A crash course in deep learning

Florent Krzakala

florent.krzakala@epfl.ch


TAKE
A
DEEP
BREATH.

A bit of history

Mark I Perceptron

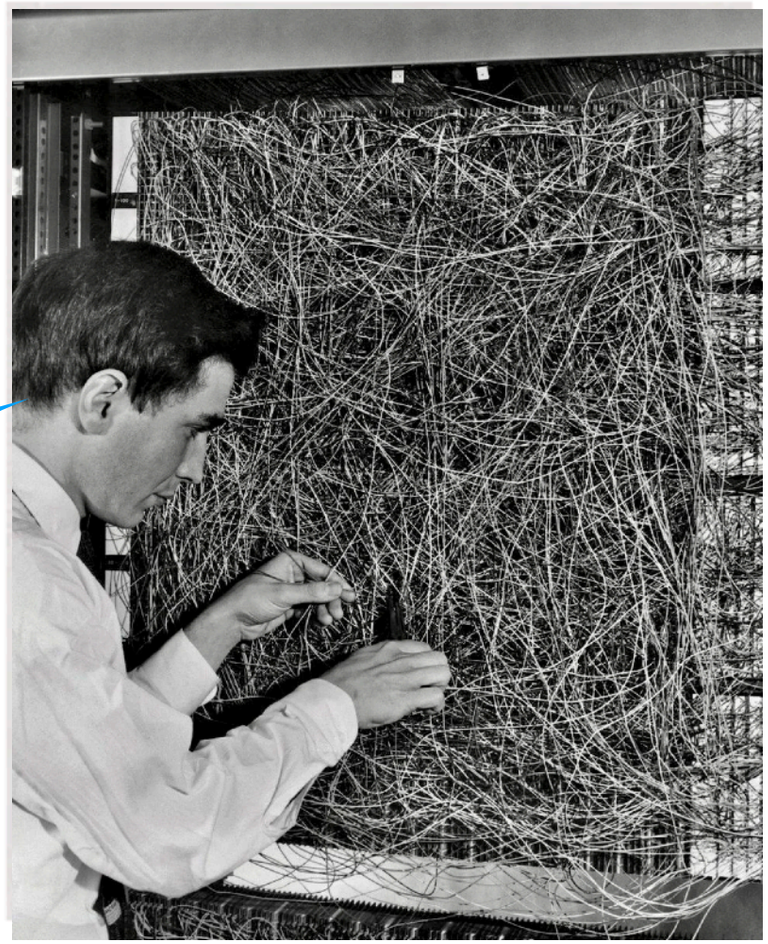
- first implementation of the perceptron algorithm
- the machine was connected to a camera that used 20x20 cadmium sulfide photocells to produce a 400-pixel image
- it recognized letter of the alphabet

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

update rule:

$$w_i(t+1) = w_i(t) + \alpha(d_j - y_j(t))x_{j,i},$$

Perceptron may eventually be able to learn, make decision, and translate languages



Rosenblatt, 1957

Mark I Perceptron

- firs
- the
  cac
- it r

NEW NAVY DEVICE LEARNS BY DOING

Psychologist Shows Embryo of Computer Designed to Read and Grow Wiser

WASHINGTON, July 7 (UPI) —The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.

The embryo—the Weather Bureau's $2,000,000 "704" computer—learned to differentiate between right and left after fifty attempts in the Navy's demonstration for newsmen.

The service said it would use this principle to build the first of its Perceptron thinking machines that will be able to read and write. It is expected to be finished in about a year at a cost of $100,000.

Dr. Frank Rosenblatt, designer of the Perceptron, conducted the demonstration. He said the machine would be the first device to think as the human brain. As do human beings, Perceptron will make mistakes at first, but will grow wiser as it gains experience, he said.

Dr. Rosenblatt, a research psychologist at the Cornell Aeronautical Laboratory, Buffalo, said Perceptrons might be fired to the planets as mechanical space explorers.

Without Human Controls

The Navy said the perceptron would be the first non-living mechanism "capable of receiving, recognizing and identifying its surroundings without any human training or control."

The "brain" is designed to remember images and information it has perceived itself. Ordinary computers remember only what is fed into them on punch cards or magnetic tape.

Later Perceptrons will be able to recognize people and call out their names and instantly translate speech in one language to speech or writing in another language, it was predicted.

Mr. Rosenblatt said in principle it would be possible to build brains that could reproduce themselves on an assembly line and which would be conscious of their existence.

1958 New York Times...

In today's demonstration, the "704" was fed two cards, one with squares marked on the left side and the other with squares on the right side.

Learns by Doing

In the first fifty trials, the machine made no distinction between them. It then started registering a "Q" for the left squares and "O" for the right squares.

Dr. Rosenblatt said he could explain why the machine learned only in highly technical terms. But he said the computer had undergone a "self-induced change in the wiring diagram."

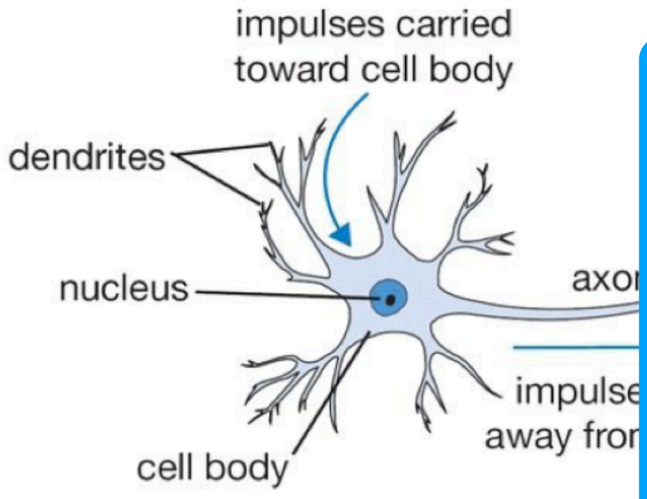
The first Perceptron will have about 1,000 electronic "association cells" receiving electrical impulses from an eye-like scanning device with 400 photo-cells. The human brain has 10,000,000,000 responsive cells, including 100,000,000 connections with the eyes.

Rosenblatt, 1957

Perception = 1 Neuron

The neuron

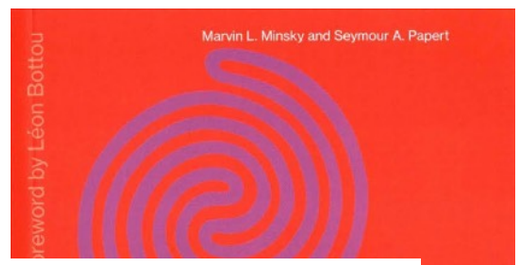
Inspired by neuroscience and human brain, but resemblances do not go too far



"Neural networks copy the human brain." I cringe every time I read something like this the press. It is wrong in multiple ways.
First, neural nets are loosely *inspired* by some aspects of the brain, just as airplanes are loosely inspired by birds.
Second the Inspiration doesn't come from the human brain. It comes from *any* animal brain: monkey, cat, rat, mouse, bird, fish, fruit fly, aplysia sea slug, all the way down to caenorhabditis elegans, the 1mm-long roundworm whose brain has exactly 302 neurons.

Yann LeCun, Facebook IA

"Perceptrons have been widely publicised as
'pattern recognition' or 'learning machines' and
as
bo
vo
is



AI winter

12 languages

Article Talk | Read Edit View history Tools

From Wikipedia, the free encyclopedia

In the history of artificial intelligence, an **AI winter** is a period of reduced funding and interest in artificial intelligence research.[1] The field has experienced several hype cycles, followed by disappointment and criticism, followed by funding cuts, followed by renewed interest years or even decades later.

The term first appeared in 1984 as the topic of a public debate at the annual meeting of AAAI (then called the "American Association of Artificial Intelligence").[2] Roger Schank and Marvin Minsky—two leading AI researchers who experienced the "winter" of the 1970s—warned the business community that enthusiasm for AI had spiraled out of control in the 1980s and that disappointment would certainly follow. They described a chain reaction, similar to a "nuclear winter", that would begin with pessimism in the AI community, followed by pessimism in the press, followed by a severe cutback in funding, followed by the end of serious research.[2] Three years later the billion-dollar AI industry began to collapse.

There were two major winters approximately 1974–1980 and 1987–2000,[3] and several smaller episodes, including the following:

Part of a series on
Artificial intelligence

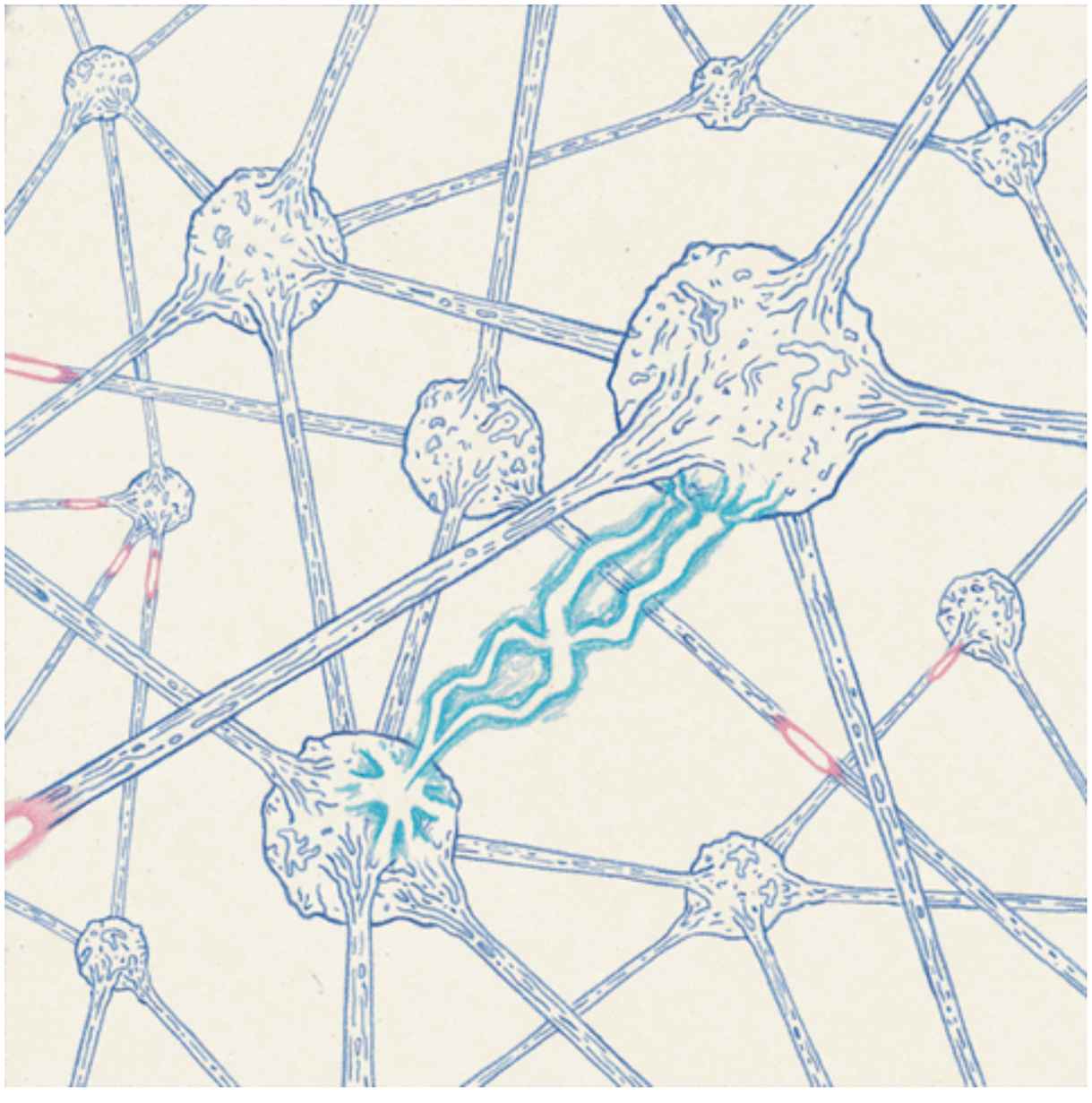
Major goals	[show]
Approaches	[show]
Applications	[show]
Philosophy	[show]
History	[hide]
Timeline · Progress · **AI winter** · AI boom	
Glossary	[show]

V • T • E

Marvin Minsky Seymour Papert, (1969).

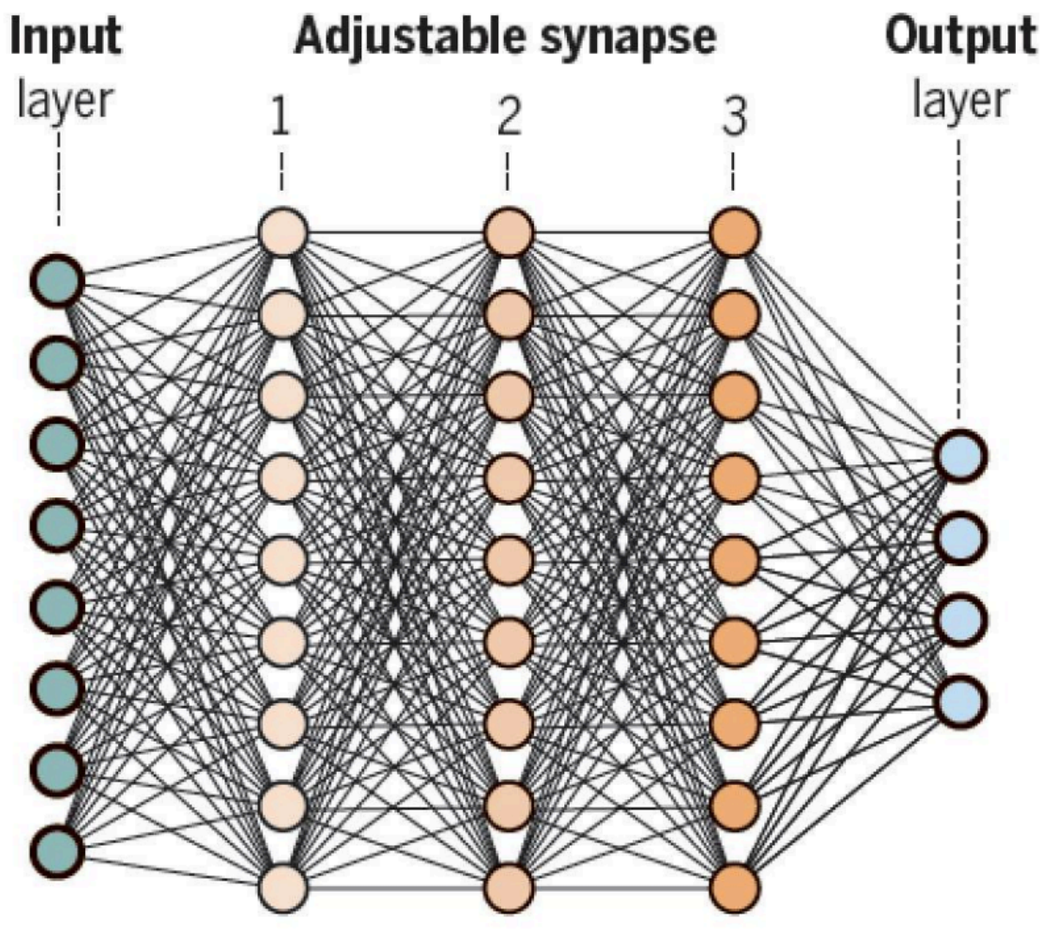
neural nets

Complex neural network



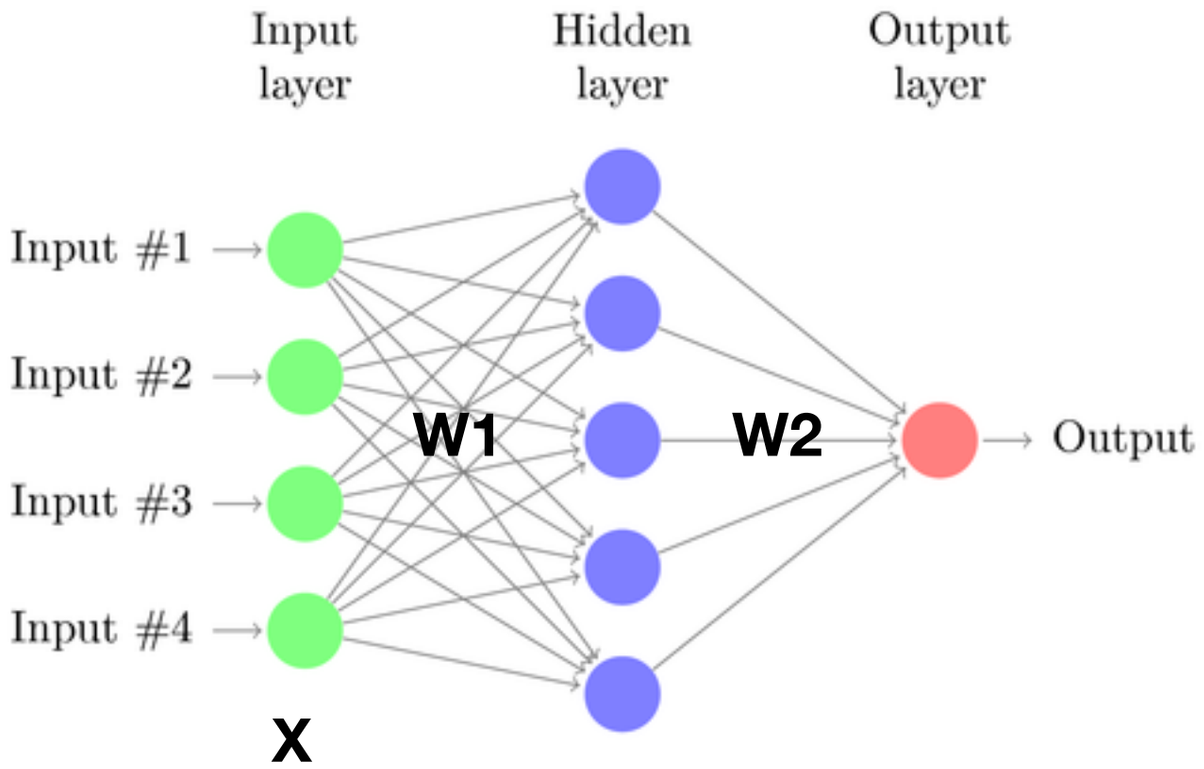
Why neural nets?

Complex neural network



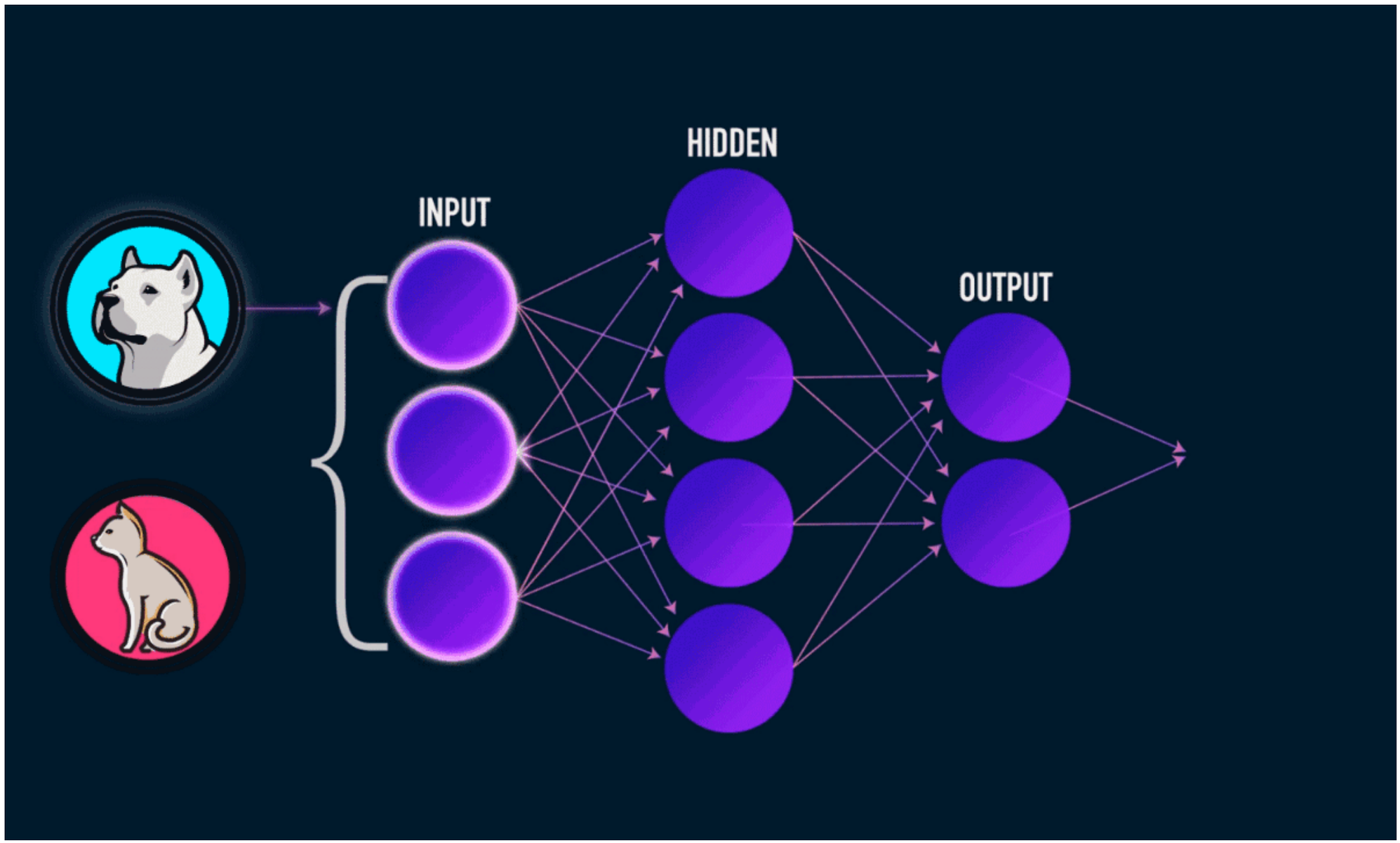
Informed AI network

2-layered networks



$$\hat{y} = \sigma_2 \left(\sum_{i=1}^{n} W_2^i \sigma_1 \left(\sum_{j=1}^{D} W_1^{ij} x_j + b_1^i \right) + b_2 \right)$$

2-layered networks



Universal Approximation Theorem: Fix a continuous function $\sigma : \mathbb{R} \to \mathbb{R}$ (activation function) and positive integers d, D. The function σ is not a polynomial if and only if, for every continuous function $f : \mathbb{R}^d \to \mathbb{R}^D$ (target function), every compact subset K of $\mathbb{R}^d$, and every $\epsilon > 0$ there exists a continuous function $f_\epsilon : \mathbb{R}^d \to \mathbb{R}^D$ (the layer output) with representation

$$f_\epsilon = W_2 \circ \sigma \circ W_1,$$

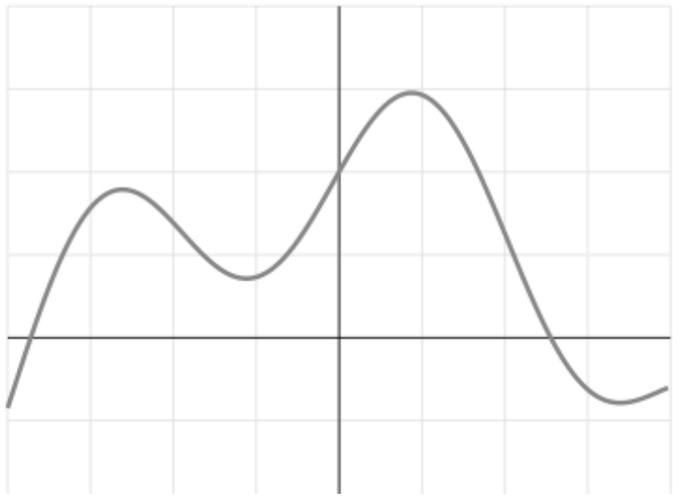
where W_2, W_1 are composable affine maps and $\circ$ denotes component-wise composition, such that the approximation bound

$$\sup_{x \in K} \|f(x) - f_\epsilon(x)\| < \varepsilon$$

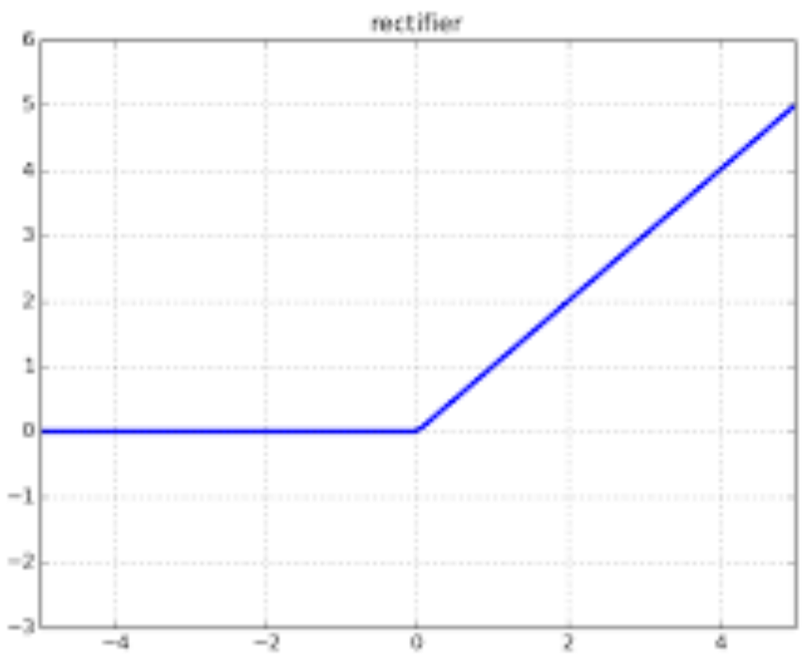
holds for any ϵ arbitrarily small (distance from f to f_ϵ can be infinitely small).

Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions



relu(x) = x if x>0 & 0 otherwise



Universal Approximation Theorem: Fix a continuous function $\sigma : \mathbb{R} \to \mathbb{R}$ (activation function) and positive integers d, D. The function σ is not a polynomial if and only if, for every continuous function $f : \mathbb{R}^d \to \mathbb{R}^D$ (target function), every compact subset K of $\mathbb{R}^d$, and every $\epsilon > 0$ there exists a continuous function $f_\epsilon : \mathbb{R}^d \to \mathbb{R}^D$ (the layer output) with representation

$$f_\epsilon = W_2 \circ \sigma \circ W_1,$$

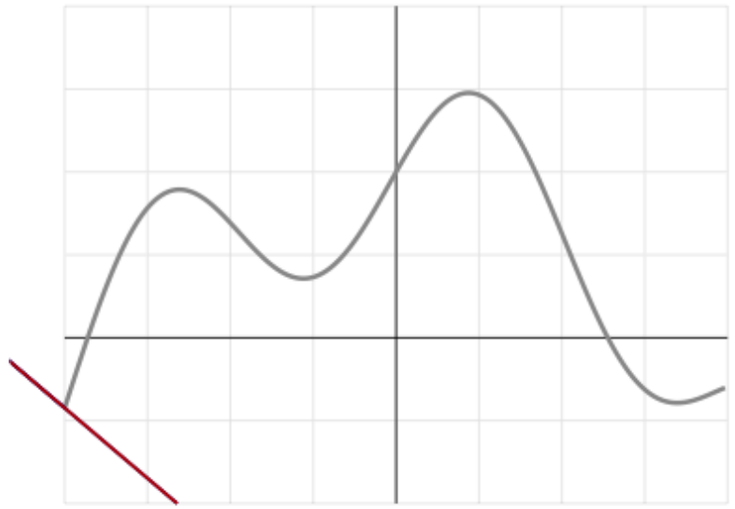
where W_2, W_1 are composable affine maps and $\circ$ denotes component-wise composition, such that the approximation bound

$$\sup_{x \in K} \|f(x) - f_\epsilon(x)\| < \varepsilon$$

holds for any ϵ arbitrarily small (distance from f to f_ϵ can be infinitely small).

Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions



Universal Approximation Theorem: Fix a continuous function $\sigma : \mathbb{R} \to \mathbb{R}$ (activation function) and positive integers d, D. The function σ is not a polynomial if and only if, for every continuous function $f : \mathbb{R}^d \to \mathbb{R}^D$ (target function), every compact subset K of $\mathbb{R}^d$, and every $\epsilon > 0$ there exists a continuous function $f_\epsilon : \mathbb{R}^d \to \mathbb{R}^D$ (the layer output) with representation

$$f_\epsilon = W_2 \circ \sigma \circ W_1,$$

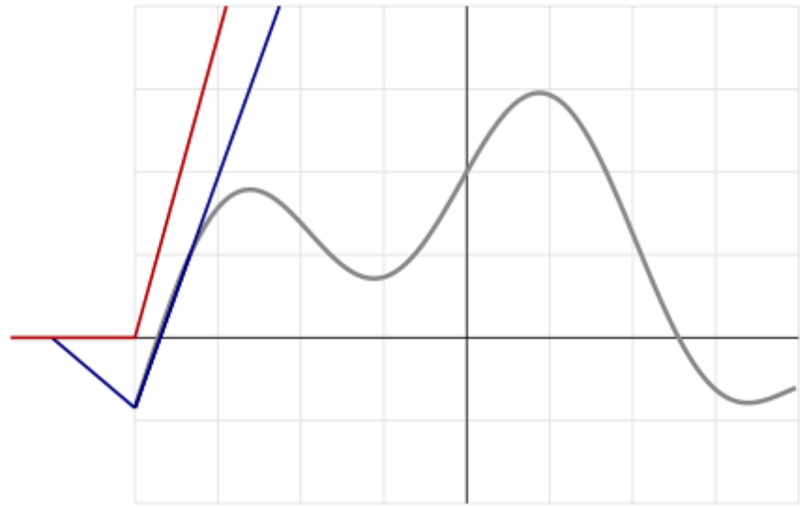
where W_2, W_1 are composable affine maps and $\circ$ denotes component-wise composition, such that the approximation bound

$$\sup_{x \in K} \|f(x) - f_\epsilon(x)\| < \varepsilon$$

holds for any ϵ arbitrarily small (distance from f to f_ϵ can be infinitely small).

Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions



Universal Approximation Theorem: Fix a continuous function $\sigma : \mathbb{R} \to \mathbb{R}$ (activation function) and positive integers d, D. The function σ is not a polynomial if and only if, for every continuous function $f : \mathbb{R}^d \to \mathbb{R}^D$ (target function), every compact subset K of $\mathbb{R}^d$, and every $\epsilon > 0$ there exists a continuous function $f_\epsilon : \mathbb{R}^d \to \mathbb{R}^D$ (the layer output) with representation

$$f_\epsilon = W_2 \circ \sigma \circ W_1,$$

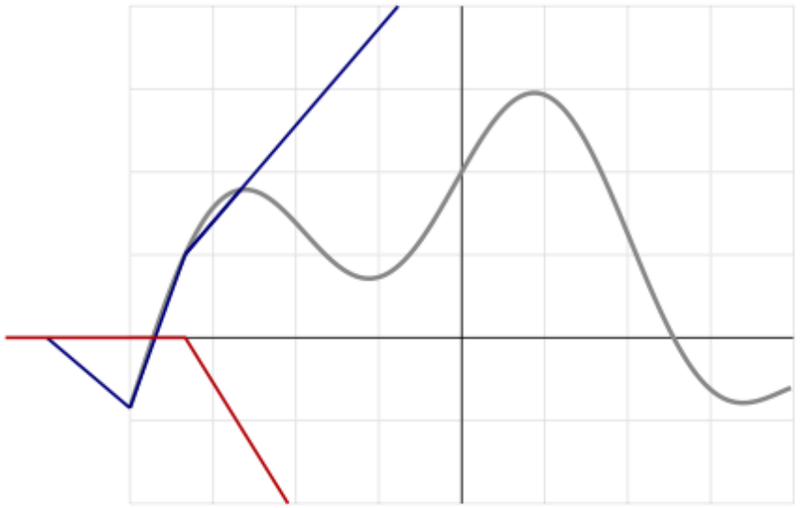
where W_2, W_1 are composable affine maps and $\circ$ denotes component-wise composition, such that the approximation bound

$$\sup_{x \in K} \|f(x) - f_\epsilon(x)\| < \varepsilon$$

holds for any ϵ arbitrarily small (distance from f to f_ϵ can be infinitely small).

Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions



Universal Approximation Theorem: Fix a continuous function $\sigma : \mathbb{R} \to \mathbb{R}$ (activation function) and positive integers d, D. The function σ is not a polynomial if and only if, for every continuous function $f : \mathbb{R}^d \to \mathbb{R}^D$ (target function), every compact subset K of $\mathbb{R}^d$, and every $\epsilon > 0$ there exists a continuous function $f_\epsilon : \mathbb{R}^d \to \mathbb{R}^D$ (the layer output) with representation

$$f_\epsilon = W_2 \circ \sigma \circ W_1,$$

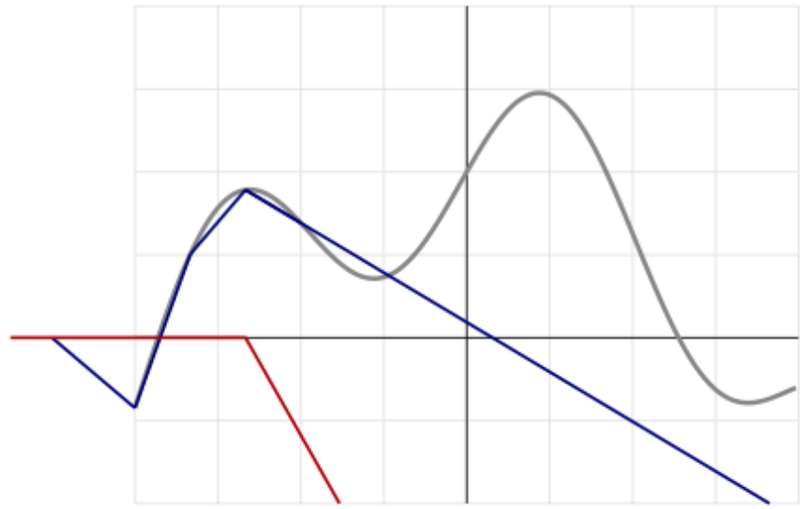
where W_2, W_1 are composable affine maps and $\circ$ denotes component-wise composition, such that the approximation bound

$$\sup_{x \in K} \|f(x) - f_\epsilon(x)\| < \varepsilon$$

holds for any ϵ arbitrarily small (distance from f to f_ϵ can be infinitely small).

Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions



Universal Approximation Theorem: Fix a continuous function $\sigma : \mathbb{R} \to \mathbb{R}$ (activation function) and positive integers d, D. The function σ is not a polynomial if and only if, for every continuous function $f : \mathbb{R}^d \to \mathbb{R}^D$ (target function), every compact subset K of $\mathbb{R}^d$, and every $\epsilon > 0$ there exists a continuous function $f_\epsilon : \mathbb{R}^d \to \mathbb{R}^D$ (the layer output) with representation

$$f_\epsilon = W_2 \circ \sigma \circ W_1,$$

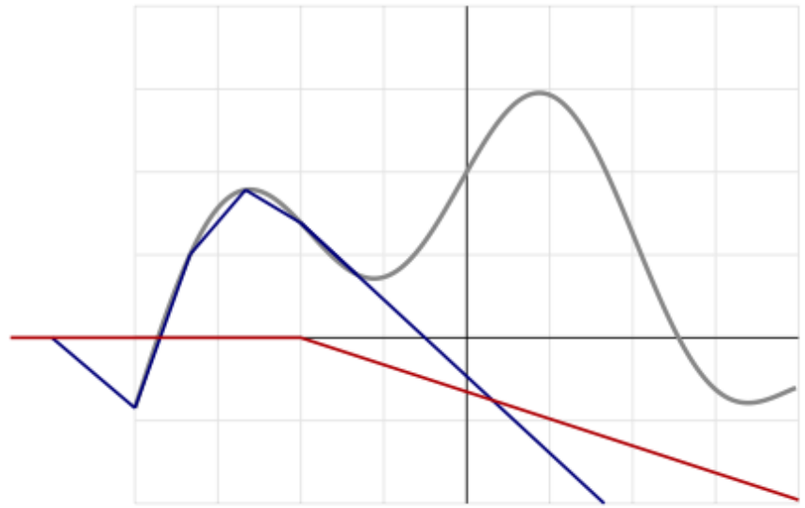
where W_2, W_1 are composable affine maps and $\circ$ denotes component-wise composition, such that the approximation bound

$$\sup_{x \in K} \|f(x) - f_\epsilon(x)\| < \varepsilon$$

holds for any ϵ arbitrarily small (distance from f to f_ϵ can be infinitely small).

Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions



Universal Approximation Theorem: Fix a continuous function $\sigma : \mathbb{R} \to \mathbb{R}$ (activation function) and positive integers d, D. The function σ is not a polynomial if and only if, for every continuous function $f : \mathbb{R}^d \to \mathbb{R}^D$ (target function), every compact subset K of $\mathbb{R}^d$, and every $\epsilon > 0$ there exists a continuous function $f_\epsilon : \mathbb{R}^d \to \mathbb{R}^D$ (the layer output) with representation

$$f_\epsilon = W_2 \circ \sigma \circ W_1,$$

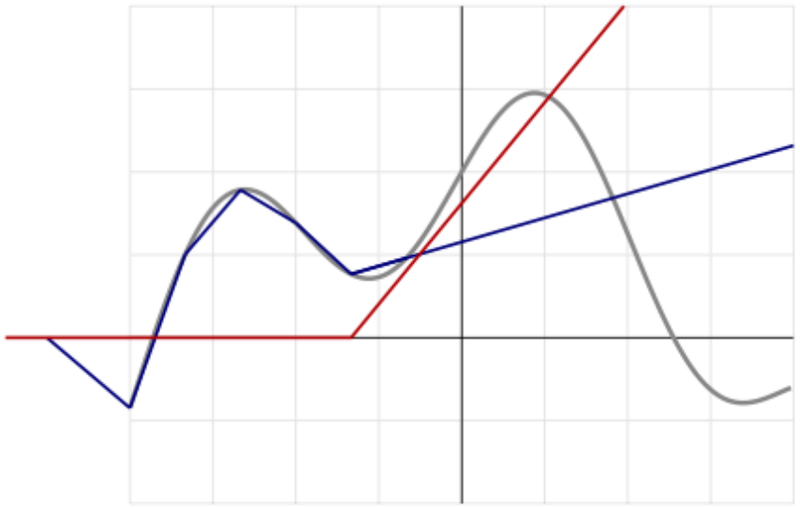
where W_2, W_1 are composable affine maps and $\circ$ denotes component-wise composition, such that the approximation bound

$$\sup_{x \in K} \|f(x) - f_\epsilon(x)\| < \varepsilon$$

holds for any ϵ arbitrarily small (distance from f to f_ϵ can be infinitely small).

Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions



Universal Approximation Theorem: Fix a continuous function $\sigma : \mathbb{R} \to \mathbb{R}$ (activation function) and positive integers d, D. The function σ is not a polynomial if and only if, for every continuous function $f : \mathbb{R}^d \to \mathbb{R}^D$ (target function), every compact subset K of $\mathbb{R}^d$, and every $\epsilon > 0$ there exists a continuous function $f_\epsilon : \mathbb{R}^d \to \mathbb{R}^D$ (the layer output) with representation

$$f_\epsilon = W_2 \circ \sigma \circ W_1,$$

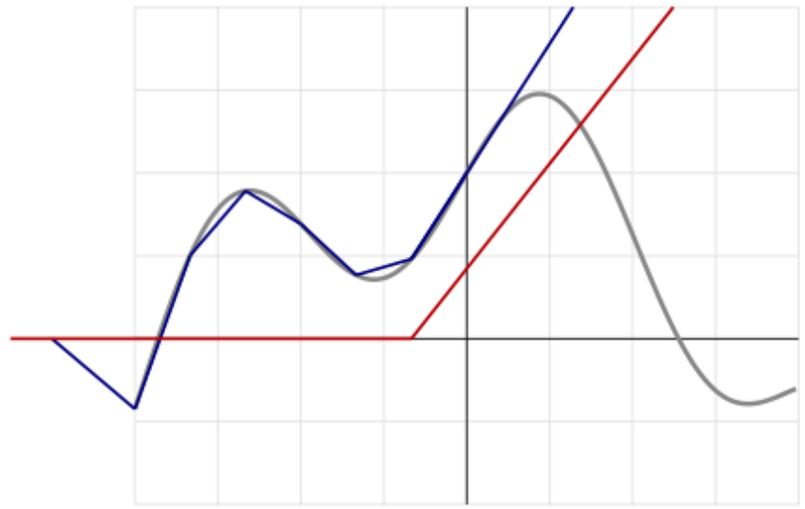
where W_2, W_1 are composable affine maps and $\circ$ denotes component-wise composition, such that the approximation bound

$$\sup_{x \in K} \|f(x) - f_\epsilon(x)\| < \varepsilon$$

holds for any ϵ arbitrarily small (distance from f to f_ϵ can be infinitely small).

Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions



Universal Approximation Theorem: Fix a continuous function $\sigma : \mathbb{R} \to \mathbb{R}$ (activation function) and positive integers d, D. The function σ is not a polynomial if and only if, for every continuous function $f : \mathbb{R}^d \to \mathbb{R}^D$ (target function), every compact subset K of $\mathbb{R}^d$, and every $\epsilon > 0$ there exists a continuous function $f_\epsilon : \mathbb{R}^d \to \mathbb{R}^D$ (the layer output) with representation

$$f_\epsilon = W_2 \circ \sigma \circ W_1,$$

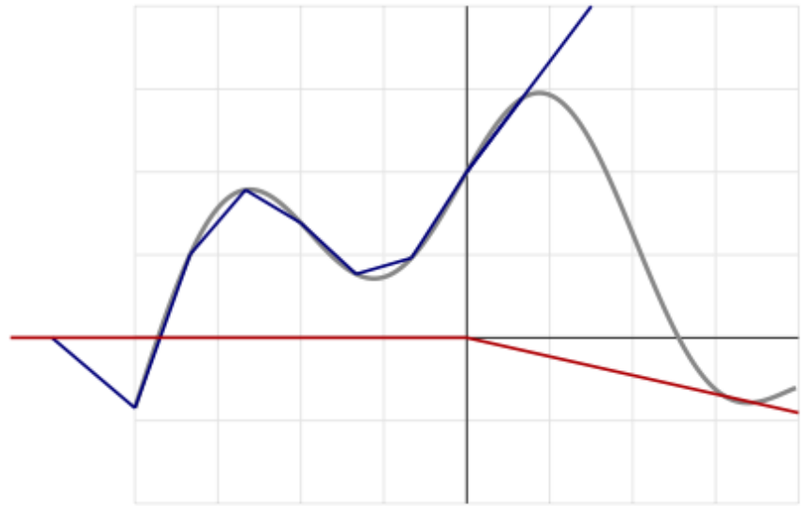
where W_2, W_1 are composable affine maps and $\circ$ denotes component-wise composition, such that the approximation bound

$$\sup_{x \in K} \|f(x) - f_\epsilon(x)\| < \varepsilon$$

holds for any ϵ arbitrarily small (distance from f to f_ϵ can be infinitely small).

Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions



Universal Approximation Theorem: Fix a continuous function $\sigma : \mathbb{R} \to \mathbb{R}$ (activation function) and positive integers d, D. The function σ is not a polynomial if and only if, for every continuous function $f : \mathbb{R}^d \to \mathbb{R}^D$ (target function), every compact subset K of $\mathbb{R}^d$, and every $\epsilon > 0$ there exists a continuous function $f_\epsilon : \mathbb{R}^d \to \mathbb{R}^D$ (the layer output) with representation

$$f_\epsilon = W_2 \circ \sigma \circ W_1,$$

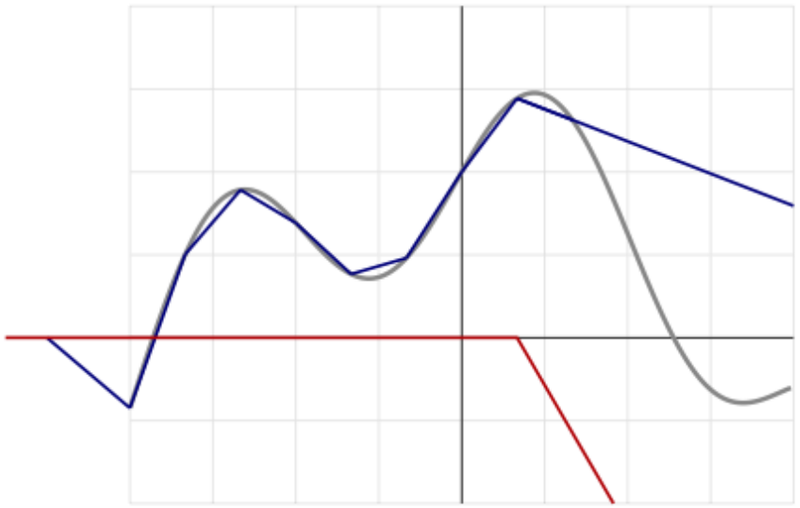
where W_2, W_1 are composable affine maps and $\circ$ denotes component-wise composition, such that the approximation bound

$$\sup_{x \in K} \|f(x) - f_\epsilon(x)\| < \varepsilon$$

holds for any ϵ arbitrarily small (distance from f to f_ϵ can be infinitely small).

Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions



Universal Approximation Theorem: Fix a continuous function $\sigma : \mathbb{R} \to \mathbb{R}$ (activation function) and positive integers d, D. The function σ is not a polynomial if and only if, for every continuous function $f : \mathbb{R}^d \to \mathbb{R}^D$ (target function), every compact subset K of $\mathbb{R}^d$, and every $\epsilon > 0$ there exists a continuous function $f_\epsilon : \mathbb{R}^d \to \mathbb{R}^D$ (the layer output) with representation

$$f_\epsilon = W_2 \circ \sigma \circ W_1,$$

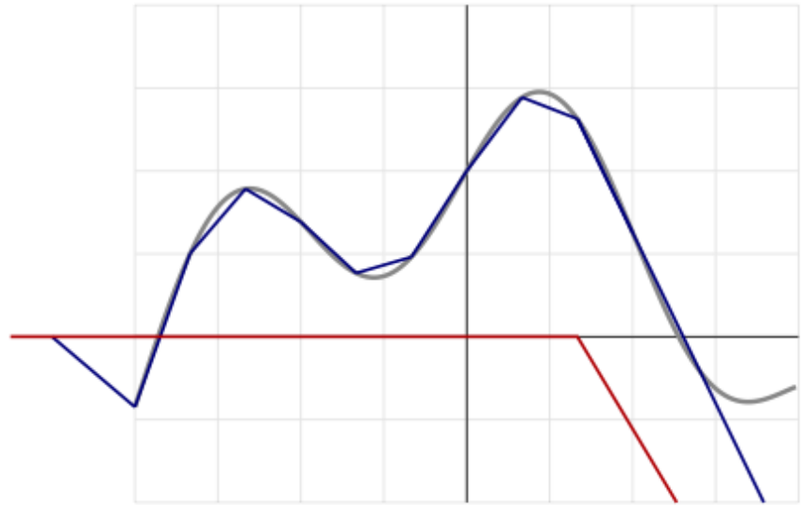
where W_2, W_1 are composable affine maps and $\circ$ denotes component-wise composition, such that the approximation bound

$$\sup_{x \in K} \|f(x) - f_\epsilon(x)\| < \varepsilon$$

holds for any ϵ arbitrarily small (distance from f to f_ϵ can be infinitely small).

Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions



Universal Approximation Theorem: Fix a continuous function $\sigma : \mathbb{R} \to \mathbb{R}$ (activation function) and positive integers d, D. The function σ is not a polynomial if and only if, for every continuous function $f : \mathbb{R}^d \to \mathbb{R}^D$ (target function), every compact subset K of $\mathbb{R}^d$, and every $\epsilon > 0$ there exists a continuous function $f_\epsilon : \mathbb{R}^d \to \mathbb{R}^D$ (the layer output) with representation

$$f_\epsilon = W_2 \circ \sigma \circ W_1,$$

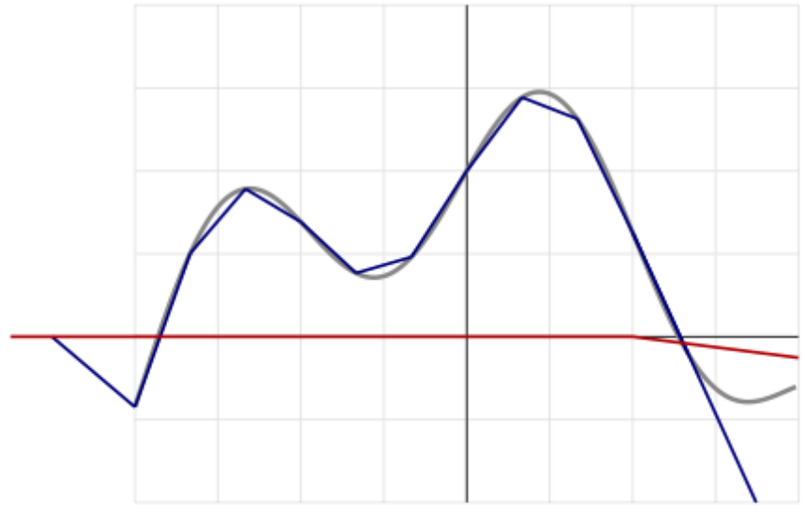
where W_2, W_1 are composable affine maps and $\circ$ denotes component-wise composition, such that the approximation bound

$$\sup_{x \in K} \|f(x) - f_\epsilon(x)\| < \varepsilon$$

holds for any ϵ arbitrarily small (distance from f to f_ϵ can be infinitely small).

Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions



Universal Approximation Theorem: Fix a continuous function $\sigma : \mathbb{R} \to \mathbb{R}$ (activation function) and positive integers d, D. The function σ is not a polynomial if and only if, for every continuous function $f : \mathbb{R}^d \to \mathbb{R}^D$ (target function), every compact subset K of $\mathbb{R}^d$, and every $\epsilon > 0$ there exists a continuous function $f_\epsilon : \mathbb{R}^d \to \mathbb{R}^D$ (the layer output) with representation

$$f_\epsilon = W_2 \circ \sigma \circ W_1,$$

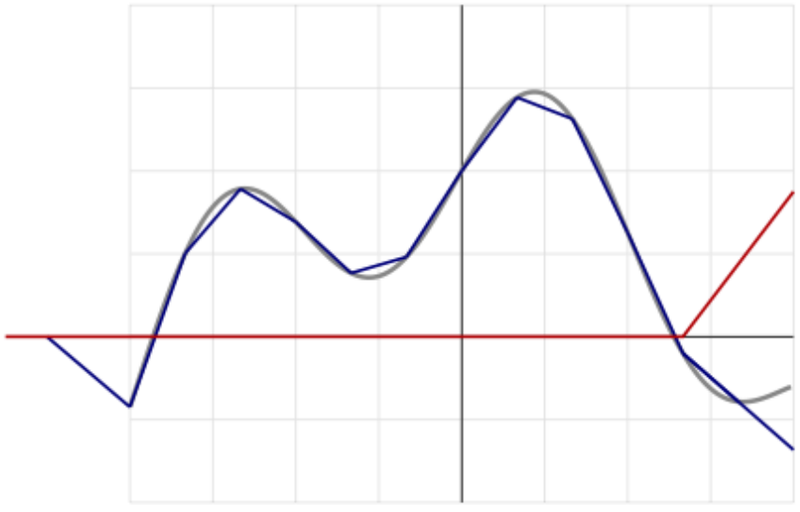
where W_2, W_1 are composable affine maps and $\circ$ denotes component-wise composition, such that the approximation bound

$$\sup_{x \in K} \|f(x) - f_\epsilon(x)\| < \varepsilon$$

holds for any ϵ arbitrarily small (distance from f to f_ϵ can be infinitely small).

Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions



Universal Approximation Theorem: Fix a continuous function $\sigma : \mathbb{R} \to \mathbb{R}$ (activation function) and positive integers d, D. The function σ is not a polynomial if and only if, for every continuous function $f : \mathbb{R}^d \to \mathbb{R}^D$ (target function), every compact subset K of $\mathbb{R}^d$, and every $\epsilon > 0$ there exists a continuous function $f_\epsilon : \mathbb{R}^d \to \mathbb{R}^D$ (the layer output) with representation

$$f_\epsilon = W_2 \circ \sigma \circ W_1,$$

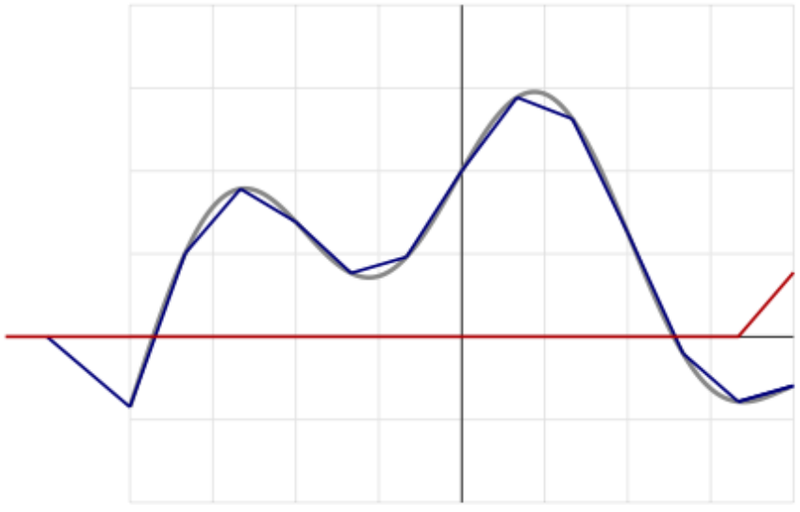
where W_2, W_1 are composable affine maps and $\circ$ denotes component-wise composition, such that the approximation bound

$$\sup_{x \in K} \|f(x) - f_\epsilon(x)\| < \varepsilon$$

holds for any ϵ arbitrarily small (distance from f to f_ϵ can be infinitely small).

Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions



Universal Approximation Theorem: Fix a continuous function $\sigma : \mathbb{R} \to \mathbb{R}$ (activation function) and positive integers d, D. The function σ is not a polynomial if and only if, for every continuous function $f : \mathbb{R}^d \to \mathbb{R}^D$ (target function), every compact subset K of $\mathbb{R}^d$, and every $\epsilon > 0$ there exists a continuous function $f_\epsilon : \mathbb{R}^d \to \mathbb{R}^D$ (the layer output) with representation

$$f_\epsilon = W_2 \circ \sigma \circ W_1,$$

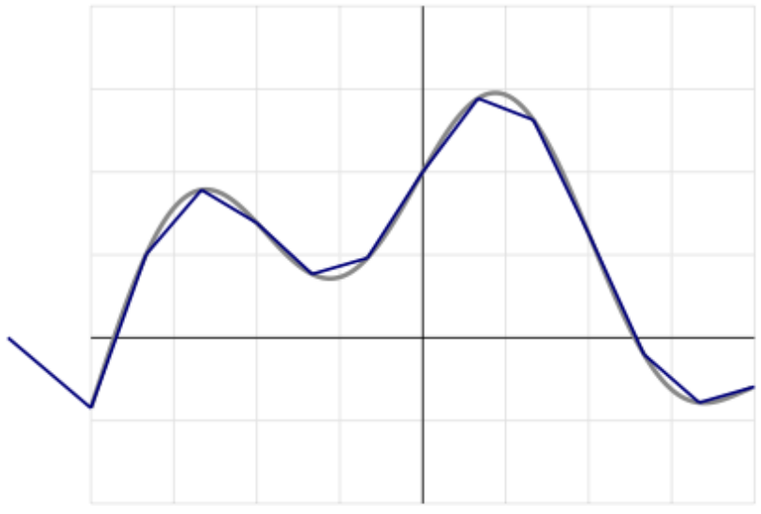
where W_2, W_1 are composable affine maps and $\circ$ denotes component-wise composition, such that the approximation bound

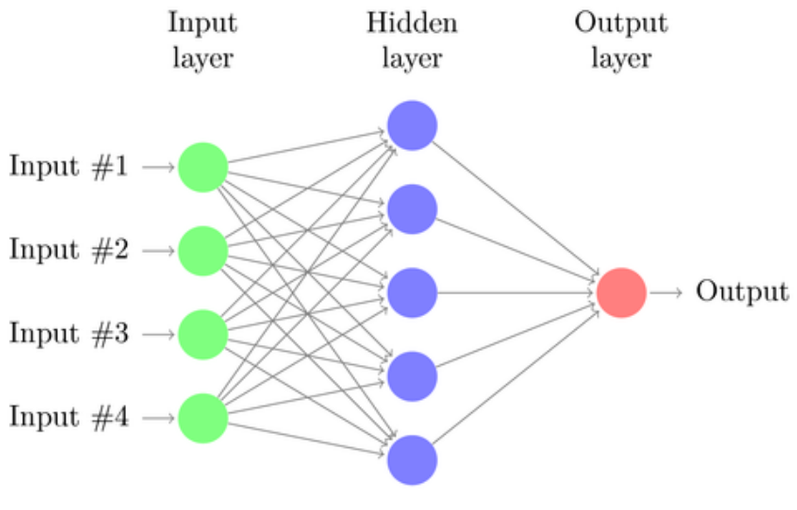
$$\sup_{x \in K} \|f(x) - f_\epsilon(x)\| < \varepsilon$$

holds for any ϵ arbitrarily small (distance from f to f_ϵ can be infinitely small).

Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions

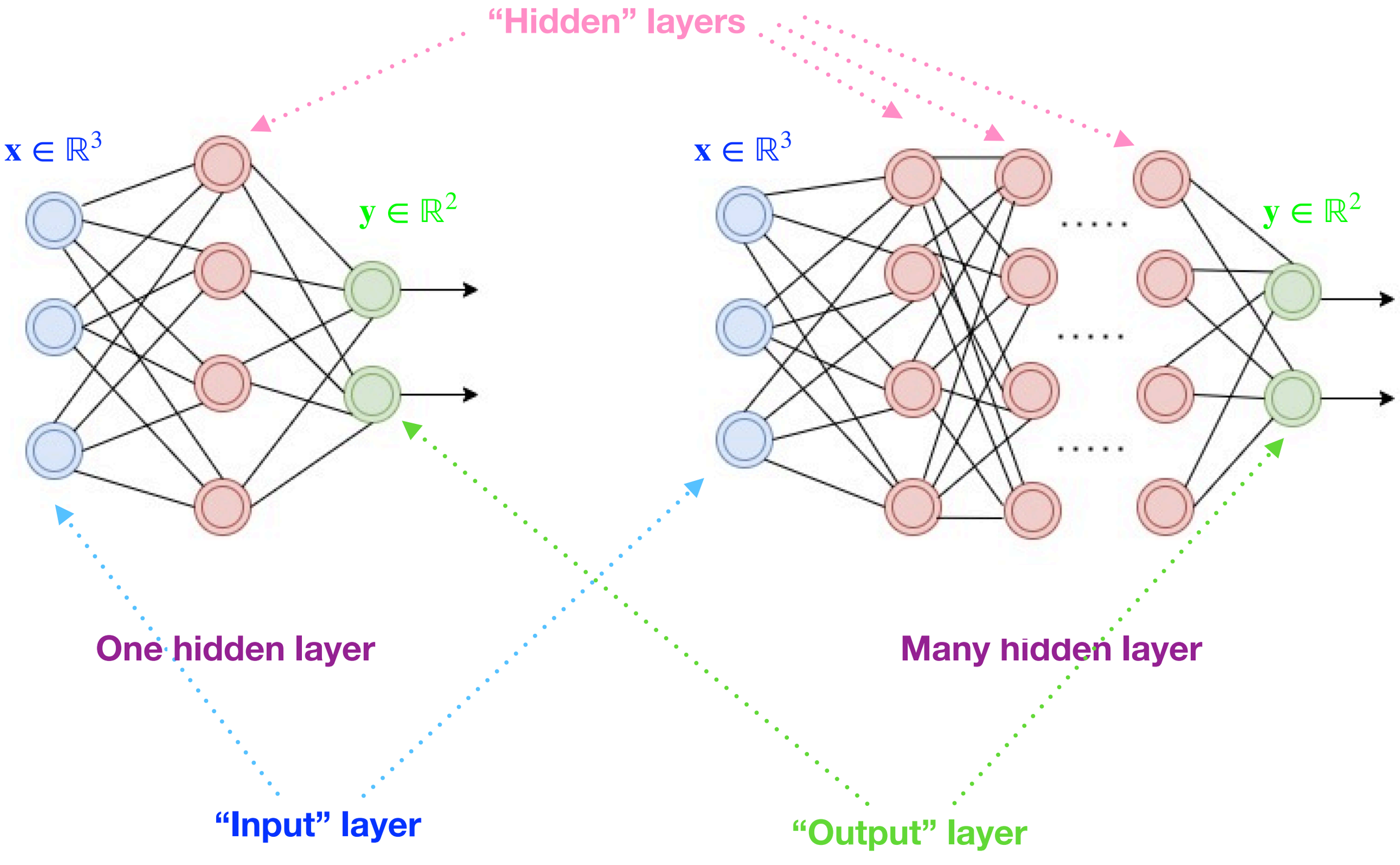




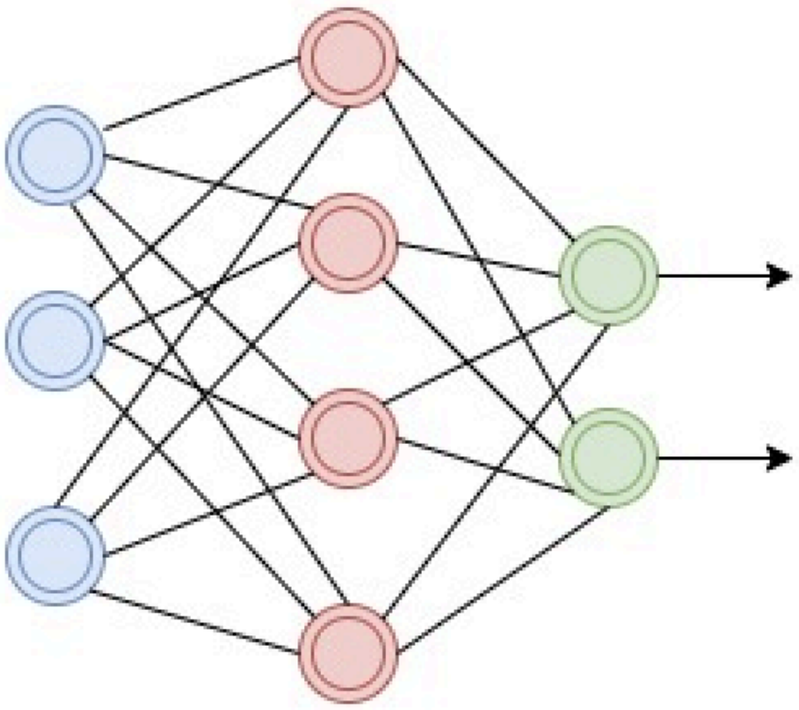
$$Y = \sum_i \alpha_i \mathrm{Relu}(\mathrm{a_i} * \mathrm{x} + \mathrm{b_i})$$

Why deep learning?

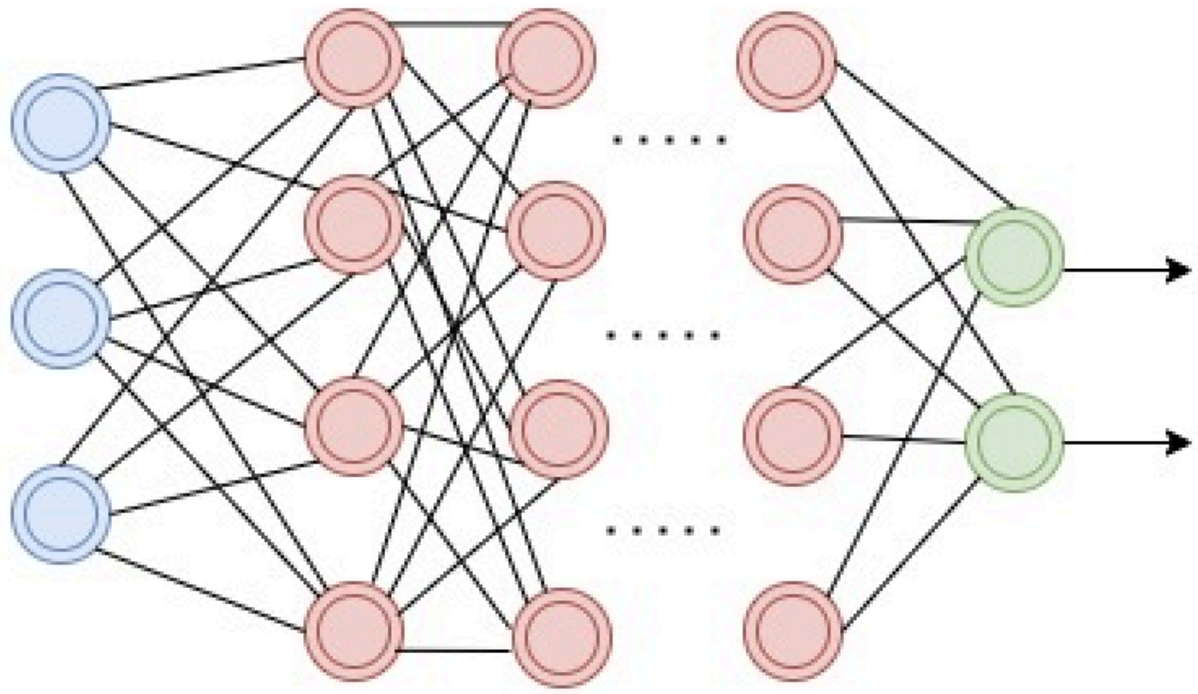
Deep vs Shallow



Deep vs Shallow



$$\hat{y} = \sigma_2\left(W_2 \,.\, \sigma_1\left(W_1\vec{x} + b_1\right) + b_2\right)$$



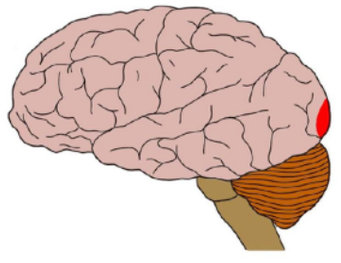
$$\hat{y} = \sigma_L\left(W_L \,.\, \sigma_{L-1}(W_{L-1}(\ldots.\sigma_1\left(W_1\vec{x} + b_1\right) + b_2\right)$$

Why deep?

(i) With the same number of nodes, one can represent more complex function with deep networks

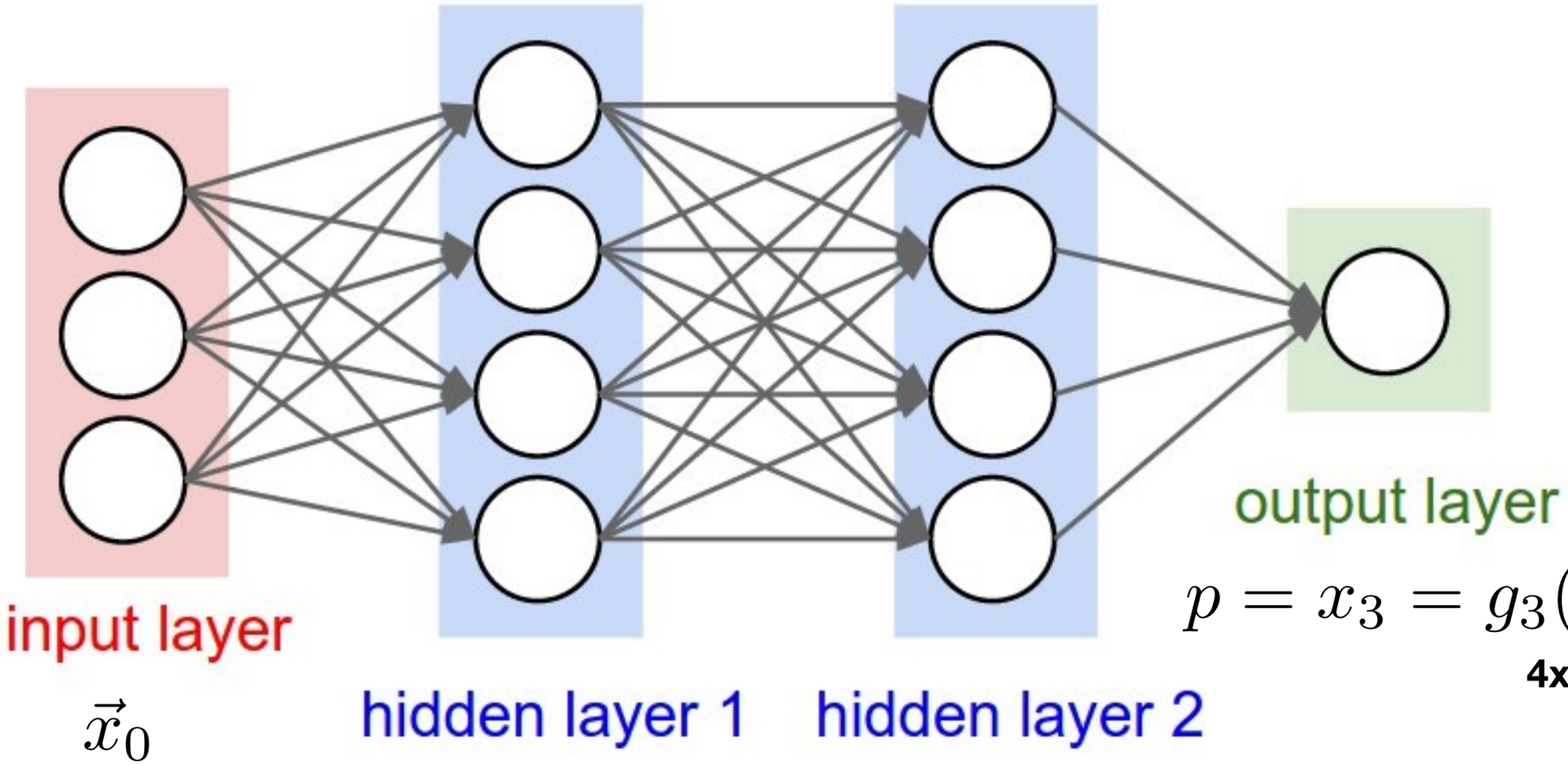
(ii) Many data are actually hierarchical…
…just think of the classification of animal species!

(iii) inspiration from the visual cortex (convnet, see next lecture)



(iv) seems to do some weird uncanny magic that somehow prevents overfiting

Feed-forward Neural networks



$$p = x_3 = g_3(W_3\vec{x}_2)$$

4x1 matrix

$\vec{x}_0$

$$\vec{x}_1 = g_1(W_1\vec{x}_0) \quad \vec{x}_2 = g_2(W_2\vec{x}_1)$$

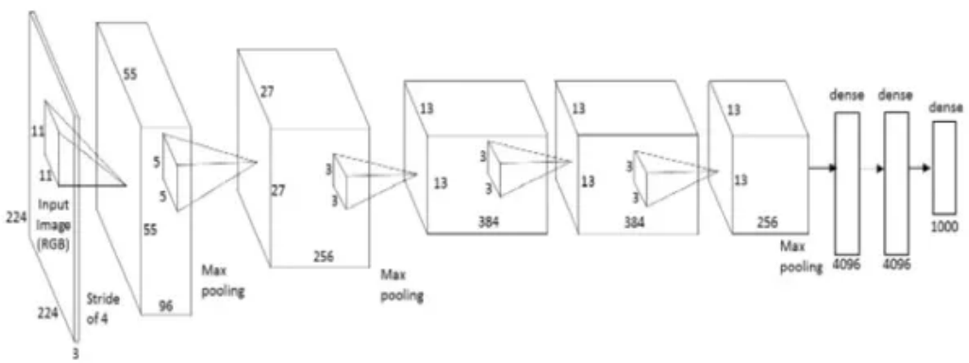
4x3 matrix **4x4 matrix**

$$p = f(\vec{x}_0) = g_3(W_3\ g_2(W_2\ g_1(W_1\vec{x}_0)))$$

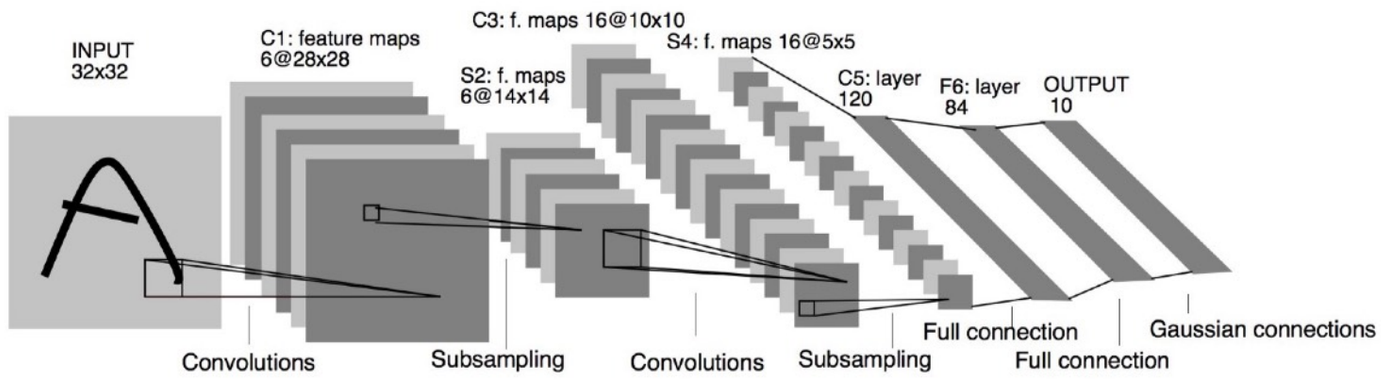
W matrices are called the « *weights* »
The functions g_n ( ) are called « *activation functions* »

Modern neural nets

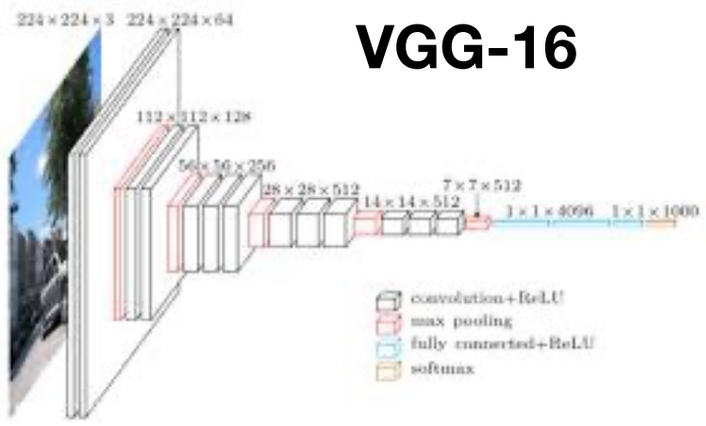
Alexnet



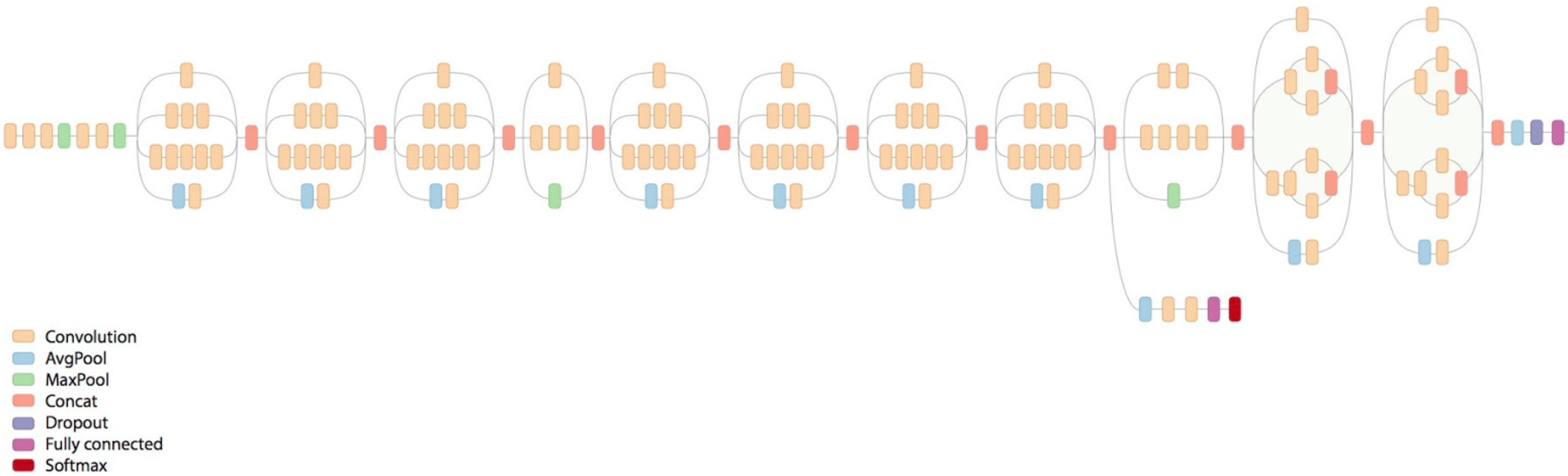
Lenet



VGG-16



Modern neural nets



inception res-net

Modern neural nets

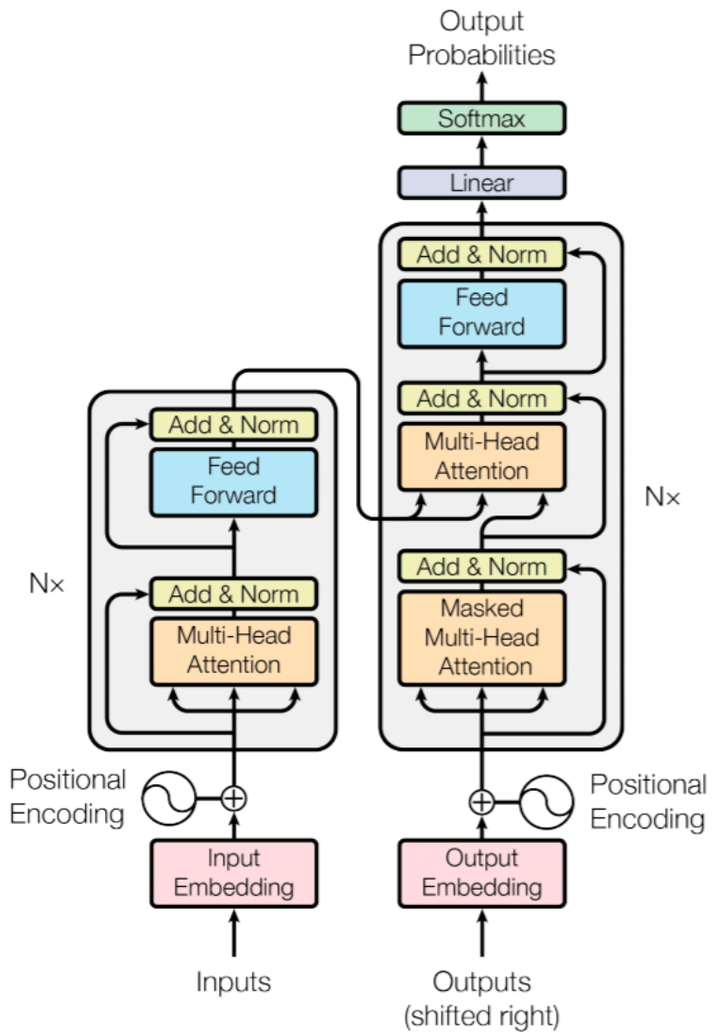


Figure 1: The Transformer - model architecture.

Transformers

How do we minimise the empirical risk for the neural network?

Optimization

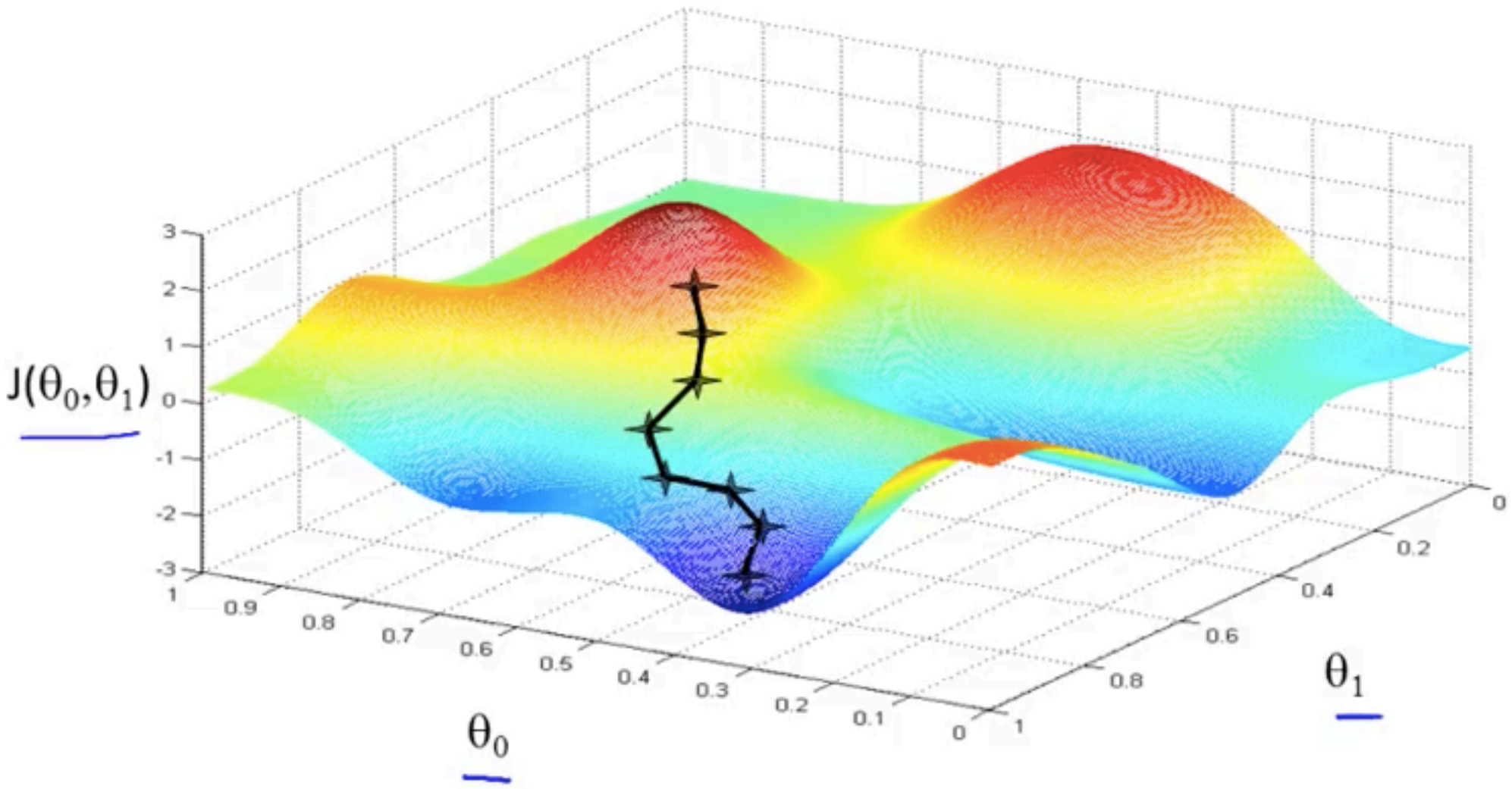
Optimization

Follow the slope!

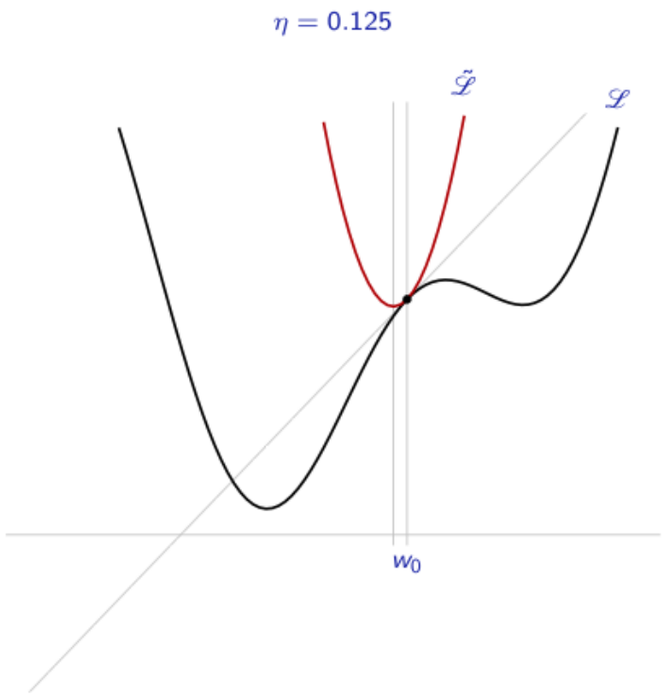
Minimising the cost function by gradient descent

$$\theta^{t+1} = \theta^t - \eta \nabla \mathscr{R}(\theta^t)$$

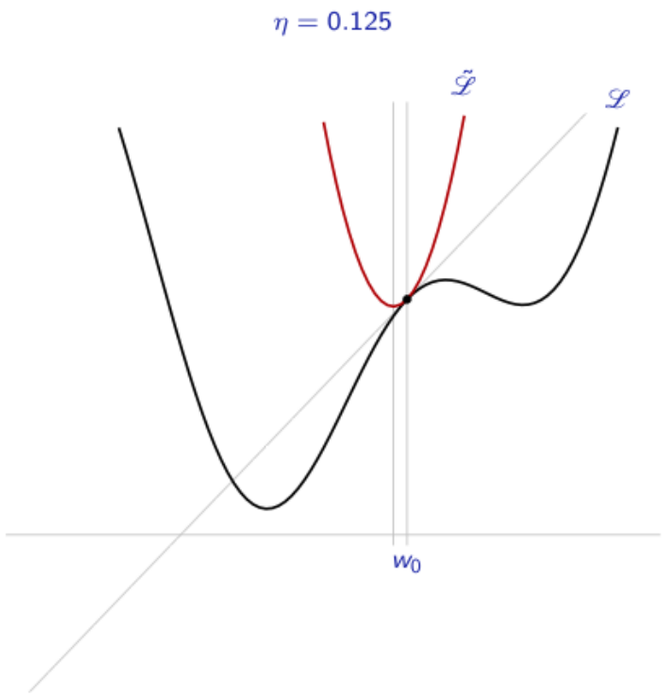
If eta small enough, converges to a (possible local) minima



