Introduction to Deep Learning

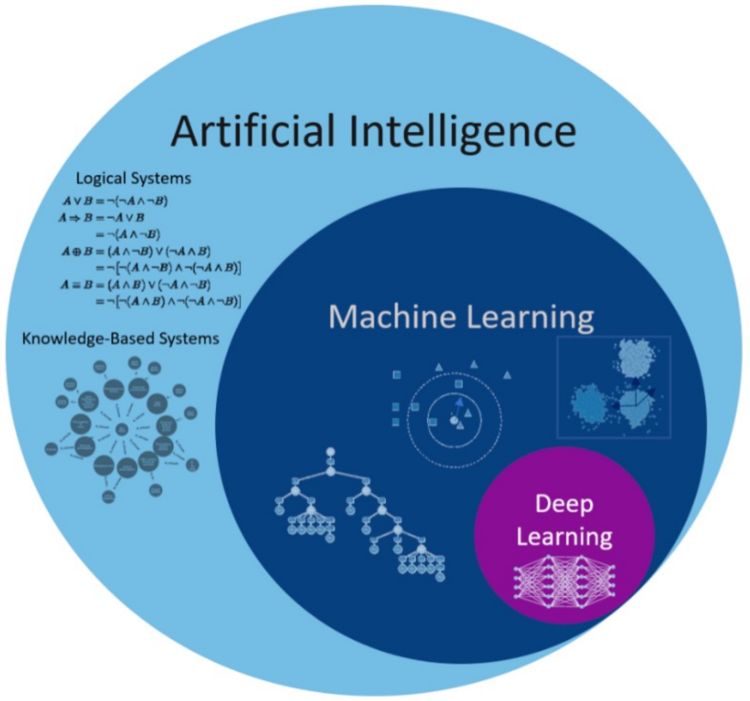
Multi-Layer Perceptrons (MLPs)

Loïc Le Folgoc

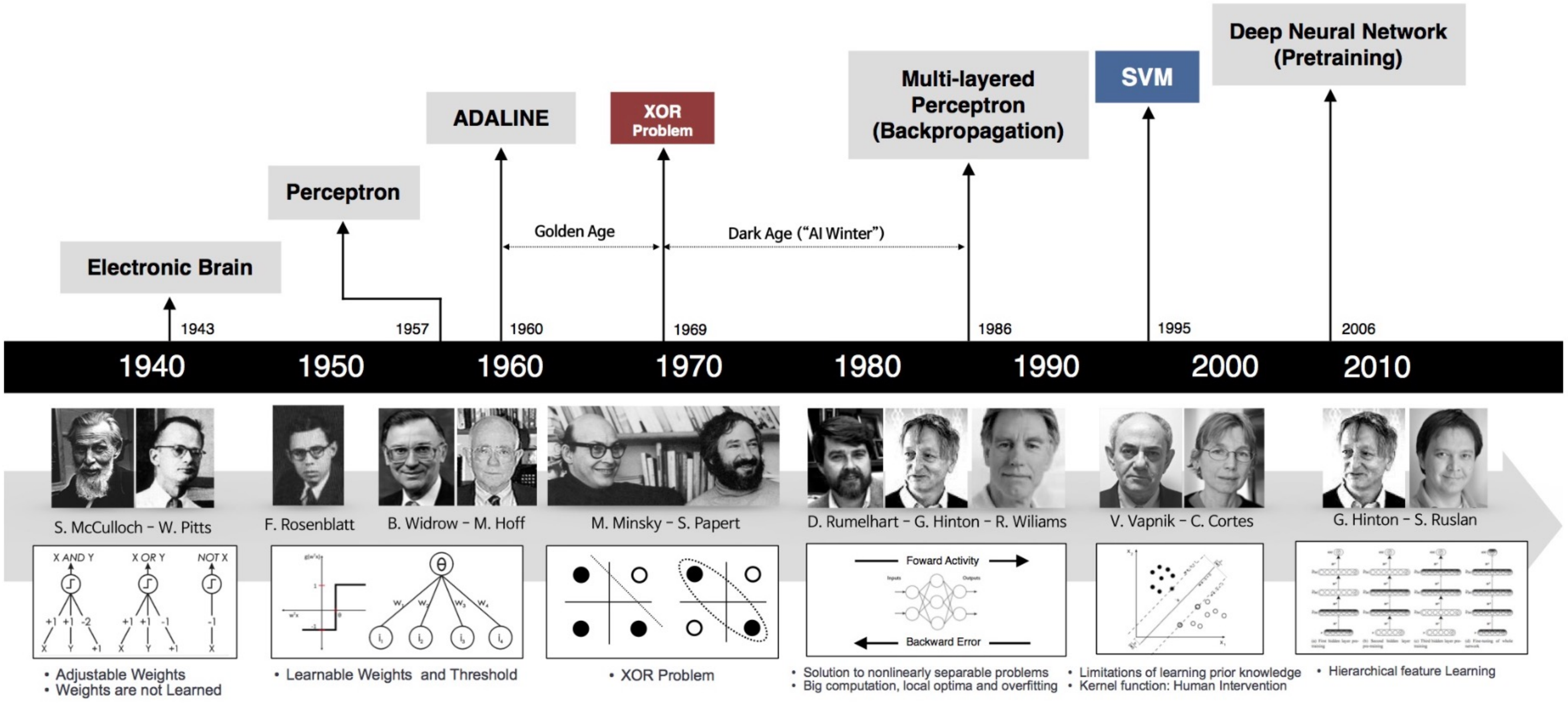
loic.lefolgoc@telecom-paris.fr

Introduction

- In this lesson, we will look at **Artificial Neural Networks** (ANNs), and in particular **Multi-Layer Perceptrons** (MLPs)
- It is a class of machine learning models with a wide range of applications, for supervised and unsupervised learning
- Recently, **deep learning** has had a huge impact on many domains, including image processing and computer vision



Evolution of AI and Deep Learning



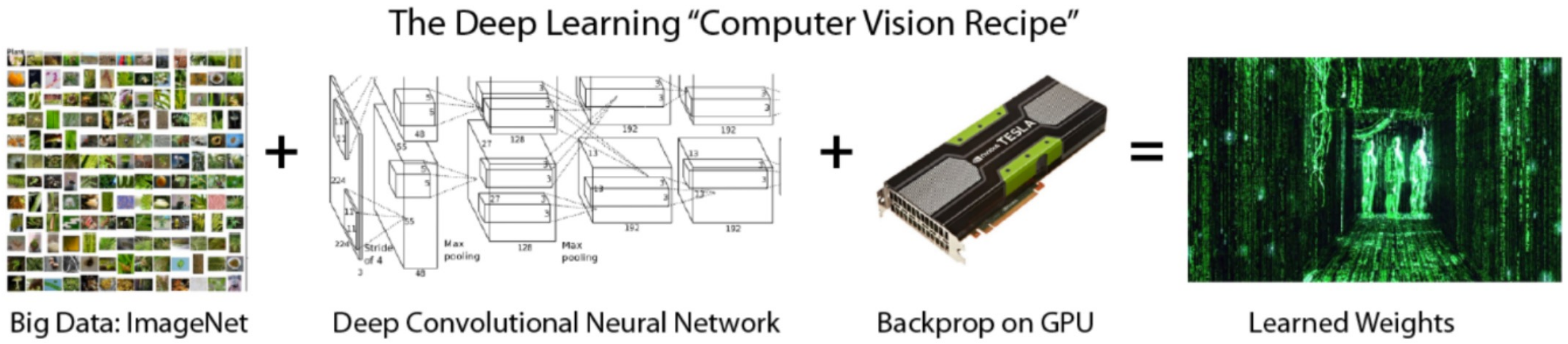
Deep Neural Network (Pretraining)
SVM
Multi-layered Perceptron (Backpropagation)
XOR Problem
ADALINE
Perceptron
Electronic Brain
Golden Age
Dark Age ("AI Winter")
1943
1957
1960
1969
1986
1995
2006
1940
1950
1960
1970
1980
1990
2000
2010
S. McCulloch – W. Pitts
F. Rosenblatt
B. Widrow – M. Hoff
M. Minsky – S. Papert
D. Rumelhart – G. Hinton – R. Wiliams
V. Vapnik – C. Cortes
G. Hinton – S. Ruslan
X AND Y
X OR Y
NOT X
Foward Activity
Backward Error
• Adjustable Weights
• Weights are not Learned
• Learnable Weights and Threshold
• XOR Problem
• Solution to nonlinearly separable problems
• Big computation, local optima and overfitting
• Limitations of learning prior knowledge
• Kernel function: Human Intervention
• Hierarchical feature Learning

Image source

Evolution of AI and Deep Learning

- Deep learning has been known by various names for a long time ("cybernetics", artificial neural networks, etc.)

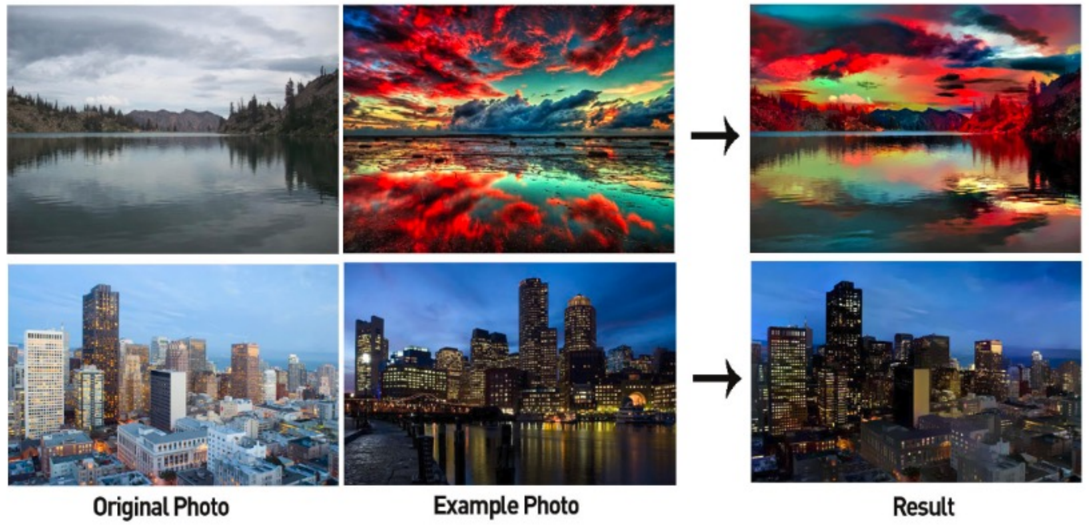
- Why has there been a recent explosion of deep learning?
  - Increase in number of large datasets (in terms of number of elements and dimensionality)
  - Increase in **computing power (GPUs)**
  - Auto-differentiation libraries

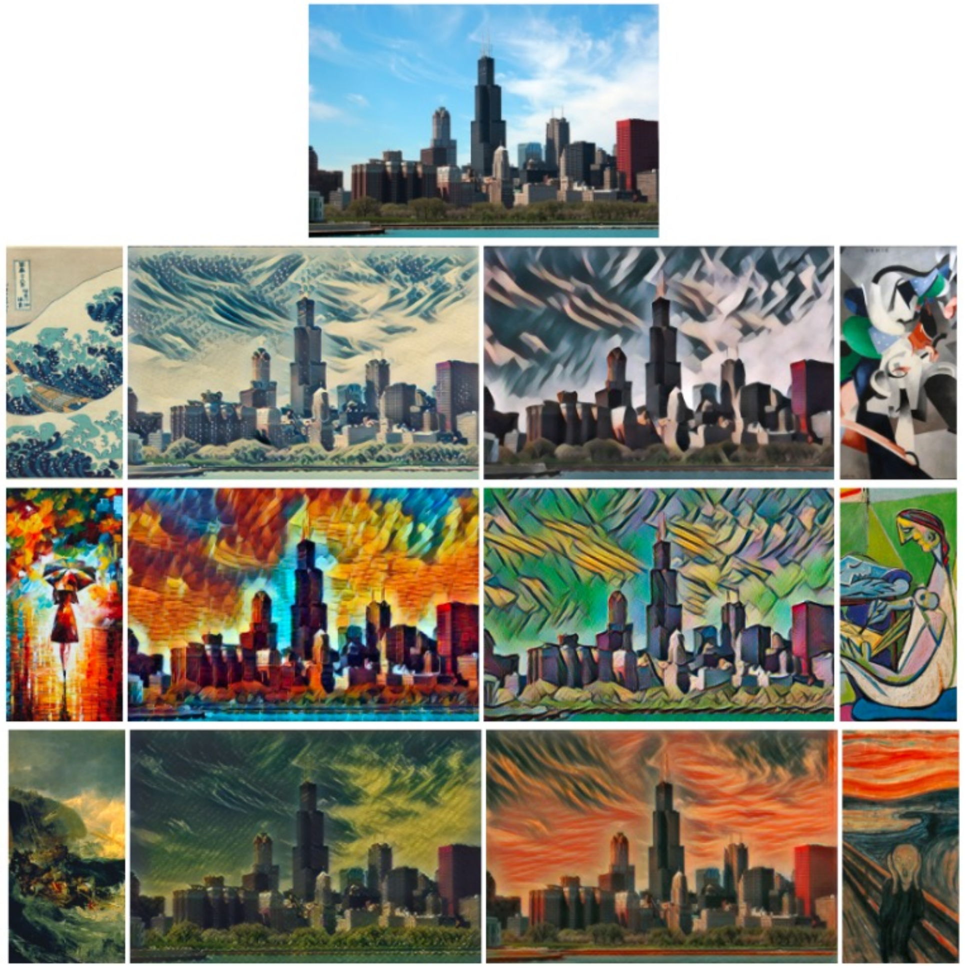

The Deep Learning "Computer Vision Recipe"
+
+
=
Big Data: ImageNet
Deep Convolutional Neural Network
Backprop on GPU
Learned Weights

Introduction

- Modern deep learning has applications in all areas of computer vision, and in a multitude of other domains

- Example: **style transfer**





Introduction

- Example: in 2016 **Google DeepMind's AlphaGo** beats Lee Sedol, one of the best player of Go in the world; in 2017 it beats Ke Jie, world #1

Artificial Neural networks / MLPs

Biological neuron

- The **brain** and its functioning **inspired early artificial intelligence**

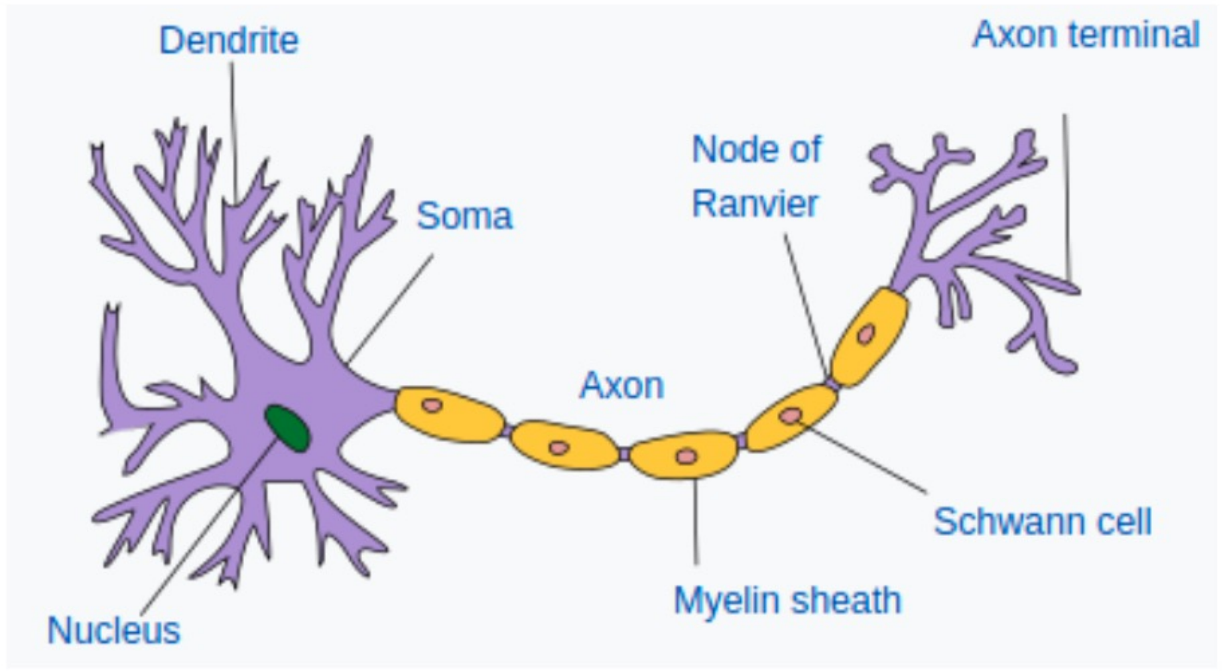


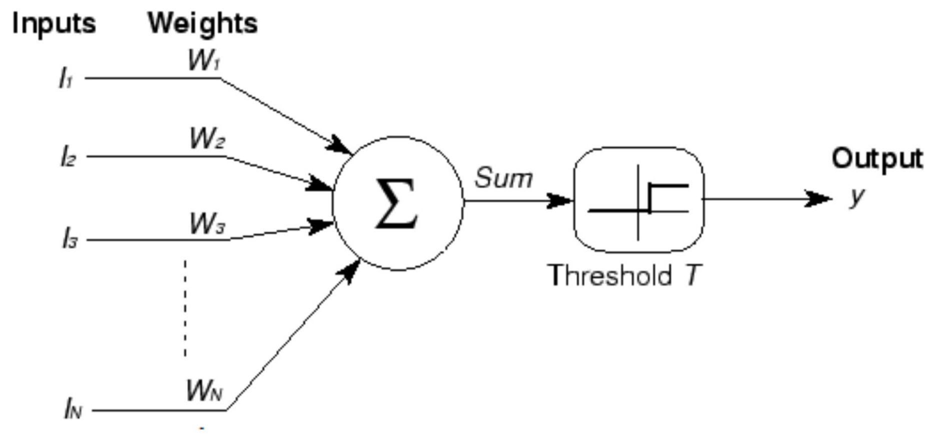
Image source: wikipedia

- Neurons receive electric signals via *dendrites*, process the signal in the *soma* and transmit the output via *synapses*

- Neurons have an “all or none” principle: under a threshold, no electrical signal is emitted; above the threshold, the same one is always emitted

Biological neuron

- A **mathematical model of the neuron** was first proposed by McCulloch and Pitts*
- The model is quite simple (it is the simplest possible ANN):
  - A series of inputs
  - A list of weights
  - A threshold T (inspired by the "all or none" principle)

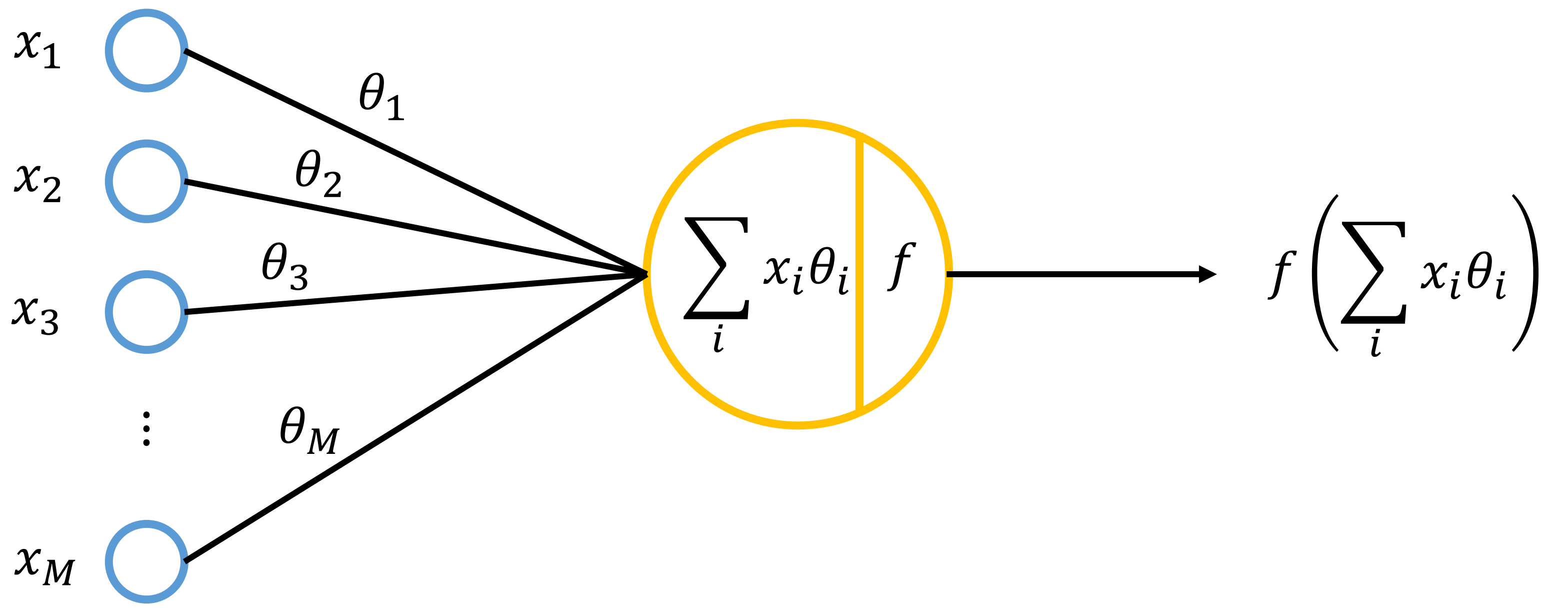


- The weights had to be set by hand; this is time consuming
- Rosenblatt** proposed an algorithm to optimize these weights, and named this model the **perceptron**

*__A logical calculus of the ideas immanent in nervous activity__, McCulloch, W., and Pitts, W., *The Bulletin of Mathematical Bio*physics, 1943
**__The perceptron: A probabilistic model for information storage and organization in the brain__, Rosenblatt, *Psychological Review*, 1958

