EECS 553 Lecture 13
Machine Learning (ECE)
[class version]

Jeff Fessler

University of Michigan
©Laura Balzano and Jeff Fessler, UM, October 12, 2026, 23:52

F26

- Exam1 - WIP
- HW5 due Fri. Oct 23; mid-semester feedback
- PA3: Fri. Nov. 6: draft report. Must use Overleaf template. See Canvas.
- Project papers feedback: nice topics!
  Teams 4, 16, 18 and A100 GPUs: let's talk after class.
- Correction:
  - **homogeneous polynomial**: $x_1^5 + 7x_1^3x_2^2 + 9x_1x_1^4$ (degree 5)
    For degree p and feature dimension d, **needs**: $m = \binom{d+p}{p} = \binom{d+p}{d}$ coefficients (including constant term).
    Previous example with $b + w_1x + w_2x^2 + w_3x^3$ was $d = 1$ and $p = 3$ so $m = \binom{d+p}{p} = \binom{4}{3} = 4 = 3 + 1$.
  - **inhomogeneous polynomial**: $c + x_1 + x_1^2 + x_1^3$ (degree 3)
    For degree p and feature dimension d, **needs**:
    $m = \binom{p+d-1}{p} = \binom{p+d-1}{d-1}$ coefficients.

Recap / Outline

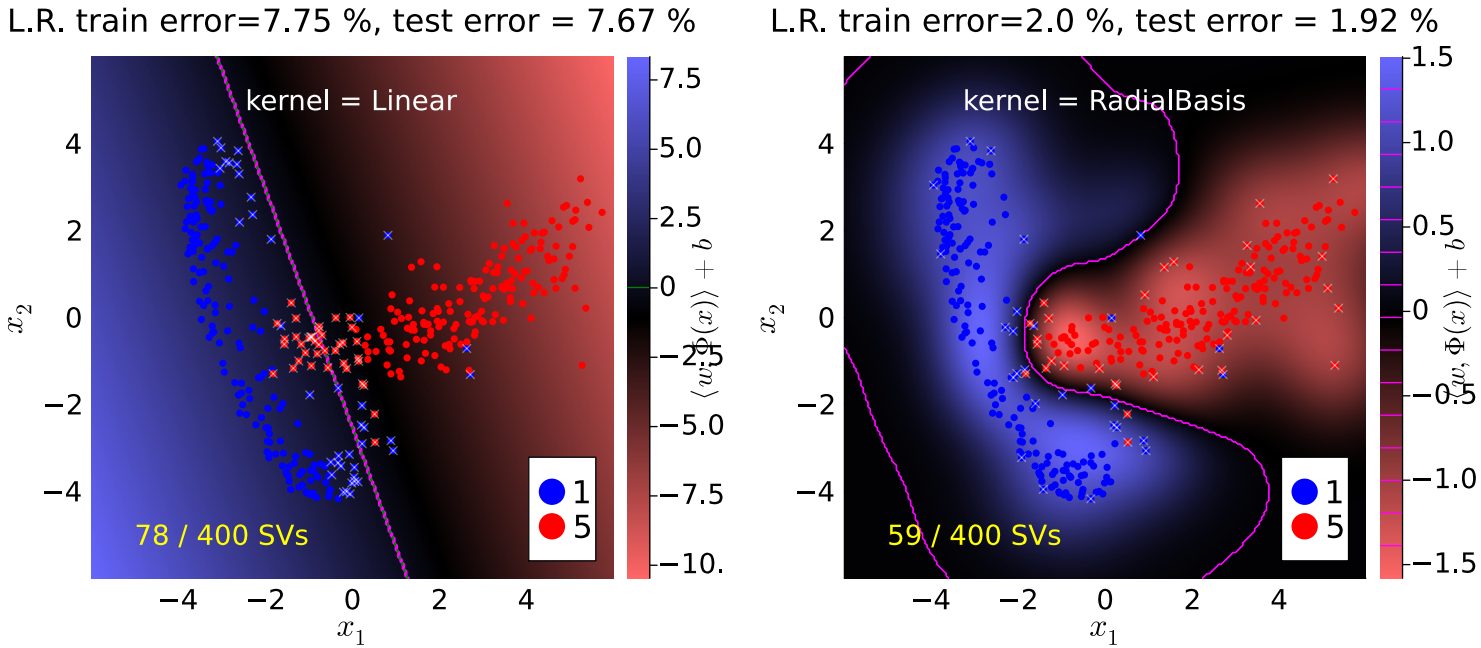
© 2026 Fessler

- Recap
  - Kernel methods
  - Kernel method for OSM, i.e., SVM
- Today

  - Kernel methods
    - Kernel recap
    - Kernel nearest neighbor classifier
    - Kernel nearest centroid classifier
    - Kernel ridge regression

  - Feed-forward neural networks / deep learning
    - Backpropagation

- We saw last time that nonlinear methods can improve classification when data has nonlinear structure.
- From demo:



- We applied the **kernel trick** (replacing inner products with $k(\cdot, \cdot)$ evaluations) to the (**kernelizable**) OSM method to get SVM.

Kernel methods: Feature map (recap)

© 2026 Fessler

- To create a nonlinear method for regression or classification, one way is to first transform each feature vector via a **nonlinear feature map**:

$$\boldsymbol{\Phi} : \mathbb{R}^d \mapsto \mathbb{R}^m,$$

  and then apply a linear method to the transformed data

$$\boldsymbol{\Phi}(\boldsymbol{x}_1), ..., \boldsymbol{\Phi}(\boldsymbol{x}_n).$$

- Because we typically have $m > d$, this is also called "lifting."

CQ: Nonlinear classifier: Recap

© 2026 Fessler

One approach to nonlinear regression/classification is to use a nonlinear function $\boldsymbol{\Phi}$.

Q13.1 What does a nonlinear binary classification function look like, using this approach?

A: $f(\boldsymbol{x}) = \boldsymbol{\Phi}(\boldsymbol{w}^\top \boldsymbol{x} + b)$

B: $f(\boldsymbol{x}) = \text{sign}\Big(\boldsymbol{\Phi}(\boldsymbol{w}^\top \boldsymbol{x}) + b\Big)$

C: $f(\boldsymbol{x}) = \text{sign}\Big(\boldsymbol{w}^\top \boldsymbol{\Phi}(\boldsymbol{x}) + b)\Big)$

D: $f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{\Phi}(\boldsymbol{x}) + b$

E: none of these

- Now we apply the kernel trick to other methods beyond OSM.

