Imperial College London

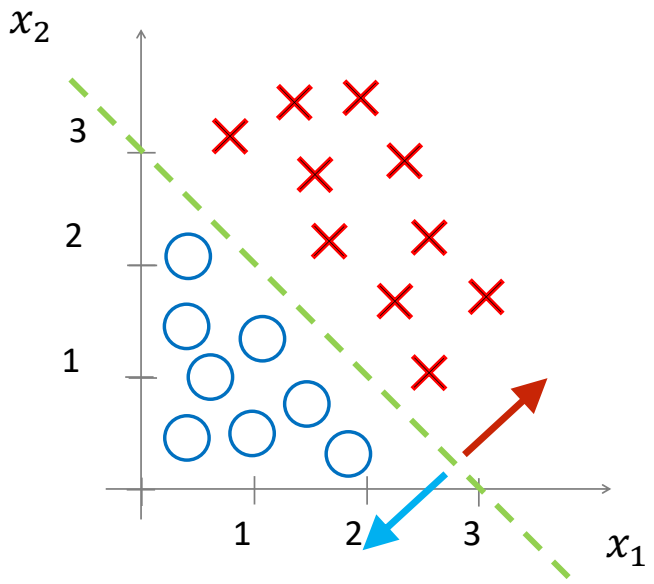
Lecture 5: Introduction to Neural Networks

Daniel Rueckert
Department of Computing
Imperial College London, UK

Neural networks

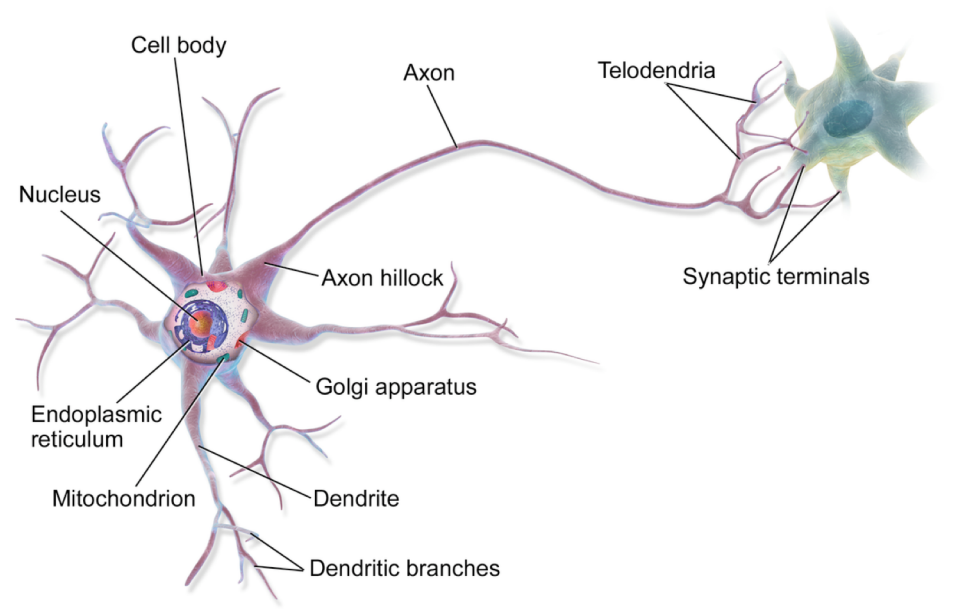
Recap: Logistic regression model $h_\Theta(\boldsymbol{x})$:

$$\hat{y} = g(\boldsymbol{x}^T\boldsymbol{\theta}) \qquad g(z) = \frac{1}{1+e^{-z}}$$

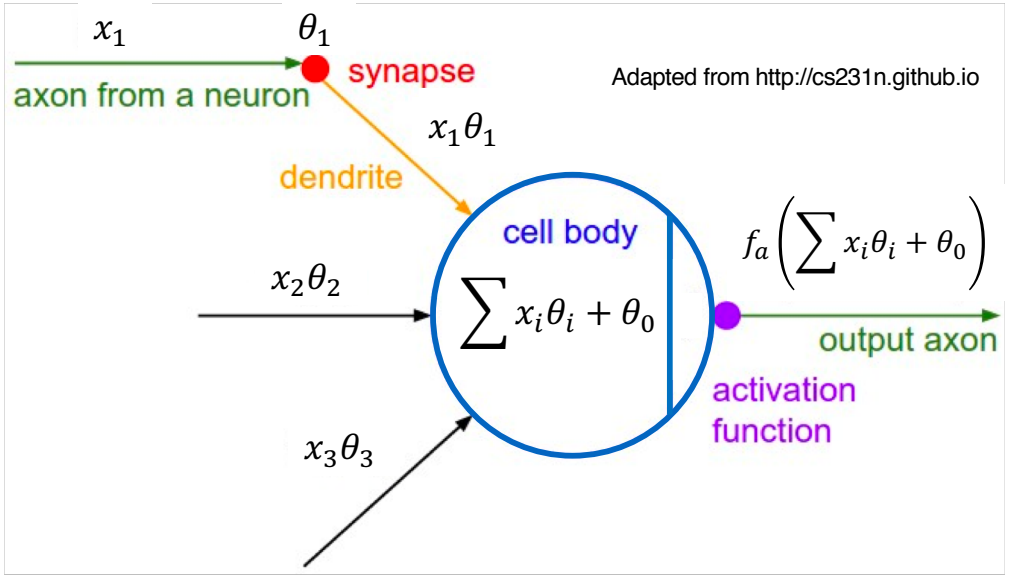


Neural networks: Some history

- First machine learning methods were inspired by how the brain works:



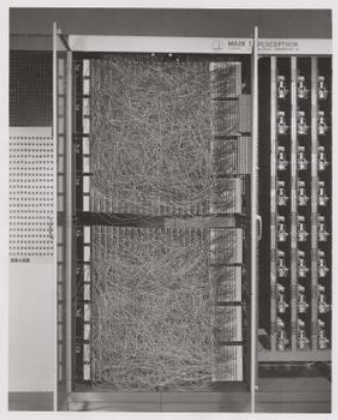
Biological neuron



Artificial neuron

Neural networks: Some history
Perceptron

- The first neural network (Frank Rosenblatt, 1957)

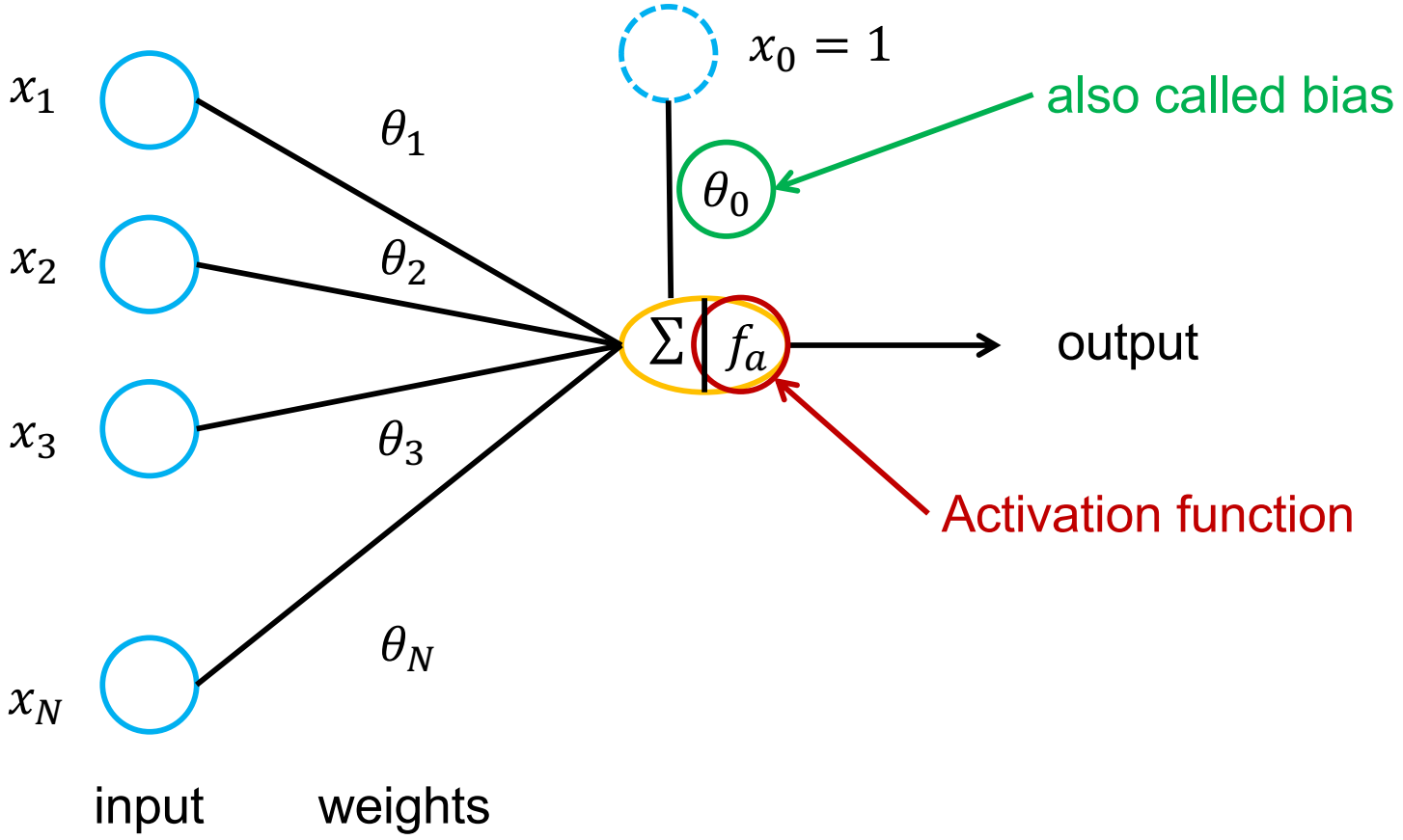


"Mark 1 perceptron“ - machine designed for image recognition: it had an array of 400 photocells, randomly connected to the "neurons". Weights were encoded in potentiometers, and weight updates during learning were performed by electric motors (source Wikipedia)

$$f_a(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \sum_i x_i \theta_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

A single-layer perceptron

- A single-layer perceptron looks as follows:



A single-layer perceptron

- Input:
  - Each input to the neuron $(x_1, \dots, x_n)$ is known as a feature
- Weights:
  - Each feature is weighted with a number to represent the strength of that input $(\theta_1, \dots, \theta_n)$
- Activation function:
  - Calculate weighted sum of inputs and pass through an activation function and threshold result to 0 or 1
  - This models the “firing rate” of a biological neuron

Training a single-layer perceptron

- Training a perceptron means to iteratively updating the weights associated with each of its inputs
- This allows to progressively approximate the underlying relationship in the given training dataset
- Once properly trained, it can be used to classify entirely new samples

Training a single-layer perceptron

1. Initialize the weights and the bias (or threshold). Weights may be initialized to 0 or to a small random value.
2. For each example in our training set $\mathbf{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N\}$ with labels $\boldsymbol{y} = \{y_1, y_2, \ldots, y_N\}$ perform the following steps over the input $\boldsymbol{x}_n$ and desired output y_n:
   a. Calculate the actual output

$$\hat{y}_n = f(\boldsymbol{x}_n^T \theta)$$

   a. For all features $m = 0, \ldots, M$ update weights

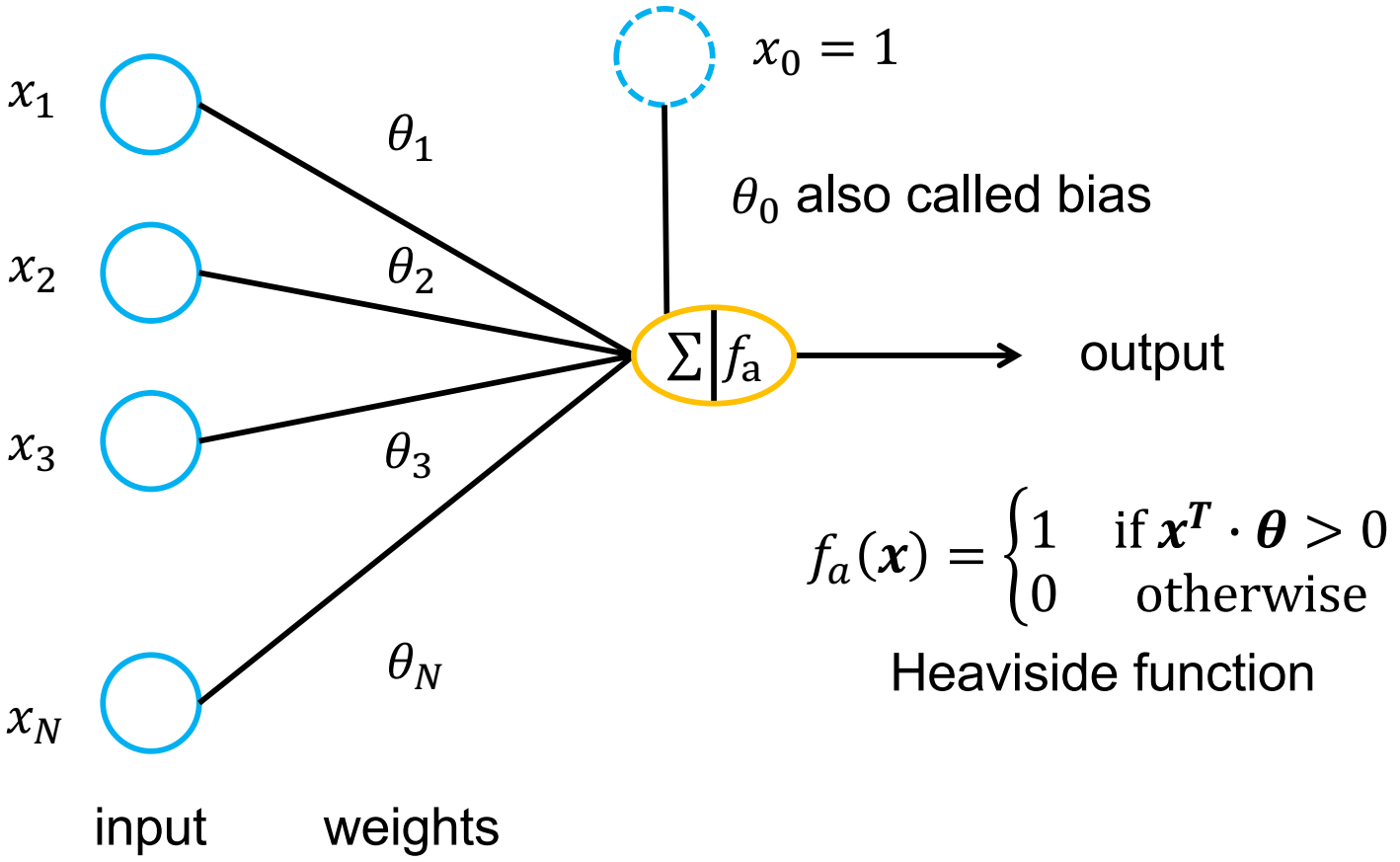
$$\theta_m^{t+1} = \theta_m^t + \tau(y_n - \hat{y}_n)x_{m,n}$$