First neural network: perceptron for binary classification

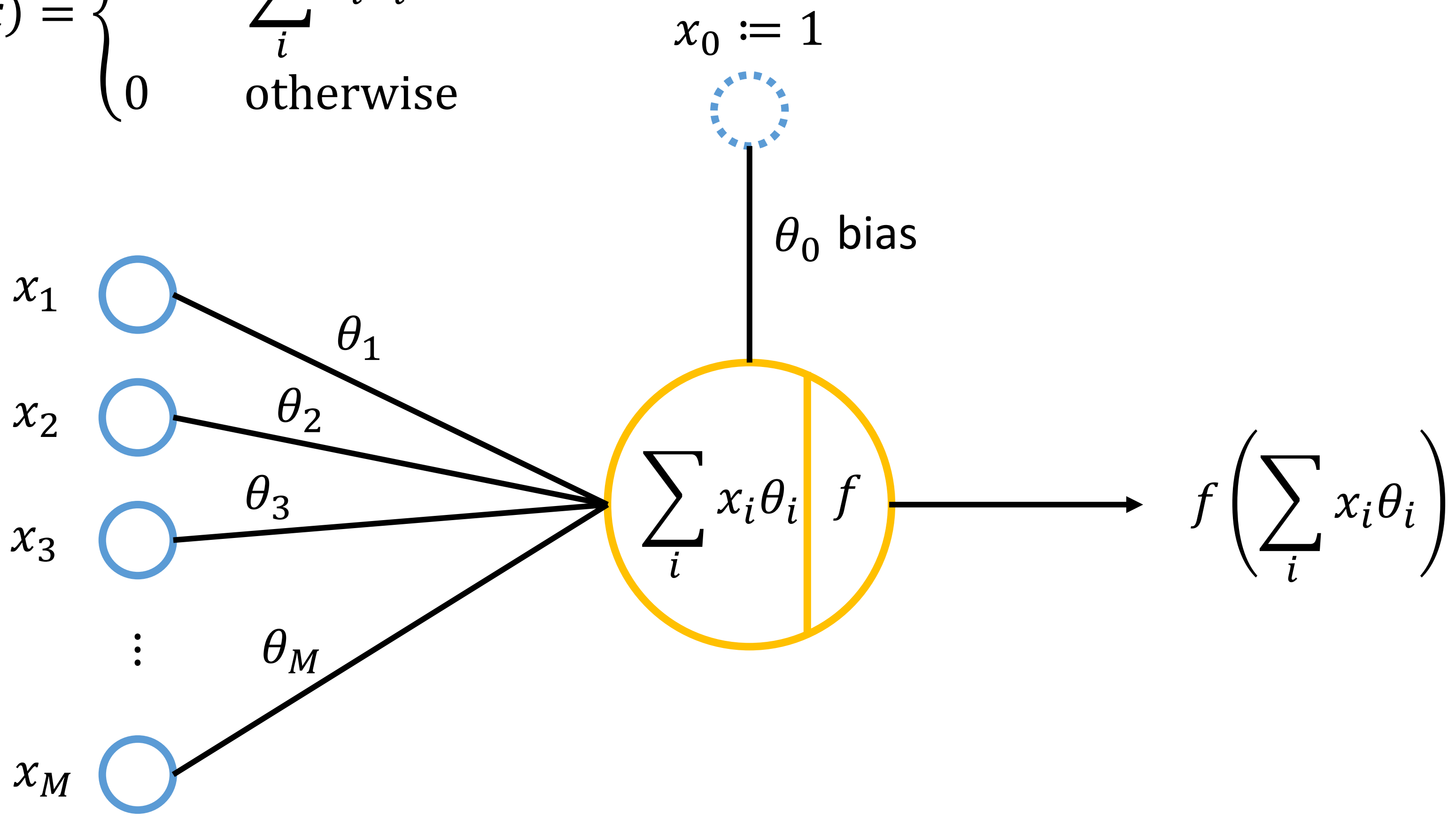
Hypothesis $$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \sum_i x_i\theta_i > 0 \\ 0 & \text{otherwise} \end{cases}$$



Input Weights Output

First neural network: perceptron for binary classification

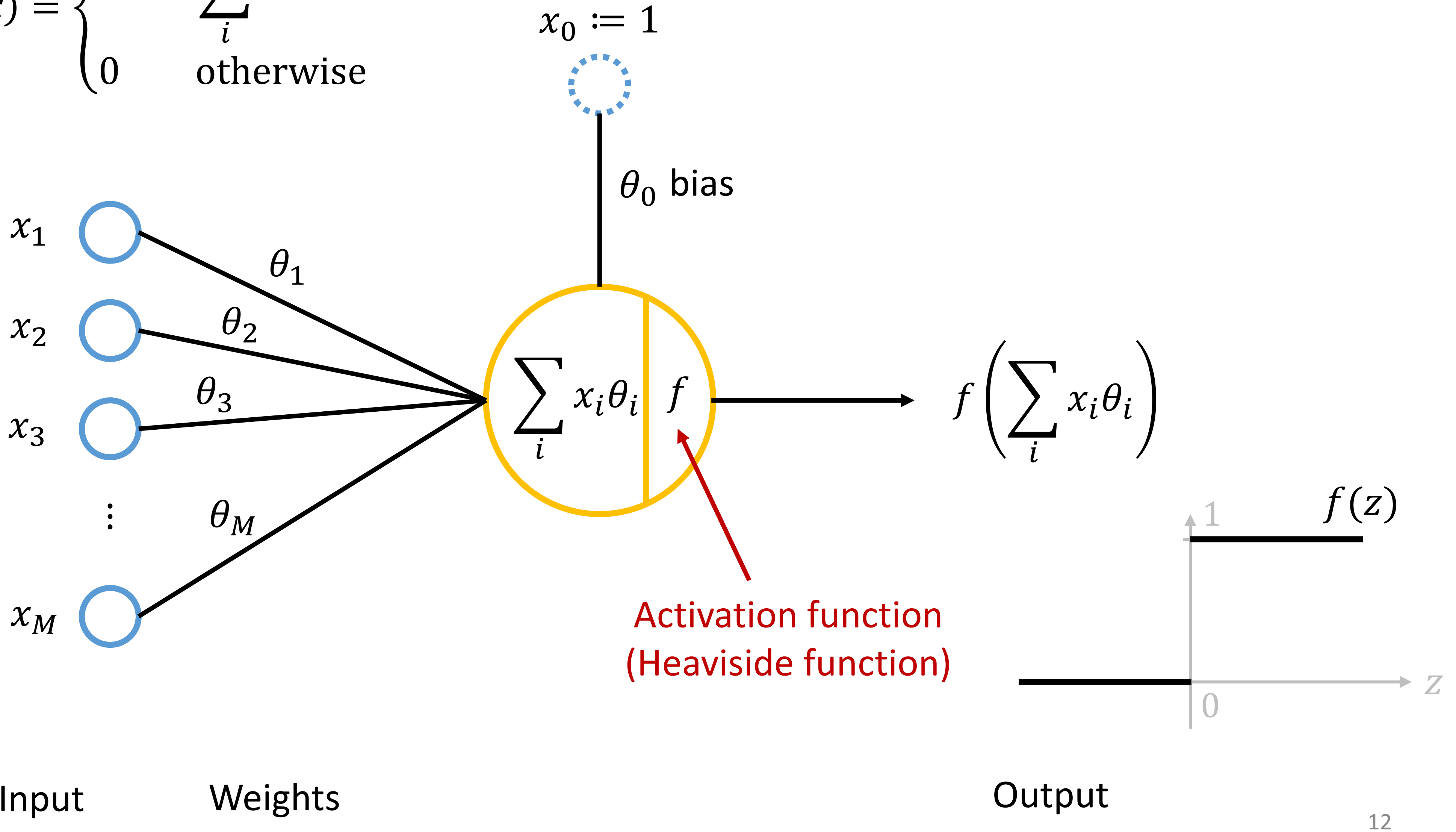
Hypothesis $h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \sum_i x_i \theta_i > 0 \\ 0 & \text{otherwise} \end{cases}$



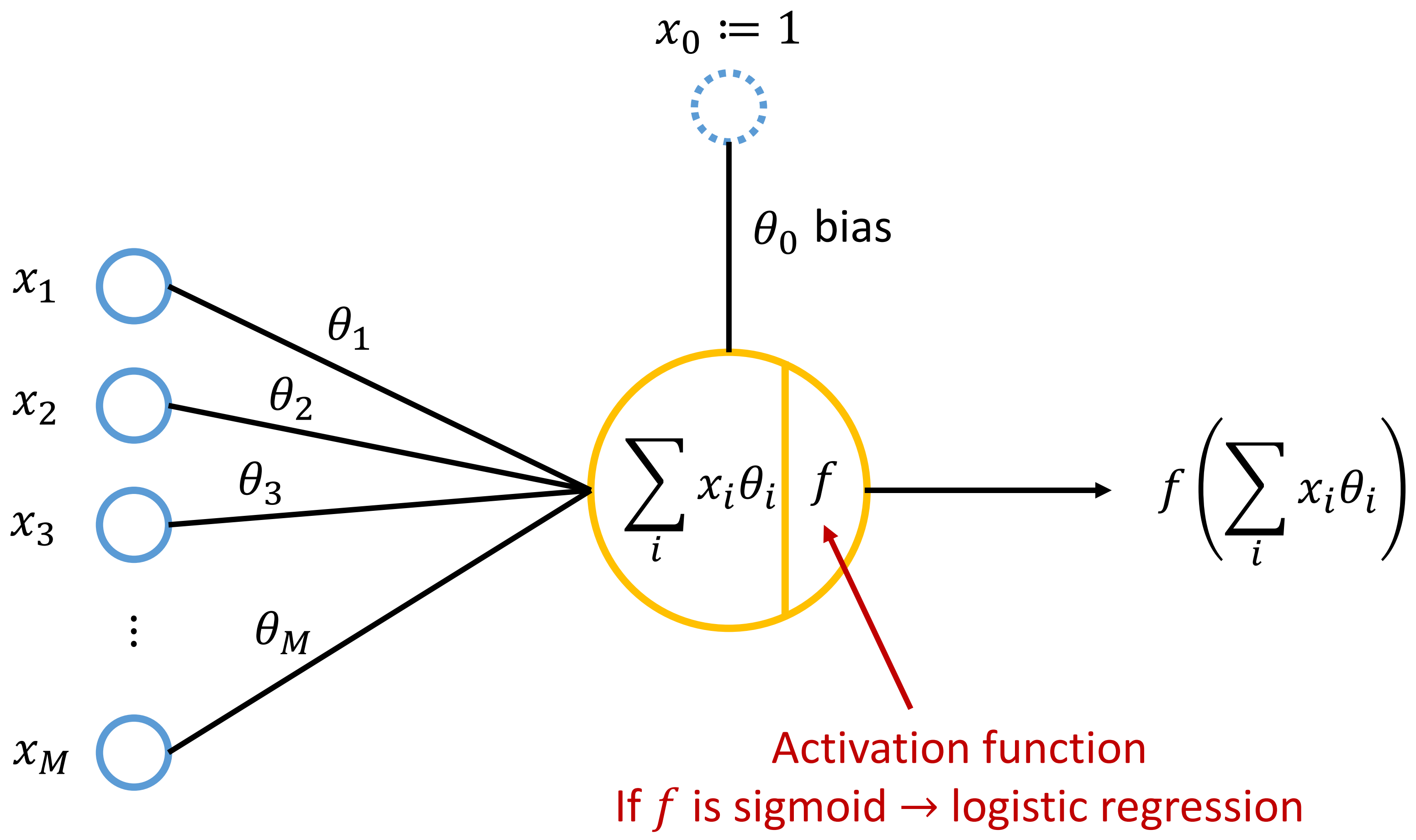
Input Weights Output

First neural network: perceptron for binary classification

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



First neural network: perceptron for binary classification



$$f(z) = \frac{1}{1+e^{-z}}$$

Input Weights Output

Training a single-layer perceptron

```
Initialize the weights $\theta_i$ to $0$ or a small random value

For each example in the training set $\boldsymbol{X} = \{\boldsymbol{x}^{(n)}\}_{n=1\cdots N}$ with labels $\{y^{(n)}\}_{n=1\cdots N}$:
    Compute the prediction $\hat{y}^{(n)} \triangleq f(\boldsymbol{\theta}^{\boldsymbol{T}}\boldsymbol{x}^{(n)})$
    For all features $i = 1 \cdots M$, update the weight:
        $\theta_i^{t+1} \coloneqq \theta_i^t - \alpha \cdot (\hat{y}^{(n)} - y^{(n)})x_i^{(n)}$
```

Training a single-layer perceptron

```
Initialize the weights $\theta_i$ to $0$ or a small random value

For each example in the training set $\boldsymbol{X} = \{\boldsymbol{x}^{(n)}\}_{n=1\cdots N}$ with
labels $\{y^{(n)}\}_{n=1\cdots N}$:

    Compute the prediction $\hat{y}^{(n)} \triangleq f(\boldsymbol{\theta}^{\boldsymbol{T}}\boldsymbol{x}^{(n)})$

    For all features $i = 1 \cdots M$, update the weight:
        $\theta_i^{t+1} \coloneqq \theta_i^t - \alpha \cdot (\hat{y}^{(n)} - y^{(n)})x_i^{(n)}$
```

learning rate

Training a single-layer perceptron

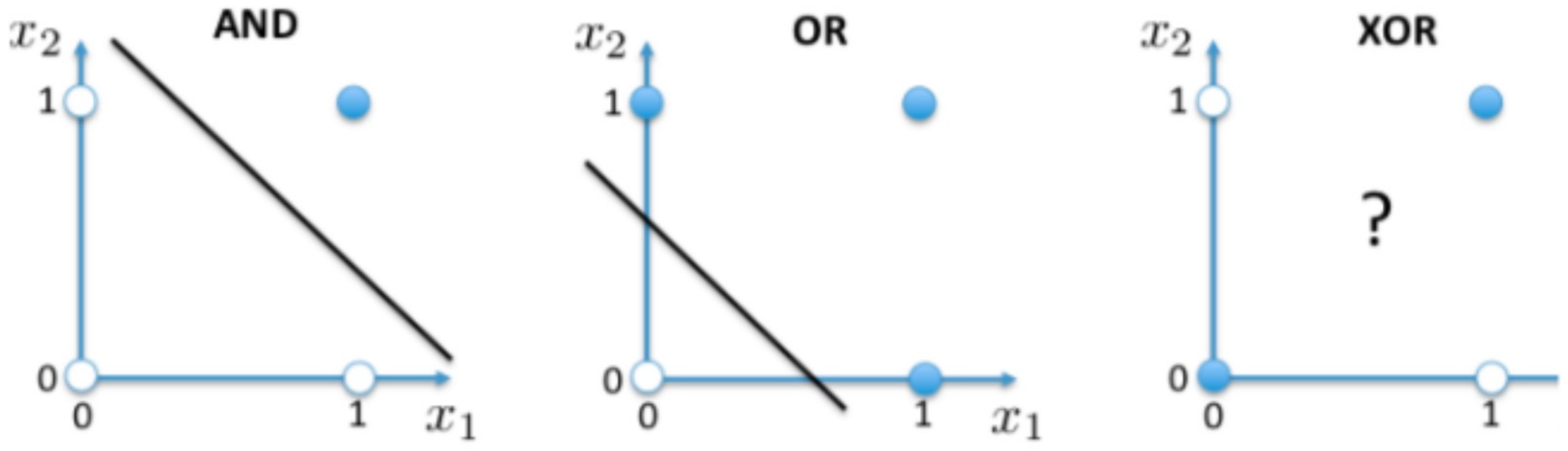
```
Initialize the weights $\theta_i$ to $0$ or a small random value

For each example in the training set $\boldsymbol{X} = \{\boldsymbol{x}^{(n)}\}_{n=1\cdots N}$ with
labels $\{y^{(n)}\}_{n=1\cdots N}$:

    Compute the prediction $\hat{y}^{(n)} \triangleq f(\boldsymbol{\theta}^{\boldsymbol{T}}\boldsymbol{x}^{(n)})$

    For all features $i = 1 \cdots M$, update the weight:
        $\theta_i^{t+1} \coloneqq \theta_i^t - \alpha \cdot (\hat{y}^{(n)} - y^{(n)}) x_i^{(n)}$
```

- Can be interpreted as **stochastic gradient descent** for a simple loss $\mathcal{L}(\boldsymbol{\theta}) \triangleq \sum_n (\hat{y}^{(n)} - y^{(n)}) \boldsymbol{\theta}^{\boldsymbol{T}} \boldsymbol{x}^{(n)}$
- NN training will generalize this approach

The perceptron and the XOR function

- Unfortunately, a significant problem was found with the perceptron

- It is a linear classifier: Minsky and Papert* showed that the **XOR function** could not be approximated by a perceptron: the **XOR is not linearly separable**

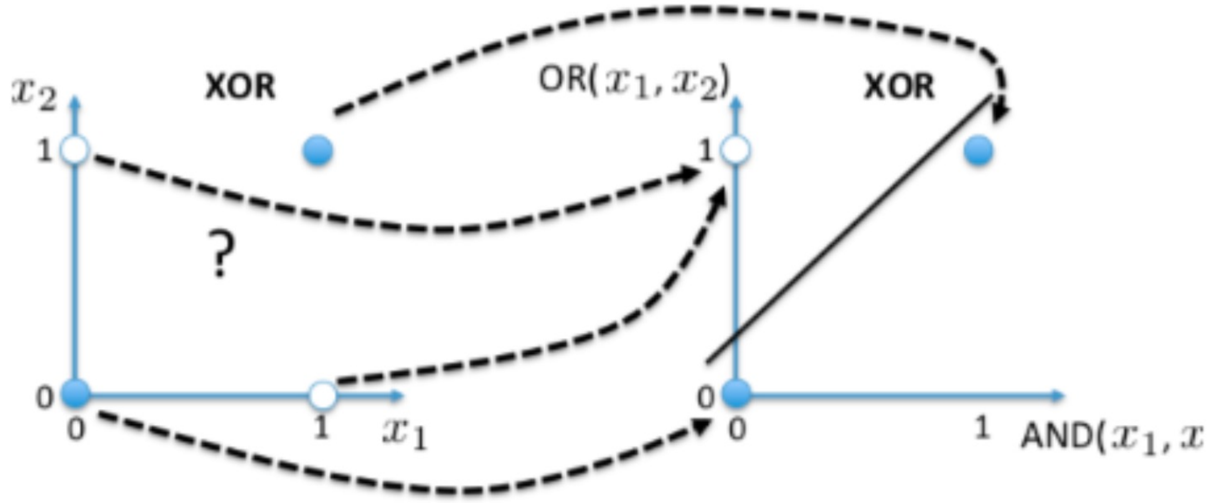


- This resulted in a long period (about 10 or so years) when neural networks fell into disfavor; this was known as the “AI winter”

*__Perceptrons: An introduction to computational geometry__, Minsky, M., and Papert, S.A., *Cambridge, MA: MIT Press*, 1969

Multi-Layer Perceptron (MLP)

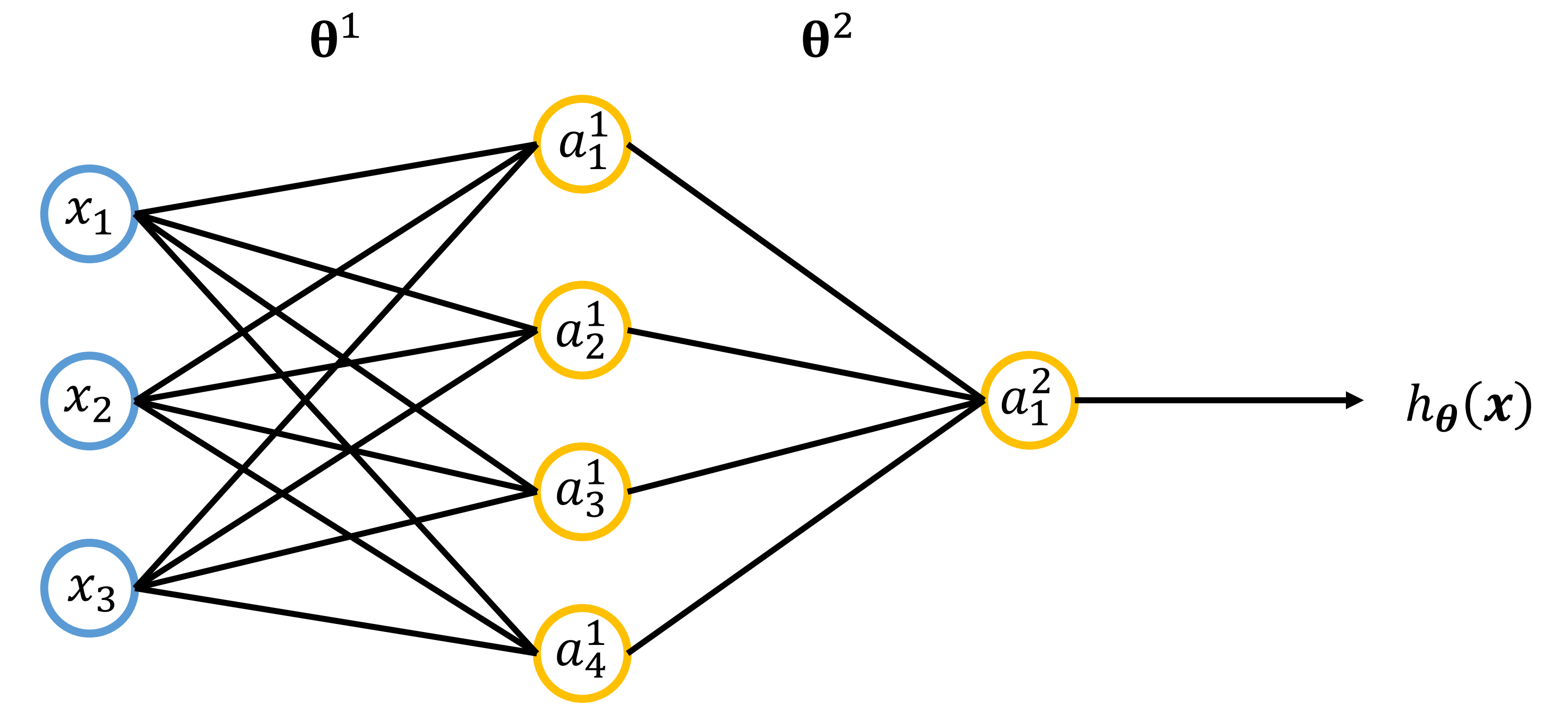
- However, a solution was found by introducing several layers (“hidden layers”) → **MLP**
  - Rumelhart et al.* showed that this could be trained to learn the XOR function



- In fact, the **universal approximation theorem**** guarantees that any continuous function may be approximated with arbitrary precision by a large enough MLP with 1 hidden layer (although the network may be unfeasibly large)

*__Learning internal representations by error propagation__, Rumelhart, D., Hinton, G., and Williams, R., *California University San Diego LA Jolla Inst. For Cognitive Science*, 1985
**__Approximations by superposition of sigmoidal functions__, Cybenko, G., *Mathematics of Control, Signals and Systems*, 1989

