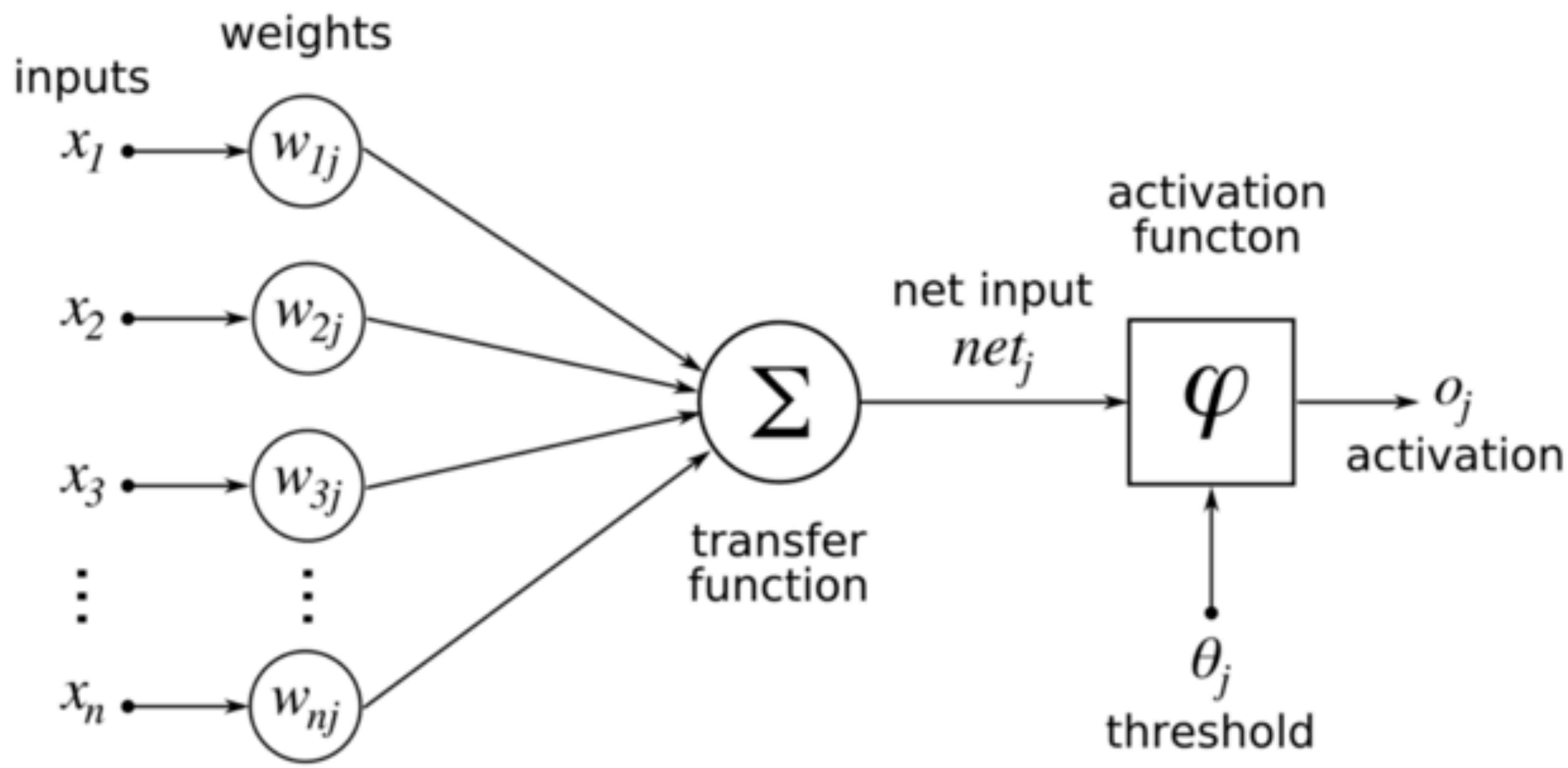
Machine Learning: Deep Learning

Yohann DE CASTRO and Aurélien GARIVIER

A Layer: the cornerstone of Deep Learning



$$f(\mathbf{x}) = \phi(\langle \mathbf{w}, \mathbf{x} \rangle - \theta)$$

Activation functions

- Linear