$\boldsymbol{x}_n$ is a vector of length M

τ is the learning rate

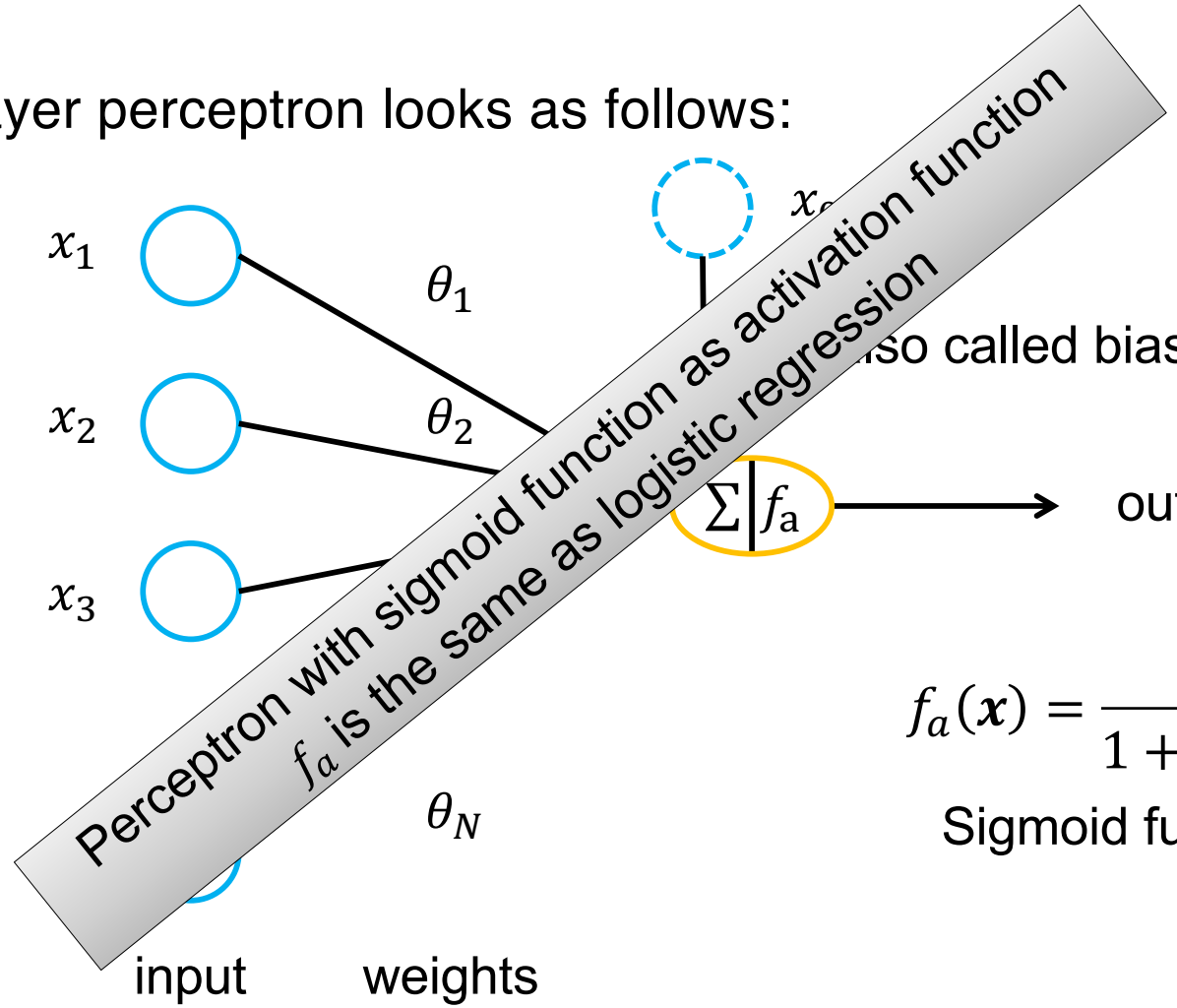
A single-layer perceptron

- A single-layer perceptron looks as follows:



A single-layer perceptron

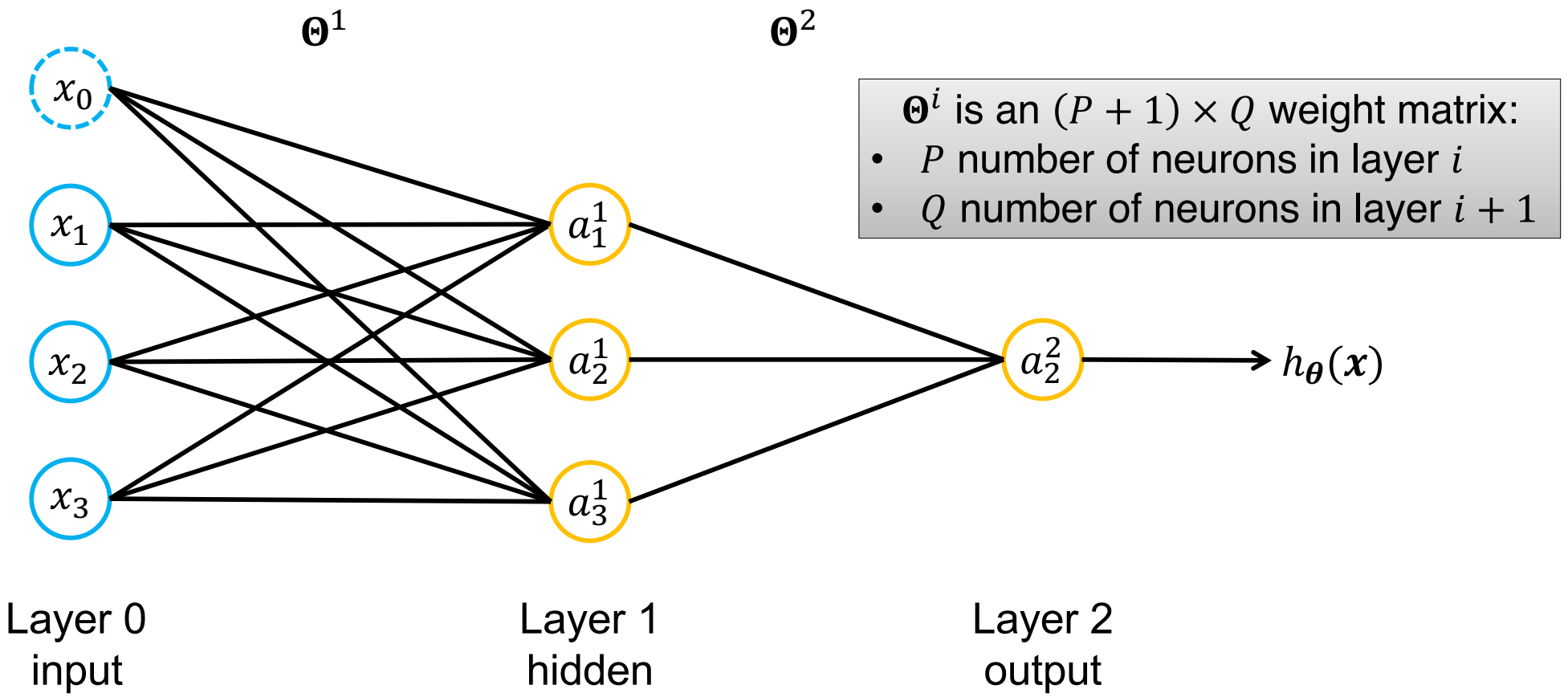
- A single-layer perceptron looks as follows:



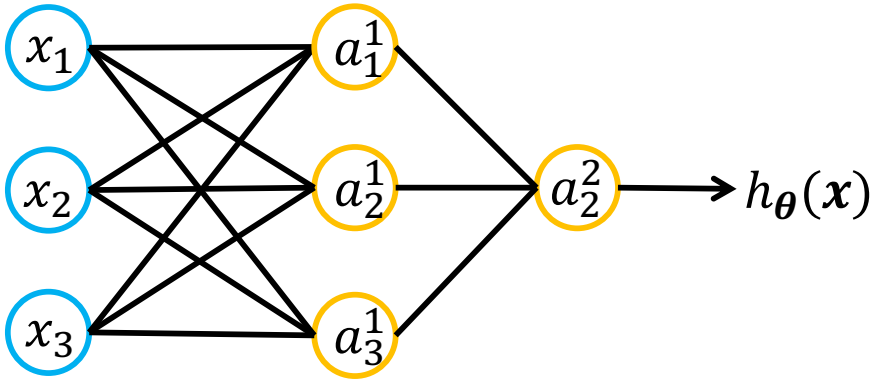
$$f_a(\boldsymbol{x}) = \frac{1}{1 + e^{-\boldsymbol{x}^T\boldsymbol{\theta}}}$$

Sigmoid function

Neural networks (or multilayer perceptron)



Neural networks (or multilayer perceptron)



$\mathbf{a}_j^i$ activation of neuron j in layer i

$\boldsymbol{\Theta}^i$ matrix of weights controlling mapping from layer $i-1$ to i

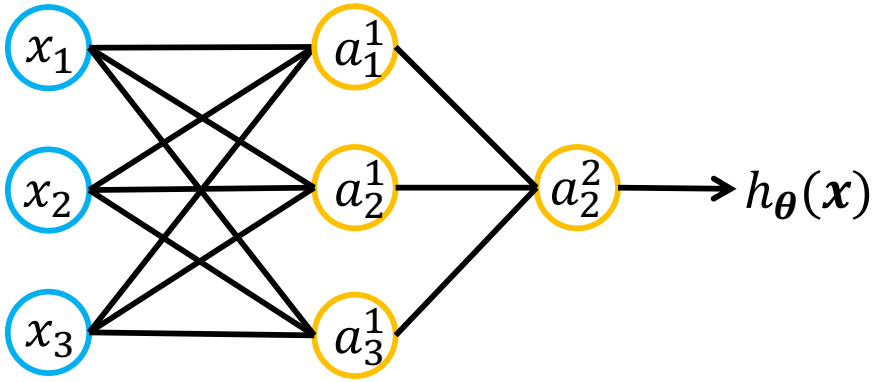
$$a_1^1 = f_a(\theta_{0,0}^1 + \theta_{1,0}^1 x_1 + \theta_{2,0}^1 x_2 + \theta_{3,0}^1 x_3)$$

$$a_2^2 = f_a(\theta_{0,1}^1 + \theta_{1,1}^1 x_1 + \theta_{2,1}^1 x_2 + \theta_{3,1}^1 x_3)$$

$$a_3^1 = f_a(\theta_{0,2}^1 + \theta_{1,2}^1 x_1 + \theta_{2,2}^1 x_2 + \theta_{3,2}^1 x_3)$$

$$h_{\boldsymbol{\theta}} = a_1^2 = f_a(\theta_{0,0}^2 + \theta_{1,0}^2 a_1 + \theta_{2,0}^2 a_2 + \theta_{3,0}^3 a_3)$$

Neural networks (or multilayer perceptron)



Elementwise application of f_a

$$\boldsymbol{a}^1 = f_a(\boldsymbol{x}^T \cdot \boldsymbol{\Theta}^1)$$

$$\boldsymbol{a}^2 = f_a(\boldsymbol{a}^1 \cdot \boldsymbol{\Theta}^2) = f_a(f_a(\boldsymbol{x}^T \cdot \boldsymbol{\Theta}^1) \cdot \boldsymbol{\Theta}^2)$$

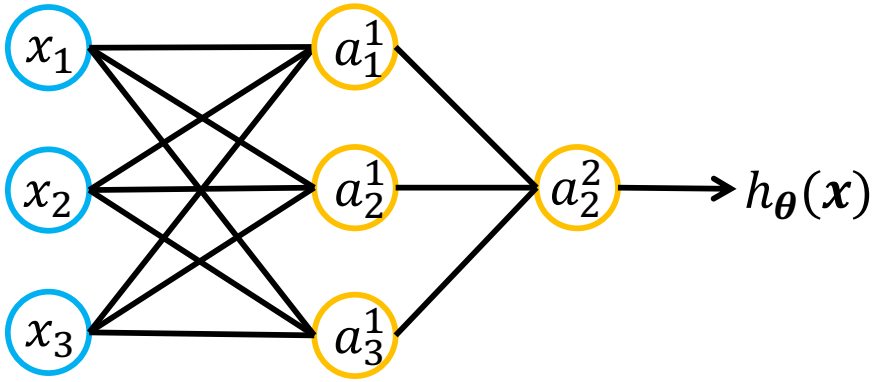
$$a_1^1 = f_a(\theta_{0,0}^1 + \theta_{1,0}^1 x_1 + \theta_{2,0}^1 x_2 + \theta_{3,0}^1 x_3)$$

$$a_2^2 = f_a(\theta_{0,1}^1 + \theta_{1,1}^1 x_1 + \theta_{2,1}^1 x_2 + \theta_{3,1}^1 x_3)$$

$$a_3^1 = f_a(\theta_{0,2}^1 + \theta_{1,2}^1 x_1 + \theta_{2,2}^1 x_2 + \theta_{3,2}^1 x_3)$$

$$h_{\boldsymbol{\theta}} = a_1^2 = f_a(\theta_{0,0}^2 + \theta_{1,0}^2 a_1 + \theta_{2,0}^2 a_2 + \theta_{3,0}^3 a_3)$$

Neural networks (or multilayer perceptron)



First layer is multiple logistic regressions with the input features

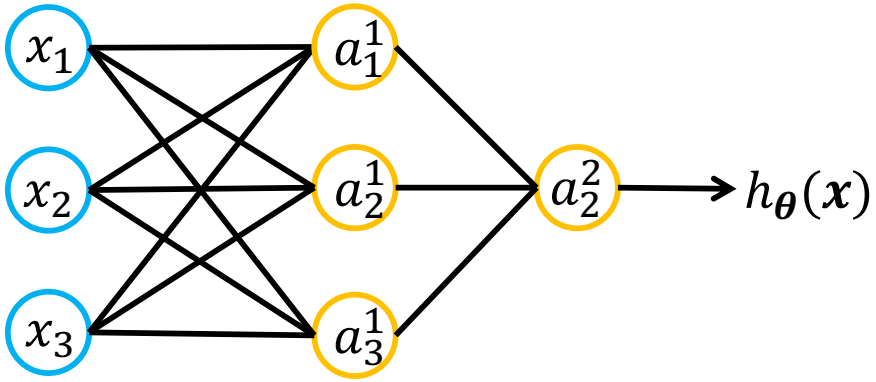
$$a_1^1 = f_a\left(\theta_{0,0}^1 + \theta_{1,0}^1 x_1 + \theta_{2,0}^1 x_2 + \theta_{3,0}^1 x_3\right) = \frac{1}{1 + e^{-(\theta_{0,0}^1 + \theta_{1,0}^1 x_1 + \theta_{2,0}^1 x_2 + \theta_{3,0}^1 x_3)}}$$

$$a_2^2 = f_a\left(\theta_{0,1}^1 + \theta_{1,1}^1 x_1 + \theta_{2,1}^1 x_2 + \theta_{3,1}^1 x_3\right) = \frac{1}{1 + e^{-(\theta_{0,1}^1 + \theta_{1,1}^1 x_1 + \theta_{2,1}^1 x_2 + \theta_{3,1}^1 x_3)}}$$

$$a_3^1 = f_a\left(\theta_{0,2}^1 + \theta_{1,2}^1 x_1 + \theta_{2,2}^1 x_2 + \theta_{3,2}^1 x_3\right) = \frac{1}{1 + e^{-(\theta_{0,2}^1 + \theta_{1,2}^1 x_1 + \theta_{2,2}^1 x_2 + \theta_{3,2}^1 x_2)}}$$

$$h_{\boldsymbol{\theta}} = a_1^2 = f_a(\theta_{0,0}^2 + \theta_{1,0}^2 a_1 + \theta_{2,0}^2 a_2 + \theta_{3,0}^3 a_3)$$

Neural networks (or multilayer perceptron)



Last layer is logistic regression with output of the previous layer as input (this allows the network to learn what features to use)

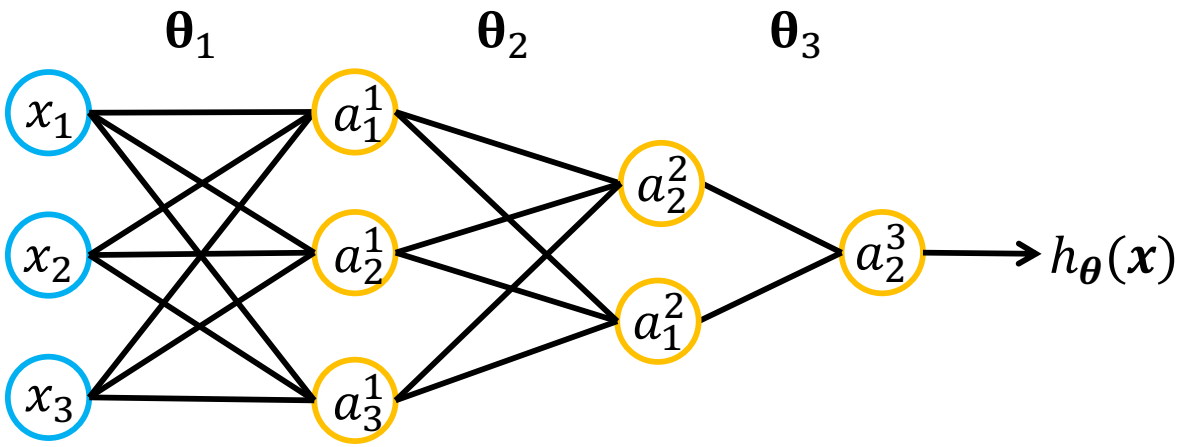
$$a_1^1 = f_a\left(\theta_{0,0}^1 + \theta_{1,0}^1 x_1 + \theta_{2,0}^1 x_2 + \theta_{3,0}^1 x_3\right)$$

$$a_2^2 = f_a\left(\theta_{0,1}^1 + \theta_{1,1}^1 x_1 + \theta_{2,1}^1 x_2 + \theta_{3,1}^1 x_3\right)$$

$$a_3^1 = f_a\left(\theta_{0,2}^1 + \theta_{1,2}^1 x_1 + \theta_{2,2}^1 x_2 + \theta_{3,2}^1 x_3\right)$$

$$h_{\boldsymbol{\theta}} = a_1^2 = f_a(\theta_{0,0}^2 + \theta_{1,0}^2 a_1 + \theta_{2,0}^2 a_2 + \theta_{3,0}^3 a_3) = \frac{1}{1 + e^{-(\theta_{0,0}^2 + \theta_{1,0}^2 a_1 + \theta_{2,0}^2 a_2 + \theta_{3,0}^3 a_3)}}$$

