $\widehat{\boldsymbol{Y}}$ is in size $|Y| \times |\mathbb{V}_d|$
   - E.g., vector $\widehat{\boldsymbol{y}}^{(2)}$ predicts the probability distribution of tokens at the position 3
6. Loss: is calculated using Negative Log Likelihood of the actual next words
   - E.g., based on the probability value of token $y^{(3)}$ in vector $\widehat{\boldsymbol{y}}^{(2)}$

Problem: in decoder self-attention, every token looks at all other tokens, namely the previous ones but also the next tokens

- Every token has access to what it is supposed to predict!

Masking attentions

- To resolve the problem in decoder's self attention, the attentions of every token to future tokens are masked

Example

- Let's assume an output sequence with 4 tokens has the following (non-normalized) attention scores in decoder's self attention:

Vectors of output (target) sequence	attended to $\boldsymbol{y}^{(1)}$	$\boldsymbol{y}^{(2)}$	$\boldsymbol{y}^{(3)}$	$\boldsymbol{y}^{(4)}$
$\boldsymbol{y}^{(1)}$	5	3	1	-4
$\boldsymbol{y}^{(2)}$	1	4	-2	3
$\boldsymbol{y}^{(3)}$	0	2	2	-3
$\boldsymbol{y}^{(4)}$	3	-1	1	4

(non-normalized) attentions scores

	$y^{(1)}$	$y^{(2)}$	$y^{(3)}$	$y^{(4)}$
$y^{(1)}$	5	3	1	-4
$y^{(2)}$	1	4	-2	3
$y^{(3)}$	0	-2	2	-3
$y^{(4)}$	3	-1	1	4

attentions masks

	$y^{(1)}$	$y^{(2)}$	$y^{(3)}$	$y^{(4)}$
$y^{(1)}$	1	0	0	0
$y^{(2)}$	1	1	0	0
$y^{(3)}$	1	1	1	0
$y^{(4)}$	1	1	1	1

applying masks to attention scores

	$y^{(1)}$	$y^{(2)}$	$y^{(3)}$	$y^{(4)}$
$y^{(1)}$	5	$-\infty$	$-\infty$	$-\infty$
$y^{(2)}$	1	4	$-\infty$	$-\infty$
$y^{(3)}$	0	-2	2	$-\infty$
$y^{(4)}$	3	-1	1	4

softmax

(final) attentions

	$y^{(1)}$	$y^{(2)}$	$y^{(3)}$	$y^{(4)}$
$y^{(1)}$	1.00	0.00	0.00	0.00
$y^{(2)}$	0.04	0.96	0.00	0.00
$y^{(3)}$	0.11	0.01	0.86	0.00
$y^{(4)}$	0.25	0.01	0.34	0.70

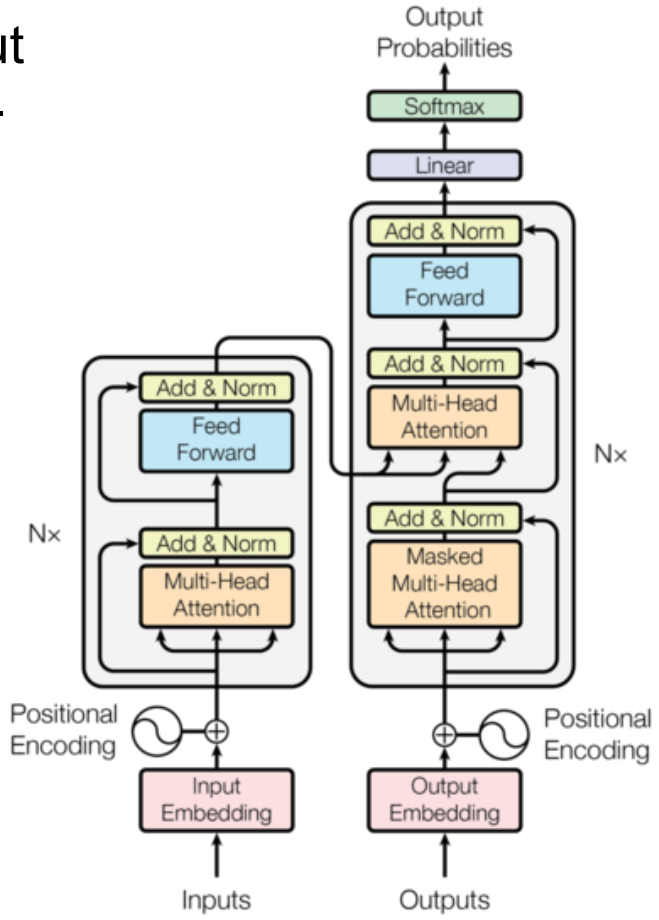
☞ In multi-head attention networks, there exists h times of such attention matrices. The same masking mechanism is applied to all of them.

Training (corrected!)

1. Data preparation: Input (source) sequence X and output (target) sequence Y are both started with $< \text{sos} >$ and ended with $< \text{eos} >$
2. Transformer encoder: outputs contextualized embeddings $\widetilde{\boldsymbol{X}}$
   - $\widetilde{\boldsymbol{X}}$ is in size $|X| \times d$, where d is the embedding vector dimensions
3. Decoder self-attention: create contextualized embeddings $\widetilde{\boldsymbol{Y}}$
   - $\widetilde{\boldsymbol{Y}}$ is in size $|Y| \times d$
4. Decoder attention: $\widetilde{\boldsymbol{Y}}$ vectors attend to $\widetilde{\boldsymbol{X}}$ vectors ($\widetilde{\boldsymbol{Y}}$ queries, $\widetilde{\boldsymbol{X}}$ values) , **while masking future tokens**
   - Result matrix is in size $|Y| \times d$
5. Prediction: Transformer decoder results are used to calculate $\widehat{\boldsymbol{Y}}$, the probability distributions of next tokens
   - $\widehat{\boldsymbol{Y}}$ is in size $|Y| \times |\mathbb{V}_d|$
   - E.g., vector $\widehat{\boldsymbol{y}}^{(2)}$ predicts the probability distribution of tokens at the position 3
6. Loss: is calculated using Negative Log Likelihood of the actual next words
   - E.g., based on the probability value of token $y^{(3)}$ in vector $\widehat{\boldsymbol{y}}^{(2)}$

Seq2seq with Transformers – summary

- In seq2seq with Transformers, encoding of input sequence is done in a single computation (non-autoregressive)
- Decoding of output sequence during training is also non-autoregressive
- However, decoding during inference is still autoregressive (as in seq2seq with RNNs):
  - Pass the 1st token, generate the 2nd token
  - Pass the 1st and 2nd tokens, generate the 3rd
  - …
- Each Transformer encoder/decoder is a block. You can stack them several times and make the network deep!