Multi-Layer Perceptron

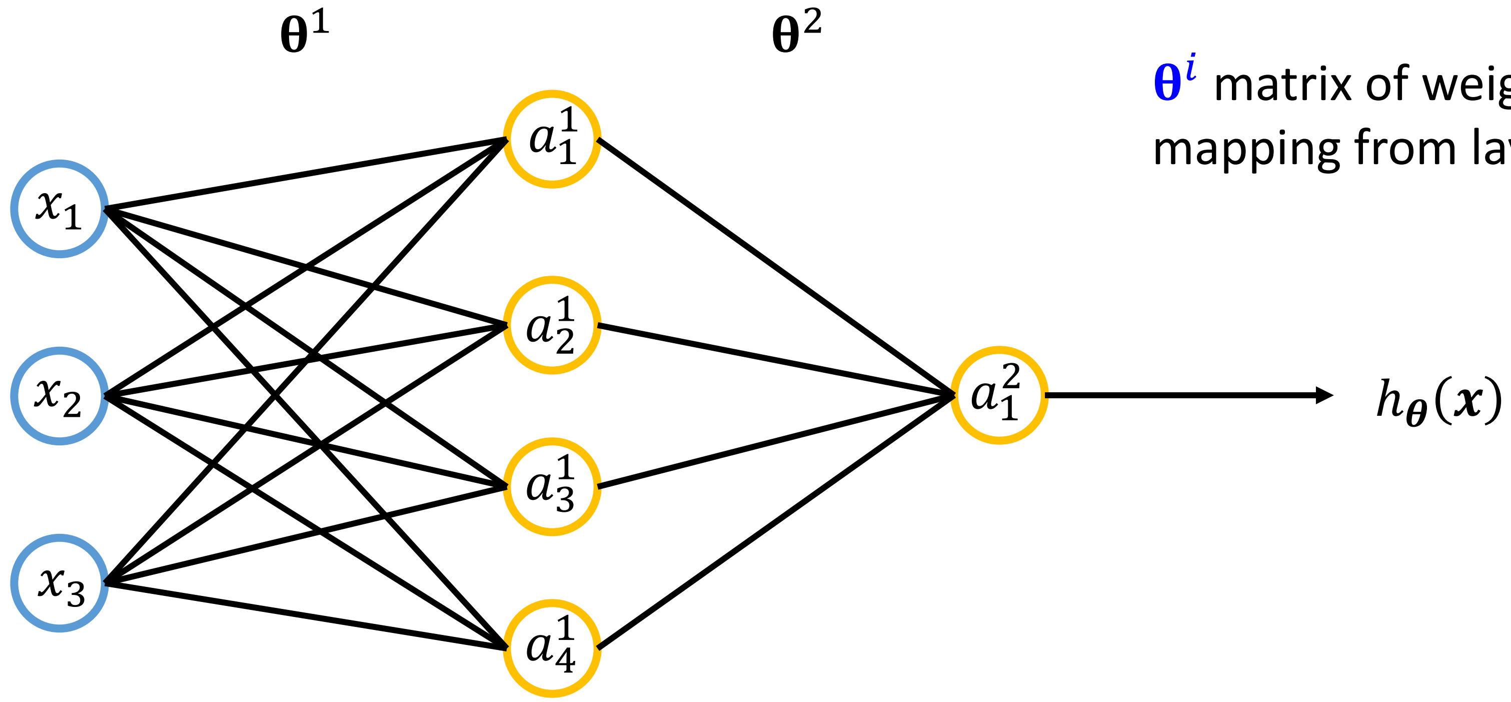


Layer 0
Input

Layer 1
Hidden

Layer 2
Output

Multi-Layer Perceptron



a_j^i activation of neuron j in layer i

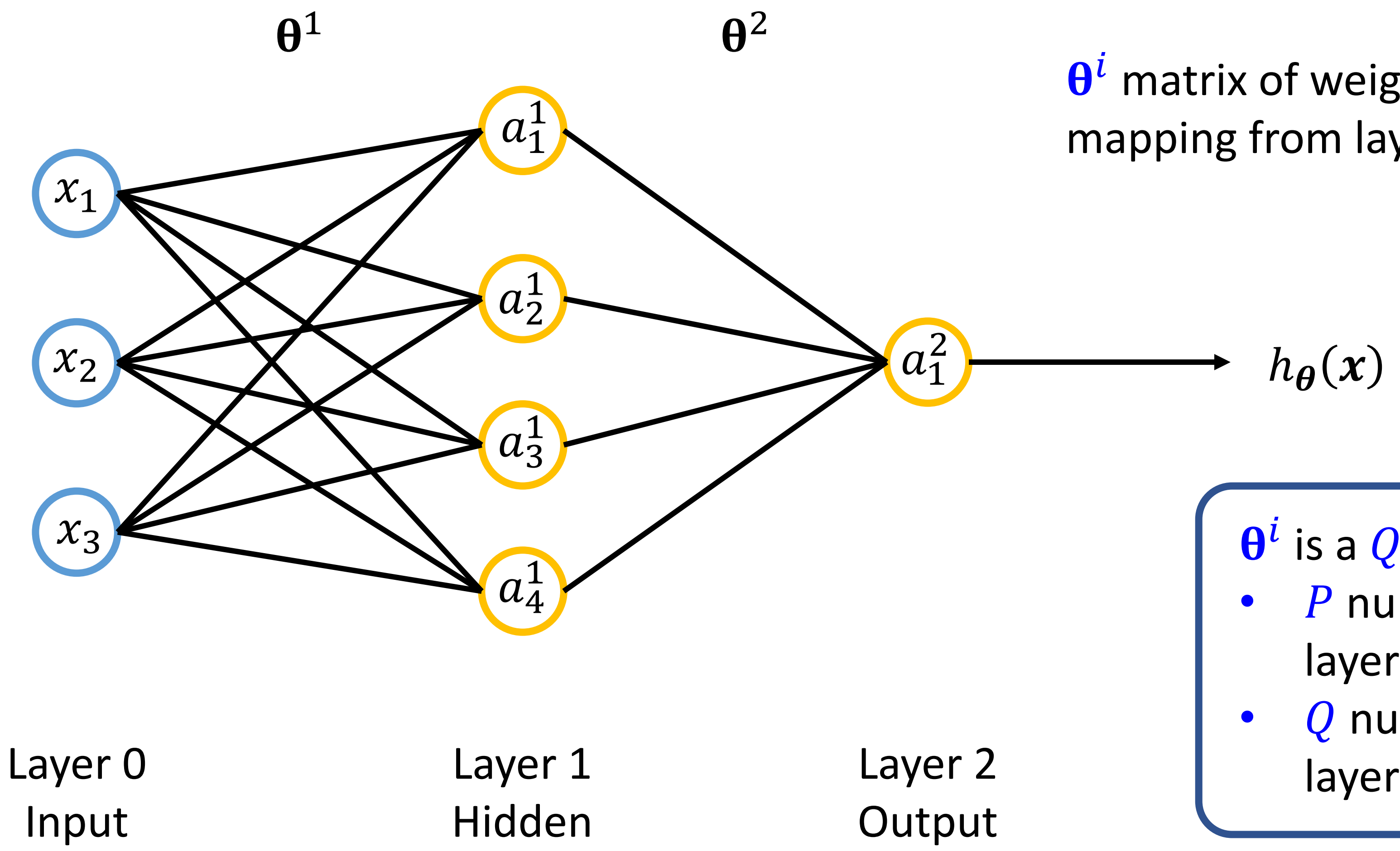
$\boldsymbol{\theta}^i$ matrix of weights controlling mapping from layer $i-1$ to layer i

Layer 0
Input

Layer 1
Hidden

Layer 2
Output

Multi-Layer Perceptron



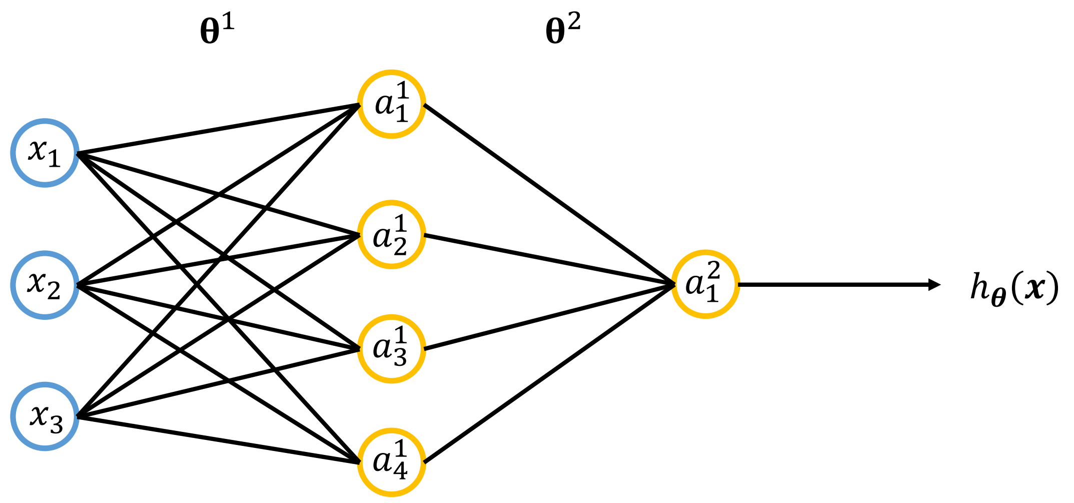
a_j^i activation of neuron j in layer i

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

$\boldsymbol{\theta}^i$ is a $Q\times(P+1)$ matrix

- P number of neurons in layer $i-1$
- Q number of neurons in layer i

Multi-Layer Perceptron



$$\boldsymbol{a}^1 \coloneqq f(\boldsymbol{\theta}^1 \boldsymbol{x})$$

$$\boldsymbol{a}^2 \coloneqq f(\boldsymbol{\theta}^2 \boldsymbol{a}^1) = f(\boldsymbol{\theta}^2 f(\boldsymbol{\theta}^1 \boldsymbol{x}))$$

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

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

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

$$a_4^1 \coloneqq f(\theta_{3,0}^1 + \theta_{3,1}^1 x_1 + \theta_{3,2}^1 x_2 + \theta_{3,3}^1 x_3)$$

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) \coloneqq a_1^2 \coloneqq f(\theta_{0,0}^2 + \theta_{0,1}^2 a_1^1 + \theta_{0,2}^2 a_2^1 + \theta_{0,3}^2 a_3^1 + \theta_{0,4}^2 a_4^1)$$

Multiclass classification: NN with multiple outputs

- So far we have considered NNs with a single output; this works for:
  - Scalar regression $y \in \mathbb{R}$
  - Binary classification $y \in \{0,1\}$

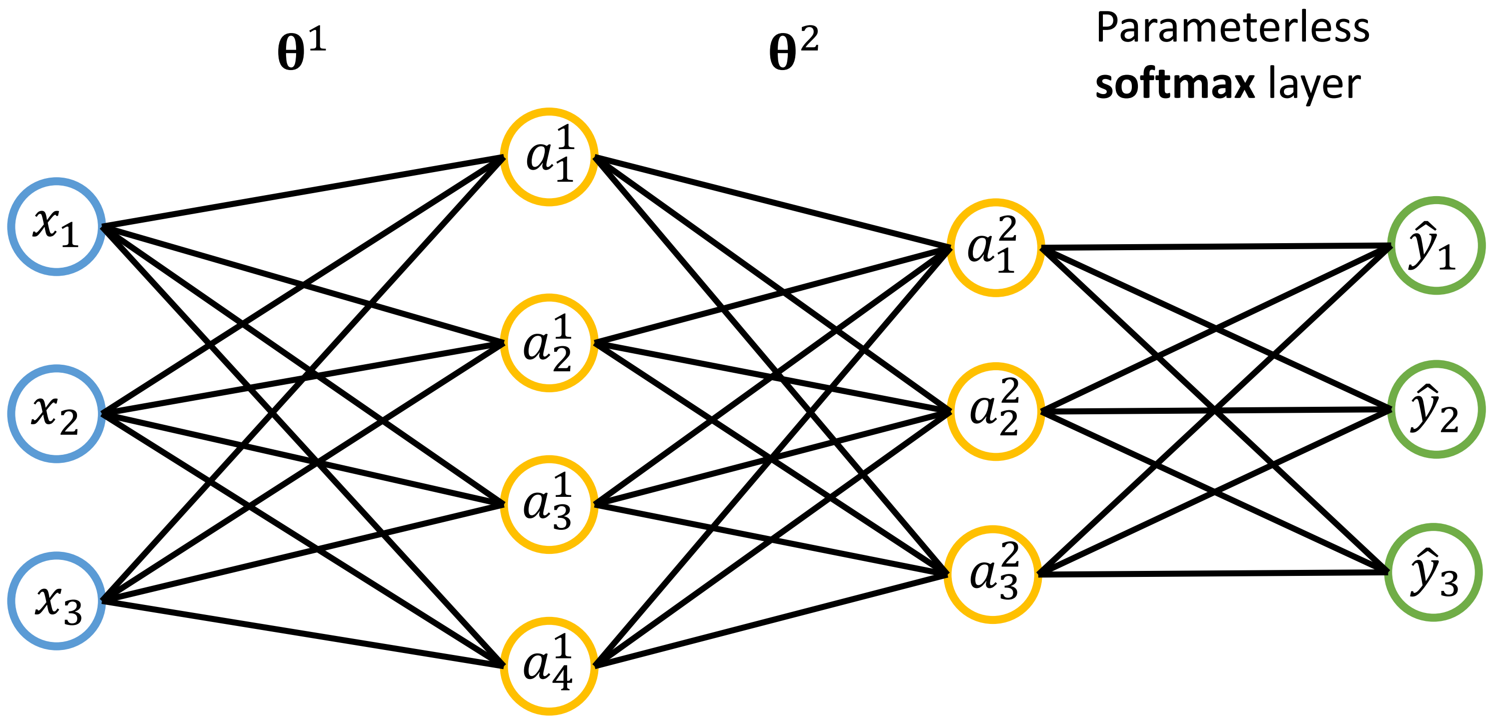
- For K classes we can use a vector $\boldsymbol{y} = (y_1, \cdots, y_K)$ such that

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

- This is called **one-hot encoding** (only one element is different from 0)
- NN outputs can represent class probabilities

Multiclass classification: NN with multiple outputs

With $K = 3$



Softmax layer

The **softmax function** transforms K real values to a K-vector of probabilities summing to 1

$$\hat{y}_k \triangleq \frac{e^{a_k}}{\sum_l e^{a_l}}$$

The outputs $\hat{y}_k$ are such that

- $\hat{y}_k \geq 0$
- $\sum_k \hat{y}_k = 1$

This is an **extension of the sigmoid** function to K classes

Loss function for K-way classification

How to train? Minimize **cross-entropy loss**, also called log-loss

- In general, cross-entropy $H(p,q)$ of probability distributions p and q

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

- We define the loss for one data point as $\mathcal{L}(\boldsymbol{y},\hat{\boldsymbol{y}}) \triangleq H(\boldsymbol{y},\hat{\boldsymbol{y}}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k$
  - where the prediction $\hat{\boldsymbol{y}}$ depends on the parameters of the model

Probabilistic interpretation:

- This is equivalent to modeling the class by a **categorical distribution** with probabilities $\hat{y}_k$:

$$\mathcal{C}(\boldsymbol{y}|\hat{\boldsymbol{y}}) = \prod_{k=1}^{K} \hat{y}_k^{\,y_k}$$

- then taking the **negative log-likelihood** $\mathcal{L}(\boldsymbol{y},\hat{\boldsymbol{y}}) \triangleq -\log \mathcal{C}(\boldsymbol{y}|\hat{\boldsymbol{y}})$ as loss function

Loss function for regression

Minimize **mean squared error**, also called sum-of-squared-differences

- We define the loss for one data point as $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) \triangleq (\hat{\boldsymbol{y}} - \boldsymbol{y})^2$
    - where the prediction $\hat{\boldsymbol{y}}$ depends on the parameters of the model

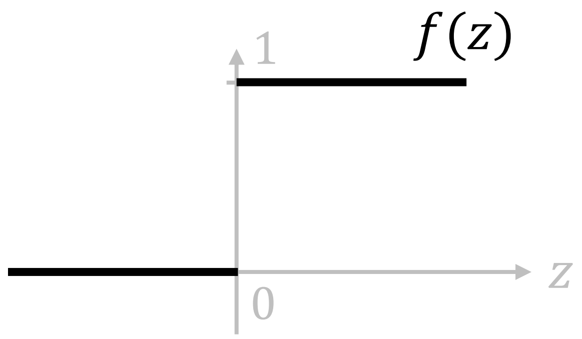
- The total loss over the training dataset is a sum/mean of individual losses over each data point

Activation functions f for hidden layers

Heaviside function

- Not differentiable

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



Activation functions f for hidden layers

Sigmoid function

$$f(z) = \frac{1}{1+e^{-z}}$$

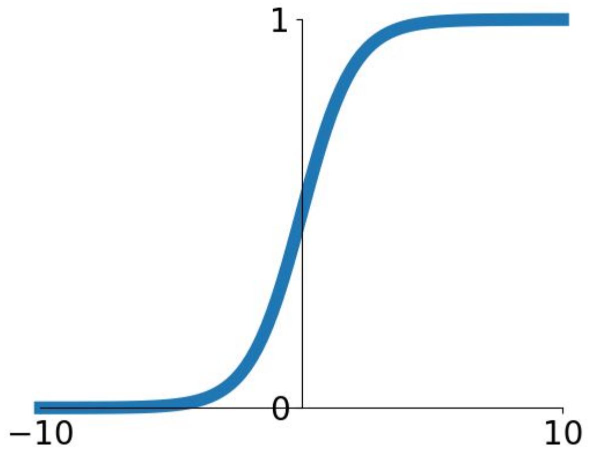
$$f'(z) = f(z)\big(1 - f(z)\big)$$

- Squashes numbers to range $(0,1)$

- Neurons can saturate:
  values close to 0 or 1
- Saturated neurons "kill" gradients
  = "vanishing gradients" issue

- Sigmoid outputs are not 0 centered
  - Ok for last layer (for binary classification)
  - Not for intermediate layers

- Exp is expensive to compute



Activation functions f for hidden layers

tanh function

$$f(z) = \tanh z = \frac{2}{1 + e^{-2z}} - 1$$

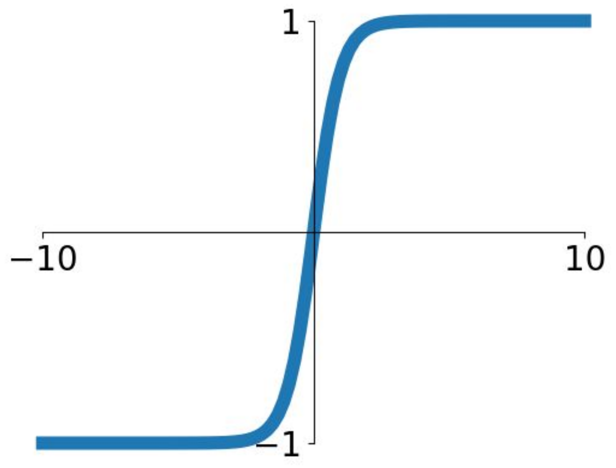
$$f'(z) = 1 - f(z)^2$$

- Rescaled and centered variant of sigmoid
- Squashes numbers to range $(-1,1)$

- tanh outputs are 0 centered
    - Good for intermediate layers

- Neurons can saturate:
  values close to -1 or 1
- Saturated neurons "kill" gradients
  = "vanishing gradients" issue

- tanh is expensive to compute



Activation functions f for hidden layers

ReLU (Rectified Linear Unit) function

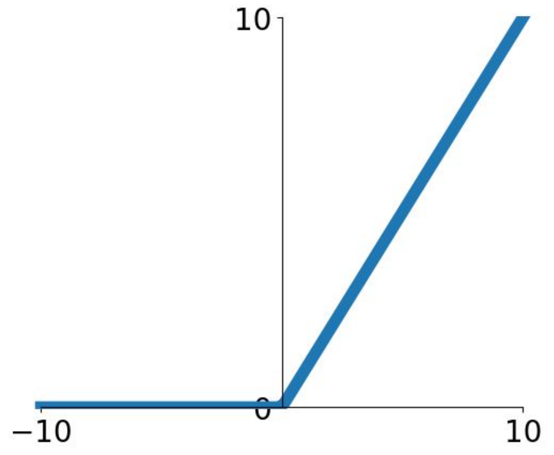
$$f(z) = \max(0, z)$$

$$f'(z) = \begin{cases} 0 \text{ for } z < 0 \\ 1 \text{ for } z \geq 0 \end{cases}$$

- Piecewise linear

- Does not saturate (positive values)
- Computationally cheap
- Converges faster than sigmoid/tanh in practice ($\approx 6\times$)

- Not zero-centered output
- "Dead neurons": no gradient to drive parameter update when input is below 0



Activation functions f for hidden layers

Leaky ReLU function

$$f(z) = \max(\alpha z, z)$$

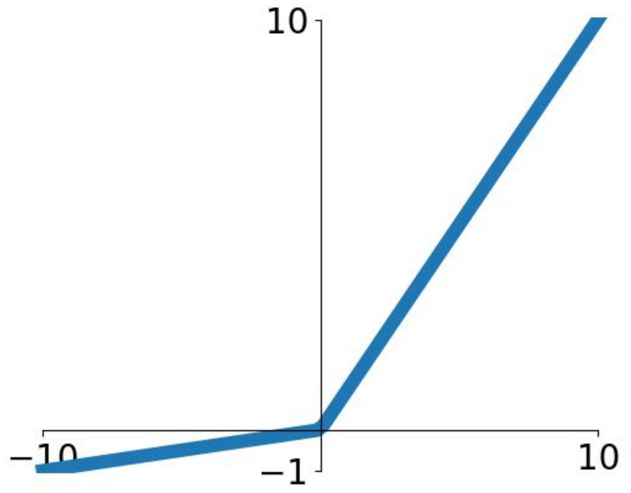
$$f'(z) = \begin{cases} \alpha \text{ for } z < 0 \\ 1 \text{ for } z \geq 0 \end{cases}$$

- Piecewise linear

- Does not saturate
- Computationally cheap
- Converges faster than sigmoid/tanh in practice ($\approx 6\times$)

- Will not die (no “dead neurons”)
- Closer to 0-mean outputs

- PReLU (Parametric ReLU) variant allows to learn slope α based on data



Activation functions f for hidden layers

ELU (Exponential Linear Unit) function

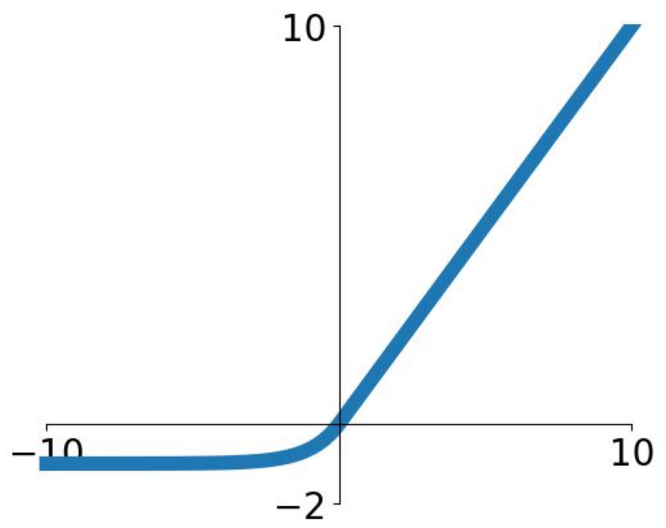
- Does not saturate (positive values)
- Computationally cheap in positive half

- Will not die (no “dead neurons”, although it does saturate in the negative half)
- Closer to 0-mean outputs

- Computationally expensive in negative half

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

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



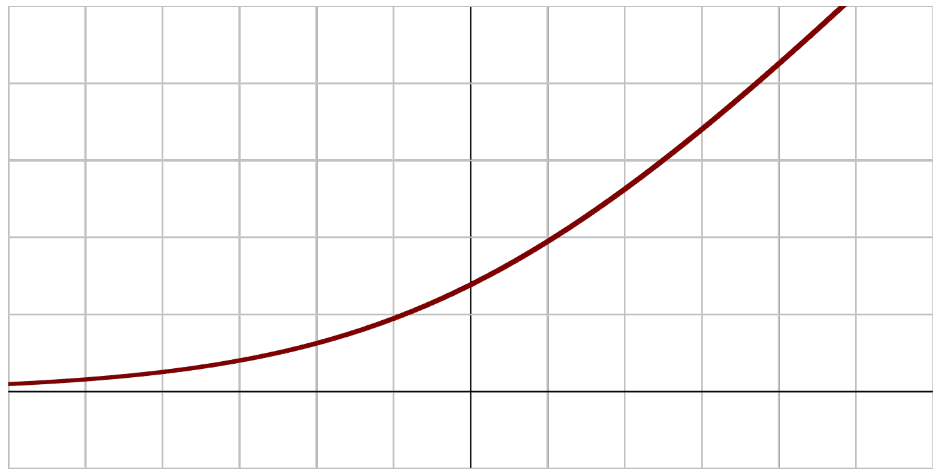
Activation functions f for hidden layers

Softplus function

- Range $(0, +\infty)$
- Useful as last layer when predicting
  a positive scalar (standard deviation, variance)
- Does not saturate (positive values)
- Will not die (no "dead neurons", although it does saturate in the negative half)
- Not 0-centered outputs
- Computationally expensive

$$f(z) = \ln(1 + e^{z})$$

$$f'(z) = \frac{1}{1 + e^{-z}}$$



Training Neural networks: backpropagation and computational graphs

Training neural networks

Define a loss function:

$$\mathcal{L}(\boldsymbol{\theta}) \triangleq \frac{1}{N}\sum_{n=1}^{N}\mathcal{L}(\boldsymbol{y}^{(\boldsymbol{n})}, \hat{\boldsymbol{y}}(\boldsymbol{x}^{(\boldsymbol{n})}, \boldsymbol{\theta}))$$

Optimize the loss, based on gradient descent:

```
repeat until convergence:
        $\boldsymbol{\theta}^{t+1} \coloneqq \boldsymbol{\theta}^{t} - \alpha \cdot \boldsymbol{\nabla}_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}^{t})$
```

Training neural networks

Define a loss function:

$$\mathcal{L}(\boldsymbol{\theta}) \triangleq \frac{1}{N}\sum_{n=1}^{N} \mathcal{L}(\boldsymbol{y}^{(\boldsymbol{n})}, \widehat{\boldsymbol{y}}(\boldsymbol{x}^{(\boldsymbol{n})}, \boldsymbol{\theta}))$$

Optimize the loss, based on gradient descent:

```
repeat until convergence:
```

$$\boldsymbol{\theta}^{t+1} \coloneqq \boldsymbol{\theta}^{t} - \alpha \cdot \boxed{\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^{t})}$$

How to compute the $\frac{\partial \mathcal{L}}{\partial \theta_i}(\boldsymbol{\theta}^t)$ for all parameters i?