Neural networks (or multilayer perceptron)



$$\boldsymbol{a}^0 = \boldsymbol{x}^T$$

$$\boldsymbol{a}^1 = f_a(\boldsymbol{a}^0 \cdot \boldsymbol{\Theta}^1)$$

$$\boldsymbol{a}^2 = f_a(\boldsymbol{a}^1 \cdot \boldsymbol{\Theta}^2) = f_a\left(f_a(\boldsymbol{a}^0 \cdot \boldsymbol{\Theta}^1) \cdot \boldsymbol{\Theta}^2\right)$$

$$\boldsymbol{a}^3 = f_a(\mathbf{a}^2 \cdot \boldsymbol{\Theta}^3) = f_a\left(f_a(\boldsymbol{a}^1 \cdot \boldsymbol{\Theta}^2) \cdot \boldsymbol{\Theta}^3\right) = f_a\left(f_a\left(f_a\left(\boldsymbol{a}^0 \cdot \boldsymbol{\Theta}^1\right) \cdot \boldsymbol{\Theta}^2\right) \cdot \boldsymbol{\Theta}^3\right)$$

Neural networks with K classes

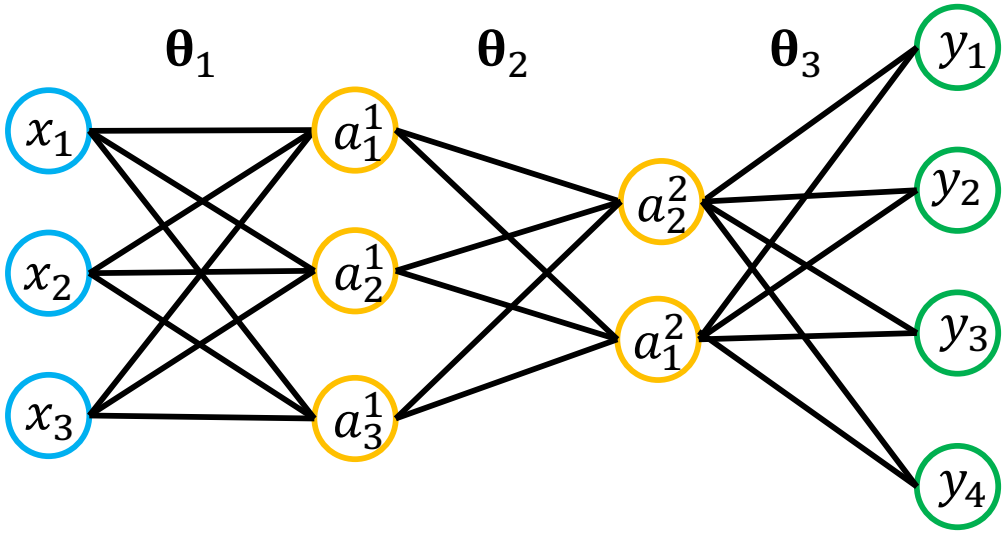
- So far we have considered neural networks with one output. This assumes binary labels described by $y \in \{0, 1\}$
- For K classes we can use a vector $\boldsymbol{y} = (y_1, \ldots, y_K)$:

$$y_k = \begin{cases} 1 & \text{if } k \text{ is the index of the true class} \\ 0 & \text{otherwise} \end{cases}$$

- Also called one-hot encoding (only one element is $\neq 0$)
- Classifier output can represent class probabilities

Neural networks with K classes

- Example with $K = 4$



Neural networks with K classes

Use of softmax activation function

- The softmax activation function rescales a vector $\boldsymbol{a}$ using

$$\hat{y}_k = \frac{\exp(a_k)}{\sum_{j=k}^{K} \exp(a_j)}$$

- The output $\hat{\boldsymbol{y}}$ has two properties
  1. $\sum_{k=1}^{K} \hat{y}_k = 1$
  2. $\hat{y}_k \geq 0 \quad \forall \hat{y}_k \in \hat{\boldsymbol{y}}$

Neural networks with K classes: Loss function

Use of cross entropy

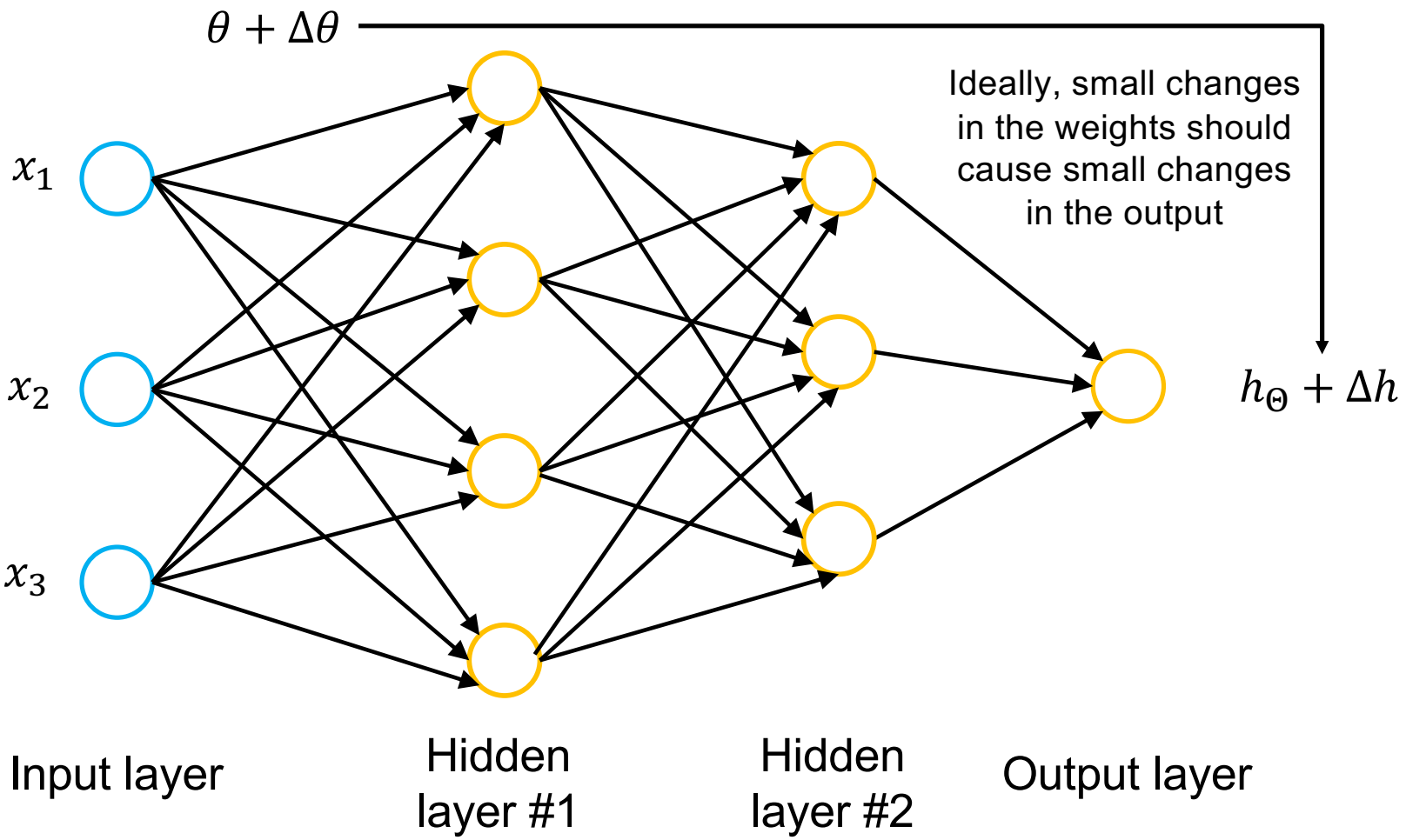
- In general, the cross entropy H of probability distributions p and q

$$H(p,q) = -\sum_{k=1}^{K} p_k \log(q_k)$$

- Assuming $p = y$ and $q = \hat{y}$ we can formulate a loss $\mathcal{L}$:

$$\mathcal{L}(y,\hat{y}) = -\log(\hat{y}_k)\Big|_{y_k=1} = -\log\left(\frac{\exp(a_k)}{\sum_{j=0}^{K}\exp(a_j)}\right)\Bigg|_{y_k=1}$$

Learning with neural networks



Learning with neural networks

However, small changes of weights $\mathbf{w}$ can lead to large changes in the output

Learning with neural networks

Recap:

- Loss function (assuming L_2 loss):

$$\mathcal{L} = \|\boldsymbol{y} - \hat{\boldsymbol{y}}\|_2^2 = \frac{1}{N}\sum_{\boldsymbol{i}} \left((y_i - h_\Theta(\boldsymbol{x_i})\right)^2$$

Number of examples in batch

$\mathcal{L}(\boldsymbol{x}_i, y_i, \Theta)$

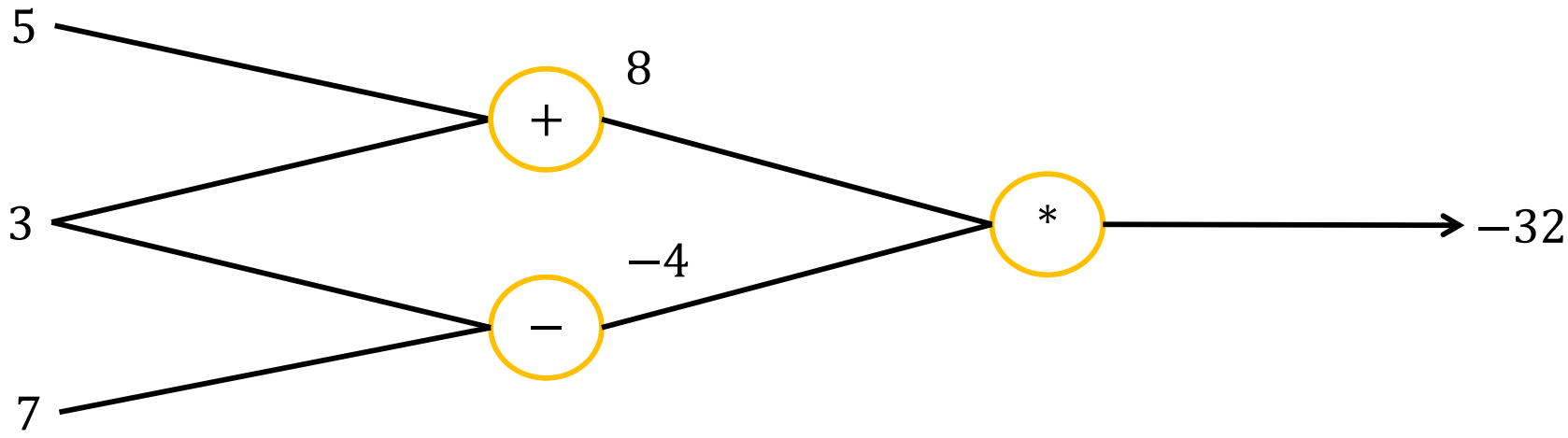
- Optimization:

```
repeat until convergence
```

$$\Theta^{k+1} := \Theta^k - \alpha \, \nabla \mathcal{L}(\Theta)$$

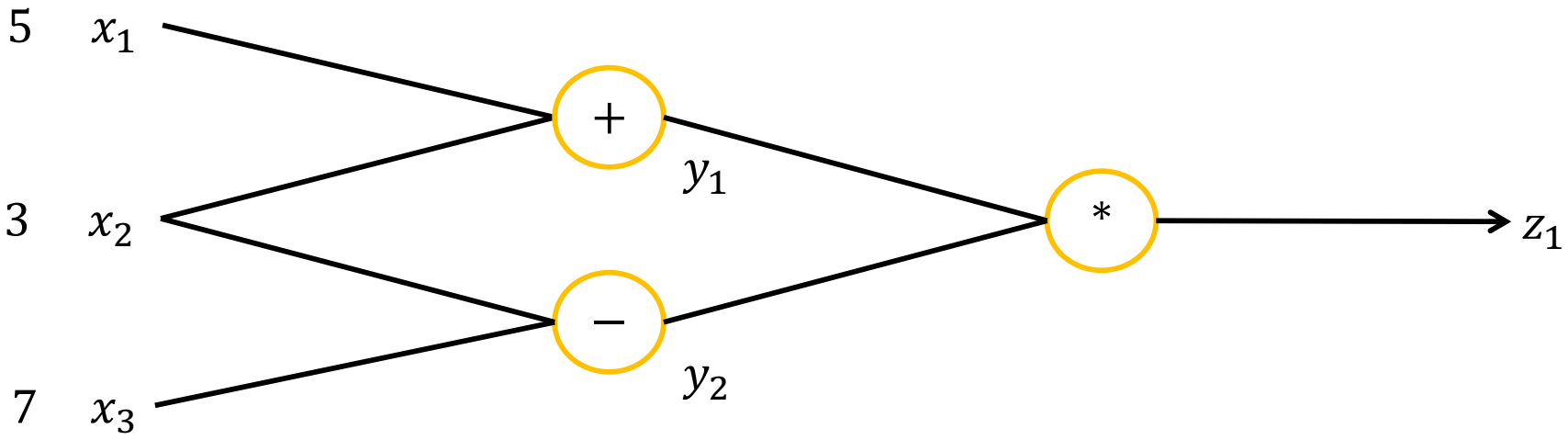
$$\nabla \mathcal{L}(\Theta) = \sum_i \nabla \mathcal{L}(\boldsymbol{x}_i, y_i, \Theta)$$

Computational graphs



$$-32 = 8 \times -4 = 5 + 3 + 3 - 7$$

Computational graphs



$$z_1 = y_1 \times y_2 = (x_1 + x_2) \times (x_2 - x_3) = x_1 x_2 - x_1 x_3 + x_2^2 - x_2 x_3$$

$$\frac{\partial z_1}{\partial x_1} = x_2 - x_3 = -4 \qquad \frac{\partial z_1}{\partial x_2} = x_1 + 2x_2 - x_3 = 4 \qquad \frac{\partial z_1}{\partial x_3} = -x_1 - x_2 = 8$$

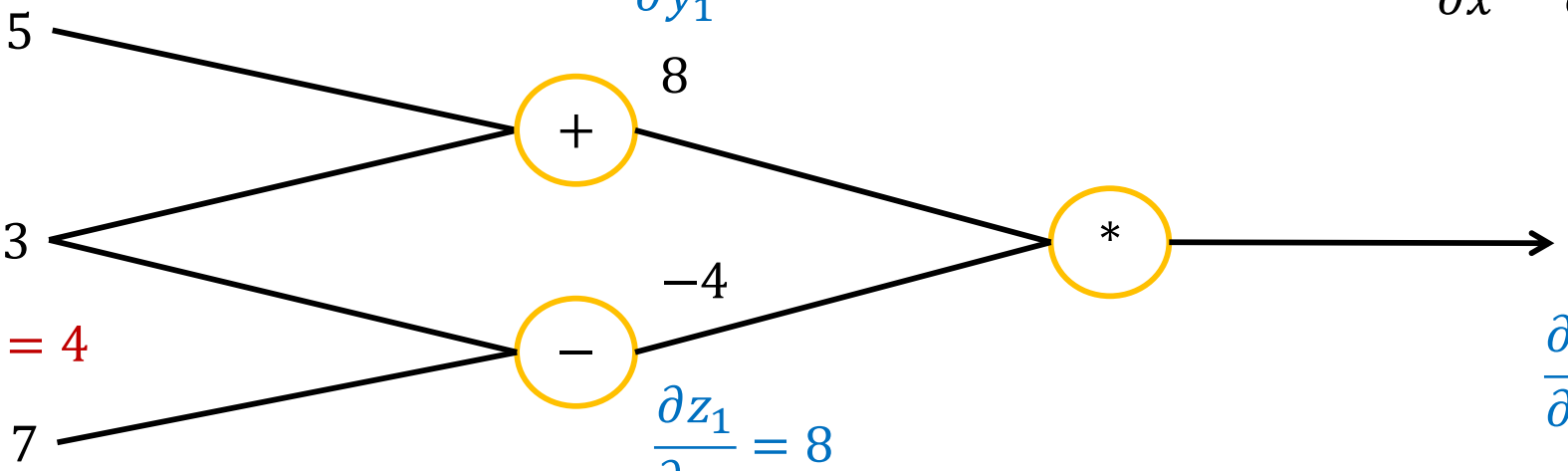
Computational graphs

$$\frac{\partial z_1}{\partial x_1} = \frac{\partial z_1}{\partial y_1}\frac{\partial y_1}{\partial x_1} = -4$$

$$\frac{\partial z_1}{\partial y_1} = -4$$

Chain rule: $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$

$$\frac{\partial z_1}{\partial x_2} = \frac{\partial z_1}{\partial y_1}\frac{\partial y_1}{\partial x_2} + \frac{\partial z_1}{\partial y_2}\frac{\partial y_2}{\partial x_2} = 4$$



