Statistical Machine Learning (BE4M33SSU)
Lecture 7: Artificial Neural Networks

Jan Drchal

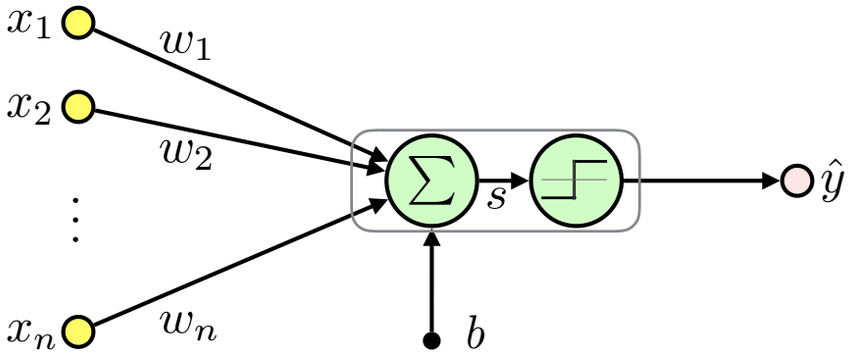
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science

Overview

Topics covered in the lecture:

- Neuron types
- Layers
- Loss functions
- Computing loss gradients via backpropagation
- Learning neural networks
- Regularization

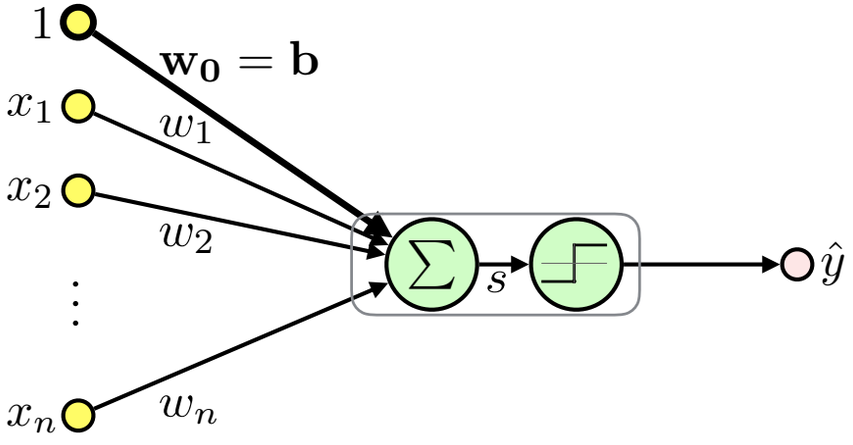
McCulloch-Pitts Perceptron



$\boldsymbol{x} = \left(x_1, x_2, \ldots, x_n\right)^T \in \mathbb{R}^n$	input
$\hat{y} \in \{-1, 1\}$	output (activity)
$\boldsymbol{w} = \left(w_1, w_2, \ldots, w_n\right)^T \in \mathbb{R}^n$	weights
$b \in \mathbb{R}$	bias (threshold)
$s \in \mathbb{R}$	inner potential
$f(s) = \begin{cases} -1 & \text{if } s < 0 \\ 1 & \text{else} \end{cases}$	activation function

$$\hat{y} = f(s) = f\left(\sum_{i=1}^{n} w_i x_i + b\right) = f\left(\boldsymbol{w} \cdot \boldsymbol{x} + b\right)$$

McCulloch-Pitts Perceptron: Treating Bias

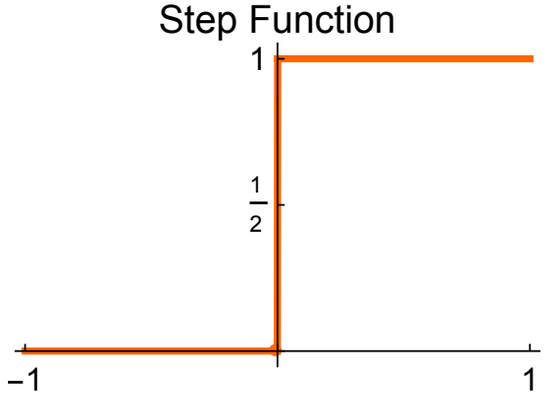


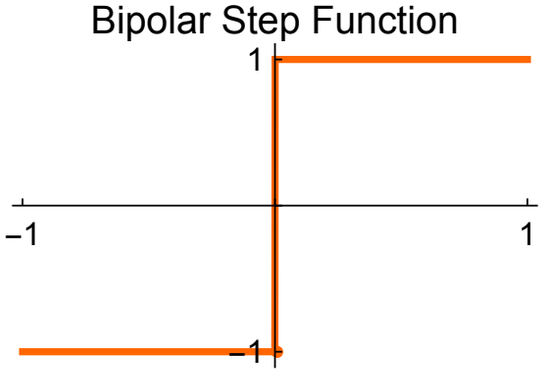
- Treat bias as an extra fixed input $x_0 = 1$ and weighted $w_0 = b$:

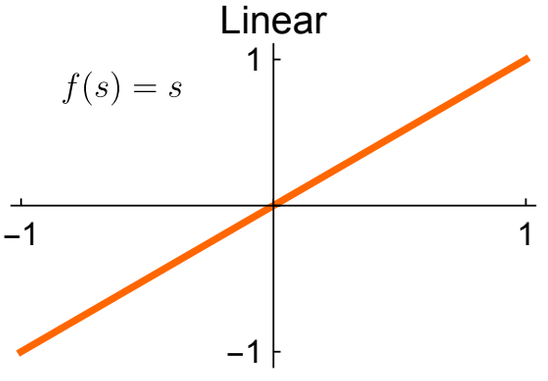
$$\hat{y} = f\left(\boldsymbol{w} \cdot \boldsymbol{x} + b\right) = f\left(\boldsymbol{w} \cdot \boldsymbol{x} + w_0 \cdot 1\right) = f\left(\boldsymbol{w'} \cdot \boldsymbol{x'}\right)$$

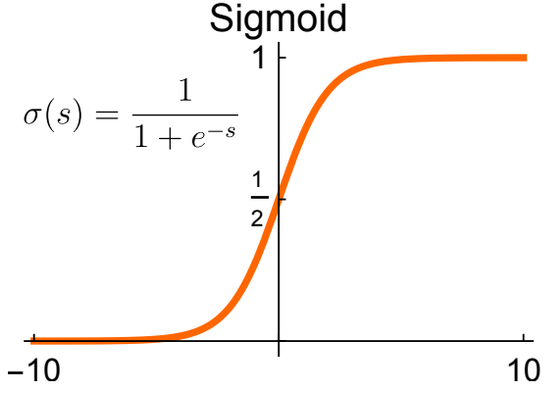
- $\boldsymbol{x'} = \left(x_0, x_1, \ldots, x_n\right)^T \in \mathbb{R}^{n+1}$
- $\boldsymbol{w'} = \left(w_0, w_1, \ldots, w_n\right)^T \in \mathbb{R}^{n+1}$
- Unless otherwise noted we will use $\boldsymbol{x}$, $\boldsymbol{w}$ instead of $\boldsymbol{x'}$, $\boldsymbol{w'}$

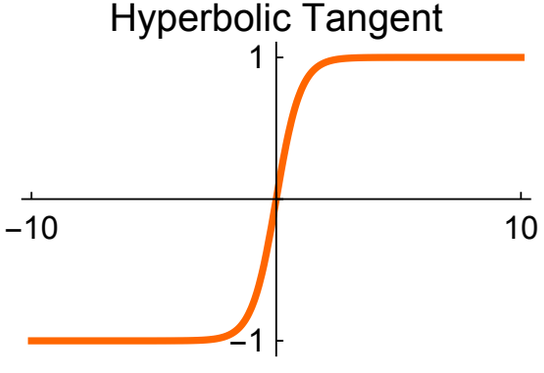
Activation Functions

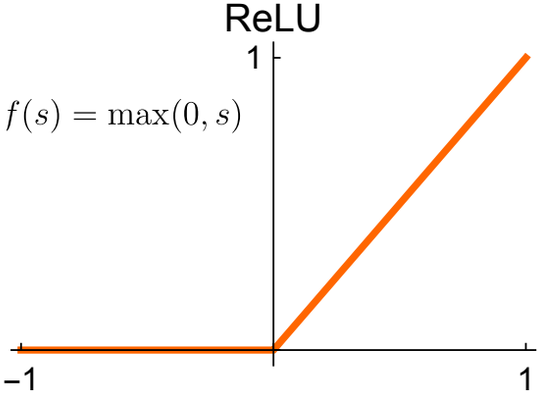












- Sigmoid: $\sigma(s) \triangleq \dfrac{1}{1+e^{-s}} = \dfrac{e^s}{e^s+1}$
- Note: $\tanh(s) = \dfrac{e^s - e^{-s}}{e^s + e^{-s}} = 2\,\sigma(s) - 1$

Linear Neuron

- Single neuron with linear activation function $\equiv$ **linear regression**:

$$\hat{y} = s = \boldsymbol{x} \cdot \boldsymbol{w}, \quad \hat{y}, s \in \mathbb{R}$$

- Inputs: $\mathbf{X} = \begin{pmatrix} 1 & x_{11} & \dots & x_{1n} \\ 1 & \vdots & \ddots & \vdots \\ 1 & x_{m1} & \dots & x_{mn} \end{pmatrix} = \begin{pmatrix} \boldsymbol{x}_1^T \\ \vdots \\ \boldsymbol{x}_m^T \end{pmatrix}$
- Targets: $\boldsymbol{y} = \big(y_1, \dots, y_m\big)^T$, $y_i \in \mathbb{R}^m$
- Outputs: $\hat{\boldsymbol{y}} = \big(\hat{y}_1, \dots, \hat{y}_m\big)^T$, $\hat{y}_i \in \mathbb{R}^m$
- For the whole dataset we get:

$$\hat{\boldsymbol{y}} = \mathbf{X}\boldsymbol{w}, \hat{\boldsymbol{y}} \in \mathbb{R}^m$$

Linear Neuron: Maximum Likelihood Estimation

- Assumption: data are Gaussian distributed with mean $\boldsymbol{x}_i \cdot \boldsymbol{w}$ and variance σ^2:

$$y_i \sim \mathcal{N}\left(\boldsymbol{x}_i \cdot \boldsymbol{w}, \sigma^2\right) = \boldsymbol{x}_i \cdot \boldsymbol{w} + \mathcal{N}\left(0, \sigma^2\right)$$

- Likelihood for i.i.d. data:

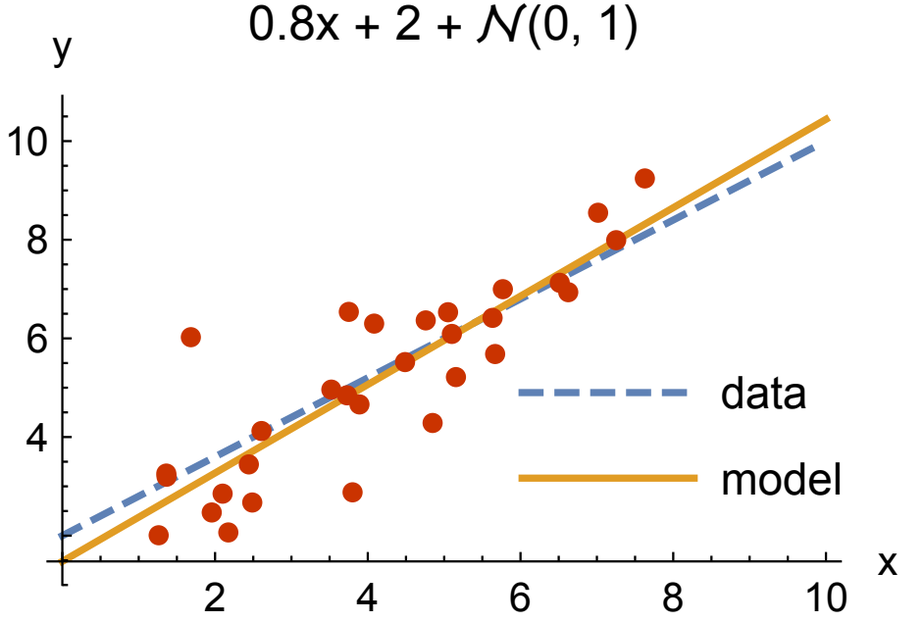
$$\begin{aligned} p\left(\boldsymbol{y}|\mathbf{X}, \boldsymbol{w}, \sigma\right) &= \prod_{i=1}^{m} p\left(y_i|\boldsymbol{x_i}, \boldsymbol{w}, \sigma\right) = \prod_{i=1}^{m} \left(2\pi\sigma^2\right)^{-\frac{1}{2}} e^{-\frac{1}{2\sigma^2}\left(y_i - \boldsymbol{w}\cdot\boldsymbol{x}_i\right)^2} = \\ &= \left(2\pi\sigma^2\right)^{-\frac{m}{2}} e^{-\frac{1}{2\sigma^2}\sum_{i=1}^{m}\left(y_i - \boldsymbol{w}\cdot\boldsymbol{x}_i\right)^2} = \\ &= \left(2\pi\sigma^2\right)^{-\frac{m}{2}} e^{-\frac{1}{2\sigma^2}\left(\boldsymbol{y} - \mathbf{X}\boldsymbol{w}\right)^T\left(\boldsymbol{y} - \mathbf{X}\boldsymbol{w}\right)} \end{aligned}$$

- Negative Log Likelihood:

$$\mathcal{L}\left(\boldsymbol{w}\right) = \frac{m}{2}\log\left(2\pi\sigma^2\right) + \frac{1}{2\sigma^2}\left(\boldsymbol{y} - \mathbf{X}\boldsymbol{w}\right)^T\left(\boldsymbol{y} - \mathbf{X}\boldsymbol{w}\right)$$

Linear Neuron: Maximum Likelihood Estimation (contd.)

- Note that $\sum_{i=1}^{m} (y_i - \boldsymbol{w} \cdot \boldsymbol{x}_i)^2 = (\boldsymbol{y} - \mathbf{X}\boldsymbol{w})^T (\boldsymbol{y} - \mathbf{X}\boldsymbol{w})$ is the **sum-of-squares** or **squared error** (SE)
- Minimization of $\mathcal{L}(\boldsymbol{w}) \equiv$ least squares estimaton
- Solving $\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = 0$ we get $\boldsymbol{w}^* = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\boldsymbol{y}$ (see seminar)
- Note $\hat{\mathbf{X}} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T$ is called the *Moore-Penrose pseudo-inverse*: if $\mathbf{X}$ is square and invertible then $\hat{\mathbf{X}} = \mathbf{X}^{-1}$ (use $(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$)

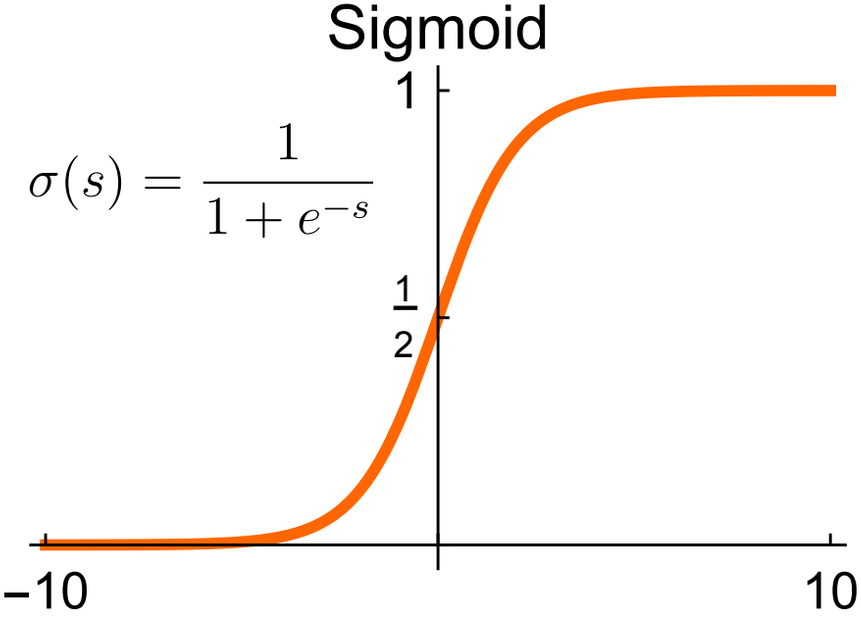


Sigmoid and Probability

- Denote: $\hat{y} = \sigma(s),\ \hat{y} \in (0, 1)$
- Sigmoid output can represent a parameter of the Bernoulli distribution:

$$p(y|\hat{y}) = \mathrm{Ber}(y|\hat{y}) = \hat{y}^y \left(1 - \hat{y}\right)^{1-y} = \begin{cases} \hat{y} & \text{for } y = 1 \\ 1 - \hat{y} & \text{for } y = 0 \end{cases}$$

- Motivation: log-odds linear model (see AE4B33RPZ)
- Binary classifier: $h(\hat{y}) = \begin{cases} 1 & \text{if } \hat{y} > \frac{1}{2} \\ 0 & \text{else} \end{cases}$



Logistic Regression

- MCP neuron using sigmoid activation function $\equiv$ **logistic regression**:

$$\hat{y} = \sigma(\boldsymbol{w} \cdot \boldsymbol{x}),\ \hat{y} \in (0, 1)$$

- Inputs: $\mathbf{X} = \begin{pmatrix} 1 & x_{11} & \dots & x_{1n} \\ 1 & \vdots & \ddots & \vdots \\ 1 & x_{m1} & \dots & x_{mn} \end{pmatrix} = \begin{pmatrix} \boldsymbol{x}_1^T \\ \vdots \\ \boldsymbol{x}_m^T \end{pmatrix}$
- Target class: $\boldsymbol{y} = \big(y_1, \dots, y_m\big)^T,\ y_i \in \{0, 1\}$
- Output class: $\hat{\boldsymbol{y}} = \big(\hat{y}_1, \dots, \hat{y}_m\big)^T,\ \hat{y}_i \in (0, 1)$

Cross-Entropy

- Likelihood, for the logistic regression:

$$p(\boldsymbol{y}|\boldsymbol{w}, \mathbf{X}) = \prod_{i=1}^{m} \mathrm{Ber}(y_i|\hat{y}_i) = \prod_{i=1}^{m} \hat{y}_i^{y_i} \left(1 - \hat{y}_i\right)^{1-y_i}$$

- Negative Log Likelihood:

$$\mathcal{L}(\boldsymbol{w}) = -\sum_{i=1}^{m} \left[y_i \log \hat{y}_i + (1 - y_i) \log \left(1 - \hat{y}_i\right)\right]$$

- This *loss function* is called the **cross-entropy**

Maximum Likelihood Estimation

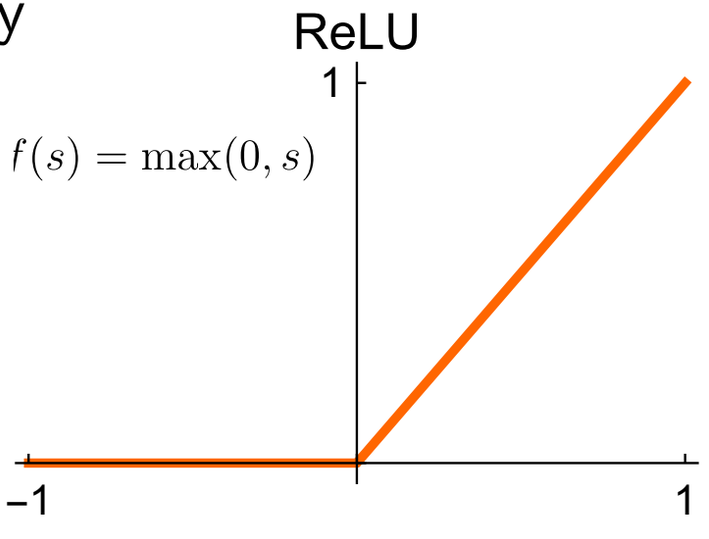
- Maximum Likelihood Estimation: $w^* = \operatorname{argmin}_w \mathcal{L}(\boldsymbol{w})$
- Derivative of the loss w.r.t. to the sigmoid argument: $\frac{\partial \mathcal{L}}{\partial s_i} = \hat{y}_i - y_i$
- Gradient w.r.t. logistic regression parameters:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = \sum_{i=1}^{m} \frac{\partial \mathcal{L}}{\partial s_i} \cdot \frac{\partial s_i}{\partial \boldsymbol{w}} = \sum_{i=1}^{m} \boldsymbol{x}_i(\hat{y}_i - y_i) = \mathbf{X}^T(\hat{\boldsymbol{y}} - \boldsymbol{y})$$

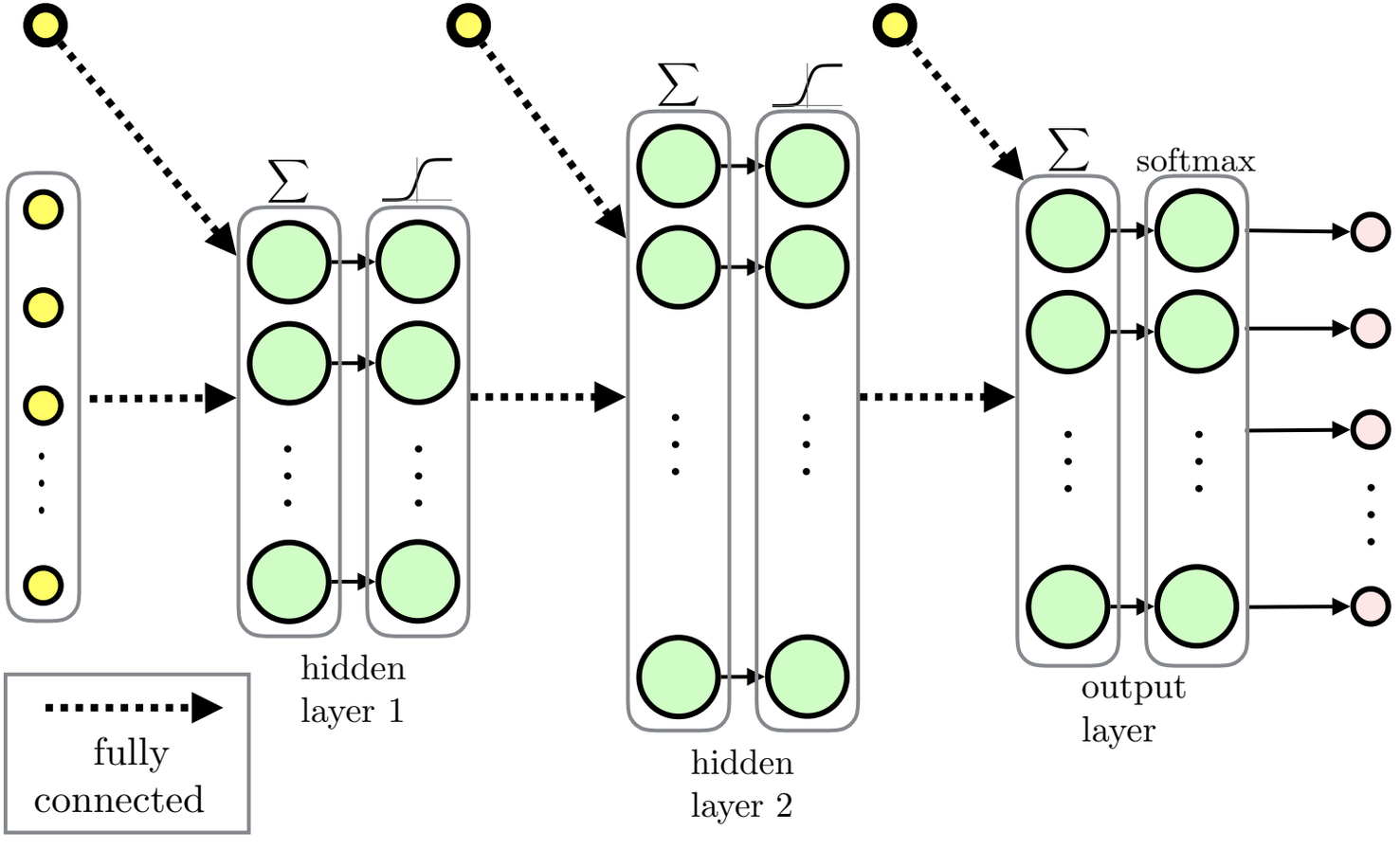
- $\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = 0$ has no analytical solution $\implies$ use numerical methods
- Hessian: $\frac{\partial^2 \mathcal{L}}{\partial \boldsymbol{w}^2} = \sum_{i=1}^{m} \hat{y}_i(1 - \hat{y}_i)\boldsymbol{x}_i \cdot \boldsymbol{x}_i = \mathbf{X}^T\mathbf{S}\mathbf{X}$,
  where $\mathbf{S} \triangleq \operatorname{diag}(\hat{y}_i(1 - \hat{y}_i))$
- Hessian is positive definite, hence the loss is convex and has unique global minimum (see AE4B33RPZ)

Rectified Linear Unit (ReLU)

