Language Technology
http://cs.lth.se/edan20/
Self-Attention and the Transformer

Pierre Nugues

Pierre.Nugues@cs.lth.se
http://cs.lth.se/pierre_nugues/

October 5, 2023

Transformers

After feedforward and recurrent networks, transformers are a third form of networks (in fact a kind of feedforward):

- An architecture proposed in 2017 based on the concept of **attention**
- Consists of a smart pipeline of matrices
- Transformers shown in this lecture, encoders, are trained with a **masked language model**
- They can learn complex lexical relations

Using Transformers

Goals of transformers:

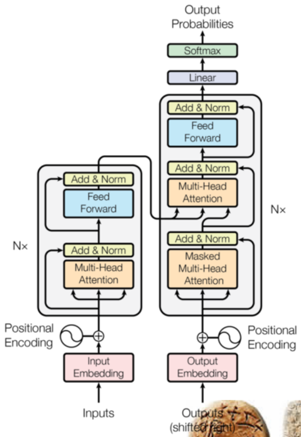
- Encapsulate a massive amount of knowledge.
- In consequence trained on very large corpora
- Sometimes marketed as the ImageNet moment (See `https://ruder.io/nlp-imagenet/`)

Transformers in practice:

- Large companies train transformers on colossal corpora, the pretrained models, requiring huge computing resources (https://arxiv.org/pdf/1906.02243.pdf)
- Mere users:
  - Reuse the models in applications
  - Fine-tune some parameters in the downstream layers

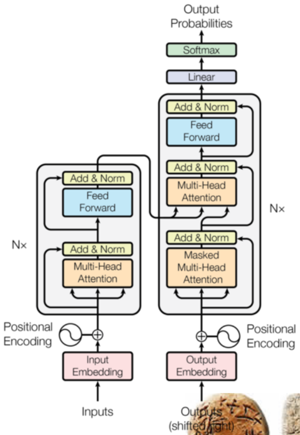
The Original Architecture

- Reference paper: *Attention Is All You Need* by Vaswani et al (2017)
  Link: `https://arxiv.org/pdf/1706.03762.pdf`
- Architecture consisting of two parts: encoder (left) and decoder (right)
- The encoder and decoder are essentially sequences of matrix multiplications with a few other functions
- Implementation in PyTorch: `https://nlp.seas.harvard.edu/2018/04/03/attention.html`



Components

- The components:
  1. Attention: The core instrument to learn complex lexical relations
  2. The encoder-decoder: The original structure. Same as with LSTM, but the recurrent structure is replaced with a feed-forward network with self-attention. The first application was for translation
  3. Encoders, the left part that can be used alone for classification tasks (BERT):
  4. Decoders, the right part that can be used to generate texts (GPT)
- We first consider attention and the encoding part



The Encoder

The encoder is a transducer: A sequence-to-sequence model with sequences of identical length.
A task exemplifying this in NLP is POS tagging.
With the phrase:

The first of their countrymen to visit Mexico...

The annotation using the UPOS tagset is:

y	Det	Adj	Adp	Pron	Noun	Part	Verb	Propn

x	The	first	of	their	countrymen	to	visit	Mexico

The **x** and **y** vectors must have the same length.

Features with a Feed-forward Network

Note the tags are from the Penn Treebank

ID	Feature vectors: $\mathbf{X}$							PPOS: $\hat{\mathbf{y}}$
	w_{i-2}	w_{i-1}	w_i	w_{i+1}	w_{i+2}	t_{i-2}	t_{i-1}	
1	BOS	BOS	**Battle**	-	tested	BOS	BOS	NN
2	BOS	Battle	**-**	tested	Japanese	BOS	NN	HYPH
3	Battle	-	**tested**	Japanese	industrial	NN	HYPH	JJ
...	...	...	...	...	...	...	...	...
19	the	first	**of**	their	countrymen	DT	JJ	IN
20	first	of	**their**	countrymen	to	JJ	IN	PRP$
21	of	their	**countrymen**	to	visit	IN	PRP$	NNS
22	their	countrymen	**to**	visit	Mexico	PRP$	NNS	TO
23	countrymen	to	**visit**	Mexico	,	NNS	TO	VB
24	to	visit	**Mexico**	,	a	TO	VB	NNP
25	visit	Mexico	**,**	a	boatload	VB	NNP	,
...	...	...	...	...	...	...	...	...
34	ashore	375	**years**	ago	.	RB	CD	NNS
35	375	years	**ago**	.	EOS	CD	NNS	RB
36	years	ago	**.**	EOS	EOS	NNS	RB	.