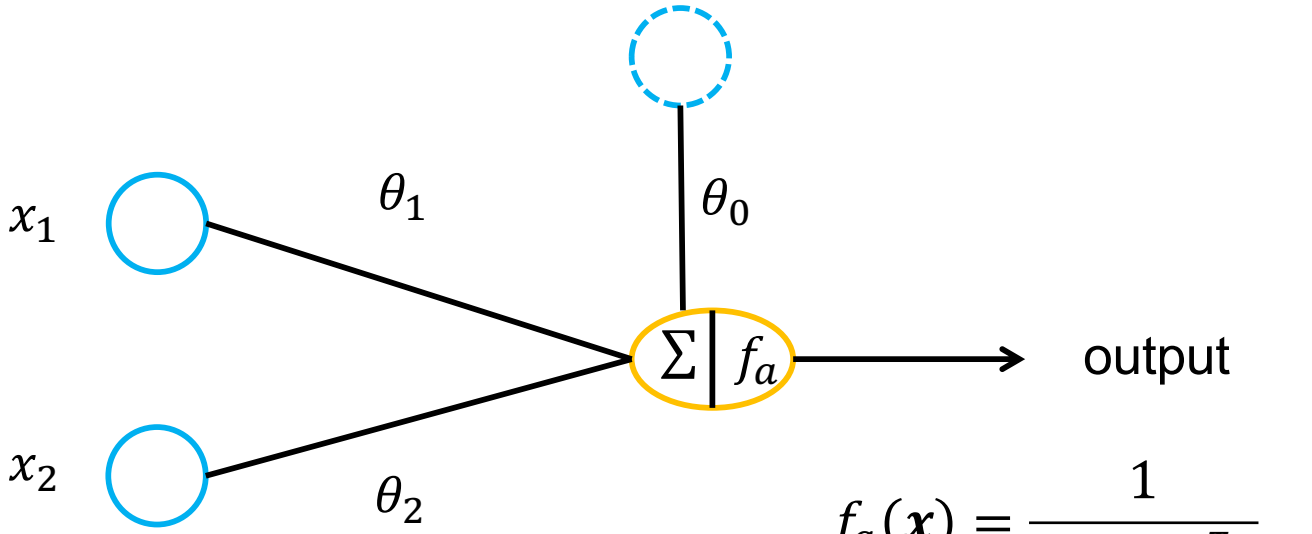
$$\frac{\partial z_1}{\partial z_1} = 1$$

$$\frac{\partial z_1}{\partial y_2} = 8$$

$$\frac{\partial z_1}{\partial x_3} = \frac{\partial z_1}{\partial y_2}\frac{\partial y_2}{\partial x_3} = -8$$

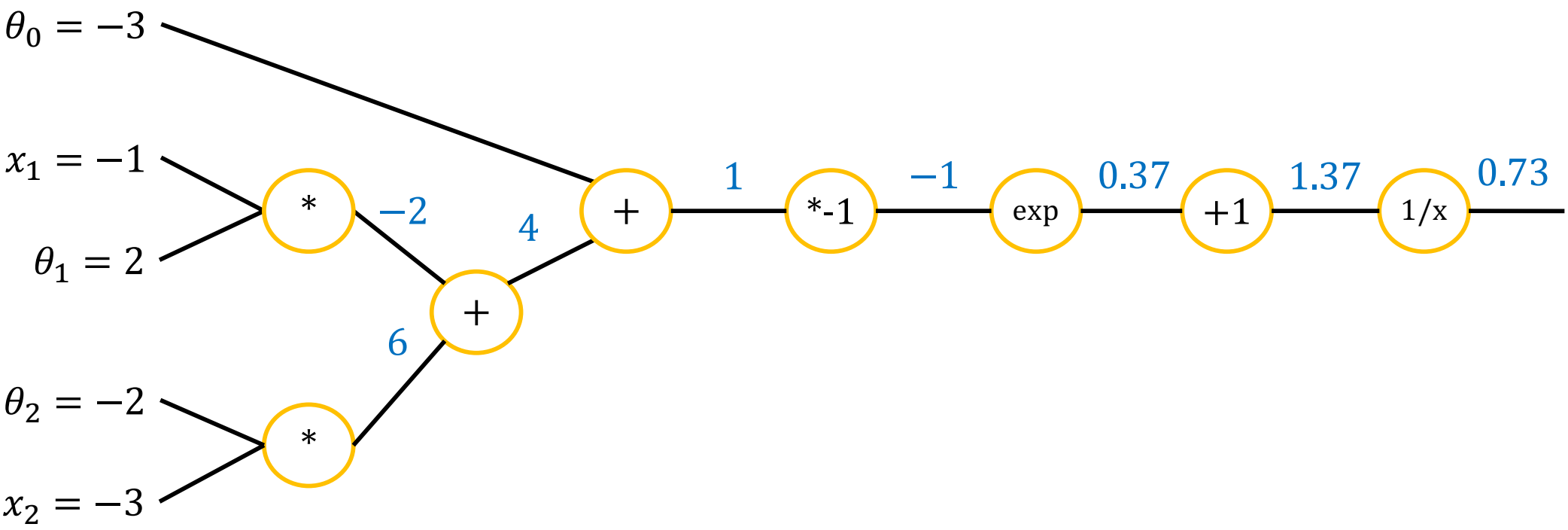
$$\frac{\partial y_1}{\partial x_1} = 1 \qquad \frac{\partial y_1}{\partial x_2} = 1 \qquad \frac{\partial y_2}{\partial x_2} = 1 \qquad \frac{\partial y_2}{\partial x_3} = -1$$

Computational graphs: Another example



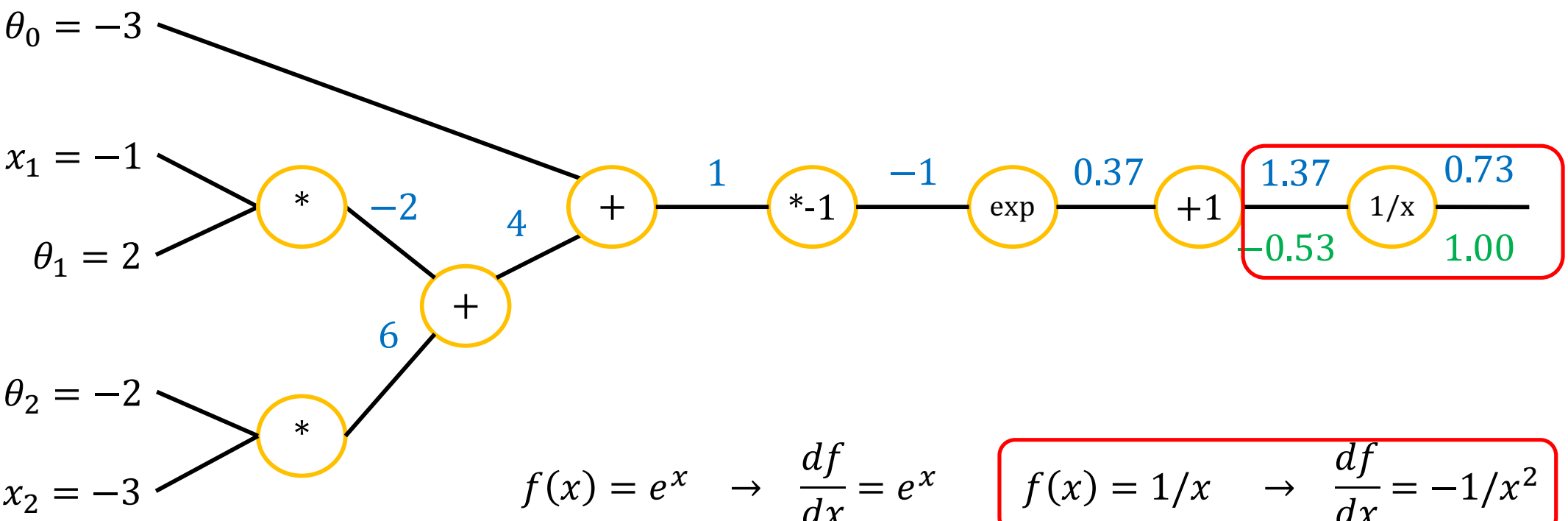
Slide adapted from Fei-Fei Li & Justin Johnson & Serena Yeung

Computational graphs



Slide adapted from Fei-Fei Li & Justin Johnson & Serena Yeung

Computational graphs



$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

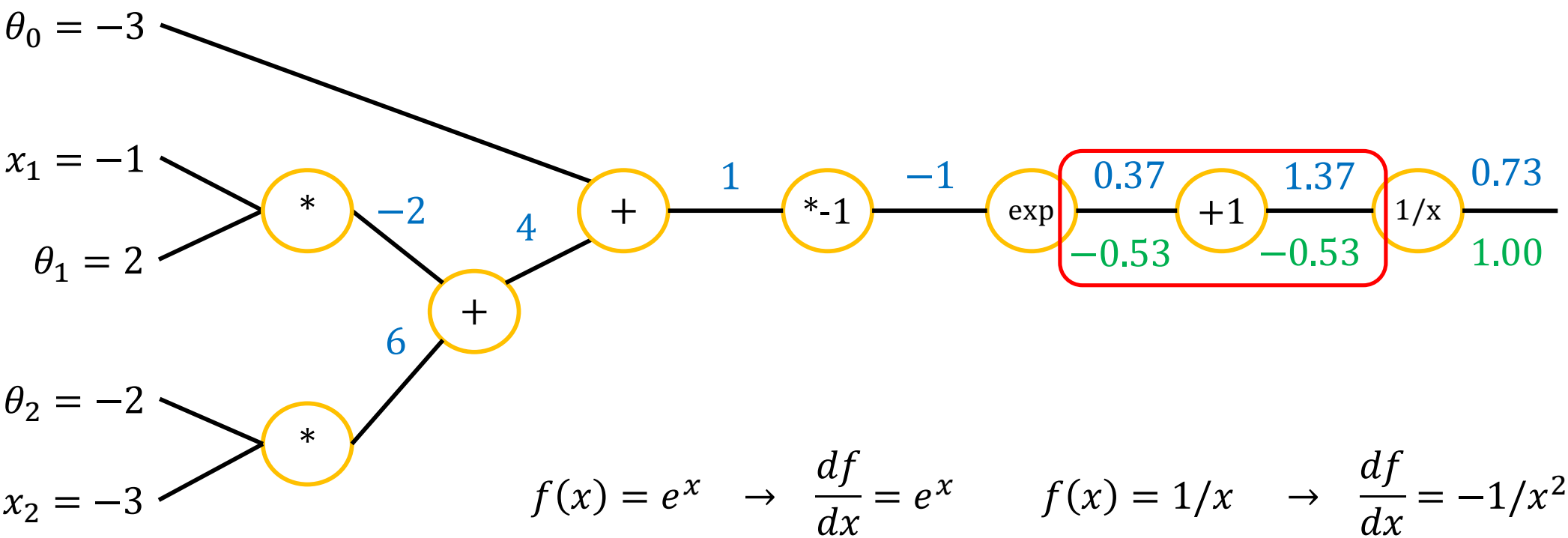
$$f(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$f(x) = 1/x \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f(x) = x + c \quad \rightarrow \quad \frac{df}{dx} = 1$$

Slide adapted from Fei-Fei Li & Justin Johnson & Serena Yeung

Computational graphs

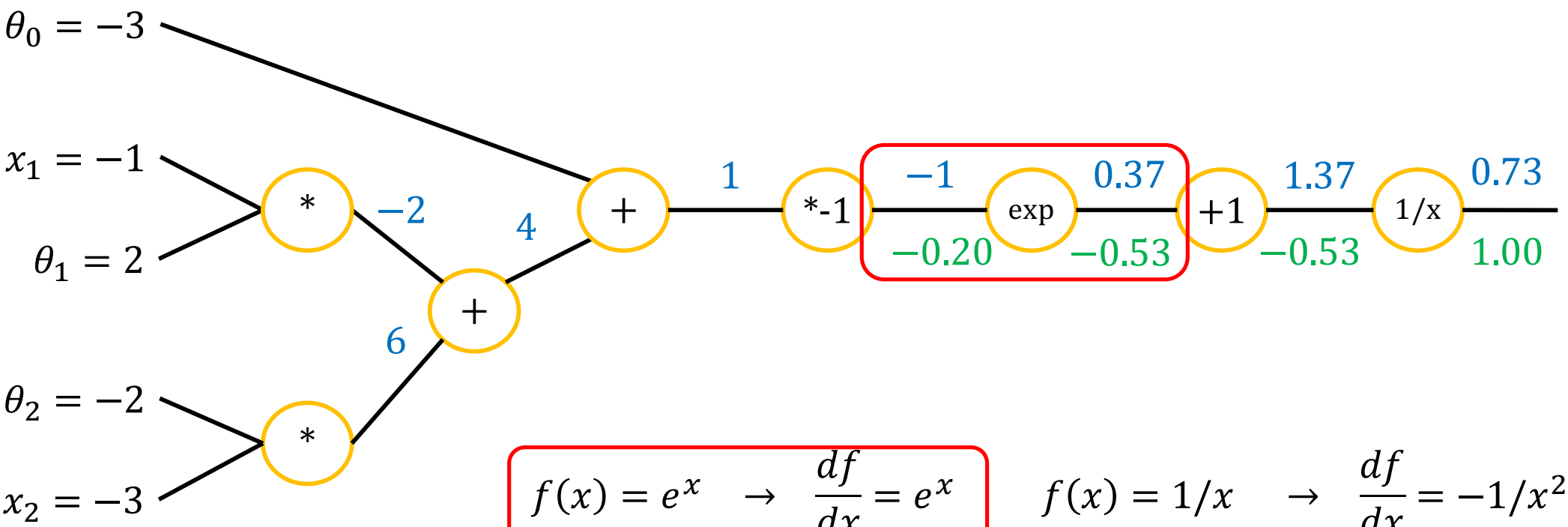


$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x \qquad f(x) = 1/x \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a \qquad f(x) = x + c \quad \rightarrow \quad \frac{df}{dx} = 1$$

Slide adapted from Fei-Fei Li & Justin Johnson & Serena Yeung

Computational graphs

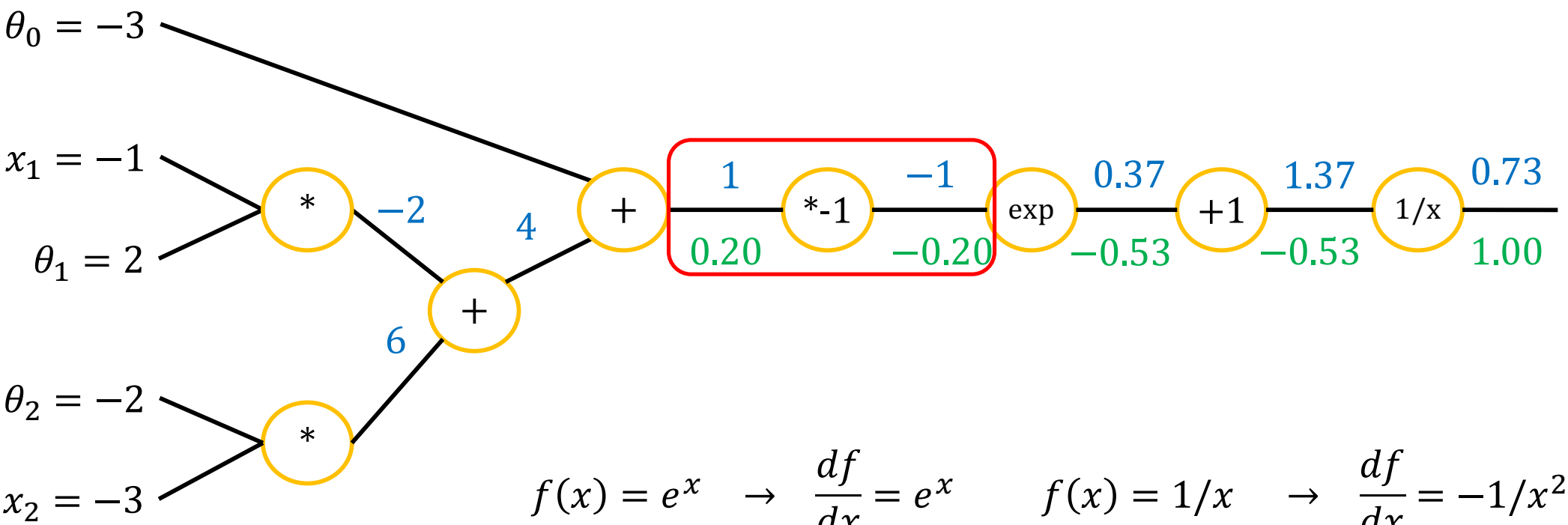


$$f(x) = e^x \rightarrow \frac{df}{dx} = e^x \qquad f(x) = 1/x \rightarrow \frac{df}{dx} = -1/x^2$$

$$f(x) = ax \rightarrow \frac{df}{dx} = a \qquad f(x) = x + c \rightarrow \frac{df}{dx} = 1$$

