



Deep Learning 1

2025-2026 – Pascal Mettes

Lecture 2

AutoDiff

Where are we

Lecture	Title	Lecture	Title
1	Intro and history of deep learning	2	AutoDiff
3	Deep learning optimization I	4	Deep learning optimization II
5	Convolutional deep learning	6	Attention-based deep learning
7	Graph deep learning	8	From supervised to unsupervised deep learning
9	Multi-modal deep learning	10	Generative deep learning
11	What doesn't work in deep learning	12	Non-Euclidean deep learning
13	Q&A	14	Deep learning for videos

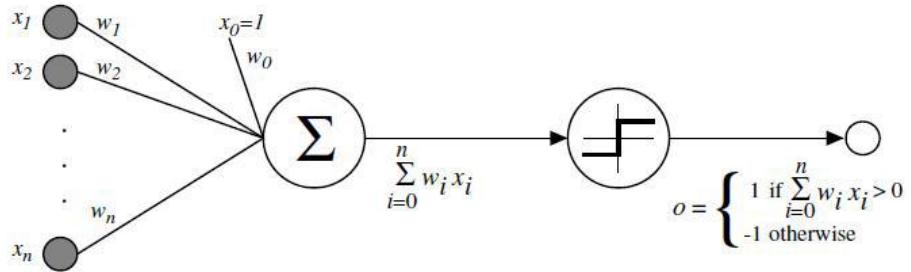
This lecture

Forward and backward propagation

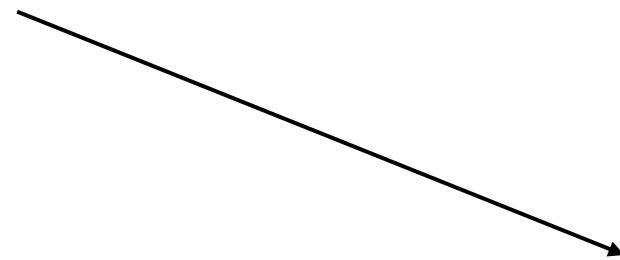
AutoDiff

AutoGrad

The story so far



Then



Now



How do deep neural networks do it?

Deep learning in one slide

A family of **parametric**, **non-linear** and **hierarchical representation learning functions**, which are **massively optimized with stochastic gradient descent** to encode **domain knowledge**, i.e. domain invariances, stationarity.

$$a_L(x; \theta_{1,\dots,L}) = h_L(h_{L-1}(\dots h_1(x, \theta_1), \theta_{L-1}), \theta_L)$$

x : input, θ_l : parameters for layer l , $a_l = h_l(x, \theta_l)$: (non-)linear function

Given training corpus $\{X, Y\}$ find optimal parameters

$$\theta^* \leftarrow \arg \min_{\theta} \sum_{(x,y) \subseteq (X,Y)} \ell(y, a_L(x; \theta_{1,\dots,L}))$$

Deep feedforward networks

A composite of functions:

$$y = f(x; \theta) = a_L(x; \theta_{1,\dots,L}) = h_L(h_{L-1}(\dots(h_1(x, \theta_1), \dots), \theta_{L-1}), \theta_L)$$

where θ_l denotes the parameters in the l -th layer.

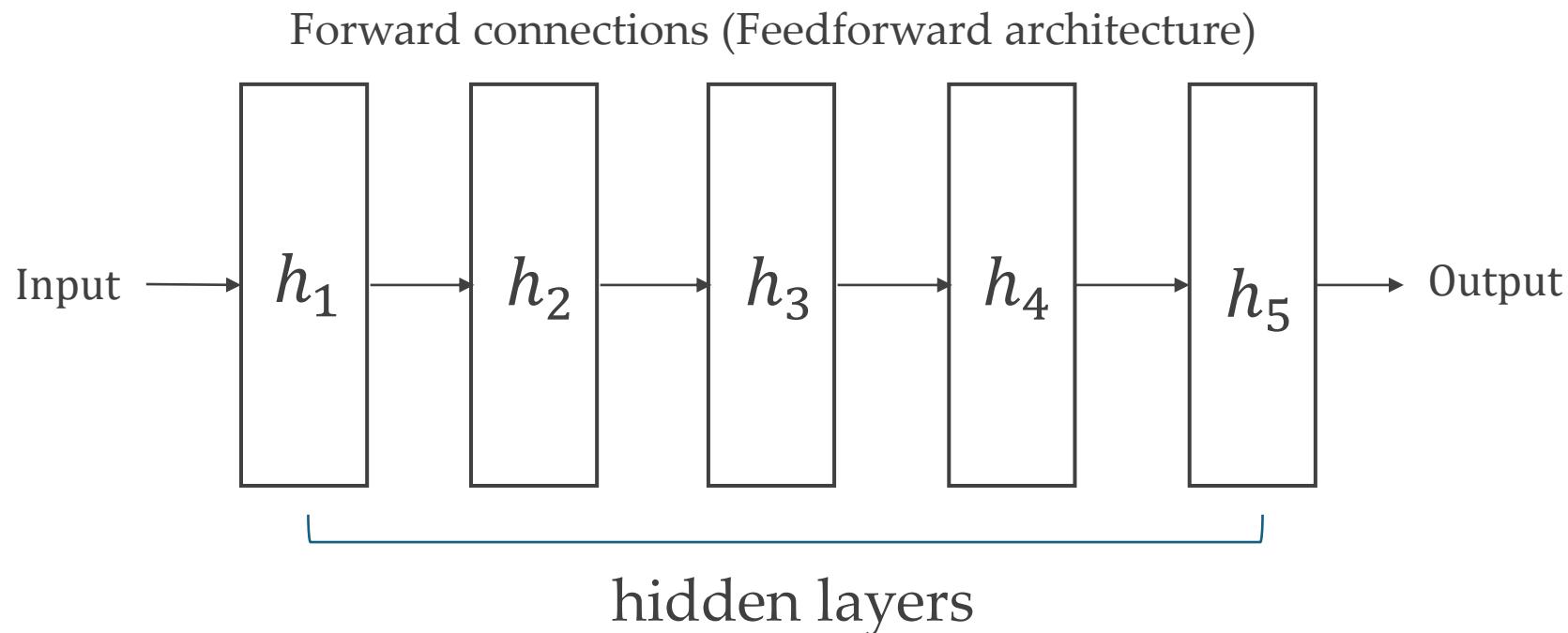
We can simplify the notation to

$$a_L = f(x; \theta) = h_L \circ h_{L-1} \circ \dots \circ h_1 \circ x$$

where each functions h_l is parameterized by parameters θ_l .

Neural networks as blocks

With the last notation, we can visualize networks as blocks:



Neural network modules

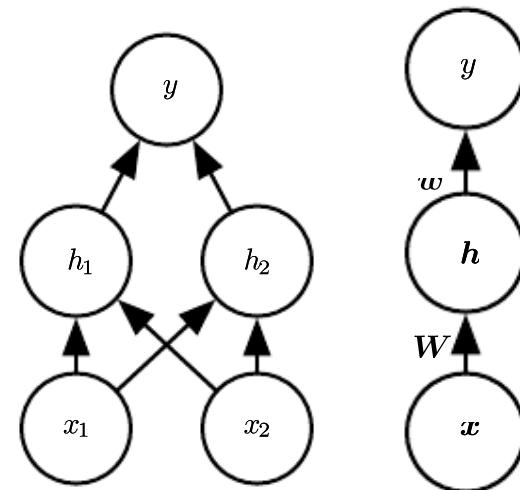
Module \Leftrightarrow Building block \Leftrightarrow Transformation \Leftrightarrow Function

A module receives as input either data x or another module's output

A module returns an output a based on its activation function $h(\dots)$

A module may or may not have trainable parameters w

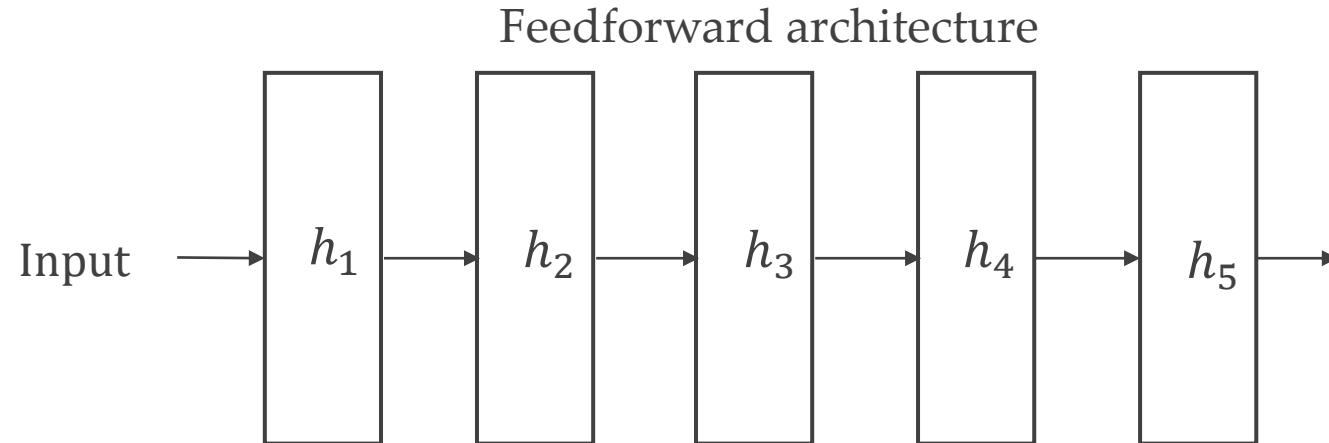
Examples: $f = Ax$, $f = \exp(x)$



Requirements

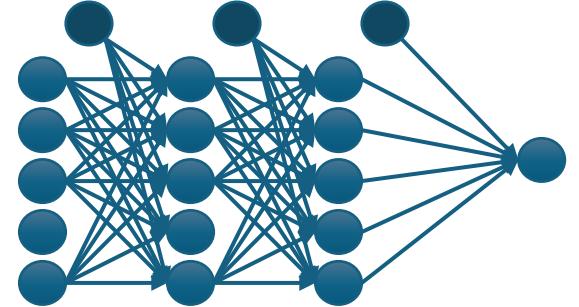
- (1) Activations must be **1st-order differentiable (almost) everywhere.**
- (2) Take special care when there are cycles in the architecture of blocks.

Most models are feedforward networks (e.g., CNNs, Transformers).



Training goal and overview

We have a dataset of inputs and outputs.



Initialize all weights and biases with random values.

Learn weights and biases through “forward-backward” propagation.

- **Forward step:** Map input to predicted output.
- **Loss step:** Compare predicted output to ground truth output.
- **Backward step:** Correct predictions by propagating gradients.

The linear / fully-connected layer

$$x \in \mathbb{R}^{1 \times M}, w \in \mathbb{R}^{N \times M}$$

$$h(x; w) = x \cdot w^T + b$$

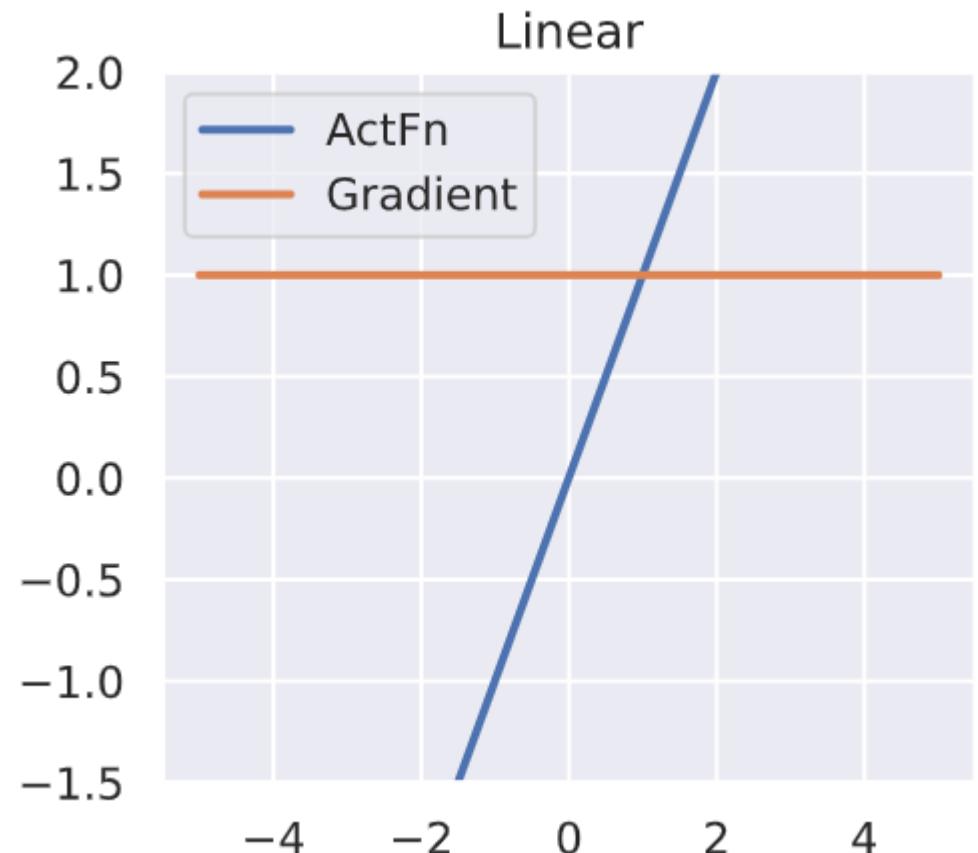
$$\frac{dh}{dx} = w$$

Identity activation function.

No activation saturation.

Hence, strong & stable gradients.

Reliable learning with linear modules.



Forward propagation

When using linear layers, essentially repeated application of perceptrons:

1. Start from the input, multiply with weights, sum, add bias.
2. Repeat for all following layers until you reach the end.

There is one main new element (next to the multiple layers):

Activation functions after each layer.

Why have activation functions?

Each hidden/output neuron is a linear sum.

A combination of linear functions is a linear function!

$$\begin{aligned}v(x) &= ax + b \\w(z) &= cz + d \\w(v(x)) &= c(ax + b) + d = (ac)x + (cb + d)\end{aligned}$$

Activation functions transforms the outputs of each neuron.

This results in non-linear functions.

Activation functions

Defines how the weighted sum of the input is transformed into an output in a layer of the network.

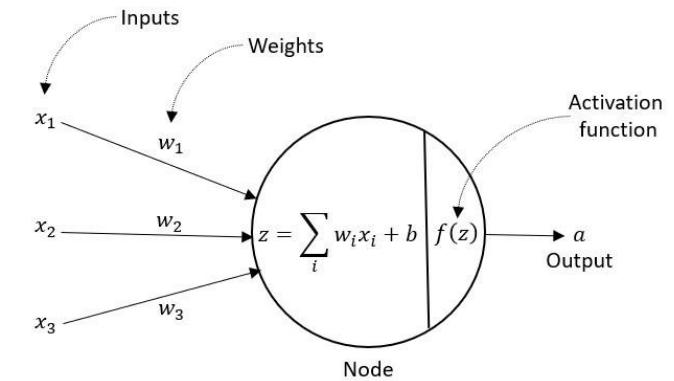
If output range limited, then called a “*squashing function*.”

The choice of activation function has a large impact on the capability and performance of the neural network.

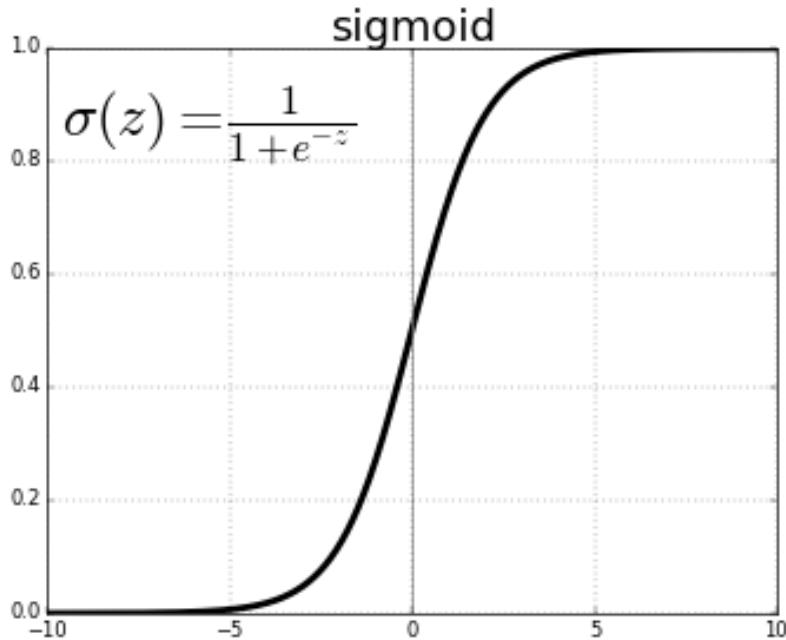
Different activation functions may be combined, but rare.

All hidden layers typically use the same activation function.

Need to be differentiable at most points.



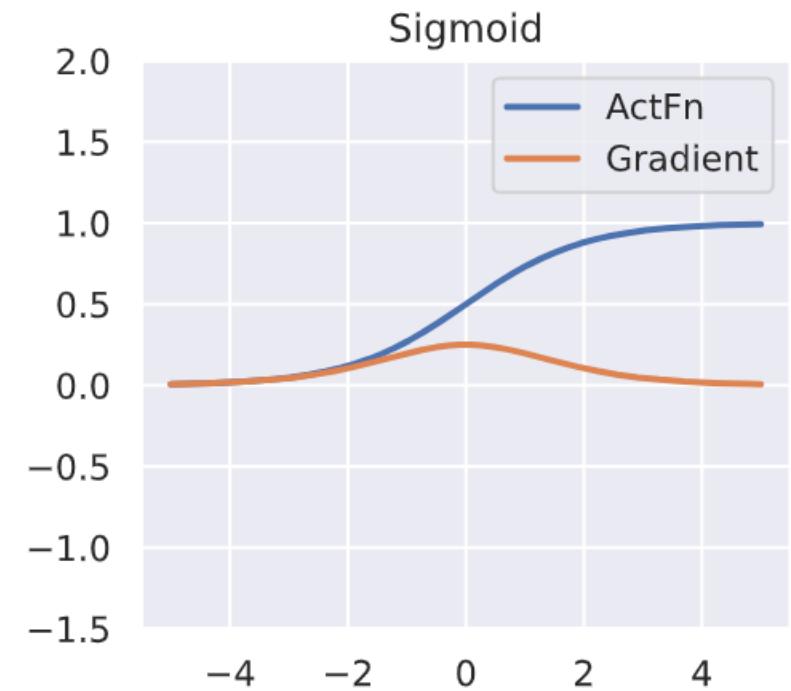
The sigmoid activation



Range: (0,1)

The problem with sigmoid activation is vanishing gradient
this because sigma is between 0 to 1 and when i multiply
them together to compute the derivative it shrinks

Differentiable: $\frac{d}{dz} \sigma(z) = \sigma(z)(1 - \sigma(z))$



The tanh activation

$\tanh(x)$ has better output range $[-1, +1]$.

Data centered around 0 (not 0.5) → stronger gradients

Less “positive” bias for next layers (mean 0, not 0.5)

Both saturate at the extreme → 0 gradients.

Easily become “overconfident” (0 or 1 decisions)

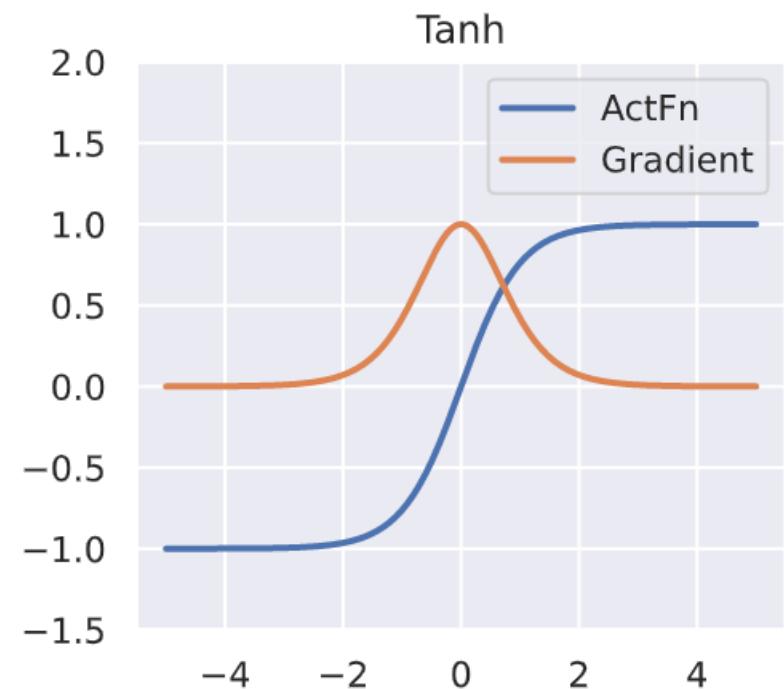
Undesirable for middle layers

Gradients $\ll 1$ with chain multiplication

$\tanh(x)$ better for middle layers.

Sigmoids for outputs to emulate probabilities.

$$h(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$
$$\frac{\partial h}{\partial x} = 1 - \tanh^2(x)$$

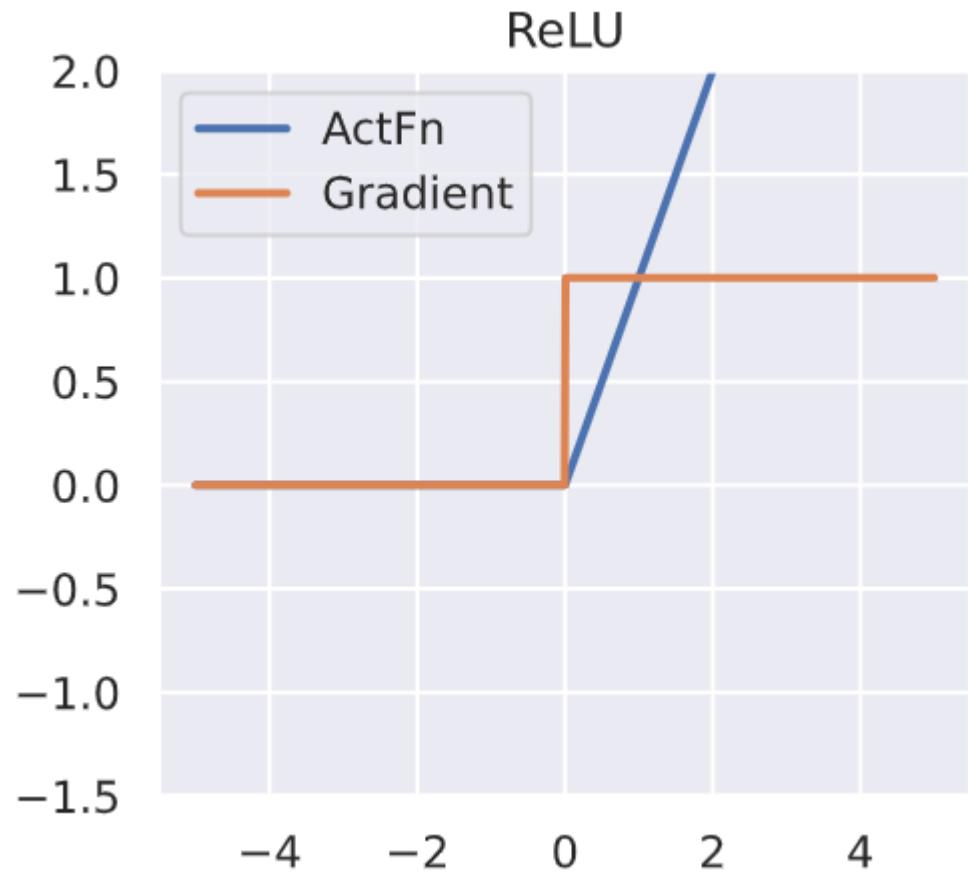


The rectified linear unit (ReLU)

ReLU

$$h(x) = \max(0, x)$$
$$\frac{\partial h}{\partial w} = \begin{cases} 1 & \text{when } x > 0 \\ 0 & \text{when } x \leq 0 \end{cases}$$

Note that it is not differentiable in 0 because from the definition of differentiability, to be differentiable the derivative has to be zero from the right and from the left



Advantages of ReLU

Sparse activation: In randomly initialized network, ~50% active.

Better gradient propagation: Fewer vanishing gradient problems compared to sigmoidal activation functions that saturate in both directions.

Efficient computation: Only comparison, addition and multiplication.

Limitations of ReLU

Non-differentiable at zero; however, it is differentiable anywhere else, and the value of the derivative at zero can be arbitrarily chosen to be 0 or 1.

Not zero-centered.

Unbounded.

Dead neurons problem: neurons can sometimes be pushed into states in which they become inactive for essentially all inputs. Higher learning rates might help.

Also known as Dying Relu problem, Since the gradient of ReLU at zero is also zero, backpropagation cannot update the weights, and the neuron remains stuck in this inactive state forever. This can cause large portions of the network to become non-functional, reducing model capacity and potentially halting learning.



ReLU vs sigmoid

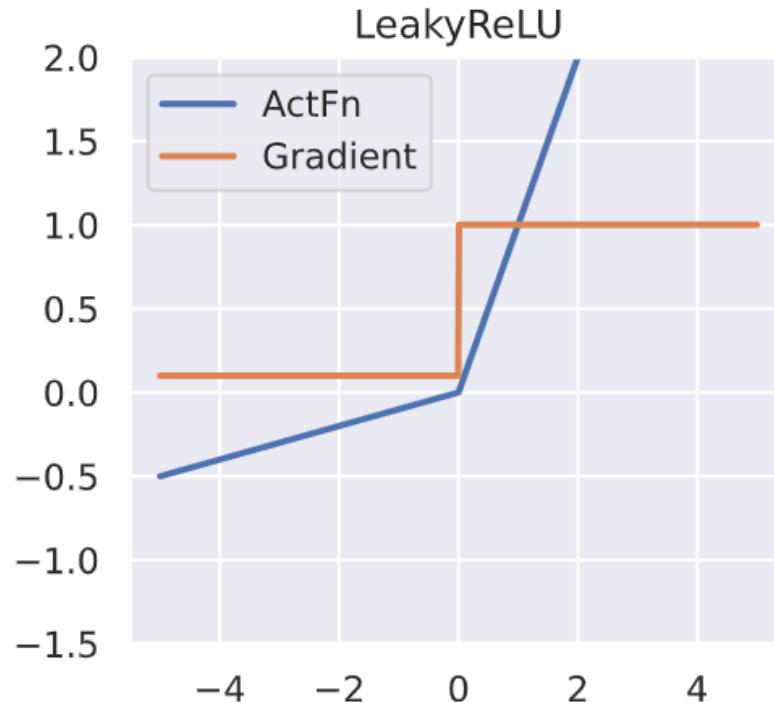
Which one is more non-linear?

In very large range the sigmoid becomes almost a straight line parallel to the x-axis, while Relu maintains its non linearity

Leaky Relu has been proposed to solve the Dying Relu problem

Leaky ReLU

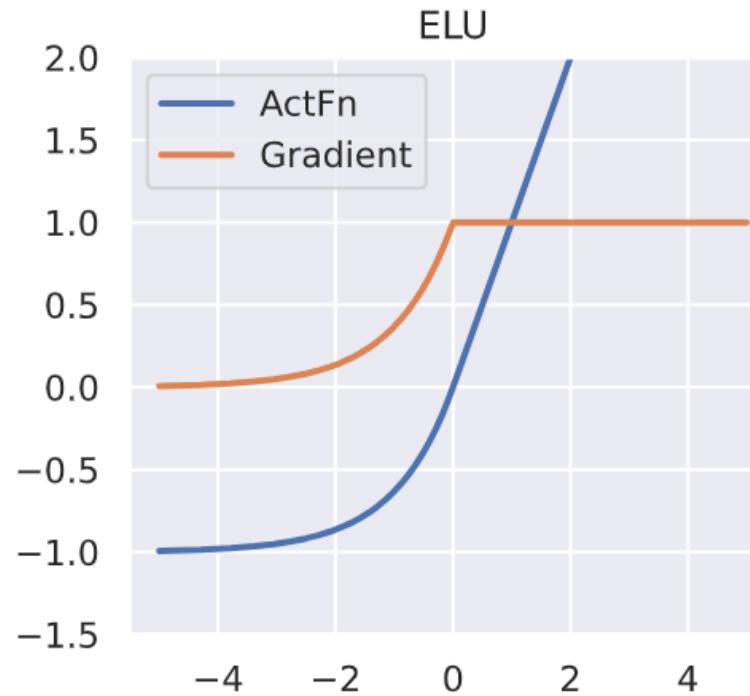
$$h(x) = \begin{cases} x, & \text{when } x > 0 \\ ax, & \text{when } x \leq 0 \end{cases}$$
$$\frac{\partial h}{\partial x} = \begin{cases} 1, & \text{when } x > 0 \\ a, & \text{when } x \leq 0 \end{cases}$$



Leaky ReLUs allow a small, positive gradient when the unit is not active.
Parametric ReLUs, or PReLU, treat a as learnable parameter.

Exponential Linear Unit (ELU)

$$h(x) = \begin{cases} x, & \text{when } x > 0 \\ \exp(x) - 1, & x \leq 0 \end{cases}$$
$$\frac{\partial h}{\partial x} = \begin{cases} 1, & \text{when } x > 0 \\ \exp(x), & x \leq 0 \end{cases}$$



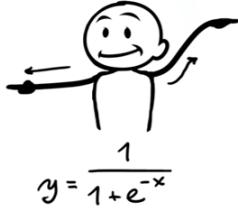
ELU is a smooth approximation to the rectifier.

It has a non-monotonic “bump” when $x < 0$.

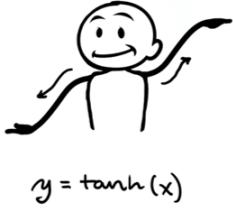
It serves as the default activation for models such as BERT.

The many flavors of activation functions

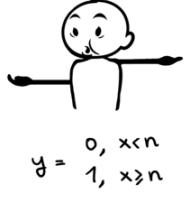
Sigmoid



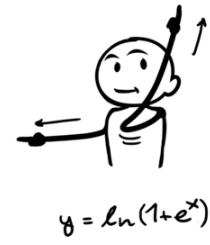
Tanh



Step Function



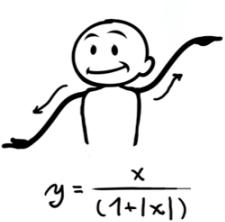
Softplus



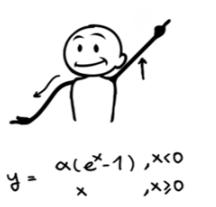
ReLU



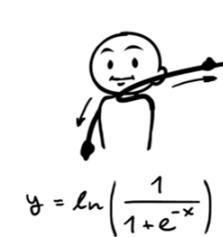
Softsign



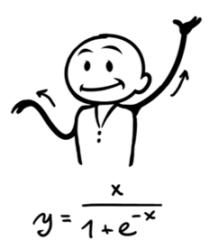
ELU



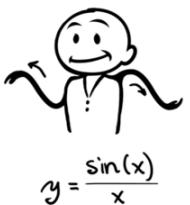
Log of Sigmoid



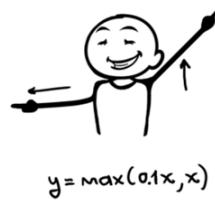
Swish



Sinc



Leaky ReLU



Mish



How to choose an activation function

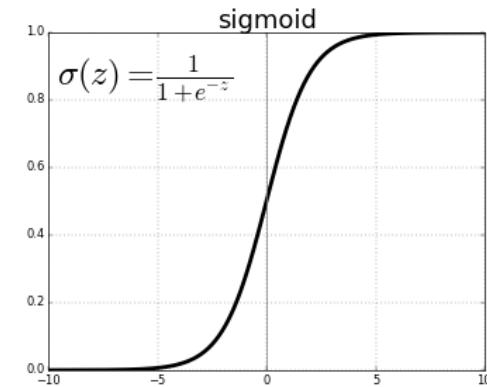
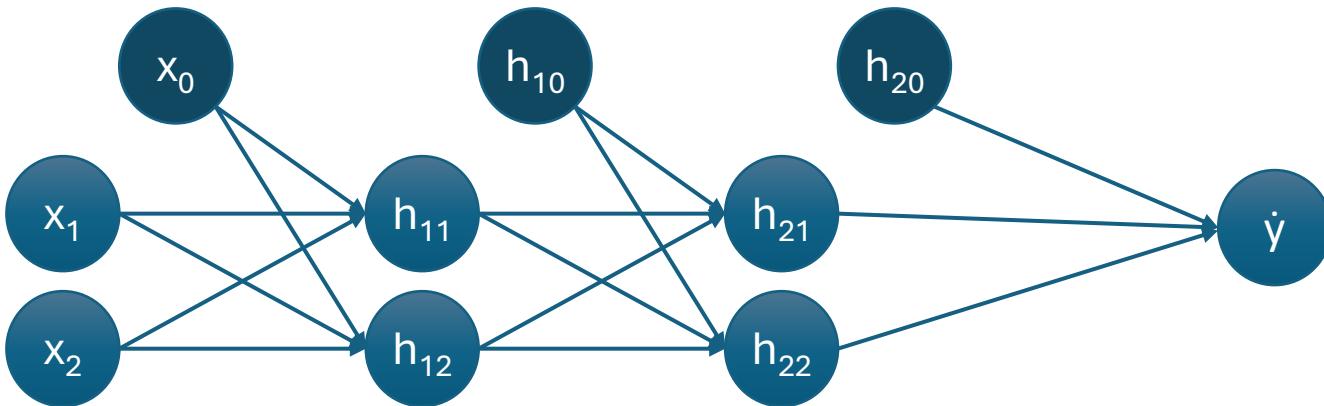
Hidden layers

- In modern neural networks, the default recommendation is to use the rectified linear unit (ReLU)
- (*Recurrent Neural Networks*: Tanh and/or Sigmoid activation function.)

Output layer

- Regression: One node, linear activation.
- Binary Classification: One node, sigmoid activation.
- Multiclass Classification: One node per class, softmax activation.

Cost function: binary classification



Let $y_i \in \{0, 1\}$ denote the binary label of example i and $p_i \in [0, 1]$ denote the output of example i

Our goal: minimize p_i if $y_i=0$, maximize if $y_i = 1$

Maximize: $p_i^{y_i} \cdot (1 - p_i)^{(1-y_i)}$

I.e. minimize: $-(y_i \cdot \log(p_i) + (1 - y_i) \cdot \log(1 - p_i))$ Negative log likelihood aka cross entropy

distillation: train a network in the labels only to get a prob distribution and then use it as target of my nn on the actual samples

Multi-class classification: softmax

there is still a binary view, we consider it only if it is correct,

Outputs probability distribution.

$\sum_{i=1}^K h(x_i) = 1$ for K classes or simply normalizes in a non-linear manner.

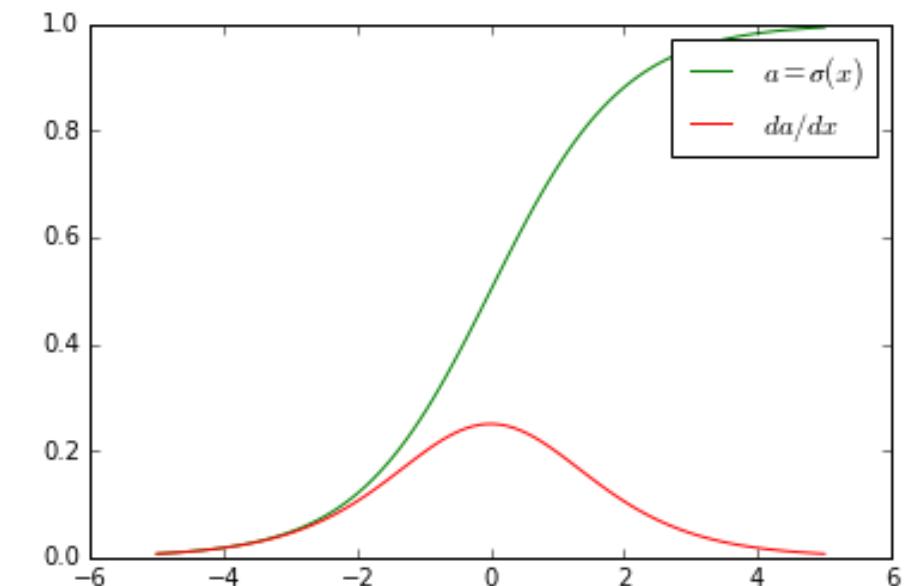
$$h(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Avoid exponentiating too large/small numbers for better stability.

$$h(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} = \frac{e^{x_i - \mu}}{\sum_j e^{x_j - \mu}}, \mu = \max_i x_i$$

the mu is the temperature, it help to normalizes because often there is one term (which is a very high peak) which dominates the distribution

Loss becomes: $-\sum_{j=1}^K y_j \log(p(x)_j)$



What about settings with multiple classes,
where each sample can have multiple
classes as prediction?

Use logistic sigmoid for each output node, where the target vector will be no more one-hot encoded

What ordinal classification?

Ordinal classification (also known as ordinal regression) is a type of supervised learning problem where the target categories have a natural order or ranking, but the differences between categories are not necessarily equal or measurable numerically. This distinguishes it from standard multi-class classification, which treats all classes as independent and unrelated.

With ordinal classification we could treat the problem as a regression one

Architecture design

The overall structure of the network:

how many units should it have

how those units should be connected to each other

Neural networks are organized into groups of units, called layers in a chain structure.

The first layer is given by

$$\mathbf{h}^{(1)} = g^{(1)} \left(\mathbf{W}^{(1)\top} \mathbf{x} + \mathbf{b}^{(1)} \right)$$

And the second layer is

$$\mathbf{h}^{(2)} = g^{(2)} \left(\mathbf{W}^{(2)\top} \mathbf{h}^{(1)} + \mathbf{b}^{(2)} \right)$$

Universal approximation theorem

Universal approximation theorem (recap: ML1)

Feedforward networks with hidden layers provide a universal approximation framework.

A large MLP with even a single hidden layer is able to represent any function provided that the network is given enough hidden units.

However, no guarantee that the training algorithm will be able to learn that function

May not be able to find the parameter values that corresponds to the desired function.

Might choose the wrong function due to overfitting.

How many hidden units?

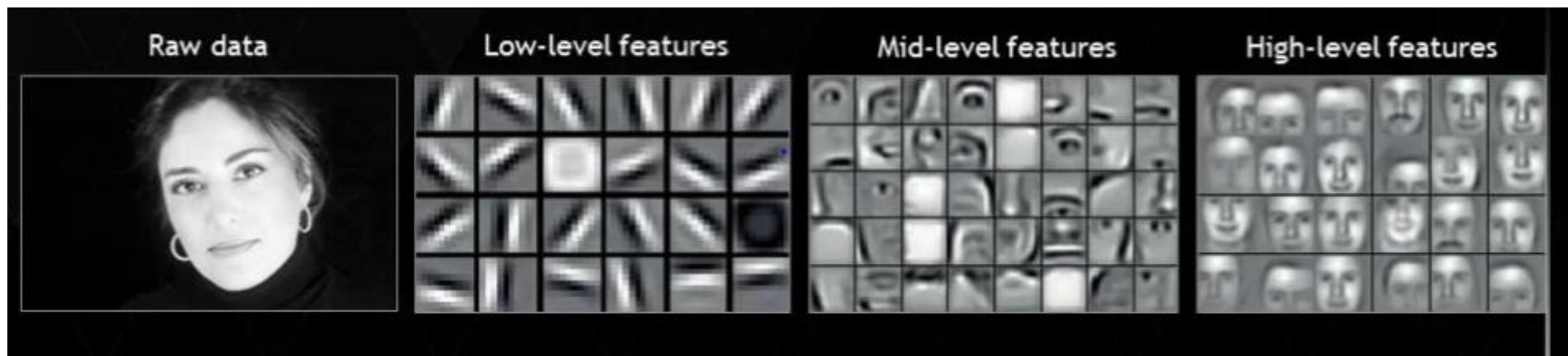
Width and depth

In the worse case, an exponential number of hidden units
a deep rectifier net can require an exponential number of hidden units with a shallow (one hidden layer) network.

We like deep models in deep learning:

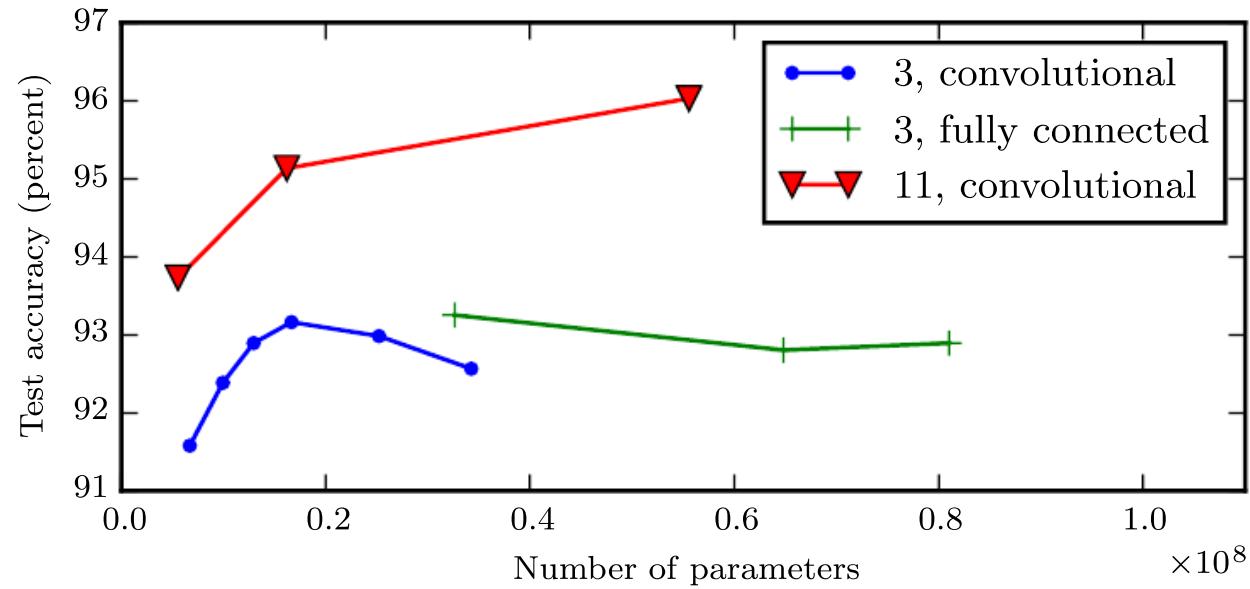
1. can reduce the number of units required to represent the desired function.
2. can reduce the amount of generalization error.
3. deeper networks often generalize better.

Deep networks and pattern hierarchies



An empirical result on width and depth

Increasing the number of parameters in layers of ConvNets without increasing depth is not nearly as effective at increasing test set performance.



How units are connected between layers also matters

FFN: A jungle of architectures

Perceptrons, MLPs

RNNs, LSTMs, GRUs

Vanilla, Variational, Denoising Autoencoders

Hopfield Nets, Restricted Boltzmann Machines

Convolutional Nets, Deconvolutional Nets

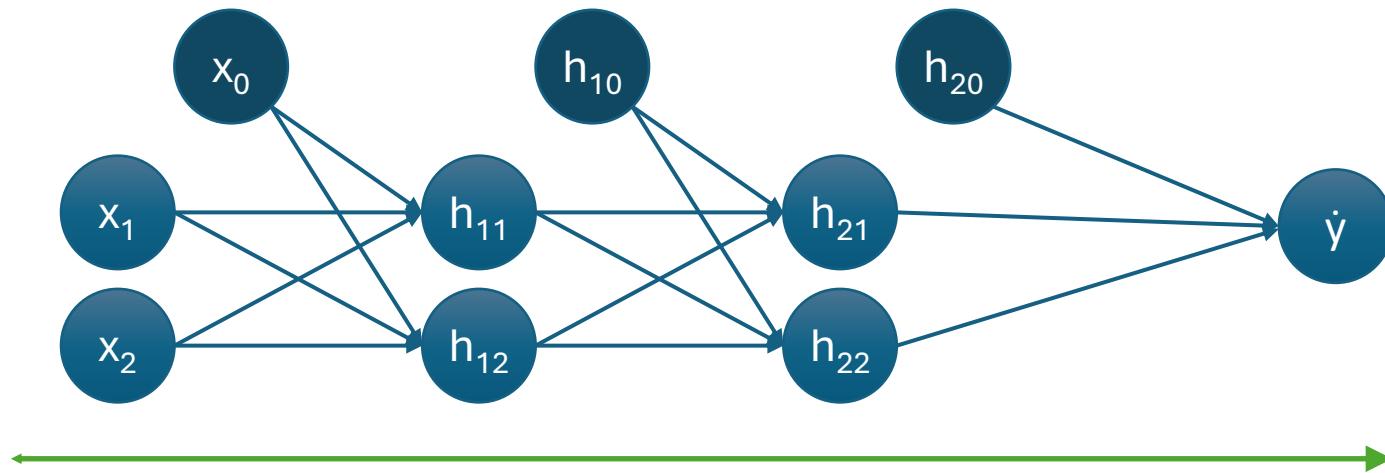
Generative Adversarial Nets

Deep Residual Nets, Neural Turing Machines

Transformers

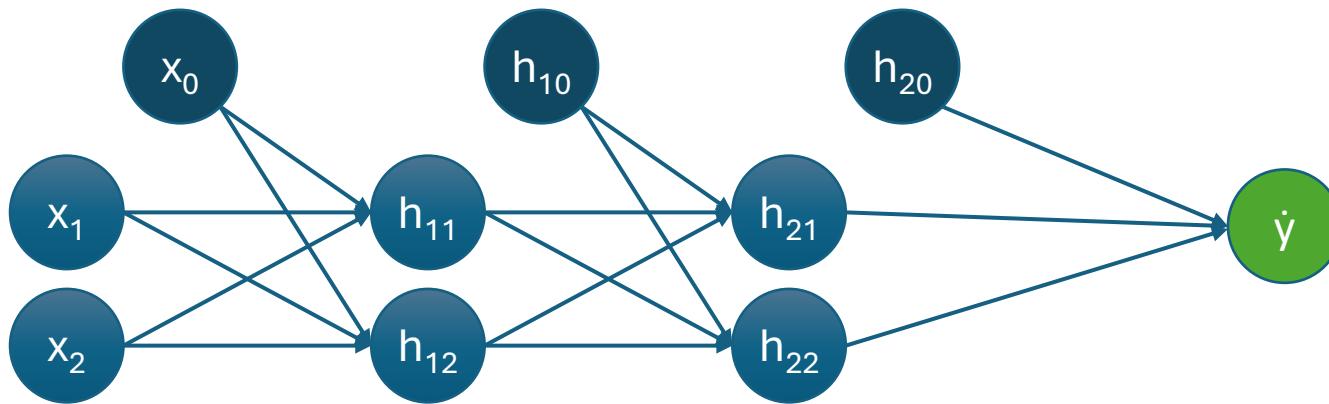
They all rely on modules

Training deep networks: summary



1. Move input through network to yield prediction

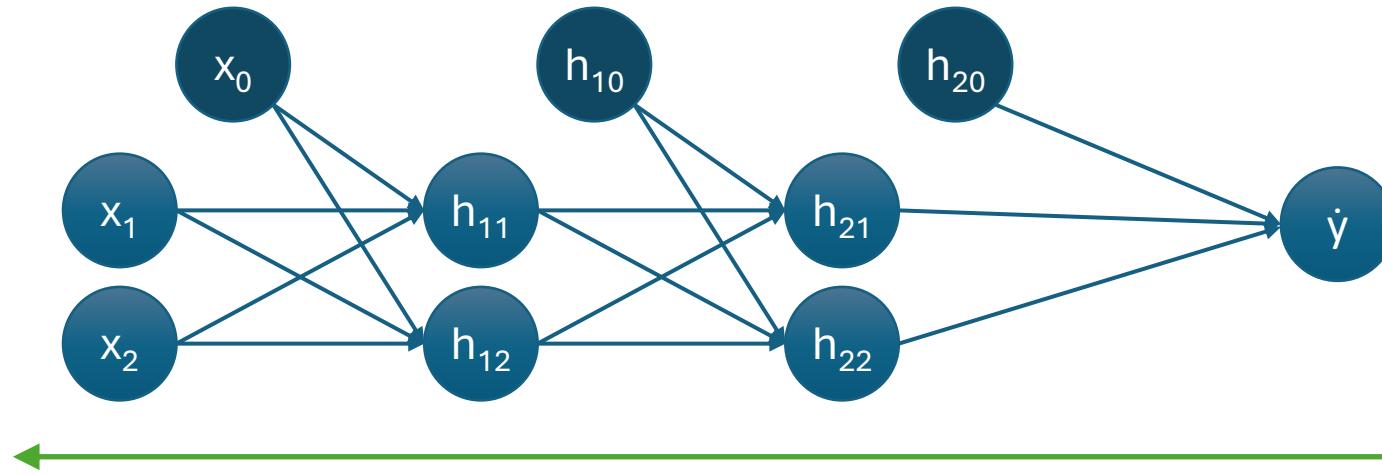
Training deep networks: summary



1. Move input through network to yield prediction
2. Compare prediction to ground truth label

Training deep networks: summary

Repeat multiple times for all training examples



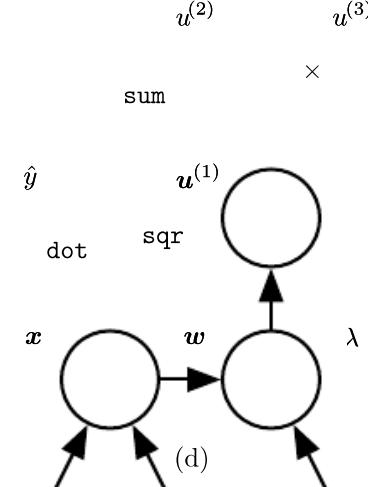
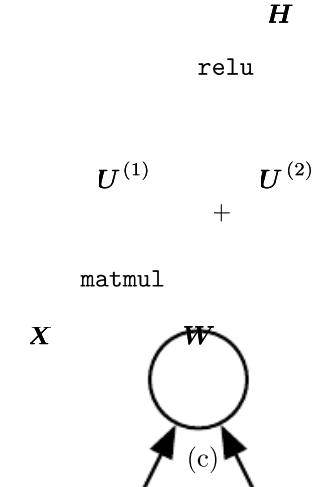
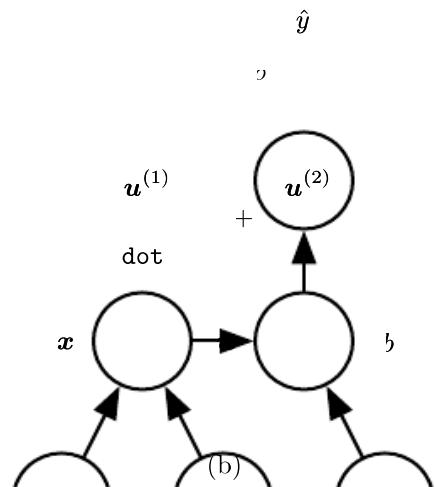
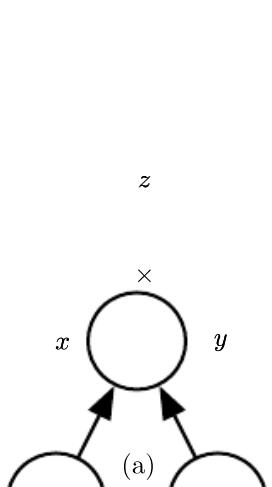
1. Move input through network to yield prediction
2. Compare prediction to ground truth label
3. Backpropagate errors to all weights

Break

Functions as computation graphs

For backprop, it is helpful to view function compositions as graphs.

Each node in the graph indicates a variable, an operation is a simple function of one or more variables.



Backprop: chain rule as an algorithm

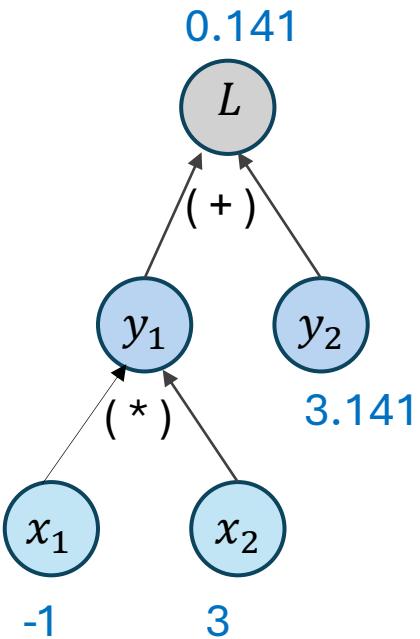
The neural network loss is a composite function of modules.

We want the gradient w.r.t. to the parameters of the l layer.

$$\frac{d\mathcal{L}}{dw_l} = \frac{d\mathcal{L}}{dh_L} \cdot \frac{dh_L}{dh_{L-1}} \cdot \dots \cdot \frac{dh_l}{dw_l} \Rightarrow \frac{d\mathcal{L}}{dw_l} = \underbrace{\frac{d\mathcal{L}}{dh_l}}_{\text{Gradient of loss w.r.t. the module output}} \cdot \underbrace{\frac{dh_l}{dw_l}}_{\text{Gradient of a module w.r.t. its parameters}}$$

Back-propagation is an algorithm that computes the chain rule, with a specific **order of operations that is highly efficient**.

Chain rule with computation graphs



$$\begin{aligned} dL / dx_1 &= ? \\ &= dL/dy_1 * dy_1/dx_1 \\ \text{now } L &= y_1 + y_2 \\ \text{so } dL/dy_1 &= 1 \\ &= 1 * dy_1/dx_1 \\ \text{now } y_1 &= x_1 * x_2 \\ \text{so } dy_1/dx_1 &= x_2 \\ &= 1 * x_2 \end{aligned}$$

Visual example of compositionality

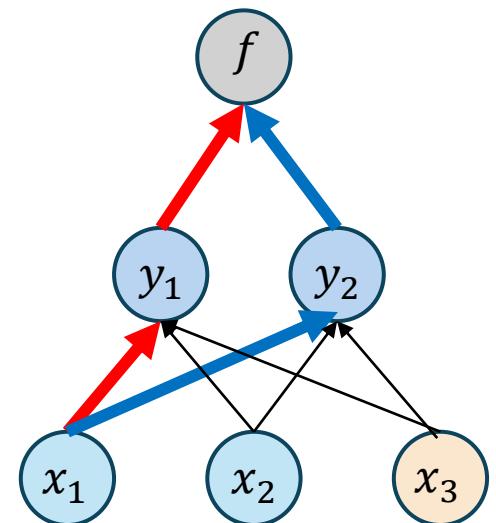
For $h = f \circ y(x)$, here f , and y 's denote functions

$$\frac{dh}{dx} = \frac{df}{dy} \frac{dy}{dx} = \begin{bmatrix} \frac{\partial f}{\partial y_1} & \frac{\partial f}{\partial y_2} \end{bmatrix} \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \frac{\partial y_1}{\partial x_3} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \frac{\partial y_2}{\partial x_3} \end{bmatrix}$$

Focusing on one of the partial derivatives: $\frac{dh}{dx_1}$

$$\frac{dh}{dx_1} = \frac{df}{dy_1} \frac{dy_1}{dx_1} + \frac{df}{dy_2} \frac{dy_2}{dx_1}$$

The partial derivative depends on all paths from f to x_i .



Why you should know about backprop

Backprop is the foundation of modern deep learning.

However, we want to have backward propagation be automated.

If we know how deep learning is broken down, we can handle any new algorithm advance, i.e., we will remain relevant.

Backprop derivation and implementation also a popular interview question!

Four ways to differentiate

Manual

Numerical

Symbolic

Automatic

Numerical differentiation

Idea: we don't care about the actual derivative, we just need to know the slope, so let's compute it directly.

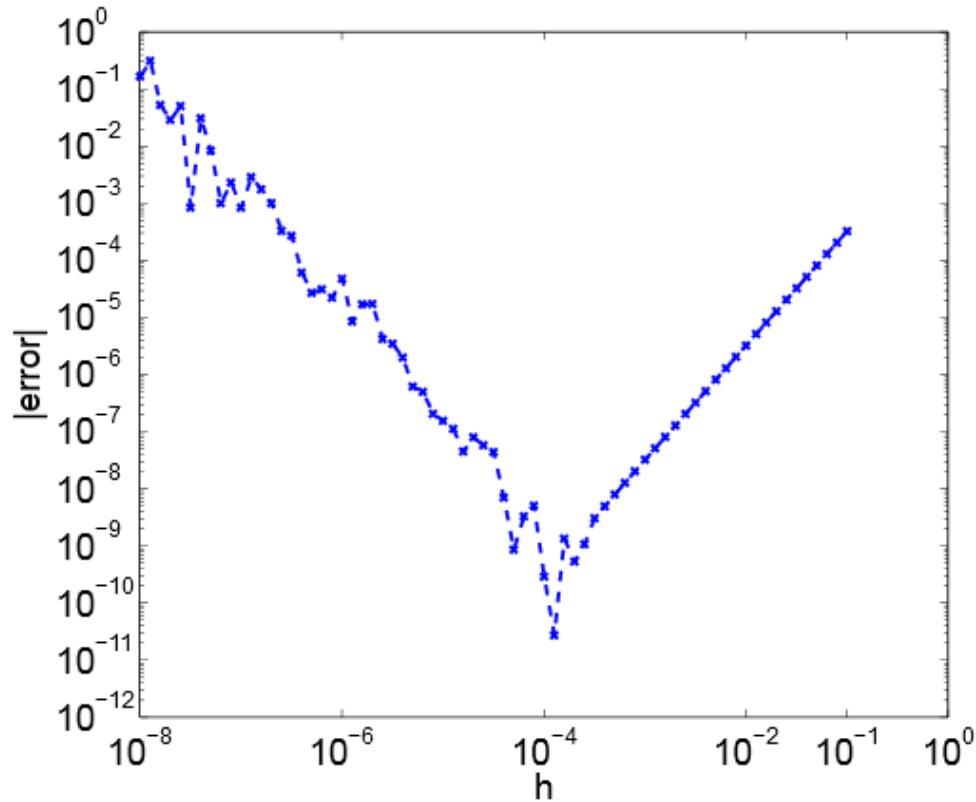
$$\frac{df}{dx} = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$

Pro: works even when function itself is unknown.

Con: unstable, sensitive to approximation errors, computationally heavy.

Truncation and round-off errors

Round-off error: h too small



Trunction error: h too big

Symbolic Differentiation

Let the computer figure out the closed-form expression through calculus.

Pro: no more approximation errors.

I'm doing too much expanding all the terms

Con: expression swell.

$$f(x) = \frac{e^{wx+b} + e^{-(wx+b)}}{e^{wx+b} - e^{-(wx+b)}}$$
$$\frac{\partial f}{\partial w} = \frac{(-xe^{-b-wx} - xe^{b+wx})(e^{-b-wx} + e^{b+wx})}{(-e^{-b-wx} + e^{b+wx})^2} + \frac{-xe^{-b-wx} + xe^{b+wx}}{-e^{-b-wx} + e^{b+wx}}$$

Automatic differentiation

Main idea:

Express functions into its variables and elementary operations.

Pre-define the derivatives of the elementary operations.

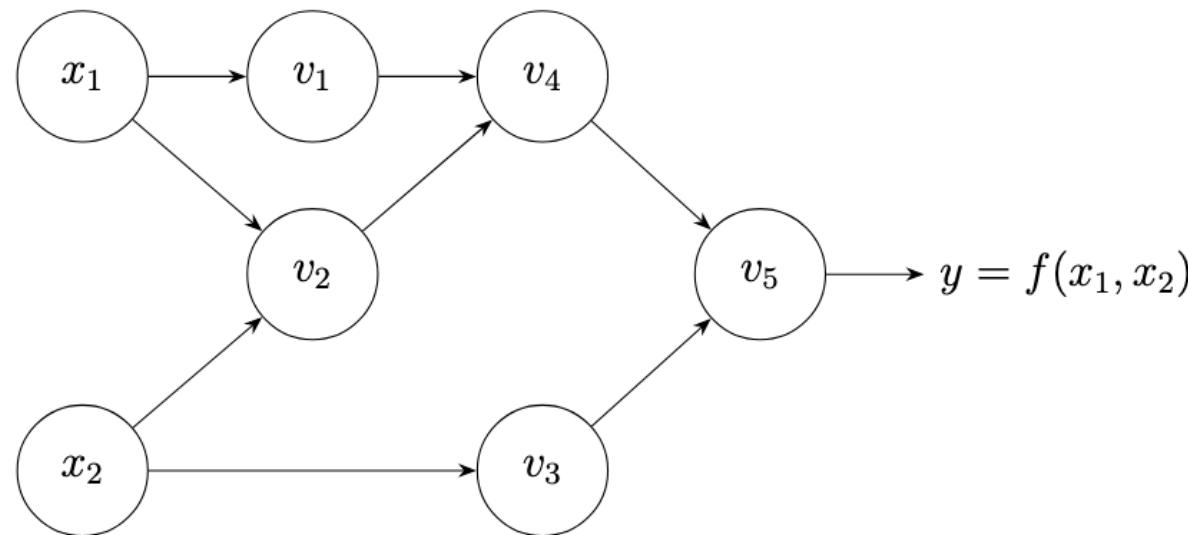
Chain the elementary derivatives without repeating operations over and over.

Forward and backward propagation by computers.

Forward mode AutoDiff by example

Consider the following function and computation graph:

$$y = f(x_1, x_2) = \log x_1 + x_1 x_2 - \sin x_2$$

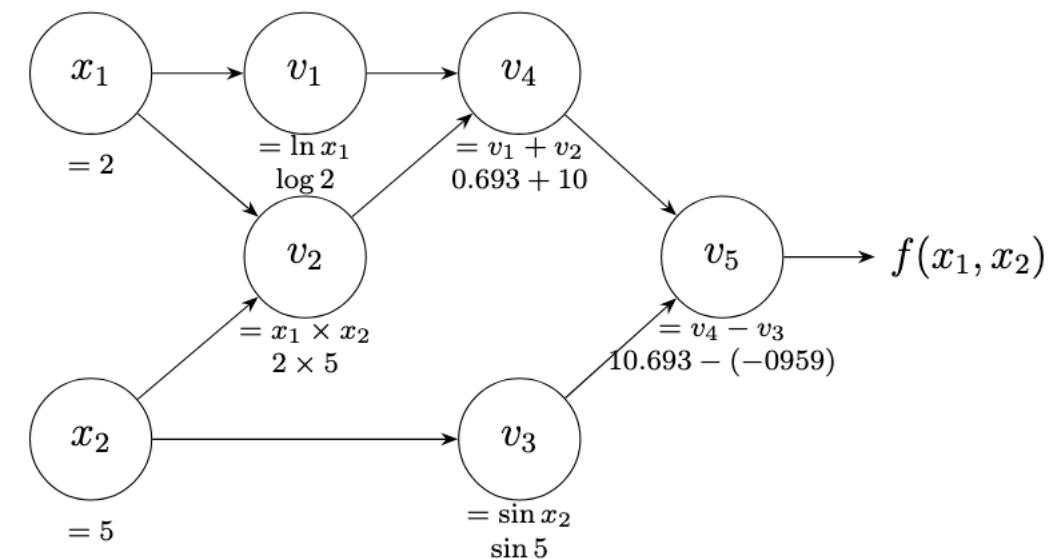


Forward mode AutoDiff by example

We first do the forward pass of this “network” for input (2,5).

$$y = f(x_1, x_2) = \log x_1 + x_1 x_2 - \sin x_2$$

x_1	$= 2$
x_2	$= 5$
<hr/>	
$v_1 = \log x_1$	$= \log 2$
$v_2 = x_1 \times x_2$	$= 2 \times 5$
$v_3 = \sin x_2$	$= \sin 5$
$v_4 = v_1 + v_2$	$= 0.693 + 10$
$v_5 = v_4 - v_3$	$= 10.693 + 0.959$
<hr/>	
$y = v_5$	$= 11.652$



Forward mode AutoDiff by example

Forward mode AutoDiff computes the derivative of the output w.r.t. one input.

Let's compute the partial derivative $\partial y / \partial x_1$

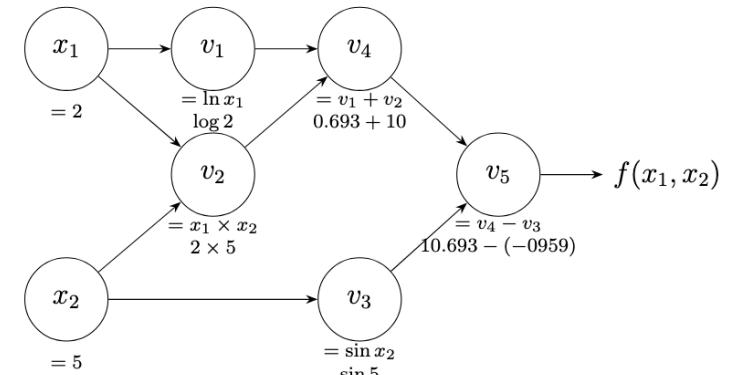
Throughout forward mode AutoDiff, we use the following notation: $\dot{v} = \frac{\partial v}{\partial x_1}$

Forward mode AutoDiff by example

Forward trace of our function:

$$y = f(x_1, x_2) = \log x_1 + x_1 x_2 - \sin x_2$$

$$\begin{array}{rcl} \dot{x}_1 &= \partial x_1 / \partial x_1 &= 1 \\ \dot{x}_2 &= \partial x_2 / \partial x_1 &= 0 \\ \hline \dot{v}_1 &= \dot{x}_1 / x_1 &= 1/2 \\ \dot{v}_2 &= \dot{x}_1 \times x_2 + \dot{x}_2 \times x_1 &= 1 \times 5 + 0 \times 2 \\ \dot{v}_3 &= \dot{x}_2 \times \cos x_2 &= 0 \times \cos 5 \\ \dot{v}_4 &= \dot{v}_1 + \dot{v}_2 &= 0.5 + 5 \\ \dot{v}_5 &= \dot{v}_4 - \dot{v}_3 &= 5.5 - 0 \\ \hline \dot{y} &= \dot{v}_5 &= 5.5 \end{array}$$



$$\dot{v}_1 = \frac{\partial v_1}{\partial x_1} \times \frac{\partial x_1}{\partial x_1} = \frac{1}{x_1} \times \frac{\partial x_1}{\partial x_1} = \frac{\dot{x}_1}{x_1} = \frac{1}{2}$$

Reverse mode AutoDiff by example

Reverse mode AutoDiff computes the derivative of the output w.r.t. all inputs.

New notation (like we normally use): $\bar{v} = \frac{\partial y}{\partial v}$

Let's do the reverse mode to calculate: $\frac{\partial y}{\partial x_2}$

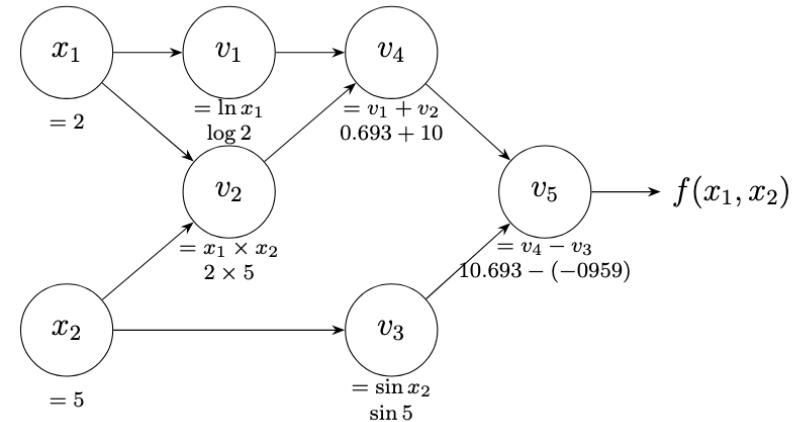
Reverse mode AutoDiff by example

$$y = f(x_1, x_2) = \log x_1 + x_1 x_2 - \sin x_2$$

If I store the already computed derivatives when going backward in the network thanks to the chain rule I will have to compute only one new derivative per node

x_1	$= 2$
x_2	$= 5$
<hr/>	<hr/>
$v_1 = \log x_1$	$= \log 2$
$v_2 = x_1 \times x_2$	$= 2 \times 5$
$v_3 = \sin x_2$	$= \sin 5$
$v_4 = v_1 + v_2$	$= 0.693 + 10$
$v_5 = v_4 - v_3$	$= 10.693 + 0.959$
<hr/>	<hr/>
y	$= v_5$
	$= 11.652$

\bar{x}_1	$= 5.5$
\bar{x}_2	$= 1.716$
<hr/>	<hr/>
$\bar{x}_1 = \bar{x}_1 + \bar{v}_1 \frac{\partial v_1}{\partial x_1}$	$= \bar{x}_1 + \bar{v}_1/x_1 = 5.5$
$\bar{x}_2 = \bar{x}_2 + \bar{v}_2 \frac{\partial v_2}{\partial x_2}$	$= \bar{x}_2 + \bar{v}_2 \times x_1 = 1.716$
$\bar{x}_1 = \bar{v}_2 \frac{\partial v_2}{\partial x_1}$	$= \bar{v}_2 \times x_2 = 5$
$\bar{x}_2 = \bar{v}_3 \frac{\partial v_3}{\partial x_2}$	$= \bar{v}_3 \times \cos x_2 = -0.284$
$\bar{v}_2 = \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	$= \bar{v}_4 \times 1 = 1$
$\bar{v}_1 = \bar{v}_4 \frac{\partial v_4}{\partial v_1}$	$= \bar{v}_4 \times 1 = 1$
$\bar{v}_3 = \bar{v}_5 \frac{\partial v_5}{\partial v_3}$	$= \bar{v}_5 \times (-1) = -1$
$\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	$= \bar{v}_5 \times 1 = 1$
<hr/>	<hr/>
$\bar{v}_5 = \bar{y}$	$= 1$



$$\begin{aligned} \bar{v}_4 &= \frac{\partial y}{\partial v_4} = \frac{\partial y}{\partial v_5} \frac{\partial v_5}{\partial v_4} \\ &= \bar{v}_5 \frac{\partial v_5}{\partial v_4} = \bar{v}_5 \times 1 \\ \bar{v}_5 &= \frac{\partial y}{\partial v_5} = \frac{\partial y}{\partial y} = 1 \end{aligned}$$

$$\bar{v}_5 = \frac{\partial y}{\partial v_5} = \frac{\partial y}{\partial y} = 1$$

AutoDiff summarized

For general function $h: \mathbb{R}^m \rightarrow \mathbb{R}^n$

m forward mode differentiations and n reverse mode differentiations.

One of the two is known as backprop, which one?

Reverse mode.

Why is reverse mode differentiation, the norm in deep learning?

We do deep learning on high-dimensional data.

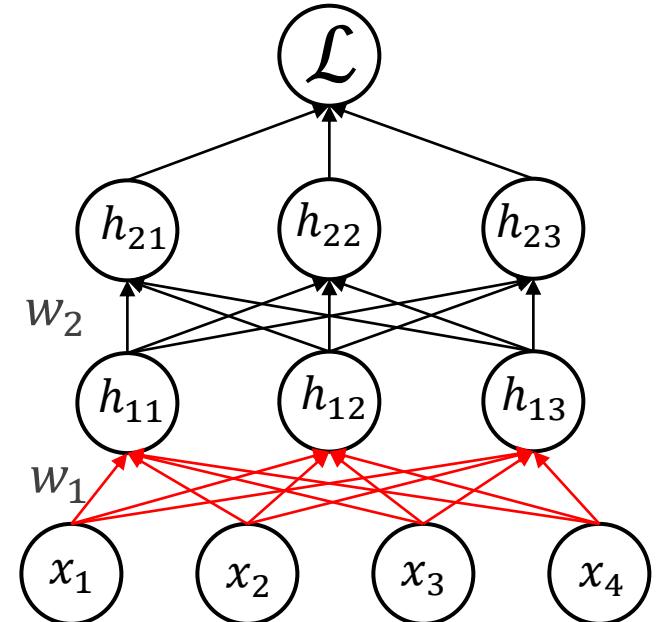
Autodiff visually

Forward propagation

$$\begin{aligned}
 \xrightarrow{\textcolor{red}{\longrightarrow}} h_0 &= x \\
 \xrightarrow{\textcolor{red}{\longrightarrow}} h_1 &= \sigma(w_1 h_0) && \rightarrow \text{Store } \textcolor{blue}{h_1}. \text{ Remember that } \partial_x \sigma = \sigma \cdot (1 - \sigma) \\
 h_2 &= \sigma(w_2 h_1) && \rightarrow \text{Store } \textcolor{red}{h_2} \\
 \mathcal{L} &= 0.5 \cdot \|l - h_2\|^2
 \end{aligned}$$

Backward propagation

$$\begin{aligned}
 \frac{d\mathcal{L}}{dh_2} &= -(y^* - \textcolor{red}{h}_2) \\
 \frac{d\mathcal{L}}{dw_2} &= \frac{d\mathcal{L}}{dh_2} \frac{dh_2}{dw_2} = \frac{d\mathcal{L}}{dh_2} \textcolor{blue}{h}_1 \sigma(w_2 h_1) (1 - \sigma(w_2 h_1)) = \frac{d\mathcal{L}}{dh_2} \textcolor{blue}{h}_1 \textcolor{red}{h}_2 (1 - \textcolor{red}{h}_2) \\
 \frac{d\mathcal{L}}{dh_1} &= \frac{d\mathcal{L}}{dh_2} \frac{dh_1}{dh_2} = \frac{d\mathcal{L}}{dh_2} w_2 \sigma(w_2 h_1) (1 - \sigma(w_2 h_1)) = \frac{d\mathcal{L}}{dh_2} w_2 \textcolor{red}{h}_2 (1 - \textcolor{red}{h}_2) \\
 \frac{d\mathcal{L}}{dw_1} &= \frac{d\mathcal{L}}{dh_1} \frac{dh_1}{dw_1} = \frac{d\mathcal{L}}{dh_1} h_0 \sigma(w_1 h_0) (1 - \sigma(w_1 h_0)) = \frac{d\mathcal{L}}{dh_1} h_0 \textcolor{blue}{h}_1 (1 - \textcolor{blue}{h}_1)
 \end{aligned}$$



Autodiff visually

Forward propagation

$$h_0 = x$$

$$h_1 = \sigma(w_1 h_0)$$

$$h_2 = \sigma(w_2 h_1)$$

$$\mathcal{L} = 0.5 \cdot \|l - h_2\|^2$$

→ Store h_1 . Remember that $\partial_x \sigma = \sigma \cdot (1 - \sigma)$
 → Store h_2

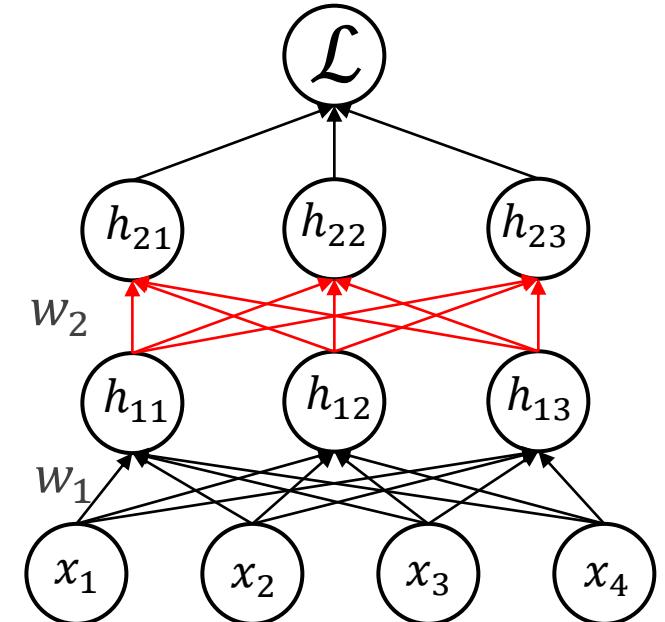
Backward propagation

$$\frac{d\mathcal{L}}{dh_2} = -(y^* - h_2)$$

$$\frac{d\mathcal{L}}{dw_2} = \frac{d\mathcal{L}}{dh_2} \frac{dh_2}{dw_2} = \frac{d\mathcal{L}}{dh_2} h_1 \sigma(w_2 h_1) (1 - \sigma(w_2 h_1)) = \frac{d\mathcal{L}}{dh_2} h_1 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dh_1} = \frac{d\mathcal{L}}{dh_2} \frac{dh_1}{dh_2} = \frac{d\mathcal{L}}{dh_2} w_2 \sigma(w_2 h_1) (1 - \sigma(w_2 h_1)) = \frac{d\mathcal{L}}{dh_2} w_2 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dw_1} = \frac{d\mathcal{L}}{dh_1} \frac{dh_1}{dw_1} = \frac{d\mathcal{L}}{dh_1} h_0 \sigma(w_1 h_0) (1 - \sigma(w_1 h_0)) = \frac{d\mathcal{L}}{dh_1} h_0 h_1 (1 - h_1)$$



Autodiff visually

Forward propagation

$$h_0 = x$$

$$h_1 = \sigma(w_1 h_0)$$

$$h_2 = \sigma(w_2 h_1)$$

$$\mathcal{L} = 0.5 \cdot \|l - h_2\|^2$$

→ Store h_1 . Remember that $\partial_x \sigma = \sigma \cdot (1 - \sigma)$
 → Store h_2

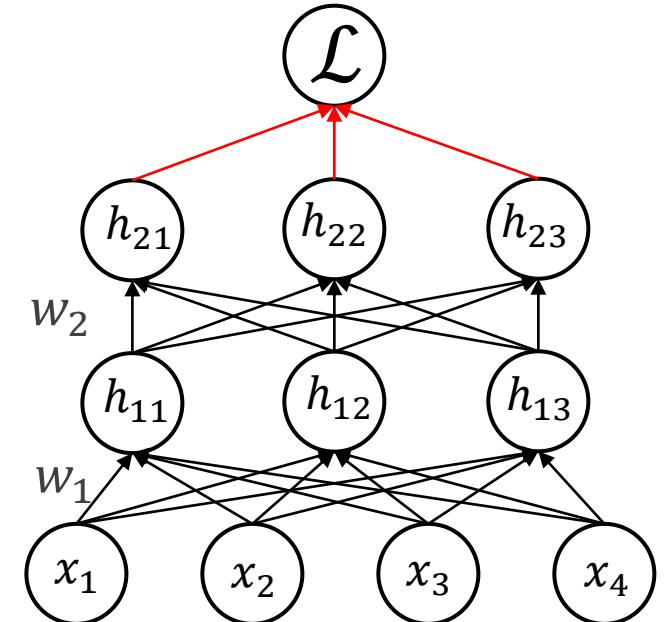
Backward propagation

$$\frac{d\mathcal{L}}{dh_2} = -(y^* - h_2)$$

$$\frac{d\mathcal{L}}{dw_2} = \frac{d\mathcal{L}}{dh_2} \frac{dh_2}{dw_2} = \frac{d\mathcal{L}}{dh_2} h_1 \sigma(w_2 h_1) (1 - \sigma(w_2 h_1)) = \frac{d\mathcal{L}}{dh_2} h_1 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dh_1} = \frac{d\mathcal{L}}{dh_2} \frac{dh_1}{dh_2} = \frac{d\mathcal{L}}{dh_2} w_2 \sigma(w_2 h_1) (1 - \sigma(w_2 h_1)) = \frac{d\mathcal{L}}{dh_2} w_2 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dw_1} = \frac{d\mathcal{L}}{dh_1} \frac{dh_1}{dw_1} = \frac{d\mathcal{L}}{dh_1} h_0 \sigma(w_1 h_0) (1 - \sigma(w_1 h_0)) = \frac{d\mathcal{L}}{dh_1} h_0 h_1 (1 - h_1)$$



Autodiff visually

Forward propagation

$$h_0 = x$$

$$h_1 = \sigma(w_1 h_0)$$

$$h_2 = \sigma(w_2 h_1)$$

$$\mathcal{L} = 0.5 \cdot \|l - h_2\|^2$$

→ Store h_1 . Remember that $\partial_x \sigma = \sigma \cdot (1 - \sigma)$
 → Store h_2

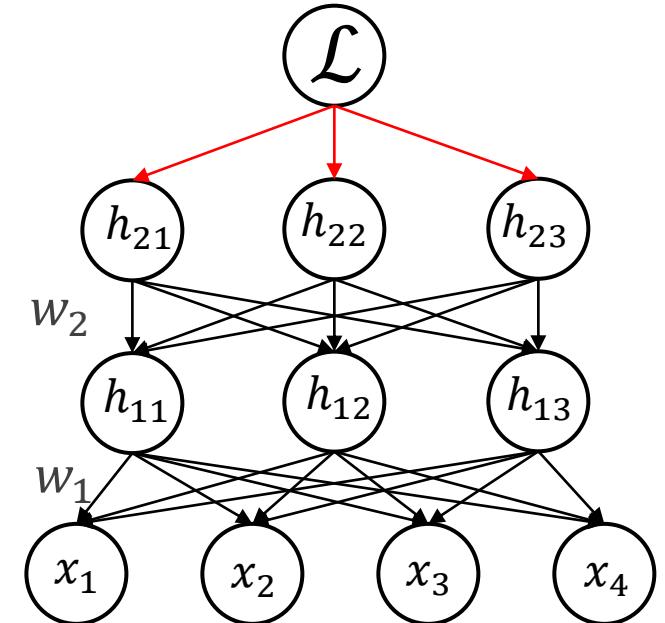
Backward propagation

$$\frac{d\mathcal{L}}{dh_2} = -(y^* - h_2)$$

$$\frac{d\mathcal{L}}{dw_2} = \frac{d\mathcal{L}}{dh_2} \frac{dh_2}{dw_2} = \frac{d\mathcal{L}}{dh_2} h_1 \sigma(w_2 h_1) (1 - \sigma(w_2 h_1)) = \frac{d\mathcal{L}}{dh_2} h_1 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dh_1} = \frac{d\mathcal{L}}{dh_2} \frac{dh_1}{dh_2} = \frac{d\mathcal{L}}{dh_2} w_2 \sigma(w_2 h_1) (1 - \sigma(w_2 h_1)) = \frac{d\mathcal{L}}{dh_2} w_2 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dw_1} = \frac{d\mathcal{L}}{dh_1} \frac{dh_1}{dw_1} = \frac{d\mathcal{L}}{dh_1} h_0 \sigma(w_1 h_0) (1 - \sigma(w_1 h_0)) = \frac{d\mathcal{L}}{dh_1} h_0 h_1 (1 - h_1)$$



Autodiff visually

Forward propagation

$$h_0 = x$$

$$h_1 = \sigma(w_1 h_0)$$

$$h_2 = \sigma(w_2 h_1)$$

$$\mathcal{L} = 0.5 \cdot \|l - h_2\|^2$$

→ Store h_1 . Remember that $\partial_x \sigma = \sigma \cdot (1 - \sigma)$
 → Store h_2

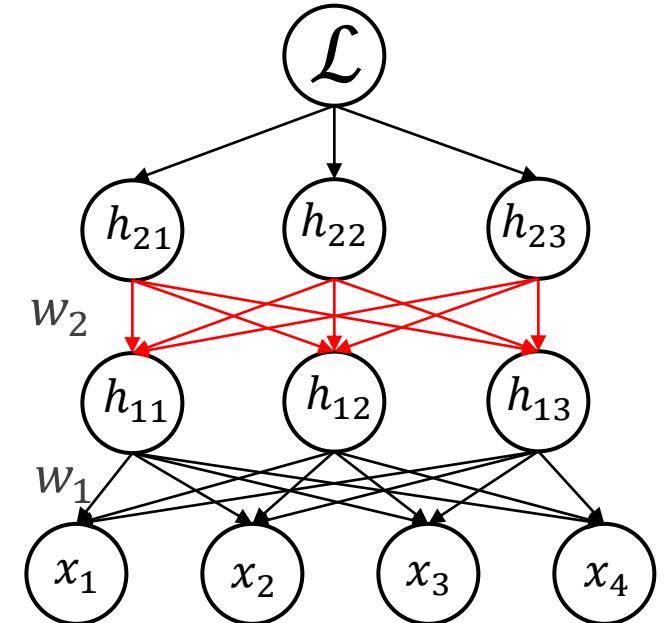
Backward propagation

$$\frac{d\mathcal{L}}{dh_2} = -(y^* - h_2)$$

$$\frac{d\mathcal{L}}{dw_2} = \frac{d\mathcal{L}}{dh_2} \frac{dh_2}{dw_2} = \frac{d\mathcal{L}}{dh_2} h_1 \sigma(w_2 h_1) (1 - \sigma(w_2 h_1)) = \frac{d\mathcal{L}}{dh_2} h_1 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dh_1} = \frac{d\mathcal{L}}{dh_2} \frac{dh_1}{dh_2} = \frac{d\mathcal{L}}{dh_2} w_2 \sigma(w_2 h_1) (1 - \sigma(w_2 h_1)) = \frac{d\mathcal{L}}{dh_2} w_2 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dw_1} = \frac{d\mathcal{L}}{dh_1} \frac{dh_1}{dw_1} = \frac{d\mathcal{L}}{dh_1} h_0 \sigma(w_1 h_0) (1 - \sigma(w_1 h_0)) = \frac{d\mathcal{L}}{dh_1} h_0 h_1 (1 - h_1)$$



Autodiff visually

Forward propagation

$$h_0 = x$$

$$h_1 = \sigma(w_1 h_0)$$

$$h_2 = \sigma(w_2 h_1)$$

$$\mathcal{L} = 0.5 \cdot \|l - h_2\|^2$$

→ Store h_1 . Remember that $\partial_x \sigma = \sigma \cdot (1 - \sigma)$
 → Store h_2

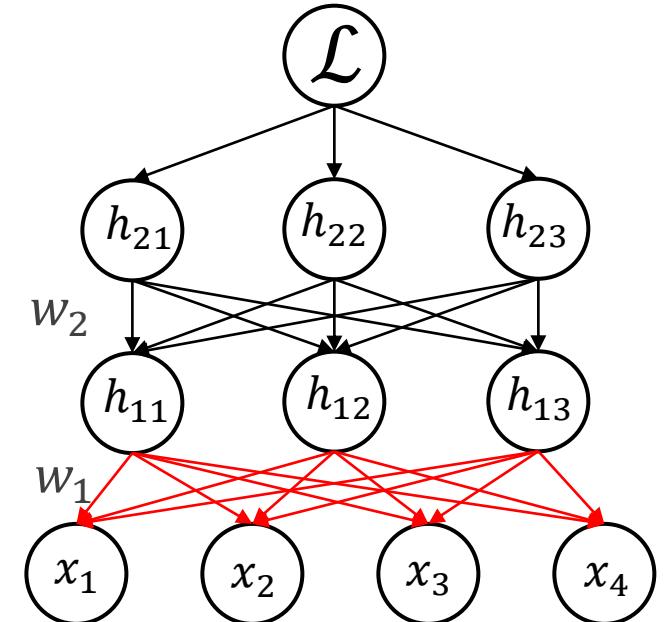
Backward propagation

$$\frac{d\mathcal{L}}{dh_2} = -(y^* - h_2)$$

$$\frac{d\mathcal{L}}{dw_2} = \frac{d\mathcal{L}}{dh_2} \frac{dh_2}{dw_2} = \frac{d\mathcal{L}}{dh_2} h_1 \sigma(w_2 h_1) (1 - \sigma(w_2 h_1)) = \frac{d\mathcal{L}}{dh_2} h_1 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dh_1} = \frac{d\mathcal{L}}{dh_2} \frac{dh_1}{dh_2} = \frac{d\mathcal{L}}{dh_2} w_2 \sigma(w_2 h_1) (1 - \sigma(w_2 h_1)) = \frac{d\mathcal{L}}{dh_2} w_2 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dw_1} = \frac{d\mathcal{L}}{dh_1} \frac{dh_1}{dw_1} = \frac{d\mathcal{L}}{dh_1} h_0 \sigma(w_1 h_0) (1 - \sigma(w_1 h_0)) = \frac{d\mathcal{L}}{dh_1} h_0 h_1 (1 - h_1)$$



Our lives without AutoDiff

```
# -*- coding: utf-8 -*-
import numpy as np
import math

# Create random input and output data
x = np.linspace(-math.pi, math.pi, 2000)
y = np.sin(x)

# Randomly initialize weights
a = np.random.randn()
b = np.random.randn()
c = np.random.randn()
d = np.random.randn()

learning_rate = 1e-6
for t in range(2000):
    # Forward pass: compute predicted y
    # y = a + b x + c x^2 + d x^3
    y_pred = a + b * x + c * x ** 2 + d * x ** 3

    # Compute and print loss
    loss = np.square(y_pred - y).sum()
    if t % 100 == 99:
        print(t, loss)

    # Backprop to compute gradients of a, b, c, d with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_a = grad_y_pred.sum()
    grad_b = (grad_y_pred * x).sum()
    grad_c = (grad_y_pred * x ** 2).sum()
    grad_d = (grad_y_pred * x ** 3).sum()

    # Update weights
    a -= learning_rate * grad_a
    b -= learning_rate * grad_b
    c -= learning_rate * grad_c
    d -= learning_rate * grad_d

print(f'Result: y = {a} + {b} x + {c} x^2 + {d} x^3')
```

Our lives with AutoDiff (AutoGrad)

```
class DynamicNet(torch.nn.Module):
    def __init__(self):
        """
        In the constructor we instantiate five parameters and assign them as members.
        """
        super().__init__()
        self.a = torch.nn.Parameter(torch.randn(()))
        self.b = torch.nn.Parameter(torch.randn(()))
        self.c = torch.nn.Parameter(torch.randn(()))
        self.d = torch.nn.Parameter(torch.randn(()))
        self.e = torch.nn.Parameter(torch.randn(()))

    def forward(self, x):
        """
        For the forward pass of the model, we randomly choose either 4, 5
        and reuse the e parameter to compute the contribution of these orders.

        Since each forward pass builds a dynamic computation graph, we can use normal
        Python control-flow operators like loops or conditional statements when
        defining the forward pass of the model.

        Here we also see that it is perfectly safe to reuse the same parameter many
        times when defining a computational graph.
        """
        y = self.a + self.b * x + self.c * x ** 2 + self.d * x ** 3
        for exp in range(4, random.randint(4, 6)):
            y = y + self.e * x ** exp
        return y
```

```
# Create Tensors to hold input and outputs.
x = torch.linspace(-math.pi, math.pi, 2000)
y = torch.sin(x)

# Construct our model by instantiating the class defined above
model = DynamicNet()

# Construct our loss function and an Optimizer. Training this strange model with
# vanilla stochastic gradient descent is tough, so we use momentum
criterion = torch.nn.MSELoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=1e-8, momentum=0.9)
for t in range(30000):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x)

    # Compute and print loss
    loss = criterion(y_pred, y)
    if t % 2000 == 1999:
        print(t, loss.item())

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    print(f'Result: {model.string()}')
```

Summary

Forward and backward propagation.

Forward and reverse automatic differentiation.

Implementation of automatic differentiation.

Learning and reflection

Understanding Deep Learning: Chapter 4

Understanding Deep Learning: Chapter 5

Understanding Deep Learning: Chapter 6.1, 7.1-7.4

Next lecture

Lecture	Title	Lecture	Title
1	Intro and history of deep learning	2	AutoDiff
3	Deep learning optimization I	4	Deep learning optimization II
5	Convolutional deep learning	6	Attention-based deep learning
7	Graph deep learning	8	From supervised to unsupervised deep learning
9	Multi-modal deep learning	10	Generative deep learning
11	What doesn't work in deep learning	12	Non-Euclidean deep learning
13	Q&A	14	Deep learning for videos



Deep Learning 1

2025-2026 – Pascal Mettes

Lecture 3

Deep learning optimization I

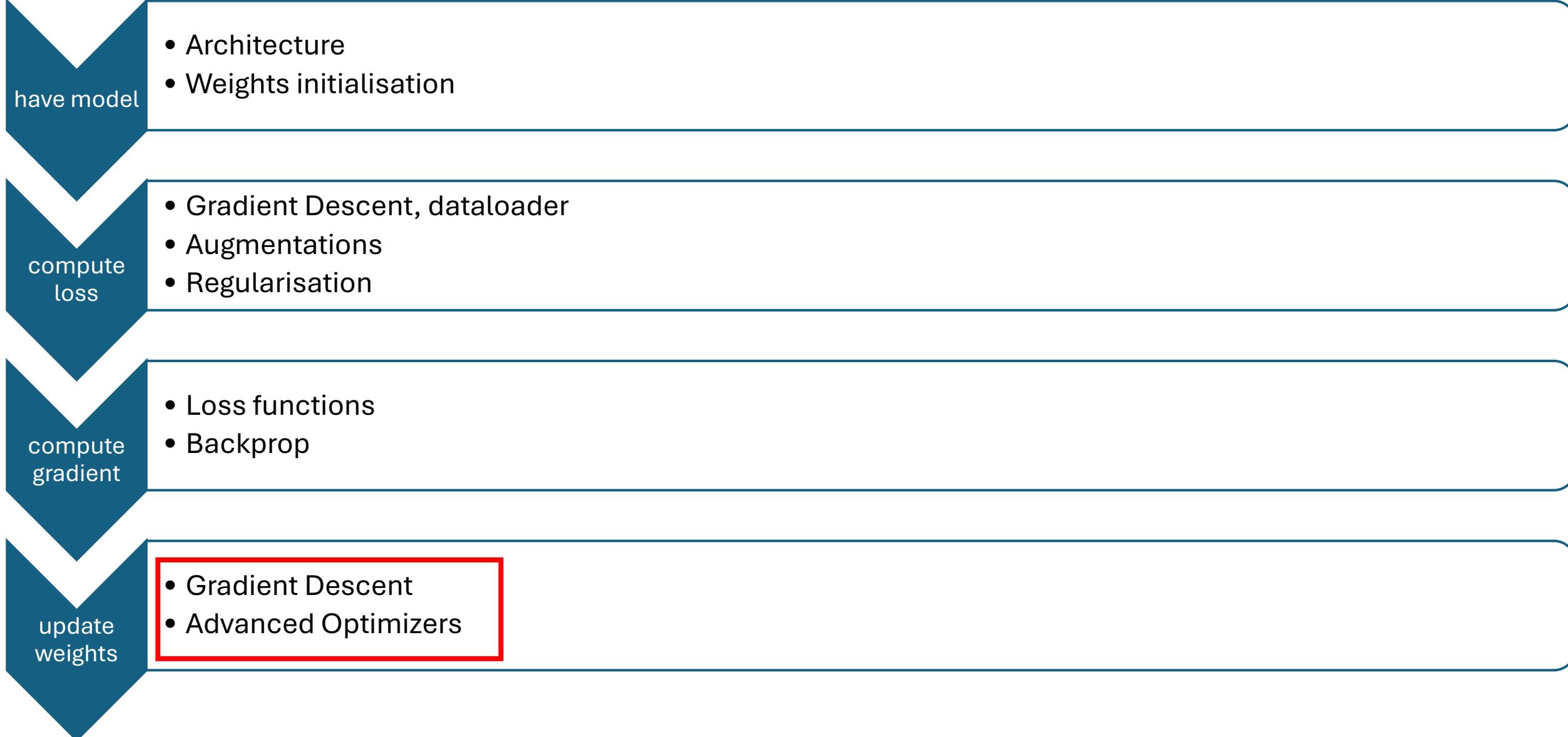
Previous lecture

Lecture	Title	Lecture	Title
1	Intro and history of deep learning	2	AutoDiff
3	Deep learning optimization I	4	Deep learning optimization II
5	Convolutional deep learning	6	Attention-based deep learning
7	Graph deep learning	8	From supervised to unsupervised deep learning
9	Multi-modal deep learning	10	Generative deep learning
11	What doesn't work in deep learning	12	Non-Euclidean deep learning
13	Q&A	14	Deep learning for videos

This lecture

Stochastic gradient descent

Advanced optimizers



Optimization versus learning

Optimization

- given a parametric definition of model and a set of data, we want to discover the optimal parameter that minimize a certain objective function, given some data.
- E.g., find the optimal flight schedule given resources and population.

Learning

- We have observed and unobserved data.
- Reduce errors on the observed data (training data) to *generalize* to unseen data (test data).
- The goal is to reduce the generalization error.

Minimizing risk

We want to optimize on observed data.

Minimizing a cost function, with extra regularizations

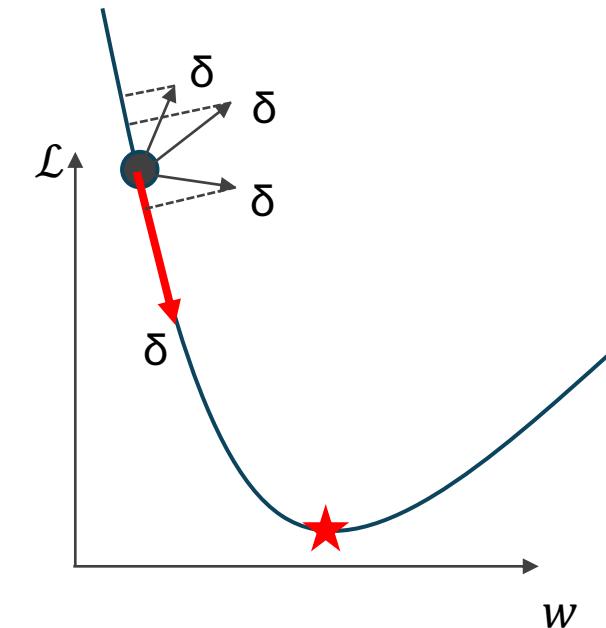
$$\min_{\mathbf{w}} \mathbb{E}_{x,y \sim p_{data}} [\mathcal{L}(f(\mathbf{x}, \mathbf{w}), y)] + \lambda \Omega(\mathbf{w})$$

where $\hat{y} = f(\mathbf{x}, \mathbf{w}) = h_L \circ h_{L-1} \circ \dots \circ h_1(\mathbf{x})$ is the prediction and each h_l comes with parameters \mathbf{w}_l .

In simple words: (1) predictions are not too wrong, while (2): not being “too geared” towards the observed data.

Problem: the true distribution p_{data} is not available.

Empirical risk minimization



In practice having p_{data} is not possible.

We only have a training set of data

$$\min_w \mathbb{E}_{x,y \sim \hat{p}_{data}} [\mathcal{L}(f(\mathbf{x}, \mathbf{w}), y)] + \lambda \Omega(\mathbf{w})$$

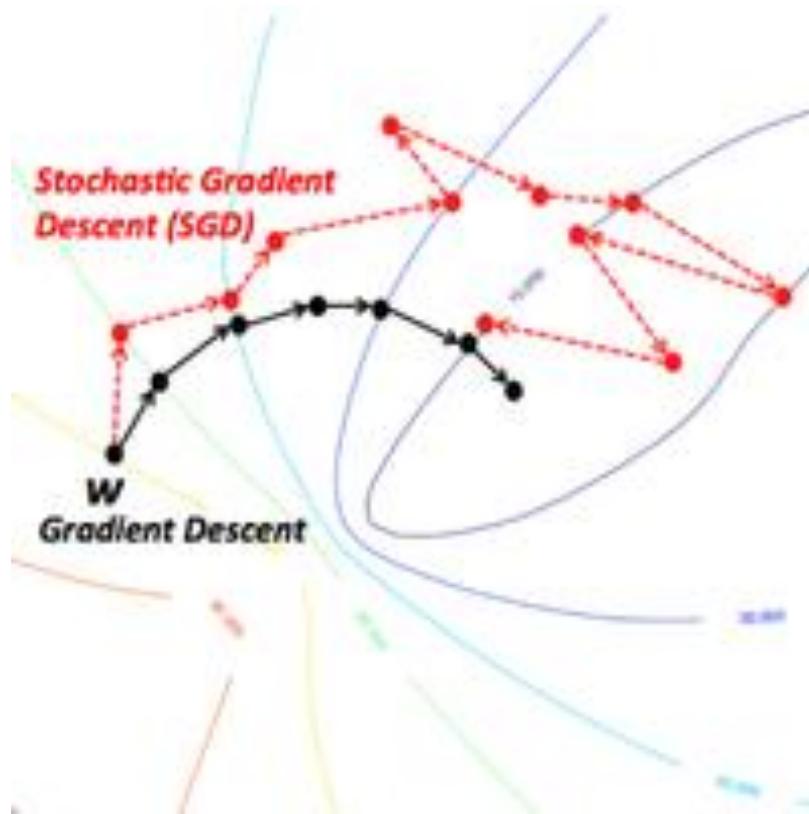
with \hat{p}_{data} the empirical data distribution, defined by a set of training examples.

To minimize any function, we take a step δ . Our best bet: the (negative) gradient

$$-\sum \frac{d}{dw} \mathcal{L}(f(\mathbf{x}, \mathbf{w}), y)$$

Gradient descent based on optimization.

Stochastic gradient descent



The problem is that the training set is too big to be used on GPU during the training phase

Instead of using the entire dataset to calculate gradients, perform parameter update for each mini-batch.

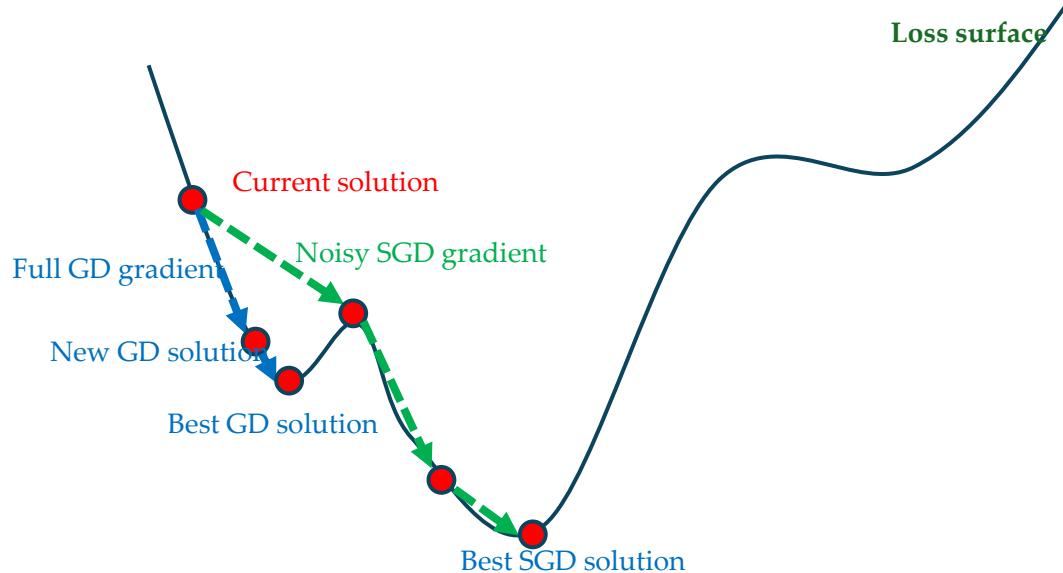
Properties of SGD

Randomness actually reduces overfitting.

Reshuffling is important!

One epoch = go through all mini-batches.

Be careful to balance class/data per batch.



```
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,  
download=True, transform=transform)  
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,  
shuffle=True, num_workers=2)
```

On batch size

Large batch size: more accurate estimation of the gradient.

Very small batch size: underutilizes hardware, but can act as regularizer.

General rule: batch size and learning rate are coupled (double BS = double LR)

(*Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour.* Goyal et al. 2017: Batchsize of 8K)

Guideline: use the largest batch size that fits on the GPU and is a power of 2.

Why does mini-batch SGD work?

Gradient descent is already an approximation; the true data distribution is unknown.

Reduced sample size does not imply reduced gradient quality.

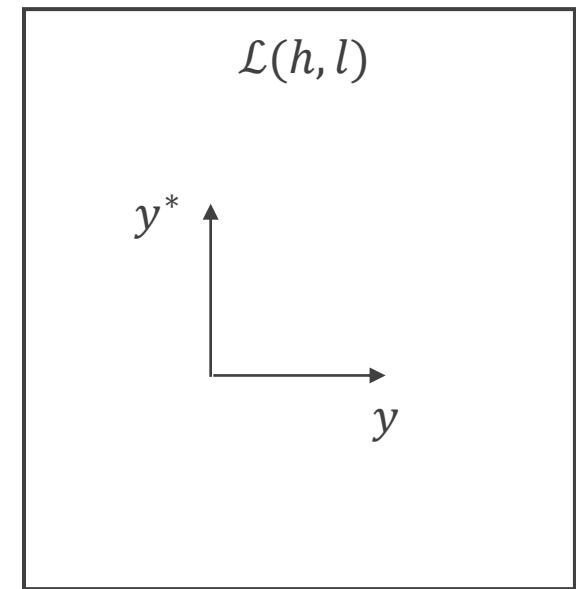
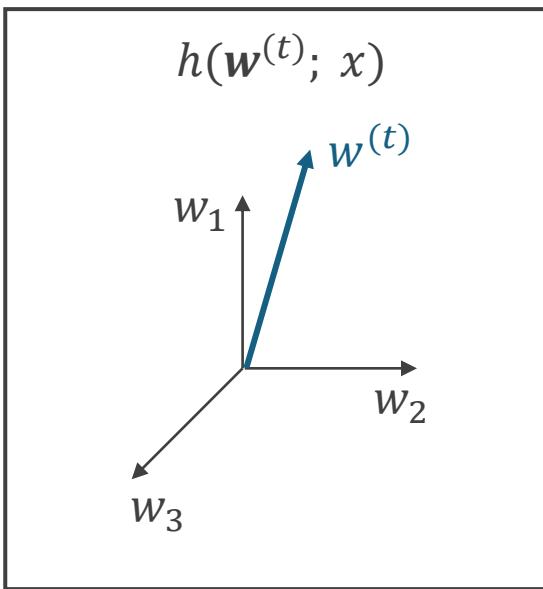
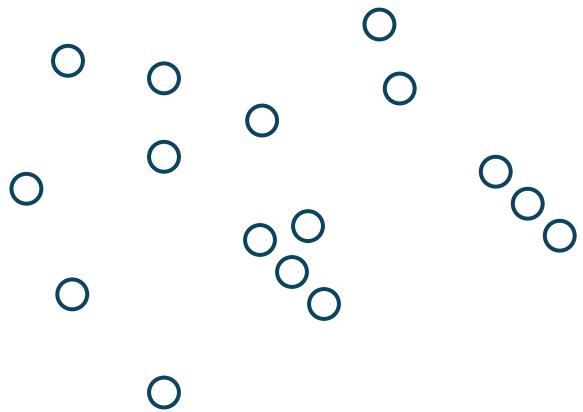
The training samples may have lots of noises or outliers or biases.

- A randomly sampled minibatch may reflect the true data generating distribution better (or worse).

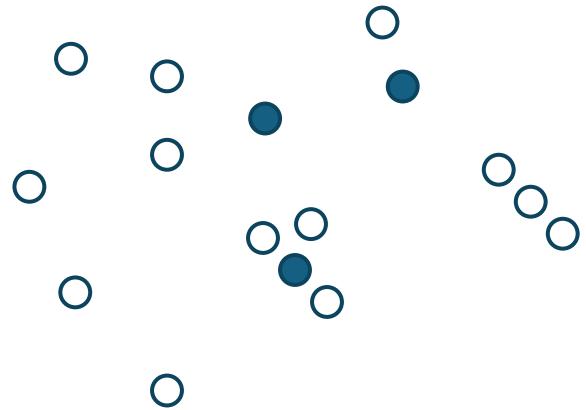
Real gradient might get stuck into a local minima.

- While *more random gradient* computed with minibatch might not.

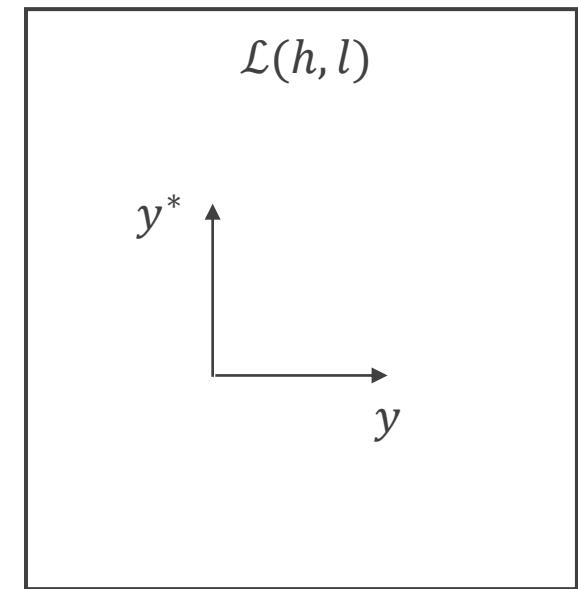
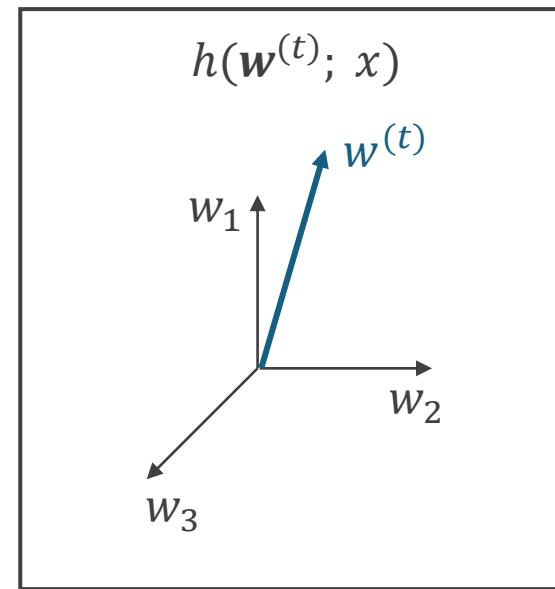
Stochastic gradient descent



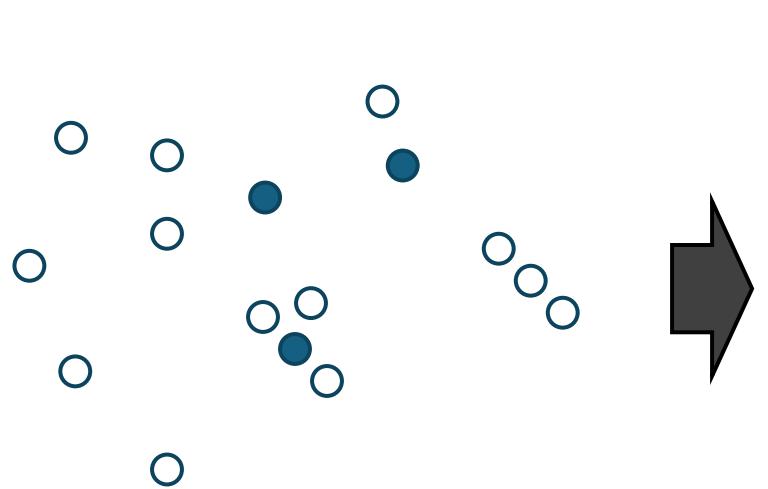
Stochastic gradient descent



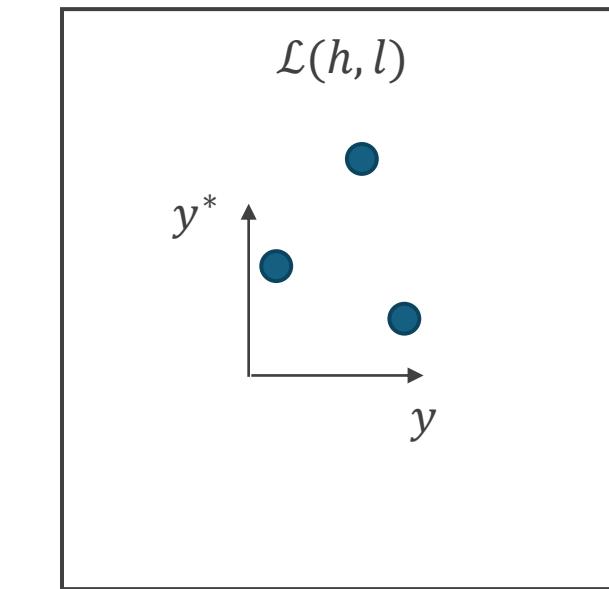
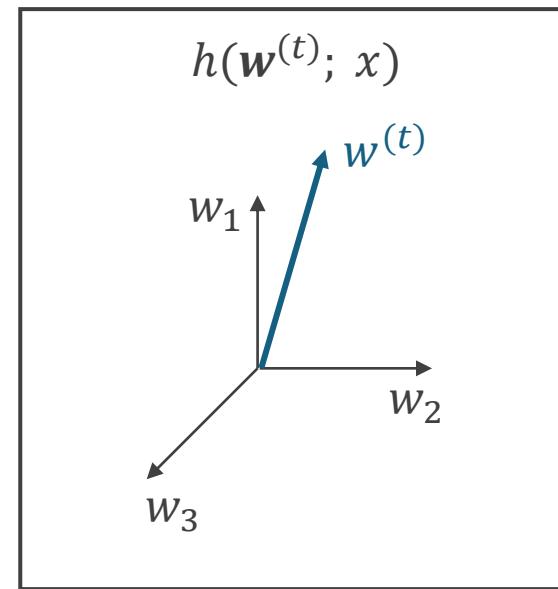
1. Sample mini-batch



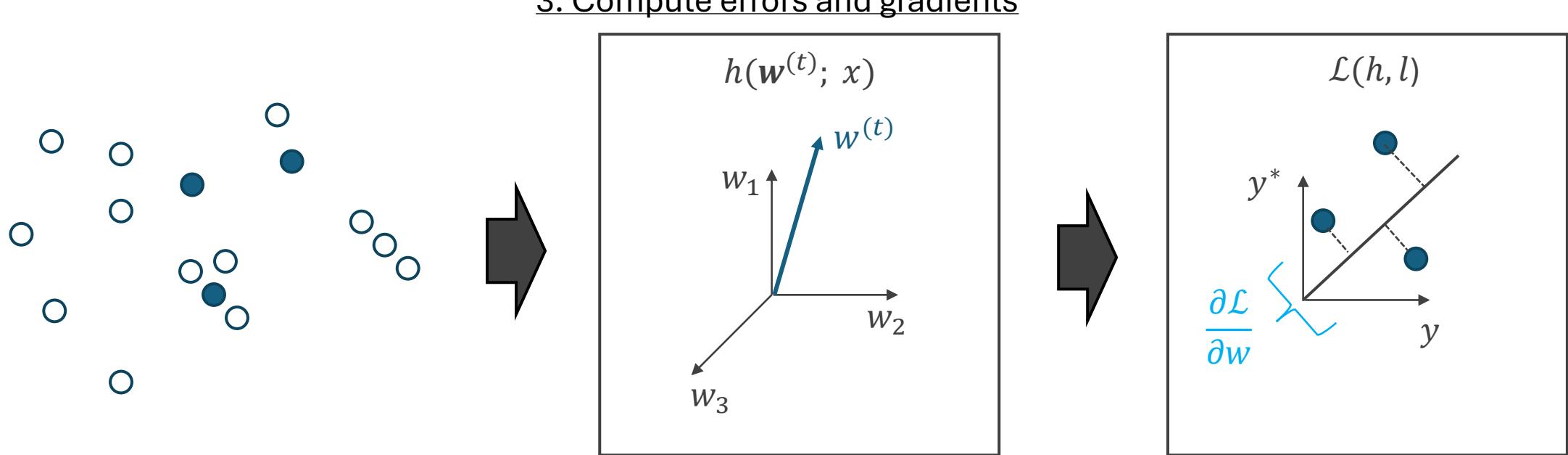
Stochastic gradient descent



2. Forward prop

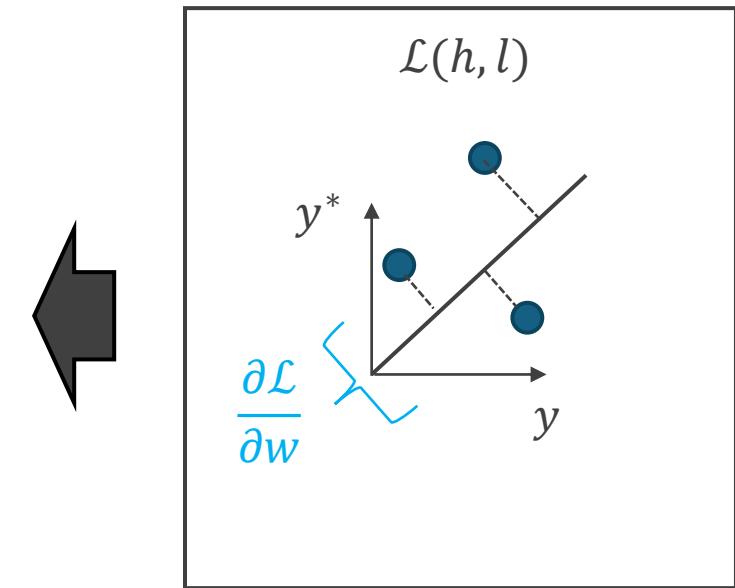
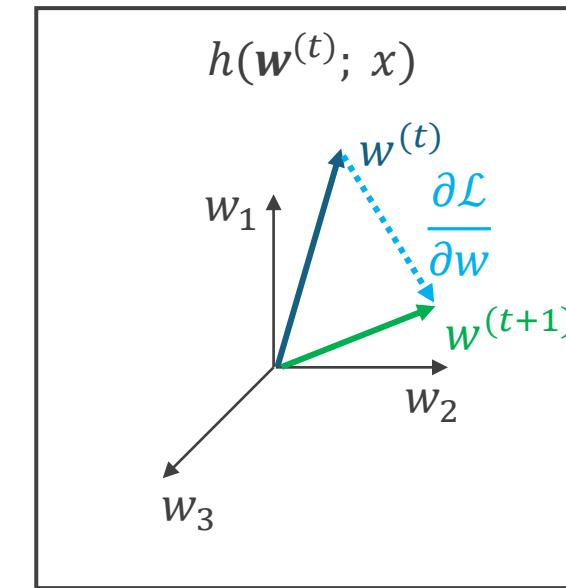
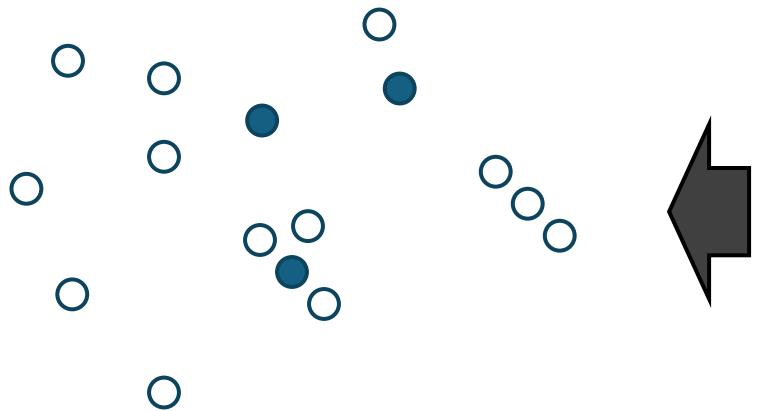


Stochastic gradient descent



Stochastic gradient descent

4. Update model parameters and repeat



In a nutshell

First, define your neural network

$$y = h_L \circ h_{L-1} \circ \cdots \circ h_1(x)$$

where each module h_l comes with parameters \mathbf{w}_l

Finding an “optimal” neural network means minimizing a loss function

$$\mathbf{w}^* = \operatorname{argmin}_{\mathbf{w}} L(\mathbf{w}) = \sum_{(x,y) \in (X,Y)} \mathcal{L}(f(x, \mathbf{w}), y) + \lambda \Omega(\mathbf{w})$$

Rely on stochastic gradient descent methods to obtain desired parameters

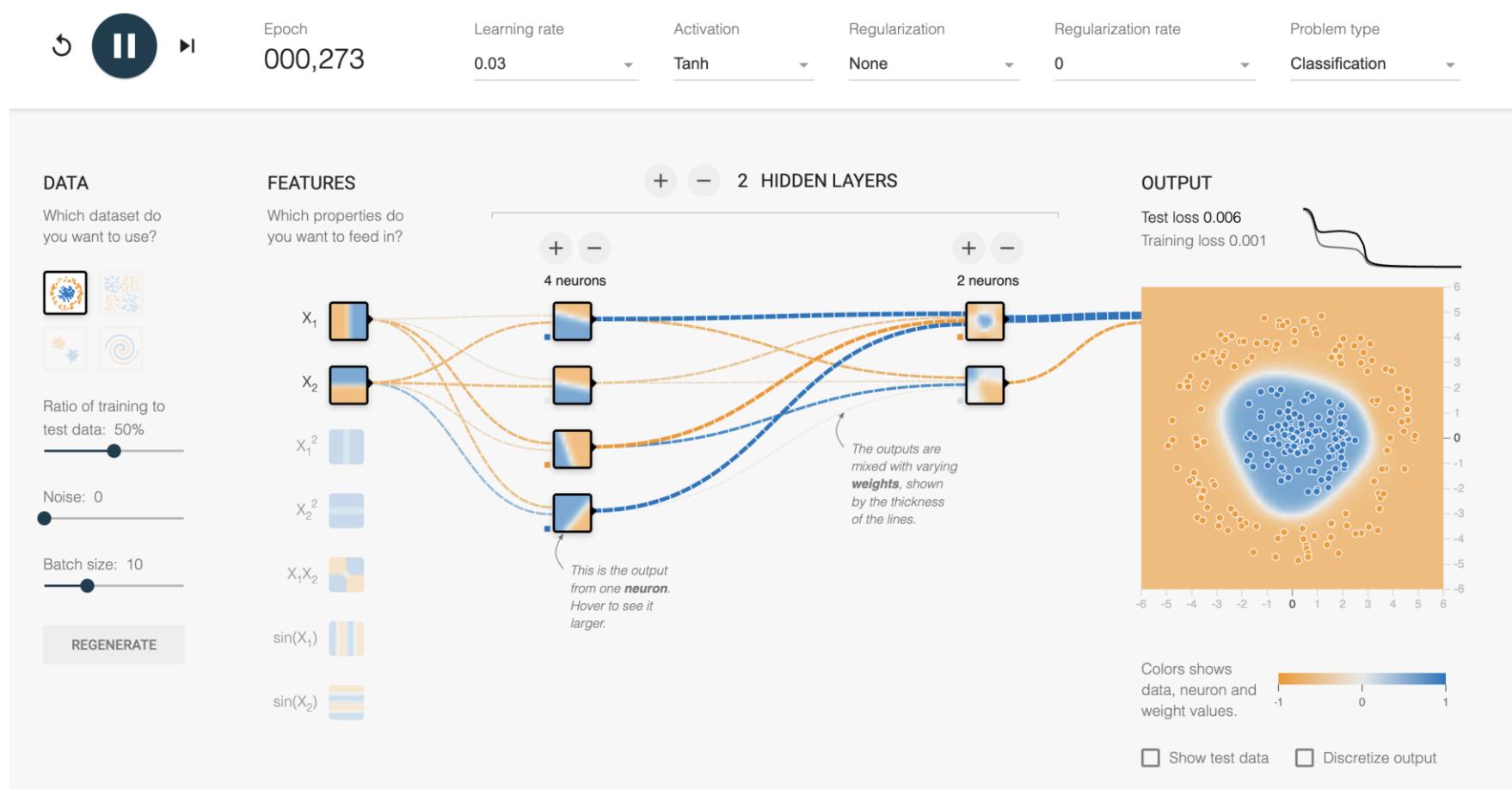
$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \frac{dL}{d\mathbf{w}}$$

where η is the step size or learning rate.

Gradient vs Stochastic Gradient Descent

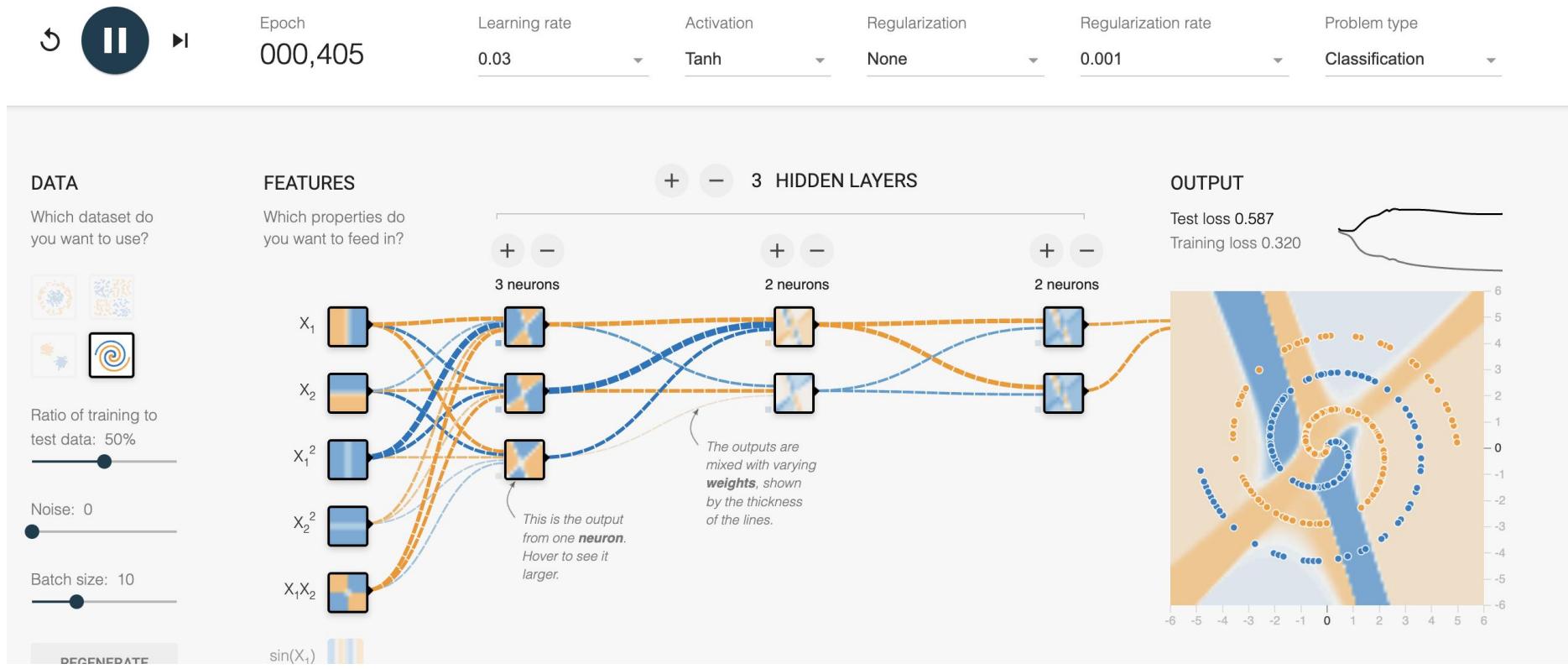
Gradient Descent	Stochastic Gradient Descent
Computes gradient using the whole training dataset	Computes gradient using a single training sample
Not suggested for huge training samples	Can be used for large training samples
Deterministic in nature	Stochastic in nature
No random shuffling of points required	Shuffling needed. More hyperparameters, e.g. batch size
Can't escape shallow local minima easily	Can escape shallow local minima more easily
Convergence is slow	Reaches the convergence faster

Practical examples



<https://playground.tensorflow.org/>

Practical examples



Challenges of optimizing deep networks

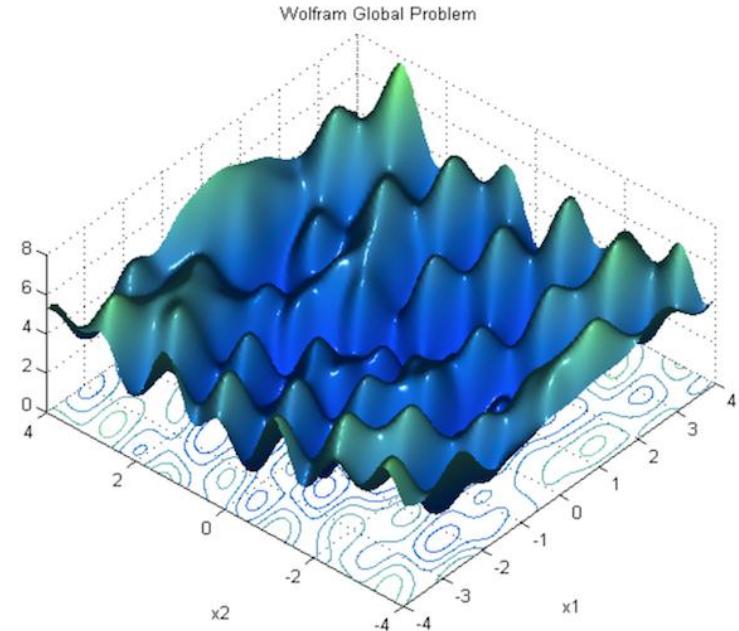
Neural network training is **non-convex** optimization.