$$\phi(x) = x$$

- Rectified linear

$$\phi(x) = \max(0, x)$$

- Sigmoid

$$\phi(x) = \frac{1}{1 + e^{-\gamma x}}$$

- Hyperbolic tangent

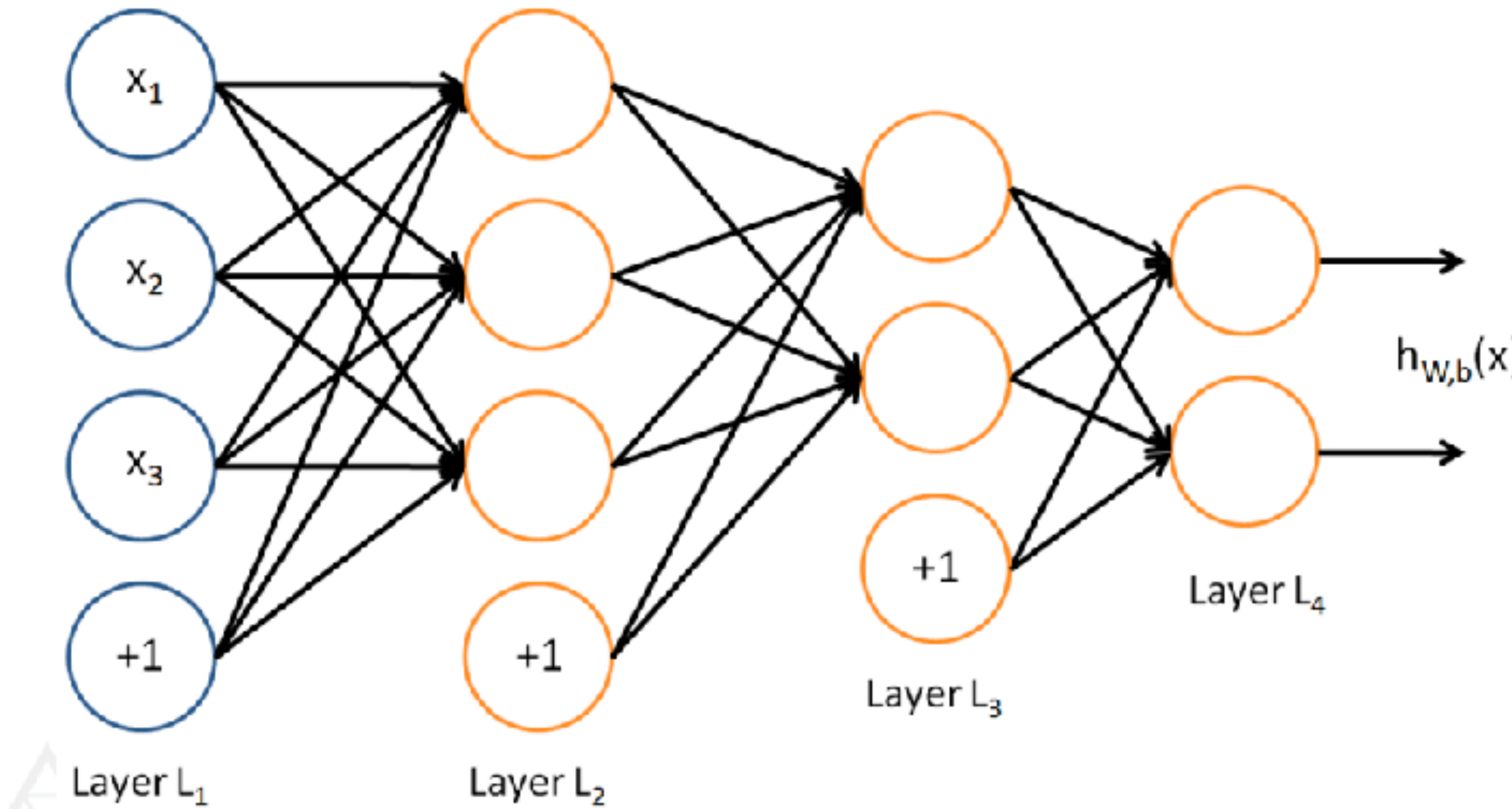
$$\phi(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Stochastic Gradient Descent