Slide adapted from Fei-Fei Li & Justin Johnson & Serena Yeung

Computational graphs

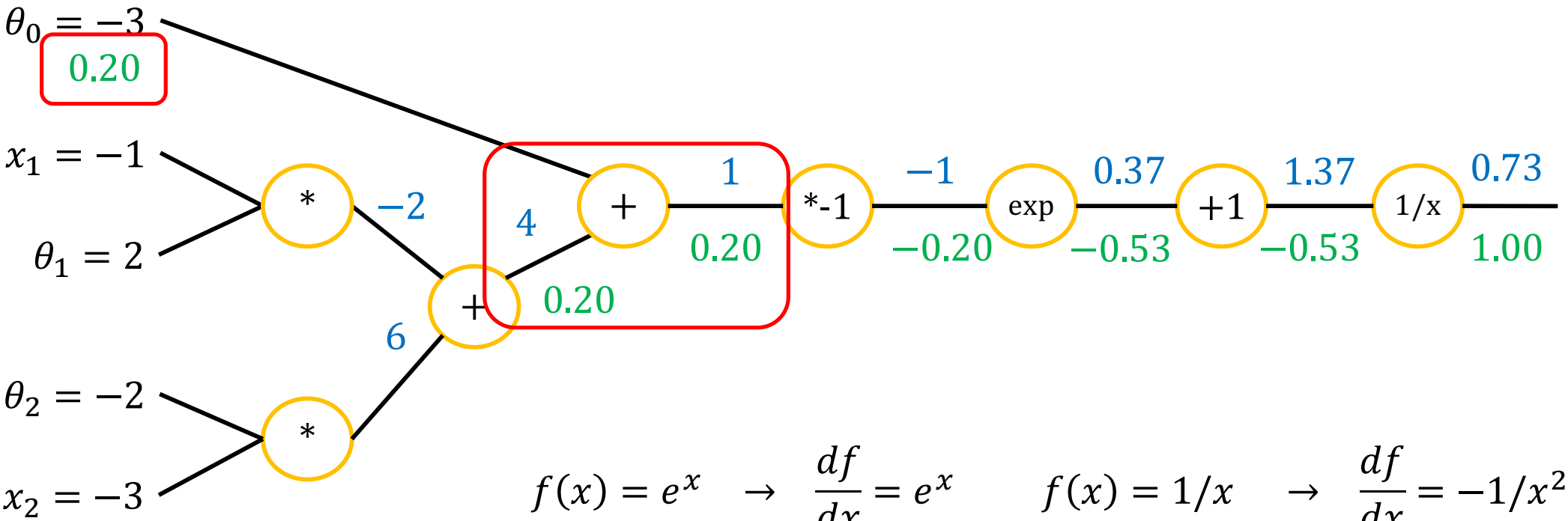


$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x \qquad f(x) = 1/x \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a \qquad f(x) = x + c \quad \rightarrow \quad \frac{df}{dx} = 1$$

Slide adapted from Fei-Fei Li & Justin Johnson & Serena Yeung

Computational graphs

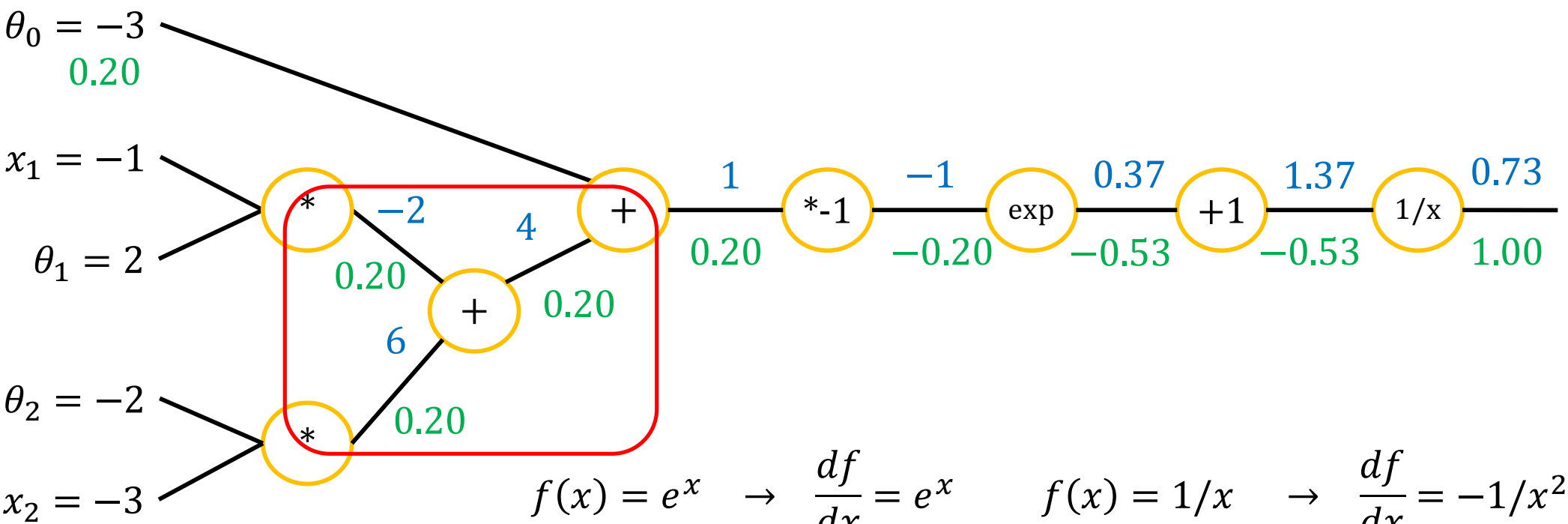


$$f(x) = e^x \rightarrow \frac{df}{dx} = e^x \qquad f(x) = 1/x \rightarrow \frac{df}{dx} = -1/x^2$$

$$f(x) = ax \rightarrow \frac{df}{dx} = a \qquad f(x) = x + c \rightarrow \frac{df}{dx} = 1$$

Slide adapted from Fei-Fei Li & Justin Johnson & Serena Yeung

Computational graphs

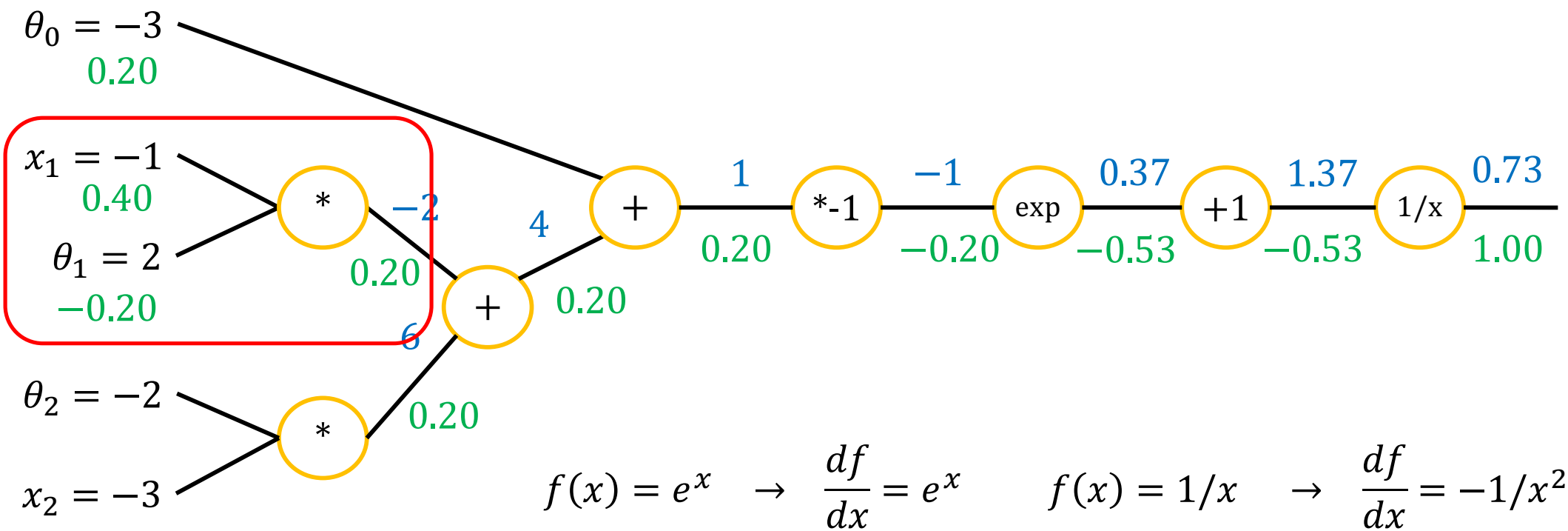


$$f(x) = e^x \rightarrow \frac{df}{dx} = e^x \qquad f(x) = 1/x \rightarrow \frac{df}{dx} = -1/x^2$$

$$f(x) = ax \rightarrow \frac{df}{dx} = a \qquad f(x) = x + c \rightarrow \frac{df}{dx} = 1$$

Slide adapted from Fei-Fei Li & Justin Johnson & Serena Yeung

Computational graphs

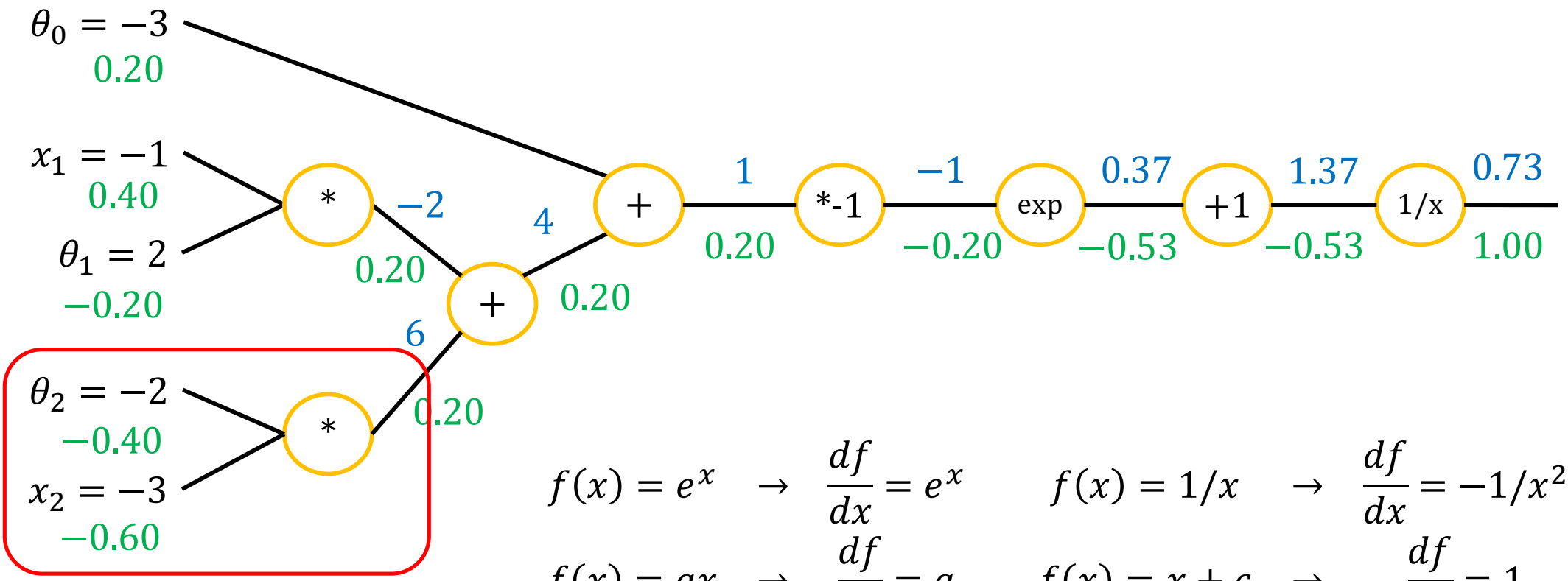


$$f(x) = e^x \rightarrow \frac{df}{dx} = e^x \qquad f(x) = 1/x \rightarrow \frac{df}{dx} = -1/x^2$$

$$f(x) = ax \rightarrow \frac{df}{dx} = a \qquad f(x) = x + c \rightarrow \frac{df}{dx} = 1$$

Slide adapted from Fei-Fei Li & Justin Johnson & Serena Yeung

Computational graphs



$$f(x) = e^x \rightarrow \frac{df}{dx} = e^x \qquad f(x) = 1/x \rightarrow \frac{df}{dx} = -1/x^2$$

$$f(x) = ax \rightarrow \frac{df}{dx} = a \qquad f(x) = x + c \rightarrow \frac{df}{dx} = 1$$

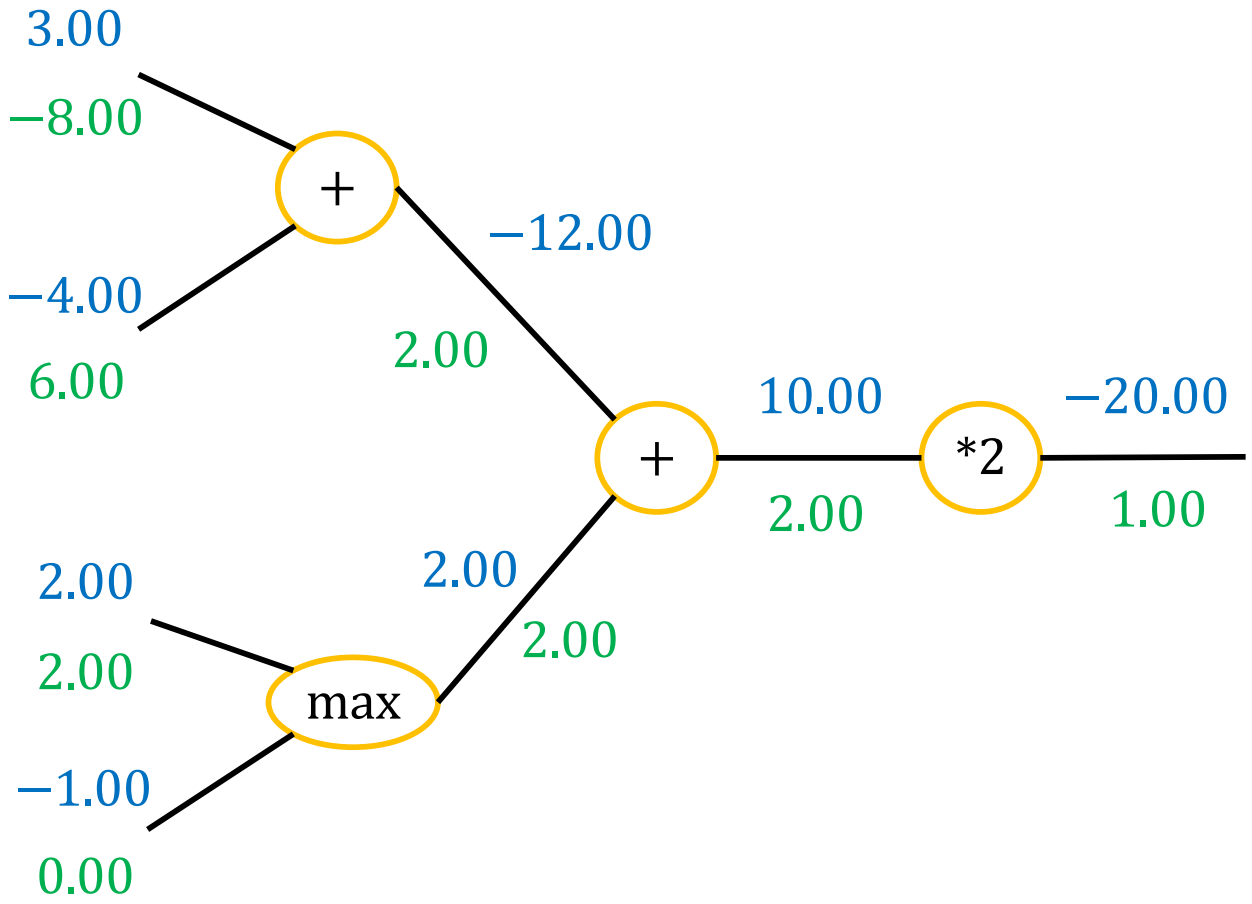
Slide adapted from Fei-Fei Li & Justin Johnson & Serena Yeung

