# Deep Learning

Lecture 8: Attention models, transformers and self-supervised learning

Prof. Stéphane Gaïffas
https://stephanegaiffas.github.io

LABORATOIRE DE PROBABILITÉS STATISTIQUE & MODÉLISATION

université PARIS DIDEROT PARIS 7

Université de Paris

# Some advanced topics

Today, we'll describe some very recent deep learning architectures and techniques. Mostly useful for computer vision and NLP

## Transformers - Attention models

Architectures involving attention mechanisms

- Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention (https://arxiv.org/abs/2006.16236)
- Rethinking Attention with Performers (https://arxiv.org/abs/2009.14794)

## Self-supervised learning

Self-supervised learning based on contrastive learning

- A Simple Framework for Contrastive Learning of Visual Representations (https://arxiv.org/abs/2002.05709)
- Bootstrap Your Own Latent: A New Approach to Self-Supervised Learning (https://arxiv.org/abs/2006.07733)

# Purely attention-based architectures

# From RNNs to transformers

- RNNs used to be the state-of-the art for machine translation, time series analysis, and more generally any sequence-to-sequence task
- Then, attention was used inside recurrent layers to improve their long-range dependency
- But RNNs are hard to scale: their recurrent nature hinders distributed computations

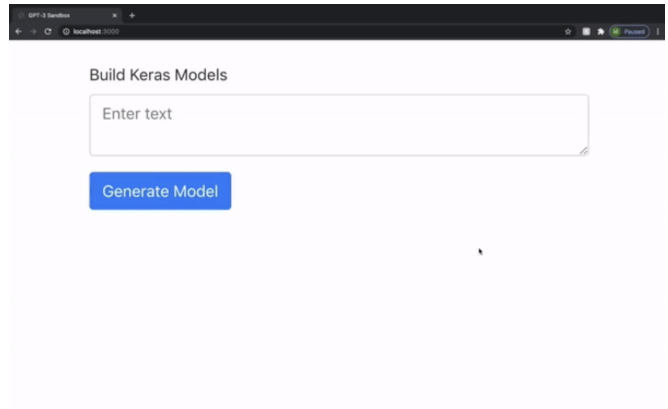
A game changer came in 2017:

- Attention is all you need by Vaswani et al. (2017) introduces the transformer architecture (https://arxiv.org/abs/1706.03762)
- Many follow-ups since then...
- Core ingredient is the Multi-Head Self-Attention layer

Led to things like

- BERT, Transformer-XL, GPT-3 (175 Billions of parameters !)

# GPT-2 and GPT-3 examples


https://app.inferkit.com/demo

# Self-attention layer

- For the first layer, input is a sequence of token embeddings

$$\mathbf{X} = [x_1, \ldots, x_L]$$

where $x_i \in \mathbb{R}^d$

- Output is a same-length sequence of (hopefully) contextualized embeddings
- For other layers, input is a sequence of contextualized embedding vectors (output of a previous self-attention layer)

# Self-attention layer

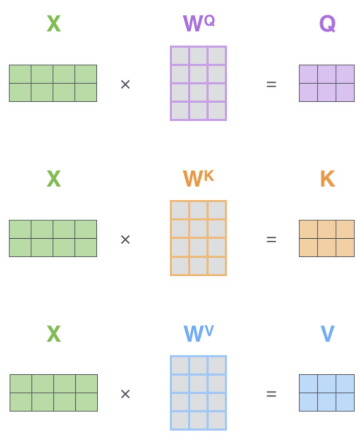
It first computes key, queries and values:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^Q, \quad \mathbf{K} = \mathbf{X}\mathbf{W}^K, \quad \mathbf{V} = \mathbf{X}\mathbf{W}^V$$

where

$$\mathbf{W}^Q \in \mathbb{R}^{d \times d_k}, \quad \mathbf{W}^K \in \mathbb{R}^{d \times d_k} \quad \text{and} \quad \mathbf{W}^V \in \mathbb{R}^{d \times d_v}$$