Two problems:

1. Embeddings have a unique association with a word (or subword)
2. Reusing the previous tags with a feed-forward architecture needs extra programming

Contextual Embeddings

Embeddings we have seen so far do not take the context into account
Attention is a way to make them aware of the context.
Consider the sentence:

> *I must go back to my ship and to my crew*
> *Odyssey, book I*

The word *ship* can be a verb or a noun with different meanings, but has only one GloVe embedding vector
Compare:

> *We process and ship your order in the most cost-efficient way possible*

from an Amazon commercial page
Self-attention will enable us to compute contextual word embeddings.

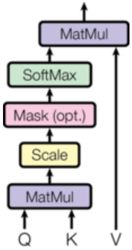
Self-Attention

In the paper *Attention is all you need*, Vaswani et al. (2017) use three kinds of vectors, queries, keys, and values. Here we will use one type corresponding to GloVe embeddings.

We compute the attention this way:

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

where d_k is the dimension of the input.



The softmax function is defined as:

$$\text{softmax}(x_1, x_2, \ldots, x_j, \ldots, x_n) = (\frac{e^{x_1}}{\sum_{i=1}^{n} e^{x_i}}, \frac{e^{x_2}}{\sum_{i=1}^{n} e^{x_i}}, \ldots, \frac{e^{x_j}}{\sum_{i=1}^{n} e^{x_i}}, \ldots, \frac{e^{x_n}}{\sum_{i=1}^{n} e^{x_i}})$$

The meaning of $QK^\intercal$

$QK^\intercal$ is the dot product of the GloVe vectors. It will tell us the similarity between the words

This is analogous to cosine similarity:

	i	must	go	back	to	my	ship	and	to	my	crew
i	1.00	0.75	0.86	0.76	0.73	0.90	0.35	0.65	0.73	0.90	0.42
must	0.75	1.00	0.85	0.68	0.87	0.69	0.42	0.69	0.87	0.69	0.45
go	0.86	0.85	1.00	0.84	0.84	0.81	0.41	0.68	0.84	0.81	0.49
back	0.76	0.68	0.84	1.00	0.83	0.76	0.49	0.77	0.83	0.76	0.51
to	0.73	0.87	0.84	0.83	1.00	0.68	0.54	0.86	1.00	0.68	0.51
my	0.90	0.69	0.81	0.76	0.68	1.00	0.38	0.63	0.68	1.00	0.44
ship	0.35	0.42	0.41	0.49	0.54	0.38	1.00	0.46	0.54	0.38	0.78
and	0.65	0.69	0.68	0.77	0.86	0.63	0.46	1.00	0.86	0.63	0.49
to	0.73	0.87	0.84	0.83	1.00	0.68	0.54	0.86	1.00	0.68	0.51
my	0.90	0.69	0.81	0.76	0.68	1.00	0.38	0.63	0.68	1.00	0.44
crew	0.42	0.45	0.49	0.51	0.51	0.44	0.78	0.49	0.51	0.44	1.00

Vaswani's attention score

The attention scores are scaled and normalized by the softmax function.

$$\text{softmax}(\frac{QK^\intercal}{\sqrt{d_k}}),$$

	i	must	go	back	to	my	ship	and	to	my	crew
i	0.36	0.05	0.07	0.05	0.04	0.19	0.01	0.02	0.04	0.19	0.01
must	0.14	0.20	0.10	0.06	0.11	0.10	0.03	0.05	0.11	0.10	0.02
go	0.18	0.09	0.14	0.09	0.08	0.13	0.02	0.04	0.08	0.13	0.02
back	0.14	0.05	0.09	0.19	0.08	0.12	0.03	0.06	0.08	0.12	0.03
to	0.11	0.11	0.09	0.09	0.15	0.08	0.04	0.07	0.15	0.08	0.03
my	0.19	0.03	0.05	0.04	0.03	0.29	0.01	0.02	0.03	0.29	0.01
ship	0.03	0.03	0.03	0.04	0.05	0.03	0.55	0.03	0.05	0.03	0.13
and	0.10	0.08	0.07	0.10	0.12	0.09	0.04	0.15	0.12	0.09	0.04
to	0.11	0.11	0.09	0.09	0.15	0.08	0.04	0.07	0.15	0.08	0.03
my	0.19	0.03	0.05	0.04	0.03	0.29	0.01	0.02	0.03	0.29	0.01
crew	0.06	0.05	0.05	0.06	0.05	0.06	0.21	0.04	0.05	0.06	0.31

Attention

We use these scores to compute the attention.