- Definition $f(s) = \max(0, s)$
- Fast to compute
- Helps with *vanishing gradients* problem: the gradient is constant for $s > 0$, while for sigmoid-like activations it becomes increasingly small
- Leads to sparse representations: $s < 0$ turns the neuron completely off
- Unbounded: use regularization to prevent numerical problems
- Might block gradient propagation $\rightarrow$ dead units $\rightarrow$ Leaky ReLU
- Satisfies the universal approximation property

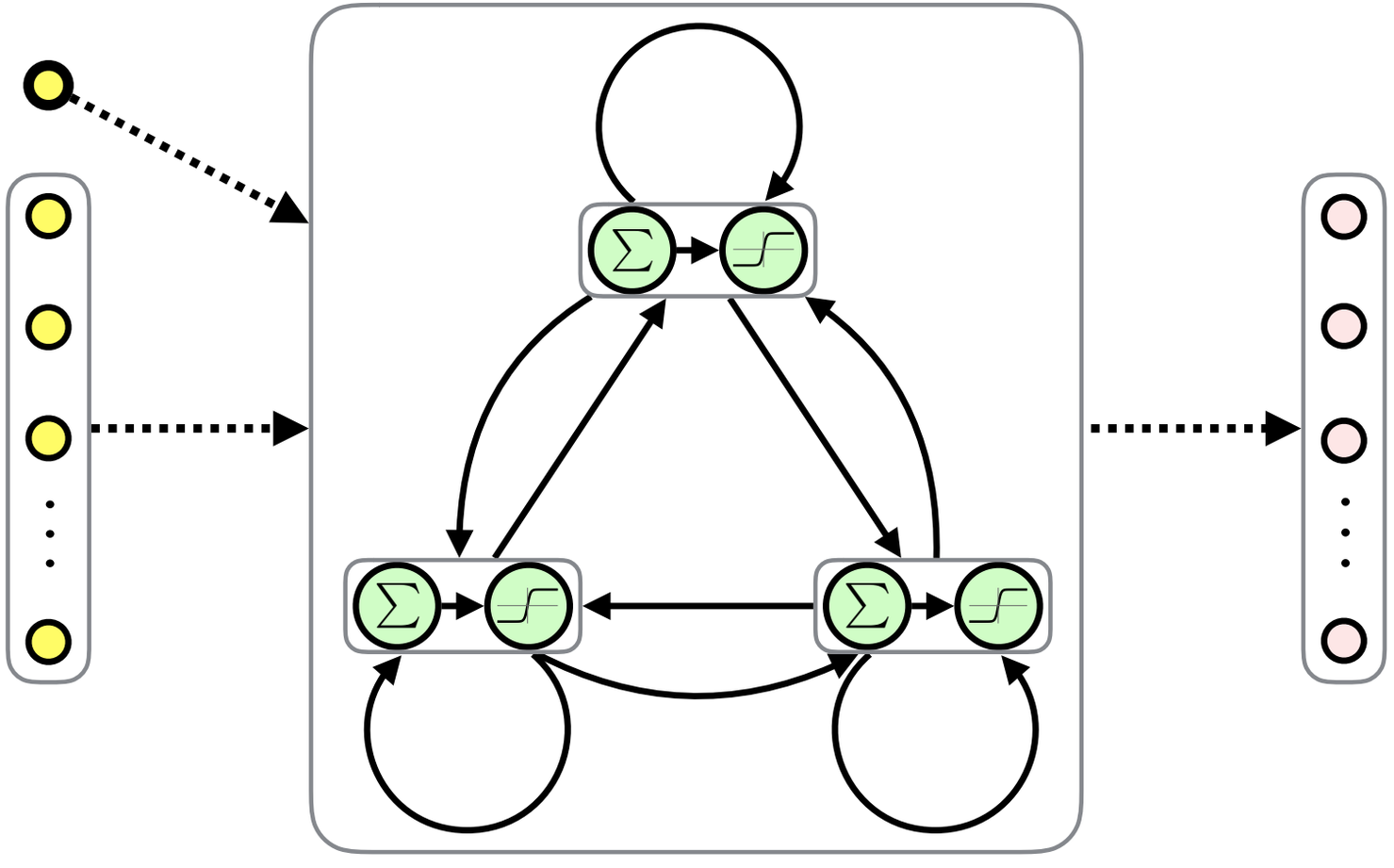


Multilayer Perceptron (MLP)



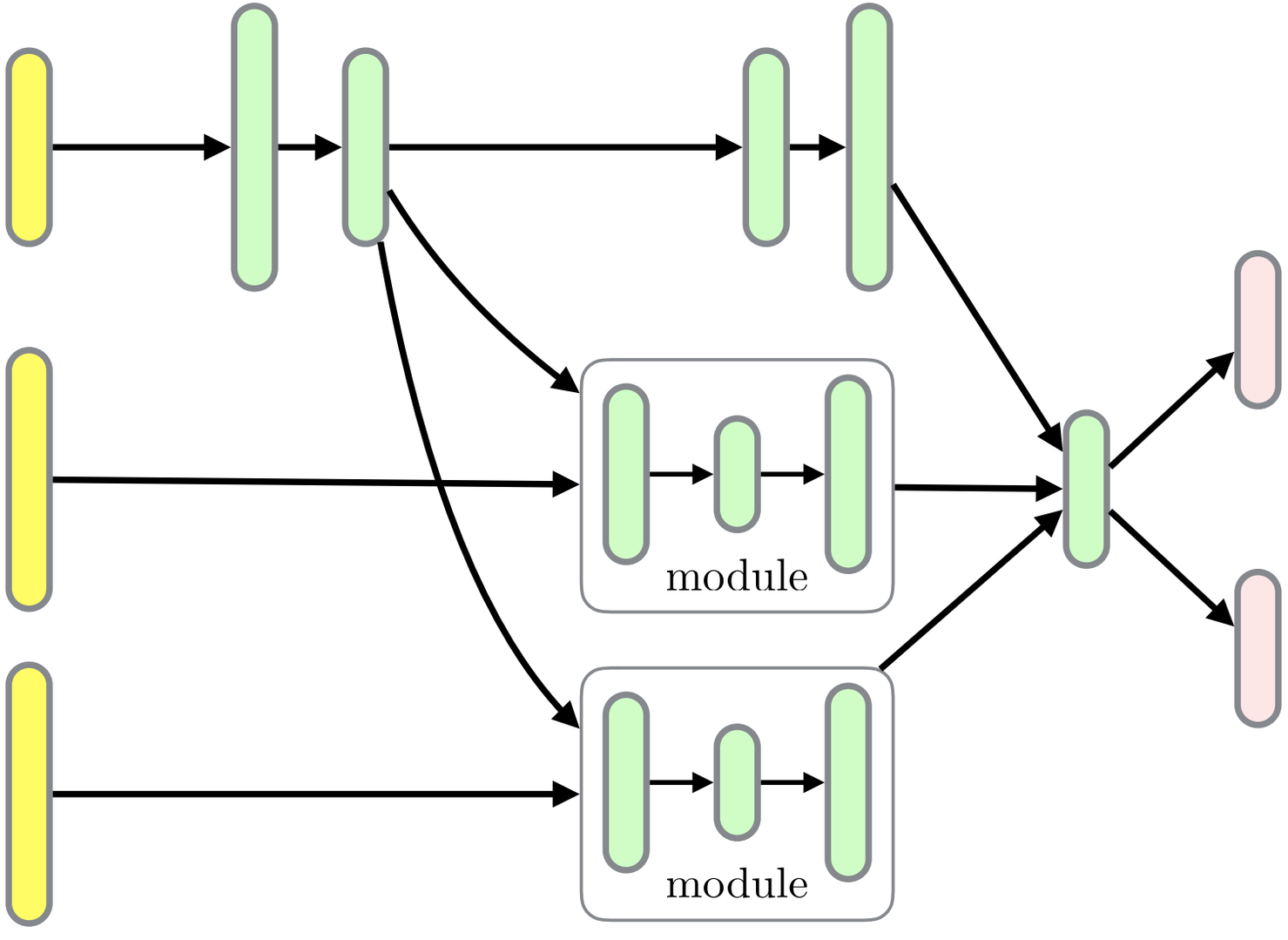
- Feed-forward ANN
- Fully-connected layers
- MLP for regression would typically use linear output layer

Recurrent Neural Network (RNN)



- Fully-Connected Recurrent Neural Network (FRNN)
- Both inputs and outputs are sequences
- Feedback connections $\rightarrow$ memory

Modular and Hierarchical Architectures

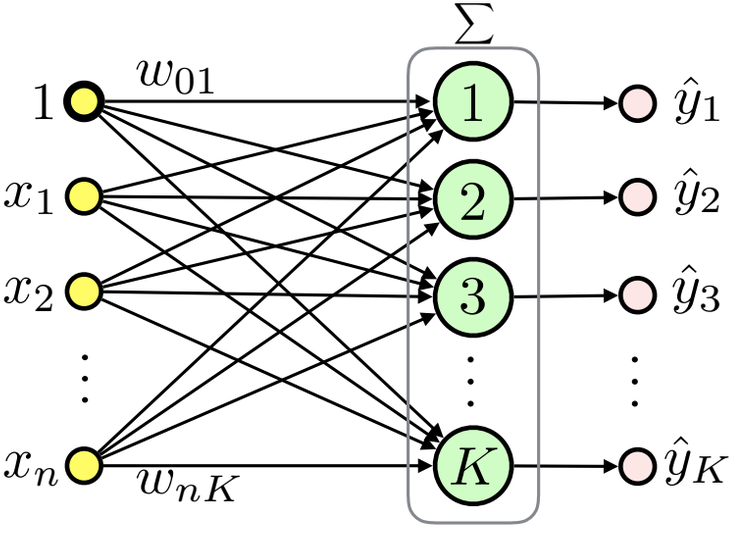


- Layers can be organized in *modules*
- Hierarchies of modules
- Module reuse

Linear Layer

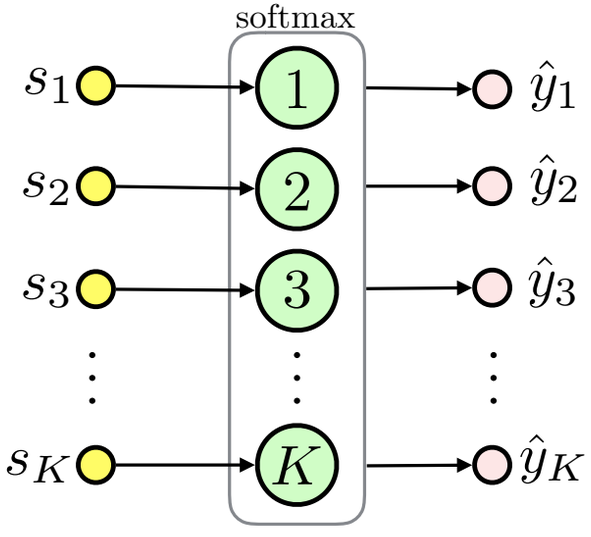
- Output k: $\hat{y}_k = \boldsymbol{x} \cdot \boldsymbol{w}_k,\ k = 0, 1, \ldots, K$
- All outputs using *weight matrix* $\mathbf{W}$: $\hat{\boldsymbol{y}} = \boldsymbol{x}^T \mathbf{W}$
- Multiple samples: $\hat{\mathbf{Y}} = \mathbf{X}\mathbf{W}$

$$\mathbf{W} = \begin{pmatrix} \boldsymbol{w}_1^T \\ \vdots \\ \boldsymbol{w}_K^T \end{pmatrix}^T = \begin{pmatrix} w_{01} & \ldots & w_{0K} \\ \vdots & \ddots & \vdots \\ w_{n1} & \ldots & w_{nK} \end{pmatrix}$$



$$\mathbf{X} = \begin{pmatrix} \boldsymbol{x}_1^T \\ \vdots \\ \boldsymbol{x}_m^T \end{pmatrix} = \begin{pmatrix} 1 & x_{11} & \ldots & x_{1n} \\ 1 & \vdots & \ddots & \vdots \\ 1 & x_{m1} & \ldots & x_{mn} \end{pmatrix} \qquad \hat{\mathbf{Y}} = \begin{pmatrix} \hat{\boldsymbol{y}}_1^T \\ \vdots \\ \boldsymbol{y}_m^T \end{pmatrix} = \begin{pmatrix} \hat{y}_{11} & \ldots & \hat{y}_{1K} \\ \vdots & \ddots & \vdots \\ \hat{y}_{m1} & \ldots & \hat{y}_{mK} \end{pmatrix}$$

Softmax Layer



- Multinominal classification
- Definition: $\sigma_k(\boldsymbol{s}) \triangleq \frac{e^{s_k}}{\sum_{c=1}^{K} e^{s_c}}$, where K is the number of classes
- Softmax represents a probability distribution: $0 \leq \sigma_k \leq 1$ for $k \in \{1 \dots K\}$ and $\sum_{c=1}^{K} \sigma_c = 1$
- Describes class membership probabilities: $p(y = k|\boldsymbol{s}) = \sigma_k(\boldsymbol{s})$

Softmax Layer (contd.)

- Target: $\boldsymbol{y} = \begin{pmatrix} y_1 & \dots & y_m \end{pmatrix}^T$, $y_i \in \{1, 2, \dots, K\}$
- One-hot encoding for sample i and class k: let $y_{ik} = \mathbb{I}\{y_i = k\}$ and $\hat{y}_{ik} = \mathbb{I}\{\hat{y}_i = k\}$
- Likelihood:

$$p(\boldsymbol{y}|\boldsymbol{w}, \mathbf{X}) = \prod_{i=1}^{m} \prod_{c=1}^{K} \hat{y}_{ic}^{y_{ic}}$$

- Negative Log Likelihood:

$$\mathcal{L}(\boldsymbol{w}) = -\sum_{i=1}^{m} \sum_{c=1}^{K} \mathbb{I}\{y_i = c\} \log(\hat{y}_{ic})$$

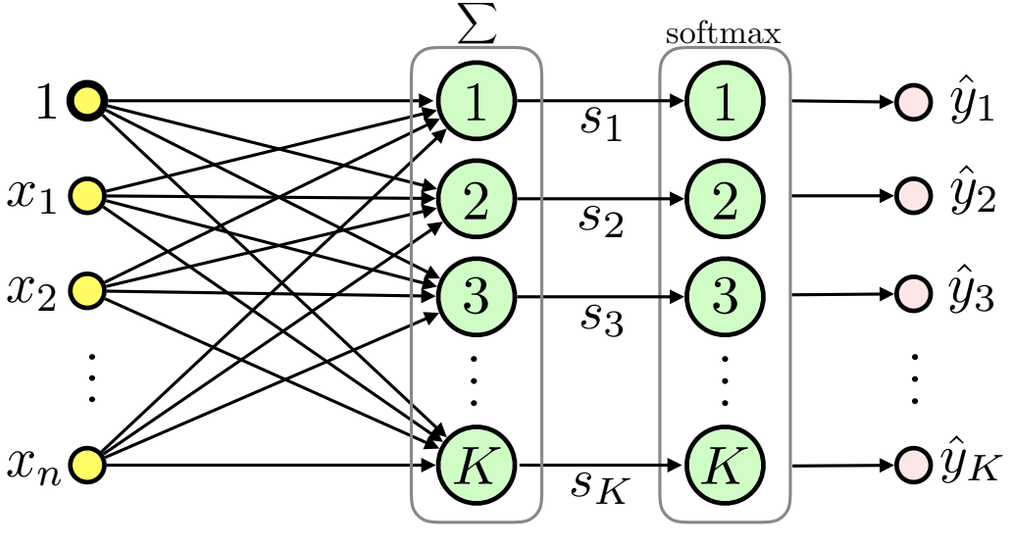
  Again the **cross-entropy**

- See seminar for the gradient

Multinominal Logistic Regression

- linear layer + softmax layer = **multinominal logistic regression**:

$$\hat{y}_k = \sigma_k(\boldsymbol{w}_k \cdot \boldsymbol{x})$$



- Classifier: $h(\boldsymbol{x}, \mathbf{W}) = \underset{k}{\operatorname{argmax}}\, \hat{y}_k$

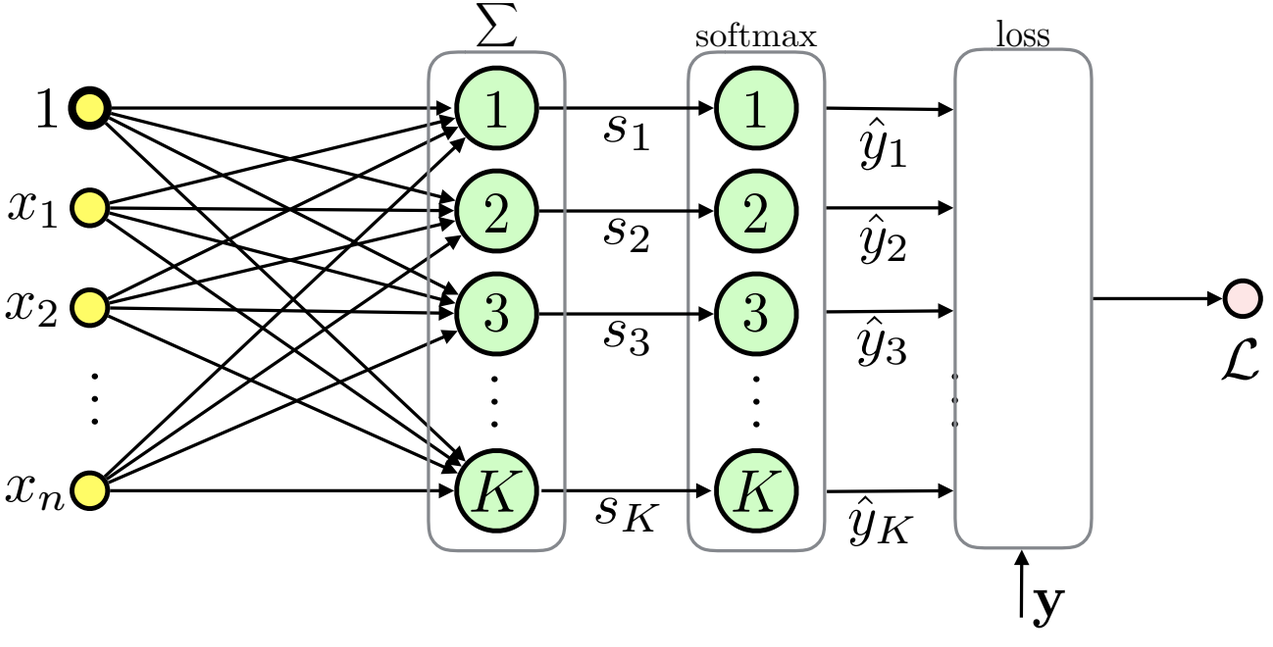
Loss Functions: Summary

problem	**suggested loss function**
binary classification	cross-entropy $-\sum_{i=1}^{m} [y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i)]$
multinominal classification	multinominal cross-entropy $-\sum_{i=1}^{m} \sum_{c=1}^{K} \mathbb{I}\{y_i = c\} \log(\hat{y}_{ic})$
regression	squared error $\sum_{i=1}^{m} (y_i - \hat{y}_i)^2$
multi-output regression	squared error $\sum_{i=1}^{m} \sum_{c=1}^{K} (y_{ic} - \hat{y}_{ic})^2$

- Mean w.r.t. to m is often used

Backpropagation Overview

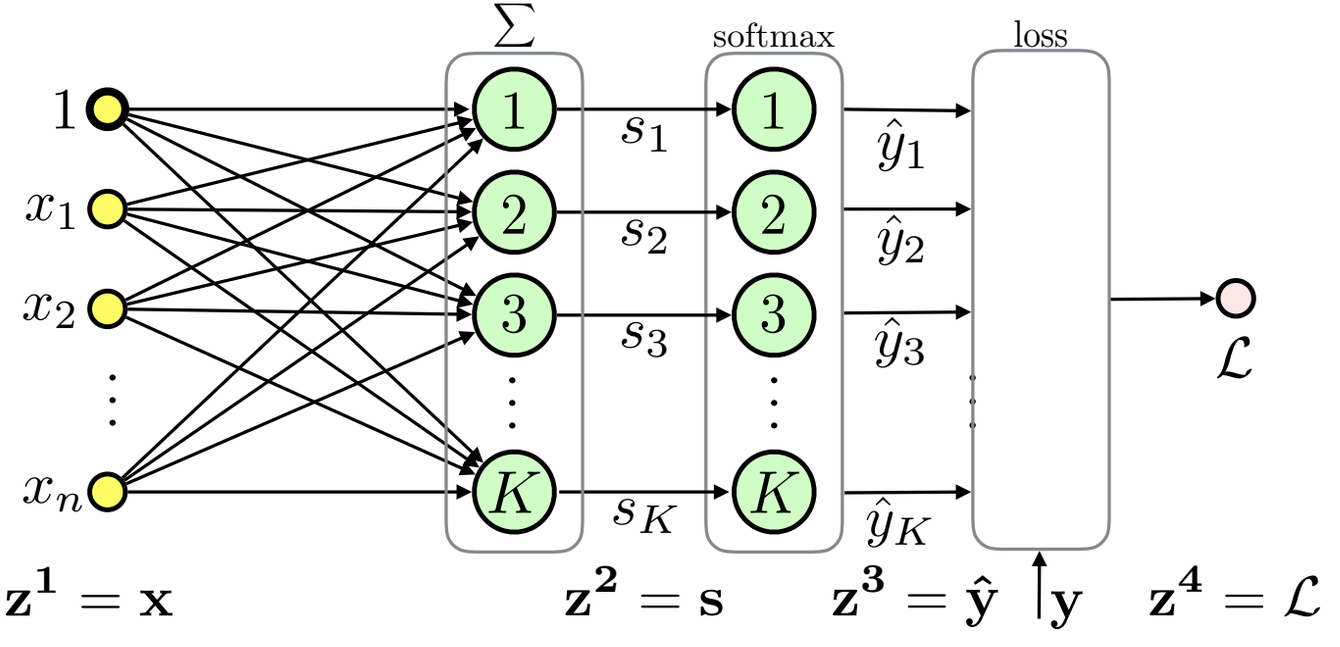
- A method to compute gradient of the *loss function* with respect to its parameters
- Here, we present the "modular" backpropagation (see Nando de Freitas' Machine Learning course: https://www.cs.ox.ac.uk/people/nando.defreitas/machinelearning/)
- Let us use multinominal logistic regression as an example



Backpropagation: the Loss Function

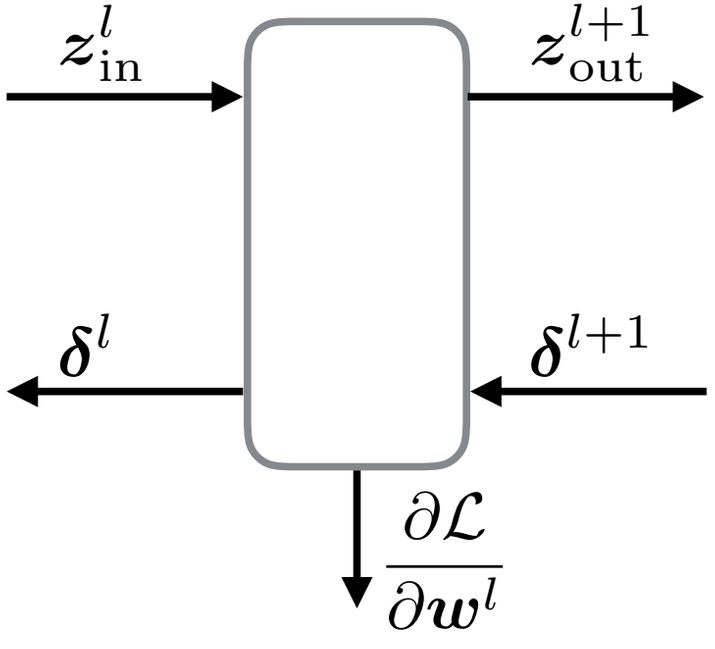
- The loss function is the multinominal cross-entropy in this case:

$$\mathcal{L}(\boldsymbol{w}) = -\sum_{i=1}^{m}\sum_{c=1}^{K} \mathbb{I}\{y_i = c\} \log\left(\frac{\exp\left(\boldsymbol{x}_i \cdot \boldsymbol{w}_c\right)}{\sum_{k=1}^{K}\exp\left(\boldsymbol{x}_i \cdot \boldsymbol{w}_k\right)}\right)$$



Backpropagation Based on Modules

- Computation of $\nabla \mathcal{L}(\boldsymbol{w})$ involves repetitive use of the *chain rule*
- We can make things simpler by divide and conquer approach
- Divide to simplest possible modules (these can be later combined into complex hierarchies)
- Represent even the loss function as a module
- Passing messages

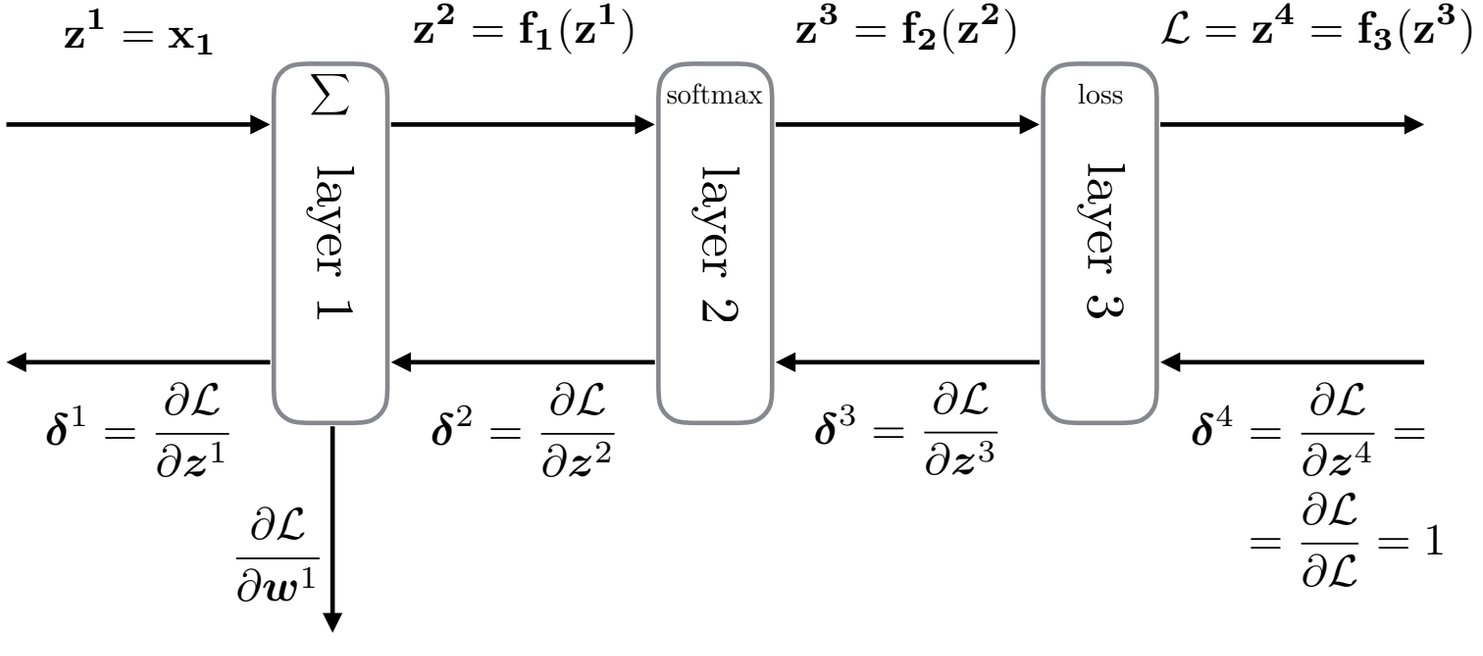


Backpropagation: Backward Pass Message

- Let $\boldsymbol{\delta}^l = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^l}$ be the sensitivity of the loss to the module output for layer l, then:

$$\delta_i^l = \frac{\partial \mathcal{L}}{\partial z_i^l} = \sum_j \frac{\partial \mathcal{L}}{\partial z_j^{l+1}} \cdot \frac{\partial z_j^{l+1}}{\partial z_i^l} = \sum_j \delta_j^{l+1} \frac{\partial z_j^{l+1}}{\partial z_i^l}$$

- We need to know how to compute derivatives of outputs w.r.t. inputs only!

