Introduction to Deep Learning

Anastasios Tefas, Paraskevi Nousi

Department of Informatics
Aristotle University of Thessaloniki

March 2023

Slides edited from: Francois Fleuret, David Silver

Overview

Introduction

1949 – Donald Hebb proposes the Hebbian Learning principle.

1951 – Marvin Minsky creates the first ANN (Hebbian learning, 40 neurons).

1958 – Frank Rosenblatt creates a perceptron to classify 20×20 images.

1959 – David H. Hubel and Torsten Wiesel demonstrate orientation selectivity and columnar organization in the cat's visual cortex.

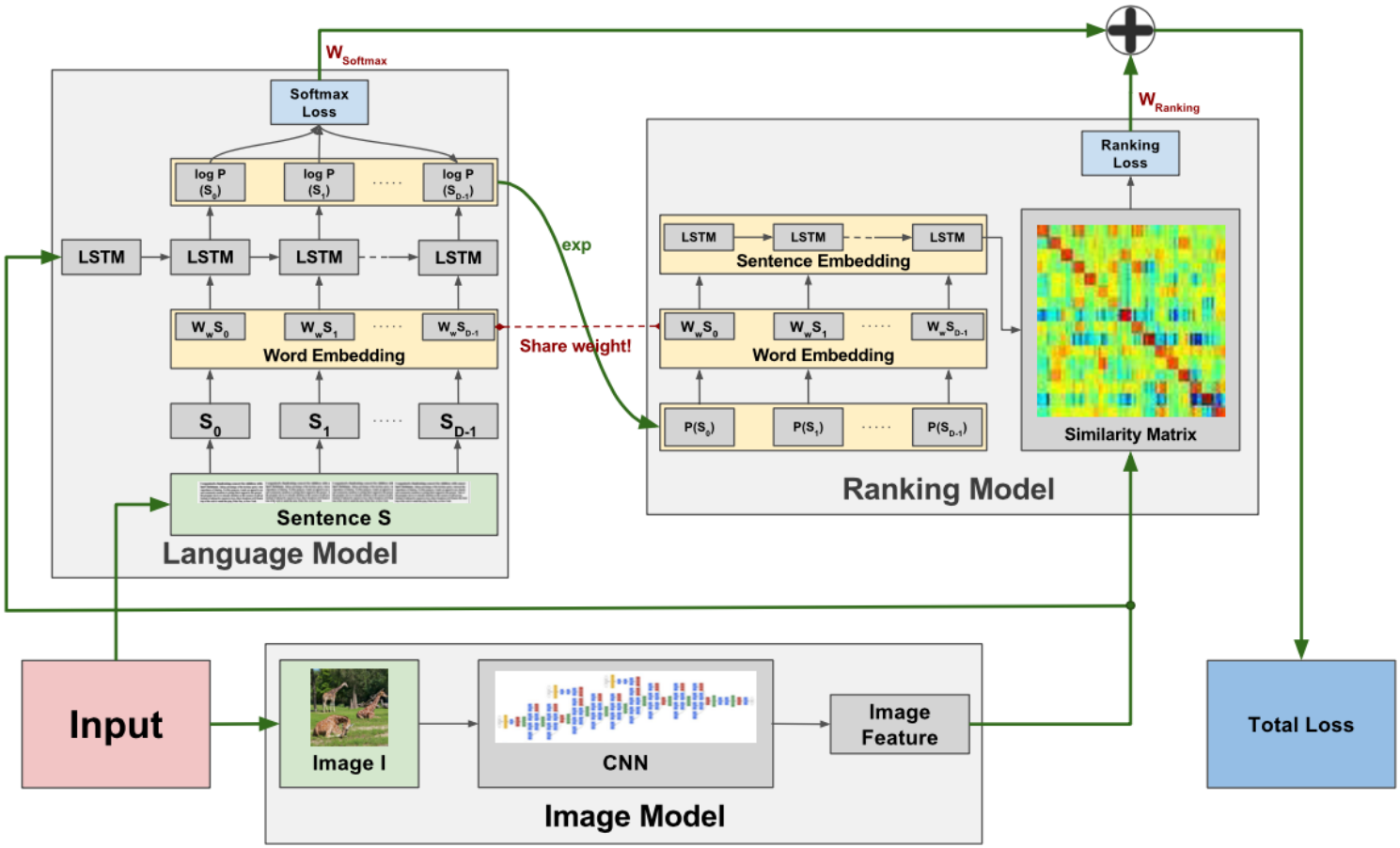
1982 – Paul Werbos proposes back-propagation for ANNs.

Deep learning is built on a natural generalization of a neural network: **a graph of tensor operators**, taking advantage of

- the chain rule (aka "back-propagation"),
- stochastic gradient decent,
- convolutions,
- parallel operations on GPUs.

This does not differ much from networks from the 90s

This generalization allows to design complex networks of operators dealing with images, sound, text, sequences, etc. and to train them end-to-end.

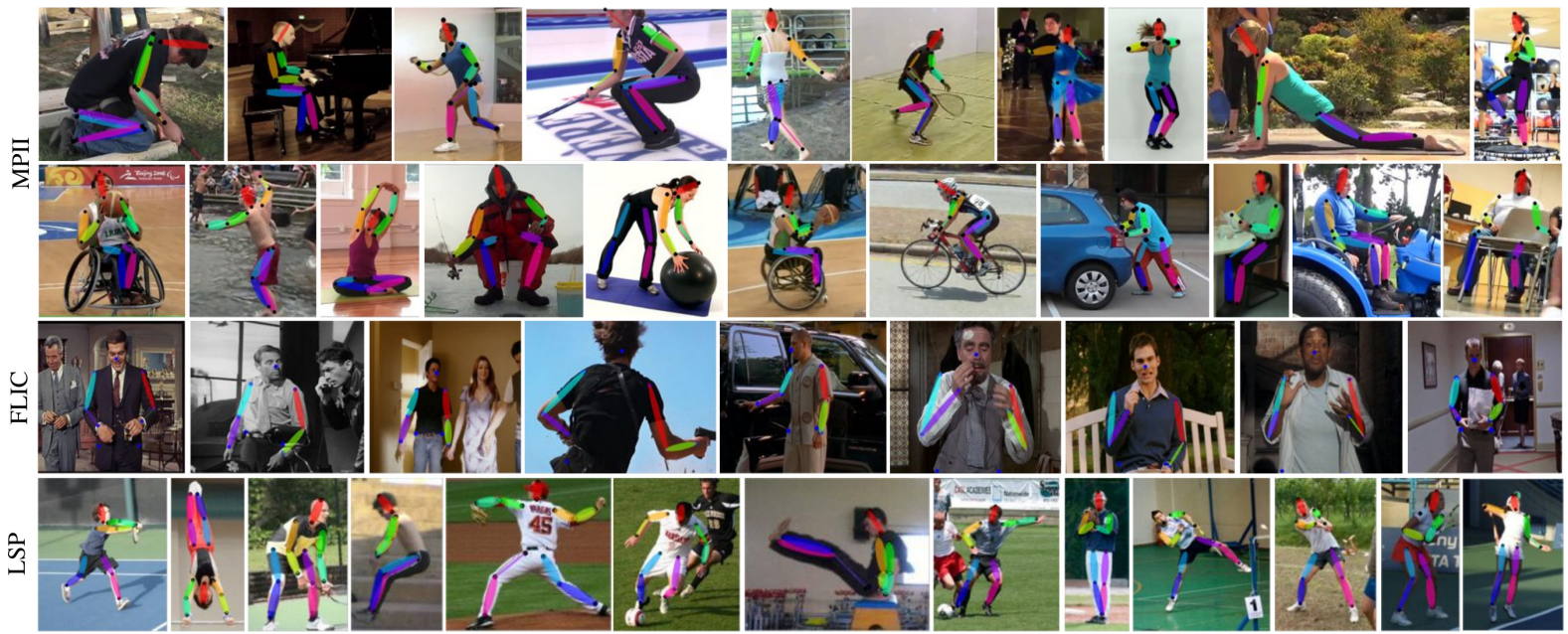


(Yeung et al., 2015)

Object detection and segmentation

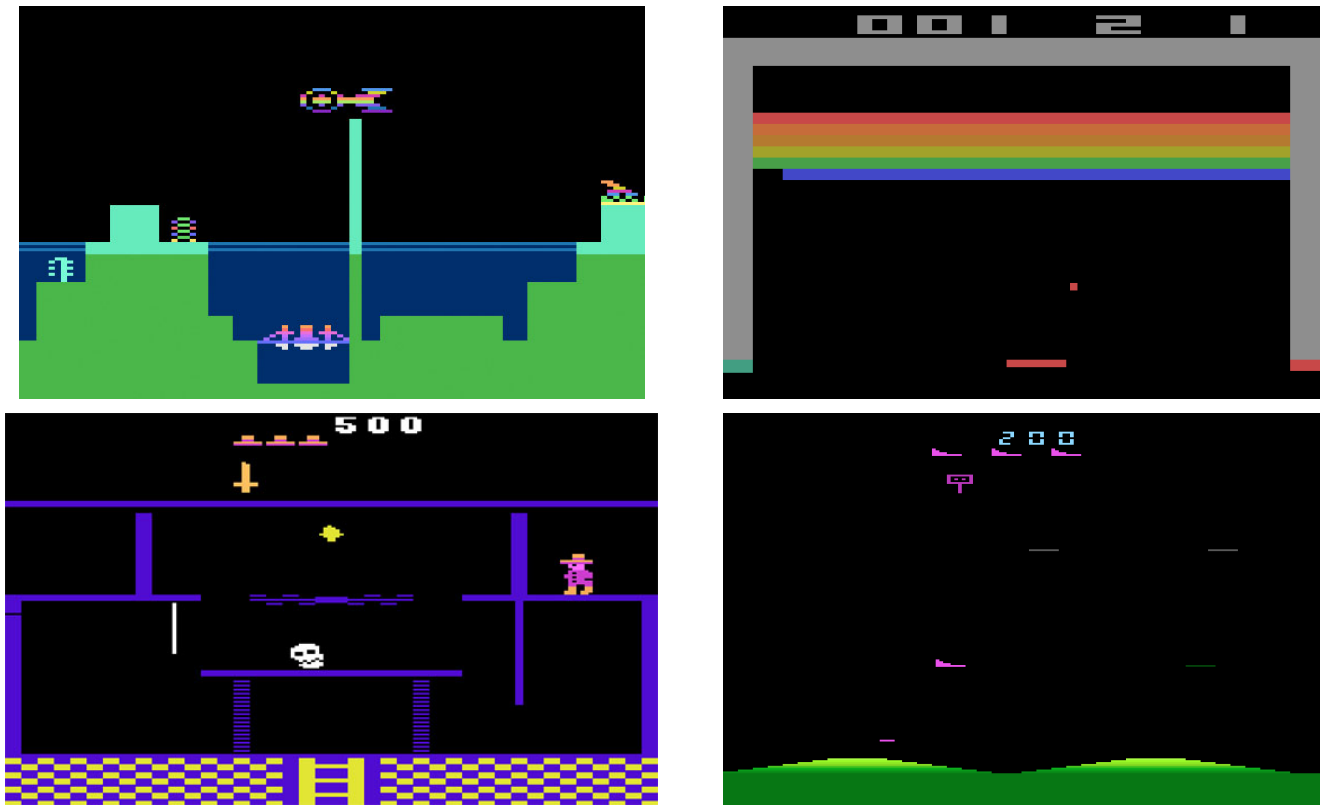
(Pinheiro et al., 2016)

Human pose estimation



(Wei et al., 2016)

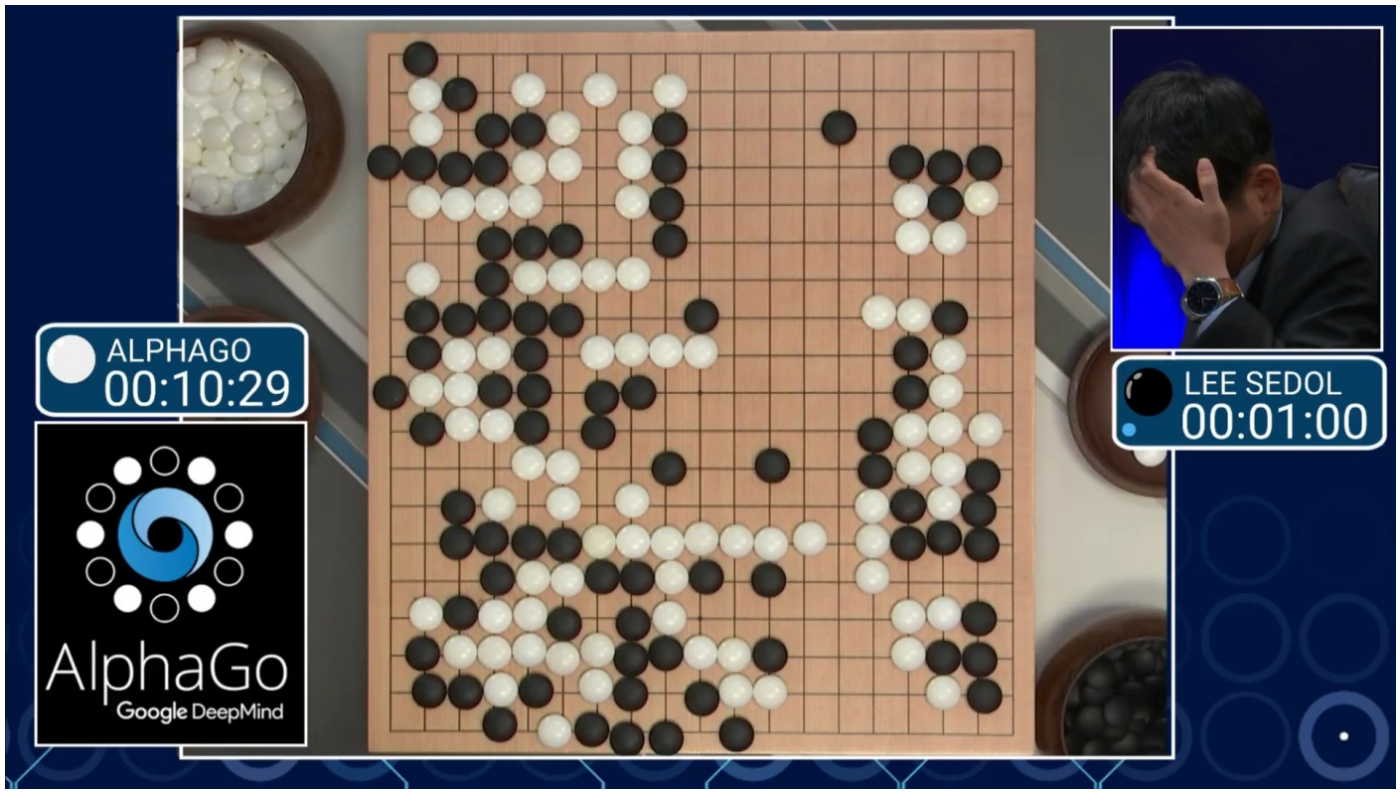
Reinforcement learning



Self-trained, plays 49 games at human level.

(Mnih et al., 2015)

Strategy games



March 2016, 4-1 against a 9-dan professional without handicap.

(Silver et al., 2016)

Image generation

(Brock et al., 2018)

Text generation

System Prompt (human-written)

```
In a shocking finding, scientist discovered a herd of unicorns living
in a remote, previously unexplored valley, in the Andes
Mountains. Even more surprising to the researchers was the fact that
the unicorns spoke perfect English.
```

Model Completion (machine-written, 10 tries)

```
The scientist named the population, after their distinctive horn,
Ovid's Unicorn. These four-horned, silver-white unicorns were
previously unknown to science.

Now, after almost two centuries, the mystery of what sparked this odd
phenomenon is finally solved.

Dr. Jorge Pérez, an evolutionary biologist from the University of La
Paz, and several companions, were exploring the Andes Mountains when
they found a small valley, with no other animals or humans. Pérez
noticed that the valley had what appeared to be a natural fountain,
surrounded by two peaks of rock and silver snow.
```

(?)

Why does it work now?

The success of deep learning is multi-factorial:

- Five decades of research in machine learning,
- CPUs/GPUs/storage developed for other purposes,
- lots of data from "the internet",
- tools and culture of collaborative and reproducible science,
- resources and efforts from large corporations.

From a practical perspective, deep learning

- lessens the need for a deep mathematical grasp,
- makes the design of large learning architectures a system/software development task,
- allows to leverage modern hardware (clusters of GPUs),
- does not plateau when using more data,
- makes large trained networks a commodity.

From Perceptron to Multilayer Perceptron

The first mathematical model for a neuron was the Threshold Logic Unit, with Boolean inputs and outputs:

$$f(x) = \mathbf{1}_{\{w \sum_i x_i + b \geq 0\}}.$$

It can in particular implement

$$\begin{aligned} or(u, v) &= \mathbf{1}_{\{u+v-0.5 \geq 0\}} && (w = 1, b = -0.5) \\ and(u, v) &= \mathbf{1}_{\{u+v-1.5 \geq 0\}} && (w = 1, b = -1.5) \\ not(u) &= \mathbf{1}_{\{-u+0.5 \geq 0\}} && (w = -1, b = 0.5) \end{aligned}$$

Hence, **any Boolean function can be build with such units.**

(McCulloch and Pitts, 1943)

The perceptron is very similar

$$f(x) = \begin{cases} 1 & \text{if} \quad \sum_i w_i \, x_i + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

but the inputs are real values and the weights can be different.

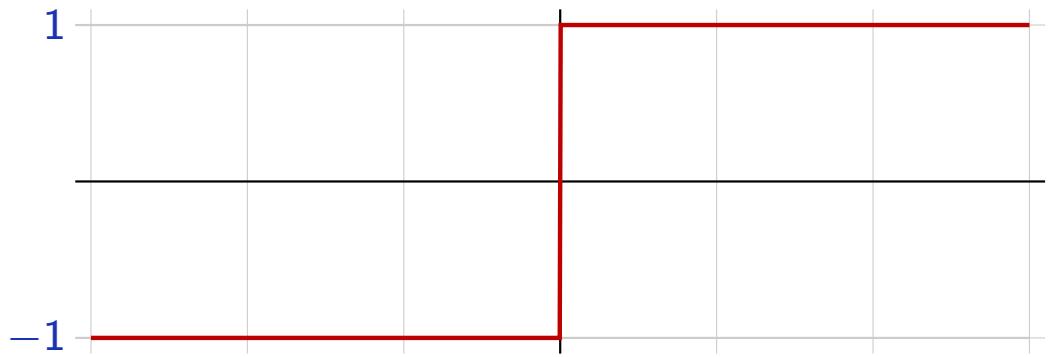
This model was originally motivated by biology, with w_i being the *synaptic weights*, and x_i and f firing rates.

It is a (very) crude biological model.

(Rosenblatt, 1957)

To make things simpler we take responses ± 1. Let

$$\sigma(x) = \begin{cases} 1 & \text{if} \quad x \geq 0 \\ -1 & \text{otherwise.} \end{cases}$$

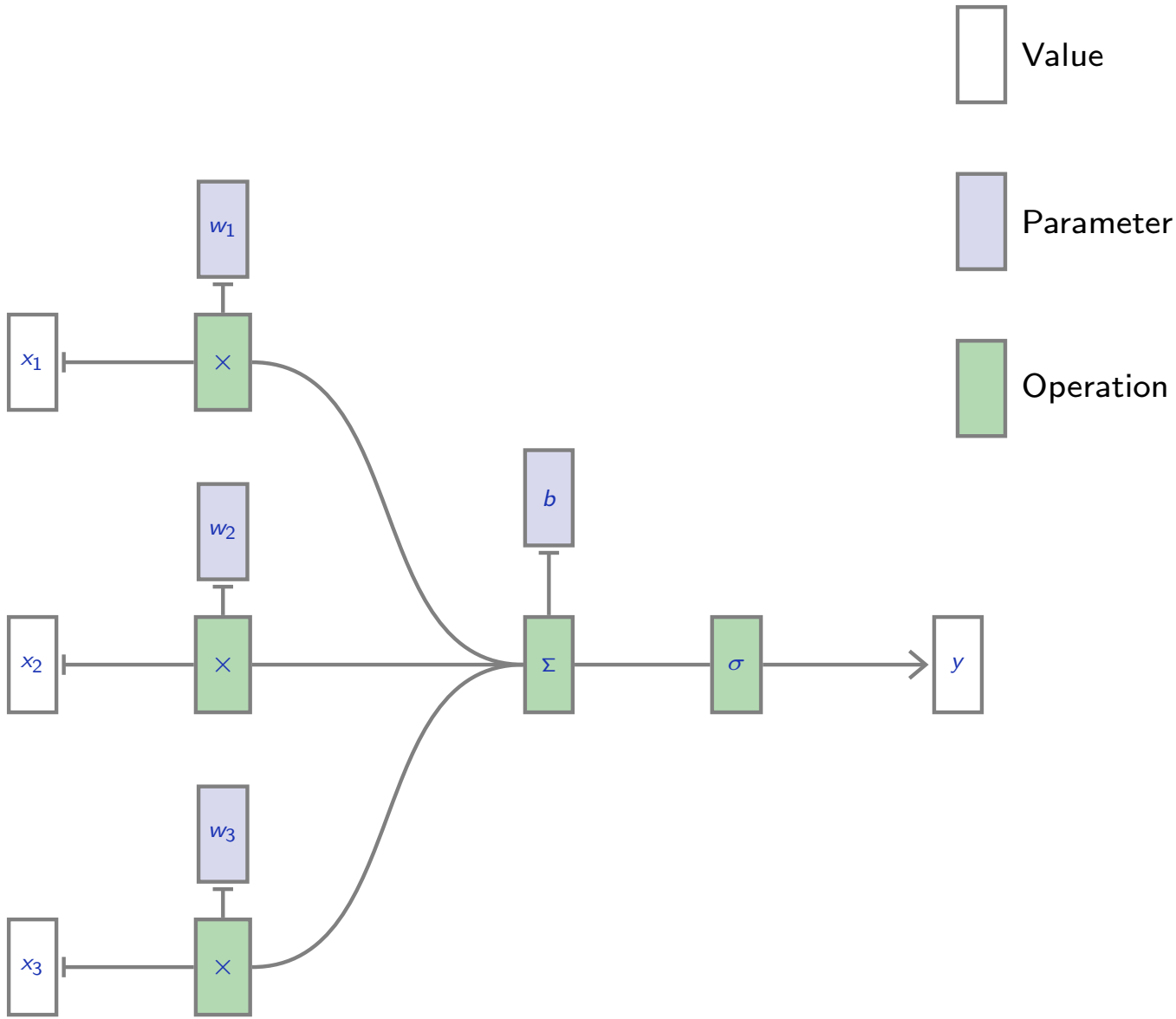


The perceptron classification rule boils down to

$$f(x) = \sigma(w \cdot x + b).$$

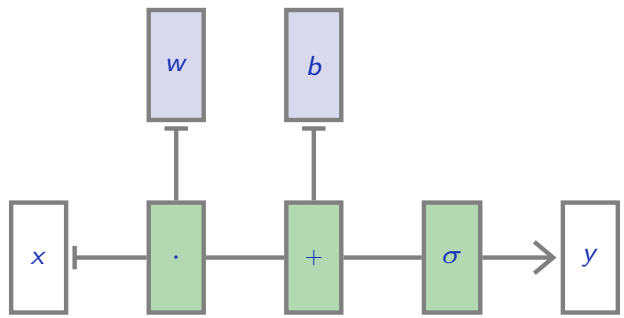
For neural networks, the function σ that follows a linear operator is called the **activation function**.

We can represent this "neuron" as follows:



We can also use tensor operations, as in

$$f(x) = \sigma(w \cdot x + b).$$



Given a training set

$$(x_n, y_n) \in \mathbb{R}^D \times \{-1, 1\}, \quad n = 1, \ldots, N,$$

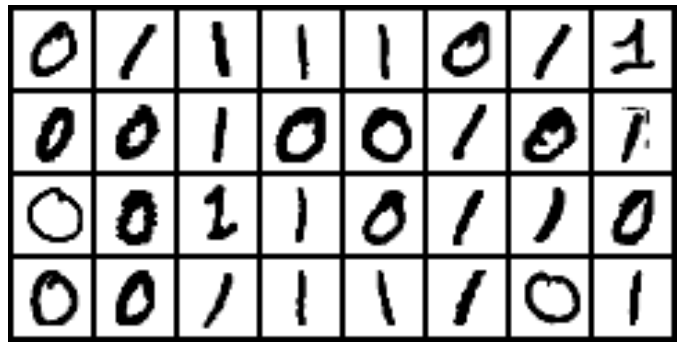
a very simple scheme to train such a linear operator for classification is the **perceptron algorithm:**

1. Start with $w^0 = 0$,
2. while $\exists n_k$ s.t. $y_{n_k} \left(w^k \cdot x_{n_k} \right) \leq 0$, update $w^{k+1} = w^k + y_{n_k} x_{n_k}$.

The bias b can be introduced as one of the ws by adding a constant component to x equal to 1.

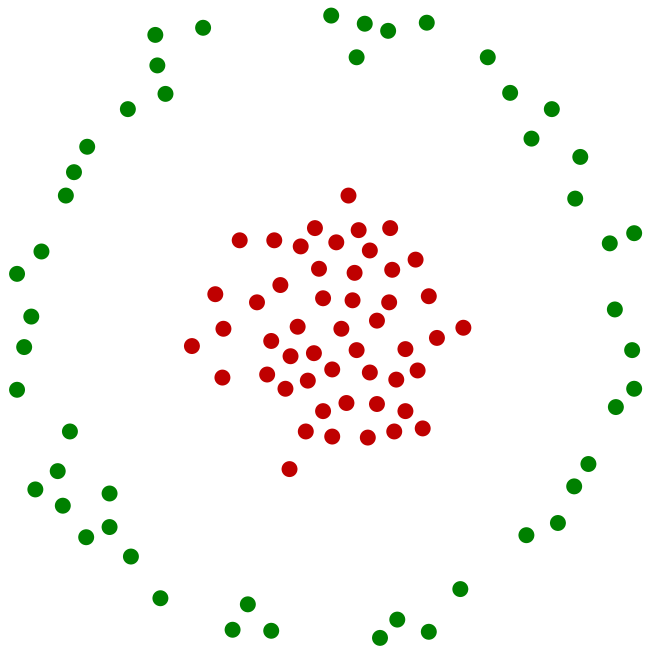
(Rosenblatt, 1957)

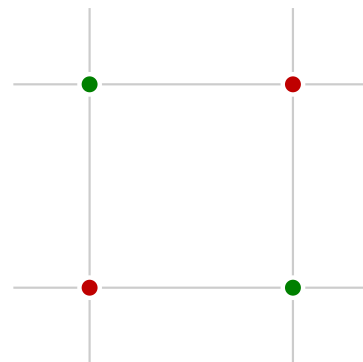
This crude algorithm works often surprisingly well. With MNIST's "0"s as negative class, and "1"s as positive one.



```
epoch 0 nb_changes 64 train_error 0.23% test_error 0.19%
epoch 1 nb_changes 24 train_error 0.07% test_error 0.00%
epoch 2 nb_changes 10 train_error 0.06% test_error 0.05%
epoch 3 nb_changes 6 train_error 0.03% test_error 0.14%
epoch 4 nb_changes 5 train_error 0.03% test_error 0.09%
epoch 5 nb_changes 4 train_error 0.02% test_error 0.14%
epoch 6 nb_changes 3 train_error 0.01% test_error 0.14%
epoch 7 nb_changes 2 train_error 0.00% test_error 0.14%
epoch 8 nb_changes 0 train_error 0.00% test_error 0.14%
```

The main weakness of linear predictors is their lack of capacity. For classification, the populations have to be **linearly separable**.





"xor"

Multilayer Perceptron

A linear classifier of the form

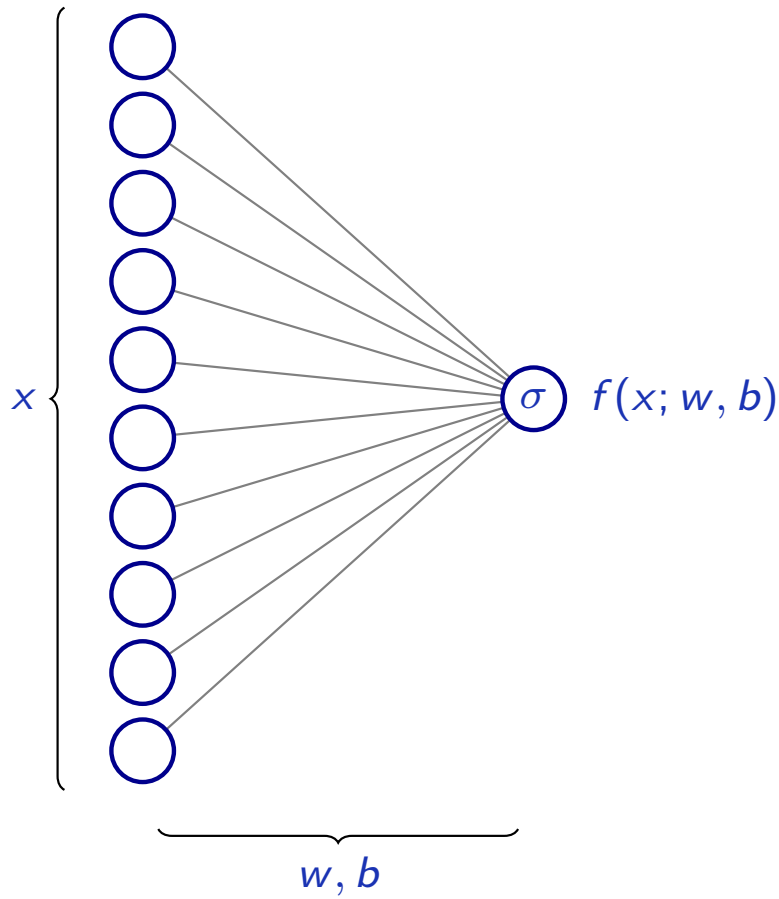
$$\begin{aligned} \mathbb{R}^D &\to \mathbb{R} \\ x &\mapsto \sigma(w \cdot x + b), \end{aligned}$$

with $w \in \mathbb{R}^D$, $b \in \mathbb{R}$, and $\sigma : \mathbb{R} \to \mathbb{R}$, can naturally be extended to a multi-dimension output by applying a similar transformation to every output

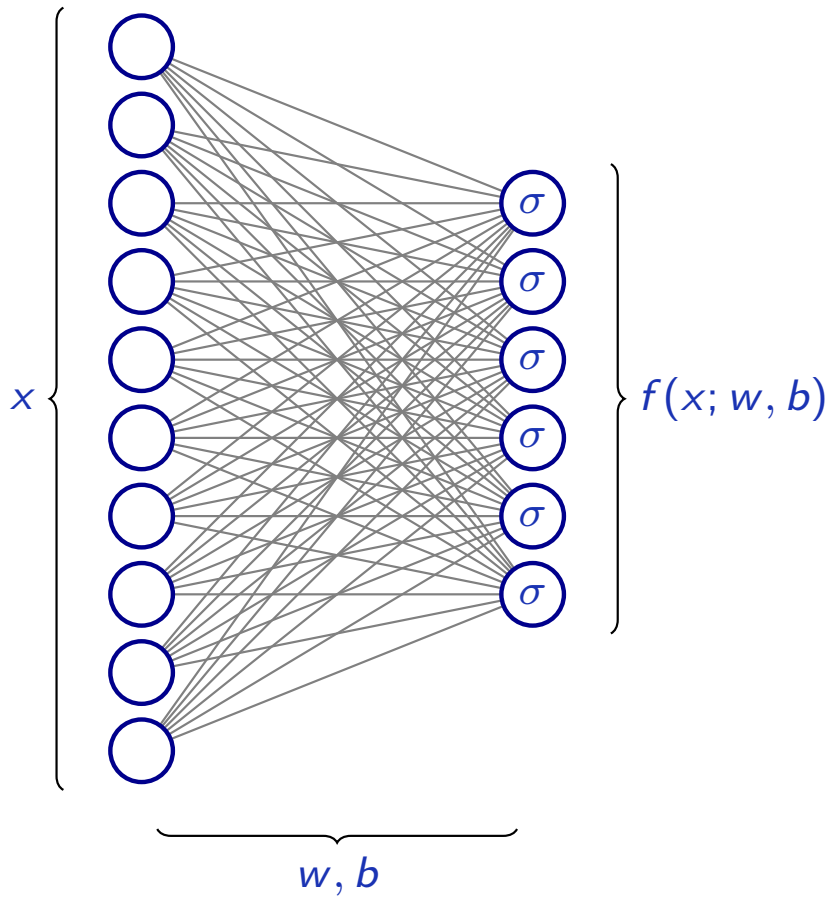
$$\begin{aligned} \mathbb{R}^D &\to \mathbb{R}^C \\ x &\mapsto \sigma(wx + b), \end{aligned}$$

with $w \in \mathbb{R}^{C \times D}$, $b \in \mathbb{R}^C$, and σ is applied component-wise.

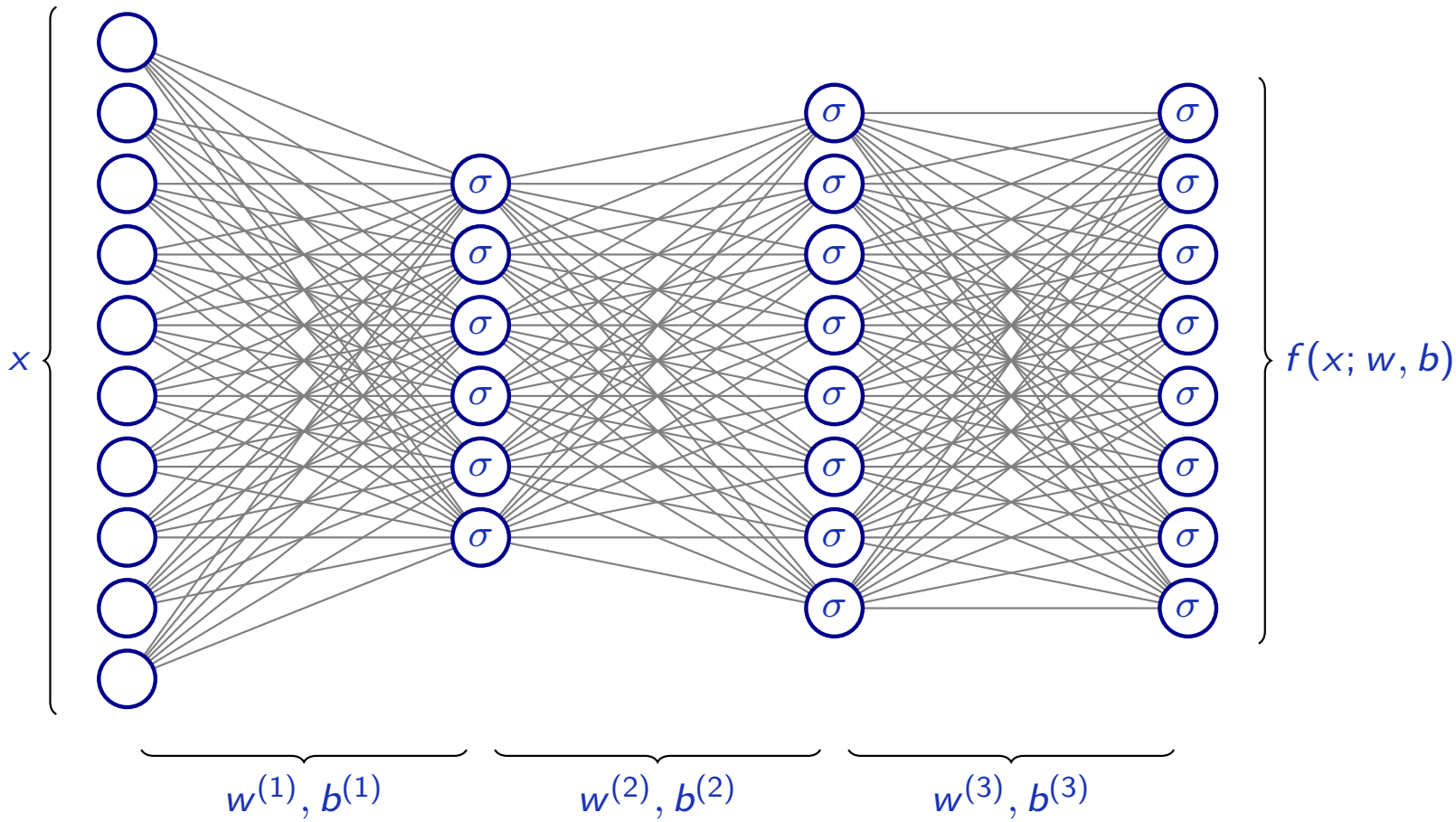
Even though it has no practical value implementation-wise, we can represent such a model as a combination of units. More importantly, we can extend it.



Even though it has no practical value implementation-wise, we can represent such a model as a combination of units. More importantly, we can extend it.



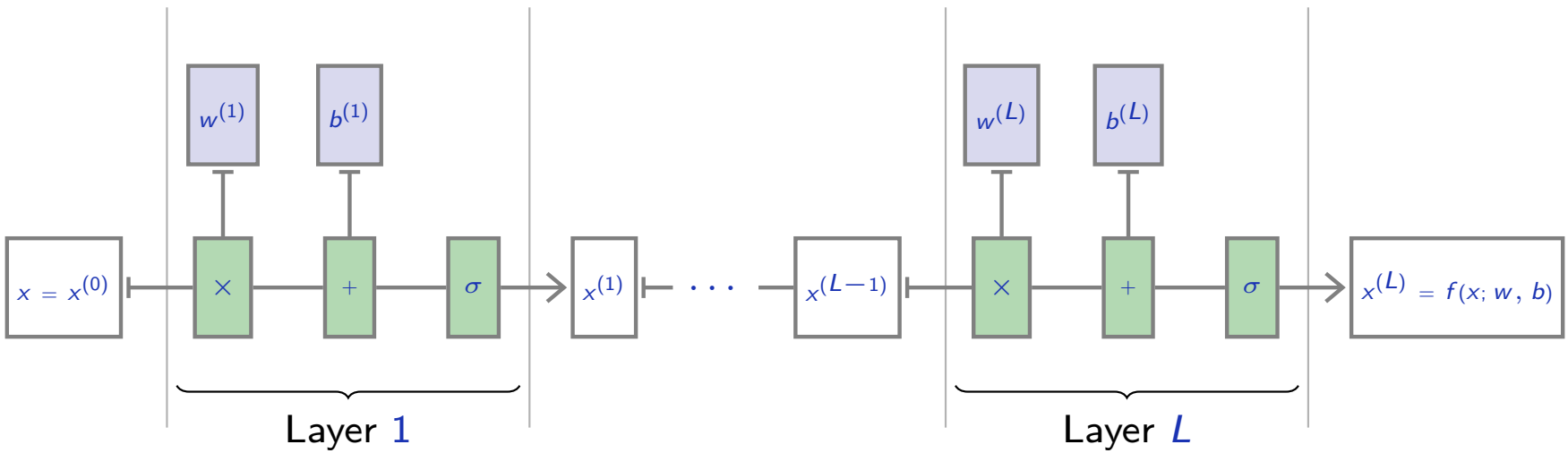
Even though it has no practical value implementation-wise, we can represent such a model as a combination of units. More importantly, we can extend it.



This latter structure can be formally defined, with $x^{(0)} = x$,

$$\forall l = 1, \ldots, L, \; x^{(l)} = \sigma\left(w^{(l)} x^{(l-1)} + b^{(l)}\right)$$

and $f(x; w, b) = x^{(L)}$.



Such a model is a **Multi-Layer Perceptron (MLP).**

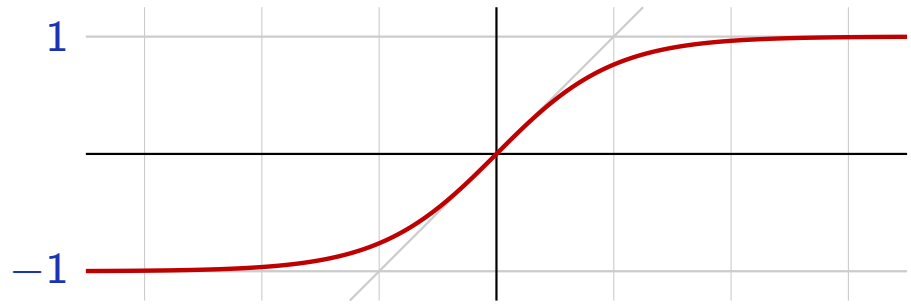
Note that if σ is an affine transformation, the full MLP is a composition of affine mappings, and itself an affine mapping.

Consequently:

The activation function σ should be non-linear, or the resulting MLP is an affine mapping with a peculiar parametrization.

The two classical activation functions are the hyperbolic tangent

$$x \mapsto \frac{2}{1 + e^{-2x}} - 1$$

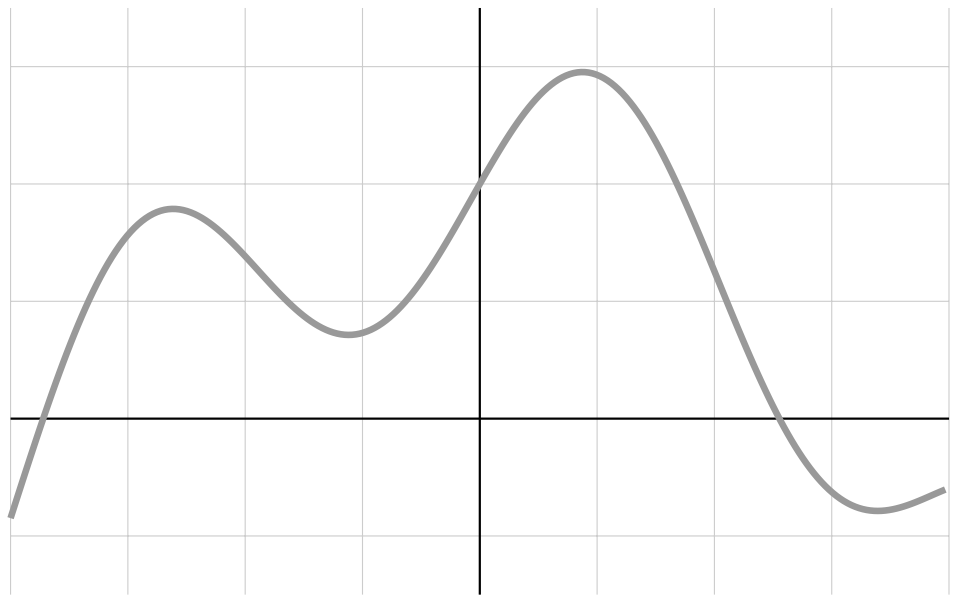


and the rectified linear unit (ReLU)

$$x \mapsto \max(0, x)$$

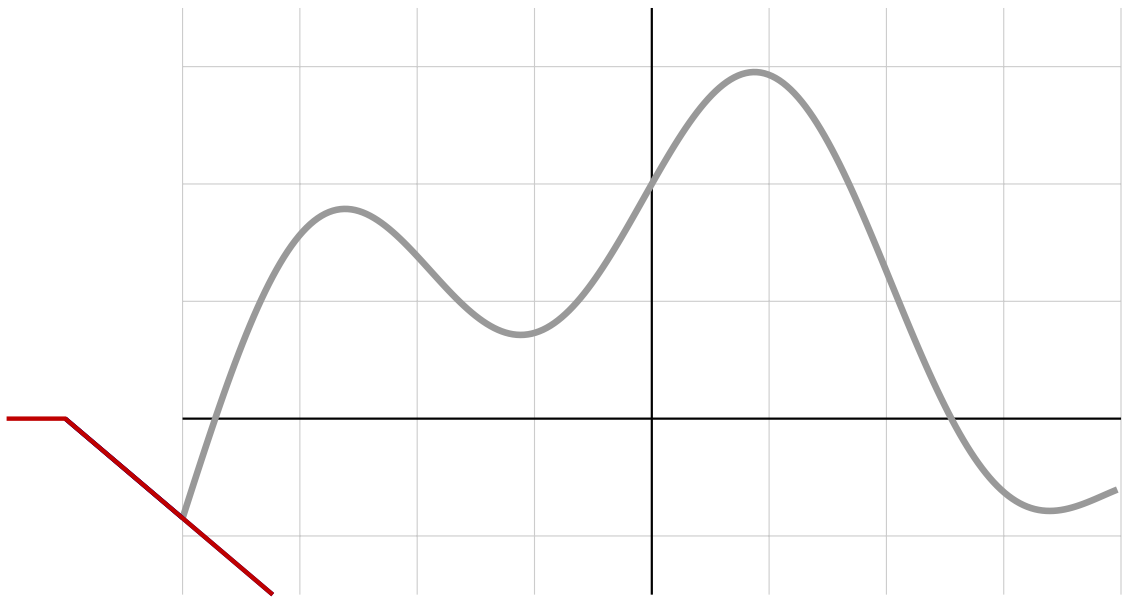
Universal approximation

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



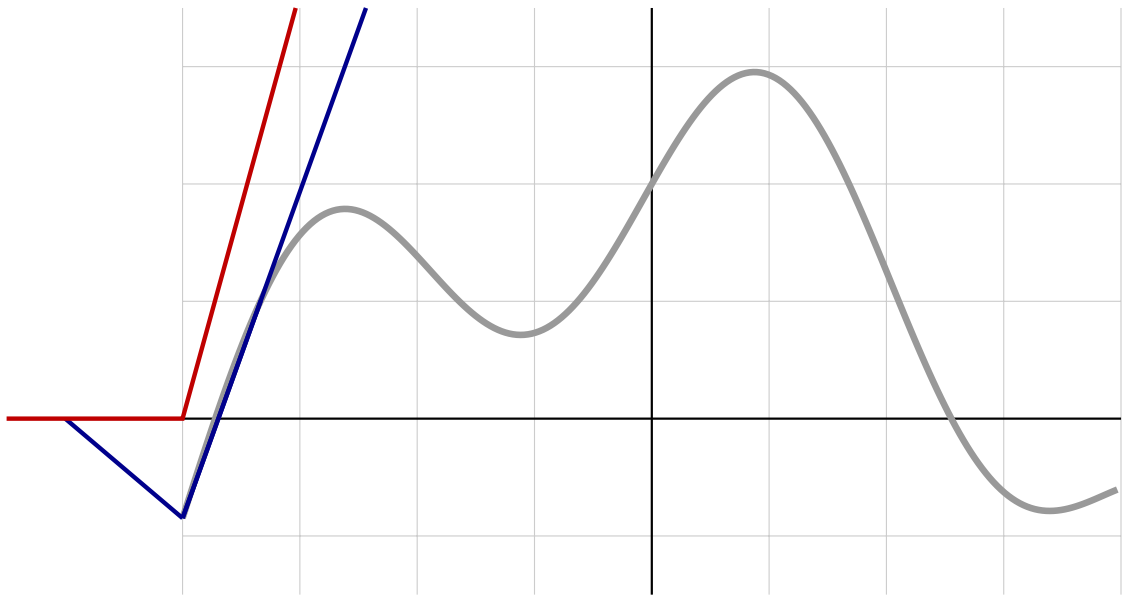
We can approximate any $\psi \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

$$f(x) = \sigma(w_1 x + b_1)$$



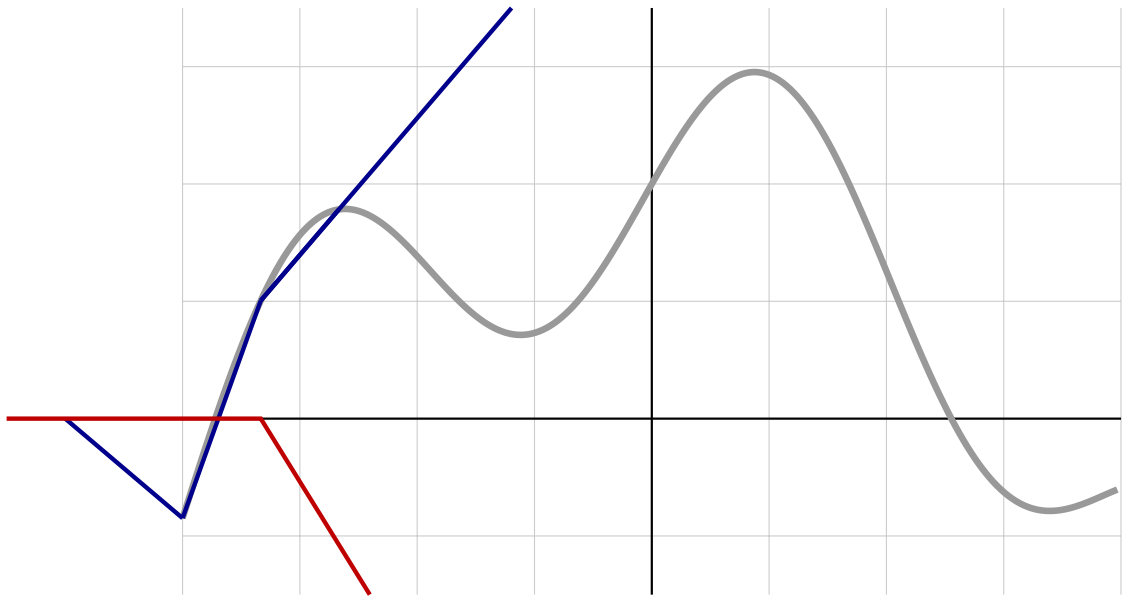
We can approximate any $\psi \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2)$$



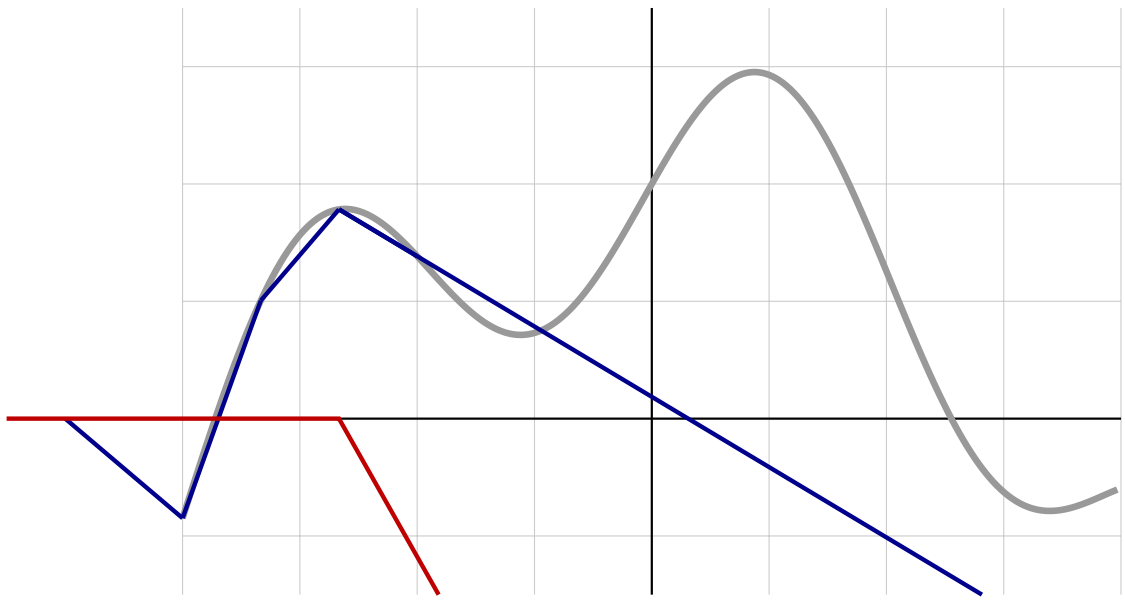
We can approximate any $\psi \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3)$$



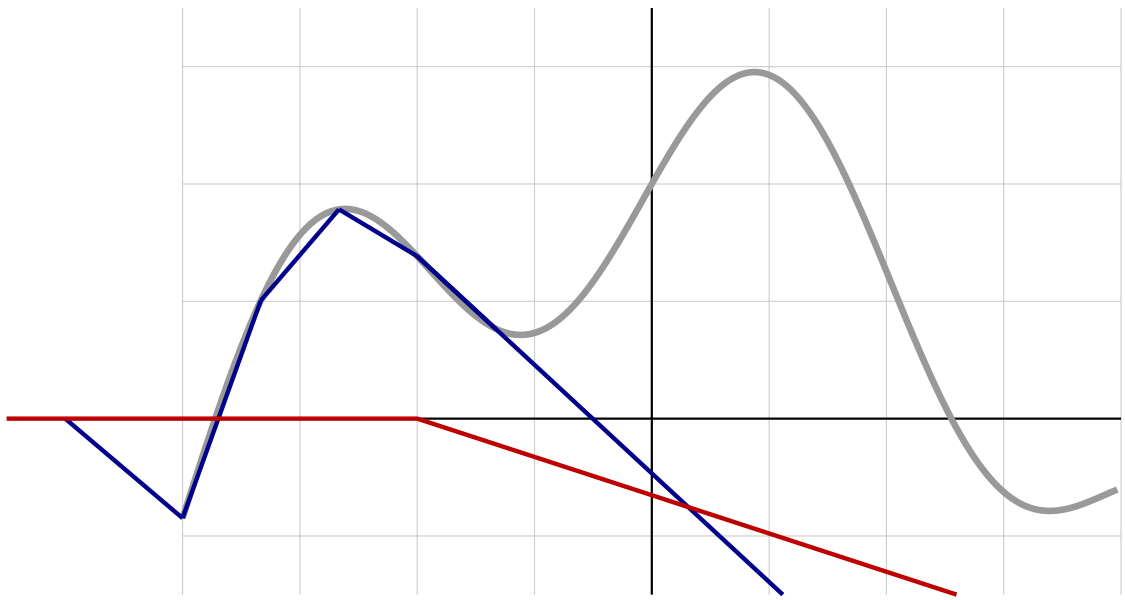
We can approximate any $\psi \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \dots$$



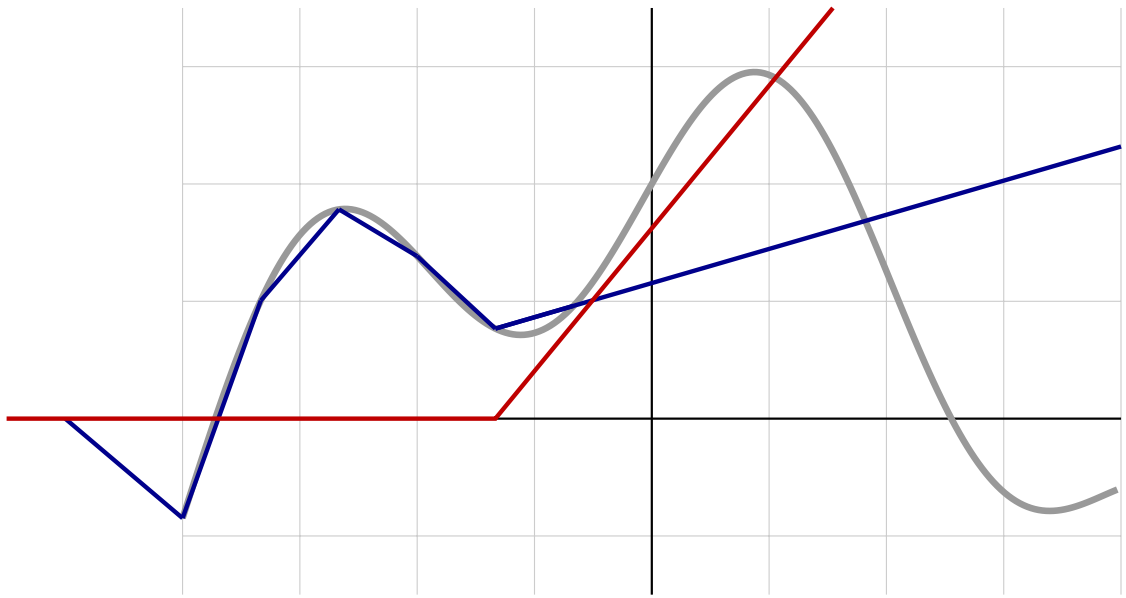
We can approximate any $\psi \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \dots$$



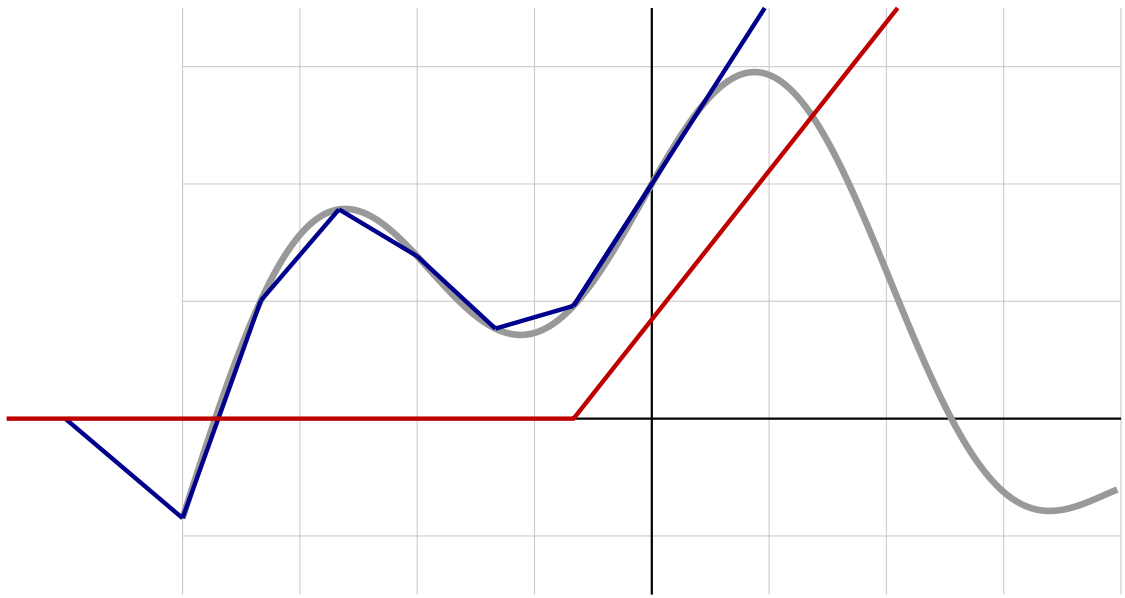
We can approximate any $\psi \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \dots$$