- Involves a function which has multiple optima.
- Extremely difficult to locate the global optimum.

This raises many problems:

- How do we avoid getting stuck in local optima?
- What is a reasonable learning rate to use?
- What if the loss surface morphology changes?
- ...

One of the main theory in deep learning is that every local minima is a global one, the function appears like a sinusoid (sin or cos). The optimization seems to work too well to find only local minima, so maybe they are all global.



Main challenges in optimization

1. Ill conditioning → a strong gradient might not even be good enough
2. Local optimization is susceptive to local minima
3. Ravines, plateaus, cliffs, and pathological curvatures
4. Vanishing and exploding gradients
5. Long-term dependencies

1. Ill conditioning

Hessian matrix H

Square matrix of second-order partial derivatives of a *scalar-valued function*.

The *Hessian* describes the local *curvature* of a function of many variables.

The Hessian matrix is symmetric.

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

1. Ill conditioning

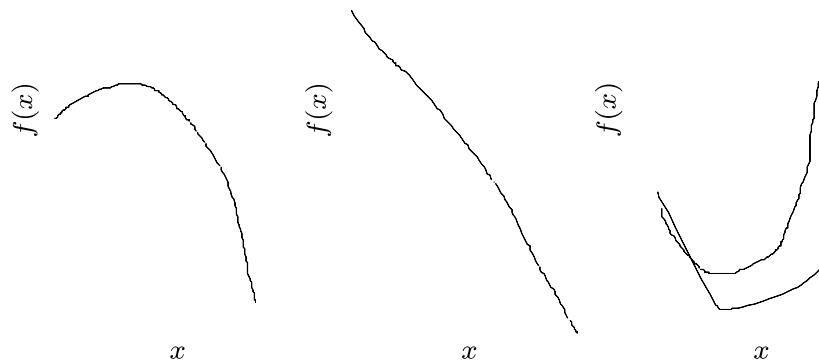
Curvature is determined by the second derivative

Negative curvature: cost function decreases faster than the gradient predicts.

No curvature: the gradient predicts the decrease correctly.

Positive curvature: the function decreases slower than expected and eventually begins to increase.

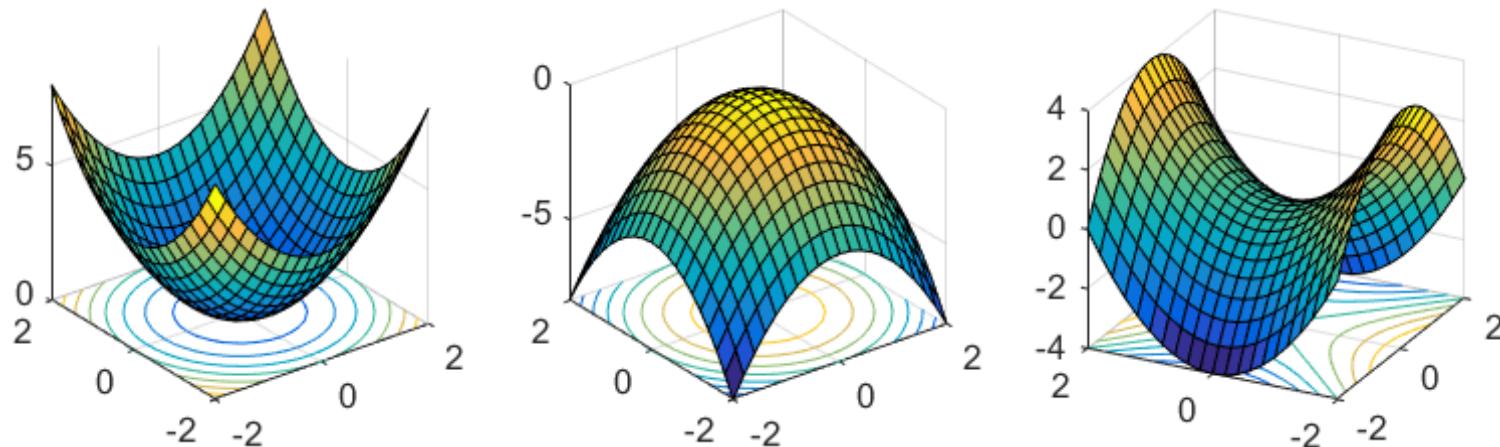
Negative curvature No curvature Positive curvature



1. Ill conditioning

Critical points – Hessian matrix

- *A local minimum:* positive definite (all its eigenvalues are positive)
- *A local maximum:* negative definite (all its eigenvalues are negative)
- *A saddle point:* at least one eigenvalue is positive and at least one eigenvalue is negative. Why is this bad?



1. Ill conditioning

Consider the Hessian matrix H has an eigenvalue decomposition.

Its condition number is

$$\overline{\max_{i,j} \left| \frac{\lambda_i}{\lambda_j} \right|}$$

This is the ratio of the magnitude of the largest (i) and smallest eigenvalue (j).

Measures show much the second derivatives differ from each other.

With a poor (large) condition number, gradient descent performs poorly.

- In one direction derivative increases rapidly, in another it increases slowly.
- It also makes it difficult to choose a good step size.

2. Local minima

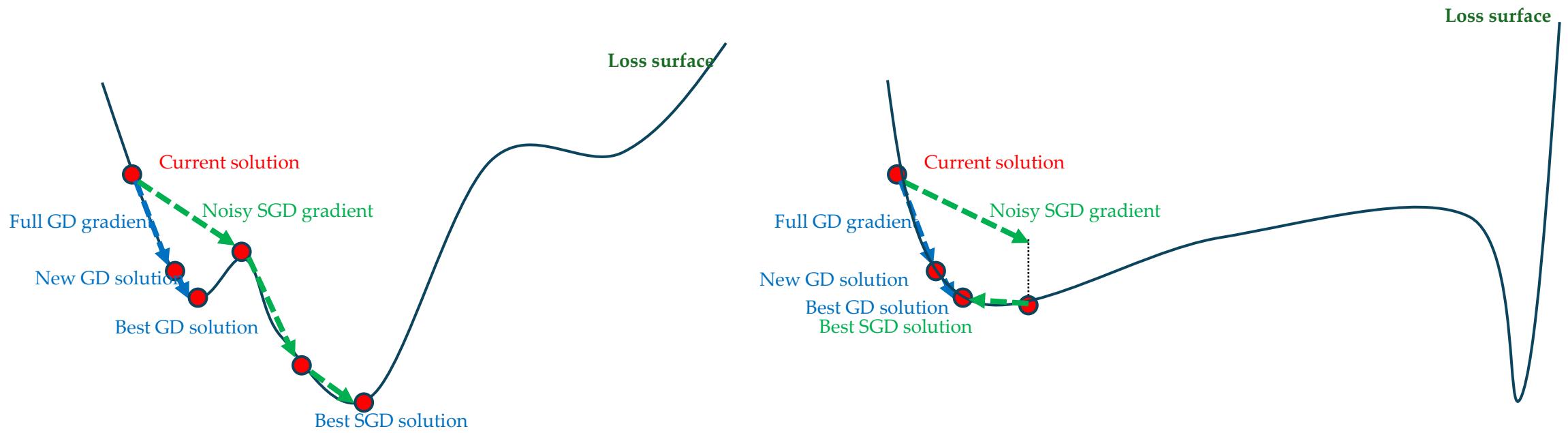
Model identifiability

- A model is said to be identifiable if a sufficiently large training set can rule out all but one setting of the model's parameters.
- Models with latent variables are often not identifiable because we can obtain equivalent models by exchanging latent variables with each other.

Local minima can be extremely numerous

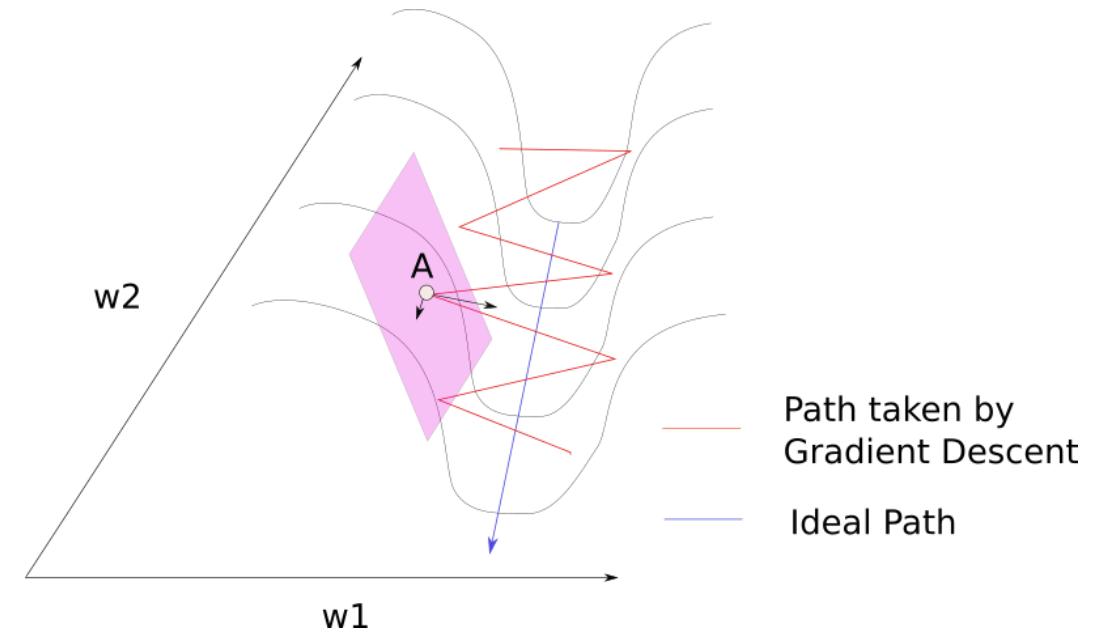
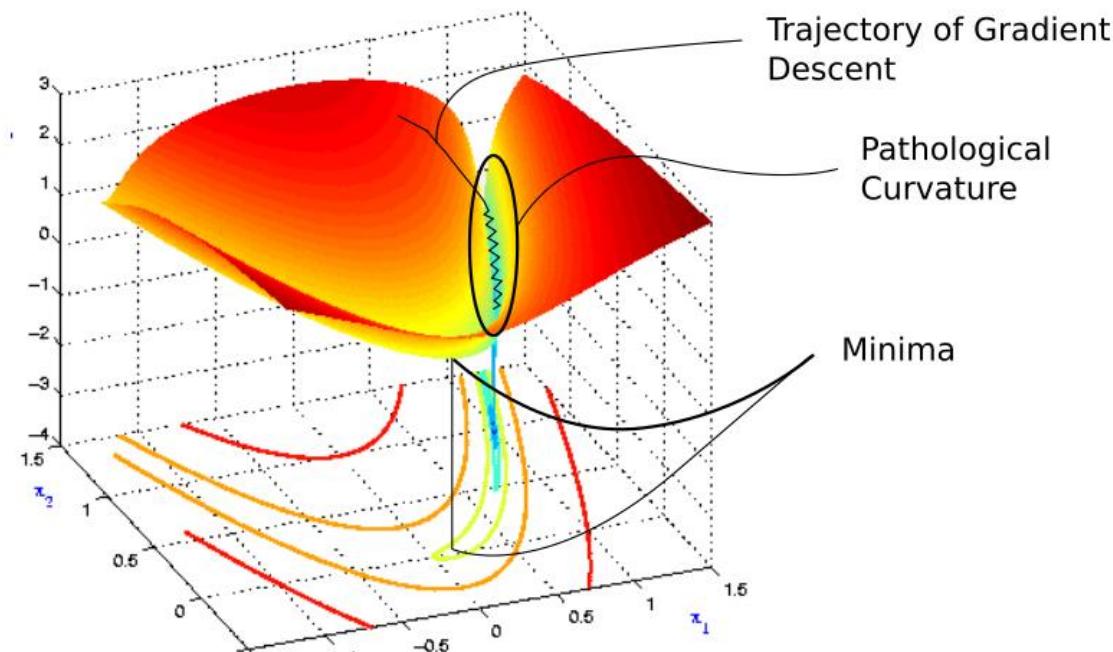
- However, all local minima from non-identifiability are equivalent in cost function value.
- Those local minima are not a problematic form of non-convexity.
- The other local minima (next slides) are.

Local minima



With gradient descent, we are blind to what the landscape looks like.

3. Ravines



Areas where the gradient is large in one direction and small in others.

3. Plateaus and flat areas

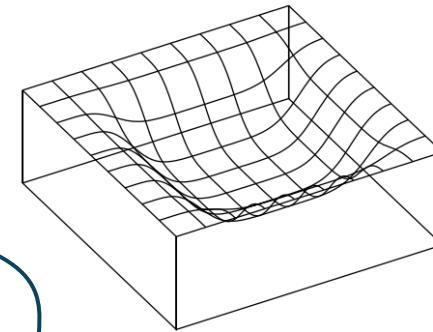
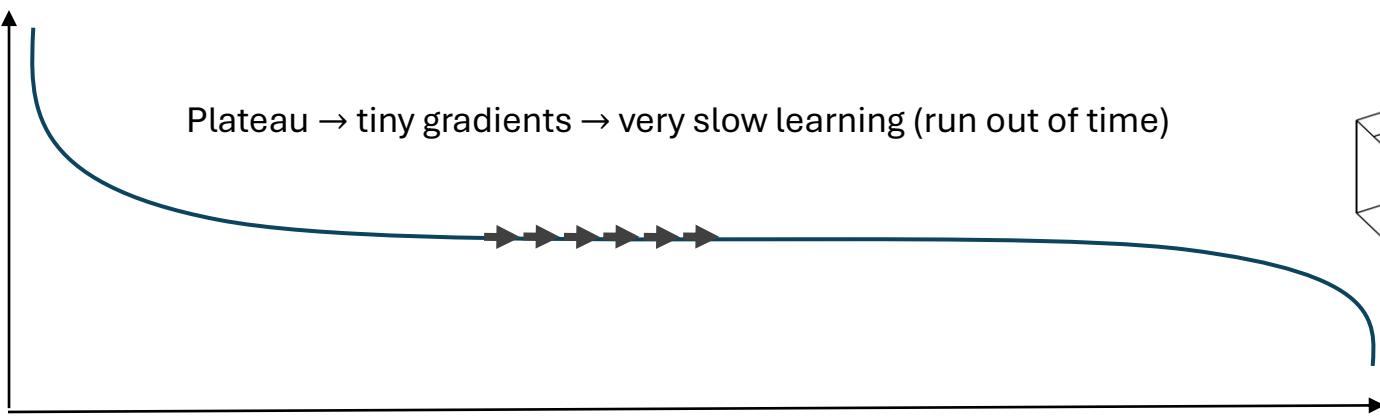


Figure 1: Example of a “flat” minimum.

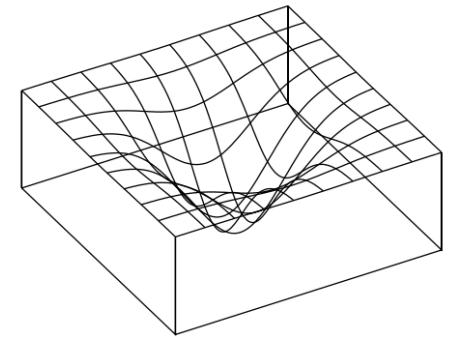


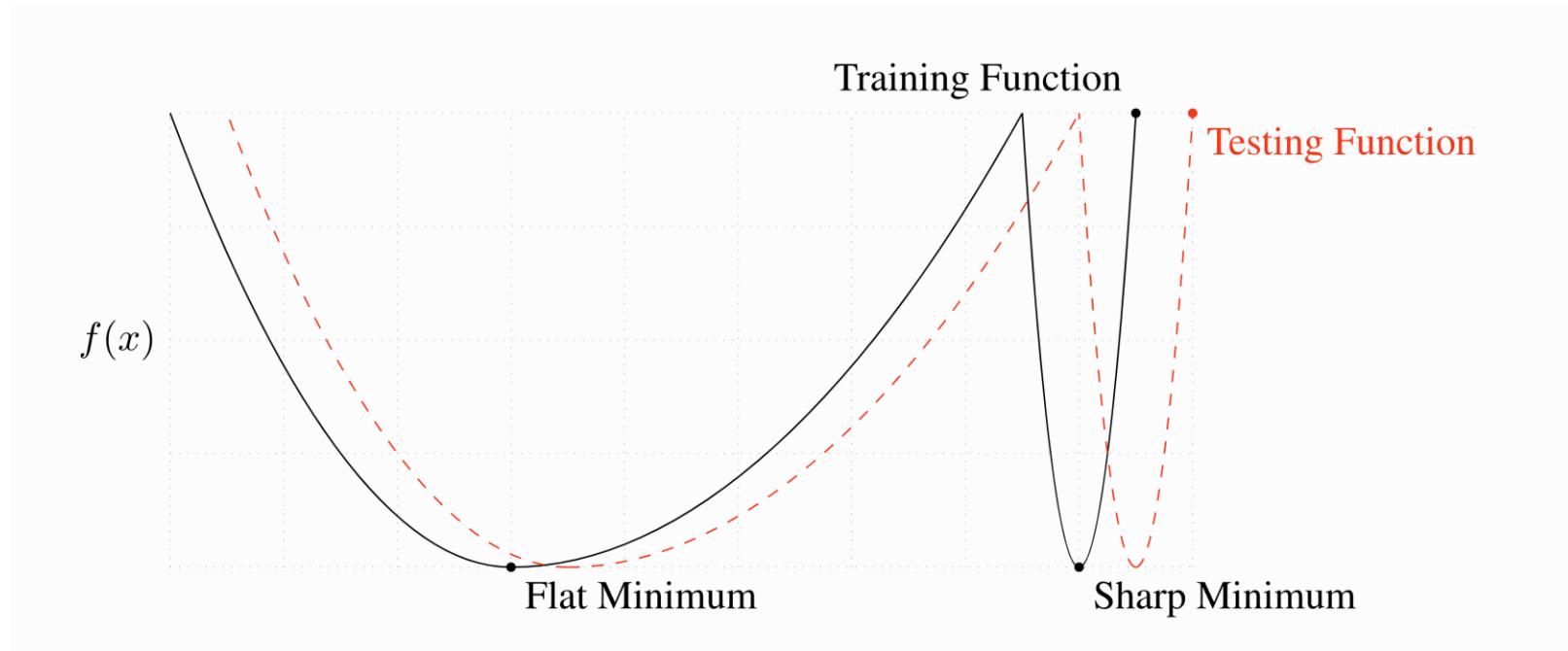
Figure 2: Example of a “sharp” minimum.

[Link](#)

Near zero gradients in flat areas, hence no learning.

Too step minima also a problem...

On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. Keskar et al. ICLR 2017



If the minimum is sharp we could be farther on the test minima respect to the cases with flatter minimum

Even if you miss the minimum of the test distribution slightly, still good results.

4. Cliffs and exploding gradients

Neural networks with many layers often have steep regions resembling cliffs.

These result from the multiplication of several large weights together.

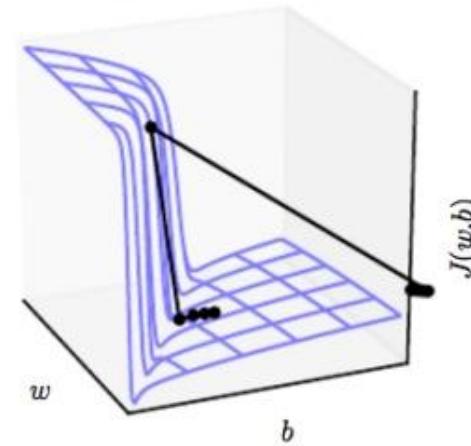
A simple trick: gradient clipping:

if $|g| > \eta$:

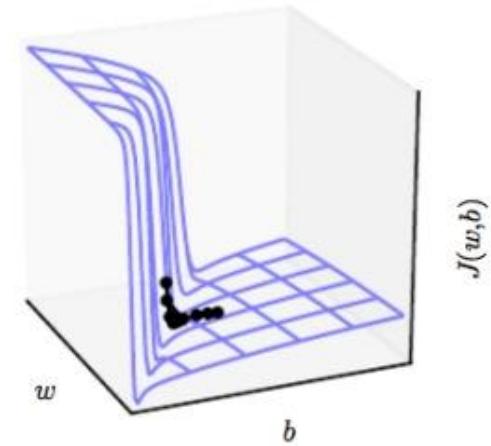
$$\mathbf{g} \leftarrow \frac{\eta \mathbf{g}}{\|\mathbf{g}\|}$$

Gradient clipping is a technique used to prevent the exploding gradient problem by limiting the magnitude of gradients

Without clipping



With clipping



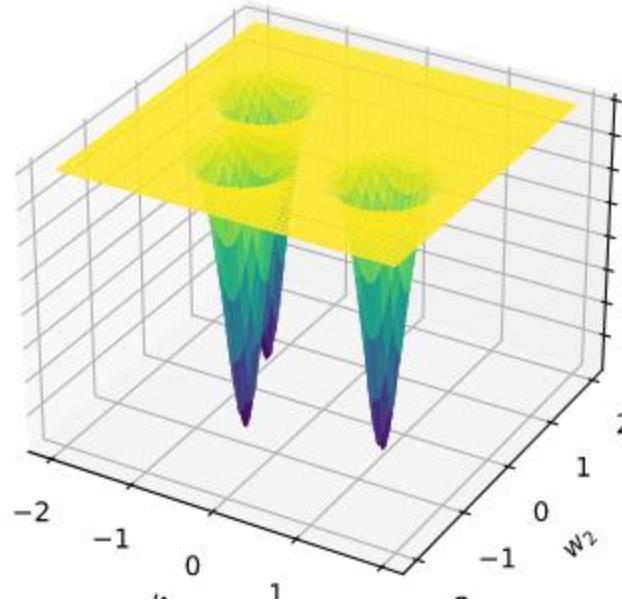
4. Flat areas and steep minima

When combining flat areas with very steep minima → very challenging

How do we even get to the area where the steep minima starts?

E.g.: temperate-scaled logits & cross-entropy: $p(y|x) = \text{softmax}(\text{logits}/0.00001)$

This is a problem because you don't know which direction follow since it is all flat, so it is difficult for gradient descent to spot local minima



5. Long-term dependencies

Especially for networks with many layers or recurrent neural networks.

The vanishing and exploding gradient problem

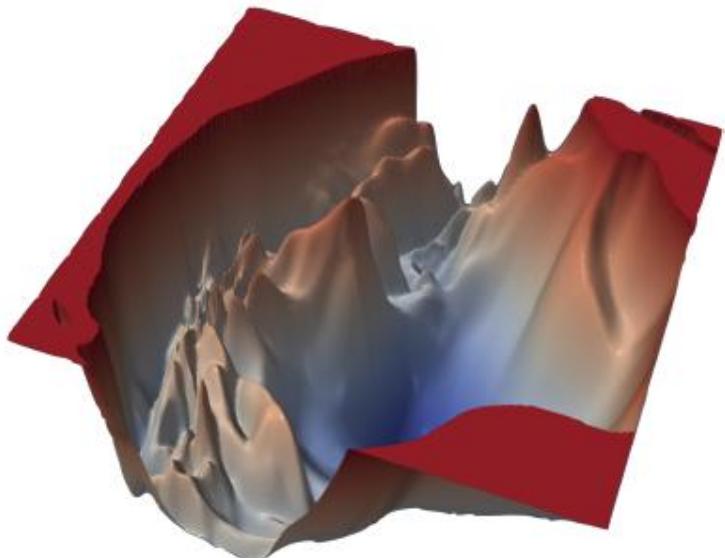
- Certain functions lead to a scaling of the gradient (potentially often).
- Vanishing gradients -> no direction to move
- Exploding gradients -> learning unstable.

For training-trajectory dependency: hard to recover from a bad start!

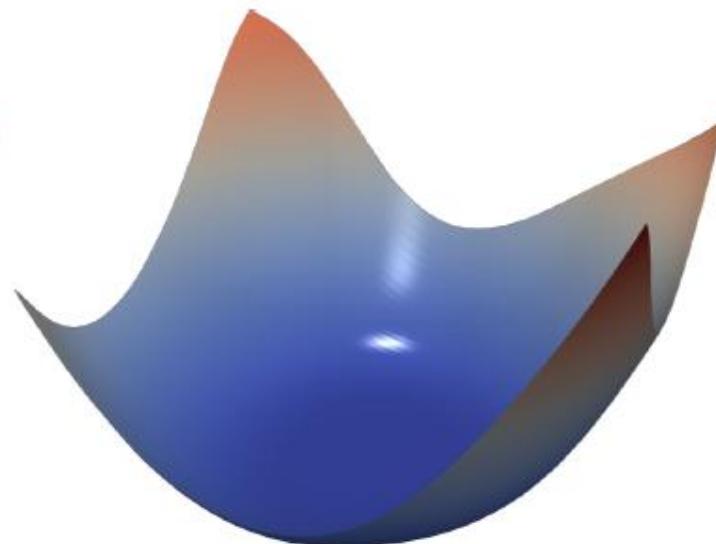
Is gradient descent even a good idea?

Global minima and local minima are nearby – Choromanska et al. (2015).

Architecture design and tricks have huge impact on loss landscapes (positively).



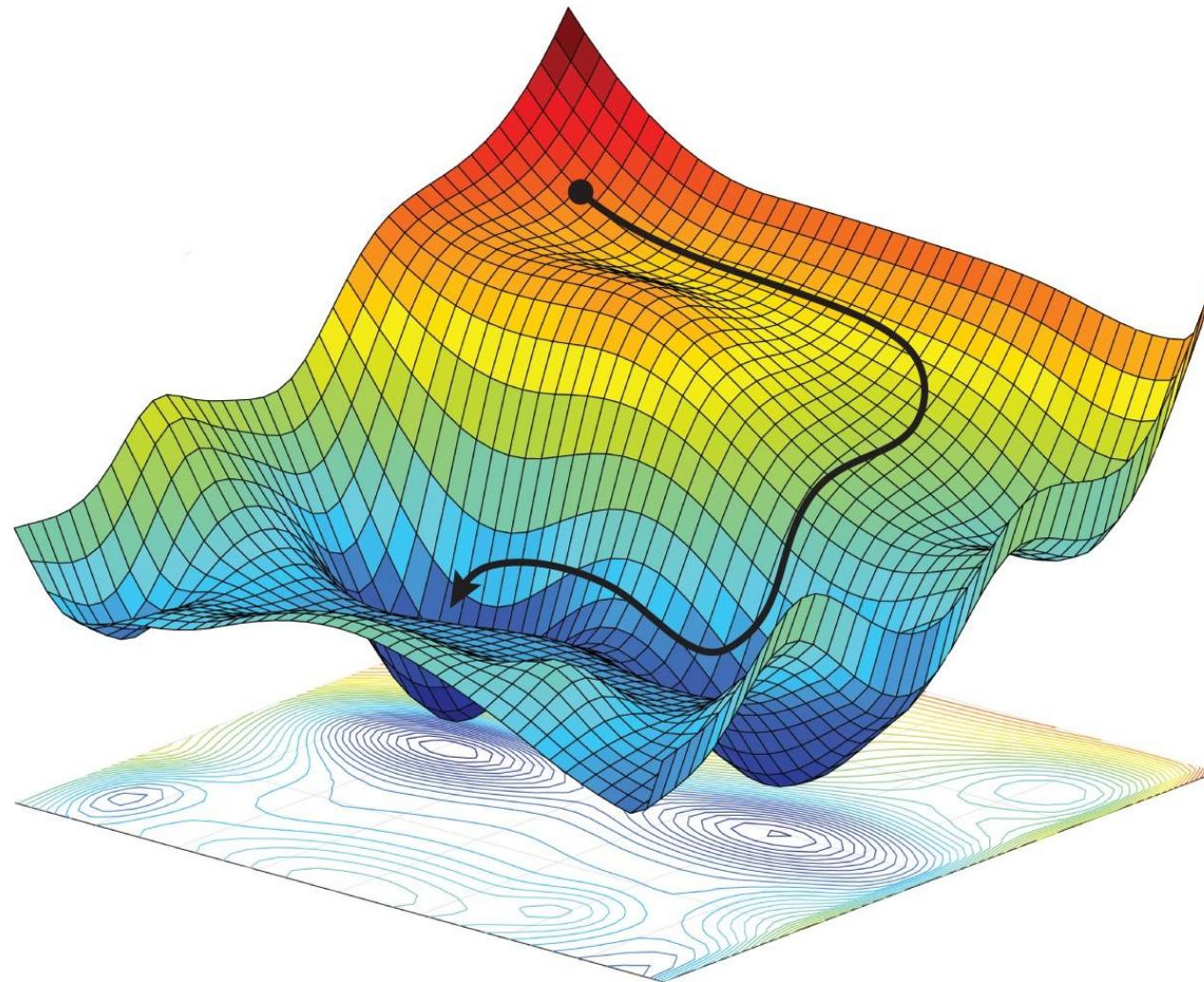
(a) ResNet-110, no skip connections



(b) DenseNet, 121 layers

Break

Advanced optimizers



Gradient descent itself can be enhanced

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \frac{d\mathcal{L}(\mathbf{w})}{d\mathbf{w}}$$

Can we improve the learning rate setting?

Can we get a better or more useful gradient?

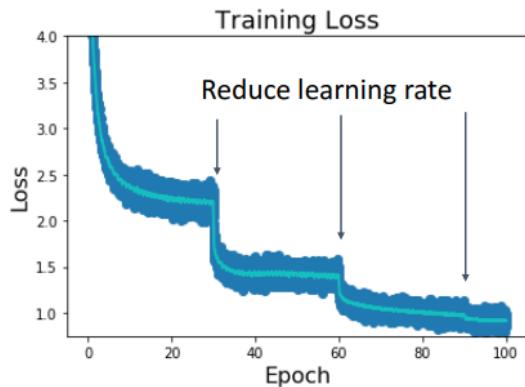
Setting the learning rate

Truly an empirical endeavour, unique to each problem and dataset.

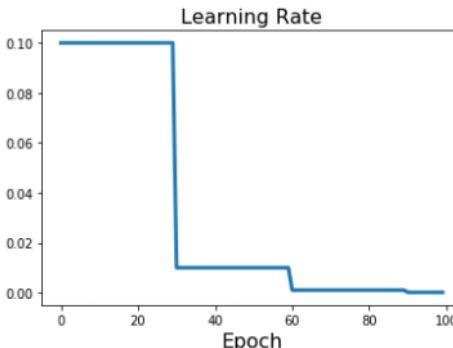
Big trick: learning rate schedulers.

Do not use a fixed learning rate but rather use a scheduler that in function of the epochs reduces the learning rate

Learning Rate Decay: Step



Step: Reduce learning rate at a few fixed points.
E.g. for ResNets, multiply LR by 0.1 after epochs
30, 60, and 90.



Improving gradient descent

Stochastic Gradient Descent with momentum.

Nesterov momentum.

Stochastic Gradient Descent with adaptive learning rates.

E.g., AdaGrad, RMSProp, Adam

Second-order approximation, such as Newton's methods.

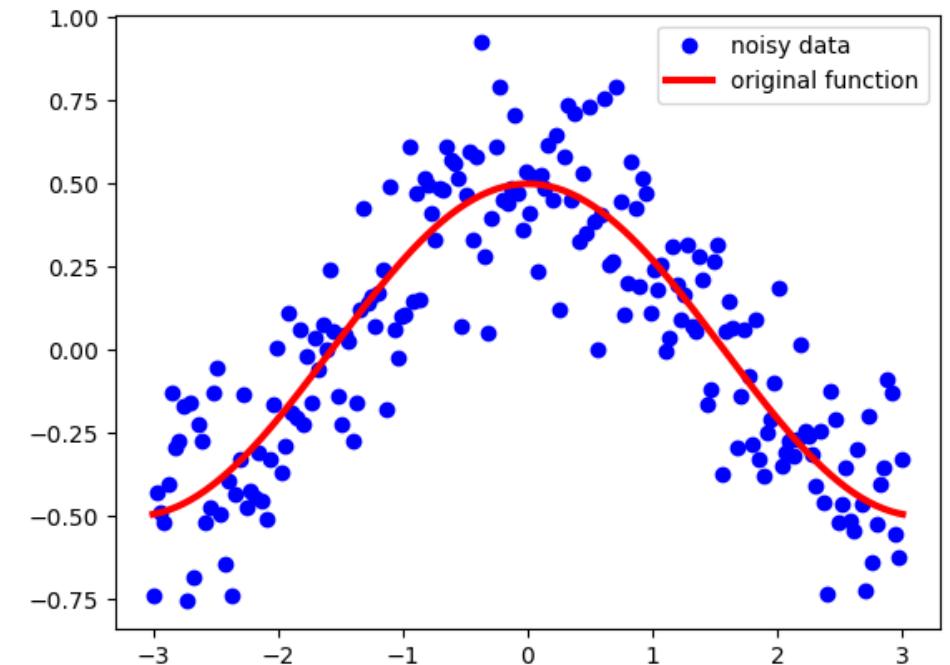
Momentum

Designed to accelerate learning, especially when loss is of high curvature.

We can understand momentum via exponentially weighted moving averages.

Suppose we have a sequence S which is noisy:

Instead of using only my mini batch I keep track also of the previous gradients, thus having a moving average. This technique helps the optimizer move faster through the parameter space and navigate challenging landscapes more effectively.



Momentum

Exponentially weighted averages:

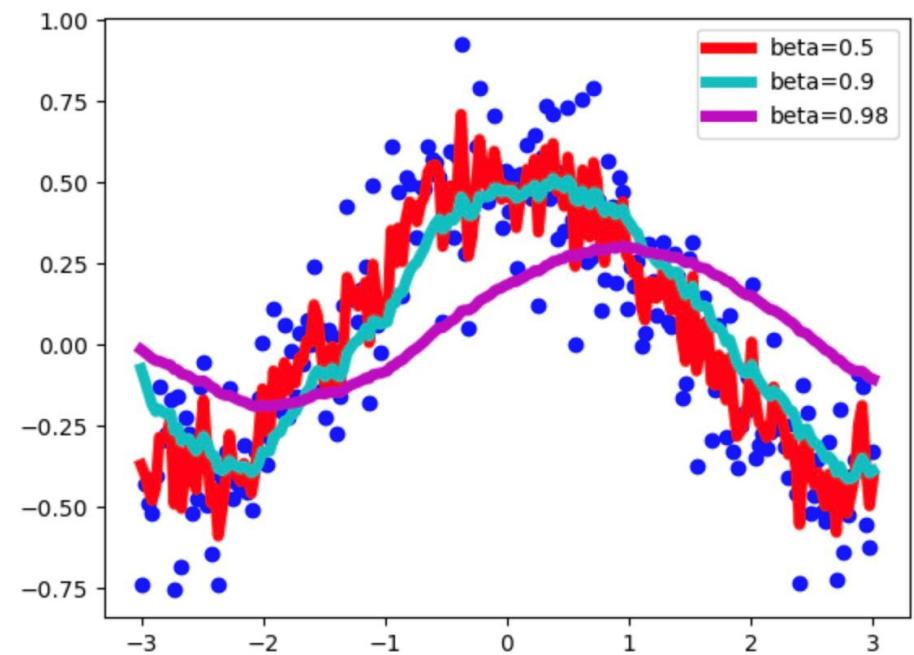
$$V_t = \beta V_{t-1} + (1 - \beta) S_t, \quad \beta \in [0, 1], \quad V_0 = 0$$

Small β leads to more fluctuations.

- $\beta=0.9$ provides a good balance

Bias correction.

- E.g., $V_1 = \beta V_0 + (1 - \beta) S_1$: biased towards V_0
- $V_t = \frac{V_t}{1 - \beta}$



Stochastic gradient descent with momentum

dampens oscillations means making less strong oscillations

Don't switch update direction all the time.

Maintain “*momentum*” from previous updates → dampens oscillations.

$$v_{t+1} = \gamma v_t + \eta_t g_t, \quad \eta_t = \text{learning rate}$$

$$w_{t+1} = w_t - v_{t+1}$$

Exponential averaging keeps steady direction.

Example: $\gamma = 0.9$ and $v_0 = 0$

- $v_1 \propto -g_1$
- $v_2 \propto -0.9g_1 - g_2$
- $v_3 \propto -0.81g_1 - 0.9g_2 - g_3$

Standard gradient descent faces several limitations that momentum addresses. In regions with steep gradients in some directions and flat gradients in others, vanilla gradient descent oscillates heavily, slowing convergence. Additionally, it can get stuck in local minima or saddle points where gradients are small. On flat plateaus where gradients diminish, progress becomes extremely slow even when far from the optimum

The algorithm maintains a velocity term that represents the accumulated momentum of previous gradient updates. At each iteration, the velocity is updated as an exponentially weighted combination of the previous velocity and the current gradient

The momentum term produces several advantages. It accelerates convergence by allowing the optimizer to build up speed in consistent directions, much like a ball rolling down a hill gains momentum to carry it over small bumps and plateaus. It dampens oscillations by smoothing out the gradient updates across iterations, reducing the zig-zag pattern common in standard gradient descent. The technique also helps escape local minima and saddle points because accumulated momentum can carry the optimization through regions of small gradients.

Adding momentum is easy

SGD

$$w_{t+1} = w_t - \alpha \nabla f(w_t)$$

```
for t in range(num_steps):
    dw = compute_gradient(w)
    w -= learning_rate * dw
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(w_t)$$

$$w_{t+1} = w_t - \alpha v_{t+1}$$