- Given training data $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)$, the **nearest neighbor classifier** assigns a test point $\boldsymbol{x}$ to the label

$$y_j \text{ where } j = \arg\min_{1 \le i \le n}$$

- More generally, a **k-nearest neighbor** (**k-NN**) classifier takes a majority "vote" among the k closest training instances.

- Suppose Φ is a feature map associated to an inner product/SPD kernel k, *i.e.*,

  for all $\boldsymbol{u}, \boldsymbol{v}$.
- Is the nearest neighbors method **kernelizable**?
- Depends on the norm.
  We focus hereafter on the **Euclidean norm**.

- To compute the nearest neighbors of a test point $\boldsymbol{x}$ we need

$$\|\boldsymbol{x} - \boldsymbol{x}_i\|_2^2 =$$

  which depends only on inner products of feature vectors $\{\boldsymbol{x}, \boldsymbol{x}_i\}$.
- When lifted we have

$$\|\boldsymbol{\Phi}(\boldsymbol{x}) - \boldsymbol{\Phi}(\boldsymbol{x}_i)\|_2^2 = \langle \boldsymbol{\Phi}(\boldsymbol{x}), \boldsymbol{\Phi}(\boldsymbol{x}) \rangle - 2 \langle \boldsymbol{\Phi}(\boldsymbol{x}), \boldsymbol{\Phi}(\boldsymbol{x}_i) \rangle + \langle \boldsymbol{\Phi}(\boldsymbol{x}_i), \boldsymbol{\Phi}(\boldsymbol{x}_i) \rangle$$

- So for any $\boldsymbol{x} \in \mathbb{R}^d$, we can easily find the **nearest neighbor** in the lifted space via:

$$j = \underset{1 \le i \le n}{\arg\min}$$

- The **nearest centroid classifier** assigns a test point to the class whose **centroid** is closest to that test point:

$$f(\boldsymbol{x}) = \underset{k \in \{1,\ldots,K\}}{\arg\min} \, \|\boldsymbol{x} - \hat{\boldsymbol{\mu}}_k\|_2^2, \quad \hat{\boldsymbol{\mu}}_k \triangleq \frac{1}{n_k} \sum_{i \,:\, y_i = k} \boldsymbol{x}_i.$$

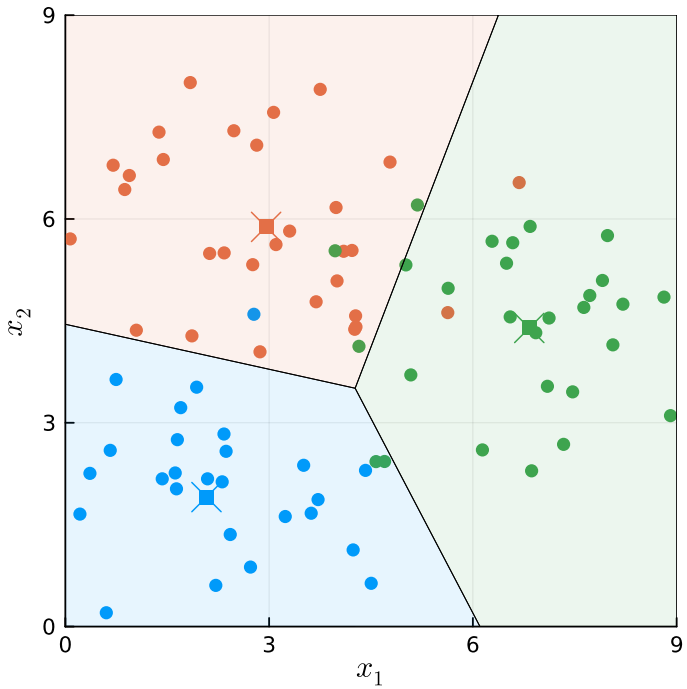
- The decision regions become **Voronoi cells**.

Q13.2 The **nearest centroid classifier** is

A: Parametric

B: Nonparametric

C: neither



- The compute here is just $O(dK)$, vs $O(dn)$ needed for 1-NN.
- For binary case with $\hat{\boldsymbol{\mu}}_{\pm 1} \triangleq \frac{1}{n_\pm} \sum_{i\,:\,y_i = \pm 1} \boldsymbol{x}_i$, the classifier is:

$$f(\boldsymbol{x}) = \operatorname{sign}\Big(\|\boldsymbol{x} - \hat{\boldsymbol{\mu}}_{-1}\|_2^2 - \|\boldsymbol{x} - \hat{\boldsymbol{\mu}}_{+1}\|_2^2 \Big).$$

- By expanding the Euclidean norm squared, this method is readily **kernelizable**:

$$f(\boldsymbol{x}) = \operatorname{sign}(\langle \hat{\boldsymbol{\mu}}_{+1} - \hat{\boldsymbol{\mu}}_{-1}, \boldsymbol{x} \rangle + b), \quad b = \langle \hat{\boldsymbol{\mu}}_{-1}, \hat{\boldsymbol{\mu}}_{-1} \rangle - \langle \hat{\boldsymbol{\mu}}_{+1}, \hat{\boldsymbol{\mu}}_{+1} \rangle$$

$$= \operatorname{sign}\left(\frac{2}{n_+} \sum_{i\,:\,y_i = +1} \underbrace{\langle \boldsymbol{x}_i, \boldsymbol{x} \rangle}_{\text{kernel}} - \frac{2}{n_-} \sum_{i\,:\,y_i = -1} \langle \boldsymbol{x}_i, \boldsymbol{x} \rangle + b \right)$$