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

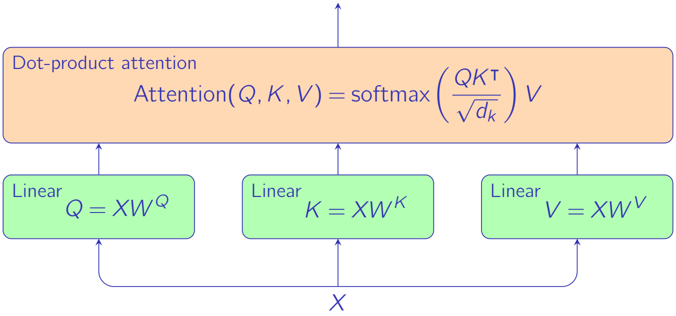
For *ship:*

```
attention_ship = (0.03 * embeddings_dict['i'] +
                  0.03 * embeddings_dict['must'] +
                  0.03 * embeddings_dict['go'] +
                  0.03 * embeddings_dict['back'] +
                  0.04 * embeddings_dict['to'] +
                  0.05 * embeddings_dict['my'] +
                  0.55 * embeddings_dict['ship'] +
                  0.03 * embeddings_dict['and'] +
                  0.05 * embeddings_dict['to'] +
                  0.03 * embeddings_dict['my'] +
                  0.13 * embeddings_dict['crew'])
```

where the *ship* vector received 13% of its value from *crew*

Code Example

Experiment: Jupyter Notebook: `ch12_attention.ipynb`
(First part of the notebook)


Dot-product attention
$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\intercal}{\sqrt{d_k}}\right) V$
Linear $Q = XW^Q$
Linear $K = XW^K$
Linear $V = XW^V$
X

Multihead Attention (I)

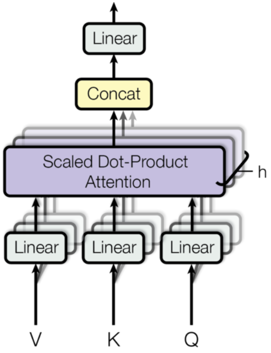
This attention is preceded by dense layers:

- If X represents complete input sequence (all the tokens), we have:

$$\begin{aligned} Q &= XW^Q, \\ K &= XW^K, \\ V &= XW^V. \end{aligned}$$

- And followed by another dense layer: W_O

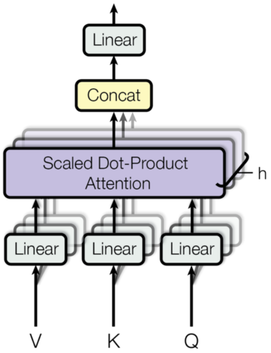
The dimension of the input (the size of the embeddings) is denoted d_{model}, for instance 100



From *Attention is all you need*, Vaswani et al. (2017)

Multihead Attention (II)

- Most architectures have parallel attentions with h outputs (called heads)
- The h heads identify and isolate different kinds of information
- We recombine them with a concatenation
- This corresponds to the two last modules concat and linear (W^O)
- To keep the same output dimension, W^Q, W^K, and W^V have a size of $(d_{model} \times d_{model}/h)$

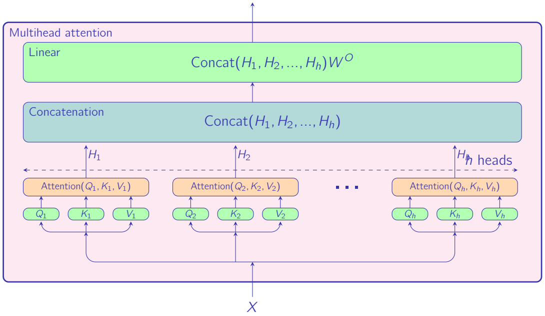


From *Attention is all you need*, Vaswani et al. (2017)

Code Example

PyTorch has an implementation of this architecture with the `https://pytorch.org/docs/stable/generated/torch.nn.MultiheadAttention.html` layer.