```
v = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    v = rho * v + dw
    w -= learning_rate * v
```

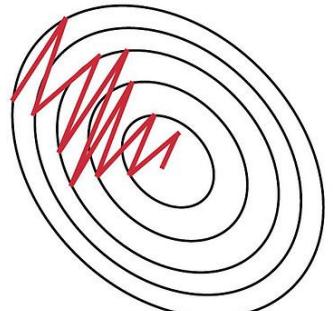
- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

Physical interpretation of momentum

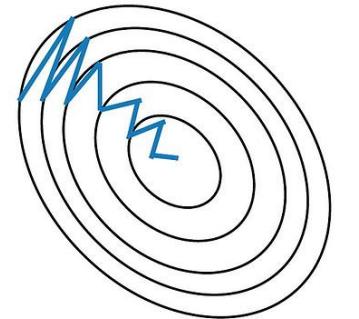
See gradient descent as rolling a ball down a hill.

The ball accumulates momentum, gaining speed down a straight path.

Momentum term increases for dimensions whose gradients point in the same direction and reduces for dimensions whose gradients change directions.



without momentum



with momentum

Nesterov momentum

at iteration 0 v_0 is initialized to 0

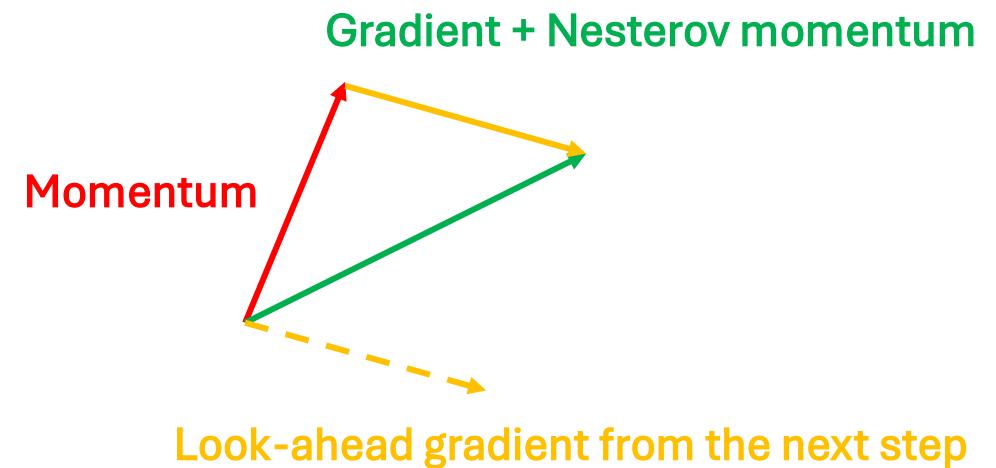
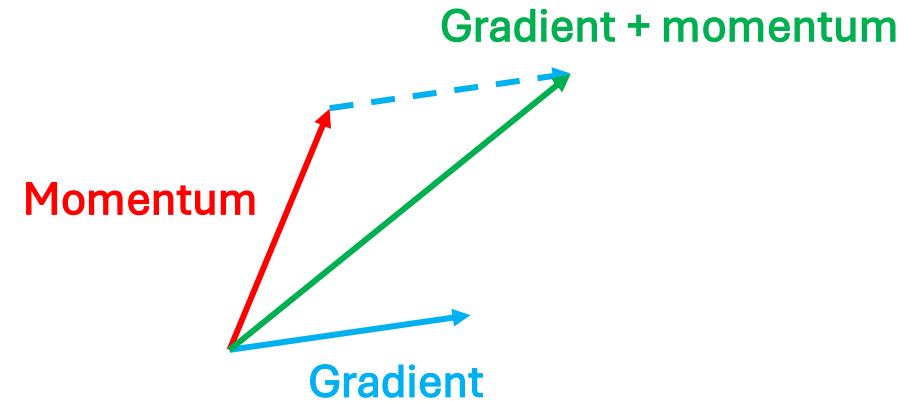
Use future gradient instead of current gradient:

$$v_{t+1} = \gamma v_t + \eta_t \nabla_w \mathcal{L}(w_t - \gamma v_t)$$

$$w_{t+1} = w_t - v_{t+1}$$

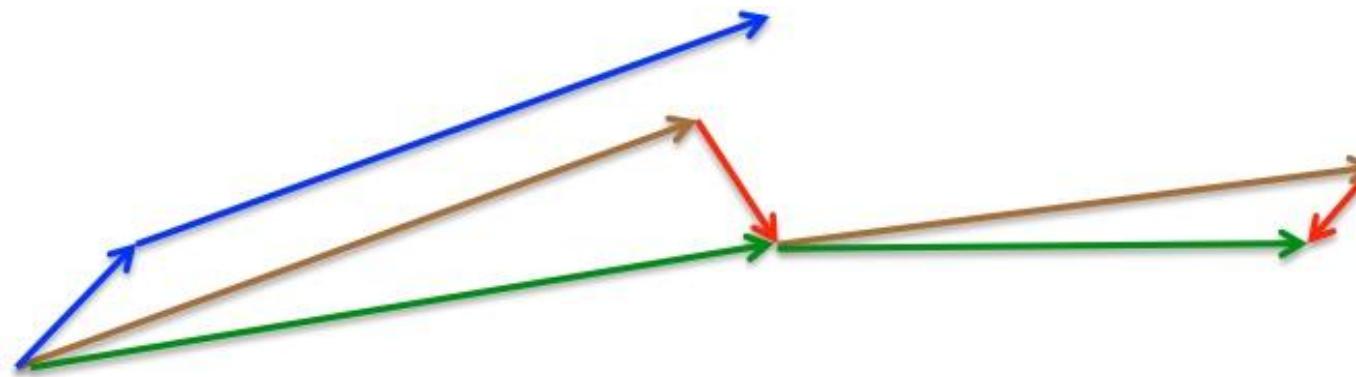
Prevents us from going too fast.

Also increases responsiveness.



Nesterov momentum

First make a big jump in the direction of the previous accumulated gradient.
Then measure the gradient where you end up and make a correction.



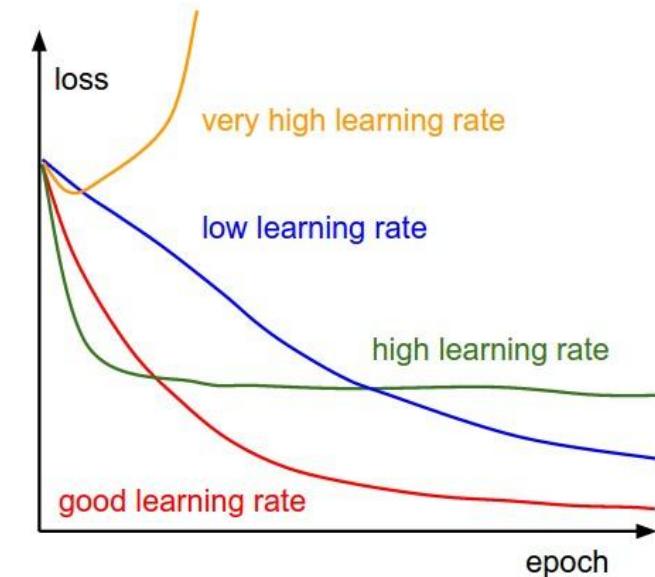
brown vector = jump;
red vector = correction;
green vector = accumulated gradient;
blue vectors = standard momentum

Adaptive step sizes

A fixed learning rate is difficult to set.

Also has significant impact on performance and sensitive.

Is it possible to have a separate adaptive learning rate for each parameter?



AdaGrad

Adaptive Gradient Algorithm – Adagrad:

- The learning rate is adapted **component-wise** to the parameters by incorporating knowledge of past observations.
- Rapid decrease in learning rates for parameters with large partial derivatives.
- Smaller decrease in learning rates for parameters with small partial derivatives.

Schedule

$$\begin{aligned} w_{t+1} &= w_t - \frac{\eta}{\sqrt{r + \varepsilon}} \odot g_t, \\ \text{where } r &= \sum_t (\nabla_w \mathcal{L})^2 \end{aligned}$$

learning rate
gradient
epsilon (avoid division by zero)
accumulation of squared gradients

I want to decrease my learning rate if I have large partial derivatives, and increase it otherwise

Instead of accumulating all squared gradients, RMSprop maintains an exponentially decaying moving average of the squared gradients.

RMSprop

Decay hyper-parameter (usually 0.9)

Schedule

- $r_t = \alpha r_{t-1} + (1 - \alpha) g_t^2$
- $v_t = \frac{\eta}{\sqrt{r_t} + \epsilon} \odot g_t$
- $w_{t+1} = w_t - v_t$



The denominator normalizes the gradient update by dividing by the root mean square of past squared gradients, with ϵ (typically 1e-8) preventing division by zero. This means parameters with large gradients receive smaller effective learning rates, while parameters with small gradients maintain relatively larger learning rates

Large gradients, e.g., too “noisy” loss surface

- Updates are tamed

tamed means to make less powerful

Small gradients, e.g., stuck in plateau of loss surface

- Updates become more aggressive

SGD with momentum and Adam are the most used

Adam

One of the most popular algorithms.

Combines RMSprop and momentum.

- Computes adaptive learning rate for each parameter.
- Keeps an exponentially decaying average of past gradients (momentum).
- Introduces bias corrections to the estimates of moments.

Can be seen as a heavy ball with friction, hence a preference for flat minima.

Adam

$$\begin{aligned}m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t && \text{momentum of my gradients} \\v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2\end{aligned}$$

RMSprop

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Bias corrections

this boost the values at the start temporarily, 1-beta at the number of steps I took

$$u_t = \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

$$w_{t+1} = w_t - u_t$$

At start t = 1, After all epochs t is very high so the value of β_1^t converges to 0, and the denominator converges to 1

Recommended values: $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$

Adaptive learning rate as RMSprop, but with momentum & bias correction

Adam is hyperparameter-free?

$$\begin{aligned}m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2\end{aligned}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

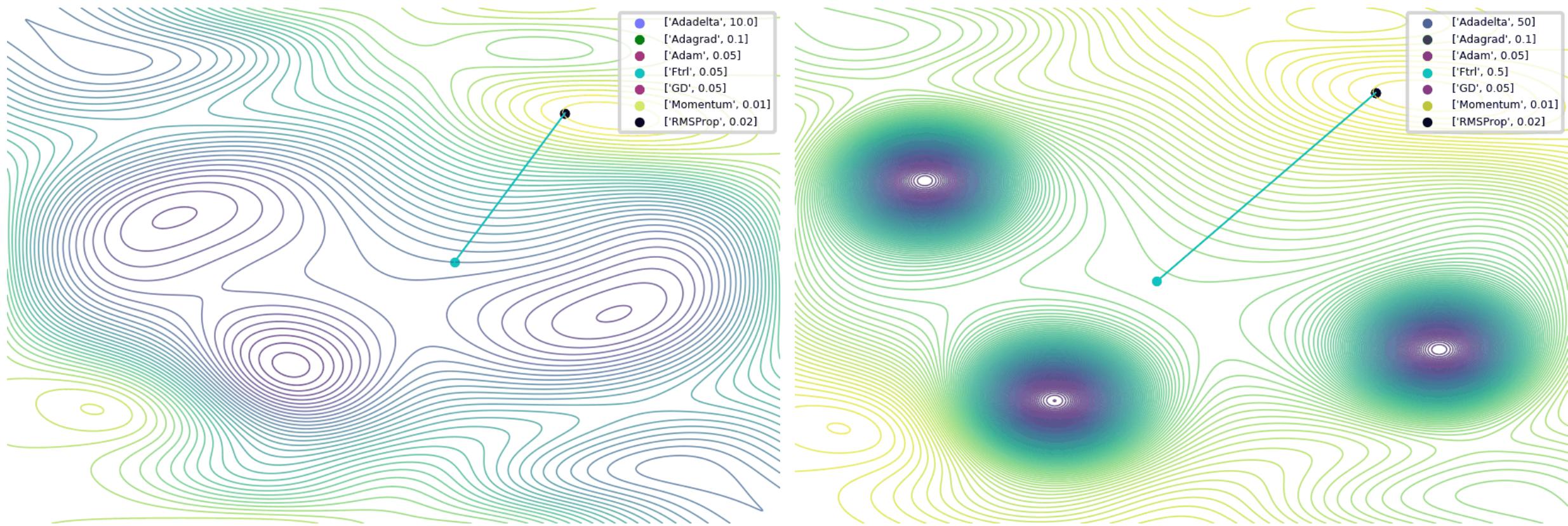
$$\begin{aligned}u_t &= \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \\w_{t+1} &= w_t - u_t\end{aligned}$$

it is NOT hyperpar free, instead it has more hyperpar respect to SGD (3 more). It allows to control over the learning rate across different network parametrs. Sometimes I will update more, other I will update less

It has 4 hyperpar, eta, epsilon, beta1, beta2

More robust to different settings, but many values to set.

Visual overview



Picture credit: [Jaewan Yun](#)

Which optimizer to use?

My go-to: SGD with momentum and learning rate decay.

For more complex models, Adam is often the preferred choice.

Adam with weight-decay (AdamW) is the standard for optimizing transformer architectures.

Oddity: even in “learning rate adjusting” optimizers like Adam, we add learning rate decays.

Interactive visualization

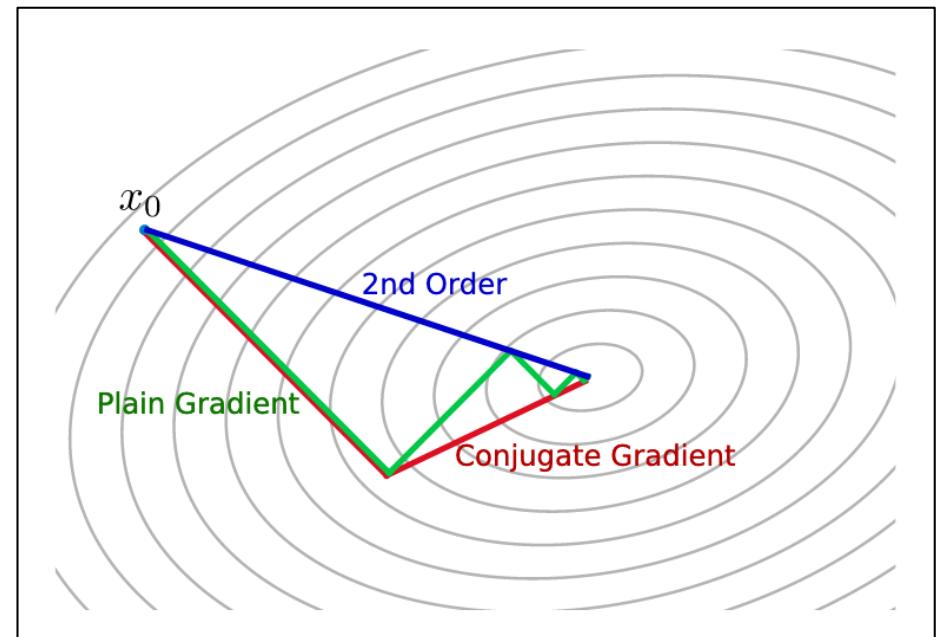
<https://emiliendupont.github.io/2018/01/24/optimization-visualization/>

Approximate second-order methods

SGD, Adam, etc are first-order: curvature information is ignored.

Benefits of second-order optimization:

- Better direction.
- Better step-size.
- Full step jumps directly to the minimum of the local squared approx.
- Additional step size reduction and
- Dampening becomes easy.



Newton's method

A second-order Taylor series expansion to approximate $J(\theta)$ near some point θ_0 , ignoring derivatives of higher order:

$$J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^\top \nabla_{\theta} J(\theta_0) + \frac{1}{2}(\theta - \theta_0)^\top H(\theta - \theta_0)$$

If we then solve for *the critical point* of this function, we obtain the **Newton parameter update rule**:

$$\theta^* = \theta_0 - H^{-1} \nabla_{\theta} J(\theta_0)$$

Computing the Hessian is expensive,
and also inverting it is worse

For a locally quadratic function, Newton's method jumps directly to the minimum.
If convex but not quadratic (there are higher-order terms), *update can be iterated*.

Why we use first order optimization

Disadvantages:

- Super slow: need to compute inverse of Hessian matrix each time.
- More restrictive: 2nd order derivative needs to be possible to compute.
- Limited impact: no major improvement found in practice.

Learning and reflection

Understanding Deep Learning: Chapter 6

Understanding Deep Learning: Chapter 7

Next lecture

Lecture	Title	Lecture	Title
1	Intro and history of deep learning	2	AutoDiff
3	Deep learning optimization I	4	Deep learning optimization II
5	Convolutional deep learning	6	Attention-based deep learning
7	Graph deep learning	8	From supervised to unsupervised deep learning
9	Multi-modal deep learning	10	Generative deep learning
11	What doesn't work in deep learning	12	Non-Euclidean deep learning
13	Q&A	14	Deep learning for videos

Thank you!



Deep Learning 1

2025-2026 – Pascal Mettes

Lecture 4

Deep learning optimization II

Previous lecture

Lecture	Title	Lecture	Title
1	Intro and history of deep learning	2	AutoDiff
3	Deep learning optimization I	4	Deep learning optimization II
5	Convolutional deep learning	6	Attention-based deep learning
7	Graph deep learning	8	From supervised to unsupervised deep learning
9	Multi-modal deep learning	10	Generative deep learning
11	What doesn't work in deep learning	12	Non-Euclidean deep learning
13	Q&A	14	Deep learning for videos

This lecture

Overfitting and regularization

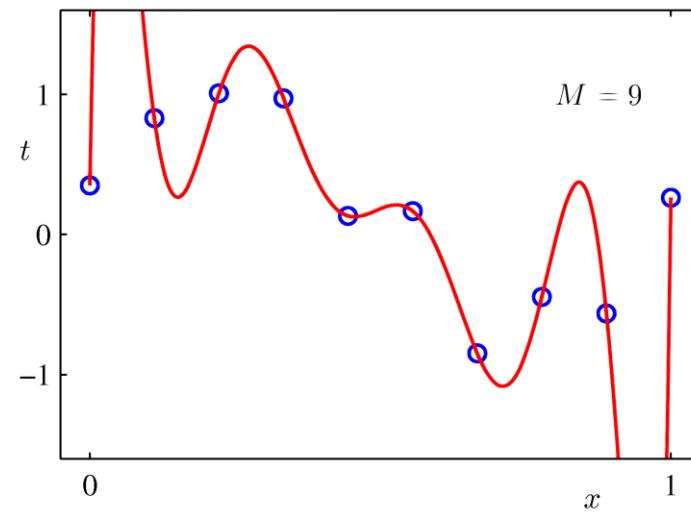
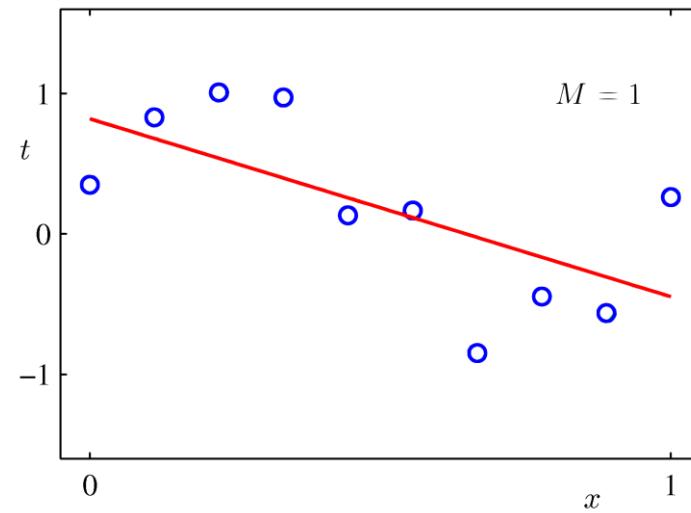
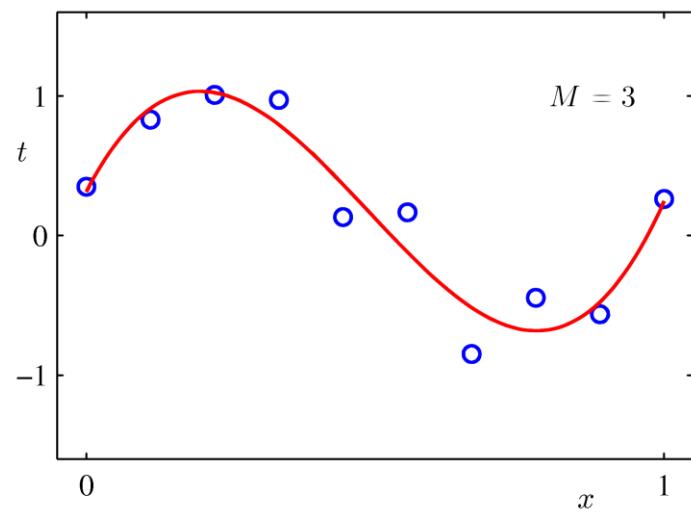
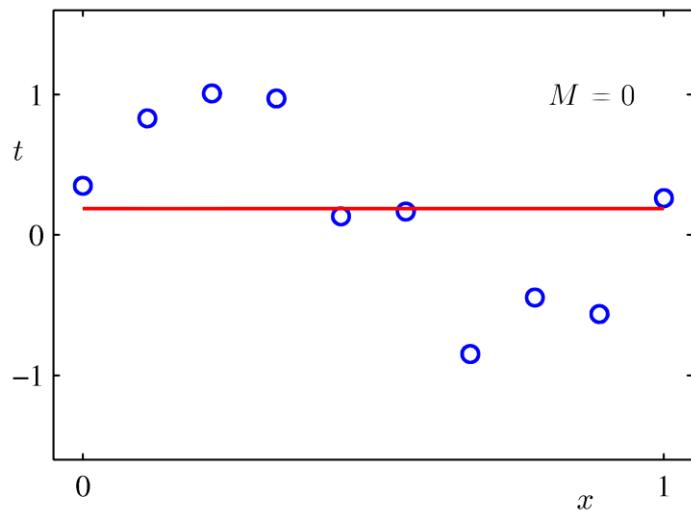
Initialization

Augmentation

Normalization

Hyperparameters

Which fit is best?



Bias-variance tradeoff

Bias

“The difference between an estimator’s expected value and the true value of the parameter being estimated”.

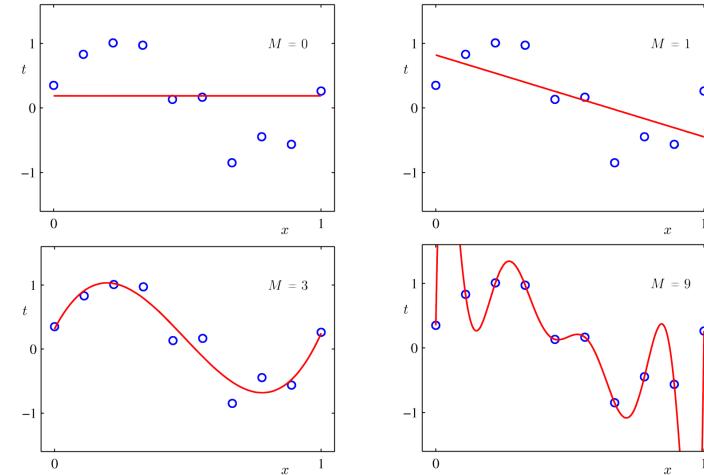
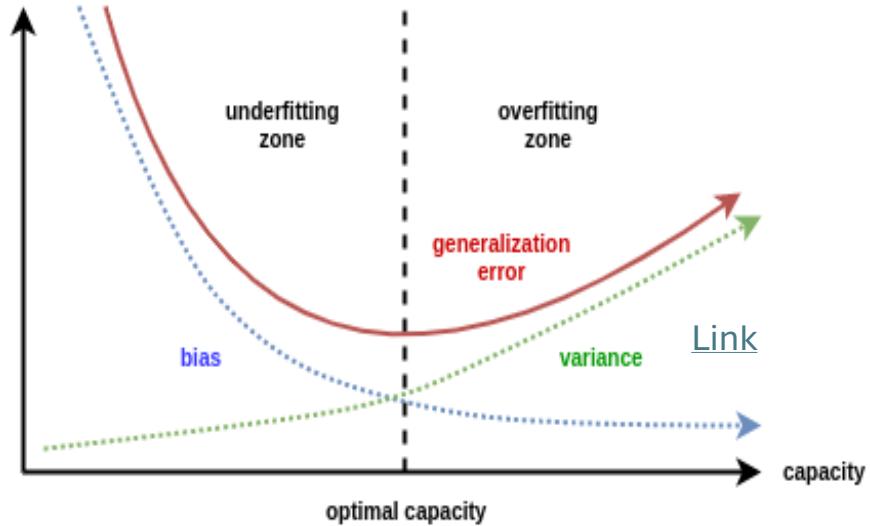
The bias error is an error from erroneous assumptions in the learning algorithm.

Variance

The amount that the estimate of the target function will change if different training data was used.

The variance is an error from sensitivity to small fluctuations in the training set.

Bias-variance tradeoff



High bias: algorithm misses relevant relations between features and targets.

Relates to underfitting, high bias is common in linear models.

High variance: algorithm uses random noise in training data for their modelling.

Relates to overfitting, high variance is common in deep non-linear models.

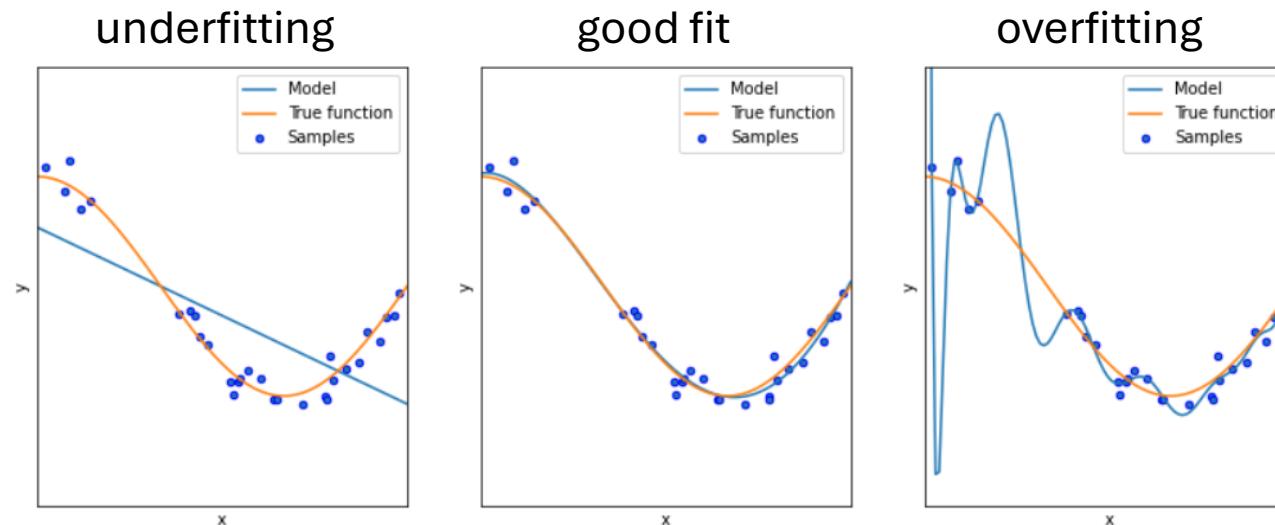
Overfitting

One of the key aspects to deal with when training neural networks.

Overfitted models perform poorly on new data from the same domain.

Low/zero training error is not automatically overfitting!

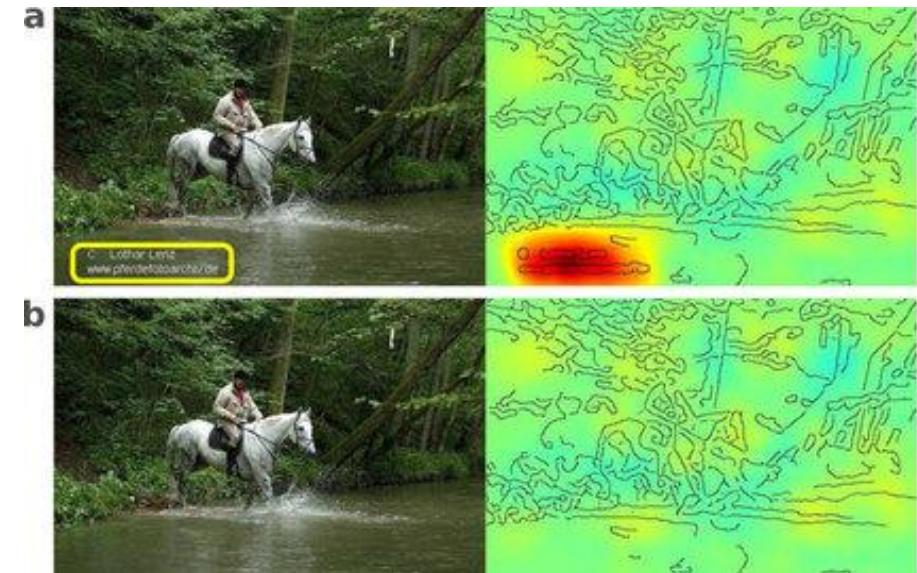
Only in combination with worse generalization as a function of training error.



Why would overfitting even happen?

We don't know the true data distribution

1. Complexity / parameter count \gg problem / data.
2. Overfitting especially common when dealing with co-occurrences.
3. Memorization (i.e., learning individual samples instead of their distribution).
4. Silly things you might have missed in your data.



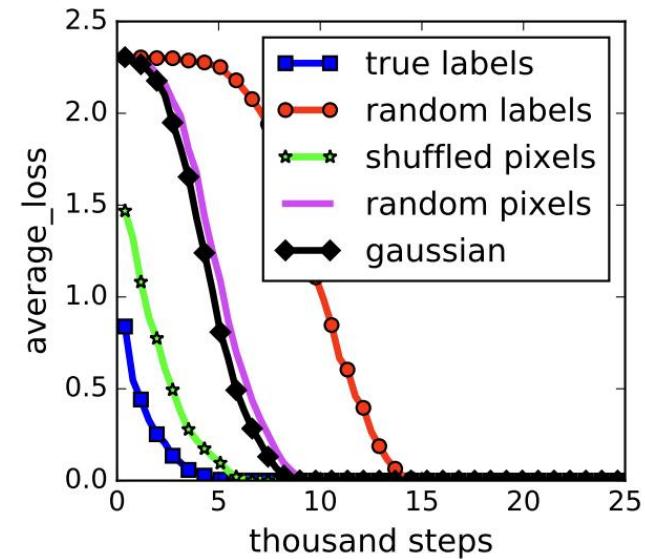
The severity of the overfitting problem

Randomization tests. At the heart of our methodology is a variant of the well-known randomization test from non-parametric statistics (Edgington & Ongena, 2007). In a first set of experiments, we train several standard architectures on a copy of the data where the true labels were replaced by random labels. Our central finding can be summarized as:

Deep neural networks easily fit random labels.

More precisely, when trained on a completely random labeling of the true data, neural networks achieve 0 training error. The test error, of course, is no better than random chance as there is no correlation between the training labels and the test labels. In other words, by randomizing labels alone we can force the generalization error of a model to jump up considerably without changing the model, its size, hyperparameters, or the optimizer. We establish this fact for several different standard architectures trained on the CIFAR10 and ImageNet classification benchmarks. While simple to state, this observation has profound implications from a statistical learning perspective:

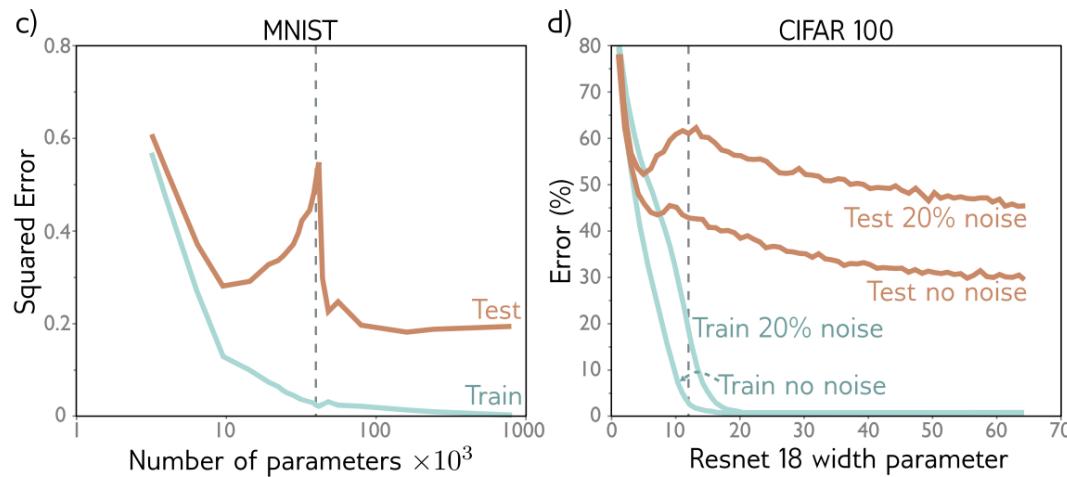
1. The effective capacity of neural networks is sufficient for memorizing the entire data set.
2. Even optimization on random labels remains easy. In fact, training time increases only by a small constant factor compared with training on the true labels.
3. Randomizing labels is solely a data transformation, leaving all other properties of the learning problem unchanged.



(a) learning curves

In Deep Learning the problem is that we can fit everything we want

Bias-variance tradeoff – the sequel



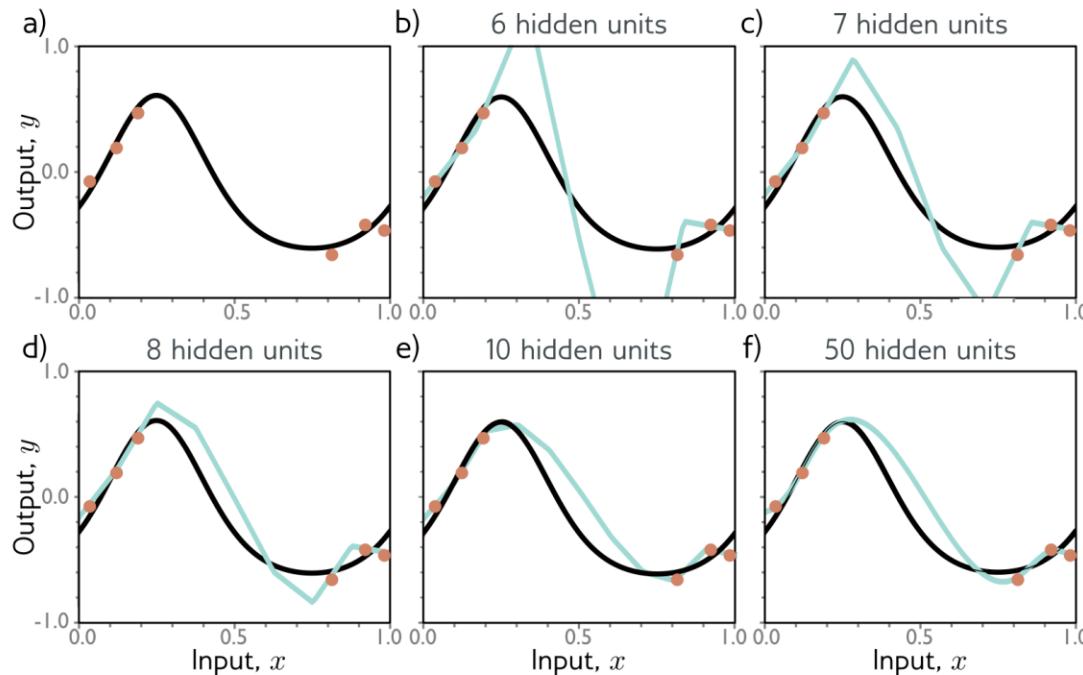
A counter-intuitive finding: when model size > dataset size, error goes down again.

This is double-descent.

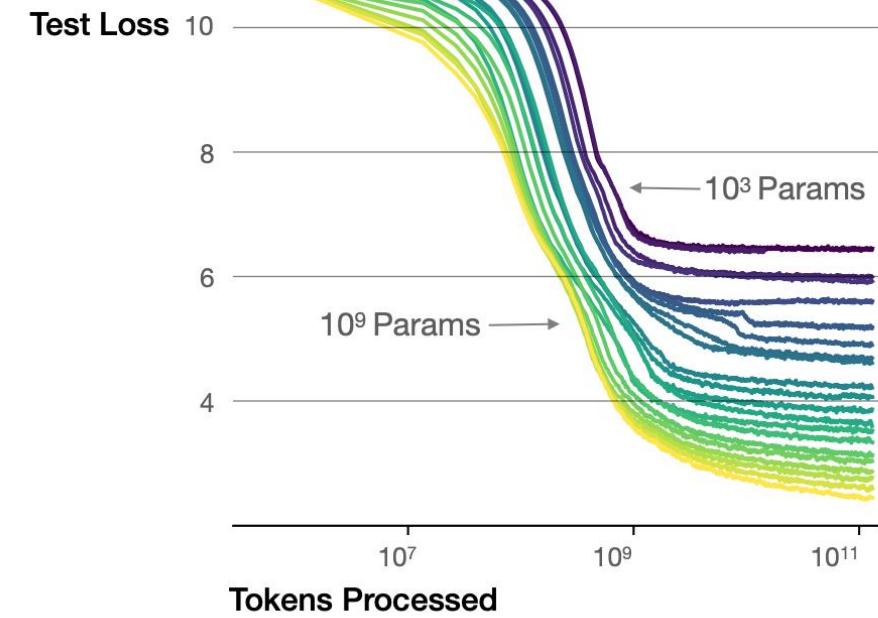
In short: one form of regularization can come from simply using bigger models, in direct conflict with the bias-variance tradeoff.

Double descent: smoothness from bigger models

Using much more complex model at a certain points (after a period of worsening) actually seems to improve the performances



Larger models require **fewer samples** to reach the same performance



Double descent: no golden ticket

In practice: simply increasing parameter count does not magically solve all problems.

Double descent only happens under some conditions, not universal.

Solution: fall back on regularization (ubiquitous in all of deep learning, including LLMs).

ℓ_2 -regularization

The ℓ_2 regularization is the most common type of all regularization techniques
Commonly known as *weight decay* or *ridge regression* (in the linear case).

The regularization term Ω is defined as the Euclidean Norm of the weight matrices.
I.e., simply the sum over all squared weight values of a weight matrix.

$$\frac{1}{2} \sum_l \|w_l\|_2^2$$

In deep learning we call it **weight decay**

ℓ_2 regularization encourages the weight values towards zero.

ℓ_2 -regularization

The loss function with ℓ_2 -regularization:

$$w^* \leftarrow \arg \min_w \sum_{(x,y) \subseteq (X,Y)} \mathcal{L}(y, a_L(x; w_1, \dots, w_L)) + \frac{\lambda}{2} \sum_l \|w_l\|_2^2$$

The ℓ_2 -regularization is added to the gradient descent update rule

$$\begin{aligned} w_{t+1} &= w_t - \eta_t (\nabla_{\theta} \mathcal{L} + \lambda w_l) \Rightarrow \\ w_{t+1} &= (1 - \lambda \eta_t) w^{(t)} - \eta_t \nabla_{\theta} \mathcal{L} \end{aligned}$$

λ is usually about $10^{-1}, 10^{-2}$



“Weight decay”, because weights
get smaller

Why would a force to zero values help prevent overfitting?

ℓ_1 -regularization

ℓ_1 -regularization is one of the most important regularization techniques

$$w^* \leftarrow \arg \min_w \sum_{(x,y) \subseteq (X,Y)} \mathcal{L}(y, a_L(x; w_1, \dots, w_L)) + \frac{\lambda}{2} \sum_l |w_l|$$

Also ℓ_1 -regularization is added to the gradient descent update rule

$$w_{t+1} = w_t - \eta_t \left(\nabla_{\theta} \mathcal{L} + \lambda \frac{w^{(t)}}{\text{sgn}(w^{(t)})} \right)$$

ℓ_1 -regularization \rightarrow sparse weights

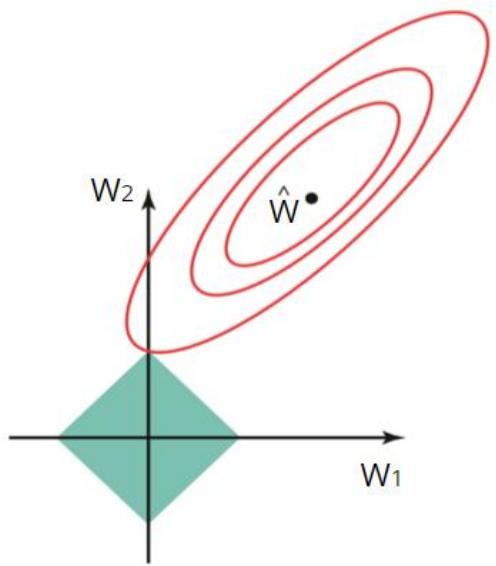
- $\lambda \nearrow \rightarrow$ more weights become 0

Sign function

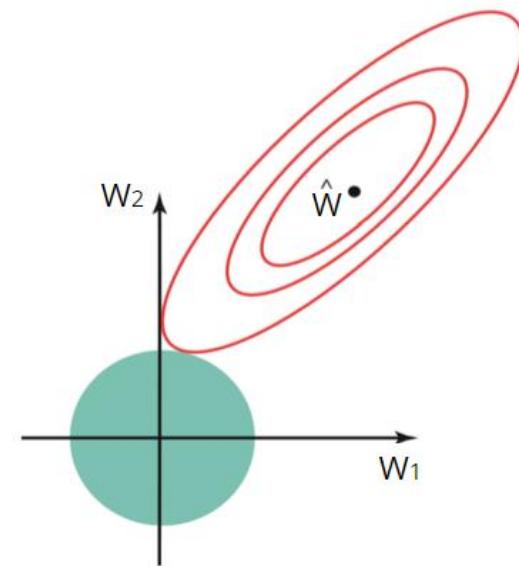
ℓ_1 regularization is sparser than ℓ_2 regularization, but why?

Due to the geometry of the constraint space

Visualizing weight decay



ℓ_1 -regularization

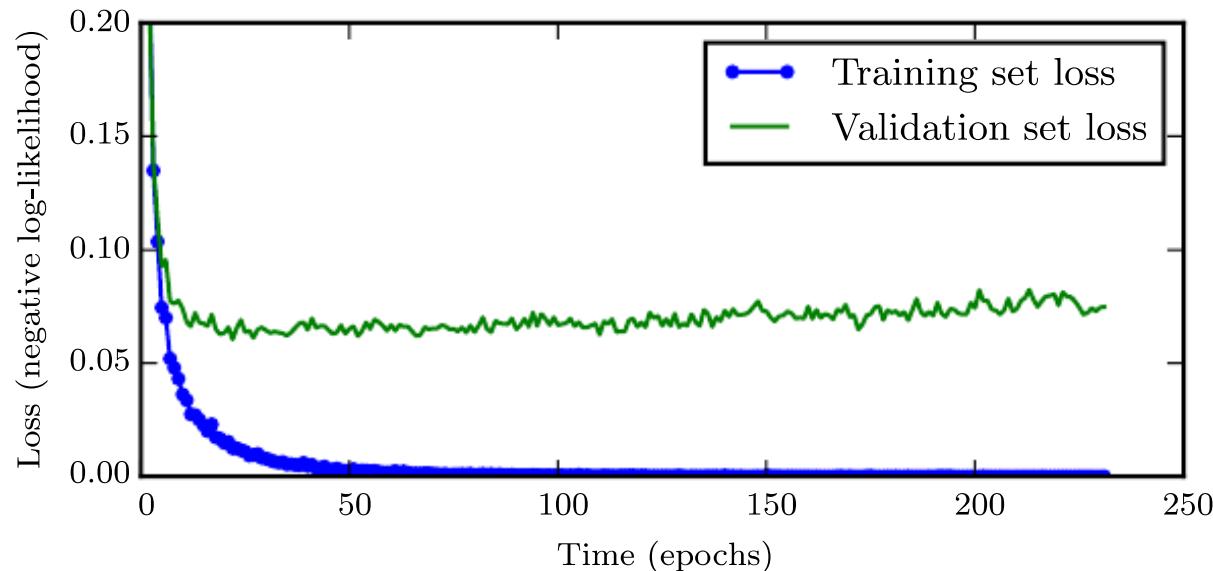


ℓ_2 -regularization

Early stopping

Even simpler solution: just stop training.

Test losses tend to increase gradually, avoid by checking with validation set.

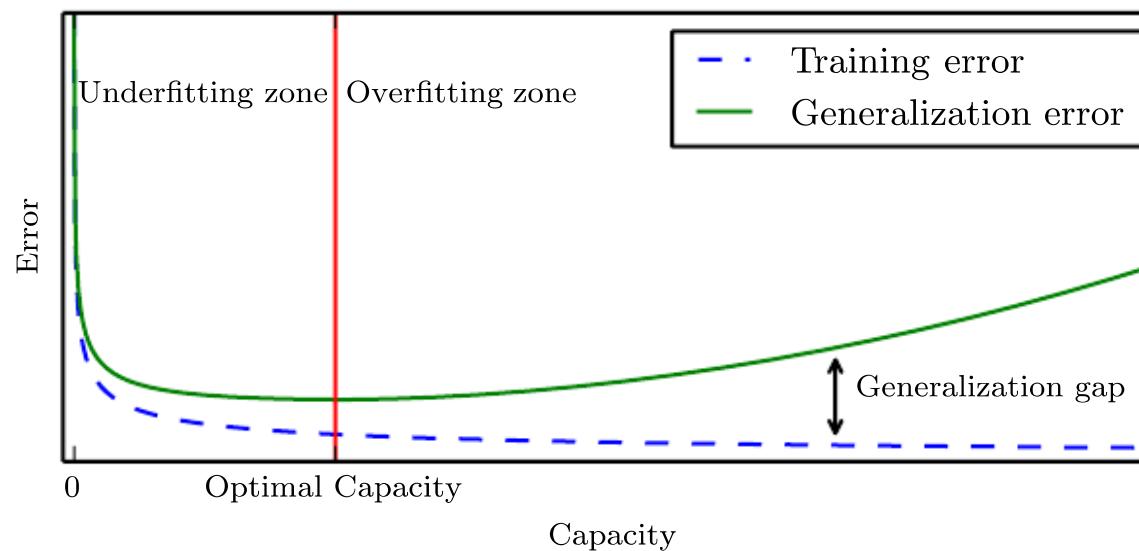


Early stopping as a hyperparameter

All we need to do is find the right epoch to stop training.

Underlying idea: bias-variance progression gradually occurs as a function of #epochs.

Naturally not possible to monitor using the training set, only with independent sets.



DropOut

The co-adaptation observation

When different hidden units in a neural networks have highly correlated behaviour.

Some of the connections will have more predictive capability than the others.

These powerful connections are learned more while the weaker ones are ignored.

Over many iterations, only a fraction of the node connections is trained.

The rest stops participating. DropOut seeks to prevent this.

Co-adaptation means that different hidden units (neurons) start relying on each other so heavily that they behave very similarly, rather than learning individually useful features.



As these stronger connections dominate, the weaker ones become less useful, stop learning, and get ignored during training. Eventually, only a small subset of the connections is actively contributing, limiting the network's ability to generalize.

Implementation of DropOut

Dropout tries to solve the co adaptation by randomly removing (or "dropping out") some units during each training step so that all units must learn to work independently and cannot rely too much on any single set of connections. This prevents any particular node from becoming too influential, reduces correlation among units, and encourages each neuron to contribute meaningfully to the prediction. As a result, the whole network becomes more robust and better at generalizing to new inputs.

During training randomly set activations to 0

- Neurons sampled at random from a Bernoulli distribution with p (eg, $p = 0.5$)

- Neuron activations reweighted by $1/p$

During testing all neurons are used

Benefits

- Reduces complex co-adaptations between neurons

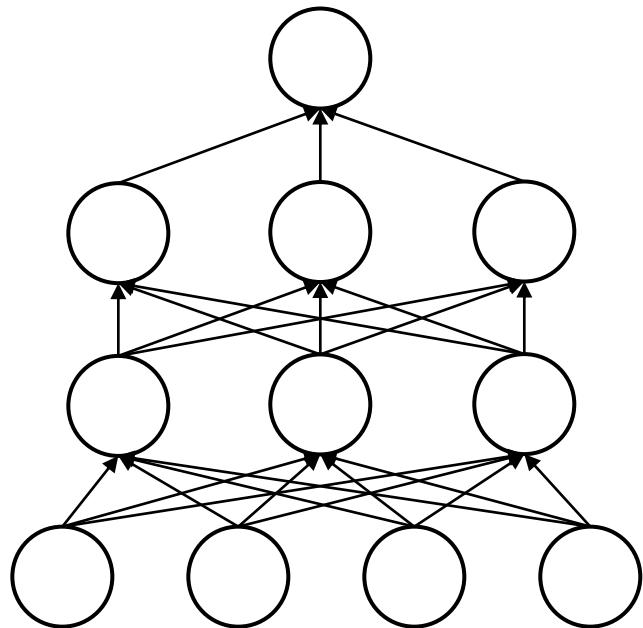
- Every neuron becomes more robust

- Decreases overfitting

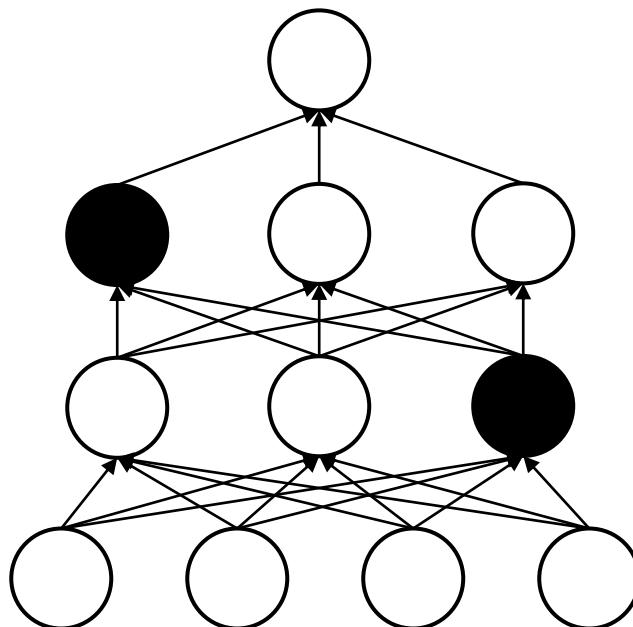


DropOut: effectively 2^n architectures

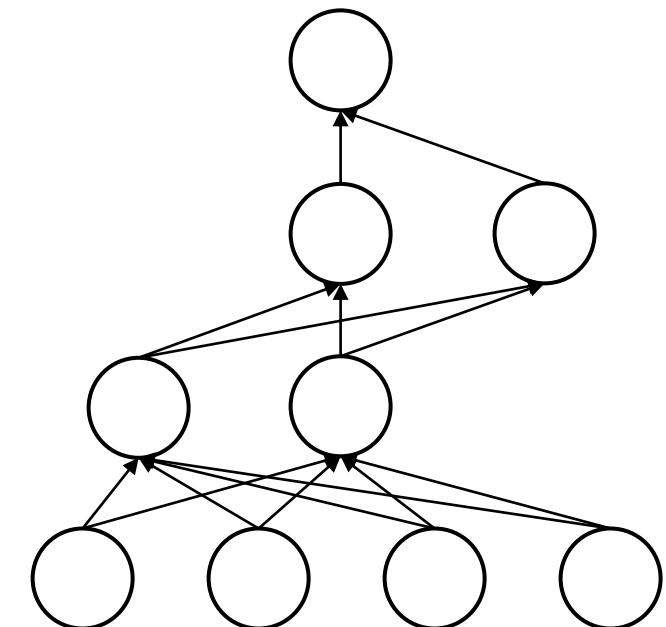
Original model



Batch 1

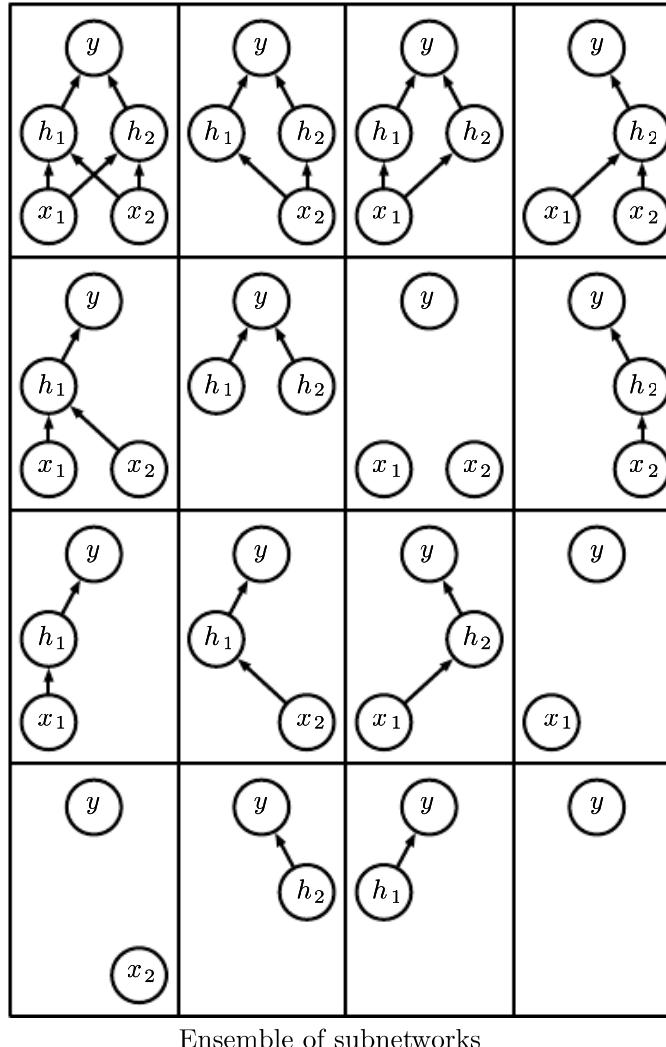
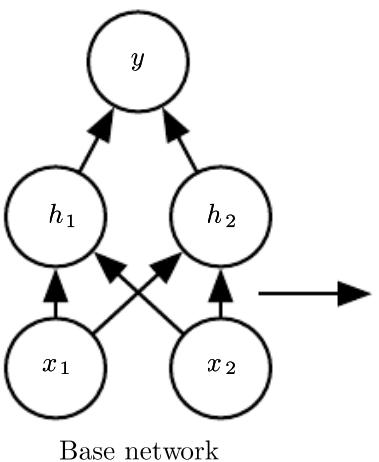


Batch 1



DropOut: effectively 2^n architectures

With dropout, each subnetwork does not have its own unique weights. Instead, every subnetwork is formed by temporarily masking (zeroing out) a different random subset of units and their outgoing connections in the same weight matrix during training. Thus, all subnetworks share the exact same weights—they just use different subsets on each forward and backward pass.



During training with dropout, the network is constantly forced to learn with different random subsets of its neurons active. This means that the network can't rely on specific neurons or very particular combinations of parameters; instead, it must learn redundant and resilient patterns that work no matter which subset is active. So, the dropout network is compelled to distribute the ability to detect features across many different paths in the network, rather than letting specialization or co-adaptation form—where certain neurons only work if others are present.

DropOut: effectively 2^n architectures

Ensembling is a well-known way to improve/regularize machine learning models.

In DropOut, each combination of selected neurons forms its own submodel.

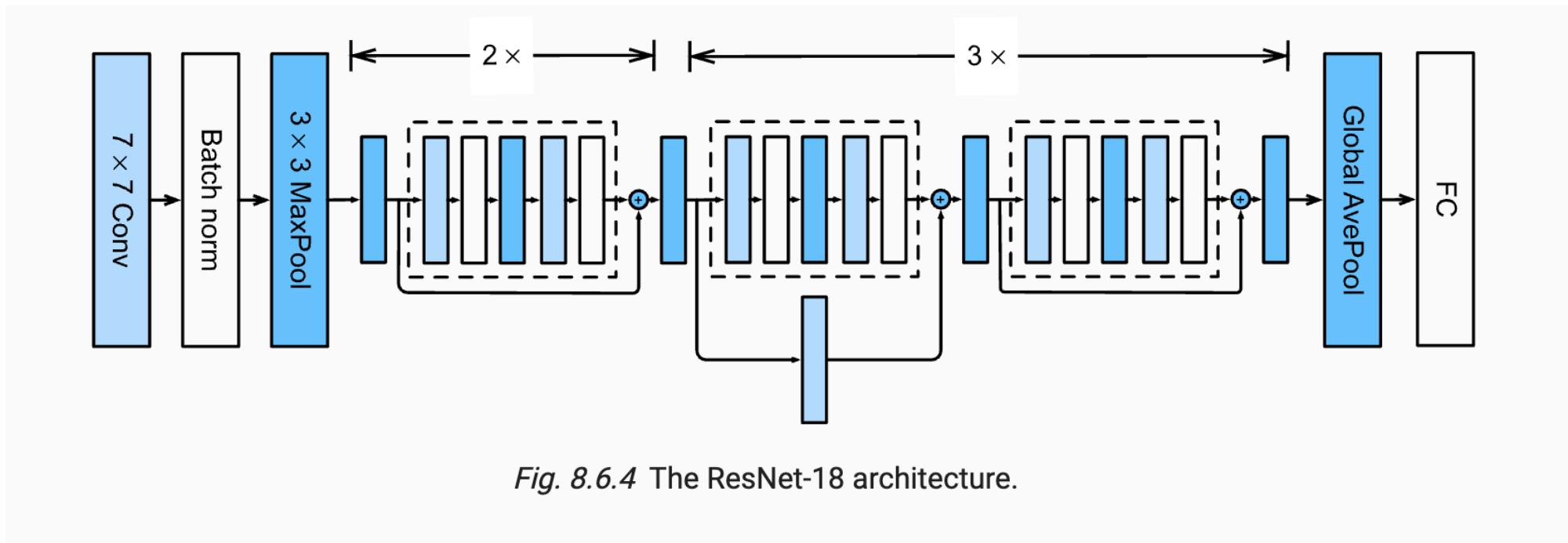
The parameters between the submodels share weights.

During testing, no need to run data on multiple models (unlike assembling).

Setting neurons to zero also breaks co-adaption/co-occurrences, i.e.:

Dropout regularizes each hidden unit to be not merely a good feature but a feature that is good in many contexts.

Network initialization



Weight initialization

To prevent layer activation outputs from exploding or vanishing gradients.

Initialize weights correctly and our objective will be achieved in the least time.

Zero Initialization

- Leads to symmetric hidden layers.
- Makes your network no better than a linear model.
- Setting biases to 0 will not create any problems.

We set it to 0 and not to any other values because it would induce a bias in the data, if a bias is needed the model will figure it out and learn it

Random Initialization

- Breaks symmetry.
- Prevents neurons from learning the same features.

Random how?

Weights initialized **to preserve the variance** of the activations

- During the forward and backward computations.
- We want similar input and output variance because of modularity.

Weights must be initialized to be different from one another

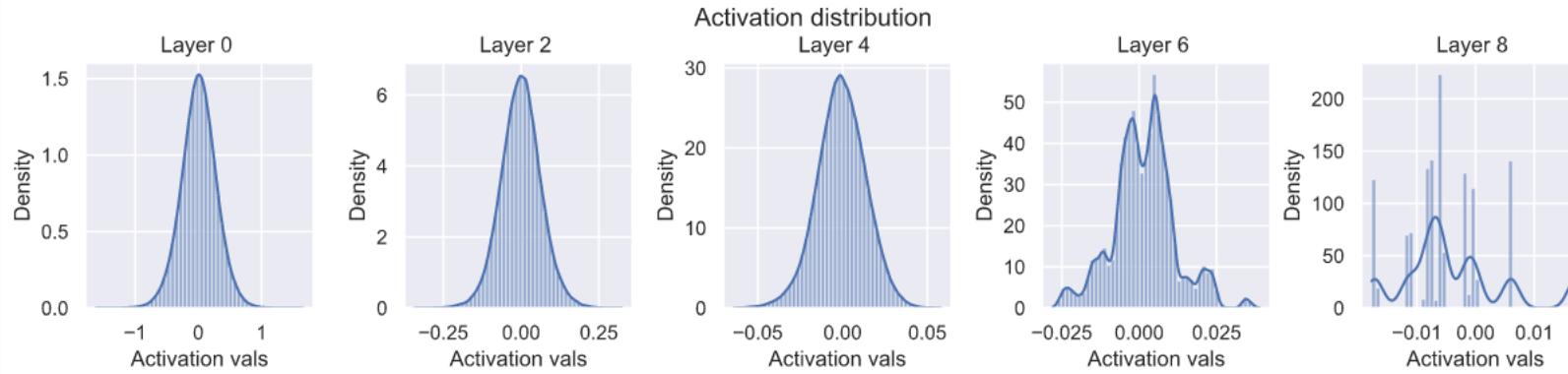
- Don't give same values to all **weights** (like all 0).
- All neurons (in one layer) generate same gradient → no learning.

Initialization depends on:

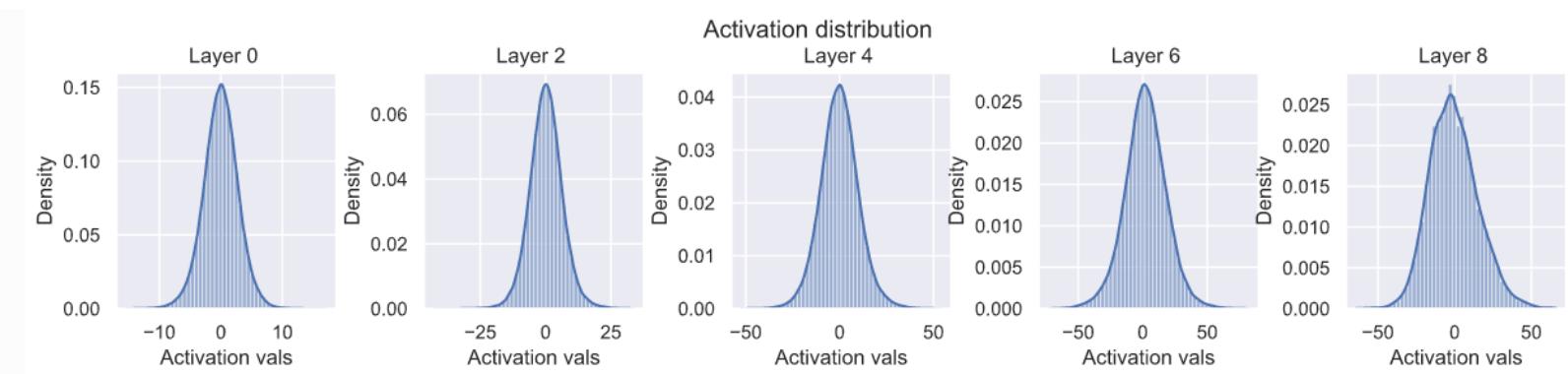
- non-linearities.
- data normalization.

Bad initialization will come back to haunt you

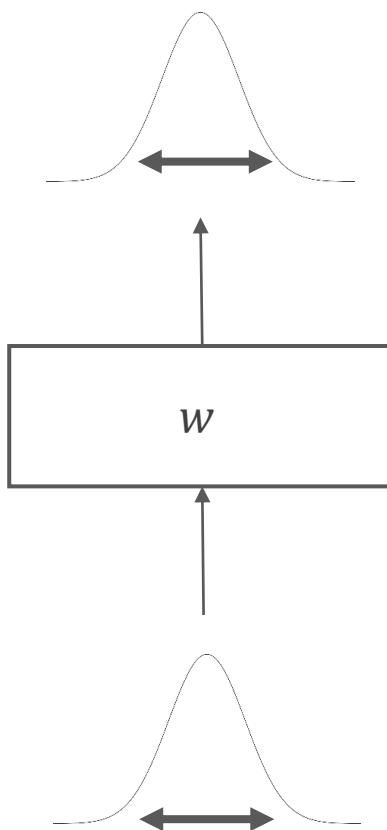
Initializing weights in every layer with same constant variance → can diminish variance in activations



Initializing weights in every layer with increasing variance → can explode the variance in activations



Preserving variance



For x and y independent

$$\text{var}(xy) = \mathbb{E}[x]^2 \text{var}(y) + \mathbb{E}[y]^2 \text{var}(x) + \text{var}(x)\text{var}(y)$$

For $a = wx \Rightarrow \text{var}(a) = \text{var}(\sum_i w_i x_i) = \sum_i \text{var}(w_i x_i) \approx d \cdot \text{var}(w_i x_i)$

$$\begin{aligned}\text{var}(w_i x_i) &= \mathbb{E}[x_i]^2 \text{var}(w_i) + \mathbb{E}[w_i]^2 \text{var}(x_i) + \text{var}(x_i)\text{var}(w_i) \\ &= \text{var}(x_i)\text{var}(w_i)\end{aligned}$$

Because we assume that x_i, w_i are unit Gaussians $\rightarrow \mathbb{E}[x_i] = \mathbb{E}[w_i] = 0$

So, the variance in our activation $\text{var}(a) \approx d \cdot \text{var}(x_i)\text{var}(w_i)$

Preserving variance

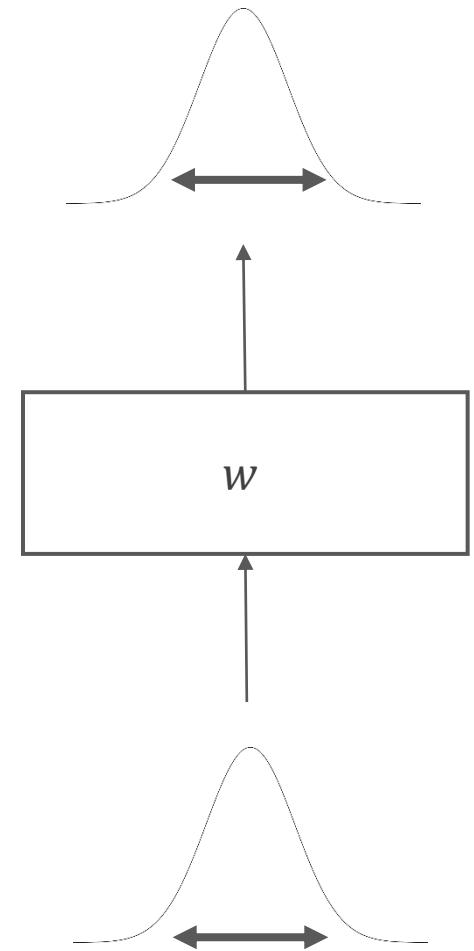
Since we want the same input and output variance

$$\text{var}(a) = d \cdot \text{var}(x_i) \text{var}(w_i) \Rightarrow \text{var}(w_i) = \frac{1}{d}$$

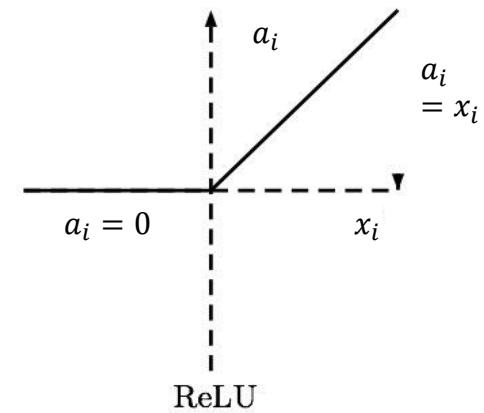
Draw random weights from

$$w \sim N(0, 1/d)$$

where d is the number of input variables to the layer



Kaiming Initialization



ReLU's return 0 half of the time: $\mathbb{E}[w_i] = 0$ but $\mathbb{E}[x_i] \neq 0$

$$\begin{aligned} var(w_i x_i) &= var(w_i)(\mathbb{E}[x_i]^2 + var(x_i)) \\ &= var(w_i)\mathbb{E}[x_i^2] \quad (var(X) = \mathbb{E}[X^2] - \mathbb{E}[X]^2) \\ \mathbb{E}[x_i^2] &= \int_{-\infty}^{\infty} x_i^2 p(x_i) dx_i = \int_{-\infty}^{\infty} \max(0, a_i)^2 p(a_i) da_i = \int_0^{\infty} a_i^2 p(a_i) da_i \\ &= 0.5 \int_{-\infty}^{\infty} a_i^2 p(a_i) da_i = 0.5 \cdot \mathbb{E}[a_i^2] = 0.5 \cdot var(a_i) \end{aligned}$$

the data
after relu
are no
longer 0
means

exam question why

Draw random weights from $w \sim N(0, 2/d)$ – Kaiming Initialization

Xavier initialization

For tanh: initialize weights from $U\left[-\sqrt{\frac{6}{d_{l-1}+d_l}}, \sqrt{\frac{6}{d_{l-1}+d_l}}\right]$

d_{l-1} is the number of input variables to the tanh layer and d_l is the number of the output variables

For a sigmoid $U\left[-4 \cdot \sqrt{\frac{6}{d_{l-1}+d_l}}, 4 \cdot \sqrt{\frac{6}{d_{l-1}+d_l}}\right]$

Random networks are already great deep learners

What's Hidden in a Randomly Weighted Neural Network? Ramanujan et al. 2019

*Hidden in a **randomly weighted Wide ResNet-50** we find a subnetwork (with random weights) **that is smaller than, but matches the performance of a ResNet-34 trained on ImageNet [4]**. Not only do these “untrained subnetworks” exist, but we provide an algorithm to effectively find them.*

there is a lot of inductive bias in our networks, so a lot of the good comes from the architecture that we use on the specific type of data

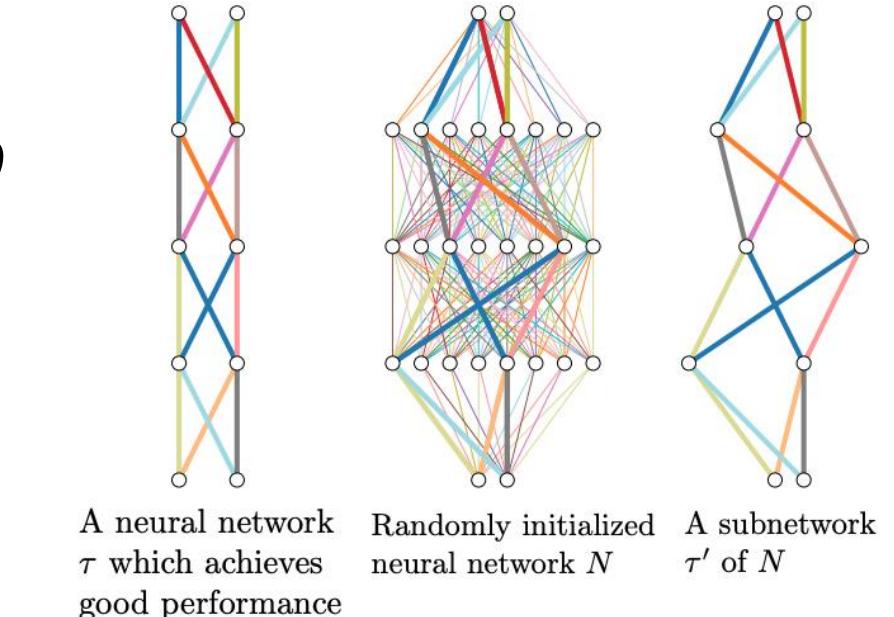


Figure 1. If a neural network with random weights (center) is sufficiently overparameterized, it will contain a subnetwork (right) that perform as well as a trained neural network (left) with the same number of parameters.

Break

Data augmentation

The best way to make a machine learning model generalize better is to train it on more data. (see: "The unreasonable effectiveness of data")

- Data* is limited in practice
- One way is to create fake data – *Data Augmentation***

we are gonna deal with invariances,
invariances are changes in the inputs
which does not make any change in
the outputs

Your neural network is only as good as the data you feed it.

By performing augmentation, we can prevent neural networks from learning or memorizing irrelevant patterns, essentially boosting overall performance.

* Labeled data

** Not that trivial. Augmentations are more than just fake data!

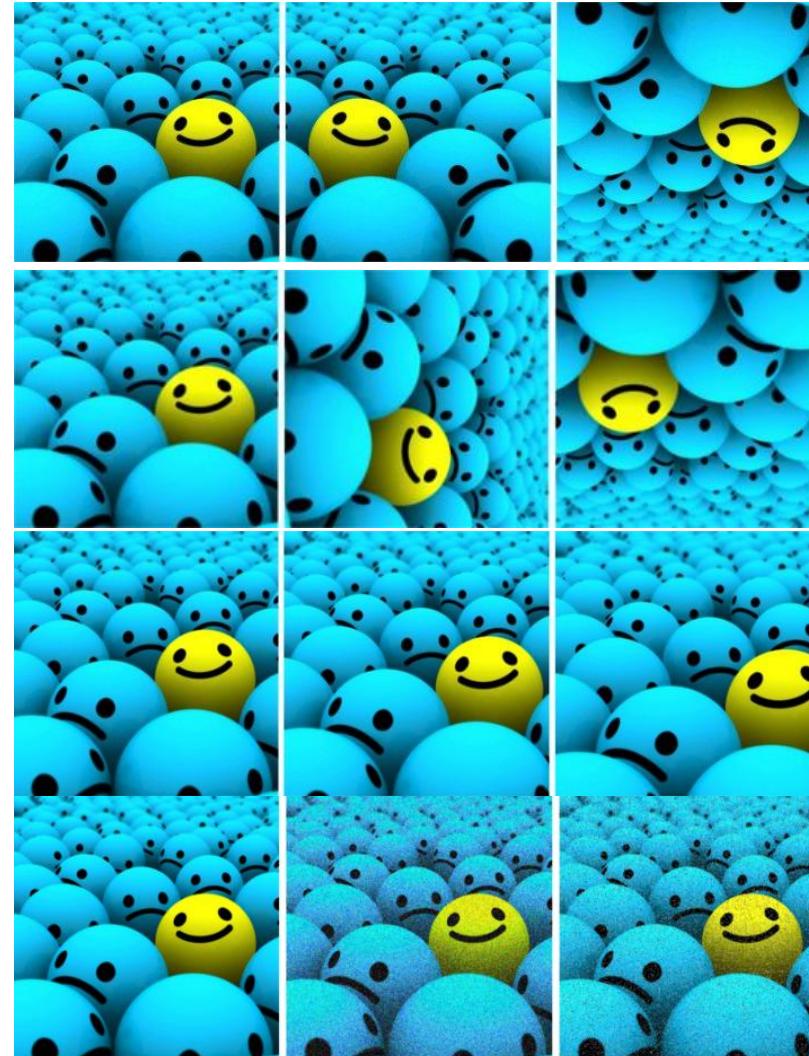
Examples of data augmentation

In computer vision

- Flip
- Rotation
- Scale
- Crop
- Translation
- Gaussian noise

Be aware of label change

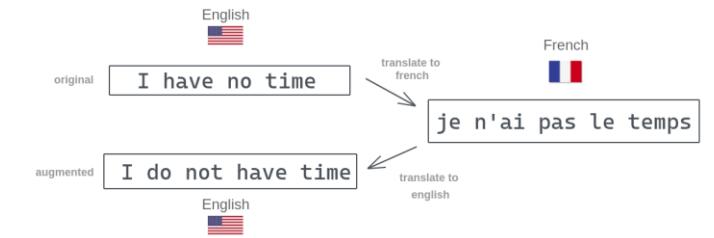
- “b” and “d”
- “6” and “9”



[Link](#)

In NLP

Backtranslation



Synonym replacement

Random insertion

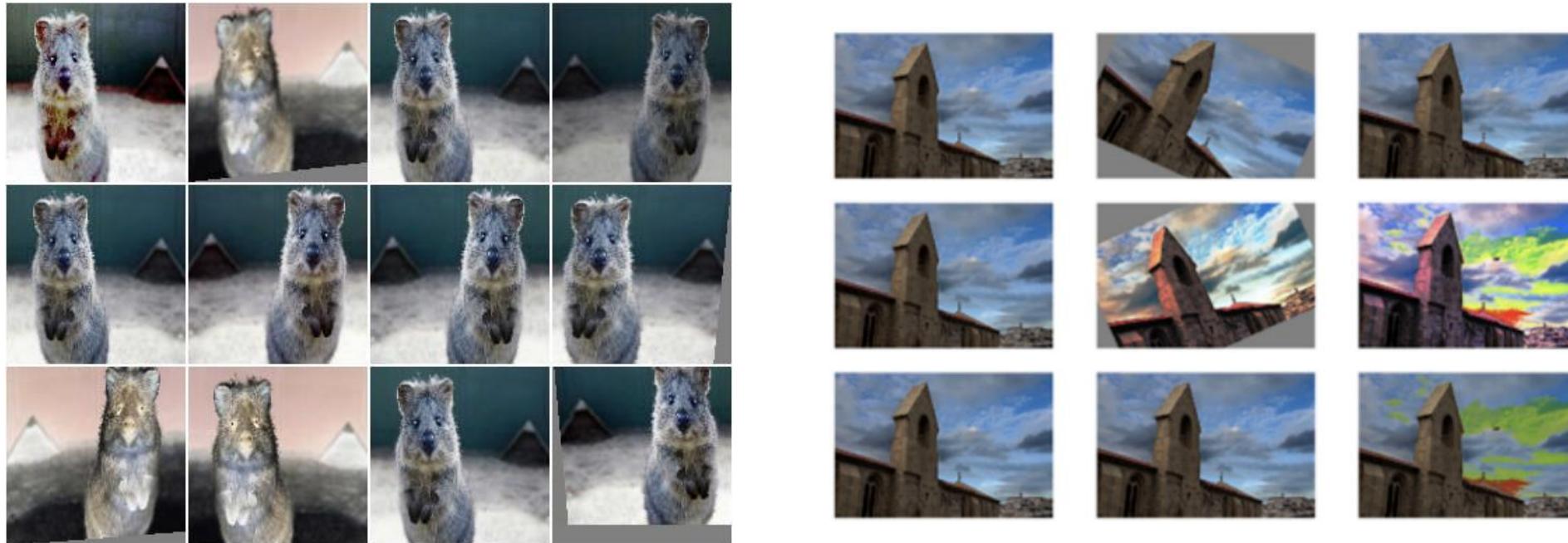
Random deletion

Random swapping

Data augmentation = pre-defined invariance

Essentially a form of injecting prior knowledge to instill **invariance**.

A dog flipped vertically is still a dog, so a network should still predict that label



Other data augmentations

Noise robustness

Adding noise to weights – uncertainty.

Adding noise to outputs - label smoothing.

Semi or self-supervised learning

Introducing a particular form of prior belief about the solution.

Multi-task learning

Shared input and parameters – improve statistical strength.

Requires statistical relationship between tasks.

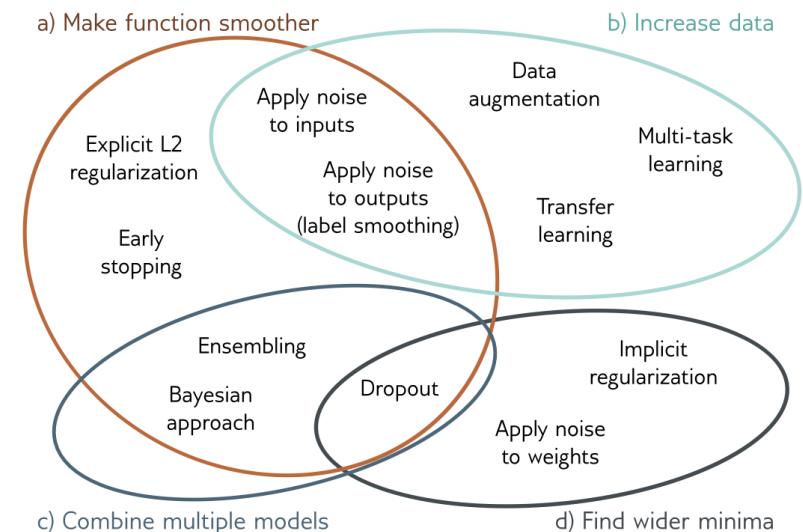
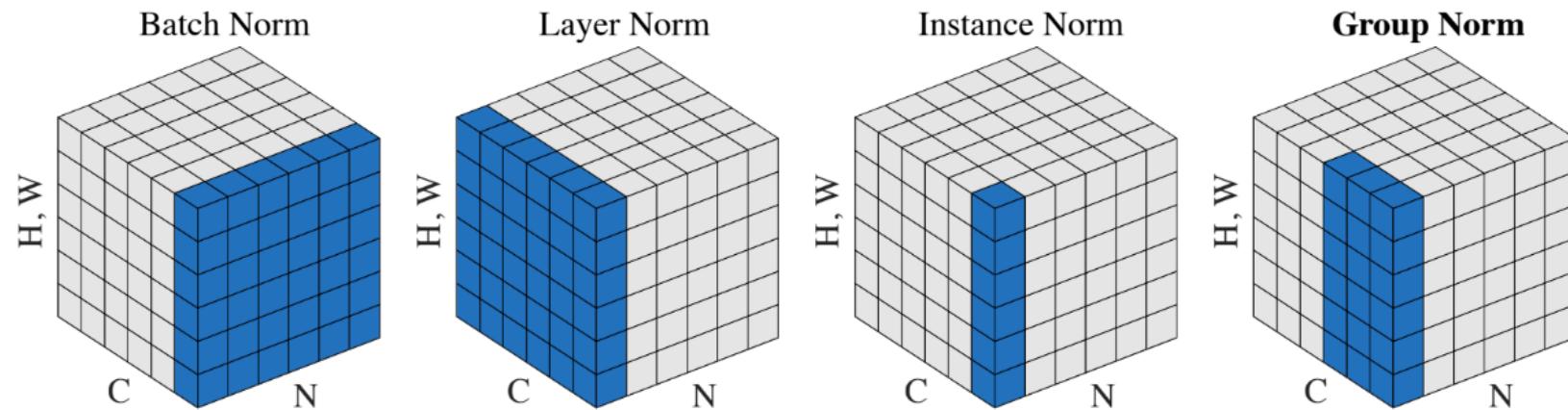


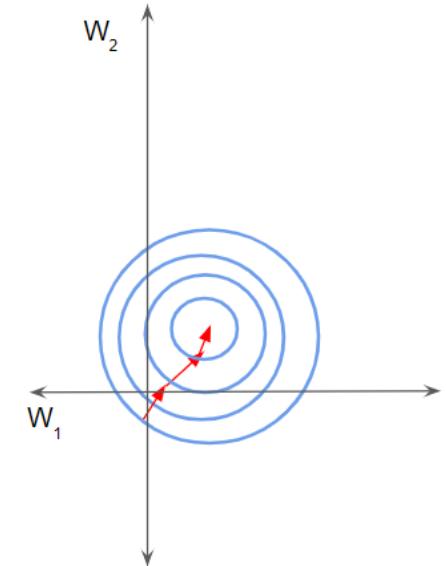
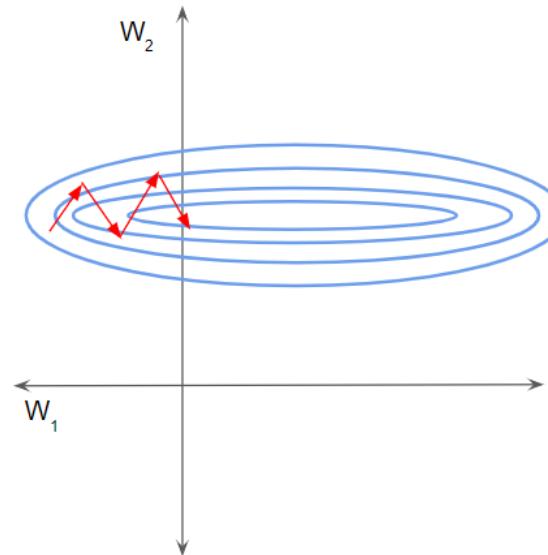
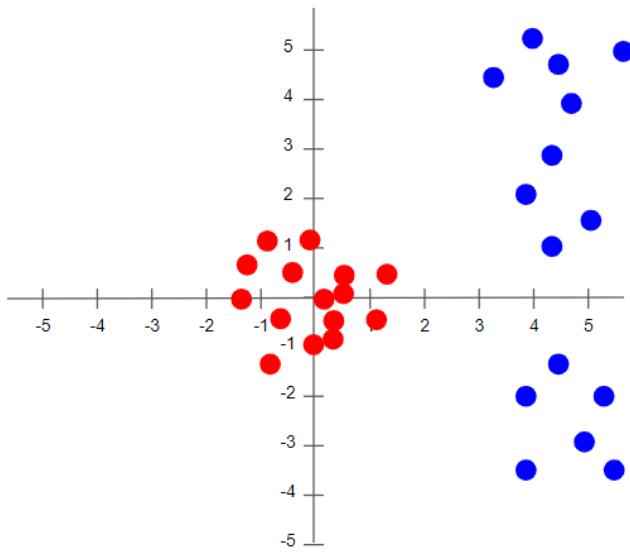
Figure 9.14 Regularization methods. The regularization methods discussed in this chapter aim to improve generalization by one of four mechanisms. a) Some methods aim to make the modeled function smoother. b) Other methods increase the effective amount of data. c) The third group of methods combine multiple models and hence mitigate against uncertainty in the fitting process. d) Finally, the fourth group of methods encourages the training process to converge to a wide minimum where small errors in the estimated parameters are less important.

Normalization



Normalization as data preprocessing

Data pre-processing brings numerical data to a common scale without distorting shape.
The reason is partly to ensure that our model can generalize appropriately.
This ensures that all the feature values are now on the same scale.



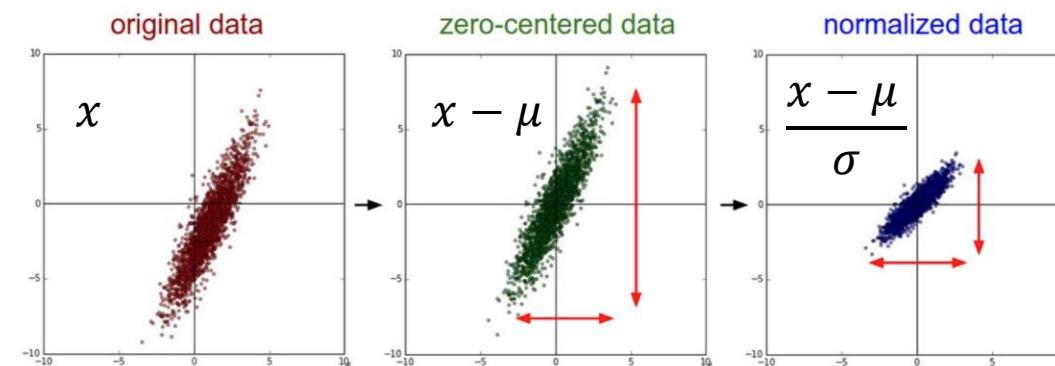
Must in deep learning: normalizing input data

Transforming the input to zero-mean, unit variance

Assume: Input variables follow a Gaussian distribution (roughly)

Subtract input by the mean

Optionally, divide by the standard deviation $N(\mu, \sigma^2) \rightarrow N(0, 1)$

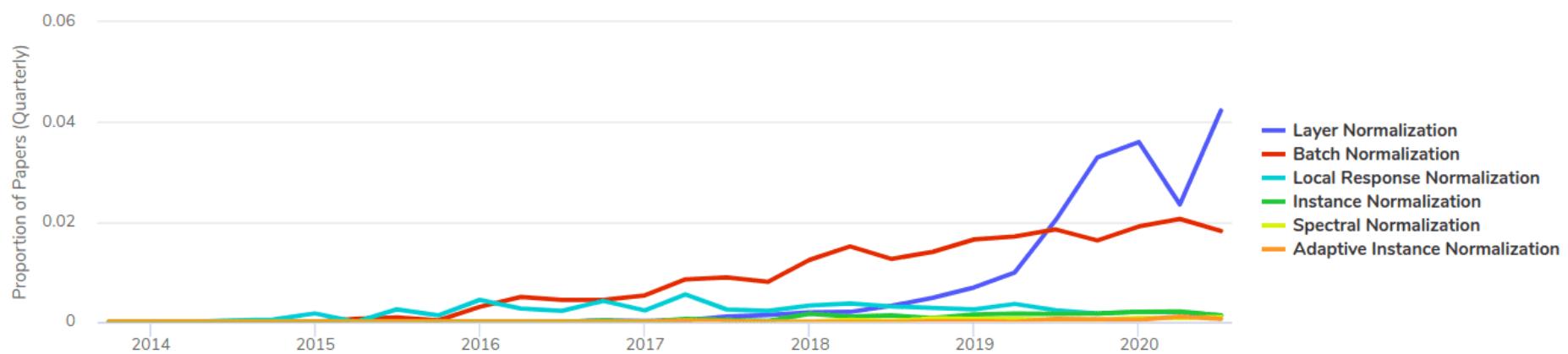
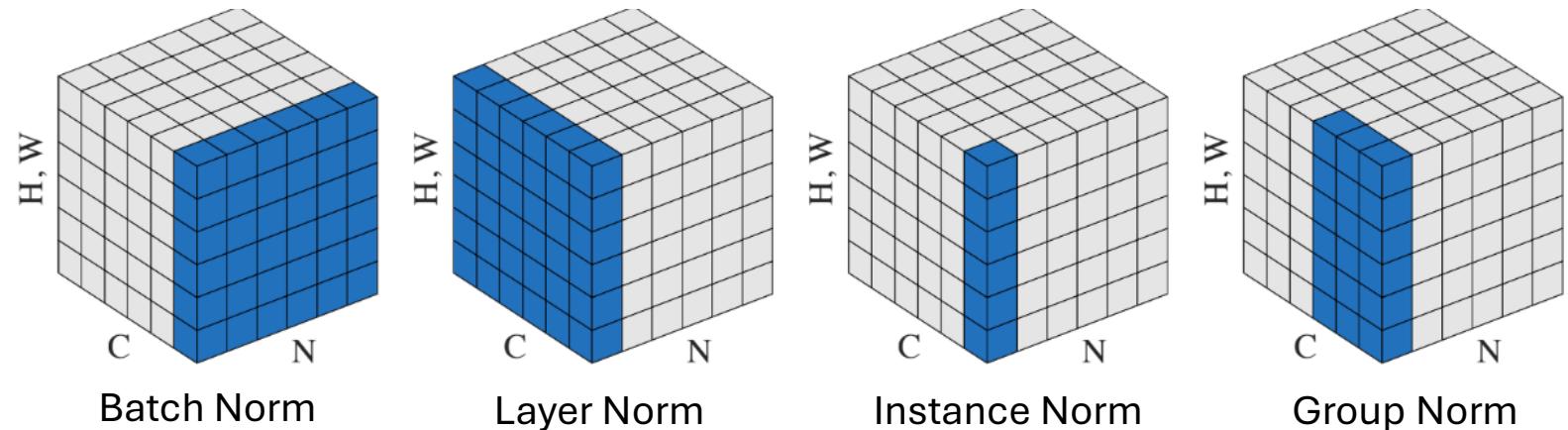


Known by all practitioners:

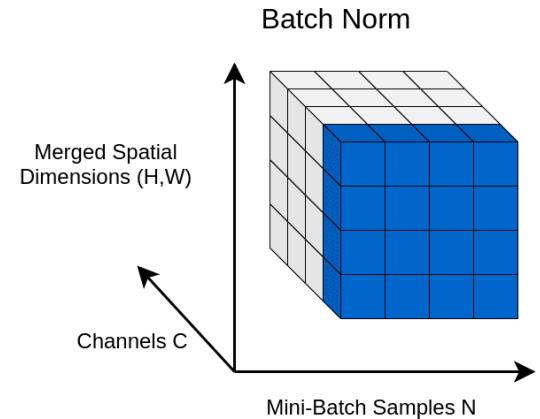
ImageNet: mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224, 0.225]

Normalizing intermediate layers

Batch normalization
Layer normalization
Instance normalization
Group normalization
Weight normalization



Batch normalization



Normalize the layer inputs with batch normalization

Normalize $a_l \sim N(0, 1)$

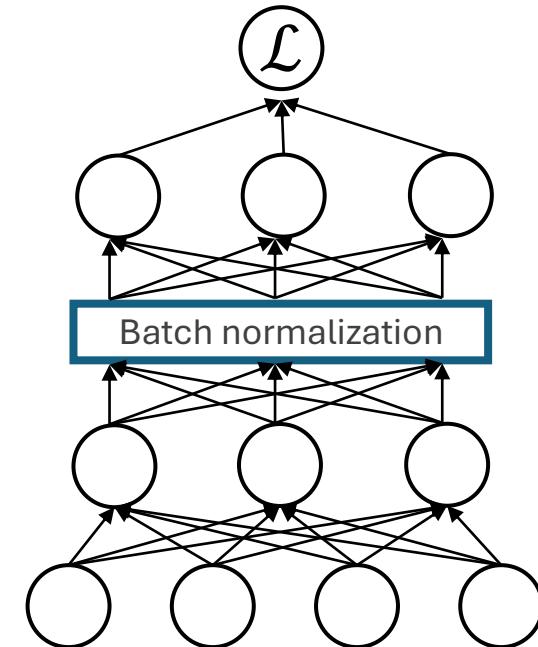
Followed by affine transformation $a_l \leftarrow \gamma a_l + \beta$

The parameters γ and β are trainable

Used for re-scaling (γ) and shifting (β) of the vector values.

Ensure the optimal values of γ and β are used.

Enable the accurate normalization of each batch.



Batchnorm algorithm

$$\mu_j \leftarrow \frac{1}{m} \sum_{i=1}^m x_{ij}$$

[compute mini-batch mean]

$$\sigma_j^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_{ij} - \mu_j)^2$$

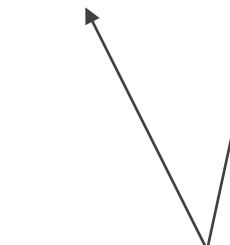
[compute mini-batch variance]

$$\hat{x}_{ij} \leftarrow \frac{x_{ij} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

[normalize input]

$$\hat{x}_{ij} \leftarrow \gamma \hat{x}_{ij} + \beta$$

[scale and shift input]



Trainable parameters

we can't normalize the layer because we would destroy what we have done.

this change the zero mean unit variance gaussian to another gaussian thus improving performance

the choice of batch size now becomes more important than before

i runs over mini-batch samples,
 j over the feature dimensions

Batchnorm at test time

there are 2 way to do testing:

Inductive: each test sample is independently check

Transductive: very often we have a data distribution during testing, and this gave us a lot intuition.

How do we ship the Batch Norm layer after training?

We might not have batches at test time

Batches are random? -> not reproducible

why moving average and not the mean of the whole train=

Usually: keep a moving average of the mean and variance during training

Plug them in at test time

In the limit, the moving average of mini-batch statistics approaches the batch statistics

$$\mu_B \leftarrow \frac{1}{m} \sum_{i=1}^m x_i$$

$$\sigma_B \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$\hat{y}_i \leftarrow \gamma \hat{x}_i + \beta$$

Benefits of batchnorm

Networks train faster

Allows higher learning rates

Makes more activation functions viable

Reduces overfitting

Simplifies the creation of deeper networks

May give better results overall

Drawbacks of batchnorm

Requires large mini-batches

Cannot work with mini-batch of size 1 ($\sigma = 0$)

Performance is sensitive to the batch size

Memory intense, all the batch statistics must be stored in the layer.

Discrepancy between training and test data

Breaks the independence between training examples in the minibatch

Not applicable to online learning

Awkward to use with recurrent neural networks

Must interleave it between recurrent layers

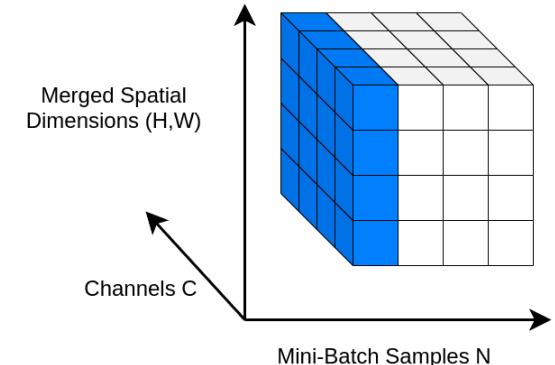
Also, store statistics per time step

Often the reason for bugs

In deep learning there are not outliers because we're gonna fit all the data anyway. A way to improve training is to randomly create outlier by changing the label (adding label noise) this make the model more robust.

We do not detect them,

Layer normalization

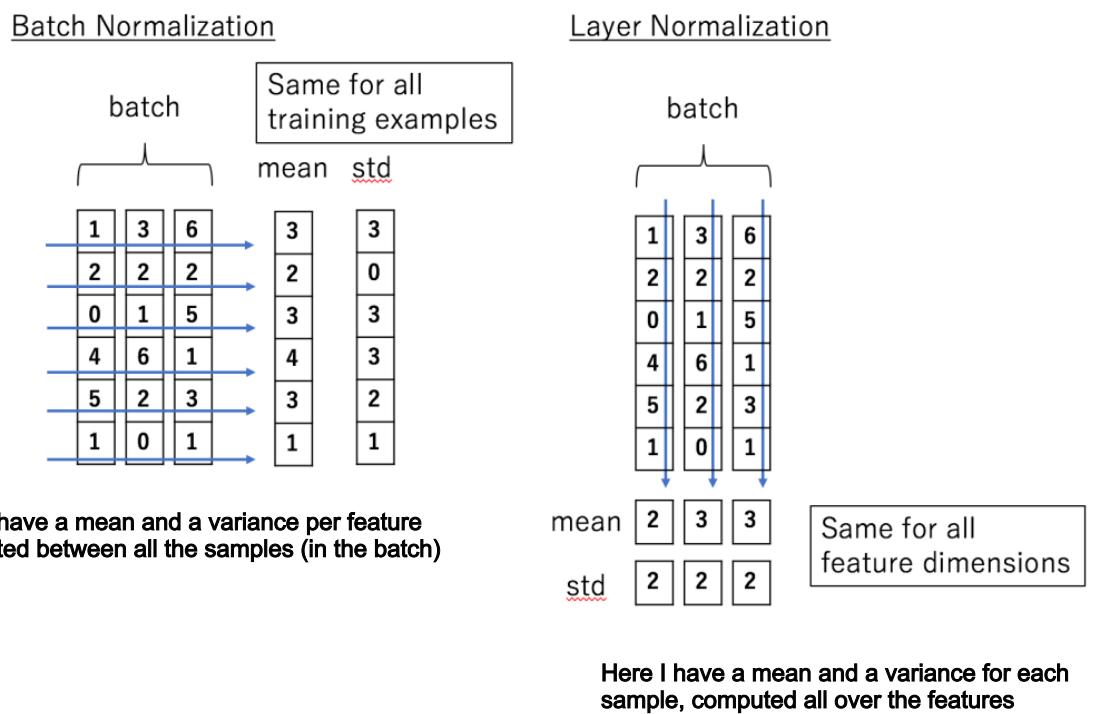


The statistics (mean and variance) are computed across all channels and spatial dimensions.

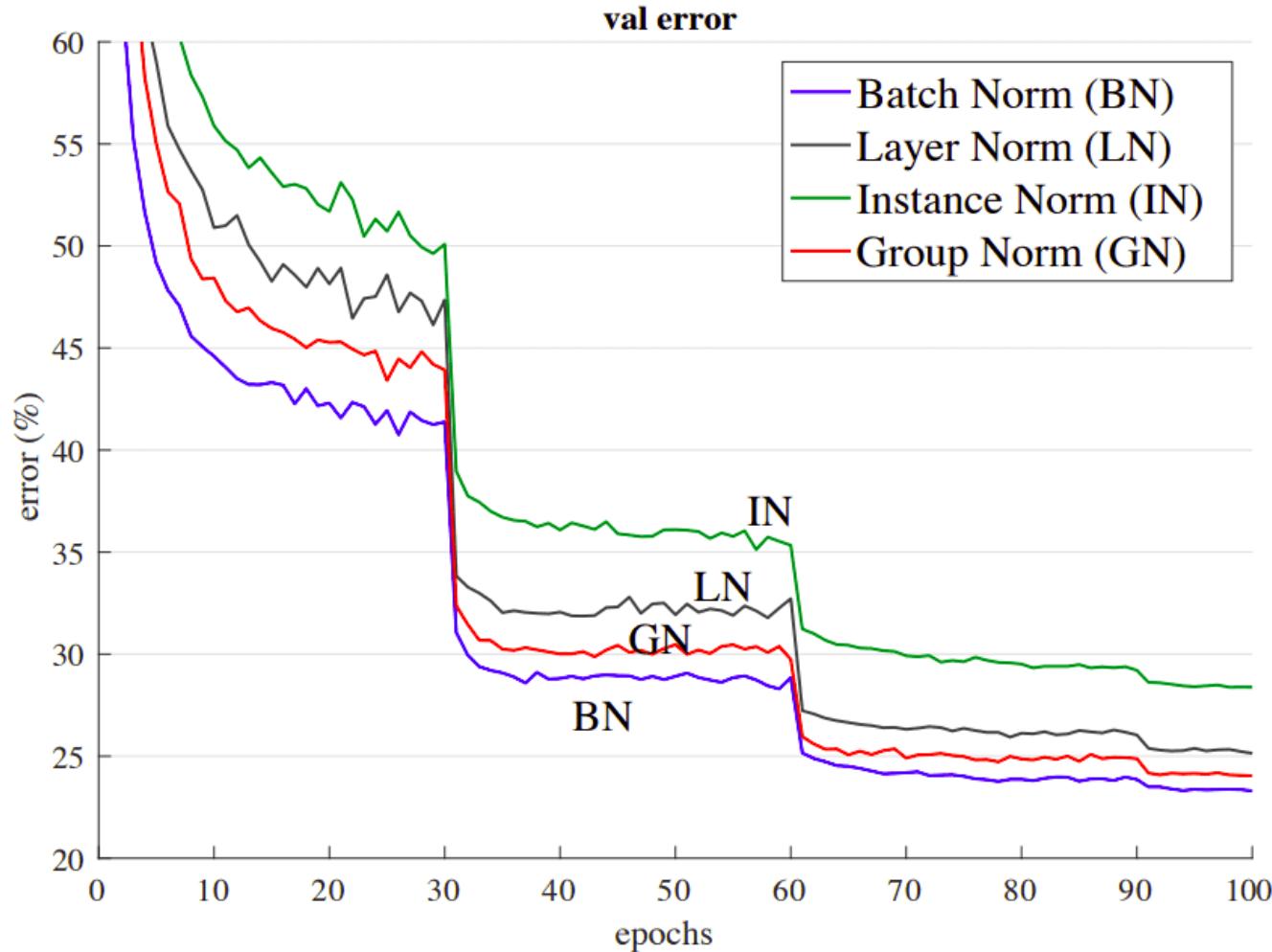
The statistics are independent of the batch.

This layer was initially introduced to handle vectors (mostly the RNN outputs).

Layer normalization performs the same computation at training and test times.



Comparing different normalizations



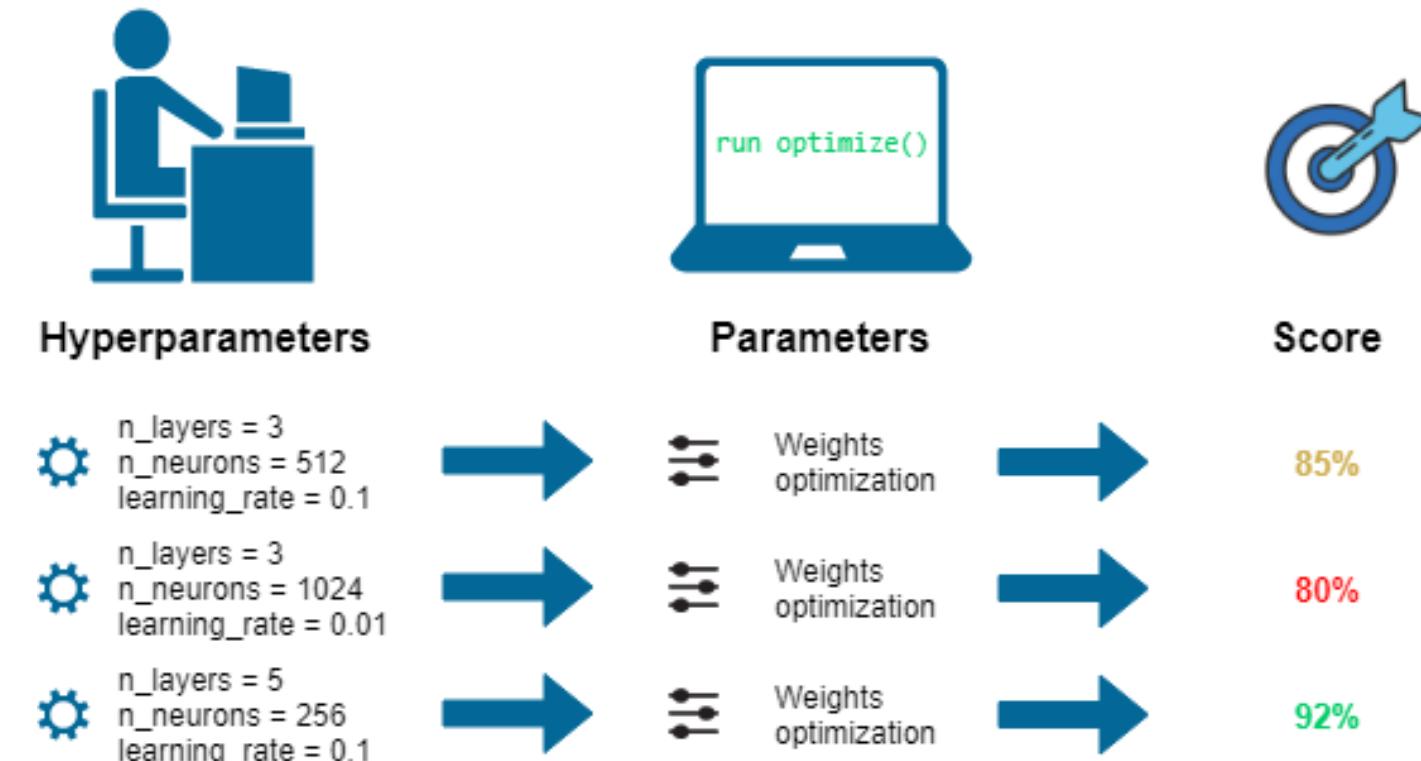
Paper list for the interested reader

- Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167.
- Salimans, T., & Kingma, D. P. (2016). Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In Advances in neural information processing systems (pp. 901-909).
- Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer normalization. arXiv preprint arXiv:1607.06450.
- Ulyanov, D., Vedaldi, A., & Lempitsky, V. (2016). Instance normalization: The missing ingredient for fast stylization. arXiv preprint arXiv:1607.08022.
- Wu, Y., & He, K. (2018). Group normalization. In Proceedings of the European conference on computer vision (ECCV) (pp. 3-19).
- Zhang, H., Dana, K., Shi, J., Zhang, Z., Wang, X., Tyagi, A., & Agrawal, A. (2018). Context encoding for semantic segmentation. In Proceedings of the IEEE conference on Computer Vision and Pattern Recognition (pp. 7151-7160).
- Santurkar, S., Tsipras, D., Ilyas, A., & Madry, A. (2018). How does batch normalization help optimization?. In Advances in Neural Information Processing Systems (pp. 2483-2493).
- Dumoulin, V., Shlens, J., & Kudlur, M. (2016). A learned representation for artistic style. arXiv preprint arXiv:1610.07629.
- Park, T., Liu, M. Y., Wang, T. C., & Zhu, J. Y. (2019). Semantic image synthesis with spatially-adaptive normalization. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 2337-2346).
- Huang, X., & Belongie, S. (2017). Arbitrary style transfer in real-time with adaptive instance normalization. In Proceedings of the IEEE International Conference on Computer Vision (pp. 1501-1510).
- Kolesnikov, A., Beyer, L., Zhai, X., Puigcerver, J., Yung, J., Gelly, S., & Houlsby, N. (2019). Big transfer (BiT): General visual representation learning. arXiv preprint arXiv:1912.11370.
- Qiao, S., Wang, H., Liu, C., Shen, W., & Yuille, A. (2019). Weight standardization. arXiv preprint arXiv:1903.10520.

Hyperparameters

All values that cannot be tuned through backprop.

Hyperparameter tuning most important part of daily deep learning life.



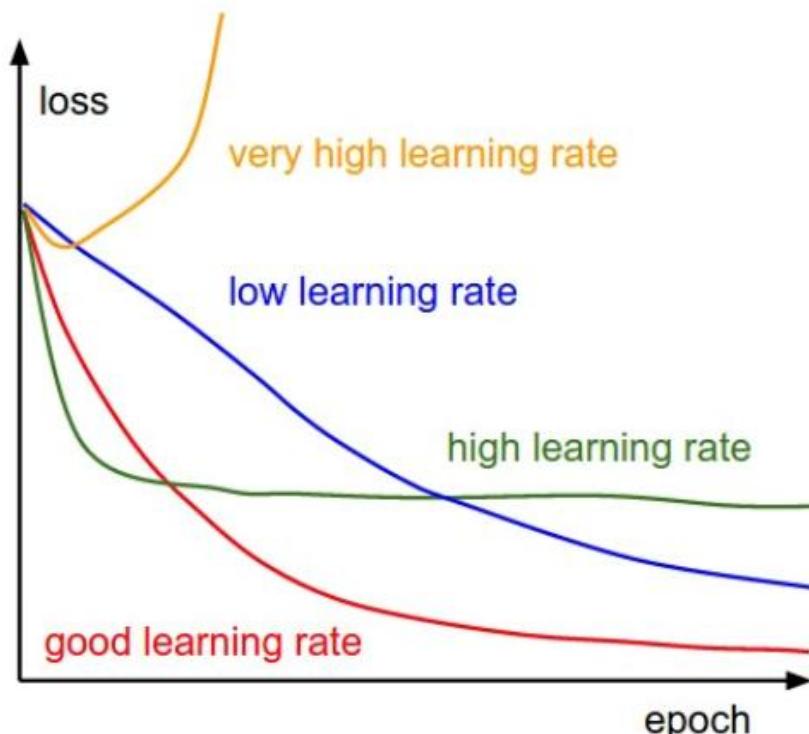
Learning rates in practice

The learning rate is probably the most influence hyperpar, start with different scale then adjust

Try several log-spaced values $10^{-1}, 10^{-2}, 10^{-3}, \dots$ on a smaller set.

Then, you narrow it down from there around where you get the lowest **validation** error.

You will learn to become and expert in learning rate pattern recognition over time.



different opinion:

One rule, doen't make it too smal if using
match norm

try to make it as big to fit GPU

REVISITING SMALL BATCH TRAINING FOR DEEP NEURAL NETWORKS

Dominic Masters and Carlo Luschi
Graphcore Research
Bristol, UK

{dominicm, carlo}@graphcore.ai

The Limit of the Batch Size

Yang You¹, Yuhui Wang¹, Huan Zhang², Zhao Zhang³, James Demmel¹, Cho-Jui Hsieh²

UC Berkeley¹, UCLA², TACC³

{youyang, demmel}@cs.berkeley.edu, yuhui-w@berkeley.edu,
huanzhang@ucla.edu, zzhang@tacc.utexas.edu, chohsieh@cs.ucla.edu

Batch size

Scaling Laws for Neural Language Models

Jared Kaplan *

Johns Hopkins University, OpenAI
jaredk@jhu.edu

Sam McCandlish*

OpenAI
sam@openai.com

Tom Henighan

OpenAI
henighan@openai.com

Tom B. Brown

OpenAI
tom@openai.com

Benjamin Chess

OpenAI
bchess@openai.com

Rewon Child

OpenAI
rewon@openai.com

Scott Gray

OpenAI
scottgray@openai.com

Alec Radford

OpenAI
alec@openai.com

Jeffrey Wu

OpenAI
jeffwu@openai.com

Dario Amodei

OpenAI
damodei@openai.com

DON'T DECAY THE LEARNING RATE,
INCREASE THE BATCH SIZE

Samuel L. Smith*, Pieter-Jan Kindermans*, Chris Ying & Quoc V. Le
Google Brain
{slsmith, pikinder, chrisying, qvl}@google.com

ABSTRACT

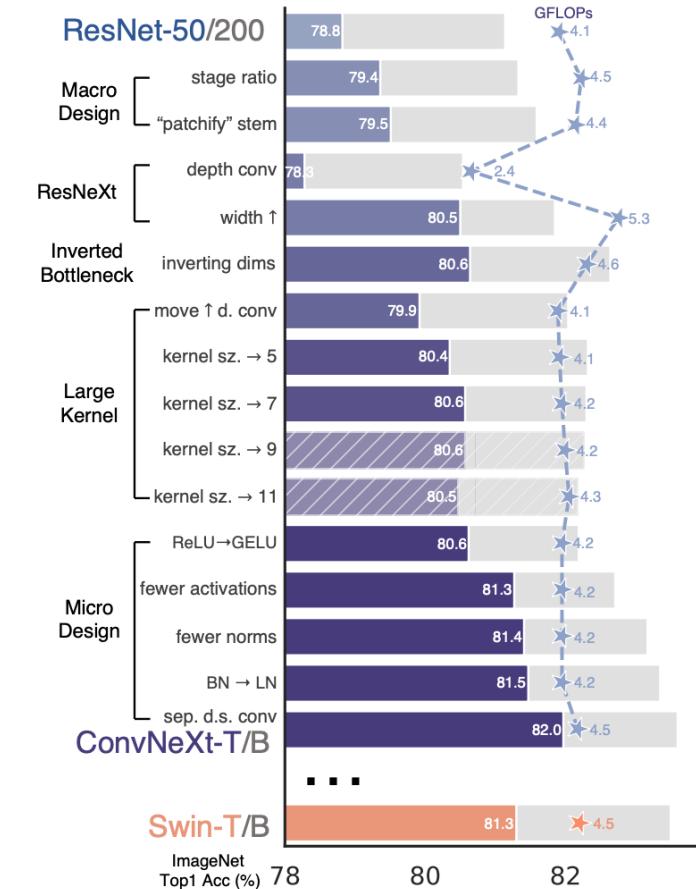
the biggest source of uncertainty is the architecture

Architecture hyperparameter tuning

Table 2: Ingredients and hyper-parameters used for ResNet-50 training in different papers. We compare existing training procedures with ours.

Procedure → Reference	Previous approaches					Ours		
	ResNet [13]	PyTorch [1]	FixRes [48]	DeiT [45]	FAMS (x4) [10]	A1	A2	A3
Train Res	224	224	224	224	224	224	224	160
Test Res	224	224	224	224	224	224	224	224
Epochs	90	90	120	300	400	600	300	100
# of forward pass	450k	450k	300k	375k	500k	375k	188k	63k
Batch size	256	256	512	1024	1024	2048	2048	2048
Optimizer	SGD-M	SGD-M	SGD-M	AdamW	SGD-M	LAMB	LAMB	LAMB
LR	0.1	0.1	0.2	1×10^{-3}	2.0	5×10^{-3}	5×10^{-3}	8×10^{-3}
LR decay	step	step	step	cosine	step	cosine	cosine	cosine
decay rate	0.1	0.1	0.1	-	$0.02^{1/400}$	-	-	-
decay epochs	30	30	30	-	1	-	-	-
Weight decay	10^{-4}	10^{-4}	10^{-4}	0.05	10^{-4}	0.01	0.02	0.02
Warmup epochs	x	x	x	5	5	5	5	5
Label smoothing ϵ	x	x	x	0.1	0.1	0.1	x	x
Dropout	x	x	x	x	x	x	x	x
Stoch. Depth	x	x	x	0.1	x	0.05	0.05	x
Repeated Aug	x	x	✓	✓	x	✓	✓	x
Gradient Clip.	x	x	x	x	x	x	x	x
H. flip	✓	✓	✓	✓	✓	✓	✓	✓
RRC	x	✓	✓	✓	✓	✓	✓	✓
Rand Augment	x	x	x	9/0.5	x	7/0.5	7/0.5	6/0.5
Auto Augment	x	x	x	x	✓	x	x	x
Mixup alpha	x	x	x	0.8	0.2	0.2	0.1	0.1
Cutmix alpha	x	x	x	1.0	x	1.0	1.0	1.0
Erasing prob.	x	x	x	0.25	x	x	x	x
ColorJitter	x	✓	✓	x	x	x	x	x
PCA lighting	✓	x	x	x	x	x	x	x
SWA	x	x	x	x	✓	x	x	x
EMA	x	x	x	x	x	x	x	x
Test crop ratio	0.875	0.875	0.875	0.875	0.875	0.95	0.95	0.95
CE loss	✓	✓	✓	✓	✓	x	x	x
BCE loss	x	x	x	x	x	✓	✓	✓
Mixed precision	x	x	x	✓	✓	✓	✓	✓
Top-1 acc.	75.3%	76.1%	77.0%	78.4%	79.5%	80.4%	79.8%	78.1%

ResNet strikes back: An improved training procedure in timm. Wightman et al. 2021



A ConvNet for the 2020s. Liu et al. CVPR 2022

Some magic starting numbers I'm aware of

or 0.1

Learning rate: 0.01 with a nice scheduler (my goto is standard multi-step)

Weight decay: 1e-4 or 5e-4

always works lol

Batch size: Biggest power of 2 that fits in memory

DropOut: 20-50% drop out rate

And make sure to compute the mean and variance of the training samples!

Babysitting deep networks

Establish baselines

Check that in the first round you get loss that corresponds to random guess

Check network with few samples

- Turn off regularization. You should predictably overfit and get a loss of 0
- Turn on regularization. The loss should be higher than before

Always a separate validation set for hyper-parameter tuning

- Compare the training and validation losses - there should be a gap, not too large

Preprocess the data (at least to have 0 mean)

Initialize weights based on activations functions Xavier or Kaiming initialization

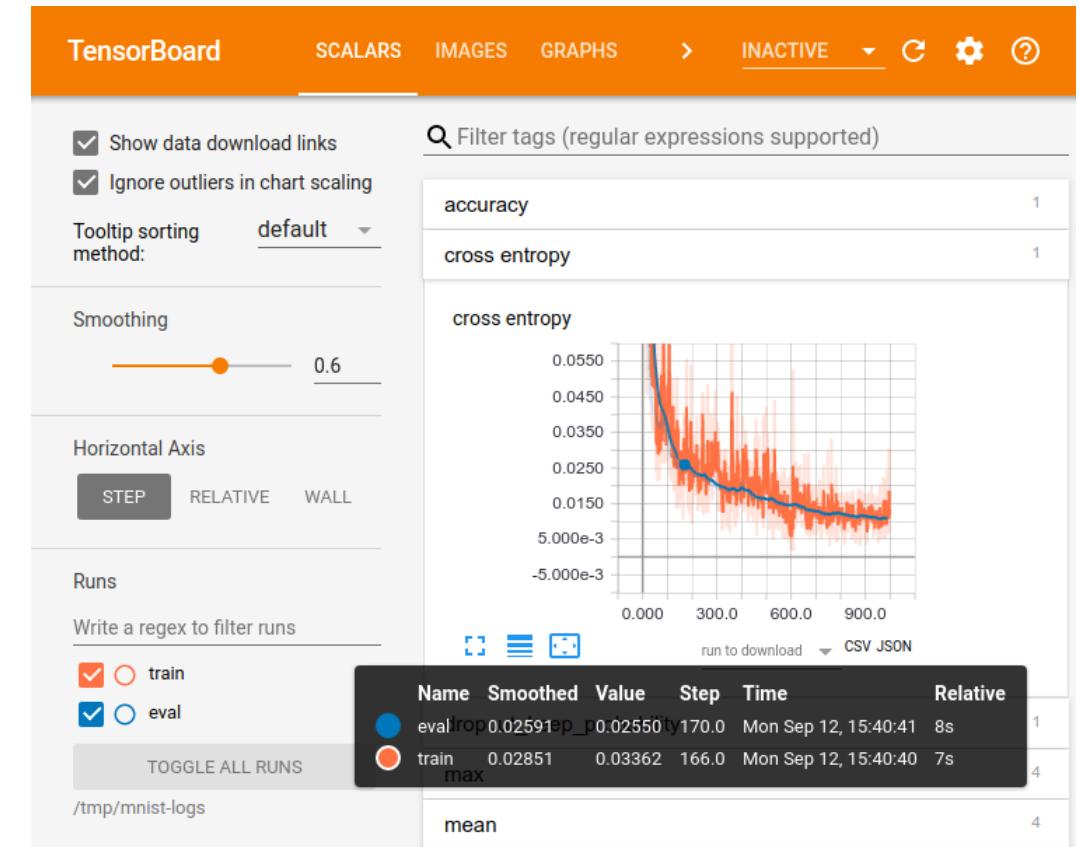
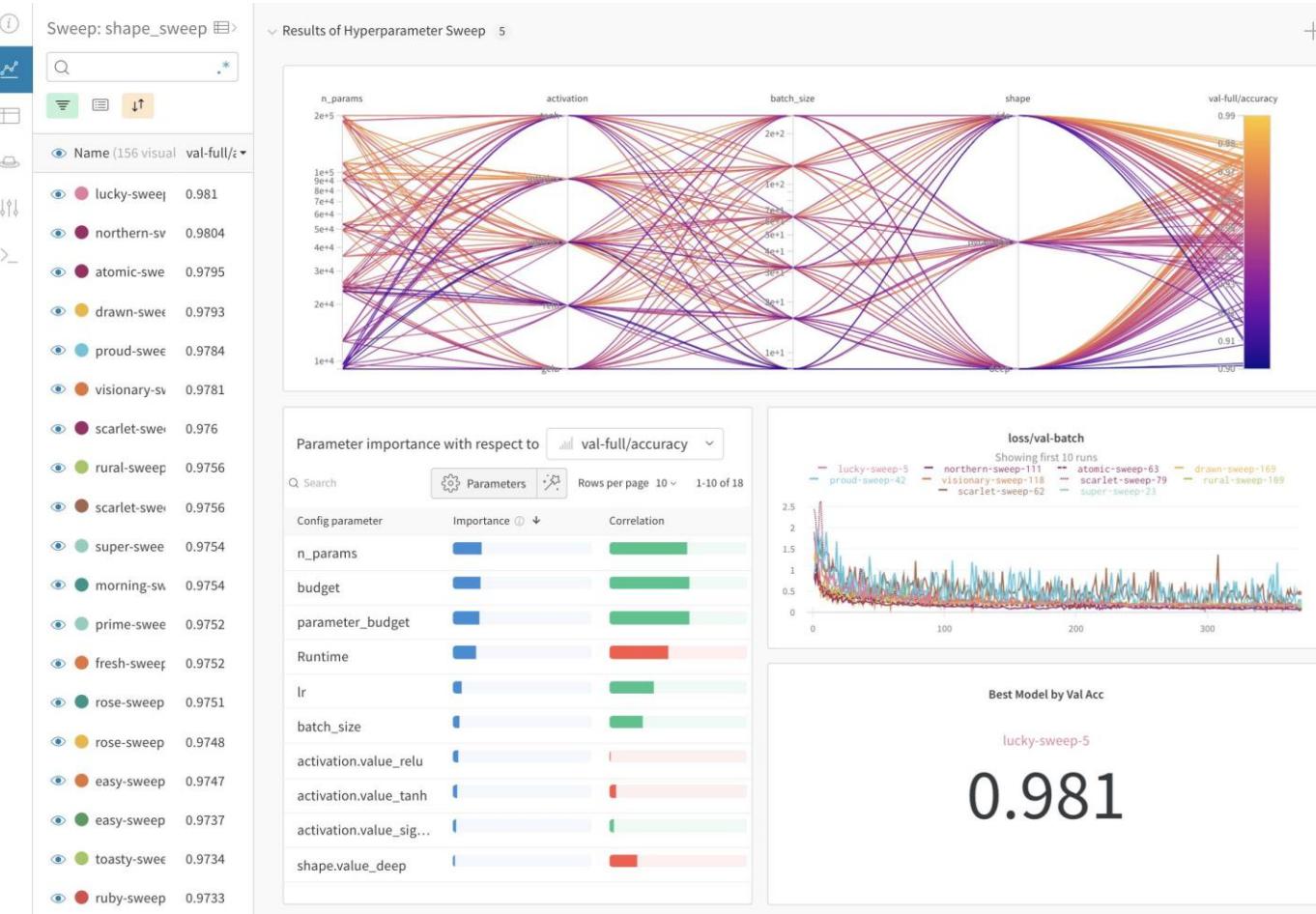
Use regularization (ℓ_2 -regularization, dropout, ...)batch normalization

Prefer residual connections, they make a difference

Use an experiment manager like tensorboard or Neptune or wand

Track further metrics that are relevant to the problem at hand

Logging tools



Summary

Tackling overfitting is key in deep learning.

Initialization: *een goed begin is het halve werk.*

Regularization, DropOut, early stopping, normalization: all dampening factors.

Your lives will be dominated by hyperparameters.

Learning and reflection

Understanding Deep Learning: Chapter 8

Understanding Deep Learning: Chapter 9

Next lecture

Lecture	Title	Lecture	Title
1	Intro and history of deep learning	2	AutoDiff
3	Deep learning optimization I	4	Deep learning optimization II
5	Convolutional deep learning	6	Attention-based deep learning
7	Graph deep learning	8	From supervised to unsupervised deep learning
9	Multi-modal deep learning	10	Generative deep learning
11	What doesn't work in deep learning	12	Non-Euclidean deep learning
13	Q&A	14	Deep learning for videos



Deep Learning 1

2025-2026 – Pascal Mettes

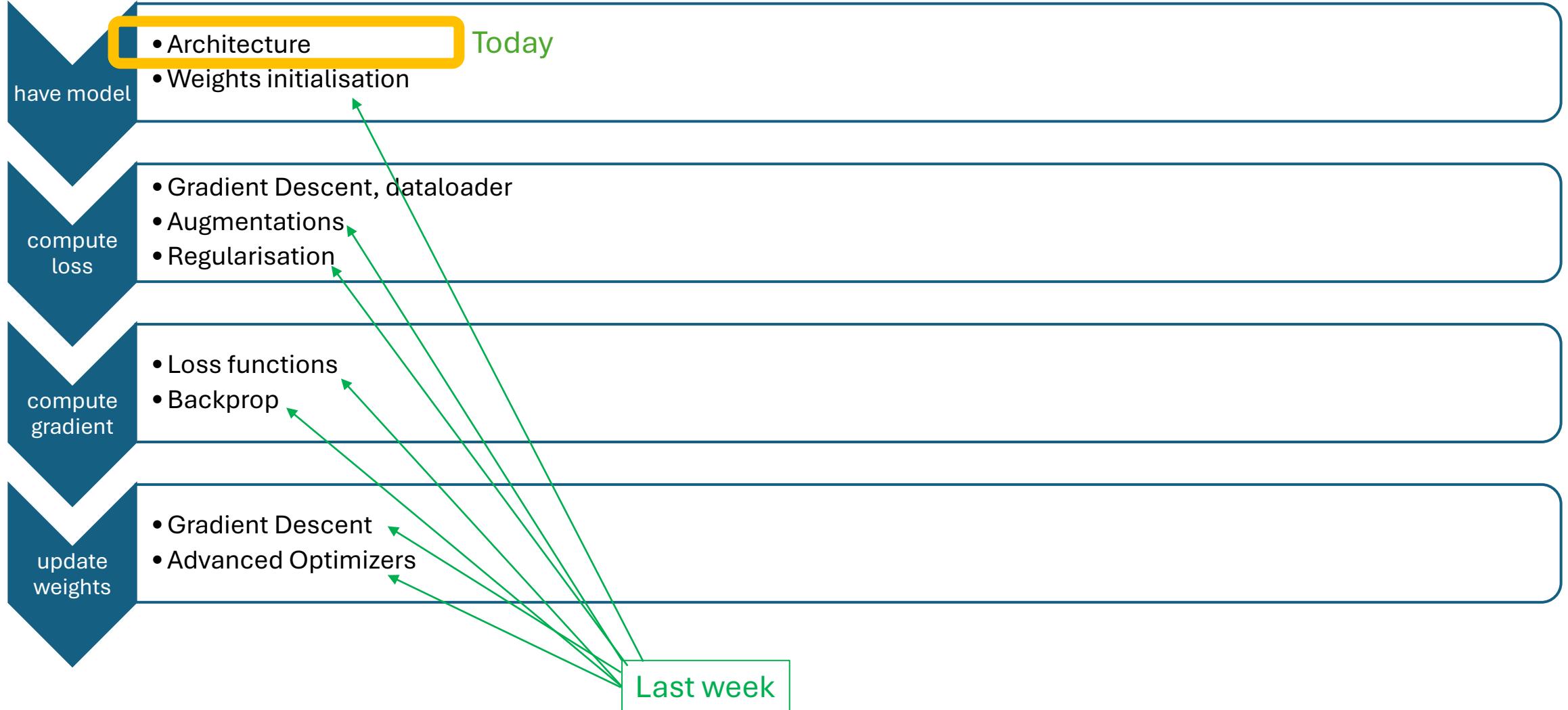
Lecture 5

Convolutional Neural Networks

Previous lecture

Lecture	Title	Lecture	Title
1	Intro and history of deep learning	2	AutoDiff
3	Deep learning optimization I	4	Deep learning optimization II
5	Convolutional deep learning	6	Attention-based deep learning
7	Graph deep learning	8	From supervised to unsupervised deep learning
9	Multi-modal deep learning	10	Generative deep learning
11	What doesn't work in deep learning	12	Non-Euclidean deep learning
13	Q&A	14	Deep learning for videos

Where are we

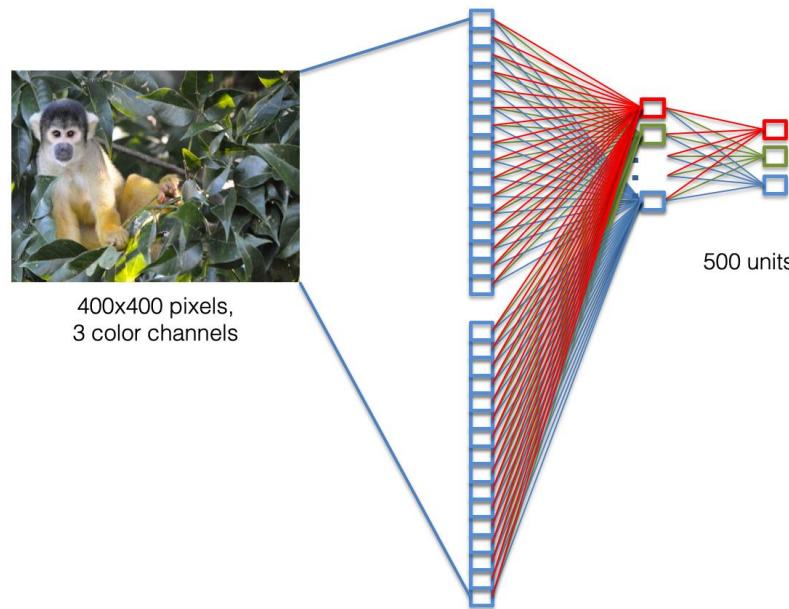


MLPs and real world data

Multi-layer perceptrons / feedforward networks assume vectorized data.

Consider an image of 400x400 pixels with 3 color channels.

How many parameters needed for a 1-layer networks with 500 hidden units?

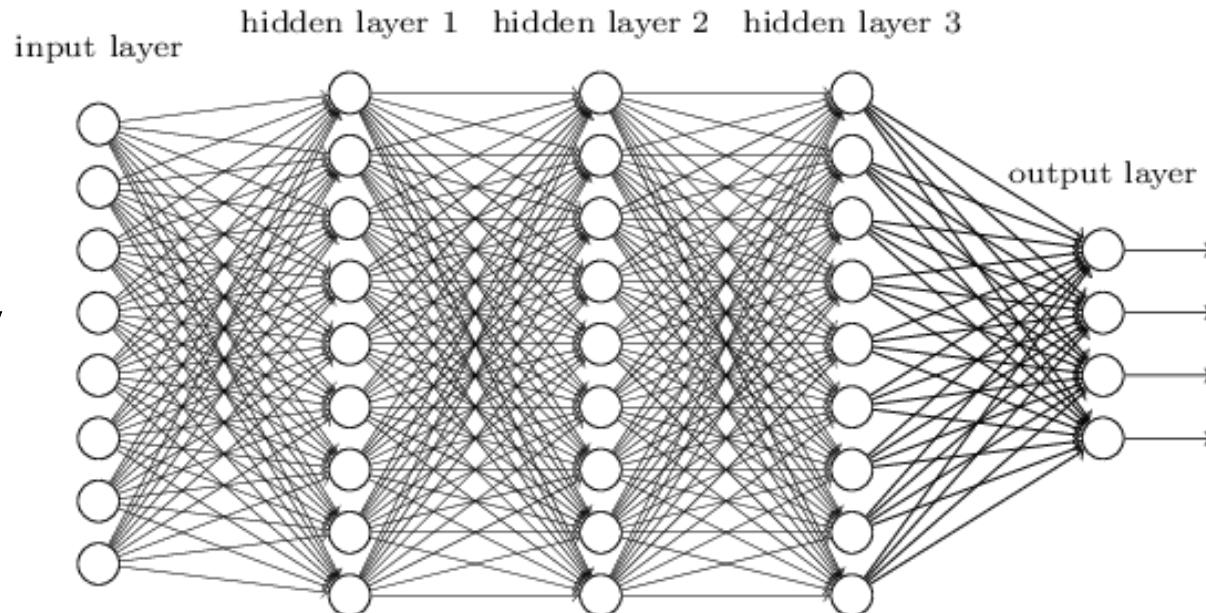


MLPs and real world data

From this fully-connected model, do we really need all the edges?

Can some of these be “shared” (equal weight)?

Can prior knowledge (“inductive biases”) be incorporated into the design?



The data that we need for deep learning is not in the form of feature vector. Feature vectors are data that have been already manipulated by human. Deep learning operates with raw data

The inductive bias is a structure in our architecture that should be reflected in the data. This should help the learning

The convolution

We have a signal $x(t)$ and a weighting function $w(a)$

We can generate a new function $s(t)$ by the following equation:

$$s(t) = \int x(a)w(t - a)da$$

We refer to $w(a)$ as a *filter or a kernel*. This operation is called convolution.

The convolution operation is typically denoted with asterisk:

$$s(t) = (x * w)(t)$$

Convolution vs linear operators

convolution is a linear layer but done locally many times

Linear operation: $f(x, w) = x^T w$

Global operation, separate weight per feature

Dimensionality of w = dimensionality of x

1 output value

Convolutional operation: $f(x, w) = x * w$

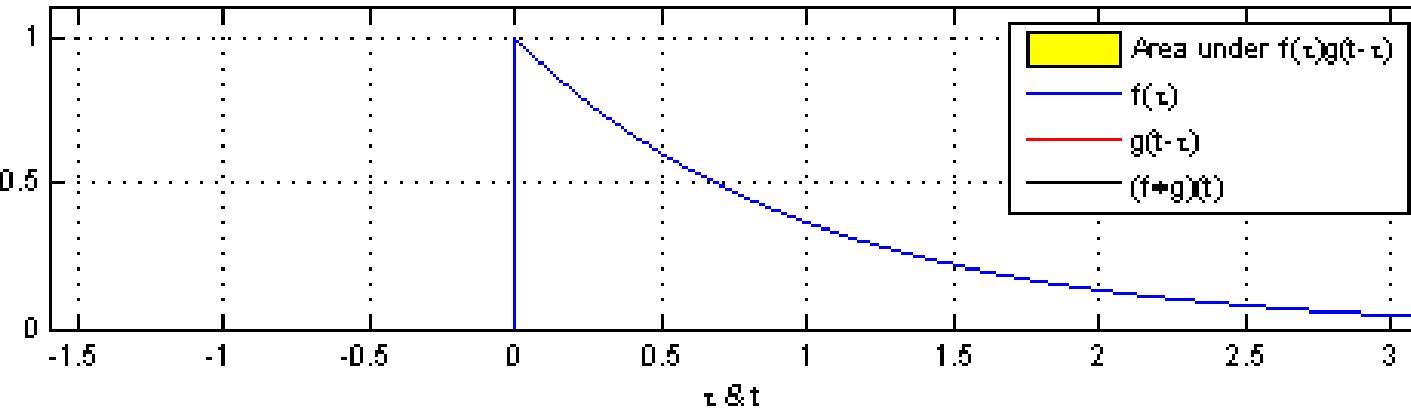
Local operation, shared weights over local regions

Dimensionality of w much smaller than dimensionality of x

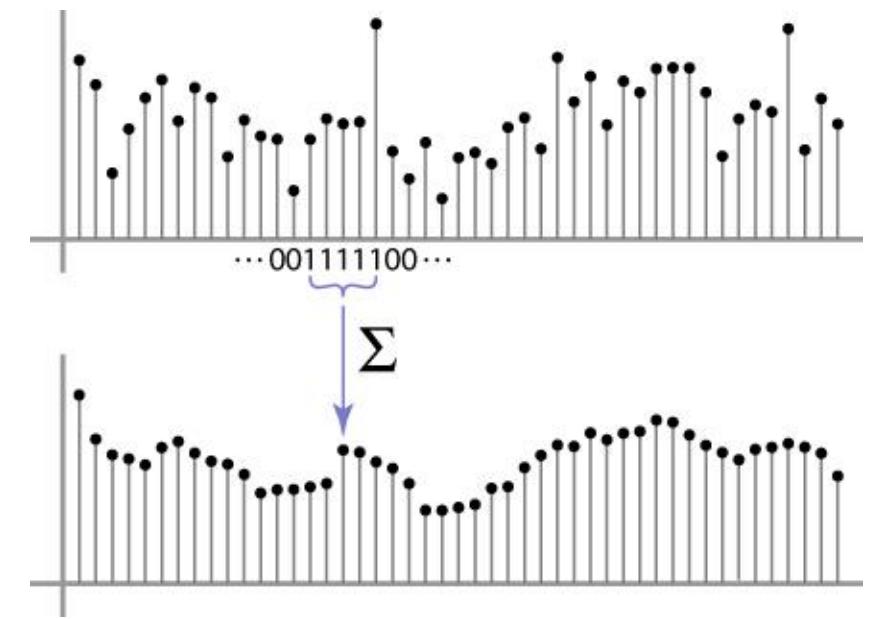
Output (almost?) same size as input

1D convolutions

The convolution $(f * g)(t)$ of two functions $f(t)$ and $g(t)$
computes the overlap in area.



It is taking the average of the signal (smoothing), always use even
(or odd?) number?? don't get it



2D convolutions

For a two-dimensional image I as our input, we want to use a two-dimensional kernel K :

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n)K(i - m, j - n)$$

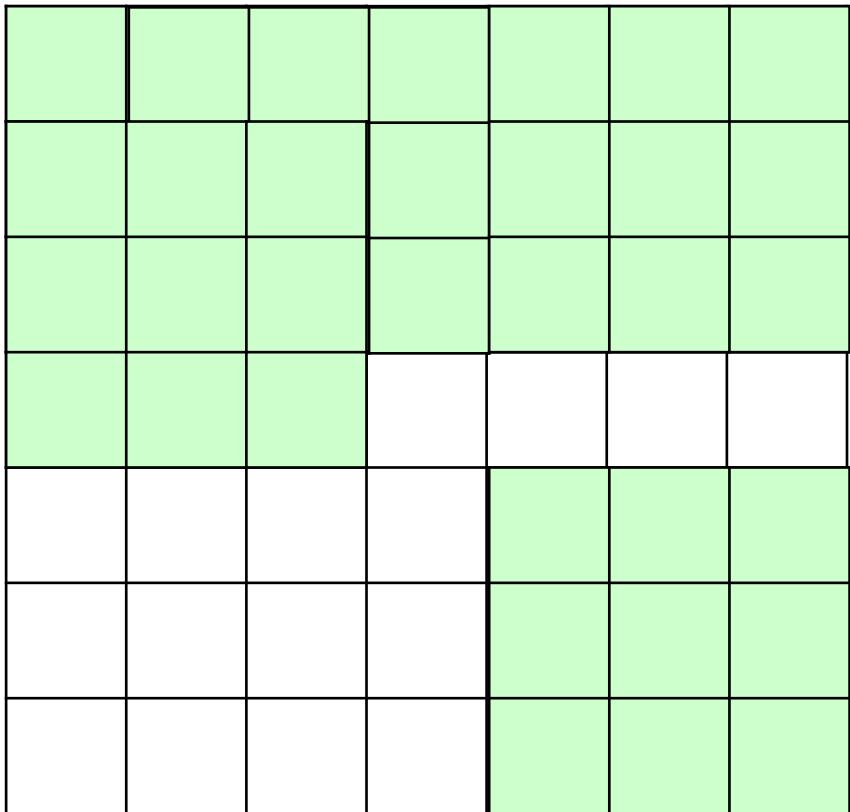
Convolution is commutative, and we can equivalently write:

$$S(i, j) = (K * I)(i, j) = \sum_m \sum_n I(i - m, j - n)K(m, n)$$

Neural networks libraries implement cross-correlation:

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(i + m, j + n)K(m, n)$$

2D convolution example



Input image: 7x7

Filter size: 3x3

Do the convolution by sliding the filter over all possible image locations.

What is the size of the output?

Other 2D convolution example

$$F[x, y]$$

$$\star$$

$$H[a, b]$$

$$G[x, y]$$

0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0	0
0	0	0	90	90	90	90	90	0	0	0
0	0	0	90	90	90	90	90	0	0	0
0	0	0	90	90	90	90	90	0	0	0
0	0	0	90	0	90	90	90	0	0	0
0	0	0	90	90	90	90	90	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	90	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0

$$\frac{1}{9} \begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix}$$

$$G = H \star F$$

Other 2D example

0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0	0
0	0	0	90	90	90	90	90	0	0	0
0	0	0	90	90	90	90	90	0	0	0
0	0	0	90	0	90	90	90	0	0	0
0	0	0	90	90	90	90	90	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	90	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0

$F[x, y]$

	0	10	20	30	30	30	20	10		
	0	20	40	60	60	60	40	20		
	0	30	60	90	90	90	60	30		
	0	30	50	80	80	90	60	30		
	0	30	50	80	80	90	60	30		
	0	20	30	50	50	60	40	20		
	10	20	30	30	30	30	20	10		
	10	10	10	0	0	0	0	0		

$G[x, y]$

Let's test your convolution intuition



Original

0	0	0
0	1	0
0	0	0



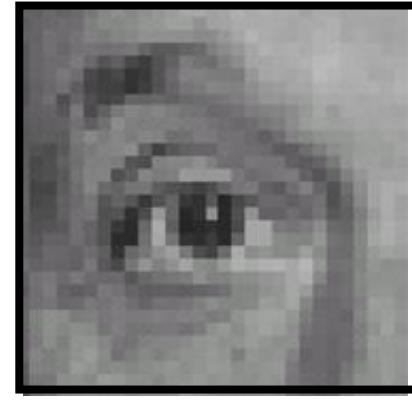
Filtered
(no change)

Let's test your convolution intuition



Original

0	0	0
0	0	1
0	0	0



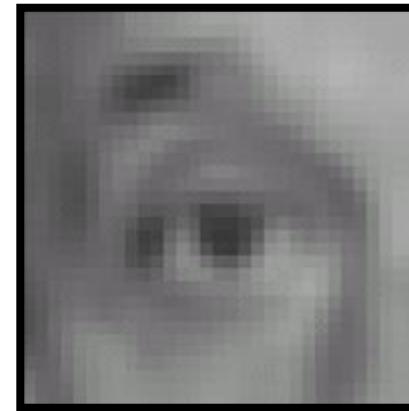
Filtered
(shift left)

Let's test your convolution intuition



Original

$$\frac{1}{9} \begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix}$$



Filtered
(blur)

Let's test your convolution intuition



Original

0	0	0
0	2	0
0	0	0

$$- \frac{1}{9} \begin{array}{|ccc|} \hline 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ \hline \end{array}$$



Filtered
(sharpening)

Motivations for convolutions

Sparse interaction, or local connectivity.

- *The receptive field of the neuron, or the filter size.*
- *The connections are local in space (width and height), but full in depth.*
- *A set of learnable filters.*

Parameters sharing, the weights are tied.

Equivariant representation (spatially): same operation at different places.

1. Sparsity

Sparse interaction, or local connectivity

- This is accomplished by making the kernel smaller than the input.
- reduces the memory requirements of the model
- improves its statistical efficiency.

s_1 s_2 s_3 s_4 s_5 s_1 s_2 s_3 s_4 s_5

x_1 x_2 x_3 x_4 x_5 x_1 x_2 x_3 x_4 x_5

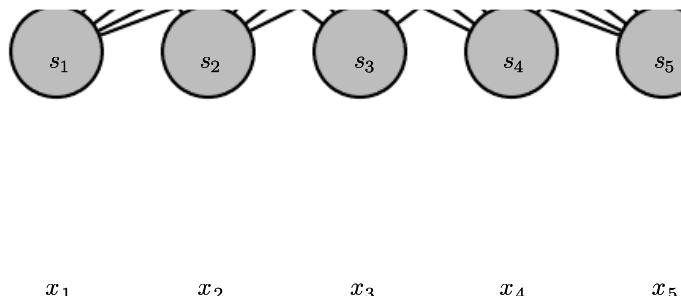
sparse connection

full connection

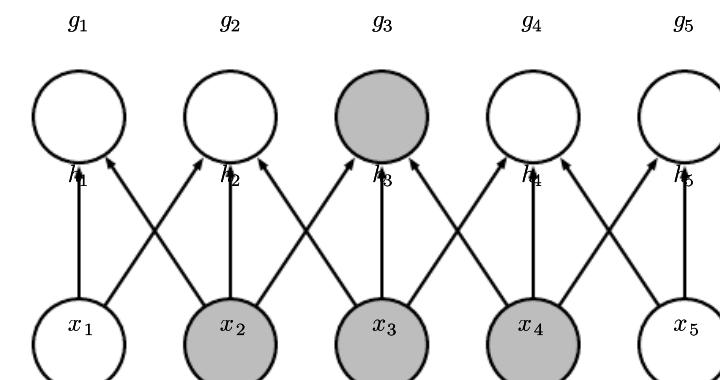
1. Sparsity

Sparse interaction, or local connectivity.

- Receptive field is the kernel/filter size.
- The receptive field of the units in the deeper layers of a convolutional network is larger than the receptive field of the units in the shallow layers.



convolution with a kernel of width 3

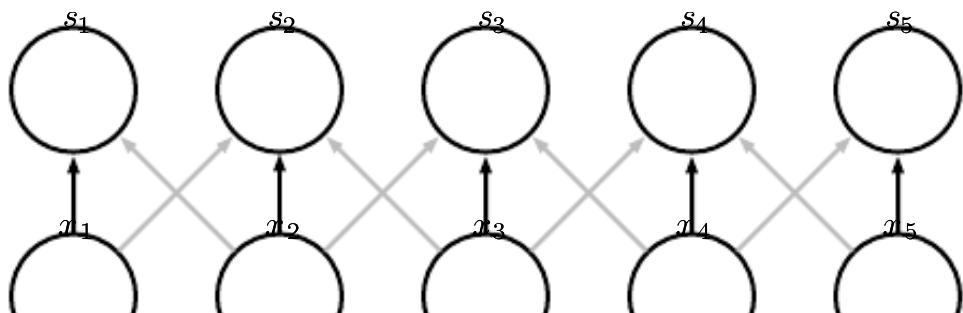


receptive field of the units in the deeper layers

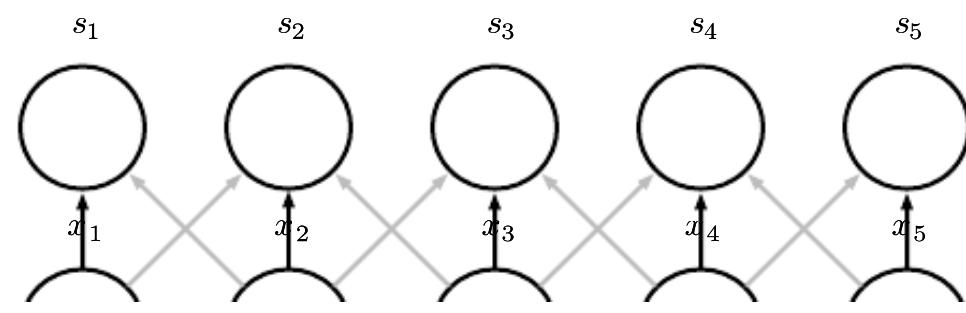
2. Parameter sharing

Parameters sharing, the weights are tied.

- refers to using the same parameter for more than one function in a model.
- each member of the kernel is used at every position of the input.



parameter sharing



no parameter sharing

3. Equivariance

Equivariant representation

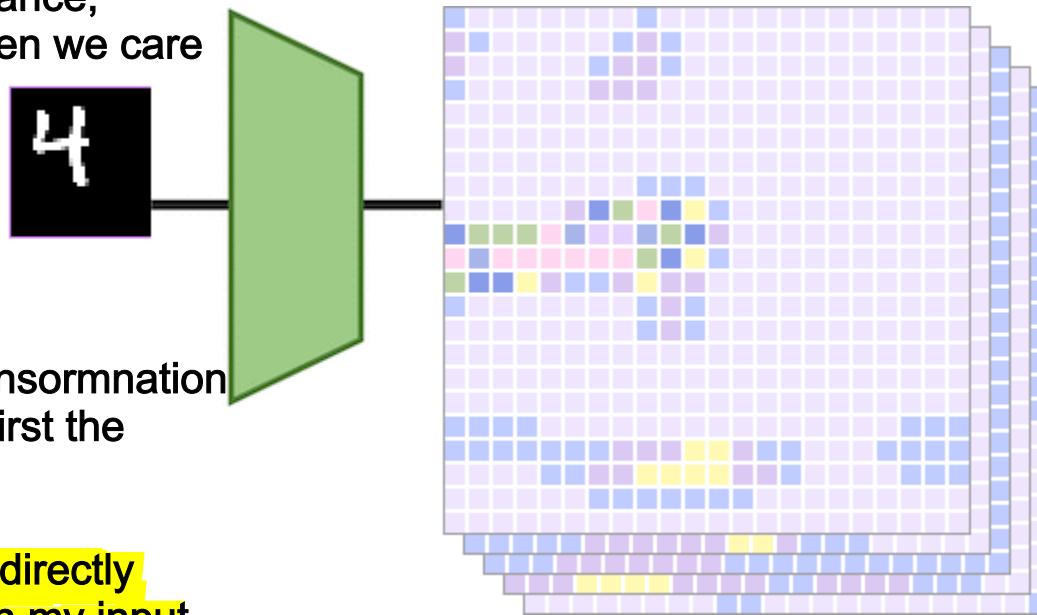
- *Parameter sharing causes the layer to have equivariance to translation.*
- *A function is equivariant if the input changes, the output changes in the same way.*
- *Why do we want equivariance to translation?*

all of the deep learning is about invariance, (invariance of the inputs). But very often we care also of equivariance.