are learned parameters and $\mathbf{X} \in \mathbb{R}^d$ is the input

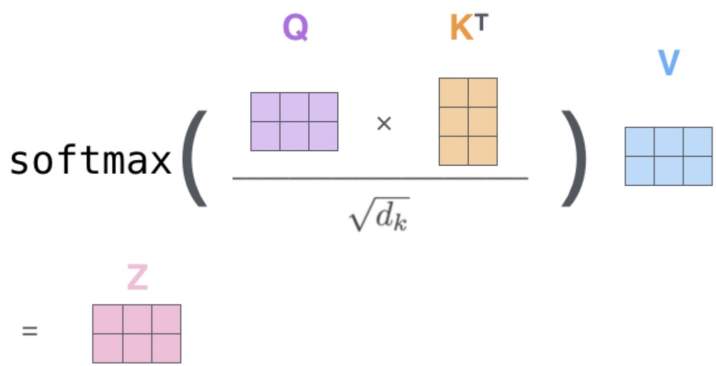



---
Credits: http://jalammar.github.io/illustrated-transformer

# Self-attention layer

And computes inner products between keys and queries and applies softmax over rows

$$\mathbf{Z} = \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$




The self-attention calculation in matrix form

———
Credits: http://jalammar.github.io/illustrated-transformer

# Multi-head self-attention layer

Combines $H$ heads of self-attention

$$\mathbf{Q}_h = \mathbf{X}\mathbf{W}_h^Q, \quad \mathbf{K}_h = \mathbf{X}\mathbf{W}_h^K, \quad \mathbf{V}_h = \mathbf{X}\mathbf{W}_h^V$$

$$\mathbf{Z}_h = \mathrm{softmax}\left(\frac{\mathbf{Q}_h\mathbf{K}_h^\top}{\sqrt{d_k}}\right)\mathbf{V}_h$$

$$\mathrm{MSA}(\mathbf{X}) = [\mathbf{Z}_1 \ \cdots \ \mathbf{Z}_H]\,\mathbf{W}^O$$

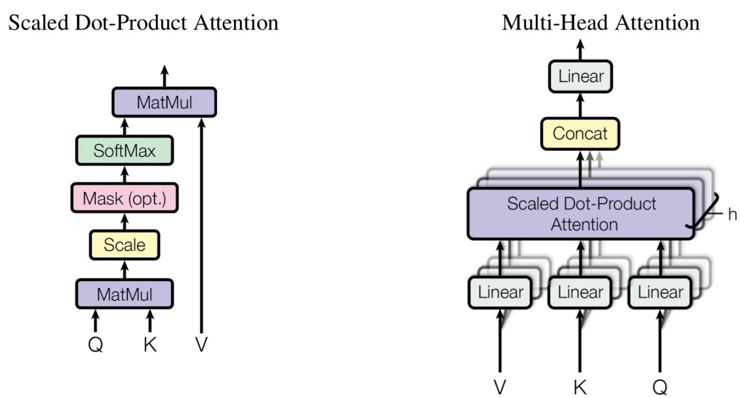
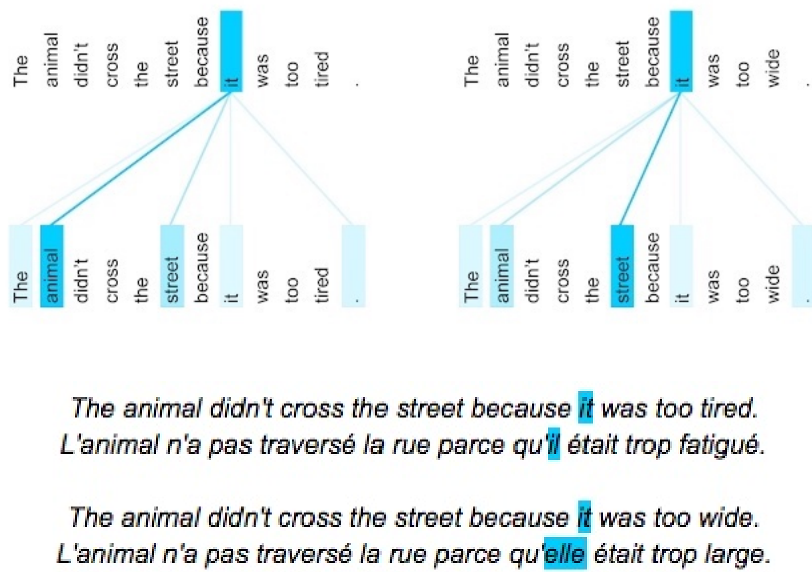



Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

# Multi-head self-attention layer

Visualization of the softmax matrix




Solves, among many others things coreference resolution (a difficult problem in machine translation)

---
Credits: https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html

# Transformer architecture

The encoder stacks several MSA layers as follows:

$$\mathbf{Y}_k = \mathrm{LayerNorm}(\mathbf{X}_k + \mathrm{MSA}(\mathbf{X}_k))$$

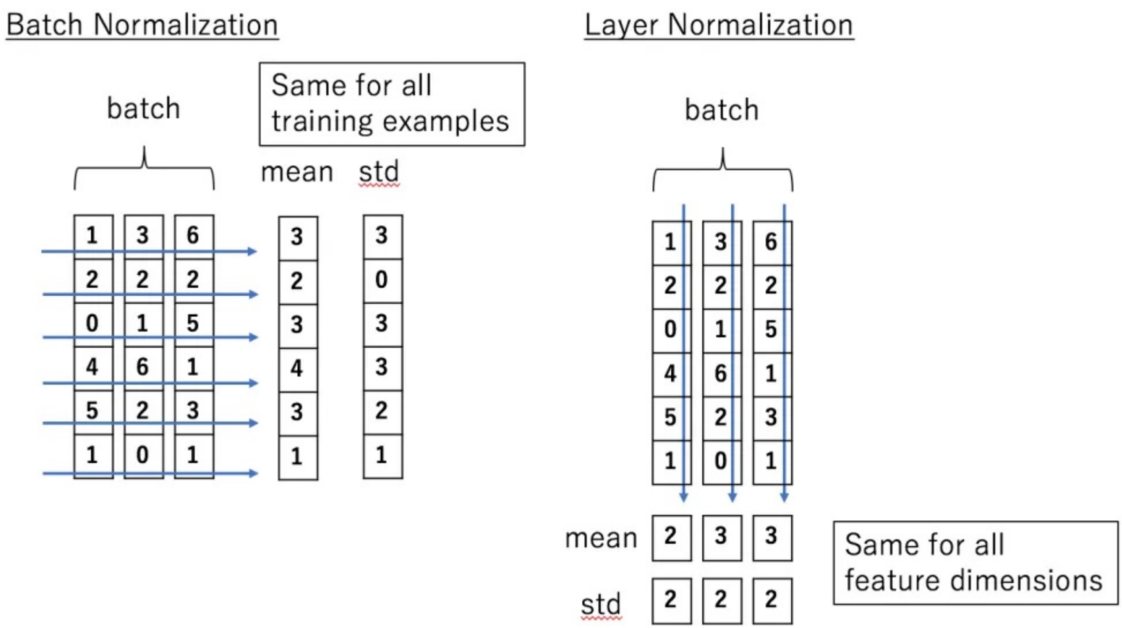
$$\mathbf{X}_{k+1} = \mathrm{LayerNorm}(\mathbf{Y}_k + \mathrm{FF}(\mathbf{Y}_k))$$

where

- $\mathrm{FF}$ is a small feed-forward network
- $\mathbf{X}_1 = \mathbf{e} =$ the sequence of $L$ token embeddings
- $\mathbf{X}_k \in \mathbb{R}^{L \times d_k}$ is the input of the $k$-th layer
- $\mathbf{X}_{k+1} \in \mathbb{R}^{L \times d_k}$ is the output of the $k$-th layer