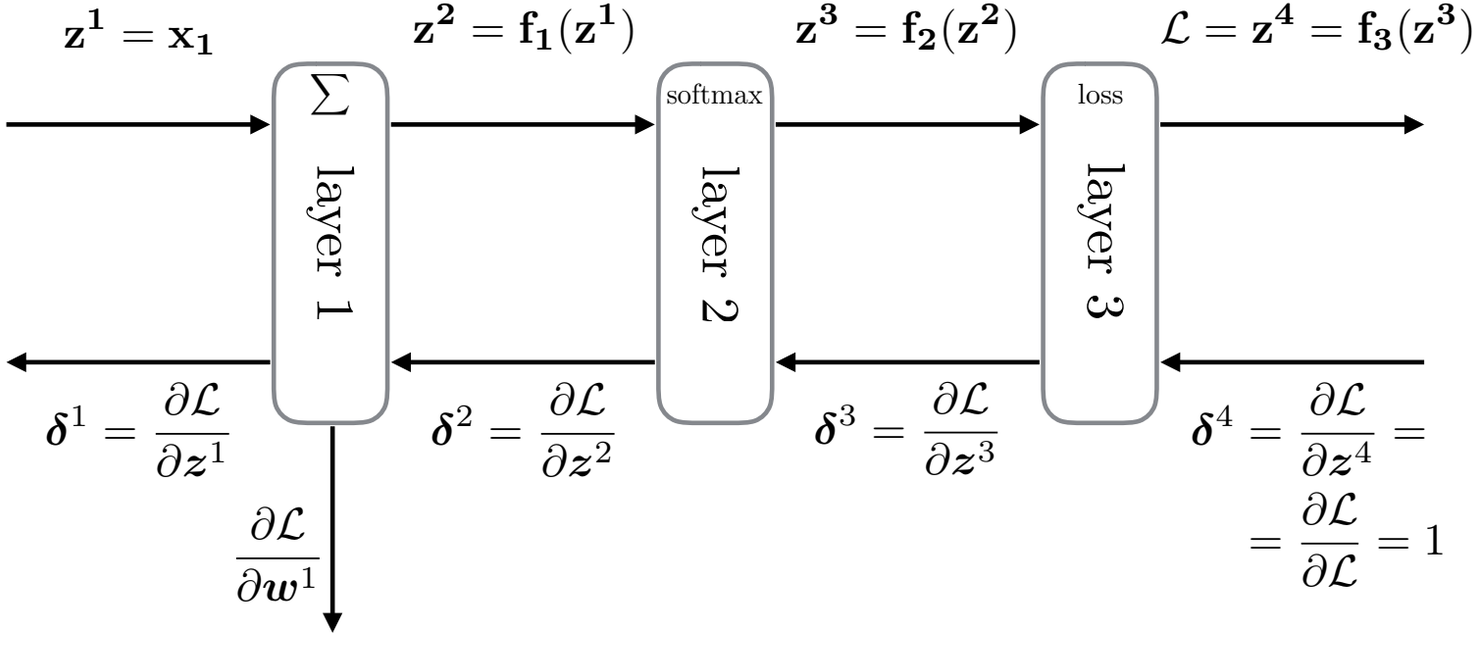


Backpropagation: Parameters

- Similarly if the module has parameters we want to know how the loss changes w.r.t. them:

$$\frac{\partial \mathcal{L}}{\partial w_i^l} = \sum_j \frac{\partial \mathcal{L}}{\partial z_j^{l+1}} \cdot \frac{\partial z_j^{l+1}}{\partial w_i^l} = \sum_j \delta_j^{l+1} \frac{\partial z_j^{l+1}}{\partial w_i^l}$$

- Derivatives of module outputs w.r.t. to the parameters are all we need



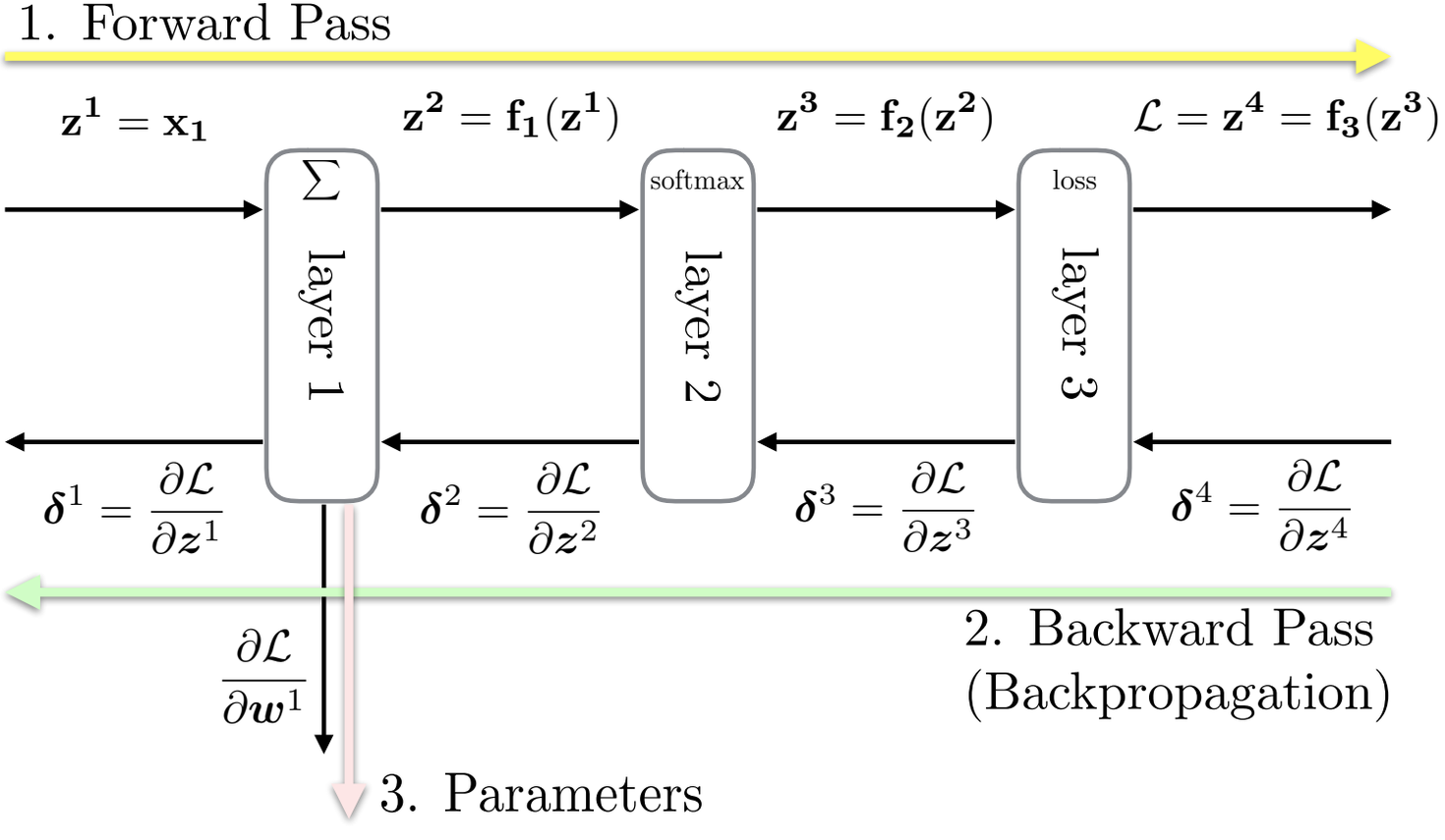
Backpropagation: Steps

- So for each module we need only to specify these three messages:

  forward: $z^{l+1} = f(z^l)$

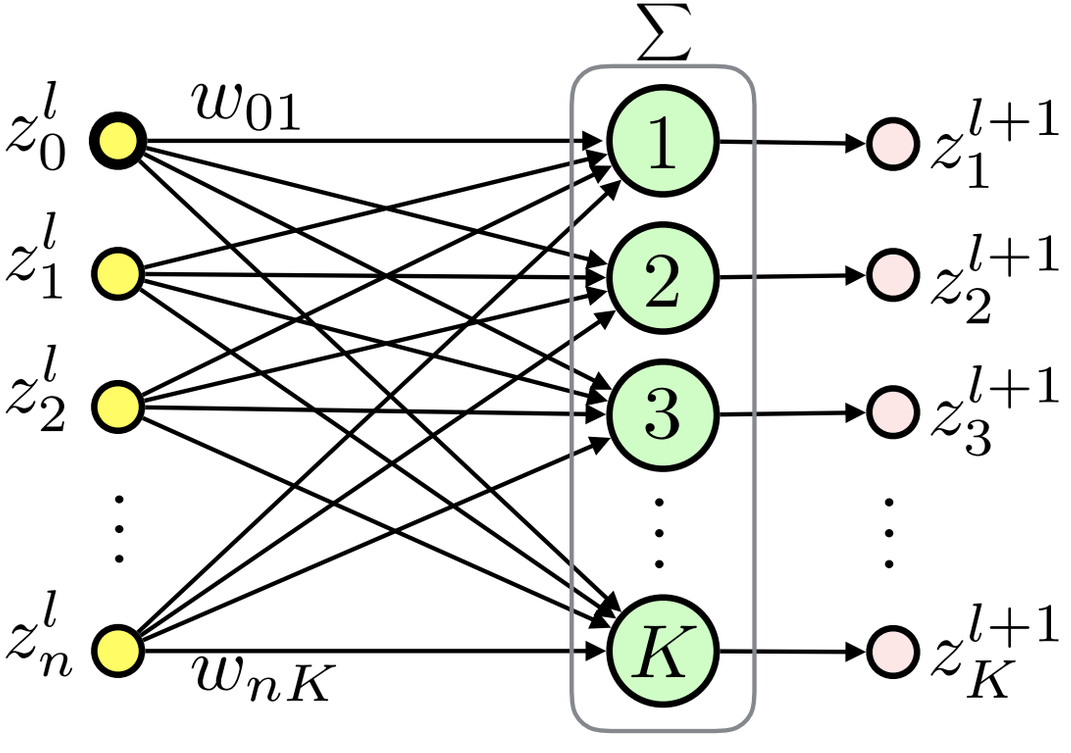
  backward: $\frac{\partial z^{l+1}}{\partial z^l}$

  parameter (optional): $\frac{\partial z^{l+1}}{\partial w^l}$



Example: Linear Layer

- **forward**: $z_j^{l+1} = \sum_{i=0}^{n} w_{ij} z_i^l, \quad j = 1, \ldots, K$
- **backward**: $\frac{\partial z_j^{l+1}}{\partial z_i^l} = w_{ij}, \quad i = 0, \ldots, n, \quad j = 1, \ldots, K$
- **parameter**: $\frac{\partial z_j^{l+1}}{\partial w_{ik}} = \mathbb{I}\{j = k\} z_i^l$



Example: Squared Error

- **forward**: $z^{l+1} = \frac{1}{2} \sum_{i=1}^{K} \left(y_i - z_i^l\right)^2, \quad i \in \{1, \ldots, n\}$
- **backward**: $\frac{\partial z_i^{l+1}}{\partial z^l} = y_i - z_i^l, \quad i \in \{1, \ldots, n\}$

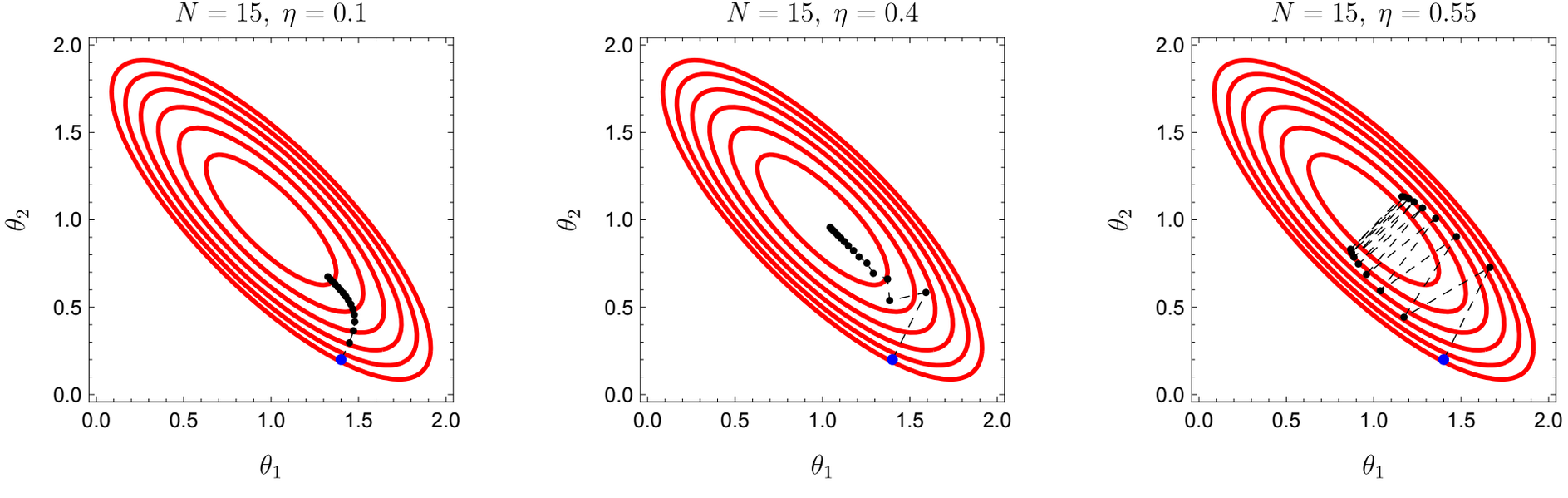
Gradient Descent

- Task: find parameters which minimize loss over the training dataset:

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, \mathcal{L}(\boldsymbol{\theta})$$

where $\boldsymbol{\theta}$ is a set of all parameters defining the ANN (e.g., all weight matrices)

- Gradient descent: $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta^{(t)} \nabla \mathcal{L}(\boldsymbol{\theta}^{(t)})$
where $\eta^{(t)} > 0$ is the **learning rate** or **step size** at iteration t



Batch, Online and Mini-Batch Learning

When to update weights?

- **(Full) Batch learning**: after all patterns are used (epoch)
  - inefficient for redundant datasets
- **Online learning**: after each training pattern
  - noise can help overcome local minima but can also harm the convergence in the final stages while fine-tuning
  - **Stochastic Gradient Descent (SGD)** does this
  - convergence *almost surely* to local minimum when $\eta^{(t)}$ decreases *appropriately* in time
- **Mini-batch learning**: after a small sample of training patterns

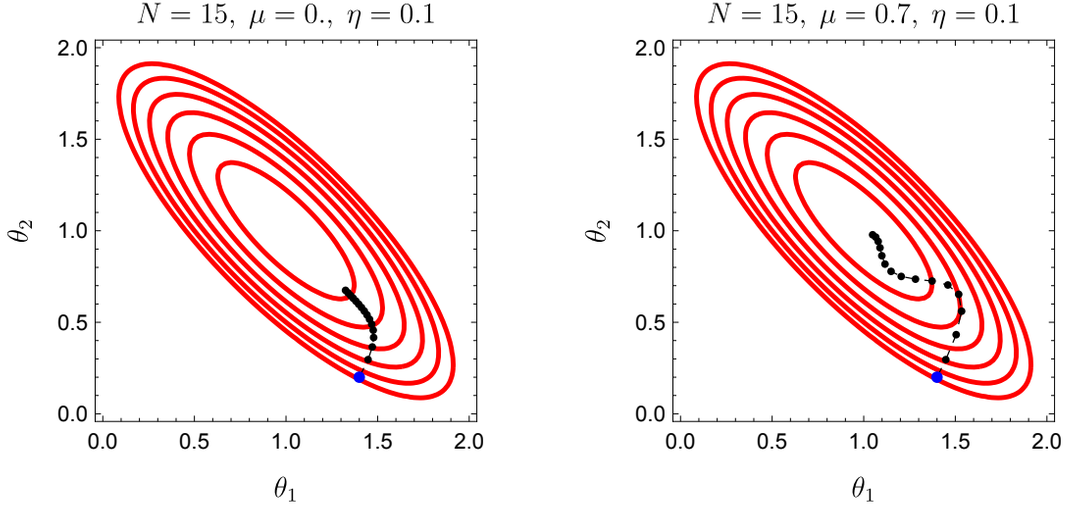
Momentum

- Simulate inertia to overcome plateaus in the error landscape:

$$\boldsymbol{v}^{(t+1)} = \mu \boldsymbol{v}^{(t)} - \eta^{(t)} \nabla \mathcal{L}(\boldsymbol{\theta}^{(t)})$$
$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + \boldsymbol{v}^{(t+1)}$$

  where $\mu \in [0, 1]$ is the *momentum parameter*

- Momentum damps oscillations in directions of high curvature
- It builds velocity in directions with consistent (possibly small) gradient

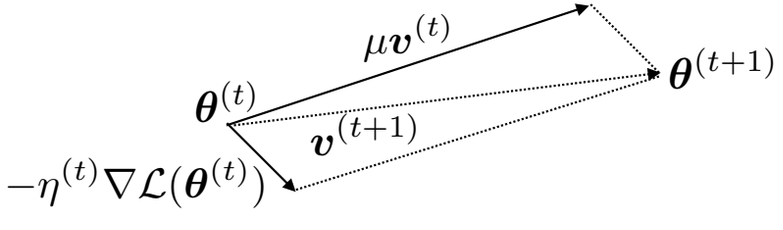


Nesterov's Momentum

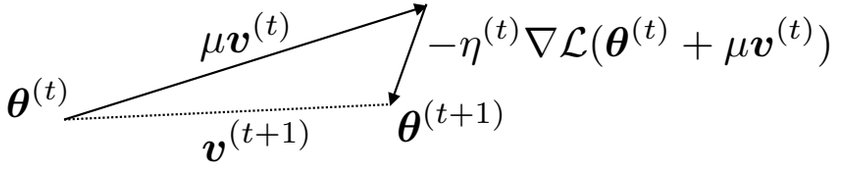
- Different approach by Nesterov (1983, convex optimization):

$$\boldsymbol{v}^{(t+1)} = \mu\boldsymbol{v}^{(t)} - \eta^{(t)}\nabla\mathcal{L}(\boldsymbol{\theta}^{(t)} + \mu\boldsymbol{v}^{(t)})$$
$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + \boldsymbol{v}^{(t+1)}$$

- While classic momentum *corrects* the velocity using gradient at $\boldsymbol{\theta}^{(t)}$, Nesterov uses $\boldsymbol{\theta}^{(t)} + \mu\boldsymbol{v}^{(t)}$ which is similar to $\boldsymbol{\theta}^{(t+1)}$
- Stronger theoretical convergence guarantees for convex functions
- Slightly better in practice
- For more details see Sutskever et al.: *On the importance of initialization and momentum in deep learning*, 2013



classic



Nesterov

Annealing the Learning Rate

- Decrease the learning rate in the course of optimization.
- **Step decay**: reduce the learning rate by a factor (e.g., $\frac{1}{2}$) every few iterations
- **Exponential decay**: set $\eta_t = \eta_0 e^{-kt}$ for the iteration t
- **Hyperbolic decay**: set $\eta_t = \frac{\eta_0}{1+kt}$

Resilient Propagation (Rprop)

- Motivation: a magnitude of gradient differs a lot for different Parameters
- Rprop (Riedmiller and Braun, 1992) does not use gradient value - the step size for each weight is adapted using its *sign*, only

1 **for** each θ_i

2 **if** $\frac{\partial \mathcal{L}^{(t-1)}}{\partial \theta_i} \cdot \frac{\partial \mathcal{L}^{(t)}}{\partial \theta_i} > 0$

3 $\Delta_i^{(t)} = \min\left(\Delta_i^{(t-1)} \cdot \eta^+, \Delta_{max}\right)$

4 **elseif** $\frac{\partial \mathcal{L}^{(t-1)}}{\partial \theta_i} \cdot \frac{\partial \mathcal{L}^{(t)}}{\partial \theta_i} < 0$

5 $\Delta_i^{(t)} = \max\left(\Delta_i^{(t-1)} \cdot \eta^-, \Delta_{min}\right)$

6 $\theta_i^{(t+1)} = \theta_i^{(t)} - \text{sign}\left(\frac{\partial \mathcal{L}^{(t)}}{\partial \theta_i}\right) \cdot \Delta_i^{(t)}$

where the step size $\Delta_i^{(t)} \in [\Delta_{min}, \Delta_{max}]$ and $0 < \eta^- < 1 < \eta^+$

- Typically order of magnitude faster than basic Gradient Descent
- Does not work well for mini-batches
- Igel and Hüsken: *Improving the Rprop Learning Algorithm*, 2000

Adagrad

- Adaptive Gradient method
- Idea: reduce learning rates for parameters having high values of gradient

$$g_i^{(t+1)} = g_i^{(t)} + \left(\frac{\partial \mathcal{L}}{\partial \theta_i^{(t)}} \right)^2$$

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \frac{\eta}{\sqrt{g_i^{(t+1)} + \epsilon}} \cdot \frac{\partial \mathcal{L}}{\partial \theta_i^{(t)}}$$

- g_i accumulates squared partial derivatives w.r.t. to the parameter θ_i
- ϵ is a small positive number to prevent division by zero
- Weakness: ever increasing g_i leads to slow convergence eventually

RMSProp

- Similar to Adagrad but employs a moving average:

$$g_i^{(t+1)} = \gamma g_i^{(t)} + (1-\gamma)\left(\frac{\partial \mathcal{L}}{\partial \theta_i^{(t)}}\right)^2$$

- γ is a *decay* parameter (typical value $\gamma = 0.9$)
- Unlike for Adagrad updates do not get infinitesimally small