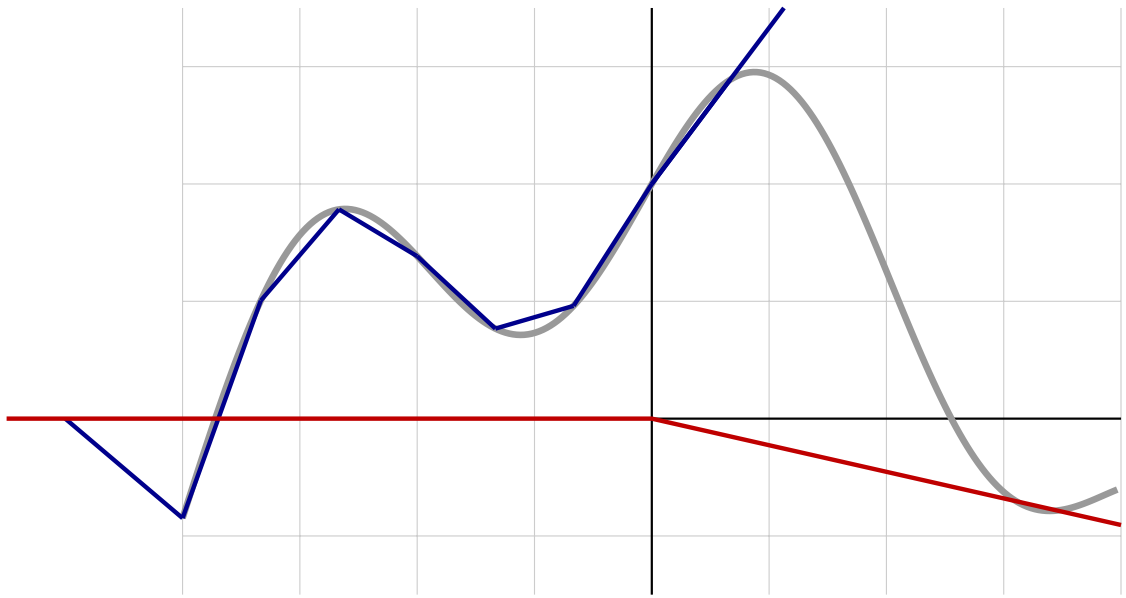
We can approximate any $\psi \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \dots$$



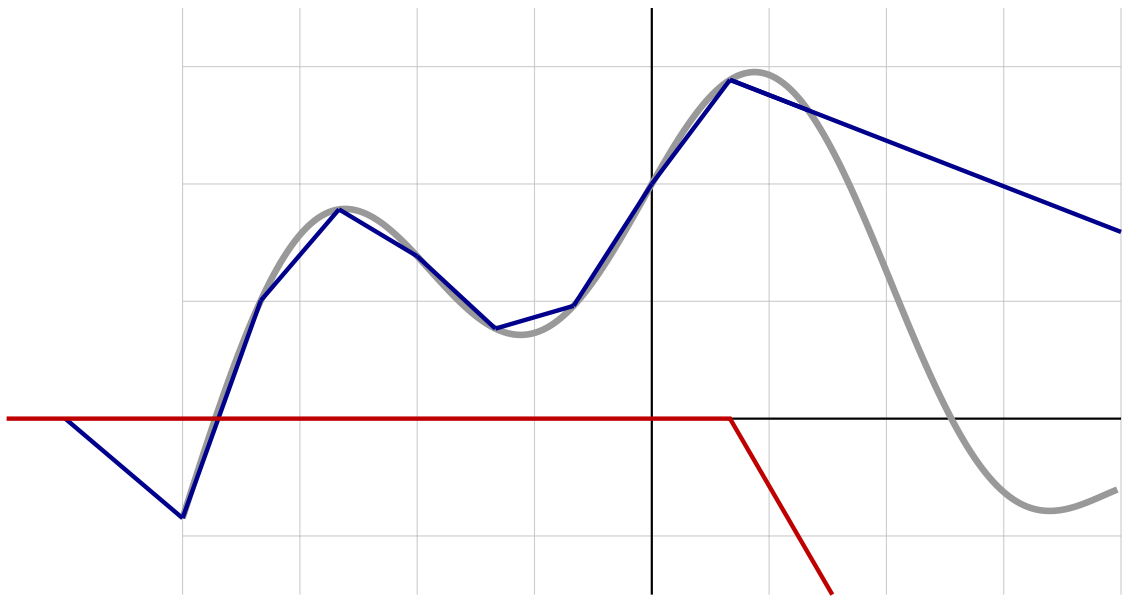
We can approximate any $\psi \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \dots$$



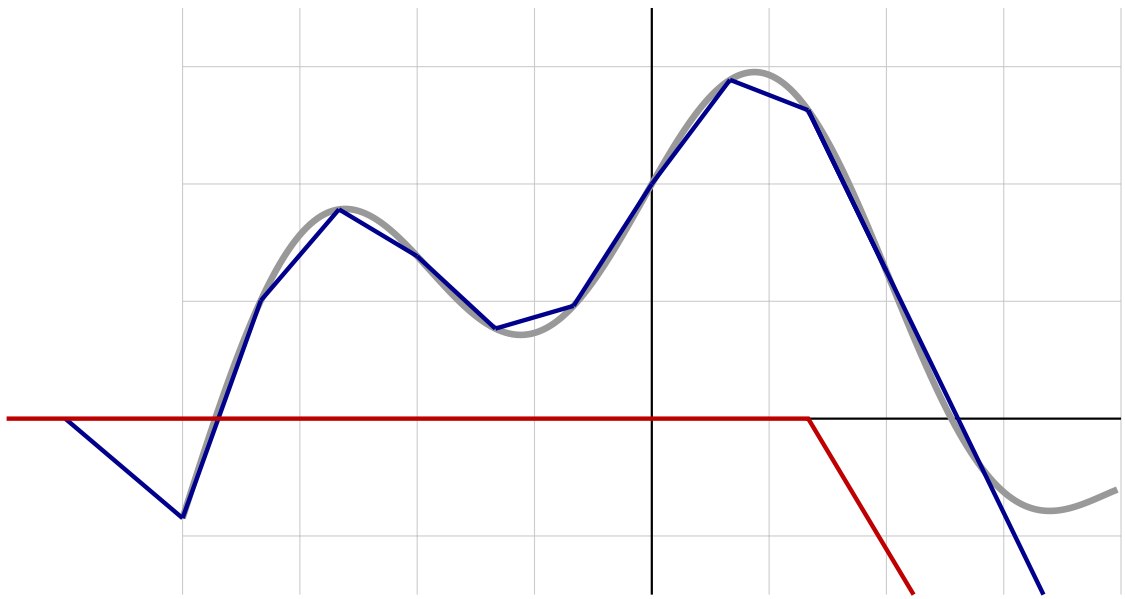
We can approximate any $\psi \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \dots$$



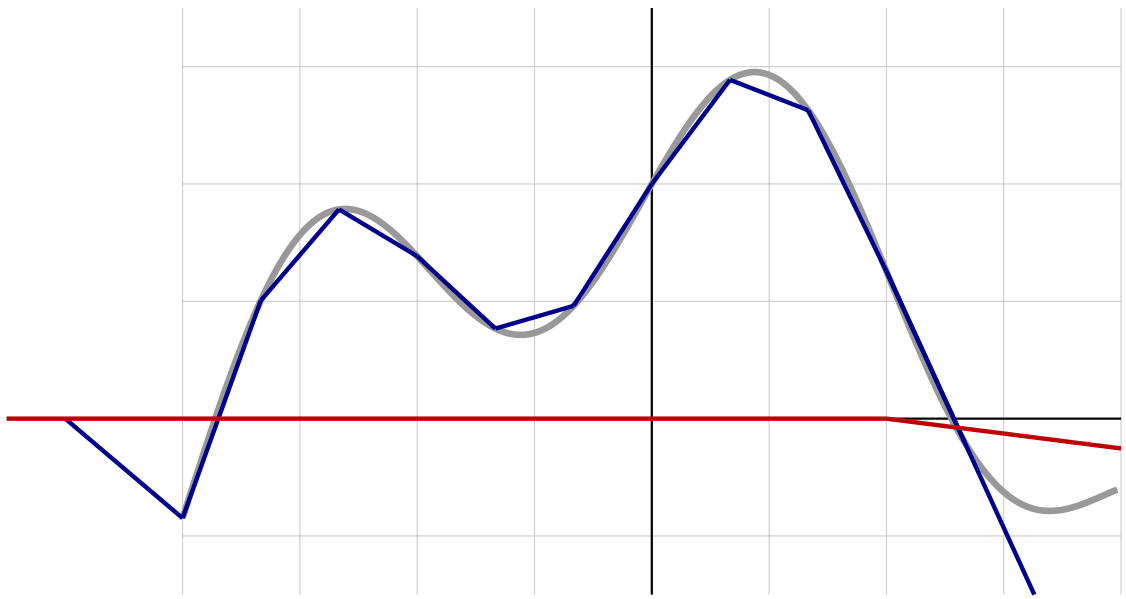
We can approximate any $\psi \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \dots$$



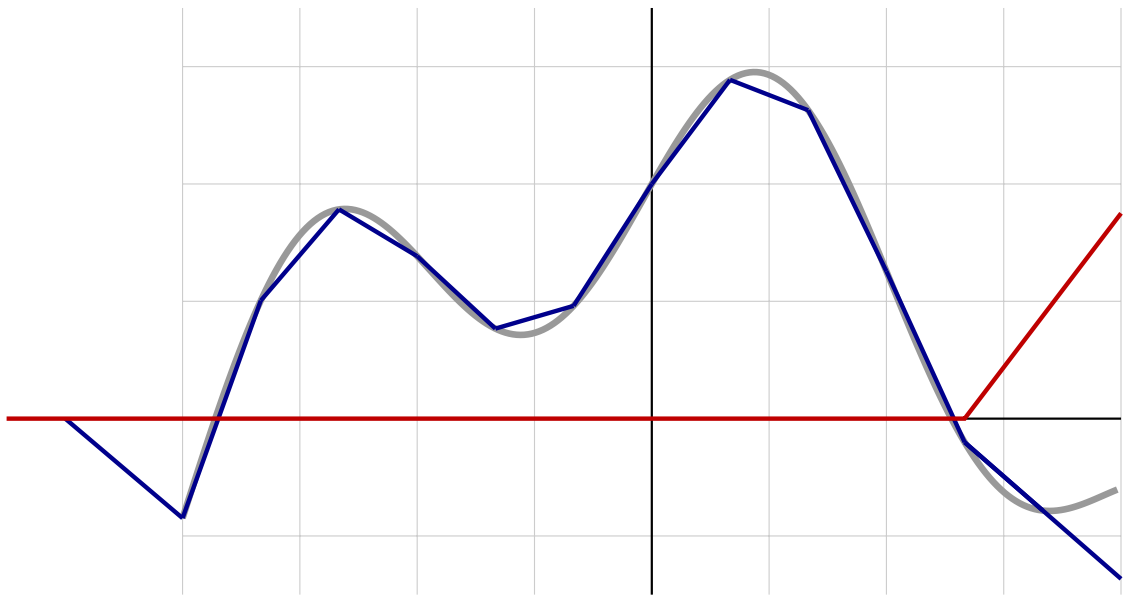
We can approximate any $\psi \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \dots$$



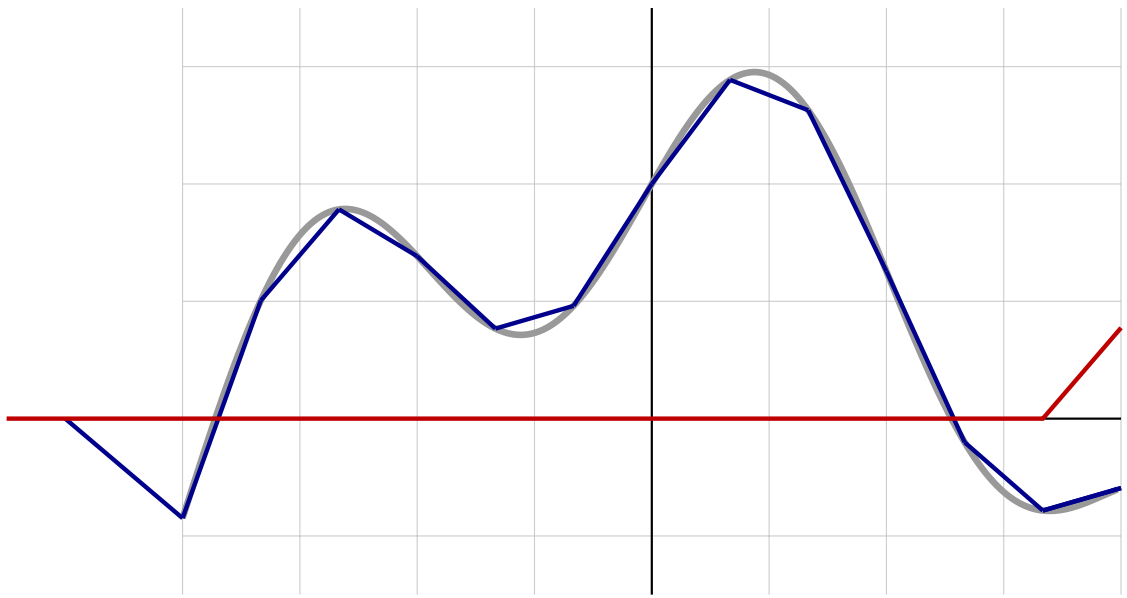
We can approximate any $\psi \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \ldots$$



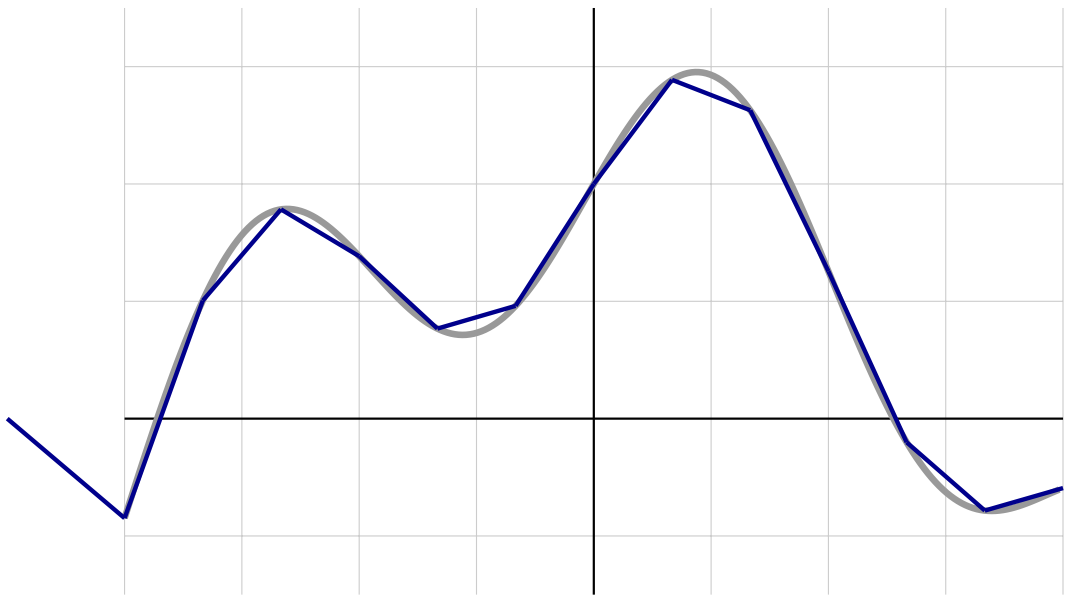
We can approximate any $\psi \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \ldots$$



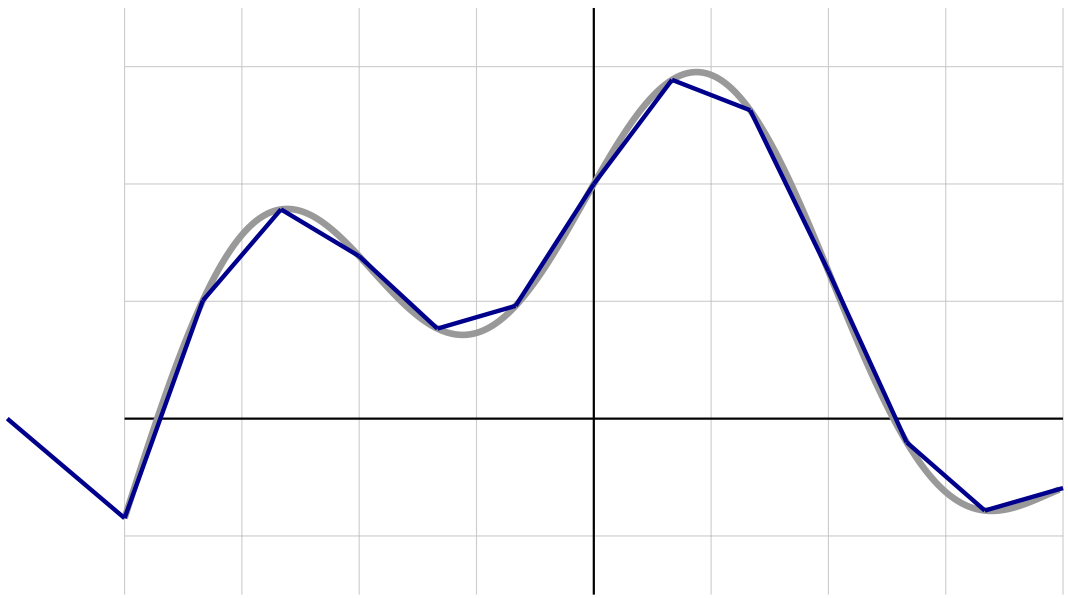
We can approximate any $\psi \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \dots$$



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

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \dots$$



This is true for other activation functions under mild assumptions.

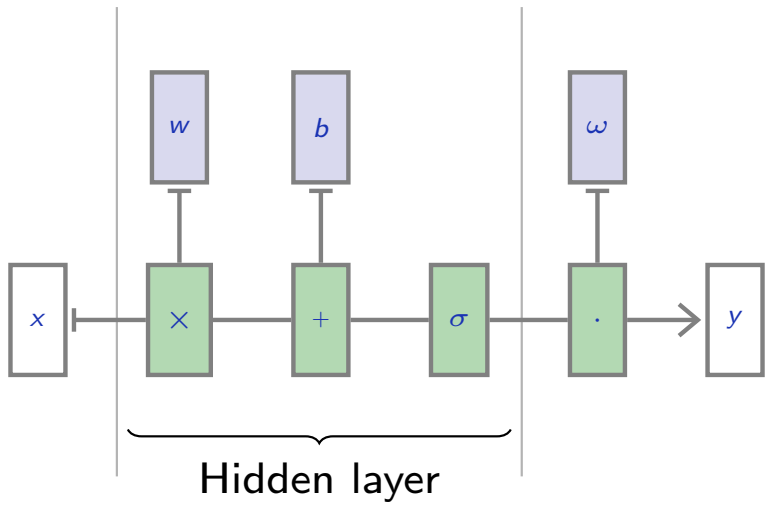
So we can approximate any continuous function

$$\psi : [0, 1]^D \to \mathbb{R}$$

with a one hidden layer perceptron

$$x \mapsto \omega \cdot \sigma(w\,x + b),$$

where $b \in \mathbb{R}^K$, $w \in \mathbb{R}^{K \times D}$, and $\omega \in \mathbb{R}^K$.



This is the **universal approximation theorem.**

A better approximation requires a larger hidden layer (larger K), and this theorem says nothing about the relation between the two.

Optimization

Gradient Descent

There is generally no *ad hoc* method. The logistic regression for instance

$$P_w(Y = 1 \mid X = x) = \sigma(w \cdot x + b), \text{ with } \sigma(x) = \frac{1}{1 + e^{-x}}$$

leads to the loss

$$\mathscr{L}(w, b) = -\sum_n \log \sigma(y_n(w \cdot x_n + b))$$

which cannot be minimized analytically.

The general minimization method used in such a case is the **gradient descent**.

Given a functional

$$\begin{aligned} f : \mathbb{R}^D &\to \mathbb{R} \\ x &\mapsto f(x_1, \ldots, x_D), \end{aligned}$$

its gradient is the mapping

$$\begin{aligned} \nabla f : \mathbb{R}^D &\to \mathbb{R}^D \\ x &\mapsto \left(\frac{\partial f}{\partial x_1}(x), \ldots, \frac{\partial f}{\partial x_D}(x) \right). \end{aligned}$$

To minimize a functional

$$\mathscr{L} : \mathbb{R}^D \to \mathbb{R}$$

the gradient descent uses local linear information to iteratively move toward a (local) minimum.

For $w_0 \in \mathbb{R}^D$, consider an approximation of $\mathscr{L}$ around w_0

$$\tilde{\mathscr{L}}_{w_0}(w) = \mathscr{L}(w_0) + \nabla \mathscr{L}(w_0)^T (w - w_0) + \frac{1}{2\eta} \|w - w_0\|^2.$$

Note that the chosen quadratic term does not depend on $\mathscr{L}$.

We have

$$\nabla \tilde{\mathscr{L}}_{w_0}(w) = \nabla \mathscr{L}(w_0) + \frac{1}{\eta}(w - w_0),$$

which leads to

$$\operatorname*{argmin}_w \tilde{\mathscr{L}}_{w_0}(w) = w_0 - \eta \nabla \mathscr{L}(w_0).$$

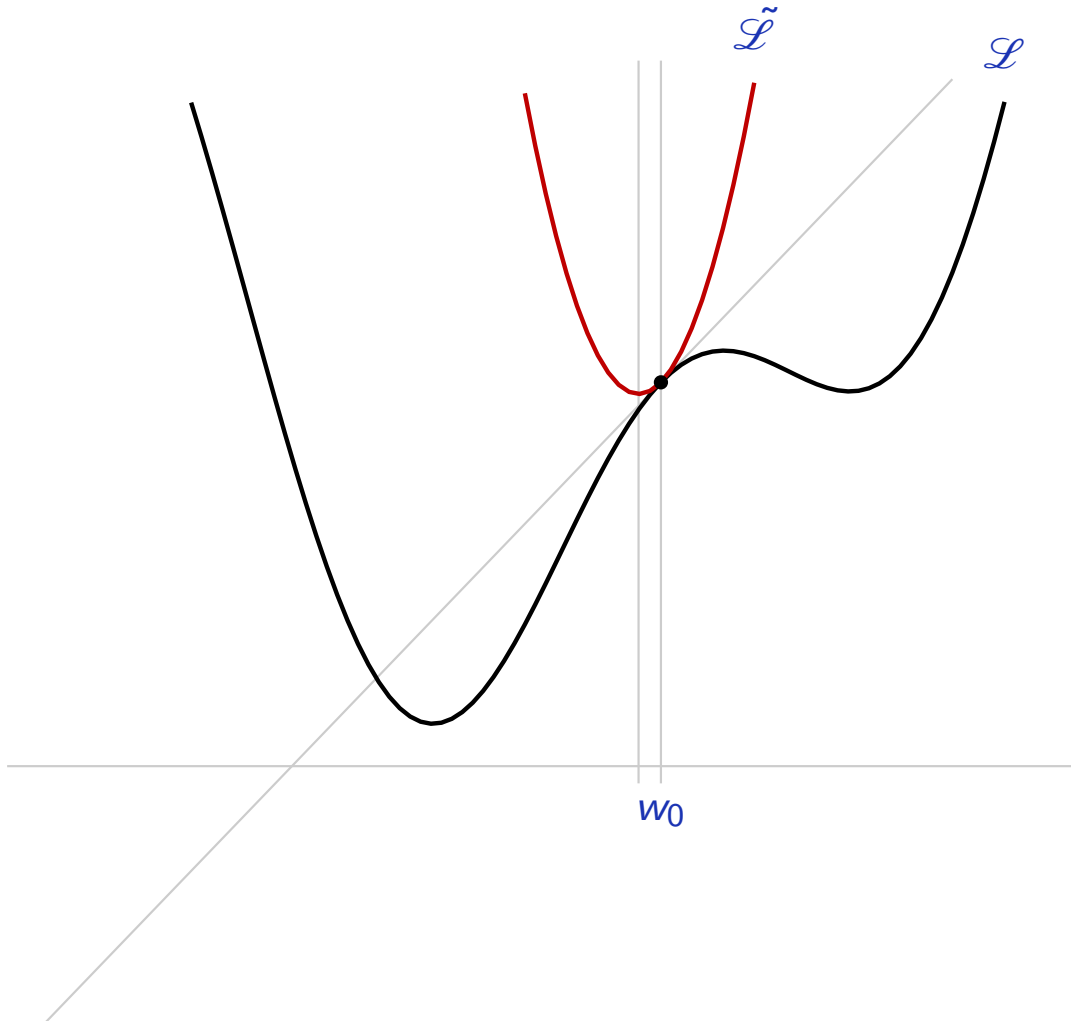
The resulting iterative rule, which goes to the minimum of the approximation at the current location, takes the form:

$$w_{t+1} = w_t - \eta \nabla \mathscr{L}(w_t),$$

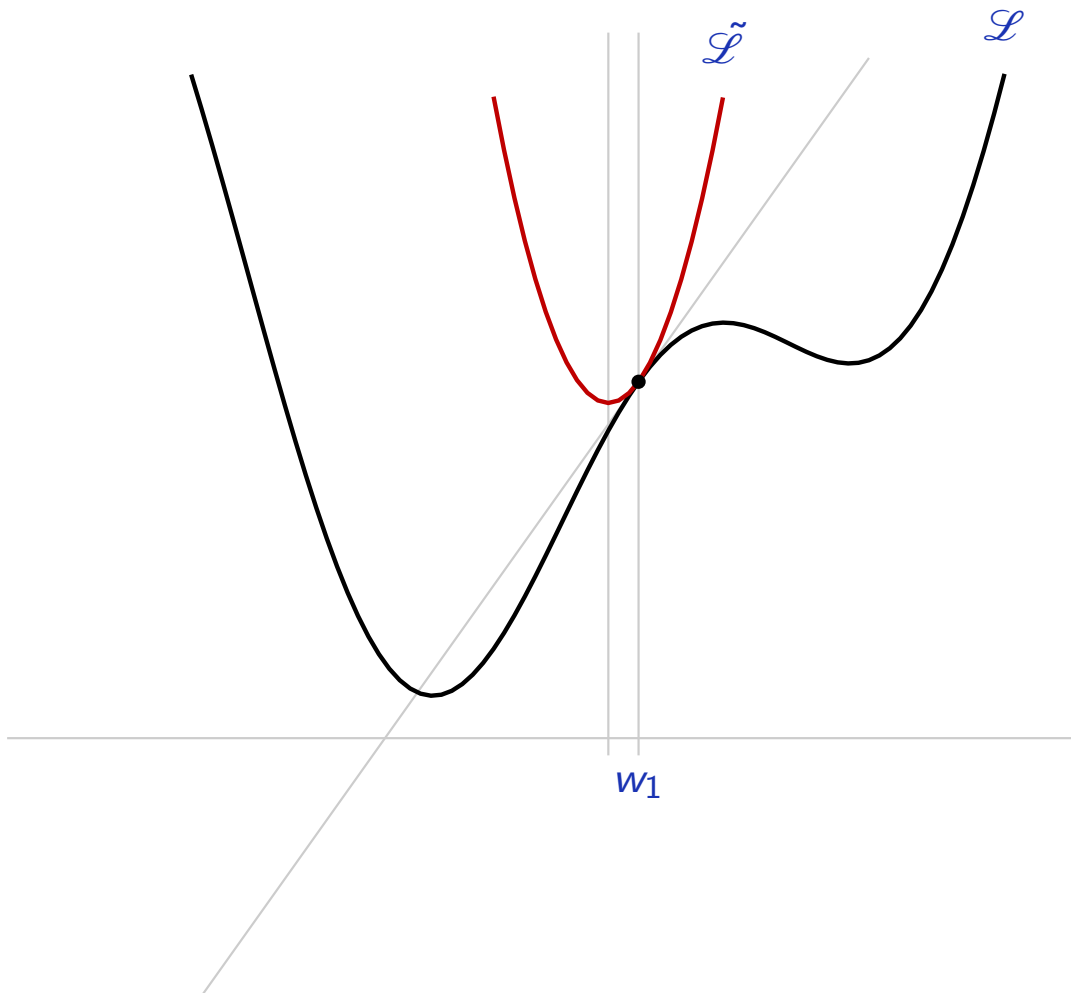
which corresponds intuitively to "following the steepest descent".

This [most of the time] eventually ends up in a **local** minimum, and the choices of w_0 and η are important.