$$\begin{aligned} x - & \rightarrow f(x) \\ | & | \\ g(x) - & \rightarrow f(g(x)) \end{aligned}$$

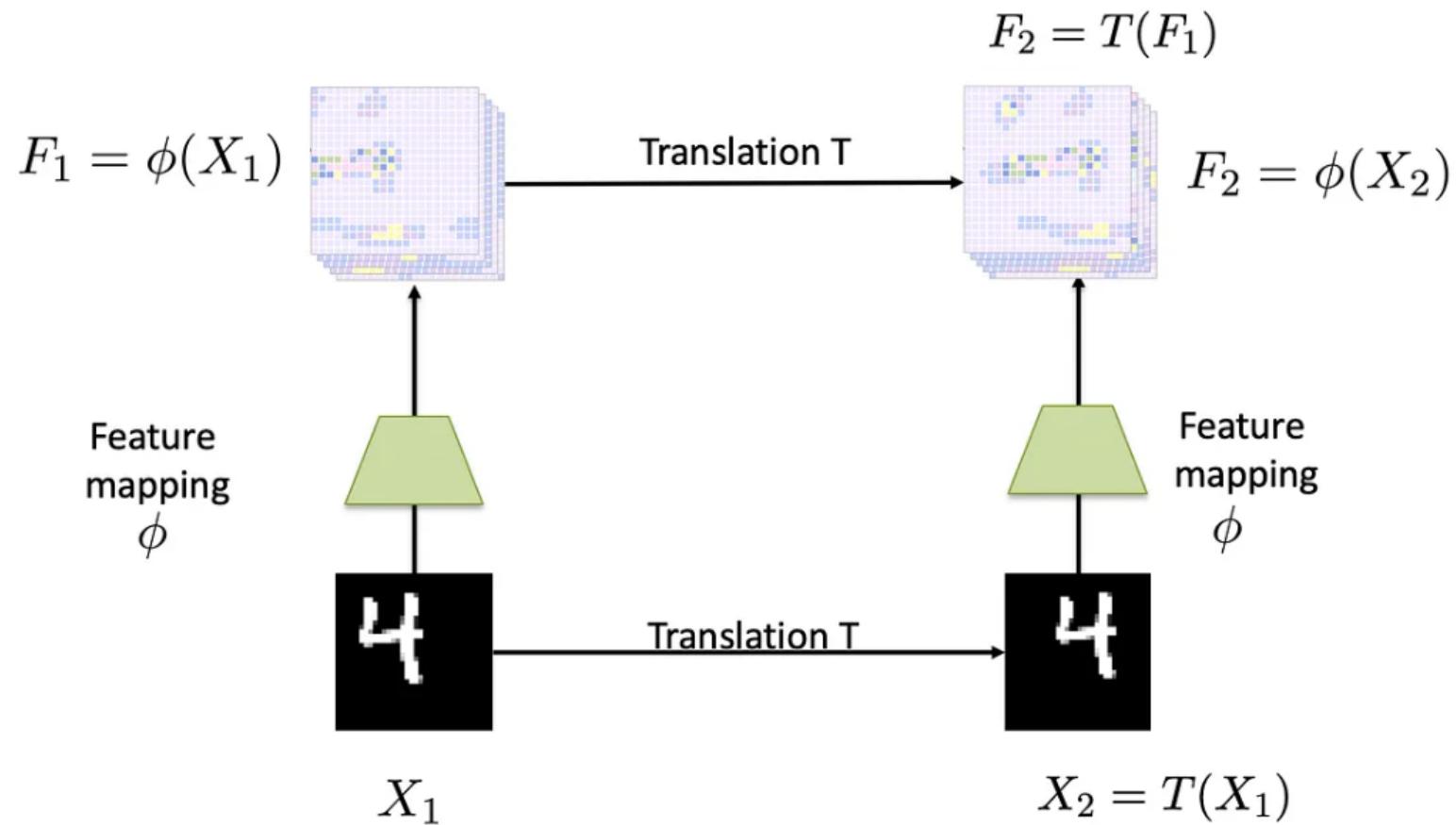
It doesn't mean if I do first a group transformation and the apply te function or if I apply first the function and then the transformation

The amount changing in my output is directly proportional to the amount changing in my input



Our real objective is to find a neural mapping from images to predictions, which is invariant to all different geometric transformations to which objects could be subject to, for instance translations, rotations and projective distortions.

an equivariant mapping is a mapping which preserves the algebraic structure of a transformation. As a particular case, a translation equivariant mapping is a mapping which, when the input is translated, leads to a translated mapping



Dealing with borders

We can't convolve border pixels, as filter windows extends beyond the image.

As is, this means that each convolution makes the image output smaller.

Solution: re-introduce a set of border pixels for every convolution.

zero-padding is the most weird to use because you add pixels which are not even in the range of the others, but it is the most widely used, so apparently it is not a big deal



Zero padding



Wrap around



Copy



Reflect

Convolutional networks pipeline

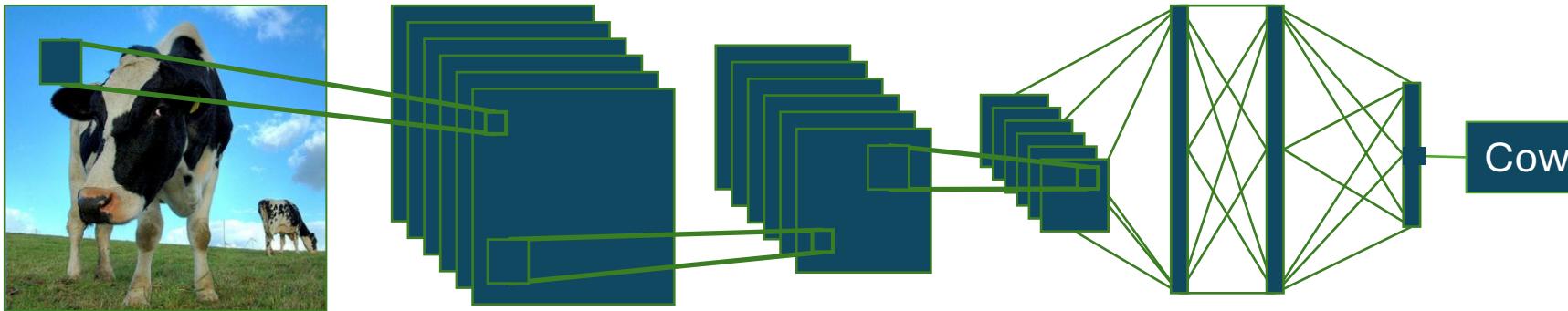


Image is the input, which should go through layers and ultimately predict label.

We want to *learn* the filter values to help us recognize classes.

3D Activations

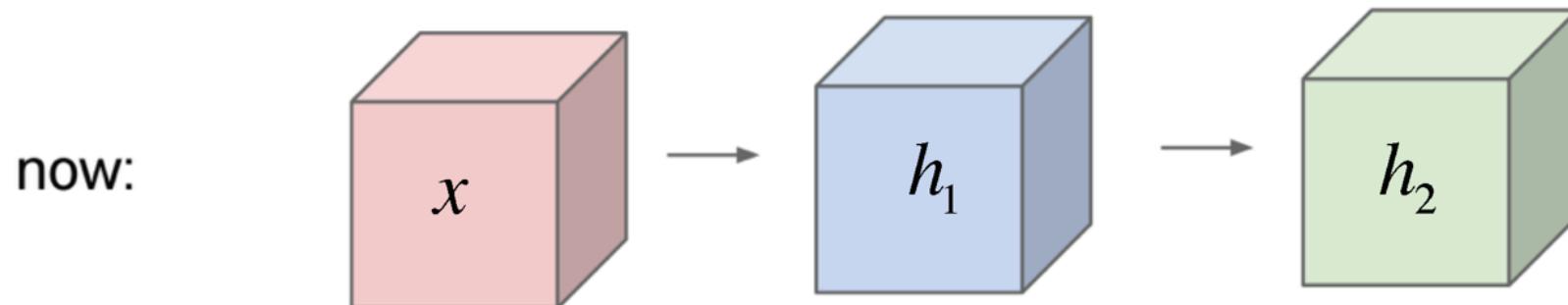
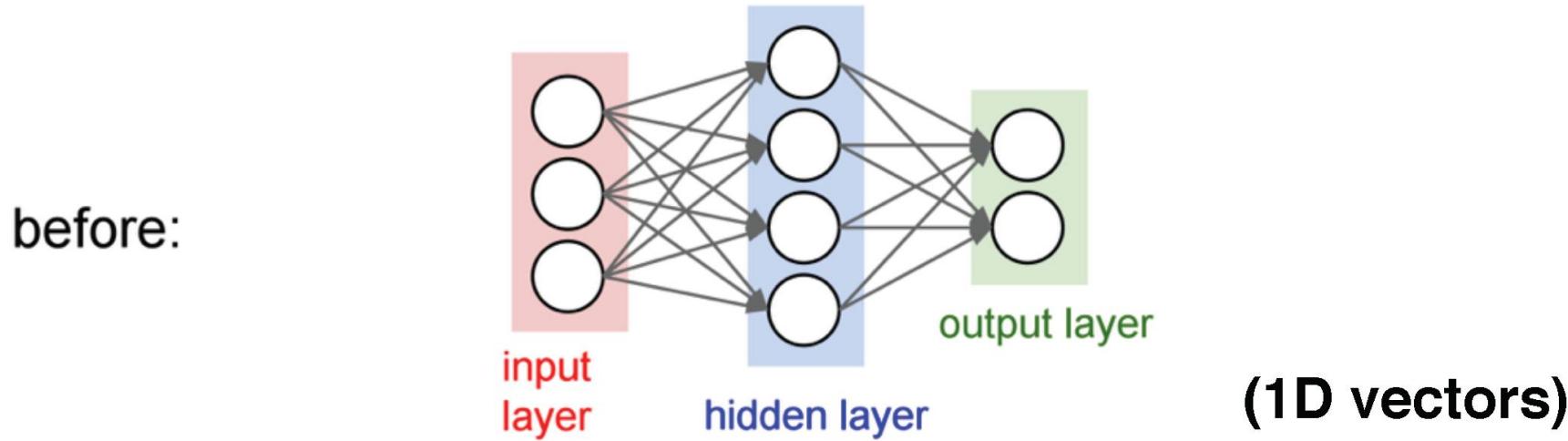
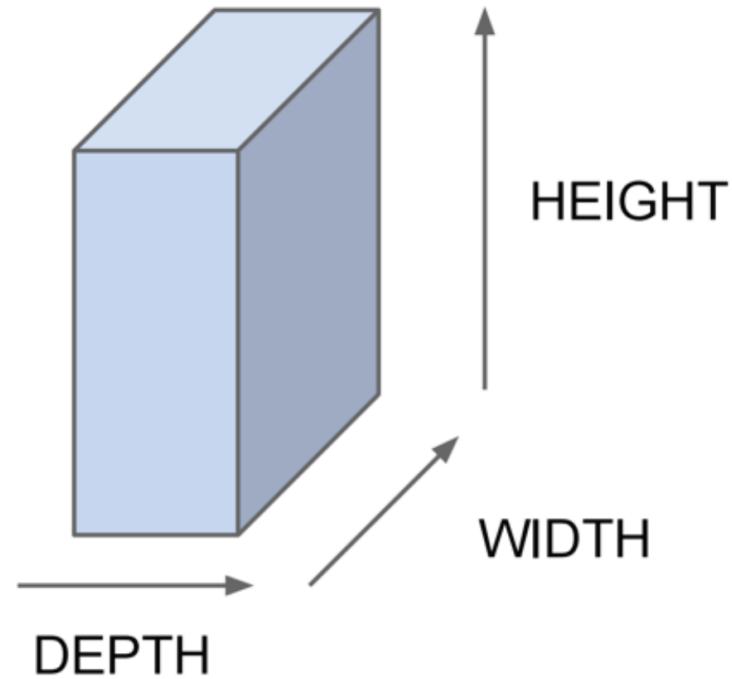


Figure: Andrej Karpathy

(3D arrays)

3D Activations

All Neural Net activations arranged in **3 dimensions**:

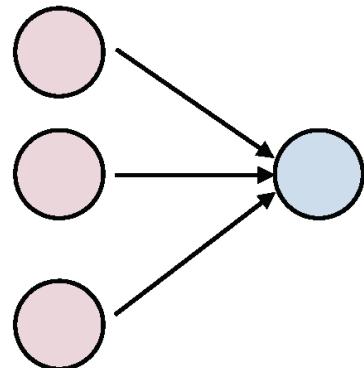


For example, a CIFAR-10 image is a $3 \times 32 \times 32$ volume
(3 depth — RGB channels, 32 height, 32 width)

Figure: Andrej Karpathy

3D Activations

1D Activations:



3D Activations:

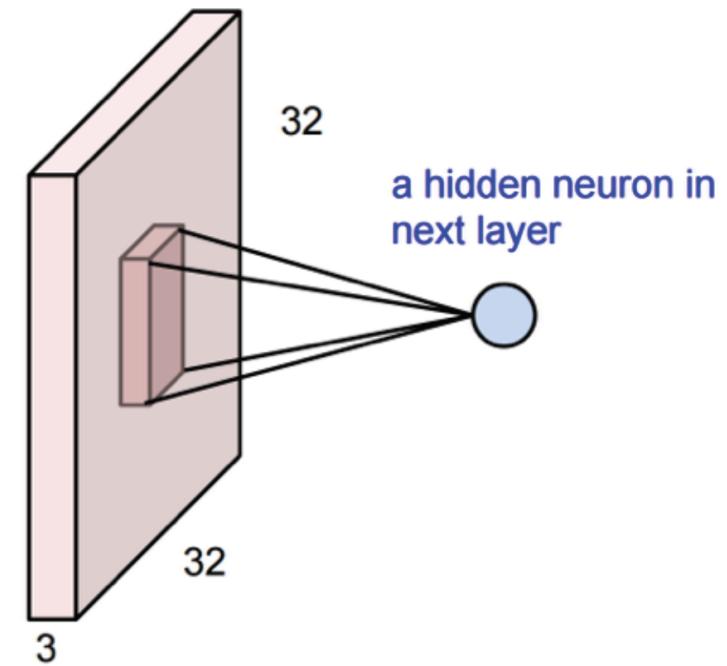
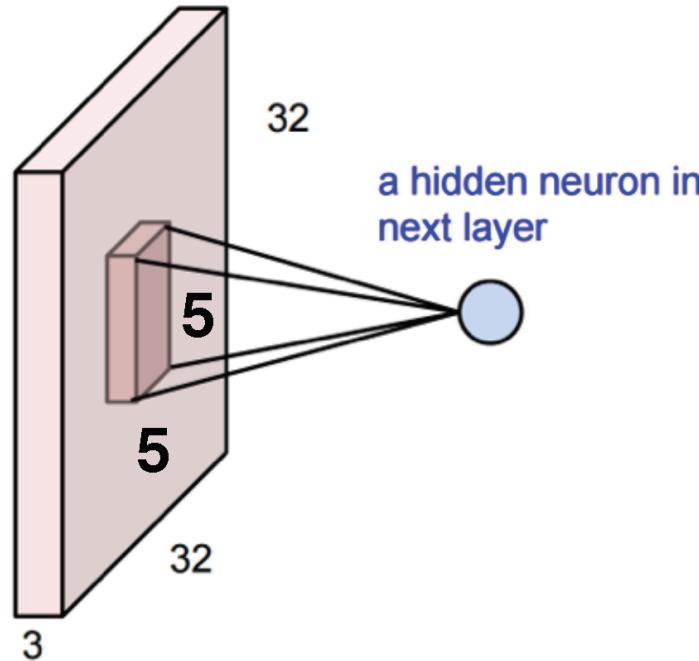


Figure: Andrej Karpathy

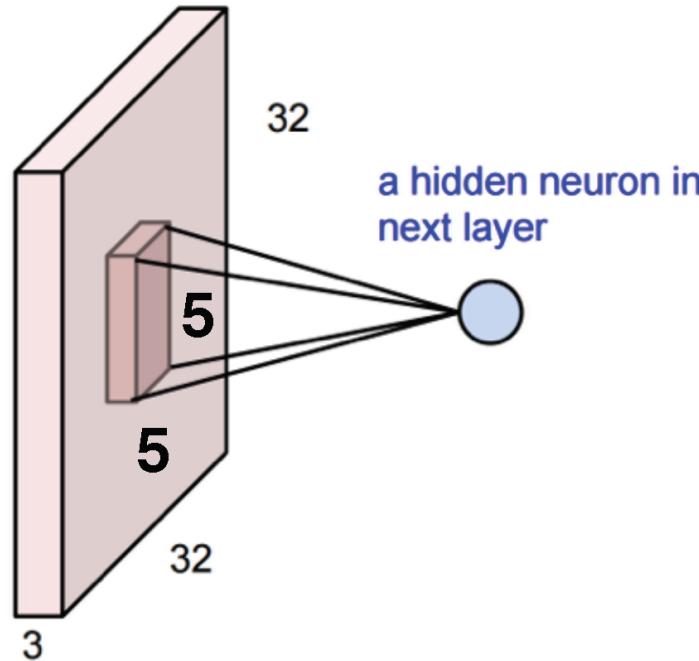
3D Activations



- The input is $3 \times 32 \times 32$
- This neuron depends on a $3 \times 5 \times 5$ chunk of the input
- The neuron also has a $3 \times 5 \times 5$ set of weights and a bias (scalar)

Figure: Andrej Karpathy

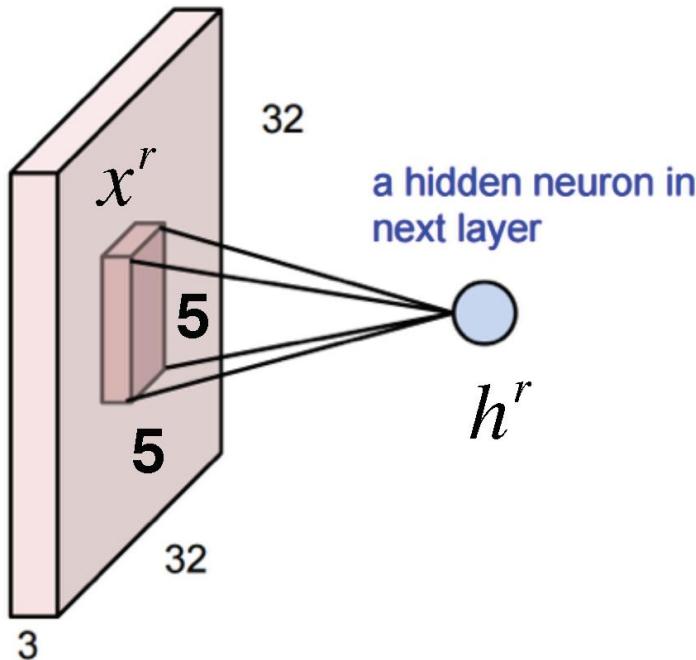
3D Activations



- The input is $3 \times 32 \times 32$
- This neuron depends on a $3 \times 5 \times 5$ chunk of the input
- The neuron also has a $3 \times 5 \times 5$ set of weights and a bias (scalar)

Figure: Andrej Karpathy

3D Activations



Example: consider the region of the input " x^r "

With output neuron h^r

Then the output is:

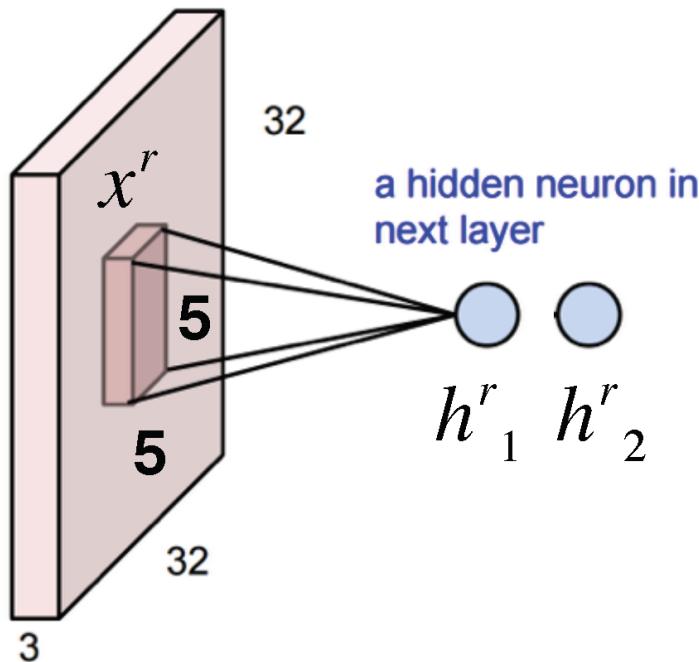
$$h^r = \sum_{ijk} x^r_{ijk} W_{ijk} + b$$

Sum over 3 axes

Figure: Andrej Karpathy

3D Activations

each individual filters
create 1 output values,
if i need multiple
outputs i need multiple
filters



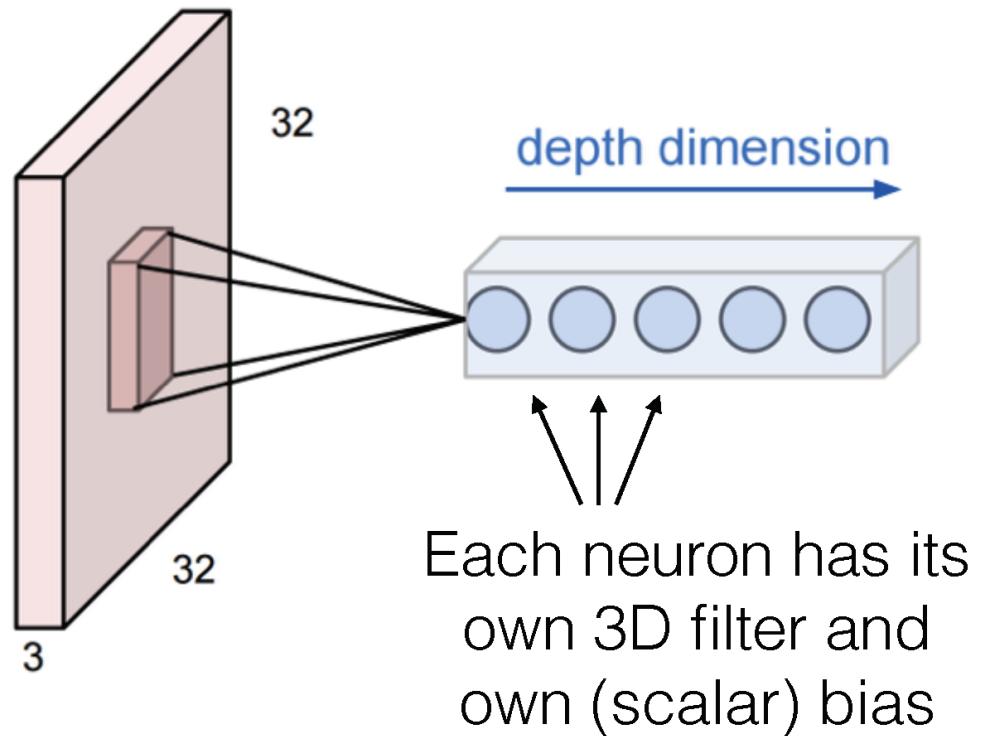
With **2** output neurons

$$h^r_1 = \sum_{ijk} x^r_{ijk} W_{1ijk} + b_1$$

$$h^r_2 = \sum_{ijk} x^r_{ijk} W_{2ijk} + b_2$$

Figure: Andrej Karpathy

3D Activations

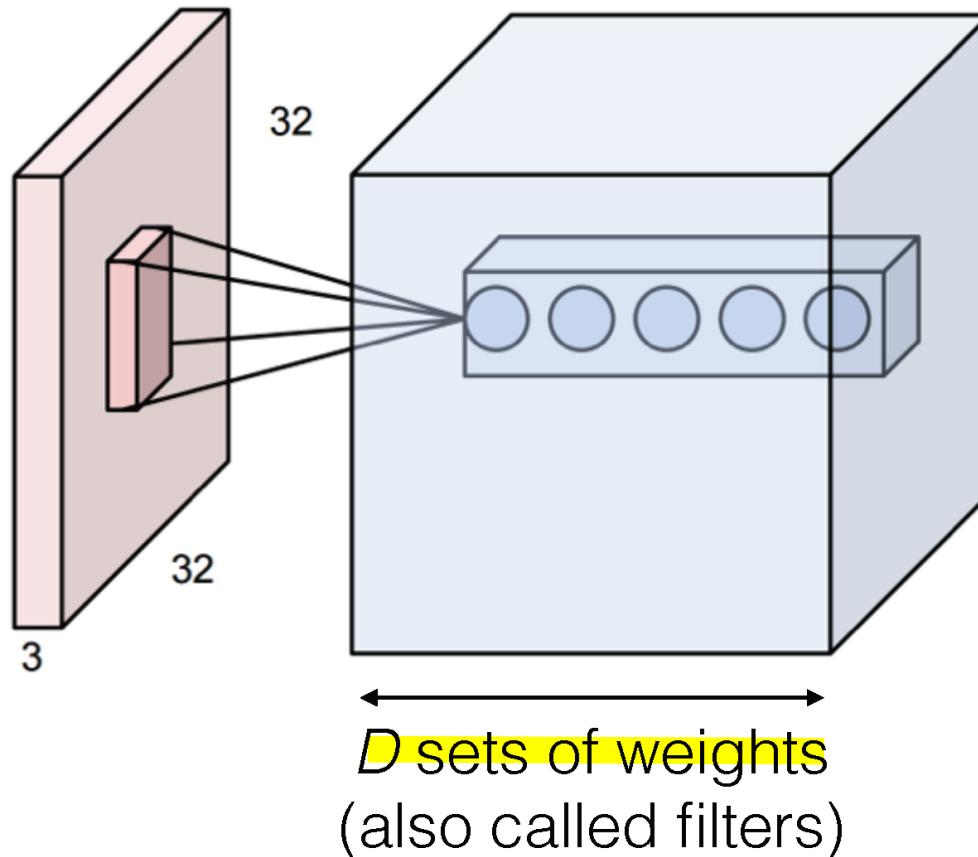


We can keep adding more outputs

These form a column in the output volume:
[depth x 1 x 1]

Figure: Andrej Karpathy

3D Activations



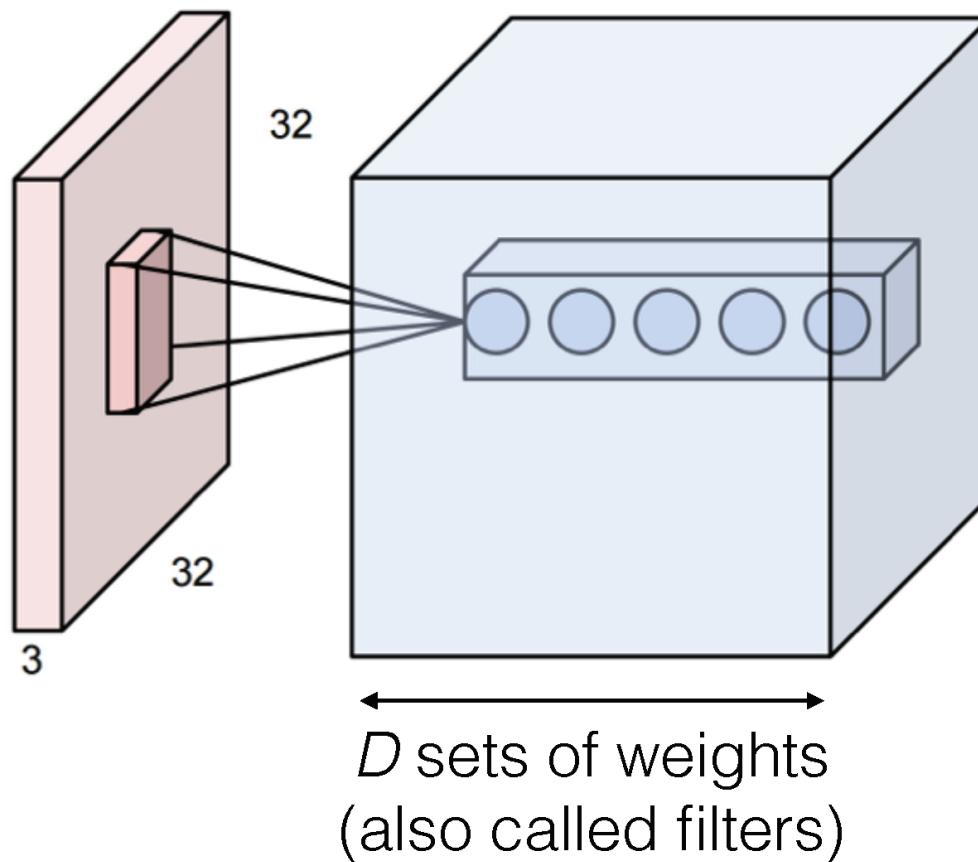
Now repeat this across the input

Weight sharing:

Each filter shares the same weights
(but each depth index has its own set of weights)

Figure: Andrej Karpathy

3D Activations



With weight sharing,
this is called
convolution

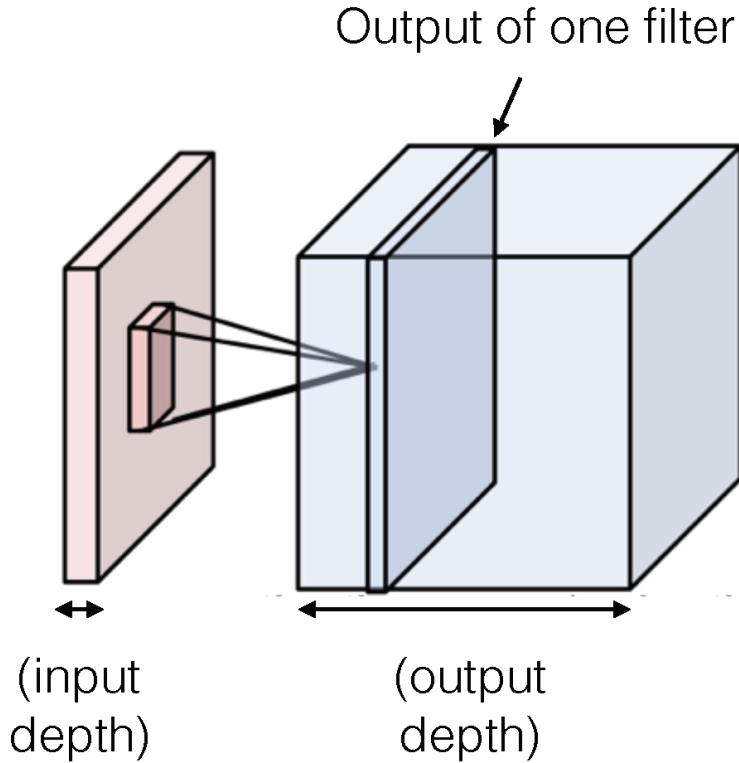
Without weight sharing,
this is called a
**locally
connected layer**

Figure: Andrej Karpathy

each filter is one slice, it creates its own image which is of 1 channel dimension. Typically the value of channels is much higher compared to the one of the input

the weight are 4 dimensionals

3D Activations



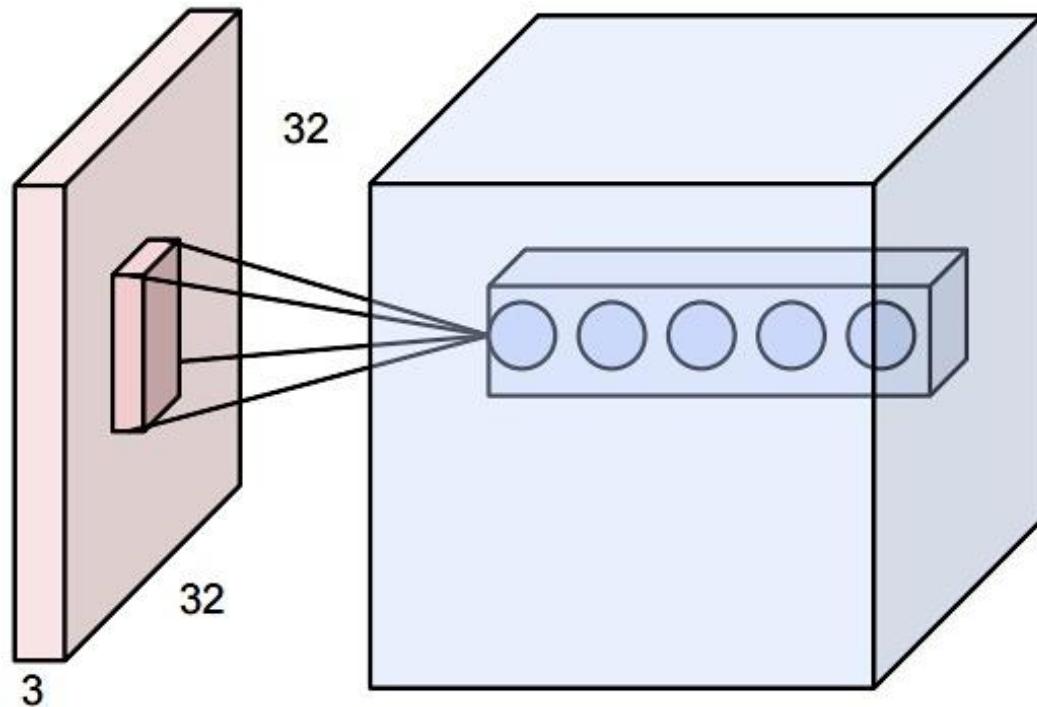
One set of weights gives
one slice in the output

To get a 3D output of depth D ,
use D different filters

In practice, ConvNets use
many filters (~64 to 1024)

All together, the weights are **4** dimensional:
(output depth, input depth, kernel height, kernel width)

Quick test



Q1: What does the 3 mean?

RGB

Q2: What is the output size for D filters (with padding)?

32x32xD

Q3: Does the output size depend on the input size or the filter size (with padding)?

Input size

Filters and layer dimensionality

First layer:

Input: $32 \times 32 \times 3$ and A filters of size $5 \times 5 \times 3$

Output: $32 \times 32 \times A$

Second layer:

Input: $32 \times 32 \times A$ and B filters of size $5 \times 5 \times [?]$

Output size: $32 \times 32 \times [?]$

3D Activations

We can unravel the 3D cube and show each layer separately:

(Input)

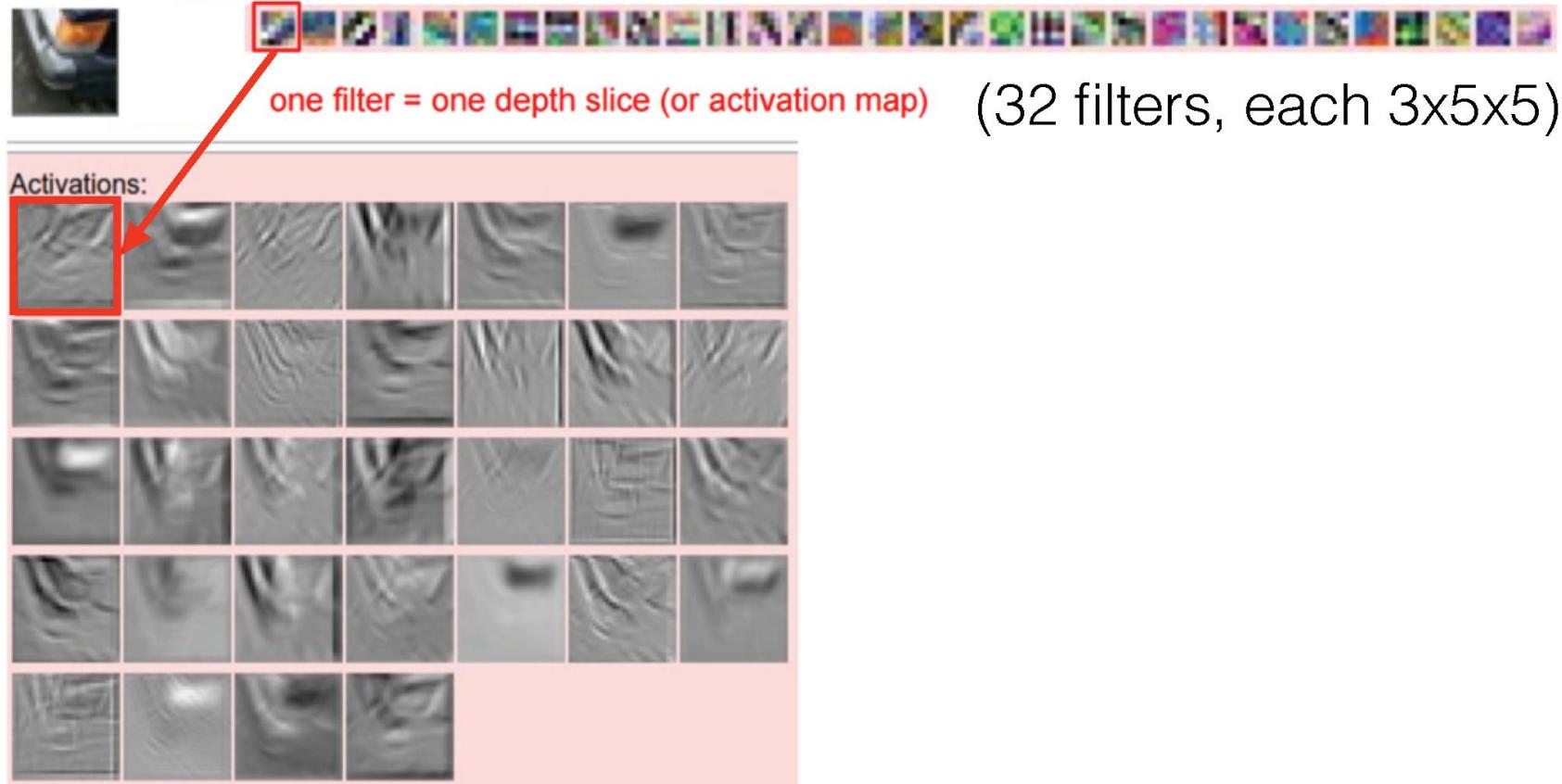


Figure: Andrej Karpathy

3D Activations

We can unravel the 3D cube and show each layer separately:

(Input)



Figure: Andrej Karpathy

3D Activations

We can unravel the 3D cube and show each layer separately:

(Input)

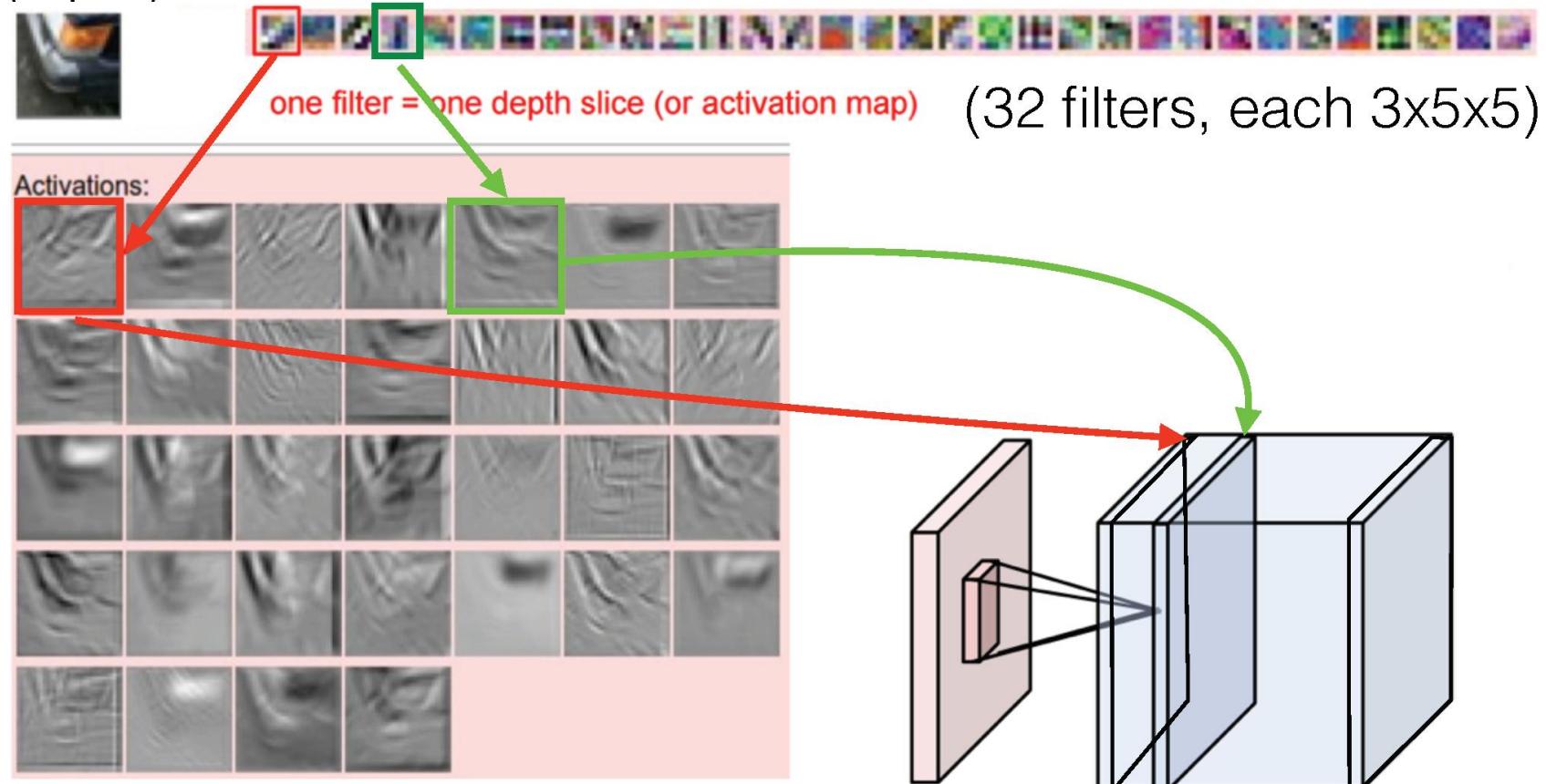


Figure: Andrej Karpathy

3D Activations

We can unravel the 3D cube and show each layer separately:

(Input)

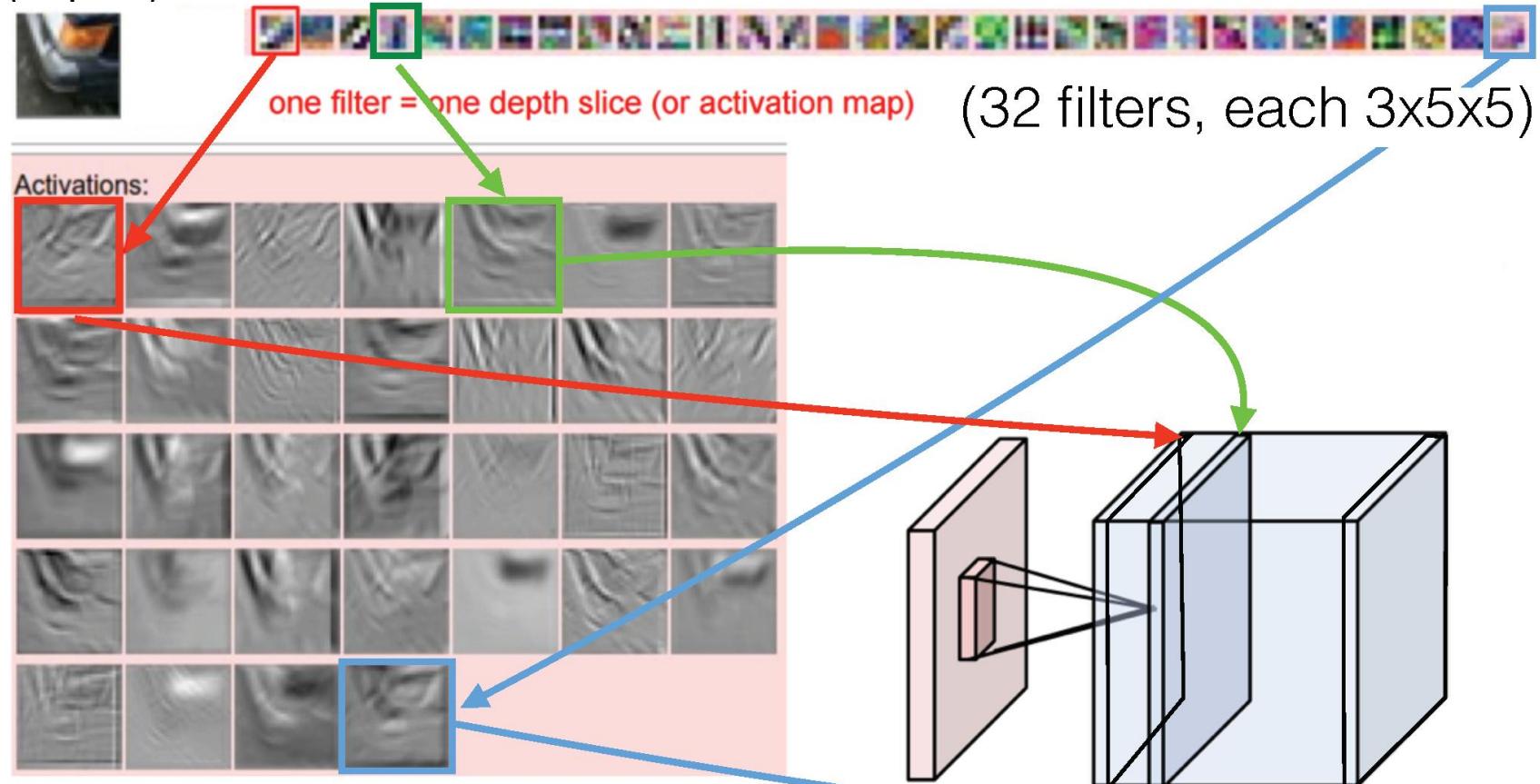
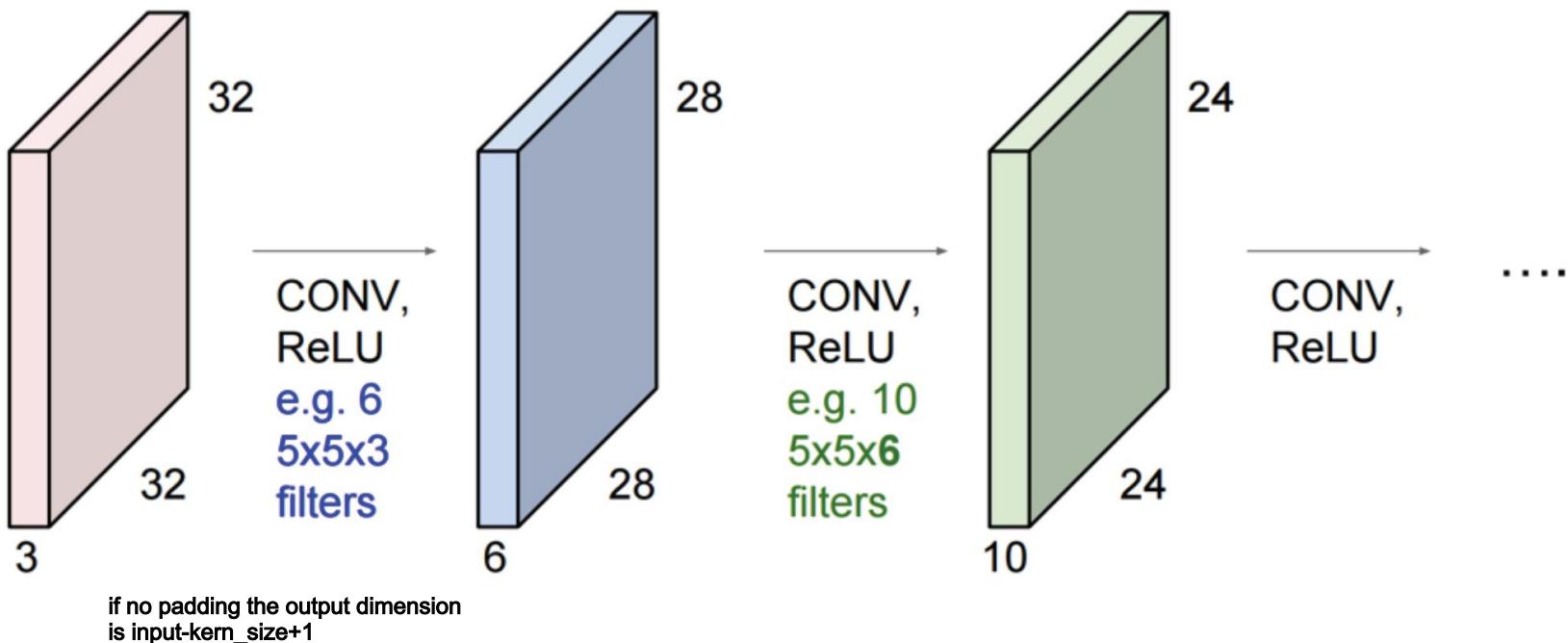


Figure: Andrej Karpathy

Putting it together

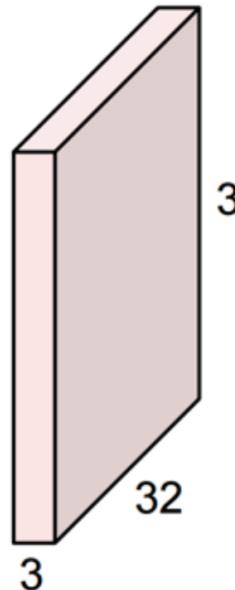
A **ConvNet** is a sequence of convolutional layers, interspersed with activation functions (and possibly other layer types)



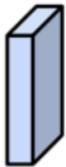
Putting it together

Convolution Layer

32x32x3 image



5x5x3 filter

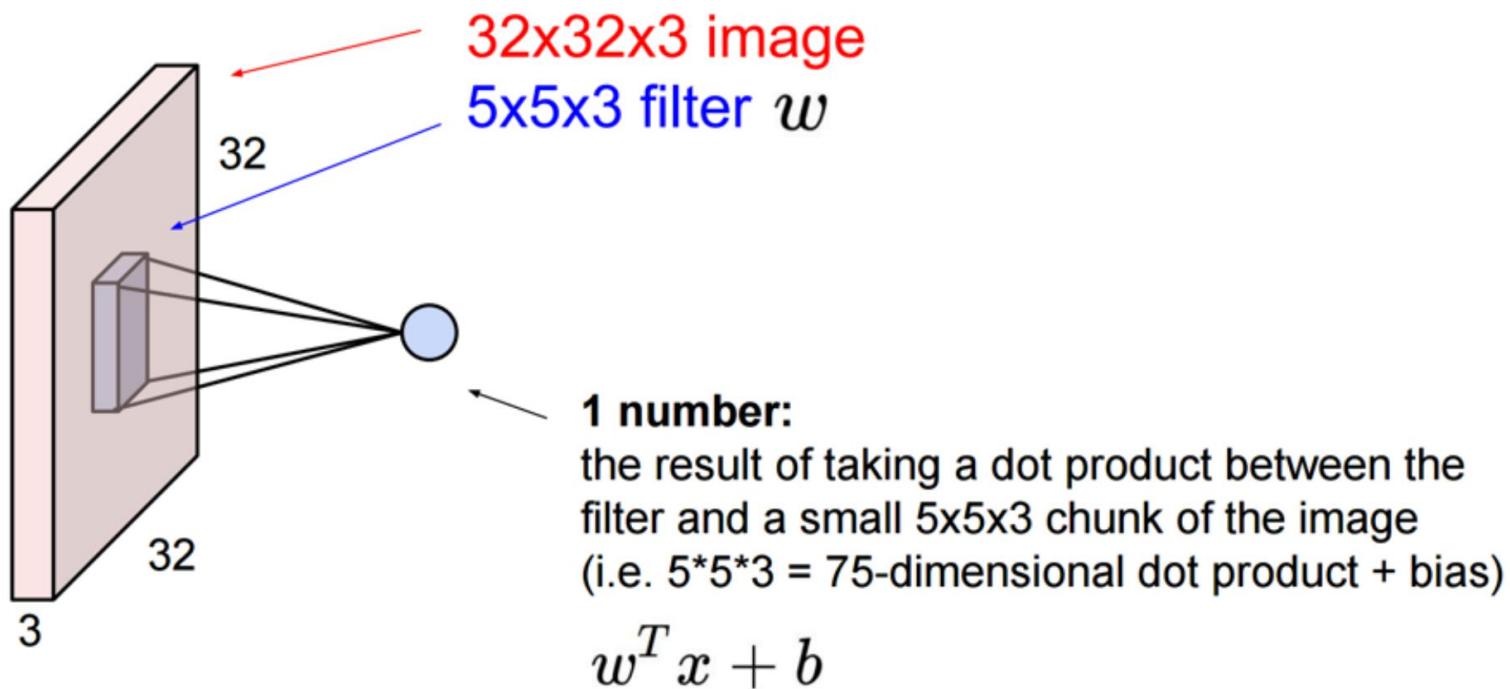


Filters always extend the full
depth of the input volume

Convolve the filter with the image
i.e. “slide over the image spatially,
computing dot products”

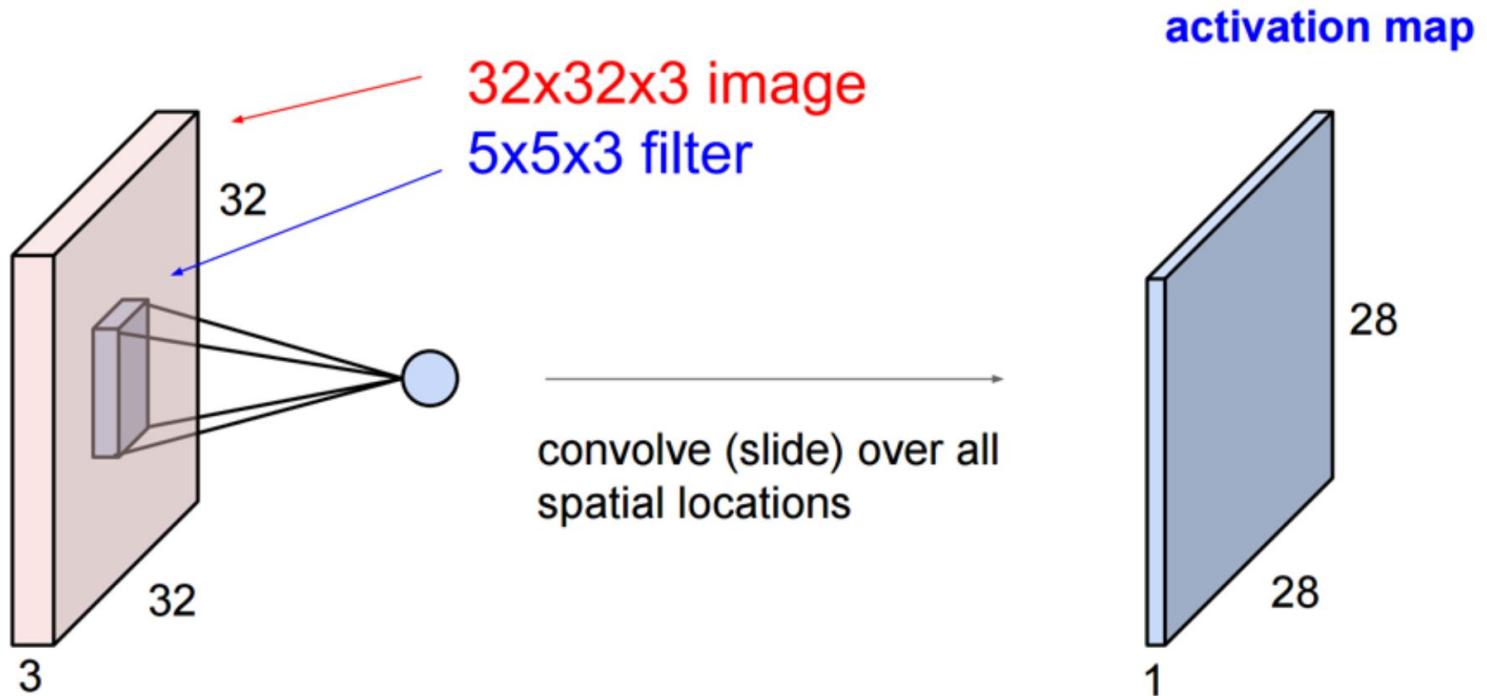
Putting it together

Convolution Layer



Putting it together

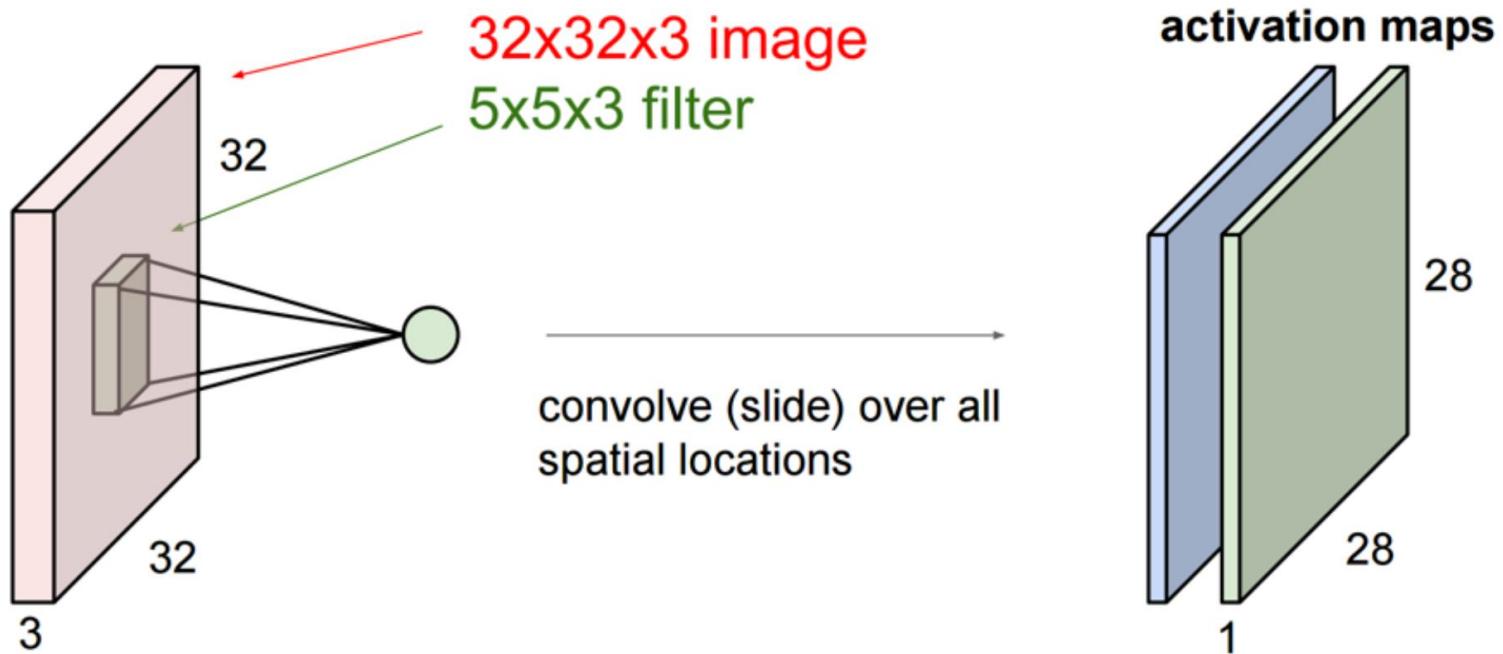
Convolution Layer



Putting it together

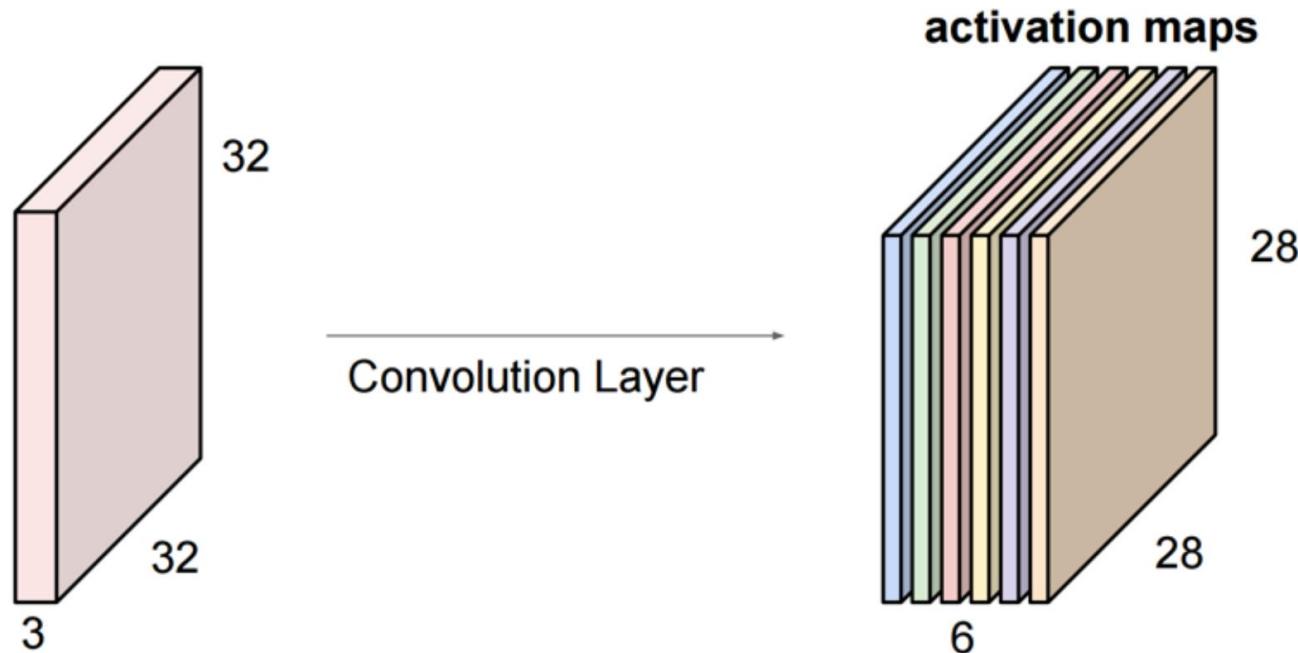
Convolution Layer

consider a second, green filter



Putting it together

For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



We stack these up to get a “new image” of size 28x28x6!

Break

Pooling

For most ConvNets, **convolution** is often followed by **pooling**:

- Creates a smaller representation while retaining the most important information
- The “max” operation is the most common
- Why might “avg” be a poor choice?

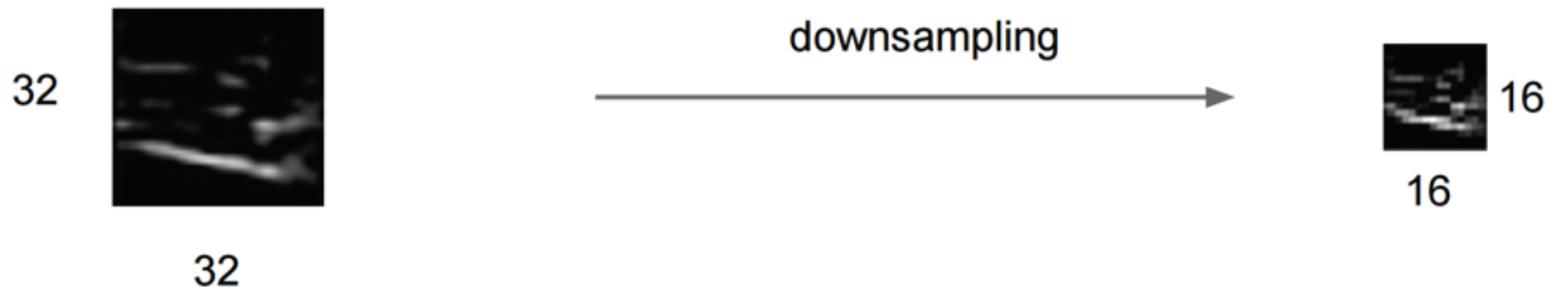
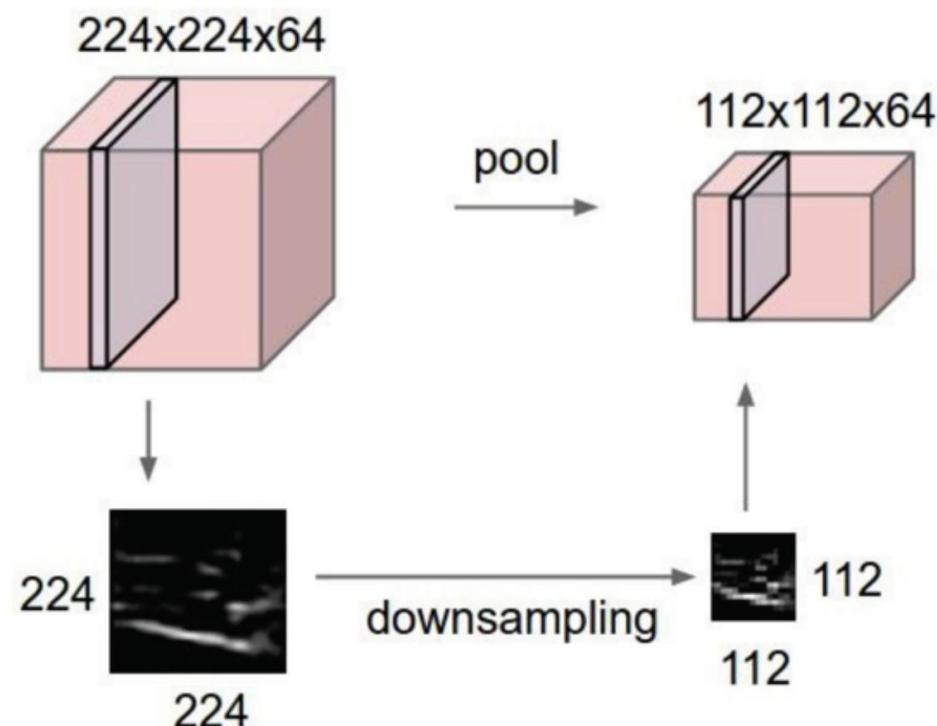


Figure: Andrej Karpathy

Pooling

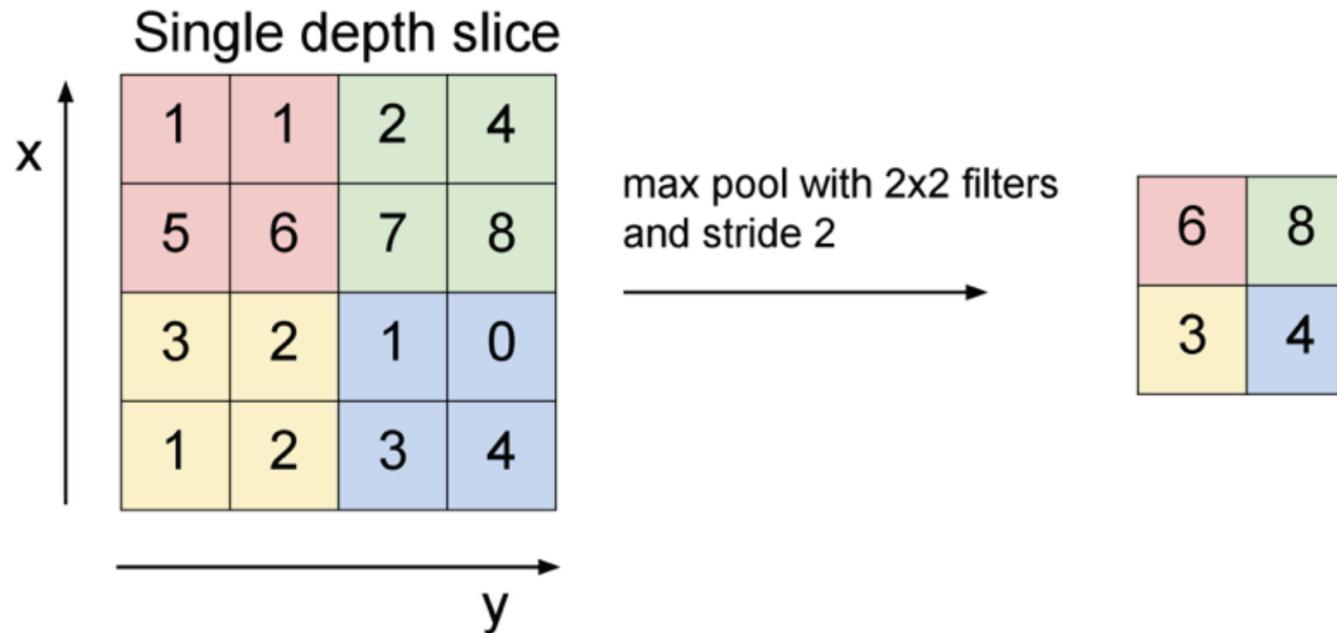
- makes the representations smaller and more manageable
- operates over each activation map independently:



Max Pooling

Why max pooling and not average?

We want to keep the information where the activation is higher



What's the backprop rule for max pooling?

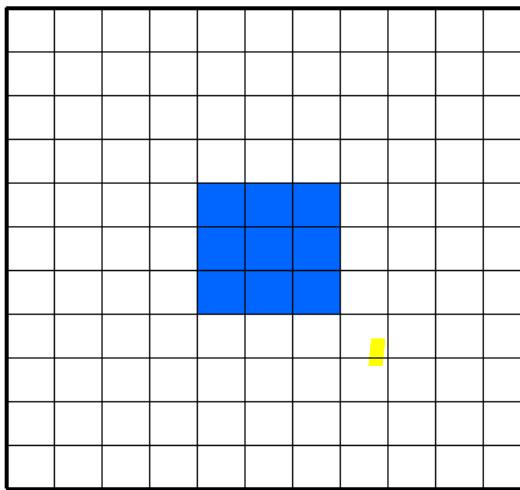
- In the forward pass, store the index that took the max
- The backprop gradient is the input gradient at that index

Figure: Andrej Karpathy

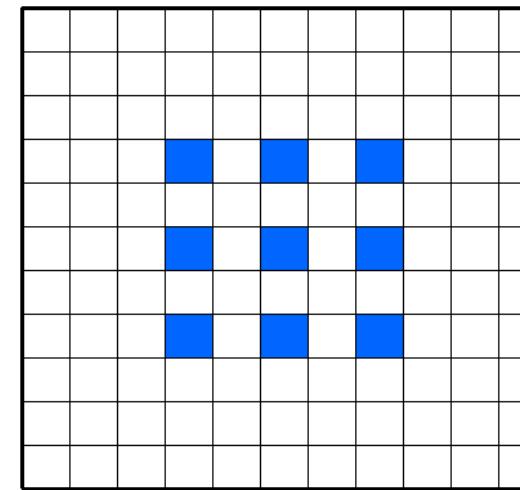
Dilated convolutions

Enlarge receptive field without increasing parameter count.

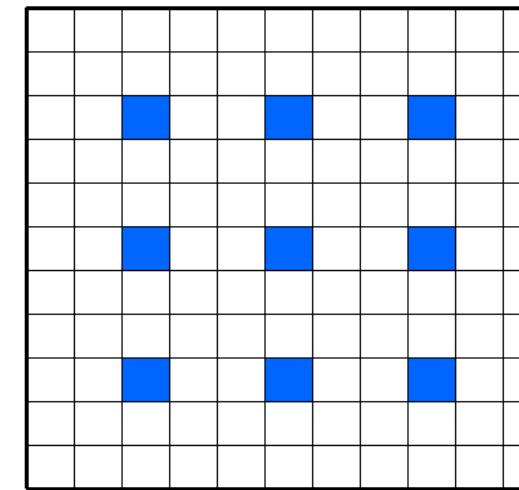
D = 1



D = 2



D = 3



1x1 convolutions

Generally, the goal of a convolutional layer is to learn local spatial filters.

Special case: 1x1 convolutions.

Example: input is [32x32x3], then a 1x1 convolution is a 3-D dot product.

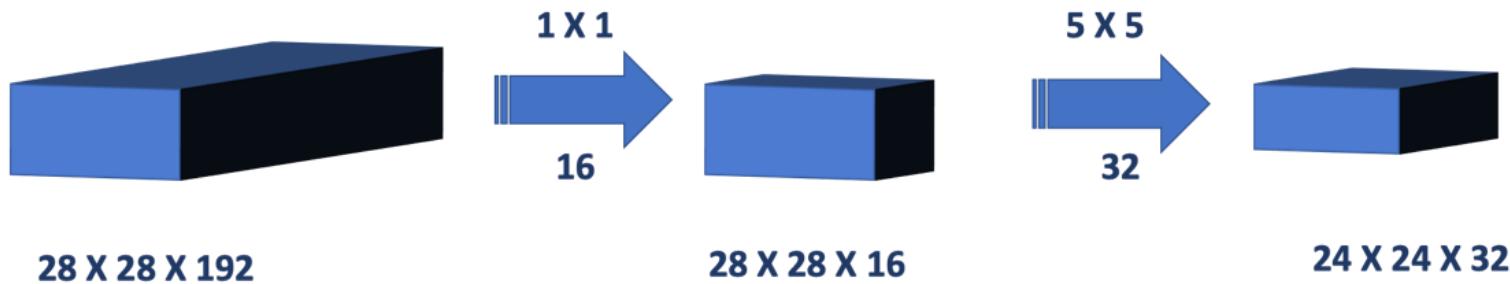
1x1 convolutions learn to mix information across channels.

1x1 convolutions

to reduce the number of channels I could do convolution with a fewer number of channels (depth) but this cost me a lot of operation and weight parameters



$$\text{Number of Operations} : (28 \times 28 \times 32) \times (5 \times 5 \times 192) = 120.422 \text{ Million Ops}$$



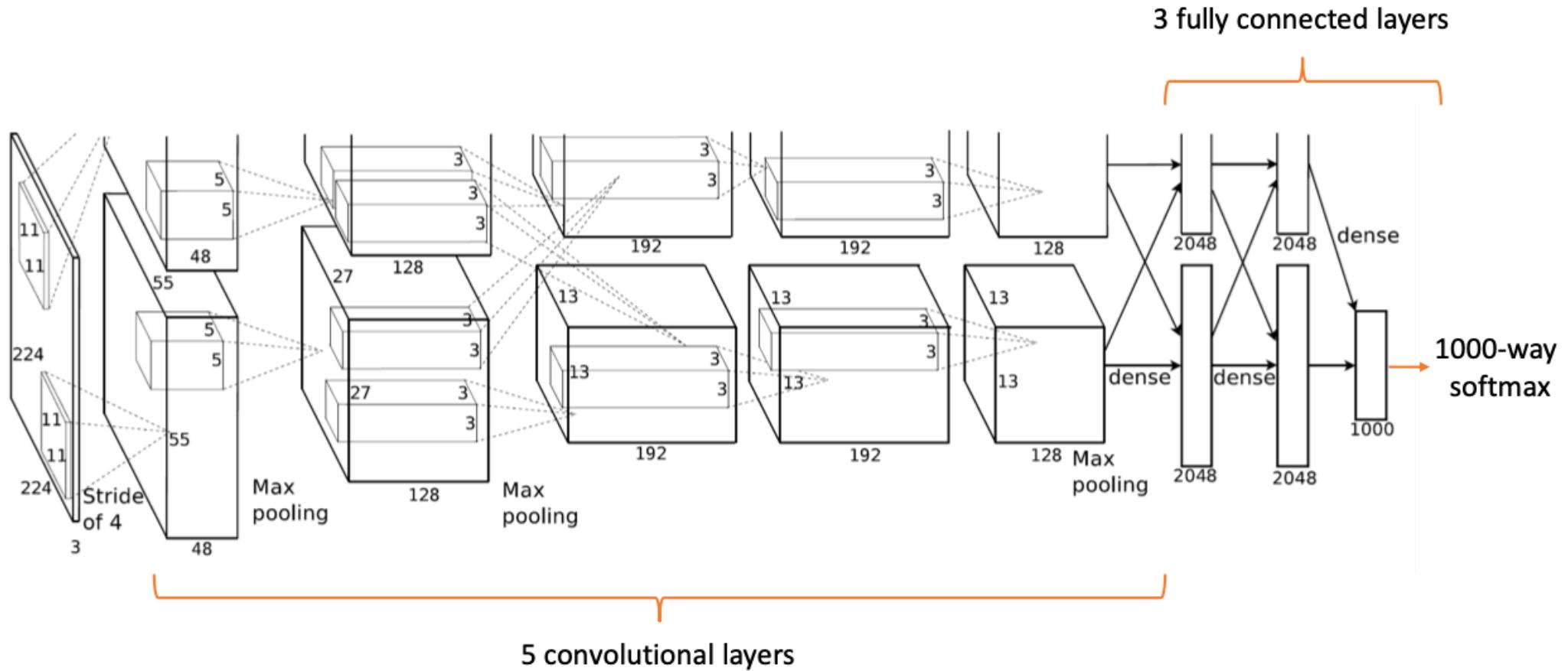
$$\text{Number of Operations for } 1 \times 1 \text{ Conv Step} : (28 \times 28 \times 16) \times (1 \times 1 \times 192) = 2.4 \text{ Million Ops}$$

$$\text{Number of Operations for } 5 \times 5 \text{ Conv Step} : (28 \times 28 \times 32) \times (5 \times 5 \times 16) = 10 \text{ Million Ops}$$

$$\text{Total Number of Operations} = 12.4 \text{ Million Ops}$$

A more practical approach is to use a linear node for each of the input channel I want and use it to reduce the dimension with much less parameters

AlexNet

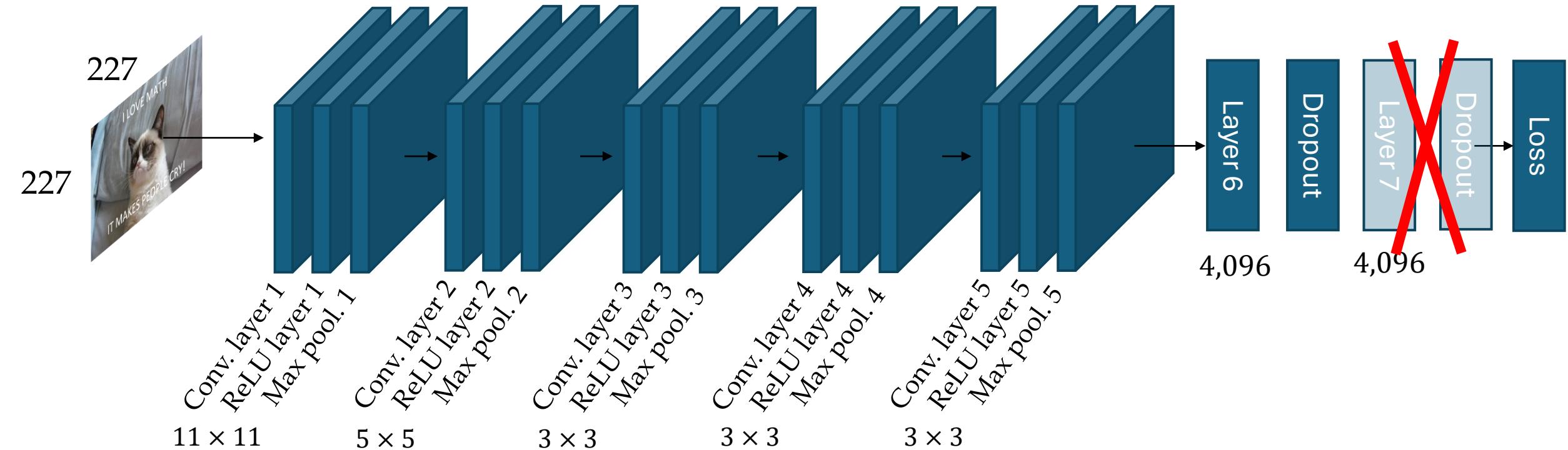


Where are most of the parameters?

Almost all the parameters are in the fully connected layer

Removing parameters

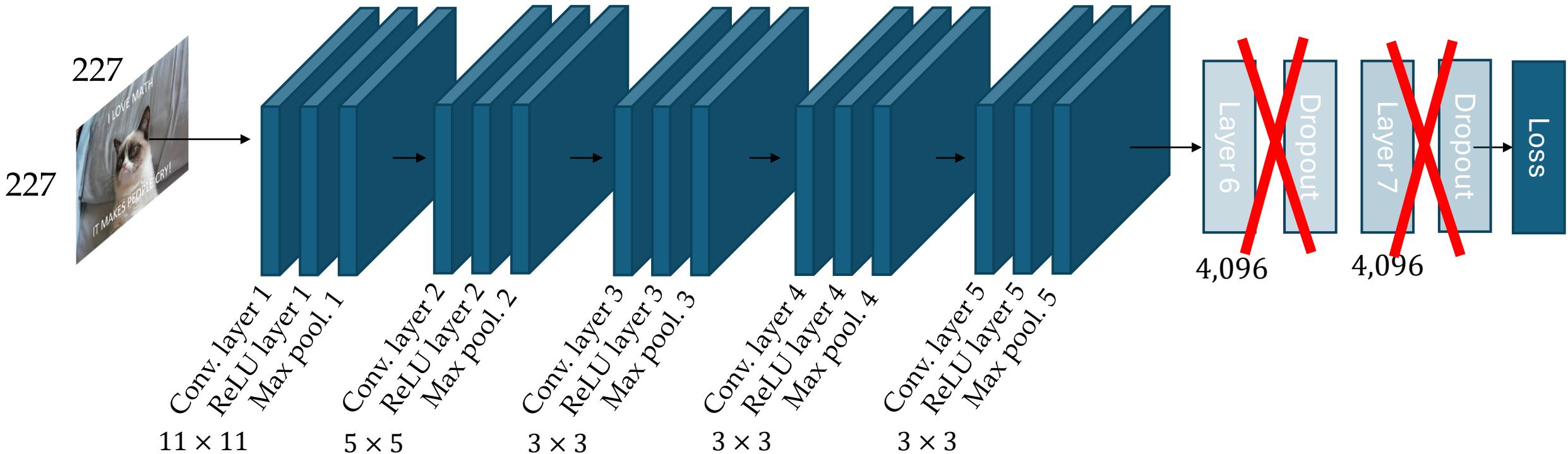
1.1% drop in performance, 16 million less parameters



Removing parameters

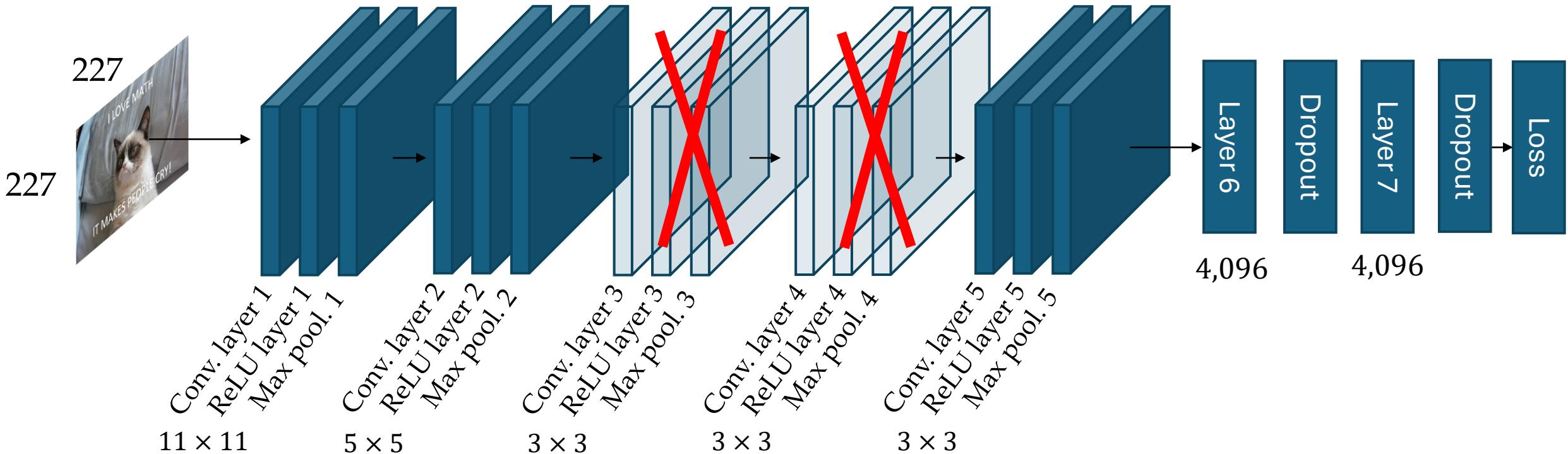
I drop most of my parameters but I perform almost the same

5.7% drop in performance, 50 million less parameters

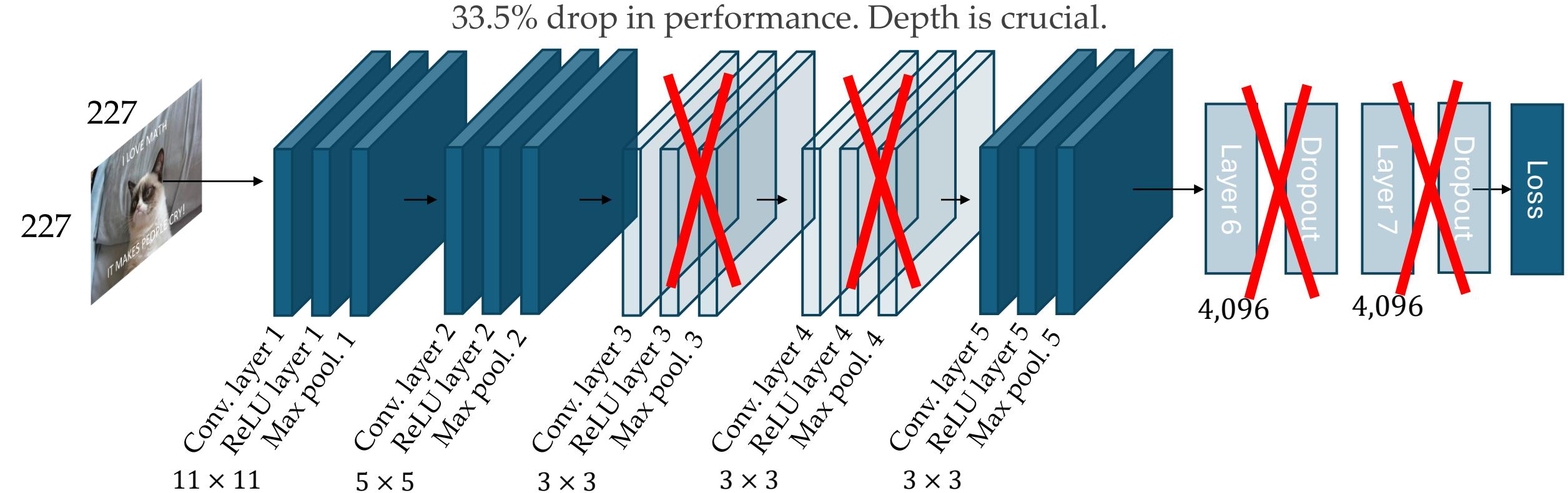


Removing parameters

3.0% drop in performance, 1 million less parameters. Why?



Removing parameters



Key ingredient for deeper networks: residuals

these elements are everywhere and rule out most of deep learning.

Let's say we have the neural network nonlinearity $a = F(x)$.

maybe it is easier to learn the difference instead of the transformation

Perhaps easier to learn a function $a = F(x)$ to model differences $a \sim \delta y$ than to model absolutes $a \sim y$.

Otherwise, you may need to model the magnitude and the direction of activations

Think of it like in input normalization → you normalize around 0

Think of it like in regression → you model differences around the mean value

“Residual idea”: Let neural networks explicitly model difference mappings

$$F(x) = H(x) - x \Rightarrow H(x) = F(x) + x$$

$F(x)$ are the stacked nonlinearities

x is the input to the nonlinear layer.

to add the input, the output must be of the same size. So we usually apply it after two conv layers so that we can transform the output of the first back into the dimension of its input, in order to be able to sum it afterwards

The residual block

They're useful because solves the problem of vanishing gradients, when I do backprop the derivative of $H(x)$ wrt x will be the der of F plus 1, this one avoid the gradient to become too small

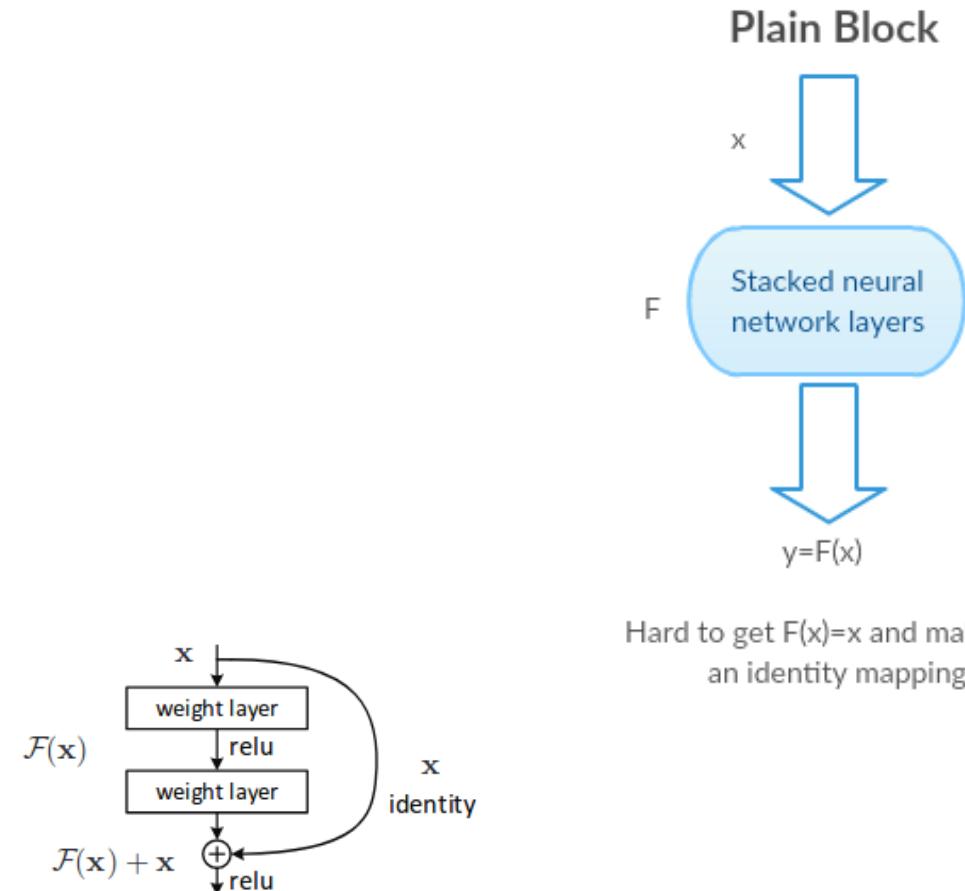
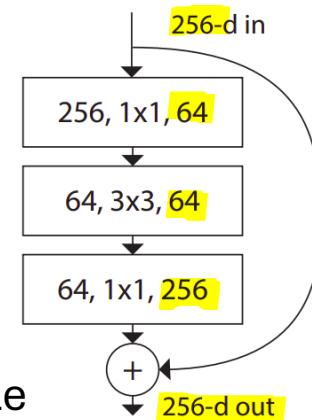
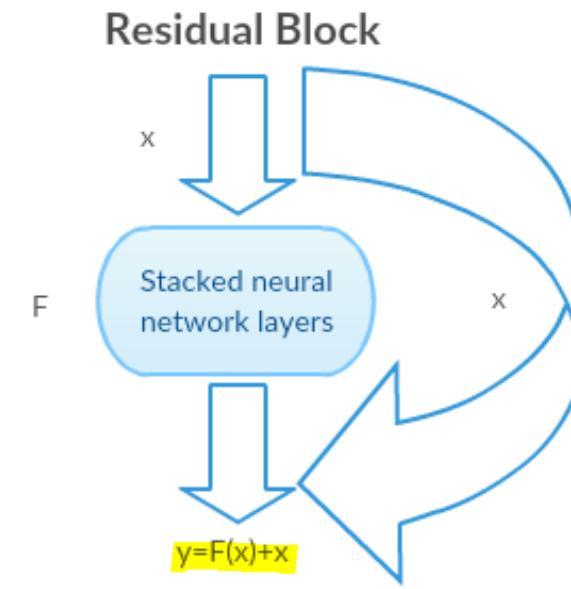


Figure 2. Residual learning: a building block.



Also it has been shown that since the residual block instead of learning a complete transformation $H(x)$, layers learn the residual function $F(x)=H(x)-x$. Learning small adjustments or residuals is considerably easier than learning the full mapping, particularly as networks grow deeper. This reformulation of the learning problem reduces the complexity of the optimization landscape, allowing gradient-based algorithms to converge more rapidly and find better solutions with fewer training iterations.

Performance degradation gone

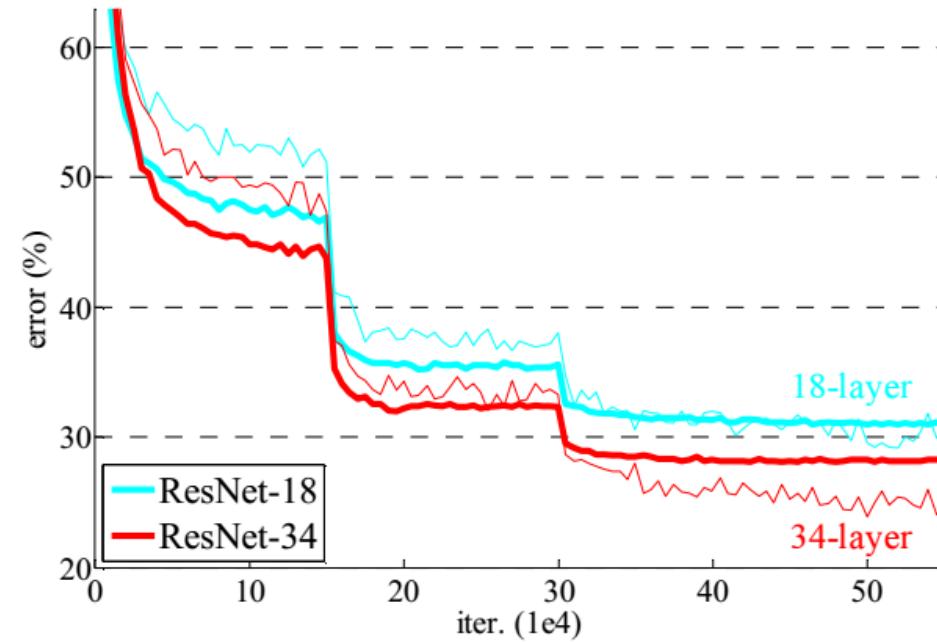
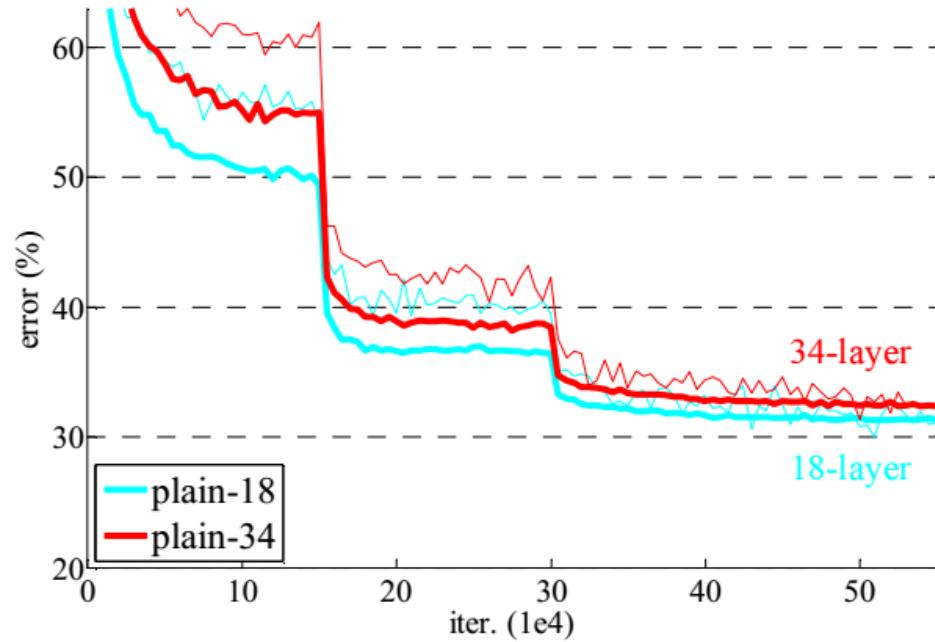
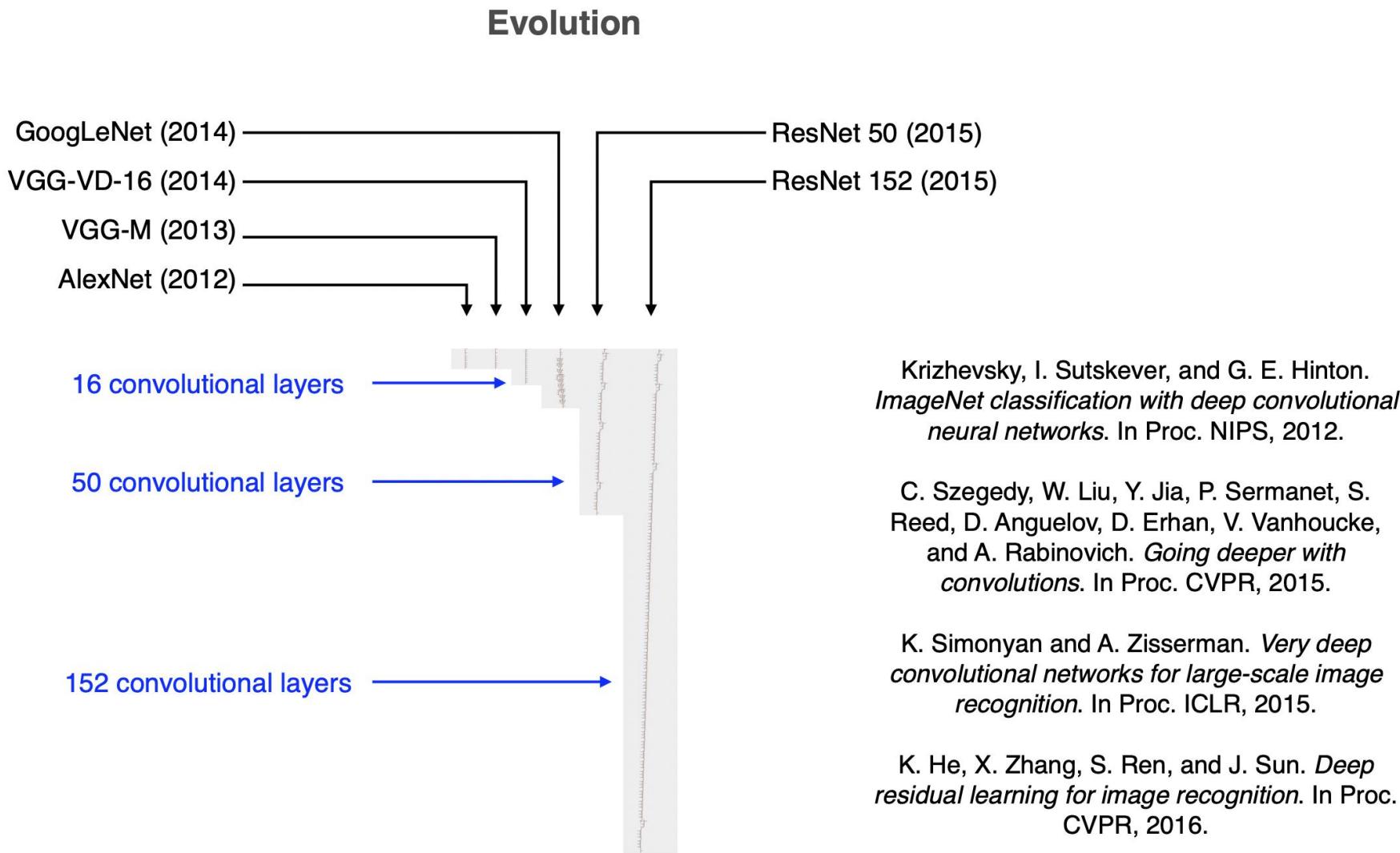


Figure 4. Training on **ImageNet**. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

ResNet is a record-breaker: (1) depth



ResNet is a record-breaker: (2) performance

Very low ImageNet errors with networks that can grow to 1000 layers.

method	top-5 err. (test)
VGG [41] (ILSVRC'14)	7.32
GoogLeNet [44] (ILSVRC'14)	6.66
VGG [41] (v5)	6.8
PReLU-net [13]	4.94
BN-inception [16]	4.82
ResNet (ILSVRC'15)	3.57

Table 5. Error rates (%) of **ensembles**. The top-5 error is on the test set of ImageNet and reported by the test server.

ResNet is a record-breaker: (3) influence

DL1 24/25:

[Deep residual learning for image recognition](#)

[K He, X Zhang, S Ren, J Sun - Proceedings of the IEEE ...](#), 2016 - openaccess.thecvf.com

... as learning **residual** functions with ... **residual networks** are easier to optimize, and can gain accuracy from considerably increased depth. On the ImageNet dataset we evaluate **residual** ...

[☆ Save](#) [⤒ Cite](#) [Cited by 242319](#) [Related articles](#) [All 65 versions](#) [⤓](#)

DL1 25/26:

[Deep residual learning for image recognition](#)

[K He, X Zhang, S Ren, J Sun - ... and pattern recognition](#), 2016 - openaccess.thecvf.com

... **Deeper** neural **networks** are more difficult to train. We present a **residual learning** framework to ease the training of **networks** that are substantially **deeper** than those used previously. ...

[☆ Save](#) [⤒ Cite](#) [Cited by 290057](#) [Related articles](#) [All 53 versions](#) [⤓](#)

Convolutional networks beyond classification

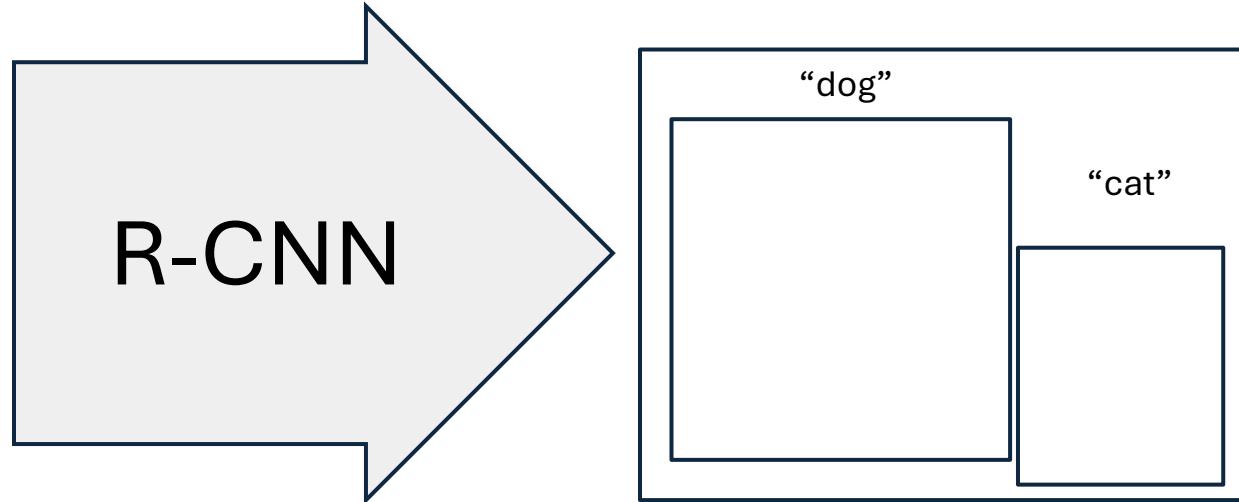
Detection, localization, and explanation

Object detection

Goal: correctly identify **where** the main objects are.

This task is called *object detection*.

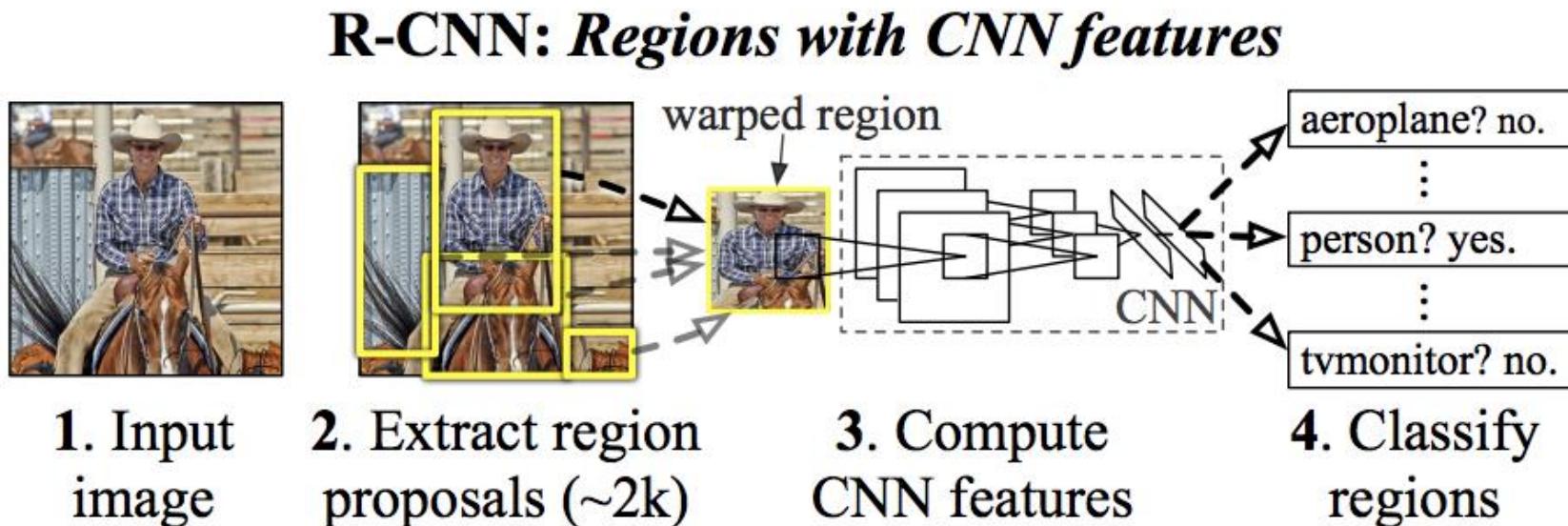
Output: bounding boxes with object classification scores.



Region-Based ConvNets (R-CNN) [2014]

Idea: each box in an image is its own image.

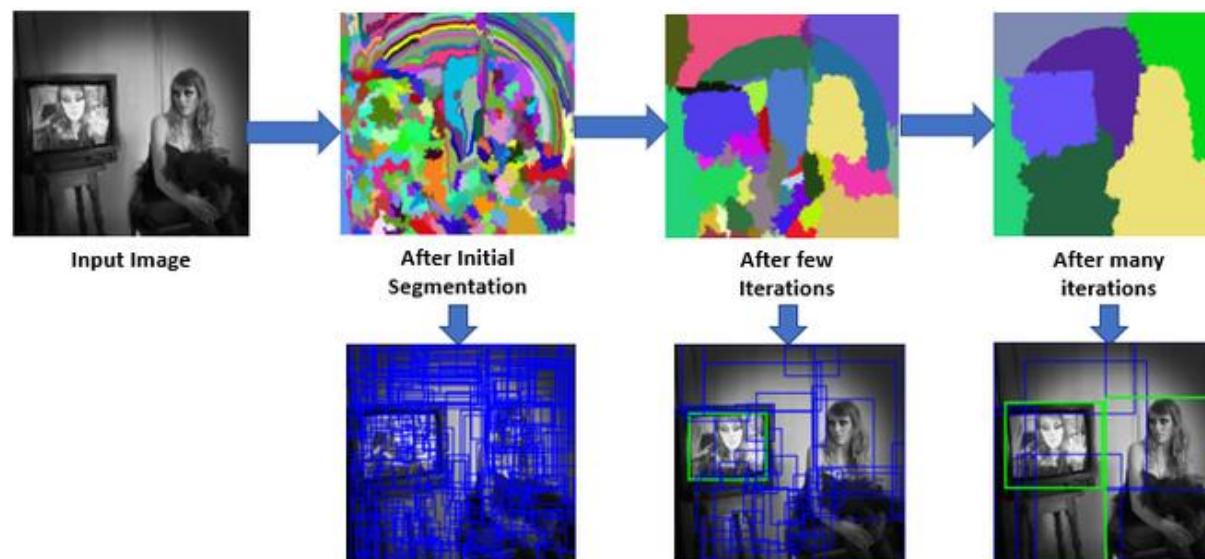
How: (1) find interesting boxes, (2) rescale each box and feed through a CNN.



Extracting region proposals

An idea from before the deep learning era. R-CNN uses an algorithm designed by the UvA: selective search.

Selective search: start from superpixels, group one-by-one. Each new grouping is a new region proposal.



Extra step: improving the region locations

Now, having found the object in the box, can we tighten the box to fit the true dimensions of the object?

We can, and this is the final step of R-CNN.

R-CNN runs a simple linear regression on the region proposal to generate tighter bounding box coordinates to get the final result.

- Inputs: sub-regions of the image corresponding to objects.
- Outputs: New bounding box coordinates for the object in the sub-region.

R-CNN summarized

R-CNN is just the following steps:

1. Generate a set of proposals for bounding boxes.
2. Run the images in the bounding boxes through a pre-trained AlexNet and finally an SVM to see what object the image in the box is.
3. Run the box through a linear regression model to output tighter coordinates for the box once the object has been classified.

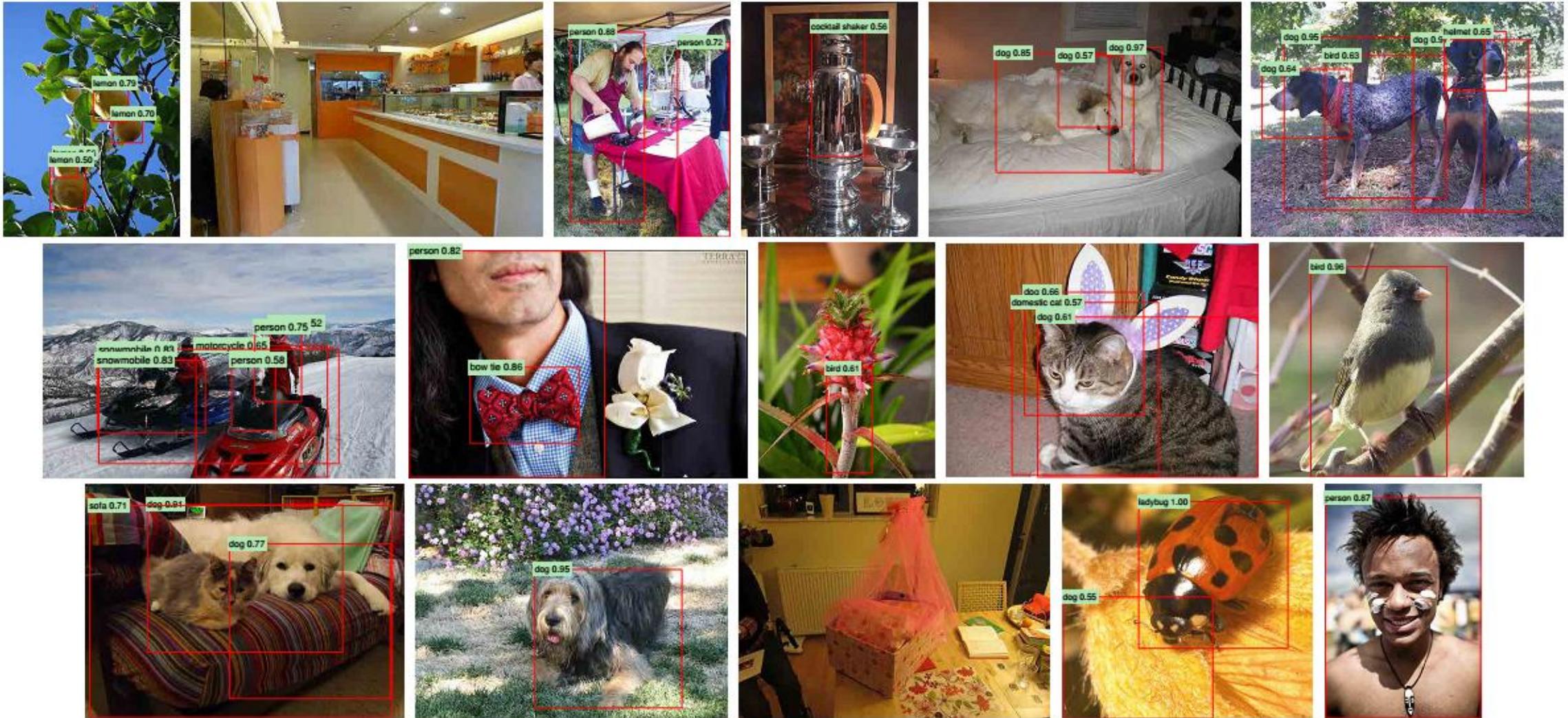
the final part of the 2 and the 3 actually happens in parallel

After the CNN extracts features from each warped region proposal and the SVM classifier determines the object class, the initial bounding boxes from selective search are often imprecise and don't tightly fit the actual objects. The linear regression model addresses this localization error by learning to predict correction offsets that adjust the bounding box coordinates.

The regression model takes the extracted CNN features from each region proposal as input and learns to predict four values that represent offsets for the bounding box coordinates. These offsets are trained to transform the original region proposal coordinates into the ground-truth bounding box coordinates. Specifically, the model is trained using the extracted features and labeled bounding boxes of each region proposal as training examples

For each object class detected, the regression predicts adjustments to the position and size of the bounding box relative to the original region proposal. This allows the final output to have much tighter and more accurate bounding boxes than what selective search initially provided, improving the mean average precision by approximately 3-4%.

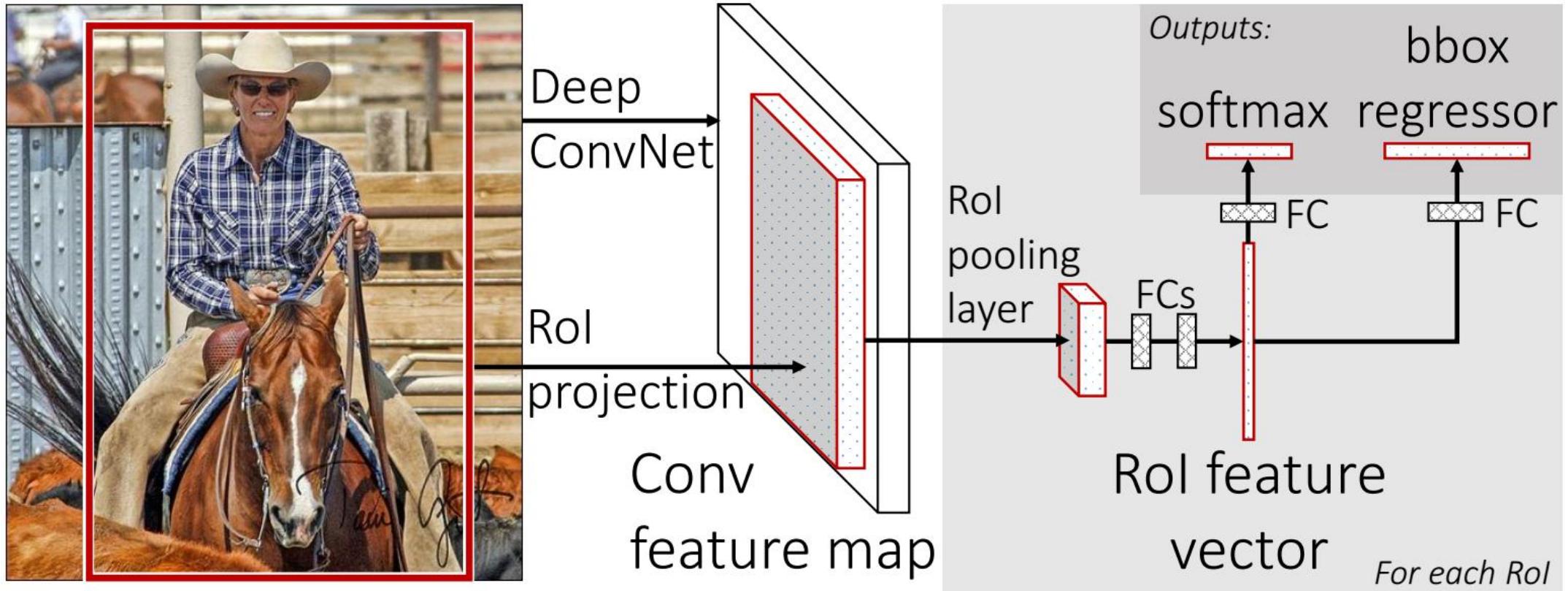
R-CNN results



What are the two major downsides to R-CNN?

1. Each region proposal requires a new pass through the network.
2. Region proposals need to be determined a priori.

Fast R-CNN [2015]



Key to Fast R-CNN: RIOPooling

For the forward pass of the CNN, a lot of proposed regions for the image invariably overlapped ...

... causing us to run the same CNN computation again and again (~2000 times!).

Why not run the CNN just once per image and then find a way to share that computation across the ~2000 proposals?

Region of Interest pooling

RoIPool shares the forward pass of a CNN for an image across its subregions.



In the image above, notice how the CNN features for each region are obtained by selecting a corresponding region from the CNN's feature map.

Then, the features in each region are pooled (usually using max pooling). So, all it takes us is one pass of the original image as opposed to ~2000!

Now region proposal are extracted use Selective search (or other onCPU algorith,) directly from the feature map.

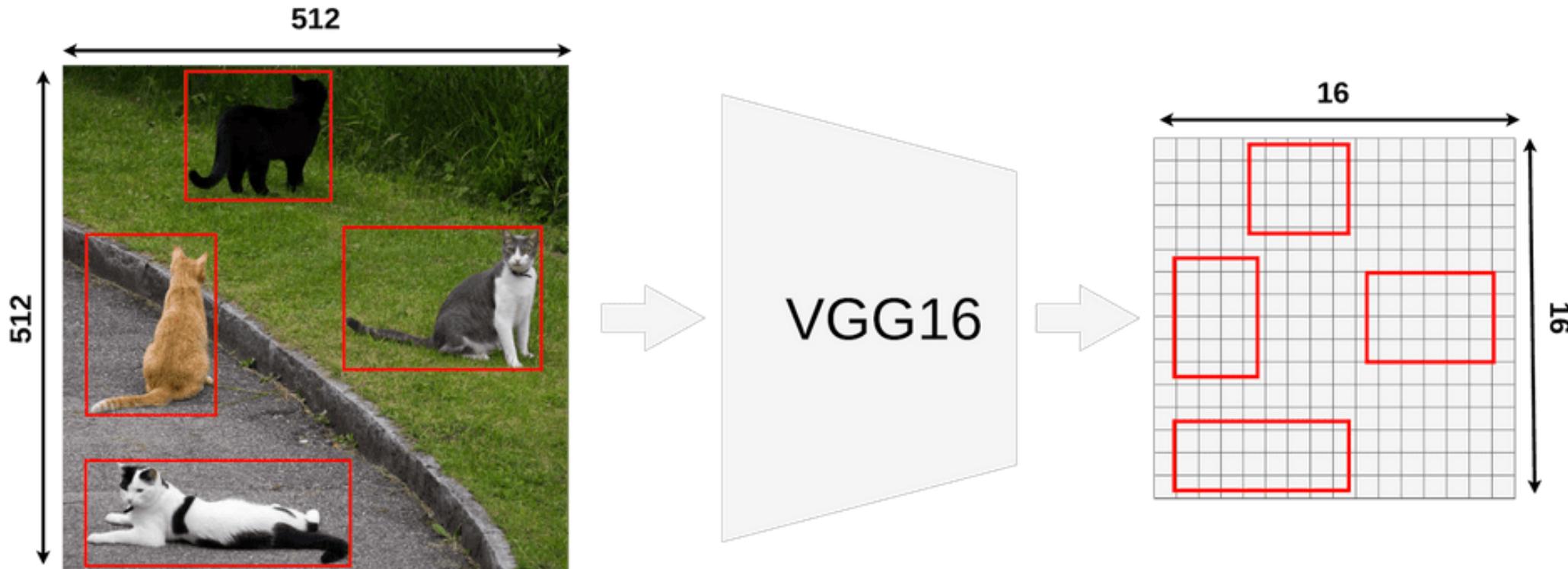
Roi pooling works by dividing the proposal region on the feature map into a grid and using max pooling within each grid cell. This produces a fixed-size feature map from each proposal, regardless of its original size. The output is then flattened into a feature vector.

Each feature vector is passed through fully connected layers, which split into two branches:

Classification branch: Predicts class scores (via softmax) for each proposal.

Bounding box regression branch: Predicts position and size offsets to refine the bounding box, making it fit the object better.

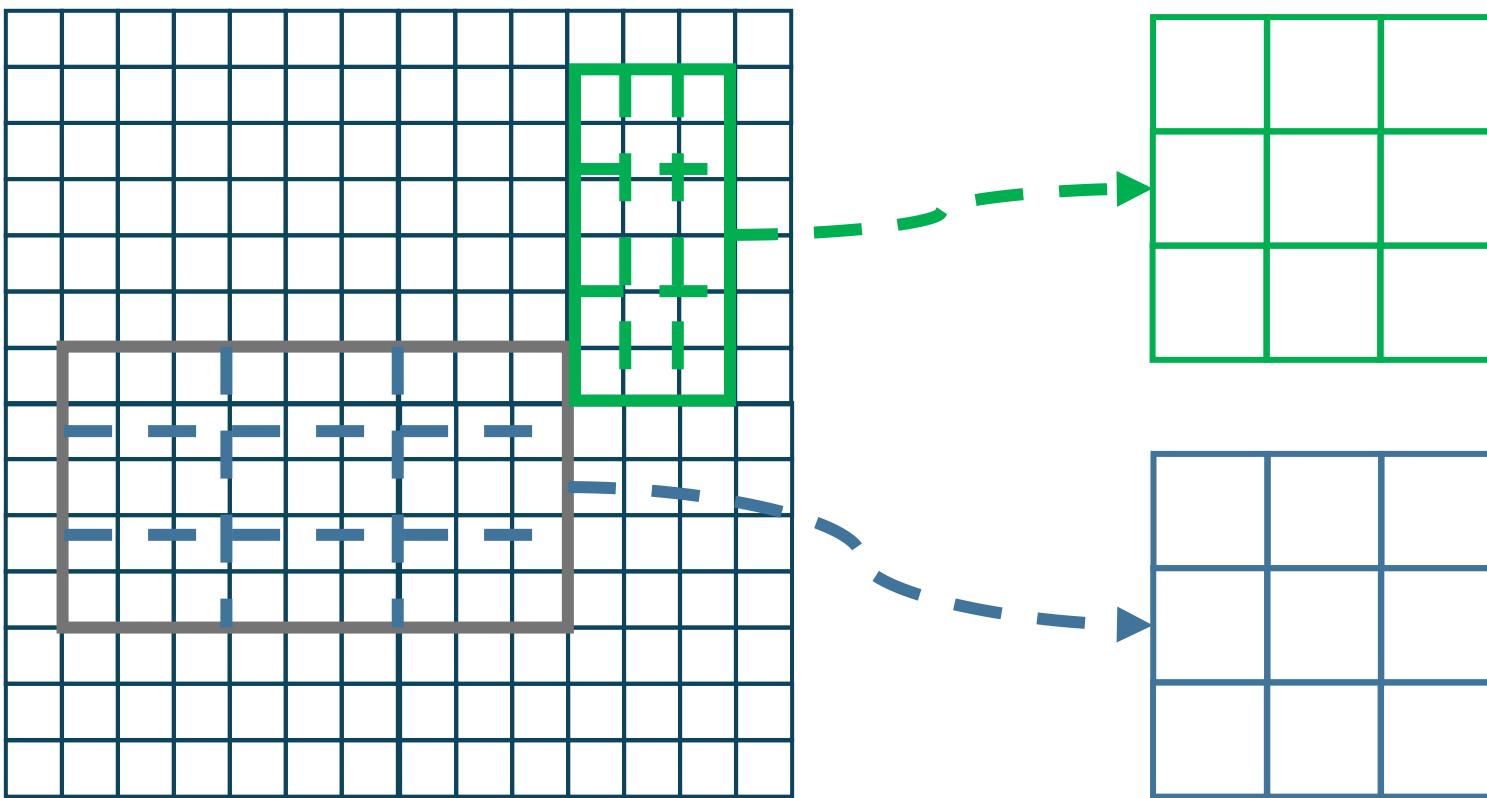
Region of Interest pooling



Region of Interest pooling

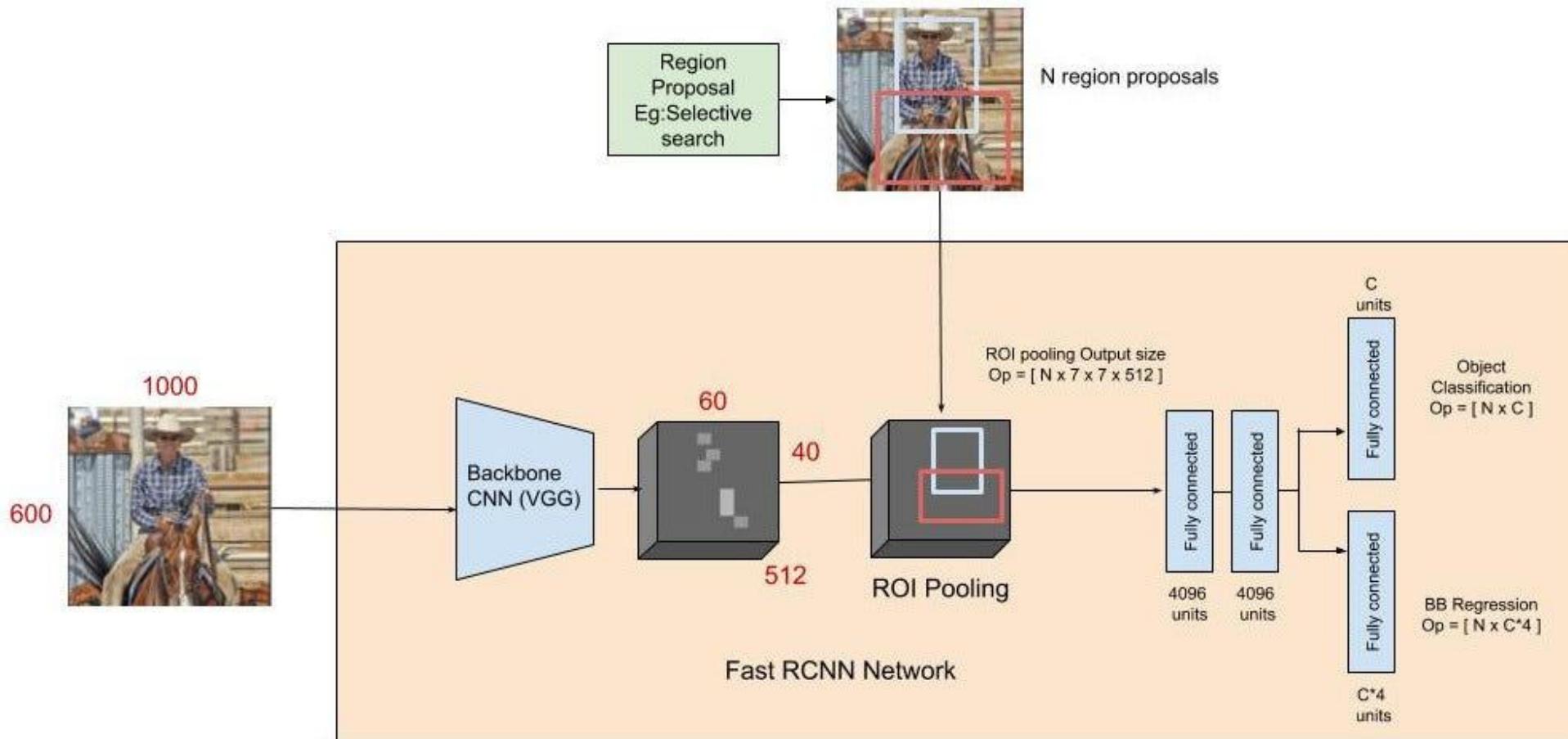
Divide feature map in $T \times T$ cells.

The cell size will change depending on the size of the candidate location.



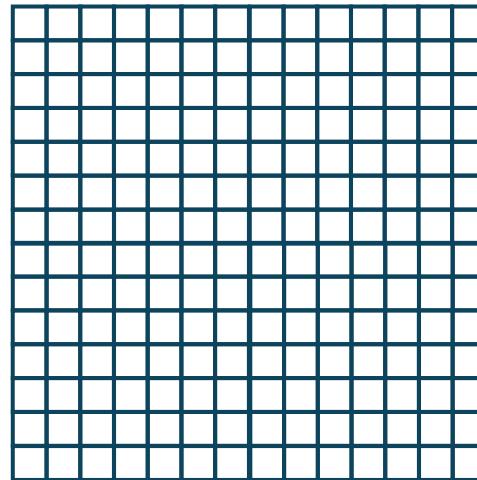
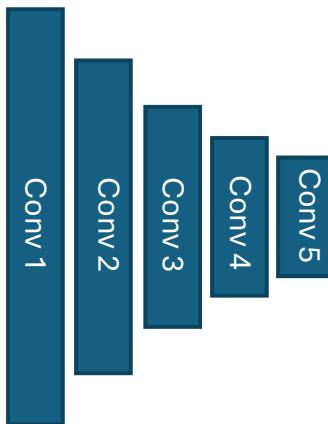
Always 3x3 no
matter the size of
candidate location

Second key to Fast R-CNN: joint training



Fast R-CNN: step-by-step

Process the whole image up to conv5.

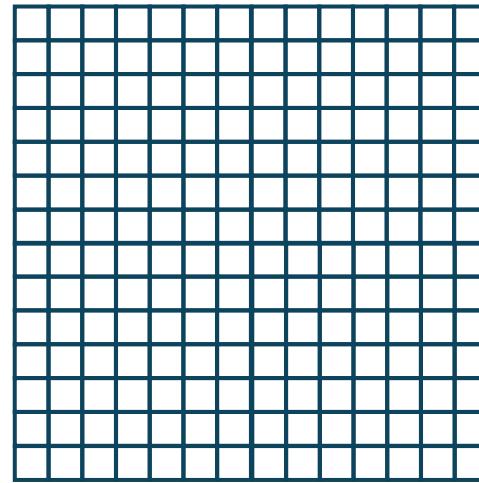
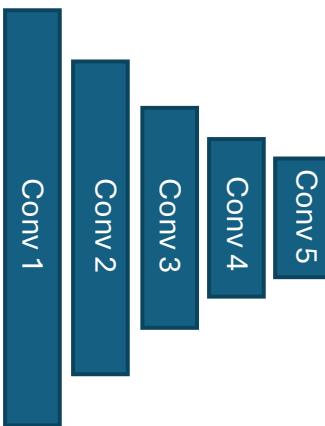


Conv 5 feature map

Fast R-CNN: step-by-step

Process the whole image up to conv5.

Compute possible locations for objects.

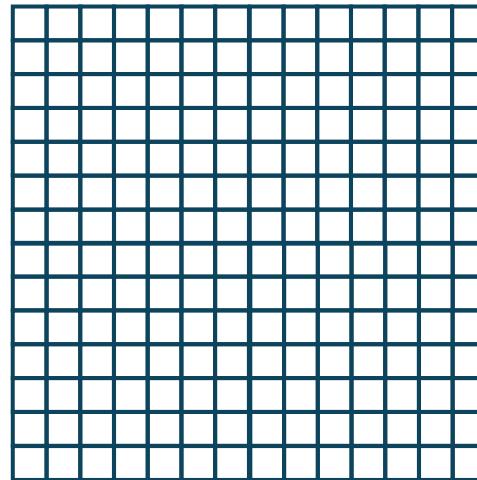
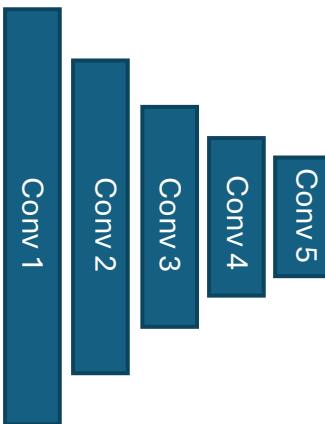


Conv 5 feature map

Fast R-CNN: step-by-step

Process the whole image up to conv5.

Compute possible locations for objects (some correct, most wrong).



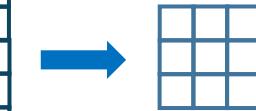
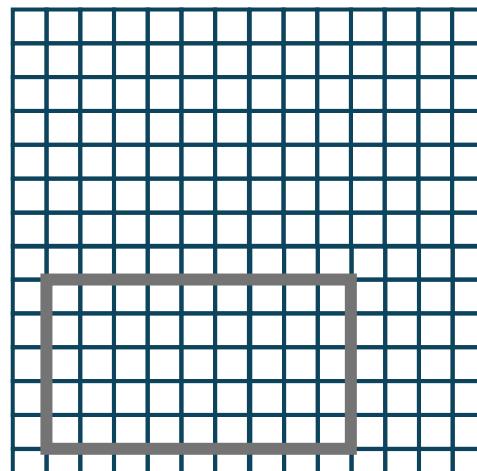
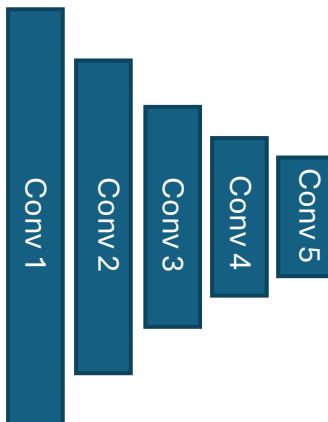
Conv 5 feature map

Fast R-CNN: step-by-step

Process the whole image up to conv5.

Compute possible locations for objects.

Given single location → ROI pooling module extracts fixed length feature.



- Always 3x3 no matter the size of candidate location

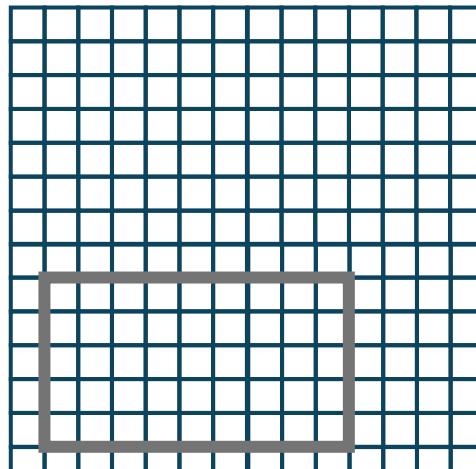
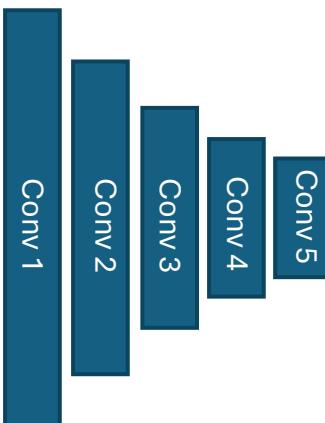
Fast R-CNN: step-by-step

Process the whole image up to conv5.

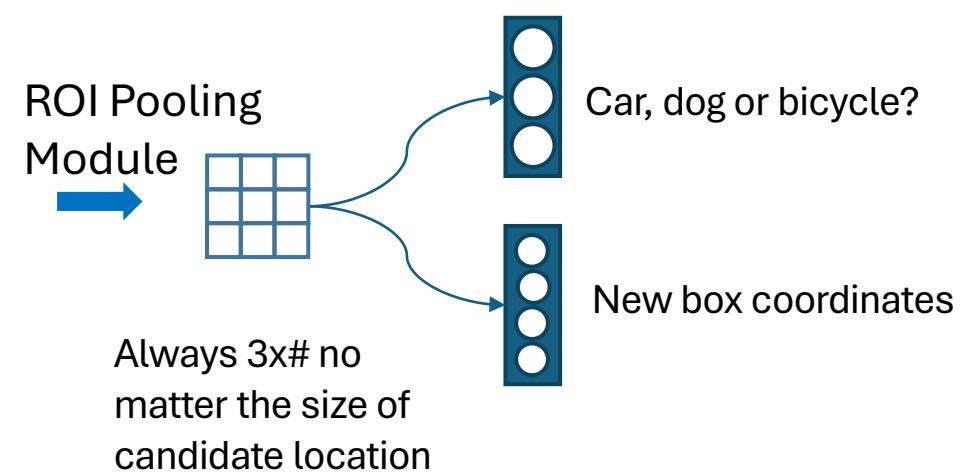
Compute possible locations for objects.

Given single location → ROI pooling module extracts fixed length feature.

Connect to two final layers, 1 for classification, 1 for box refinement.



Conv 5 feature map



Pros and cons of Fast R-CNN

Reuse convolutions for different candidate boxes

Compute feature maps only once

Region-of-Interest pooling

Define stride relatively → box width divided by predefined number of “poolings” T

Fixed length vector

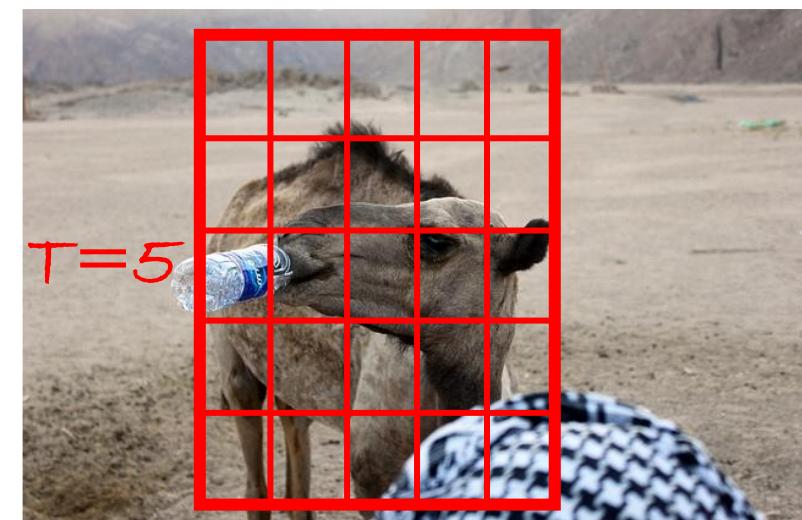
“End-to-end” training!

(Very) Accurate object detection

(Very) Faster

- Less than a second per image

External box proposals needed



Faster R-CNN [2017]

Still a bottleneck: the region proposer.

First step: generating a bunch of potential bounding boxes/regions.

In Fast R-CNN, these proposals were created using Selective Search, a fairly slow process that was found to be the bottleneck of the overall process...

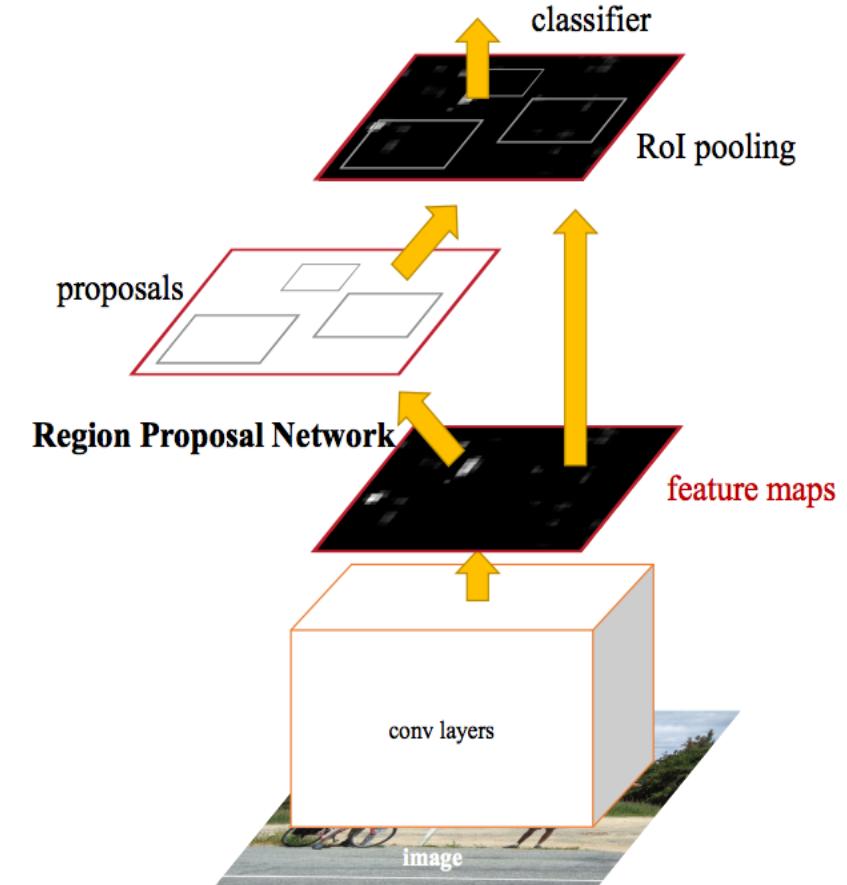
Faster R-CNN

Faster R-CNN adds a Fully Convolutional Network on top of the features of the CNN creating what's known as the Region Proposal Network.

The **Region Proposal Network** works by passing a sliding window over the CNN feature map and at each window, outputting k potential bounding boxes and scores for how good each of those boxes is expected to be.

In such a way, we create k such common aspect ratios we call anchor boxes. For each such anchor box, we output one bounding box and score per position in the image.

We then pass each such bounding box that is likely to be an object into Fast R-CNN to generate a classification and tightened bounding boxes.



Faster R-CNN

Fast R-CNN → external candidate locations.

Faster R-CNN → deep network proposes candidate locations.

Slide the feature map → k anchor boxes per slide.

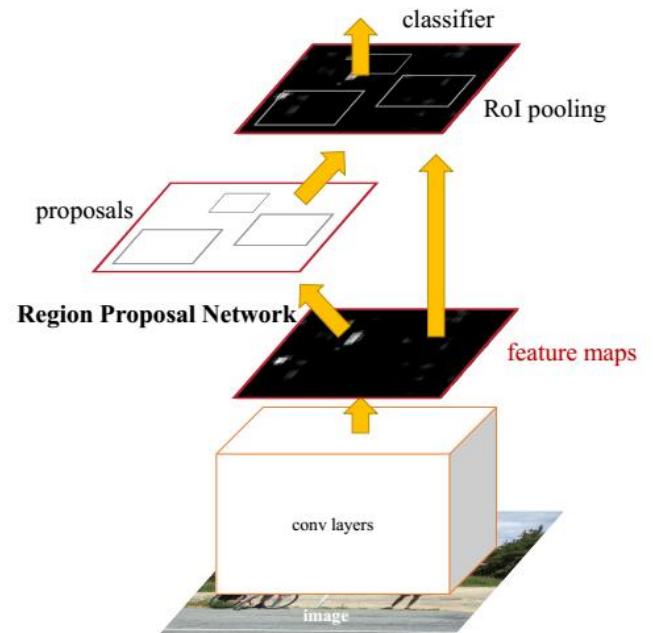
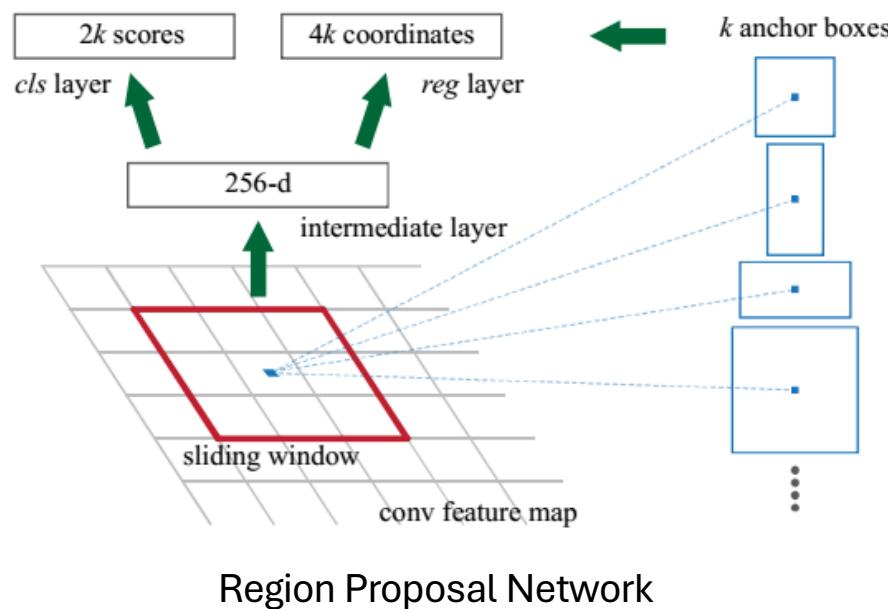


Figure 2: Faster R-CNN is a single, unified network for object detection. The RPN module serves as the ‘attention’ of this unified network.

Focal loss

Cross-entropy is a problem for object detection.

Many boxes evaluated, a lot of small losses lead to a huge bias.

Solution: add an exponent to the cross-entropy term!

In object detection, networks evaluate many bounding boxes (thousands), and most are negative examples (background, not containing objects). This creates two issues:

Many small losses from easy negative examples dominate the total loss

The model becomes biased toward predicting background, struggling to learn from rare positive examples

The exponent control how much the probability on x axis influence the loss

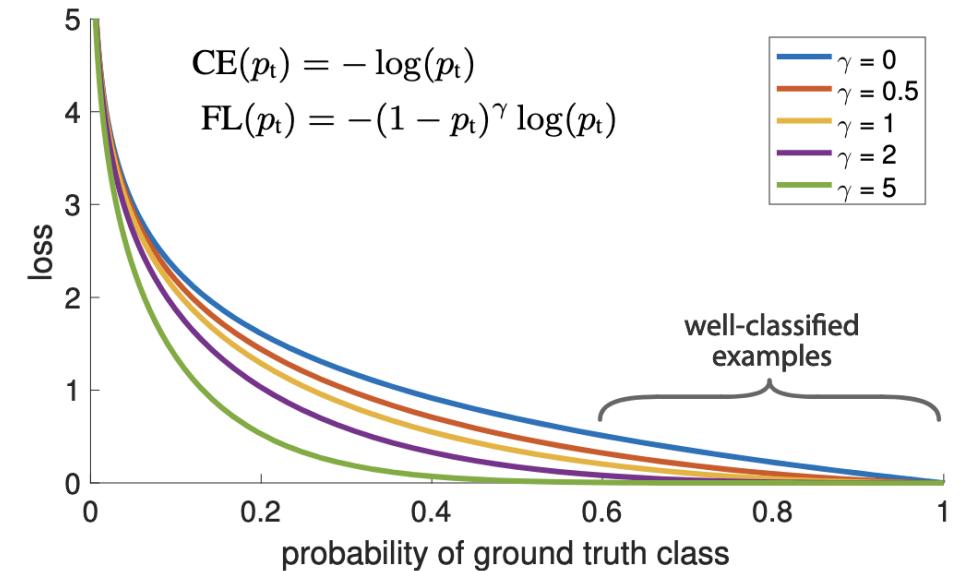


Figure 1. We propose a novel loss we term the *Focal Loss* that adds a factor $(1 - p_t)^\gamma$ to the standard cross entropy criterion. Setting $\gamma > 0$ reduces the relative loss for well-classified examples ($p_t > .5$), putting more focus on hard, misclassified examples. As our experiments will demonstrate, the proposed focal loss enables training highly accurate dense object detectors in the presence of vast numbers of easy background examples.

Mask R-CNN

Extending Faster R-CNN for Pixel Level Segmentation.

So far, we've seen how we've been able to use CNN features in many interesting ways to effectively locate different objects in an image with bounding boxes.

Can we extend such techniques to go one step further and locate exact pixels of each object instead of just bounding boxes?

This problem is known as *image segmentation*.

Mask R-CNN

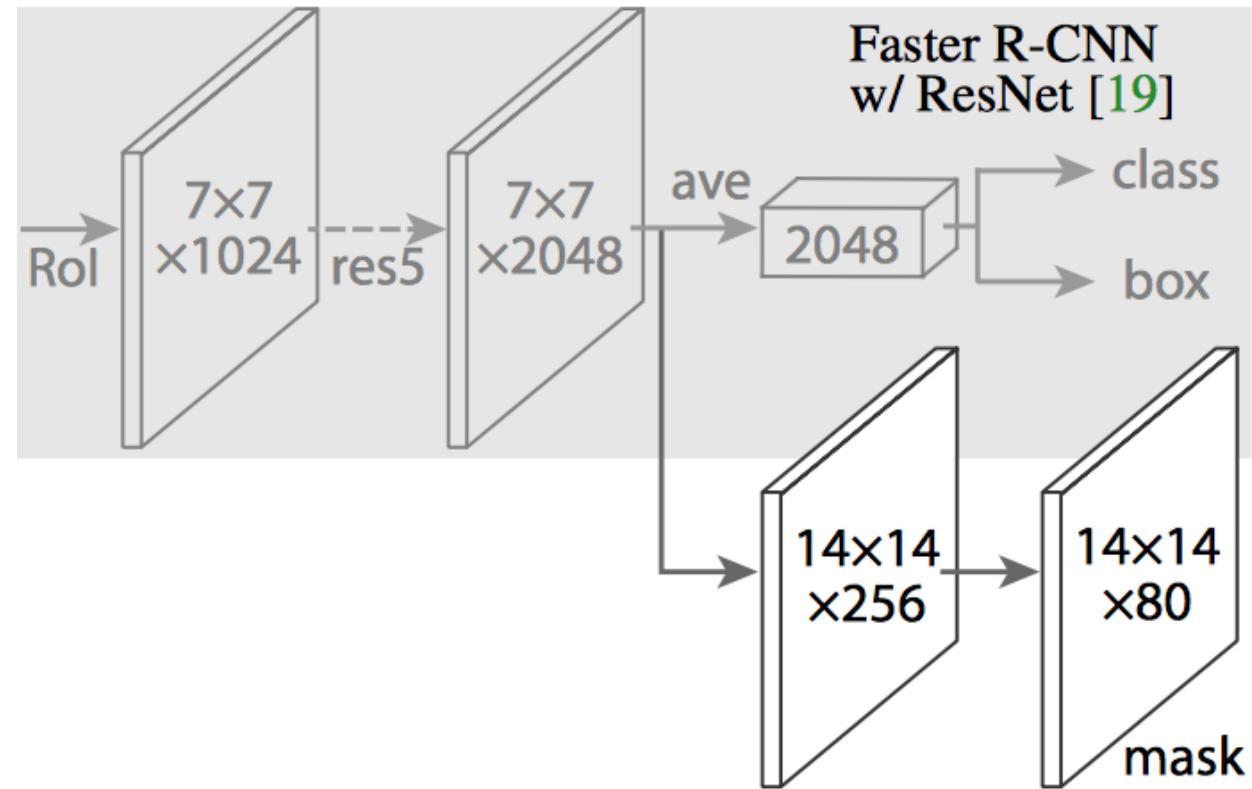
Given that Faster R-CNN works so well for object detection, could we extend it to also carry out pixel level segmentation?

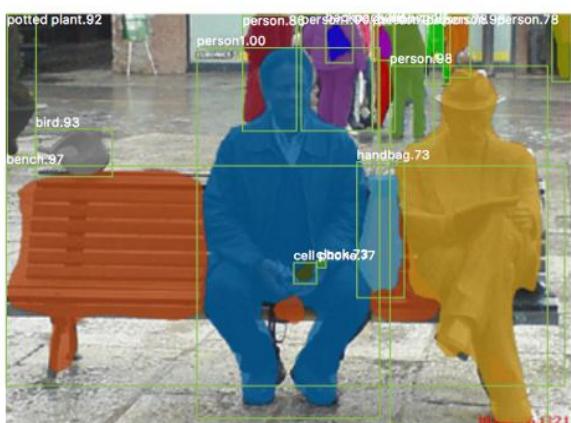
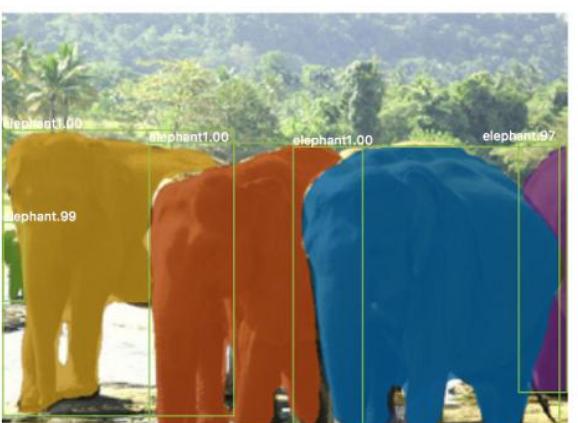
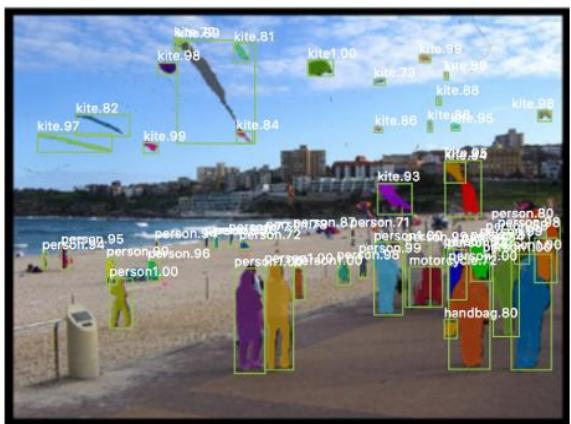
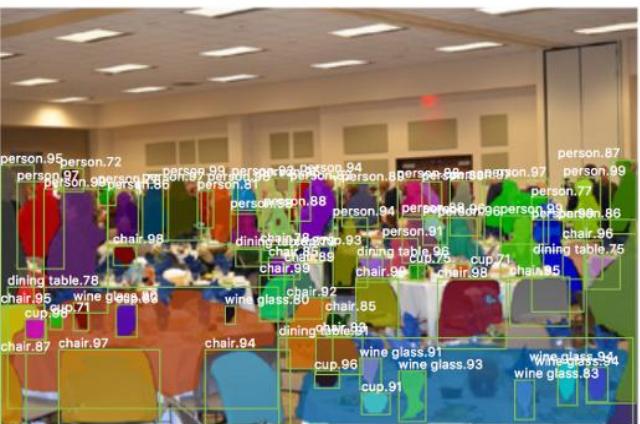
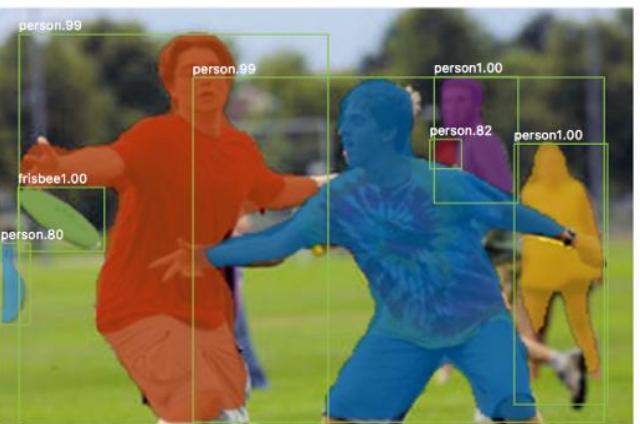
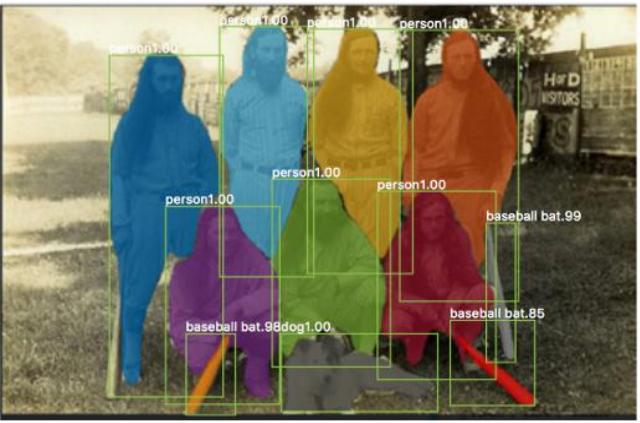
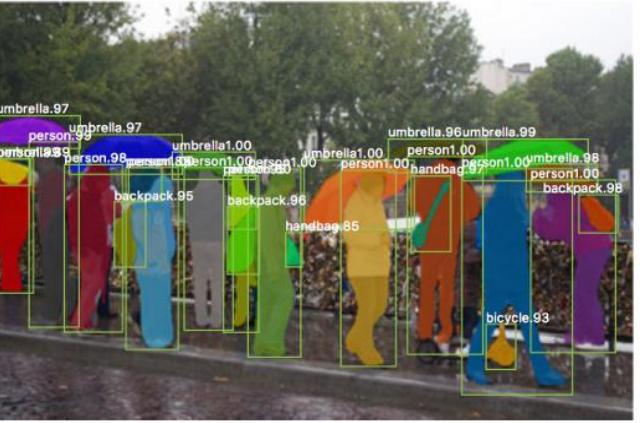
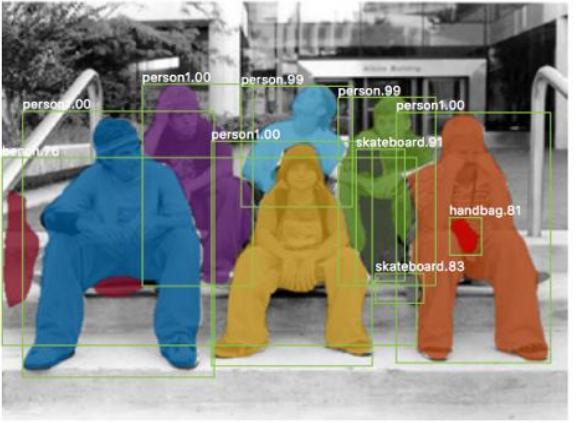
Mask R-CNN does this by adding a branch to Faster R-CNN that outputs a binary mask that says whether or not a given pixel is part of an object.

Here are its inputs and outputs:

- Inputs: CNN Feature Map.
- Outputs: Matrix with 1s on all locations where the pixel belongs to the object and 0s elsewhere (this is known as a binary mask).

Mask R-CNN

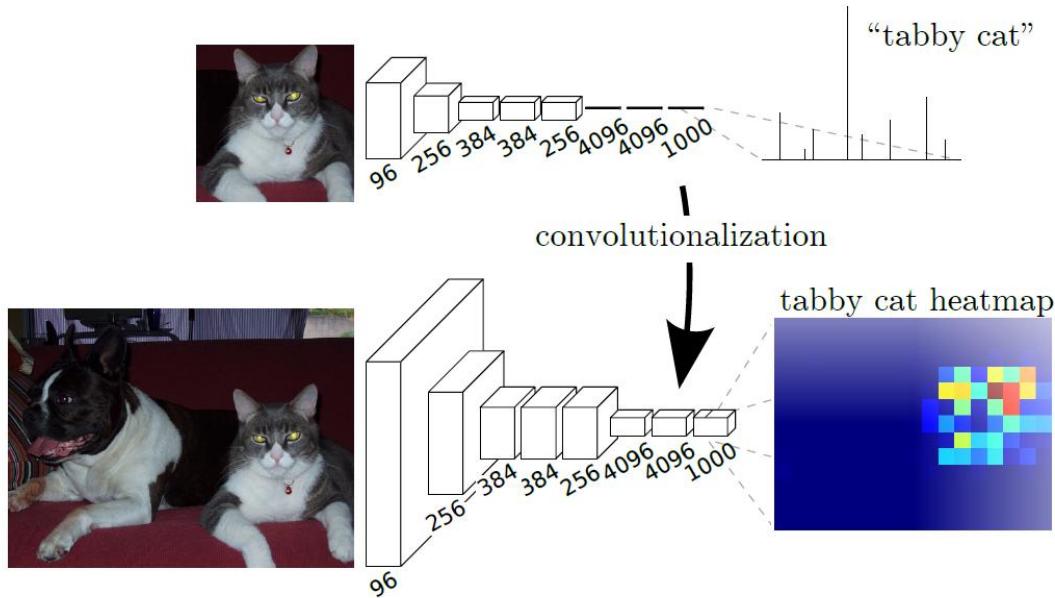




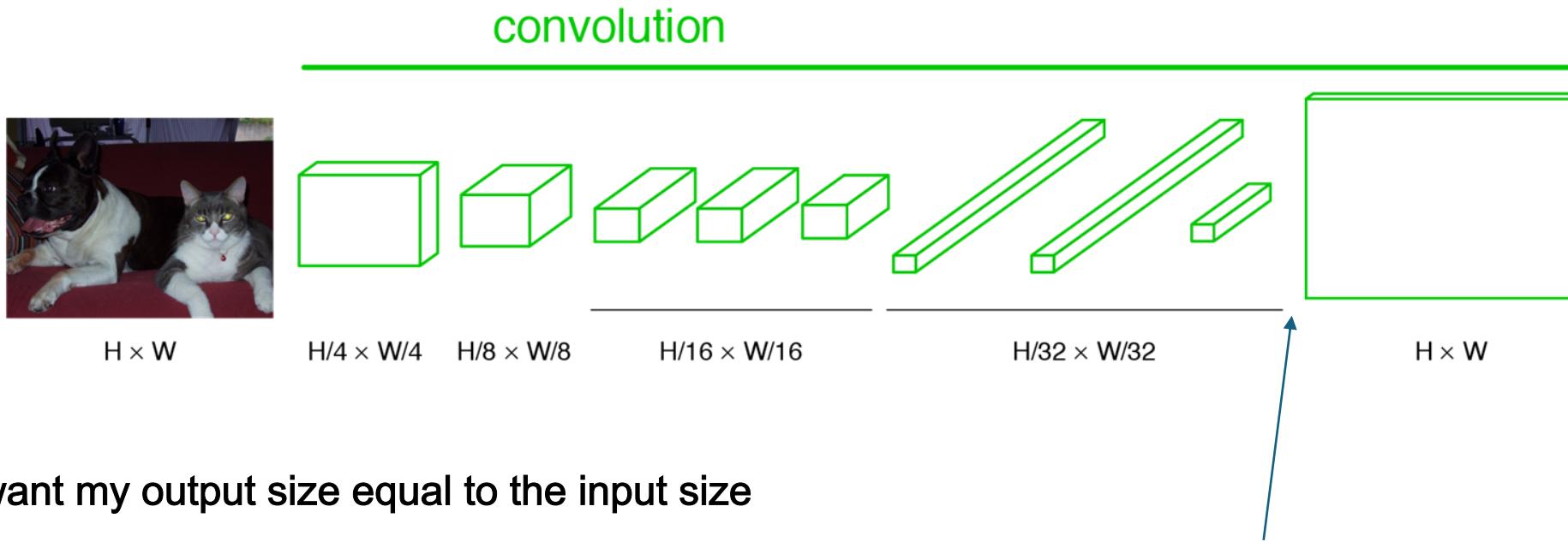
Towards pixel-level classification

Mask R-CNN classifies pixels within boxes.

Instead of first detecting objects and then segmenting them, can we also just classify each pixel directly to get a full segmentation mask?



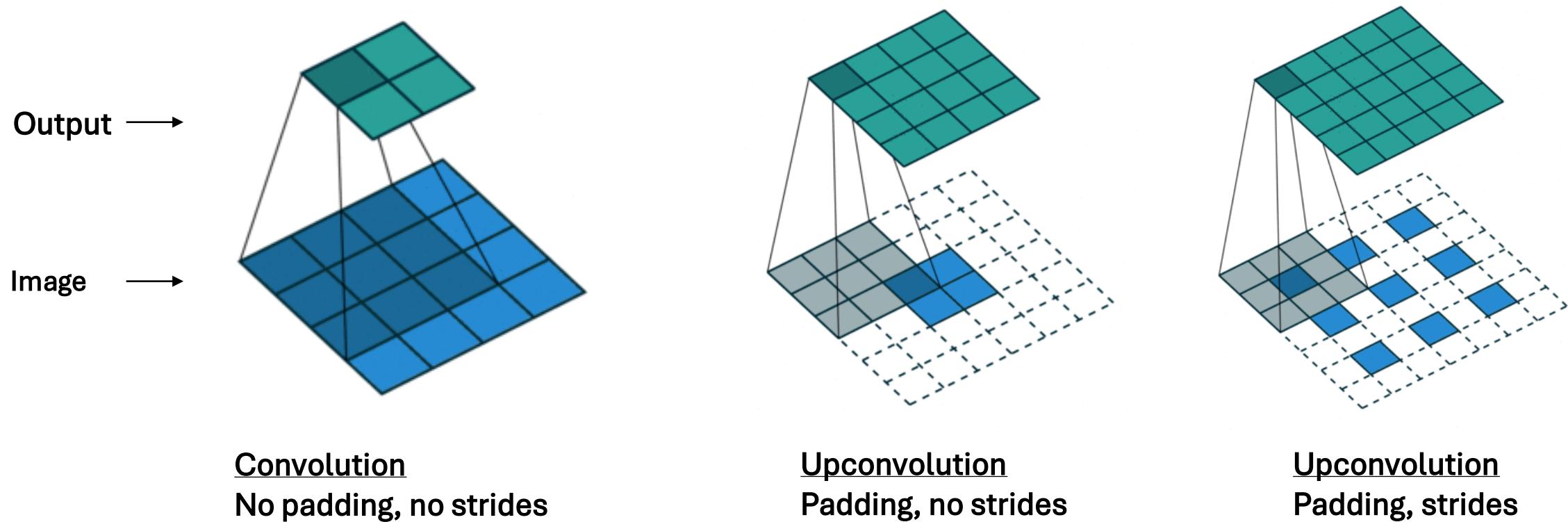
From a bottleneck back to a full image



I want my output size equal to the input size

Some upsampling
algorithm to return
us to $H \times W$

Deconvolutions



Convolution
No padding, no strides

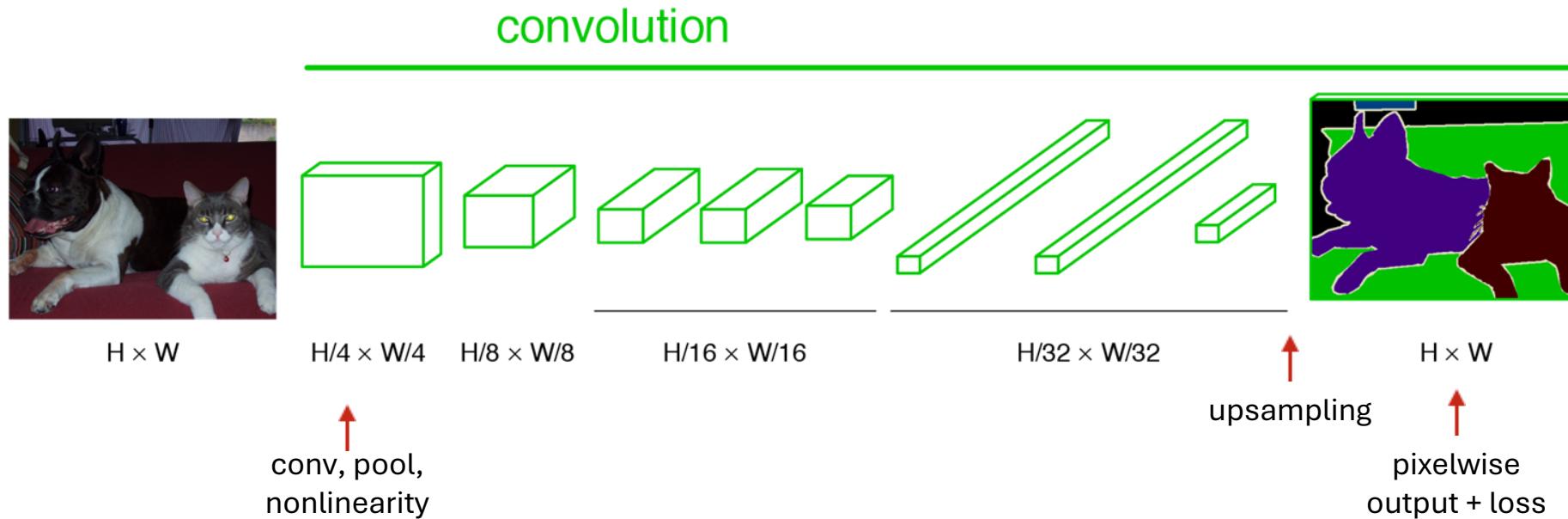
Upconvolution
Padding, no strides

Upconvolution
Padding, strides

More visualizations:

https://github.com/vdumoulin/conv_arithmetic

End-to-end pixel-level optimization



This lecture

Convolutions.

Convolutional networks.

Convolutional networks for detection and segmentation.

Learning and reflection

Chapter 9, Deep Learning Book

Chapter 10, Understanding Deep Learning Book

Learning and reflection

Understanding Deep Learning: Chapter 10

Understanding Deep Learning: Chapter 11

Next lecture

Lecture	Title	Lecture	Title
1	Intro and history of deep learning	2	AutoDiff
3	Deep learning optimization I	4	Deep learning optimization II
5	Convolutional deep learning	6	Attention-based deep learning
7	Graph deep learning	8	From supervised to unsupervised deep learning
9	Multi-modal deep learning	10	Generative deep learning
11	What doesn't work in deep learning	12	Non-Euclidean deep learning
13	Q&A	14	Deep learning for videos

Thank you



Deep Learning 1

2025-2026 – Pascal Mettes

Lecture 6

Attention-based Deep Learning

Previous lecture

Lecture	Title	Lecture	Title
1	Intro and history of deep learning	2	AutoDiff
3	Deep learning optimization I	4	Deep learning optimization II
5	Convolutional deep learning	6	Attention-based deep learning
7	Graph deep learning	8	From supervised to unsupervised deep learning
9	Multi-modal deep learning	10	Generative deep learning
11	What doesn't work in deep learning	12	Non-Euclidean deep learning
13	Q&A	14	Deep learning for videos

This lecture

Sequential modelling

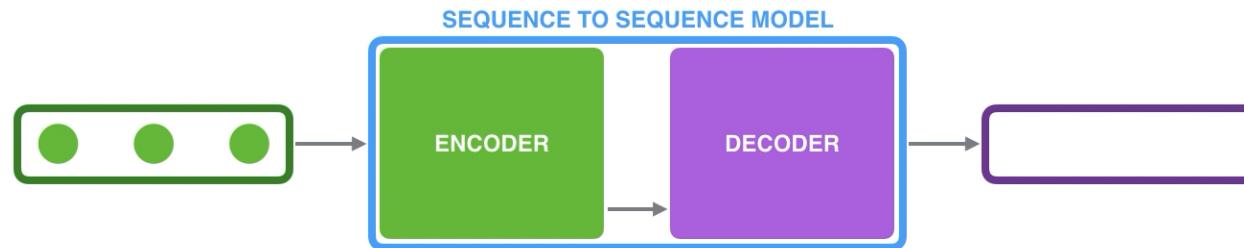
Attention and self-attention

Transformers

Language and vision transformers

Sequence2sequence models

we need a sort of memory that looks at the sequence and compute an overview that can be stored (a small ed fixed length representation)

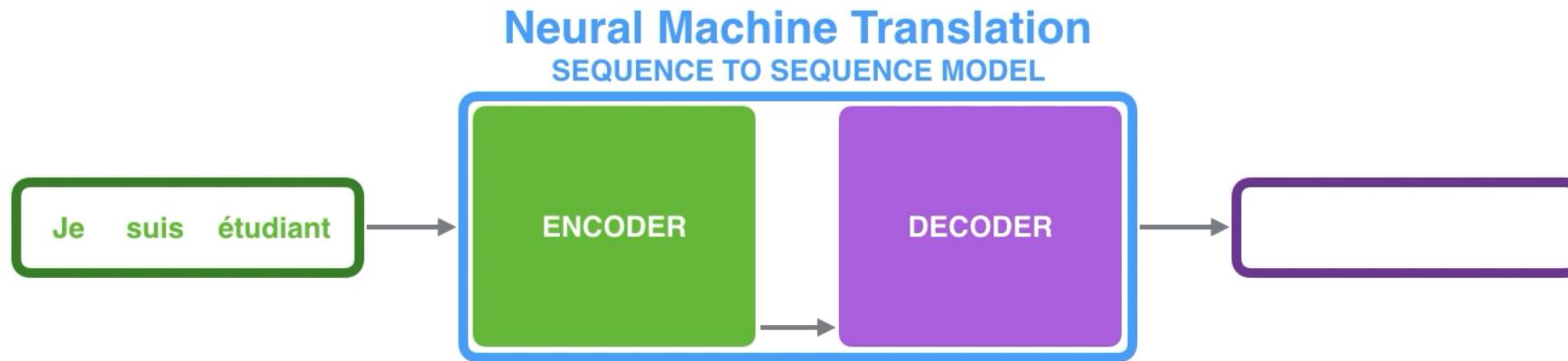


Encoder-decoder models with 2 parts:

1. **Encoder:** takes a variable-length sequence of elements as the input and transforms it into a context representation with a fixed-size.
2. **Decoder:** maps the encoded state of a fixed size to a variable-length sequence of elements.

Was a backbone for many problems, especially in language processing.

Sequence2sequence example



Encoder: reads sentence in one language and converts to context vector.

Decoder: outputs a translation from context vector.

Sequence2sequence example: encoder

An encoder encodes the input sentence, a sequence of vectors $x = (x_1, x_2, \dots, x_{T_x})$, into a context vector c .

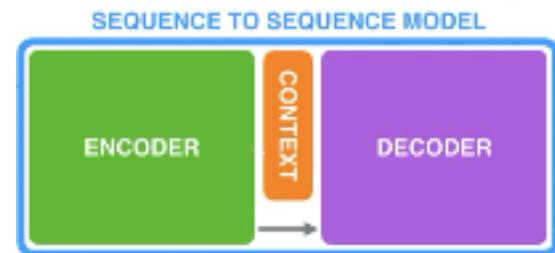
A common approach is to use an RNN/LSTM, such that:

$$h_t = f(x_t, h_{t-1}), \quad \text{it takes the input and the previous hidden state to compute the new hidden state}$$
$$c = q(\{h_1, \dots, h_{T_x}\}).$$

$h_t \in \mathbb{R}$ is a hidden state at time-step t ,

c is the context vector generated from the sequence of hidden states,

f and q are nonlinear functions.



$$h_t = f(x_t, h_{t-1}) \quad c = q(\{h_1, \dots, h_{T_x}\})$$

Sequence2sequence example: decoder

The decoder is trained to predict the next word y_t , given the context vector c and all the previously predicted words $\{y_1, \dots, y_{t-1}\}$.

It defines a probability over the translation y by decomposing the joint probability into the conditionals:

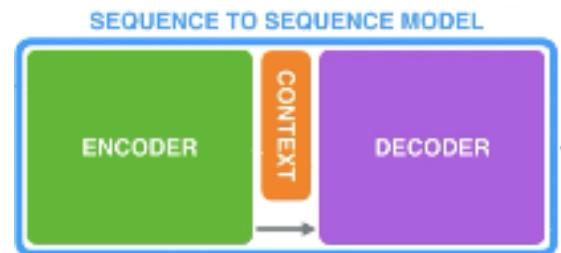
$$p(y) = \prod_{t=1}^T p(y_t | \{y_1, \dots, y_{t-1}\}, c), \text{ where } y = (y_1, \dots, y_T).$$

With an RNN decoder, each conditional probability is modeled as:

$$p(y_t | \{y_1, \dots, y_{t-1}\}, c) = g(y_{t-1}, s_t, c), \text{ where}$$

g is a nonlinear (multi-layered) function.

s_t is the hidden state of the RNN.



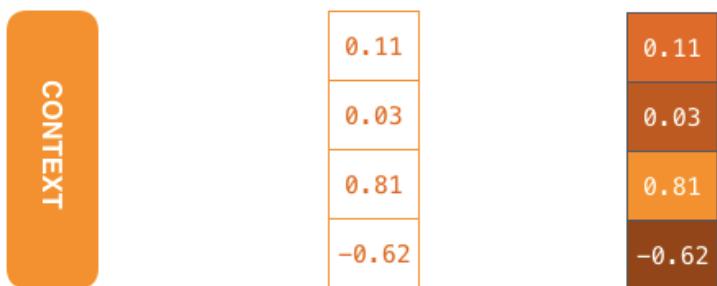
$$h_t = f(x_t, h_{t-1}) \quad c = q(\{h_1, \dots, h_{T_x}\})$$

the problem here is that I am compressing something, so every time I'm compressing I may be forgetting something

Limitation of seq2seq models

The model needs to compress all necessary information of a source sequence into the fixed-length context vector c , i.e., there is a bottleneck.

this bottleneck make us forget a lot



The context is a vector of floats, and its size is the number of hidden units in the encoder RNN.

When dealing with long sequences, it is challenging for the model to compress all information into a fixed-length context - due to vanishing gradients.



A simple solution: attention

Attention strives to overcome the bottleneck issue of the encoder-decoder.

Allows the model to focus on all relevant inputs to decode the output.

Neural Machine Translation example: for every new word to generate, the model searches for a set of positions in the input where the most relevant information is concentrated.

A huge breakthrough in language processing, and later in all other domains.

Attention

Stop encoding the whole input in a fixed-length vector.

Instead encode the input into a sequence of vectors.

Then focus on a subset of these vectors adaptively while decoding the output.

I.e., simply stop squashing all information!

now the context is everything

Attention formally

what I want from my new word is to attend from other previous word

Now each conditional probability of the decoder is defined as:

$$p(y_t | \{y_1, \dots, y_{t-1}\}, x) = g(y_{t-1}, s_i, c_i), \text{ where}$$

s_i is a hidden state for time i , computed by:

$$s_i = f(s_{i-1}, y_{i-1}, c_i).$$

The probability is conditioned on a distinct context vector c_i for each target word y_i (unlike the conventional encoder–decoder).

The context vector c_i depends on a sequence of annotations (h_1, \dots, h_{t-1}) to which an encoder maps the input sentence.

Each annotation h_i contains information about the whole input,
with a strong focus on the parts surrounding the i -th word of the input sequence.

the context is everything, how do we do aggregation?

the attention is the weight of each element in my sequence

Attention formally

The context vector c_i is computed as a *weighted sum* of these annotations h_j :

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j,$$

where the weight α_{ij} of each annotation h_j is computed by the probability:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})},$$

where $e_{ij} = a(s_{i-1}, h_j)$ is an alignment model,

It scores how well the inputs around position j and the output at position i match.

The alignment model a is parametrized as a feedforward neural net,

And is jointly trained with all the other components of the model.

Attention formally

The weight α_{ij} reflects the importance of the annotation h_j , with respect to the previous hidden state s_{i-1} when deciding the next state s_i and generating y_i .

This implements a mechanism of attention in the decoder.

The decoder decides which parts of the source sentence to pay attention to.

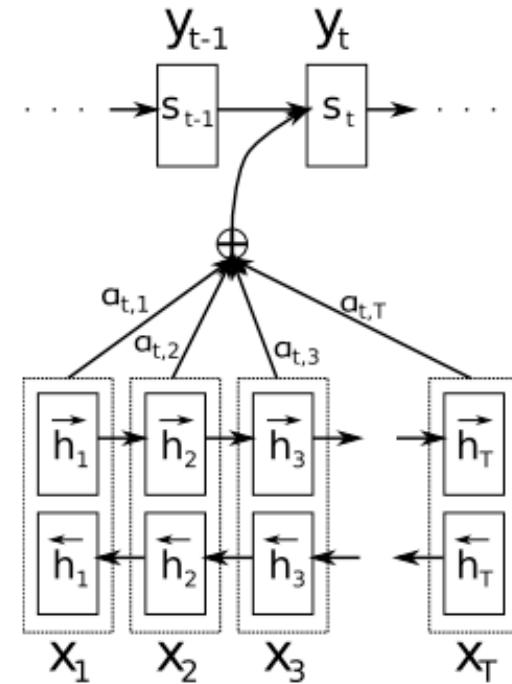
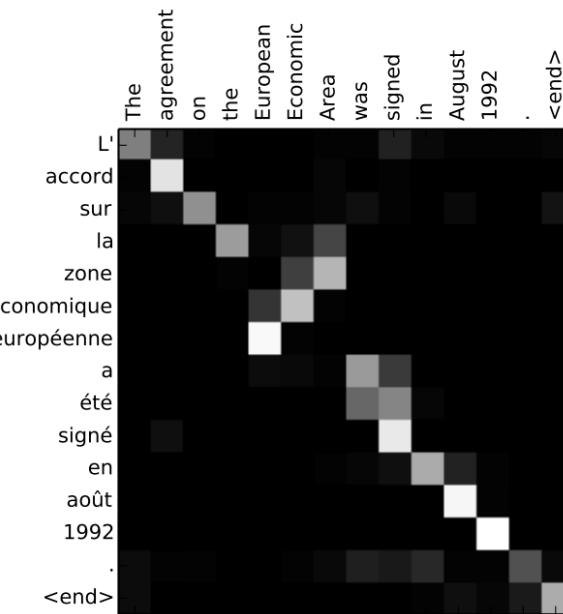
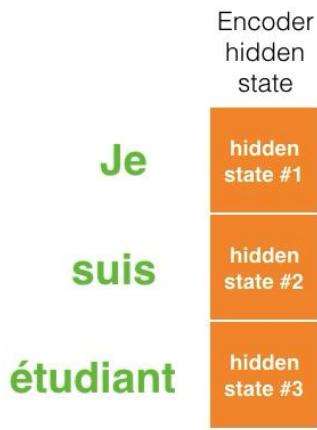


Fig. 1 Graphical illustration of attention model trying to generate the t -th target word y_t given a source sentence (x_1, x_2, \dots, x_T) ¹

Visualizing attention



<https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/>

instead of sequence-to sequence self attention, we apply the attention to the same sequence, computing the similarity within each pair of token

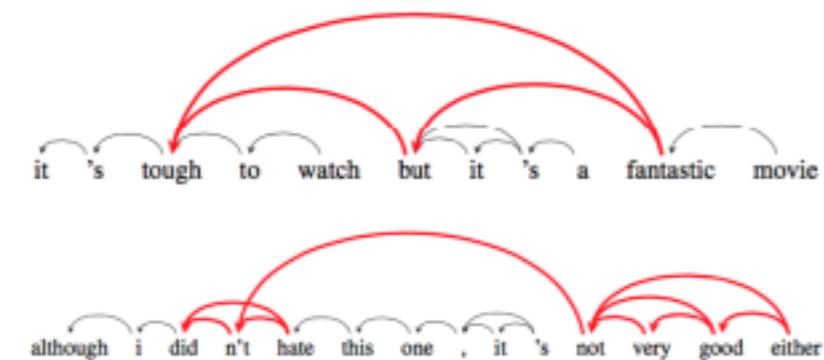
Self-attention

Self-attention (or intra-attention): relates parts of sequence with each other.

Results is a representation of the whole sequence.

In general terms, it can be seen as an operation on sets of elements.

The FBI is chasing a criminal on the run .
The FBI is chasing a criminal on the run .
The FBI is chasing a criminal on the run .
The FBI is chasing a criminal on the run .
The FBI is chasing a criminal on the run .
The FBI is chasing a criminal on the run .
The FBI is chasing a criminal on the run .
The FBI is chasing a criminal on the run .
The FBI is chasing a criminal on the run .
The FBI is chasing a criminal on the run .



The transformer

Attention Is All You Need

Ashish Vaswani*
Google Brain
avaswani@google.com

Noam Shazeer*
Google Brain
noam@google.com

Niki Parmar*
Google Research
nikip@google.com

Jakob Uszkoreit*
Google Research
usz@google.com

Llion Jones*
Google Research
llion@google.com

Aidan N. Gomez* †
University of Toronto
aidan@cs.toronto.edu

Lukasz Kaiser*
Google Brain
lukaszkaiser@google.com

Illia Polosukhin* ‡
illia.polosukhin@gmail.com

the transformer is an architecture that do self attention over and over to apply it to the full sentence token by token

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.0 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature.

The transformer

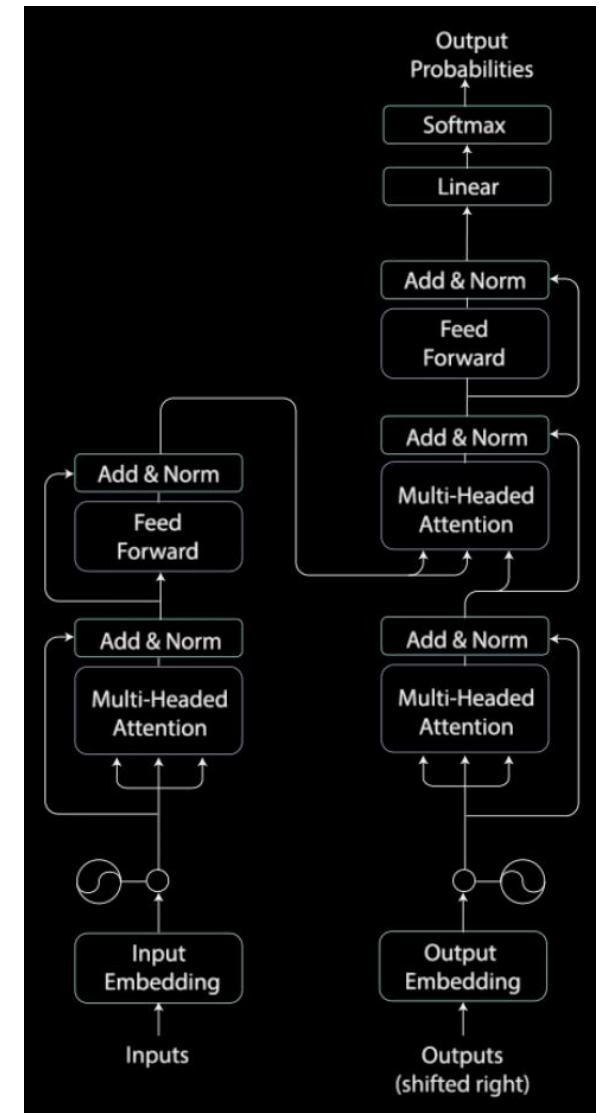
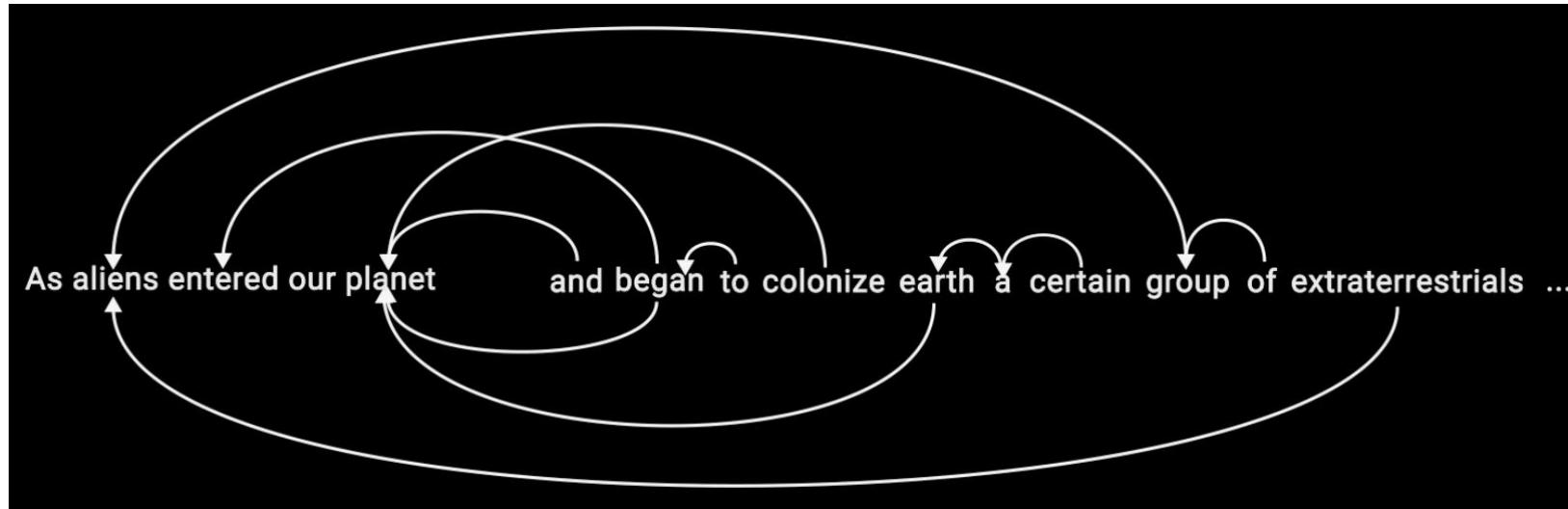
An encoder-decoder model based on (self-)attention mechanisms:

- Without any recurrence and convolutions.
- Referred to as NLP's ImageNet moment.
- It completely changed the landscape of deep learning models.
- Dominates modern deep learning.
- Key behind LLMs.

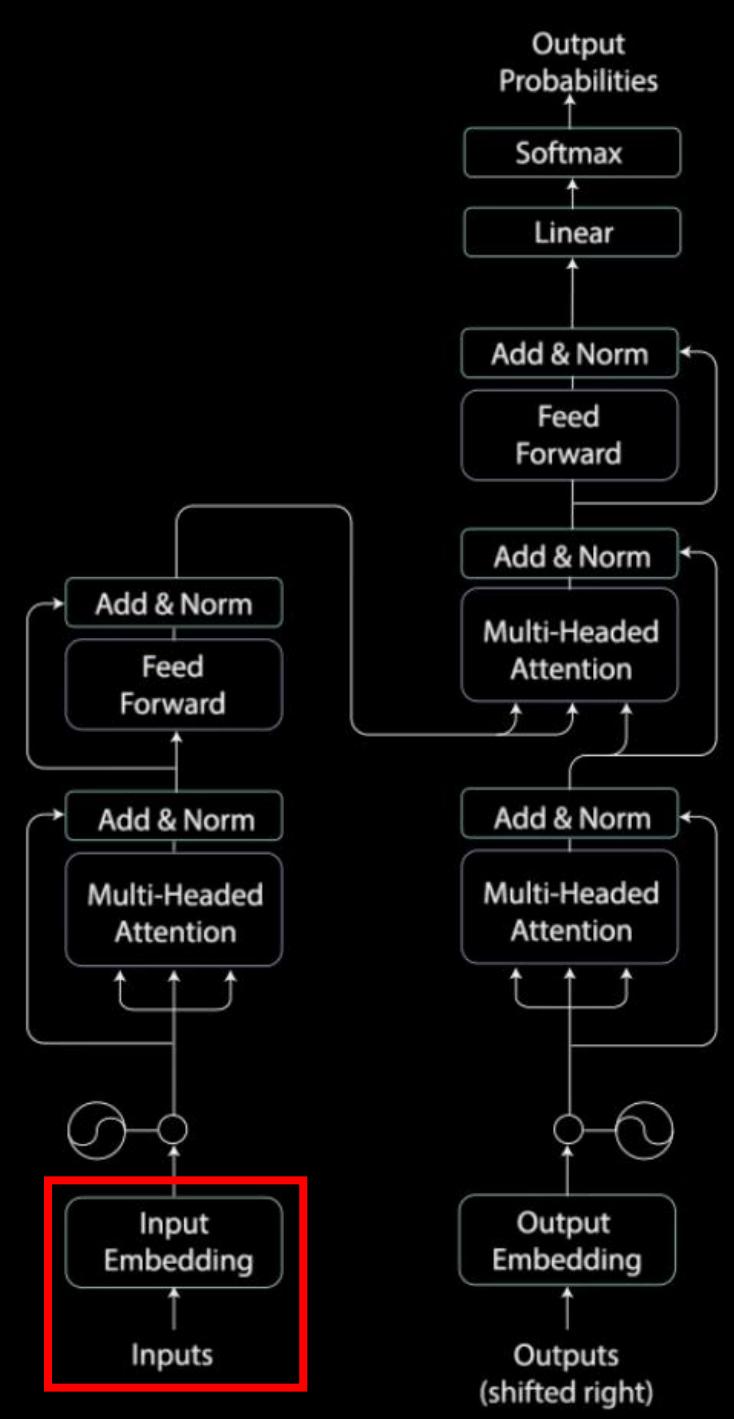
Important concepts:

- Queries, keys, values.
- Scaled dot-product attention.
- Multi-head (self-)attention.

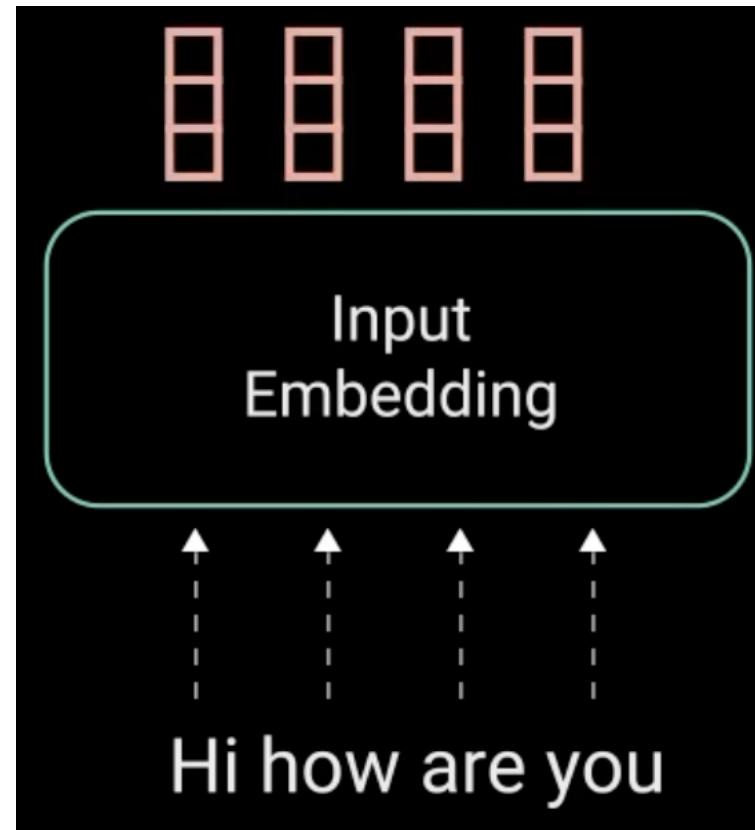
Understanding transformers: let's do a forward pass

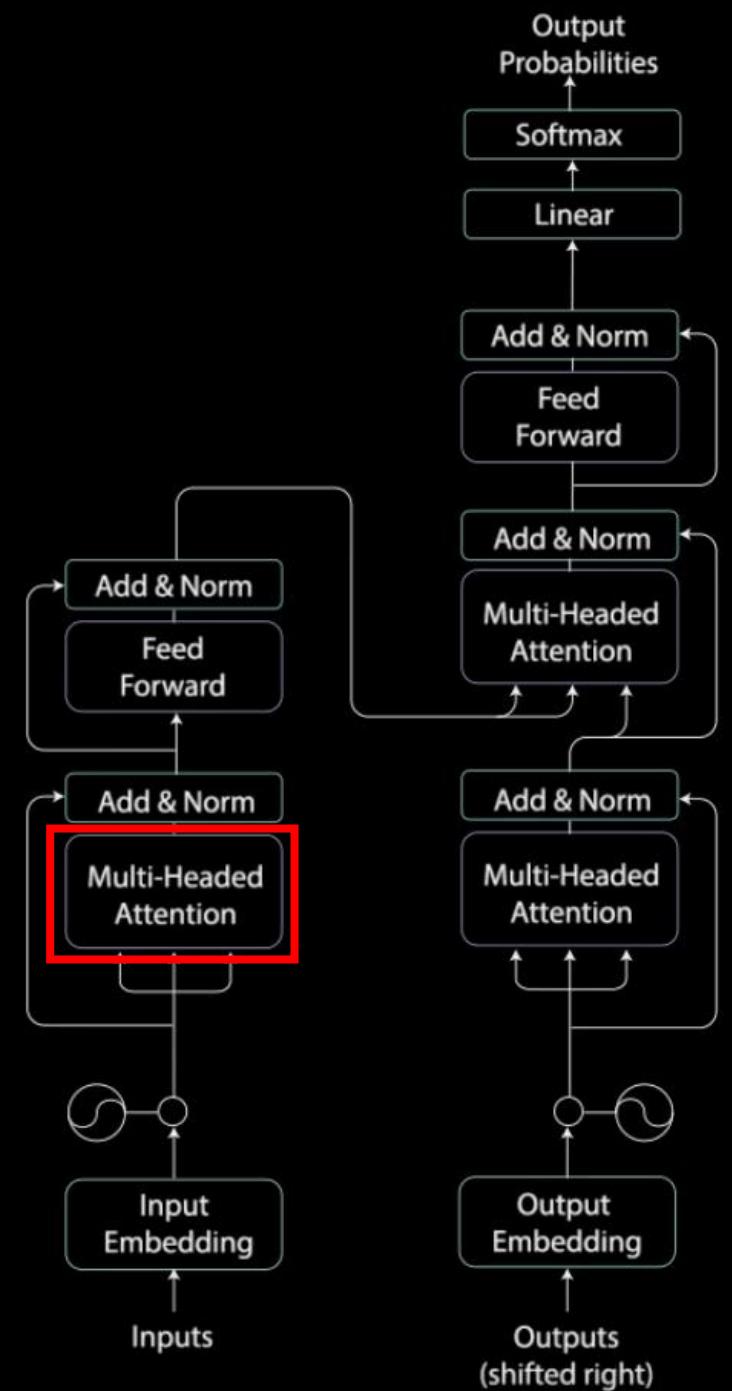


Thank you to Michael Phi for the visualizations.

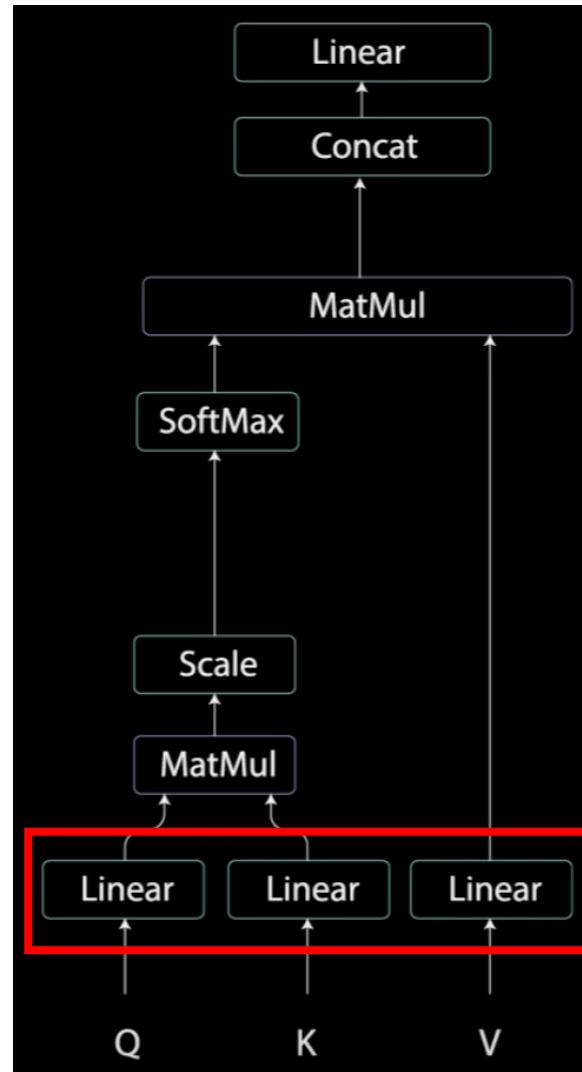


Features vector are one hot encoded,

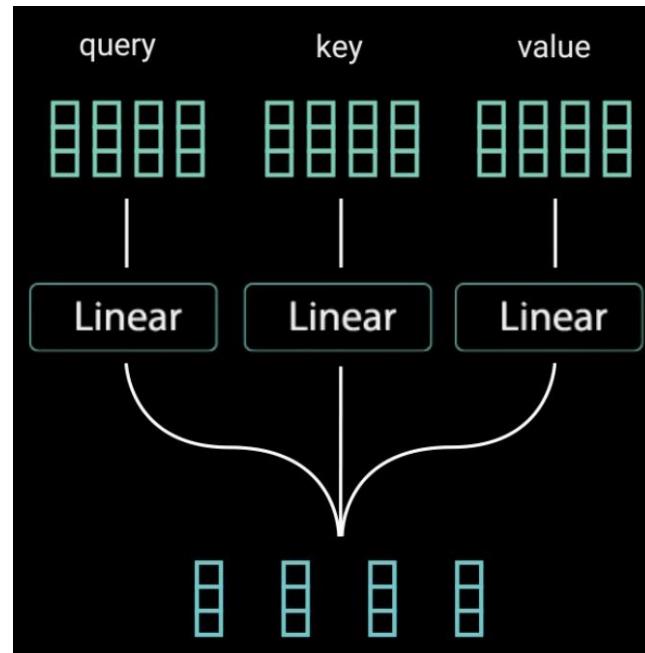


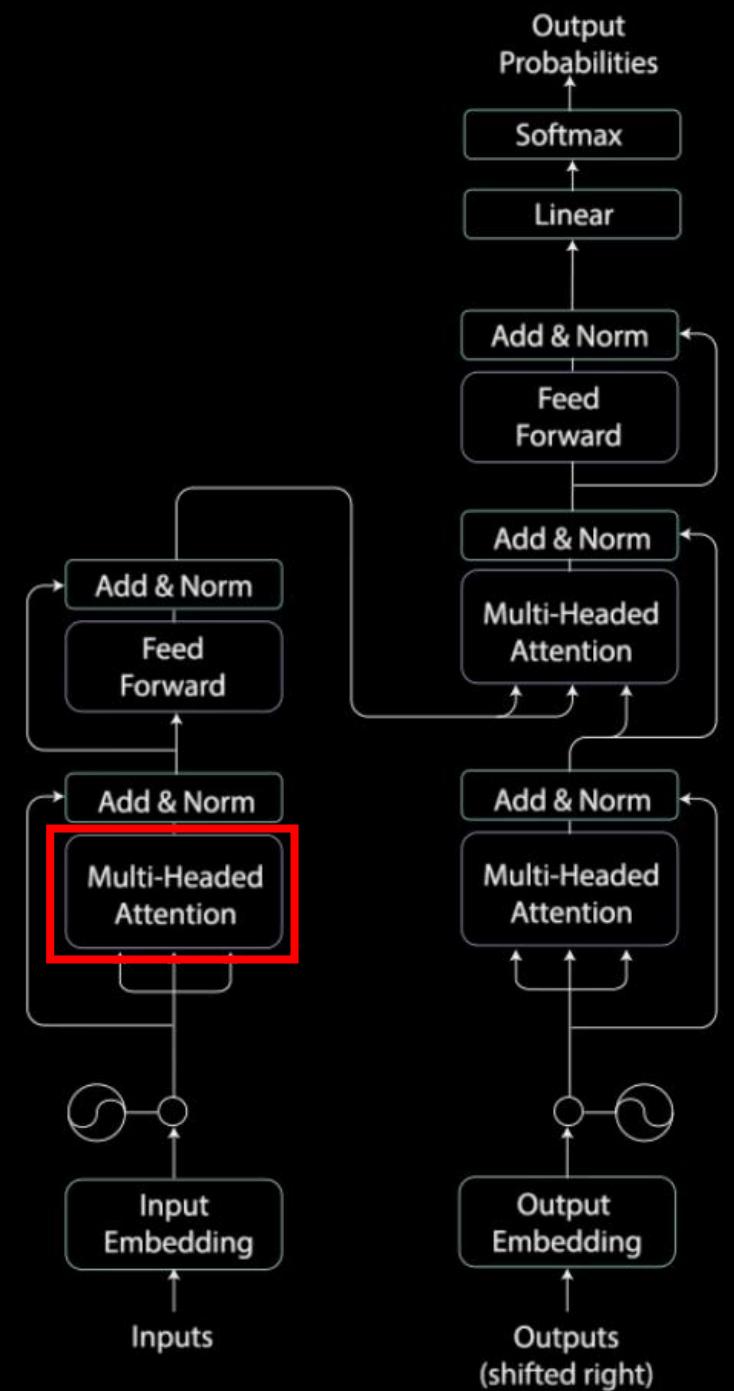


I have a matrix as input, the idea of self attention is to transforms each of the feature vectors so that the output is of the same length but the representation is transformed. The surrounding is transforming using myself. The important thing is that input and output are the same

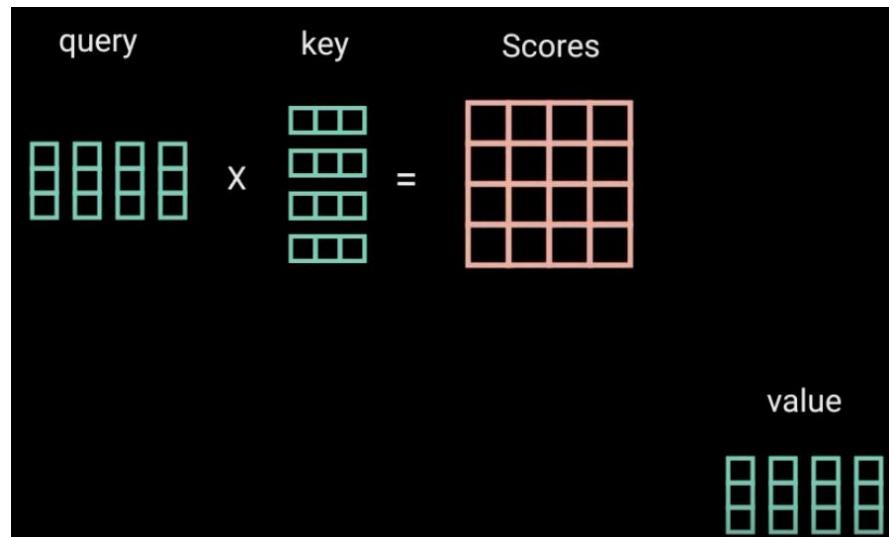
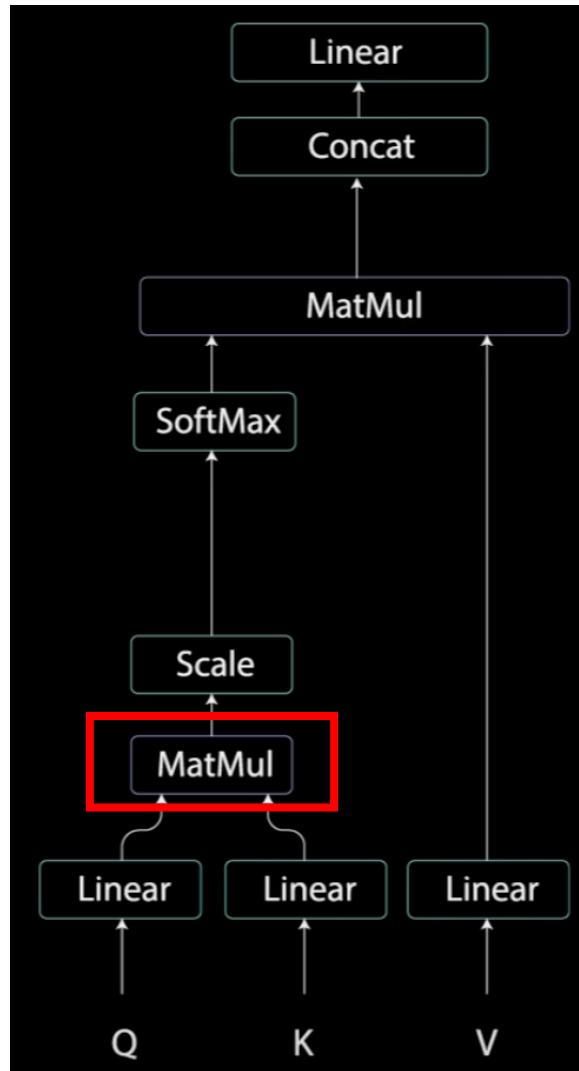


I'm transforming my feature vector to 3 linear layer MLP

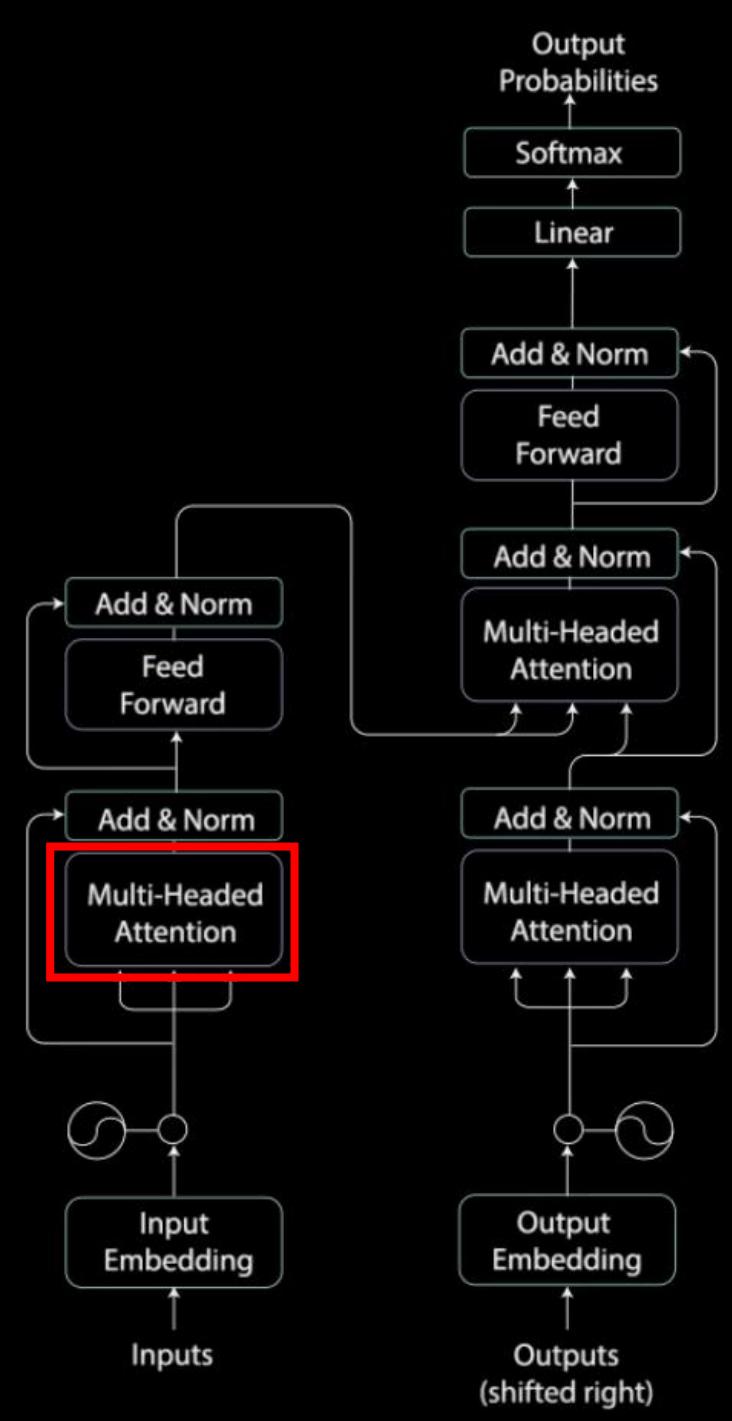




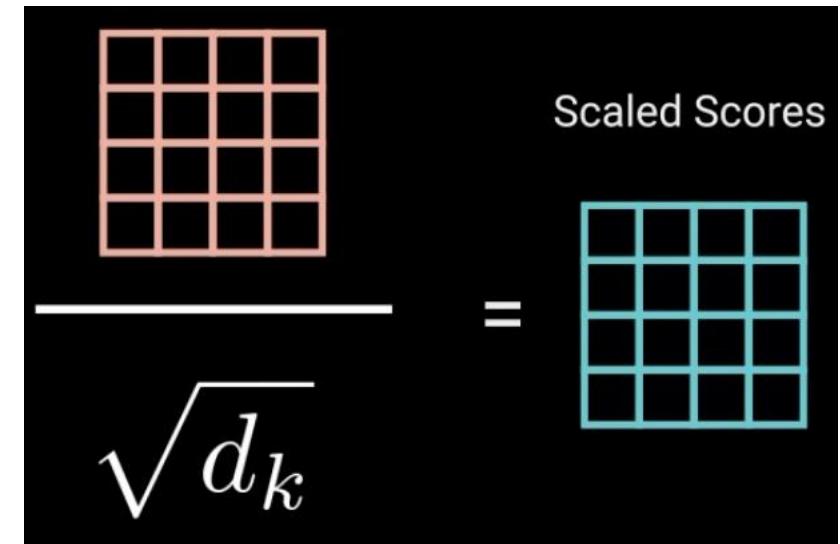
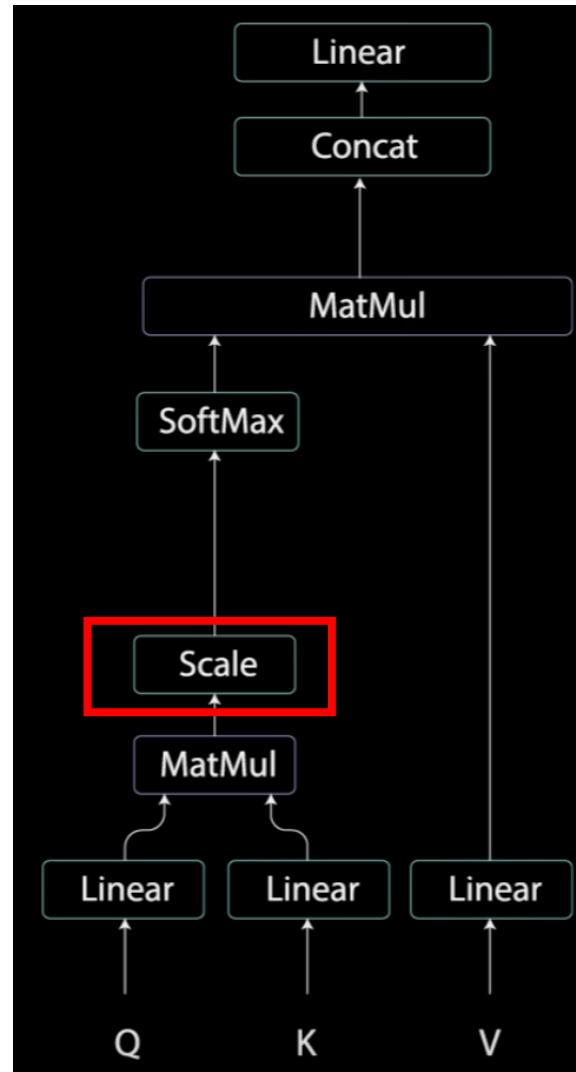
Now I combine the first 2 matreces by doing matrix multiplication. The dot product is the measure of similarity. so we obtain a matrix of score pair similarities

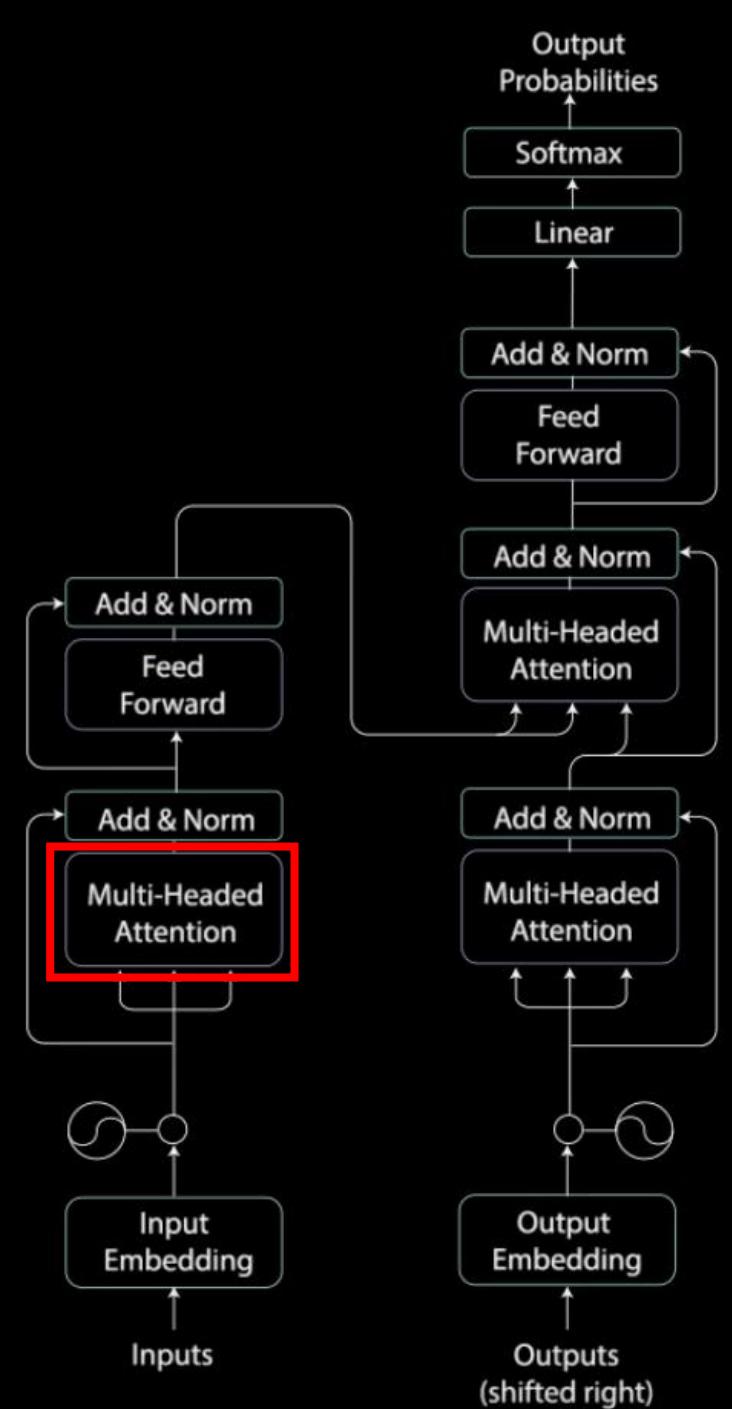


	Hi	how	are	you
Hi	98	27	10	12
how	27	89	31	67
are	10	31	91	54
you	12	67	54	92

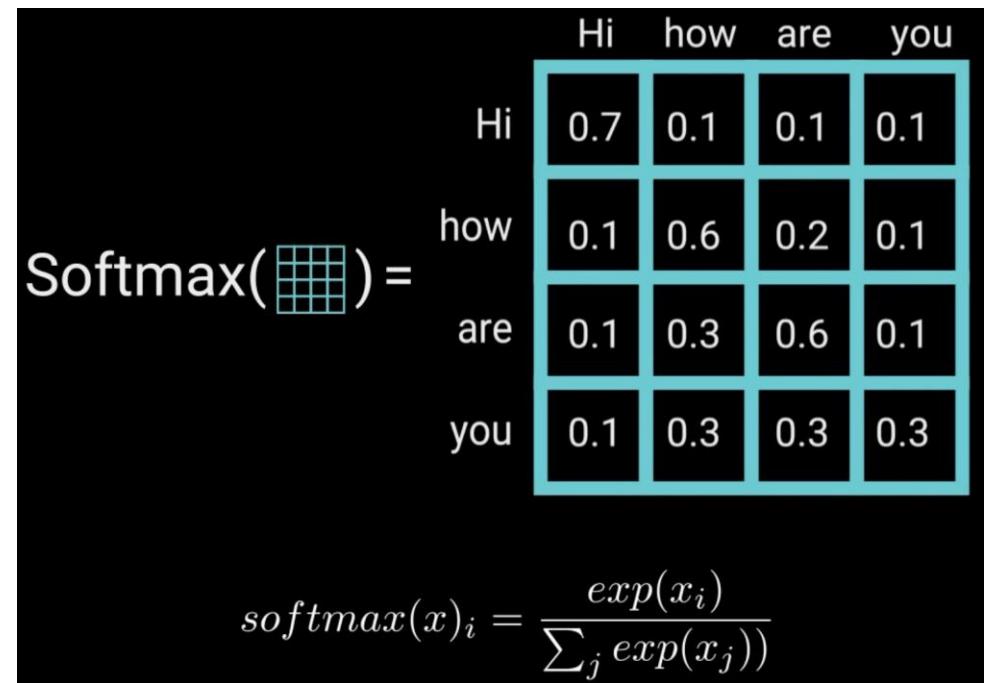
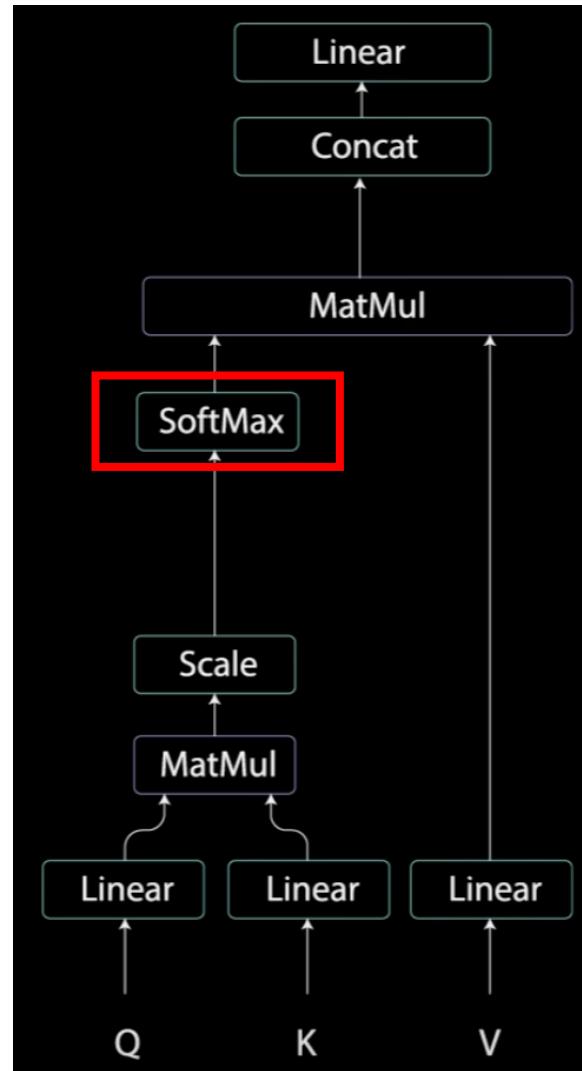


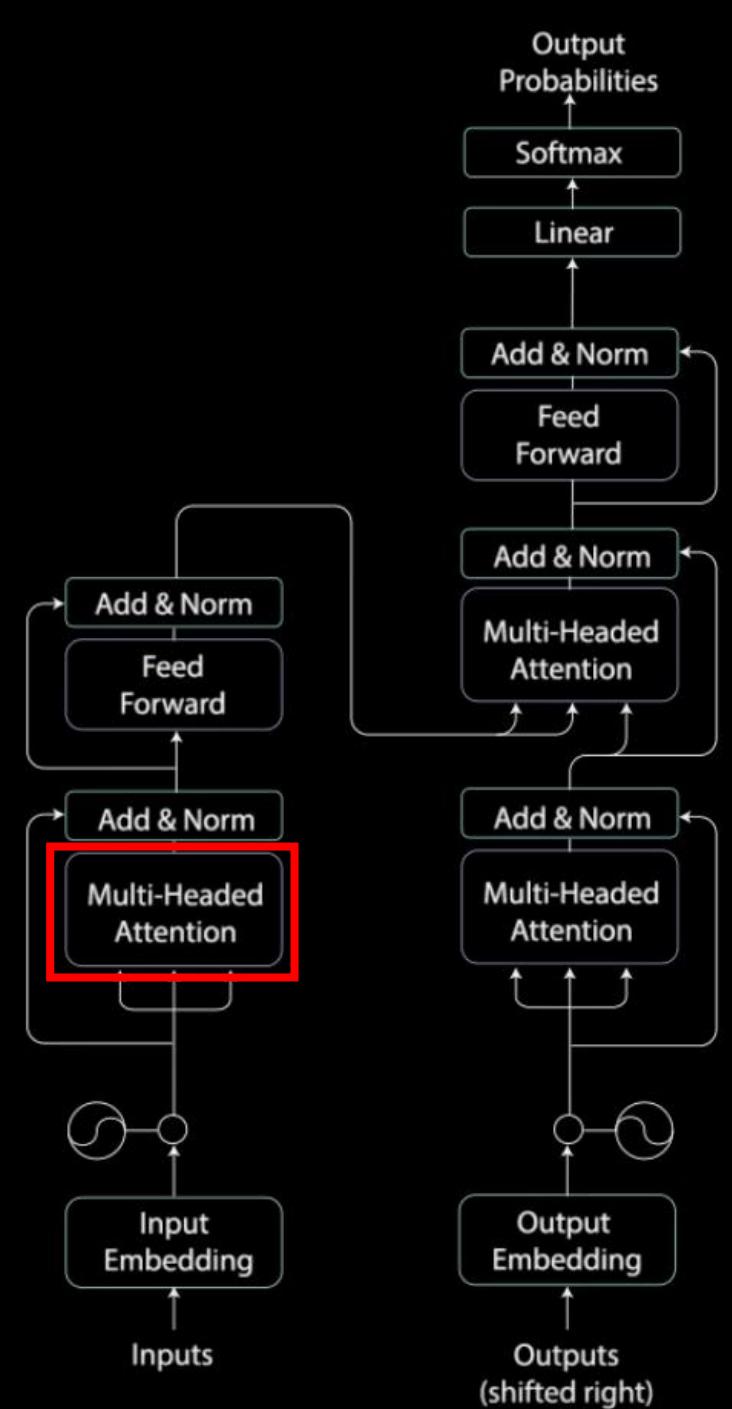
I'm gonna take the similarities and I'm gonna divide them for the square root of the dimensionalities of K



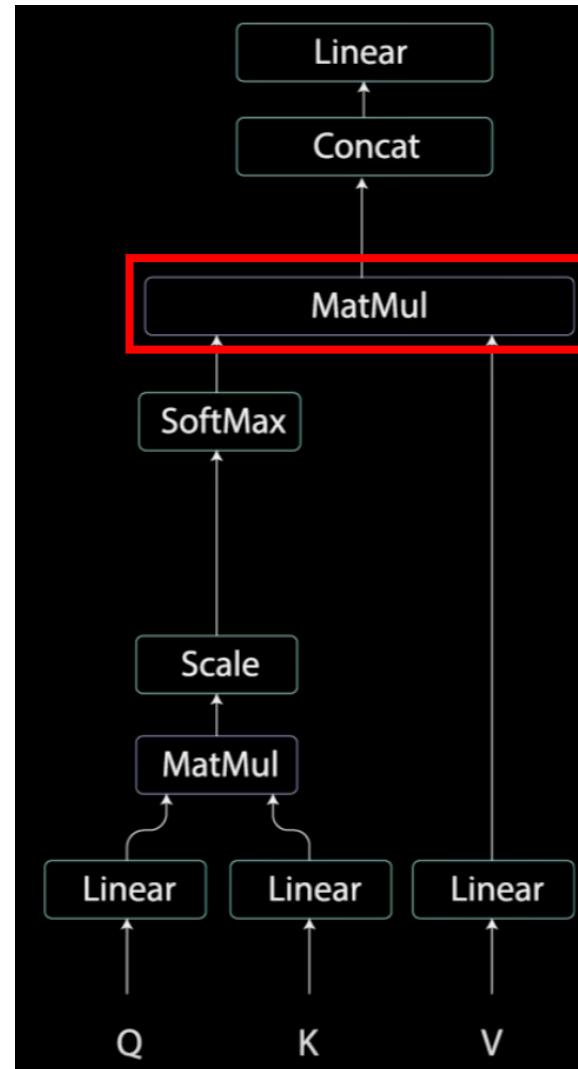


I want for each words the similarity to all the other words, so I'm gonna go through the softmax. For each word I want the relative attention to all the other words in the sentence.

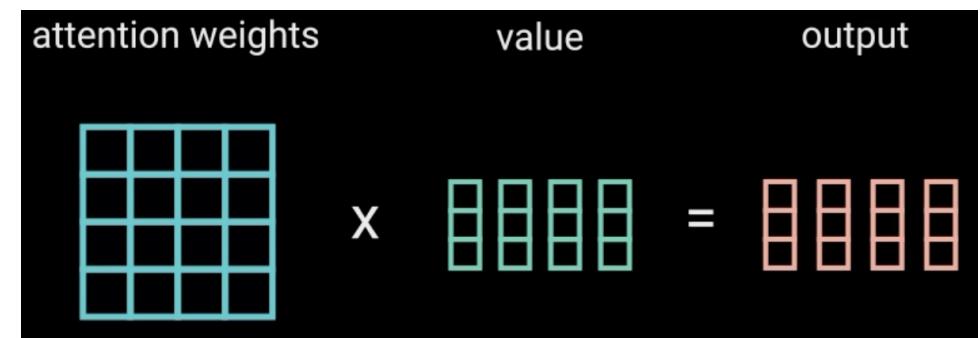




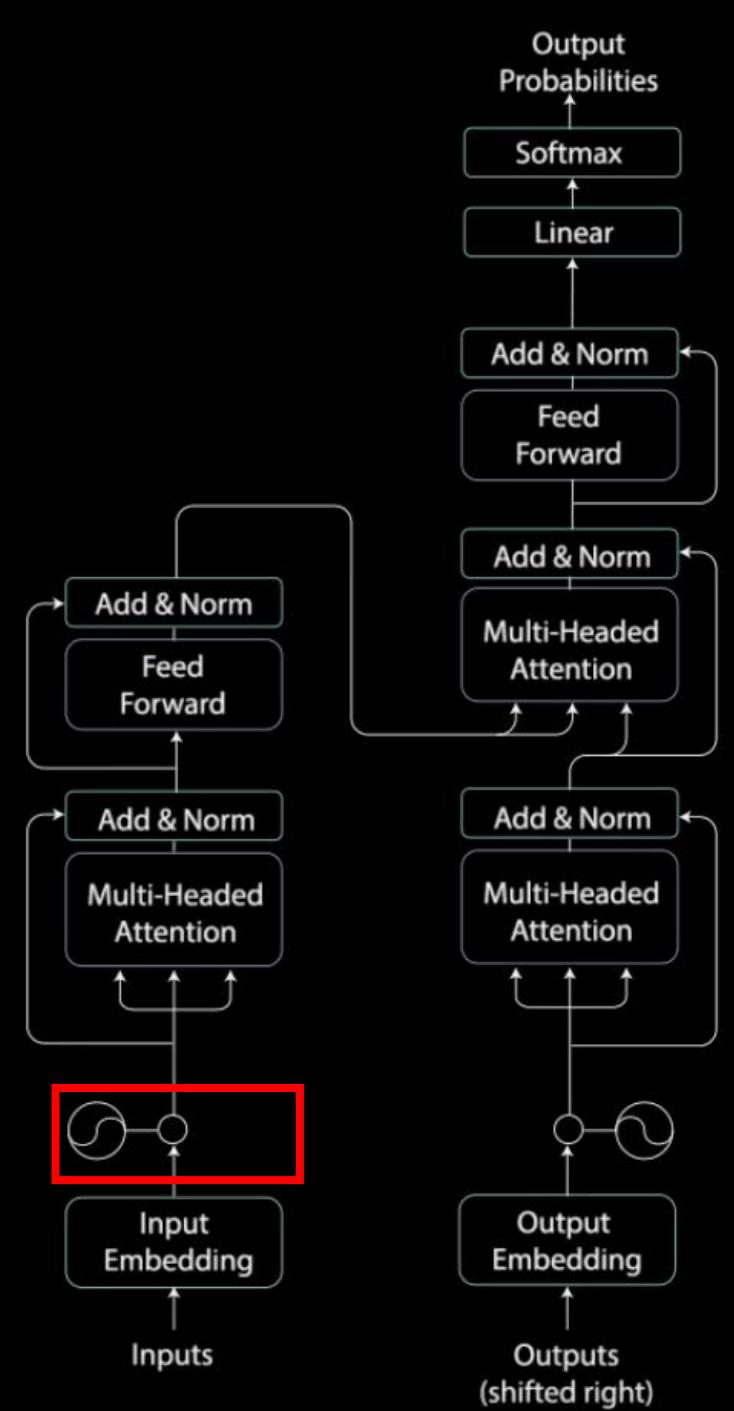
I have the old feature vector and I want to compute a new feature vector using the matrix of attention weights. here it is a true matrix multiplication



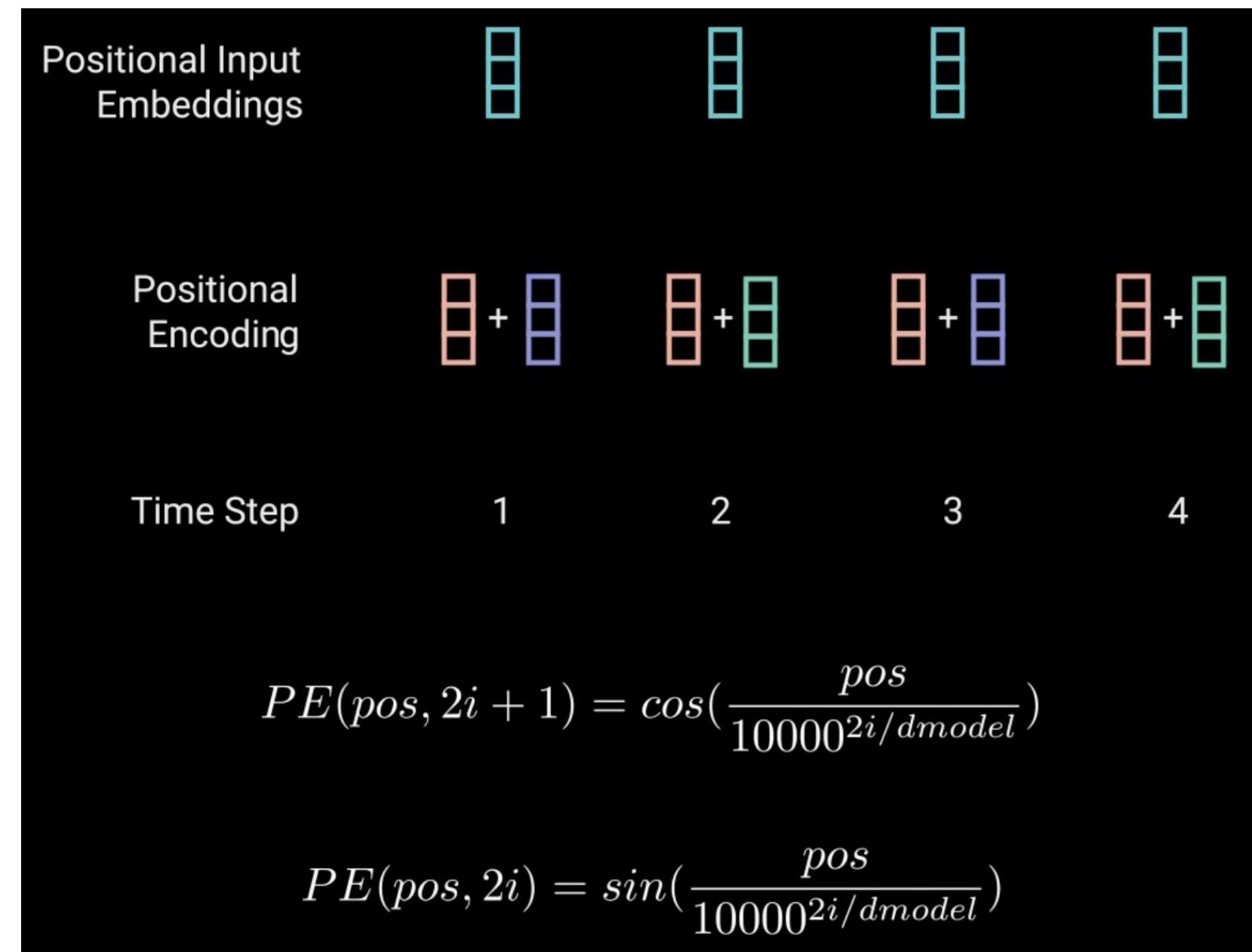
Achtung! wrong matrix size display:
value should be on the left i think

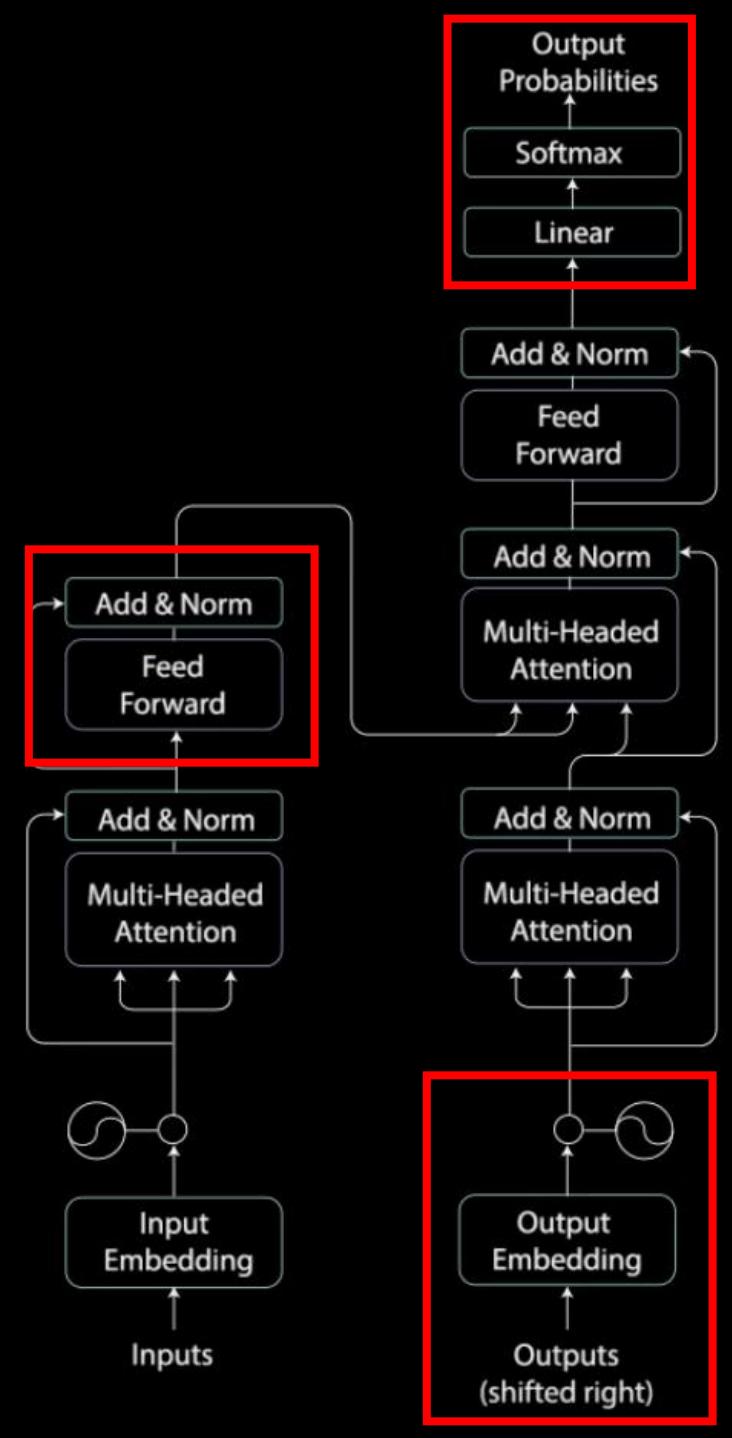


The location in which elements occurs is an interesting thing that we should capture. Recurrency is too strict, but self-attention is too loose. What we are going to do is to use the matrix of self attention and update the values inside based on the position of the elements. We want a function that based on the index gives us some values and we will use these values to update the feature vectors



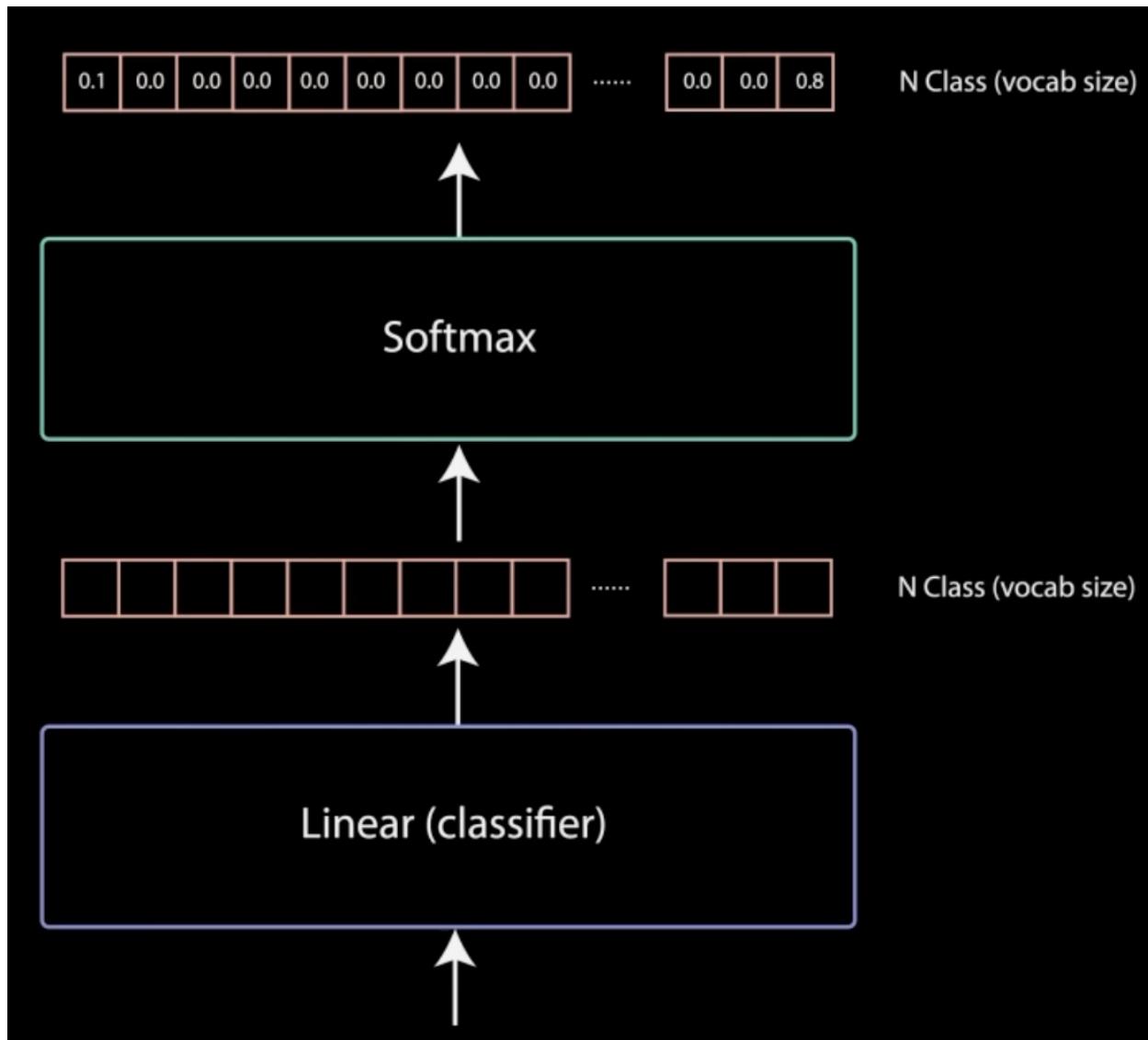
why this denominator? because it works





Now I'm doing the same thing with the output, (autoregressively) (bottom right). I'm going to do the cross attention with the transformed outputs with the transformed inputs, this vectors goes throuth a linear layer and a softmax in order to produce

there is an iteration at the end, when we predict the new token(word) we use it to start again and predict the next one. We stop only when we predict the EOS token



At a limited scale this is very difficult to train, but at large scale it works very well

The left hand part is computed only one time for all the sentence. Then for each token we compute only the right hand side using the attention computed on the LHS

Queries, keys, and values

The Transformer paper redefines the attention mechanism by providing a generic definition based on **queries**, **keys**, **values**.

Intuition: Use the **query** of the target and the **key** of the input to calculate a matching score.

These matching scores act as the weights of the **value** vectors.

$$\begin{array}{ccc} \mathbf{x} & \times & \mathbf{W}^Q \\ \begin{matrix} \text{green} \\ \text{grid} \end{matrix} & \times & \begin{matrix} \text{purple} \\ \text{grid} \end{matrix} \\ & = & \begin{matrix} \text{purple} \\ \text{grid} \end{matrix} \end{array}$$
$$\begin{array}{ccc} \mathbf{x} & \times & \mathbf{W}^K \\ \begin{matrix} \text{green} \\ \text{grid} \end{matrix} & \times & \begin{matrix} \text{orange} \\ \text{grid} \end{matrix} \\ & = & \begin{matrix} \text{orange} \\ \text{grid} \end{matrix} \end{array}$$
$$\begin{array}{ccc} \mathbf{x} & \times & \mathbf{W}^V \\ \begin{matrix} \text{green} \\ \text{grid} \end{matrix} & \times & \begin{matrix} \text{blue} \\ \text{grid} \end{matrix} \\ & = & \begin{matrix} \text{blue} \\ \text{grid} \end{matrix} \end{array}$$

The input consists of queries and keys of dimension d_k , and values of dimension d_v .

$$\begin{aligned} & \text{softmax} \left(\frac{\mathbf{Q} \times \mathbf{K}^T}{\sqrt{d_k}} \right) \mathbf{V} \\ &= \mathbf{Z} \end{aligned}$$

Compute the dot products of the query with all keys, divide each by $\sqrt{d_k}$, and apply a softmax function to obtain the weights on the values.

Attention: dot product with scaling

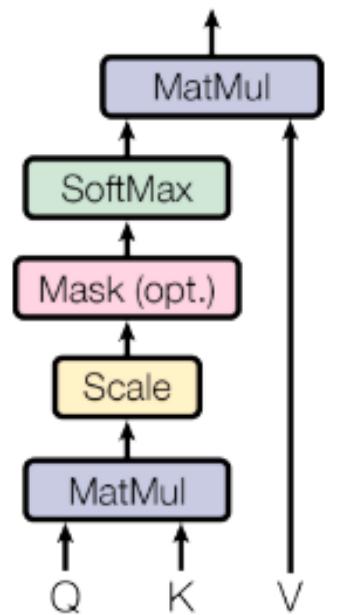
Attention mechanism in transformers are “scaled dot-product attention”.

Dot product computes similarity between queries and keys.

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

For large values of d_k the dot products grow large in magnitude, pushing the softmax function into regions with extremely small gradients.

The scaling by $\frac{1}{\sqrt{d_k}}$ counteracts this.



1. The Meaning: "Don't Mix Concepts"

Imagine the

k

k -th column of your embedding matrix

V

V represents the concept of "plurality" (singular vs. plural).

Word 1 ("Cats") has a high value in this column.

Word 2 ("run") has a neutral value.

When you multiply the attention matrix

A

A by this specific "plurality" column:

You are asking: "Is the word I am attending to plural?"

You are NOT asking: "Is the word I am attending to red? Or happy?"

By isolating the multiplication to just this one column (direction), the math ensures that the "plurality" information in the output comes exclusively from the "plurality" information of the input words. It prevents the "color" of one word from accidentally changing the "plurality" of another.

Multi-headed attention

I can do attention multiple times in parallel

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

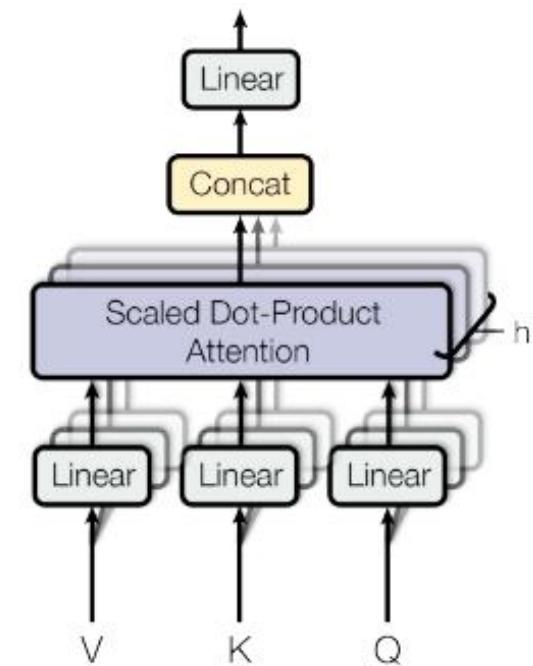
It is beneficial to linearly project the Q, K and V, h times with different, linear projections to d_k , d_k and d_v dimensions.

The attention function is performed in parallel, on each of these projected versions of Q, K and V.

These are concatenated and again projected, resulting in the final values.

Similar to the multiple filters in each convolutional layer.

$$W^O_i \in \mathbb{R}^{hd_v \times d_{model}}.$$



Multi-headed self-attention

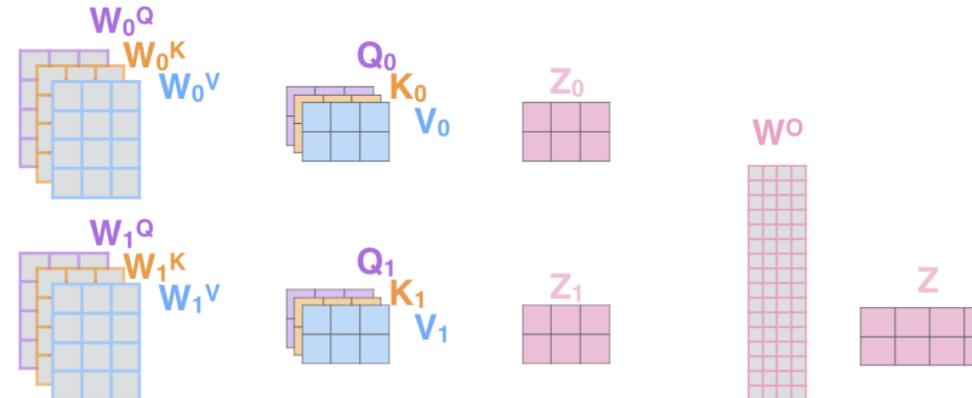
For multi-head self-attention: the **queries**, **keys**, **values** are equal to the input representation or from the previous (encoding/decoding) layer.

1) This is our
input sentence* 2) We embed
each word*

3) Split into 8 heads.
We multiply X or
 R with weight matrices

4) Calculate attention
using the resulting
 $Q/K/V$ matrices

5) Concatenate the resulting Z matrices,
then multiply with weight matrix W^O to
produce the output of the layer



* In all encoders other than #0,
we don't need embedding.
We start directly with the output
of the encoder right below this one



$$\text{softmax}\left(\frac{Q \times K^T}{\sqrt{d_k}}\right) = Z$$

Q K^T V

Break

Attention: dot product with scaling

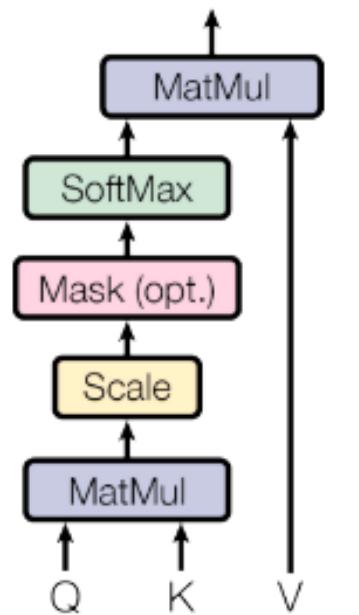
Attention mechanism in transformers are “scaled dot-product attention”.

Dot product computes similarity between queries and keys.

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

For large values of d_k the dot products grow large in magnitude, pushing the softmax function into regions with extremely small gradients.

The scaling by $\frac{1}{\sqrt{d_k}}$ counteracts this.



Multi-headed self-attention

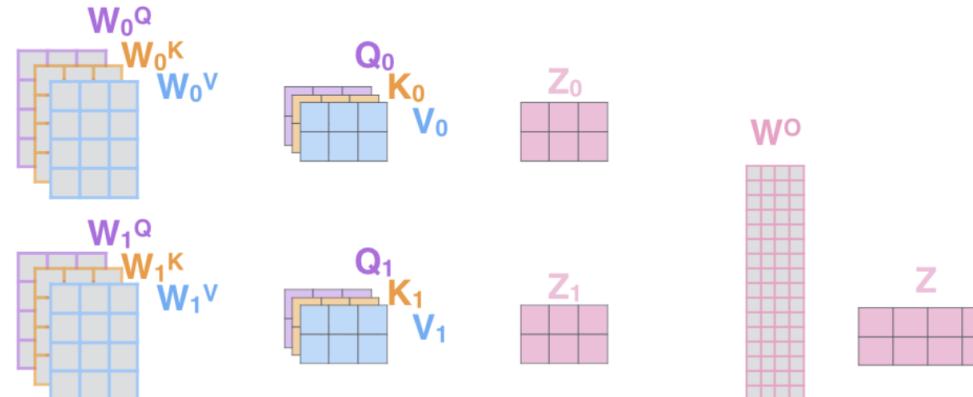
For multi-head self-attention: the **queries**, **keys**, **values** are equal to the input representation or from the previous (encoding/decoding) layer.

1) This is our
input sentence* 2) We embed
each word*

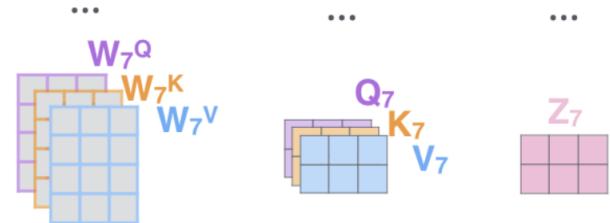
3) Split into 8 heads.
We multiply X or
 R with weight matrices

4) Calculate attention
using the resulting
 $Q/K/V$ matrices

5) Concatenate the resulting Z matrices,
then multiply with weight matrix W^O to
produce the output of the layer

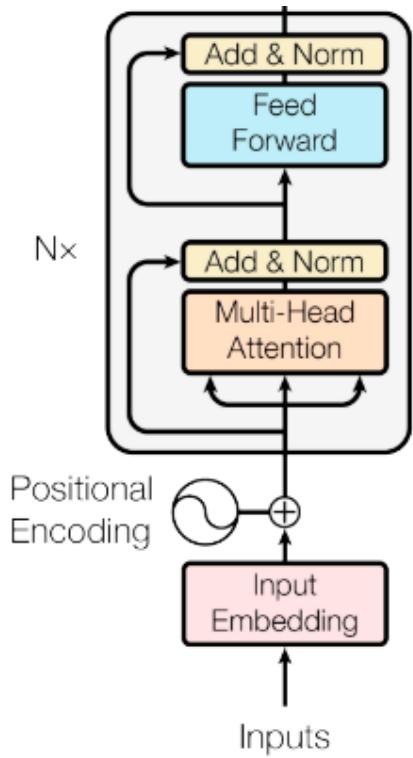


* In all encoders other than #0,
we don't need embedding.
We start directly with the output
of the encoder right below this one



$$\text{softmax}\left(\frac{\mathbf{Q} \times \mathbf{K}^T}{\sqrt{d_k}}\right) \mathbf{V} = \mathbf{Z}$$

Transformer encoder

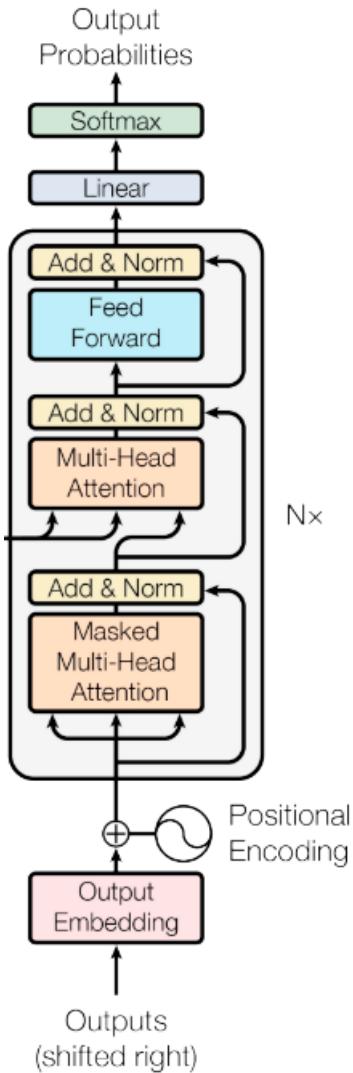


The encoder consists of $N = 6$ identical layers

Each encoder layer has 2 sub-layers: multi-head attention and fully connected feed-forward network.

Each sub-layer has a residual connection around it, followed by layer normalization.

Transformer decoder

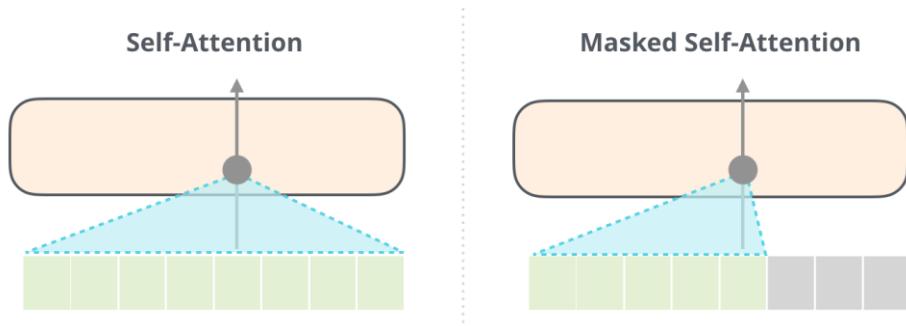


The decoder also consists of $N = 6$ identical layers

- A decoder layer is identical to the encoder layer,
- It has an additional 3rd sub-layer,
- Performs multi-head attention over the output of the encoder.

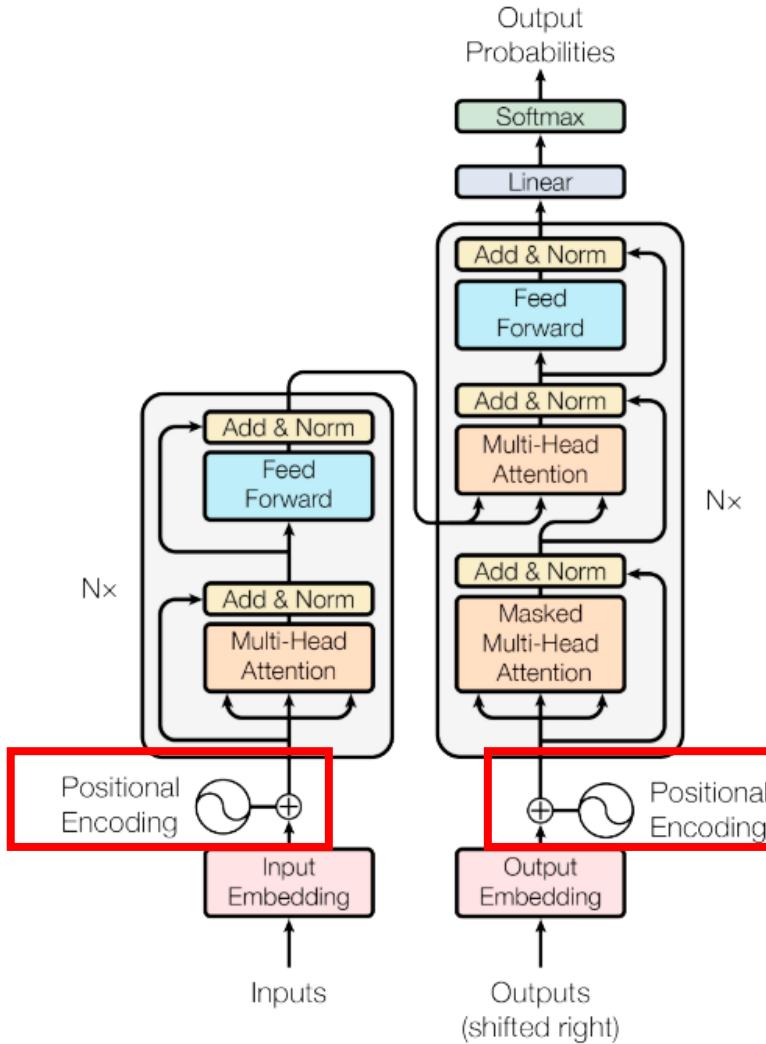
The masked self-attention sub-layer in the decoder prevents positions from attending to subsequent positions.

- the predictions for position i can depend only on the known outputs at positions $< i$.



Position encoding

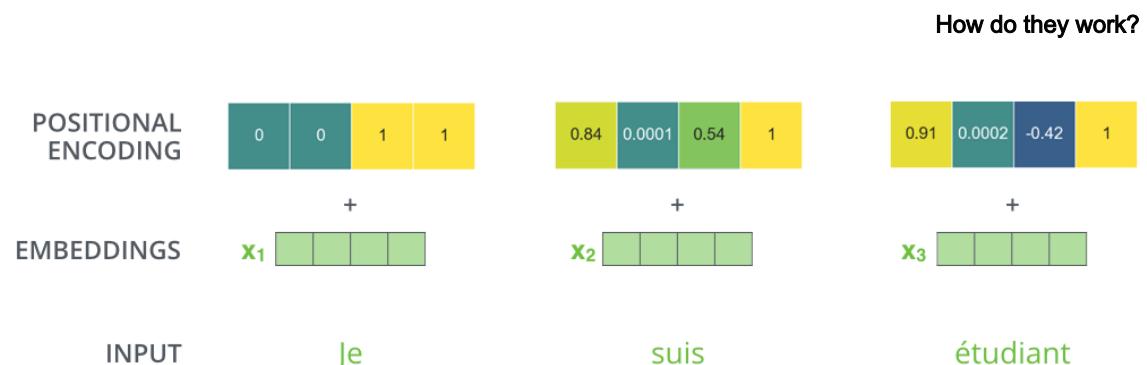
attention is permutation invariant but orders matter and it should matter



Attention is a permutation-invariant operation.

A pure attention module will return the same output regardless of the order of its inputs.

Solution: Positional encodings are added to the input in order to make use of the order of the sequence.



Positional encoding in the transformer

Intuitively: Positional encodings follow a specific pattern that the model learns

To determine the position of each word / the distance between words in the sequence.

Encode spatial, temporal, and modality identity... they can be learned or fixed.

The original Transformer uses sine and cosine functions of different frequencies:

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

Positional encoding in code