Given a loss function $l(y, f(\mathbf{x}))$, a training set $\mathcal{A} = \{(\mathbf{x}, y)\}$

1. Draw a random sample $(\mathbf{x}_i, y_i)$
2. Compute gradient $\delta_i = \frac{\partial l(y_i, f(\mathbf{x}_i))}{\partial \mathbf{w}}$
3. Apply gradient $\mathbf{w} \leftarrow \mathbf{w} - \eta \delta_i$

Multiple layer networks



Multiple layer networks

Set layer i as the function:

$$f_i(\mathbf{x}) = [\sigma(\mathbf{w}_{ij}^\top \mathbf{x} + \theta_{ij})]_j$$

with

- $\mathbf{w}_{ij}, \theta_{ij}$ the weights and bias of neuron j
- σ the activation function

Create a network by composing L layers:

$$F(\mathbf{x}) = f_L \circ \cdots \circ f_1(\mathbf{x})$$

Training procedure

ERM principle

$$\min_{\{\mathbf{w}_i\}_i} \mathbb{E}_{(\mathbf{x},y)} \left[l(y, F(\mathbf{x})) \right]$$

Stochastic gradient descent

$$\forall i, \mathbf{w}_i \leftarrow \mathbf{w}_i - \eta \mathbb{E}_{(\mathbf{x},y)} \left[\frac{\partial l(y, F(\mathbf{x}))}{\partial \mathbf{w}_i} \right]$$

Monte-Carlo estimation with mini-batch strategy

$$\mathbb{E}_{(\mathbf{x},y)} \left[\frac{\partial l(y, F(\mathbf{x}))}{\partial \mathbf{w}_i} \right] \approx \frac{1}{N} \sum_n \frac{\partial l(y_n, F(\mathbf{x}_n))}{\partial \mathbf{w}_i}$$

Backpropagation

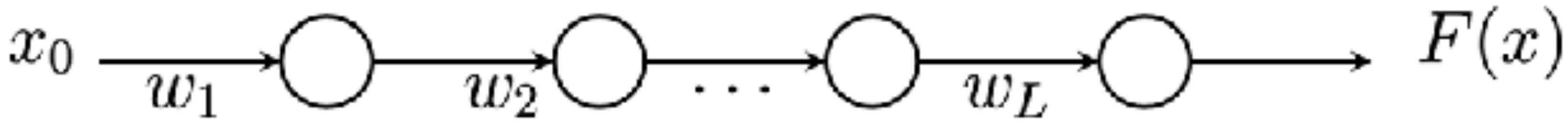
- Denote $\mathbf{x}_k = f_k \circ \cdots \circ f_1(\mathbf{x})$ the k-th intermediate output
- Denote $g_k(\mathbf{x}_k) = f_L \circ \cdots f_{k+1}(\mathbf{x}_k)$ the output computed from $\mathbf{x}_k$
- Remark $\forall k, F(\mathbf{x}) = g_k(\sigma(\mathbf{w}_k^\top \mathbf{x}_{k-1} + \theta_k))$

Chain rule (Leibnitz notation)

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial z}\frac{\partial z}{\partial x}$$

Backpropagation

Single neuron chain



$$
\begin{aligned}
\frac{\partial l(y, F(\mathbf{x}))}{\partial w_k} &= \frac{\partial l(y, F(\mathbf{x}))}{\partial F(\mathbf{x})} \frac{\partial F(\mathbf{x})}{\partial w_k} \\
&= l'(y, F(\mathbf{x})) \frac{\partial \sigma(w_L \mathbf{x}_{L-1} + \theta_L)}{\partial w_k} \\
&= l'(y, F(\mathbf{x})) \sigma'(\mathbf{x}_L) \frac{\partial w_L \mathbf{x}_{L-1} + \theta_L}{\partial w_k} \\
&= l'(y, F(\mathbf{x})) \sigma'(\mathbf{x}_L) w_L \frac{\partial \mathbf{x}_{L-1}}{\partial w_k}
\end{aligned}
$$