Adam (Adaptive Moment Estimation)

- Kingma and Ba: *Adam: A Method for Stochastic Optimization*, 2014

$$m_i^{(t+1)} = \beta_1 m_i^{(t)} + (1-\beta_1)\frac{\partial \mathcal{L}}{\partial \theta_i^{(t)}} \qquad \hat{m}_i^{(t+1)} = \frac{m_i^{(t+1)}}{1-\beta_1 m_i^{(t)}}$$

$$v_i^{(t+1)} = \beta_2 v_i^{(t)} + (1-\beta_2)\left(\frac{\partial \mathcal{L}}{\partial \theta_i^{(t)}}\right)^2 \qquad \hat{v}_i^{(t+1)} = \frac{v_i^{(t+1)}}{1-\beta_2 v_i^{(t)}}$$

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \eta \frac{\hat{m}_i^{(t+1)}}{\sqrt{\hat{v}_i^{(t+1)}} + \epsilon}$$

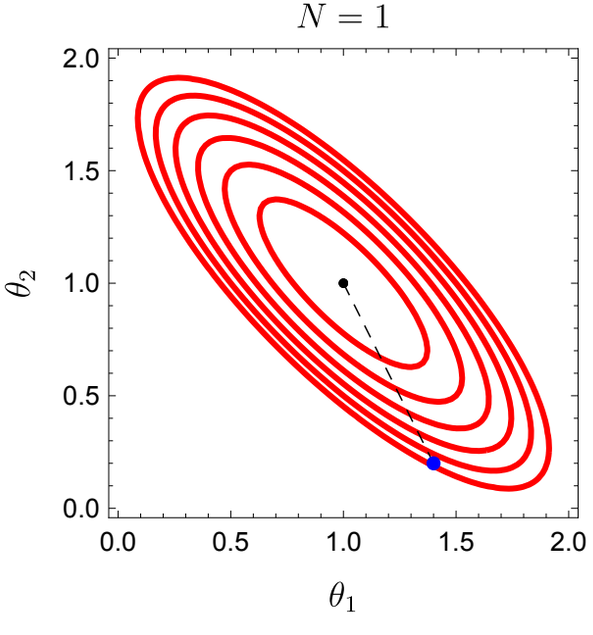
- m_i and v_i are first and second raw moment estimates of gradient (mean and uncentered variance)
- $\hat{m}_i$ and $\hat{v}_i$ are corrections of m_i and v_i as these are biased to zero due to initialization $m_i = v_i = 0$
- β_1 and β_2 are *decay* parameters

Second Order Methods

- Newton's method for optimization:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \left(\nabla^2 \mathcal{L}(\boldsymbol{\theta}^{(t)})\right)^{-1} \nabla \mathcal{L}(\boldsymbol{\theta}^{(t)})$$

- No parameters needed
- Need to compute the Hessian matrix and invert it!
- Approximations, e.g., L-BFGS are not suitable for mini-batches



Regularization

- How to deal with overfitting?
  - get more data
  - find simpler model, search for optimal architecture, e.g., number, type and size of layers
  - use *regularization*
- Most types of regularization are based on penalties for model complexity
- Bayesian point of view: introduce prior distribution on model parameters

L2 Regularization

- Recall the solution for the linear regression $\boldsymbol{w}^* = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\boldsymbol{y}$
- What if $\mathbf{X}^T\mathbf{X}$ has no inverse?
- We can modify the solution by adding a small element to the diagonal:

$$\boldsymbol{w}^* = \left(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}\right)^{-1}\mathbf{X}^T\boldsymbol{y}, \quad \lambda > 0$$

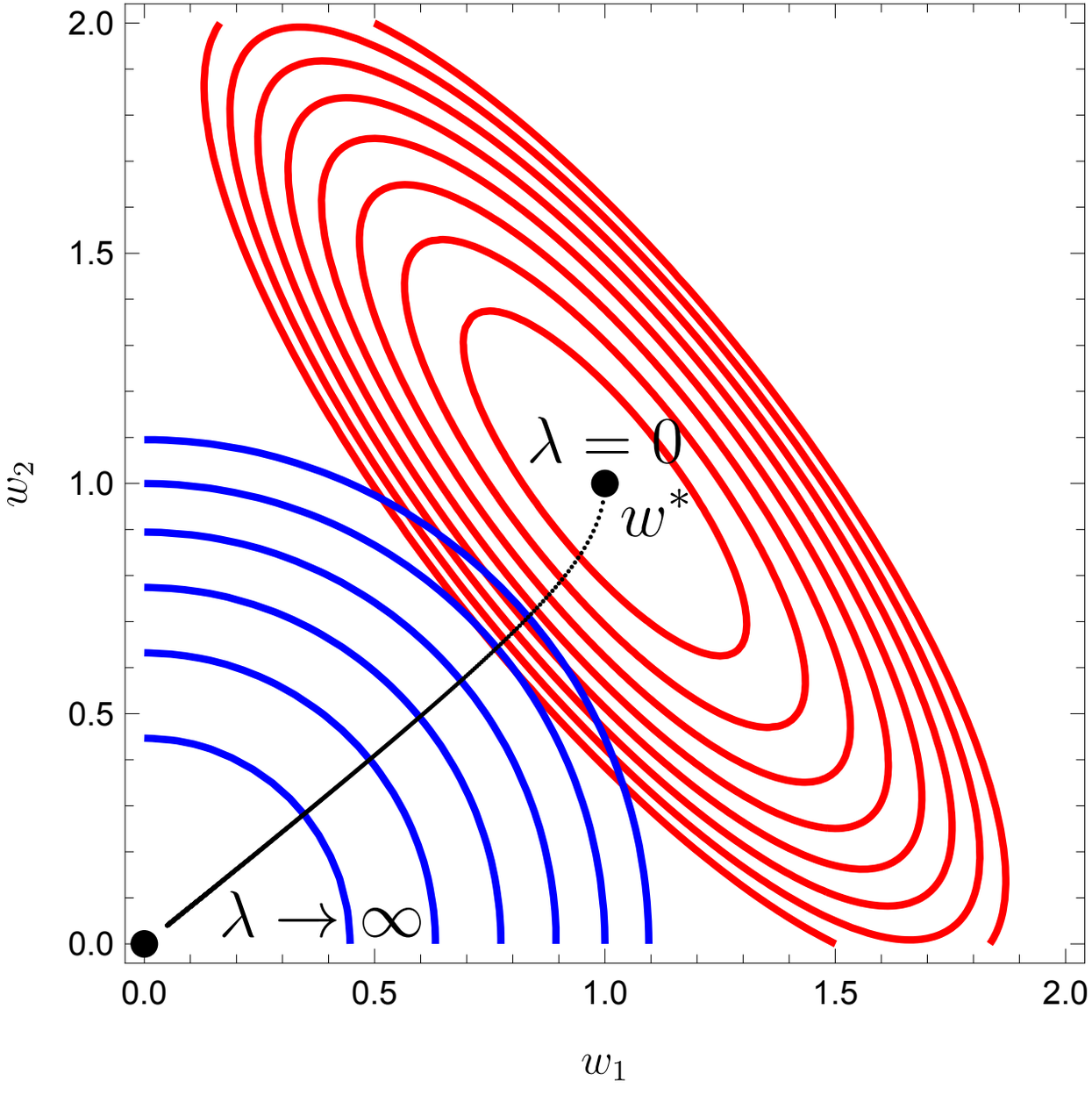
- It turns out that this approach no only helps with inverting $\mathbf{X}^T\mathbf{X}$ but it also improves model generalization
- It is the solution of the *regularized* loss function:

$$\mathcal{L}\left(\boldsymbol{w}\right) = \left(\boldsymbol{y} - \mathbf{X}\boldsymbol{w}\right)^T\left(\boldsymbol{y} - \mathbf{X}\boldsymbol{w}\right) + \lambda\boldsymbol{w}^T\boldsymbol{w}$$

  this one is called the **L2 regularization**, see seminar for the derivation
- The term $\lambda\boldsymbol{w}^T\boldsymbol{w} = \lambda\left\|\boldsymbol{w}\right\|_2^2$ minimizes the size of the weight vector
- Note that we omit bias in $\lambda\boldsymbol{w}^T\boldsymbol{w}$

L2 Regularization (contd.)

- L2 regularization is also called the ridge regression or the **weight decay**



L2 Regularization as Gaussian Prior

- Recall the likelihood:

$$p\left(\boldsymbol{y}|\boldsymbol{w},\mathbf{X}\right)=\left(2\pi\sigma^{2}\right)^{-\frac{m}{2}}e^{-\frac{1}{2\sigma^{2}}(\boldsymbol{y}-\mathbf{X}\boldsymbol{w})^{T}(\boldsymbol{y}-\mathbf{X}\boldsymbol{w})}$$

- Define a Gaussian prior with zero mean and variance σ_0^2 for the parameters:

$$p\left(\boldsymbol{w}\right)=\left(2\pi\sigma_{0}^{2}\right)^{-\frac{1}{2}}e^{-\frac{1}{2\sigma_{0}^{2}}\boldsymbol{w}^{T}\boldsymbol{w}}$$

- Then the posterior is:

$$p(\boldsymbol{w}|\boldsymbol{y},\mathbf{X})=\frac{p(\boldsymbol{y}|\boldsymbol{w},\mathbf{X})\cdot p(\boldsymbol{w})}{p(\boldsymbol{y}|\mathbf{X})}$$

The denominator does not depend on the parameters $\boldsymbol{w}$:

$$p(\boldsymbol{w}|\boldsymbol{y},\mathbf{X})\propto p(\boldsymbol{y}|\boldsymbol{w},\mathbf{X})\cdot p(\boldsymbol{w})$$

MAP Estimate

- Maximizing $p(\boldsymbol{w}|\boldsymbol{y}, \mathbf{X})$ gives us the Maximum a posteriori (MAP) estimate:

$$\boldsymbol{w}_{MAP} = \underset{\boldsymbol{w}}{\operatorname{argmax}}\, p(\boldsymbol{w}|\boldsymbol{y}, \mathbf{X}) = \underset{\boldsymbol{w}}{\operatorname{argmin}}\, (-\log p(\boldsymbol{w}|\boldsymbol{y}, \mathbf{X}))$$

where

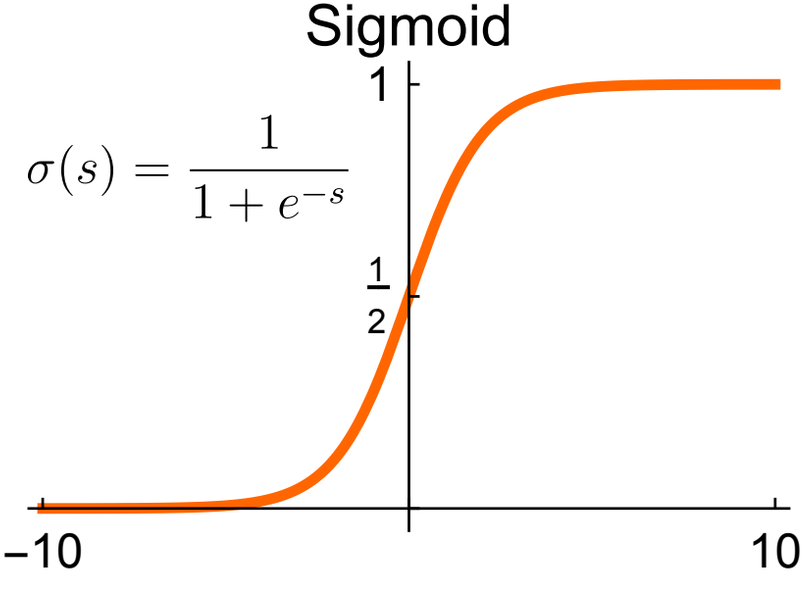
$$-\log p(\boldsymbol{w}|\boldsymbol{y}, \mathbf{X}) = \frac{1}{2\sigma^2} (\boldsymbol{y} - \mathbf{X}\boldsymbol{w})^T (\boldsymbol{y} - \mathbf{X}\boldsymbol{w}) + \frac{1}{2\sigma_0^2} \boldsymbol{w}^T \boldsymbol{w} + C$$

- We can omit C, define $\lambda = \dfrac{\sigma^2}{\sigma_0^2}$ and minimize the loss function we already know:

$$\mathcal{L}(\boldsymbol{w}) = (\boldsymbol{y} - \mathbf{X}\boldsymbol{w})^T (\boldsymbol{y} - \mathbf{X}\boldsymbol{w}) + \lambda \boldsymbol{w}^T \boldsymbol{w}$$

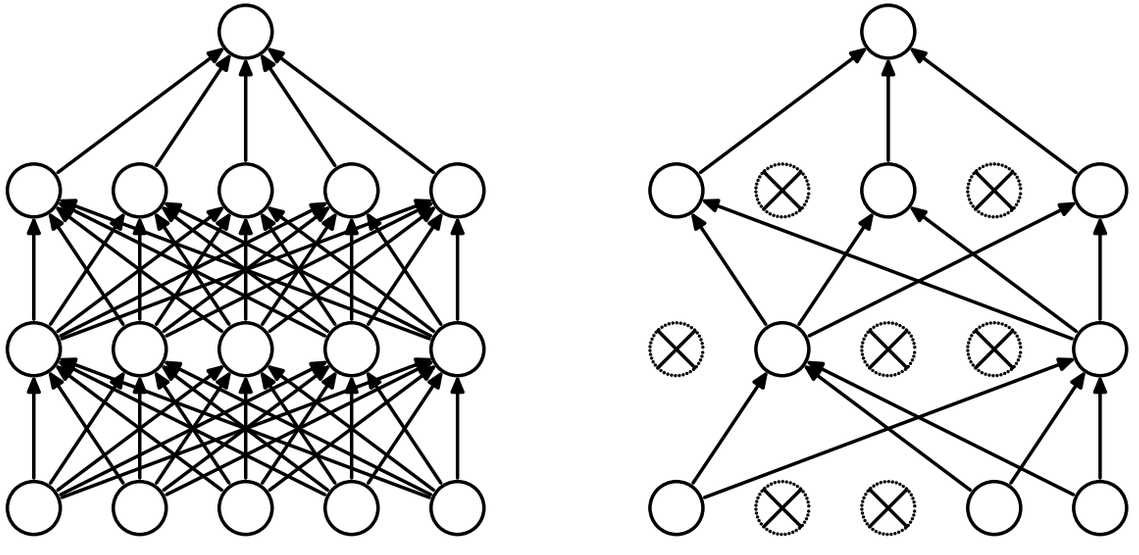
Weight Decay Discussion

- Having zero mean Gaussian prior keeps the weights smaller
- Weight decay is widely used for most types of layers in ANNs
- Intuition: sigmoid-like neurons kept near zero potential (via small weights) behave similarly to linear neurons
- The same works for other models, e.g., polynomial regression
- λ is usually set using cross validation



Dropout

- Idea: average many neural networks, share weights to make this computationally feasible
- For each training example omit a unit with probability p (often $p = 0.5$)
- This is like sampling from 2^U networks where U is the number of units
- Typically only a small amount of 2^U networks is actually sampled
- Prevents coadaptation of feature detectors



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

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

Dropout (contd.)

- How to make predictions with networks using dropout?
- Scale outputs (output weights) of all affected units by p
- For a linear unit taking inputs from the dropout layer we have:

$$s = \sum_{i=1}^{n} w_i \delta_i x_i, \quad p(\delta_i|p) = \mathrm{Ber}(\delta_i|p)$$

$$\mathbb{E}(s) = \sum_{i=1}^{n} w_i \mathbb{E}(\delta_i) x_i = p \sum_{i=1}^{n} w_i x_i$$

  where the expectation is computed over all 2^n configurations

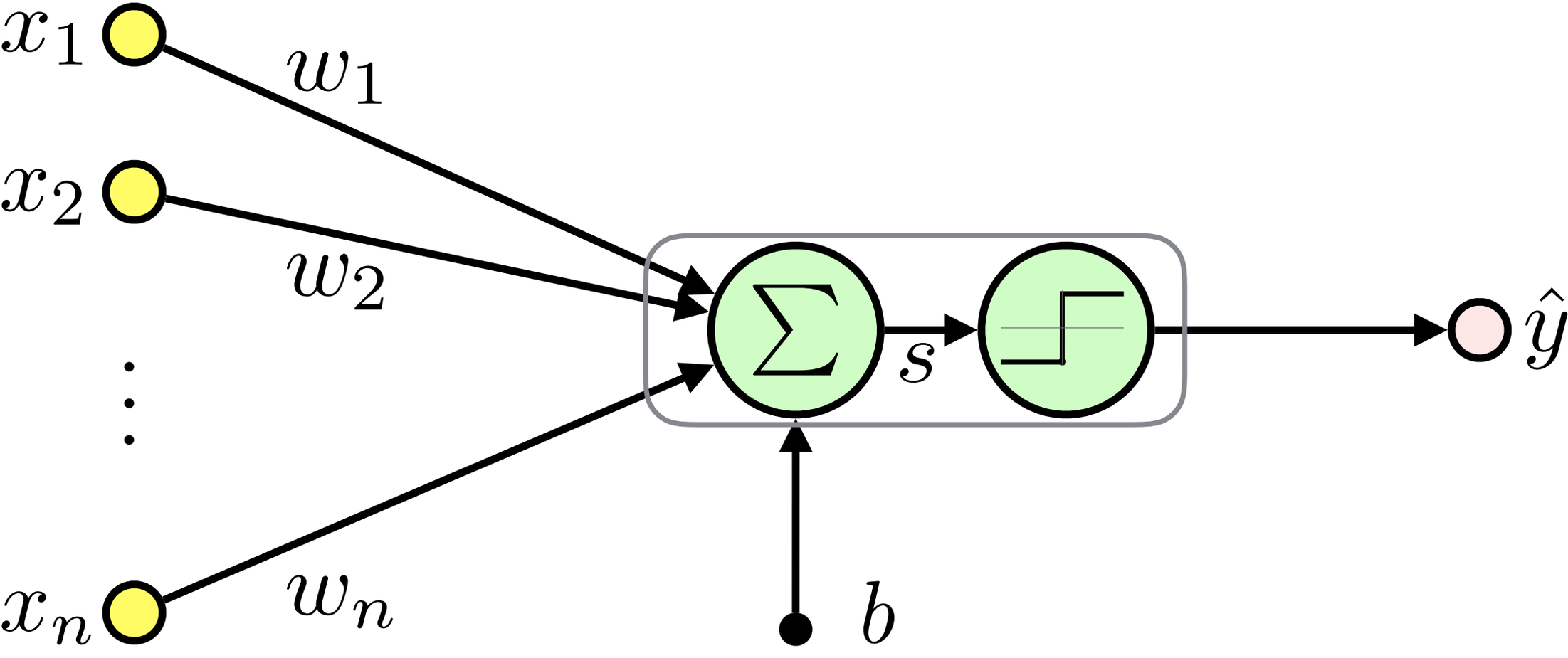
- For general neural networks we still get a good approximation of the expectation when scaling by p
- See Baldi and Sadowski: *The Dropout Learning Algorithm*, 2014
- When used for inputs higher values of p are suggested

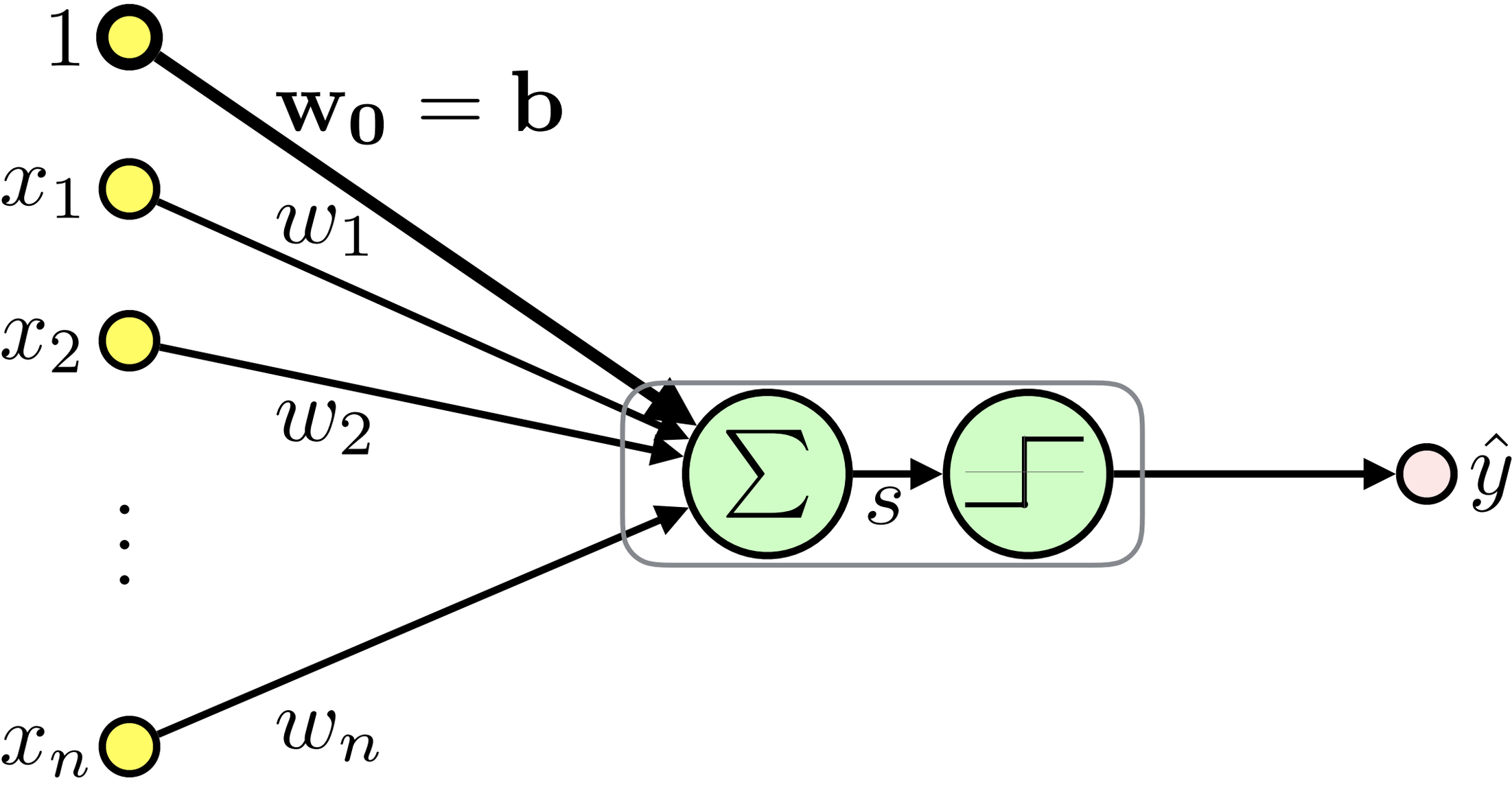
Other Regularization Approaches

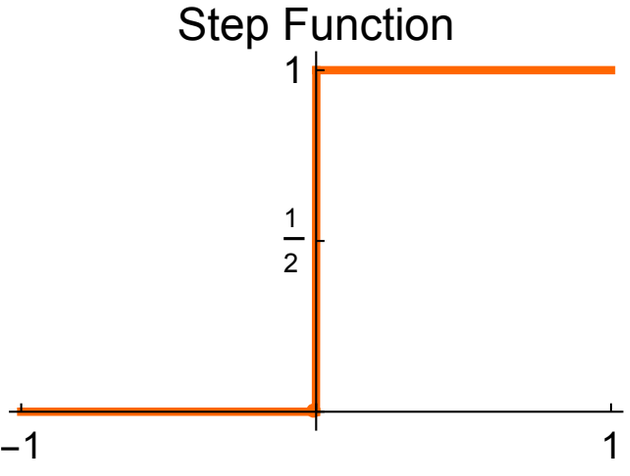
- L1 regularization: sum absolute values, i.e., use $\lambda \|\boldsymbol{w}\|_1$
- Early stopping: start with small weights, stop when validation loss starts to grow
- Randomize inputs: same as the weight decay for linear neurons
- Noisy weights
- DropConnect: connection-based dropout
- Other weight sharing approaches: Convolutional Neural Networks
- Model averaging

