Statistical Machine Learning (BE4M33SSU)
Lecture 6: 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

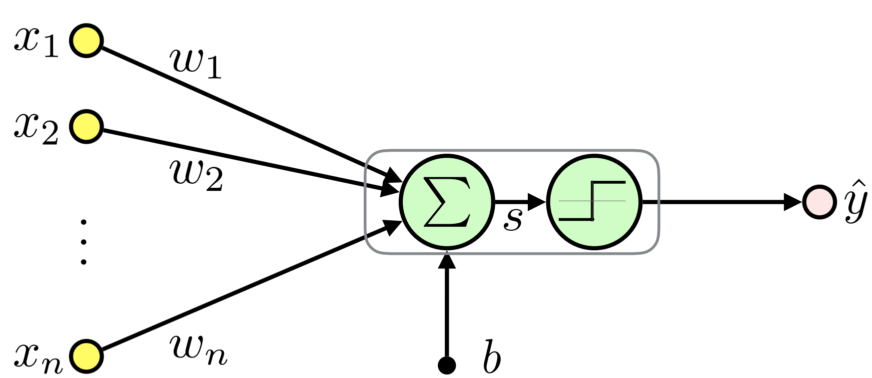
Neural Networks Overview

- Training examples: $\mathcal{T}^m = \{(x_i, y_i) \in (\mathcal{X} \times \mathcal{Y}) \mid i = 1, \ldots, m\}$, 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:

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

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

McCulloch-Pitts Perceptron (1943)

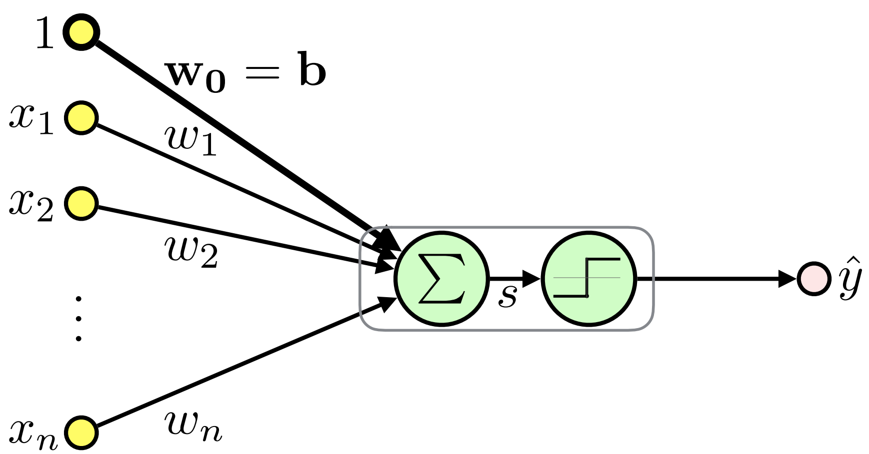


$\boldsymbol{x} = \left(x_1, x_2, \ldots, x_n\right)^T \in \mathbb{R}^n$	input (feature vector)
$\boldsymbol{w} = \left(w_1, w_2, \ldots, w_n\right)^T \in \mathbb{R}^n$	weights
$b \in \mathbb{R}$	bias (threshold)
$s = \langle \boldsymbol{w}, \boldsymbol{x} \rangle + b \in \mathbb{R}$	inner potential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

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

- It is the linear classifier we have already seen.

McCulloch-Pitts Perceptron: Treating Bias

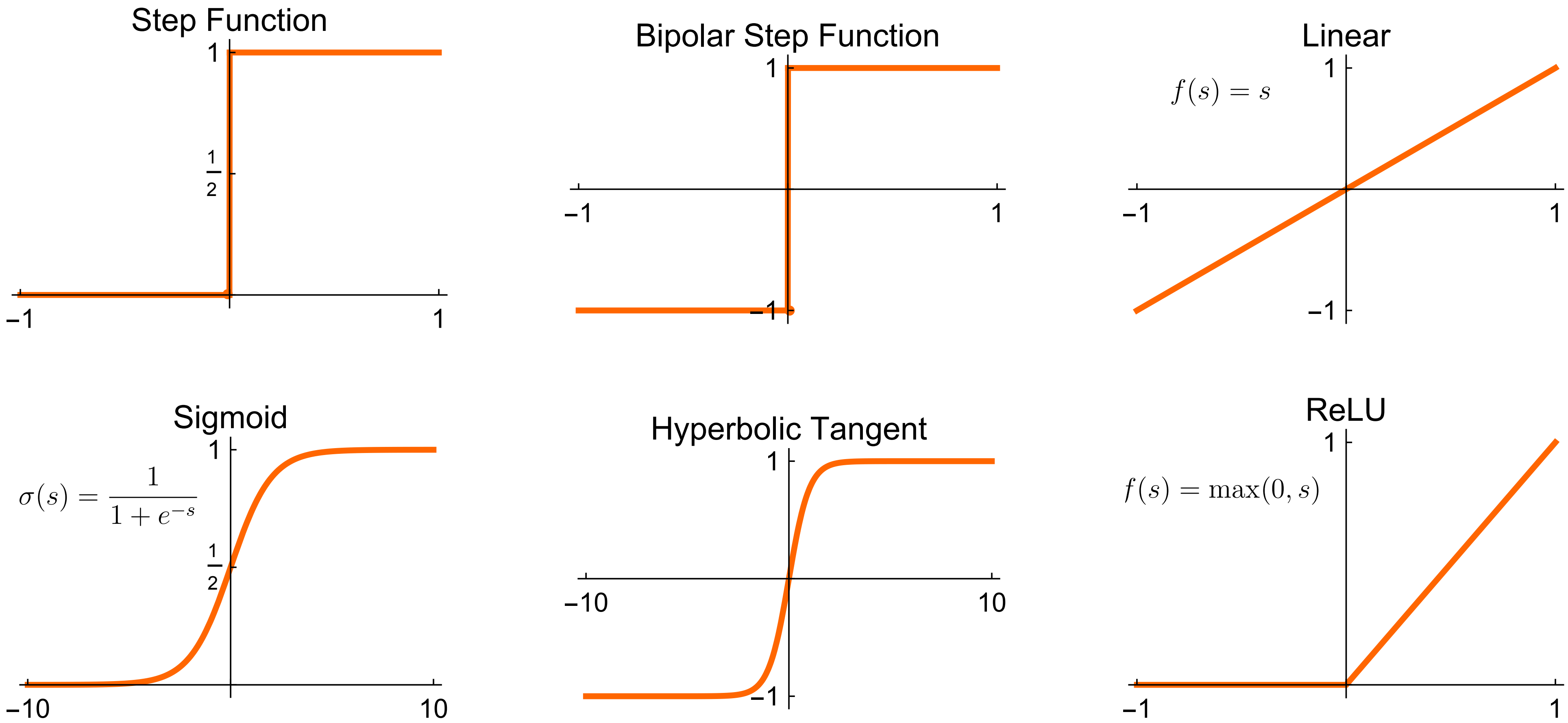


- Treat bias as an extra fixed input $x_0 = 1$ weighted $w_0 = b$:

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

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

Activation Functions



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

Linear Neuron

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

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

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

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

Linear Neuron: Maximum Likelihood Estimation

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

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

- Likelihood for i.i.d. data:

$$p\left(\boldsymbol{y} | \boldsymbol{w}, \mathbf{X}, \sigma\right) = \prod_{i=1}^{m} p\left(y_i | \boldsymbol{w}, \boldsymbol{x_i}, \sigma\right) = \prod_{i=1}^{m} \left(2\pi\sigma^2\right)^{-\frac{1}{2}} e^{-\frac{1}{2\sigma^2}\left(y_i - \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle\right)^2} =$$

$$= \left(2\pi\sigma^2\right)^{-\frac{m}{2}} e^{-\frac{1}{2\sigma^2}\sum_{i=1}^{m}\left(y_i - \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle\right)^2} =$$

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

- Negative Log Likelihood (switching to minimization):

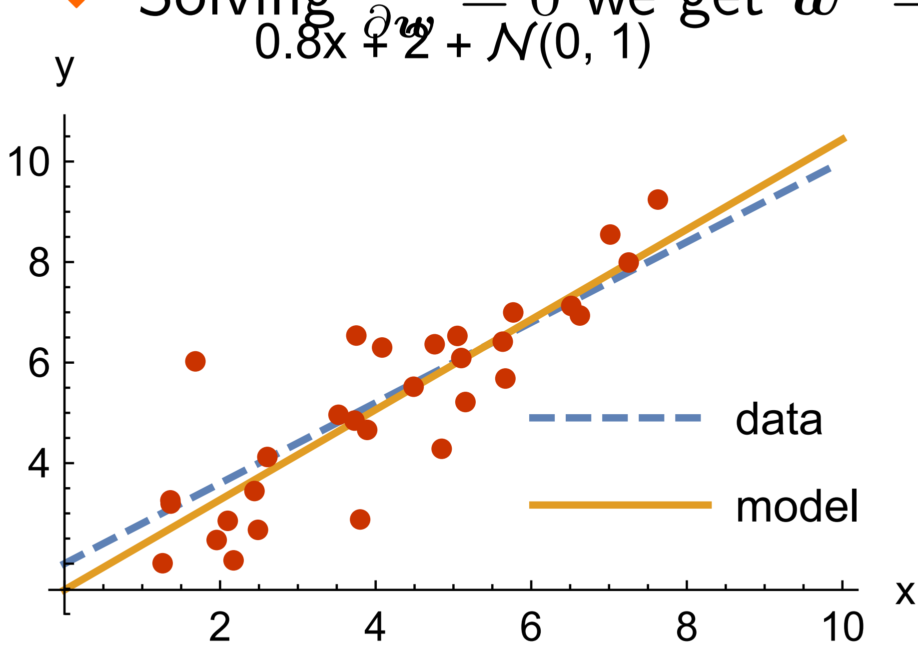
$$\mathcal{L}\left(\boldsymbol{w}\right) = \frac{m}{2}\log\left(2\pi\sigma^2\right) + \frac{1}{2\sigma^2}\left(\boldsymbol{y} - \mathbf{X}\boldsymbol{w}\right)^T\left(\boldsymbol{y} - \mathbf{X}\boldsymbol{w}\right)$$

- Note that

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

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

- Minimization of $\mathcal{L}(\boldsymbol{w}) \equiv$ least squares estimation
- 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)



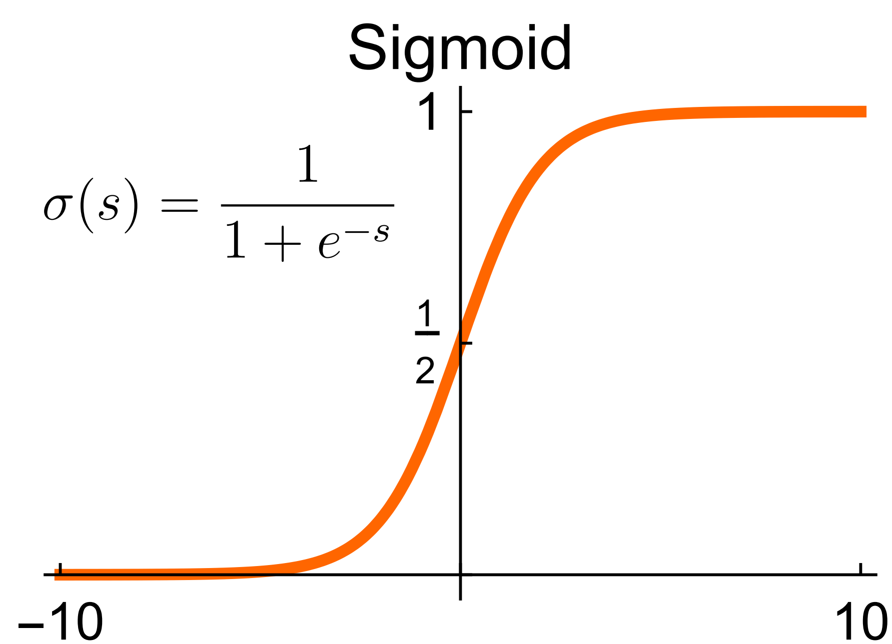
Logistic Sigmoid and Probability

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

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

- 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**:

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

- 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: $\boldsymbol{y} = \left(y_1, \dots, y_m\right)^T, \, y_i \in \{0, 1\}$
- Output class: $\hat{\boldsymbol{y}} = \left(\hat{y}_1, \dots, \hat{y}_m\right)^T, \, \hat{y}_i \in (0, 1)$
- 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:

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

- Negative Log Likelihood:

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

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

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

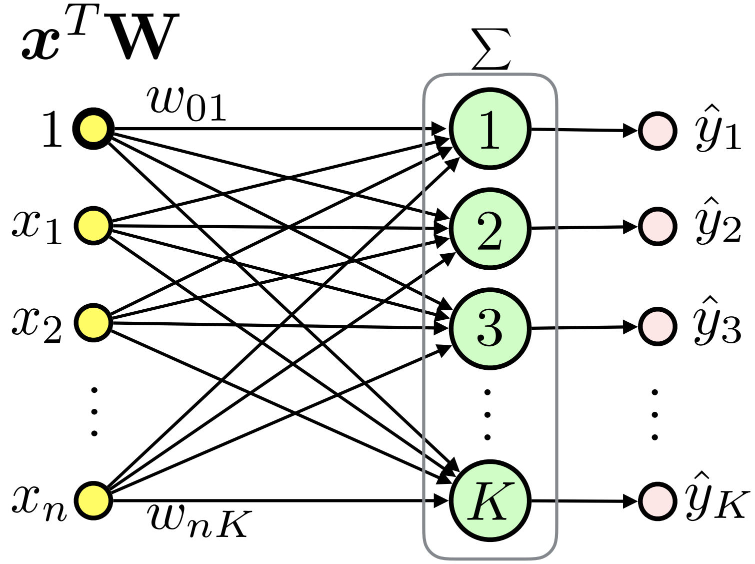
- Gradient w.r.t. logistic regression parameters:

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

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

Linear Layer

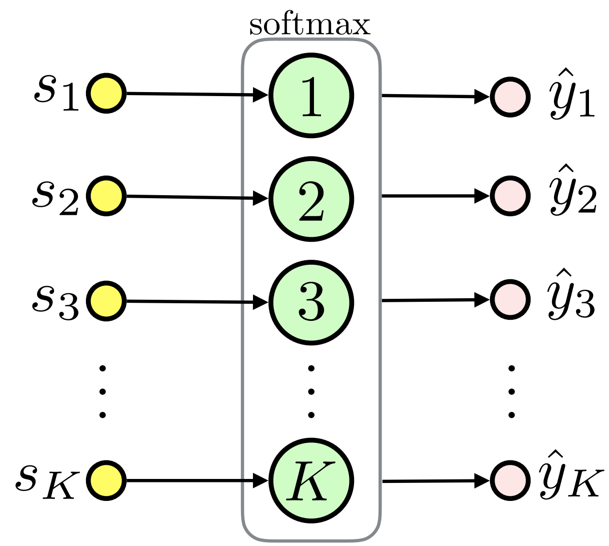
- Multiple linear neurons: linear (dense, fully-connected, affine) layer
- Output k: $\hat{y}_k = \langle \boldsymbol{x}, \boldsymbol{w}_k \rangle$, $k = 1, 2, \ldots, K$
- All outputs using *weight matrix* $\mathbf{W}$: $\hat{\boldsymbol{y}} = \boldsymbol{x}^T \mathbf{W}$
- Multiple samples: $\hat{\mathbf{Y}}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



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

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

Softmax Layer



- Multinominal classification, K mutually exclusive classes
- Definition: $\sigma_k(\boldsymbol{s}) \triangleq \dfrac{e^{s_k}}{\sum_{c=1}^{K} e^{s_c}}$, 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: $p(y=k|\boldsymbol{s}) = \sigma_k(\boldsymbol{s})$
- Softmax input (vector $\boldsymbol{s}$) is often called called the *logits*

Softmax Layer MLE

- 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 $y_{ik} \triangleq [\![y_i = k]\!]$
- Likelihood:

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

- Negative Log Likelihood:

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

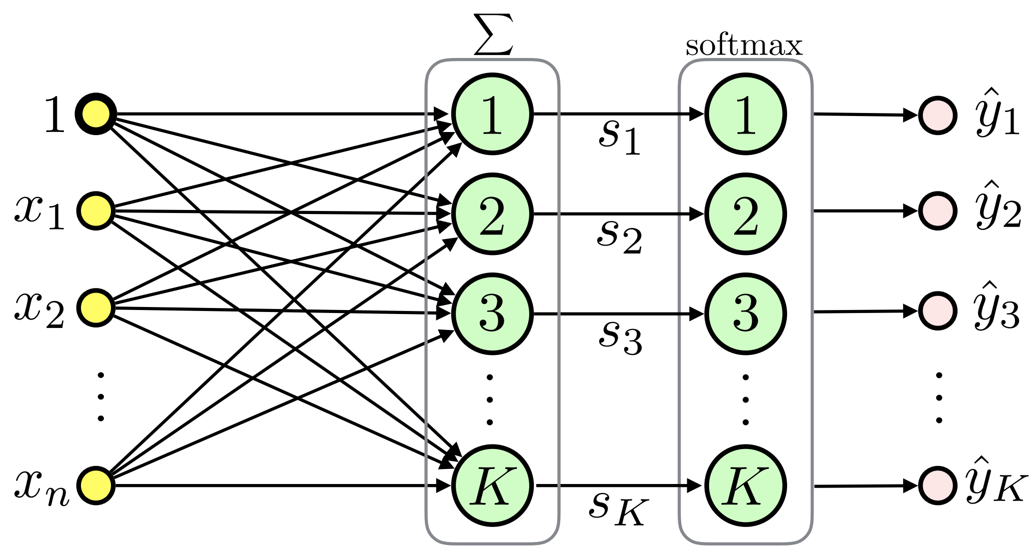
  Again the **cross-entropy**

- See seminar for the gradient

Multinominal Logistic Regression

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

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

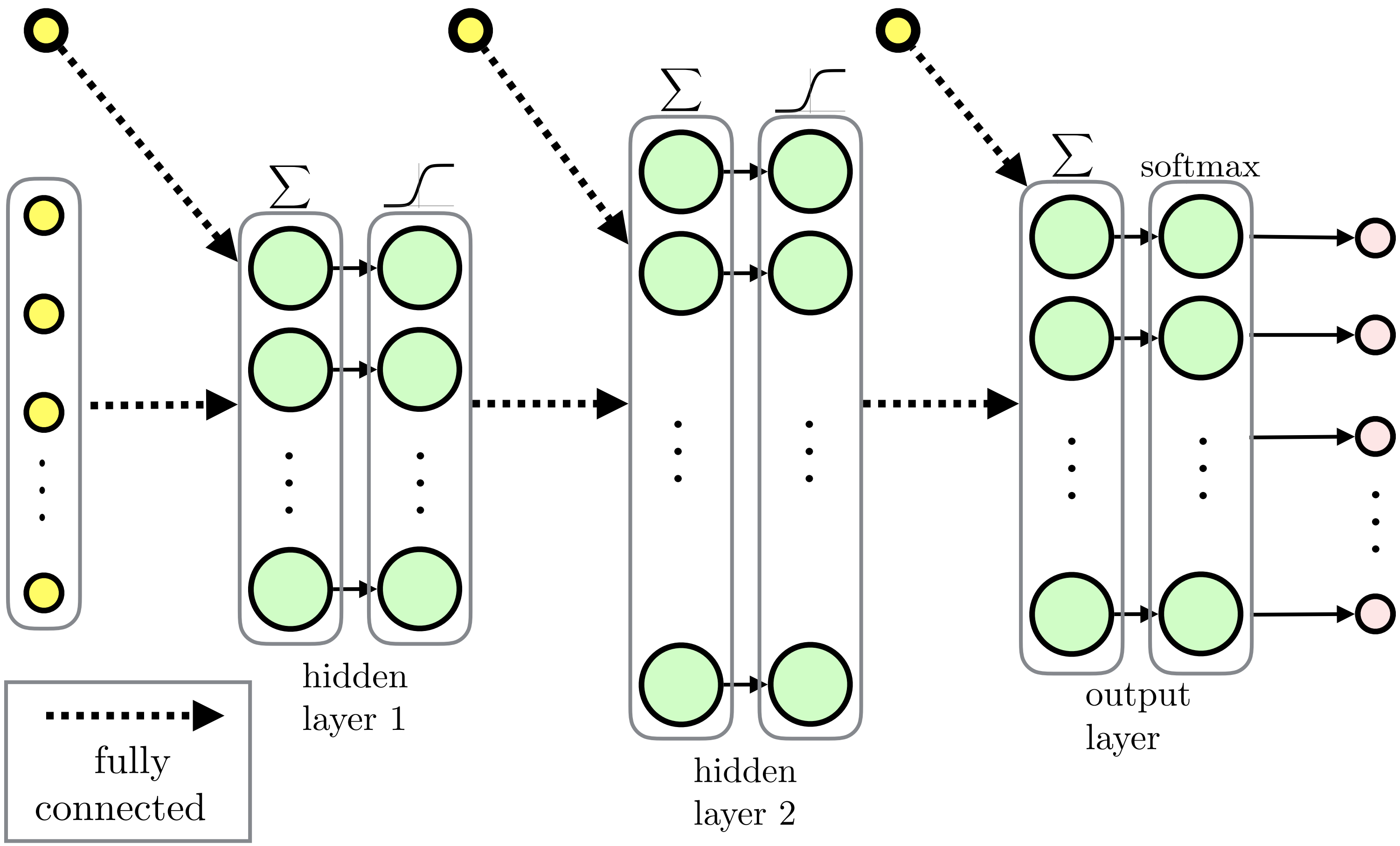


- Classifier: $h(\boldsymbol{x}, \mathbf{W}) = \operatorname*{argmax}_k \hat{y}_k$

