

Statistical Machine Learning (BE4M33SSU) Artificial Neural Networks

Jan Drchal

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

Outline

Topics covered in the lecture:

- ◆ Neuron types
- ◆ Layers
- ◆ Loss functions
- ◆ Computing loss gradients via backpropagation
- ◆ Gradient Descent
- ◆ Parameter initialization
- ◆ Regularization

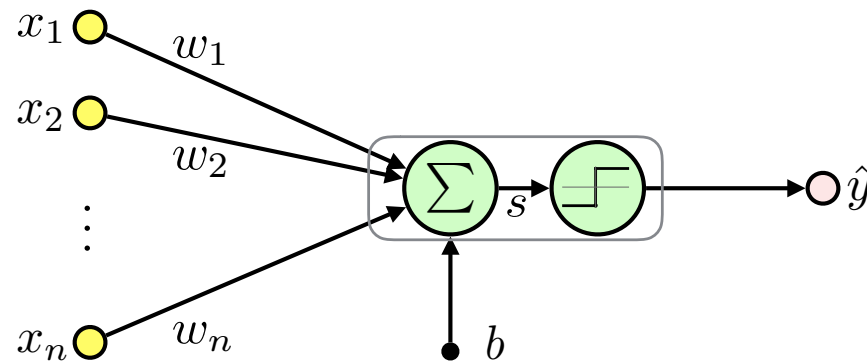
Neural Networks Overview

- ◆ Training examples: $\mathcal{T}^m = \{(x_i, y_i) \in (\mathcal{X} \times \mathcal{Y}) \mid i = 1, \dots, m\}$, where $\mathcal{X} \subseteq \mathbb{R}^n$ and $\mathcal{Y} \subseteq \mathbb{R}^K$
- ◆ Neural network is a composition of simple linear or non-linear functions (neurons) parametrized by *weights* and *biases*
- ◆ Here we consider $\mathcal{H}$ a hypothesis space of neural networks having a fixed architecture
- ◆ Learning methods are based on Empirical Risk Minimization:

$$R_{\mathcal{T}^m}(h_{\theta}) = \frac{1}{m} \sum_{i=1}^m \ell(y_i, h_{\theta}(x_i)),$$

where $h_{\theta} \in \mathcal{H}$ denotes a neural network parametrized by θ

McCulloch-Pitts Perceptron (1943)



$\mathbf{x} = (x_1, x_2, \dots, x_n)^T \in \mathbb{R}^n$ input (feature vector)

$\mathbf{w} = (w_1, w_2, \dots, w_n)^T \in \mathbb{R}^n$ weights

$b \in \mathbb{R}$ bias (threshold)

$s = \langle \mathbf{w}, \mathbf{x} \rangle + b \in \mathbb{R}$ inner potential

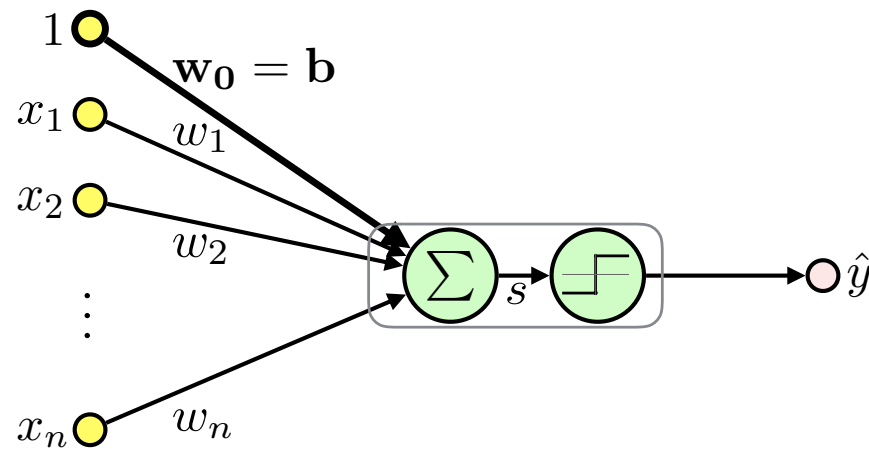
$f(s) = \begin{cases} -1 & \text{if } s < 0 \\ 1 & \text{else} \end{cases}$ activation function

$\hat{y} = h_{(\mathbf{w}, b)}(\mathbf{x}) \in \{-1, 1\}$ output (activity)

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

◆ It is the linear classifier we have already seen.

McCulloch-Pitts Perceptron: Treating Bias

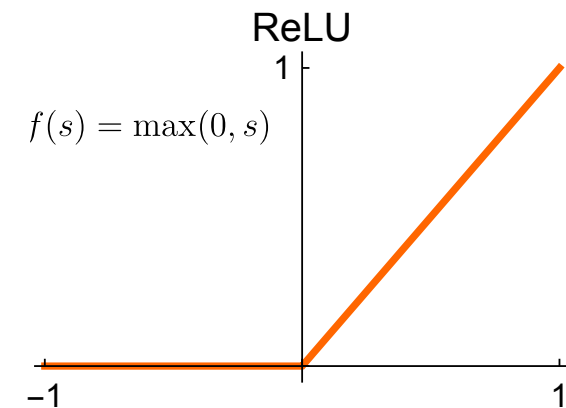
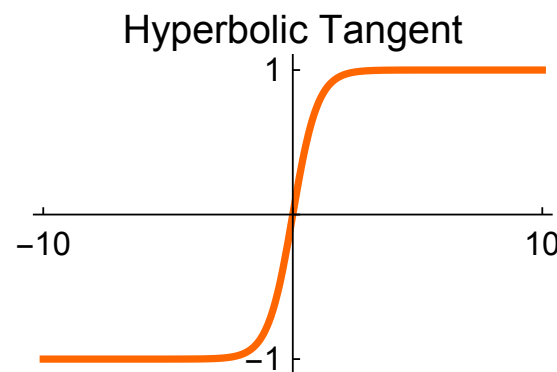
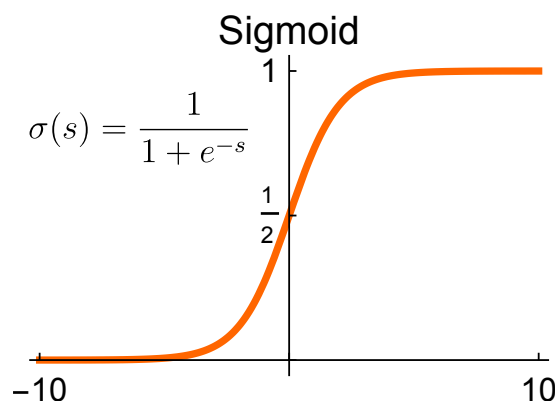
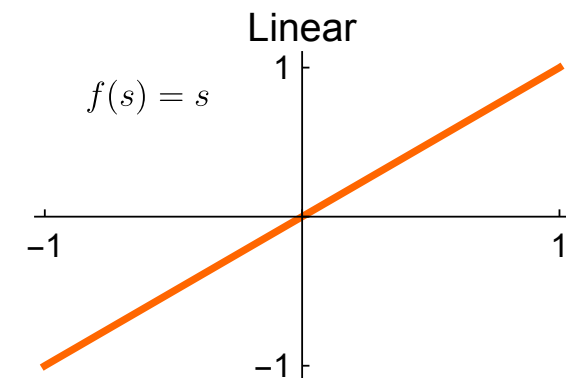
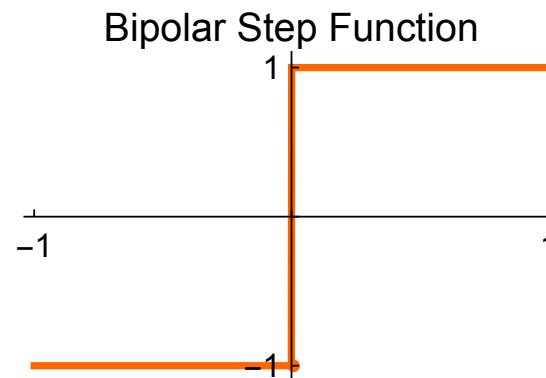
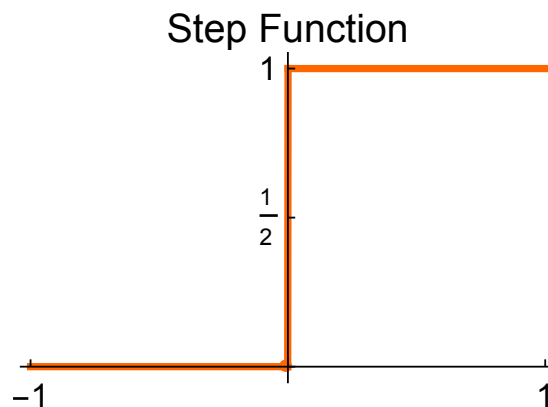


- ◆ Treat bias as an extra fixed input $x_0 = 1$ weighted $w_0 = b$:

$$\hat{y} = f(\langle \mathbf{w}, \mathbf{x} \rangle + b) = f(\langle \mathbf{w}, \mathbf{x} \rangle + w_0 \cdot 1) = f(\langle \mathbf{w}', \mathbf{x}' \rangle)$$

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

Activation Functions



◆ Logistic sigmoid: $\sigma(s) \triangleq \frac{1}{1 + e^{-s}} = \frac{e^s}{e^s + 1}$

◆ Note: $\tanh(s) = \frac{e^s - e^{-s}}{e^s + e^{-s}} = 2\sigma(2s) - 1$

Linear Neuron

- ◆ Training examples: $\mathcal{T}^m = \{(\mathbf{x}_i, y_i) \in (\mathbb{R}^{n+1} \times \mathbb{R}) \mid i = 1, \dots, m\}$
- ◆ Single neuron with linear activation function $\equiv$ **linear regression**:

$$\hat{y} = s = \langle \mathbf{x}, \mathbf{w} \rangle, \quad \hat{y} \in \mathbb{R}$$

$$\text{◆ Inputs: } \mathbf{X} = \begin{pmatrix} 1 & x_{11} & \dots & x_{1n} \\ 1 & \vdots & \ddots & \vdots \\ 1 & x_{m1} & \dots & x_{mn} \end{pmatrix} = \begin{pmatrix} \mathbf{x}_1^T \\ \vdots \\ \mathbf{x}_m^T \end{pmatrix}$$

$$\text{◆ Targets: } \mathbf{y} = (y_1, \dots, y_m)^T, \quad y_i \in \mathbb{R}$$

$$\text{◆ Outputs: } \hat{\mathbf{y}} = (\hat{y}_1, \dots, \hat{y}_m)^T, \quad \hat{y}_i \in \mathbb{R}$$

- ◆ For the whole dataset we get:

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

Linear Neuron: Maximum Likelihood Estimation

- ◆ Assumption: data are Gaussian distributed with mean $\langle \mathbf{x}_i, \mathbf{w} \rangle$ and variance σ^2 :

$$y_i \sim \mathcal{N}(\langle \mathbf{x}_i, \mathbf{w} \rangle, \sigma^2) = \langle \mathbf{x}_i, \mathbf{w} \rangle + \mathcal{N}(0, \sigma^2)$$

- ◆ Likelihood for i.i.d. data:

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

- ◆ Negative Log Likelihood (switching to minimization):

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

Linear Neuron: Maximum Likelihood Estimation (contd.)

◆ Note that

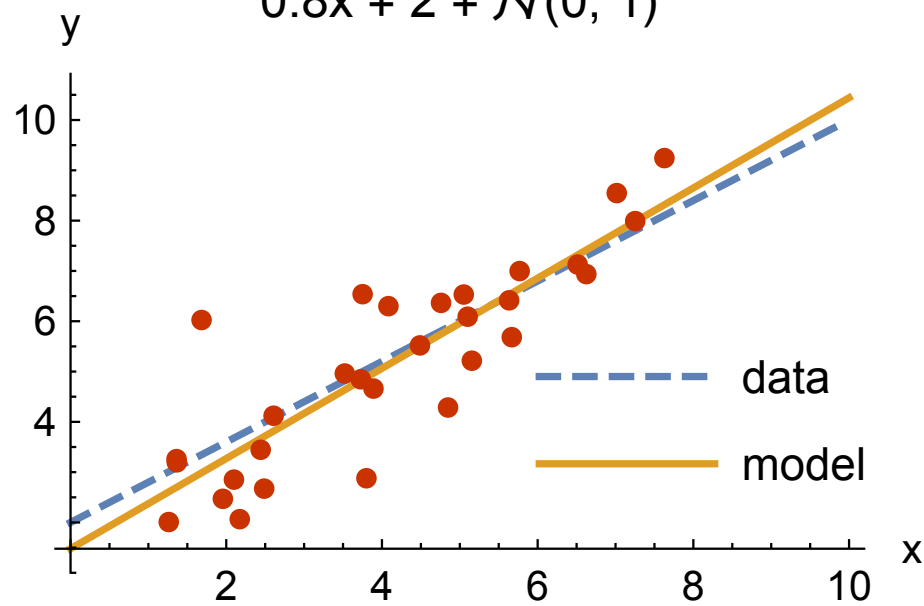
$$\sum_{i=1}^m \underbrace{(y_i - \langle \mathbf{w}, \mathbf{x}_i \rangle)^2}_{\ell(y_i, \hat{y}_i)} = (\mathbf{y} - \mathbf{X}\mathbf{w})^T (\mathbf{y} - \mathbf{X}\mathbf{w})$$

is the **sum-of-squares** or **squared error** (SE)

◆ Minimization of $\mathcal{L}(\mathbf{w}) \equiv$ least squares estimation

◆ Solving $\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = 0$ we get $\mathbf{w}^* = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$ (see seminar)

$0.8x + 2 + \mathcal{N}(0, 1)$



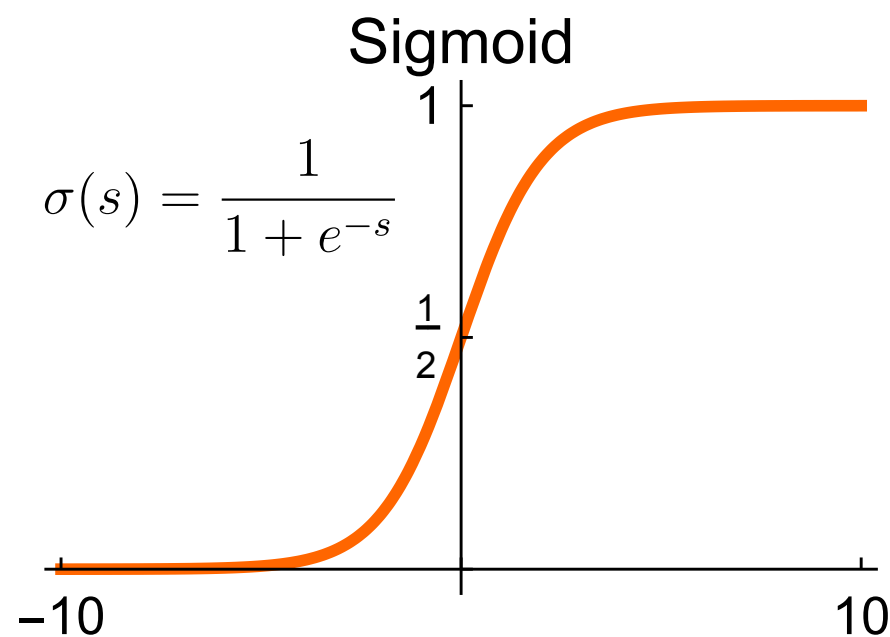
Logistic Sigmoid and Probability

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

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

- ◆ Models confidence of the positive class $y = 1$
- ◆ 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**:

$$\hat{y} = \sigma(\langle \mathbf{w}, \mathbf{x} \rangle), \quad \hat{y} \in (0, 1)$$

- ◆ Inputs: $\mathbf{X} = \begin{pmatrix} 1 & x_{11} & \dots & x_{1n} \\ 1 & \vdots & \ddots & \vdots \\ 1 & x_{m1} & \dots & x_{mn} \end{pmatrix} = \begin{pmatrix} \mathbf{x}_1^T \\ \vdots \\ \mathbf{x}_m^T \end{pmatrix}$
- ◆ Target class: $\mathbf{y} = (y_1, \dots, y_m)^T, y_i \in \{0, 1\}$
- ◆ Output class: $\hat{\mathbf{y}} = (\hat{y}_1, \dots, \hat{y}_m)^T, \hat{y}_i \in (0, 1)$
- ◆ Note that the logistic regression (including the decision rule) actually solves a classification task

Logistic Regression MLE Leads to the Cross-Entropy

- ◆ Likelihood, for the logistic regression:

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

- ◆ Negative Log Likelihood:

$$\mathcal{L}(\mathbf{w}) = \sum_{i=1}^m \underbrace{-[y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i)]}_{\ell(y_i, \hat{y}_i)}$$

- ◆ This *loss function* is called the **cross-entropy**
- ◆ The $\ell(y_i, \hat{y}_i)$ is the negative log probability of the correct answer $y_i \in \{0, 1\}$ given by the model output $\hat{y}_i \in (0, 1)$

Maximum Likelihood Estimation

- ◆ Maximum Likelihood Estimation: $w^* = \underset{w}{\operatorname{argmin}} \mathcal{L}(w)$
- ◆ Derivative of the loss w.r.t. to the sigmoid argument:

$$\frac{\partial \mathcal{L}}{\partial s_i} = \hat{y}_i - y_i \quad (\text{see seminar})$$

- ◆ Gradient w.r.t. logistic regression parameters:

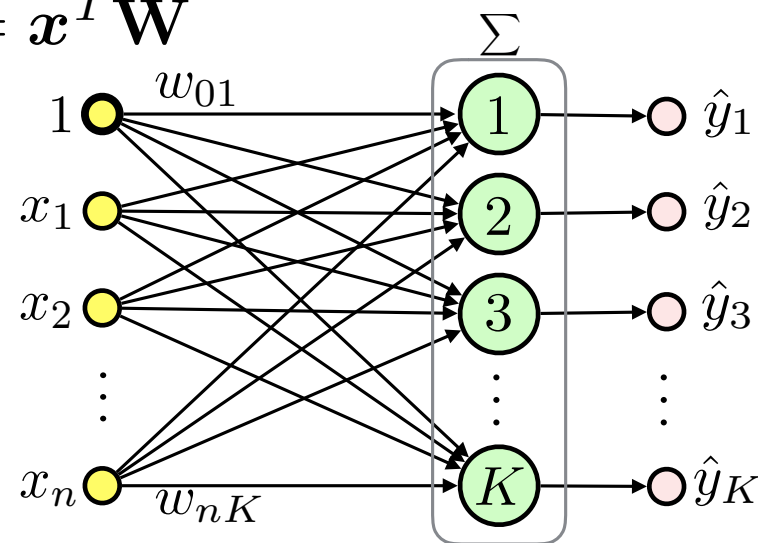
$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \sum_{i=1}^m \frac{\partial \mathcal{L}}{\partial s_i} \cdot \frac{\partial s_i}{\partial \mathbf{w}} = \sum_{i=1}^m \mathbf{x}_i (\hat{y}_i - y_i) = \mathbf{X}^T (\hat{\mathbf{y}} - \mathbf{y})$$

- ◆ $\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \mathbf{0}$ has no analytical solution $\implies$ use numerical methods

Linear Layer

- ◆ Multiple linear neurons: linear (dense, fully-connected, affine) layer
- ◆ Output k : $\hat{y}_k = \langle \mathbf{x}, \mathbf{w}_k \rangle$, $k = 1, 2, \dots, K$
- ◆ All outputs using *weight matrix* $\mathbf{W}$: $\hat{\mathbf{y}} = \mathbf{x}^T \mathbf{W}$
- ◆ Multiple samples: $\hat{\mathbf{Y}} = \mathbf{X} \mathbf{W}$

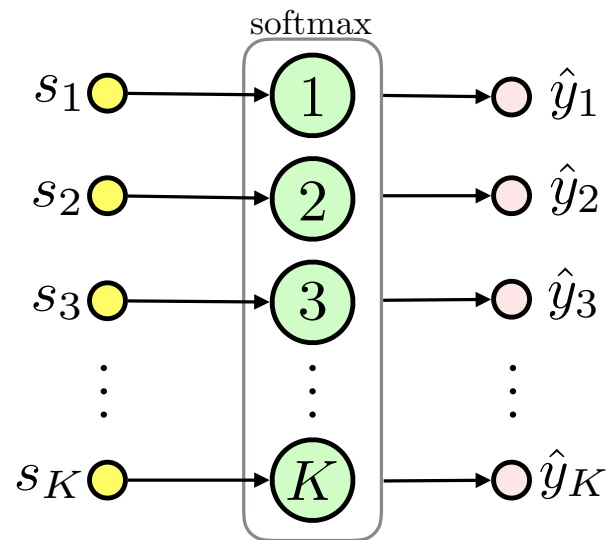
$$\mathbf{W} = (\mathbf{w}_1 \dots \mathbf{w}_K) = \begin{pmatrix} w_{01} & \dots & w_{0K} \\ \vdots & \ddots & \vdots \\ w_{n1} & \dots & w_{nK} \end{pmatrix}$$



$$\mathbf{X} = \begin{pmatrix} \mathbf{x}_1^T \\ \vdots \\ \mathbf{x}_m^T \end{pmatrix} = \begin{pmatrix} 1 & x_{11} & \dots & x_{1n} \\ 1 & \vdots & \ddots & \vdots \\ 1 & x_{m1} & \dots & x_{mn} \end{pmatrix}$$

$$\hat{\mathbf{Y}} = \begin{pmatrix} \hat{\mathbf{y}}_1^T \\ \vdots \\ \mathbf{y}_m^T \end{pmatrix} = \begin{pmatrix} \hat{y}_{11} & \dots & \hat{y}_{1K} \\ \vdots & \ddots & \vdots \\ \hat{y}_{m1} & \dots & \hat{y}_{mK} \end{pmatrix}$$

Softmax Layer



- ◆ Multinomial classification, K mutually exclusive classes
- ◆ Definition: $\sigma_k(\mathbf{s}) \triangleq \frac{e^{s_k}}{\sum_{c=1}^K e^{s_c}}$, where K is the number of classes
- ◆ Softmax represents a categorical probability distribution: $\sigma_k \in (0, 1)$ for $k \in \{1 \dots K\}$ and $\sum_{k=1}^K \sigma_k = 1$
- ◆ Describes class membership probabilities: $p(y = k | \mathbf{s}) = \sigma_k(\mathbf{s})$
- ◆ Softmax input (vector $\mathbf{s}$) is often called the *logits*

Softmax Layer MLE

- ◆ Target: $\mathbf{y} = (y_1 \dots y_m)^T$, $y_i \in \{1, 2, \dots, K\}$
- ◆ One-hot encoding for sample i and class k : let $y_{ik} \triangleq [y_i = k]$
- ◆ Likelihood:

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

- ◆ Negative Log Likelihood:

$$\mathcal{L}(\mathbf{w}) = - \sum_{i=1}^m \sum_{c=1}^K y_{ic} \log(\hat{y}_{ic})$$