Agenda

- Transformers
- seq2seq with Transformers
- **BERT**
- Model compression

Contextualized word embeddings with LM objective

All of these models are Transformer models

ELMo
Oct 2017
Training:
800M words
42 GPU days

GPT
June 2018
Training
800M words
240 GPU days

BERT
Oct 2018
Training
3.3B words
256 TPU days
~320–560
GPU days

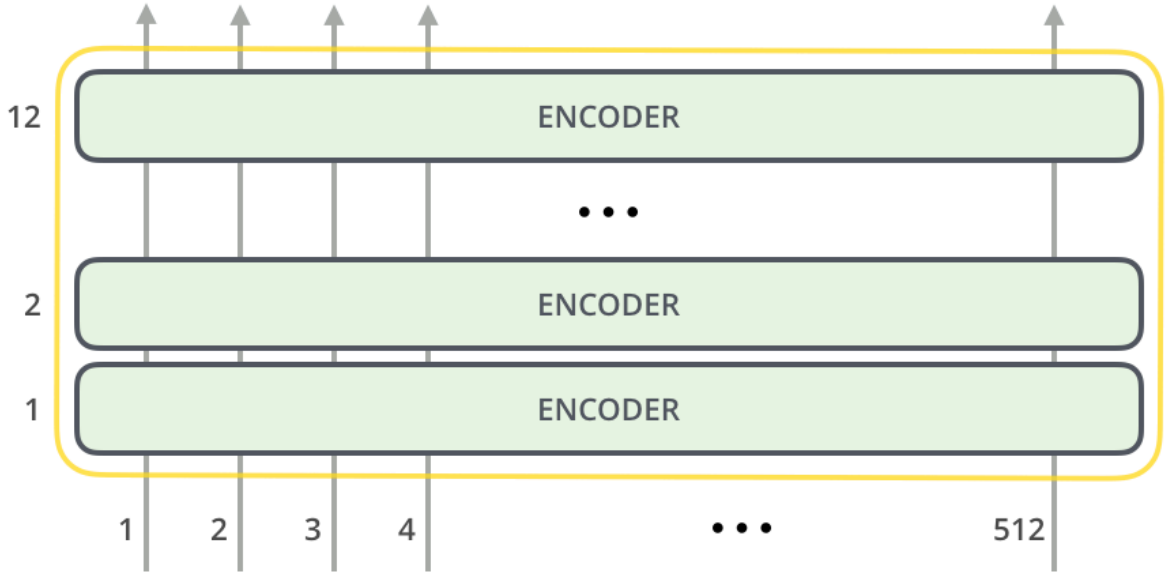
GPT-2
Feb 2019
Training
40B words
~2048 TPU v3
days according to
a reddit thread

XL-Net,
ERNIE,
Grover
RoBERTa, T5
July 2019—

Source: http://web.stanford.edu/class/cs224n/

BERT

Bidirectional Encoder Representation from Transformers



- BERT: multi-layer Transformer encoder
  - WordPiece for tokenization
  - Trained with Masked Language Model
  - Sentence pair encoding for relation between sequences

Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019, June). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proc. of NAACL-HLT (2019)*

WordPiece tokenization

- WordPiece is a descendent of Byte Pair Encoding, with the differences:
  - In WordPiece, selecting character pairs for merging is based on "*minimizing the language model likelihood of the training data*"*
  - WordPiece indicates internal subwords with "*##*" special symbol
    - E.g. *"unavoidable"* → [*"un"*, *"##avoid"*, *"##able"*]

- After creating the vocabulary list, tokenization (*decoding)* of a given sequence is done using MaxMatch algorithm*
  - A greedy (and fast) tokenization decoding algorithm used in BERT
  - E.g. *"natural language processing"* →

  pre-tokenization: [*"natural"*, *"language"*, *"processing"*] →

  tokens: [*"natural"*, *"lang"*, *"##uage"*, *"process"*, *"##ing"*]

* Learn the details here: "*Speech and Language Processing (3rd ed.) D. Jurafsky and J. H. Martin*", section 2.4.3. https://web.stanford.edu/~jurafsky/slp3/2.pdf

Schuster, M., & Nakajima, K. (2012). Japanese and Korean voice search.
In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*

Masked Language Model (MLM)

- So far, we define Language Modeling as the task of predicting the next word
  - **Limitation:** in this definition, we are constrained to using only left or right context, while language understanding is bidirectional!

- Masked Language Model: mask out $k\%$ of the input sequence, and then predict the masked words in output:

sequence: Jim made spaghetti for his girlfriend and he was very proud!

Input: Jim made [MASK] for his girlfriend and [MASK] was very proud!