Loss Functions: Summary

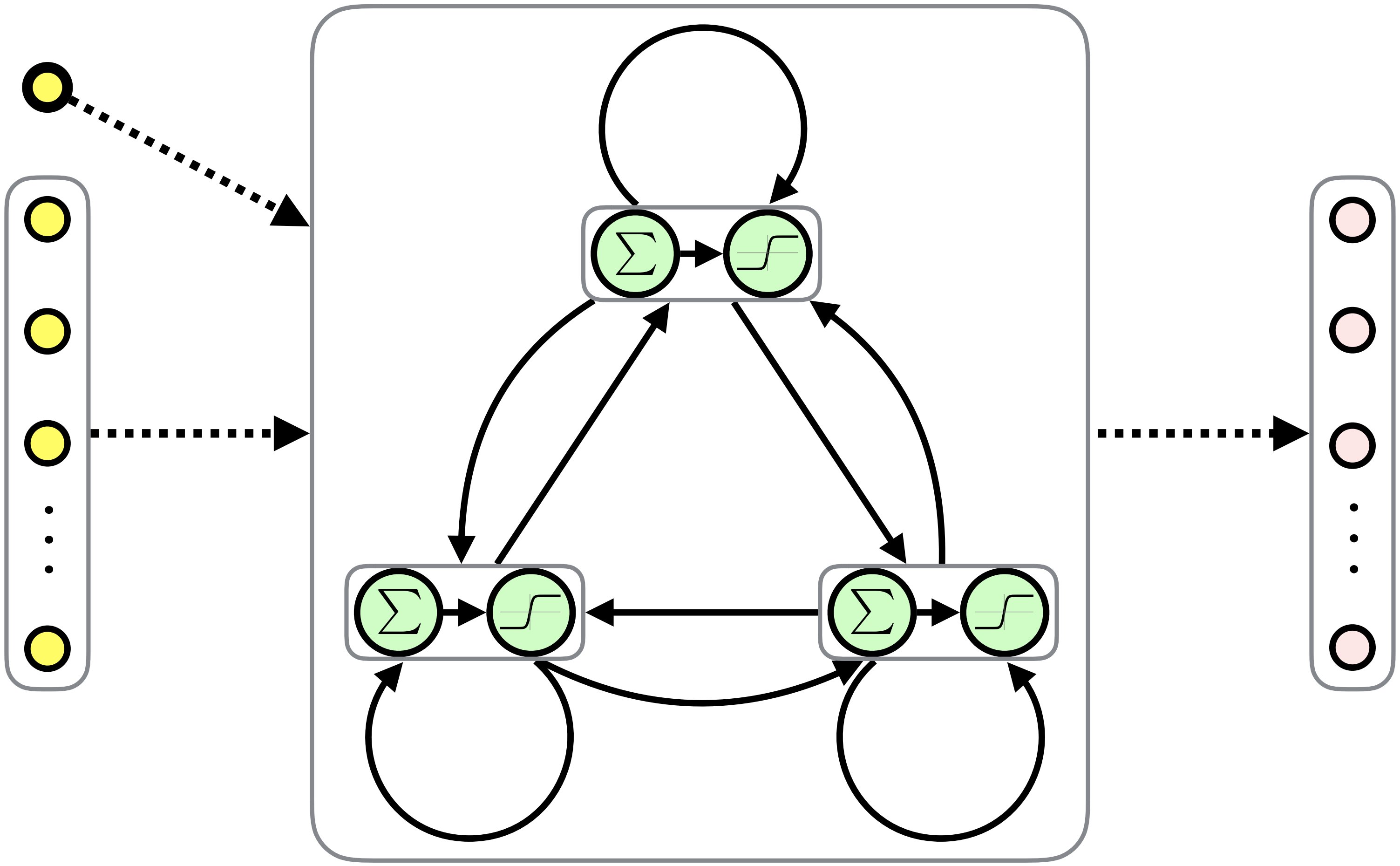
problem	**output**	**suggested loss function**
binary classification	sigmoid neuron	cross-entropy $-\frac{1}{m}\sum_{i=1}^{m}\left[y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)\right]$
multinominal classification	softmax	multinominal cross-entropy $-\frac{1}{m}\sum_{i=1}^{m}\sum_{c=1}^{K} y_{ic}\log(\hat{y}_{ic})$
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)



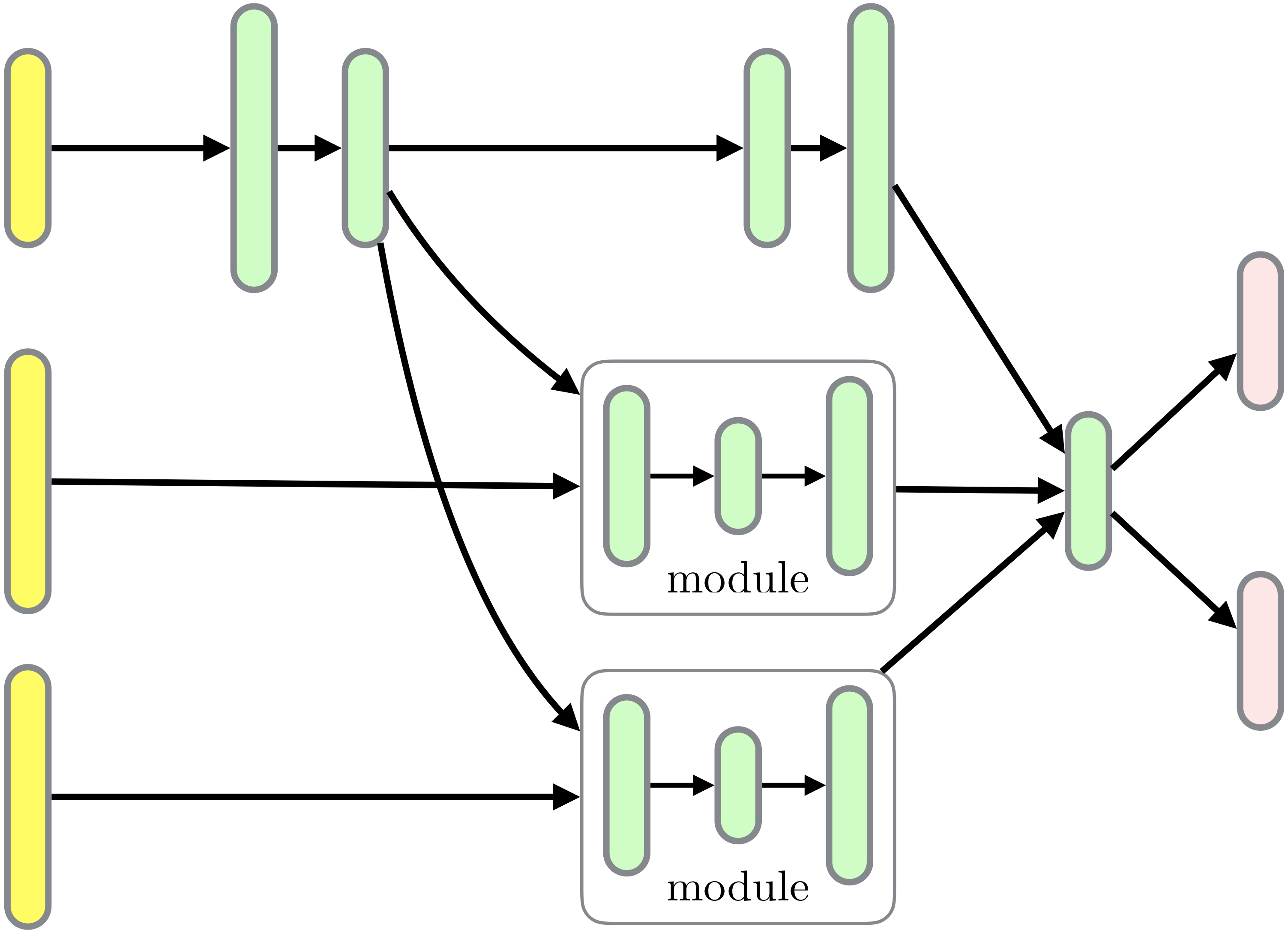
- Feed-forward ANN
- Fully-connected layers
- MLP for regression would typically use linear output layer

Recurrent Neural Network (RNN)



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

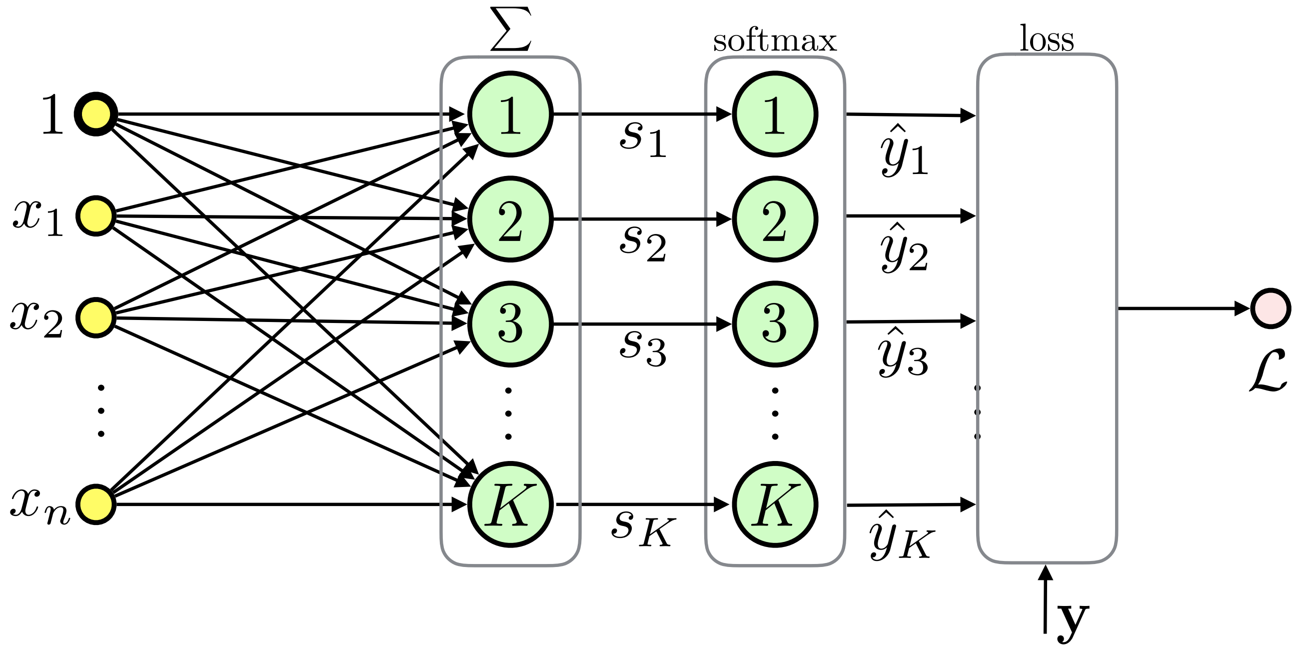
Modular and Hierarchical Architectures



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

Backpropagation Overview

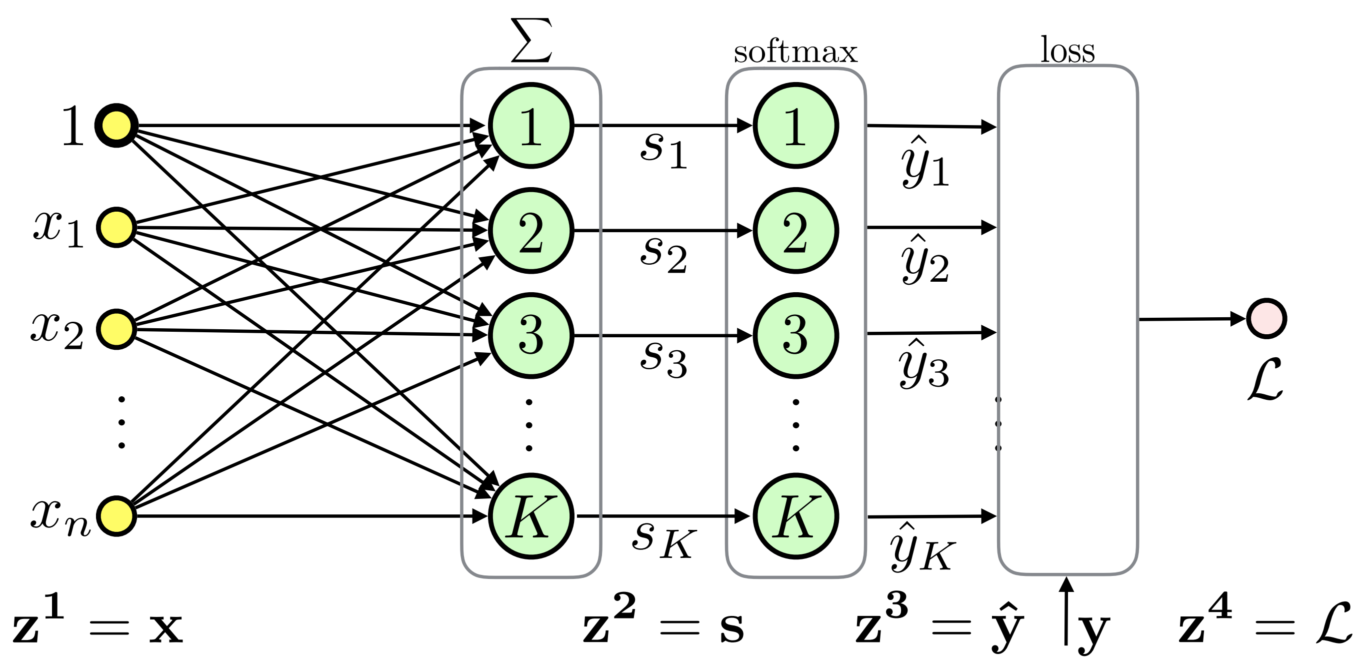
- A method to compute a gradient of the *loss function* with respect to its parameters: $\nabla\mathcal{L}(\boldsymbol{w})$
- $\nabla\mathcal{L}(\boldsymbol{w})$ is in turn used by optimization methods like gradient descent
- Here, we present the "modular" backpropagation (see Nando de Freitas' Machine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



Backpropagation: the Loss Function

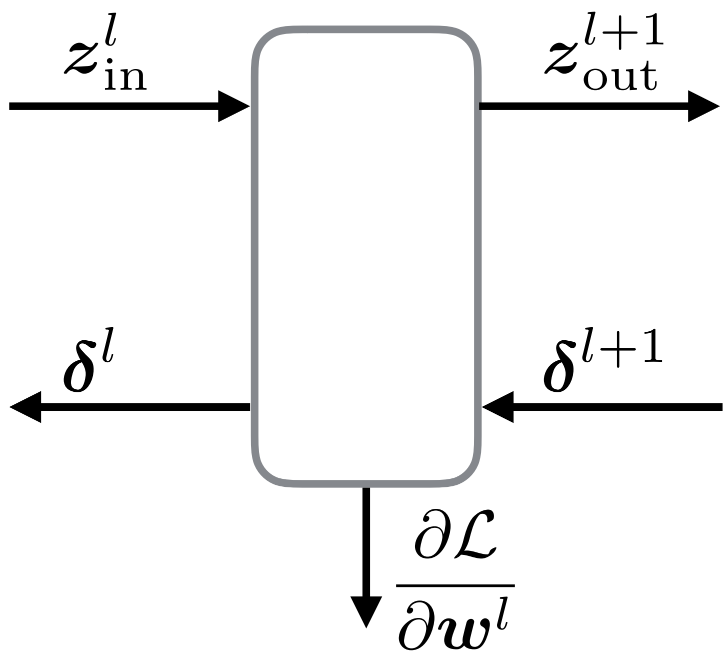
- The loss function is the multinominal cross-entropy in this case:

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



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 (that can be later combined into complex networks)
- Represent even the loss function as a module
- Passing messages