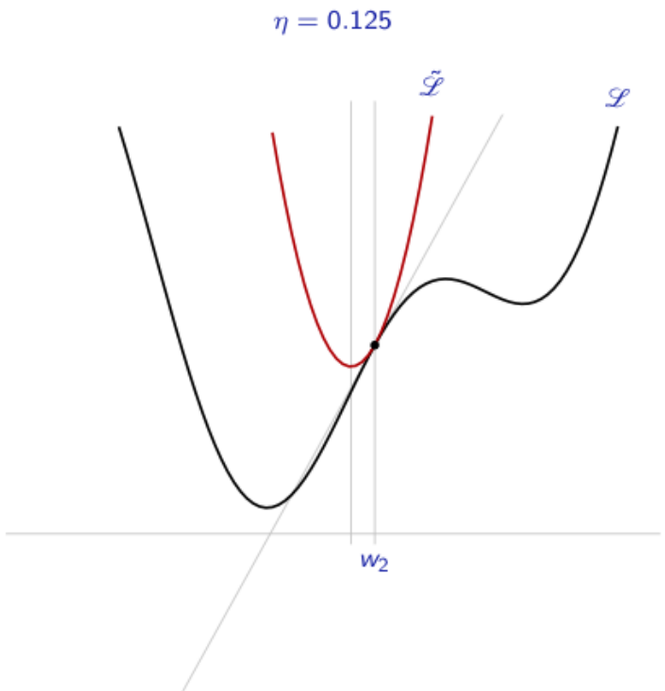
Gradient descent



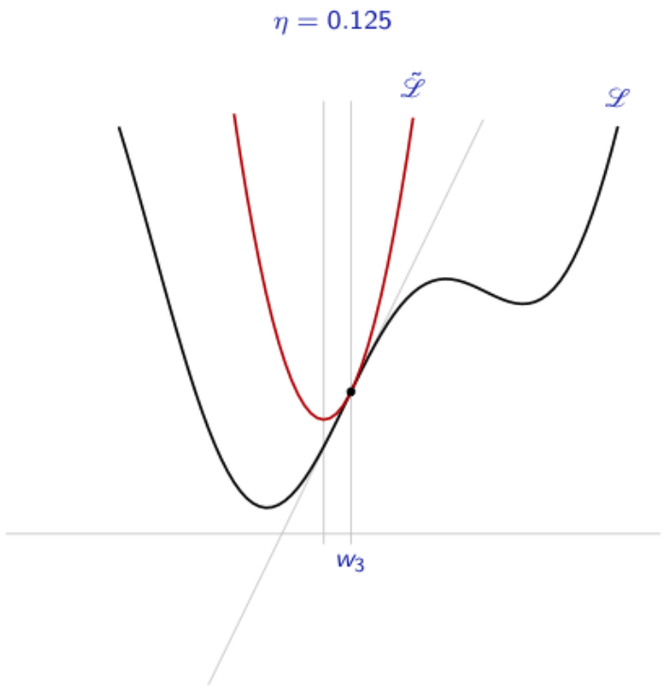
Gradient descent



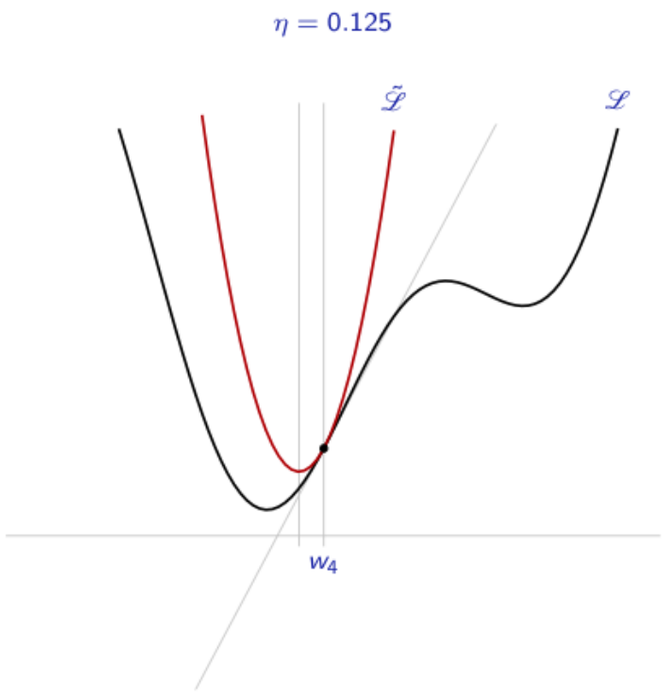
Gradient descent



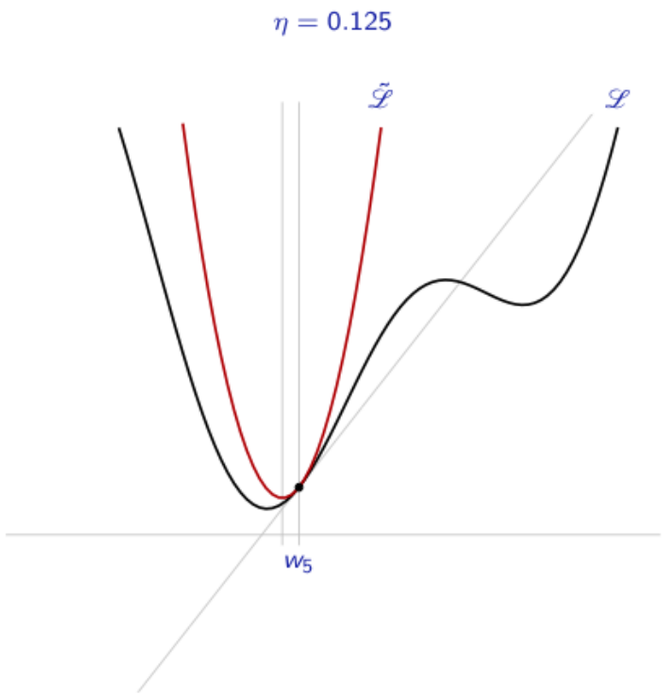
Gradient descent



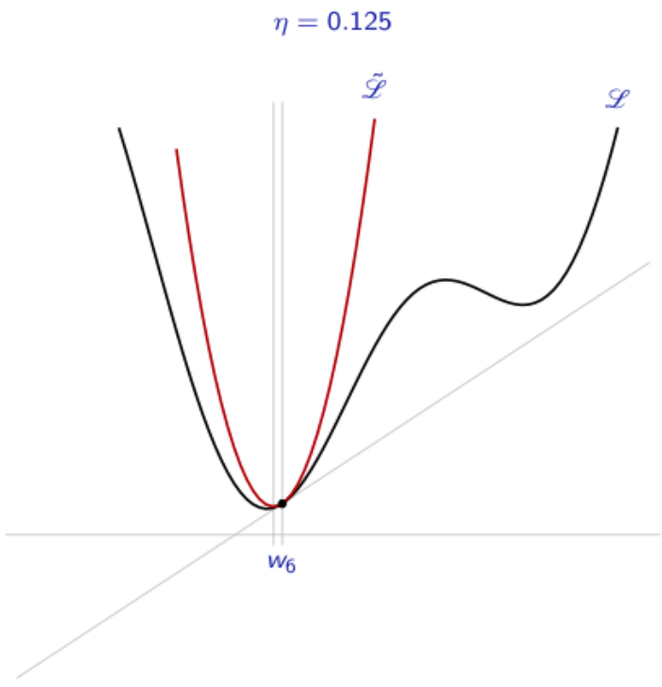
Gradient descent



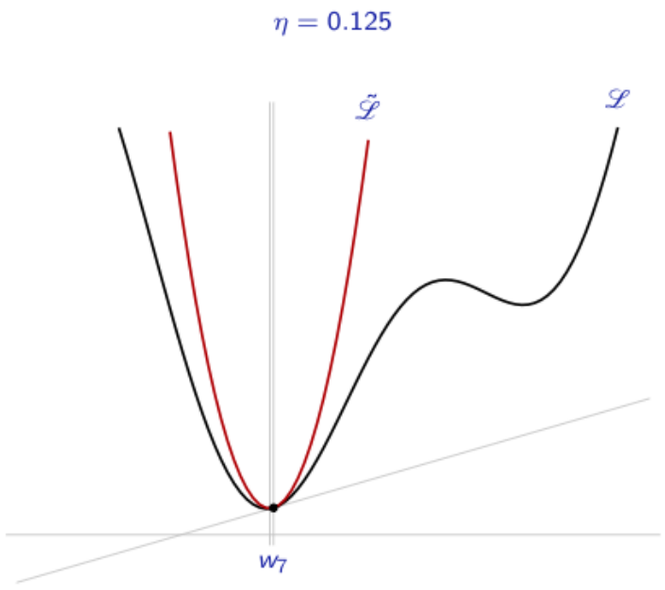
Gradient descent



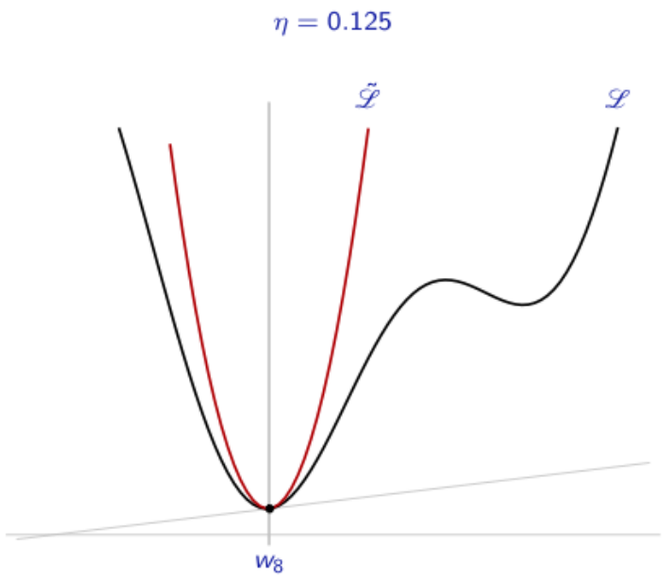
Gradient descent



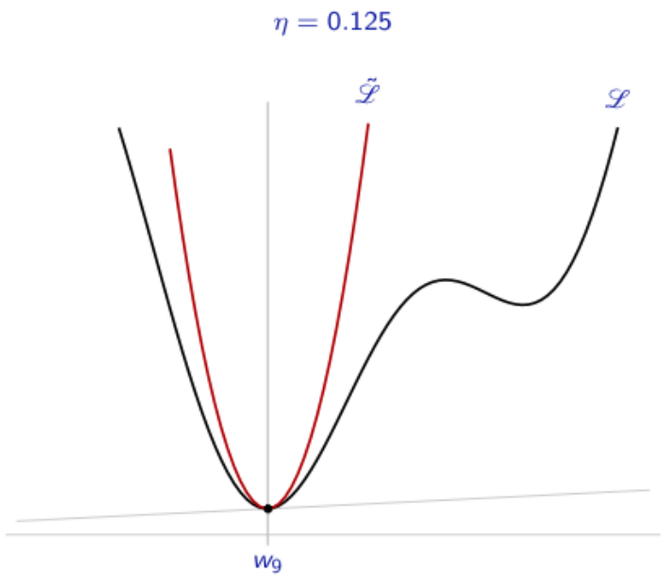
Gradient descent



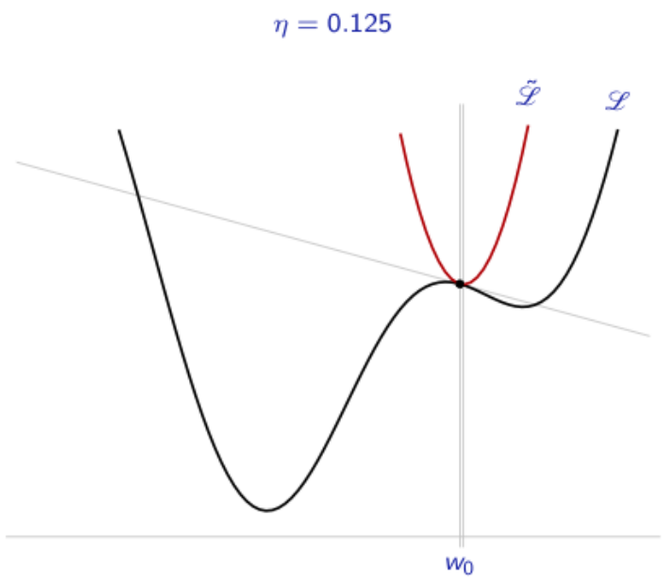
Gradient descent



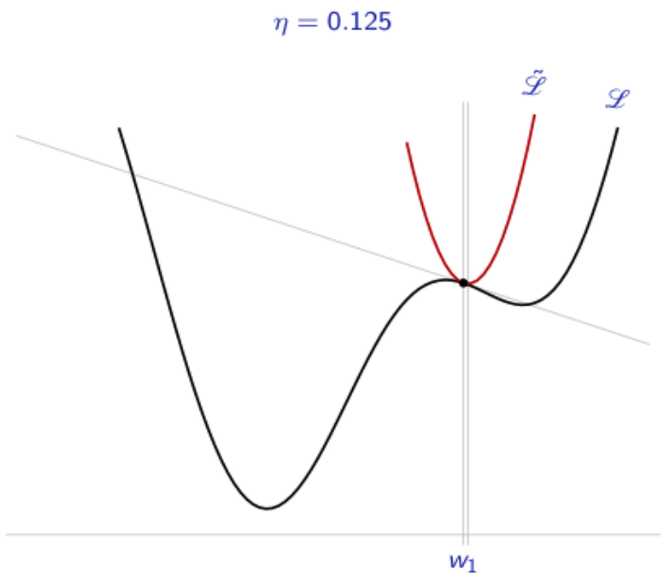
Gradient descent



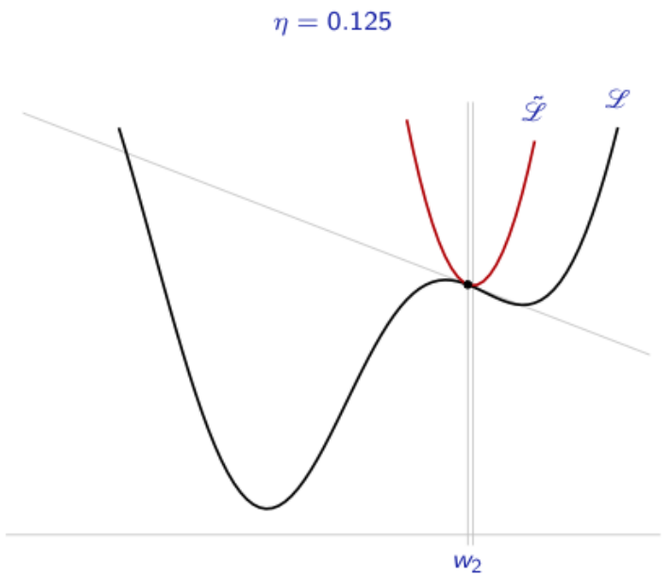
Gradient descent



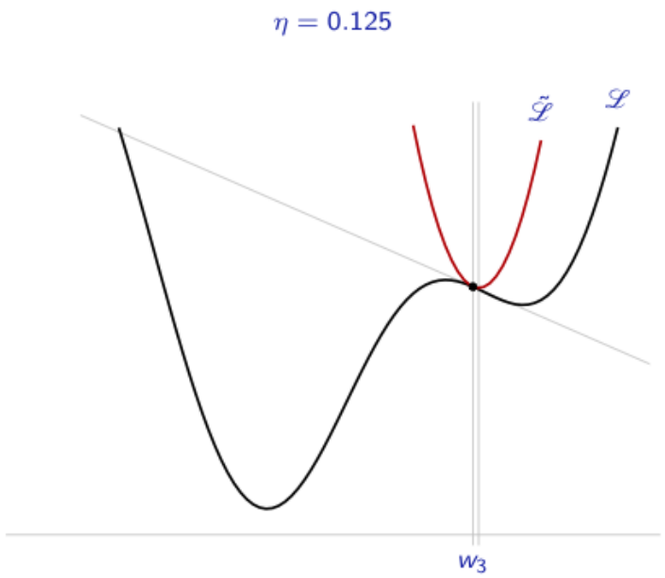
Gradient descent



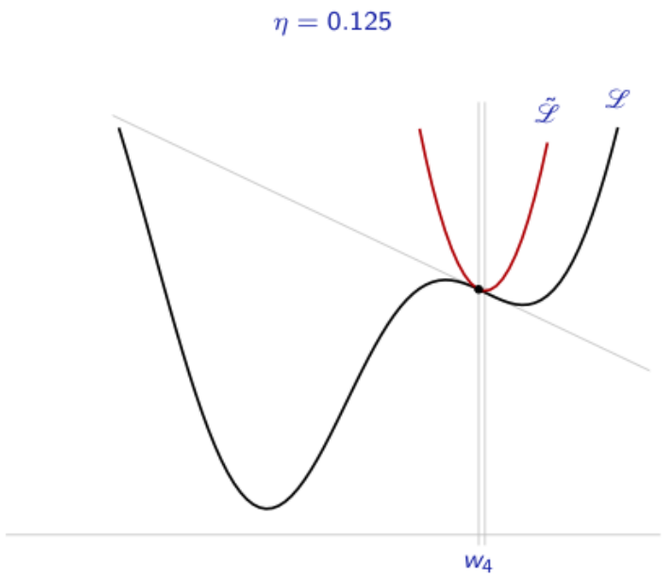
Gradient descent



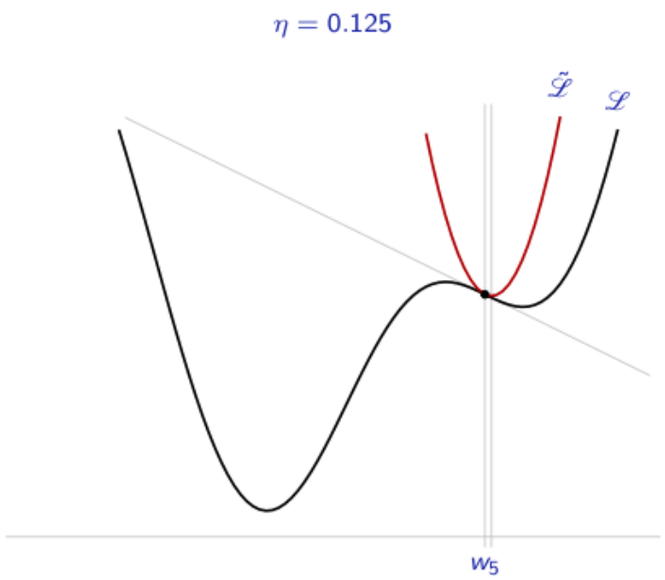
Gradient descent



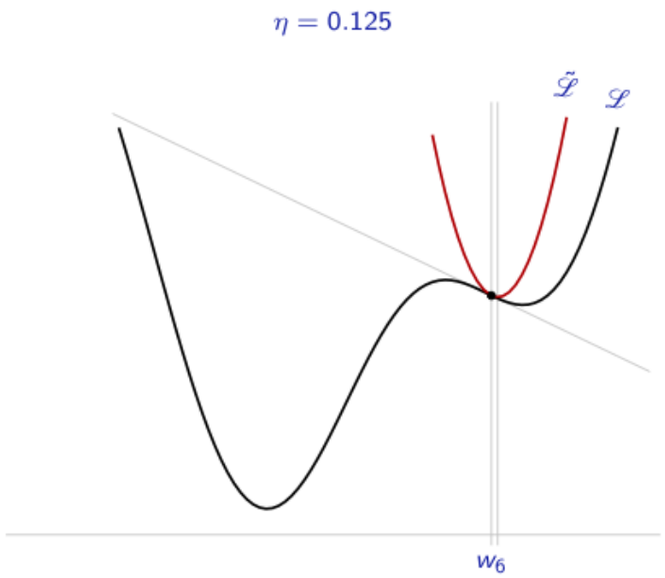
Gradient descent



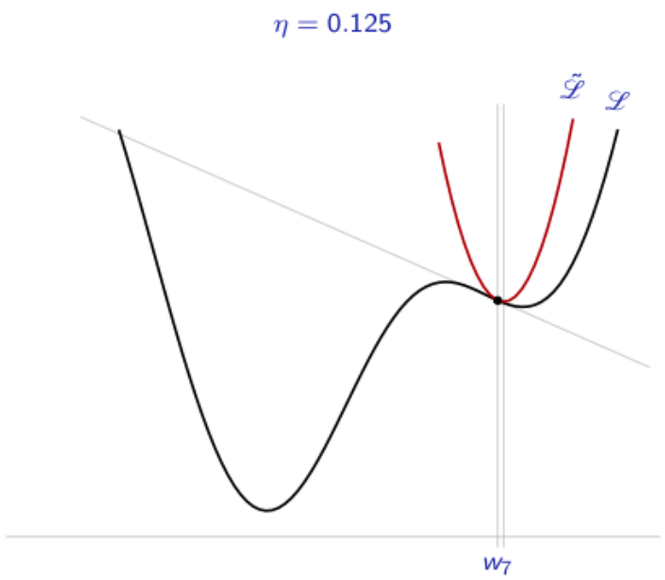
Gradient descent



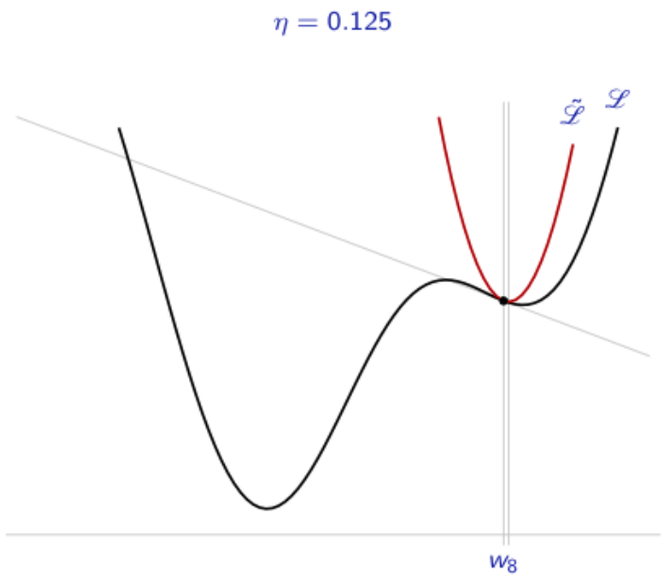
Gradient descent



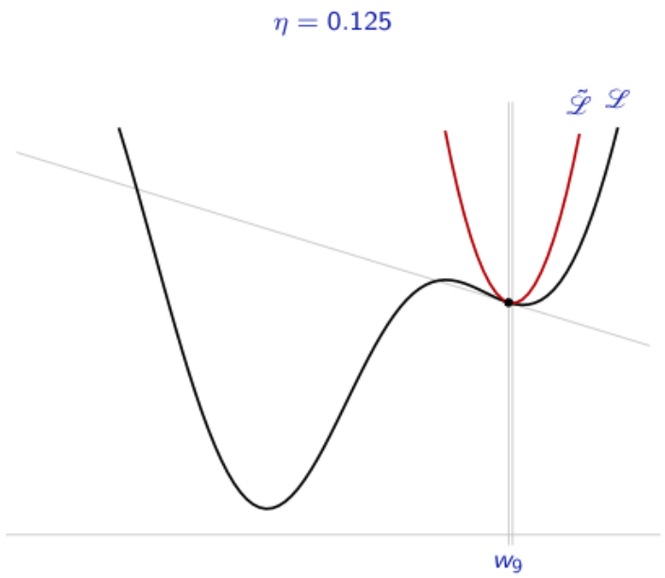
Gradient descent



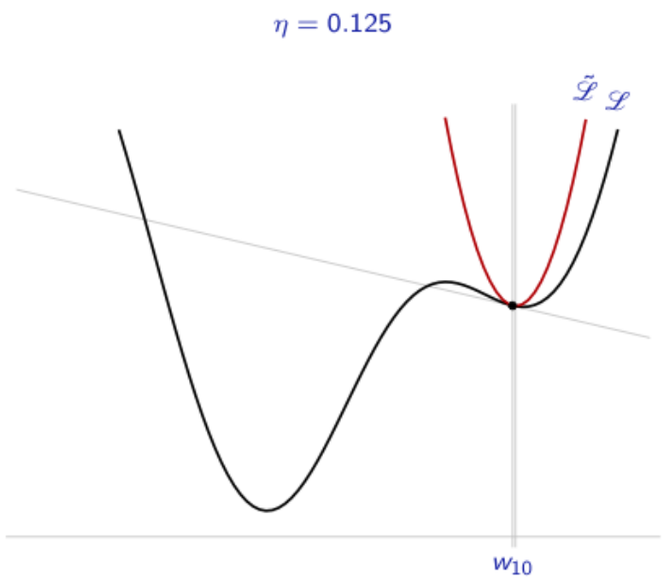
Gradient descent



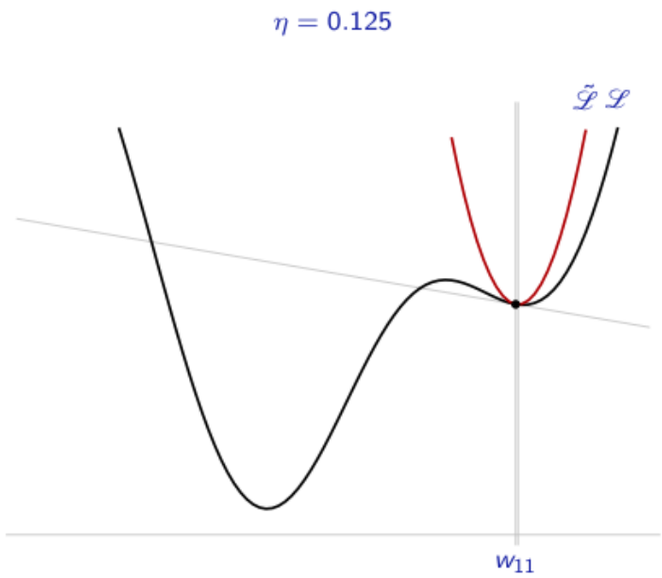
Gradient descent



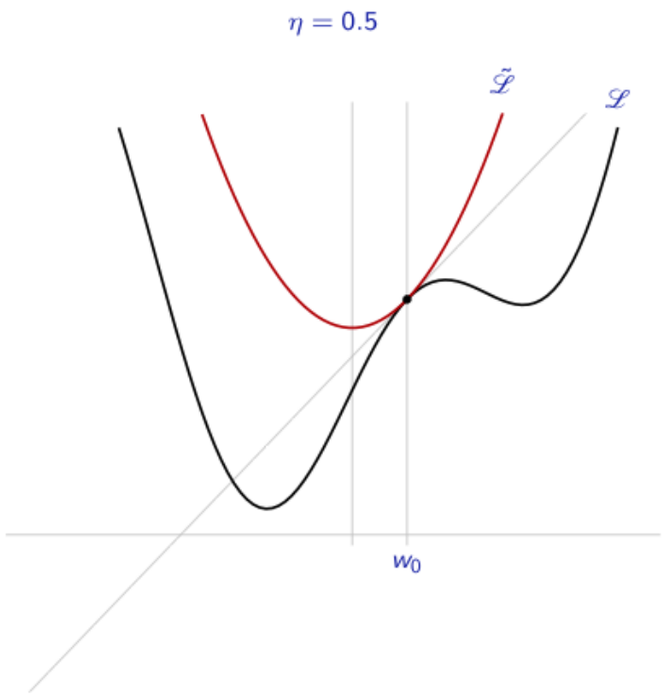
Gradient descent



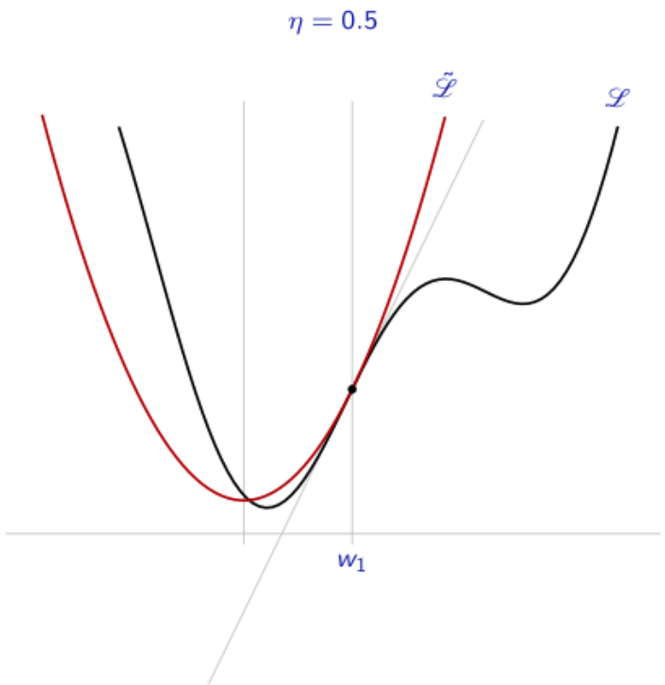
Gradient descent



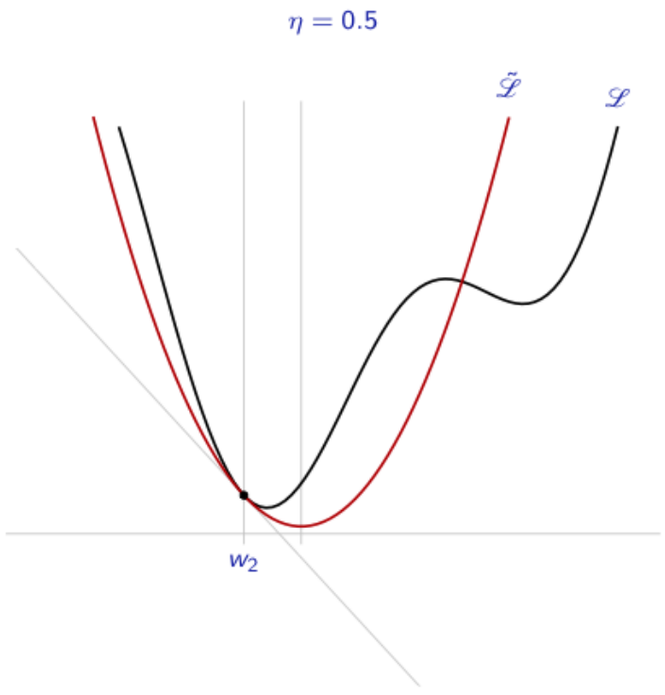
Gradient descent



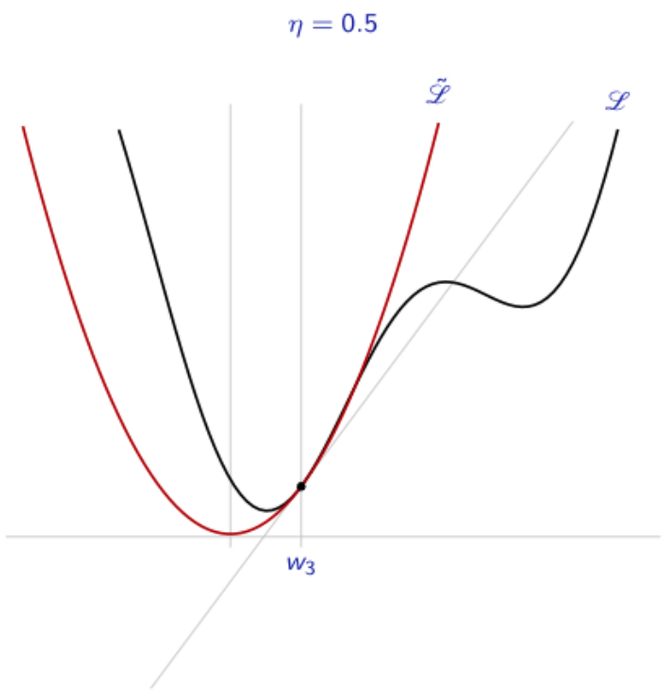
Gradient descent



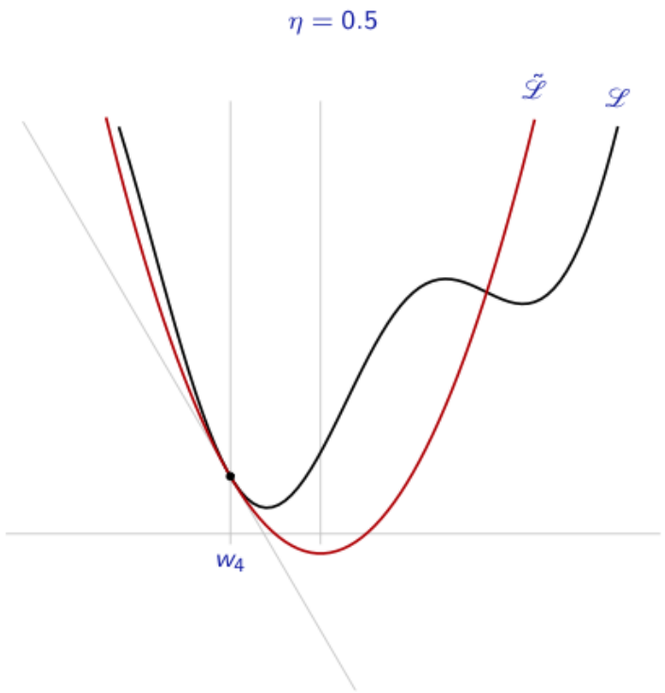
Gradient descent



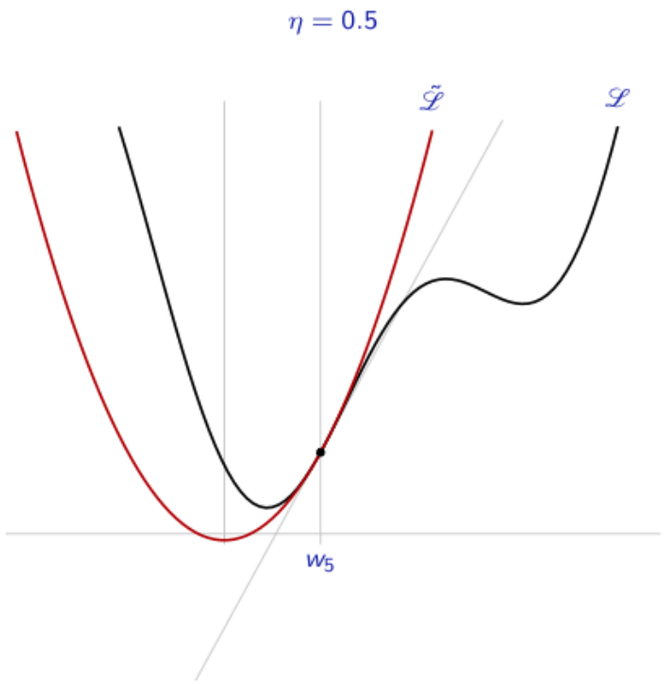
Gradient descent



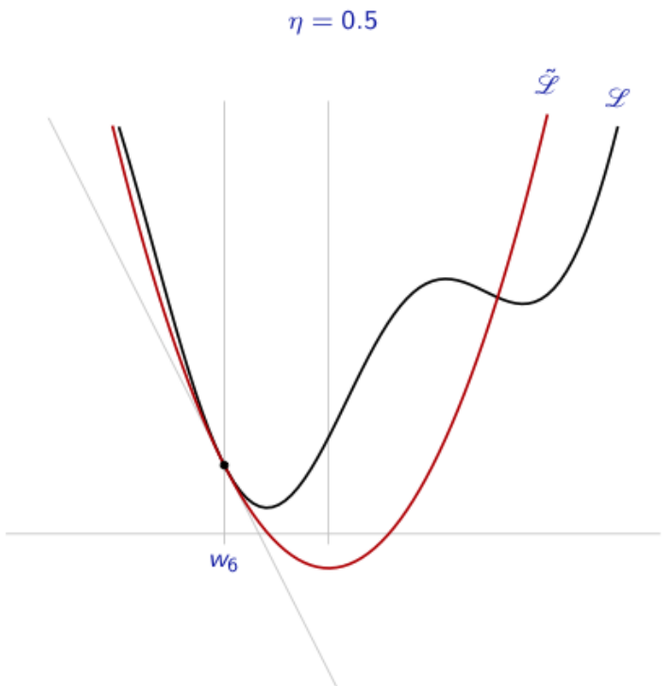
Gradient descent



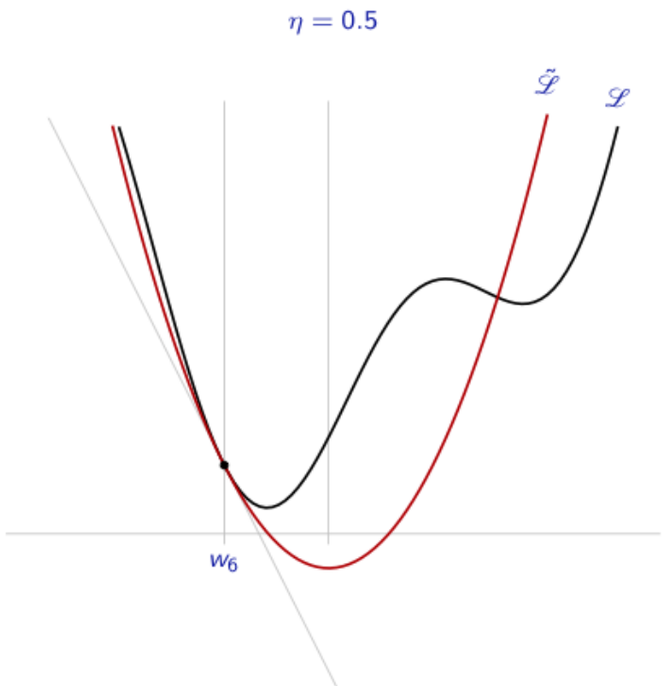
Gradient descent



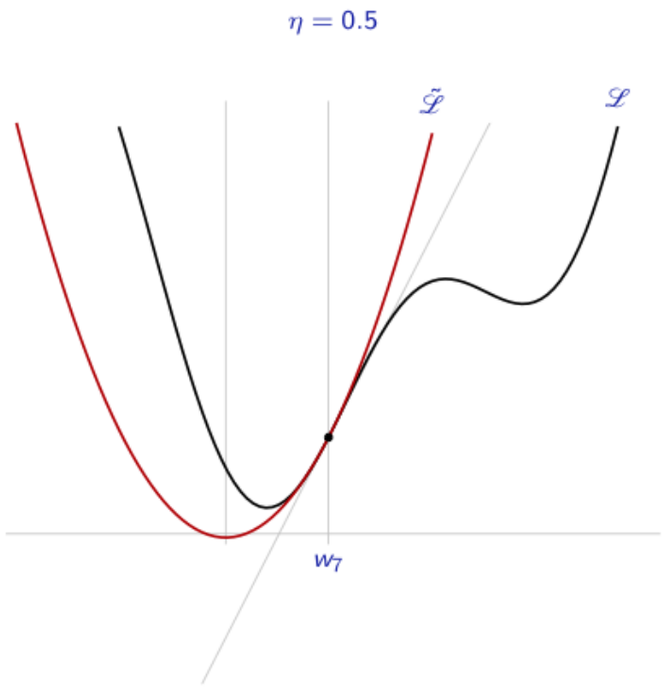
Gradient descent



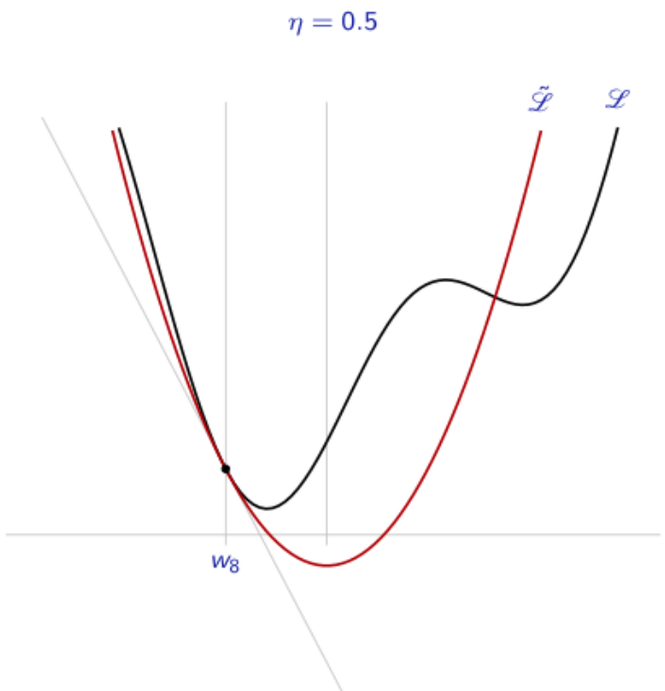
Gradient descent



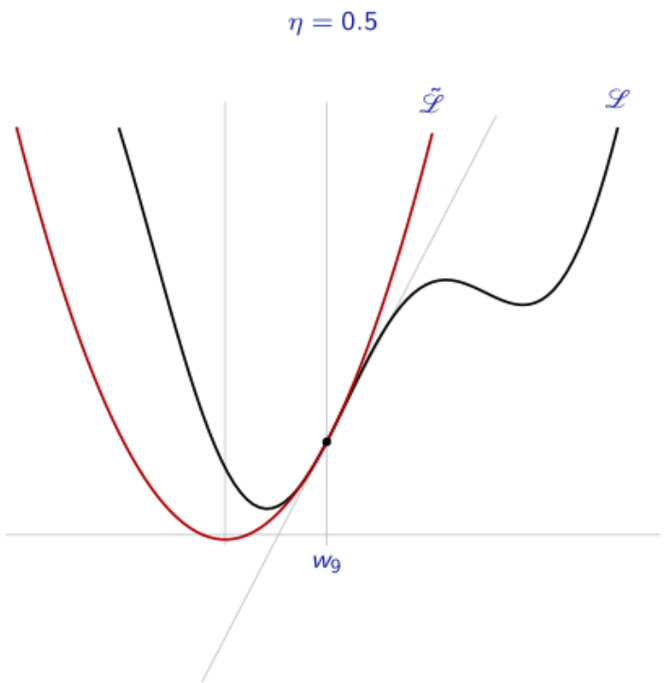
Gradient descent



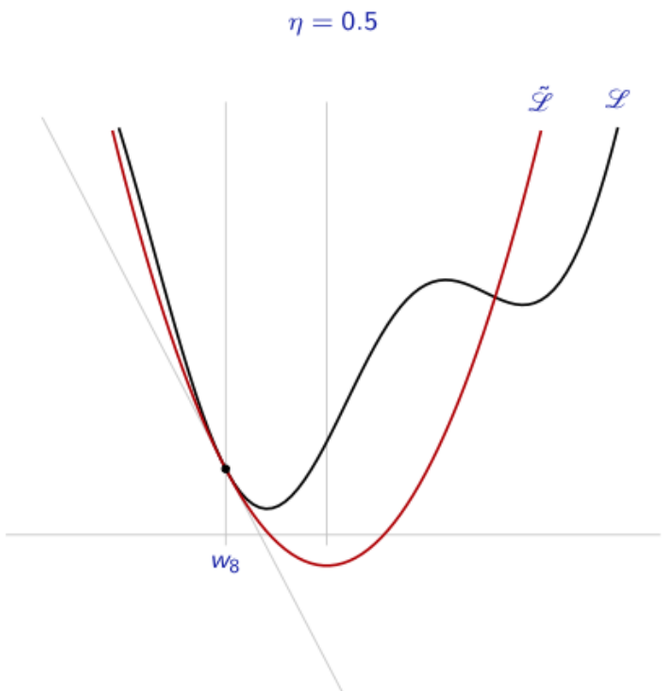
Gradient descent



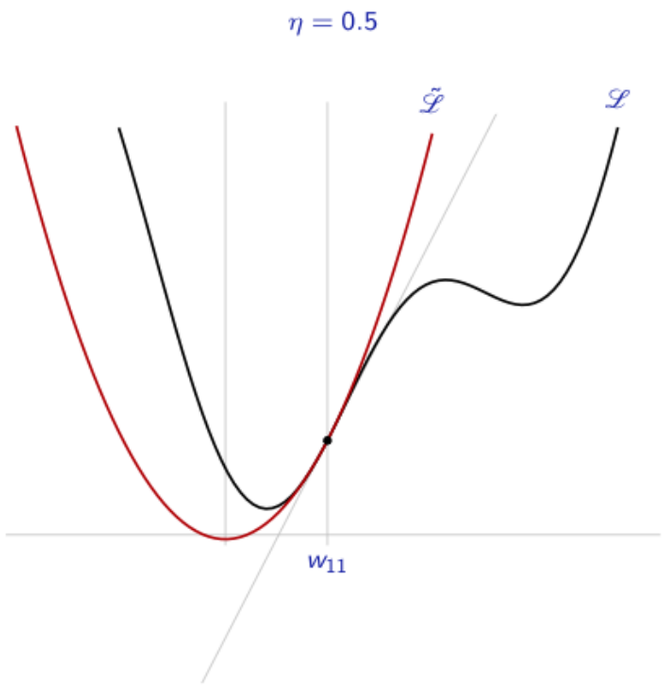
Gradient descent



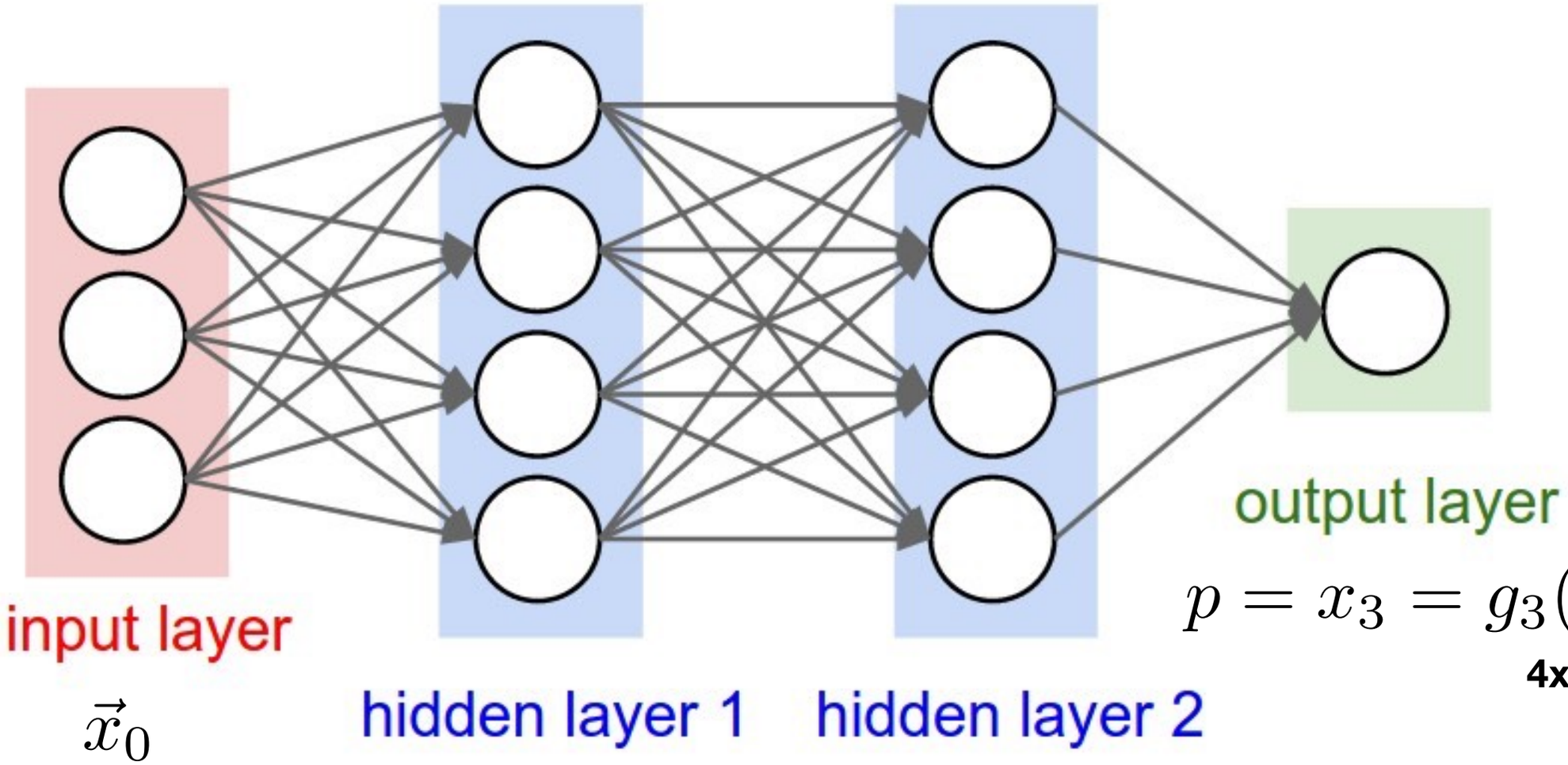
Gradient descent



Gradient descent



Feed-forward Neural networks



$$p = x_3 = g_3(W_3\vec{x}_2)$$

4x1 matrix

$\vec{x}_0$

$$\vec{x}_1 = g_1(W_1\vec{x}_0) \quad \vec{x}_2 = g_2(W_2\vec{x}_1)$$

4x3 matrix **4x4 matrix**

$$p = f(\vec{x}_0) = g_3(W_3\ g_2(W_2\ g_1(W_1\vec{x}_0)))$$

W matrices are called the « *weights* »
The functions g_n ( ) are called « *activation functions* »

How to compute the gradient efficiently?

$$\vec{x}_0 \quad \vec{x}_1 = g_1(\overbrace{W_1\vec{x}_0}^{\vec{h}_1}) \quad \dots \quad \vec{x}_n = g_n(\overbrace{W_n\vec{x}_{n-1}}^{\vec{h}_n}) \quad \dots \quad p = g_L(\overbrace{W_L\vec{x}_{L-1}}^{\vec{h}_L})$$

Feed-forward

Compute the loss $L = \frac{(y-p)^2}{2}$

Back-propagation of errors

$$e_j^1 = {g_1}'(h_j^1)\sum_i W_{ij}^2 e_i^2 \quad \dots \quad e_j^n = {g_n}'(h_j^n)\sum_i W_{ij}^{n+1} e_i^{n+1} \quad \dots \quad e^L = {g_L}'(h^L)(p-y)$$

Once this is done, gradients are given by $\frac{\partial L}{\partial W_{ab}^l} = x_b^{l-1} e_a^l$

Demonstration by the chain rule of derivatives

$$L = \frac{(y-p)^2}{2} \qquad \frac{\partial L}{\partial w_{ab}^{(l)}} = \textbf{?}$$

$$\frac{\partial L}{\partial w_{ab}^{(l)}} = \overbrace{(p-y)g'^{(L)}(h^{(L)})}^{e^L = {g_L}'(h^L)(p-y)} \sum_k w_k^{(L)} \frac{\partial x_k^{(L-1)}}{\partial w_{ab}^{(l)}} \quad \Rightarrow \quad \frac{\partial L}{\partial w_{ab}^{(l)}} = \sum_k w_k^{(L)} \frac{\partial x_k^{(L-1)}}{w_{ab}^{(l)}} e^L$$

$$\frac{\partial L}{\partial w_{ab}^{(l)}} = \sum_k w_k^{(L)} \left(\frac{\partial}{\partial w_{ab}^{(l)}} g^{(L-1)} \left[\sum_{k'} w_{kk'}^{(L-1)} x_{k'}^{(L-2)} \right] \right) e^L$$

$$\frac{\partial L}{\partial w_{ab}^{(l)}} = \sum_{k'} \frac{\partial x_{k'}^{(L-2)}}{\partial w_{ab}^{(l)}} \sum_k w_{kk'}^{(L-1)} \underbrace{w_k^{(L)} \left(g^{(L-1)'}[h_k^{L-1}] \right) e^L}_{e_k^{L-1}} = \sum_{k'} \frac{\partial x_{k'}^{(L-2)}}{\partial w_{ab}^{(l)}} \sum_k w_{kk'}^{(L-1)} e_k^{L-1}$$

...

$$\frac{\partial L}{\partial w_{ab}^{(l)}} = \sum_k \frac{\partial x_k^{(n-2)}}{w_{ab}^{(l)}} \sum_i w_{ik}^{(n-1)} e_i^{(n-1)}$$

...

$$\frac{\partial L}{\partial w_{ab}^{(l)}} = \sum_k \frac{\partial x_k^{(l)}}{w_{ab}^{(l)}} \sum_i w_{ik}^{(l+1)} e_i^{(l+1)} = x_b^{(l-1)} e_a^{(l)}$$

How to compute the gradient efficiently?

$$\vec{x}_0 \quad \vec{x}_1 = g_1(\overbrace{W_1\vec{x}_0}^{\vec{h}_1}) \quad \dots \quad \vec{x}_n = g_n(\overbrace{W_n\vec{x}_{n-1}}^{\vec{h}_n}) \quad \dots \quad p = g_L(\overbrace{W_L\vec{x}_{L-1}}^{\vec{h}_L})$$

Feed-forward

Compute the loss $L = \frac{(y-p)^2}{2}$

Back-propagation of errors

$$e_j^1 = {g_1}'(h_j^1)\sum_i W_{ij}^2 e_i^2 \quad \dots \quad e_j^n = {g_n}'(h_j^n)\sum_i W_{ij}^{n+1} e_i^{n+1} \quad \dots \quad e^L = {g_L}'(h^L)(p-y)$$

Once this is done, gradients are given by $\frac{\partial L}{\partial W_{ab}^l} = x_b^{l-1} e_a^l$

Minimising the cost function by gradients descent

$$\theta^{t+1} = \theta^t - \eta \nabla \mathscr{R}\left(\theta^t, \{\mathbf{x}\}_{i=1,\ldots n}, \{y\}_{i=1,\ldots n}\right)$$

If eta small enough, converge to a (possible local) minima

Standard (or "batch") gradient descent

Compute the gradient by averaging the derivative of the loss is the entire training set

$$\nabla \mathscr{R}\left(\theta^t, \{\mathbf{x}\}_{i=1,\ldots n}, \{y\}_{i=1,\ldots n}\right) = \frac{1}{n}\sum_{i=1}^{n} \nabla \mathscr{L}(\mathbf{x}_i, y_i, \theta^t)$$

Batch gradient is the average gradient over *all data in the training set*

Gradient descent

Batch gradient descent

$$\theta^{t+1} = \theta^t - \eta \nabla \mathscr{R}\left(\theta^t, \{\mathbf{x}\}_{i=1,\ldots n}, \{y\}_{i=1,\ldots n}\right)$$

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

$$\nabla \mathscr{R}\left(\theta^t, \{\mathbf{x}\}_{i=1,\ldots n}, \{y\}_{i=1,\ldots n}\right) = \frac{1}{n} \sum_{i=1}^{n} \nabla \mathscr{L}(\mathbf{x}_i, y_i, \theta^t)$$

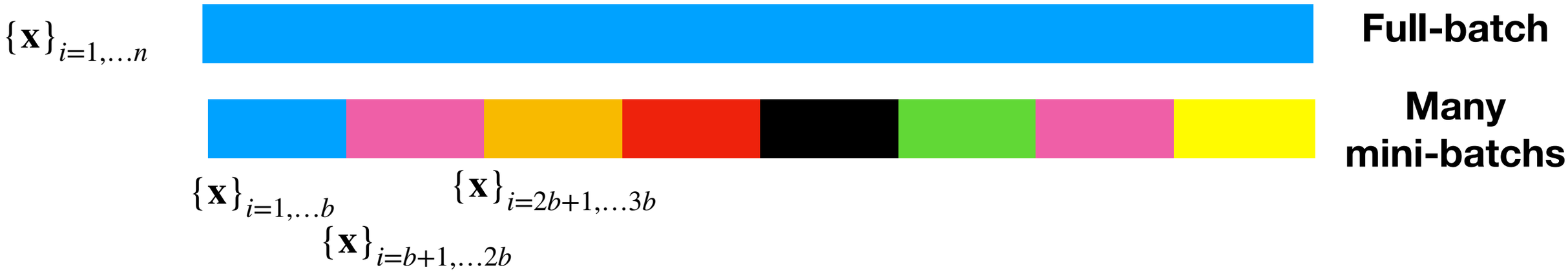
Batch gradient is the average gradient over *all data in the training set*

Gradient descent

Mini-batch gradient descent

$$\theta^{t+(1/n_b)} = \theta^t - \eta \nabla \mathscr{R}\left(\theta^t, \{\mathbf{x}\}_{i=n_1,\dots n_2}, \{y\}_{i=n_1,\dots n_2}\right)$$

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```



$$\nabla \mathscr{R}\left(\theta^t, \{\mathbf{x}\}_{i=i_1,\dots i_2}, \{y\}_{i=i_1,\dots i_2}\right) = \frac{1}{i_2 - i_1} \sum_{i=i_1}^{i_2} \nabla \mathscr{L}(\mathbf{x}_i, y_i, \theta^t)$$

Mini-Batch gradient is the average gradient over *all data in one mini-batch*

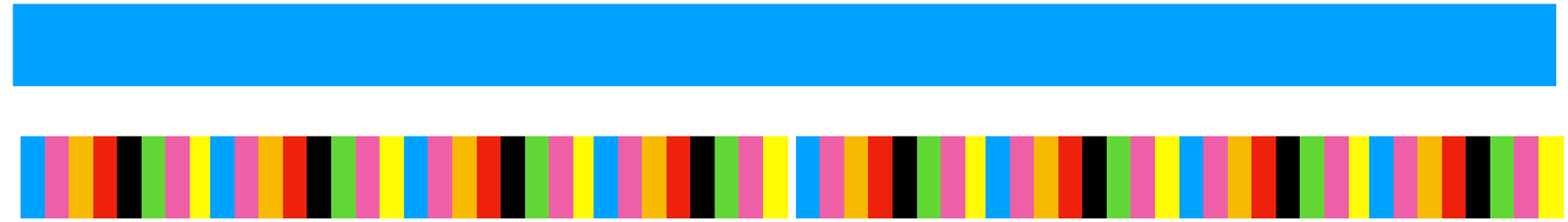
Gradient descent

Stochastic gradient descent

$$\theta^{t+(1/n)} = \theta^t - \eta \nabla \mathscr{R}\left(\theta^t, \{\mathbf{x}\}_i, \{y\}_i\right)$$

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

$\{\mathbf{x}\}_{i=1,\ldots n}$



Full-batch

Mini-batchs Of size 1…

$$\nabla \mathscr{R}\left(\theta^t, \{\mathbf{x}\}_i, \{y\}_i\right) = \nabla \mathscr{L}(\mathbf{x}_i, y_i, \theta^t)$$

SGD gradient is the gradient for *one element in the training set*

Why Mini-batch gradient descent?

$$\theta^{t+(1/n_b)} = \theta^t - \eta \nabla \mathscr{R} \left(\theta^t, \{\mathbf{x}\}_{i=n_1,\ldots n_2}, \{y\}_{i=n_1,\ldots n_2} \right)$$

- The model update frequency is hi
  faster and **memory efficient** *(c*
- **Maybe?** Effective noise in the dy
  Could works better than full batch
- **In practice:** This is the only way
  networks

The Tradeoffs of Large Scale Learning

Léon Bottou
NEC laboratories of America
Princeton, NJ 08540, USA
leon@bottou.org

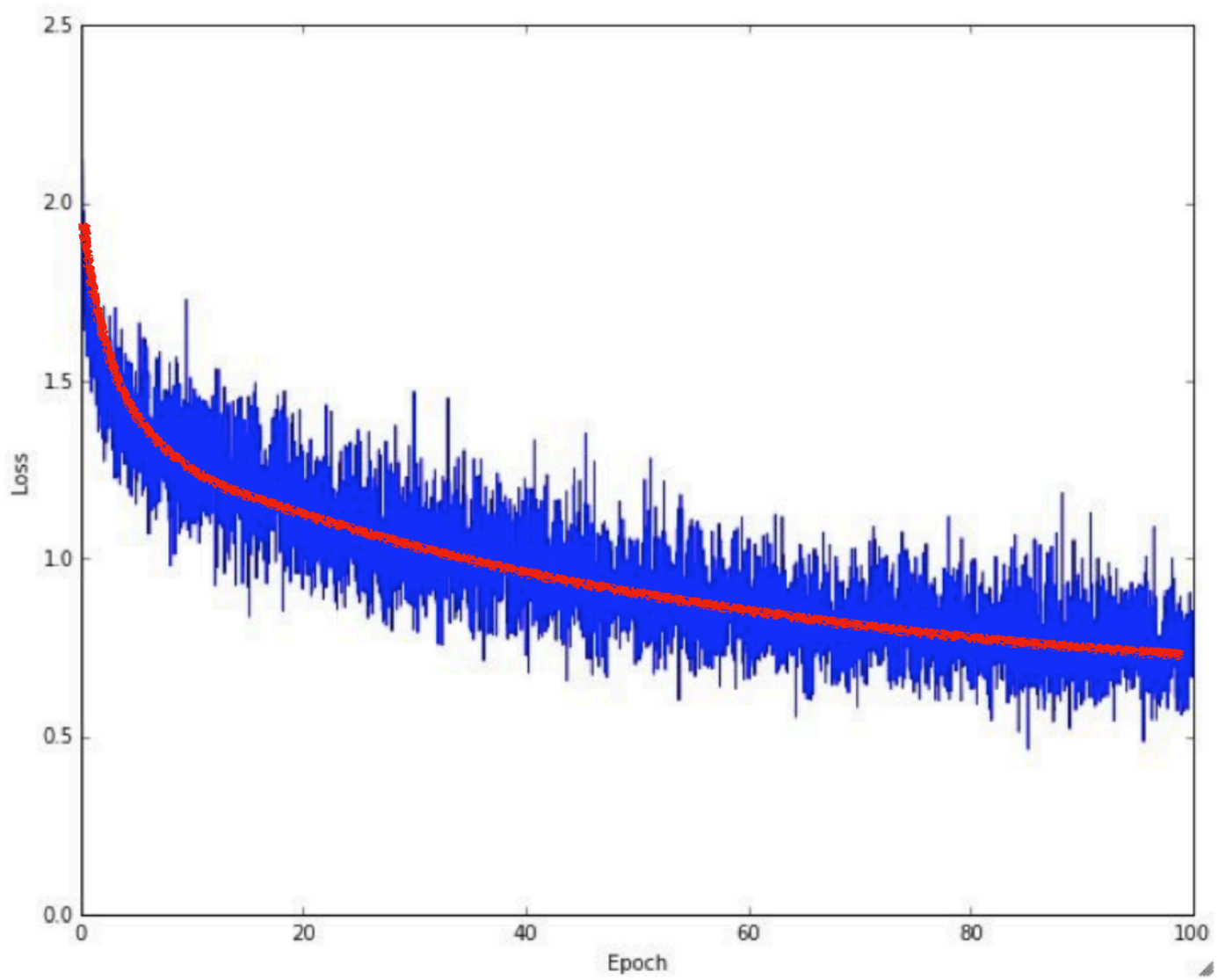
Olivier Bousquet
Google Zürich
8002 Zurich, Switzerland
olivier.bousquet@m4x.org

Abstract

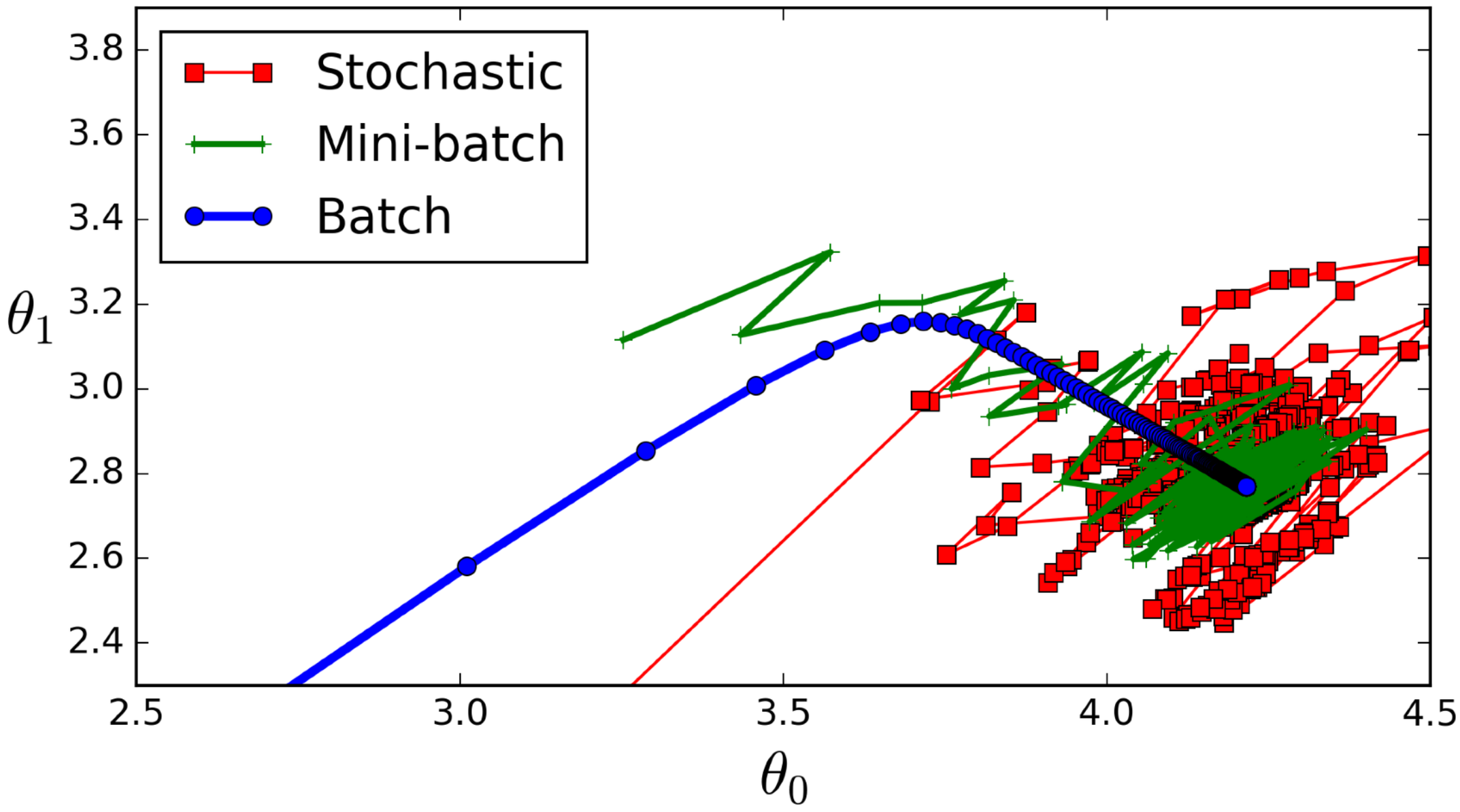
This contribution develops a theoretical framework that takes into account the effect of approximate optimization on learning algorithms. The analysis shows distinct tradeoffs for the case of small-scale and large-scale learning problems. Small-scale learning problems are subject to the usual approximation–estimation tradeoff. Large-scale learning problems are subject to a qualitatively different tradeoff involving the computational complexity of the underlying optimization algorithms in non-trivial ways.

Mini-batch gradient descent

- Example of optimization progress while training a neural network
- Showing loss over mini-batches as it goes down over time

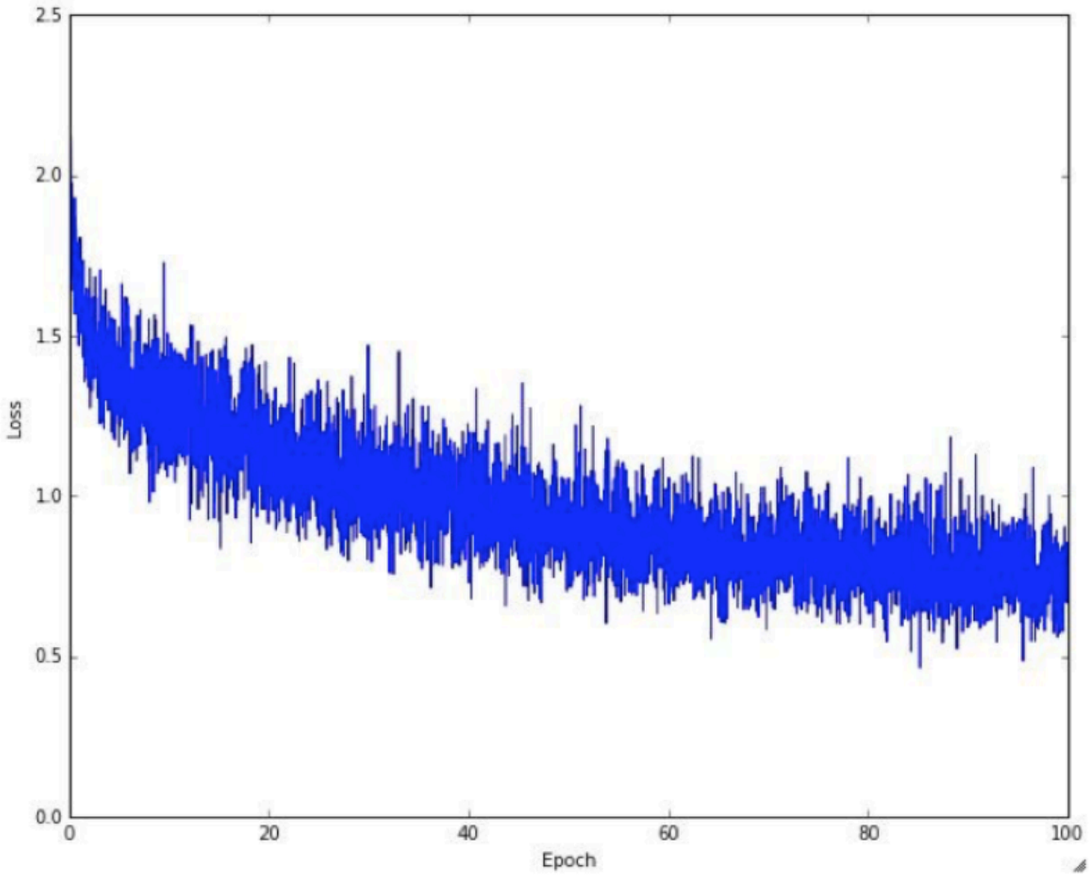


Batch vs mini-batches

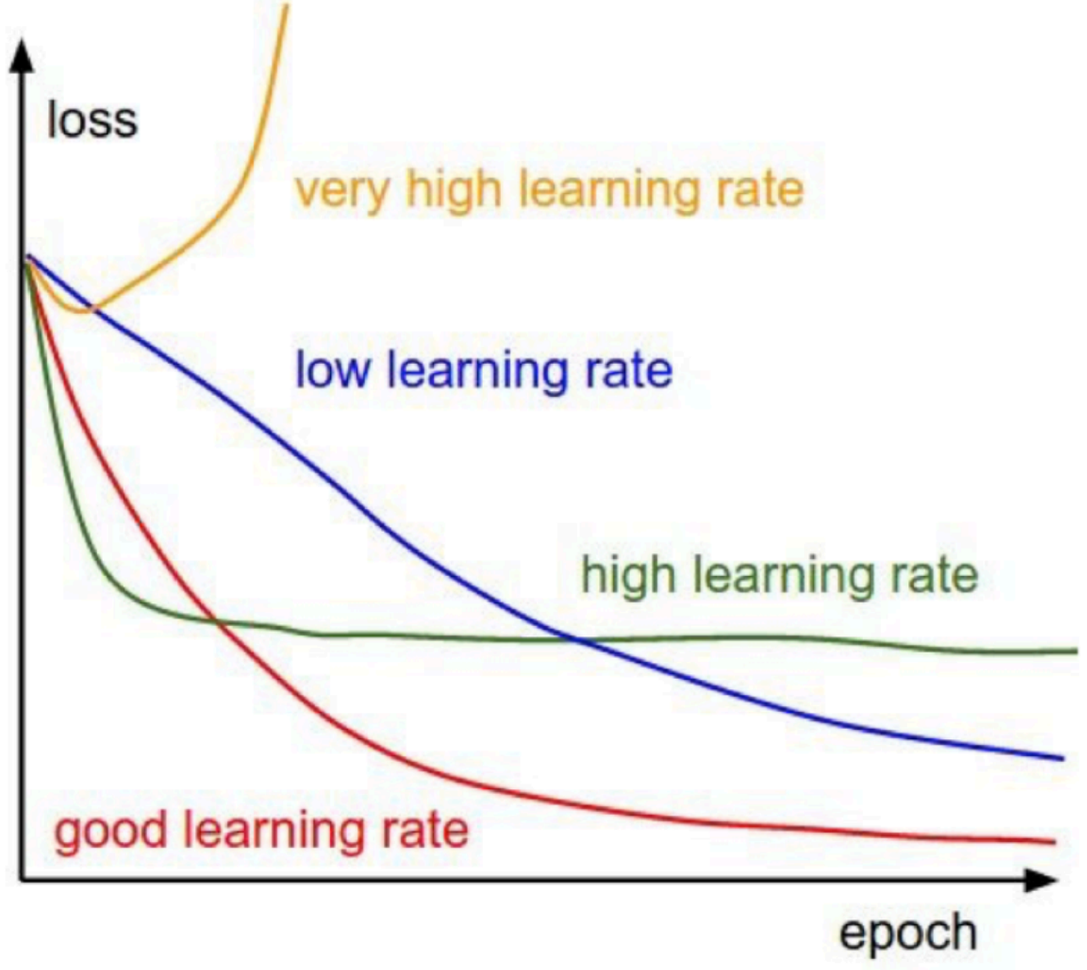


Mini-batch gradient descent

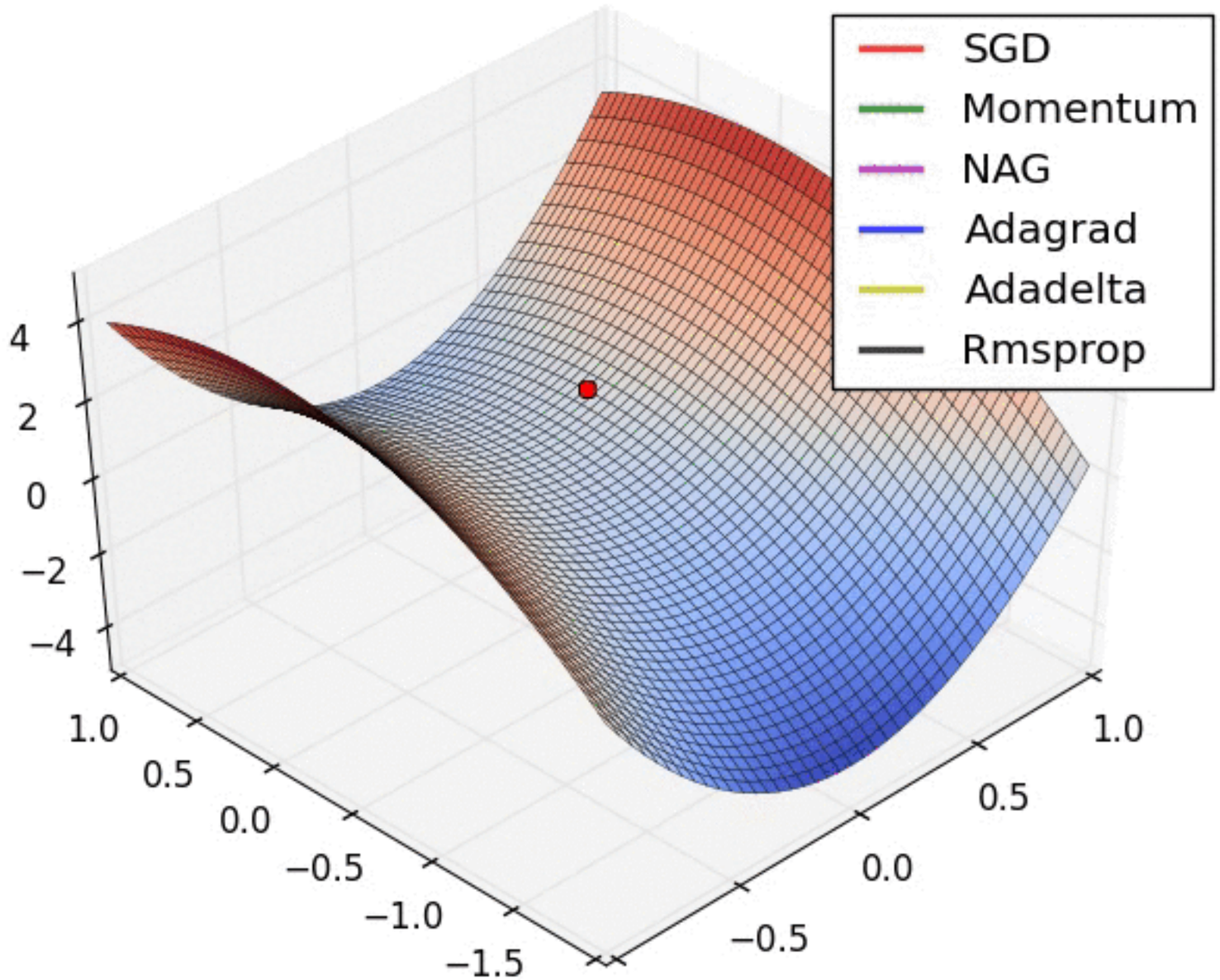
- Example of optimization progress while training a neural network
- **Epoch** = one full pass of the training dataset through the network



The effects of step size (or “learning rate”)



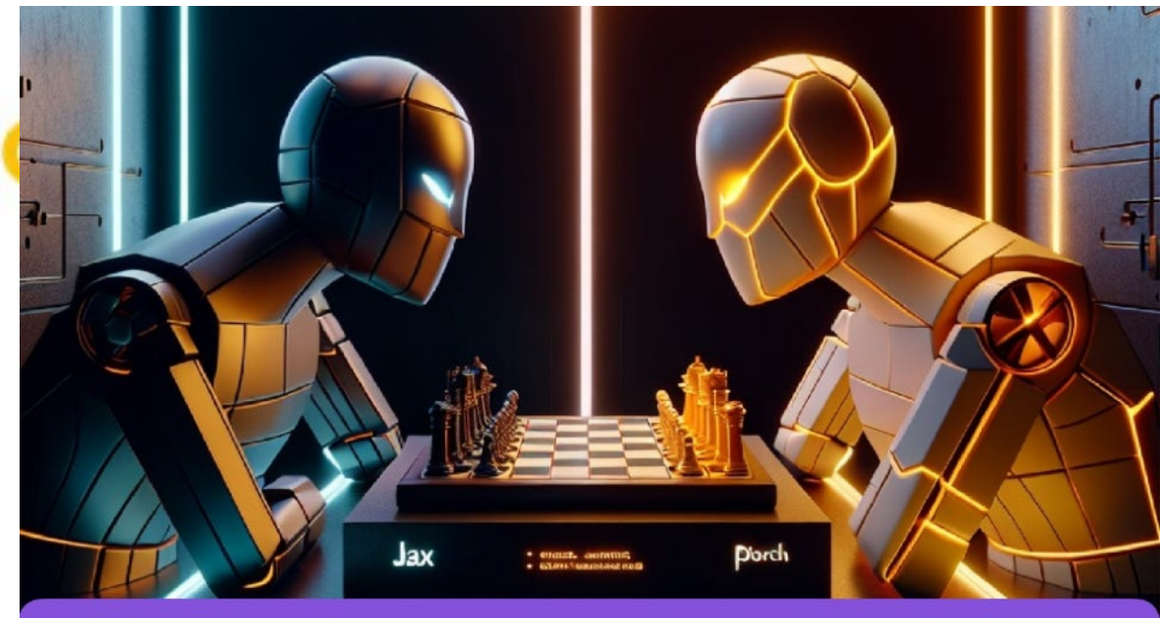
Many mini-batch algorithms
(but we shall discuss them later)



Using Neural nets!

