Deep Learning

A journey from feature extraction and engineering to end-to-end pipelines

Part 2: Neural Networks

Andrei Bursuc

With slides from A. Karpathy, F. Fleuret, J. Johnson, S. Yeung, G. Louppe, Y. Avrithis ...

Empirical risk minimization

Consider a function $f: \mathrm{X} \to \mathrm{Y}$ produced by some learning algorithm. The predictions of this function can be evaluated through a loss

$$\ell : \mathrm{Y} \times \mathrm{Y} \to \mathrm{R}$$

such that $\ell(y, f(\mathbf{x})) \geq 0$ measures how close is the prediction $f(\mathbf{x})$ from y.

For example,

- for classification:

$$\ell(y, f(\mathbf{x})) = \mathbf{1}_{y \neq f(\mathbf{x})}$$

- for regression:

$$\ell(y, f(\mathbf{x})) = (y - f(\mathbf{x}))^2$$

Let us denote as F the hypothesis space, i.e. the set of all functions f than can be produced by the chosen learning algorithm.

We are looking for a function $f \in \mathrm{F}$ with a small expected risk (or generalization error)

$$R(f) = \mathrm{E}_{(\mathbf{x}, y) \sim P(X, Y)}[\ell(y, f(\mathbf{x}))].$$

This means that for a given data generating distribution and for a given hypothesis space, the optimal model is

$$f_* = \arg \min_{f \in \mathrm{F}} R(f).$$

Unfortunately, since $P(X, Y)$ is unknown, the expected risk cannot be evaluated and the optimal model cannot be determined.

However, given training data $\mathbf{d} = \{(\mathbf{x}_i, y_i) \,|\, i = 1, \ldots, N\}$, we can compute an estimate, the empirical risk (or training error)

$$\hat{R}(f, \mathbf{d}) = \frac{1}{N} \sum_{(\mathbf{x}_i, y_i) \in \mathbf{d}} \ell(y_i, f(\mathbf{x}_i)).$$

This estimate can be used for finding a good enough approximation of f_*, giving rise to the empirical risk minimization principle:

$$f_*^{\mathbf{d}} = \arg \min_{f \in \mathrm{F}} \hat{R}(f, \mathbf{d})$$

Most machine learning algorithms, including neural networks, implement empirical risk minimization.

Under regularity assumptions, empirical risk minimizers converge:

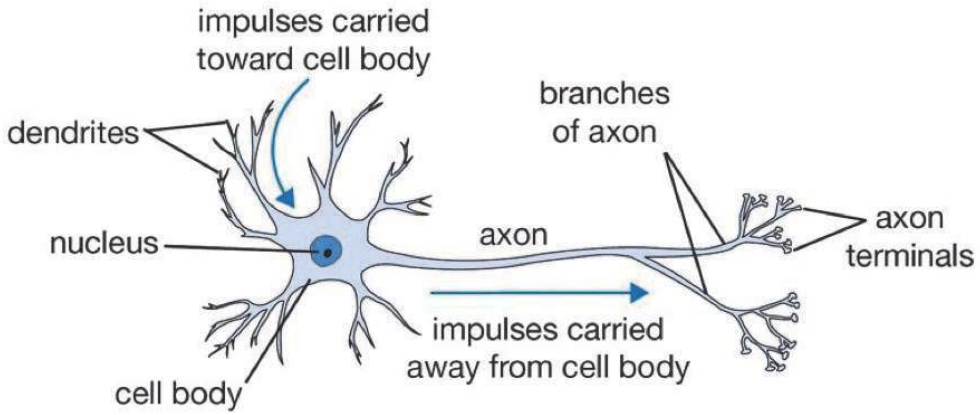
$$\lim_{N \to \infty} f_*^{\mathbf{d}} = f_*$$

This is why tuning the parameters of the model to make it work on the training data is a reasonable thing to do.

Neural Networks

The neuron

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

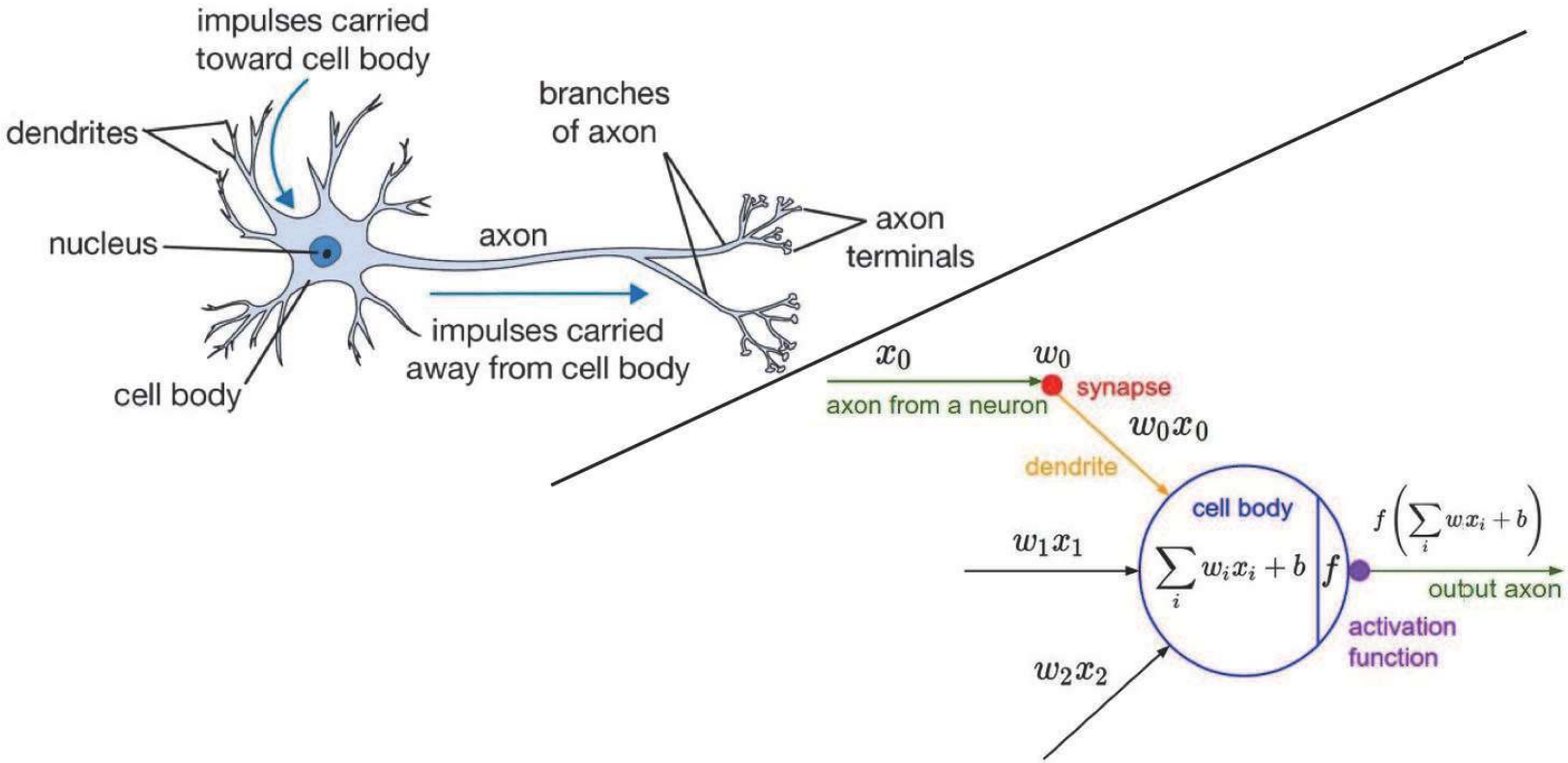


- In fact there several types of neurons with different functions and the metaphor does not hold everywhere

Slide credit: A. Karpathy

The neuron

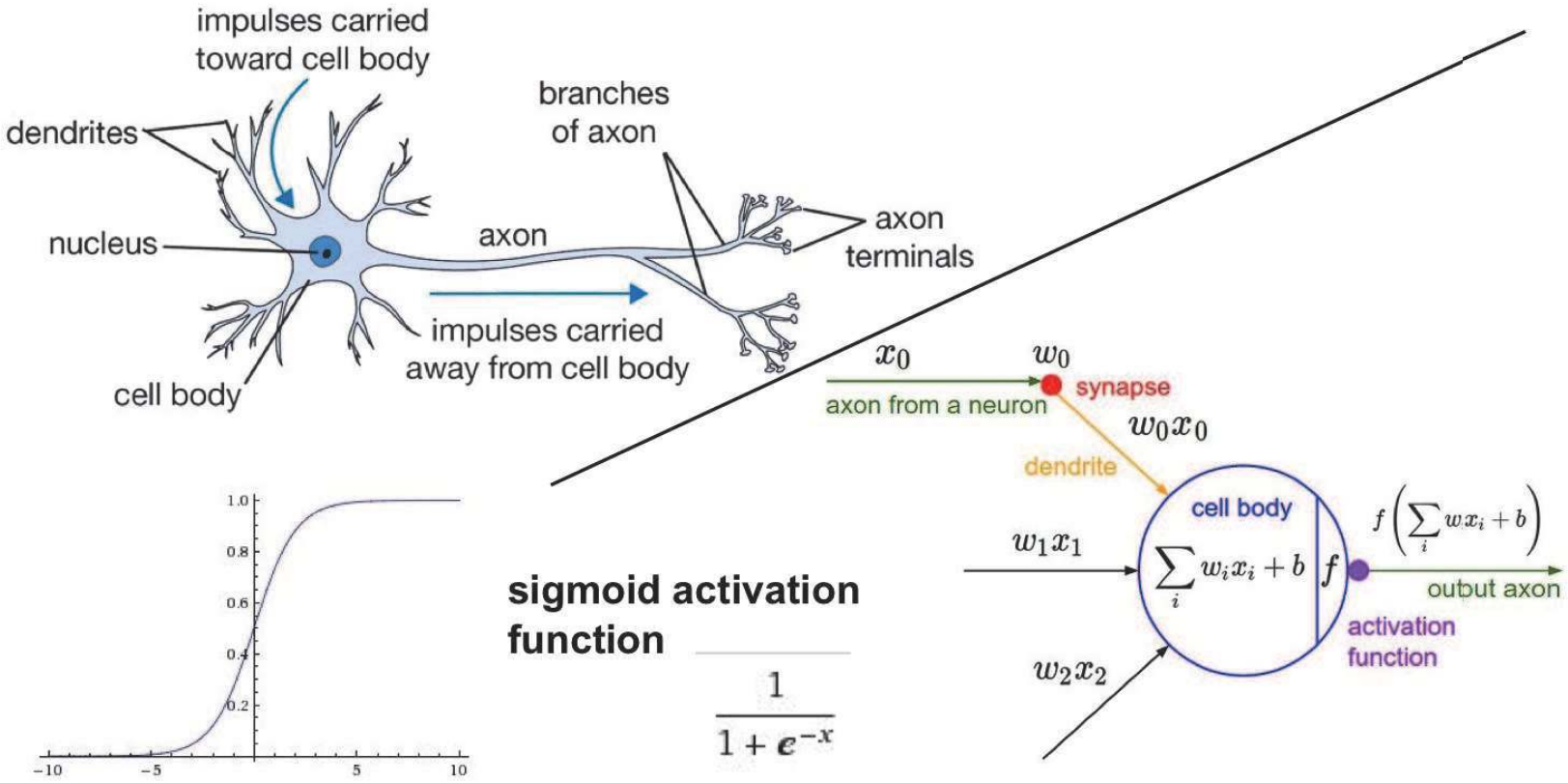
Inspired by neuroscience and human brain, but resemblances do not go too far



Slide credit: A. Karpathy

Neural Networks

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



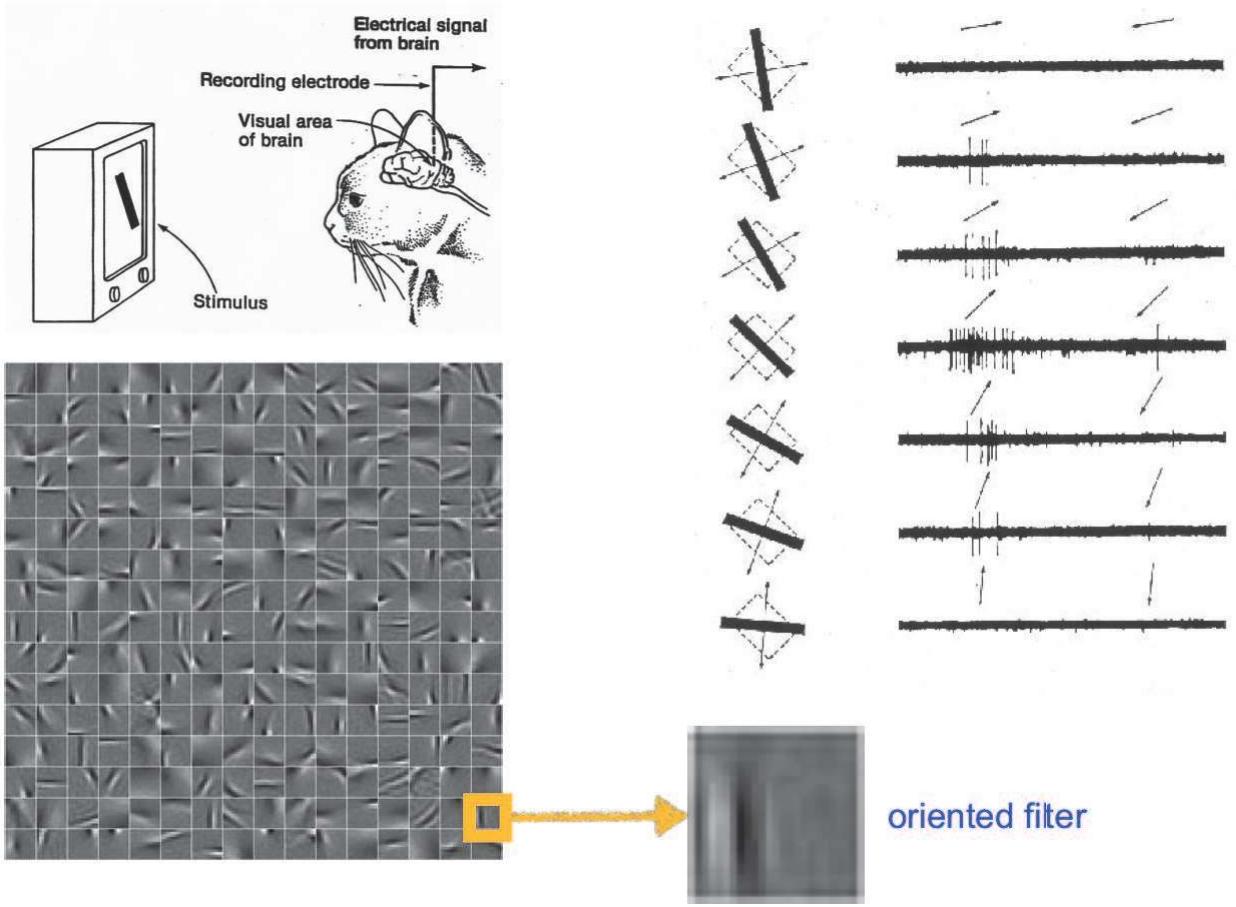
Slide credit: A. Karpathy

Discovery of oriented cells in the visual cortex

Find out more from video

Hubel& Wiesel, 1959

Discovery of oriented cells in the visual cortex



Hubel& Wiesel, 1959

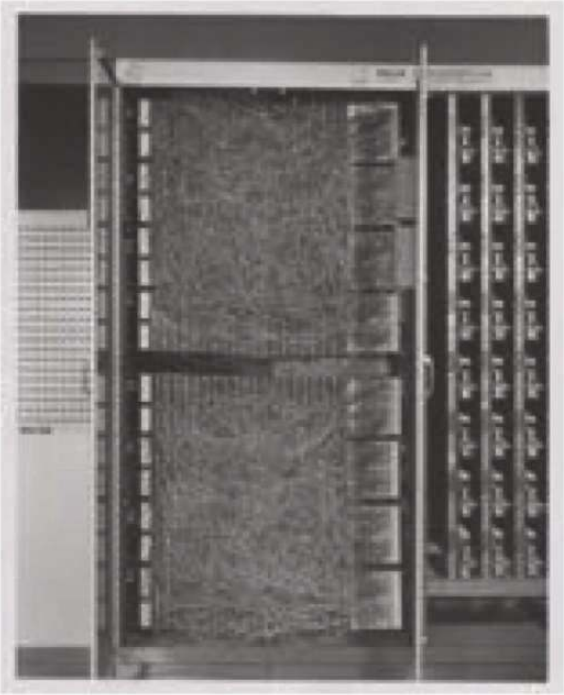
Mark I Perceptron

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

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

update rule:

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



Rosenblatt, 1957

Threshold Logic Unit

The Threshold Logic Unit (McCulloch and Pitts, 1943) was the first mathematical model for a neuron. Assuming Boolean inputs and outputs, it is defined as:

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

This unit can implement:

- $\text{or}(a, b) = 1_{\{a+b-0.5 \geq 0\}}$
- $\text{and}(a, b) = 1_{\{a+b-1.5 \geq 0\}}$
- $\text{not}(a) = 1_{\{-a+0.5 \geq 0\}}$

Therefore, any Boolean function can be built which such units.

Perceptron

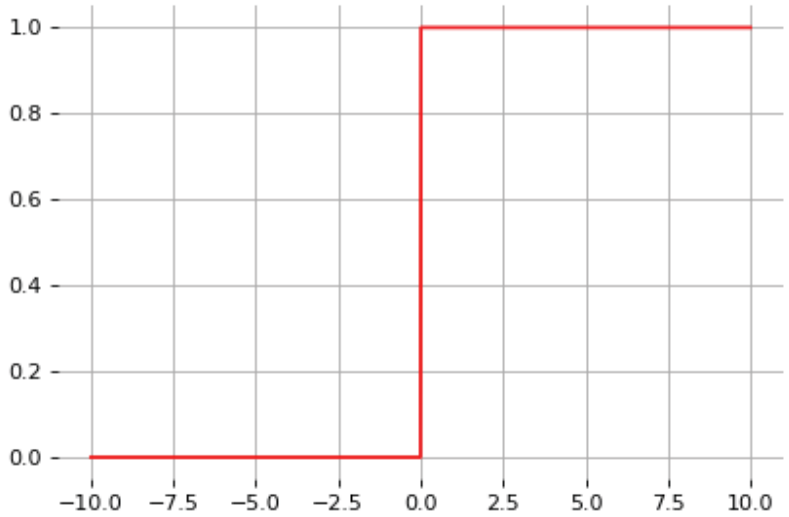
The perceptron (Rosenblatt, 1957) is very similar, except that the inputs are real:

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

- This model was originally motivated by biology, with w_i being synaptic weights and x_i and f firing rates.
- This is a cartoonesque biological model.

Let us define the activation function:

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



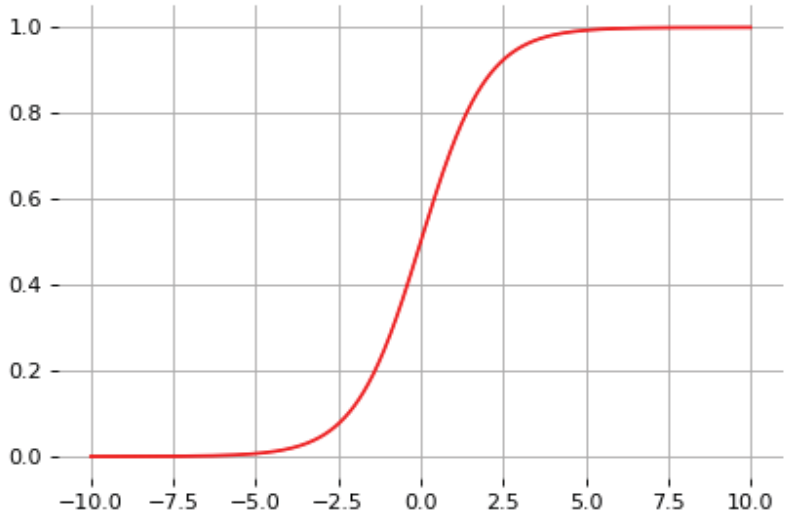
Therefore, the perceptron classification rule can be rewritten as

$$f(\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x} + b).$$

Note that the sigmoid function

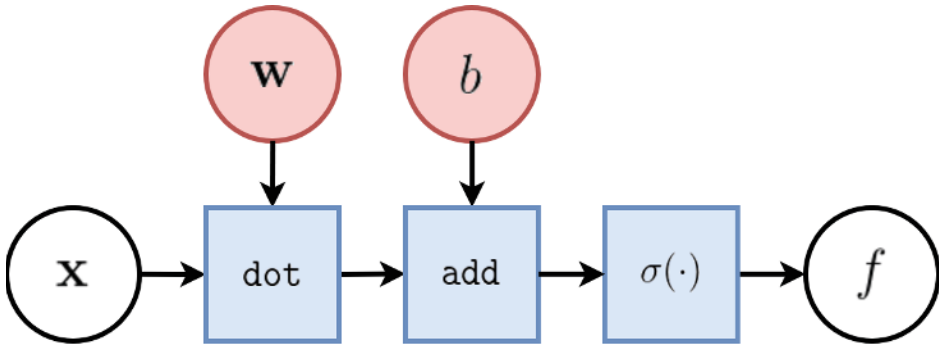
$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

looks like a soft heavyside:



Therefore, the overall model $f(\mathbf{x}; \mathbf{w}, b) = \sigma(\mathbf{w}^T\mathbf{x} + b)$ is very similar to the perceptron.

In terms of tensor operations, the computational graph of f can be represented as:



where

- white nodes correspond to inputs and outputs;
- red nodes correspond to model parameters;
- blue nodes correspond to intermediate operations, which themselves produce intermediate output values (not represented).

This unit is the core component all neural networks!

Gradient descent

Let $\mathrm{L}(\theta)$ denote a loss function defined over model parameters θ (e.g., $\mathbf{w}$ and b).

To minimize $\mathrm{L}(\theta)$, gradient descent uses local linear information to iteratively move towards a (local) minimum.

For $\theta_0 \in \mathrm{R}^d$, a first-order approximation around θ_0 can be defined as

$$\hat{\mathrm{L}}(\theta_0 + \epsilon) = \mathrm{L}(\theta_0) + \epsilon^T \nabla_\theta \mathrm{L}(\theta_0) + \frac{1}{2\eta} ||\epsilon||^2.$$

A minimizer of the approximation $\hat{\mathrm{L}}(\theta_0 + \epsilon)$ is given for

$$\begin{aligned}\nabla_\epsilon \hat{\mathrm{L}}(\theta_0 + \epsilon) &= 0 \\ &= \nabla_\theta \mathrm{L}(\theta_0) + \frac{1}{\eta}\epsilon,\end{aligned}$$

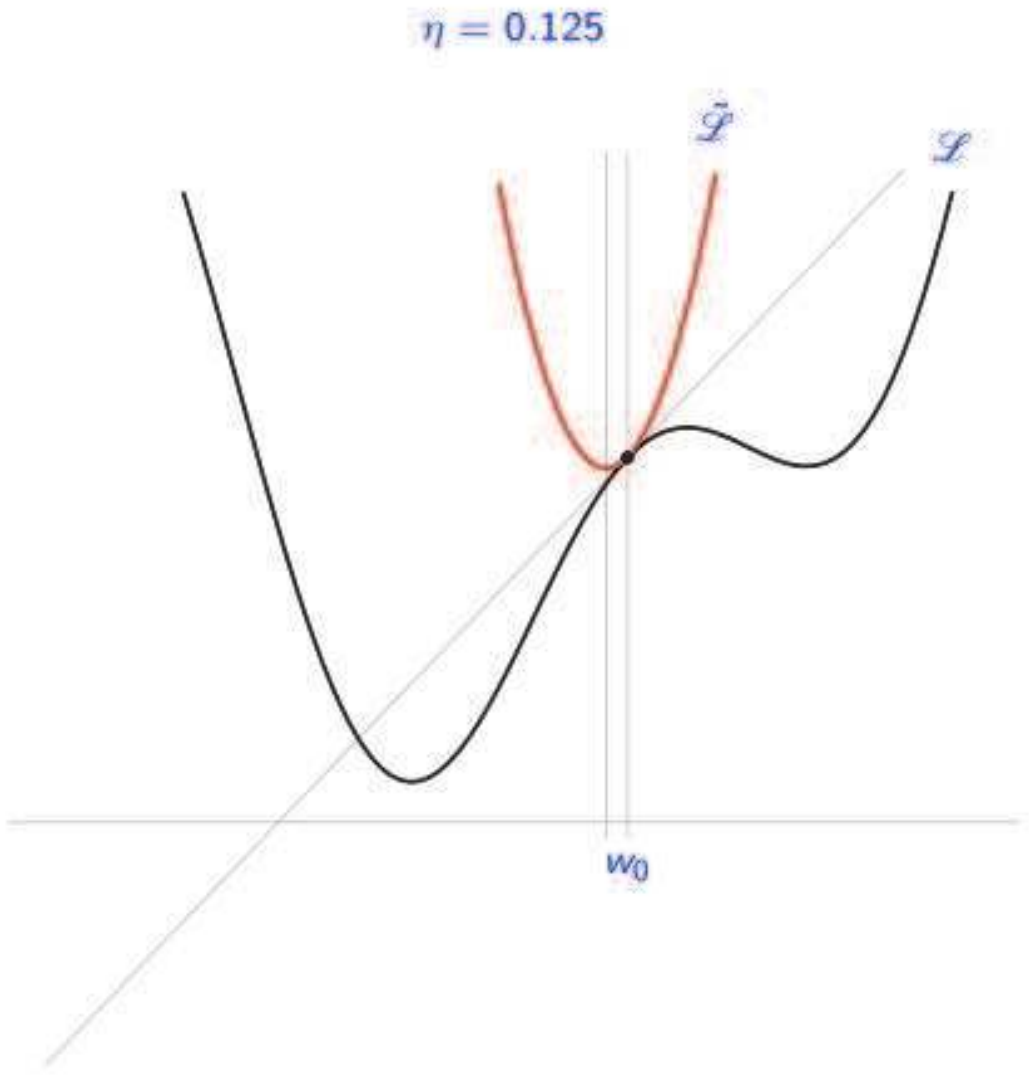
which results in the best improvement for the step $\epsilon = -\eta \nabla_\theta \mathrm{L}(\theta_0)$.

Therefore, model parameters can be updated iteratively using the update rule:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathrm{L}(\theta_t)$$

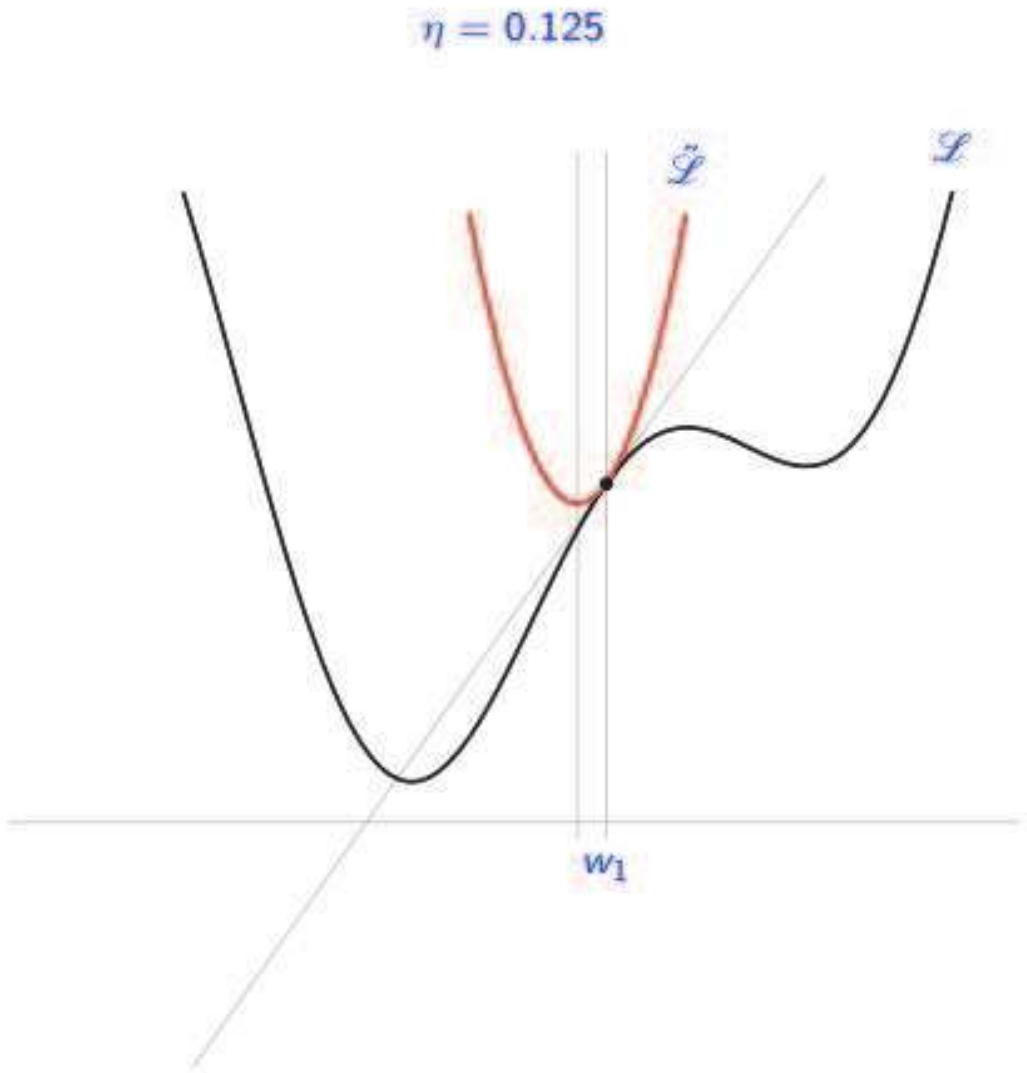
Notes:

- θ_0 are the initial parameters of the model;
- η is the learning rate;
- both are critical for the convergence of the update rule.



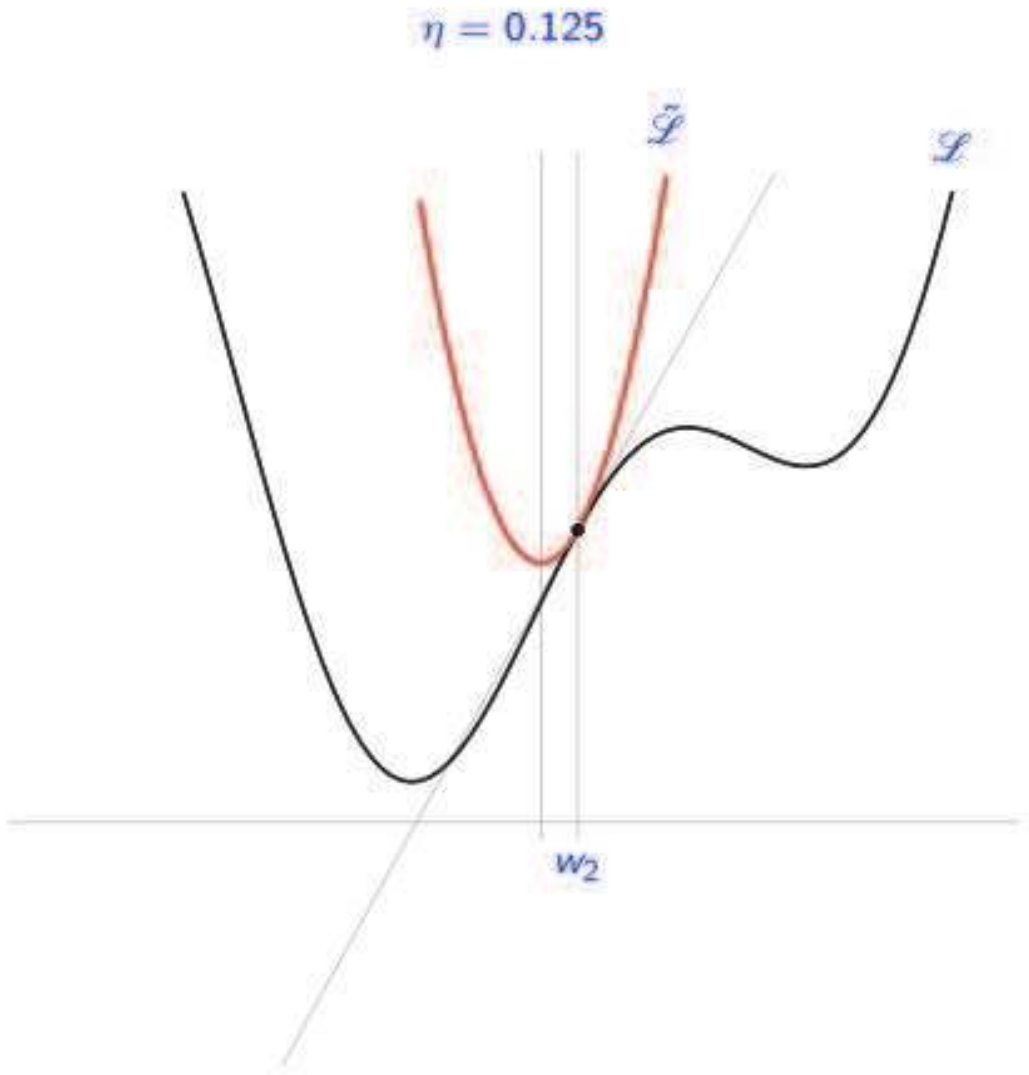
Example 1: Convergence to a global minima

Slide credit: F. Fleuret



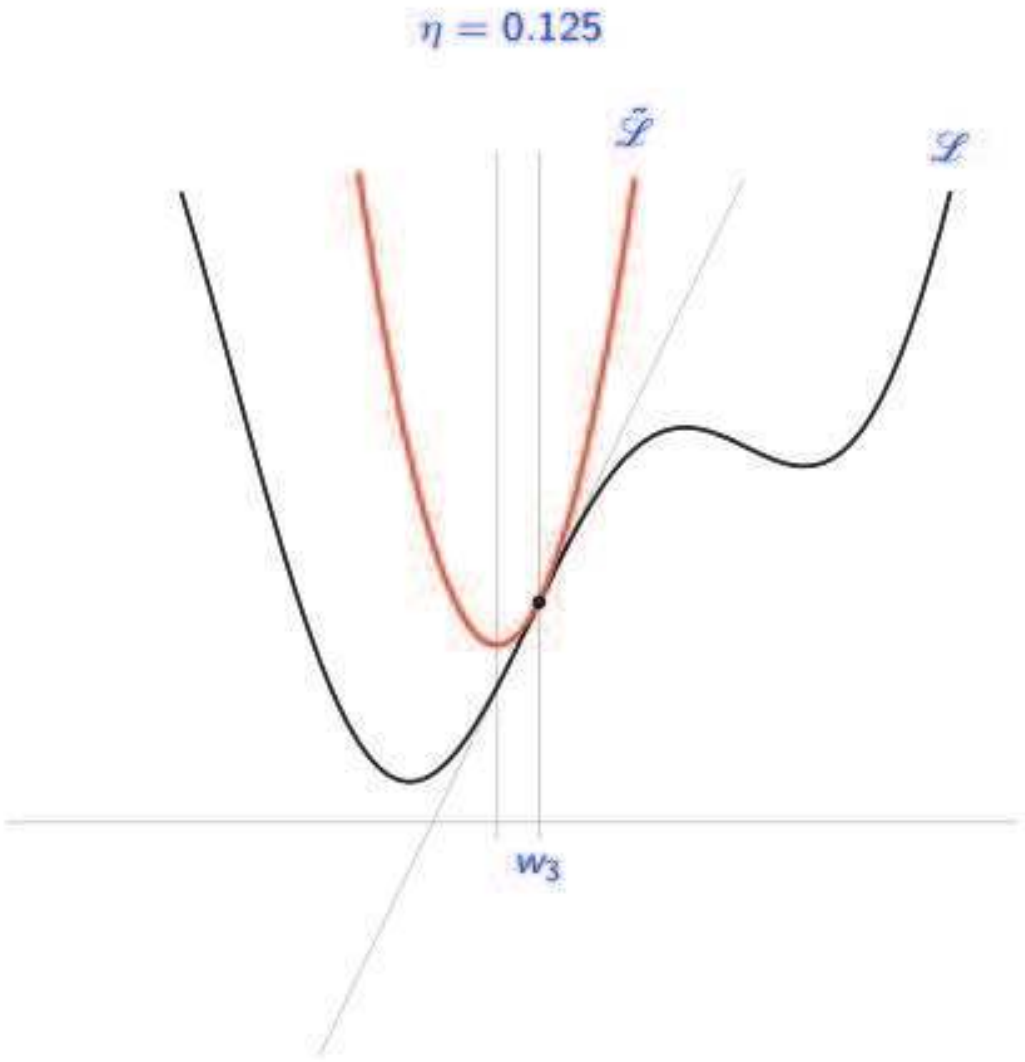
Example 1: Convergence to a global minima

Slide credit: F. Fleuret



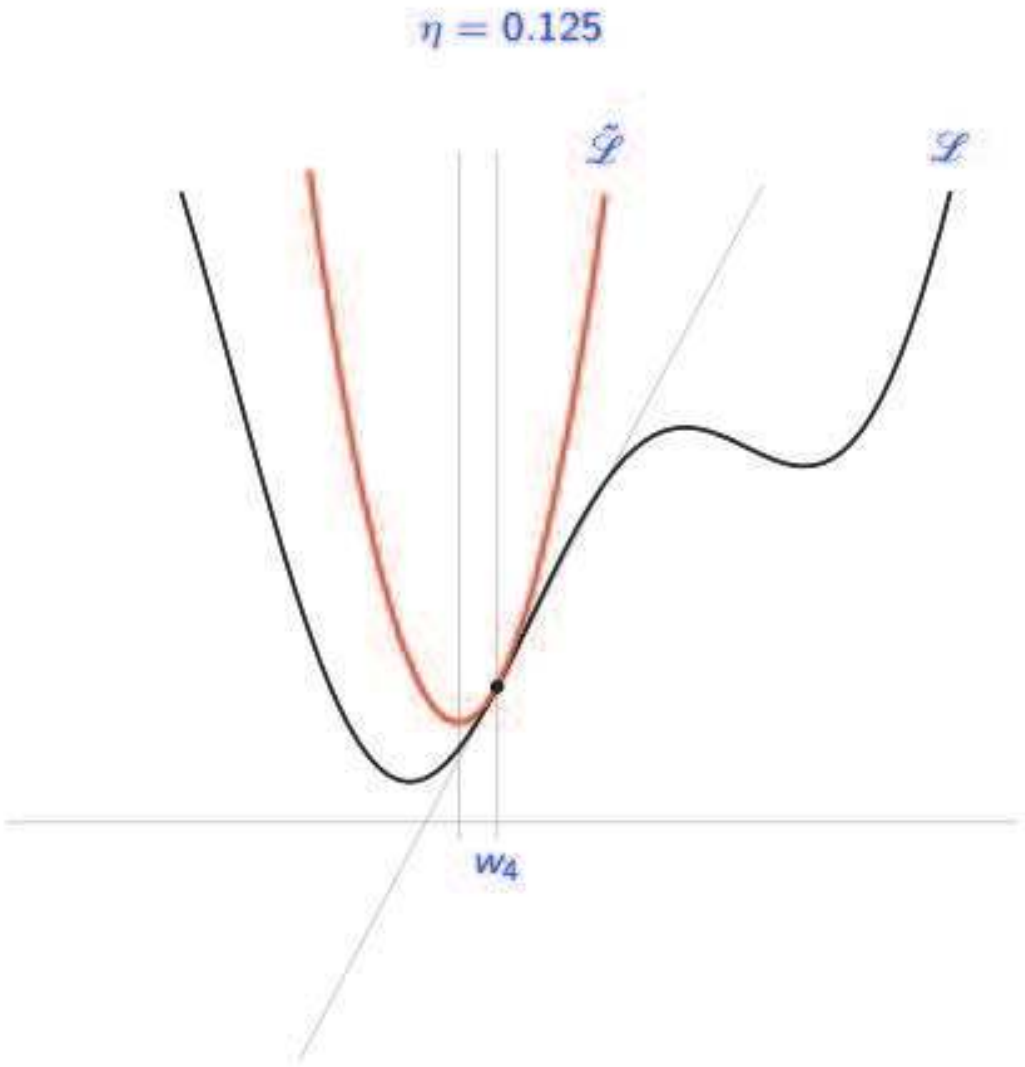
Example 1: Convergence to a global minima

Slide credit: F. Fleuret



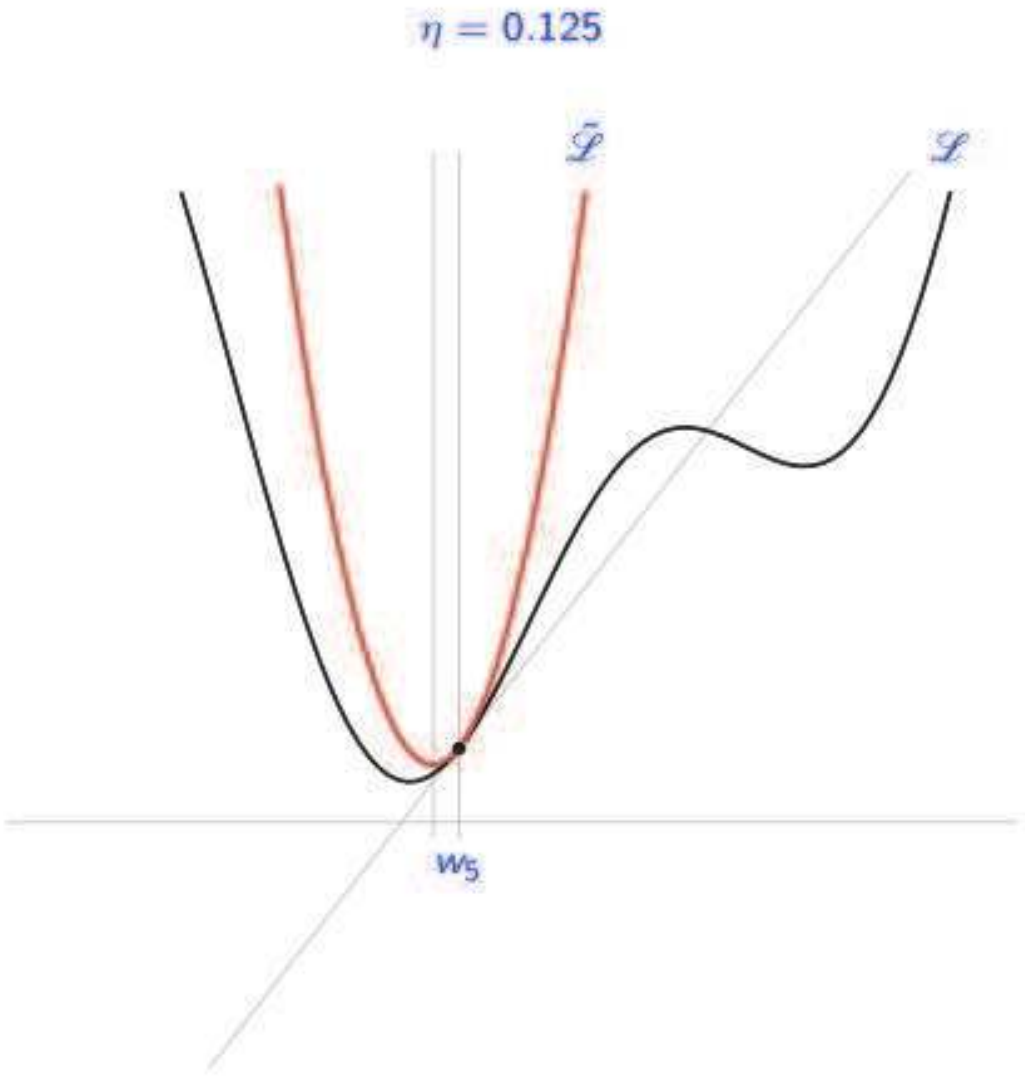
Example 1: Convergence to a global minima