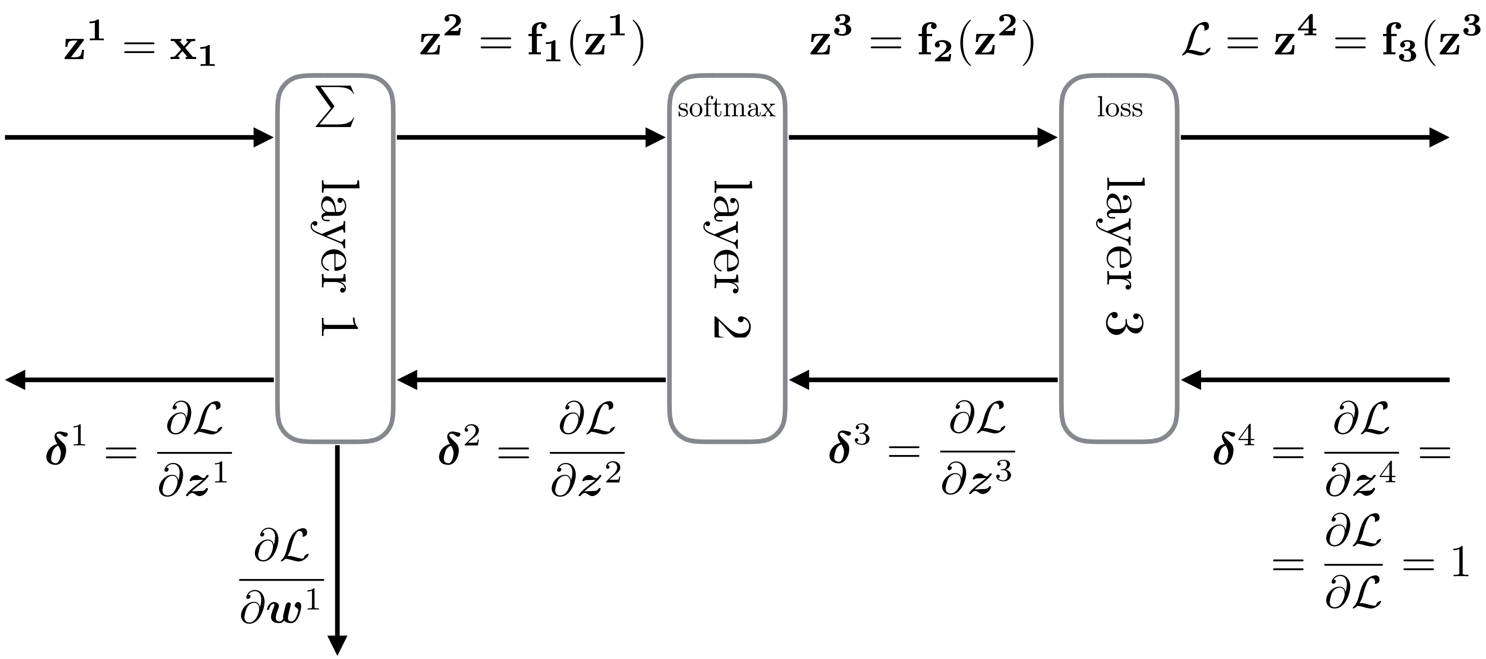


Backpropagation: Backward Pass Message

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

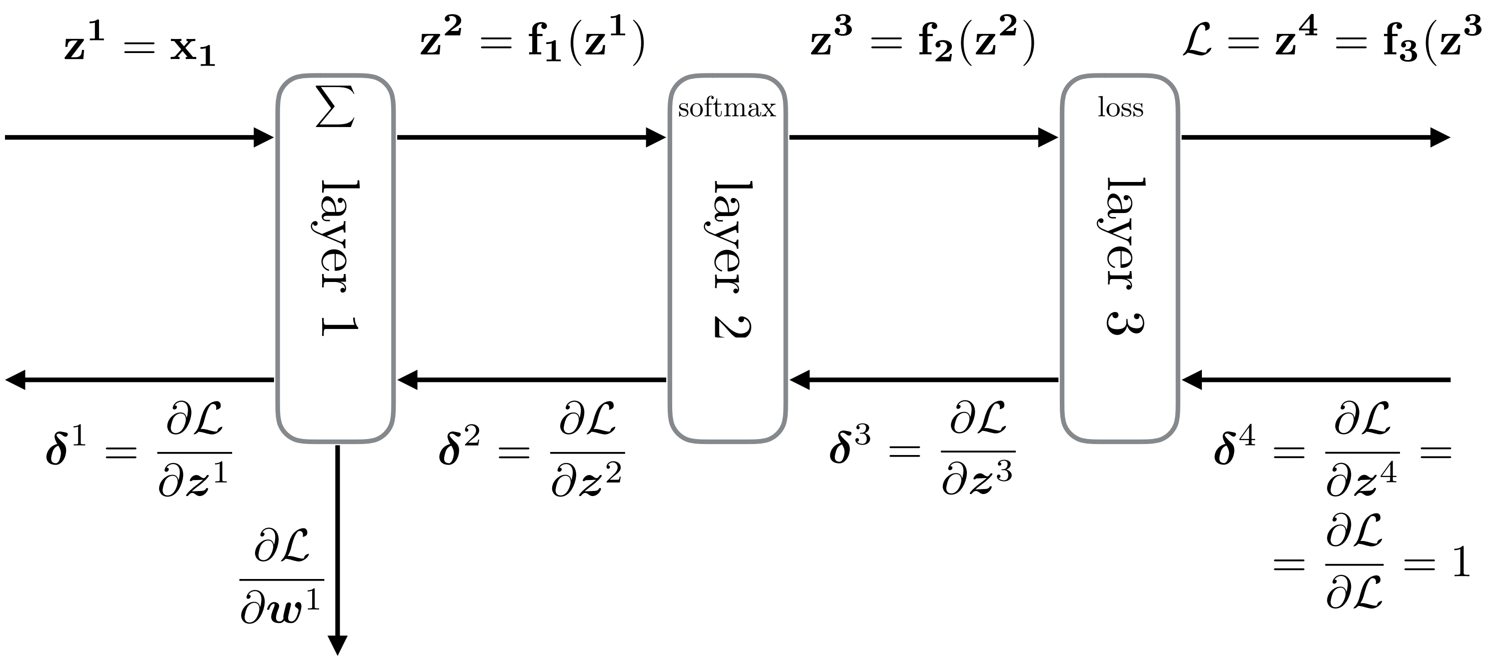


Backpropagation: Parameters

- 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



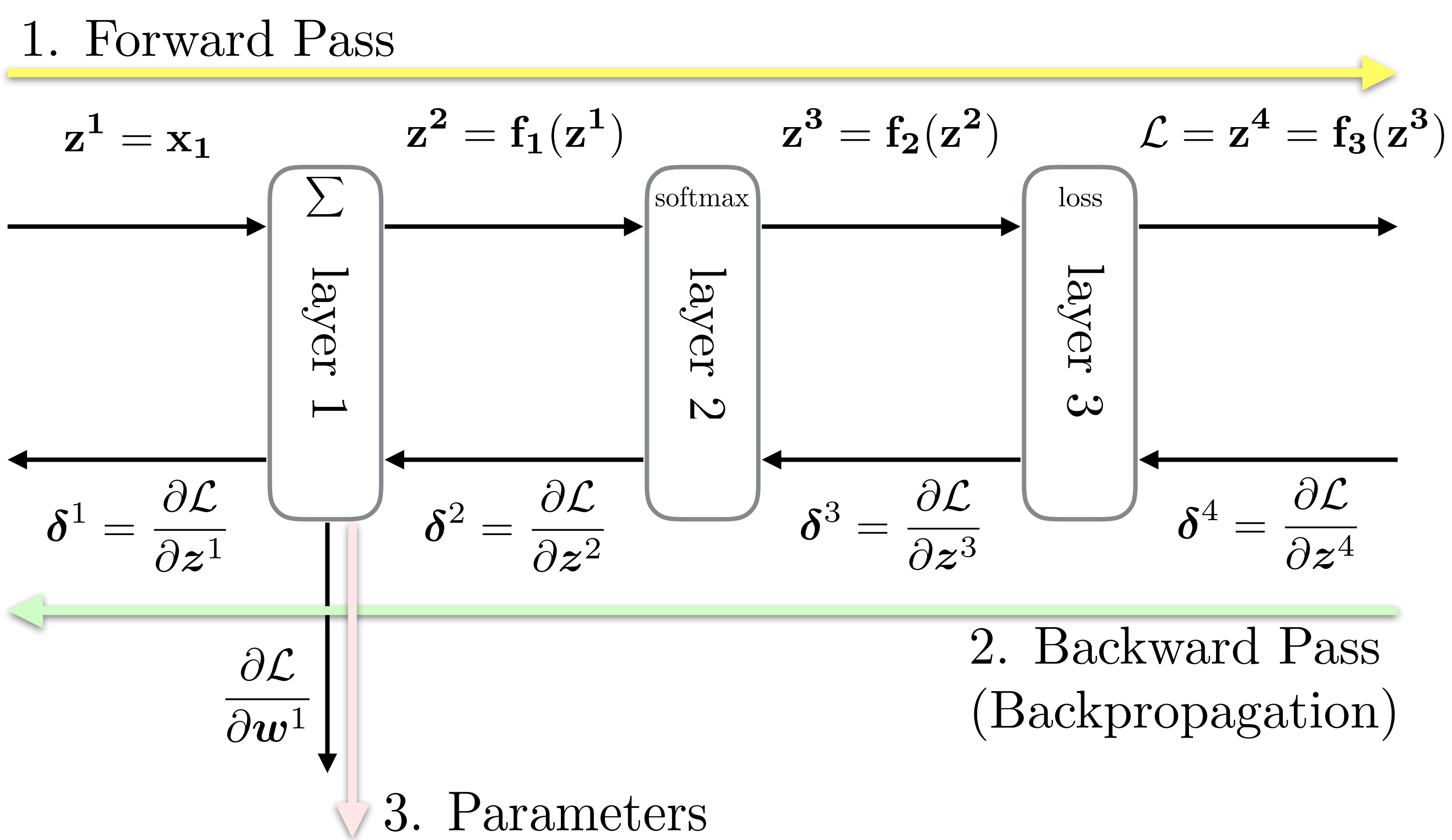
Backpropagation: Steps

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

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

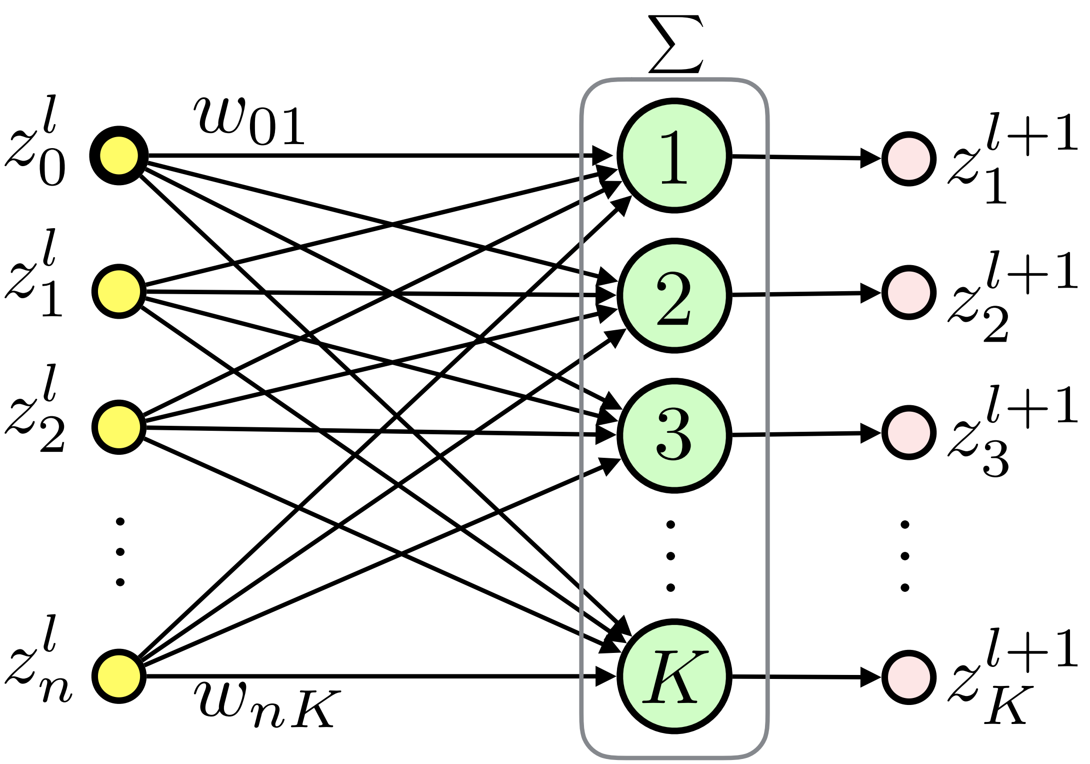
backward: $\dfrac{\partial \boldsymbol{z}^{l+1}}{\partial \boldsymbol{z}^l}$

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



Example: Linear Layer

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



Example: Squared Error

- **forward**: $z^{l+1} = \left(y - z^l\right)^2$
- **backward**: $\dfrac{\partial z^{l+1}}{\partial z^l} = 2(z^l - y), \quad i \in \{1, \ldots, n\}$

Gradient Descent (GD)

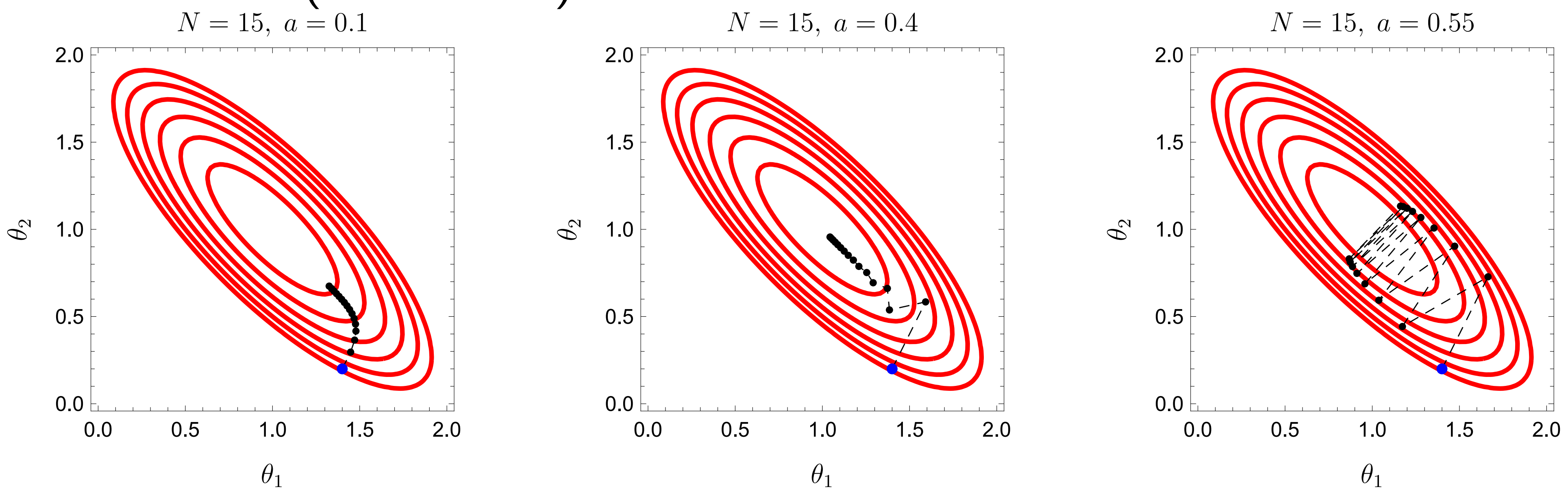
- **Task**: find parameters which minimize loss over the training dataset:

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

where $\boldsymbol{\theta}$ is a set of all parameters defining the ANN

- Gradient descent: $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha_k \nabla \mathcal{L}(\boldsymbol{\theta}_k)$
where $\alpha_k > 0$ is the **learning rate** or **stepsize** at iteration k

- More about (Stochastic) Gradient Descent the next time . . .



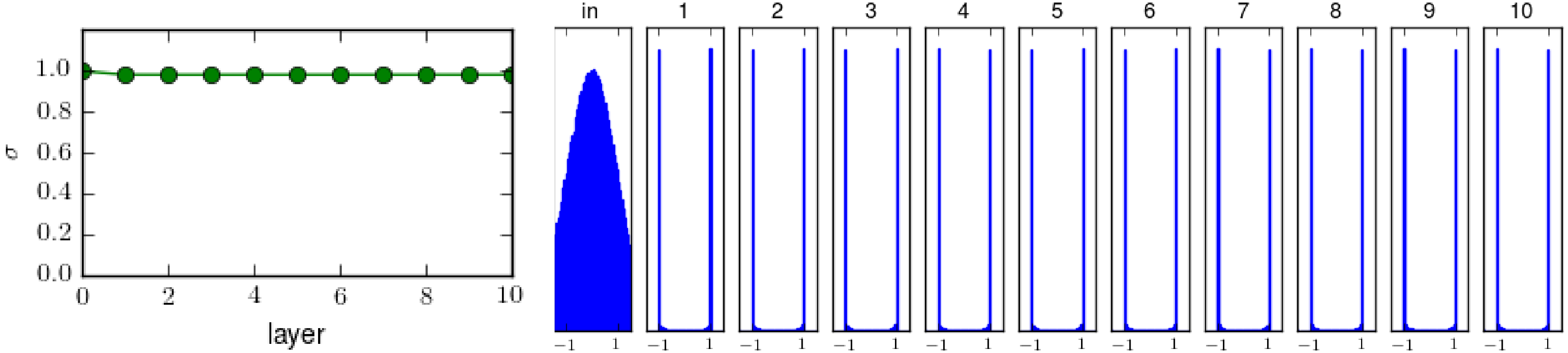
Parameter Initialization

- Is it a good idea to initially set all weights to a constant?
- **No.** All neurons would behave the same: the same δs 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 might get into trouble

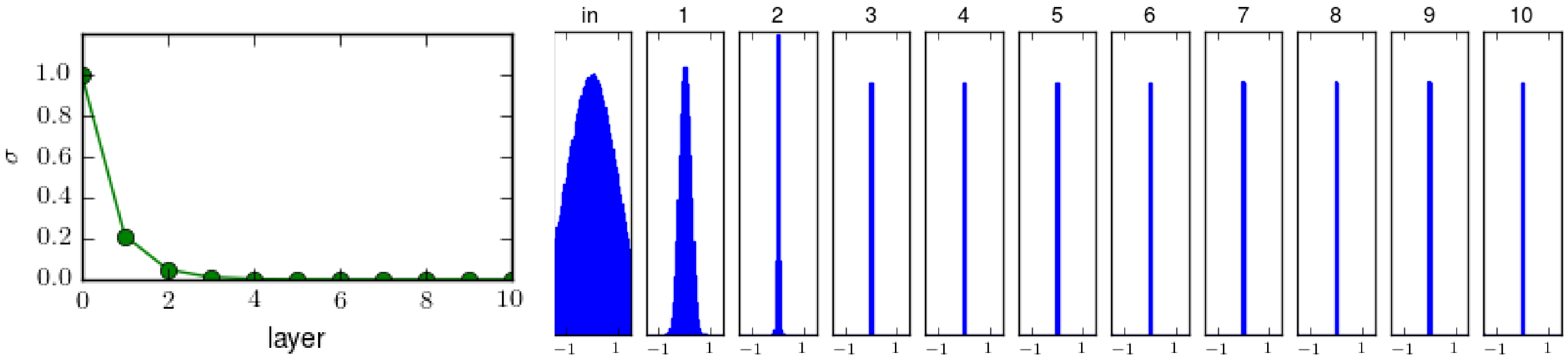
Gaussian Initialization Example

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

$\sigma = 1$

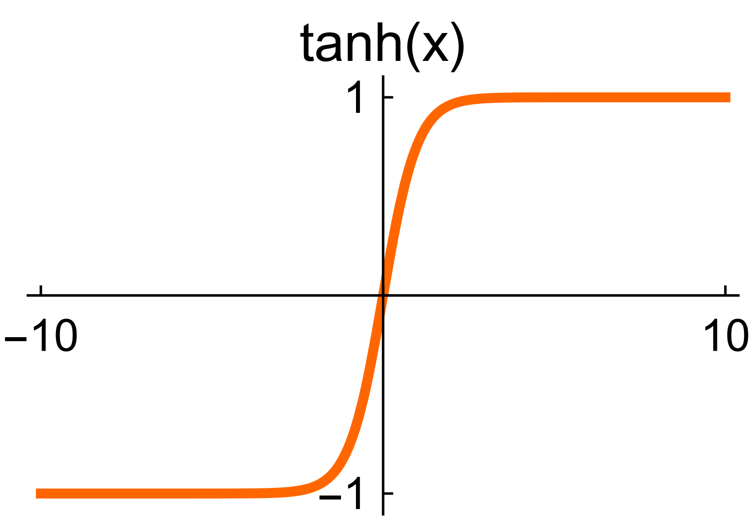


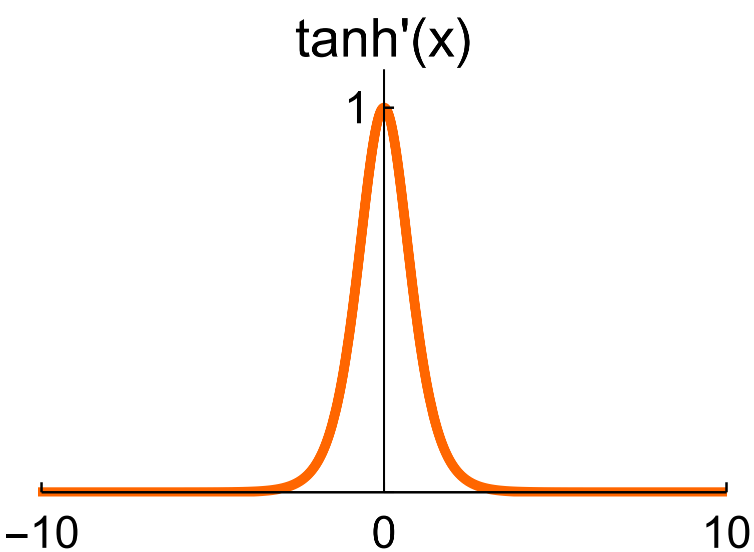
$\sigma = 0.01$



Vanishing Gradient

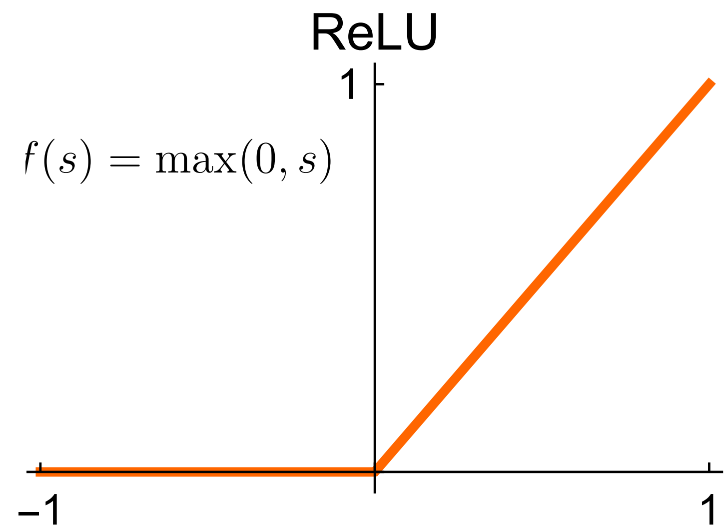
- For large σ (saturation) the $\tanh$ derivative is almost zero
- For small σ (output close to zero):
  - the derivative is at most 1,
  - the weights are very small and $\frac{\partial z_j^{l+1}}{\partial z_i^l} = w_{ij}$ holds for the preceding linear layer
- In both cases: $\delta \to 0$ as the number 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 $\rightarrow$ dead units $\rightarrow$ Leaky ReLU
- Unbounded: use regularization to prevent numerical problems



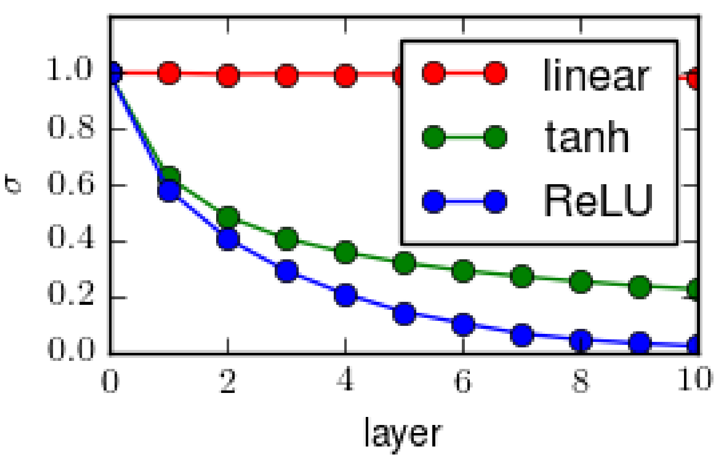
Xavier Initialization

- 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., $\mathbb{E}(x_i) = \mathbb{E}(w_i) = 0$:

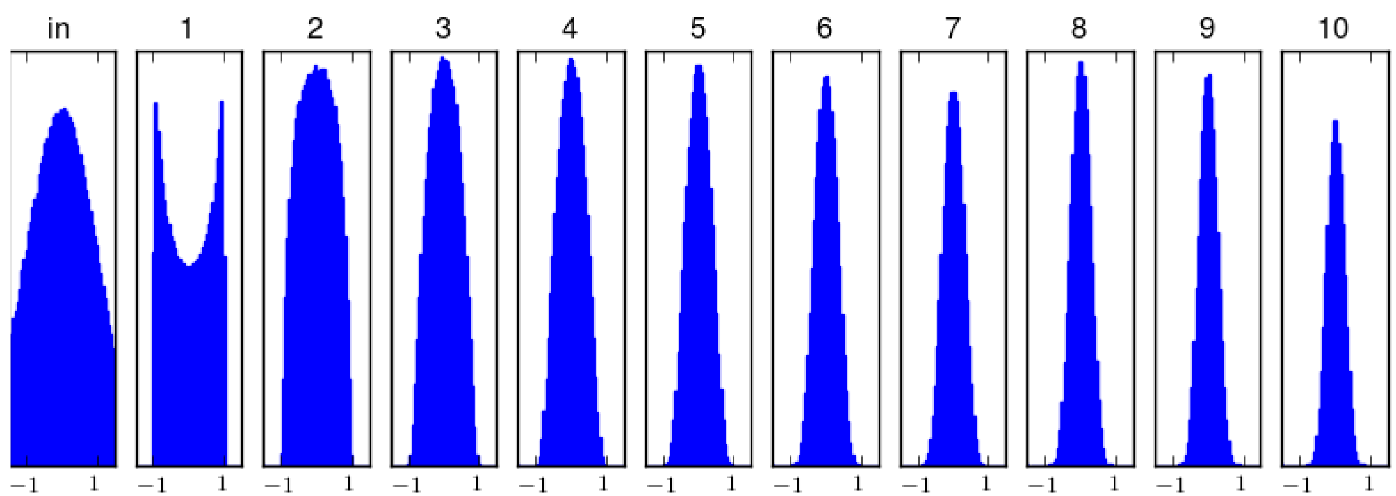
$$
\begin{aligned}
\mathrm{Var}(s) &= \mathrm{Var}\left(\sum_i w_i x_i\right) = \sum_i \mathrm{Var}(w_i x_i) = \\
&= \sum_i \mathbb{E}\left([w_i x_i - \mathbb{E}(w_i x_i)]^2\right) = \sum_i \mathbb{E}\left([w_i x_i - \mathbb{E}(w_i)\mathbb{E}(x_i)]^2\right) = \\
&= \sum_i \mathbb{E}(w_i^2 x_i^2) = \sum_i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

Xavier Initialization (contd.)