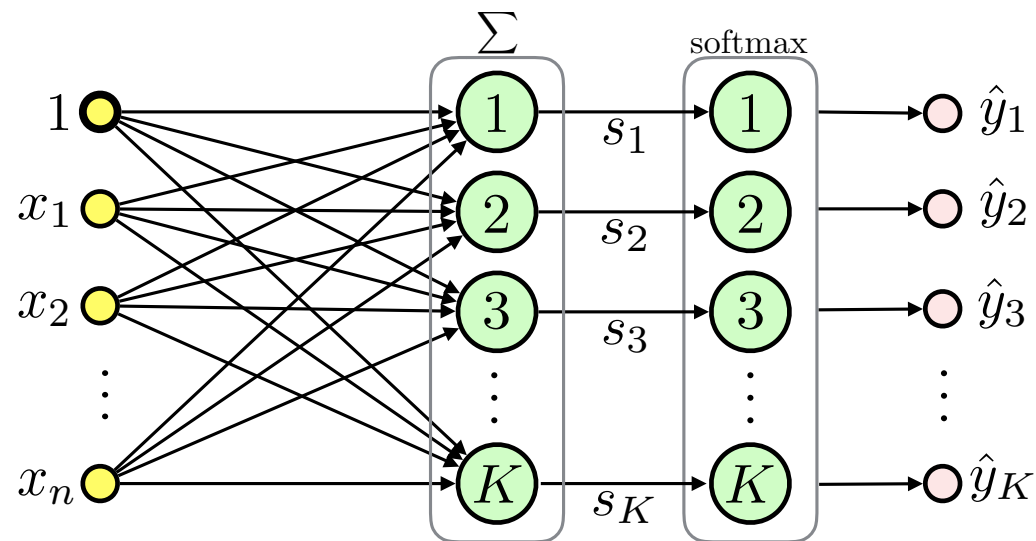
Again the **cross-entropy**

- ◆ See seminar for the gradient

Multinomial Logistic Regression

- ◆ linear layer + softmax layer = **multinomial logistic regression**:

$$\hat{y}_k = \sigma_k(\mathbf{x}^T \mathbf{W})$$

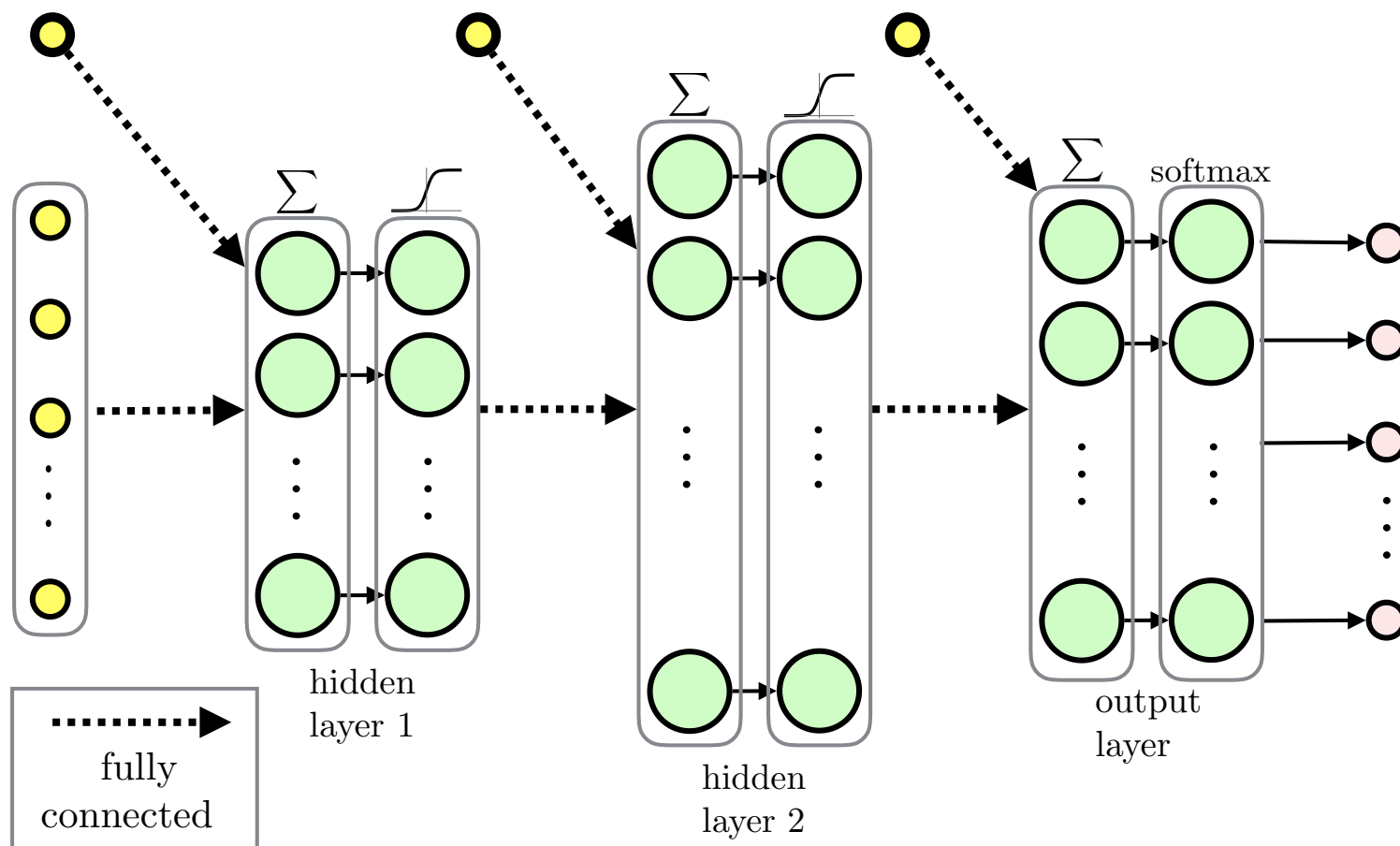


- ◆ Classifier: $h(\mathbf{x}, \mathbf{W}) = \underset{k}{\operatorname{argmax}} \hat{y}_k$

Loss Functions: Summary

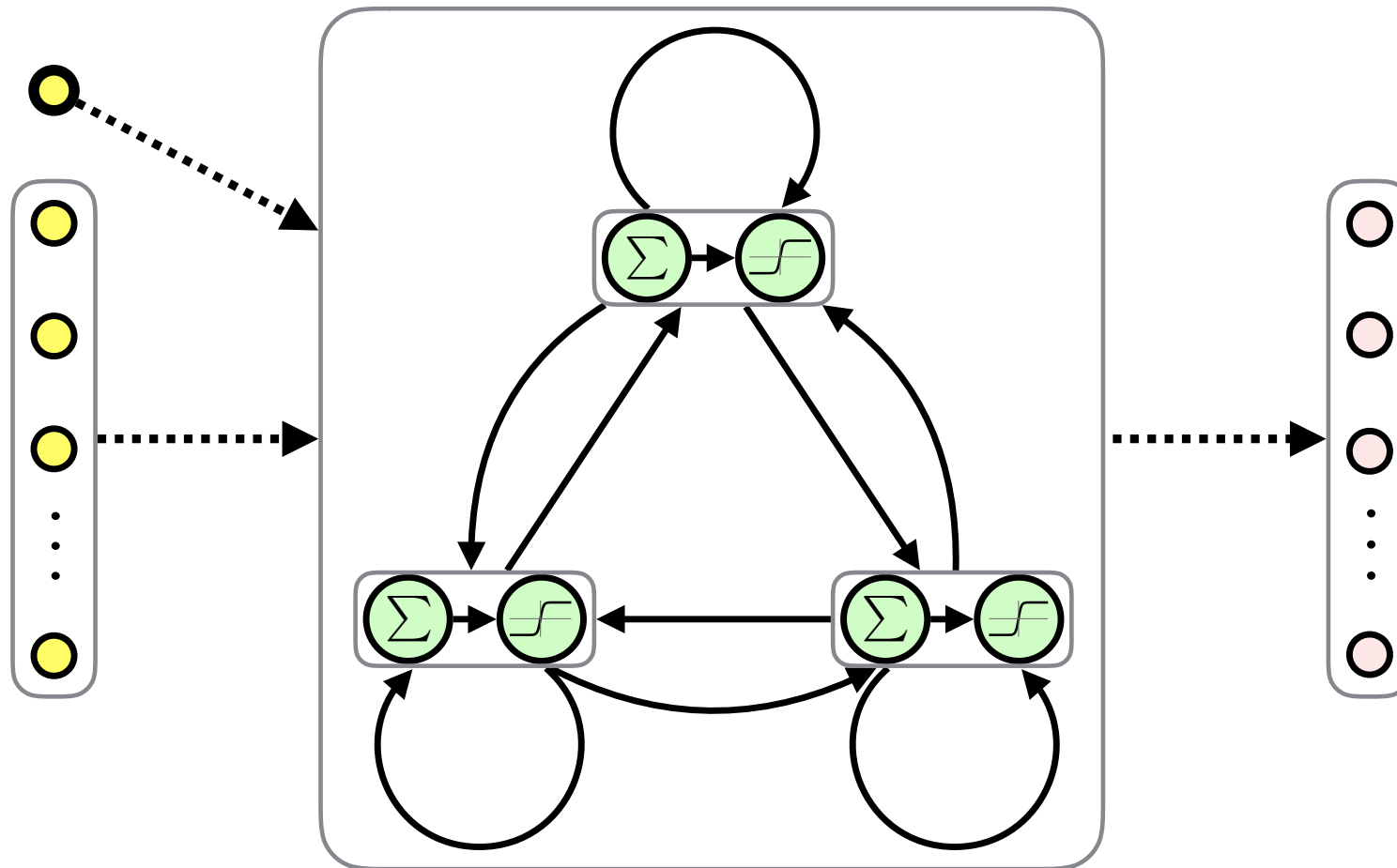
problem	output	suggested loss function
binary classification	sigmoid neuron	cross-entropy $-\frac{1}{m} \sum_{i=1}^m [y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i)]$
multinomial classification	softmax	multinomial cross-entropy $-\frac{1}{m} \sum_{i=1}^m \sum_{c=1}^K y_{ic} \log(\hat{y}_{ic})$
regression	linear neuron	mean squared error $\frac{1}{m} \sum_{i=1}^m (y_i - \hat{y}_i)^2$
multi-output regression	linear layer	mean squared error $\frac{1}{m} \sum_{i=1}^m \sum_{c=1}^K (y_{ic} - \hat{y}_{ic})^2$

Multilayer Perceptron (MLP)



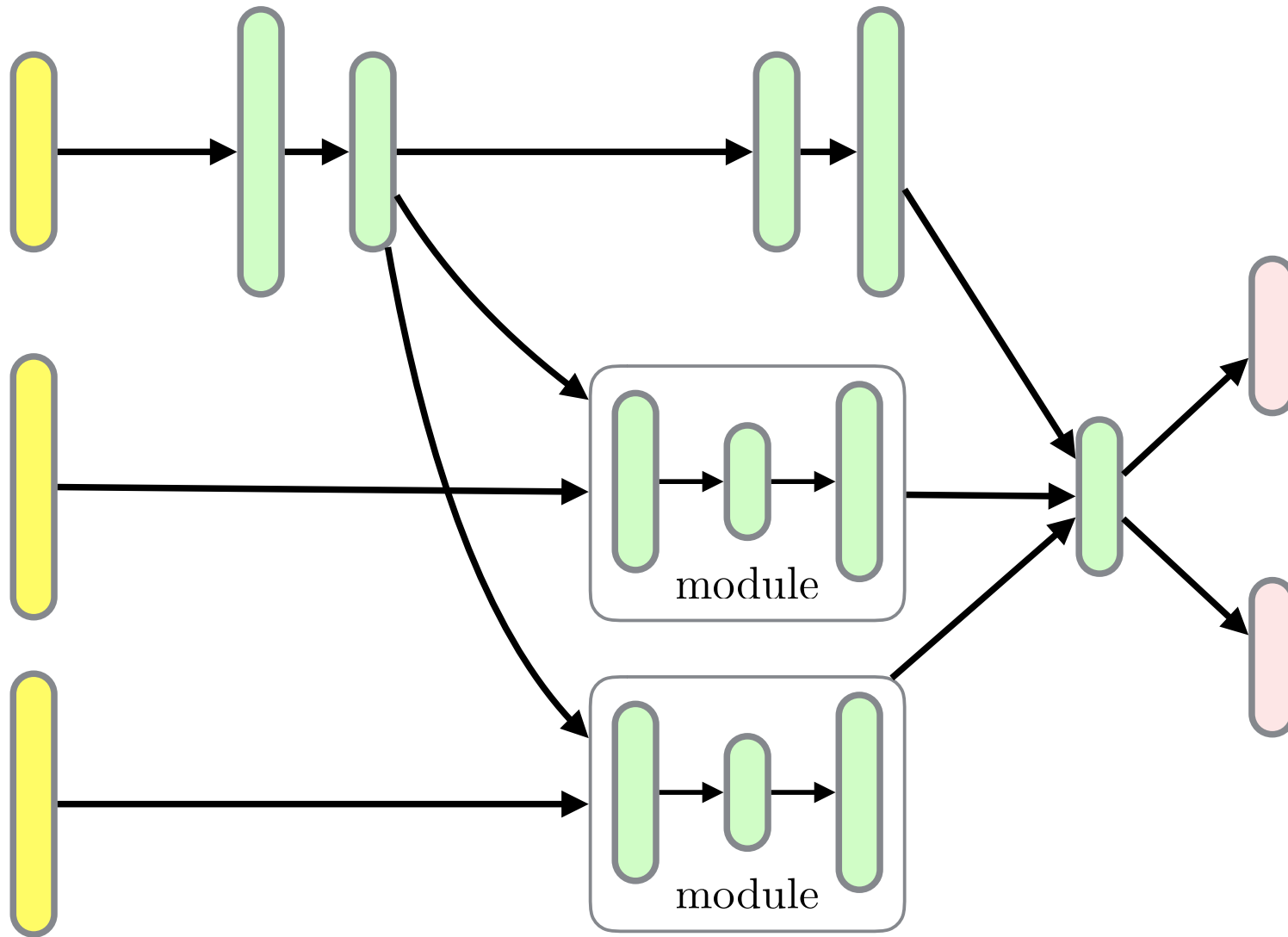
- ◆ Feed-forward ANN
- ◆ Fully-connected layers
- ◆ MLP for regression would typically use linear output layer

Recurrent Neural Network (RNN)



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

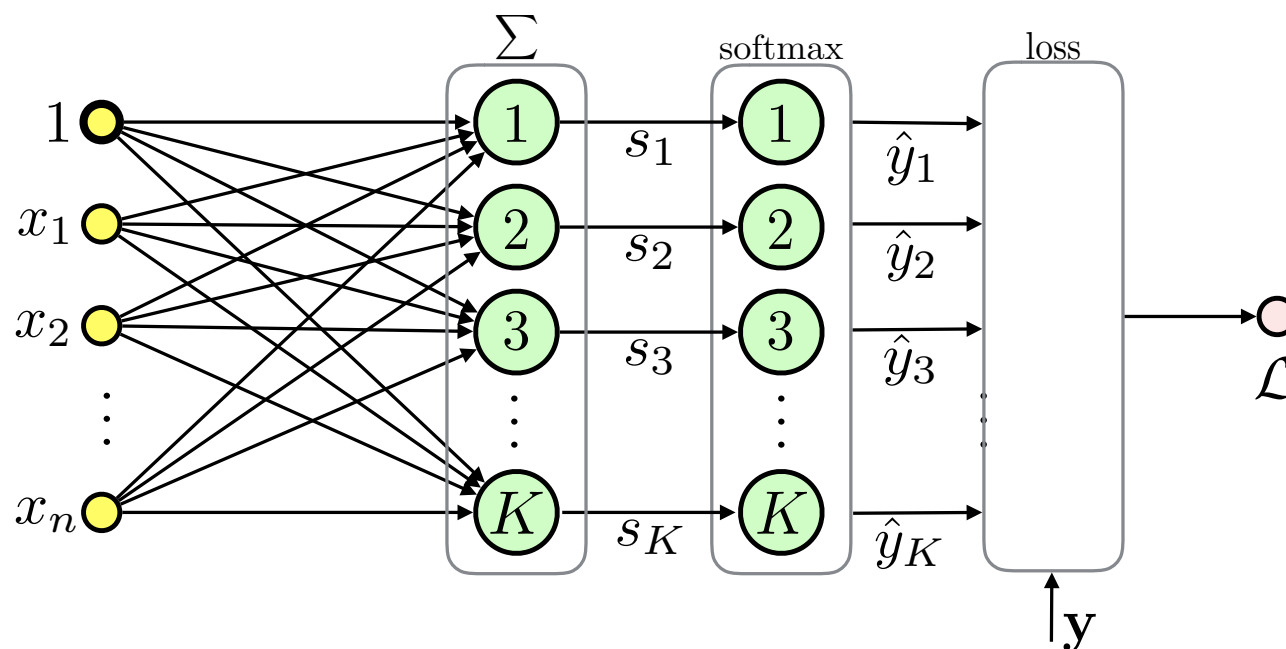
Modular and Hierarchical Architectures



- ◆ Directed Acyclic Graphs (DAGs)
- ◆ Layers can be organized in *modules*
- ◆ Hierarchies of modules, module reuse

Backpropagation Overview

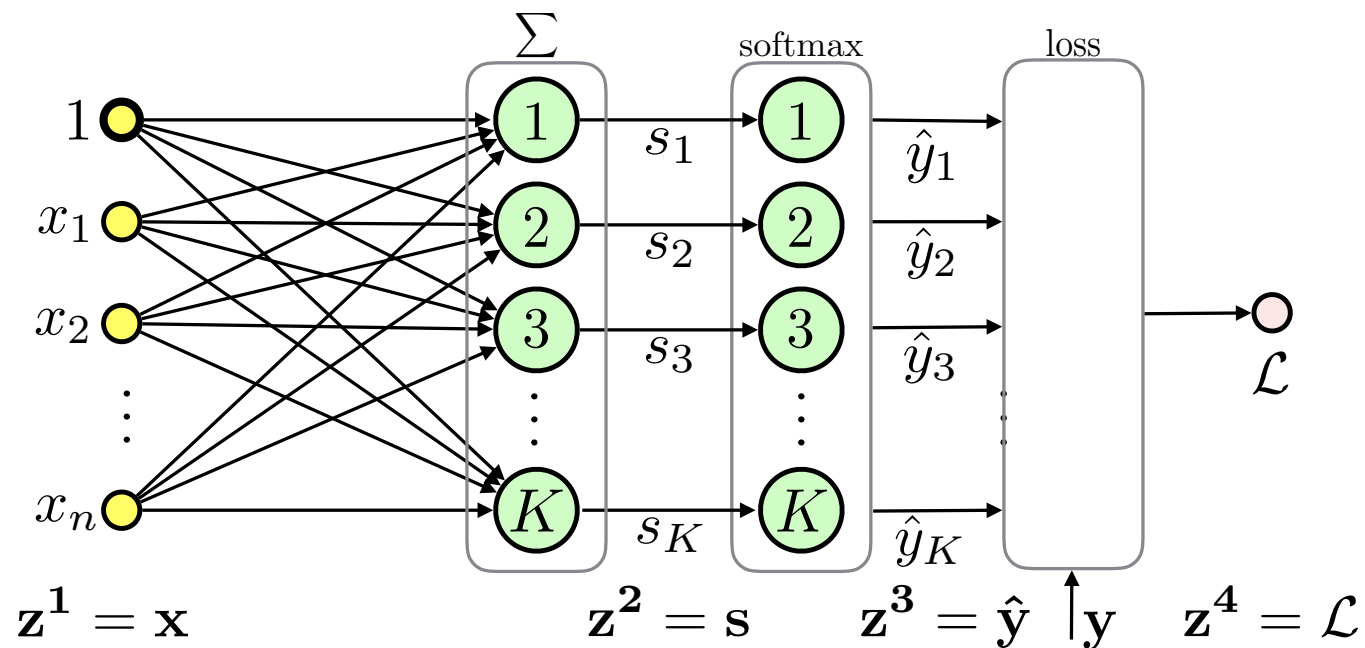
- ◆ A method to compute a gradient of the *loss function* with respect to its parameters: $\nabla \mathcal{L}(\mathbf{w})$
- ◆ $\nabla \mathcal{L}(\mathbf{w})$ is in turn used by optimization methods like gradient descent
- ◆ Here, we present the "modular" backpropagation (see Nando de Freitas' Machine Learning course: <https://www.cs.ox.ac.uk/people/nando.defreitas/machinelearning/>)
- ◆ Let us use multinomial logistic regression as an example



Backpropagation: the Loss Function

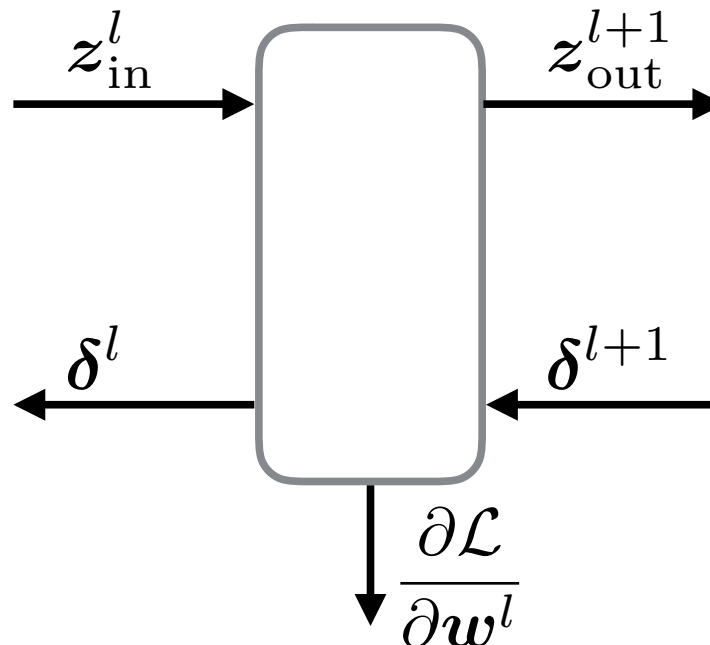
- ◆ The loss function is the multinomial cross-entropy in this case:

$$\mathcal{L}(\mathbf{w}) = -\frac{1}{m} \sum_{i=1}^m \sum_{c=1}^K [y_i = c] \log \left(\frac{\exp(\langle \mathbf{x}_i, \mathbf{w}_c \rangle)}{\sum_{k=1}^K \exp(\langle \mathbf{x}_i, \mathbf{w}_k \rangle)} \right)$$



Backpropagation Based on Modules

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

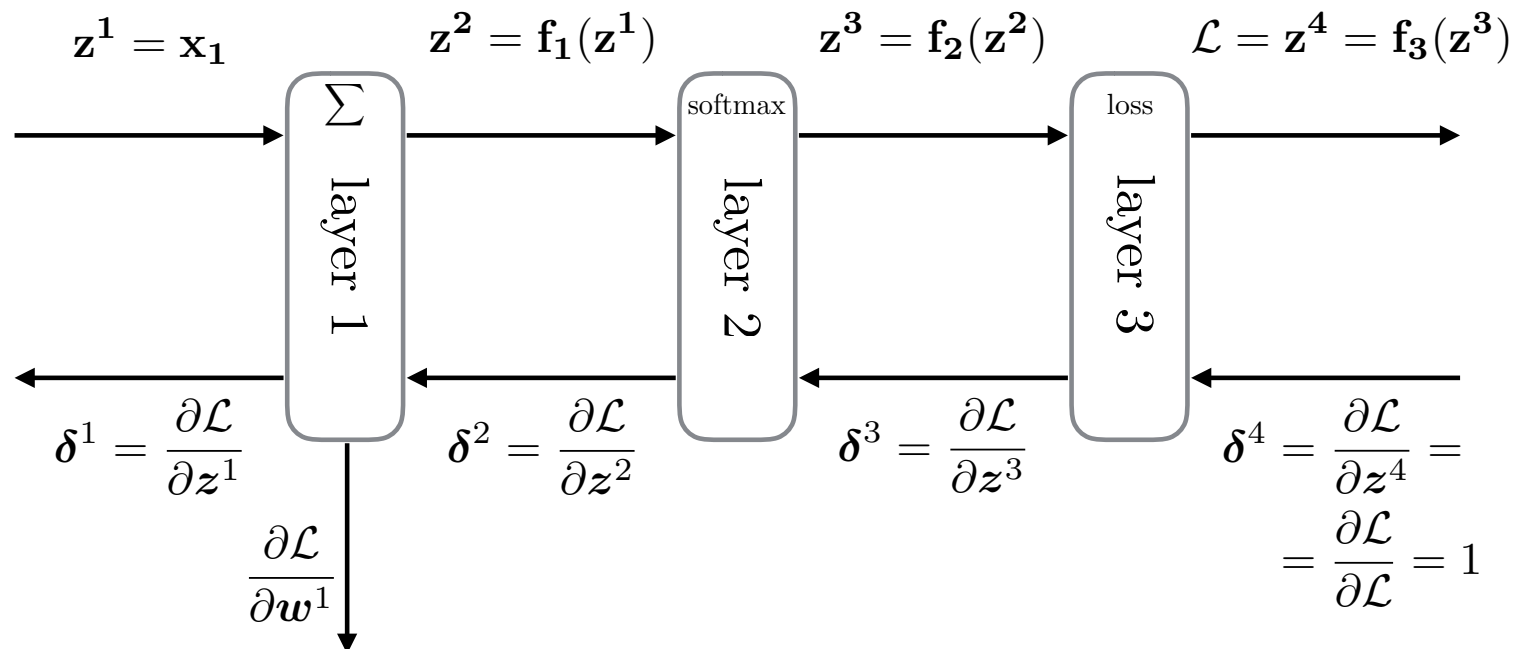


Backpropagation: Backward Pass Message

- Let $\delta^l = \frac{\partial \mathcal{L}}{\partial z^l}$ be the sensitivity of the loss to the module input 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!