Many Python Frameworks

- Pytorch & Torch
- TensorFlow
- Caffe
- Caffe2
- Chainer
- CNTK
- DSSTNE
- DyNet
- Gensim
- Gluon
- Keras
- Mxnet
- Paddle
- BigDL
- RIP: Theano & Ecosystem



Pytorch

```
class NeuralNetwork(nn.Module):
   def __init__(self):
      super(NeuralNetwork, self).__init__()
      self.flatten = nn.Flatten()
      self.linear_relu_stack = nn.Sequential(
      nn.Linear(28*28, 512),
      nn.ReLU(),
      nn.Linear(512, 512),
      nn.ReLU(),
      nn.Linear(512, 10),
      nn.ReLU()
  )
```

```
def forward(self, x):
   x = self.flatten(x)
   logits = self.linear_relu_stack(x)
   return logits
```

Note that the backward pass (backpropgation) is done automatically by pytorch (This is called *automatic differentiation*!)

No need to code back propagation if you use pytorch!

Gradient descents a gogo

How to compute the gradient efficiently?

$$\vec{x}_0 \quad \vec{x}_1 = g_1(\overbrace{W_1\vec{x}_0}^{\vec{h}_1}) \quad \dots \quad \vec{x}_n = g_n(\overbrace{W_n\vec{x}_{n-1}}^{\vec{h}_n}) \quad \dots \quad p = g_L(\overbrace{W_L\vec{x}_{L-1}}^{\vec{h}_L})$$

Feed-forward

Compute the loss $L = \frac{(y-p)^2}{2}$

Back-propagation of errors

$$e_j^1 = g_1{}'(h_j^1)\sum_i W_{ij}^2 e_i^2 \quad \dots \quad e_j^n = g_n{}'(h_j^n)\sum_i W_{ij}^{n+1} e_i^{n+1} \quad \dots \quad e^L = g_L{}'(h^L)(p-y)$$

Once this is done, gradients are given by $\frac{\partial L}{\partial W_{ab}^l} = x_b^{l-1} e_a^l$

Gradient descent

Batch gradient descent

$$\theta^{t+1} = \theta^t - \eta \nabla \mathscr{R} \left(\theta^t, \{\mathbf{x}\}_{i=1,\dots n}, \{y\}_{i=1,\dots n} \right)$$

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

Mini-batch gradient descent

$$\theta^{t+(1/n_b)} = \theta^t - \eta \nabla \mathscr{R} \left(\theta^t, \{\mathbf{x}\}_{i=n_1,\dots n_2}, \{y\}_{i=n_1,\dots n_2} \right)$$

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

Stochastic gradient descent

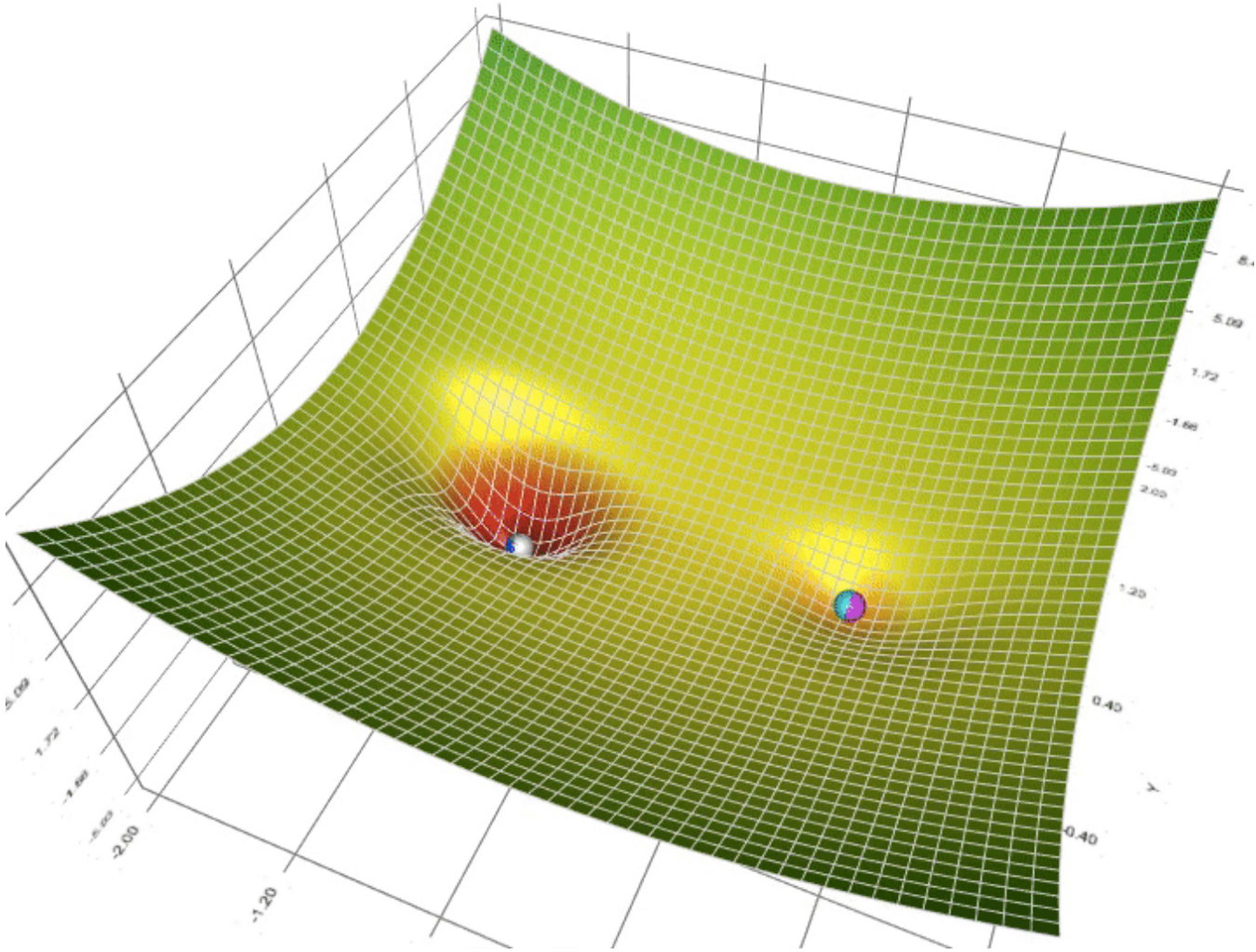
$$\theta^{t+(1/n)} = \theta^t - \eta \nabla \mathscr{R} \left(\theta^t, \{\mathbf{x}\}_i, \{y\}_i \right)$$

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```


bag of
tricks

Gradient descent

A physics analogy



“Speed”

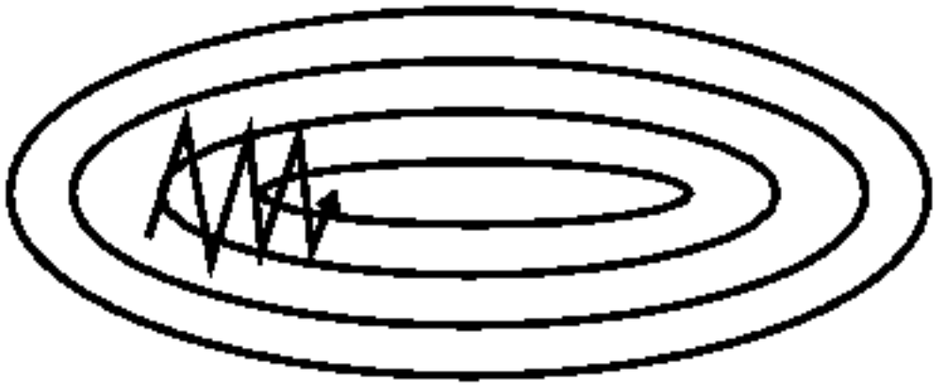
$$\mathbf{v}^{t+1} = \eta \nabla f(\theta^t)$$

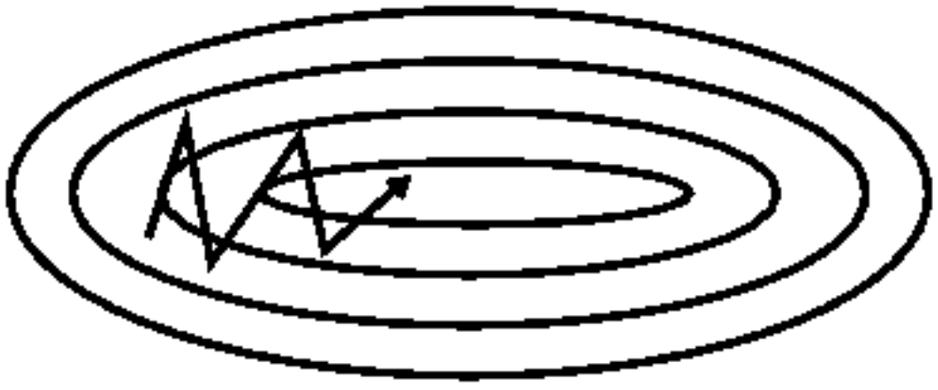
$$\theta^{t+1} = \theta^t - \mathbf{v}^{t+1}$$

“Movement”

Momentum

Keep the ball rolling on the same direction



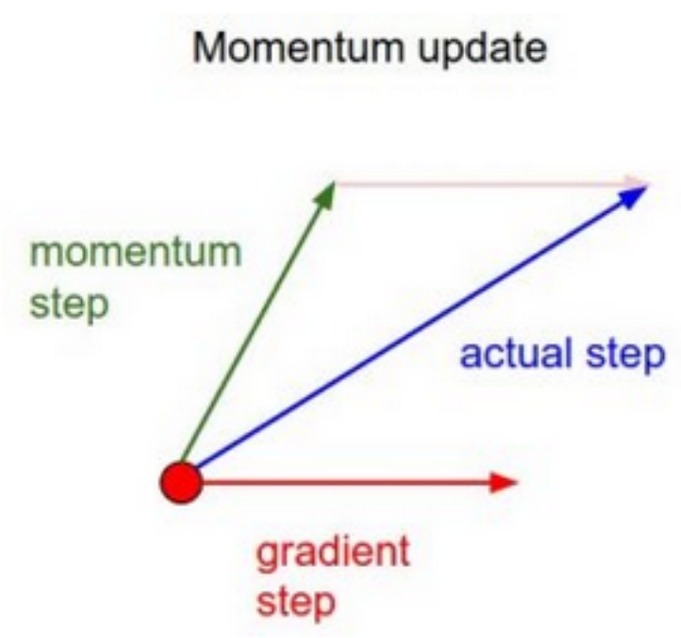


“Speed” change with the gradient of the force

$$\mathbf{v}^{t+1} = \gamma \mathbf{v}^{t} + \eta \nabla f(\theta^{t})$$

$$\theta^{t+1} = \theta^{t} - \mathbf{v}^{t+1}$$

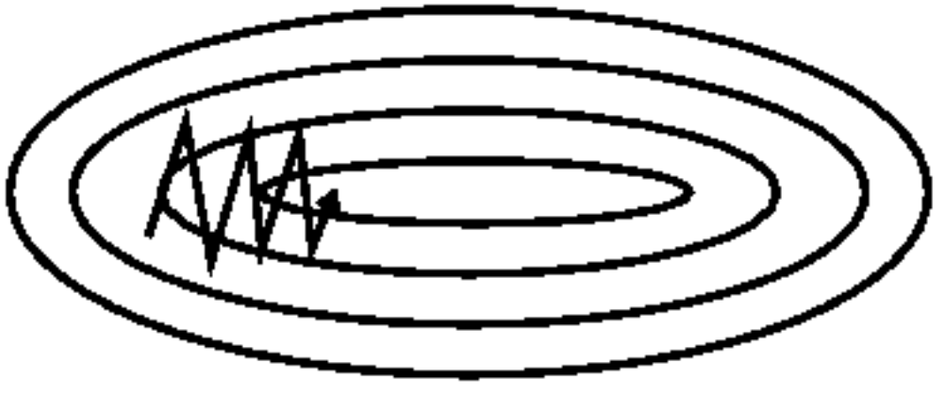
“Movement”

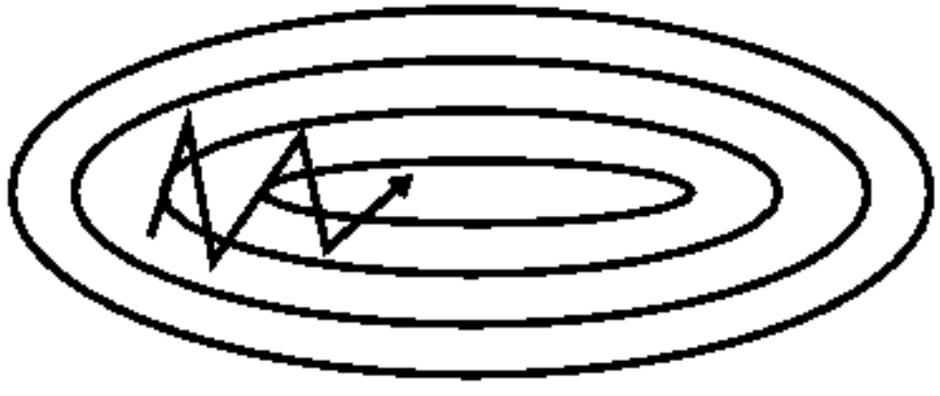


« Effective averaging of previous directions »

Nesterov acceleration

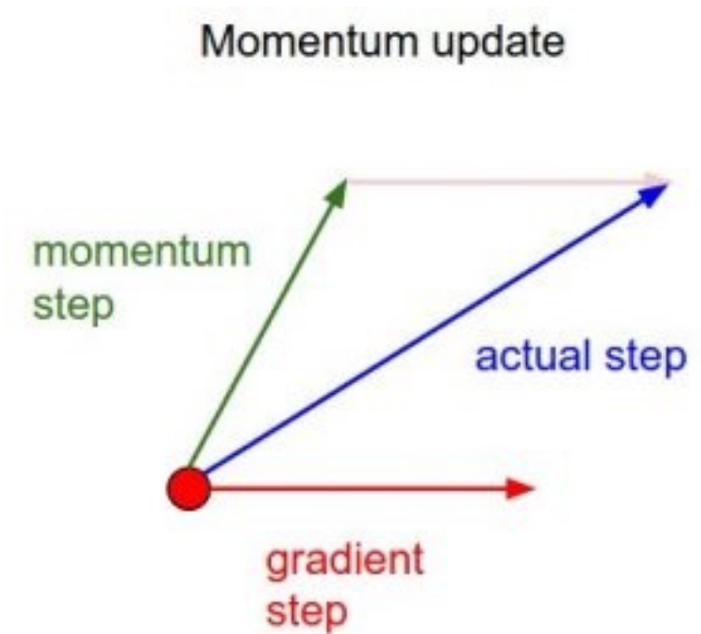
A slightly more clever ball

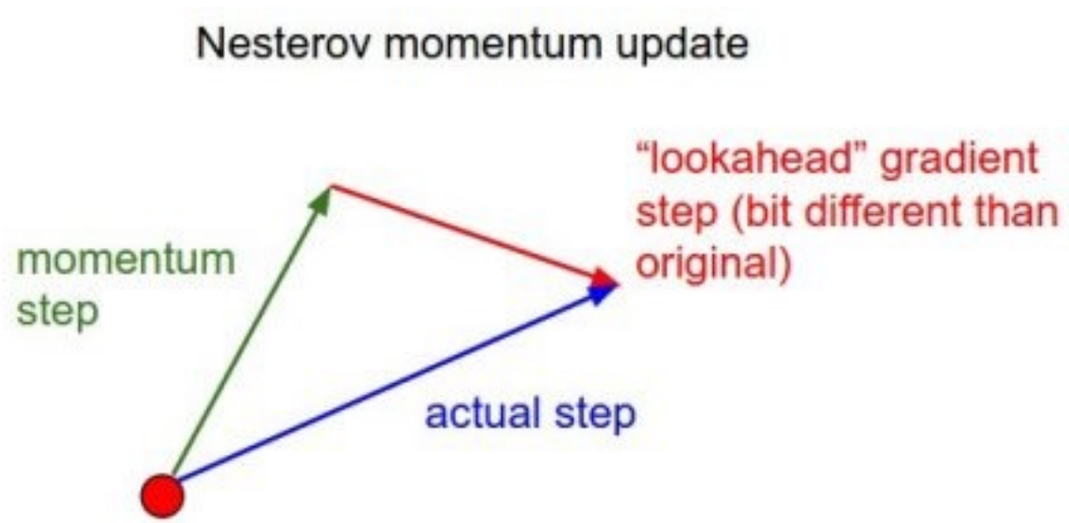




$$\mathbf{v}^{t+1} = \gamma \mathbf{v}^t + \eta \nabla f(\theta^t - \gamma \mathbf{v}^t)$$

$$\theta^{t+1} = \theta^t - \mathbf{v}^{t+1}$$





Pytorch optimizer

class `torch.optim.SGD`***(params, lr=<object object>, momentum=0, dampening=0, weight_decay=0, nesterov=False)*** [source]

Implements stochastic gradient descent (optionally with momentum).

Nesterov momentum is based on the formula from On the importance of initialization and momentum in deep learning.

Parameters:

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*) – learning rate
- **momentum** (*float, optional*) – momentum factor (default: 0)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)
- **dampening** (*float, optional*) – dampening for momentum (default: 0)
- **nesterov** (*bool, optional*) – enables Nesterov momentum (default: False)

Example

```
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
>>> optimizer.zero_grad()
>>> loss_fn(model(input), target).backward()
>>> optimizer.step()
```

Adaptive learning rates

$$\theta^{t+1} = \theta^t - \eta \nabla f(\theta^t)$$

What about this guy ?

Adagrad:

Adagrad scales γ for each parameter according to the history of gradients (previous steps)

$$\theta^{t+1} = \theta^t - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla f(\theta^t)$$

G is a diagonal matrix that contains the sum of all (squared) gradient so far
When the gradient is very large, learning rate is reduced and vice-versa.

$$G_{t+1} = G_t + (\nabla f)^2$$

With adagrad, one does not need to manually adapt γ at each steps…

… but the problem is that eventually all update on gradients goes to zero!

Adaptive learning rates

Adagrad:

Adagrad scales γ for each parameter according to the history of gradients (previous steps)

$$\theta^{t+1} = \theta^t - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla f(\theta^t)$$

G is a diagonal matrix that contains the sum of all (squared) gradient so far
When the gradient is very large, learning rate is reduced and vice-versa.

$$G_{t+1} = G_t + (\nabla f)^2$$

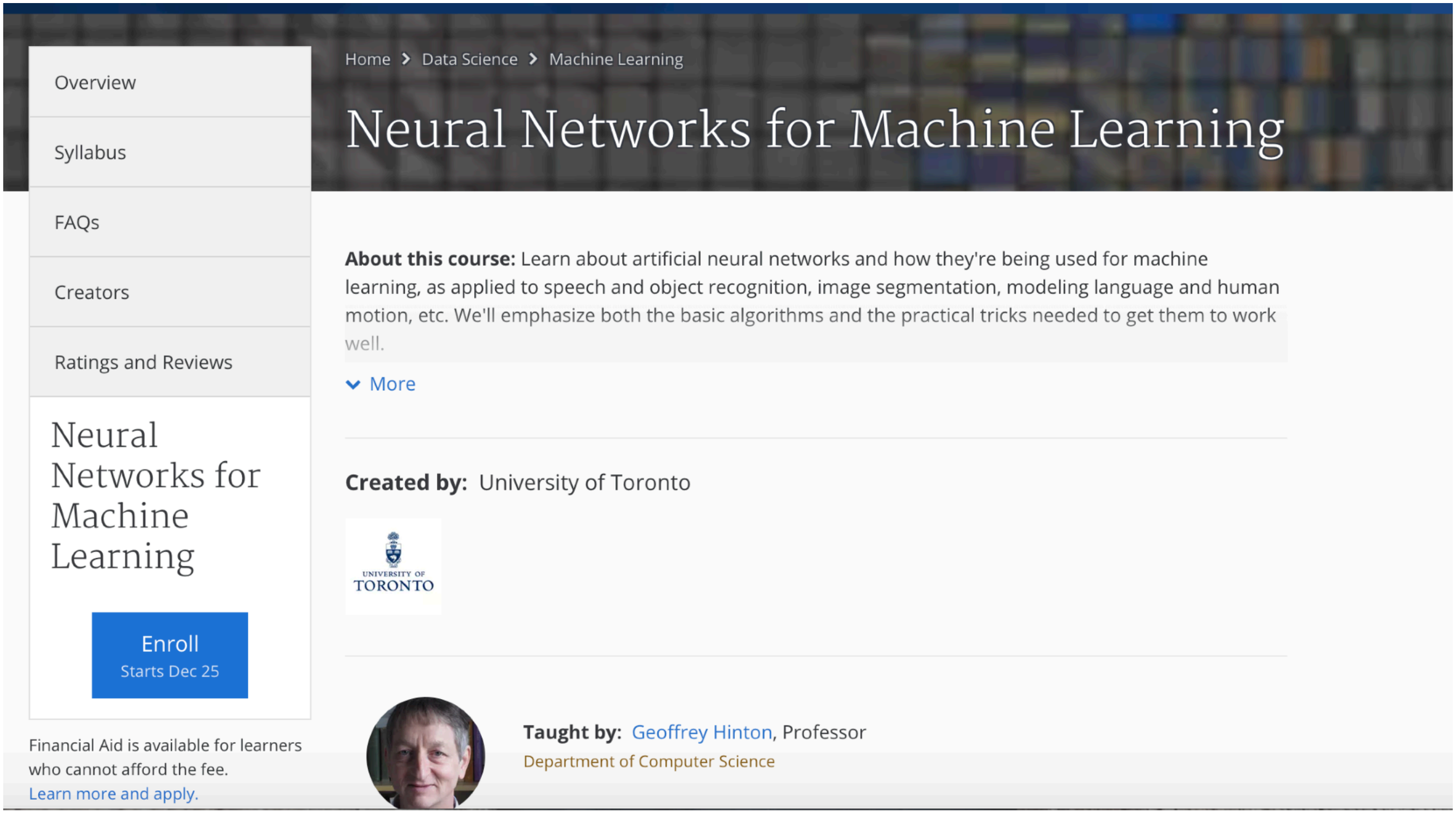
RMSprop

The only difference RMSprop has with Adagrad is that the term is calculated by exponentially decaying moving average (like we did in momentum for the gradient itself!) instead of the sum of gradients.

$$G_{t+1} = \gamma G_t + (1 - \gamma)(\nabla f)^2$$

RMSprop

Proposed by G. Hinton during his coursera lecture



Adaptive learning rates

ADAM= Adaptive learning rate + Momentum

Adam: Adaptive Moment Estimation

Adam also keeps an exponentially decaying average of past gradients, similar to momentum

$$G_t = \beta_2 G_{t-1} + (1-\beta_2)(\nabla f)^2$$

$$M_t = \beta_1 M_{t-1} + (1-\beta_1)(\nabla f)$$

These are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method.

$$\hat{M}_t = \frac{M_t}{1-\beta_1^{\mathbf{t}}} \quad \hat{G}_t = \frac{G_t}{1-\beta_2^{\mathbf{t}}}$$

$$\theta^{t+1} = \theta^t - \frac{\eta}{\sqrt{\hat{G}_t + \epsilon}} \hat{M}_t$$

Pytorch

ADAM

CLASS `torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False)` [SOURCE]

Implements Adam algorithm.

$$
\begin{aligned}
&\rule{110mm}{0.4pt} \\
&\textbf{input} : \gamma \text{ (lr)}, \beta_1, \beta_2 \text{ (betas)}, \theta_0 \text{ (params)}, f(\theta) \text{ (objective)} \\
&\hspace{13mm} \lambda \text{ (weight decay)}, \ amsgrad \\
&\textbf{initialize} : m_0 \leftarrow 0 \text{ (first moment)}, v_0 \leftarrow 0 \text{ (second moment)}, \ \widehat{v_0}^{max} \leftarrow 0 \\
&\rule{110mm}{0.4pt} \\
&\textbf{for } t = 1 \textbf{ to } \ldots \textbf{ do} \\
&\quad g_t \leftarrow \nabla_\theta f_t(\theta_{t-1}) \\
&\quad \textbf{if } \lambda \neq 0 \\
&\quad\quad g_t \leftarrow g_t + \lambda \theta_{t-1} \\
&\quad m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t \\
&\quad v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \\
&\quad \widehat{m_t} \leftarrow m_t / \big(1 - \beta_1^t\big) \\
&\quad \widehat{v_t} \leftarrow v_t / \big(1 - \beta_2^t\big) \\
&\quad \textbf{if } amsgrad \\
&\quad\quad \widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t}) \\
&\quad\quad \theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \big(\sqrt{\widehat{v_t}^{max}} + \epsilon\big) \\
&\quad \textbf{else} \\
&\quad\quad \theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \big(\sqrt{\widehat{v_t}} + \epsilon\big) \\
&\rule{110mm}{0.4pt}
\end{aligned}
$$

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float, optional*) – learning rate (default: 1e-3)
- **betas** (*Tuple*[*float, float*], *optional*) – coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability (default: 1e-8)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)
- **amsgrad** (*boolean, optional*) – whether to use the AMSGrad variant of this algorithm from the paper On the Convergence of Adam and Beyond (default: False)

Pytorch

NADAM

ADAM+Nesterov

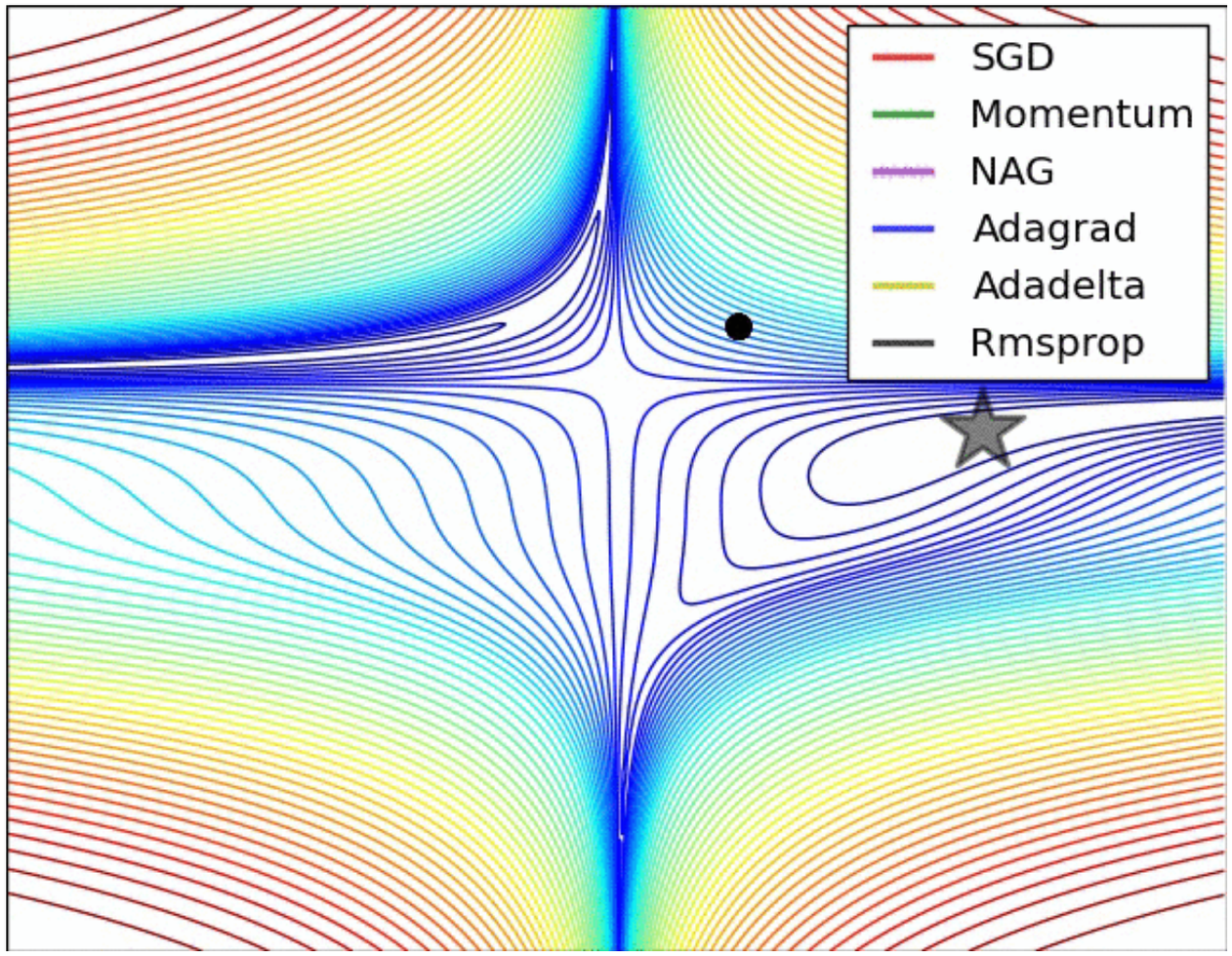
CLASS `torch.optim.NAdam(`*`params`*`, `*`lr=0.002`*`, `*`betas=(0.9, 0.999)`*`, `*`eps=1e-08`*`, `*`weight_decay=0`*`, `*`momentum_decay=0.004`*`)` [SOURCE]

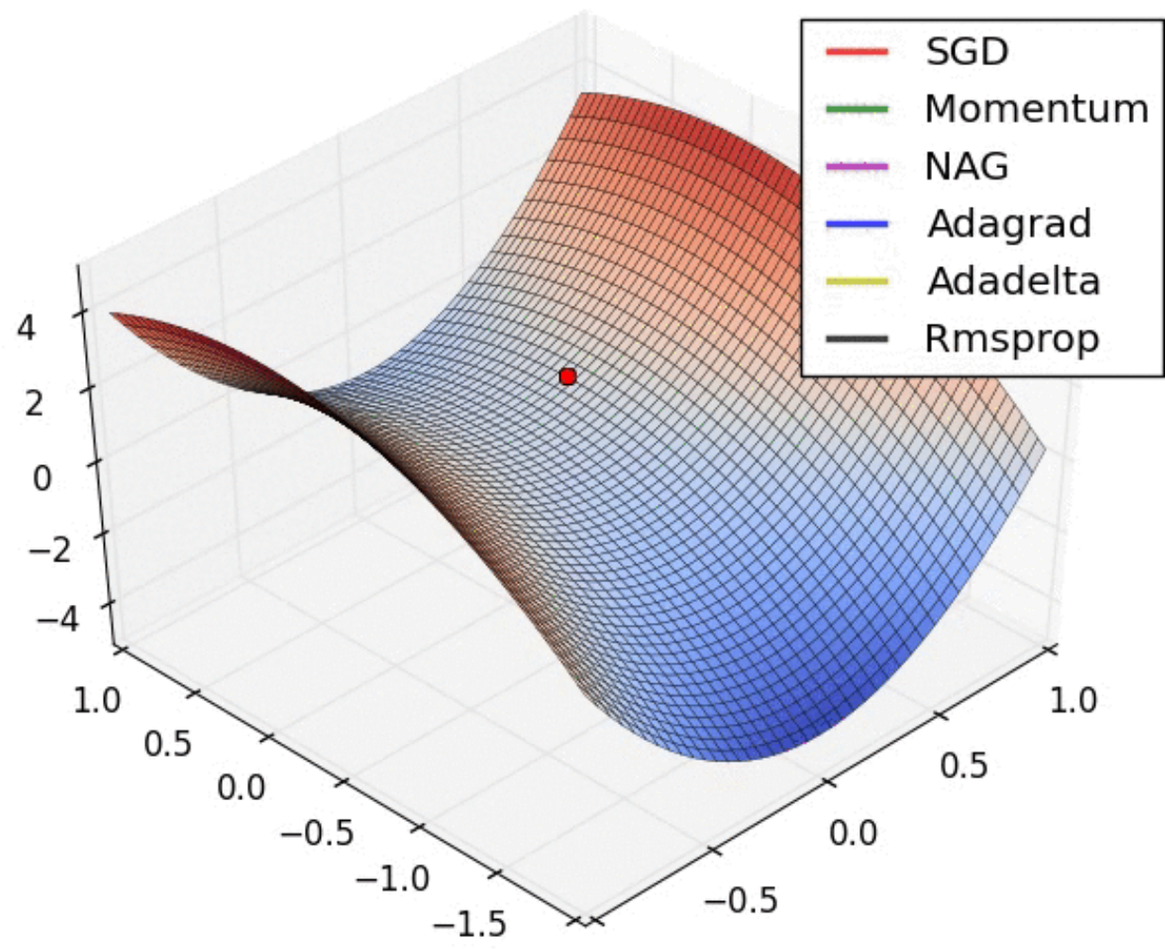
Implements NAdam algorithm.

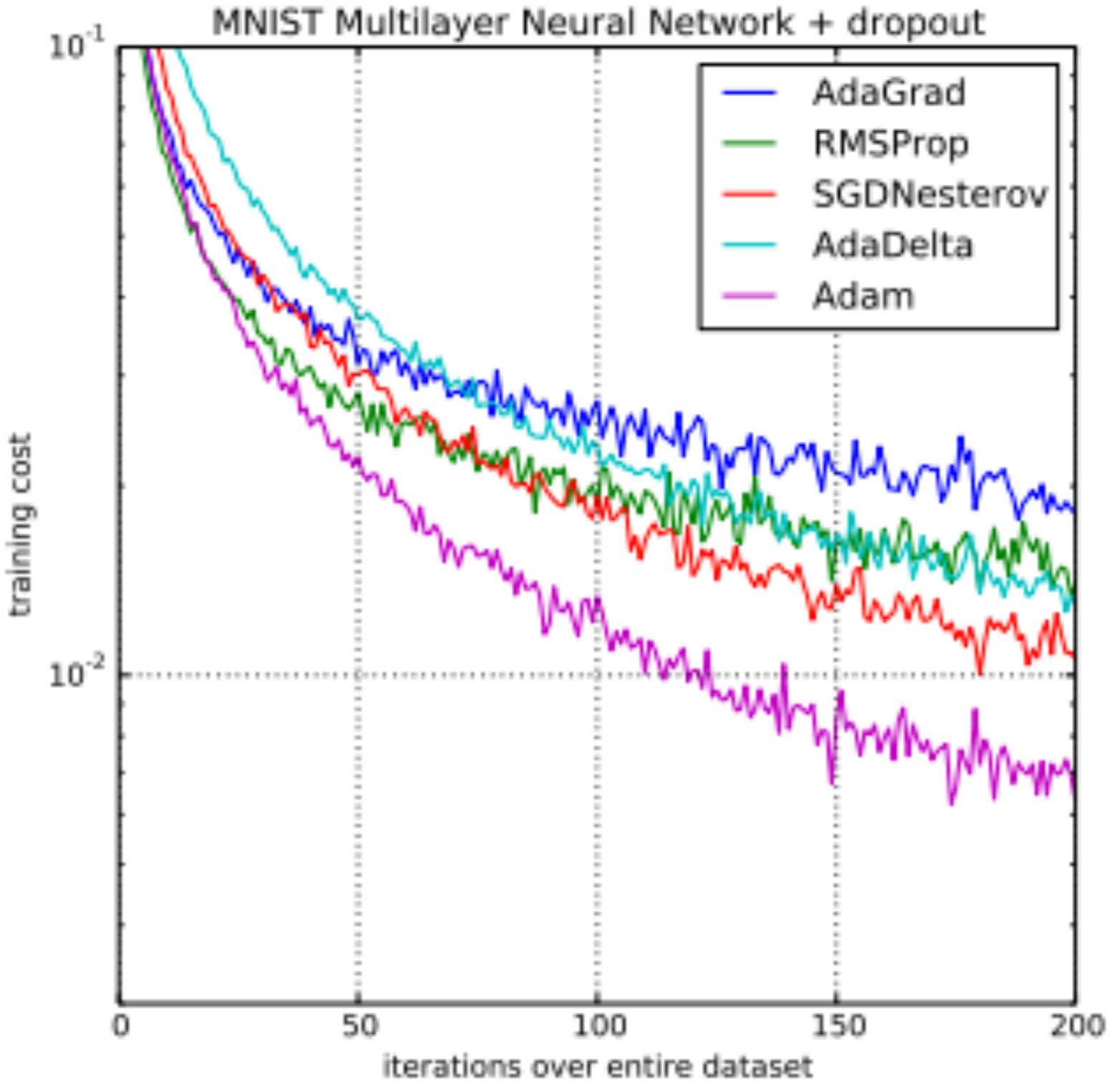
$$
\begin{aligned}
&\rule{110mm}{0.4pt} \\
&\textbf{input} : \gamma_t \text{ (lr)}, \ \beta_1, \beta_2 \text{ (betas)}, \ \theta_0 \text{ (params)}, \ f(\theta) \text{ (objective)} \\
&\hspace{13mm} \lambda \text{ (weight decay)}, \ \psi \text{ (momentum decay)} \\
&\textbf{initialize} : m_0 \leftarrow 0 \text{ (first moment)}, v_0 \leftarrow 0 \text{ (second moment)} \\
&\rule{110mm}{0.4pt} \\
&\textbf{for } t = 1 \textbf{ to } \ldots \textbf{ do} \\
&\quad g_t \leftarrow \nabla_\theta f_t(\theta_{t-1}) \\
&\quad if \ \lambda \neq 0 \\
&\quad\quad g_t \leftarrow g_t + \lambda \theta_{t-1} \\
&\quad \mu_t \leftarrow \beta_1 \big(1 - \frac{1}{2} 0.96^{t\psi}\big) \\
&\quad \mu_{t+1} \leftarrow \beta_1 \big(1 - \frac{1}{2} 0.96^{(t+1)\psi}\big) \\
&\quad m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t \\
&\quad v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \\
&\quad \widehat{m_t} \leftarrow \mu_{t+1} m_t / (1 - \prod_{i=1}^{t+1} \mu_i) \\
&\quad\quad + (1 - \mu_t) g_t / (1 - \prod_{i=1}^{t} \mu_i) \\
&\quad \widehat{v_t} \leftarrow v_t / \big(1 - \beta_2^t\big) \\
&\quad \theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \big(\sqrt{\widehat{v_t}} + \epsilon\big) \\
&\rule{110mm}{0.4pt} \\
&\textbf{return } \theta_t \\
&\rule{110mm}{0.4pt}
\end{aligned}
$$

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float, optional*) – learning rate (default: 2e-3)
- **betas** (*Tuple*[*float, float*], *optional*) – coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability (default: 1e-8)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)
- **momentum_decay** (*float, optional*) – momentum momentum_decay (default: 4e-3)


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop

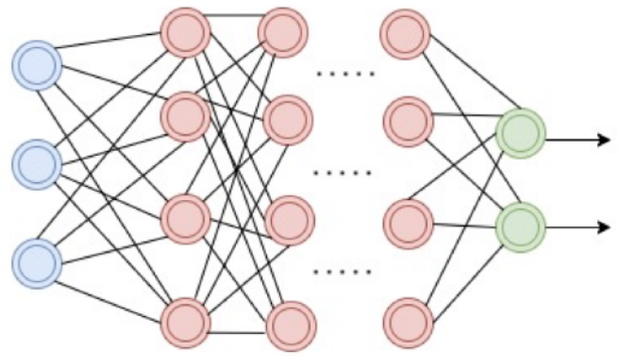

SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop
4
2
0
−2
−4
1.0
0.5
0.0
−0.5
−1.0
−1.5
1.0
0.5
0.0
−0.5


MNIST Multilayer Neural Network + dropout
AdaGrad
RMSProp
SGDNesterov
AdaDelta
Adam
10^{-1}
10^{-2}
training cost
0
50
100
150
200
iterations over entire dataset

In summary

Neural networks are parametric functions of the form:

$$\hat{y} = \sigma_L \left(W_L \cdot \sigma_{L-1}(W_{L-1}(\ldots . \sigma_1 \left(W_1 \vec{x} + b_1 \right) + b_2 \right)$$



They are “trained” by finding the “Weights” using gradient descent to minimise the empirical risk. In practice, this is done using mini-batchs

$$\theta^{t+(1/n_b)} = \theta^t - \eta \nabla \mathscr{R} \left(\theta^t, \{\mathbf{x}\}_{i=n_1,\ldots n_2}, \{y\}_{i=n_1,\ldots n_2} \right)$$

$$\nabla \mathscr{R} \left(\theta^t, \{\mathbf{x}\}_{i=i_1,\ldots i_2}, \{y\}_{i=i_1,\ldots i_2} \right) = \frac{1}{i_2 - i_1} \sum_{i=i_1}^{i_2} \nabla \mathscr{L}(\mathbf{x}_i, y_i, \theta^t)$$

$\{\mathbf{x}\}_{i=1,\ldots b}$

$\{\mathbf{x}\}_{i=b+1,\ldots 2b}$

$\{\mathbf{x}\}_{i=2b+1,\ldots 3b}$

Many mini-batchs

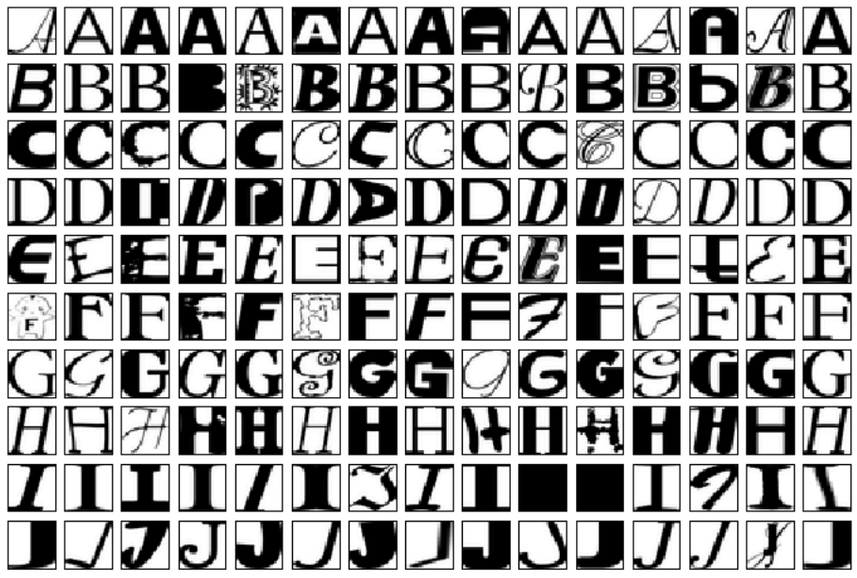
Le prix Turing honore les pères du deep learning Yann LeCun, Yoshua Bengio et Geoffrey Hinton

Le Français Yann LeCun, le Britannique Geoffrey Hinton et le Canadien Yoshua Bengio, les trois pères fondateurs du deep learning, ont reçu le 27 mars le prix Turing, équivalent du Nobel d'informatique, pour leur apport à l'intelligence artificielle.

- A bag of tricks: dropout, batchnorm, etc…
- Special layers: embedding, convolutions, pooling, etc…
- Convolution Networks (CNN)

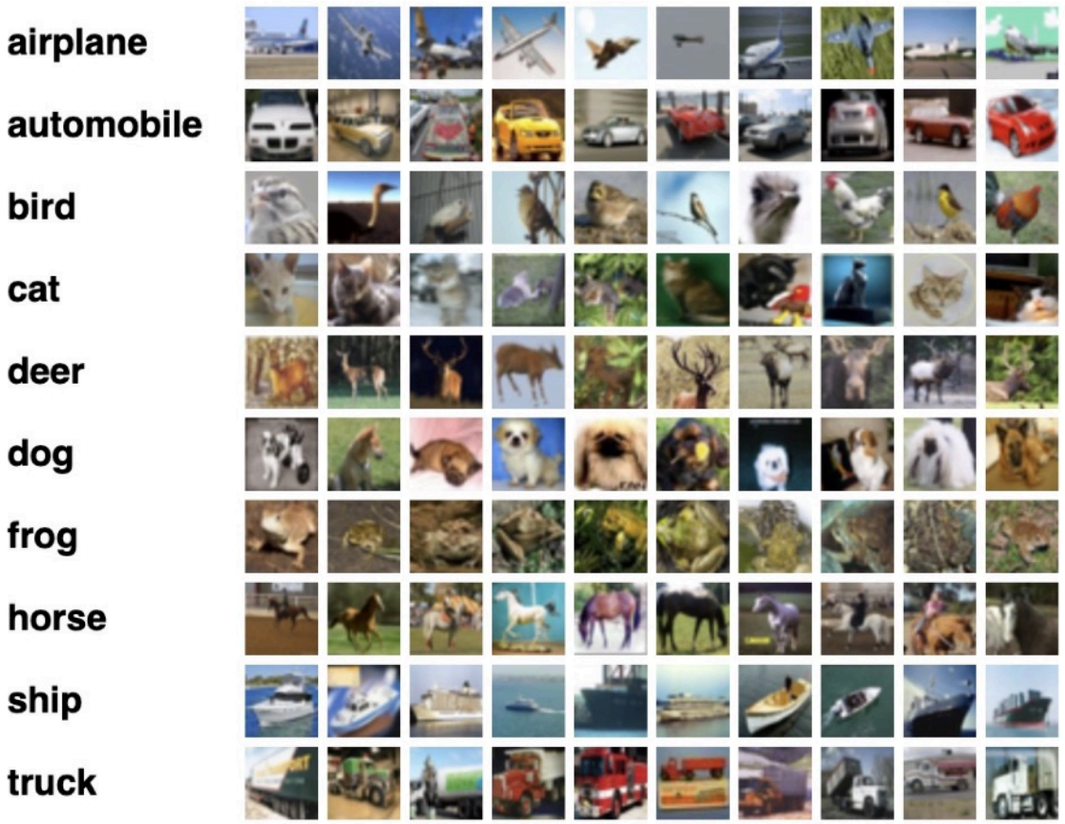
“Computer, recognise simple characters”

MNIST



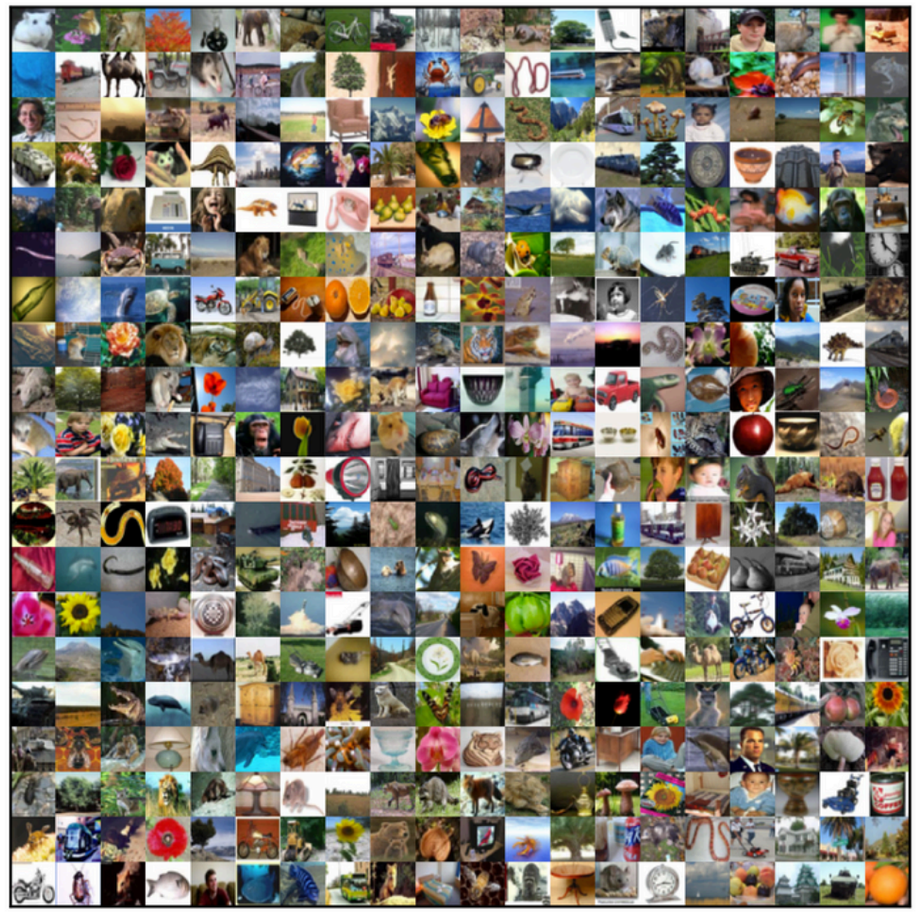
notMNIST

“Computer, recognise images”



CIFAR-10

60000 images, 10 classes



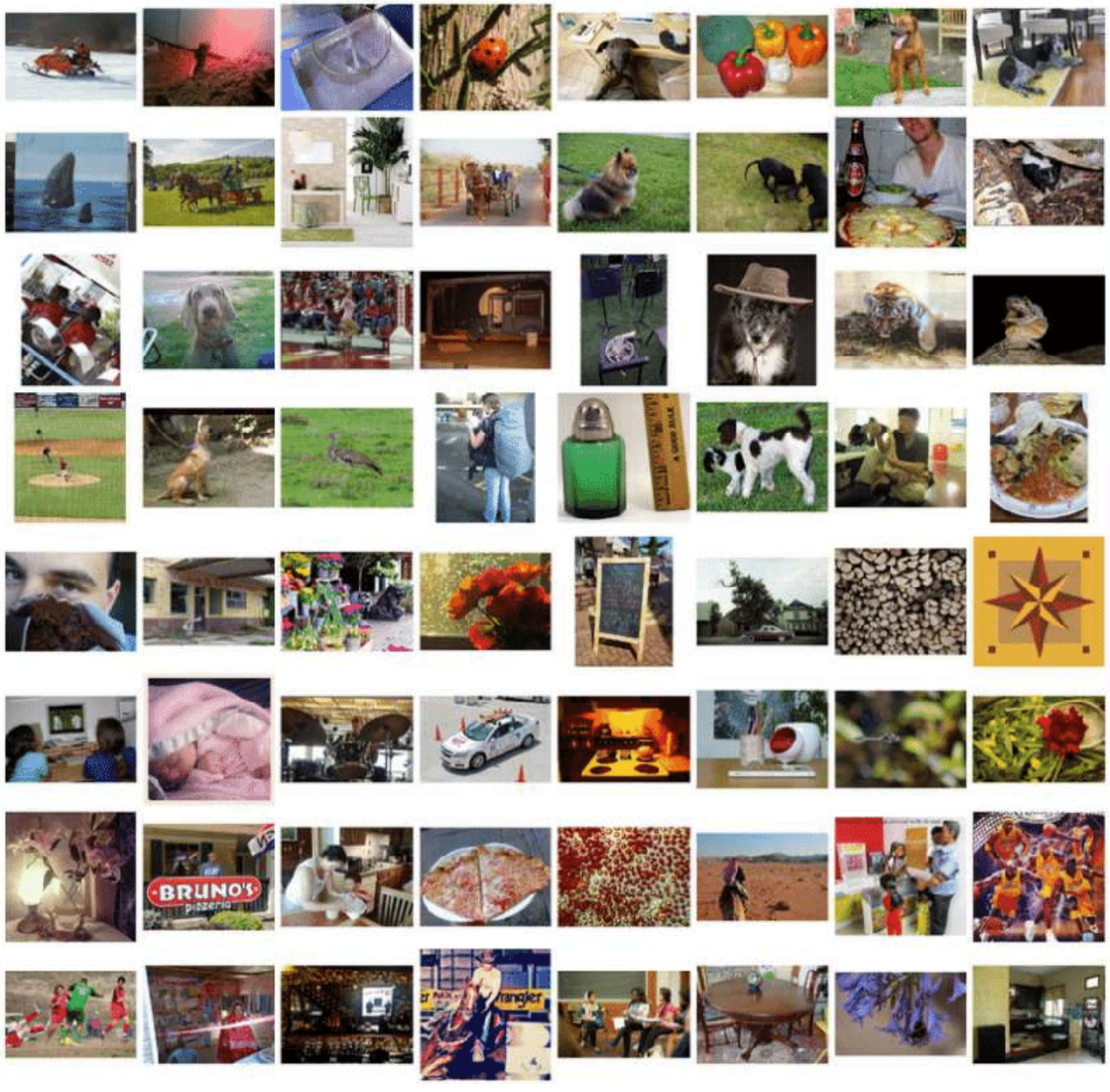
CIFAR-100

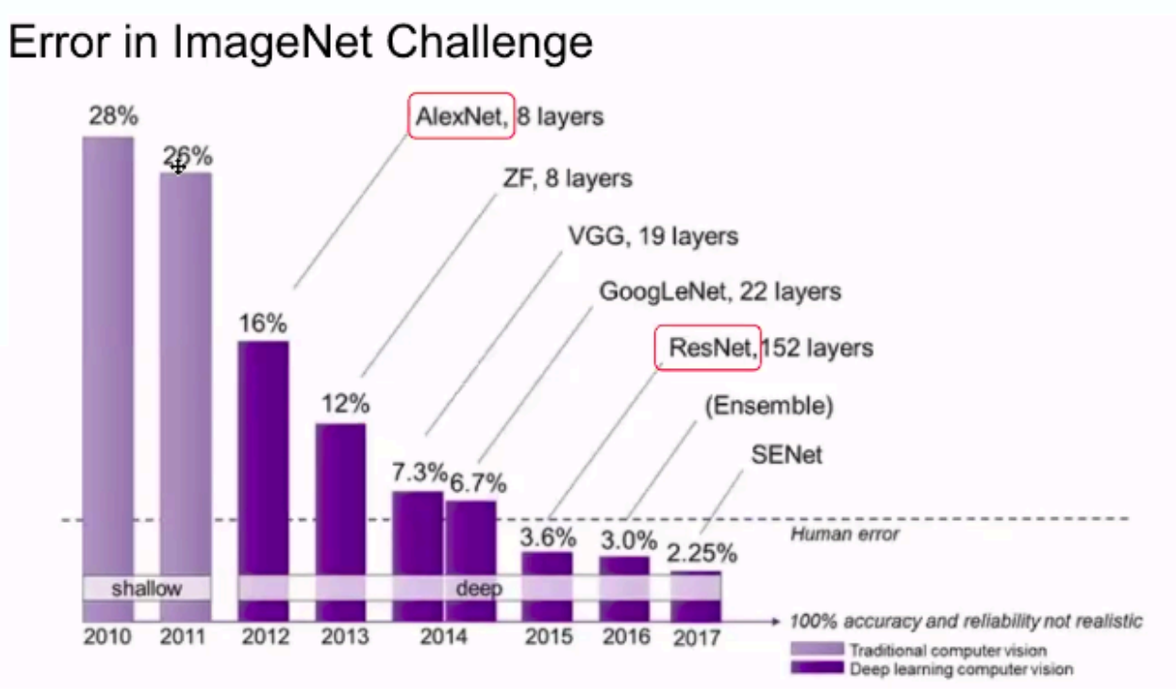
60000 images, 100 classes

ImageNet

From Wikipedia, the free encyclopedia

The **ImageNet** project is a large visual database designed for use in visual object recognition software research. More than 14 million[1][2] images have been hand-annotated by the project to indicate what objects are pictured and in at least one million of the images, bounding boxes are also provided.[3] ImageNet contains more than 20,000 categories[2] with a typical category, such as "balloon" or "strawberry", consisting of several hundred images.[4] The database of annotations of third-party image URLs is freely available directly from ImageNet, though the actual images are not owned by ImageNet.[5] Since 2010, the ImageNet project runs an annual software contest, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), where software programs compete to correctly classify and detect objects and scenes. The challenge uses a "trimmed" list of one thousand non-overlapping classes.[6]

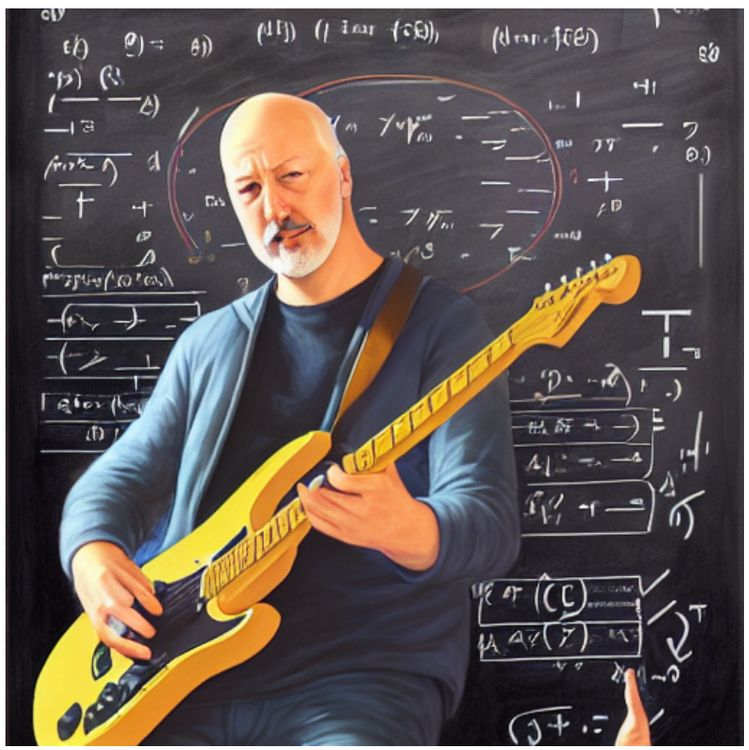




“Computer, drive my car”

“Computer, drive my car”

Computer, make a portrait of myself



Playing a Stratocaster guitar in front of a blackboard full of complex equations

As a mystical wise person, minimalist iconography, green and blue vibes

As obi-wan Kenobi in Star-Wars

How does this works ?