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



$\tanh$

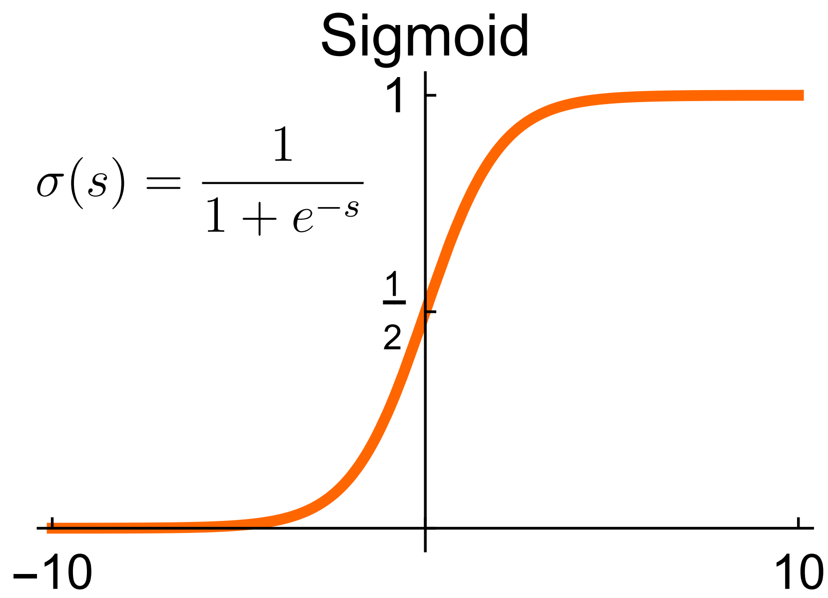


Regularization

- 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 $\boldsymbol{w}^T\boldsymbol{w} = \|\boldsymbol{w}\|_2^2$
- 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:

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

- 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:

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

where

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

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

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

- 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 Improves Numerical Stability

- 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 the solution is the minimizer of our *regularized* loss function:

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

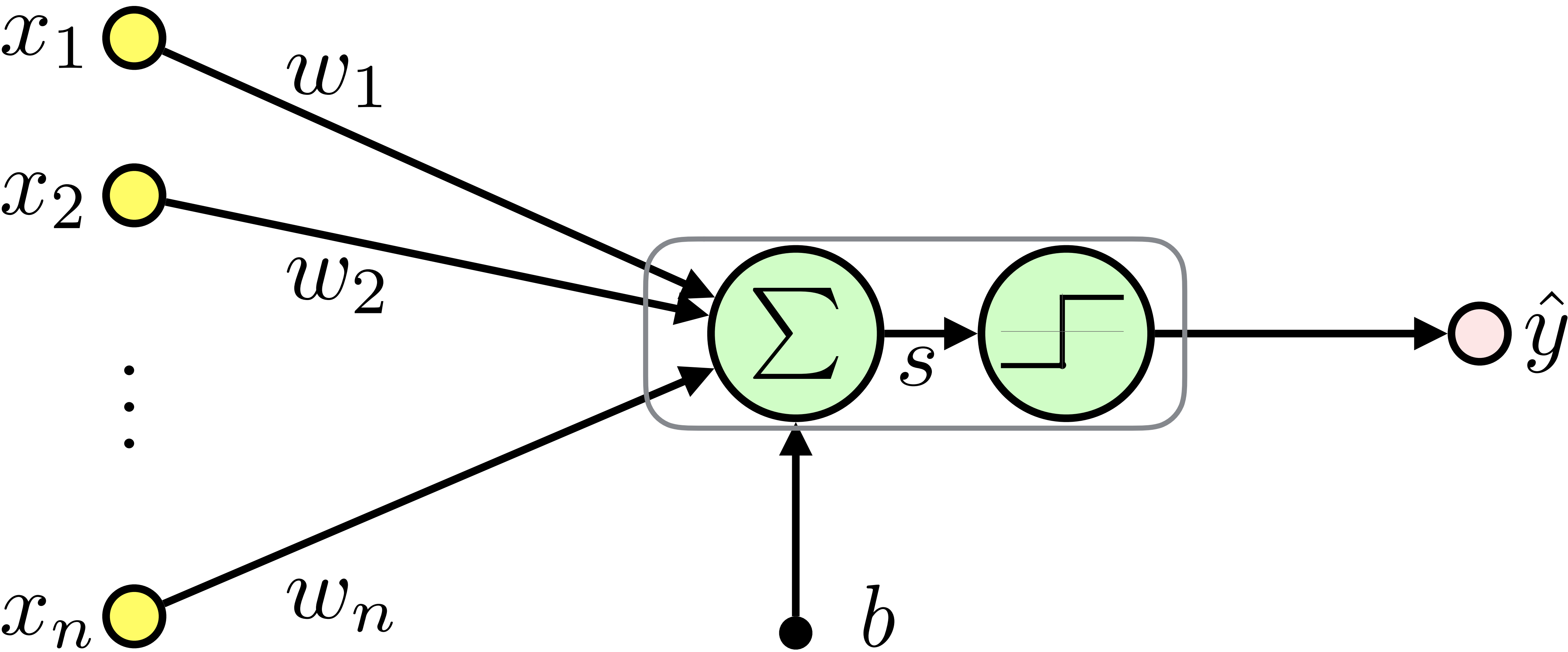
see seminar for the derivation

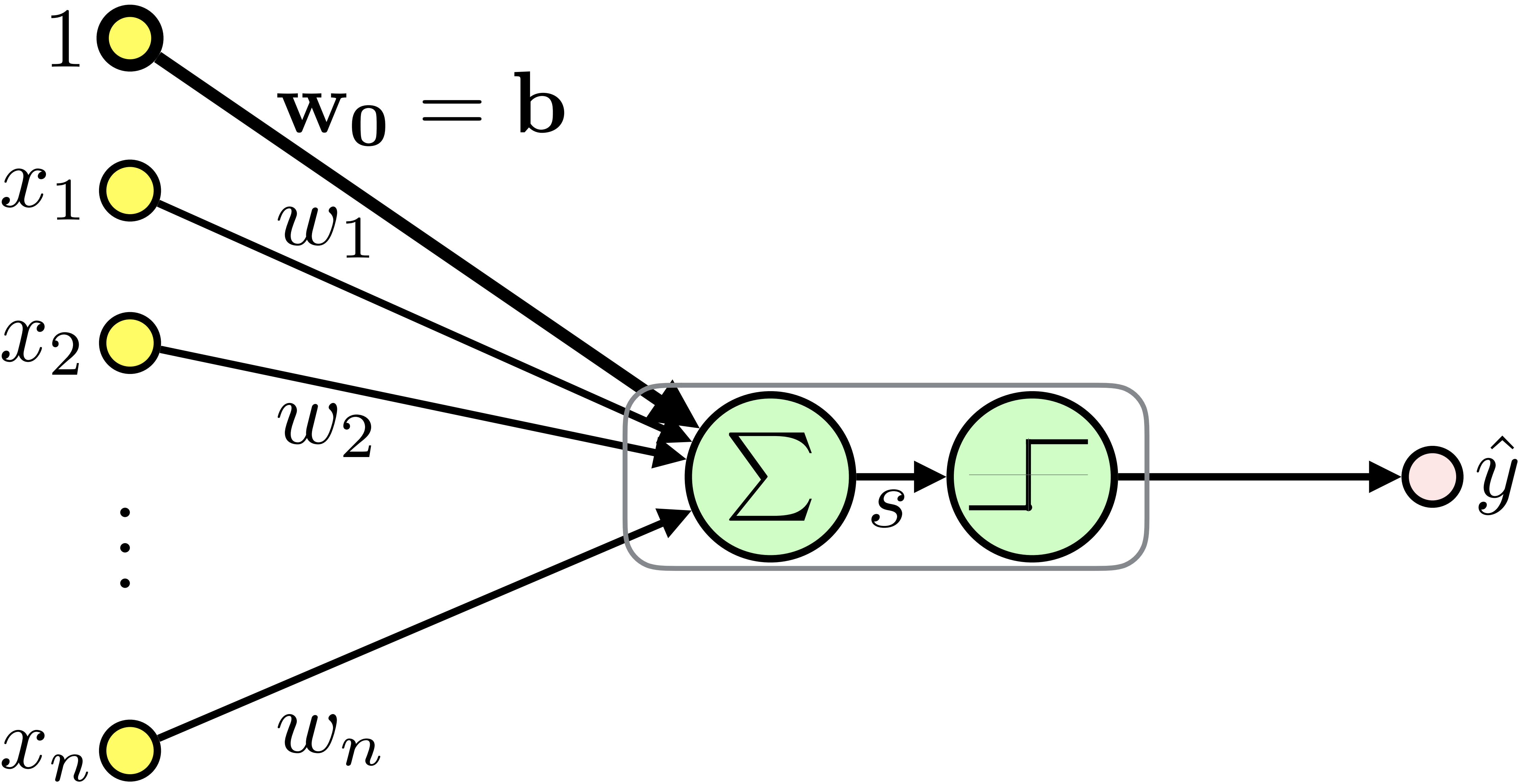
Other Regularization Related Approaches

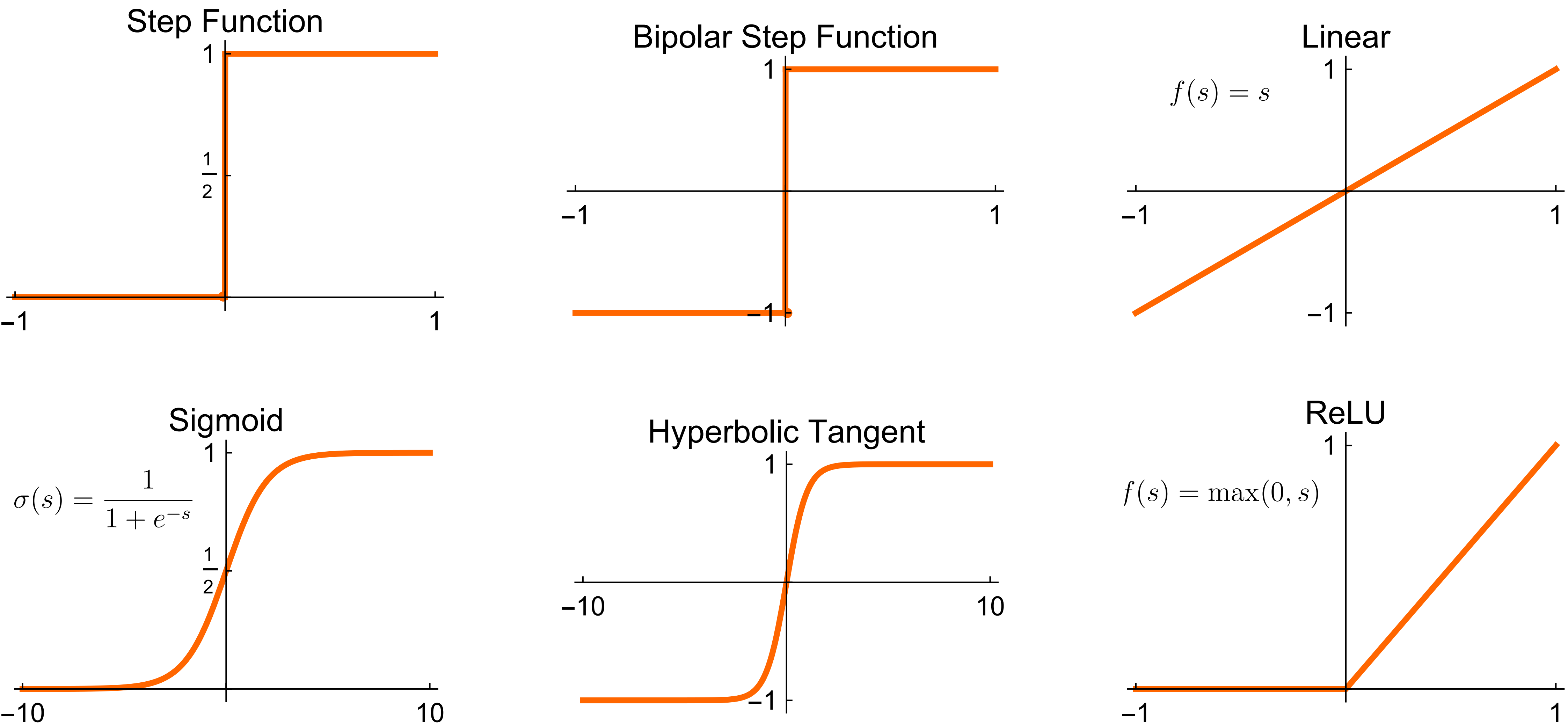
- L1 regularization: sum absolute values, i.e., use $\lambda \|\boldsymbol{w}\|_1$
- Randomize inputs: same as the weight decay for linear neurons
- Dataset augmentation
- Early stopping: start with small weights, stop when validation loss starts to grow, often used for limited time-budget
- Weight sharing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

Next Lecture

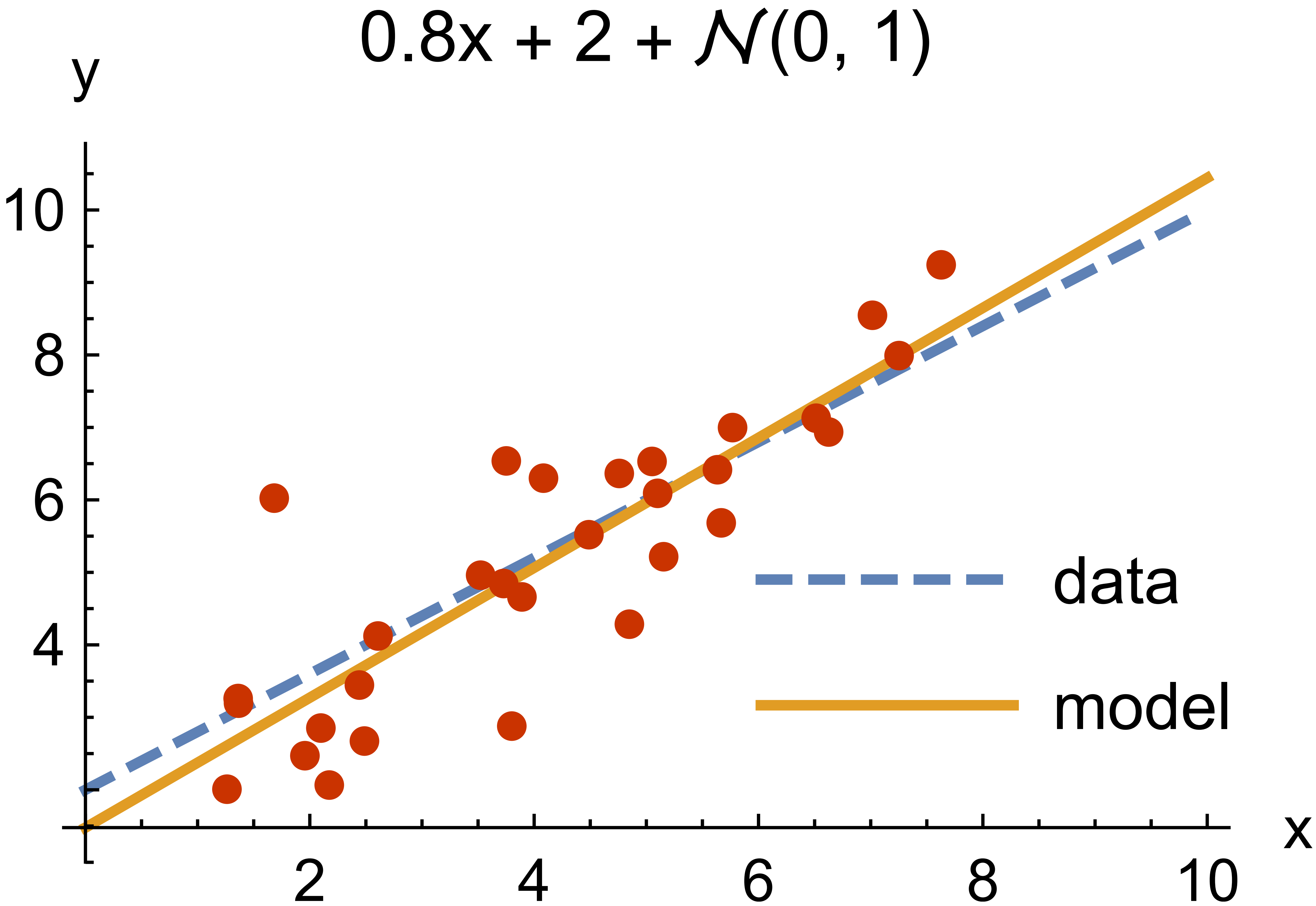
- Stochastic Gradient Descent
- Deep Neural Networks
- Convolutional Neural Networks
- Transfer learning