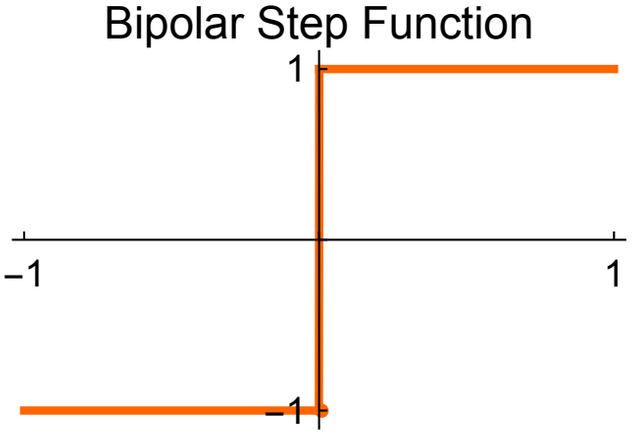
Next Lecture

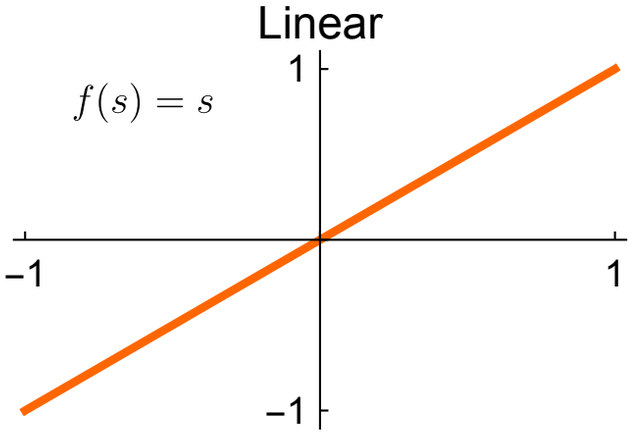
- Deep Neural Networks
- Convolutional Neural Networks
- Autoencoders
- Transfer learning