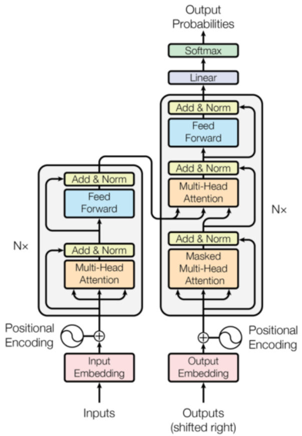
Experiment: Jupyter Notebook: `ch12_attention.ipynb` (Second part of the notebook)


Multihead attention
Linear
$\text{Concat}(H_1, H_2, ..., H_h)W^O$
Concatenation
$\text{Concat}(H_1, H_2, ..., H_h)$
H_1
H_2
H_h
h heads
$\text{Attention}(Q_1, K_1, V_1)$
$\text{Attention}(Q_2, K_2, V_2)$
...
$\text{Attention}(Q_h, K_h, V_h)$
Q_1 K_1 V_1
Q_2 K_2 V_2
Q_h K_h V_h
X

Transformers: The Encoder

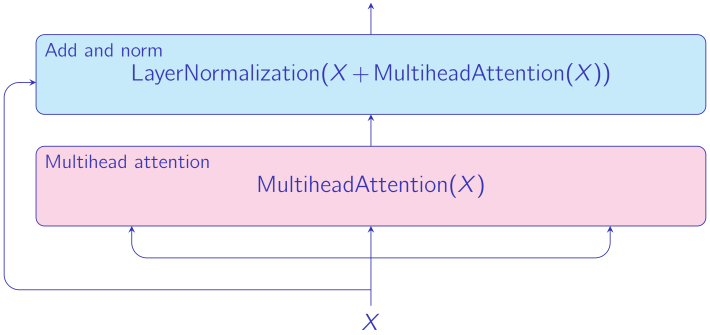
In transformers, the encoder is a structure, where:

1. The first part of the layer is a multihead attention;
2. We reinject the input to the attention output in the form of an addition:
$$X + \mathrm{Attention}(Q, K, Q).$$
This operation is called a skip or residual connection, which improves stability.
3. The result in then normalized per instance, i.e. a unique sequence, defined as:
$$x_{i,j_{norm}} = \frac{x_{i,j} - \bar{x}_{i,.}}{\sigma_{x_{i,.}}}.$$
The input distribution is more stable and improves the convergence
4. It is followed by dense layers.



Left part, from *Attention is all you need*, Vaswani et al. (2017)

Residual Networks



The residual networks are not completely understood. See
`https://arxiv.org/abs/1911.07013`

Residual Network Loss

Experimental results show a better loss profile. Source: `https://arxiv.org/abs/1712.09913`

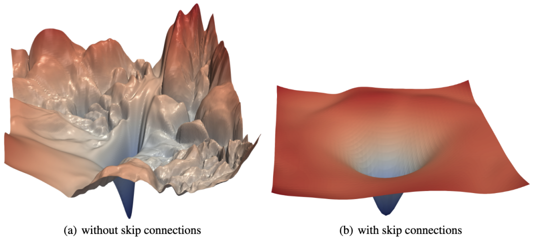
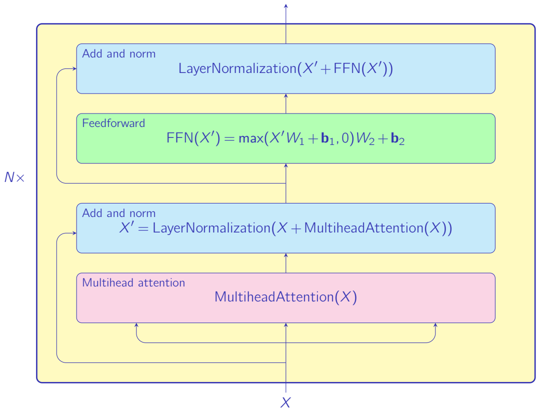


Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

The Encoder



Add and norm
$\text{LayerNormalization}(X' + \text{FFN}(X'))$
Feedforward
$\text{FFN}(X') = \max(X'W_1 + \mathbf{b}_1, 0)W_2 + \mathbf{b}_2$
$N\times$
Add and norm
$X' = \text{LayerNormalization}(X + \text{MultiheadAttention}(X))$
Multihead attention
$\text{MultiheadAttention}(X)$
X

