MIT 9.520/6.860, Fall 2017
Statistical Learning Theory and Applications

Class 21: Neural Nets and Deep Representations

Road map

Last class:

- Part II: Data representations by **unsupervised learning**
  - Dictionary Learning
  - PCA
  - Sparse coding
  - K-means, K-flats

This class:

- Part III: **Deep** data representations (unsupervised, supervised)
  - Neural Networks basics
  - Autoencoders
  - ConvNets

Why learning?

Ideally: automatic, autonomous learning

- with as **little prior information** as possible,

  but also.....

- ...with as **little human supervision** as possible.

$$f(x) = \langle w, \Phi(x) \rangle_{\mathcal{F}}, \quad \forall x \in \mathcal{X}$$

Two-step learning scheme:

- **supervised or unsupervised learning** of $\Phi : \mathcal{X} \to \mathcal{F}$
- *supervised learning* of w in $\mathcal{F}$

Neural networks

Data representation schemes that involve **multiple layers**.

- Explicit parametrization of $\Phi(x) \in \mathcal{F}$
  - Nonlinear features
  - Linear projections and pointwise nonlinearities
- Multiple layers, multiple maps $\Phi_l, l = 1 \dots L$.
- Compositional $\Phi_{l-1} \circ \Phi_l(x)$
- Additional constraints on Φ_l
  - Locality
  - Sparsity
  - Covariance: tied values
  - Invariance: pooling
- Joint learning of $(\Phi(x), w)$.

In practice all is multilayer! (an old slide)

Pipeline

Raw data processing:

- compute some **low level** features,
- learn some **mid level** representation,
- ...
- use **supervised** learning.

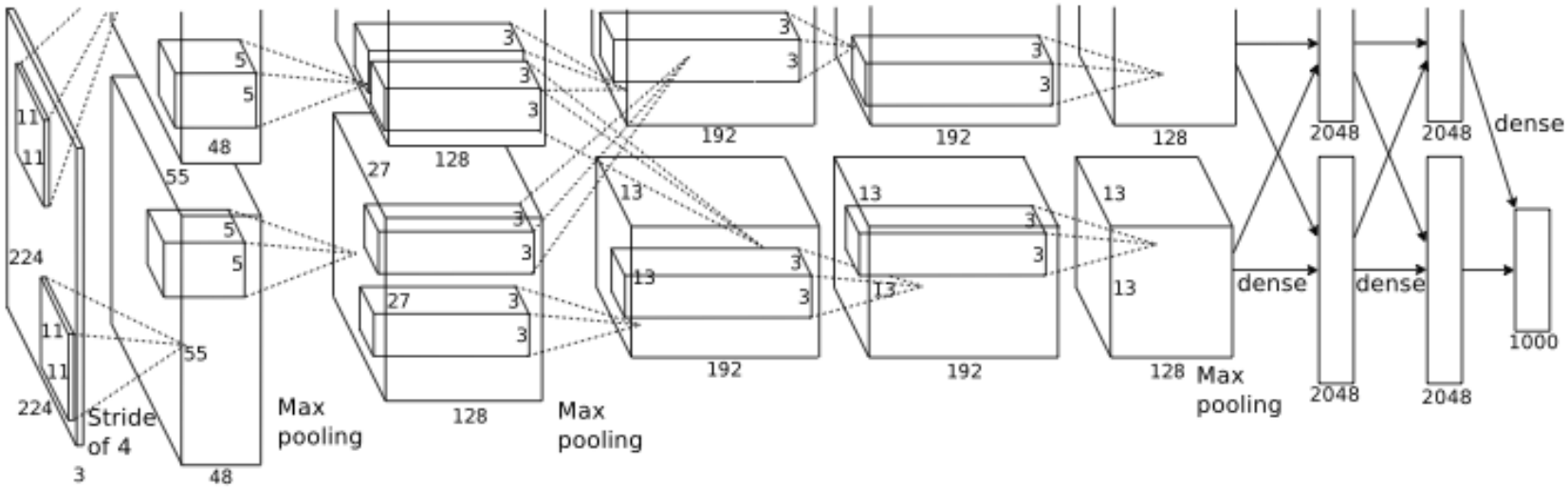
These stages are often done separately:

- Features by design, e.g. kernels OR
- Unsupervised feature learning
  - Use unlabeled data and reconstruction error loss.
- *Is it possible to design* **end-to-end** *learning systems?*

In practice all is deep learning! (updated slide)

Pipeline

- design some **wild- but "differentiable"** multilayer architecture.
- proceed with **end-to-end** learning!



Architecture (rather than feature) engineering.

Road Map

Part A: Neural networks basics

- Setting and definitions
- Learning, optimization

Part B: Architectures

- Auto-encoders
- Convolutional neural networks

Shallow nets

$$f(x) = w^\top \Phi(x), \quad \underbrace{x \mapsto \Phi(x)}_{\textbf{Fixed}}$$

.

Examples

- Dictionaries

$$\Phi(x) = \cos(B^\top x) = (\cos(\beta_1^\top x), \ldots, \cos(\beta_p^\top x))$$

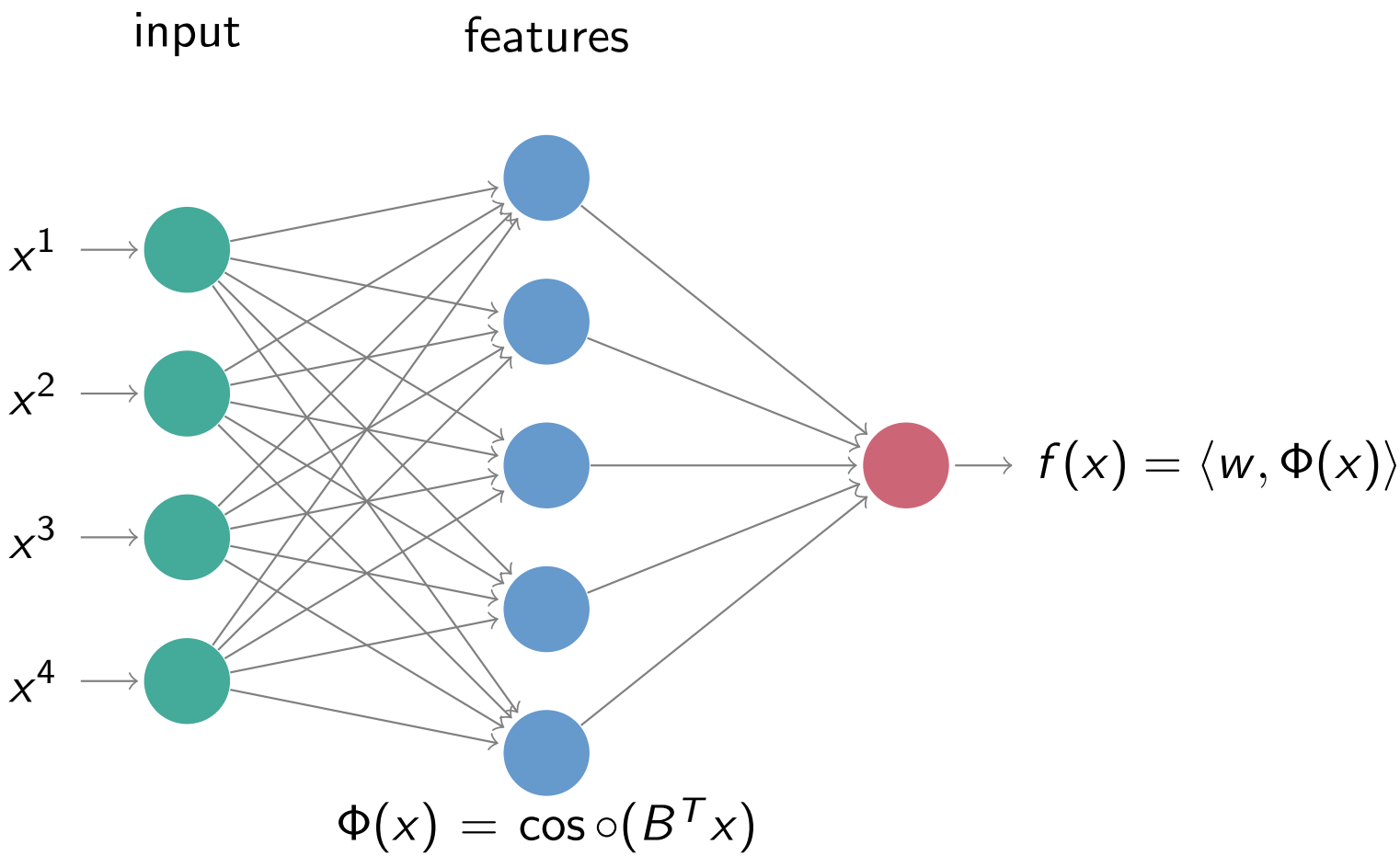
with $B = \beta_1, \ldots, \beta_p$ fixed frequencies.

- Kernel methods

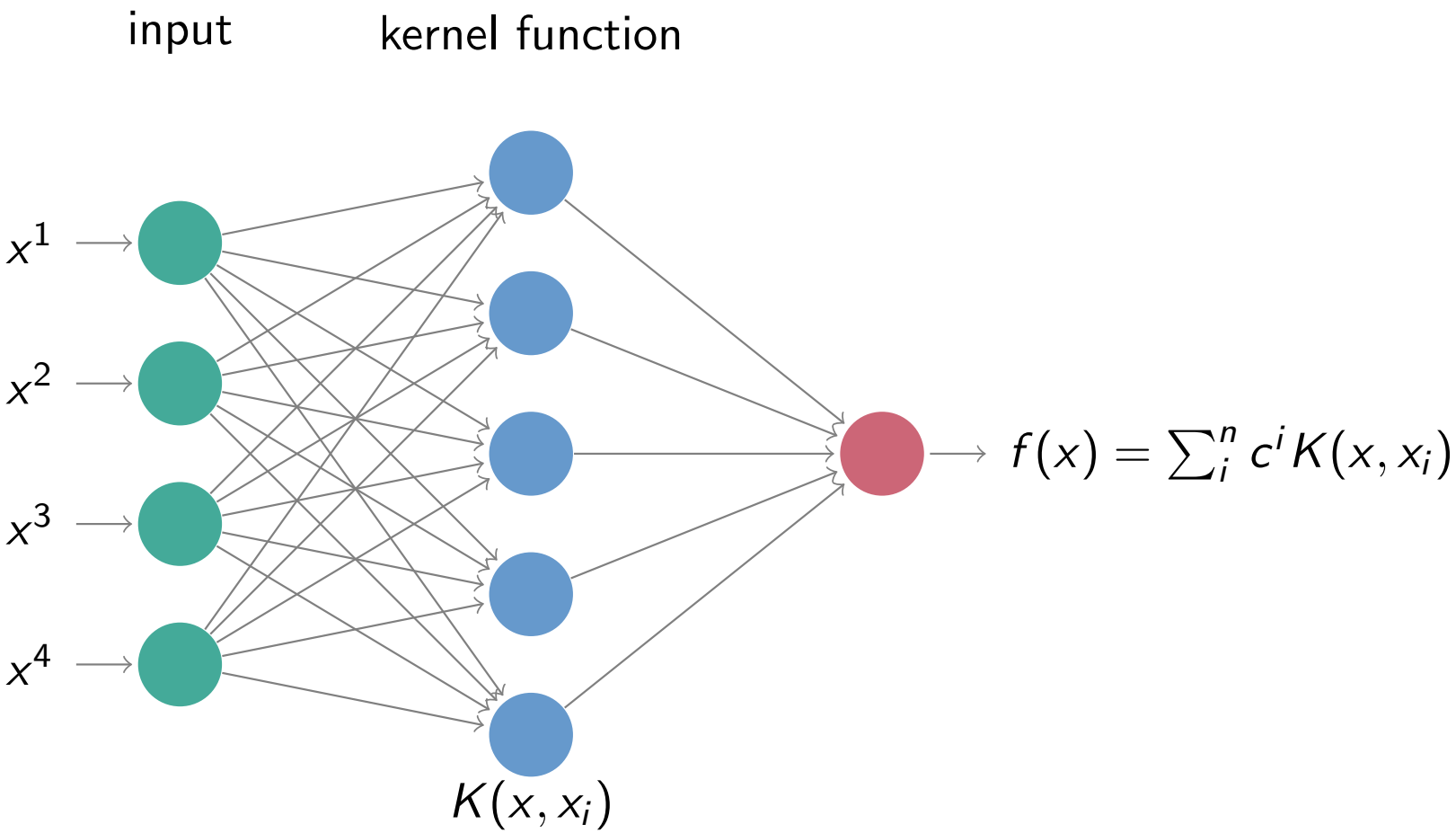
$$\Phi(x) = (e^{-\|\beta_1 - x\|^2}, \ldots, e^{-\|\beta_n - x\|^2})$$

with $\beta_1 = x_1, \ldots, \beta_n = x_n$ the input points.

Example: dictionaries



Example: kernel methods



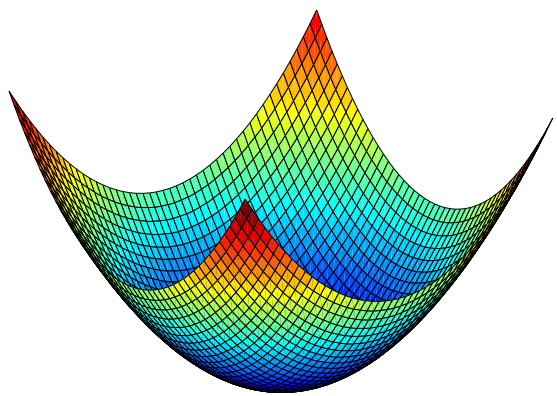
Shallow nets (cont.)

$$f(x) = w^\top \Phi(x), \quad \underbrace{x \mapsto \Phi(x)}_{\textbf{Fixed}}$$

Empirical Risk Minimization (ERM)

$$\min_w \sum_{i=1}^{n} (y_i - w^\top \Phi(x_i))^2$$

Note: Function f depends linearly on w: ERM problem is **convex**!