Patterns in the backwards flow

add gate:
- gradient distributor

max gate:
- gradient router

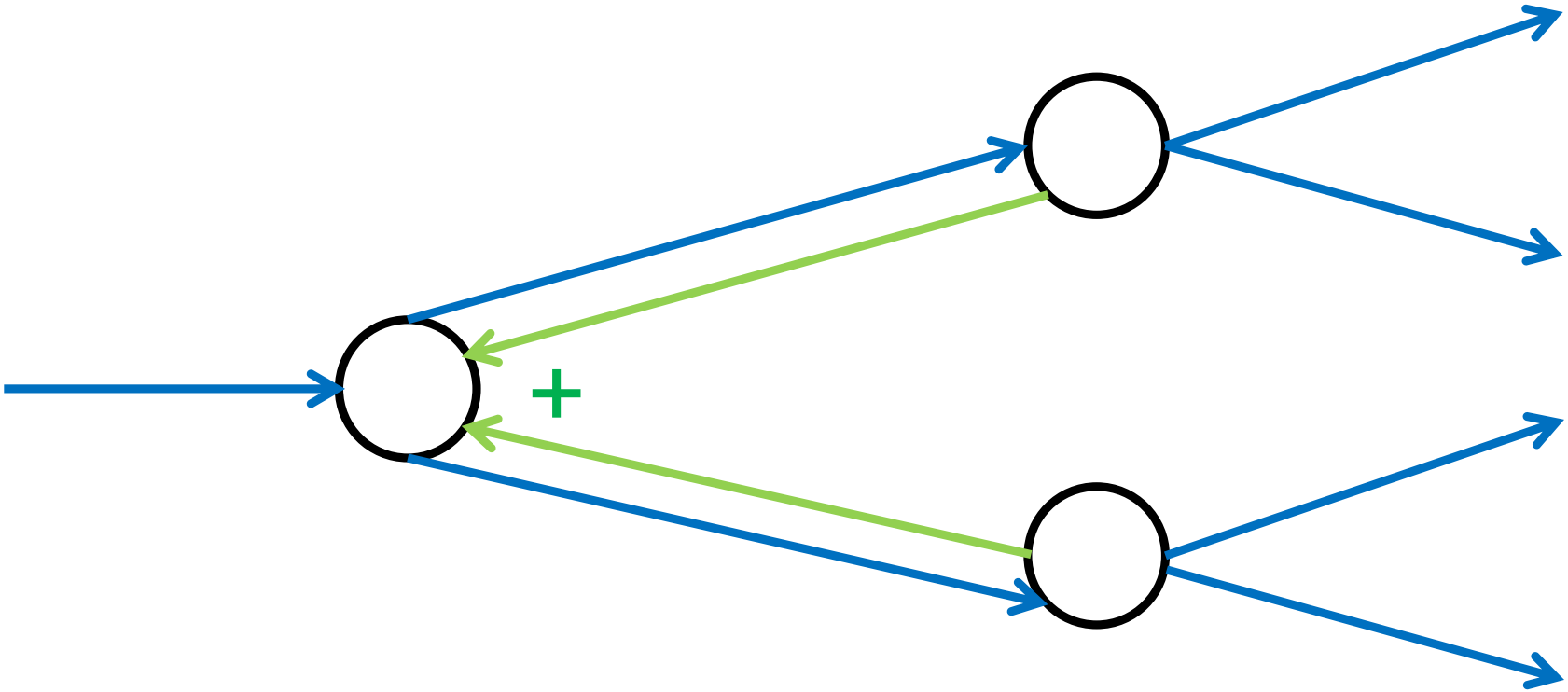
mul gate:
- gradient switcher



Slide adapted from Fei-Fei Li & Justin Johnson & Serena Yeung

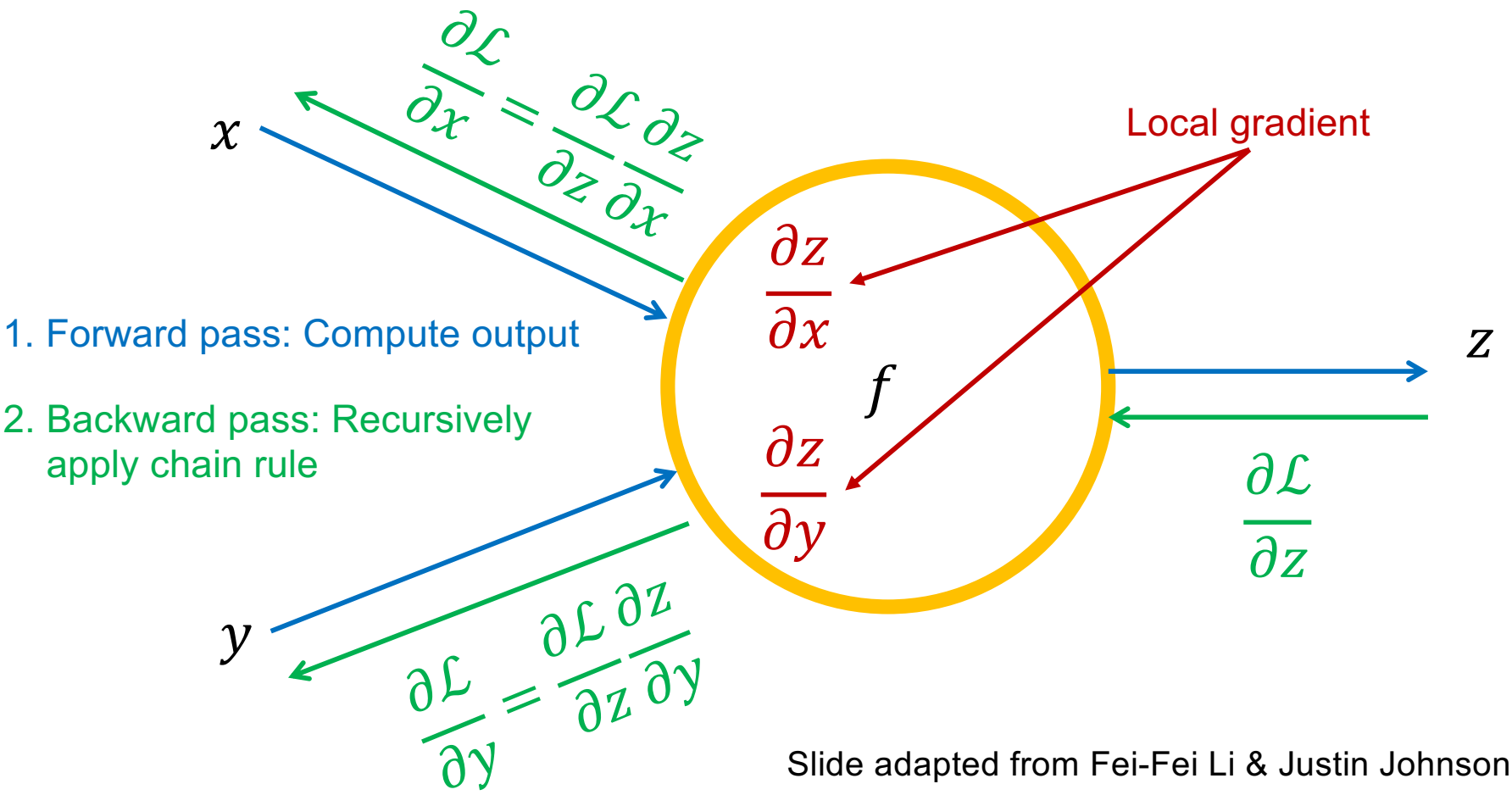
Patterns in the backwards flow

Gradients add at branches:



Slide adapted from Fei-Fei Li & Justin Johnson & Serena Yeung

Back-propagation



Slide adapted from Fei-Fei Li & Justin Johnson & Serena Yeung

Back-propagation

- Generic optimization algorithm:
  1. Forward pass: Make a prediction and measure the error
  2. Backward pass: Go through each layer in reverse to measure the error contribution from each connection (gradient calculation step)
  3. Adjust connection weights to reduce the error (gradient descent step)

Back-propagation algorithm

Neural networks: Activation function

- Each node in the neural network passes its computation through activation function f_a.
  - f_a defines the output of that node, or "neuron," given an input or set of inputs.
  - f_a can be regarded as a transfer function
  - f_a should be a (piecewise) linear and nonlinear activation functions (why?)
- Typically, all nodes in the neural network have the same activation function for each class of neurons **but**
  – activation functions for hidden units and output nodes can be different
- In biologically inspired neural networks, the activation function is usually an abstraction:
  – It represents the rate of action potential firing in the cell.
  – In its simplest form, this function is binary: Either the neuron is firing or not.

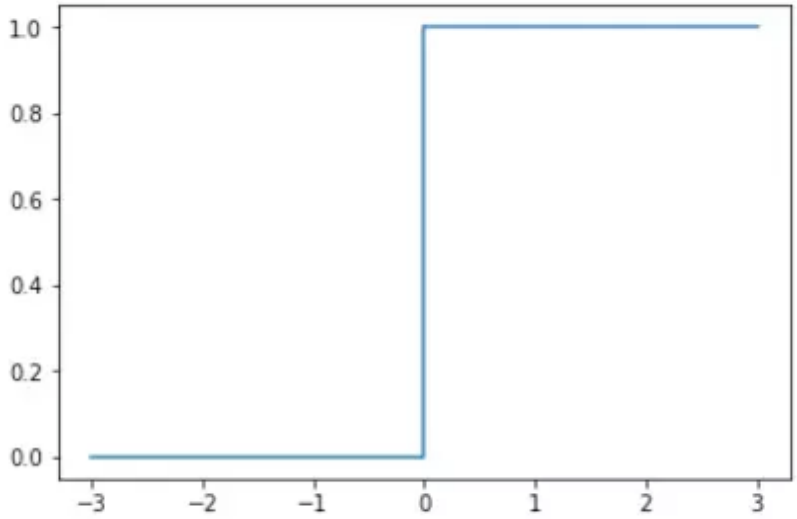
Neural networks: Activation function

- Heaviside function
  – Thresholds input at 0

$$f_a(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

- Advantages
  – Simple

- Disadvantages
  – Not differentiable

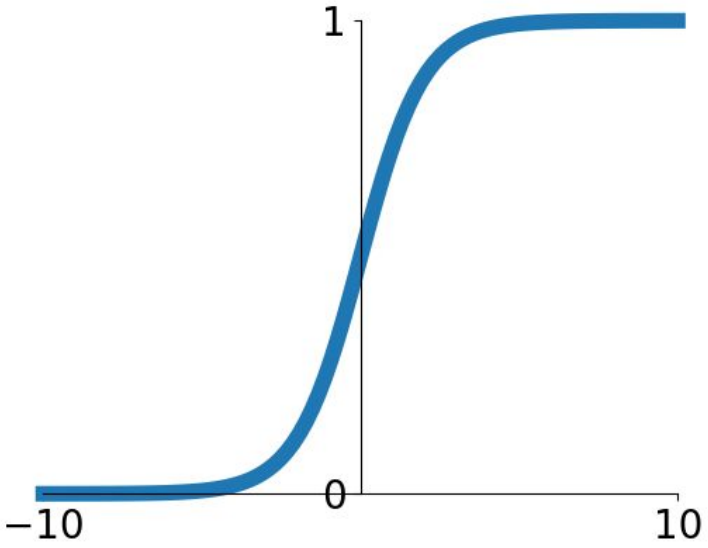


Neural networks: Activation function

- Sigmoid function
  - Squashes numbers to range [0,1]
- Advantages
  - Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron
- Disadvantages
  - Saturated neurons “kill” the gradients
  - Sigmoid outputs are not zero-centered
  - Exp is a compute expensive

$$f_a(x) = \frac{1}{1 + e^{-x}}$$

$$f_a'(x) = f_a(x)\big(1 - f_a(x)\big)$$



Neural networks: Activation function

- Tanh function
  - Scaled version of the sigmoid function:

$$\tanh(x) = 2\,\mathrm{sigmoid}(2x) - 1$$

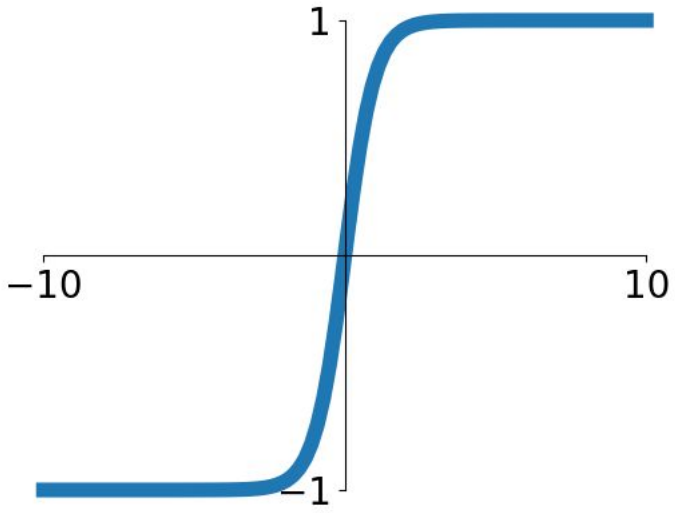
  - Squashes numbers to range [0,1]

- Advantages
  - Outputs are zero-centered

- Disadvantages
  - Saturated neurons “kill” the gradients
  - Tanh is a compute expensive

$$f_a(x) = \tanh x = \frac{2}{1 + e^{-2x}} - 1$$

$$f_a'(x) = 1 - f_a(x)^2$$



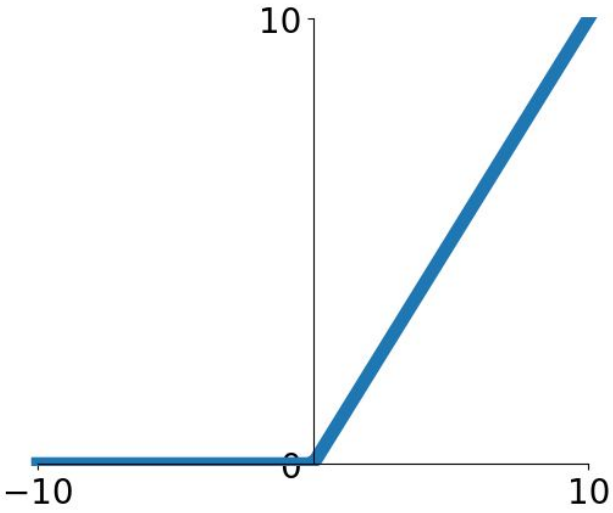
Neural networks: Activation function

- RELU function
  - Piecewise linear function

$$f_a(x) = \max(0, x)$$

$$f_a'(x) = \begin{cases} 0 \; for \; x < 0 \\ 1 \; for \; x \geq 0 \end{cases}$$

- Advantages
  - Does not saturate (in +region)
  - Very computationally efficient
  - Converges much faster than sigmoid/tanh in practice (e.g. 6x)
  - More biologically plausible than sigmoid



- Disadvantages
  - Not zero-centered output
  - Dead RELUs

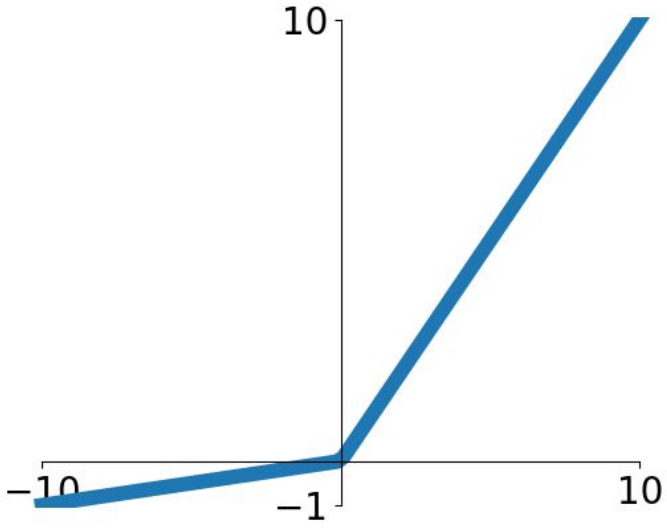
Neural networks: Activation function

- Leaky RELU function
  - Piecewise linear function

$$f_a(x) = \max(cx, x)$$

$$f_a'(x) = \begin{cases} c \; for \; x < 0 \\ 1 \; for \; x \geq 0 \end{cases}$$