Slide credit: F. Fleuret



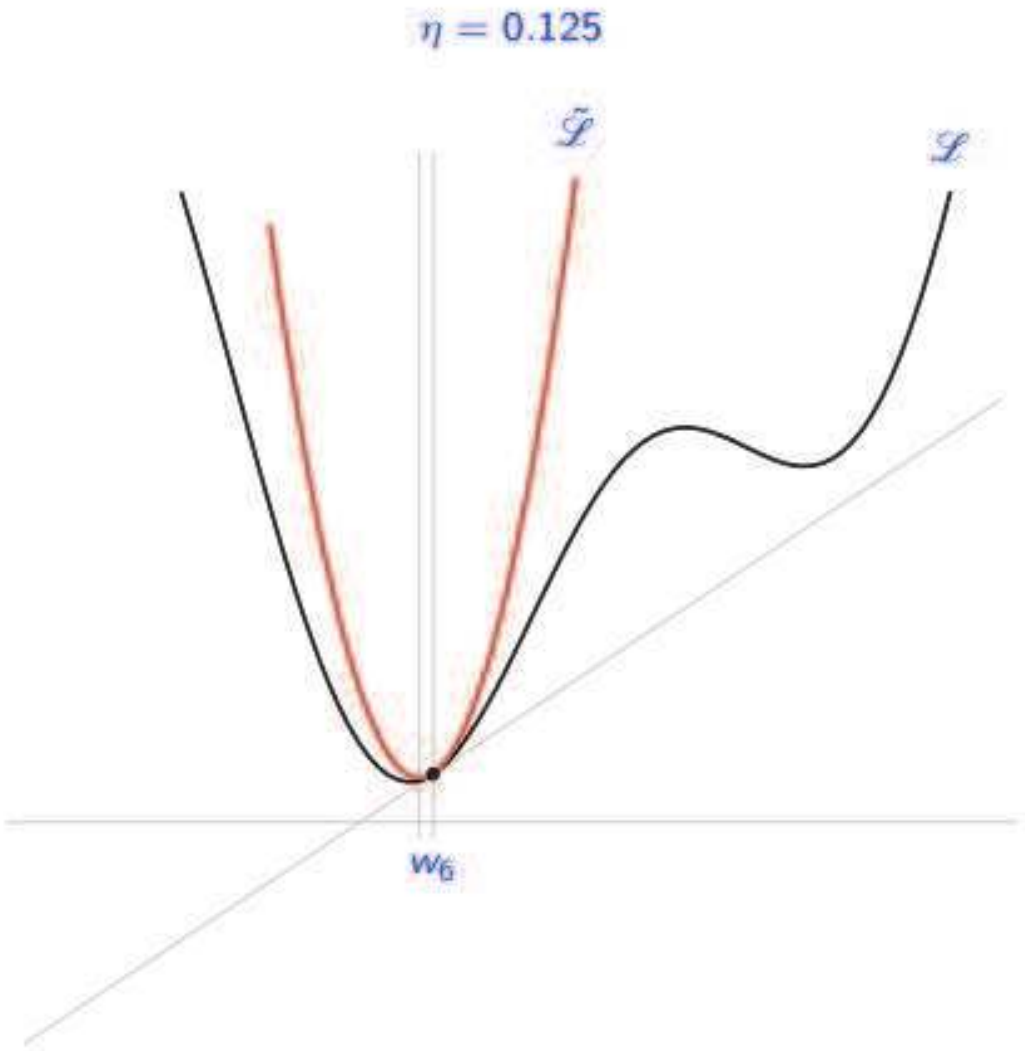
Example 1: Convergence to a global minima

Slide credit: F. Fleuret



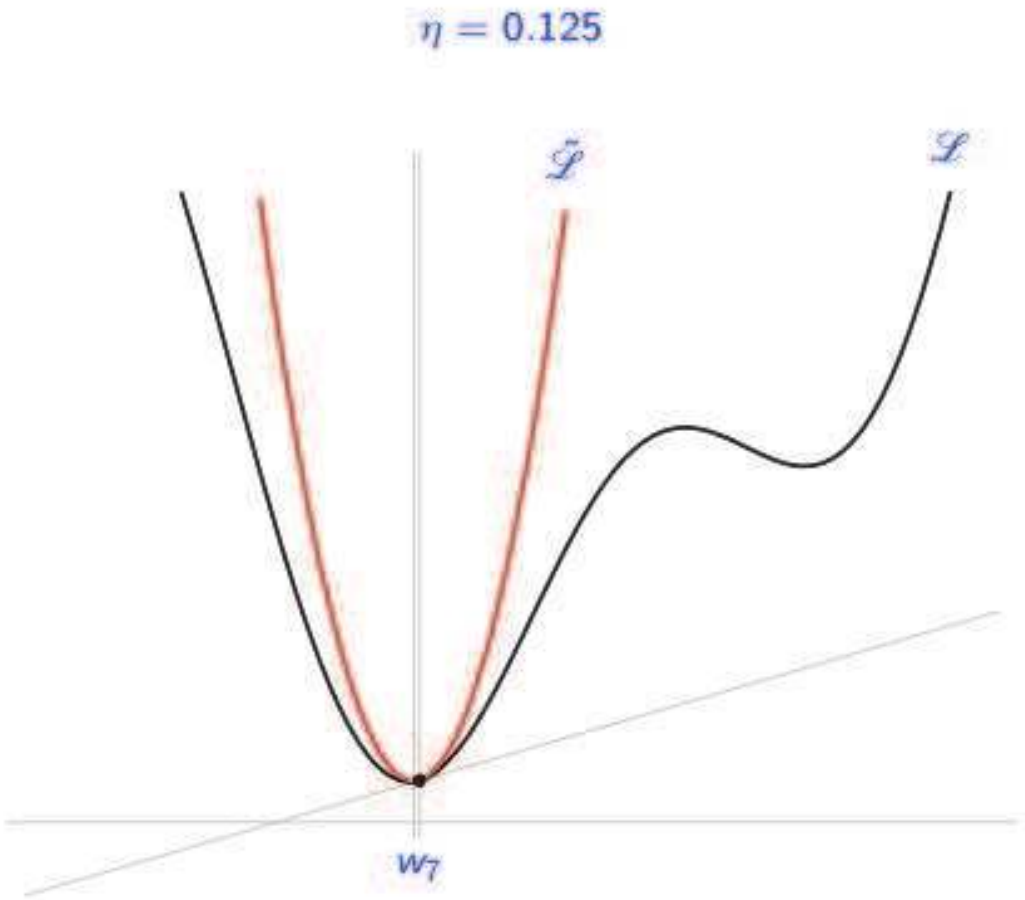
Example 1: Convergence to a global minima

Slide credit: F. Fleuret



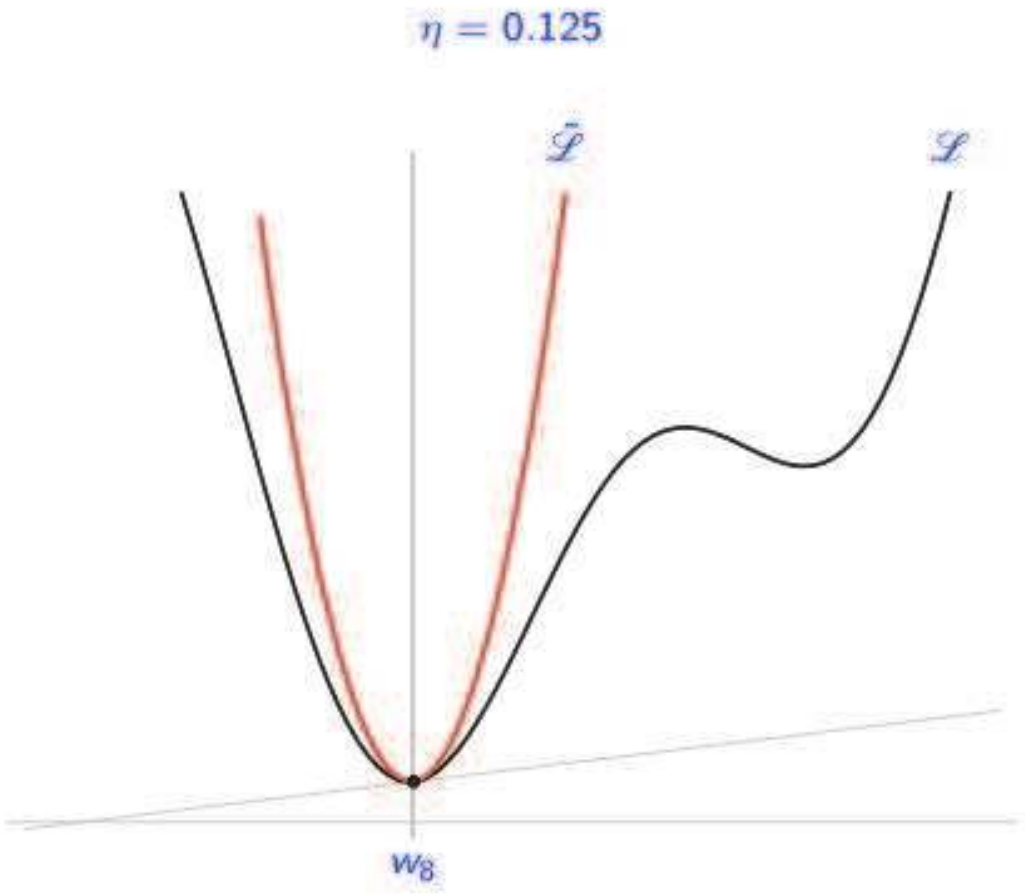
Example 1: Convergence to a global minima

Slide credit: F. Fleuret



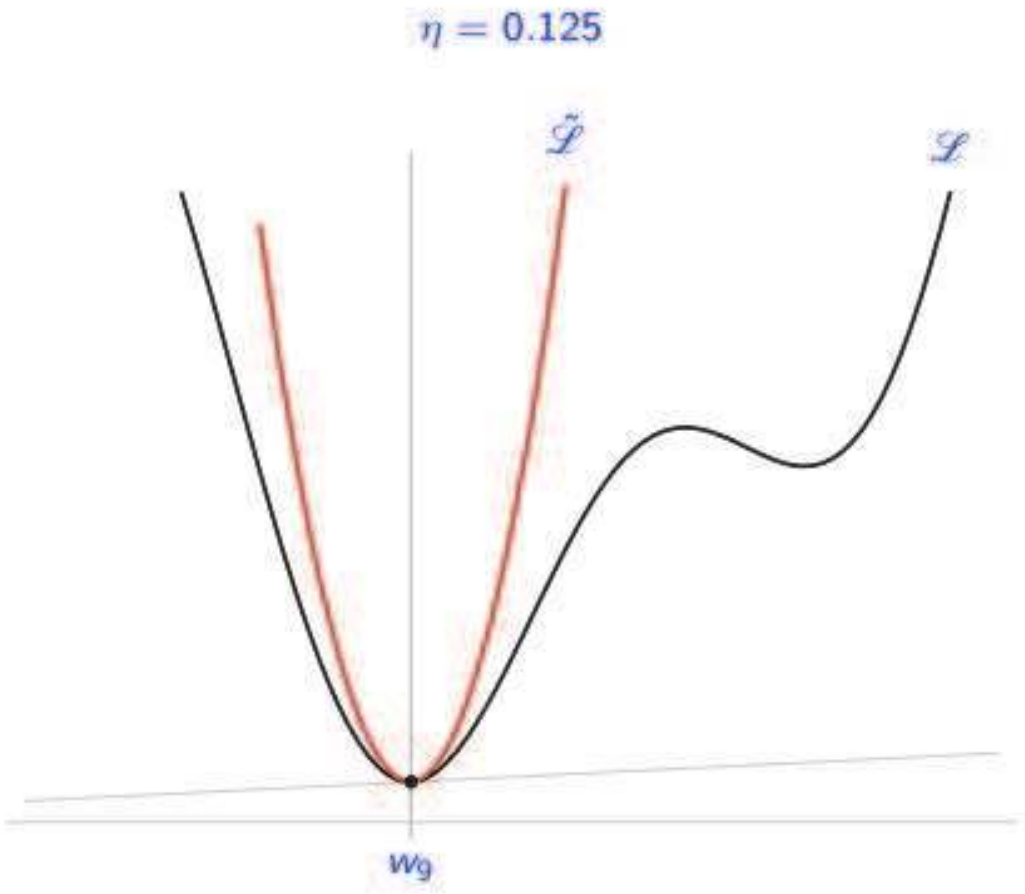
Example 1: Convergence to a global minima

Slide credit: F. Fleuret



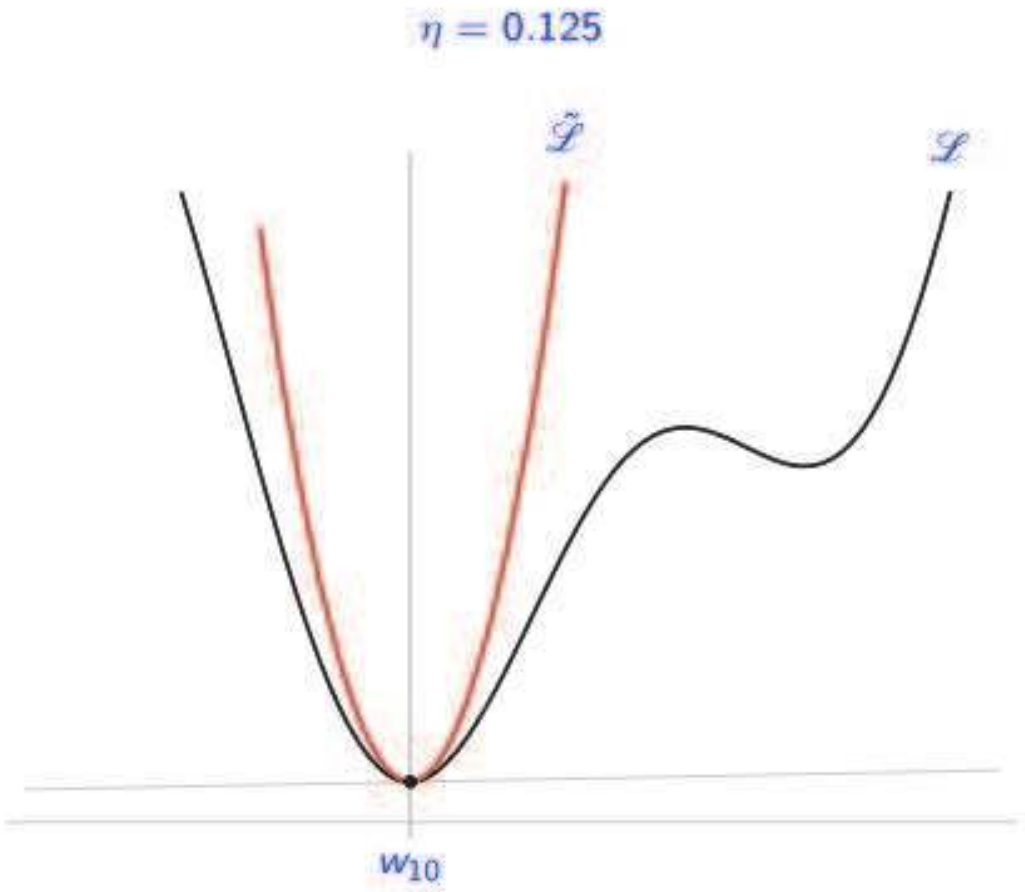
Example 1: Convergence to a global minima

Slide credit: F. Fleuret



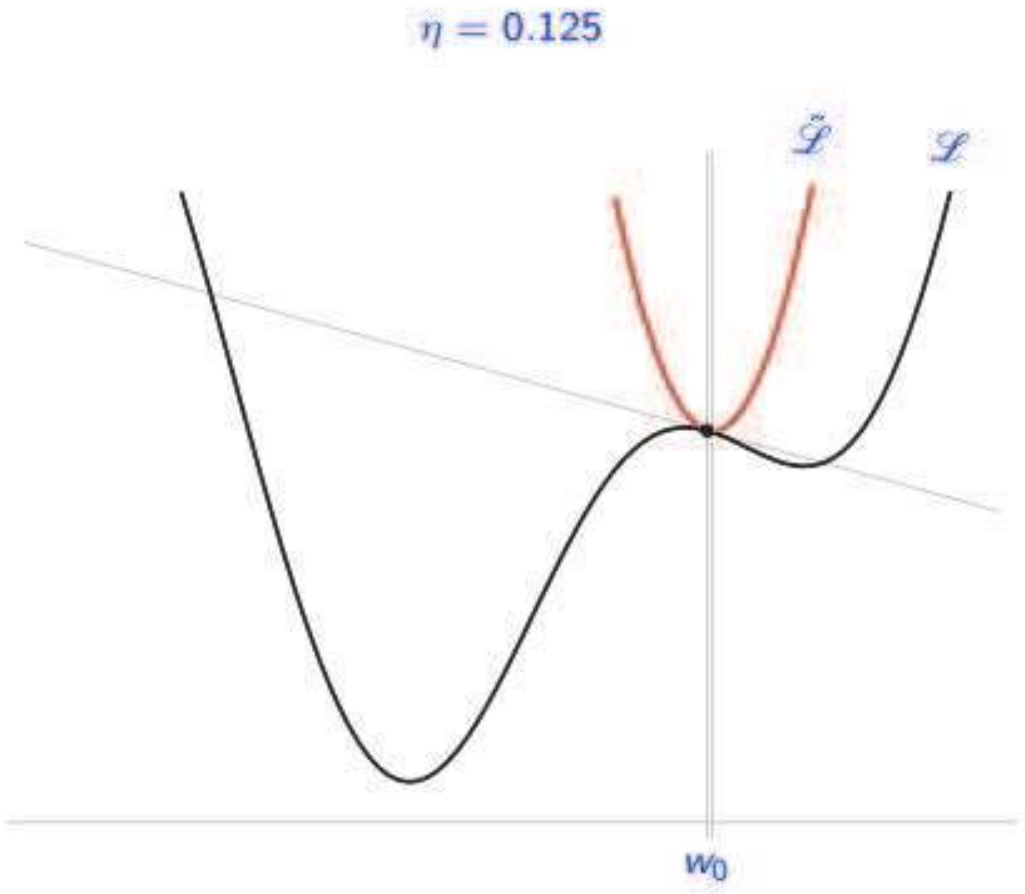
Example 1: Convergence to a global minima

Slide credit: F. Fleuret



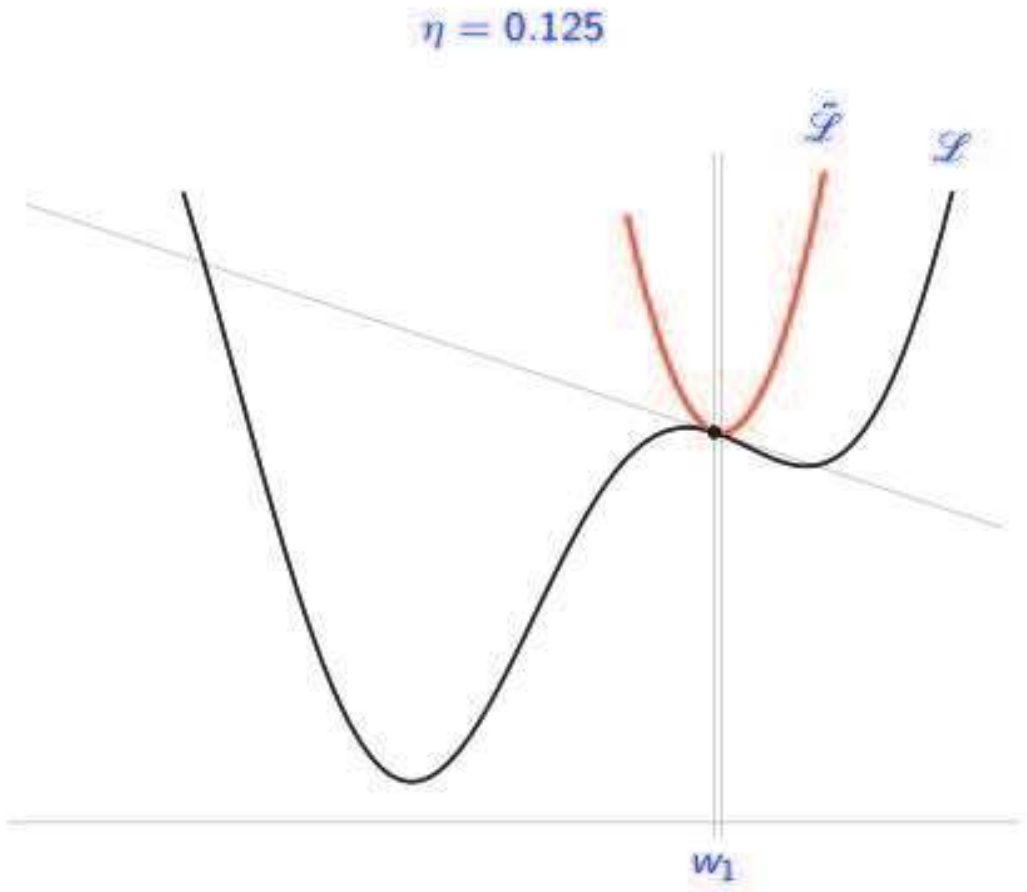
Example 1: Convergence to a global minima

Slide credit: F. Fleuret



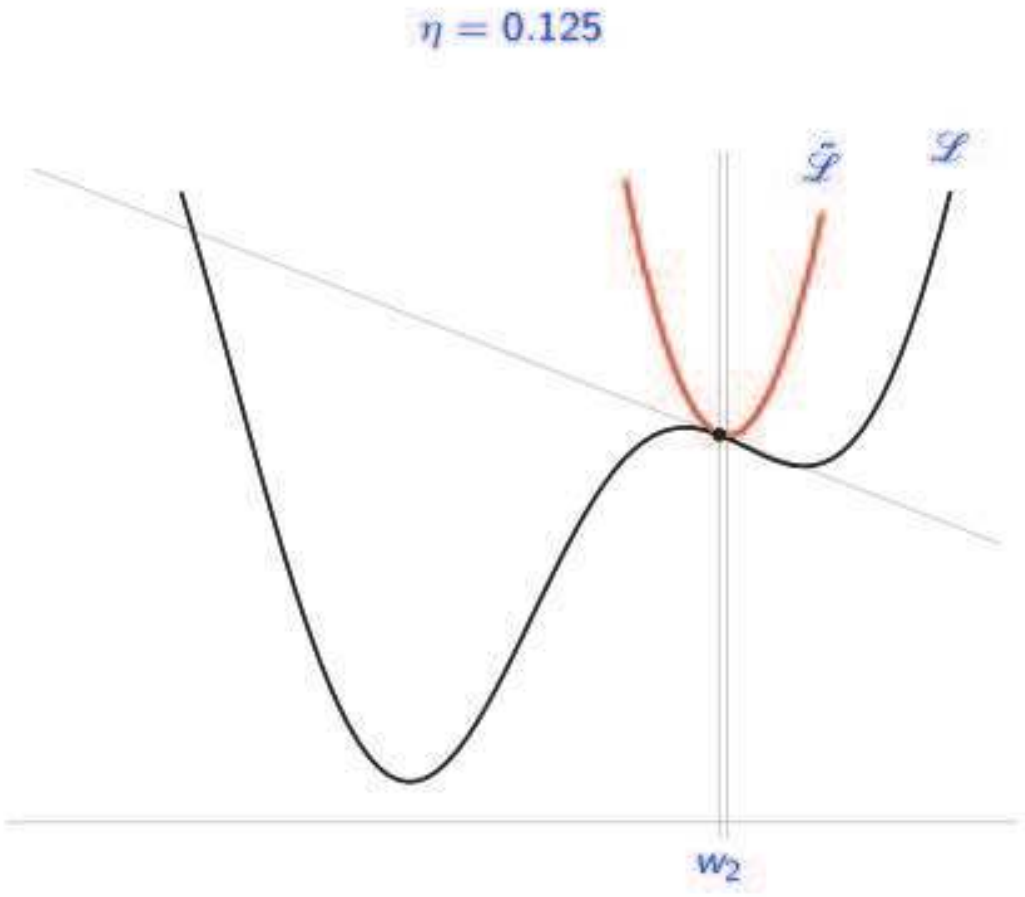
Example 2: Convergence to a local minima

Slide credit: F. Fleuret



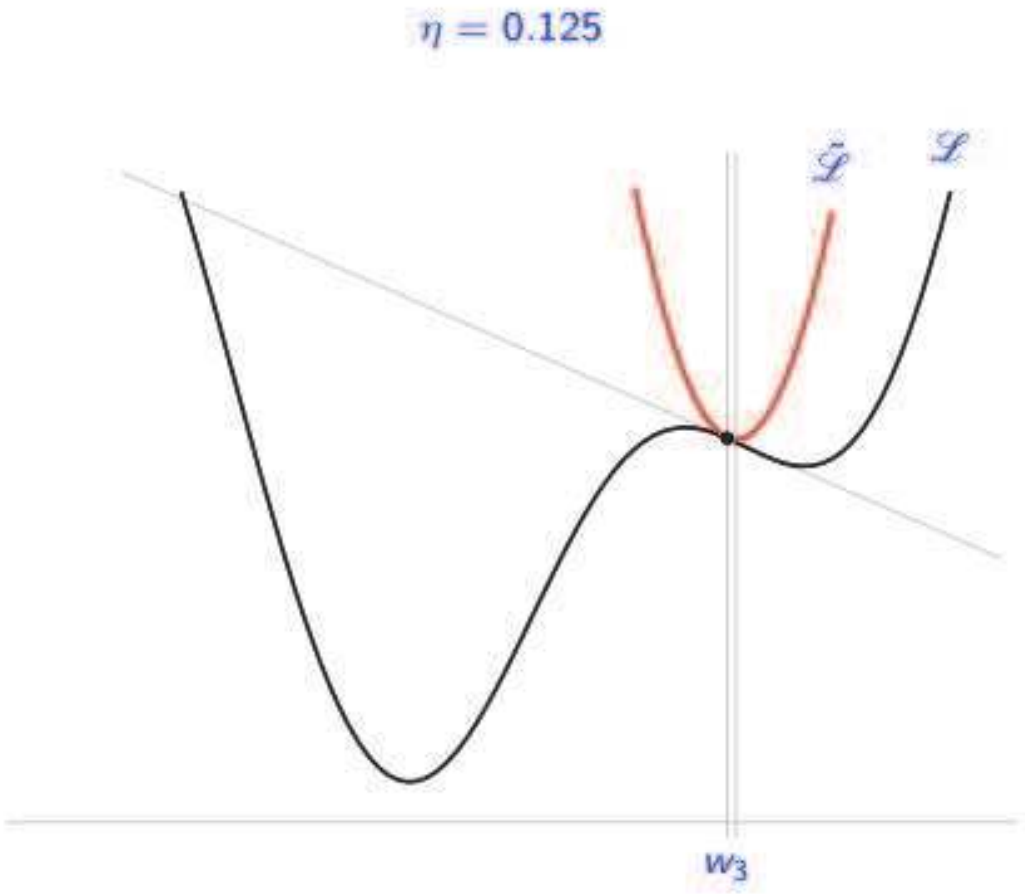
Example 2: Convergence to a local minima

Slide credit: F. Fleuret



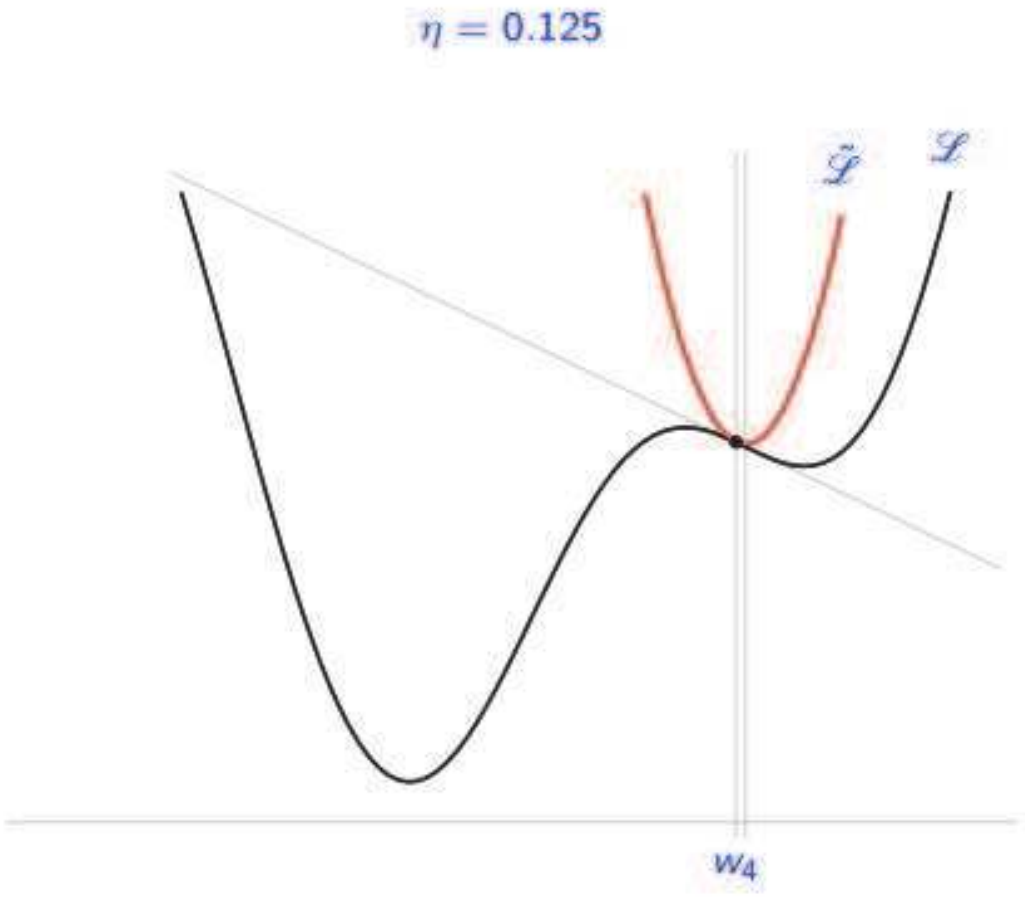
Example 2: Convergence to a local minima

Slide credit: F. Fleuret



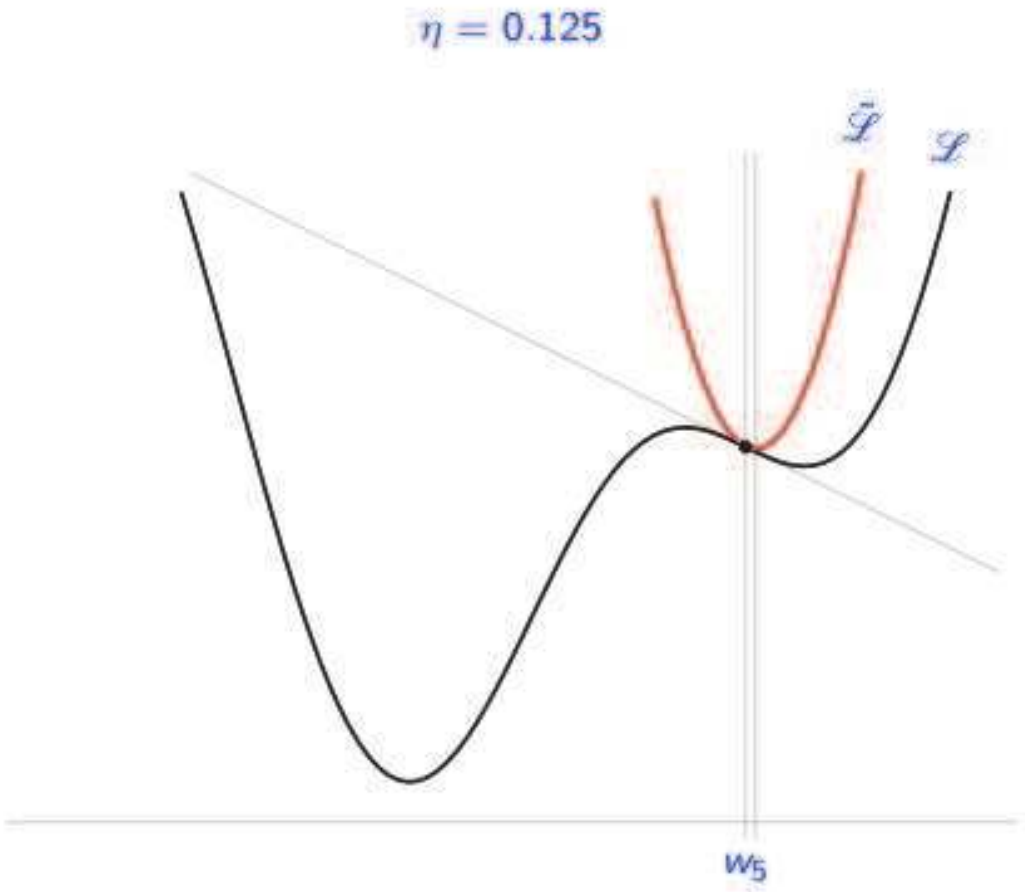
Example 2: Convergence to a local minima

Slide credit: F. Fleuret



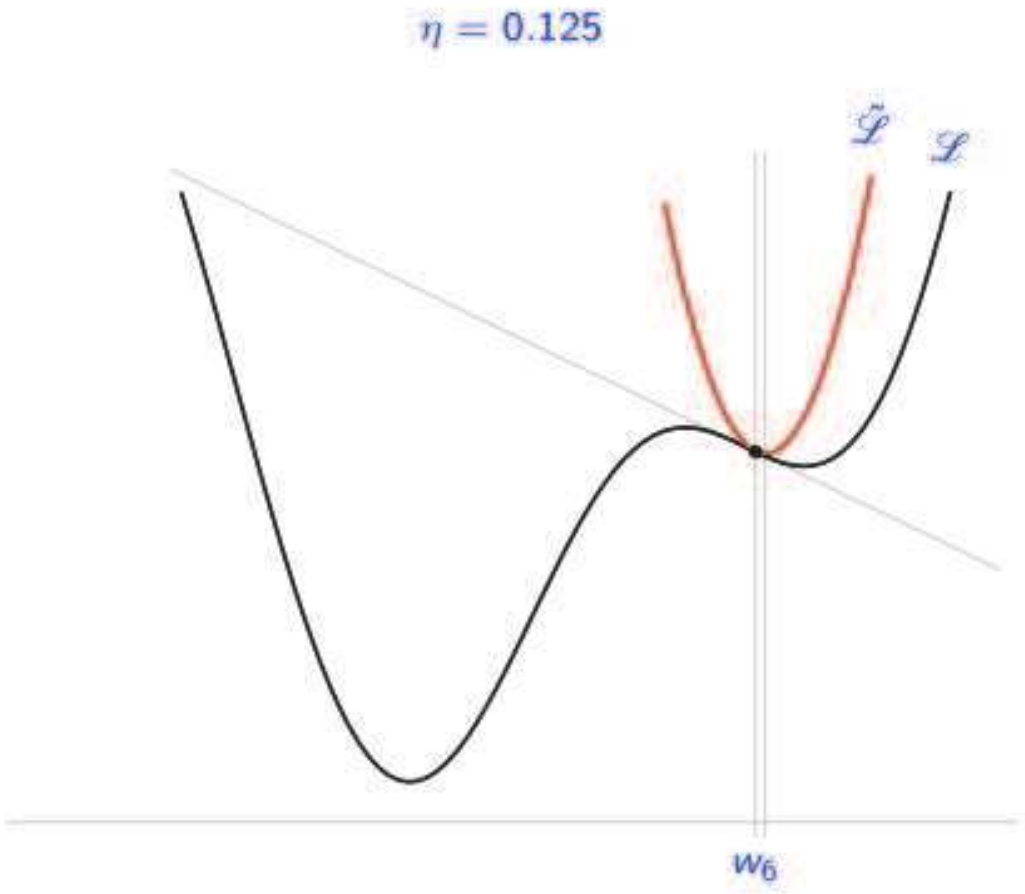
Example 2: Convergence to a local minima

Slide credit: F. Fleuret



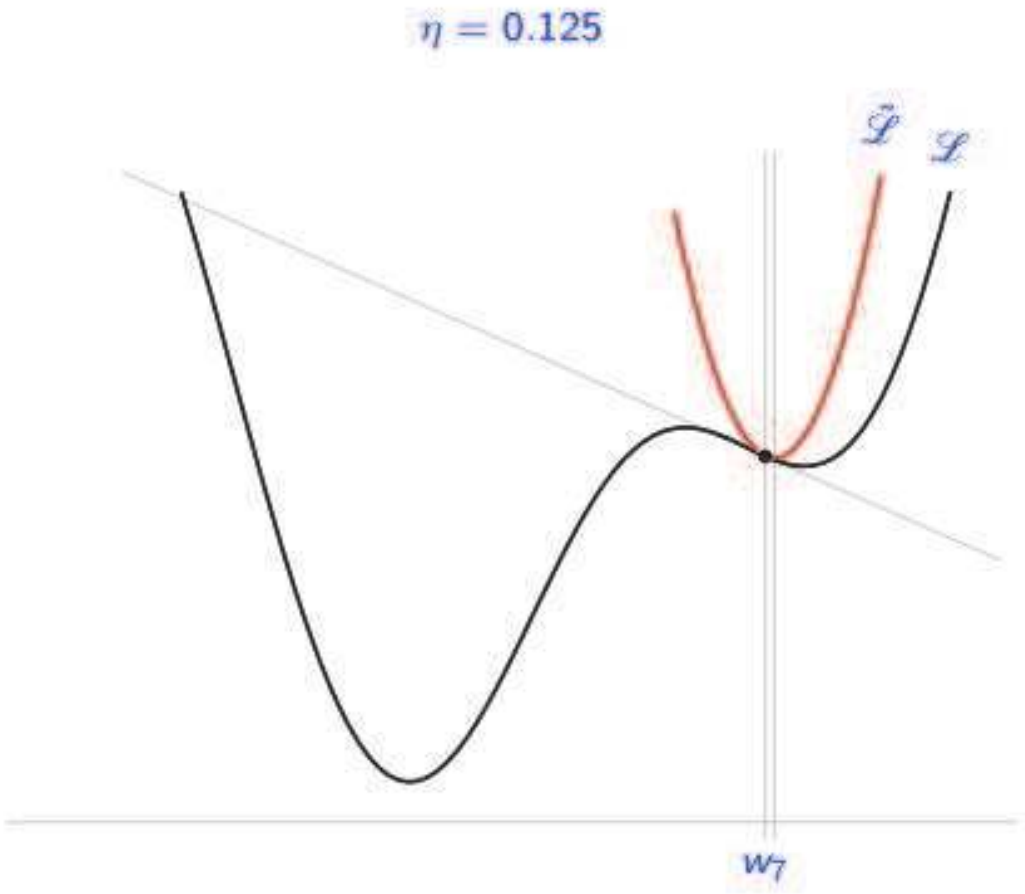
Example 2: Convergence to a local minima

Slide credit: F. Fleuret



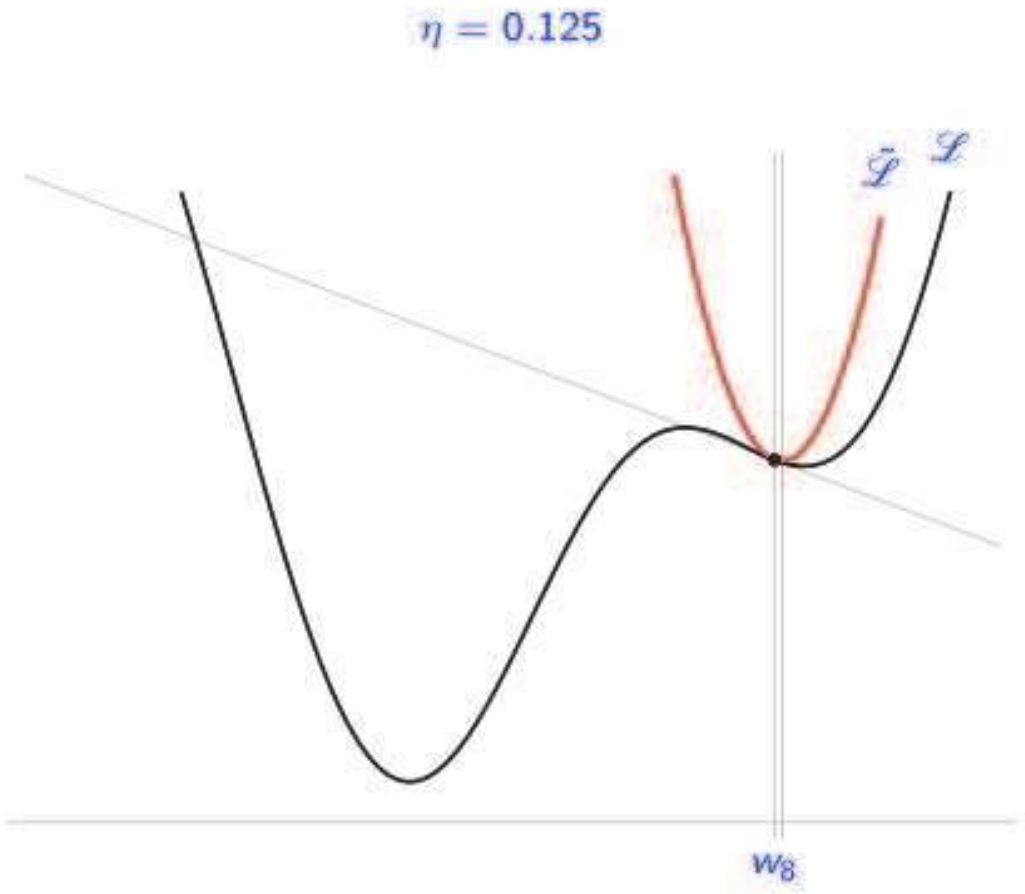
Example 2: Convergence to a local minima

Slide credit: F. Fleuret



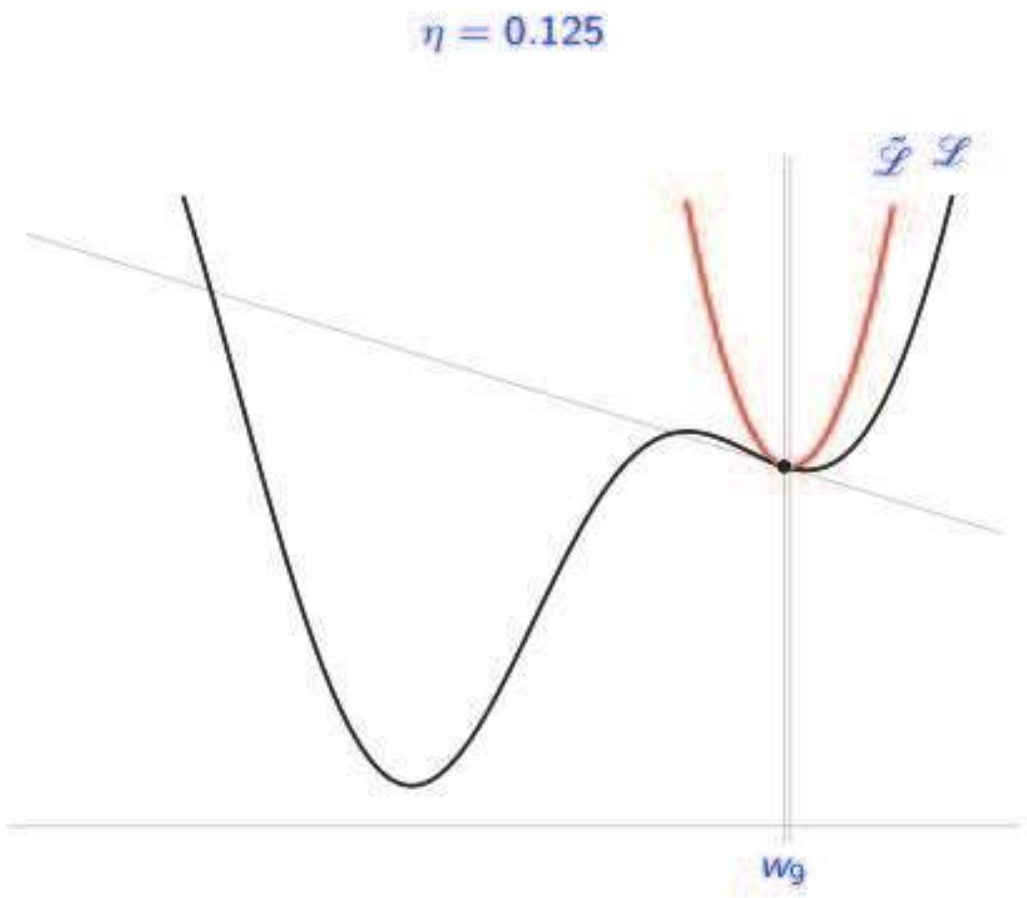
Example 2: Convergence to a local minima