Interlude: optimization by Gradient Descent (GD)

Batch gradient descent

$$w^{(t+1)} = w^{(t)} - \gamma \nabla_w \widehat{\mathcal{E}}(w^{(t)})$$

where

$$\widehat{\mathcal{E}}(w) = \sum_{i=1}^{n} (y_i - w^\top \Phi(x_i))^2$$

so that

$$\nabla_w \widehat{\mathcal{E}}(w) = -2 \sum_{i=1}^{n} \Phi(x_i)^\top (y_i - w^\top \Phi(x_i))$$

Gradient descent illustrated: step size

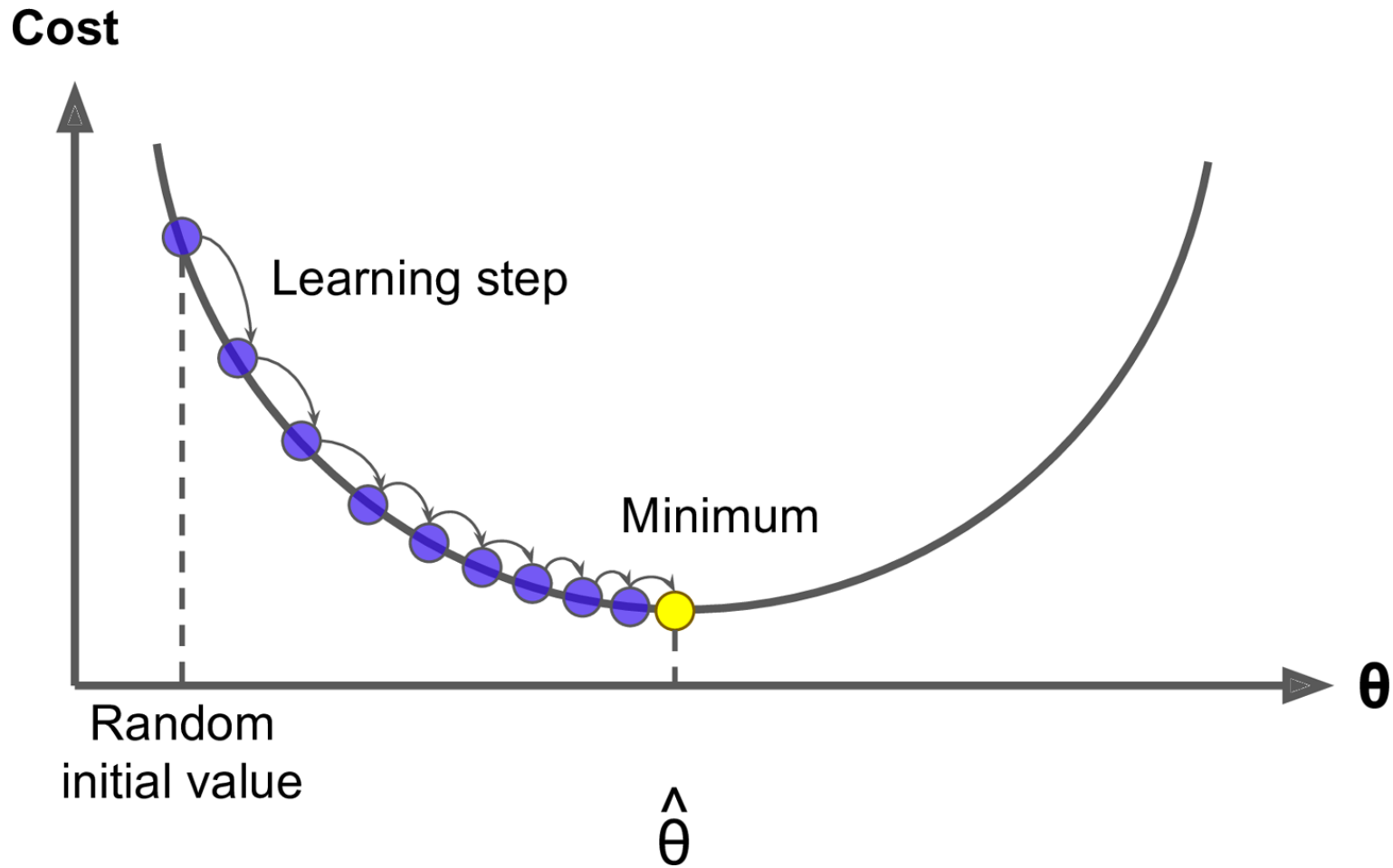


Image: *A. Geron, "Hands-on ML with scikit-learn and tensorFlow", 2017.*

Gradient descent illustrated: small step size

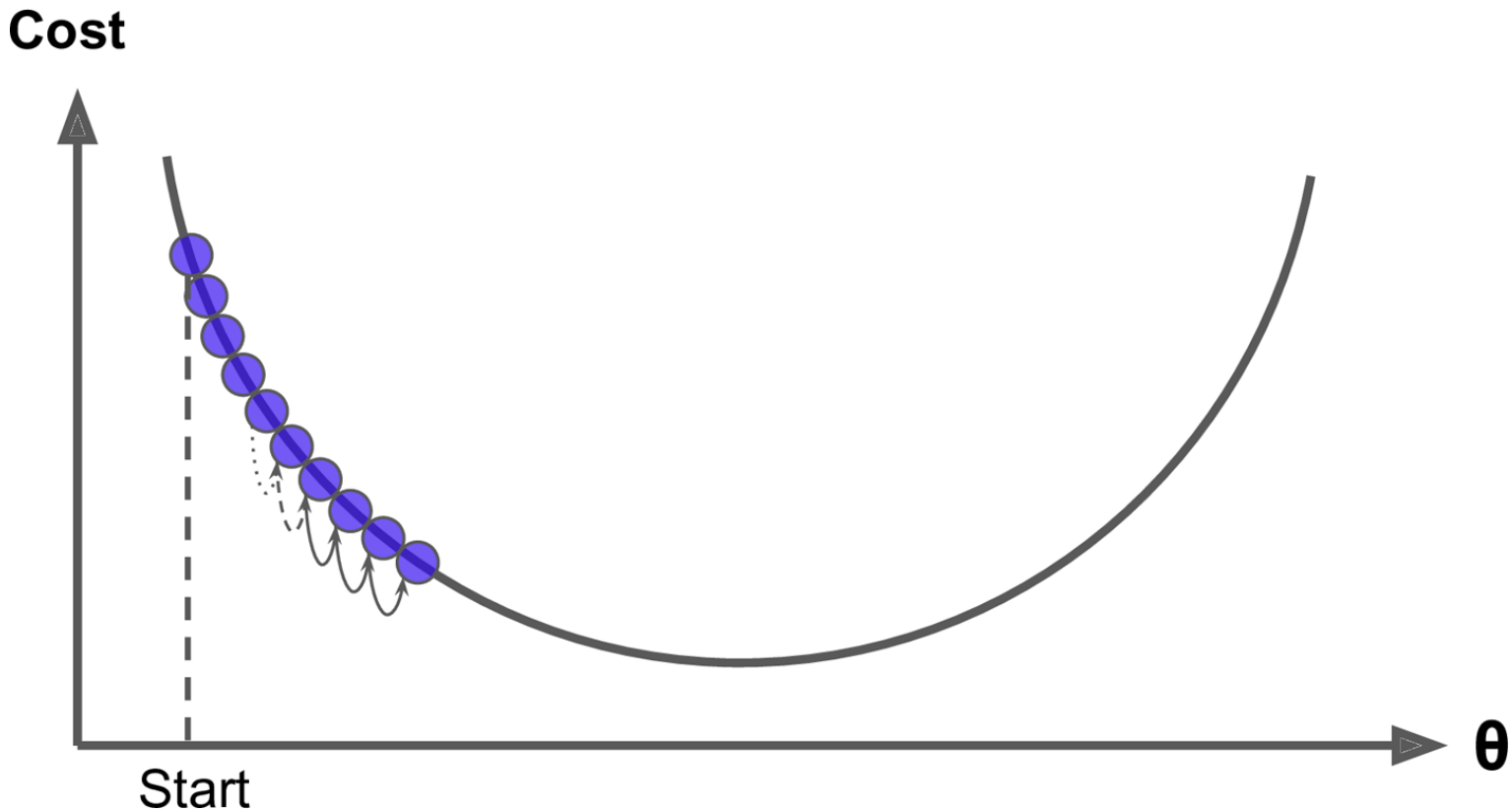


Image: *A. Geron, "Hands-on ML with scikit-learn and tensorFlow", 2017.*

Gradient descent illustrated: large step size

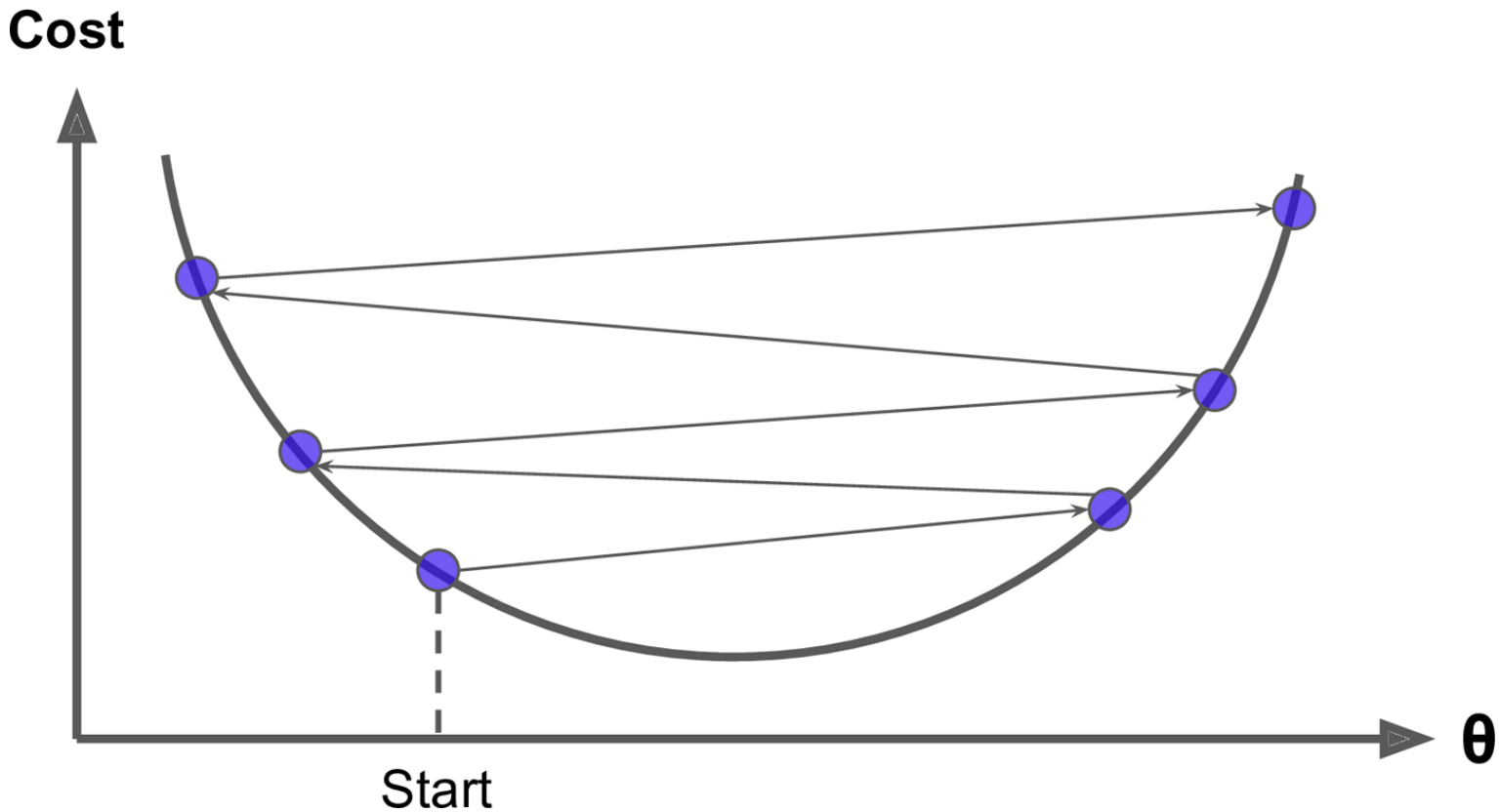


Image: *A. Geron, "Hands-on ML with scikit-learn and tensorFlow", 2017.*

Interlude: optimization by Gradient Descent (GD)

$$w^{(t+1)} = w^{(t)} + 2\gamma \sum_{i=1}^{n} \Phi(x_i)^\top (y_i - w^\top \Phi(x_i))$$

- **Constant step-size** depending on the *curvature* (Hessian norm)

- Iterative scheme is a **descent** method.
  - every step is a descent direction for the loss function

    $$\widehat{\mathcal{E}}(w^{(t+1)}) < \widehat{\mathcal{E}}(w^{(t)}),$$

    except when $w^{(t)}$ is optimal.

Stochastic gradient descent (SGD)

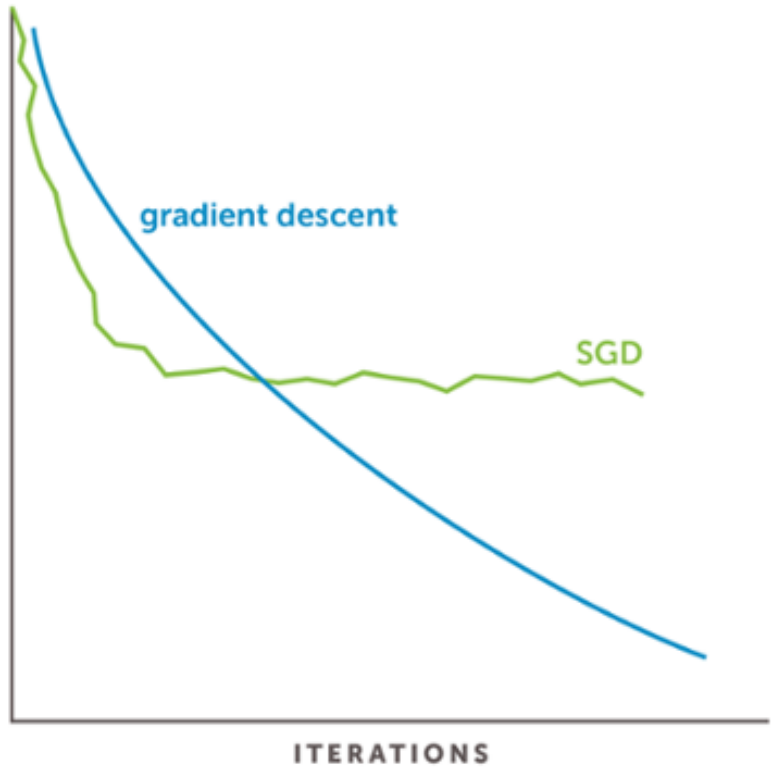
$$w_{t+1} = w_t + 2\gamma_t \Phi(x_t)^\top (y_t - w_t^\top \Phi(x_t))$$

Compare to

$$w_{t+1} = w_t + 2\gamma \sum_{i=1}^{n} \Phi(x_i)^\top (y_i - w_t^\top \Phi(x_i))$$

- Decaying step-size $\gamma = 1/\sqrt{t}$
- Lower **iteration cost**
- Multiple passes (**epochs**) over data needed
- Not a **descent** method (SGD?)

SGD vs GD



SGD behavior (vs. GD):

- not a descend method: each update can reduce or enlarge the loss,
- converges faster: more frequent updates,
- takes longer to reach global or local minimum: needs multiple passes.
  - regularization: early stopping, less prone to overfitting.