```
class PositionalEncoding(nn.Module):

    def __init__(self, d_model: int, dropout: float = 0.1, max_len: int = 5000):
        self.dropout = nn.Dropout(p=dropout)
        position = torch.arange(max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))
        pe = torch.zeros(max_len, 1, d_model)
        pe[:, 0, 0::2] = torch.sin(position * div_term)
        pe[:, 0, 1::2] = torch.cos(position * div_term)

    def forward(self, x: Tensor) -> Tensor:
        """ Args: x: Tensor, shape [seq_len, batch_size, embedding_dim] """
        x = x + self.pe[:x.size(0)]
        return self.dropout(x)
```

Pros:

- Transformer operates on data in parallel which accelerates the learning process, compared to RNN which operates sequentially
- Transformer can deal with long-term dependencies in sequences

Cons:

- Transformer scales quadratically with the number of inputs
- They are memory-intense and require lots of data and long training

It scales quadratically due to the similarity matrix which is $N \times N$. There are many better self solutions than quadratic.

Summary

Encoder-decoder is a useful architecture for many deep learning problems

Traditional encoder-decoder for NMT has the issue with the context vector

Attention mechanism overcomes the problem, by learning to select important features

Transformer is the first model that entirely relies on attention.

Recommended papers that started it all

Neural Machine Translation by Learning to Align and Translate, Bahdanu et al.
ICLR (2015)

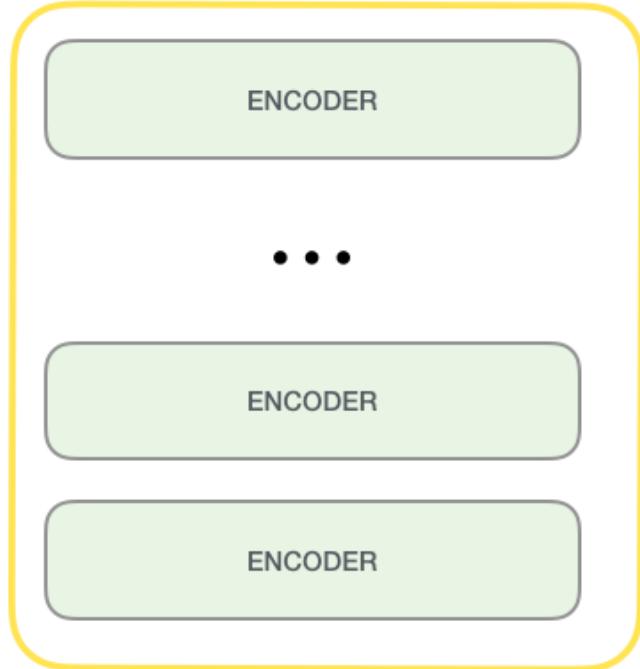
Long Short-Term Memory-Networks for Machine Reading, Cheng et al. (2016)

Show, Attend and Tell: Neural Image Caption Generation with Visual Attention,
Xu et al. (2016)

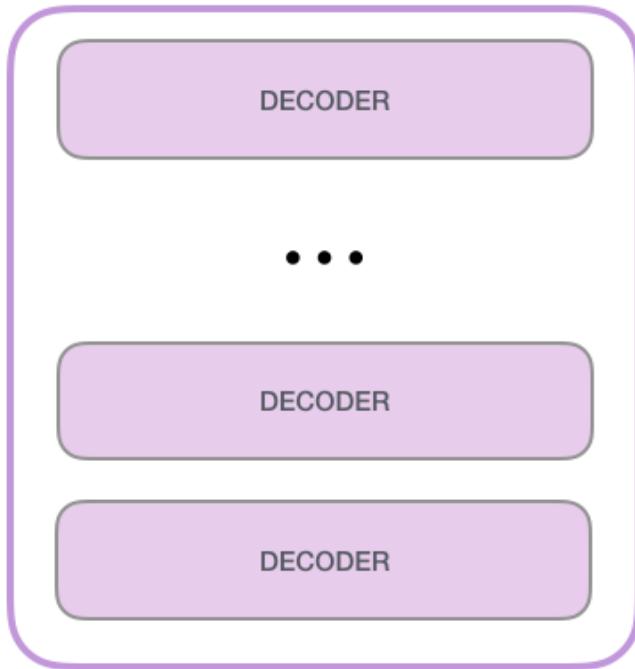
Attention Is All You Need, Vaswani et al. (2017)



BERT



GPT-2



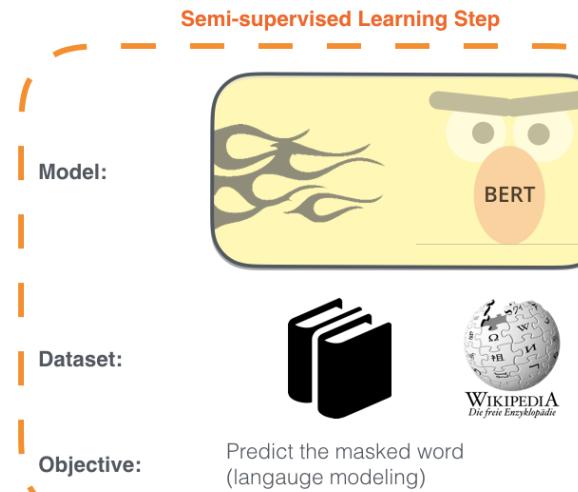
BERT: Bidirectional Encoder Representations from Transformers

Idea: pre-train bidirectional representations from unlabeled text, by jointly conditioning on both left and right context.

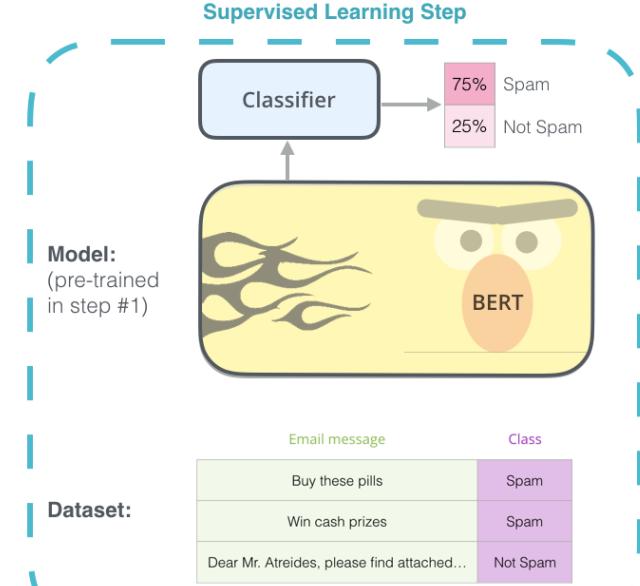
A pre-trained BERT model can be fine-tuned with just one additional output layer to create SOTA models for NLP tasks.

1 - Semi-supervised training on large amounts of text (books, wikipedia..etc).

The model is trained on a certain task that enables it to grasp patterns in language. By the end of the training process, BERT has language-processing abilities capable of empowering many models we later need to build and train in a supervised way.



2 - Supervised training on a specific task with a labeled dataset.



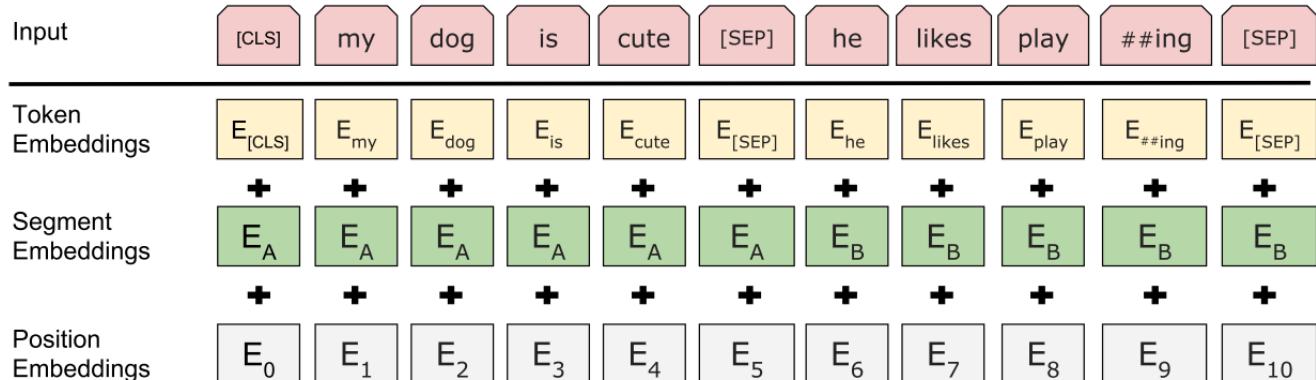
BERT input representation

The input representation: single sentence OR a pair of sentences in one sequence.

The complete input is the sum of the token embeddings, the segmentation embeddings and the position embeddings.

The first token of every sequence is always a special classification token ([CLS]).

- The final hidden state corresponding to this token is used as the aggregate sequence representation for classification tasks.



Sentence pairs are packed together into a single sequence, separated with a special token ([SEP]).

Also, a learned embedding is added to every token indicating whether it belongs to sentence A or sentence B.

BERT pre-training

Task #1: Masked Language Modelling (MLM)

- Mask a percentage of the input tokens at random with a special token [MASK], and then predict those masked tokens.

Task #2: Next Sentence Prediction (NSP)

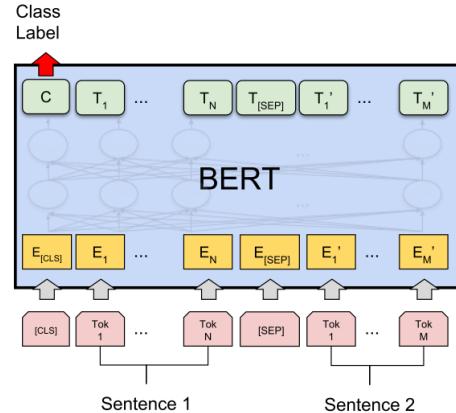
- Binary prediction whether the next sentence is a correct one.
- When choosing the sentences A and B, 50% of the time B is the actual next sentence that follows A and 50% of the time it is a random sentence from the corpus.

Datasets: BooksCorpus (800M words) and Wikipedia (2,500M words).

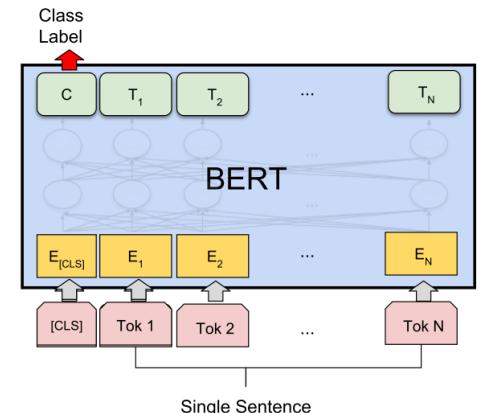
BERT fine-tuning

Fine-tuning is straightforward since the self-attention mechanism allows BERT to model many downstream tasks.

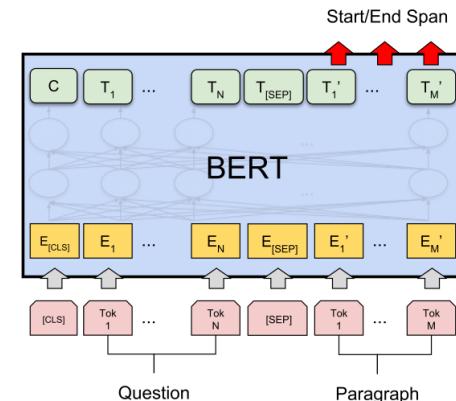
It is relatively inexpensive compared to pre-training.



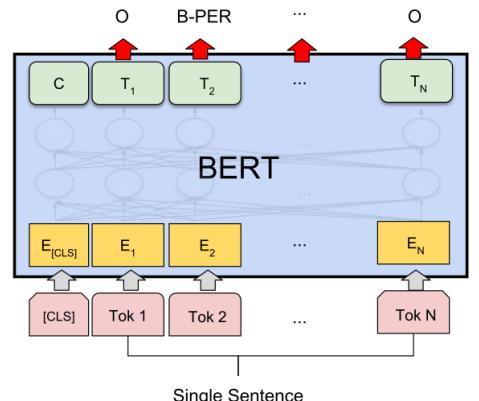
(a) Sentence Pair Classification Tasks:
MNLI, QQP, QNLI, STS-B, MRPC,
RTE, SWAG



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



(c) Question Answering Tasks:
SQuAD v1.1



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

A family of BERT models

RoBERTa: A Robustly Optimized BERT Pretraining Approach,

- More data, Longer training, Larger batches

ALBERT: A Lite BERT for Self-supervised Learning of Language Representations

DeBERTa

DistilBERT

CamamBERT

RoBERT

ClinicalBERT

GPT

Generative Pretraining by Transformers == GPT¹

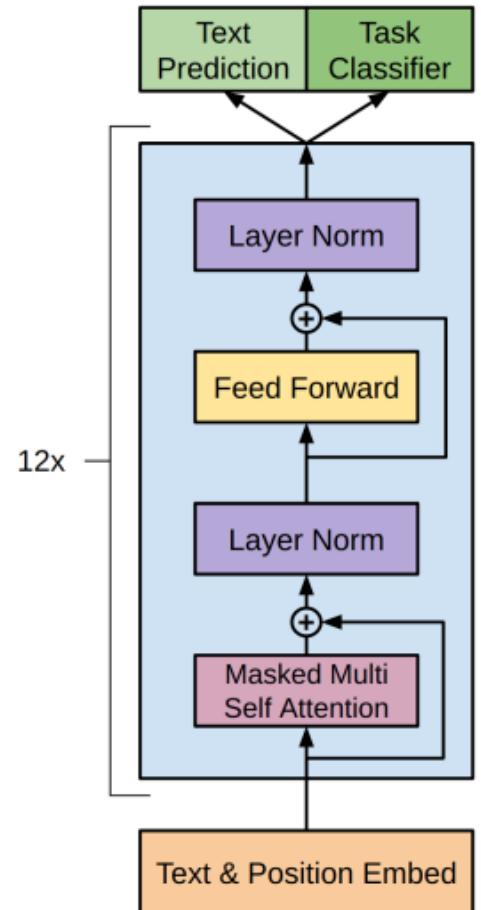
- A pre-trained unidirectional Transformer decoder

The idea: To train a generative language model using “unlabeled” data.

And then fine-tune it on specific downstream tasks.

Unsupervised pre-training:

- Given an unsupervised corpus of tokens, use a standard language modeling objective (to predict the next token in the sequence given previous tokens) .



GPT 1 to 3

GPT1¹ : Proves that language modeling serves as an effective pre-training objective which helps the model to generalize well

GPT2² : uses a larger dataset for training and adds additional parameters to build a stronger language model.

GPT3³ : even larger than GPT2, can automatically generate high-quality paragraphs.

- Performs well on tasks on which it was never explicitly trained on, like writing SQL queries and codes given natural language description of task.

For text and images we can just scale and we keep improving. The architecture is always pretty much the same

	GPT-1	GPT-2	GPT-3
Parameters	117 Million	1.5 Billion	175 Billion
Decoder Layers	12	48	96
Hidden Layer	768	1600	12288
Batch Size	64	512	3.2M

¹ Improving Language Understanding by Generative Pre-Training, Radford et al. (2018)

² Language Models are Unsupervised Multitask Learners, Radford et al. (2019)

³ Language Models are Few-Shot Learners, Radford et al. (2020)

GPT enables in-context learning

The three settings we explore for in-context learning

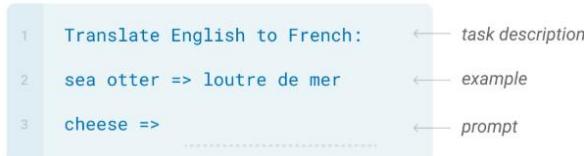
Zero-shot

The model predicts the answer given only a natural language description of the task. No gradient updates are performed.



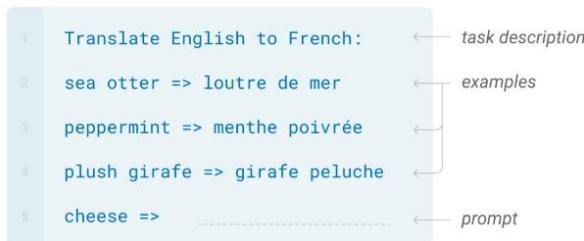
One-shot

In addition to the task description, the model sees a single example of the task. No gradient updates are performed.



Few-shot

In addition to the task description, the model sees a few examples of the task. No gradient updates are performed.



Traditional fine-tuning (not used for GPT-3)

Fine-tuning

The model is trained via repeated gradient updates using a large corpus of example tasks.



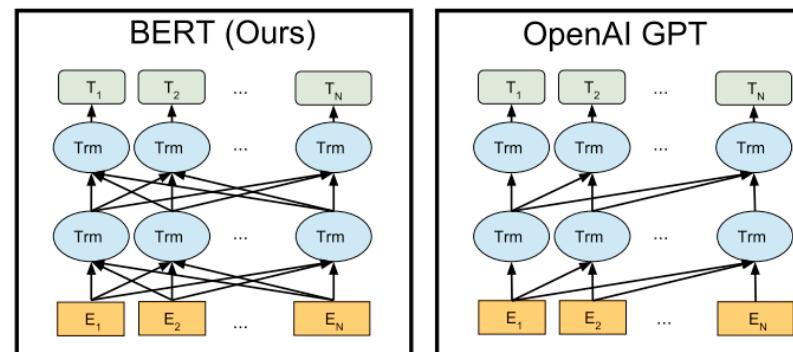
GPT versus BERT

The key difference between the BERT and GPT, is that GPT is a unidirectional Transformer decoder, whereas BERT is bidirectional Transformer encoder.

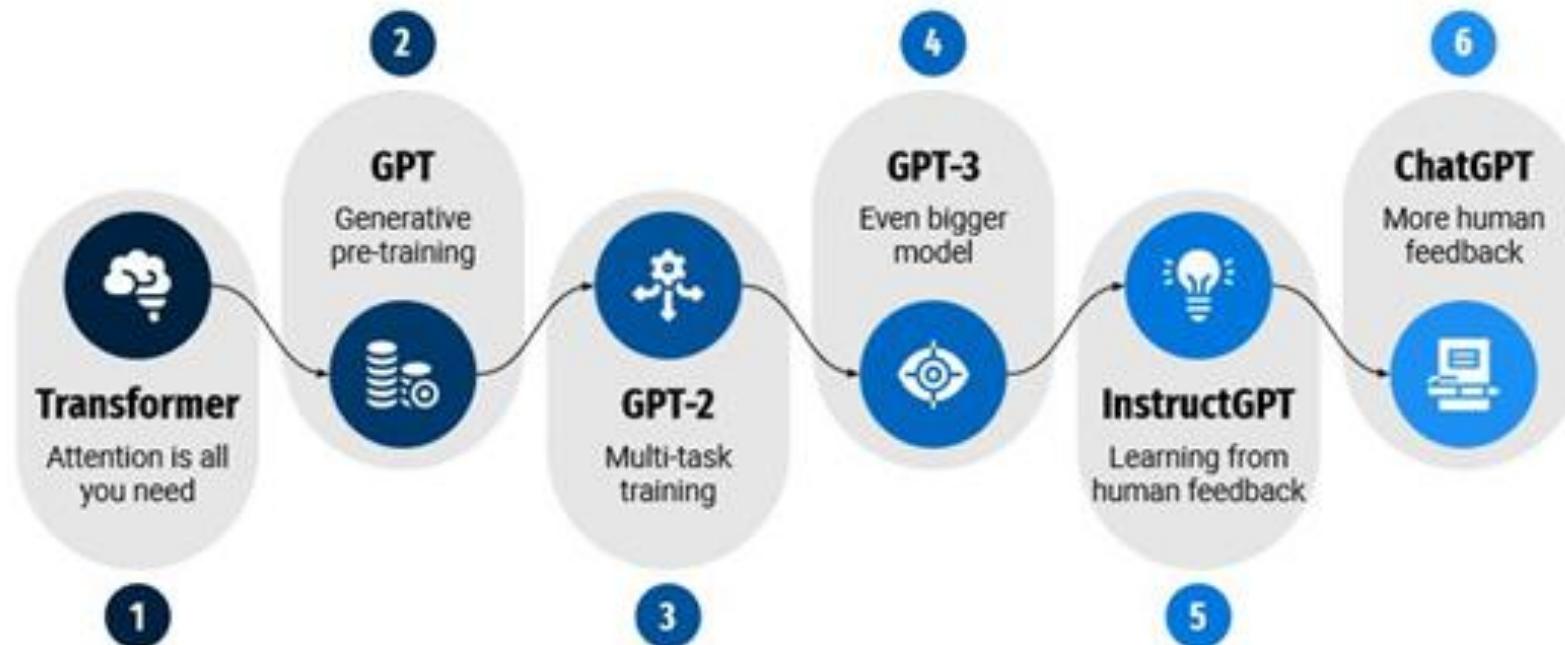
GPT outputs one token at a time, just like traditional language models.

- *After each token is produced, that token is added to the sequence of inputs.*
- *That new sequence becomes the input to the model in its next step.*
- *This is an idea called “auto-regression”.*

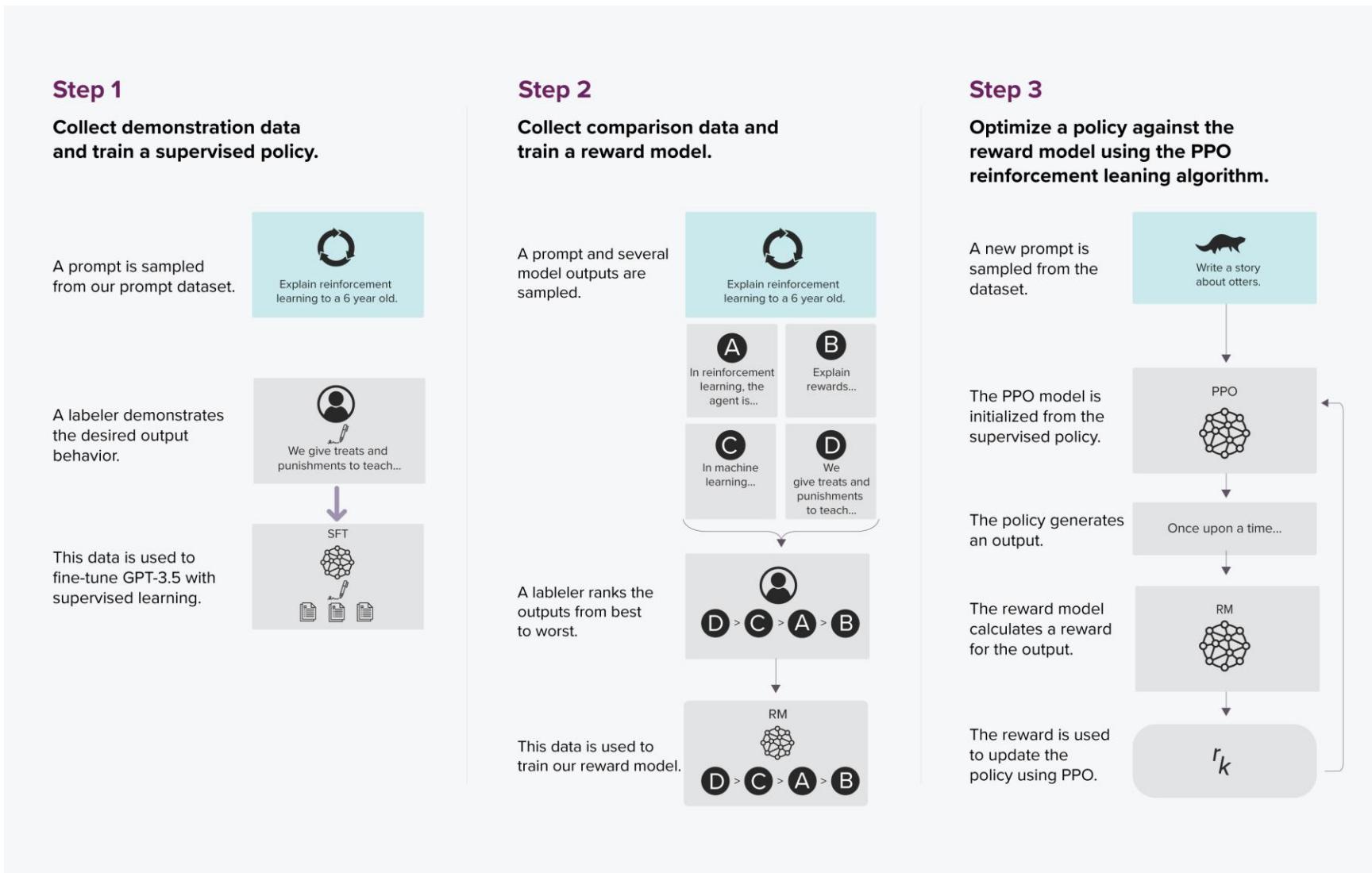
In losing auto-regression, BERT gained the ability to incorporate the context on both sides of a word.



What is ChatGPT?



Reinforcement learning from human feedback



Why is ChatGPT so convincing

It is a result of exposure to extreme scale.

It's network consists of billions of parameters.

It is a product of careful and elaborate hyperparameter tuning.

Human feedback RL creates output formats that we like.

In short, ChatGPT has condensed the internet and gives it to us in a pleasing narrative style.

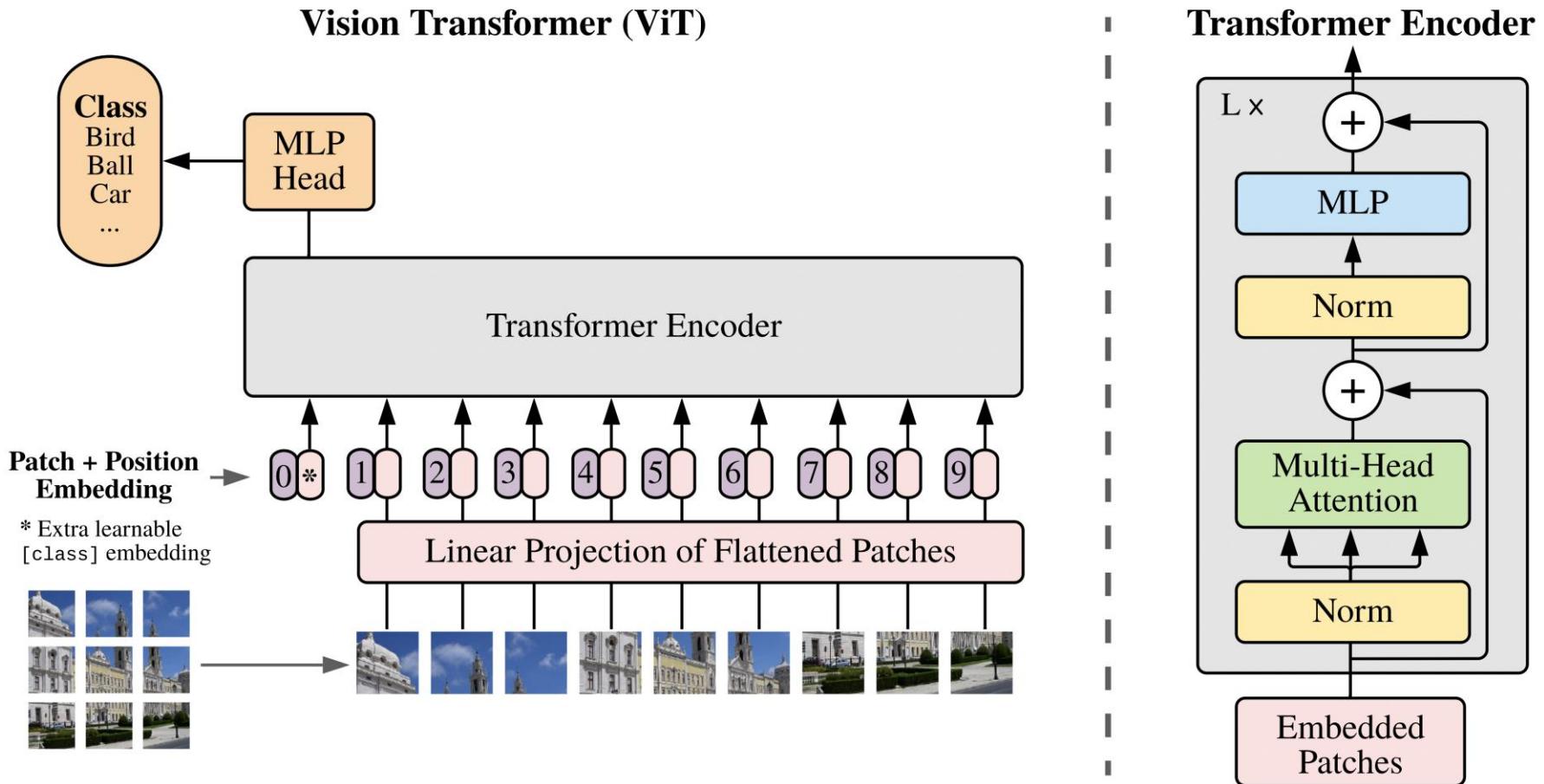
Vision transformers

Attention for text is intuitive, but it has also proven to be important for vision.

Transformers assume the input is a sequence of tokens.

What is a token in the context of an image?

Vision Transformer (ViT)



The linear projection is a fully connected layer, that transforms the patches into feature vectors

Vision Transformer (ViT)

Like BERT’s [CLS] token, a learnable embedding is prepended to the sequence of embedded patches:

- the classification is done on this token (with an MLP)

“Position encodings” are added to the patch embeddings to retain positional information. (attention by itself doesn’t have any notion of ordering/space)

- These vectors are also simply learned