Backpropagation

$$
\begin{aligned}
\frac{\partial \mathbf{x}_{L-1}}{\partial w_k} &= \frac{\partial \sigma(w_{L-1}\mathbf{x}_{L-2} + \theta_{L-1})}{\partial w_k} \\
&= \sigma'(\mathbf{x}_{L-1}) w_{L-1} \frac{\partial \mathbf{x}_{L-2}}{\partial w_k} \\
\frac{\partial \mathbf{x}_{k+t+1}}{\partial w_k} &= \sigma'(\mathbf{x}_{k+t+1}) w_{k+t+1} \frac{\partial \mathbf{x}_{k+t}}{\partial w_k} \\
\frac{\partial \mathbf{x}_{k+1}}{\partial w_k} &= \mathbf{x}_k \\
\frac{\partial l(y, F(\mathbf{x}))}{\partial w_k} &= l'(y, F(\mathbf{x})) \prod_{t=k+1}^{L} \sigma'(\mathbf{x}_t) w_t \mathbf{x_k}
\end{aligned}
$$

Backpropagation

$$\frac{\partial \mathbf{x}_{k+t,j}}{\partial \mathbf{w}_k} = \sum_i \sigma(\mathbf{x}_{k+t+1,i}) \mathbf{w}_{k+t+1,i} \frac{\partial \mathbf{x}_{k+t+1,i}}{\partial \mathbf{w}_k}$$

$$\frac{\partial \mathbf{x}_k}{\partial \mathbf{w}_k} = \sigma'(\mathbf{x}_k)\mathbf{x}_{k-1}$$

Recursion

$$\delta_{L,i} = l'(y, F(\mathbf{x}))\sigma'(\mathbf{x}_L)$$

$$\delta_k = \mathbf{w}_{k+1}(\sigma'(\mathbf{x}_{k+1}) \circ \delta_{k+1})$$

$$\frac{\partial l(y, F(\mathbf{x}))}{\partial \mathbf{w}_k} = \delta_k \circ \mathbf{x}_k$$

Algorithm

Forward pass

- Compute and store $\forall k, \mathbf{x}_k$

Backward pass

- Compute $l'(y, g_{k+1}(\mathbf{x}_{k+1}))$
- $\forall k$, compute δ_k
- Update $\mathbf{w}_k$ using $l'(y, g_{k+1}(\mathbf{x}_{k+1}))$, δ_k and $\mathbf{x}_k$

In practice, machine learning libraries (tensorflow, pytorch, ...) have auto-grad features ($F(\mathbf{x})$ is built from operators with known derivative)

Convolutional neural networks

Discrete convolution

$$h(\mathbf{x})(t) = \sum_u \mathbf{x}(t+u)\mathbf{w}(u)$$

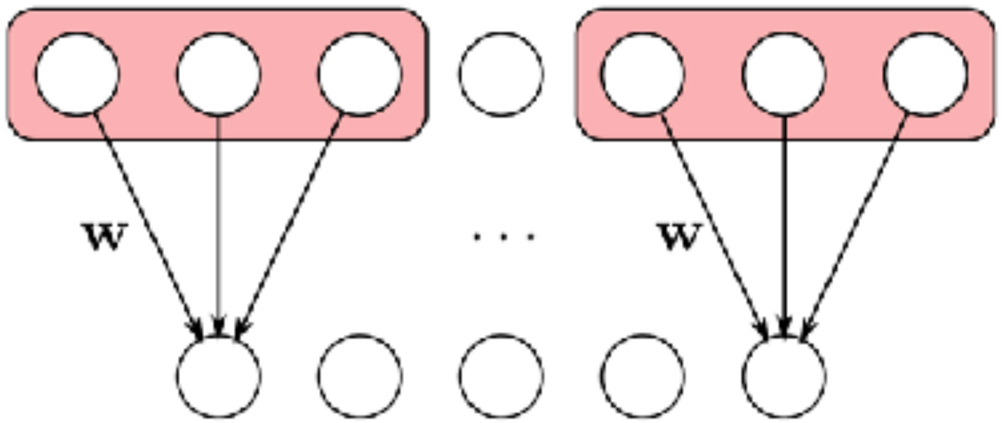
Extension to vector valued signals

$$h(\mathbf{x})(t) = \sum_u \langle \mathbf{x}(t+u), \mathbf{w}(u) \rangle$$

Convolutional neuron

$$f(\mathbf{x}) = \left[\sigma \left(\sum_u \langle \mathbf{x}(t+u), \mathbf{w}(u) \rangle + \theta \right) \right]_t$$