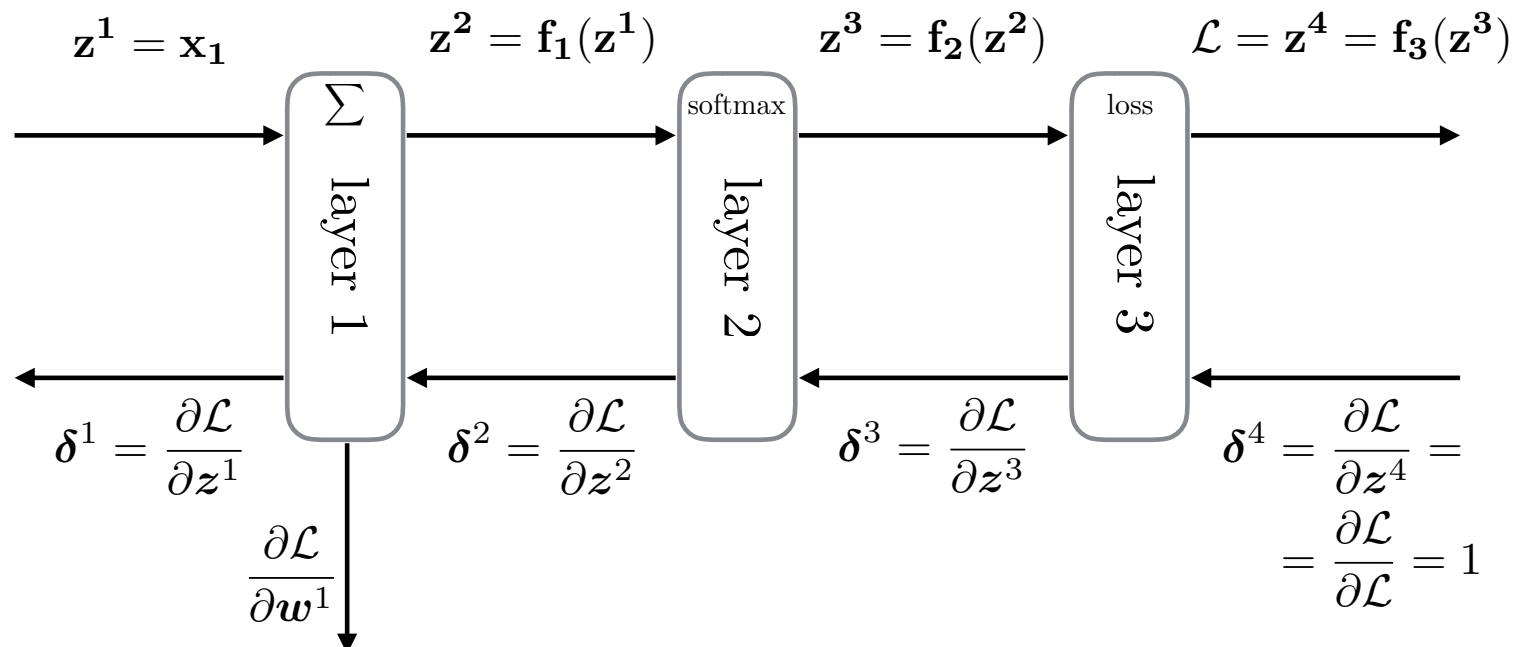


Backpropagation: Parameters

- ◆ 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



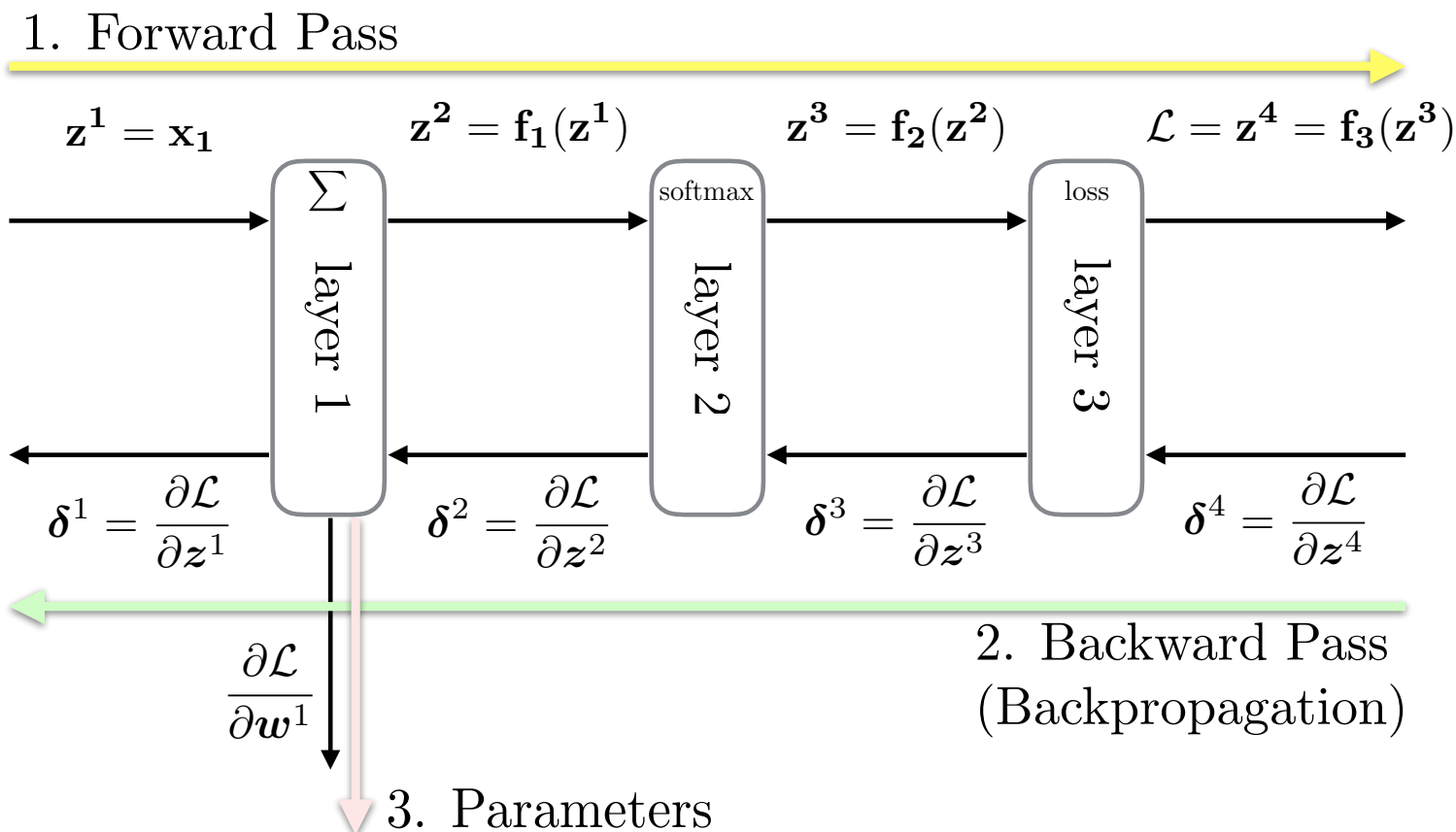
Backpropagation: Steps

- 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): $\frac{\partial z^{l+1}}{\partial w^l}$

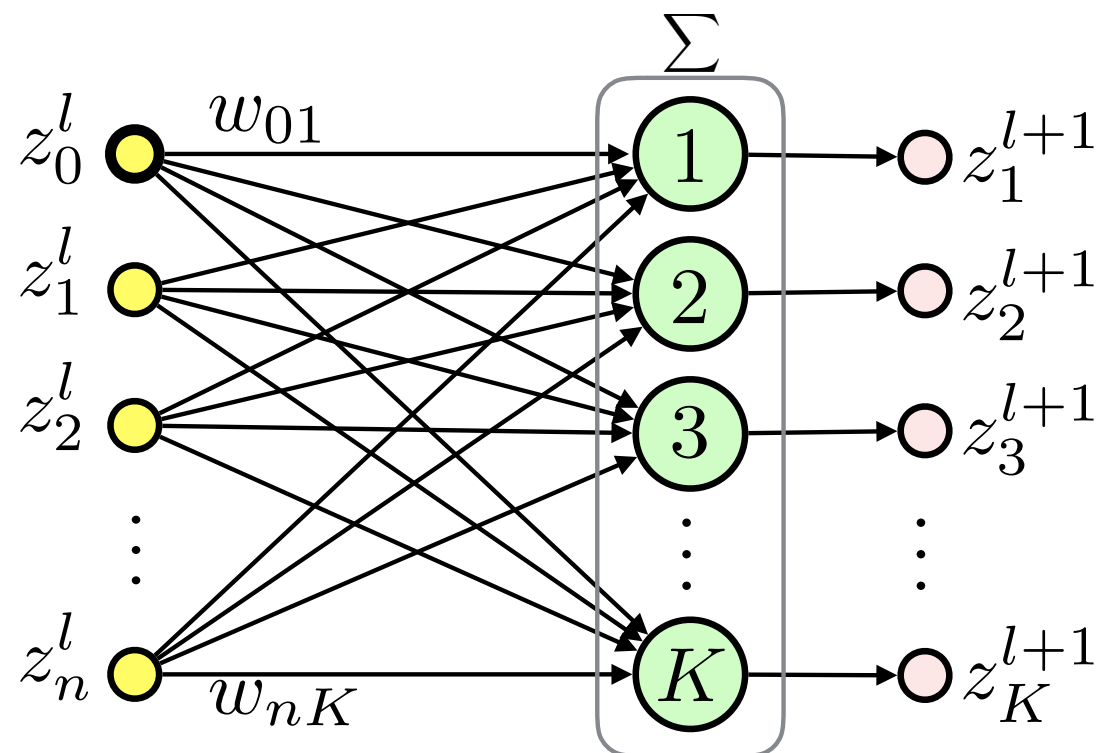


Example: Linear Layer

◆ **forward:** $z_j^{l+1} = \sum_{i=0}^n w_{ij} z_i^l, \quad j = 1, \dots, K$

◆ **backward:** $\frac{\partial z_j^{l+1}}{\partial z_i^l} = w_{ij}, \quad i = 0, \dots, n, \quad j = 1, \dots, K$

◆ **parameter:** $\frac{\partial z_j^{l+1}}{\partial w_{ik}} = [j = k] z_i^l$



Example: Squared Error

- ◆ **forward:** $z^{l+1} = (y - z^l)^2$
- ◆ **backward:** $\frac{\partial z^{l+1}}{\partial z^l} = 2(z^l - y), \quad i \in \{1, \dots, n\}$

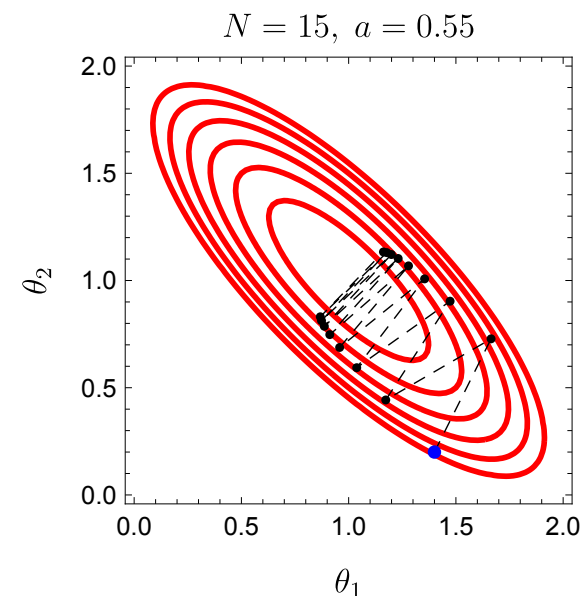
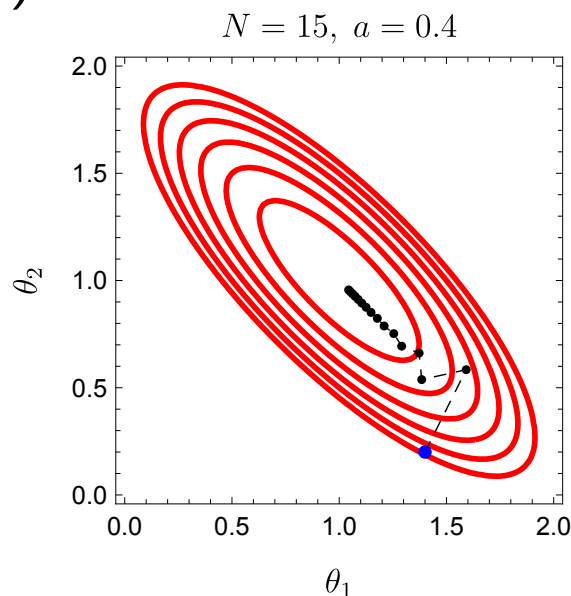
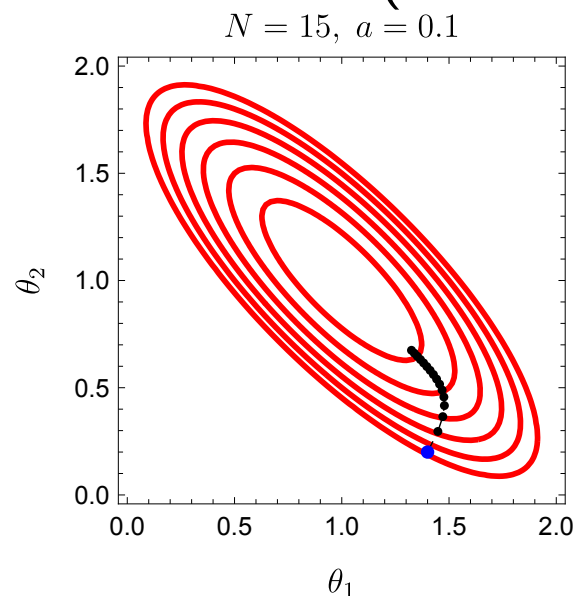
Gradient Descent (GD)

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

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

where θ is a set of all parameters defining the ANN

- ◆ Gradient descent: $\theta_{k+1} = \theta_k - \alpha_k \nabla \mathcal{L}(\theta_k)$
where $\alpha_k > 0$ is the **learning rate** or **stepsize** at iteration k
- ◆ More about (Stochastic) Gradient Descent the next time . . .



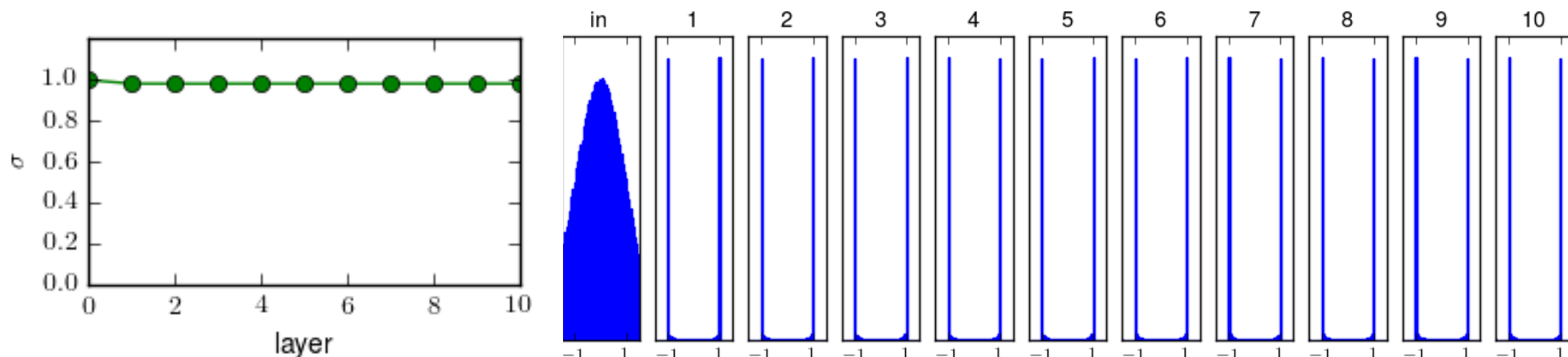
Parameter Initialization

- ◆ Is it a good idea to initially set all weights to a constant?
- ◆ **No.** All neurons would behave the same: the same δ s are backpropagated. We need to *break the symmetry*
- ◆ Use small random numbers, e.g., sample from a Gaussian distribution with zero mean:
 - works well for shallow networks,
 - for deep networks, we may get into trouble

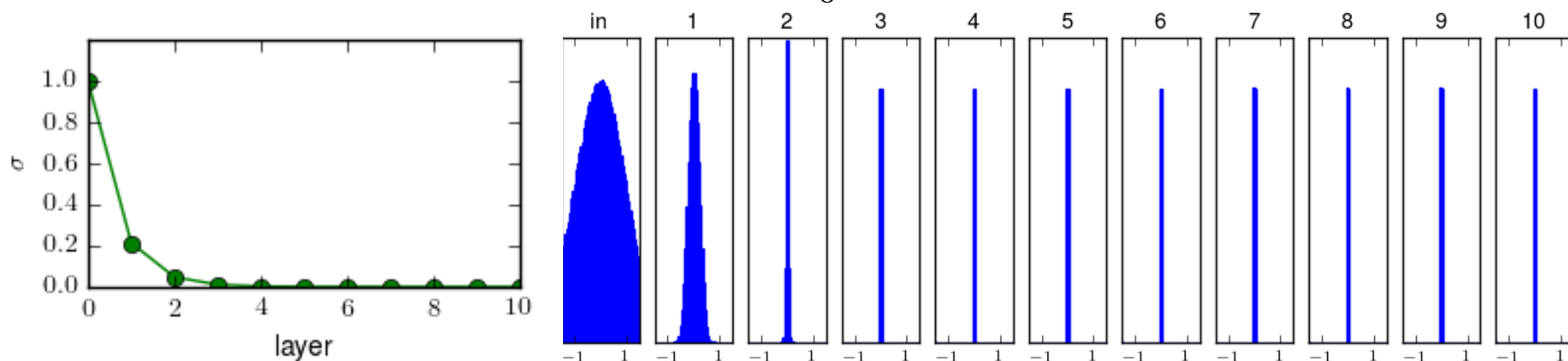
Gaussian Initialization Example

- ◆ MLP, 10 tanh layers, 500 units each. Each input fed with $\mathcal{N}(0, 1)$
- ◆ Weights initialized to $\mathcal{N}(0, \sigma_0^2)$; showing layer output distributions

$$\sigma_0 = 1$$

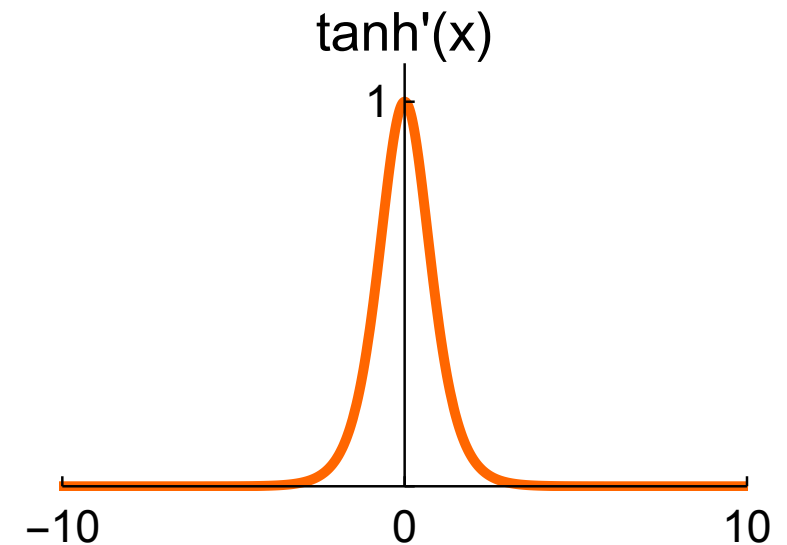
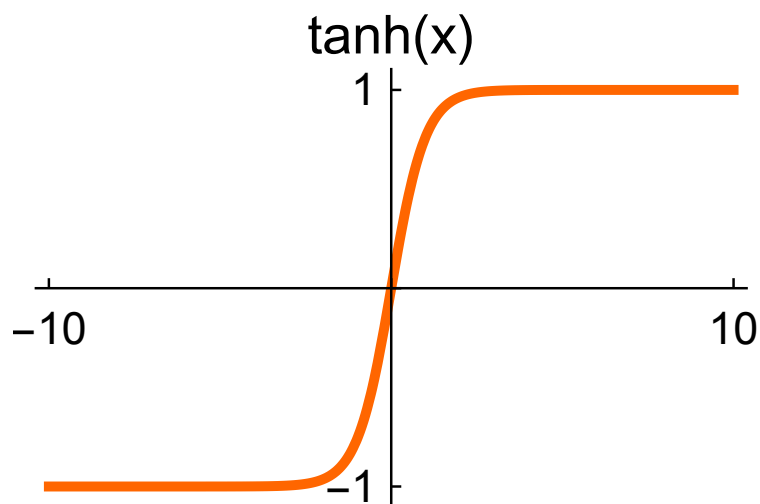


$$\sigma_0 = 0.01$$



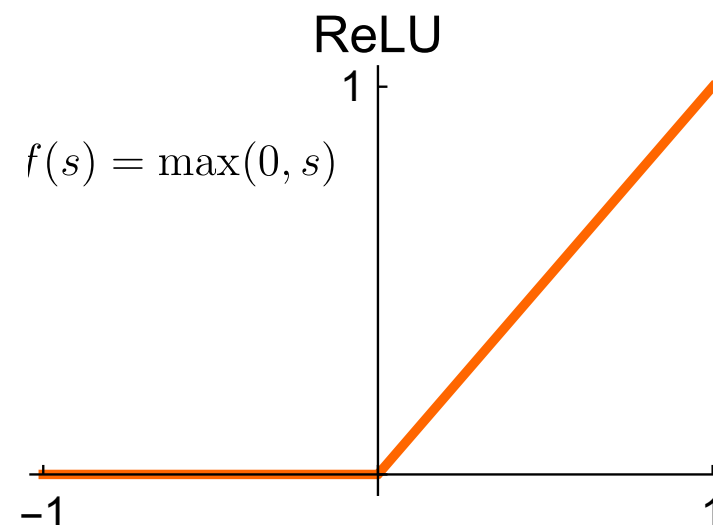
Vanishing Gradient

- ◆ For large σ_0 (saturation) the tanh derivative is almost zero
- ◆ For small σ_0 (output close to zero):
 - the derivative is at most 1,
 - the weights are very small and $\frac{\partial z_j^{l+1}}{\partial z_i^l} = w_{ij}$ holds for the preceding linear layer
- ◆ In both cases: $\delta \rightarrow 0$ as the number of layers increases