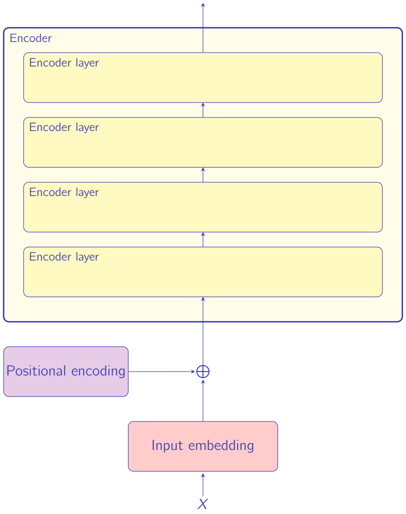
Positional Encoding

Vaswani et al. proposed two techniques to add information on the word positions. Both consist of vectors of dimension d_{model} that are summed with the input embeddings:

- The first one consists of trainable position embeddings, i.e. index i is associated with a vector of dimension d_{model} that is summed with the embedding of the input word at index i;
- The other consists of fixed vectors encoding the word positions. For a word at index i, they are defined by two functions:

$$\begin{aligned} PE(i,2j) &= \sin\left(\frac{i}{10000^{\frac{2j}{d_{model}}}}\right), \\ PE(i,2j+1) &= \cos\left(\frac{i}{10000^{\frac{2j}{d_{model}}}}\right). \end{aligned}$$

Positional Encoding



PyTorch Encoders

PyTorch has a class to create an encoder layer (https://pytorch.org/docs/stable/generated/torch.nn.TransformerEncoderLayer.html):

```
encoder_layer = nn.TransformerEncoderLayer(d_model, nheads)
enc_output = encoder_layer(input)
```

and another to create a stack of N layers (https://pytorch.org/docs/stable/generated/torch.nn.TransformerEncoder.html#torch.nn.TransformerEncoder):

```
encoder = nn.TransformerEncoder(encoder_layer, num_layers)
```

```
encoder = nn.TransformerEncoder(encoder_layer, 6)
enc_output = encoder(input)
```

Training Transformer Encoders

Transformer encoders, such as BERT, are often trained on masked language models with two tasks:

1. For a sentence, predict masked words: We replace 15% of the tokens with a specific mask token and we train the model to predict them. This is just a cloze test;
2. For a pair of sentences, predict if the second one is the successor of the first one;

Taking the two first sentences from the *Odyssey*:

> *Tell me, O Muse, of that ingenious hero who travelled far and wide after he had sacked the famous town of Troy.*
> *Many cities did he visit, and many were the nations with whose manners and customs he was acquainted;*

Masked language models

We add two special tokens: `[CLS]` at the start of the first sentence and `[SEP]` at the end of both sentences, and the token `[MASK]` to denote the words to predict.

We would have for the first task:

> *[CLS] Tell me, O Muse, of that [MASK] hero who travelled far and wide [MASK] he had sacked the [MASK] town of Troy. [SEP]*

For the second task, we would have as input:

> *[CLS] Tell me, O Muse, of that [MASK] hero who travelled far and wide [MASK] he had sacked the [MASK] town of Troy. [SEP] Many cities did he [MASK visit], and many were the [MASK nations] with whose manners [MASK and] customs he was acquainted; [SEP]*

where the prediction would return that the second sentence is the next one (as opposed to random sequences)

Positional Embeddings

BERT (the first encoder-based transformer, Devlin et al. (2019)) maps each token to three embedding vectors:

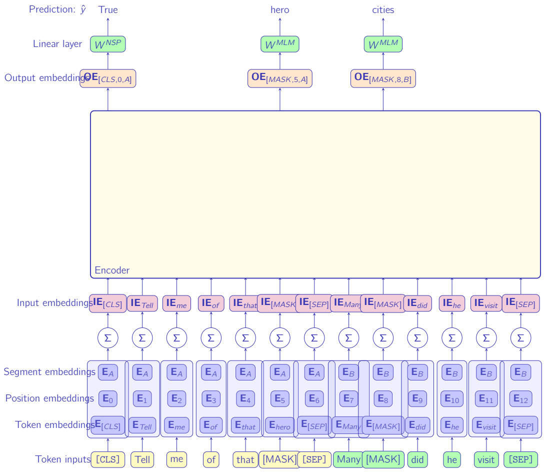
- the token embedding,
- the position of the token in the sentence (positional embeddings), and
- the segment embeddings (we will skip this part).

The three kinds of embeddings are learnable vectors.
In the BERT base version, each embedding vector has 768 dimensions.
Let us consider two sentences simplified from the *Odyssey:*

Tell me of that hero. Many cities did he visit.