predict: spaghetti he

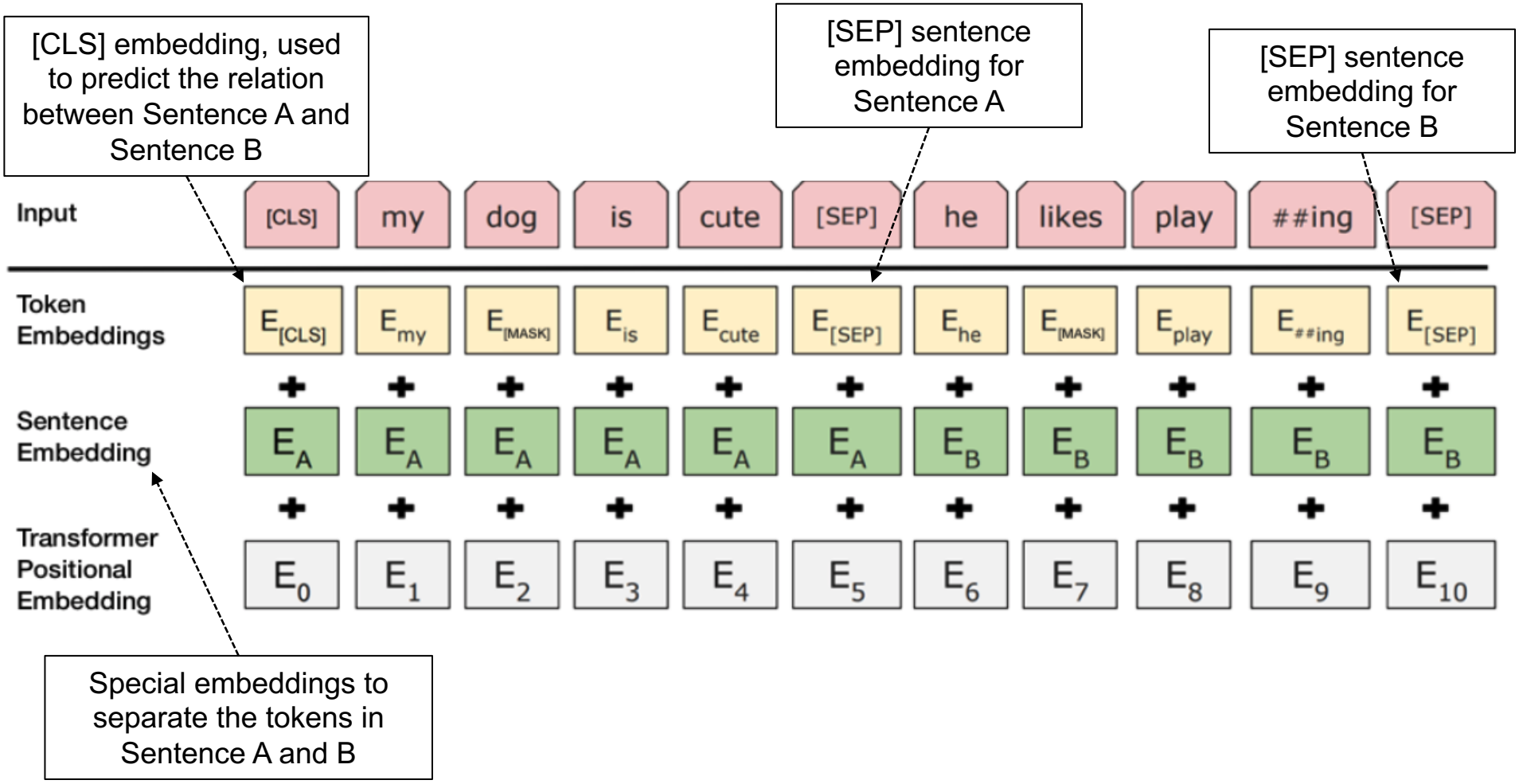
Sentence pair encoding

- Many NLP tasks require either …
  - an encoding of a sequence
    - like document classification/clustering)
  - or an estimation of the relation between two sequences
    - like question answering, information retrieval, natural language inference, paraphrasing, etc.

- BERT learns relationships between sentences by training a binary classifier over sentence pairs
  - The binary classifier predicts whether Sentence B is actual sentence that proceeds Sentence A, or a random sentence
  - The binary classifier is jointly optimized with the MLM objective
  - Each sentence is embedded by [SEP] special token
  - Relation between sentence is predicted on the output of [CLS] special token

Input embedding

The input embeddings of BERT are the sum of these three types of embeddings:



BERT Model training

- Train on Wikipedia + BookCorpus

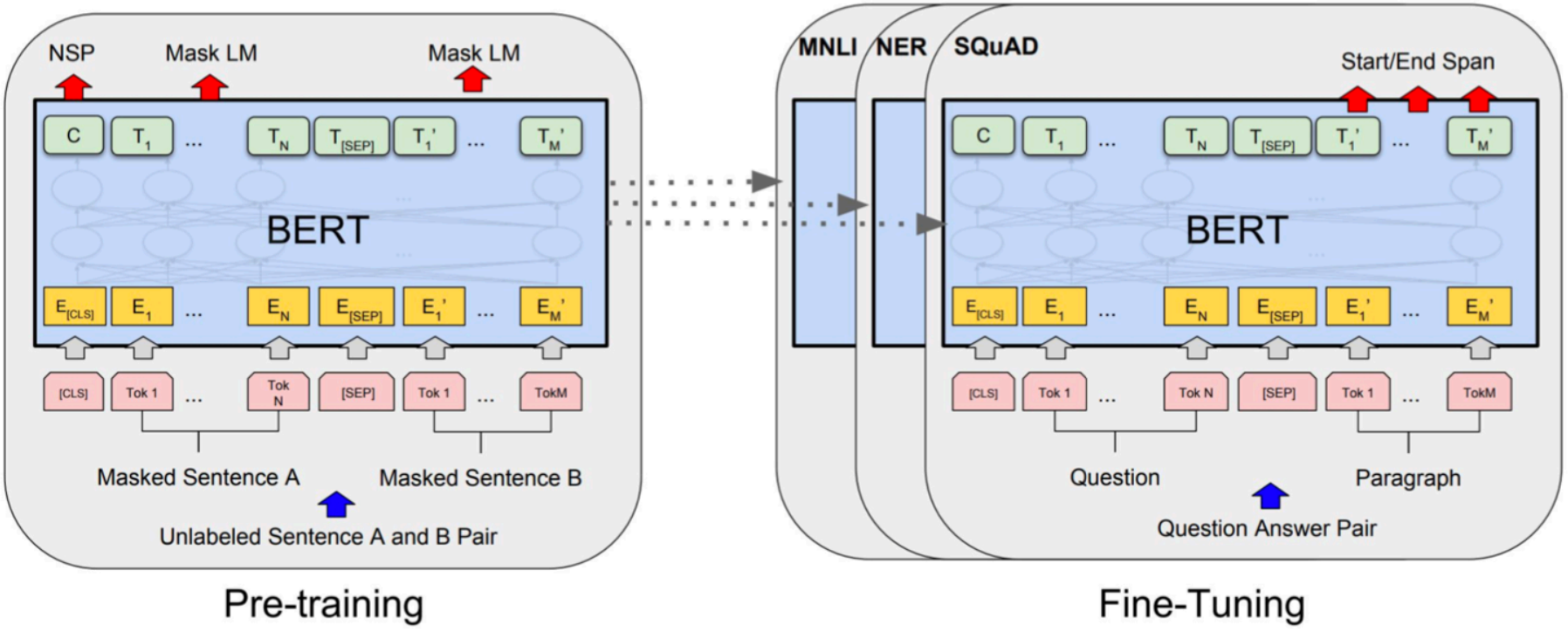
- Well-known model sizes:
  - BERT-Base: 12-layer, 768-hidden, 12-head, 110M parameters*
  - BERT-Large: 24-layer, 1024-hidden, 16-head, 340M parameters*
- Number of vocabularies around 30K (due to WordPiece)
- Fairly expensive training
  - four days on 4 to 16 Cloud TPUs

- Practical resources:
  - https://github.com/google-research/bert
  - Library to have BERT models in PyTorch: https://huggingface.co/transformers/

* For comparison, a (static) word embedding like word2vec with vocabulary size 200K and vector size 768 has 153M parameters