Rectified Linear Unit (ReLU)

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



Xavier Initialization

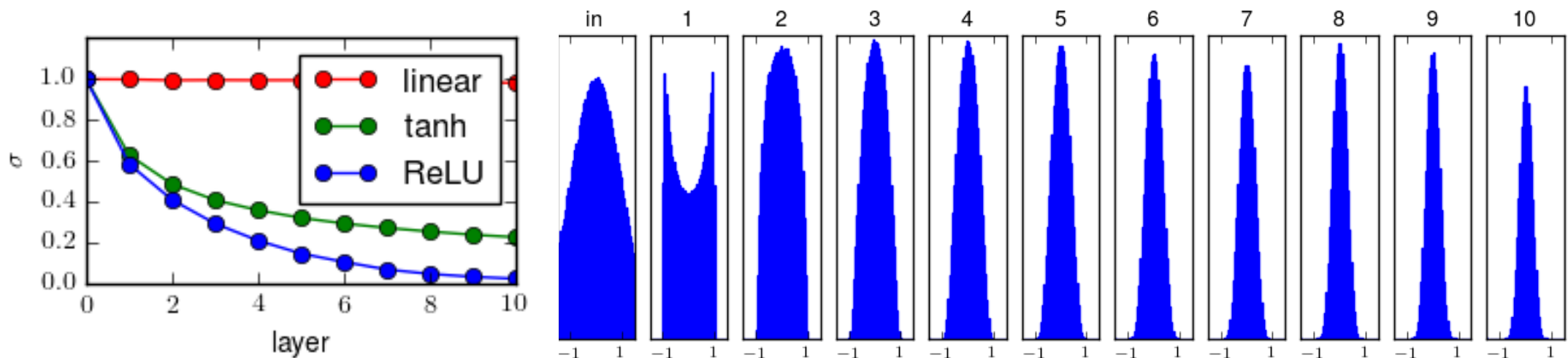
- ◆ Glorot and Bengio: *Understanding the difficulty of training deep feedforward neural networks*, 2010
- ◆ For the linear neuron $s = \sum_i w_i x_i$, let w_i and x_i be independent random variables, w_i and x_i are i.i.d., $\mathbb{E}(x_i) = \mathbb{E}(w_i) = 0$:

$$\begin{aligned}
 \text{Var}(s) &= \text{Var} \left(\sum_i w_i x_i \right) = \sum_i \text{Var}(w_i x_i) = \\
 &= \sum_i \mathbb{E} \left([w_i x_i - \mathbb{E}(w_i x_i)]^2 \right) = \sum_i \mathbb{E} \left([w_i x_i - \mathbb{E}(w_i) \mathbb{E}(x_i)]^2 \right) = \\
 &= \sum_i \mathbb{E}(w_i^2 x_i^2) = \sum_i \mathbb{E}(w_i^2) \mathbb{E}(x_i^2) = \\
 &= \sum_i \mathbb{E}([w_i - \mathbb{E}(w_i)]^2) \mathbb{E}([x_i - \mathbb{E}(x_i)]^2) = \\
 &= \sum_i \text{Var}(x_i) \text{Var}(w_i) = n_{\text{in}} \text{Var}(x) \text{Var}(w)
 \end{aligned}$$

Xavier Initialization (contd.)

- ◆ We have $\text{Var}(s) = n_{\text{in}} \text{Var}(x) \text{Var}(w)$
- ◆ We want $\text{Var}(s) = \text{Var}(x)$
- ◆ Choose $\text{Var}(w) = \frac{1}{n_{\text{in}}}$
- ◆ Works well for tanh as it is linear near zero
- ◆ Do not forget to standardize ANN input data

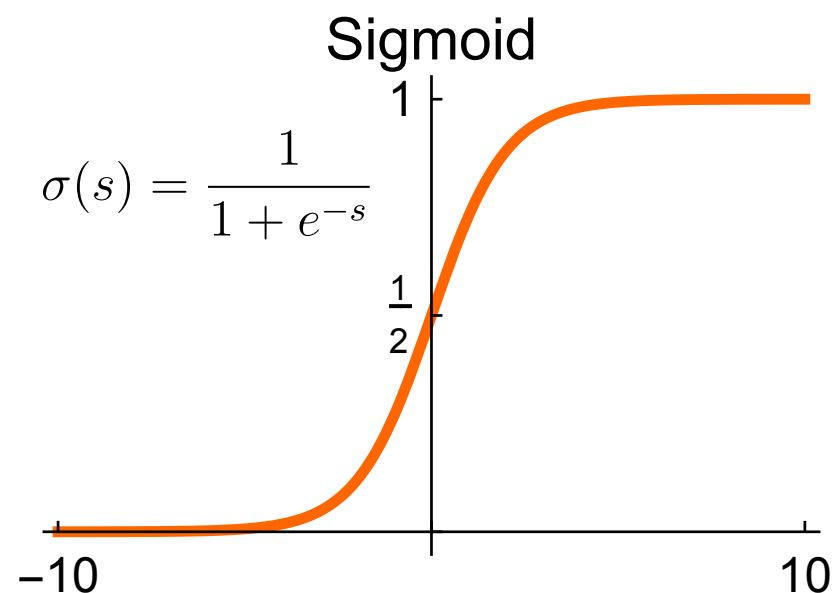
tanh



- ◆ How to deal with overfitting?
 - get more data
 - find a simpler model, search for optimal architecture, e.g., number, type, and size of layers
 - constrain model by *regularization*
- ◆ Most types of regularization are based on constraining the parameter space
- ◆ Bayesian point of view: introduce prior distribution on model parameters

L2 Regularization (Weight Decay): Motivation

- ◆ Limit hypothesis space by limiting the size of the weight vector
- ◆ For L2 regularization we will push down $\mathbf{w}^T \mathbf{w} = \|\mathbf{w}\|_2^2$
- ◆ You already know this from SVMs!
- ◆ *Intuition:* sigmoid-like neurons kept near zero potential (via small weights) behave similarly to linear neurons
- ◆ L2 regularization (weight decay): zero-mean Gaussian prior



Example: L2 Regularization for Linear Regression

- ◆ Recall the linear regression likelihood:

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

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

$$p(\mathbf{w}) = (2\pi\sigma_0^2)^{-\frac{1}{2}} e^{-\frac{1}{2\sigma_0^2}\mathbf{w}^T\mathbf{w}}$$

- ◆ Then the posterior is:

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

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

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

MAP Estimate

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

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

where

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

- ◆ We can omit C , define $\lambda = \frac{\sigma^2}{\sigma_0^2}$ and minimize the loss function:

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

- ◆ The term $\lambda \mathbf{w}^T \mathbf{w} = \lambda \|\mathbf{w}\|_2^2$ minimizes the size of the weight vector
- ◆ Note that we omit bias in $\lambda \mathbf{w}^T \mathbf{w}$

L2 Regularization Improves Numerical Stability

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

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

- ◆ It turns out that the solution is the minimizer of our *regularized* loss function:

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

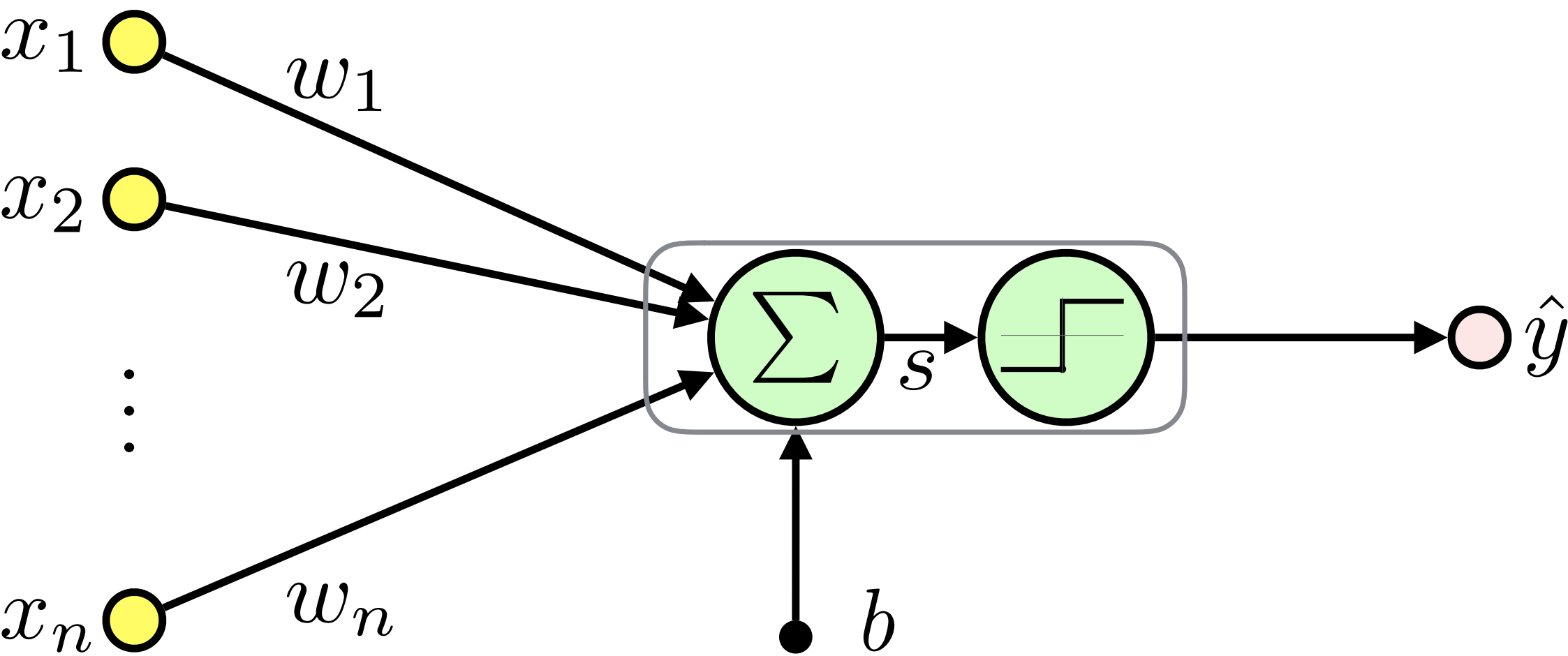
see seminar for the derivation

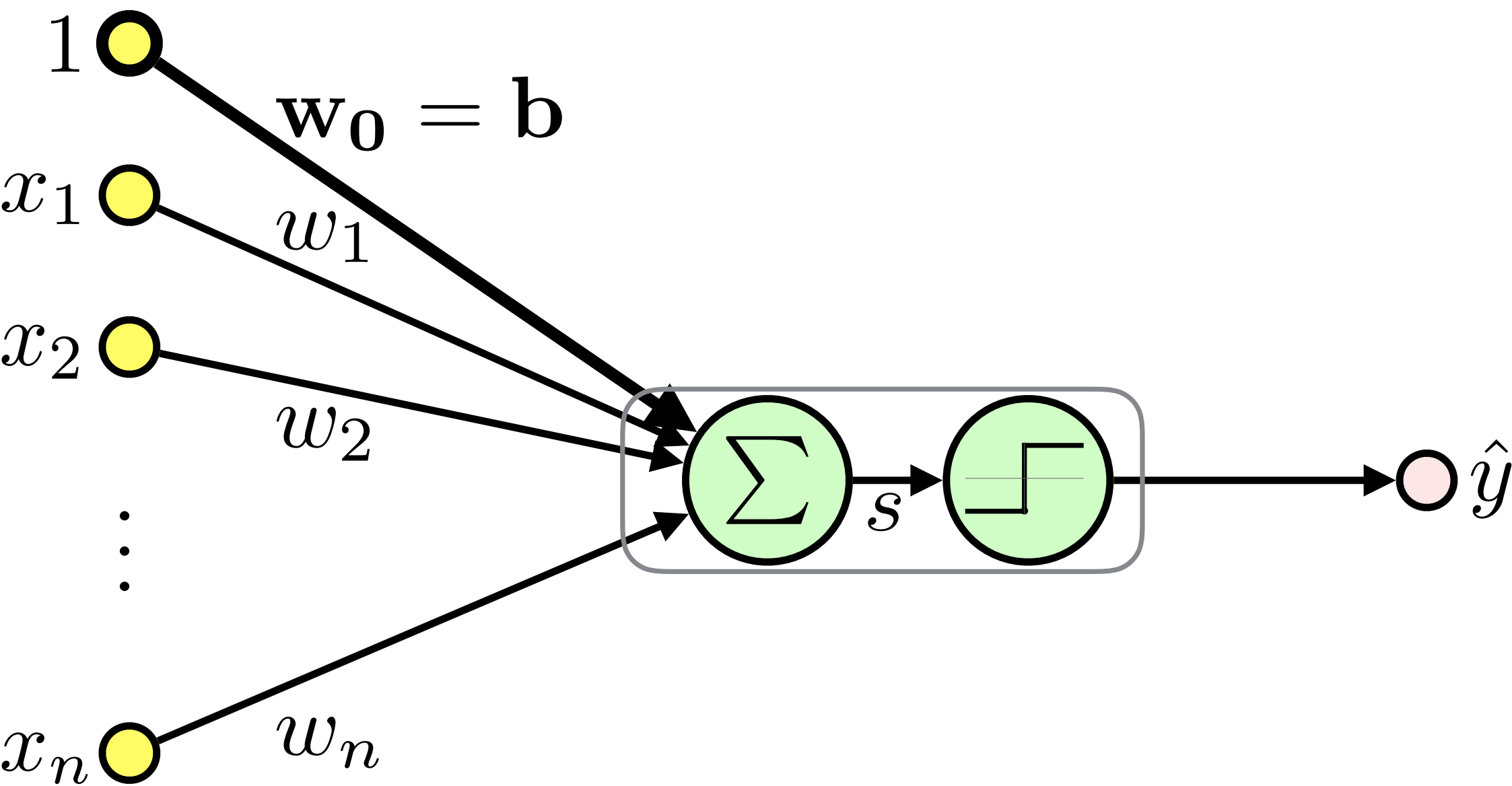
Other Regularization Related Approaches

- ◆ L1 regularization: sum absolute values, i.e., use $\lambda \|w\|_1$
- ◆ Randomize inputs: same as the weight decay for linear neurons
- ◆ Dataset augmentation
- ◆ Early stopping: start with small weights, stop when validation loss starts to grow, often used for limited time-budget
- ◆ Weight sharing and sparse connectivity: Convolutional Neural Networks (next lecture)
- ◆ Model averaging (see lectures on Ensembling)
- ◆ Dropout and DropConnect

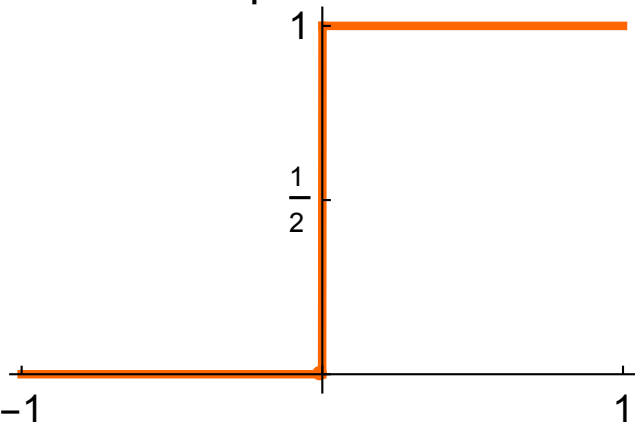
Next Lecture

- ◆ Stochastic Gradient Descent
- ◆ Deep Neural Networks
- ◆ Convolutional Neural Networks
- ◆ Transfer learning