Slide credit: F. Fleuret



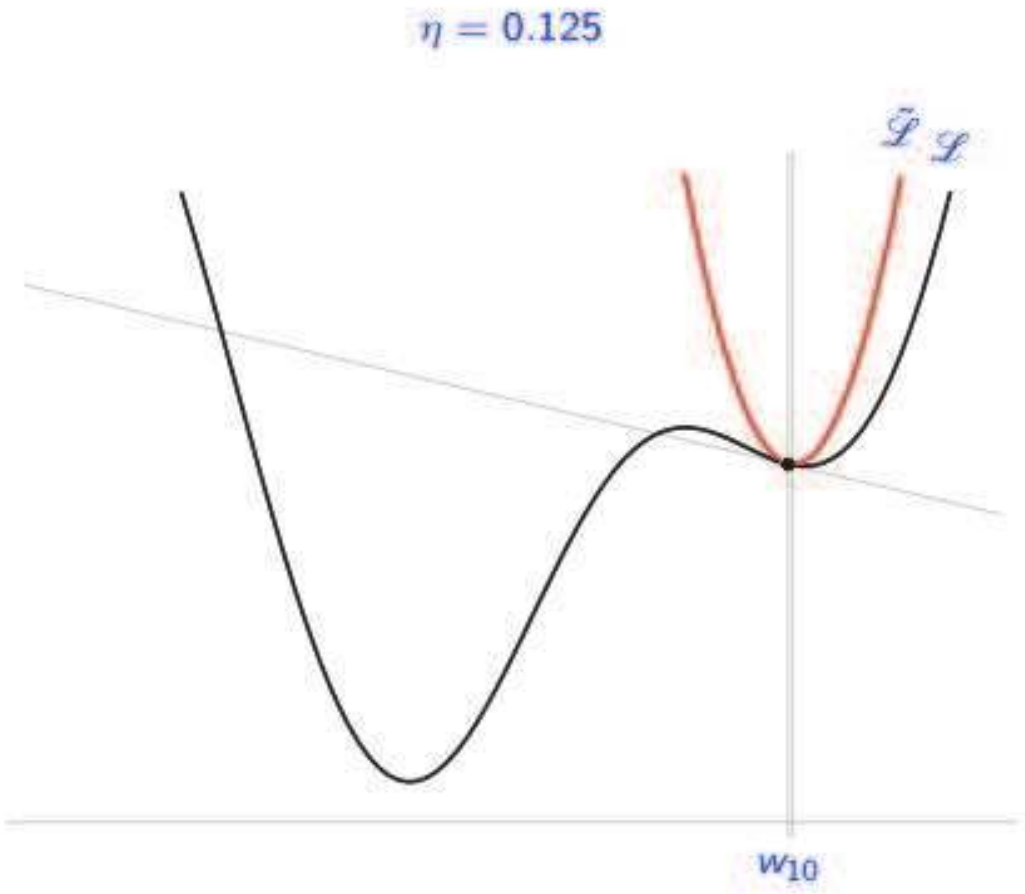
Example 2: Convergence to a local minima

Slide credit: F. Fleuret



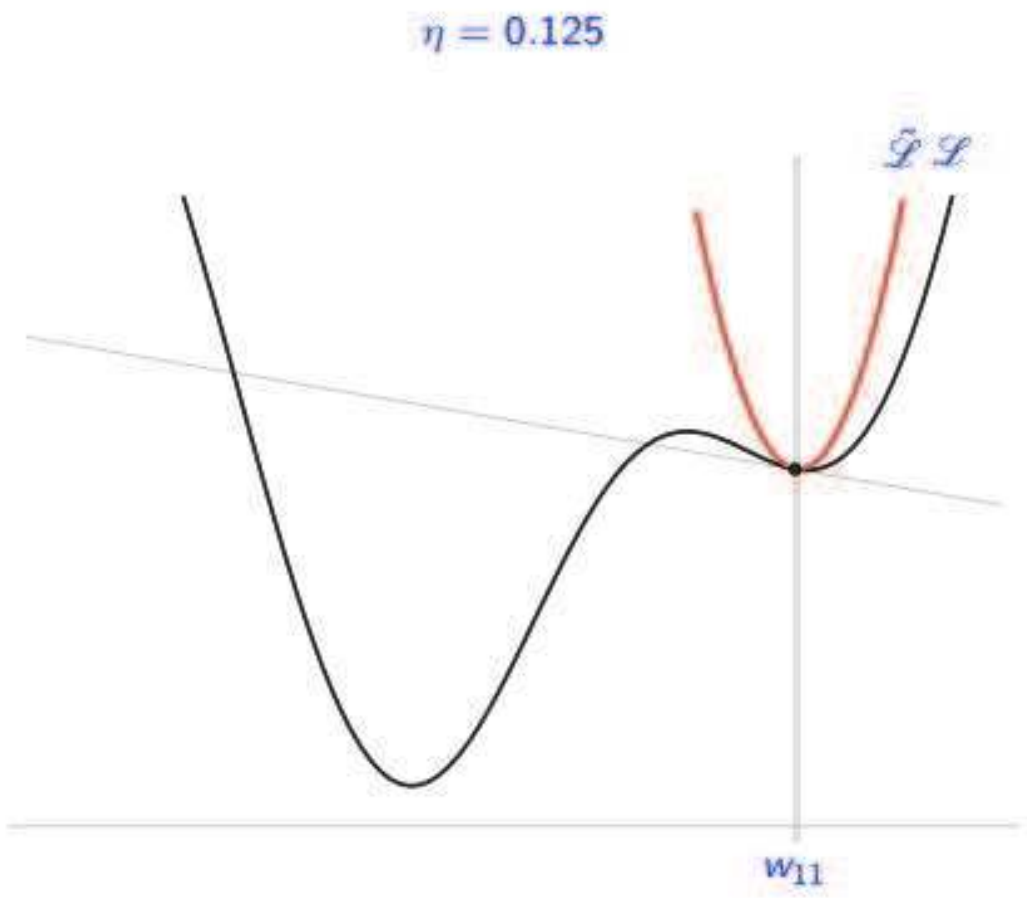
Example 2: Convergence to a local minima

Slide credit: F. Fleuret



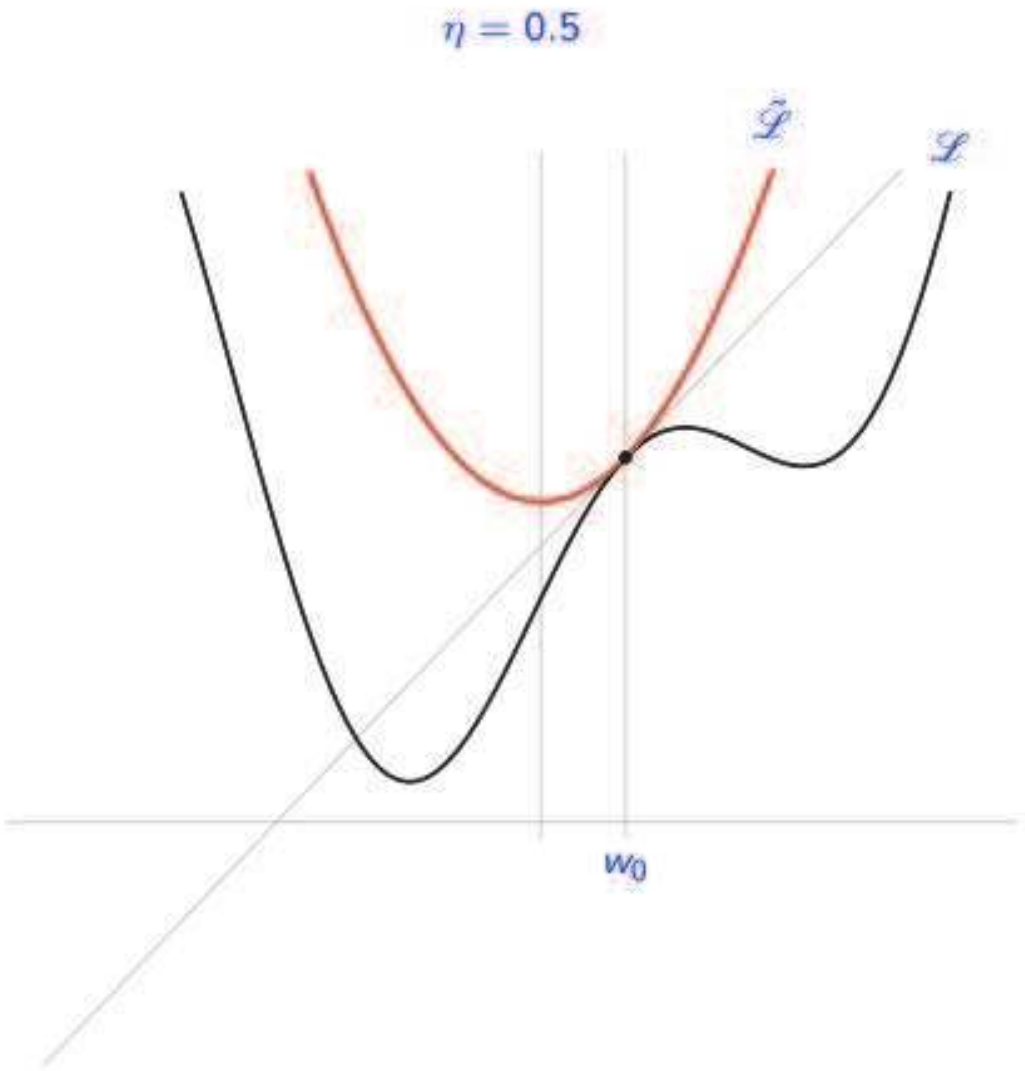
Example 2: Convergence to a local minima

Slide credit: F. Fleuret



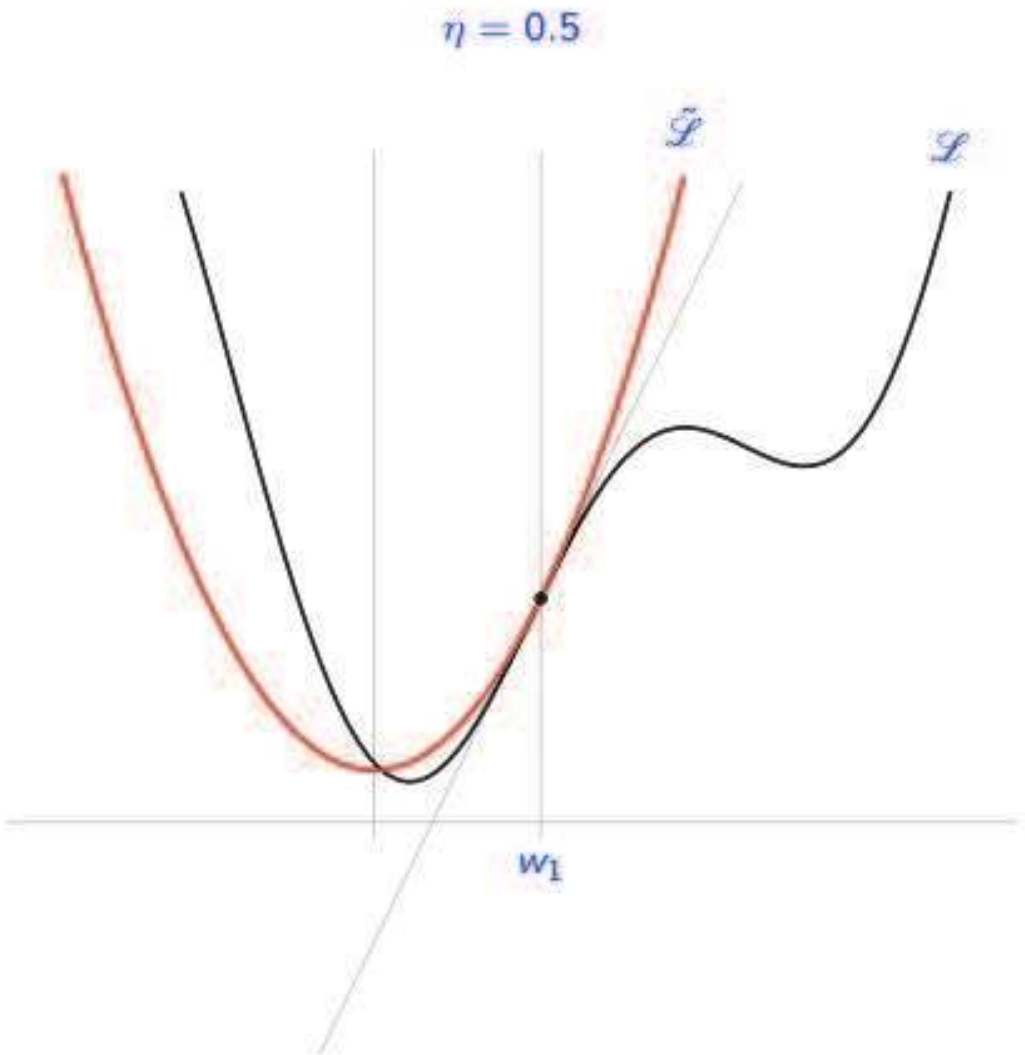
Example 2: Convergence to a local minima

Slide credit: F. Fleuret



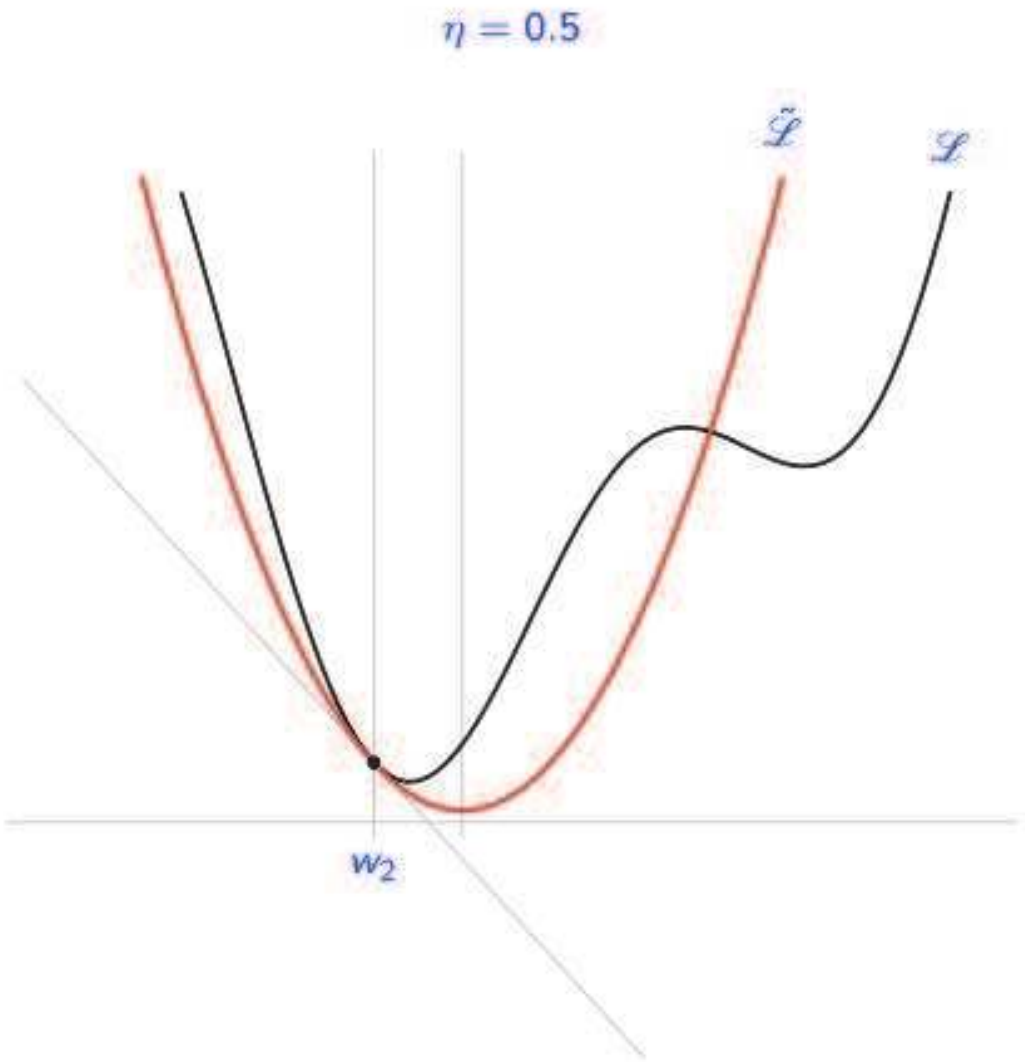
Example 3: Divergence due to a too large learning rate

Slide credit: F. Fleuret



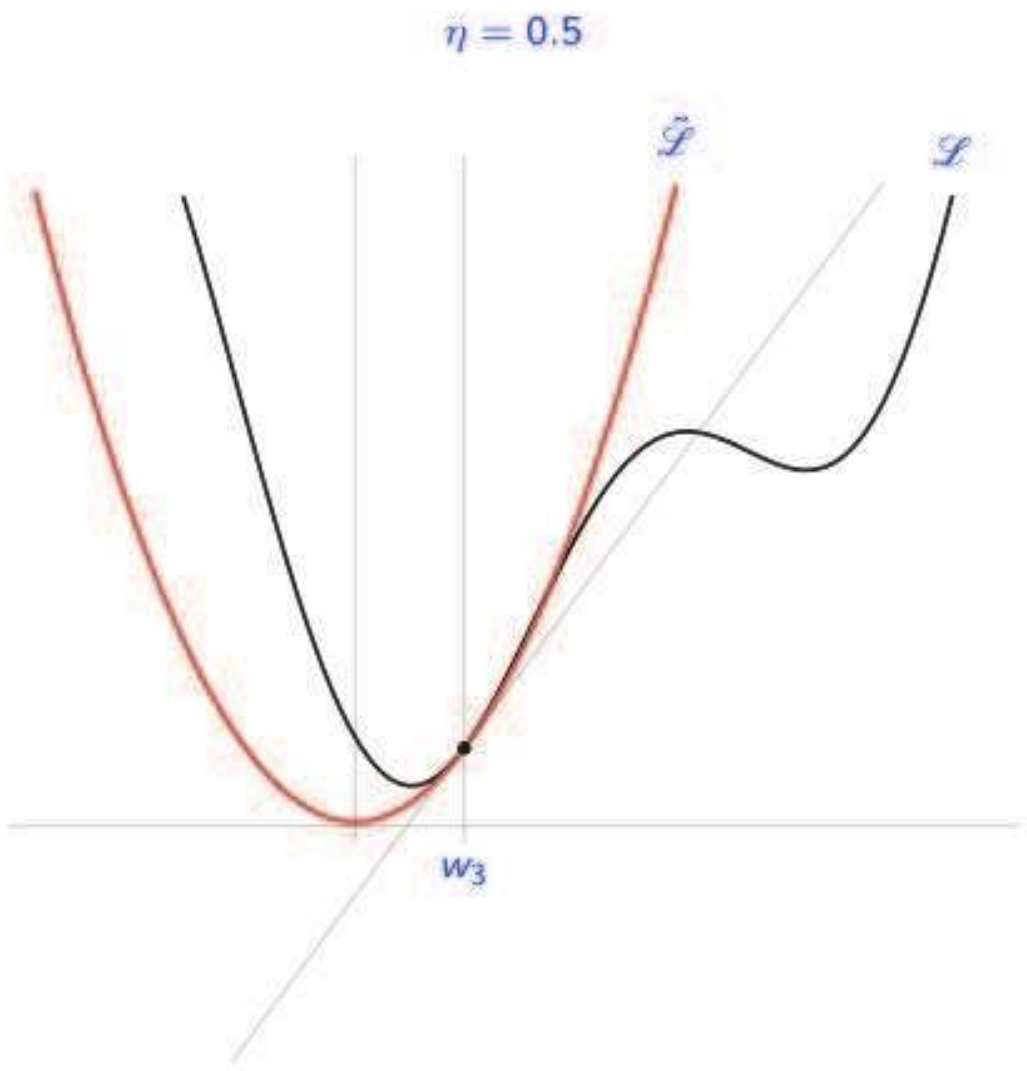
Example 3: Divergence due to a too large learning rate

Slide credit: F. Fleuret



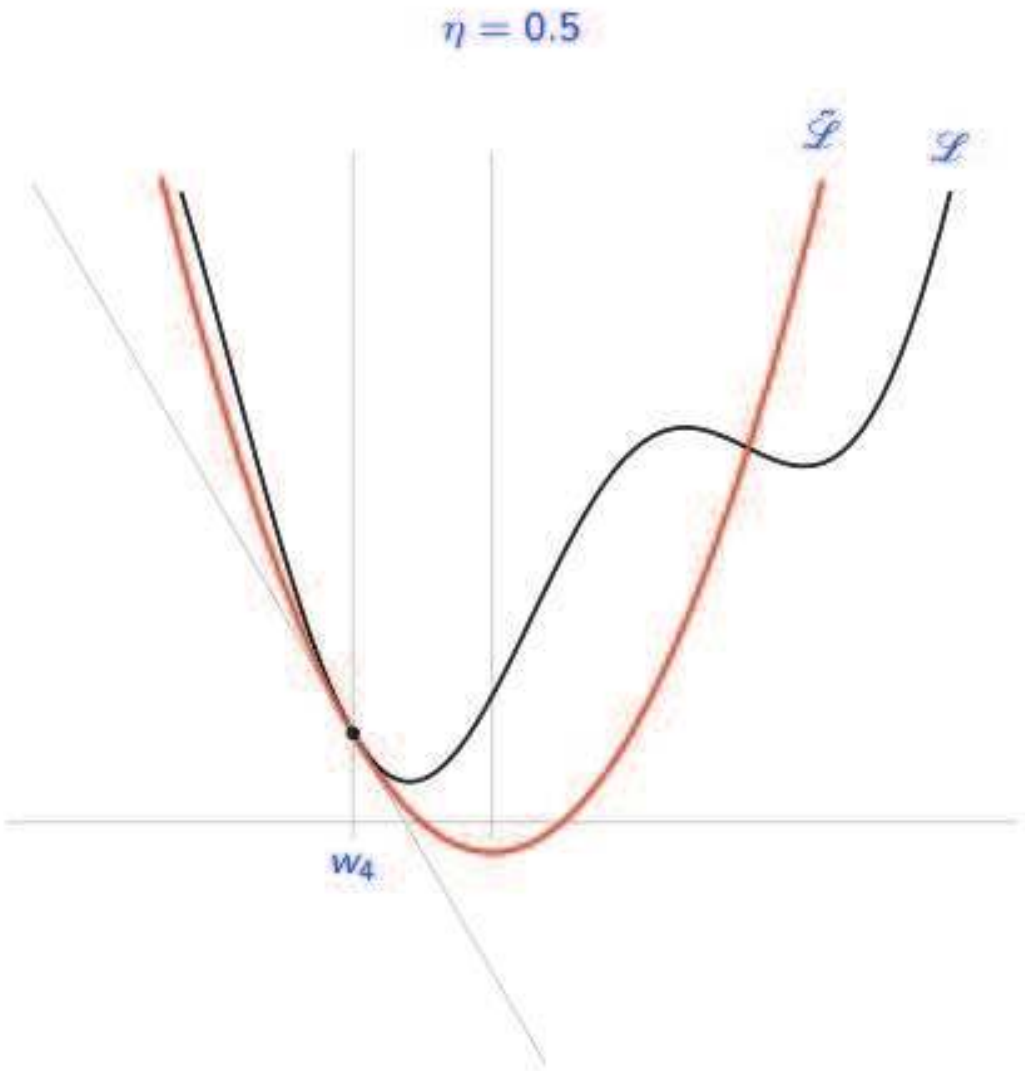
Example 3: Divergence due to a too large learning rate

Slide credit: F. Fleuret



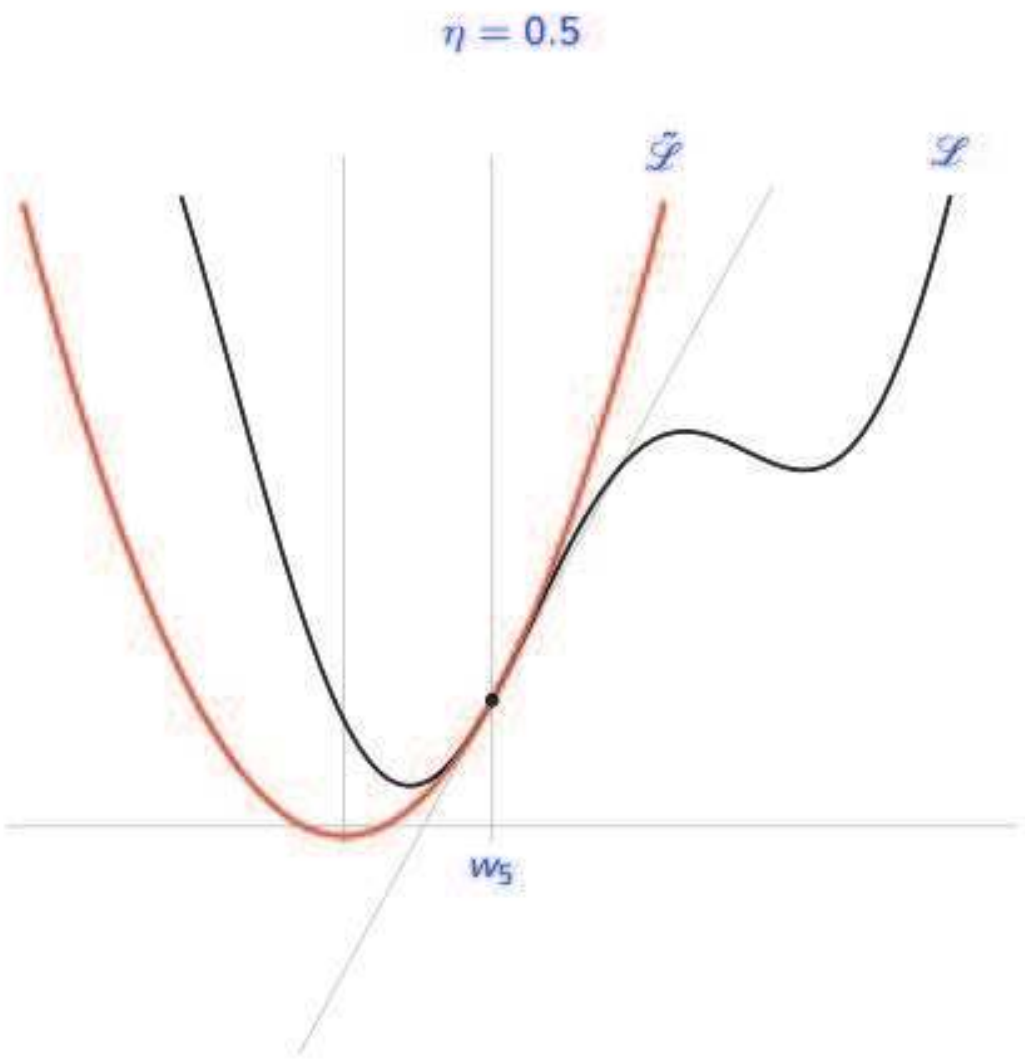
Example 3: Divergence due to a too large learning rate

Slide credit: F. Fleuret



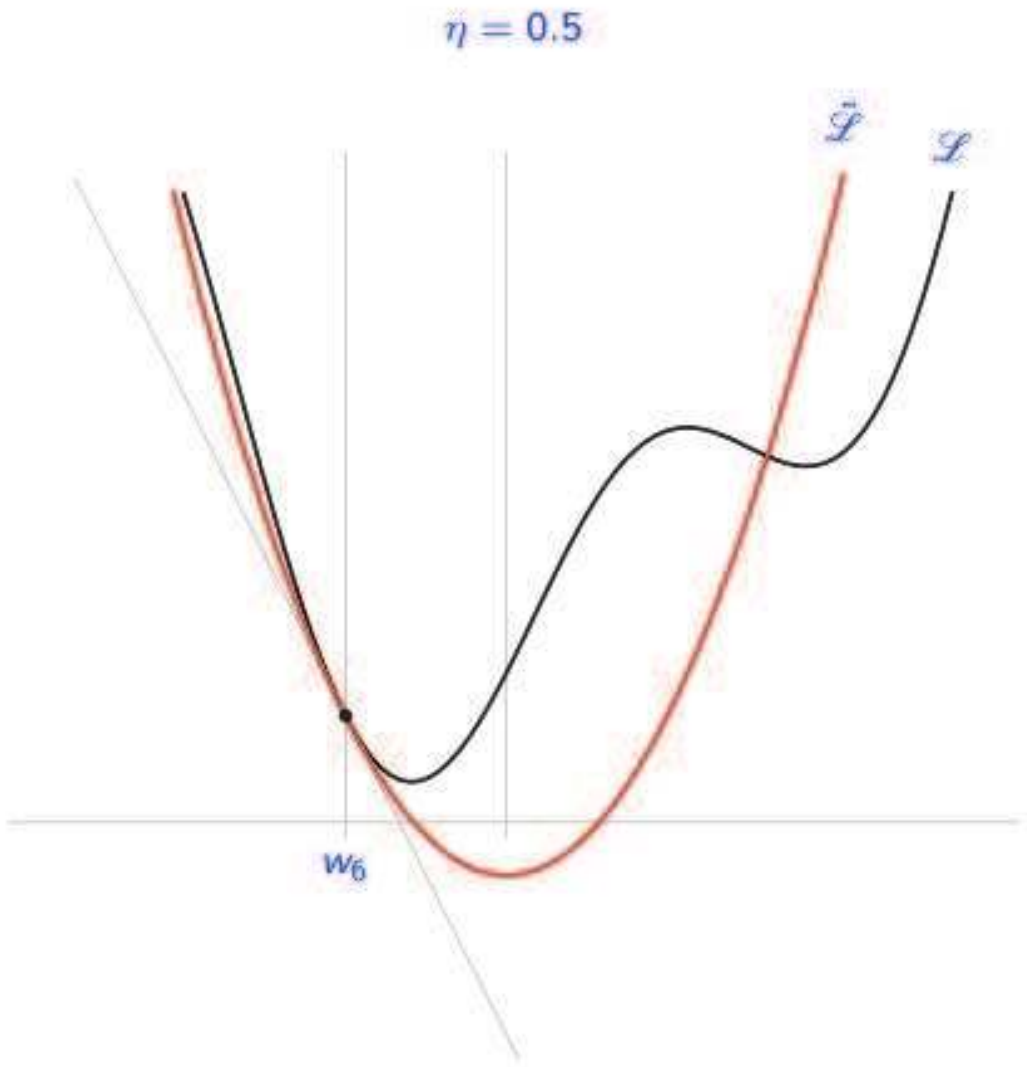
Example 3: Divergence due to a too large learning rate

Slide credit: F. Fleuret



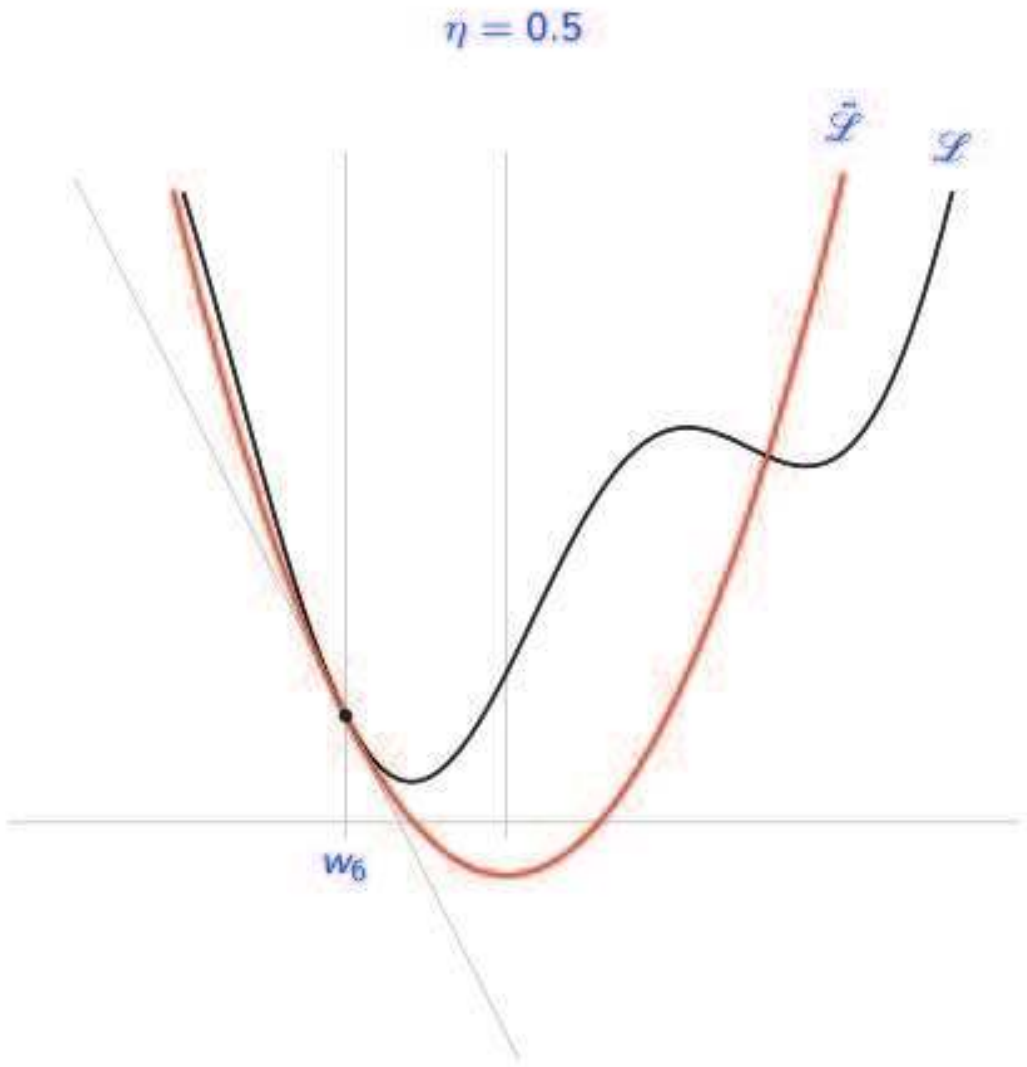
Example 3: Divergence due to a too large learning rate

Slide credit: F. Fleuret



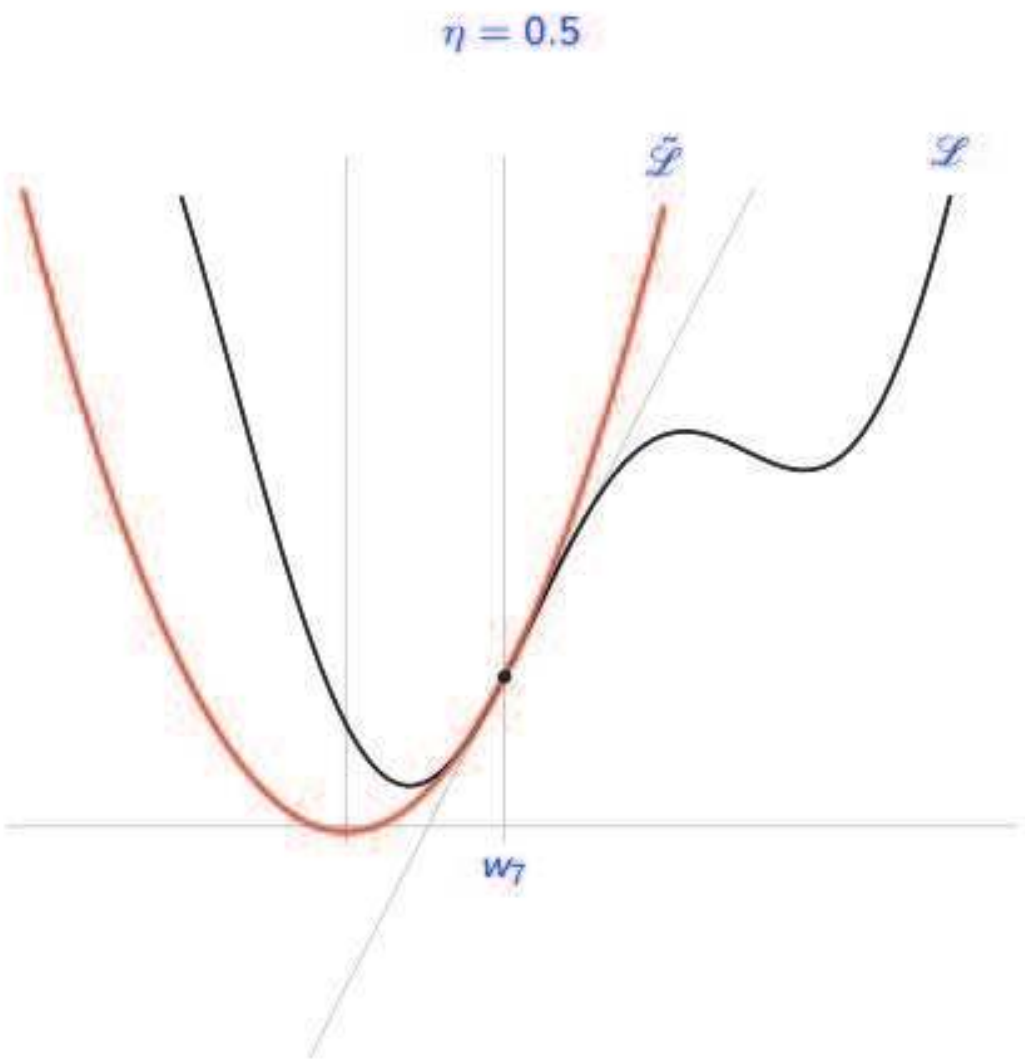
Example 3: Divergence due to a too large learning rate

Slide credit: F. Fleuret



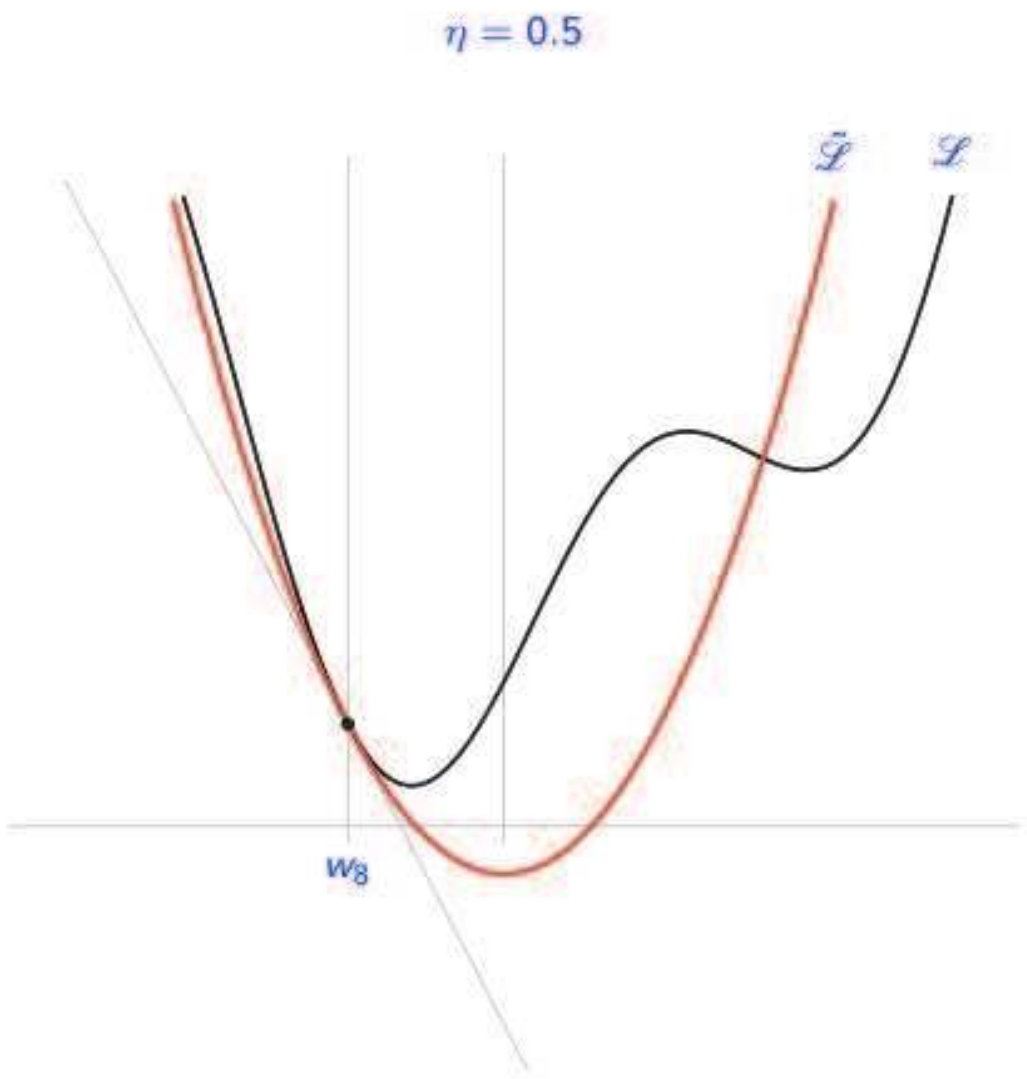
Example 3: Divergence due to a too large learning rate

Slide credit: F. Fleuret



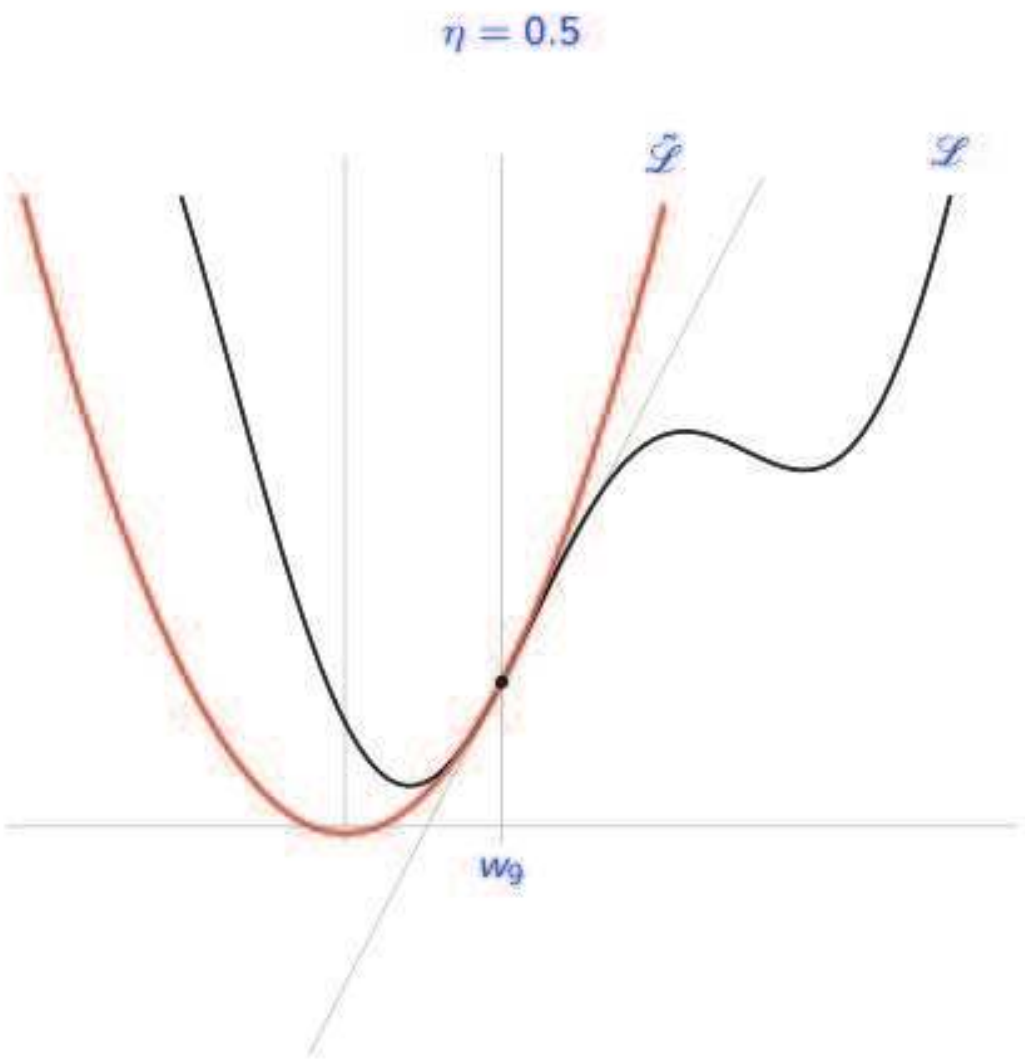
Example 3: Divergence due to a too large learning rate