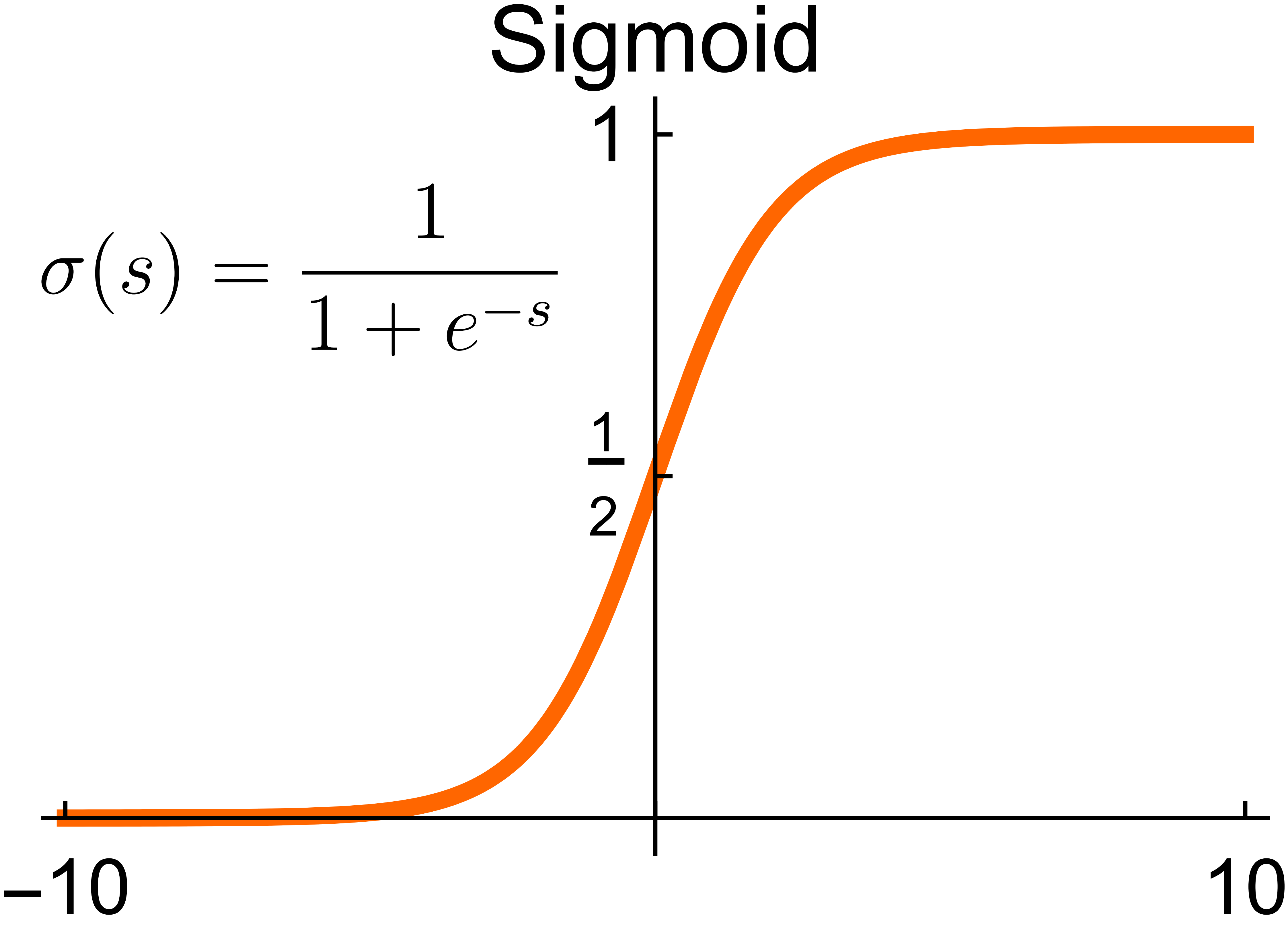

x_1
x_2
x_n
w_1
w_2
w_n
Σ
s
b
$\hat{y}$

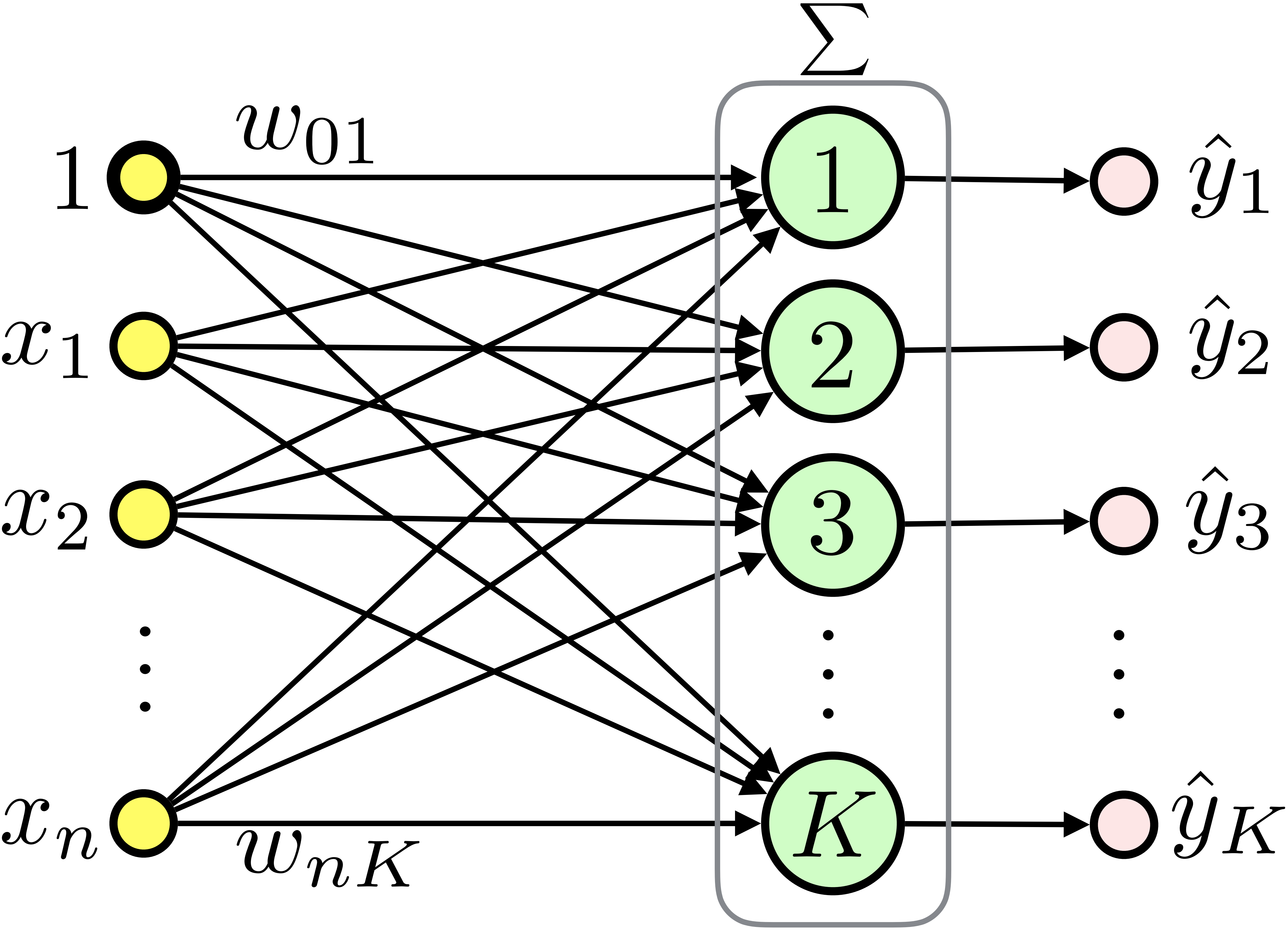

1
$\mathbf{w_0 = b}$
x_1
w_1
x_2
w_2
x_n
w_n
Σ
s
$\hat{y}$

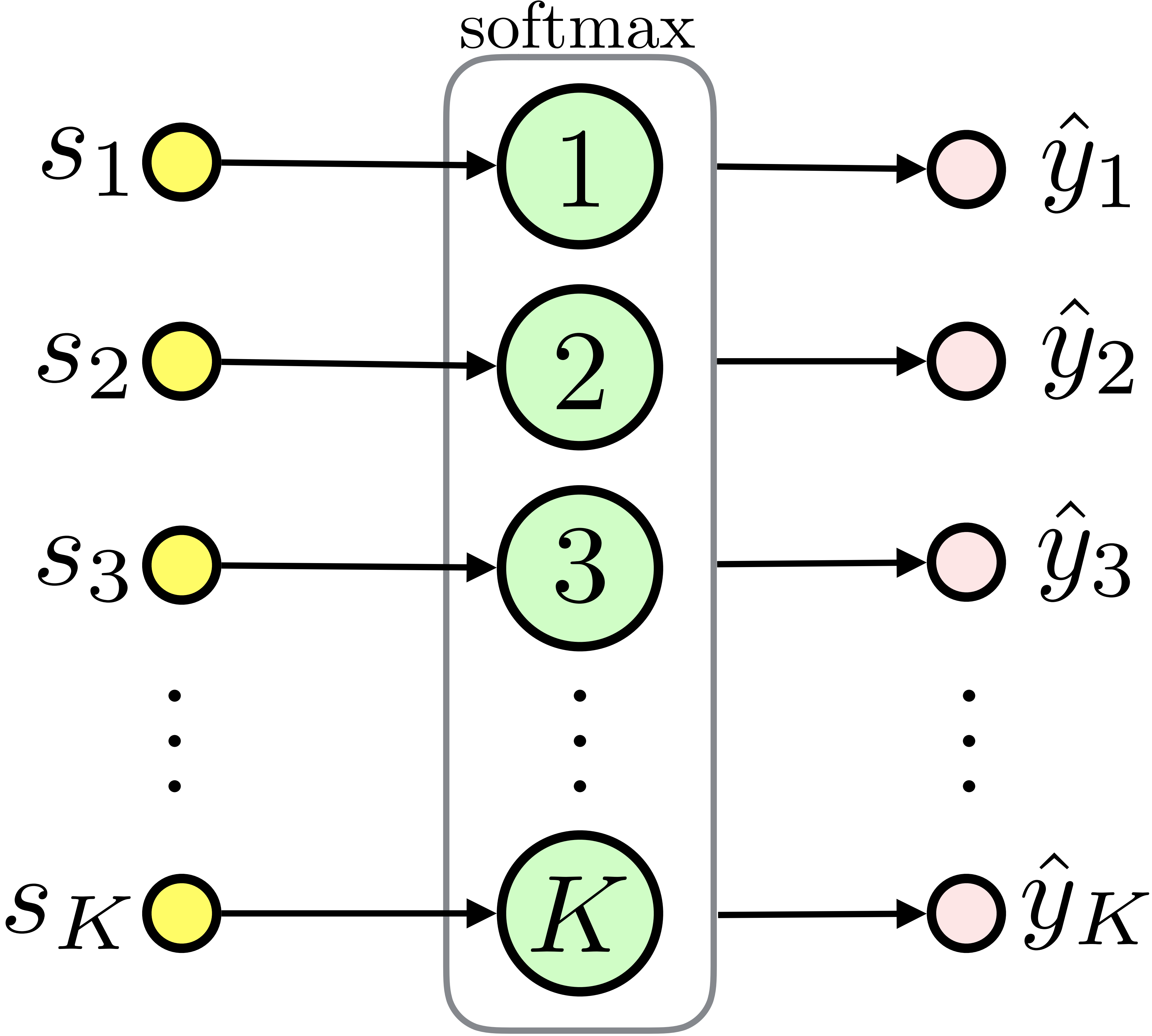


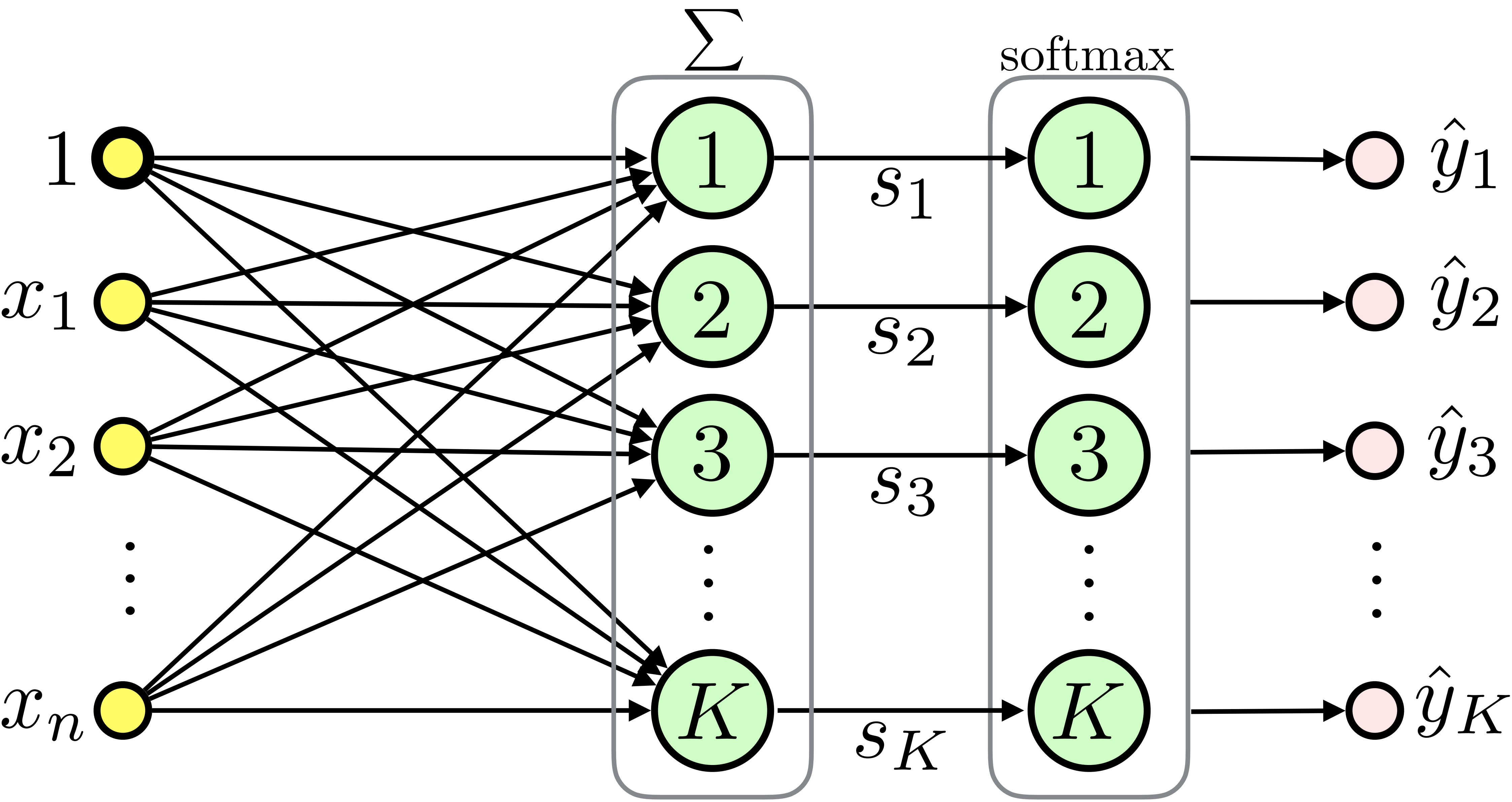
Step Function
1
$\frac{1}{2}$
–1
1
Bipolar Step Function
1
–1
1
–1
Linear
1
$f(s) = s$
–1
1
–1
Sigmoid
1
$\sigma(s) = \frac{1}{1+e^{-s}}$
$\frac{1}{2}$
–10
10
Hyperbolic Tangent
1
–10
10
–1
ReLU
1
$f(s) = \max(0, s)$
–1
1