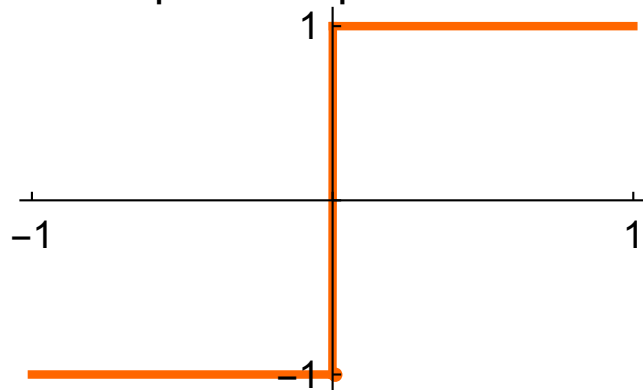




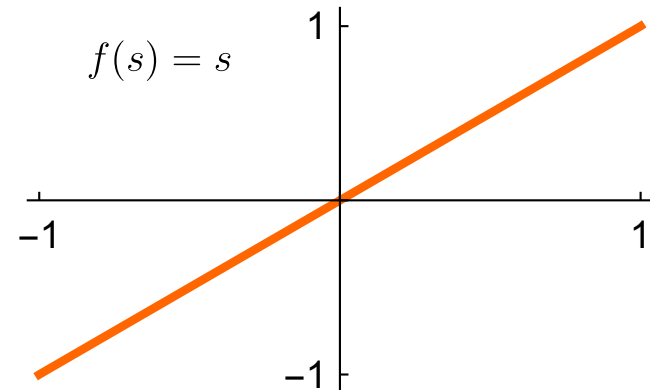
Step Function



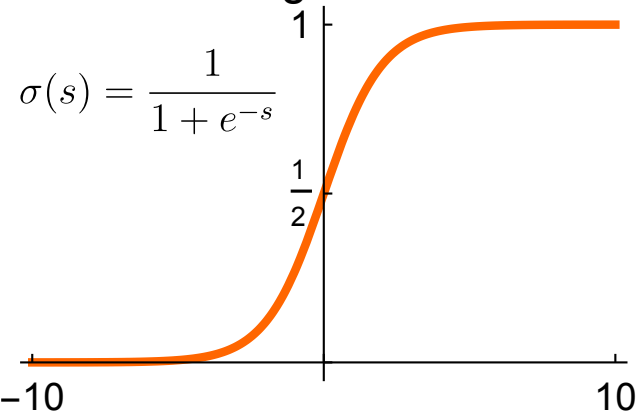
Bipolar Step Function



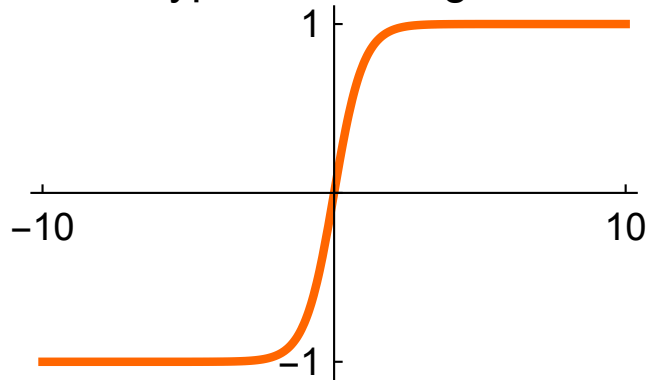
Linear



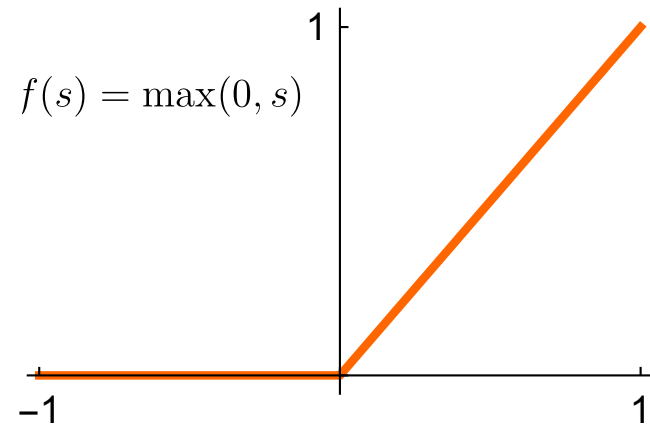
Sigmoid



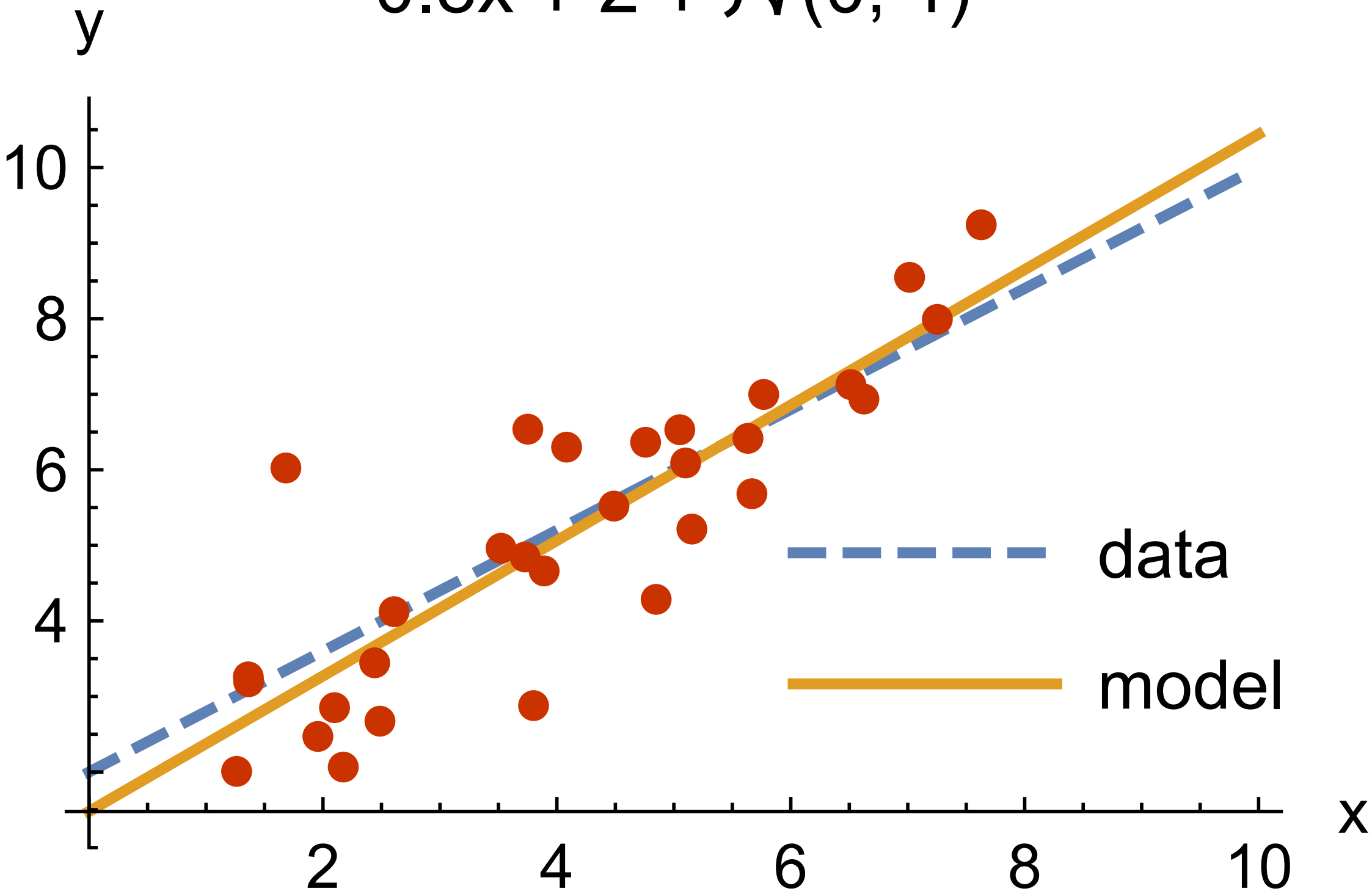
Hyperbolic Tangent



ReLU

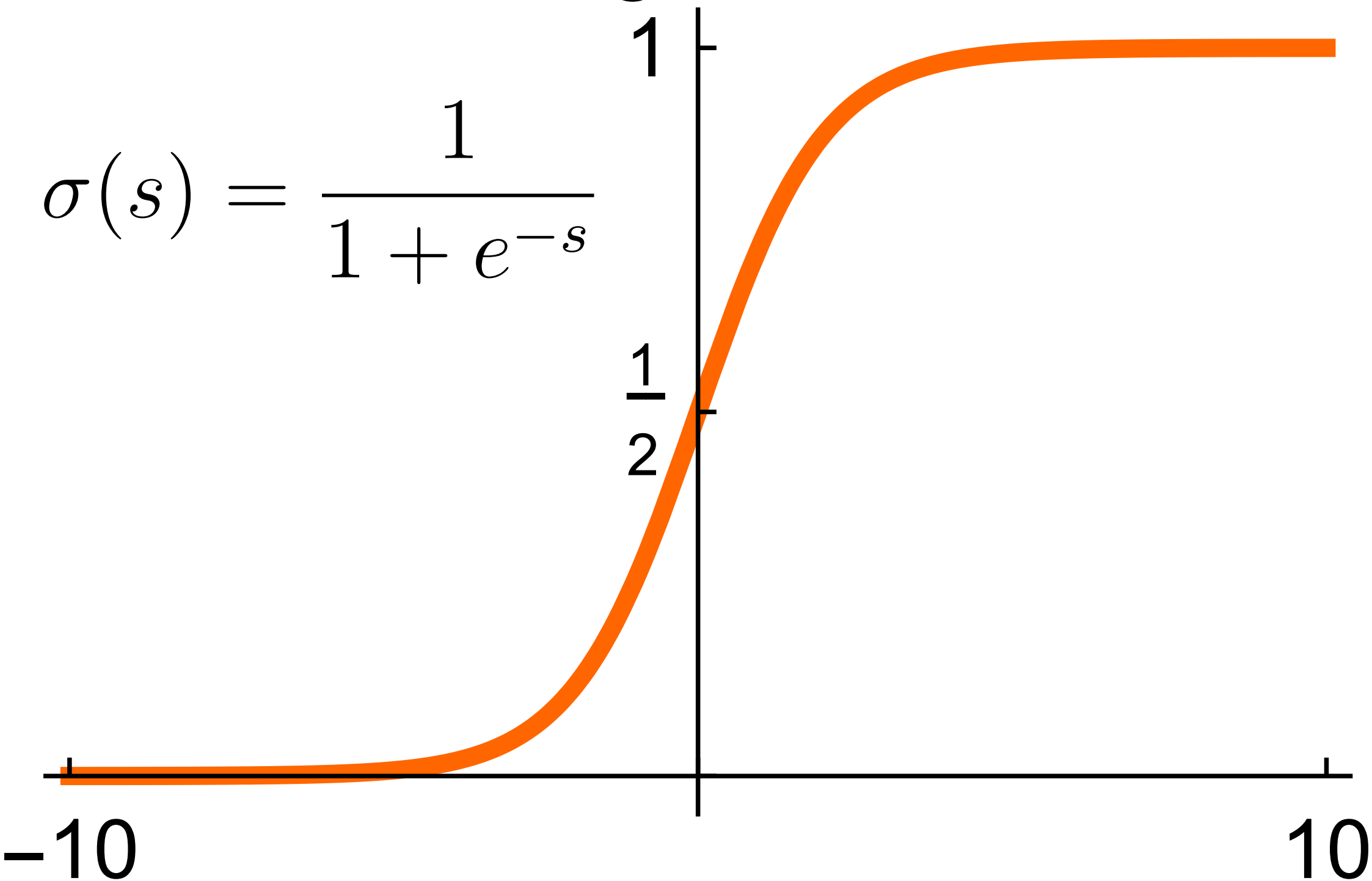


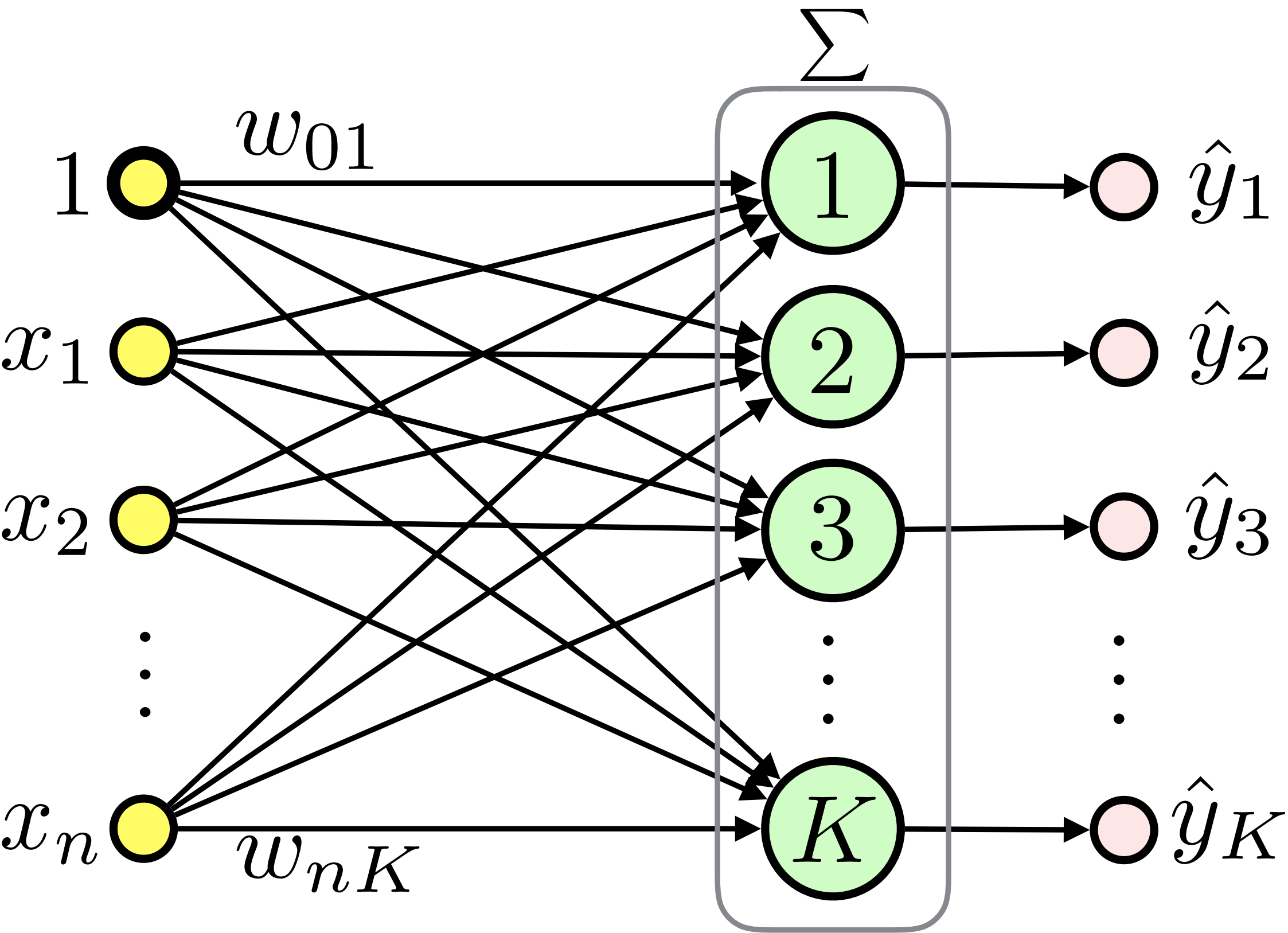