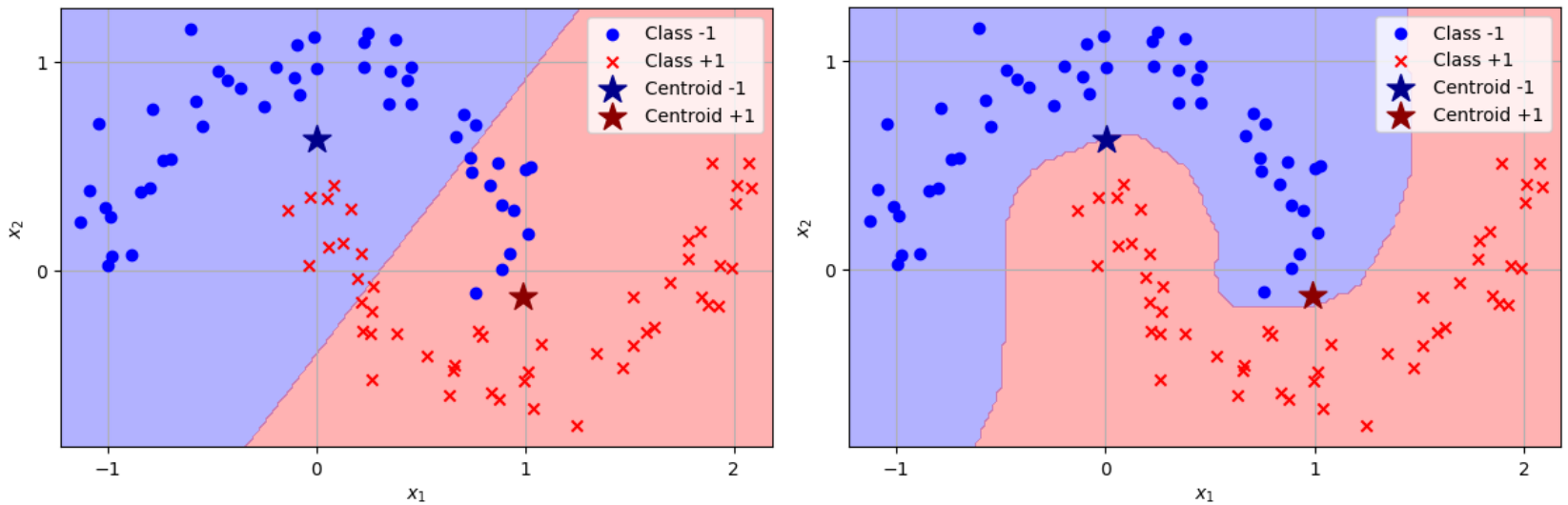
$$b = \frac{1}{n_+^2} \sum_{\substack{i\,:\,y_i = +1 \\ j\,:\,y_j = +1}} \underbrace{\langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle}_{\text{kernel}} - \frac{1}{n_-^2} \sum_{\substack{i\,:\,y_i = -1 \\ j\,:\,y_j = -1}} \langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle .$$

Demo: Kernel nearest centroid classifier

© 2026 Fessler

PYTHON colab demo

(Select different datasets, kernels, and bandwidths)



For this synthetic data, kernel nearest centroid works far better than ordinary nearest centroid, at the price of more memory and compute.

Q13.3 Kernel nearest centroid is

A: parametric B: nonparametric C: insufficient information

Kernel ridge regression: Overview

© 2026 Fessler

- We will show that **ridge regression** is **kernelizable**, and use the kernel trick to extend it to a nonlinear regression method called **kernel ridge regression** (**KRR**).
- KRR has numerous applications, even quantitative MRI [3].
- To simplify the presentation, we focus on ridge regression *without offset*:

$$\hat{f}(\boldsymbol{x}) = \widehat{\boldsymbol{w}}^\top \boldsymbol{x}, \quad \widehat{\boldsymbol{w}} =$$

- Because some $\boldsymbol{\Phi}$ choices contain a constant term, as with the inhomogeneous polynomial kernel, the offset is not always needed.
- KRR with offset is described in Prof. Scott's lecture notes.

For $\boldsymbol{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$, $\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_1^\top \\ \vdots \\ \boldsymbol{x}_n^\top \end{bmatrix}$,
the **ridge regression** (**RR**) training problem with $\lambda \geq 0$ is:

$$\widehat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \frac{1}{n} \sum_{i=1}^{n} (\boldsymbol{w}^\top \boldsymbol{x}_i - y_i)^2 + \lambda \|\boldsymbol{w}\|_2^2 .$$

Q13.4 The solution is:

A: $\widehat{\boldsymbol{w}} = (\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I})^{-1} \boldsymbol{X}^\top \boldsymbol{y}$

B: $\widehat{\boldsymbol{w}} = \boldsymbol{X}^\top (\boldsymbol{X} \boldsymbol{X}^\top + \lambda \boldsymbol{I})^{-1} \boldsymbol{y}$

C: $\widehat{\boldsymbol{w}} = (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{y}$

D: $\widehat{\boldsymbol{w}} = \boldsymbol{X}^\top (\boldsymbol{X} \boldsymbol{X}^\top)^{-1} \boldsymbol{y}$

E: none of these

Kernelizing ridge regression I

- Note $\boldsymbol{X}^\top \boldsymbol{X}$ is *not* the **Gram matrix** of the training data, *i.e.*,
- Idea: apply a special case of the **matrix inversion lemma** called the **push-through identity** [4, p. 41]:

$$(a\boldsymbol{I} + \boldsymbol{B}\boldsymbol{D})^{-1}\boldsymbol{B} = \boldsymbol{B}(a\boldsymbol{I} + \boldsymbol{D}\boldsymbol{B})^{-1},$$

  with $\boldsymbol{B} = \boldsymbol{X}^\top$ and $\boldsymbol{D} = \boldsymbol{X}$, to get Gram matrix $\boldsymbol{X}\boldsymbol{X}^\top$.
- After simplification, the RR predictor without offset is

$$\begin{aligned}\hat{f}(\boldsymbol{x}) = \widehat{\boldsymbol{w}}^\top \boldsymbol{x} &= \boldsymbol{y}^\top \boldsymbol{X}(\boldsymbol{X}^\top \boldsymbol{X} + n\lambda \boldsymbol{I})^{-1}\boldsymbol{x} \\ &= \boldsymbol{y}^\top (\boldsymbol{X}\boldsymbol{X}^\top + n\lambda \boldsymbol{I})^{-1}\boldsymbol{X}\boldsymbol{x}.\end{aligned}$$

Rewriting with inner products:

$$\begin{aligned}\hat{f}(\boldsymbol{x}) &= \boldsymbol{y}^\top (\boldsymbol{X}\boldsymbol{X}^\top + n\lambda \boldsymbol{I})^{-1} \boldsymbol{X}\boldsymbol{x} \\ &= \boldsymbol{y}^\top (\boldsymbol{K} + n\lambda \boldsymbol{I})^{-1} \boldsymbol{k}(\boldsymbol{x})\end{aligned}$$

where the **kernel matrix** (Gram matrix) is

$$\boldsymbol{K} \triangleq \begin{bmatrix} \langle \boldsymbol{x}_1, \boldsymbol{x}_1 \rangle & \cdots & \langle \boldsymbol{x}_1, \boldsymbol{x}_n \rangle \\ \vdots & \ddots & \vdots \\ \langle \boldsymbol{x}_n, \boldsymbol{x}_1 \rangle & \cdots & \langle \boldsymbol{x}_n, \boldsymbol{x}_n \rangle \end{bmatrix}, \qquad \boldsymbol{k}(\boldsymbol{x}) = \begin{bmatrix} \langle \boldsymbol{x}_1, \boldsymbol{x} \rangle \\ \vdots \\ \langle \boldsymbol{x}_n, \boldsymbol{x} \rangle \end{bmatrix}.$$

To save test-time compute, precompute and store the vector

CQ: Kernel ridge regression

© 2026 Fessler

To define the KRR method, we simply replace all the inner products above (in both $\boldsymbol{K}$ and $\boldsymbol{k}$) with kernel values $k(\cdot, \cdot)$.