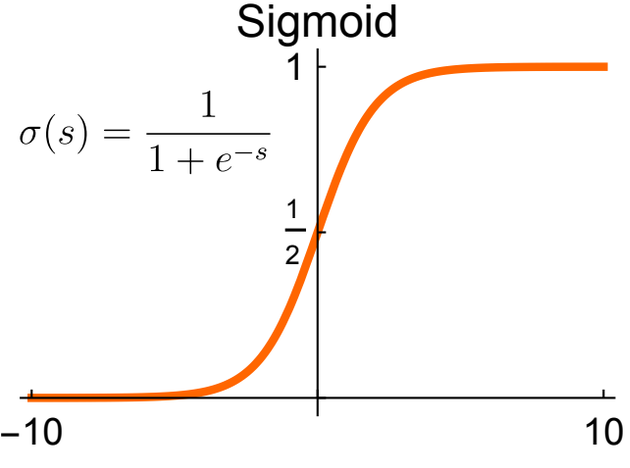

x_1
w_1
x_2
w_2
x_n
w_n
Σ
s
b
$\hat{y}$

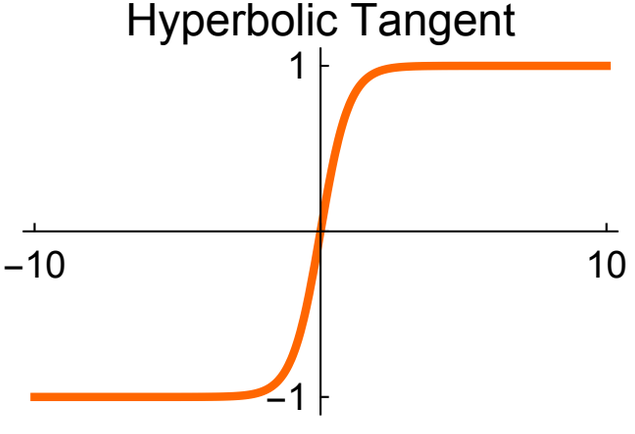

1
$\mathbf{w_0 = b}$
x_1
w_1
x_2
w_2
x_n
w_n
$\sum$
s
$\hat{y}$

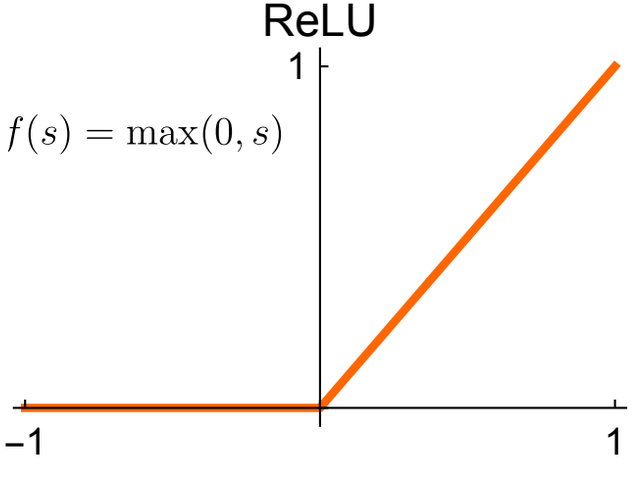

Step Function
1
1/2
−1
1

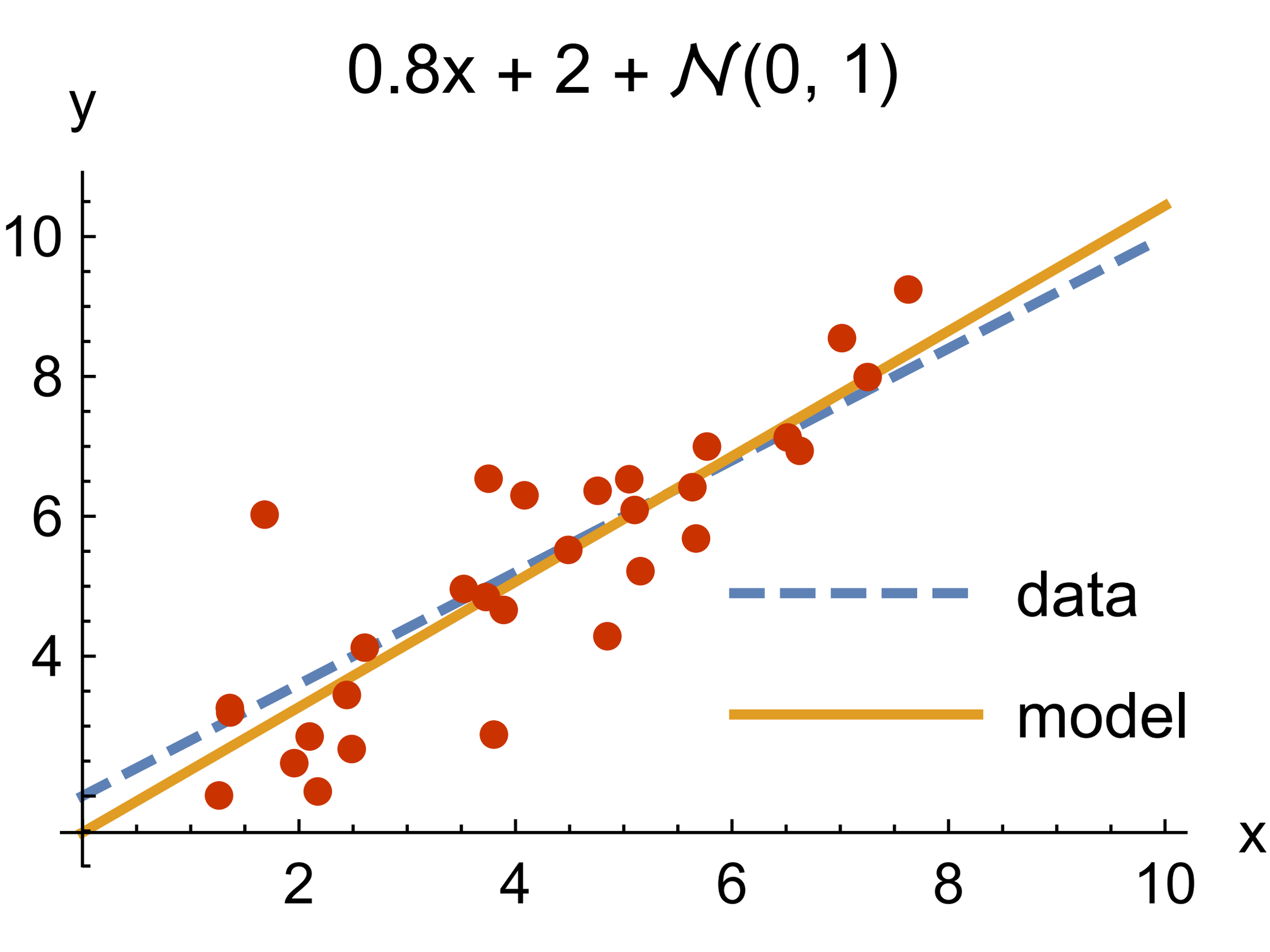

Bipolar Step Function
1
−1
1
−1

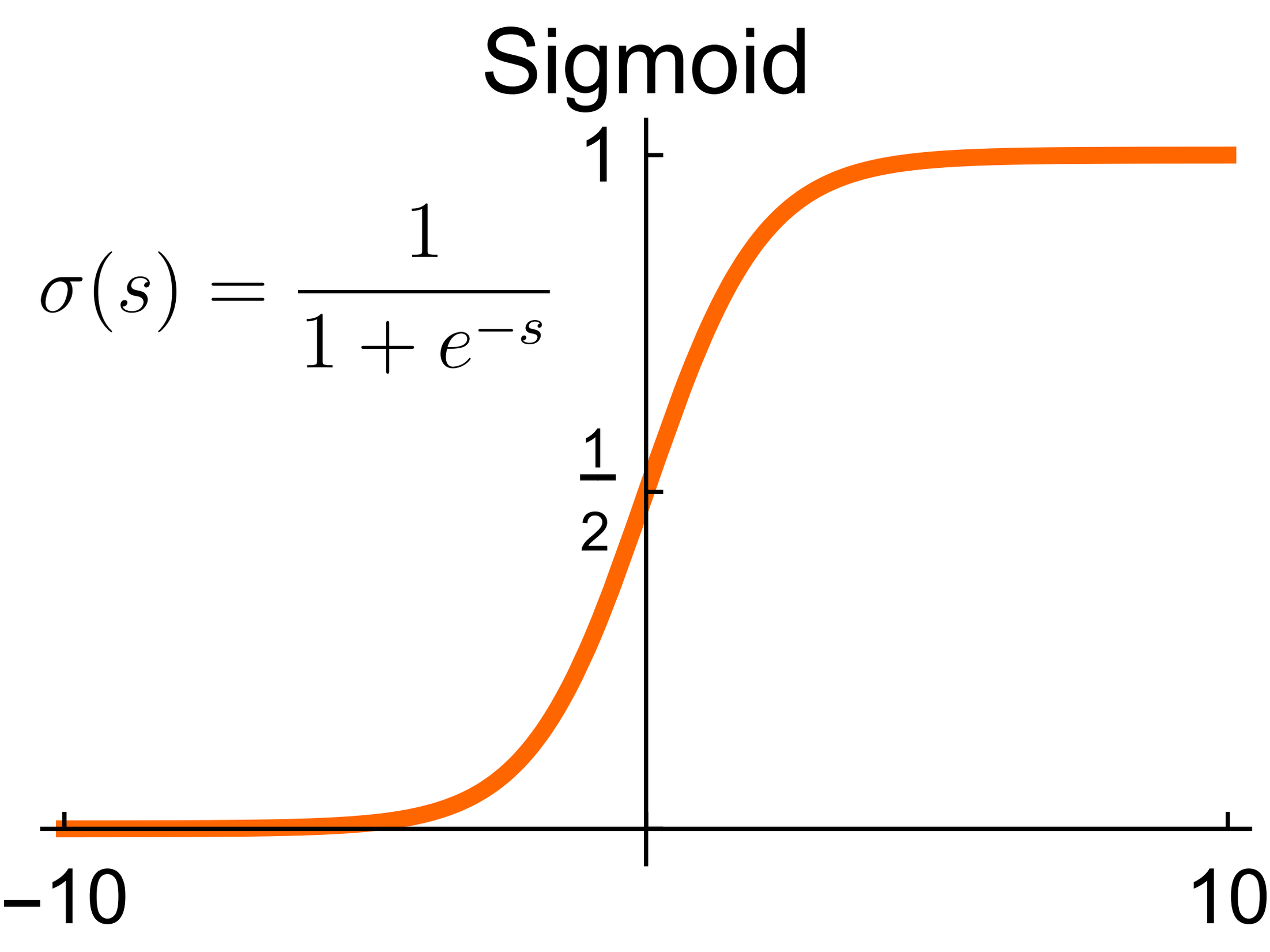

Linear
$f(s) = s$
1
−1
1
−1

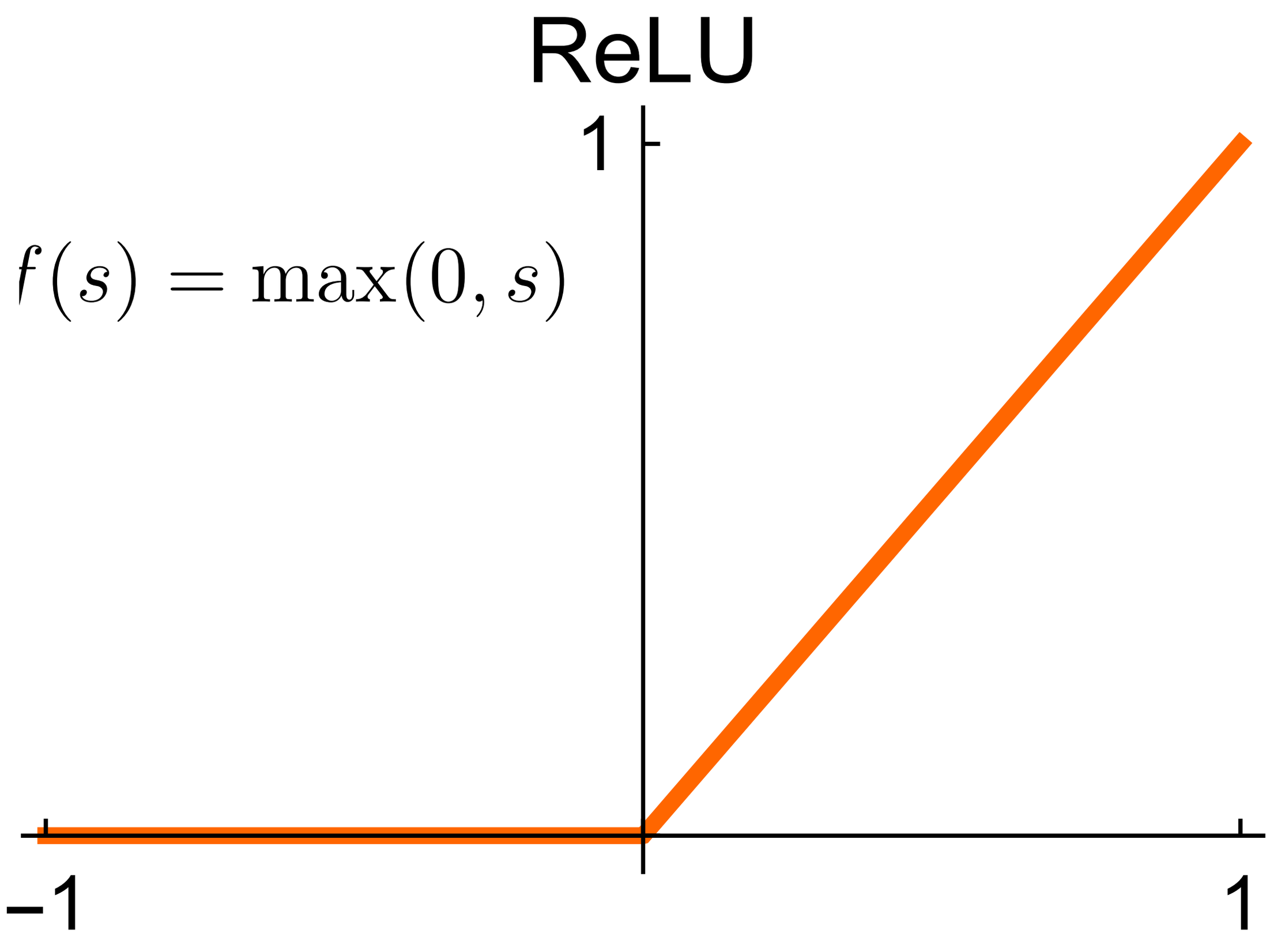

Sigmoid
$\sigma(s) = \frac{1}{1+e^{-s}}$
1
1/2
−10
10

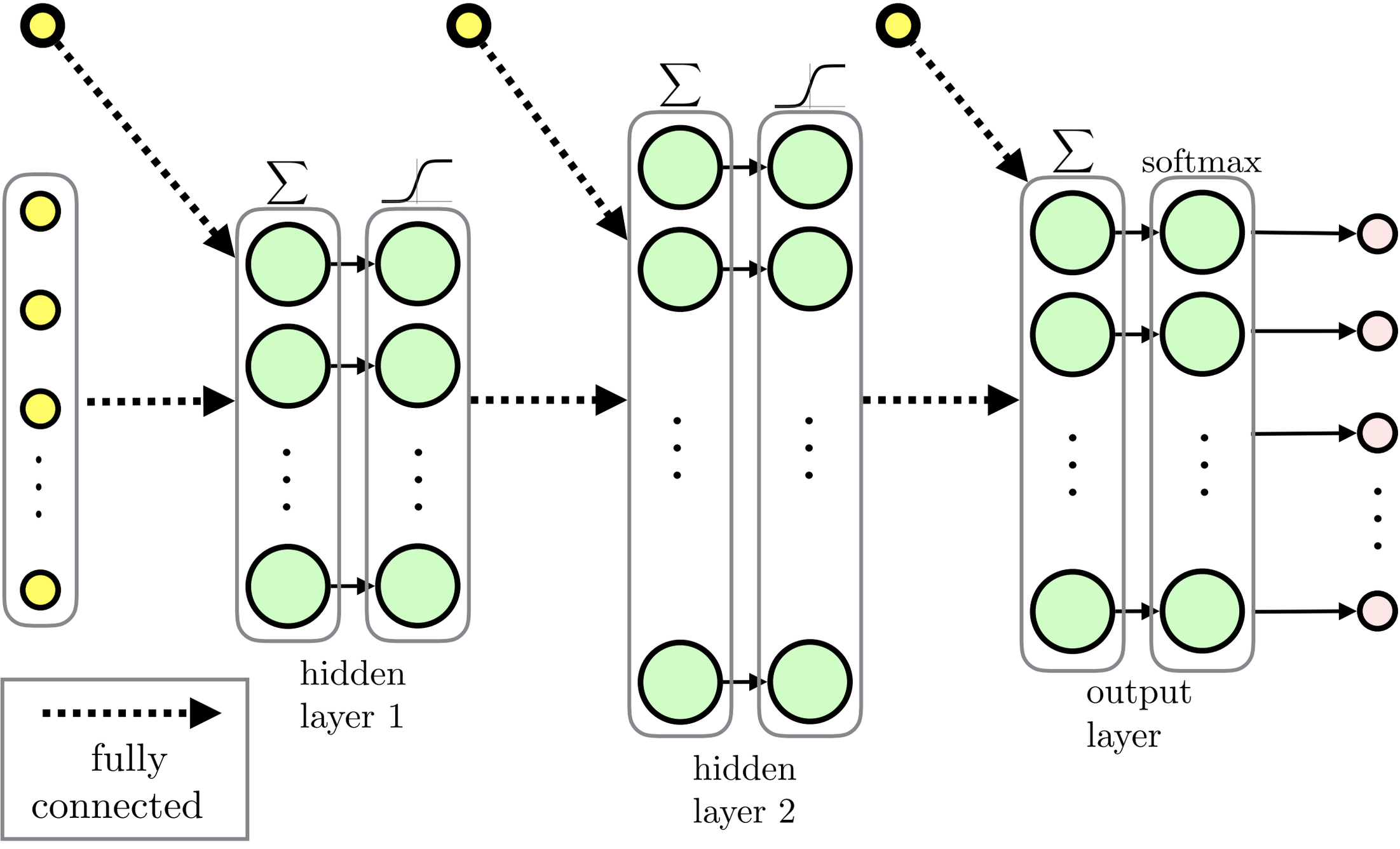

Hyperbolic Tangent
1
−10
10
−1

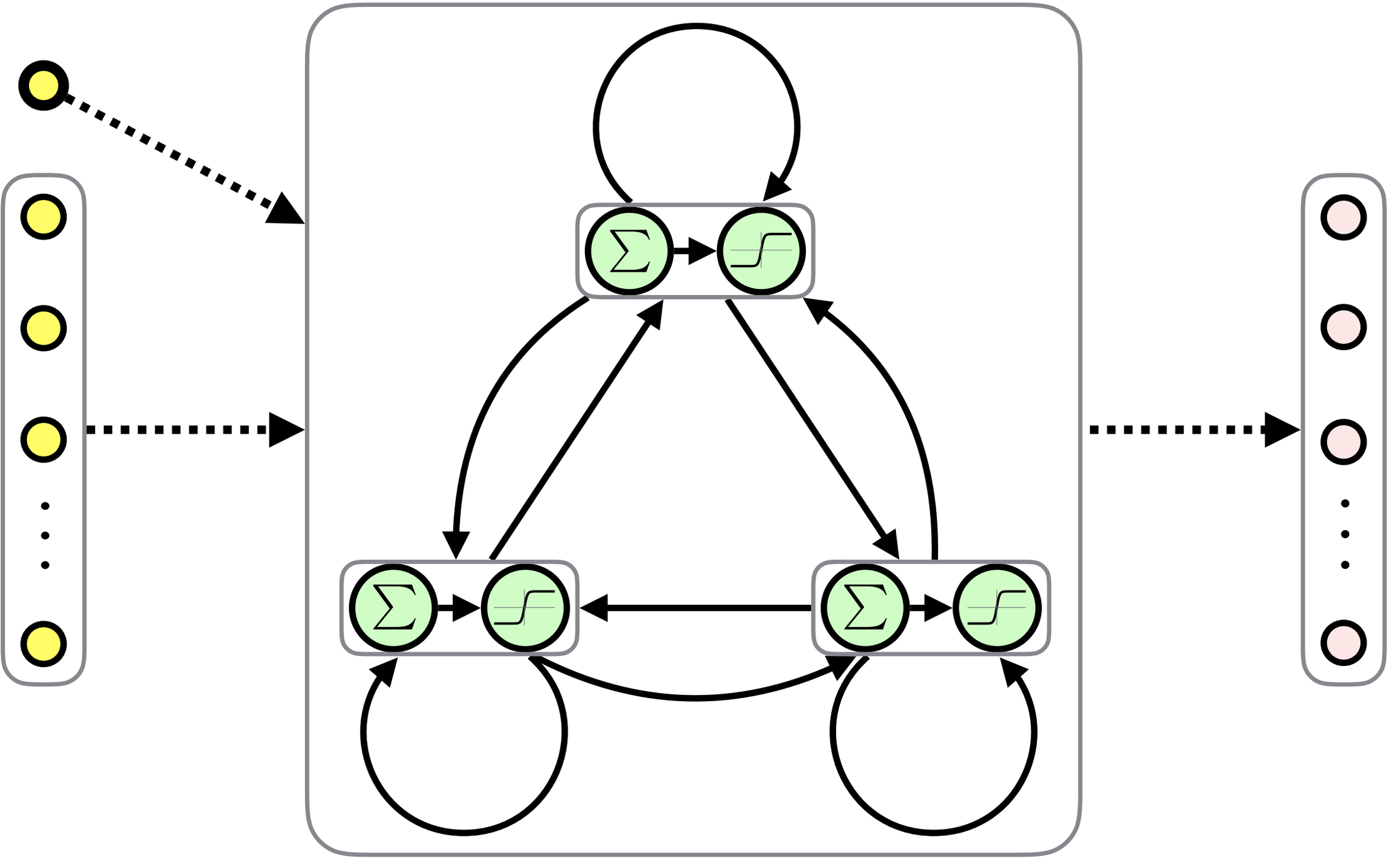

ReLU
$f(s) = \max(0, s)$
1
−1
1

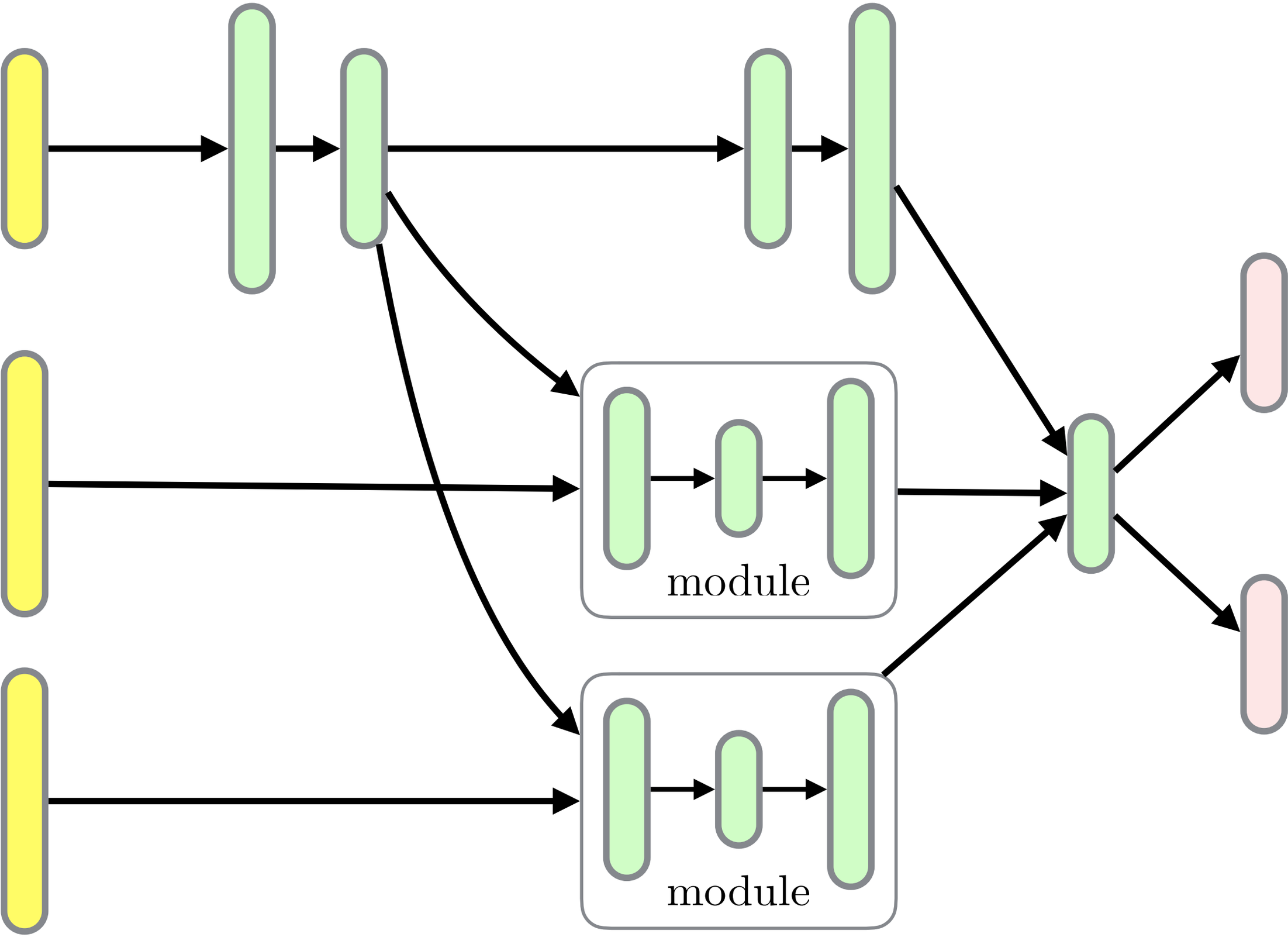

0.8x + 2 + 𝒩(0, 1)
y
10
8
6
4
2
4
6
8
10
x
data
model

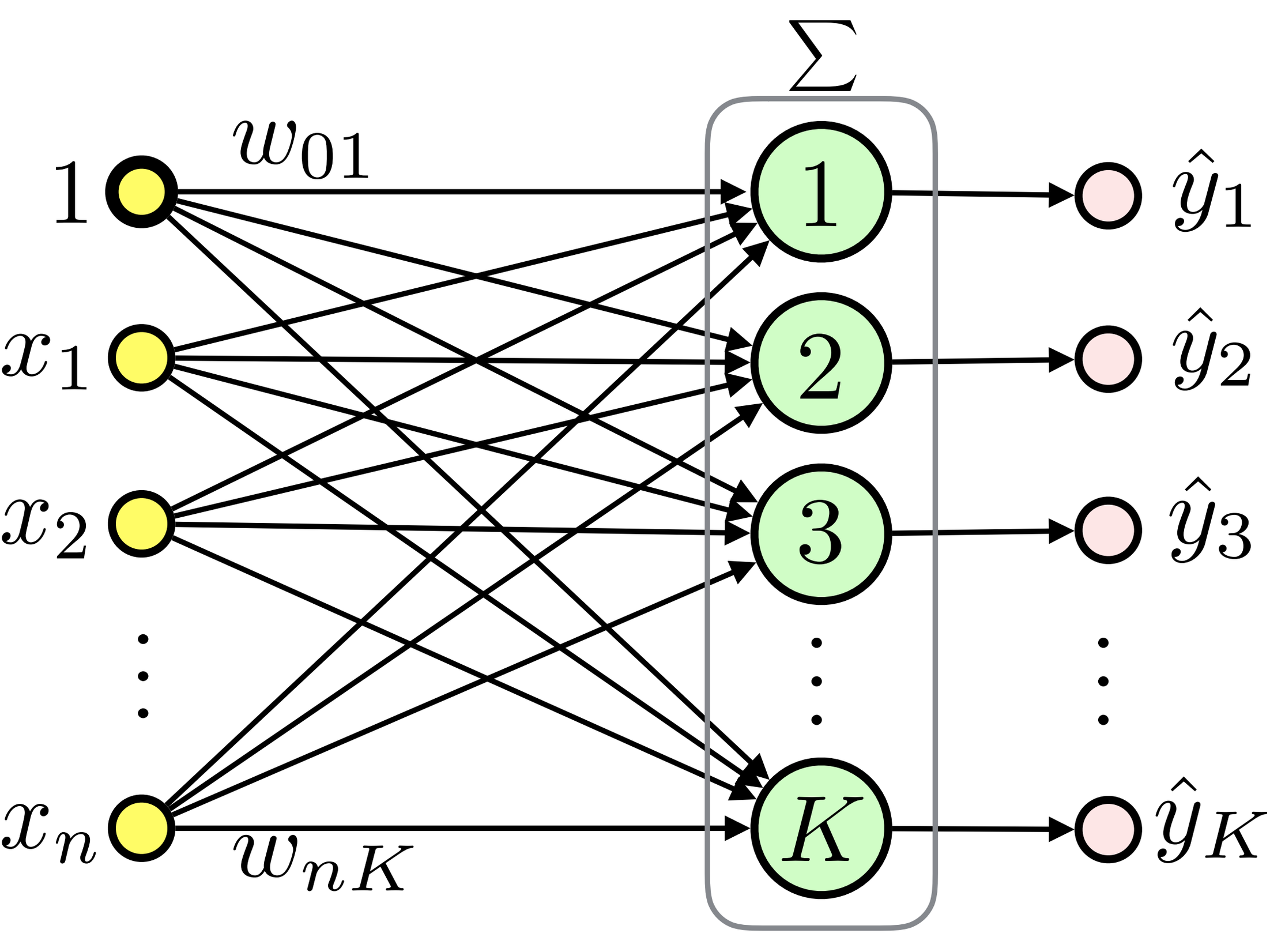

Sigmoid
$\sigma(s) = \frac{1}{1+e^{-s}}$
1
$\frac{1}{2}$
−10
10

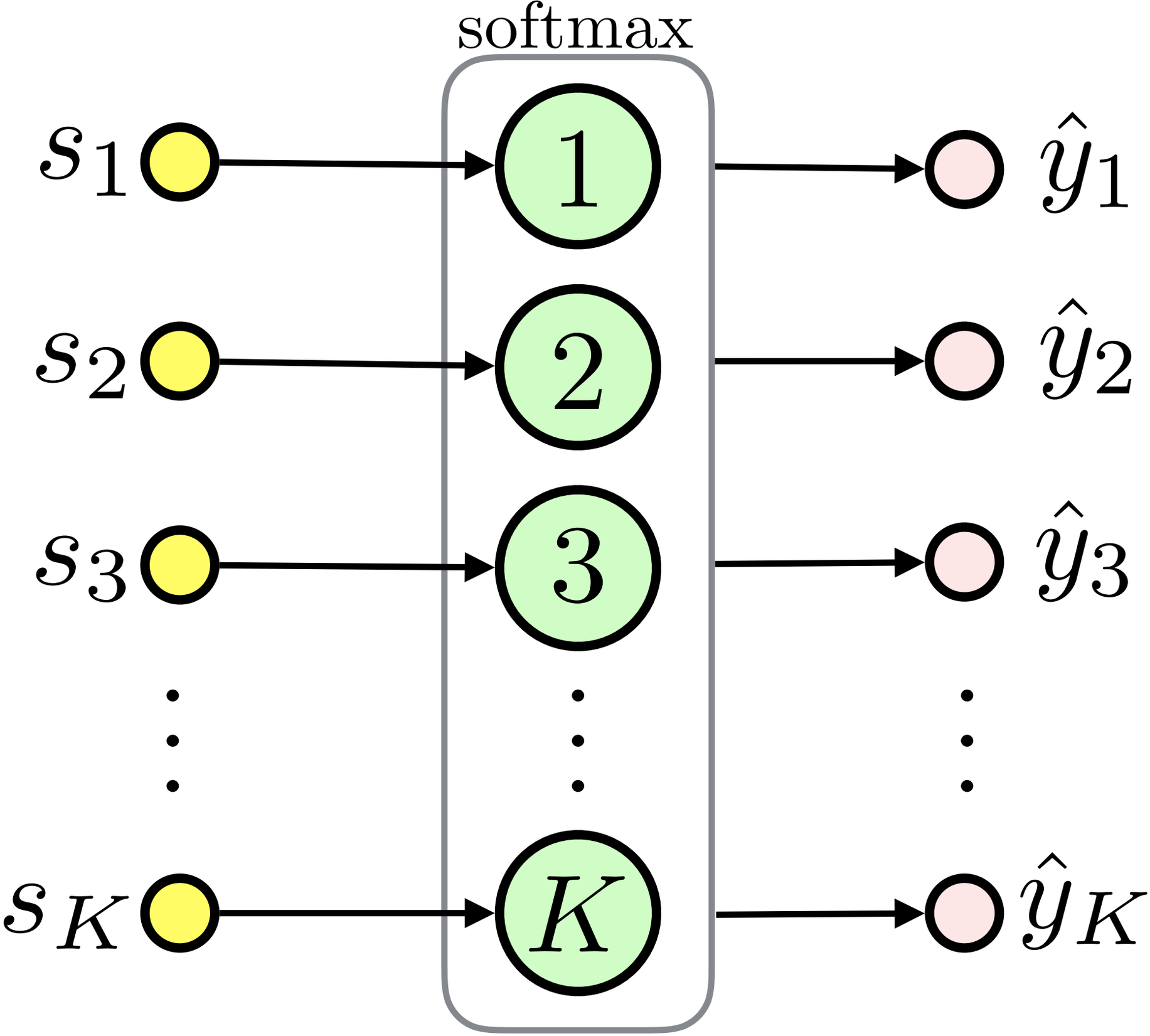

ReLU
$f(s) = \max(0, s)$
1
−1
1

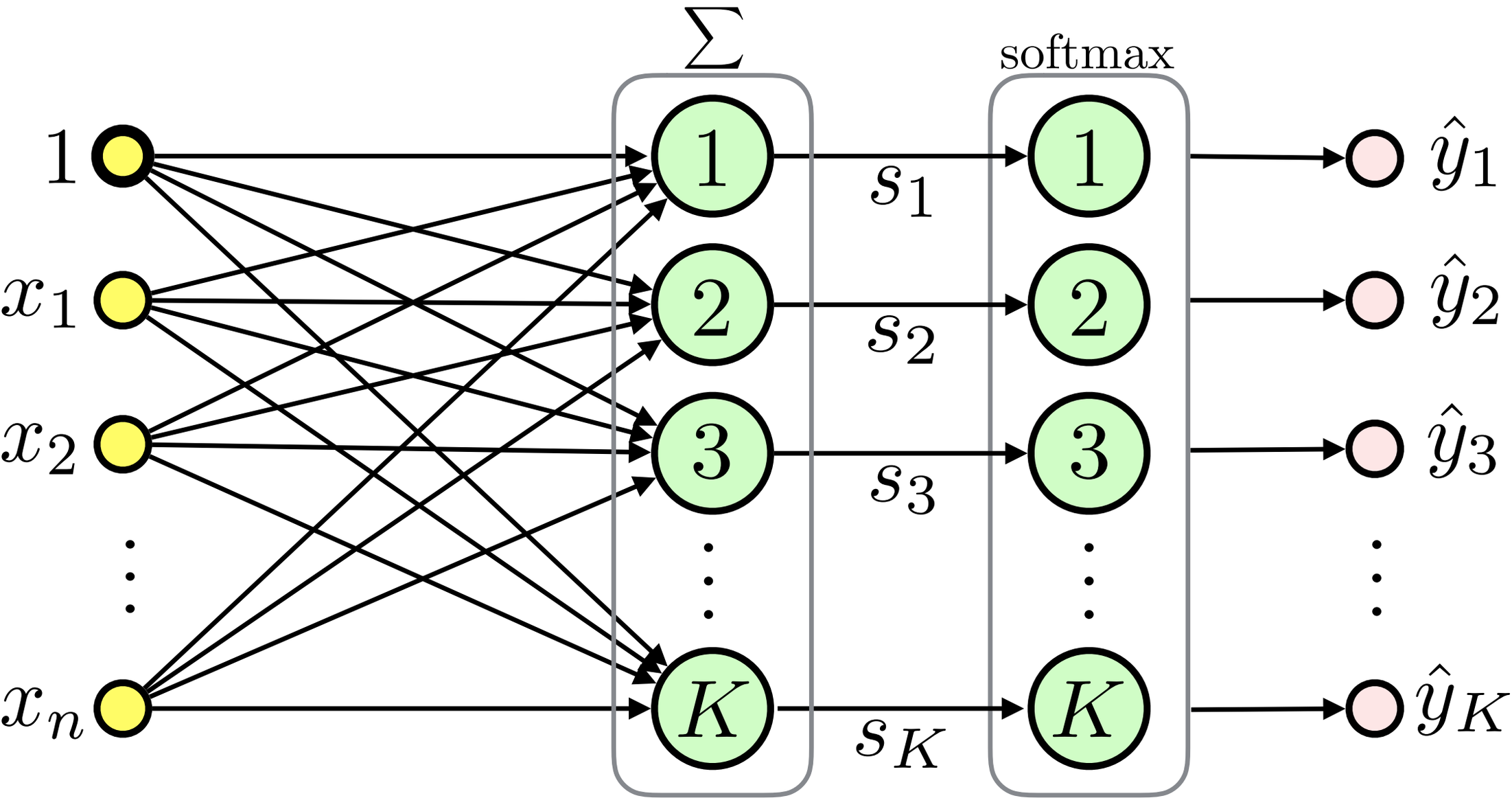

Σ
Σ
Σ
softmax
hidden
layer 1
hidden
layer 2
output
layer
fully
connected

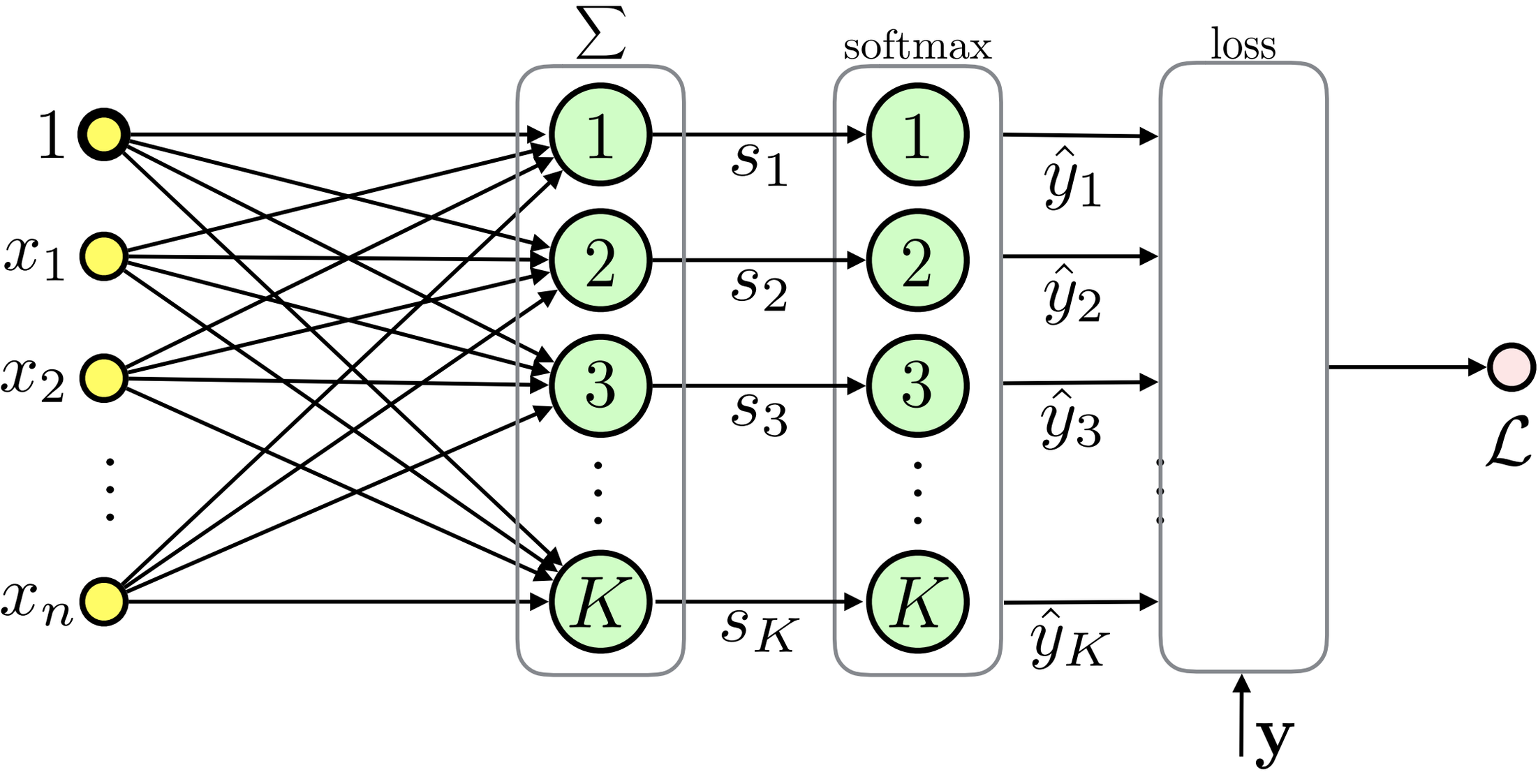


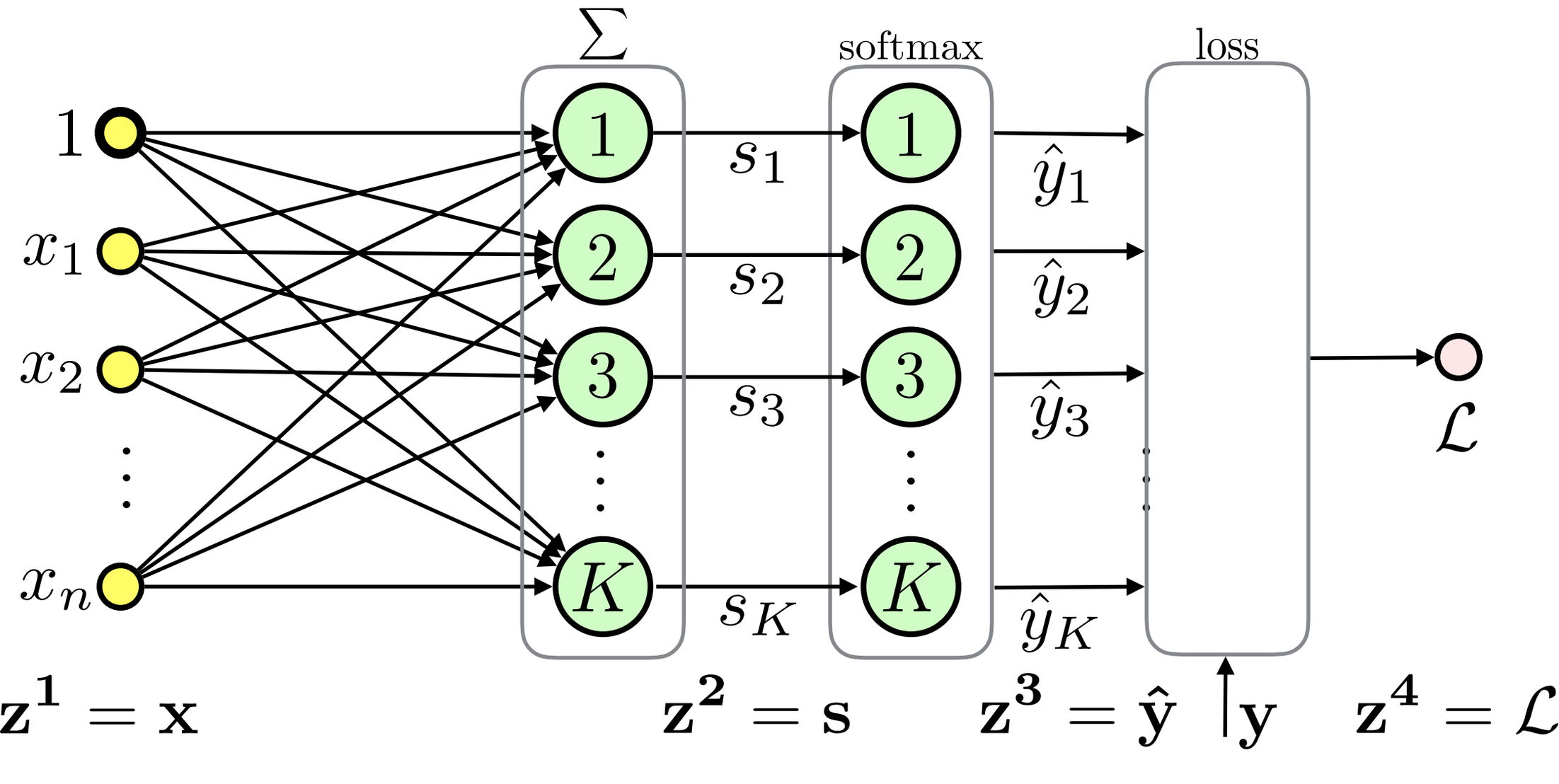

module
module

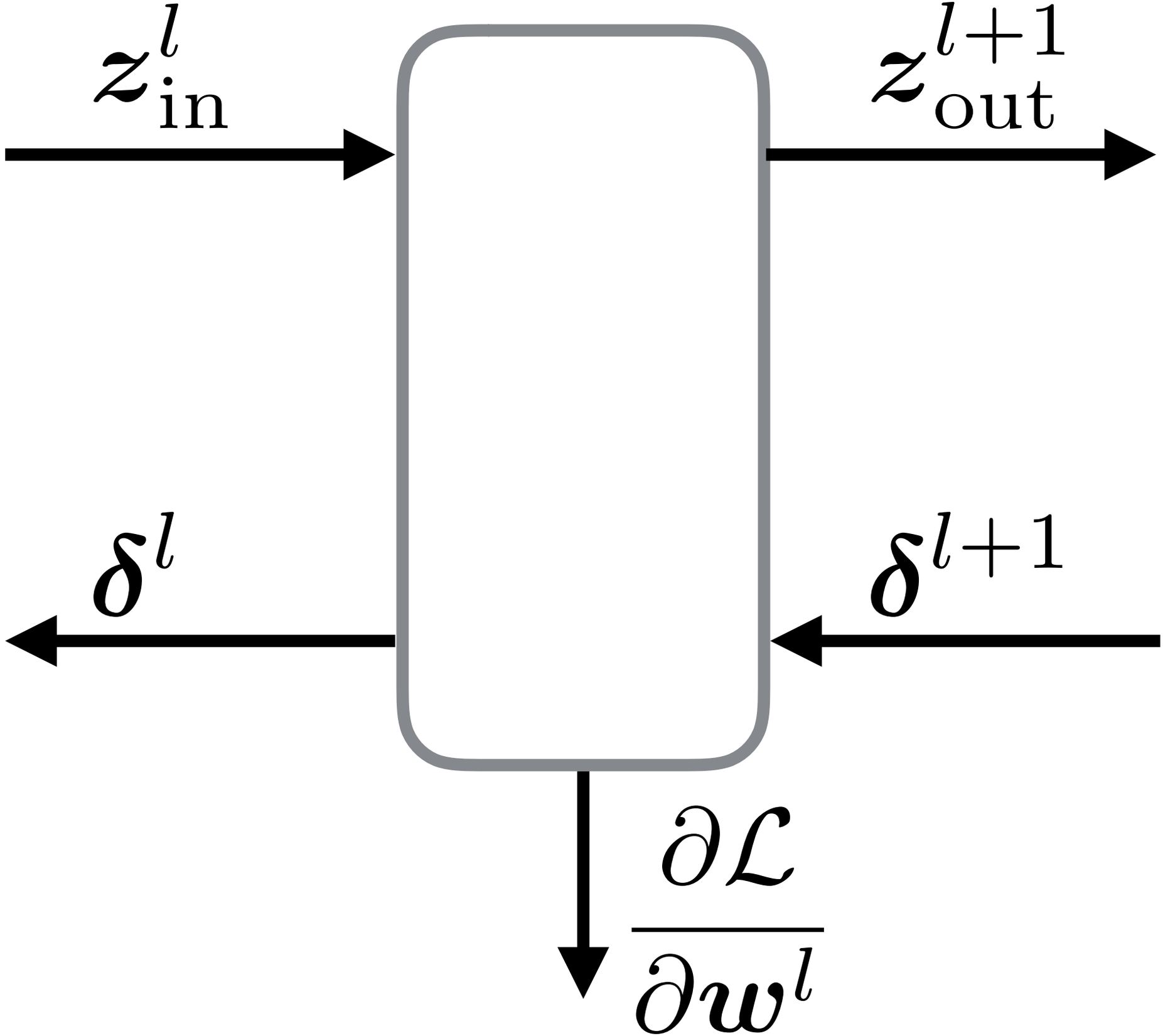

$\sum$
w_{01}
1
x_1
x_2
x_n
w_{nK}
1
2
3
K
$\hat{y}_1$
$\hat{y}_2$
$\hat{y}_3$
$\hat{y}_K$