Batch vs. Mini-batch vs. Stochastic GD

Parameter convergence for linear regression

$$f(x) = w^T x, \quad w = (\theta_1, \theta_0)$$

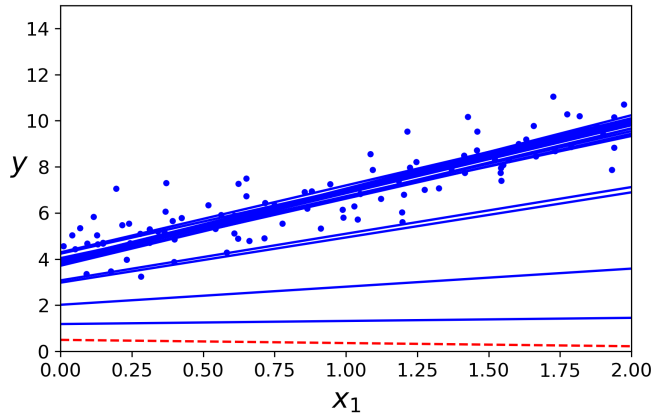
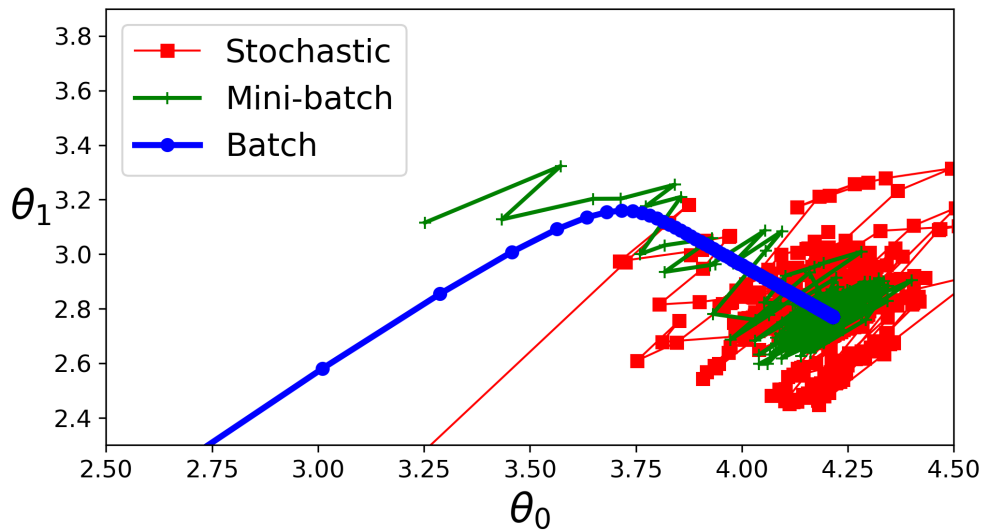


Figure: SGD iterations



Summary so far

Given data $(x_1, y_1), \ldots, (x_n, y_n)$ and a fixed representation Φ

- Consider
$$f(x) = w^\top \Phi(x)$$
- Find w by SGD
$$w_{t+1} = w_t + 2\gamma_t \Phi(x_t)^\top (y_t - w^\top \Phi(x_t))$$

Can we jointly learn Φ?

Shallow neural networks

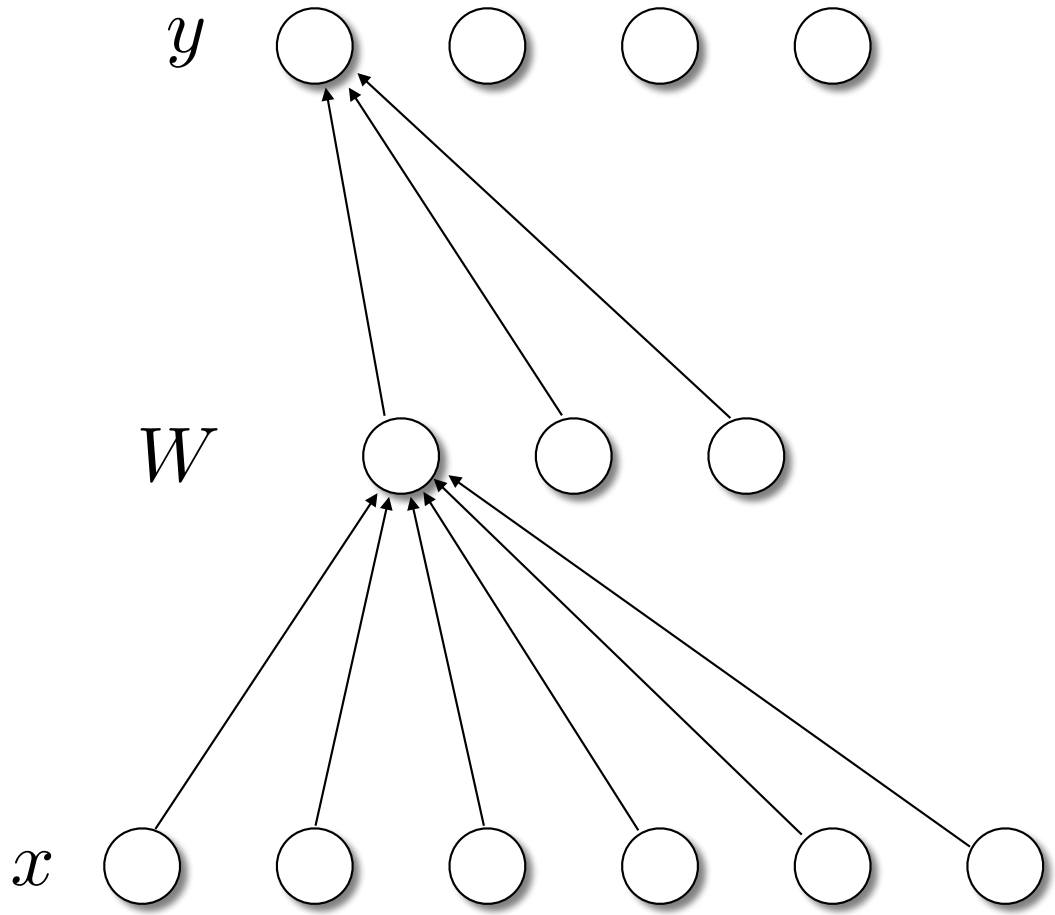
Neural networks correspond to a **specific** choice of the feature space

$$\mathcal{F} \subset \left\{ \Phi : \ \forall x \in \mathcal{X}, \ \ \Phi(x) = \sigma\left(W^T x + b\right) \right\}$$

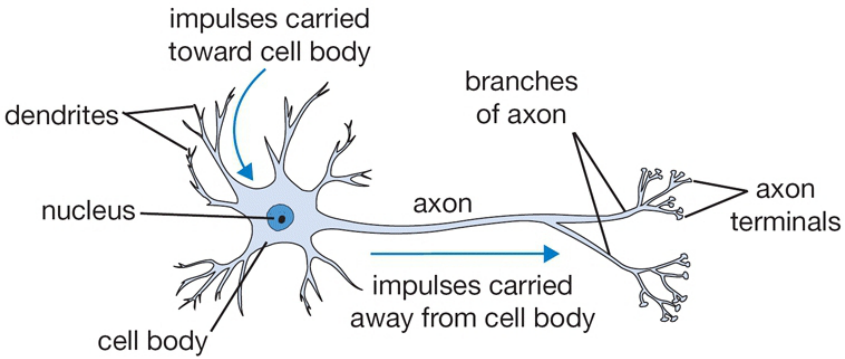
where:

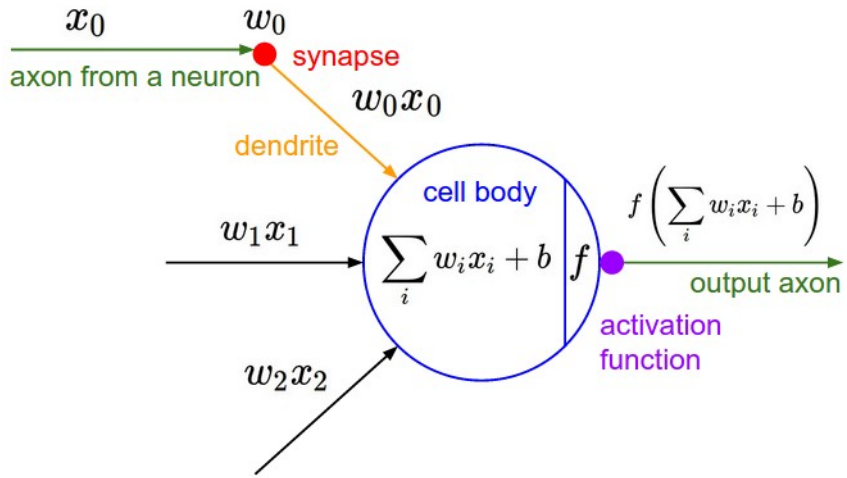
- $\sigma : \mathcal{X} \to \mathcal{X}$; **activation** operator defined component-wise by **activation functions** $s : \mathbb{R} \to \mathbb{R}$
- W is $p \times d$ **weight** matrix
- $b \in \mathbb{R}^d$ is an **offset** vector.

Neural networks illustrated



Neurons





- Each neuron computes an **inner product** using a column of the weight matrix W.
- The non-linearity σ is the **neuron activation** function.

Image: *Stanford CS231n: CNNs for Visual Recognition, 2017.*

Neural nets vs kernel methods

Learning with kernels

Given K, find the coefficients $c^1, \ldots, c^n$ in the linear expansion

$$f(x) = \sum_{i=1}^{n} c^i K(x_i, x)$$

by typically solving a **convex** problem.

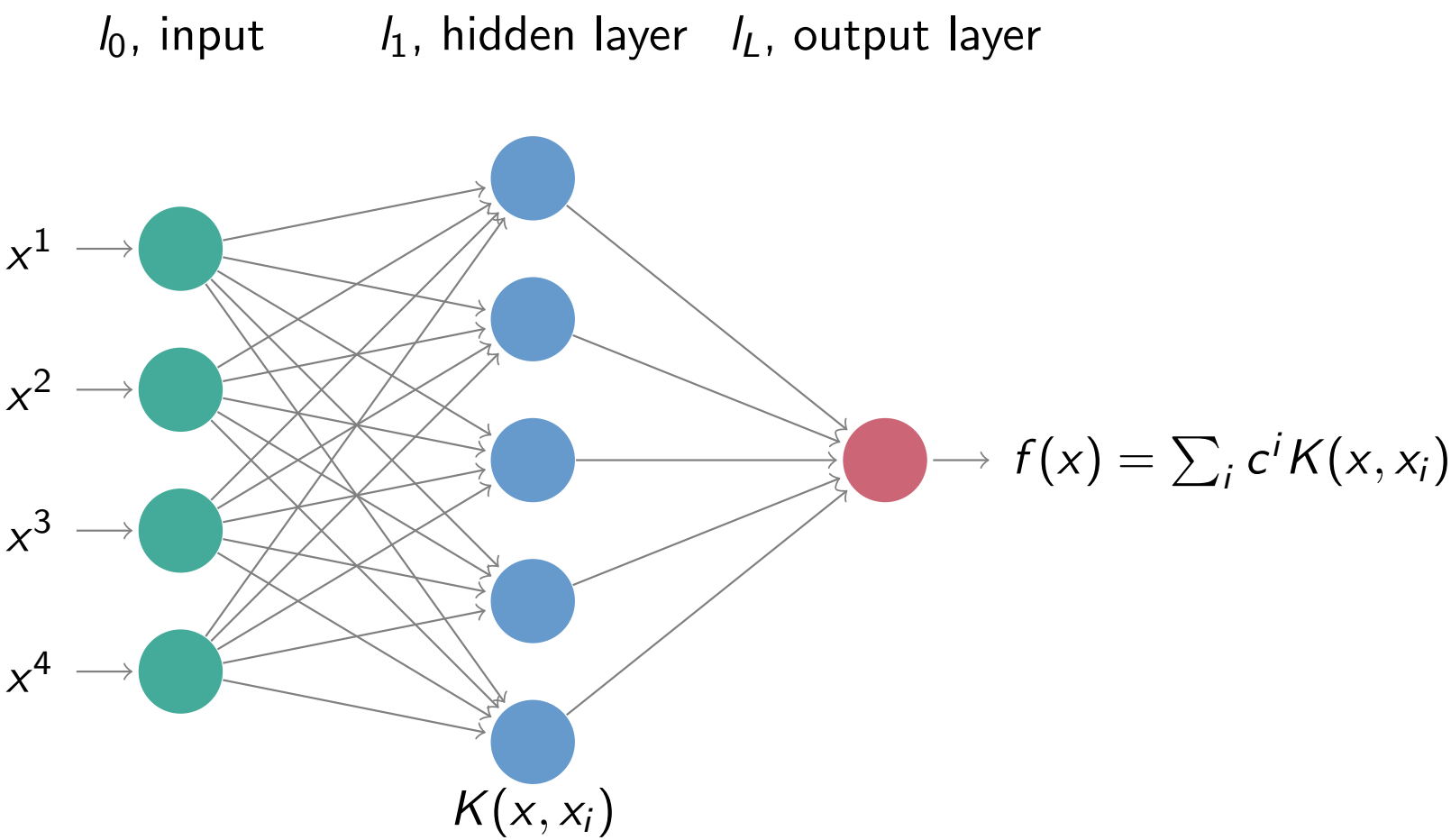
Learning in neural nets

Find the coefficients $c^1, \ldots, c^n$ and the weights $W^1, \ldots, W^p, b^1, \ldots, b^p$

$$f(x) = \sum_{j=1}^{p} c^j s\left(\left\langle W^j, x \right\rangle + b^j\right)$$

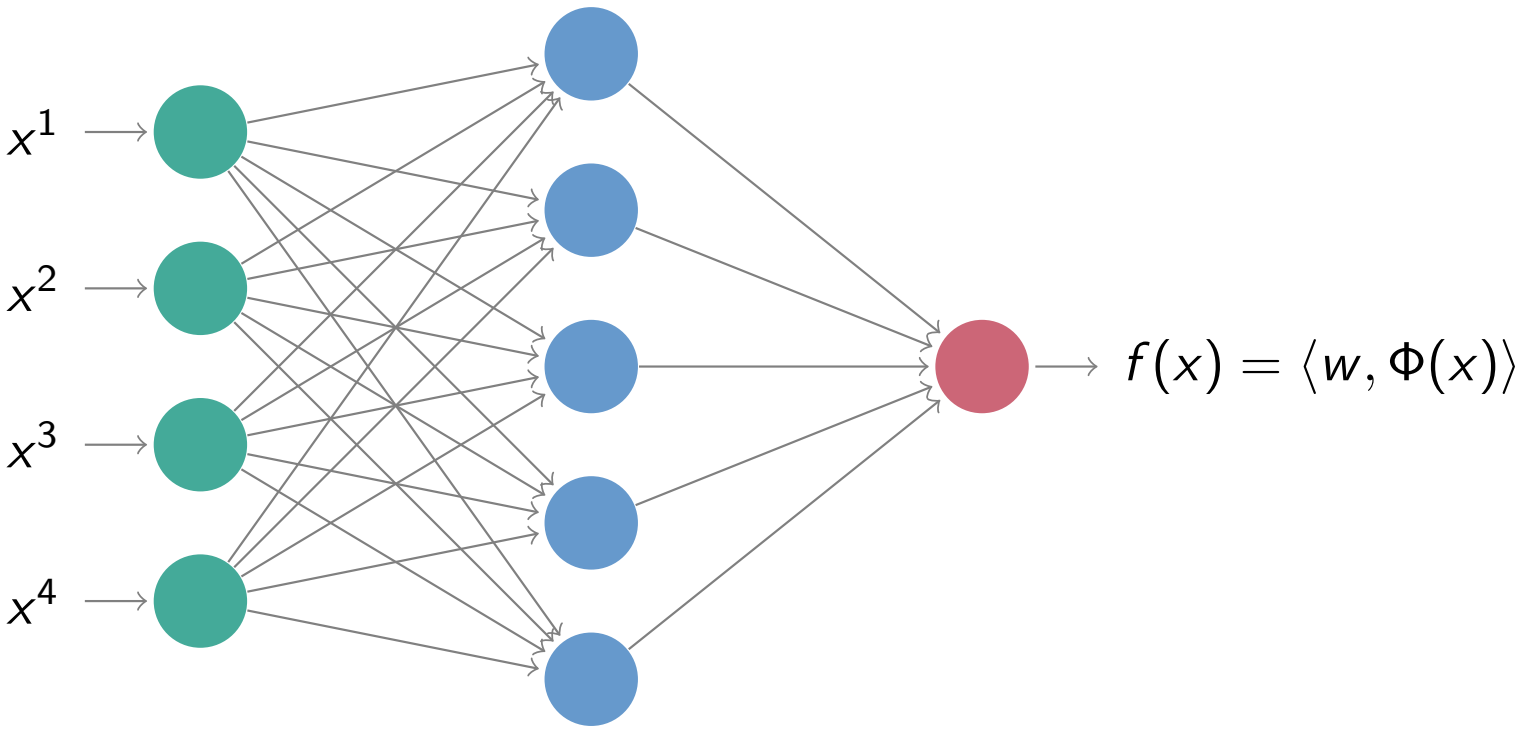
by typically solving a **non-convex** problem.