# Transformer architecture

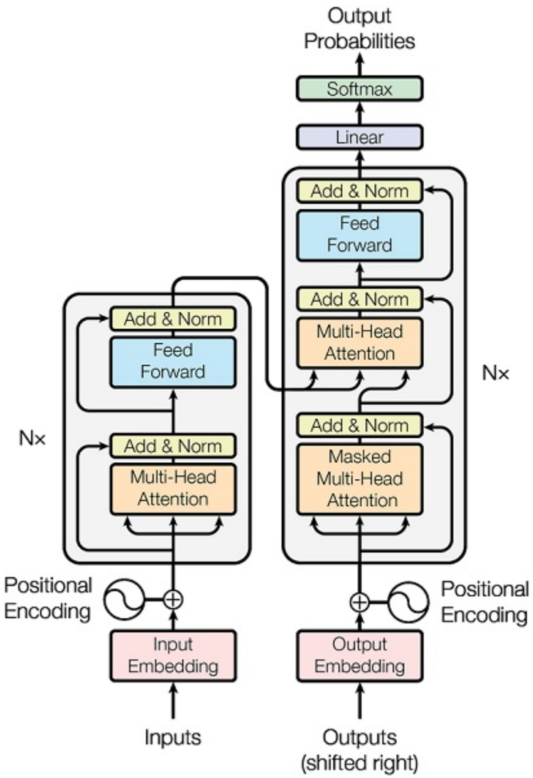
Layer normalization versus batch normalization




---
https://arxiv.org/abs/1607.06450

# Transformer architecture

Usually uses an encoder / decoder architecture



# Transformer architecture

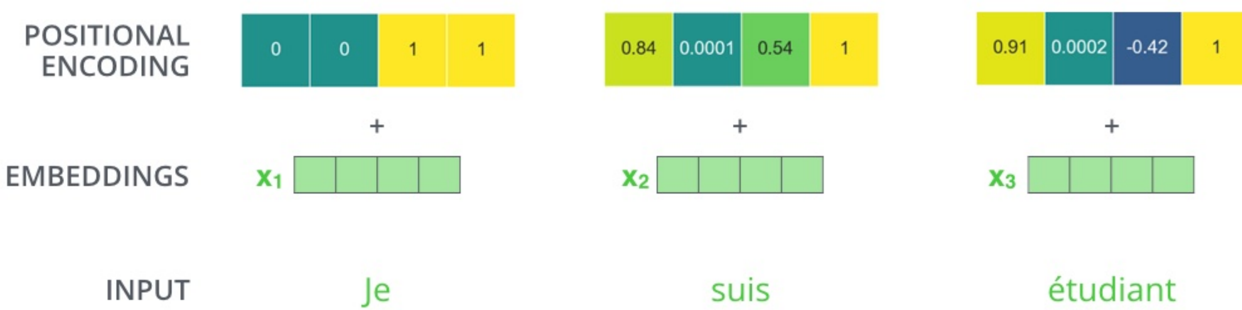
Usually uses an encoder / decoder architecture