Convolutional neural networks

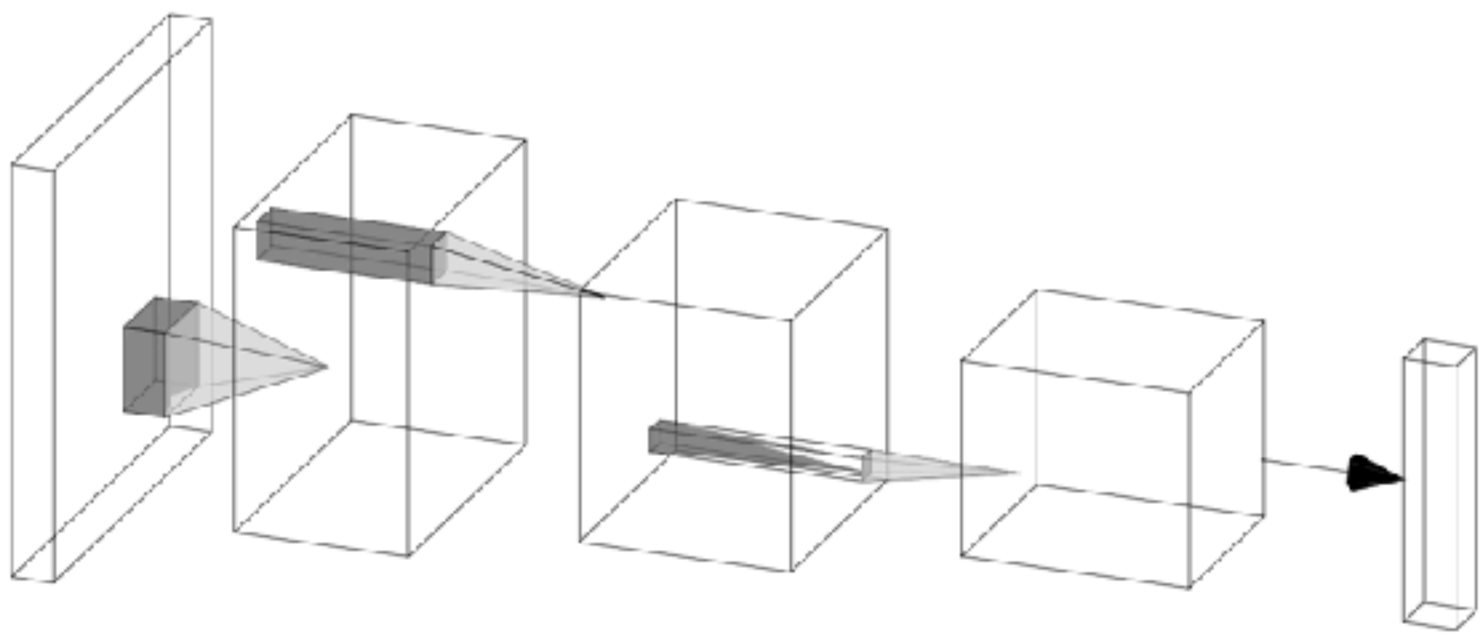


- Local connectivity
- Shared weight mechanism (much less weights)
- Location invariance

Inner product on a sliding window

Convolutional neural networks

2D case (images)



- Each neuron is a pattern detector
- The pattern is a combination of previous layer patterns

Pooling

- Small translation invariance (k steps)

$$m(\mathbf{x})(t) = \max_u(\mathbf{x}(t+u)), |u| \leq k$$

- Global pooling

$$h(\mathbf{x}) = \frac{1}{|\mathbf{x}|}\sum_t \mathbf{x}(t)$$

Max (detection score) or Sum (counting score) pooling depending on signal properties

Batch Normalization

Covariate shift:

- $\mathbf{w}_k$ update depends on the distribution $\mathbf{x}_k$
- Backpropagation changes $\mathbf{x}_k$ distribution