Computations in kernels



Computations in neural nets

l_0, input l_1, hidden layer l_L, output layer



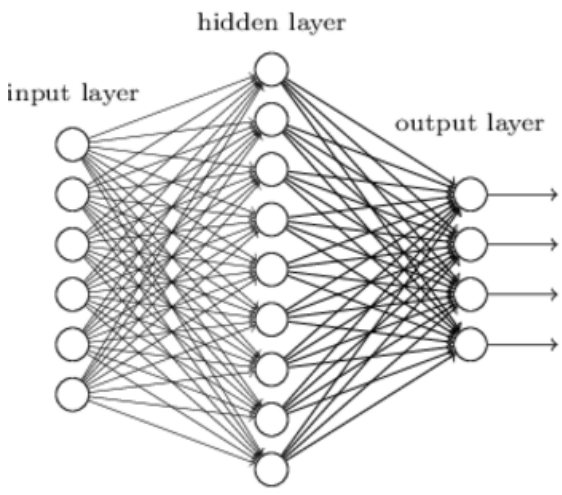
Neural nets vs kernel methods (cont.)

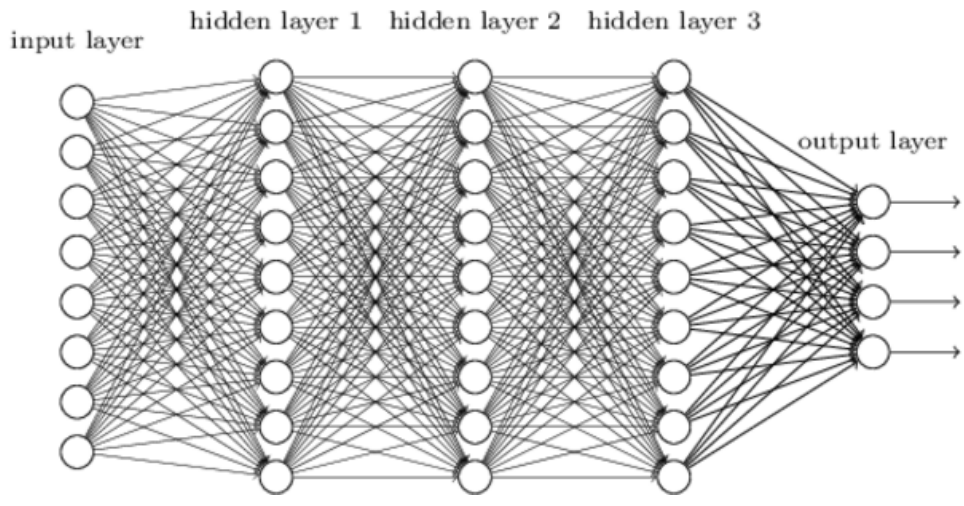
$$f(x) = \sum_{j=1}^{p} c^j s\left(\left\langle W^j, x \right\rangle\right) \quad \text{vs} \quad f(x) = \sum_{i=1}^{n} c^i K(x_i, x)$$

A comparison

- kernel methods lead to **convex** problems,
- the weights (*centers*) are the **training points**,
- ... but with a **fixed representation**,
- ... and are *currently* prohibitive for large scale learning (**memory**!).

Deep neural networks





Depth: function spaces by composition

Given input space $\mathcal{X}$, e.g. $\mathcal{X} = \mathbb{R}^d$ and output space $\mathcal{Y}$, e.g. $\mathcal{Y} = \mathbb{R}^T$:

- Sequence of **domains**

$$\mathcal{X}_\ell = \mathbb{R}^{d_\ell}, \quad d_\ell \in \mathbb{N}, \quad \ell = 1, \ldots, L,$$

such that $\mathcal{X}_1 = \mathcal{X}$ and $\mathcal{X}_L = \mathcal{Y}$, hence $d_1 = d$, $d_L = T$.

- Sequence of **function spaces**

$$\overline{\mathcal{H}}_\ell \subset \{h \ : \ h : \mathcal{X}_{\ell-1} \to \mathcal{X}_\ell\}, \quad \ell = 2, \ldots, L$$

and

$$\mathcal{H}_\ell = \{f : \mathcal{X}_1 \to \mathcal{X}_\ell \ : \ f = f_\ell \circ \cdots \circ f_1, \quad f_j \in \mathcal{H}_j\}, \quad j = 1, \ldots, \ell.$$

Depth: function parametrization

Deep neural nets correspond to specific **compositional function spaces**

$$\overline{\mathcal{H}}_\ell \subset \left\{ h \ : \ \forall x \in \mathcal{X}_{\ell-1}, \quad h(x) = \sigma_\ell \left(W_\ell^T x + b_\ell \right) \right\}, \quad \ell = 2, \ldots, L$$

where:

- $\sigma_\ell : \mathcal{X}_\ell \to \mathcal{X}_\ell$; **activation** operators defined component-wise by **activation functions**, $s_\ell : \mathbb{R} \to \mathbb{R}$
- W_ℓ are $d_{\ell-1} \times d_\ell$ **weight** matrices, and
- $b_\ell \in \mathbb{R}^{d_\ell}$ are **offset** vectors.

Neural network with L **layers** *(*$L-2$ **hidden***),* d_ℓ **units** *per layer.*

Supervised neural nets: regression

Regression: $\mathcal{X}_L = \mathcal{Y} = \mathbb{R}$

last activation function can be chosen to be the **identity**

$$f(x) = \langle w_L, h(x_{L-1}) \rangle + b_L \in \mathbb{R}.$$

Equivalently, writing one step of the **recursion**:

$$f(x) = \sum_{j=1}^{d_{L-1}} w_L^j s_{L-1} \left(\left\langle W_{L-1}^j, h(x_{L-2}) \right\rangle + b_{L-1}^j \right) + b_L.$$

Supervised neural nets: classification

Classification: $\mathcal{X}_L = \mathcal{Y} = [1, \ldots, T]$

last activation function can be chosen to be the **softmax**

$$f(x) = \sigma(\langle W_L, h(x_{L-1})\rangle + b_L) \quad \sigma : \mathbb{R}^T \to [0,1]^T,$$

where W_L is $T \times d_{l-1}$ and

$$s(a^j) = \frac{e^{a^j}}{\sum_{j=1}^{T} e^{a^j}}, \quad a \in \mathbb{R}^T, \quad s : \mathbb{R} \to [0,1]$$

- **Softmax regression** (multinomial logistic regression).
- Probability distribution over T outputs.
- $\arg\max_j(f(x)^j)$ for classification.

Summary: Deep neural networks

Basic idea: **compose** simply **parameterized** representations

$$\Phi = \Phi_L \circ \cdots \circ \Phi_2 \circ \Phi_1$$

Let $d_0 = D$ and

$$\Phi_\ell : \mathbb{R}^{d_{\ell-1}} \to \mathbb{R}^{d_\ell}, \quad \ell = 1, \ldots, L$$

and in particular

$$\Phi_\ell = \sigma \circ W_\ell, \quad \ell = 1, \ldots, L$$

where

$$W_\ell : \mathbb{R}^{d_{\ell-1}} \to \mathbb{R}^{d_\ell}, \quad \ell = 1, \ldots, L$$

linear/affine and σ is a non linear map acting component-wise

$$\sigma : \mathbb{R} \to \mathbb{R}.$$

Summary: Deep neural nets

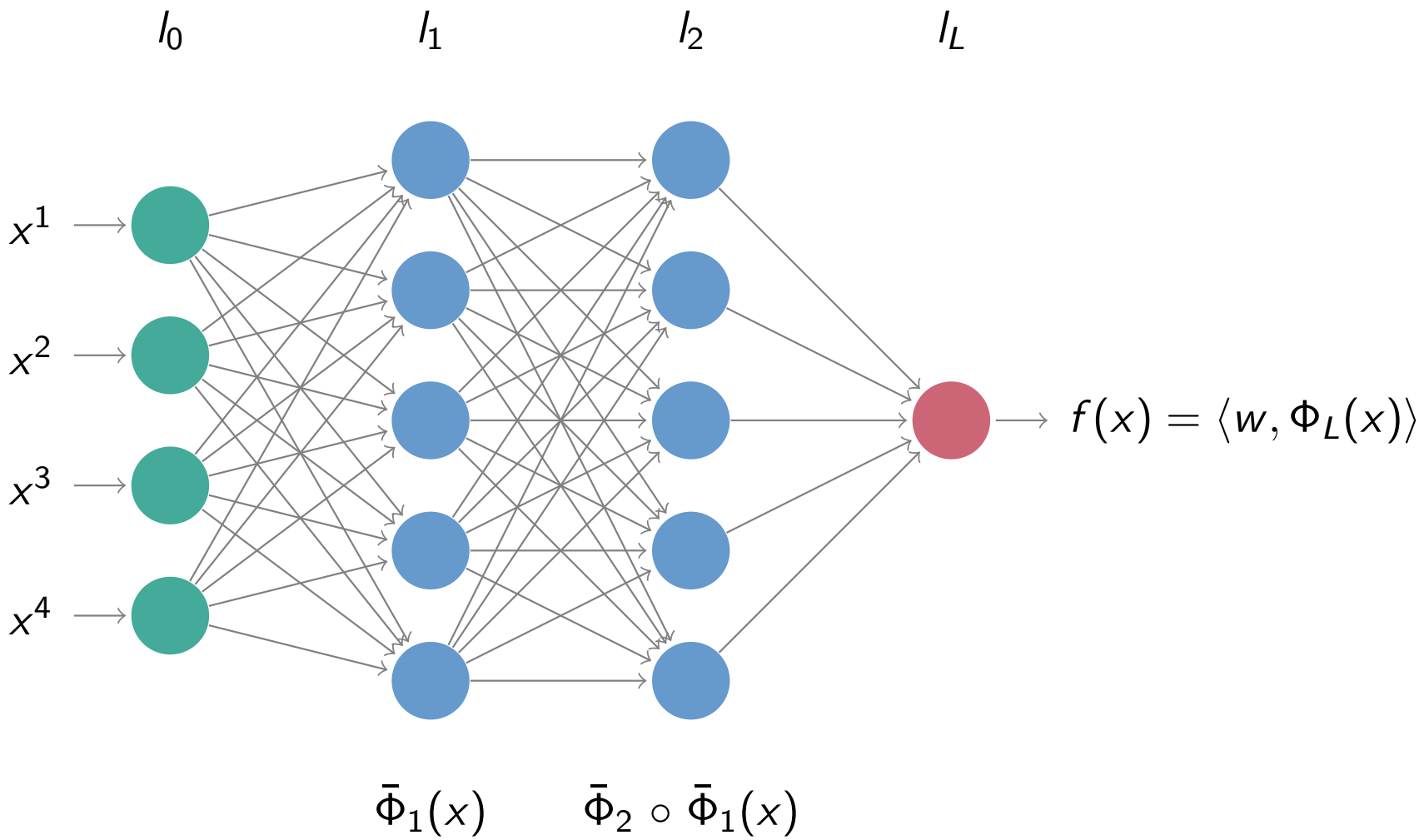
$$f(x) = w^\top \Phi_L(x), \qquad \underbrace{\Phi_L = \overline{\Phi}_L \circ \cdots \circ \overline{\Phi}_1}_{\text{compositional representation}}$$

$$\overline{\Phi}_1 = \sigma \circ W_1 \quad \ldots \quad \overline{\Phi}_L = \sigma \circ W_L$$

ERM

$$\min_{w,(W_j)_j} \frac{1}{n} \sum_{i=1}^{n} (y_i - w^\top \Phi_L(x_i))^2$$

Computations: Deep neural nets



Neural networks jargon

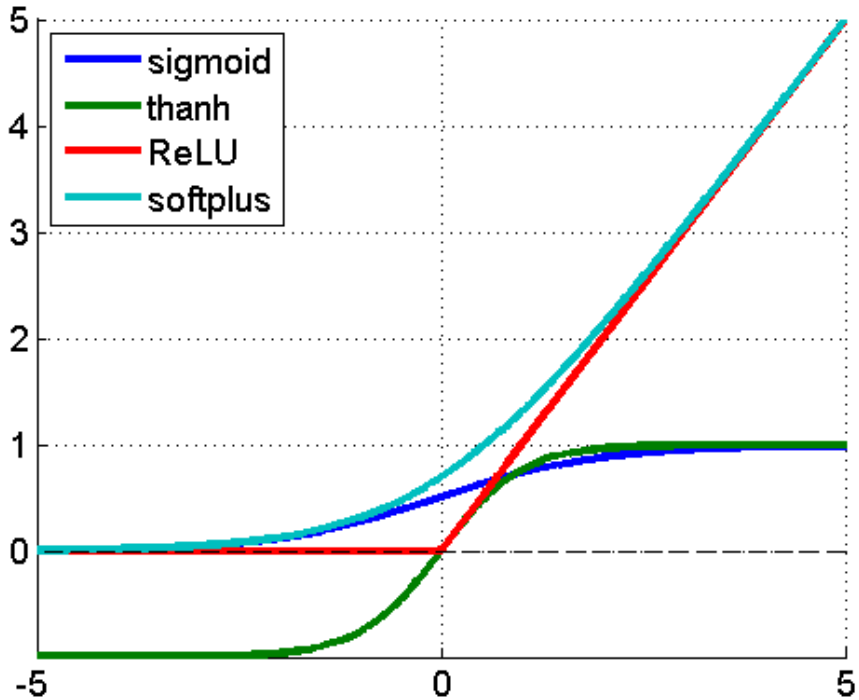
$$\Phi_L(x) = \sigma(W_L \dots \sigma(W_2 \sigma(W_1 x)))$$

- **hidden layer**: any intermediate representation $\Phi_\ell, \ell = \{1, L-1\}$.
- number of **hidden units**: dimensionalities $(d_\ell)_\ell$
- **activation function**: nonlinearity σ

Activation functions

For $\alpha \in \mathbb{R}$

- **sigmoid** (logistic) $s(\alpha) = 1/(1 + e^{-\alpha})$,
- **hyperbolic tangent** $s(\alpha) = (e^{\alpha} - e^{-\alpha})/(e^{\alpha} + e^{-\alpha})$,
- **ReLU** (ramp, hinge) $s(\alpha) = |\alpha|_+$,
- **softplus** $s(\alpha) = \log(1 + e^{\alpha})$.



Note: If the activation is **linear**: equivalent to a **single linear layer**.