From the chain rule to forward and backward passes

Consider $\theta \mapsto f(\theta)$, $y \mapsto g(y)$, $z \mapsto h(z)$ and a function $\mathcal{L}(\theta) \triangleq h(g(f(\theta)))$.

Suppose we want to compute $\frac{\partial \mathcal{L}}{\partial \theta}(\theta^*)$

By the chain rule,

$$\frac{\partial \mathcal{L}}{\partial \theta}(\theta^*) = \frac{\partial h}{\partial z}(g(f(\theta^*))) \cdot \frac{\partial g}{\partial y}(f(\theta^*)) \cdot \frac{\partial f}{\partial \theta}(\theta^*)$$

- We need to compute, in order, from θ^*: $y^* \coloneqq f(\theta^*)$, $z^* \coloneqq g(y^*)$ → **"forward pass"**
- **"Backward pass"**: we can then compute, in order:
  1. $\frac{\partial h}{\partial z}(z^*)$
  2. $\frac{\partial h \circ g}{\partial y}(y^*) = \frac{\partial h}{\partial z}(z^*) \cdot \frac{\partial g}{\partial y}(y^*)$
  3. $\frac{\partial \mathcal{L}}{\partial \theta}(\theta^*) = \frac{\partial h \circ g}{\partial y}(y^*) \cdot \frac{\partial f}{\partial \theta}(\theta^*)$

From the chain rule to forward and backward passes

Consider $\theta \mapsto f(\theta)$, $y \mapsto g(y)$, $z \mapsto h(z)$ and a function $\mathcal{L}(\theta) \triangleq h(g(f(\theta)))$.

Suppose we want to compute $\frac{\partial \mathcal{L}}{\partial \theta}(\theta^*)$

By abuse of notation, we also write:

- $\mathcal{L}(z) \triangleq h(z)$
- $\mathcal{L}(y) \triangleq h(g(y))$

By the chain rule,

$$\frac{\partial \mathcal{L}}{\partial \theta}(\theta^*) = \frac{\partial h}{\partial z}(g(f(\theta^*))) \cdot \frac{\partial g}{\partial y}(f(\theta^*)) \cdot \frac{\partial f}{\partial \theta}(\theta^*)$$

- We need to compute, in order, from θ^*: $y^* \coloneqq f(\theta^*)$, $z^* \coloneqq g(y^*)$ → **"forward pass"**
- **"Backward pass"**: we can then compute, in order:
  1. $\frac{\partial h}{\partial z}(z^*)$
  2. $\frac{\partial h \circ g}{\partial y}(y^*) = \frac{\partial h}{\partial z}(z^*) \cdot \frac{\partial g}{\partial y}(y^*)$
  3. $\frac{\partial \mathcal{L}}{\partial \theta}(\theta^*) = \frac{\partial h \circ g}{\partial y}(y^*) \cdot \frac{\partial f}{\partial \theta}(\theta^*)$

From the chain rule to forward and backward passes

Consider $\theta \mapsto f(\theta)$, $y \mapsto g(y)$, $z \mapsto h(z)$ and a function $\mathcal{L}(\theta) \triangleq h(g(f(\theta)))$.

Suppose we want to compute $\frac{\partial \mathcal{L}}{\partial \theta}(\theta^*)$

By abuse of notation, we also write:

- $\mathcal{L}(z) \triangleq h(z)$
- $\mathcal{L}(y) \triangleq h(g(y))$

By the chain rule,

$$\frac{\partial \mathcal{L}}{\partial \theta}(\theta^*) = \frac{\partial h}{\partial z}(g(f(\theta^*))) \cdot \frac{\partial g}{\partial y}(f(\theta^*)) \cdot \frac{\partial f}{\partial \theta}(\theta^*)$$

- We need to compute, in order, from θ^*: $y^* \coloneqq f(\theta^*)$, $z^* \coloneqq g(y^*)$ → **"forward pass"**
- **"Backward pass"**: we can then compute, in order:
  1. $\frac{\partial h}{\partial z}(z^*)$ → we also note this quantity $\frac{\partial \mathcal{L}}{\partial z}$
  2. $\frac{\partial h \circ g}{\partial y}(y^*) = \frac{\partial h}{\partial z}(z^*) \cdot \frac{\partial g}{\partial y}(y^*)$ → we also note this quantity $\frac{\partial \mathcal{L}}{\partial y} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial y}$
  3. $\frac{\partial \mathcal{L}}{\partial \theta}(\theta^*) = \frac{\partial h \circ g}{\partial y}(y^*) \cdot \frac{\partial f}{\partial \theta}(\theta^*)$ → $\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial \theta}$

From the chain rule to forward and backward passes

Consider $\theta \mapsto f(\theta)$, $y \mapsto g(y)$, $z \mapsto h(z)$ and a function $\mathcal{L}(\theta) \triangleq h(g(f(\theta)))$.

By abuse of notation, we also write:

- $\mathcal{L}(z) \triangleq h(z)$
- $\mathcal{L}(y) \triangleq h(g(y))$

Suppose we want to compute $\frac{\partial \mathcal{L}}{\partial \theta}(\theta^*)$

By the chain rule,

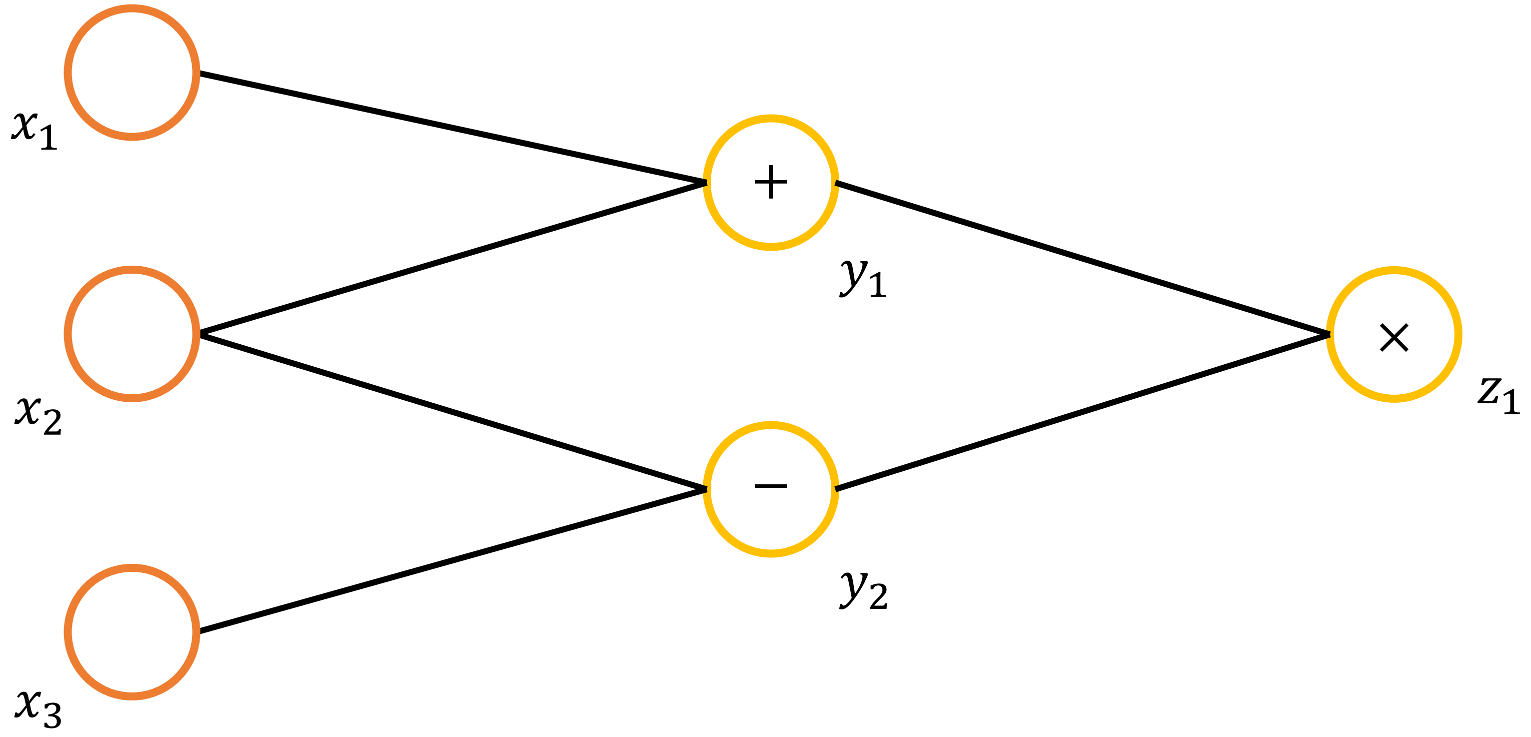
$$\frac{\partial \mathcal{L}}{\partial \theta}(\theta^*) = \frac{\partial h}{\partial z}(g(f(\theta^*))) \cdot \frac{\partial g}{\partial y}(f(\theta^*)) \cdot \frac{\partial f}{\partial \theta}(\theta^*)$$

- We need to compute, in order, from θ^*: $y^* \coloneqq f(\theta^*)$, $z^* \coloneqq g(y^*)$ $\to$ **"forward pass"**
- **"Backward pass"**: we can then compute, in order:

$$\left.\begin{array}{lll} 1. & \frac{\partial h}{\partial z}(z^*) & \to \text{we also note this quantity } \frac{\partial \mathcal{L}}{\partial z} \\ 2. & \frac{\partial h \circ g}{\partial y}(y^*) = \frac{\partial h}{\partial z}(z^*) \cdot \frac{\partial g}{\partial y}(y^*) & \to \text{we also note this quantity } \frac{\partial \mathcal{L}}{\partial y} \\ 3. & \frac{\partial \mathcal{L}}{\partial \theta}(\theta^*) = \frac{\partial h \circ g}{\partial y}(y^*) \cdot \frac{\partial f}{\partial \theta}(\theta^*) & \to \frac{\partial \mathcal{L}}{\partial \theta} \end{array}\right\}$$

Compute the derivative of the output w.r.t. all intermediate variables, starting from the "deepest" variable

Computational graphs, an example

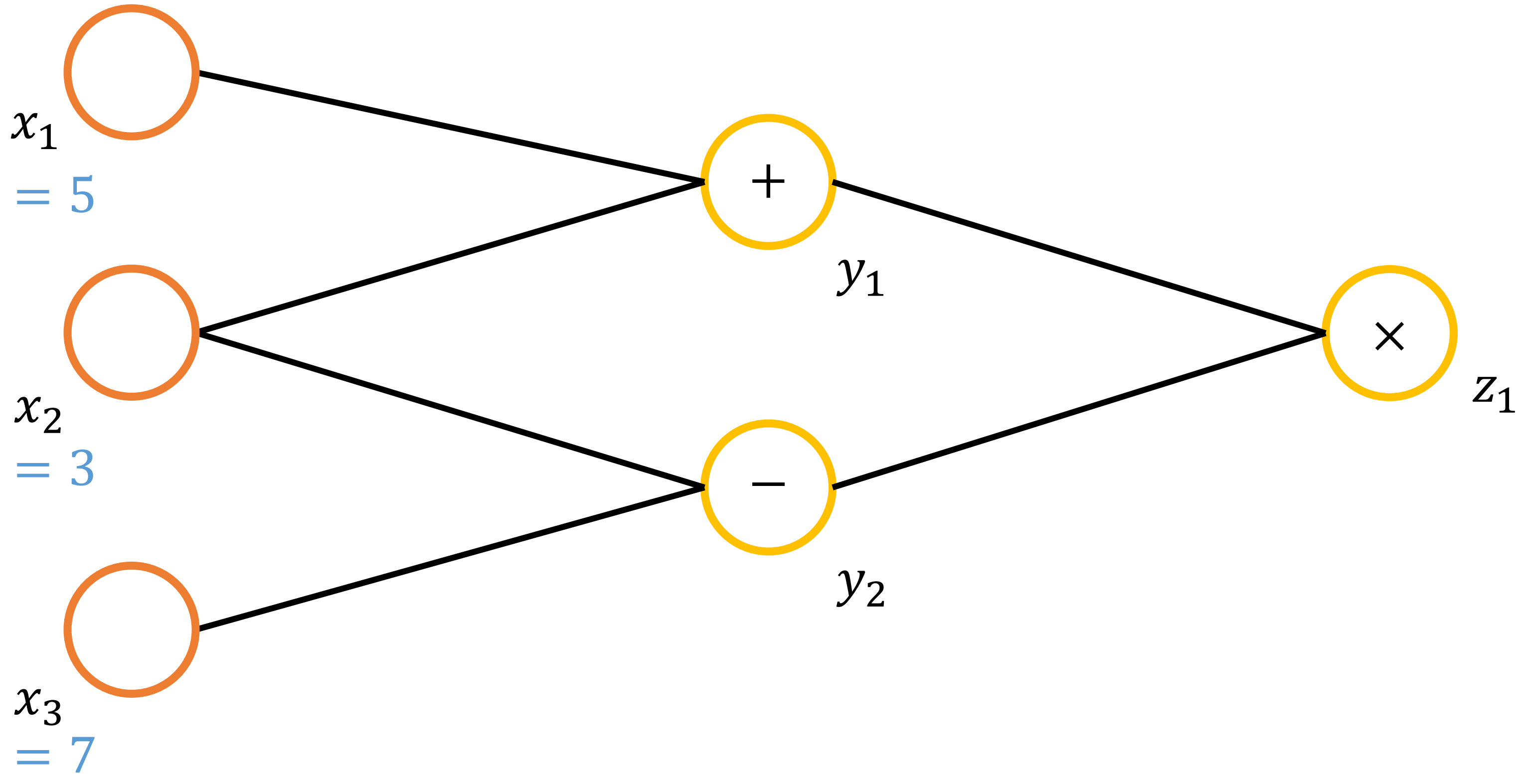


$$z_1 = y_1 \times y_2 = (x_1 + x_2) \times (x_2 - x_3) = x_1x_2 - x_1x_3 + x_2^2 - x_2x_3$$

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

Computational graphs, an example

forward pass

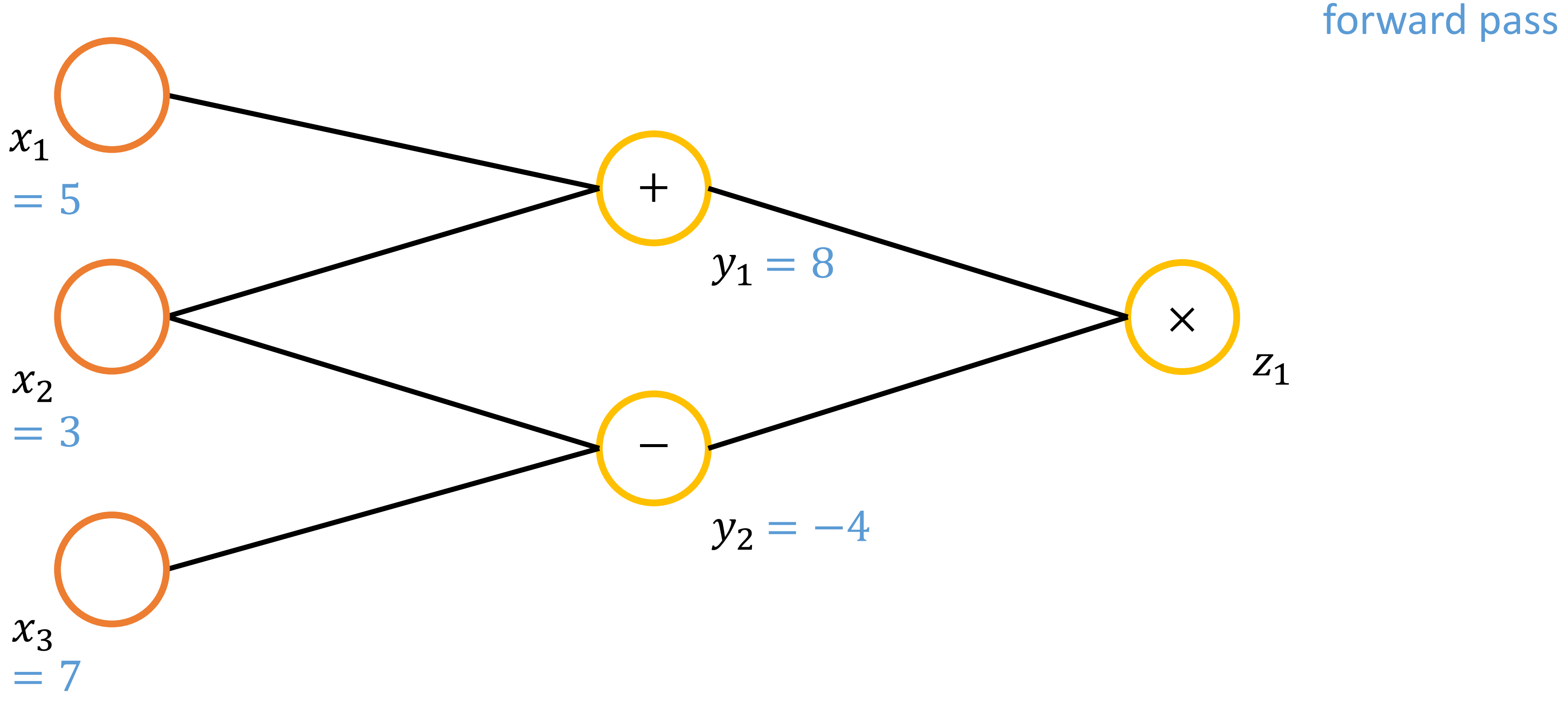


$$z_1 = y_1 \times y_2 = (x_1 + x_2) \times (x_2 - x_3) = x_1x_2 - x_1x_3 + x_2^2 - x_2x_3$$

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

Inspired by D. Rueckert's slides

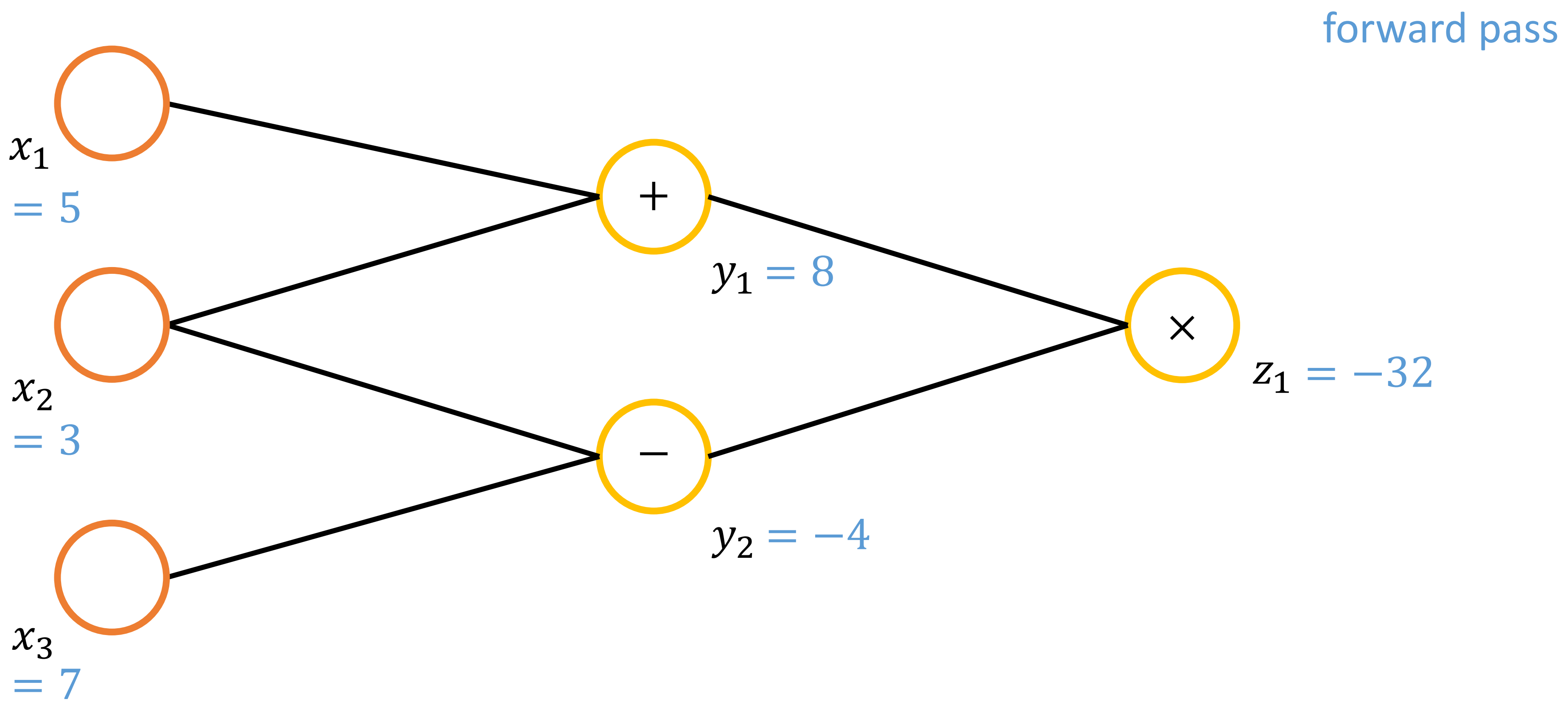
Computational graphs, an example



$$z_1 = y_1 \times y_2 = (x_1 + x_2) \times (x_2 - x_3) = x_1x_2 - x_1x_3 + x_2^2 - x_2x_3$$