$0.8x + 2 + \mathcal{N}(0, 1)$
y
x
2
4
6
8
10
data
model

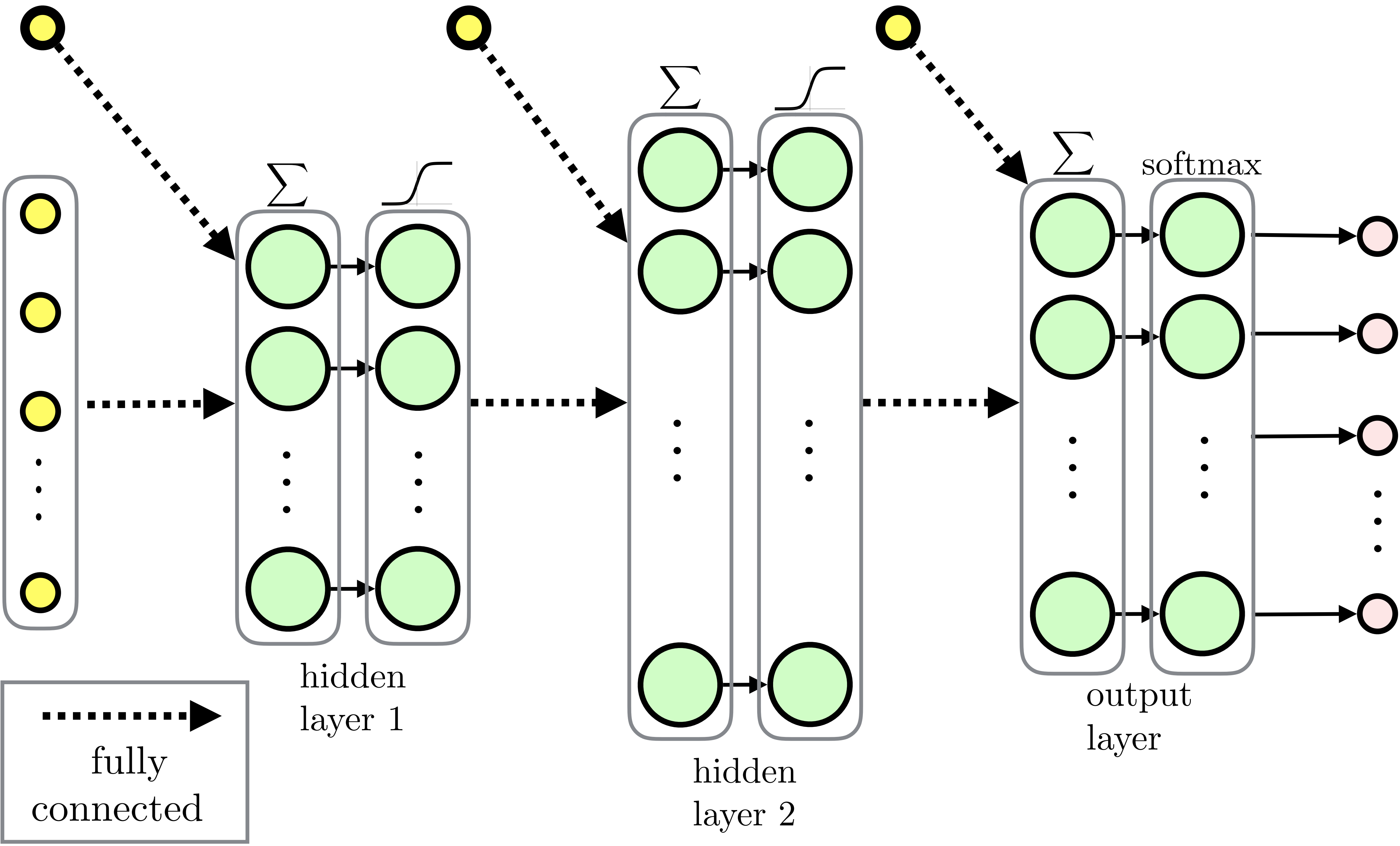

Sigmoid
$\sigma(s) = \frac{1}{1+e^{-s}}$
1
$\frac{1}{2}$
−10
10


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

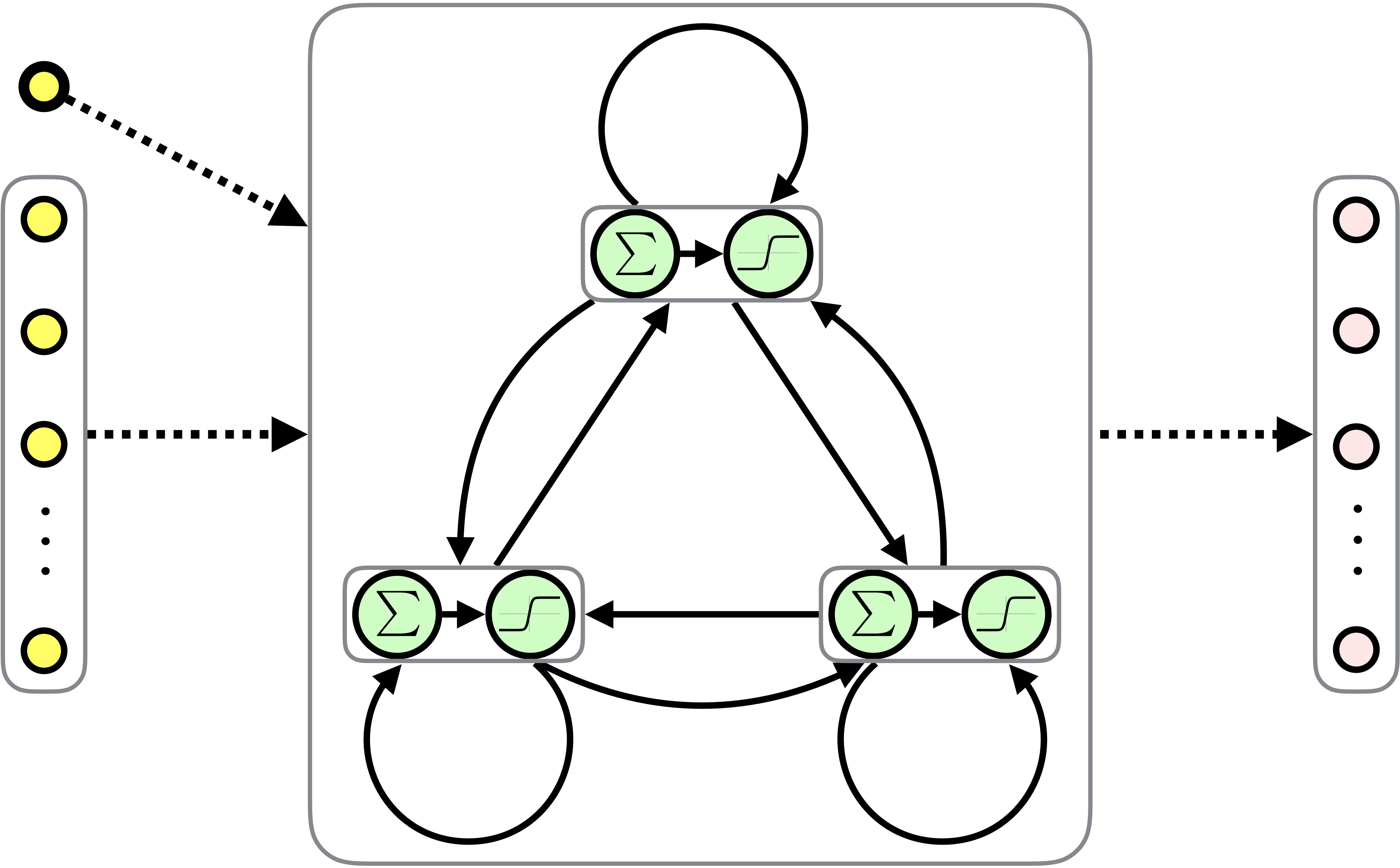

softmax
s_1
s_2
s_3
s_K
1
2
3
K
$\hat{y}_1$
$\hat{y}_2$
$\hat{y}_3$
$\hat{y}_K$

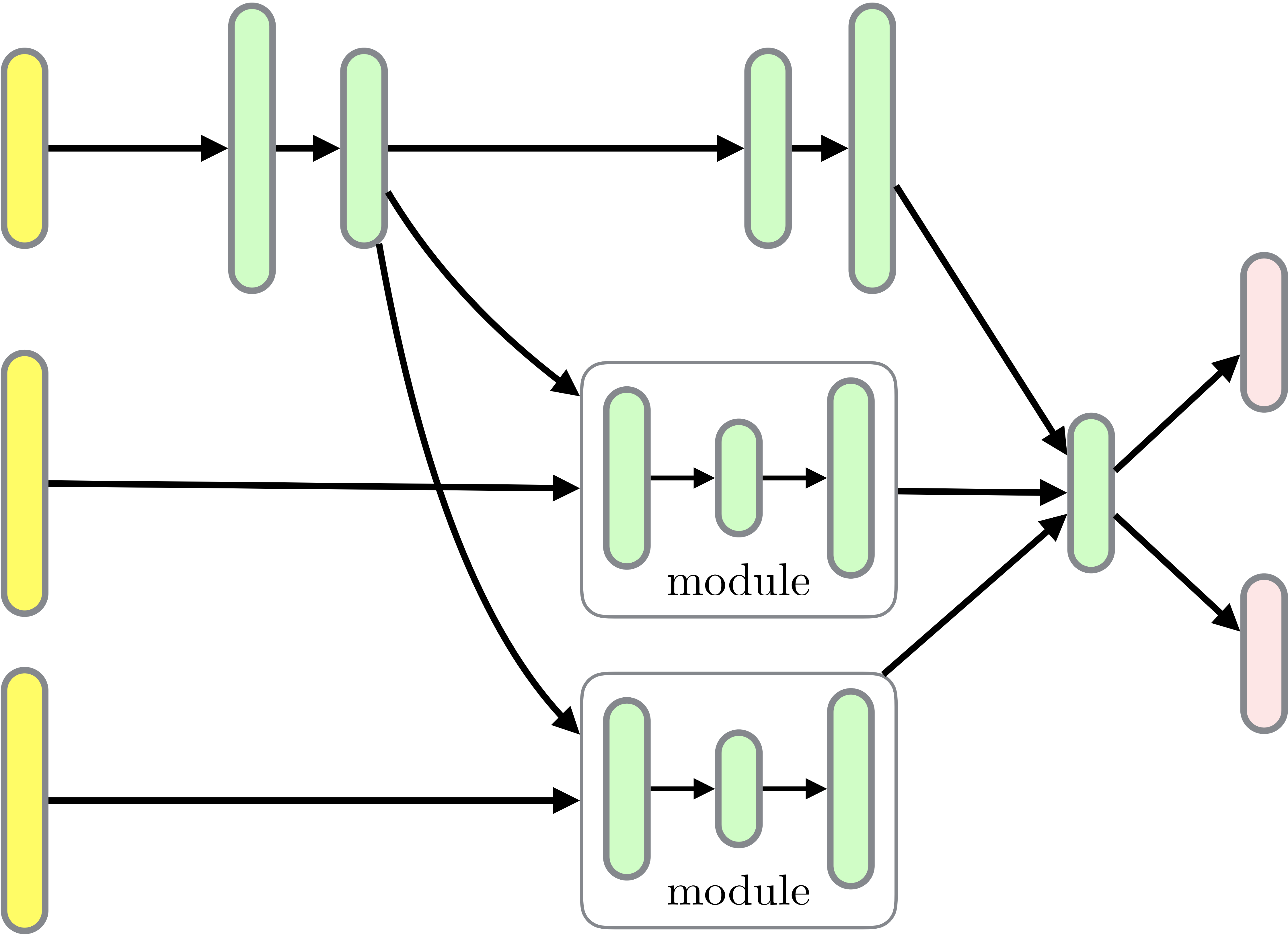


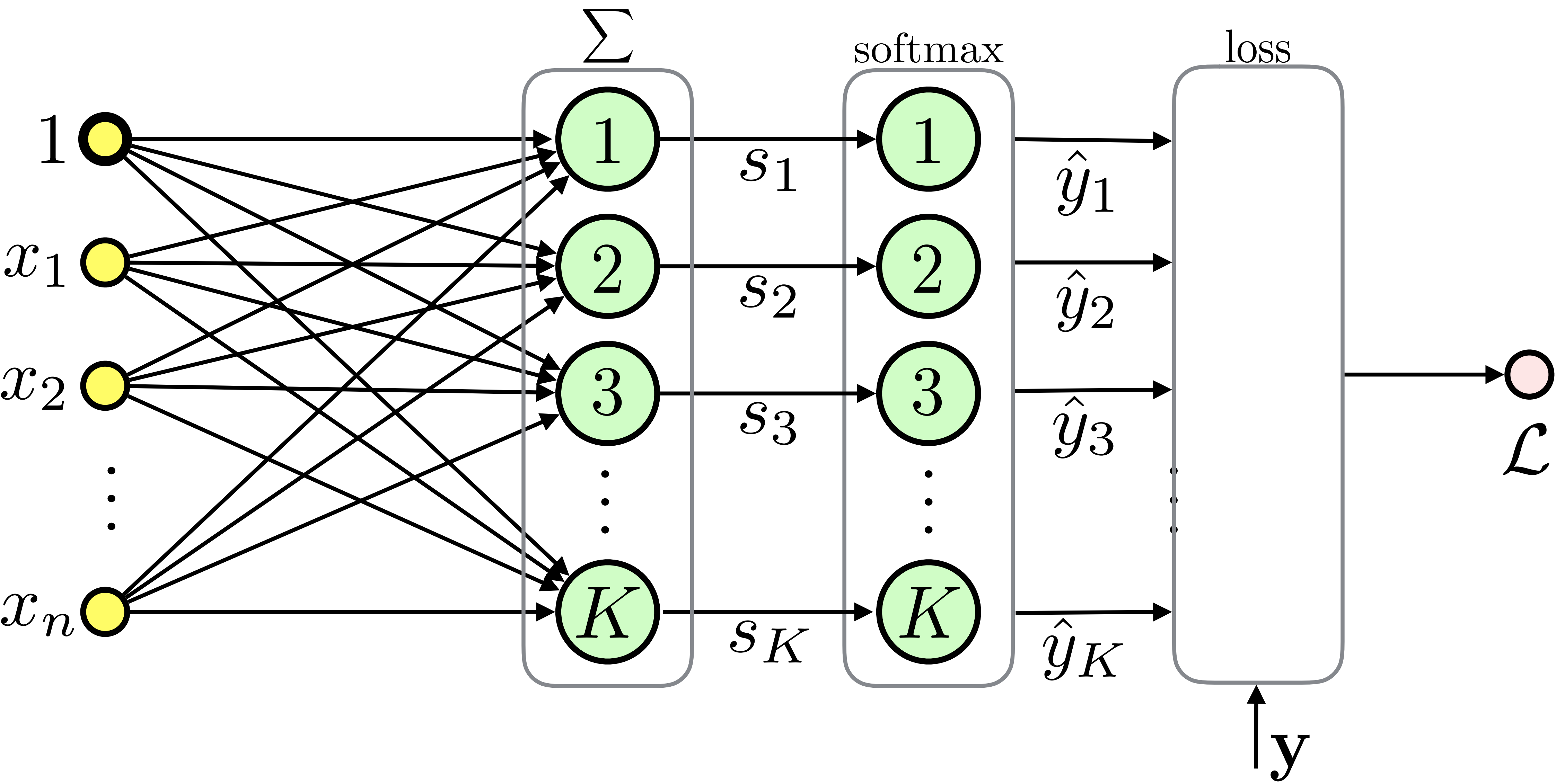
Σ
softmax
1
x_1
x_2
x_n
1
2
3
K
s_1
s_2
s_3
s_K
1
2
3
K
$\hat{y}_1$
$\hat{y}_2$
$\hat{y}_3$
$\hat{y}_K$



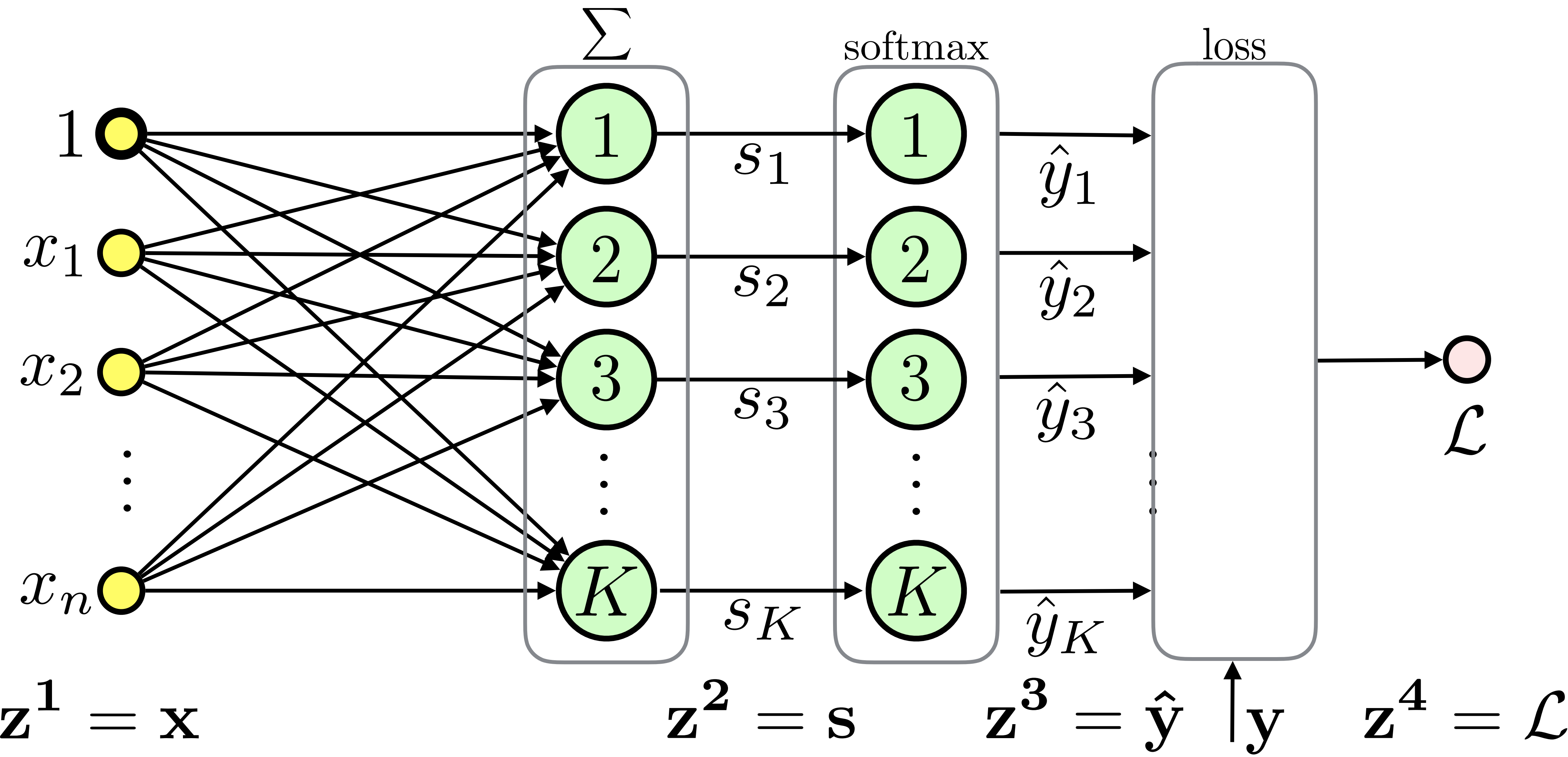
$\sum$
softmax
hidden
layer 1
hidden
layer 2
output
layer
fully
connected



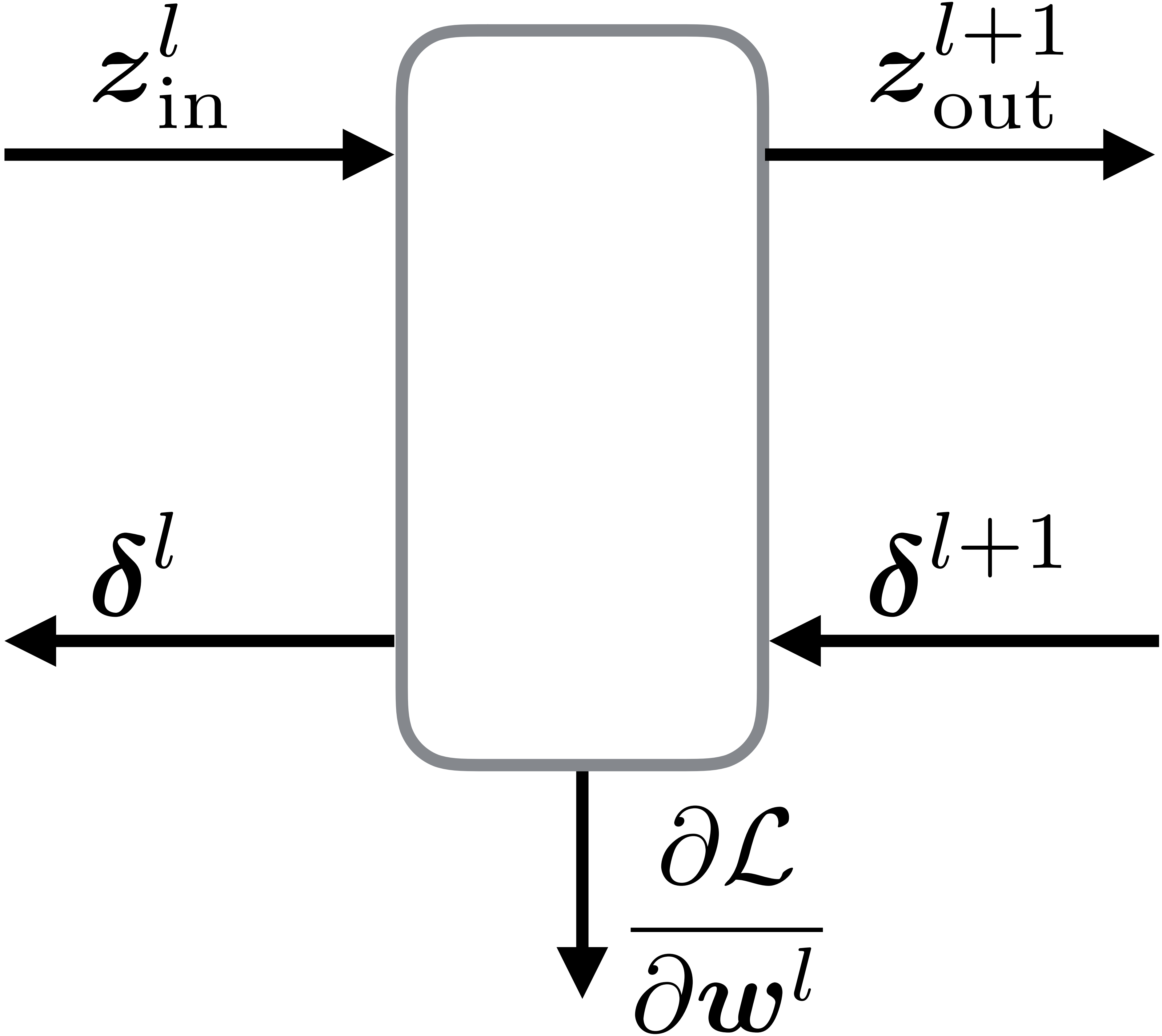

module
module



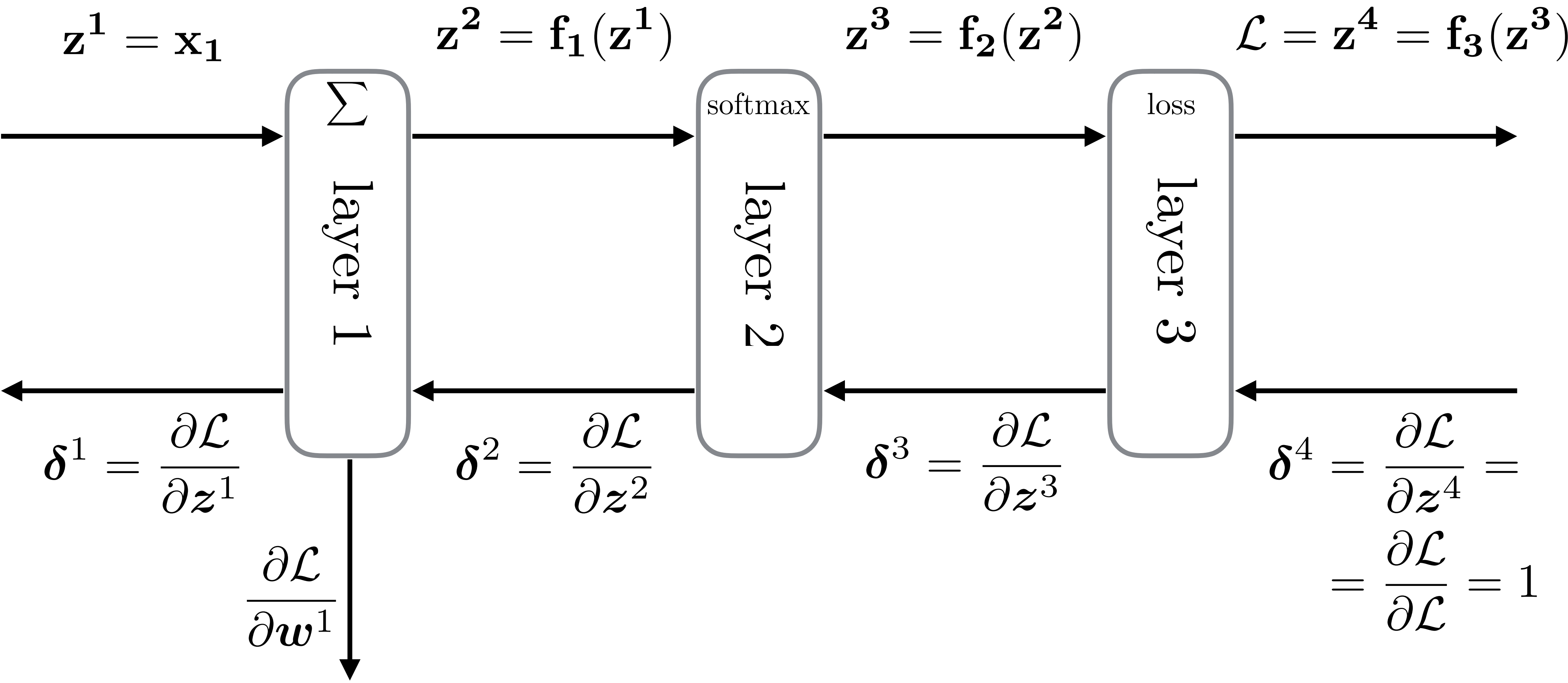
$\sum$
softmax
loss
1
x_1
x_2
x_n
1
2
3
K
s_1
s_2
s_3
s_K
1
2
3
K
$\hat{y}_1$
$\hat{y}_2$
$\hat{y}_3$
$\hat{y}_K$
$\mathbf{y}$
$\mathcal{L}$



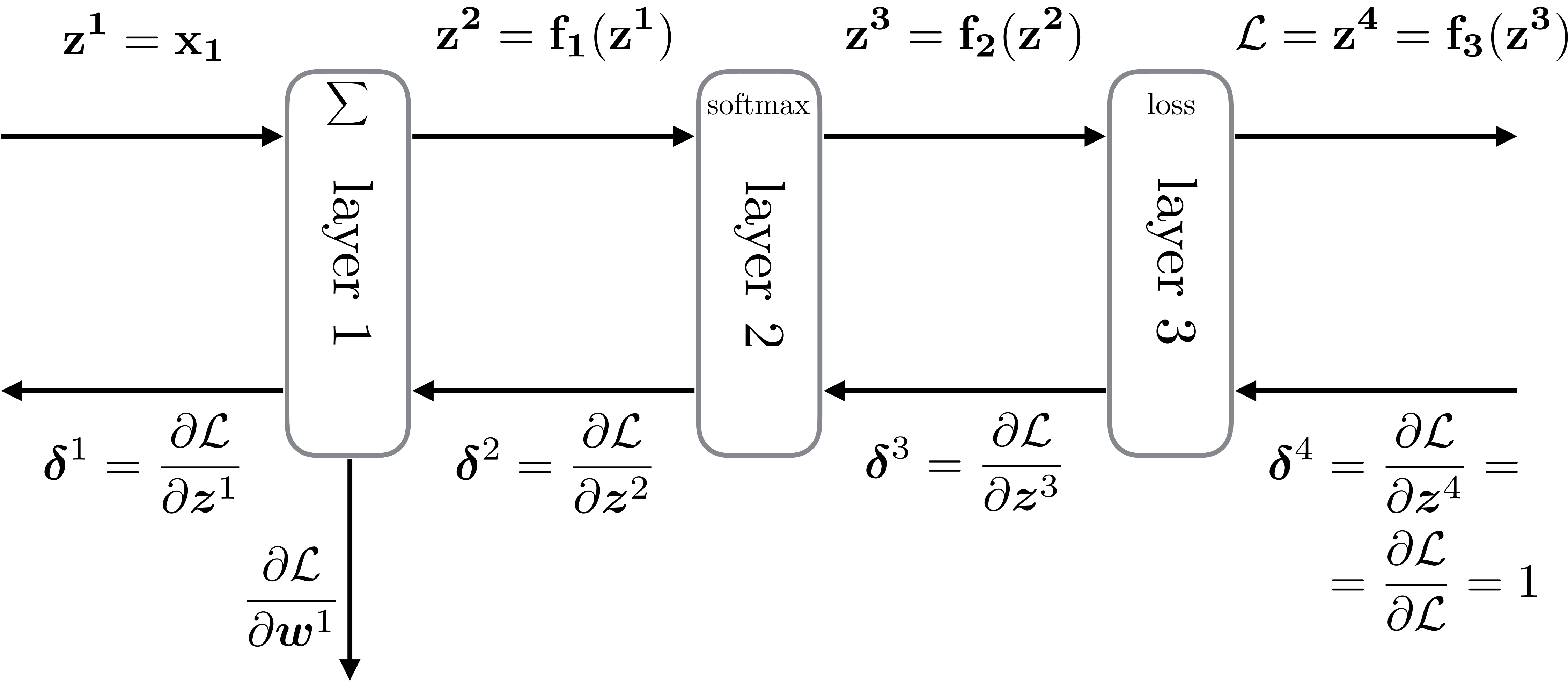
Σ
softmax
loss
1
x_1
x_2
x_n
1
2
3
K
s_1
s_2
s_3
s_K
$\hat{y}_1$
$\hat{y}_2$
$\hat{y}_3$
$\hat{y}_K$
$\mathcal{L}$
$\mathbf{z^1} = \mathbf{x}$
$\mathbf{z^2} = \mathbf{s}$
$\mathbf{z^3} = \hat{\mathbf{y}}$
$\mathbf{y}$
$\mathbf{z^4} = \mathcal{L}$



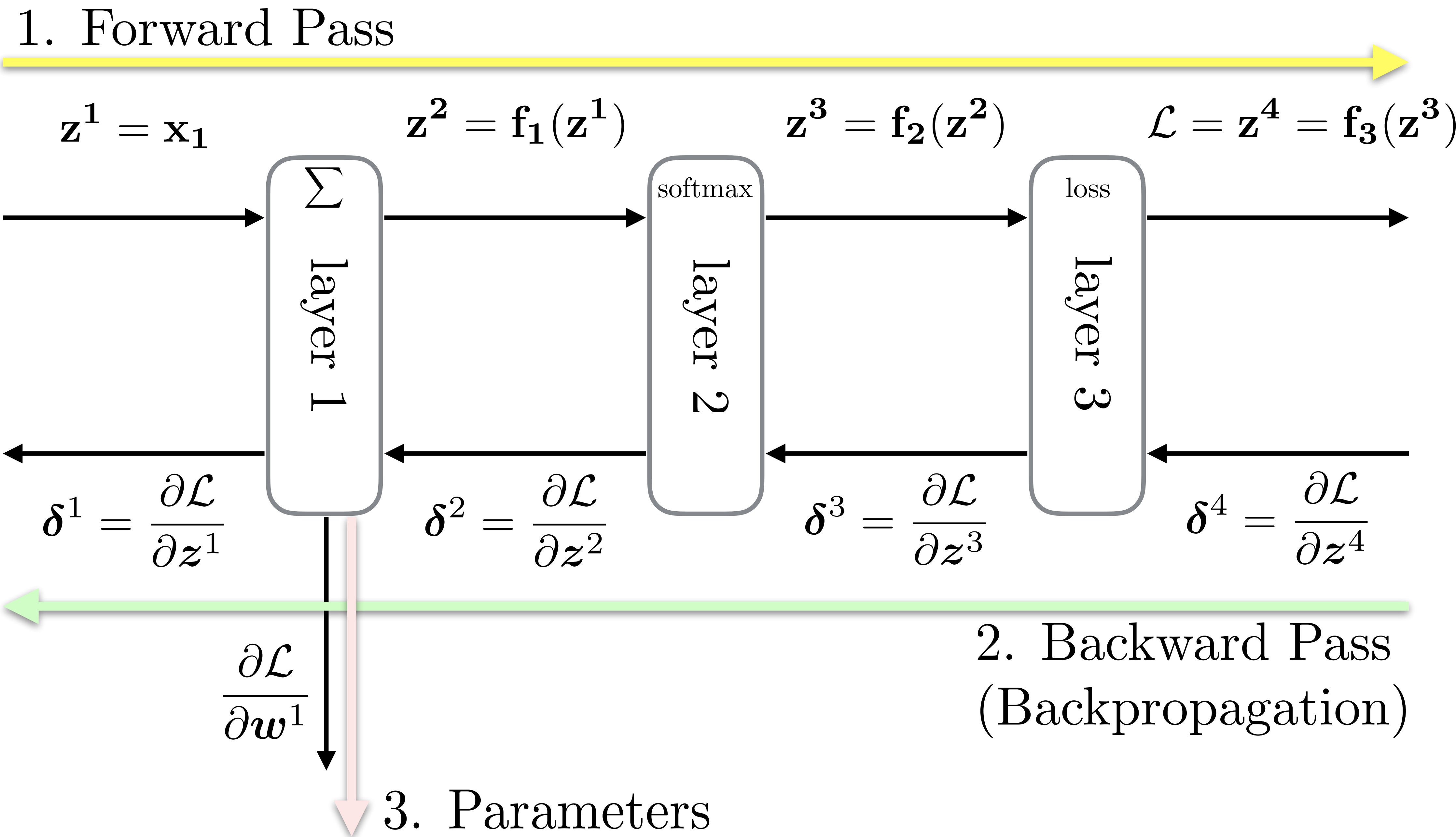
$\boldsymbol{z}_{\mathrm{in}}^{l}$
$\boldsymbol{z}_{\mathrm{out}}^{l+1}$
$\boldsymbol{\delta}^{l}$
$\boldsymbol{\delta}^{l+1}$
$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}^{l}}$