Logistic and Softmax regression

Recall **logistic regression** loss (linear activation):

$$f(x) = \langle w, \Phi(x) \rangle$$

$$V(f(x), y) = \log(1 + e^{-yf(x)}) = -yf(x) + \log(1 + e^{yf(x)})$$

"Cross-entropy" loss (single output, logistic activation):

$$f(x) = s(\langle w, \Phi(x) \rangle), \quad s(\alpha) = (1 + e^{-a})$$

$$V(f(x), y) = -\left(y \log(f(x)) + (1 - y) \log(1 - f(x))\right) = -yf(x) + \log(1 + e^{yf(x)})$$

"Cross-entropy" loss (multiple outputs, softmax activation):

$$f(x) = \sigma(\langle W, \Phi(x) \rangle), \quad s(a^j) = e^{a^j} / \sum_{j=1}^{T} e^{a^j}$$

$$V(f(x), y) = -\sum_{j=1}^{T} y^j \log(f(x)^j) = -\sum_{j=1}^{T} y^j \left(f(x)^j - \sum_{i=1}^{T} \log(e^{f(x)^i}) \right)$$

Questions with deep networks

$$f_{w,(W_\ell)_\ell}(x) = w^\top \Phi_{(W_\ell)_\ell}(x), \qquad \Phi_{(W_\ell)_\ell} = \sigma(W_L \dots \sigma(W_2 \sigma(W_1 x)))$$

1. **Approximation:** how **rich** are the models?
2. **Optimization:** can we **train** efficiently?
3. **Generalization:** from **finite data**? overfitting?

TP will discuss these (**next classes!**)

Neural networks function spaces

Consider nonlinear space of functions of the form $f_{w,(W_\ell)_\ell} : \mathbb{R}^D \to \mathbb{R}$,

$$f_{w,(W_\ell)_\ell}(x) = w^\top \Phi_{(W_\ell)_\ell}(x), \qquad \Phi_{(W_\ell)_\ell} = \sigma(W_L \dots \sigma(W_2 \sigma(W_1 x)))$$

where $w, (W_\ell)_\ell$ may vary.

Very little structure . . . but we can:

- train by **gradient descent** (next)
- get (some) **approximation/statistical** guarantees (later, next classes)

One layer NN: learning

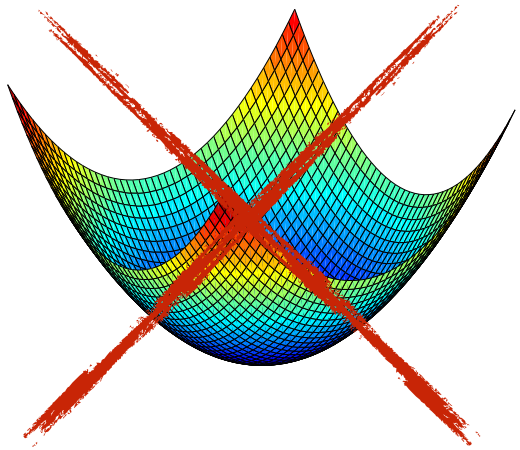
Consider single hidden layer:

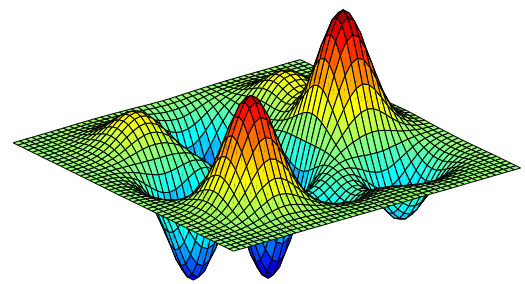
$$f_{(w,W)}(x) = w^\top \sigma(Wx) = \sum_{j=1}^{p} w_j \sigma\left(x^\top W^j\right)$$

and ERM

$$\min_{w,W} \widehat{\mathcal{E}}(w, W), \qquad \widehat{\mathcal{E}}(w, W) = \sum_{i=1}^{n} (y_i - f_{(w,W)}(x_i))^2.$$

Problem is non-convex! (possibly smooth depending on σ)





Gradient descent: non-convex problems

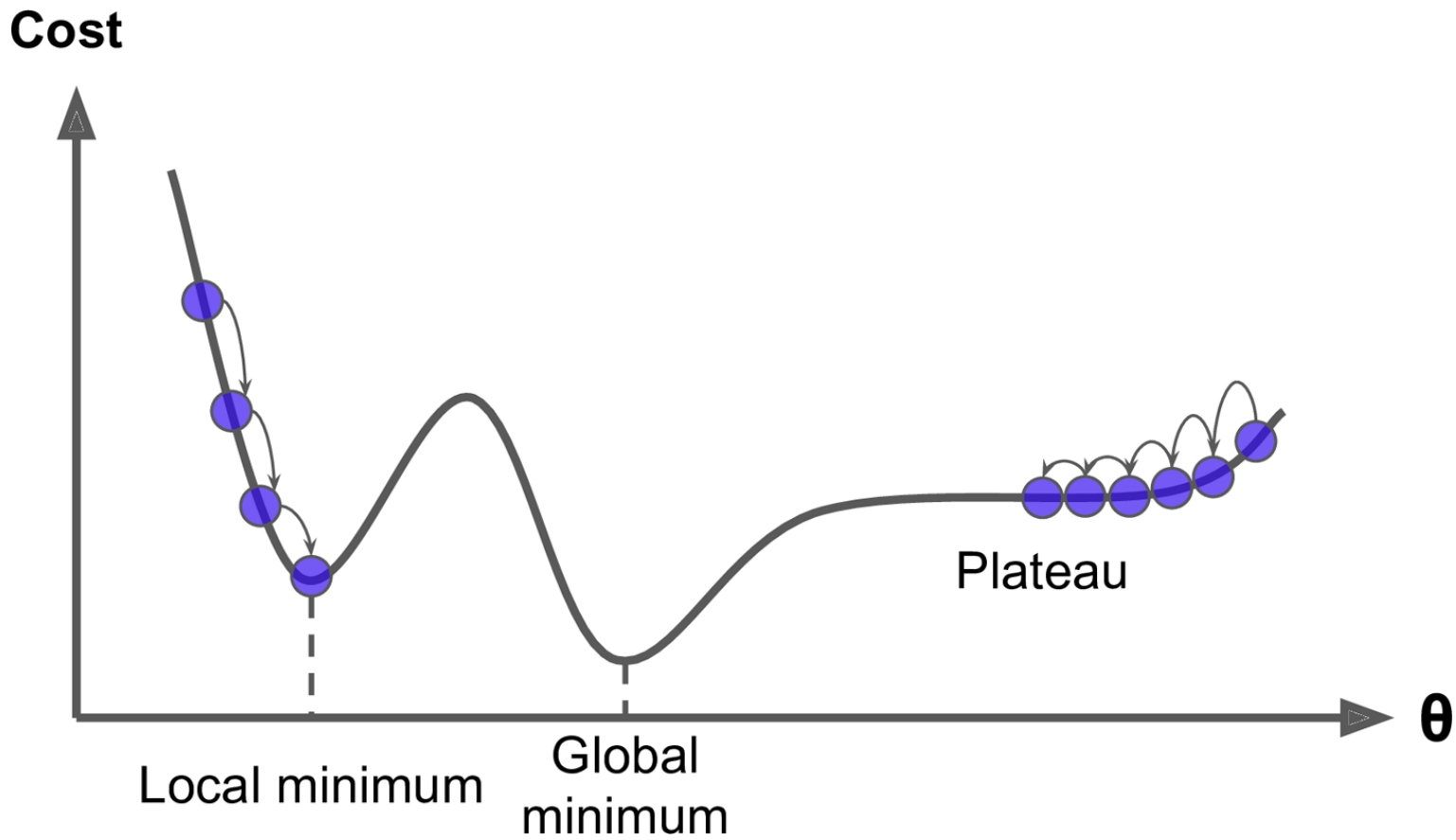


Image: *A. Geron, "Hands-on ML with scikit-learn and tensorFlow", 2017.*

Back-propagation & GD

$$\min_{w,W} \widehat{\mathcal{E}}(w, W), \qquad \widehat{\mathcal{E}}(w, W) = \sum_{i=1}^{n} (y_i - f_{(w,W)}(x_i))^2.$$

Approximate minimizer is computed via **GD** iterations

$$\begin{aligned} w_j^{t+1} &= w_j^t - \gamma_t \frac{\partial \widehat{\mathcal{E}}}{\partial w_j}(w^t, W^t) \\ W_{j,k}^{t+1} &= W_{j,k}^t - \gamma_t \frac{\partial \widehat{\mathcal{E}}}{\partial W_{j,k}}(w^{t+1}, W^t) \end{aligned}$$

where the step-size $(\gamma_t)_t$ is the **learning rate**.

Back-propagation & chain rule

$$\frac{\partial \widehat{\mathcal{E}}}{\partial w_j}(w, W) = \frac{\partial \widehat{\mathcal{E}}}{\partial f_{(w,W)}} \frac{\partial f_{(w,W)}}{\partial w_j}$$

$$\frac{\partial \widehat{\mathcal{E}}}{\partial W_{j,k}}(w, W) = \frac{\partial \widehat{\mathcal{E}}}{\partial f_{(w,W)}} \frac{\partial f_{(w,W)}}{\partial \sigma(W_j^\top \cdot)} \frac{\partial \sigma(W_j^\top \cdot)}{\partial W_{j,k}}$$

Direct computations show that:

$$\frac{\partial \widehat{\mathcal{E}}}{\partial w_j}(w, W) = -2 \sum_{i=1}^{n} \underbrace{\left(y_i - f_{(w,W)}(x_i)\right)}_{\Delta_i} \underbrace{\sigma(W_j^\top x_i)}_{h_{i,j}}$$

$$\frac{\partial \widehat{\mathcal{E}}}{\partial W_{j,k}}(w, W) = -2 \sum_{i=1}^{n} \underbrace{\left(y_i - f_{(w,W)}(x_i)\right)\sigma'(W_j^\top x_i)}_{\eta_{i,j}} w_j x_i^k$$

$$\eta_{i,j} = \Delta_i \sigma'(W_j^\top x_i)$$

Back-propagation equations

$$\eta_{i,j} = \Delta_i(\dots)$$

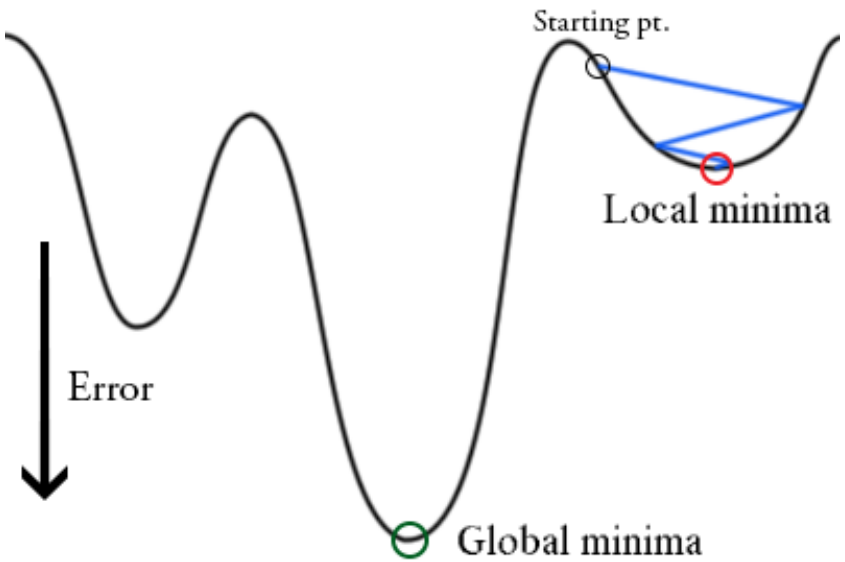
Weight updates are performed in two steps:

- **Forward pass**: compute function values with weights
- **Backward pass**: compute errors and propagate

SGD (and non-convexity)

$$w_j^{(t+1)} = w_j^{(t)} - \gamma_t 2(y_t - f_{(w^{(t)}, W^{(t)})}(x_t))\sigma((W_j^{(t)})^\top x_t)$$

$$W_{j,k}^{(t+1)} = W_{j,k}^{(t)} - \gamma_t (y_t - f_{(w^{(t+1)}, W^{(t)})}(x_t)) w_j^{(t+1)} \sigma'((W_j^{(t)})^\top x_t) x_t^k$$



Remarks

- Optimization by **gradient methods**– typically SGD
- **Online** update rules are potentially biologically plausible– **Hebbian learning** rules describing neuron **plasticity**
- **Multiple layers** can be analogously considered
- **Multiple step-size per layers** can be considered
- **NO** convergence guarantees
- Making things work:
  - **Initialization** is tricky- more later
  - **Activation function**
  - **Regularization**: weight, stochastic
  - **Normalization**: weight, batch, layer, ...
  - **Accelerated** optimization/GD
  - Training set **augmentation**
  - More tricks later

Questions

1. *Optimization: can we train efficiently?*
   - Why does SG iterative method work?
2. **Approximation:** how **rich** are the models?
   - What is the benefit of multiple layers?
3. **Generalization:** from **finite data**? overfitting?

TP will discuss these (**next classes!**)

Approximation theory

One-layer, see [Pinkus '99] and references therein,

$$f_u(x) = \sum_{j=1}^{u} w^j \sigma\left(x^\top W^j\right)$$

Universality, if σ is not a polynomial,

$$\lim_{u\to\infty} \min_{f_u} \|f_u - f\| = 0, \qquad \forall f \in C(\mathbb{R}^D)$$

Approximation rates for smooth σ,

$$\min_{f_u} \max_{f\in W^{2,s}} \|f_u - f\| \lesssim u^{-\frac{s}{d}}$$

where $W^{2,s}$ is the space of functions with s (integrable) derivatives

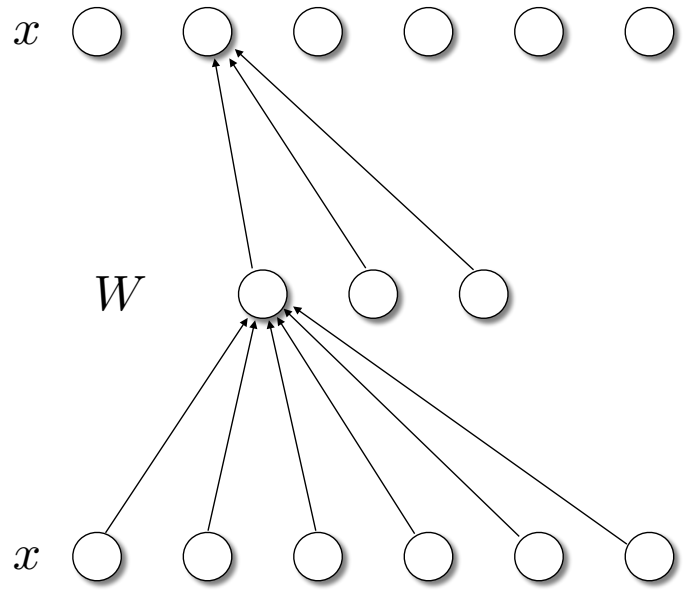
- **Representation** via Kolmogorov Superposition theorem
- Is Sobolev the right **smoothness class**? [Barron '93, Poggio, Mhaskar et al. '16]
- **Multiple layers**: TP will discuss (**next classes!**)

Unsupervised learning with neural networks

Autoencoders: parametric encodings by learning to reconstruct.

- Unlabeled data abound
- Pre-training: weights to initialize supervised learning
- (Nonlinear) dimensionality reduction: bottleneck features
- Embeddings for metrics: similarity, ranking, one-shot, ...

Autoencoders



- Neural network with **one input layer, one output layer and one (or more) hidden layers**.
- Output layer has **equally** many nodes as the input layer.
- Trained to **predict the input**.