Fix each neuron's distribution by standardization

$$h(\mathbf{x}) = \gamma \frac{\mathbf{x} - \mu}{\sigma} + \theta$$

μ and σ are moving average, γ, θ are trainable

$$\mu \leftarrow (1 - \eta)\mu + \frac{\eta}{N} \sum_i \mathbf{x}_i$$

$$\sigma \leftarrow (1 - \eta)\sigma + \frac{\eta}{N} \sqrt{\sum_i (\mathbf{x}_i - \mu)^2}$$

Classification

Output a probability for each class

- Independent classes: binary crossentropy

$$\sigma(x) = \frac{1}{1+e^{-x}} \text{ sigmoid output layer activation}$$

$$l(y, F(\mathbf{x})) = (y-1)\log(1-F(\mathbf{x})) - y\log F(\mathbf{x})$$

- Exclusive classes: categorical crossentropy

$$\sigma(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \text{ softmax activation}$$

$$l(y, F(\mathbf{x})) = -\sum_i y_i \log F(\mathbf{x})[i]$$

Stochastic gradient descent

Objective function

$$F(w) = \sum_{0 \leq i \leq n} f_i(w)$$

Gradient descent

$$w^{k+1} = w^k - \eta \nabla F(w^k)$$

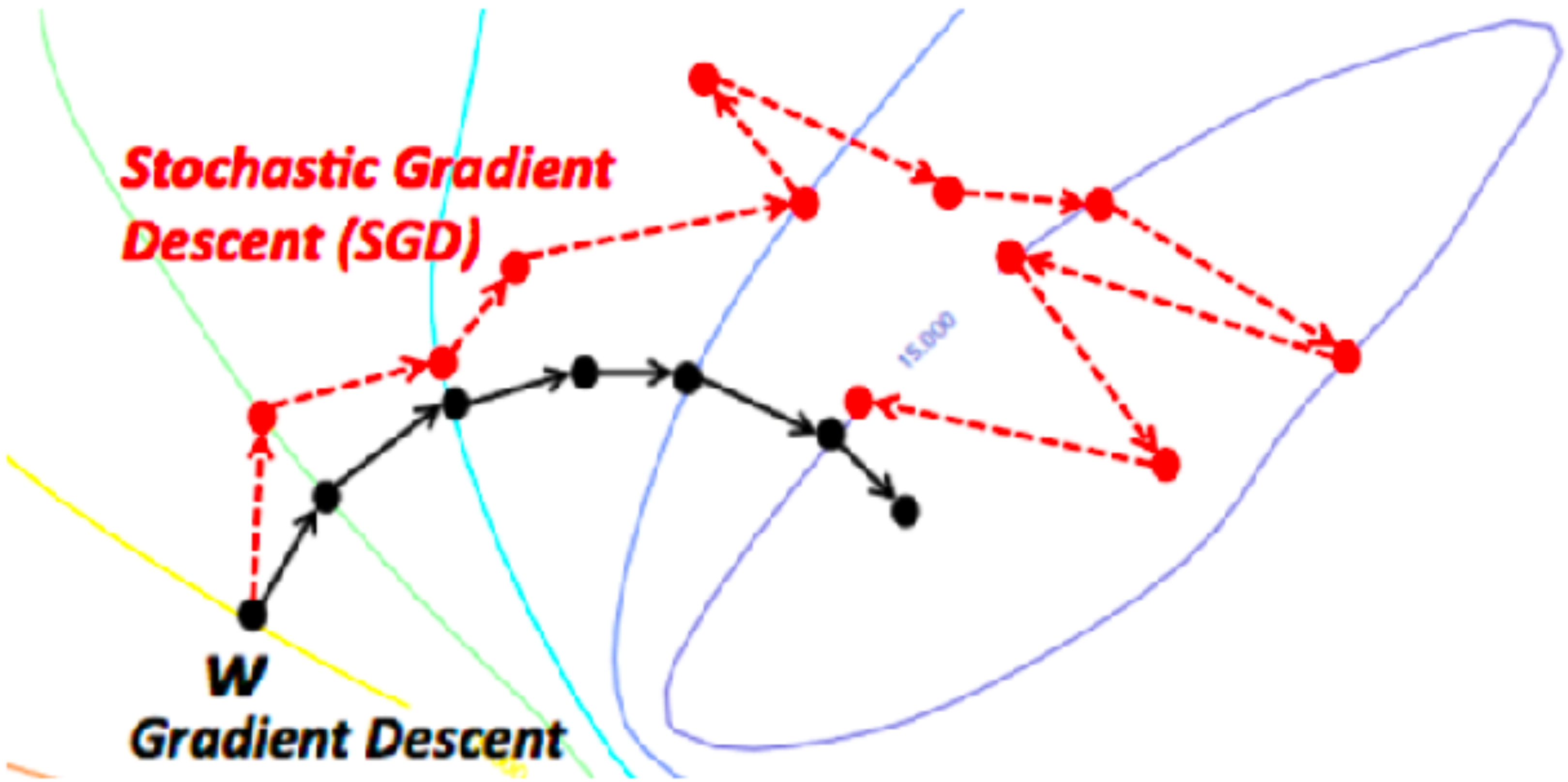
Stochastic gradient descent

$$w^{k+1} = w^k - \eta \nabla f_i(w^k), \quad i \sim \mathcal{U}(0, n)$$

Mini-batch stochastic gradient descent

$$w^{k+1} = w^k - \eta \sum_j \nabla f_{i_j}(w^k), \quad \forall j, i_j \sim \mathcal{U}(0, n)$$

Stochastic gradient descent



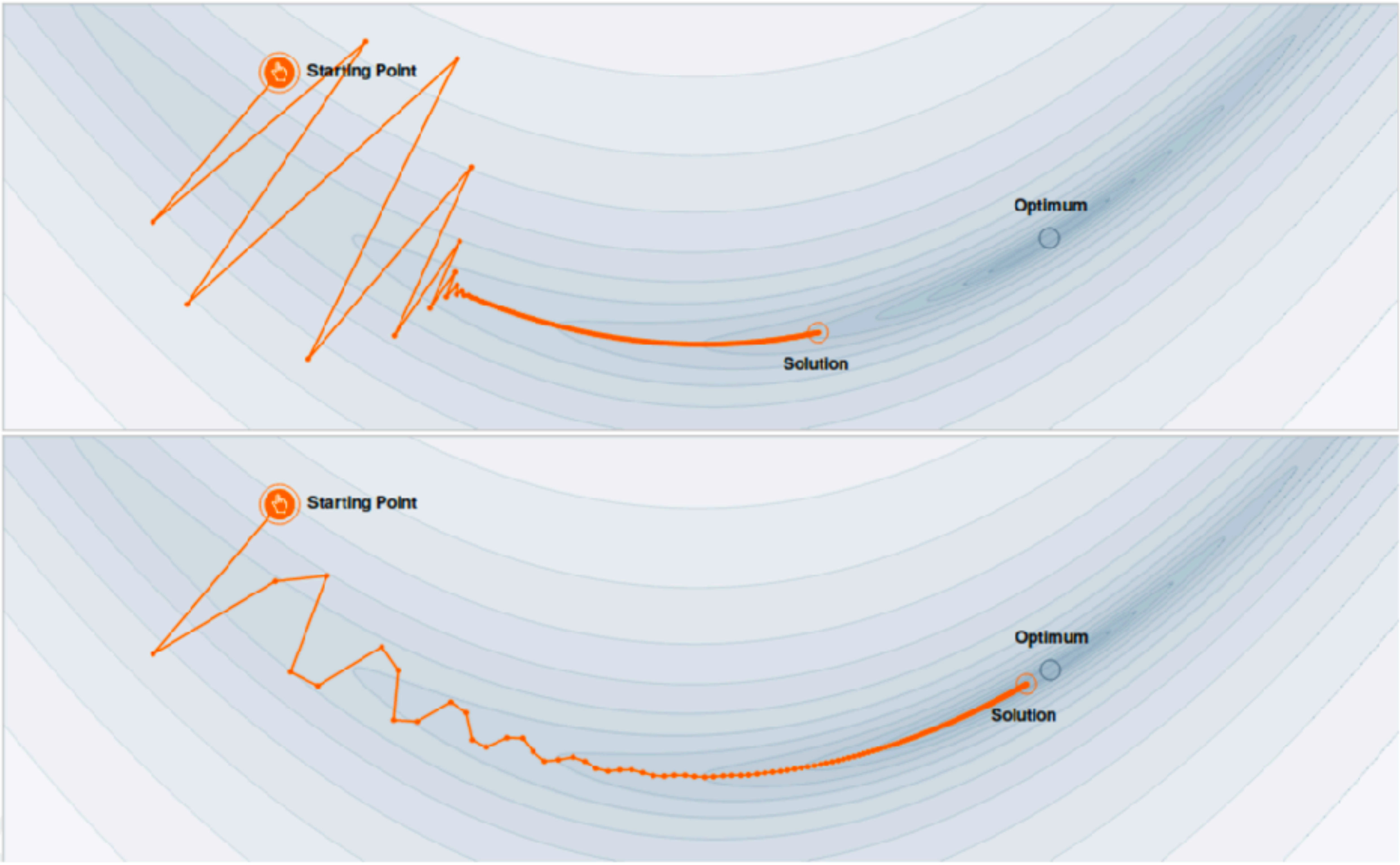
Decrease learning rate to ensure convergence