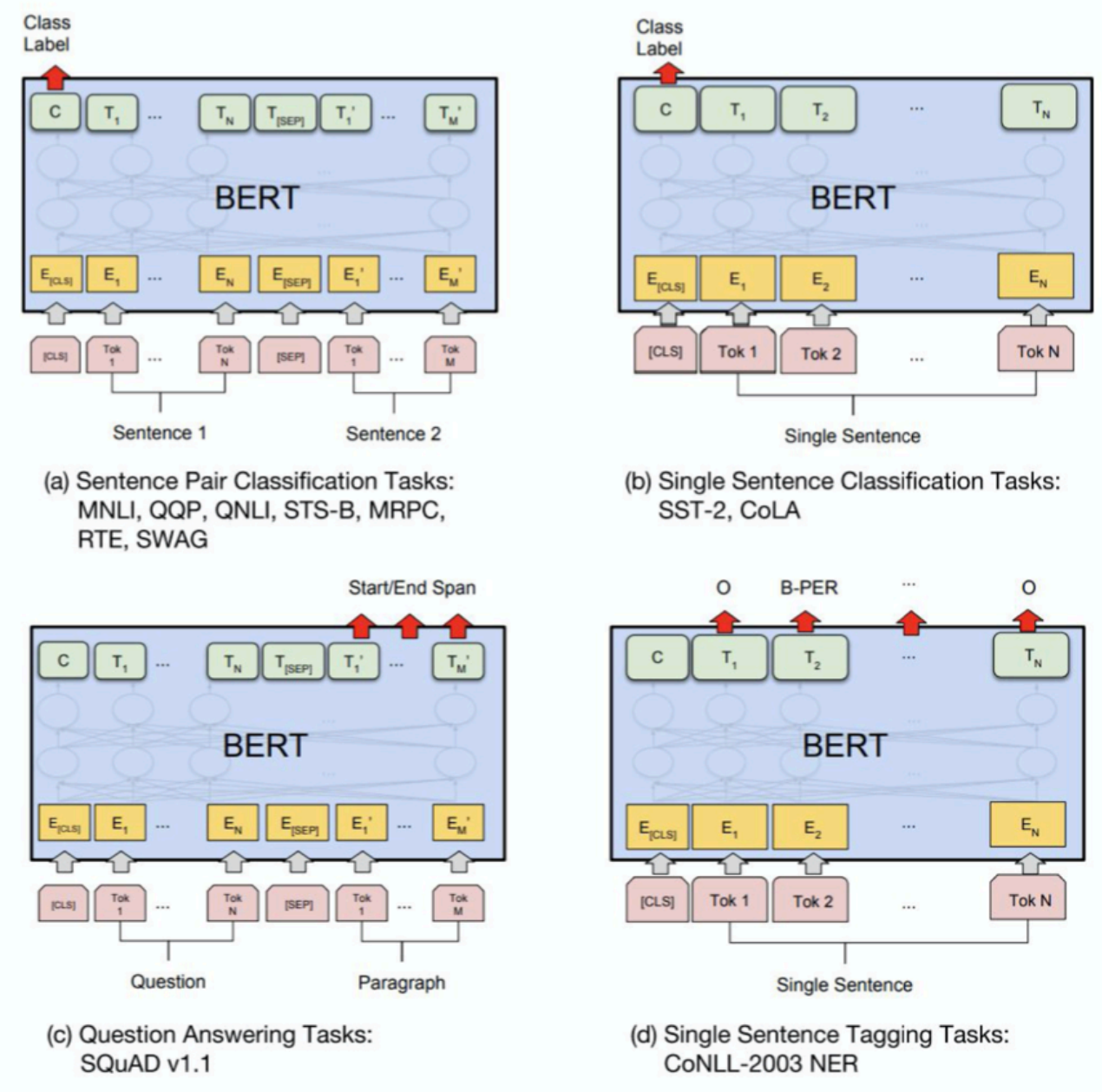
Fine tuning

- Fine tuning: in an end-task, learn a classifier on the top layer, and update all (or some) of parameters



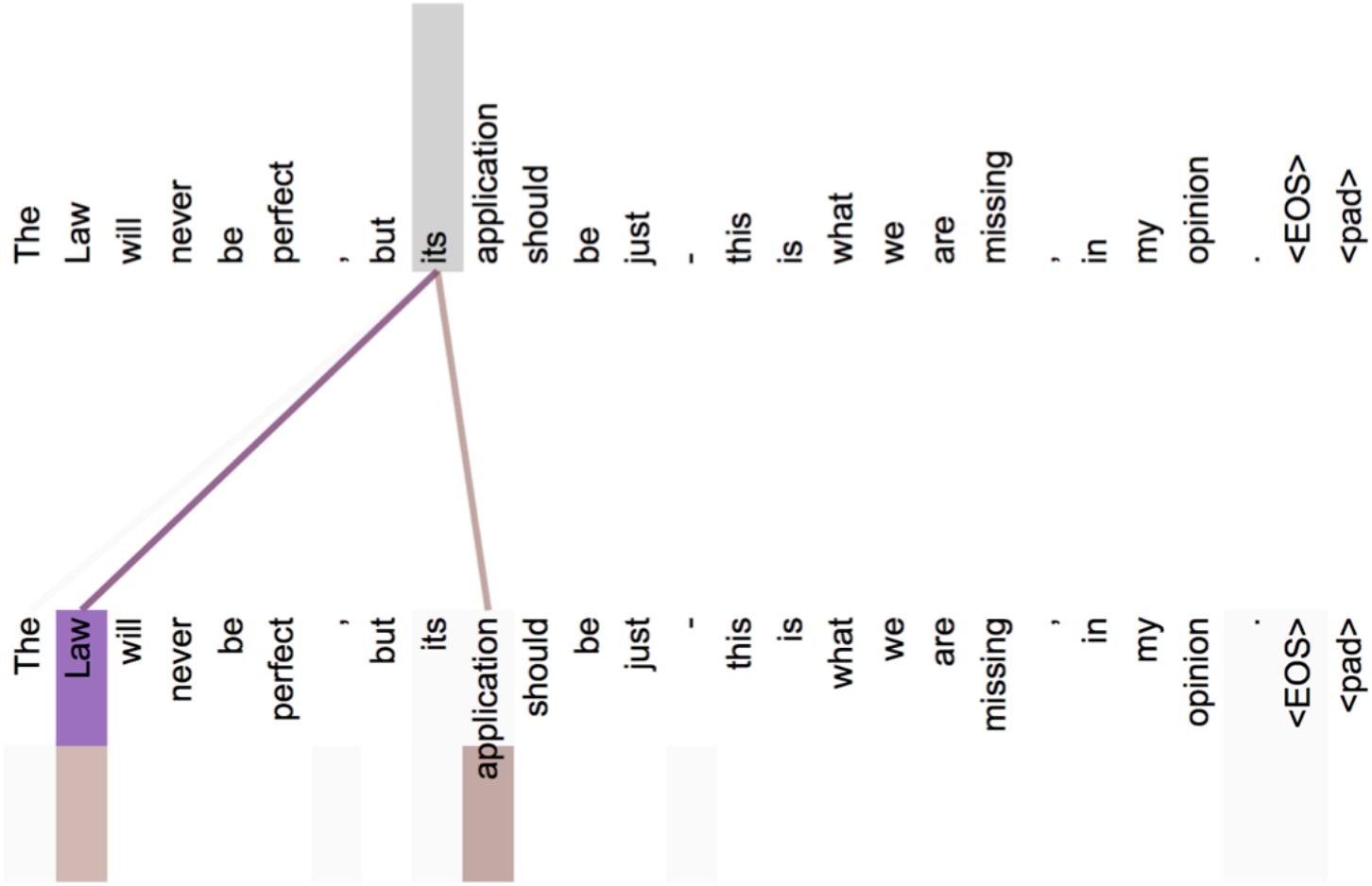
Source: http://web.stanford.edu/class/cs224n/