Autoencoders (cont.)

- Autoencoder with a hidden layer of k units
- **Representation-reconstruction** pair with $\mathcal{X} = \mathbb{R}^d, \mathcal{F}_k = \mathbb{R}^k, k < d$
- **Encoder:**

$$\Phi : \mathcal{X} \to \mathcal{F}_k, \quad \Phi(x) = \sigma\left(W_\Phi x\right), \quad \forall x \in \mathcal{X}$$

- **Decoder:**

$$\Psi : \mathcal{F}_k \to \mathcal{X}, \quad \Psi(z) = \sigma\left(W_\Psi z\right), \quad \forall z \in \mathcal{F}_k.$$

- Code: $z \in \mathcal{F}_k$

Autoencoders (cont.)

$\mathcal{X} = \mathbb{R}^d, \mathcal{F}_k = \mathbb{R}^k, k < d$

$$\Phi : \mathcal{X} \to \mathcal{F}_k, \quad \Phi(x) = \sigma\left(W_\Phi x\right), \quad \forall x \in \mathcal{X}$$
$$\Psi : \mathcal{F}_k \to \mathcal{X}, \quad \Psi(z) = \sigma\left(W_\Psi z\right), \quad \forall z \in \mathcal{F}_k.$$

Reconstruction:

$$x' = \Psi \circ \Phi(x) = \sigma\left(W_\Psi \sigma\left(W_\Phi x\right)\right)$$

- If **tied-weights**: $W_\Psi = W_\Phi^T = W \in \mathbb{R}^{k \times d}$
- Φ, Ψ can be made **deep and compositional**.

Autoencoders & dictionary learning

$$\Phi(x) = \sigma\left(W_\Psi x\right), \quad \Psi(z) = \sigma\left(W_\Phi z\right)$$

- **Dictionary learning**: weights can be seen as dictionary **atoms**.

$$\min_{\Phi,\Psi} \frac{1}{n} \sum_{i=1}^{n} \|x_i - \Psi \circ \Phi(x_i)\|^2 = \min_{W_\Psi, W_\Phi} \frac{1}{n} \sum_{i=1}^{n} \|x_i - \sigma\left(W_\Psi \sigma\left(W_\Phi x\right)\right)\|^2$$

Notes:

- Connections with so called **energy models** [LeCun et al.].
- **Probabilistic/Bayesian** interpretations/formulation (e.g. Boltzmann machines [Hinton, Salagutinov, 2006]).

Linear autoencoders & PCA

$$\Phi(x) = \sigma(Wx), \quad \Psi(z) = \sigma\left(W^T z\right)$$

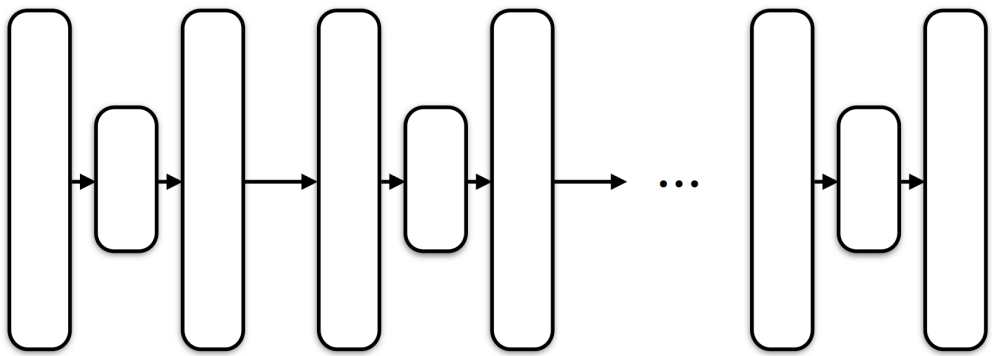
$$\min_{\Phi,\Psi} \frac{1}{n} \sum_{i=1}^{n} \|x_i - \Psi \circ \Phi(x_i)\|^2 = \min_{W} \frac{1}{n} \sum_{i=1}^{n} \left\|x_i - WW^\top x\right\|^2$$

- If we let
$$\mathcal{W} = \{W : \mathcal{F} \to \mathcal{X}, \text{ linear} \mid W^T W = I\}.$$
the solution to the autoencoder loss is PCA.
- Multiple linear layers collapse to one.

Stacked auto-encoders

Multiple layers of auto-encoders can be **stacked** [Hinton et al '06]...

$$\underbrace{(\Phi_1 \circ \Psi_1)}_{\text{Autoencoder}} \circ (\Phi_2 \circ \Psi_2) \cdots \circ (\Phi_\ell \circ \Psi_\ell)$$



...with the potential of obtaining **richer** representations.

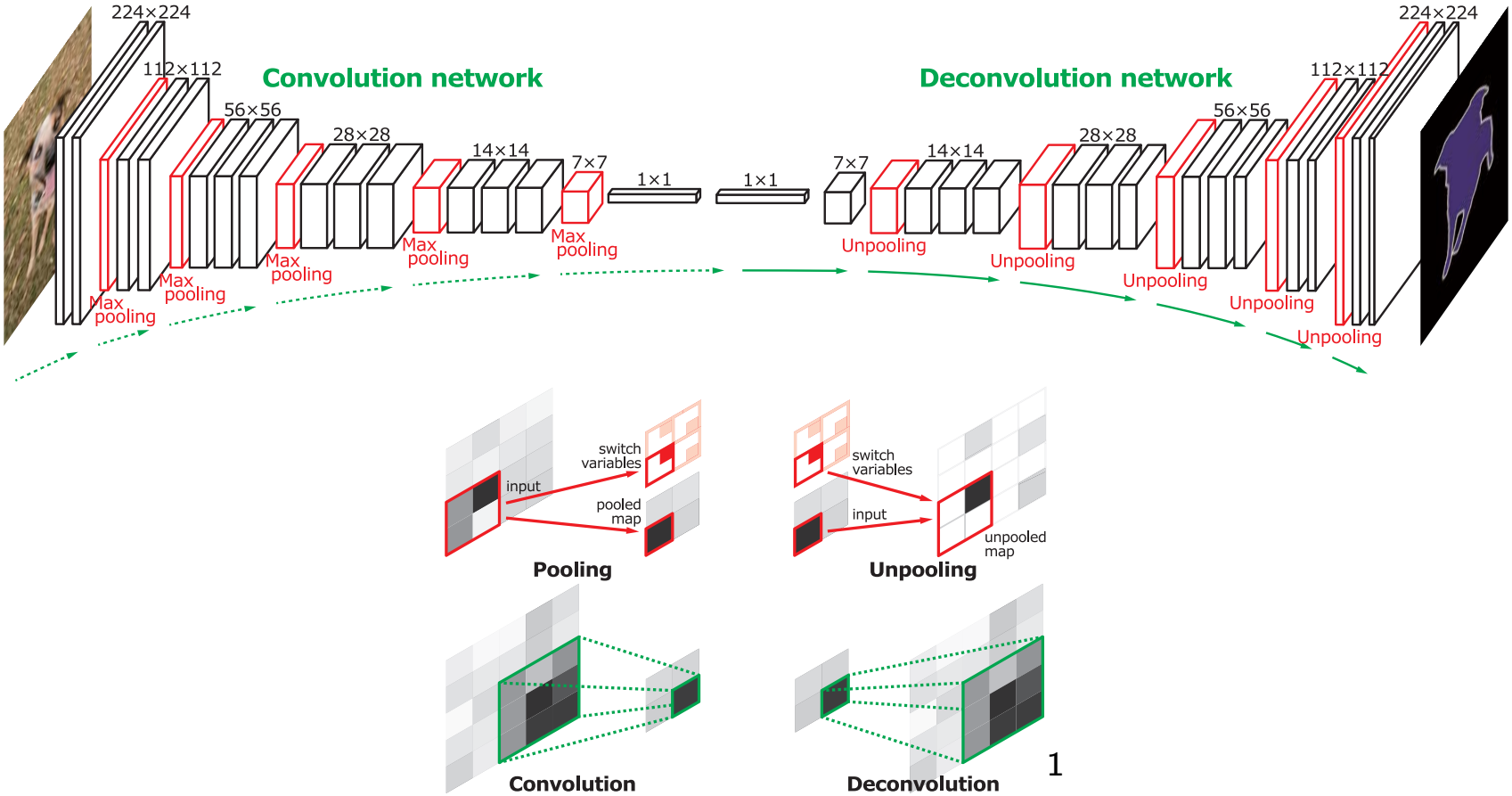
Are autoencoders useful?

- Pre-training has not delivered:
  - large-scale, supervised training works best.

- Unsupervised, self-supervised, weakly-supervised learning in vision.
- Data visualization, dimensionality reduction.
- Latent space description, distribution learning and sampling.

- Ongoing work: **denoising** autoencoders, **sparse** autoencoders, **contrastive** autoencoders, **transforming** autoencoders, **variational** autoencoders ...

Convolutional auto-encoders

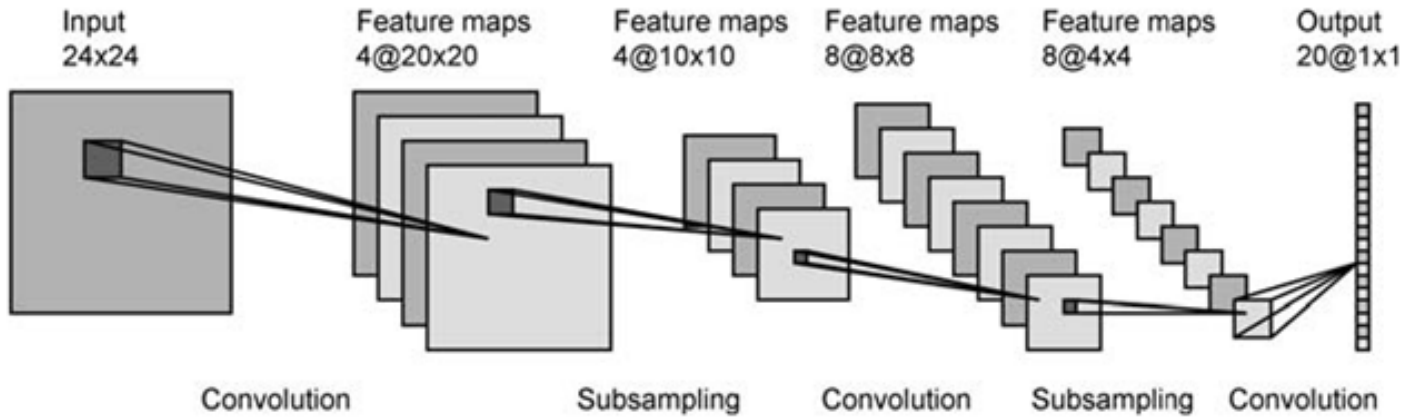


[1]

- Deconvolution
- Max un-pooling
- Tied encoder/decoder weights

[1] Image: Noh et. al.,Learning Deconvolution Network for Semantic Segmentation, ICCV 2015

Convolutional neural networks



Connectivity is designed in specific way (convolutional weight structure):

- Weights are **localized** in the input domain.
- Weights are **repeated** across the input domain.
- Weights have progressively **larger support**.
- **Pooling and subsampling** for robustness and reduced parameters.

Convolutional layers

$$\Phi : \mathcal{X} \to \mathcal{F}, \quad \Phi(x) = \sigma \circ W(x)$$

- representation by **filtering** $W : \mathcal{X} \to \mathcal{F}'$,
- representation by **pooling** $\sigma : \mathcal{F}' \to \mathcal{F}$.

Note: σ, W are more structured than in **(densely-connected)** NN.

Convolution and filtering

Weight matrix W is made of blocks

$$W = (G_{t_1}, \ldots, G_{t_T})$$

Each block is a *convolution matrix* of a single filter (template) t

$$G_t = (g_1 t, \ldots, g_N t)$$

$\{g_i\}$ is a transformation, e.g. circular shift, shift, ...

$$G_t = \begin{bmatrix} t^1 & t^2 & t^3 & \ldots & t^d \\ t^d & t^1 & t^2 & \ldots & t^{d-1} \\ t^{d-1} & t^d & t^1 & \ldots & t^{d-2} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ t^2 & t^3 & t^4 & \ldots & t^1 \end{bmatrix} \qquad G_t = \begin{bmatrix} t^1 & t^2 & t^3 & \ldots & 0 & 0 & 0 \\ 0 & t^1 & t^2 & \ldots & 0 & 0 & 0 \\ 0 & 0 & t^1 & \ldots & 0 & 0 & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & 0 & \ldots & t^1 & t^2 & t^3 \end{bmatrix}$$

Convolution and filtering

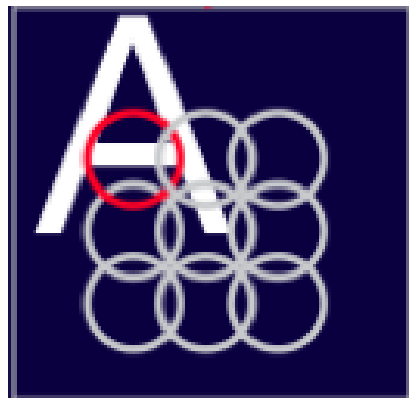
Weight matrix W is made of blocks:

$$W = (G_{t_1}, \ldots, G_{t_T}), \quad G_{t_i} = (g_1 t_i, \ldots, g_N t_i)$$

For all $x \in \mathcal{X}$,

$$(Wx)_{(j,i)} = x^\top g_i t_j, \quad Wx = ((t_1 \star x), \ldots, (t_T \star x)),$$

where $t_j \star x = (x^\top g_i t_j)$ is the **convolution operator**.

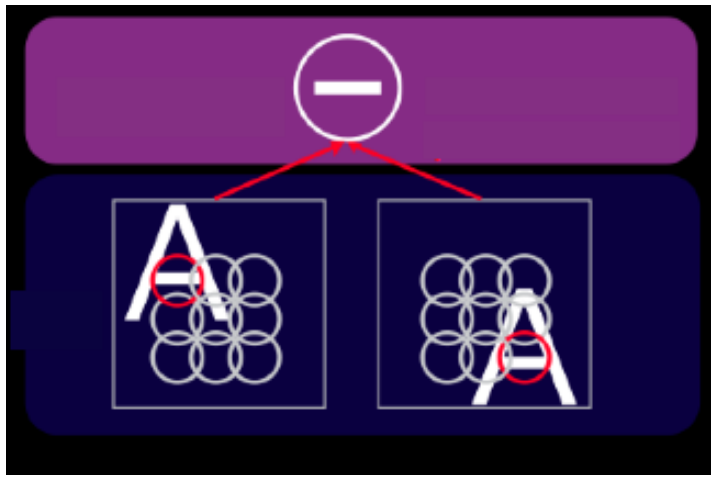


Note: In standard (densely-connected) neural nets $Wx = x^\top t_1, \ldots, x^\top t_T$

Pooling

A **Pooling** map **aggregates** the values corresponding to same filter

$$x \star t = x^\top g_1 t, \ldots, x^\top g_N t,$$



Can be followed by (or seen as a form of) **subsampling**.

Pooling functions

For some nonlinear activation σ, e.g. $\sigma(\cdot) = |\cdot|_+$, let

$$\beta = \sigma(x \star t) = \left(\sigma(x^\top g_1 t), \ldots, \sigma(x^\top g_N t)\right).$$

Examples

- max pooling

$$\max_{j=1,\ldots,N} \beta^j,$$

- average pooling

$$\frac{1}{N}\sum_{j=1}^{N} \beta^j,$$

- ℓ_p pooling

$$\|\beta\|_p = \left(\sum_{j=1}^{N} |\beta^j|^p\right)^{\frac{1}{p}}.$$

Why pooling?