Slide credit: F. Fleuret



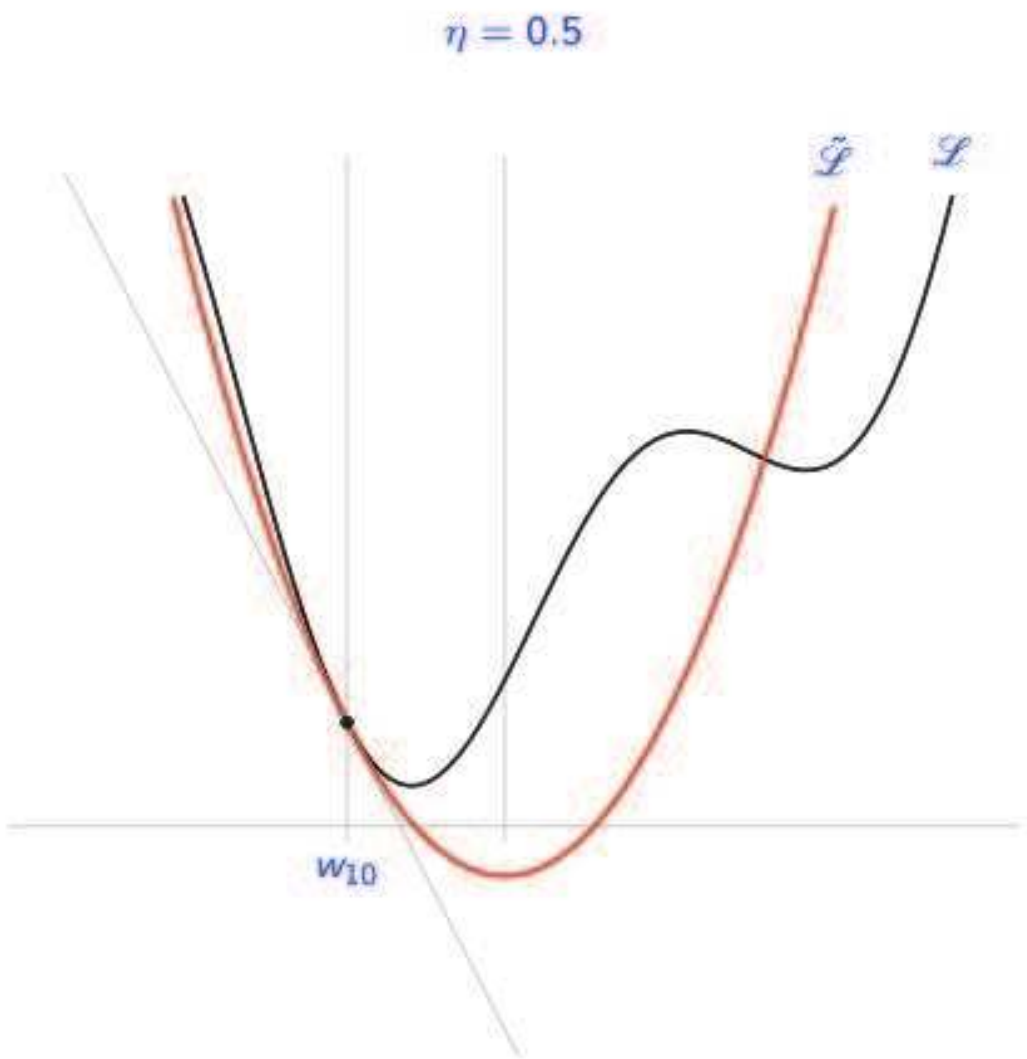
Example 3: Divergence due to a too large learning rate

Slide credit: F. Fleuret



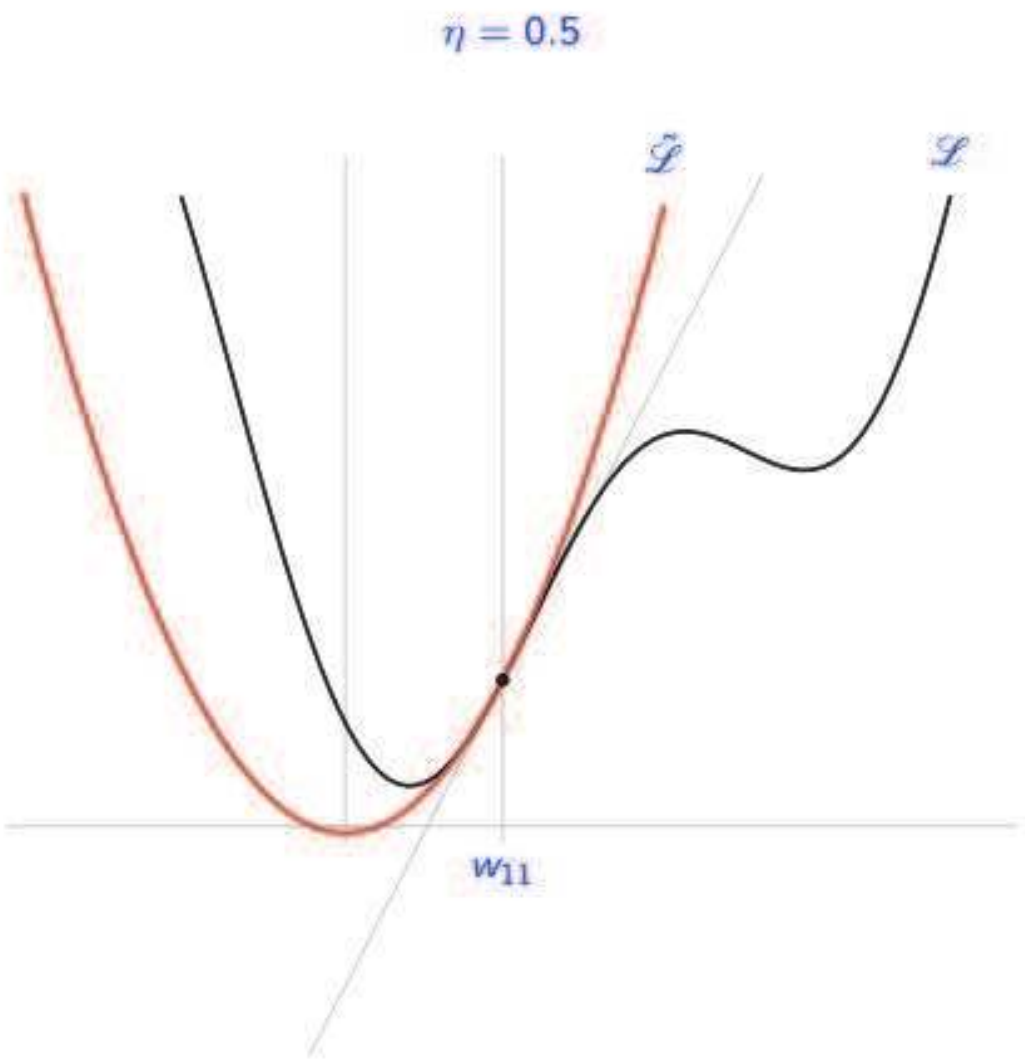
Example 3: Divergence due to a too large learning rate

Slide credit: F. Fleuret



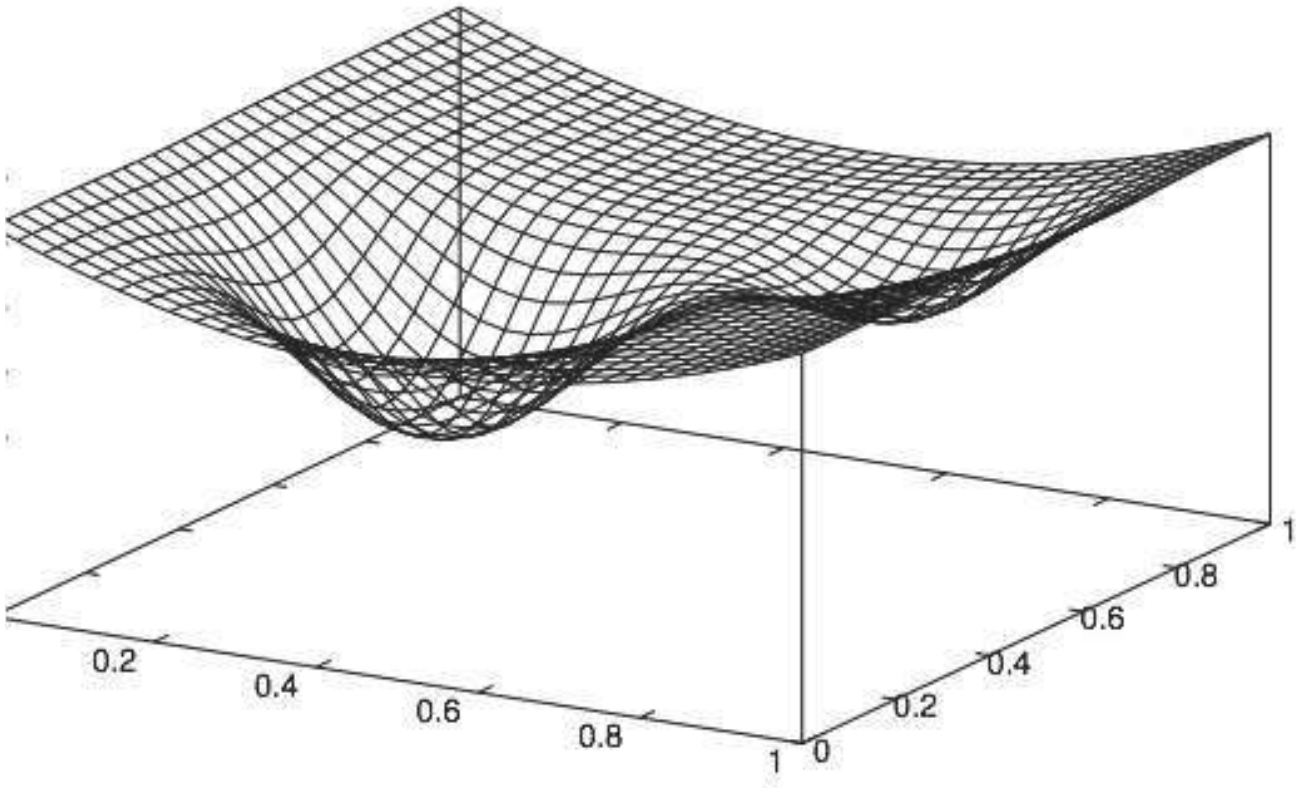
Example 3: Divergence due to a too large learning rate

Slide credit: F. Fleuret

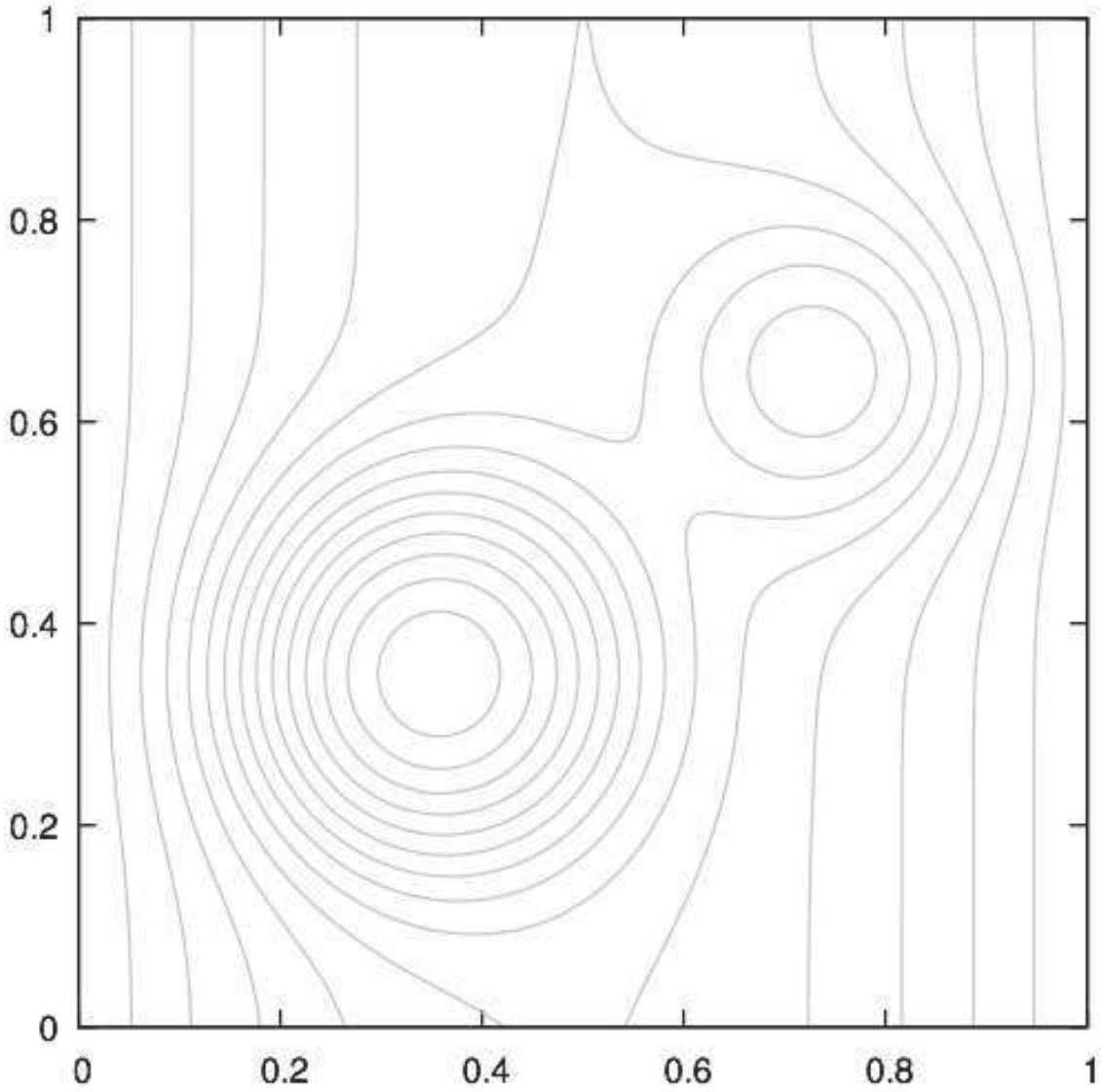


Example 3: Divergence due to a too large learning rate

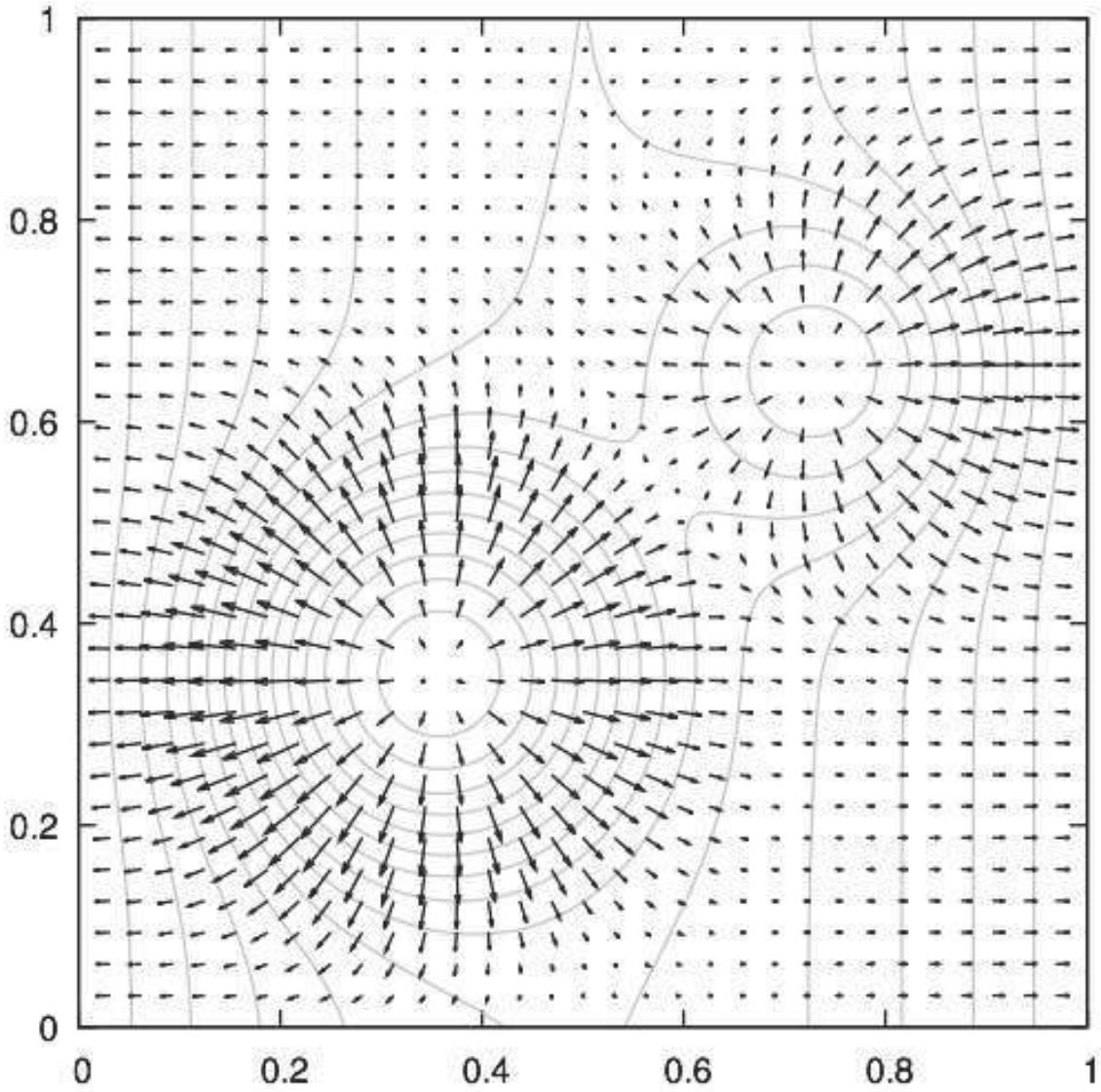
Slide credit: F. Fleuret



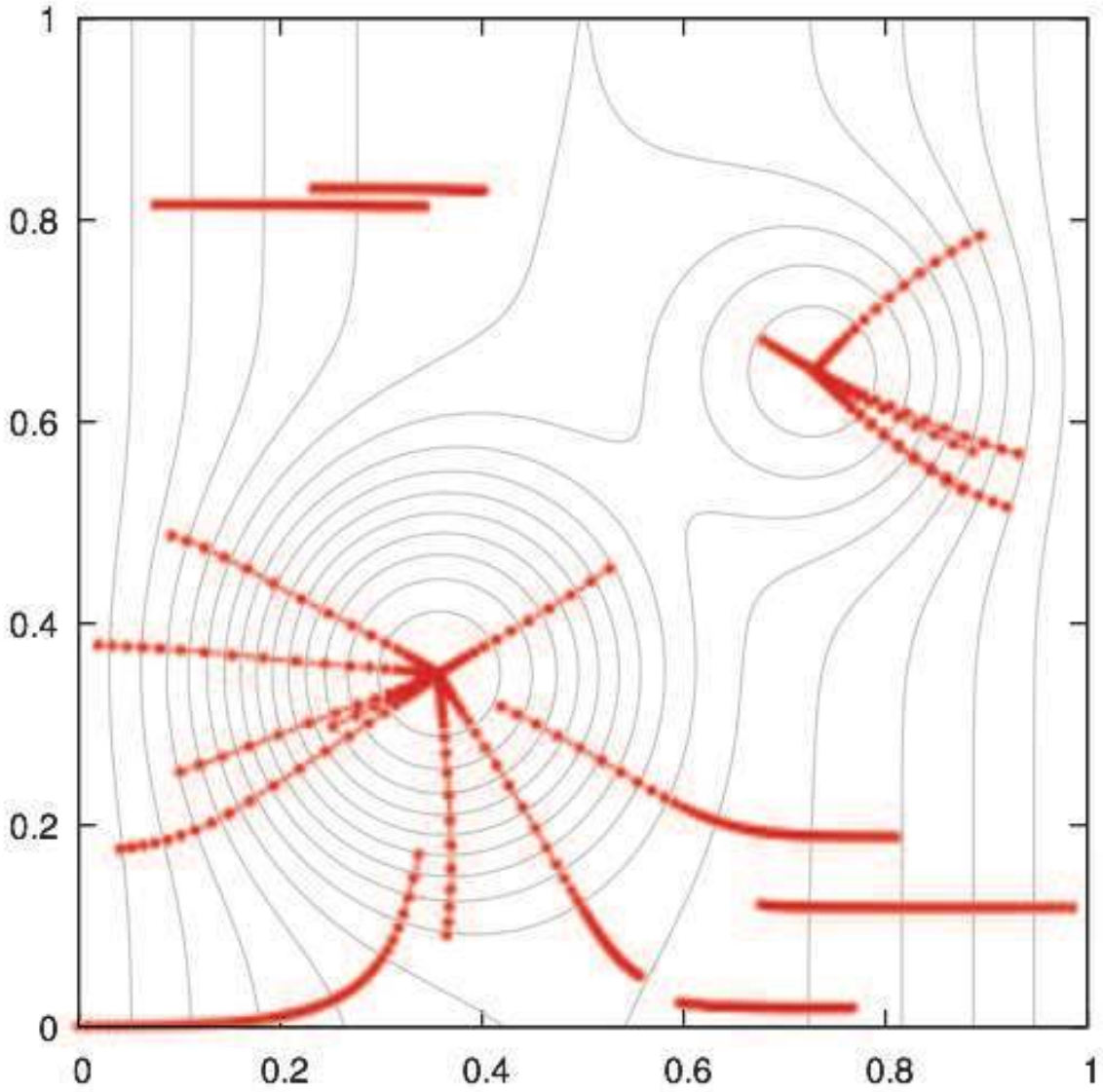
Slide credit: F. Fleuret



Slide credit: F. Fleuret



Slide credit: F. Fleuret



Slide credit: F. Fleuret

Stochastic gradient descent

In the empirical risk minimization setup, $\mathrm{L}(\theta)$ and its gradient decompose as

$$\mathrm{L}(\theta) = \frac{1}{N} \sum_{\mathbf{x}_i, y_i \in \mathbf{d}} \ell(y_i, f(\mathbf{x}_i; \theta))$$

$$\nabla \mathrm{L}(\theta) = \frac{1}{N} \sum_{\mathbf{x}_i, y_i \in \mathbf{d}} \nabla \ell(y_i, f(\mathbf{x}_i; \theta)).$$

Therefore, in batch gradient descent the complexity of an update grows linearly with the size N of the dataset.

More importantly, since the empirical risk is already an approximation of the expected risk, it should not be necessary to carry out the minimization with great accuracy.

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

- It takes a lot of time to compute

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

- It takes a lot of time to compute
- It is an empirical estimation of an hidden quantity, and any partial sum would similarly be an unbiased empirical estimate, although more noisy.

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

- It takes a lot of time to compute
- It is an empirical estimation of an hidden quantity, and any partial sum would similarly be an unbiased empirical estimate, although more noisy.
- It is computed incrementally

$$\nabla \mathrm{L}(\theta) = \frac{1}{N} \sum_{\mathbf{x}_i, y_i \in \mathbf{d}} \nabla \ell(y_i, f(\mathbf{x}_i; \theta)).$$

Instead, stochastic gradient descent uses as update rule:

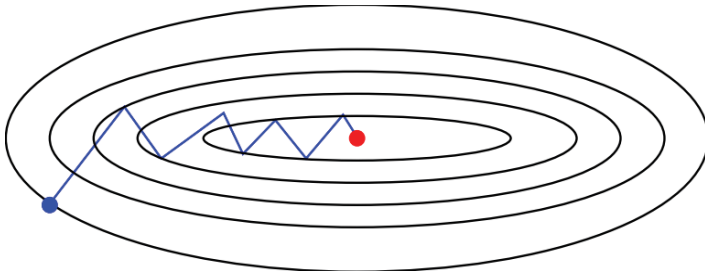
$$\theta_{t+1} = \theta_t - \eta \nabla \ell(y_{i(t+1)}, f(\mathbf{x}_{i(t+1)}; \theta_t))$$

- Iteration complexity is independent of N.
- The stochastic process $\{\theta_t | t = 1, \dots\}$ depends on the examples $i(t)$ picked randomly at each iteration.

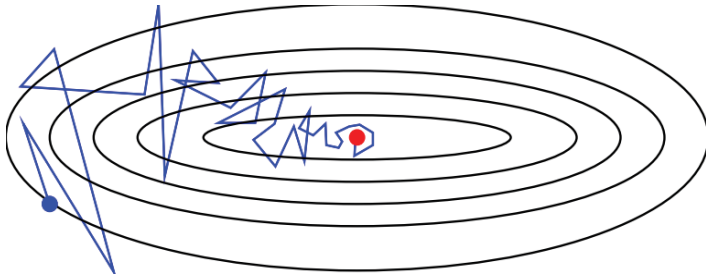
Instead, stochastic gradient descent uses as update rule:

$$\theta_{t+1} = \theta_t - \eta \nabla \ell(y_{i(t+1)}, f(\mathbf{x}_{i(t+1)}; \theta_t))$$

- Iteration complexity is independent of N.
- The stochastic process $\{\theta_t | t = 1, \dots\}$ depends on the examples $i(t)$ picked randomly at each iteration.



Batch gradient descent



Stochastic gradient descent

Why is stochastic gradient descent still a good idea?

- Informally, averaging the update

$$\theta_{t+1} = \theta_t - \eta \nabla \ell(y_{i(t+1)}, f(\mathbf{x}_{i(t+1)}; \theta_t))$$

  over all choices $i(t+1)$ restores batch gradient descent.
- Formally, if the gradient estimate is unbiased, e.g., if

$$\begin{aligned} \mathrm{E}_{i(t+1)}[\nabla \ell(y_{i(t+1)}, f(\mathbf{x}_{i(t+1)}; \theta_t))] &= \frac{1}{N} \sum_{\mathbf{x}_i, y_i \in \mathbf{d}} \nabla \ell(y_i, f(\mathbf{x}_i; \theta_t)) \\ &= \nabla \mathrm{L}(\theta_t) \end{aligned}$$

  then the formal convergence of SGD can be proved, under appropriate assumptions.

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.

$$\theta_{t+1} = \theta_t - \eta \sum_{b=1}^{B} \nabla \ell(y_{i(t+1)}, f(\mathbf{x}_{i(t+1)}; \theta_t))$$

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

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.

$$\theta_{t+1} = \theta_t - \eta \sum_{b=1}^{B} \nabla \ell(y_{i(t+1)}, f(\mathbf{x}_{i(t+1)}; \theta_t))$$

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.

$$\theta_{t+1} = \theta_t - \eta \sum_{b=1}^{B} \nabla \ell(y_{i(t+1)}, f(\mathbf{x}_{i(t+1)}; \theta_t))$$

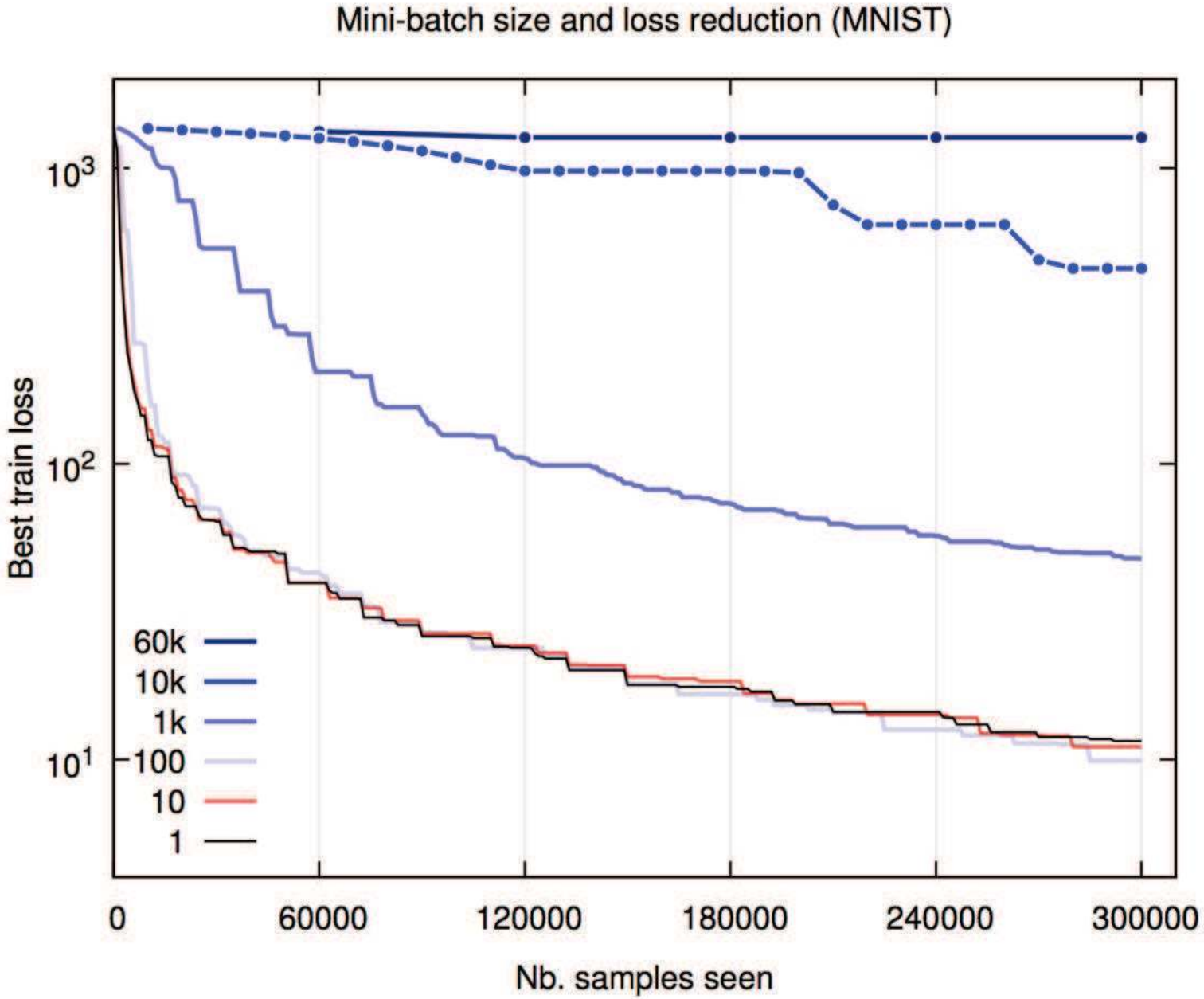
The order $n(t, b)$ to visit the samples can either be sequential, or uniform sampling, usually without replacement.

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.

$$\theta_{t+1} = \theta_t - \eta \sum_{b=1}^{B} \nabla \ell(y_{i(t+1)}, f(\mathbf{x}_{i(t+1)}; \theta_t))$$

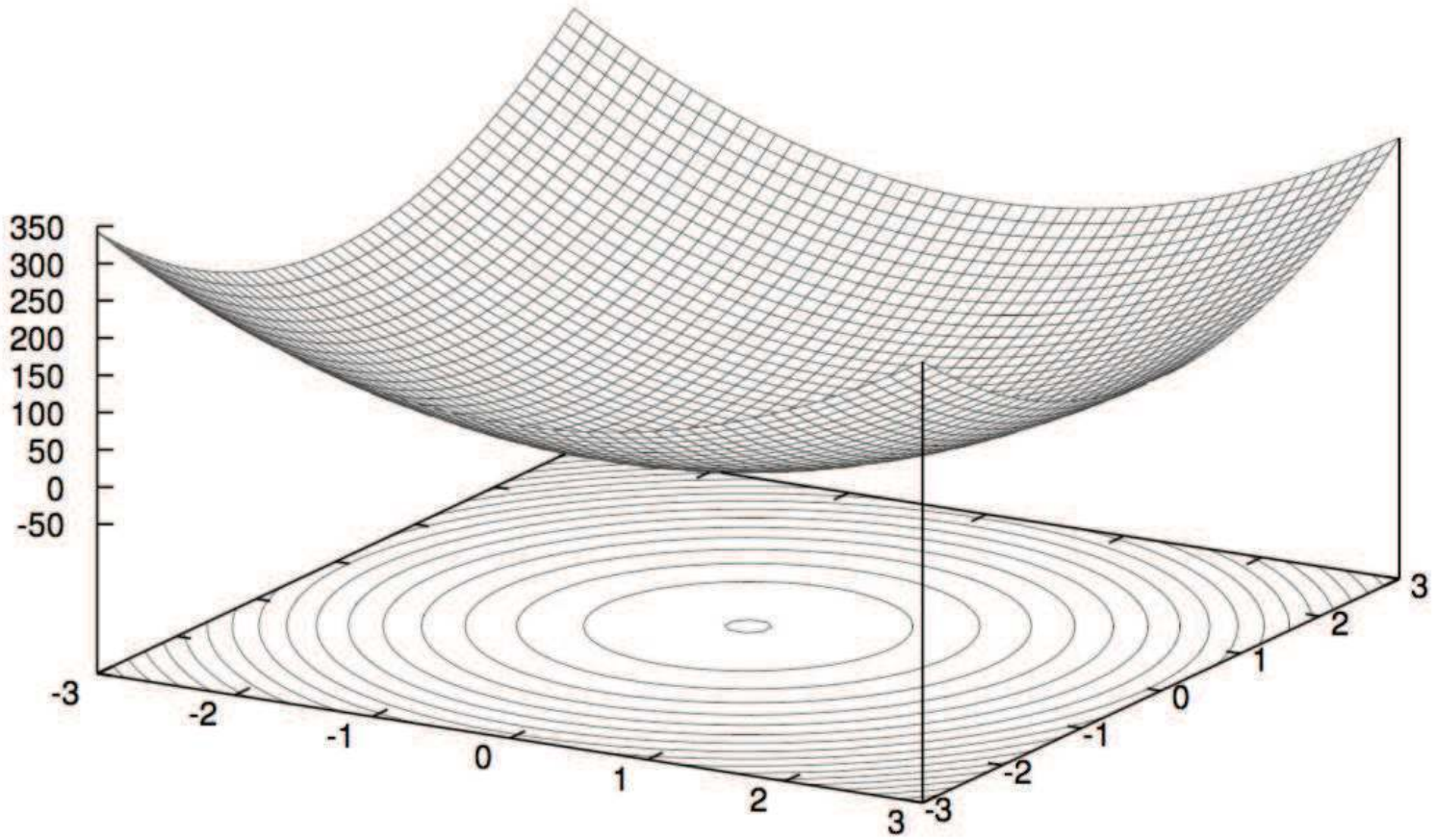
The order $n(t, b)$ to visit the samples can either be sequential, or uniform sampling, usually without replacement.

The stochastic behavior of this procedure helps evade local minima.


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

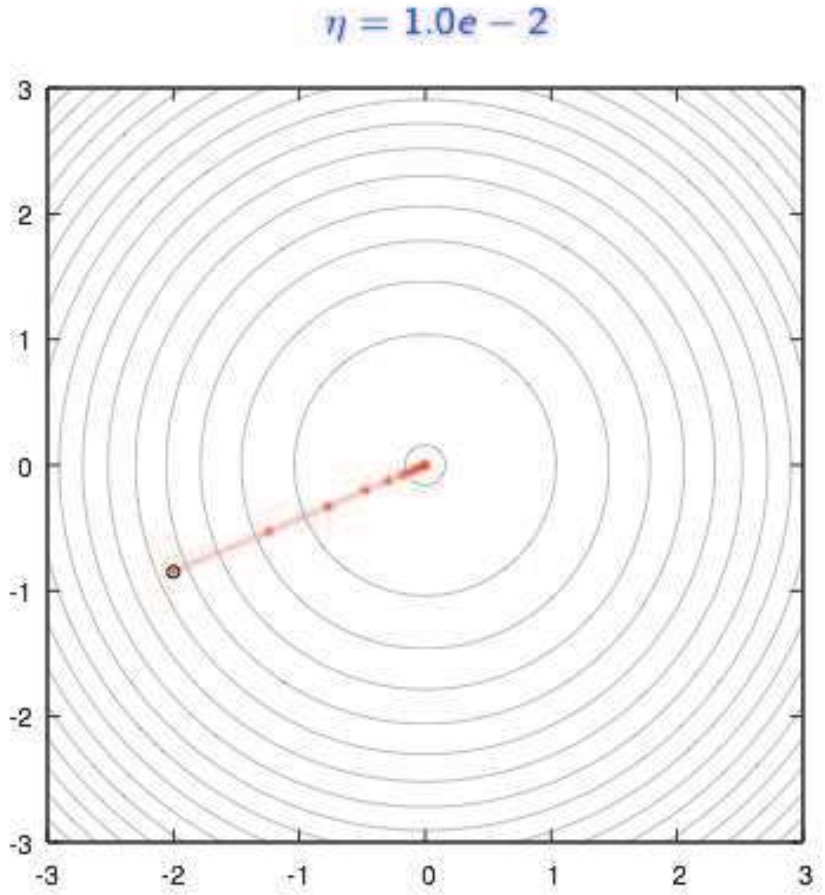
Limitations of gradient descent

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.



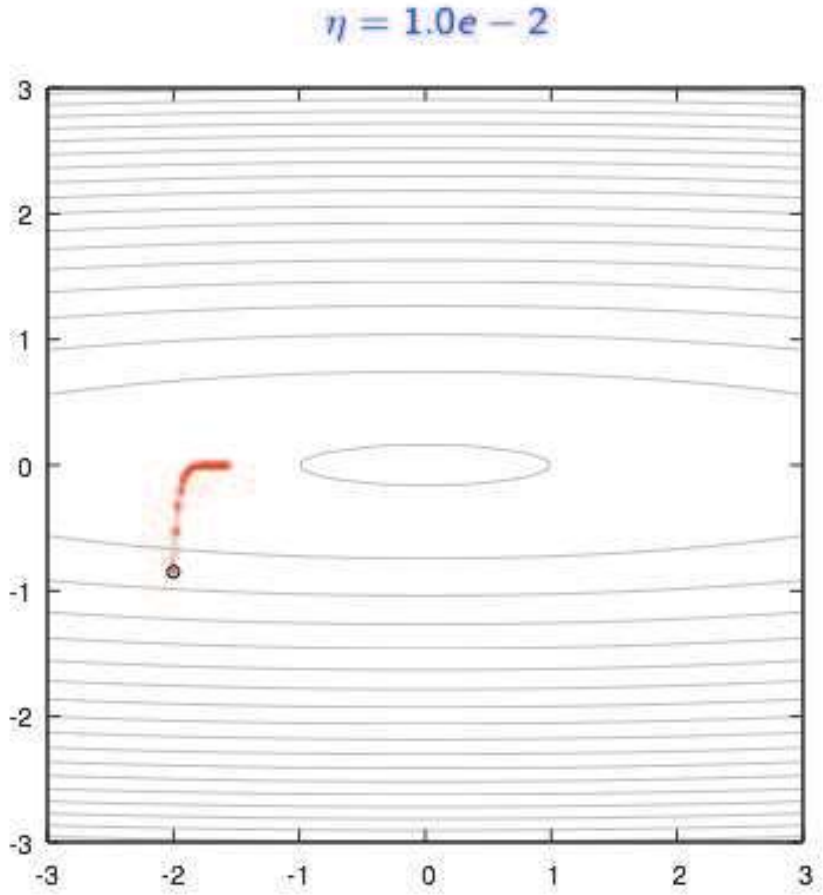
Limitations of gradient descent

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.



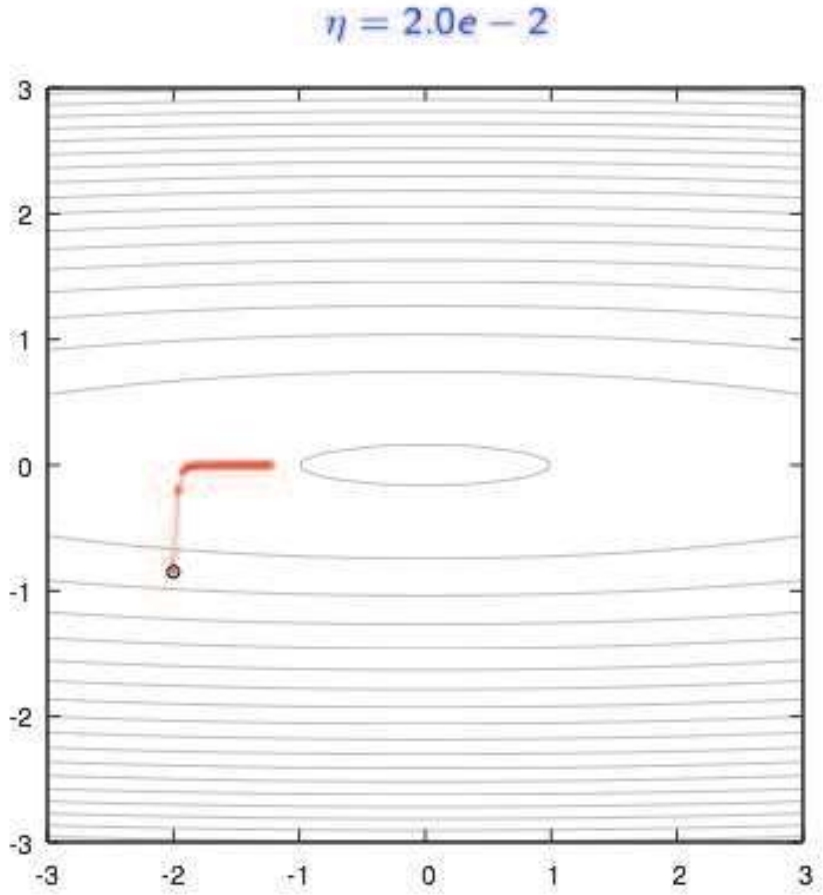
Limitations of gradient descent

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.



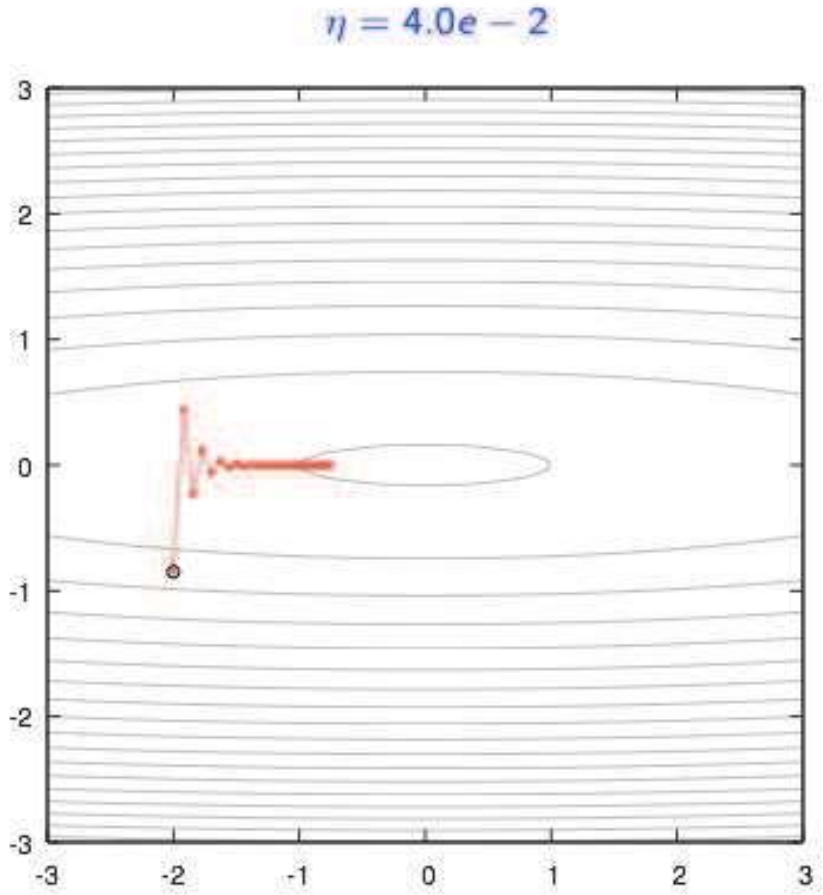
Limitations of gradient descent

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.



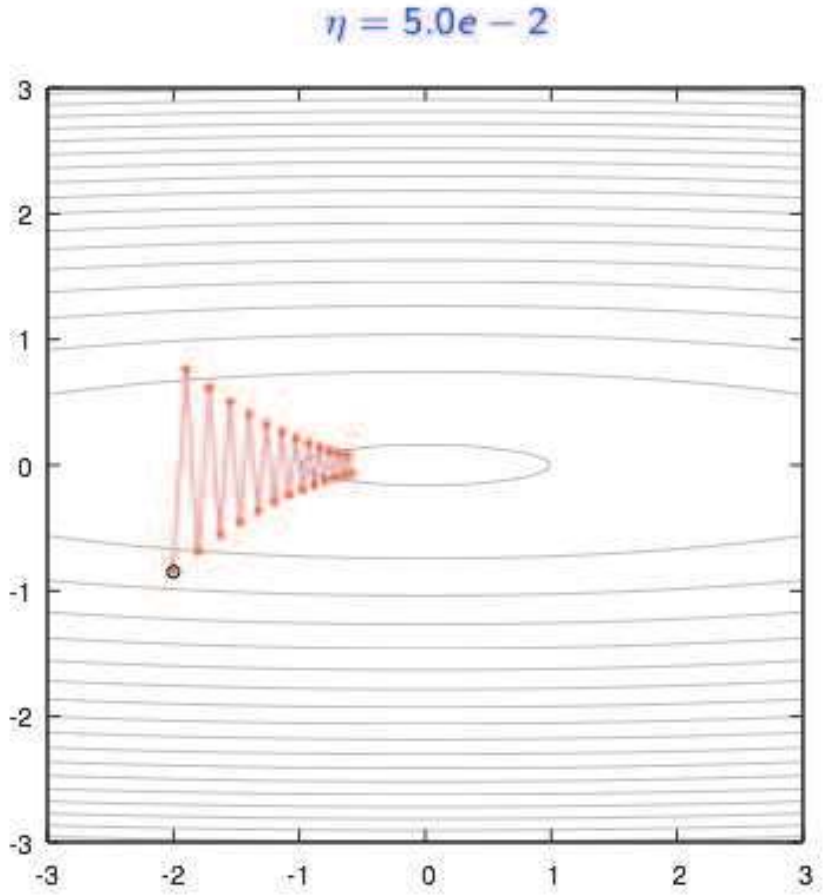
Limitations of gradient descent

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.