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

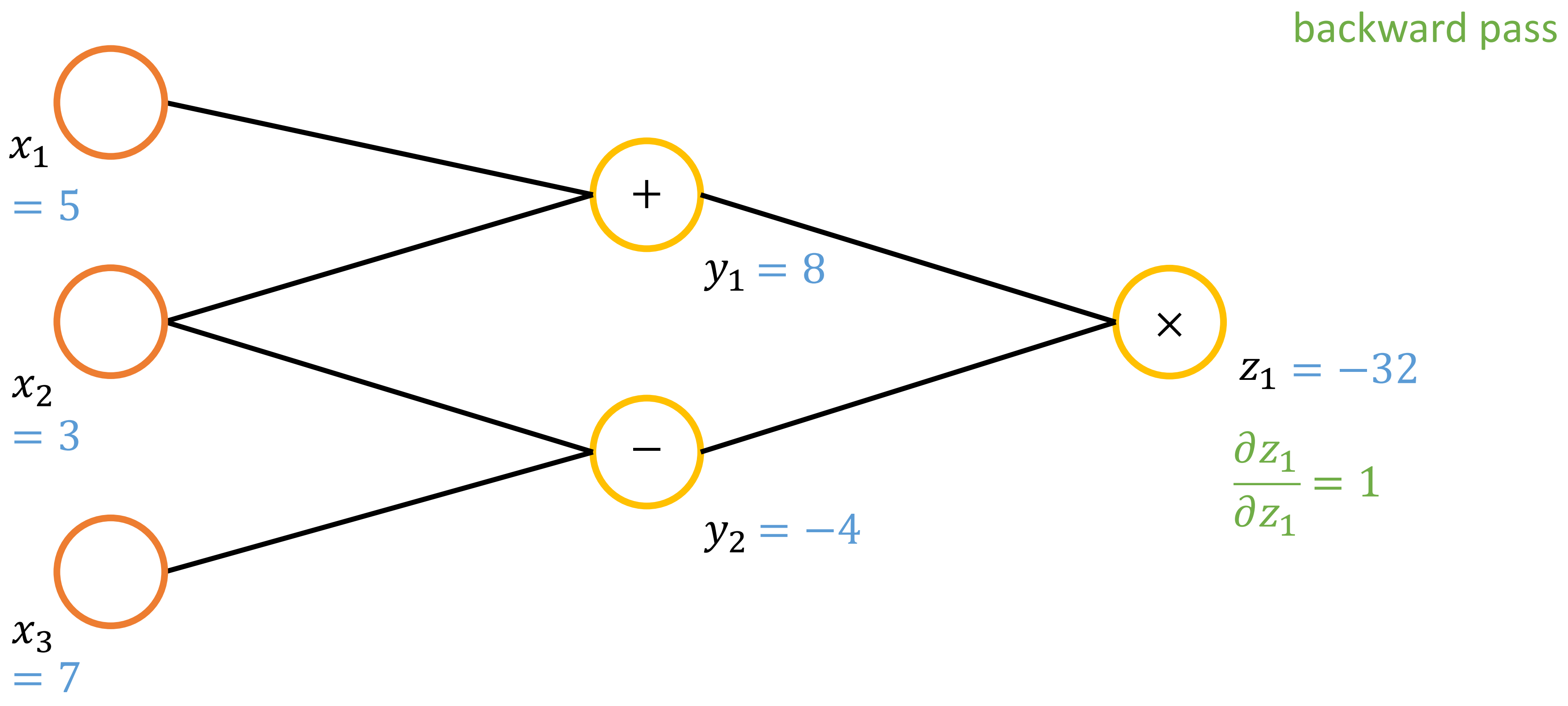
Computational graphs, an example



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

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

Computational graphs, an example



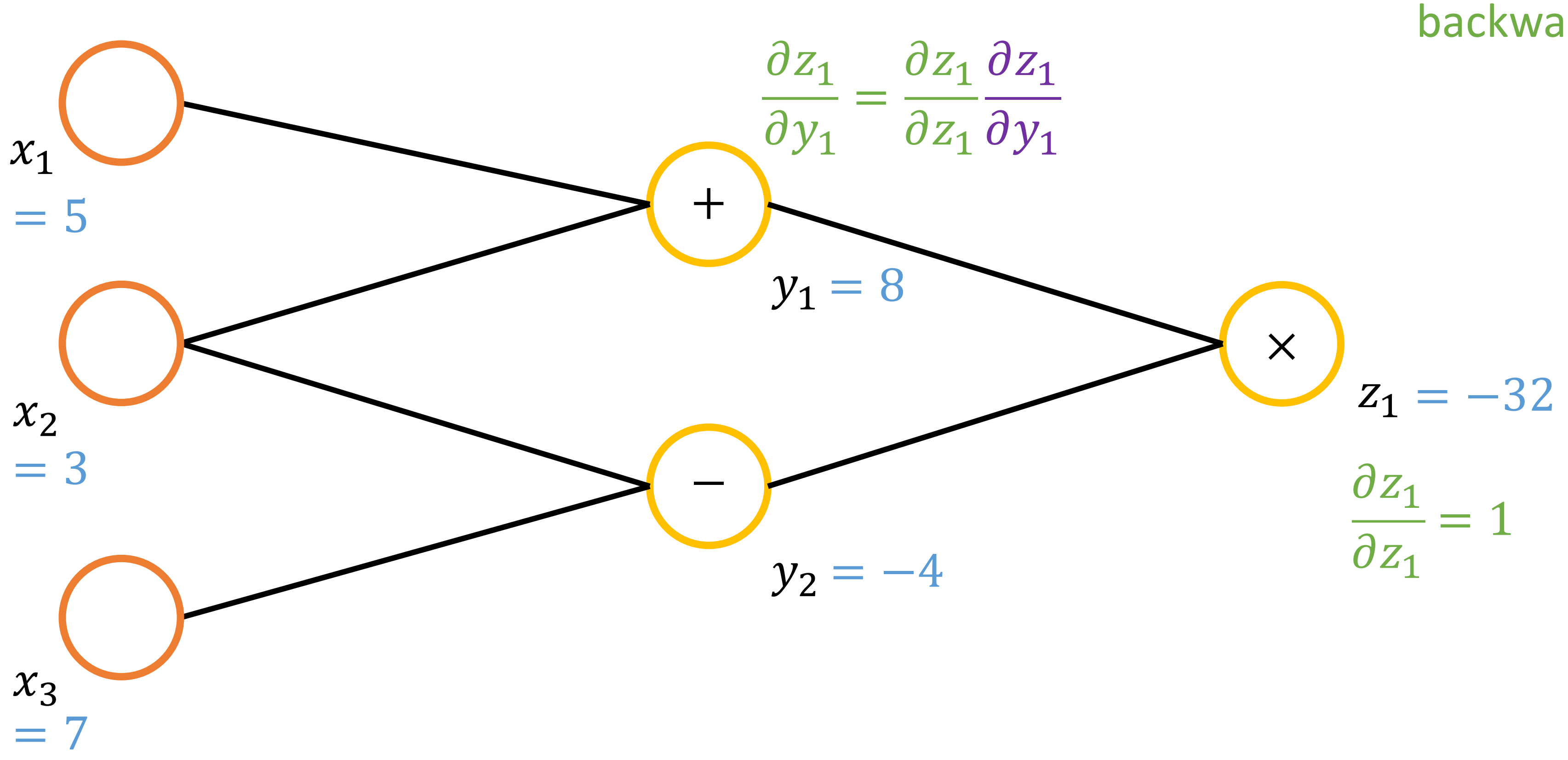
$$z_1 = y_1 \times y_2 = (x_1 + x_2) \times (x_2 - x_3) = x_1 x_2 - x_1 x_3 + x_2^2 - x_2 x_3$$

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

Inspired by D. Rueckert's slides

Computational graphs, an example

backward pass



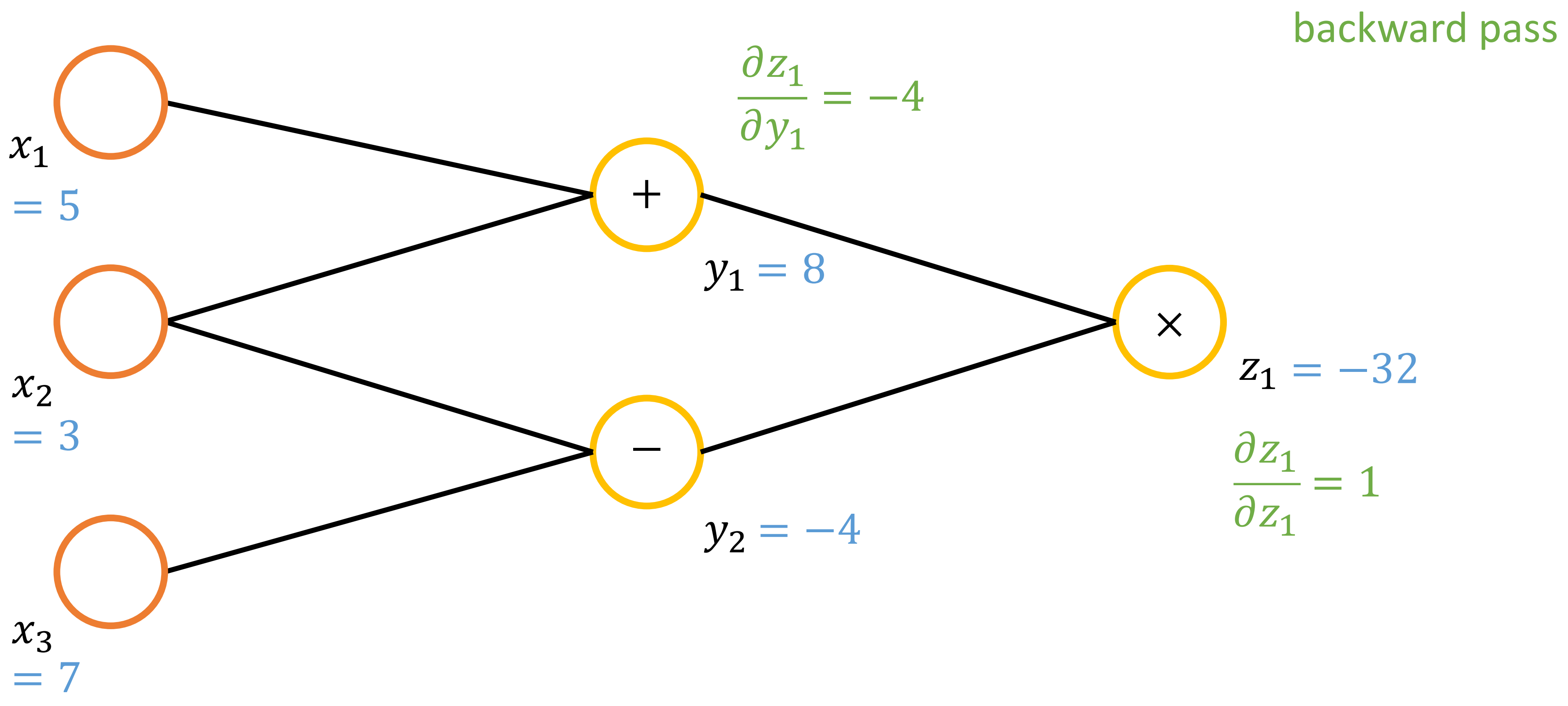
$\frac{\partial z_1}{\partial y_1} = \frac{\partial z_1}{\partial z_1}\frac{\partial z_1}{\partial y_1}$
$x_1 = 5$
$x_2 = 3$
$x_3 = 7$
$+$
$-$
$\times$
$y_1 = 8$
$y_2 = -4$
$z_1 = -32$
$\frac{\partial z_1}{\partial z_1} = 1$

$$z_1 = y_1 \times y_2 = (x_1 + x_2) \times (x_2 - x_3) = x_1x_2 - x_1x_3 + x_2^2 - x_2x_3$$

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

Inspired by D. Rueckert's slides

Computational graphs, an example



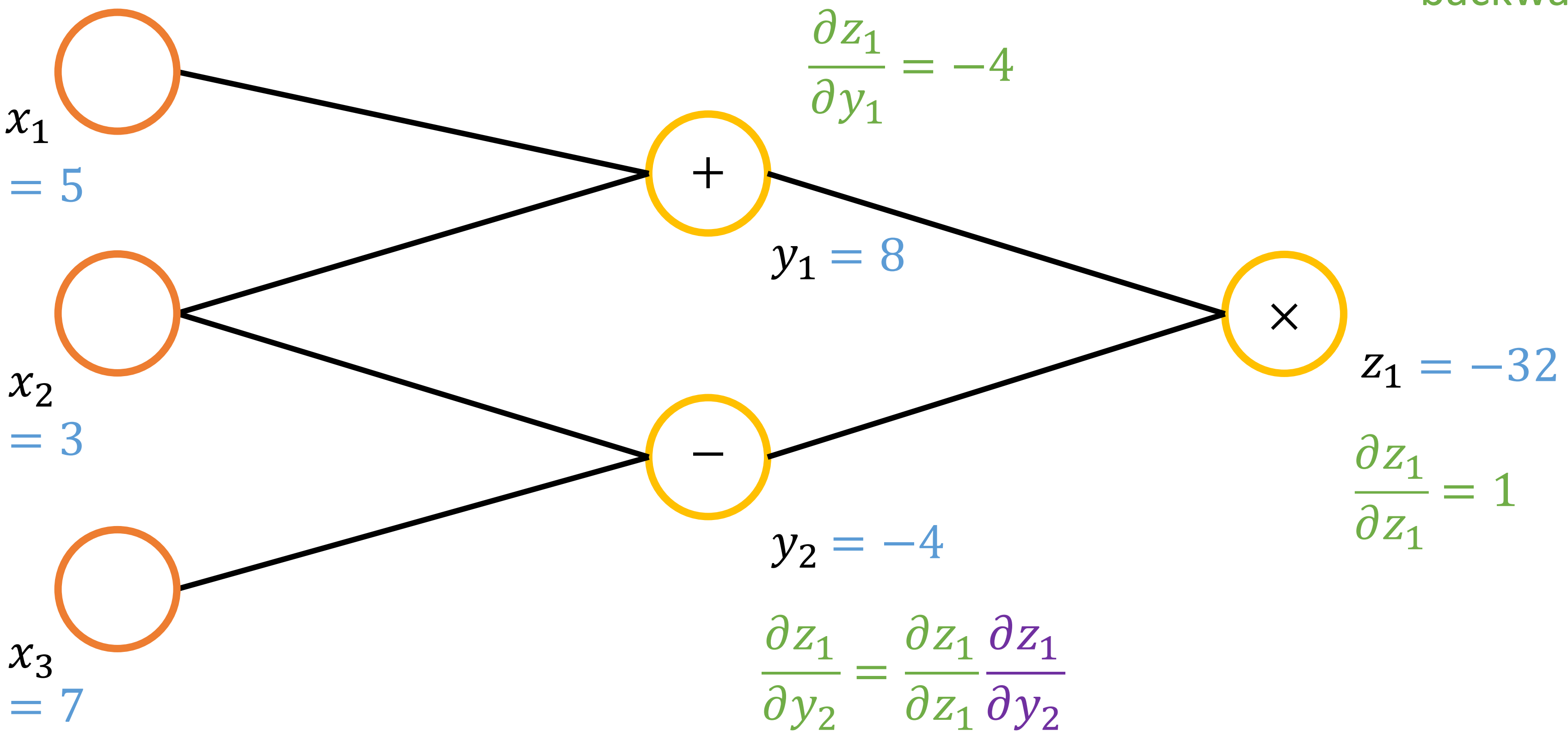
$$z_1 = y_1 \times y_2 = (x_1 + x_2) \times (x_2 - x_3) = x_1 x_2 - x_1 x_3 + x_2^2 - x_2 x_3$$

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

Inspired by D. Rueckert's slides

Computational graphs, an example

backward pass



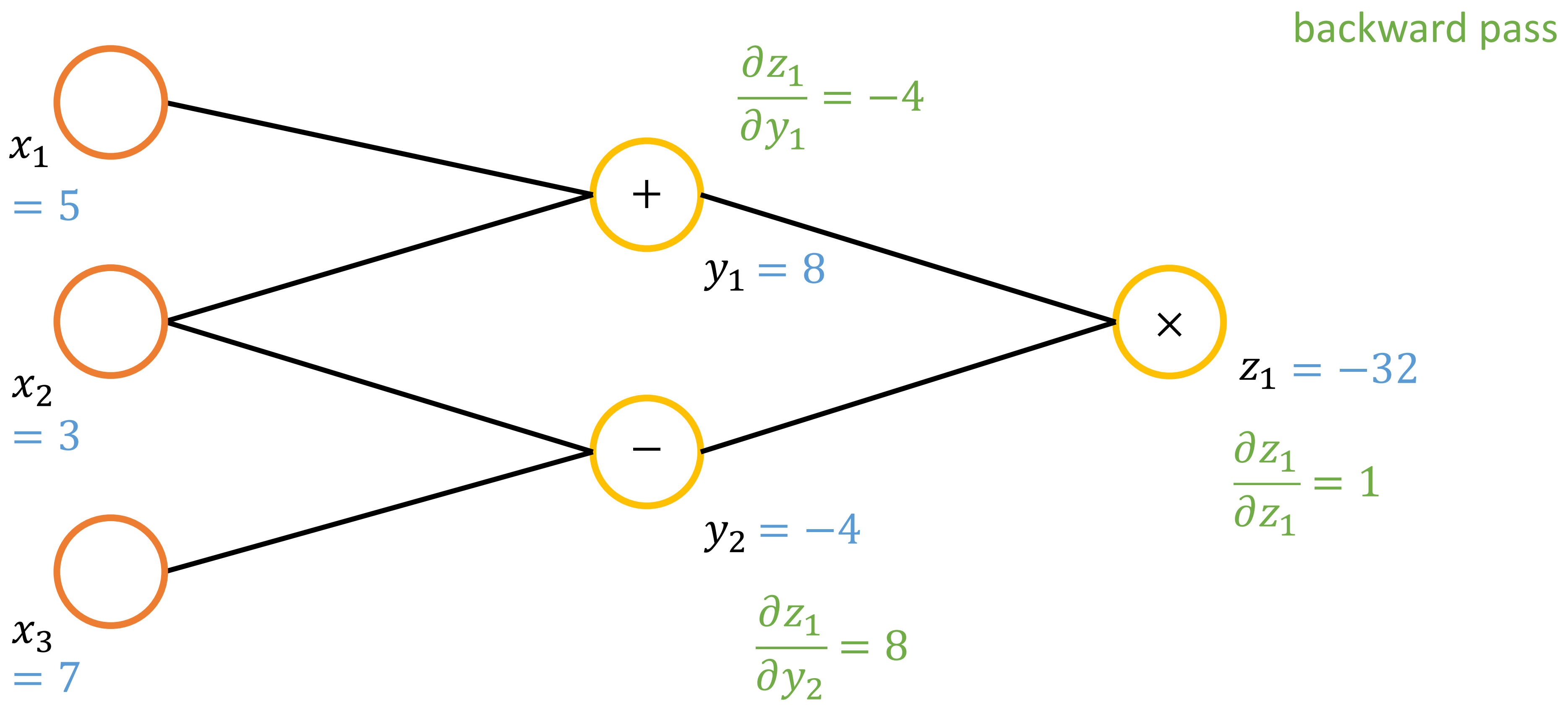
$x_1 = 5$
$x_2 = 3$
$x_3 = 7$
$+$
$-$
$\times$
$y_1 = 8$
$y_2 = -4$
$z_1 = -32$
$\frac{\partial z_1}{\partial y_1} = -4$
$\frac{\partial z_1}{\partial z_1} = 1$
$\frac{\partial z_1}{\partial y_2} = \frac{\partial z_1}{\partial z_1}\frac{\partial z_1}{\partial y_2}$

$$z_1 = y_1 \times y_2 = (x_1 + x_2) \times (x_2 - x_3) = x_1x_2 - x_1x_3 + x_2^2 - x_2x_3$$

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

Inspired by D. Rueckert's slides

Computational graphs, an example

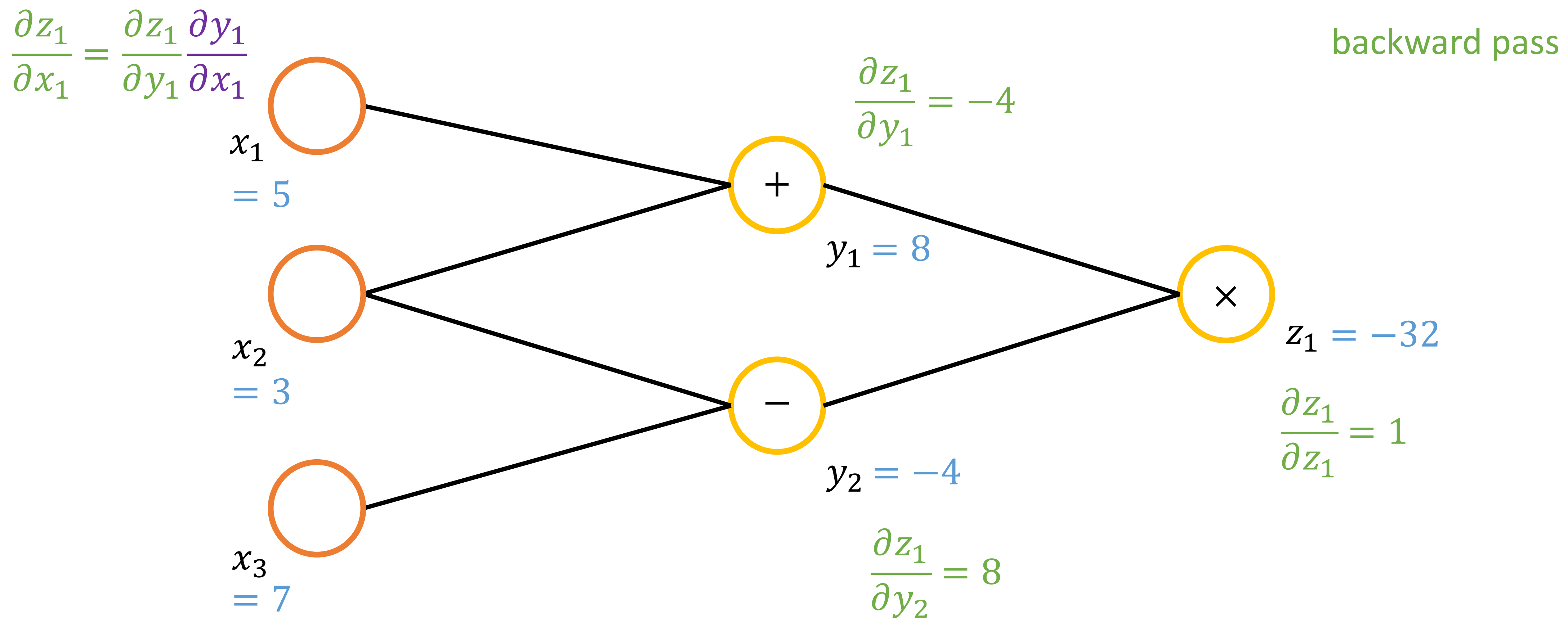

backward pass
$\frac{\partial z_1}{\partial y_1} = -4$
$x_1 = 5$
$+$
$y_1 = 8$
$x_2 = 3$
$\times$
$z_1 = -32$
$\frac{\partial z_1}{\partial z_1} = 1$
$-$
$y_2 = -4$
$x_3 = 7$
$\frac{\partial z_1}{\partial y_2} = 8$