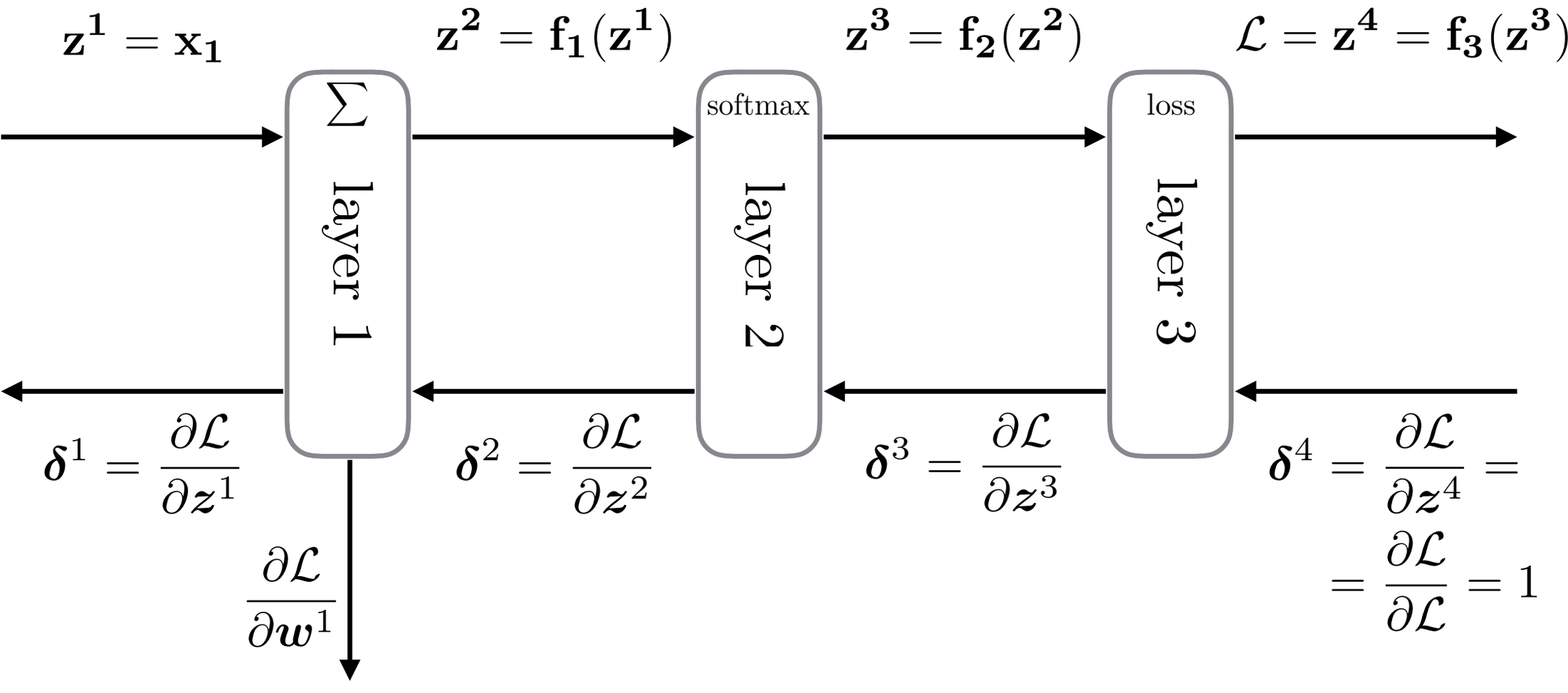

softmax
s_1
s_2
s_3
s_K
1
2
3
K
$\hat{y}_1$
$\hat{y}_2$
$\hat{y}_3$
$\hat{y}_K$

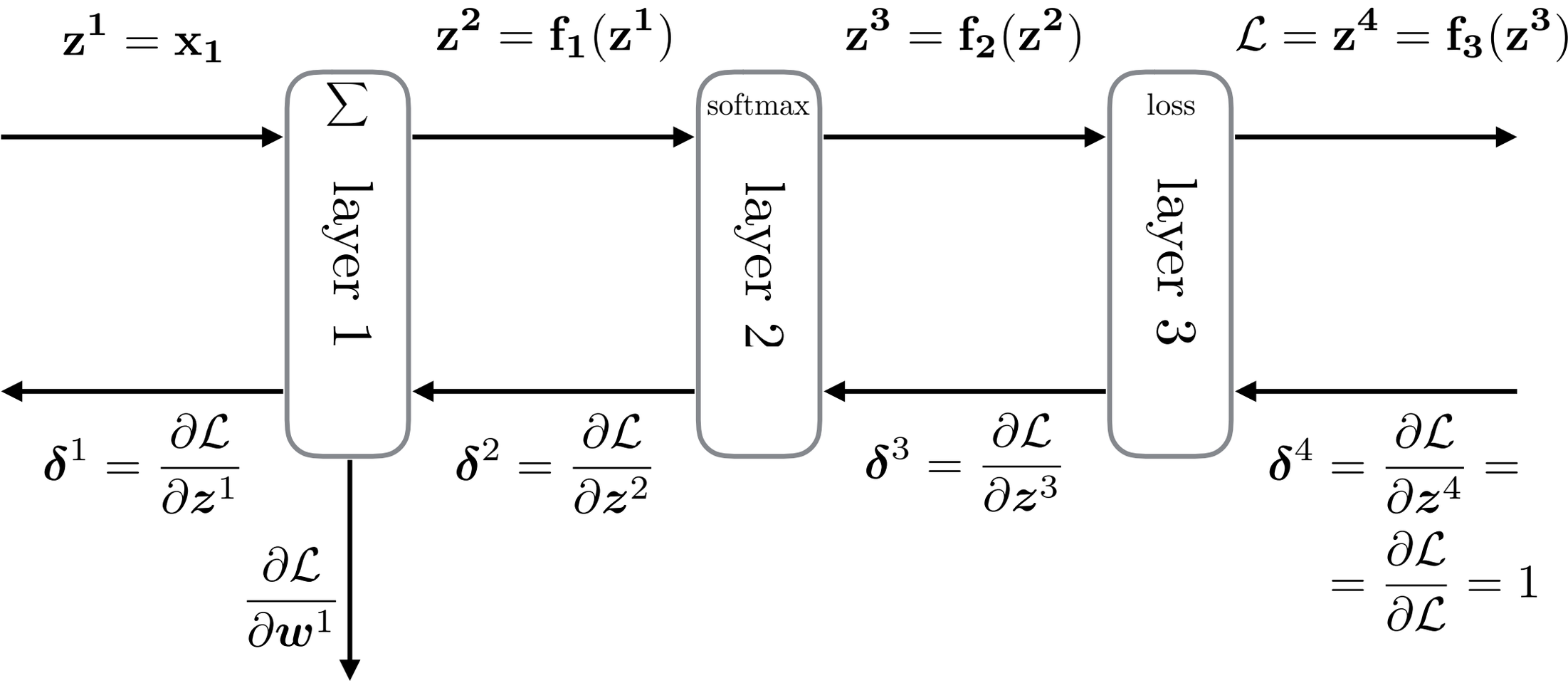

$\sum$
softmax
1
x_1
x_2
x_n
1
2
3
K
s_1
s_2
s_3
s_K
1
2
3
K
$\hat{y}_1$
$\hat{y}_2$
$\hat{y}_3$
$\hat{y}_K$

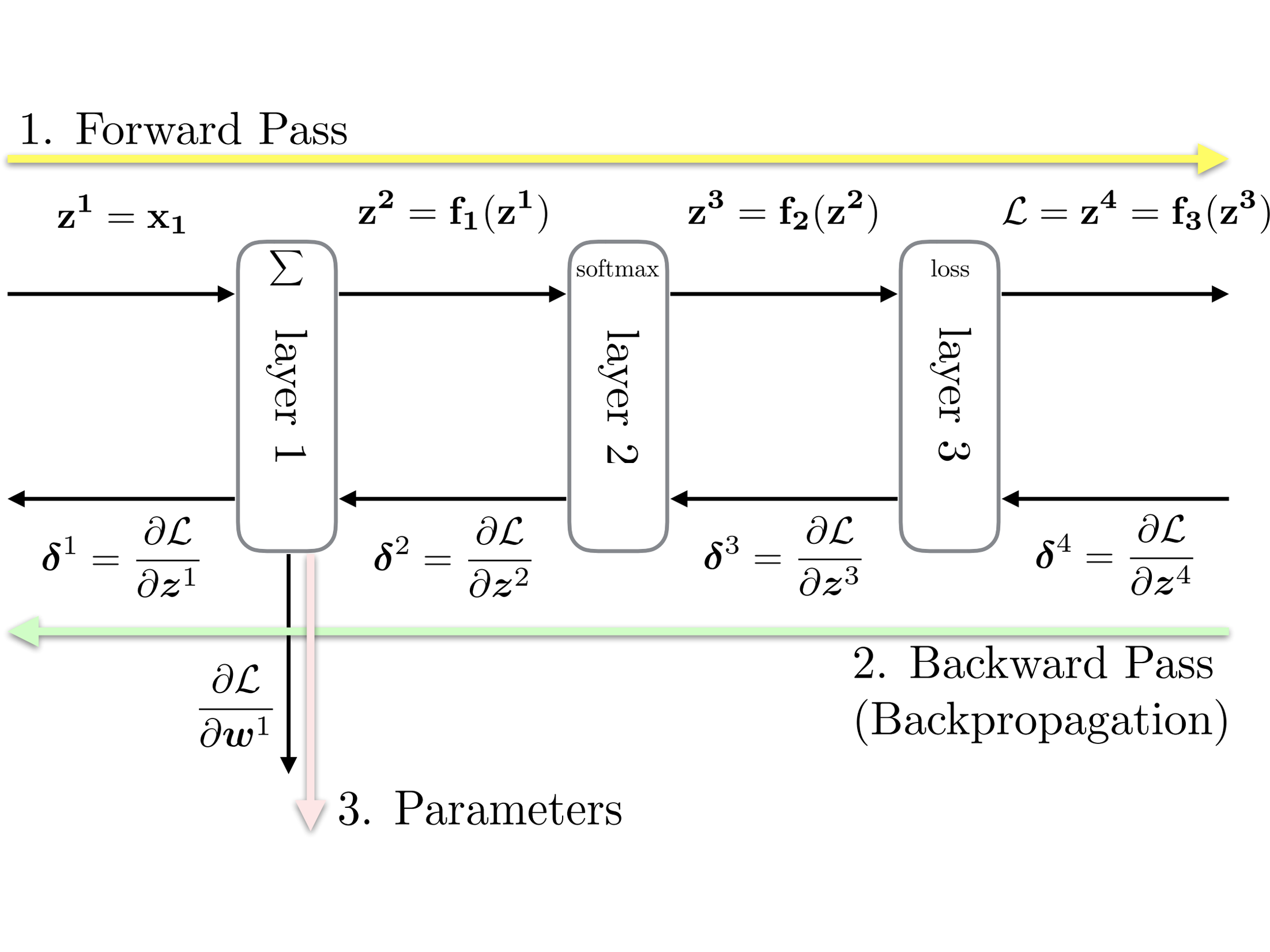

$\sum$
softmax
loss
1
x_1
x_2
x_n
1
2
3
K
s_1
s_2
s_3
s_K
1
2
3
K
$\hat{y}_1$
$\hat{y}_2$
$\hat{y}_3$
$\hat{y}_K$
$\mathcal{L}$
$\mathbf{y}$

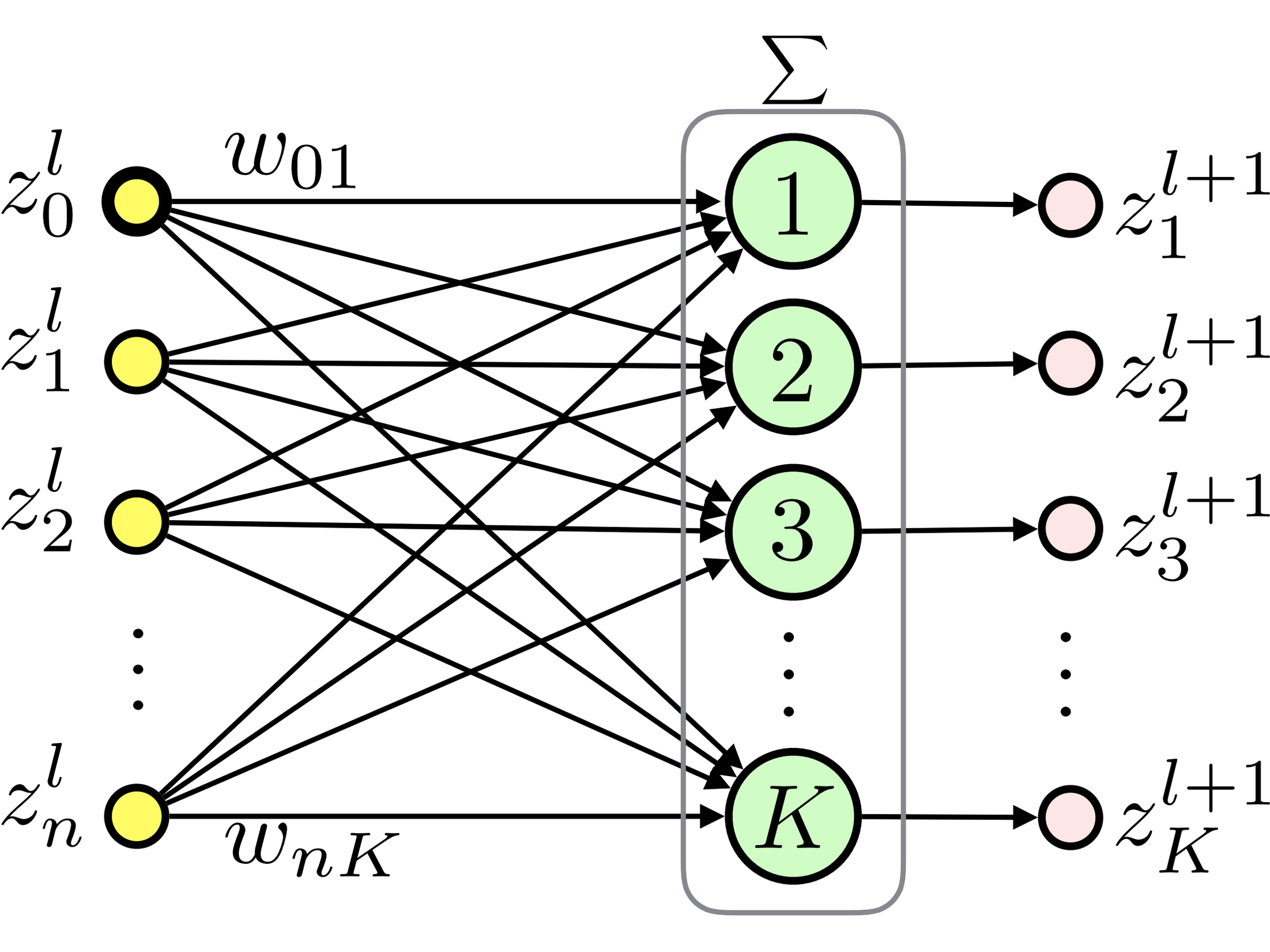

$\sum$
softmax
loss
1
x_1
x_2
x_n
1
2
3
K
s_1
s_2
s_3
s_K
1
2
3
K
$\hat{y}_1$
$\hat{y}_2$
$\hat{y}_3$
$\hat{y}_K$
$\mathcal{L}$
$\mathbf{z^1 = x}$
$\mathbf{z^2 = s}$
$\mathbf{z^3 = \hat{y}}$
$\mathbf{y}$
$\mathbf{z^4 = \mathcal{L}}$

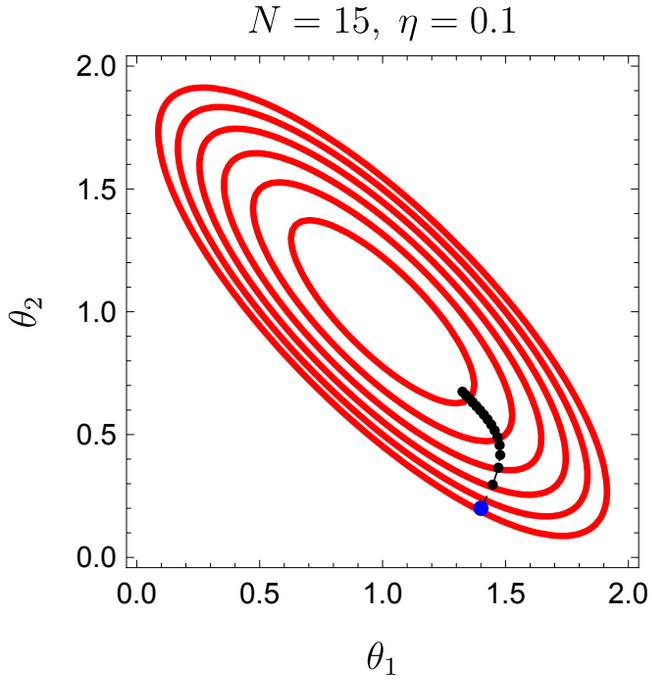

z_{in}^{l}
z_{out}^{l+1}
δ^{l}
δ^{l+1}
$\frac{\partial \mathcal{L}}{\partial w^{l}}$

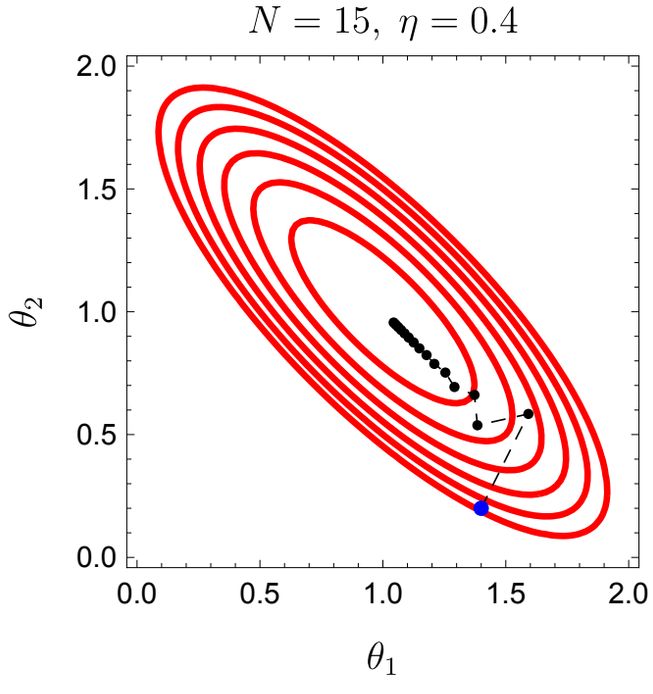

$\mathbf{z^1 = x_1}$
$\mathbf{z^2 = f_1(z^1)}$
$\mathbf{z^3 = f_2(z^2)}$
$\mathcal{L} = \mathbf{z^4 = f_3(z^3)}$
$\sum$ layer 1
softmax
layer 2
loss
layer 3
$\boldsymbol{\delta}^1 = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^1}$
$\boldsymbol{\delta}^2 = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^2}$
$\boldsymbol{\delta}^3 = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^3}$
$\boldsymbol{\delta}^4 = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^4} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} = 1$
$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}^1}$

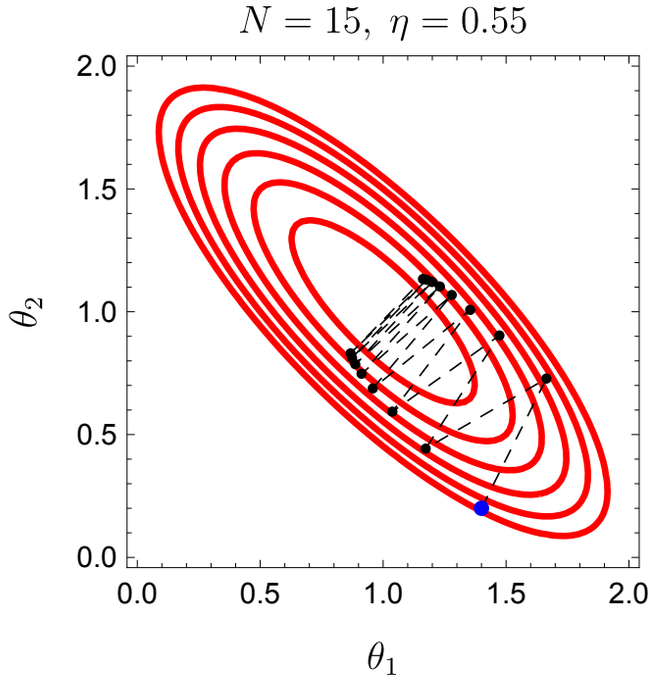

$\mathbf{z^1 = x_1}$
$\mathbf{z^2 = f_1(z^1)}$
$\mathbf{z^3 = f_2(z^2)}$
$\mathcal{L} = \mathbf{z^4 = f_3(z^3)}$
$\sum$ layer 1
softmax
layer 2
loss
layer 3
$\boldsymbol{\delta}^1 = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^1}$
$\boldsymbol{\delta}^2 = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^2}$
$\boldsymbol{\delta}^3 = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^3}$
$\boldsymbol{\delta}^4 = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^4} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} = 1$
$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}^1}$

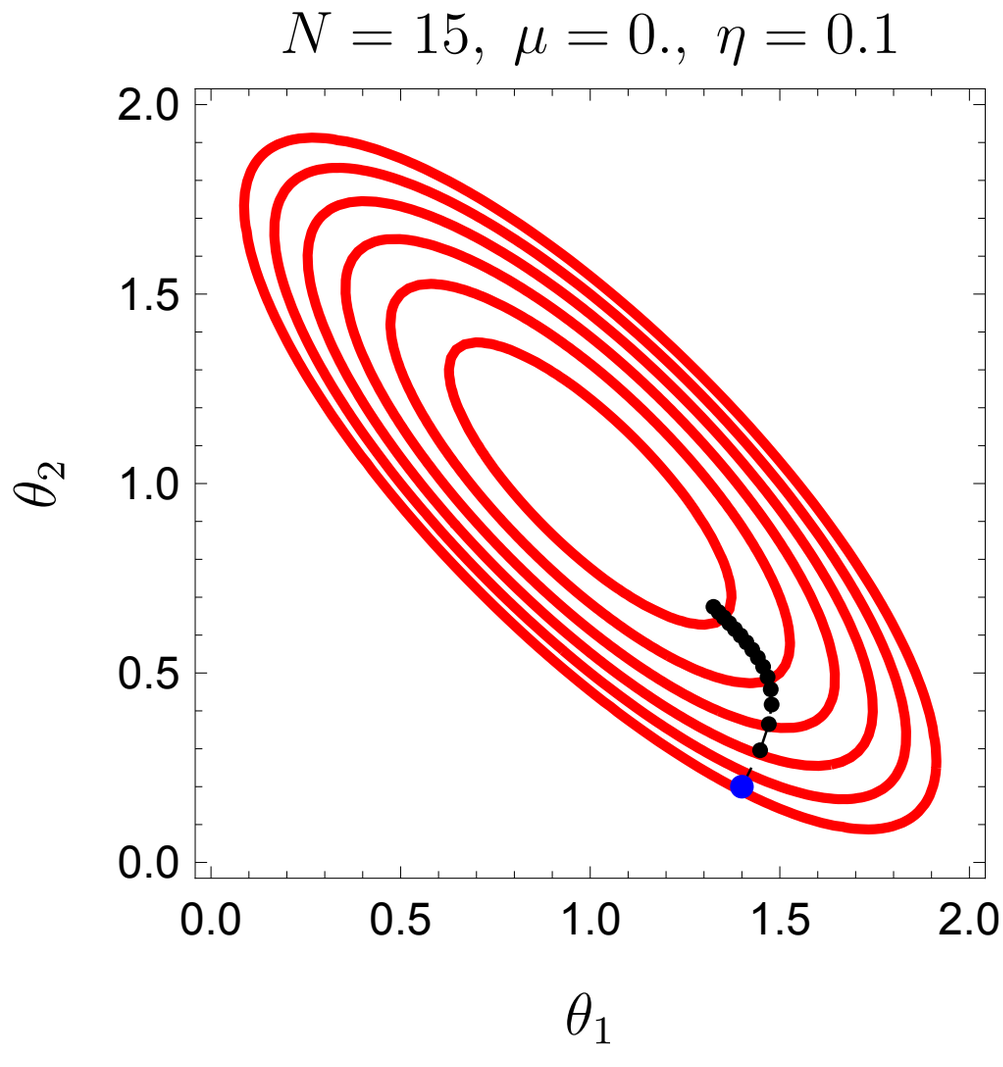

1. Forward Pass
$\mathbf{z^1 = x_1}$
$\mathbf{z^2 = f_1(z^1)}$
$\mathbf{z^3 = f_2(z^2)}$
$\mathcal{L} = \mathbf{z^4 = f_3(z^3)}$
$\sum$ layer 1
softmax
layer 2
loss
layer 3
$\boldsymbol{\delta}^1 = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^1}$
$\boldsymbol{\delta}^2 = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^2}$
$\boldsymbol{\delta}^3 = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^3}$
$\boldsymbol{\delta}^4 = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^4}$
2. Backward Pass
(Backpropagation)
$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}^1}$
3. Parameters