Momentum

$$z^{k+1} = \gamma z^k + (1-\gamma)\nabla f_i(w), \quad i \sim \mathcal{U}(0,n)$$
$$w^{k+1} = w^k - \eta z^{k+1}$$

- Cancel noise by averaging gradients
- Keeps the speed (avoid decreasing learning rates)
- $\gamma = 0.9$ typical values ($\gamma = 0.99$ for very small batch size)

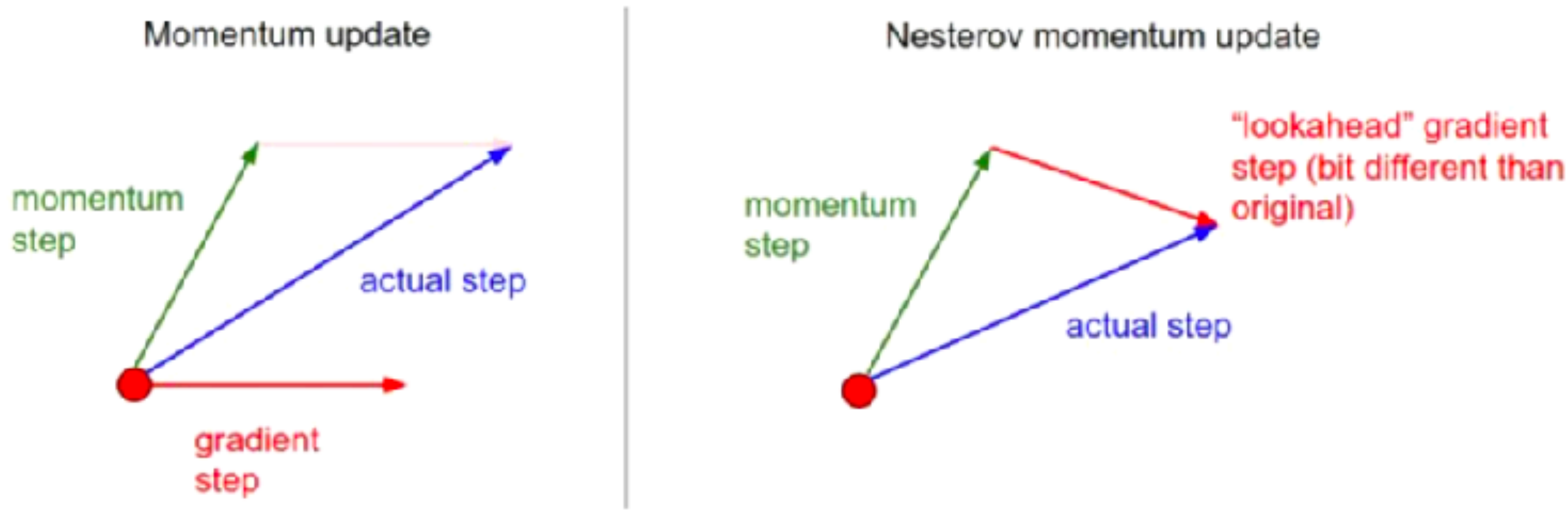
SGD, back to rehab!



Nesterov Momentum

Look ahead gradient

$$z^{k+1} = \gamma z^k + (1-\gamma)\nabla f_i(w^k - \eta z^k)$$
$$w^{k+1} = w^k - \eta z^k$$



Learning rate a.k.a. size of the gradient steps