Q13.5 If k is such that $k(\boldsymbol{u}, \boldsymbol{v}) = \langle \boldsymbol{\Phi}(\boldsymbol{u}), \boldsymbol{\Phi}(\boldsymbol{v}) \rangle$, then (ignoring offset b) the above KRR method is equivalent to

A: Apply ridge regression to training data $\{(\boldsymbol{x}_i, y_i)\}$, and then transform the solution into a nonlinear space with $\boldsymbol{\Phi}$.

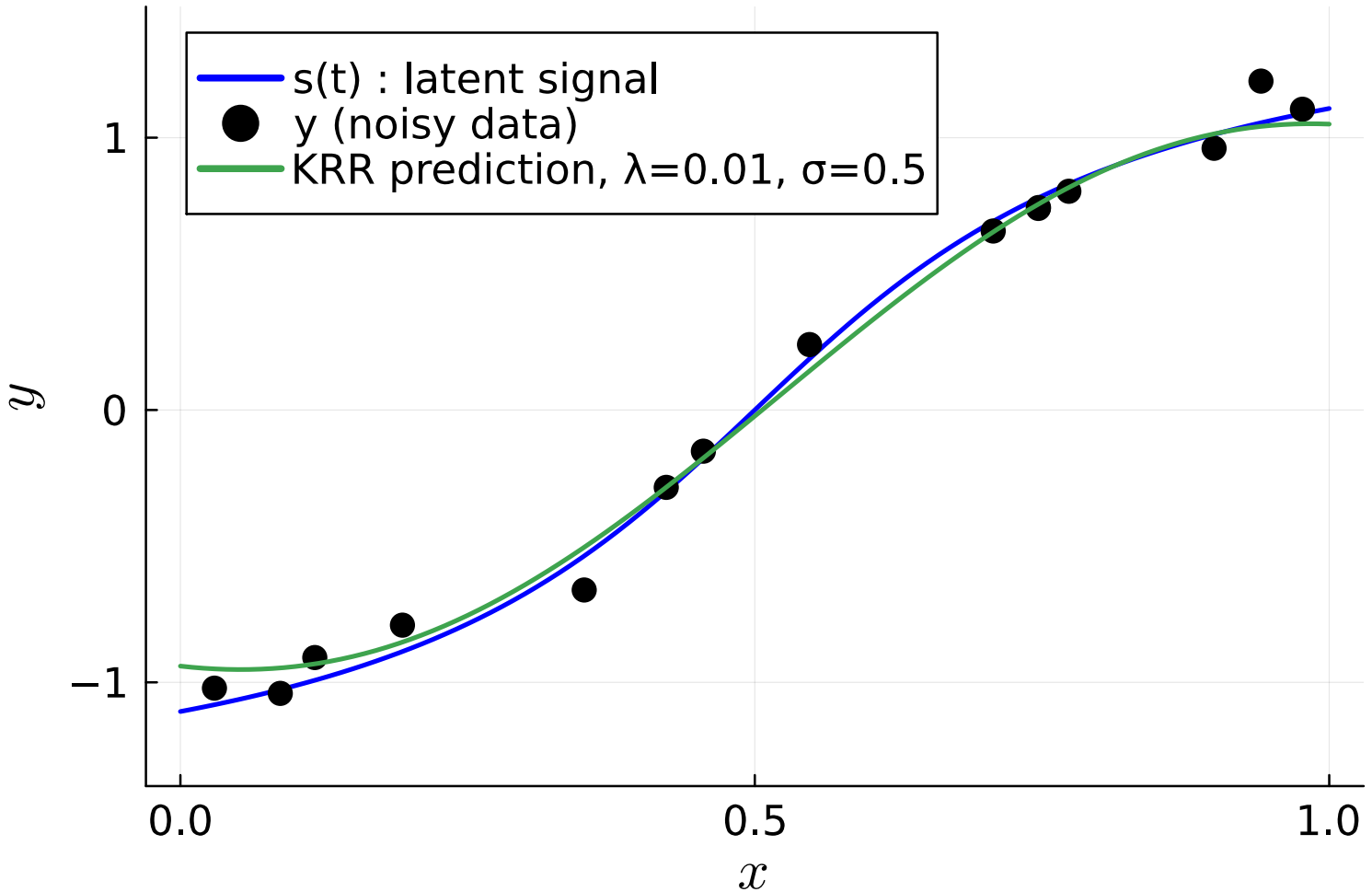
B: Apply a nonlinear $\boldsymbol{\Phi}$ to the ridge regularization term.

C: Apply the feature map $\boldsymbol{\Phi}$ to all feature vectors, and then apply ridge regression in the new feature space.

D: Apply the feature map $\boldsymbol{\Phi}$ to all training data, and then apply ridge regression to the transformed data.

E: none of these

JULIA KRR demo with hyperparameters λ and Gaussian kernel parameter σ hand selected.



Theorem 1 (Nonparametric Representer Theorem). *Suppose we are given a nonempty set $\mathcal{X}$, a positive definite real-valued kernel k on $\mathcal{X} \times \mathcal{X}$, a training sample $(x_1, y_1), \ldots, (x_m, y_m) \in \mathcal{X} \times \mathbf{R}$, a strictly monotonically increasing real-valued function g on $[0, \infty[$, an arbitrary cost function $c : (\mathcal{X} \times \mathbf{R}^2)^m \to \mathbf{R} \cup \{\infty\}$, and a class of functions*

$$\mathcal{F} = \left\{ f \in \mathbf{R}^{\mathcal{X}} \,\middle|\, f(\cdot) = \sum_{i=1}^{\infty} \beta_i k(\cdot, z_i), \beta_i \in \mathbf{R}, z_i \in \mathcal{X}, \|f\| < \infty \right\}. \tag{13}$$