- Advantages
  - Does not saturate
  - Computationally efficient
  - Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
  - Will not “die”



Parametric Rectifier (PReLU):

$f_a(x) = \max(cx, x)$

Can optimize c via back-propagation

Neural networks: Activation function

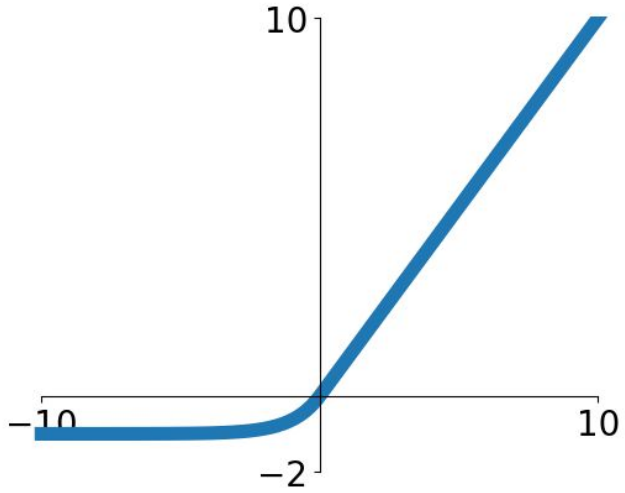
- Exponential Linear Units (ELU)

$$f_a(x) = \begin{cases} x & \text{if} \quad x > 0 \\ \alpha(\exp(x) - 1) & \text{if} \quad x \leq 0 \end{cases}$$

$$f_a'(x) = \begin{cases} 1 & \text{if} \quad x > 0 \\ f_a(x) + \alpha & \text{if} \quad x \leq 0 \end{cases}$$

- Advantages:
  - All benefits of ReLU
  - Closer to zero mean outputs
  - Negative saturation regime
  - Compared with Leaky ReLU adds some robustness to noise

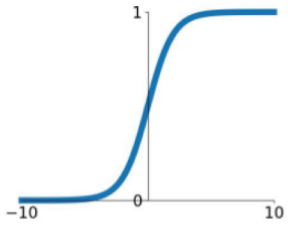
- Disadvantages
  - Compute expensive



Summary: Activation functions

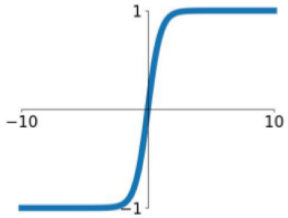
Sigmoid

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



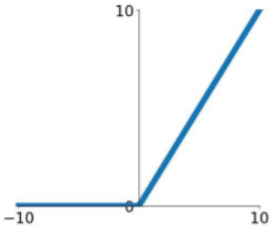
tanh

$$\tanh(x)$$



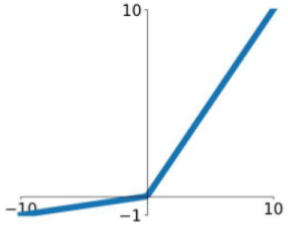
ReLU

$$\max(0, x)$$



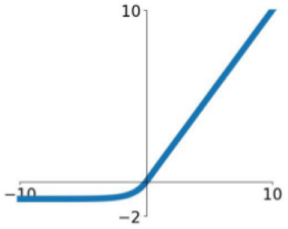
Leaky ReLU

$$\max(0.1x, x)$$



ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Optimizing neural networks: Stochastic gradient descent

- Recall that the loss function is:

$$\mathcal{L} = \sum_{i=1}^{N} \big((h_\Theta(\boldsymbol{x_i}), y_i \big)$$

- For datasets with large N (e.g. ImageNet has more than 14M annotated examples) it becomes unfeasible to frequently evaluate the loss function or its gradient.
- Instead we use stochastic gradient descent where the loss function and its gradient is approximated from minibatch of n examples from the training set (with $n \ll N$)

Optimizing neural networks: Stochastic gradient descent

Initialize parameters Θ and learning rate τ

Repeat

Sample a minibatch of n examples $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ with labels $\{y_1, \ldots, y_n\}$ from the training set

Compute gradient estimate: $\widehat{\nabla}_\Theta = \frac{1}{n} \nabla_\Theta \sum_{i=1}^{n} \mathcal{L}(h_\Theta(\boldsymbol{x_i}), y_i)$

Apply update: $\Theta \coloneqq \Theta - \tau \widehat{\nabla}_\Theta$

Until convergence

$\tau_k = (1 - \alpha)\tau_0 + \alpha \tau_{k_{max}}$ with $\alpha = \frac{k}{k_{max}}$

After iteration k_{max} τ_k is left unchanged

Optimizing neural networks: Stochastic gradient descent

Size of minibatch depends on several factors:

- Large batches provide a more accurate estimate of the gradient but with less than linear returns. This is because the standard error of the mean from n samples is given by $\sigma/\sqrt{n}$. For example, consider an estimate of the mean from $n = 100$ samples and another one from $n = 10{,}000$ samples. The latter requires 100 times more computation but is only 10 times more accurate.
- Multicore computing architectures are underutilized by extremely small batch sizes. This motivates an absolute minimum batch size.

Optimizing neural networks: Stochastic gradient descent

Size of minibatch depends on several factors (cont.):

- Memory utilization typically scales linearly with the batch size. On GPU architectures with limited memory this typically constrains the maximum batch size
- Small batch sizes can have the effect of regularization (more about regularization later) as it adds noise to the learning process. (The generalisation ability is often best with a batch size of 1, but small batch sizes require small learning rates and can lead to slow convergence).

Optimizing neural networks: Stochastic gradient descent with momentum

Initialize parameters Θ, $\boldsymbol{v}$, learning rate τ and momentum parameter φ

Repeat

Sample a minibatch of n examples $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ with labels $\{y_1, \ldots, y_n\}$ from the training set

Compute gradient estimate: $\widehat{\nabla}_\Theta = \frac{1}{n} \nabla_\Theta \sum_{i=1}^{n} \mathcal{L}(h_\Theta(\boldsymbol{x}_i), y_i)$

Compute velocity update: $\boldsymbol{v} \coloneqq \varphi \boldsymbol{v} - \tau \widehat{\nabla}_\Theta$

Apply update: $\Theta \coloneqq \Theta + \boldsymbol{v}$

until convergence

Optimizing neural networks: Stochastic gradient descent with Nesterov momentum

Initialize parameters Θ, $\boldsymbol{v}$, learning rate τ and momentum parameter φ

Repeat

Sample a minibatch of n examples $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ with labels $\{y_1, \ldots, y_n\}$ from the training set

Apply interim update $\widetilde{\Theta} \coloneqq \Theta + \varphi \boldsymbol{v}$

Compute gradient estimate: $\widehat{\nabla}_{\widetilde{\Theta}} = \frac{1}{n} \nabla_{\widetilde{\Theta}} \sum_{i=1}^{n} \mathcal{L}(h_{\widetilde{\Theta}}(\boldsymbol{x}_i), y_i)$

Compute velocity update: $\boldsymbol{v} \coloneqq \varphi \boldsymbol{v} - \tau \widehat{\nabla}_{\widetilde{\Theta}}$

Apply update: $\Theta \coloneqq \Theta + \boldsymbol{v}$

until convergence

Optimizing neural networks: AdaGrad

Initialize parameters Θ, learning rate τ, small constant δ and gradient accumulation vector $\boldsymbol{r} = 0$

Repeat

Sample a minibatch of n examples $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ with labels $\{y_1, \ldots, y_n\}$ from the training set

Compute gradient estimate: $\hat{\nabla}_\Theta = \frac{1}{n} \nabla_\Theta \sum_{i=1}^{n} \mathcal{L}(h_\Theta(\boldsymbol{x}_i), y_i)$

Compute $\boldsymbol{r} \coloneqq \boldsymbol{r} + \hat{\nabla}_\Theta \odot \hat{\nabla}_\Theta$

Apply update: $\Theta \coloneqq \Theta - \boxed{\frac{\tau}{\delta + \sqrt{\boldsymbol{r}}}} \odot \hat{\nabla}_\Theta$

until convergence

Division and square root applied element-wise

Optimizing neural networks: RMSProp

Initialize parameters Θ, learning rate τ, decay rate ρ, small constant δ and gradient accumulation vector $\boldsymbol{r} = 0$

Repeat

Sample a minibatch of n examples $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ with labels $\{y_1, \ldots, y_n\}$ from the training set

Compute gradient estimate: $\hat{\nabla}_\Theta = \frac{1}{n} \nabla_\Theta \sum_{i=1}^{n} \mathcal{L}(h_\Theta(\boldsymbol{x}_i), y_i)$

Compute $\boldsymbol{r} \coloneqq \rho \boldsymbol{r} + (1 - \rho) \hat{\nabla}_\Theta \odot \hat{\nabla}_\Theta$

Apply update: $\Theta \coloneqq \Theta - \boxed{\frac{\tau}{\delta + \sqrt{\boldsymbol{r}}}} \odot \hat{\nabla}_\Theta$

until convergence

Division and square root applied element-wise

Optimizing neural networks: Adaptive moments (Adam)

Initialize parameters Θ, learning rate τ, decay rates ρ_1 and ρ_2, small constant δ and $\boldsymbol{r}, \boldsymbol{s}, t = 0$

Repeat

Sample a minibatch of n examples $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ with labels $\{y_1, \ldots, y_n\}$ from the training set

Compute gradient estimate: $\widehat{\nabla}_\Theta = \frac{1}{n} \nabla_\Theta \sum_{i=1}^{n} \mathcal{L}(h_\Theta(\boldsymbol{x}_i), y_i)$

Compute $t \coloneqq t + 1$

Compute $\boldsymbol{s} \coloneqq \rho_1 \boldsymbol{s} + (1 - \rho_1)\widehat{\nabla}_\Theta$ and $\boldsymbol{r} \coloneqq \rho_2 \boldsymbol{r} + (1 - \rho_2)\widehat{\nabla}_\Theta \odot \widehat{\nabla}_\Theta$

Compute $\hat{\boldsymbol{s}} \coloneqq \frac{\boldsymbol{s}}{1-\rho_1^t}$ and $\hat{\boldsymbol{r}} \coloneqq \frac{\boldsymbol{r}}{1-\rho_2^t}$

Apply update: $\Theta \coloneqq \Theta - \tau \frac{\hat{\boldsymbol{s}}}{\delta + \sqrt{\hat{\boldsymbol{r}}}}$

until convergence

Learning with neural networks: Summary

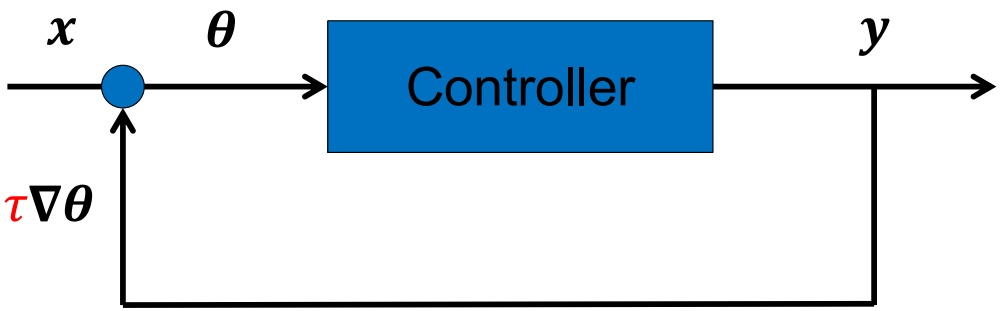
Back propagation

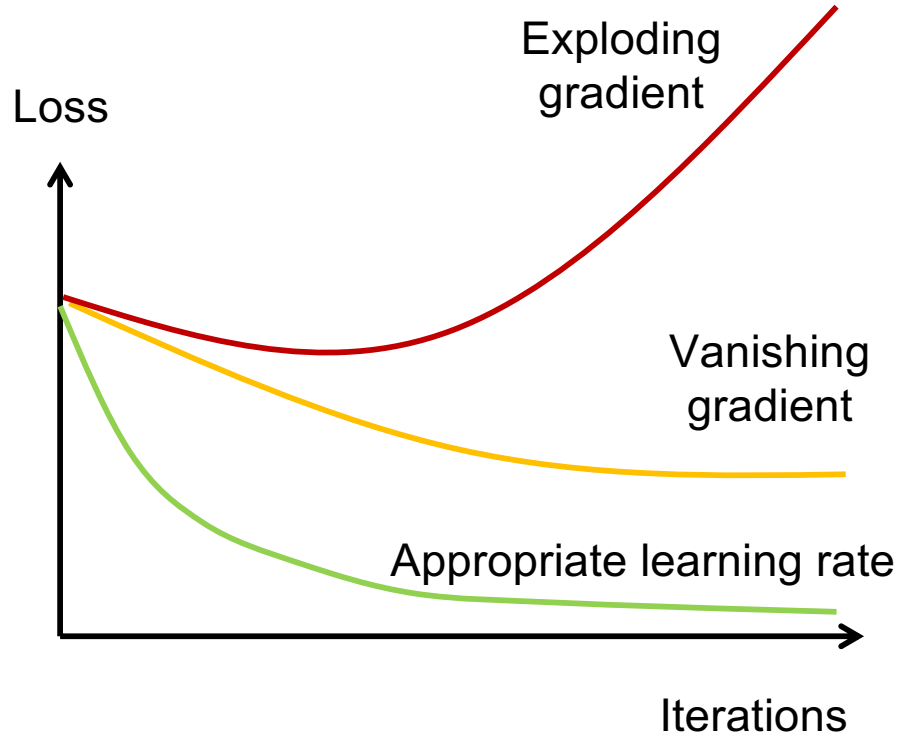
- Uses forward pass through the network (evaluation of the function h)
- Recursive application of chain rule
- Computationally very efficient if implemented correctly (**use vectorized forms!**)
- BP can be used with different gradient-based optimization schemes

Consequences:

- Product of partials → number errors multiply
- Product of partials → vanishing or exploding gradients

Learning with neural networks: Summary





- If τ is too high → positive feedback → loss grows without
- If τ is too small → negative feedback → gradient vanishes
- Choice of τ is **critical!**

Regularisation

Regularisation is useful in situations where the solution of the optimization problem is ill-posed:

- Overdetermined
- Underdetermined (more common in ML, leading to overfitting)

To avoid overfitting we can:

- Early stopping
- Adding L^p regularisation regularization
- Dropout
- Max-norm regularisation
- Data augmentation

Regularisation

Early stopping:

- Interrupt training when its performance on the validation set starts dropping
- Requires during training to evaluate the model on the validation set at regular intervals (say every 50 steps) and save snapshot result for next comparison
- Works best if combined with other regularization methods

Regularisation: L^2 or L^1

L^2 Regularisation

- Also known as ridge regression or Tikhonov regularisation
- Update cost function by adding another term (regularization term):

$$\tilde{\mathcal{L}} = \mathcal{L}_\Theta + \lambda \|\Theta\|_2^2$$

L^1 Regularisation

- Also known as Lasso regression
- Update cost function by adding another term (regularization term):

$$\tilde{\mathcal{L}} = \mathcal{L}_\Theta + \lambda \|\Theta\|_1$$

Regularisation: Max-norm

- For each neuron, the weights $\boldsymbol{\Theta}$ of the incoming connections are constrained such that

$$\|\Theta\|_2 \leq r$$

  – r is the max-norm hyperparameter

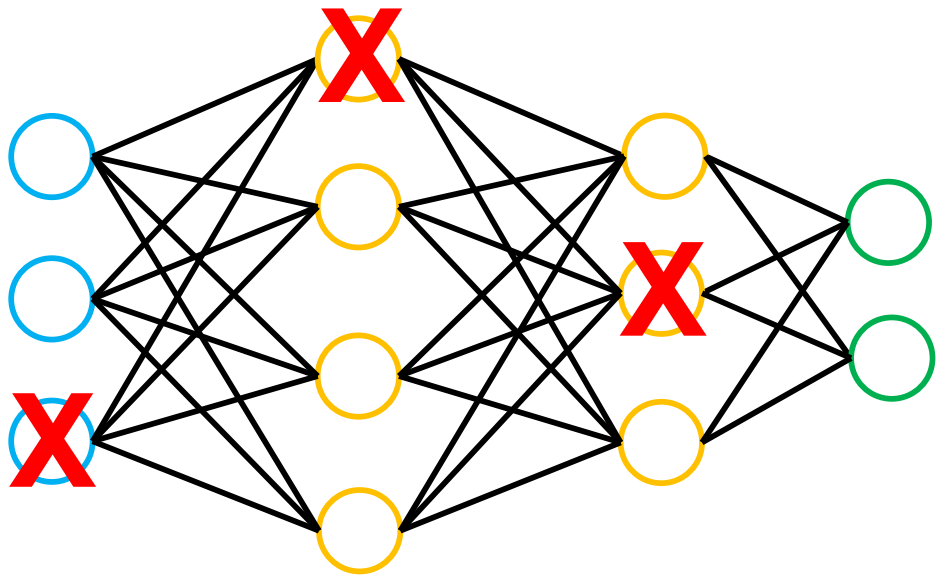
- Compute $\|\boldsymbol{\Theta}\|_2$ after each training step and clip θ if needed:

$$\boldsymbol{\Theta} \leftarrow \boldsymbol{\Theta} \frac{r}{\|\boldsymbol{\Theta}\|_2}$$