$$0.8x + 2 + \mathcal{N}(0, 1)$$

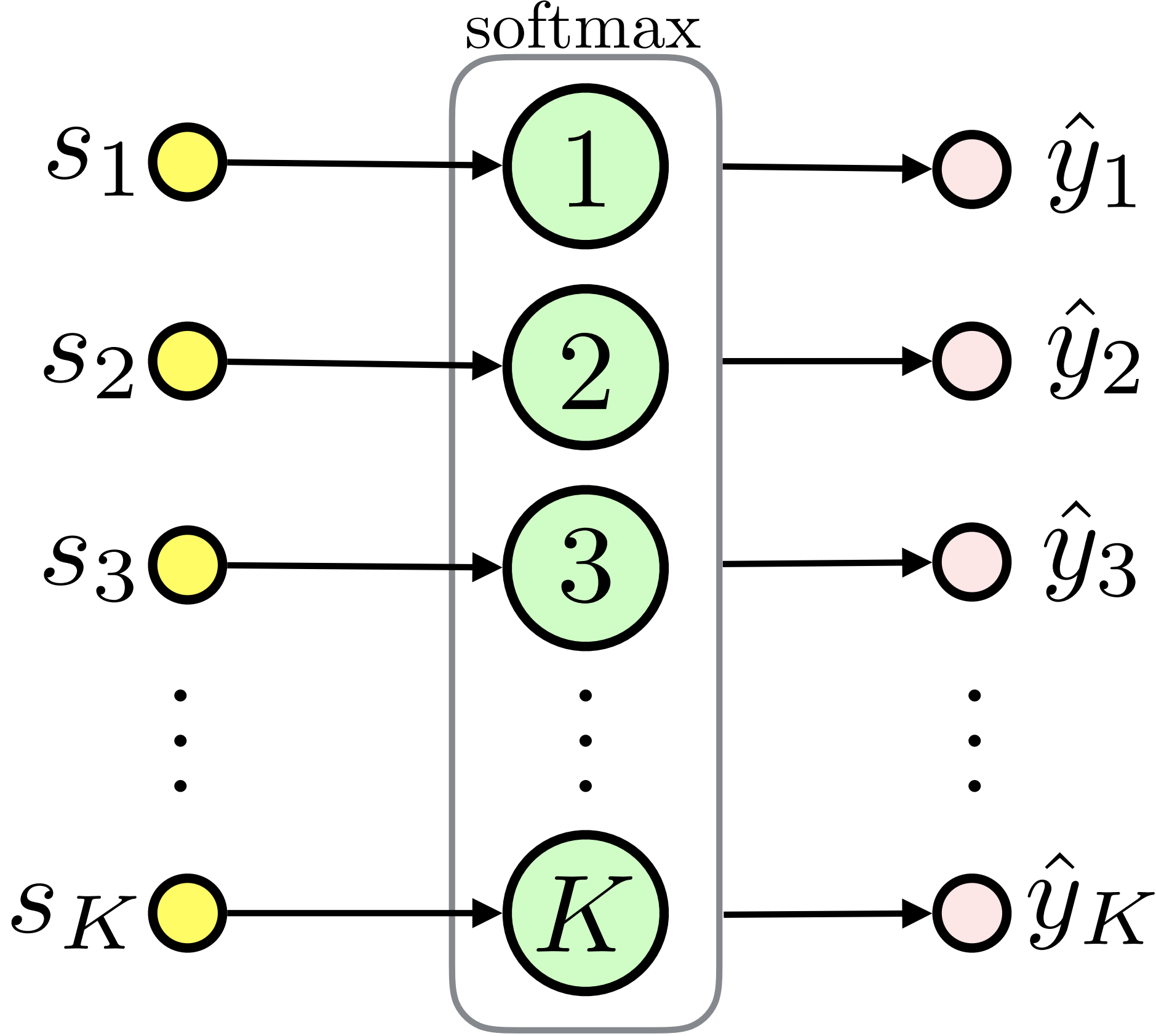


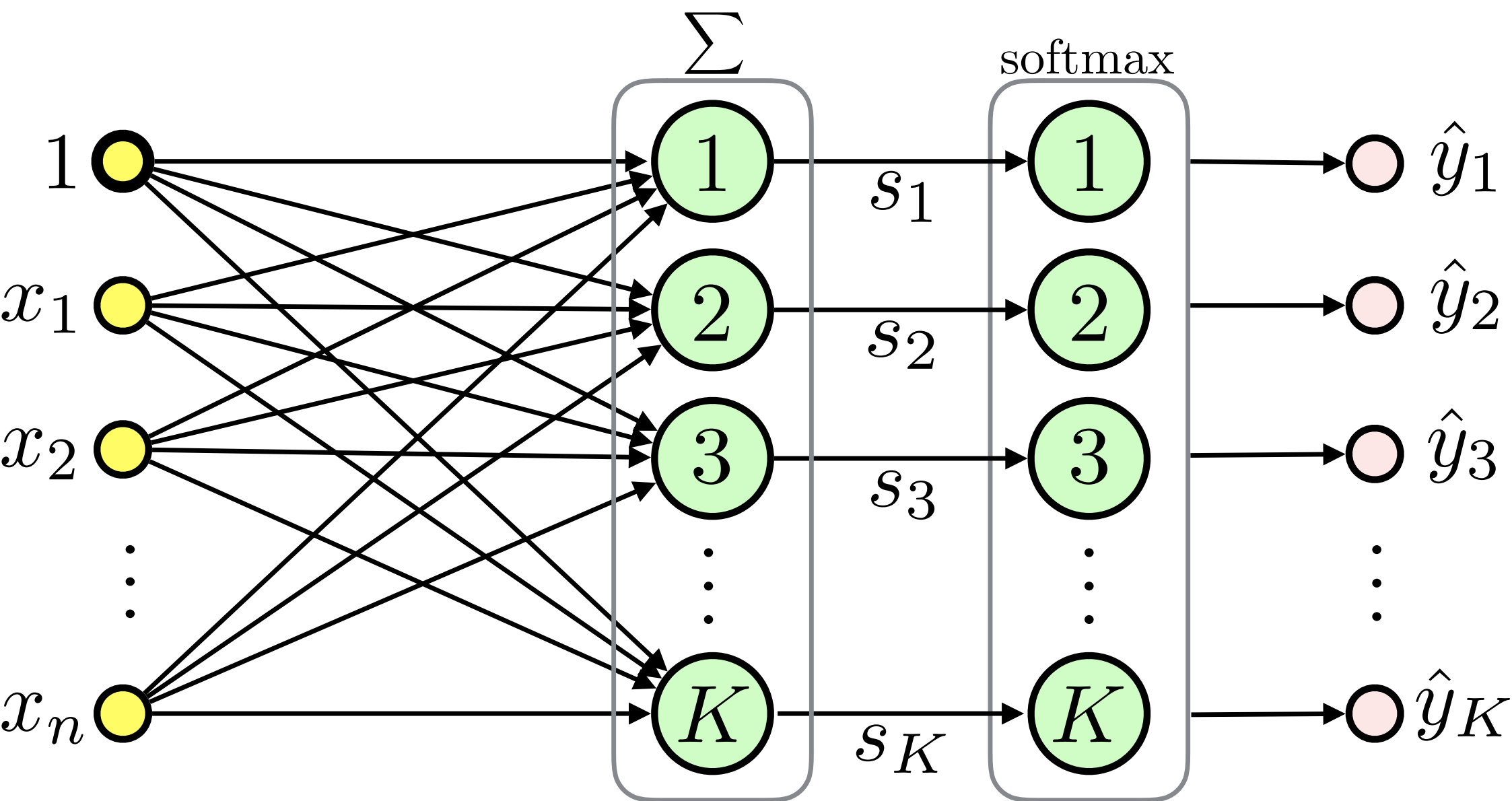
Sigmoid

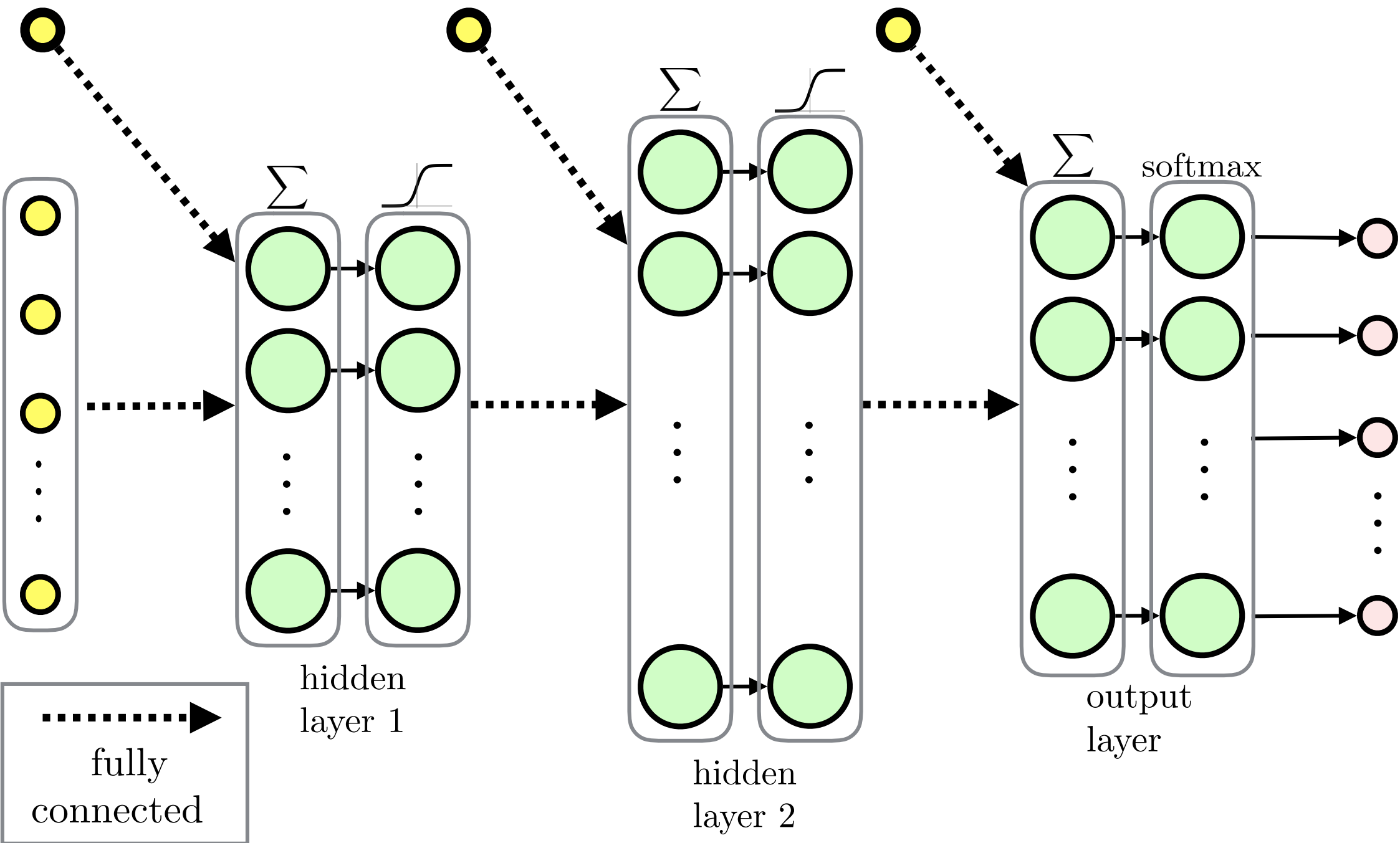
$$\sigma(s) = \frac{1}{1 + e^{-s}}$$

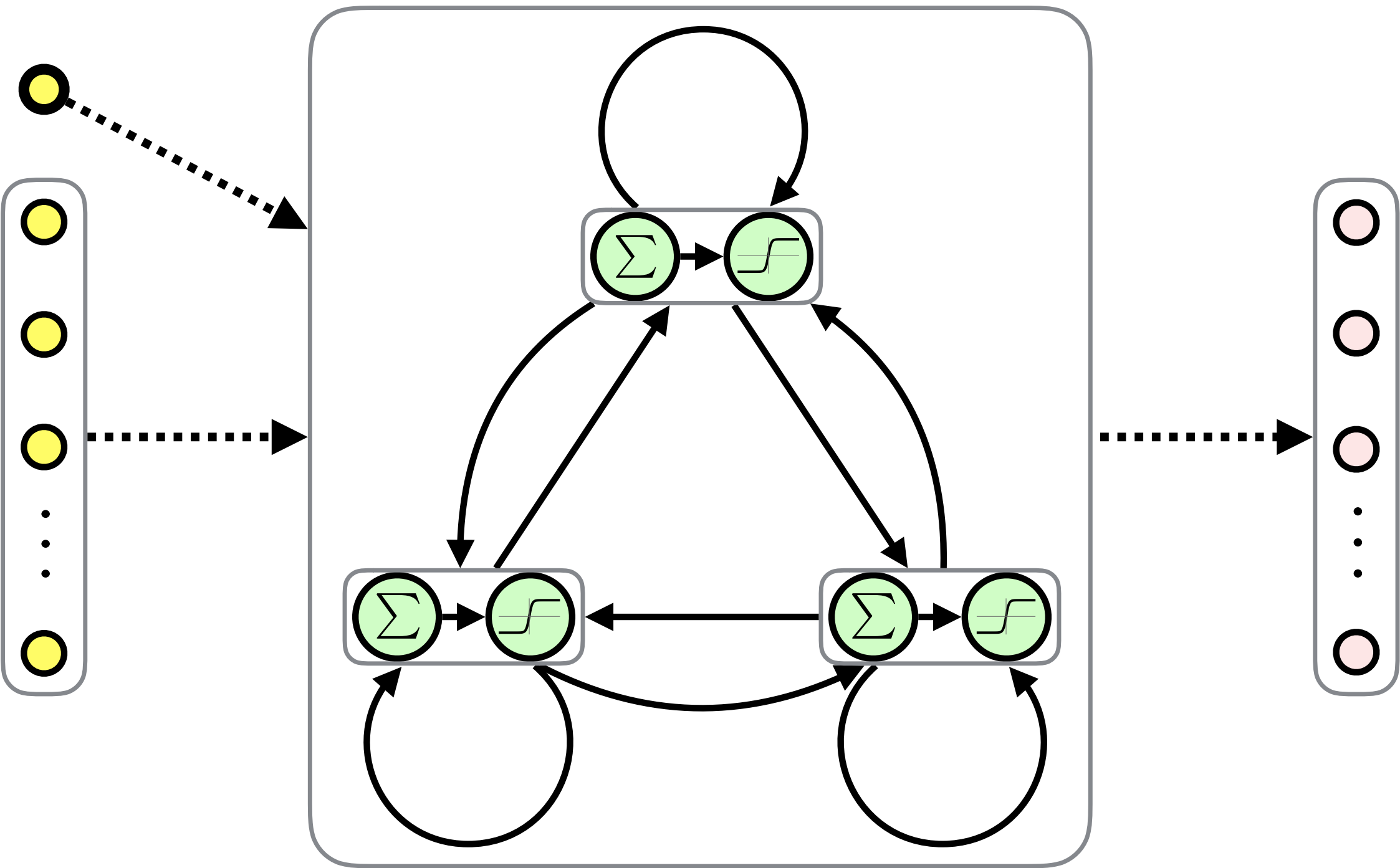


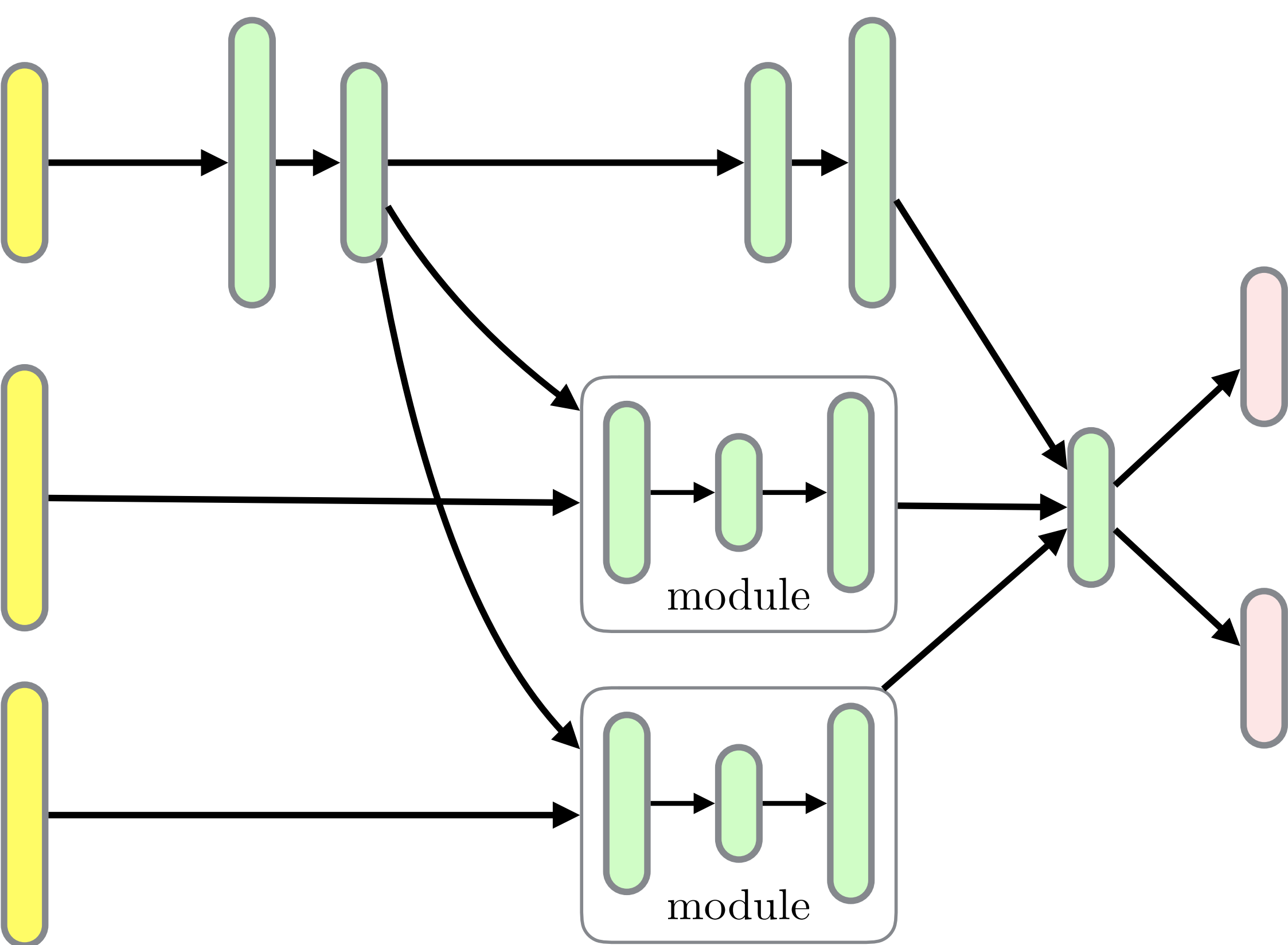


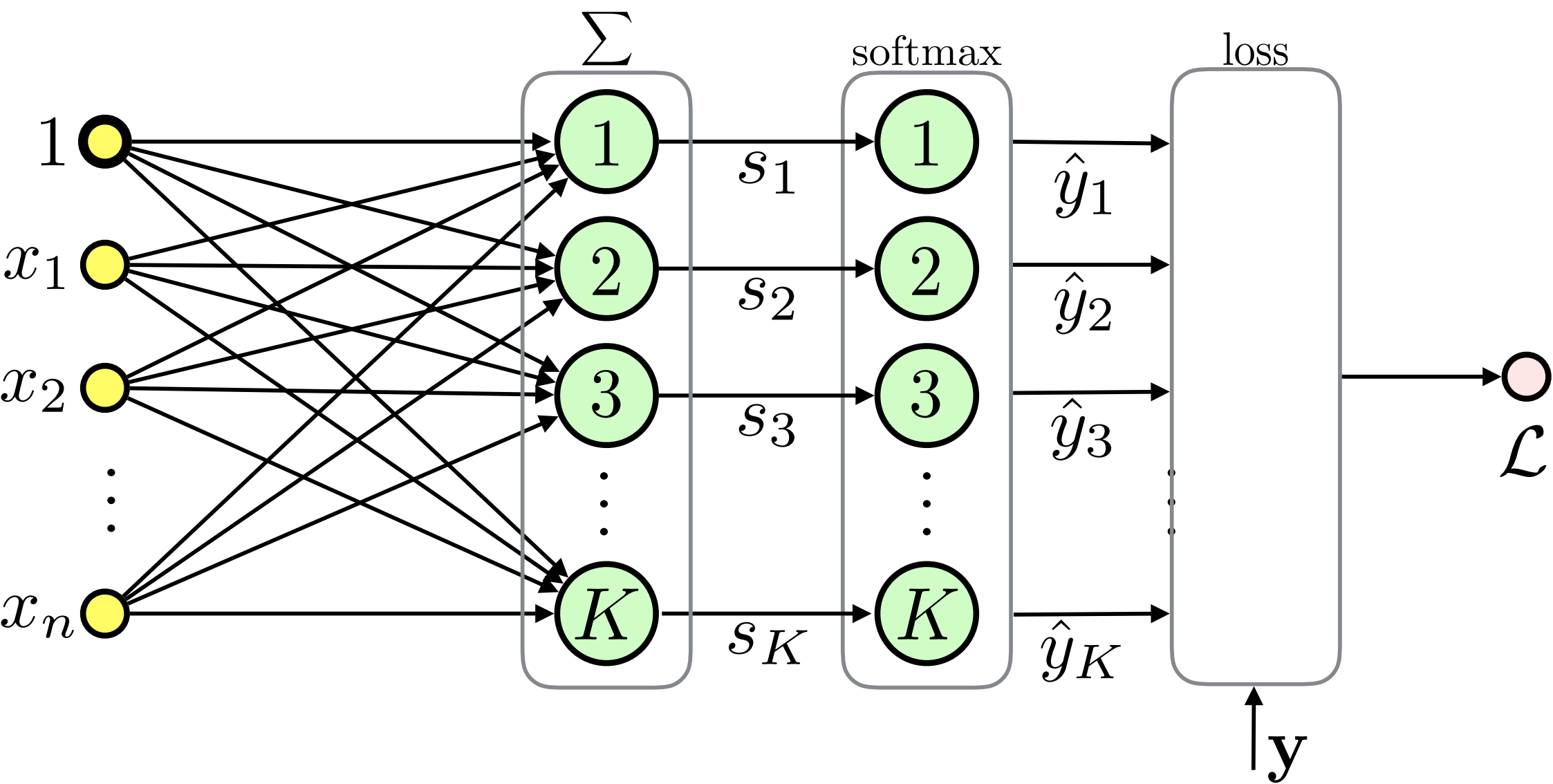


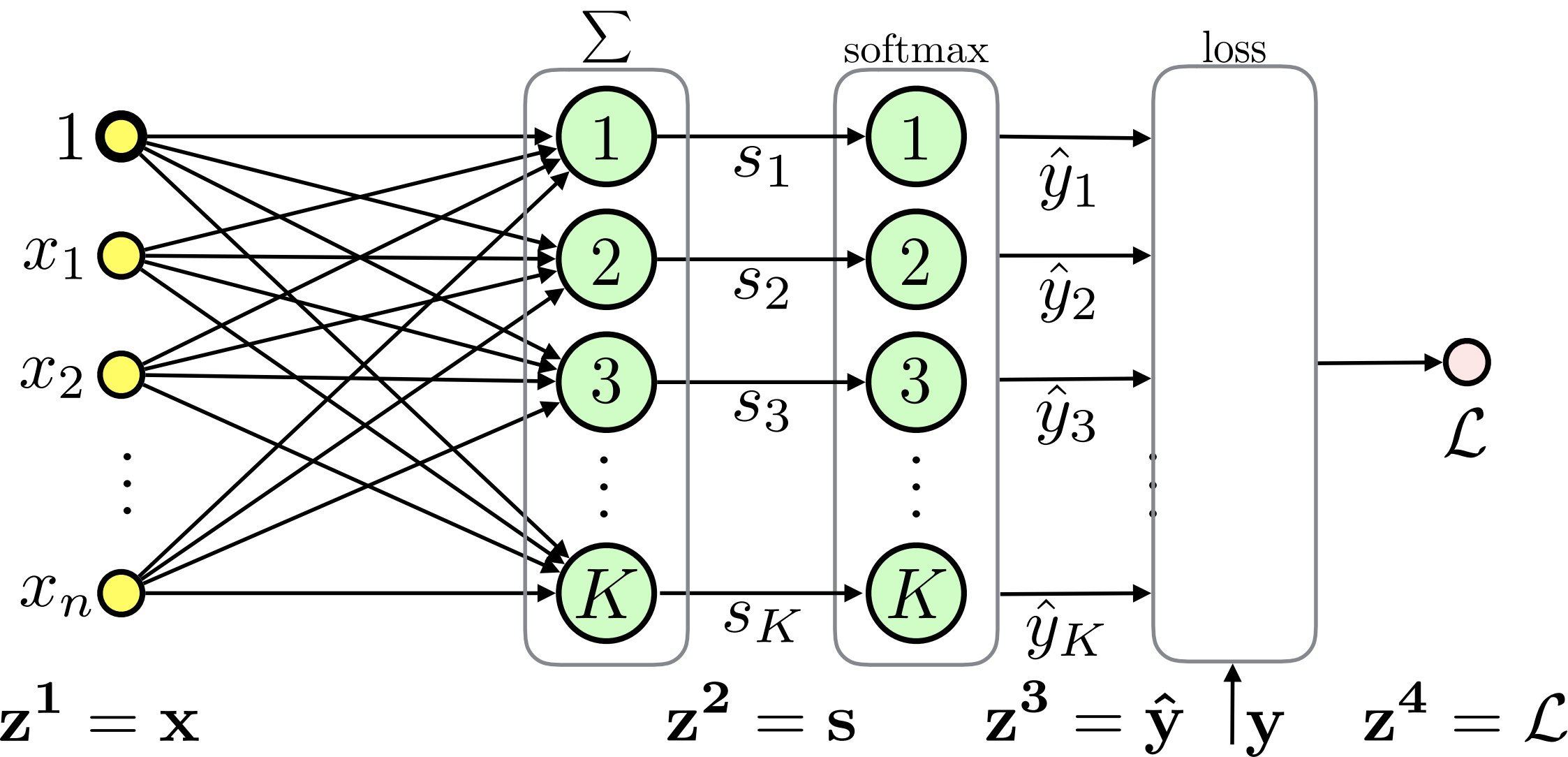


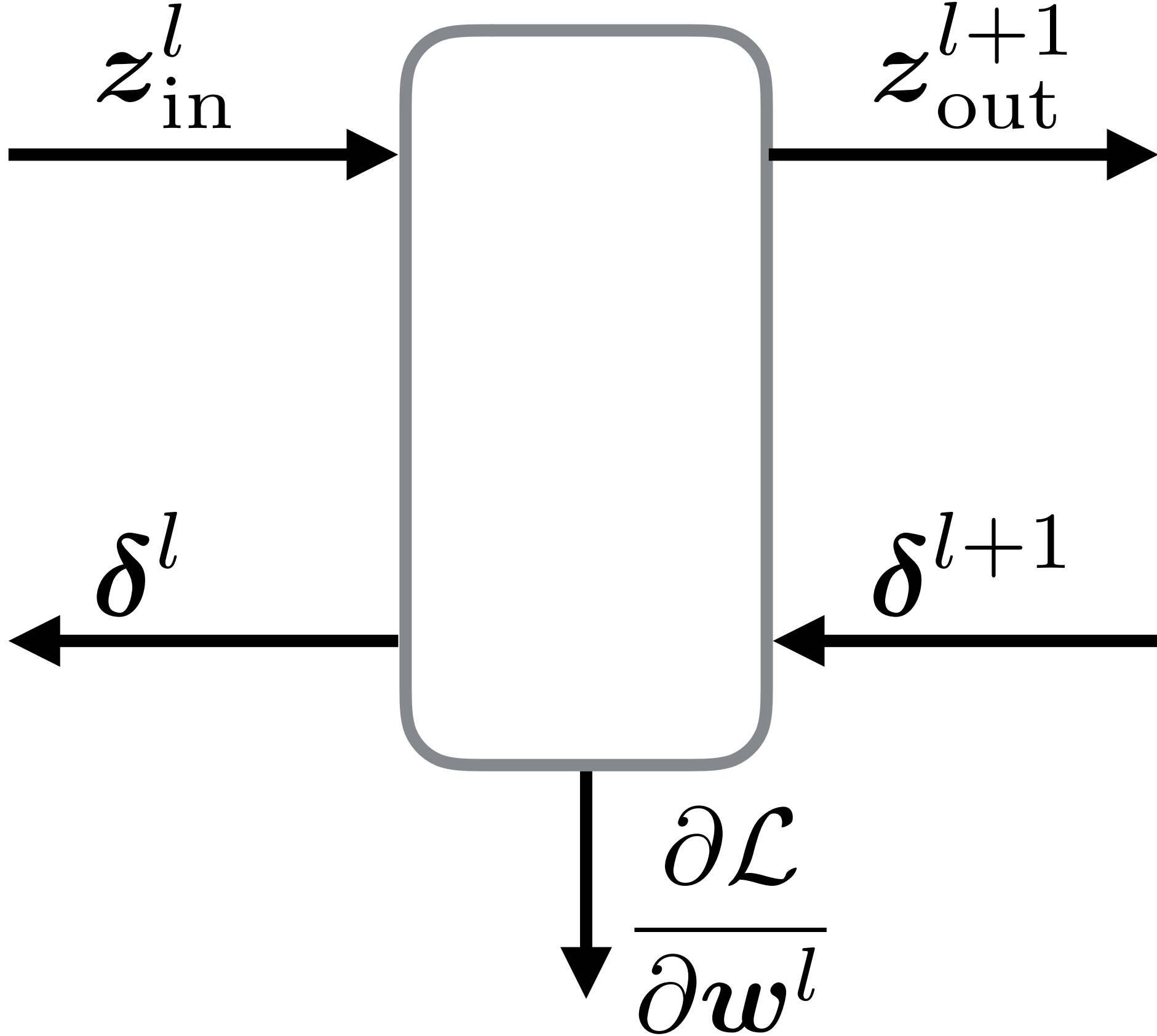