Here, $\|\cdot\|$ is the norm in the RKHS H_k associated with k, i.e. for any $z_i \in \mathcal{X}, \beta_i \in \mathbf{R}$ ($i \in \mathbf{N}$),

$$\left\| \sum_{i=1}^{\infty} \beta_i k(\cdot, z_i) \right\|^2 = \sum_{i=1}^{\infty} \sum_{j=1}^{\infty} \beta_i \beta_j k(z_i, z_j). \tag{14}$$

Then any $f \in \mathcal{F}$ minimizing the regularized risk functional

$$c\left((x_1, y_1, f(x_1)), \ldots, (x_m, y_m, f(x_m))\right) + g\left(\|f\|\right) \tag{15}$$

admits a representation of the form

$$f(\cdot) = \sum_{i=1}^{m} \alpha_i k(\cdot, x_i). \tag{16}$$

[6]

- Neural networks are a class of ML models where the **prediction function** is a composition of many "layers" of functions.
- Typically trained with surrogate loss in ERM framework:

$$\text{Empirical Risk:} \quad = \frac{1}{n} \sum_{i=1}^{n} L(y_i, \underbrace{f_\theta(\boldsymbol{x}_i)}).$$

- For many problems (having sufficient training data), NNs achieve state-of-the-art performance, *e.g.*, generative AI, object recognition, speech recognition, language translation
- Today: **feedforward neural networks** where information flows in a single direction (the network is a **directed acyclic graph**).

© 2026 Fessler

All answers accepted:

Q13.6 How familiar are you with neural networks?

A: I haven't heard of them

B: I've heard a lot about them

C: I've used them on a project

D: I'm familiar with their theory

E: I'm familiar in both theory and practice

Feedforward networks: Overview

© 2026 Fessler

- A **neural network** (**NN**) (or artificial NN) is a **prediction function** loosely inspired by **biological neural networks**.
- The network generally has linear computation between **layers** of the network and nonlinearities at nodes.
- Feedforward networks do not have any **feedback** connections.

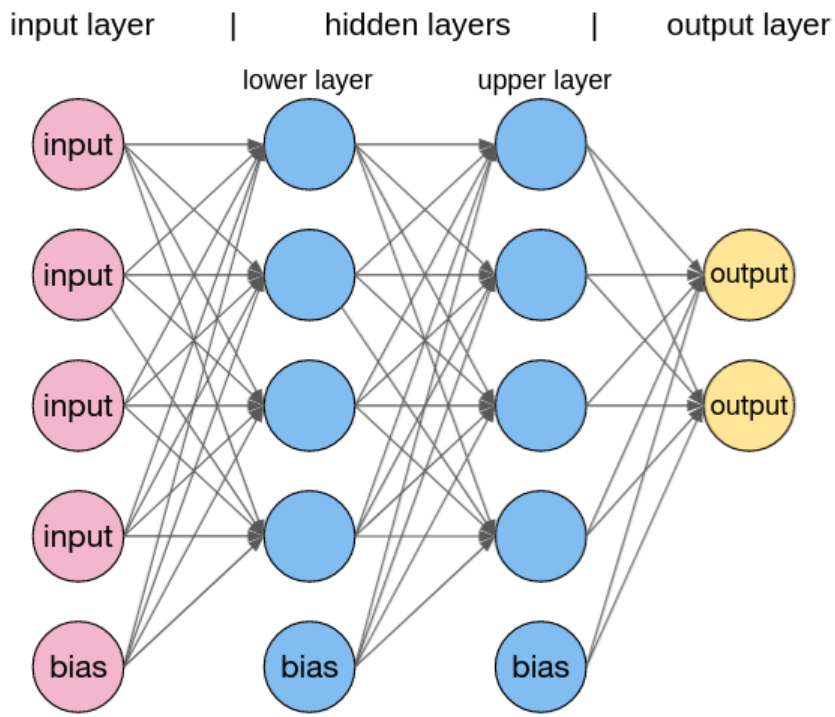


Figure courtesy [7].

- Feedforward networks *can* have connections, *e.g.*, **ResNet**.

For data $(\boldsymbol{x}, \boldsymbol{y}) \in \mathbb{R}^d \times \mathbb{R}^k$, one such prediction function is:

$$\hat{\boldsymbol{y}} = \sigma_3\left(\boldsymbol{W}^{(3)}\sigma_2\left(\boldsymbol{W}^{(2)}\sigma_1\left(\boldsymbol{W}^{(1)}\boldsymbol{x}\right)\right)\right)$$

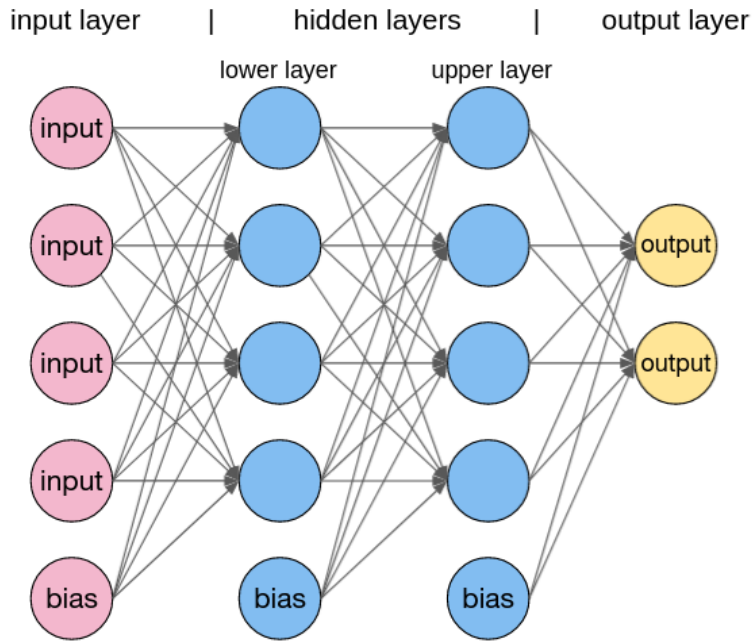


Figure courtesy [7].