Pooling can provide robustness, even **invariance** to the transformations.

- Filtering: covariant map
- Pooling: invariant map

Invariance & selectivity

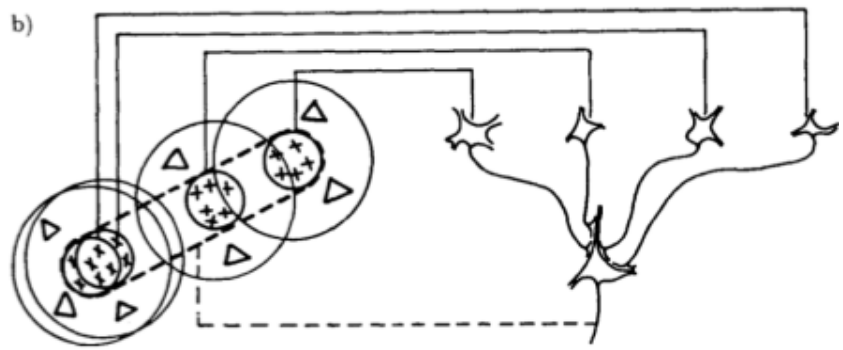
A good representation should be

- **invariant** to **semantically irrelevant** transformations.
- **discriminative** with respect to **semantically relevant** information.

Basic computations

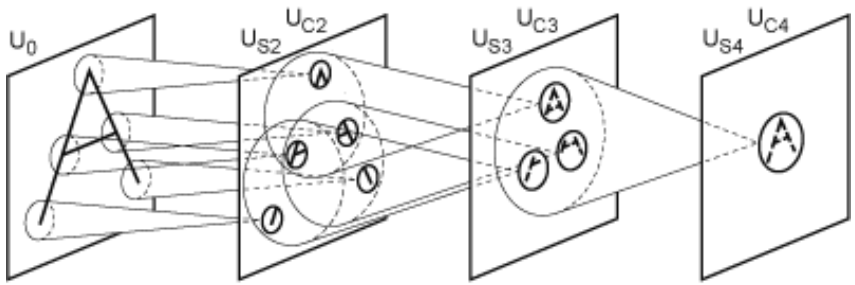
V1 in visual cortex: [Hubel and Wiesel]

- Simple cells: $\{\langle x, gt \rangle\}$
- Complex cells: $\sum_g \sigma\left(\langle x, gt \rangle\right)$

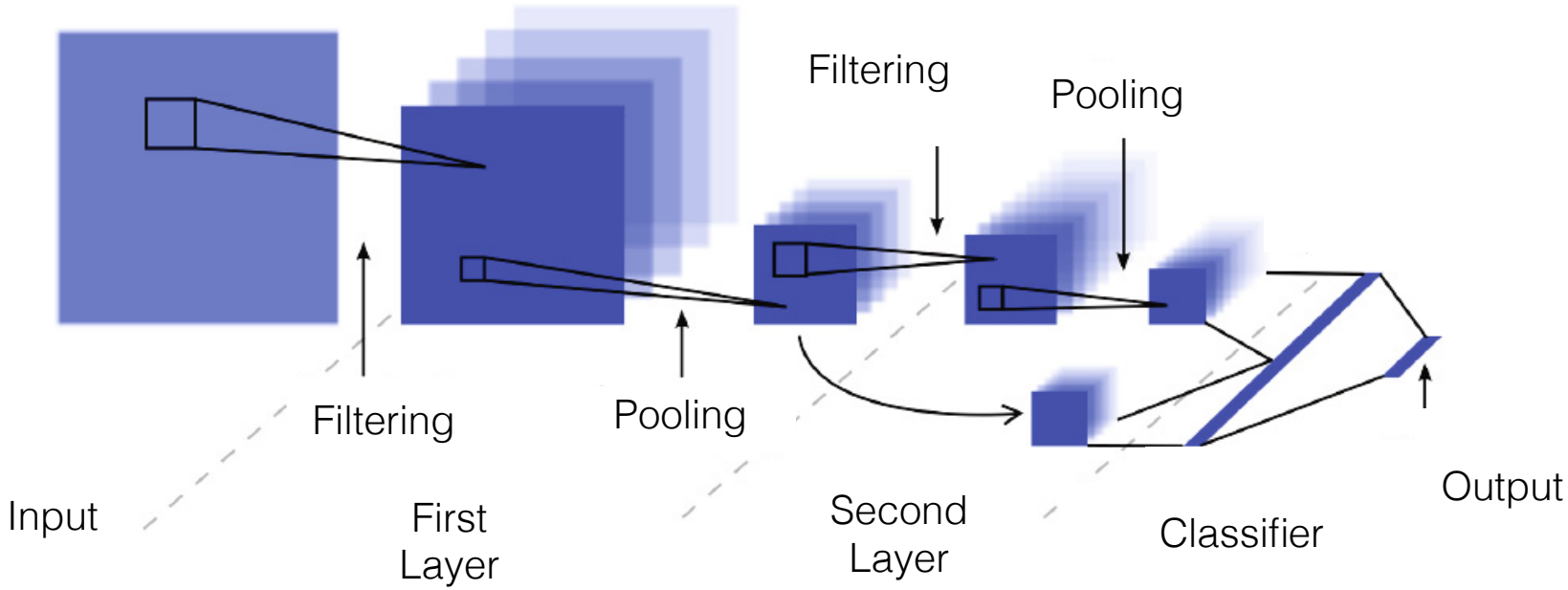


Convolutional networks: [Fukushima, LeCun, Poggio]

- Convolution filters: $\{\langle x, gt \rangle\}$
- Pooling: $\sum_g \sigma\left(\langle x, gt \rangle\right)$



Deep convolutional networks



In practice:

- multiple convolution layers are **stacked**,
- pooling is not global, but over a local subset of the transformations,
- filter size (receptive field) increases by **layers**.

Theory

$$\Phi_L(x) = \sigma(W_L \dots \sigma(W_2(\sigma(W_1 x)))$$

- **No pooling**: metric properties of networks with random weights – connection with compressed sensing [Giryes et al. '15]
- **Invariance**

$$x' = gx \Rightarrow \Phi(x') = \Phi(x)$$

[Anselmi et al. '12, R. Poggio '15, Mallat '12, Soatto, Chiuso '13] and covariance for multiple layers [Anselmi et al. '12].

- **Selectivity/Maximal Invariance**, i.e. injectivity modulo transformations

$$\Phi(x') = \Phi(x) \Rightarrow x' = gx$$

[R. Poggio '15, Soatto, Chiuso '15]

Theory (cont.)

- **Similarity** preservation

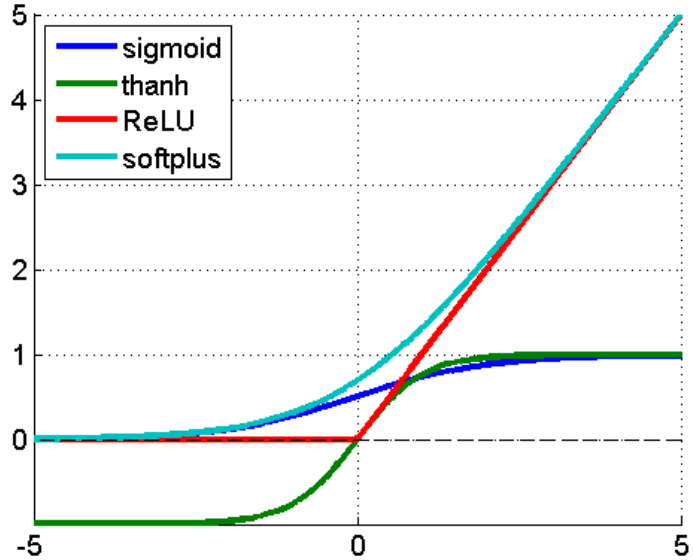
$$\|\Phi(x') - \Phi(x)\| \asymp \min_g \|x' - gx\| ???$$

- Stability to **diffeomorphisms** [Mallat, '12]

$$\|\Phi(x) - \Phi(d(x))\| \lesssim \|d\|_\infty \|x\|$$

- **Reconstruction**: connection to phase retrieval/one bit compressed sensing [Bruna et al '14].
- **Weight sharing**: fewer parameters to learn!

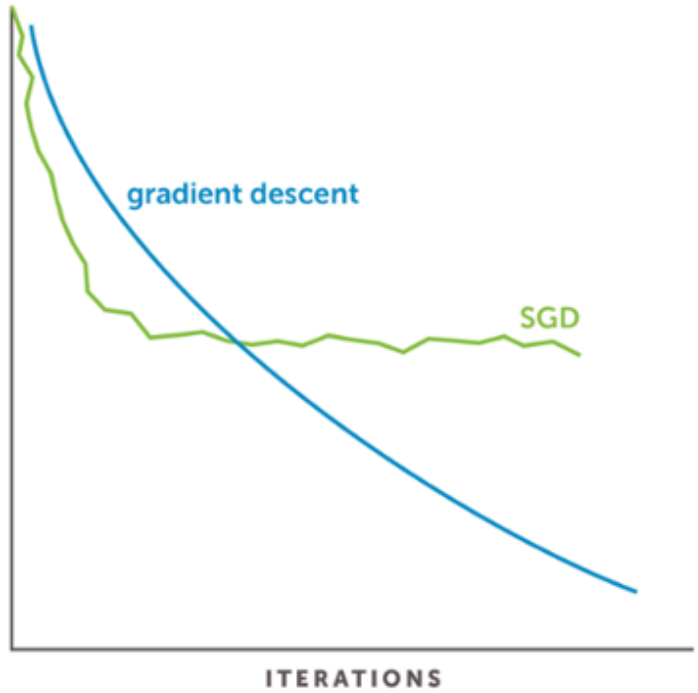
Which activation function?



- Biological motivation
- Rich function spaces
- Avoid vanishing gradient
- Fast gradient computation

ReLU: Has the last two properties. Work best in practice!

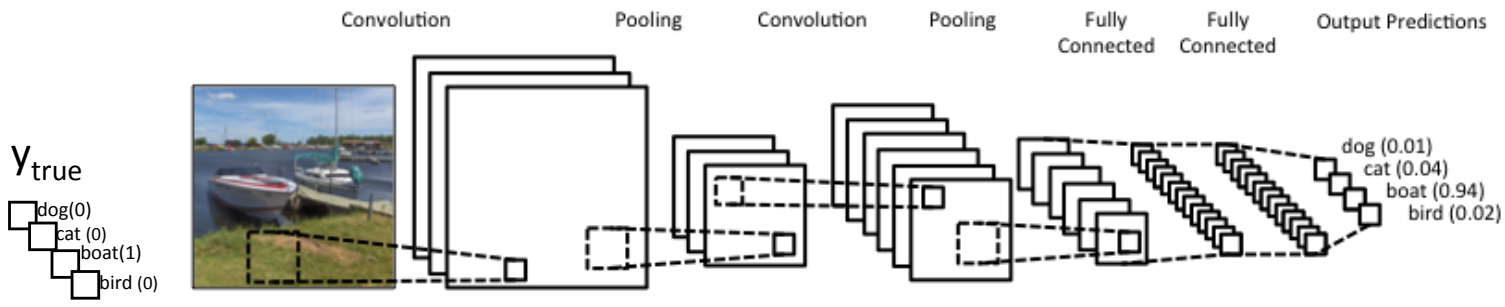
SGD is slow...



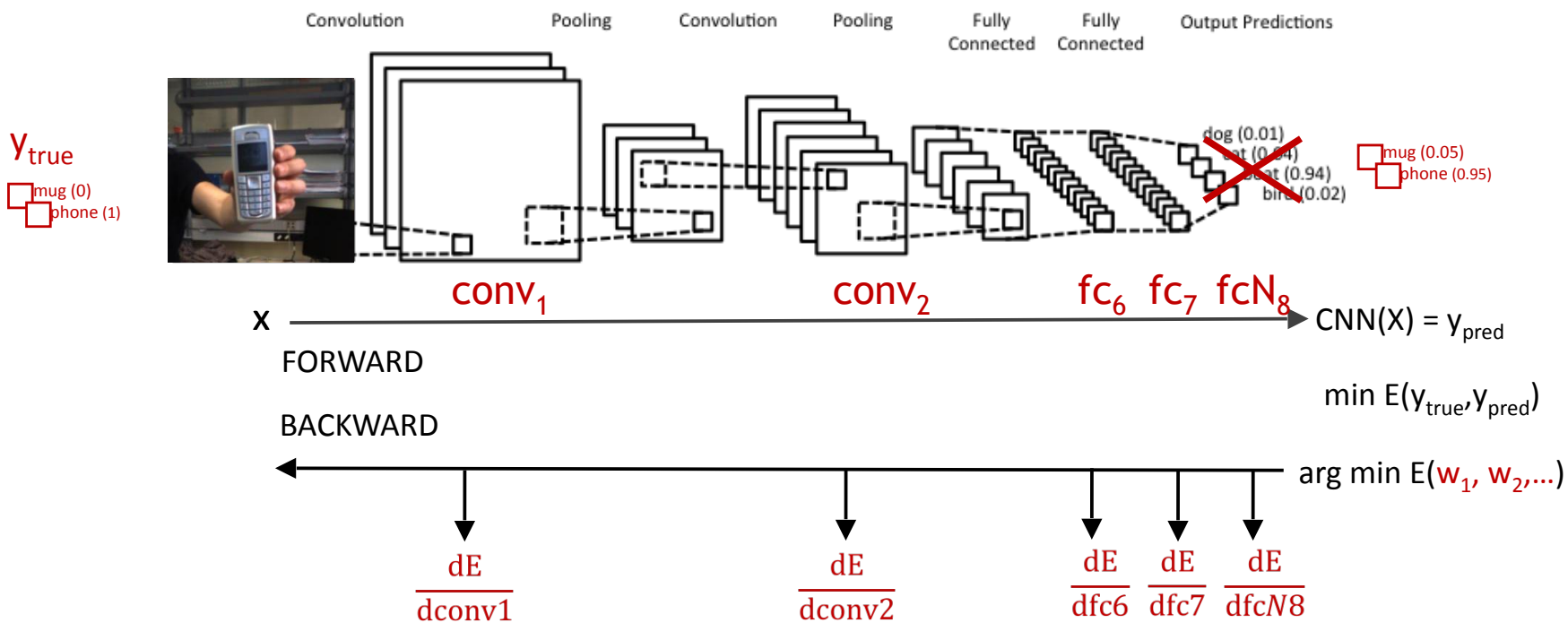
Accelerations

- **Momentum**
- Nesterov's method
- Adam
- Adagrad
- ...