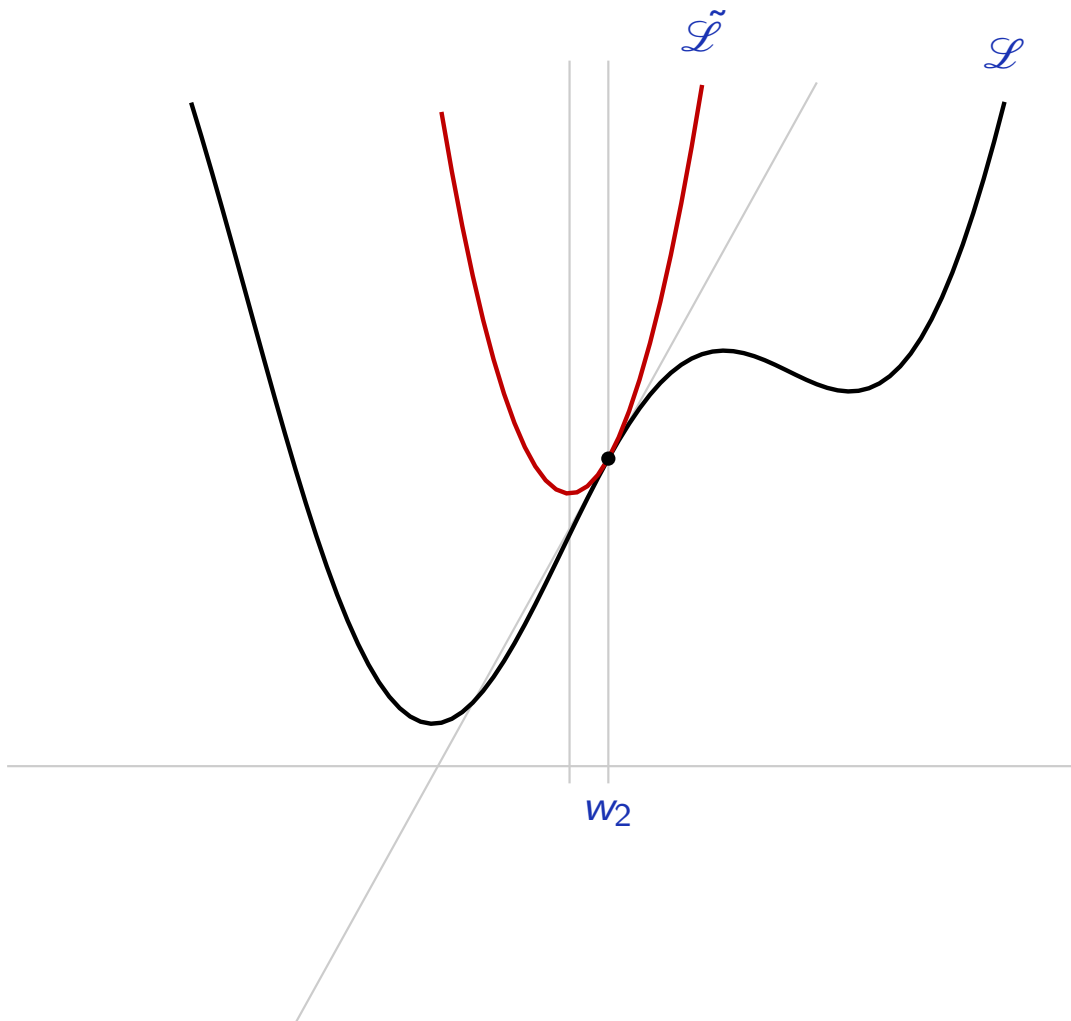
$\eta = 0.125$



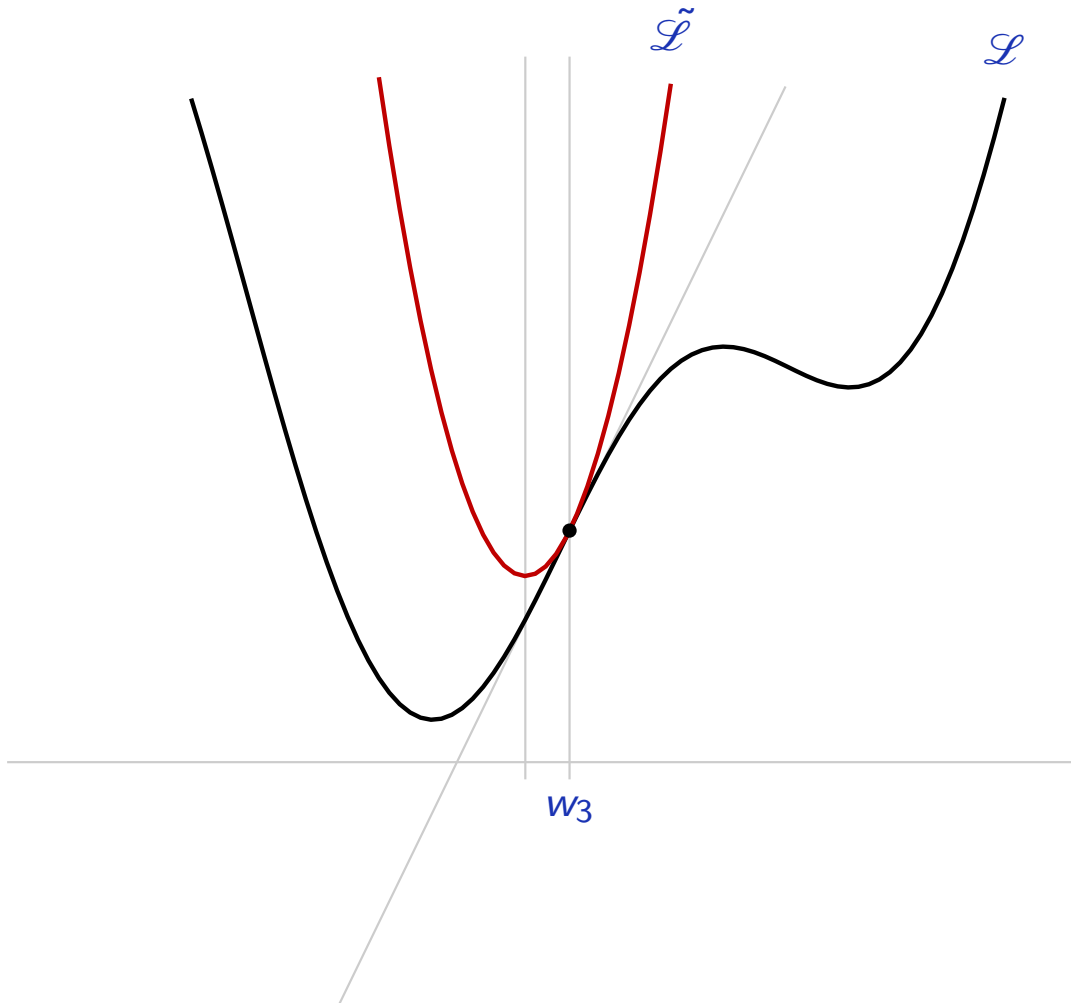
$\eta = 0.125$



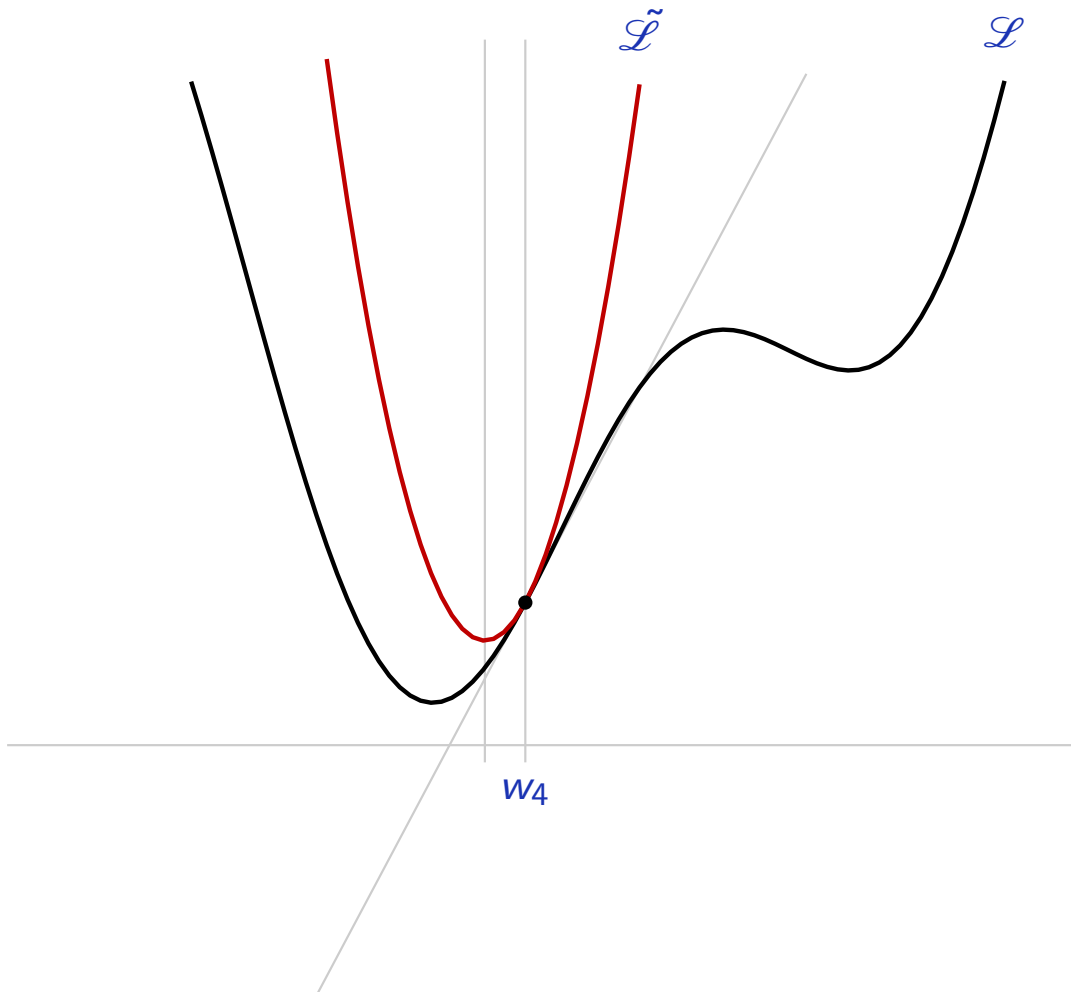
$\eta = 0.125$



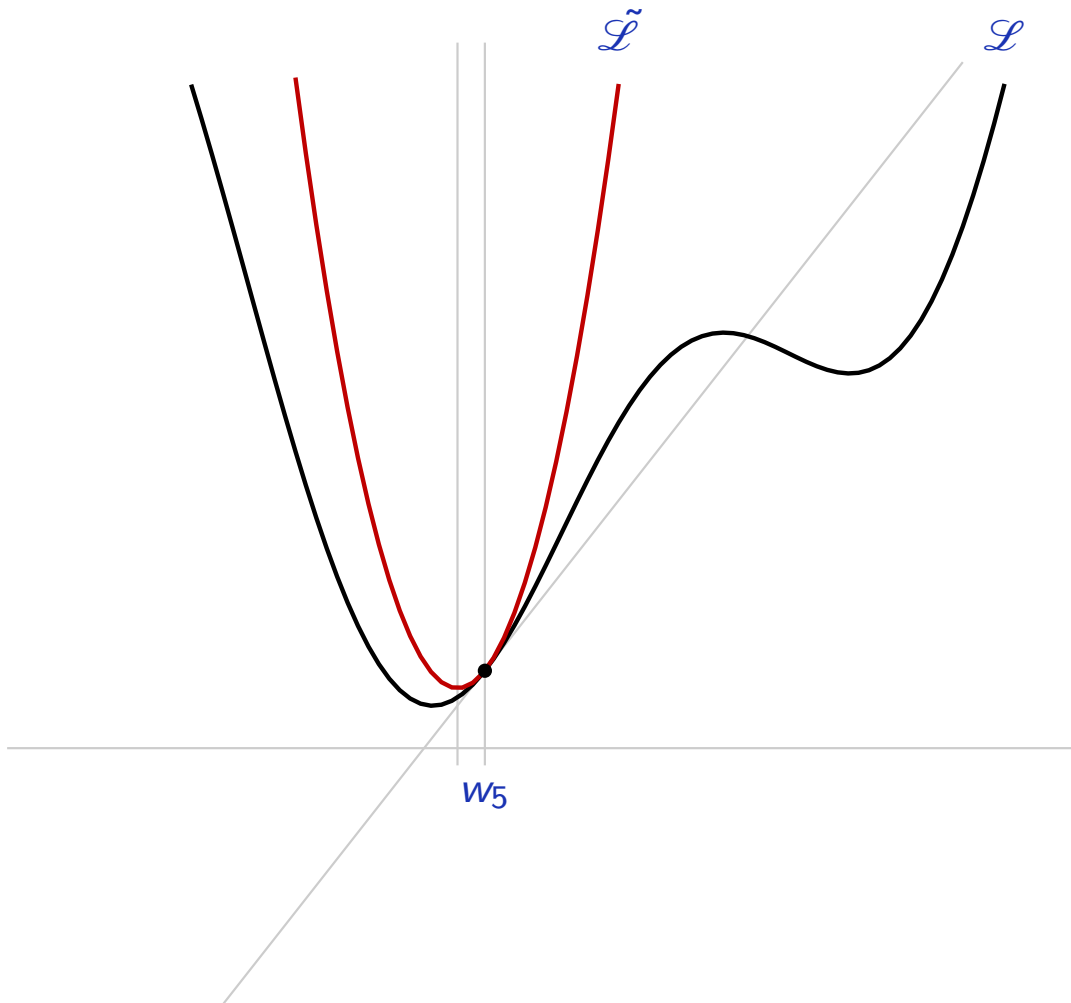
$\eta = 0.125$



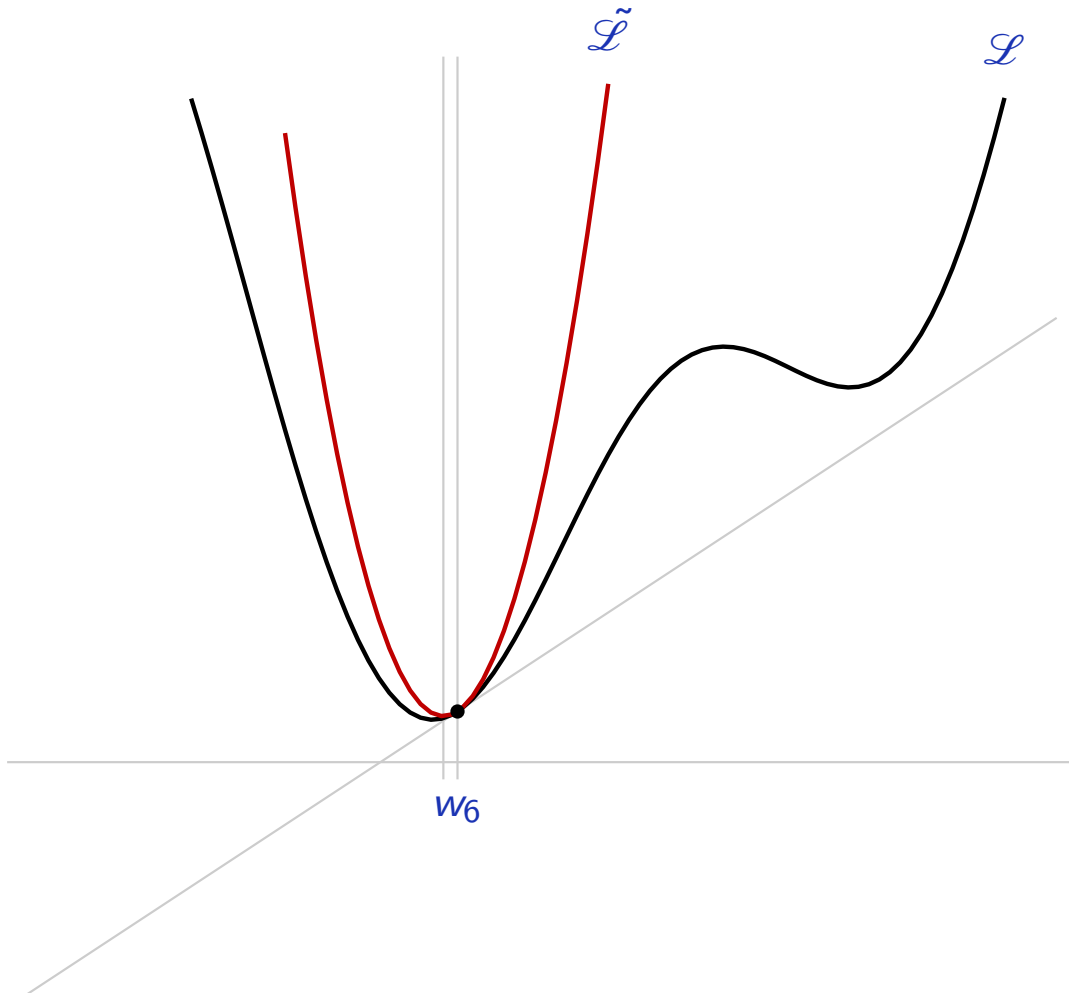
$\eta = 0.125$



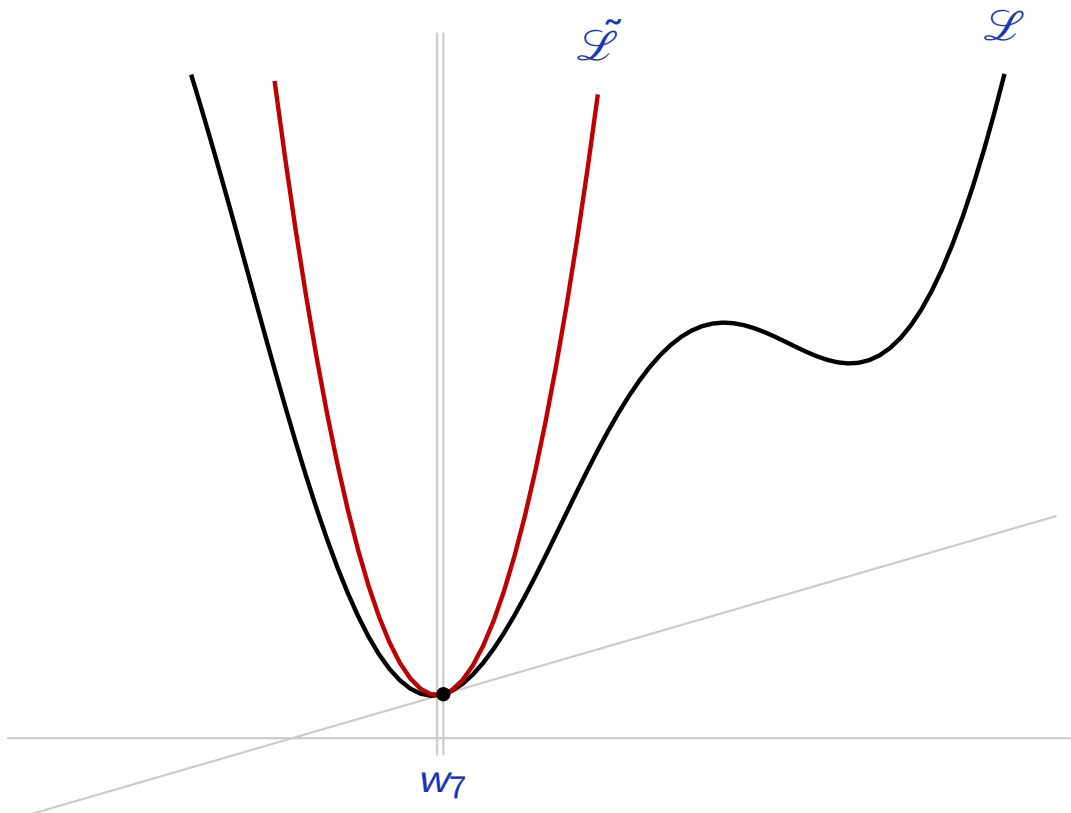
$\eta = 0.125$



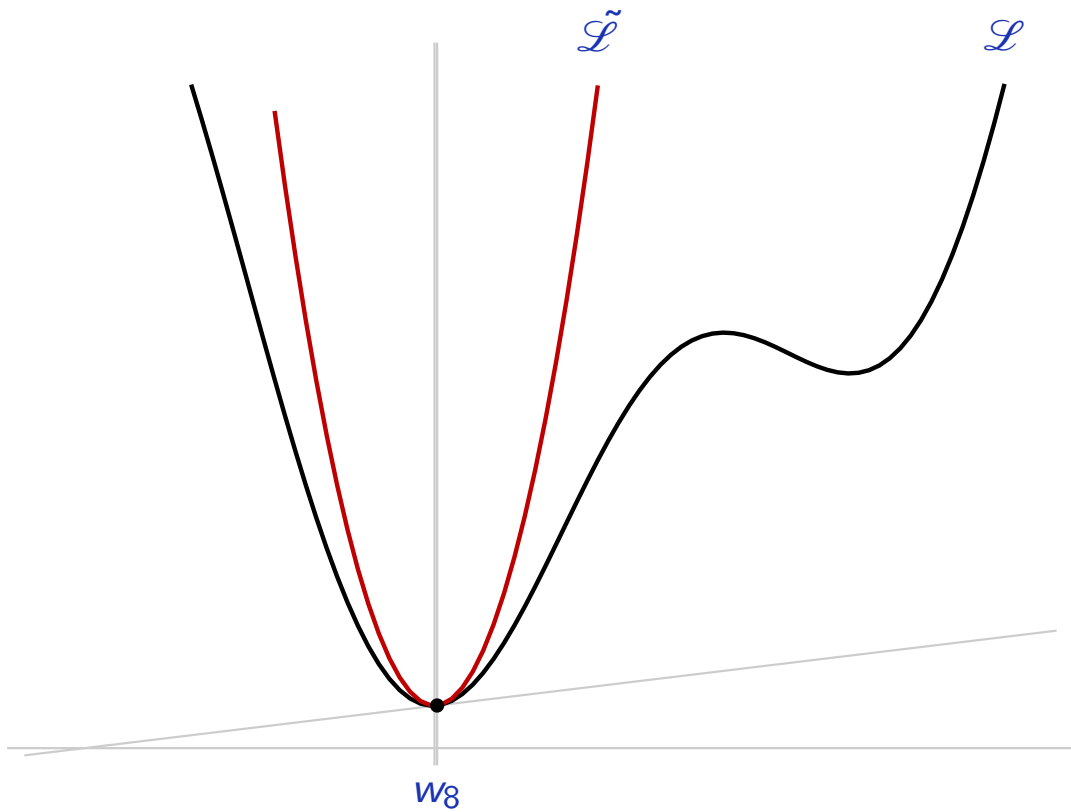
$\eta = 0.125$



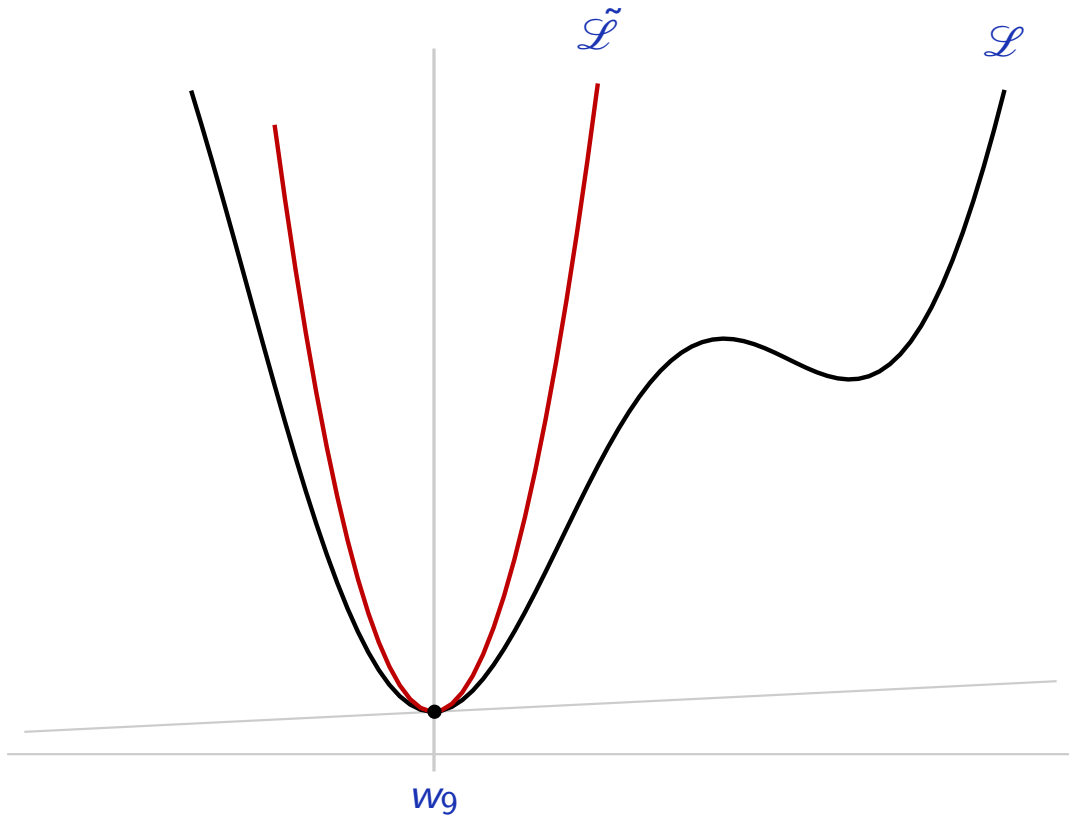
$\eta = 0.125$



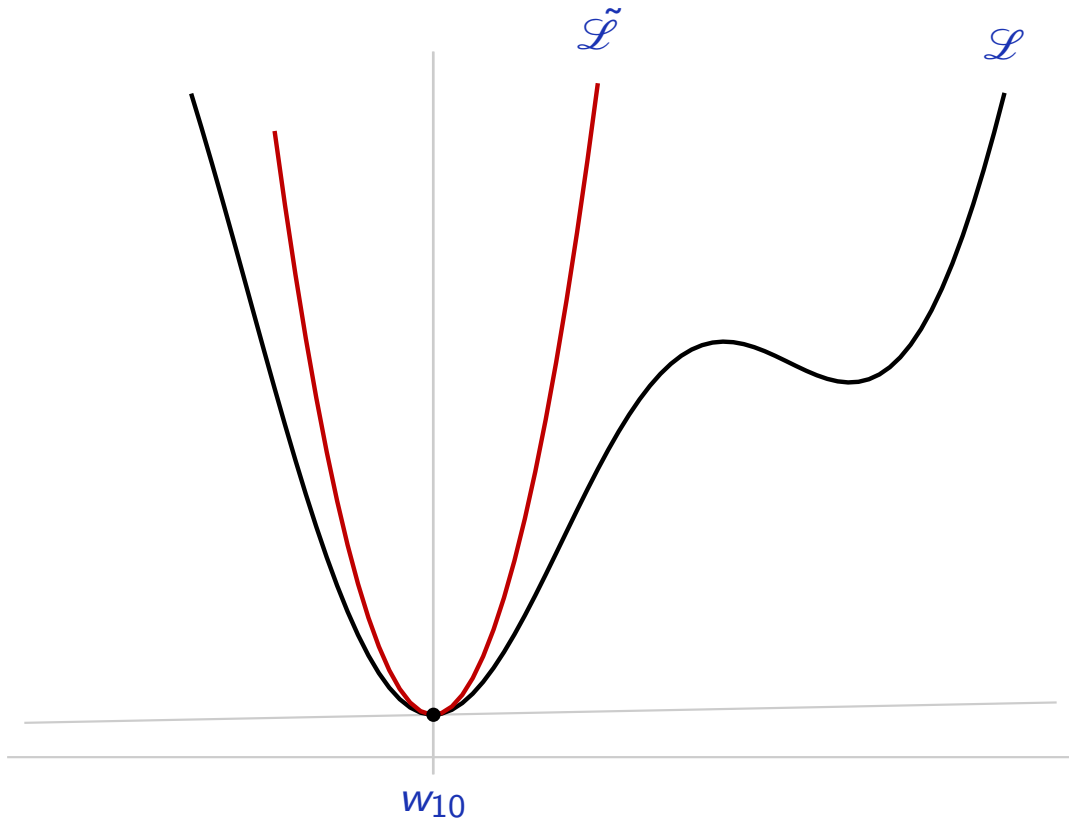
$\eta = 0.125$



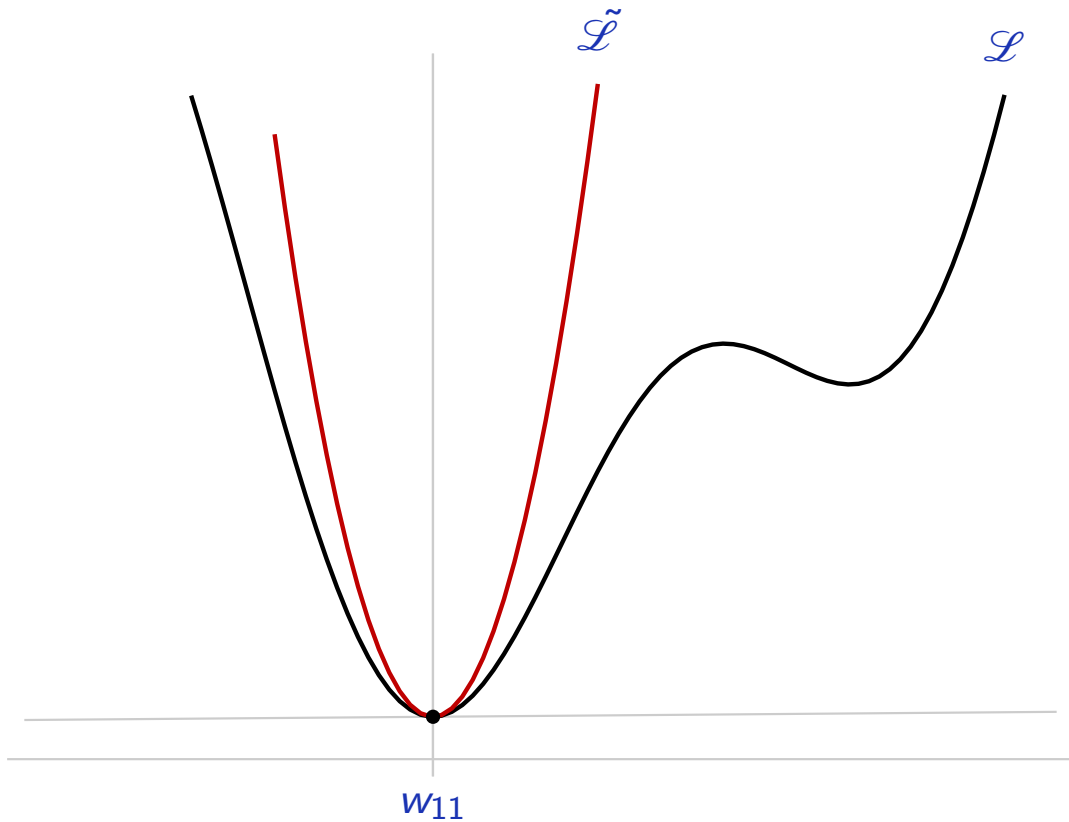
$\eta = 0.125$

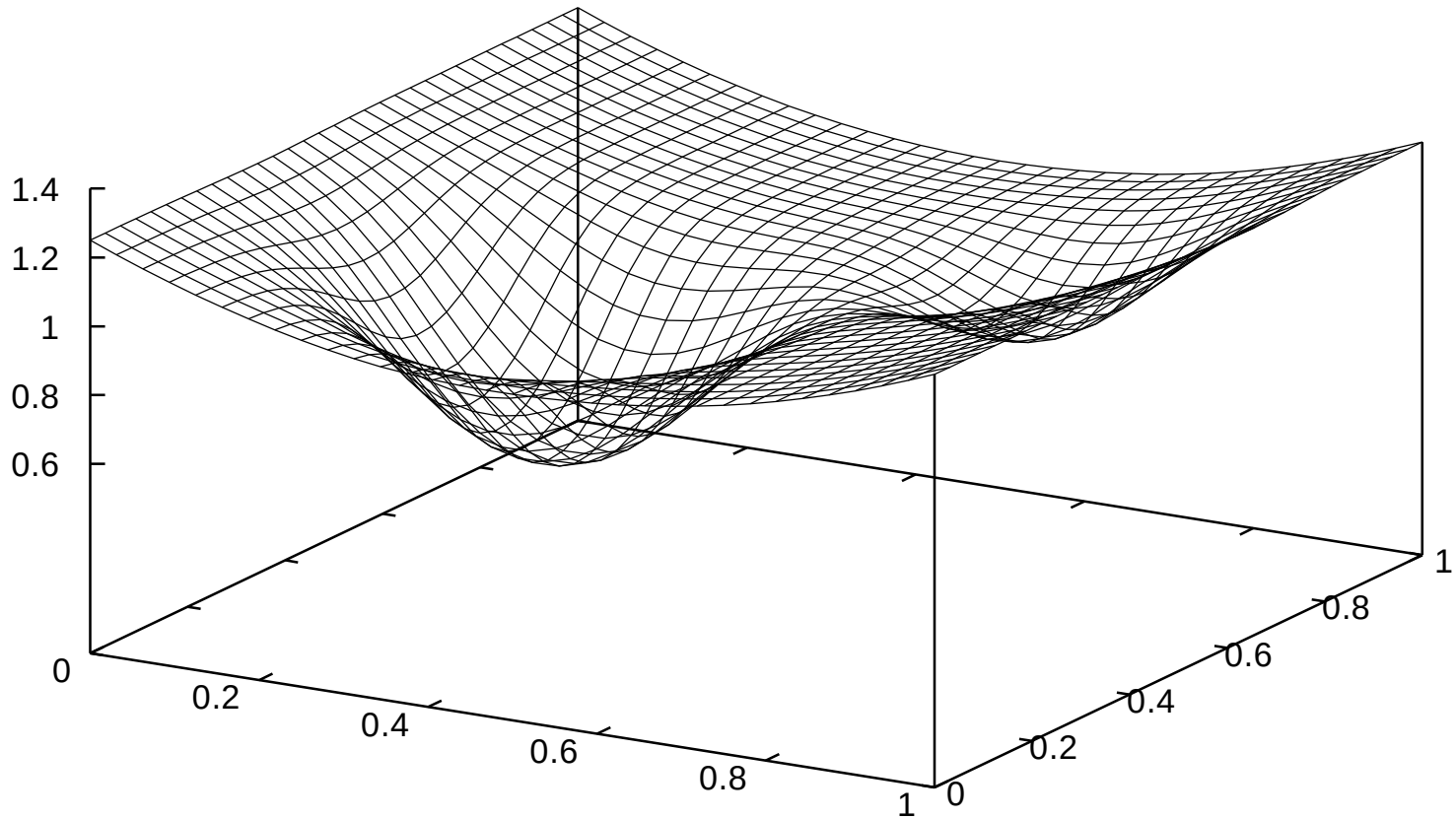


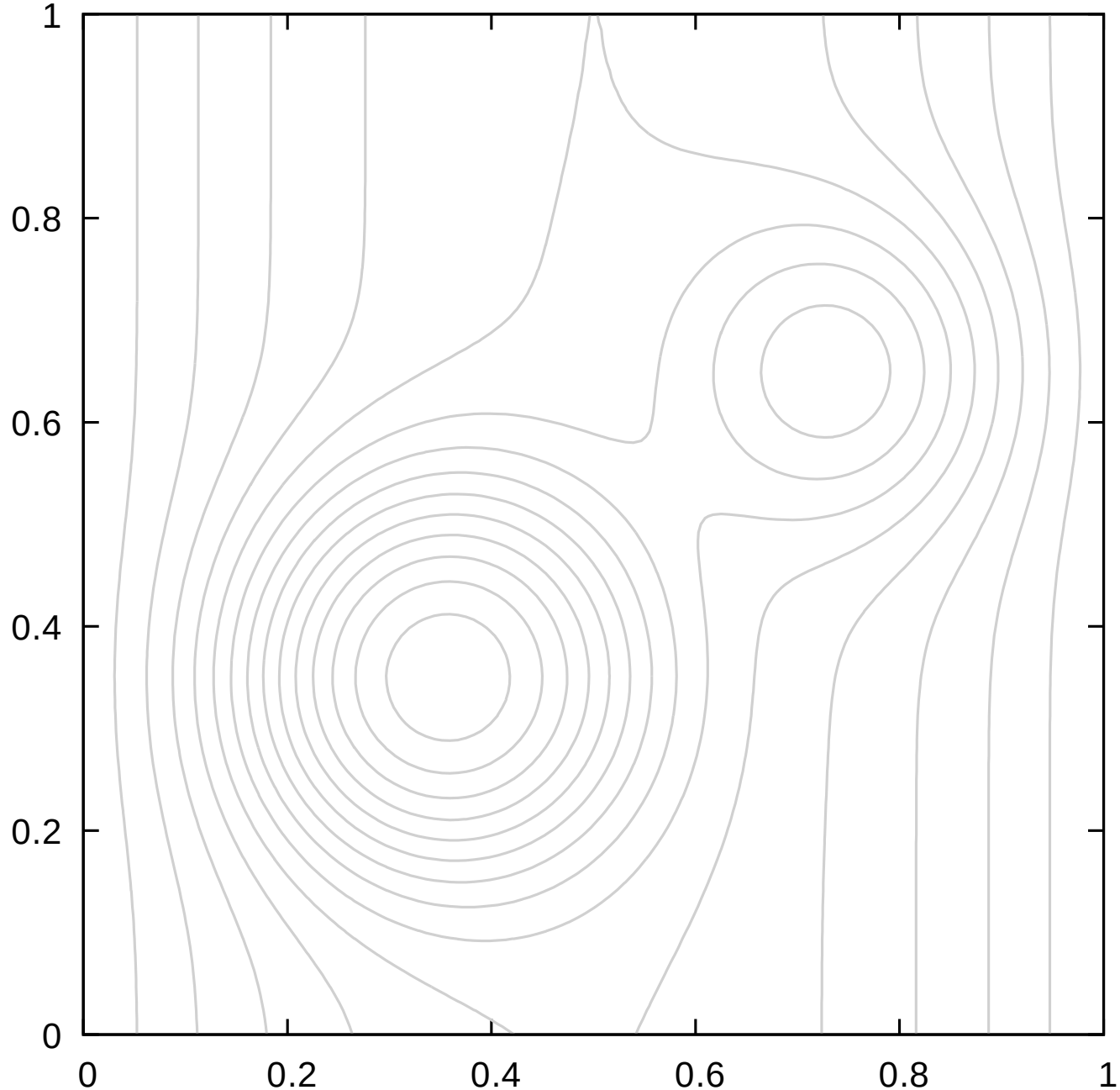
$\eta = 0.125$

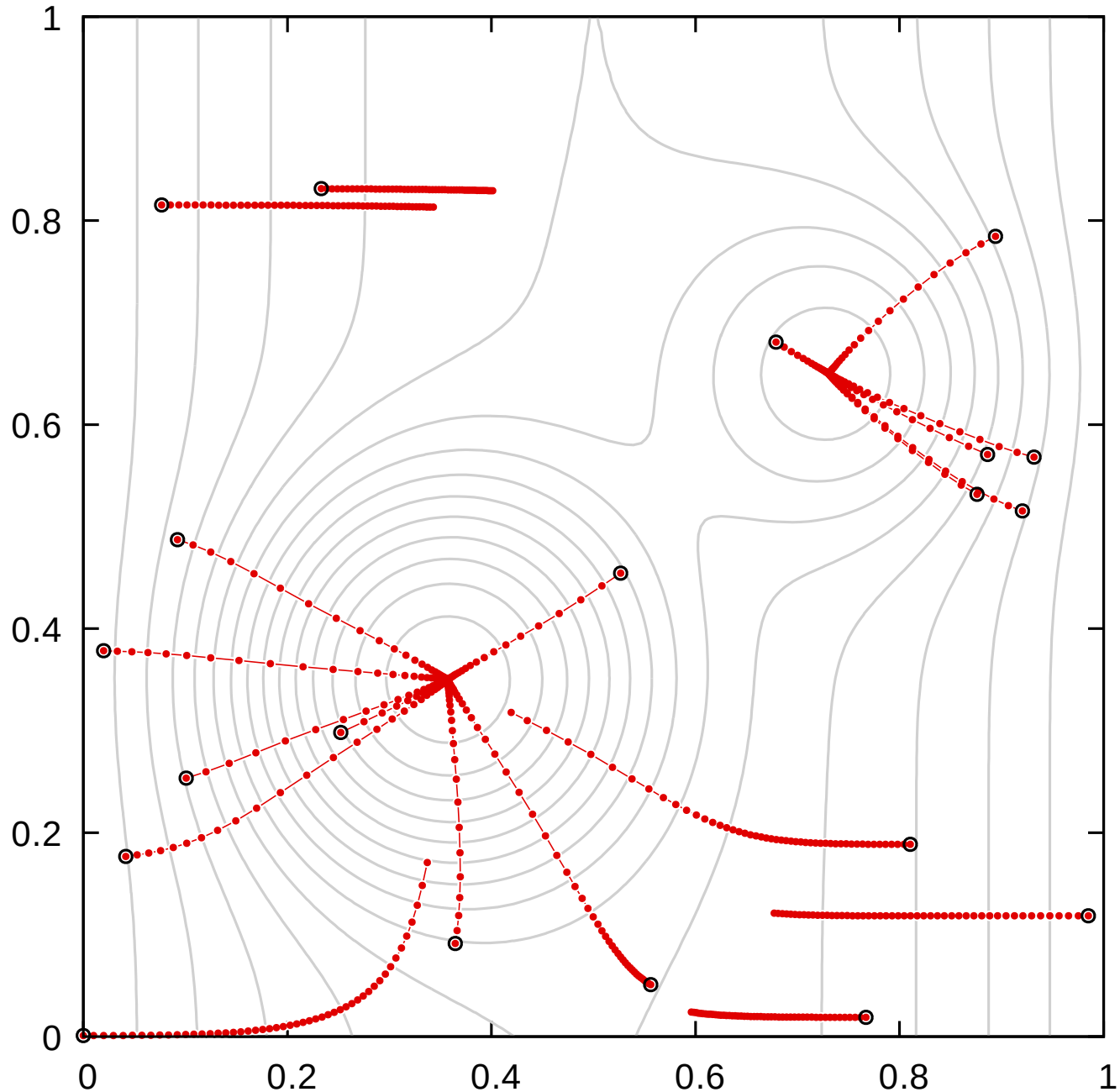


$\eta = 0.125$




1.4
1.2
1
0.8
0.6
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1


1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1


1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1

We want to train an MLP by minimizing a loss over the training set

$$\mathscr{L}(w, b) = \sum_n \ell(f(x_n; w, b), y_n).$$

To use gradient descent, we need the expression of the gradient of the per-sample loss $\ell_n = \ell(f(x_n; w, b), y_n)$ with respect to the parameters, *e.g.*

$$\frac{\partial \ell_n}{\partial w_{i,j}^{(l)}} \quad \text{and} \quad \frac{\partial \ell_n}{\partial b_i^{(l)}}.$$

The core principle of the back-propagation algorithm is the "chain rule" from differential calculus:

$$(g \circ f)' = (g' \circ f) f'.$$

The linear approximation of a composition of mappings is the product of their individual linear approximations.

This generalizes to longer compositions and higher dimensions

$$J_{f_N \circ f_{N-1} \circ \cdots \circ f_1}(x) = J_{F_N}(f_{N-1}(\dots(x))) \dots J_{f_3}(f_2(f_1(x))) J_{f_2}(f_1(x)) J_{f_1}(x)$$

where $J_f(x)$ is the Jacobian of f at x, that is the matrix of the linear approximation of f in the neighborhood of x.

Forward pass

Compute the activations.

$$x^{(0)} = x, \quad \forall l = 1, \ldots, L, \begin{cases} s^{(l)} = w^{(l)} x^{(l-1)} + b^{(l)} \\ x^{(l)} = \sigma\left(s^{(l)}\right) \end{cases}$$

Backward pass

Compute the derivatives of the loss wrt the activations.

$$\begin{cases} \left[\frac{\partial \ell}{\partial x^{(L)}}\right] \text{ from the definition of } \ell \\ \text{if } l < L, \left[\frac{\partial \ell}{\partial x^{(l)}}\right] = \left(w^{(l+1)}\right)^T \left[\frac{\partial \ell}{\partial s^{(l+1)}}\right] \end{cases} \qquad \left[\frac{\partial \ell}{\partial s^{(l)}}\right] = \left[\frac{\partial \ell}{\partial x^{(l)}}\right] \odot \sigma'\left(s^{(l)}\right)$$

Compute the derivatives of the loss wrt the parameters.

$$\left[\!\left[\frac{\partial \ell}{\partial w^{(l)}}\right]\!\right] = \left[\frac{\partial \ell}{\partial s^{(l)}}\right] \left(x^{(l-1)}\right)^T \qquad \left[\frac{\partial \ell}{\partial b^{(l)}}\right] = \left[\frac{\partial \ell}{\partial s^{(l)}}\right].$$

Gradient step

Update the parameters.

$$w^{(l)} \leftarrow w^{(l)} - \eta \left[\!\left[\frac{\partial \ell}{\partial w^{(l)}}\right]\!\right] \qquad b^{(l)} \leftarrow b^{(l)} - \eta \left[\frac{\partial \ell}{\partial b^{(l)}}\right]$$

Regarding computation, since the costly operation for the forward pass is

$$s^{(l)} = w^{(l)} x^{(l-1)} + b^{(l)}$$

and for the backward

$$\left[\frac{\partial \ell}{\partial x^{(l)}}\right] = \left(w^{(l+1)}\right)^T \left[\frac{\partial \ell}{\partial s^{(l+1)}}\right]$$

and

$$\left[\!\left[\frac{\partial \ell}{\partial w^{(l)}}\right]\!\right] = \left[\frac{\partial \ell}{\partial s^{(l)}}\right] \left(x^{(l-1)}\right)^T,$$

the rule of thumb is that the backward pass is twice more expensive than the forward one.

Mini-batch Gradient Descent

While it makes sense in principle to compute the gradient exactly, in practice:

- It takes time to compute (more exactly **all our time!**).
- It is an empirical estimation of an hidden quantity, and any partial sum is also an unbiased estimate, although of greater variance.
- It is computed incrementally

$$\nabla \mathscr{L}(w_t) = \sum_{n=1}^{N} \nabla \ell_n(w_t),$$

and when we compute ℓ_n, we have already computed $\ell_1, \ldots, \ell_{n-1}$, and we could have a better estimate of w^* than w_t.

The **stochastic gradient descent** consists of updating the parameters w_t after every sample

$$w_{t+1} = w_t - \eta \nabla \ell_{n(t)}(w_t).$$

However this does not benefit from the speed-up of batch-processing.

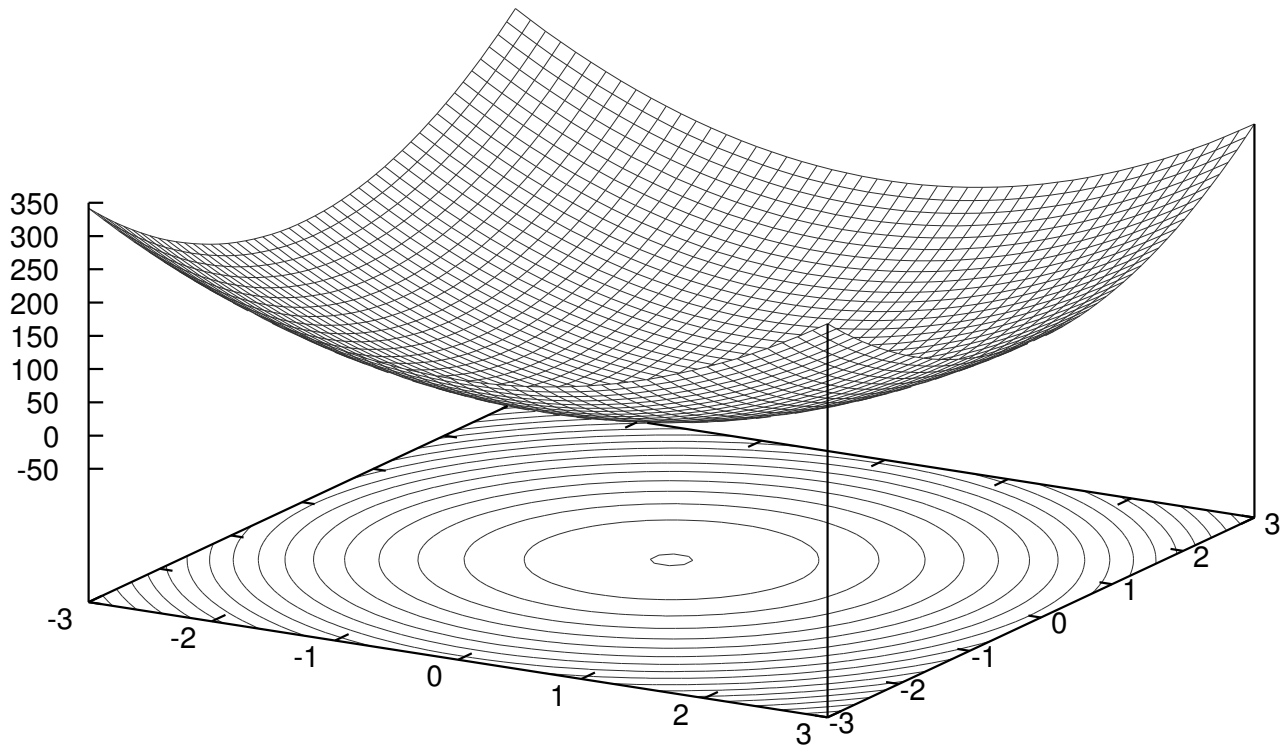
The **mini-batch stochastic gradient descent** is the standard procedure for deep learning. It consists of visiting the samples in "mini-batches", each of a few tens of samples, and updating the parameters each time.

$$w_{t+1} = w_t - \eta \sum_{b=1}^{B} \nabla \ell_{n(t,b)}(w_t).$$

The order $n(t, b)$ to visit the samples can either be sequential, or uniform sampling, usually without replacement.

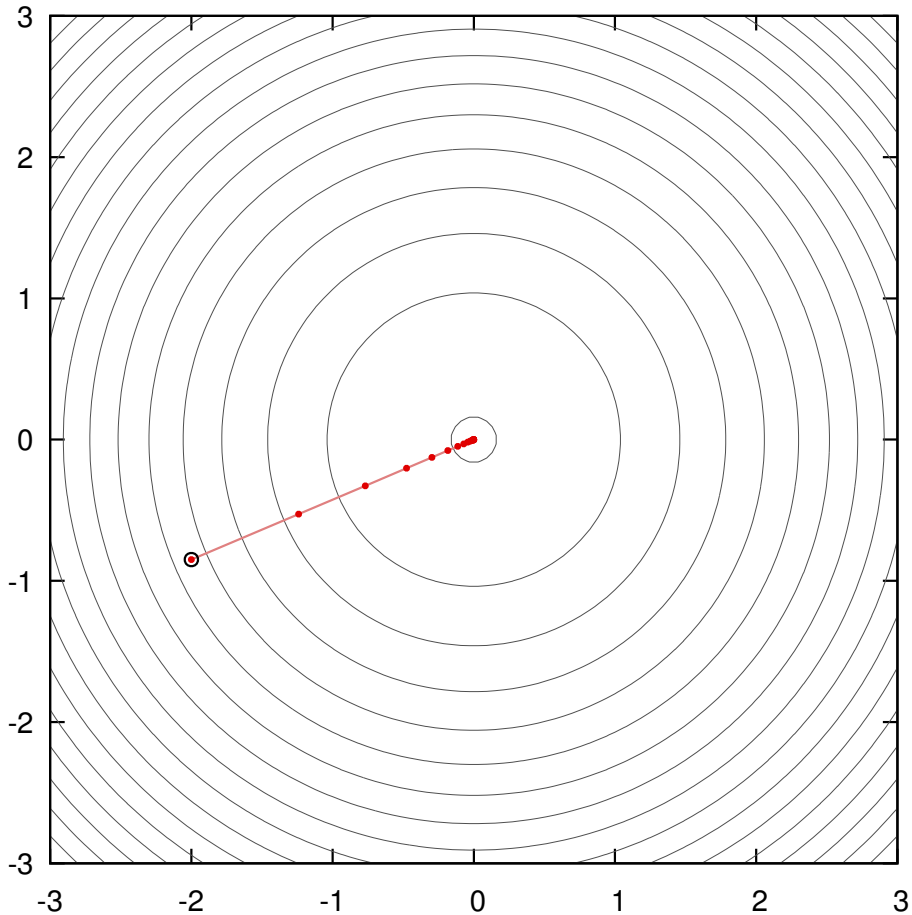
The stochastic behavior of this procedure helps evade local minima.

The gradient descent method makes a strong assumption about the magnitude of the "local curvature" to fix the step size, and about its isotropy so that the same step size makes sense in all directions.



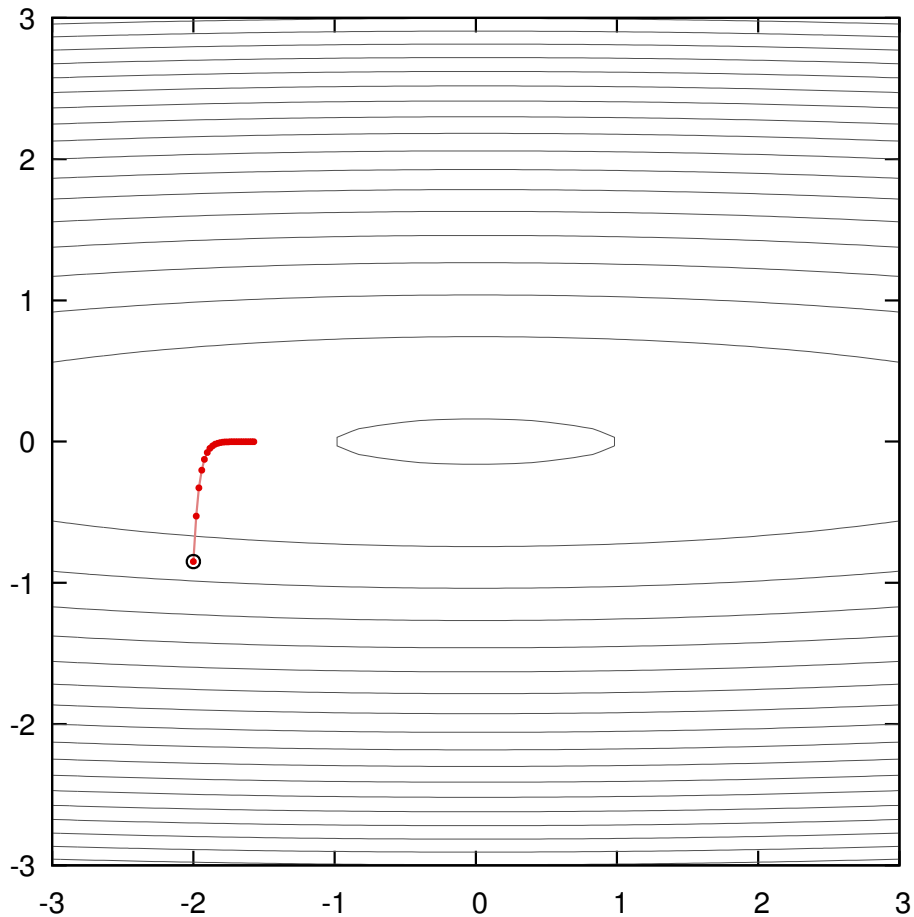
The gradient descent method makes a strong assumption about the magnitude of the "local curvature" to fix the step size, and about its isotropy so that the same step size makes sense in all directions.

$$\eta = 1.0e-2$$



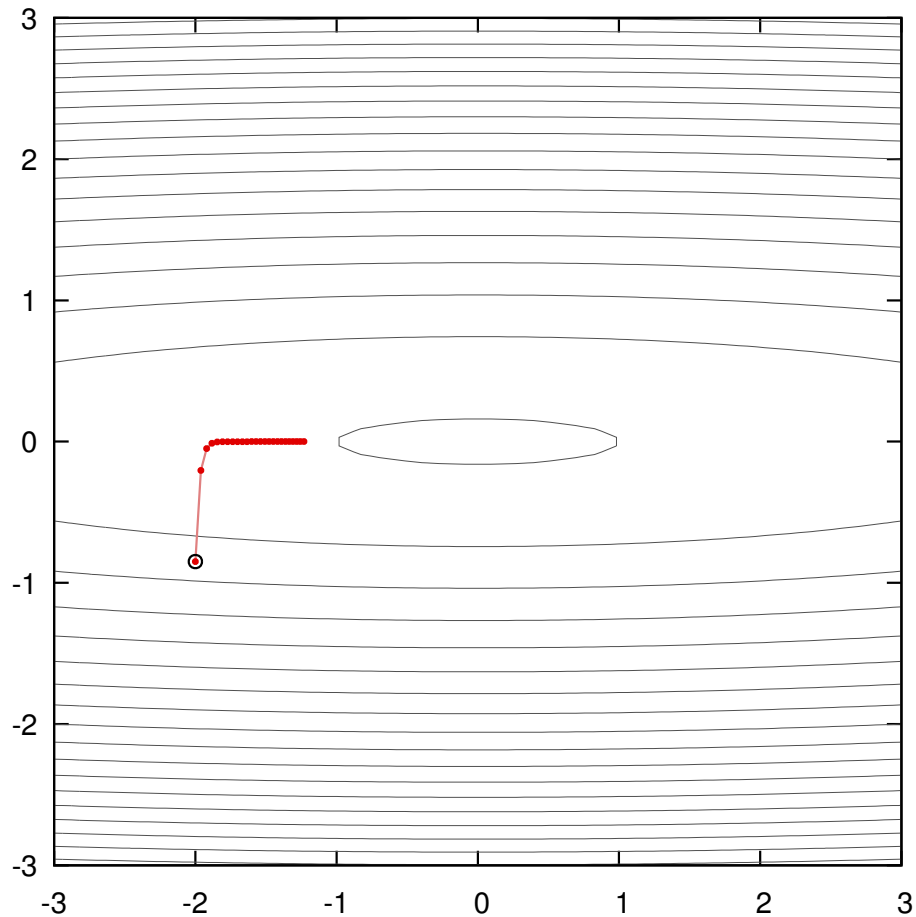
The gradient descent method makes a strong assumption about the magnitude of the "local curvature" to fix the step size, and about its isotropy so that the same step size makes sense in all directions.

$$\eta = 1.0e-2$$



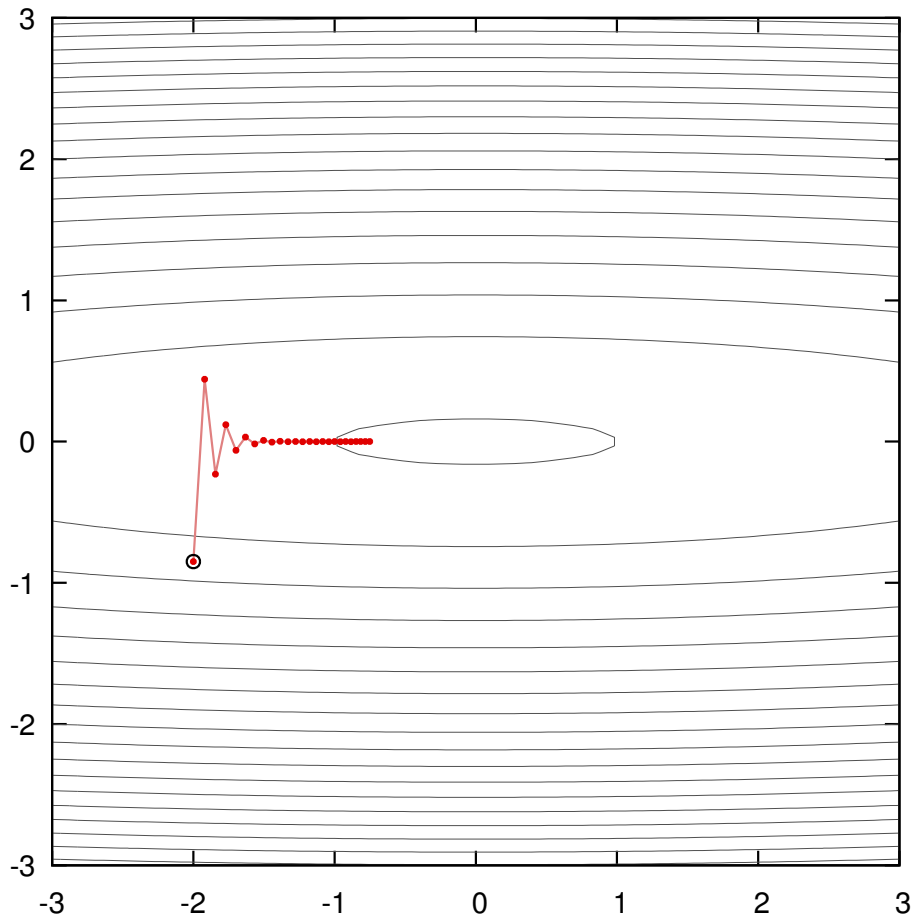
The gradient descent method makes a strong assumption about the magnitude of the "local curvature" to fix the step size, and about its isotropy so that the same step size makes sense in all directions.

$$\eta = 2.0e-2$$



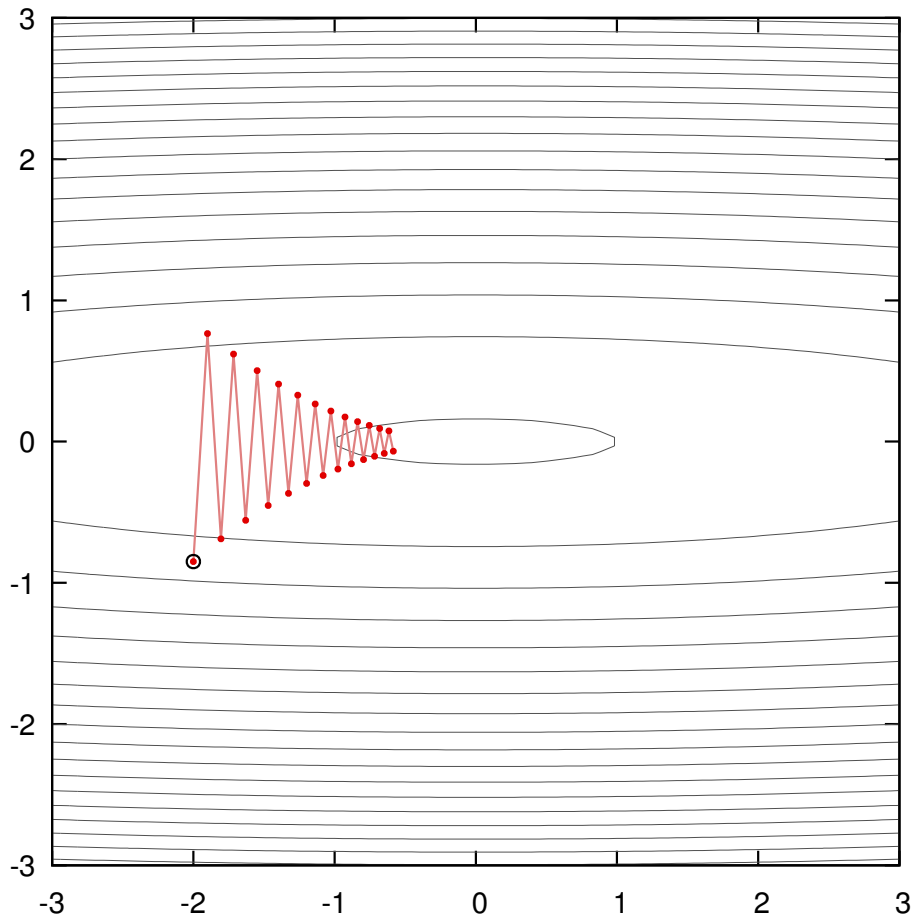
The gradient descent method makes a strong assumption about the magnitude of the "local curvature" to fix the step size, and about its isotropy so that the same step size makes sense in all directions.

$$\eta = 4.0e-2$$



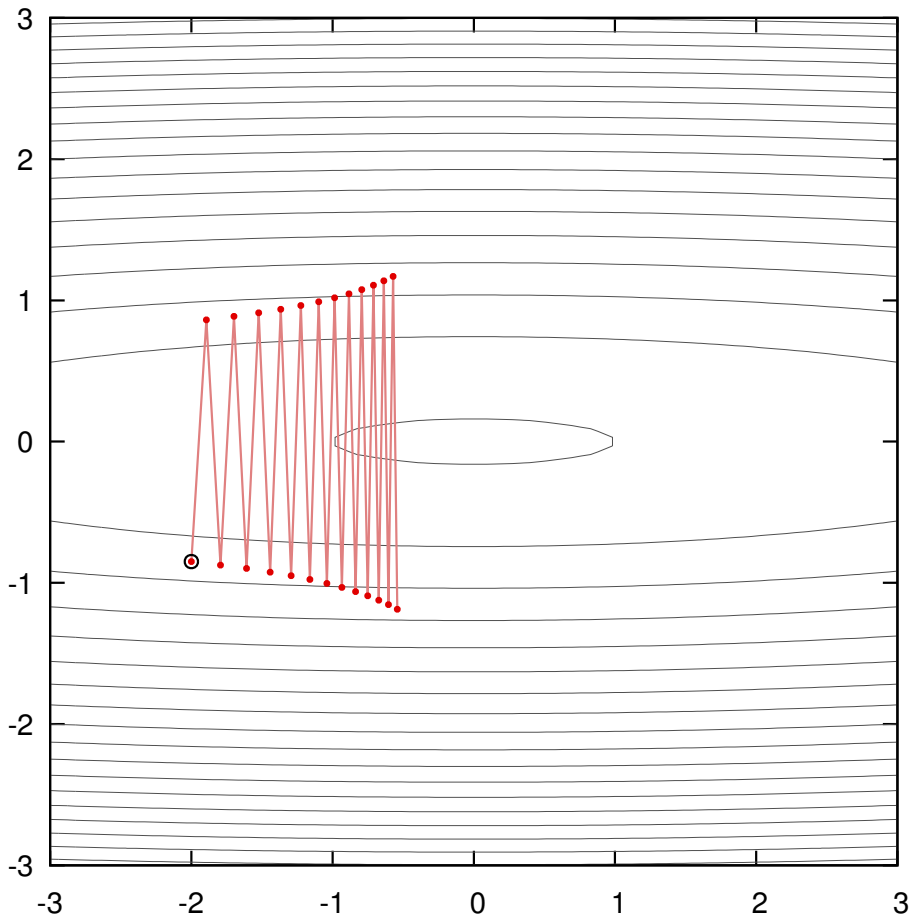
The gradient descent method makes a strong assumption about the magnitude of the "local curvature" to fix the step size, and about its isotropy so that the same step size makes sense in all directions.

$$\eta = 5.0e-2$$



The gradient descent method makes a strong assumption about the magnitude of the "local curvature" to fix the step size, and about its isotropy so that the same step size makes sense in all directions.

$$\eta = 5.3e-2$$



Some optimization methods leverage higher-order moments, in particular second order to use a more accurate local model of the functional to optimize.

However for a fixed computational budget, the complexity of these methods reduces the total number of iterations, and the eventual optimization is worse.

Deep-learning generally relies on a smarter use of the gradient, using statistics over its past values to make a "smarter step" with the current one.

The "vanilla" mini-batch stochastic gradient descent (SGD) consists of

$$w_{t+1} = w_t - \eta g_t,$$

where

$$g_t = \sum_{b=1}^{B} \nabla \ell_{n(t,b)}(w_t)$$

is the gradient summed over a mini-batch.

The first improvement is the use of a "momentum" to add inertia in the choice of the step direction

$$u_t = \gamma u_{t-1} + \eta g_t$$
$$w_{t+1} = w_t - u_t.$$

(Rumelhart et al., 1986)

With $\gamma = 0$, this is the same as vanilla SGD.

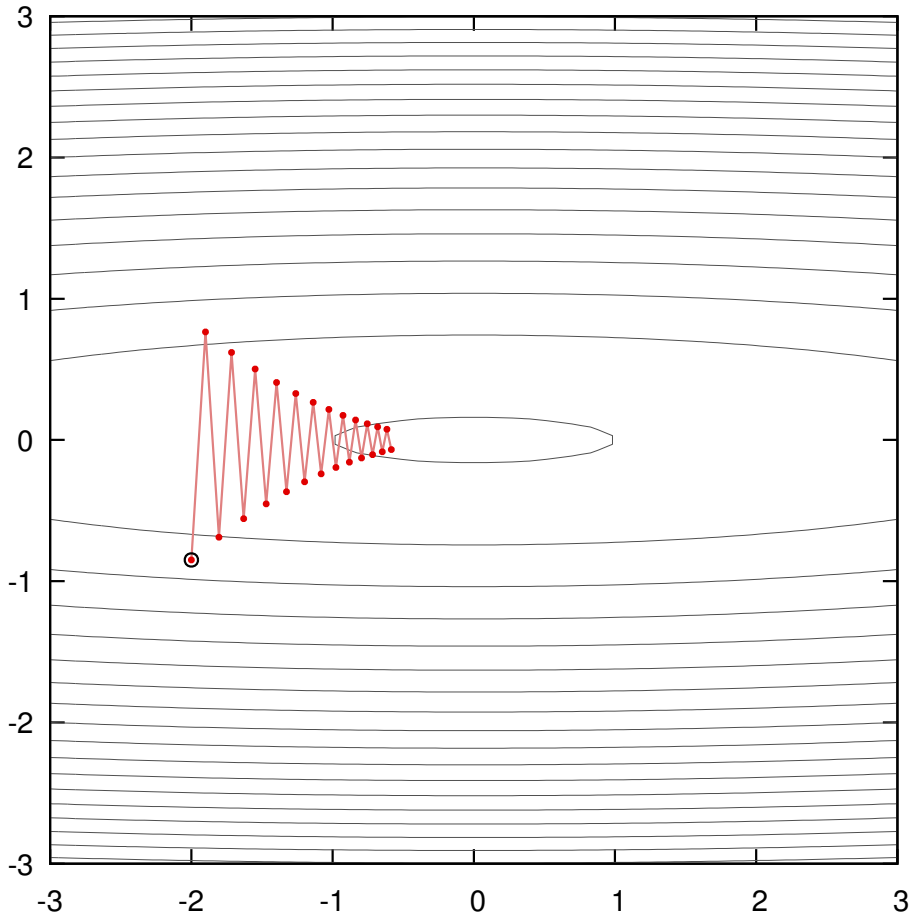
With $\gamma > 0$, this update has three nice properties:

- it can "go through" local barriers,
- it accelerates if the gradient does not change much:

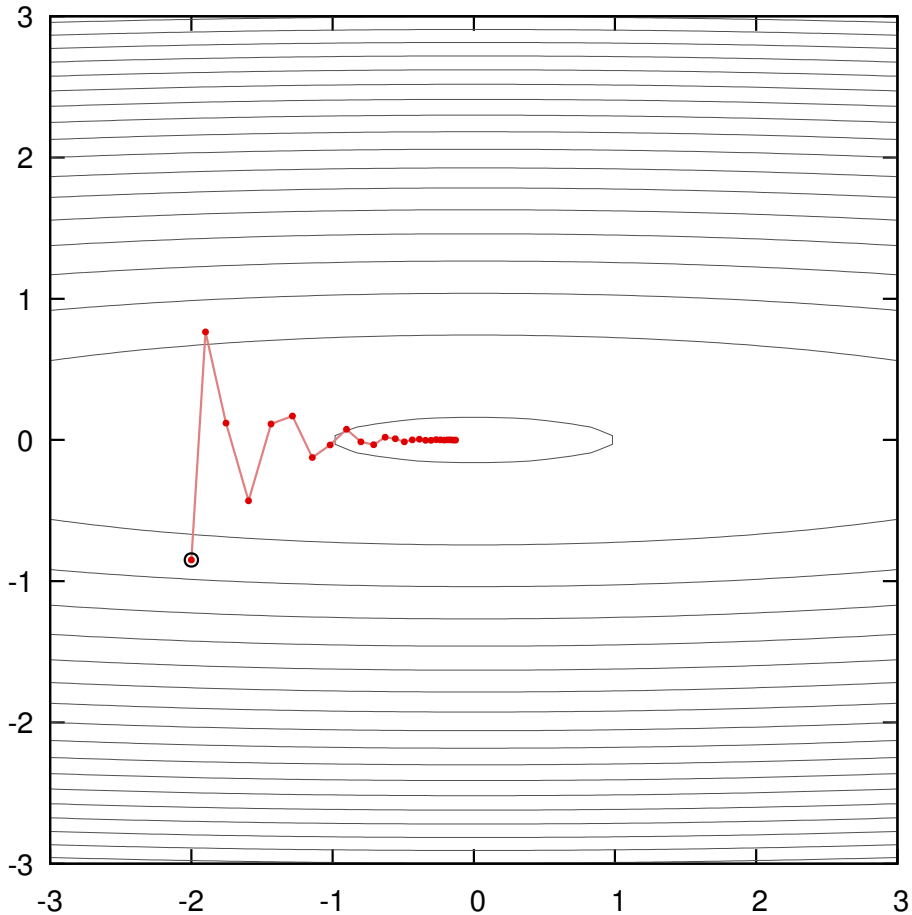
$$(u = \gamma u + \eta g) \Rightarrow \left(u = \frac{\eta}{1-\gamma} g\right),$$

- it dampens oscillations in narrow valleys.

$\eta = 5.0e-2, \gamma = 0$



$\eta = 5.0e-2, \gamma = 0.5$



Another class of methods exploits the statistics over the previous steps to compensate for the anisotropy of the mapping.

The Adam algorithm uses moving averages of each coordinate and its square to rescale each coordinate separately.

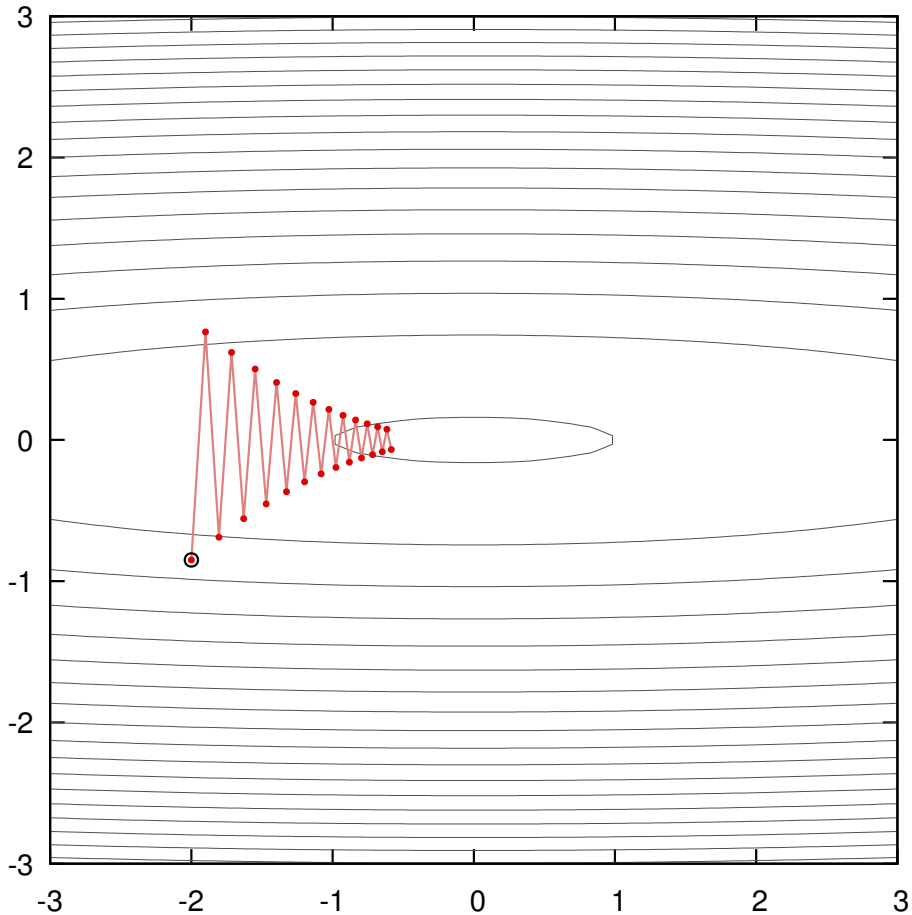
The update rule is, **on each coordinate separately**

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\
w_{t+1} &= w_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t
\end{aligned}
$$

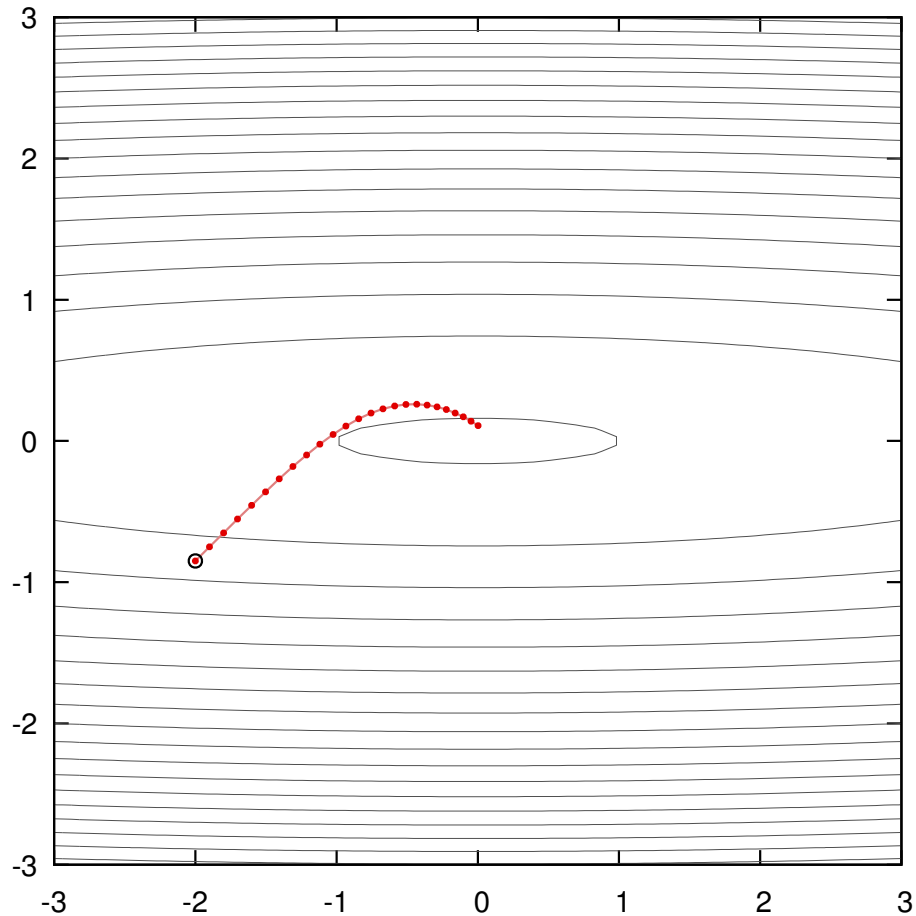
(Kingma and Ba, 2014)

This can be seen as a combination of momentum, with $\hat{m}_t$, and a per-coordinate re-scaling with $\hat{v}_t$.

$\eta = 5.0e-2$



Adam, $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 1e-8, \eta = 1.0e-1$



These two core strategies have been used in multiple incarnations:

- Nesterov's accelerated gradient,
- Adagrad,
- Adadelta,
- RMSprop,
- AdaMax,
- Nadam ...

Reinforcement Learning

Tutorial: Deep Reinforcement Learning

David Silver, Google DeepMind

Reinforcement Learning in a nutshell

RL is a general-purpose framework for decision-making

- RL is for an agent with the capacity to act
- Each action influences the agent's future state
- Success is measured by a scalar reward signal
- Goal: select actions to maximise future reward

Deep Learning in a nutshell

DL is a general-purpose framework for representation learning

- Given an objective
- Learn representation that is required to achieve objective
- Directly from raw inputs
- Using minimal domain knowledge

Deep Reinforcement Learning: AI = RL + DL

We seek a single agent which can solve any human-level task

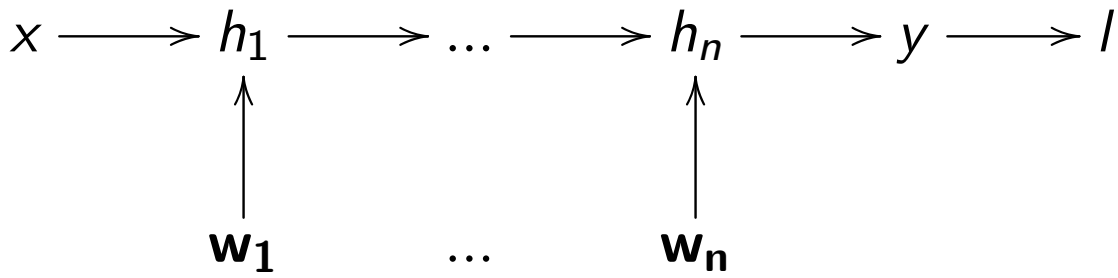
- RL defines the objective
- DL gives the mechanism
- RL + DL = general intelligence

Examples of Deep RL @DeepMind

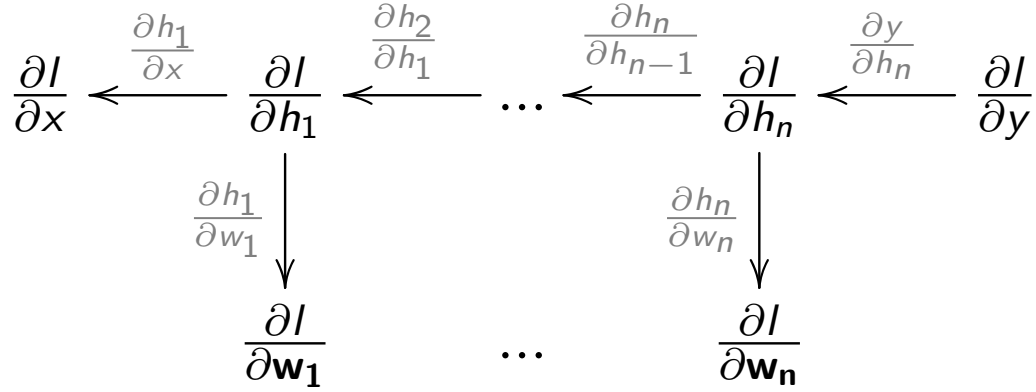
- Play games: Atari, poker, Go, ...
- Explore worlds: 3D worlds, Labyrinth, ...
- Control physical systems: manipulate, walk, swim, ...
- Interact with users: recommend, optimise, personalise, ...