Tricks…. & convnets

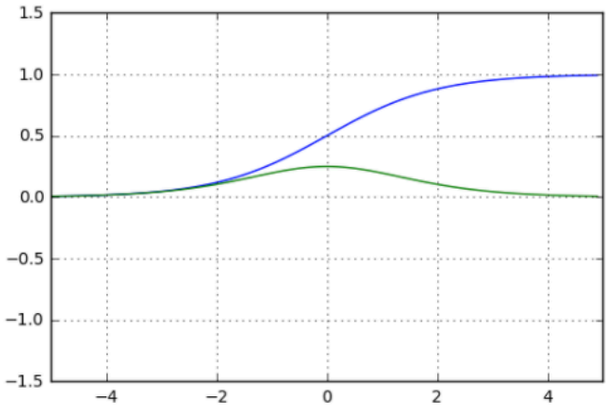
1: Tricks of the trade

1

Initialization of the weight

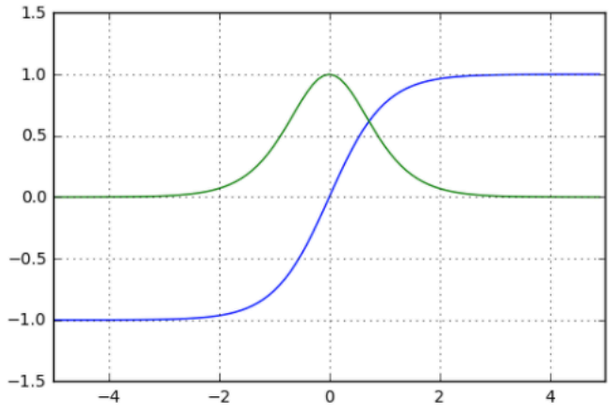
- Weights need to be small enough
  - around origin for symmetric activation functions (tanh, sigmoid)
    → stimulate activation functions near their linear regime
  - larger gradients → faster training
- Weights need to be large enough
  - otherwise signal is too weak for any serious learning

RELU prevent vanishing gradients
(but dead relus can exist! -> Leaky relu!)



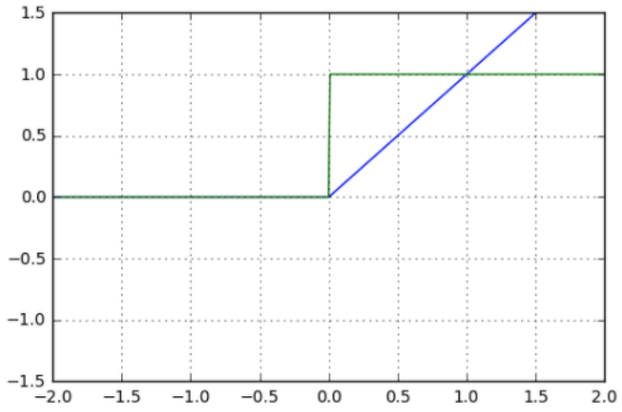
$$\text{sigm}(x) = \frac{1}{1+e^{-x}}$$

$$\text{sigm}'(x) = \text{sigm}(x)(1-\text{sigm}(x))$$



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

$$\tanh'(x) = 1-\tanh(x)^2$$



$$\text{relu}(x) = \max(0,x)$$

$$\text{relu}'(x) = 1_{x>0}$$

1

Initialization of the weight

Xavier Initialization

$$\mathcal{N}(0, \frac{2}{N_{in} + N_{out}})$$

glorot_uniform

```
glorot_uniform(seed=None)
```

Glorot uniform initializer, also called Xavier uniform initializer.

It draws samples from a uniform distribution within [-limit, limit] where `limit` is `sqrt(6 / (fan_in + fan_out))` where `fan_in` is the number of input units in the weight tensor and `fan_out` is the number of output units in the weight tensor.

Arguments

- **seed**: A Python integer. Used to seed the random generator.

Returns

1

Initialization of the weight

Kaiming-He initialization $\mathcal{N}(0, \frac{2}{N_{in}})$

* Scale the incoming weight to have a O(1) variable
* The factor 2 depends on activation: ReLUs ground to 0 the linear activation about half the Time -> Double weight variance for Relu to adapt

he_normal

```
he_normal(seed=None)
```

He normal initializer.

It draws samples from a truncated normal distribution centered on 0 with `stddev = sqrt(2 / fan_in)` where `fan_in` is the number of input units in the weight tensor.

Arguments

- **seed**: A Python integer. Used to seed the random generator.

Returns

1

Initialization of the weight

Kaiming-He initialization $\mathcal{N}(0, \frac{2}{N_{in}})$

The same type of reasoning can be applied to other activation functions

From `torch/nn/init.py`:

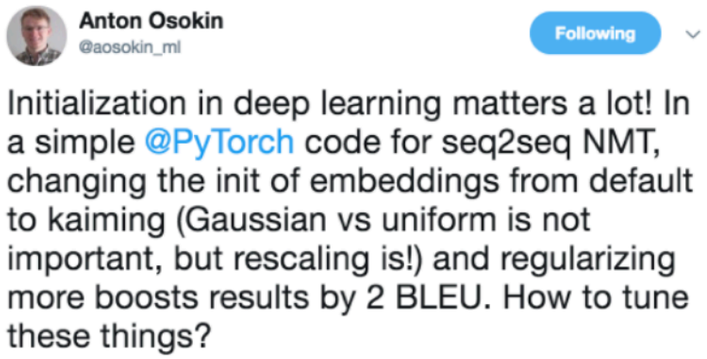
```python
def calculate_gain(nonlinearity, param=None):
    linear_fns = ['linear', 'conv1d', 'conv2d', 'conv3d', 'conv_transpose1d', 'conv_transpose2d', 'conv_transpose3d']
    if nonlinearity in linear_fns or nonlinearity == 'sigmoid':
        return 1
    elif nonlinearity == 'tanh':
        return 5.0 / 3
    elif nonlinearity == 'relu':
        return math.sqrt(2.0)
    elif nonlinearity == 'leaky_relu':
        if param is None:
            negative_slope = 0.01
        elif not isinstance(param, bool) and isinstance(param, int) or isinstance(param, float):
            # True/False are instances of int, hence check above
            negative_slope = param
        else:
            raise ValueError("negative_slope {} not a valid number".format(param))
        return math.sqrt(2.0 / (1 + negative_slope ** 2))
    else:
        raise ValueError("Unsupported nonlinearity {}".format(nonlinearity))
```

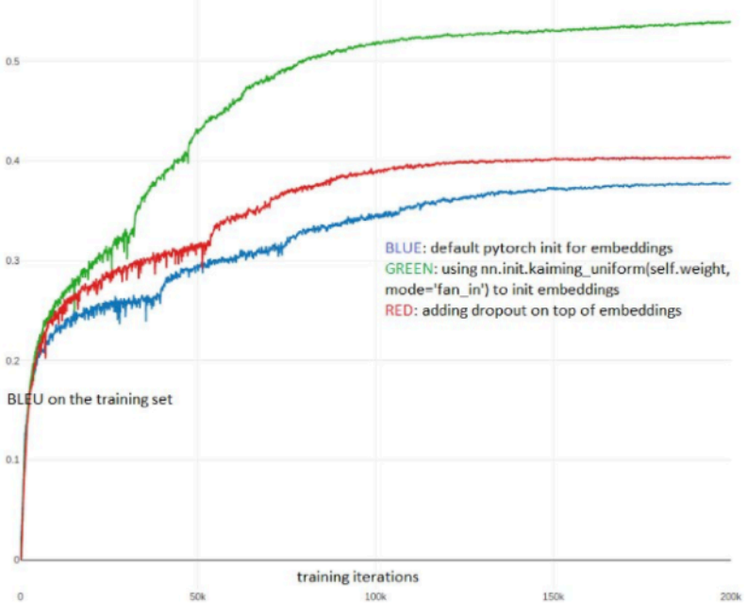
Weight initialization

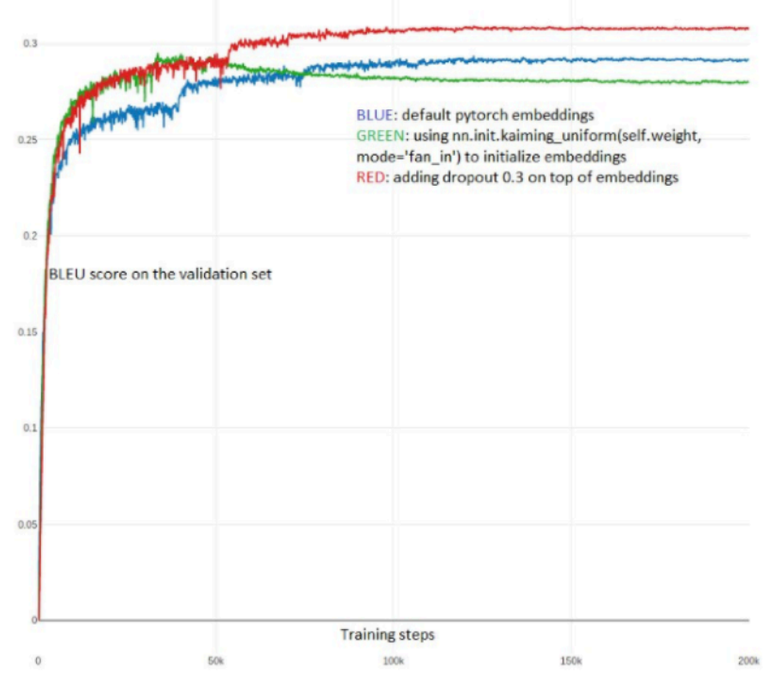
Does it actually matter that much?

Weight initialization

Does it actually matter that much?







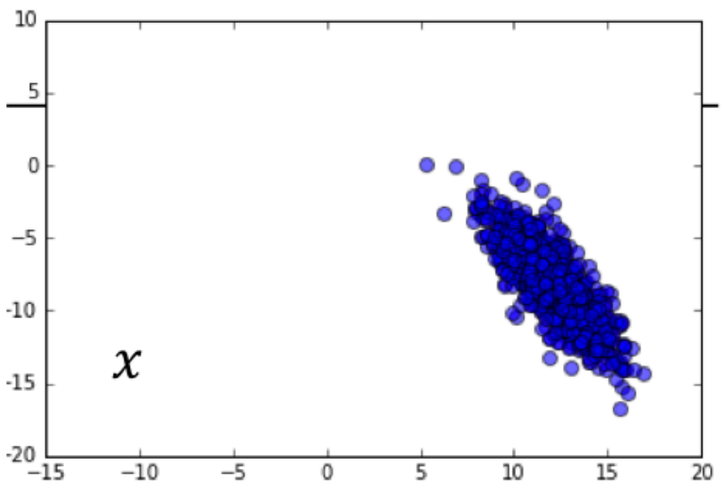
Data pre-processing

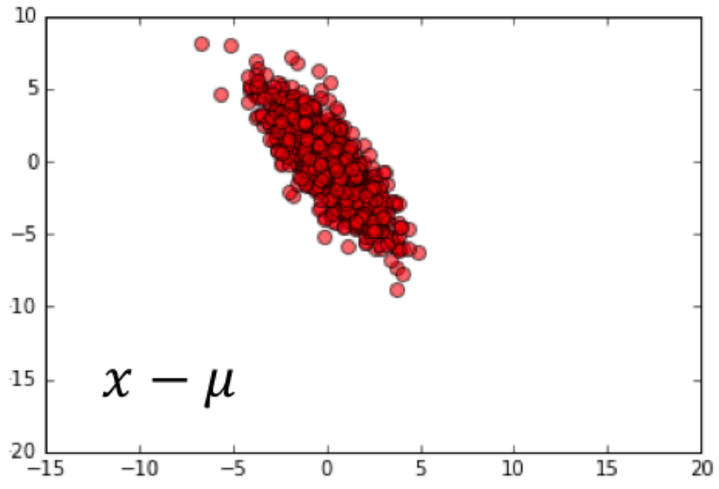
sklearn.preprocessing.StandardScaler

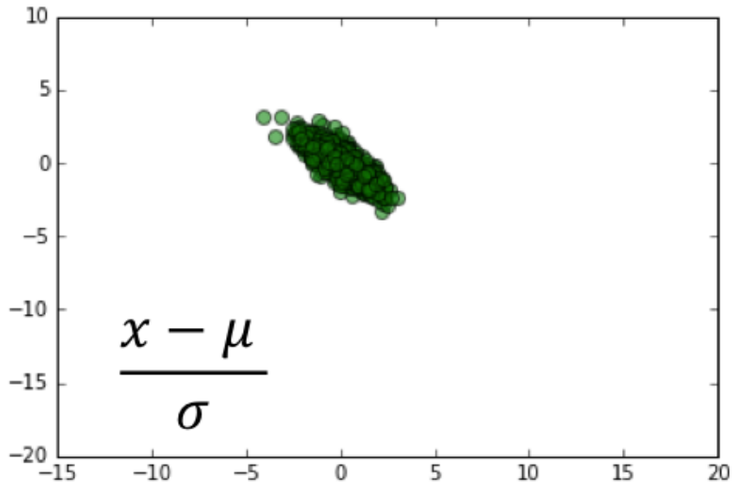
class sklearn.preprocessing.StandardScaler(*, *copy=True*, *with_mean=True*, *with_std=True*)

- Network is forced to find non-trivial correlations between inputs
- Decorrelated inputs → better optimization

- Input variables follow a more of less Gaussian distribution
- In practice:
  - compute mean and standard deviation
    - per pixel: (μ, σ^2)
    - per color channel:







2

Batch Normalization

```
from keras.layers.normalization import BatchNormalization
model = Sequential()
# think of this as the input layer
model.add(Dense(64, input_dim=16, init='uniform'))
model.add(BatchNormalization())
model.add(Activation('tanh'))
model.add(Dropout(0.5))
# think of this as the hidden layer
model.add(Dense(64, init='uniform'))
model.add(BatchNormalization())
model.add(Activation('tanh'))
model.add(Dropout(0.5))
# think of this as the output layer
model.add(Dense(2, init='uniform'))
model.add(BatchNormalization())
model.add(Activation('softmax'))
# optimiser and loss function
model.compile(loss='binary_crossentropy', optimizer=sgd)
```

During training, we normalise the activations of the previous layer for each batch:

We normalise in order to maintains the mean activation close to 0 and the activation standard deviation close to 1 befo

the activation function

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

3

What do you do when do not have enough data?

You create more!

Data augmentation

- Changing the pixels without changing the label
- Train on transformed data
- Widely used

Original

Flip

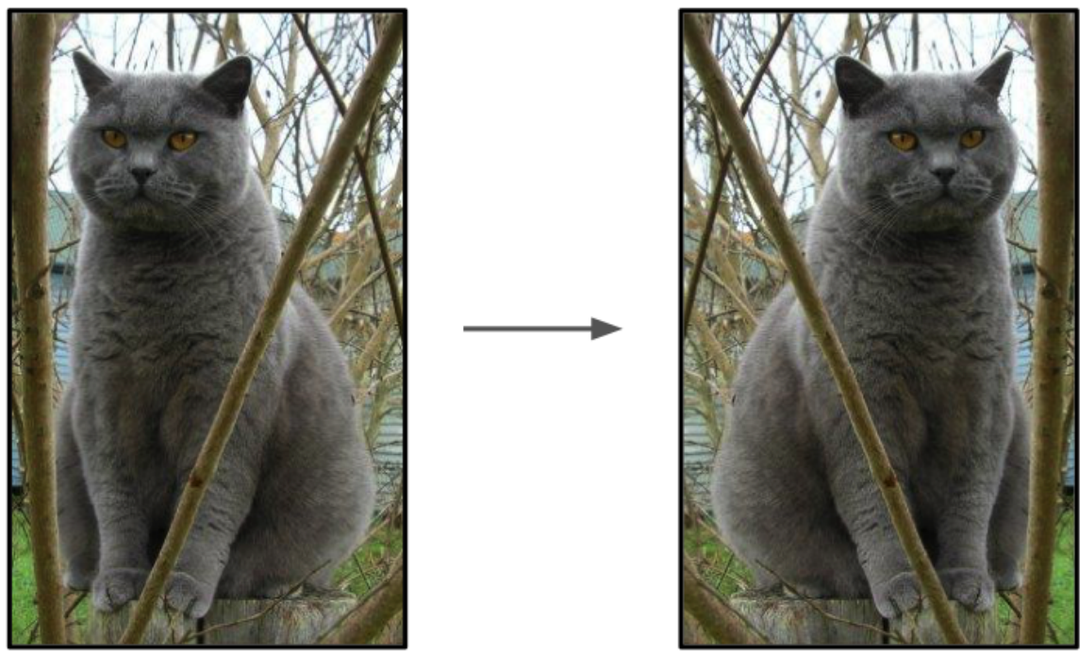
Random crop

Contrast

Tint

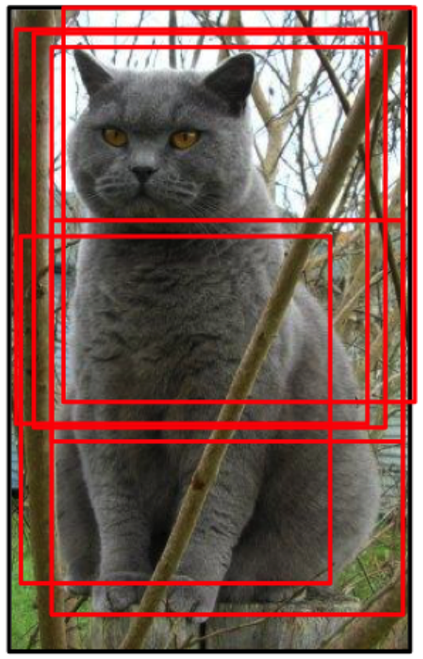
Data augmentation

Horizontal flips



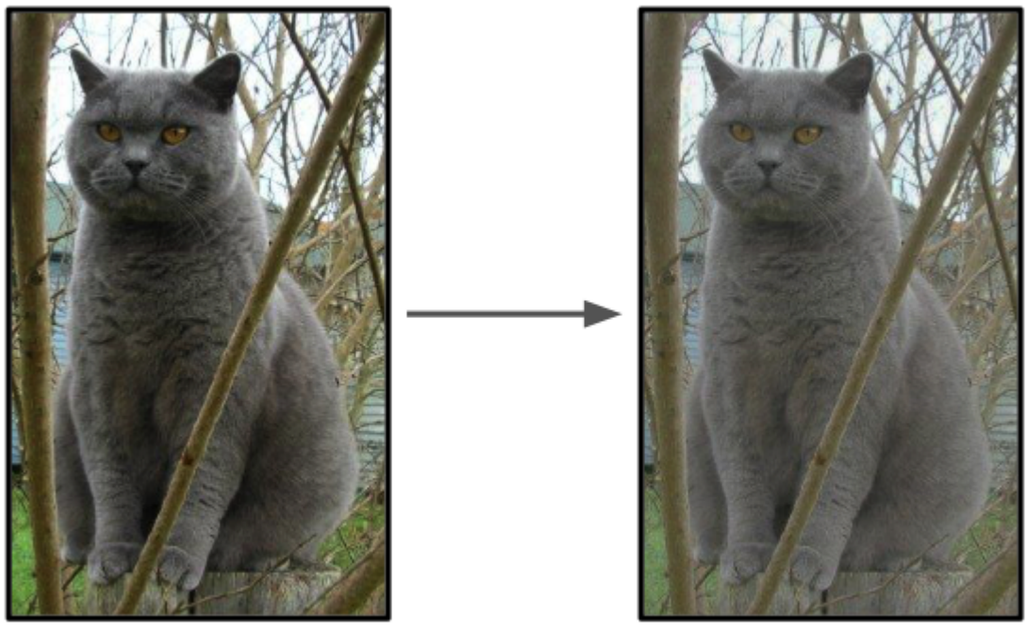
Data augmentation

Random crops/scales



Data augmentation

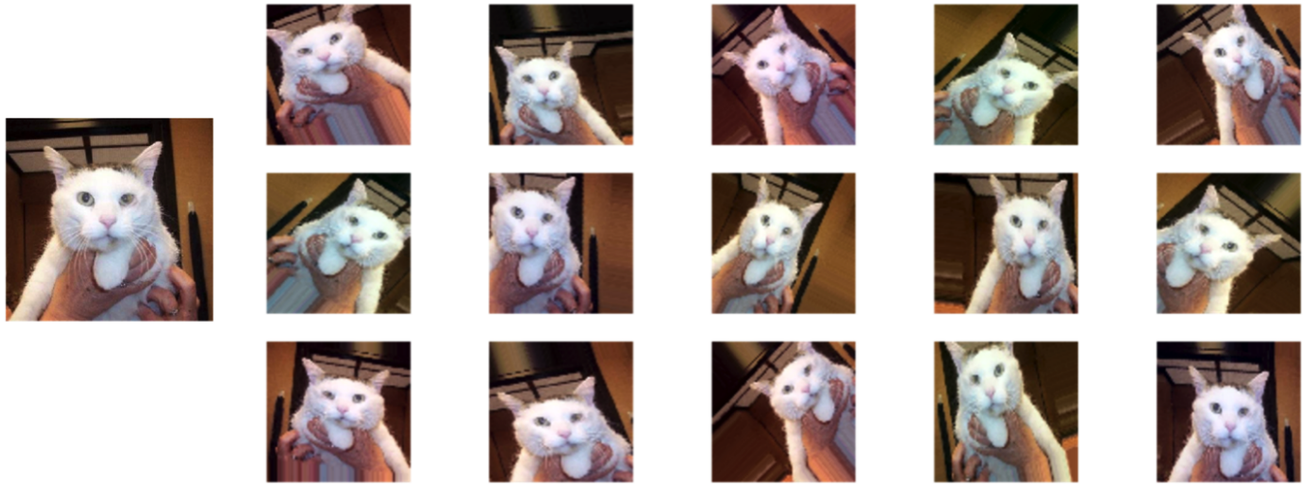
Color jitter



- randomly jitter color, brightness, contrast, etc.

Data augmentation

- Various techniques can be mixed
- Domain knowledge helps in finding new data augmentation techniques
- Very useful for small datasets



3

Data augmentation

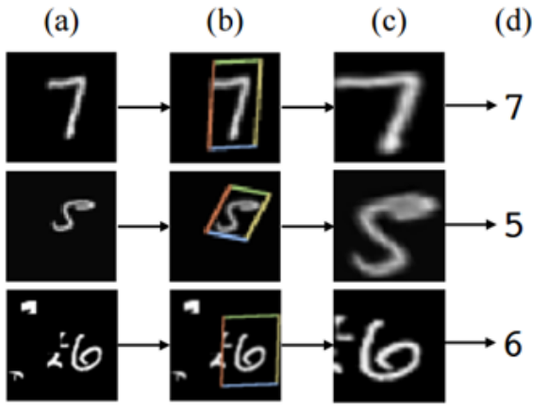
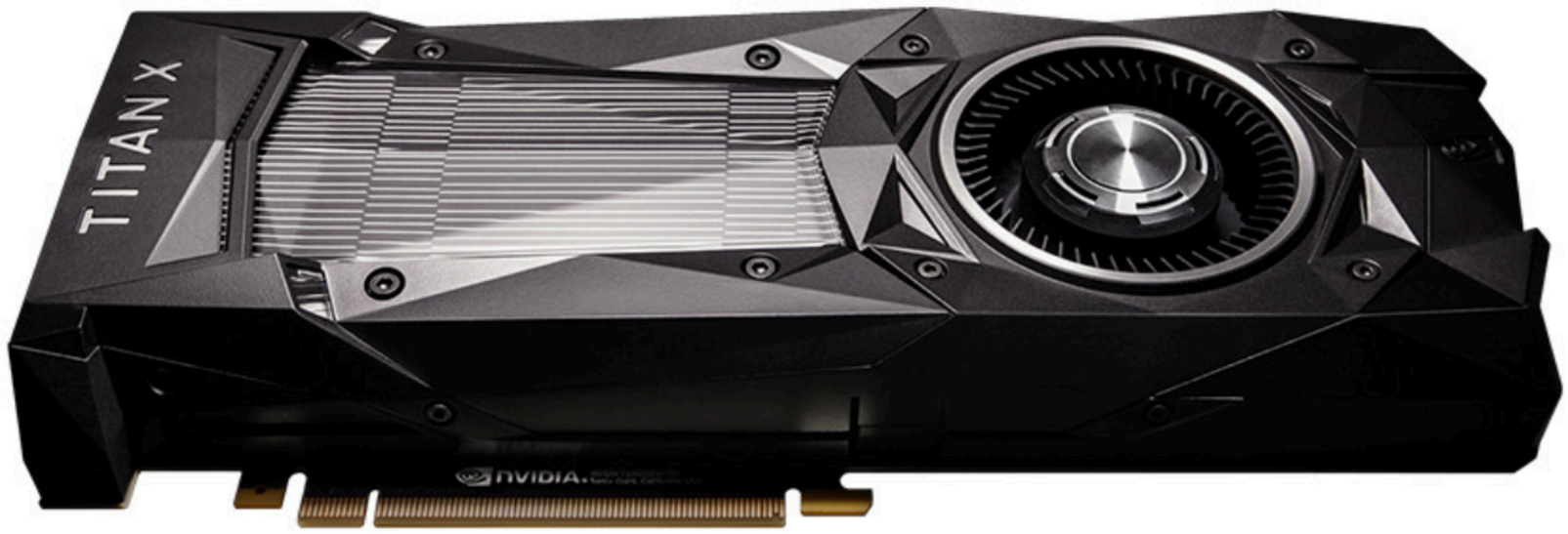


Figure 1: The result of using a spatial transformer as the first layer of a fully-connected network trained for distorted MNIST digit classification. (a) The input to the spatial transformer network is an image of an MNIST digit that is distorted with random translation, scale, rotation, and clutter. (b) The localisation network of the spatial transformer predicts a transformation to apply to the input image. (c) The output of the spatial transformer, after applying the transformation. (d) The classification prediction produced by the subsequent fully-connected network on the output of the spatial transformer. The spatial transformer network (a CNN including a spatial transformer module) is trained end-to-end with only class labels – no knowledge of the groundtruth transformations is given to the system.

Great overview of the function of a Spatial Tranfomer module

GPUs



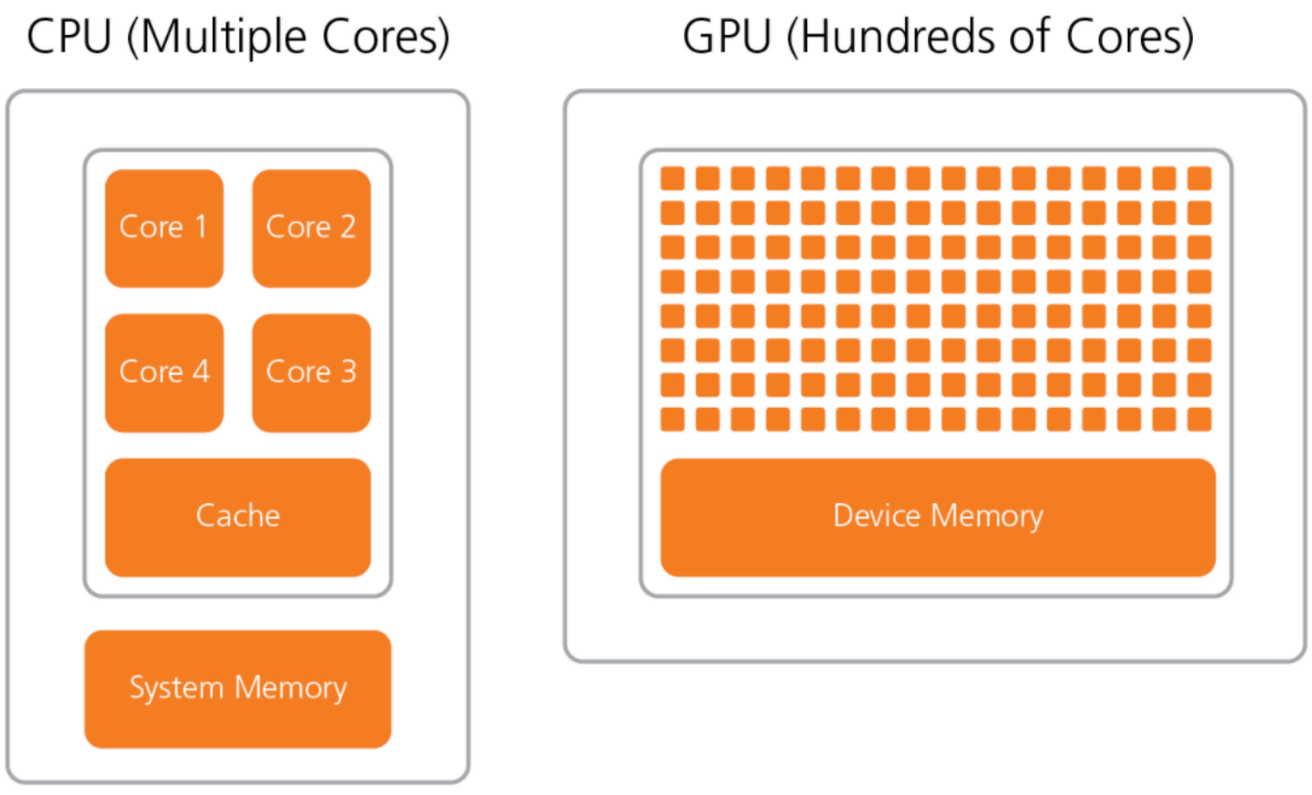
CPU vs GPU

CPU

GPU

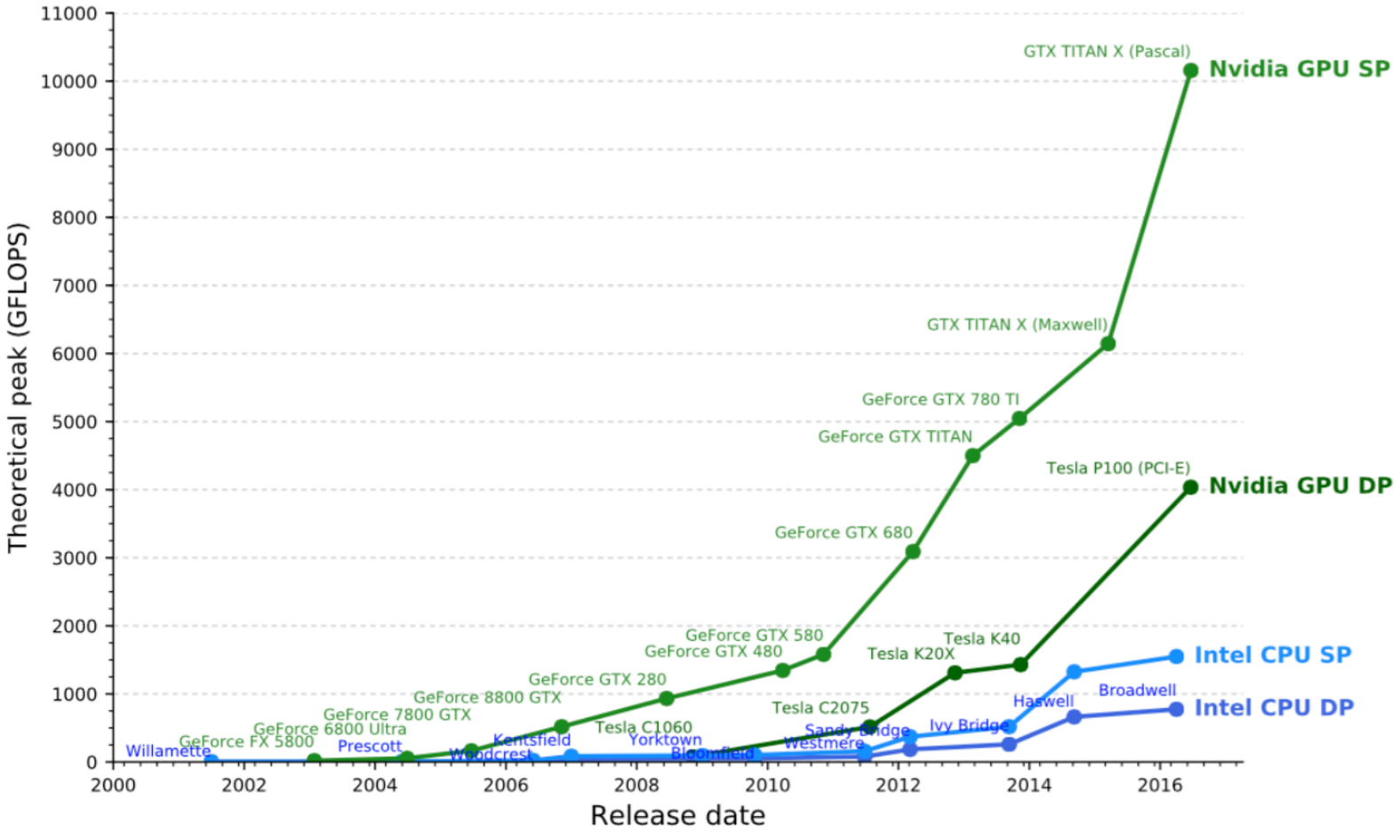
CPU vs GPU

- CPU:
  - fewer cores; each core is faster and more powerful
  - useful for sequential tasks
- GPU:
  - more cores; each core is slower and weaker
  - great for parallel tasks



CPU vs GPU

- SP = single precision, 32 bits / 4 bytes
- DP = double precision, 64 bits / 8 bytes



4

Market Summary > NVIDIA Corporation

303.90 USD

+280.56 (1,202.06%) ↑ past 5 years

NASDAQ: NVDA

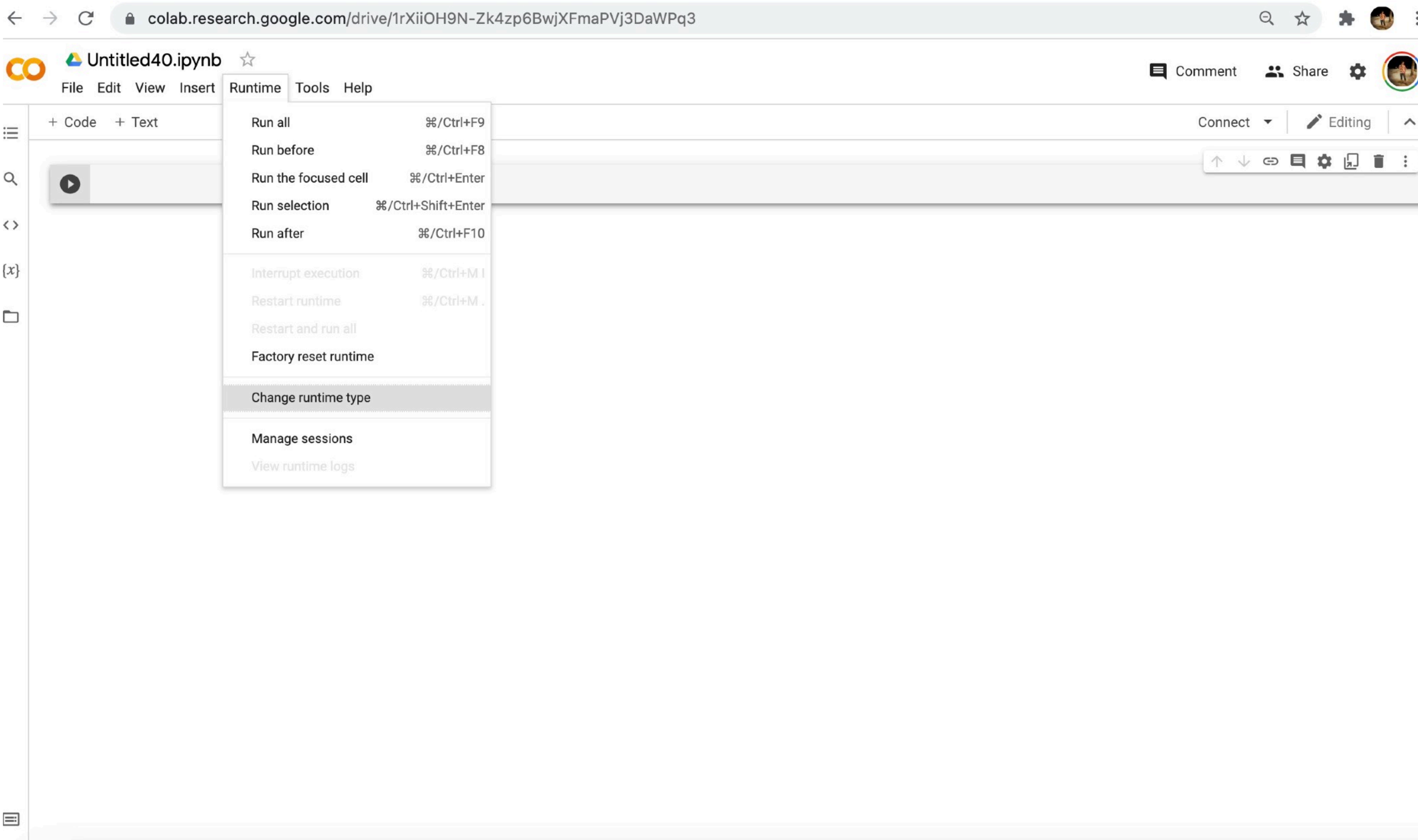
+ Follow

Closed: 15 Nov, 05:34 GMT-5 • Disclaimer

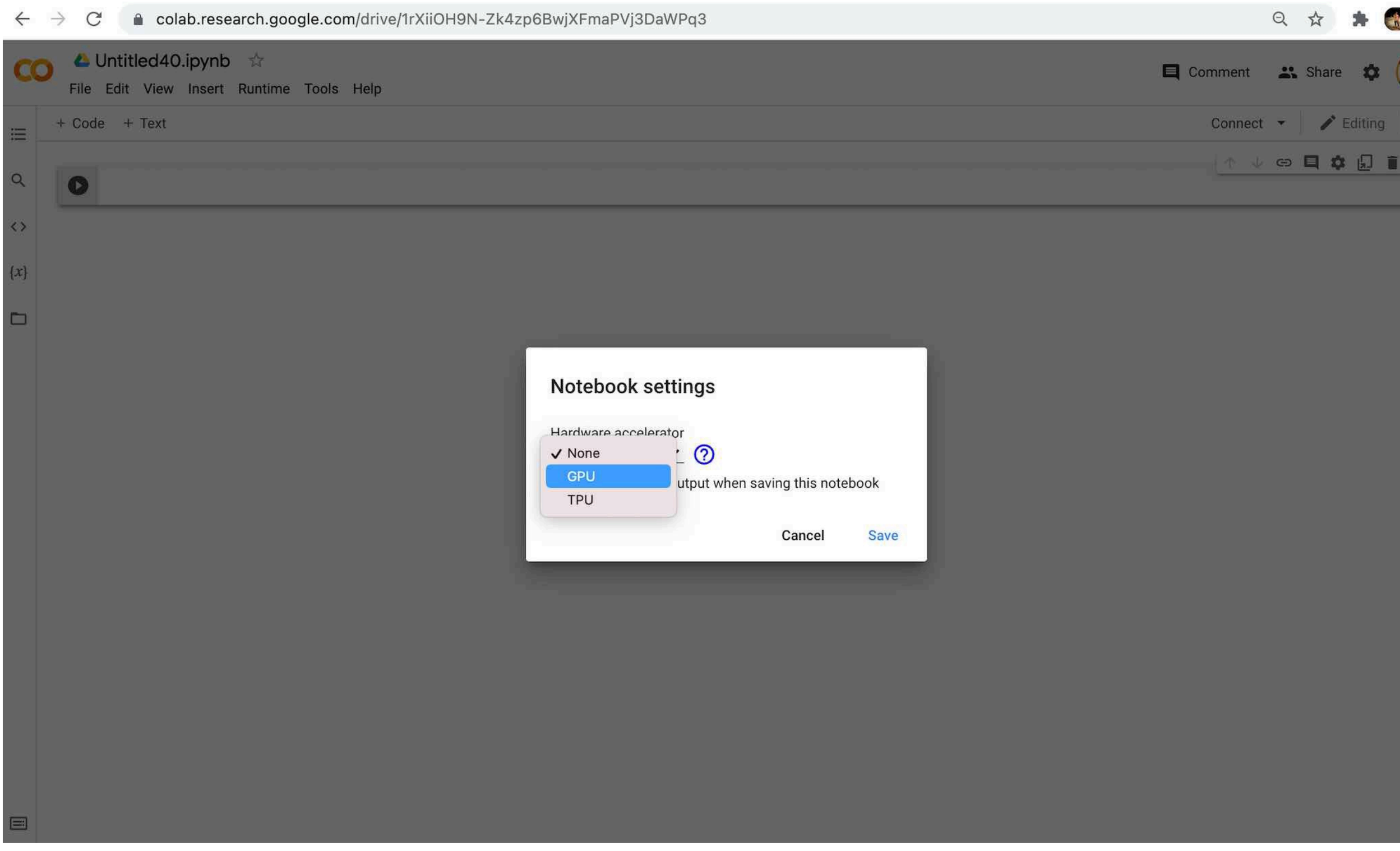
Pre-market 303.76 −0.14 (0.046%)

1D | 5D | 1M | 6M | YTD | 1Y | 5Y | Max

4


colab.research.google.com/drive/1rXiiOH9N-Zk4zp6BwjXFmaPVj3DaWPq3
Untitled40.ipynb
File Edit View Insert Runtime Tools Help
Comment
Share
+ Code + Text
Connect
Editing
Run all ⌘/Ctrl+F9
Run before ⌘/Ctrl+F8
Run the focused cell ⌘/Ctrl+Enter
Run selection ⌘/Ctrl+Shift+Enter
Run after ⌘/Ctrl+F10
Interrupt execution ⌘/Ctrl+M I
Restart runtime ⌘/Ctrl+M .
Restart and run all
Factory reset runtime
Change runtime type
Manage sessions
View runtime logs

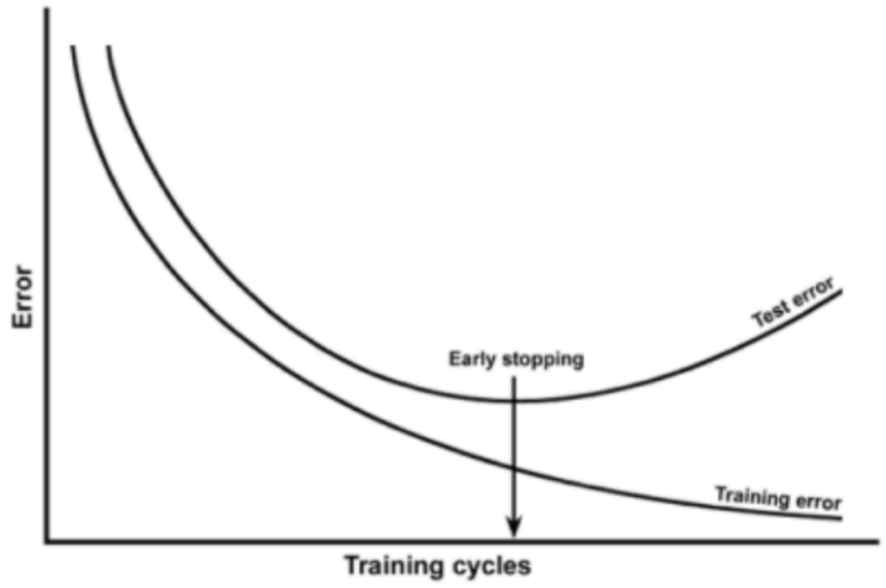
4


colab.research.google.com/drive/1rXiiOH9N-Zk4zp6BwjXFmaPVj3DaWPq3
Untitled40.ipynb
File Edit View Insert Runtime Tools Help
Comment
Share
+ Code
+ Text
Connect
Editing
Notebook settings
Hardware accelerator
None
GPU
TPU
utput when saving this notebook
Cancel
Save

2: Regularization

Early stopping

- To avoid overfitting another popular technique is early stopping
- Monitor performance on validation set
- Training the network will decrease training error, as well validation error (although with a slower rate usually)
- Stop when validation error starts increasing
  - most likely the network starts to overfit
  - use a *patience* term to let it degrade for a while and then stop



Remember this?

The linear model revisited: regularisation

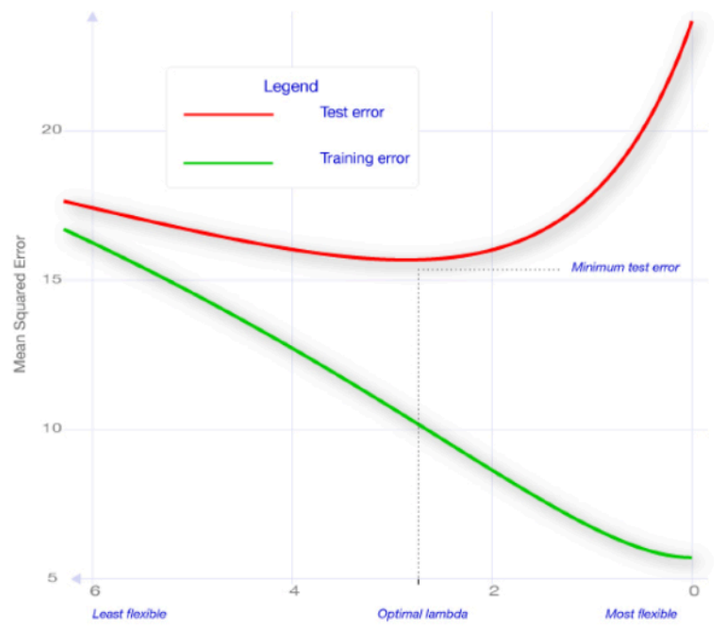
Replace *By*

$$\hat{\theta} = \text{argmin}(||\mathbf{Y} - \mathrm{A}\theta||_2^2) \qquad \hat{\theta} = \text{argmin}(||\mathbf{Y} - \mathrm{A}\theta||_2^2) + \mathrm{g}(\theta)$$

L2-regularization aka Tikhonov regularization
aka Ridge regression aka Weight decay

$$\hat{\theta} = \text{argmin}(||\mathbf{Y} - \mathrm{A}\theta||_2^2) + \Gamma||\theta||_2^2$$

Find the best Γ
using cross-validation



Weight Decay

= ℓ_2 *regularisation = Ridge = Tikhonov*

Regularization

L2 regularization:

$$\|\theta\|_2 = (w_1)^2 + (w_2)^2 + \dots$$

- New loss function to be minimized

$$\mathrm{L}'(\theta) = L(\theta) + \lambda \frac{1}{2}\|\theta\|_2 \quad \text{Gradient:} \quad \frac{\partial \mathrm{L}'}{\partial w} = \frac{\partial \mathrm{L}}{\partial w} + \lambda w$$

Update:

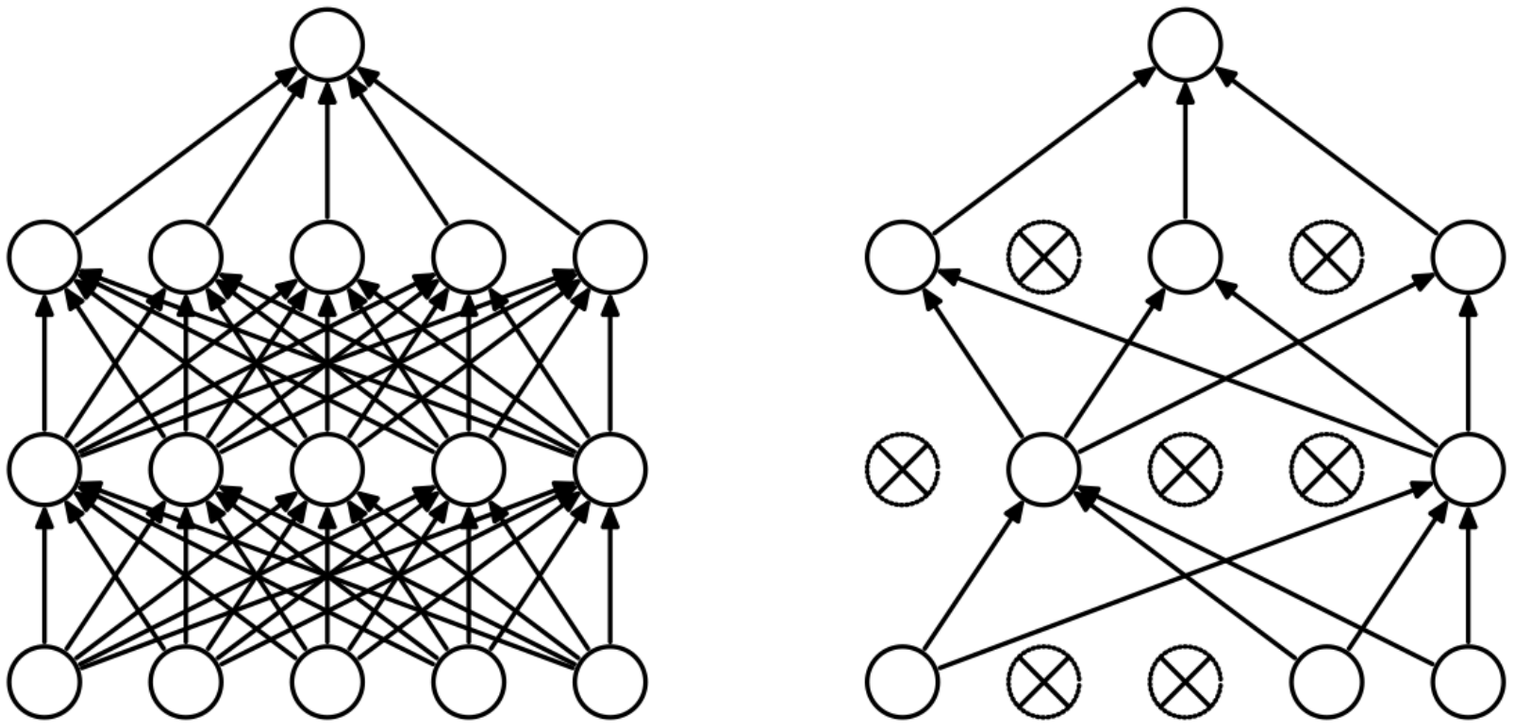
$$w^{t+1} \rightarrow w^t - \eta \frac{\partial \mathrm{L}'}{\partial w} = w^t - \eta\left(\frac{\partial \mathrm{L}}{\partial w} + \lambda w^t\right)$$

$$= (1 - \eta\lambda) w^t - \eta \frac{\partial \mathrm{L}}{\partial w}$$

Weight Decay

$(1-\eta\lambda)w^t$: Closer to zero

Dropout



(a) Standard Neural Net

(b) After applying dropout.

Each time we load an example into a minibatch, we randomly sample a different binary mask to apply to all of the input and hidden units in the network. The mask for each unit is sampled independently from all of the others

Dropout

Journal of Machine Learning Research 15 (2014) 1929-1958 Submitted 11/13; Published 6/14

Dropout: A Simple Way to Prevent Neural Networks from Overfitting

Nitish Srivastava NITISH@CS.TORONTO.EDU
Geoffrey Hinton HINTON@CS.TORONTO.EDU
Alex Krizhevsky KRIZ@CS.TORONTO.EDU
Ilya Sutskever ILYA@CS.TORONTO.EDU
Ruslan Salakhutdinov RSALAKHU@CS.TORONTO.EDU
Department of Computer Science
University of Toronto
10 Kings College Road, Rm 3302
Toronto, Ontario, M5S 3G4, Canada.