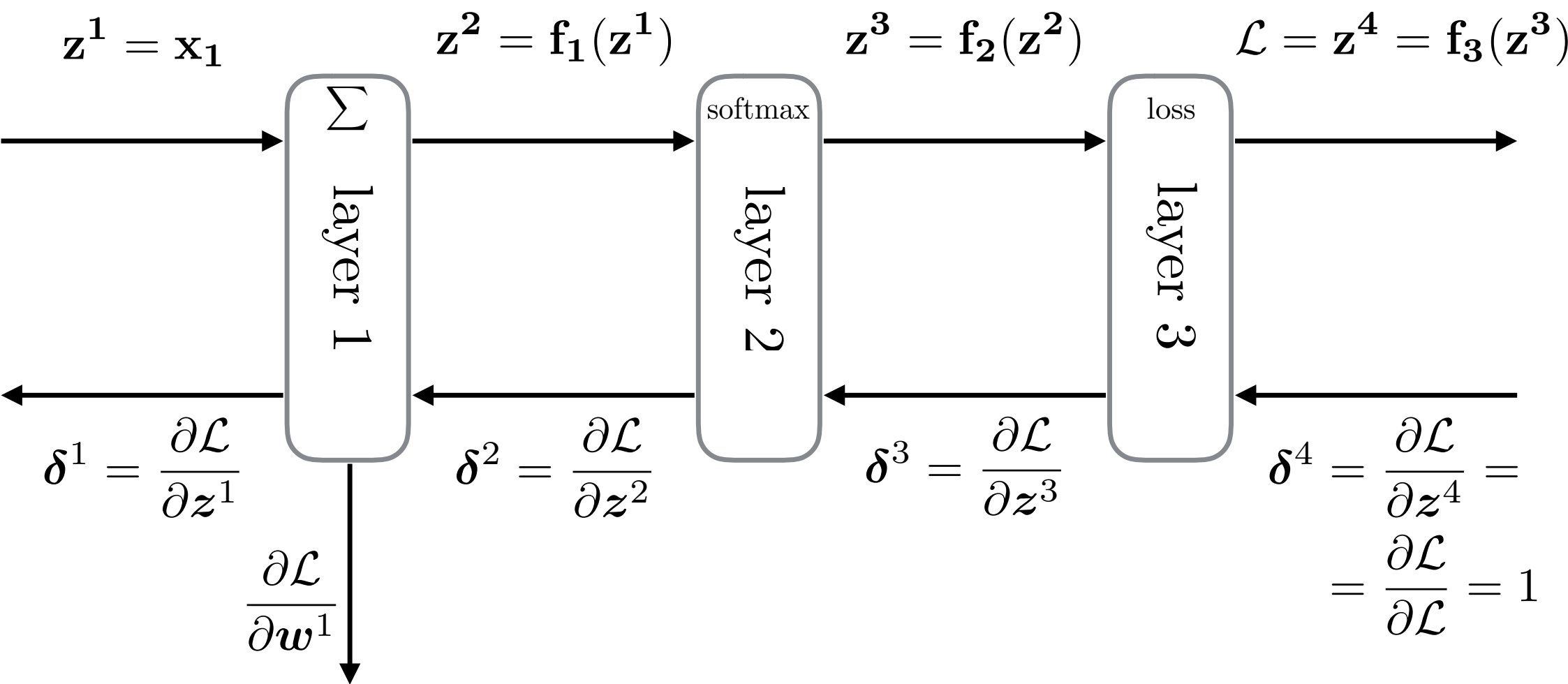


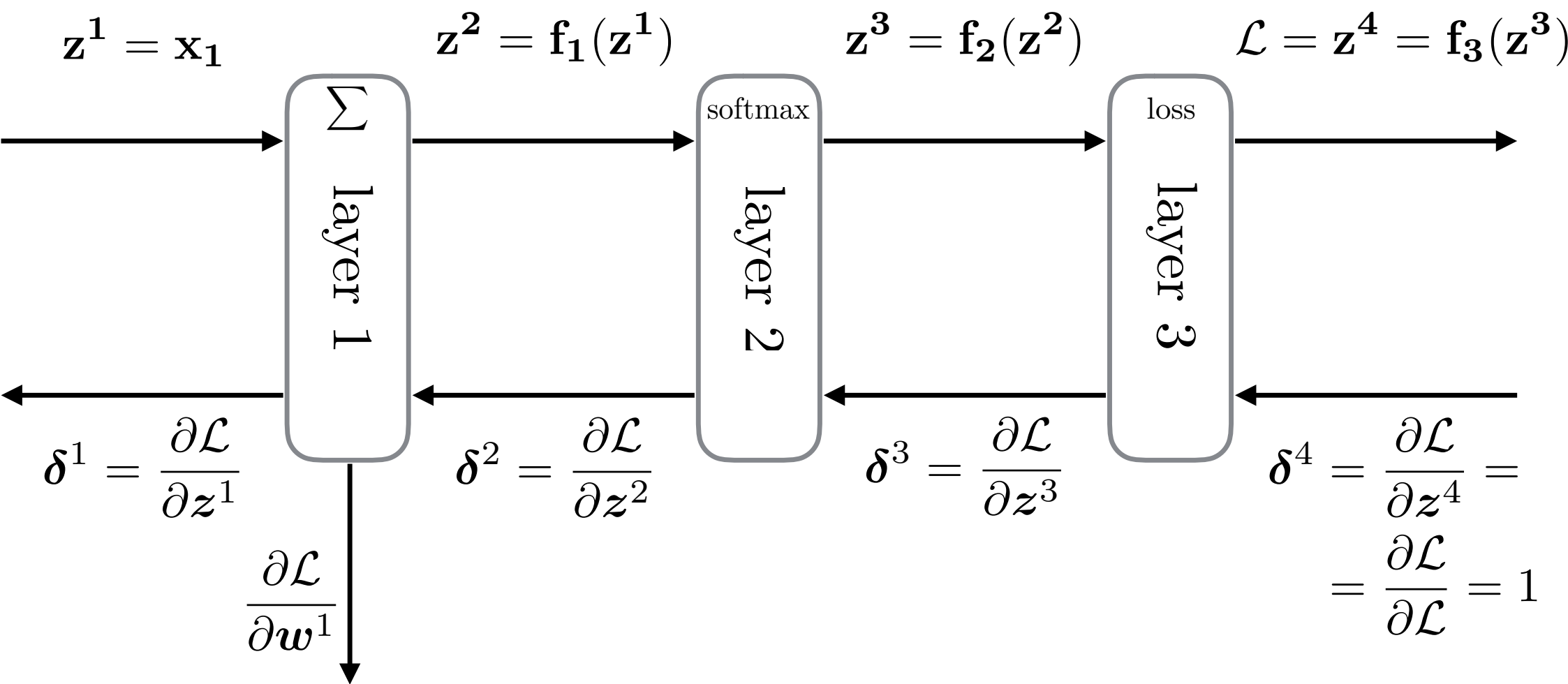




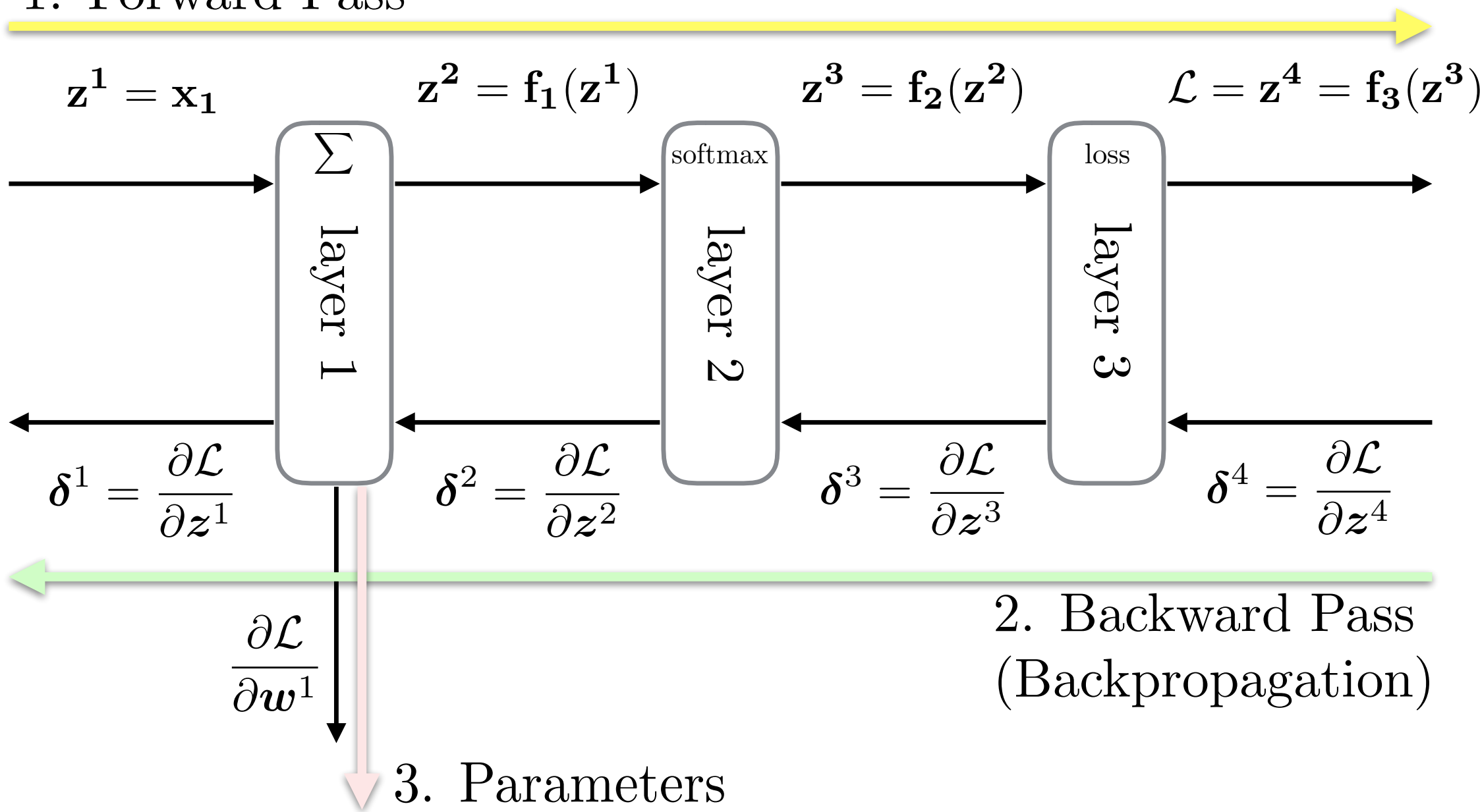


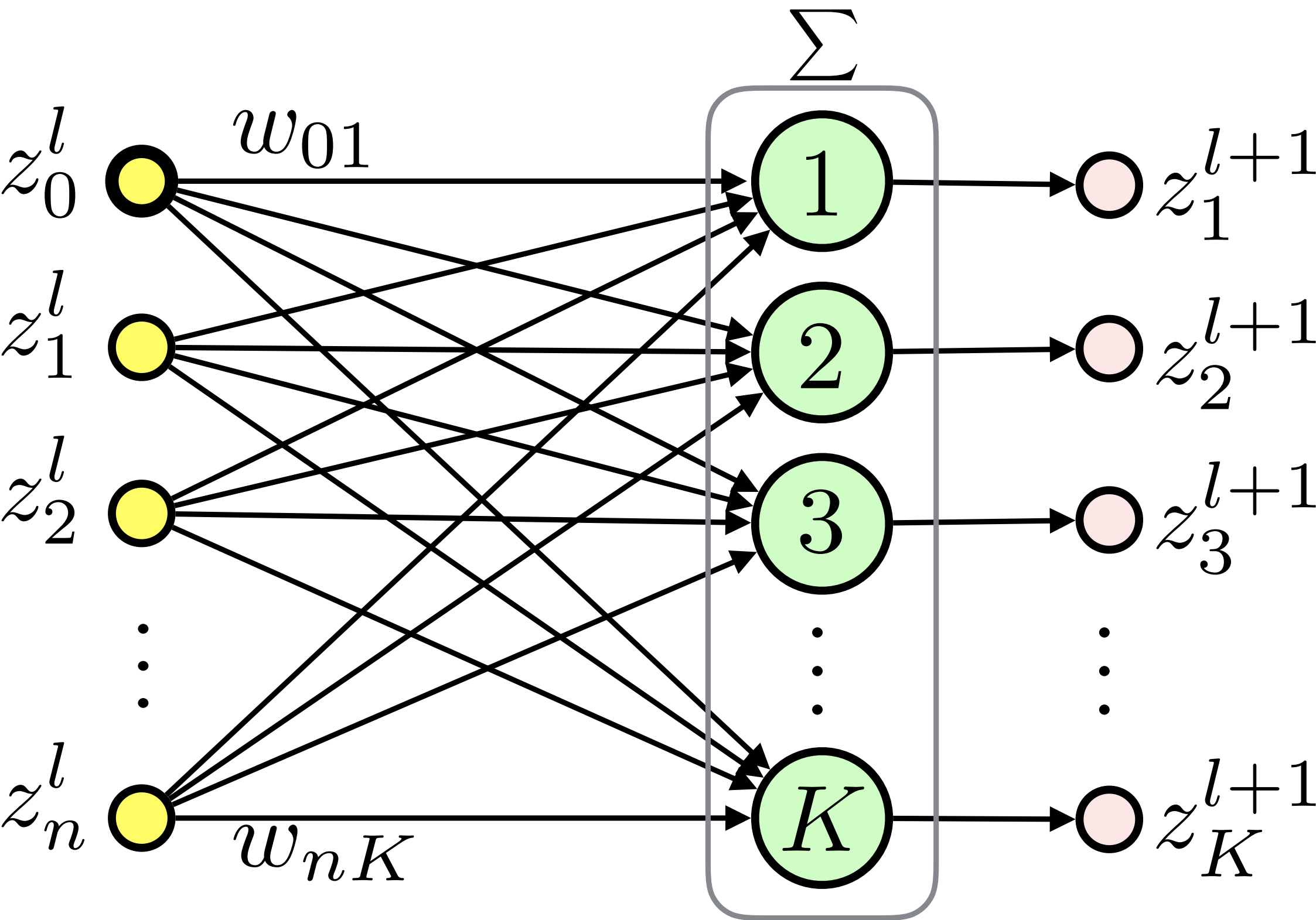




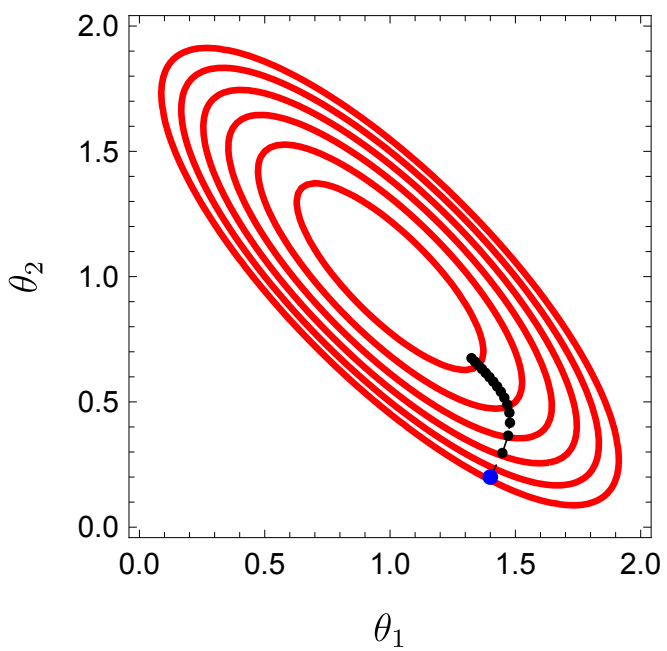


1. Forward Pass

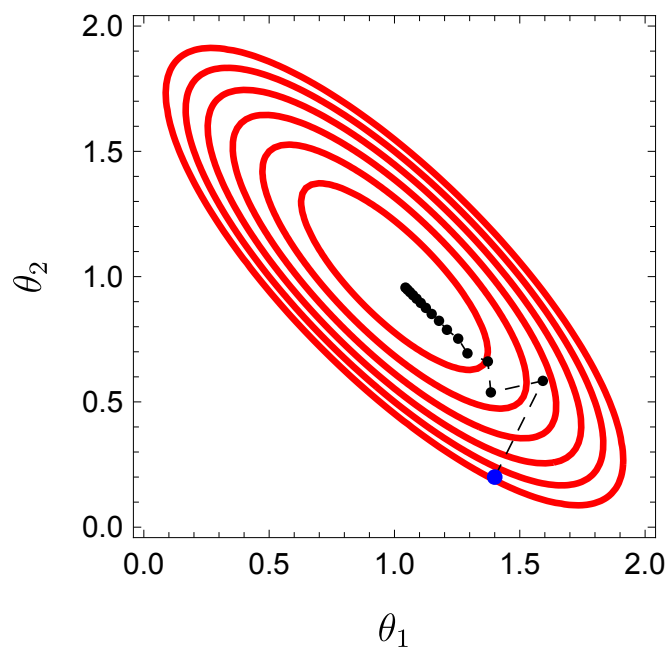




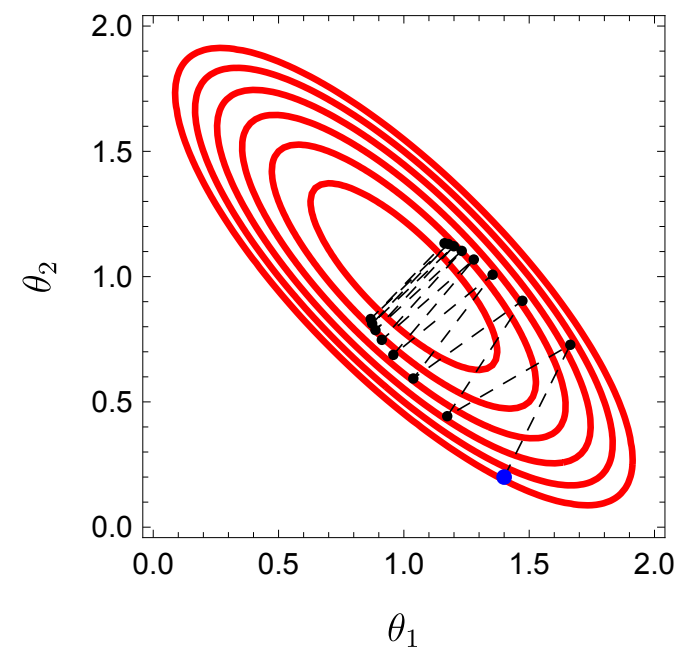
$N = 15, a = 0.1$

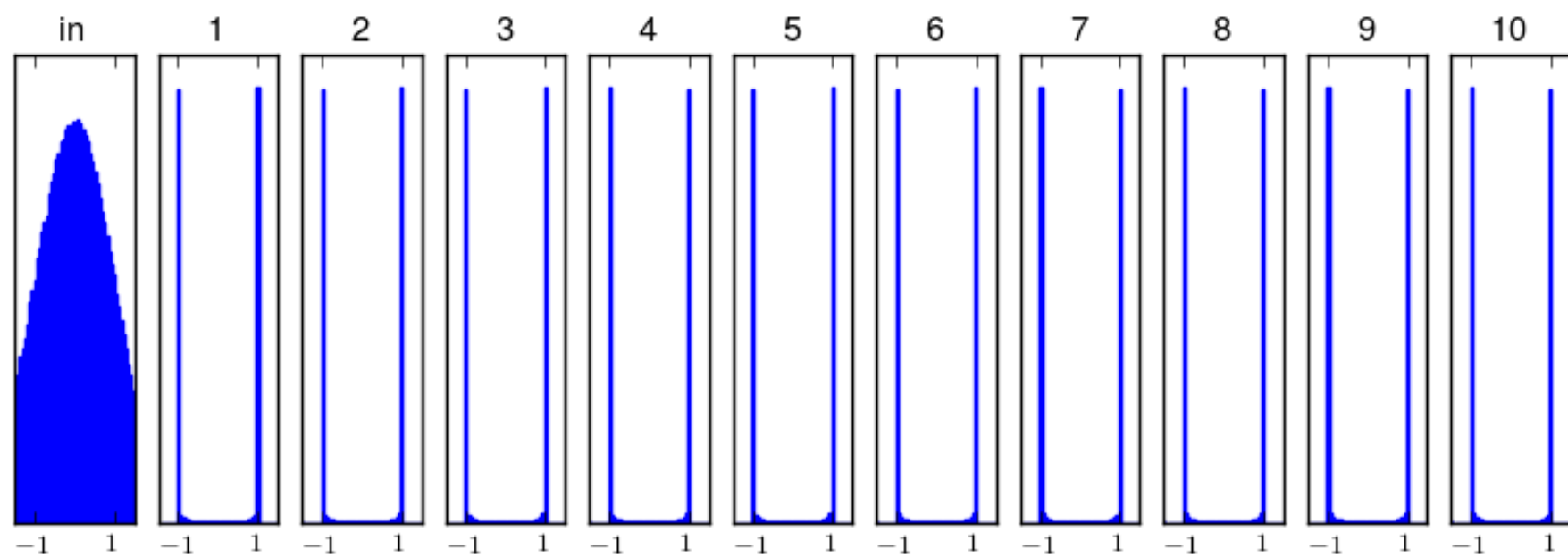


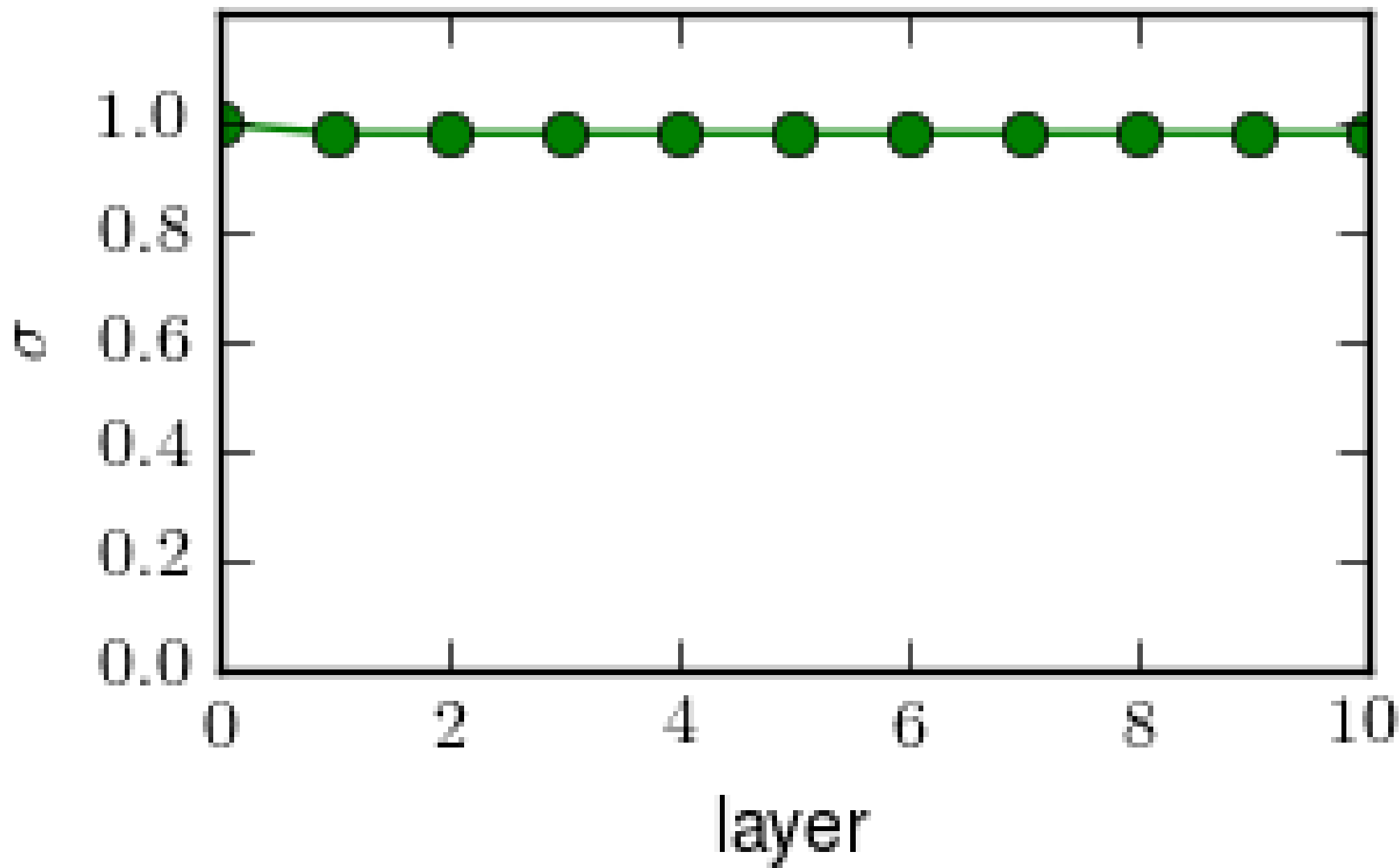
$N = 15, a = 0.4$