```
# pos_embed has entry for class token, concat then add
if self.cls_token is not None:
    x = torch.cat((self.cls_token.expand(x.shape[0], -1, -1), x), dim=1)
x = x + self.pos_embed
```

Attention versus convolution

Convolution is local and shared.

Attention is global.

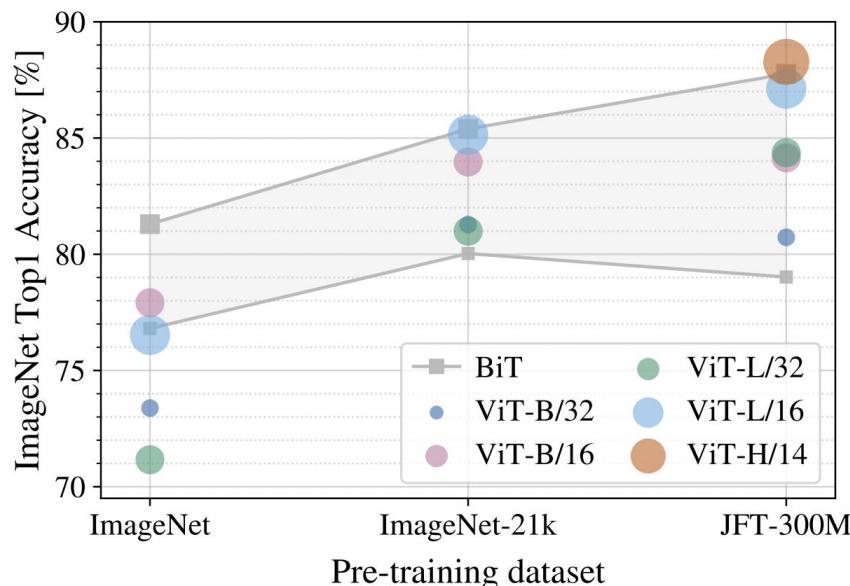
Hence both bring different views to visual representation learning.

Transformers impose less structure, useful in large-scale settings.

Training a ViT is more difficult

Original paper required pre-training on ImageNet-22k (14M images) to achieve good performances.

DeiT (data efficient image transformers) paper showed training with ImageNet-1k possible if more augmentations and regularisations are used.

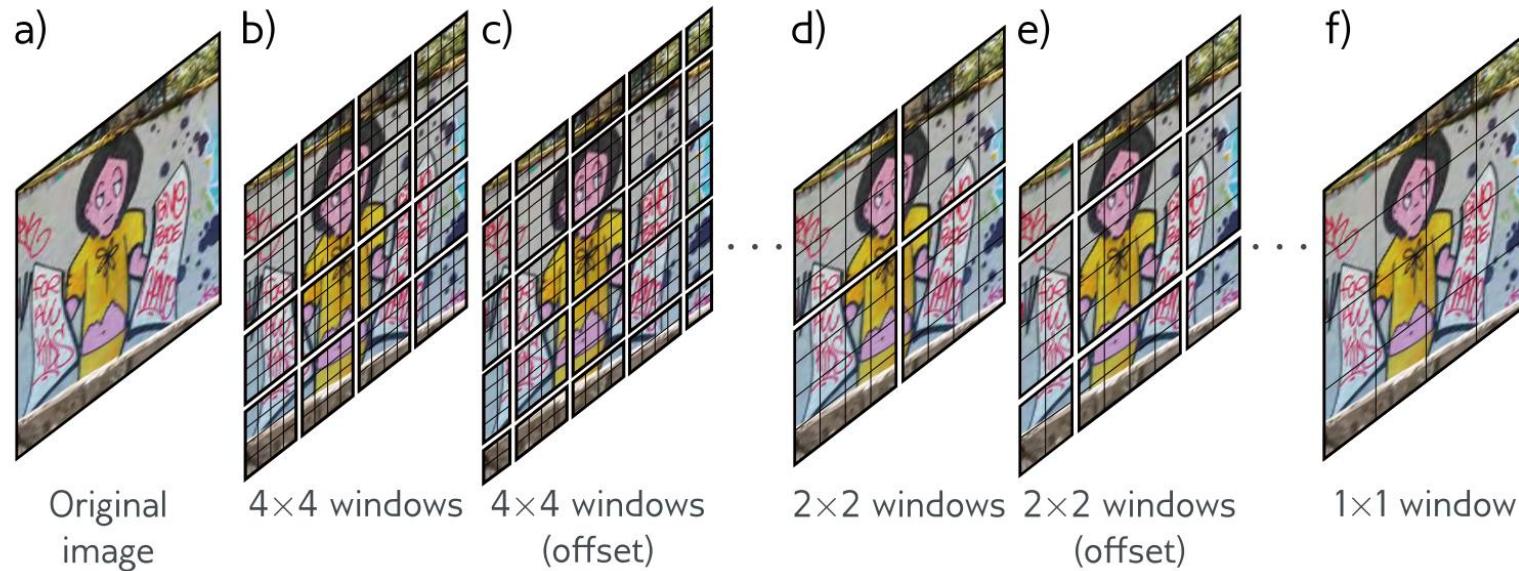


Ablation on ↓			Pre-training				Fine-tuning				Rand-Augment				top-1 accuracy			
	none: DeiT-B	adamw	adamw	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	Exp. Moving Avg.		
optimizer	SGD	adamw	adamw	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✗	74.5	77.3		
	adamw	SGD	SGD	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✗	81.8	83.1		
data augmentation	adamw	adamw	adamw	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✗	79.6	80.4		
	adamw	adamw	adamw	✓	✗	✗	✓	✓	✓	✓	✓	✓	✓	✗	81.2	81.9		
	adamw	adamw	adamw	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✗	78.7	79.8		
	adamw	adamw	adamw	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓	✗	80.0	80.6		
regularization	adamw	adamw	adamw	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓	✗	75.8	76.7		
	adamw	adamw	adamw	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✗	4.3*	0.1		
	adamw	adamw	adamw	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✗	3.4*	0.1		
	adamw	adamw	adamw	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✗	76.5	77.4		
	adamw	adamw	adamw	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	81.3	83.1		
	adamw	adamw	adamw	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	81.9	83.1		

Swin Transformer: return of locality

Idea: mostly looking at “local” neighborhood, so can save some computation
(remember attention is $O(n^2)$) or gain some accuracy by modelling this.

Strong performance but slow models.



Convolutions and attention are complimentary

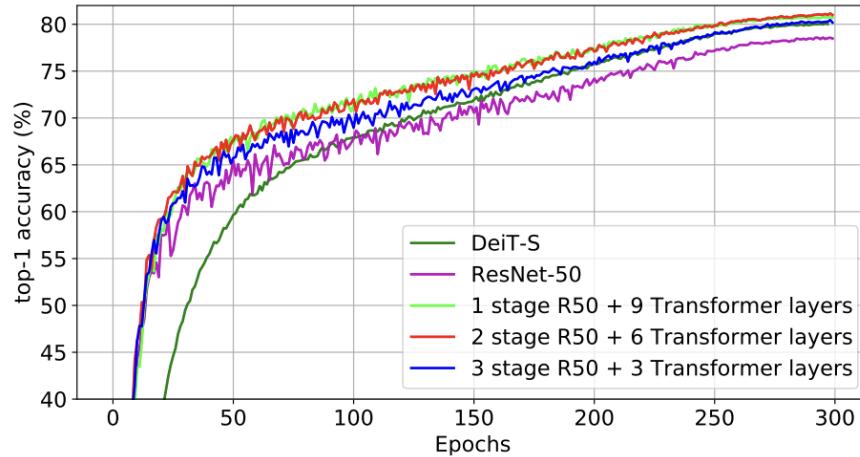
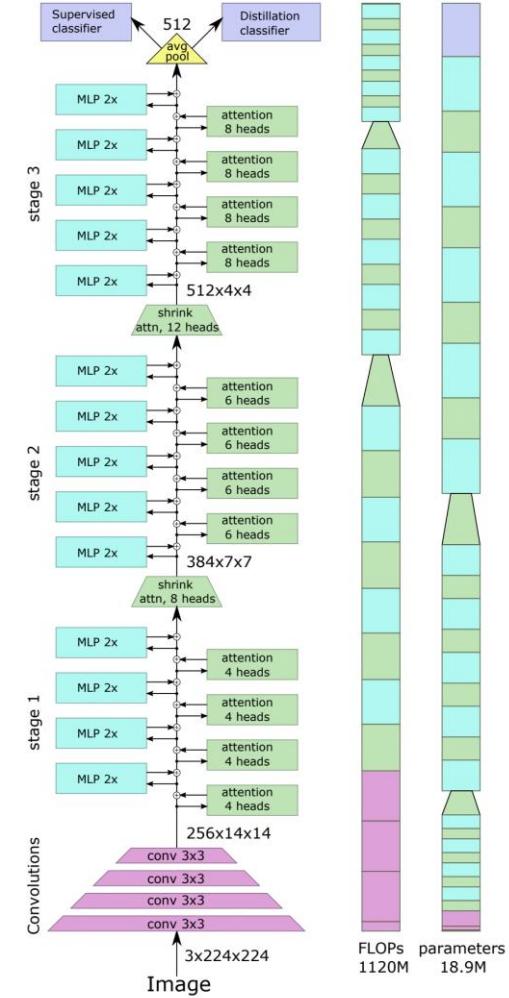


Figure 3: Models with convolutional layers show a faster convergence in the early stages compared to their DeiT counterpart.



Summary

Sequential modelling

Attention and self-attention

Transformers

Language and vision transformers

Next lecture

Lecture	Title	Lecture	Title
1	Intro and history of deep learning	2	AutoDiff
3	Deep learning optimization I	4	Deep learning optimization II
5	Convolutional deep learning	6	Attention-based deep learning
7	Graph deep learning	8	From supervised to unsupervised deep learning
9	Multi-modal deep learning	10	Generative deep learning
11	What doesn't work in deep learning	12	Non-Euclidean deep learning
13	Q&A	14	Deep learning for videos

Learning and reflection

Understanding Deep Learning, Chapter 12

Thank you!



Deep Learning 1

2025-2026 – Pascal Mettes

Lecture 7

Graph Deep Learning

Previous lecture

Lecture	Title	Lecture	Title
1	Intro and history of deep learning	2	AutoDiff
3	Deep learning optimization I	4	Deep learning optimization II
5	Convolutional deep learning	6	Attention-based deep learning
7	Graph deep learning	8	From supervised to unsupervised deep learning
9	Multi-modal deep learning	10	Generative deep learning
11	What doesn't work in deep learning	12	Non-Euclidean deep learning
13	Q&A	14	Deep learning for videos

I have to leave a 10:10 on the dot to be able to make it to a PhD committee at 11:00 in the city center.

This lecture

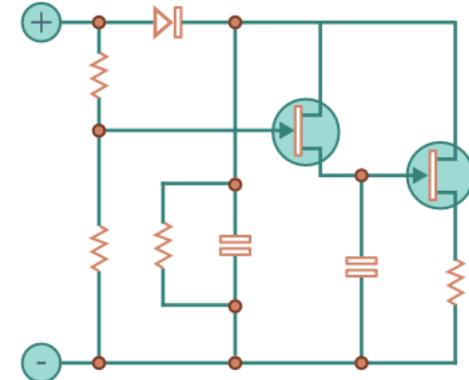
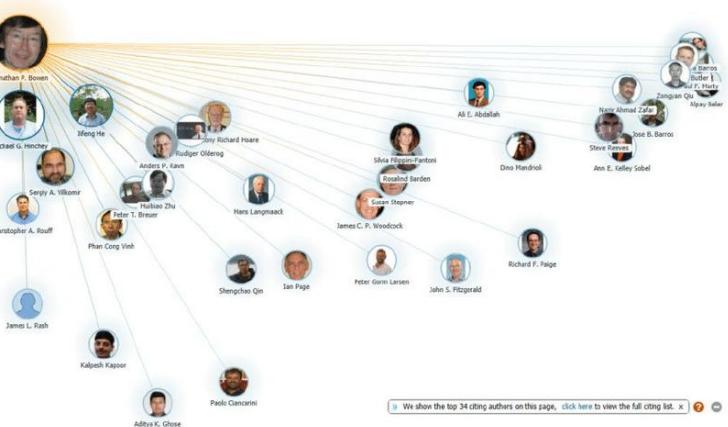
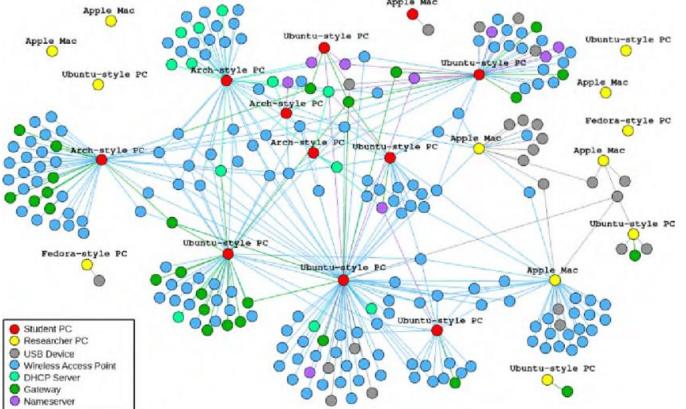
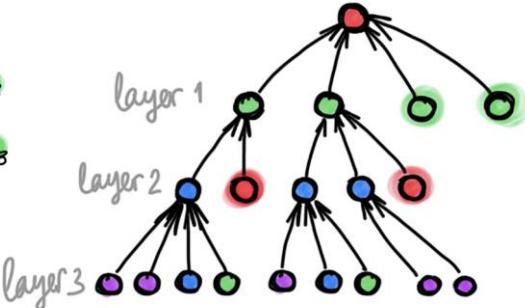
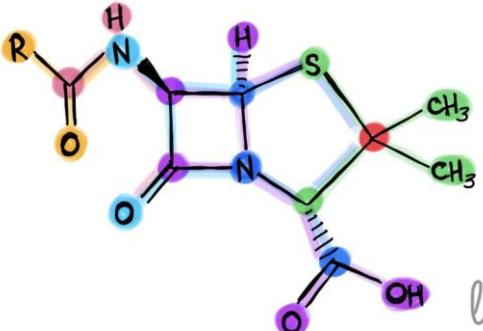
Graphs

Graph convolutions

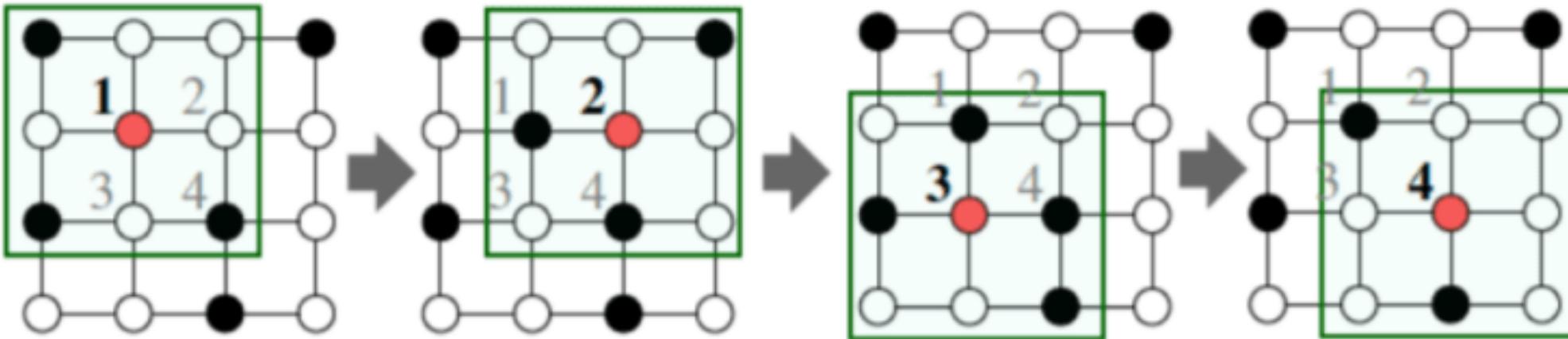
Graph attention

Graph applications

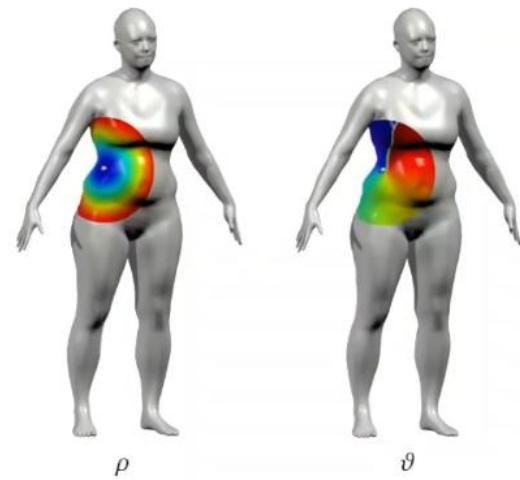
Graphs, more common than you think



Many structures are special cases of graphs

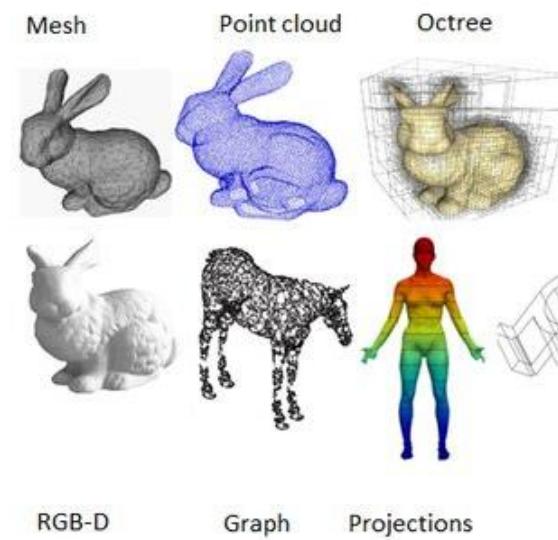
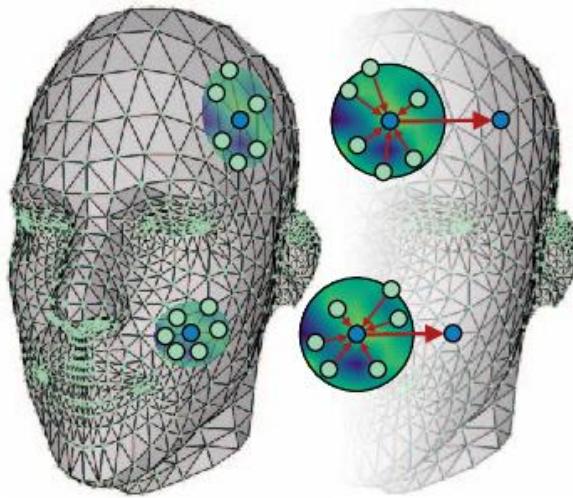
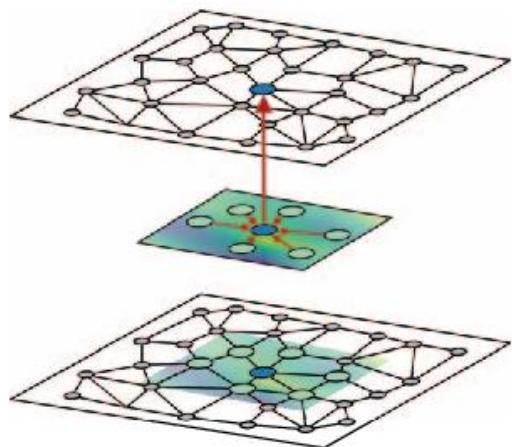
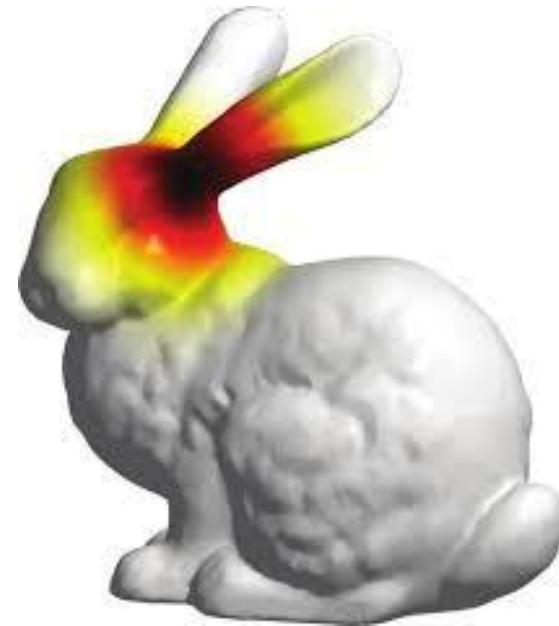


Graphs as geometry



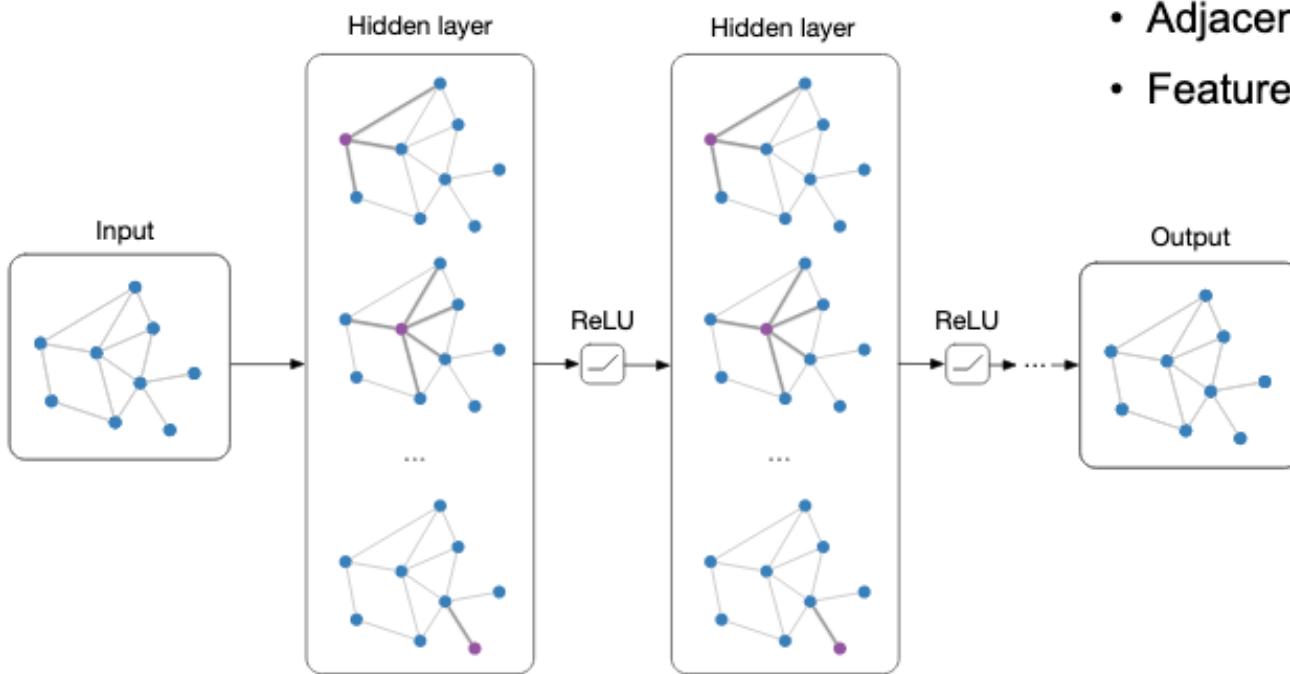
ρ

ϑ



What are graph networks?

The bigger picture:



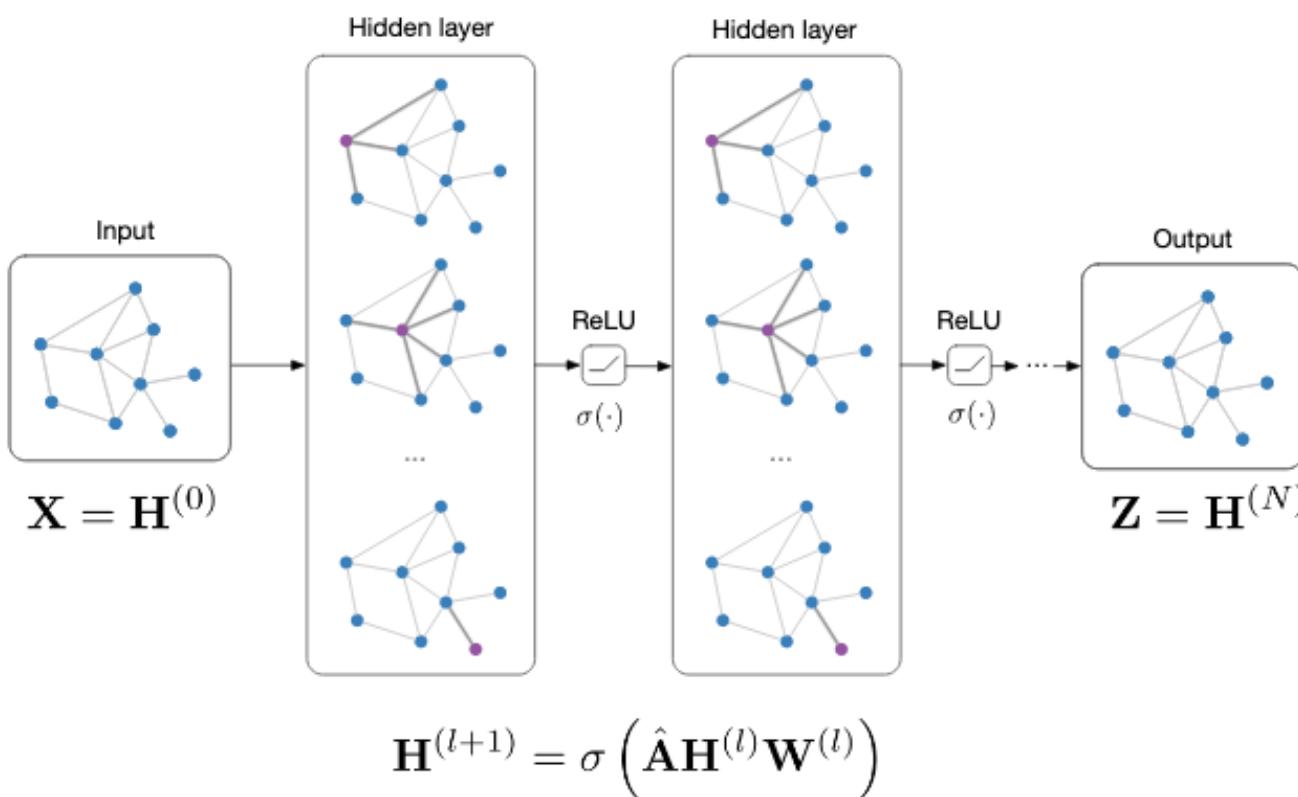
Notation: $\mathcal{G} = (\mathbf{A}, \mathbf{X})$

- Adjacency matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$
- Feature matrix $\mathbf{X} \in \mathbb{R}^{N \times F}$

Main idea: Pass messages between pairs of nodes & agglomerate

Graph networks

Input: Feature matrix $\mathbf{X} \in \mathbb{R}^{N \times E}$, preprocessed adjacency matrix $\hat{\mathbf{A}}$



Node classification:

$$\text{softmax}(\mathbf{z}_n)$$

e.g. Kipf & Welling (ICLR 2017)

Graph classification:

$$\text{softmax}(\sum_n \mathbf{z}_n)$$

e.g. Duvenaud et al. (NIPS 2015)

Link prediction:

$$p(A_{ij}) = \sigma(\mathbf{z}_i^T \mathbf{z}_j)$$

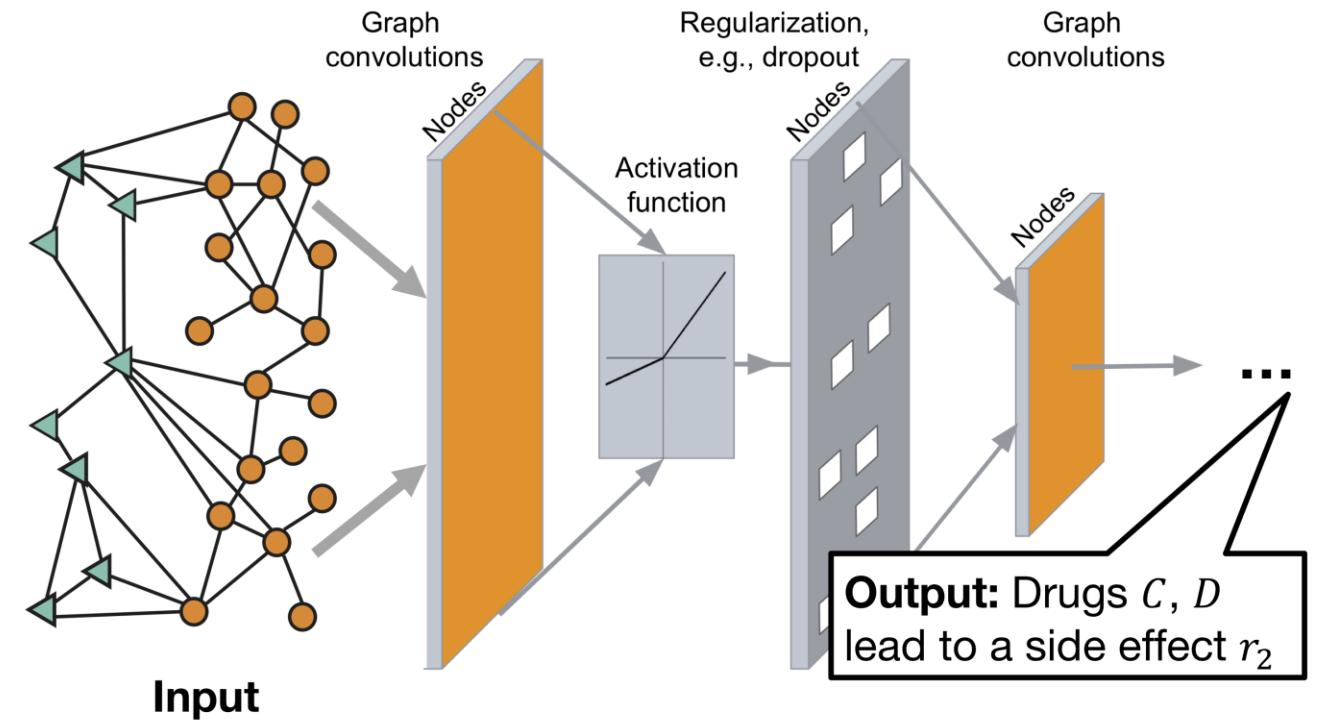
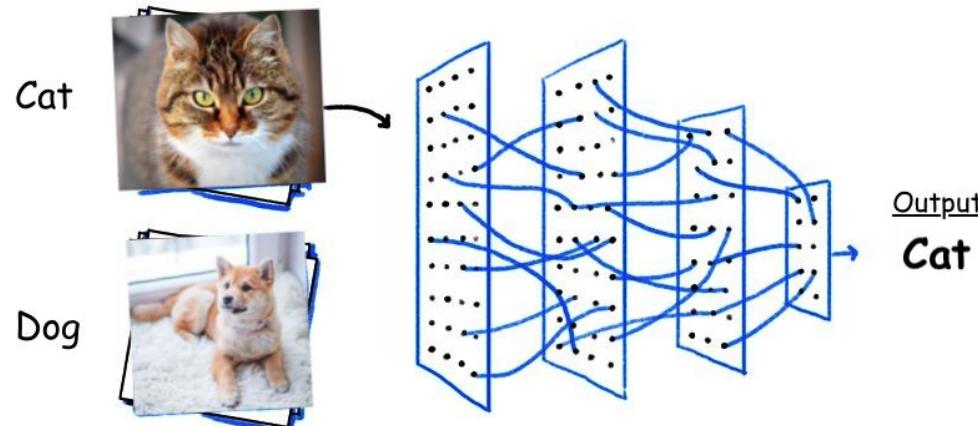
Kipf & Welling (NIPS BDL 2016)

“Graph Auto-Encoders”

1) Graph classification

Make a prediction over the entire graph.

Akin to assigning a label to an entire image.



2) Node classification

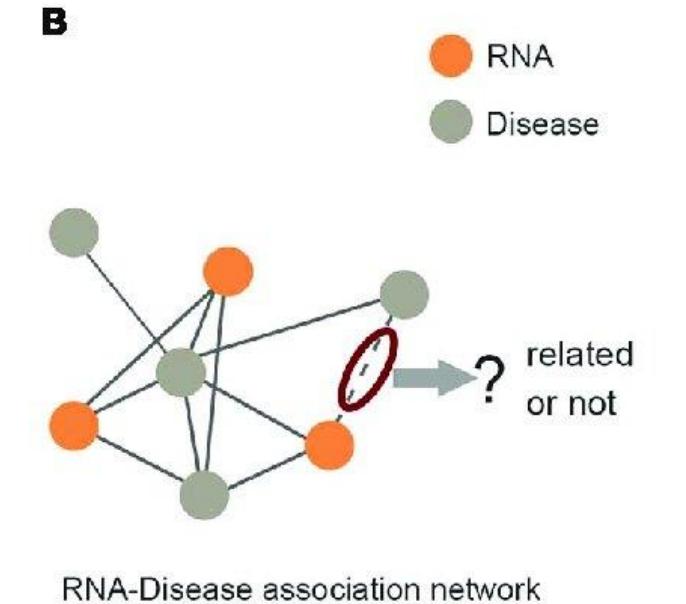
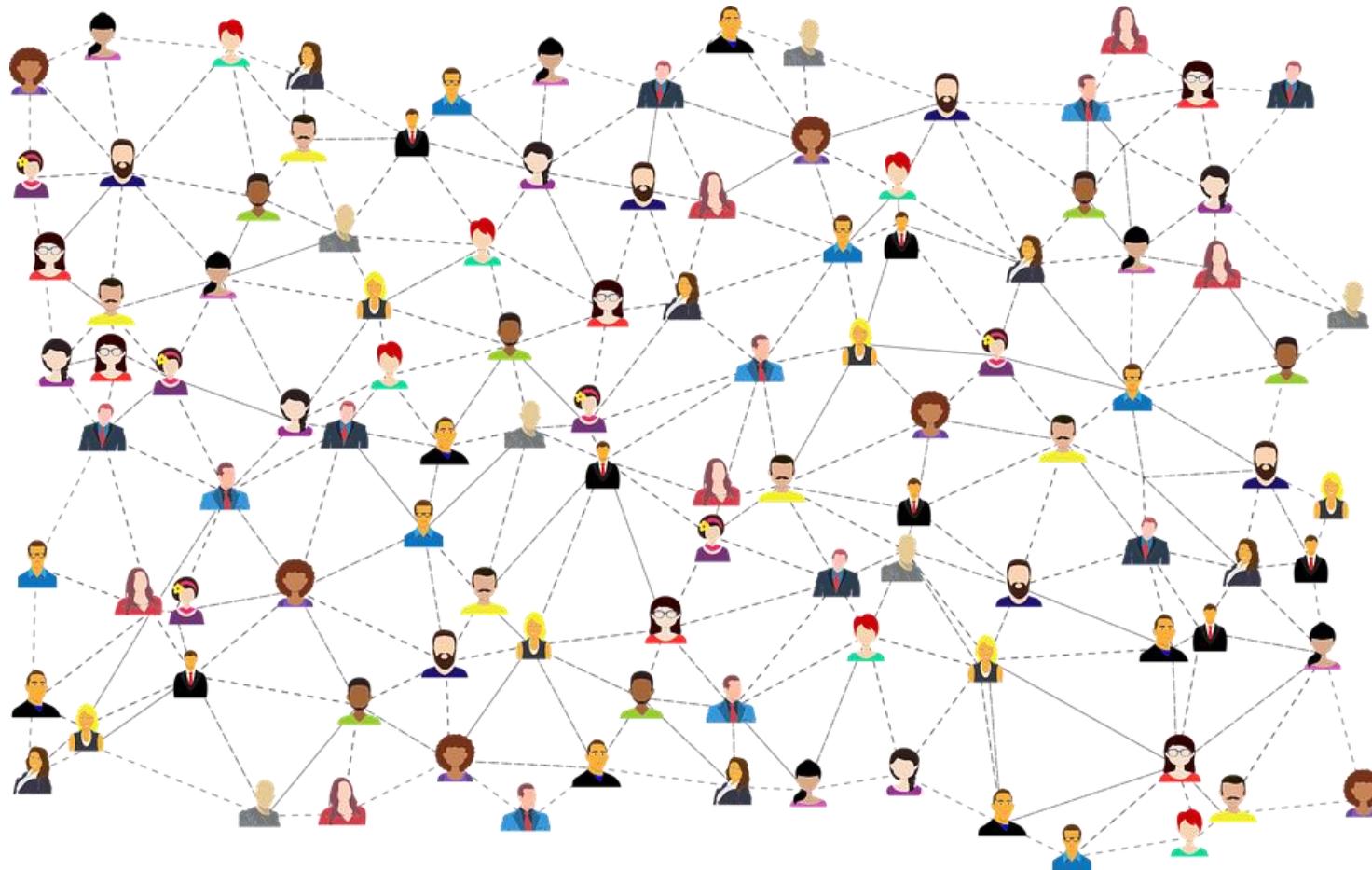
Make a prediction for each individual node.

Akin to segmentation for images.



3) Link prediction

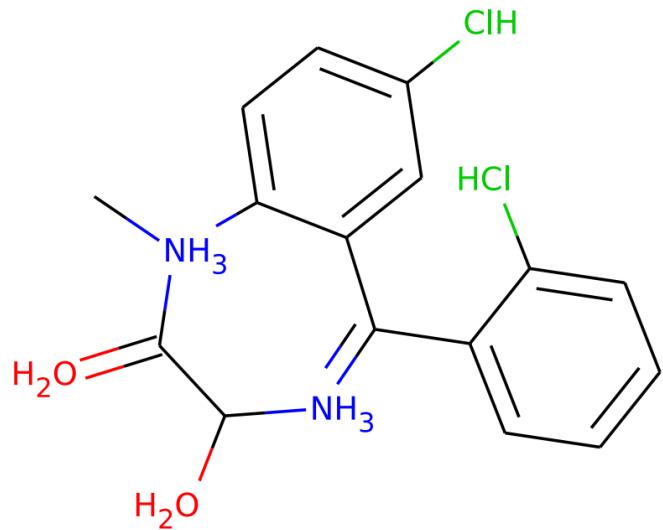
Make a prediction for each edge between two nodes.



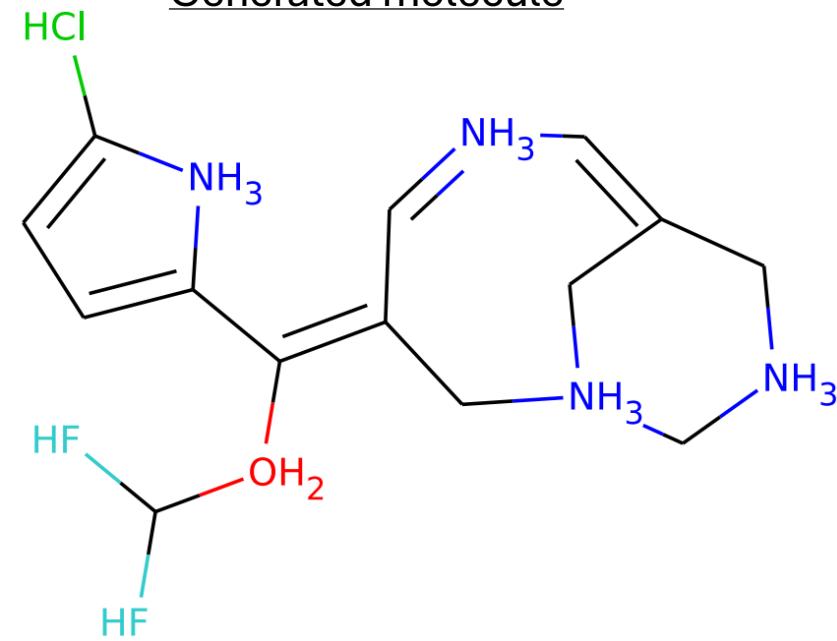
4) Graph generation

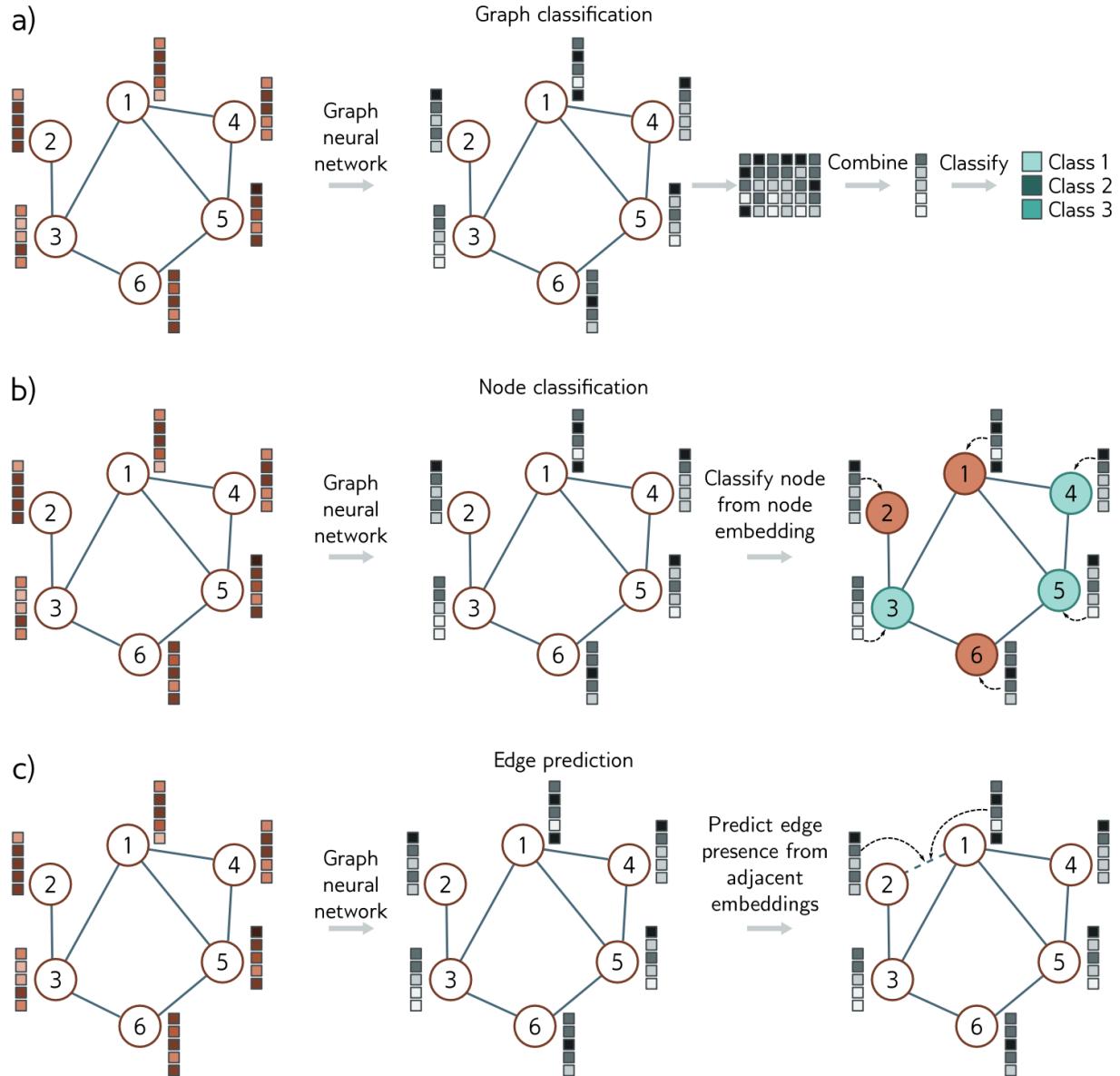
Similar in spirit to image/text generation, topic of next week.

Example molecule



Generated molecule

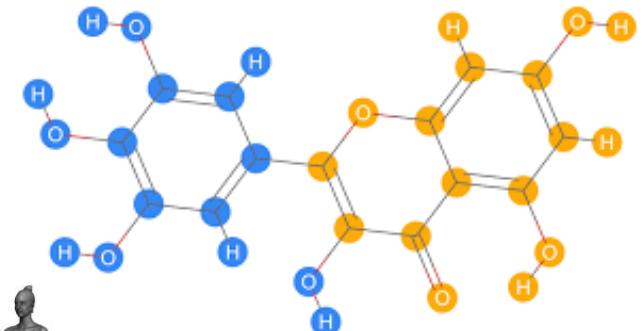




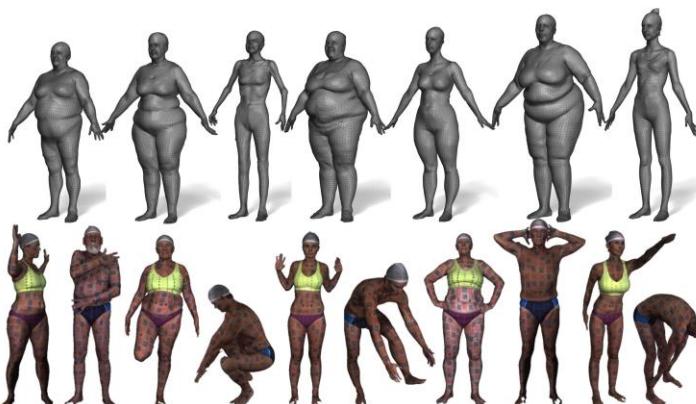
Graphs can be dynamic

Graphs have fixed structures.

but the features are very variable, and also this is an assumption, the graph structure could change, nodes and edges could appear and disappear



But many are subject to change.



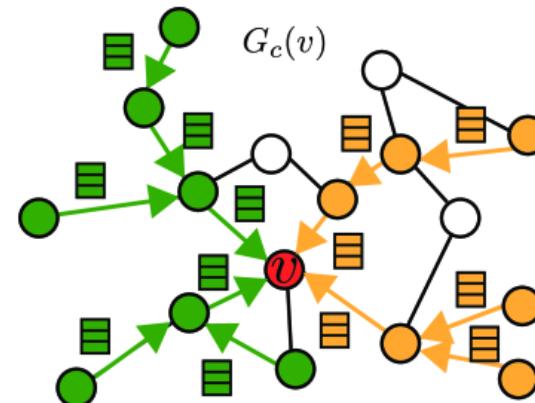
In practice, this change can even be gradual and continuous.

Regular structures and graphs

Regular structures are a subset of graphs. I.e., images are grid graphs.



- Convolution + pooling
- Local neighborhood: fixed window
- Constant number of neighbors
- With fixed ordering
- Translation equivariance



- Message passing + coarsening
- Local neighborhood: 1-hop
- Different number of neighbors
- No ordering of neighbors
- Local permutation equivariance

Labelled graph	Degree matrix	Adjacency matrix	Laplacian matrix
<pre> graph LR 6((6)) --- 4((4)) 4 --- 5((5)) 4 --- 3((3)) 5 --- 1((1)) 3 --- 2((2)) </pre>	$\begin{pmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$	$\begin{pmatrix} 2 & -1 & 0 & 0 & -1 & 0 \\ -1 & 3 & -1 & 0 & -1 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3 & -1 & -1 \\ -1 & -1 & 0 & -1 & 3 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 \end{pmatrix}$

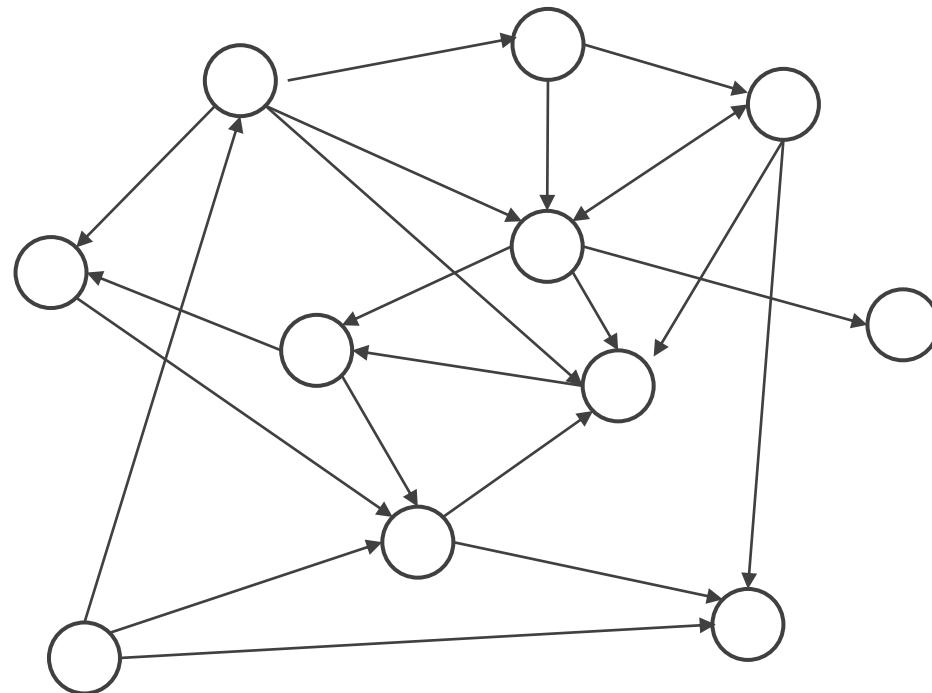
Definition of a graph

(in deep learning)

Directed graphs

Vertices $\mathcal{V} = \{1, \dots, n\}$, also called “nodes”

Edges $\mathcal{E} = \{(i, j) : i, j \in \mathcal{V}\} \subseteq \mathcal{V} \times \mathcal{V}$ (directed)

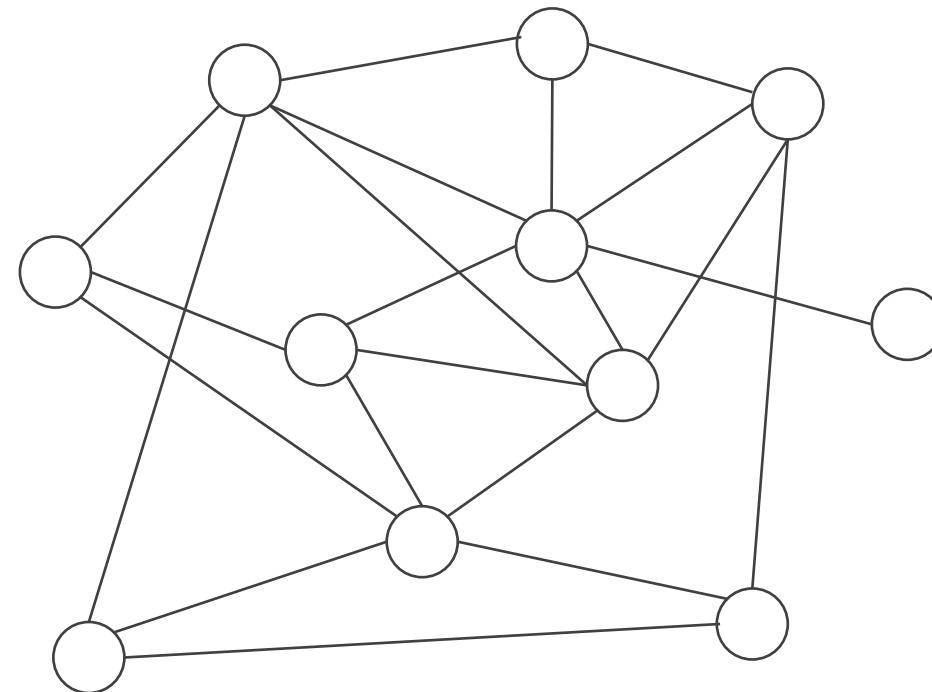


Undirected graphs

Vertices $\mathcal{V} = \{1, \dots, n\}$

Edges $\mathcal{E} = \{(i, j) : i, j \in \mathcal{V}\} \subseteq \mathcal{V} \times \mathcal{V}$ (directed)

Edges $\mathcal{E} = \{\{i, j\} : i, j \in \mathcal{V}\} \subseteq \mathcal{V} \times \mathcal{V}$ (undirected)



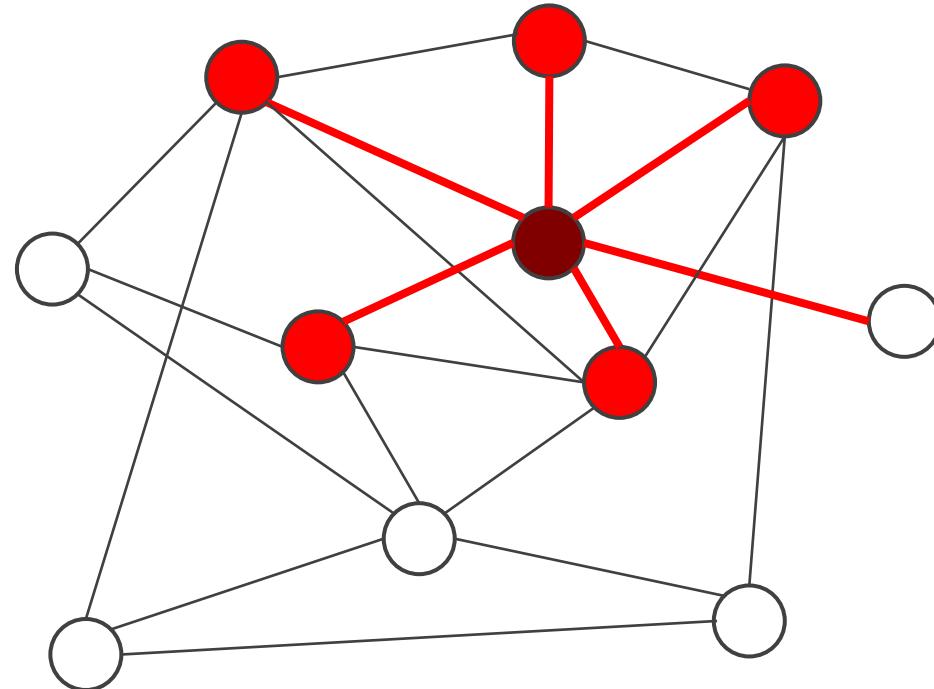
Graph neighborhood

The neighborhood of a node consists of all nodes directly connected to it

$$\mathcal{N}(i) = \{j: (i, j) \in \mathcal{E}\}$$

The **degree** of a node is the number of neighbors: $d_i = |\mathcal{N}(i)|$

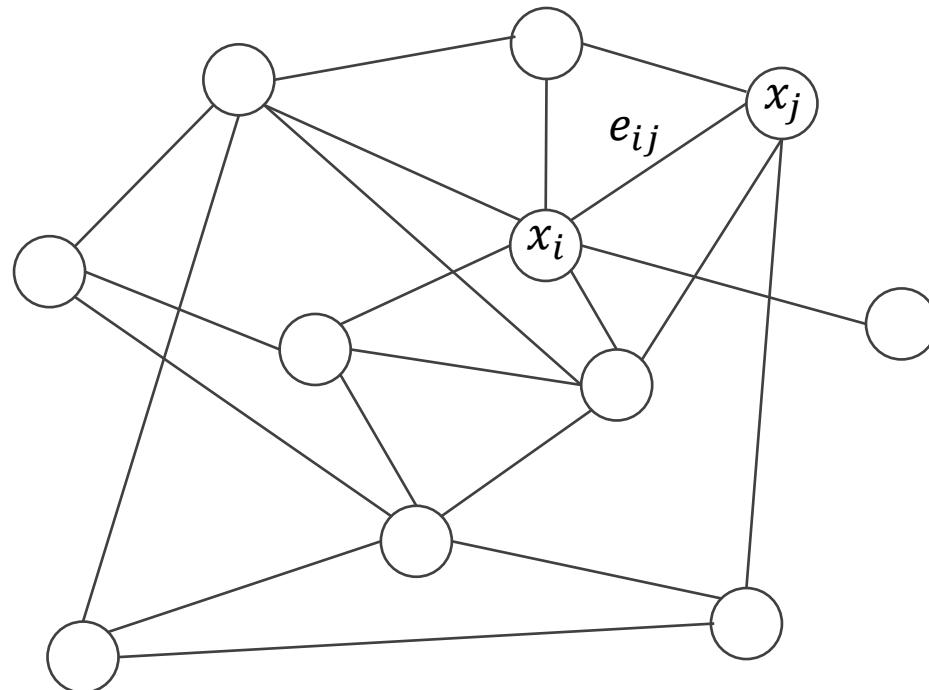
The diagonal matrix D contains all degrees per node



Attributes

Node features $\mathbf{x}: \mathcal{V} \rightarrow \mathbb{R}^d, X = (\mathbf{x}_1, \dots, \mathbf{x}_n)$

Edge features $e_{ij}: \mathcal{E} \rightarrow \mathbb{R}^{d'}$



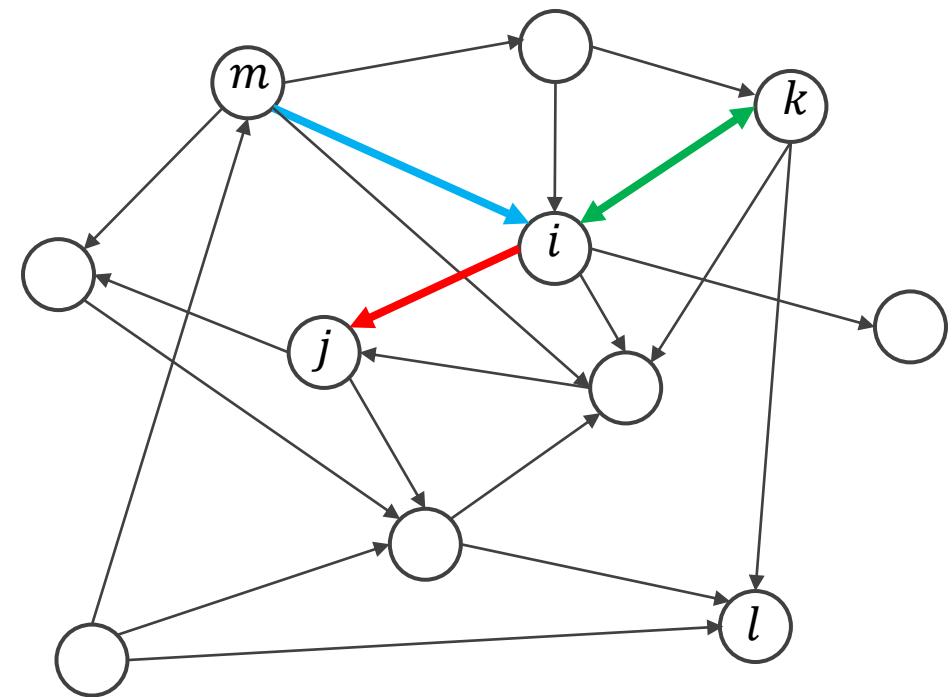
Adjacency matrix

An $n \times n$ matrix A , for n nodes

$$A_{ij} = \begin{cases} 1 & \text{if } (i,j) \in \varepsilon \\ 0 & \text{if } (i,j) \notin \varepsilon \end{cases}$$

$(A^z)_{ij}$: number of paths that go from i to j in z steps

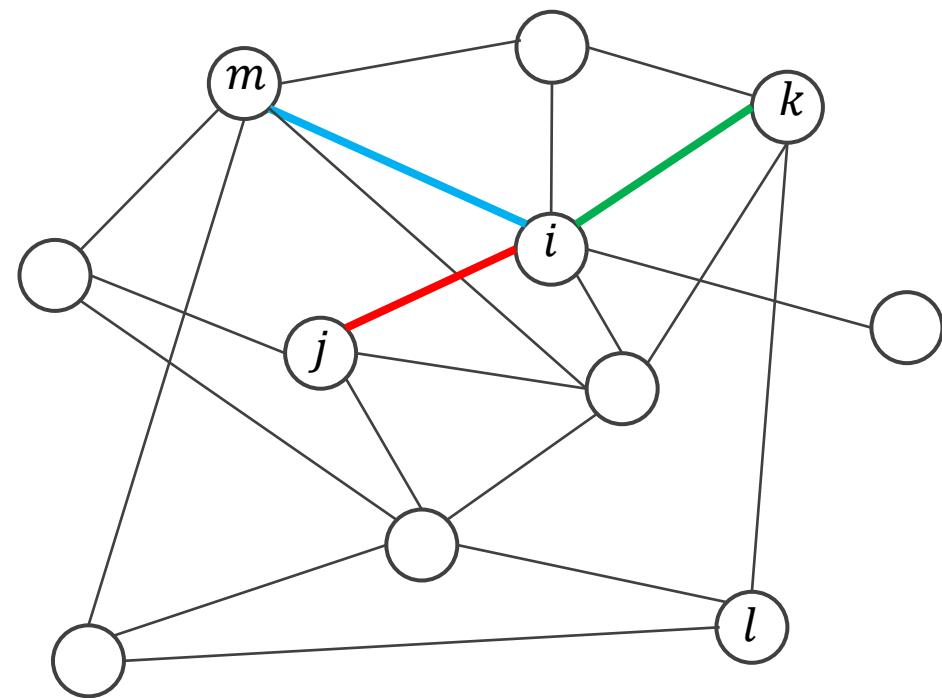
	i	j	k	l	m
i		1	1	0	
j					
k	1				
l	0			1	
m					



Adjacency matrix for undirected graphs

The adjacency matrix is symmetric for undirected graphs.

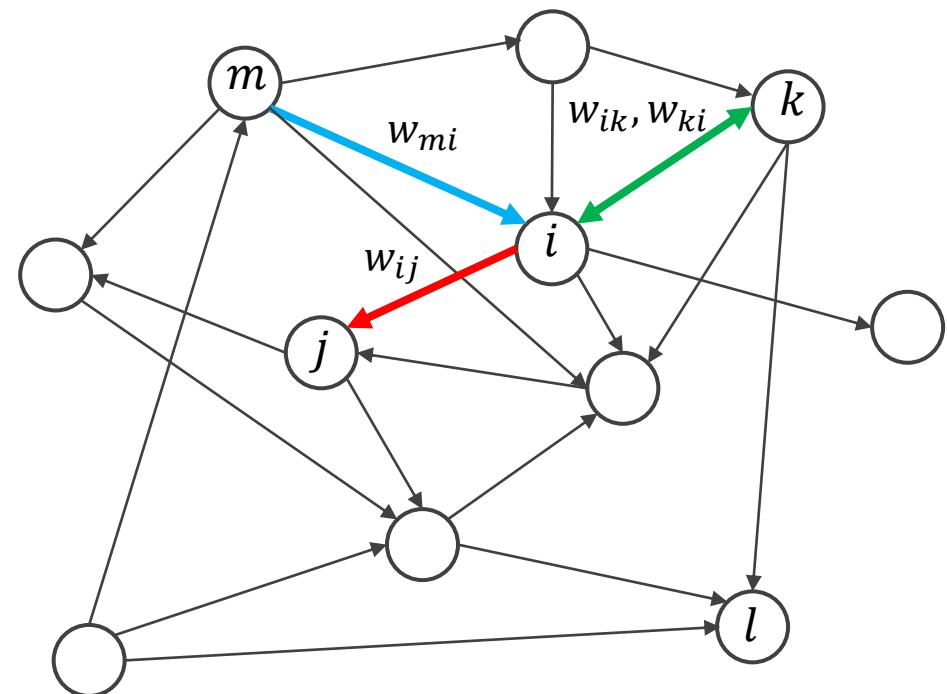
	i	j	k	l	m
i		1	1	0	1
j	1				
k		1			
l		0			
m		1			



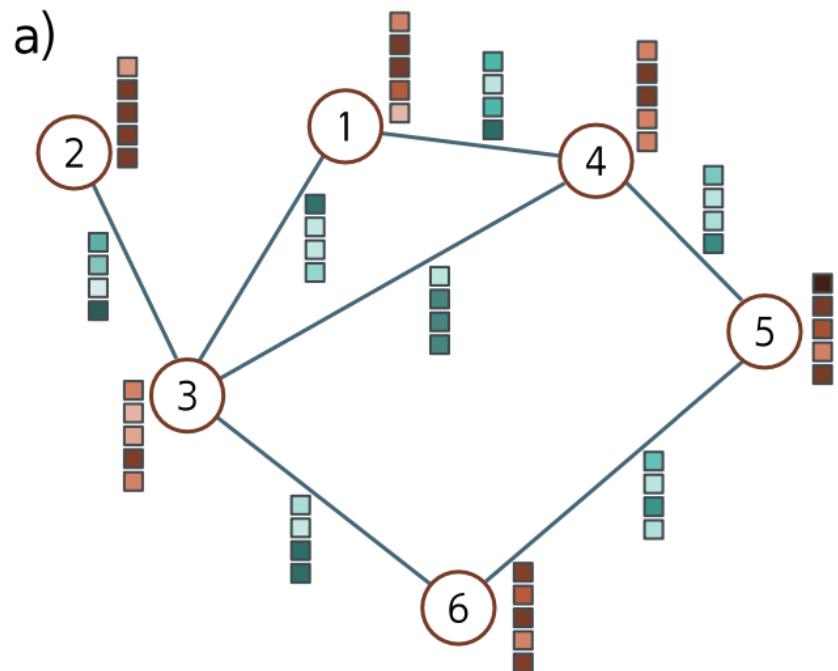
Weighted adjacency matrix

When the edges have weights, so does the adjacency matrix.

	i	j	k	l	m
i		w_{ij}	w_{ik}	0	
j					
k		w_{ki}			
l	0				
m			w_{mi}		



Final graph input representation



b)

Adjacency matrix, \mathbf{A}
 $N \times N$

	1	2	3	4	5	6
1	■	□	□	■	□	□
2	□	■	□	■	□	□
3	■	■	■	□	■	■
4	■	■	■	■	■	■
5	■	■	■	■	■	■
6	■	■	■	■	■	■

c)

Node data, \mathbf{X}
 $D \times N$

1	2	3	4	5	6
■	■	■	■	■	■
■	■	■	■	■	■
■	■	■	■	■	■
■	■	■	■	■	■
■	■	■	■	■	■
■	■	■	■	■	■

d)

Edge data, \mathbf{E}
 $D_E \times E$

1	1	2	3	3	4	5
3	■	■	■	■	■	■
4	■	■	■	■	■	■
3	■	■	■	■	■	■
6	■	■	■	■	■	■
5	■	■	■	■	■	■
6	■	■	■	■	■	■

Back to graph networks

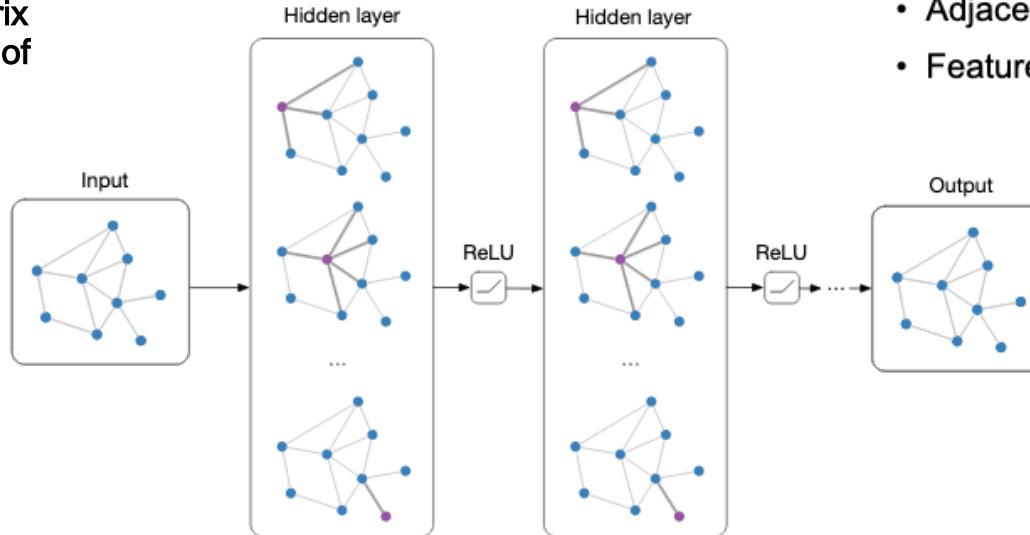
We can do a lot of processing on this data structure.

But the pre-defined features are raw inputs.

Graph networks do 1 thing: transform the feature vector per node over layers.

The bigger picture:

every layer we use the adjacency matrix
to help changing the representation of
the nodes (feature matrix)

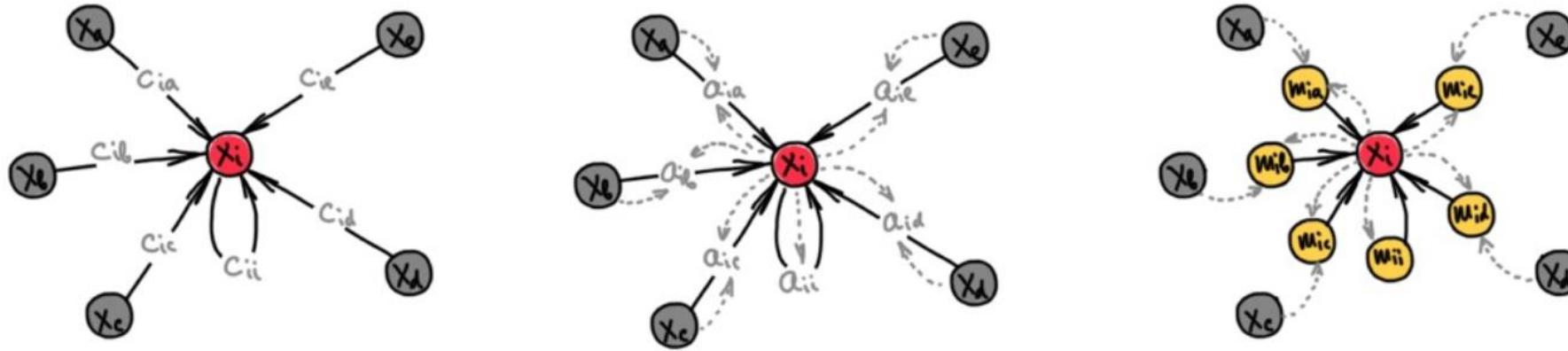


Notation: $\mathcal{G} = (\mathbf{A}, \mathbf{X})$

- Adjacency matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$
- Feature matrix $\mathbf{X} \in \mathbb{R}^{N \times F}$

In the end we obtain an NxN
output feature matrix

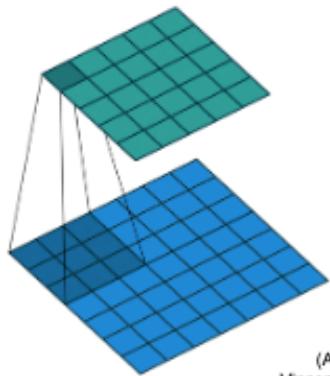
Three perspectives to graph networks



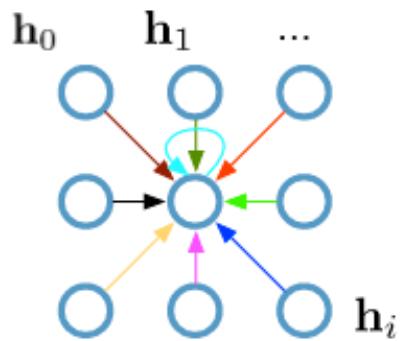
Three “flavours” of GNNs, left-to-right: convolutional, attentional, and general nonlinear message passing flavours. All are forms of message passing. Figure adapted from P. Veličković.

Graph layer as a convolution layer

**Single CNN layer
with 3x3 filter:**



(Animation by
Vincent Dumoulin)



Update for a single pixel:

- Transform messages individually $\mathbf{W}_i h_i$
- Add everything up $\sum_i \mathbf{W}_i h_i$

$h_i \in \mathbb{R}^F$ are (hidden layer) activations of a pixel/node

Full update:

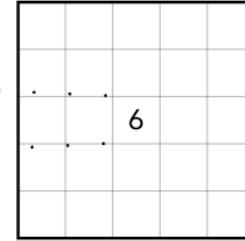
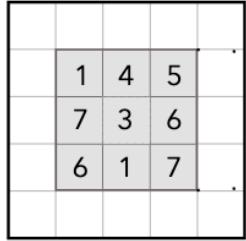
$$\mathbf{h}_4^{(l+1)} = \sigma \left(\mathbf{W}_0^{(l)} \mathbf{h}_0^{(l)} + \mathbf{W}_1^{(l)} \mathbf{h}_1^{(l)} + \dots + \mathbf{W}_8^{(l)} \mathbf{h}_8^{(l)} \right)$$

Which assumptions from images are no longer valid?

Number of neighbors per node no longer fixed.

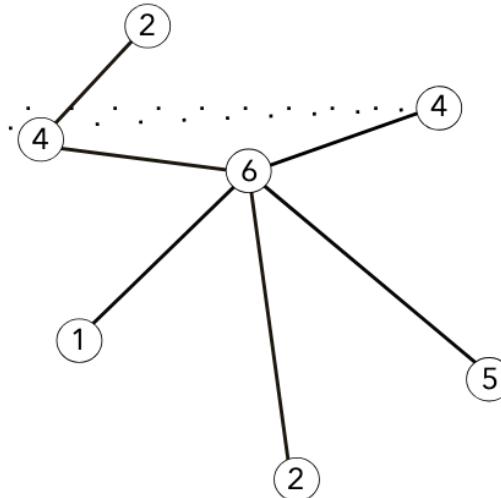
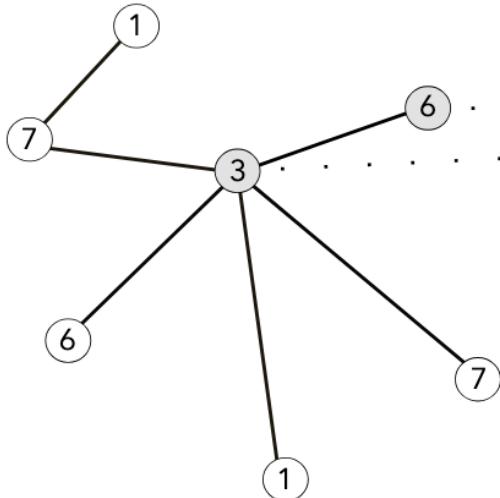
No more ordering between neighbours.

Extending convolutions to graphs



Convolution in CNNs

CNNs perform localized convolutions. Neighbours participating in the convolution at the center pixel are highlighted in gray.



Localized Convolution in GNNs

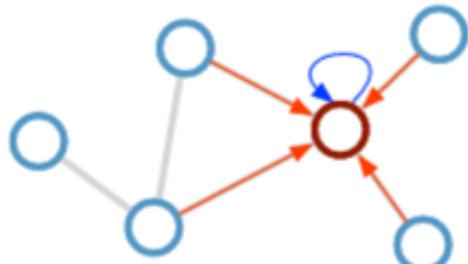
GNNs can perform localized convolutions mimicking CNNs. Hover over a node to see its immediate neighbourhood highlighted on the left. The structure of this neighbourhood changes from node to node.

Graph convolution layer

Consider this undirected graph:



Calculate update for node in red:



Update rule: $\mathbf{h}_i^{(l+1)} = \sigma \left(\mathbf{h}_i^{(l)} \mathbf{W}_0^{(l)} + \sum_{j \in \mathcal{N}_i} \frac{1}{c_{ij}} \mathbf{h}_j^{(l)} \mathbf{W}_1^{(l)} \right)$

Scalability: subsample messages [Hamilton et al., NIPS 2017]

Desirable properties:

- Weight sharing over all locations
- Invariance to permutations
- Linear complexity $O(E)$
- Applicable both in transductive and inductive settings

Limitations:

- Requires gating mechanism / residual connections for depth
- Only indirect support for edge features

for one layer we define one degree of separation.

adding layer we increase the degree of separation i.e how much information one node capture from distant nodes

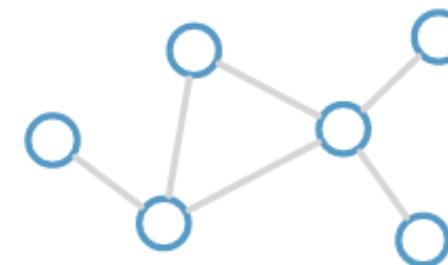
Stacking graph convolution layers

Each layer aggregates information from their direct neighbors.

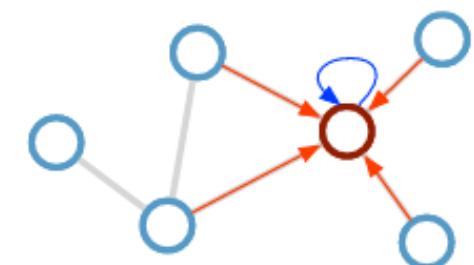
At the end of each layer, we add a non-linearity such as a ReLU.

We can increase complexity and receptive field simply by stacking multiple layers.

Consider this undirected graph:



Calculate update for node in red:



Update rule:
$$\mathbf{h}_i^{(l+1)} = \sigma \left(\mathbf{h}_i^{(l)} \mathbf{W}_0^{(l)} + \sum_{j \in \mathcal{N}_i} \frac{1}{c_{ij}} \mathbf{h}_j^{(l)} \mathbf{W}_1^{(l)} \right)$$

Graph convolution layer in matrix form

$$f(X, A) := \sigma \left(D^{-1/2} (A + I) D^{-1/2} X W \right)$$

$A \in \mathbb{R}^{n \times n}$:= The adjacency matrix

$I \in \mathbb{R}^{n \times n}$:= The identity matrix

$D \in \mathbb{R}^{n \times n}$:= The degree matrix of $A + I$

$X \in \mathbb{R}^{n \times d}$:= The input data (i.e., the per-node feature vectors)

$W \in \mathbb{R}^{d \times w}$:= The layer's weights

$\sigma(\cdot)$:= The activation function (e.g., ReLU)

Let's break it down

$$f(\mathbf{X}, \mathbf{A}) := \sigma\left(\mathbf{D}^{-1/2}(\mathbf{A} + \mathbf{I})\mathbf{D}^{-1/2}\mathbf{XW}\right)$$

The equation is enclosed in a large dashed rectangular bracket. Inside, there are three nested brackets: a top bracket labeled "Add self-loops", a middle bracket labeled "Normalize adjacency matrix", and a bottom bracket labeled "Aggregate". The "Aggregate" bracket also contains a small bracket labeled "Update" at its bottom right corner.

Let's break it down

Add ones to diagonal, needed because each node should pass its own vector through.

the diagonal of adjacency matrix has all zero, but we want to use the informations about self nodes so we add a diagonal of 1 with ones

$$f(\mathbf{X}, \mathbf{A}) := \sigma\left(\mathbf{D}^{-1/2} (\mathbf{A} + \mathbf{I}) \mathbf{D}^{-1/2} \mathbf{XW}\right)$$

The diagram illustrates the components of the equation $f(\mathbf{X}, \mathbf{A})$. It consists of three nested dashed boxes:

- The innermost box contains the term $\mathbf{A} + \mathbf{I}$, with a red box highlighting \mathbf{I} . This is labeled "Add self-loops".
- The middle box contains the term $\mathbf{D}^{-1/2} (\mathbf{A} + \mathbf{I}) \mathbf{D}^{-1/2}$, with a bracket below it labeled "Normalize adjacency matrix".
- The outermost box contains the entire expression $\sigma(\mathbf{D}^{-1/2} (\mathbf{A} + \mathbf{I}) \mathbf{D}^{-1/2} \mathbf{XW})$, with a bracket below it labeled "Aggregate".

Below the middle box, there is a bracket labeled "Update".

Let's break it down

This step essentially normalizes the adjacency matrix. I will show how in a few slides.

The matrix D is a diagonal matrix with the degree of the nodes on the diagonal. nodes with huge grades obtain a higher sum from neighbours, this creates an imbalance in activations between nodes that needs to be normalized

$$f(\mathbf{X}, \mathbf{A}) := \sigma\left(\boxed{\mathbf{D}^{-1/2}} (\mathbf{A} + \mathbf{I}) \boxed{\mathbf{D}^{-1/2}} \mathbf{XW}\right)$$

The diagram illustrates the components of the equation:

- $\boxed{\mathbf{D}^{-1/2}}$: Add self-loops
- $(\mathbf{A} + \mathbf{I})$: Normalize adjacency matrix
- $\boxed{\mathbf{D}^{-1/2}}$: Aggregate
- \mathbf{XW} : Update

Let's break it down

Just a standard linear layer and a non-linearity.

Cant see the convolution here

$$f(\mathbf{X}, \mathbf{A}) := \sigma(\mathbf{D}^{-1/2}(\mathbf{A} + \mathbf{I})\mathbf{D}^{-1/2}\mathbf{XW})$$

The diagram illustrates the components of the function $f(\mathbf{X}, \mathbf{A})$ using curly braces and labels:

- Add self-loops:** A brace under the term $\mathbf{A} + \mathbf{I}$.
- Normalize adjacency matrix:** A brace under the term $\mathbf{D}^{-1/2}(\mathbf{A} + \mathbf{I})\mathbf{D}^{-1/2}$.
- Aggregate:** A brace under the term \mathbf{XW} .
- Update:** A brace under the entire expression $\sigma(\mathbf{D}^{-1/2}(\mathbf{A} + \mathbf{I})\mathbf{D}^{-1/2}\mathbf{XW})$.

Two specific terms are highlighted with red boxes:

- σ (non-linearity)
- \mathbf{XW} (linear layer)

Let's break it down

Just a standard linear layer and a non-linearity.

$$f(\mathbf{X}, \mathbf{A}) := \sigma\left(\boxed{\mathbf{D}^{-1/2}(\mathbf{A} + \mathbf{I})\mathbf{D}^{-1/2}\mathbf{XW}}\right)$$

The diagram illustrates the components of the equation $f(\mathbf{X}, \mathbf{A})$. It shows a red box around the term $\mathbf{D}^{-1/2}(\mathbf{A} + \mathbf{I})\mathbf{D}^{-1/2}$. Below this box, a bracket labeled "Add self-loops" points to the identity matrix \mathbf{I} . To the left of the red box, another bracket labeled "Normalize adjacency matrix" points to the product of $\mathbf{D}^{-1/2}$ and \mathbf{A} . Below the red box, a bracket labeled "Aggregate" points to the product of $\mathbf{D}^{-1/2}$ and \mathbf{XW} . At the bottom, a bracket labeled "Update" points to the entire expression $\sigma(\text{red box})$.

Rewriting into 2 steps

$$\tilde{\mathbf{A}} := \mathbf{D}^{-1/2}(\mathbf{A} + \mathbf{I})\mathbf{D}^{-1/2}$$

$$\tilde{A}_{i,j} := \begin{cases} \frac{1}{\sqrt{d_{i,i}d_{j,j}}}, & \text{if there is an edge between node } i \text{ and } j \\ 0, & \text{otherwise} \end{cases}$$

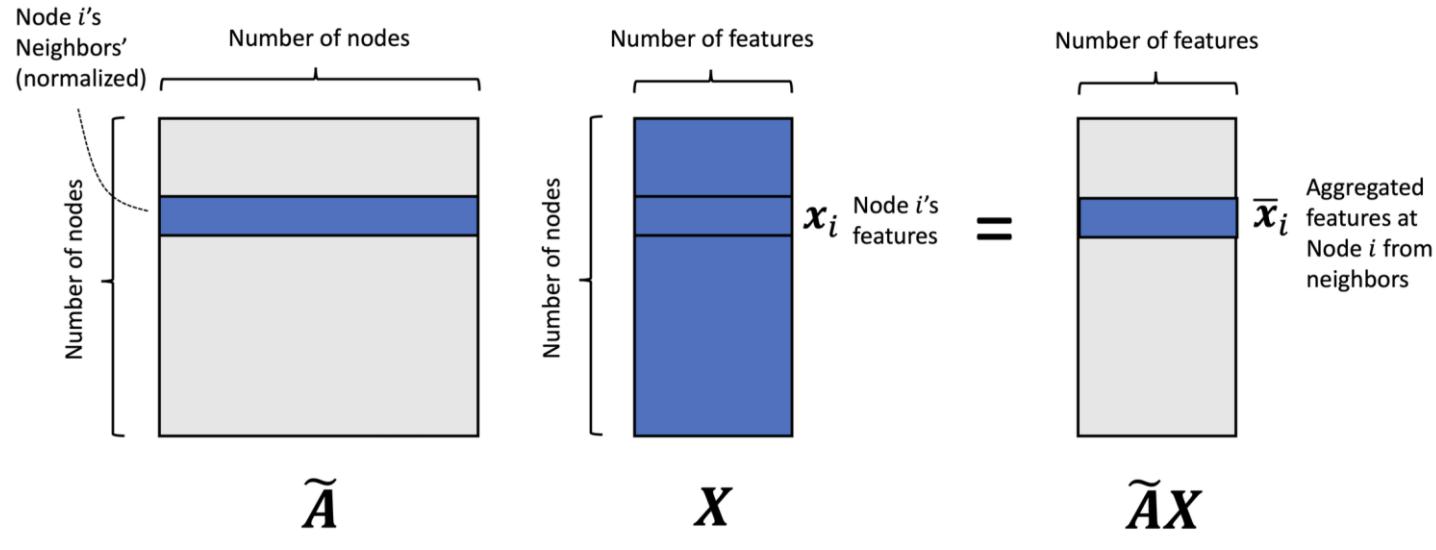
$$f(X, A) := \sigma(\tilde{\mathbf{A}}\mathbf{X}\mathbf{W})$$

$$\mathbf{D} := \begin{bmatrix} d_{1,1} & 0 & 0 & \dots & 0 \\ 0 & d_{2,2} & 0 & \dots & 0 \\ 0 & 0 & d_{3,3} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & d_{n,n} \end{bmatrix}$$

$$\mathbf{D}^{-1/2} := \begin{bmatrix} \frac{1}{\sqrt{d_{1,1}}} & 0 & 0 & \dots & 0 \\ 0 & \frac{1}{\sqrt{d_{2,2}}} & 0 & \dots & 0 \\ 0 & 0 & \frac{1}{\sqrt{d_{3,3}}} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & \frac{1}{\sqrt{d_{n,n}}} \end{bmatrix}$$

$$f(X, A) := \sigma(\tilde{A}XW)$$

Left side of the equation



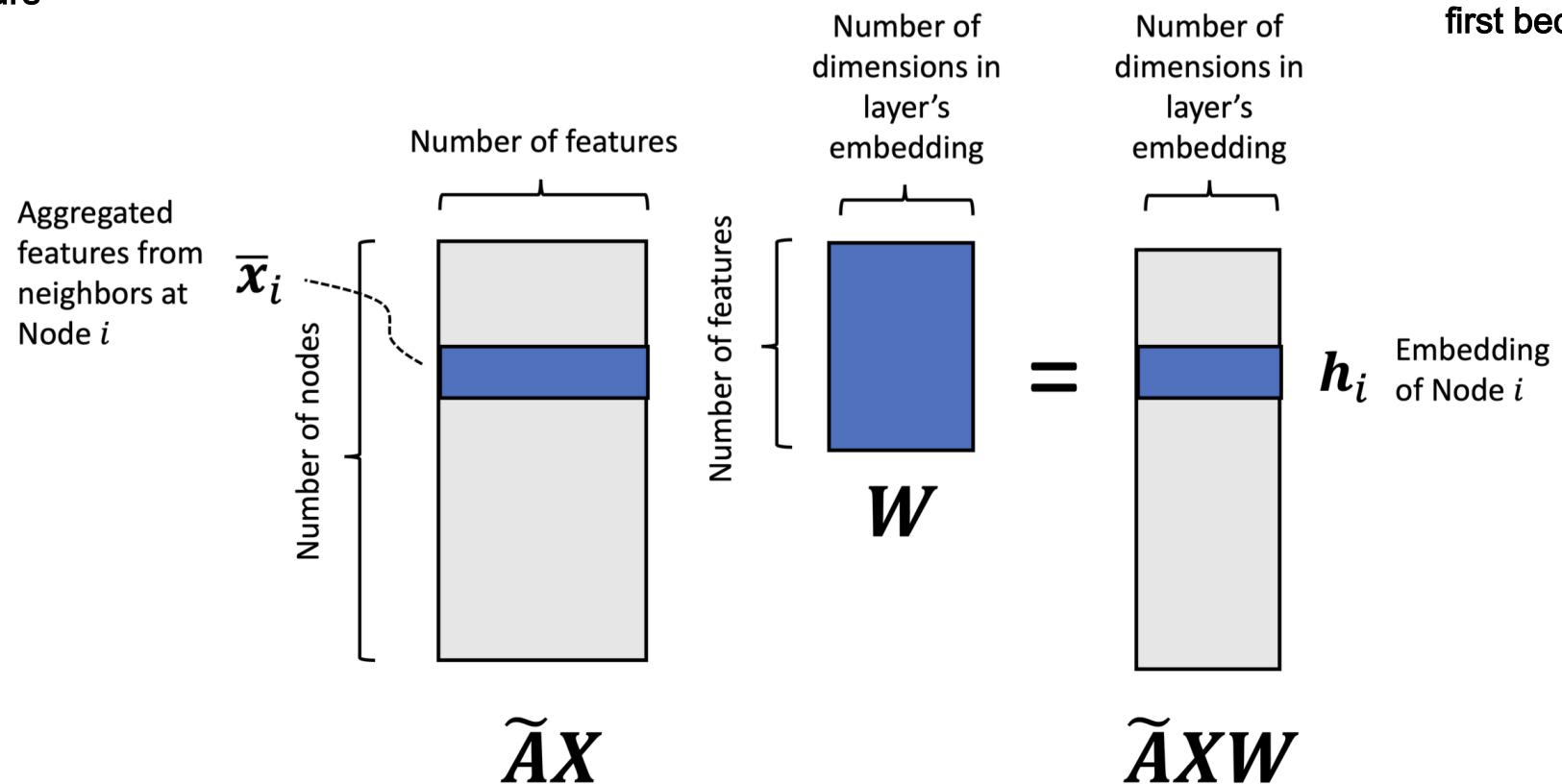
$$\begin{aligned}\bar{x}_i &= \sum_{j=1}^n \tilde{a}_{i,j} x_j \\ &= \sum_{j \in \text{Neigh}(i)} \tilde{a}_{i,j} x_j \\ &= \sum_{j \in \text{Neigh}(i)} \frac{1}{\sqrt{d_{i,i} d_{j,j}}} x_j\end{aligned}$$

For each feature i 'm summing all the values of that specifico feature from the nodes in the neighborood

$$f(X, A) := \sigma(\tilde{A}XW)$$

Right side of the equation

the core concept is doing an MLP and summing the neighbours



It can be easier to think as a linear transformations of the whole feature and then apply to it the graph structure captured by A

but it is more efficient to do AX first because A may be sparse

Why add a normalization step?

Do we even need it? Let's see what happens without it:

$$\hat{A} := A + I$$

Normalization dropped, only
self-loop retrained

$$f_{\text{unnormalized}}(X, A) := \sigma(\hat{A}XW)$$

Layer update remains the same

$$\bar{x}_i = \sum_{j=1}^n \hat{a}_{i,j}x_j$$

$$= \sum_{j=1}^n \mathbb{I}(j \in \text{Neigh}(i))x_j$$

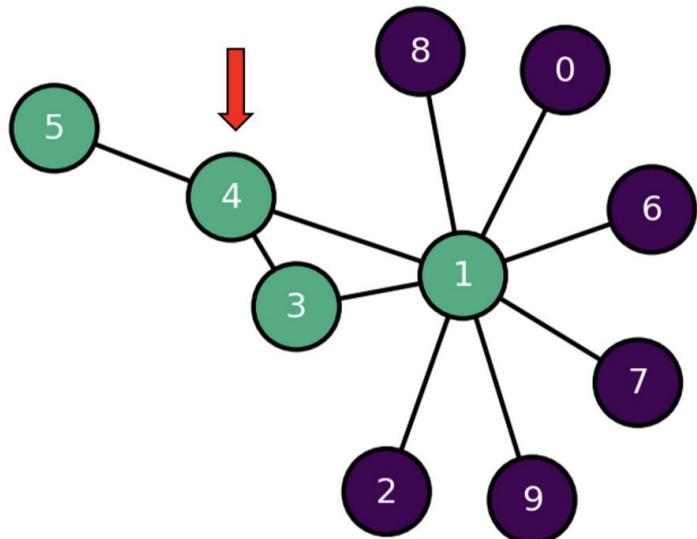
$$= \sum_{j \in \text{Neigh}(i)} x_j$$

Problem! More neighbors = bigger sum.
Huge bias when training graph networks.

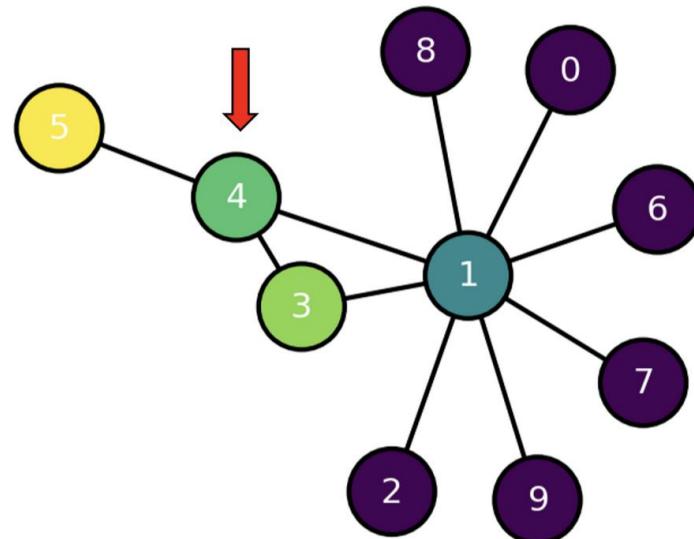
Why not simply divide by the node degree?

this is a totally viable way
to implement a graph
neural network. here I have
a uniform weighting

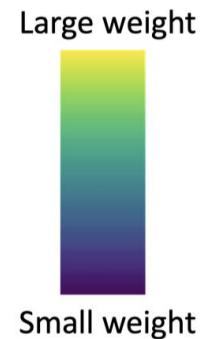
$$D^{-1}\hat{A}$$



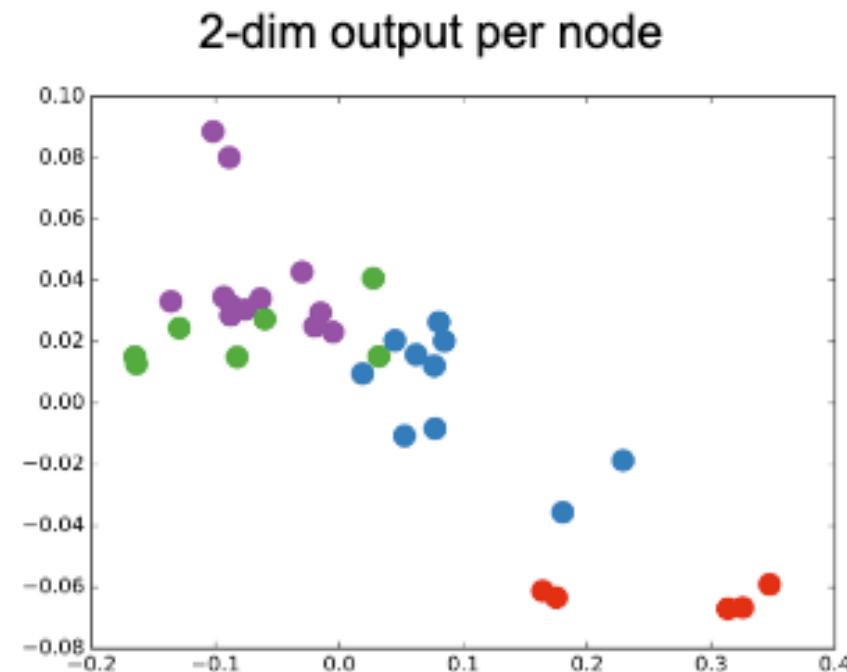
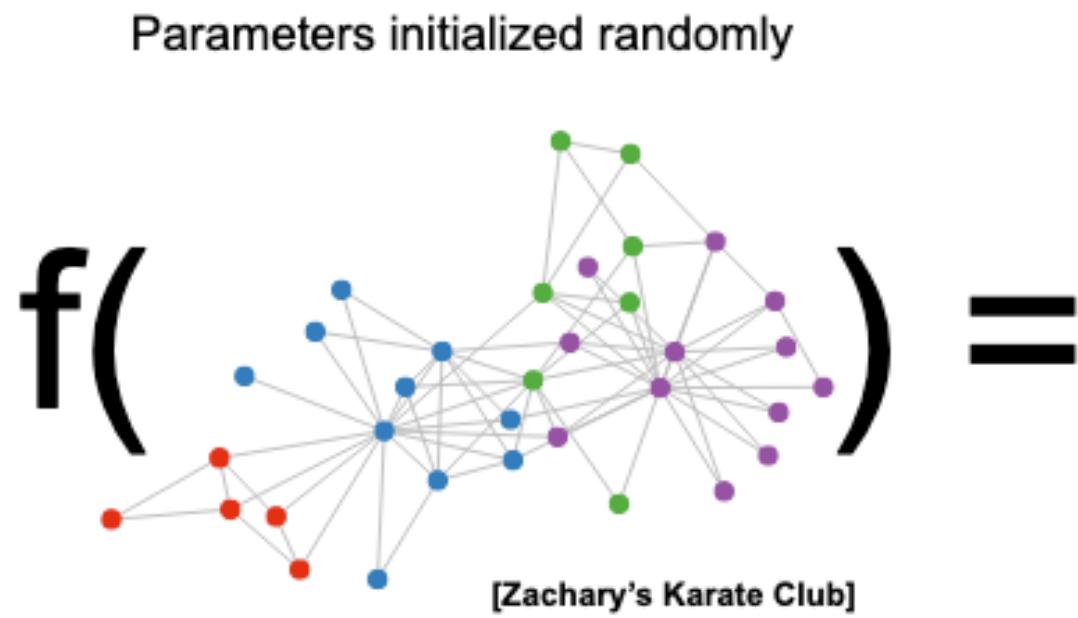
$$D^{-1/2}\hat{A}D^{-1/2}$$



but this one, use the joint
degree between edges. It
makes something more unique
more important.
This implementation is
incorporating some second
order information



Visualizing node representations



Alternative: graph layer as attention

Similar but including attention as *aggregation*: $y_i = h\left(\sum_{j \in \mathcal{N}(i)} a_{ij} \mathbf{z}_j\right)$

Using self-attention:

$$a_{ij} = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}(i)} \exp(e_{ik})},$$

where e_{ij} are the self-attention weights (like query == key)

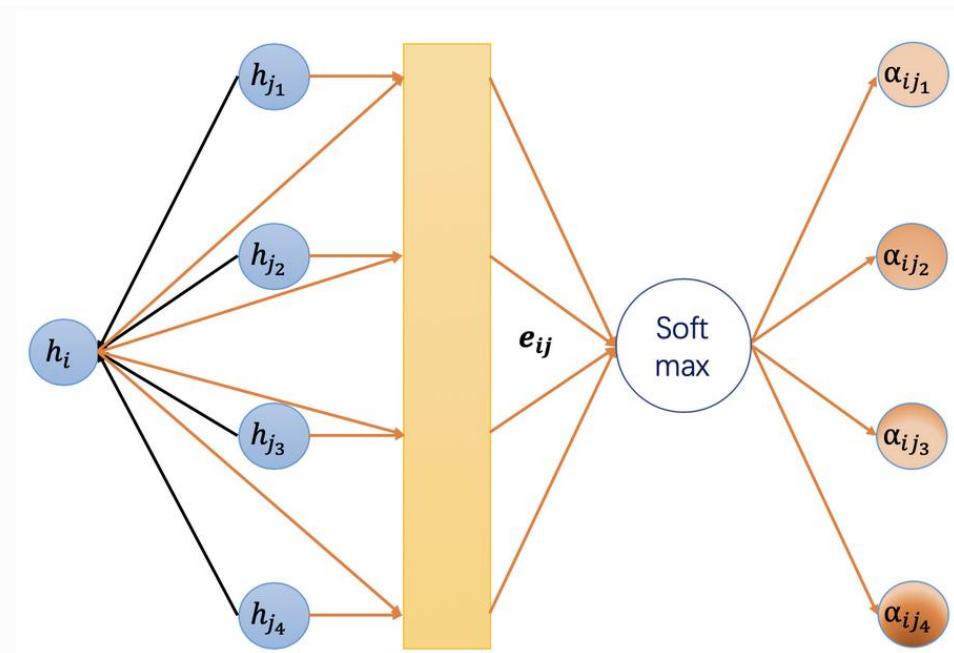
$$e_{ij} = \text{LeakyReLU}\left([\mathbf{x}_i \mathbf{W}, \mathbf{x}_j \mathbf{W}] \cdot \mathbf{u}\right)$$

\mathbf{u} is a weight vector.

This bracket notation stands for concatenation

Flexibility in Learning Relationships: Concatenation allows the attention mechanism to learn asymmetric relationships between nodes. When you concatenate $[\mathbf{x}_j \mathbf{W}, \mathbf{x}_i \mathbf{W}]$ the network can learn that the importance of node j to node i might be different from the importance of i to j even in undirected graphs.

The four steps of a graph attention layer



$$z_i^{(l)} = W^{(l)} h_i^{(l)}, \quad (1)$$

$$e_{ij}^{(l)} = \text{LeakyReLU}(\vec{a}^{(l)T} (z_i^{(l)} || z_j^{(l)})), \quad (2)$$

$$\alpha_{ij}^{(l)} = \frac{\exp(e_{ij}^{(l)})}{\sum_{k \in \mathcal{N}(i)} \exp(e_{ik}^{(l)})}, \quad (3)$$

$$h_i^{(l+1)} = \sigma \left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij}^{(l)} z_j^{(l)} \right), \quad (4)$$

Connecting graphs, convolutions, and transformers

Transformers operate on a complete graph (adjacency matrix with all 1's).

With attention-based GCN, we recover the Transformer.

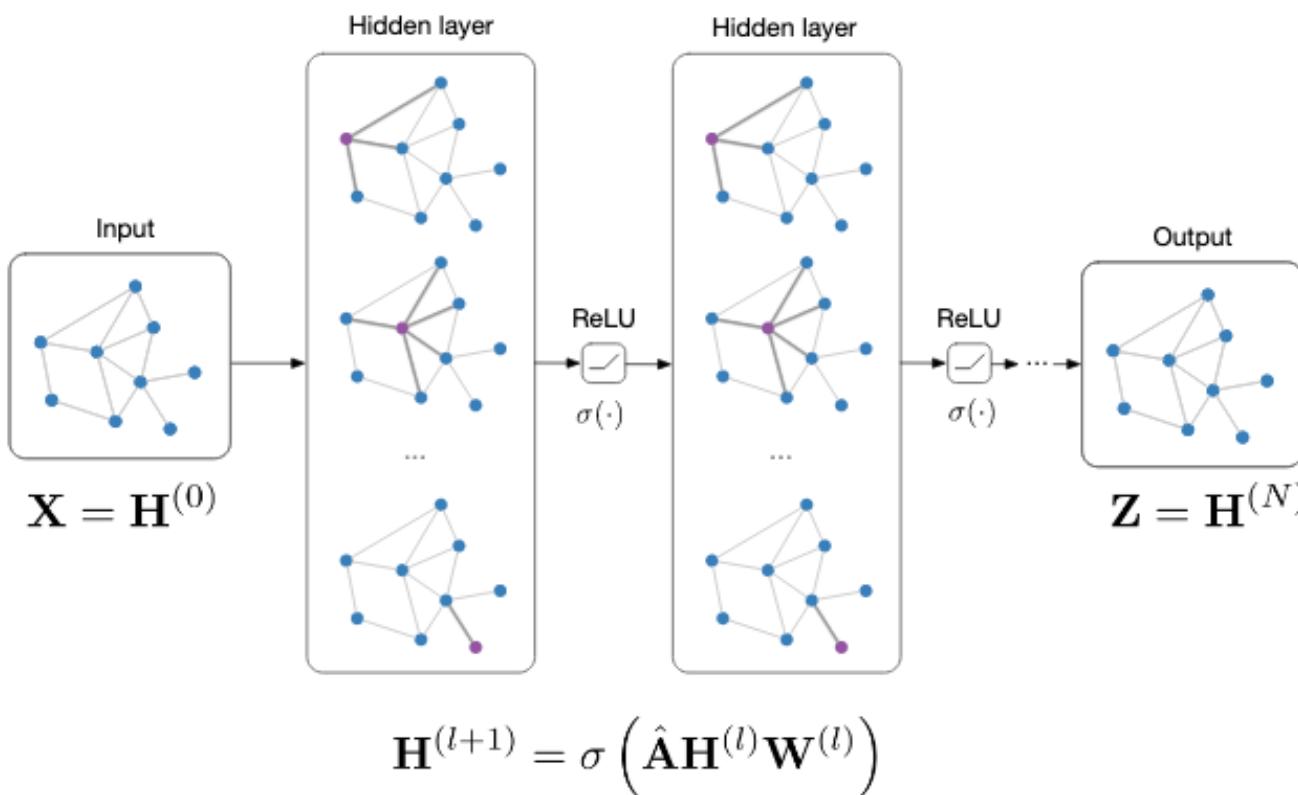
equivariancy:

Architecture	Domain Ω	Symmetry group \mathfrak{G}
CNN	Grid	Translation
<i>Spherical CNN</i>	Sphere / SO(3)	Rotation SO(3)
<i>Intrinsic / Mesh CNN</i>	Manifold	Isometry $\text{Iso}(\Omega)$ / Gauge symmetry SO(2)
GNN	Graph	Permutation Σ_n
<i>Deep Sets</i>	Set	Permutation Σ_n
<i>Transformer</i>	Complete Graph	Permutation Σ_n
LSTM	1D Grid	Time warping

Optimizing graph networks

Input: Feature matrix $\mathbf{X} \in \mathbb{R}^{N \times E}$, preprocessed adjacency matrix $\hat{\mathbf{A}}$

after K layers I have only new node feature, the structure of the graphs remain the same



Node classification:

$$\text{softmax}(\mathbf{z}_n)$$

e.g. Kipf & Welling (ICLR 2017)

Graph classification:

$$\text{softmax}(\sum_n \mathbf{z}_n)$$

e.g. Duvenaud et al. (NIPS 2015)

Link prediction:

$$p(A_{ij}) = \sigma(\mathbf{z}_i^T \mathbf{z}_j)$$

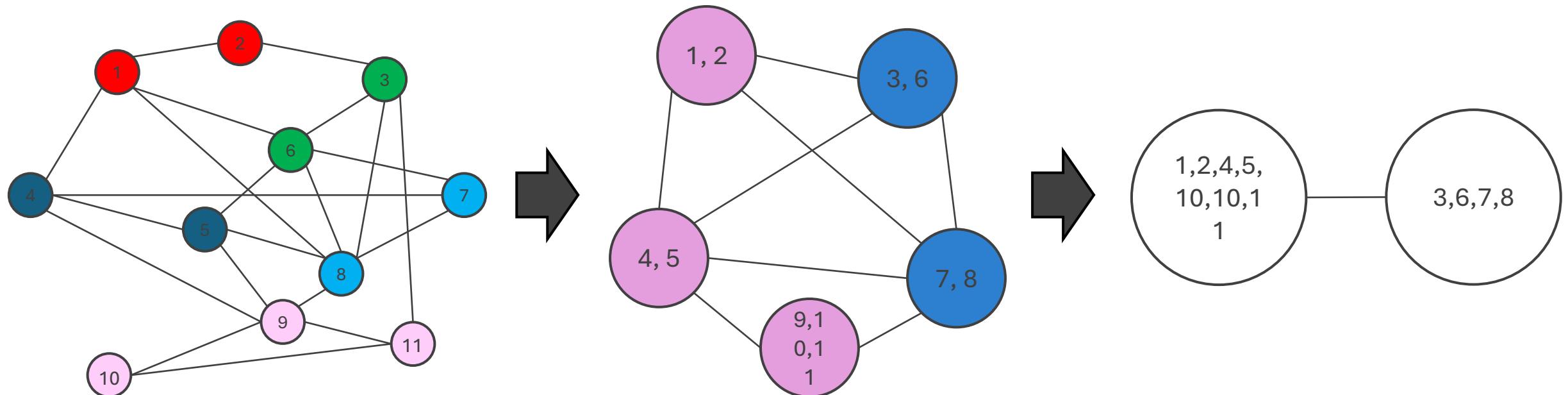
Kipf & Welling (NIPS BDL 2016)

“Graph Auto-Encoders”

Pooling in graph networks

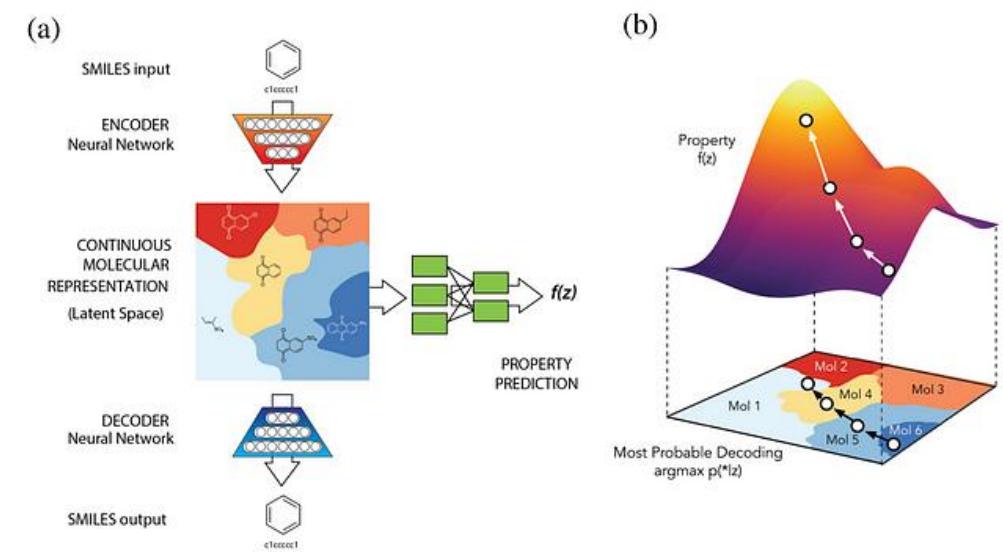
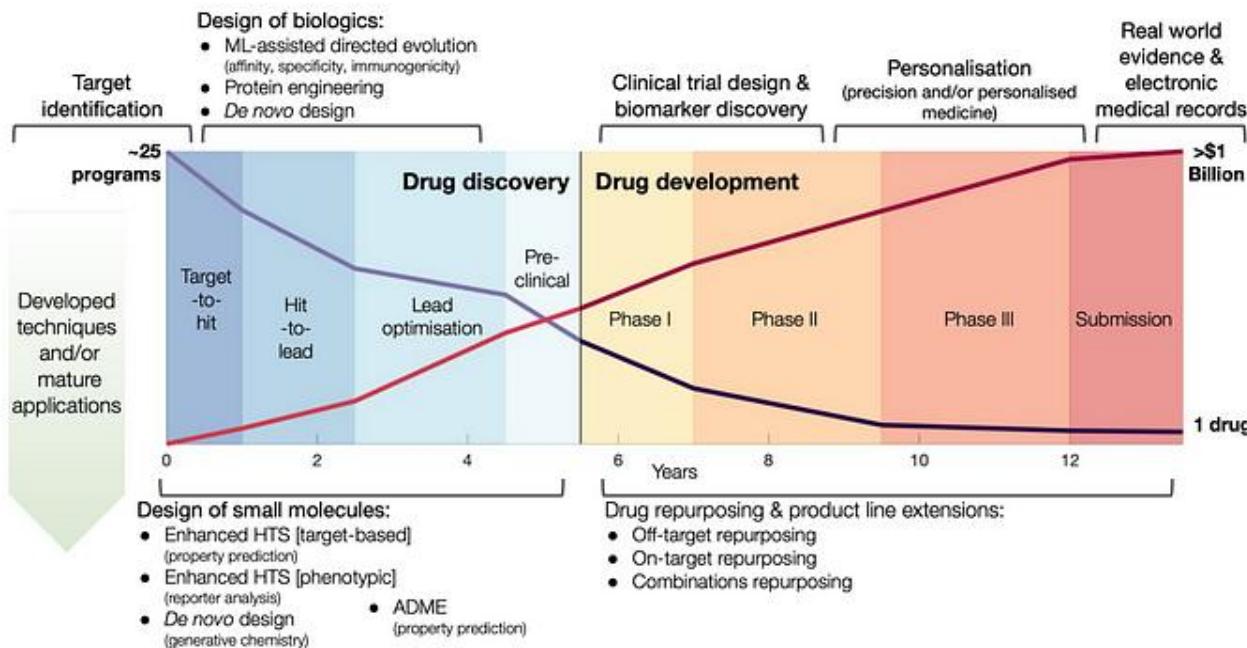
Specifically for graph classification, pooling is an optional operators.

Pool nodes together to save compute, requires updating the adjacency matrix.

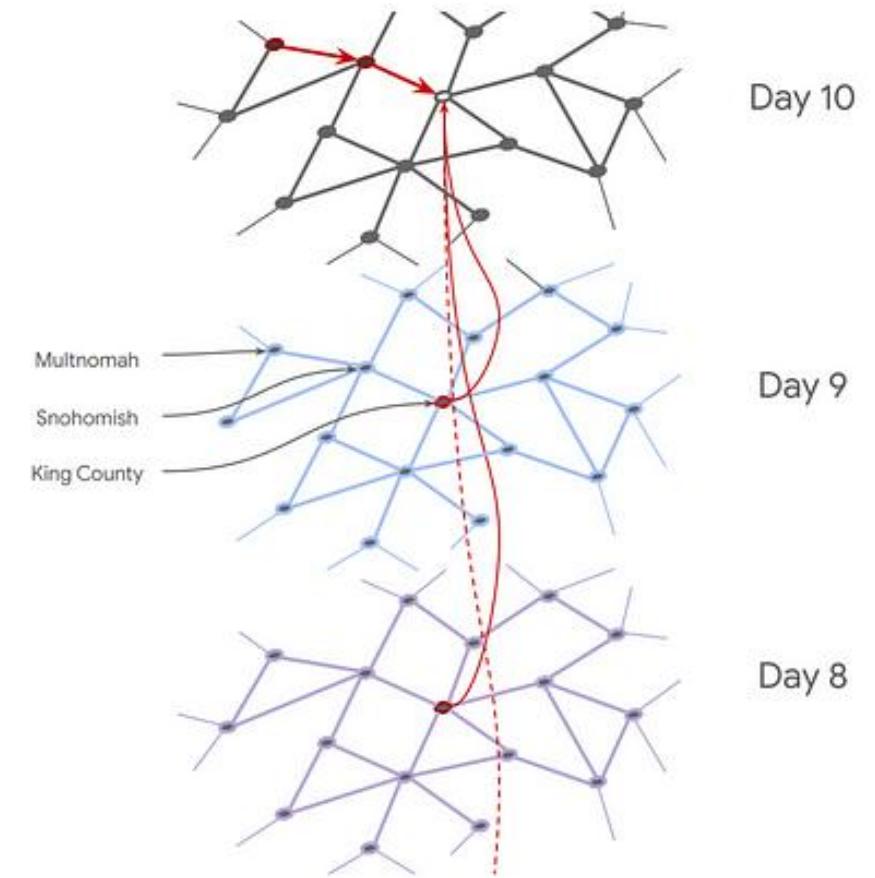
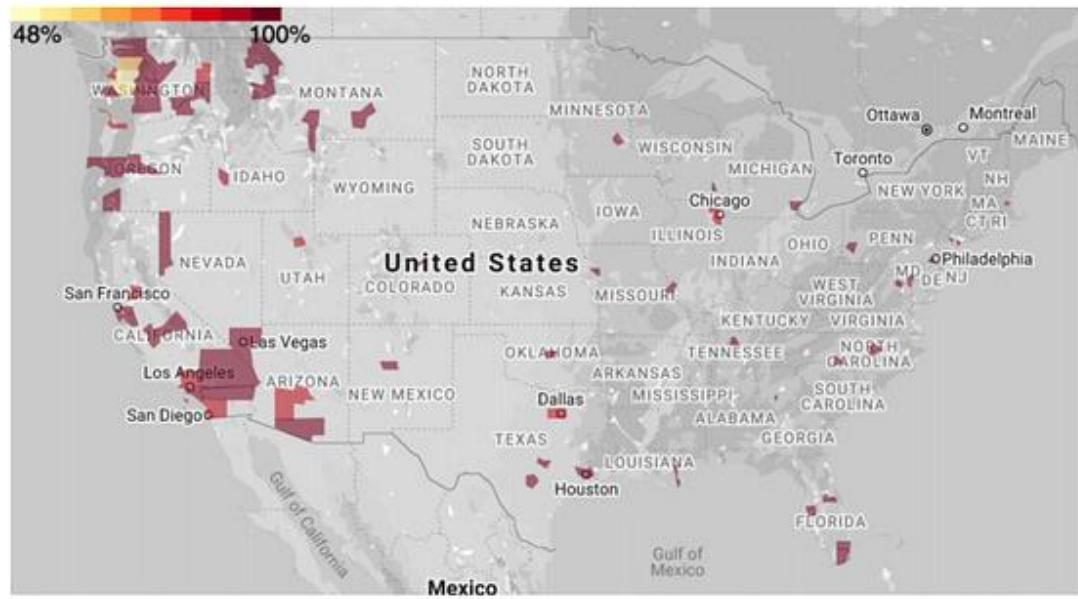


Applications of graph networks

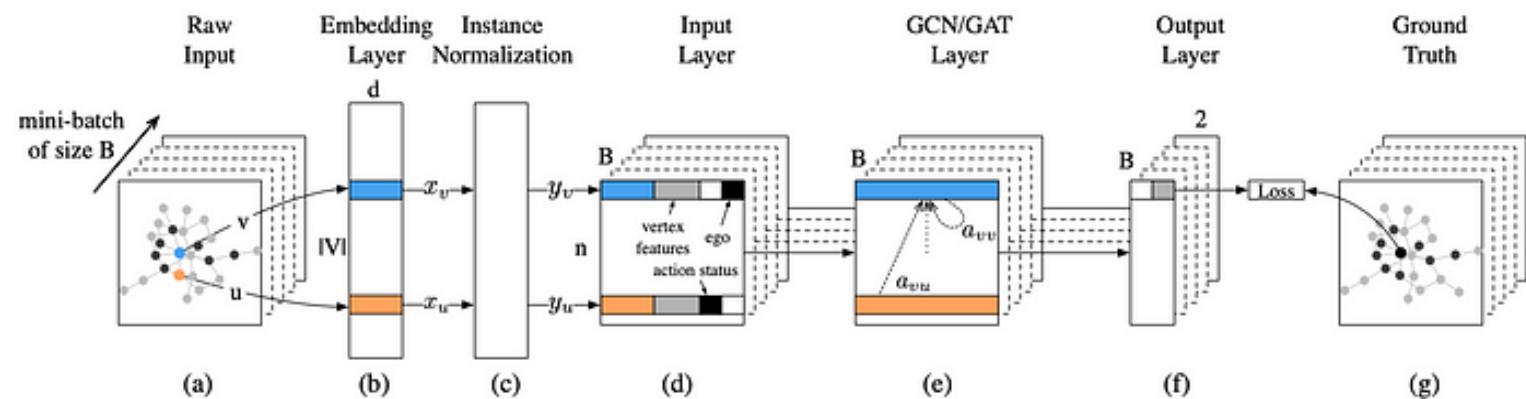
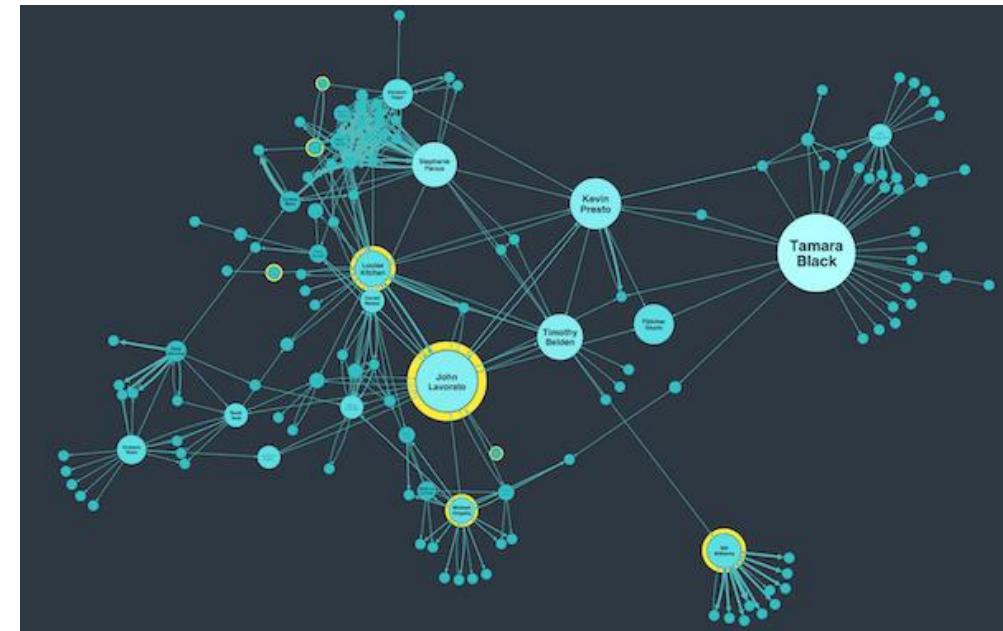
Drug discovery



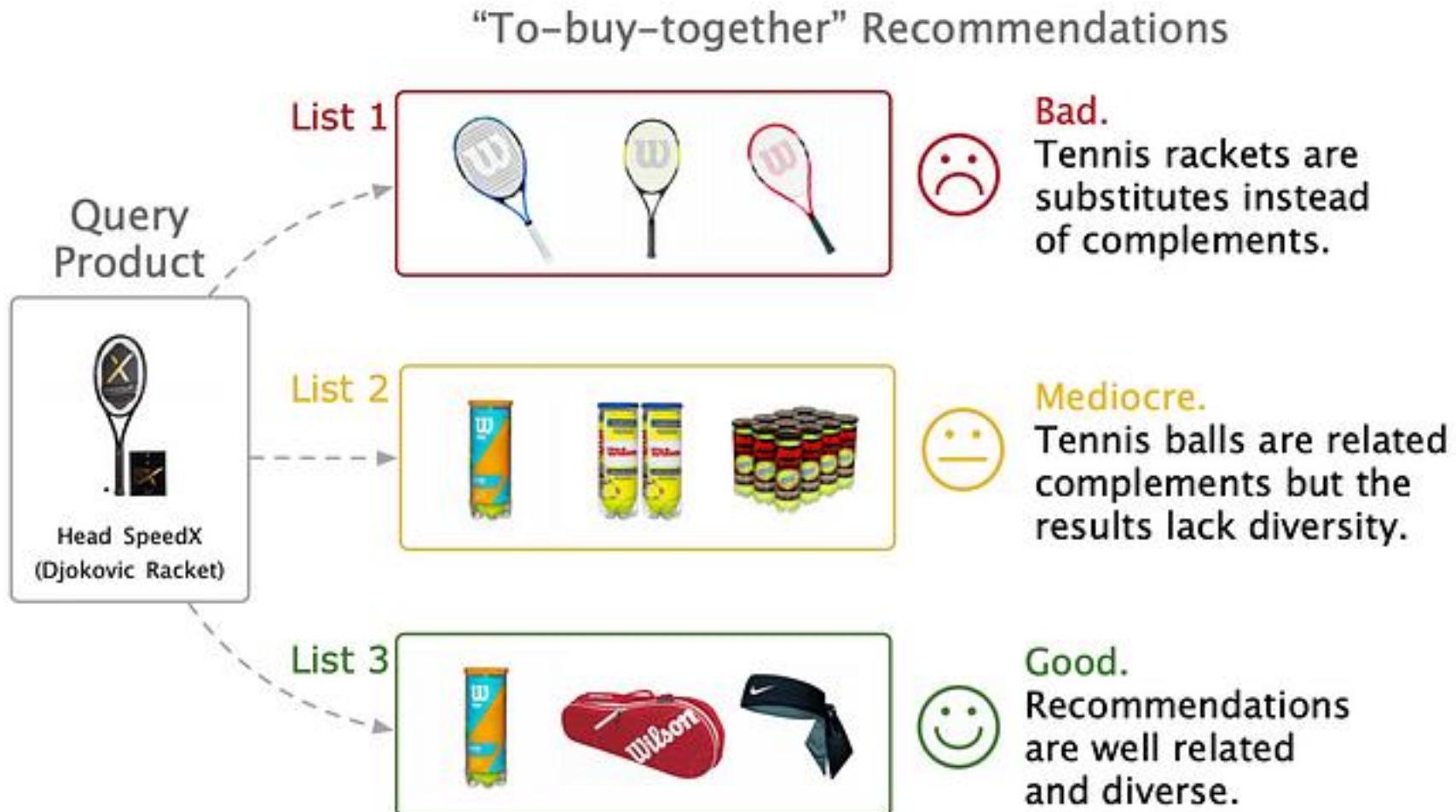
Modeling the spread of deceases



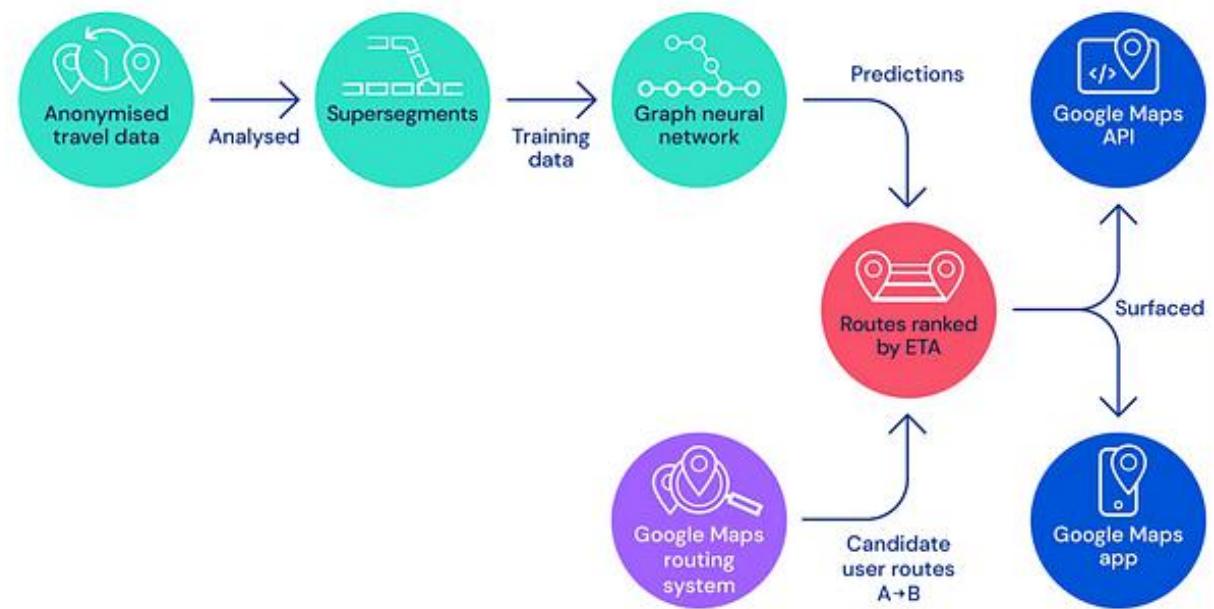
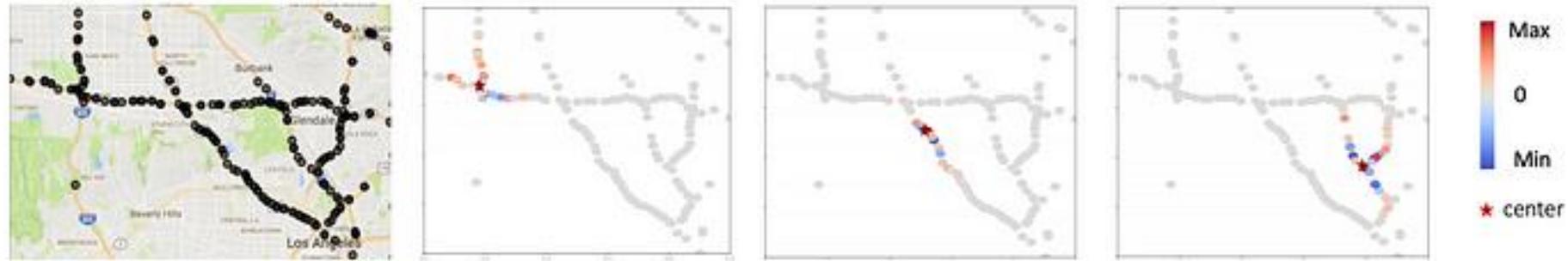
Social networks



Recommendation

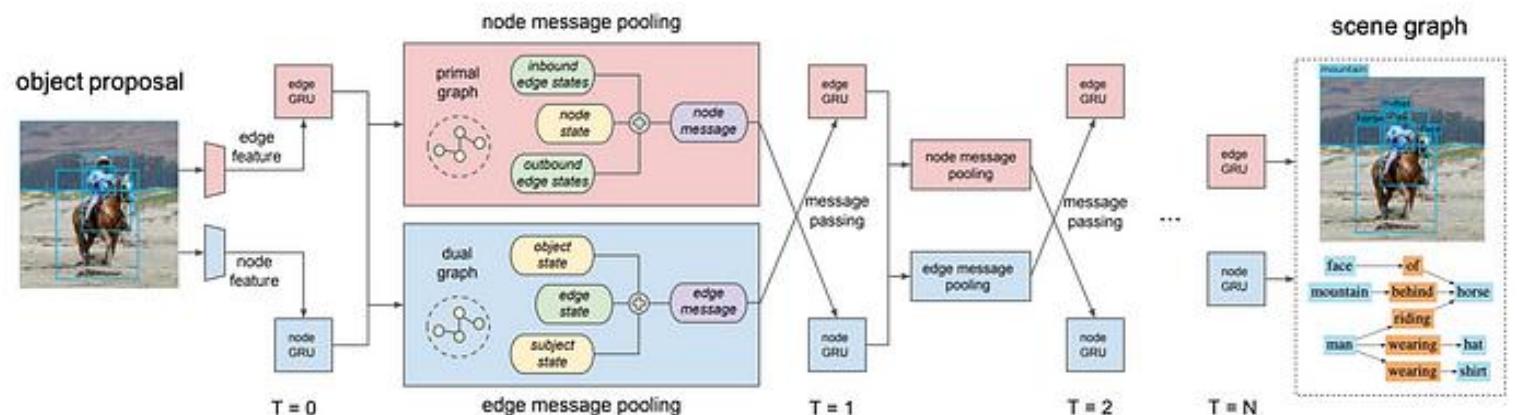
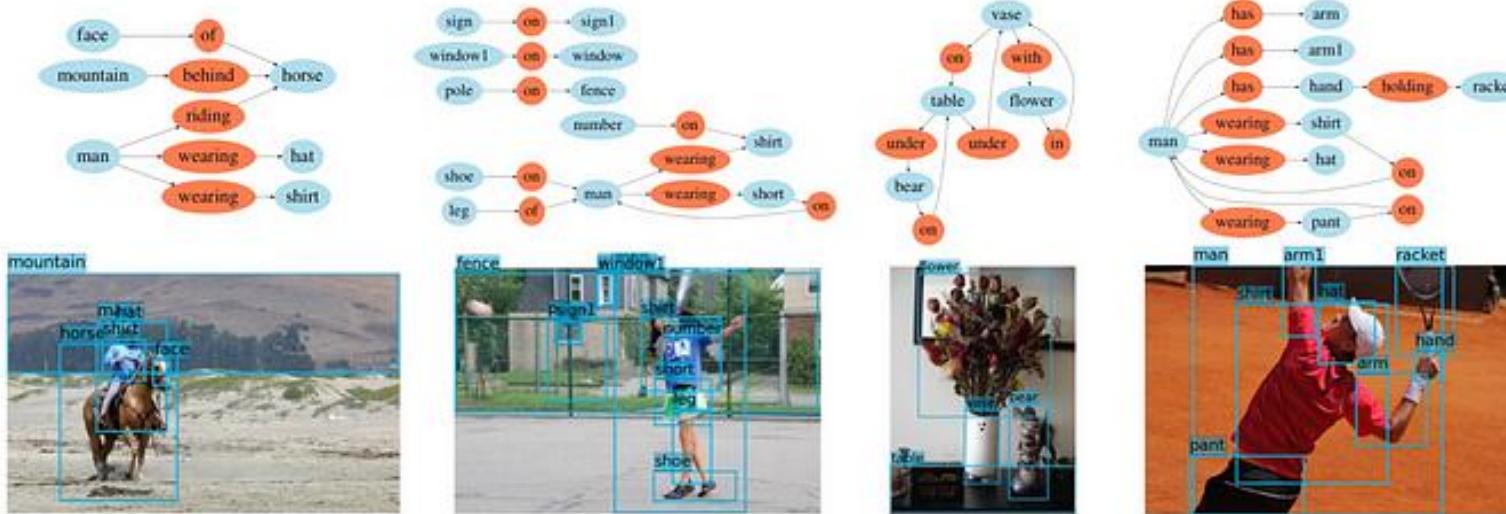


Traffic forecasting

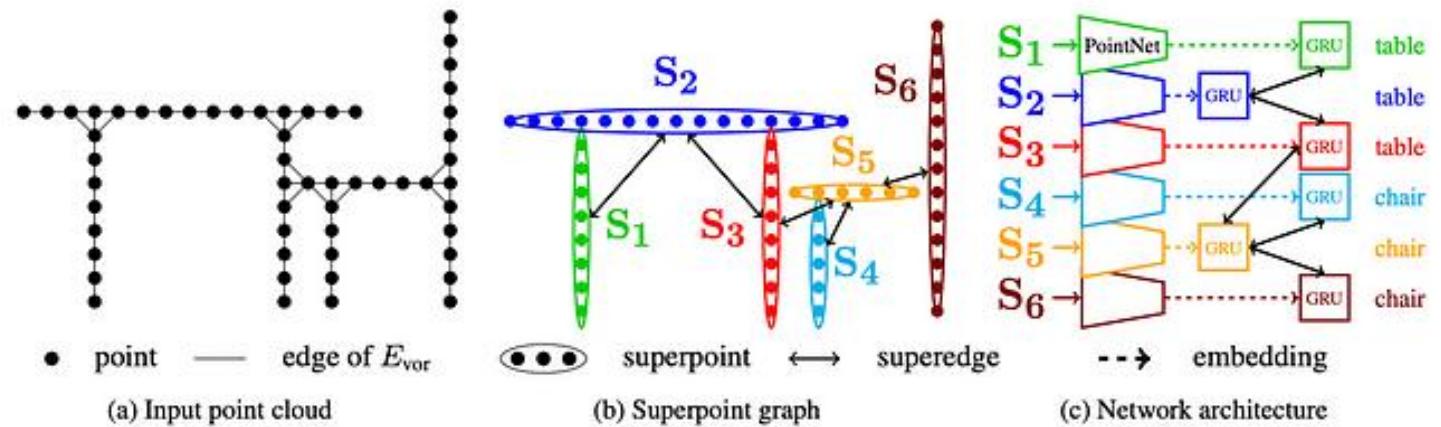
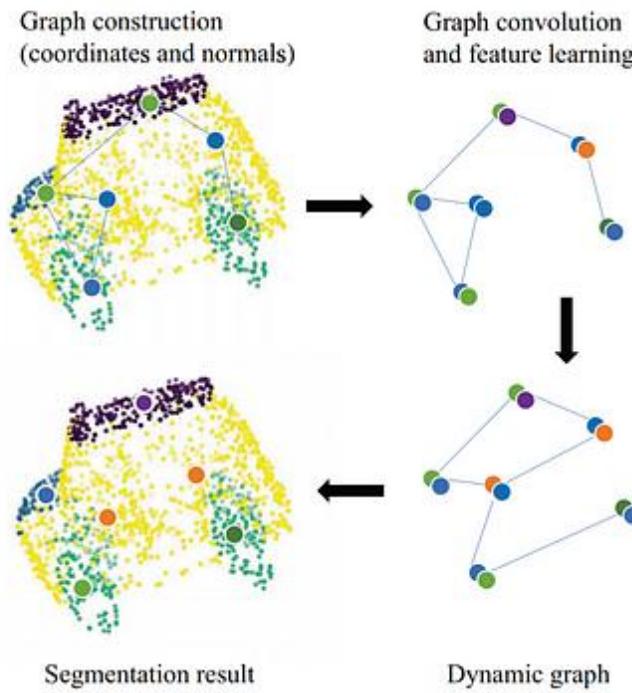


The model architecture for determining optimal routes and their travel time.

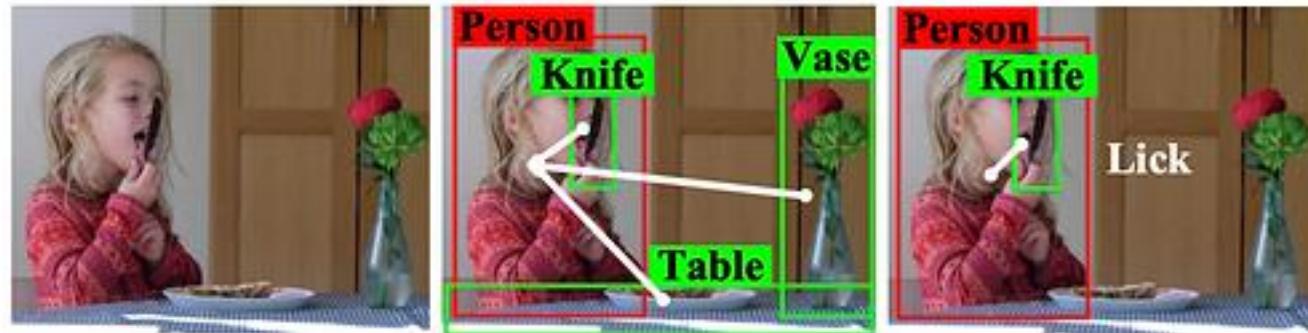
Scene graph generation of visual data



Point cloud classification



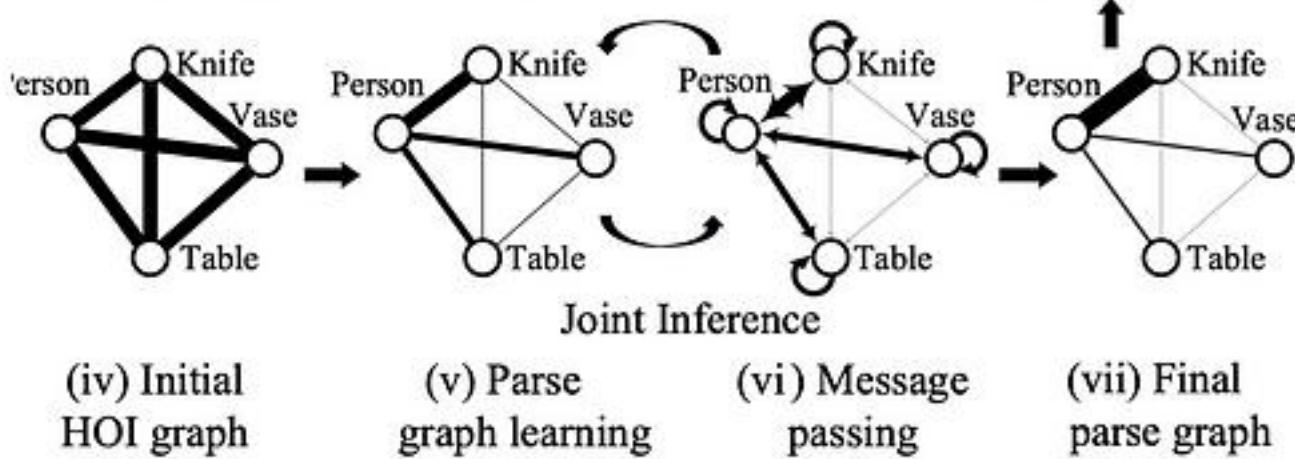
Object interactions



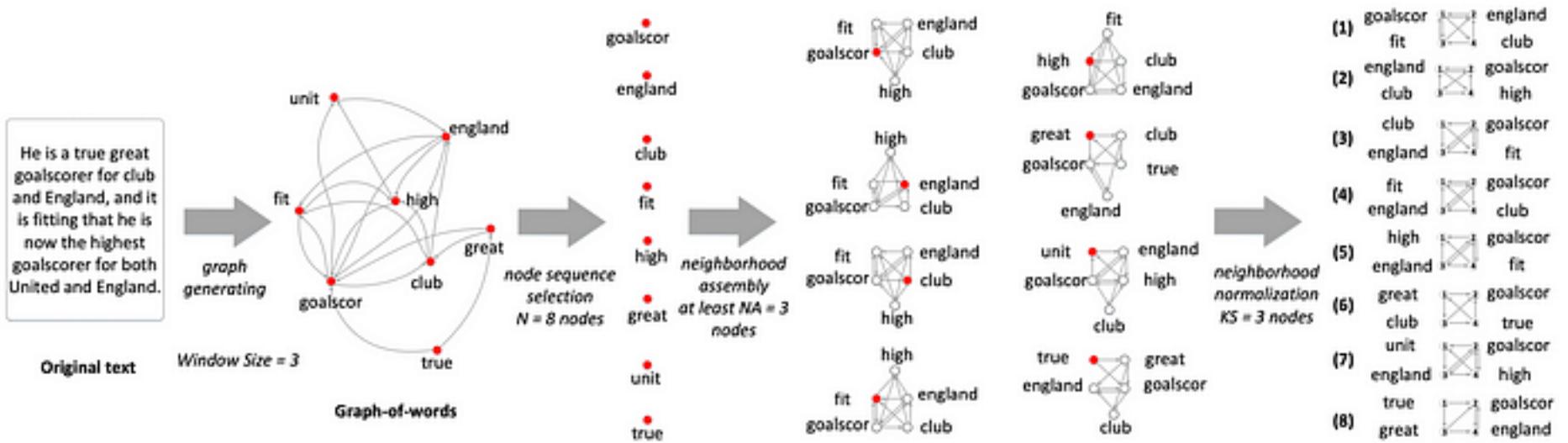
(i) Image

(ii) HOI candidates

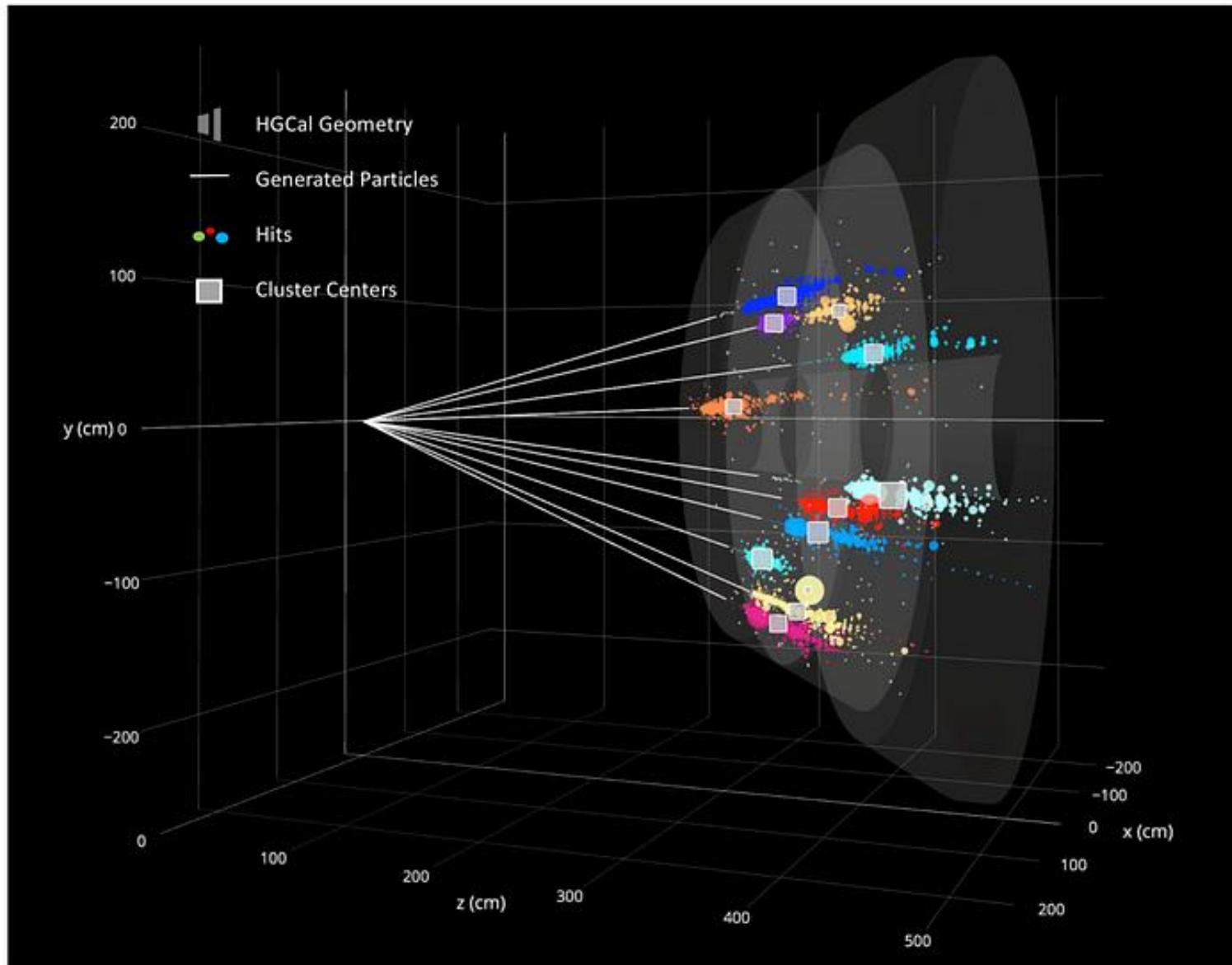
(iii) HOI result



Text classification



Particle physics



Next lecture

Lecture	Title	Lecture	Title
1	Intro and history of deep learning	2	AutoDiff
3	Deep learning optimization I	4	Deep learning optimization II
5	Convolutional deep learning	6	Attention-based deep learning
7	Graph deep learning	8	From supervised to unsupervised deep learning
9	Multi-modal deep learning	10	Generative deep learning
11	What doesn't work in deep learning	12	Non-Euclidean deep learning
13	Q&A	14	Deep learning for videos

Learning and reflection

Understanding Deep Learning: Chapter 13

Thank you!



Deep Learning 1

2025-2026 – Pascal Mettes

Lecture 8

From supervised to unsupervised deep learning

Previous lecture

Lecture	Title	Lecture	Title
1	Intro and history of deep learning	2	AutoDiff
3	Deep learning optimization I	4	Deep learning optimization II
5	Convolutional deep learning	6	Attention-based deep learning
7	Graph deep learning	8	From supervised to unsupervised deep learning
9	Multi-modal deep learning	10	Generative deep learning
11	What doesn't work in deep learning	12	Non-Euclidean deep learning
13	Q&A	14	Deep learning for videos

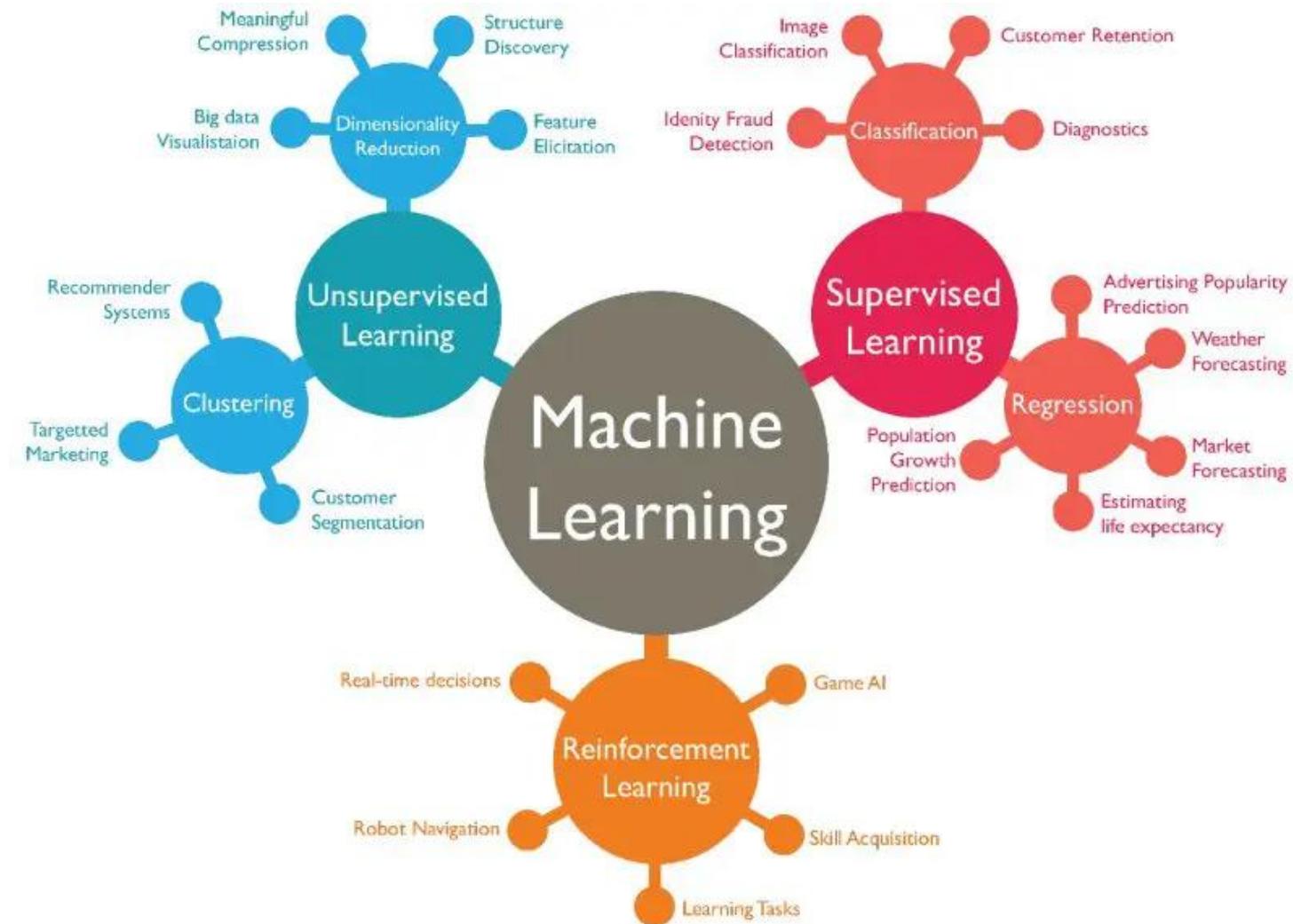
This lecture

Self-supervised learning for vision

Self-supervised learning for language

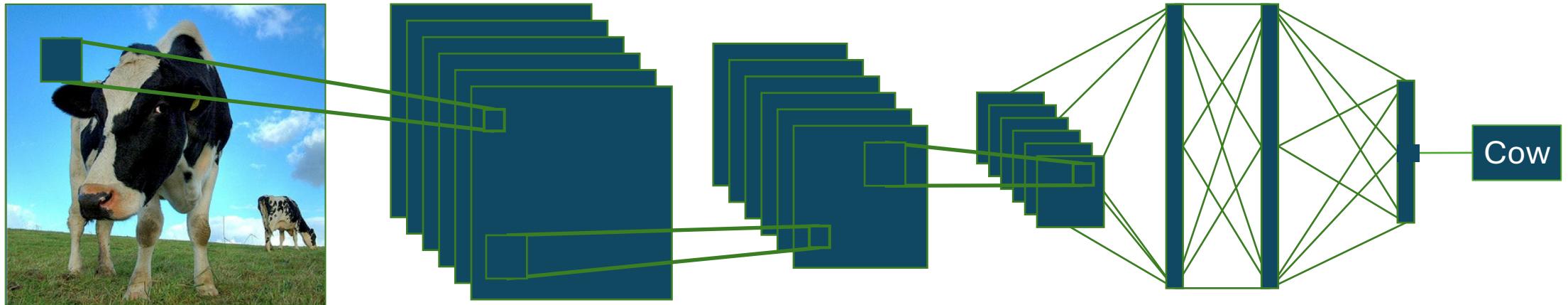
In between supervised and self-supervised learning

Traditional pillars of machine learning



Strength and weakness of supervision in DL

Supervision makes it possible to propagate signals back to train networks.



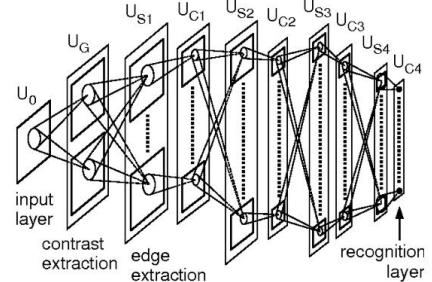
Classification labels no longer the backbone of latest models, why?

Self-supervised learning

Data as fuel for deep learning

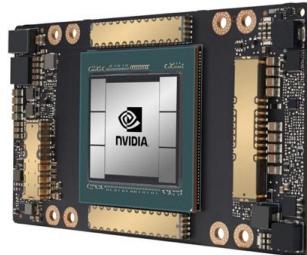
Algorithms

Deep neural networks



Hardware

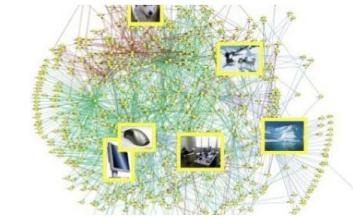
GPUs



Data

Large scale datasets

IMAGENET



The bottleneck of data

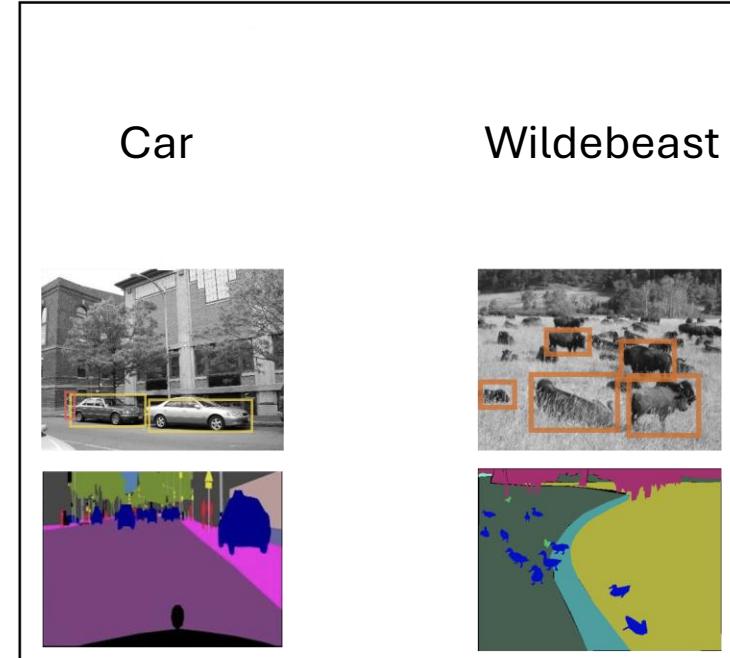
often data are very cheap

Images are often cheap



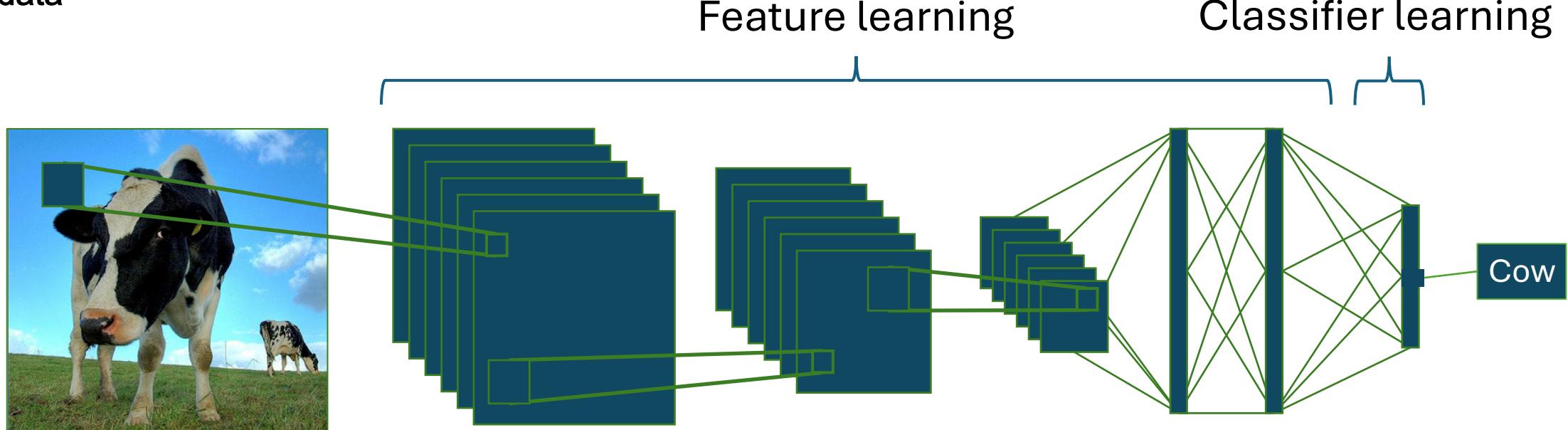
Supervised
Learning

But manual annotations are expensive:
e.g. 30min per image / requiring experts



The two stages of deep learning

all the layers before may not need the labels, they would benefit more from more data



The final layer requires labels, but is that also true for all other layers?

Solving the problem of expensive annotations: self

The idea is that we are going to impose a certain structure on the data, which the network need to learn



Self-supervision

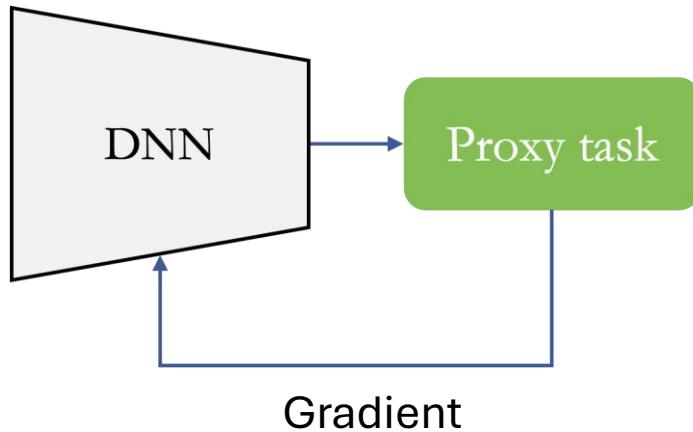
Extract a supervisory signal from the raw data

Main idea of self-supervised learning

Phase 1: Pretraining



Unlabelled data
+ transformations

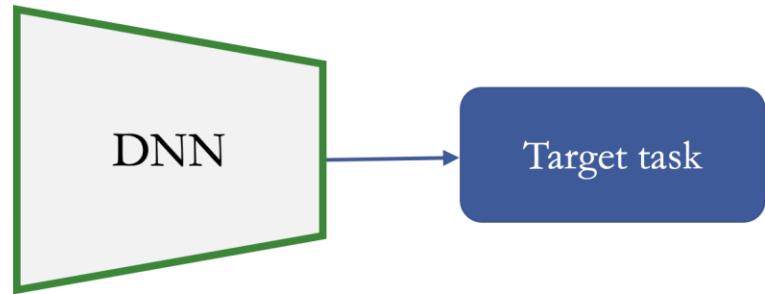


we're gonna define a **custom task** over the data to perform the training without really need all the labels

Phase 2: Downstream tasks



(Sparse) labeled data



Why do we want self-supervised learning?

Reason 1: Scalability



ImageNet
~1 million annotated images



Instagram
~50 billion images floating about

The web is filled with unanontated data.

Reason 2: Generalizability

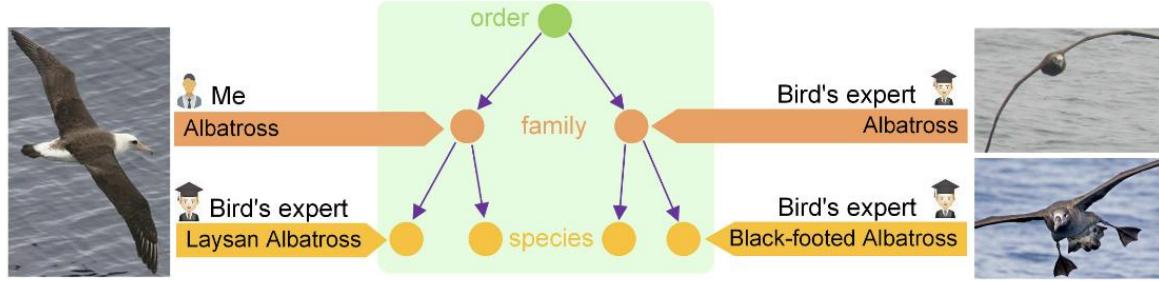


things change over time, but a label is static, so we eventually need to relabel

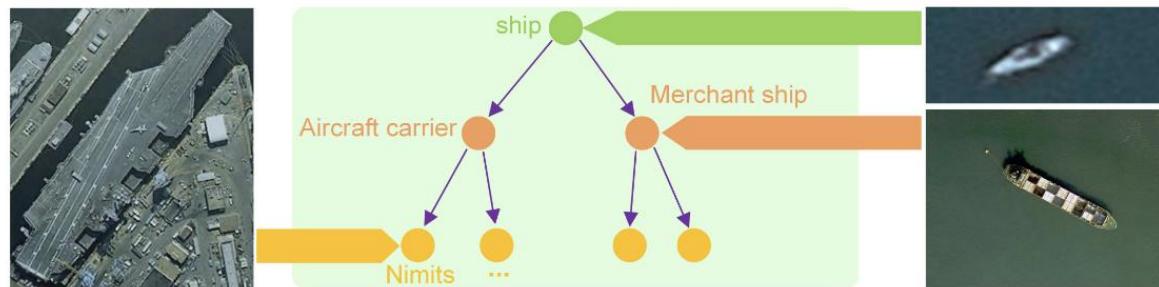


We want models that generalize to many domains and shifts.

Reason 3: Label are not perfect



(a) Differences in domain knowledge and interference from the image occlusion.



(b) Large variations of image resolutions.

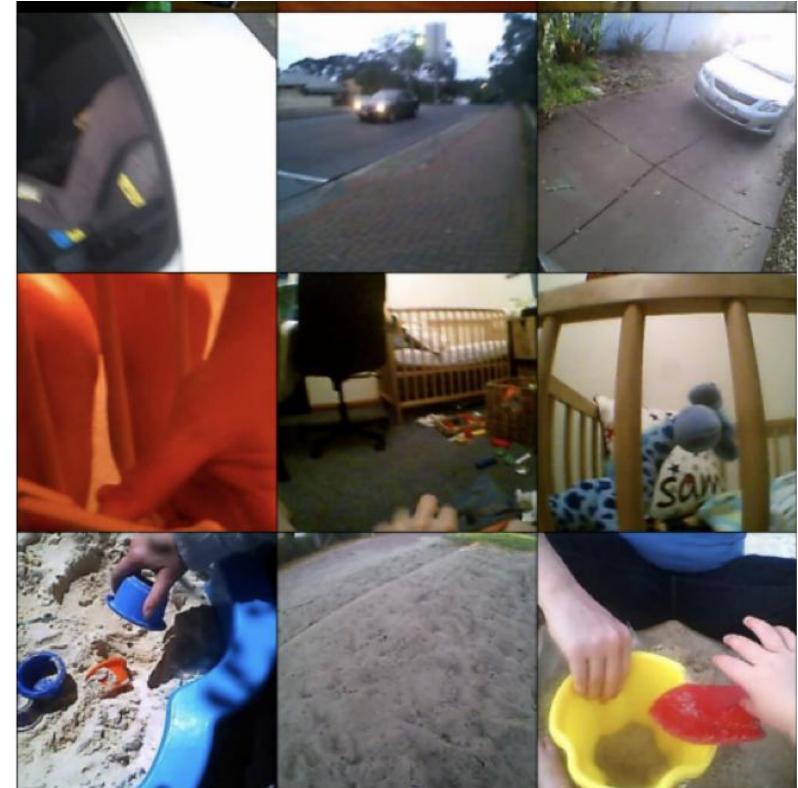
Chen et al. 2022

Labels can be ambiguous, biased, or simply wrong.

Reason 4: Humans are self-supervised

The screenshot shows a website header with 'Meta AI' and navigation links for 'Research', 'Publications', and 'P...'. Below this, a 'RESEARCH' section features the title 'Self-supervised learning: The dark matter of intelligence' and the date 'March 4, 2021'. The main content area contains a paragraph about self-supervised learning, with the first sentence highlighted in green.

As babies, we learn how the world works largely by observation. We form generalized predictive models about objects in the world by learning concepts such as object permanence and gravity. Later in life, we observe the world, act on it, observe again, and build hypotheses to explain how our actions change our environment by trial and error.



Still a lot of lessons from human learning that can be transferred.

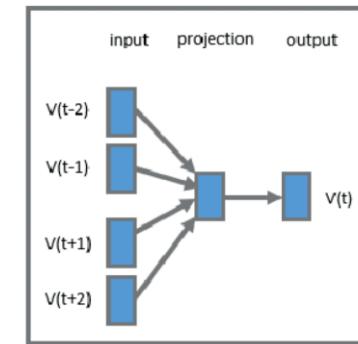
How do we train deep networks
without labels?

Self-supervised visual learning

The first popular domain for self-supervised learning.

Main idea: exploit the structure of images and videos to learn without labels.

Goal is not to develop new algorithms, but borrow losses from supervised learning and think of your own loss functions.



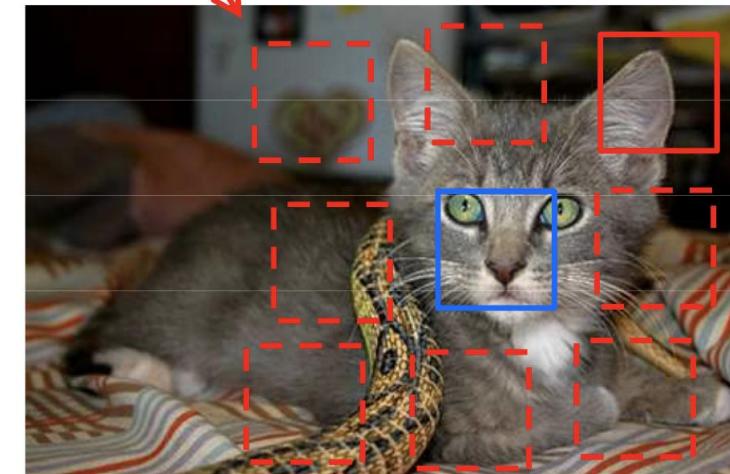
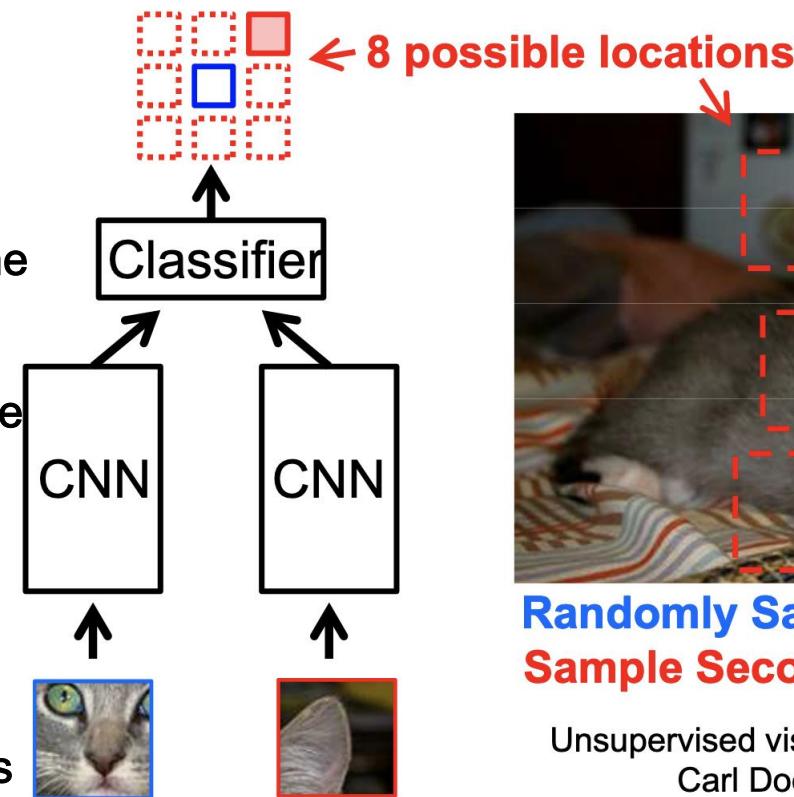
Motivated from NLP

Early attempt: relative positioning

Randomly select one patch from an image, then sample a second patch from one of its 8 neighboring positions

Classification task: The network receives both patches as input and must predict which of the 8 possible spatial locations the second patch came from relative to the first

Automatic labels: Since you extracted the patches yourself, you know their relative positions - this becomes your free supervisory signal for backpropagation

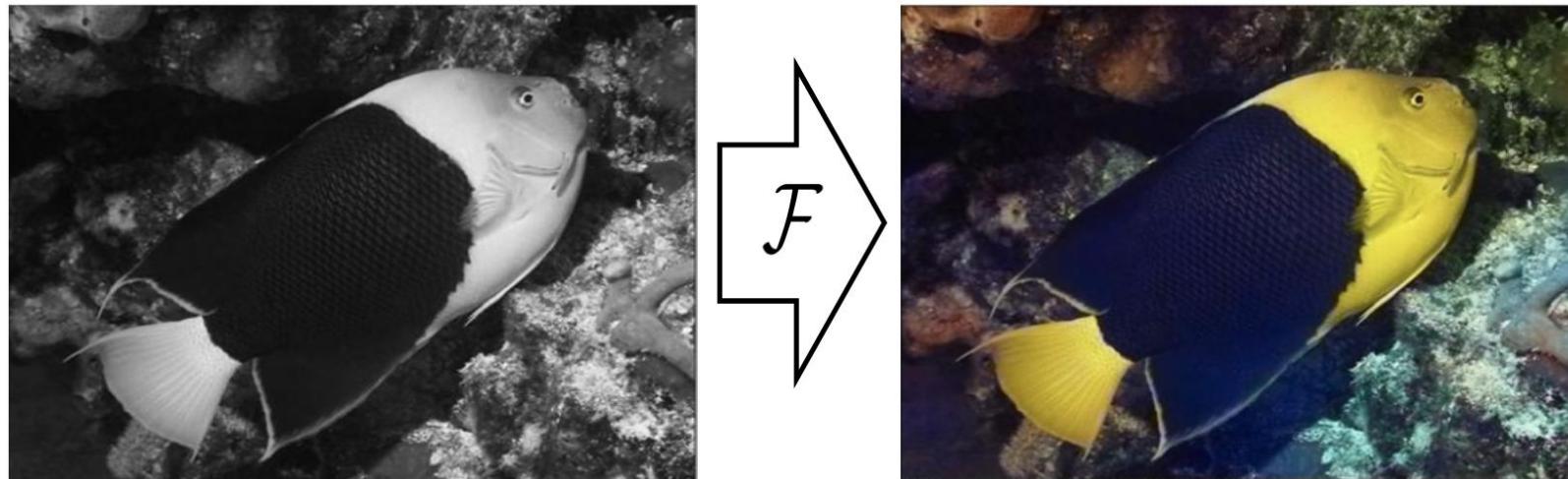


**Randomly Sample Patch
Sample Second Patch**

Unsupervised visual representation learning by context prediction,
Carl Doersch, Abhinav Gupta, Alexei A. Efros, ICCV 2015

Learning by coloring

remove the color and learn to add them back



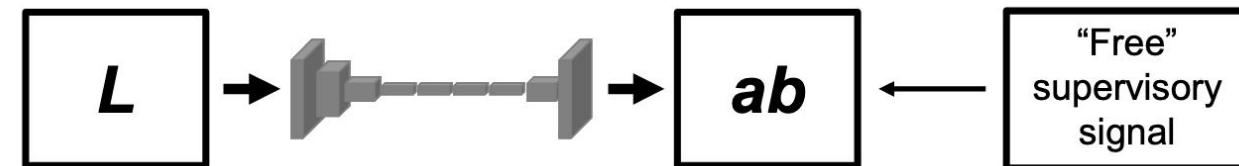
Grayscale image: L channel

$$\mathbf{X} \in \mathbb{R}^{H \times W \times 1}$$

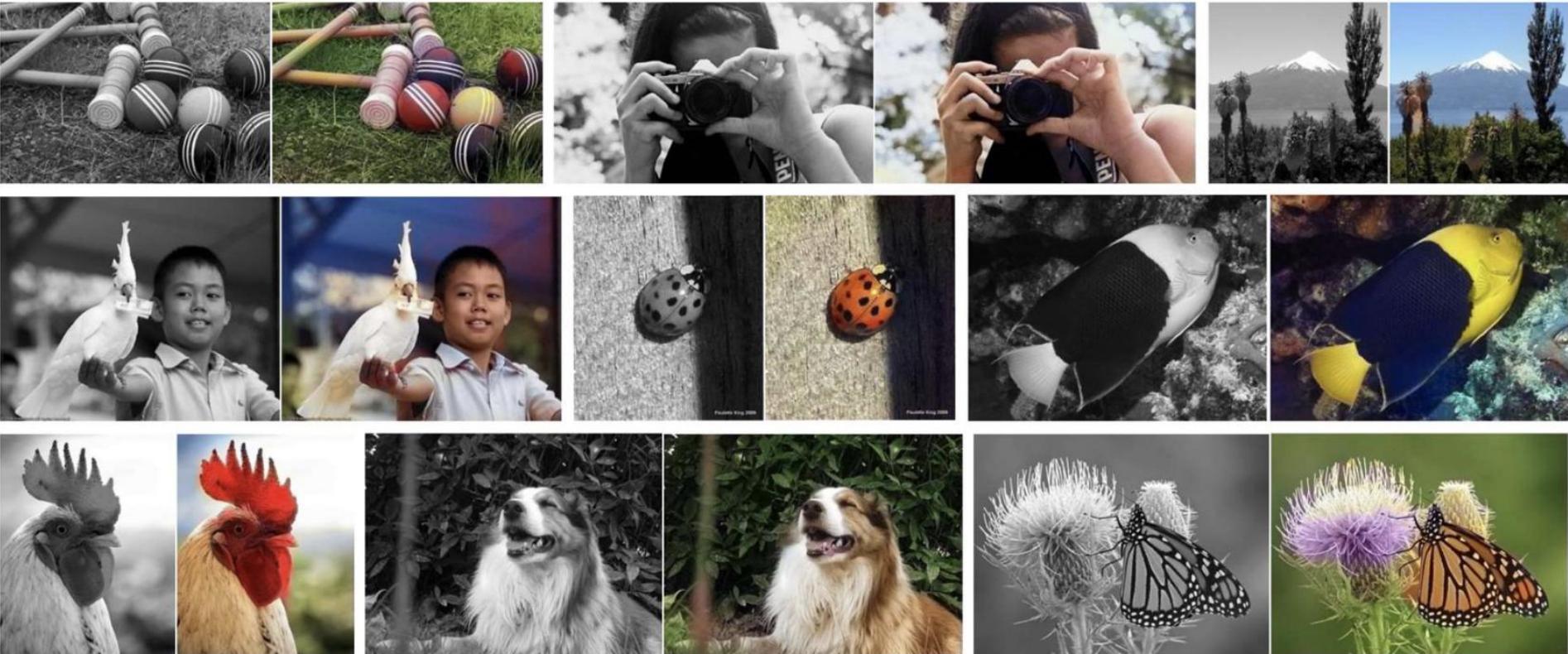
so basically in self supervised learning from a modified input we want to learn to reconstruct the original input

Concatenate (L, ab)

$$(\mathbf{X}, \hat{\mathbf{Y}})$$



Visual results



The representations for learning color can be used to initialize a new network.

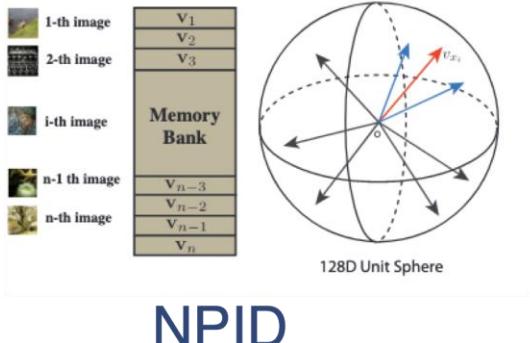
Learning by rotations



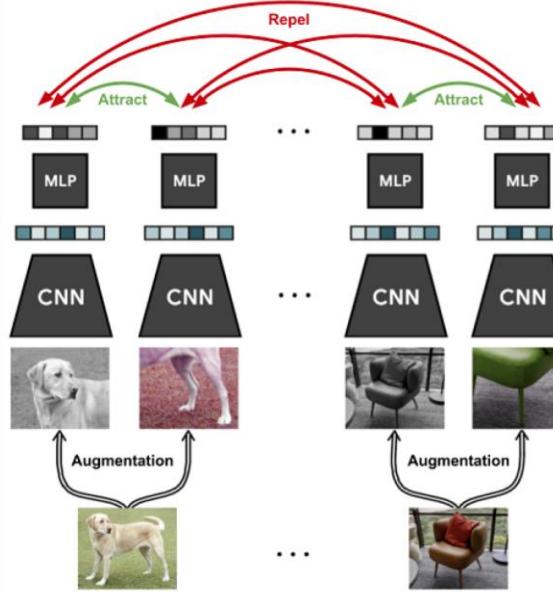
Assumption: if we know the object, we understand which rotation is most natural.

Modern approach: contrastive self-supervision

I'm gonna take an image from a batch and apply to it a set of different transformation, i want group my similar images together and respingere the others



I want to cluster (attract) similar even though transformed images, and repel the different ones



SimCLR

Enforces image uniqueness and augmentation invariance.

contrastive learning, it want alignments and uniformity

the clustering wants groups as tight as possible (alignment) and as far as possible from each other (uniformity)

The contrastive loss for positive pairs i, j :

$$\ell_{i,j} = -\log \frac{\exp(\text{sim}(z_i, z_j)/\tau)}{\sum_{k=1}^{2N} [k \neq i] \exp(\text{sim}(z_i, z_k)/\tau)},$$

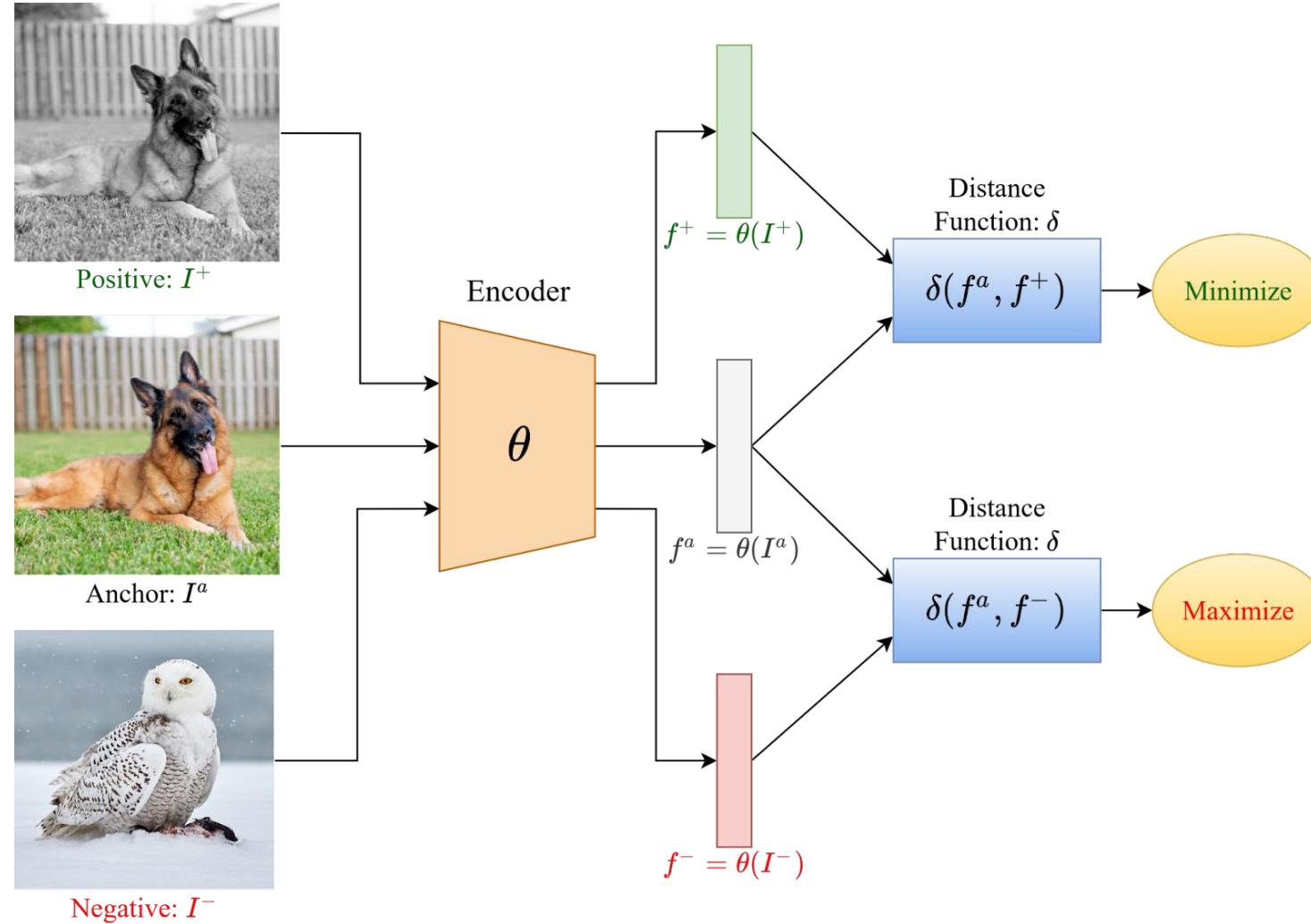
with z_i, z_j embeddings for images i and j ,
 τ a temperature, $\text{sim}()$ is the dot-product

"non-parametric" softmax

contrastive learning is not tighted to self supervised is just a type of loss

note that the dimension of the denominatore is a function of the batch size and not the number of class like in the original softmax

How to train with contrastive losses



Self-supervised learning is conservative supervised learning

In supervised learning the learning is by labelled examples, because for each samples someone is gonna telling me the label.
In self-supervised instead, the learning is by rule, there are a set of transformations which doesn't change the semantic of my sample

Contrastive supervised learning:

Pull samples of same class together, push others away.

Contrastive self-supervised learning:

Pull augmented versions of same sample together, push others away.

Self-supervised learning is learning augmentation invariance



(a) Original



(b) Crop and resize



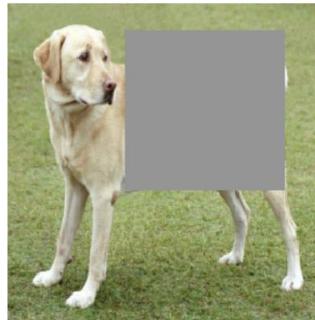
(c) Crop, resize (and flip)



(d) Color distort. (drop)



(e) Color distort. (jitter)

(f) Rotate $\{90^\circ, 180^\circ, 270^\circ\}$ 

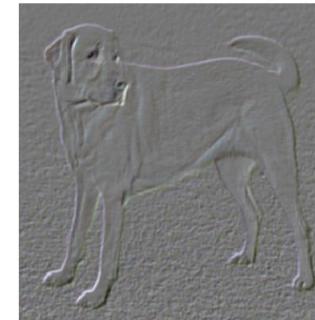
(g) Cutout



(h) Gaussian noise



(i) Gaussian blur



(j) Sobel filtering

We want augmentations of a sample to lead to the same embedding representation.

Self-supervised video learning

Videos have a super strong extra signal to learn from: time.

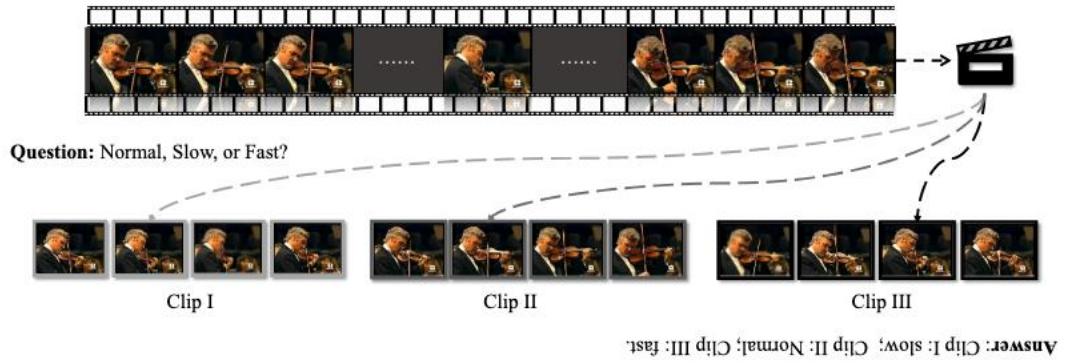
How can time be used for pretext tasks?

Predict temporal order.

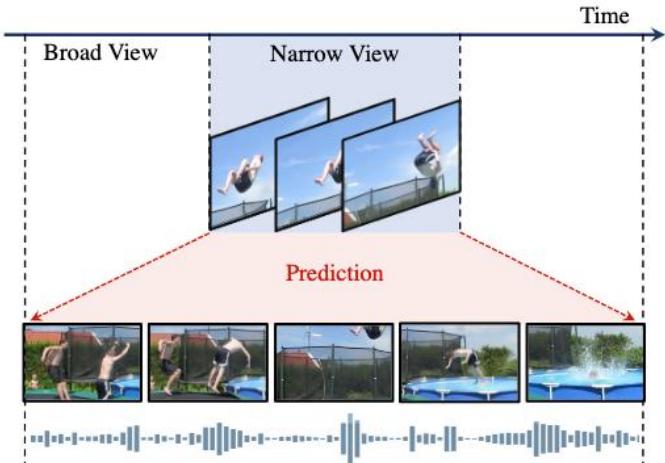
Predict whether video is played in reverse or not.

Predict alignment between video and audio.

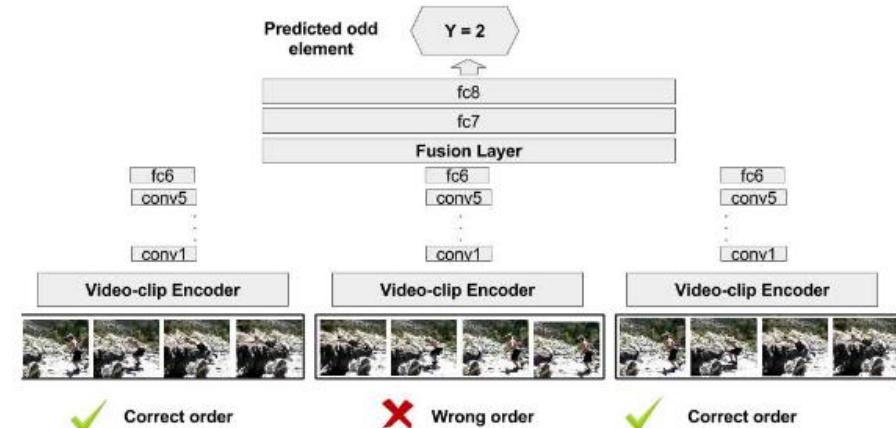
Examples of self-supervised video learning



Wang et al. (2020): Predict pace.



Recasens et al. (2021): Narrow to broad prediction.



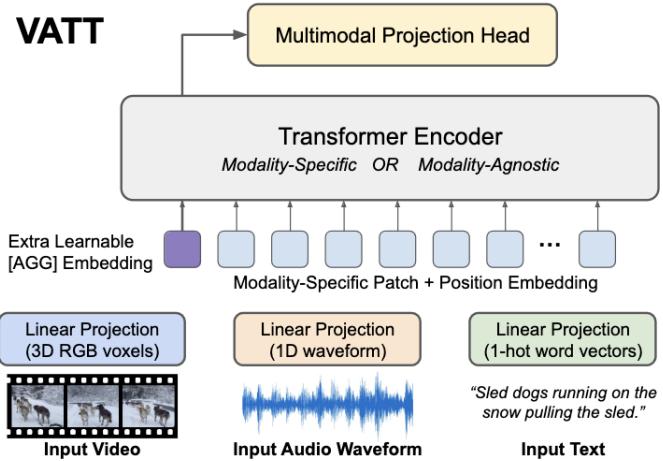
Basura et al. (2017): Predict odd-one-out.



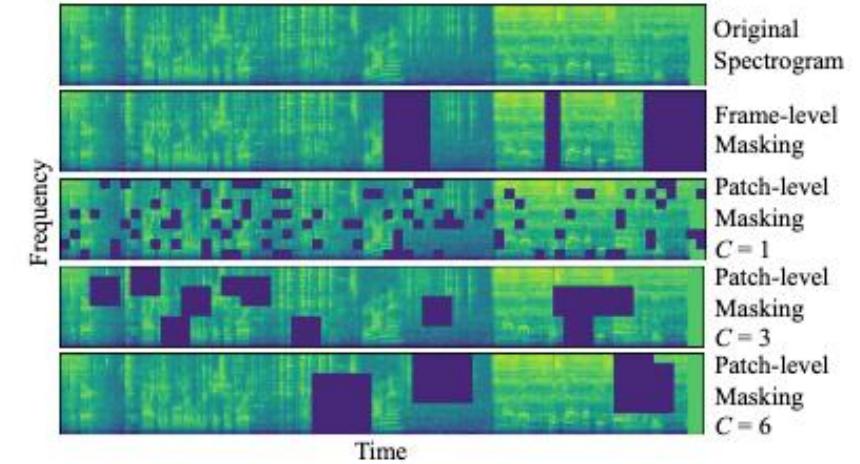
Xu et al. (2019): Predict clip order.

Break

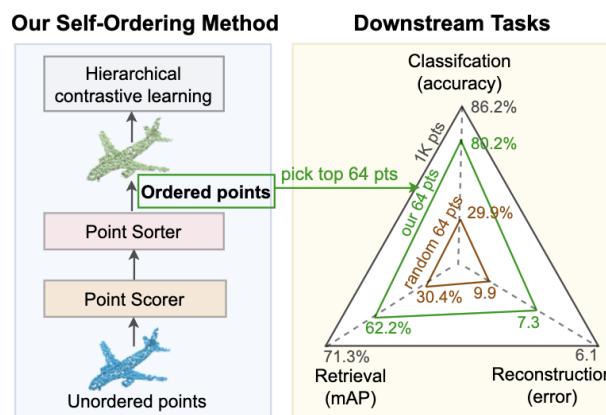
Self-supervised learning on other modalities



[Akbari et al. NeurIPS 2021]



[Gong et al. AAAI 2022]

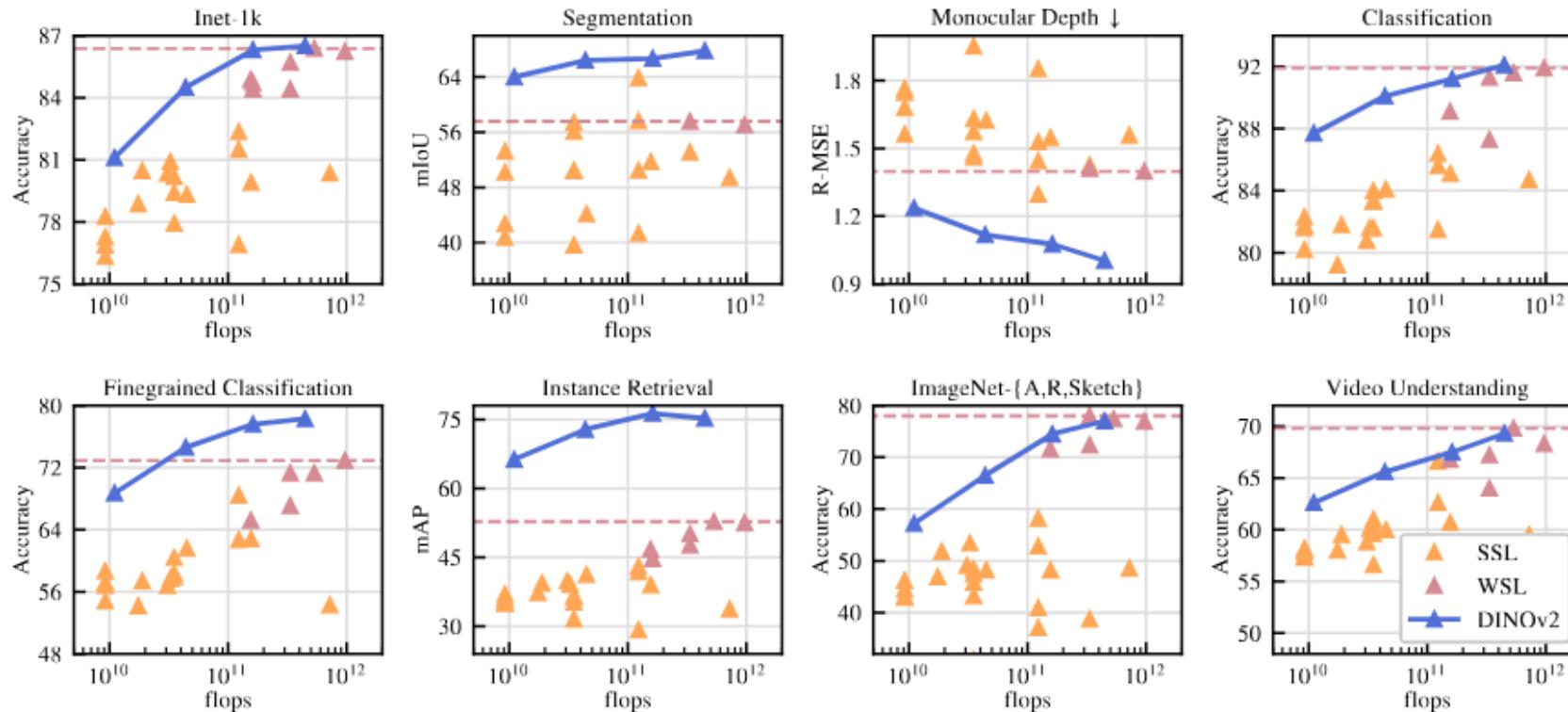


[Yang et al. ICCV 2023]

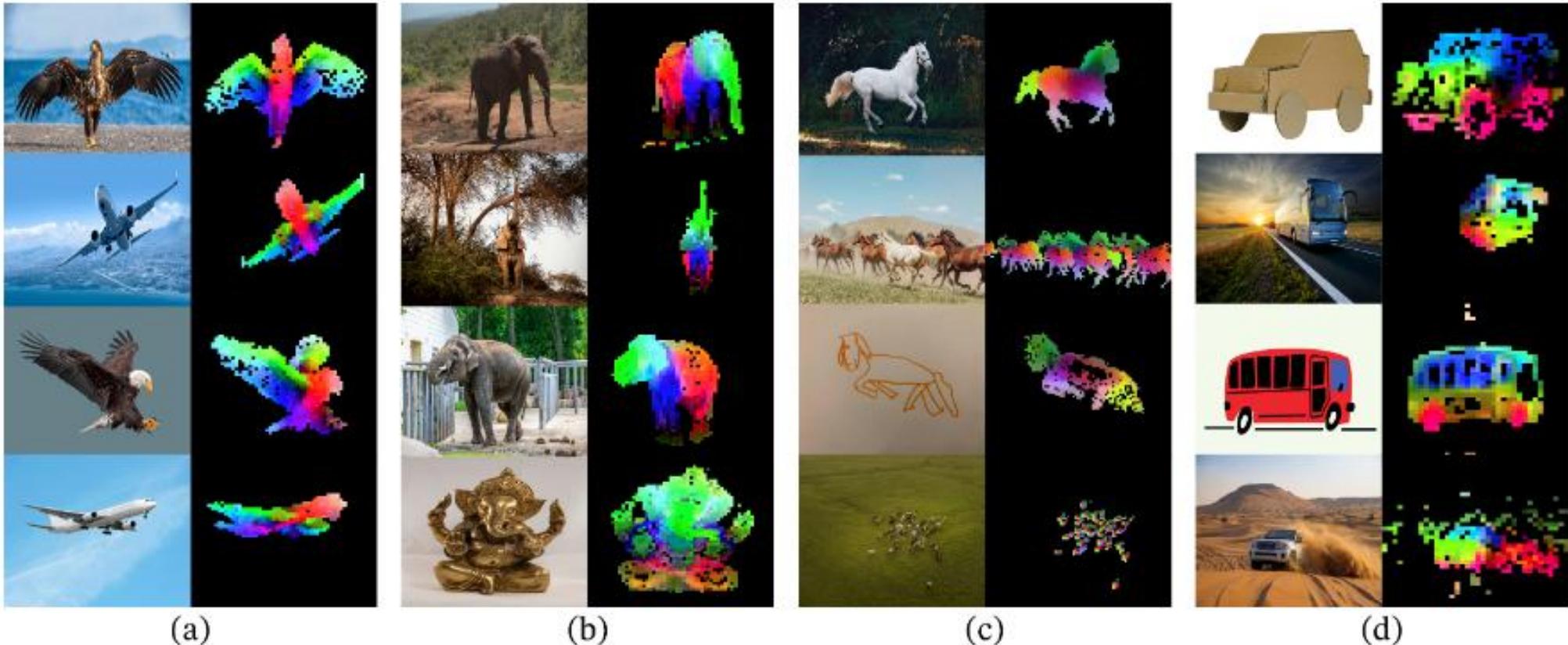
How good are self-supervised models? A DINOv2 study

DINOv2 (2024): Contrastive learning + patch-level masking + tricks + 142M dataset.

a model trained on many transformation without label, works better than a model trained on fewer samples but with label. Because the formers are able to learn structural invariances



How good are self-supervised models? A DINOv2 study



Supervised networks struggle when being deployed in new settings,
self-supervised networks thrive in such settings.

Self-supervised learning for language

What if our data is a collection of sentences?

why is it not unsupervised?
because the human wrote the phrases
in an ordered way

“The quick brown fox jumps over the lazy dog.”

I.e., how can we train Large Language Models on the internet?

Masked Language Modelling

Main idea is simple: remove some tokens and predict them.

Set of classification labels: All tokens.

Targets: Tokens that were removed.

Just like self-supervised visual learning, the problem falls back to a standard classification setup, but now with “free labels”.

Masked Language Modelling

Standard setting: Sample 15% of tokens and replace with [MASK].

“The quick brown [MASK] jumps over the [MASK] dog.”

Modified MLM: Sample 15% of tokens. Replace 80% with [MASK], 10% with random token, and 10% left unchanged.

“The quick brown [oven] jumps over the [maybe] dog.”

“The quick brown [fox] jumps over the [lazy] dog.”

Modified Masked Language Modelling

For 80% of the sampled tokens, we simply need to predict the masked input.

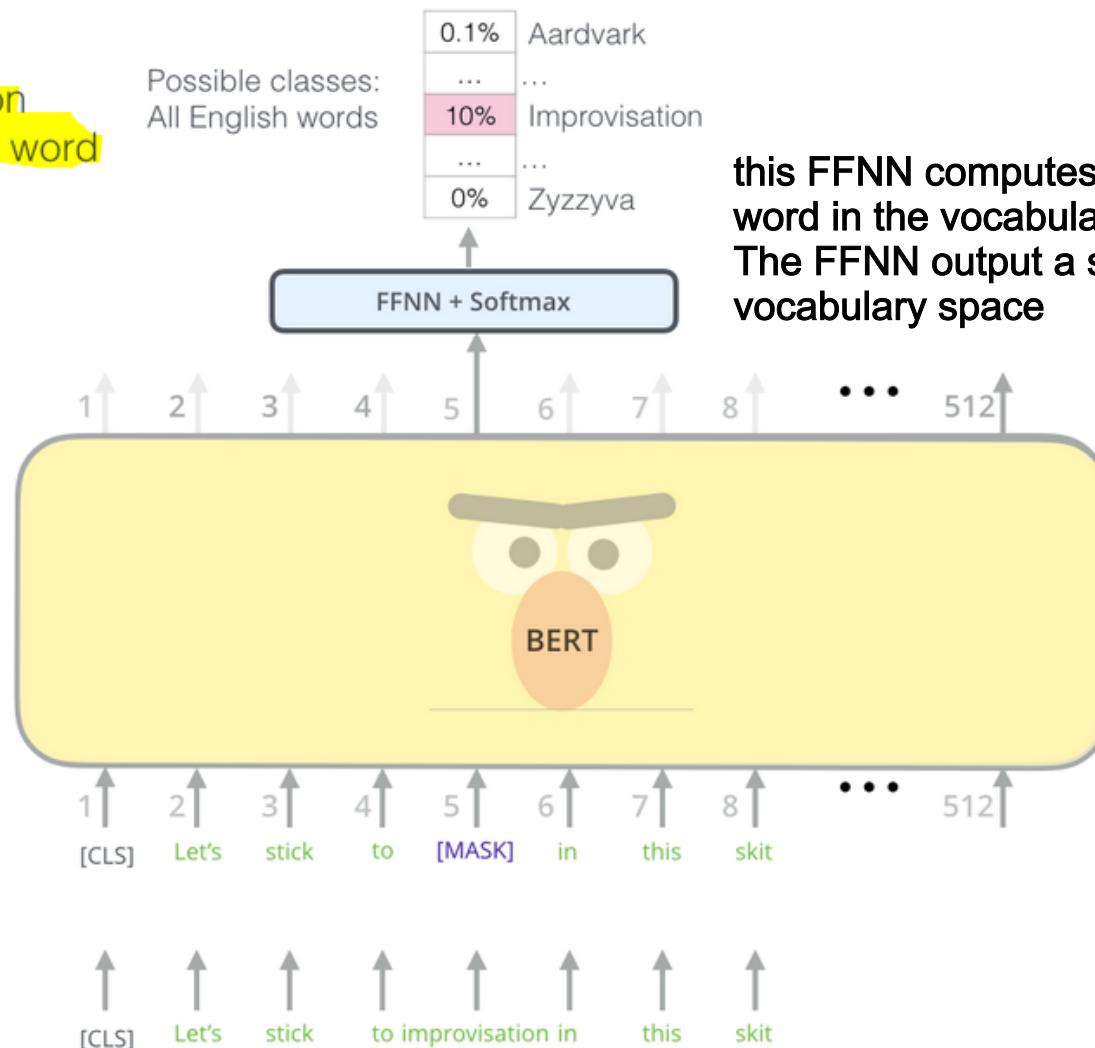
For 10% of the tokens, the model needs to figure out that the word needs to be replaced.

For the remaining 10%, the model needs to figure out to do nothing.

Use the output of the
masked word's position
to predict the masked word

Randomly mask
15% of tokens

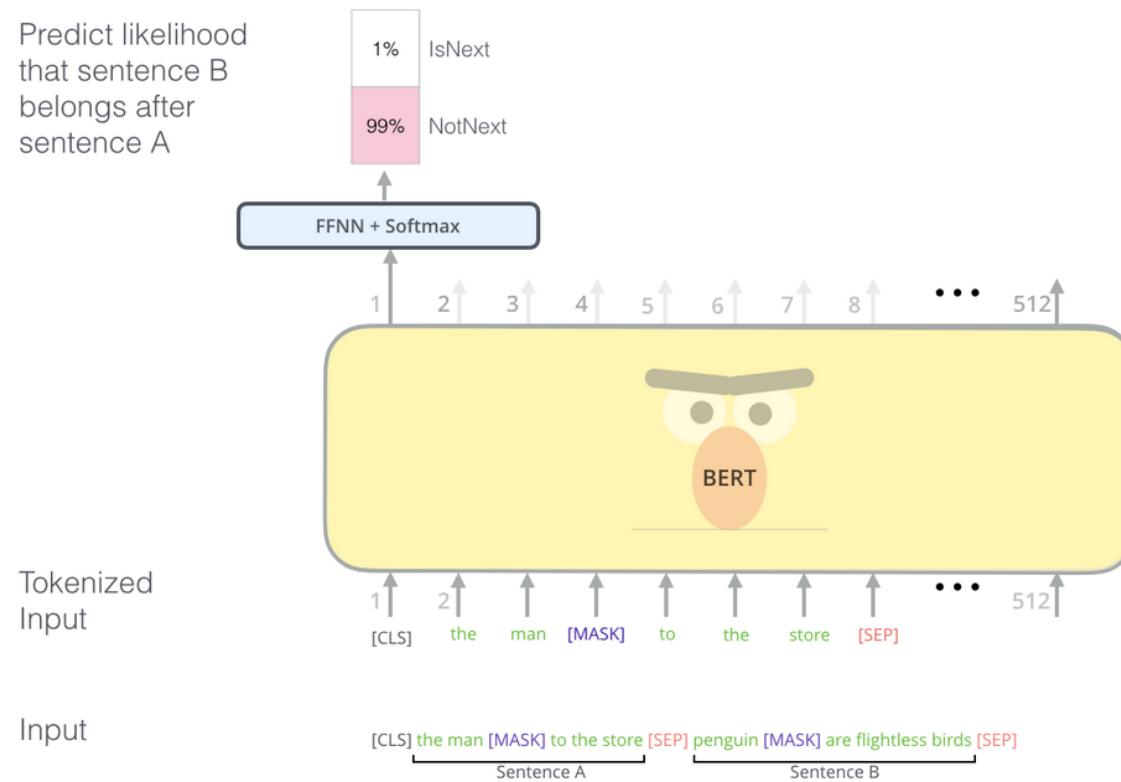
Input



this FFNN computes the scores (logits) for each word in the vocabulary being the masked word. The FFNN output a score vector in the vocabulary space

Next Sentence Prediction

Main idea: Given two sentences, predict whether the first follows the second.



How to automatically generate “labels” for this setting?

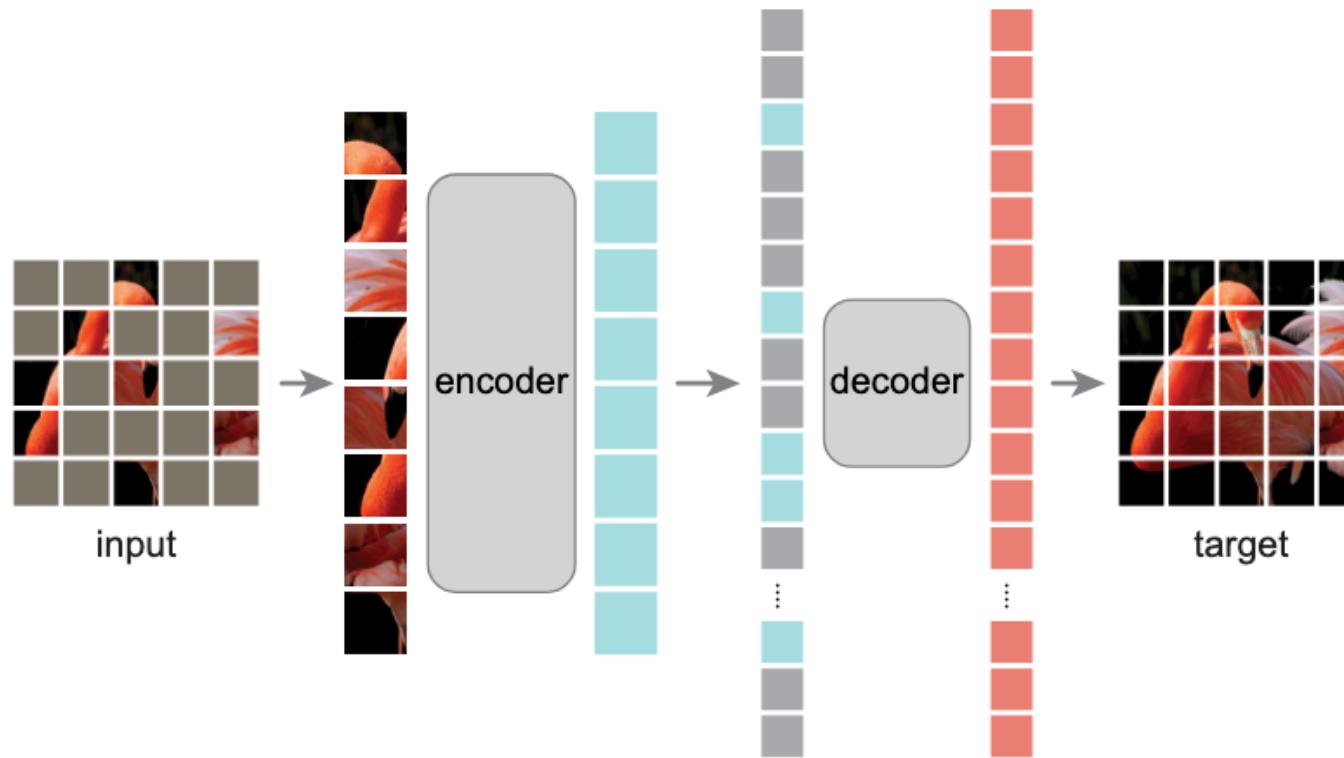
Take two sentences that actually appear consecutively in the original text corpus. Since they were originally adjacent, you label this pair as “IsNext” or positive

Take the first sentence, then randomly select a second sentence from somewhere else in the corpus (from a different document or a different part of the same document). Since these were not originally adjacent, you label this pair as “NotNext” or negative.

MLM in vision: masked autoencoding

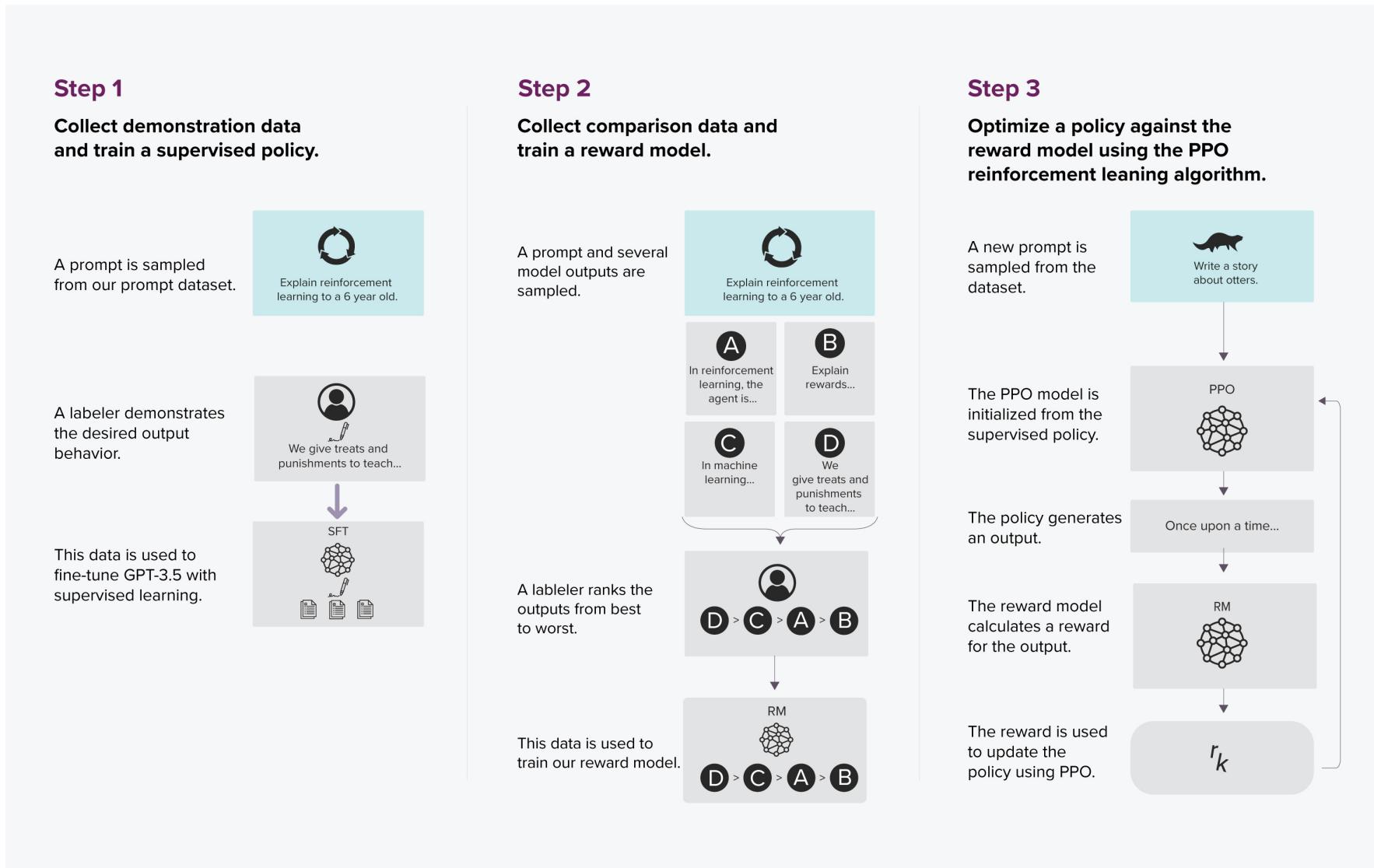
Many ideas from one domain are inspiration for the next domain.

I crossed out some patches, I
feed them to the net and I
learn to predict them back



Revisiting RLHF

reinforcement learning human feedback



RLHF from a supervision perspective

Self-supervision (GPT) is not enough for Large Language Models (ChatGPT).

Prompt: Explain why we need water to survive as humans.

(Imaginary) GPT: Explain why humans die when not given any water.

What is going on here?

We lack alignment with the intent of the user input.

This needs human supervision.

Step 1

Collect demonstration data and train a supervised policy.

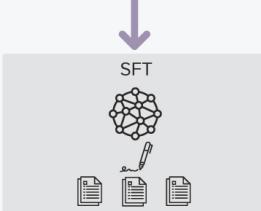
A prompt is sampled from our prompt dataset.



A labeler demonstrates the desired output behavior.



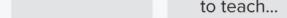
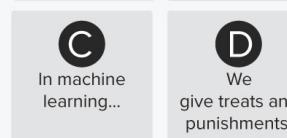
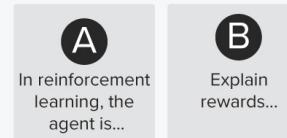
This data is used to fine-tune GPT-3.5 with supervised learning.



Step 2

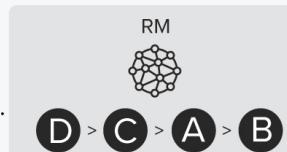
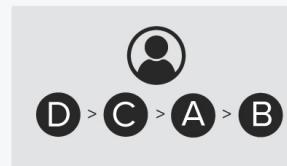
Collect comparison data and train a reward model.

A prompt and several model outputs are sampled.



A labeler ranks the outputs from best to worst.

This data is used to train our reward model.



Step 3

Optimize a policy against the reward model using the PPO reinforcement leaning algorithm.

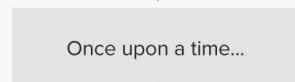
A new prompt is sampled from the dataset.



The PPO model is initialized from the supervised policy.



The policy generates an output.



The reward model calculates a reward for the output.



The reward is used to update the policy using PPO.

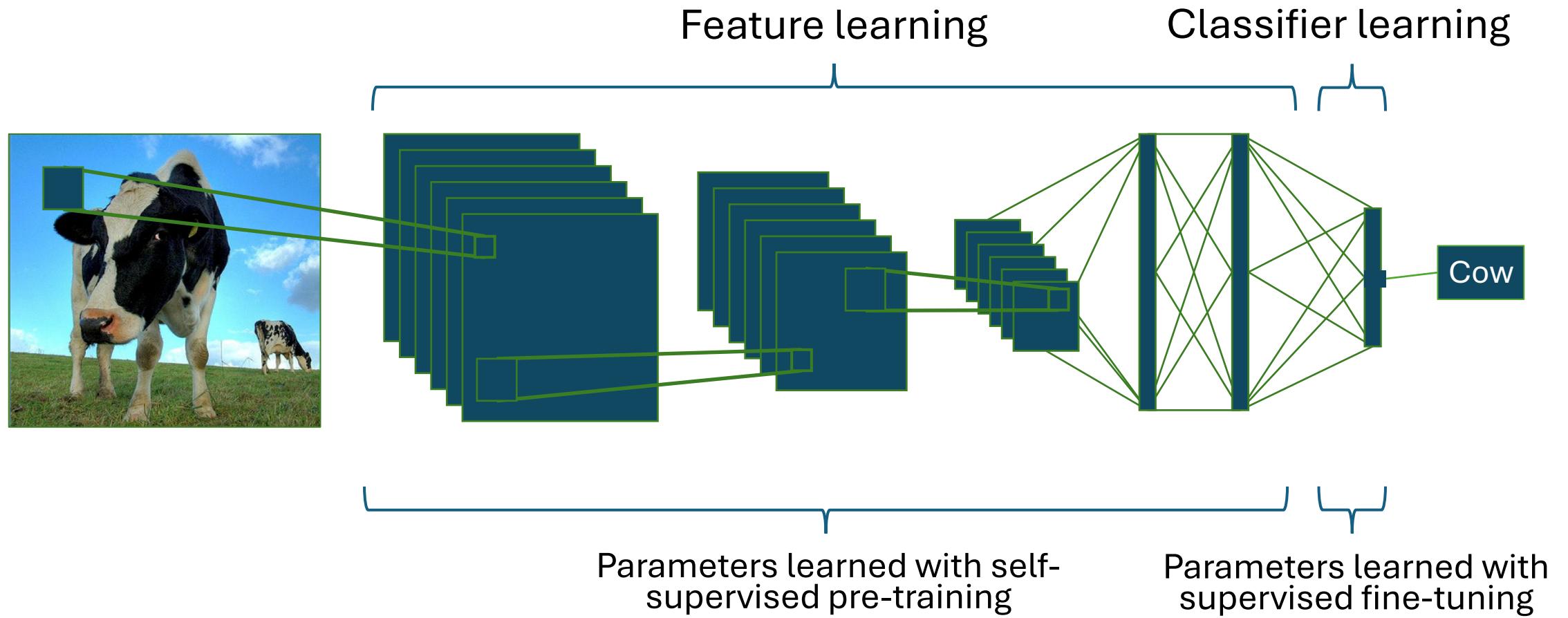


LLM = Transformer + self-supervised
pre-training + human aligned tuning

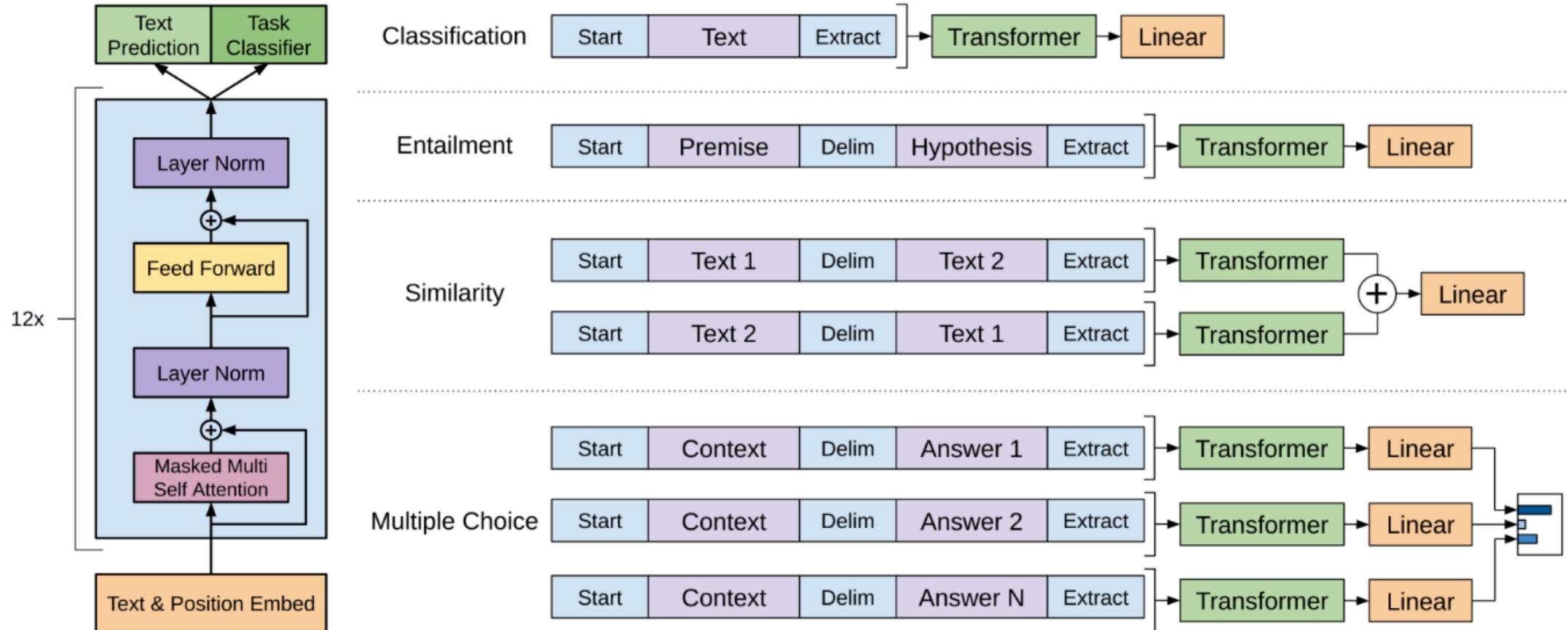
Discussion: why are they so
good? And is this all you need for
deep learning?

Is there something in between
supervised and self-supervised
learning?

Classical setup: pre-training and fine-tuning



Examples of fine-tuning in language

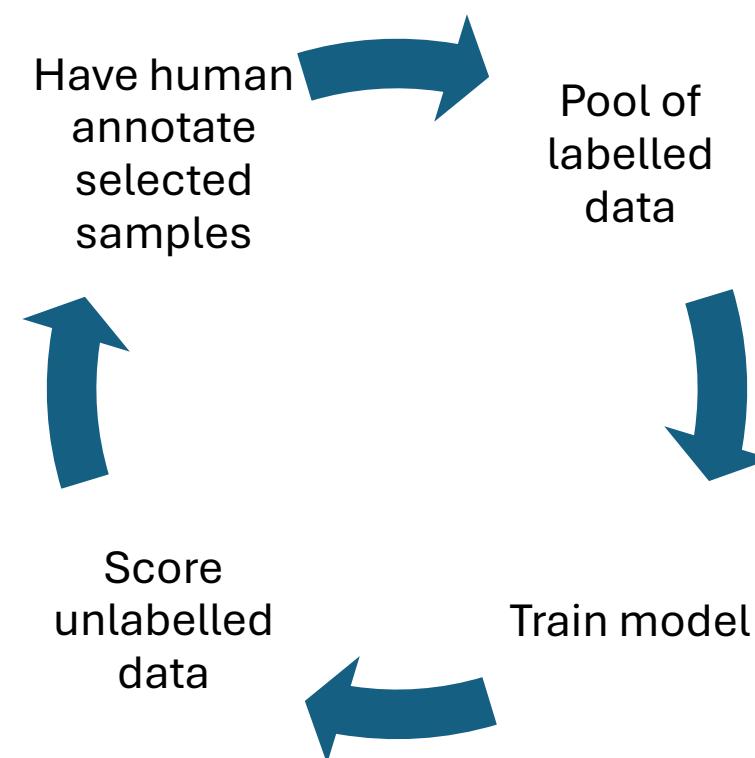


Active learning

i have 2 sets, one unlabelled and one labelled, I want to use the feature learned from the labeled set on the unlabelled set

Assume we only have an unlabelled training set. Is it possible to simultaneously label samples and train a model?

Active learning is a machine learning approach where you start with mostly unlabeled data and iteratively select the most informative samples for a human to label, making the labeling process more efficient.



Which samples to select?

Random: randomly select (ignore scoring).

Most uncertain: closest to the decision boundary, or lowest norm in embedding space (second to last layer), or highest likelihood entropy.

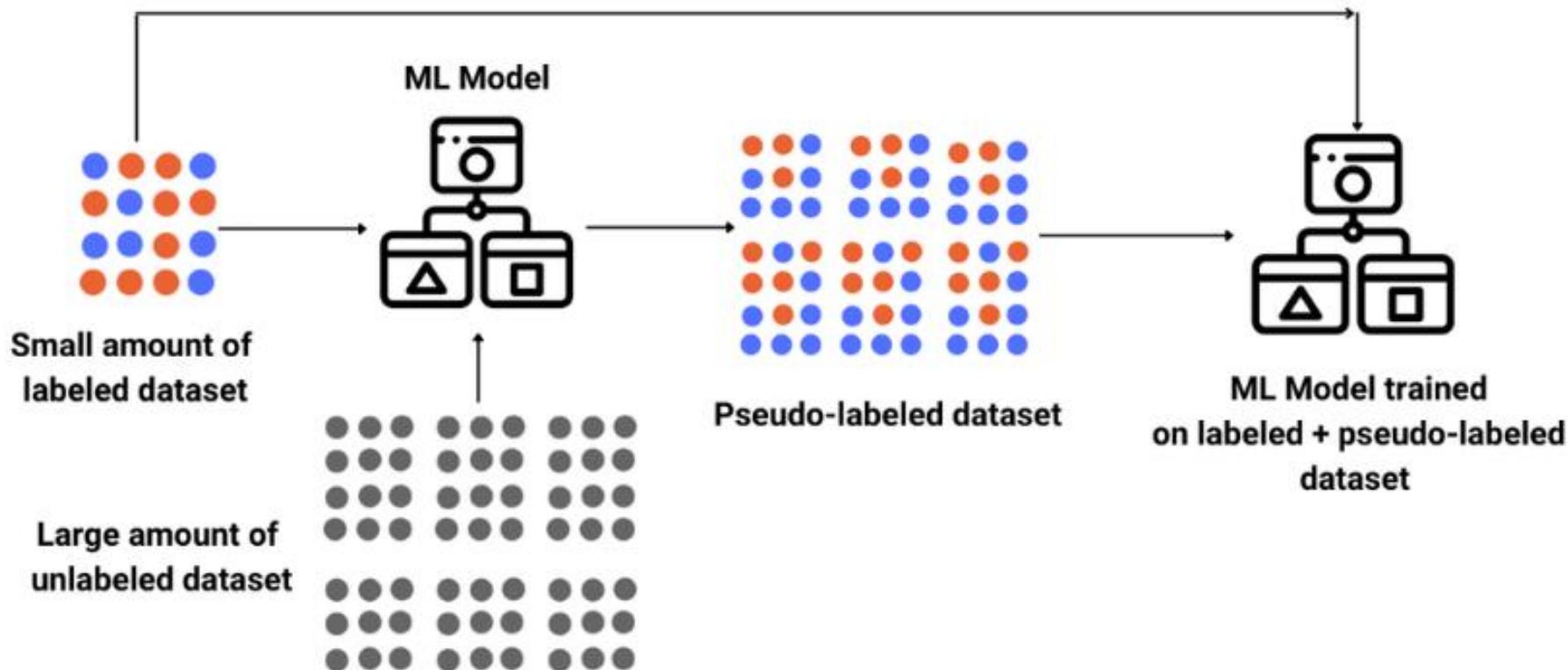
Group-based metrics: Uniformity over classes to avoid biases.

Mix: Combine a mix of X% random and (100-X)% uncertain+group.

Semi-supervised learning

pseudo-labeling (or self-training)

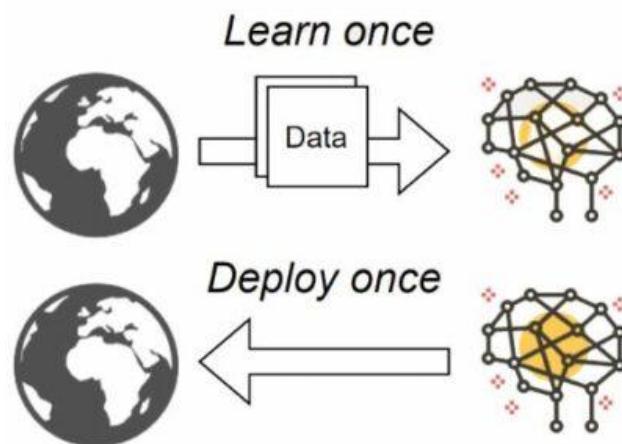
can be seen as a static moment in time of active learning



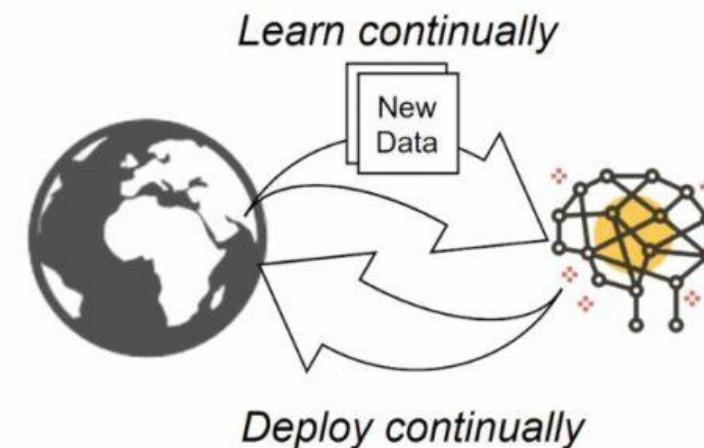
Continual learning

In the real world, there is no such thing as a static dataset.

Static ML



Adaptive ML



catastrophic
forgetting

Shockingly, we can't just train on new data! Much more in lecture 11.

Is self-supervised learning truly
without supervision?

My view on supervised vs self-supervised learning

Supervised learning

Label by sample.

Invariance defined
at global semantic level.

Self-supervised learning

Label by rule.

Invariance defined
at geometric or local semantic
level.

Self-supervised learning is conservative supervised learning from pre-defined invariances.

Next lecture

Lecture	Title	Lecture	Title
1	Intro and history of deep learning	2	AutoDiff
3	Deep learning optimization I	4	Deep learning optimization II
5	Convolutional deep learning	6	Attention-based deep learning
7	Graph deep learning	8	From supervised to unsupervised deep learning
9	Multi-modal deep learning	10	Generative deep learning
11	What doesn't work in deep learning	12	Non-Euclidean deep learning
13	Q&A	14	Deep learning for videos

Learning and reflection

TODO

Thank you



Deep Learning 1

2025-2026 – Pascal Mettes

Lecture 9

Multi-modal deep learning

Previous lecture

Lecture	Title	Lecture	Title
1	Intro and history of deep learning	2	AutoDiff
3	Deep learning optimization I	4	Deep learning optimization II
5	Convolutional deep learning	6	Attention-based deep learning
7	Graph deep learning	8	From supervised to unsupervised deep learning
9	Multi-modal deep learning	10	Generative deep learning
11	What doesn't work in deep learning	12	Non-Euclidean deep learning
13	Q&A	14	Deep learning for videos

This lecture

Vision-language models.

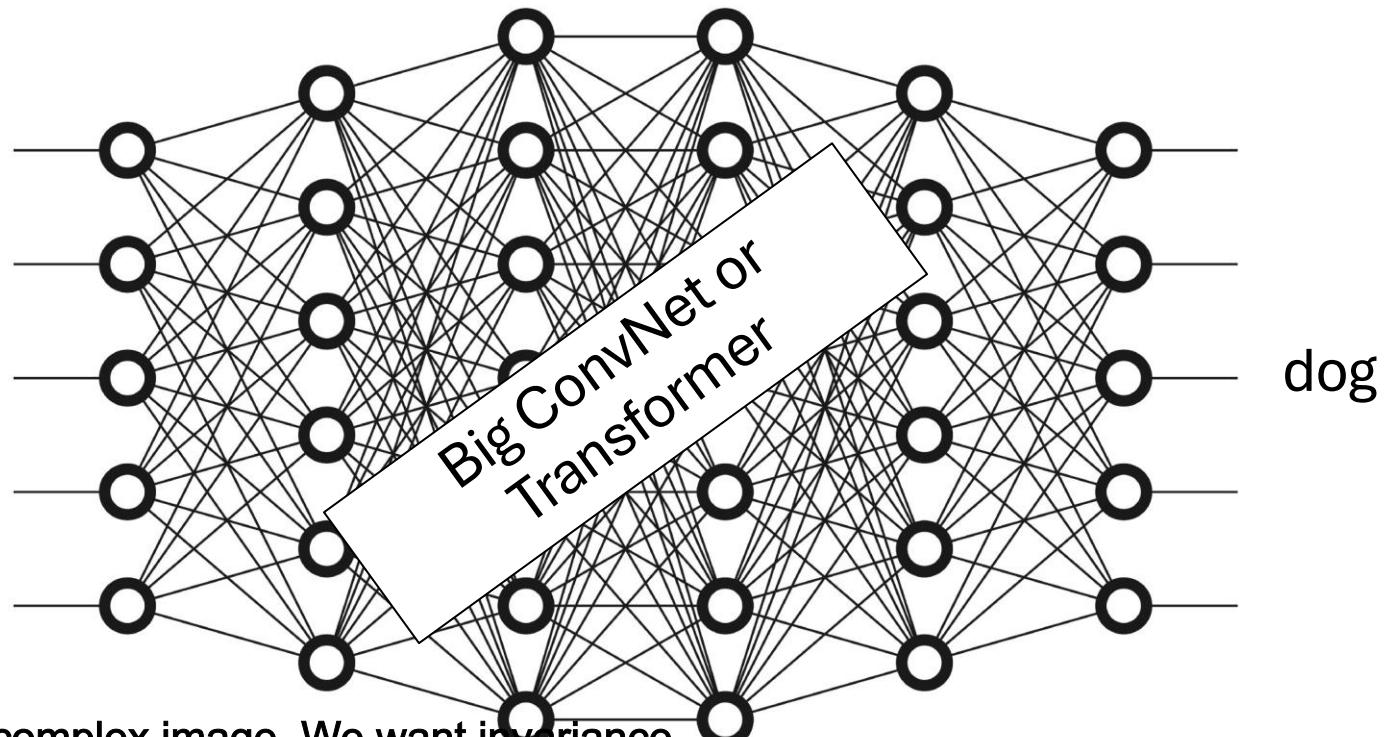
Prompting and improving vision-language models.

Multi-modal LLMs.

Canonical supervised learning



Also I can generalize only to thing that I've seen during training



here we have a very simple label for a very complex image. We want invariance

If I add different labels for classification the time to classify increase.

Even if I have more labels they are going to be not enough and the computational complexity increase

Issues with supervised classification

Labels are deemed independent.

A false assumption. Any mistake is equally bad, which leads to real-world issues.

An image is more complex than a label.

An image is a complex scene, with multiple objects in interaction.

What happens when we see something we didn't train on?

We can never generalize to new settings.

Deep learning beyond class labels: the origins

some tasks which
seems very difficult
becomes doable if we
have an intermediate
description



Which bird species is this?

American Robin
White head
White belly

Crow
Black head
Black belly

Painted Bunting (male)
Blue head
Red belly

so I want to built an attribute classifier in order to generalize to any new category

ML model from input to attributes and then an hard coded inference systems that goes from attributes to class

Zero-shot learning

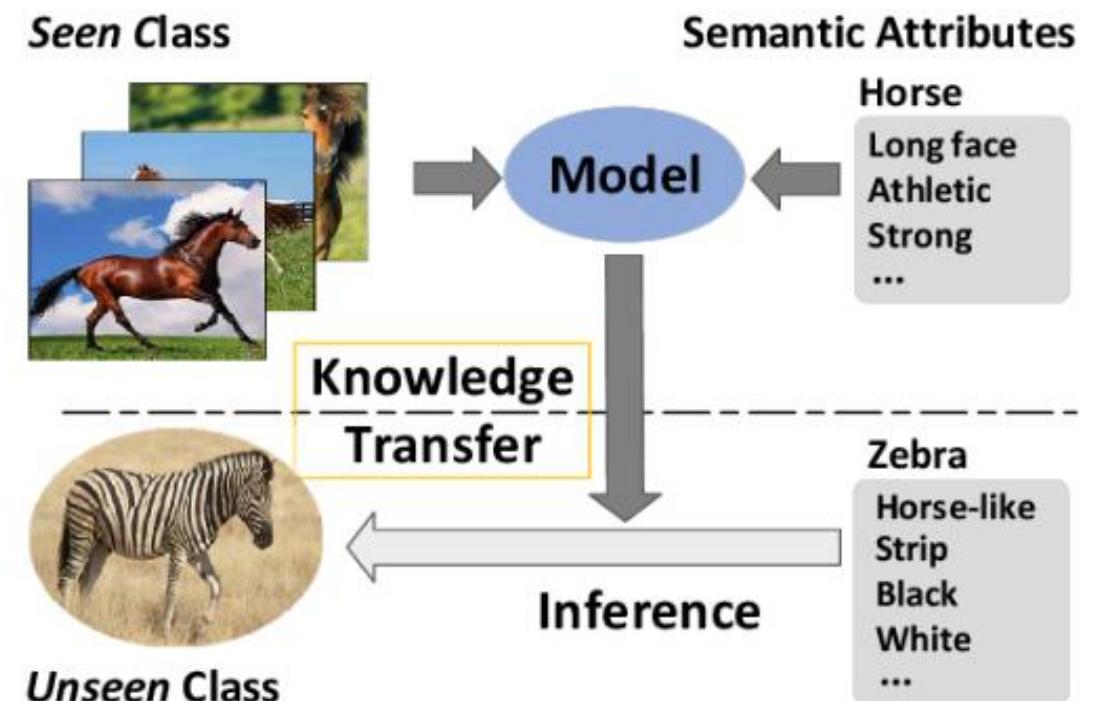
the inference systems looks at the nearest neighbor in the attributes space

Original idea: Instead of predicting a class label per image, predict a set of shared attribute labels.

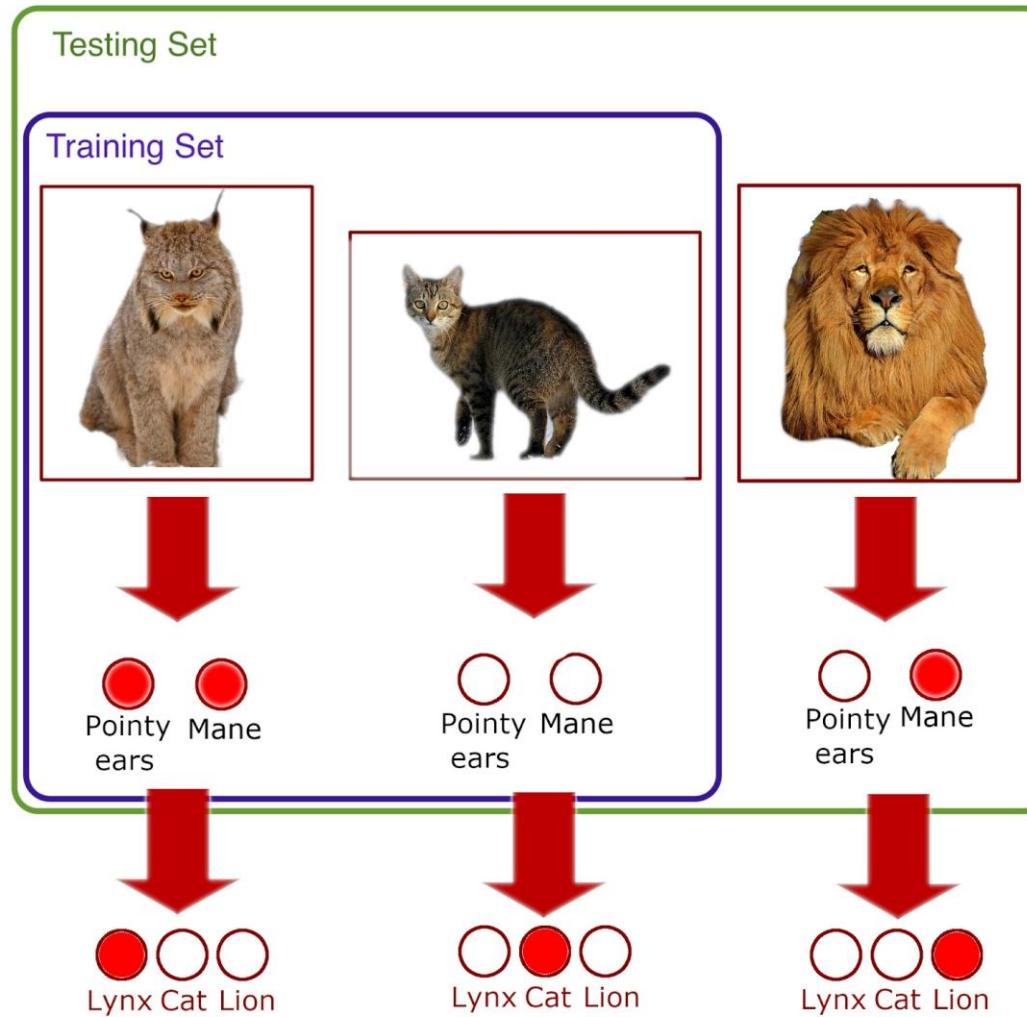
For each class, pre-specify the correct attributes.

During inference, predict all attributes and select class with highest attribute similarity.

we want the labels in a semantic space represented as a continuous vector



Recognition beyond the training set



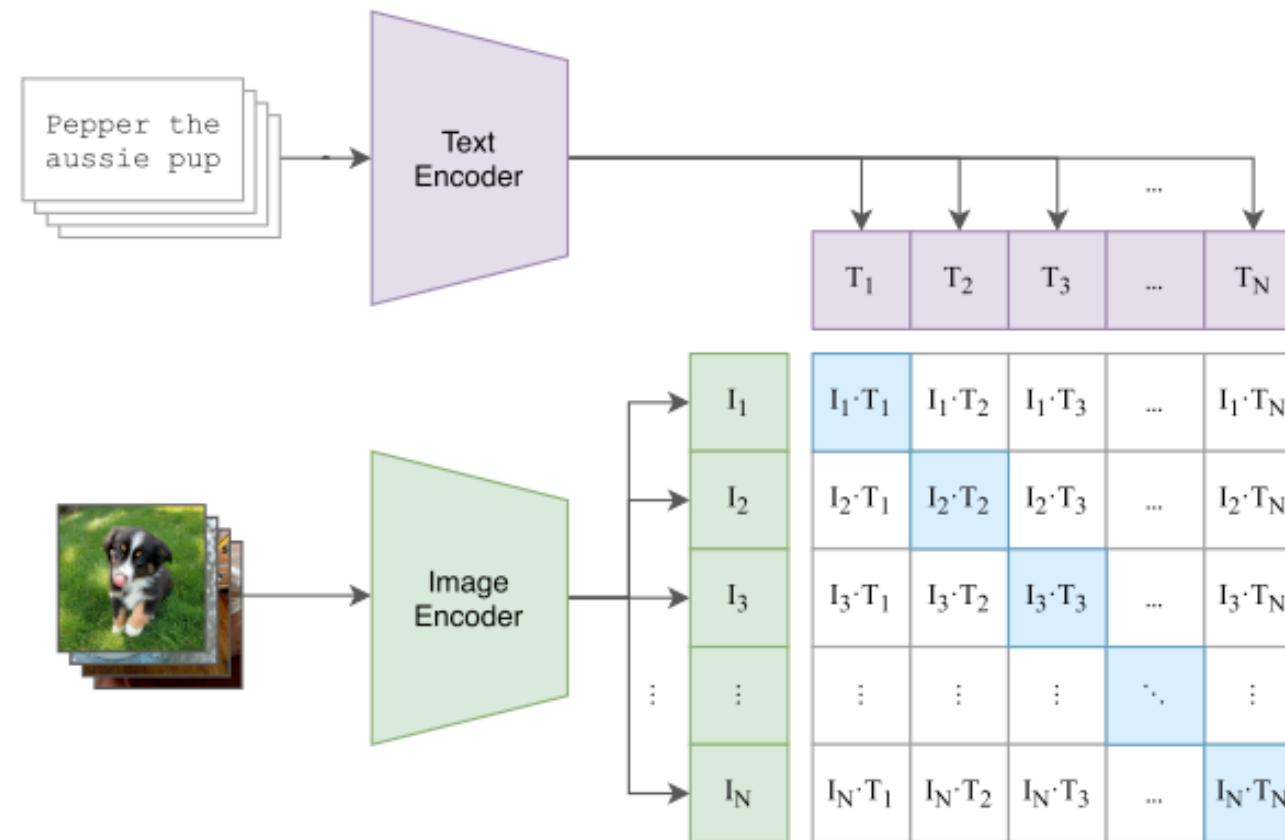
Vision-language models

extremely simple

we are gonna treat semantic as important, giving it a separate NN.

The label instead of being an element of a set is now a sentence

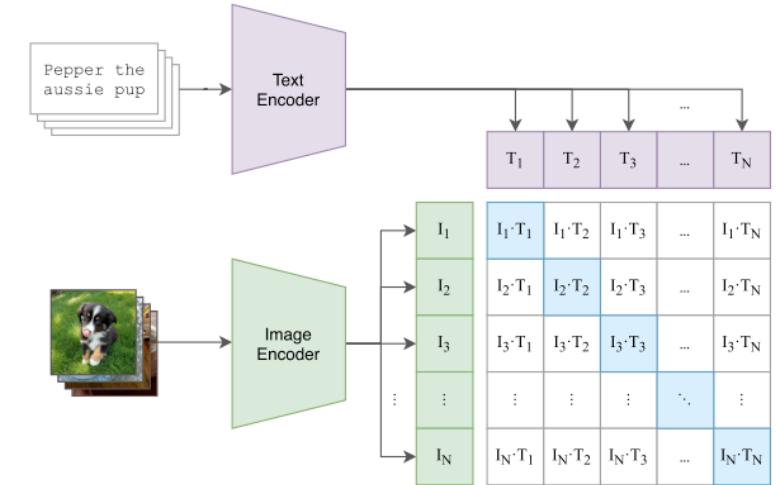
CLIP: Contrastive Language-Image Pre-training



The idea behind CLIP

semantic is more than just a label

Treat semantics beyond labels.



Two encoders: one for an image, one for a sentence describing it.

we are gonna use contrastive learning so similar samples are gonna be pushed togheeter, while different samples will push the different ones away from each other

Align both with a contrastive loss.

Pull image-text pair together, push other pairs in batch away.

Powerful approach, simple implementation