Input:	[CLS]	Tell	me	of	that	hero	[SEP]	Many	cities	did	he	visit	[SEP]
Token	$\mathbf{E}_{\text{[CLS]}}$	$\mathbf{E}_{\text{tell}}$	$\mathbf{E}_{\text{me}}$	$\mathbf{E}_{\text{of}}$	$\mathbf{E}_{\text{that}}$	$\mathbf{E}_{\text{hero}}$	$\mathbf{E}_{\text{[SEP]}}$	$\mathbf{E}_{\text{many}}$	$\mathbf{E}_{\text{cities}}$	$\mathbf{E}_{\text{did}}$	$\mathbf{E}_{\text{he}}$	$\mathbf{E}_{\text{visit}}$	$\mathbf{E}_{\text{[SEP]}}$
Segment	$\mathbf{E}_A$	$\mathbf{E}_A$	$\mathbf{E}_A$	$\mathbf{E}_A$	$\mathbf{E}_A$	$\mathbf{E}_A$	$\mathbf{E}_A$	$\mathbf{E}_B$	$\mathbf{E}_B$	$\mathbf{E}_B$	$\mathbf{E}_B$	$\mathbf{E}_B$	$\mathbf{E}_B$
Position	$\mathbf{E}_0$	$\mathbf{E}_1$	$\mathbf{E}_2$	$\mathbf{E}_3$	$\mathbf{E}_4$	$\mathbf{E}_5$	$\mathbf{E}_6$	$\mathbf{E}_7$	$\mathbf{E}_8$	$\mathbf{E}_9$	$\mathbf{E}_{10}$	$\mathbf{E}_{11}$	$\mathbf{E}_{12}$

BERT Pretraining



Prediction: $\hat{y}$
True
hero
cities
Linear layer
W^{NSP}
W^{MLM}
W^{MLM}
Output embeddings
$\mathbf{OE}_{[CLS,0,A]}$
$\mathbf{OE}_{[MASK,5,A]}$
$\mathbf{OE}_{[MASK,8,B]}$
Encoder
Input embeddings
$\mathbf{IE}_{[CLS]}$ $\mathbf{IE}_{Tell}$ $\mathbf{IE}_{me}$ $\mathbf{IE}_{of}$ $\mathbf{IE}_{that}$ $\mathbf{IE}_{[MASK]}$ $\mathbf{IE}_{[SEP]}$ $\mathbf{IE}_{Many}$ $\mathbf{IE}_{[MASK]}$ $\mathbf{IE}_{did}$ $\mathbf{IE}_{he}$ $\mathbf{IE}_{visit}$ $\mathbf{IE}_{[SEP]}$
Σ
Segment embeddings
$\mathbf{E}_A$ $\mathbf{E}_A$ $\mathbf{E}_A$ $\mathbf{E}_A$ $\mathbf{E}_A$ $\mathbf{E}_A$ $\mathbf{E}_A$ $\mathbf{E}_B$ $\mathbf{E}_B$ $\mathbf{E}_B$ $\mathbf{E}_B$ $\mathbf{E}_B$ $\mathbf{E}_B$
Position embeddings
$\mathbf{E}_0$ $\mathbf{E}_1$ $\mathbf{E}_2$ $\mathbf{E}_3$ $\mathbf{E}_4$ $\mathbf{E}_5$ $\mathbf{E}_6$ $\mathbf{E}_7$ $\mathbf{E}_8$ $\mathbf{E}_9$ $\mathbf{E}_{10}$ $\mathbf{E}_{11}$ $\mathbf{E}_{12}$
Token embeddings
$\mathbf{E}_{[CLS]}$ $\mathbf{E}_{Tell}$ $\mathbf{E}_{me}$ $\mathbf{E}_{of}$ $\mathbf{E}_{that}$ $\mathbf{E}_{hero}$ $\mathbf{E}_{[SEP]}$ $\mathbf{E}_{Many}$ $\mathbf{E}_{[MASK]}$ $\mathbf{E}_{did}$ $\mathbf{E}_{he}$ $\mathbf{E}_{visit}$ $\mathbf{E}_{[SEP]}$
Token inputs
[CLS] Tell me of that [MASK] [SEP] Many [MASK] did he visit [SEP]

Model size

Transformers are trained on large corpora like the colossal clean crawled corpus (https://arxiv.org/abs/2104.08758) and encapsulate semantics found in text in the form of numerical matrices.