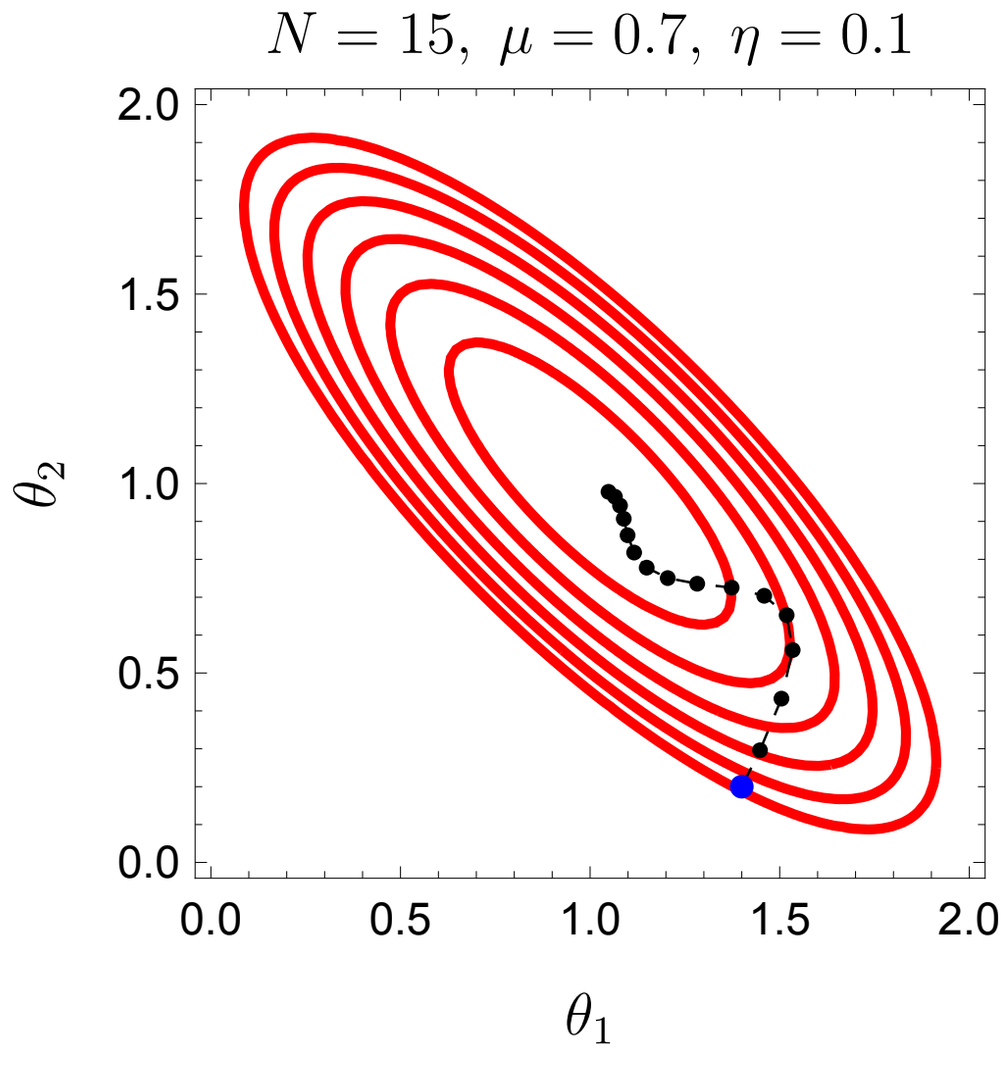

Σ
z_0^l
w_{01}
z_1^l
z_2^l
z_n^l
w_{nK}
1
2
3
K
z_1^{l+1}
z_2^{l+1}
z_3^{l+1}
z_K^{l+1}

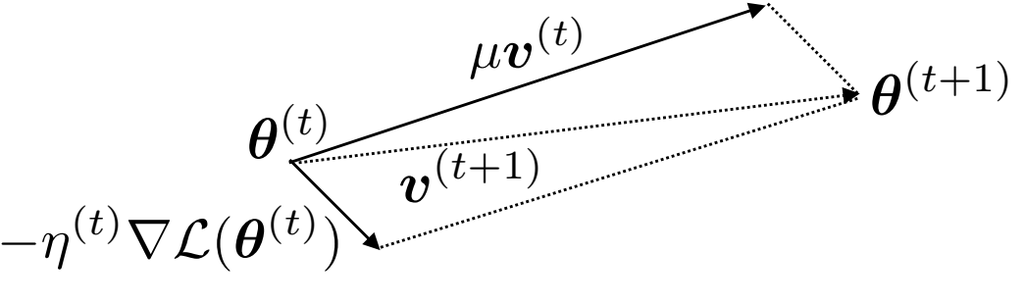

$N = 15,\ \eta = 0.1$
2.0
1.5
1.0
0.5
0.0
θ_2
0.0
0.5
1.0
1.5
2.0
θ_1


$N = 15,\ \eta = 0.4$
2.0
1.5
1.0
0.5
0.0
θ_2
0.0
0.5
1.0
1.5
2.0
θ_1

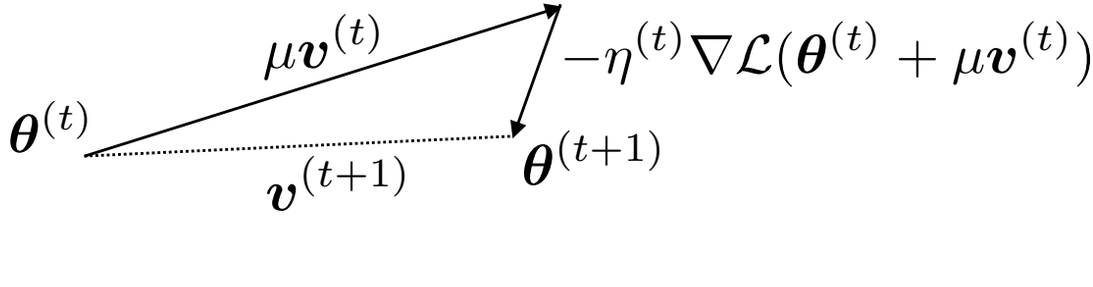

$N = 15,\ \eta = 0.55$
2.0
1.5
1.0
0.5
0.0
θ_2
0.0
0.5
1.0
1.5
2.0
θ_1


$N = 15,\ \mu = 0.,\ \eta = 0.1$
2.0
1.5
1.0
0.5
0.0
θ_2
0.0
0.5
1.0
1.5
2.0
θ_1

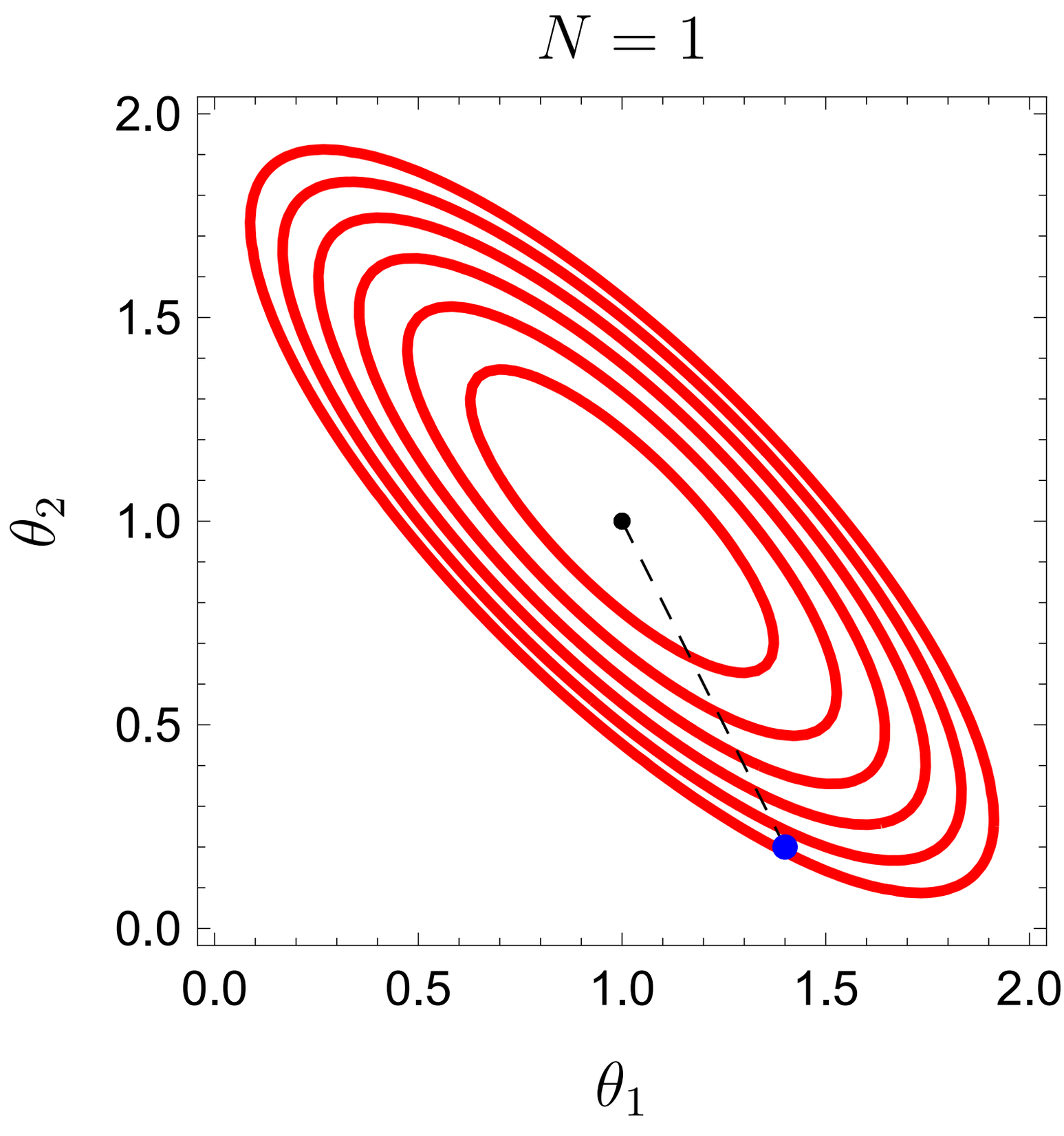

$N = 15,\ \mu = 0.7,\ \eta = 0.1$
2.0
1.5
1.0
0.5
0.0
θ_2
0.0
0.5
1.0
1.5
2.0
θ_1

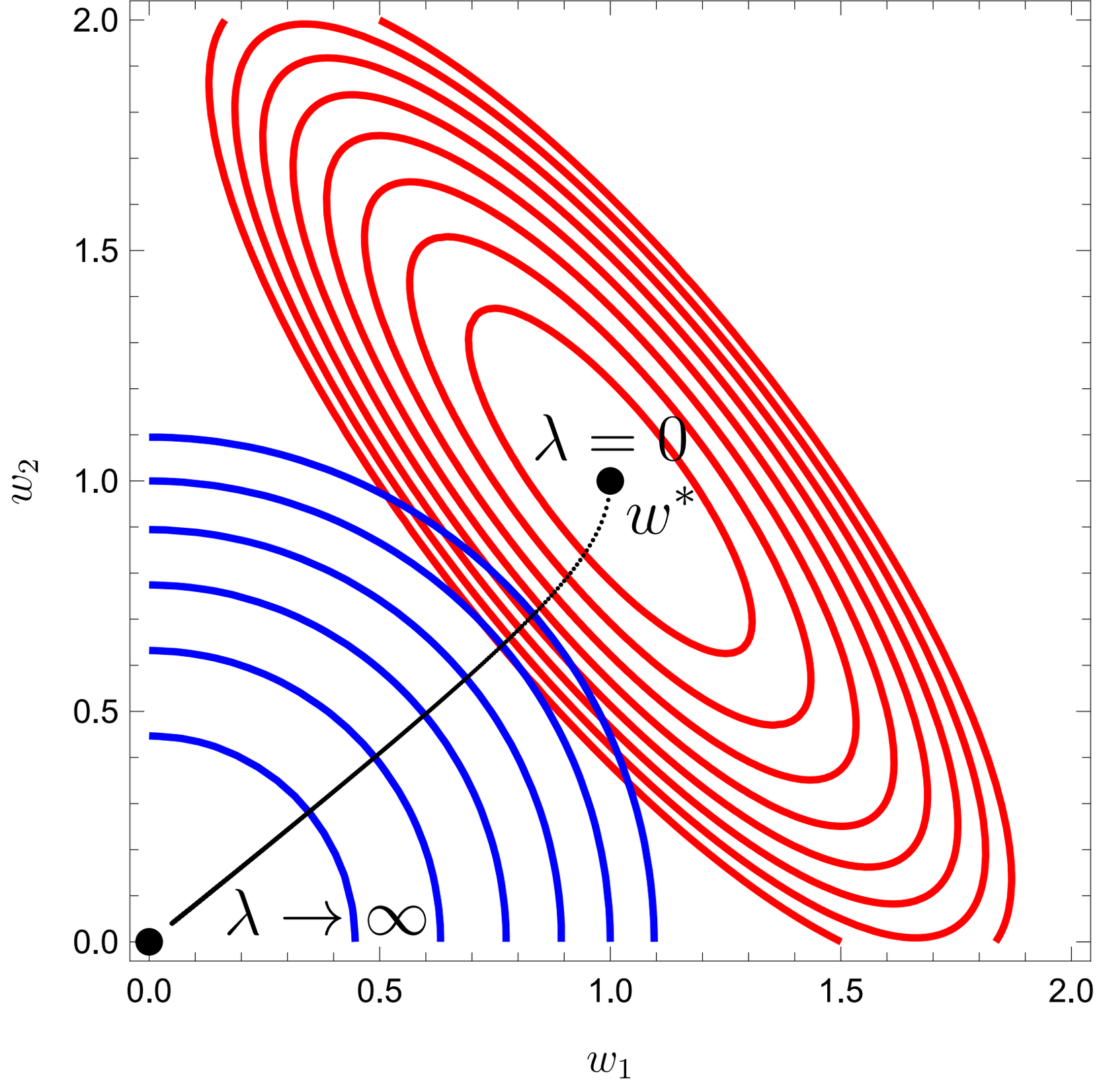


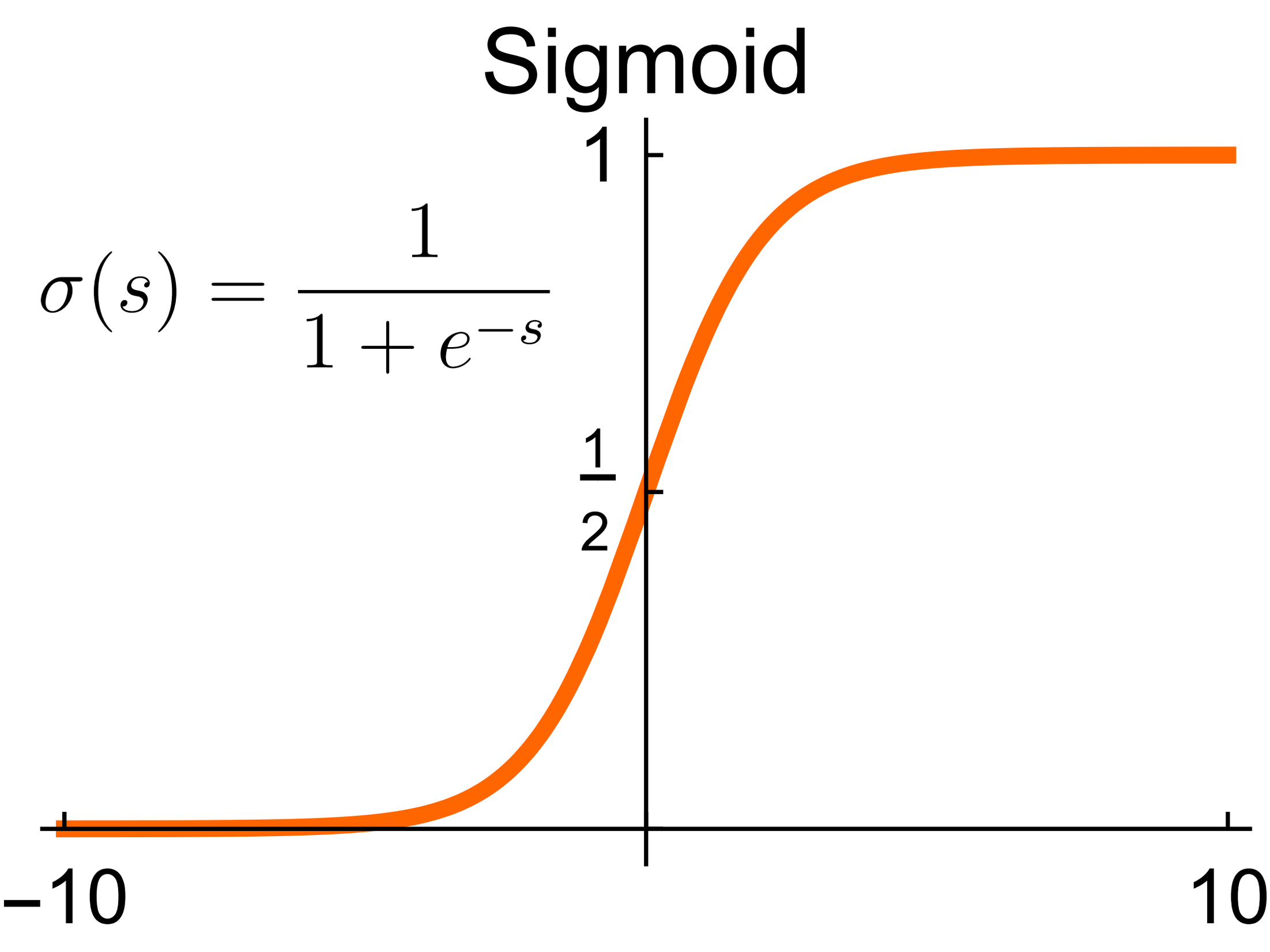
classic

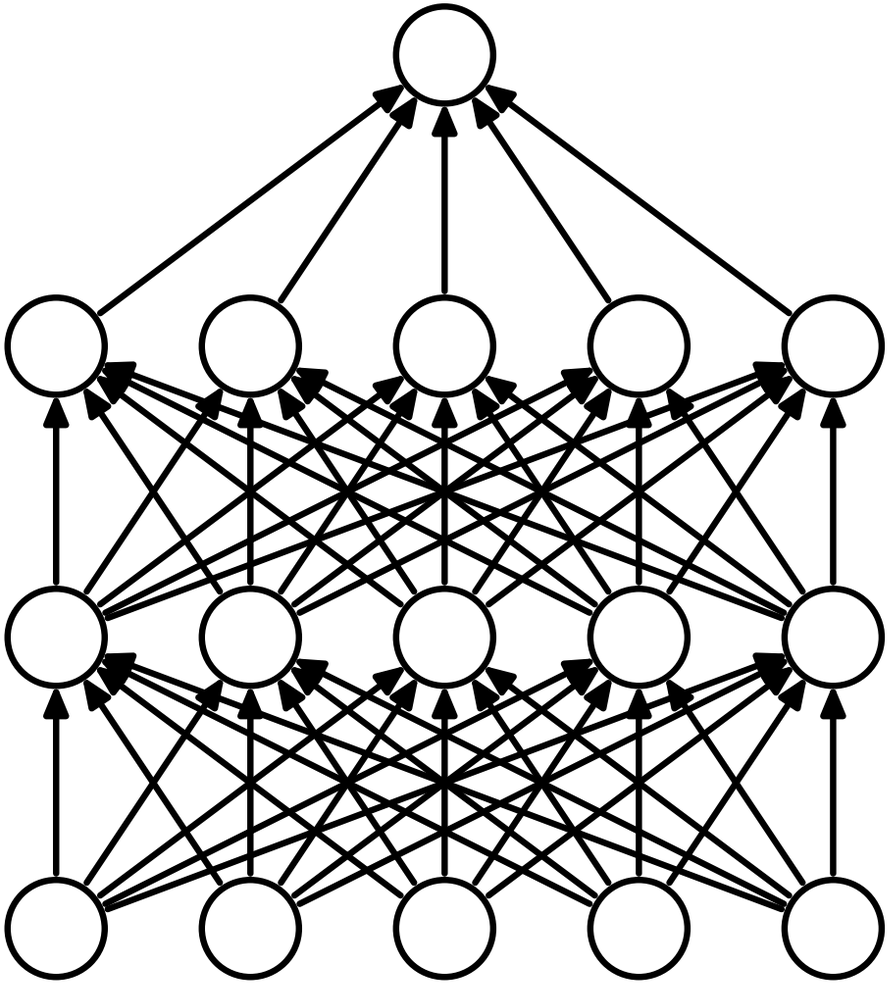


Nesterov

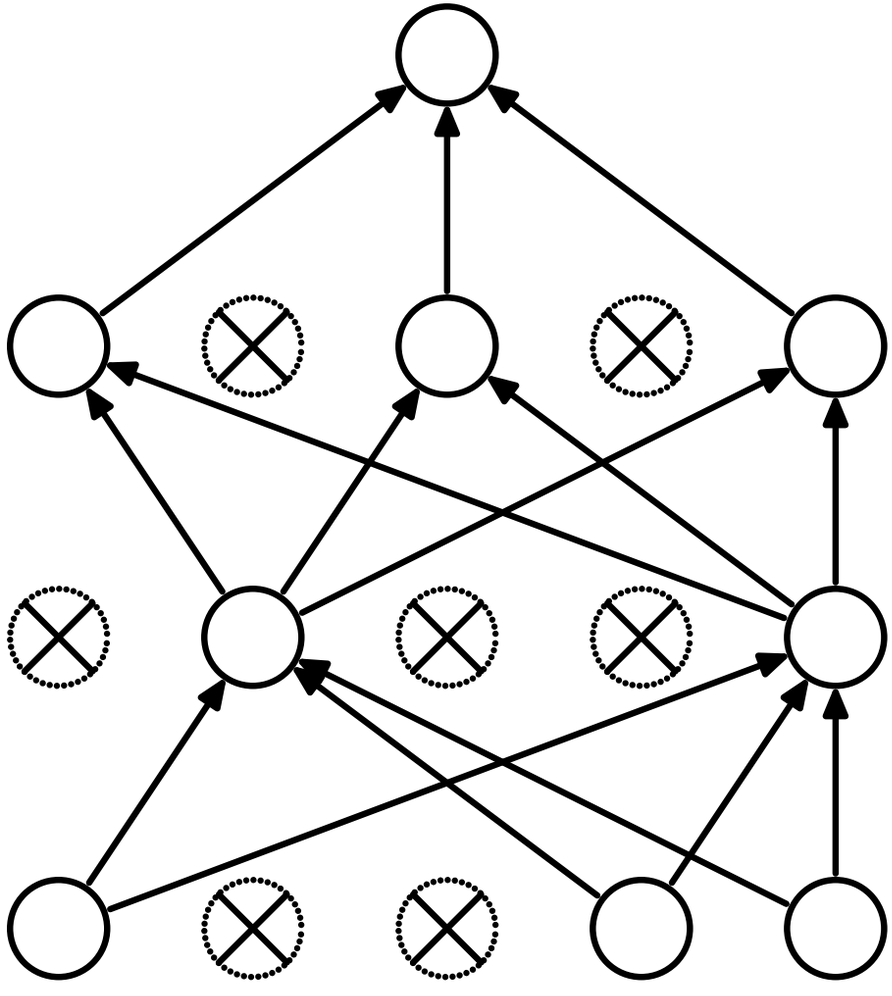

$N = 1$
2.0
1.5
1.0
0.5
0.0
θ_2
0.0
0.5
1.0
1.5
2.0
θ_1


2.0
1.5
1.0
0.5
0.0
w_2
0.0
0.5
1.0
1.5
2.0
w_1
$\lambda = 0$
w^*
$\lambda \to \infty$


Sigmoid
$\sigma(s) = \frac{1}{1+e^{-s}}$
1
$\frac{1}{2}$
−10
10



(a) Standard Neural Net



(b) After applying dropout.