Fine tuning


Class Label
C
T1
TN
T[SEP]
T1'
TM'
BERT
E[CLS]
E1
EN
E[SEP]
E1'
EM'
[CLS]
Tok 1
Tok N
[SEP]
Tok 1
Tok M
Sentence 1
Sentence 2
(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG
Class Label
C
T1
T2
TN
BERT
E[CLS]
E1
E2
EN
[CLS]
Tok 1
Tok 2
Tok N
Single Sentence
(b) Single Sentence Classification Tasks: SST-2, CoLA
Start/End Span
C
T1
TN
T[SEP]
T1'
TM'
BERT
E[CLS]
E1
EN
E[SEP]
E1'
EM'
[CLS]
Tok 1
Tok N
[SEP]
Tok 1
Tok M
Question
Paragraph
(c) Question Answering Tasks: SQuAD v1.1
O
B-PER
O
C
T1
T2
TN
BERT
E[CLS]
E1
E2
EN
[CLS]
Tok 1
Tok 2
Tok N
Single Sentence
(d) Single Sentence Tagging Tasks: CoNLL-2003 NER

Multi-head attention visualization

- A selected example:



Visualization tool: https://github.com/jessevig/bertviz

Some results

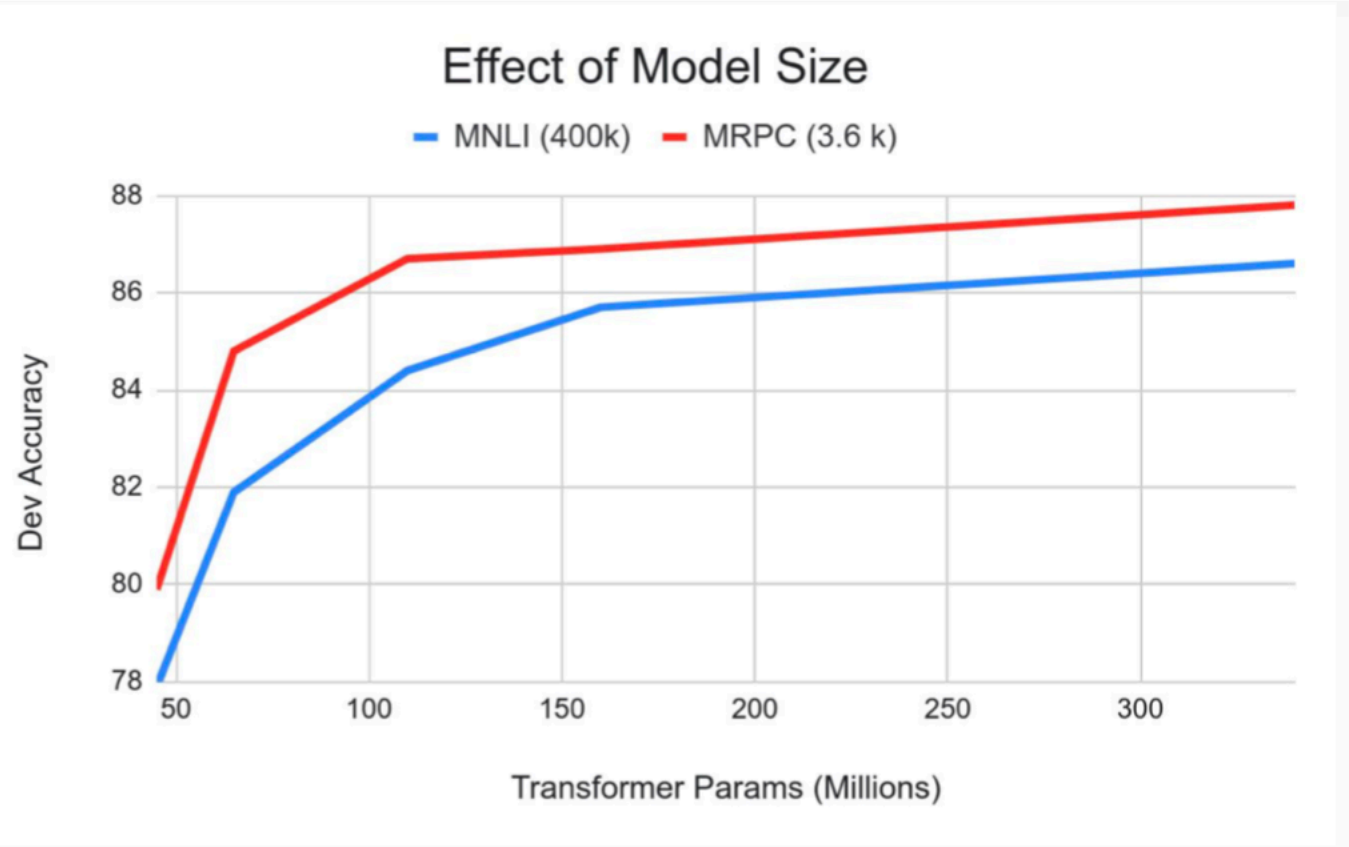
- A generic, deep, pre-trained model for NLP that can simply be plugged in (almost) every task!

System	MNLI-(m/mm) 392k	QQP 363k	QNLI 108k	SST-2 67k	CoLA 8.5k	STS-B 5.7k	MRPC 3.5k	RTE 2.5k	**Average** -
Pre-OpenAI SOTA	80.6/80.1	66.1	82.3	93.2	35.0	81.0	86.0	61.7	74.0
BiLSTM+ELMo+Attn	76.4/76.1	64.8	79.9	90.4	36.0	73.3	84.9	56.8	71.0
OpenAI GPT	82.1/81.4	70.3	88.1	91.3	45.4	80.0	82.3	56.0	75.2
$\text{BERT}_{\text{BASE}}$	84.6/83.4	71.2	90.1	93.5	52.1	85.8	88.9	66.4	79.6
$\text{BERT}_{\text{LARGE}}$	**86.7/85.9**	**72.1**	**91.1**	**94.9**	**60.5**	**86.5**	**89.3**	**70.1**	**81.9**