---

From: https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html

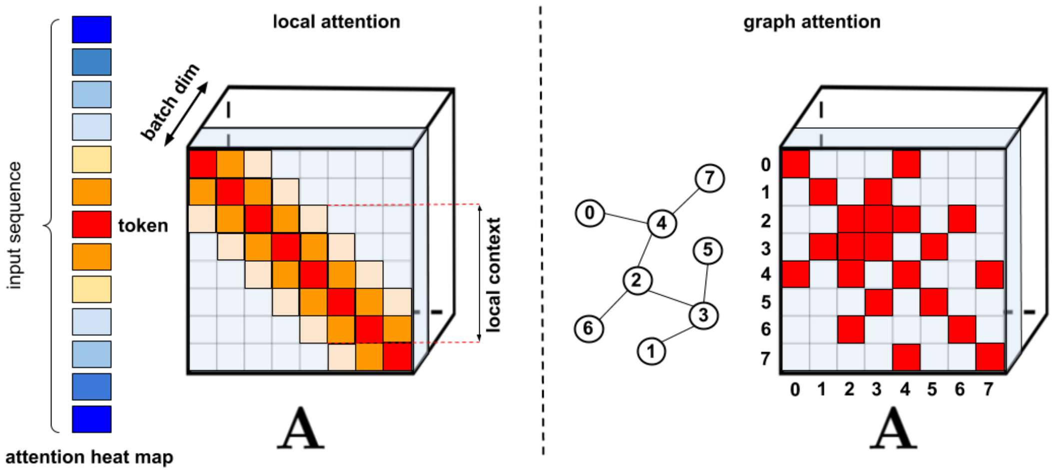
# Positional embeddings

- As such, token embeddings do not change with their position in the sequence
- A strategy is positional embeddings: either deterministic or trained
- Just add each positional embedding to each token embedding before pushing the tensor in the architecture
- Original implementation uses 512 cosines and sines
- Example with only 2 cosines and sines:



# Quadratic complexity of MSA

- The MSA layer has memory and computational complexity $O(L^2 d)$
- Huge demand of computational power and saturates GPU memory for long sequences ($L$ large)
- Some follow-up works propose strategies to solve this



# Quadratic complexity of MSA

- The MSA layer has memory and computational complexity $O(L^2d)$
- Huge demand of computational power and saturates GPU memory for long sequences ($L$ large)
- Some follow-up works propose strategies to solve this

## Graph Neural Networks and Graph Attention Networks

- Graph Attention Networks (https://arxiv.org/abs/1710.10903)
- Graph Neural Networks: A Review of Methods and Applications (https://arxiv.org/abs/1812.08434)

# Quadratic complexity of MSA

- The MSA layer has memory and computational complexity $O(L^2 d)$
- Huge demand of computational power and saturates GPU memory for long sequences ($L$ large)
- Some follow-up works propose strategies to solve this

### Graph Neural Networks and Graph Attention Networks

- Graph Attention Networks (https://arxiv.org/abs/1710.10903)
- Graph Neural Networks: A Review of Methods and Applications (https://arxiv.org/abs/1812.08434)

### Linear transformers

- Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention (https://arxiv.org/abs/2006.16236)
- Rethinking Attention with Performers (https://arxiv.org/abs/2009.14794)

# Quadratic complexity of MSA

Bottleneck is the computation of the softmax attention

$$\mathbf{Z} = \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right)\mathbf{V}$$

that we can rewrite more generally as

$$Z_i = \frac{\sum_{j=1}^{L} \mathrm{sim}(Q_i, K_j) V_j}{\sum_{j=1}^{L} \mathrm{sim}(Q_i, K_j)}$$

for $i = 1, \ldots, L$ where

$$\mathrm{sim}(q, k) = \exp\left(\frac{q^\top k}{\sqrt{d}}\right)$$

# Linear transformers

https://arxiv.org/abs/2006.16236 uses a kernel trick: replace $\mathrm{sim}(q, k)$ by

$$\mathrm{sim}(q, k) = \phi(q)^\top \phi(k)$$

for a feature mapping $\phi$. And consider in practice just a simple activation function

$$\phi(z) = \mathrm{elu}(z) + 1$$

where $\mathrm{elu}(z) = z$ if $z > 0$ and $\mathrm{elu}(z) = \alpha(e^z - 1)$ if $z < 0$

This solves the memory and computational bottlenecks because of

$$Z_i = \frac{\sum_{j=1}^{L} \phi(Q_i)^\top \phi(K_j) V_j}{\sum_{j=1}^{L} \phi(Q_i)^\top \phi(K_j)} = \frac{\phi(Q_i)^\top \sum_{j=1}^{L} \phi(K_j) V_j}{\phi(Q_i)^\top \sum_{j=1}^{L} \phi(K_j)}$$

No need to compute explicitly the attention matrix anymore !