Editor: Yoshua Bengio

Abstract

Deep neural nets with a large number of parameters are very powerful machine learning systems. However, overfitting is a serious problem in such networks. Large networks are also slow to use, making it difficult to deal with overfitting by combining the predictions of many different large neural nets at test time. Dropout is a technique for addressing this problem. The key idea is to randomly drop units (along with their connections) from the neural network during training. This prevents units from co-adapting too much. During training, dropout samples from an exponential number of different "thinned" networks. At test time, it is easy to approximate the effect of averaging the predictions of all these thinned networks by simply using a single unthinned network that has smaller weights. This significantly reduces overfitting and gives major improvements over other regularization methods. We show that dropout improves the performance of neural networks on supervised learning

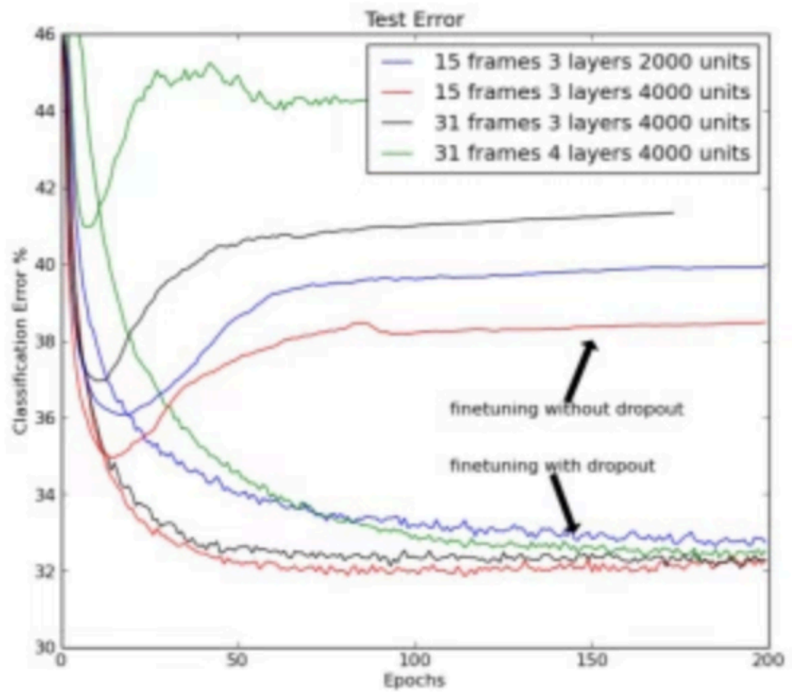
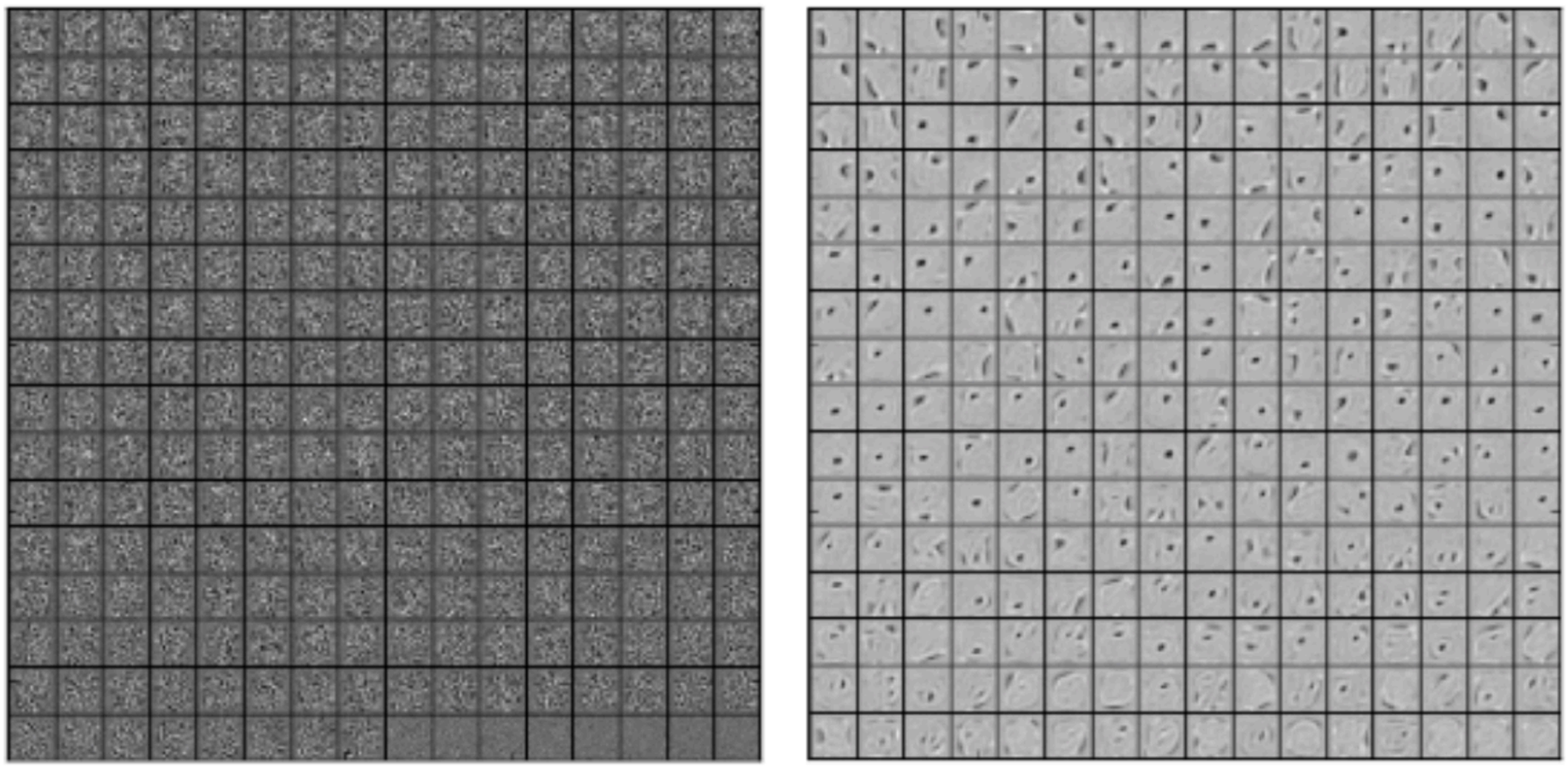


Fig. 2: The frame *classification* error rate on the core test set of the TIMIT benchmark. Comparison of standard and dropout finetuning for different network architectures. Dropout of 50% of the hidden units and 20% of the input units improves classification.

Dropout

Features learned on MNIST with one hidded layer autoencoders having 256 rectified linear units



(a) Without dropout

(b) Dropout with $p = 0.5$.

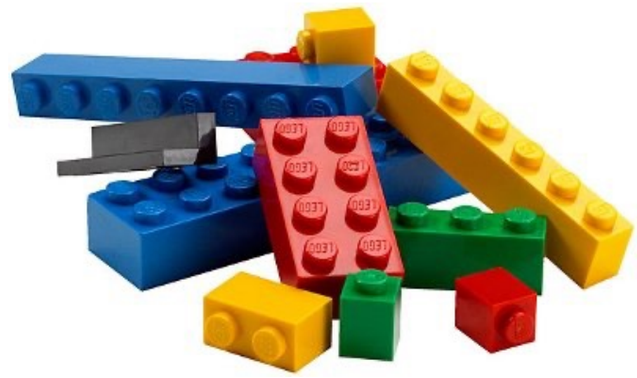
Easy to implement with pytorch:
This is just another layer!

```
class Net(nn.Module):
  def __init__(self):
    super(Net,self).__init__()
    self.conv1=nn.Conv2d(1,32,3,1)
    self.conv1_bn=nn.BatchNorm2d(32)

    self.conv2=nn.Conv2d(32,64,3,1)
    self.conv2_bn=nn.BatchNorm2d(64)

    self.dropout1=nn.Dropout(0.25)

    self.fc1=nn.Linear(9216,128)
    self.fc1_bn=nn.BatchNorm1d(128)

    self.fc2=nn.Linear(128,10)
  def forward(self,x):
    x=self.conv1(x)
    x=F.relu(self.conv1_bn(x))

    x=self.conv2(x)
    x=F.relu(self.conv2_bn(x))

    x=F.max_pool2d(x,2)
    x=self.dropout1(x)

    x=torch.flatten(x,1)

    x=self.fc1(x)
    x=F.relu(self.fc1_bn(x))

    x=self.fc2(x)
    output=F.log_softmax(x,dim=1)
    return output
```

Batchnorm

Dropout

Playing Lego



```
class LeNet(Module):
        def __init__(self, numChannels, classes):
                # call the parent constructor
                super(LeNet, self).__init__()

                # initialize first set of CONV => RELU => POOL layers
                self.conv1 = Conv2d(in_channels=numChannels, out_channels=20,
                        kernel_size=(5, 5))
                self.relu1 = ReLU()
                self.maxpool1 = MaxPool2d(kernel_size=(2, 2), stride=(2, 2))

                # initialize second set of CONV => RELU => POOL layers
                self.conv2 = Conv2d(in_channels=20, out_channels=50,
                        kernel_size=(5, 5))
                self.relu2 = ReLU()
                self.maxpool2 = MaxPool2d(kernel_size=(2, 2), stride=(2, 2))

                # initialize first (and only) set of FC => RELU layers
                self.fc1 = Linear(in_features=800, out_features=500)
                self.relu3 = ReLU()

                # initialize our softmax classifier
                self.fc2 = Linear(in_features=500, out_features=classes)
                self.logSoftmax = LogSoftmax(dim=1)
```

2007 NIPS Tutorial on:

Deep Belief Nets

Geoffrey Hinton
Canadian Institute for Advanced Research
&
Department of Computer Science
University of Toronto

How many layers should we use and how wide should they be?

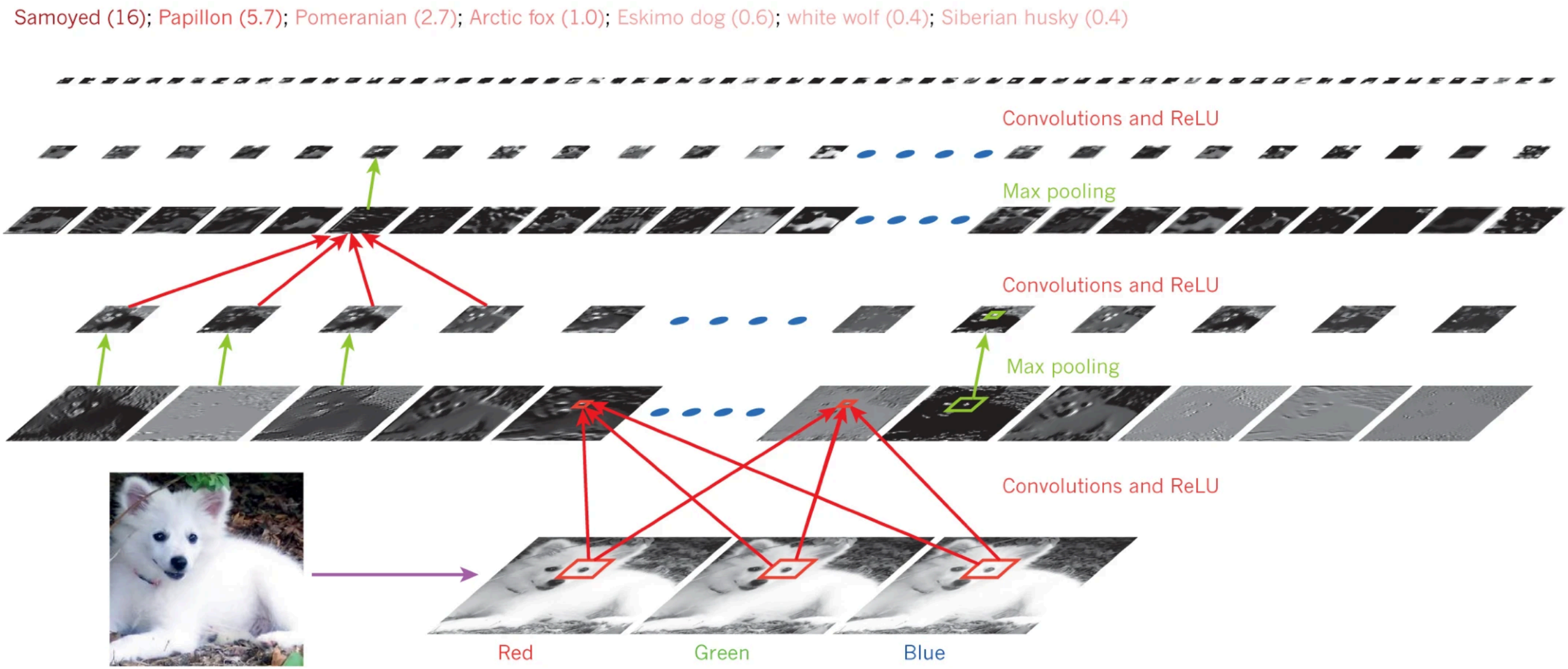
(I am indebted to Karl Rove for this slide)

- How many lines of code should an AI program use and how long should each line be?
  - This is obviously a silly question.
- Deep belief nets give the creator a lot of freedom.
  - How best to make use of that freedom depends on the task.
  - With enough narrow layers we can model any distribution over binary vectors (Sutskever & Hinton, 2007)
- If freedom scares you, stick to convex optimization of shallow models that are obviously inadequate for doing Artificial Intelligence.

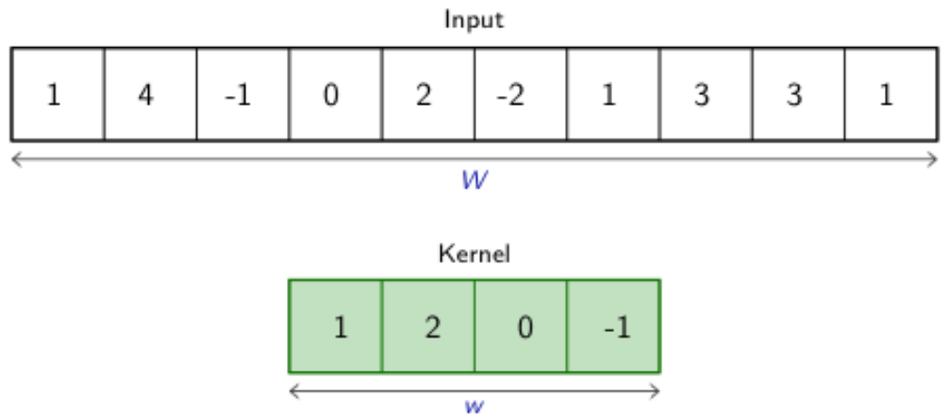
3: Special Layers

Convolutional and pooling layers

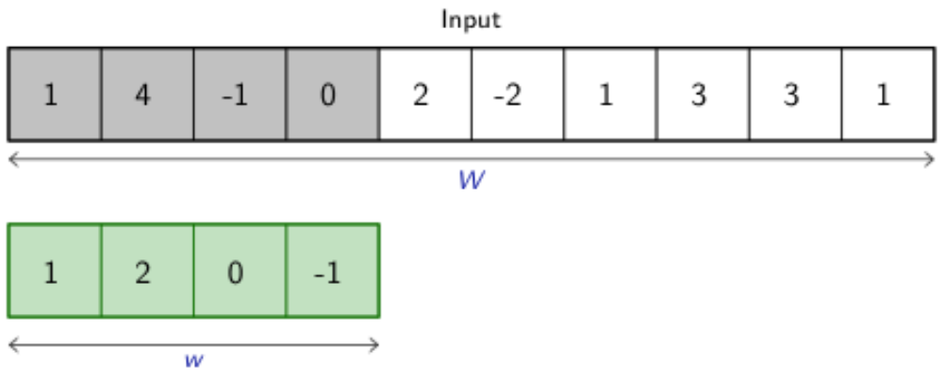
Fundamental for images & sounds!