This results in large models (Devlin et al., 2019):

> *In this work, we denote the number of layers (i.e., Transformer blocks) as L, the hidden size as H, and the number of self-attention heads as A. We primarily report results on two model sizes: BERT$_{BASE}$ (L=12, H=768, A=12, Total Parameters=110M) and BERT$_{LARGE}$ (L=24, H=1024, A=16, Total Parameters=340M).*

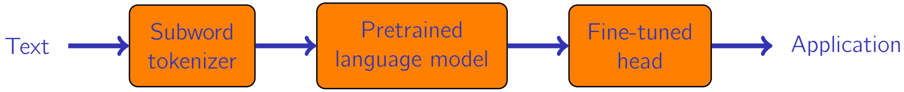
Transformers can then act as pre-trained models for a variety of tasks.

See the list from Huggingface

Finally, an interesting reading: https://sayakpaul.medium.com/an-interview-with-colin-raffel-research-scientist-at-google-5

Typical Usage

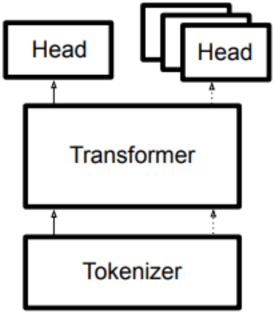
The overall trend:
Words (or subwords) $\rightarrow$ Large models (huge matrices) $\rightarrow$ Applications



- In general, language processing requires significant processing power;
- This overall resulted in massive improvements in most areas of NLP.

Applying Transformers

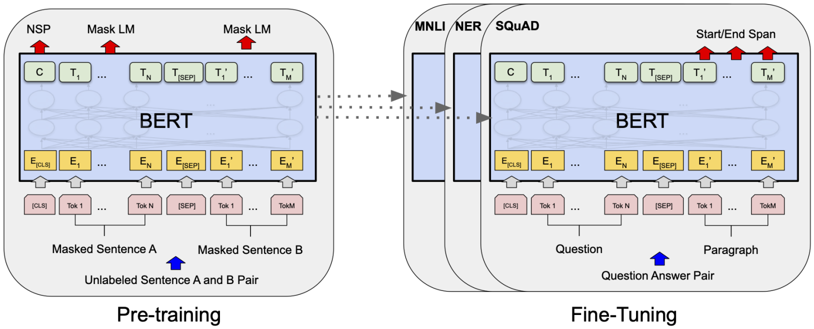
- Matrices in the transformer architecture encapsulate massive knowledge from text.
- We can apply them to tasks beyond what they have been trained for (masked language model)
- We use them as pretrained models and fine-tune a so-called dedicated *head*.
- The simplest head is a logistic regression



Picture from Wolf et al., Transformers: State-of-the-art Natural Language Processing, EMNLP Demos 2020.

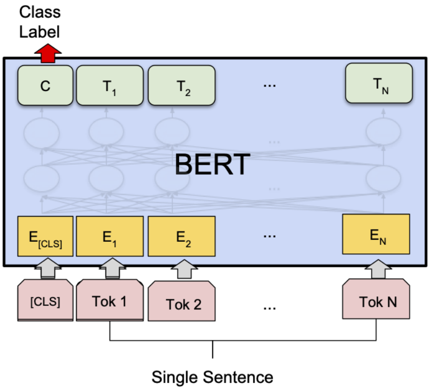
Transfer Learning

Transfer learning consists of a costly **pretraining** step and an adaptation to applications called **fine-tuning**.



from Devlin et al., *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, 2019. Note the picture comes from the second version of the paper from 2019

Application: Sentence Classification



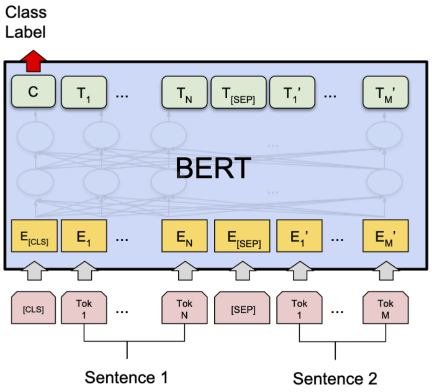
(b) Single Sentence Classification Tasks: SST-2, CoLA

from Devlin et al., *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, 2019

Code Example

Experiment: Jupyter Notebook:
https://github.com/nlp-with-transformers/notebooks/blob/main/02_classification.ipynb