# Linear transformers

https://arxiv.org/abs/2009.14794 uses the same kernel trick

But uses random projections

The trick relies on the following simple remark:

$$\mathrm{sim}(q,k) = e^{q^\top k} = e^{\frac{1}{2}\|q\|^2} K(q,k) e^{\frac{1}{2}\|k\|^2}$$

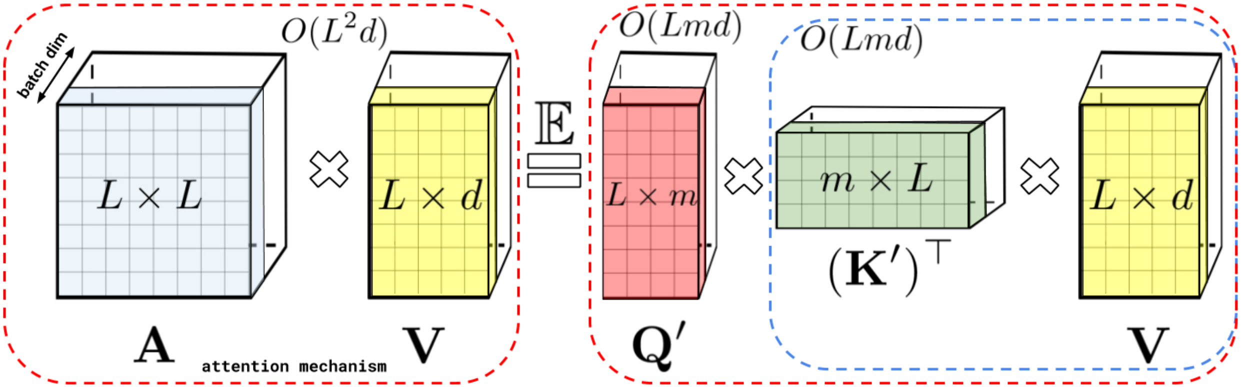
where $K(q,k) = e^{-\frac{1}{2}\|q-k\|^2}$ is the Gaussian kernel so that

$$\mathrm{sim}(q,k) = e^{-\frac{1}{2}(\|q\|^2+\|k\|^2)}\ \mathbb{E}_{\omega\sim\mathcal{N}(0,\mathbf{I}_d)}\left[e^{\omega^\top(q+k)}\right]$$

$$\approx e^{-\frac{1}{2}(\|q\|^2+\|k\|^2)}\ \frac{1}{M}\sum_{i=1}^{M} e^{\omega_m^\top(q+k)}$$

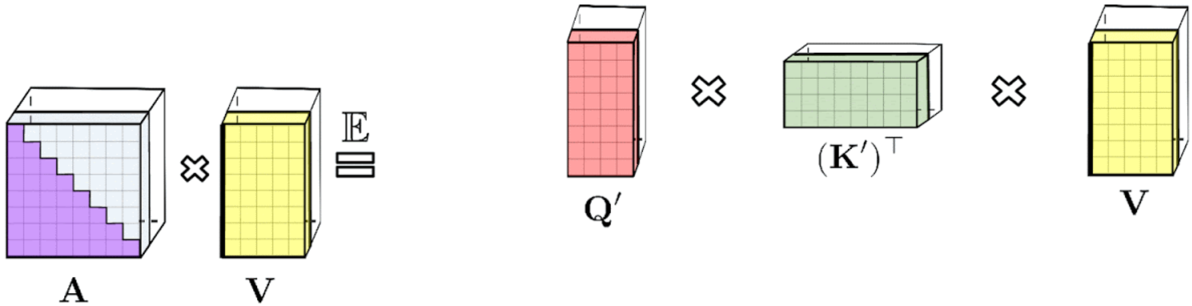
with $\omega_1,\ldots,\omega_M$ i.i.d $\mathcal{N}(0,\mathbf{I}_d)$

# Linear transformers




---
Credits: https://arxiv.org/abs/2009.14794

# Linear transformers




---
Credits: https://arxiv.org/abs/2009.14794

# Self-supervised learning

# Self-supervised learning

Self-supervised learning uses pretext tasks hence the name self-supervised. For NLP a strategy called Masked Language Modeling does the following:

- Selects 15% of the tokens at random in a sequence
- among them, replace 80% by the `MASK` token, 10% by a random code and leave the remaining 10% unchanged
- Predict the token hidden behind the `MASK` token