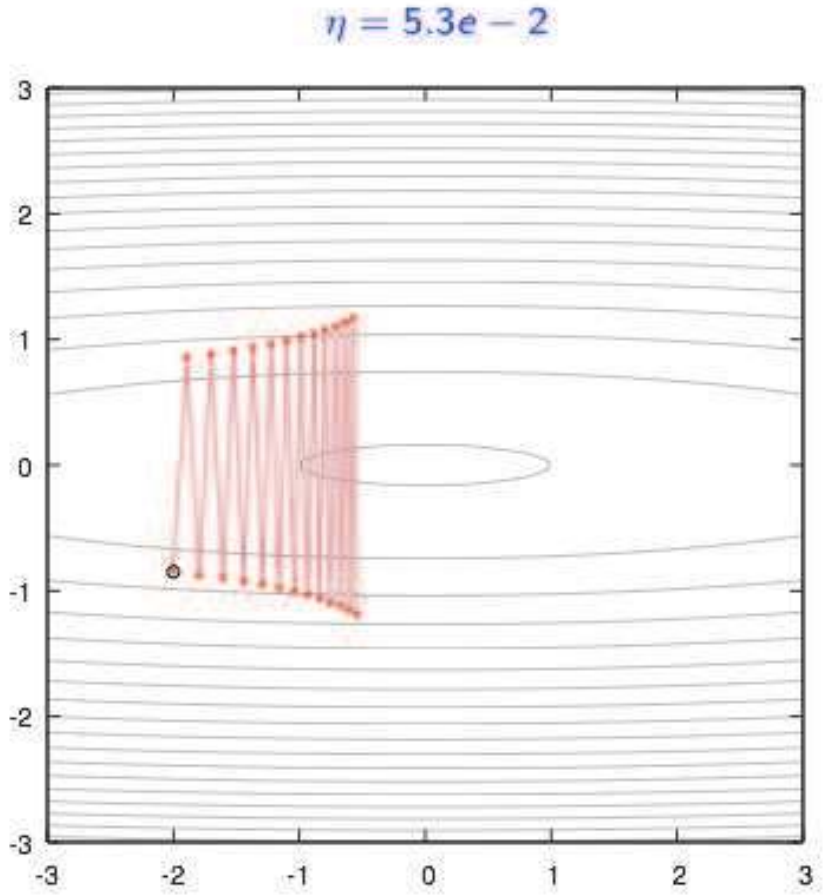
Limitations of gradient descent

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.



Limitations of gradient descent

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.



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

However the resulting computational overhead reduces the number of iterations for a fixed budget, and it is rarely at the advantage of these methods.

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.

Momentum and moment estimation

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

$$\theta_{t+1} = \theta_t - \eta g_t$$

where

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

is the gradient summed over a mini-batch

Momentum and moment estimation

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

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

$$\theta_{t+1} = \theta_t - u_t$$

Momentum and moment estimation

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

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

$$\theta_{t+1} = \theta_t - u_t$$

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

Momentum and moment estimation

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

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

$$\theta_{t+1} = \theta_t - u_t$$

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

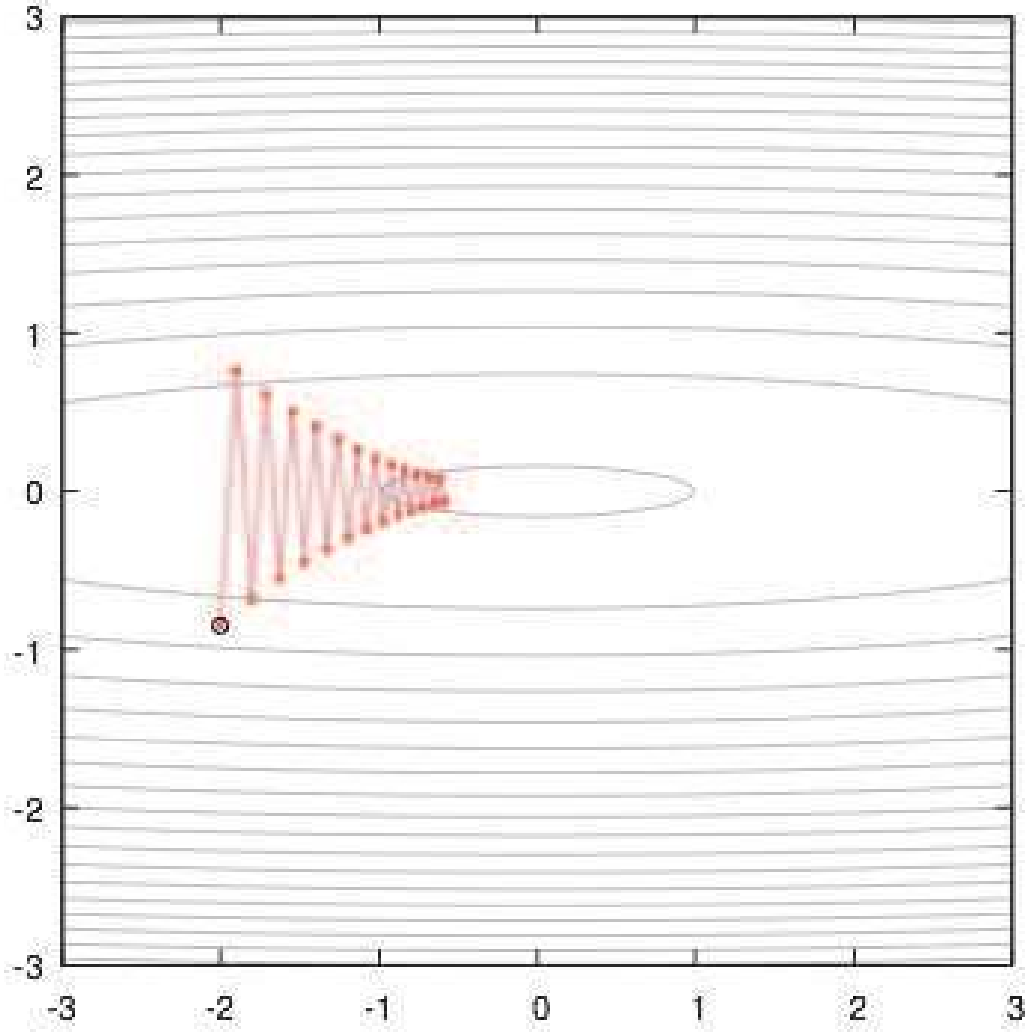
With $\gamma > 0$, this update:

- can go through local barriers
- accelerate if the gradient does not change much
- dampnes oscillations in narrow valleys

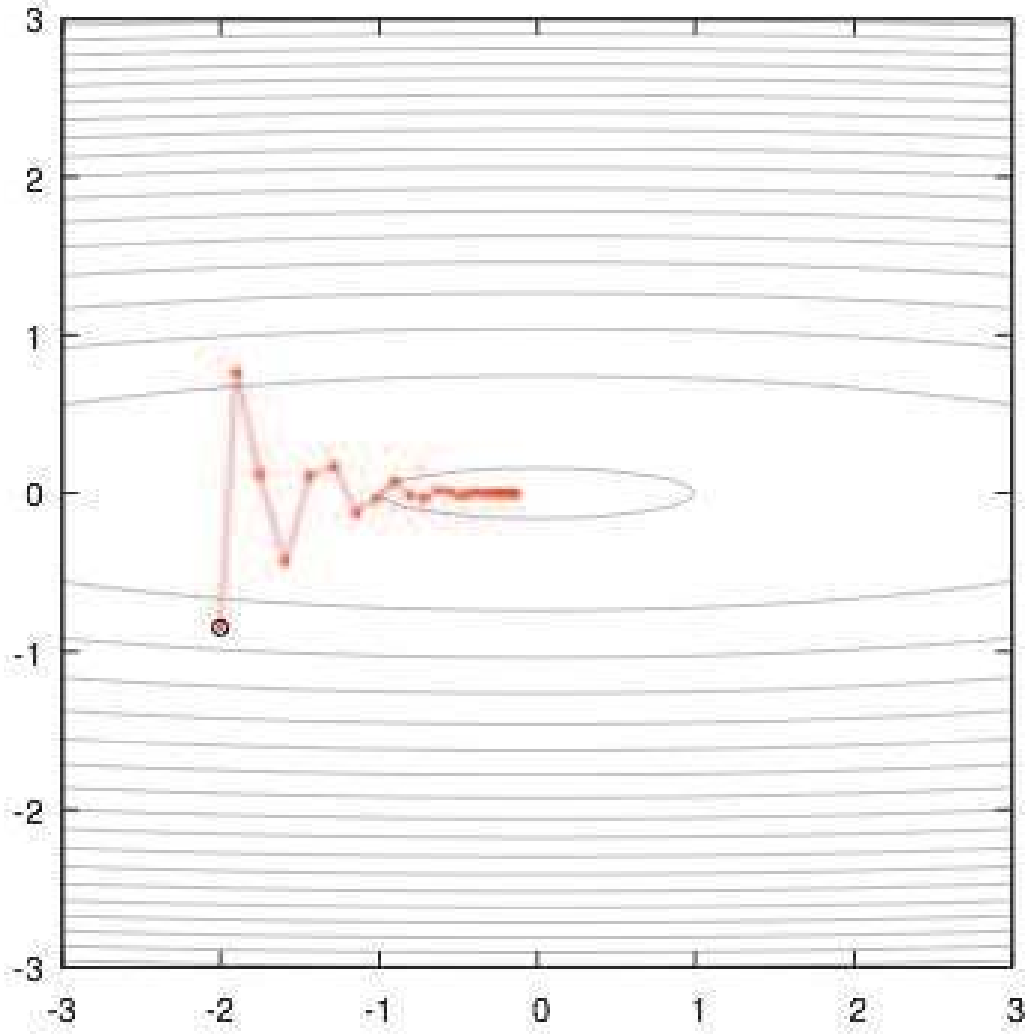
γ is typically set to 0.9

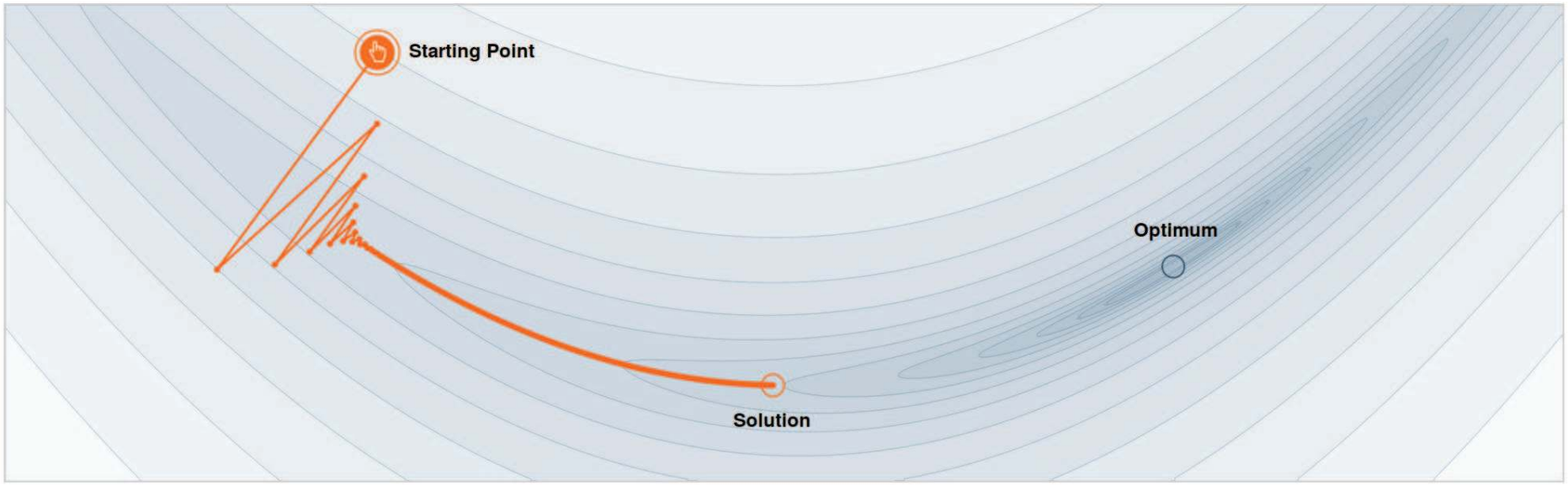
Rumelhart et al., 1986

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



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





Step-size α = 0.0030

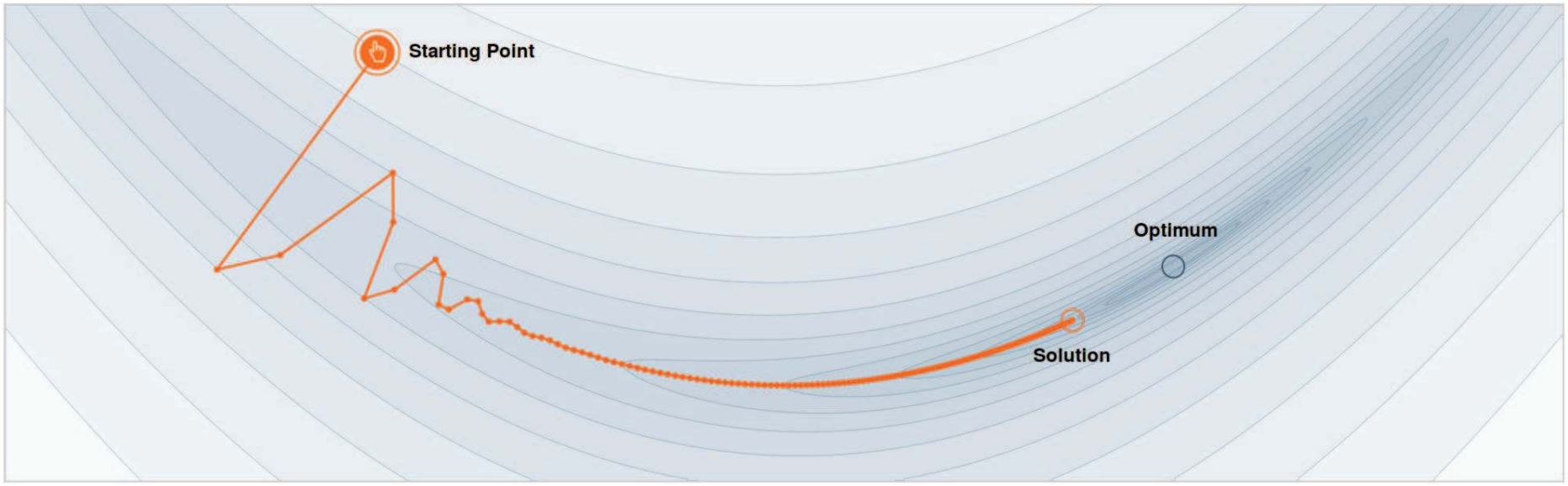
0 0.003 0.006

Momentum β = 0.0

0.00 0.500 0.990

We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

Why Momentum Really Works



Step-size α = 0.0030

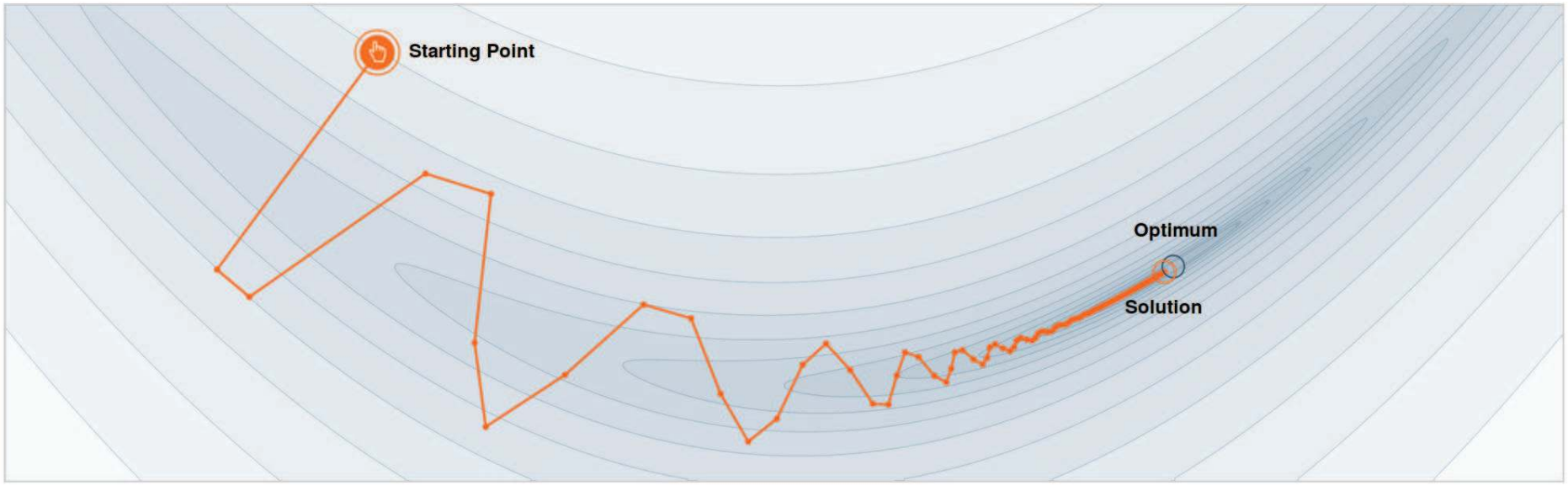
0 0.003 0.006

Momentum β = 0.60

0.00 0.500 0.990

We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

Why Momentum Really Works



Step-size α = 0.0030

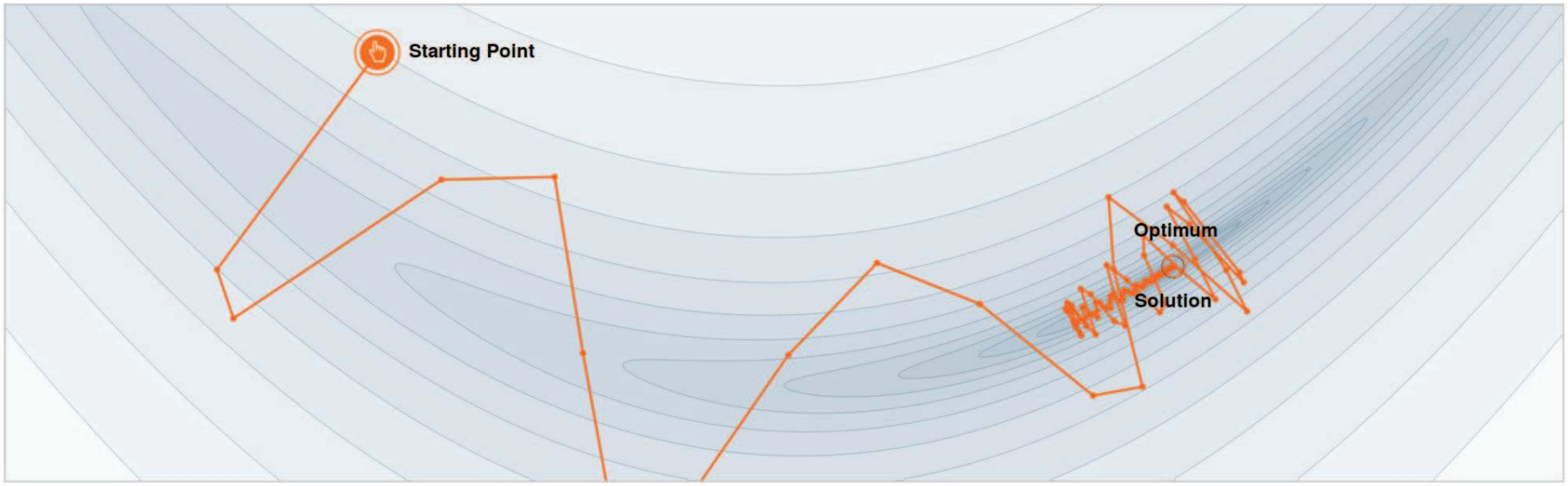
0 0.003 0.006

Momentum β = 0.80

0.00 0.500 0.990

We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

Why Momentum Really Works



Step-size α = 0.0030

0 0.003 0.006

Momentum β = 0.90

0.00 0.500 0.990

We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

Why Momentum Really Works

Momentum and moment estimation

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.

Momentum and moment estimation

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

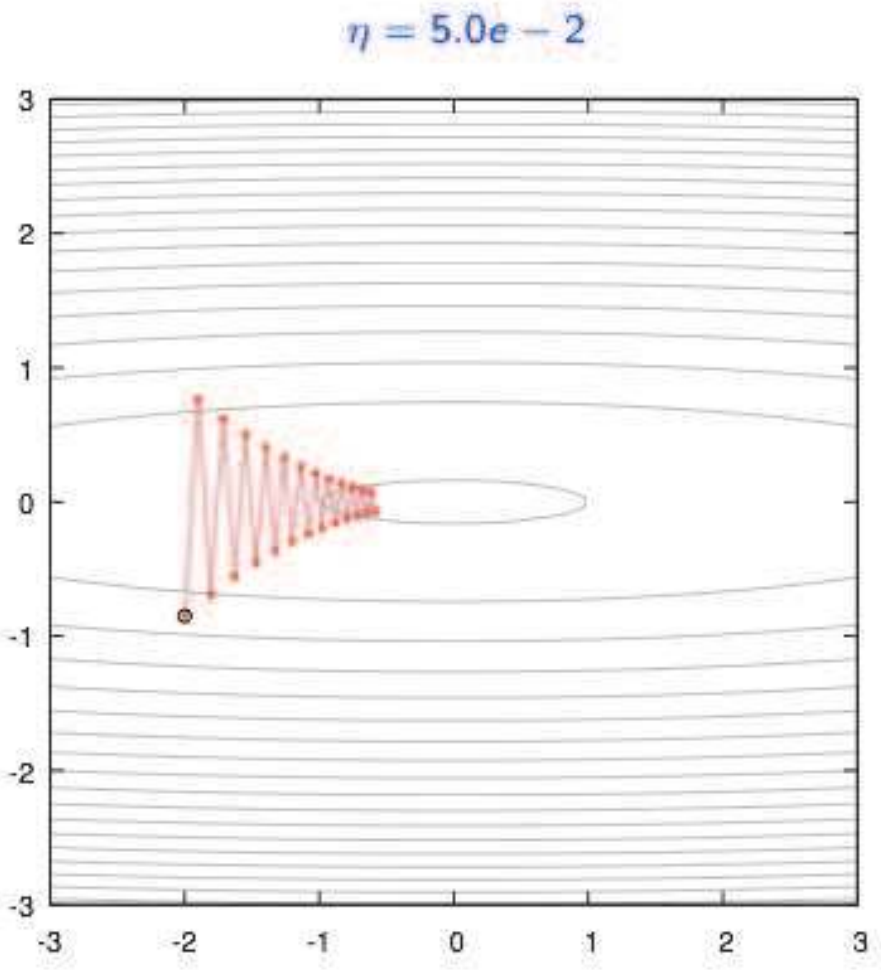
Kingma and Ba, 2014

Adam

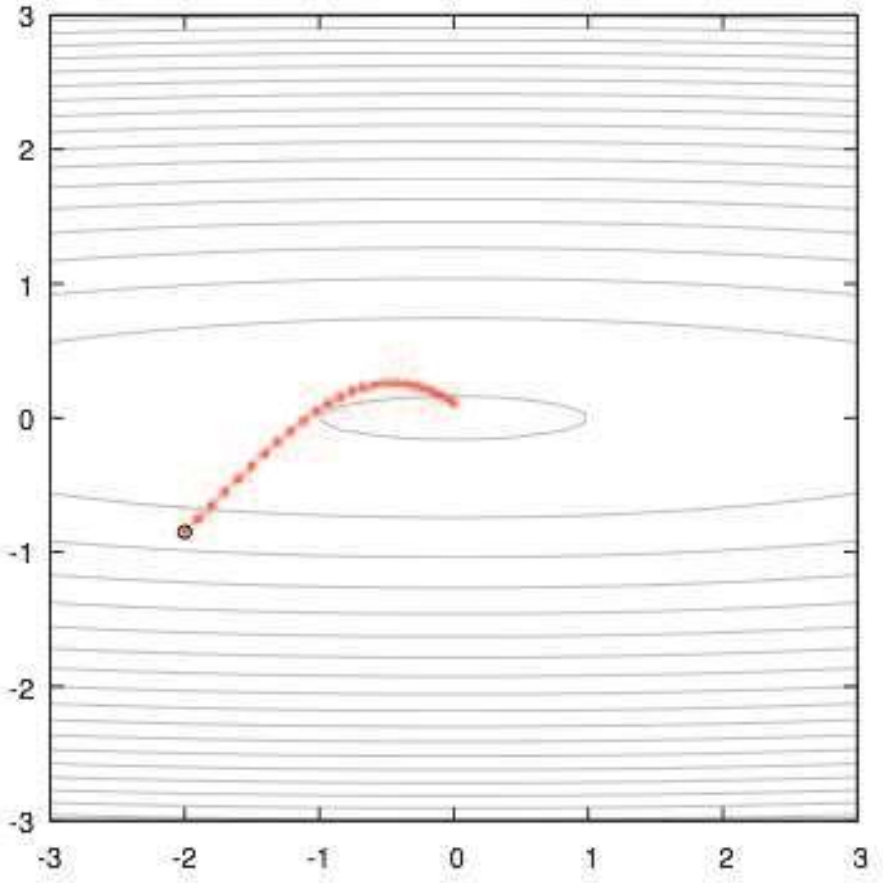
$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
\hat{m}_t &= \frac{m_t}{1-\beta_1} \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{v}_t &= \frac{v_t}{1-\beta_2} \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t
\end{aligned}
$$

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

Kingma and Ba, 2014


$\eta = 5.0e-2$
3
2
1
0
-1
-2
-3
-3
-2
-1
0
1
2
3

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



Optimizers

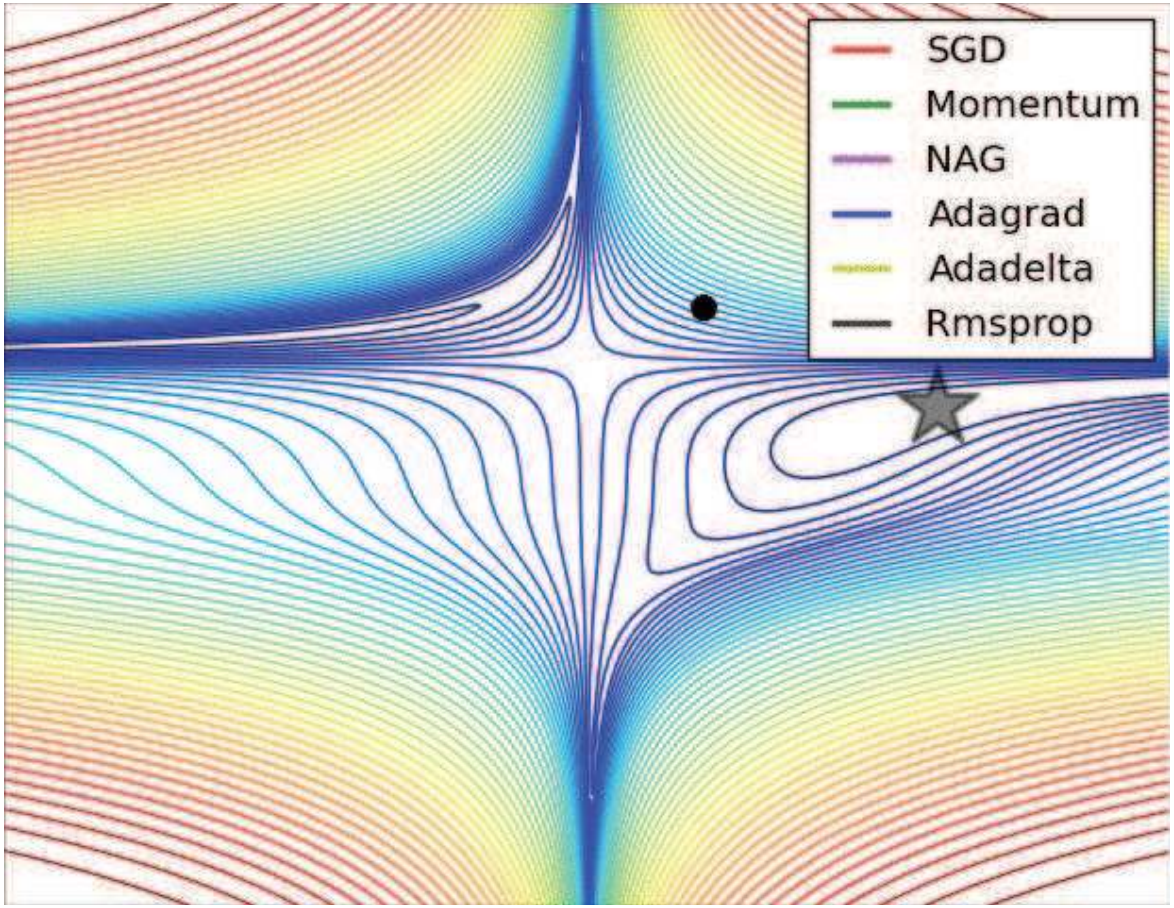
- SGD (with Nesterov momentum)
  - Simple to implement
  - Very sensitive to initial value of η
  - Need learning rate scheduling
- Adam: adaptive learning rate scale for each param
  - Global η set to 0.001 often works well enough
  - Good default choice of optimizer (often)
- But well-tuned SGD with LR scheduling can generalize better than Adam...
- Active area of research

Optimizers

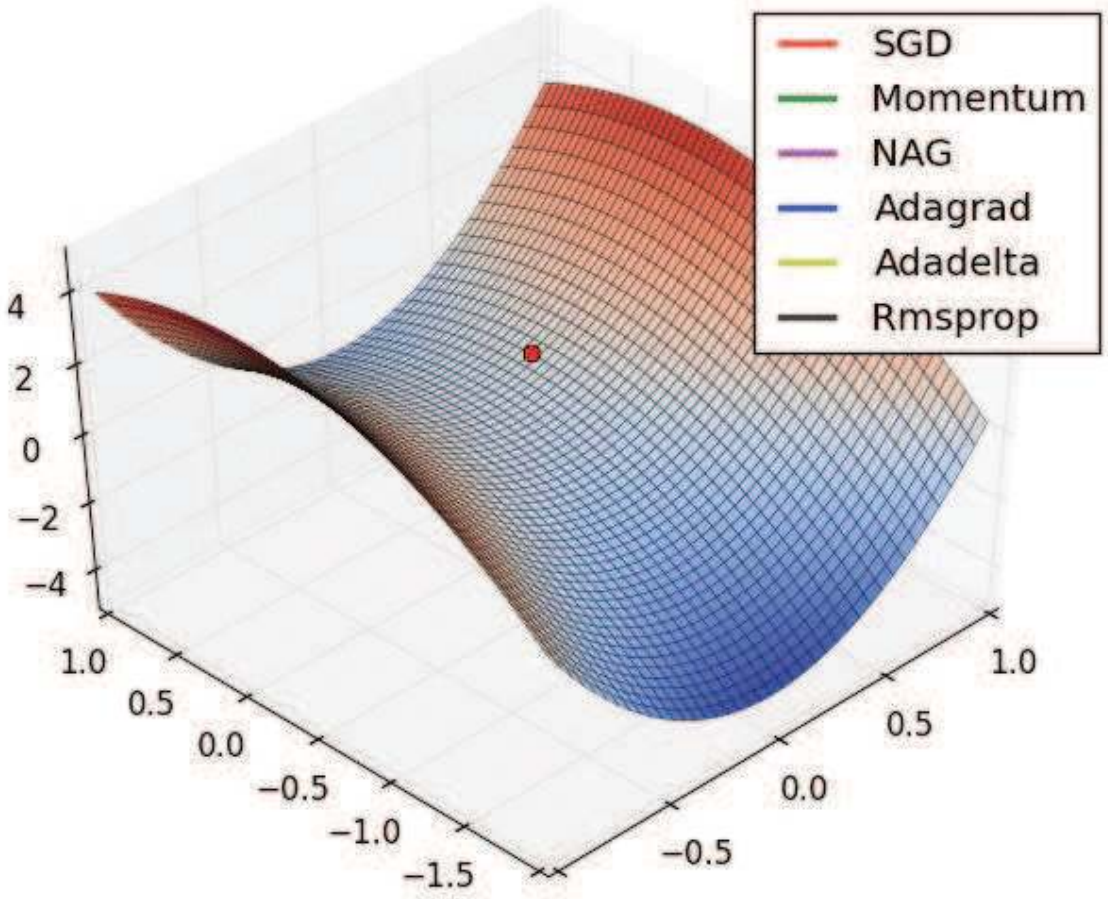
Many other derived optimizers readily available in most frameworks:

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

Optimizers



Optimizers


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

Layers

So far we considered the logistic unit $h = \sigma\left(\mathbf{w}^T\mathbf{x} + b\right)$, where $h \in \mathrm{R}$, $\mathbf{x} \in \mathrm{R}^p$, $\mathbf{w} \in \mathrm{R}^p$ and $b \in \mathrm{R}$.

These units can be composed in parallel to form a layer with q outputs:

$$\mathbf{h} = \sigma(\mathbf{W}^T\mathbf{x} + \mathbf{b})$$

where $\mathbf{h} \in \mathrm{R}^q$, $\mathbf{x} \in \mathrm{R}^p$, $\mathbf{W} \in \mathrm{R}^{p \times q}$, $b \in \mathrm{R}^q$ and where $\sigma(\cdot)$ is upgraded to the element-wise sigmoid function.

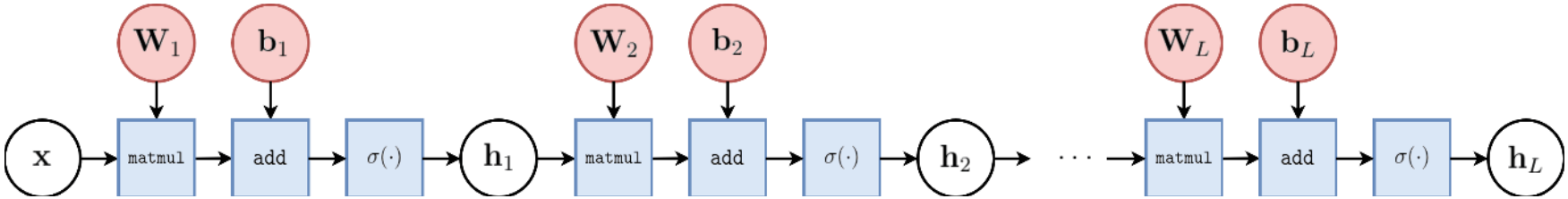
Multi-layer perceptron

Similarly, layers can be composed in series, such that:

$$
\begin{aligned}
\mathbf{h}_0 &= \mathbf{x} \\
\mathbf{h}_1 &= \sigma(\mathbf{W}_1^T \mathbf{h}_0 + \mathbf{b}_1) \\
&\dots \\
\mathbf{h}_L &= \sigma(\mathbf{W}_L^T \mathbf{h}_{L-1} + \mathbf{b}_L) \\
f(\mathbf{x}; \theta) &= \mathbf{h}_L
\end{aligned}
$$

where θ denotes the model parameters $\{\mathbf{W}_k, \mathbf{b}_k, \dots \mid k = 1, \dots, L\}$.

- This model is the multi-layer perceptron, also known as the fully connected feedforward network.
- Optionally, the last activation σ can be skipped to produce unbounded output values $\hat{y} \in \mathrm{R}$.


$\mathbf{W}_1$
$\mathbf{b}_1$
$\mathbf{W}_2$
$\mathbf{b}_2$
$\mathbf{W}_L$
$\mathbf{b}_L$
$\mathbf{x}$
matmul
add
$\sigma(\cdot)$
$\mathbf{h}_1$
matmul
add
$\sigma(\cdot)$
$\mathbf{h}_2$
$\cdots$
matmul
add
$\sigma(\cdot)$
$\mathbf{h}_L$

To minimize $\mathrm{L}(\theta)$ with stochastic gradient descent, we need the gradient $\nabla_\theta \ell(\theta_t)$.

Therefore, we require the evaluation of the (total) derivatives

$$\frac{\mathrm{d}\ell}{\mathrm{d}\mathbf{W}_k}, \frac{\mathrm{d}\ell}{\mathrm{d}\mathbf{b}_k}$$

of the loss ℓ with respect to all model parameters $\mathbf{W}_k$, $\mathbf{b}_k$, for $k = 1, \ldots, L$.

These derivatives can be evaluated automatically from the computational graph of ℓ using automatic differentiation.

Automatic differentiation

Consider a 1-dimensional output composition $f \circ g$, such that

$$y = f(\mathbf{u})$$
$$\mathbf{u} = g(x) = (g_1(x), \ldots, g_m(x)).$$

The chain rule of total derivatives states that

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \sum_{k=1}^{m} \frac{\partial y}{\partial u_k} \underbrace{\frac{\mathrm{d}u_k}{\mathrm{d}x}}_{\text{recursive case}}$$

- Since a neural network is a composition of differentiable functions, the total derivatives of the loss can be evaluated by applying the chain rule recursively over its computational graph.
- The implementation of this procedure is called (reverse) automatic differentiation (AD).
- AD is not numerical differentiation, nor symbolic differentiation.

As a guiding example, let us consider a simplified 2-layer MLP and the following loss function:

$$f(\mathbf{x}; \mathbf{W}_1, \mathbf{W}_2) = \sigma\left(\mathbf{W}_2^T \sigma\left(\mathbf{W}_1^T \mathbf{x}\right)\right)$$

$$\ell(y, \hat{y}; \mathbf{W}_1, \mathbf{W}_2) = \text{cross\_entropy}(y, \hat{y}) + \lambda\left(||\mathbf{W}_1||_2 + ||\mathbf{W}_2||_2\right)$$

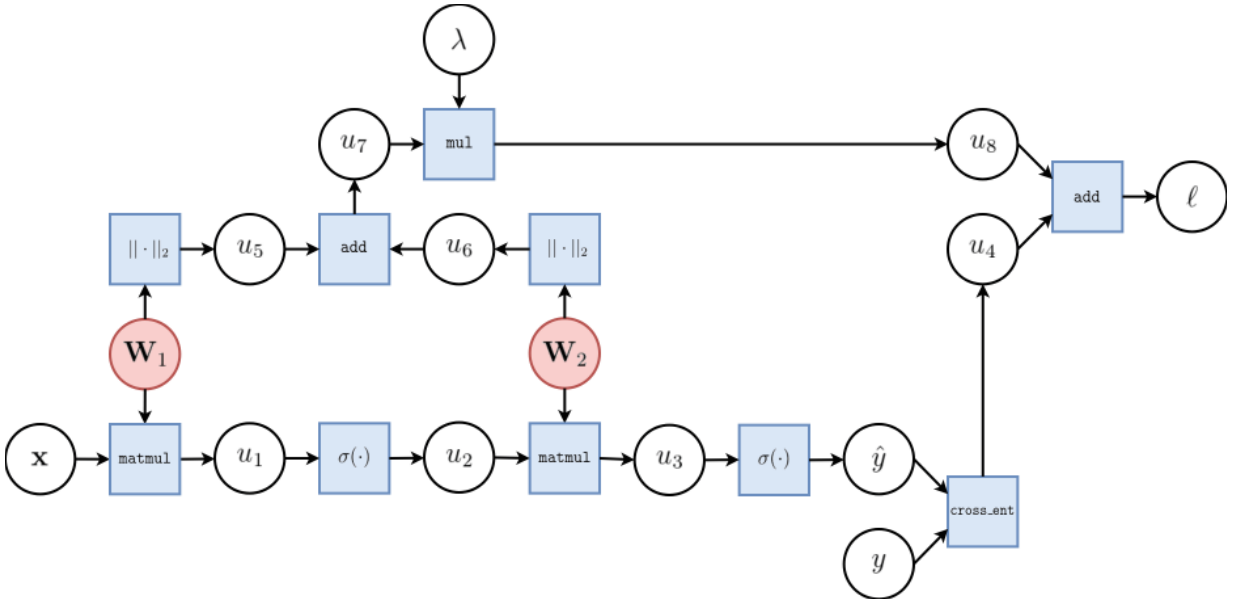
for $\mathbf{x} \in \mathrm{R}^{\mathrm{p}}$, $y \in \mathrm{R}$, $\mathbf{W}_1 \in \mathrm{R}^{p \times q}$ and $\mathbf{W}_2 \in \mathrm{R}^{q}$.

As a guiding example, let us consider a simplified 2-layer MLP and the following loss function:

$$f(\mathbf{x}; \mathbf{W}_1, \mathbf{W}_2) = \sigma\left(\mathbf{W}_2^T \sigma\left(\mathbf{W}_1^T \mathbf{x}\right)\right)$$

$$\ell(y, \hat{y}; \mathbf{W}_1, \mathbf{W}_2) = \text{cross\_entropy}(y, \hat{y}) + \lambda\left(||\mathbf{W}_1||_2 + ||\mathbf{W}_2||_2\right)$$

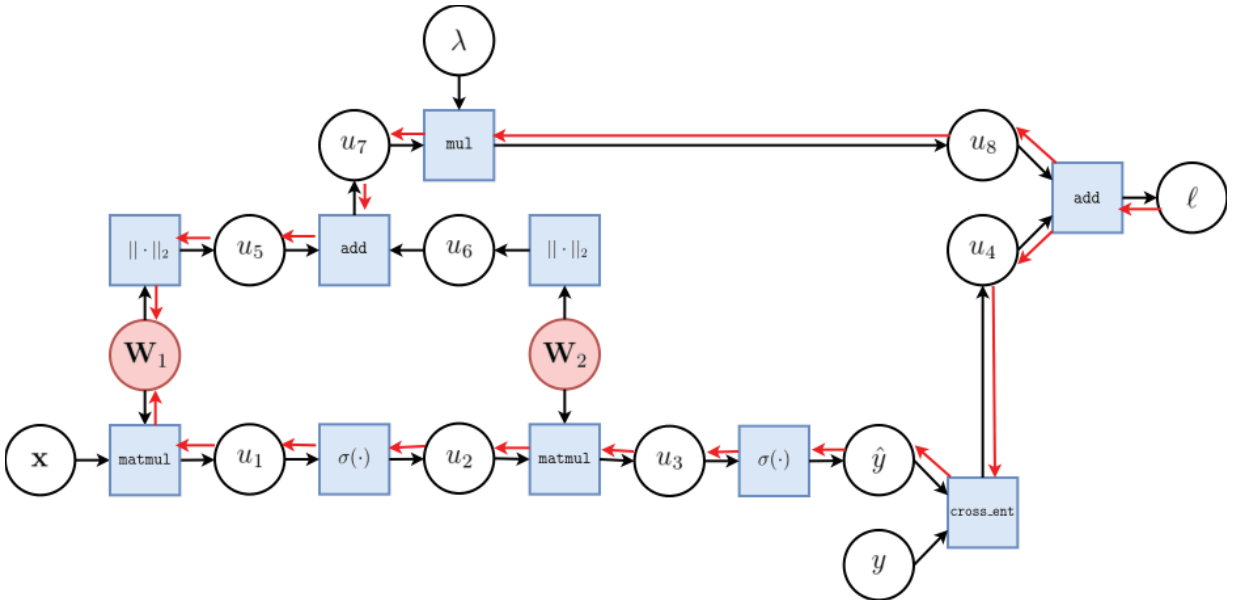
for $\mathbf{x} \in \mathrm{R}^{\mathrm{p}}$, $y \in \mathrm{R}$, $\mathbf{W}_1 \in \mathrm{R}^{p \times q}$ and $\mathbf{W}_2 \in \mathrm{R}^{q}$.