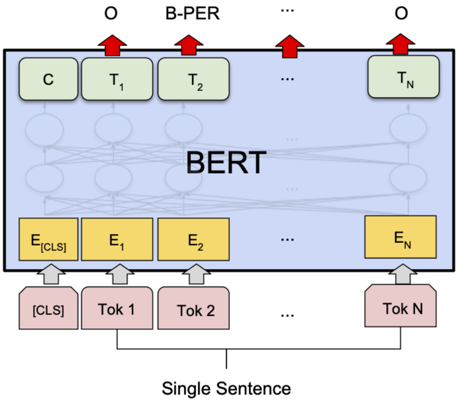
Application: Sentence Pair Classification



(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG

from Devlin et al., *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, 2019

Application: Sequence Tagging

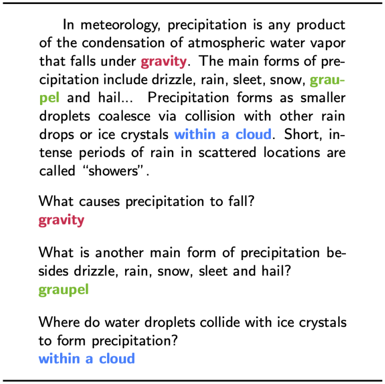


(d) Single Sentence Tagging Tasks: CoNLL-2003 NER

from Devlin et al., *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, 2019

Stanford Question Answering Dataset (SQuAD)

- Consists of 100,000 questions and paragraphs from wikipedia containing the answers
- The answer is a segment in the text (factoid QA)
- Complemented by SQuAD 2.0 with unanswerable questions
- SQuAD started an intense competition. See the impressive leaderboard `https://rajpurkar.github.io/SQuAD-explorer/`



In meteorology, precipitation is any product of the condensation of atmospheric water vapor that falls under **gravity**. The main forms of precipitation include drizzle, rain, sleet, snow, **graupel** and hail... Precipitation forms as smaller droplets coalesce via collision with other rain drops or ice crystals **within a cloud**. Short, intense periods of rain in scattered locations are called "showers".

What causes precipitation to fall?
gravity

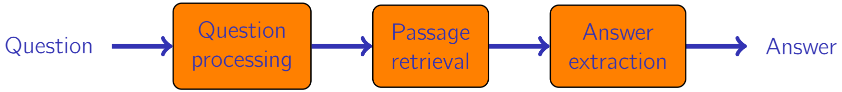
What is another main form of precipitation besides drizzle, rain, snow, sleet and hail?
graupel

Where do water droplets collide with ice crystals to form precipitation?
within a cloud

Figure 1: Question-answer pairs for a sample passage in the SQuAD dataset. Each of the answers is a segment of text from the passage.

Form Rajpurkar et al., *SQuAD: 100,000+ Questions for Machine Comprehension of Text*, 2016

Question Answering

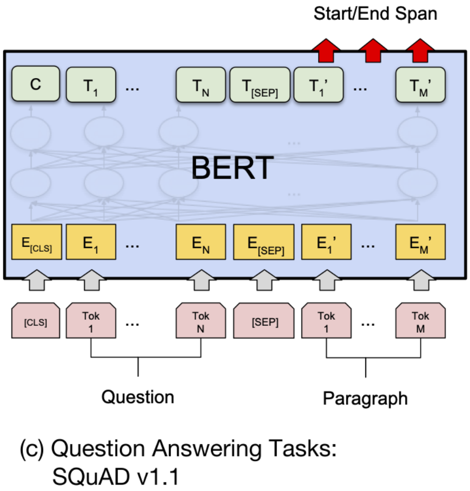


Question parsing and classification: Syntactic parsing, entity recognition, answer classification

Document retrieval. Extraction and ranking of passages: Indexing, vector space model.

Extraction and ranking of answers: Answer parsing, entity recognition

Application: Question Answering



from Devlin et al., *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, 2019

Architectures

- Original (sequence-to-sequence): T5, mT5
- Encoder (MLM): BERT, mBERT, RoBERTa, XML-RoBERTa
- Decoder (autoregressive): GPT, LLaMA

You will apply the first architecture in the next lab.
Concerning BERT, see Devlin's slides on *Contextual Word Representations with BERT and Other Pre-trained Language Models*

Evolutions in NLP

Text processing architectures have shifted from pipelines of linguistic modules to large language models fined-tuned on specific applications.
Regular benchmarks on popular applications:

- Classification and textual entailment
- Named entity recognition
- Question answering
- Translation
- Summarization and generation

See:

- GLUE (`https://gluebenchmark.com/`)
- SuperGLUE (`https://super.gluebenchmark.com/`)
- Machine translation (`https://www.statmt.org/wmt22/`)