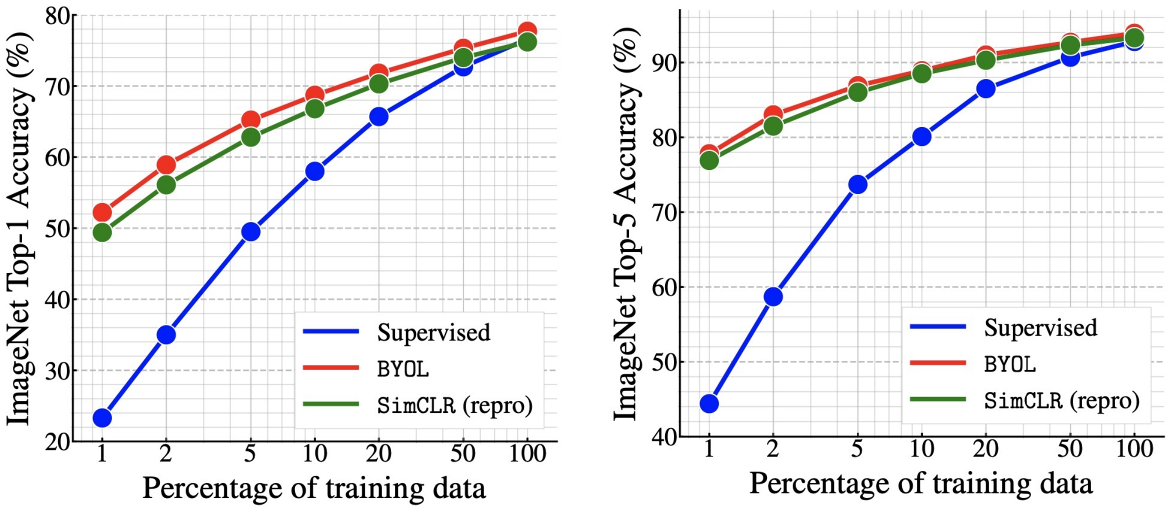
This self-supervised strategy is one of the core ingredient of BERT

- BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding (https://arxiv.org/abs/1810.04805)

Other strategies involve sequence order prediction, among others

# Self-supervised learning

Recent very impressive results in computer vision




(a) Top-1 accuracy

(b) Top-5 accuracy

Let's explain this variant :

- A Simple Framework for Contrastive Learning of Visual Representations (https://arxiv.org/abs/2002.05709)

---

From https://arxiv.org/abs/2006.07733

# Self-supervised learning

The main ingredients for self-supervised learning (SimCLR version)

- A stochastic data augmentation module. Transforms each input $x_i$ into randomly data-augmented versions $\tilde{x}_i$ and $\tilde{x}_j$. The pair $(\tilde{x}_i, \tilde{x}_j)$ is called a positive pair.