Users/designers must specify:

- Number of hidden layers
- Number of nodes per layer
- Nonlinear activation functions
- Whether or not to include bias at each layer
- Whether the layers are fully connected

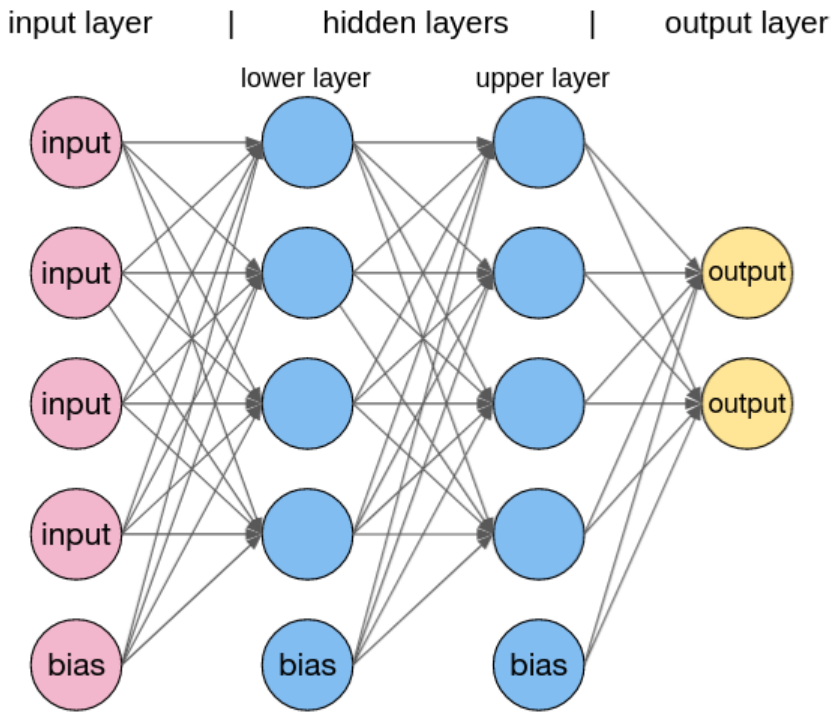


Figure courtesy [7].

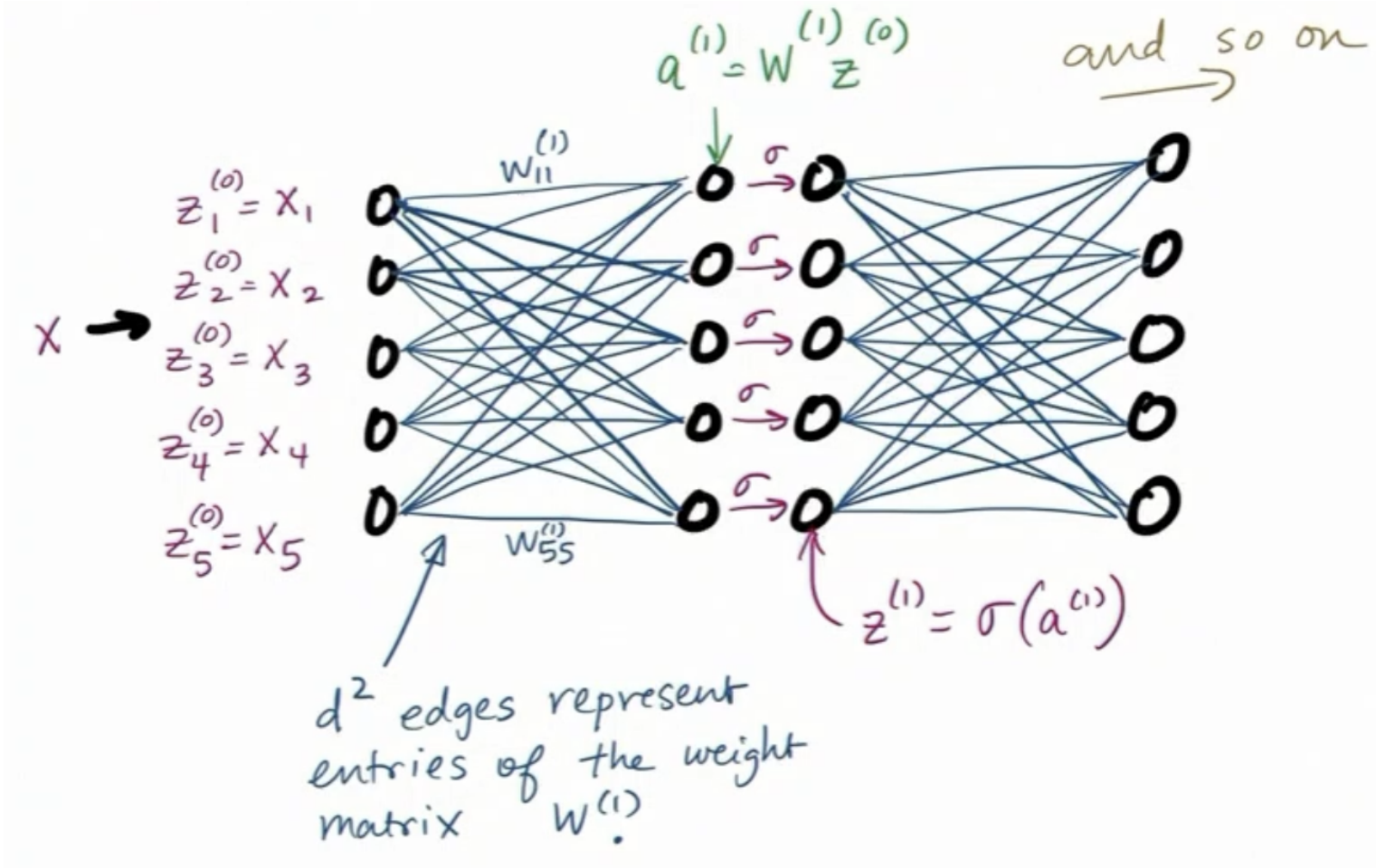
Feedforward networks are also called

- Input layer:
- Hidden layer $1 \le l < L$:

$$\boldsymbol{a}^{(l)} =$$

$$\boldsymbol{z}^{(l)} = .$$

- If bias is desired, prepend a 1 to any $\boldsymbol{z}^{(l)}$, $0 \le l < L$, and add left column of weights to $\boldsymbol{W}^{(l+1)}$
- Output layer: $\hat{\boldsymbol{y}} = f(\boldsymbol{x}) = \boldsymbol{z}^{(L)} =$ namely when
- Evaluation of the output $\hat{\boldsymbol{y}} = f(\boldsymbol{x})$ from an input $\boldsymbol{x}$ is called **forward propagation**.