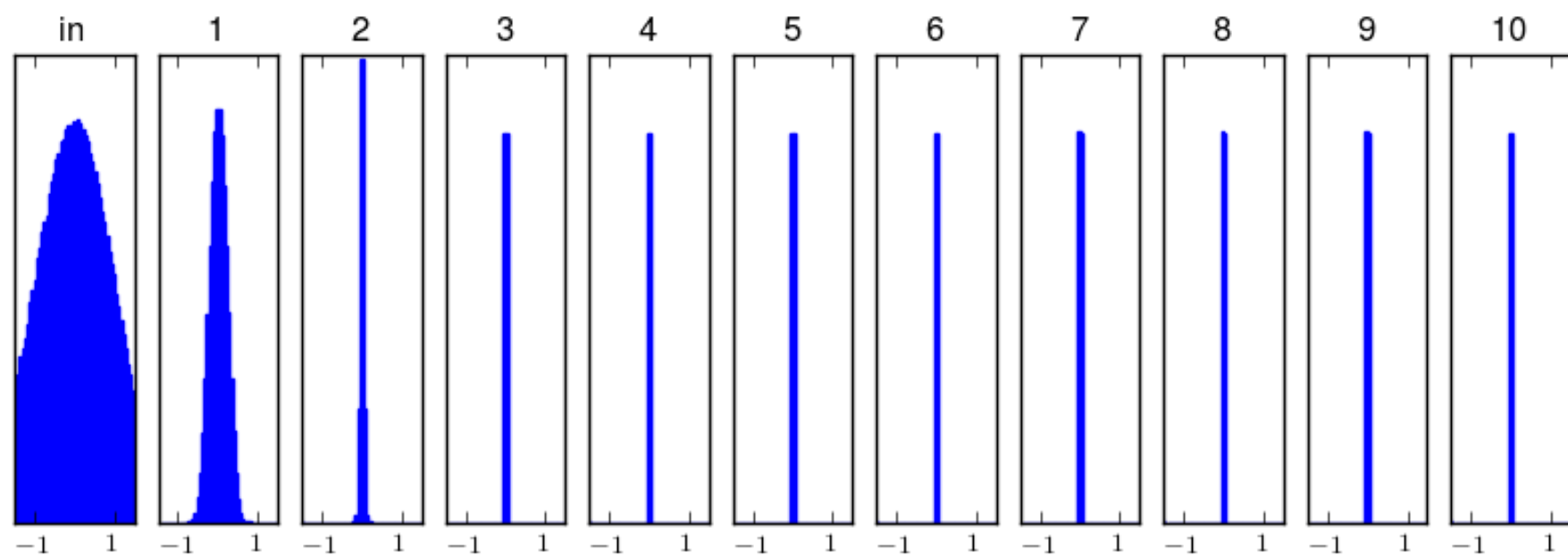


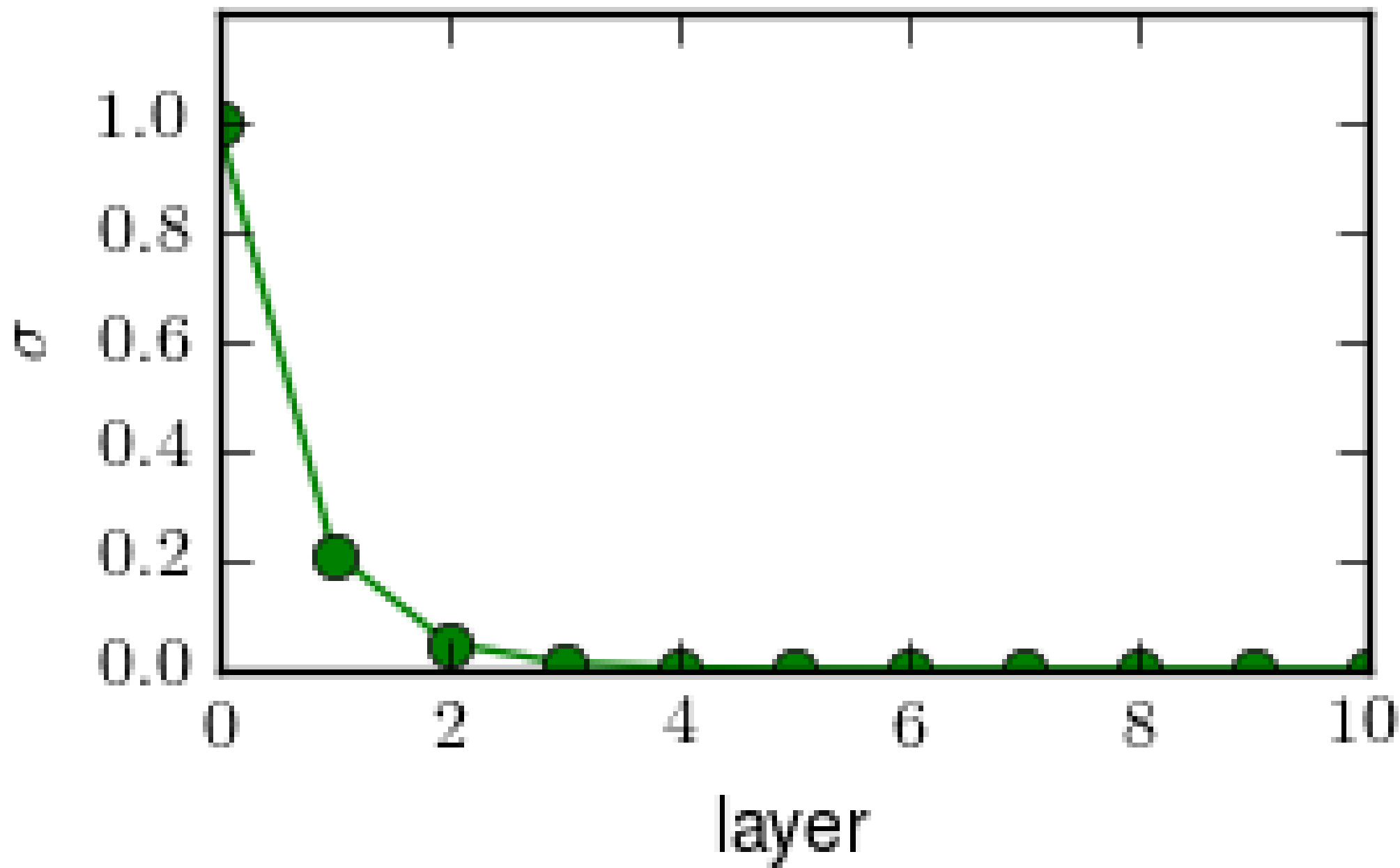
$N = 15, a = 0.55$



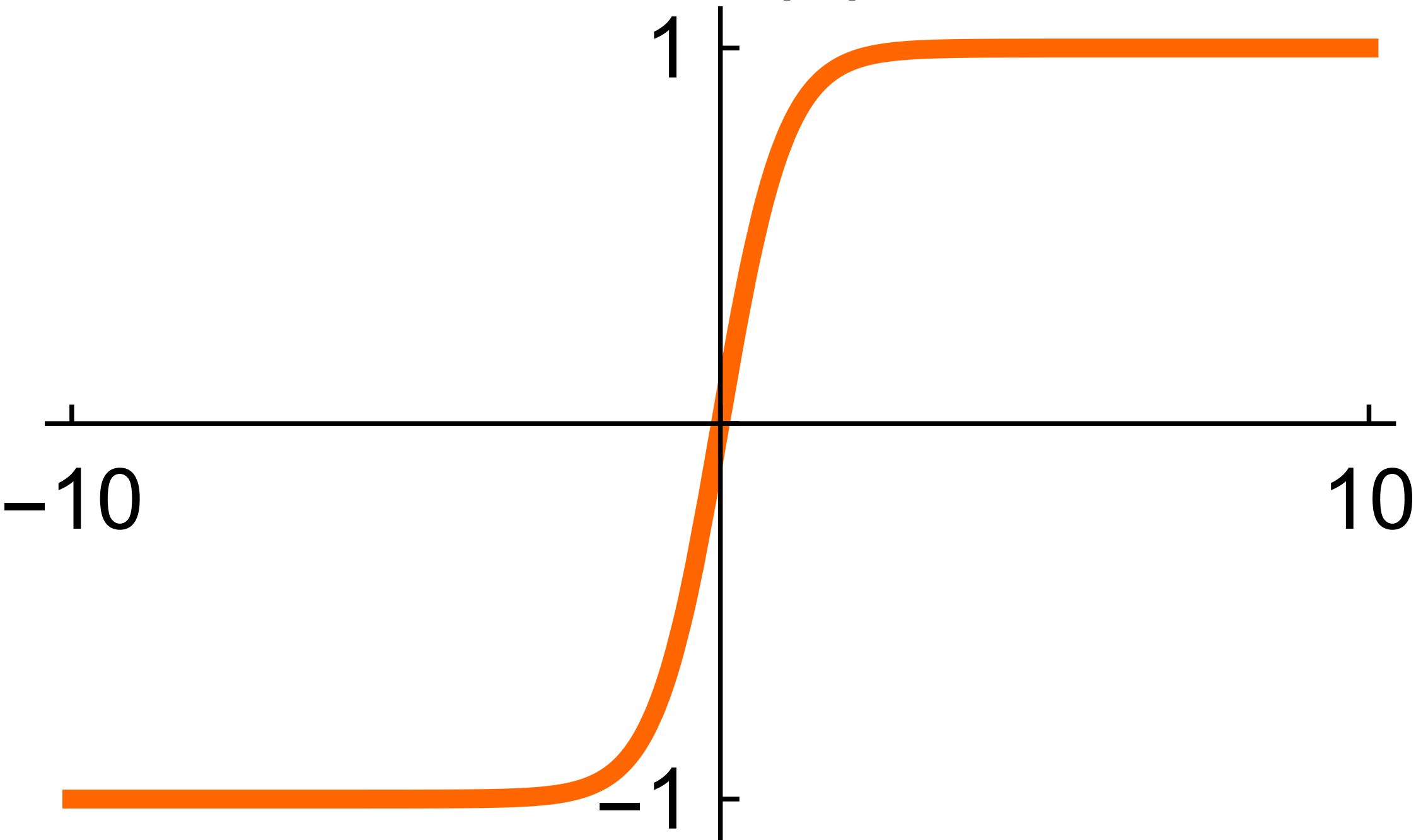








$\tanh(x)$



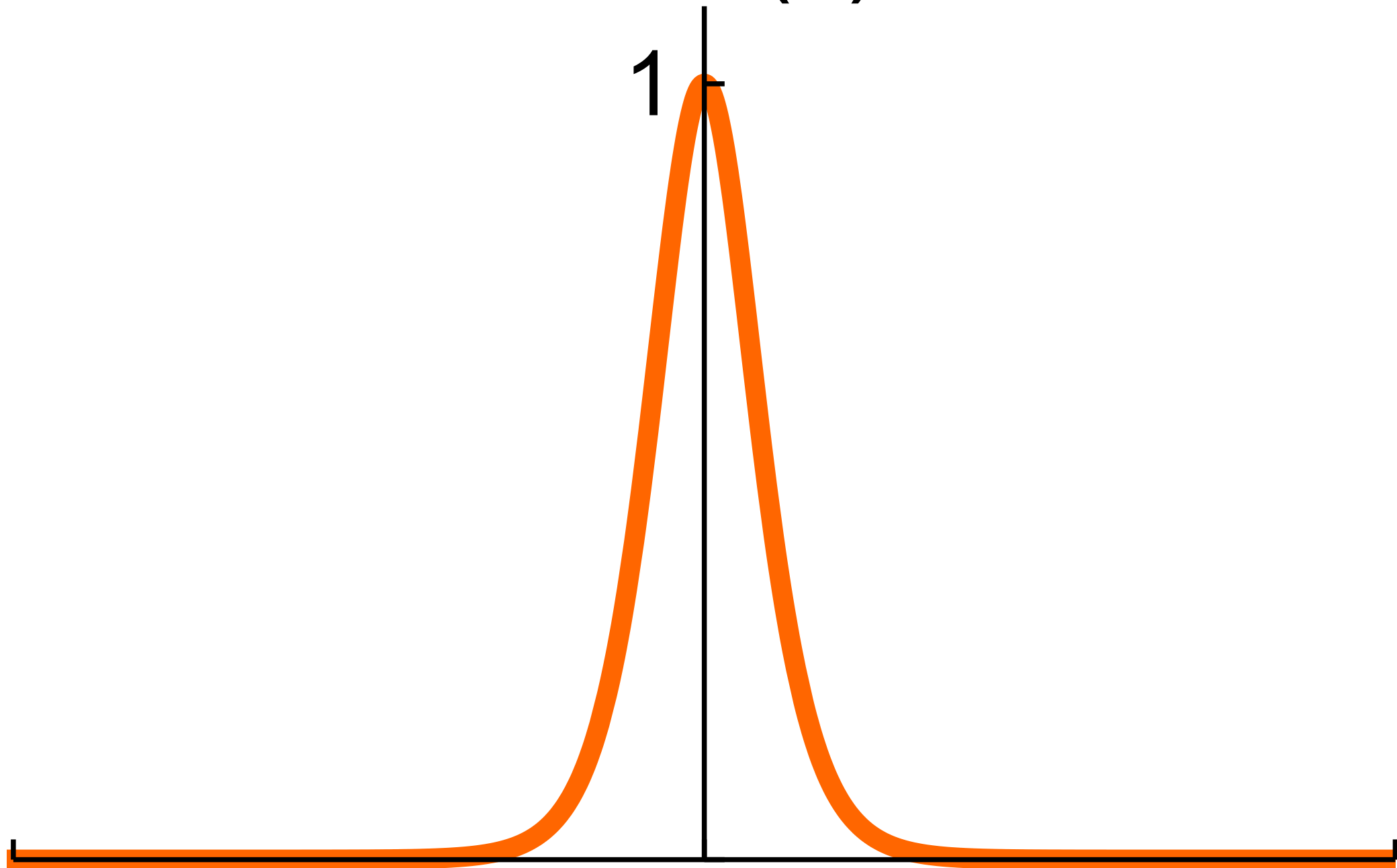
$\tanh'(x)$

1

-10

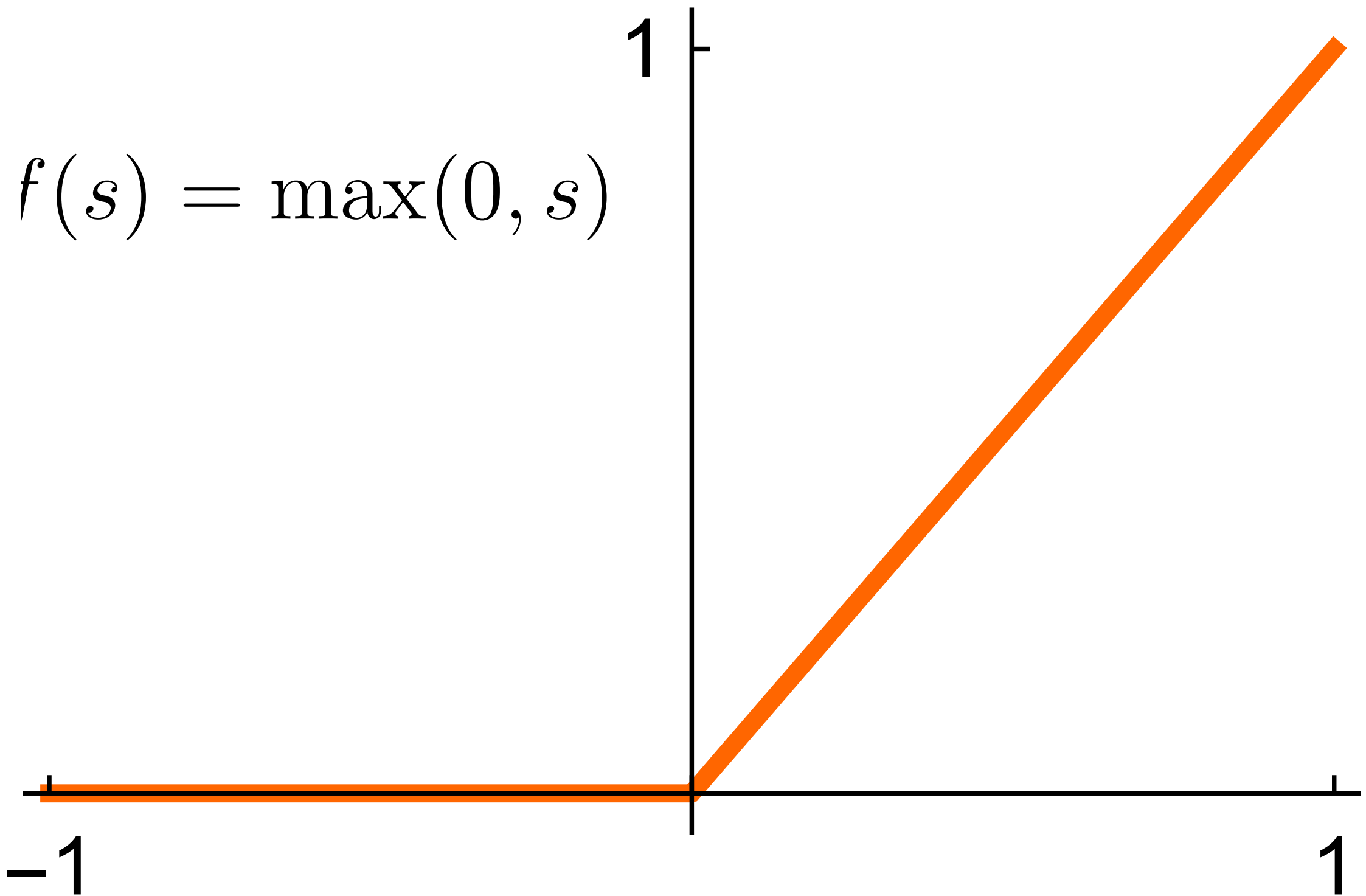
0

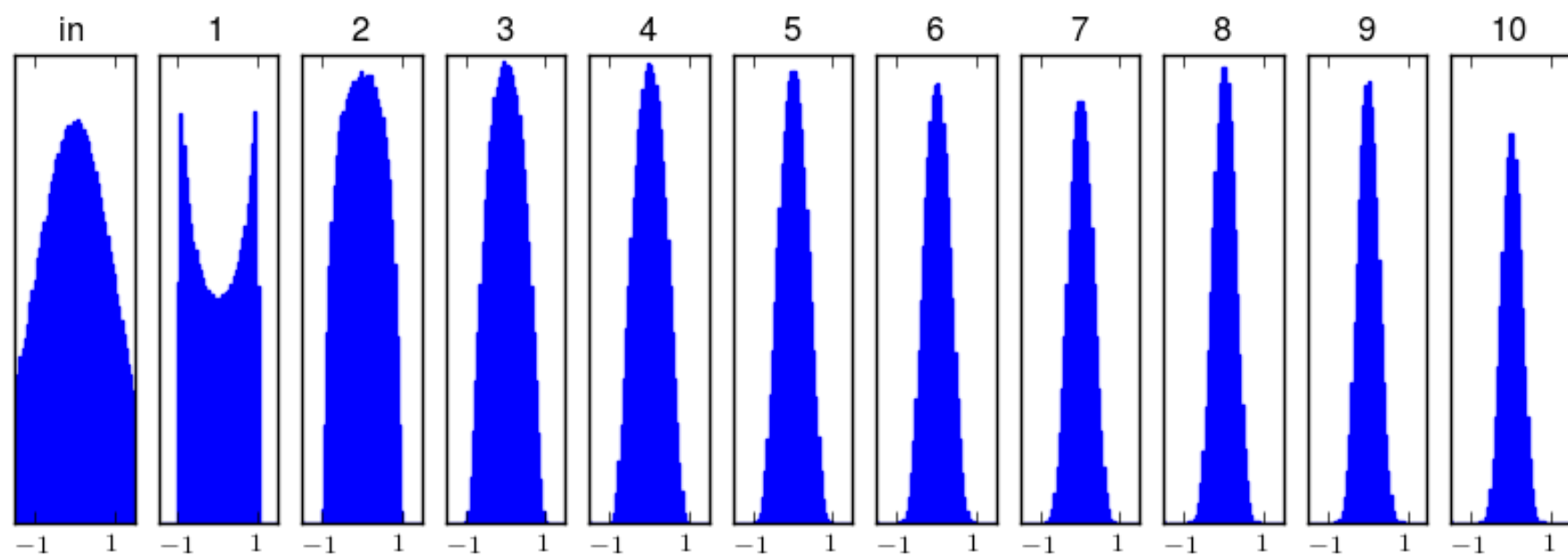
10

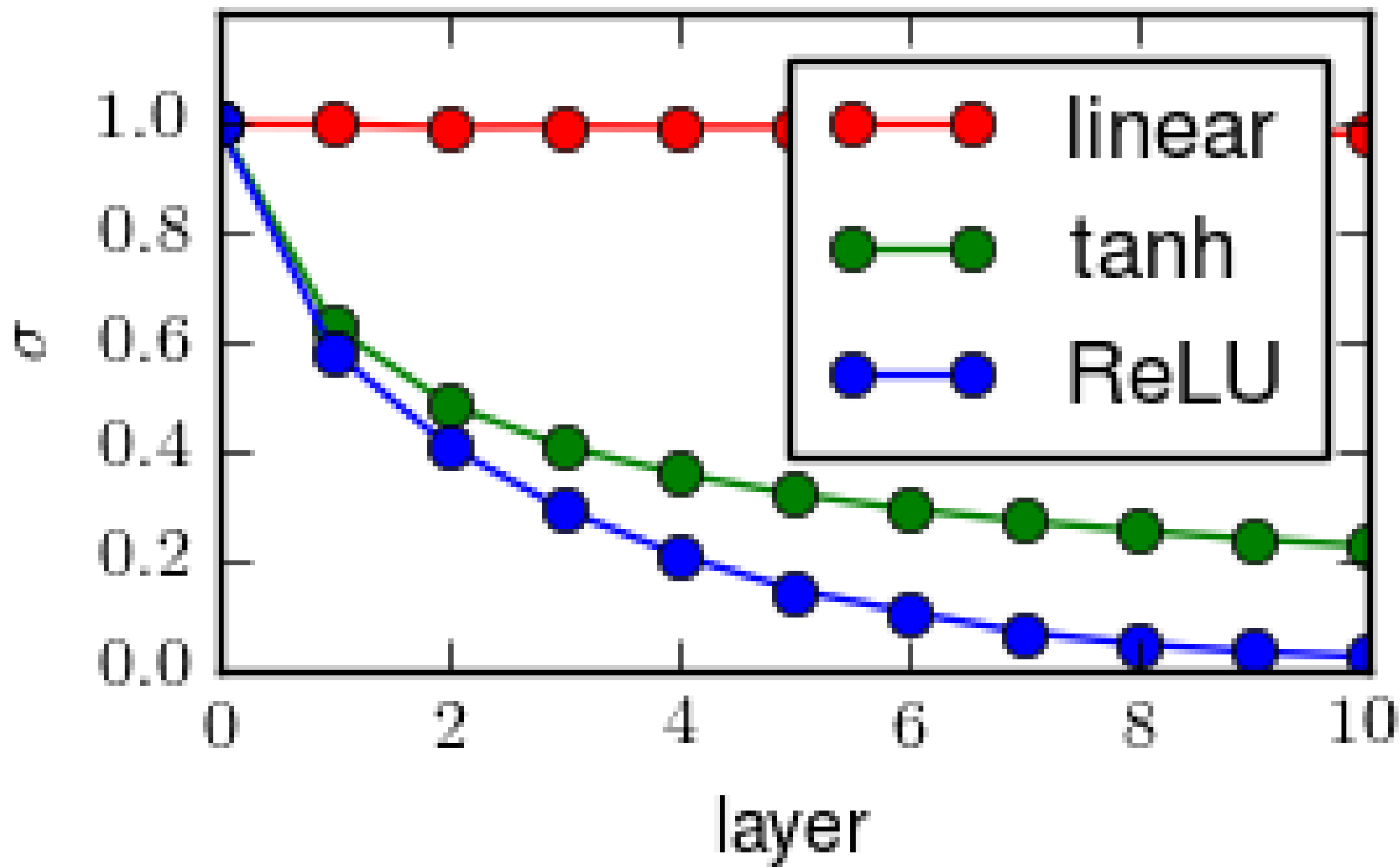


ReLU

$$f(s) = \max(0, s)$$







Sigmoid

$$\sigma(s) = \frac{1}{1 + e^{-s}}$$