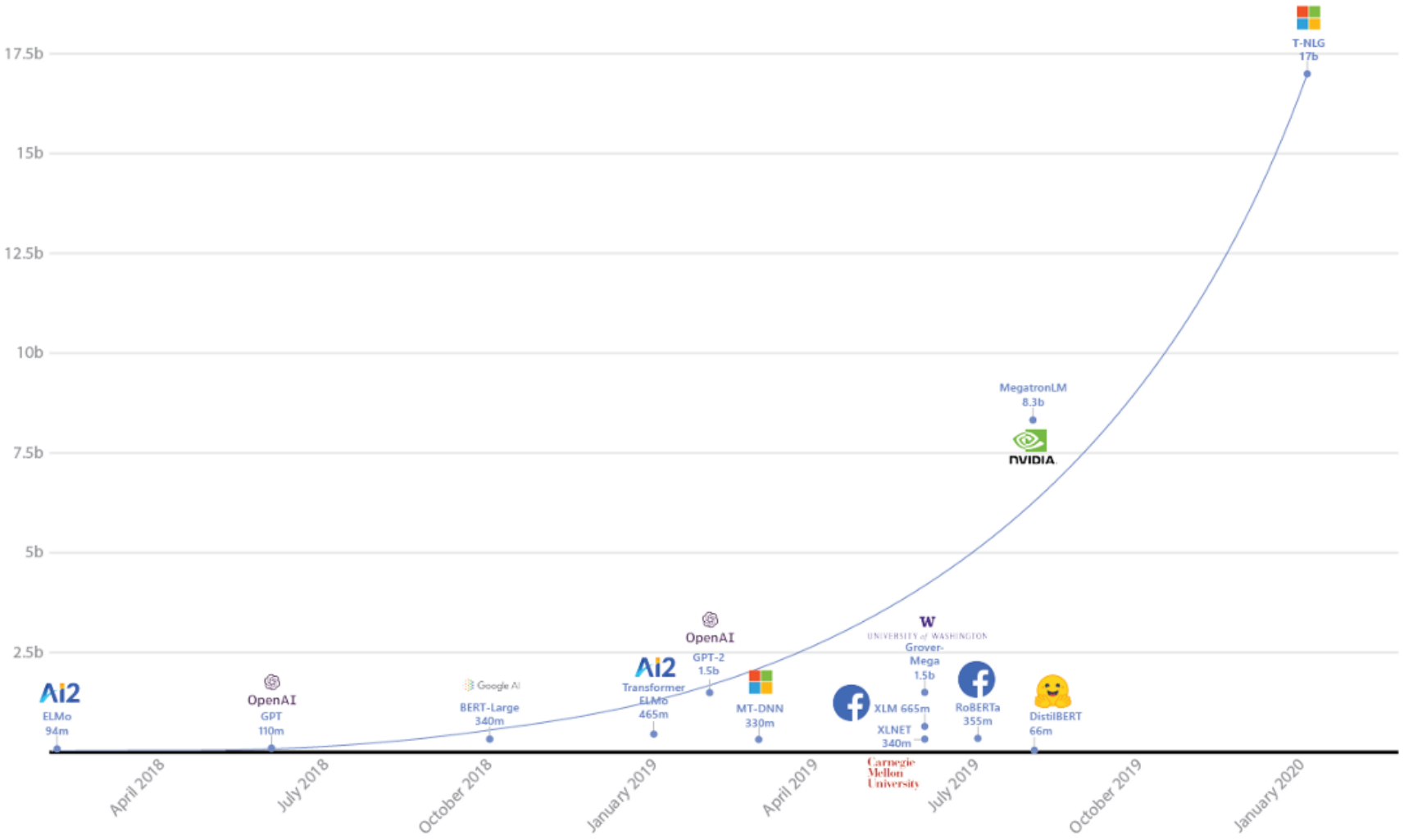
Agenda

- Transformers
- seq2seq with Transformers
- BERT
- **Model compression**

Bigger models work better!

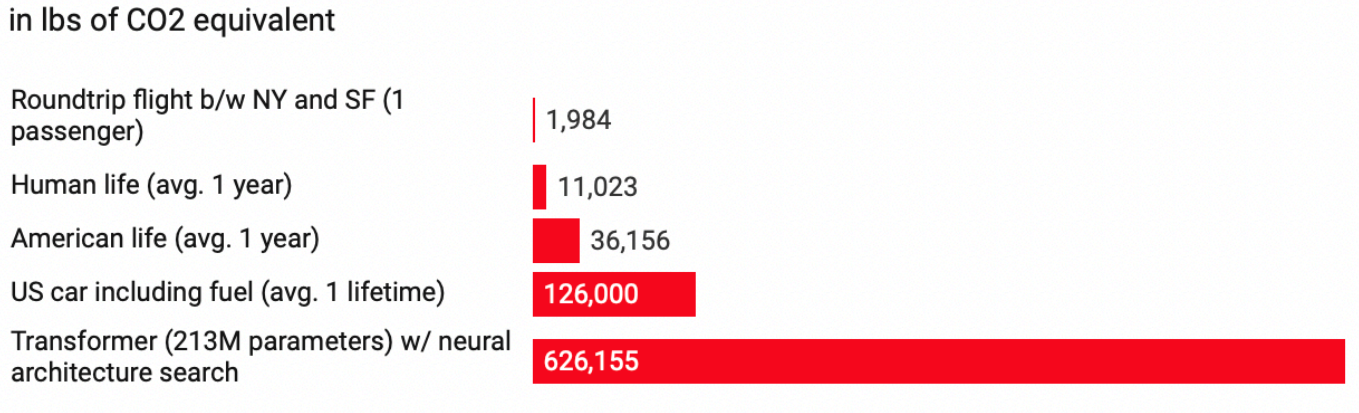


Trend!



Source: https://www.groundai.com/project/distilbert-a-distilled-version-of-bert-smaller-faster-cheaper-and-lighter/1

Carbon footprint of NLP



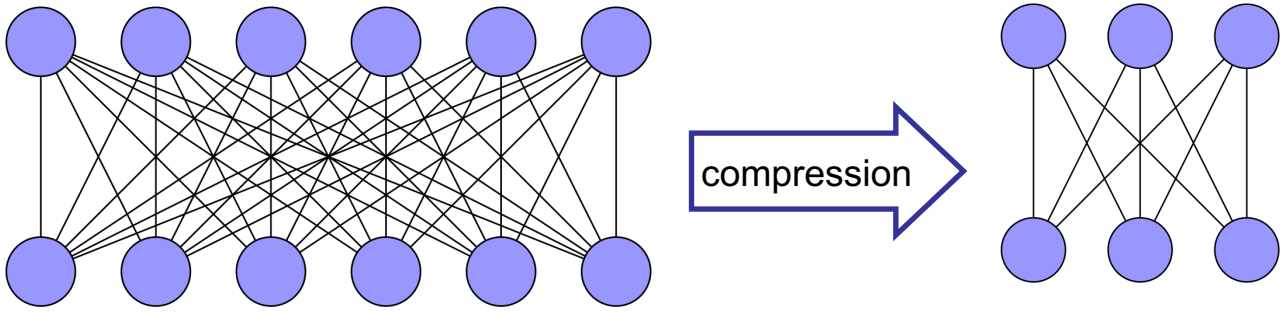
Model	Hardware	Power (W)	Hours	kWh·PUE	CO_2e	Cloud compute cost
$\text{Transformer}_{base}$	P100x8	1415.78	12	27	26	\$41–\$140
Transformer_{big}	P100x8	1515.43	84	201	192	\$289–\$981
ELMo	P100x3	517.66	336	275	262	\$433–\$1472
BERT_{base}	V100x64	12,041.51	79	1507	1438	\$3751–\$12,571
BERT_{base}	TPUv2x16	—	96	—	—	\$2074–\$6912
NAS	P100x8	1515.43	274,120	656,347	626,155	\$942,973–\$3,201,722
NAS	TPUv2x1	—	32,623	—	—	\$44,055–\$146,848
GPT-2	TPUv3x32	—	168	—	—	\$12,902–\$43,008