- Typically $\boldsymbol{W}^{(1)}$ is
- In JULIA, use broadcaset for element-wise application:

```
z1 = sigma.(a1)
```

- K classes
- Represent classes by indicator vectors (**one-hot** encoding):

$$\begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \quad \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

- Last layer $\boldsymbol{z}^{(L)} = \psi(\boldsymbol{a}^{(L)})$ transforms $\boldsymbol{a}^{(L)} \in \mathbb{R}^K$ to a probability vector on the **unit simplex** Δ^{K-1} in $\mathbb{R}^K$, using the **softmax function**:

$$\psi : \mathbb{R}^K \mapsto \Delta^{K-1} \subset \mathbb{R}^K, \quad \psi_k(\boldsymbol{a}) = \frac{\mathrm{e}^{a_k}}{\sum_{j=1}^K \mathrm{e}^{a_j}}, \; k = 1, \ldots, K.$$

- Given training data $(\boldsymbol{x}_1, \boldsymbol{y}_1), \ldots, (\boldsymbol{x}_n, \boldsymbol{y}_n)$, need to learn the network parameters aka **weights**

$$\boldsymbol{\theta} = (\boldsymbol{W}^{(1)}, \ldots, \boldsymbol{W}^{(L)})$$

- Typical loss functons (ERM)
  - Squared error loss (regression):

$$R(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \| f_{\boldsymbol{\theta}}(\boldsymbol{x}_i) - \boldsymbol{y}_i \|_2^2 .$$

  - **Cross-entropy** loss (for classification) with $\sigma_L =$ **softmax** ψ:

$$R(\boldsymbol{\theta}) = -\frac{1}{n} \sum_{i=1}^{n} \sum_{k=1}^{K} \underbrace{y_{ik}}_{\text{one-hot}} \log[f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)]_k .$$

    Leads to faster training and better generalization than squared error for classification problems [5, p. 235].

Training the NN weights: Optimization

© 2026 Fessler

- Invariances and **data augmentation**
- We typically apply some version of **SGD** to $R(\boldsymbol{\theta})$, *e.g.*, **Adam** (adaptive moment estimation) [9]
- Selecting learning rates (step sizes) is an art...
- Improving optimization methods is an active research area, *e.g.*, Muon [10] and Scion [11].

TL;DR version: **backpropagation** is an efficient means computing gradients using the chain rule [1, p. 434].

- Tweak notation i vs n $R(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N}$
  R_n: loss associated with training sample $\boldsymbol{x}_n, \; n = 1, \ldots, N$
- Focus on computing
- An efficient algorithm for computing this gradient is called **backpropagation**
- More notation:

Symbol	meaning
$w_{ij}^{(l)}$	(i,j) entry of $\boldsymbol{W}^{(l)}$
$a_{ni}^{(l)}$	ith entry of $\boldsymbol{a}^{(l)}$ when input is $\boldsymbol{x}_n$
$z_{ni}^{(l)}$	ith entry of $\boldsymbol{z}^{(l)}$ when input is $\boldsymbol{x}_n$

- From definitions:

$$\boldsymbol{a}^{(l)} = \boldsymbol{W}^{(l)} \boldsymbol{z}^{(l-1)} \implies a_{ni}^{(l)} = \sum_j w_{ij}^{(l)} z_{nj}^{(l-1)}$$

$$\boldsymbol{z}^{(l)} = \sigma_l(\boldsymbol{a}^{(l)}) \implies z_{ni}^{(l)} = \sigma_l(a_{ni}^{(l)}) \text{ (element-wise)} .$$

- These quantities are known (at current value of $\boldsymbol{\theta}$) via **forward propagation** starting from $l = 1$ and working forward to $l = L$.

- $\nabla R_n(\boldsymbol{\theta})$ consists of elements

- By the **chain rule** for function composition:

$$\frac{\partial R_n(\boldsymbol{\theta})}{\partial w_{ij}^{(l)}} = \frac{\partial R_n(\boldsymbol{\theta})}{\partial a_{ni}^{(l)}} \frac{\partial a_{ni}^{(l)}}{\partial w_{ij}^{(l)}} = \delta_{ni}^{(l)} z_{nj}^{(l-1)},$$

where

$$\delta_{ni}^{(l)} \triangleq \frac{\partial R_n(\boldsymbol{\theta})}{\partial a_{ni}^{(l)}}.$$

The main idea behind the **backpropagation** algorithm is to compute $\delta_{ni}^{(L)}$ starting at the *output* layer, $l = L$ and to work *backwards* toward the input layer.

Q13.7 By the chain rule for partial derivatives,

$$\delta_{ni}^{(l)} = \frac{\partial R_n(\boldsymbol{\theta})}{\partial a_{ni}^{(l)}} = \quad ?$$

Below, k indexes the nodes in layer $l+1$.

A: $\sum_k \frac{\partial R_n(\boldsymbol{\theta})}{\partial a_{nk}^{(l)}}$

B: $\sum_k \frac{\partial R_n(\boldsymbol{\theta})}{\partial a_{nk}^{(l+1)}} \frac{\partial a_{nk}^{(l+1)}}{\partial a_{ni}^{(l)}}$

C: $\sum_k \frac{\partial R_n(\boldsymbol{\theta})}{\partial a_{nk}^{(l+1)}} \frac{\partial a_{nk}^{(l)}}{\partial a_{ni}^{(l+1)}}$

D: none of these

- Applying the definition of $\delta_{nk}^{(l)}$ to the preceding answer at $l+1$:

$$\delta_{ni}^{(l)} = \frac{\partial R_n(\boldsymbol{\theta})}{\partial a_{ni}^{(l)}} = \sum_k \delta_{nk}^{(l+1)} \frac{\partial a_{nk}^{(l+1)}}{\partial a_{ni}^{(l)}}.$$

- Because

$$a_{nk}^{(l+1)} = \sum_j w_{kj}^{(l+1)} z_{nj}^{(l)} = \sum_j w_{kj}^{(l+1)} \sigma_l(a_{nj}^{(l)}),$$

  we have (using **Lagrange's notation**):

$$\frac{\partial a_{nk}^{(l+1)}}{\partial a_{ni}^{(l)}} =$$

- Combining gives the recursion for the δ_{ni} terms:

$$\delta_{ni}^{(l)} = \sum_k \delta_{nk}^{(l+1)} w_{ki}^{(l+1)} \sigma_l'(a_{ni}^{(l)}).$$

- It remains to determine $\delta_{ni}^{(L)}$.
- Here it depends on the loss function
- Consider the simplest regression case
  - squared error loss
  - scalar outputs y_n
  - identity output activation $\sigma^{(L)}(a) = a$

$$R_n(\boldsymbol{\theta}) = (f_{\boldsymbol{\theta}}(\boldsymbol{x}_n) - y_n)^2 = \left(z_{n1}^{(L)} - y_n\right)^2 = \left(a_{n1}^{(L)} - y_n\right)^2,$$

so

$$\delta_{n1}^{(L)} = \frac{\partial R_n(\boldsymbol{\theta})}{\partial a_{n1}^{(L)}} = 2\left(a_{n1}^{(L)} - y_n\right).$$