The total derivative $\frac{\mathrm{d}\ell}{\mathrm{d}\mathbf{W}_1}$ can be computed backward, by walking through all paths from ℓ to $\mathbf{W}_1$ in the computational graph and accumulating the terms:

$$\frac{\mathrm{d}\ell}{\mathrm{d}\mathbf{W}_1} = \frac{\partial \ell}{\partial u_8}\frac{\mathrm{d}u_8}{\mathrm{d}\mathbf{W}_1} + \frac{\partial \ell}{\partial u_4}\frac{\mathrm{d}u_4}{\mathrm{d}\mathbf{W}_1}$$

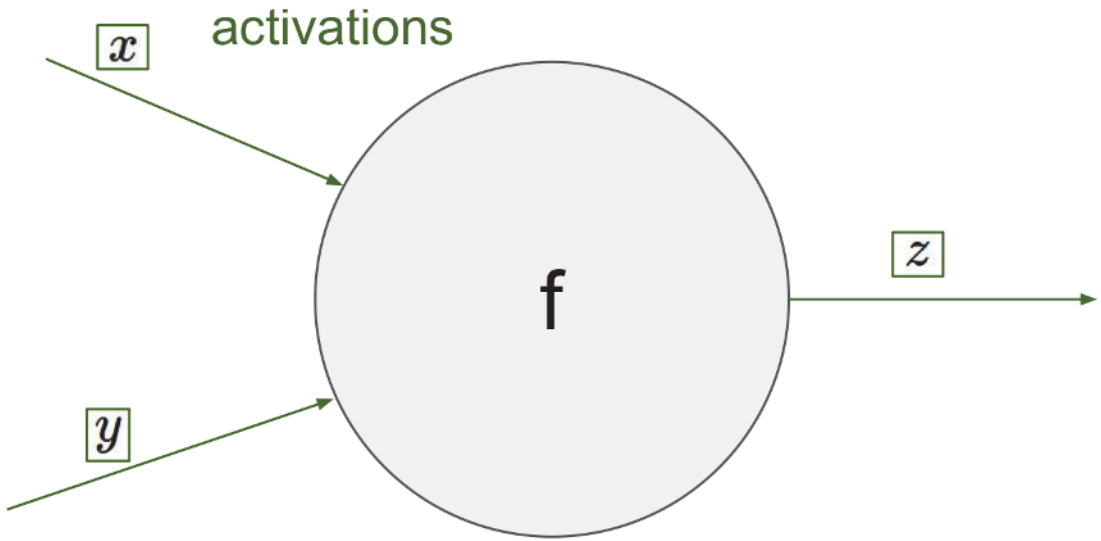
$$\frac{\mathrm{d}u_8}{\mathrm{d}\mathbf{W}_1} = \dots$$



- This algorithm is known as reverse-mode automatic differentiation, also called backpropagation.
- An equivalent procedure can be defined to evaluate the derivatives in forward mode, from inputs to outputs.
- Automatic differentiation generalizes to N inputs and M outputs.
  - if $N \gg M$, reverse-mode automatic differentiation is computationally more efficient.
  - otherwise, if $M \gg N$, forward automatic differentiation is better.
- Since differentiation is a linear operator, AD can be implemented efficiently in terms of matrix operations.

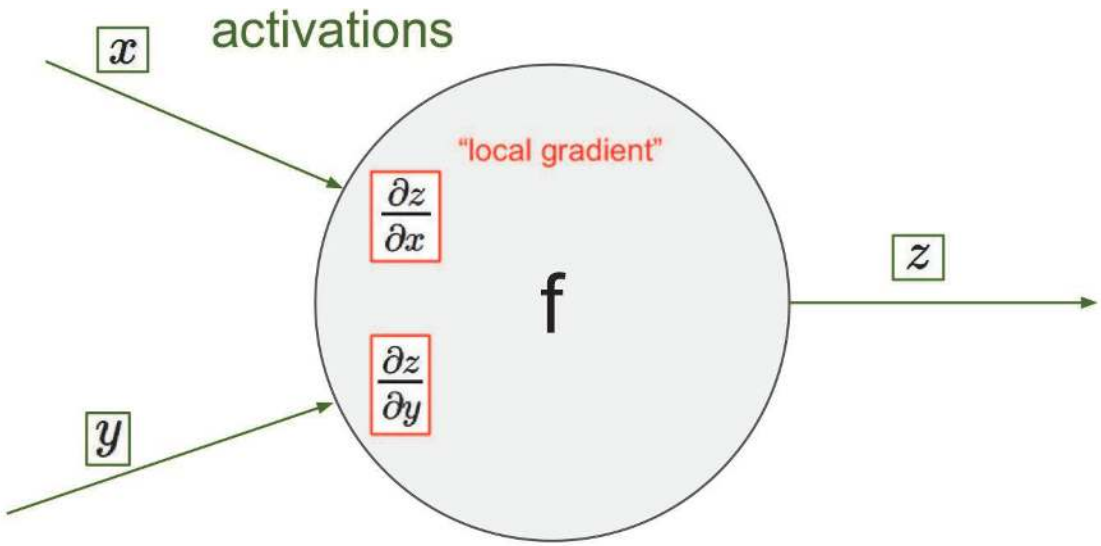
Backpropagation

Inside a single unit/neuron/function



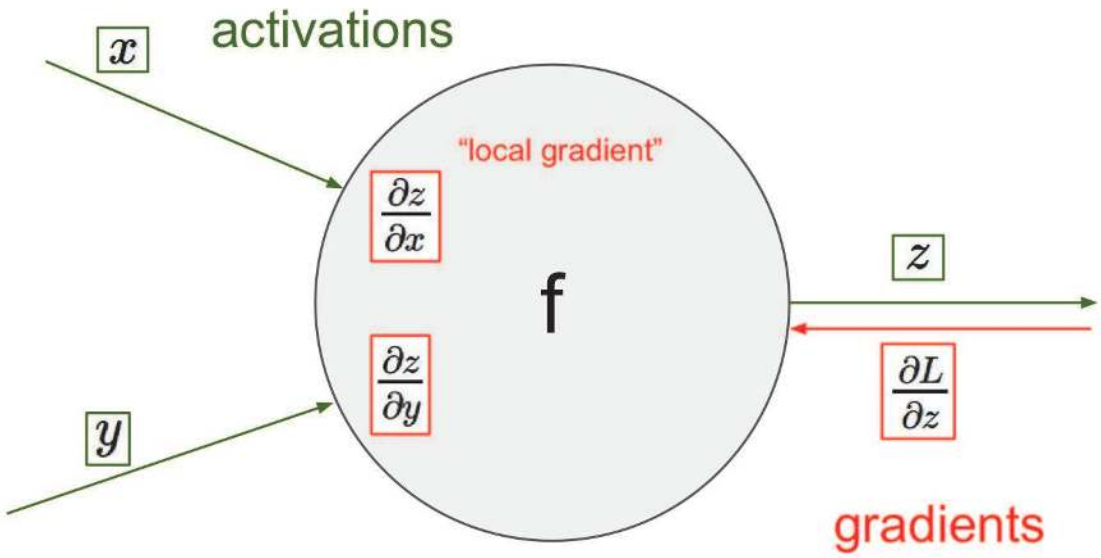
Backpropagation

Inside a single unit/neuron/function



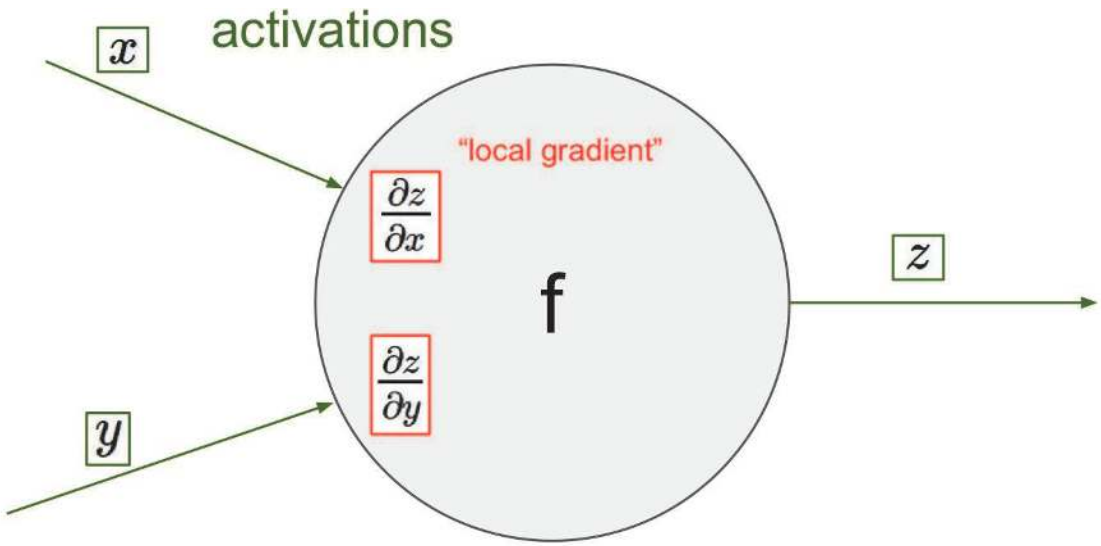
Backpropagation

Inside a single unit/neuron/function



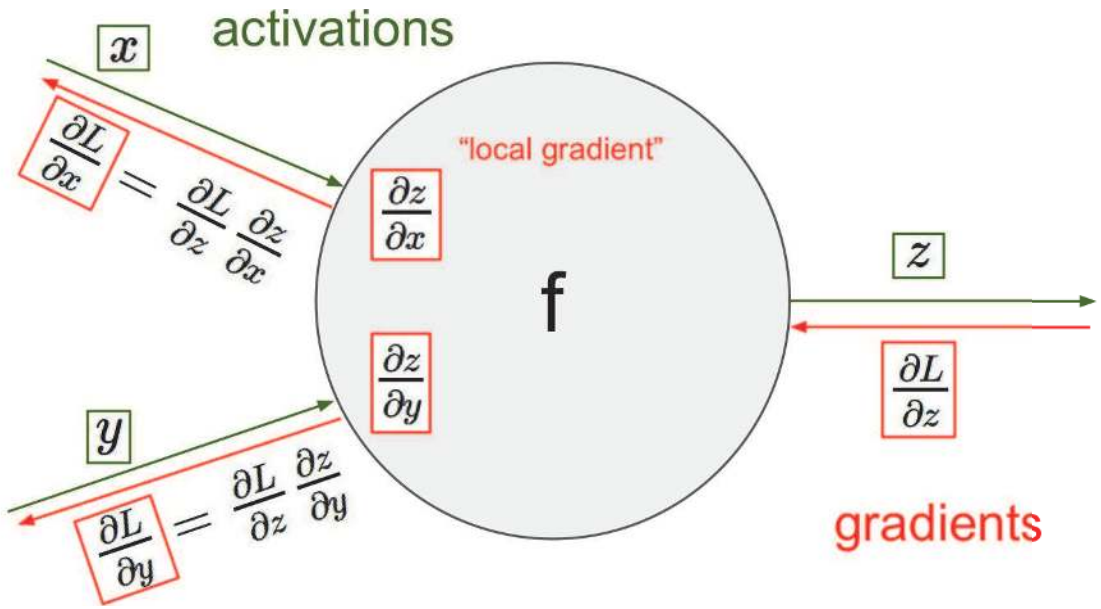
Backpropagation

Inside a single unit/neuron/function



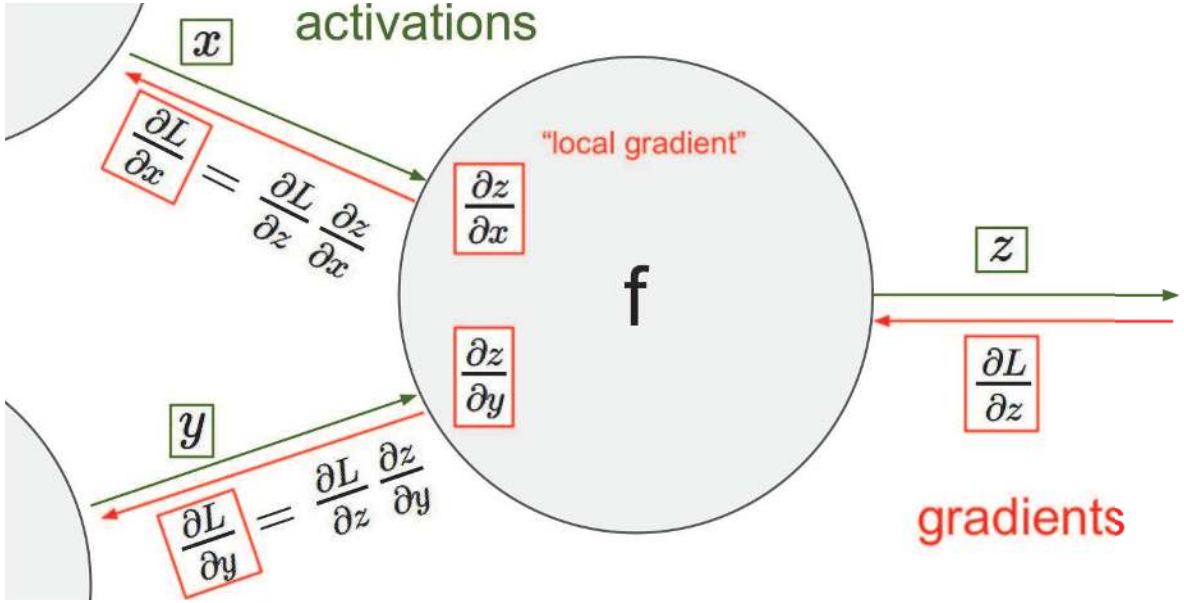
Backpropagation

Inside a single unit/neuron/function



Backpropagation

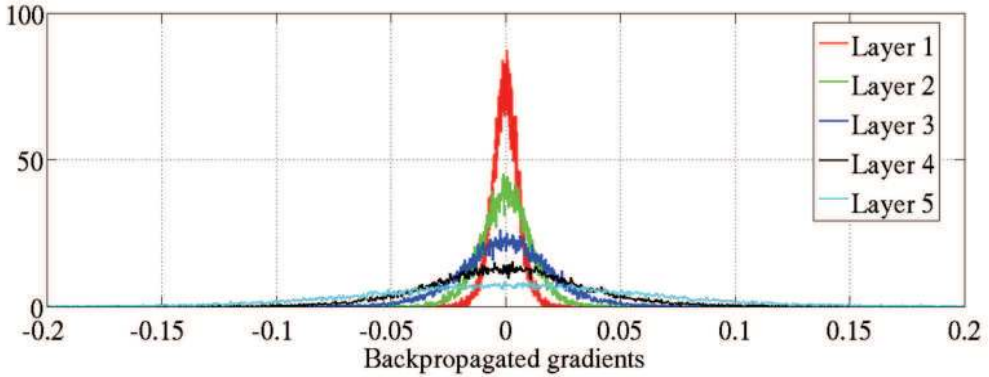
Inside a single unit/neuron/function



Vanishing gradients

Training deep MLPs with many layers has for long (pre-2011) been very difficult due to the vanishing gradient problem.

- Small gradients slow down, and eventually block, stochastic gradient descent.
- This results in a limited capacity of learning.



Backpropagated gradients normalized histograms (Glorot and Bengio, 2010).
Gradients for layers far from the output vanish to zero.

Consider a simplified 3-layer MLP, with $x, w_1, w_2, w_3 \in \mathrm{R}$, such that

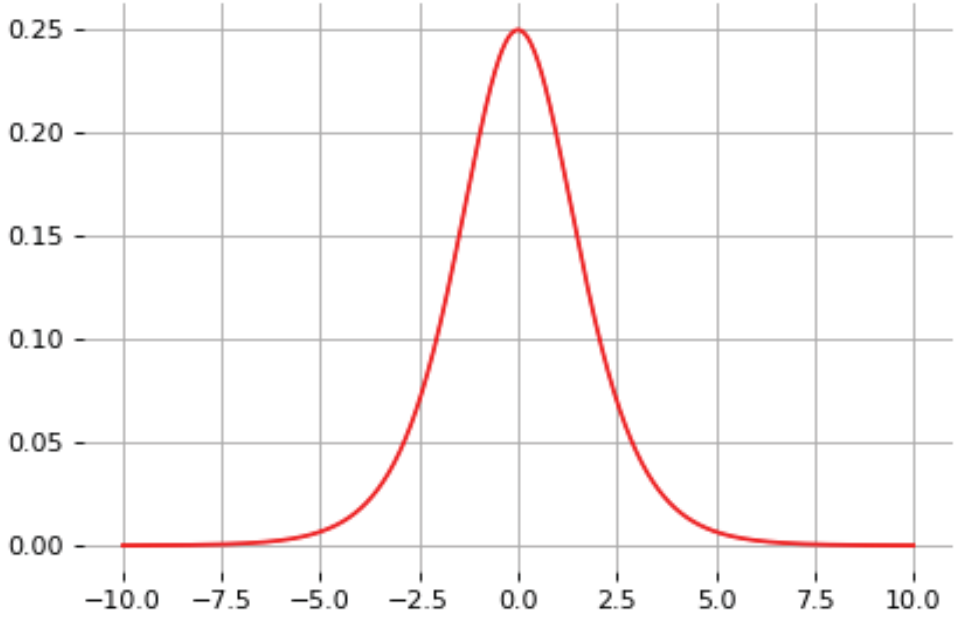
$$f(x; w_1, w_2, w_3) = \sigma\Big(w_3\sigma\Big(w_2\sigma\Big(w_1 x\Big)\Big)\Big).$$

Under the hood, this would be evaluated as

$$\begin{aligned} u_1 &= w_1 x \\ u_2 &= \sigma(u_1) \\ u_3 &= w_2 u_2 \\ u_4 &= \sigma(u_3) \\ u_5 &= w_3 u_4 \\ \hat{y} &= \sigma(u_5) \end{aligned}$$

and its derivative $\frac{\mathrm{d}\hat{y}}{\mathrm{d}w_1}$ as

The derivative of the sigmoid activation function σ is:



$$\frac{\mathrm{d}\sigma}{\mathrm{d}x}(x) = \sigma(x)(1 - \sigma(x))$$

Notice that $0 \leq \frac{\mathrm{d}\sigma}{\mathrm{d}x}(x) \leq \frac{1}{4}$ for all x.

Assume that weights w_1, w_2, w_3 are initialized randomly from a Gaussian with zero-mean and small variance, such that with high probability $-1 \leq w_i \leq 1$.

Then,

$$\frac{\mathrm{d}\hat{y}}{\mathrm{d}w_1} = \underbrace{\frac{\partial \sigma(u_5)}{\partial u_5}}_{\leq \frac{1}{4}} \underbrace{w_3}_{\leq 1} \underbrace{\frac{\partial \sigma(u_3)}{\partial u_3}}_{\leq \frac{1}{4}} \underbrace{w_2}_{\leq 1} \underbrace{\frac{\sigma(u_1)}{\partial u_1}}_{\leq \frac{1}{4}} x$$

This implies that the gradient $\frac{\mathrm{d}\hat{y}}{\mathrm{d}w_1}$ exponentially shrinks to zero as the number of layers in the network increases.

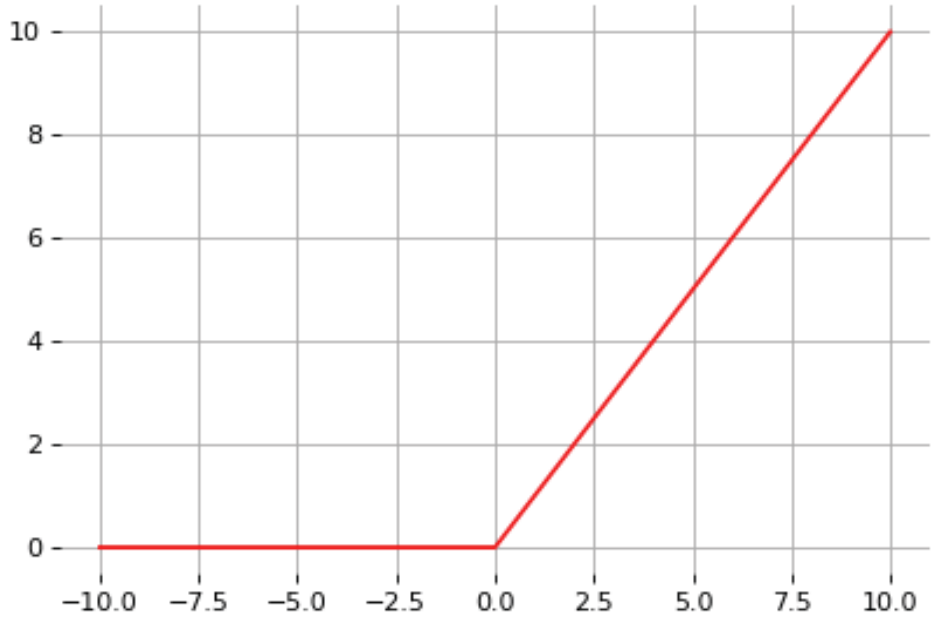
Hence the vanishing gradient problem.

- In general, bounded activation functions (sigmoid, tanh, etc) are prone to the vanishing gradient problem.
- Note the importance of a proper initialization scheme.

Rectified linear units

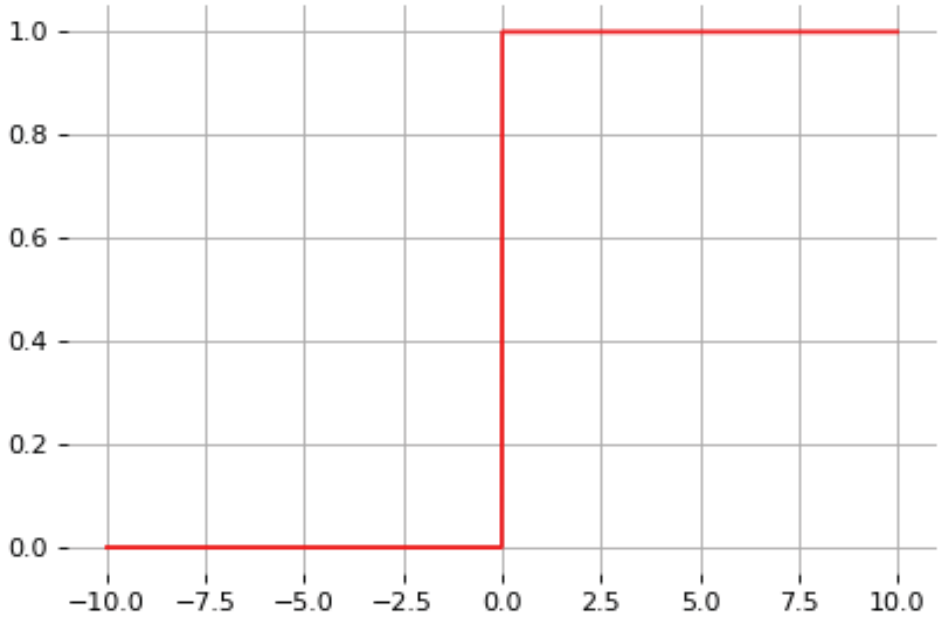
Instead of the sigmoid activation function, modern neural networks are for most based on rectified linear units (ReLU) (Glorot et al, 2011):

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



Note that the derivative of the ReLU function is

$$\frac{\mathrm{d}}{\mathrm{d}x}\mathrm{ReLU}(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{otherwise} \end{cases}$$



For $x = 0$, the derivative is undefined. In practice, it is set to zero.

Therefore,

$$\frac{\mathrm{d}\hat{y}}{\mathrm{d}w_1} = \underbrace{\frac{\partial\sigma(u_5)}{\partial u_5}}_{=1} w_3 \underbrace{\frac{\partial\sigma(u_3)}{\partial u_3}}_{=1} w_2 \underbrace{\frac{\partial\sigma(u_1)}{\partial u_1}}_{=1} x$$

This solves the vanishing gradient problem, even for deep networks! (provided proper initialization)

Note that:

- The ReLU unit dies when its input is negative, which might block gradient descent.
- This is actually a useful property to induce sparsity.
- This issue can also be solved using leaky ReLUs, defined as

$$\mathrm{LeakyReLU}(x) = \max(\alpha x, x)$$

for a small $\alpha \in \mathrm{R}^+$ (e.g., $\alpha = 0.1$).

Universal approximation

Theorem. (Cybenko 1989; Hornik et al, 1991) Let $\sigma(\cdot)$ be a bounded, non-constant continuous function. Let I_p denote the p-dimensional hypercube, and $C(I_p)$ denote the space of continuous functions on I_p. Given any $f \in C(I_p)$ and $\epsilon > 0$, there exists $q > 0$ and $v_i, w_i, b_i, i = 1, \ldots, q$ such that

$$F(x) = \sum_{i \leq q} v_i \sigma(w_i^T x + b_i)$$

satisfies $\sup_{x \in I_p} |f(x) - F(x)| < \epsilon$.

- It guarantees that even a single hidden-layer network can represent any classification problem in which the boundary is locally linear (smooth);
- It does not inform about good/bad architectures, nor how they relate to the optimization procedure.

Theorem (Barron, 1992) The mean integrated square error between the estimated network $\hat{F}$ and the target function f is bounded by

$$O\left(\frac{C_f^2}{q} + \frac{qp}{N}\log N\right)$$

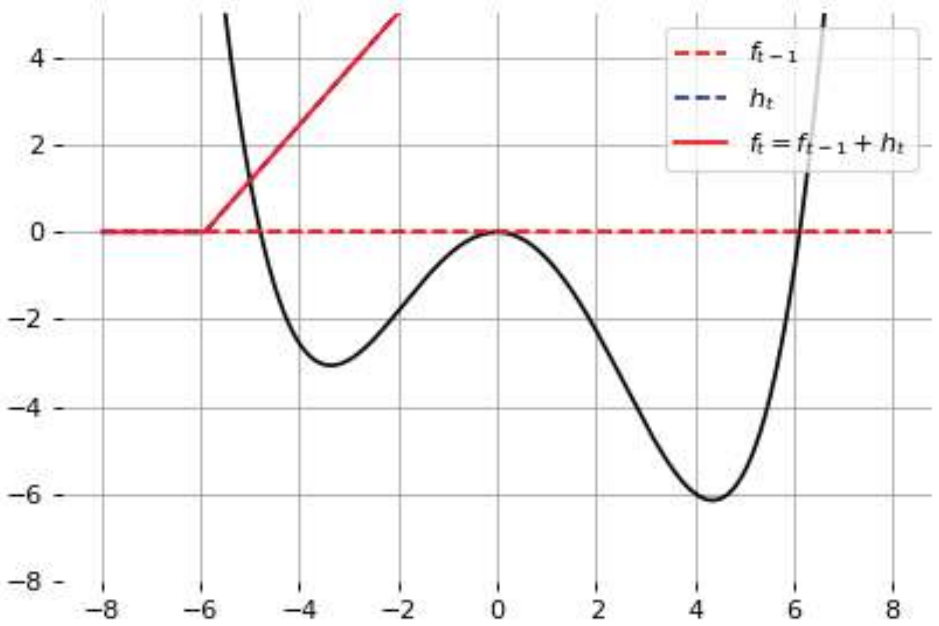
where N is the number of training points, q is the number of neurons, p is the input dimension, and C_f measures the global smoothness of f.

- Combines approximation and estimation errors.
- Provided enough data, it guarantees that adding more neurons will result in a better approximation.

Consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

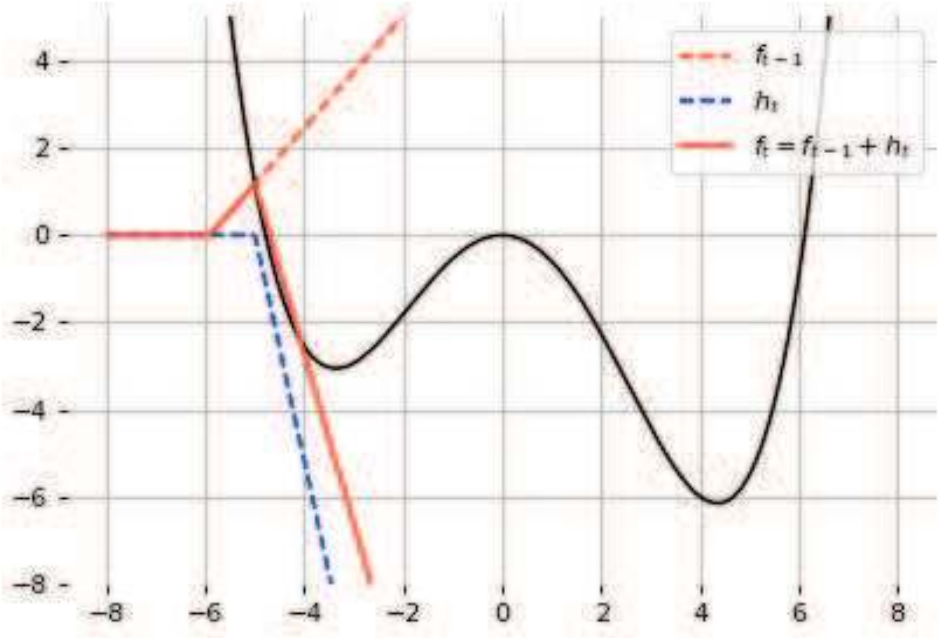
This model can approximate any smooth 1D function, provided enough hidden units.



Consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

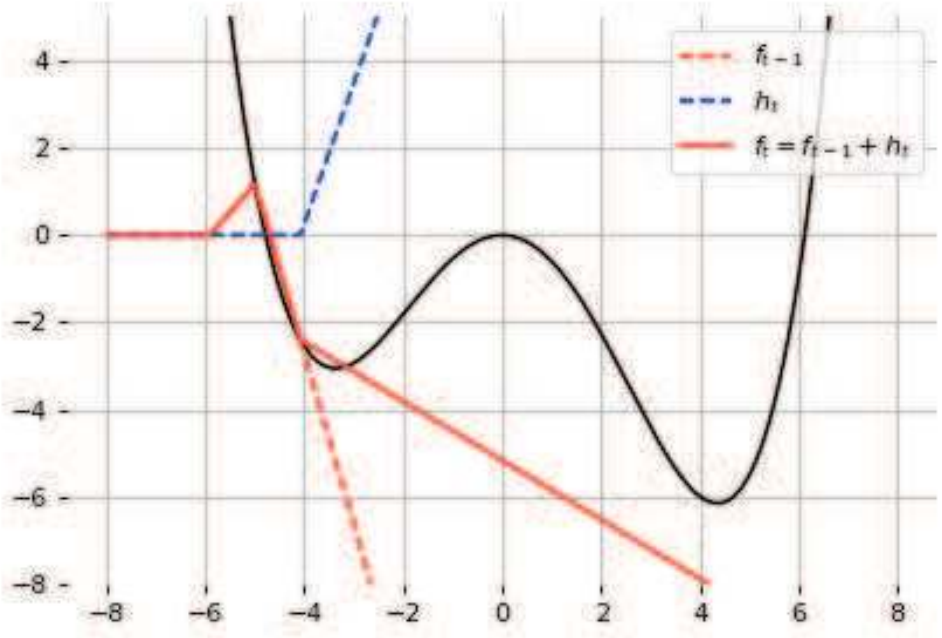
This model can approximate any smooth 1D function, provided enough hidden units.



Consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

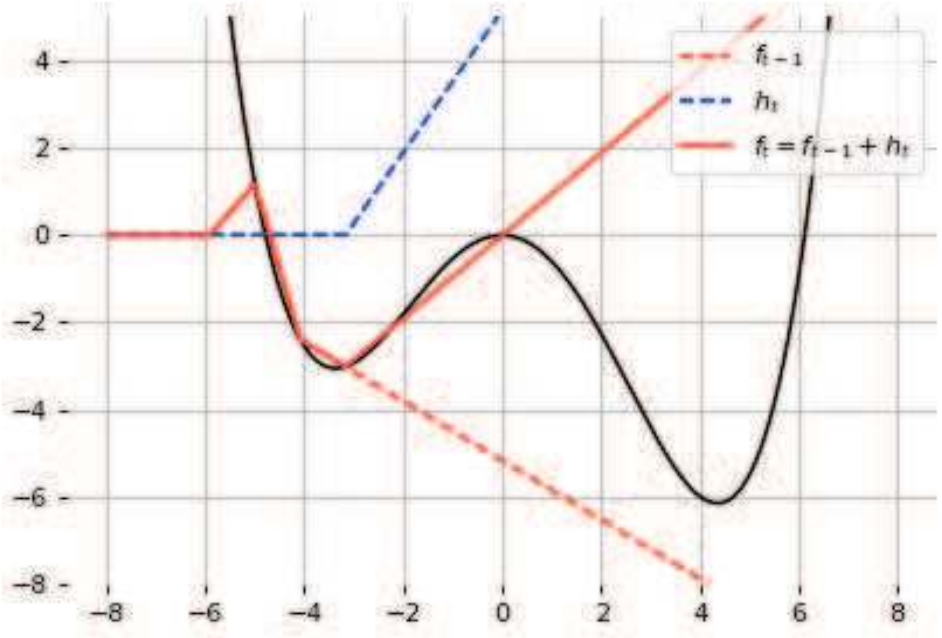
This model can approximate any smooth 1D function, provided enough hidden units.



Consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

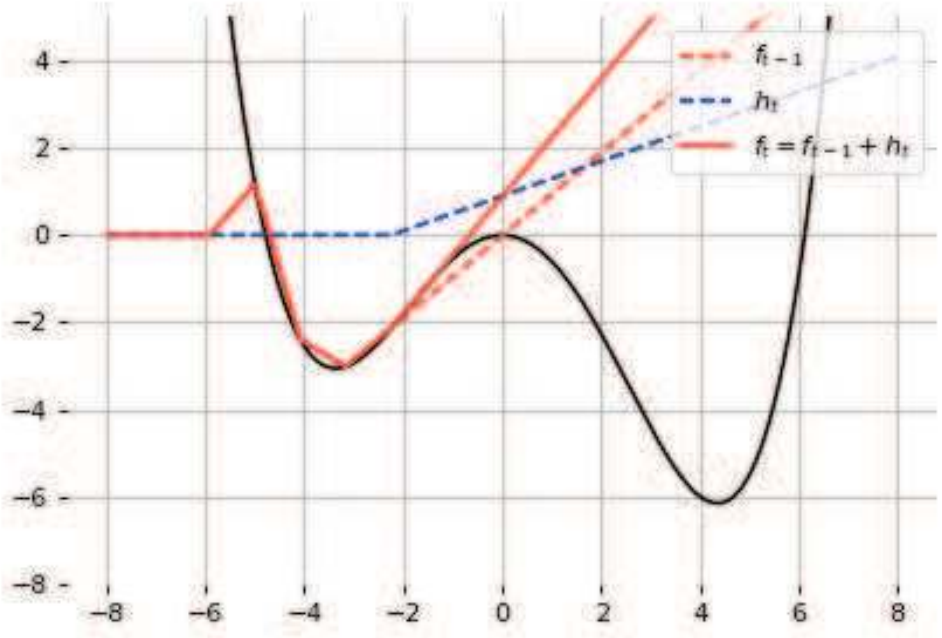
This model can approximate any smooth 1D function, provided enough hidden units.



Consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

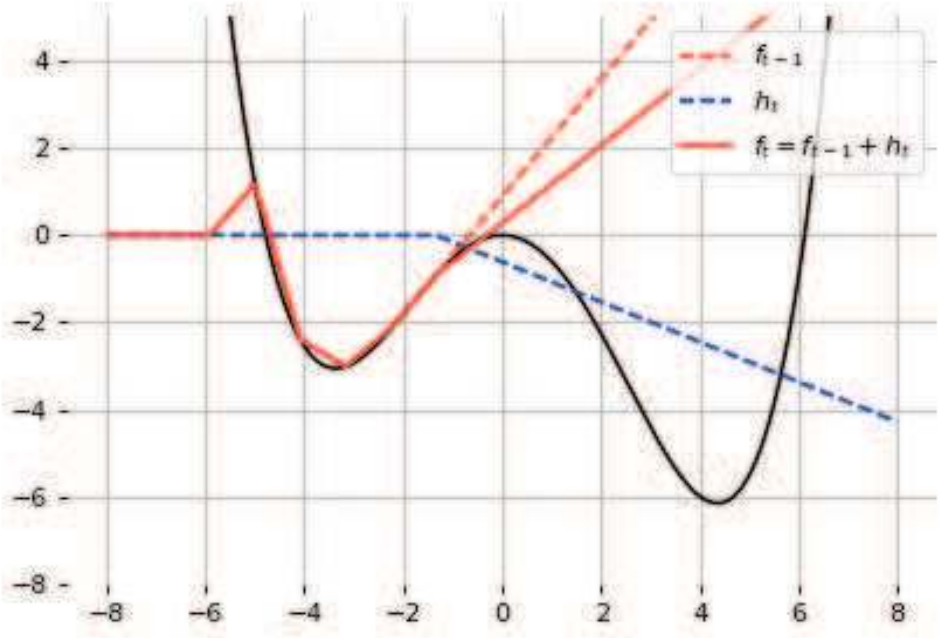
This model can approximate any smooth 1D function, provided enough hidden units.



Consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

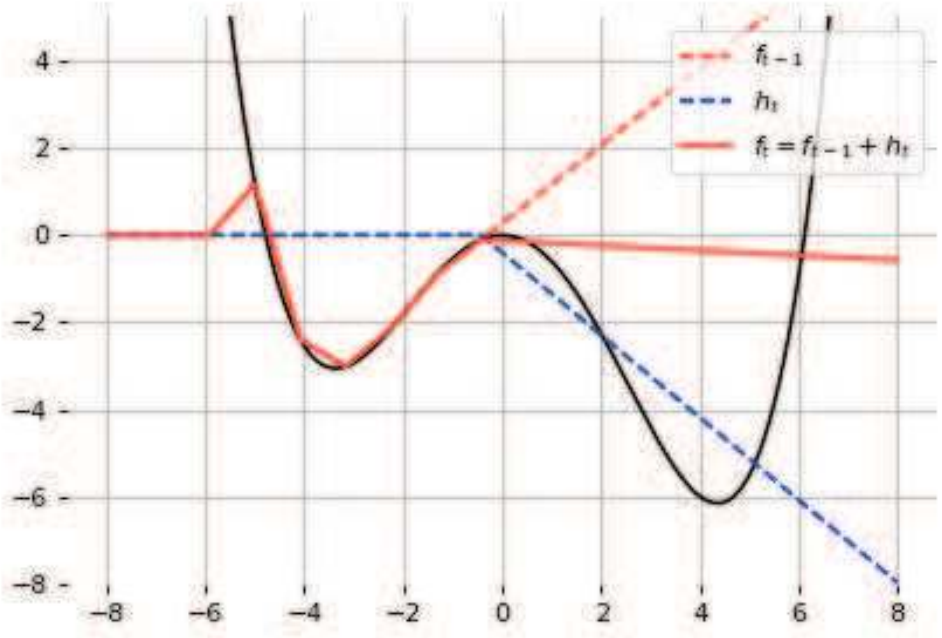
This model can approximate any smooth 1D function, provided enough hidden units.



Consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

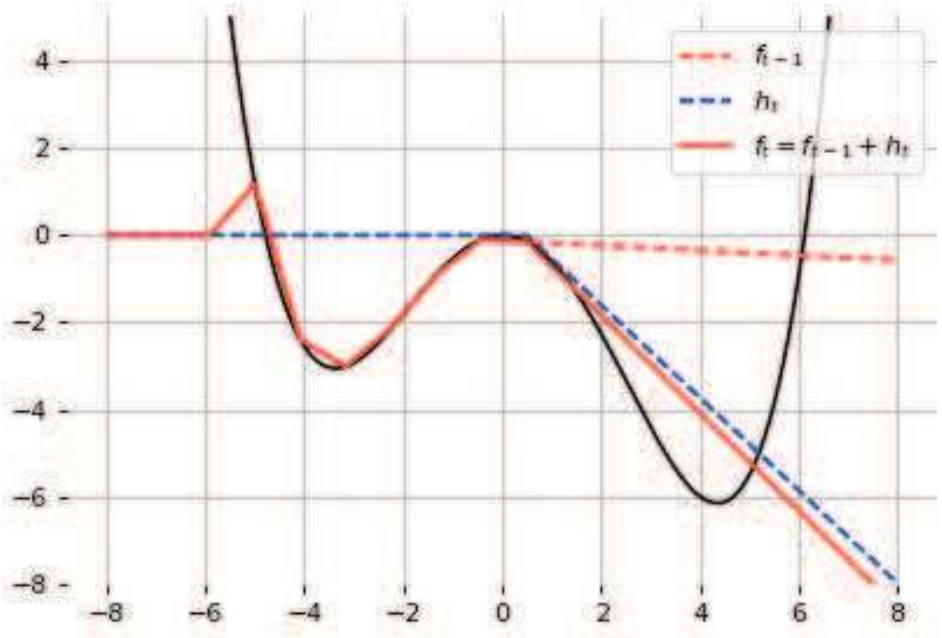
This model can approximate any smooth 1D function, provided enough hidden units.



Consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

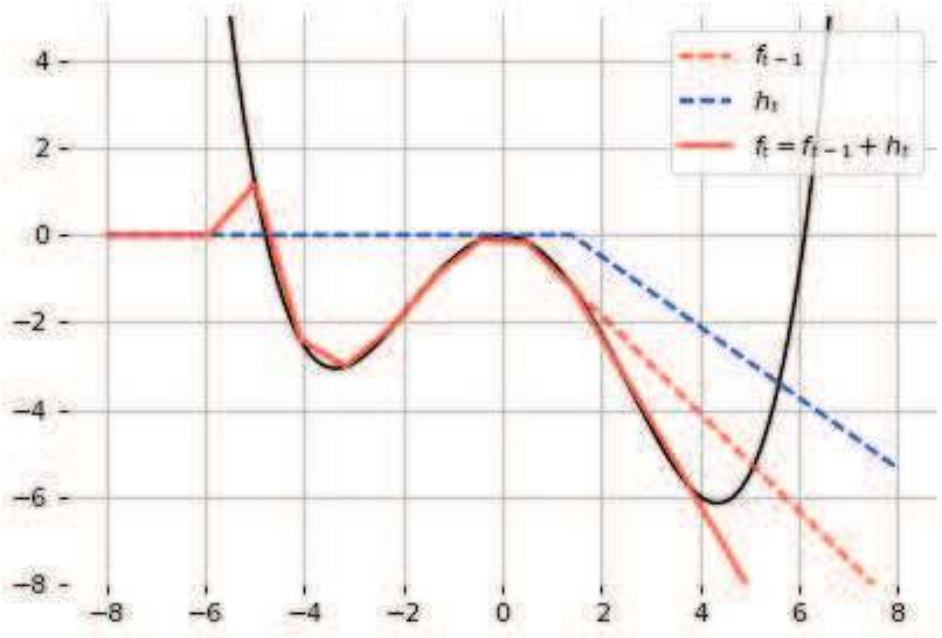
This model can approximate any smooth 1D function, provided enough hidden units.



Consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

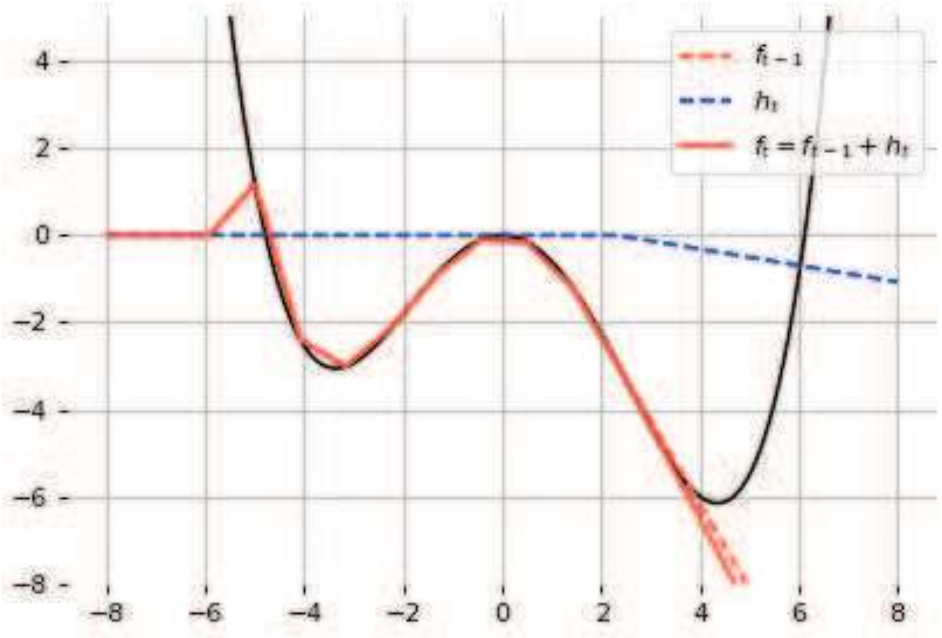
This model can approximate any smooth 1D function, provided enough hidden units.



Consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

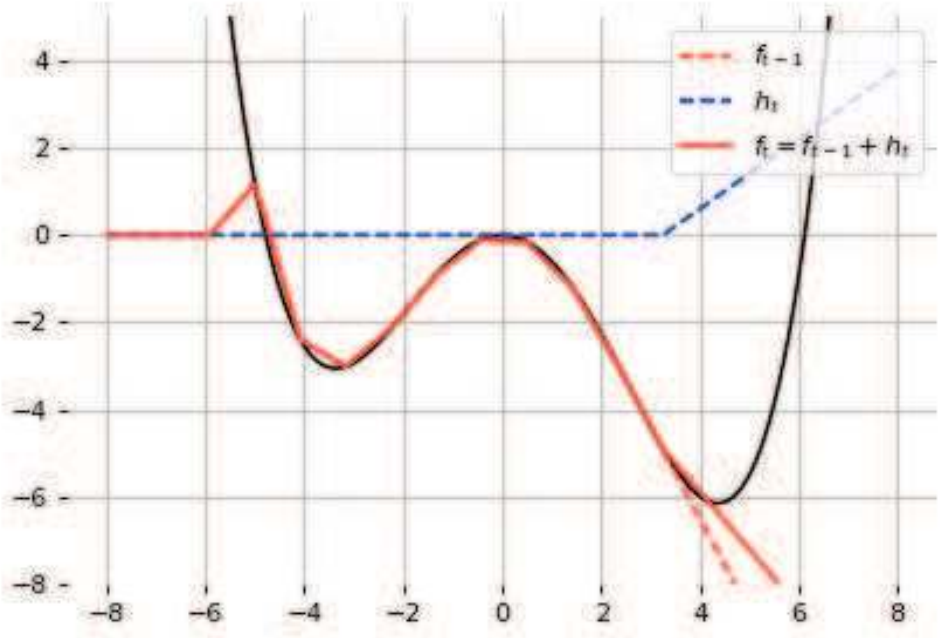
This model can approximate any smooth 1D function, provided enough hidden units.



Consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

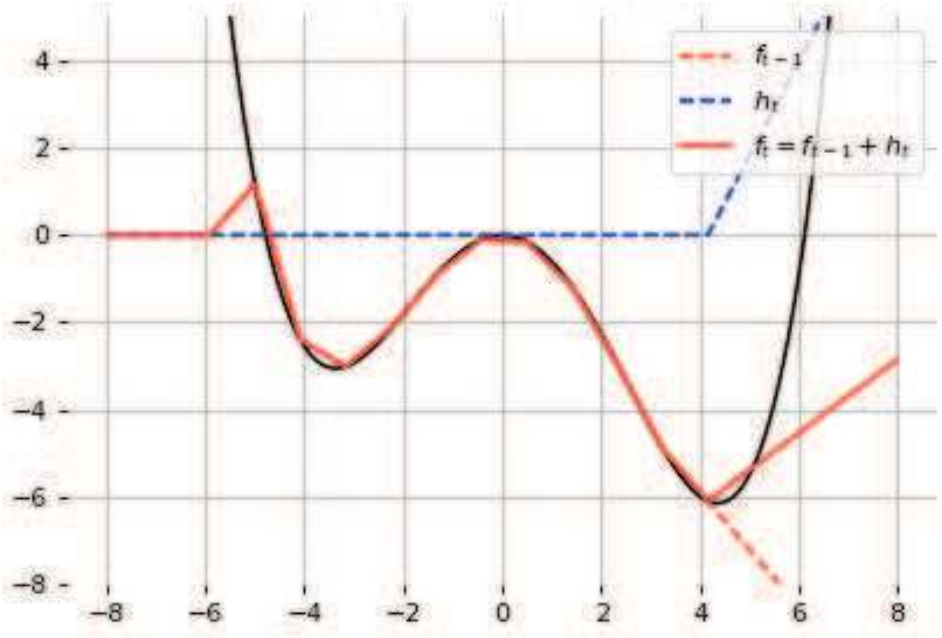
This model can approximate any smooth 1D function, provided enough hidden units.



Consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

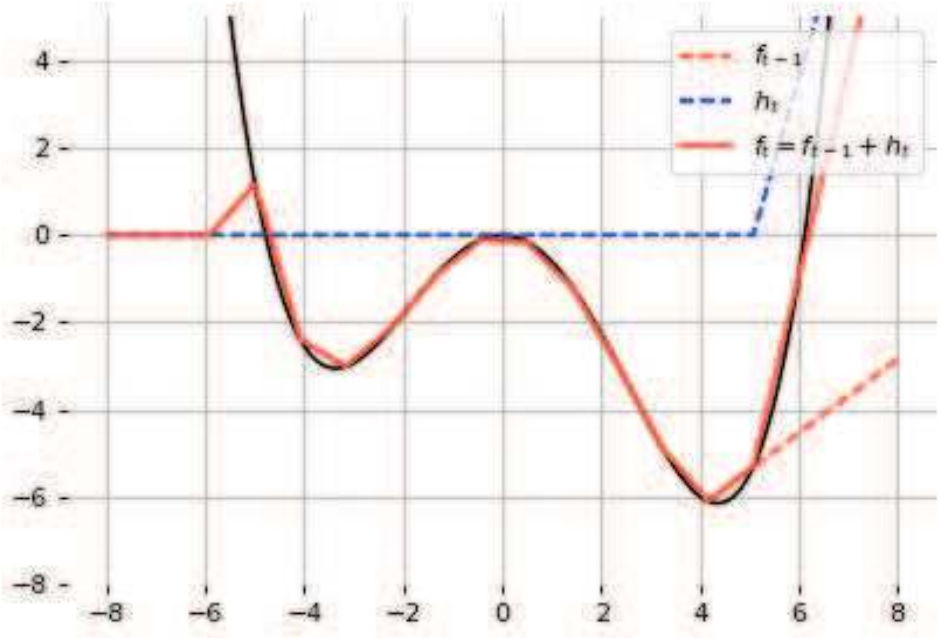
This model can approximate any smooth 1D function, provided enough hidden units.



Consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

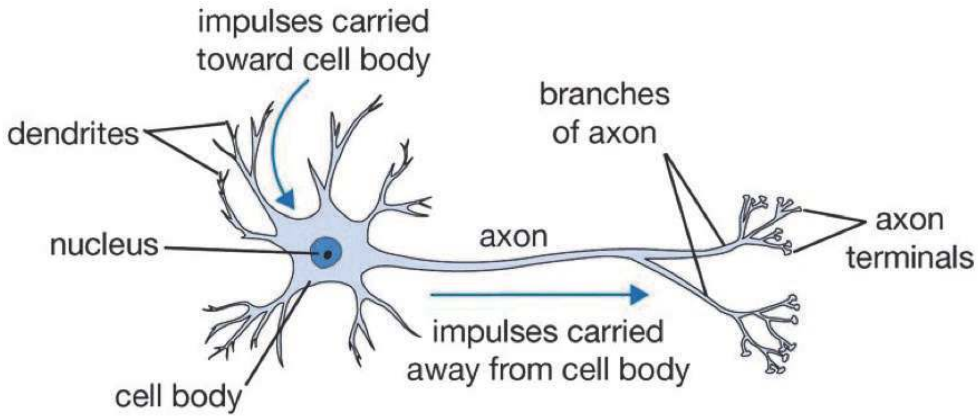
This model can approximate any smooth 1D function, provided enough hidden units.



Neural Network for classification

The neuron

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



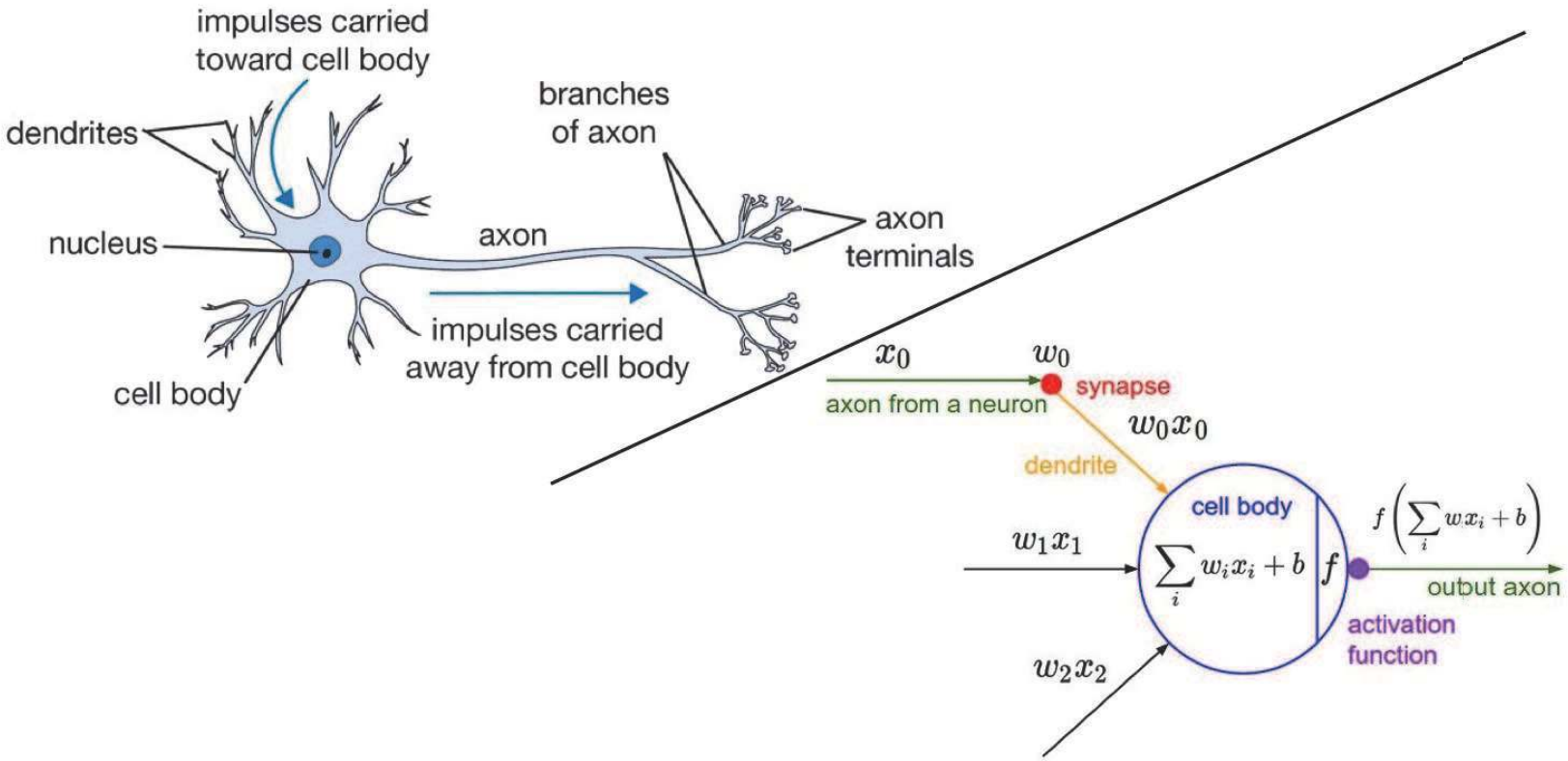
- In fact there several types of neurons with different functions and the metaphor does not hold everywhere

Slide credit: A. Karpathy

Neural Network for classification

The neuron

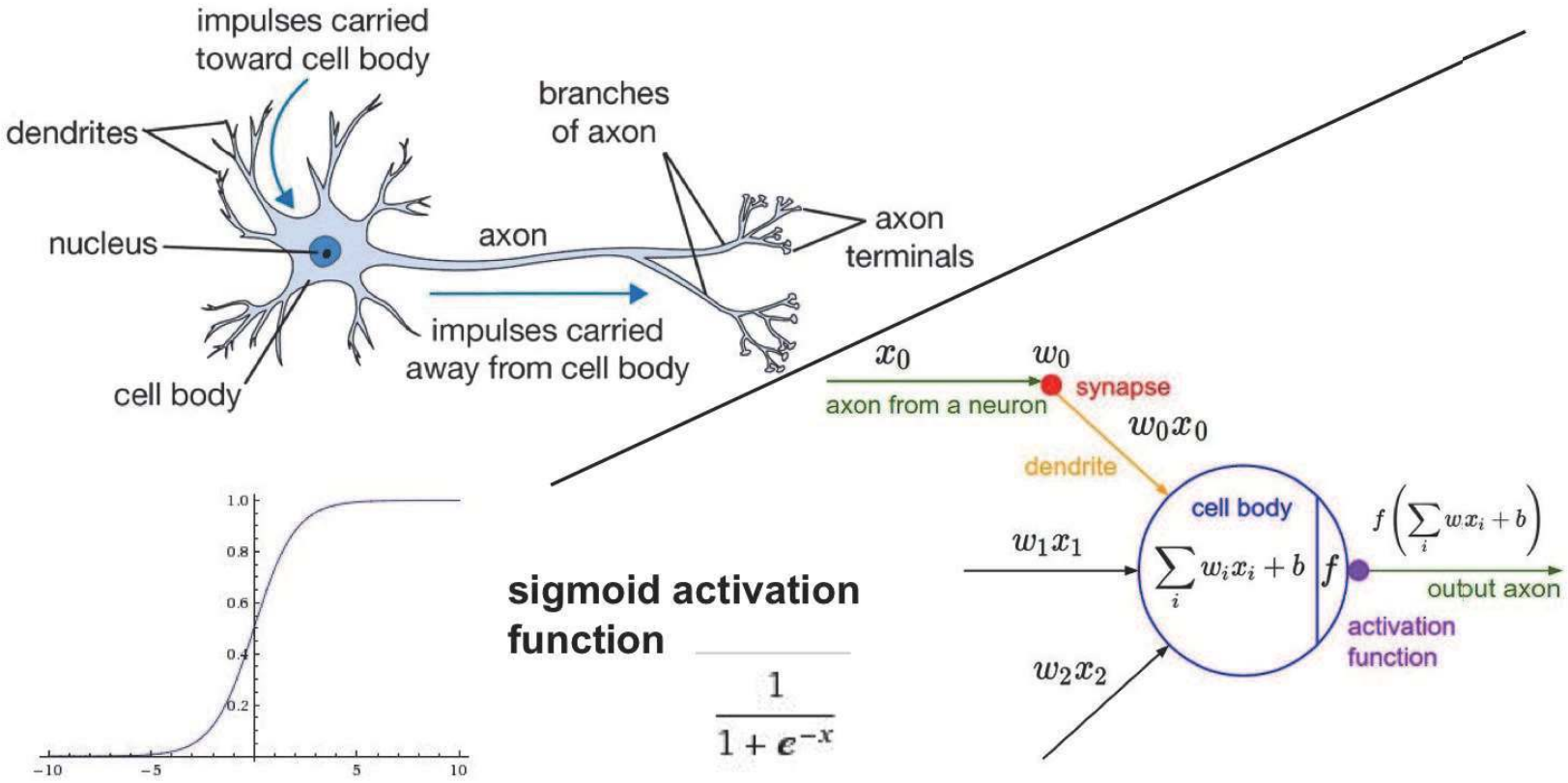
Inspired by neuroscience and human brain, but resemblances do not go too far



Slide credit: A. Karpathy

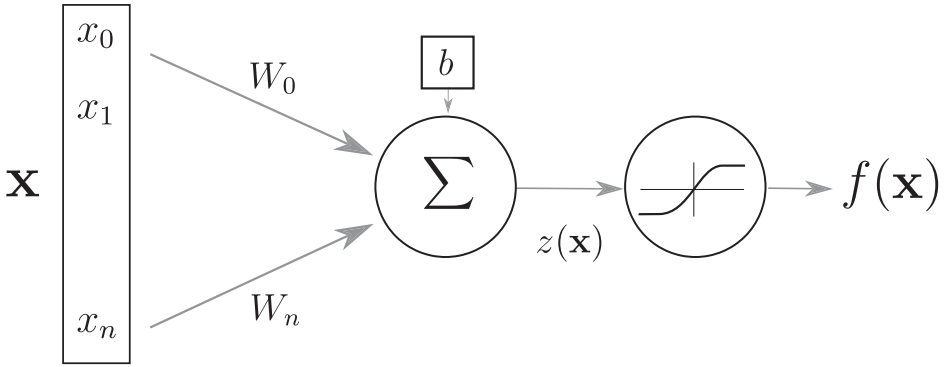
Neural Network for classification

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



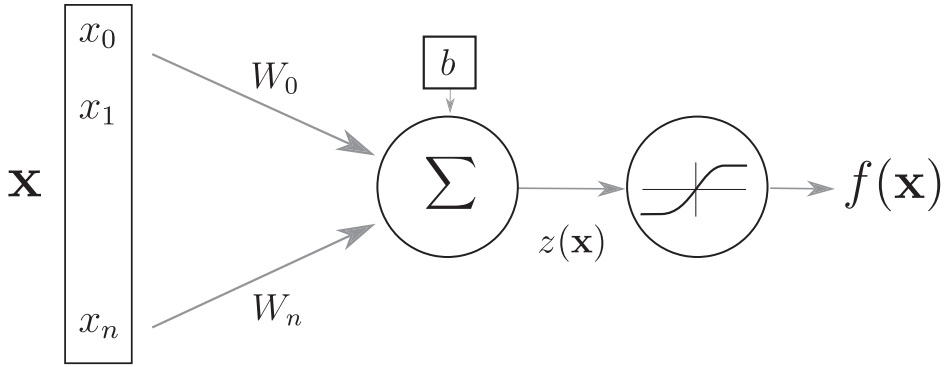
Slide credit: A. Karpathy

Artificial Neuron



Slide credit: C. Ollion & O. Grisel

Artificial Neuron



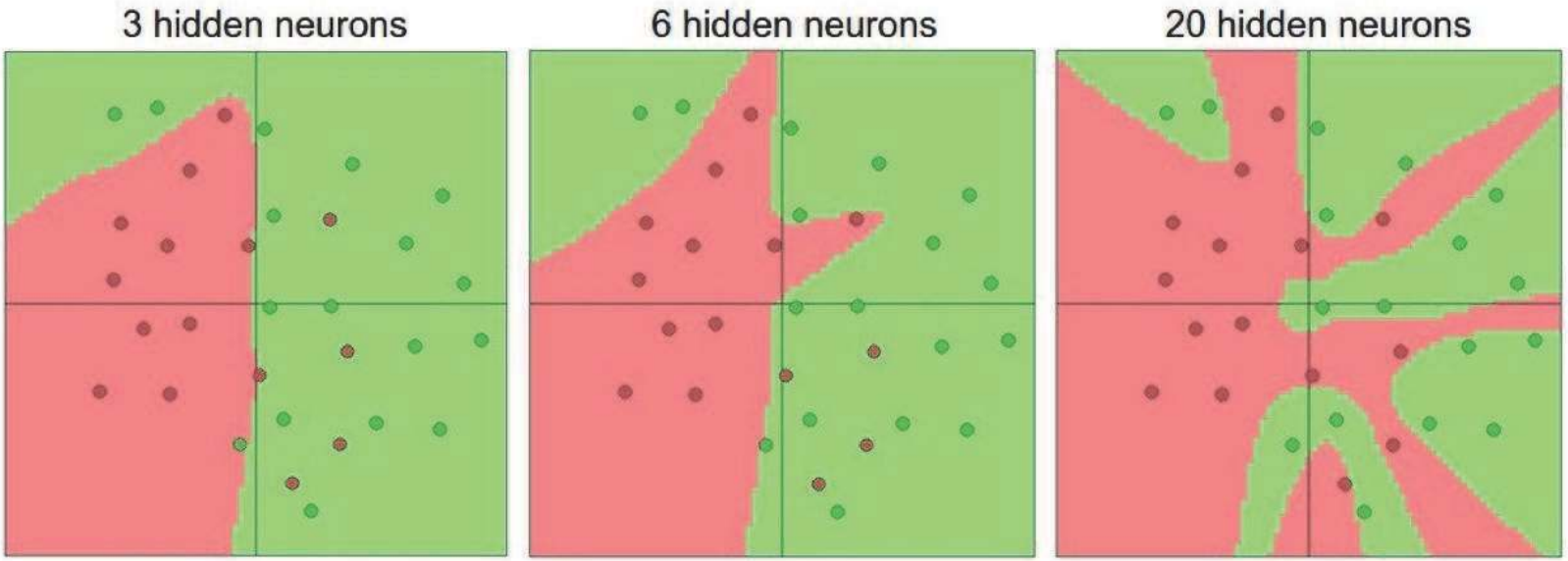
$$z(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$$

$$f(\mathbf{x}) = g(\mathbf{w}^T\mathbf{x} + b)$$

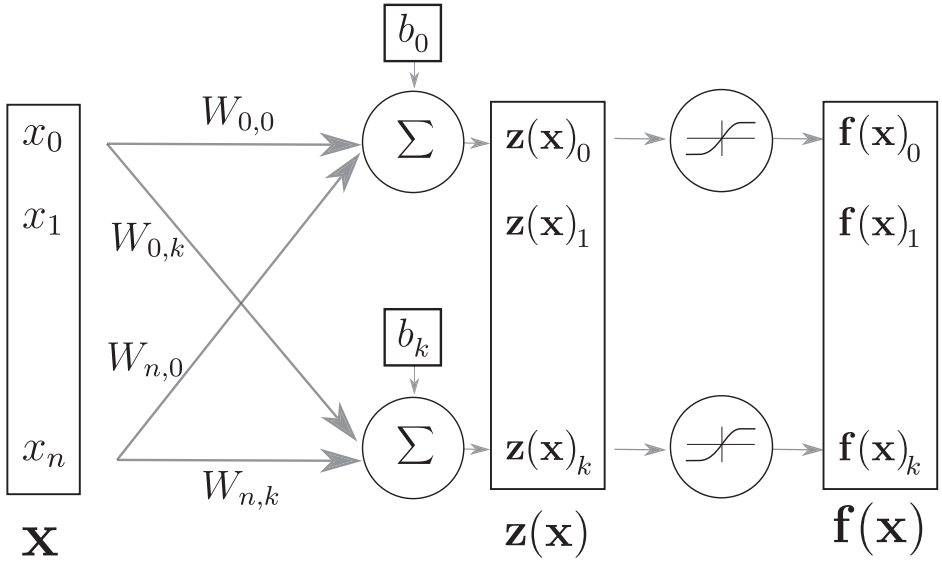
- $\mathbf{x}, f(\mathbf{x})$ input and output
- $z(\mathbf{x})$ pre-activation
- $\mathbf{w}, b$ weights and bias
- g activation function

Slide credit: C. Ollion & O. Grisel

More neurons -> more capacity

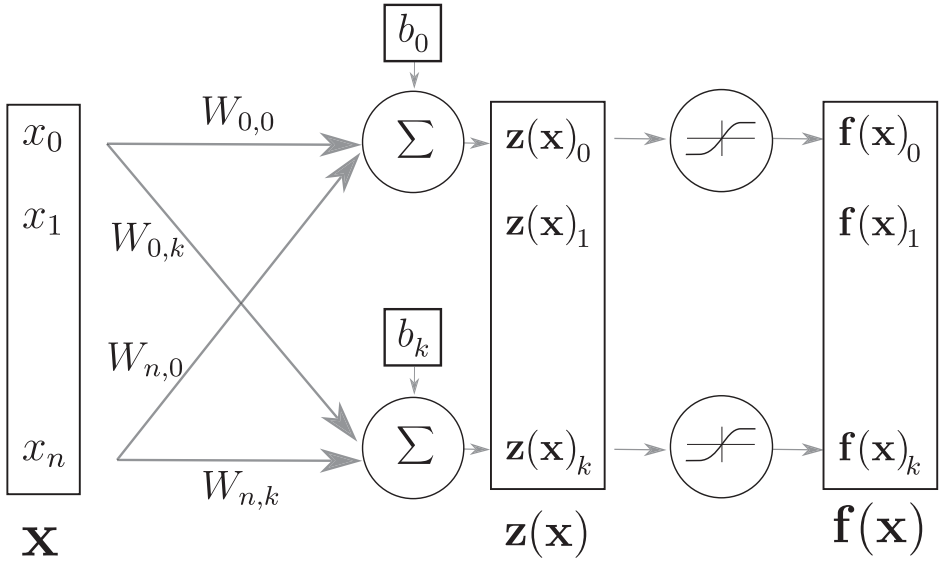


Layer of Neurons



Slide credit: C. Ollion & O. Grisel

Layer of Neurons

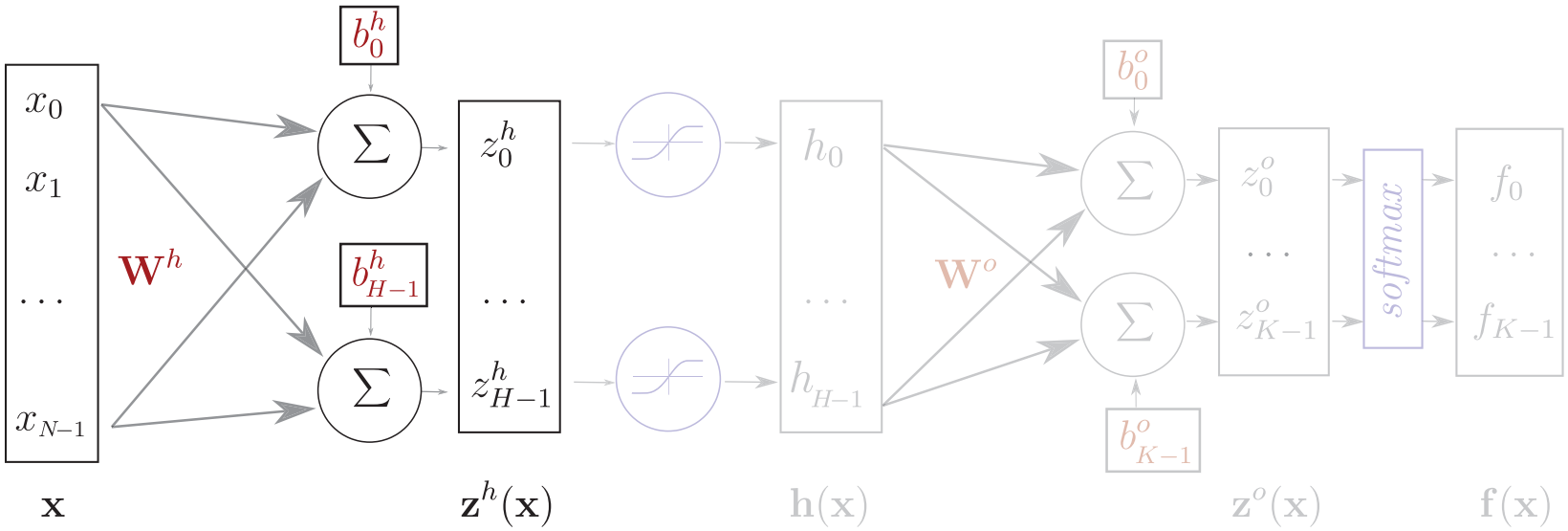


$$\mathbf{f}(\mathbf{x}) = g(\mathbf{z}(\mathbf{x})) = g(\mathbf{W}\mathbf{x} + \mathbf{b})$$

- $\mathbf{W}$, $\mathbf{b}$ now matrix and vector

Slide credit: C. Ollion & O. Grisel

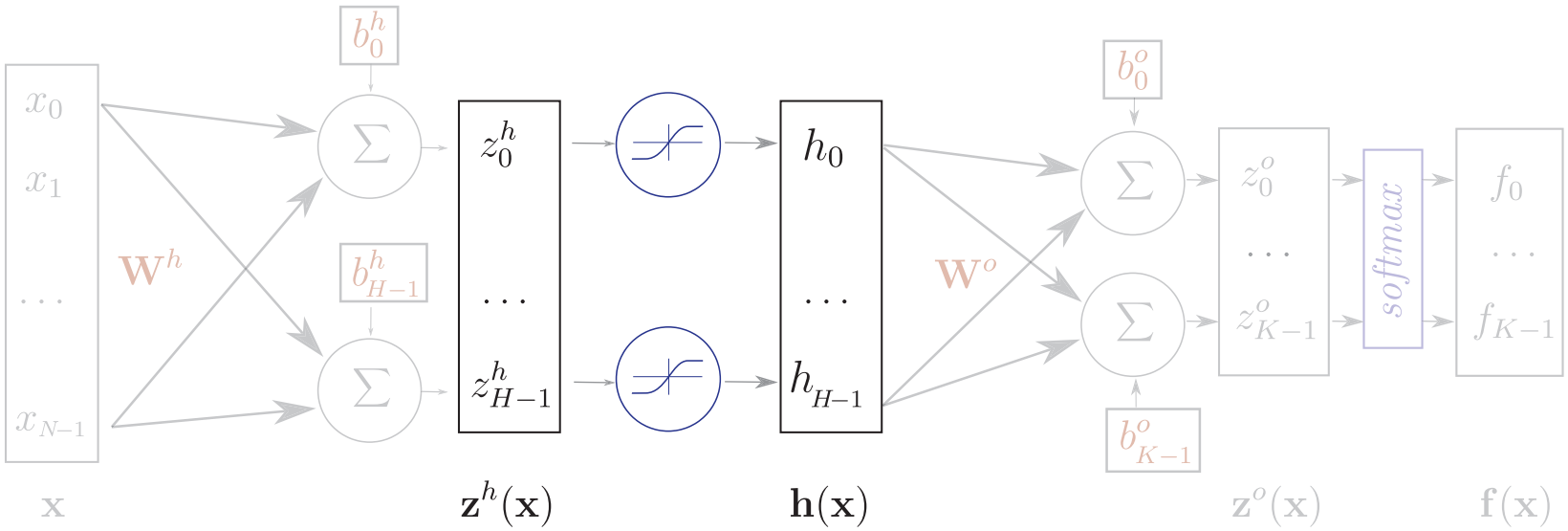
One Hidden Layer Network



- $\mathbf{z}^h(\mathbf{x}) = \mathbf{W}^h\mathbf{x} + \mathbf{b}^h$
- $\mathbf{h}(\mathbf{x}) = g(\mathbf{z}^h(\mathbf{x})) = g(\mathbf{W}^h\mathbf{x} + \mathbf{b}^h)$
- $\mathbf{z}^o(\mathbf{x}) = \mathbf{W}^o\mathbf{h}(\mathbf{x}) + \mathbf{b}^o$
- $\mathbf{f}(\mathbf{x}) = softmax(\mathbf{z}^o) = softmax(\mathbf{W}^o\mathbf{h}(\mathbf{x}) + \mathbf{b}^o)$

Slide credit: C. Ollion & O. Grisel

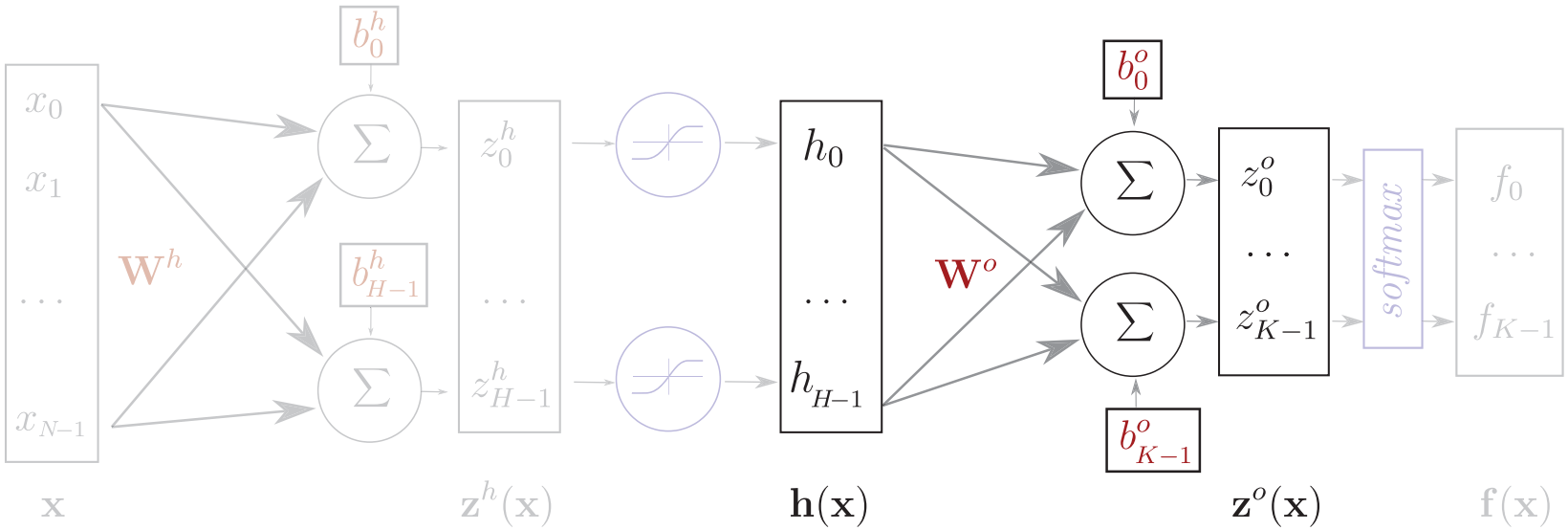
One Hidden Layer Network



- $\mathbf{z}^h(\mathbf{x}) = \mathbf{W}^h\mathbf{x} + \mathbf{b}^h$
- $\mathbf{h}(\mathbf{x}) = g(\mathbf{z}^h(\mathbf{x})) = g(\mathbf{W}^h\mathbf{x} + \mathbf{b}^h)$
- $\mathbf{z}^o(\mathbf{x}) = \mathbf{W}^o\mathbf{h}(\mathbf{x}) + \mathbf{b}^o$
- $\mathbf{f}(\mathbf{x}) = softmax(\mathbf{z}^o) = softmax(\mathbf{W}^o\mathbf{h}(\mathbf{x}) + \mathbf{b}^o)$

Slide credit: C. Ollion & O. Grisel

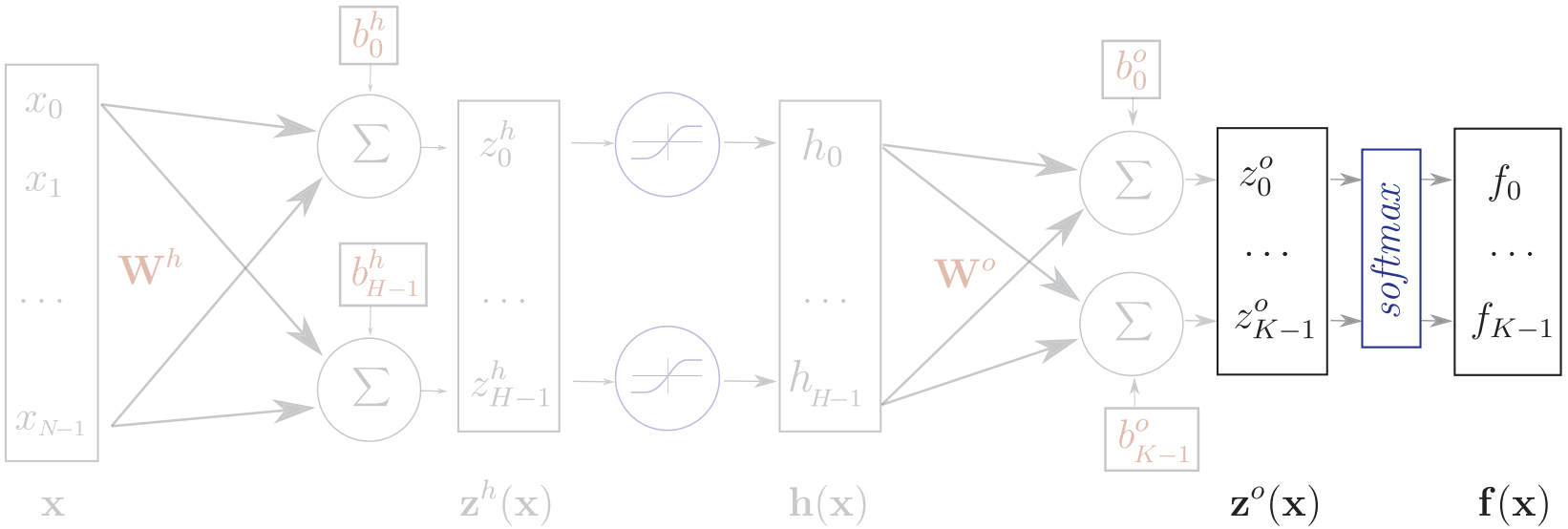
One Hidden Layer Network



- $\mathbf{z}^h(\mathbf{x}) = \mathbf{W}^h\mathbf{x} + \mathbf{b}^h$
- $\mathbf{h}(\mathbf{x}) = g(\mathbf{z}^h(\mathbf{x})) = g(\mathbf{W}^h\mathbf{x} + \mathbf{b}^h)$
- $\mathbf{z}^o(\mathbf{x}) = \mathbf{W}^o\mathbf{h}(\mathbf{x}) + \mathbf{b}^o$
- $\mathbf{f}(\mathbf{x}) = softmax(\mathbf{z}^o) = softmax(\mathbf{W}^o\mathbf{h}(\mathbf{x}) + \mathbf{b}^o)$

Slide credit: C. Ollion & O. Grisel

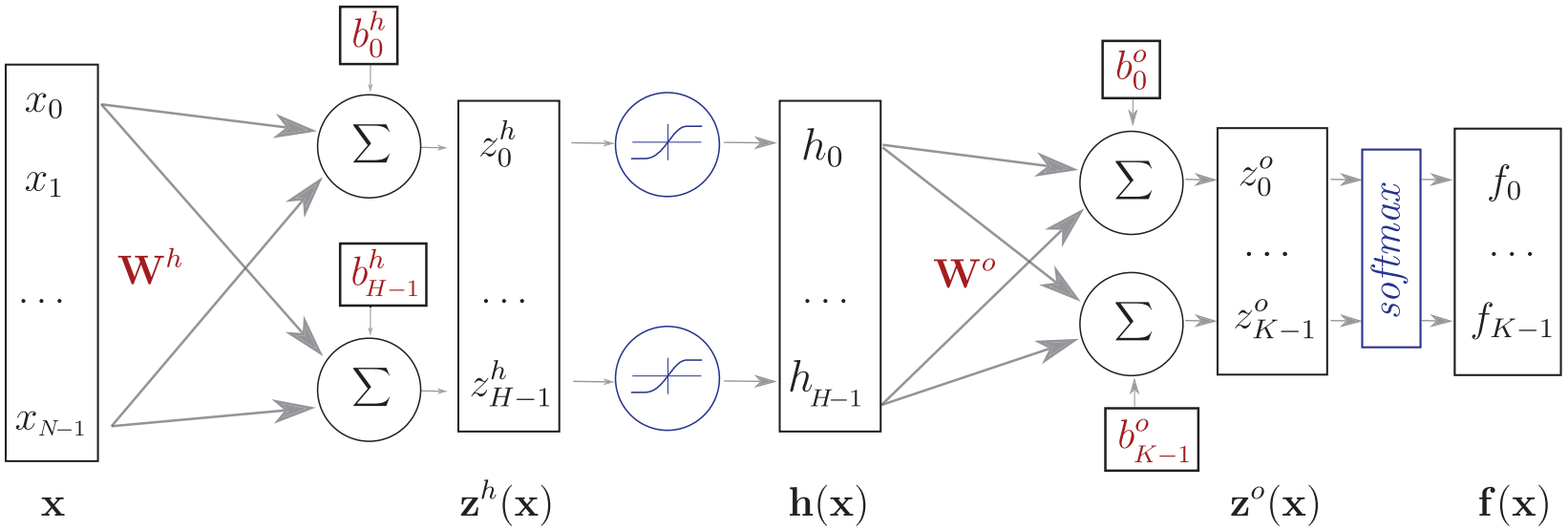
One Hidden Layer Network



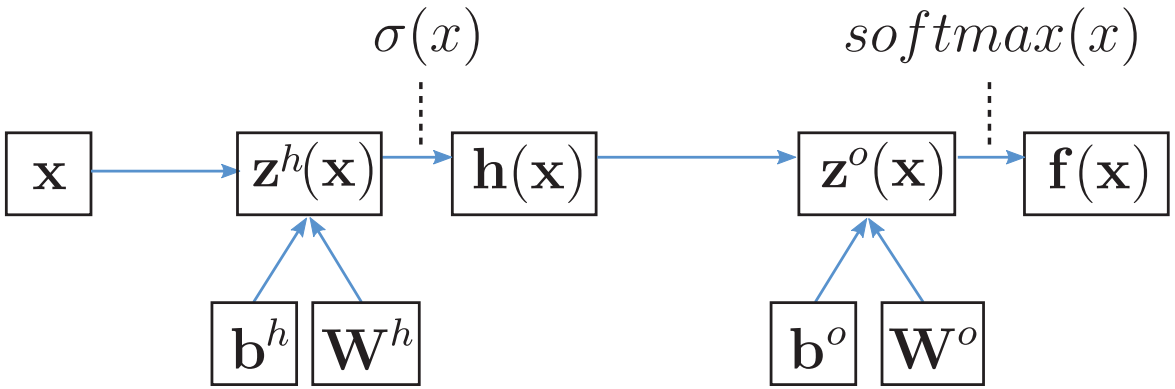
- $\mathbf{z}^h(\mathbf{x}) = \mathbf{W}^h\mathbf{x} + \mathbf{b}^h$
- $\mathbf{h}(\mathbf{x}) = g(\mathbf{z}^h(\mathbf{x})) = g(\mathbf{W}^h\mathbf{x} + \mathbf{b}^h)$
- $\mathbf{z}^o(\mathbf{x}) = \mathbf{W}^o\mathbf{h}(\mathbf{x}) + \mathbf{b}^o$
- $\mathbf{f}(\mathbf{x}) = softmax(\mathbf{z}^o) = softmax(\mathbf{W}^o\mathbf{h}(\mathbf{x}) + \mathbf{b}^o)$

Slide credit: C. Ollion & O. Grisel

One Hidden Layer Network

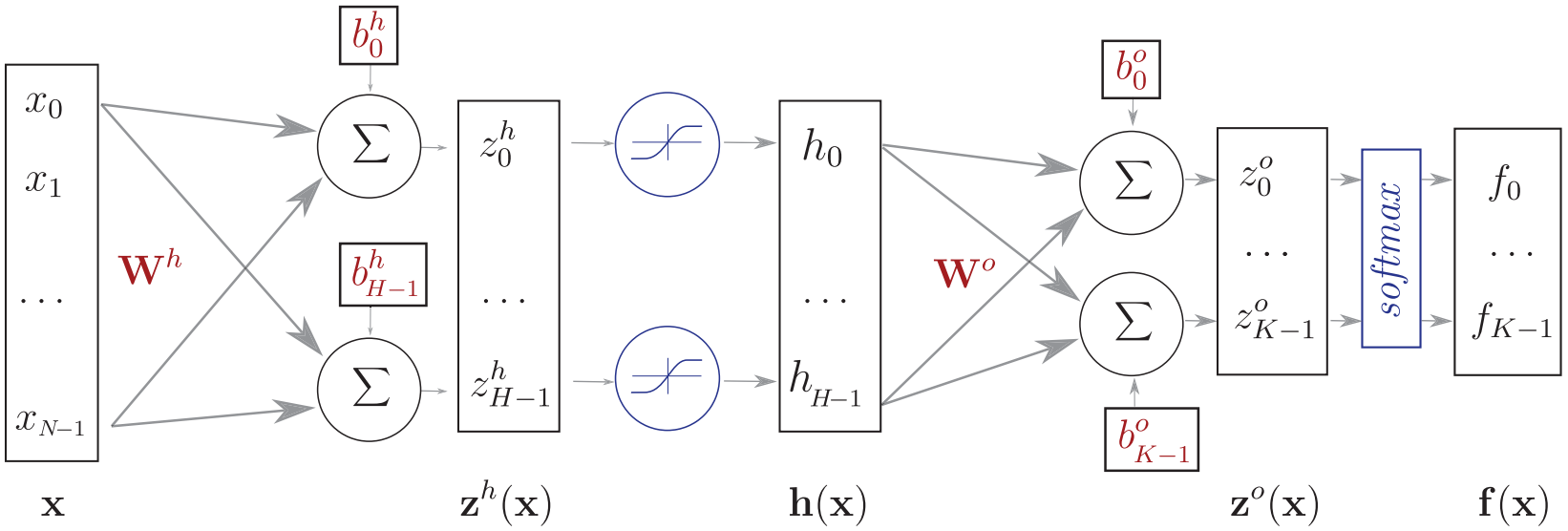


Alternate representation



Slide credit: C. Ollion & O. Grisel

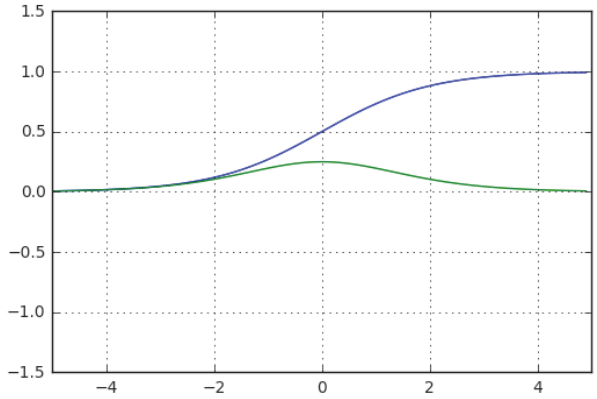
One Hidden Layer Network



PyTorch implementation

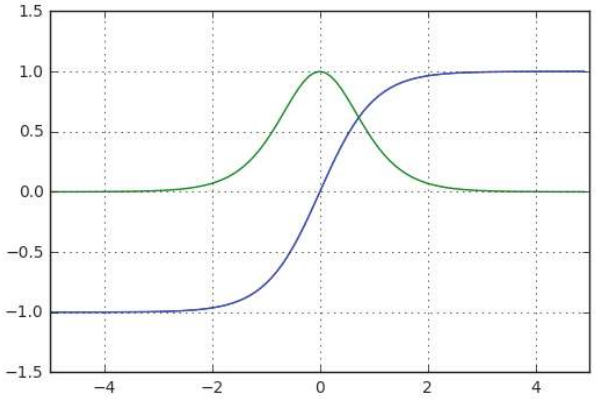
```
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),   # weight matrix dim [D_in x H]
    torch.nn.Tanh(),
    torch.nn.Linear(H, D_out),   # weight matrix dim [H x D_out]
    torch.nn.Softmax(),
)
```