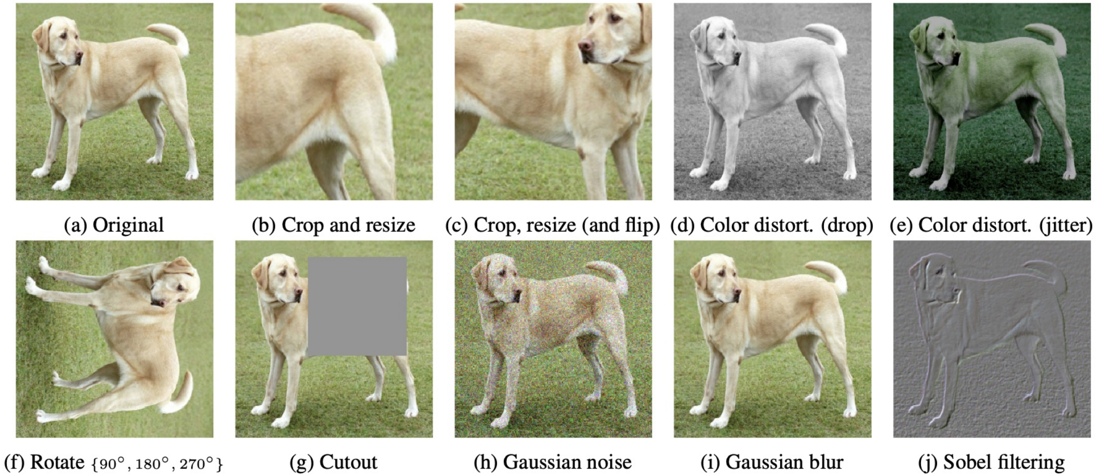



*Figure 4.* Illustrations of the studied data augmentation operators. Each augmentation can transform data stochastically with some internal parameters (e.g. rotation degree, noise level). Note that we *only* test these operators in ablation, the *augmentation policy used to train our models* only includes *random crop (with flip and resize)*, *color distortion*, and *Gaussian blur*. (Original image cc-by: Von.grzanka)

———
From https://arxiv.org/abs/2002.05709

# Self-supervised learning

The main ingredients for self-supervised learning (SimCLR version)

- An encoder $f(\cdot)$ (for instance a ResNet50) that we want to train. We compute with it $h_i = f(\tilde{x}_i)$ and $h_j = f(\tilde{x}_j)$
- A projection head $g(\cdot)$ given by a simple feed-forward network, such as a 1-hidden layer network

$$z_i = g(h_i) = \mathbf{W}^{(2)} \, \mathrm{ReLU}(\mathbf{W}^{(1)} \, h_i))$$

- Create data-augmentations pairs $\tilde{x}_{k\,k=1,...,2N}$ of the size $N$ mini-batch. On a positive pair $(i, j)$ we compute the contrastive loss

$$\ell(i, j) = -\log \left( \frac{e^{\mathrm{sim}(z_i, z_j)/\tau}}{\sum_{k=1}^{2N} \mathbf{1}_{k \neq i} e^{\mathrm{sim}(z_i, z_k)/\tau}} \right)$$

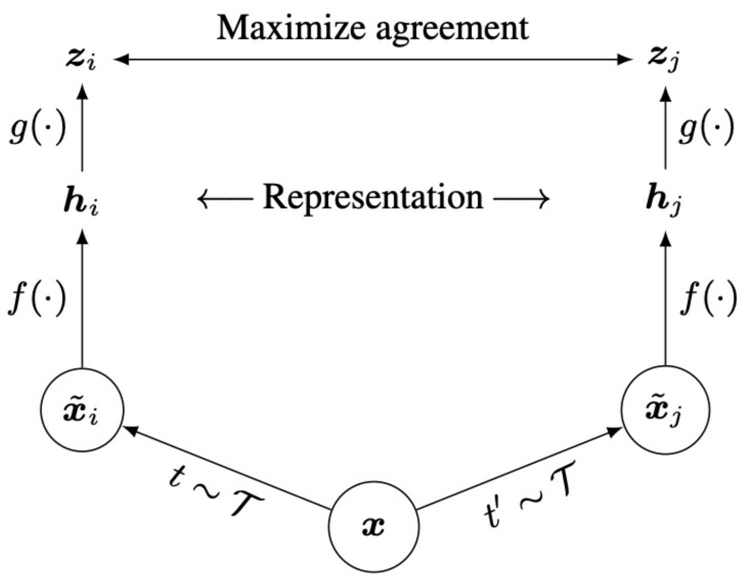
where $\mathrm{sim}(u, v) = u^\top v / \|u\| \|v\|$ is the cosine similarity

# Self-supervised learning

The main ingredients for self-supervised learning (SimCLR version)

- The loss on the data-augmented mini-batch $\tilde{x}_{k\,k=1,...,2N}$ is given by

$$\frac{1}{2N}\sum_{k=1}^{N}\left(\ell(2k-1,2k)+\ell(2k,2k-1)\right)$$




---
From: https://arxiv.org/abs/2002.05709

# Self-supervised learning

The main ingredients for self-supervised learning (SimCLR version)

# Self-supervised learning

- This version of self-supervised learning requires the use of large mini-batches
- So that enough negatives are used in the contrastive loss
- Strong computational and memory footprint

A convincing alternative approach is:

- Bootstrap Your Own Latent: A New Approach to Self-Supervised Learning (https://arxiv.org/abs/2006.07733)

# Thank You !