Deep Representations

- A deep representation is a composition of many functions



- Its gradient can be backpropagated by the chain rule



Deep Neural Network

A deep neural network is typically composed of:

- Linear transformations

$$h_{k+1} = Wh_k$$

- Non-linear activation functions

$$h_{k+2} = f(h_{k+1})$$

- A loss function on the output, e.g.
  - Mean-squared error $l = ||y^* - y||^2$
  - Log likelihood $l = \log \mathbb{P}[y^*]$

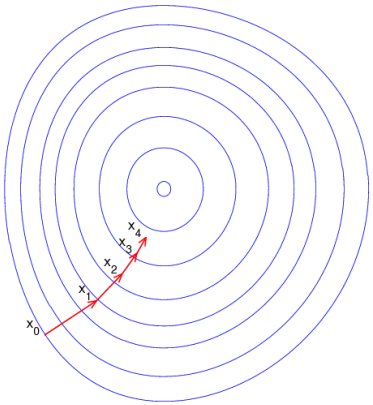
Training Neural Networks by Stochastic Gradient Descent

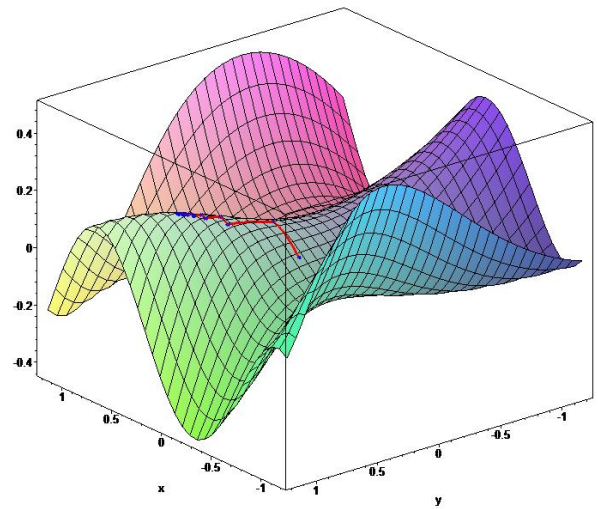
- Sample gradient of expected loss $L(\mathbf{w}) = \mathbb{E}[l]$

$$\frac{\partial l}{\partial \mathbf{w}} \sim \mathbb{E}\left[\frac{\partial l}{\partial \mathbf{w}}\right] = \frac{\partial L(\mathbf{w})}{\partial \mathbf{w}}$$

- Adjust $\mathbf{w}$ down the sampled gradient

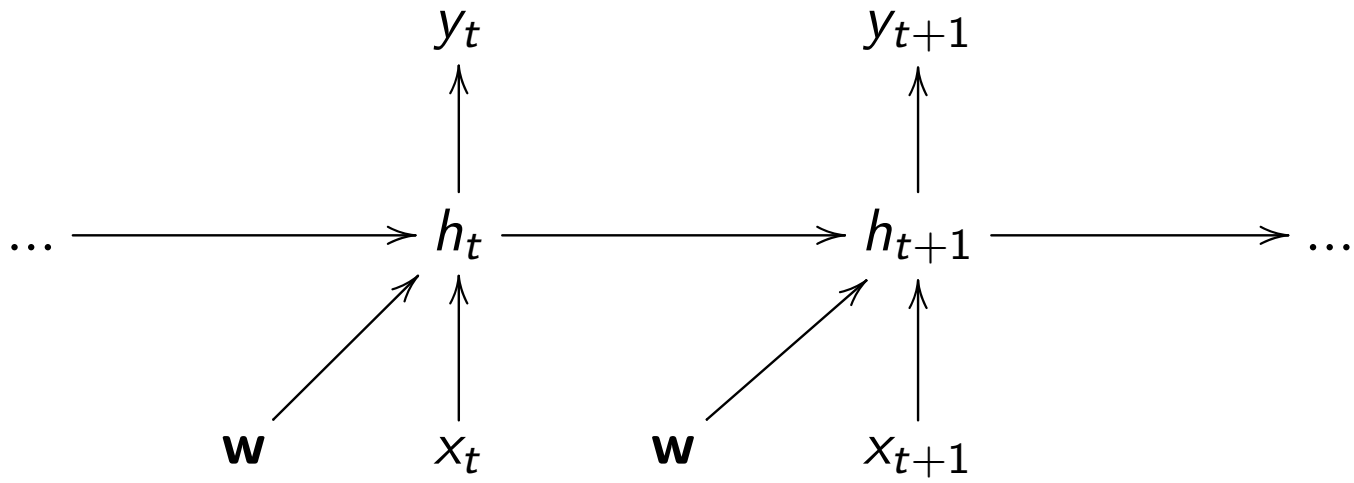
$$\Delta w \propto \frac{\partial l}{\partial \mathbf{w}}$$



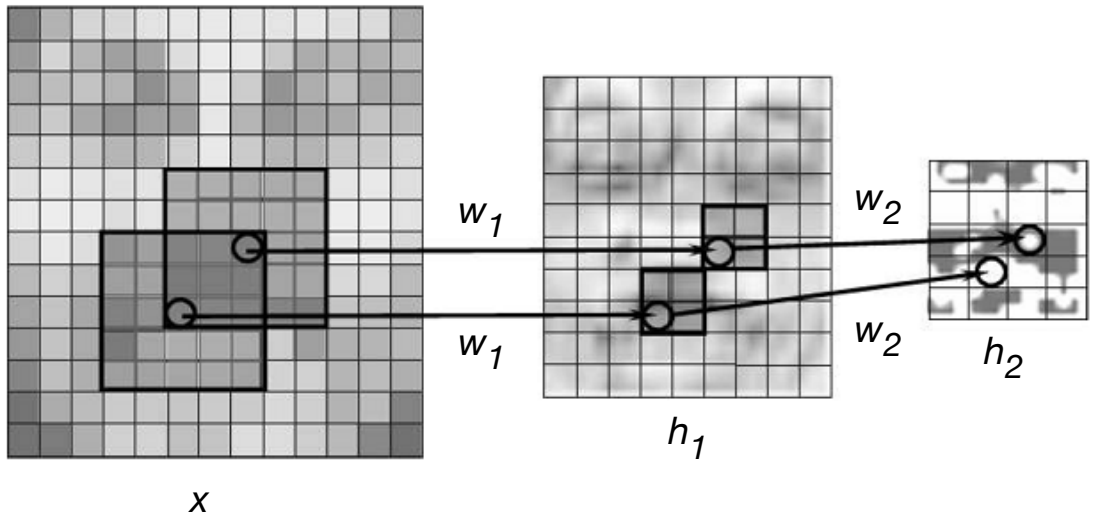


Weight Sharing

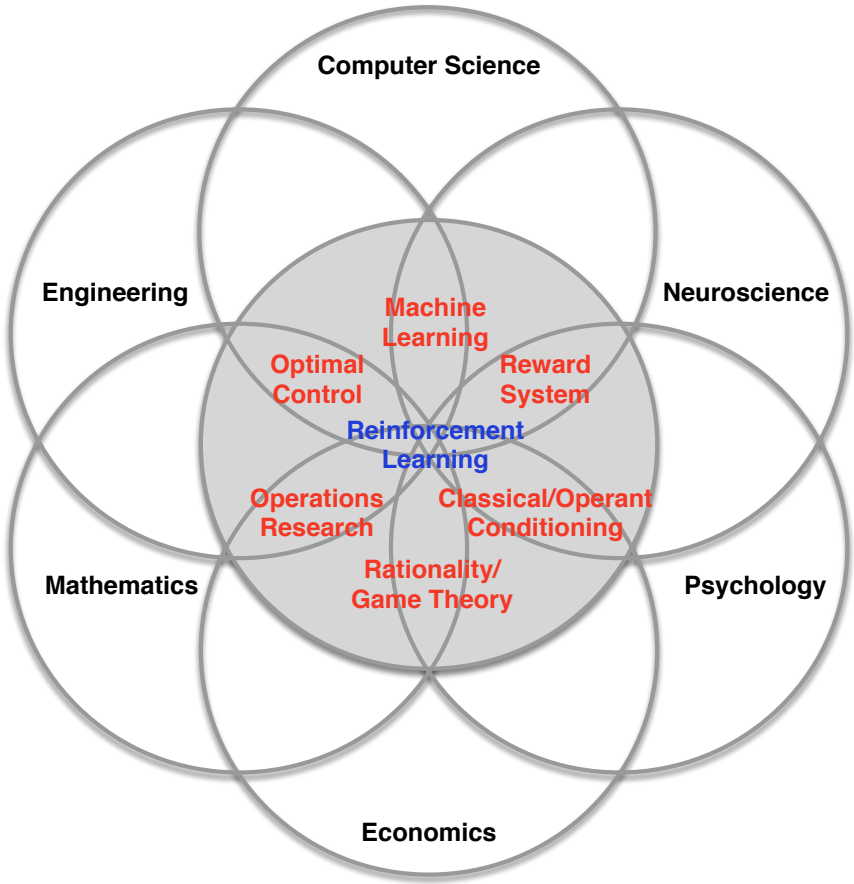
Recurrent neural network shares weights between time-steps



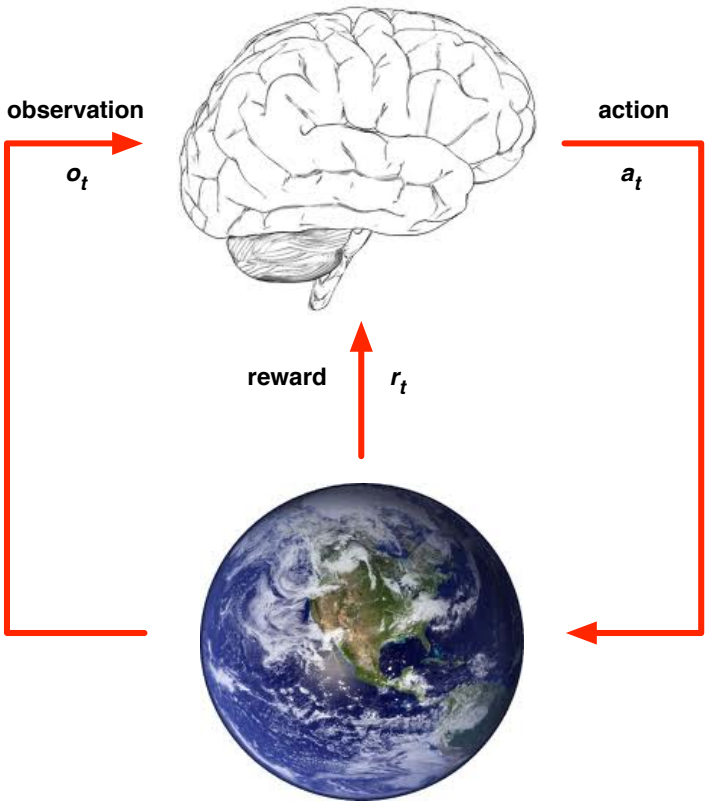
Convolutional neural network shares weights between local regions



Many Faces of Reinforcement Learning



Agent and Environment



- At each step t the agent:
  - Executes action a_t
  - Receives observation o_t
  - Receives scalar reward r_t
- The environment:
  - Receives action a_t
  - Emits observation o_{t+1}
  - Emits scalar reward r_{t+1}

State

- Experience is a sequence of observations, actions, rewards

$$o_1, r_1, a_1, ..., a_{t-1}, o_t, r_t$$

- The state is a summary of experience

$$s_t = f(o_1, r_1, a_1, ..., a_{t-1}, o_t, r_t)$$

- In a fully observed environment

$$s_t = f(o_t)$$

Major Components of an RL Agent

- An RL agent may include one or more of these components:
  - Policy: agent's behaviour function
  - Value function: how good is each state and/or action
  - Model: agent's representation of the environment

Policy

- A policy is the agent's behaviour
- It is a map from state to action:
  - Deterministic policy: $a = \pi(s)$
  - Stochastic policy: $\pi(a|s) = \mathbb{P}[a|s]$

Value Function

- A value function is a prediction of future reward
  - "How much reward will I get from action a in state s?"
- Q-value function gives expected total reward
  - from state s and action a
  - under policy π
  - with discount factor γ

$$Q^{\pi}(s,a) = \mathbb{E}\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots \mid s, a\right]$$

Value Function

- A value function is a prediction of future reward
  - "How much reward will I get from action a in state s?"
- Q-value function gives expected total reward
  - from state s and action a
  - under policy π
  - with discount factor γ

$$Q^{\pi}(s,a) = \mathbb{E}\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots \mid s, a\right]$$

- Value functions decompose into a Bellman equation

$$Q^{\pi}(s,a) = \mathbb{E}_{s',a'}\left[r + \gamma Q^{\pi}(s',a') \mid s, a\right]$$

Optimal Value Functions

- An optimal value function is the maximum achievable value

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) = Q^{\pi^*}(s,a)$$

Optimal Value Functions

- An optimal value function is the maximum achievable value

$$Q^*(s,a) = \max_\pi Q^\pi(s,a) = Q^{\pi^*}(s,a)$$

- Once we have Q^* we can act optimally,

$$\pi^*(s) = \operatorname*{argmax}_a Q^*(s,a)$$

Optimal Value Functions

- An optimal value function is the maximum achievable value

$$Q^*(s,a) = \max_\pi Q^\pi(s,a) = Q^{\pi^*}(s,a)$$

- Once we have Q^* we can act optimally,

$$\pi^*(s) = \operatorname*{argmax}_a Q^*(s,a)$$

- Optimal value maximises over all decisions. Informally:

$$\begin{aligned} Q^*(s,a) &= r_{t+1} + \gamma \max_{a_{t+1}} r_{t+2} + \gamma^2 \max_{a_{t+2}} r_{t+3} + \ldots \\ &= r_{t+1} + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) \end{aligned}$$

Optimal Value Functions

- An optimal value function is the maximum achievable value

$$Q^*(s,a) = \max_\pi Q^\pi(s,a) = Q^{\pi^*}(s,a)$$

- Once we have Q^* we can act optimally,

$$\pi^*(s) = \operatorname*{argmax}_a Q^*(s,a)$$

- Optimal value maximises over all decisions. Informally:

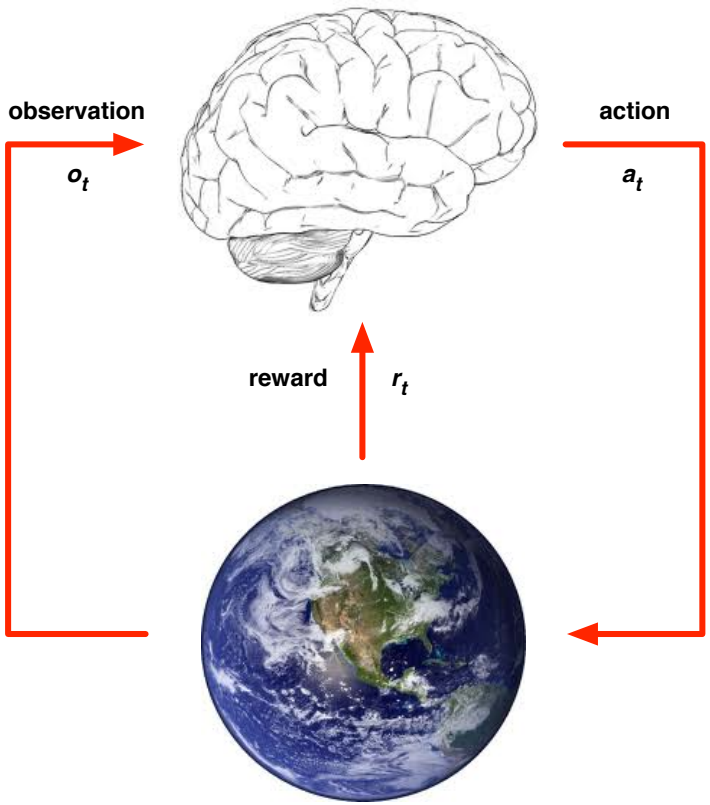
$$\begin{aligned} Q^*(s,a) &= r_{t+1} + \gamma \max_{a_{t+1}} r_{t+2} + \gamma^2 \max_{a_{t+2}} r_{t+3} + \ldots \\ &= r_{t+1} + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) \end{aligned}$$

- Formally, optimal values decompose into a Bellman equation

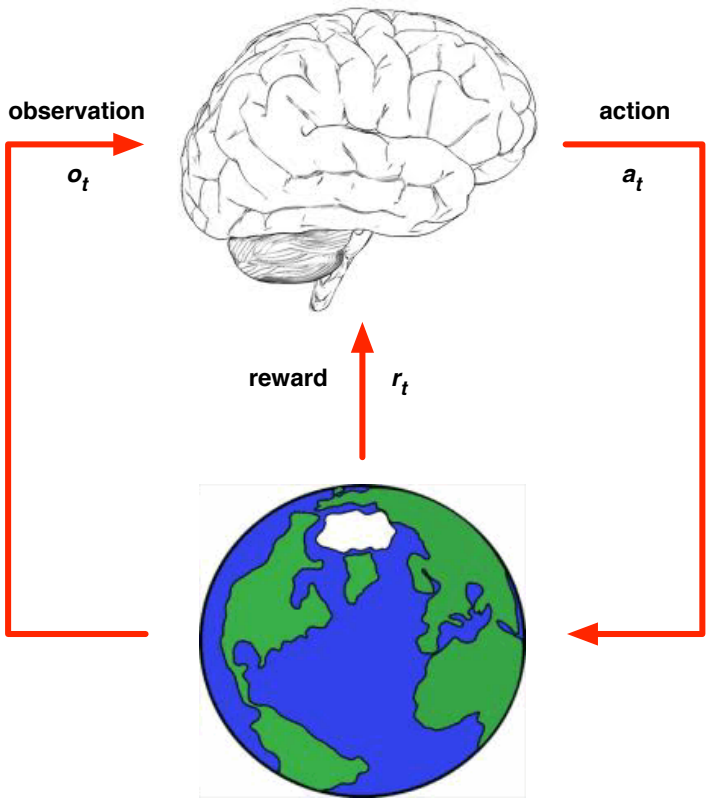
$$Q^*(s,a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q^*(s',a') \mid s,a \right]$$

Value Function Demo

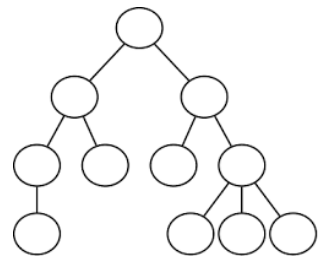
Model



Model



- Model is learnt from experience
- Acts as proxy for environment
- Planner interacts with model
- e.g. using lookahead search



Approaches To Reinforcement Learning

Value-based RL

- Estimate the optimal value function $Q^*(s, a)$
- This is the maximum value achievable under any policy

Policy-based RL

- Search directly for the optimal policy π^*
- This is the policy achieving maximum future reward

Model-based RL

- Build a model of the environment
- Plan (e.g. by lookahead) using model

Deep Reinforcement Learning

- Use deep neural networks to represent
  - Value function
  - Policy
  - Model
- Optimise loss function by stochastic gradient descent

Convolutional Neural Networks

If they were handled as normal "unstructured" vectors, large-dimension signals such as sound samples or images would require models of intractable size.

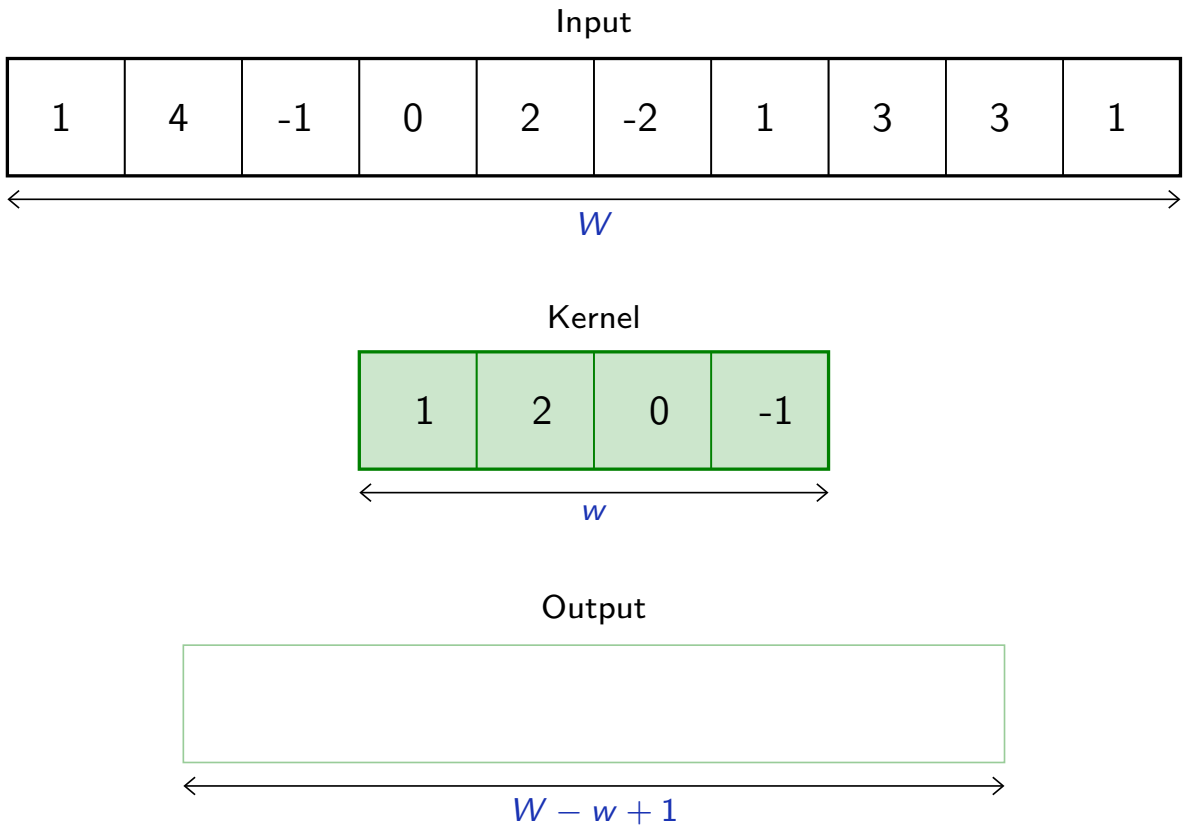
For instance a linear layer taking a 256×256 RGB image as input, and producing an image of same size would require

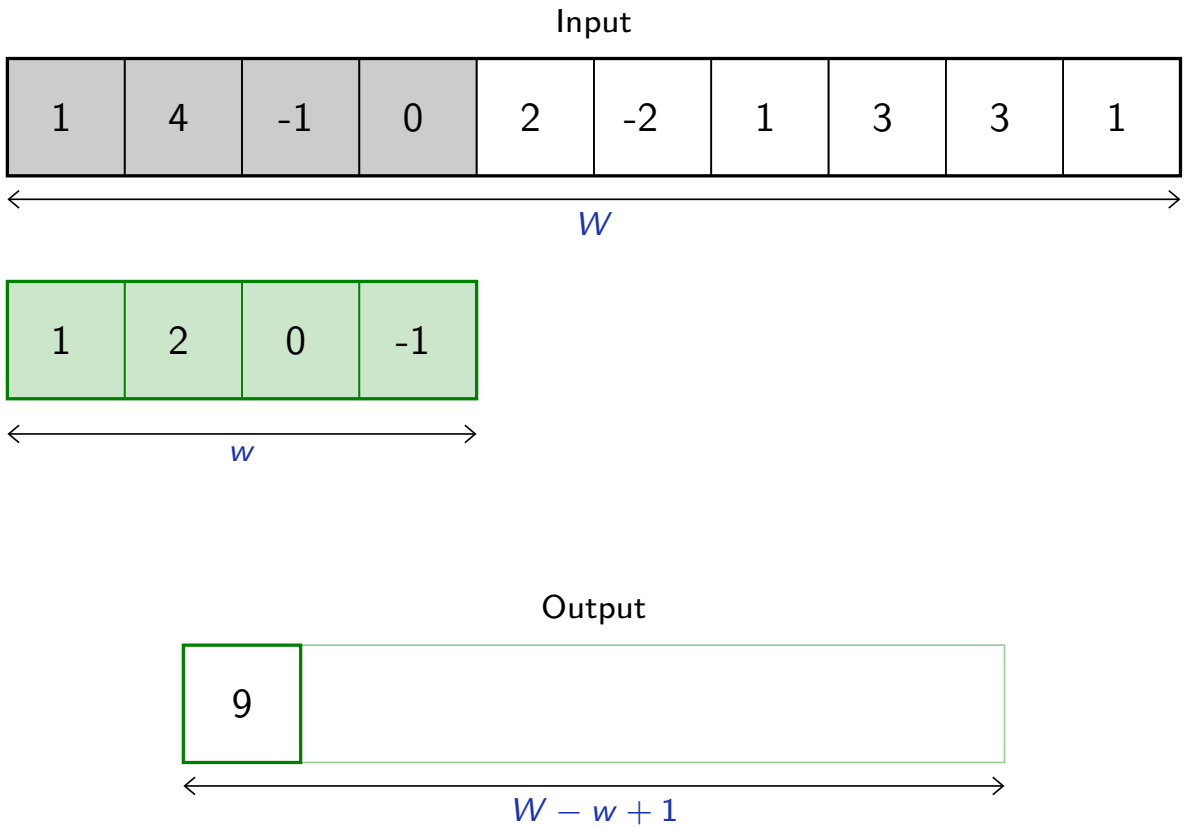
$$(256 \times 256 \times 3)^2 \simeq 3.87e{+}10$$

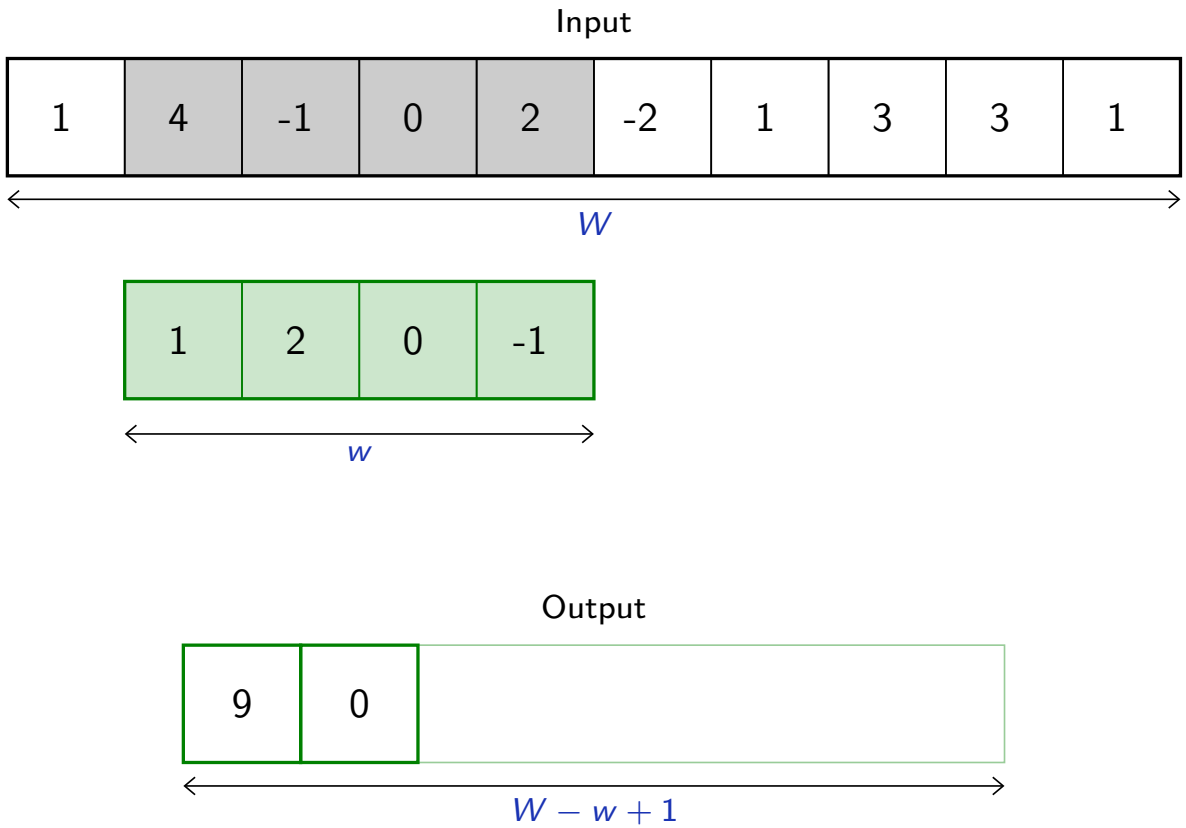
parameters, with the corresponding memory footprint ($\simeq$150Gb !), and excess of capacity.

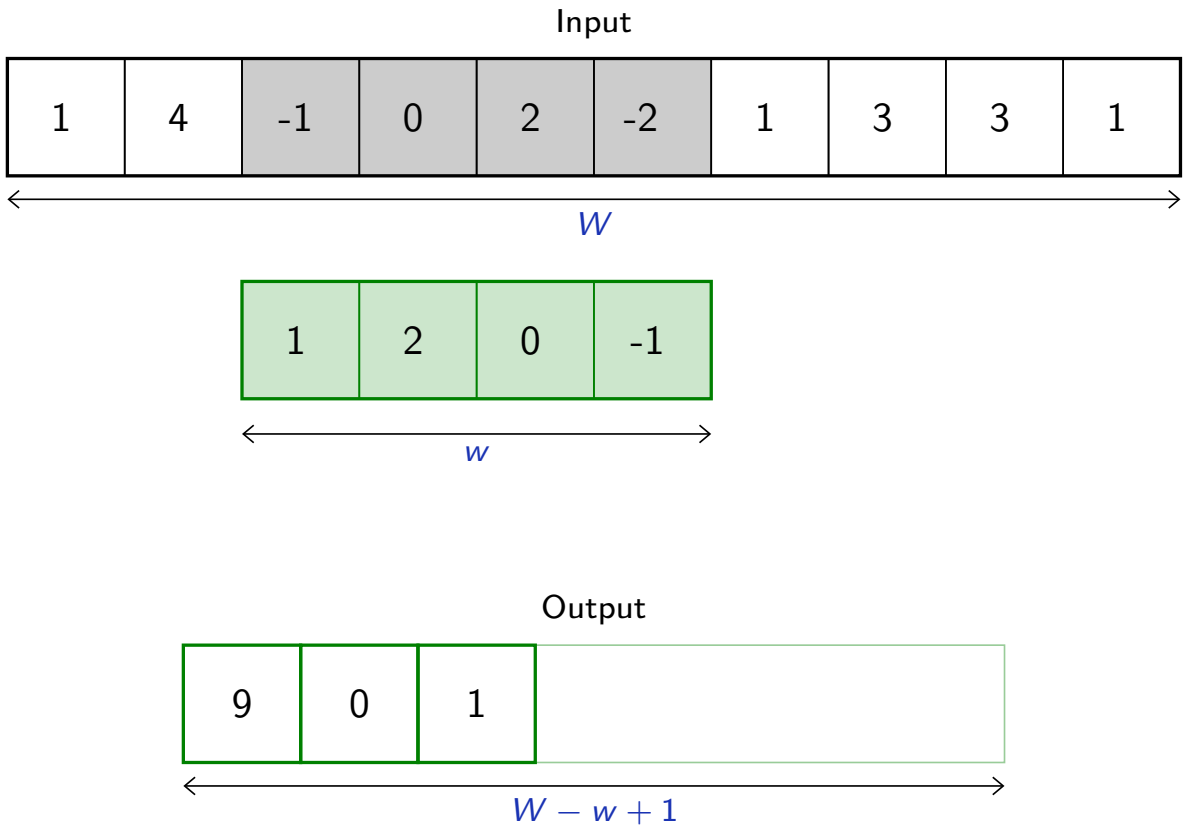
Moreover, this requirement is inconsistent with the intuition that such large signals have some "invariance in translation". **A representation meaningful at a certain location can / should be used everywhere.**

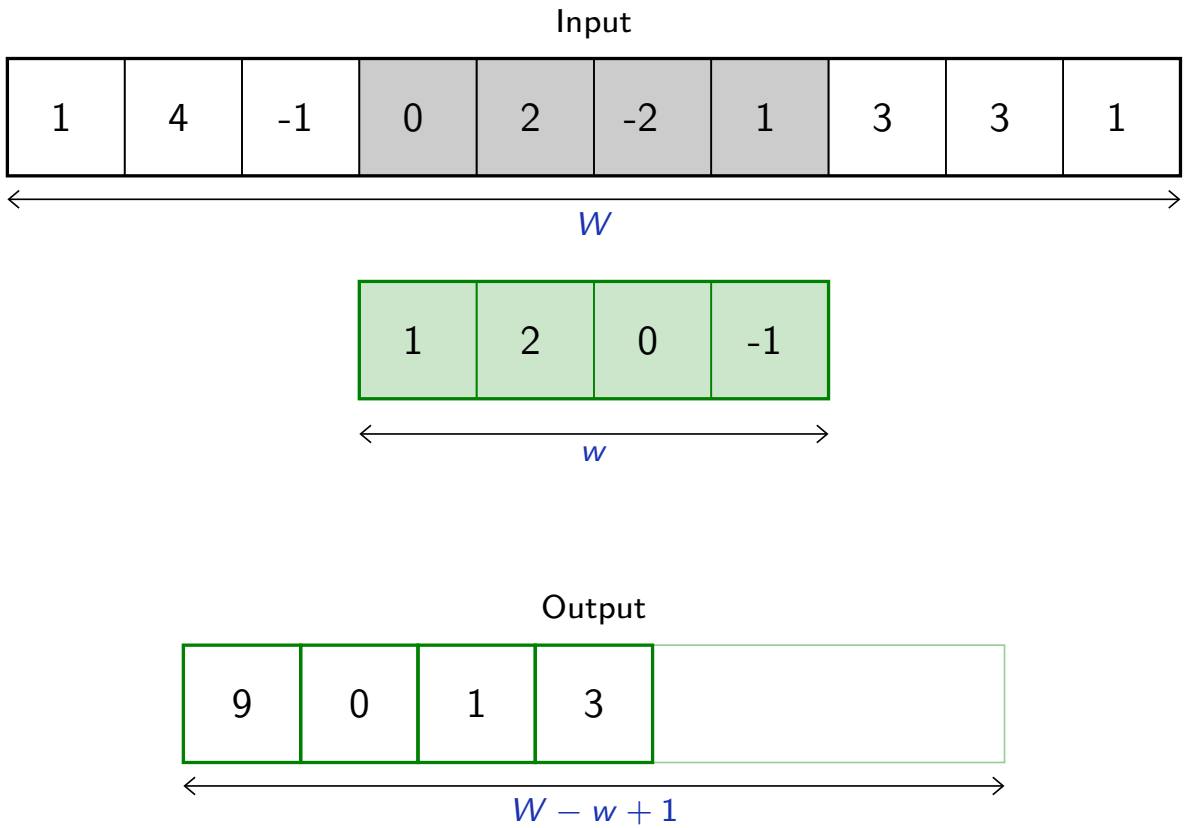
A convolution layer embodies this idea. It applies the same linear transformation locally, everywhere, and preserves the signal structure.

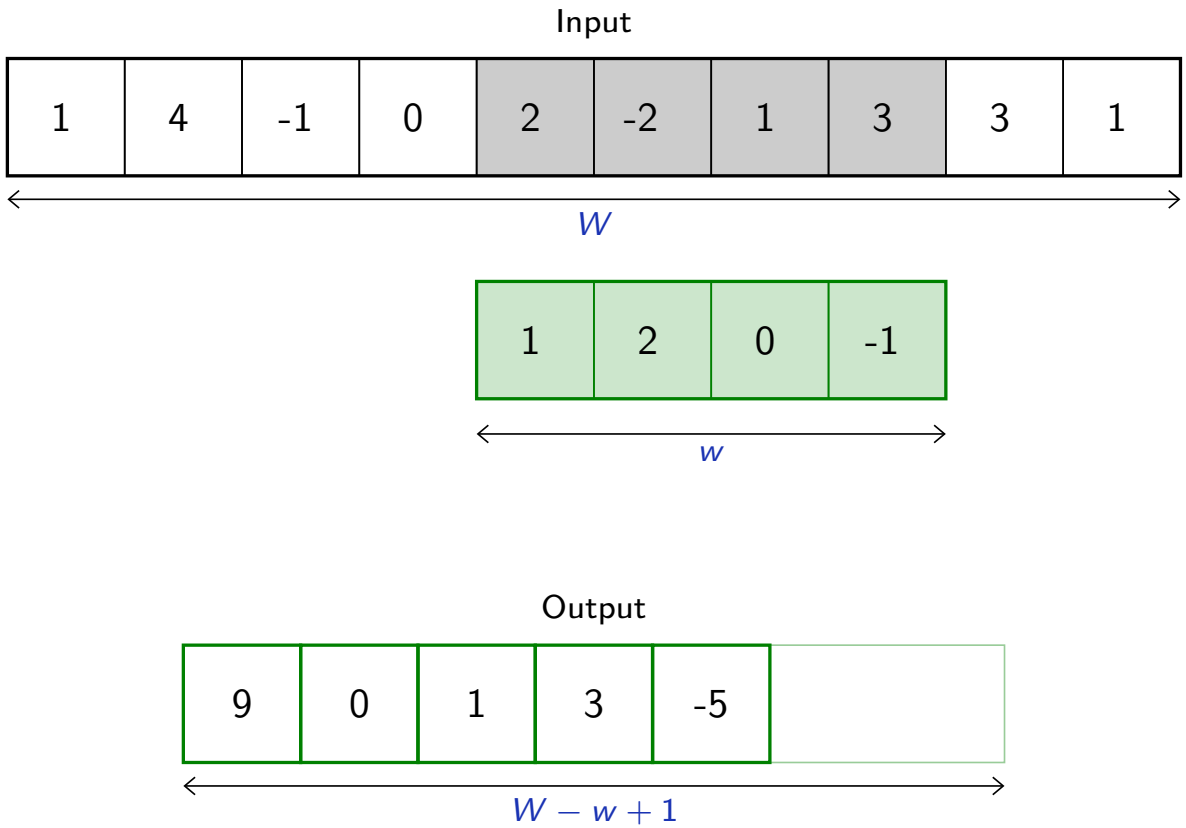

Input
1
4
-1
0
2
-2
1
3
3
1
W
Kernel
1
2
0
-1
w
Output
W − w + 1

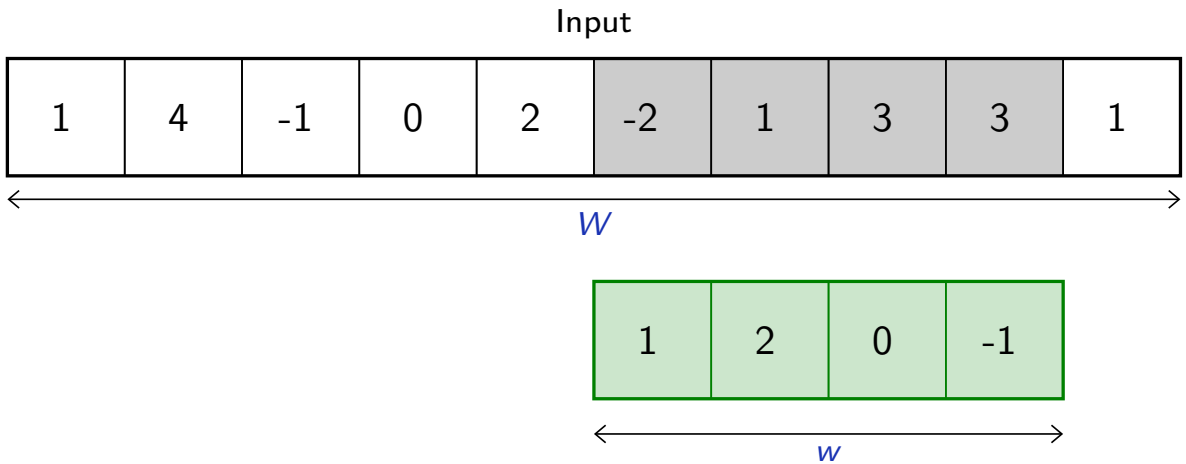

Input
1
4
-1
0
2
-2
1
3
3
1
W
1
2
0
-1
w
Output
9
W − w + 1

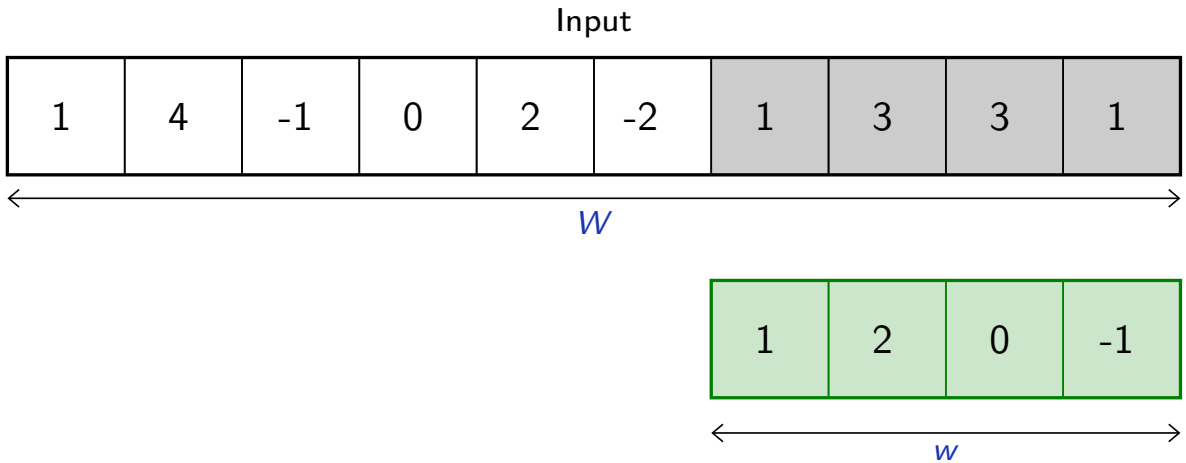

Input
1
4
-1
0
2
-2
1
3
3
1
W
1
2
0
-1
w
Output
9
0
W − w + 1

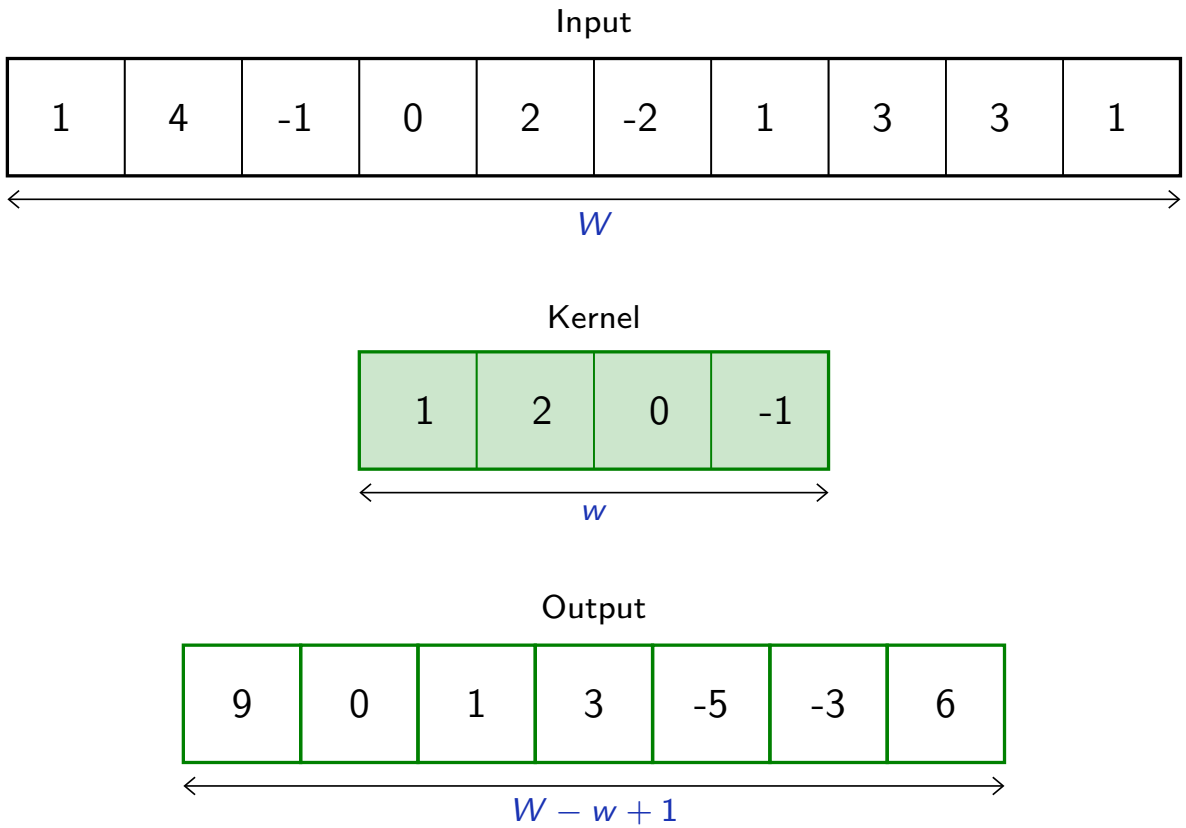

Input
1
4
-1
0
2
-2
1
3
3
1
W
1
2
0
-1
w
Output
9
0
1
W − w + 1


Input
1
4
-1
0
2
-2
1
3
3
1
W
1
2
0
-1
w
Output
9
0
1
3
W − w + 1


Input
1
4
-1
0
2
-2
1
3
3
1
W
1
2
0
-1
w
Output
9
0
1
3
-5
W − w + 1


Input
1
4
-1
0
2
-2
1
3
3
1
W
1
2
0
-1
w
Output
9
0
1
3
-5
-3
W − w + 1


Input
1
4
-1
0
2
-2
1
3
3
1
W
1
2
0
-1
w
Output
9
0
1
3
-5
-3
6
W − w + 1


Input
1
4
-1
0
2
-2
1
3
3
1
W
Kernel
1
2
0
-1
w
Output
9
0
1
3
-5
-3
6
W − w + 1

Formally, in 1d, given

$$x = (x_1, \ldots, x_W)$$

and a "convolution kernel" (or "filter") of width w

$$u = (u_1, \ldots, u_w)$$

the convolution $x \circledast u$ is a vector of size $W - w + 1$, with

$$\begin{aligned}(x \circledast u)_i &= \sum_{j=1}^{w} x_{i-1+j}\, u_j \\ &= (x_i, \ldots, x_{i+w-1}) \cdot u\end{aligned}$$

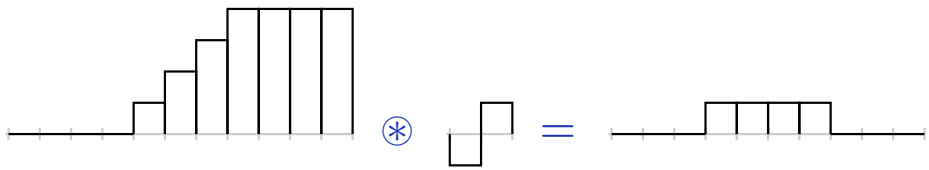
for instance

$$(1, 2, 3, 4) \circledast (3, 2) = (3 + 4, 6 + 6, 9 + 8) = (7, 12, 17).$$

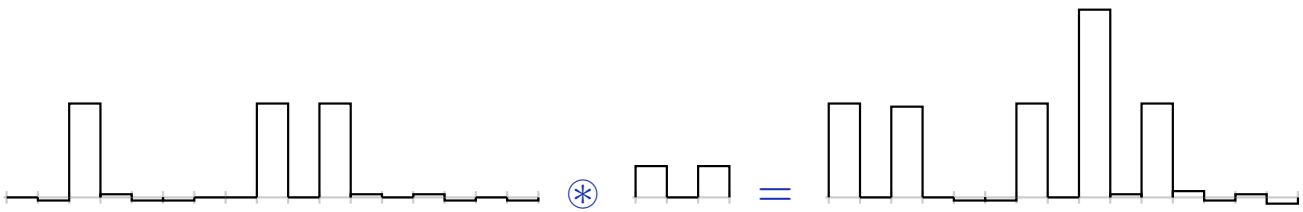
This differs from the usual convolution since the kernel and the signal are both visited in increasing index order.

Convolution can implement in particular differential operators, *e.g.*

$$(0,0,0,0,1,2,3,4,4,4,4) \circledast (-1,1) = (0,0,0,1,1,1,1,0,0,0).$$



or crude "template matcher", *e.g.*



Both of these computation examples are indeed "invariant by translation".

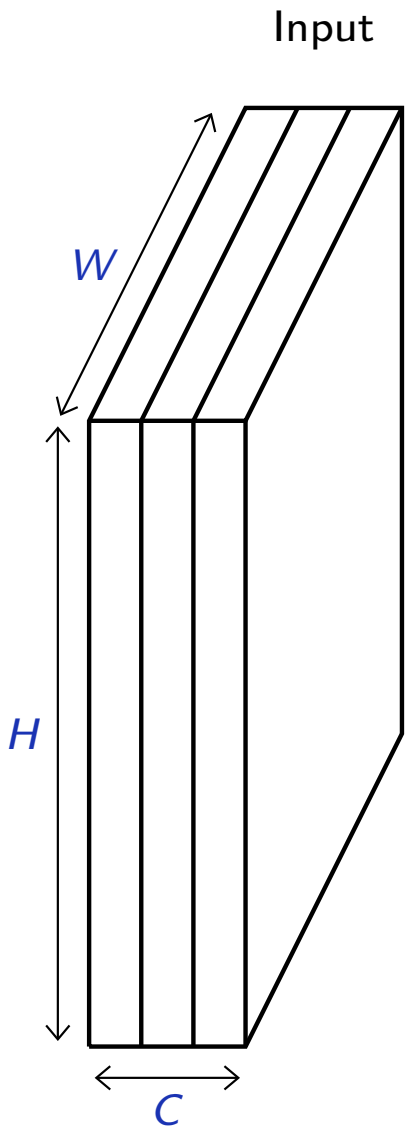
It generalizes naturally to a multi-dimensional input, although specification can become complicated.

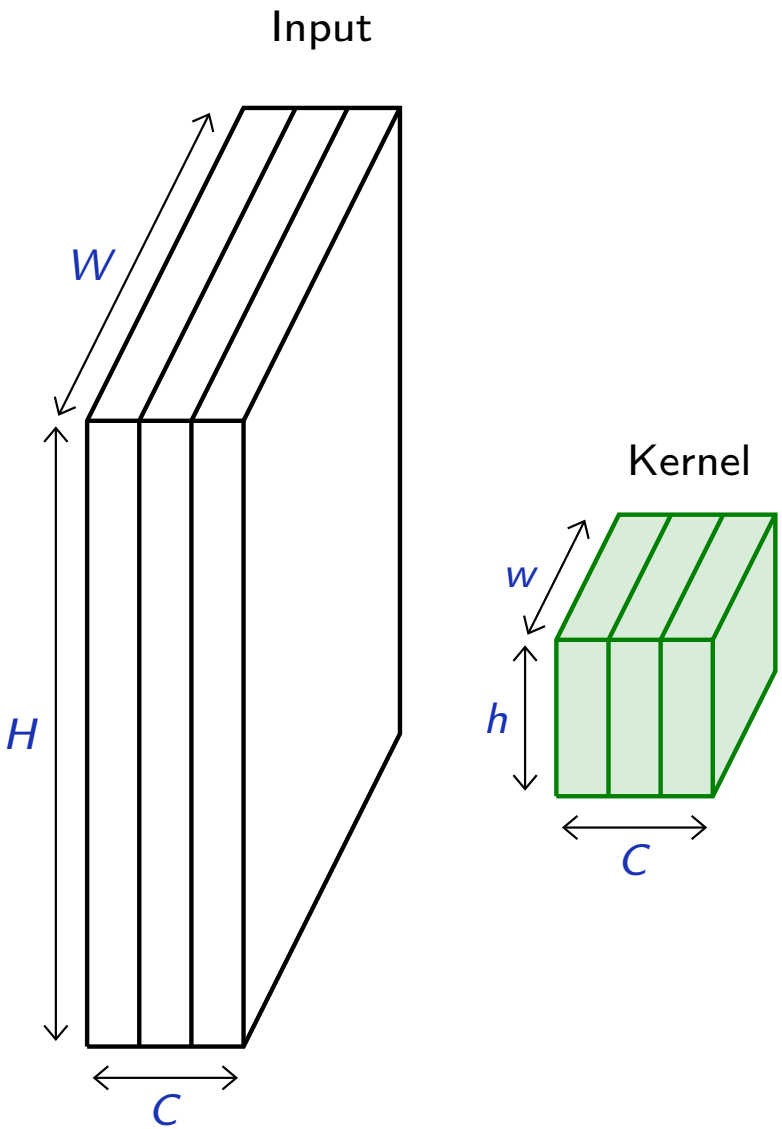
Its most usual form for "convolutional networks" processes a 3d tensor as input (*i.e.* a multi-channel 2d signal) to output a 2d tensor. The kernel is not swiped across channels, just across rows and columns.

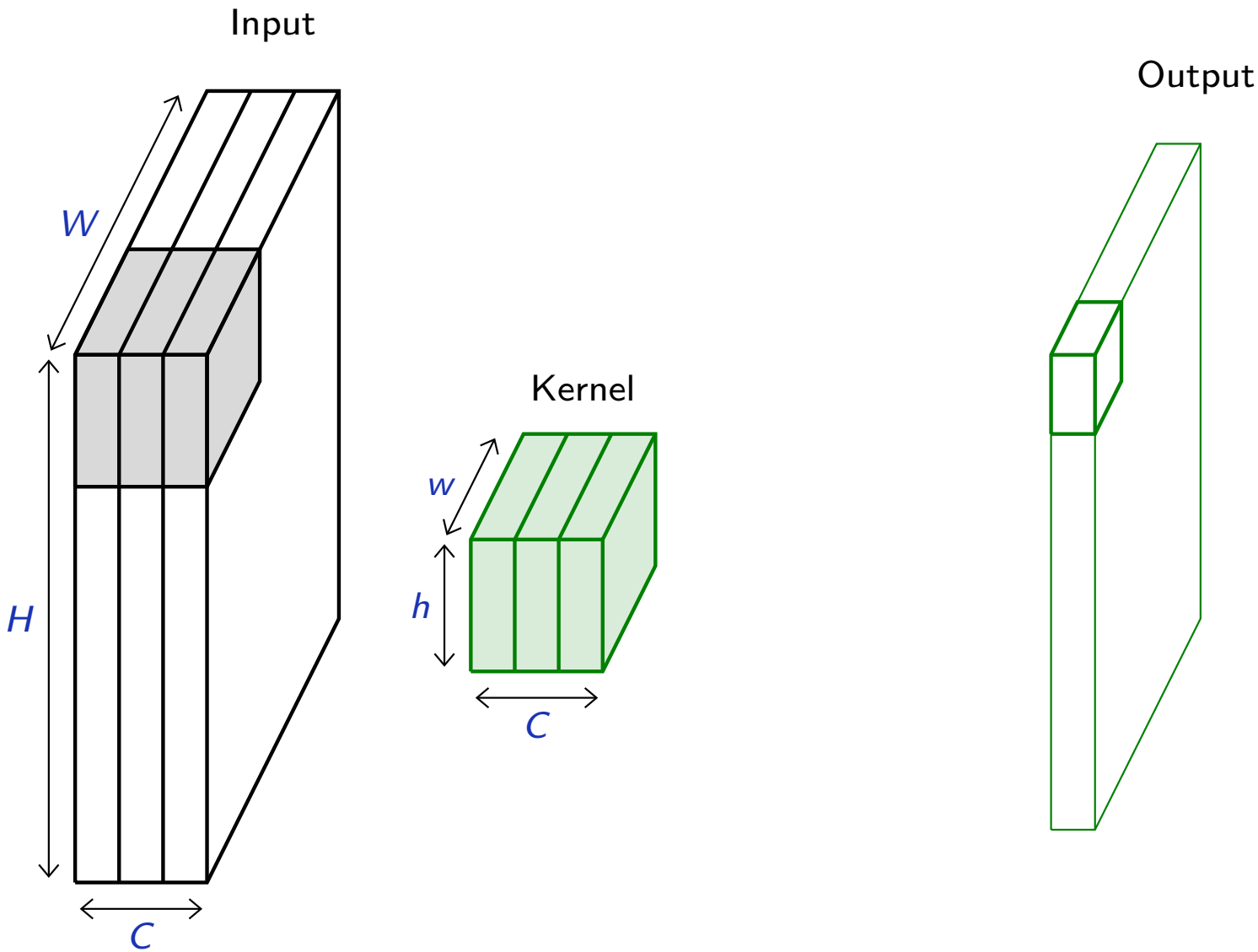
In this case, if the input tensor is of size $C \times H \times W$, and the kernel is $C \times h \times w$, the output is $(H - h + 1) \times (W - w + 1)$.

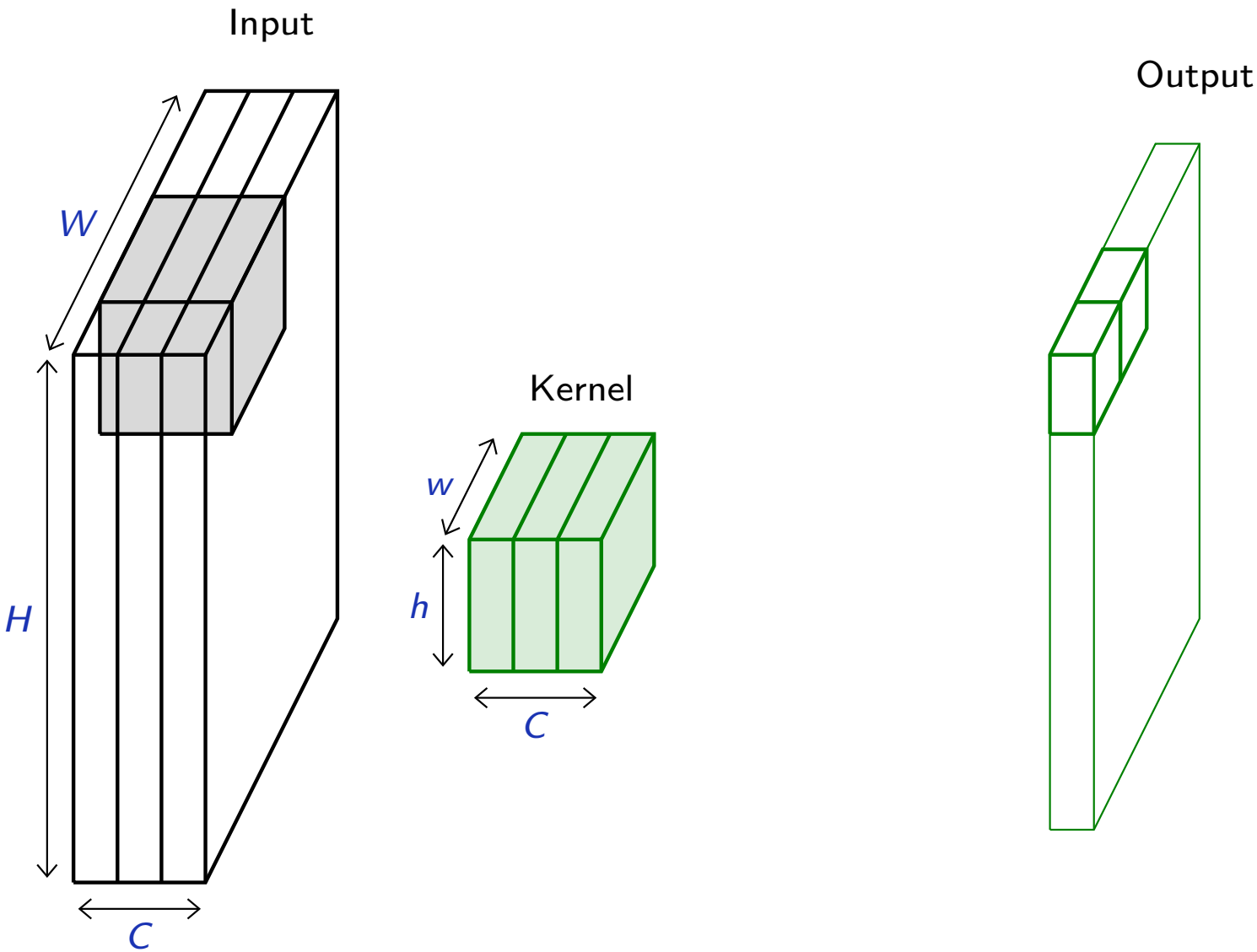
We say "2d signal" even though it has C channels, since it is a feature vector indexed by a 2d location without structure on the feature indexes.

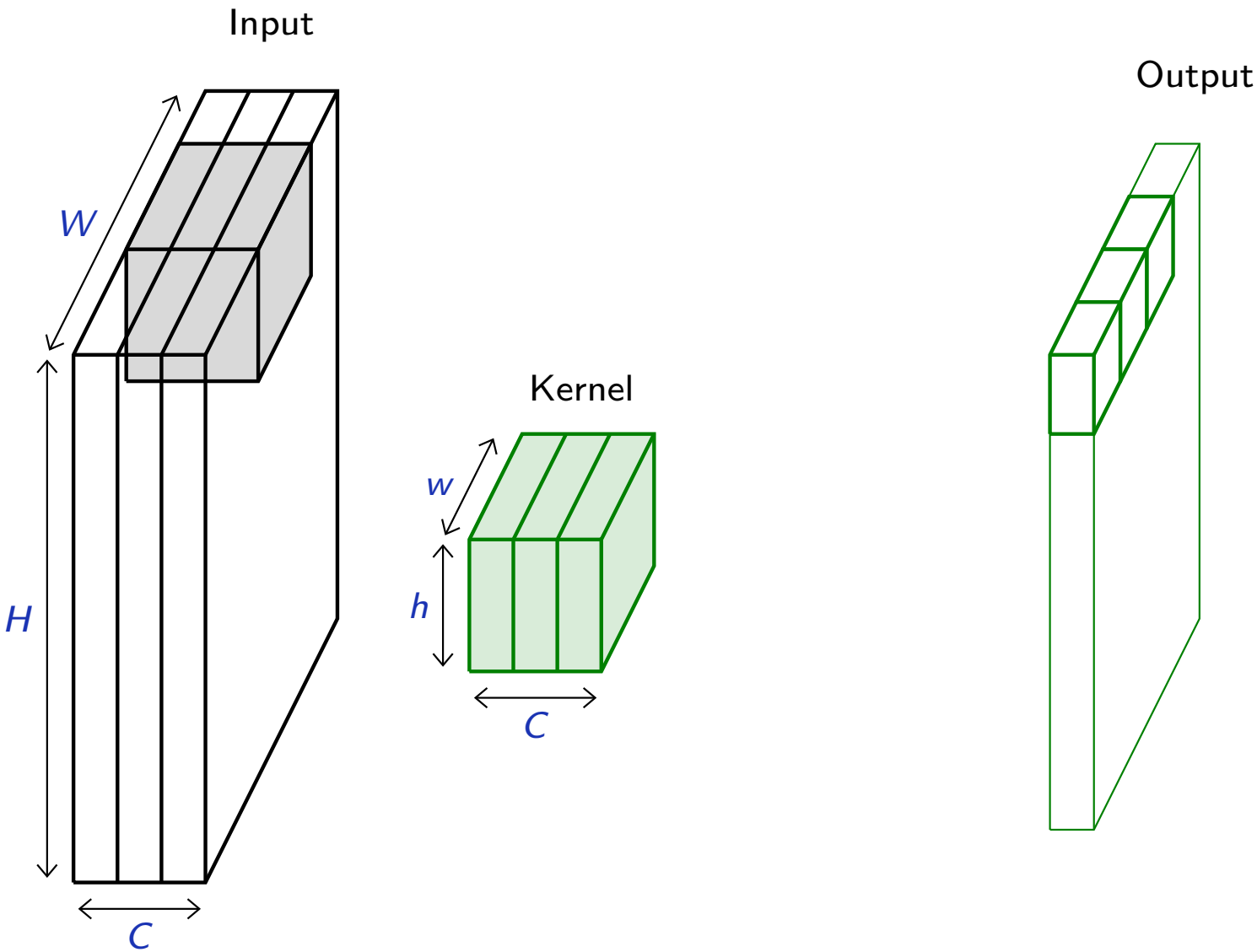
In a standard convolution layer, D such convolutions are combined to generate a $D \times (H - h + 1) \times (W - w + 1)$ output.