```
# image_encoder - ResNet or Vision Transformer
# text_encoder - CBOW or Text Transformer
# I[n, h, w, c] - minibatch of aligned images
# T[n, l] - minibatch of aligned texts
# W_i[d_i, d_e] - learned proj of image to embed
# W_t[d_t, d_e] - learned proj of text to embed
# t - learned temperature parameter

# extract feature representations of each modality
I_f = image_encoder(I) #[n, d_i]
T_f = text_encoder(T) #[n, d_t]

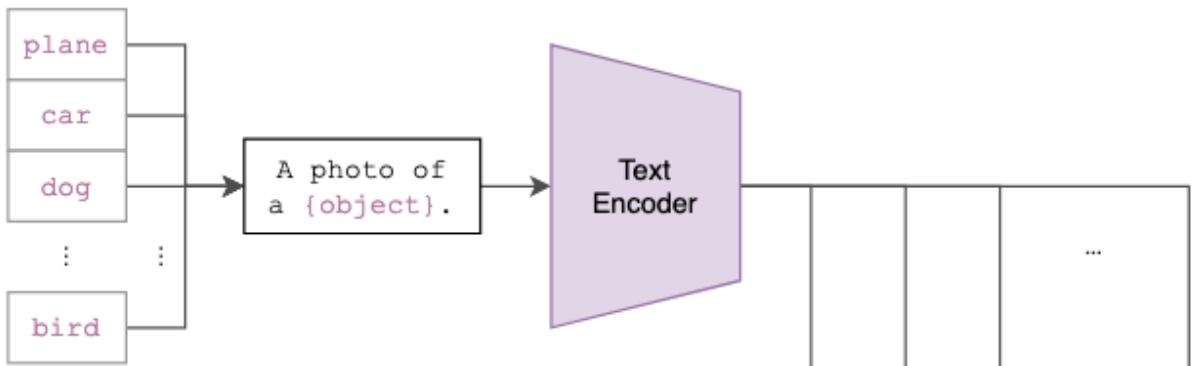
# joint multimodal embedding [n, d_e]
I_e = l2_normalize(np.dot(I_f, W_i), axis=1)
T_e = l2_normalize(np.dot(T_f, W_t), axis=1) what the l2 normalization does is
# scaled pairwise cosine similarities [n, n] that only the directions matters, it
logits = np.dot(I_e, T_e.T) * np.exp(t) projects them on the unit sphere