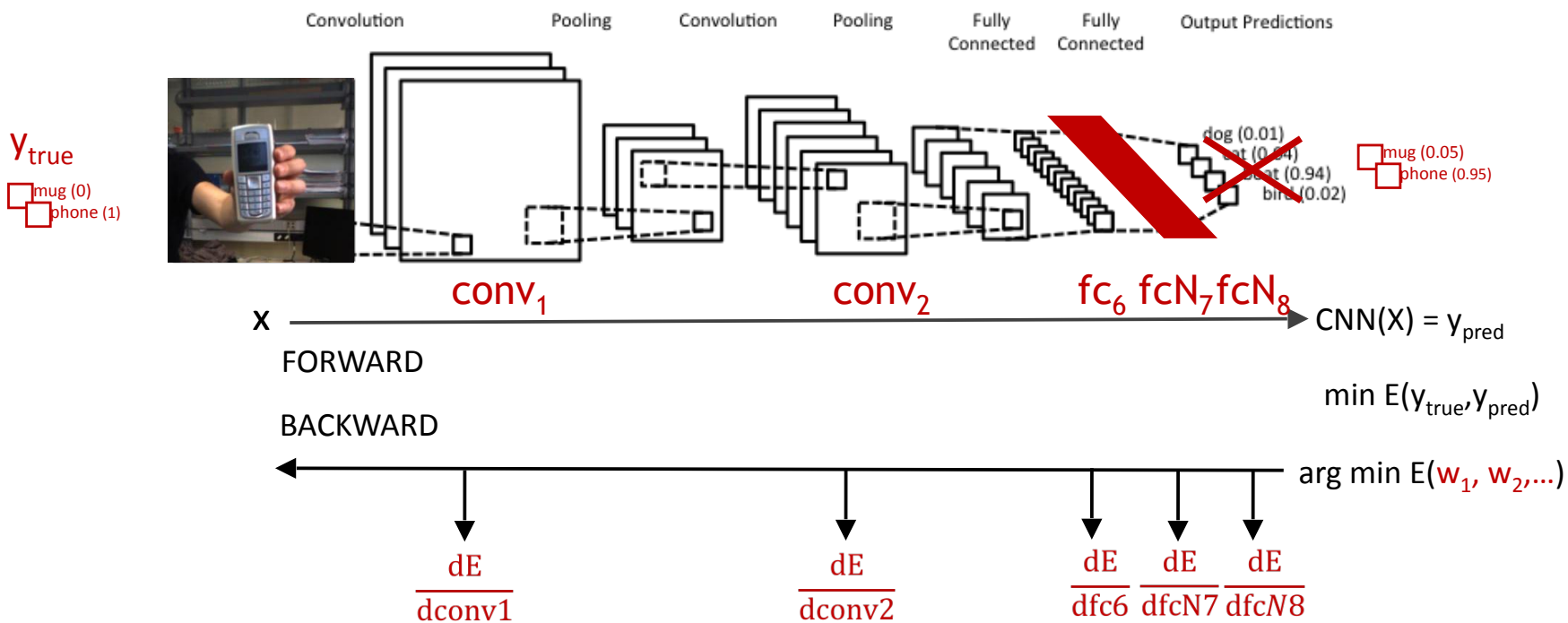
Initialization & fine tuning



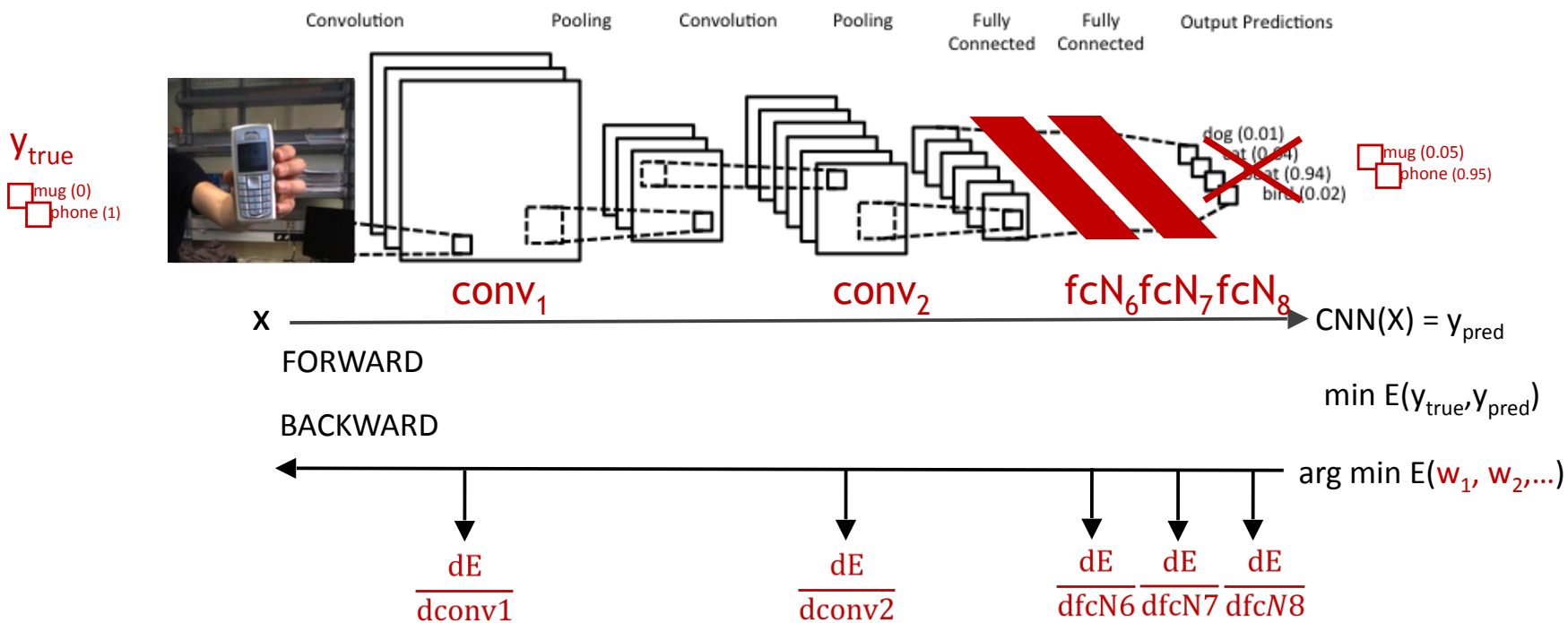
Initialization & fine tuning



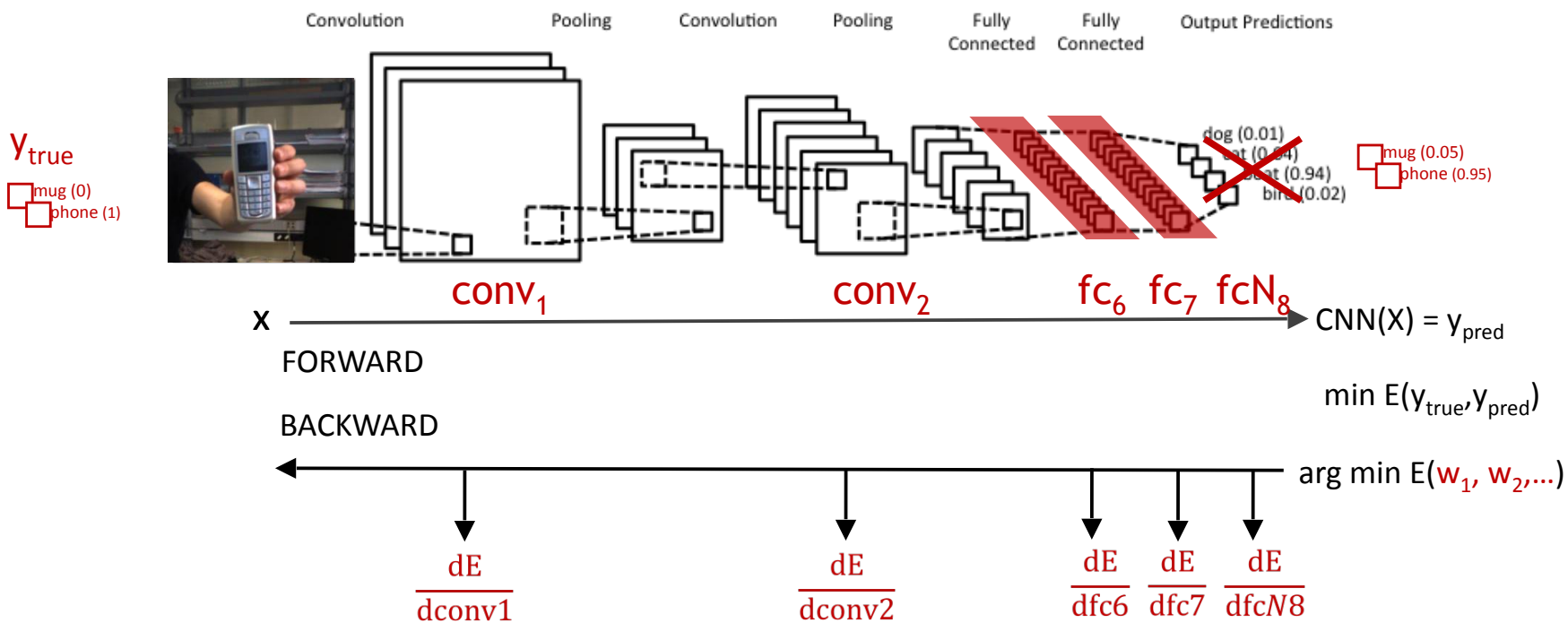
Initialization & fine tuning



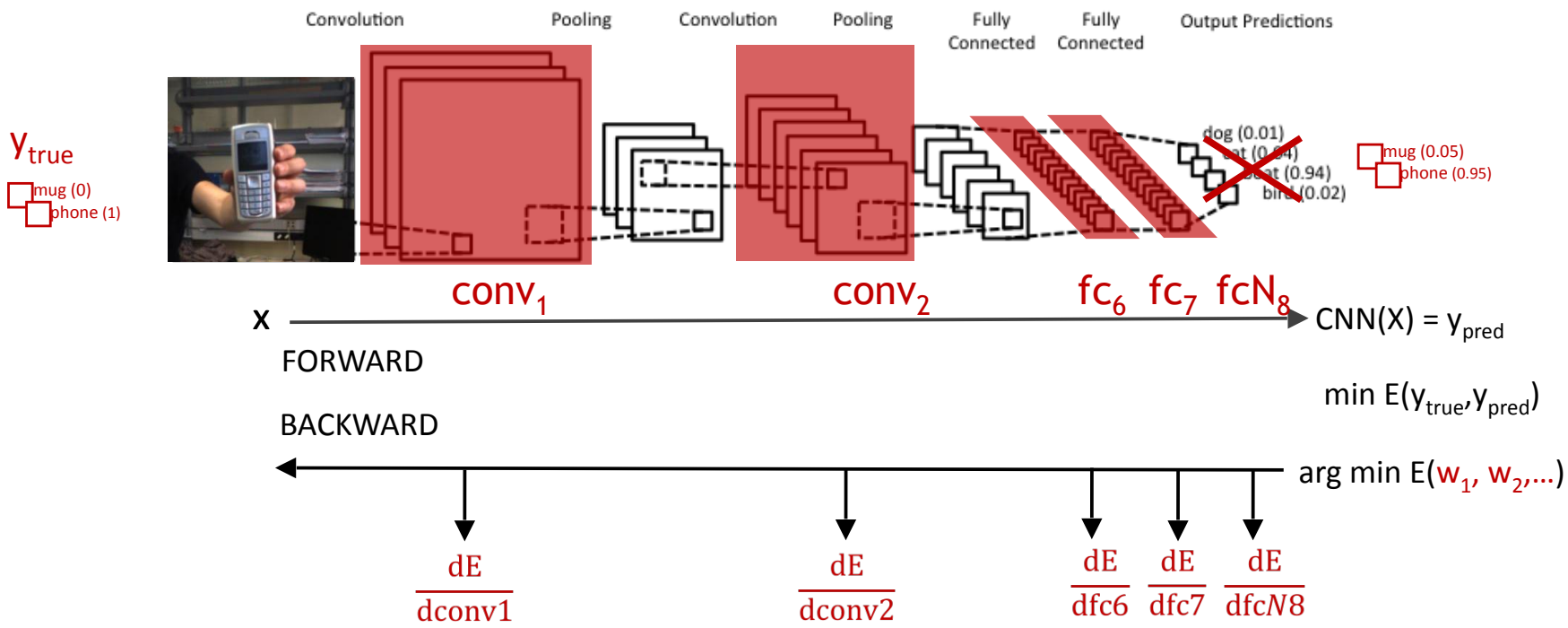
Initialization & fine tuning



Initialization & fine tuning



Initialization & fine tuning



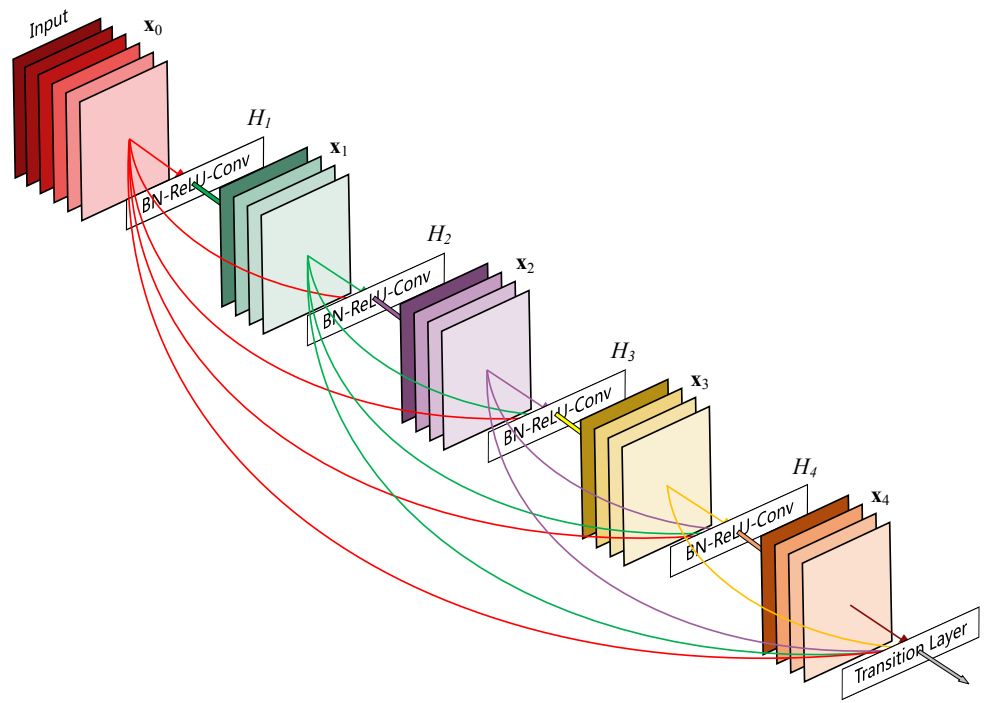
- Learning layers from scratch/from pre-learned initialization
- Learning layers more/less aggressively using different step-sizes

Training protocol(s)

- Learning at different layers
  - Initialization
  - Learning rates
- Mini-batch size
- Further aspect: regularization!
  - Weight constraints
  - Drop-out
- Batch normalization
- Data augmentation
- ...

Advances in architectures, state-of-the-art

- Supervision (AlexNet)
- GoogLeNet (Inception)
- Batch normalization (BN-Inception)
- Residual networks (ResNet)
- Dense networks (DenseNet)[2]



[2]Image: Huang et. al., Densely Connected Convolutional Networks, CVPR 2017

Wrap-up/Remarks

This class:

- Learning representations with deep networks
- Learning deep networks
- Unsupervised: Autoencoders
- Supervised: CNNs
- Convolutions as a strong prior/regularization

Other architectures:

- GANs, Recurrent NNs/LSTMs, ...

Next classes:

- Approximation
- Optimization
- Generalization/overfitting
- Deep learning, CNNs and visual cortex