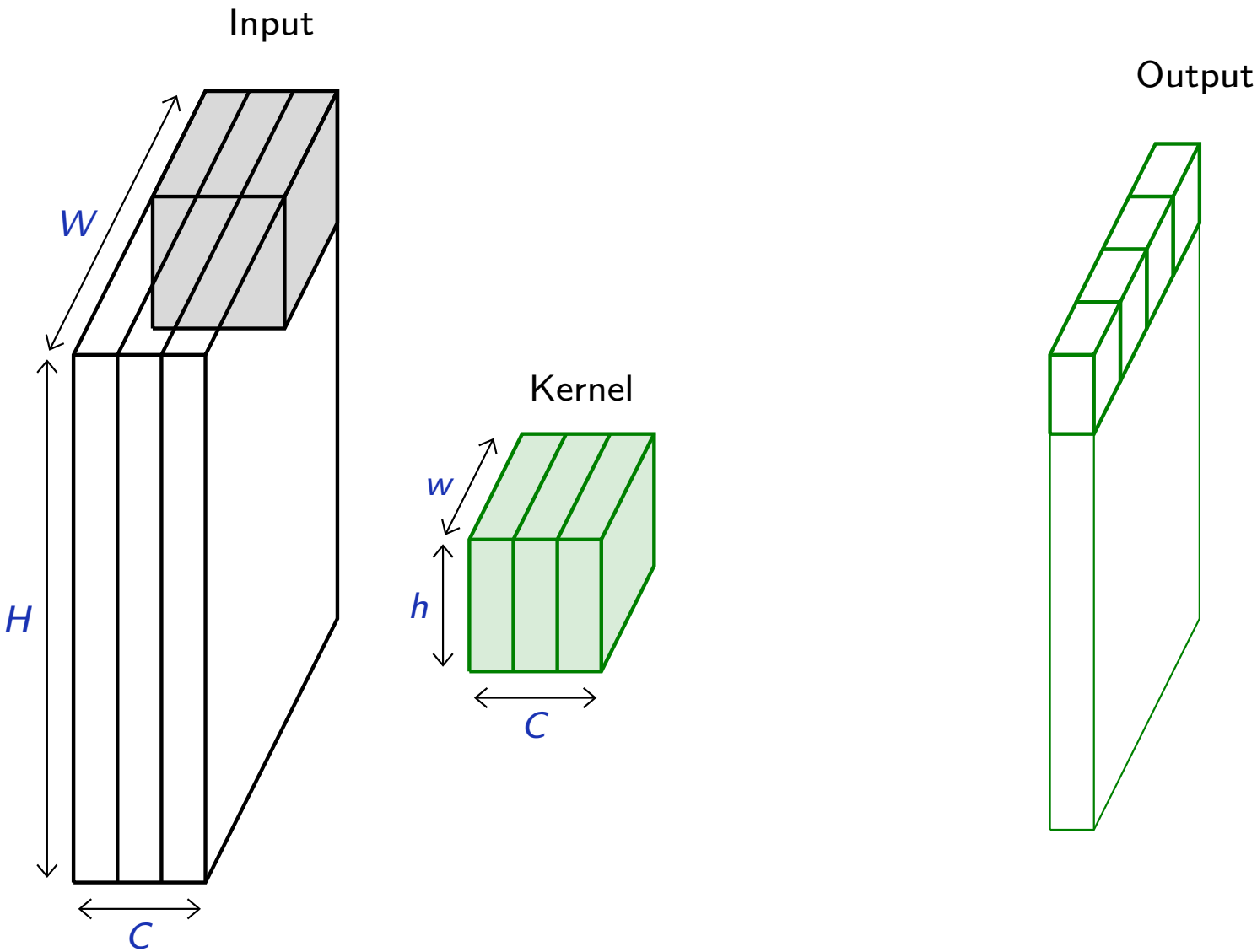

Input
W
H
C

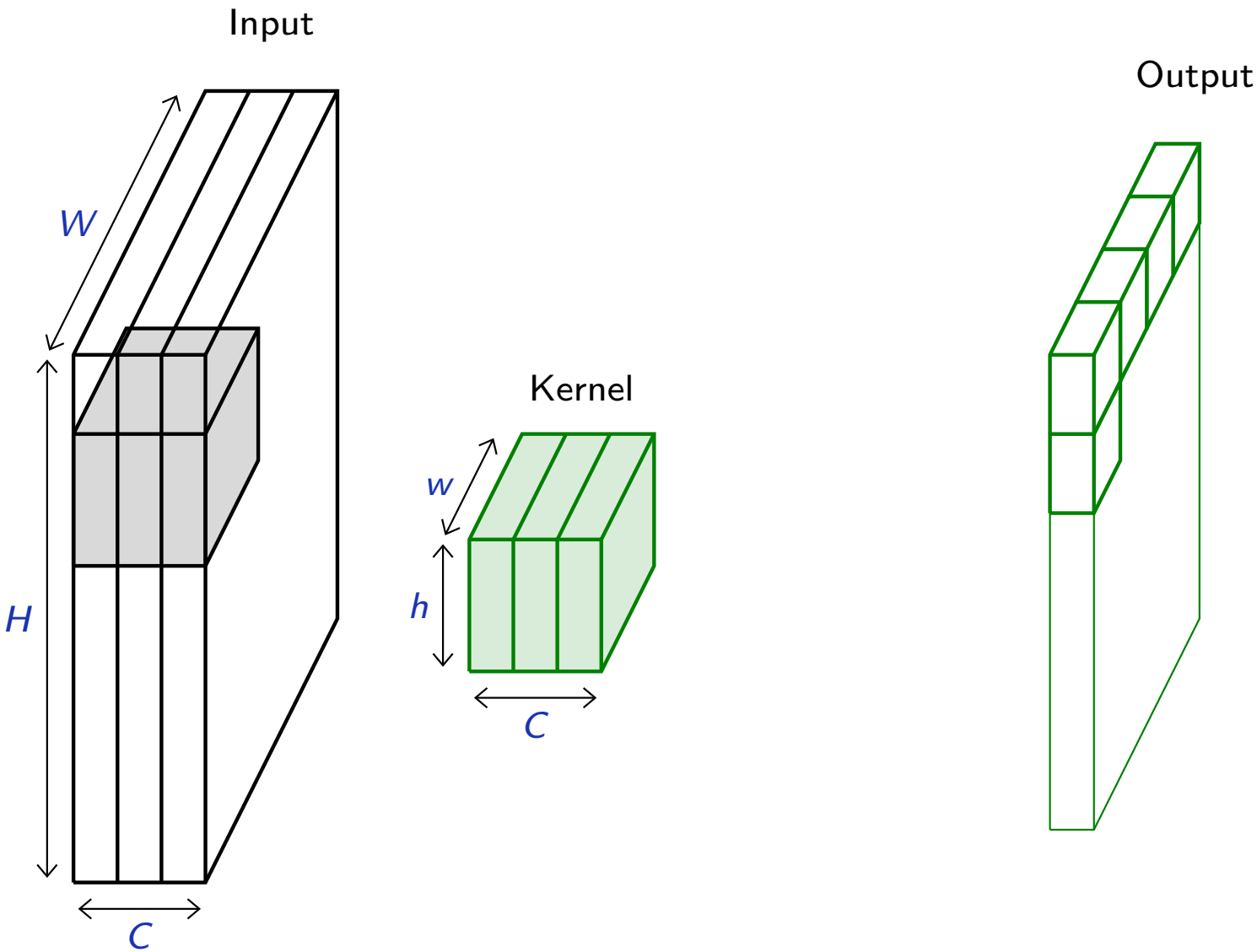

Input
W
H
C
Kernel
w
h
C

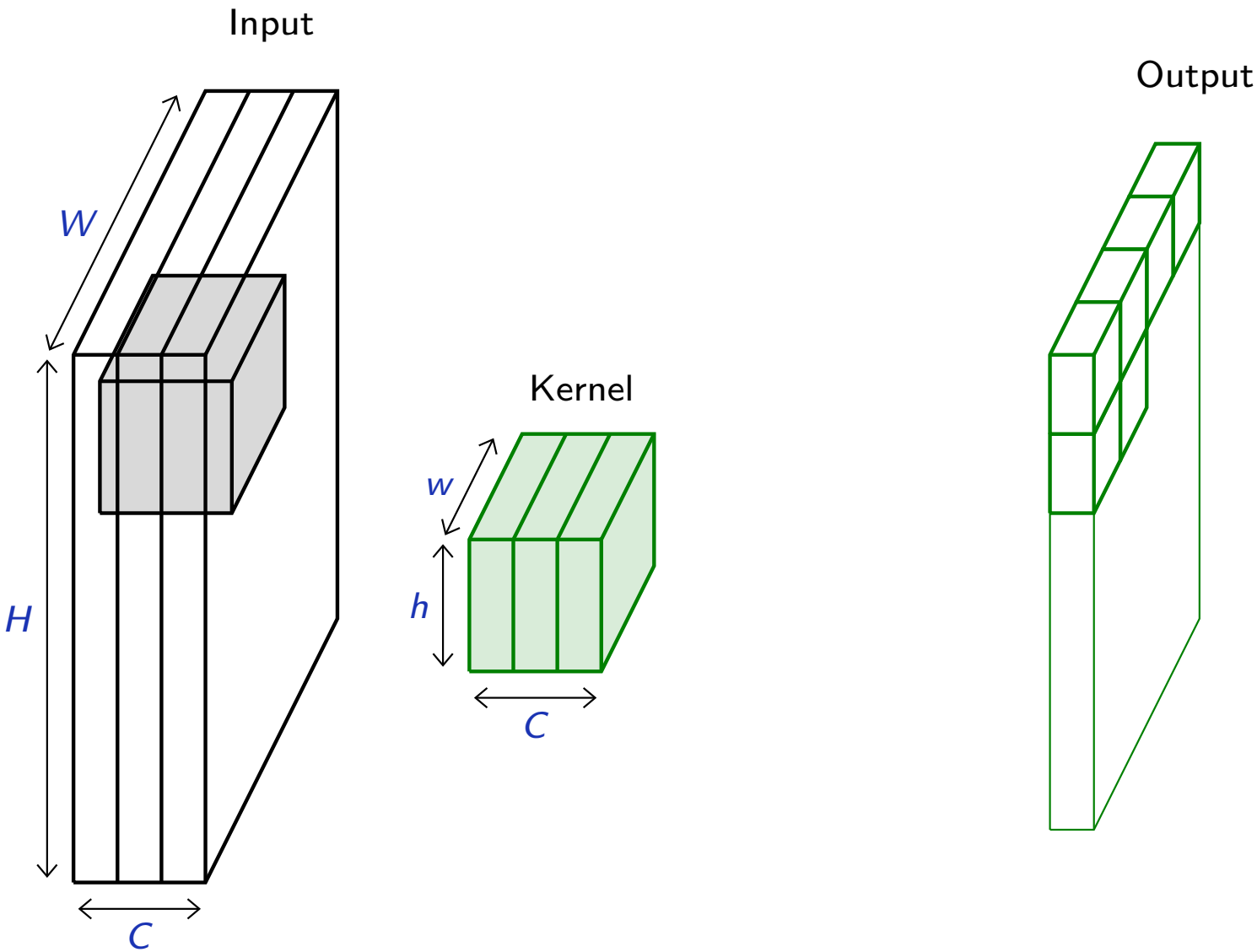

Input
W
H
C
Kernel
w
h
C
Output

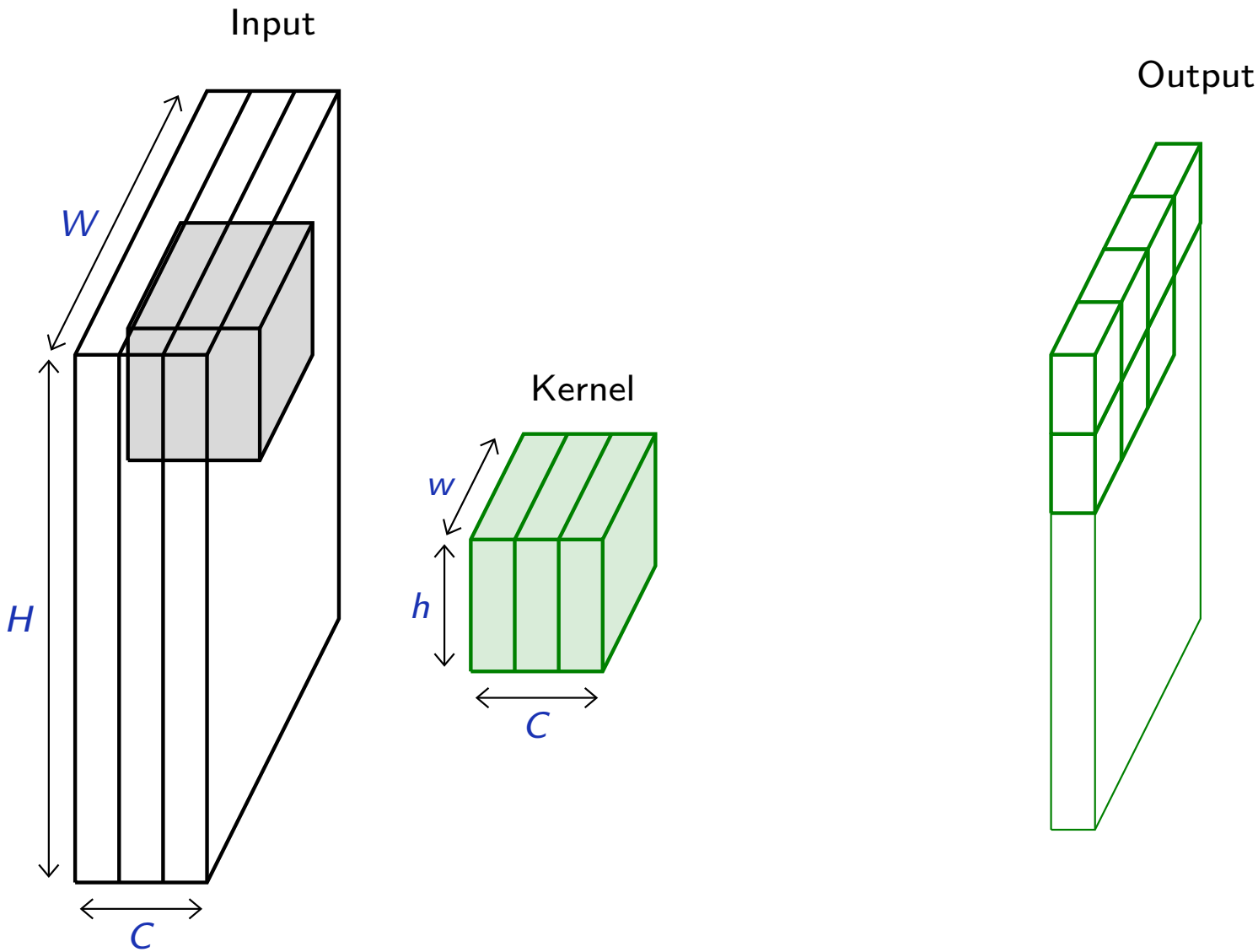

Input
W
H
C
Kernel
w
h
C
Output

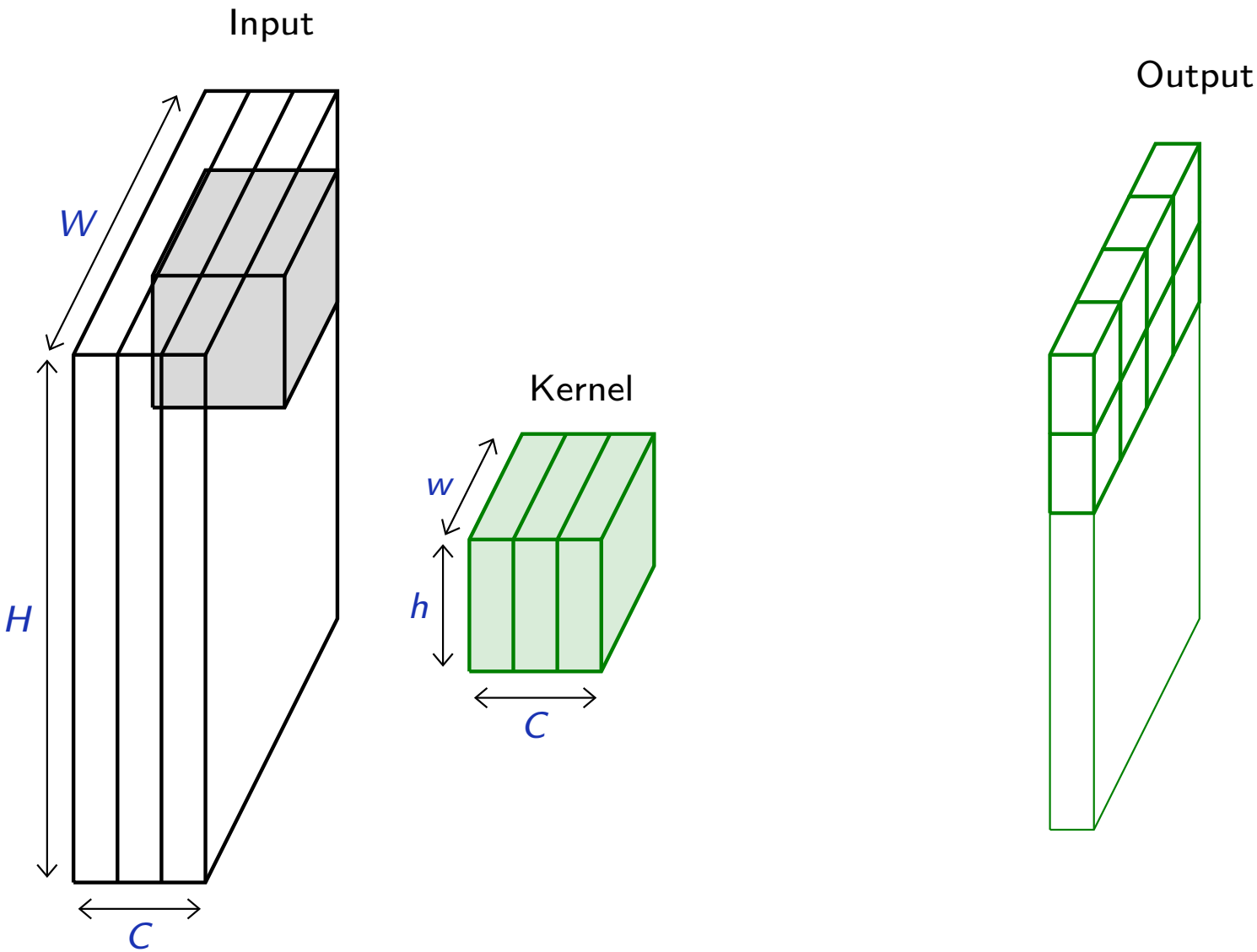

Input
W
H
C
Kernel
w
h
C
Output

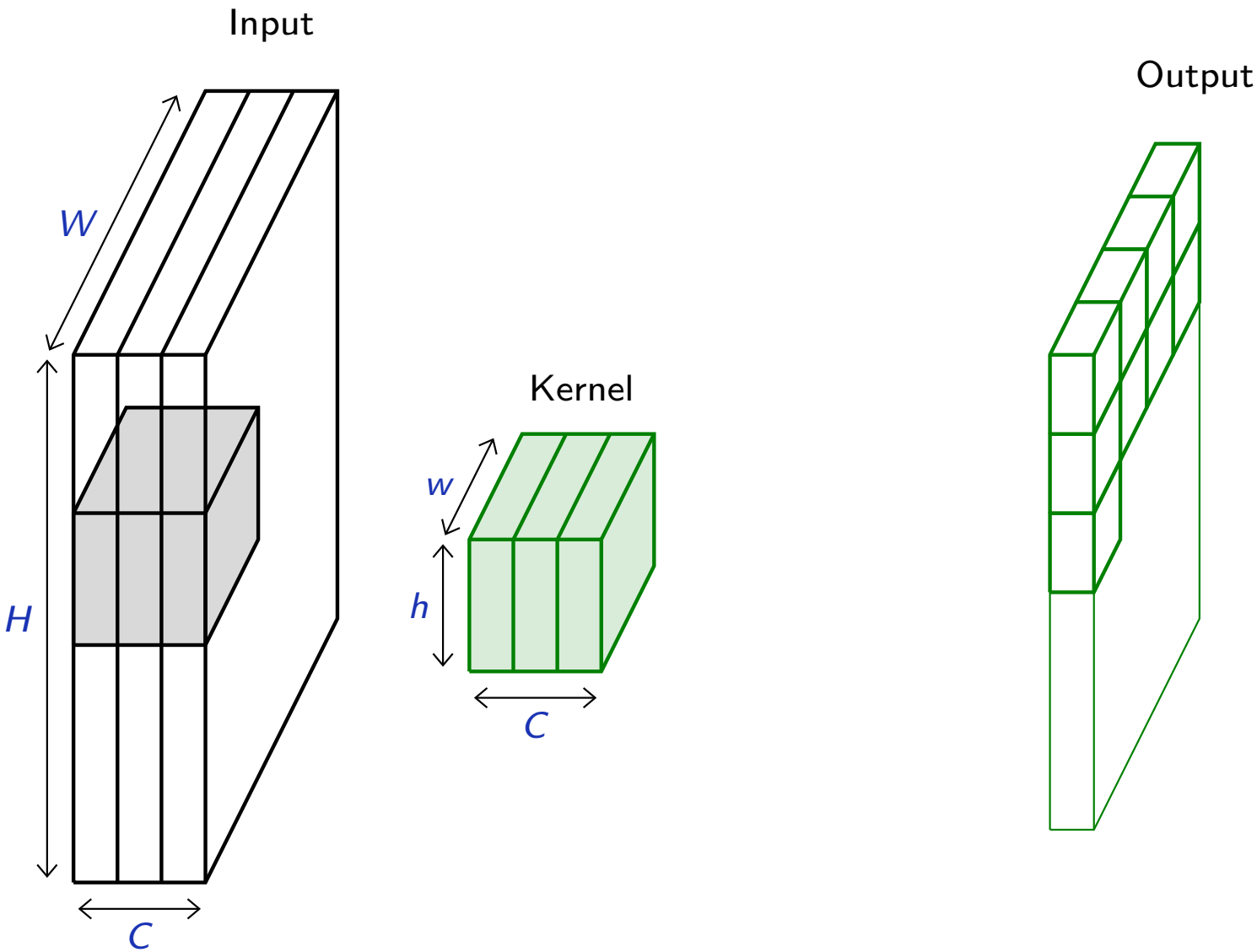

Input
W
H
C
Kernel
w
h
C
Output

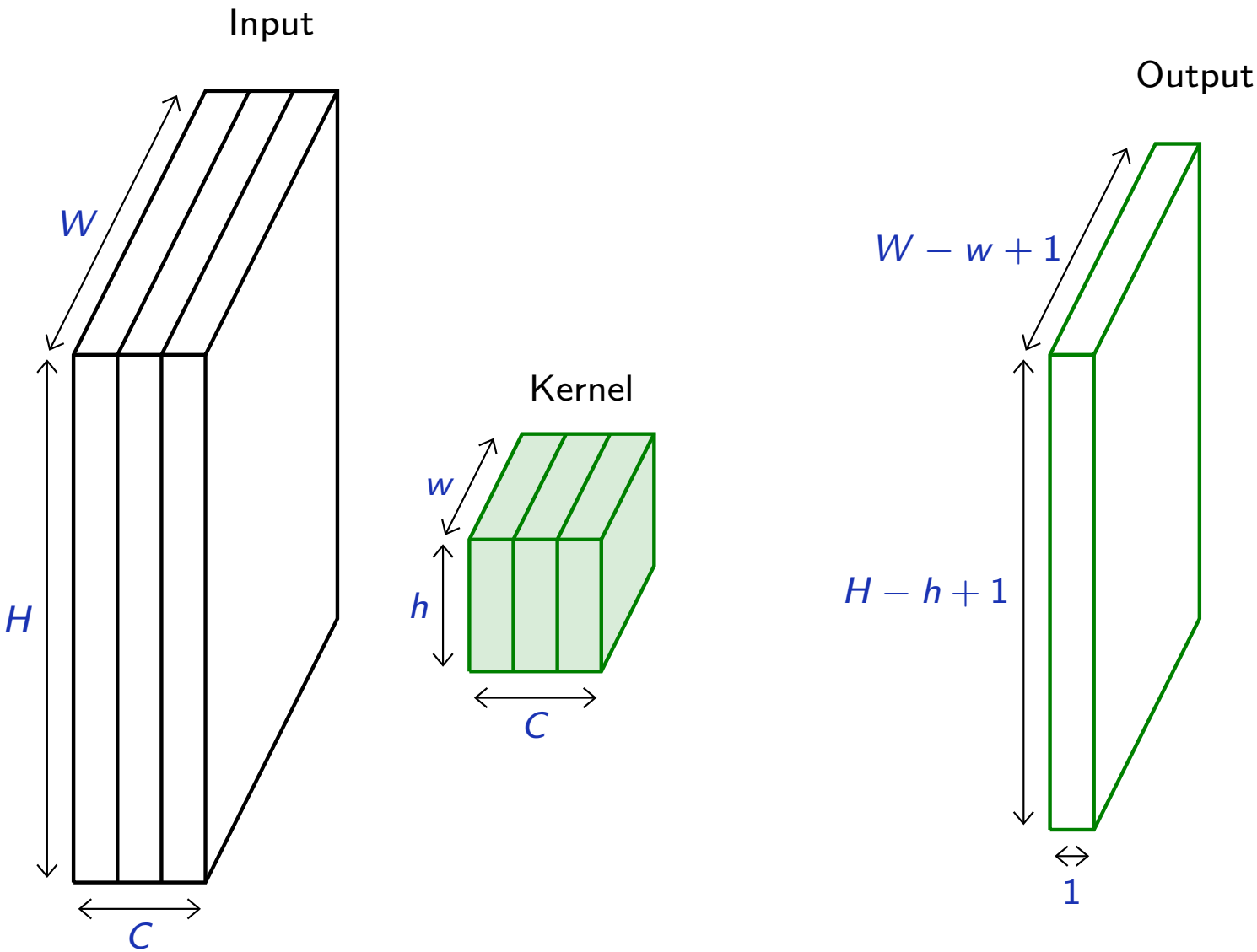

Input
W
H
C
Kernel
w
h
C
Output

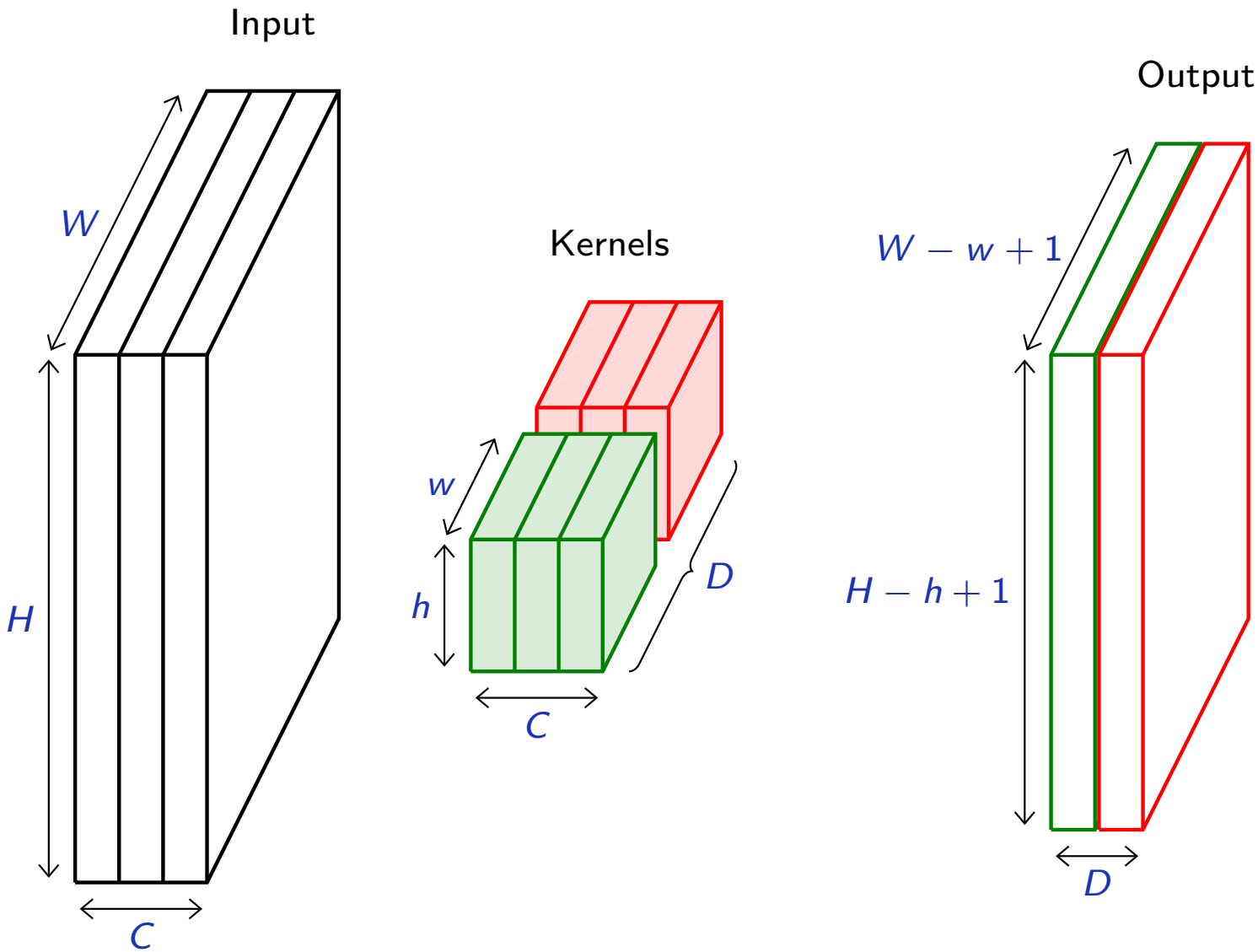

Input
W
H
C
Kernel
w
h
C
Output

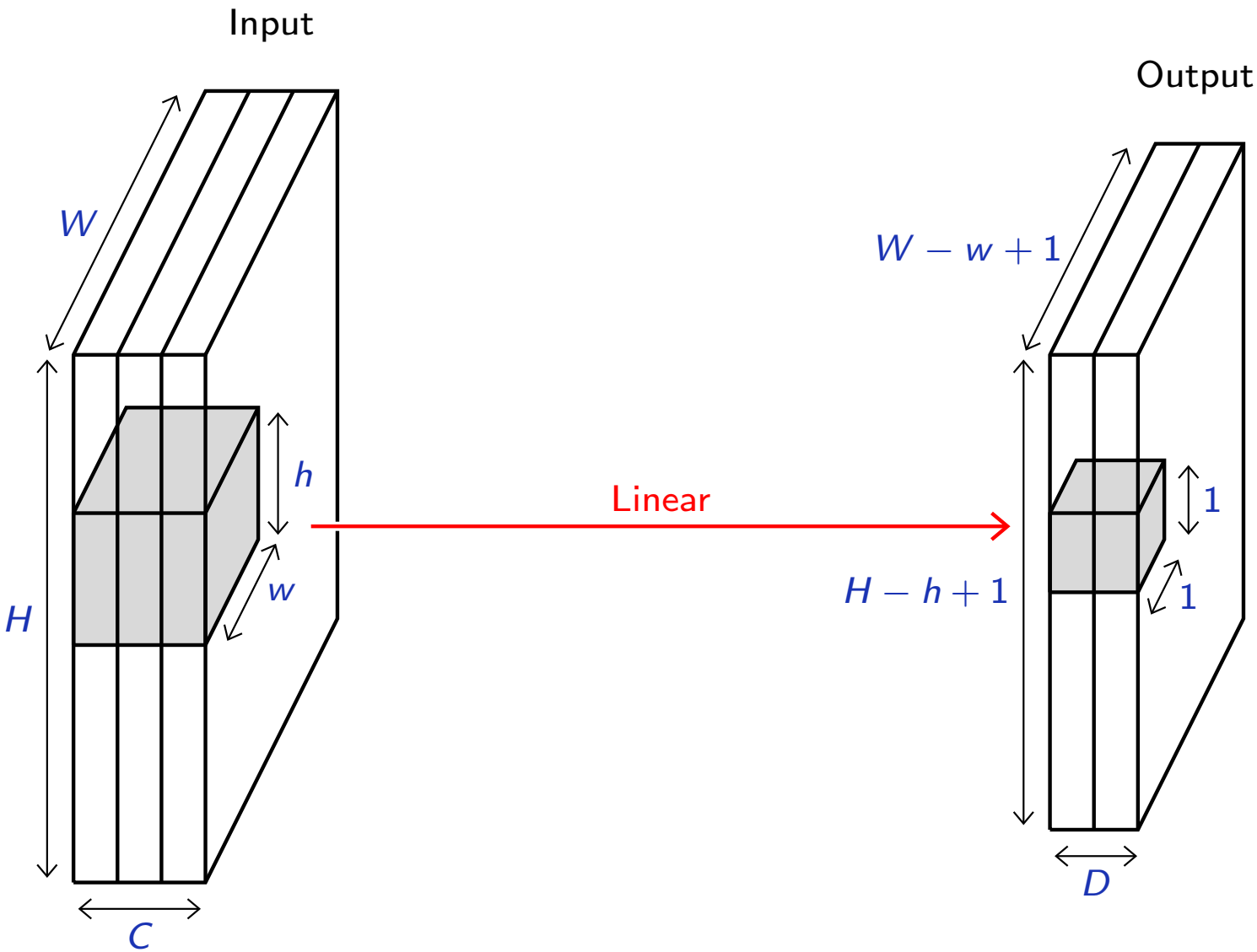

Input
W
H
C
Kernel
w
h
C
Output


Input
W
H
C
Kernel
w
h
C
Output


Input
W
H
C
Kernel
w
h
C
Output


Input
W
H
C
Kernel
w
h
C
Output
W − w + 1
H − h + 1
1


Input
W
H
C
Kernels
w
h
D
C
Output
W − w + 1
H − h + 1
D


Input
Output
W
H
h
w
C
Linear
W − w + 1
H − h + 1
1
1
D

Note that a convolution **preserves the signal support structure**.

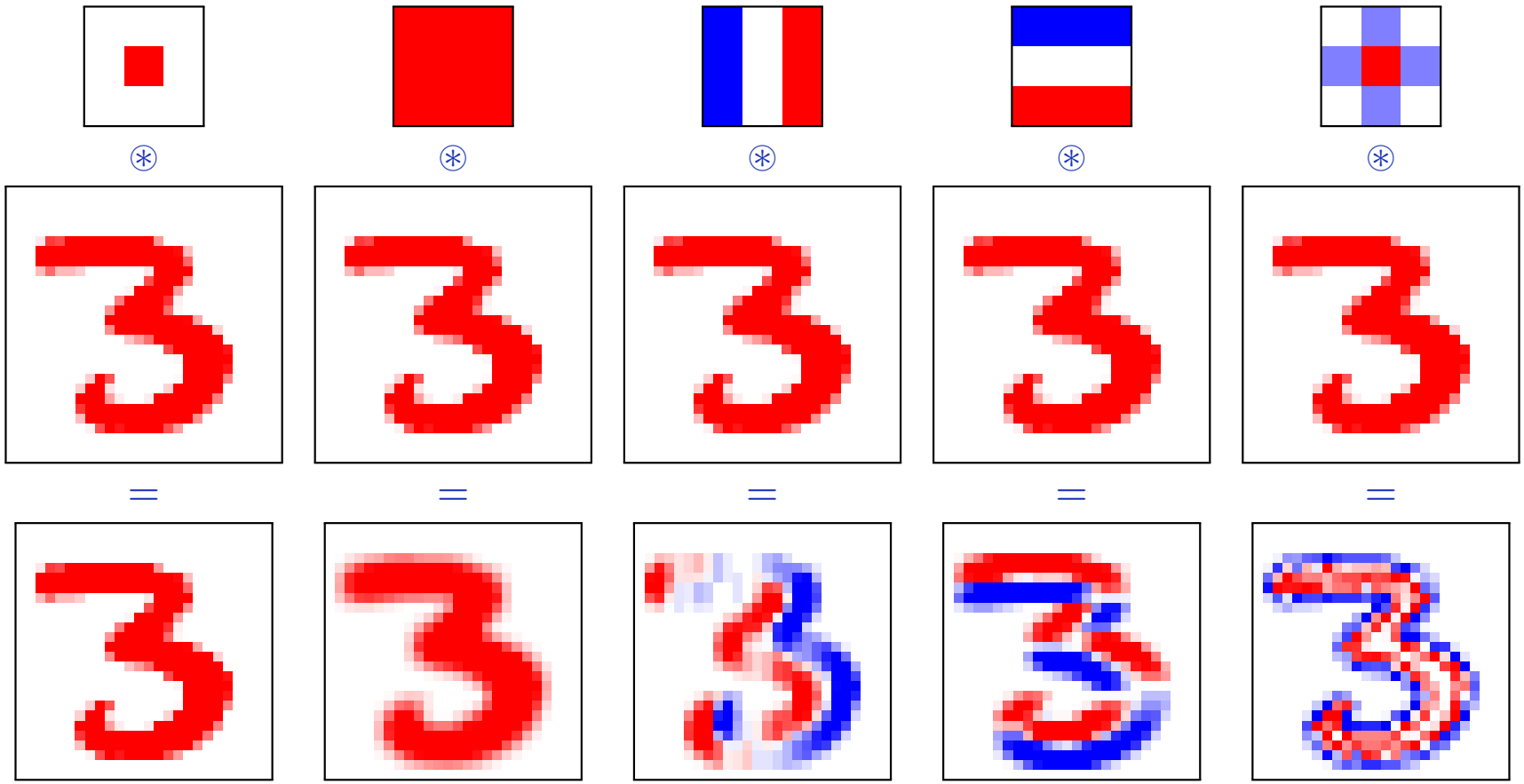
A 1d signal is converted into a 1d signal, a 2d signal into a 2d, and neighboring parts of the input signal influence neighboring parts of the output signal.

A 3d convolution can be used if the channel index has some metric meaning, such as time for a series of grayscale video frames. Otherwise swiping across channels makes no sense.

We usually refer to one of the channels generated by a convolution layer as an **activation map.**

The sub-area of an input map that influences a component of the output as the **receptive field** of the latter.

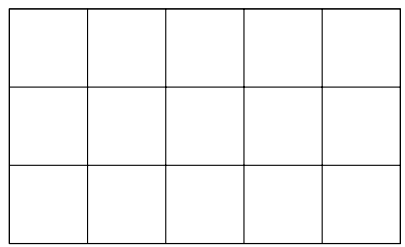
In the context of convolutional networks, a standard linear layer is called a **fully connected layer** since every input influences every output.



Convolutions have two additional standard parameters:

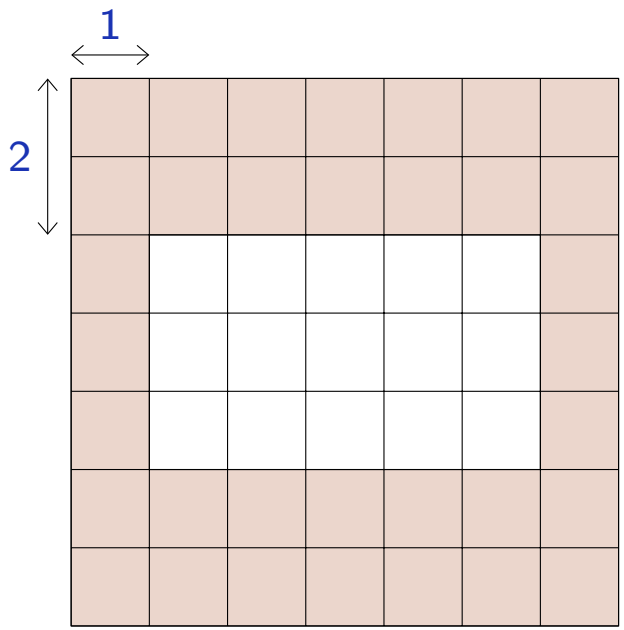
- The **padding** specifies the size of a zeroed frame added around the input,
- the **stride** specifies a step size when moving the kernel across the signal.

Here with $C \times 3 \times 5$ as input



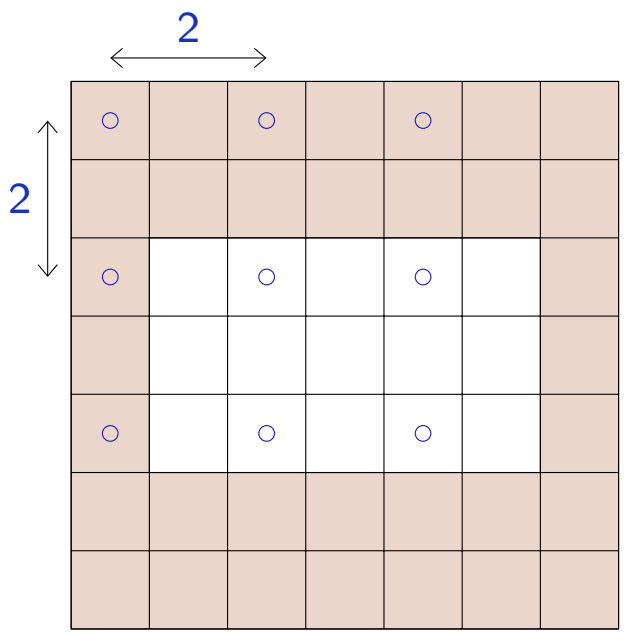
Input

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$



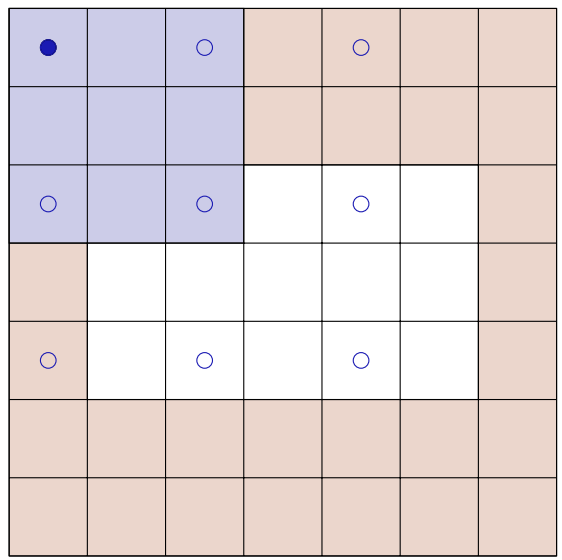
Input

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$

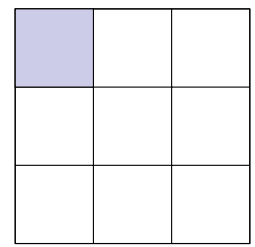


Input

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$, and a kernel of size $C \times 3 \times 3$

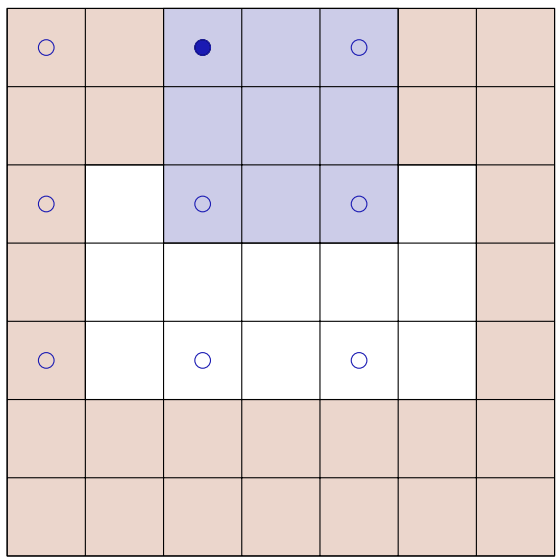


Input

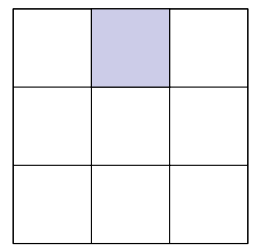


Output

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$, and a kernel of size $C \times 3 \times 3$

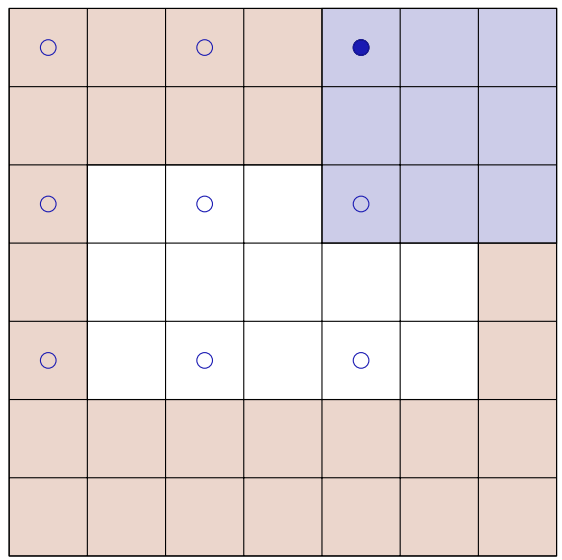


Input

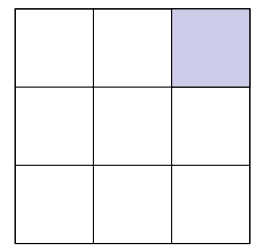


Output

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$, and a kernel of size $C \times 3 \times 3$

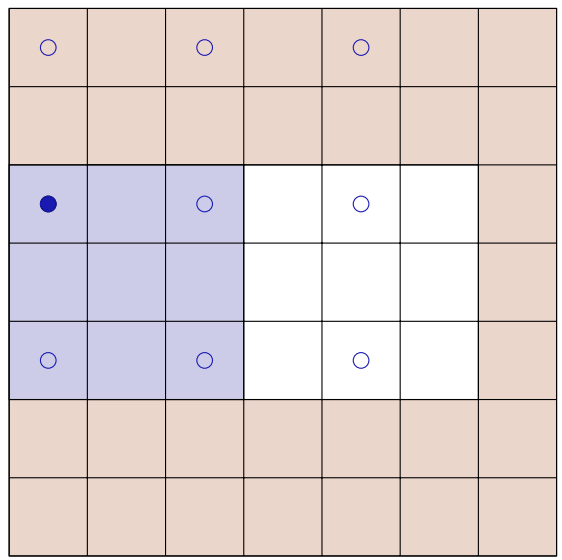


Input

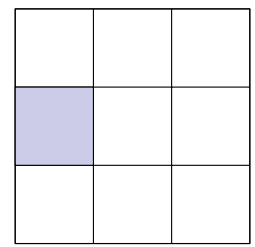


Output

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$, and a kernel of size $C \times 3 \times 3$

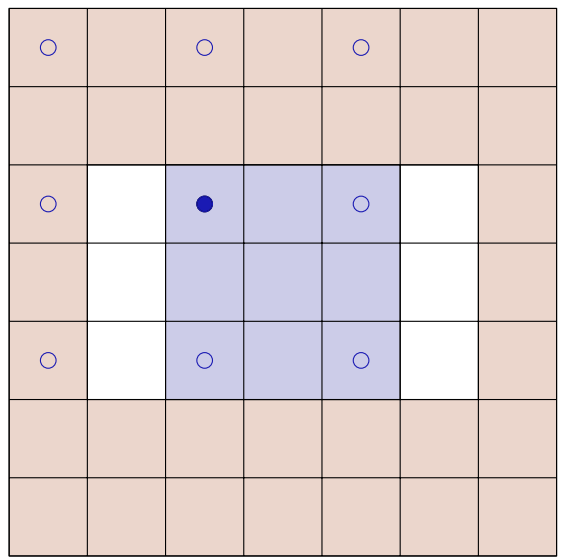


Input

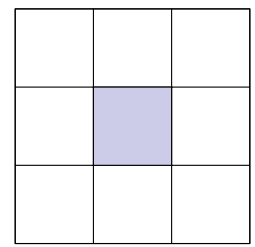


Output

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$, and a kernel of size $C \times 3 \times 3$

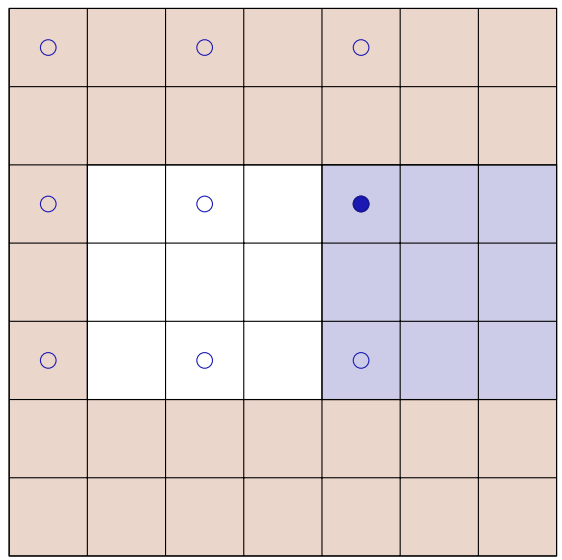


Input

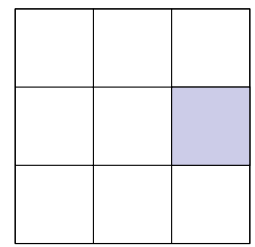


Output

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$, and a kernel of size $C \times 3 \times 3$

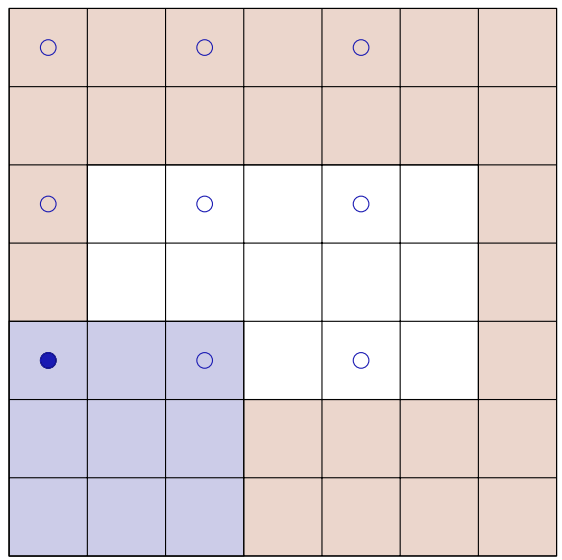


Input

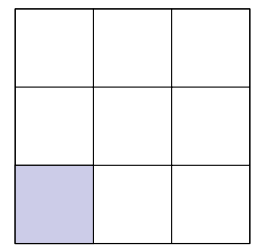


Output

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$, and a kernel of size $C \times 3 \times 3$

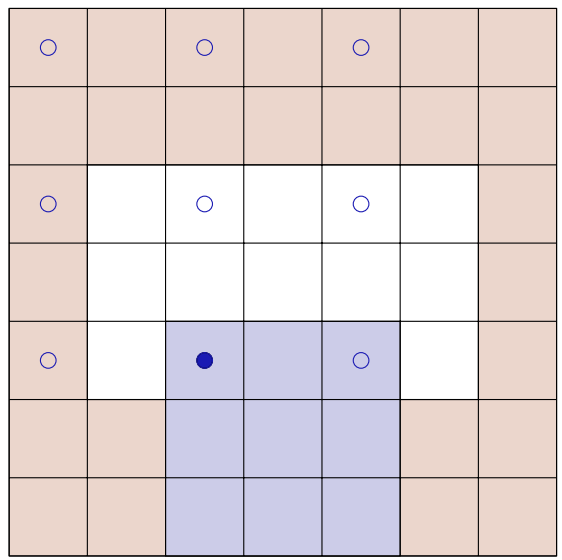


Input

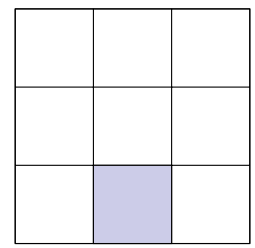


Output

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$, and a kernel of size $C \times 3 \times 3$

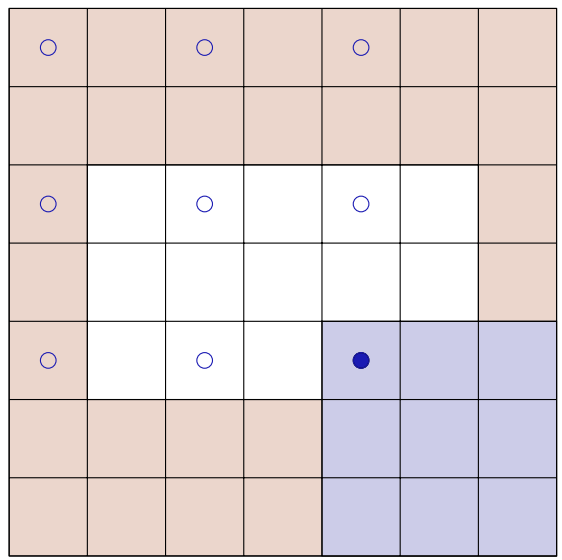


Input

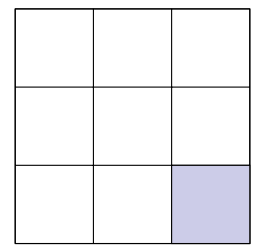


Output

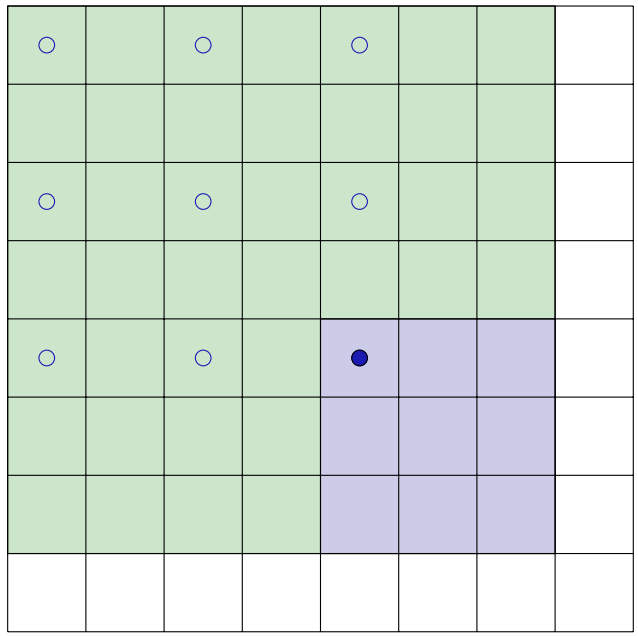
Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$, and a kernel of size $C \times 3 \times 3$, the output is $1 \times 3 \times 3$.



Input



Output

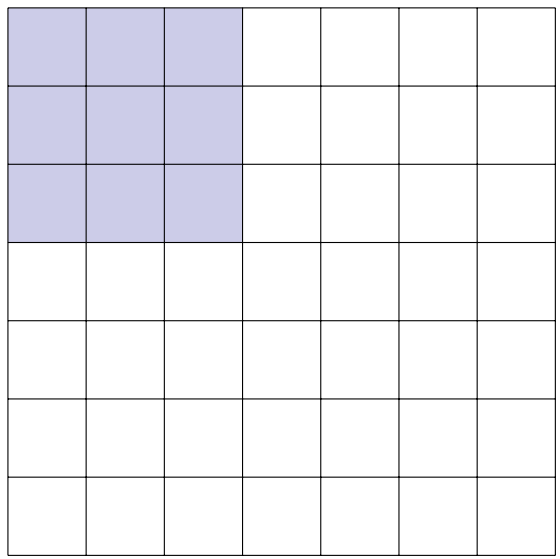


A convolution with a stride greater than 1 may not cover the input map completely, hence may ignore some of the input values.

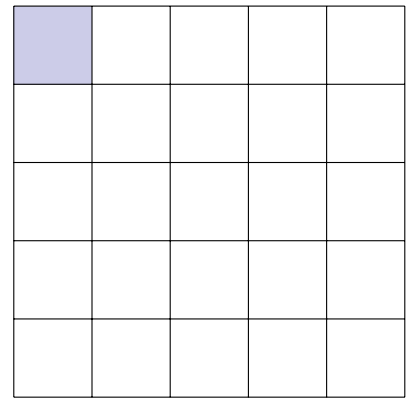
Convolution operations admit one more standard parameter that we have not discussed yet: The dilation, which modulates the expansion of the filter support (Yu and Koltun, 2015).

It is 1 for standard convolutions, but can be greater, in which case the resulting operation can be envisioned as a convolution with a regularly sparsified filter.

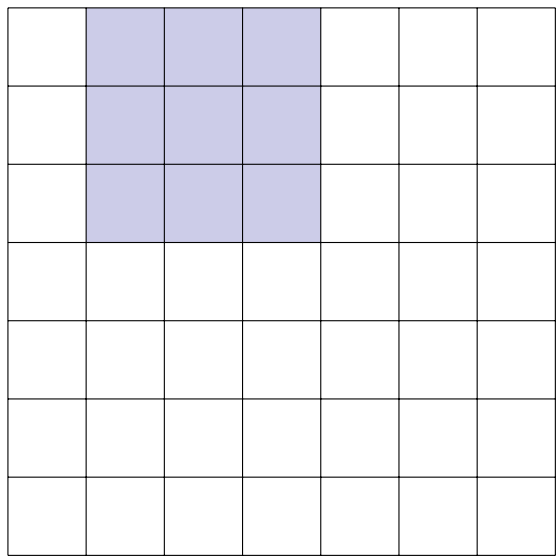
This notion comes from signal processing, where it is referred to as *algorithme à trous*, hence the term sometime used of "convolution à trous".