Strubell, E., Ganesh, A., & McCallum, A.. Energy and Policy Considerations for Deep Learning in NLP. In *Proceedings of ACL* (2019).
Source: https://www.technologyreview.com/2019/06/06/239031/training-a-single-ai-model-can-emit-as-much-carbon-as-five-cars-in-their-lifetimes/


Home
Shared Task
Organization
Program Comittee
SustaiNLP 2020
First Workshop on Simple and Efficient Natural Language Processing
Workshop at EMNLP 2020

Model compression

- Model compression methods reduce the size of a neural model in the sense of memory size
  - Usually applied as a post-processing step, but can also be done during training
- A compressed model
  - Less energy consumption, good for climate!
  - Efficient in practice:
    - Usually faster inference time
    - better suited to low-resource settings (i.e. on mobile phones)

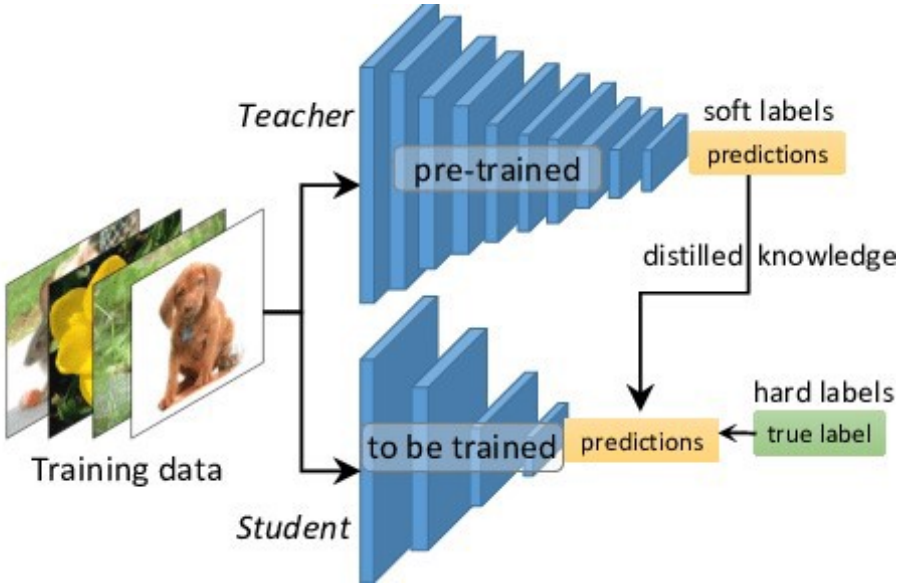


Model compression methods

- Knowledge distillation
  - A smaller model (student) is trained to reproduce the *behavior* of a larger model (teacher)
  - Student *mimics* teachers output or internal representations

Example: DistilBERT

- Distillation loss is defined according to the prediction probabilities of a pre-trained BERT
- Reduce the size to 40% while retaining 97% of performance on GLUE tasks



Sanh, V., Debut, L., Chaumond, J., & Wolf, T. (2019). DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*.
Figure source: https://towardsdatascience.com/knowledge-distillation-simplified-dd4973dbc764

Model compression methods

- Pruning
  - To reduce the extent of a network by removing the superfluous and unnecessary *neurons*, *nodes*, *heads*, *etc.*
  - Pruning can be done after training or during training

A common (post-processing) procedure

1. Train the model
2. Remove the unnecessary units, selected based on their
   - magnitudes, gradients, activations, etc.
3. Fine-tune the pruned network
4. Repeat the last two steps iteratively

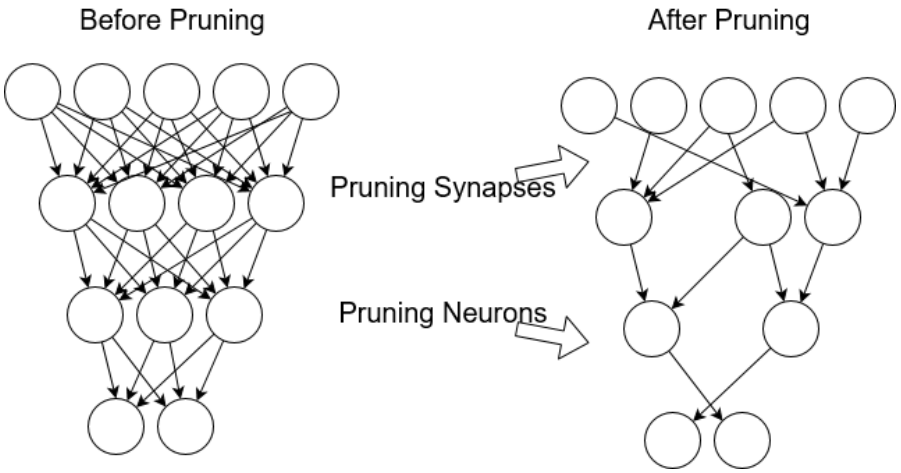


Figure source: https://software.intel.com/content/www/us/en/develop/articles/compression-and-acceleration-of-high-dimensional-neural-networks.html

Model compression methods

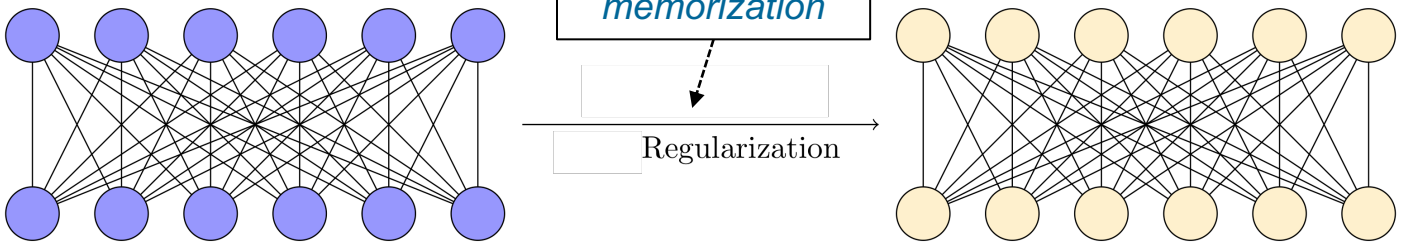
- Quantization
  - Quantization methods decrease the numerical precision of model parameters
    - For instance by turning the 32-bit float parameters of a pre-trained model to 8-bit integers

Over-parameterization – a paradox!