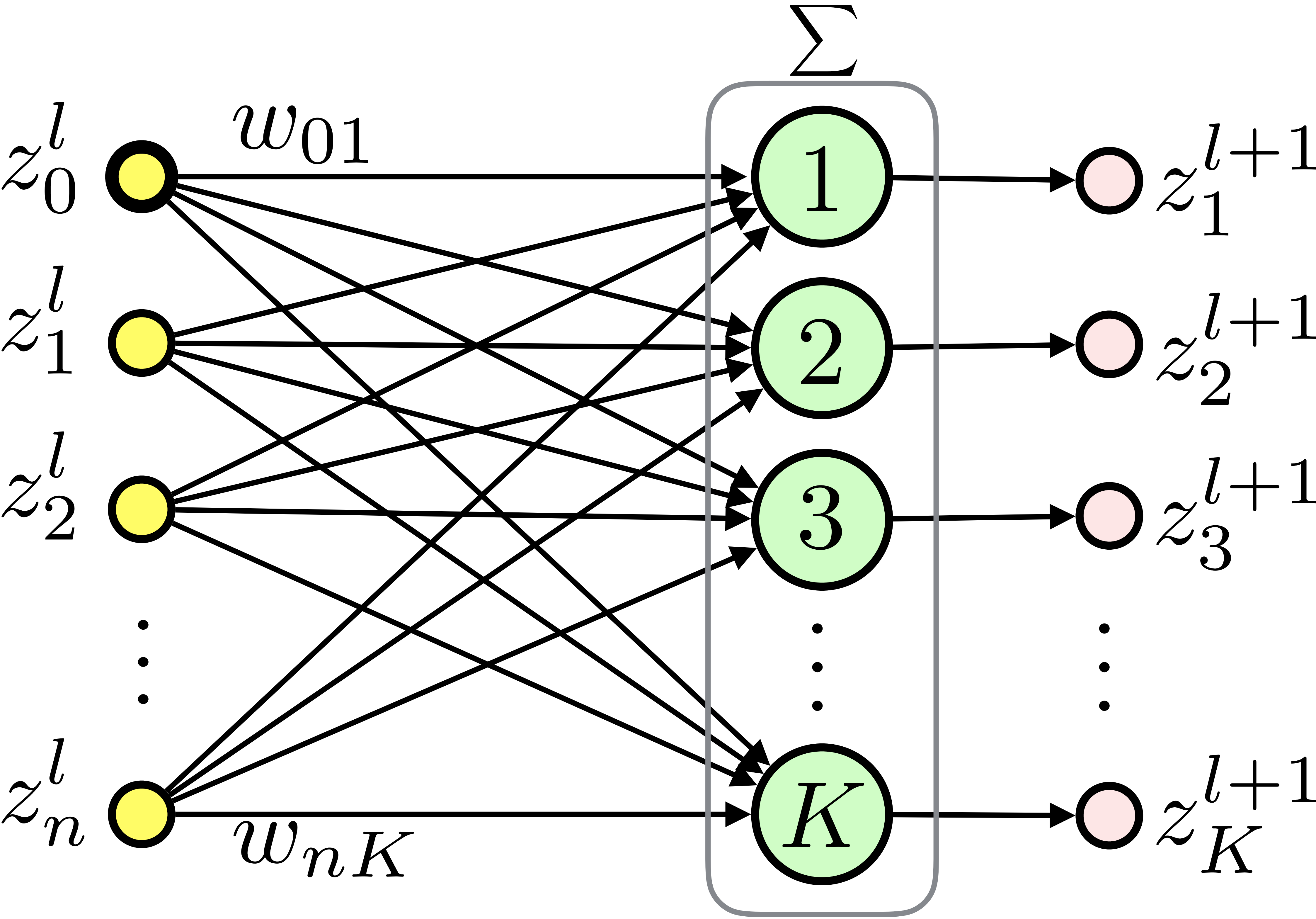


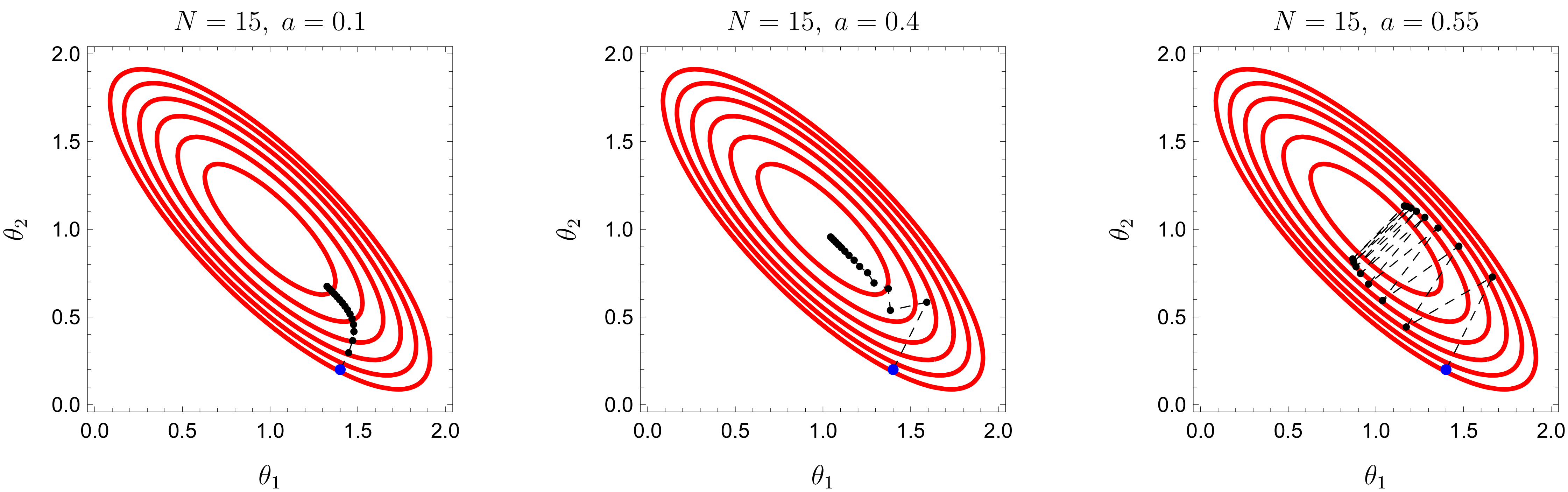
$\mathbf{z^1} = \mathbf{x_1}$
$\mathbf{z^2} = \mathbf{f_1}(\mathbf{z^1})$
$\mathbf{z^3} = \mathbf{f_2}(\mathbf{z^2})$
$\mathcal{L} = \mathbf{z^4} = \mathbf{f_3}(\mathbf{z^3})$
$\sum$
layer 1
softmax
layer 2
loss
layer 3
$\boldsymbol{\delta}^1 = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^1}$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



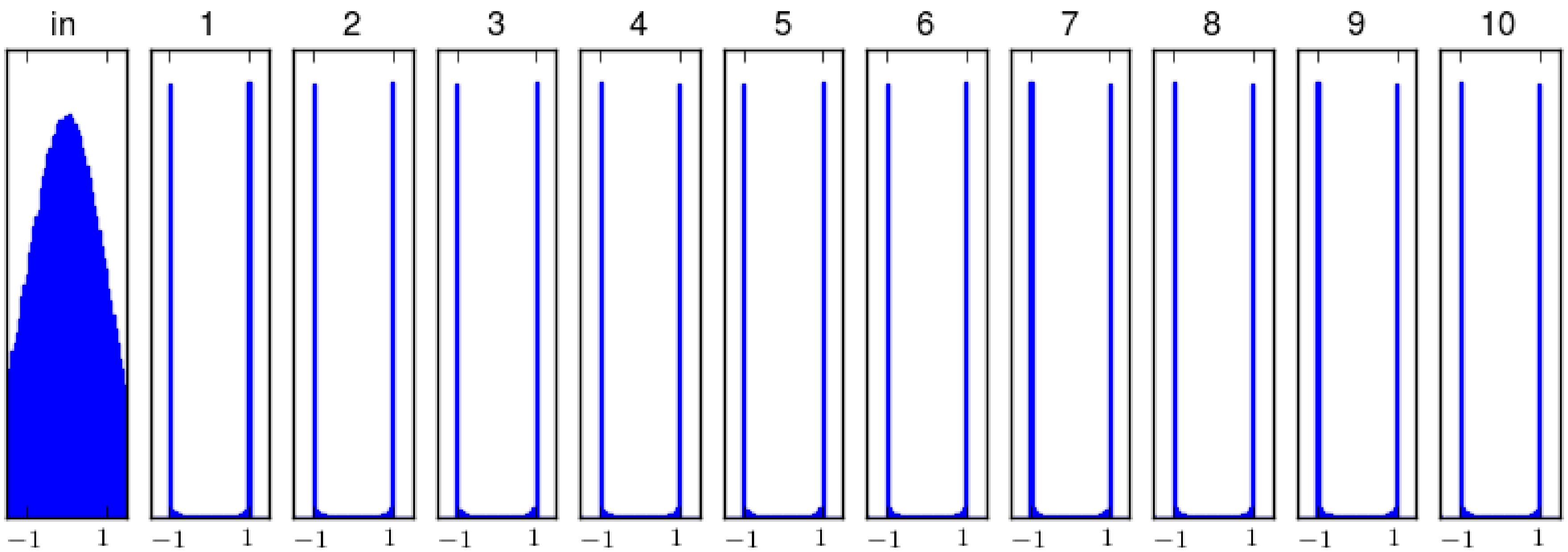
$\mathbf{z^1} = \mathbf{x_1}$
$\mathbf{z^2} = \mathbf{f_1}(\mathbf{z^1})$
$\mathbf{z^3} = \mathbf{f_2}(\mathbf{z^2})$
$\mathcal{L} = \mathbf{z^4} = \mathbf{f_3}(\mathbf{z^3})$
$\sum$
layer 1
softmax
layer 2
loss
layer 3
$\boldsymbol{\delta}^1 = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^1}$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

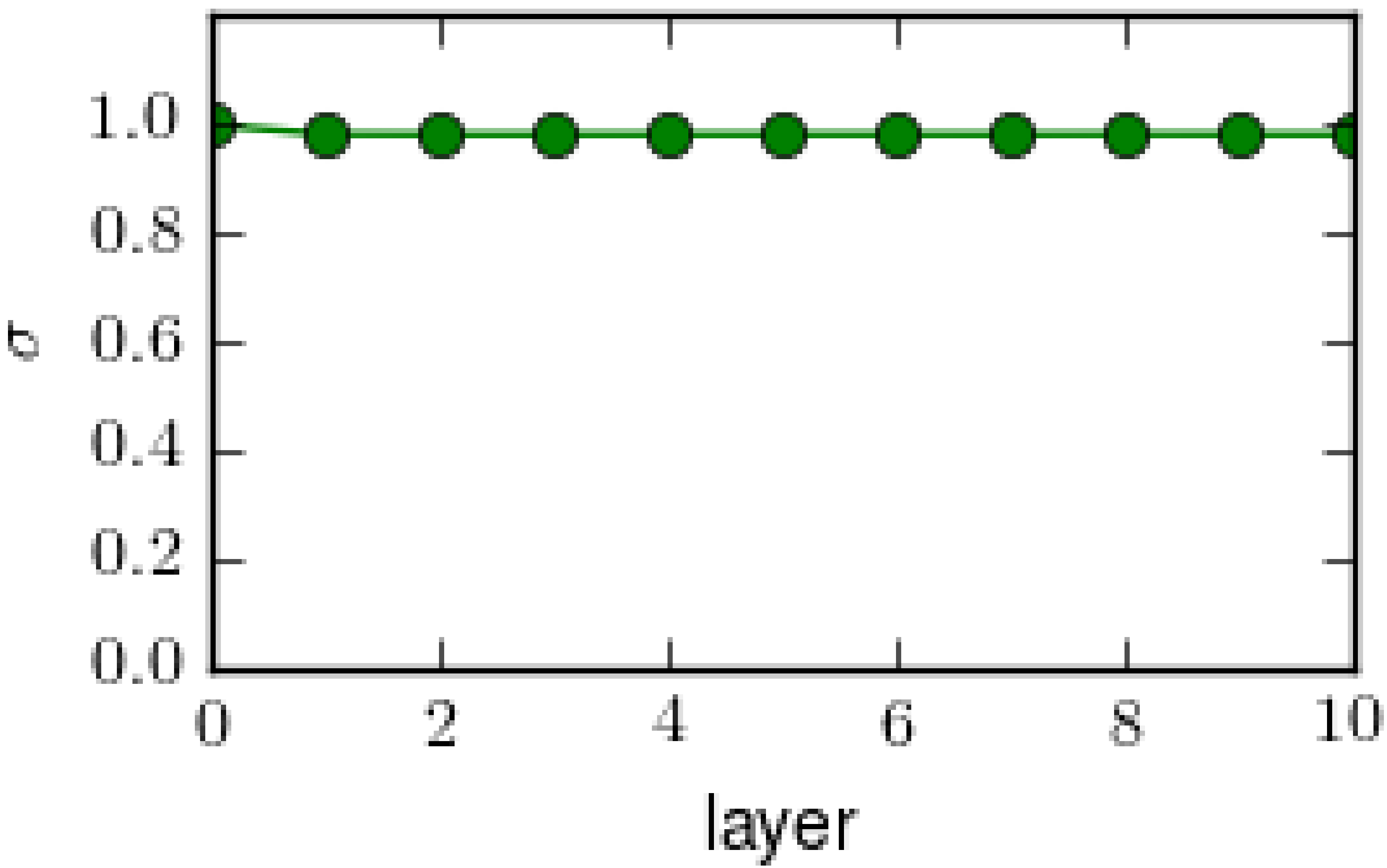

1. Forward Pass
$\mathbf{z^1 = x_1}$
$\mathbf{z^2 = f_1(z^1)}$
$\mathbf{z^3 = f_2(z^2)}$
$\mathcal{L} = \mathbf{z^4 = f_3(z^3)}$
Σ
layer 1
softmax
layer 2
loss
layer 3
$\boldsymbol{\delta}^1 = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^1}$
$\boldsymbol{\delta}^2 = \frac{\partial \mathcal{L}}{\partial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


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

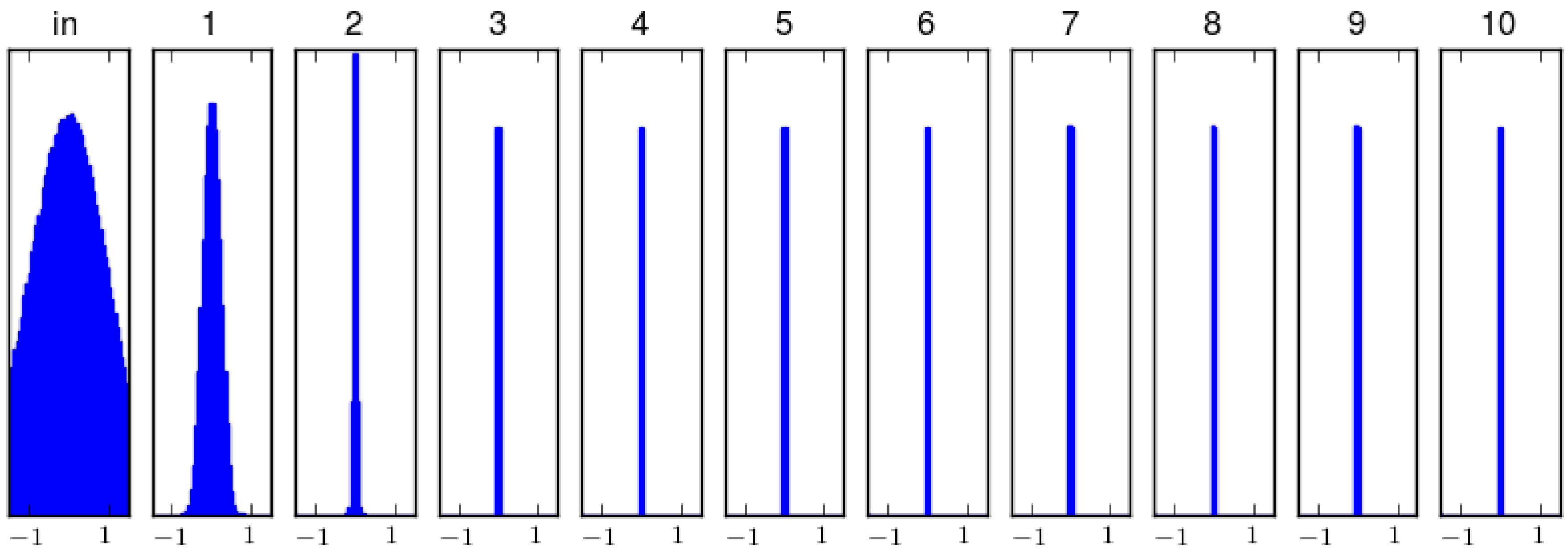


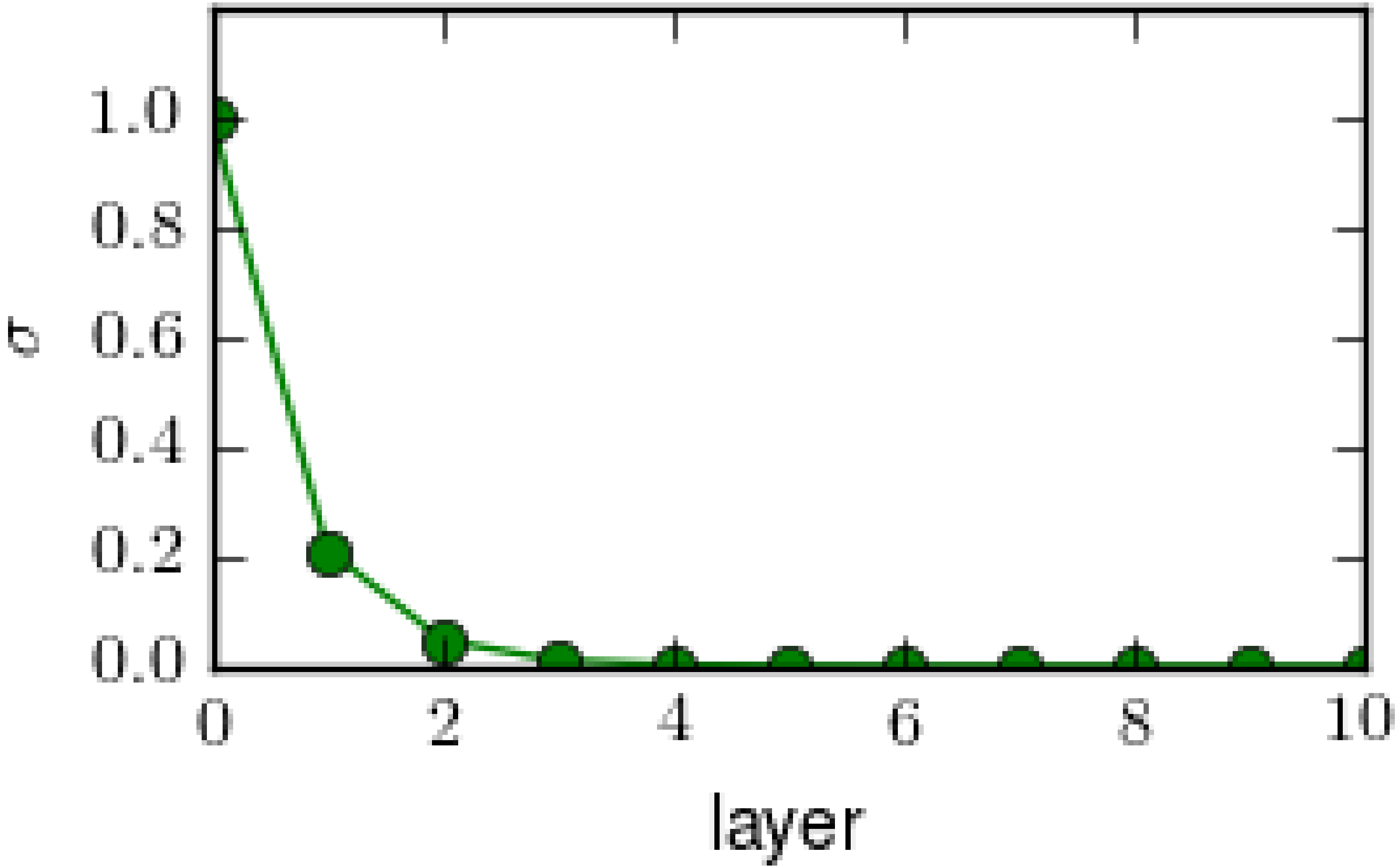
$N = 15,\ a = 0.1$
$N = 15,\ a = 0.4$
$N = 15,\ a = 0.55$
θ_1
θ_2
0.0
0.5
1.0
1.5
2.0