- Classification:
  - Cross-entropy loss
  - σ_l is softmax
-

$$R_n(\boldsymbol{\theta}) = -\sum_k y_{nk} \log[f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)]_k = -\sum_k y_{nk} \log z_{nk}^{(L)}$$
$$= -\sum_k y_{nk} \log \psi_k(\boldsymbol{a}_n^{(L)}),$$

so

$$\delta_{ni}^{(L)} = -\sum_k y_{nk} \frac{1}{\psi_k(\boldsymbol{a}_n^{(L)})} \frac{\partial \psi_k(\boldsymbol{a}_n^{(L)})}{\partial a_{ni}^{(L)}}.$$

Now use this property of the derivative of softmax:

$$\frac{\partial \psi_k(\boldsymbol{a})}{\partial a_i} = \begin{cases} \psi_k(\boldsymbol{a})(1 - \psi_i(\boldsymbol{a})), & i = k \\ -\psi_k(\boldsymbol{a})\psi_i(\boldsymbol{a}), & i \neq k. \end{cases}$$

Backpropagation - summary

© 2026 Fessler

Forward pass:

Using current weights $\boldsymbol{\theta}$, compute $f(\boldsymbol{x}_n)$ and *store* intermediate values $a_{ni}^{(l)}, z_{nj}^{(l)}$

Initialize backward pass:

For $i = 1$ to d_L

Compute $\delta_{ni}^{(L)}$ (using loss function)

End

Backward pass:

For $l = L - 1$ down to 1

For $i = 1$ to d_l

$\delta_{ni}^{(l)} = \sum_k \delta_{nk}^{(l+1)} w_{ki}^{(l+1)} \sigma_l'(a_{ni}^{(l)})$

For $j = 1$ to d_{l-1}

$\frac{\partial R_n(\boldsymbol{\theta})}{\partial w_{ij}^{(l)}} \leftarrow \delta_{ni}^{(l)} z_{nj}^{(l-1)}$

End

End

End

Optimization options

- Use this gradient of R_n in SGD:
  $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \alpha_t \nabla R_n(\boldsymbol{\theta}_{t-1})$
- Or repeat for every n and accumulate to get full gradient $\nabla R(\boldsymbol{\theta}_{t-1})$.
- Or combine multiple samples in a **mini-batch** to exploit GPU SIMD.

Deep feedforward network remarks

- Note: each hidden unit passes and receives information to and from only those units to which it is connected $\implies$ parallel implementation.
- In general, $R(\boldsymbol{\theta})$ will have several local minima. Because the overall cost function is nonconvex, convergence to a global minimizer is not guaranteed.
- To avoid overfitting, regularization is often used.
- Initializing $\boldsymbol{\theta}_0$: typically small iid Gaussian, often with layer-dependent variances (Kaiming He, 2015) [12], or sometimes random orthogonal matrices.
- When training a NN with backpropagation of the squared error loss, the resulting network is a LS fit to the Bayes discriminant function [8, p. 303].
- **universal approximation theorems** state that sufficiently large NNs (like basic MLPs) can approximate any continuous function
- Everything discussed so far has been understood since the 1980s

References I

[1] K. P. Murphy.
Probabilistic machine learning: An introduction.
MIT Press, 2022.

[2] Ehsan Elhamifar and René Vidal.
Sparse manifold clustering and embedding.
Advances in neural information processing systems, 24, 2011.

[3] G. Nataraj, J-F. Nielsen, C. D. Scott, and J. A. Fessler.
Dictionary-free MRI PERK: Parameter estimation via regression with kernels.
IEEE Trans. Med. Imag., 37(9):2103–14, September 2018.

[4] J. A. Fessler and R. R. Nadakuditi.
Linear algebra for data science, machine learning, and signal processing.
Cambridge, 2024.

[5] C. Bishop.
Pattern recognition and machine learning.
Springer, 2006.

[6] Bernhard Schölkopf, Ralf Herbrich, and Alex J Smola.
A generalized representer theorem.
In *International conference on computational learning theory*, pages 416–426. Springer, 2001.

[7] SangGyu An.
Introduction to how an multilayer perceptron works but without complicated math, 2021.
`https://medium.com/codex/introduction-to-how-an-multilayer-perceptron-works-but-without-complicated-math-a423979897ac`.

[8] R. O. Duda, P. E. Hart, and D. G. Stork.
Pattern classification.
Wiley, New York, 2001.

References II

[9] D. P. Kingma and J. Ba.
Adam: A method for stochastic optimization, 2014.

[10] K. Jordan.
Muon: An optimizer for hidden layers in neural networks, 2024.

[11] T. Pethick, W. Xie, K. Antonakopoulos, Z. Zhu, A. Silveti-Falls, and V. Cevher.
Training deep learning models with norm-constrained LMOs.
In *Proc. Intl. Conf. Mach. Learn*, 2025.

[12] K. He, X. Zhang, S. Ren, and J. Sun.
Delving deep into rectifiers: surpassing human-level performance on imageNet classification.
In *Proc. Intl. Conf. Comp. Vision*, pages 1026–34, 2015.

[13] Simone Bianco, Remi Cadene, Luigi Celona, and Paolo Napoletano.
Benchmark analysis of representative deep neural network architectures.
IEEE access, 6:64270–64277, 2018.

[14] Tim Dettmers.
Deep learning in a nutshell: Core concepts, 2015.
https://developer.nvidia.com/discover/convolution.

[15] Niranjan Kumar.
Visualizing convolution neural networks using pytorch, 2019.
https://towardsdatascience.com/visualizing-convolution-neural-networks-using-pytorch-3dfa8443e74e.

[16] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner.
Gradient-based learning applied to document recognition.
Proceedings of the IEEE, 86(11):2278–2324, 1998.

References III

[17] Jinde Shubham.
What exactly does cnn see?, 2018.
https://becominghuman.ai/what-exactly-does-cnn-see-4d436d8e6e52.

[18] O. Ronneberger, P. Fischer, and T. Brox.
U-net: convolutional networks for biomedical image segmentation.
In *Medical Image Computing and Computer-Assisted Intervention*, pages 234–41, 2015.

[19] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei.
Imagenet: A large-scale hierarchical image database.
In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.