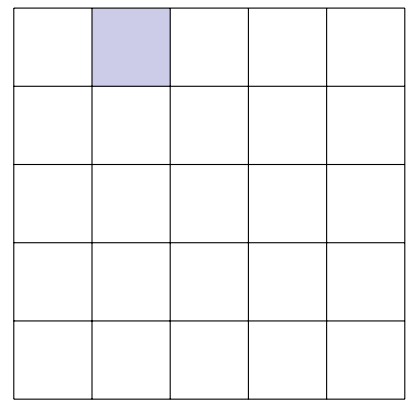
Input



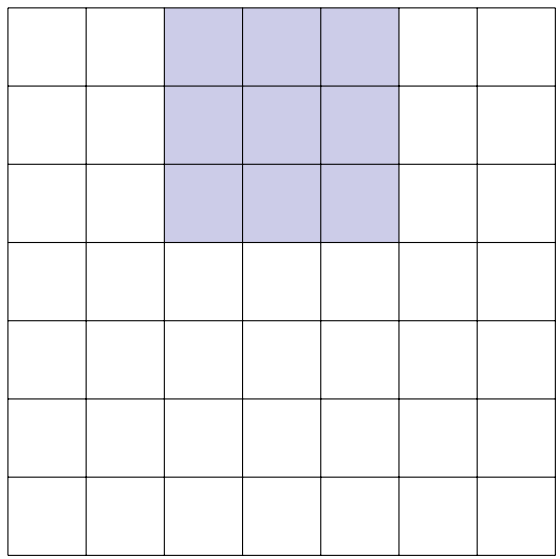
Output



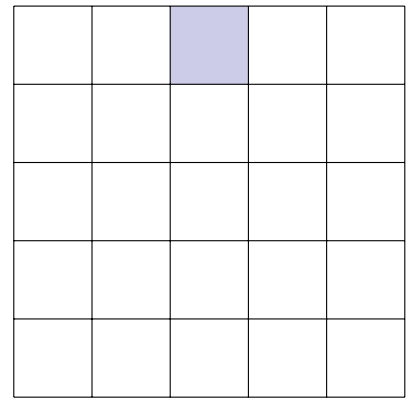
Input



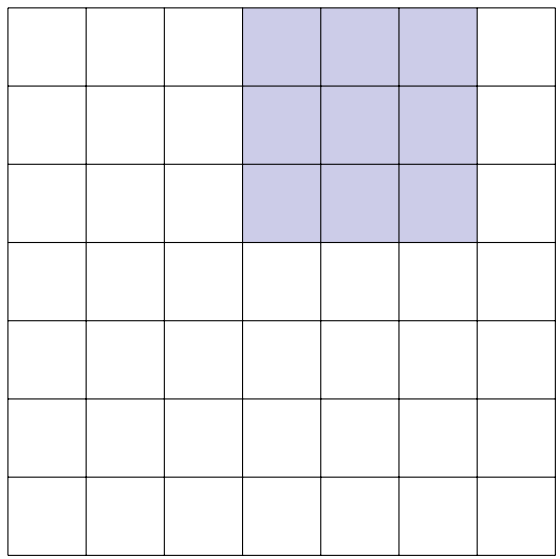
Output



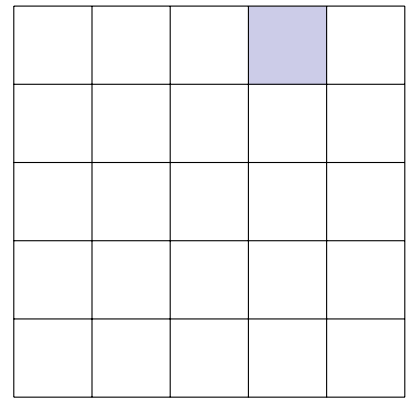
Input



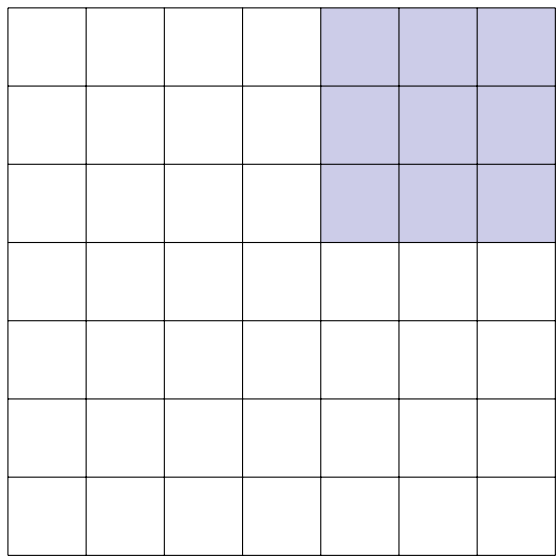
Output



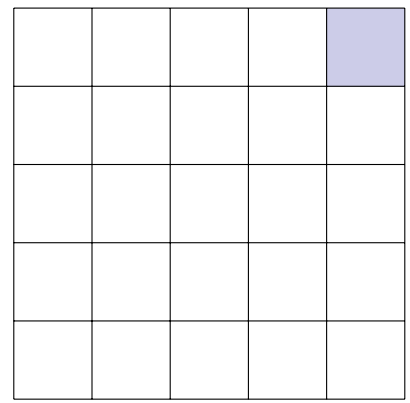
Input



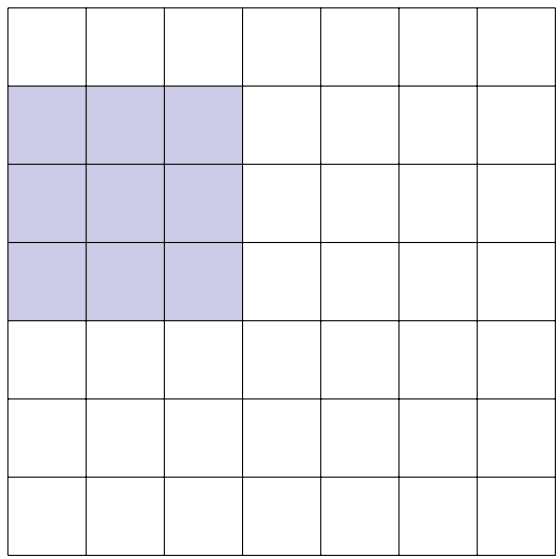
Output



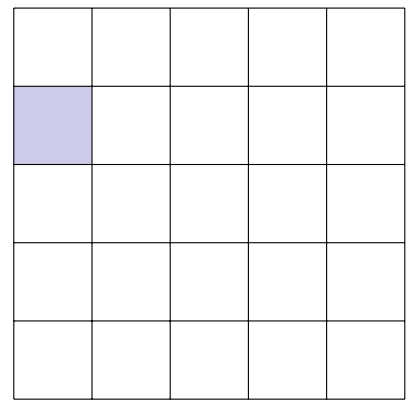
Input



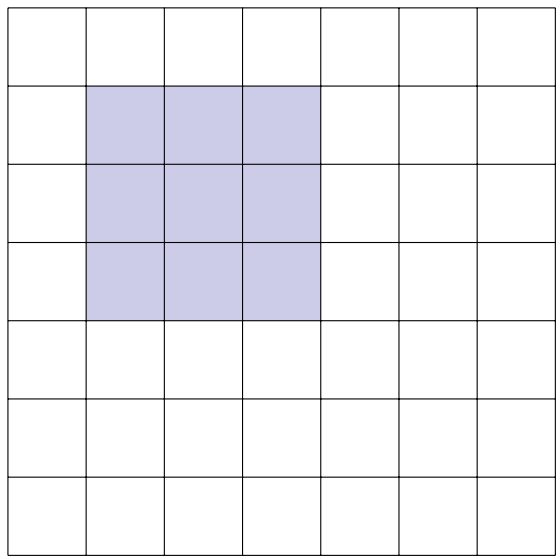
Output



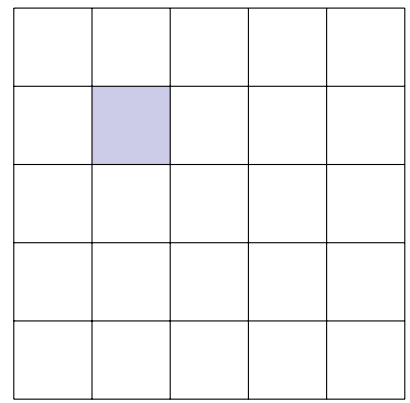
Input



Output

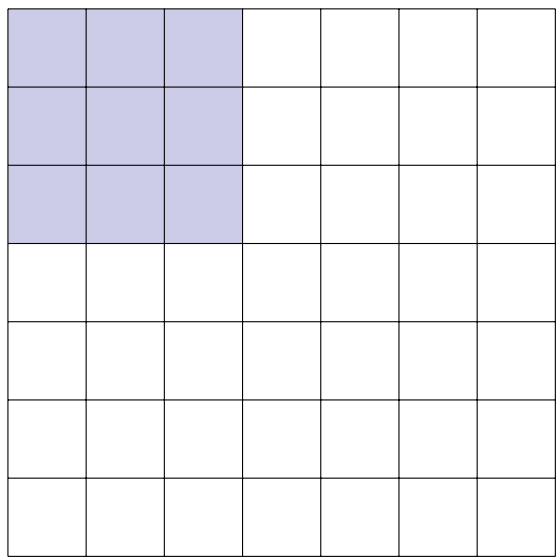


Input

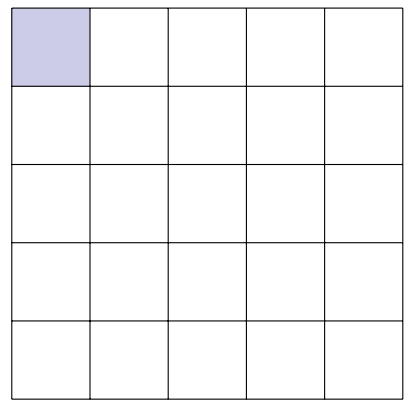


Output

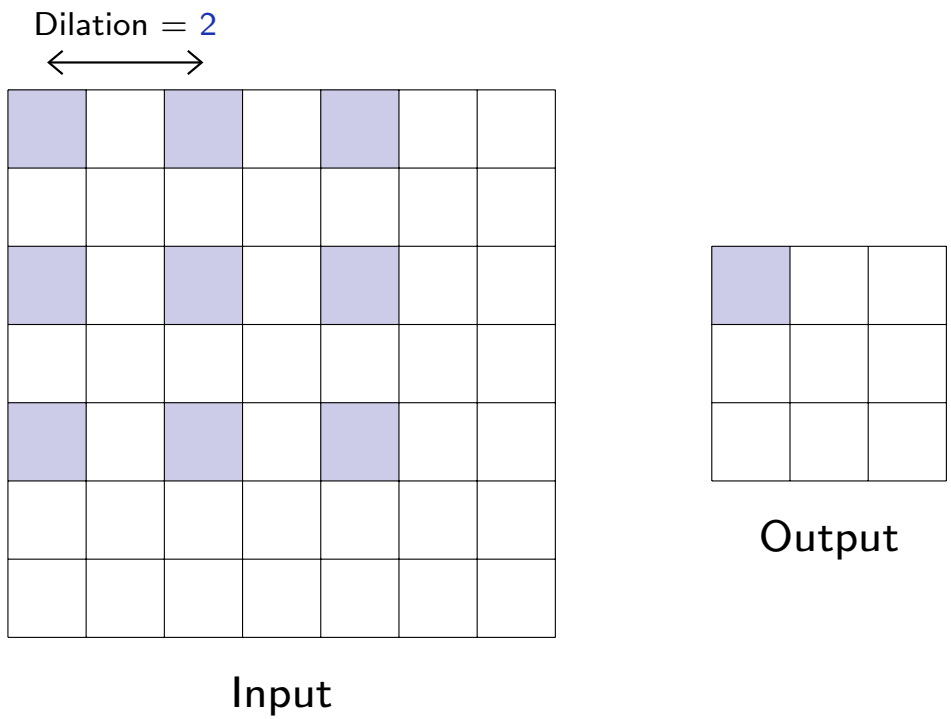
Dilation $= 1$

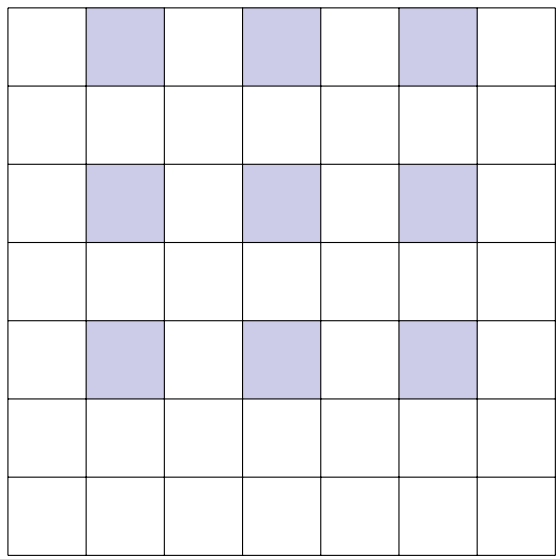


Input

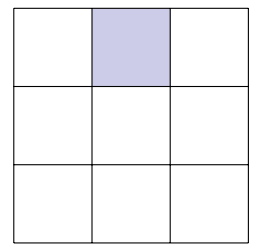


Output

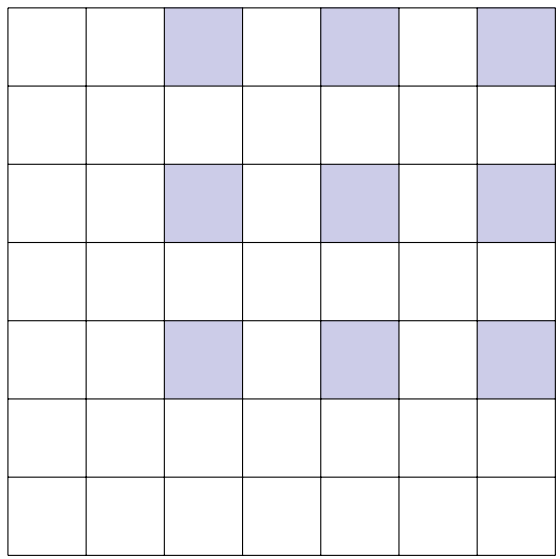

Dilation = 2
Input
Output



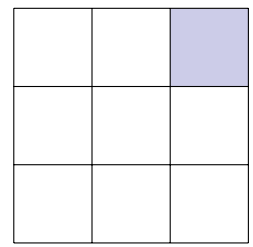
Input



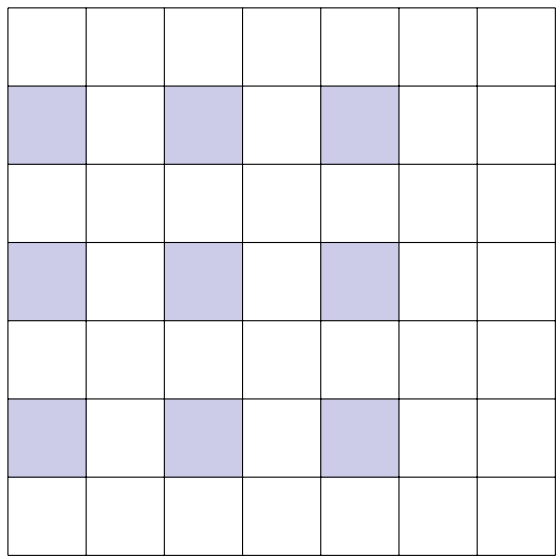
Output



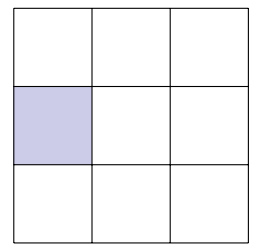
Input



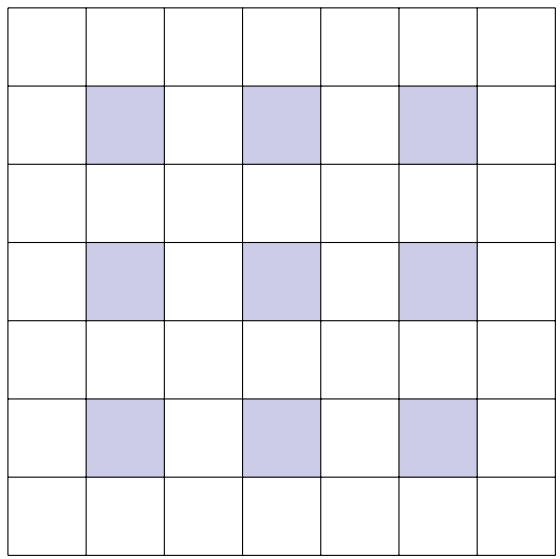
Output



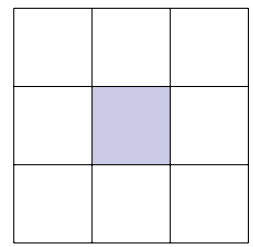
Input



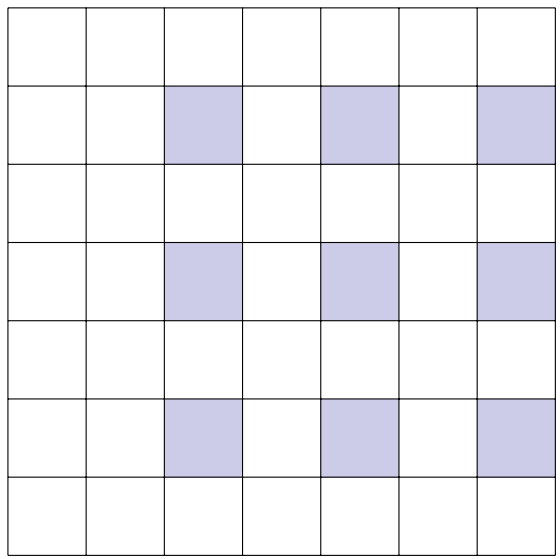
Output



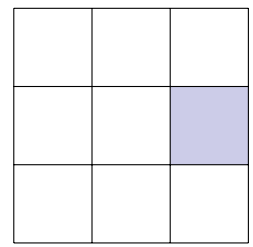
Input



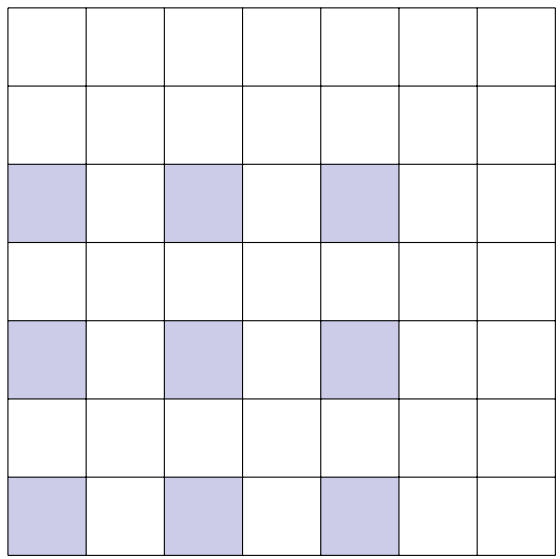
Output



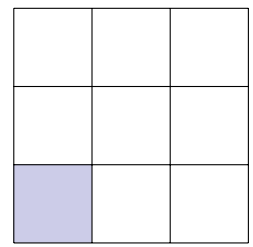
Input



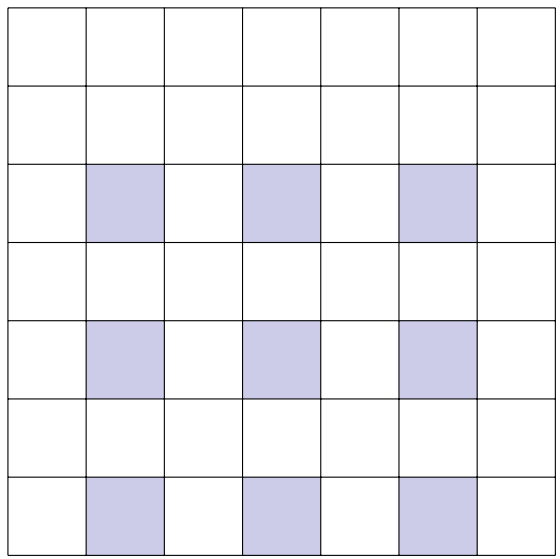
Output



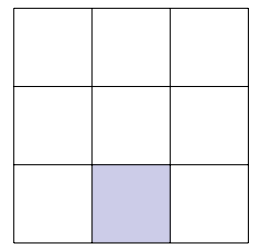
Input



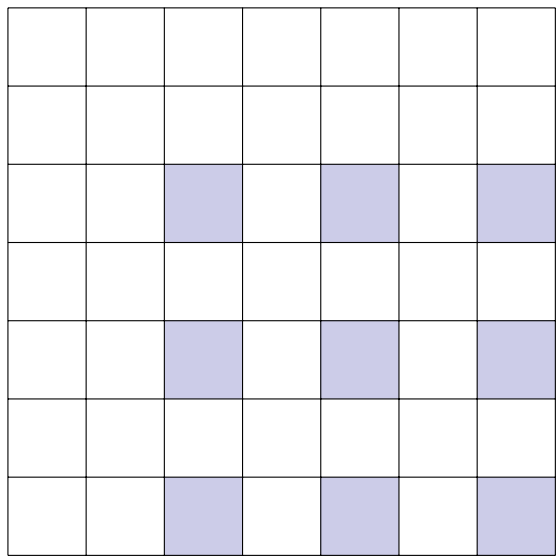
Output



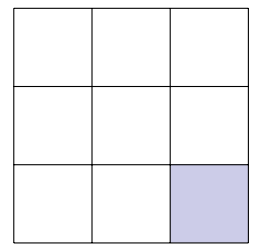
Input



Output



Input



Output

Having a dilation greater than one increases the units' receptive field size without increasing the number of parameters.

Convolutions with stride or dilation strictly greater than one reduce the activation map size, for instance to make a final classification decision.

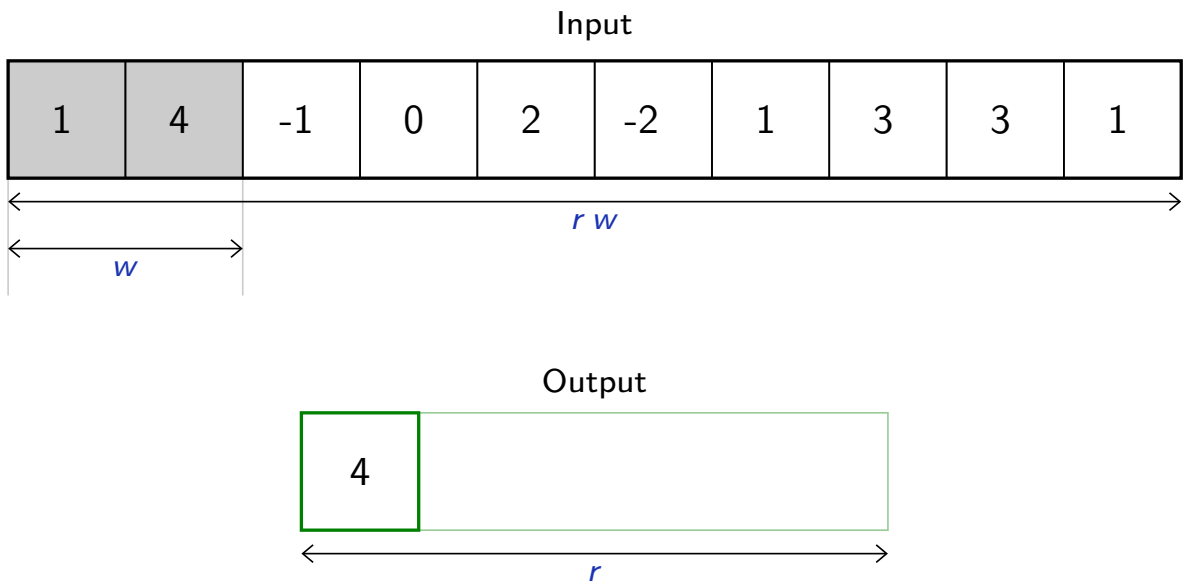
Such convolutions allows to build networks without pooling layers.

The historical approach to compute a low-dimension signal (*e.g.* a few scores) from a high-dimension one (*e.g.* an image) was to use **pooling** operations.

Such an operation aims at grouping several activations into a single "more meaningful" one.

The most standard type of pooling is the **max-pooling**, which computes max values over non-overlapping blocks.

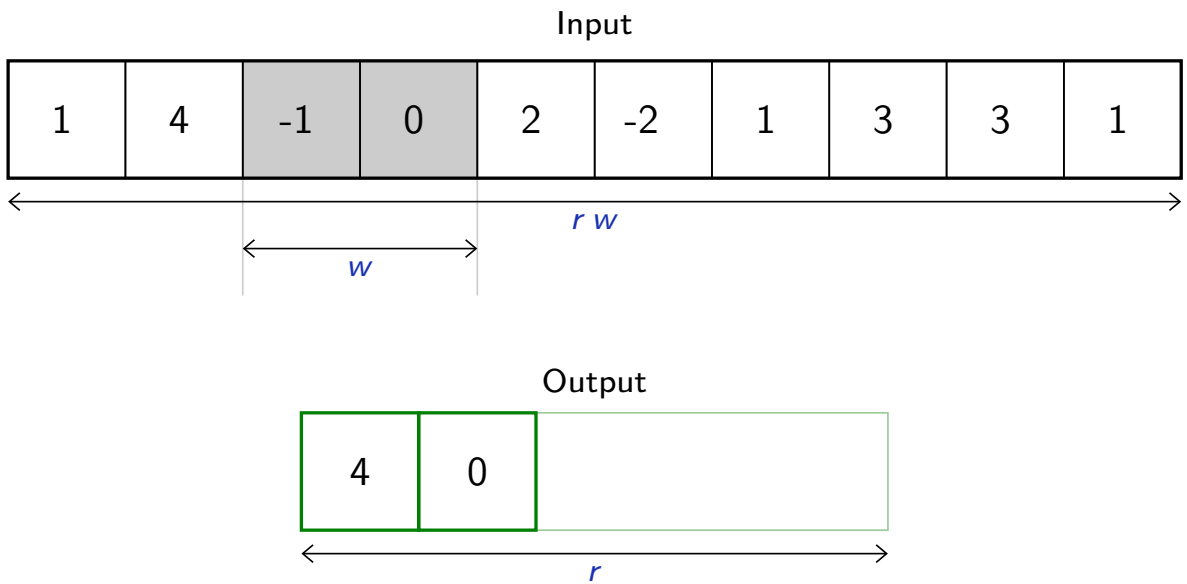
For instance in 1d with a kernel of size 2:



The **average pooling** computes average values per block instead of max values.

The most standard type of pooling is the **max-pooling**, which computes max values over non-overlapping blocks.

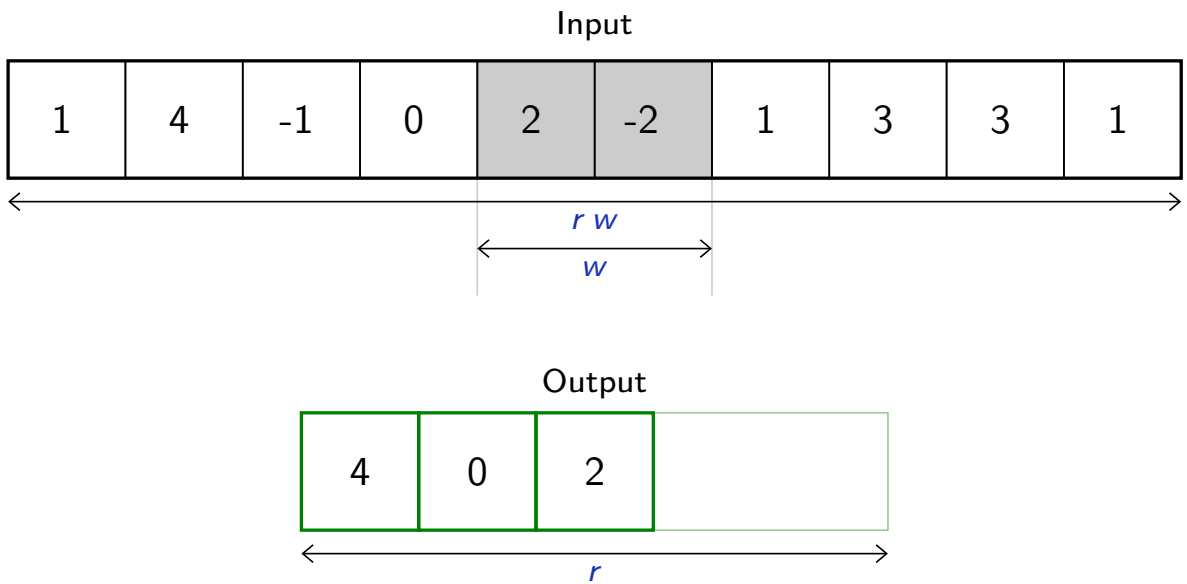
For instance in 1d with a kernel of size 2:



The **average pooling** computes average values per block instead of max values.

The most standard type of pooling is the **max-pooling**, which computes max values over non-overlapping blocks.

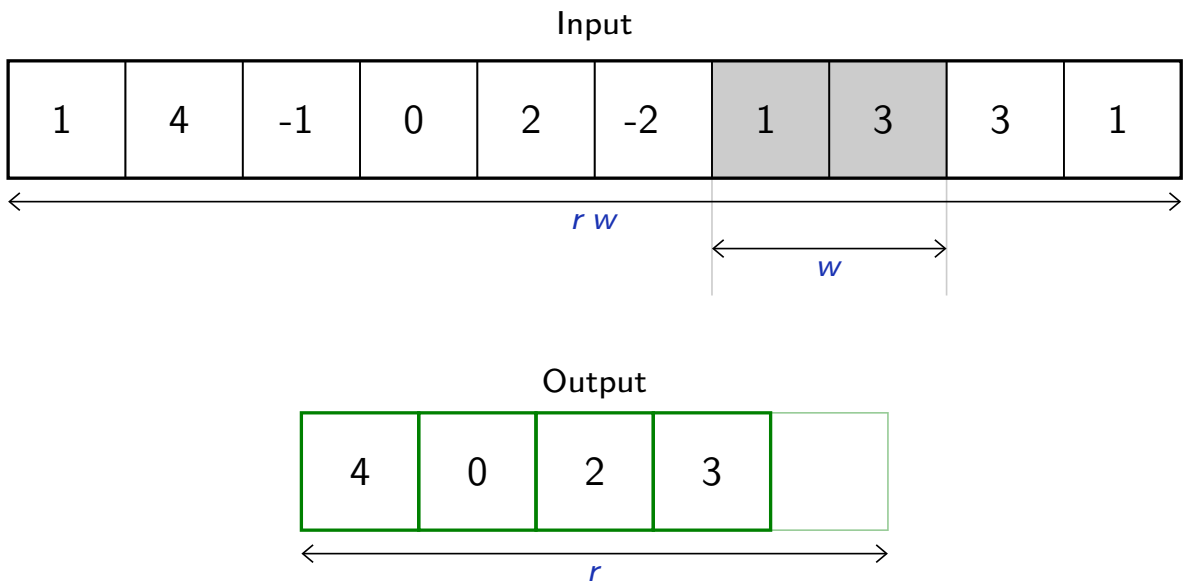
For instance in 1d with a kernel of size 2:



The **average pooling** computes average values per block instead of max values.

The most standard type of pooling is the **max-pooling**, which computes max values over non-overlapping blocks.

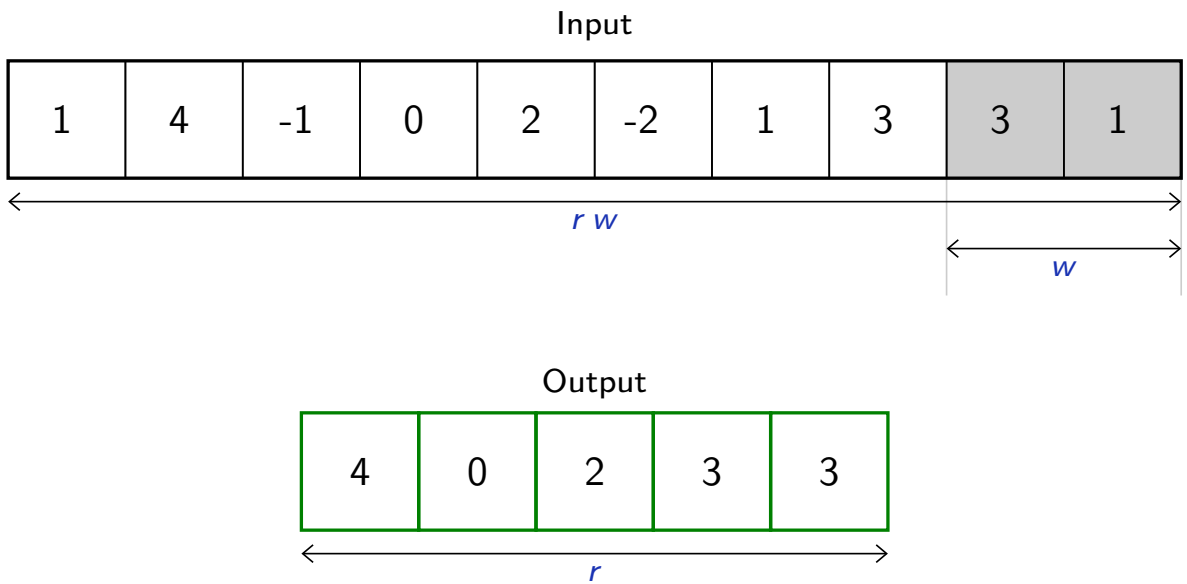
For instance in 1d with a kernel of size 2:



The **average pooling** computes average values per block instead of max values.

The most standard type of pooling is the **max-pooling**, which computes max values over non-overlapping blocks.

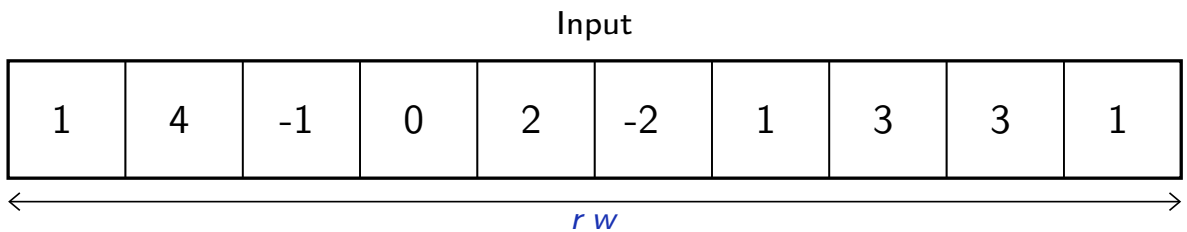
For instance in 1d with a kernel of size 2:

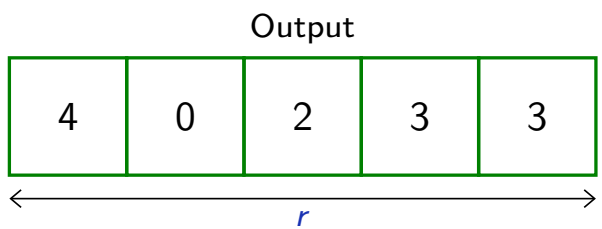


The **average pooling** computes average values per block instead of max values.

The most standard type of pooling is the **max-pooling**, which computes max values over non-overlapping blocks.

For instance in 1d with a kernel of size 2:

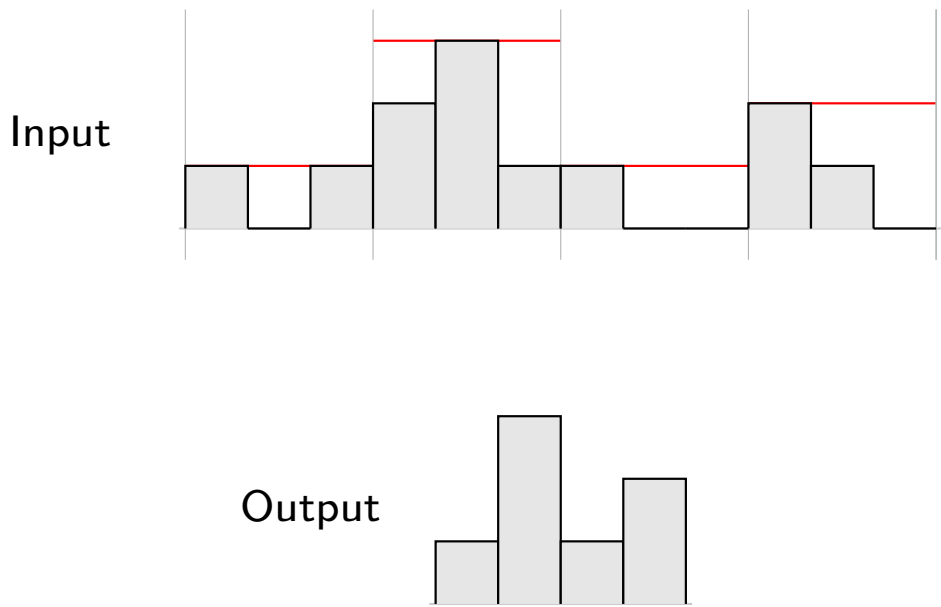




The **average pooling** computes average values per block instead of max values.

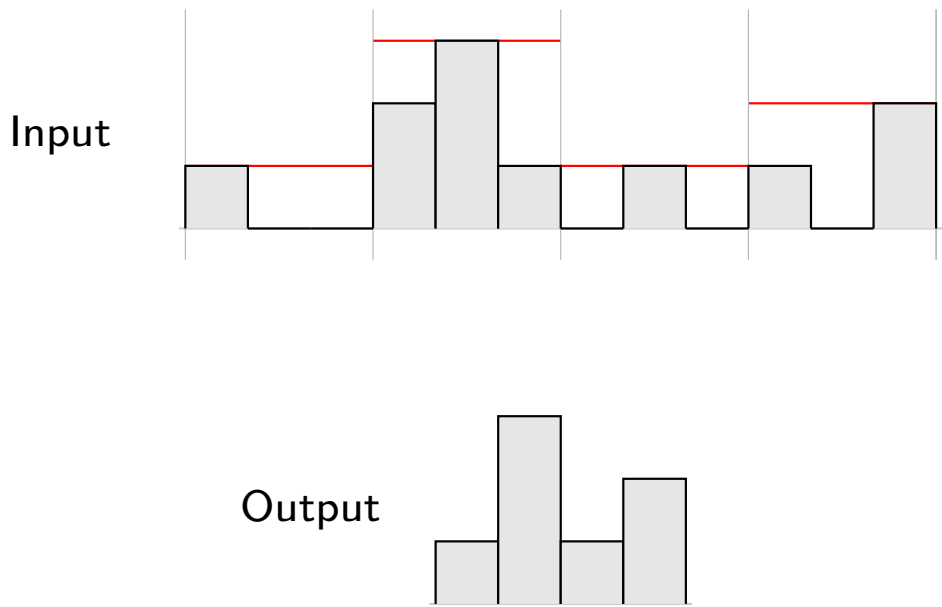
Pooling provides invariance to any permutation inside one of the cell.

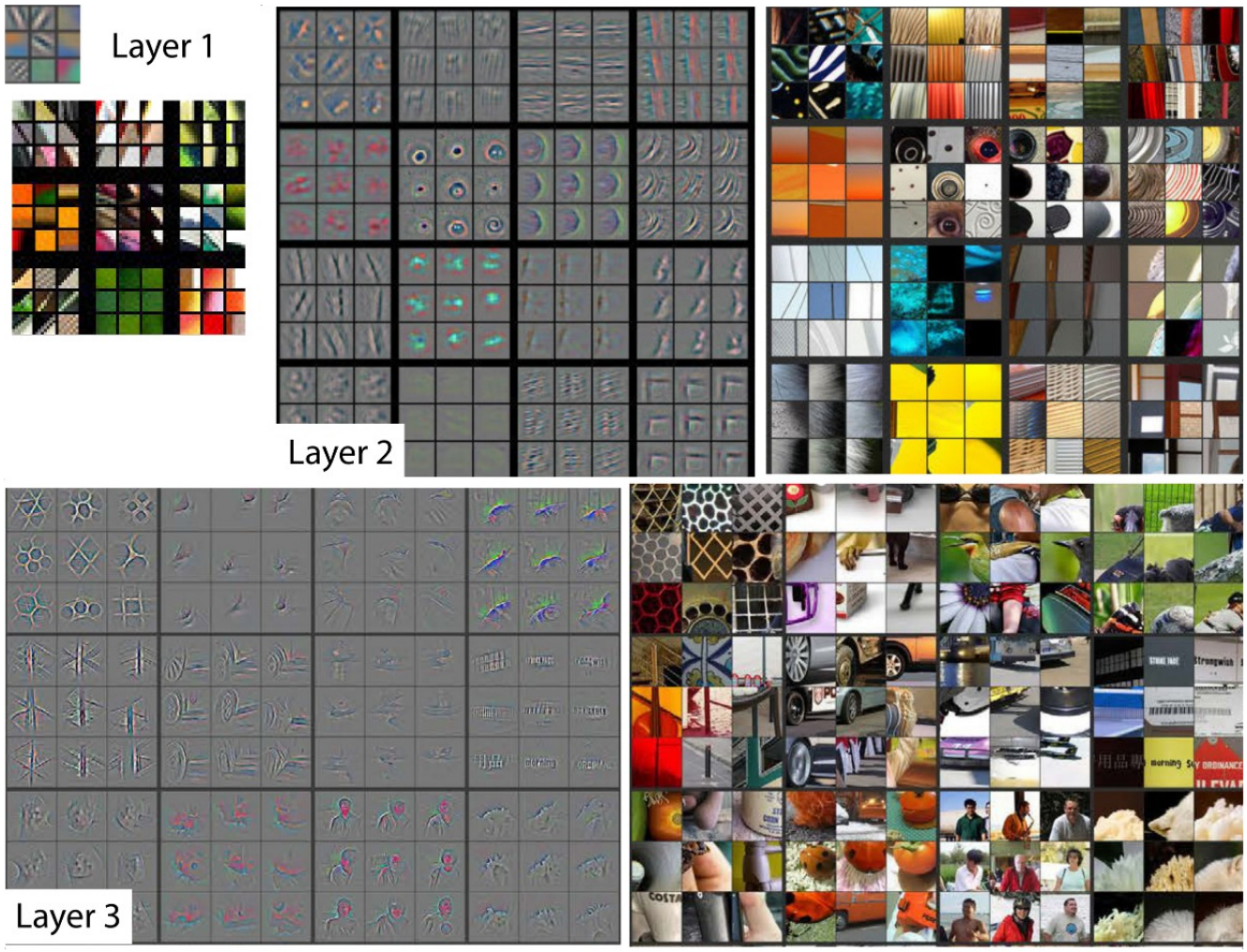
More practically, it provides a pseudo-invariance to deformations that result into local translations.



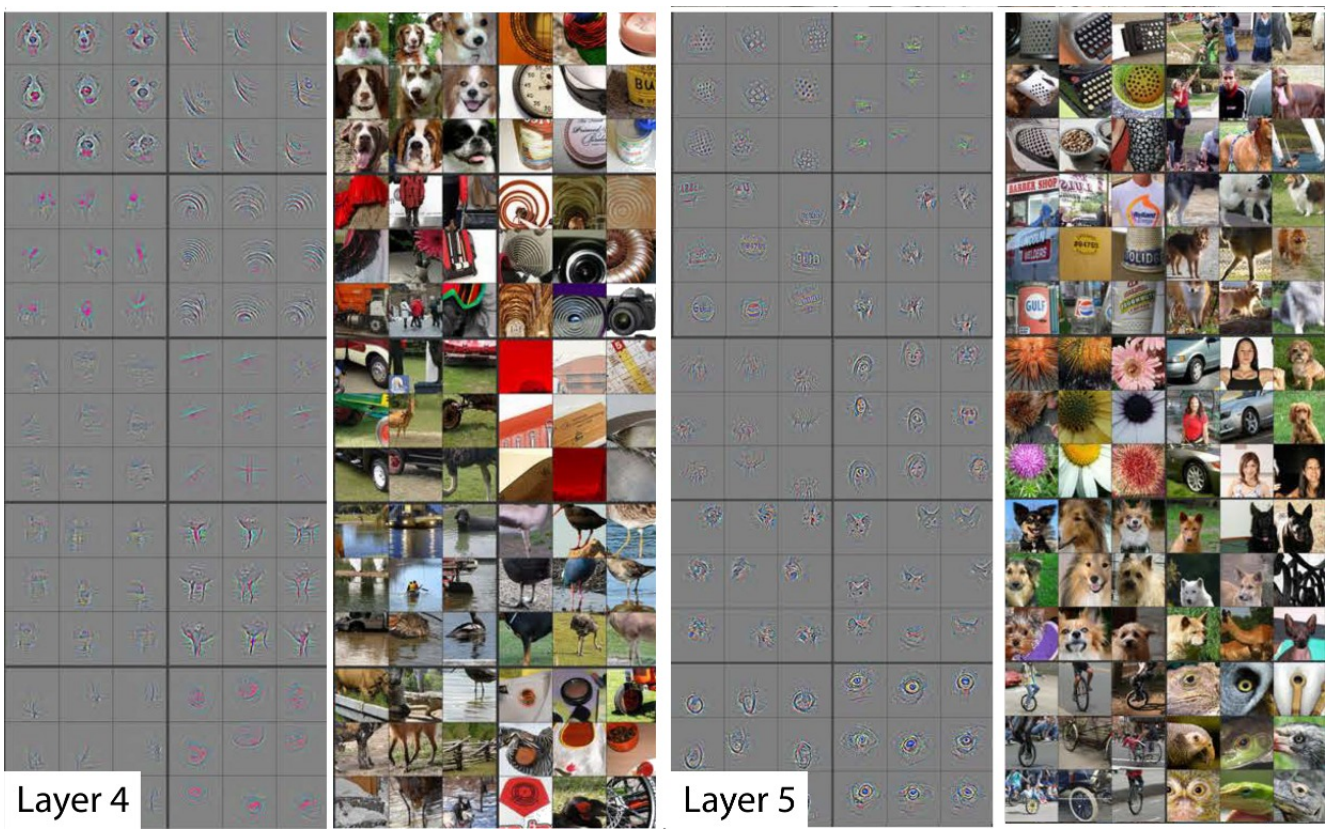
Pooling provides invariance to any permutation inside one of the cell.

More practically, it provides a pseudo-invariance to deformations that result into local translations.

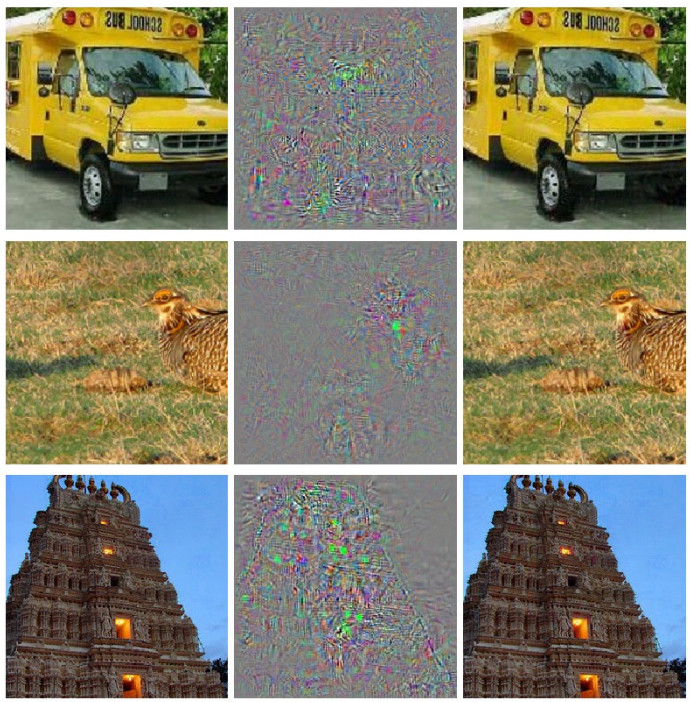


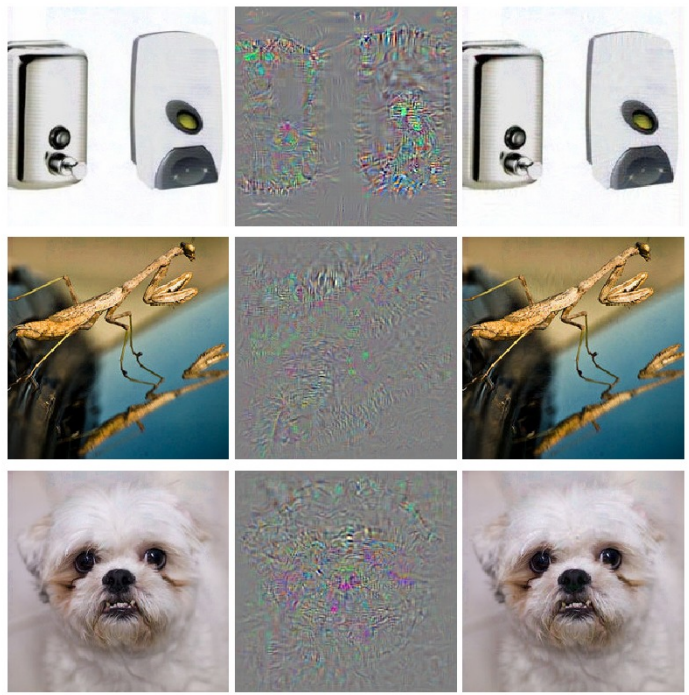


(Zeiler and Fergus, 2014)

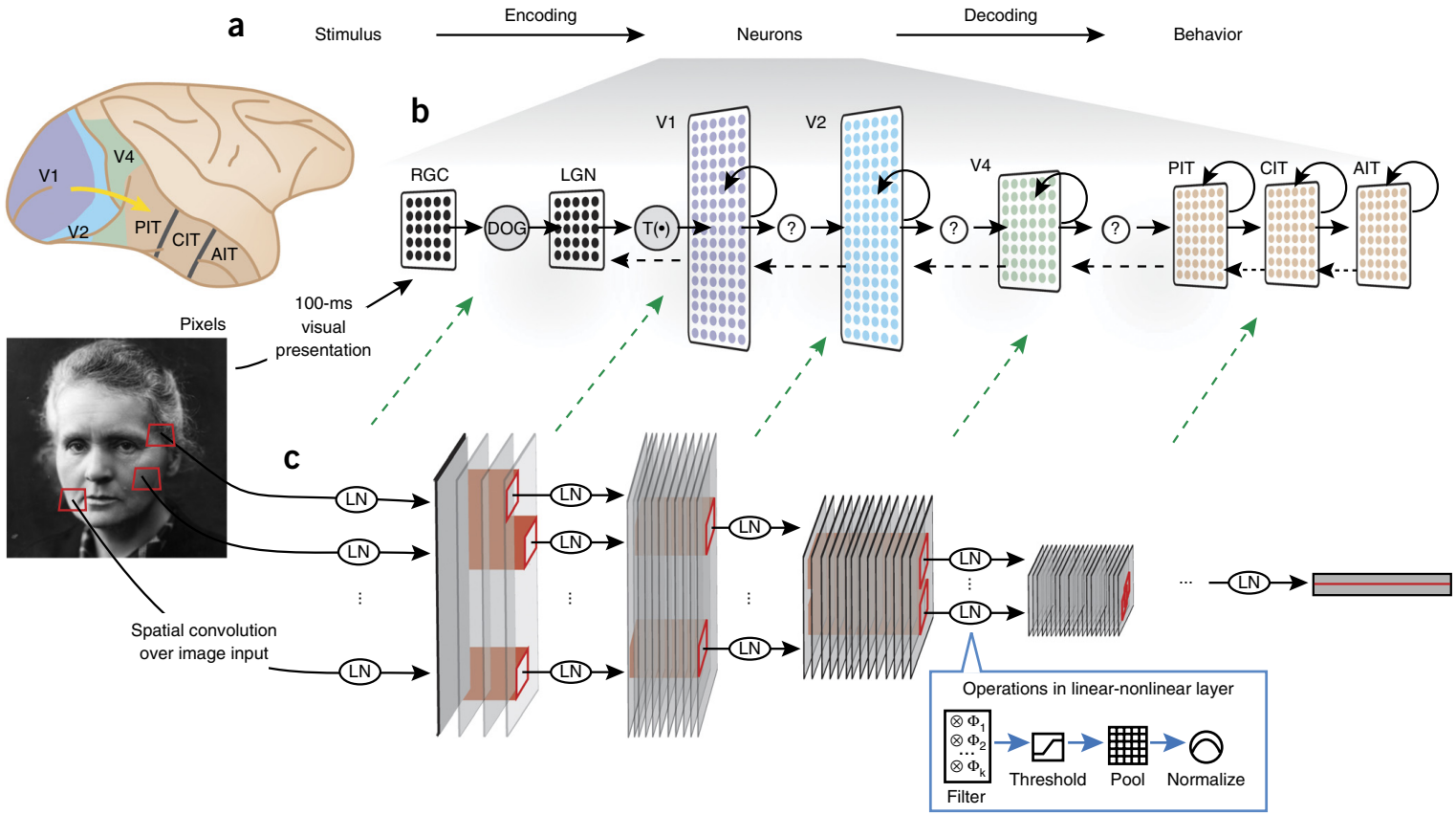


(Zeiler and Fergus, 2014)





(Szegedy et al., 2014)



(Yamins and DiCarlo, 2016)

Gradient Vanishing & Residual Neural Networks

Consider the gradient estimation for a standard MLP:

Forward pass

$$\forall n,\ x^{(0)} = x, \quad \forall l = 1, \dots, L, \ \begin{cases} s^{(l)} = w^{(l)} x^{(l-1)} + b^{(l)} \\ x^{(l)} = \sigma\left(s^{(l)}\right) \end{cases}$$

Backward pass

$$\begin{cases} \left[\frac{\partial \ell}{\partial x^{(L)}}\right] = \nabla_1 \ell\left(x^{(L)}\right) \\ \text{if } l < L, \left[\frac{\partial \ell}{\partial x^{(l)}}\right] = \left(w^{(l+1)}\right)^T \left[\frac{\partial \ell}{\partial s^{(l+1)}}\right] \end{cases}$$

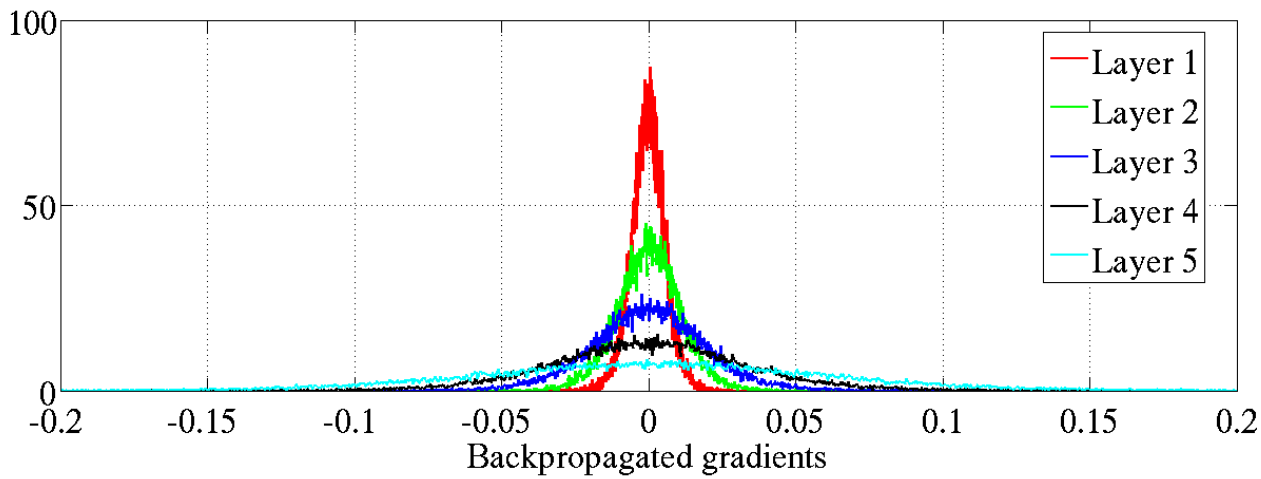
$$\left[\frac{\partial \ell}{\partial s^{(l)}}\right] = \left[\frac{\partial \ell}{\partial x^{(l)}}\right] \odot \sigma'\left(s^{(l)}\right)$$

$$\left[\!\left[\frac{\partial \ell}{\partial w^{(l)}}\right]\!\right] = \left[\frac{\partial \ell}{\partial s^{(l)}}\right] \left(x^{(l-1)}\right)^T \qquad \left[\frac{\partial \ell}{\partial b^{(l)}}\right] = \left[\frac{\partial \ell}{\partial s^{(l)}}\right].$$

We have

$$\left[\frac{\partial \ell}{\partial x^{(l)}}\right] = \left(w^{(l+1)}\right)^T \left(\sigma'\left(s^{(l)}\right) \odot \left[\frac{\partial \ell}{\partial x^{(l+1)}}\right]\right).$$

so the gradient "vanishes" exponentially with the depth if the ws are ill-conditioned or the activations are in the saturating domain of σ.