  – Here r increases the amount of regularisation

Regularisation: Dropout

- At every training step, every neuron (input or hidden) has a probability p of being temporarily “dropped out”:

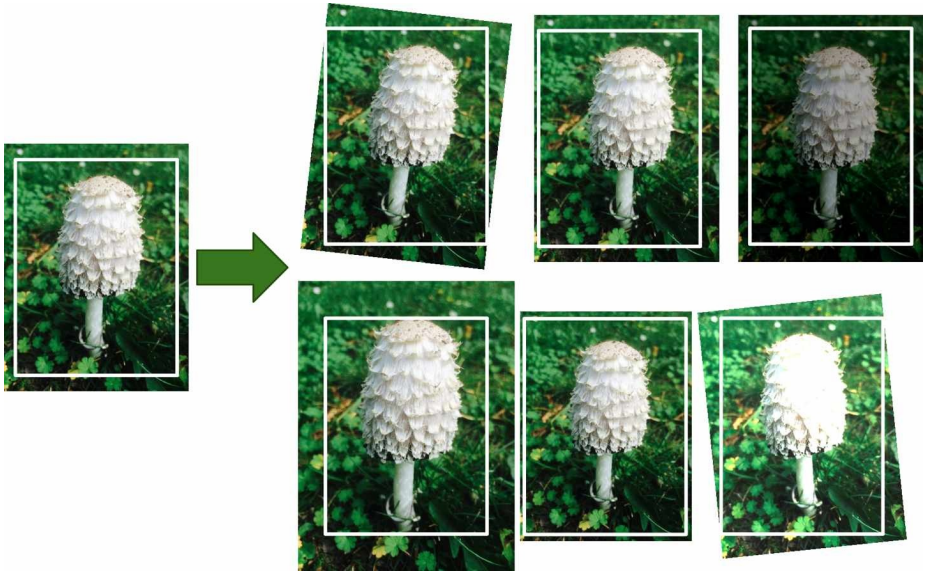


Hyperparameter p is called “dropout rate”, typically set to $p = 0.5$

This can be thought of as an ensemble learning method!

Regularisation: Data augmentation

- Generate new training instances from existing ones, artificially boosting the size of the training set
- New instances need to be realistic, e.g.
  - Include rotate, shift (translate), resize (scale), flip (reflect)
  - Add different lightning conditions or noise that can be learnt



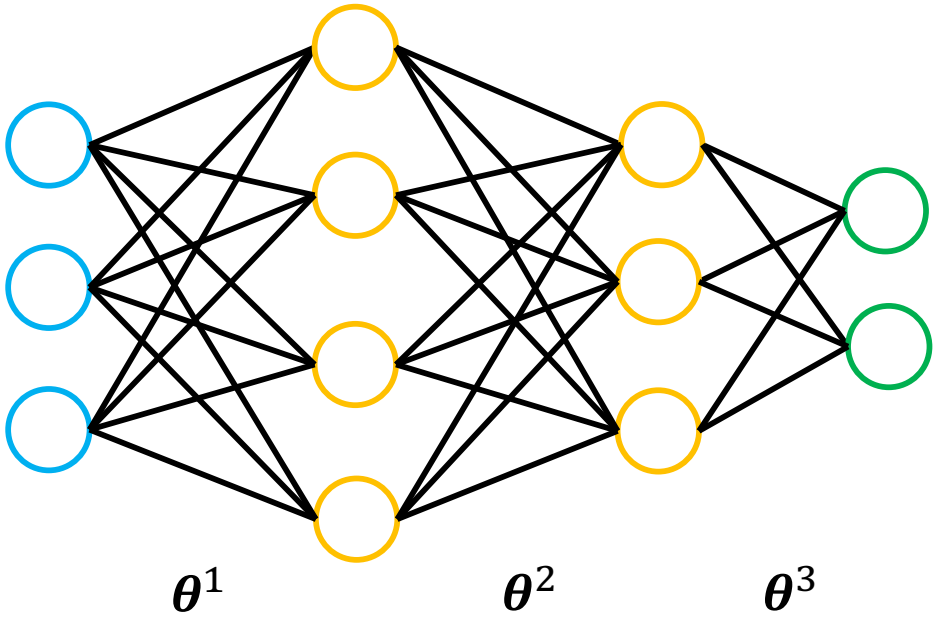
Example:

Mushroom classification task where training set can be augmented by generating new training instances

Implementation: Weight initialization

How to initialize weights?

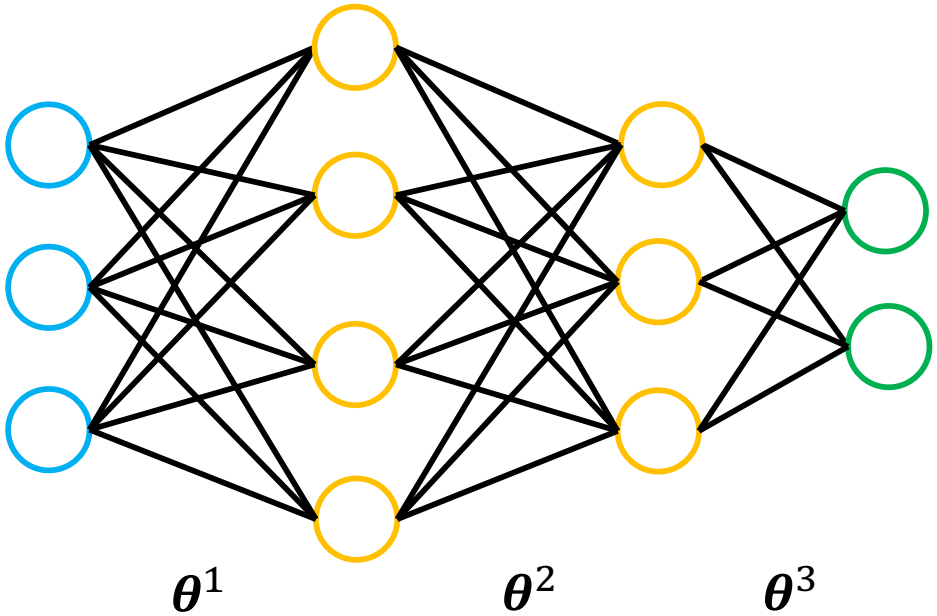
1. Constant weights, e.g. $\theta_{i,j}^{k} = 0$ Bad idea! Why?



Implementation: Weight initialization

How to initialize weights?

1. Constant weights, e.g. $\theta_{i,j}^{k} = 0$
2. Small random numbers, e.g. $\theta_{i,j}^{k} = 0.01 \cdot \mathcal{N}(0,1)$ or $\mathcal{N}(0,1)/\sqrt{n}$



Scaling by $\sqrt{n}$ avoids increasing variance with increasing number of inputs n

Implementation: Weight initialization

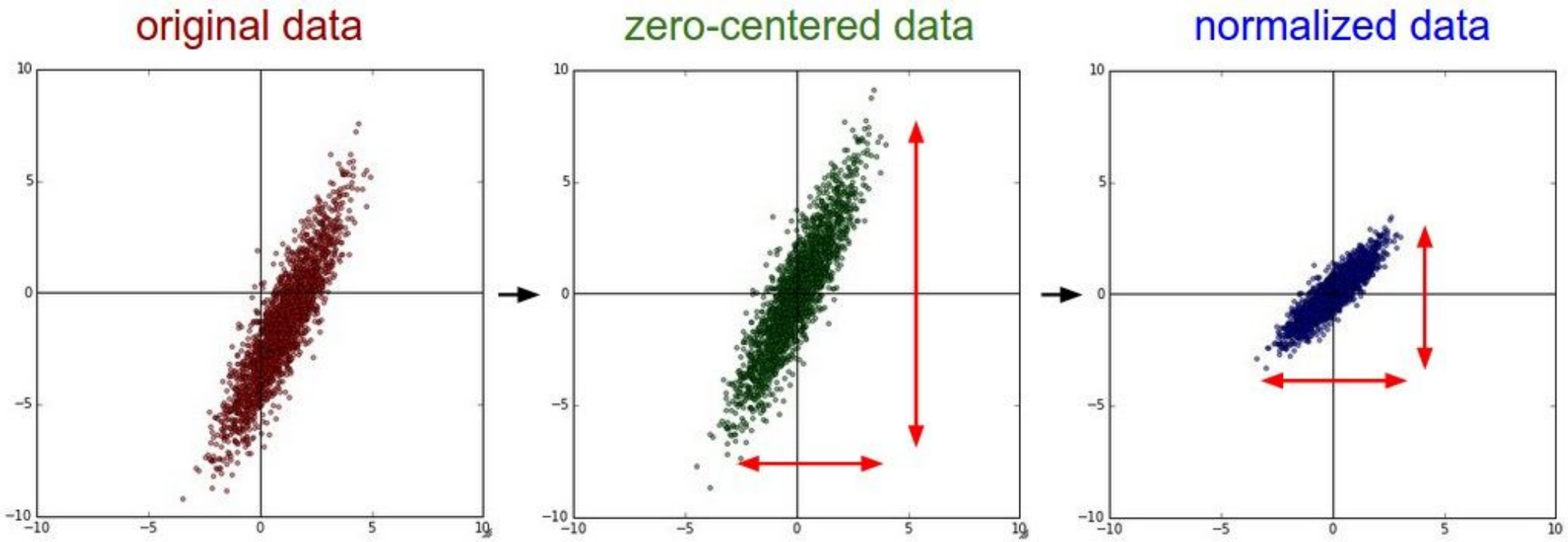
How to initialize weights?

1. Constant weights, e.g. $\theta_{i,j}^{k} = 0$
2. Small random numbers, e.g. $\theta_{i,j}^{k} = 0.01 \cdot \mathcal{N}(0,1)$ or $\mathcal{N}(0,1)/\sqrt{n}$

Other possibilities:

- X. Glorot and Y. Bengio (2010): $\theta_{i,j}^{k} = \mathcal{U}\left(-\frac{1}{\sqrt{n}}, +\frac{1}{\sqrt{n}}\right)$ where $\mathcal{U}(-a, +a)$ is the uniform distribution in the interval $-a$ and $+a$ and n is the number of inputs (also called *Xavier initialization*)
- He et al. (2015): $\theta_{i,j}^{k} = \mathcal{N}(0, \sqrt{2n})$

Implementation: Normalise data



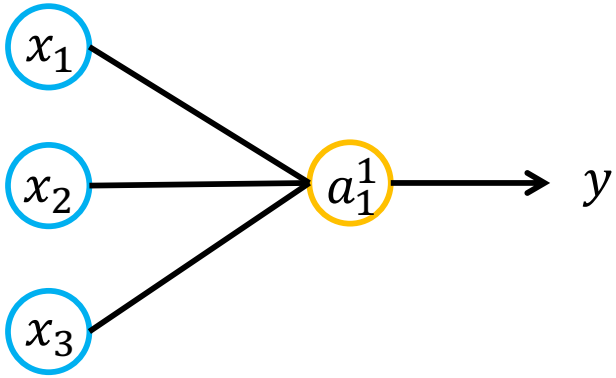

```
X -= np.mean(X, axis = 0)
```

```
X /= np.std(X, axis = 0)
```

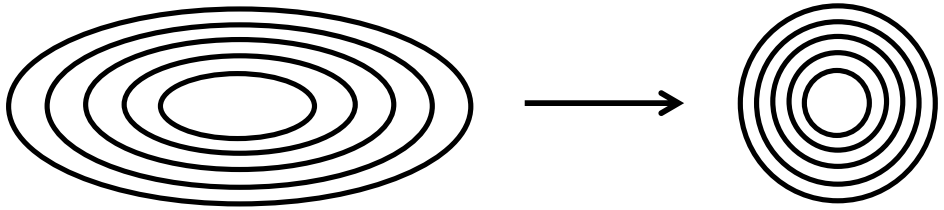
Implementation: Batch normalisation

- Simple trick to enable a wider ranger of hyperparameters for training
- Simplifies training of deep networks

$$\boldsymbol{\mu} = \frac{1}{N}\sum \boldsymbol{x}^i \qquad \boldsymbol{\sigma}^2 = \frac{1}{N}\sum \left(\boldsymbol{x}^i - \boldsymbol{\mu}\right) \cdot \left(\boldsymbol{x}^i - \boldsymbol{\mu}\right)$$

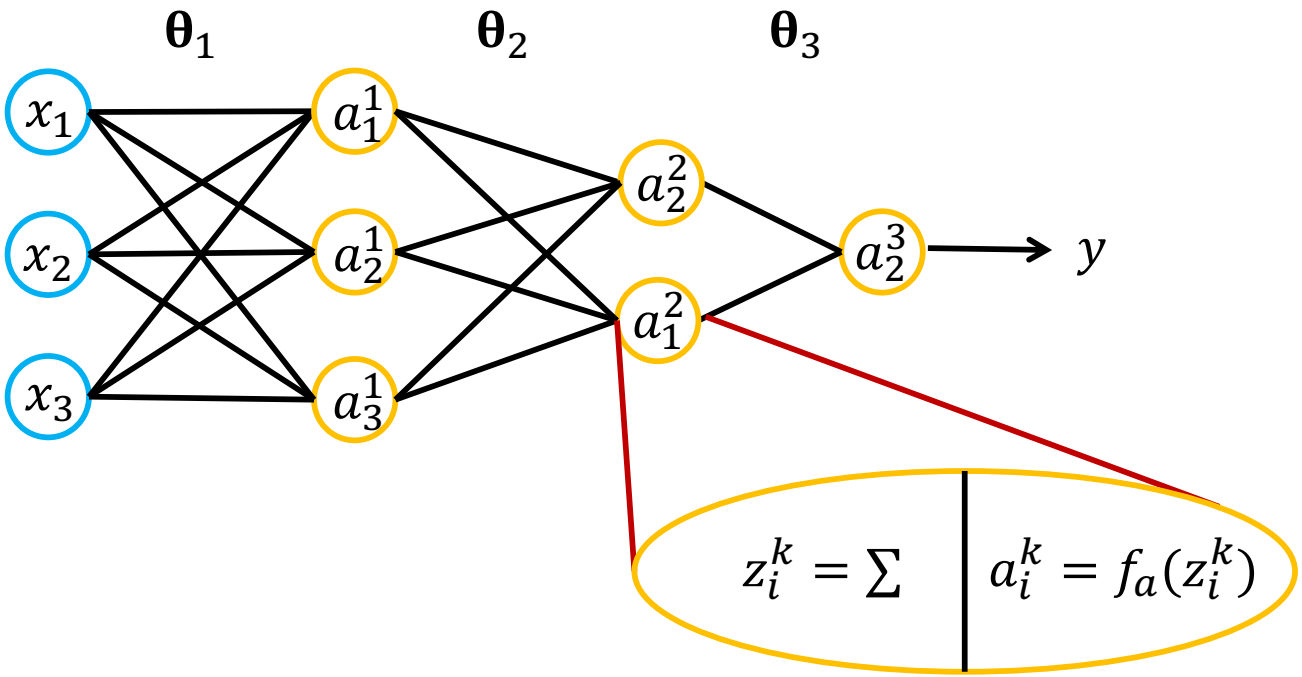


$$\widetilde{\boldsymbol{X}} = \frac{\boldsymbol{X} - \boldsymbol{\mu}}{\boldsymbol{\sigma}^2}$$



Implementation: Batch normalisation

- Simple trick to enable a wider ranger of hyperparameters for training
- Simplifies training of deep networks



Can normalize a^k in order to train $\boldsymbol{\theta}^{k+1}$ faster

Alternatively, normalize z^k

Implementation: Batch normalisation

- Given some intermediate values in the NN, i.e. $z_1, \ldots, z_m$

$$\mu = \frac{1}{N_B}\sum z_i \qquad \sigma^2 = \frac{1}{N_B}\sum (z_i - \mu)(z_i - \mu)$$

$$z_i^{norm} = \frac{z_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

$$\widetilde{z_i} = \gamma z_i^{norm} + \beta$$

Learnable parameters during optimization

Use $\widetilde{z_i}$ instead z_i