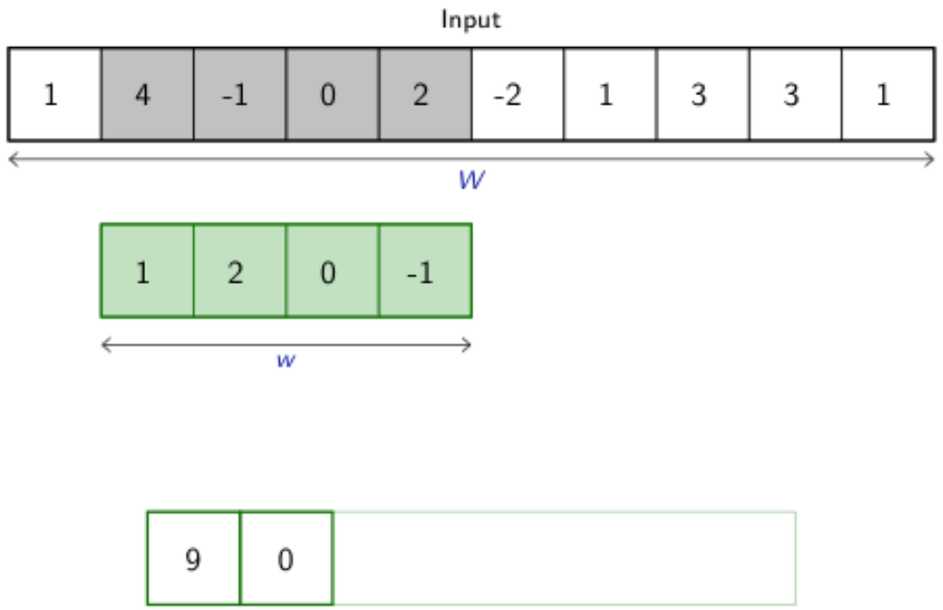
Convolution 1d



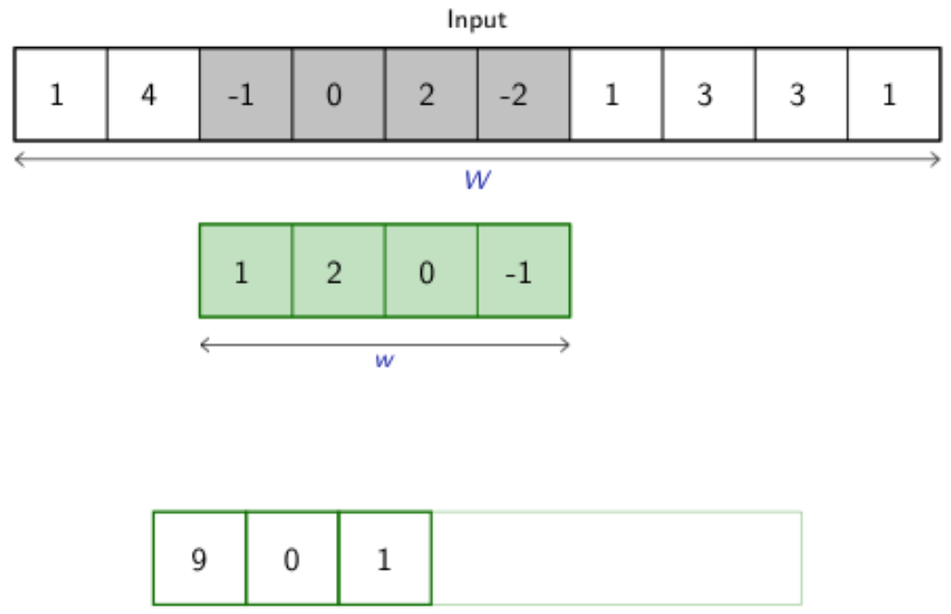
Convolution 1d



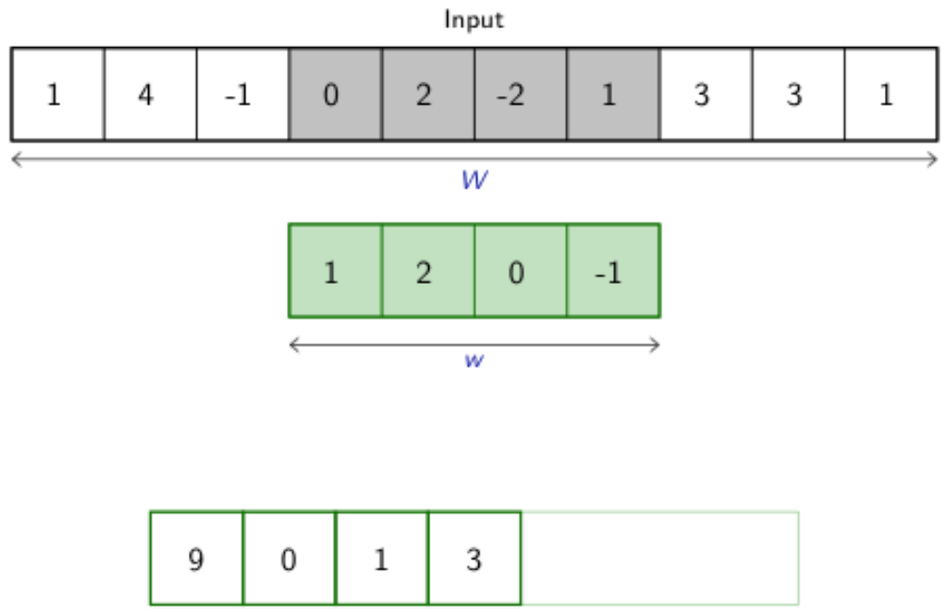
Convolution 1d



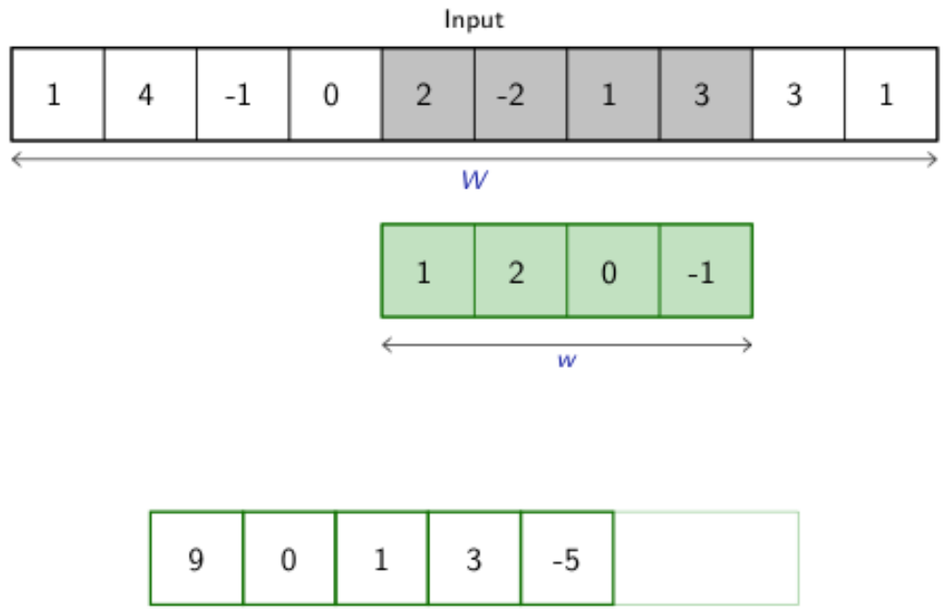
Convolution 1d



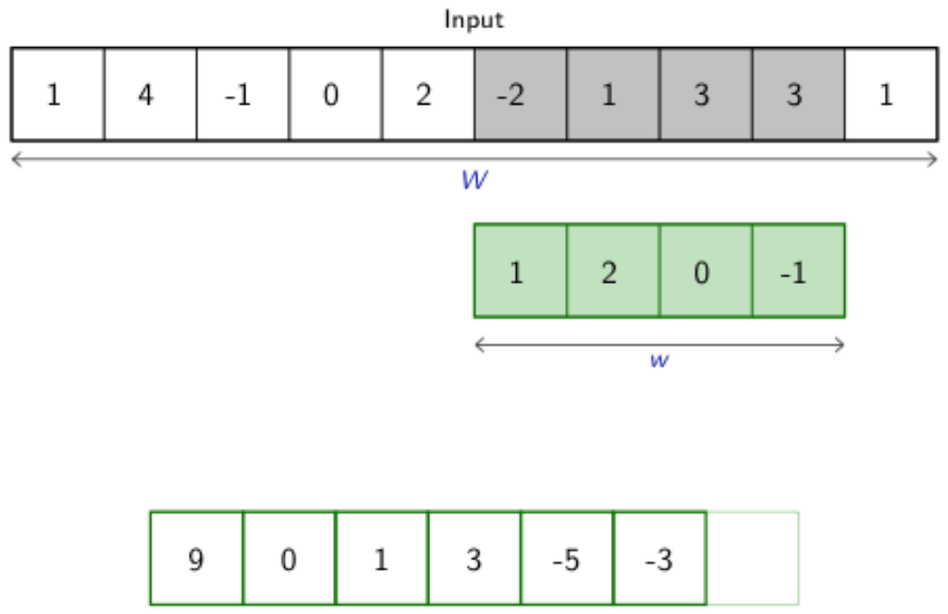
Convolution 1d



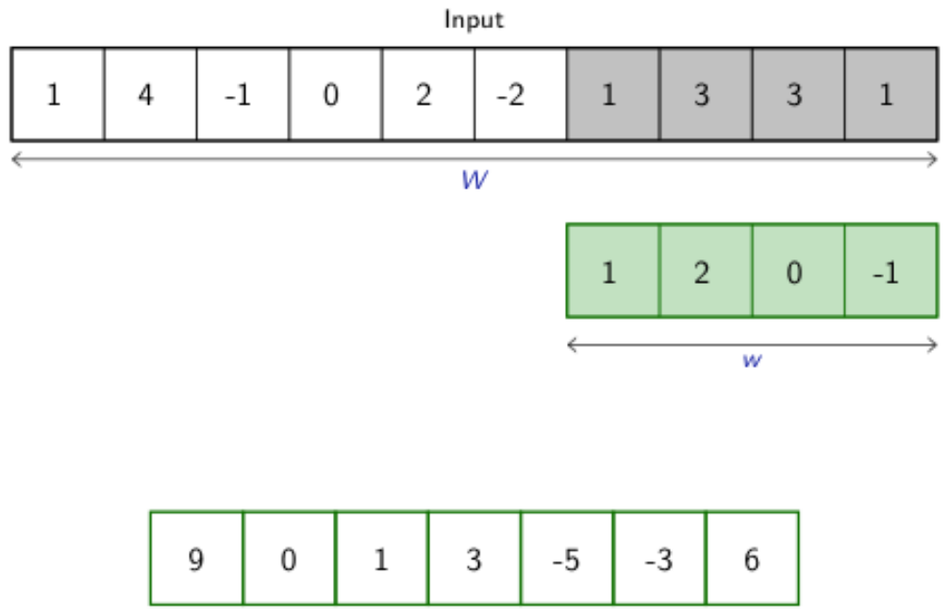
Convolution 1d



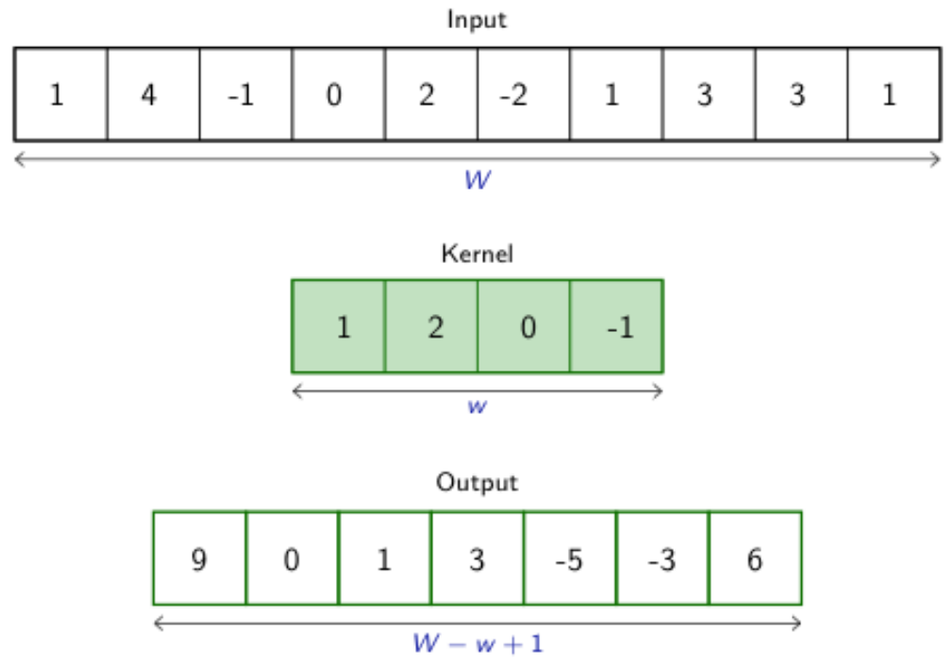
Convolution 1d



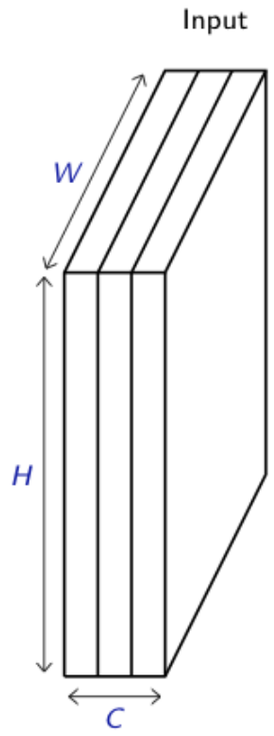
Convolution 1d



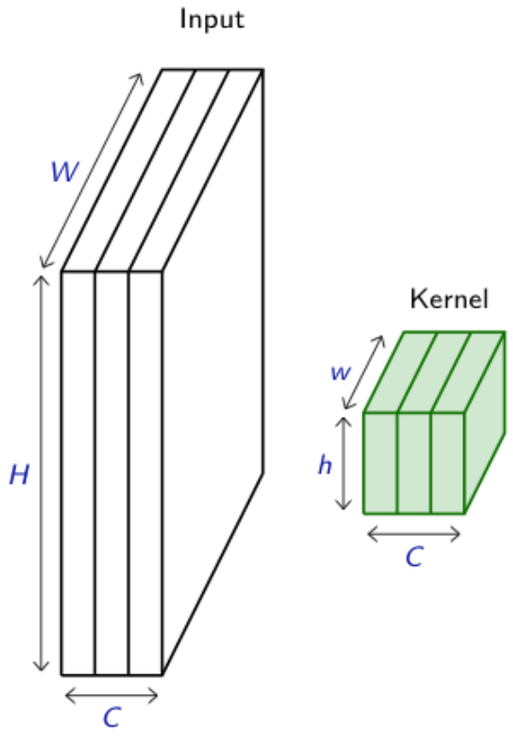
Convolution 1d



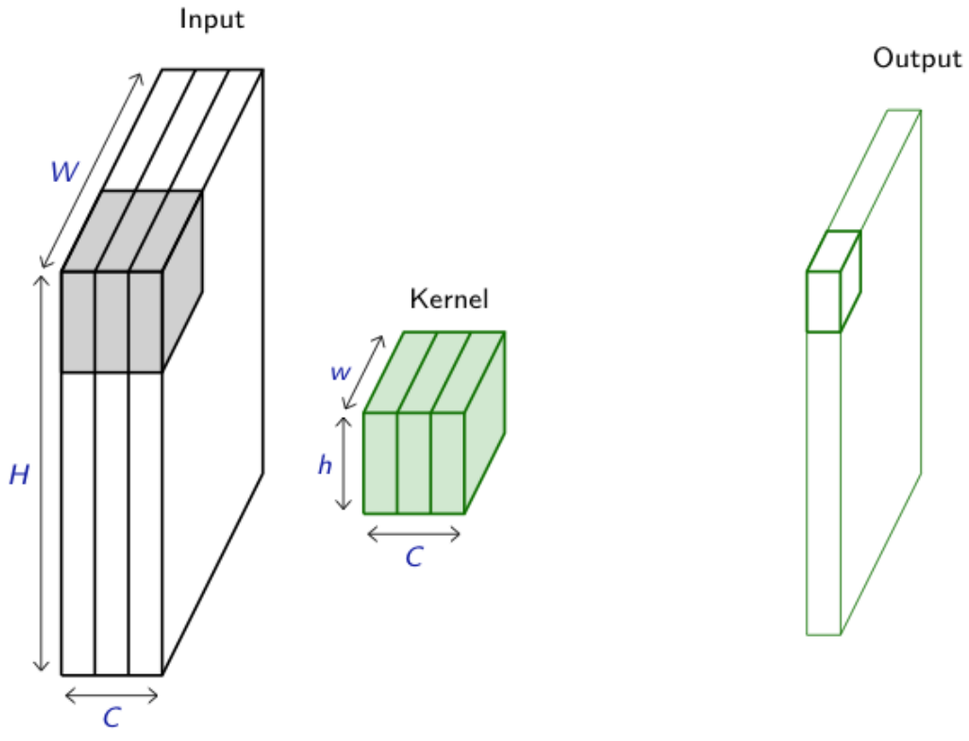
Convolution 2d



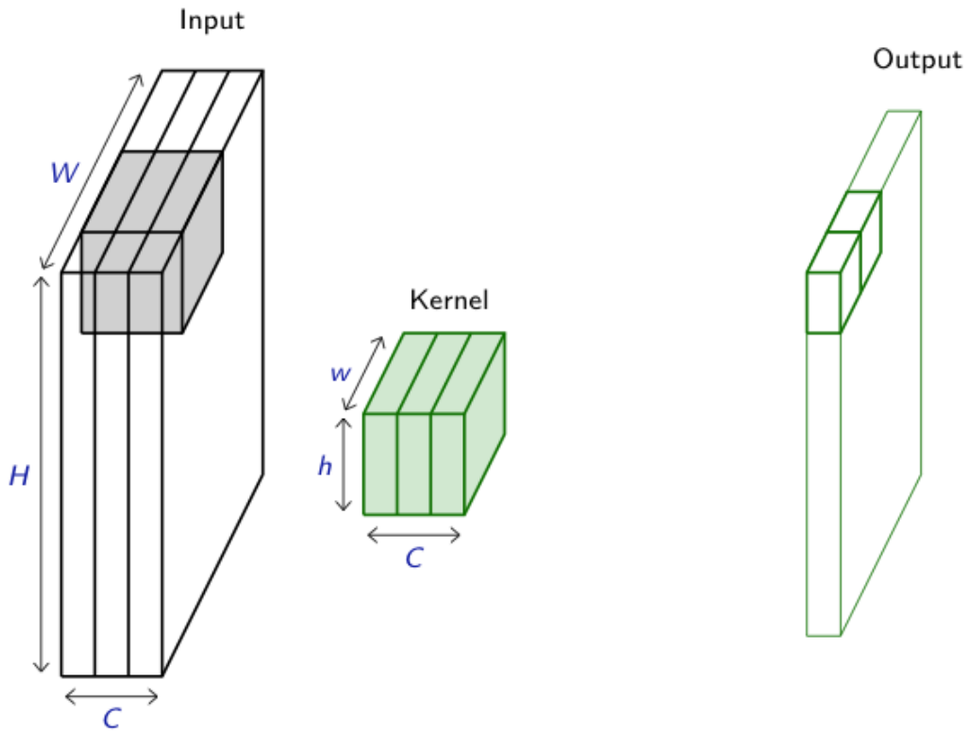
Convolution 2d



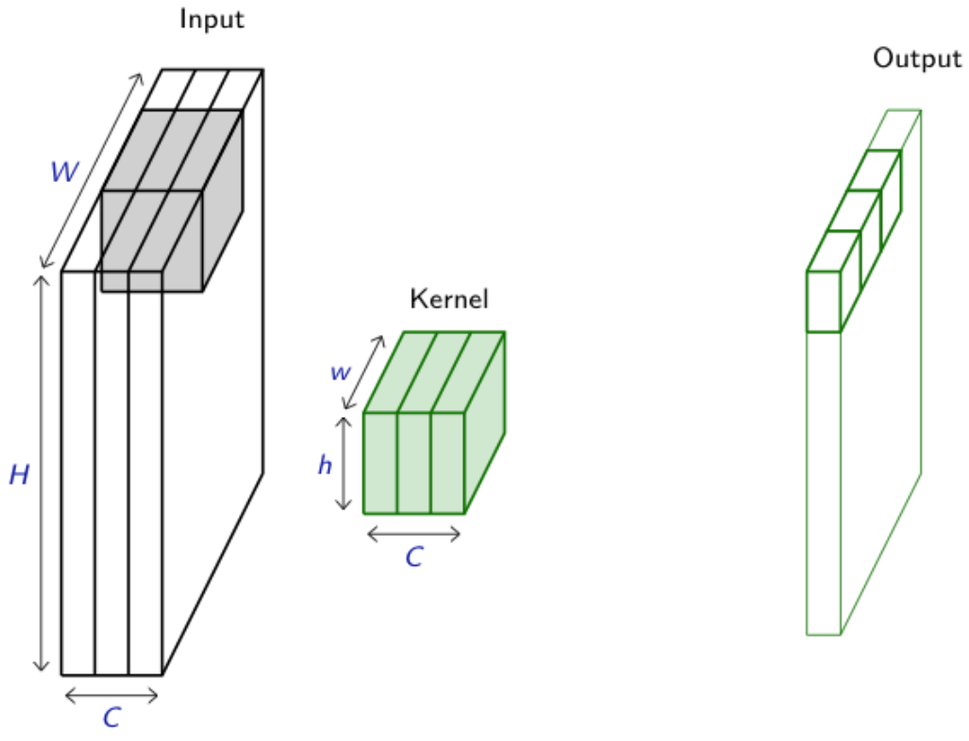
Convolution 2d



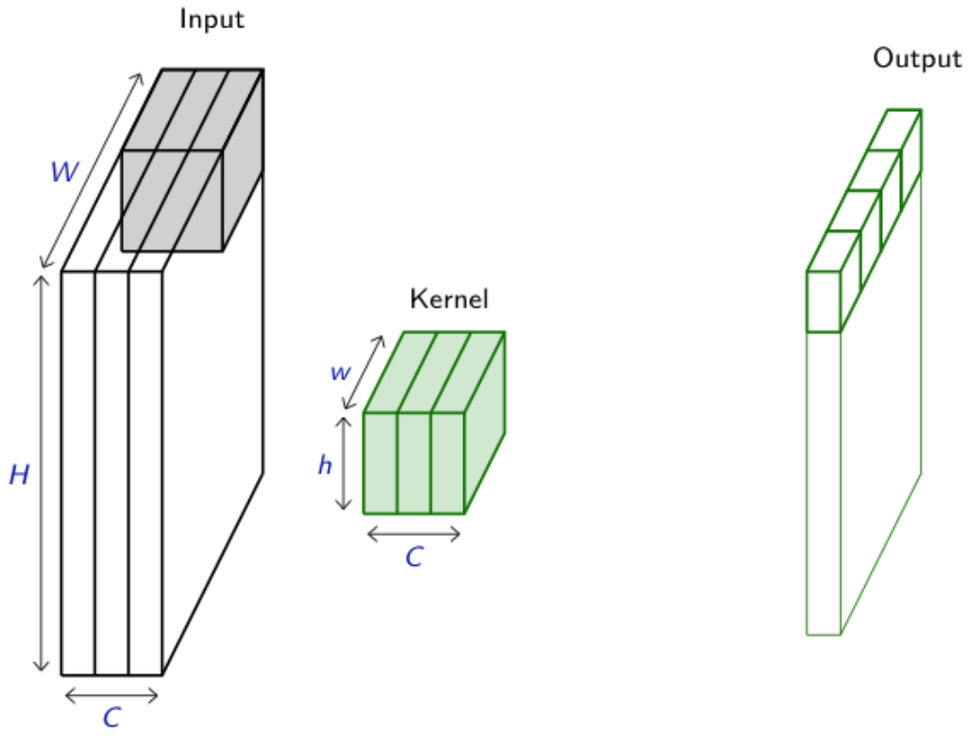
Convolution 2d



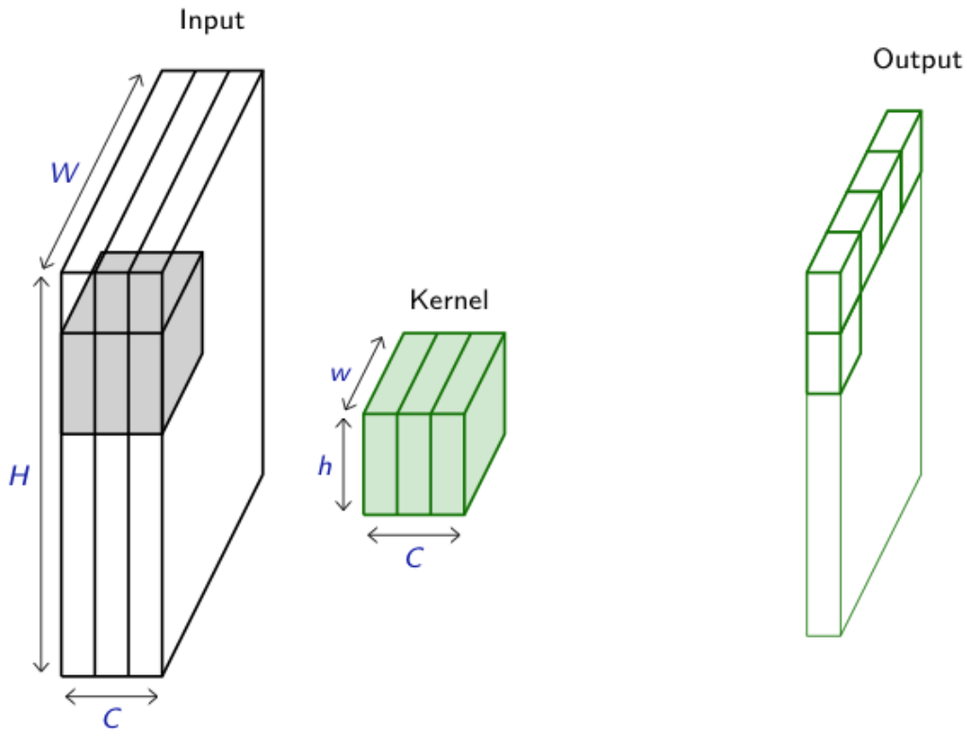
Convolution 2d



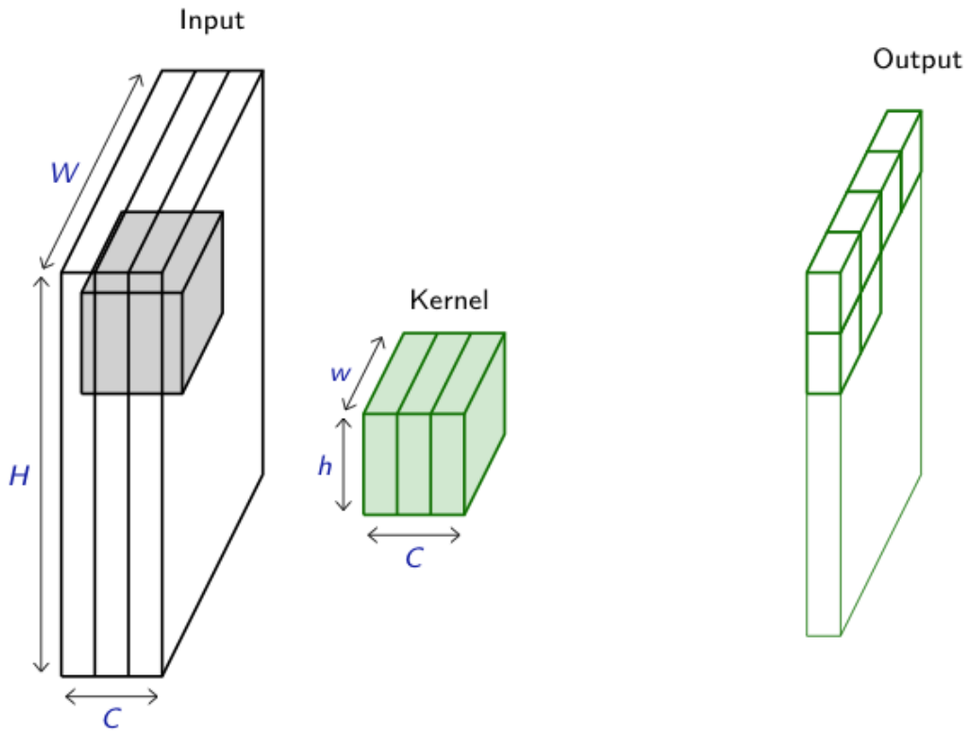
Convolution 2d



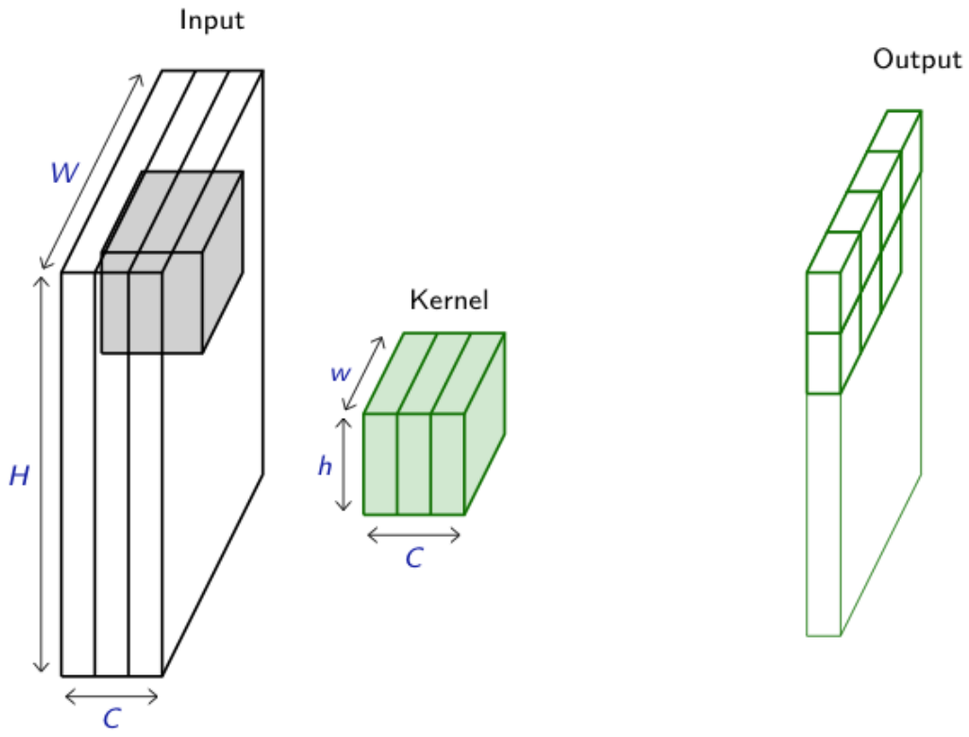
Convolution 2d



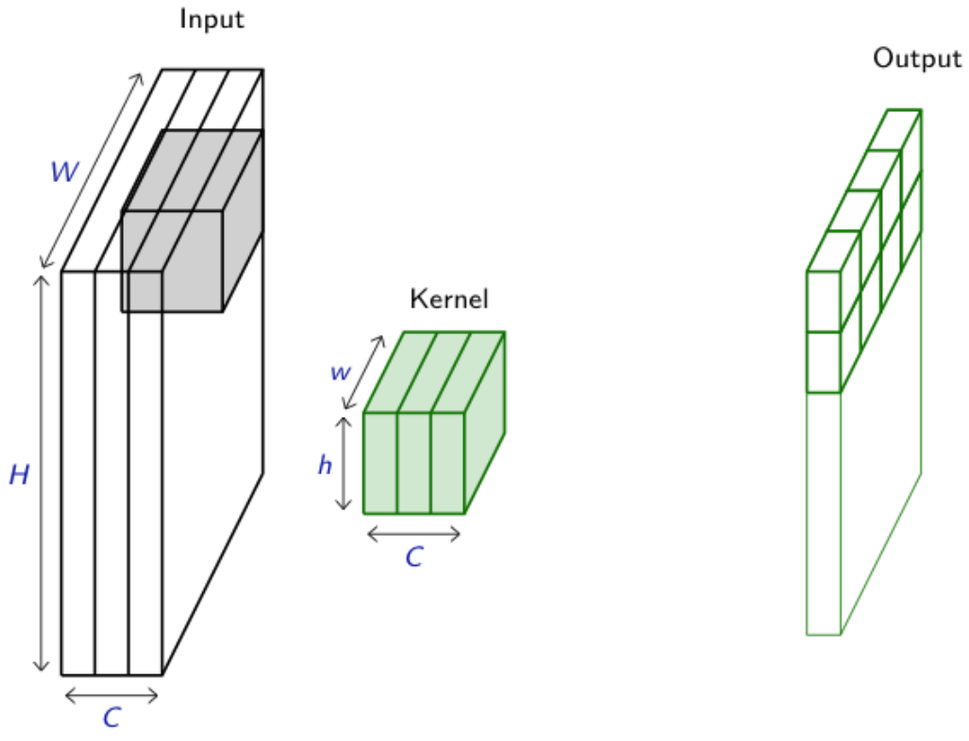
Convolution 2d



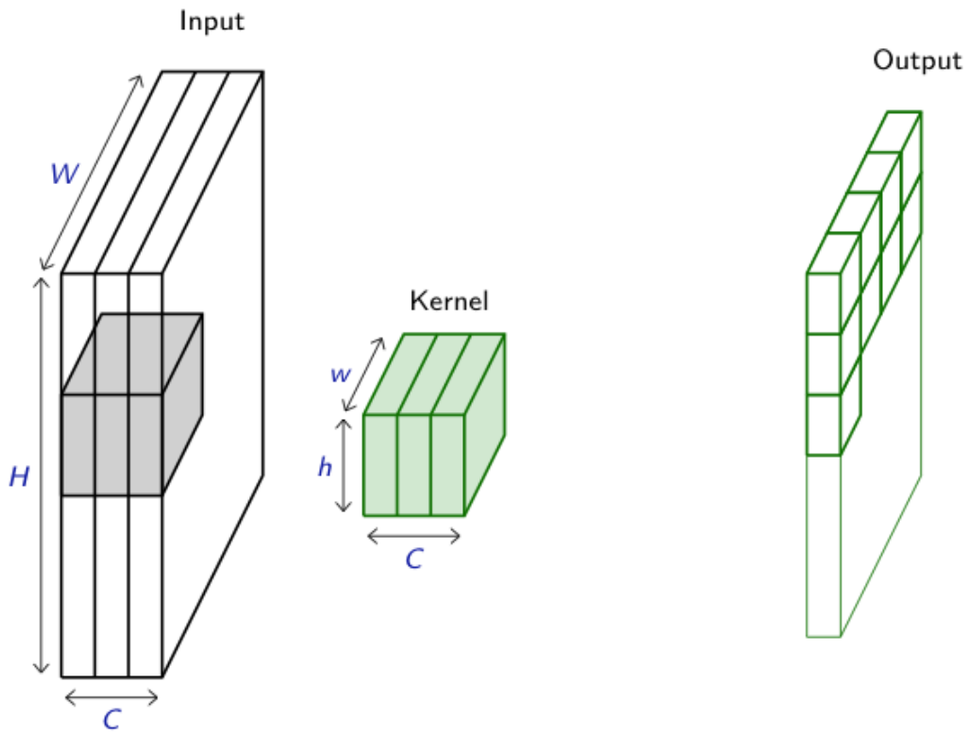
Convolution 2d



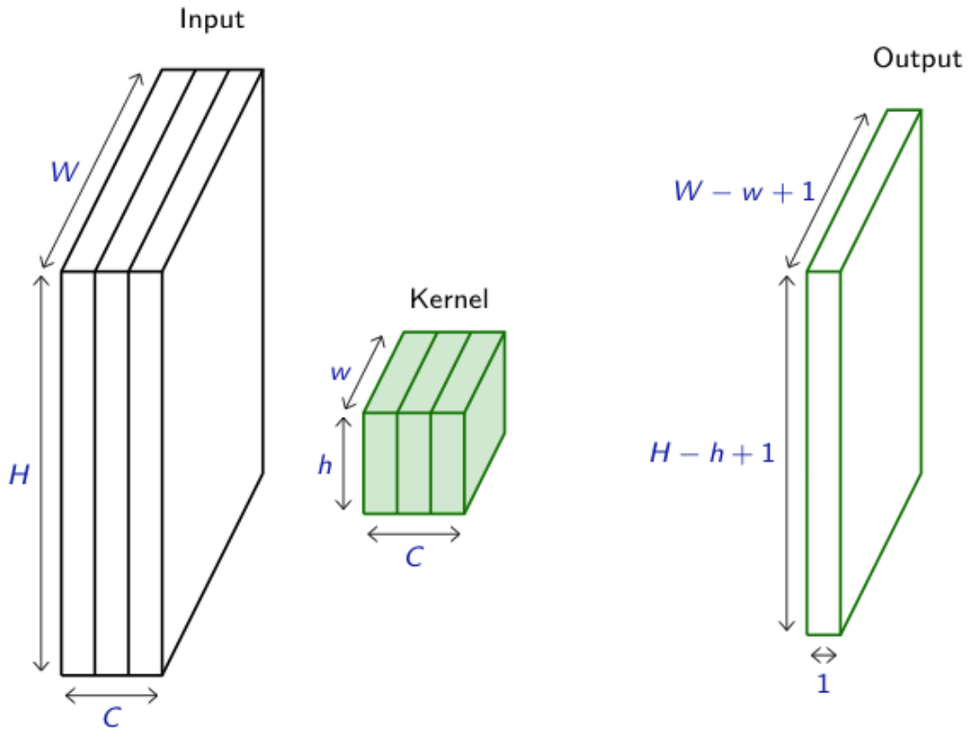
Convolution 2d



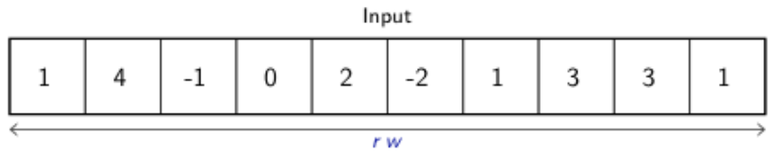
Convolution 2d



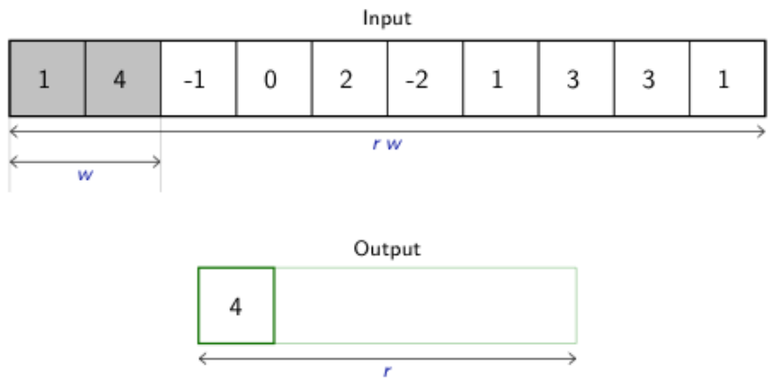
Convolution 2d



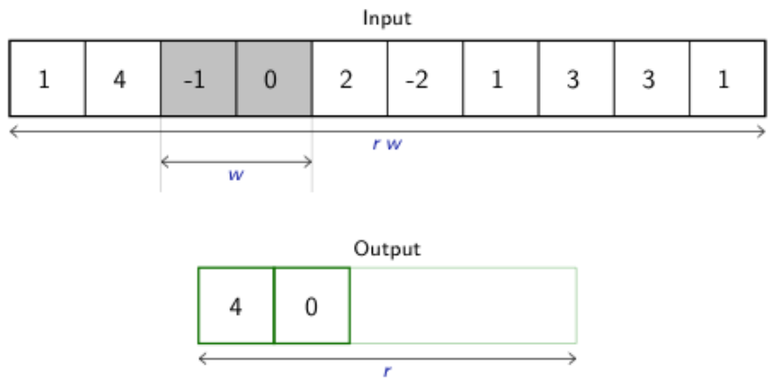
Max-Pooling 1d



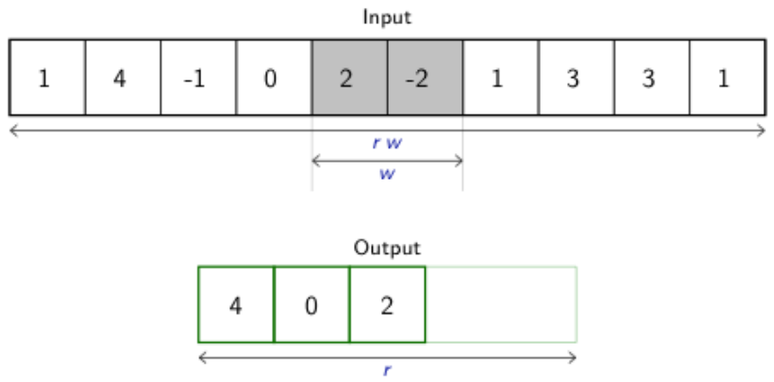
Max-Pooling 1d



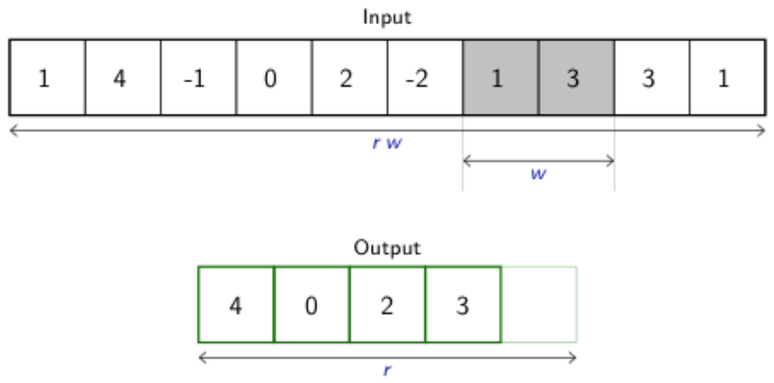
Max-Pooling 1d



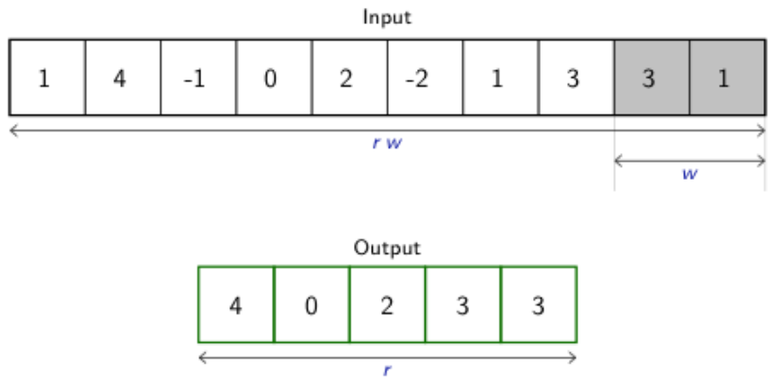
Max-Pooling 1d



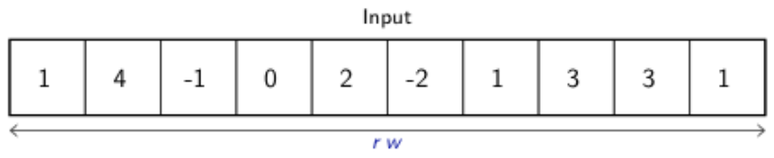
Max-Pooling 1d

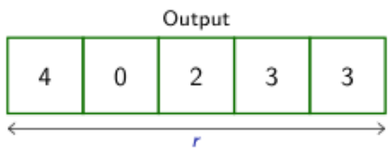


Max-Pooling 1d

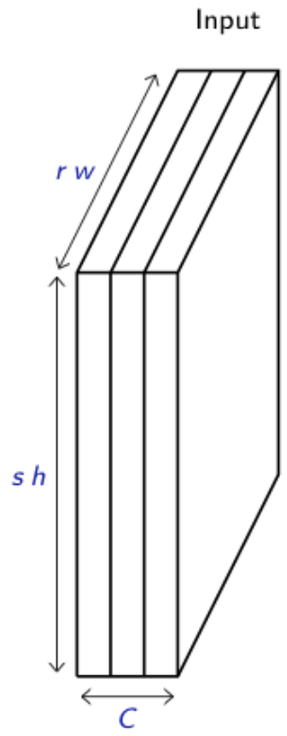


Max-Pooling 1d

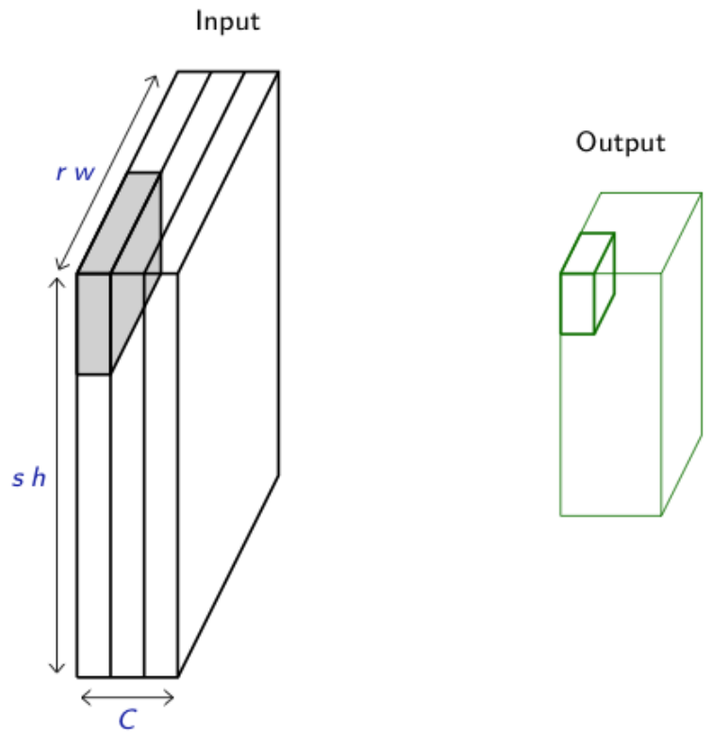




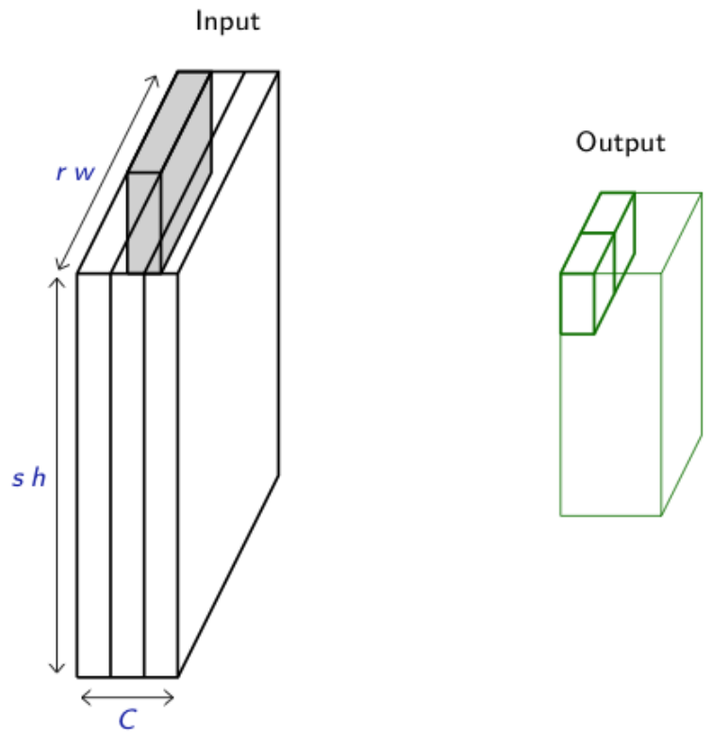
Max-Pooling 2d



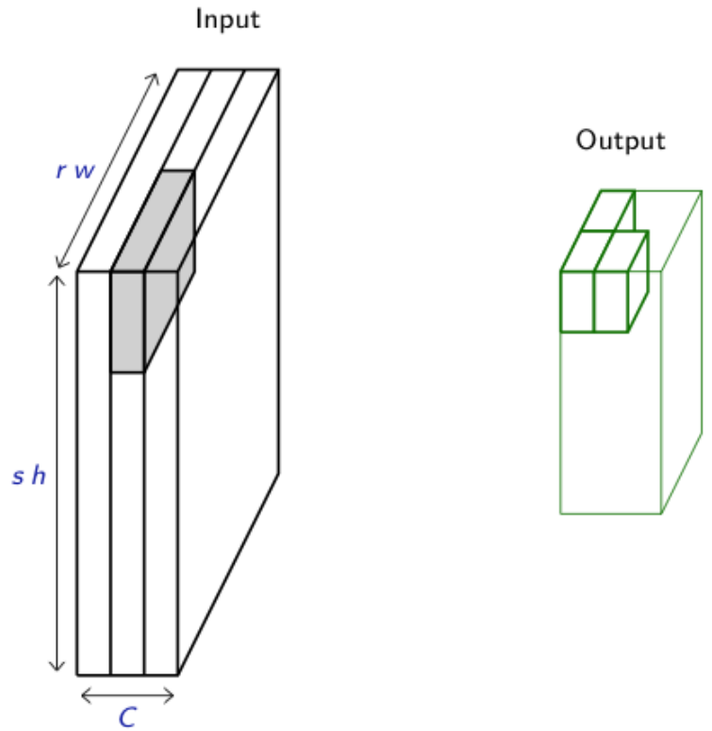
Max-Pooling 2d



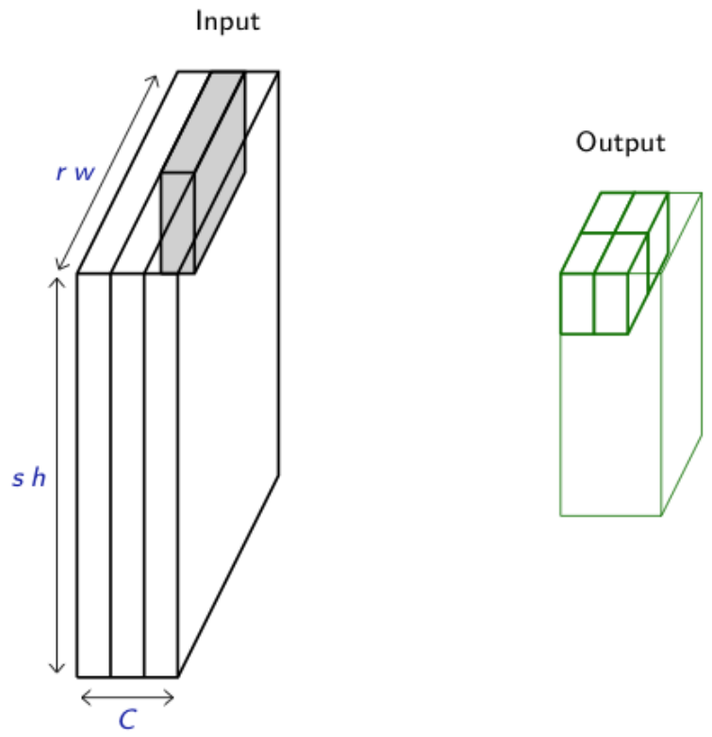
Max-Pooling 2d



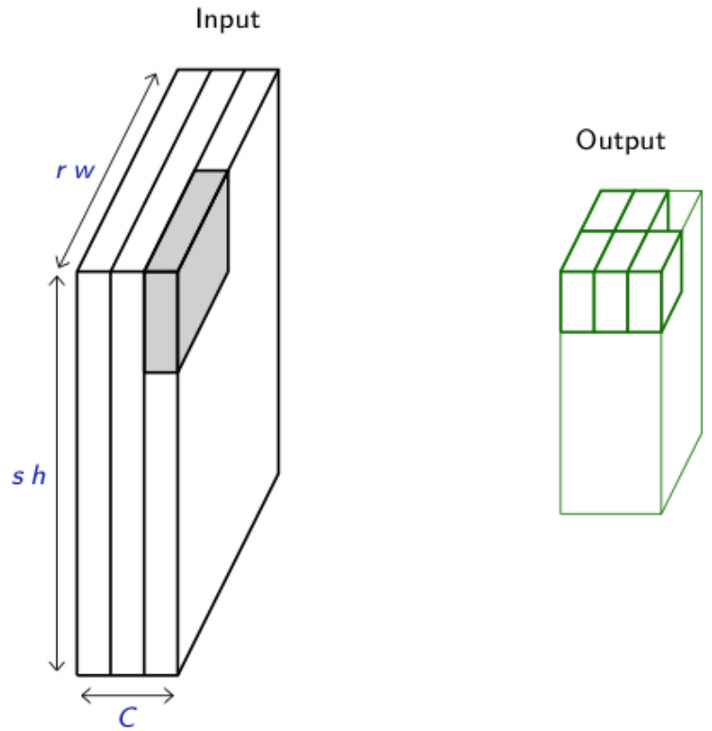
Max-Pooling 2d



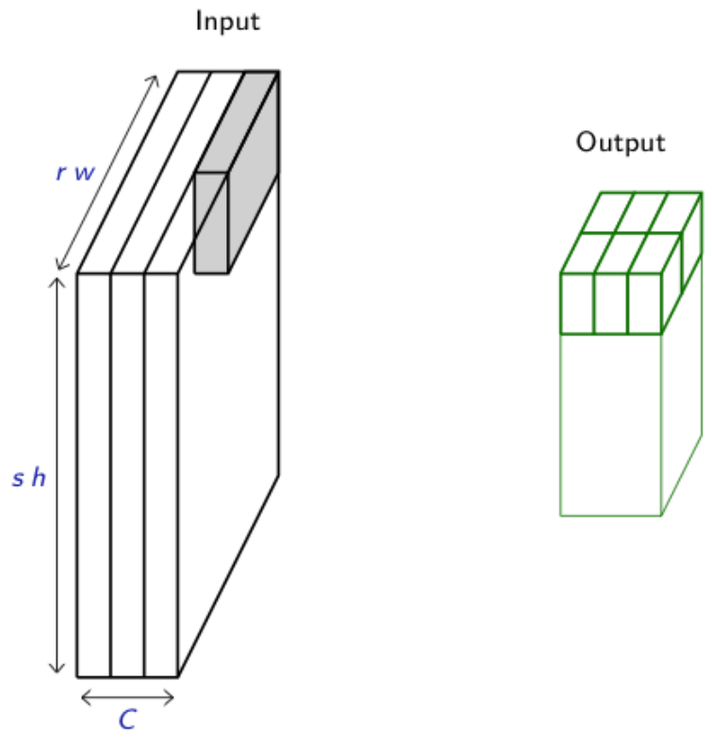
Max-Pooling 2d



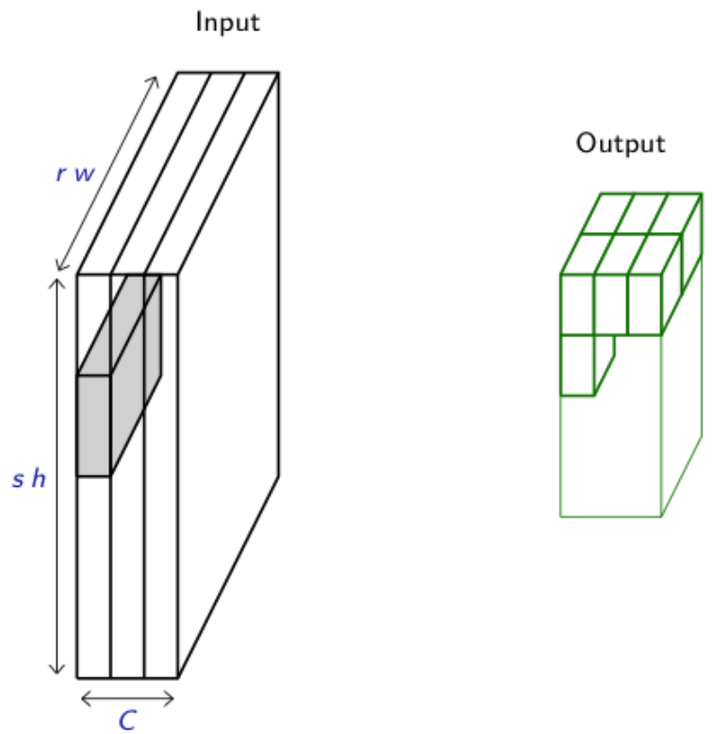
Max-Pooling 2d



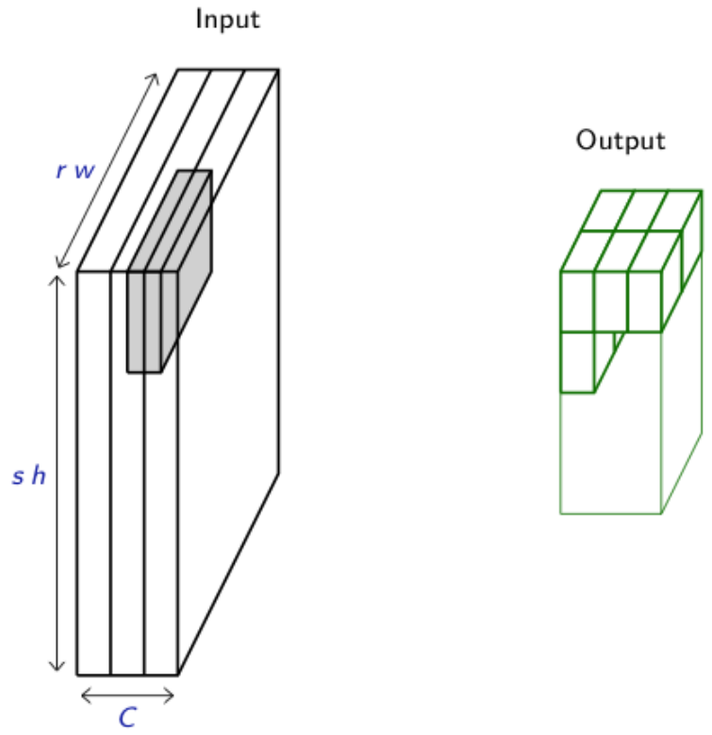
Max-Pooling 2d



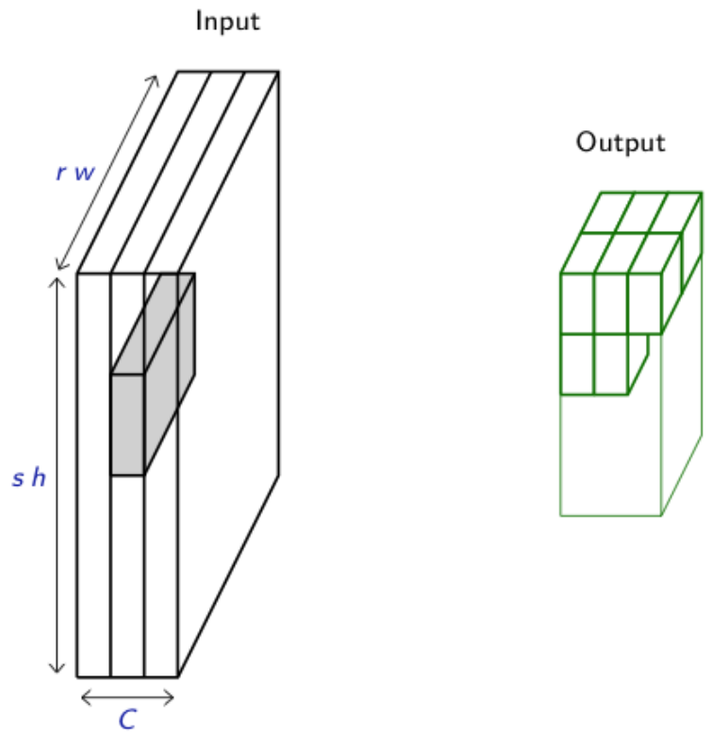
Max-Pooling 2d



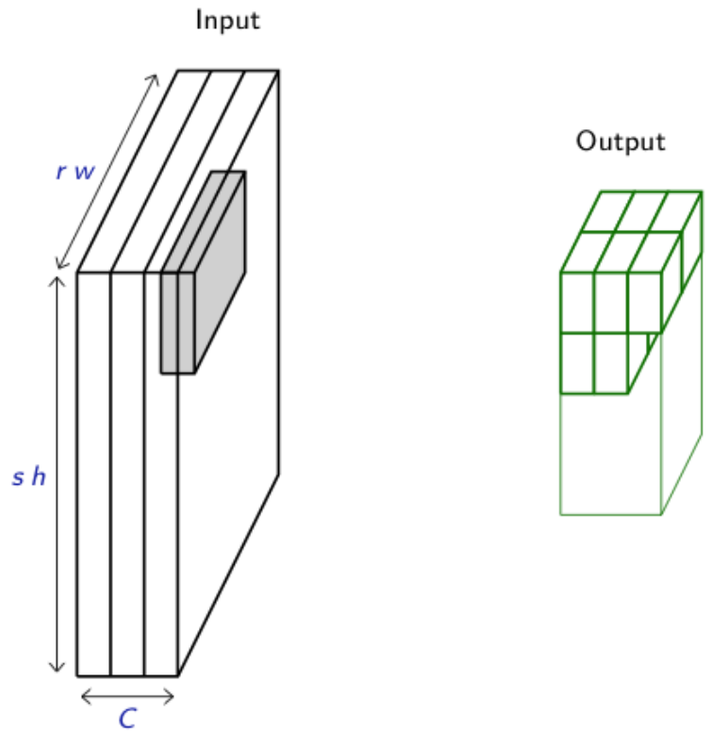
Max-Pooling 2d



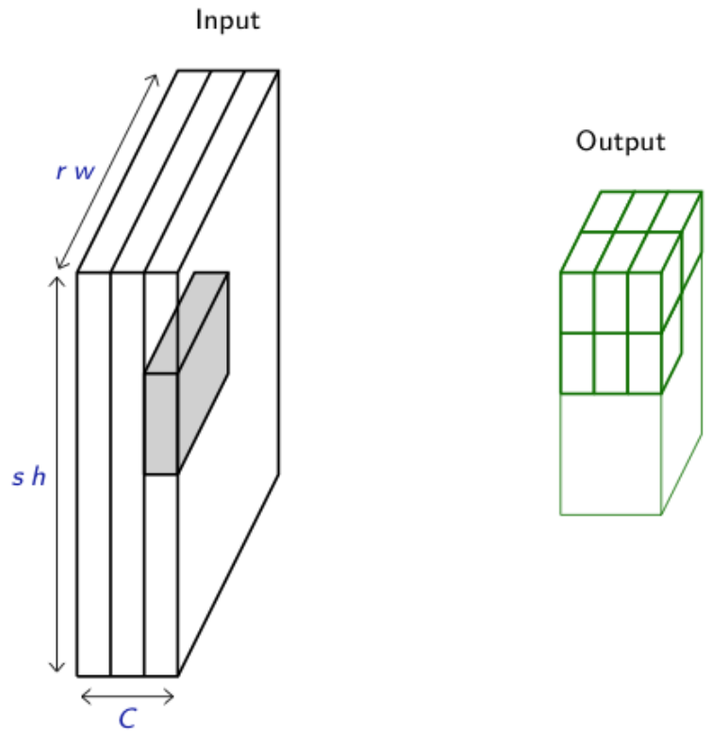
Max-Pooling 2d



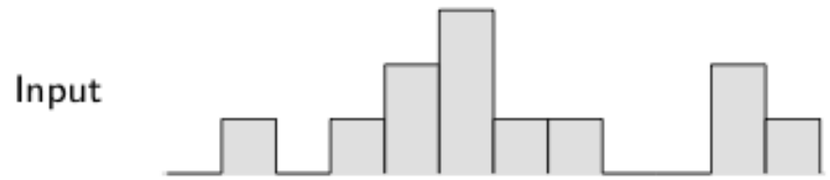
Max-Pooling 2d



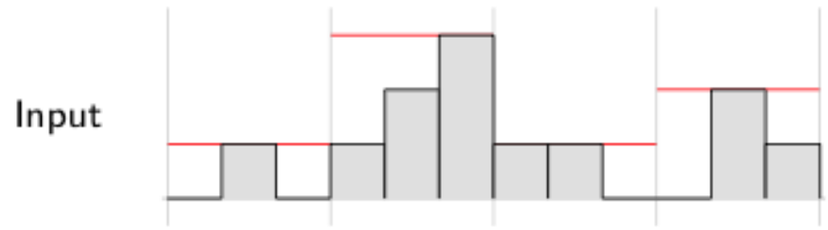
Max-Pooling 2d



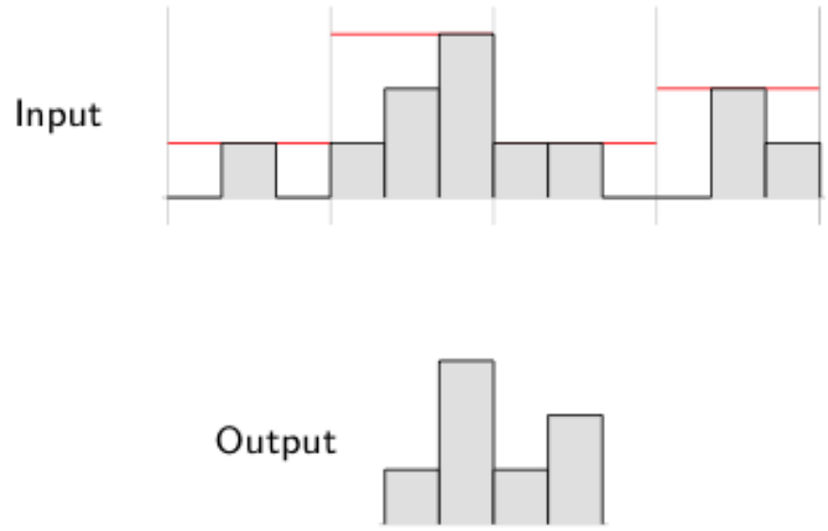
Translation invariance from pooling



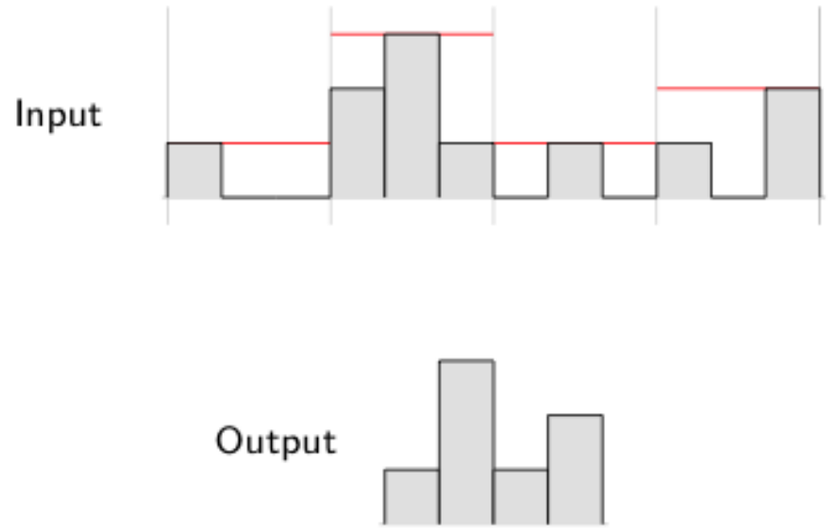
Translation invariance from pooling



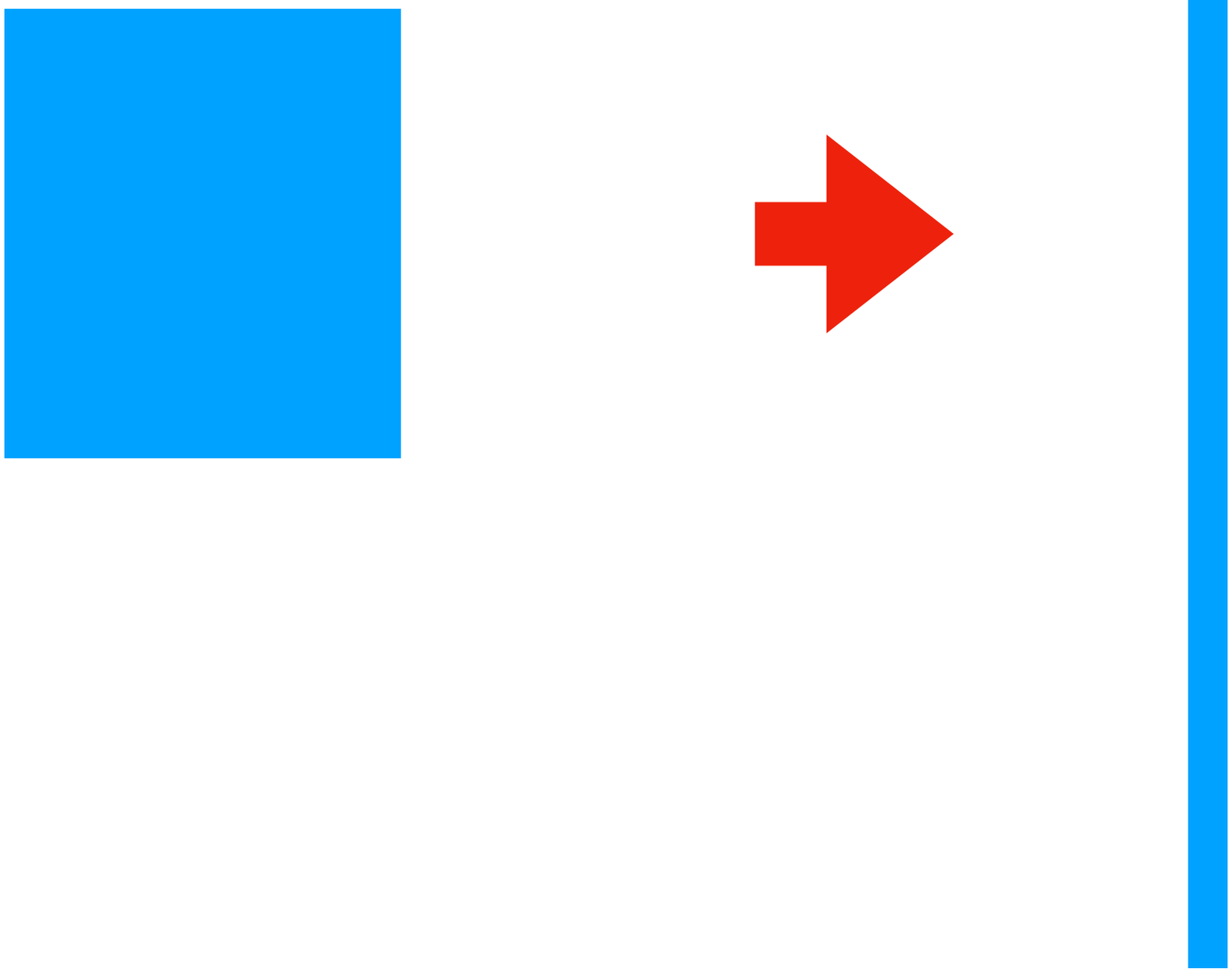
Translation invariance from pooling



Translation invariance from pooling



Flatten
2D->1D



4: conv-nets

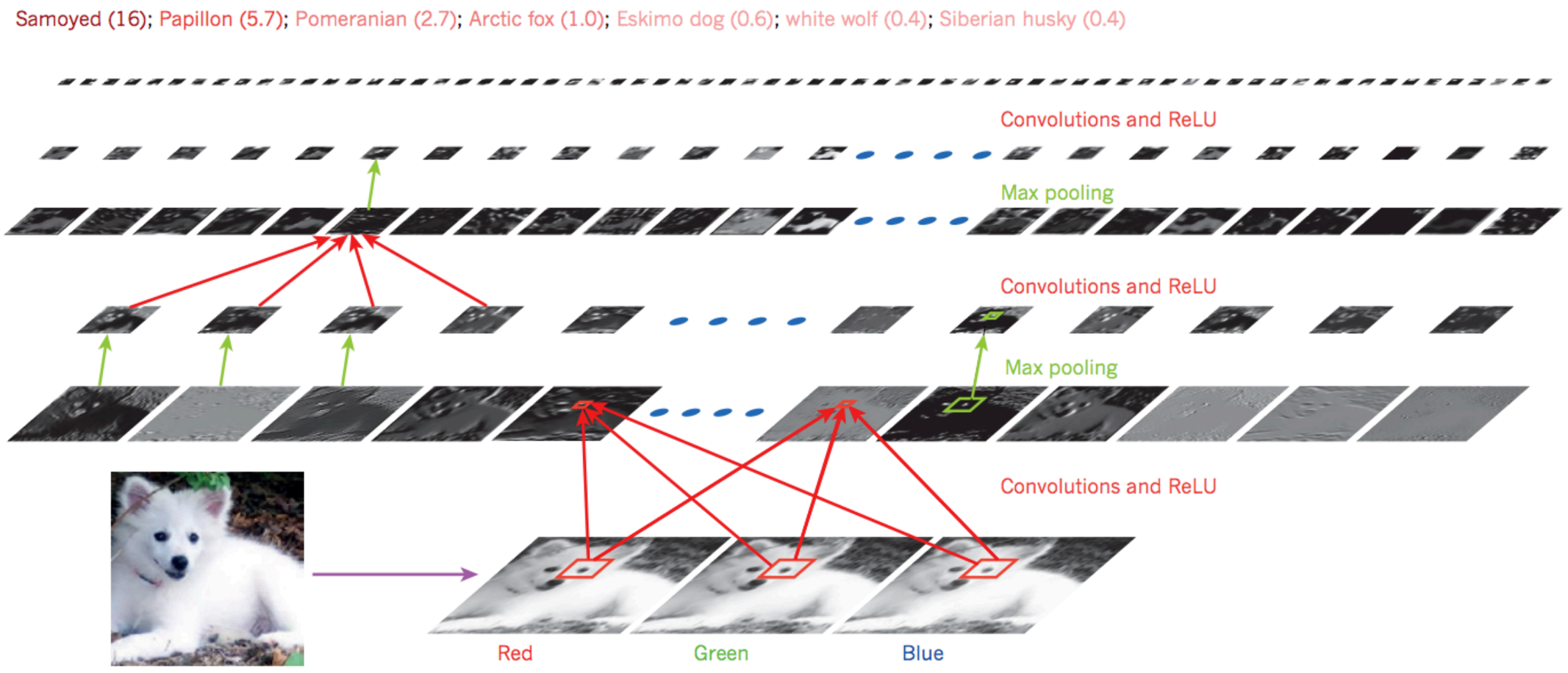
REVIEW

doi:10.1038/nature14539

Deep learning

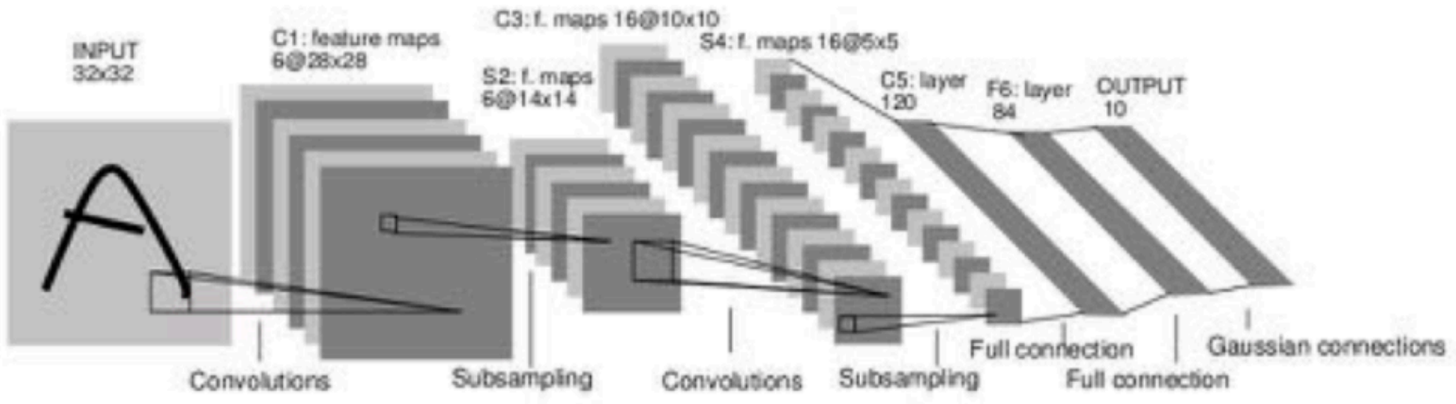
Yann LeCun[1,2], Yoshua Bengio[3] & Geoffrey Hinton[4,5]

Deep learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction. These methods have dramatically improved the state-of-the-art in speech recognition, visual object recognition, object detection and many other domains such as drug discovery and genomics. Deep learning discovers intricate structure in large data sets by using the backpropagation algorithm to indicate how a machine should change its internal parameters that are used to compute the representation in each layer from the representation in the previous layer. Deep convolutional nets have brought about breakthroughs in processing images, video, speech and audio, whereas recurrent nets have shone light on sequential data such as text and speech.



ConvNet

- Neural network with specialized connectivity structure
- Stack multiple stage of feature extractors
- Higher stages compute more global, more invariant features
- Classification layer at the end



LeNet5

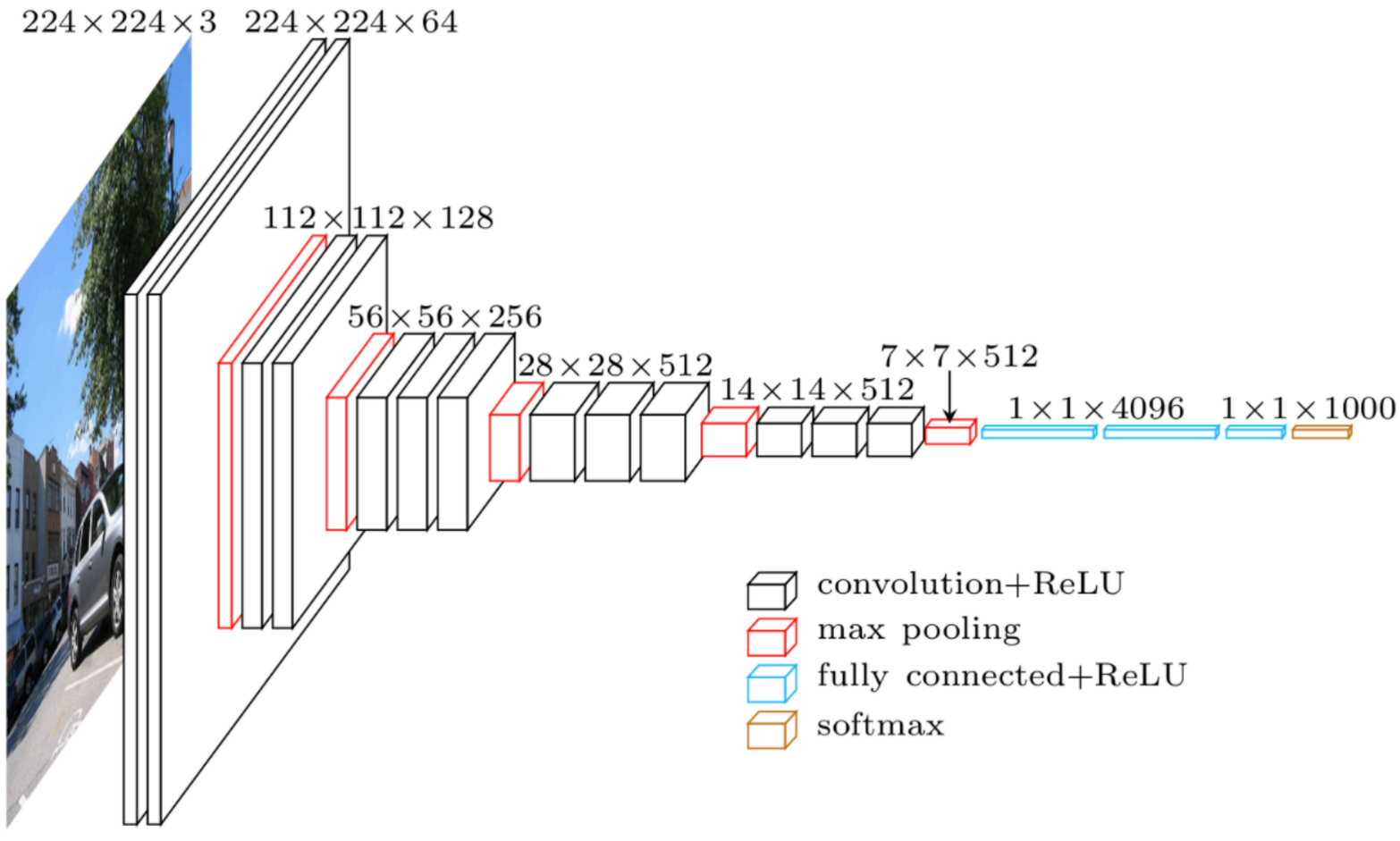
10 classes, input 1 x 28 x 28

```
(features): Sequential (
(0): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
(1): ReLU (inplace)
(2): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
(3): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
(4): ReLU (inplace)
(5): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
)
(classifier): Sequential (
(0): Linear (400 -> 120)
(1): ReLU (inplace)
(2): Linear (120 -> 84)
(3): ReLU (inplace)
(4): Linear (84 -> 10) )
```

AlexNet

```
(features): Sequential (
(0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
(1): ReLU (inplace)
(2): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
(3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
(4): ReLU (inplace)
(5): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
(6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(7): ReLU (inplace)
(8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(9): ReLU (inplace)
(10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(11): ReLU (inplace)
(12): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
)

(classifier): Sequential (
(0): Dropout (p = 0.5)
(1): Linear (9216 -> 4096)
(2): ReLU (inplace)
(3): Dropout (p = 0.5)
(4): Linear (4096 -> 4096)
(5): ReLU (inplace)
(6): Linear (4096 -> 1000)
)
```

VGG-16


224 × 224 × 3
224 × 224 × 64
112 × 112 × 128
56 × 56 × 256
28 × 28 × 512
14 × 14 × 512
7 × 7 × 512
1 × 1 × 4096
1 × 1 × 1000
convolution+ReLU
max pooling
fully connected+ReLU
softmax

```
model = Sequential()
model.add(ZeroPadding2D((1, 1), input_shape=(3, 224, 224)))
model.add(Convolution2D(64, 3, 3, activation='relu'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(64, 3, 3, activation='relu'))
model.add(MaxPooling2D((2, 2), strides=(2, 2)))

model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(128, 3, 3, activation='relu'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(128, 3, 3, activation='relu'))
model.add(MaxPooling2D((2, 2), strides=(2, 2)))

model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(256, 3, 3, activation='relu'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(256, 3, 3, activation='relu'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(256, 3, 3, activation='relu'))
model.add(MaxPooling2D((2, 2), strides=(2, 2)))

model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(MaxPooling2D((2, 2), strides=(2, 2)))

model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu'))

# Add another conv layer with ReLU + GAP
model.add(Convolution2D(num_input_channels, 3, 3, activation='relu', border_mode="same"))
model.add(AveragePooling2D((14, 14)))
model.add(Flatten())
# Add the W layer
model.add(Dense(nb_classes, activation='softmax'))
```

VGG-19

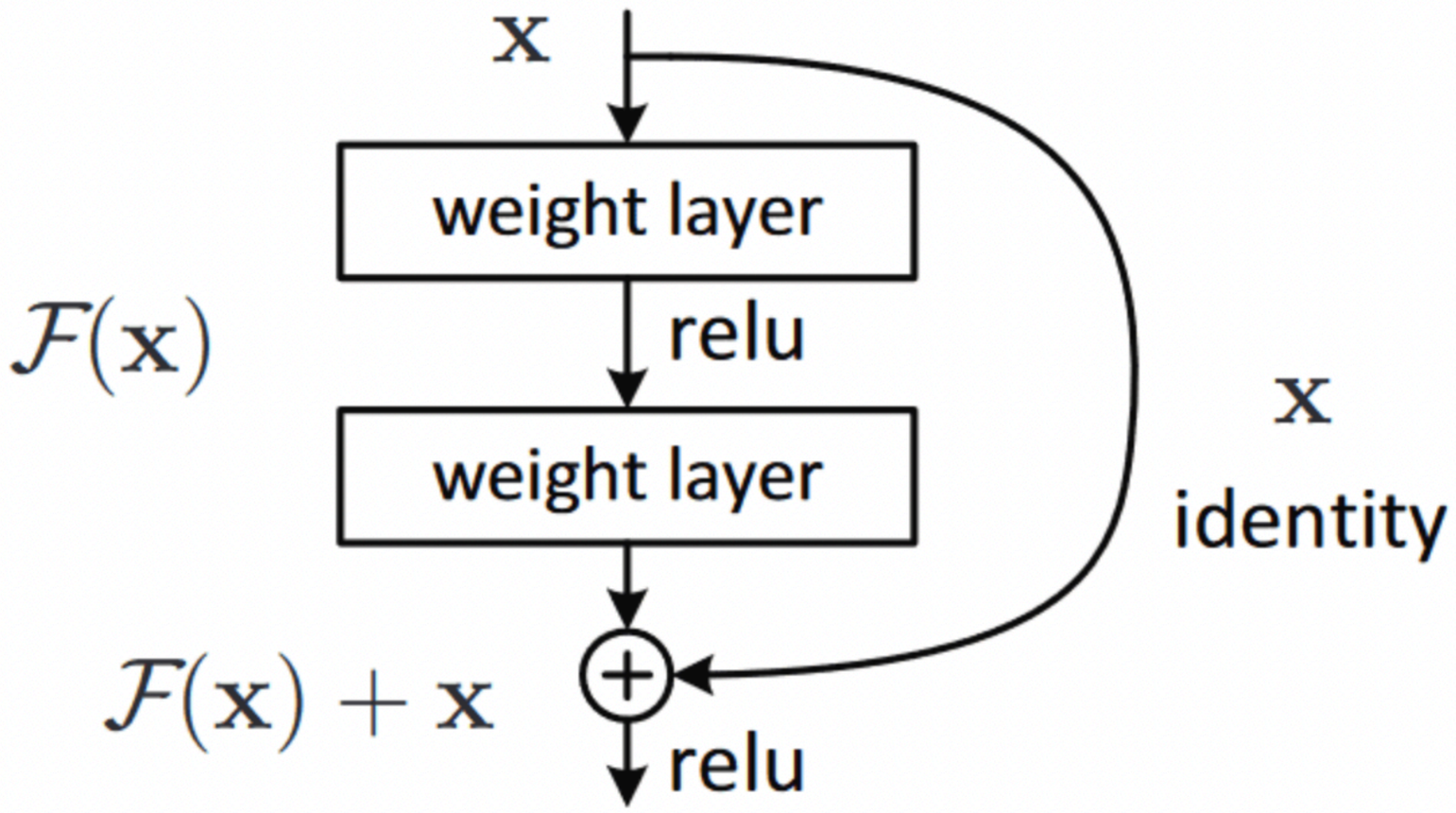
```
(0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(1): ReLU (inplace)
(2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(3): ReLU (inplace)
(4): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
(5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(6): ReLU (inplace)
(7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(8): ReLU (inplace)
(9): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
(10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(11): ReLU (inplace)
(12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(13): ReLU (inplace)
(14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(15): ReLU (inplace)
(16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(17): ReLU (inplace)
(18): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
(19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(20): ReLU (inplace)
(21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(22): ReLU (inplace)
(23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(24): ReLU (inplace)
(25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(26): ReLU (inplace)
(27): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
(28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(29): ReLU (inplace)
...
```

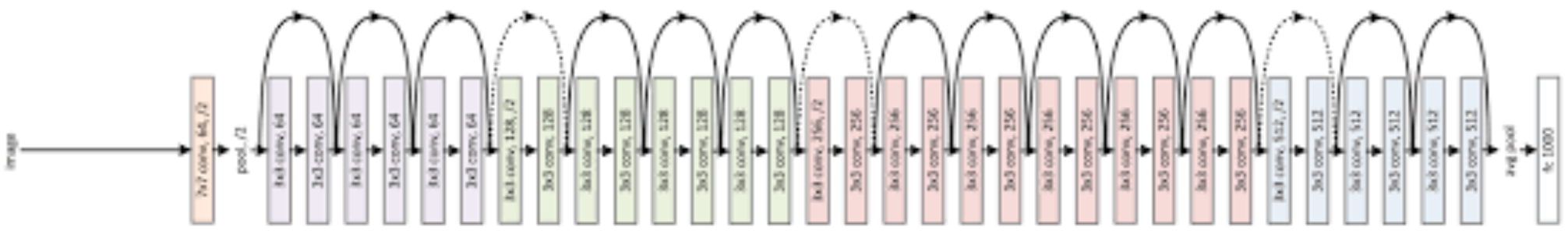
A saturation point

If we continue stacking more layers on a CNN:

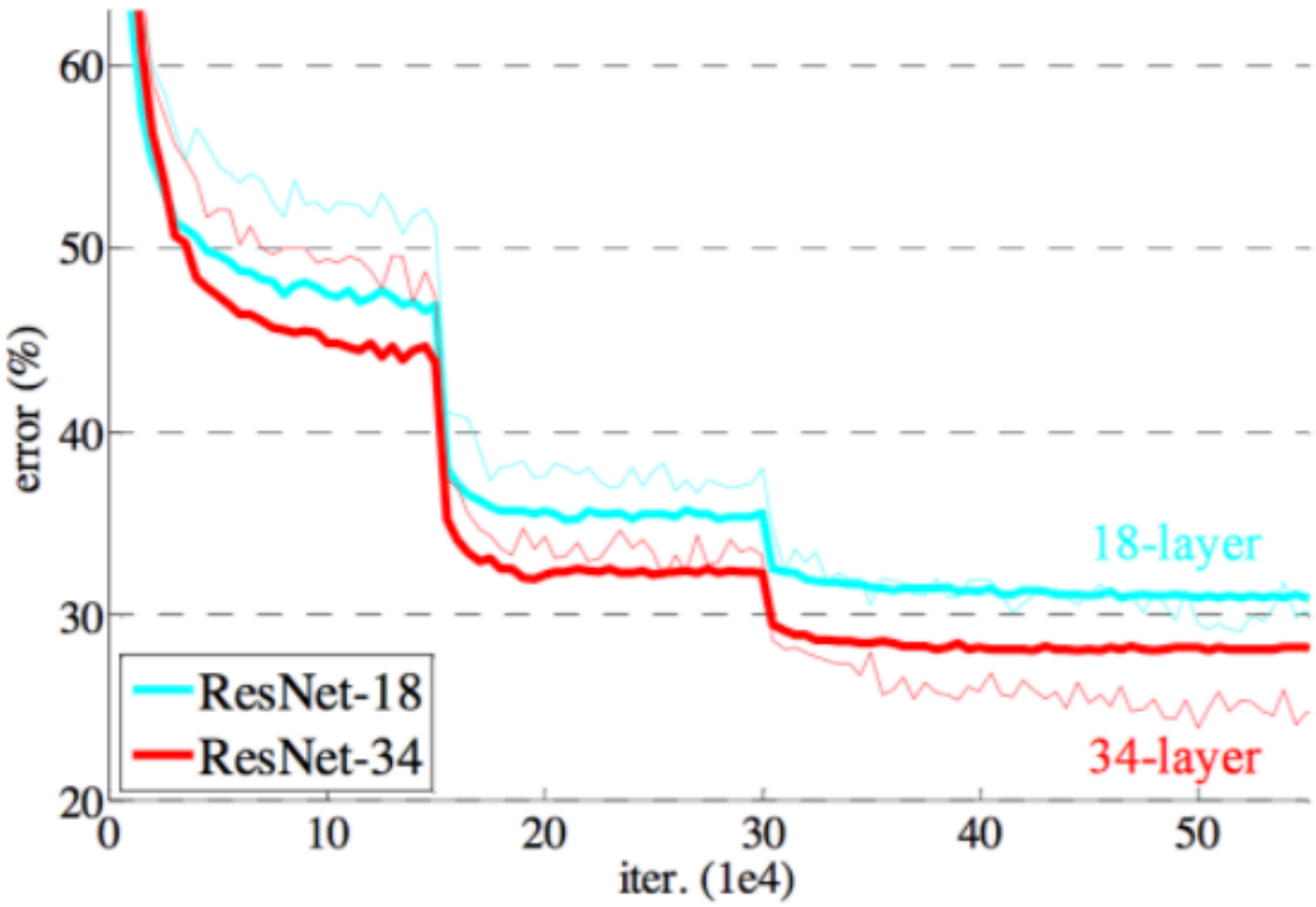
Deeper models are harder to optimize

Resnets: Skiped-connections

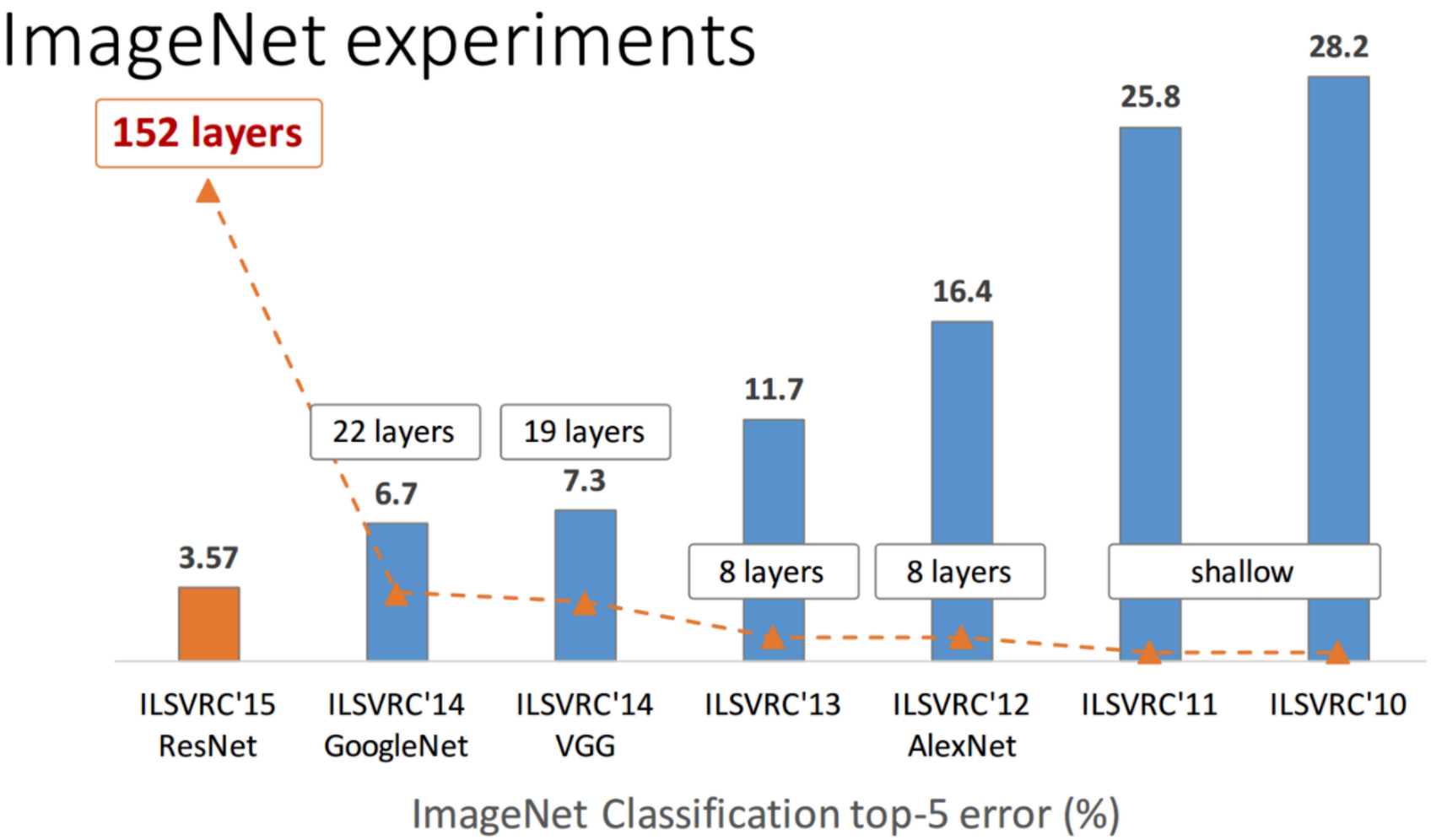




Resnets: Skiped-connections



Deeper is better

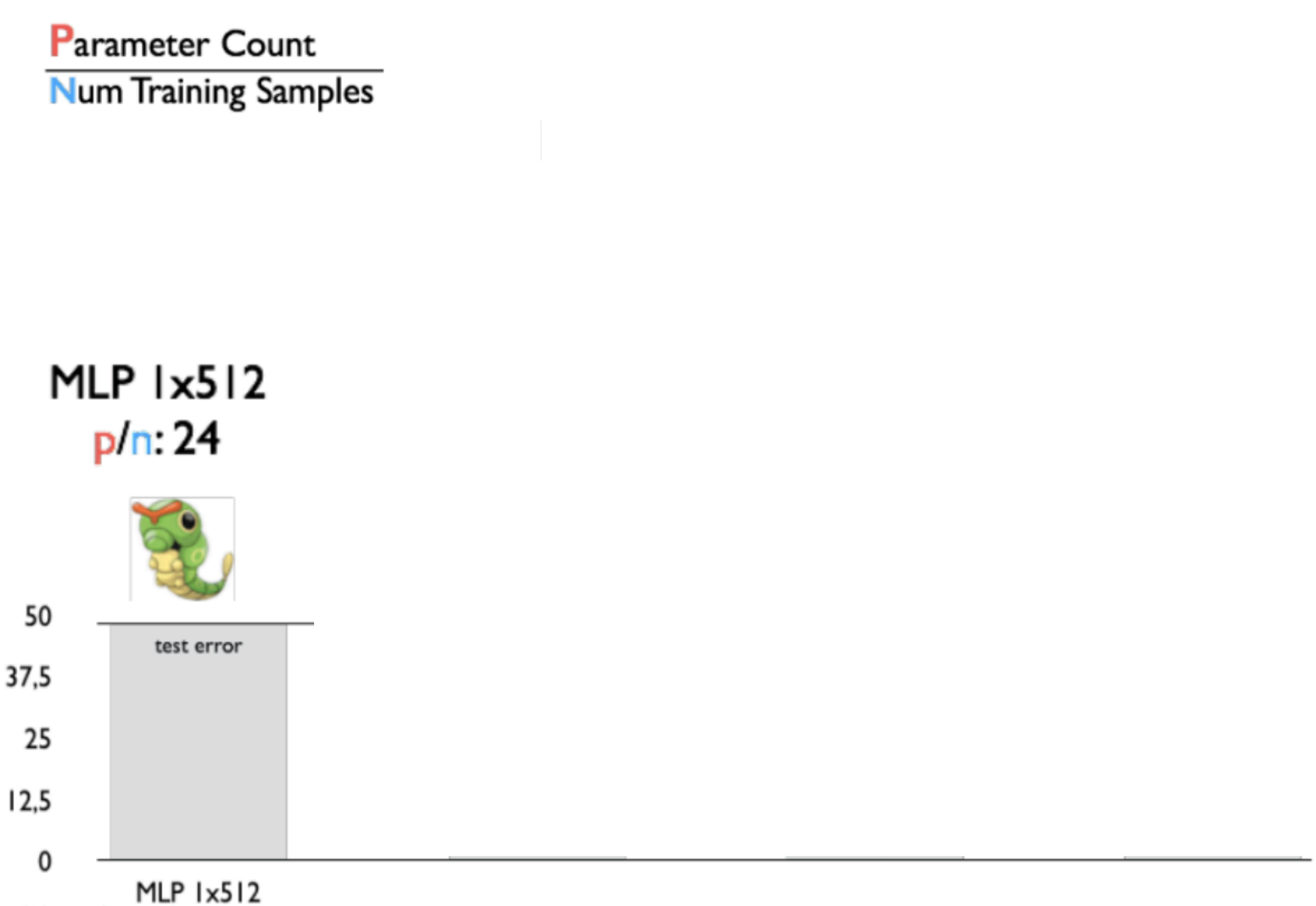


Regularization

Parameter Count
Num Training Samples

MLP 1x512

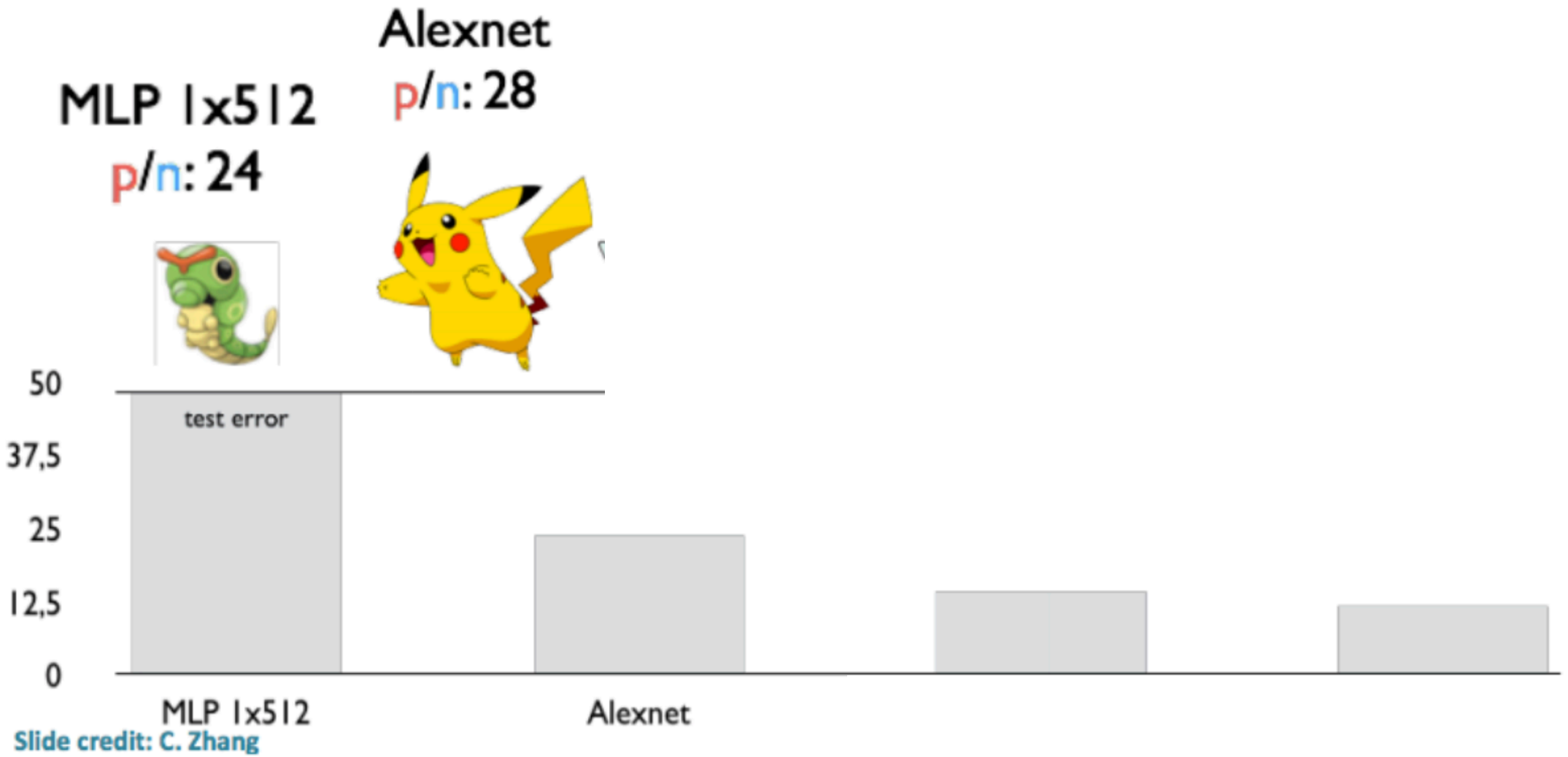
p/n: 24



Slide credit: C. Zhang

Regularization

$$\frac{\text{Parameter Count}}{\text{Num Training Samples}}$$

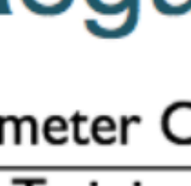


Slide credit: C. Zhang

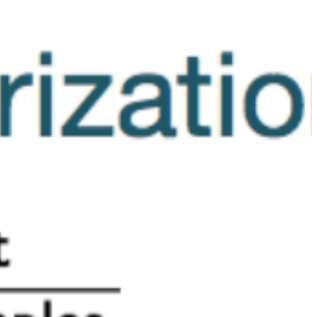
Regularization

Parameter Count
Num Training Samples

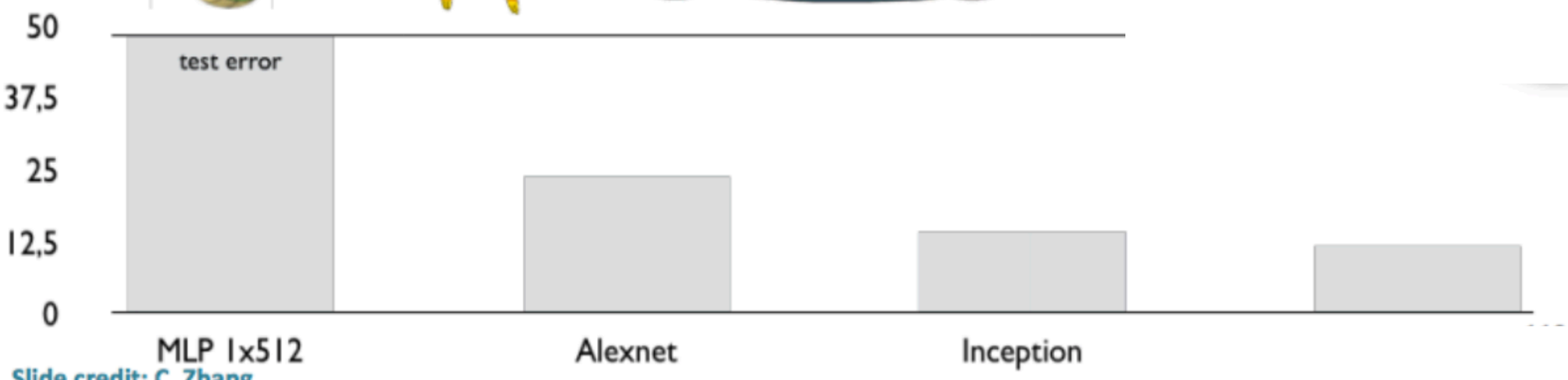
MLP 1x512
p/n: 24



Alexnet
p/n: 28



Inceptior
p/n: 33

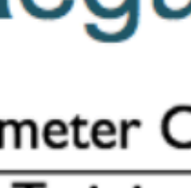


Slide credit: C. Zhang

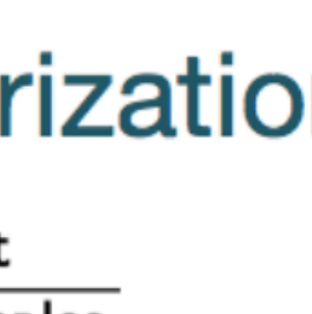
Regularization

Parameter Count
Num Training Samples

MLP 1x512
p/n: 24

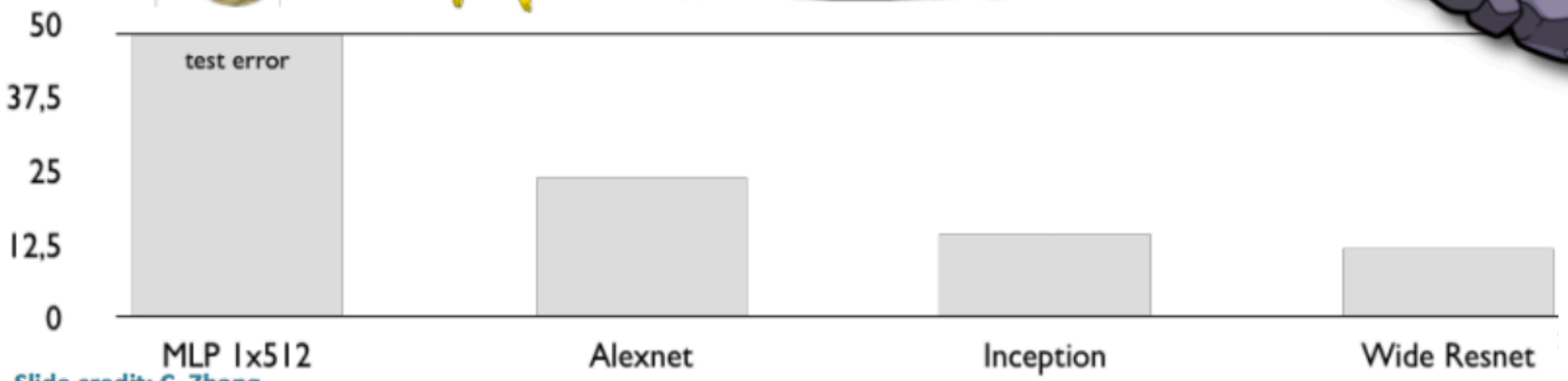


Alexnet
p/n: 28



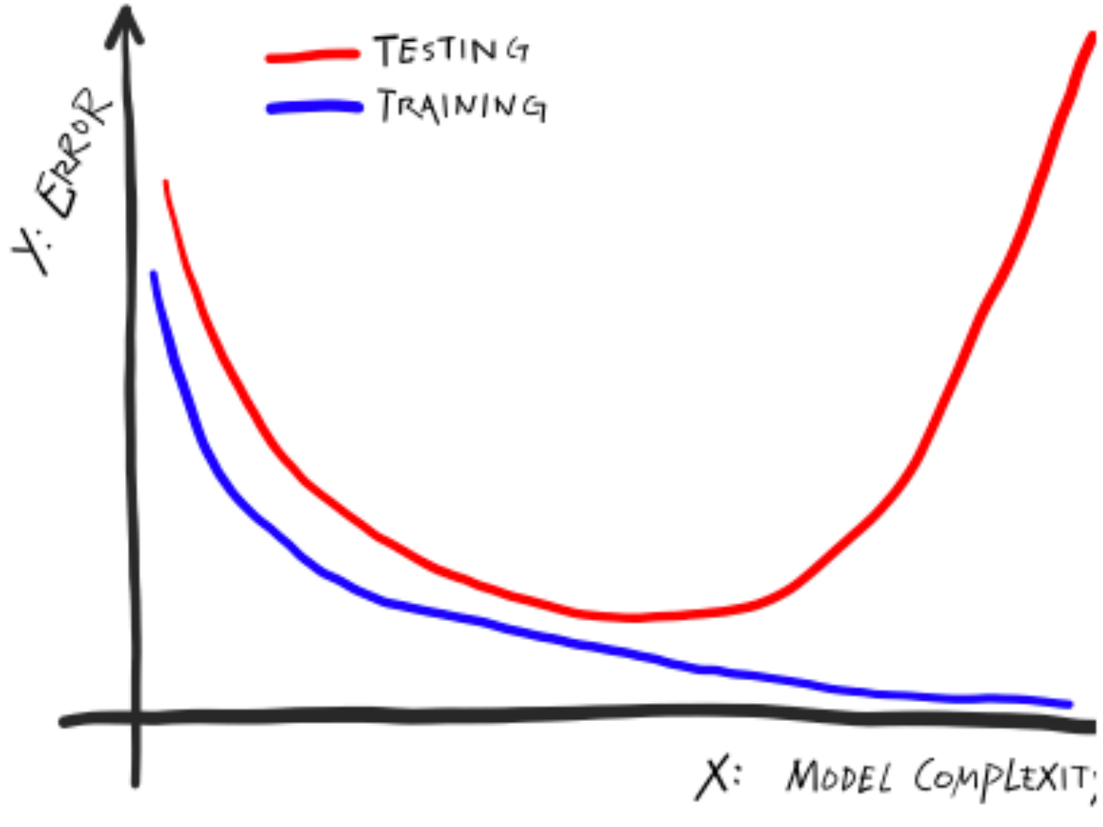
Inception
p/n: 33

Wide Resnet
p/n: 179



Slide credit: C. Zhang

Regularization



Deep Learning

Slide credit: C. Zhang

We learn from KNN that we should be careful and not use too many parameters….

… So how come deep learning works in the overparamterized regime?

Short answer: We do not fully understand!

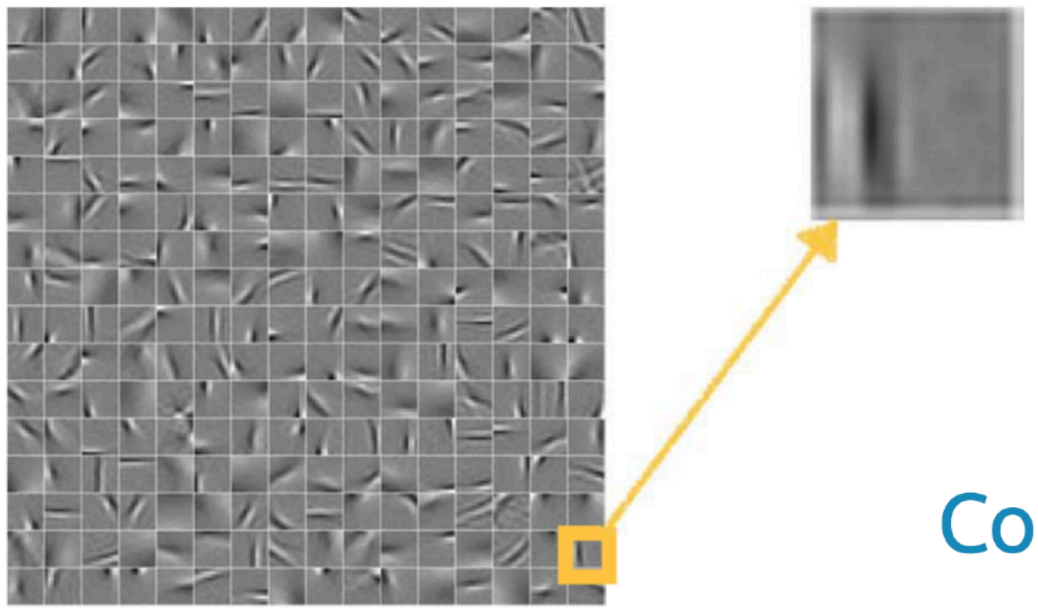
Long answer: Gradient descent is magic

We know they are many set of weights that minimise the loss, and most of them are bad at generalisation, but gradient descent seems to be biased to go toward the "good" ones: This is called the *"implicit regularisation"* of gradient descent

What is learned in conv-nets?

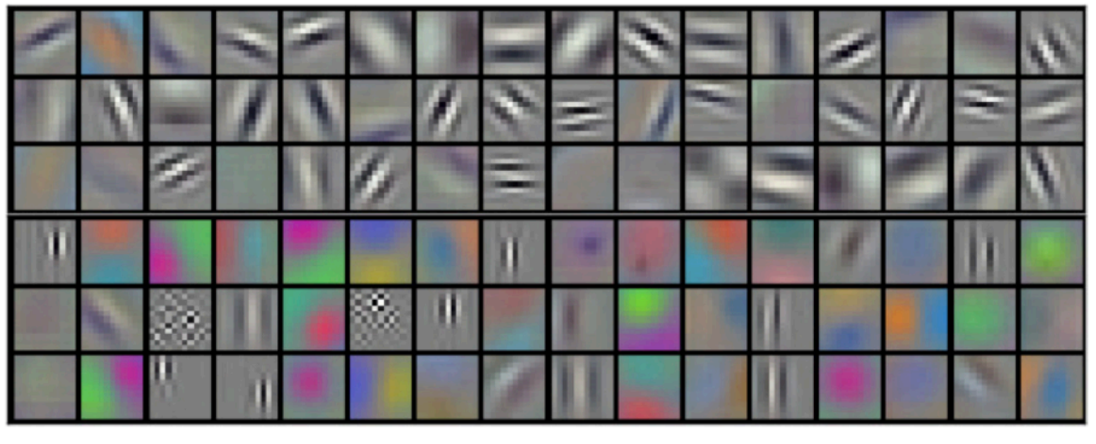
Convolutions

- A bank of 256 filters (learned from data)
- Each filter is 1d (it applies to a grayscale image)
- Each filter is 16 x 16 pixels

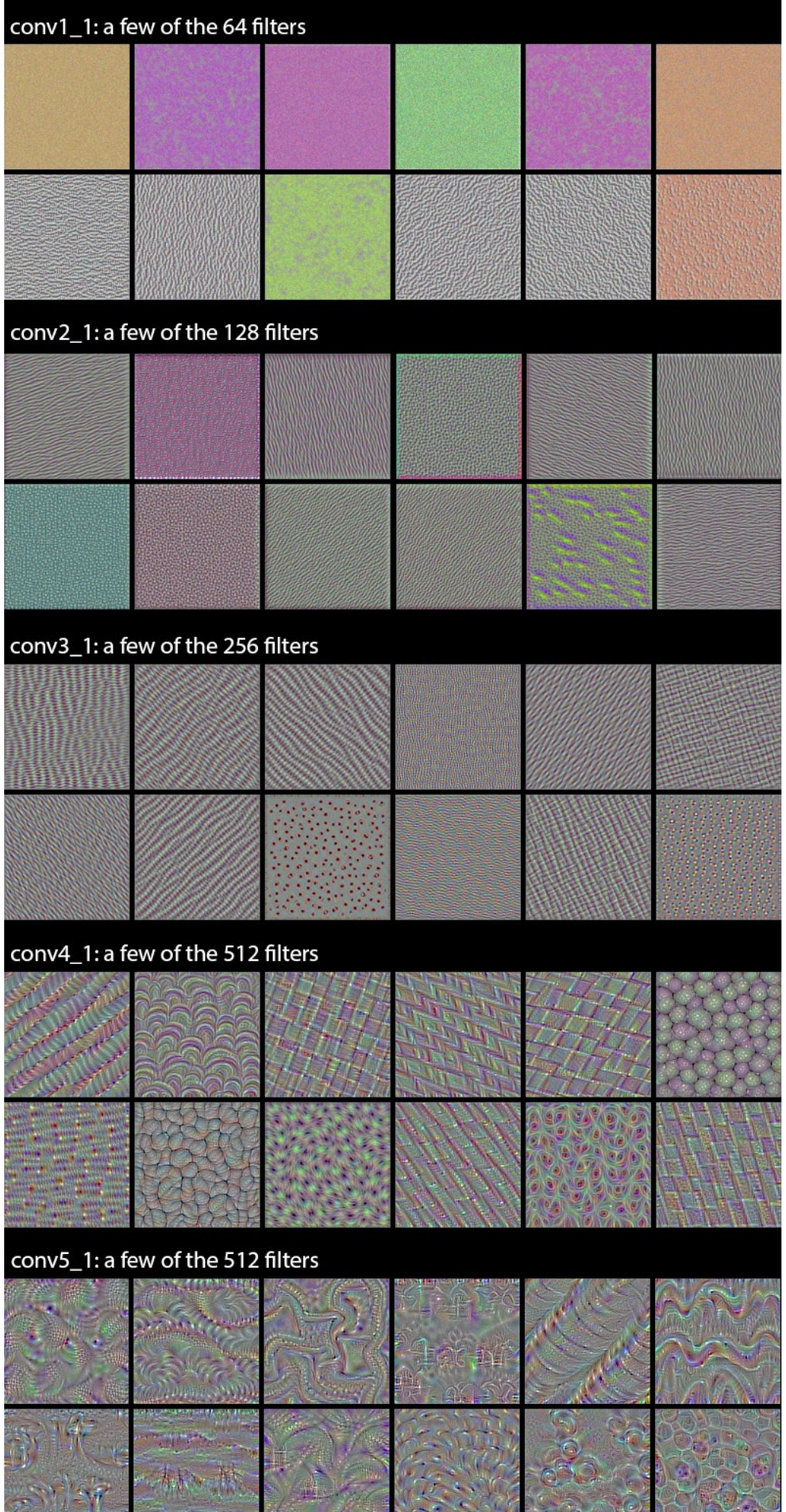


Convolutions

- A bank of 256 filters (learned from data)
- 3D filters for RGB inputs

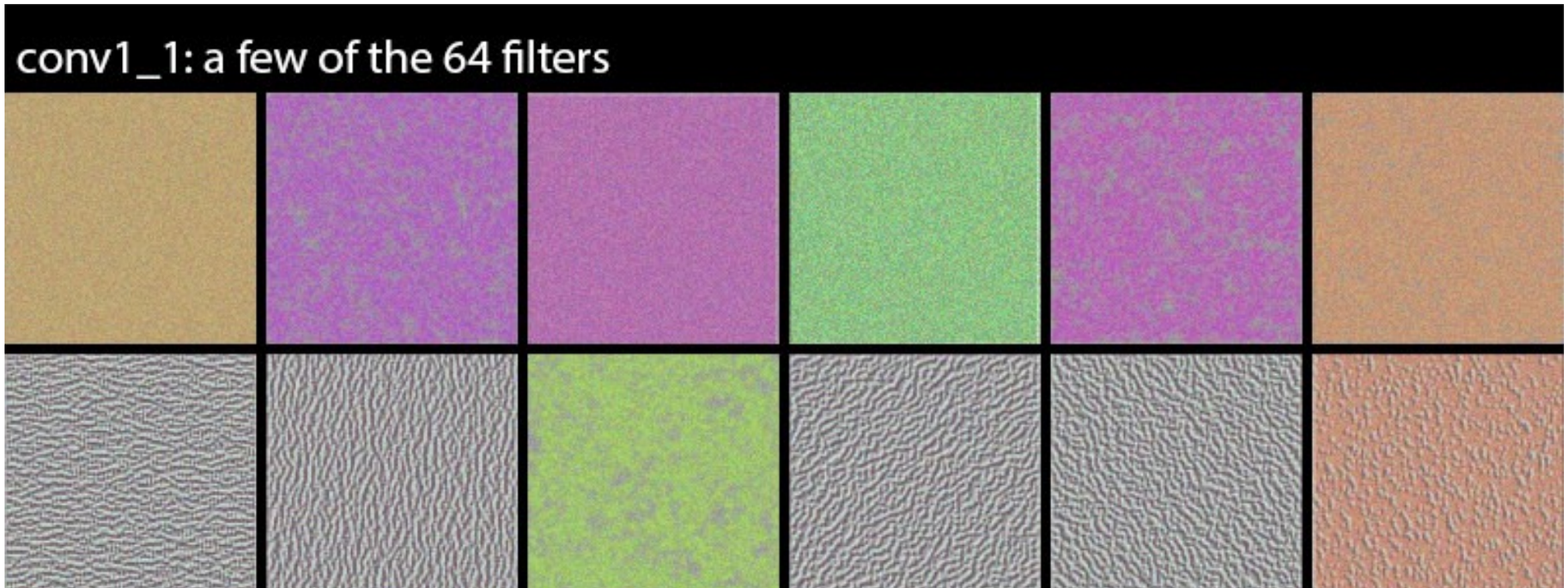


What sort of images maximise the activity for a given neutron in each layers?



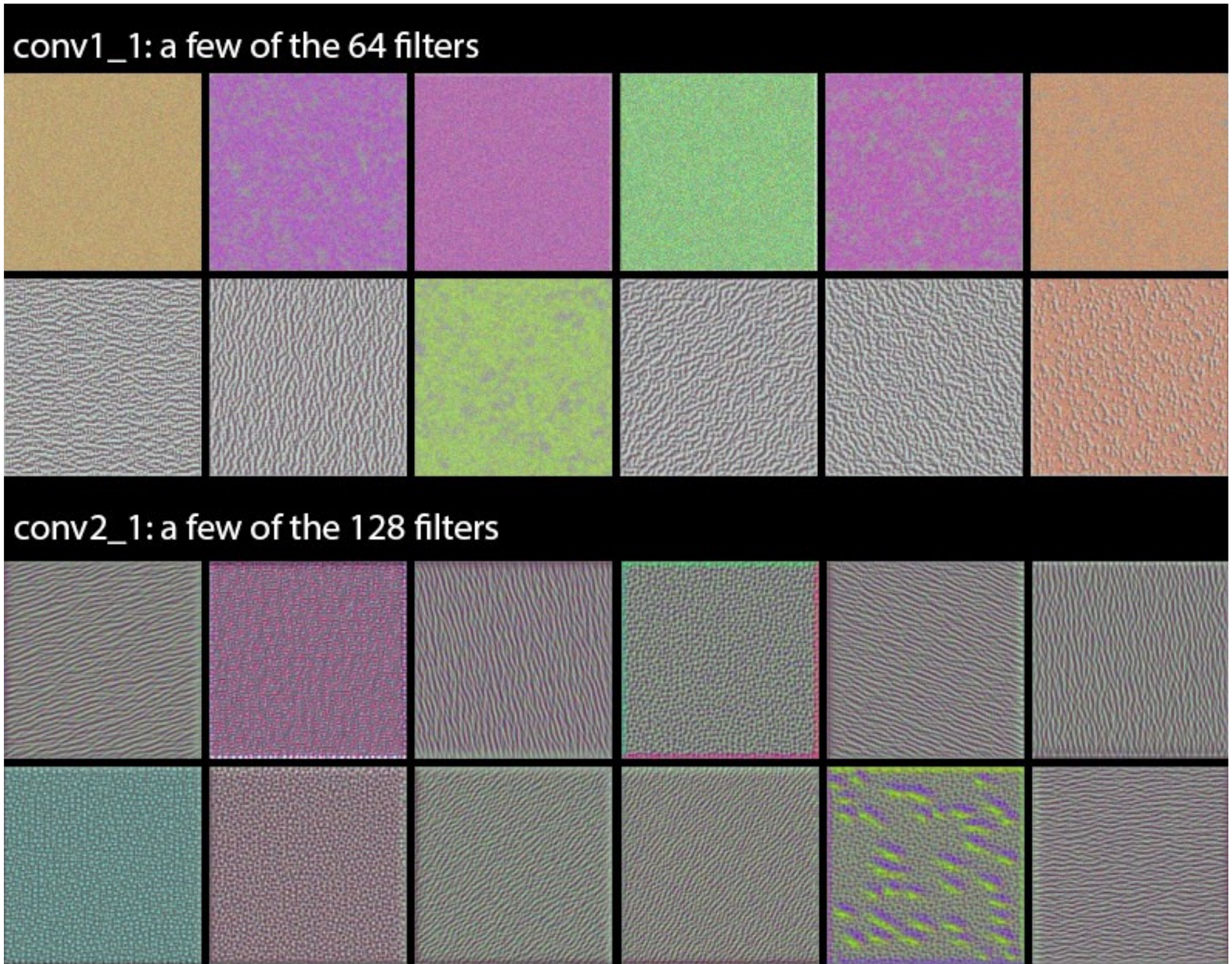
Slide credit: Francois Chollet

What sort of images maximise the activity for a given neutron in each layers?



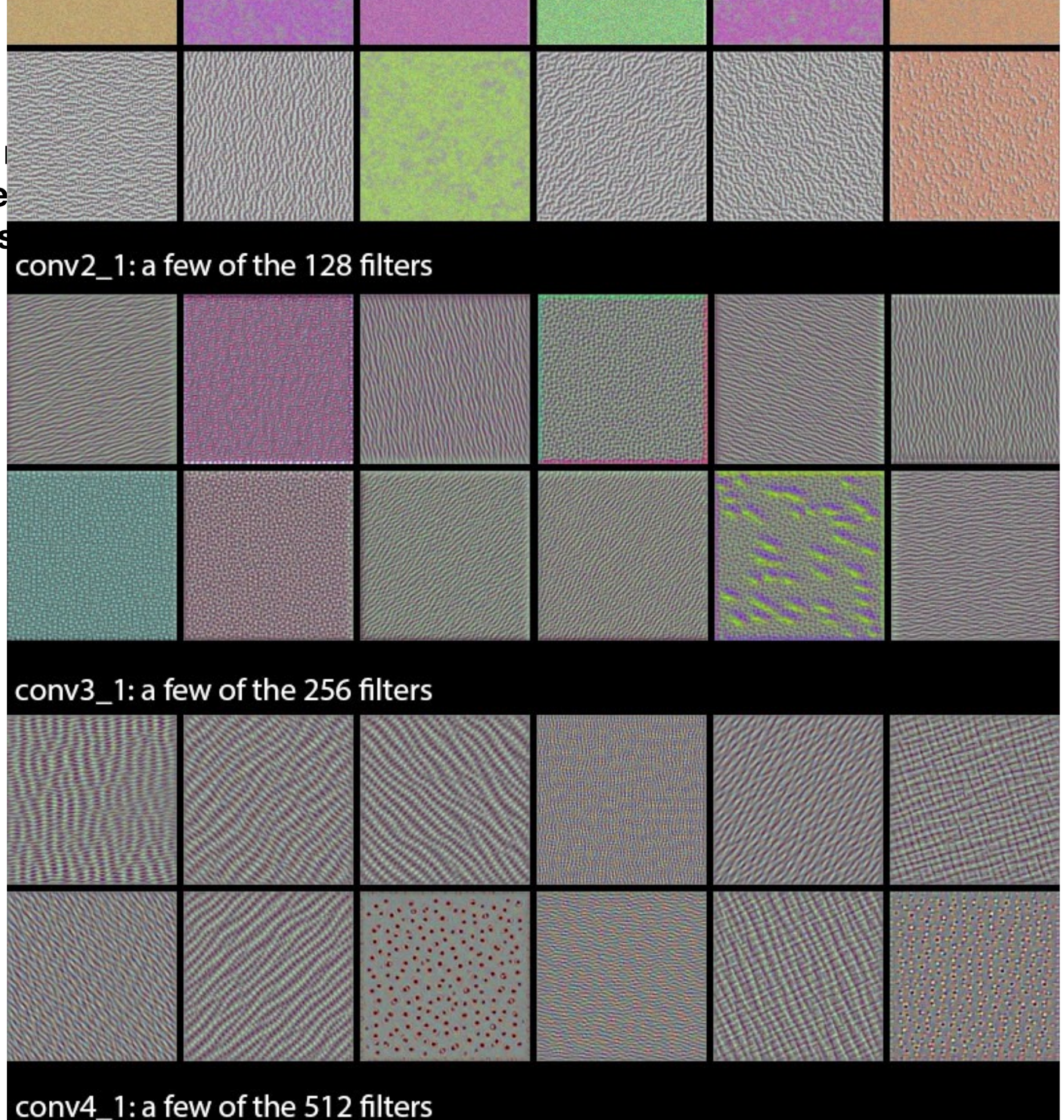
Slide credit: Francois Chollet

What sort of images maximise the activity for a given neutron in each layers?



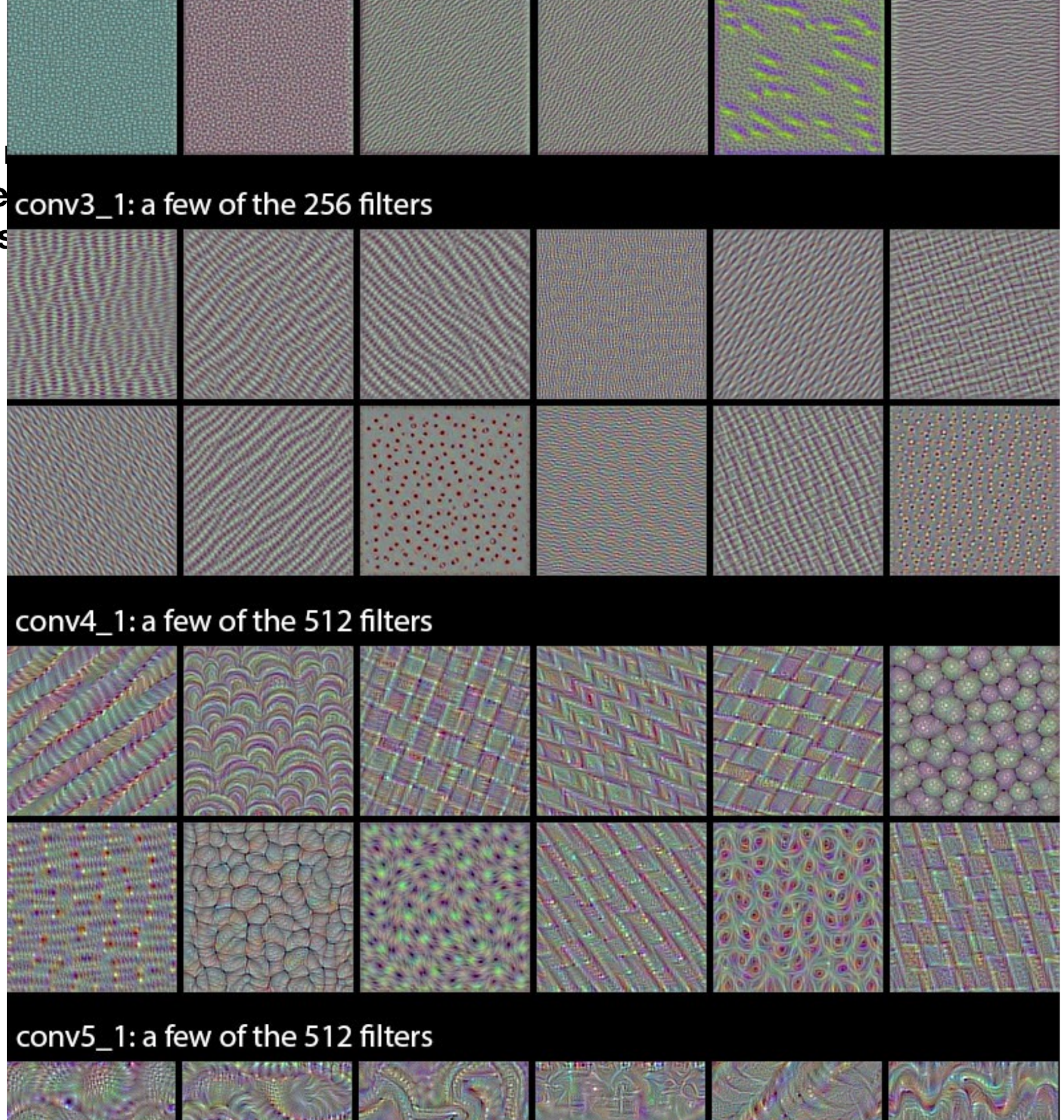
Slide credit: Francois Chollet

What sort of images
the activity for a give
in each layers



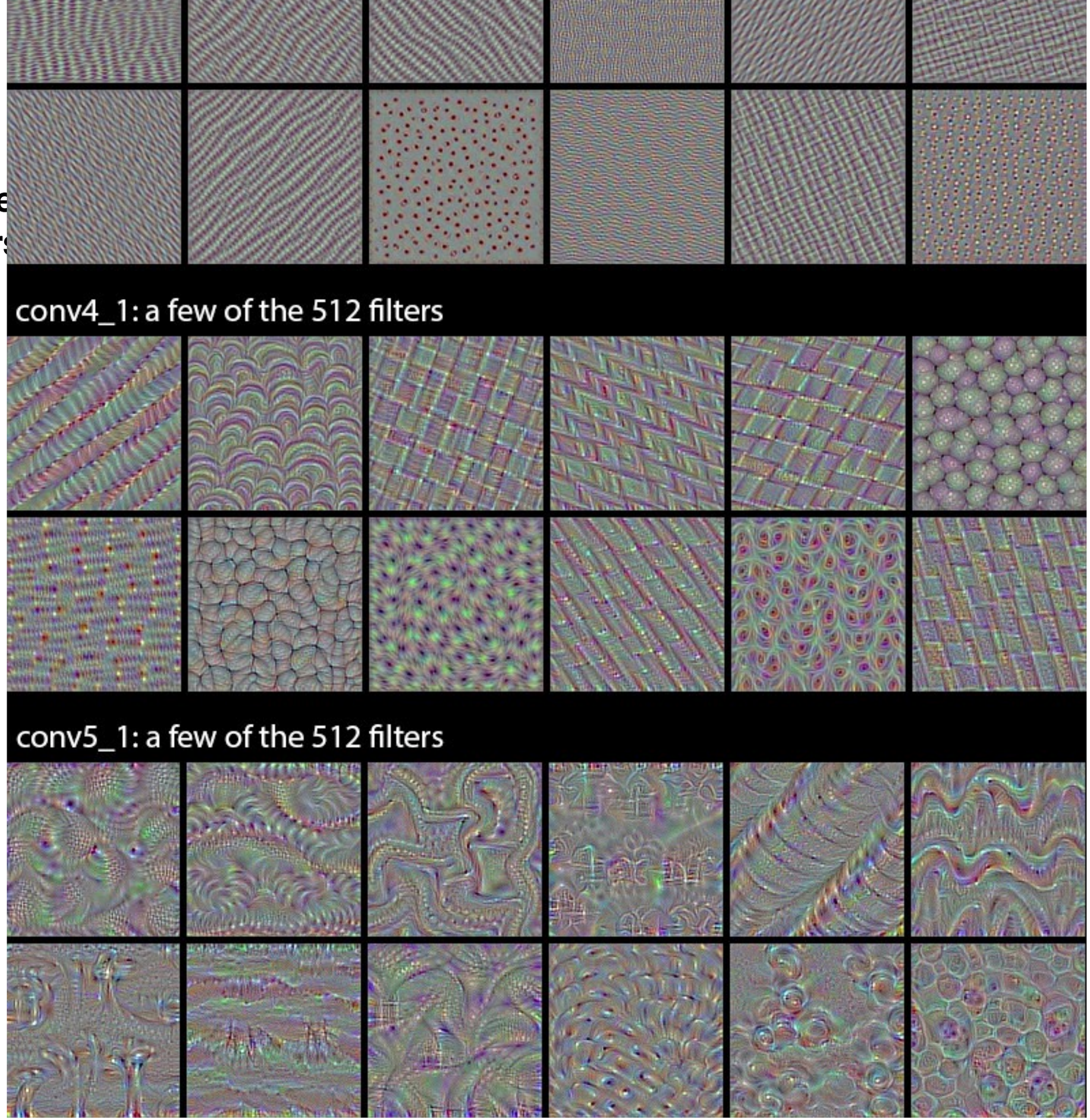
Slide credit: Francois Chollet

What sort of images
the activity for a give
in each layers



Slide credit: Francois Chollet

What sort of images
the activity for a give
in each layer



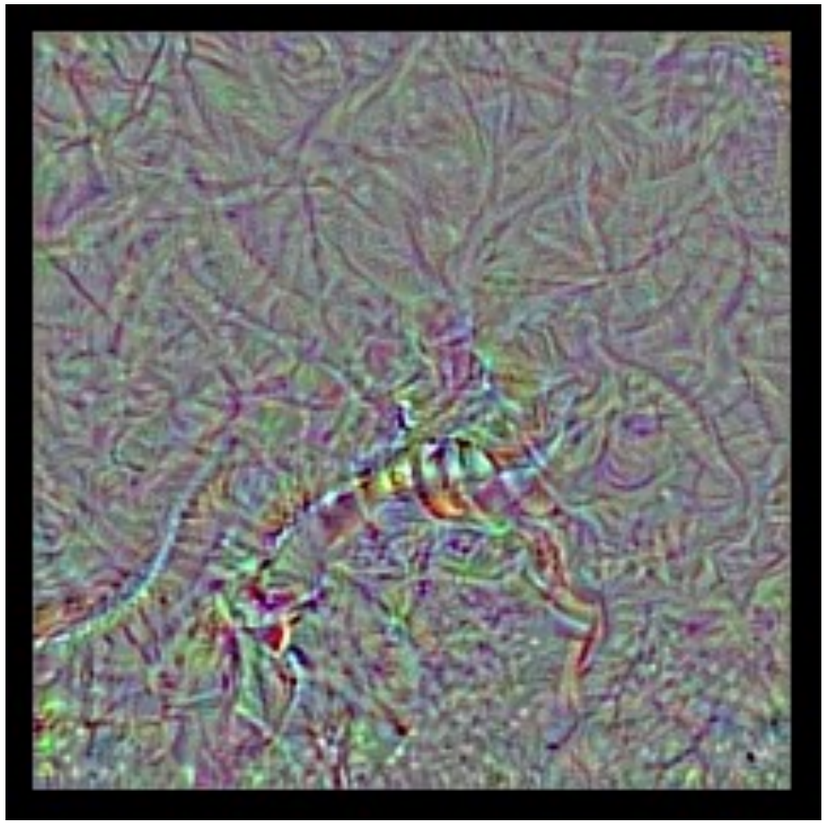
Slide credit: Francois Chollet

What sort of images maximise the activity for a given neutron in each layers?

What sort of images maximise the activity for <u>the final neutron</u> For a given category?

Let's try with a see-snake!
(vgg-16 trainde on imagenet With hundred categories)

This is a see-snake
« I am 99% positive! »



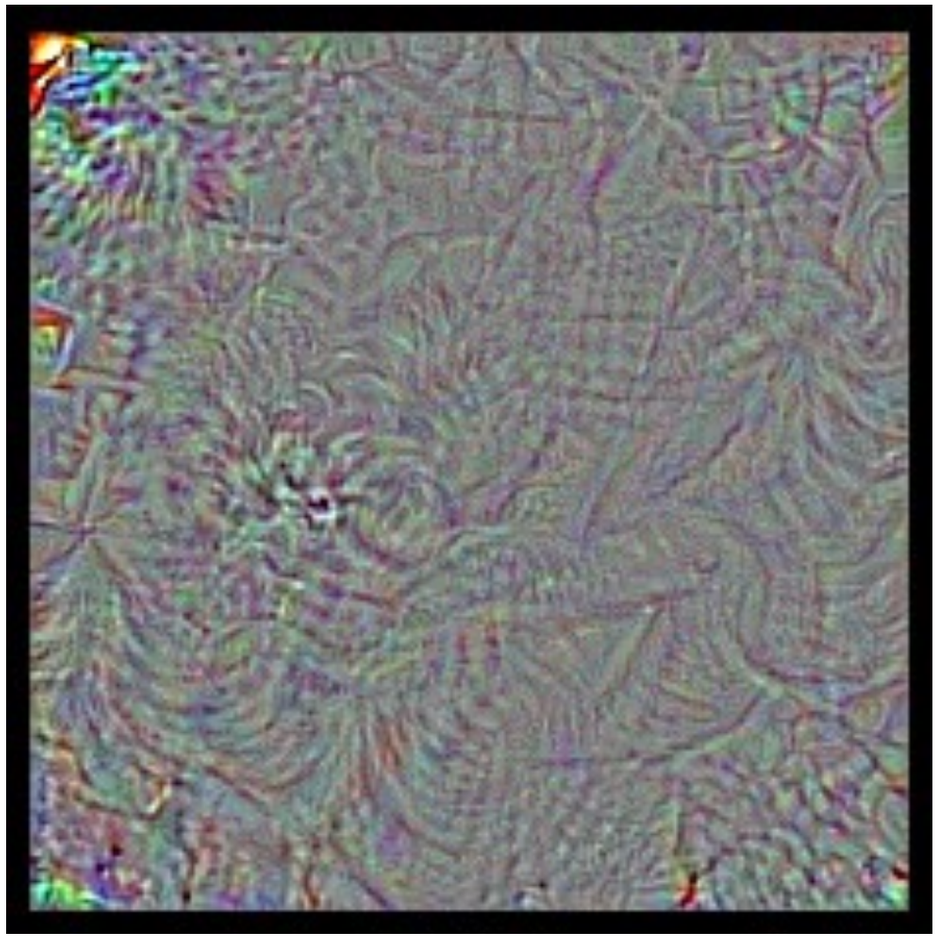
Slide credit: Francois Chollet

What sort of images maximise the activity for a given neutron in each layers?

What sort of images maximise the activity for <u>the final neutron</u> For a given category?

Let's try with a magpie!
(vgg-16 trainde on imagenet With hundred categories)

This is a magpie
« I am 99% positive! »



Slide credit: Francois Chollet

Computer Vision is Easy: Transfer Learning

(state of art result in few minutes)

Transfer Learning with CNNs

1. Train on ImageNet

2. If small dataset: fix all weights (treat CNN as fixed feature extractor), retrain only the classifier

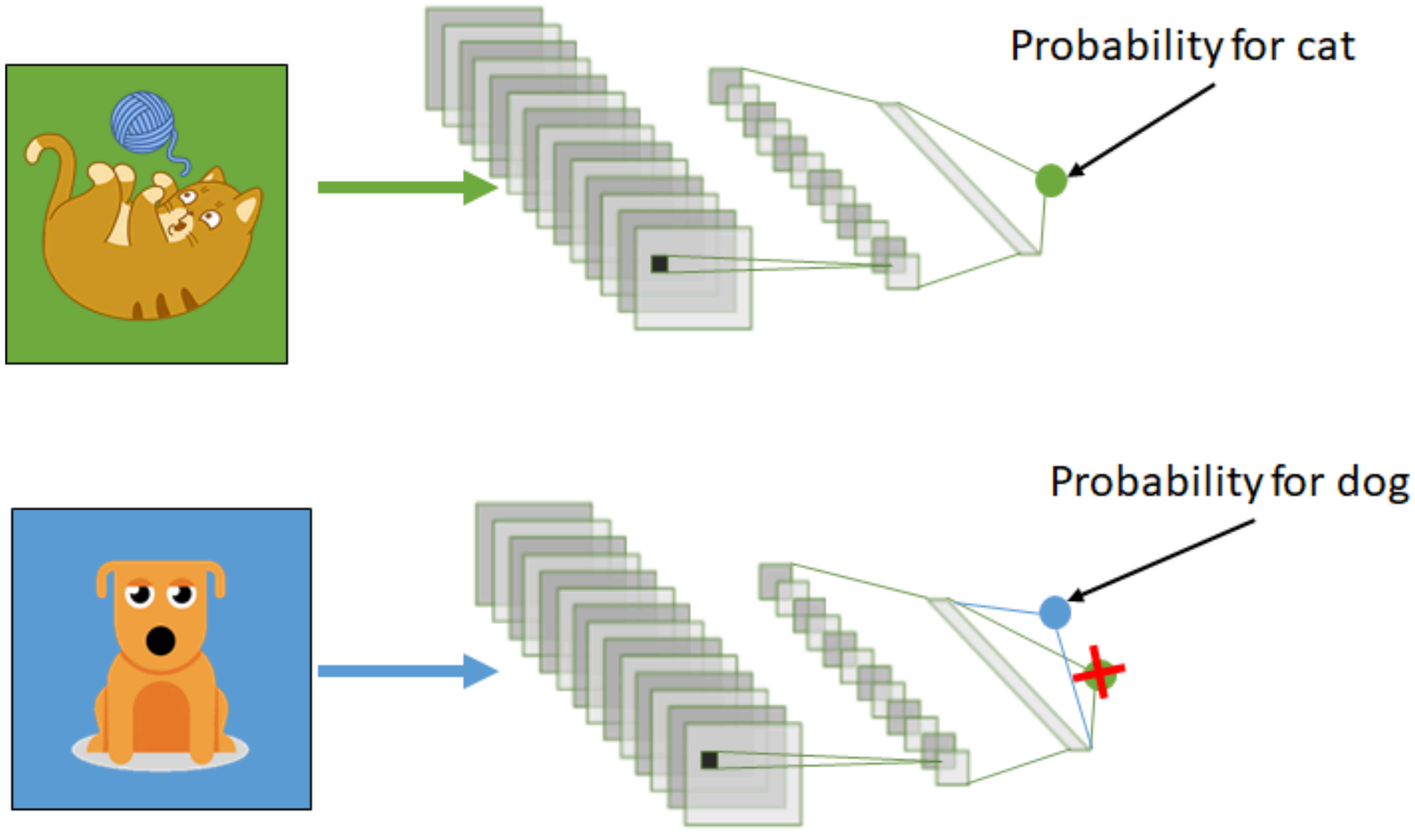
i.e. swap the Softmax layer at the end

3. If you have medium sized dataset, **"finetune"** instead: use the old weights as initialization, train the full network or only some of the higher layers

retrain bigger portion of the network, or even all of it.

Computer Vision is Easy: Transfer Learning

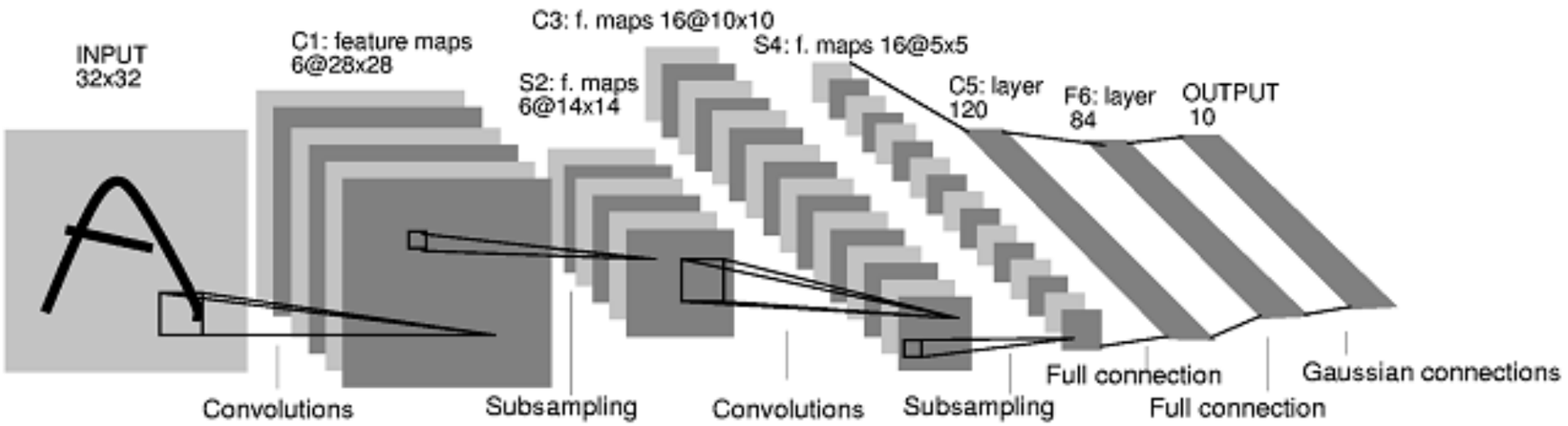
(state of art result in few minutes)



In summary

Convolutional neural nets are the state of there art for images

(NB: Well, Vision Transformer are just as good actually)



They are made by adding Convolution and pooling.
Regularization (dropout, batch norm) also help.
This allows to solve almost any supervised vision problem:
all computer vision is now using convents