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

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

Inspired by D. Rueckert's slides

Computational graphs, an example



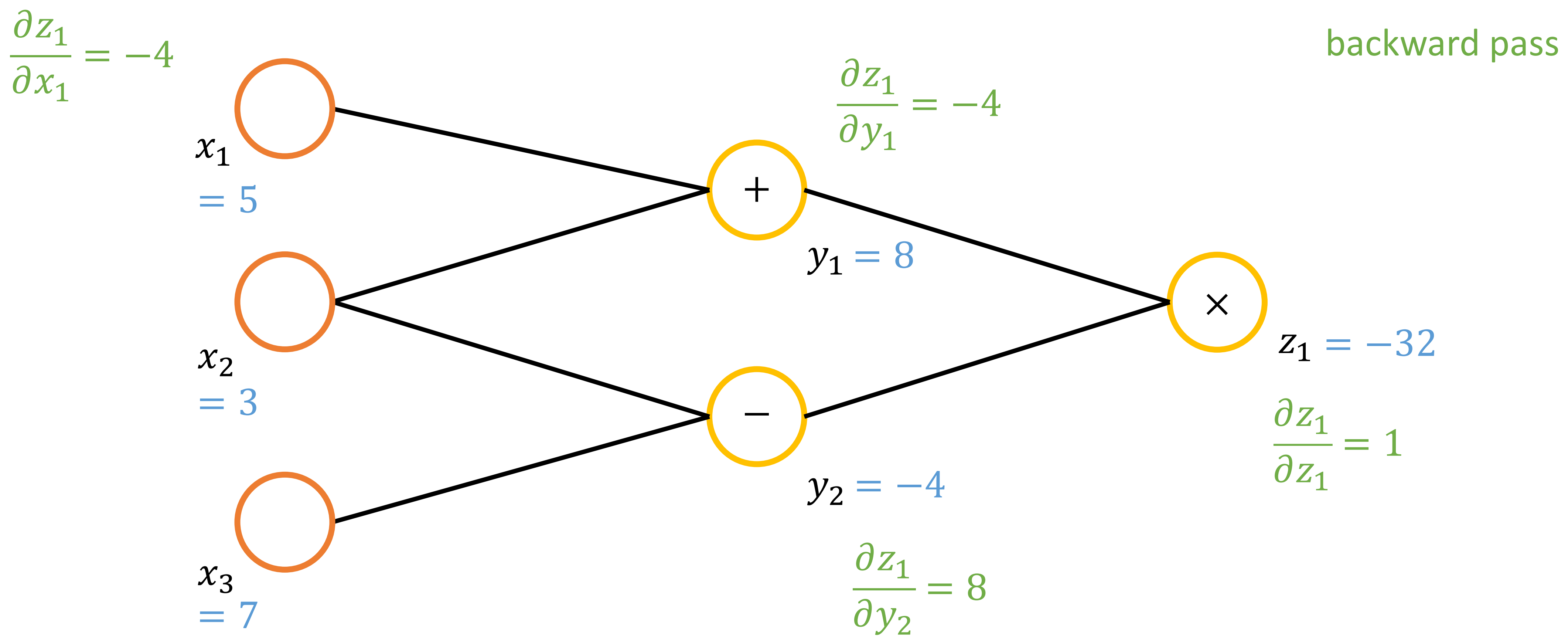
$\frac{\partial z_1}{\partial x_1} = \frac{\partial z_1}{\partial y_1}\frac{\partial y_1}{\partial x_1}$
backward pass
$\frac{\partial z_1}{\partial y_1} = -4$
$x_1 = 5$
$+$
$y_1 = 8$
$x_2 = 3$
$\times$
$z_1 = -32$
$-$
$\frac{\partial z_1}{\partial z_1} = 1$
$y_2 = -4$
$x_3 = 7$
$\frac{\partial z_1}{\partial y_2} = 8$

$$z_1 = y_1 \times y_2 = (x_1 + x_2) \times (x_2 - x_3) = x_1x_2 - x_1x_3 + x_2^2 - x_2x_3$$

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

Inspired by D. Rueckert's slides

Computational graphs, an example



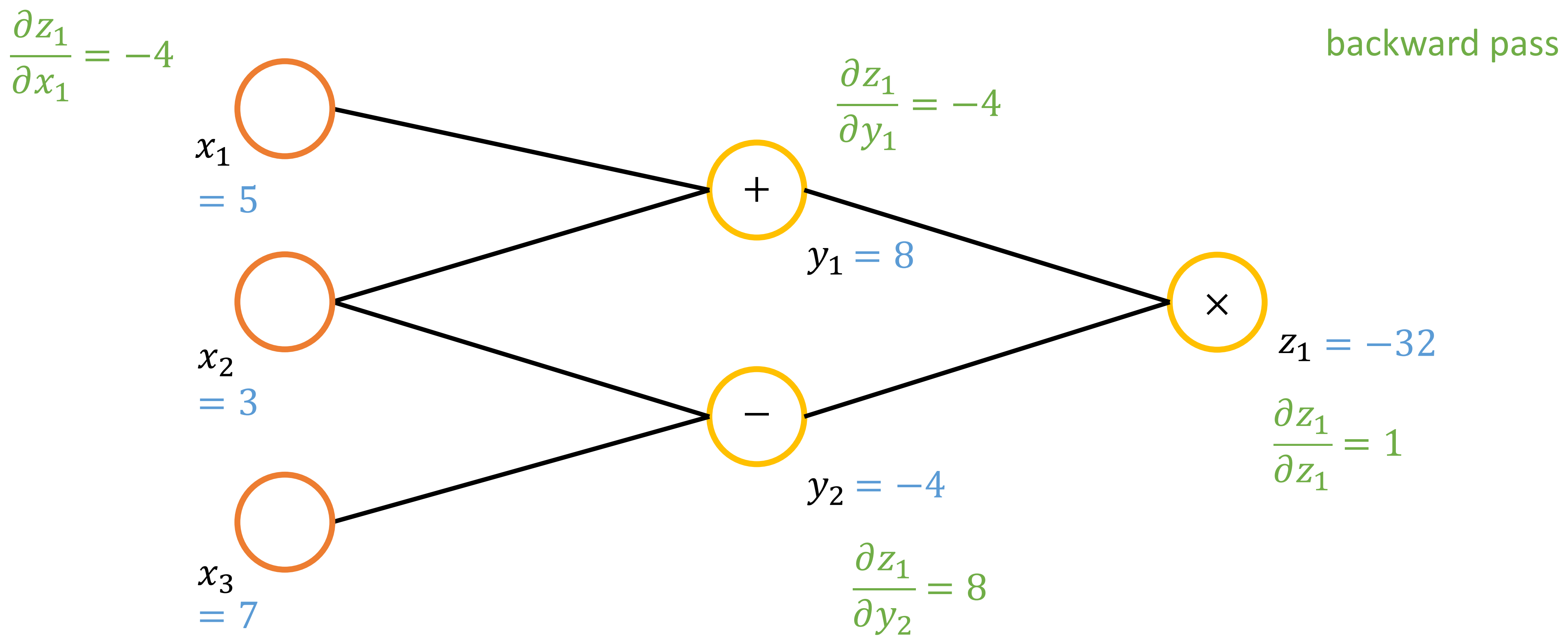
$\frac{\partial z_1}{\partial x_1} = -4$
backward pass
$\frac{\partial z_1}{\partial y_1} = -4$
x_1
$= 5$
$+$
$y_1 = 8$
$\times$
$z_1 = -32$
x_2
$= 3$
$-$
$\frac{\partial z_1}{\partial z_1} = 1$
$y_2 = -4$
x_3
$= 7$
$\frac{\partial z_1}{\partial y_2} = 8$

$$z_1 = y_1 \times y_2 = (x_1 + x_2) \times (x_2 - x_3) = x_1x_2 - x_1x_3 + x_2^2 - x_2x_3$$

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

Inspired by D. Rueckert's slides

Computational graphs, an example


backward pass
$\frac{\partial z_1}{\partial x_1} = -4$
$x_1 = 5$
$x_2 = 3$
$x_3 = 7$
$+$
$-$
$\times$
$\frac{\partial z_1}{\partial y_1} = -4$
$y_1 = 8$
$y_2 = -4$
$\frac{\partial z_1}{\partial y_2} = 8$
$z_1 = -32$
$\frac{\partial z_1}{\partial z_1} = 1$

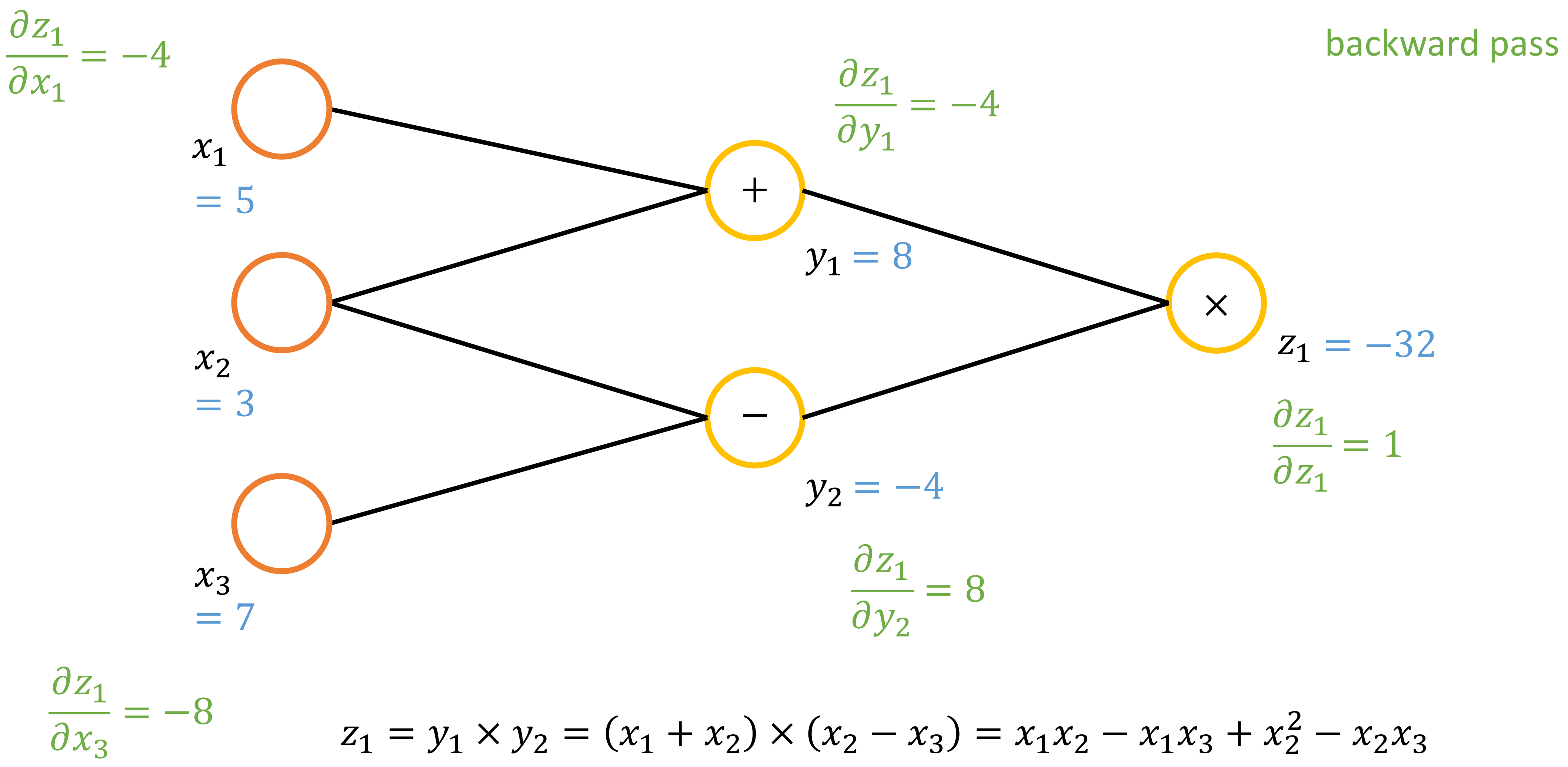
$$\frac{\partial z_1}{\partial x_3} = \frac{\partial z_1}{\partial y_2}\frac{\partial y_2}{\partial x_3}$$

$$z_1 = y_1 \times y_2 = (x_1 + x_2) \times (x_2 - x_3) = x_1x_2 - x_1x_3 + x_2^2 - x_2x_3$$

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

Inspired by D. Rueckert's slides

Computational graphs, an example

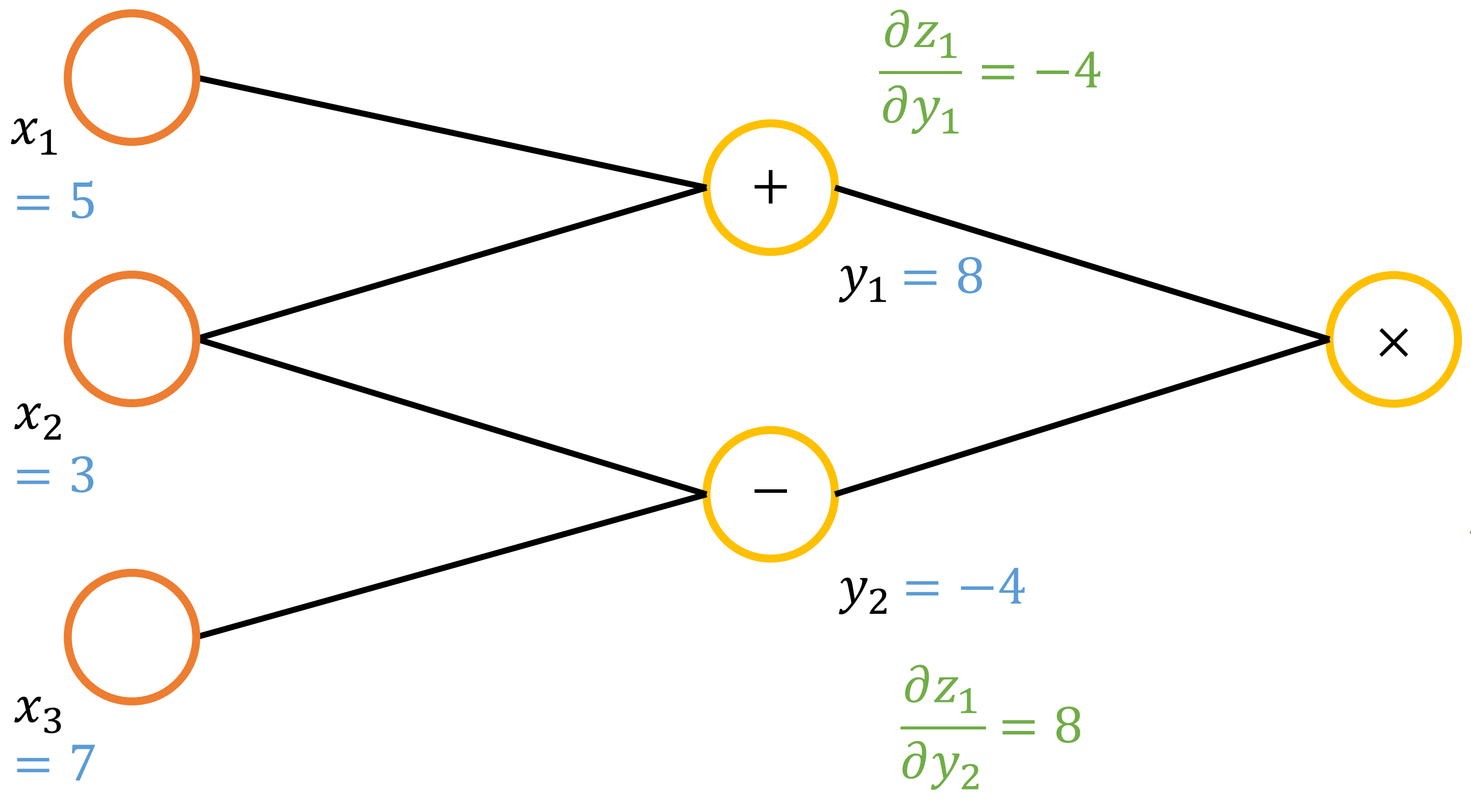

$\frac{\partial z_1}{\partial x_1} = -4$
$x_1 = 5$
$x_2 = 3$
$x_3 = 7$
$\frac{\partial z_1}{\partial x_3} = -8$
$+$
$-$
$y_1 = 8$
$y_2 = -4$
$\frac{\partial z_1}{\partial y_1} = -4$
$\frac{\partial z_1}{\partial y_2} = 8$
$\times$
$z_1 = -32$
$\frac{\partial z_1}{\partial z_1} = 1$
backward pass

$$z_1 = y_1 \times y_2 = (x_1 + x_2) \times (x_2 - x_3) = x_1x_2 - x_1x_3 + x_2^2 - x_2x_3$$

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

Inspired by D. Rueckert's slides

Computational graphs, an example



backward pass

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

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

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

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

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

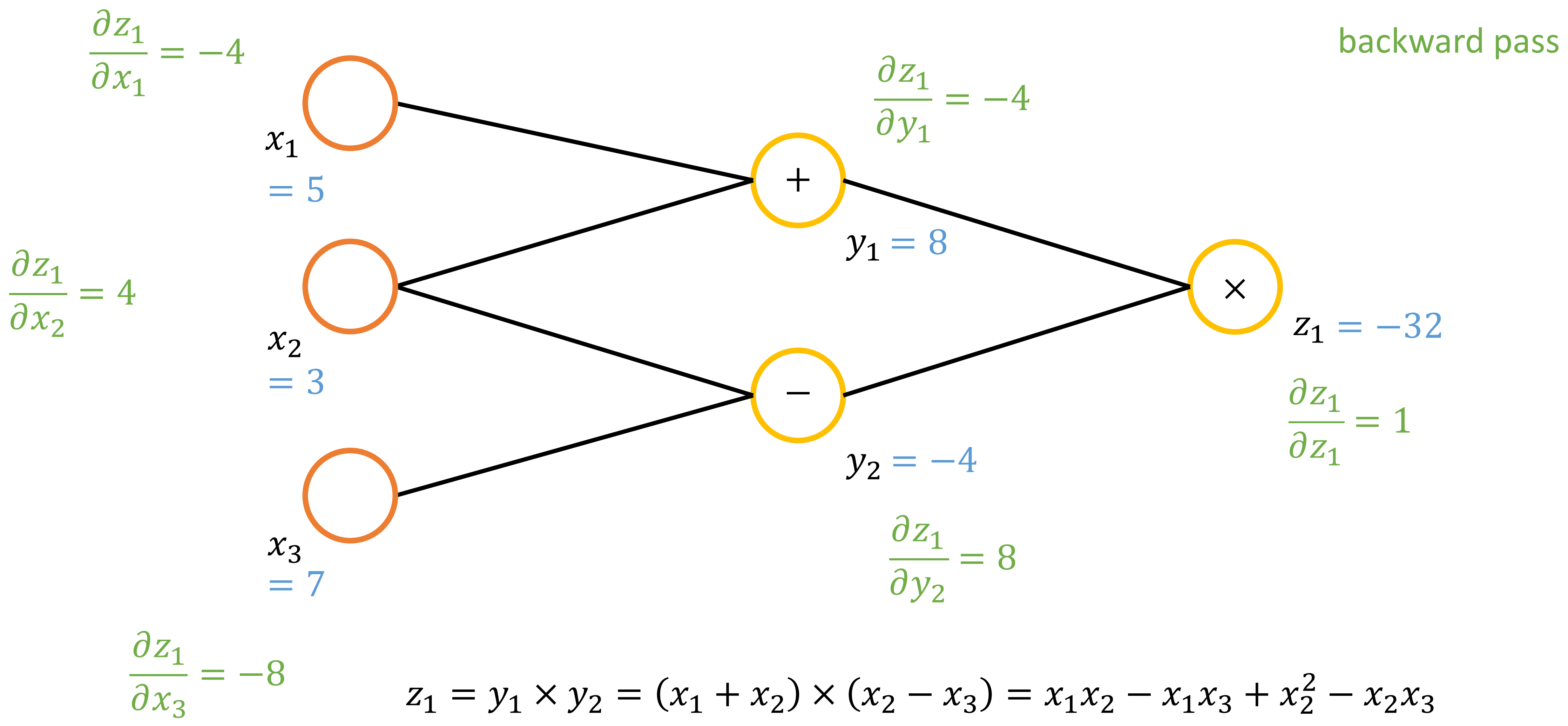
$$\frac{\partial z_1}{\partial x_3} = -8$$

$$z_1 = y_1 \times y_2 = (x_1 + x_2) \times (x_2 - x_3) = x_1x_2 - x_1x_3 + x_2^2 - x_2x_3$$

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

Inspired by D. Rueckert's slides

Computational graphs, an example



$\frac{\partial z_1}{\partial x_1} = -4$
$\frac{\partial z_1}{\partial x_2} = 4$
$\frac{\partial z_1}{\partial x_3} = -8$
$x_1 = 5$
$x_2 = 3$
$x_3 = 7$
$+$
$-$
$\times$
$\frac{\partial z_1}{\partial y_1} = -4$
$y_1 = 8$
$y_2 = -4$
$\frac{\partial z_1}{\partial y_2} = 8$
$z_1 = -32$
$\frac{\partial z_1}{\partial z_1} = 1$
backward pass

$$z_1 = y_1 \times y_2 = (x_1 + x_2) \times (x_2 - x_3) = x_1x_2 - x_1x_3 + x_2^2 - x_2x_3$$