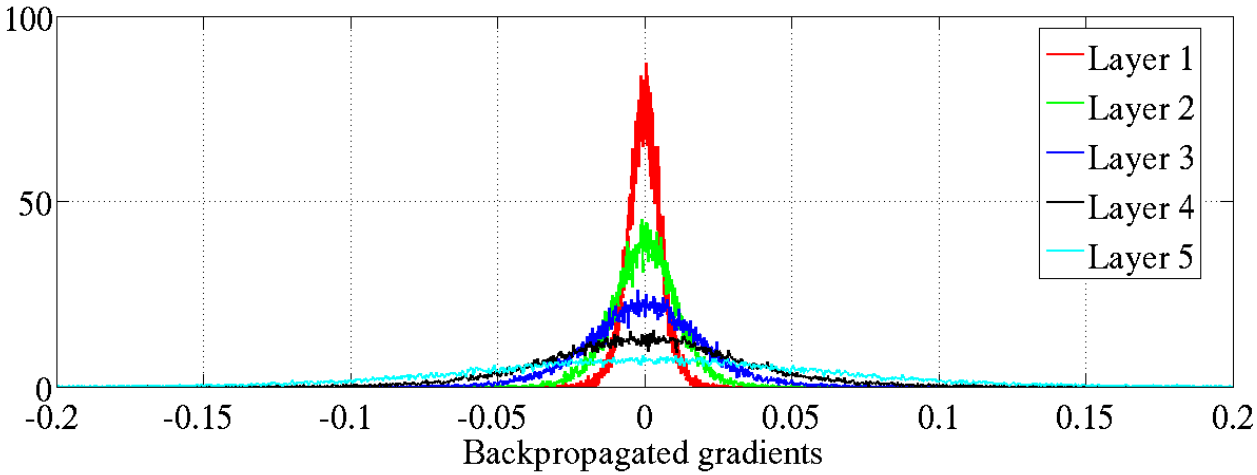


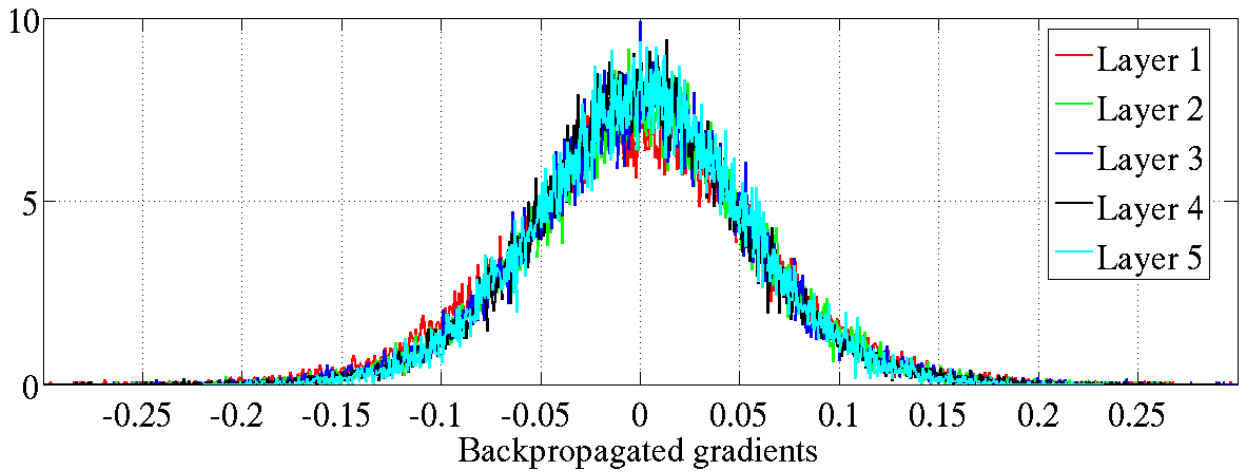
(Glorot and Bengio, 2010)

The analysis for the weight initialization relies on controlling

$$\mathbb{V}\left(\frac{\partial \ell}{\partial w_{i,j}^{(l)}}\right) \text{ and } \mathbb{V}\left(\frac{\partial \ell}{\partial b_{i}^{(l)}}\right)$$

where the parameters and inputs are randomized, so that **weights evolve at the same rate across layers during training,** and no layer reaches a saturation behavior before others.





(Glorot and Bengio, 2010)

Batch Normalization

- Batch Normalization reduces sensitivity to **initialization**

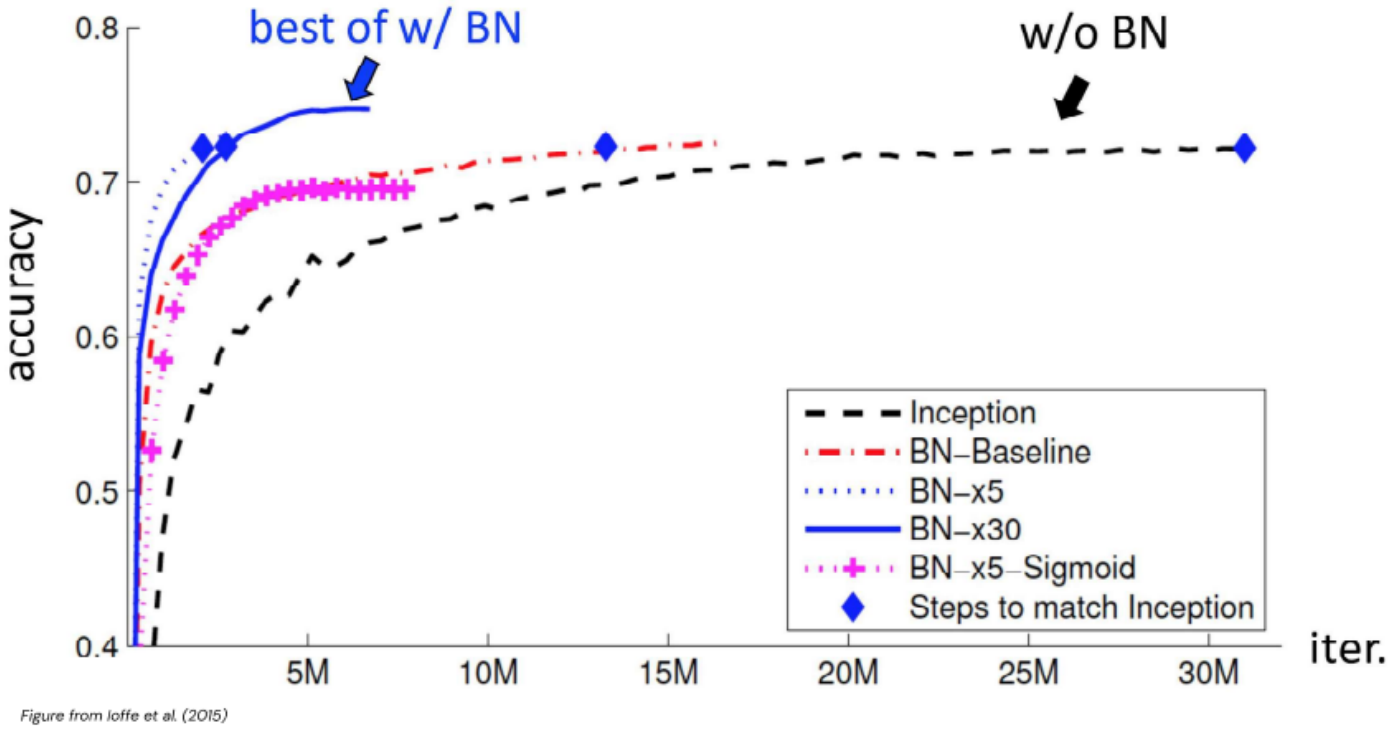
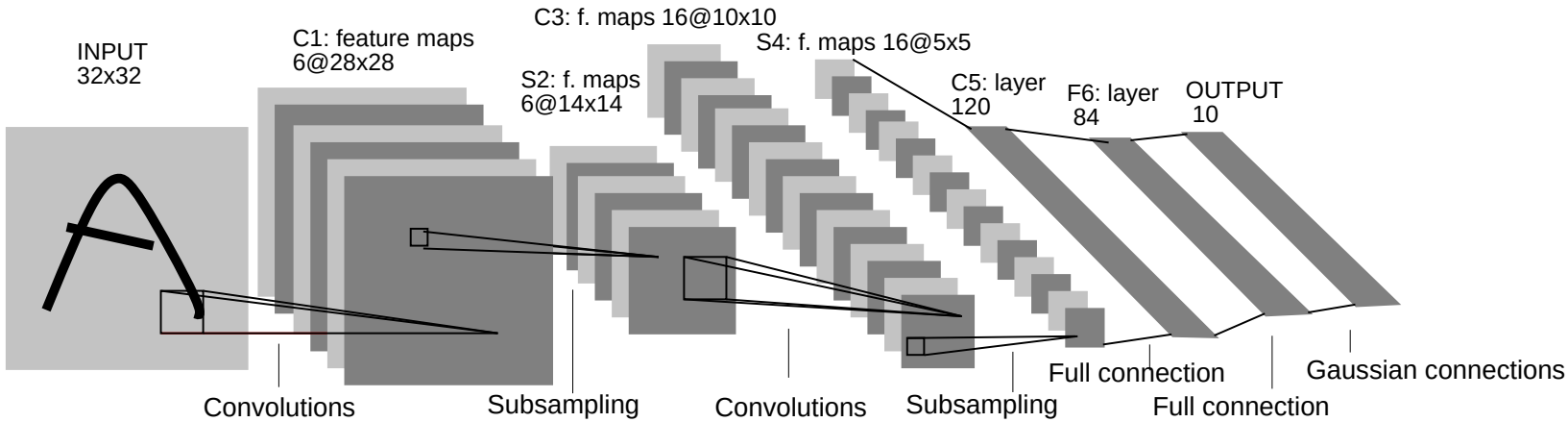


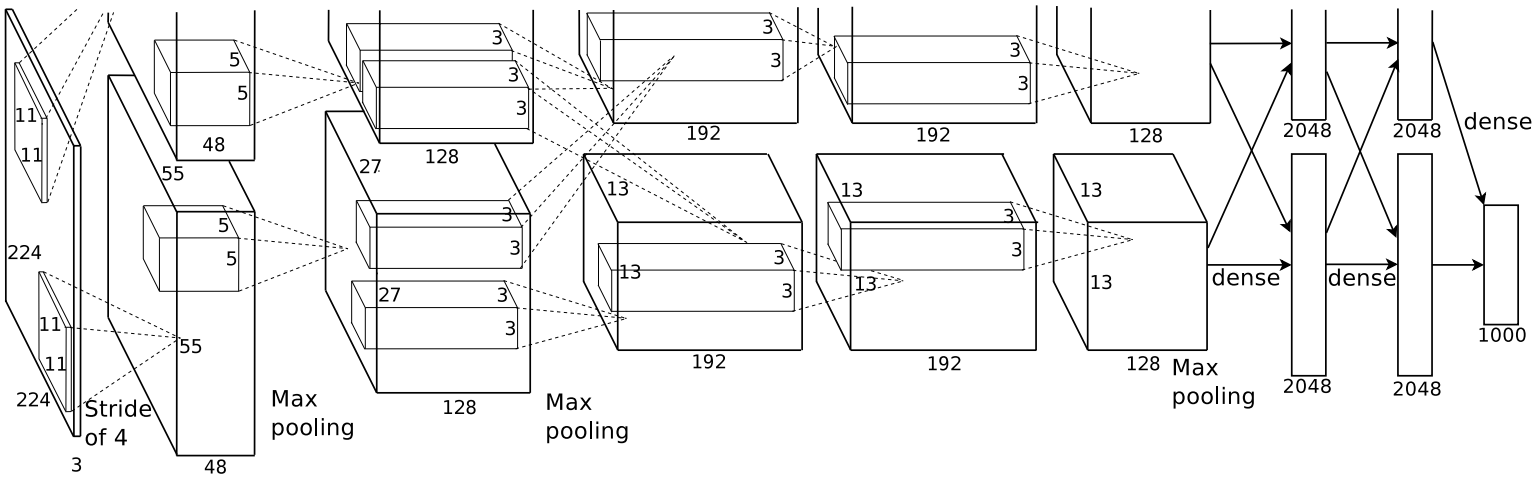
Figure: Batch Normalization

LeNet-5



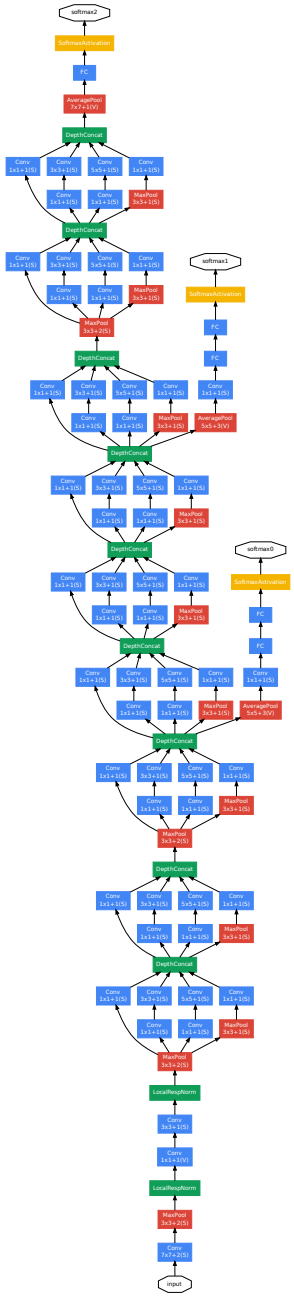
(leCun et al., 1998)

AlexNet



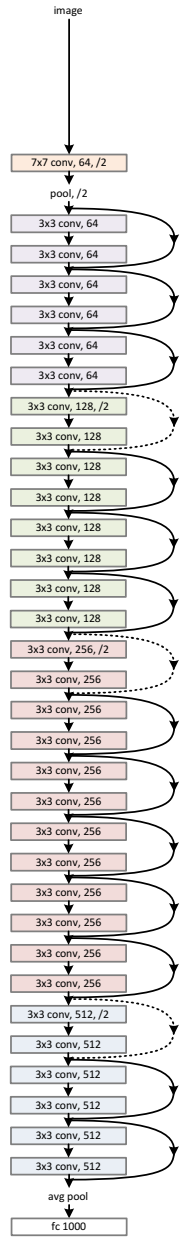
(Krizhevsky et al., 2012)

GoogLeNet

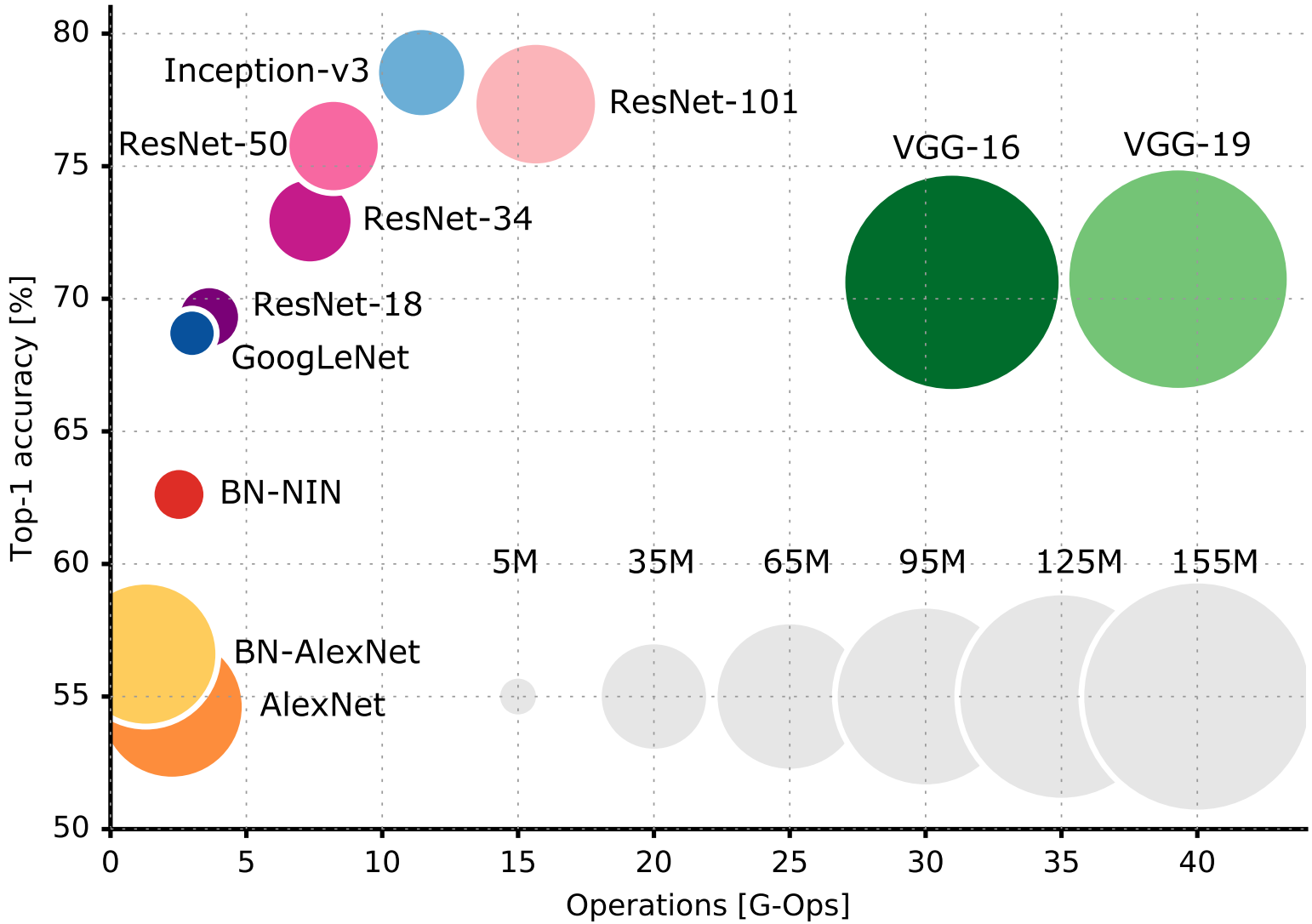


(Szegedy et al., 2015)

Resnet



(He et al., 2015)



(Canziani et al., 2016)

ResNets

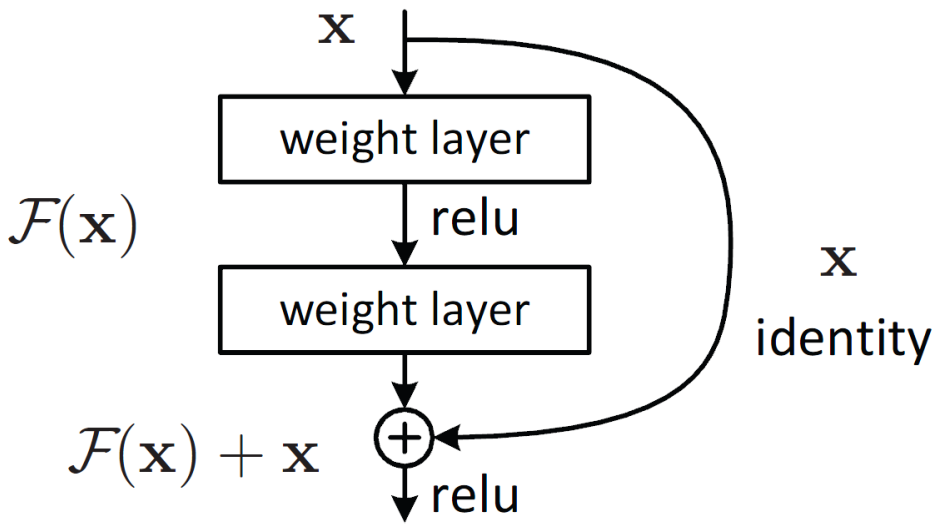


Figure: Residual Connection Block

ResNets

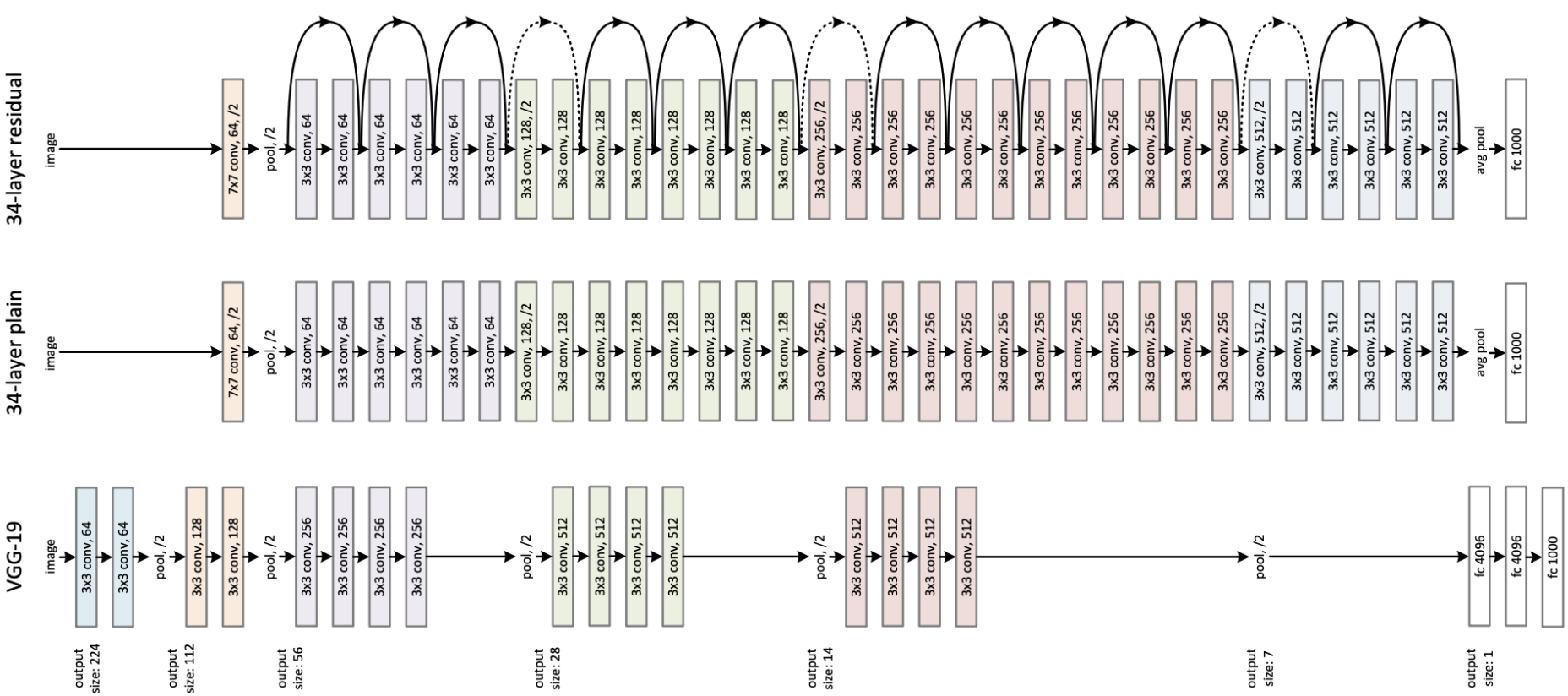


Figure: ResNet Architecture

Training & Deployment

Hyperparameters

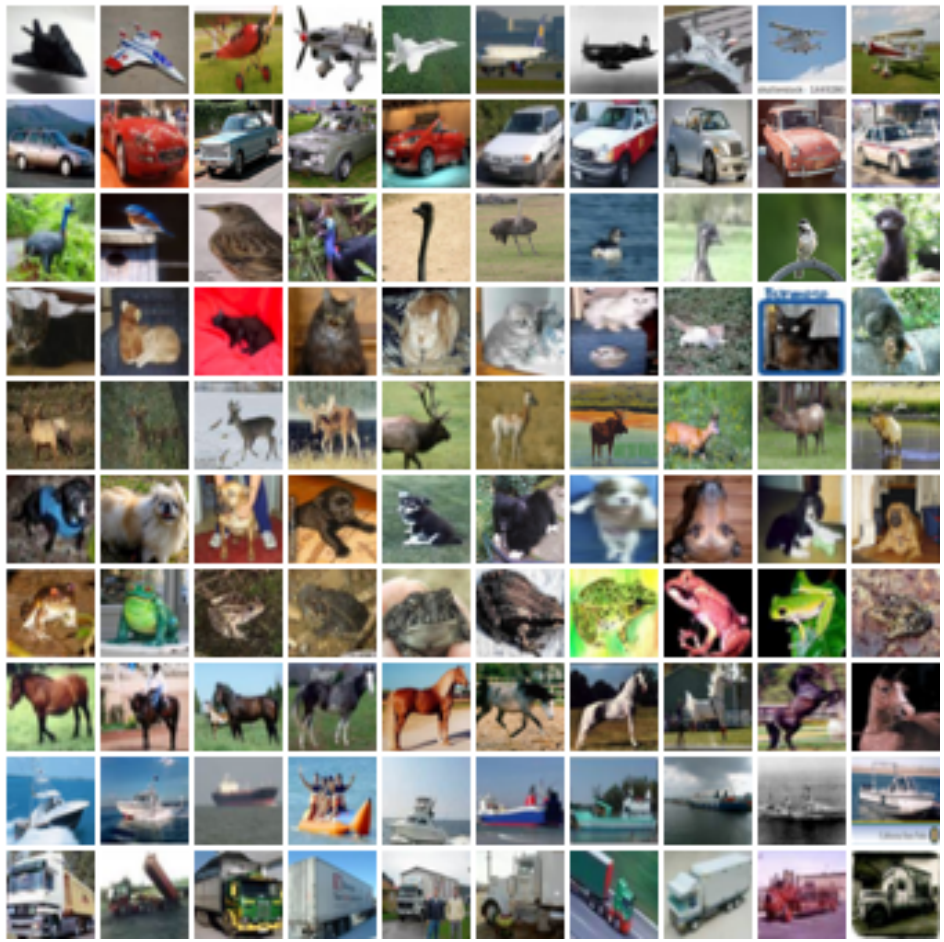
Regarding the learning rate, for training to succeed it has to

- reduce the loss quickly $\Rightarrow$ large learning rate,
- not be trapped in a bad minimum $\Rightarrow$ large learning rate,
- not bounce around in narrow valleys $\Rightarrow$ small learning rate, and
- not oscillate around a minimum $\Rightarrow$ small learning rate.

These constraints lead to a general policy of using **a larger step size first, and a smaller one in the end.**

The practical strategy is to look at the losses and error rates across epochs and pick a learning rate and learning rate adaptation. For instance by reducing it at discrete pre-defined steps, or with a geometric decay.

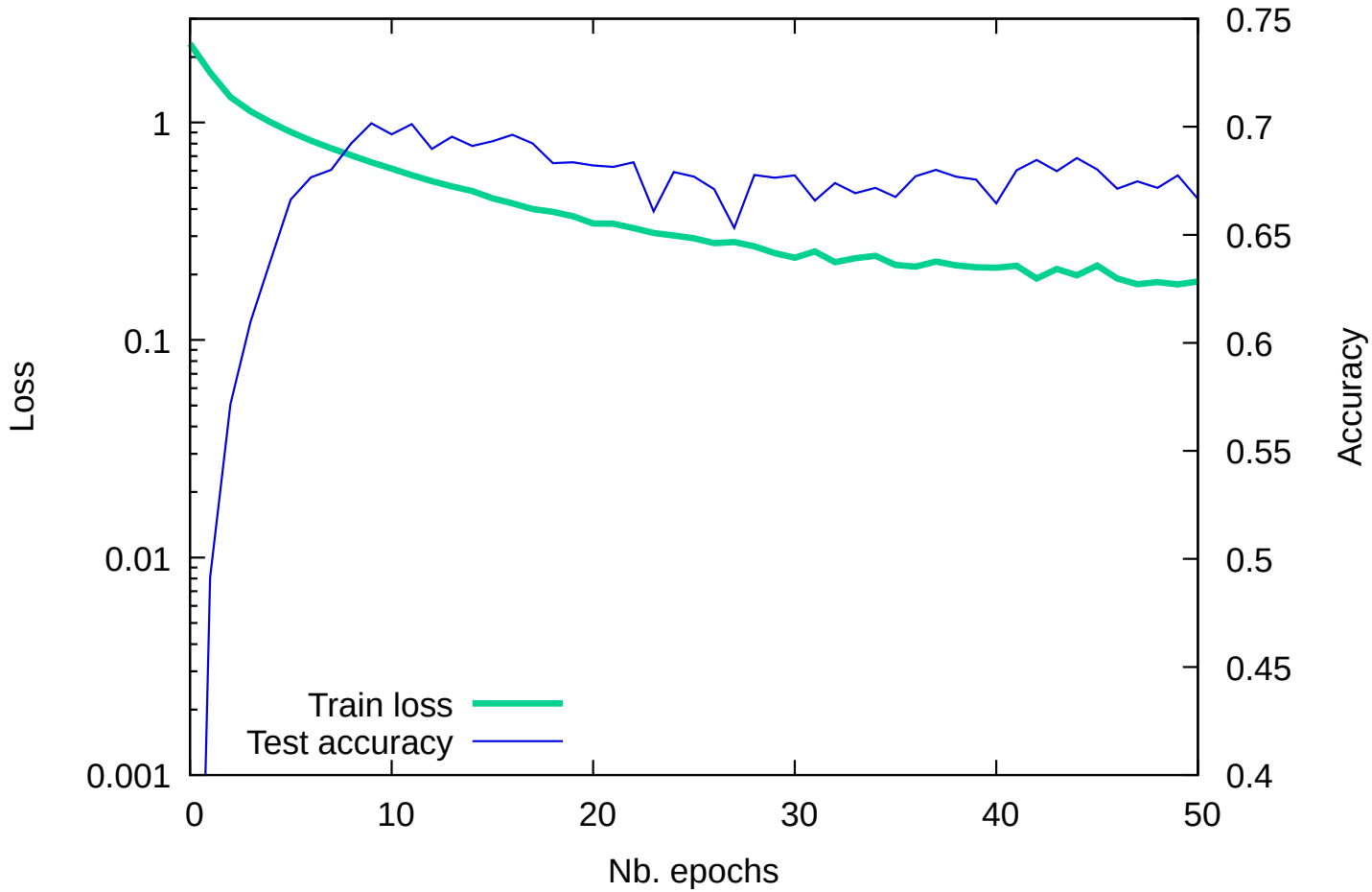
CIFAR10 data-set



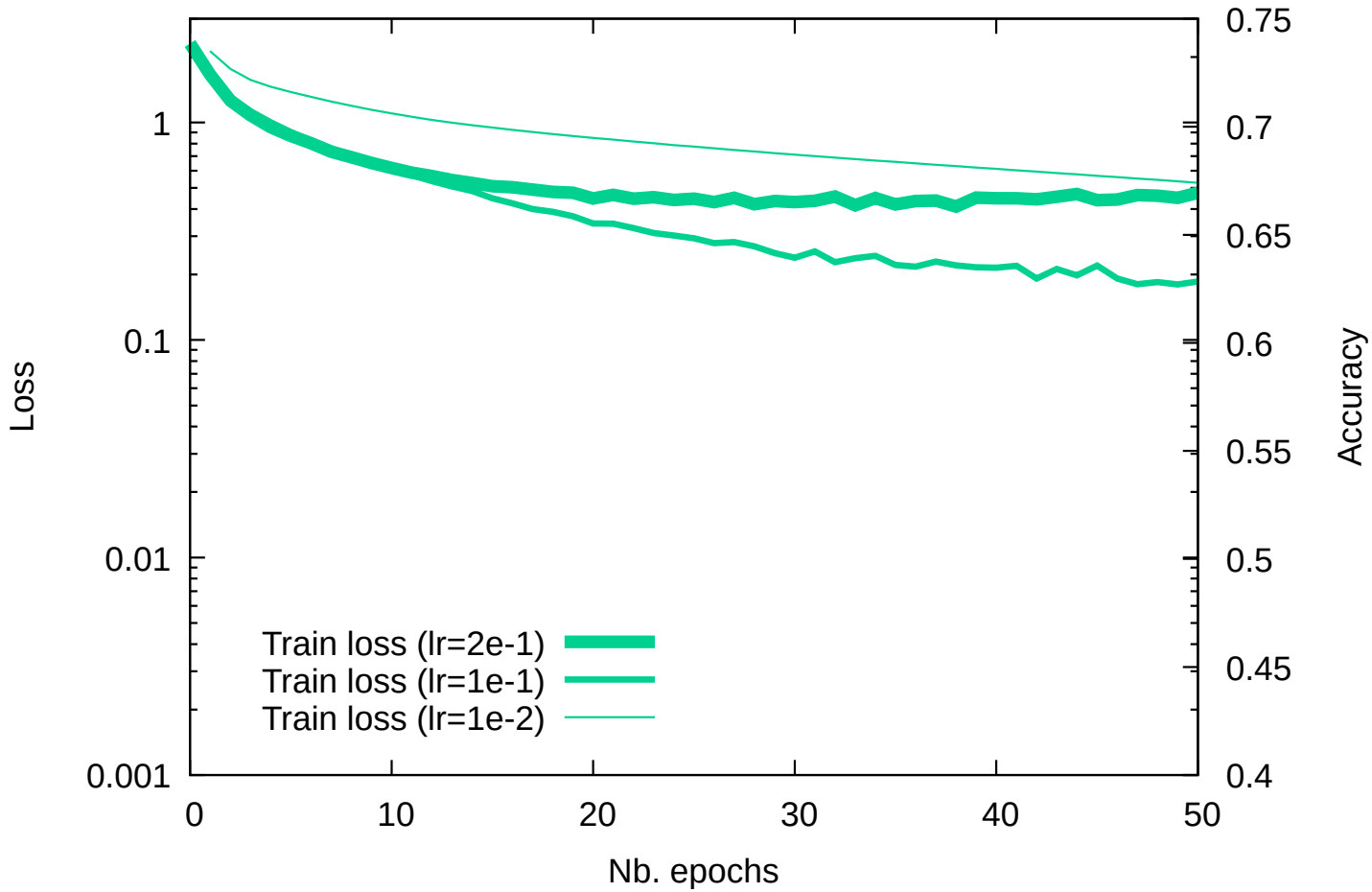
32×32 color images, $50,000$ train samples, $10,000$ test samples.

(Krizhevsky, 2009, chap. 3)

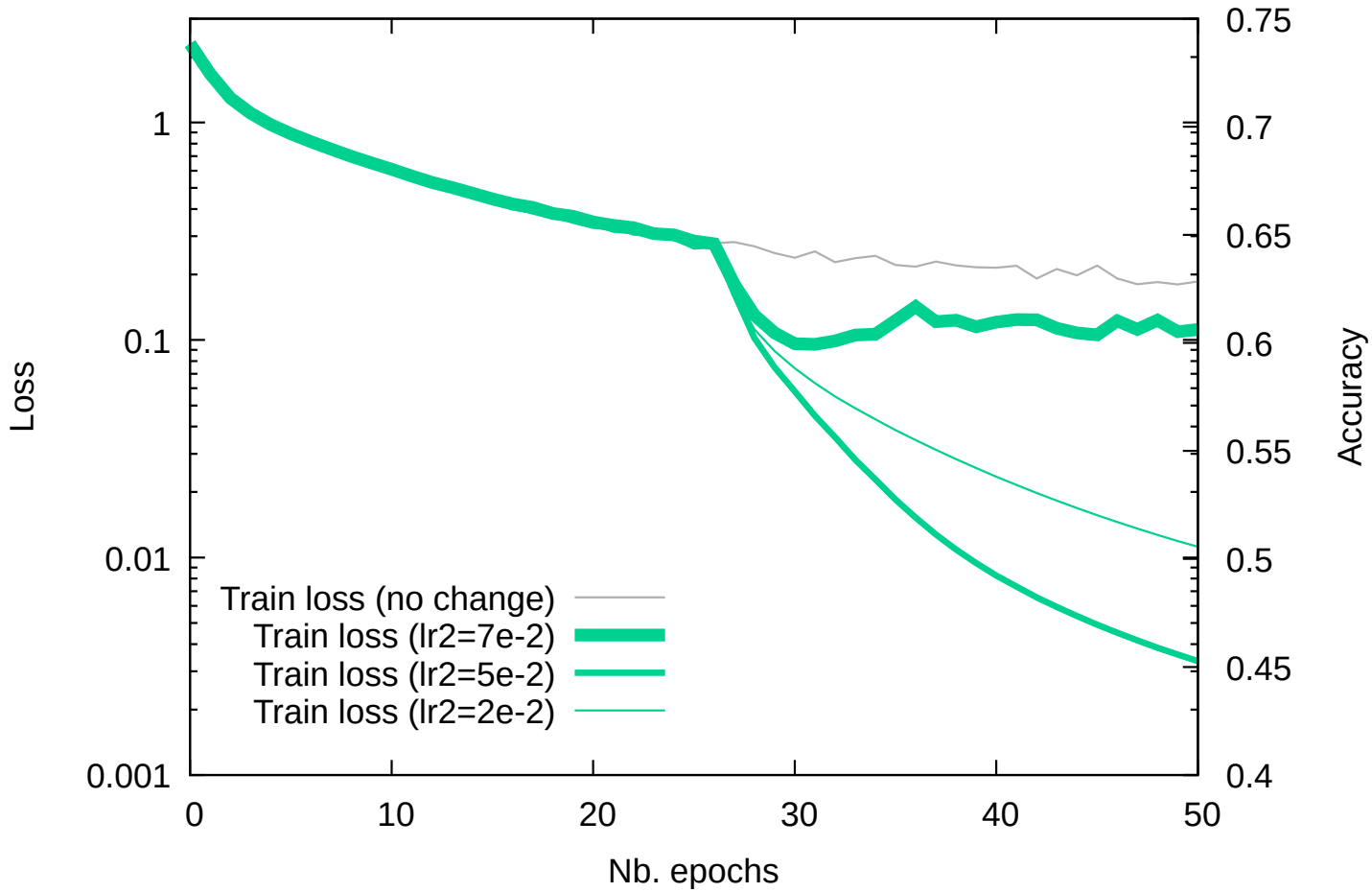
Small convnet on CIFAR10, cross-entropy, batch size 100, $\eta = 1e-1$.



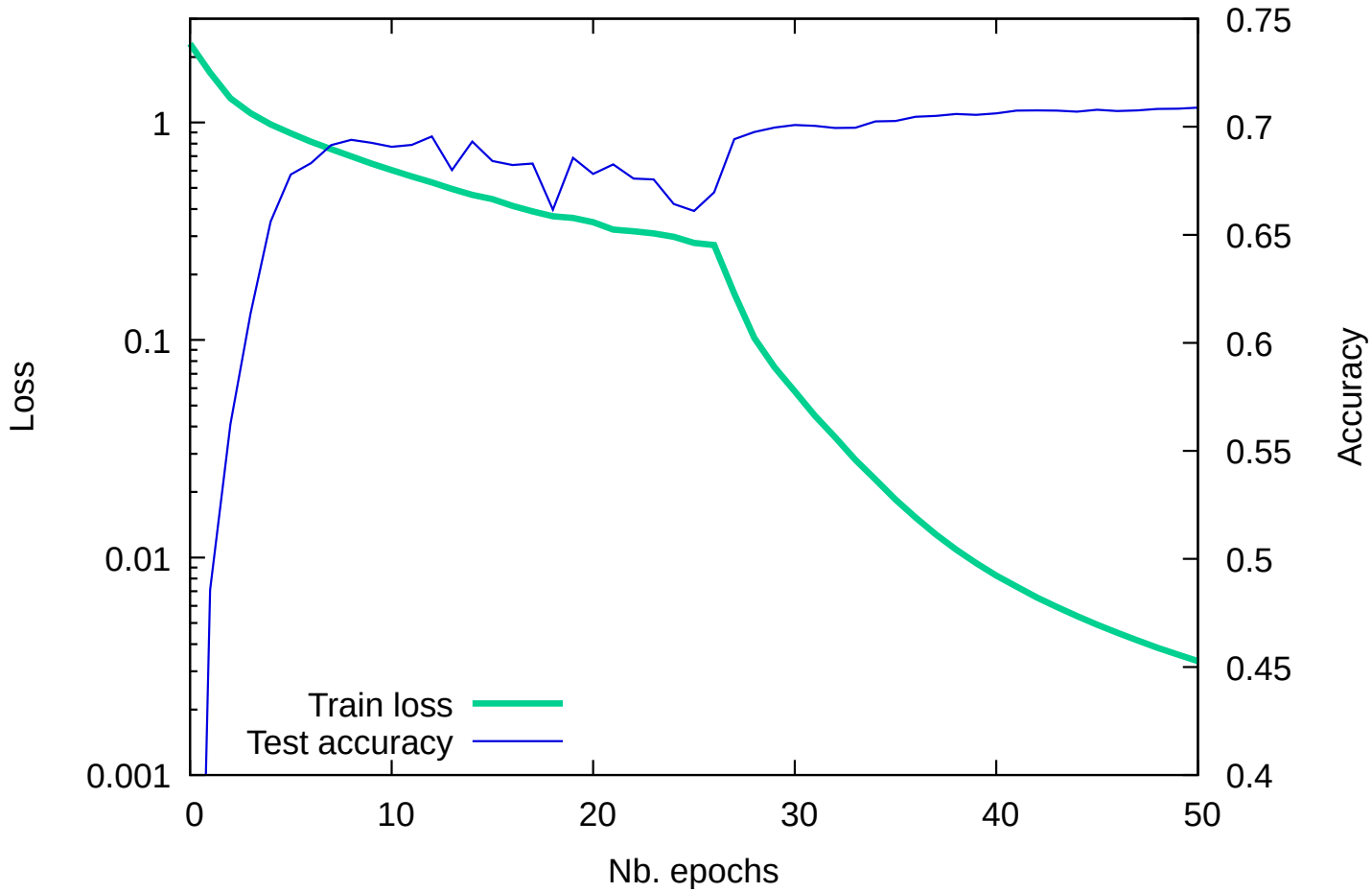
Small convnet on CIFAR10, cross-entropy, batch size 100



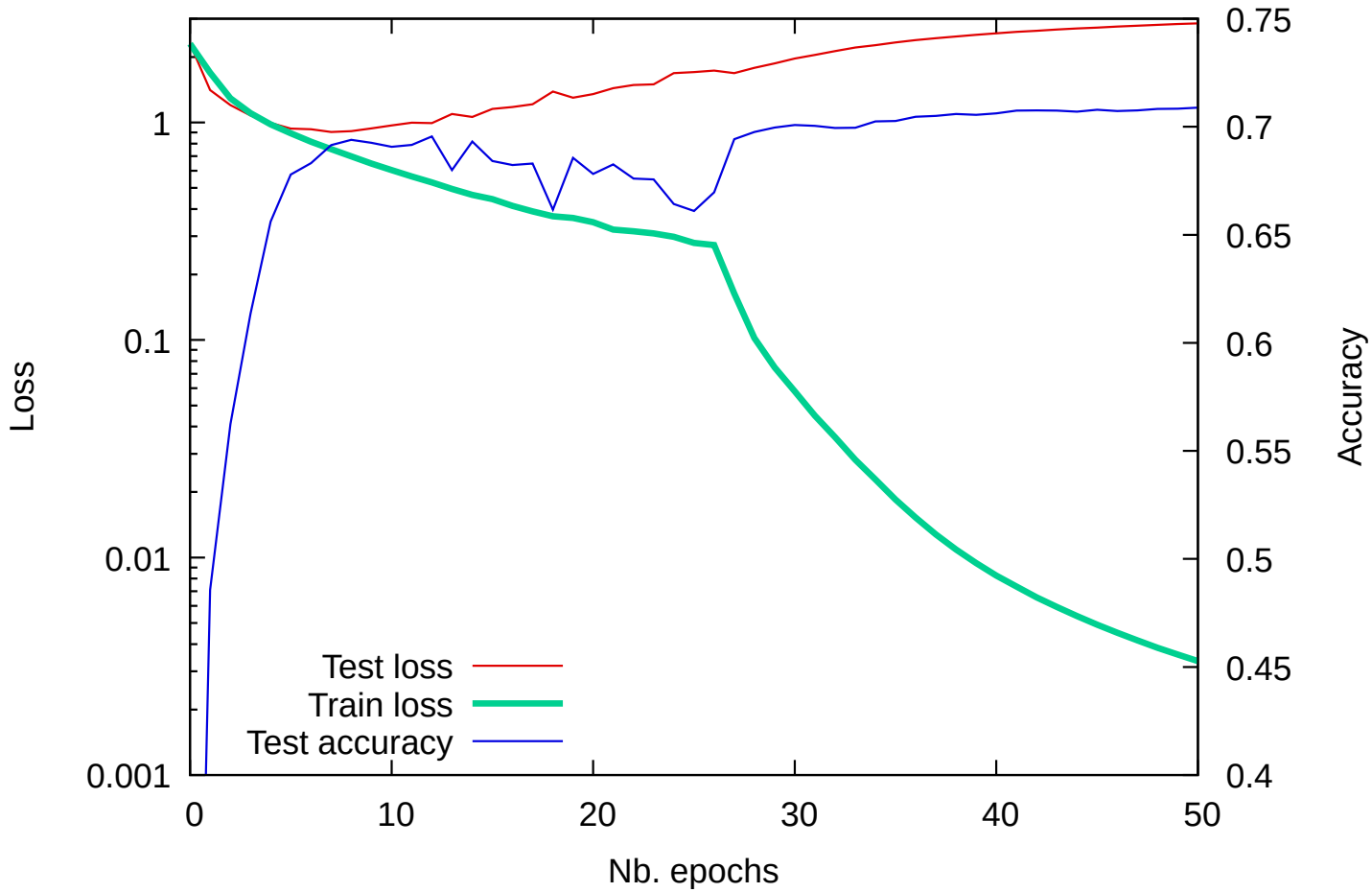
Using $\eta = 1e-1$ for 25 epochs, then reducing it.



Using $\eta = 1e-1$ for 25 epochs, then $\eta = 5e-2$.



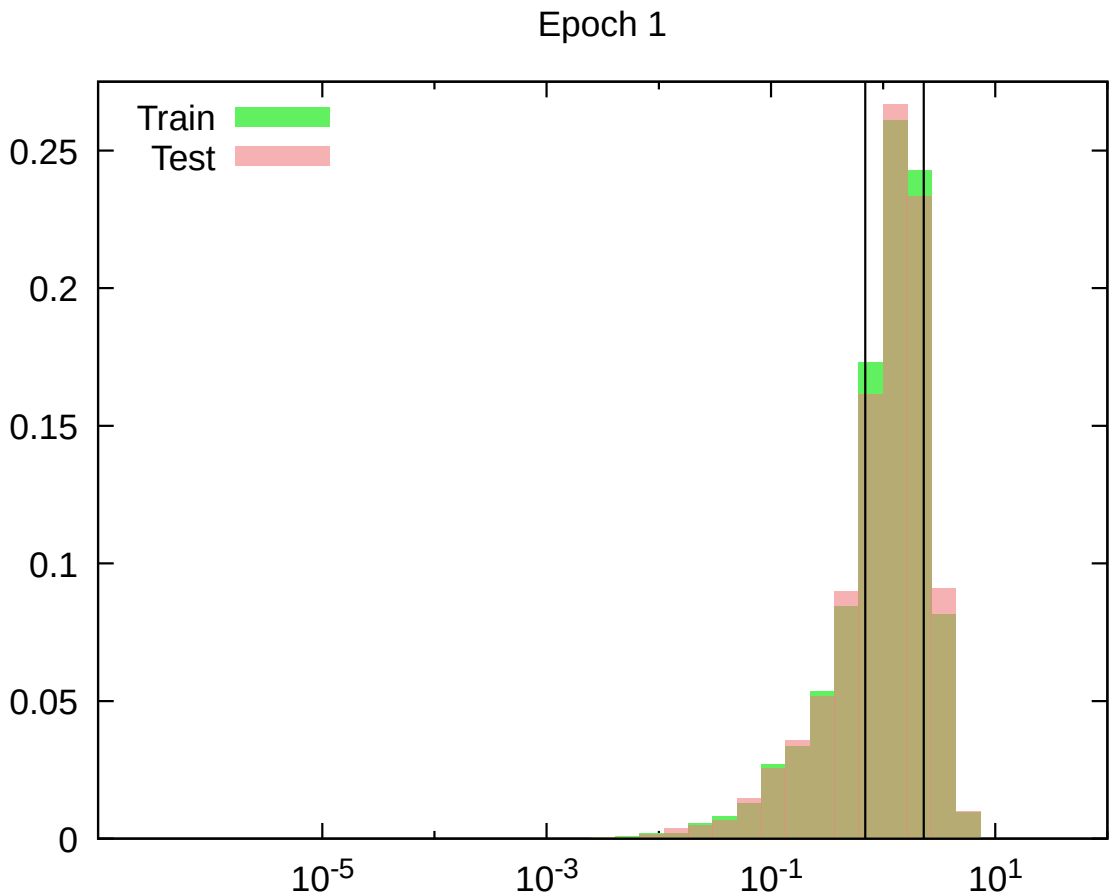
While the test error still goes down, the test loss may increase, as it gets even worse on misclassified examples, and decreases less on the ones getting fixed.



We can plot the train and test distributions of the per-sample loss

$$\ell = -\log\left(\frac{\exp(f_Y(X;w))}{\sum_k \exp(f_k(X;w))}\right)$$

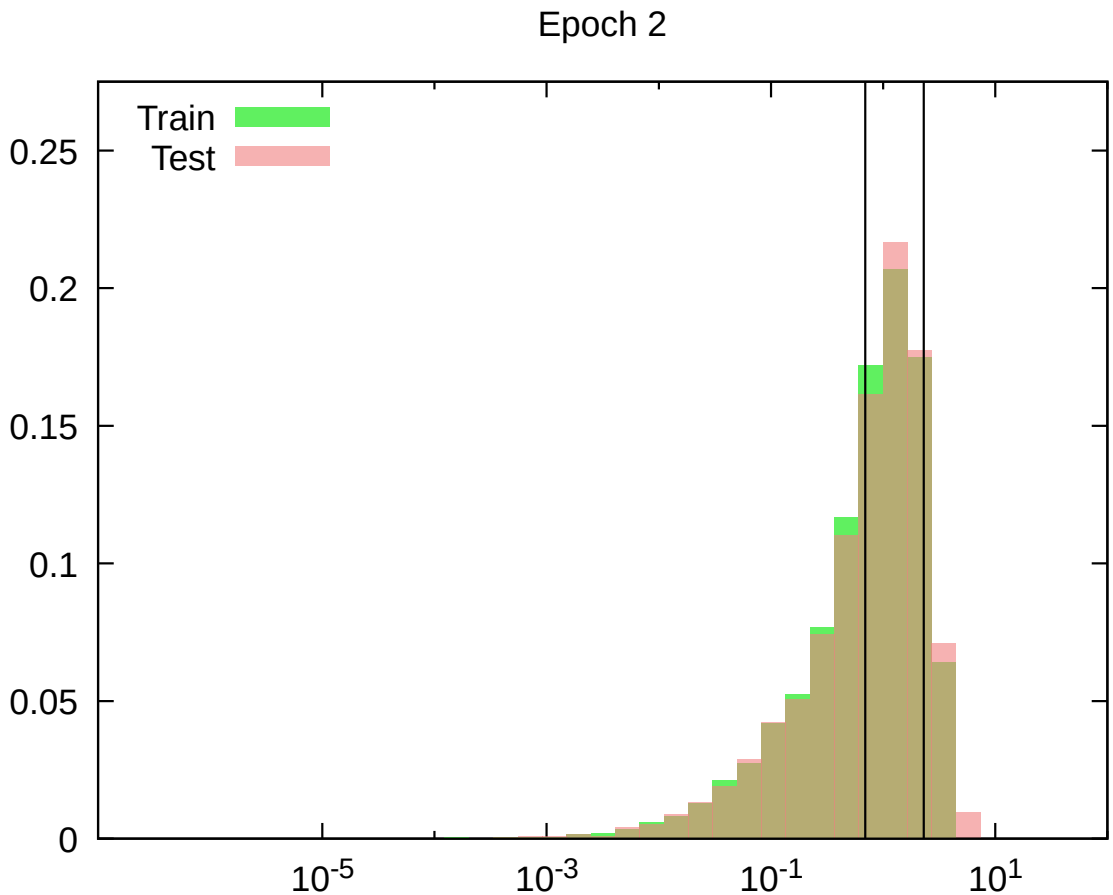
through epochs to visualize the over-fitting.



We can plot the train and test distributions of the per-sample loss

$$\ell = -\log\left(\frac{\exp(f_Y(X;w))}{\sum_k \exp(f_k(X;w))}\right)$$

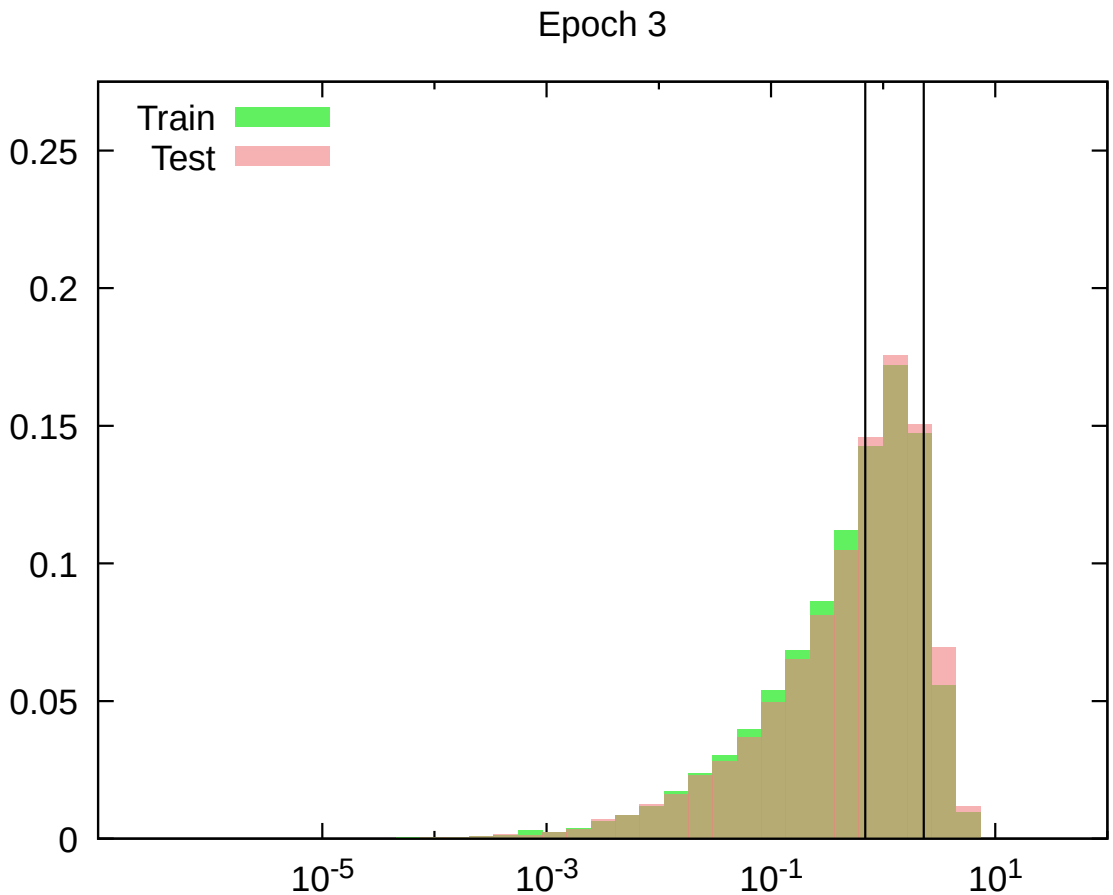
through epochs to visualize the over-fitting.



We can plot the train and test distributions of the per-sample loss

$$\ell = -\log\left(\frac{\exp(f_Y(X;w))}{\sum_k \exp(f_k(X;w))}\right)$$

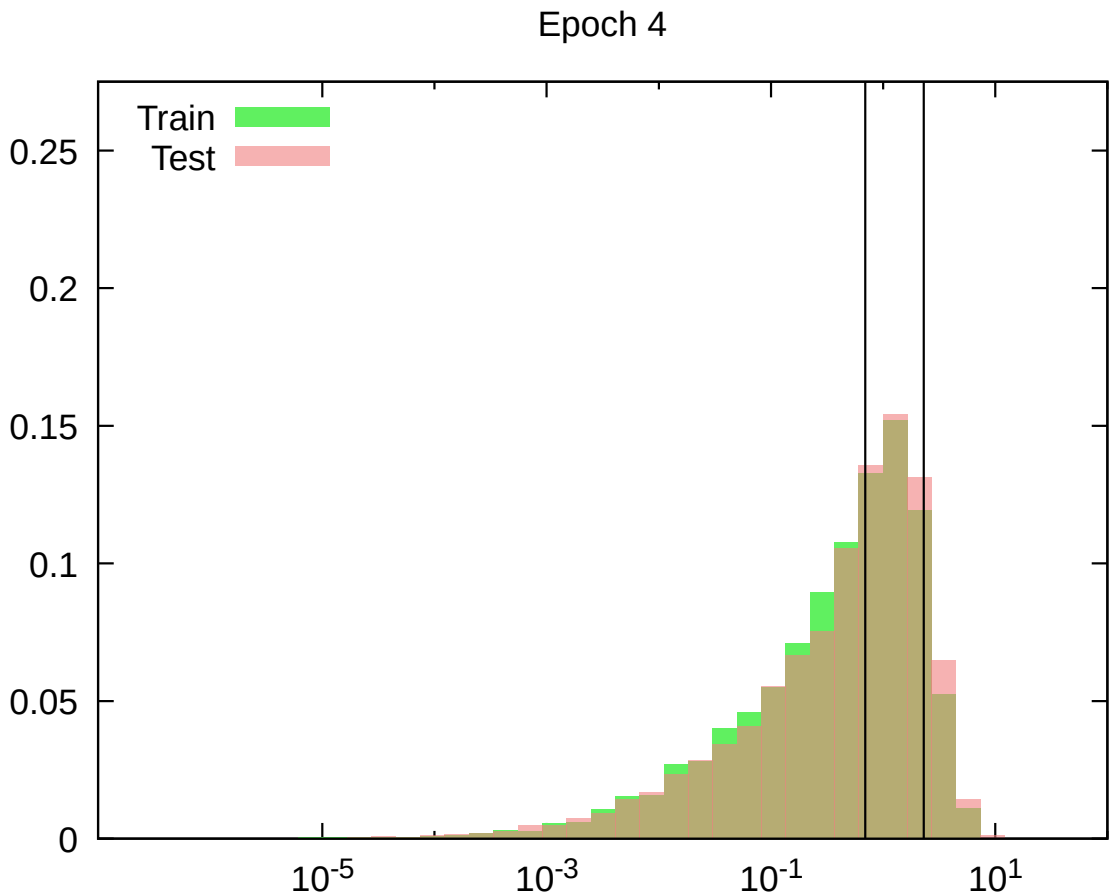
through epochs to visualize the over-fitting.