in
1
2
3
4
5
6
7
8
9
10
−1
1

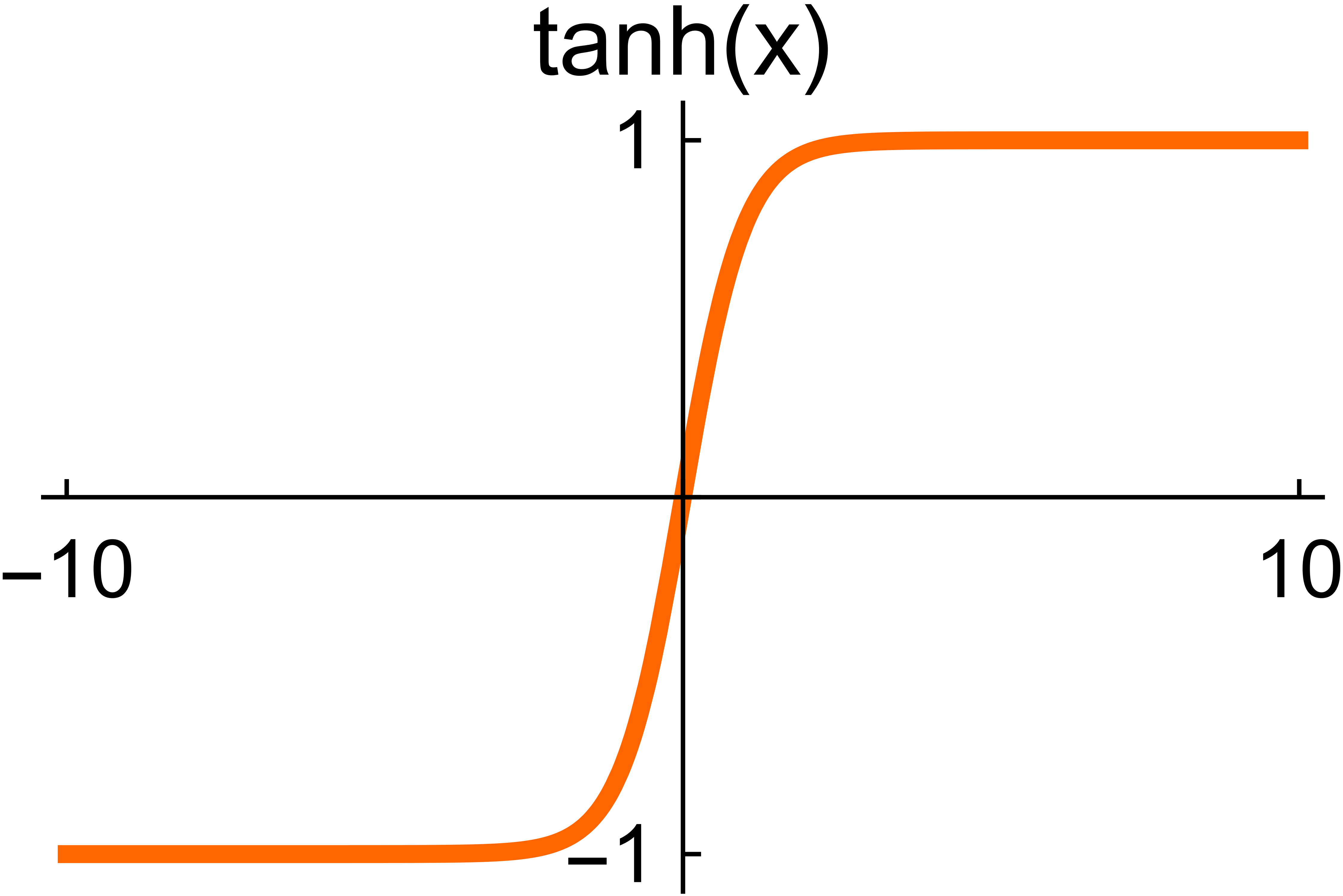


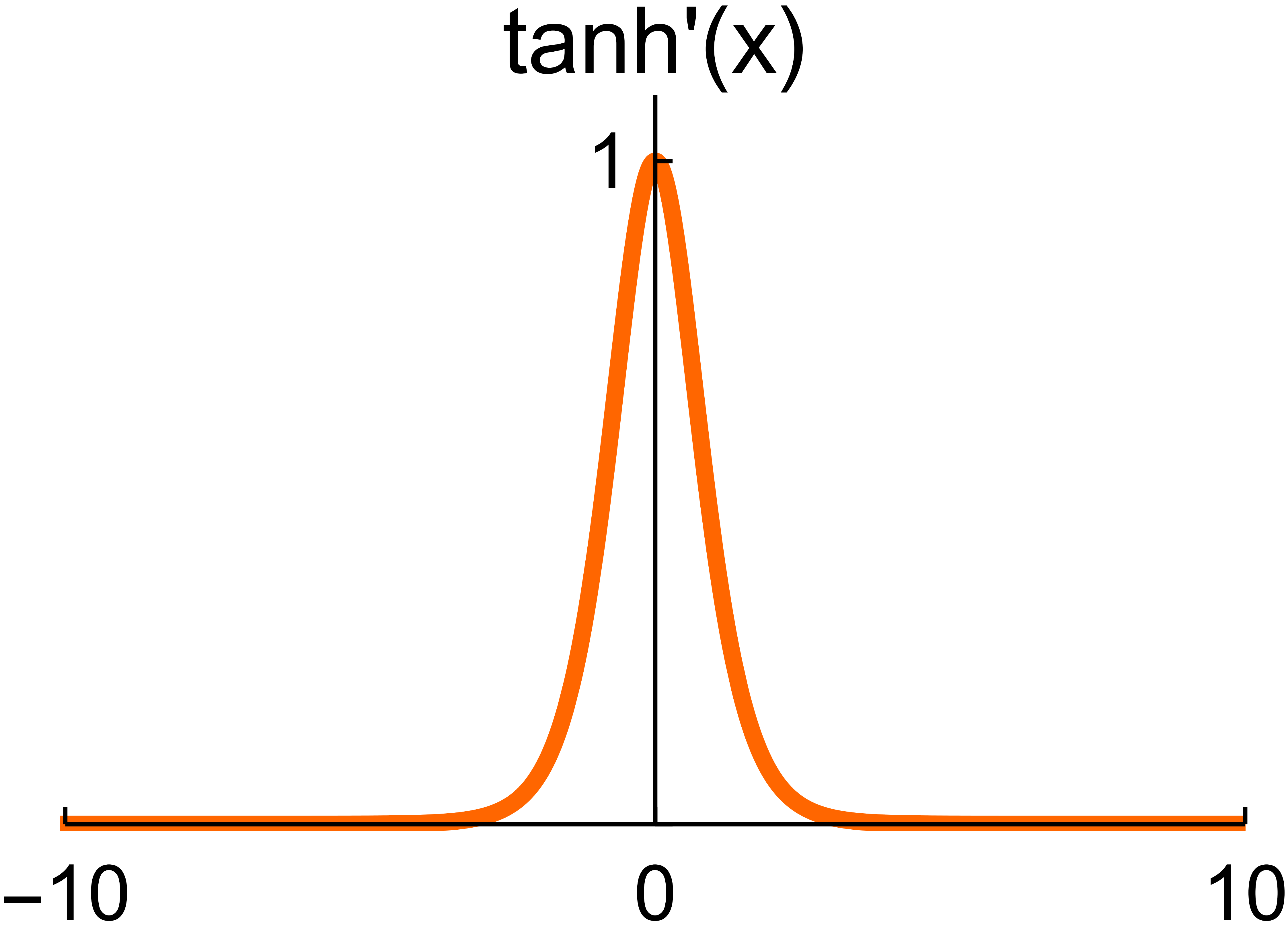
σ
1.0
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
10
layer

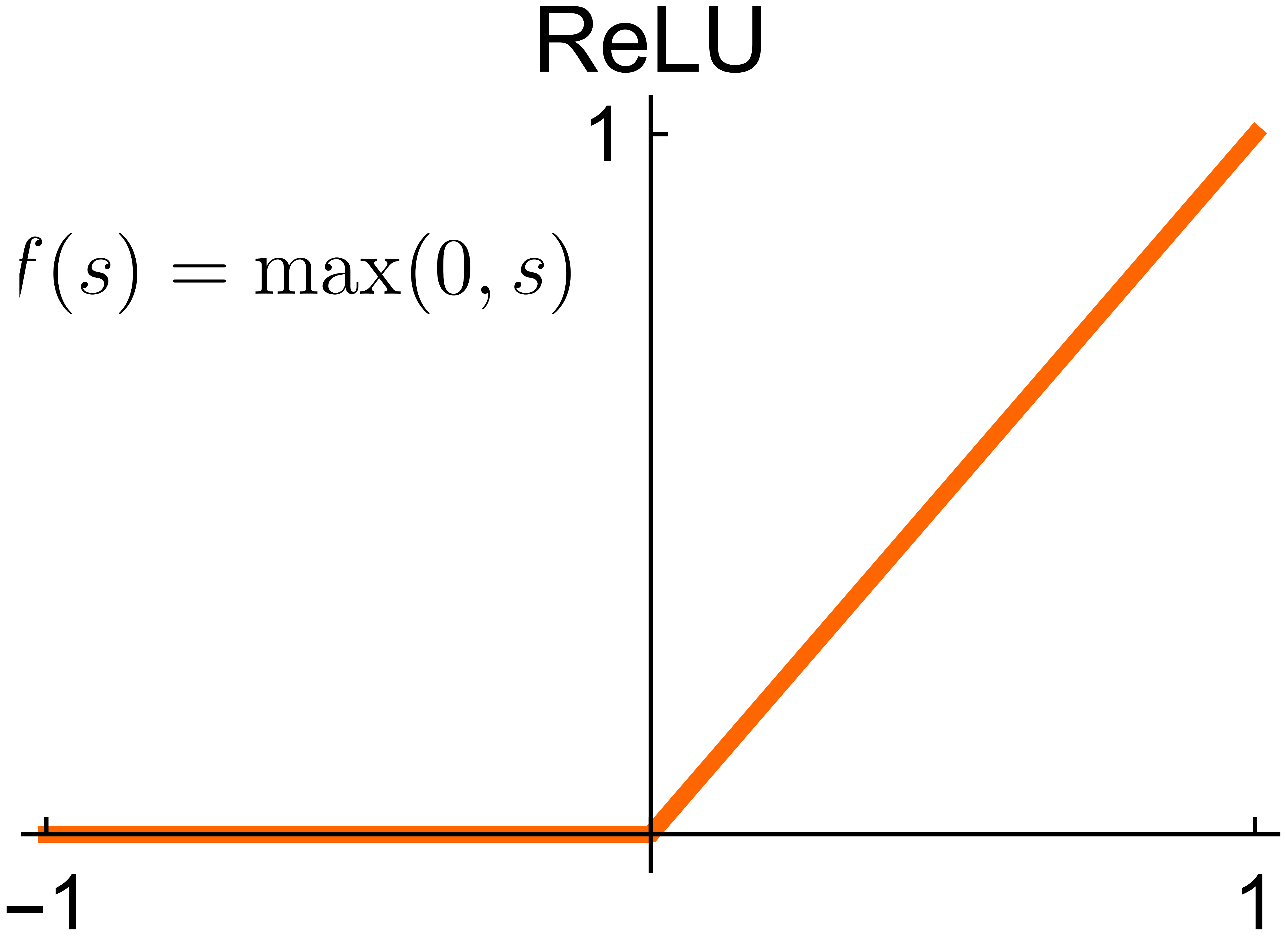

in
1
2
3
4
5
6
7
8
9
10
−1
1

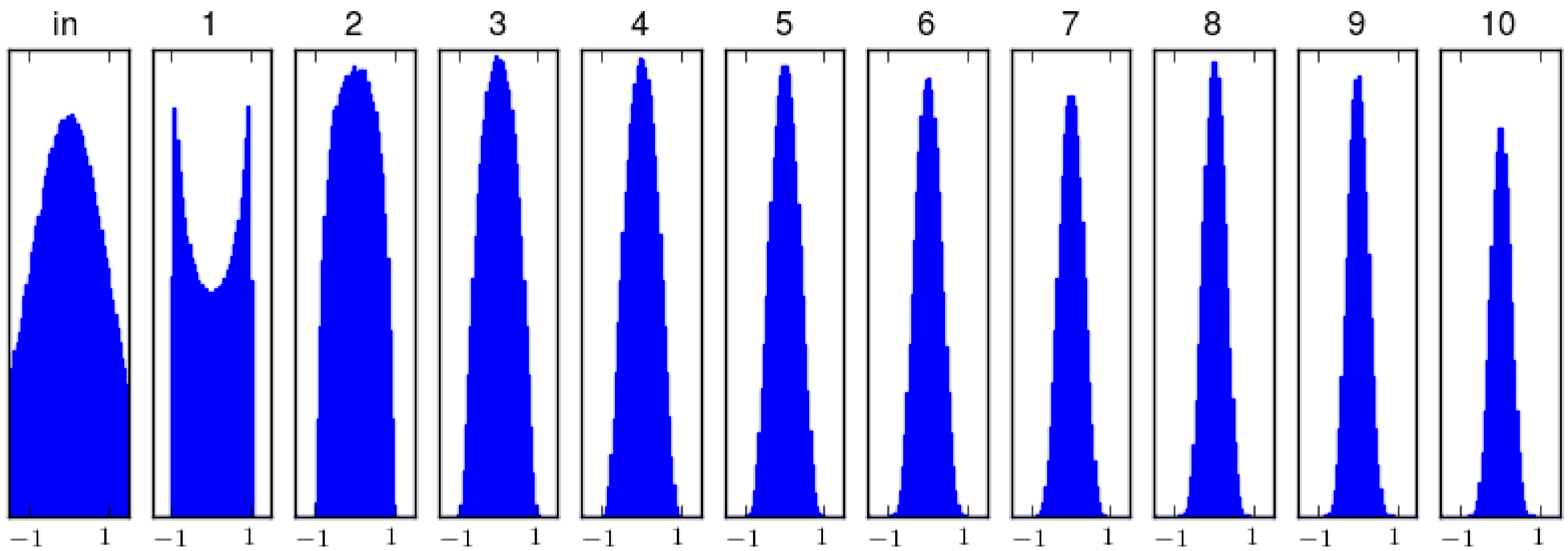


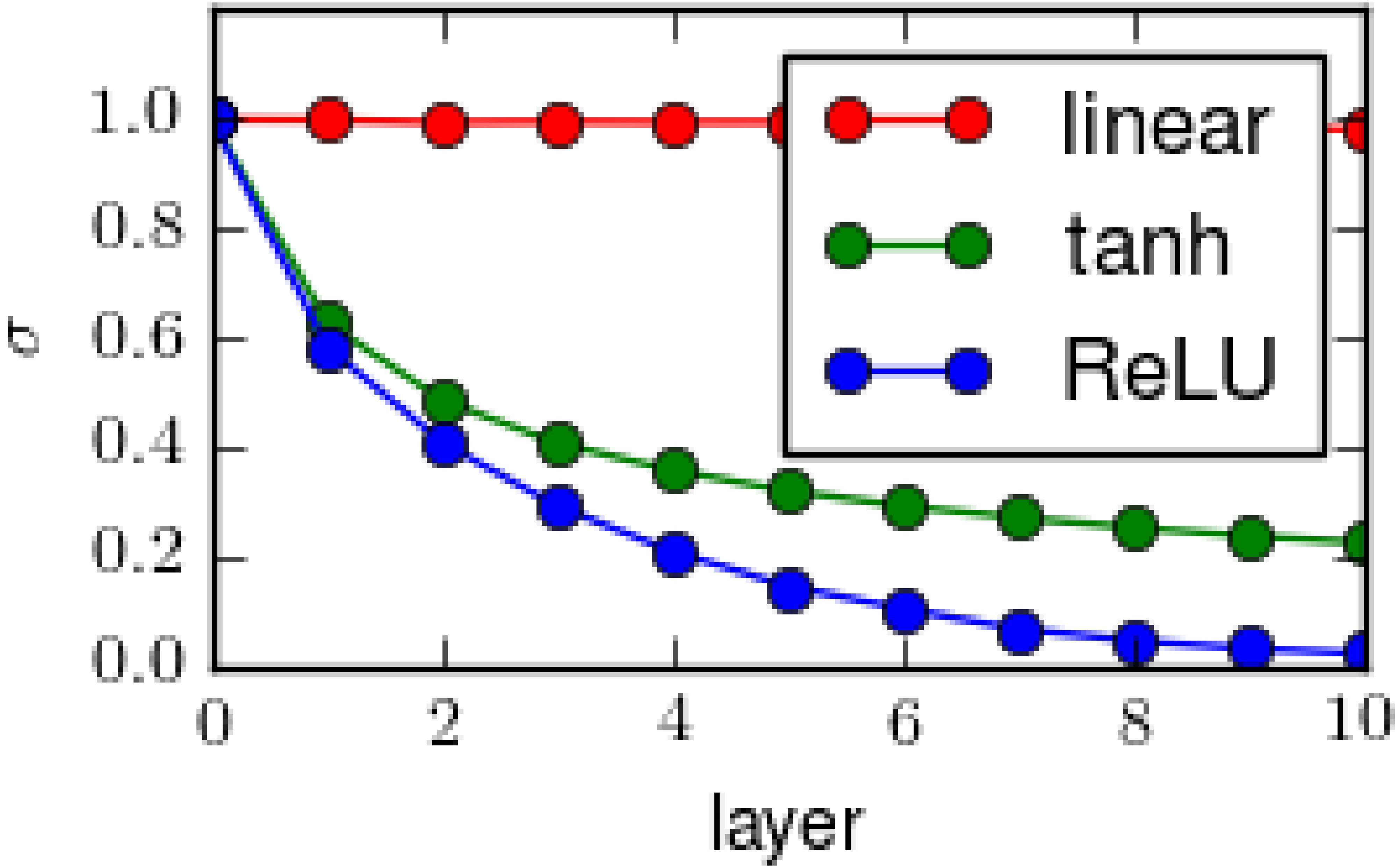
σ
1.0
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
10
layer


tanh(x)
1
−1
−10
10

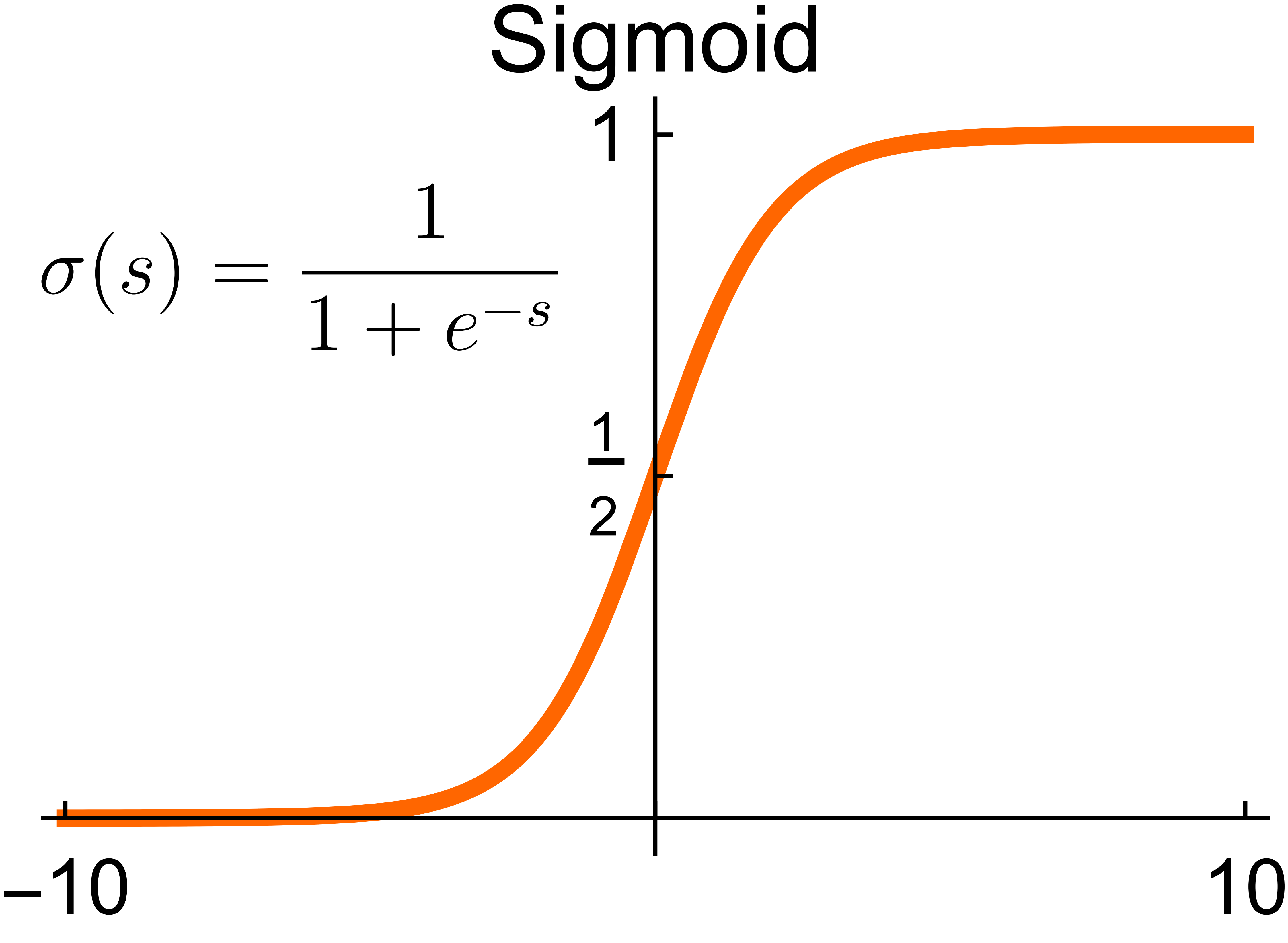

tanh'(x)
1
−10
0
10


ReLU
1
$f(s) = \max(0, s)$
−1
1


in
1
2
3
4
5
6
7
8
9
10
−1
1



linear
tanh
ReLU
1.0
0.8
0.6
0.4
0.2
0.0
σ
0
2
4
6
8
10
layer



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