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

Inspired by D. Rueckert's slides

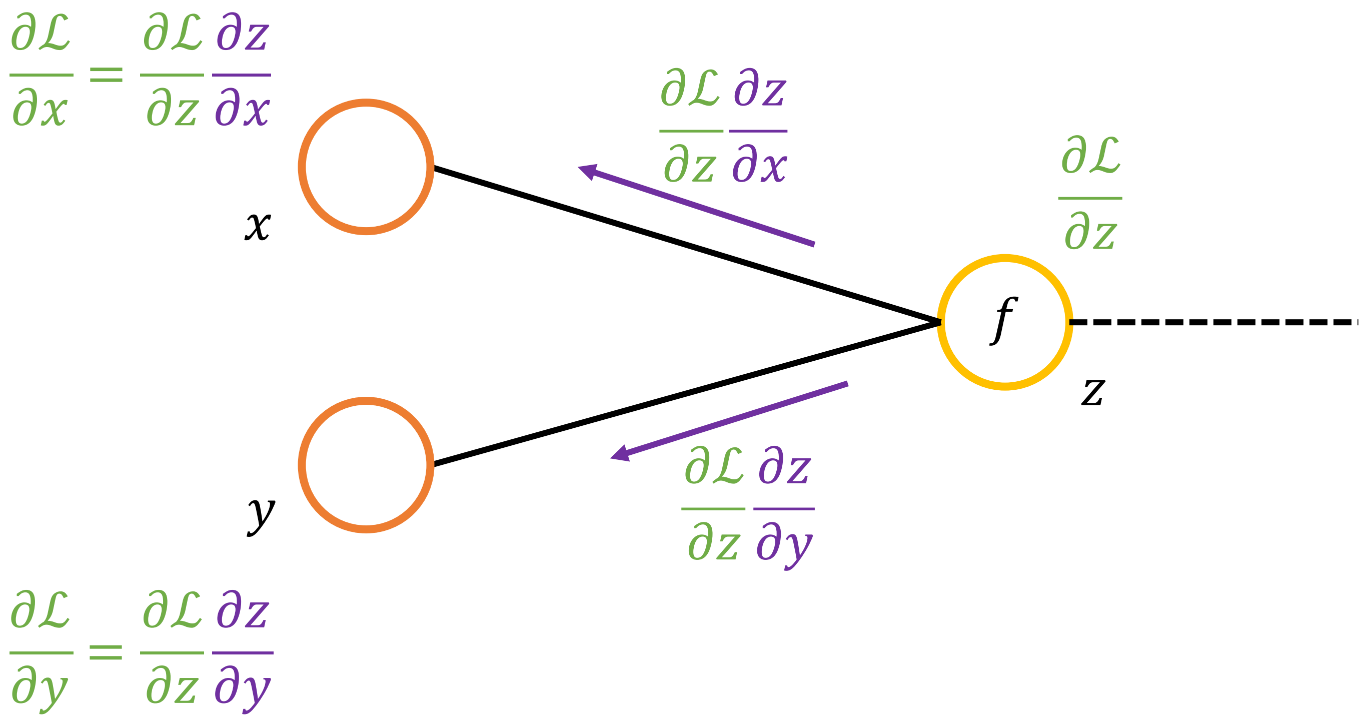
Computational graphs

- The input variables, the learnable parameters are nodes
- The results of elementary operations between nodes are nodes

- Nodes store
  - `node.value`, the result of the operation, computed in the **forward pass**
  - `node.grad`, the gradient of the final output w.r.t. the node variable (at `node.value`), computed in the **backward pass**

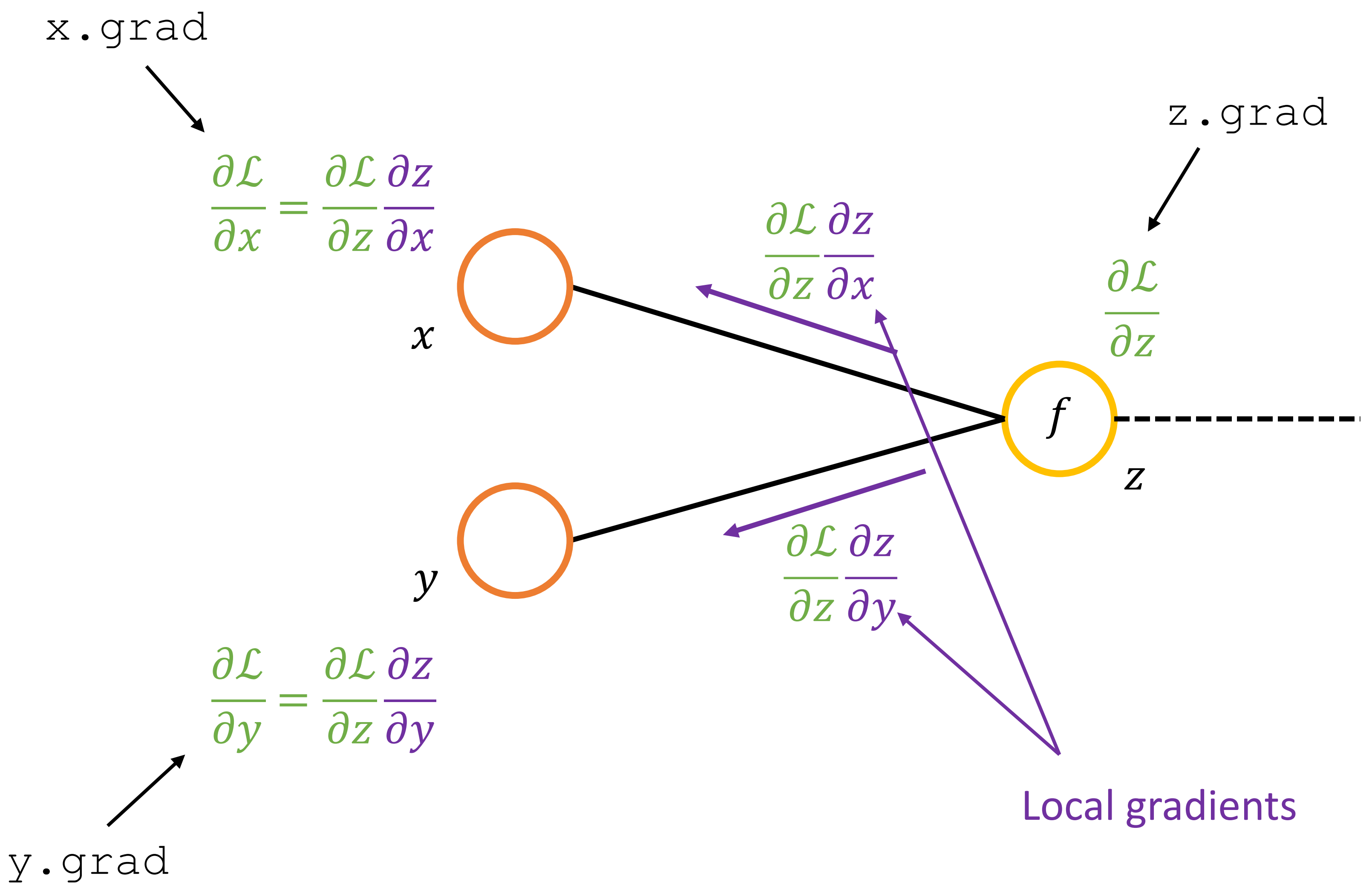
- **Graphs** are generally **dynamic**
  - No “fixed”, pre-defined graph that the forward pass is forced to follow
  - The forward pass is defined implicitly by a sequence of operations
  - A backward graph is created on-the-fly during the forward pass (nodes are added to the graph with each operation) so that the gradients can be backpropagated during the backward pass

Backpropagation



$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial x}$
x
$\frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial x}$
$\frac{\partial \mathcal{L}}{\partial z}$
f
z
y
$\frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial y}$
$\frac{\partial \mathcal{L}}{\partial y} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial y}$

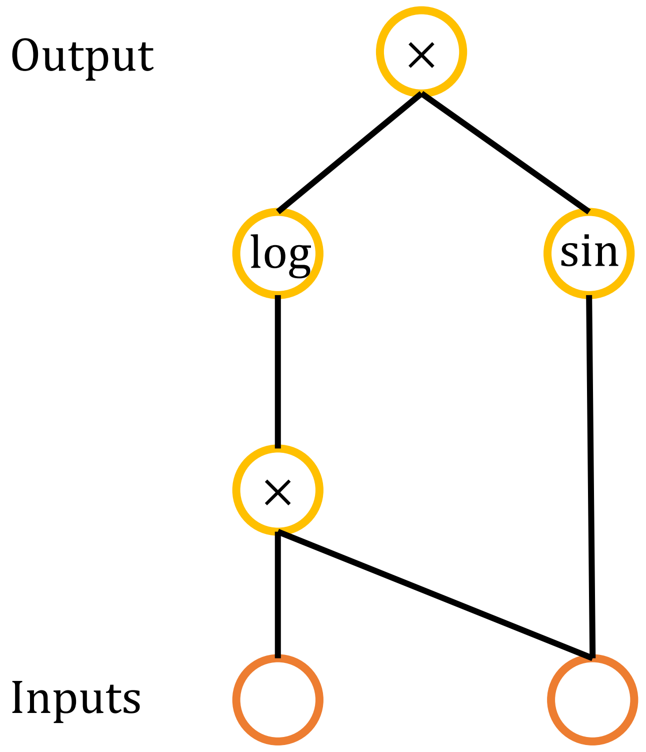
Backpropagation



x.grad
$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial x}$
x
y
$\frac{\partial \mathcal{L}}{\partial y} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial y}$
y.grad
$\frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial x}$
$\frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial y}$
z.grad
$\frac{\partial \mathcal{L}}{\partial z}$
f
z
Local gradients

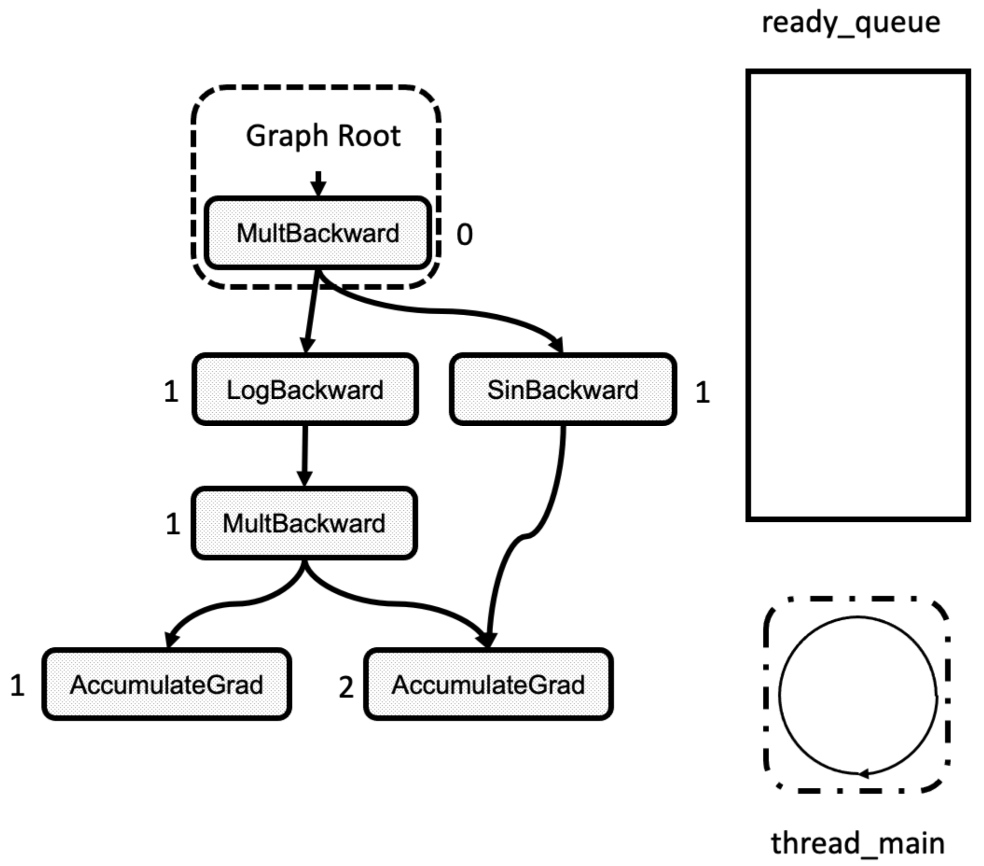
Computational graph execution

Forward pass

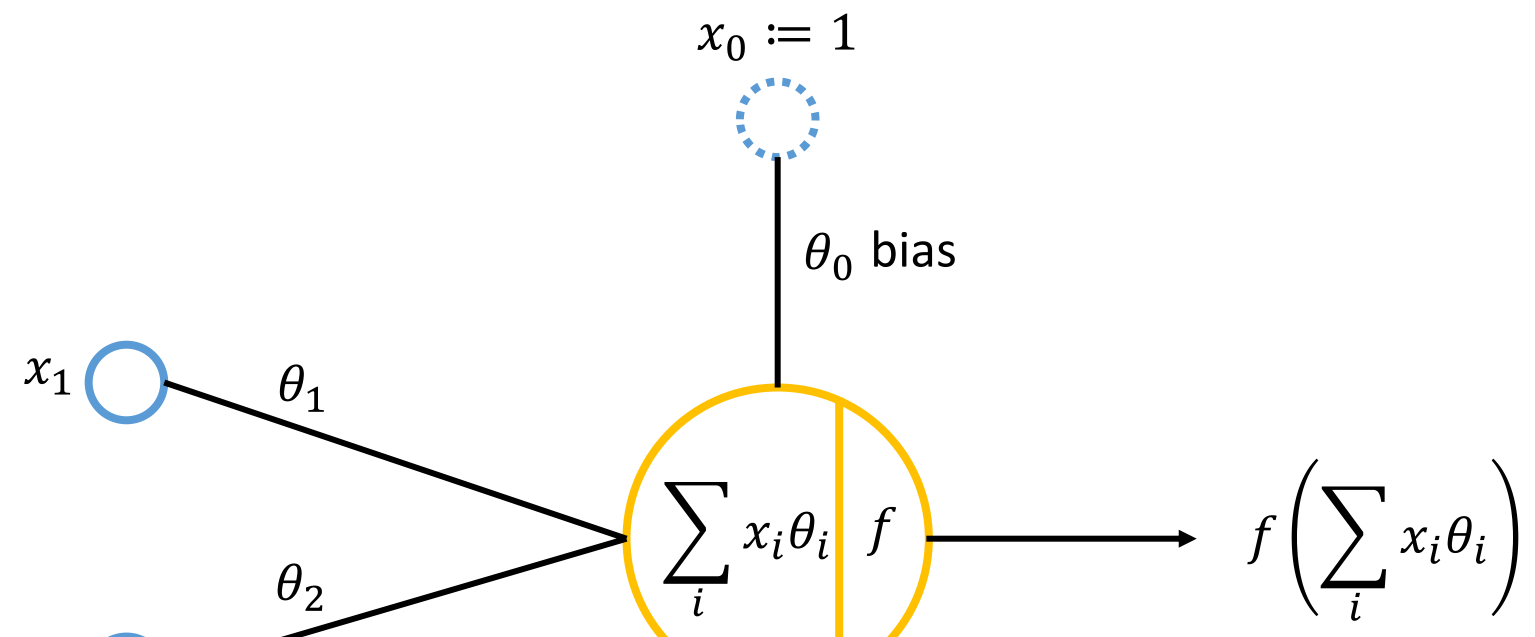


Backward pass

(animated source: pytorch.org)