Element-wise activation functions



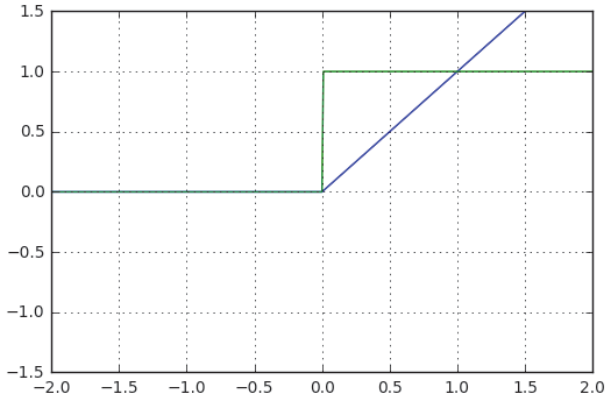
$$\text{sigm}(x) = \frac{1}{1+e^{-x}}$$

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



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

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



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

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

- blue: activation function
- green: derivative

Slide credit: C. Ollion & O. Grisel

Element-wise activation functions

- Many other activation functions available:

Softmax function

$$softmax(\mathbf{x}) = \frac{1}{\sum_{i=1}^{n} e^{x_i}} \cdot \begin{bmatrix} e^{x_1} \\ e^{x_2} \\ \vdots \\ e^{x_n} \end{bmatrix}$$

$$\frac{\partial softmax(\mathbf{x})_i}{\partial x_j} = \begin{cases} softmax(\mathbf{x})_i \cdot (1 - softmax(\mathbf{x})_i) & i = j \\ -softmax(\mathbf{x})_i \cdot softmax(\mathbf{x})_j & i \neq j \end{cases}$$

Softmax function

$$softmax(\mathbf{x}) = \frac{1}{\sum_{i=1}^{n} e^{x_i}} \cdot \begin{bmatrix} e^{x_1} \\ e^{x_2} \\ \vdots \\ e^{x_n} \end{bmatrix}$$

$$\frac{\partial softmax(\mathbf{x})_i}{\partial x_j} = \begin{cases} softmax(\mathbf{x})_i \cdot (1 - softmax(\mathbf{x})_i) & i = j \\ -softmax(\mathbf{x})_i \cdot softmax(\mathbf{x})_j & i \neq j \end{cases}$$

- vector of values in (0, 1) that add up to 1
- $p(Y = c \,|\, X = \mathbf{x}) = \text{softmax}(\mathbf{z}((\mathbf{x}))_c$
- the pre-activation vector $\mathbf{z}(\mathbf{x})$ is often called "the logits"

Slide credit: C. Ollion & O. Grisel

Training the network

Find parameters $\theta = (\mathbf{W}^h; \mathbf{b}^h; \mathbf{W}^o; \mathbf{b}^o)$ that minimize the negative log likelihood (or cross entropy)

Training the network

Find parameters $\theta = (\mathbf{W}^h; \mathbf{b}^h; \mathbf{W}^o; \mathbf{b}^o)$ that minimize the negative log likelihood (or cross entropy)

The loss function for a given sample $s \in S$:

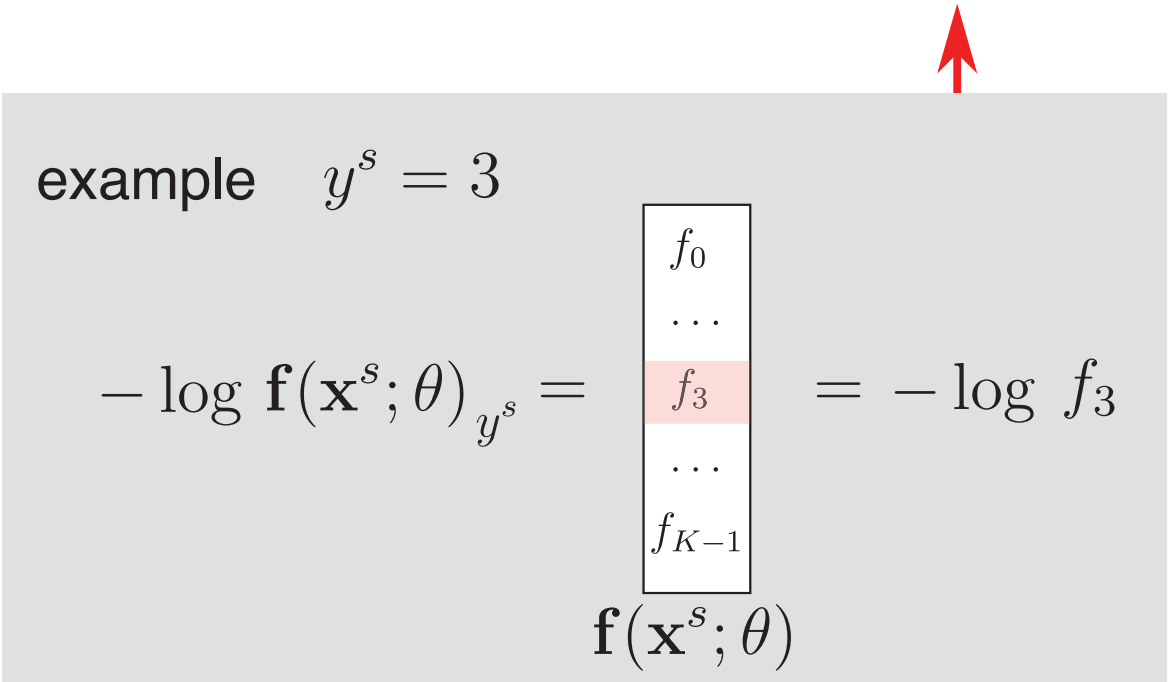
$$l(\mathbf{f}(\mathbf{x}^s; \theta), y^s) = nll(\theta; \mathbf{x}^s, y^s) = -\log \mathbf{f}(\mathbf{x}^s; \theta)_{y^s}$$

Training the network

Find parameters $\theta = (\mathbf{W}^h; \mathbf{b}^h; \mathbf{W}^o; \mathbf{b}^o)$ that minimize the negative log likelihood (or cross entropy)

The loss function for a given sample $s \in S$:

$$l(\mathbf{f}(\mathbf{x}^s; \theta), y^s) = nll(\theta; \mathbf{x}^s, y^s) = -\log \mathbf{f}(\mathbf{x}^s; \theta)_{y^s}$$



Training the network

Find parameters $\theta = (\mathbf{W}^h; \mathbf{b}^h; \mathbf{W}^o; \mathbf{b}^o)$ that minimize the negative log likelihood (or cross entropy)

The loss function for a given sample $s \in S$:

$$l(\mathbf{f}(\mathbf{x}^s; \theta), y^s) = nll(\theta; \mathbf{x}^s, y^s) = -\log \mathbf{f}(\mathbf{x}^s; \theta)_{y^s}$$

The cost function is the negative likelihood of the model computed on the full training set (for i.i.d. samples):

$$L_S(\theta) = -\frac{1}{|S|} \sum_{s \in S} \log \mathbf{f}(\mathbf{x}^s; \theta)_{y^s} + \lambda \Omega(\theta)$$

Training the network

Find parameters $\theta = (\mathbf{W}^h; \mathbf{b}^h; \mathbf{W}^o; \mathbf{b}^o)$ that minimize the negative log likelihood (or cross entropy)

The loss function for a given sample $s \in S$:

$$l(\mathbf{f}(\mathbf{x}^s; \theta), y^s) = nll(\theta; \mathbf{x}^s, y^s) = -\log \mathbf{f}(\mathbf{x}^s; \theta)_{y^s}$$

The cost function is the negative likelihood of the model computed on the full training set (for i.i.d. samples):

$$L_S(\theta) = -\frac{1}{|S|} \sum_{s \in S} \log \mathbf{f}(\mathbf{x}^s; \theta)_{y^s} + \lambda \Omega(\theta)$$

$\lambda \Omega(\theta) = \lambda(||W^h||^2 + ||W^o||^2)$ is an optional regularization term.

Stochastic Gradient Descent

Initialize θ randomly

Stochastic Gradient Descent

Initialize θ randomly

For E epochs perform:

- Randomly select a small batch of samples ($B \subset S$)

Stochastic Gradient Descent

Initialize θ randomly

For E epochs perform:

- Randomly select a small batch of samples ($B \subset S$)
- Compute gradients: $\Delta = \nabla_\theta L_B(\theta)$

Stochastic Gradient Descent

Initialize θ randomly

For E epochs perform:

- Randomly select a small batch of samples ($B \subset S$)
- Compute gradients: $\Delta = \nabla_\theta L_B(\theta)$
- Update parameters: $\theta \leftarrow \theta - \eta\Delta$
- $\eta > 0$ is called the learning rate

Stochastic Gradient Descent

Initialize θ randomly

For E epochs perform:

- Randomly select a small batch of samples ($B \subset S$)
- Compute gradients: $\Delta = \nabla_\theta L_B(\theta)$
- Update parameters: $\theta \leftarrow \theta - \eta\Delta$
- $\eta > 0$ is called the learning rate

Stop when reaching criterion

- nll stops decreasing when computed on validation set

Loss functions

Discrete output (classification)

- Binary classification: $y \in [0, 1]$
  - $Y | X = \mathbf{x} \sim Bernoulli(b = f(\mathbf{x}; \theta))$
  - output function: $logistic(x) = \frac{1}{1 + e^{-x}}$
  - loss function: binary cross-entropy
- Multiclass classification: $y \in [0, K - 1]$
  - $Y | X = \mathbf{x} \sim Multinoulli(\mathbf{p} = \mathbf{f}(\mathbf{x}; \theta))$
  - output function: $softmax$
  - loss function: categorical cross-entropy

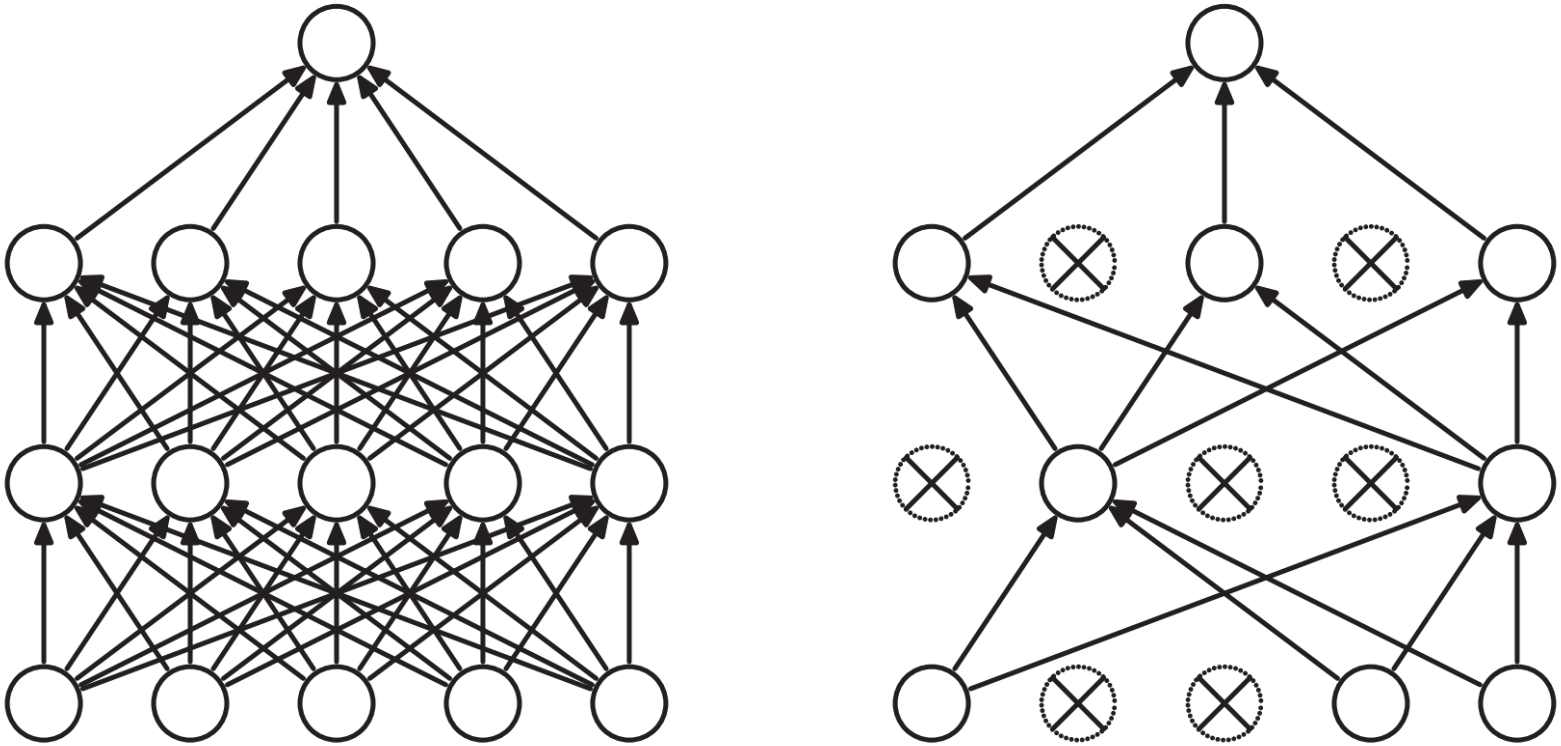
Continuous output (regression)

- Continuous output: $\mathbf{y} \in \mathrm{R}^n$
  - $Y \mid X = \mathbf{x} \sim \mathrm{N}(\mu = \mathbf{f}(\mathbf{x}; \theta), \sigma^2 \mathbf{I})$
  - output function: Identity
  - loss function: square loss
- Heteroschedastic if $\mathbf{f}(\mathbf{x}; \theta)$ predicts both μ and σ^2
- Mixture Density Network (multimodal output)
  - $Y \mid X = \mathbf{x} \sim GMM_{\mathbf{x}}$
  - $\mathbf{f}(\mathbf{x}; \theta)$ predicts all the parameters: the means, covariance matrices and mixture weights

Going deeper

Dropout

- First "deep" regularization technique
- Remove units at random during the forward pass on each sample
- Put them all back during test



(a) Standard Neural Net (b) After applying dropout.

Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Srivastava et al., JMLR 2014

Dropout

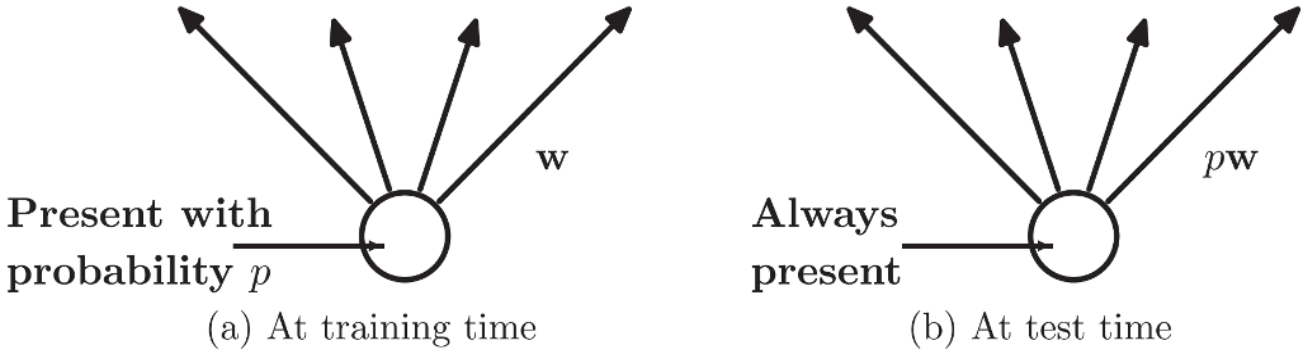
Interpretation

- Reduces the network dependency to individual neurons and distributes representation
- More redundant representation of data

Ensemble interpretation

- Equivalent to training a large ensemble of shared-parameters, binary-masked models
- Each model is only trained on a single data point
- A network with dropout can be interpreted as an ensemble of 2^N models with heavy weight sharing (Goodfellow et al., 2013)

Dropout

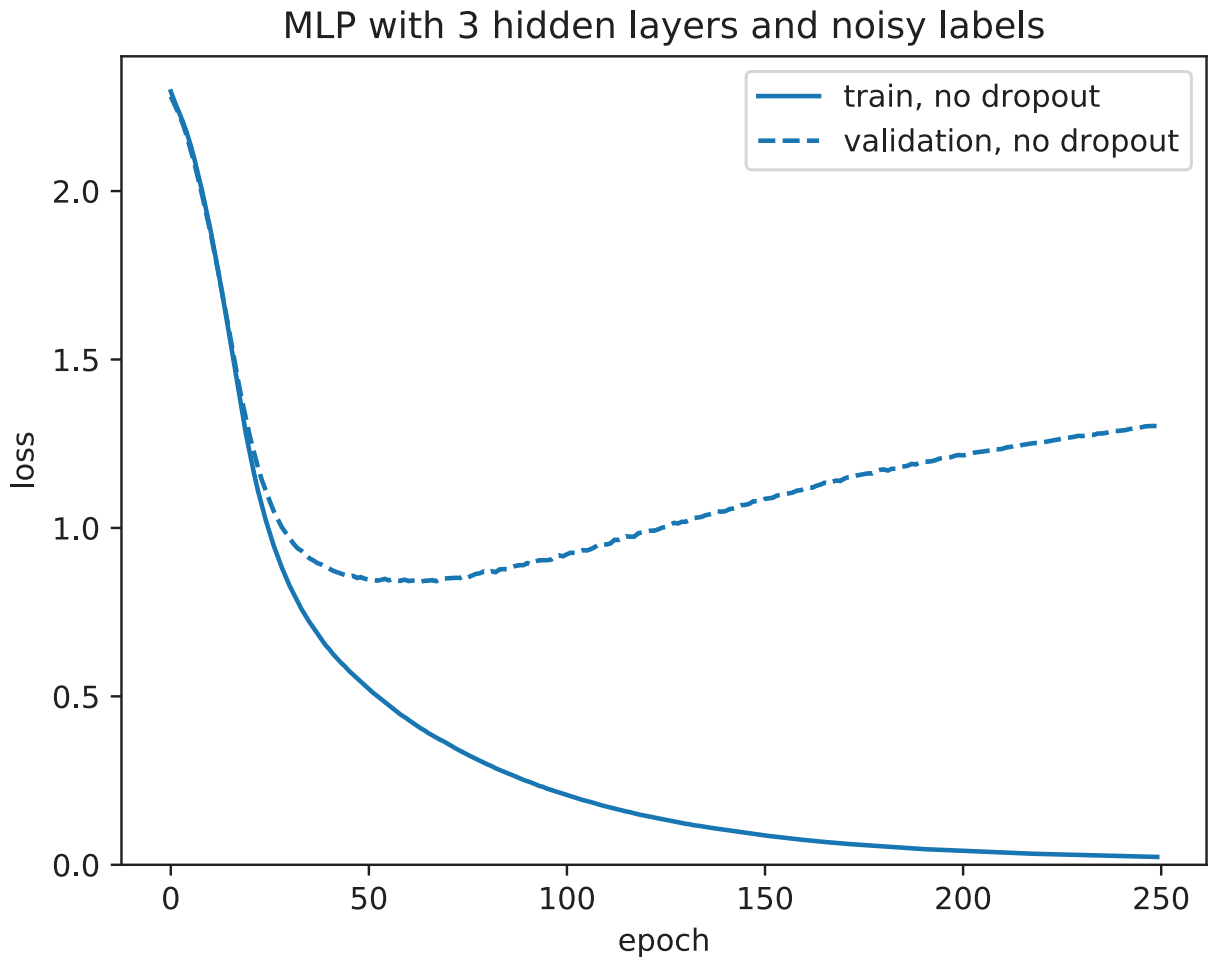


(a) At training time (b) At test time

- One has to decide on which units/layers to use dropout, and with what probability p units are dropped.
- During training, for each sample, as many Bernoulli variables as units are sampled independently to select units to remove.
- To keep the means of the inputs to layers unchanged, the initial version of dropout was multiplying activations by p during test.
- The standard variant is the "inverted dropout": multiply activations by $\frac{1}{1-p}$ during training and keep the network untouched during test.

Dropout

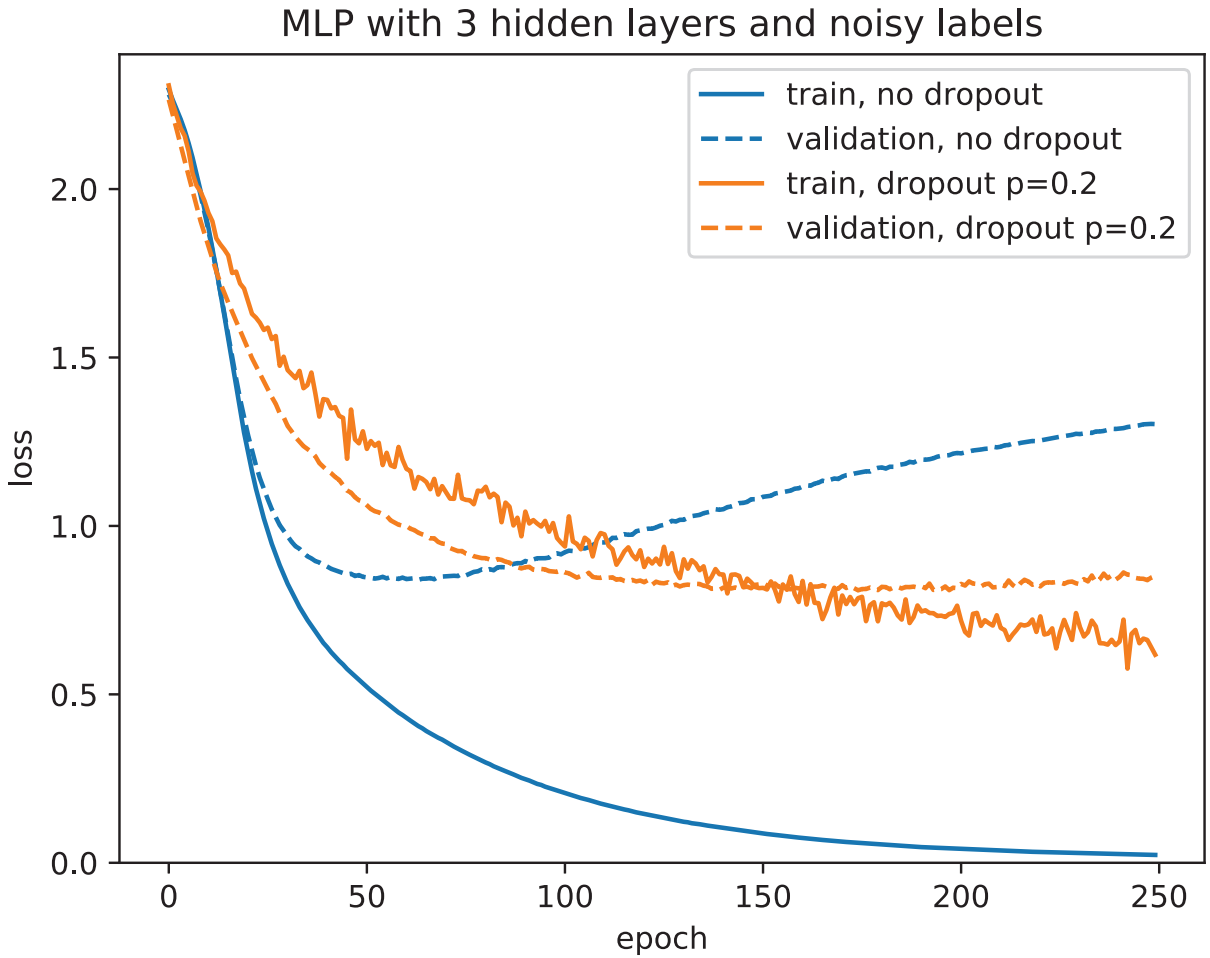
Overfitting noise



Slide credit: C. Ollion & O. Grisel

Dropout

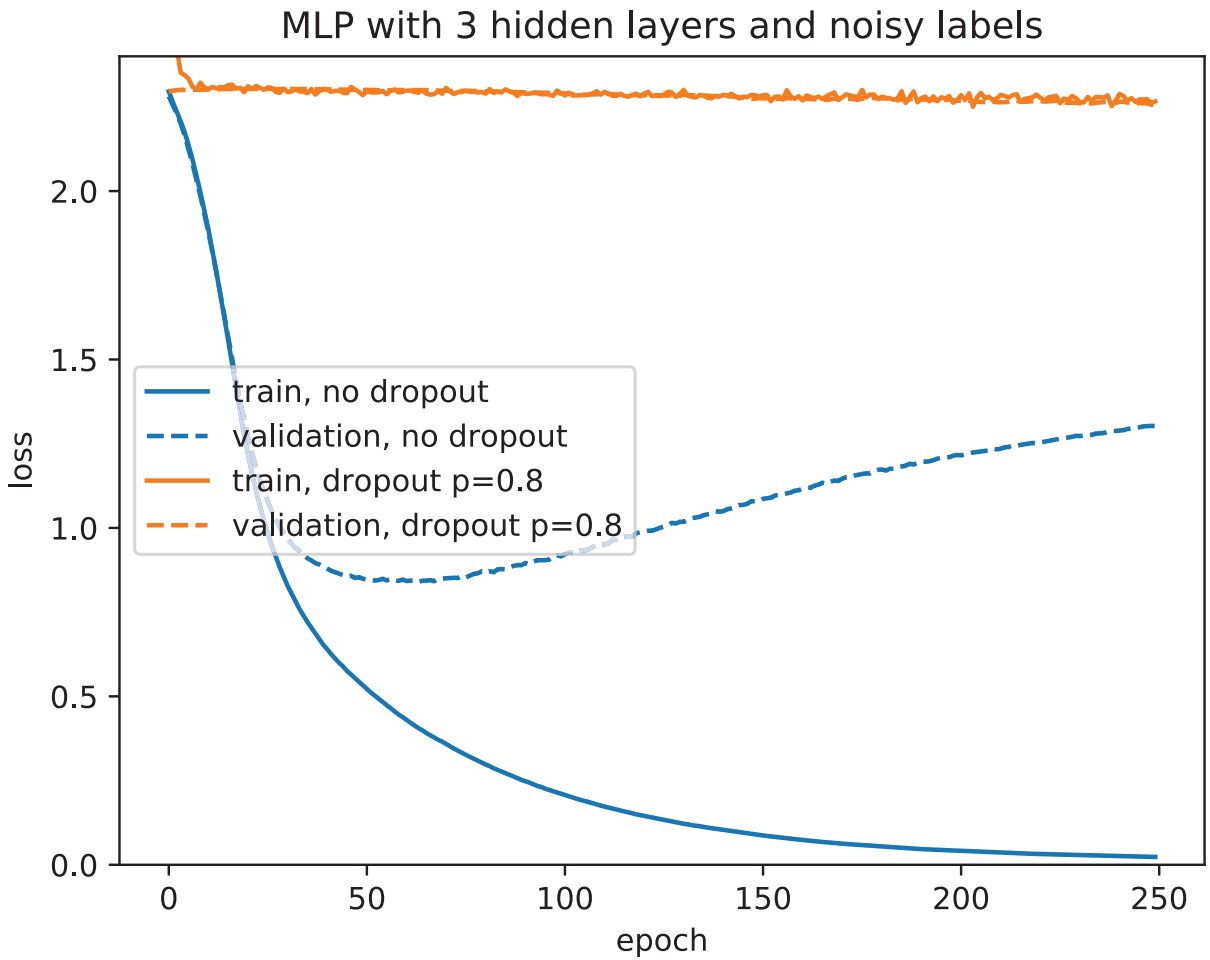
A bit of Dropout



Slide credit: C. Ollion & O. Grisel

Dropout

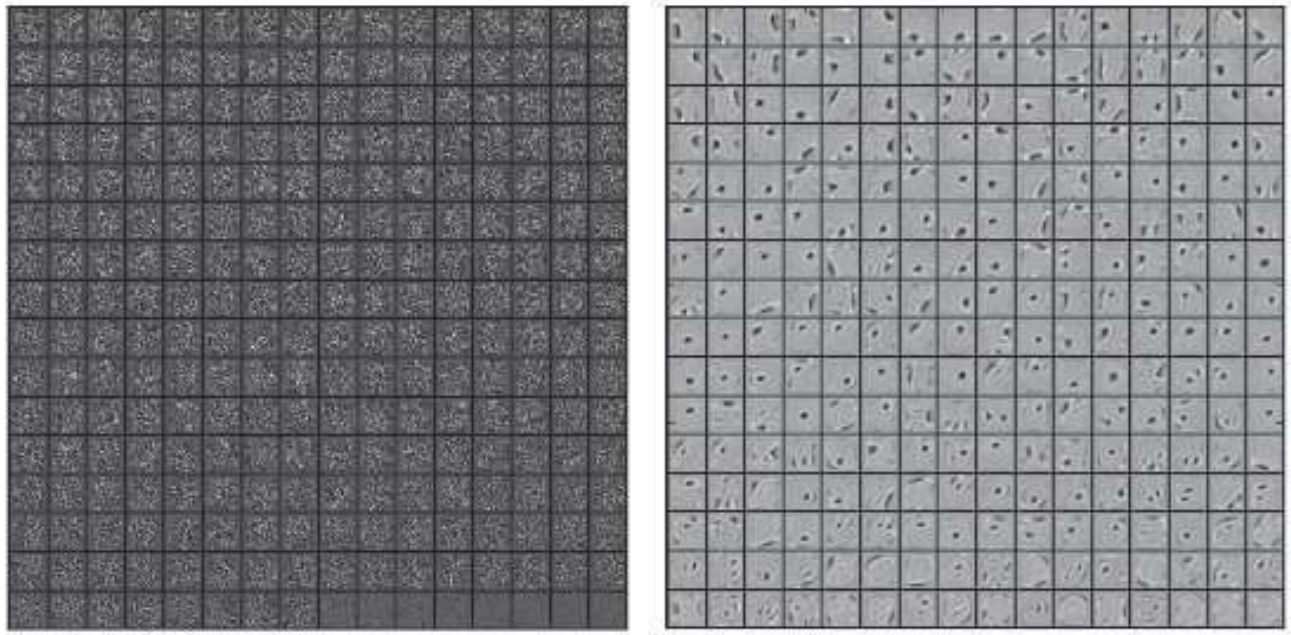
Too much: underfitting



Slide credit: C. Ollion & O. Grisel

Dropout

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



(a) Without dropout

(b) Dropout with $p = 0.5$.

Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Srivastava et al., JMLR 2014

Dropout

```
>>> x = Variable(torch.Tensor(3, 9).fill_ (1.0), requires_grad = True)
>>> x.data
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
[torch.FloatTensor of size 3x9]

>>> dropout = nn.Dropout(p = 0.75)
>>> y = dropout(x)
>>> y.data
4 0 4 4 4 0 4 0 0
4 0 0 0 0 0 0 0 0
0 0 0 0 4 0 4 0 4
[torch.FloatTensor of size 3x9]

>>> l = y.norm(2, 1).sum()
>>> l.backward()
>>> x.grad.data
1.7889 0.0000 1.7889 1.7889 0.0000 0.0000 1.7889 0.0000 0.0000
4.0000 0.0000 0.0000 1.7889 0.0000 0.0000 0.0000 2.3094 0.0000
0.0000 0.0000 0.0000 0.0000 2.3094 0.0000 0.0000 0.0000 2.3094
[torch.FloatTensor of size 3x9]
```

Dropout

For a given network

```
model = nn.Sequential(nn.Linear(10, 100), nn.ReLU(),
                      nn.Linear(100, 50), nn.ReLU(),
                      nn.Linear(50, 2));
```

Dropout

For a given network

```
model = nn.Sequential(nn.Linear(10, 100), nn.ReLU(),
                      nn.Linear(100, 50), nn.ReLU(),
                      nn.Linear(50, 2));
```

we can simply add dropout layers

```
model = nn.Sequential(nn.Linear(10, 100), nn.ReLU(),
                      nn.Dropout(),
                      nn.Linear(100, 50), nn.ReLU(),
                      nn.Dropout(),
                      nn.Linear(50, 2));
```

Dropout

A model using dropout has to be set in "train" or "test" mode

Dropout

A model using dropout has to be set in "train" or "test" mode

The method `nn.Module.train(mode)` recursively sets the flag `training` to all sub-modules.

```
>>> dropout = nn.Dropout()
>>> model = nn.Sequential(nn.Linear(3, 10), dropout, nn.Linear(10, 3))
>>> dropout.training
True
>>> model.train(False)
Sequential (
(0): Linear (3 -> 10) (1): Dropout (p = 0.5) (2): Linear (10 -> 3)
)
>>> dropout.training
False
```

Spatial Dropout

As pointed out by Tompson et al. (2015), units in a 2d activation map are generally locally correlated, and dropout has virtually no effect.

They proposed SpatialDropout, which drops channels instead of individual units.

Slide credit: F. Fleuret

Spatial Dropout

```
>>> dropout2d = nn.Dropout2d()
>>> x = Variable(Tensor(2, 3, 2, 2).fill_(1.0))
>>> dropout2d(x)
Variable containing:
(0 ,0 ,.,.) =
0 0
0 0

(0 ,1 ,.,.) =
0 0
0 0

(0 ,2 ,.,.) =
2 2
2 2

(1 ,0 ,.,.) =
2 2
2 2

(1 ,1 ,.,.) =
0 0
0 0

(1 ,2 ,.,.) =
2 2
2 2
[torch.FloatTensor of size 2x3x2x2]
```

Batch normalization

We saw that maintaining proper statistics of the activations and derivatives was a critical issue to allow the training of deep architectures.

It is the main motivation behind weight initialization rules (we'll cover them later).

Batch normalization

We saw that maintaining proper statistics of the activations and derivatives was a critical issue to allow the training of deep architectures.

It is the main motivation behind weight initialization rules (we'll cover them later).

A different approach consists of explicitly forcing the activation statistics during the forward pass by re-normalizing them.

Batch normalization proposed by Ioffe and Szegedy (2015) was the first method introducing this idea.

Batch normalization

Normalize activations in each mini-batch before activation function: speeds up and stabilizes training (less dependent on init)

Batch normalization forces the activation first and second order moments, so that the following layers do not need to adapt to their drift.

Batch normalization

Normalize activations in each mini-batch before activation function: speeds up and stabilizes training (less dependent on init)

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

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

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

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

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

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

Batch normalization: Accelerating deep network training by reducing internal covariate shift, Ioffe and Szegedy, ICML 2015

Batch normalization

During training batch normalization shifts and rescales according to the mean and variance estimated on the batch.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

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

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

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

As for dropout, the model behaves differently during train and test.

Batch normalization

At inference time, use average and standard deviation computed on the whole dataset instead of batch

Widely used in ConvNets, but requires the mini-batch to be large enough to compute statistics in the minibatch.

Batch normalization

As dropout, batch normalization is implemented as a separate module `torch.BatchNorm1d` that processes the input components separately.

```
>>> x = torch.Tensor(10000, 3).normal_()
>>> x = x * torch.Tensor([2, 5, 10]) + torch.Tensor([-10, 25, 3])
>>> x = Variable(x)
>>> x.data.mean(0)
-9.9898
24.9165
2.8945
[torch.FloatTensor of size 3]

>>> x.data.std(0)
2.0006
5.0146
9.9501
[torch.FloatTensor of size 3]
```

Batch normalization

Since the module has internal variables to keep statistics, it must be provided with the sample dimension at creation.

```
>>> bn = nn.BatchNorm1d(3)
>>> bn.bias.data = torch.Tensor([2, 4, 8])
>>> bn.weight.data = torch.Tensor([1, 2, 3])
>>> y = bn(x)
>>> y.data.mean(0)

2.0000
4.0000
8.0000
[torch.FloatTensor of size 3]
>>> y.data.std(0)

1.0000
2.0001
3.0001
[torch.FloatTensor of size 3]
```

Batch normalization

`BatchNorm2d` example

```
>>> x = Variable(torch.randn(20, 100, 35, 45))
>>> bn2d = nn.BatchNorm2d(100)
>>> y = bn2d(x)
>>> x.size()

torch.Size([20, 100, 35, 45])
>>> bn2d.weight.data.size()

torch.Size([100])
>>> bn2d.bias.data.size()

torch.Size([100])
```

Batch normalization

Results on ImageNet LSVRC 2012:

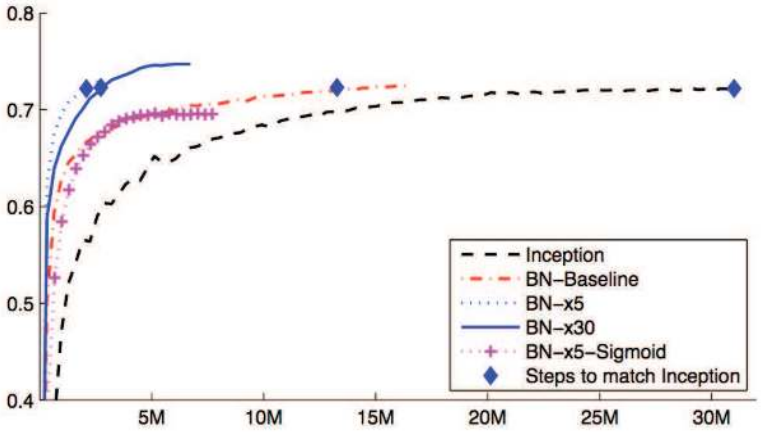


Figure 2: *Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.*

Model	Steps to 72.2%	Max accuracy
Inception	$31.0 \cdot 10^6$	72.2%
BN-Baseline	$13.3 \cdot 10^6$	72.7%
BN-x5	$2.1 \cdot 10^6$	73.0%
BN-x30	$2.7 \cdot 10^6$	74.8%
BN-x5-Sigmoid		69.8%

Figure 3: *For Inception and the batch-normalized variants, the number of training steps required to reach the maximum accuracy of Inception (72.2%), and the maximum accuracy achieved by the network.*

Batch normalization: Accelerating deep network training by reducing internal covariate shift, Ioffe and Szegedy, ICML 2015

Batch normalization

Results on ImageNet LSVRC 2012:

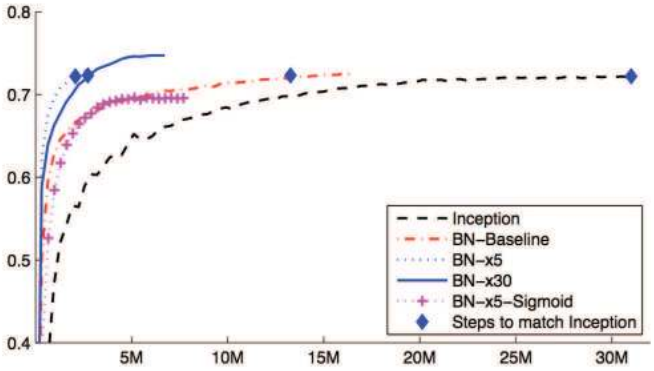


Figure 2: *Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.*

Model	Steps to 72.2%	Max accuracy
Inception	$31.0 \cdot 10^6$	72.2%
BN-Baseline	$13.3 \cdot 10^6$	72.7%
BN-x5	$2.1 \cdot 10^6$	73.0%
BN-x30	$2.7 \cdot 10^6$	74.8%
BN-x5-Sigmoid		69.8%

Figure 3: *For Inception and the batch-normalized variants, the number of training steps required to reach the maximum accuracy of Inception (72.2%), and the maximum accuracy achieved by the network.*

- learning rate can be greater
- dropout and local normalization are not necessary
- L^2 regularization influence should be reduced

Batch normalization: Accelerating deep network training by reducing internal covariate shift, Ioffe and Szegedy, ICML 2015

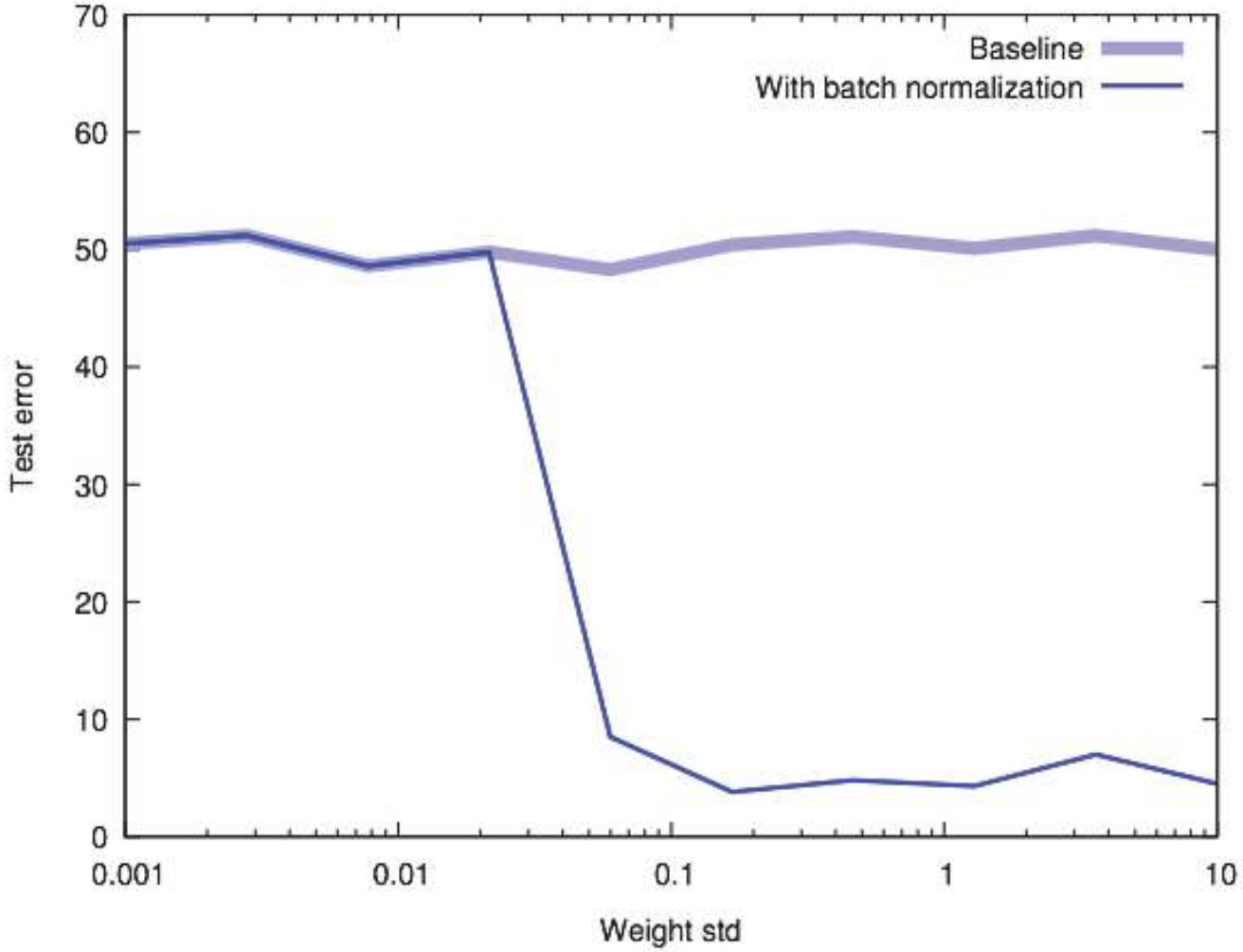
Batch normalization

Deep MLP on a 2d "disc" toy example, with naive Gaussian weight initialization, cross-entropy, standard SGD, $\eta = 0.1$.

```
def create_model(with_batchnorm, nc = 32, depth = 16):
    modules = []

    modules.append(nn.Linear(2, nc))
    if with_batchnorm: modules.append(nn.BatchNorm1d(nc))
    modules.append(nn.ReLU())

    for d in range(depth):
        modules.append(nn.Linear(nc, nc))
        if with_batchnorm: modules.append(nn.BatchNorm1d(nc))
        modules.append(nn.ReLU())

    modules.append(nn.Linear(nc, 2))

    return nn.Sequential(*modules)
```

Slide credit: F. Fleuret

Batch normalization



Slide credit: F. Fleuret