We can plot the train and test distributions of the per-sample loss

$$\ell = -\log\left(\frac{\exp(f_Y(X;w))}{\sum_k \exp(f_k(X;w))}\right)$$

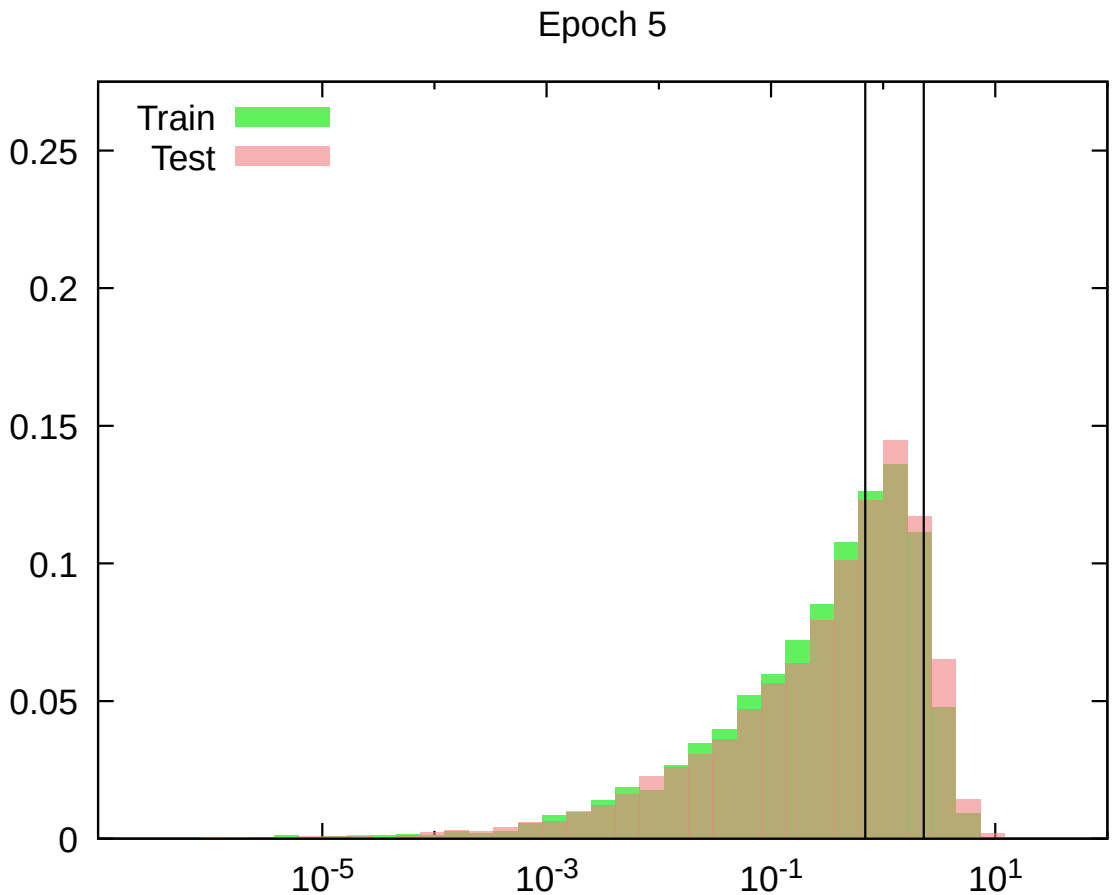
through epochs to visualize the over-fitting.



We can plot the train and test distributions of the per-sample loss

$$\ell = -\log\left(\frac{\exp(f_Y(X;w))}{\sum_k \exp(f_k(X;w))}\right)$$

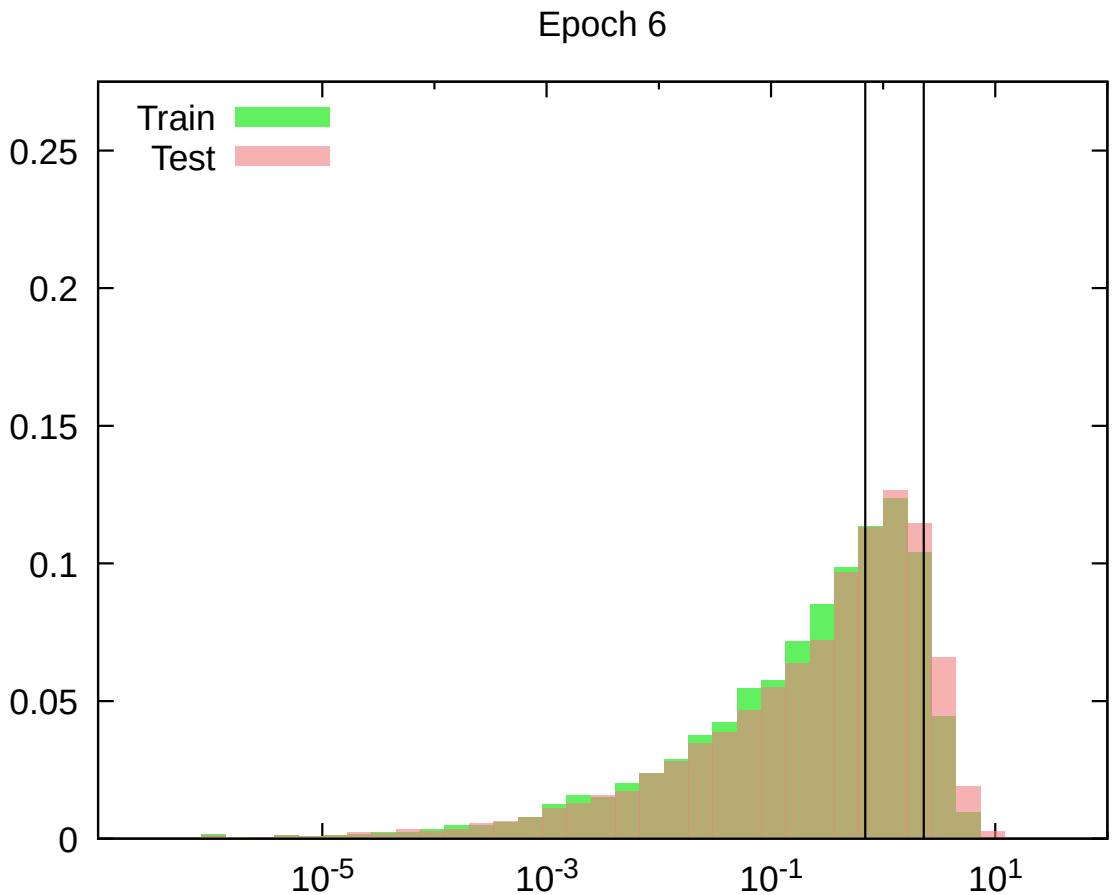
through epochs to visualize the over-fitting.



We can plot the train and test distributions of the per-sample loss

$$\ell = -\log\left(\frac{\exp(f_Y(X;w))}{\sum_k \exp(f_k(X;w))}\right)$$

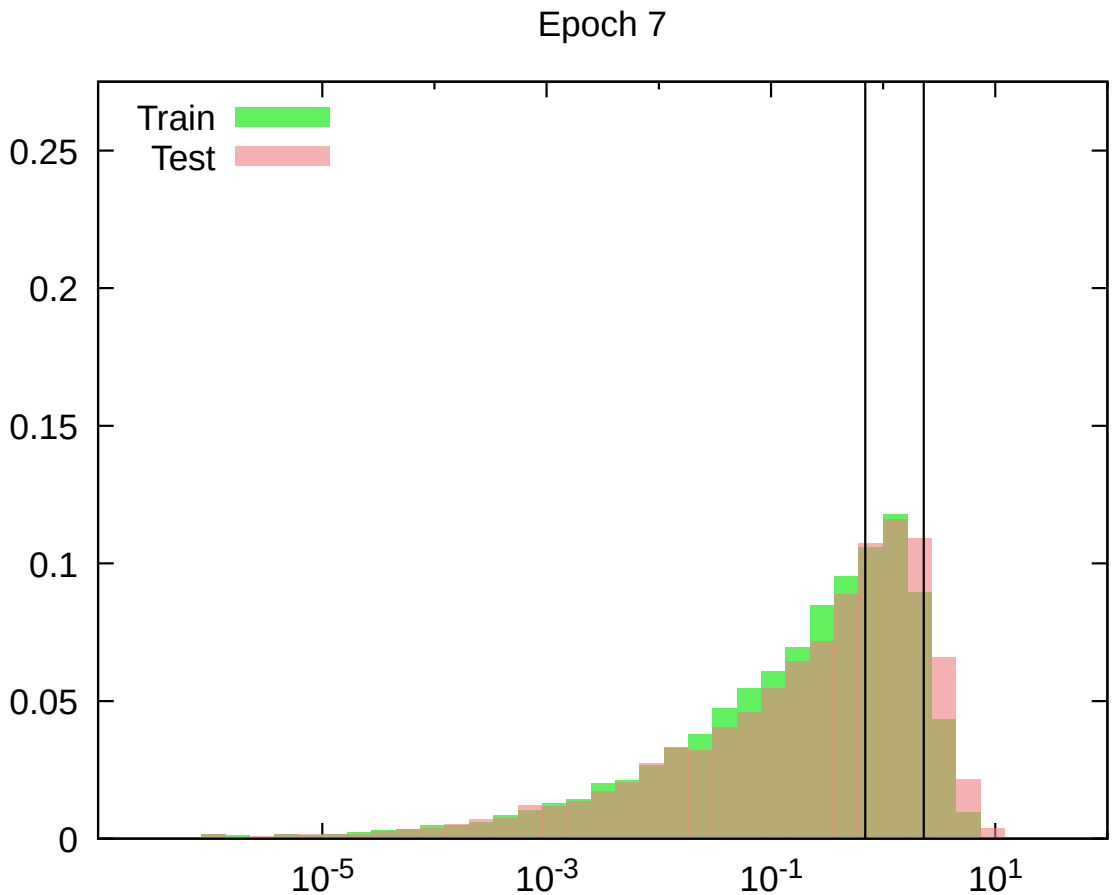
through epochs to visualize the over-fitting.



We can plot the train and test distributions of the per-sample loss

$$\ell = -\log\left(\frac{\exp(f_Y(X;w))}{\sum_k \exp(f_k(X;w))}\right)$$

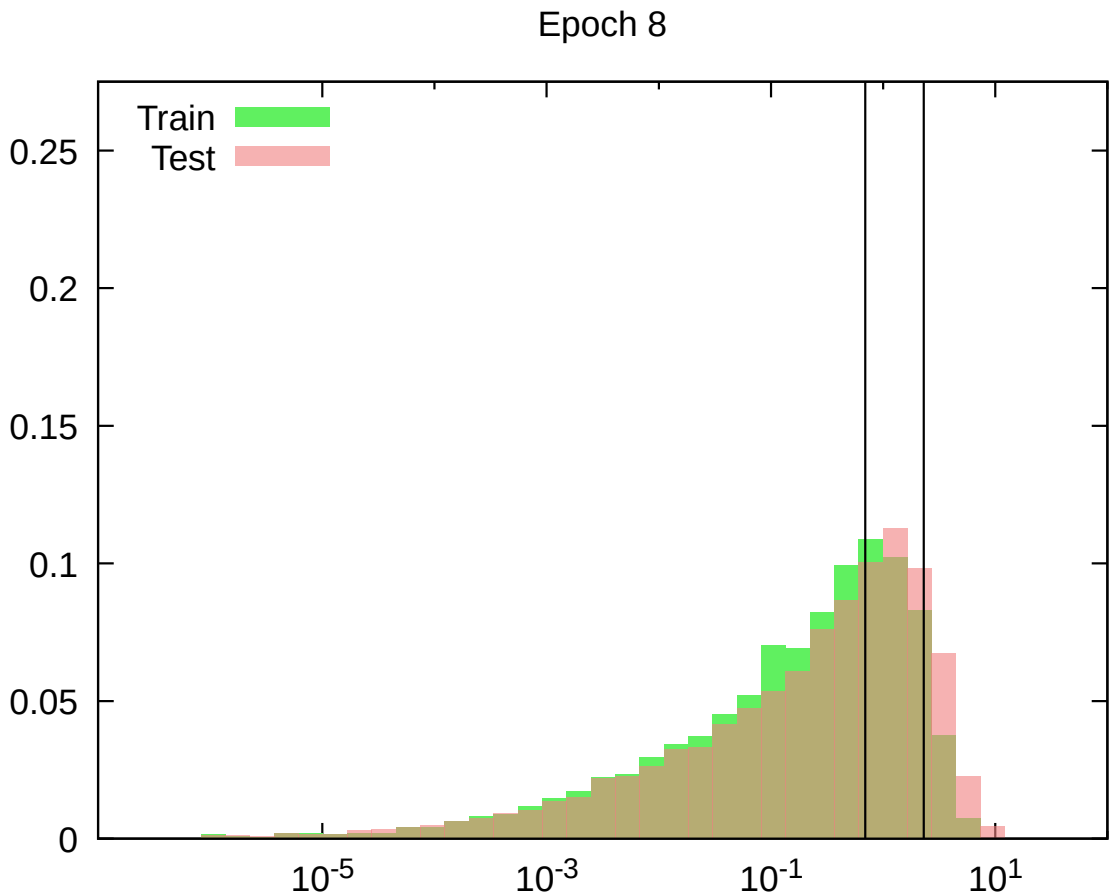
through epochs to visualize the over-fitting.



We can plot the train and test distributions of the per-sample loss

$$\ell = -\log\left(\frac{\exp(f_Y(X;w))}{\sum_k \exp(f_k(X;w))}\right)$$

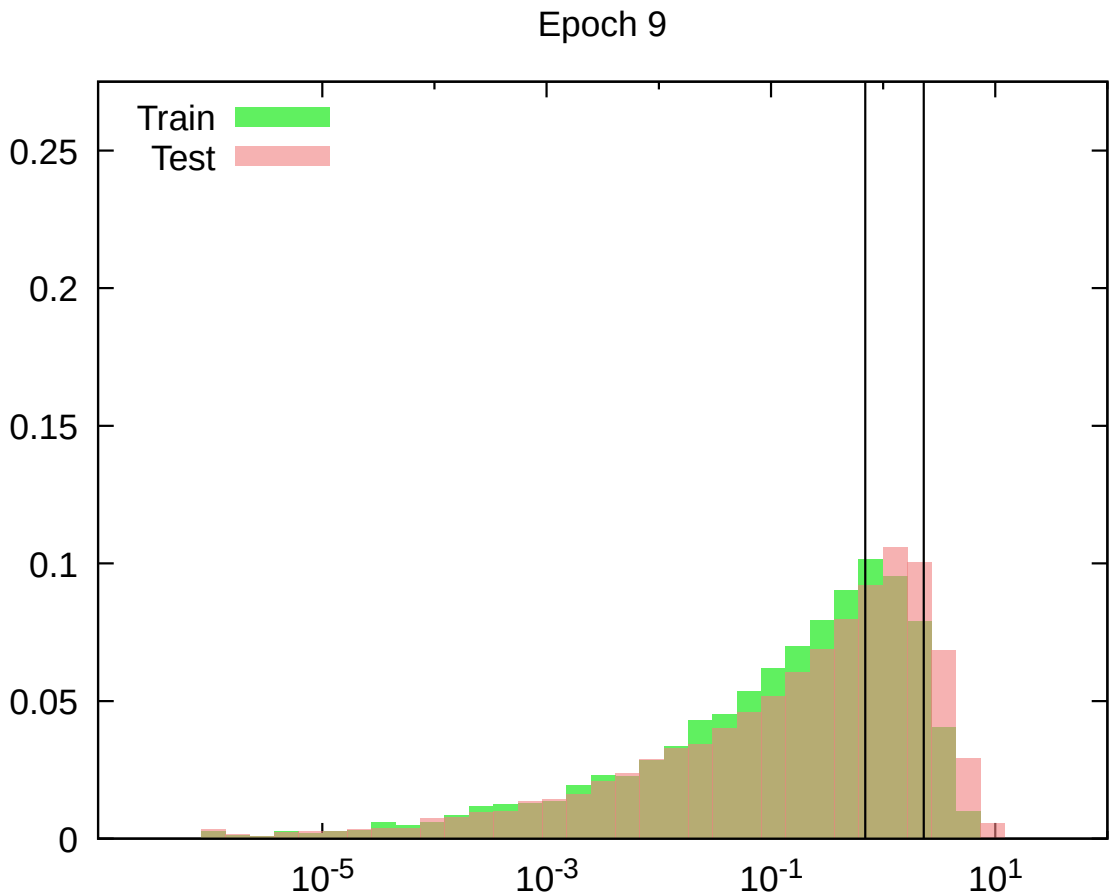
through epochs to visualize the over-fitting.



We can plot the train and test distributions of the per-sample loss

$$\ell = -\log\left(\frac{\exp(f_Y(X;w))}{\sum_k \exp(f_k(X;w))}\right)$$

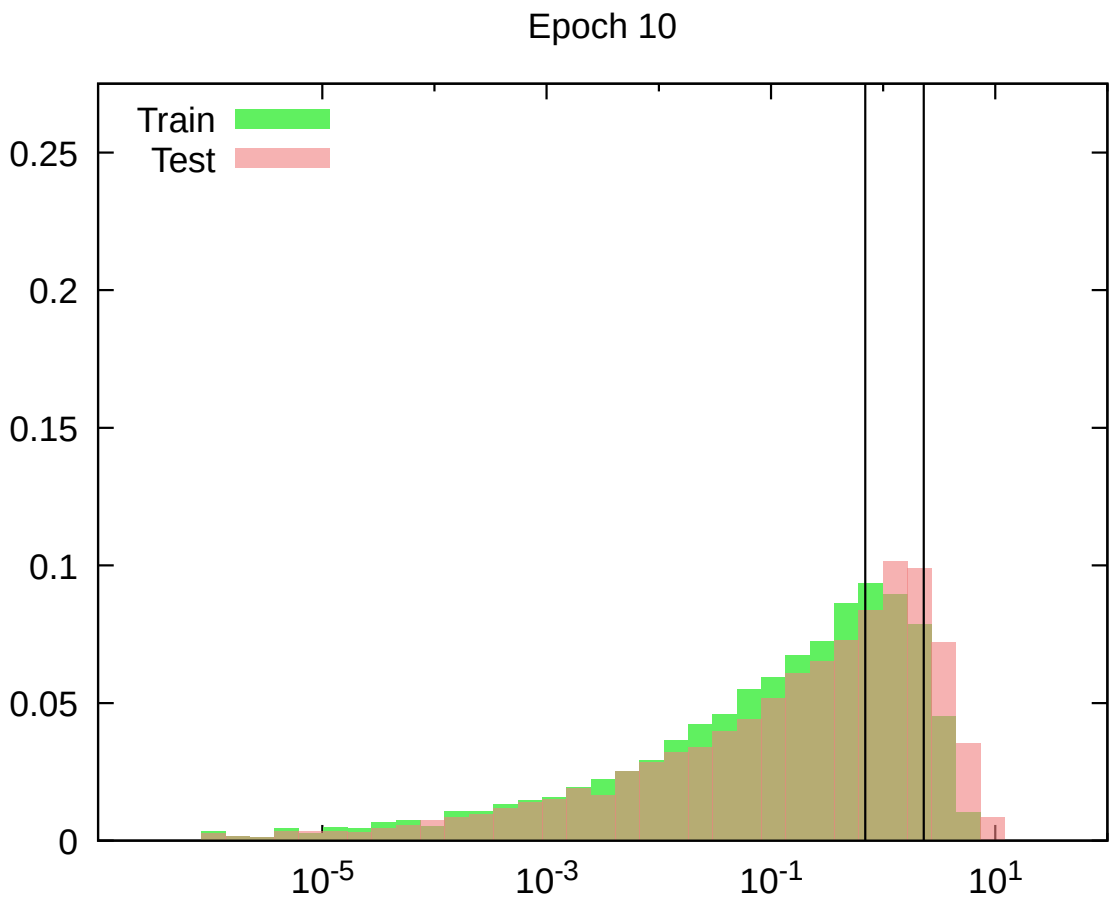
through epochs to visualize the over-fitting.



We can plot the train and test distributions of the per-sample loss

$$\ell = -\log\left(\frac{\exp(f_Y(X;w))}{\sum_k \exp(f_k(X;w))}\right)$$

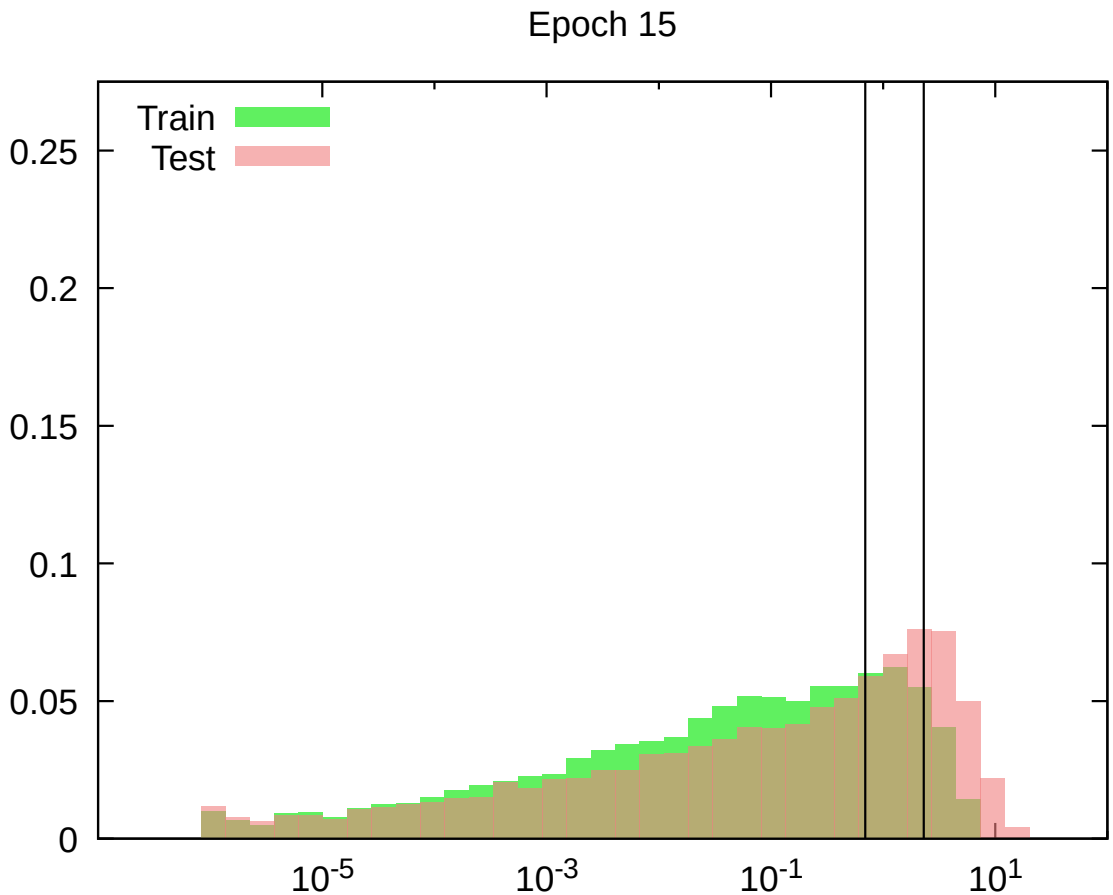
through epochs to visualize the over-fitting.



We can plot the train and test distributions of the per-sample loss

$$\ell = -\log\left(\frac{\exp(f_Y(X;w))}{\sum_k \exp(f_k(X;w))}\right)$$

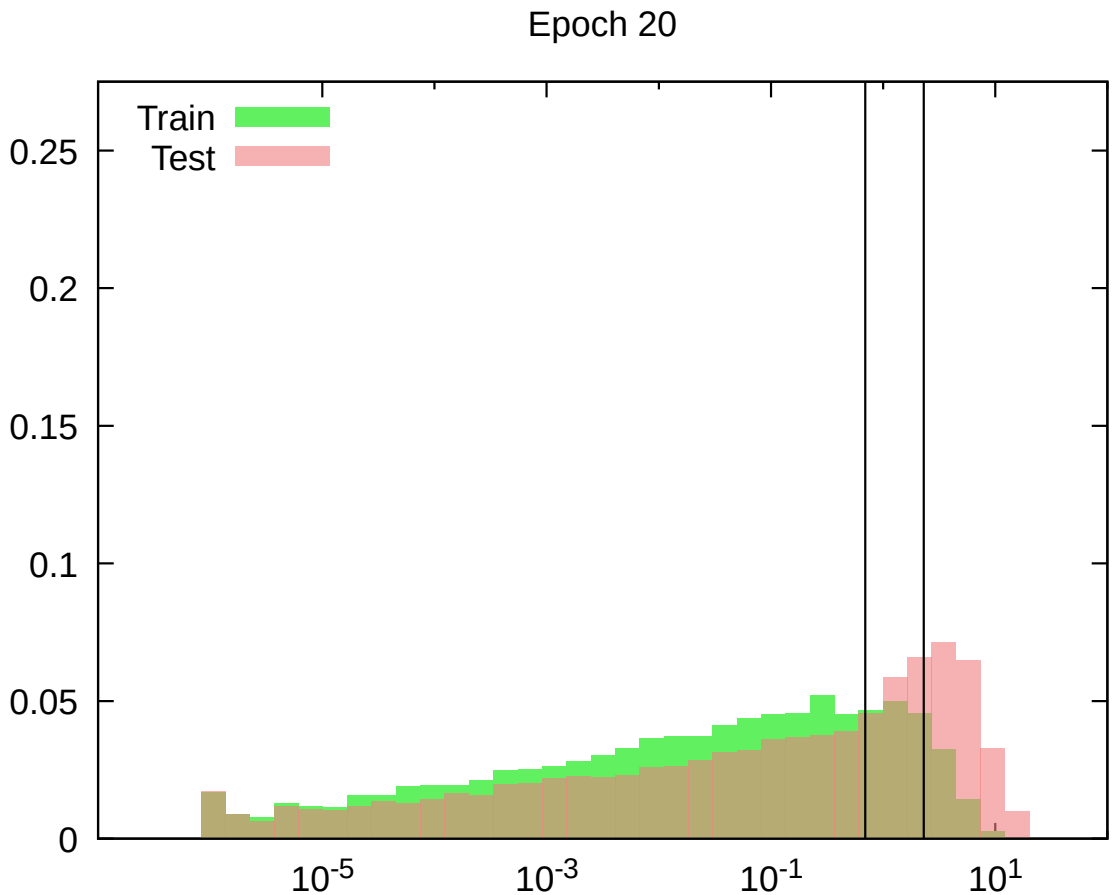
through epochs to visualize the over-fitting.



We can plot the train and test distributions of the per-sample loss

$$\ell = -\log\left(\frac{\exp(f_Y(X;w))}{\sum_k \exp(f_k(X;w))}\right)$$

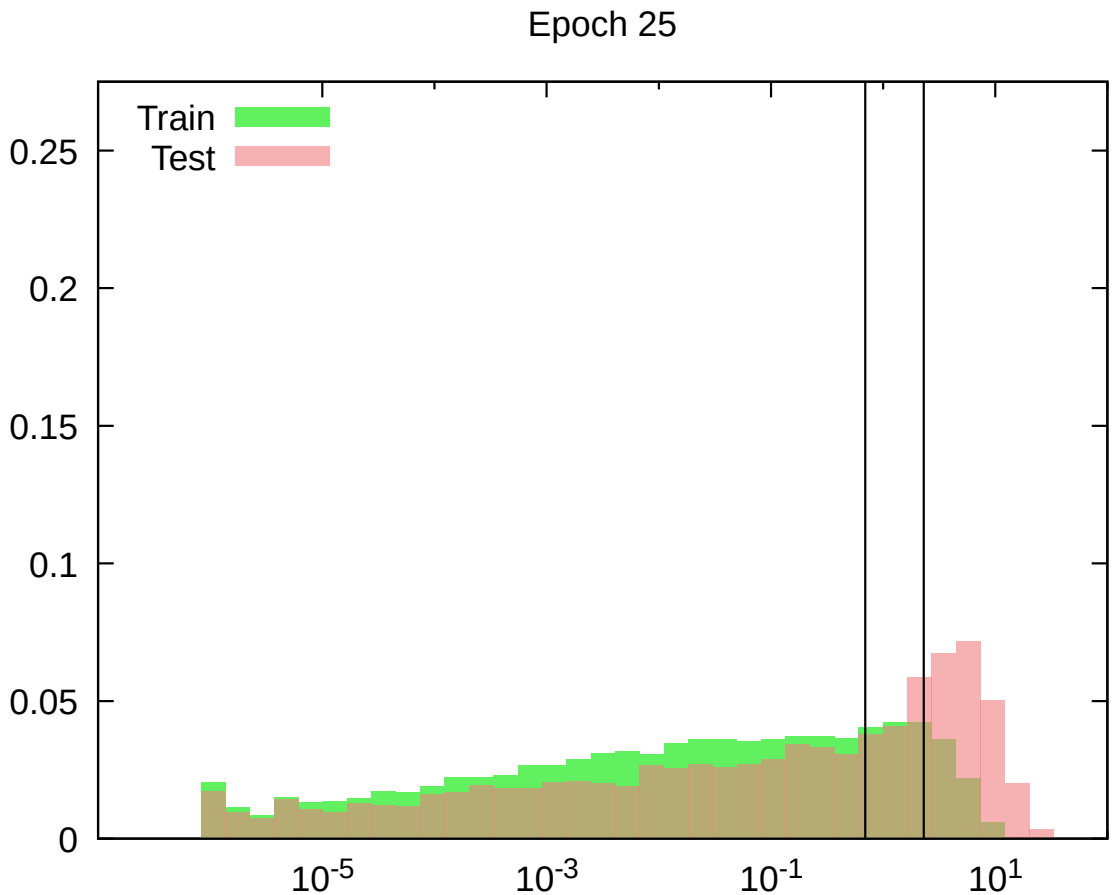
through epochs to visualize the over-fitting.



We can plot the train and test distributions of the per-sample loss

$$\ell = -\log\left(\frac{\exp(f_Y(X;w))}{\sum_k \exp(f_k(X;w))}\right)$$

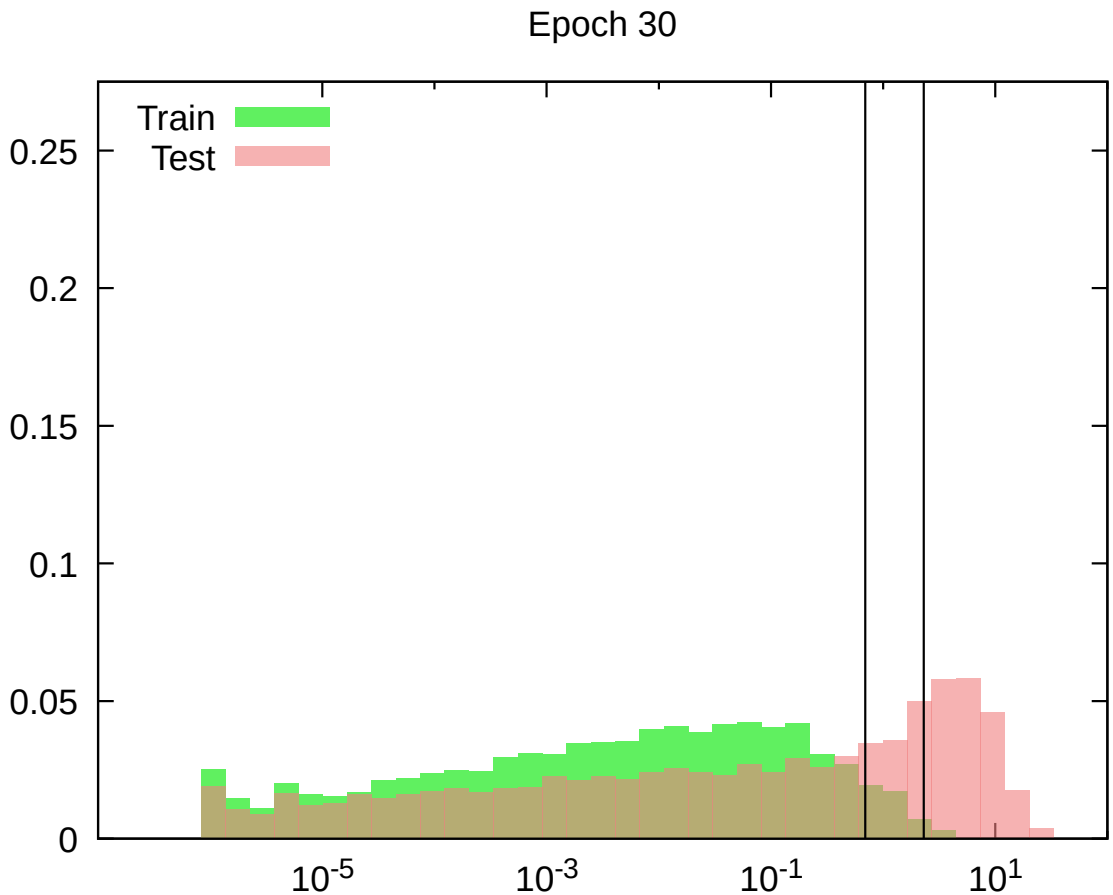
through epochs to visualize the over-fitting.



We can plot the train and test distributions of the per-sample loss

$$\ell = -\log\left(\frac{\exp(f_Y(X;w))}{\sum_k \exp(f_k(X;w))}\right)$$

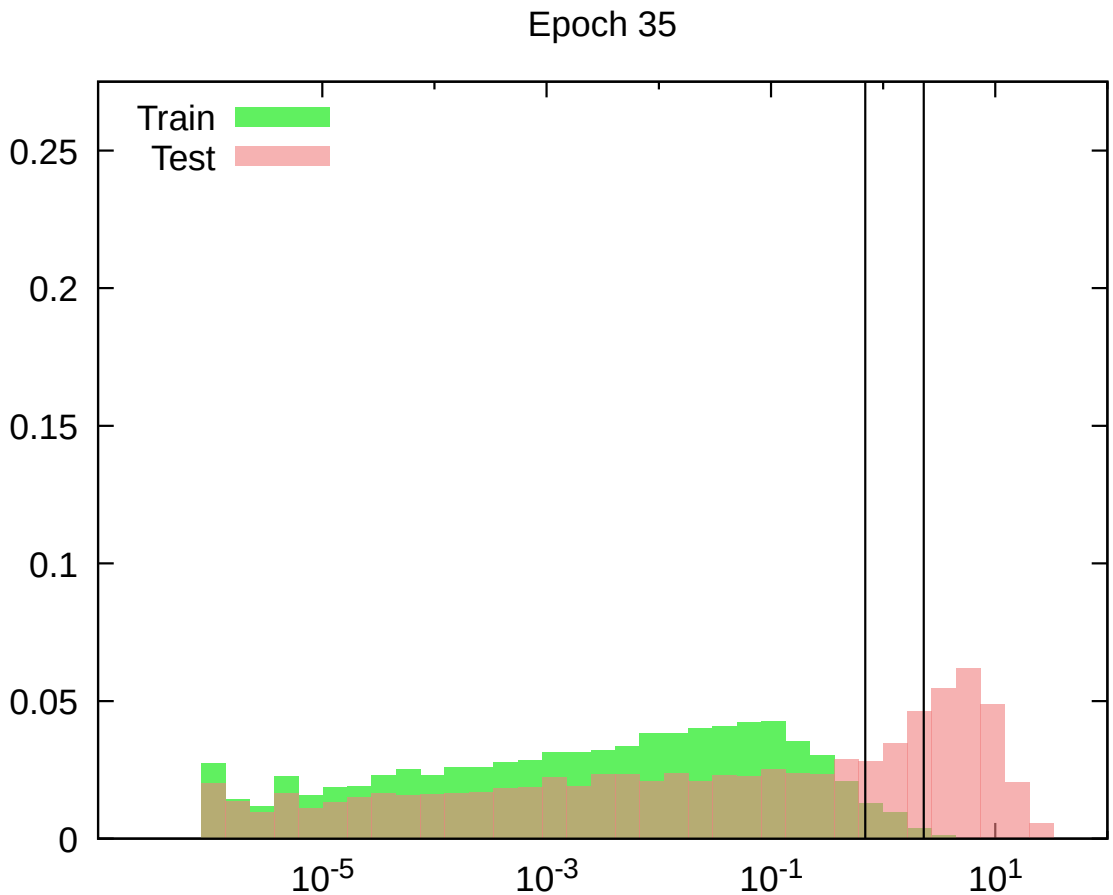
through epochs to visualize the over-fitting.



We can plot the train and test distributions of the per-sample loss

$$\ell = -\log\left(\frac{\exp(f_Y(X;w))}{\sum_k \exp(f_k(X;w))}\right)$$

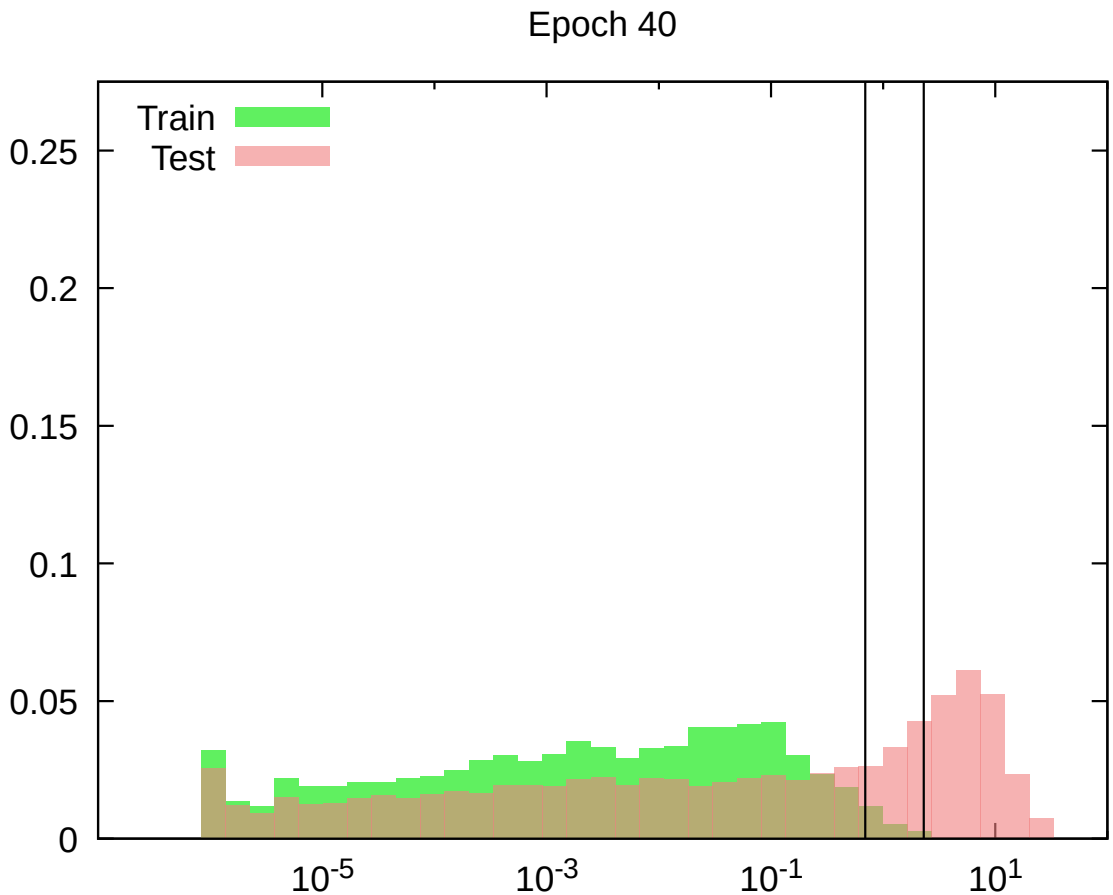
through epochs to visualize the over-fitting.



We can plot the train and test distributions of the per-sample loss

$$\ell = -\log\left(\frac{\exp(f_Y(X;w))}{\sum_k \exp(f_k(X;w))}\right)$$

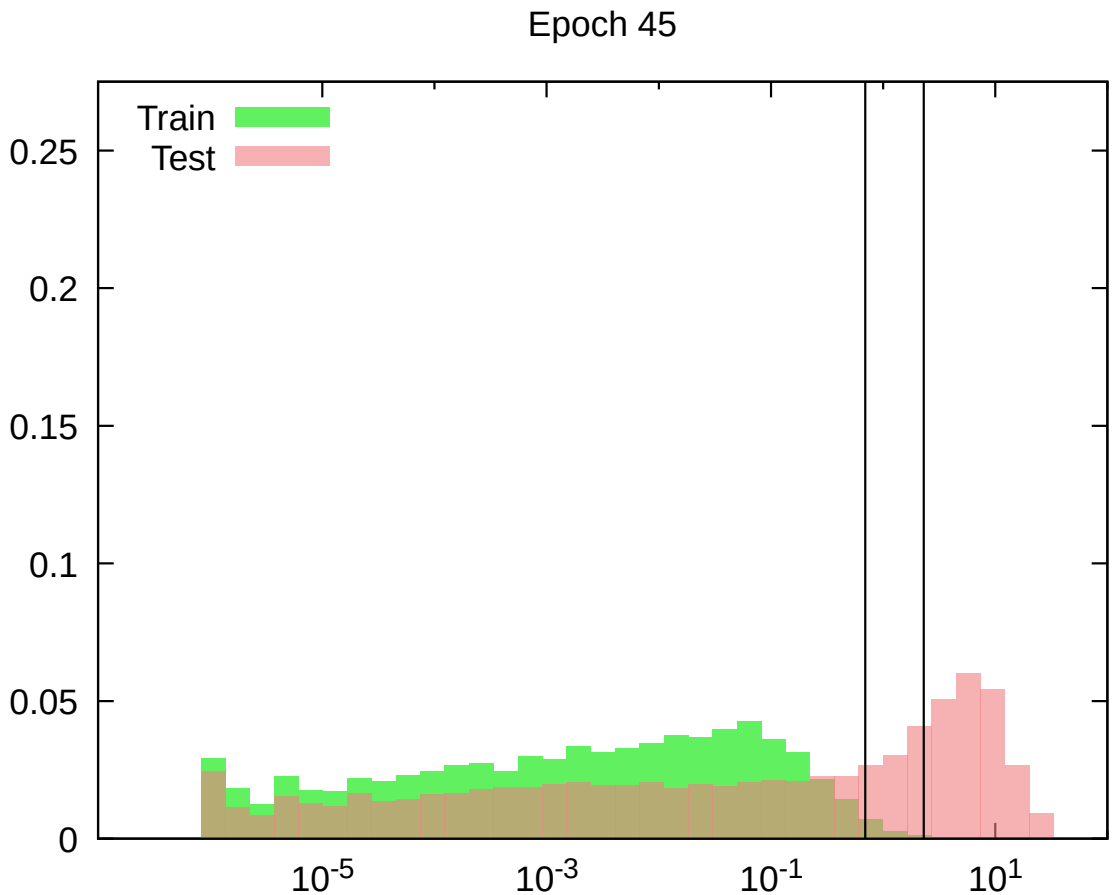
through epochs to visualize the over-fitting.



We can plot the train and test distributions of the per-sample loss

$$\ell = -\log\left(\frac{\exp(f_Y(X;w))}{\sum_k \exp(f_k(X;w))}\right)$$

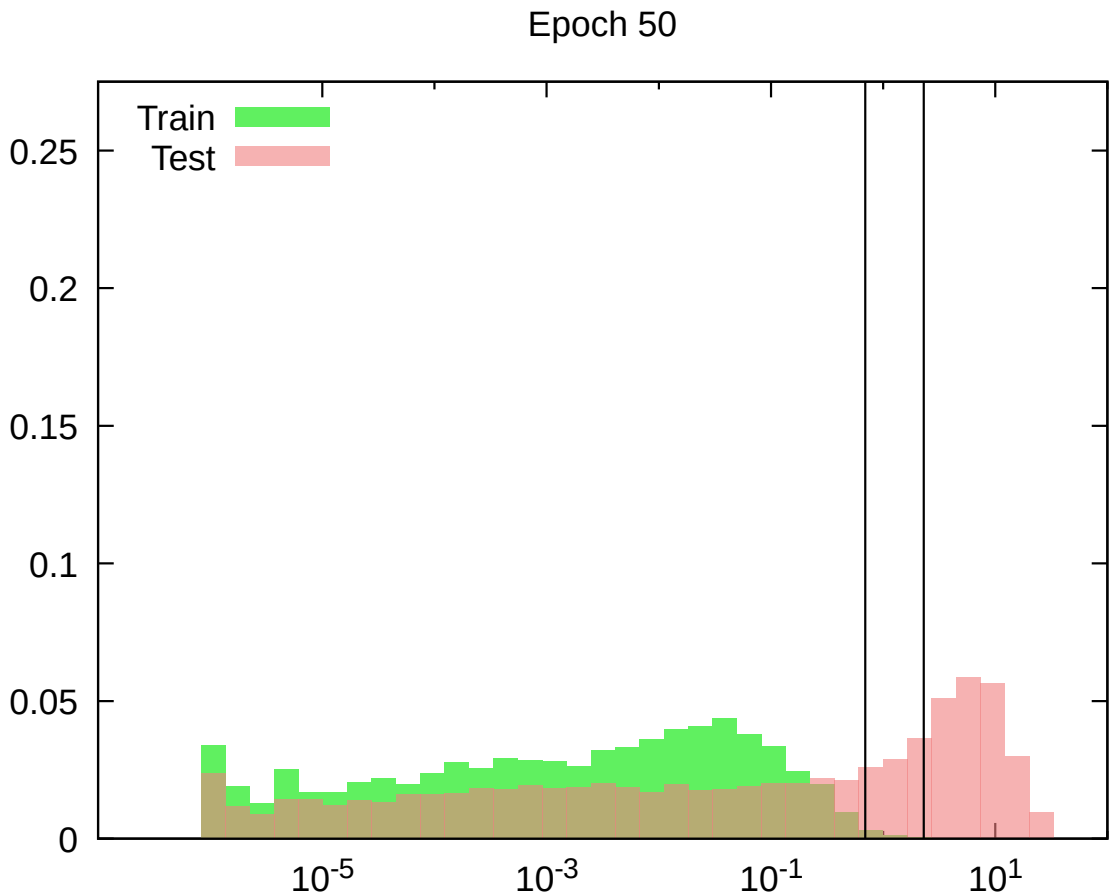
through epochs to visualize the over-fitting.

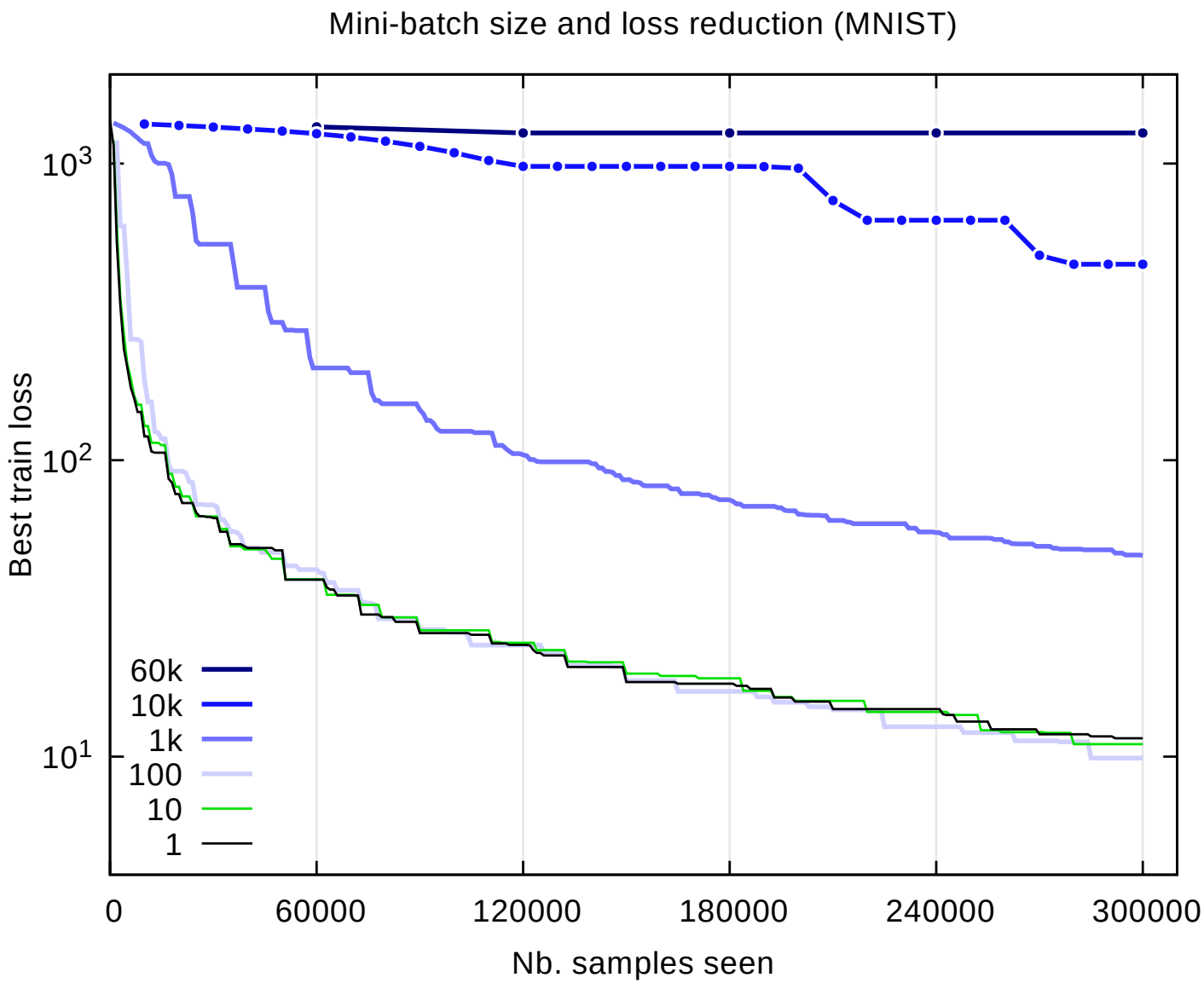


We can plot the train and test distributions of the per-sample loss

$$\ell = -\log\left(\frac{\exp(f_Y(X;w))}{\sum_k \exp(f_k(X;w))}\right)$$

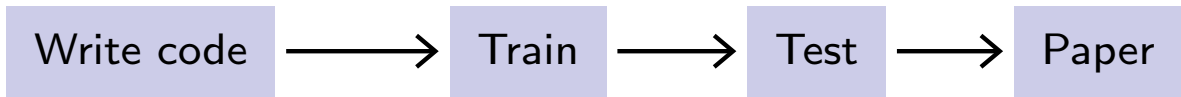
through epochs to visualize the over-fitting.



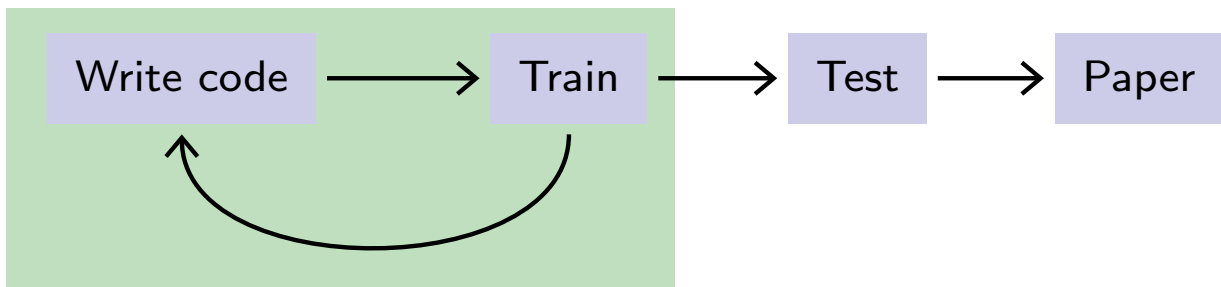

Mini-batch size and loss reduction (MNIST)
Best train loss
10^3
10^2
10^1
60k
10k
1k
100
10
1
0
60000
120000
180000
240000
300000
Nb. samples seen

Model Pipelines

The ideal development cycle is

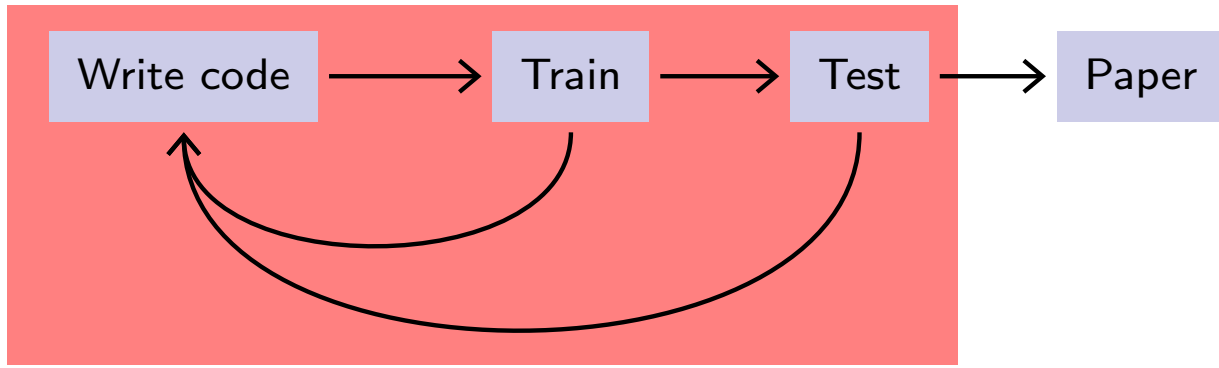


or in practice something like

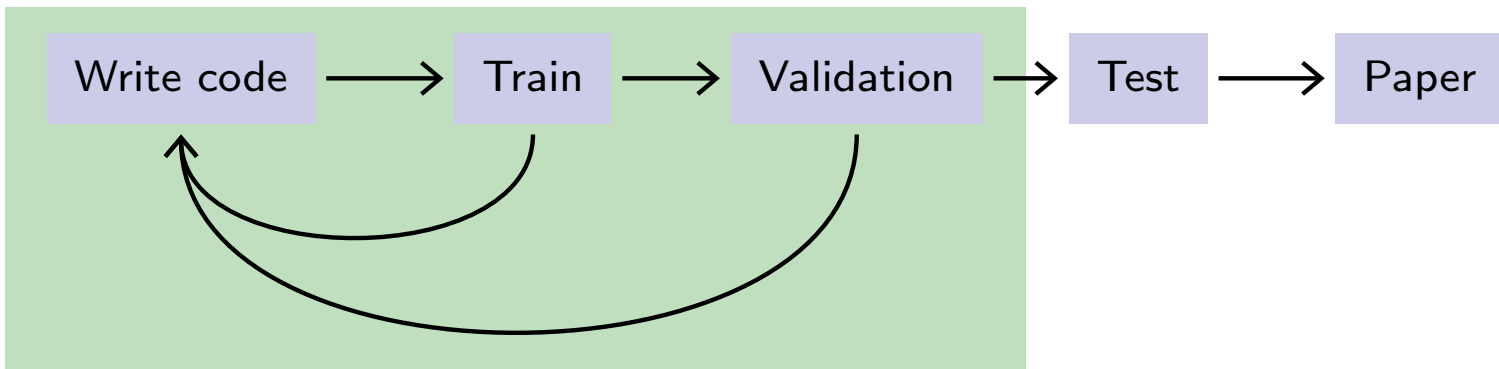


There may be over-fitting, but it does not bias the final performance evaluation.

Unfortunately, it often looks like

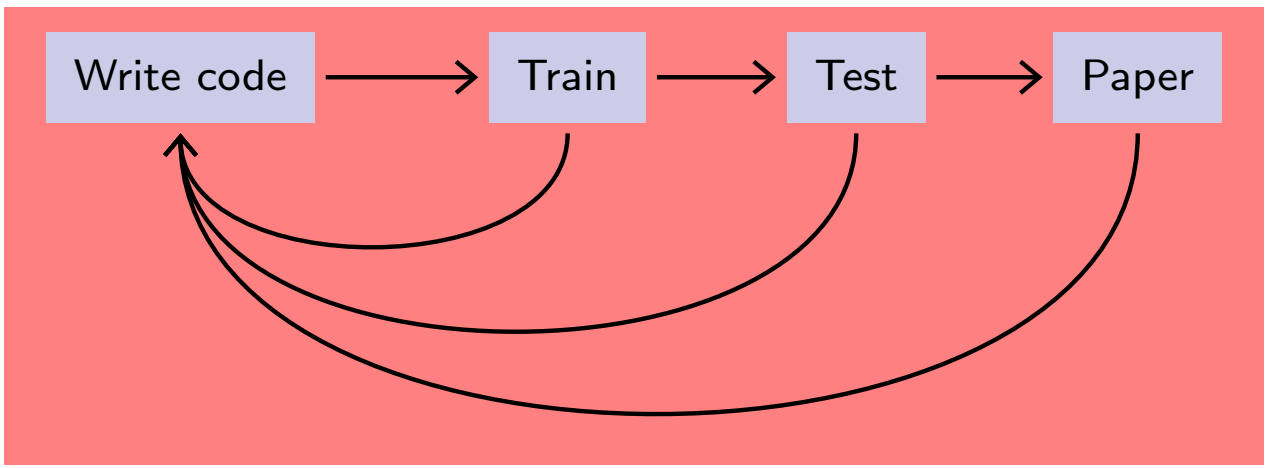


This should be avoided at all costs. The standard strategy is to have a separate validation set for the tuning.



Some data-sets (MNIST!) have been used by thousands of researchers, over millions of experiments, in hundreds of papers.

The global overall process looks more like



"Cheating" in machine learning, from bad to "are you kidding?":

- "Early evaluation stopping",
- meta-parameter (over-)tuning,
- data-set selection,
- algorithm data-set specific clauses,
- seed selection.

Top-tier conferences are demanding regarding experiments, and are biased against "complicated" pipelines.

The community pushes toward accessible implementations, reference data-sets, leader boards, and constant upgrades of benchmarks.

Thank you - Questions?