Create a model with much higher capacity

Apply extreme regularization to avoid *data memorization*

Trained model with "simple" optimization (i.e. SGD)



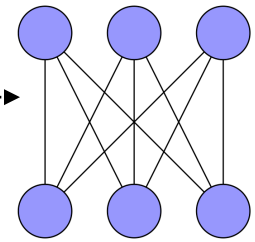
Regularization

Over-Parameterize

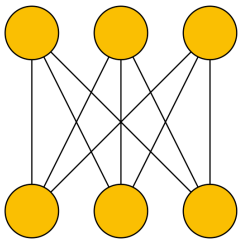
The trained model can be compressed to the appropriate size model!

Model Compression

A model with appropriate capacity that *in principle* can solve the problem



The compressed model with even up to 10% of the size of the trained model without any/little loss in test performance!



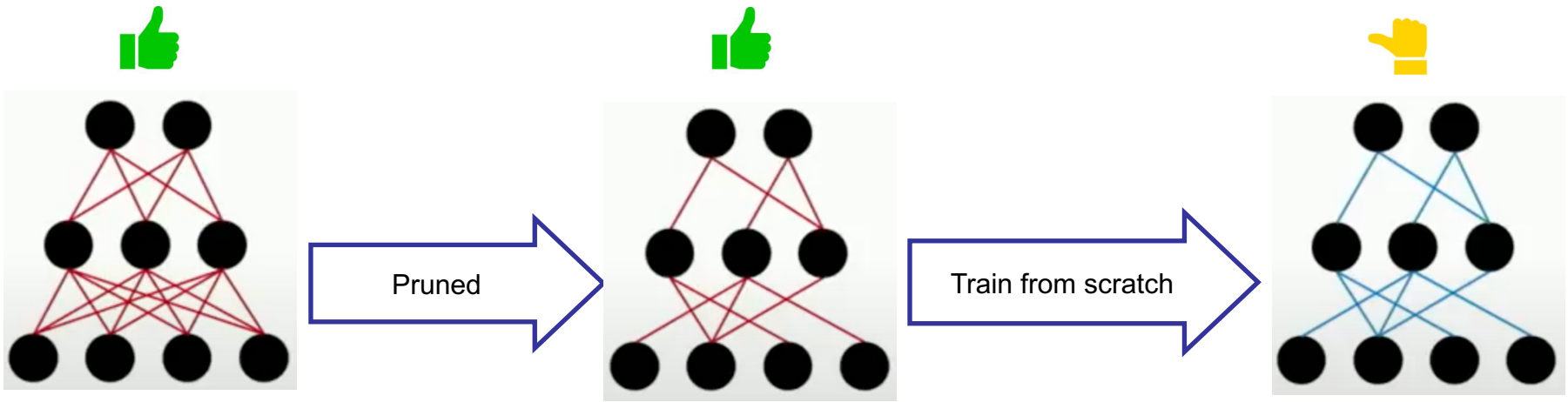
Over-parameterization is in paradox to classical ML doctrine of *"Thou shalt start simple, and then go complex."*

Figure adopted from here (a nice read): http://mitchgordon.me/machine/learning/2020/01/13/do-we-really-need-model-compression.html

Can we then start from small compressed models?!

Consider compression with pruning…

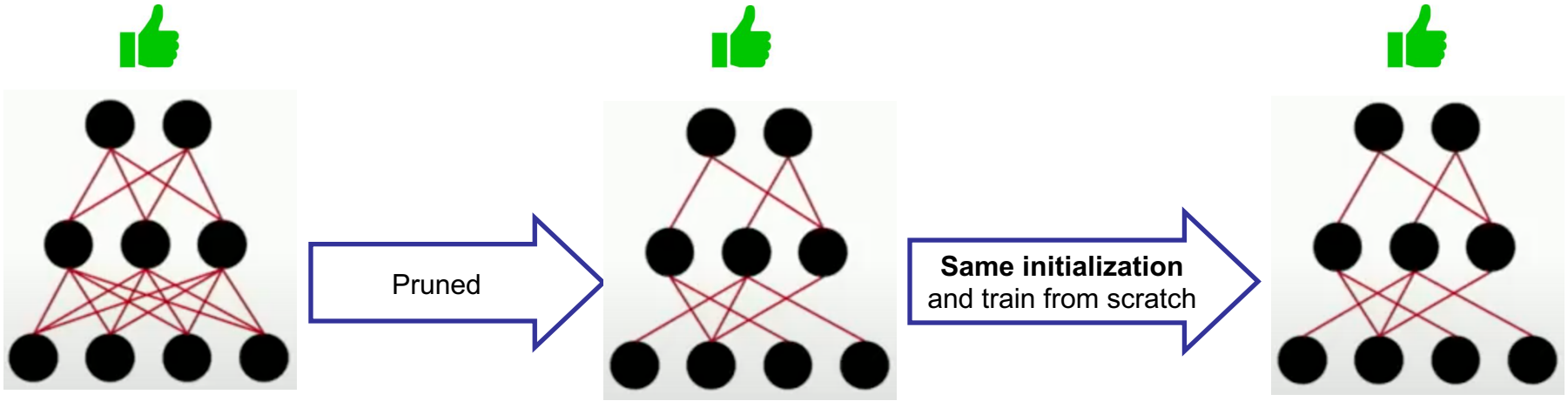
- What if we take the pruned model, and re-train it from scratch (with new initializations)?
  - It does not reach the same performance as the compressed model



Can we then start from small compressed models?!

Consider compression with pruning…

- What if we take the pruned model, and re-train it from scratch (with new initializations)?
  - It does not reach the same performance as the compressed model
- However, if we take the structure of the pruned model, and use the same initialization as the original model …
  - We achieve the same or even better results!



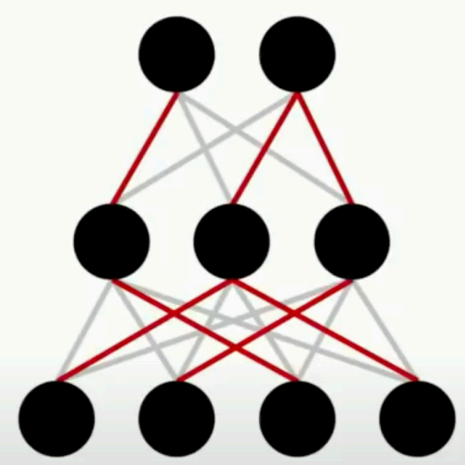
Lottery ticket hypothesis

- The Lottery Ticket Hypothesis (rephrased):

*dense, trainable networks contain sparse trainable subnetworks (i.e., winning tickets) that are equally capable**

** When trained in isolation, they reach test prediction comparable to the original network in a similar number of iterations.*

- Though, we don't know (yet) beforehand how to find these subnetworks
  - What are their structures?
  - What are their initializations?
- But if we do … we can achieve the same results by training much smaller networks



Recap

- Transformer encoder
- Seq2seq models with Transformer encoder and decoder
- BERT pre-training & fine tuning
- Over-parameterization & model compression