# symmetric loss function
labels = np.arange(n)
loss_i = cross_entropy_loss(logits, labels, axis=0)
loss_t = cross_entropy_loss(logits, labels, axis=1)
loss = (loss_i + loss_t)/2 t excacerbaet the means of the norm, (ask),
return very ora distribution
```

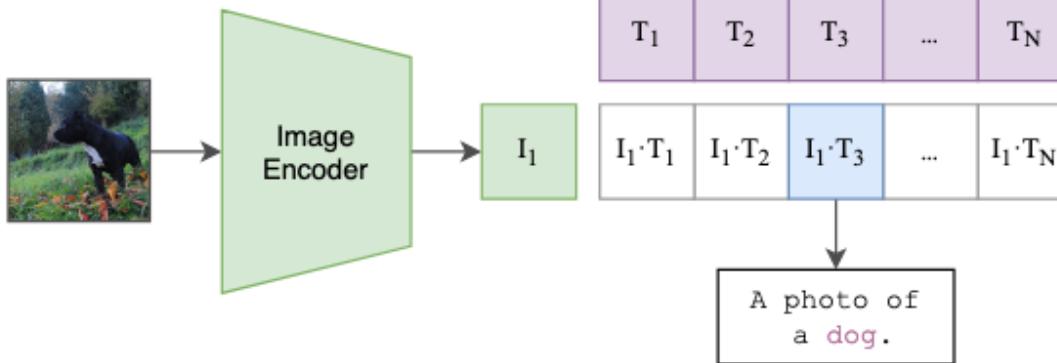
this works really well, the amount of data for which I can get a label now is much higher

How to use CLIP for “normal” classification?

(2) Create dataset classifier from label text



(3) Use for zero-shot prediction



just add at the beginning the string:

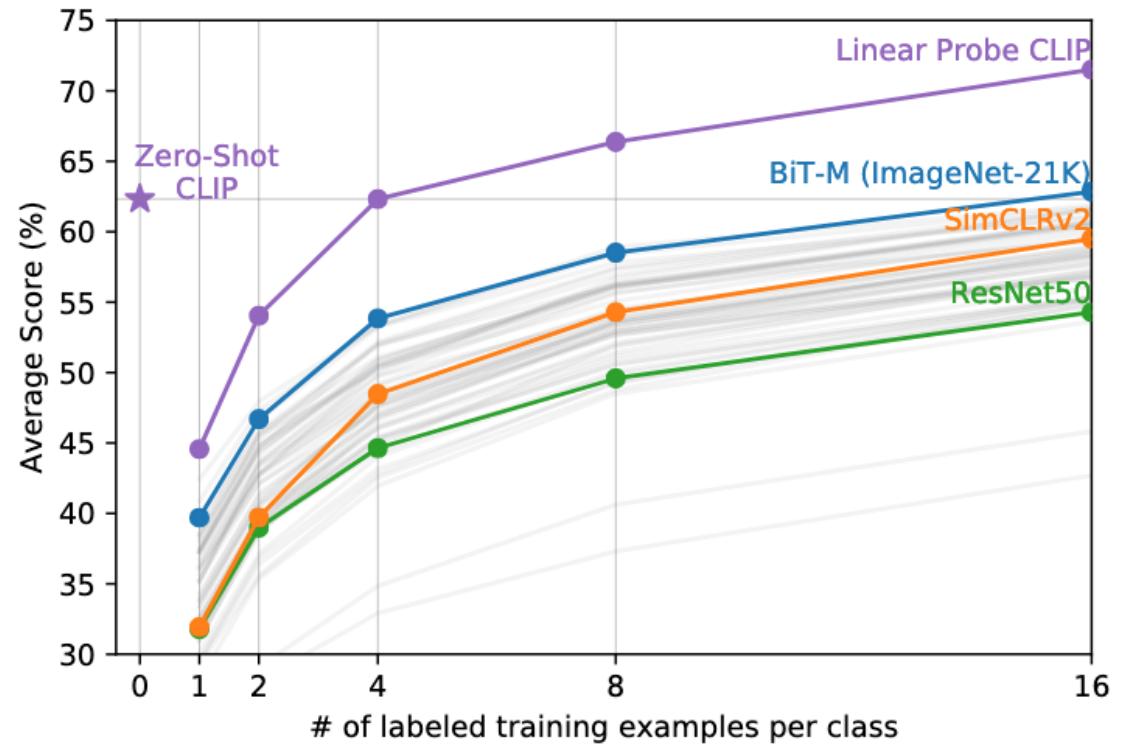
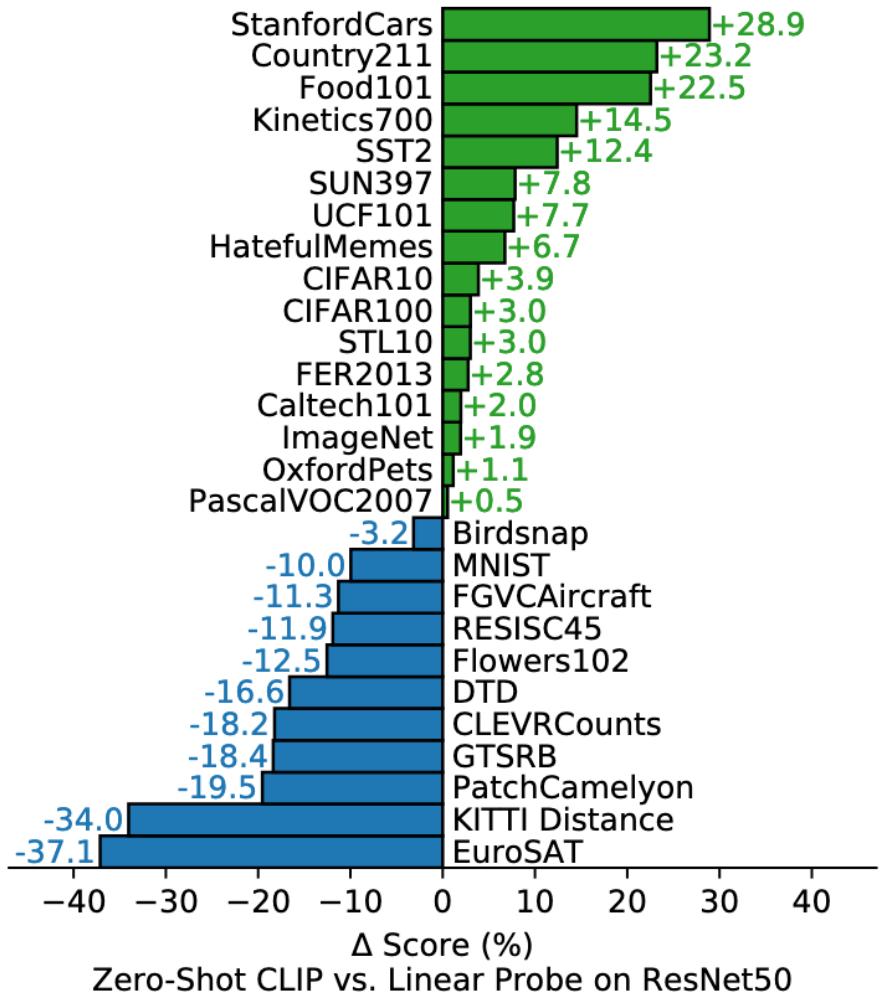
"A photo of a " {klass}

Could I use the predictions as before, obtain the predicted sentence and comput a similarity with the calss?

Maybe not because it would require to train a new encoder to embed the classes, I want to use a model already pre trained

Issues:
What about the length of the sequence?

Pre-training is so powerful, no training needed



Keys behind the success of CLIP

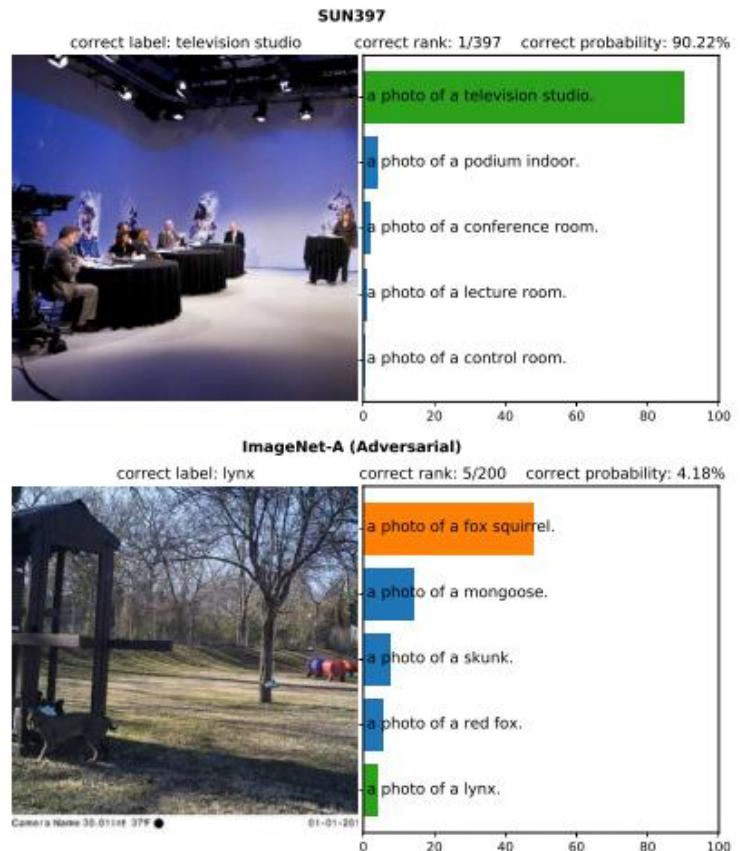
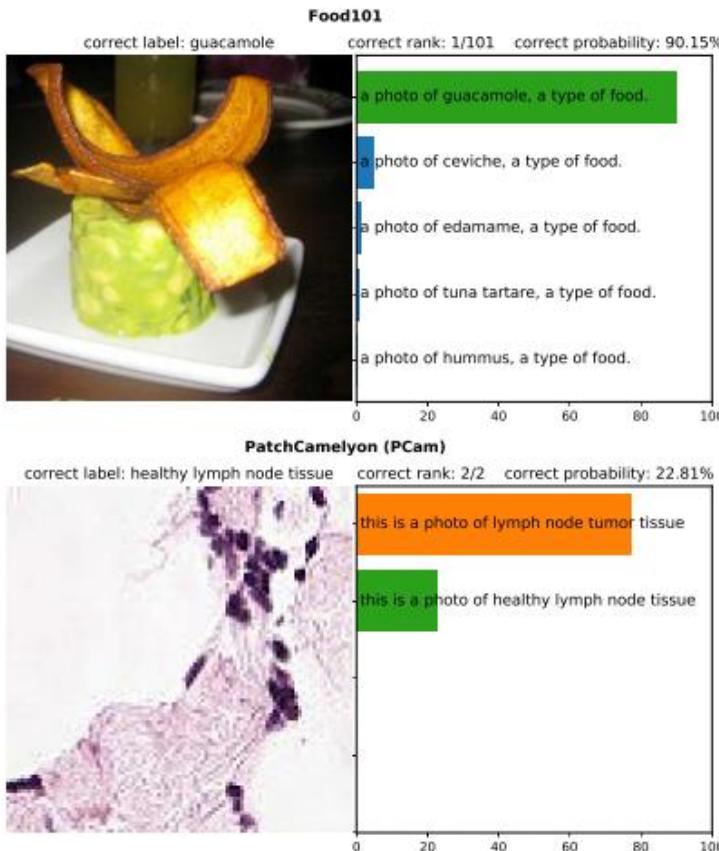
Pre-training on up to 5 billion image-text pairs (~100x bigger than ImageNet).

Sentences are more complex than labels.

Semantics is treated as continuous instead of discrete.

Zero-shot examples of CLIP

Predicted probability of top 5 classes for each example.



Robustness of CLIP

	Dataset Examples					ImageNet	Zero-Shot	ResNet101	CLIP	Δ Score
	ImageNet	ImageNetV2	ImageNet-R	ObjectNet	ImageNet Sketch					
ImageNet						76.2	76.2	0%		
ImageNetV2						64.3	70.1	+5.8%		
ImageNet-R						37.7	88.9	+51.2%		
ObjectNet						32.6	72.3	+39.7%		
ImageNet Sketch						25.2	60.2	+35.0%		
ImageNet-A						2.7	77.1	+74.4%		

Visualizing distribution shift for bananas, a class shared across 5 of the 7 natural distribution shift datasets.

		Food101	CIFAR10	CIFAR100	Birdsnap	SUN397	Cars	Aircraft	VOC2007	DTD	Caltech101	Pets	Flowers	MNIST	FER2013	STL10*	EuroSAT	REISISC45	GTSRB	KITTI	Country211	PCAM	UCF101	Kinetics700	CLEVR	HatefulMeme:	SSCF	ImageNet
LM RN50	LM RN50	81.3	82.8	61.7	44.2	69.6	74.9	44.9	85.5	71.5	82.8	85.5	91.1	96.6	60.1	95.3	93.4	84.0	73.8	70.2	19.0	82.9	76.4	51.9	51.2	65.2	76.8	65.2
CLIP-RN	50	86.4	88.7	70.3	56.4	73.3	78.3	49.1	87.1	76.4	88.2	89.6	96.1	98.3	64.2	96.6	95.2	87.5	82.4	70.2	25.3	82.7	81.6	57.2	53.6	65.7	72.6	73.3
	101	88.9	91.1	73.5	58.6	75.1	84.0	50.7	88.0	76.3	91.0	92.0	96.4	98.4	65.2	97.8	95.9	89.3	82.4	73.6	26.6	82.8	84.0	60.3	50.3	68.2	73.3	75.7
	50x4	91.3	90.5	73.0	65.7	77.0	85.9	57.3	88.4	79.5	91.5	92.5	97.8	98.5	68.1	97.8	96.4	89.7	85.5	59.4	30.3	83.0	85.7	62.6	52.5	68.0	76.6	78.2
	50x16	93.3	92.2	74.9	72.8	79.2	88.7	62.7	89.0	79.1	93.5	93.7	98.3	98.9	68.7	98.6	97.0	91.4	89.0	69.2	34.8	83.5	88.0	66.3	53.8	71.1	80.0	81.5
	50x64	94.8	94.1	78.6	77.2	81.1	90.5	67.7	88.9	82.0	94.5	95.4	98.9	98.9	71.3	99.1	97.1	92.8	90.2	69.2	40.7	83.7	89.5	69.1	55.0	75.0	81.2	83.6
CLIP-ViT	B/32	88.8	95.1	80.5	58.5	76.6	81.8	52.0	87.7	76.5	90.0	93.0	96.9	99.0	69.2	98.3	97.0	90.5	85.3	66.2	27.8	83.9	85.5	61.7	52.1	66.7	70.8	76.1
	B/16	92.8	96.2	83.1	67.8	78.4	86.7	59.5	89.2	79.2	93.1	94.7	98.1	99.0	69.5	99.0	97.1	92.7	86.6	67.8	33.3	83.5	88.4	66.1	57.1	70.3	75.5	80.2
	L/14	95.2	98.0	87.5	77.0	81.8	90.9	69.4	89.6	82.1	95.1	96.5	99.2	99.2	72.2	99.7	98.2	94.1	92.5	64.7	42.9	85.8	91.5	72.0	57.8	76.2	80.8	83.9
	L/14-336px	95.9	97.9	87.4	79.9	82.2	91.5	71.6	89.9	83.0	95.1	96.0	99.2	99.2	72.9	99.7	98.1	94.9	92.4	69.2	46.4	85.6	92.0	73.0	60.3	77.3	80.5	85.4

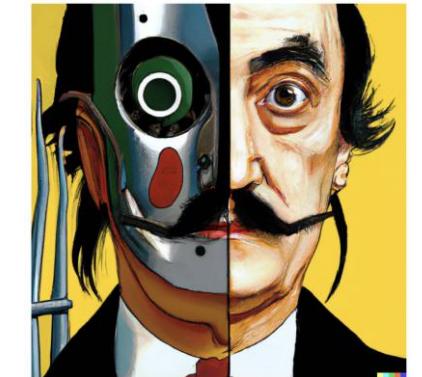
Performance of various pre-trained models over 27 datasets

Using CLIP in other models

The joint multimodal embedding space of CLIP enables its usage in many other downstream tasks in a zero-shot fashion.

- DALLE-2¹ – a text-guided image generation model,
- CLIP4Clip² - video-language retrieval model,
- GroupViT³ – semantic segmentation model - in a zero-shot manner.

“Vibrant portrait painting of Salvador Dalí with a robotic half.”



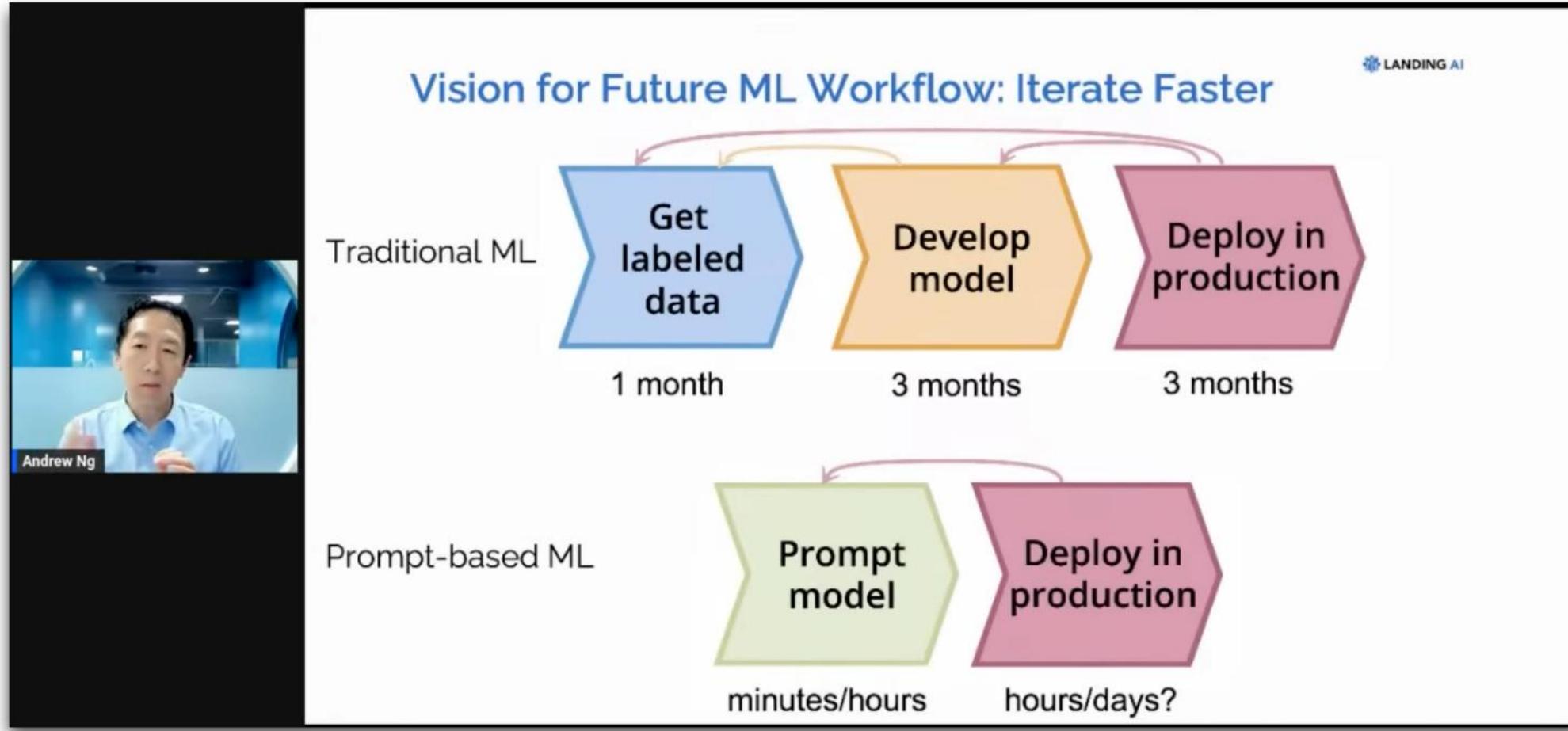
vibrant portrait painting of Salvador Dalí with a robotic half face

¹ Hierarchical Text-Conditional Image Generation with CLIP Latents, Ramesh et al. (2022)

² CLIP4Clip: An Empirical Study of CLIP for End to End Video Clip Retrieval , Luo et al. (2022)

³ GroupViT: Semantic Segmentation Emerges from Text Supervision, Xu et al. (2022)

A peek into the future



Limitations of CLIP

Models like CLIP simply provide a similarity score between text and image.

Lack the ability to generate language - less suitable to more open-ended tasks.

Require a fixed prompt mechanism to deal with standard classification.

Uses short context windows, ignore hierarchies.

Fine-tuning can actually decrease performance.

how do we formulate the sentence has a huge impact on the performances

Prompt engineering

a bad photo of a {}.
a photo of many {}.
a sculpture of a {}.
a photo of the hard to see {}.
a low resolution photo of the {}.
a rendering of a {}.
graffiti of a {}.
a bad photo of the {}.
a cropped photo of the {}.
a tattoo of a {}.
the embroidered {}.
a photo of a hard to see {}.
a bright photo of a {}.
a photo of a clean {}.
a photo of a dirty {}.
a dark photo of the {}.
a drawing of a {}.
a photo of my {}.
the plastic {}.
a photo of the cool {}.
a close-up photo of a {}.
a black and white photo of the {}.
a painting of the {}.
a painting of a {}.

a pixelated photo of the {}.
a sculpture of the {}.
a bright photo of the {}.
a cropped photo of a {}.
a plastic {}.
a photo of the dirty {}.
a jpeg corrupted photo of a {}.
a blurry photo of the {}.
a photo of the {}.
a good photo of the {}.
a rendering of the {}.
a {} in a video game.
a photo of one {}.
a doodle of a {}.
a close-up photo of the {}.
a photo of a {}.
the origami {}.
the {} in a video game.
a sketch of a {}.
a doodle of the {}.
a origami {}.
a low resolution photo of a {}.
the toy {}.
a rendition of the {}.

a photo of the clean {}.
a photo of a large {}.
a rendition of a {}.
a photo of a nice {}.
a photo of a weird {}.
a blurry photo of a {}.
a cartoon {}.
art of a {}.
a sketch of the {}.
a embroidered {}.
a pixelated photo of a {}.
itap of the {}.
a jpeg corrupted photo of the {}.
a good photo of a {}.
a plushie {}.
a photo of the nice {}.
a photo of the small {}.
a photo of the weird {}.
the cartoon {}.
art of the {}.
a drawing of the {}.
a photo of the large {}.
a black and white photo of a {}.
the plushie {}.

A slight change in wording could lead to big changes in performance

https://github.com/openai/CLIP/blob/main/notebooks/Prompt_Engineering_for_ImageNet.ipynb

The effect of prompt engineering

A way to solve the fact that different templates got different scores is to use an ensemble of templates and average the accuracies obtained

Dataset	Prompt	Accuracy
Caltech101	a [CLASS].	82.68
	a photo of [CLASS].	80.81
	a photo of a [CLASS].	86.29
	[V] ₁ [V] ₂ ... [V] _M [CLASS].	91.83
Flowers102	a photo of a [CLASS].	60.86
	a flower photo of a [CLASS].	65.81
	a photo of a [CLASS], a type of flower .	66.14
	[V] ₁ [V] ₂ ... [V] _M [CLASS].	94.51
Describable Textures (DTD)	a photo of a [CLASS].	39.83
	a photo of a [CLASS] texture .	40.25
	[CLASS] texture.	42.32
	[V] ₁ [V] ₂ ... [V] _M [CLASS].	63.58
EuroSAT	a photo of a [CLASS].	24.17
	a satellite photo of [CLASS].	37.46
	a centered satellite photo of [CLASS].	37.56
	[V] ₁ [V] ₂ ... [V] _M [CLASS].	83.53

A slight change in wording could lead to big changes in performance

What makes a good prompt?

A person riding a
motorcycle on a dirt road.



Two dogs play in the grass.



Can we learn to prompt instead?

Caltech101



Prompt	Accuracy
a [CLASS].	82.68
a photo of [CLASS].	80.81
a photo of a [CLASS].	86.29
[V] ₁ [V] ₂ ... [V] _M [CLASS].	91.83

(a)

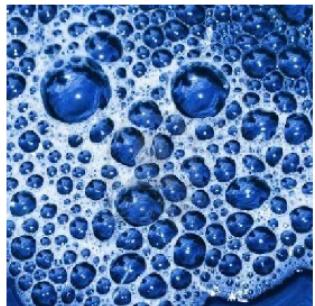
Flowers102



Prompt	Accuracy
a photo of a [CLASS].	60.86
a flower photo of a [CLASS].	65.81
a photo of a [CLASS], a type of flower.	66.14
[V] ₁ [V] ₂ ... [V] _M [CLASS].	94.51

(b)

Describable Textures (DTD)



Prompt	Accuracy
a photo of a [CLASS].	39.83
a photo of a [CLASS] texture.	40.25
[CLASS] texture.	42.32
[V] ₁ [V] ₂ ... [V] _M [CLASS].	63.58

(c)

EuroSAT

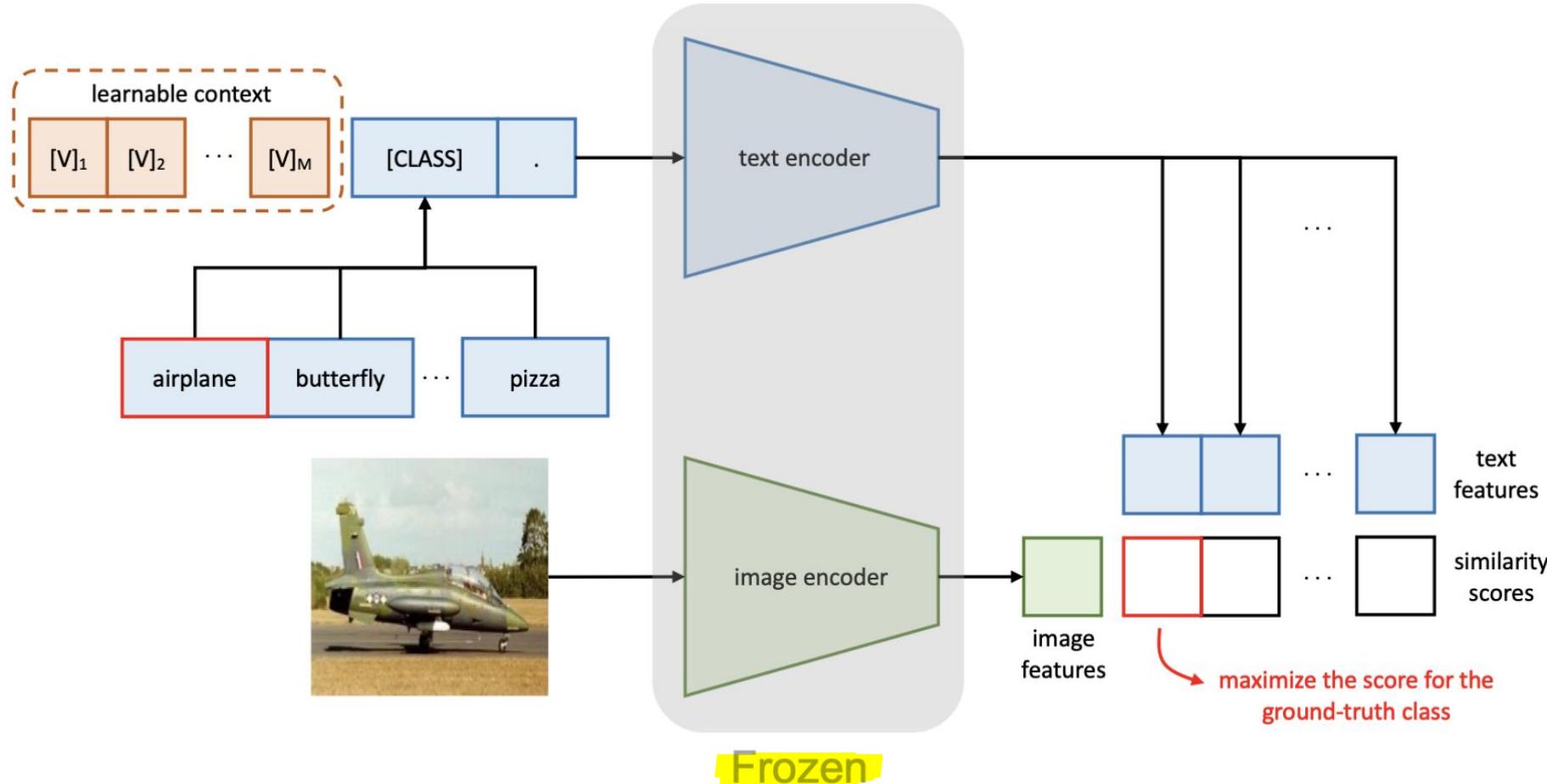


Prompt	Accuracy
a photo of a [CLASS].	24.17
a satellite photo of [CLASS].	37.46
a centered satellite photo of [CLASS].	37.56
[V] ₁ [V] ₂ ... [V] _M [CLASS].	83.53

(d)

Learning to prompt for vision-language models

the token are not anymore words, we are gonna let go semantic and maximize my performance on the task



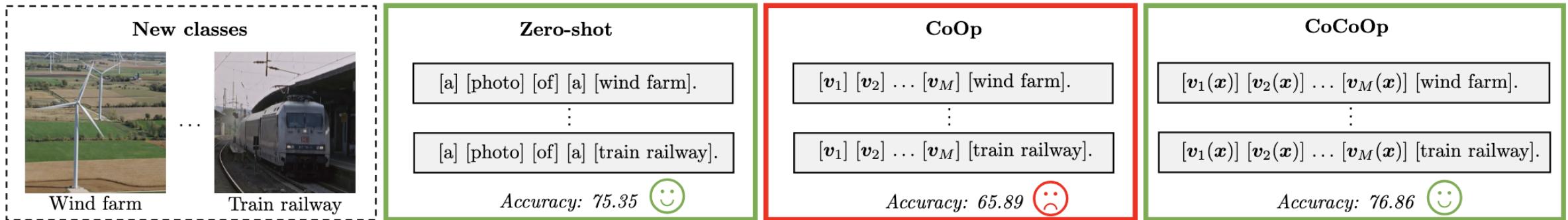
CoCoOp: Beyond statically learned prompts

overfits and does not generalize to wider unseen classes

Issue in CoOp: The prompt is overfit to wider unseen classes.

Why? The prompt embeddings ignore the visual instance.

it has a bias towards whatever it has seen most often before



the prompt embedding ignores the visual instance. All I'm gonna do here is.

each token will be also dependent on my image. The image embedding is not only used to compute the similarity logits but also to compute a meta token that will be combined with the tokens that I'm gonna learn

“Conditional Prompt Learning for Vision-Language Models.” Zhou et al. CVPR 2022.

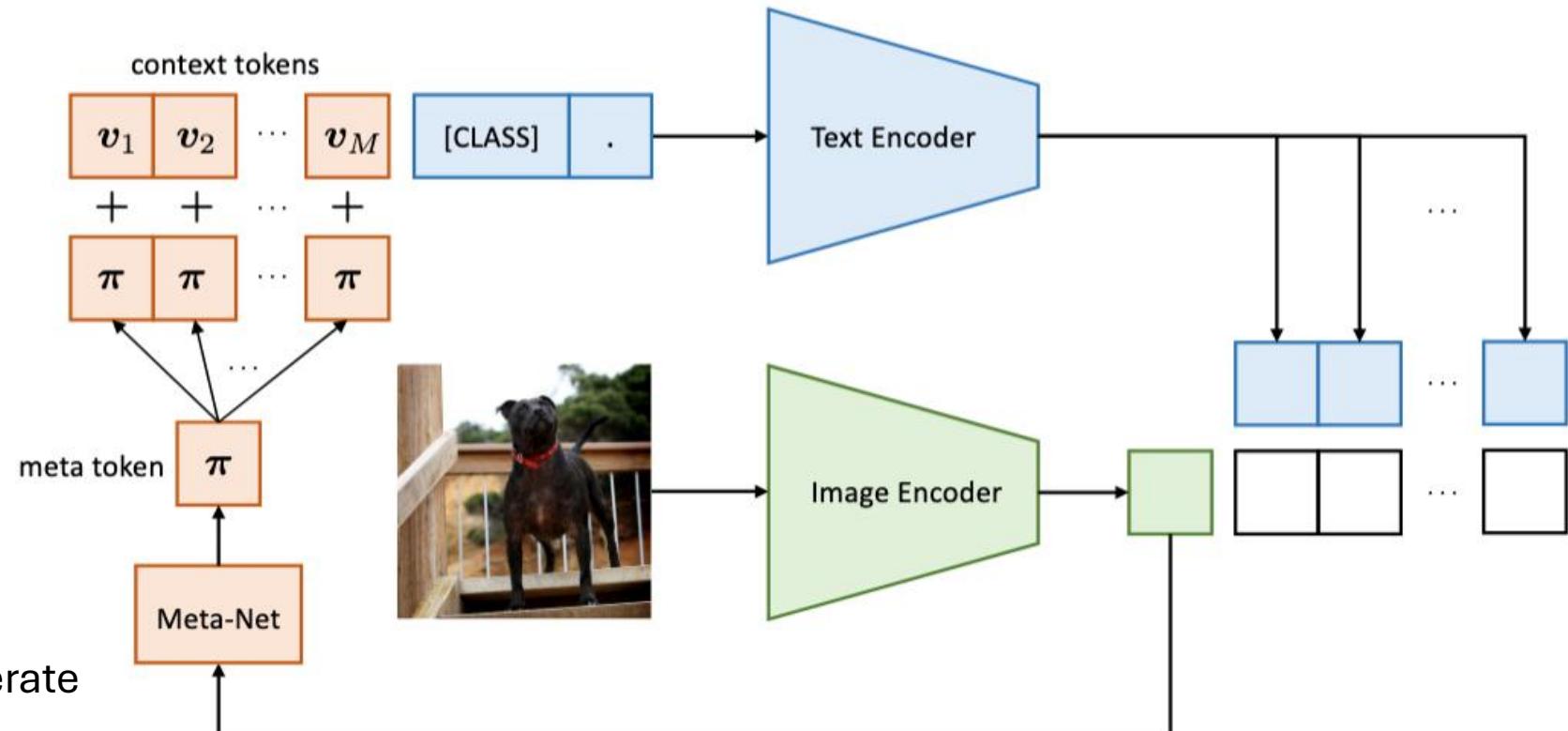
Conditional prompt learning

I'm going to bias the tokens with what I'm seeing in the image, hoping that this will debias the scores

I have no more any interpretability, this are just random tokens

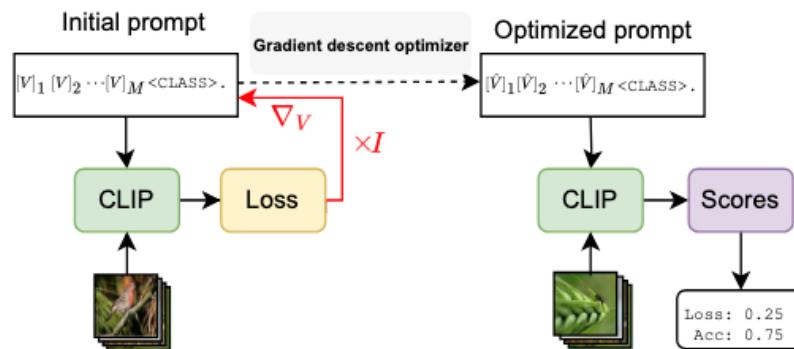
Component 1: Context vectors (ala CoOp).

Component 2:
A network that learns to generate
language-conditional token.

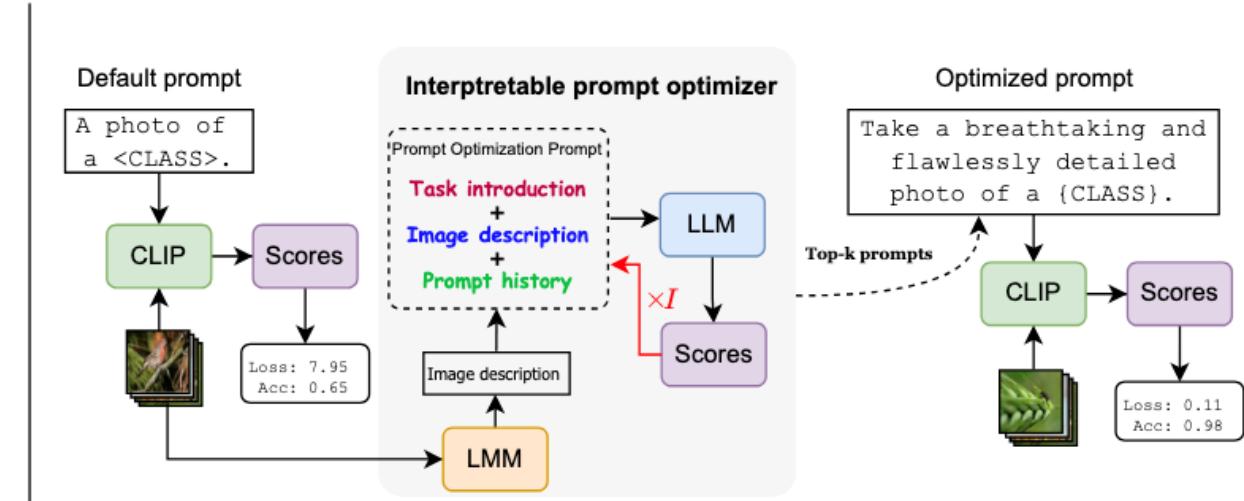


Interpretable prompt learning

Issue with all prompt learners: we have no clue what they learned.



(a) Gradient-based prompt optimization.



(b) Interpretable prompt optimization.

in context learning, the LLm works to refine the sentences in order to improve the scores, at each iteration wed compute new scores from new sentences and in the end the new sentences really improve the original scvores

“IPO: Interpretable Prompt Optimization for Vision-Language Models.” Du et al. NeurIPS 2024.

Summarizing prompts

Prompts make it possible to have in-context generalization.

First attempt: manually curate a prompt.

Prompt learning optimizes prompt tokens, massively improving results.

Interpretable prompt learning makes prompt learning understandable, but require LLM probes (which can become quite expensive).

Break

Looking beyond CLIP

In CLIP, the text encoder is learnt from scratch, why not start from an LLM?

How to efficiently fine-tune VLMs?

What about other modalities?

CLIP is flat, but the real-world is hierarchical. How can we fix this?

Using an LLM: Flamingo

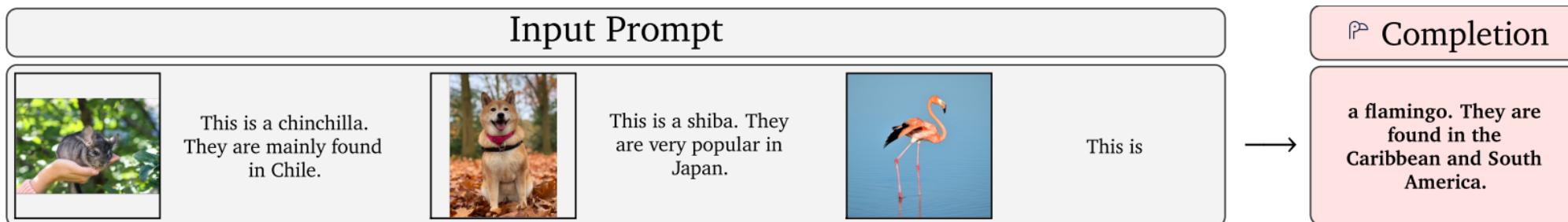
Flamingo is a Transformer-based architecture for multimodal few-shot tasks (image captioning, visual dialogue or visual question answering).

Able to learn from only a few input/output examples i.e., *in few-shot settings*.

It processes arbitrarily interleaved images and text as prompt;

And it generates output text in an open-ended manner.

Performs in-context learning (like GPT) but with images and text as context.



Pre-training Flamingo

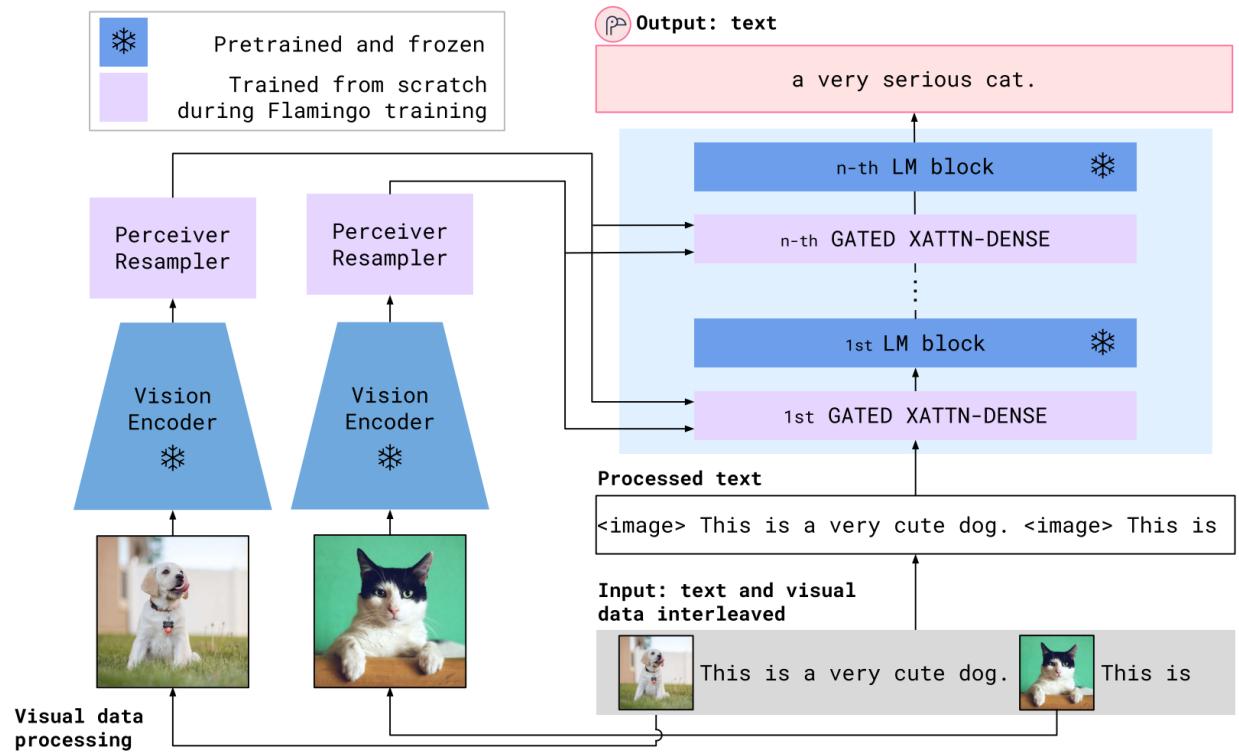
Vision side: an encoder with contrastive text-image approach, à la CLIP.

Language side: existing autoregressive LM trained on a large text corpus.

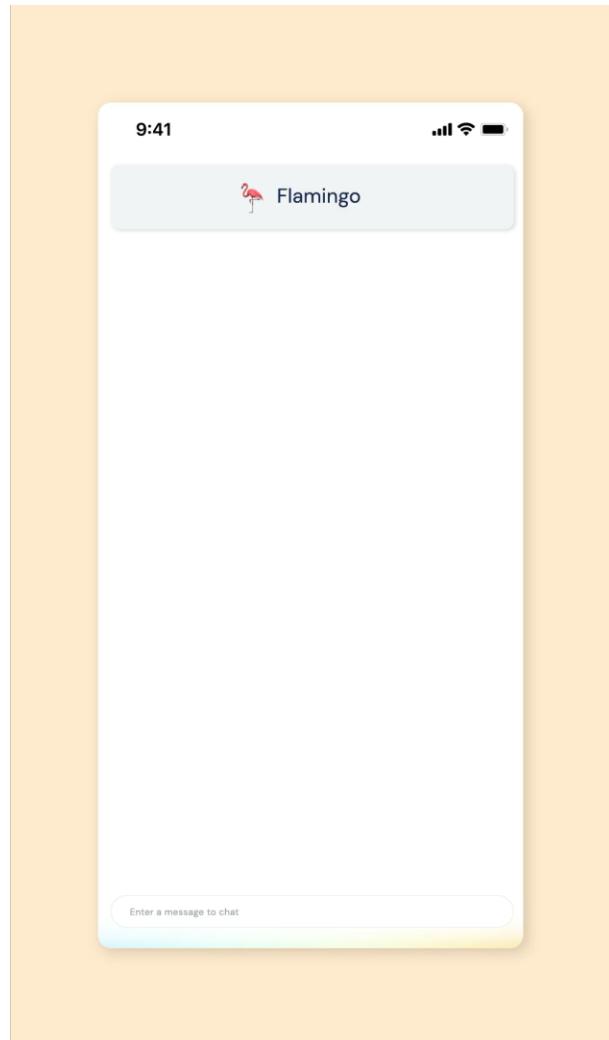
Linked via a learnable attention component (the Perceiver).

It outputs a fixed-size set of visual tokens.

Which are used to condition the frozen LM, trained to generate text.



Example output of Flamingo



Which city is this?

This is a street sign in New York City.

Which street corner is it?

It is the intersection of Madison Avenue and East 42nd Street.

Which subway station is nearby?

The closest subway station is Grand Central Station.

I would like you to read the color and then say the color it is written in. For example:

PURPLE

Color is "Purple" and it is written in red.

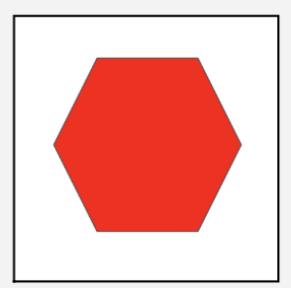
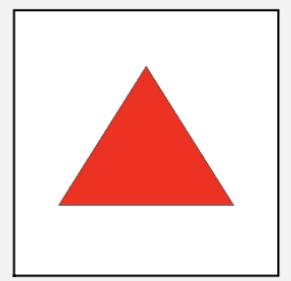
GREEN

Color is "Green" and it is written in blue.

YELLOW

Color is "Yellow" and it is written in green.

BLACK



What is the difference between these two images?

The first one is a triangle and the second one is a hexagon.

Which one has larger internal angles?

The hexagon has larger internal angles.

Looking beyond CLIP

In CLIP, the text encoder is learnt from scratch, why not start from an LLM?

How to efficiently fine-tune VLMs?

What about other modalities?

CLIP is flat, but the real-world is hierarchical. How can we fix this?

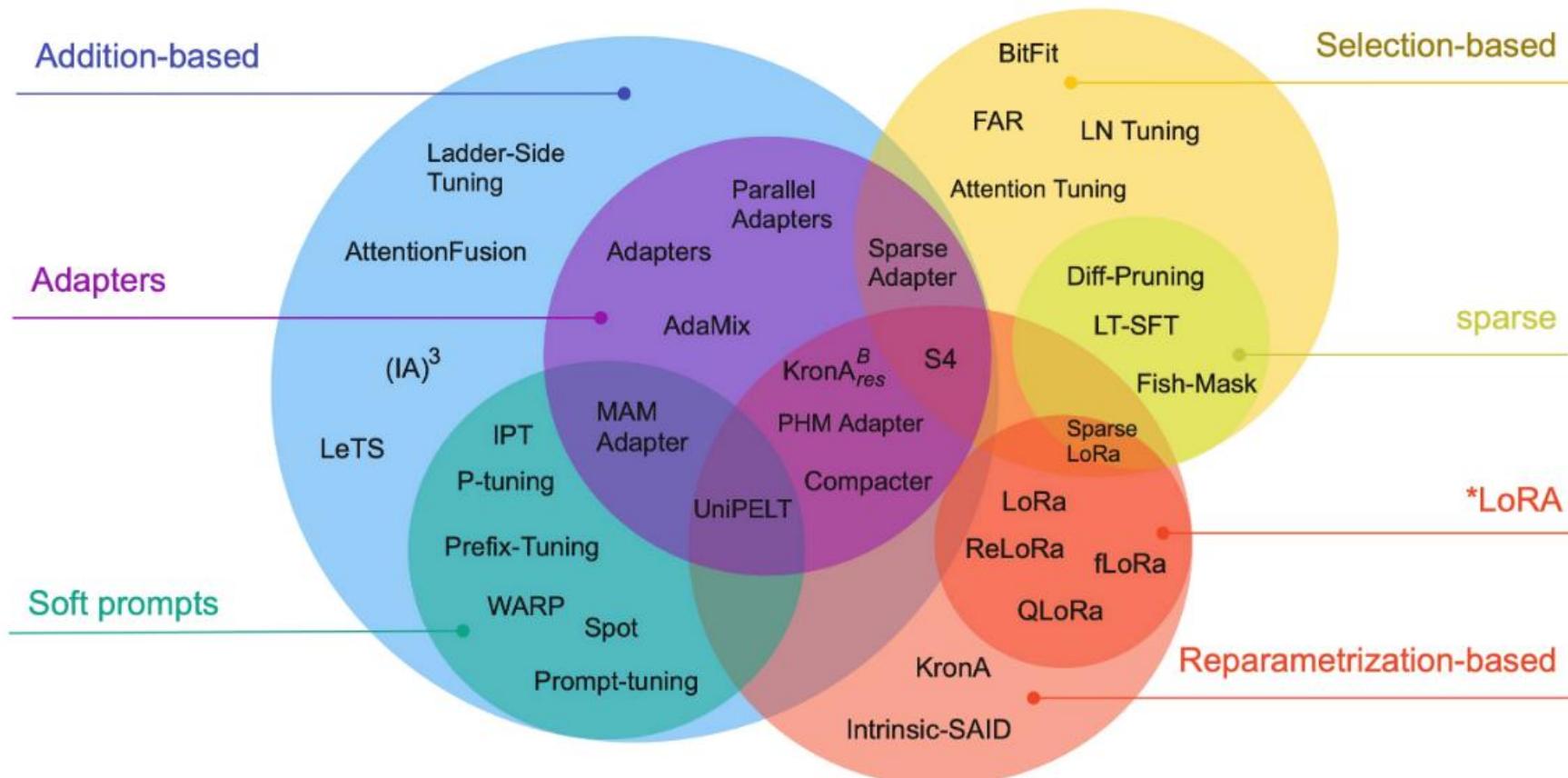
Fine-tuning VLMs

Sometimes, in-context learning is not enough, we need to alter the parameters.

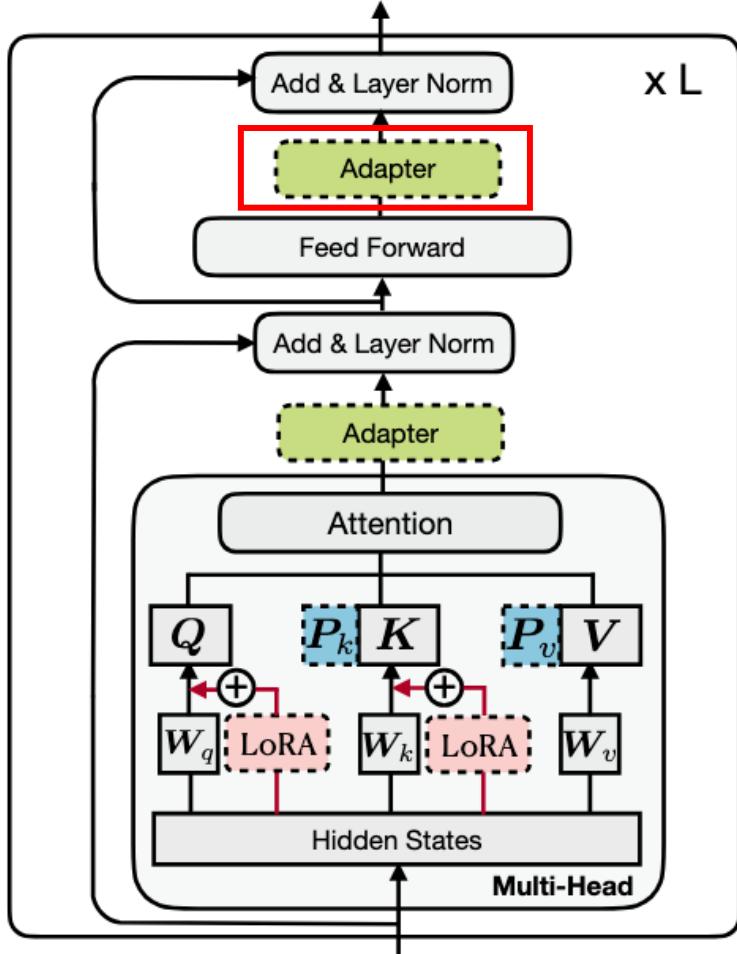
E.g., when exposed to a new task, new language, or any other new scope.

Problem! LLMs have huge networks. Us poor common folk cannot simply update a 10B parameter network.

Parameter efficient fine-tuning



Adding adapters

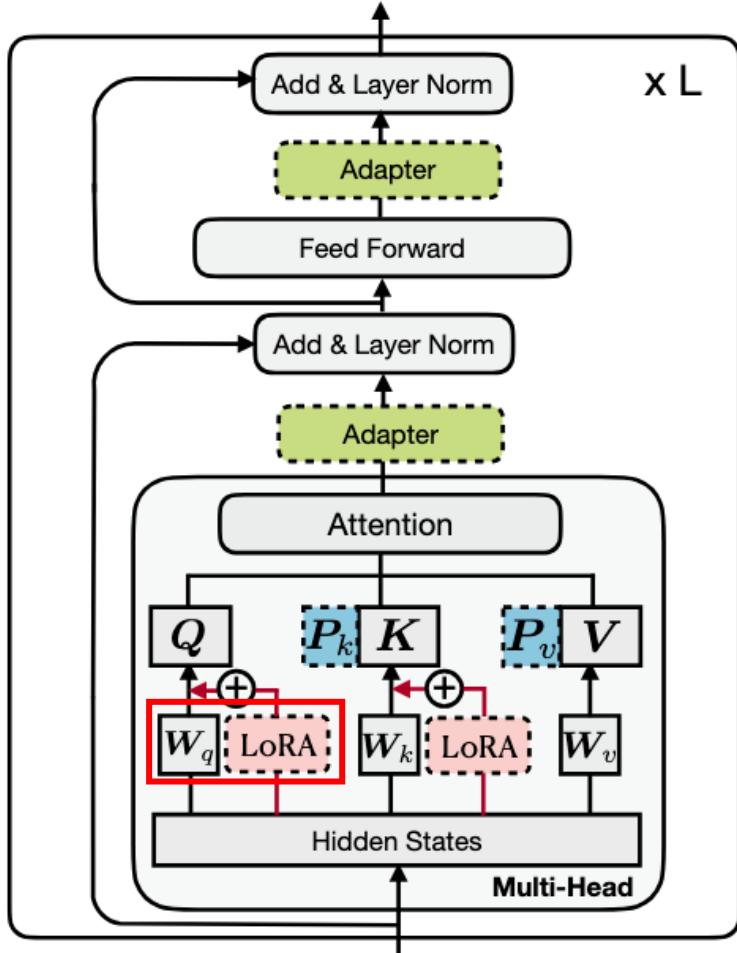


Let's add some small network blocks.

During fine-tuning, we keep the main architecture fixed and only tune the adapters.

Strong performance while only needing a full computation graph of a small subset.

Adding LoRA

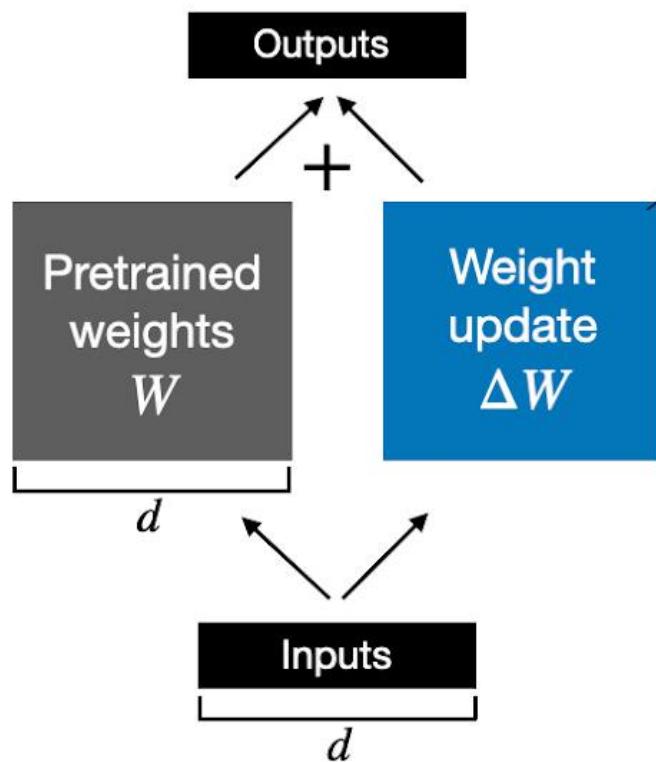


Most of the parameters of transformers are in the linear layers. Maybe these such matrices has a lower intrinsic dimensionality?

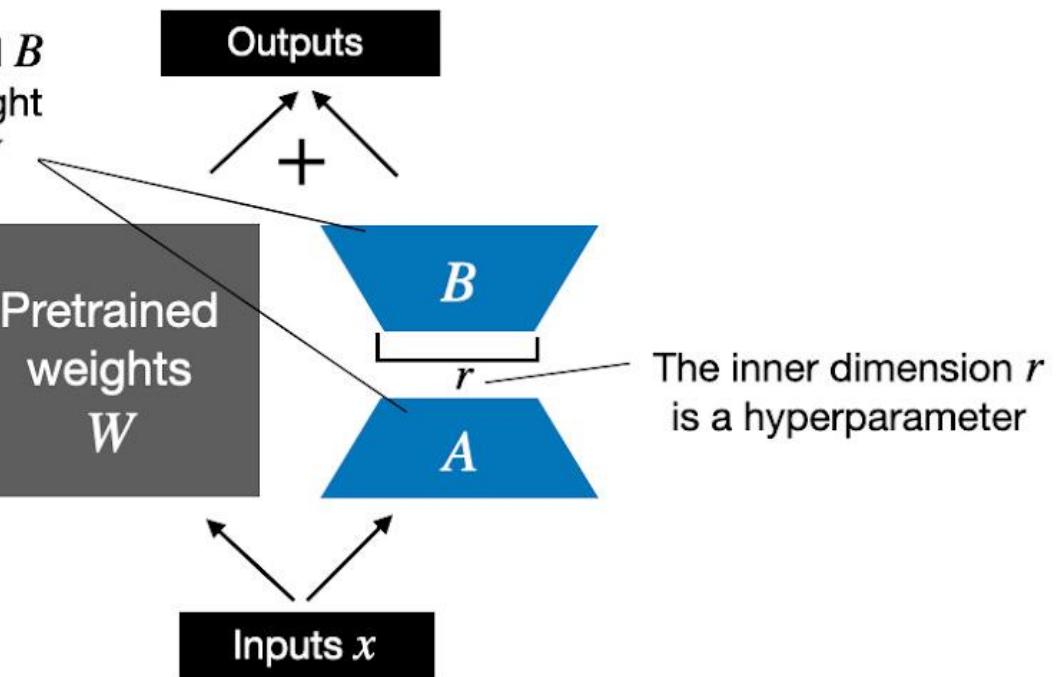
Can we learn a low-rank approximation of these huge matrices during fine-tuning?

LoRA

Weight update in regular finetuning



LoRA matrices A and B approximate the weight update matrix ΔW



The inner dimension r is a hyperparameter

Looking beyond CLIP

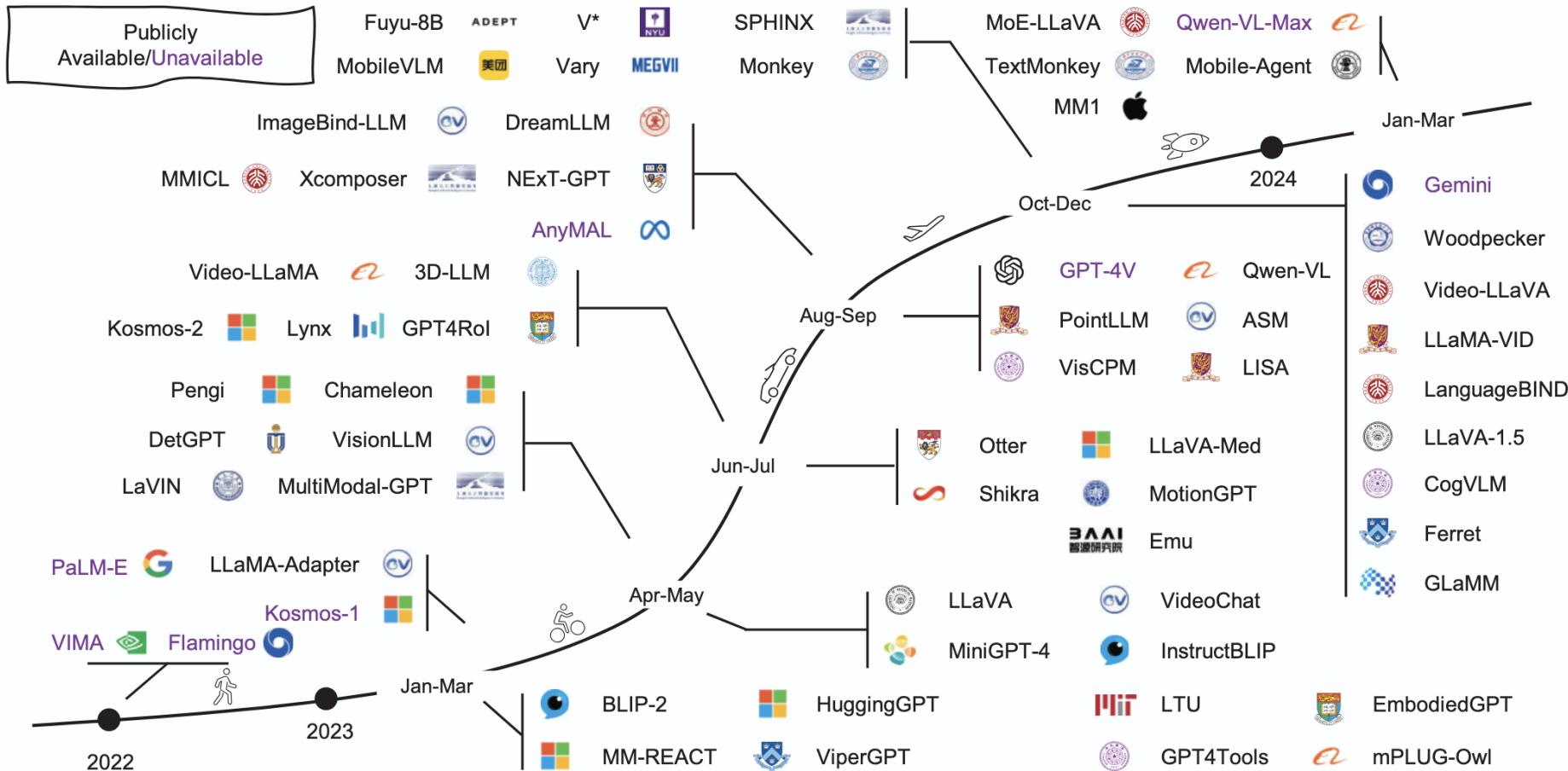
In CLIP, the text encoder is learnt from scratch, why not start from an LLM?

How to efficiently fine-tune VLMs?

What about other modalities?

CLIP is flat, but the real-world is hierarchical. How can we fix this?

Multimodal Large Language Models



“A survey on multimodal large language models.” Yin et al. NSR 2024.

Main idea of MLLMs

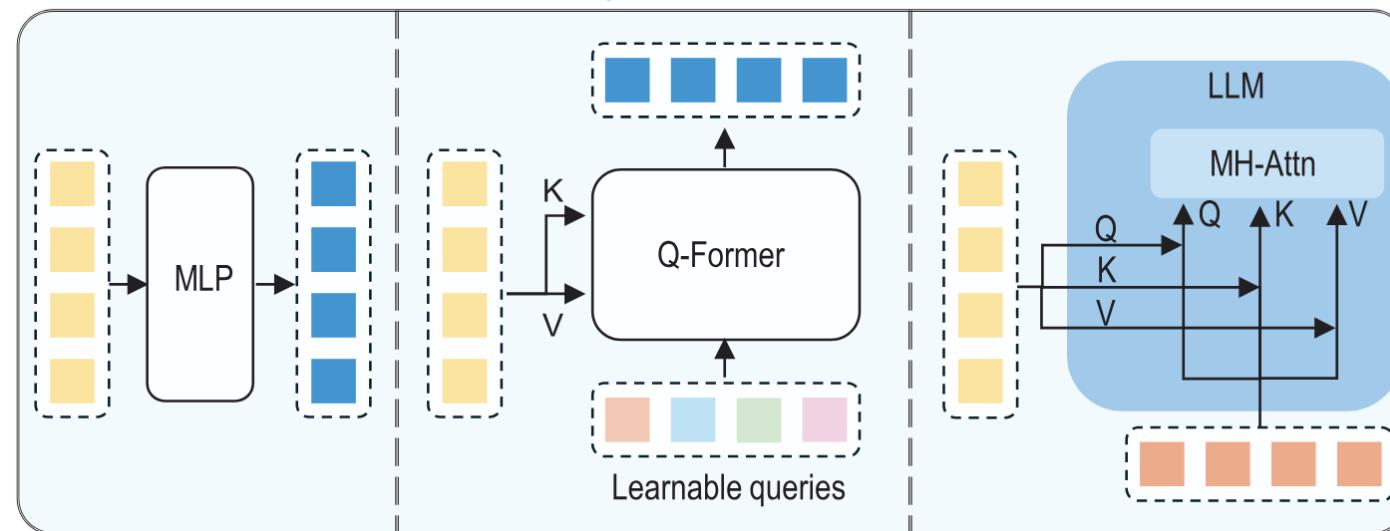
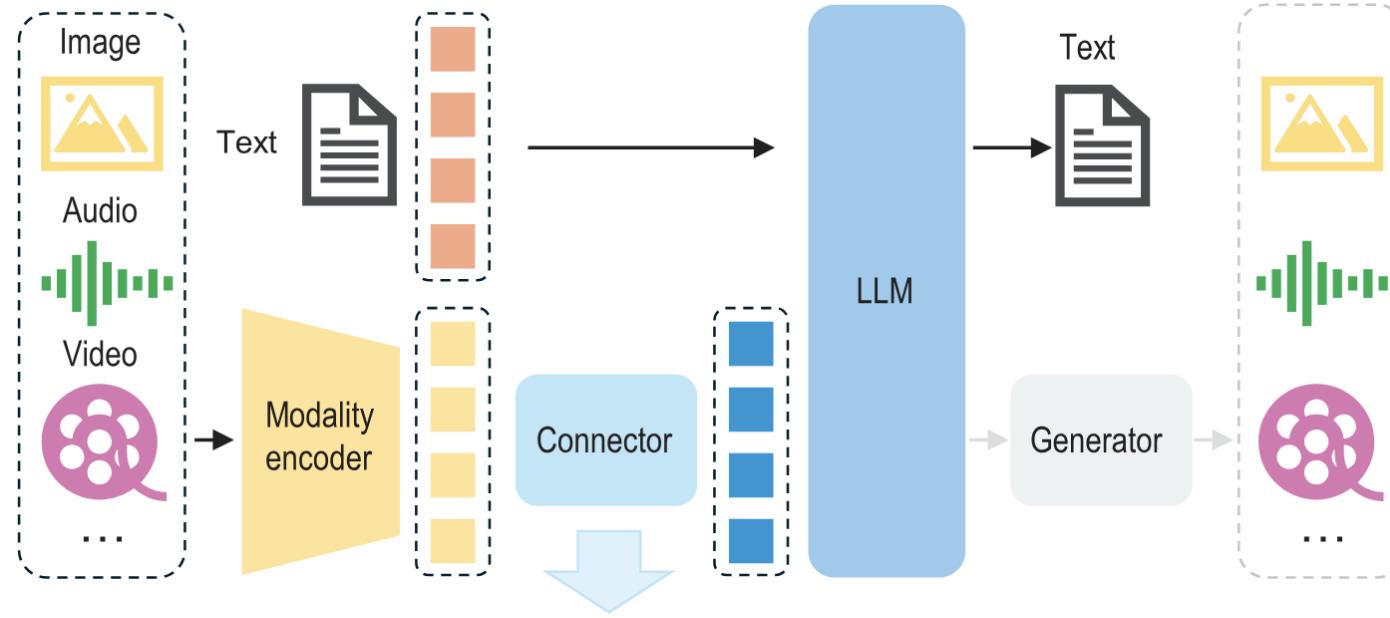
More is more: different modalities provide complimentary views.

For example, visually a bike is close to a motor and far from a leafblower.

In audio however, the motor and leafblower are close, far from a bike.

You can soon expect extensions of popular LLMs to video, audio, and more.

Multimodal Large Language Model framework



Case study 1: MLLMs and video popularity

Video-based MLLMs can deal with concrete tasks, such as object recognition.

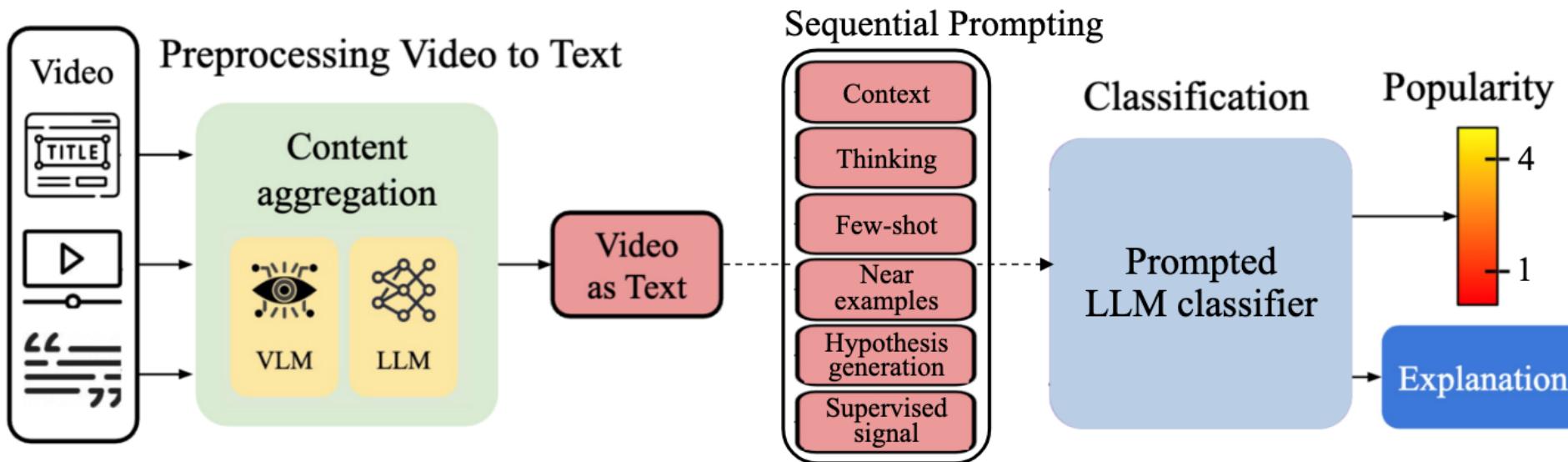
Video popularity is however cultural, social, and time-dependent.

Is it possible for an MLLM to predict whether a video will be popular, without fine-tuning the modal to this task?

“Large Language Models Are Natural Video Popularity Predictors.” Kayal et al. ACL 2025.

Case study 1: MLLMs and video popularity

Main idea: bring all the multi-modal inputs to one modality, namely text.



Case study 1: MLLMs and video popularity

Outcome: prompting an MLLM works better than training a modal specifically to solve the task at hand, with explainability as a bonus.

Video content



Title: Mexico vs. Brazil Highlights | International Friendly

Text summary

The video starts with Brazil attacking. Andreas Pereira scores a goal for Brazil after Mexico's Edson Alvarez dives in

retrieved videos



LLM hypothesis

1. highlight intense matches, finals, or close games
2. feature well-known or popular teams/players

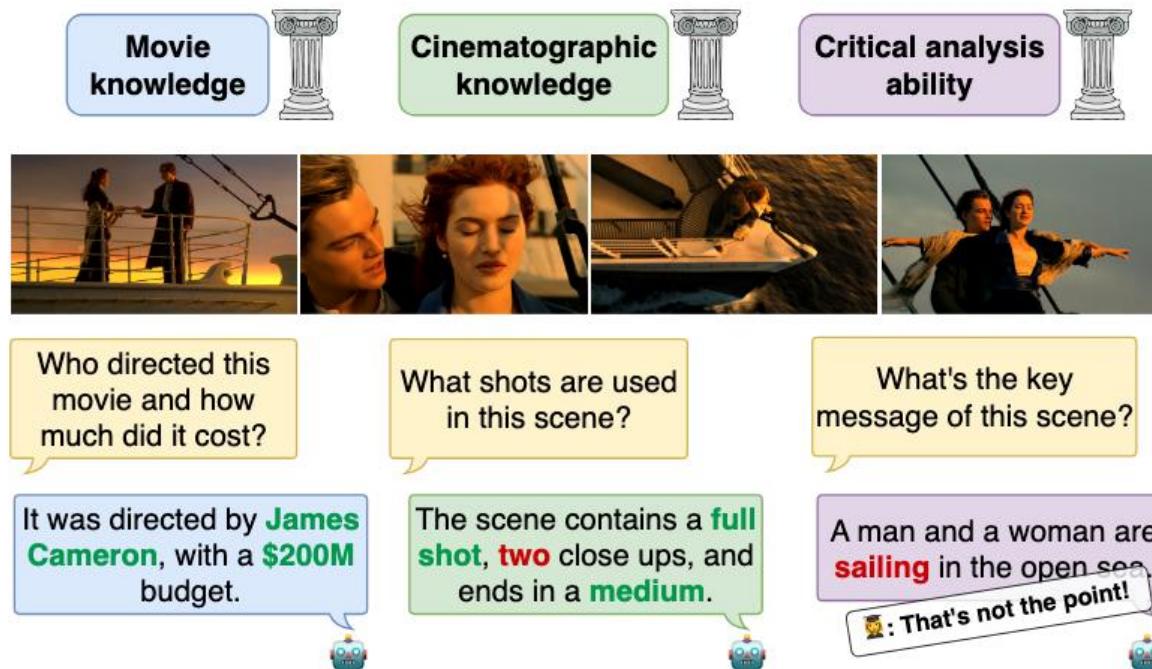
Prediction : 4 / 4

LLM explanation

1. The match is between Brazil and Mexico, Brazil being one of the biggest national teams in the world.
2. The video features several star players including Vinicius and Richarlison.

Case study 2: MLLMs and movies

Are MLLMs Movie Buffs? No.



“Are Multimodal LMMs Movie Buffs.” Brett et al. under review.

Case study 2: MLLMs and movies

MLLMs are decent at predicting meta-categories and basic cinematographics.



GT: Action, Adventure, Sci-Fi
Pred: Action, Adventure, Sci-Fi



GT: 100M+ USD
Pred: 10-50M USD



GT: Extreme close-up
Pred: Extreme close-up



GT: Close-up
Pred: Extreme close-up

But did they get this from viewing the visuals, or because the LLM scanned IMDb?

Case study 2: MLLMs and movies

We can ask the MLLM to act as a film studies student. Then we can give its output to a film studies teacher to evaluate.



[...] The scene begins with a close-up of a woman lying on an operating table, wearing a hospital gown and an oxygen mask. The camera then pans out to reveal a group of surgeons dressed in surgical gowns, masks, and caps, preparing for the operation. The lighting is dim, with a blue hue dominating the scene, creating a clinical and sterile atmosphere. [...]



Incorrect, the first scene is a full shot of the whole group.

Accurate comment on the lighting!

No comments on the fact that she's flying and the surgeons turn into aliens?



The use of shadows and low lighting creates an atmosphere of suspense [...] The camera angles are carefully chosen to emphasize the power dynamics at play, with the man in the suit often positioned higher than the police officer. [...] The use of slow motion during key moments adds to the dramatic effect [...]



Comments on lighting and tone are accurate.

Good assessment of the power dynamics based on positioning!

There is no slow motion.



Conclusion: MLLMs don't really use the visual modality and hallucinate everything.

Looking beyond CLIP

In CLIP, the text encoder is learnt from scratch, why not start from an LLM?

How to efficiently fine-tune VLMs?

What about other modalities?

CLIP is flat, but the real-world is hierarchical. How can we fix this?

Next lecture

Lecture	Title	Lecture	Title
1	Intro and history of deep learning	2	AutoDiff
3	Deep learning optimization I	4	Deep learning optimization II
5	Convolutional deep learning	6	Attention-based deep learning
7	Graph deep learning	8	From supervised to unsupervised deep learning
9	Multi-modal deep learning	10	Generative deep learning
11	What doesn't work in deep learning	12	Non-Euclidean deep learning
13	Q&A	14	Deep learning for videos

Learning and reflection

Multi-modal deep learning not covered in Understanding Deep Learning book.

Thank you



Deep Learning 1

2025-2026 – Pascal Mettes

Lecture 10

Generative deep learning

Previous lecture

Lecture	Title	Lecture	Title
1	Intro and history of deep learning	2	AutoDiff
3	Deep learning optimization I	4	Deep learning optimization II
5	Convolutional deep learning	6	Attention-based deep learning
7	Graph deep learning	8	From supervised to unsupervised deep learning
9	Multi-modal deep learning	10	Generative deep learning
11	What doesn't work in deep learning	12	Non-Euclidean deep learning
13	Q&A	14	Deep learning for videos

This lecture

Generative learning 1: The variational era

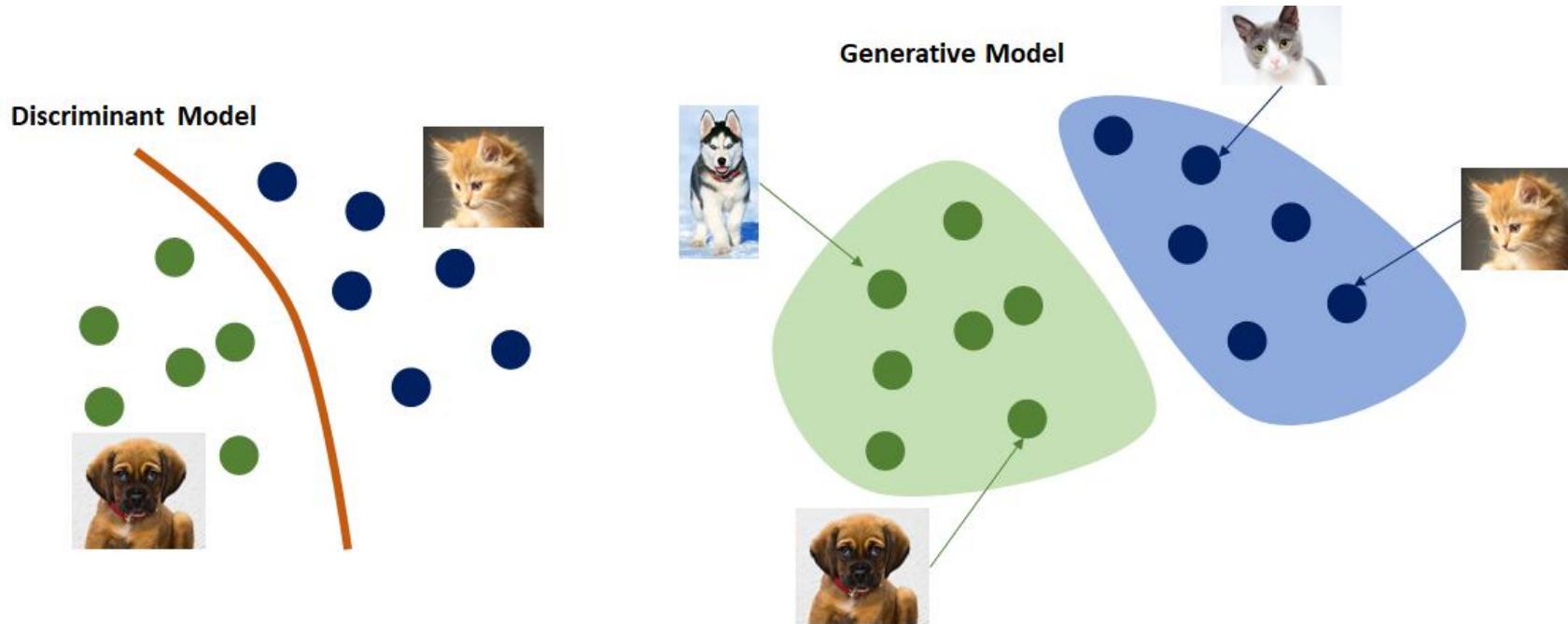
Generative learning 2: The adversarial era

Generative learning 3: The diffusion era

Generative versus discriminative learning

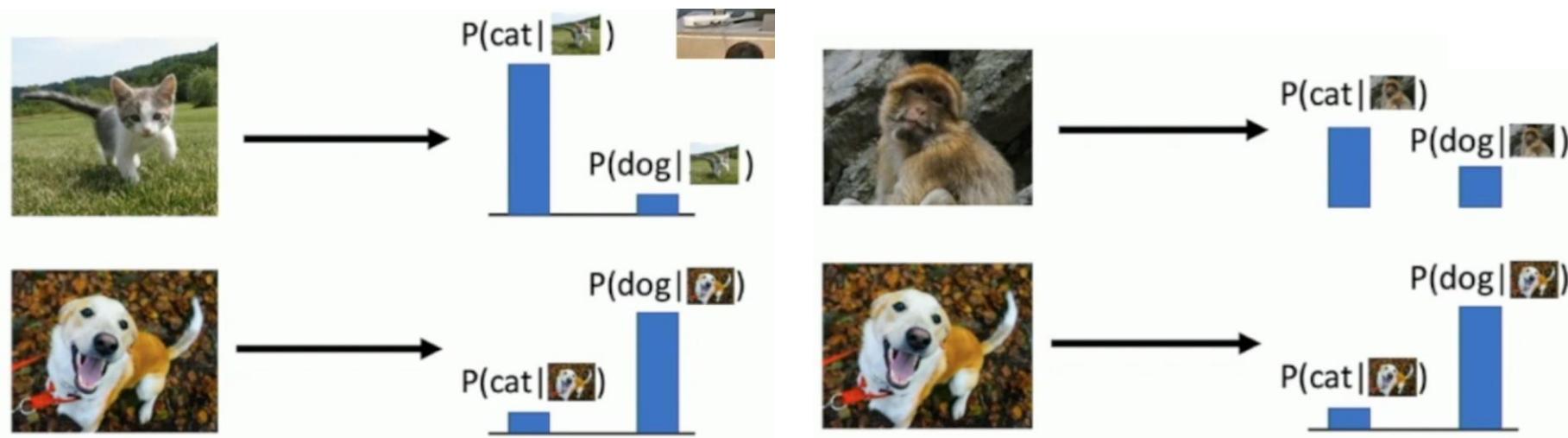
usually we want to do LEFT, now we are gonna see how to do right

$$p(y|x) \text{ vs } p(x)$$

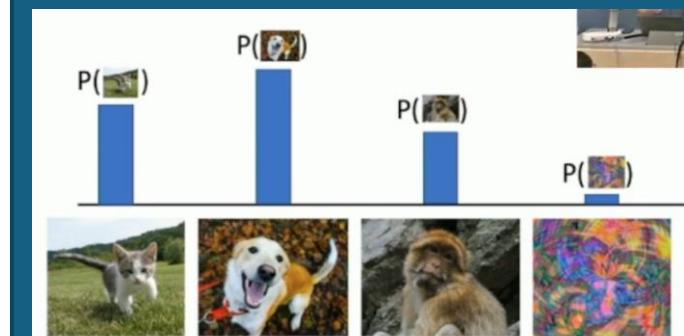


The importance of generative learning

Discriminative model: p is normalized for *outputs*, but not for inputs:



Generative model:
 p is normalized for inputs



The importance of generative learning

Learn the distribution of data itself.

Physics: Model its laws, predict planet motions, etc.

Economics: Forecast financial patterns.

Idem for math, biology, geology, ...

Make data and models more interpretable.

Generate new samples.

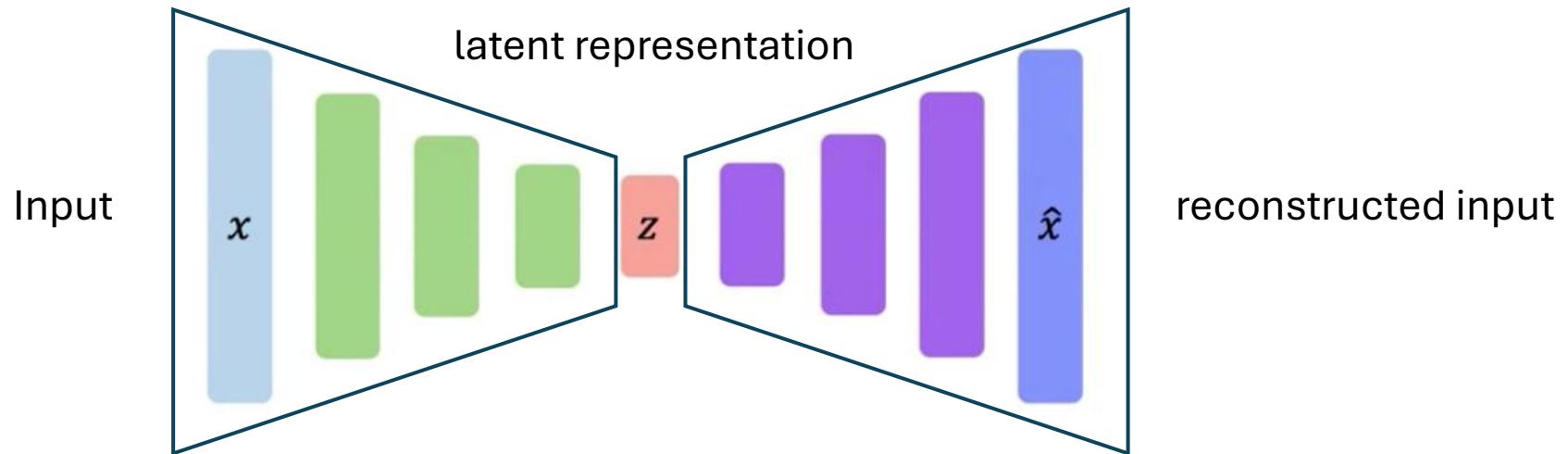
Enhance discriminative models.

Generative learning 1: The variational era

The autoencoder

The autoencoder is a feedforward network with a bottleneck layer.

Optimization is easy: minimize error between input and reconstructed output.



it is like an identity function but since I'm going through a bottleneck it cannot learn exactly the input, this bottleneck force it to learn the essential elements of the data because it cannot memorize everything

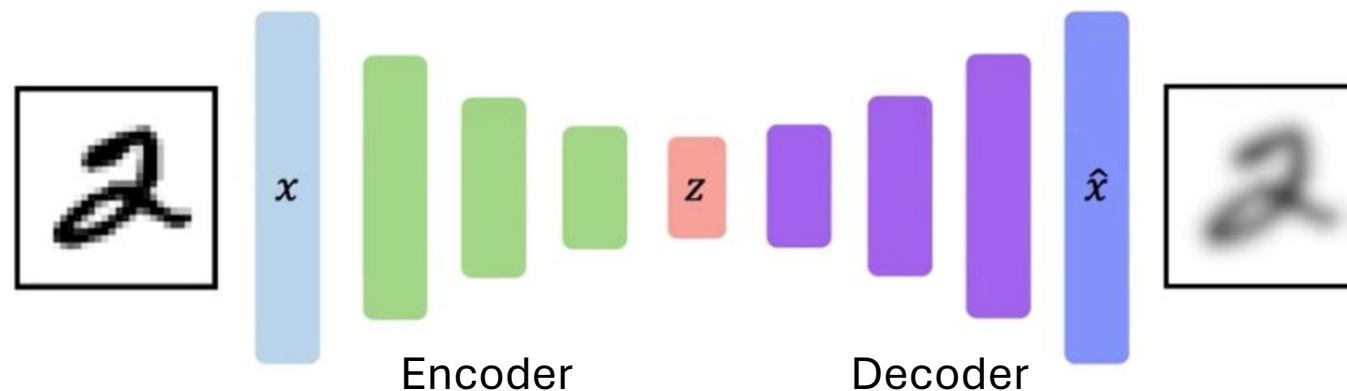
Structure of the autoencoder

Two parts: an encoder and a decoder.

Encoder: Map from input to bottleneck.

Decoder: Map from bottleneck to output.

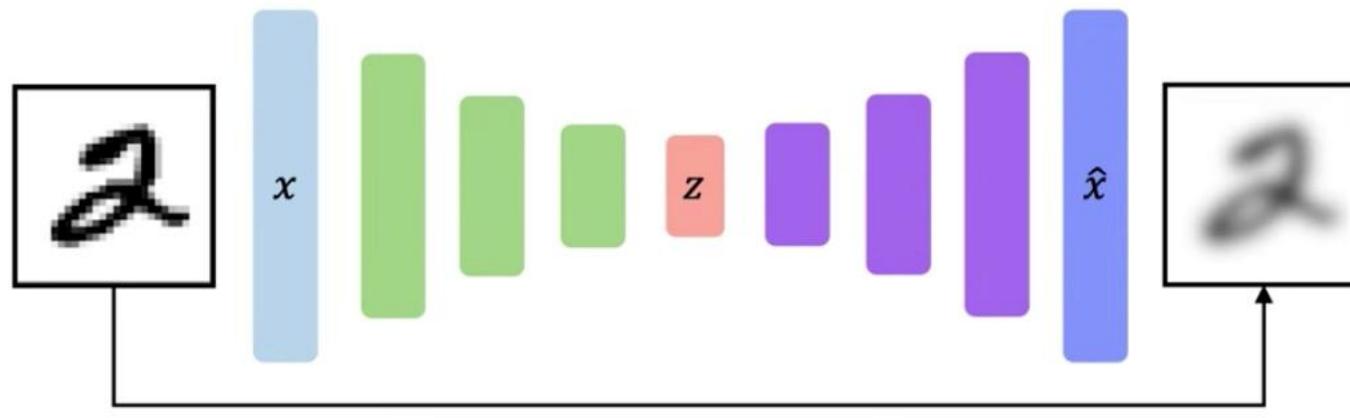
There are no restrictions on the architecture and choice of layers.



Training autoencoders

Simply minimize the error between output and input!

Unsupervised: no labels used to train parameters, hence the “auto” part.



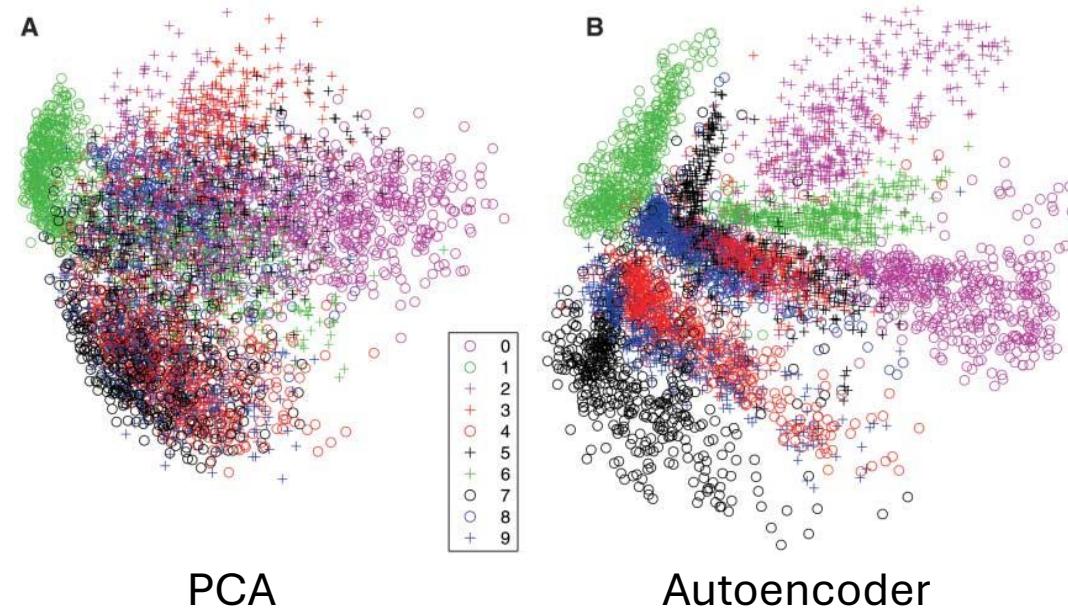
$$\mathcal{L}(x, \hat{x}) = \|x - \hat{x}\|^2$$

Purpose of autoencoders

Learning lower-dimensional feature representations.

Compression / invariance / redundancy removal.

Fig. 3. (A) The two-dimensional codes for 500 digits of each class produced by taking the first two principal components of all 60,000 training images. (B) The two-dimensional codes found by a 784-1000-500-250-2 autoencoder. For an alternative visualization, see (8).



Generative models from autoencoders?

We can technically generate samples by picking points in latent space z .

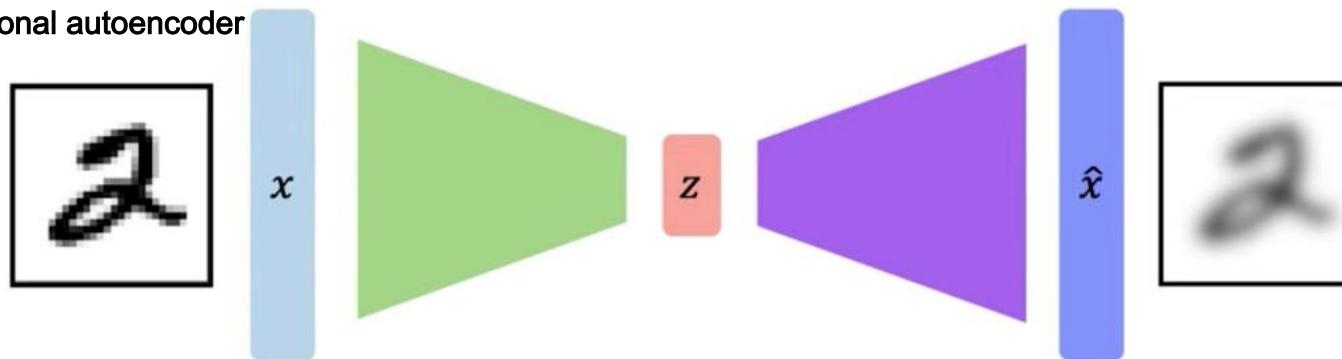
Let's draw z on the board and see how we can generate new samples!

One technique is to sample from space z , but I obtain random noise. Z is unconstrained so I obtain noise. I'm very unlikely to sample something close to a real image. The cue are that the samples in the embedded space are not random but came from the original distribution, the more I sample near one of those points the more I'll generate an image like the one which those point represents.

We can use a Gaussian mixture model to sample from the space.

One problem: the images are ugly, they are blurry and ugly

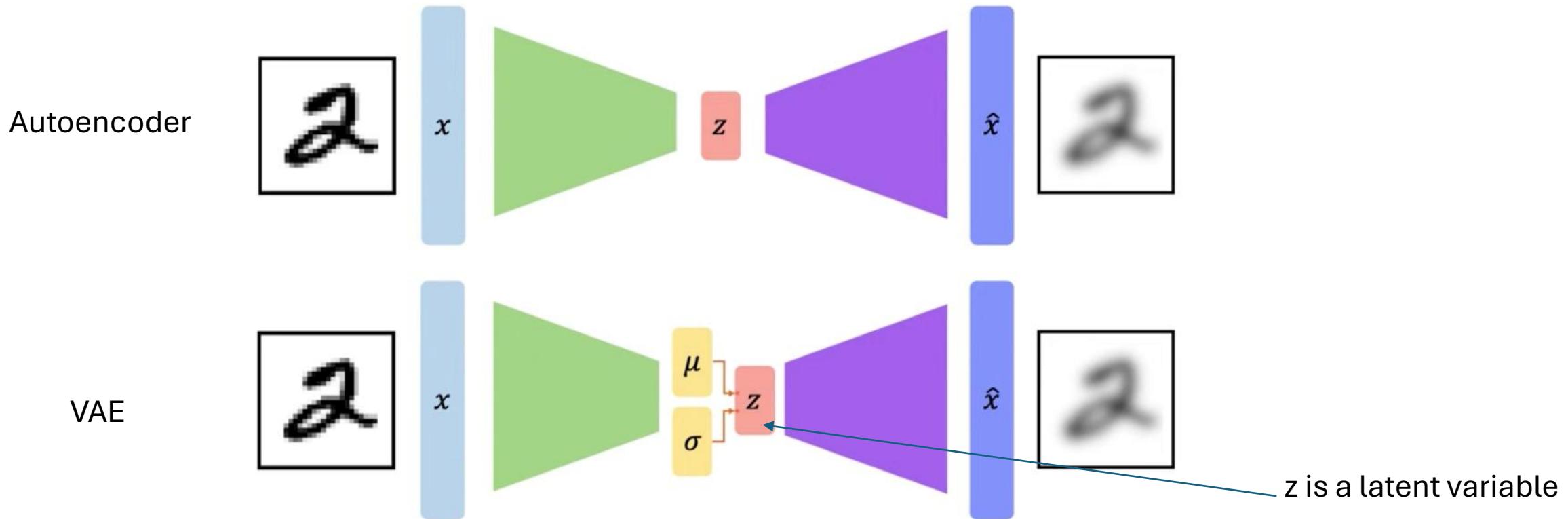
We add a constraint to obtain the variational autoencoder



what are the points in the
embedded space?

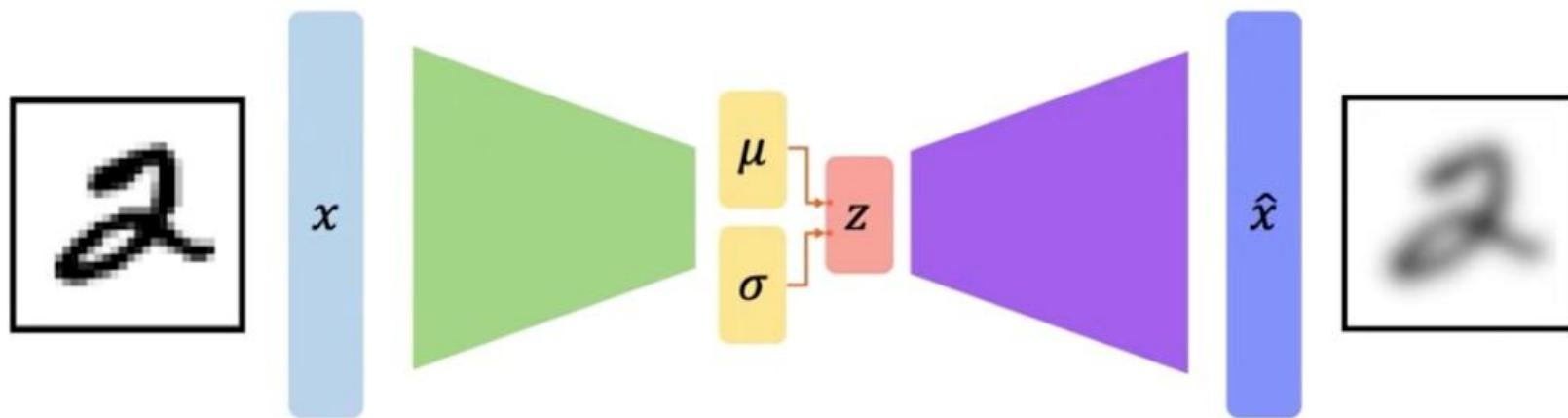
Variational autoencoder

Natural solution: constrain the latent space to follow a Gaussian distribution.



Solving VAEs intuitively

Which loss or losses would you use to get an autoencoder with a latent space that has a density alin to a Gaussian?

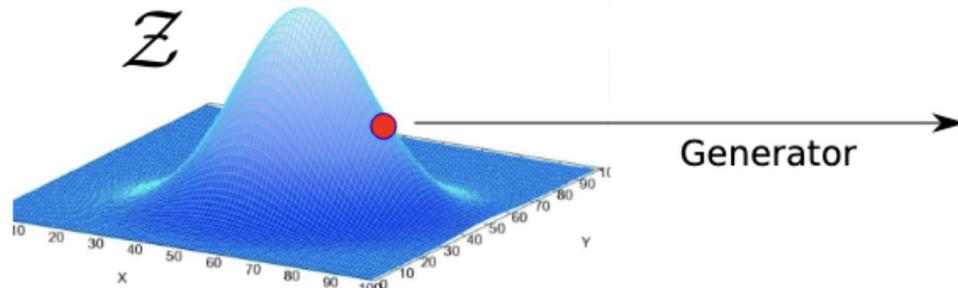


i split it into 2 parts, from the input to the embedded space I want it to be a gaussian, then, from z to the reconstruction i will compute the loss, we have a second loss in the middle

Main idea of VAEs

Train encoder and decoder with Gaussian constraint.

During inference, sample from the constrained space to produce new samples.

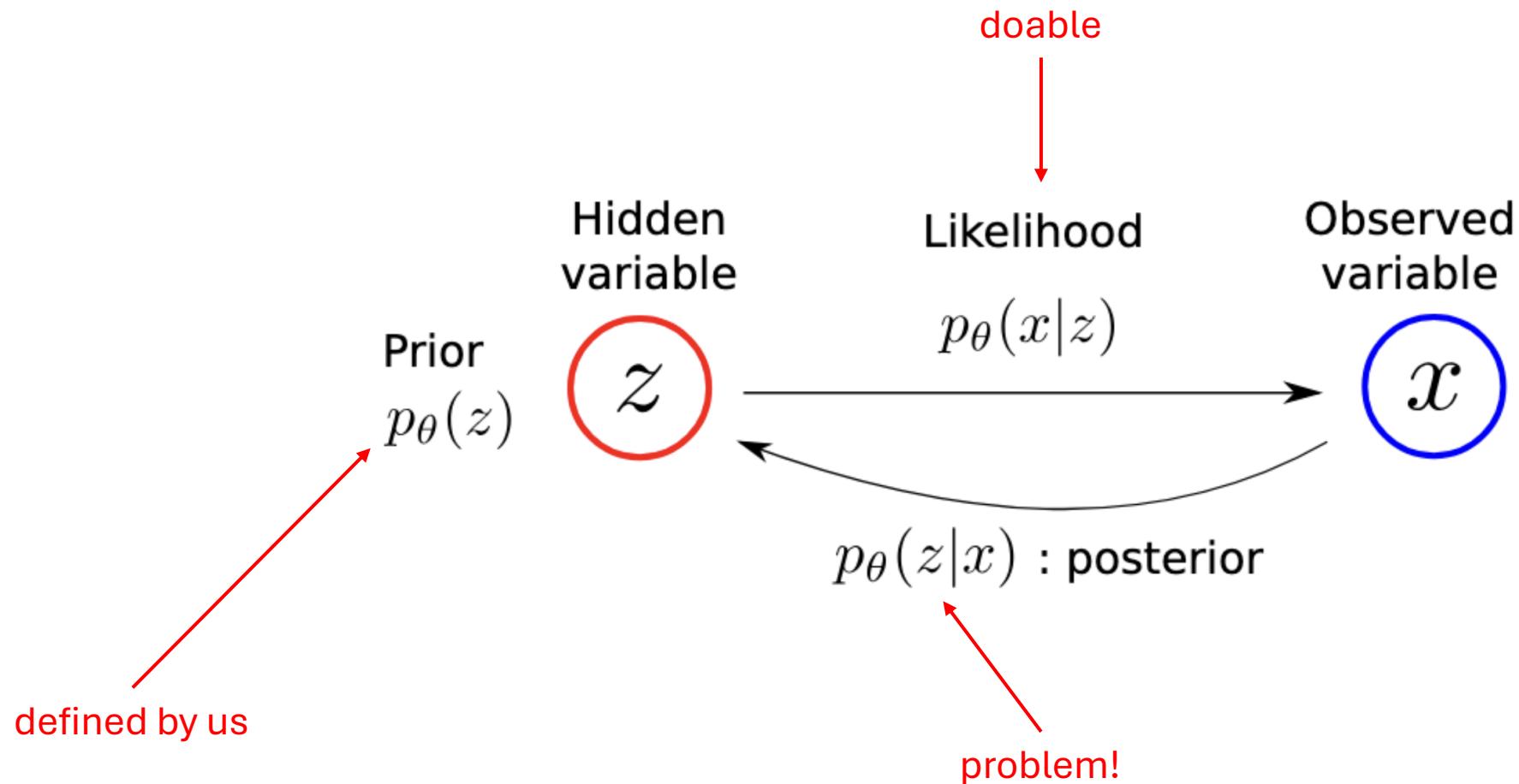


Probabilistic model in latent space

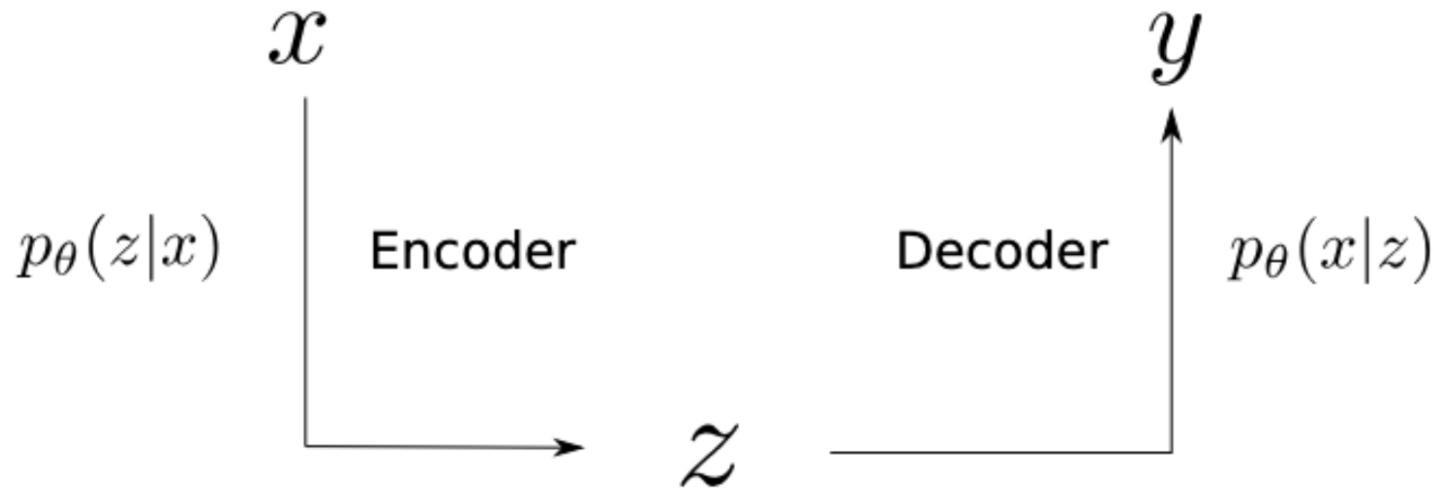


Synthesis of random image

Probability density functions in a VAE



Analogies between Bayes and autoencoders



Encoder : posterior $p_{\theta}(z|x)$

Decoder : likelihood $p_{\theta}(x|z)$

Solving the inference problem

Distribution $p_\theta(z|x)$ is unknown, can we approximate it with a network?

Our goal: Approximate $p_\theta(z|x)$ with $q_\phi(z|x)$ as:

$$q_\phi^* = \arg \min_{q_\phi} KL(q_\phi(z|x) \parallel p_\theta(z|x))$$

But we don't know $p_\theta(z|x)$, so we've gained nothing.

We need to optimize this objective in another way.

The ELBO

evidence lower bound

What will we set as our objective if the marginal log-likelihood is intractible?

$$\log p_\theta(x) = \text{ELBO}(q_\phi) + KL(q_\phi(z|x) \parallel p_\theta(z|x))$$

↑ ↑ ↑
intractable our way in unknown

The KL divergence on the right is positive only, so ELBO is a lower bound.

Maximizing this ELBO minimizes the KL divergence on the right.

The Evidence Lower BOund

the RHS is the Standard gaussian (0 mean and unit variance)

the LHS is the actual distribution obtained by the encoder

The ELBO consists of two parts, with both known.

by minimizing this we force it to be a Standard Gaussian

$$\text{ELBO}(q_\phi) = \mathbb{E}_{q_\phi} [\log(p_\theta(x|z))] - KL(q_\phi(z|x) || p_\theta(z))$$



This is our
reconstruction error.



This is our
way to enforce the prior.

this forces the latent variables to
distribute like a gaussian

The LHS represents the expected log-likelihood, which is the log-probability the model assigns to the observed data given the latent variable. Maximizing this term is equivalent to minimizing the reconstruction error, because higher log-likelihood means the output of the decoder matches the input data more closely

We can implement these two parts as losses to *maximize*.

This maximization is only the form that the loss takes, in reality we want to maximize the LHS and minimize the RHS

VAEs summarized

We extend autoencoders with an alternative with a Gaussian as latent.

Direct optimization requires calculating the posterior, which is not feasible.

Instead we approximate it with a simpler (learned) function.

This function is optimized through the ELBO.

VAEs beyond the lecture

Optimization requires backpropagating through random variables.

This is not differentiable, solution: reparametrisation trick.

You will figure this out as part of assignment 3!

For in-depth derivations, this resource by Yuge Shi recommended:

How I learned to stop worrying and write ELBO (and its gradients) in a billion ways

<https://yugeten.github.io/posts/2020/06/elbo/>

Generative learning 2: The adversarial era

Explicit versus implicit density

With $p^*(x)$ being the real distribution:

- The model p_θ assigns high density to samples taken from the true distribution p^* :

$$x \sim p^*(x) \implies p_\theta(x) \text{ is "high".}$$

Explicit density

- Samples taken from the model p_θ behave similarly to real samples from p^* :

$$x \sim p_\theta(x) \implies p^*(x) \text{ is "high".}$$

Implicit density

Why learn implicit densities?

Learning explicit densities is hard.

Often in practice, we care only about how something looks.

This is the idea behind the Generative Adversarial Network (GAN).

GAN: High quality generation through game theory



Synthetic Data Generation for Fraud Detection using GANs

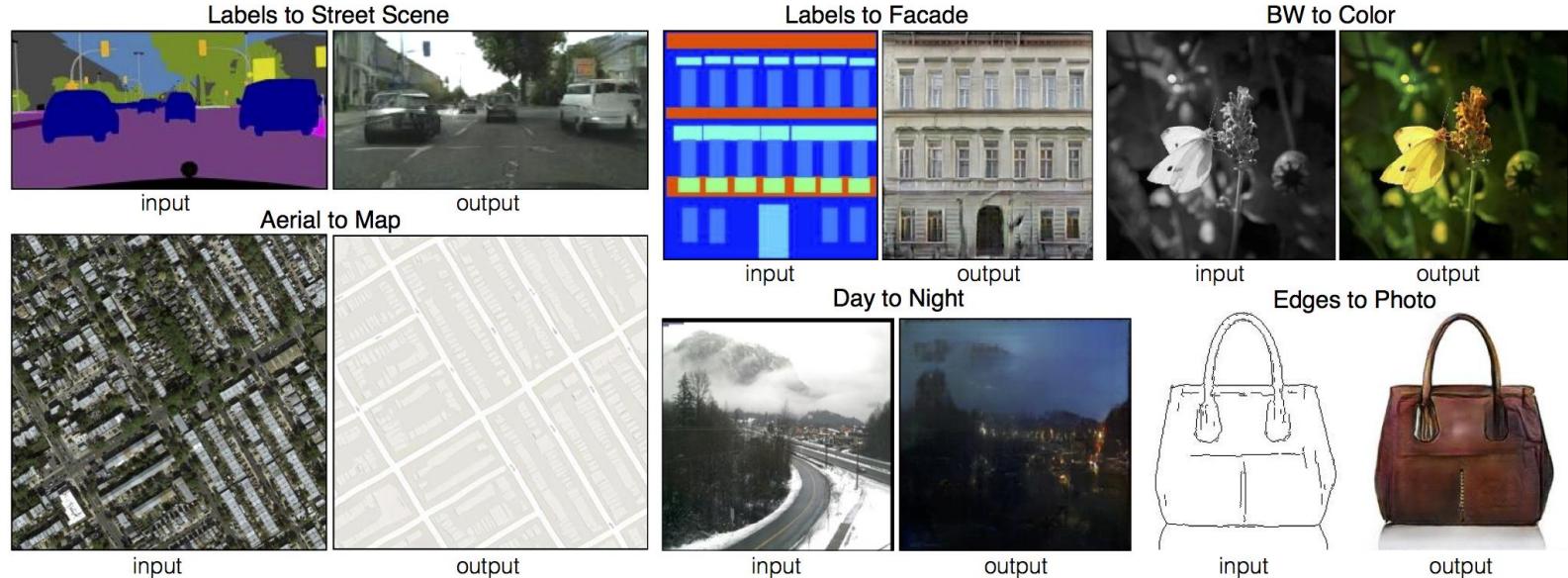
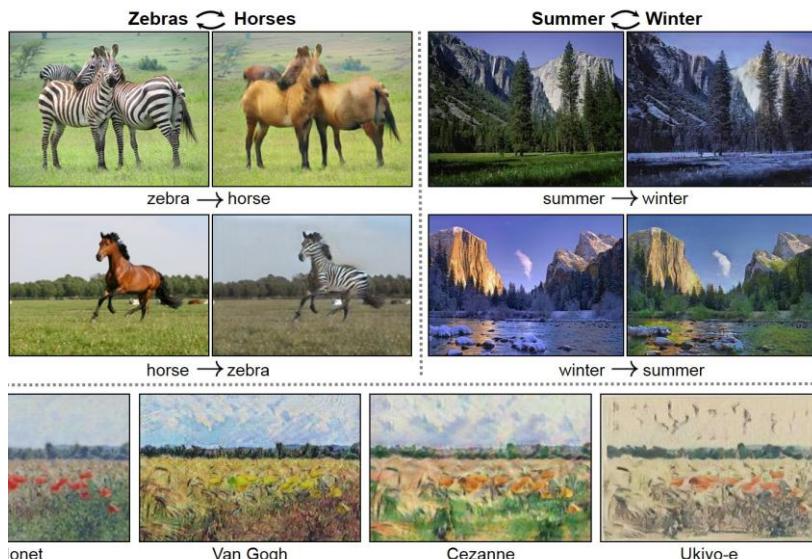
Charitos Charitou
Department of Computer Science
City, University of London
London, UK
charitos.charitou@city.ac.uk

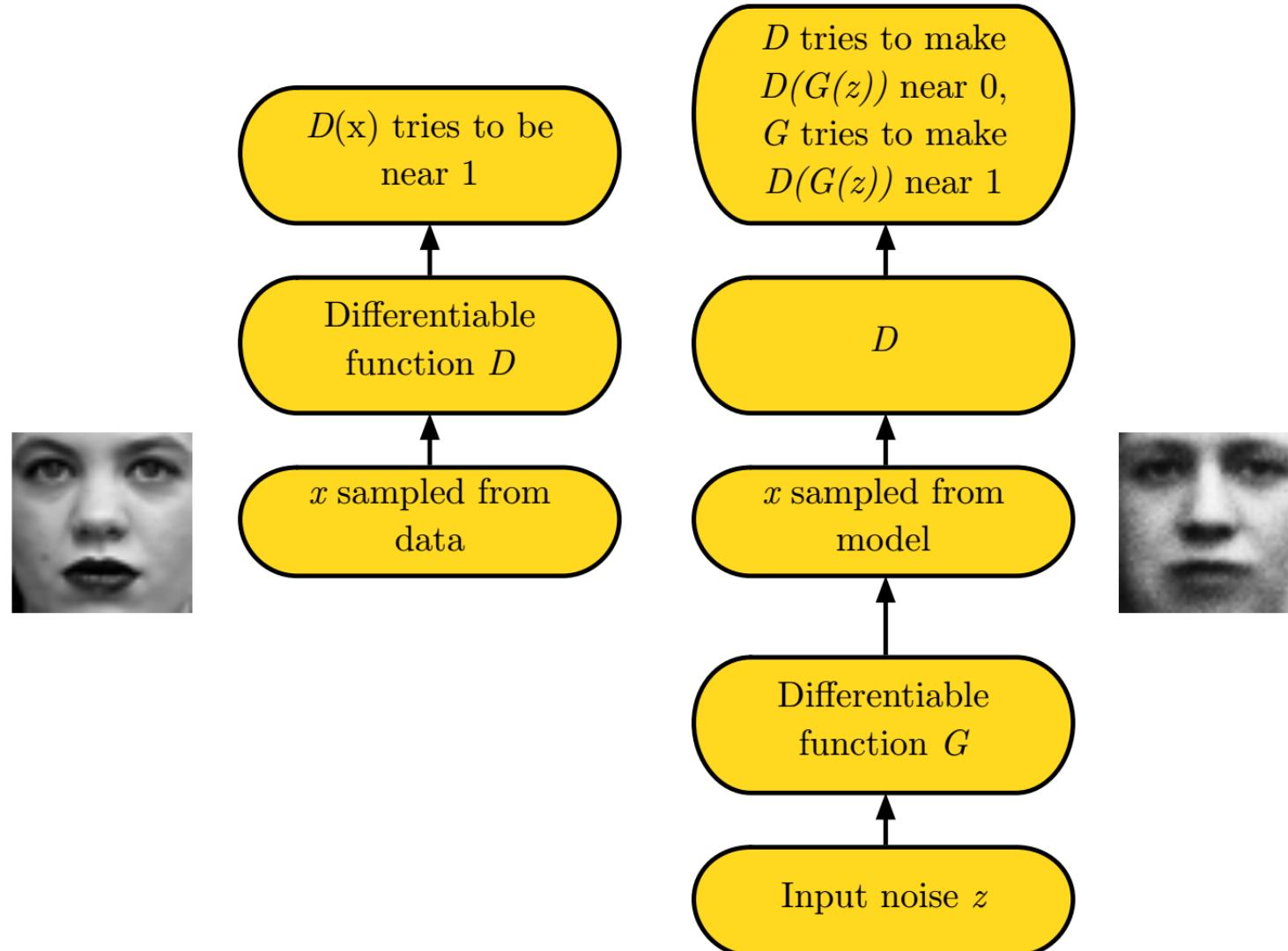
Generative Adversarial Networks recover features in astrophysical images of galaxies beyond the deconvolution limit

Kevin Schawinski,^{1*} Ce Zhang,^{2†} Hantian Zhang,² Lucas Fowler,¹ and Gokula Krishnan Sathanam²

¹Institute for Astronomy, Department of Physics, ETH Zurich, Wolfgang-Pauli-Strasse 27, CH-8093, Zürich, Switzerland

²Systems Group, Department of Computer Science, ETH Zurich, Universitätstrasse 6, CH-8006, Zürich, Switzerland





What is a GAN?

Generative

You can sample novel inputs and “create” what never existed.

Adversarial

Train two models: a generator (creator) and a discriminator (evaluator).

Network

Implemented as a deep network and learned with backprop.

Intuition behind GANs

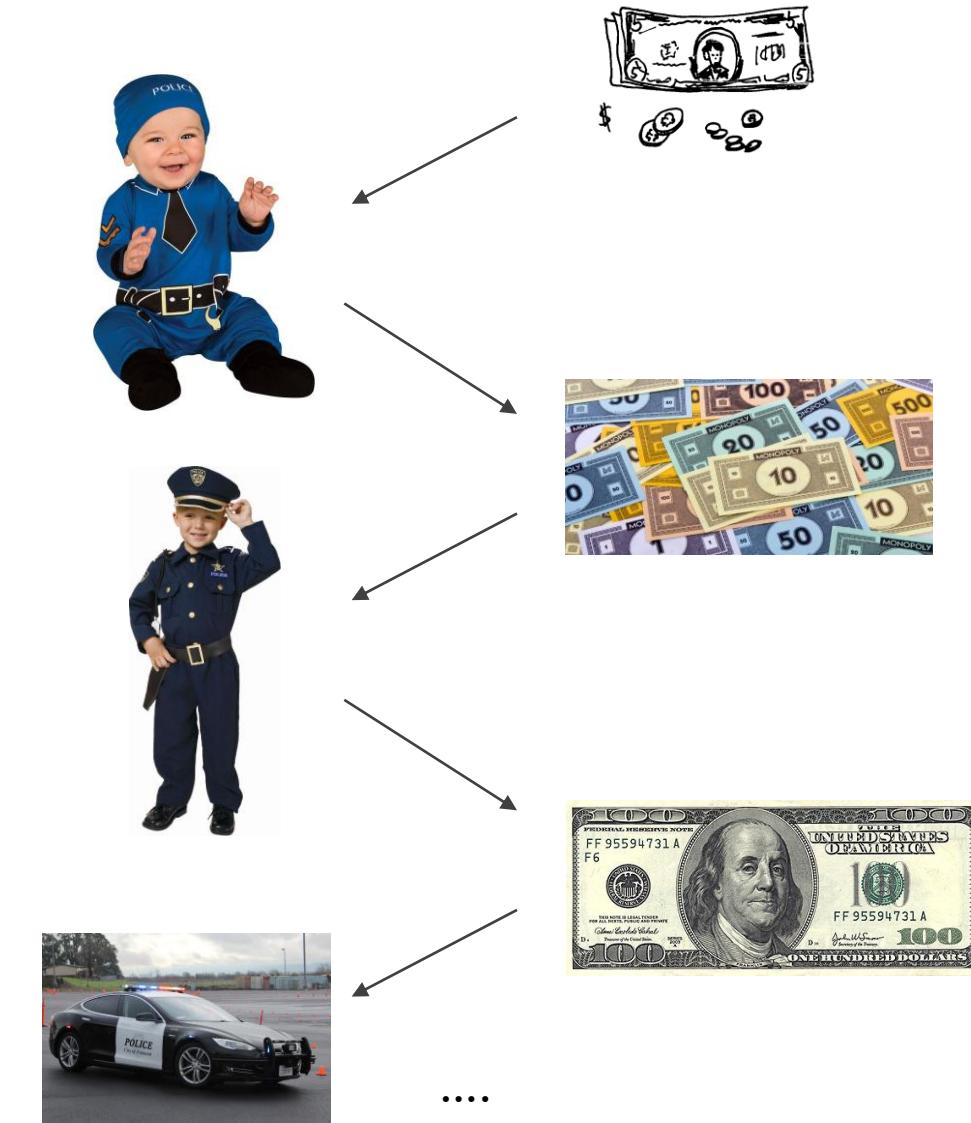
Police: wants to detect fake money as reliably as possible.

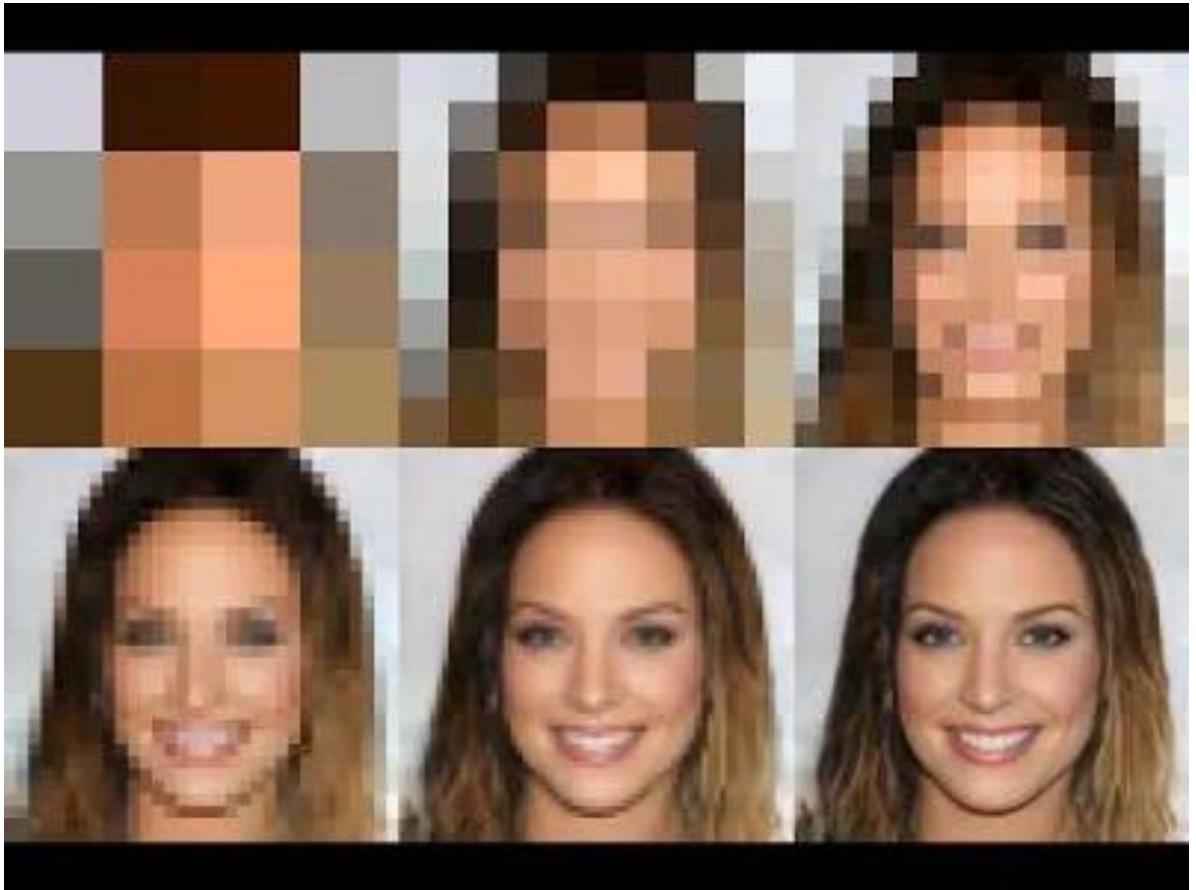
Counterfeiter: wants to make as realistic fake money as possible.

At beginning: both have no clue.

The police forces the counterfeiter to get better as it compares it to real money (and vice versa).

Convergent solution ~ Nash equilibrium



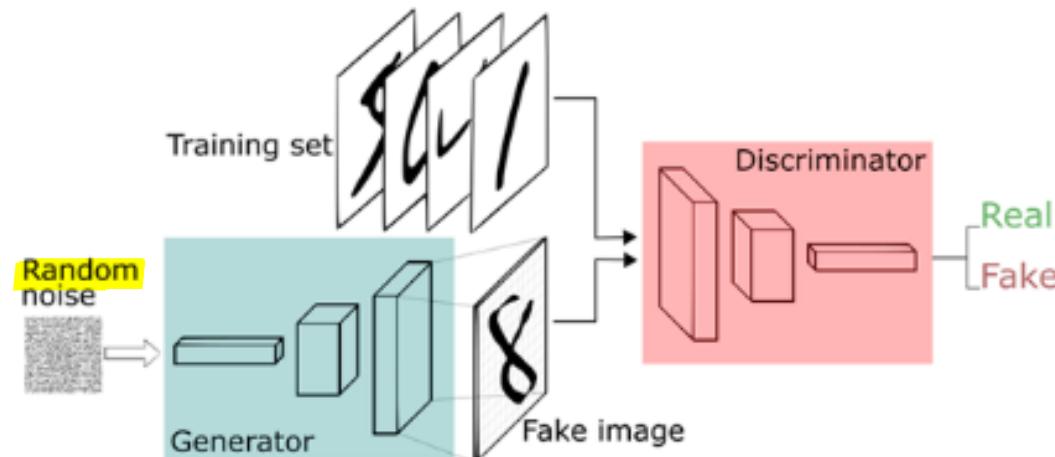


GAN architecture

The GAN comprises two neural networks:

Generator network $x = G(\mathbf{z}; \theta_G)$

Discriminator network $y = D(x; \theta_D) = \begin{cases} +1, & \text{if } x \text{ is predicted 'real'} \\ 0, & \text{if } x \text{ is predicted 'fake'} \end{cases}$



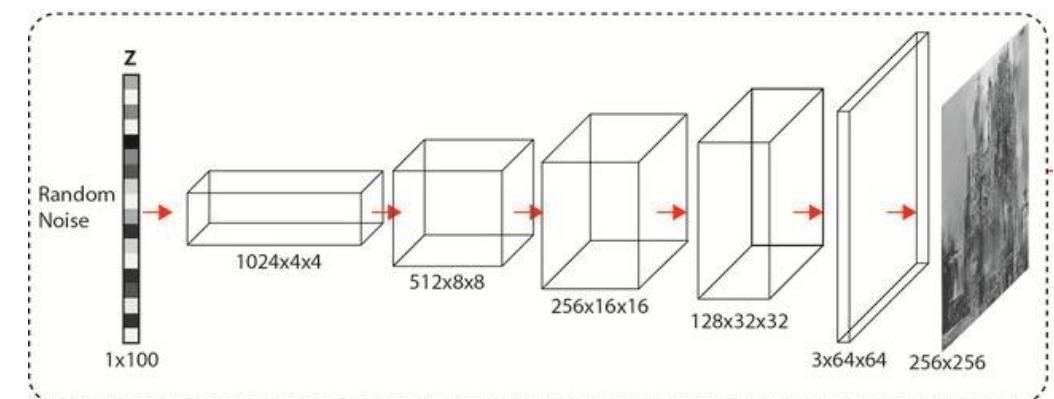
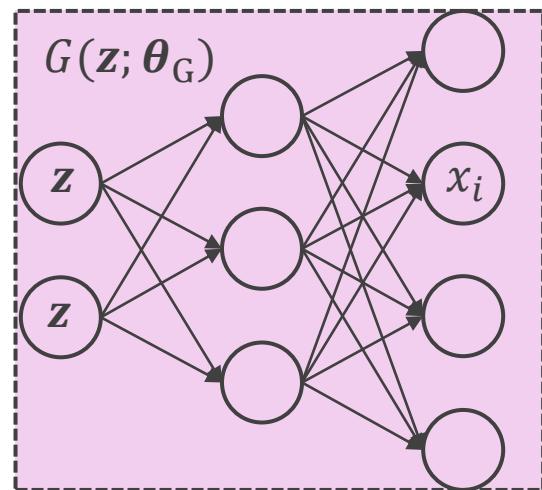
GAN generator $x = G(z; \theta_G)$

Any differentiable neural network.

Starts with some random, typically lower dimensional input z .

Various density functions for the noise variable z .

$z \sim \mathcal{N}(0,1)$ or
 $z \sim \text{Uniform}(0,1)$
...



GAN discriminator $y = D(x; \theta_D)$

Any differentiable neural network.

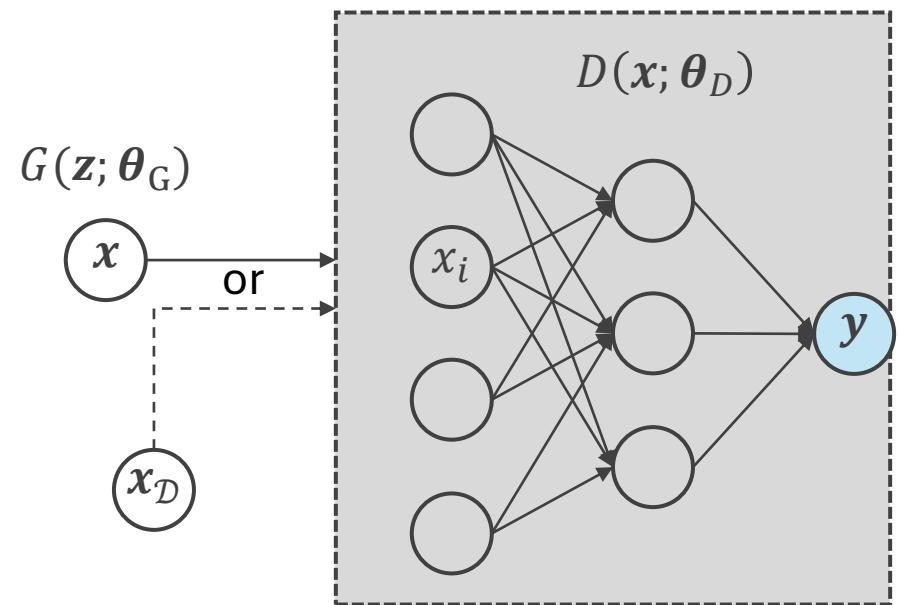
Receives as inputs:

- either real images from the training set
- or generated images from the generator
- usually a mix of both in mini-batches

Must recognize the real from the fake inputs.

The discriminator loss:

$$\begin{aligned} J_D(\theta_D, \theta_G) &= \frac{1}{2} \text{BCE}(Data, 1) + \frac{1}{2} \text{BCE}(fake, 0) \\ &= -\frac{1}{2} \mathbb{E}_{x \sim p_{data}} [\log D(x)] - \frac{1}{2} \mathbb{E}_{z} [\log(1 - D(G(z)))] \\ &= -\frac{1}{2} \mathbb{E}_{x \sim p_{data}} [\log D(x)] - \frac{1}{2} \mathbb{E}_{x \sim p_{generator}} [\log(1 - D(x))] \end{aligned}$$



Binary Cross Entropy loss:

$$l_n = -w_n [y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n)],$$

However, in the context of GANs, the loss appears as a weighted sum because the discriminator is trained on two separate batches with different objectives in the same training step.

GAN implementation

The discriminator is just a standard neural network.

The generator looks like an inverse discriminator (or like a decoder).

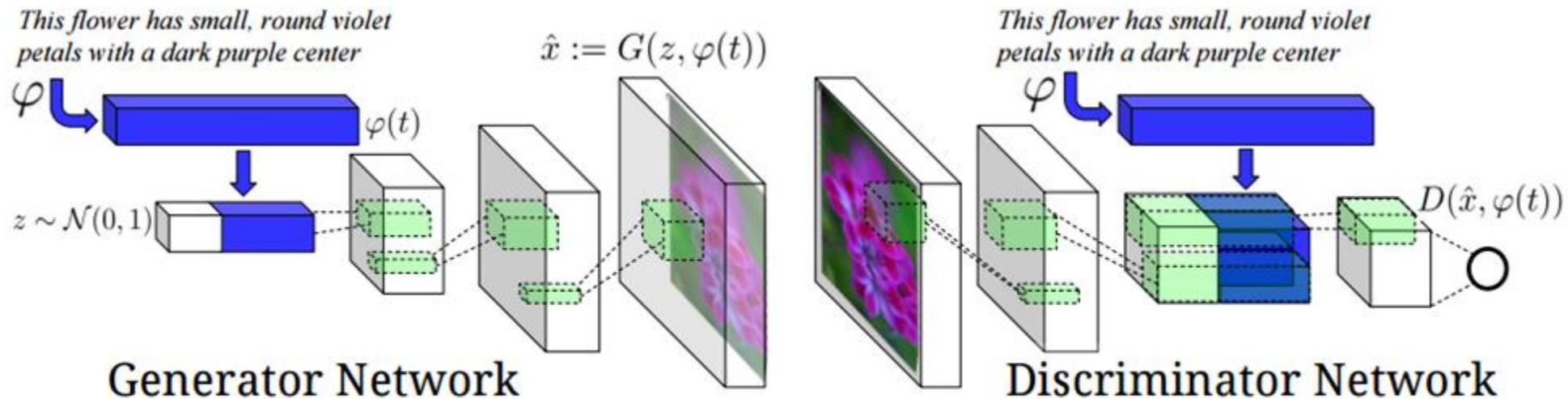


Figure 2. Our text-conditional convolutional GAN architecture. Text encoding $\varphi(t)$ is used by both generator and discriminator. It is projected to a lower-dimensions and depth concatenated with image feature maps for further stages of convolutional processing.

How to train a GAN

Given a generated image, we do not have its “equivalent” image in our batch.

The model generates some random images independent of comparison batch.

How can we get meaningful gradients?

The minimax loss

Simplest case: Generator loss is negative discriminator loss (“zero-sum game”).

$$J_G = -J_D$$

The lower the generator loss, the higher the discriminator loss

Symmetric definitions

Our learning objective then becomes

$$V = -J_D(\theta_D, \theta_G)$$

$D(x) = 1 \rightarrow$ The discriminator believes that x is a true image

$D(G(z)) = 1 \rightarrow$ The discriminator believes that $G(z)$ is a true image

So overall loss:

$$\text{Minimize}_G \text{Maximize}_D J_D$$

Heuristic non-saturating loss

Discriminator loss (maximize likelihood of correctly labelling real/fake data).

$$J_D = -\frac{1}{2} \mathbb{E}_{x \sim p_{data}} \log D(x) - \frac{1}{2} \mathbb{E}_{z \sim p_z} \log(1 - D(G(z)))$$

Generator loss (maximize likelihood of discriminator being wrong).

$$J_G = -\frac{1}{2} \mathbb{E}_{z \sim p_z} \log(D(G(z)))$$

Generator learns even when discriminator is too good on real images.

Training GANs is a pain

1. Vanishing gradients
2. Batchnorm
3. Convergence
4. Mode collapse

1. Vanishing gradients

If the discriminator is quite bad
→ the generator gets confused
→ no reasonable generator gradients

If the discriminator is near perfect
→ gradients go to 0, no learning anymore

Bad early in the training.

Easier to train the discriminator than generator.

the discriminator very quickly is too powerful but if it is too weak the generator cannot learn to generate good images. There is a very narrow span

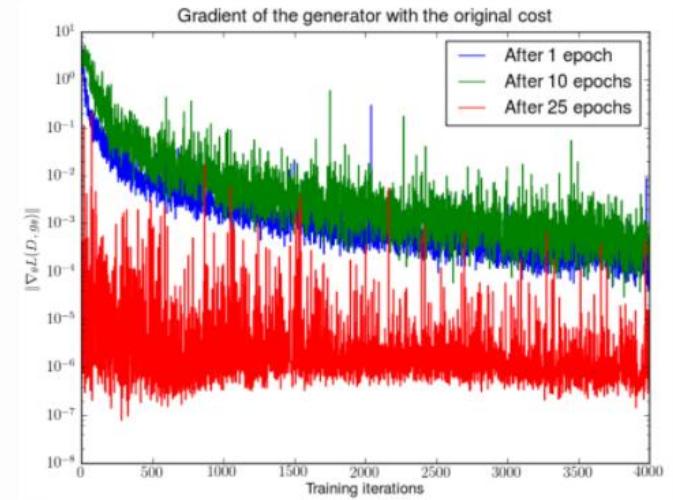


Fig. 5. First, a DCGAN is trained for 1, 10 and 25 epochs. Then, with the **generator fixed**, a discriminator is trained from scratch and measure the gradients with the original cost function. We see the gradient norms **decay quickly** (in log scale), in the best case 5 orders of magnitude after 4000 discriminator iterations. (Image source: [Arjovsky and Bottou, 2017](#))

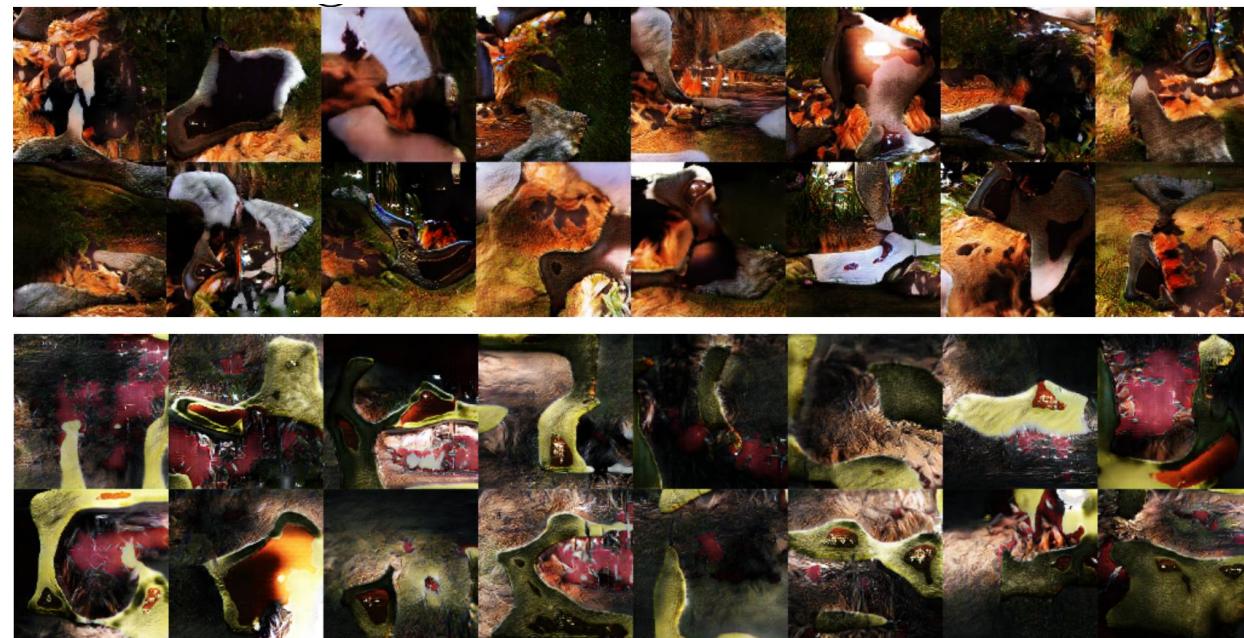
2. Batchnorm

Batch-normalization causes strong intra-batch correlation. because it computes mean and variances across different samples

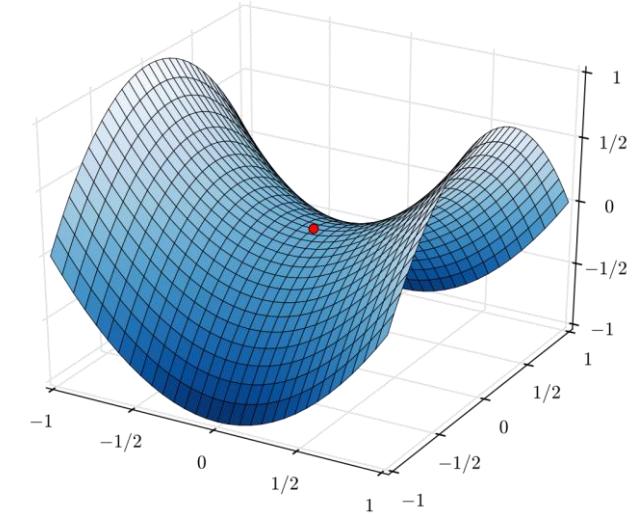
Generations look smooth but awkward.

Easiest solution: drop batchnorm.

(See e.g. StyleGAN)



3. Convergence



Optimization is tricky and unstable.

- finding a saddle point does not imply a global minimum
- A saddle point is also sensitive to disturbances

An equilibrium might not even be reached (models can train for weeks).

Mode-collapse is the most severe form of non-convergence.

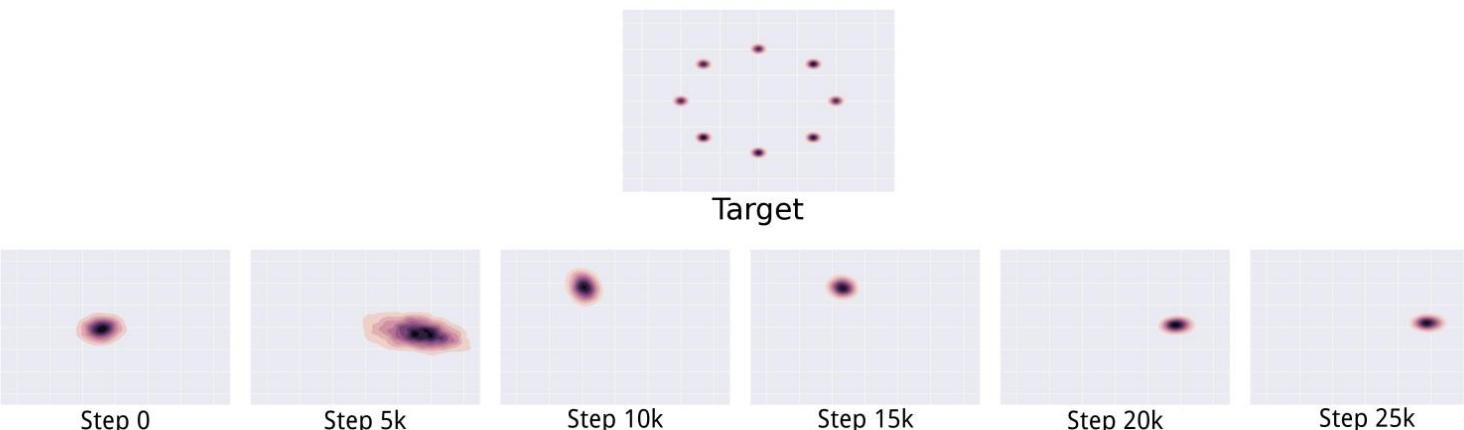
4. Mode collapse

Discriminator converges to the correct distribution

Generator however places all mass in the most likely point

All other modes are ignored (underestimating variance)

Low diversity in generating samples



I want to increase the variance of my data



Bias in generative models



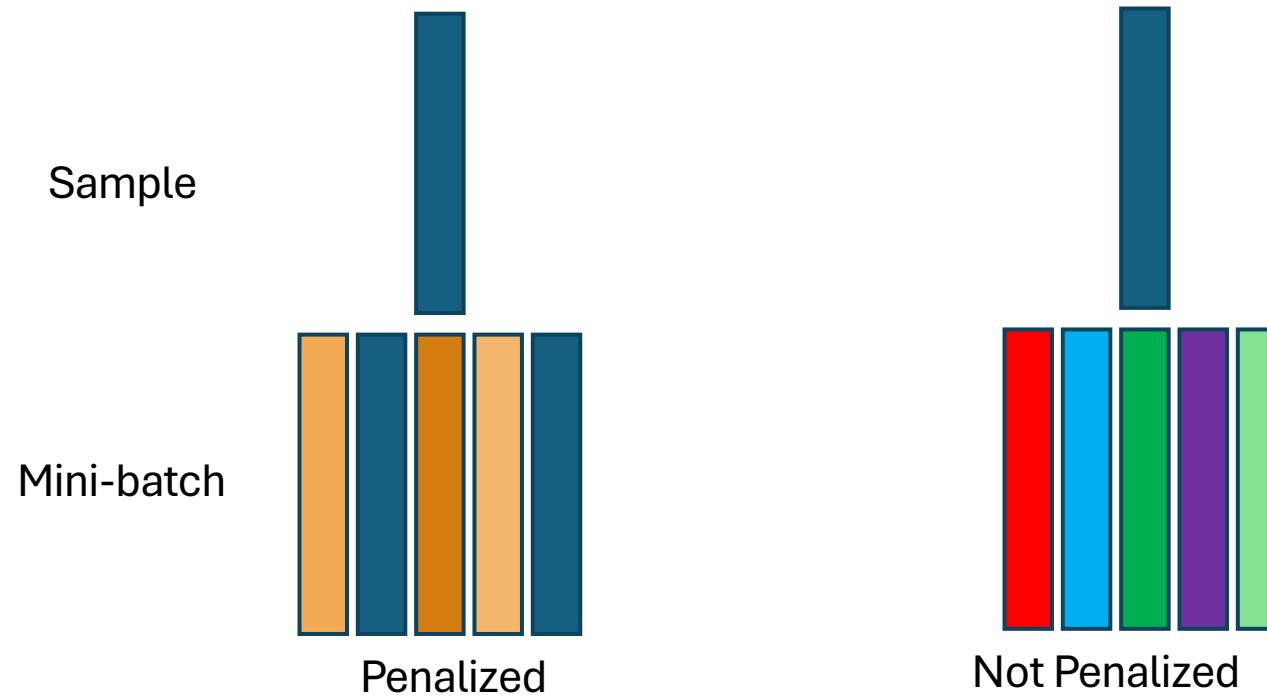
Imperfect ImaGANation: Implications of GANs Exacerbating Biases on Facial Data Augmentation and Snapchat Face Lenses

Niharika Jain^{*}, Alberto Olmo^{*}, Sallik Sengupta^{*}, Lydia Manikonda^{*}, and Subbarao Kambhampati[†]

^{*}Arizona State University, Tempe, Arizona; [†]Rensselaer Polytechnic Institute, Troy, New York

Addressing mode collapse

A simple trick: compare each sample to others in a mini-batch and add a penalty for high similarities within a mini-batch.

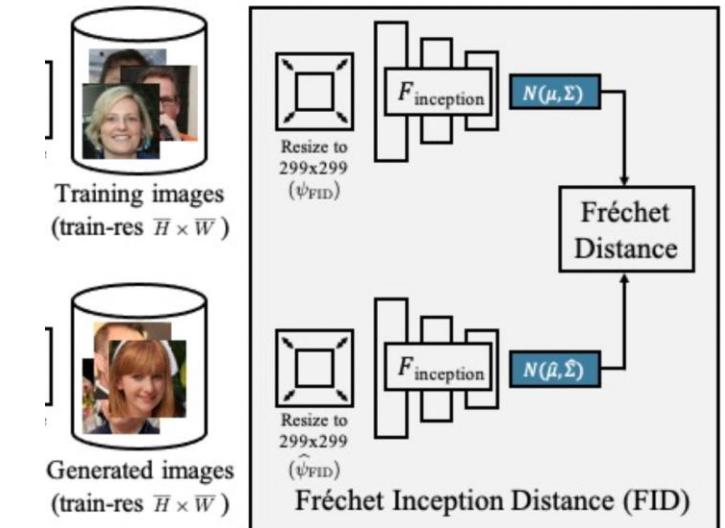


How do we actually evaluate GANs?

General problem for generative learning, how to know if your model is better?

There are image quality measures like FID, but these are naturally limited.

Best estimate: ask people to rank manually.



Generative learning 3: The diffusion era

Back to autoencoders and Gaussians

VAEs try to directly learn a mapping from inputs to a Gaussian.

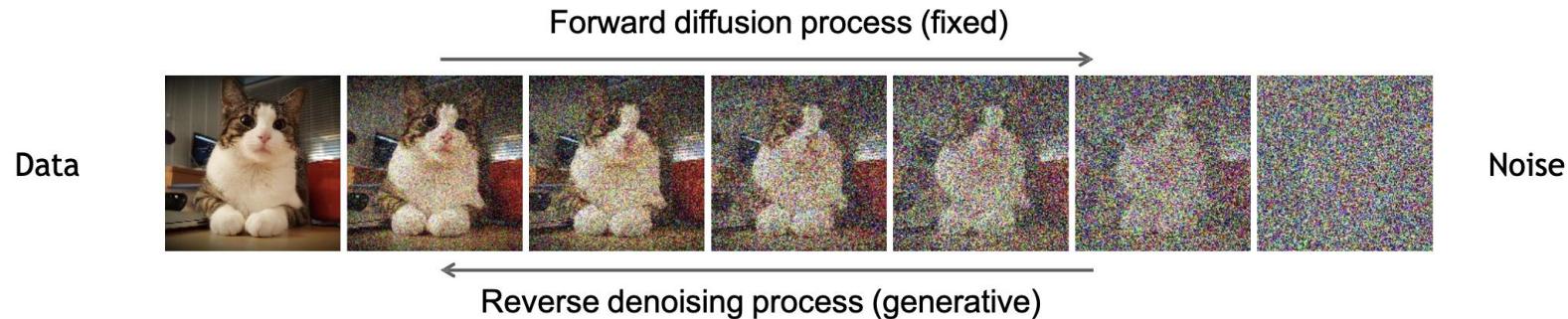
This seems like a big leap, what if we do this in a gradual manner?

I'll give a short primer here, diffusion models come back in CV2 and FoMo.

Denoising diffusion models

Forward process: take input can gradually add noise (encoder).

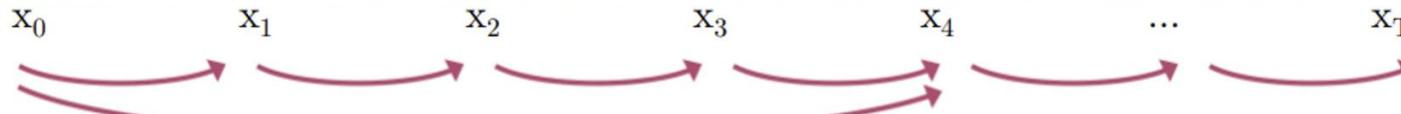
Reverse process: learn to generate data from noise (decoder).



Forward process of diffusion models



$x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow x_4 \rightarrow \dots \rightarrow x_T$



Markov
Property

$$q(\mathbf{x}_t | \mathbf{x}_{t-1}) := \mathcal{N}(\mathbf{x}_t; \sqrt{1 - \beta_t} \mathbf{x}_{t-1}, \beta_t \mathbf{I})$$

$$q(x_t | x_0) = \mathcal{N}(x_t; \sqrt{\bar{\alpha}_t} x_0, (1 - \bar{\alpha}_t) \mathbf{I})$$

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t} \mathbf{x}_0 + \sqrt{(1 - \bar{\alpha}_t)} \boldsymbol{\epsilon} \quad \text{where } \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$$

$$\alpha_t := 1 - \beta_t \text{ and } \bar{\alpha}_t := \prod_{s=0}^t \alpha_s$$

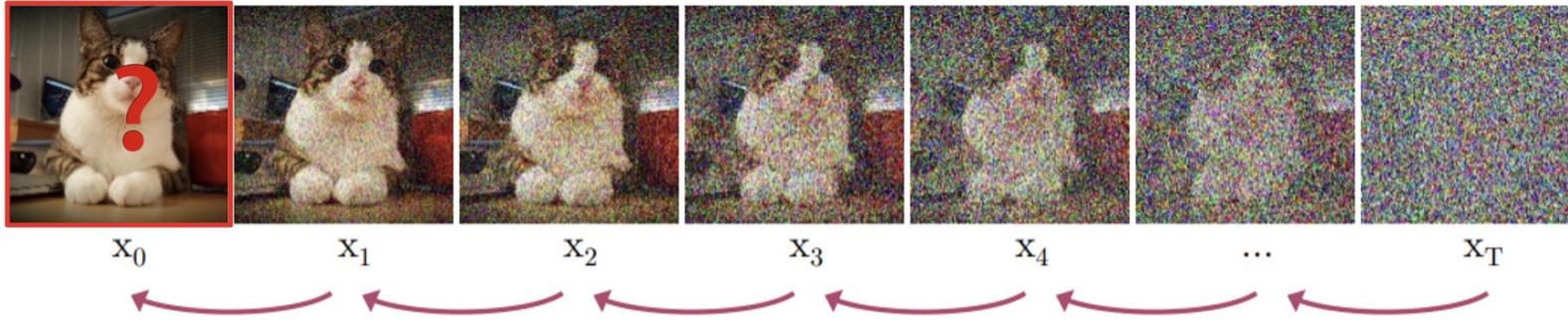
hyperparameters are the number of steps and the variance schedule

Scales down the input and adds noise.

Diffusion Kernel

Variance schedule.

Reverse process of diffusion models

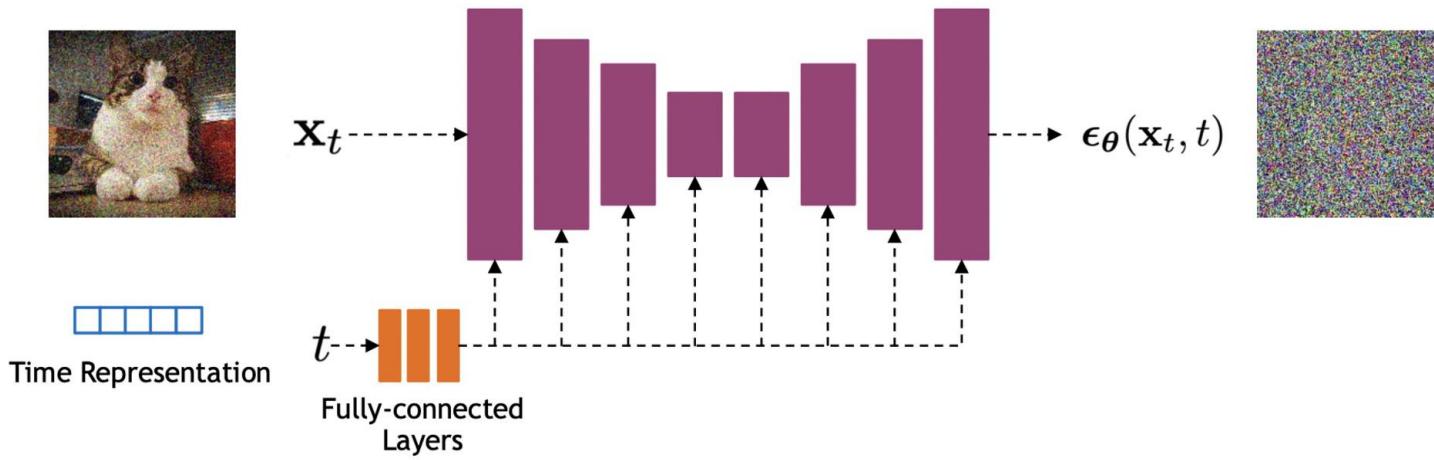


$$q(\mathbf{x}_{t-1} | \mathbf{x}_t, \mathbf{x}_0) = \mathcal{N}(\mathbf{x}_{t-1}; \tilde{\boldsymbol{\mu}}_t(\mathbf{x}_t, \mathbf{x}_0), \tilde{\beta}_t \mathbf{I})$$



$$p_\theta(x_{t-1} | x_t) := \mathcal{N}(x_{t-1}; \boxed{\mu_\theta(x_t, t)}, \Sigma_\theta(x_t, t))$$

Training diffusion models

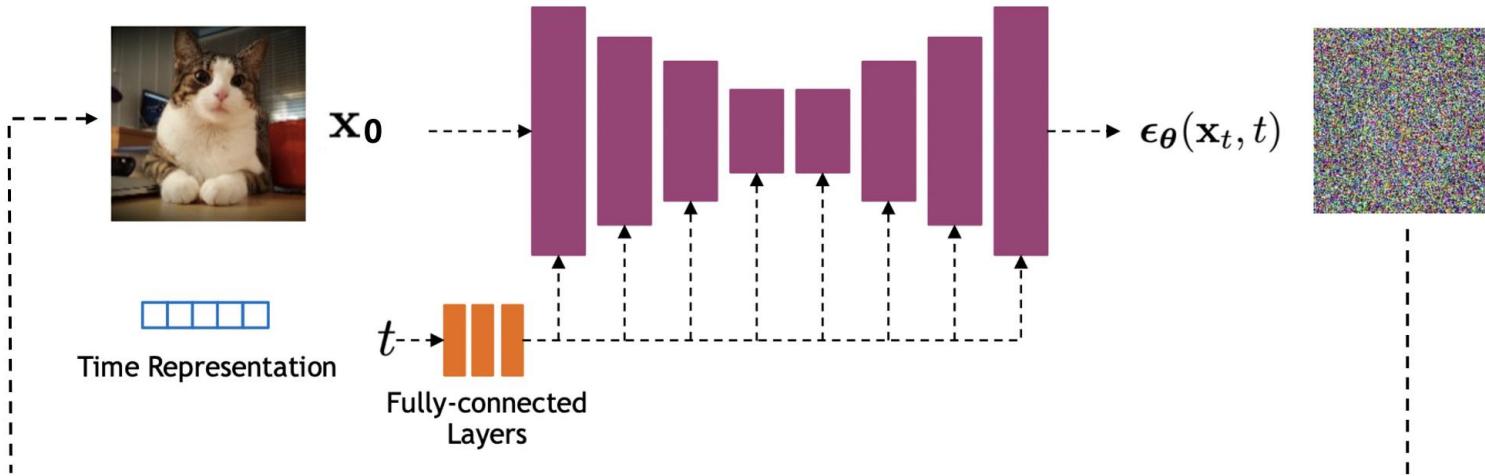


Algorithm 1 Training

```
1: repeat
2:    $\mathbf{x}_0 \sim q(\mathbf{x}_0)$ 
3:    $t \sim \text{Uniform}(\{1, \dots, T\})$ 
4:    $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ 
5:   Take gradient descent step on
     
$$\nabla_\theta \|\boldsymbol{\epsilon} - \mathbf{z}_\theta(\sqrt{\bar{\alpha}_t} \mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t} \boldsymbol{\epsilon}, t)\|^2$$

6: until converged
```

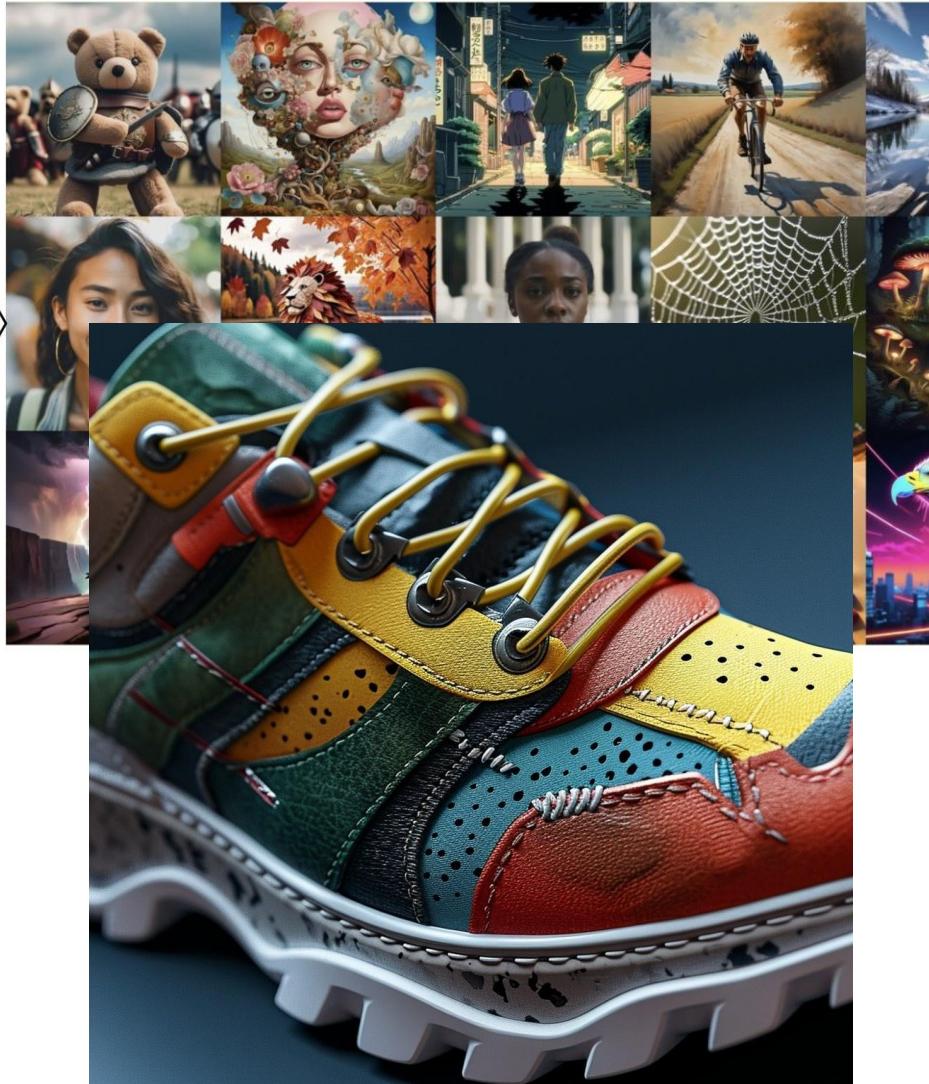
Sampling from diffusion models



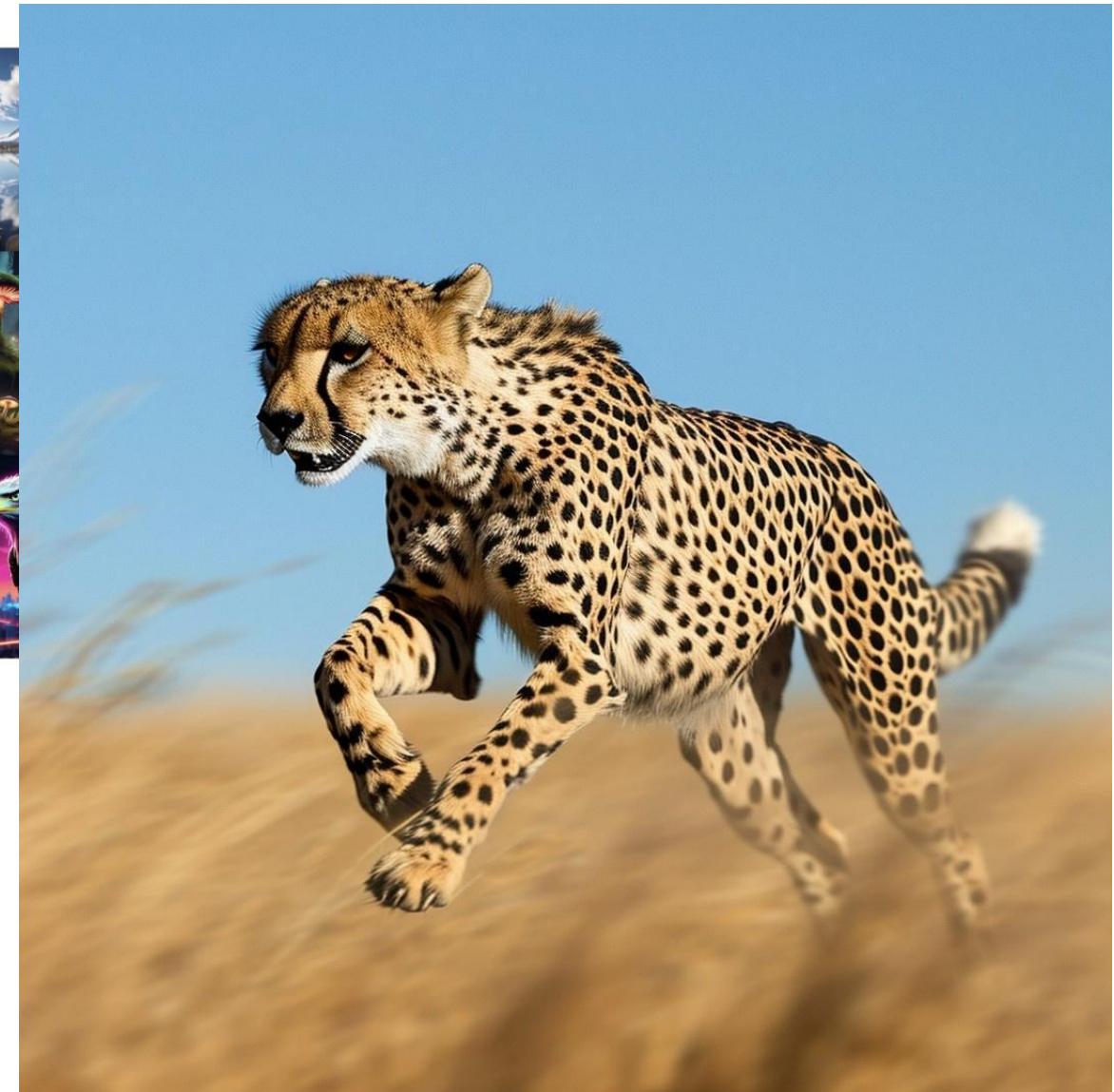
Algorithm 2 Sampling

```
1:  $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ 
2: for  $t = T, \dots, 1$  do
3:    $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$  if  $t > 1$ , else  $\mathbf{z} = \mathbf{0}$ 
4:    $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left( \mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \mathbf{z}_{\theta}(\mathbf{x}_t, t) \right) + \sigma_t \mathbf{z}$ 
5: end for
6: return  $\mathbf{x}_0$ 
```

Impact of diffusion models



Diffusion



Impact of diffusion models

Key behind state-of-the-art generative models:

Midjourney, Imagen, Sora, Dall-E, Stable Diffusion

Intuitive process, as we operate directly in pixel space.

Downside: as we directly operate in pixel space, memory is an issue,

especially when dealing with higher resolutions.

Extensions: Latent diffusion models, conditional diffusion models...

we can do diffusion in the embedding space

Ethical side of generative learning

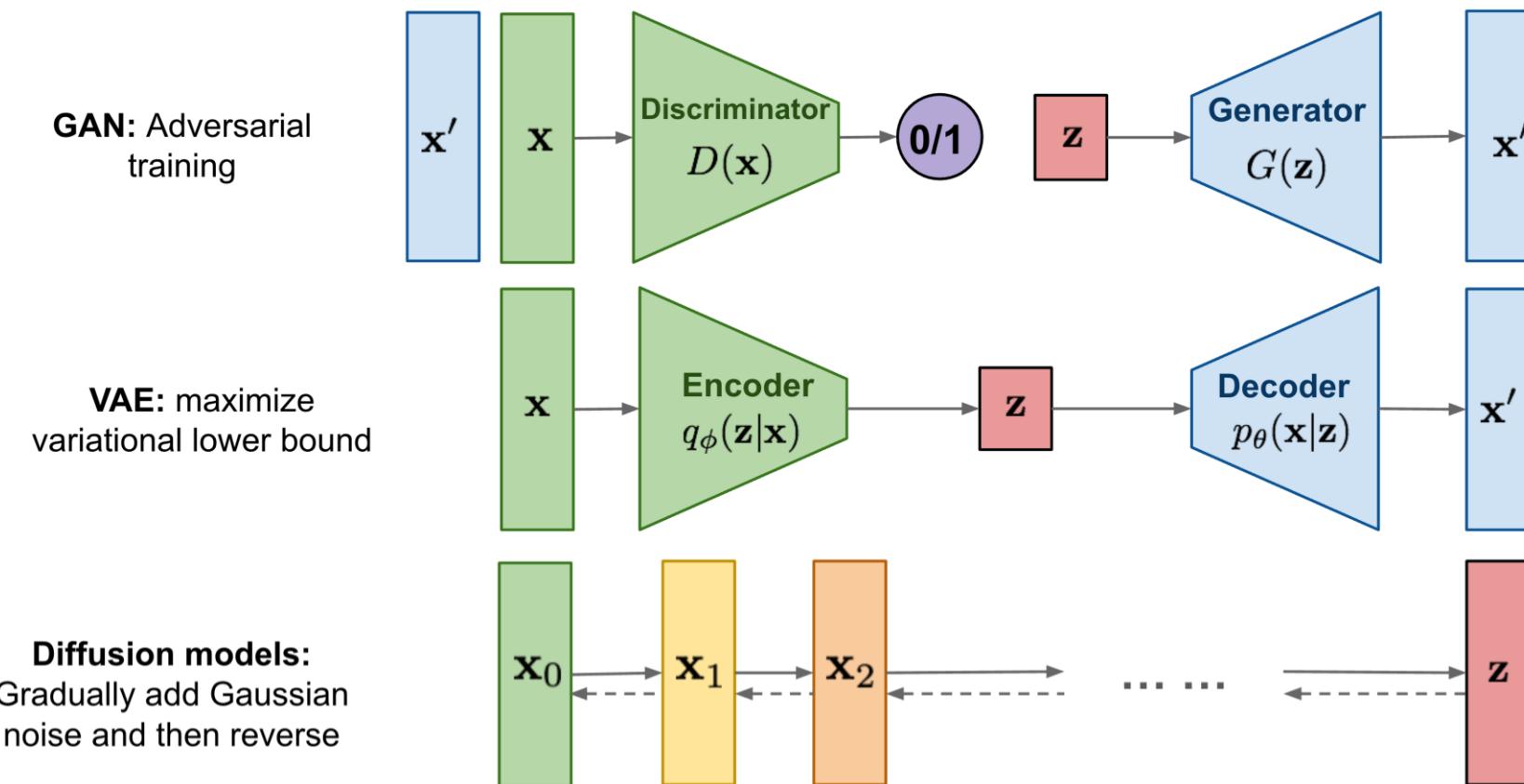
Line between real and generated data becomes blurry.

Data on which large models are trained break IP rights.

Fears for deepfakes, mass manipulation, massive changes in job markets.

AI is no longer only in the lab, we need to consider its real-world impact.

Models summarized



Advances topics for your interest

Conditional generative learning: "*Create a picture of a cat on a sunny day.*"

Flow matching.

Generative learning on dynamic data.

Next lecture

Lecture	Title	Lecture	Title
1	Intro and history of deep learning	2	AutoDiff
3	Deep learning optimization I	4	Deep learning optimization II
5	Convolutional deep learning	6	Attention-based deep learning
7	Graph deep learning	8	From supervised to unsupervised deep learning
9	Multi-modal deep learning	10	Generative deep learning
11	What doesn't work in deep learning	12	Non-Euclidean deep learning
13	Q&A	14	Deep learning for videos

Learning and reflection

Understanding Deep Learning, Chapter 15

Understanding Deep Learning, Chapter 17

Understanding Deep Learning, Chapter 18