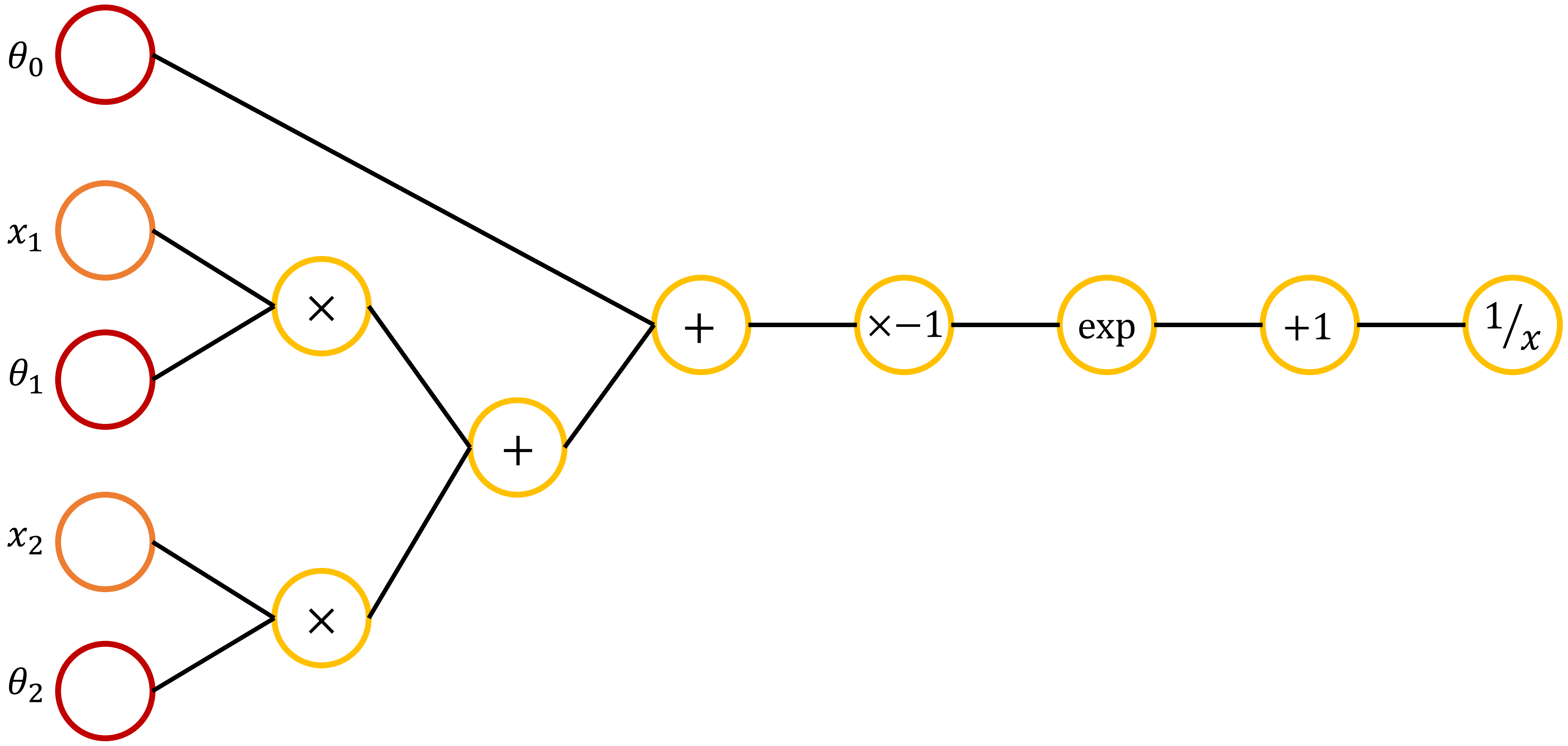
Another example



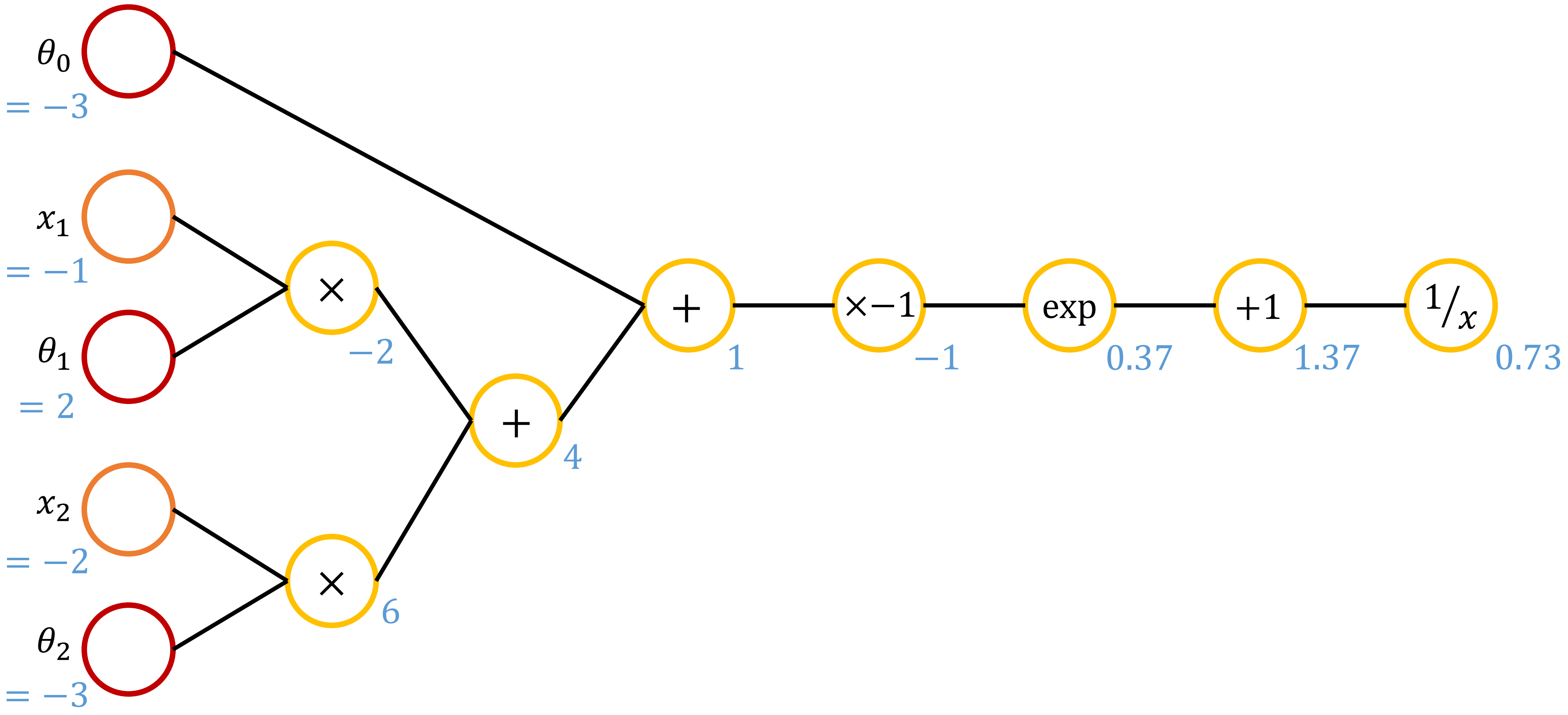
$$f(z) = \frac{1}{1 + e^{-z}}$$

Inputs Weights Output

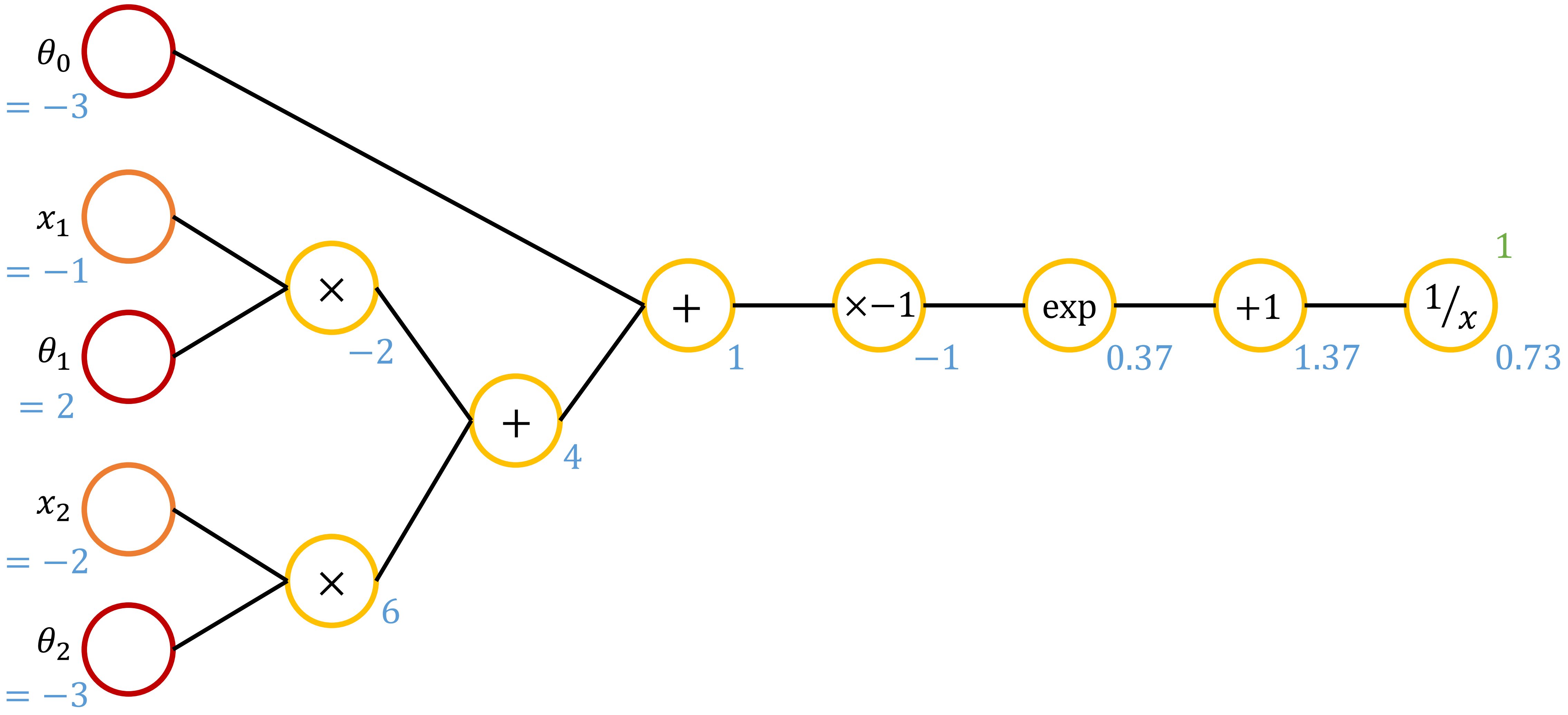
Another example



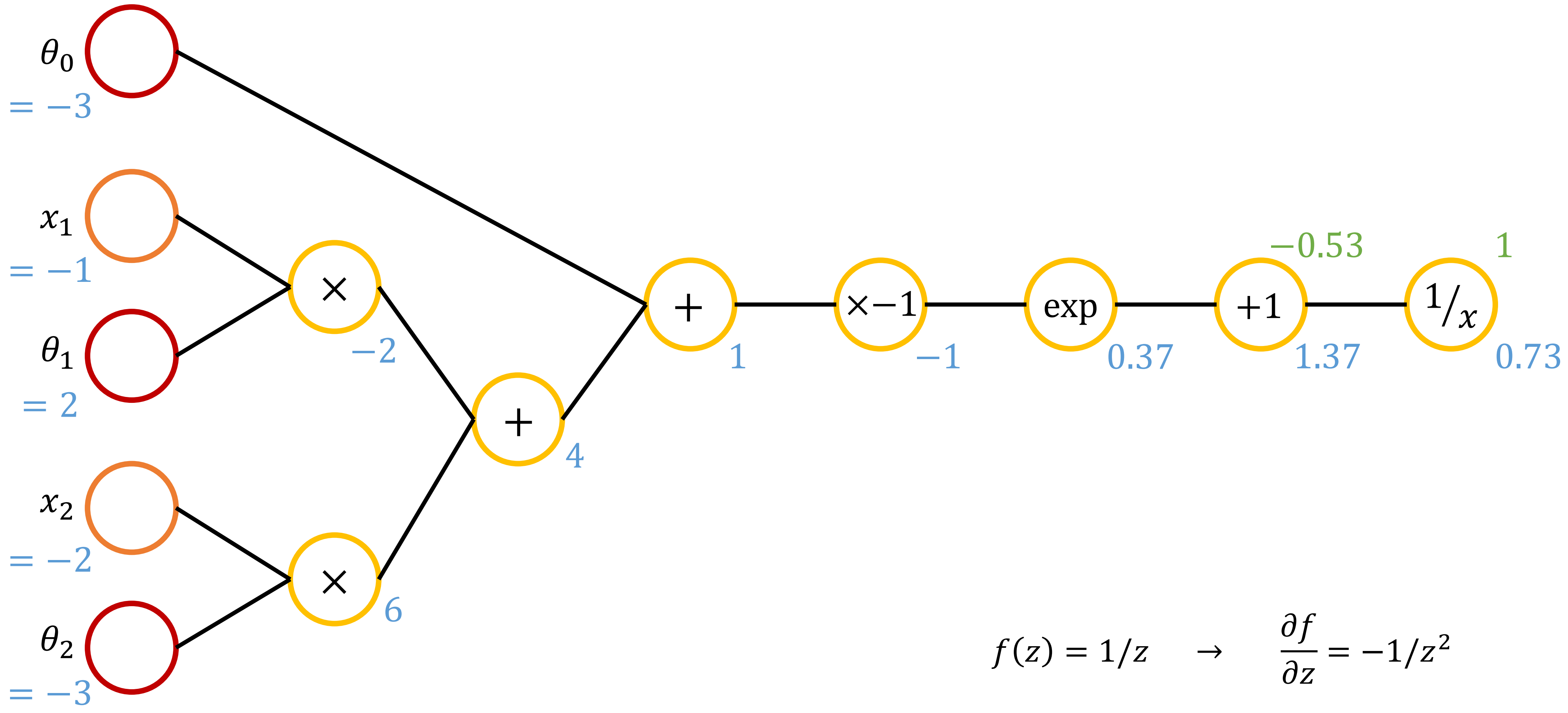
Another example



Another example

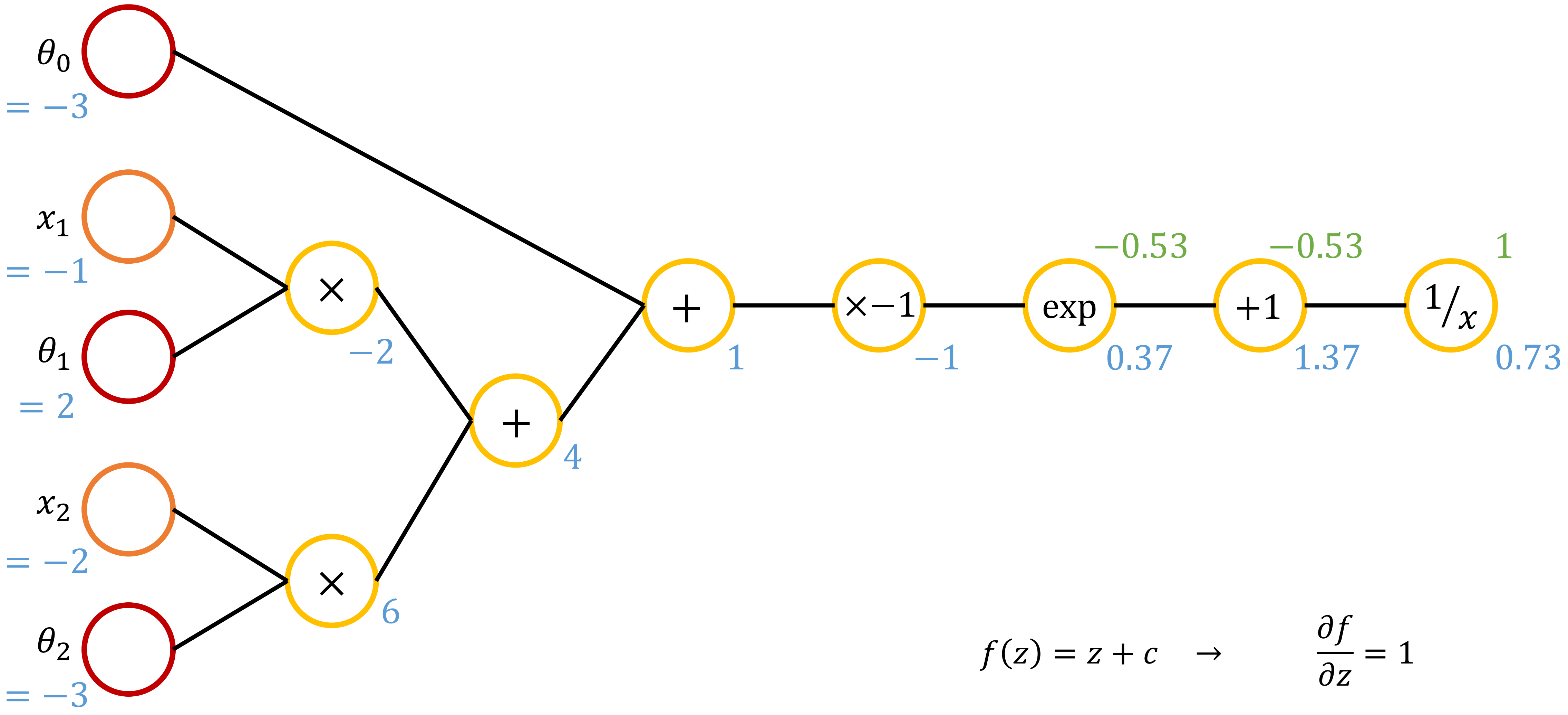


Another example



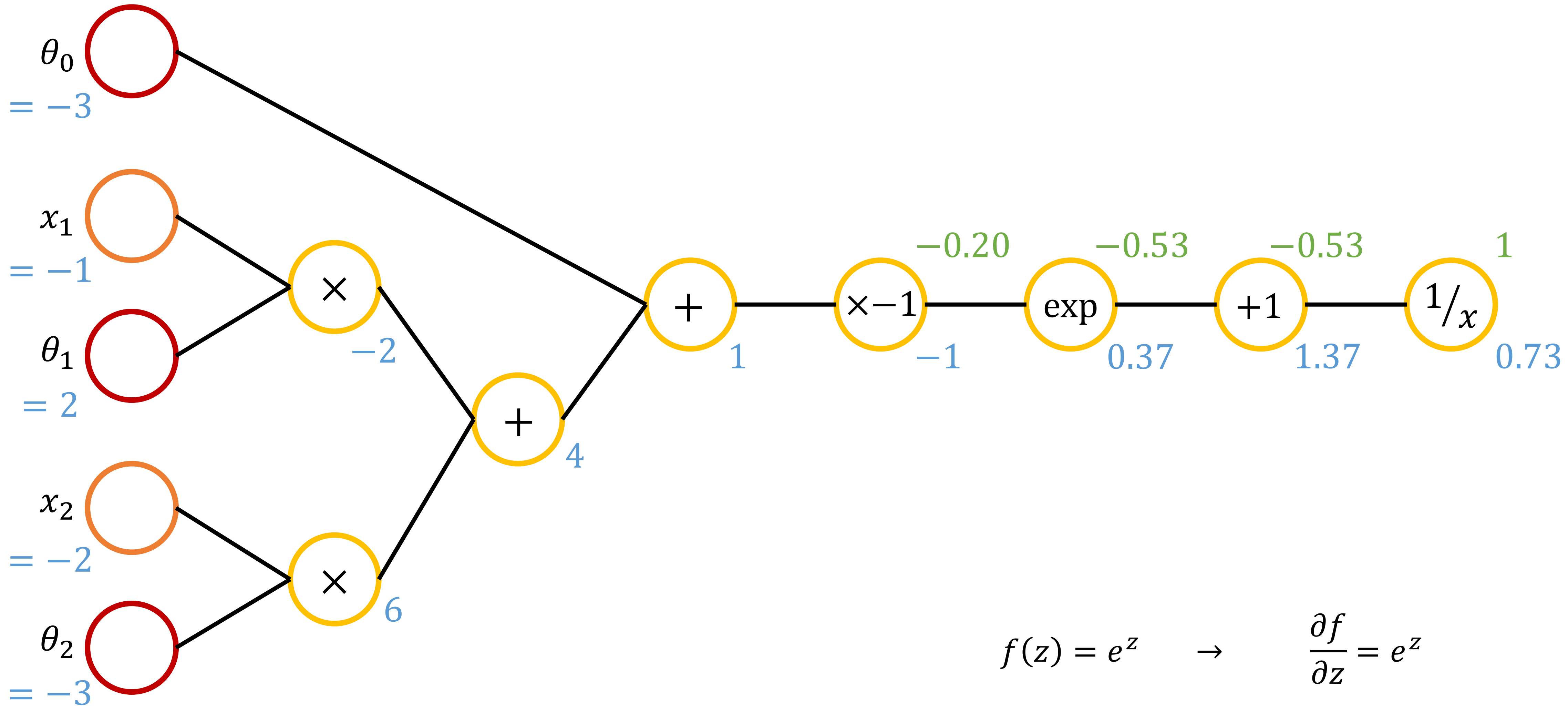
$$f(z) = 1/z \quad \rightarrow \quad \frac{\partial f}{\partial z} = -1/z^2$$

Another example



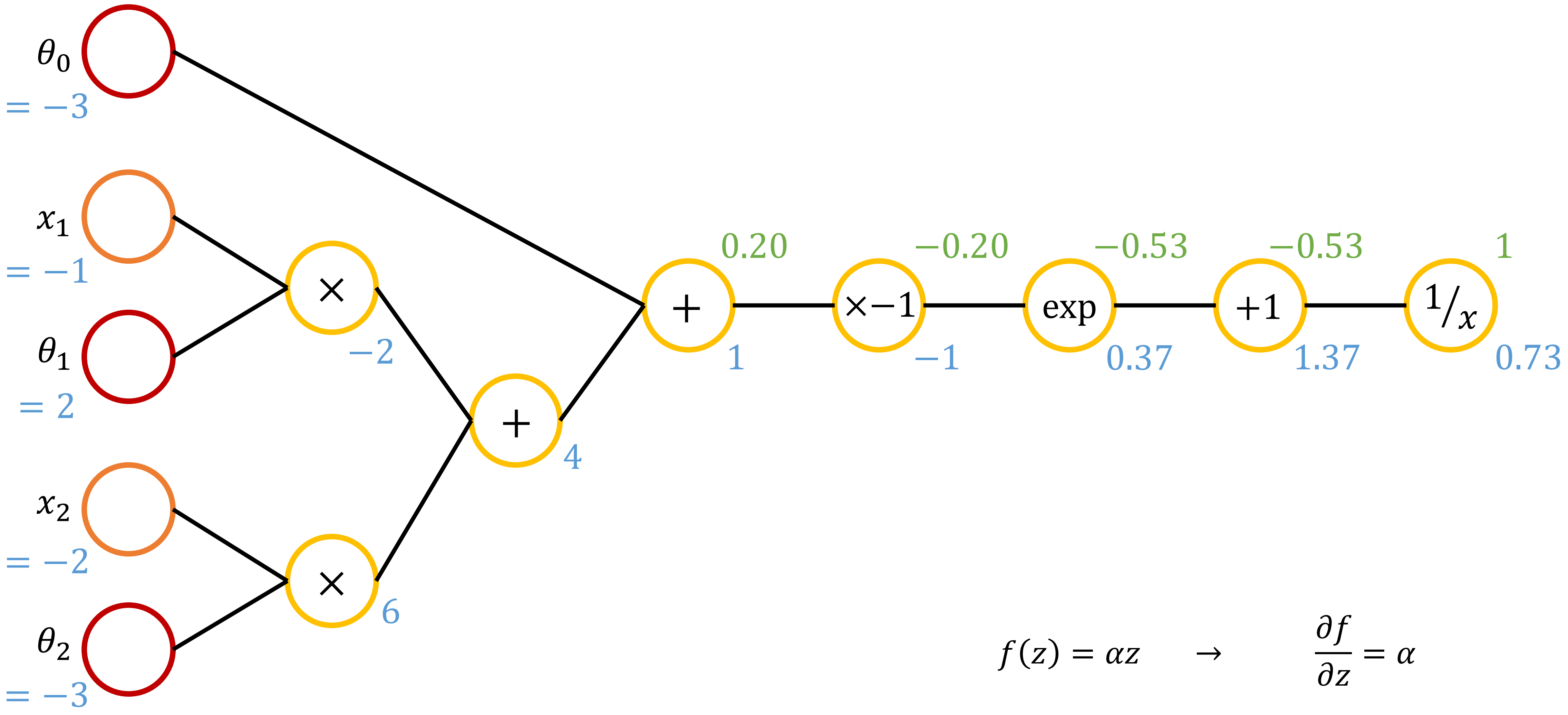
$$f(z) = z + c \quad \rightarrow \quad \frac{\partial f}{\partial z} = 1$$

Another example



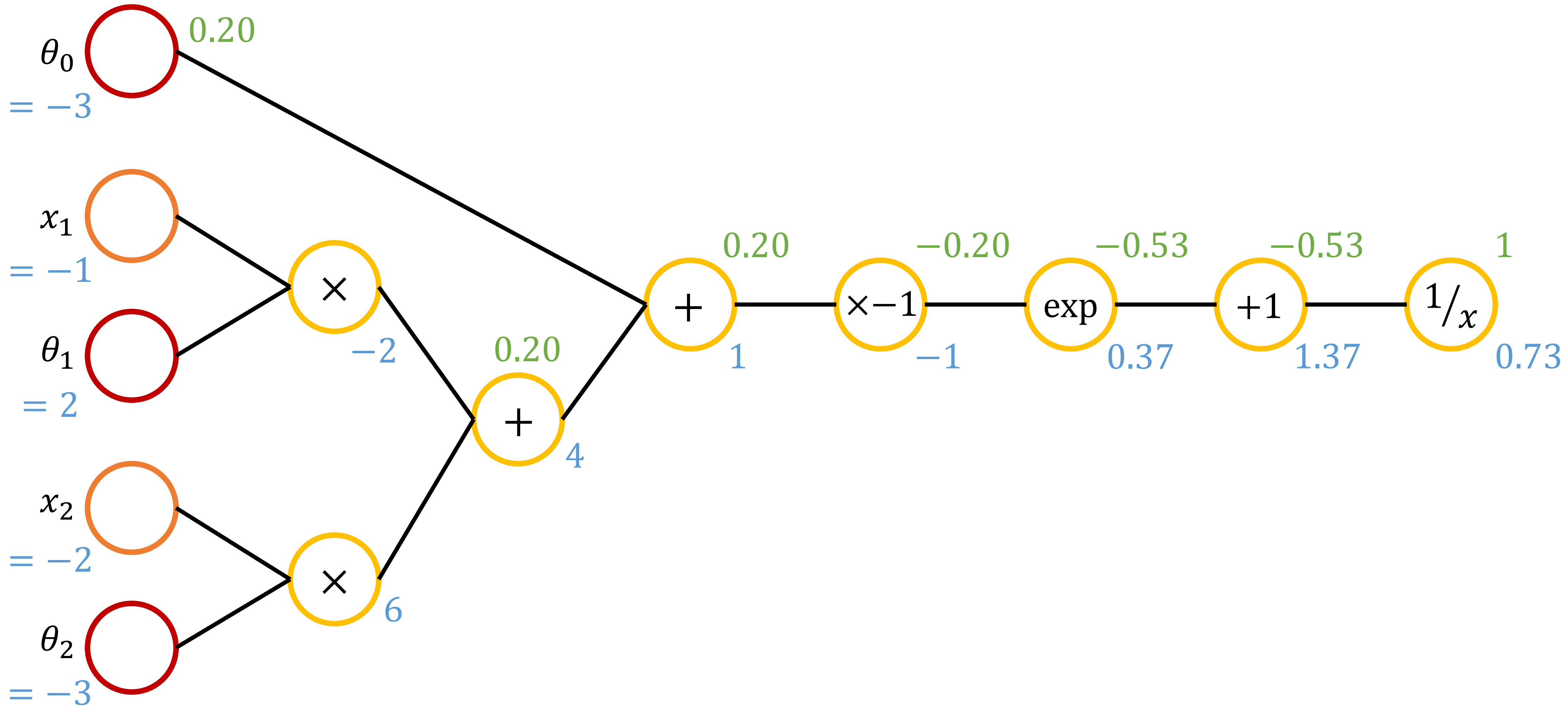
$$f(z) = e^z \quad \rightarrow \quad \frac{\partial f}{\partial z} = e^z$$

Another example

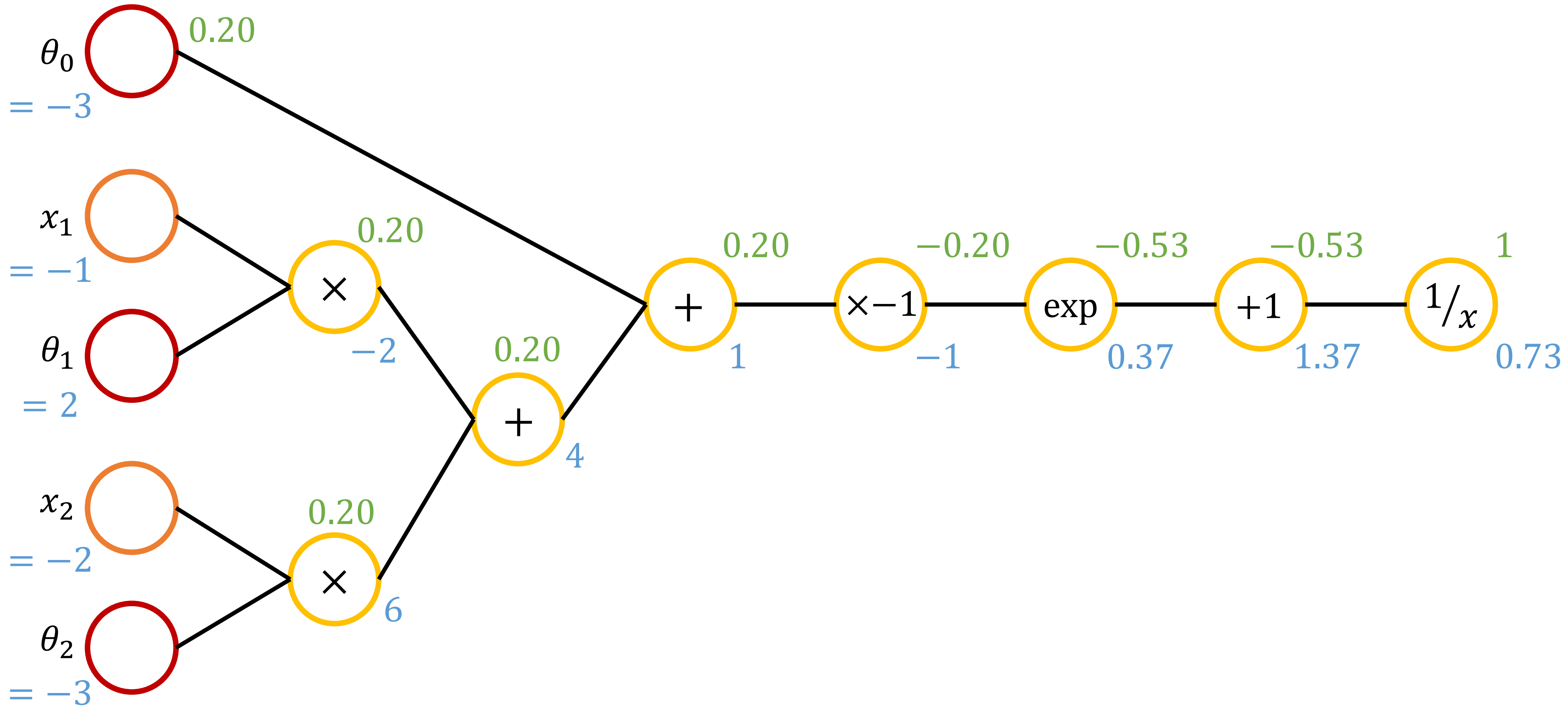


$$f(z) = \alpha z \quad \rightarrow \quad \frac{\partial f}{\partial z} = \alpha$$

Another example

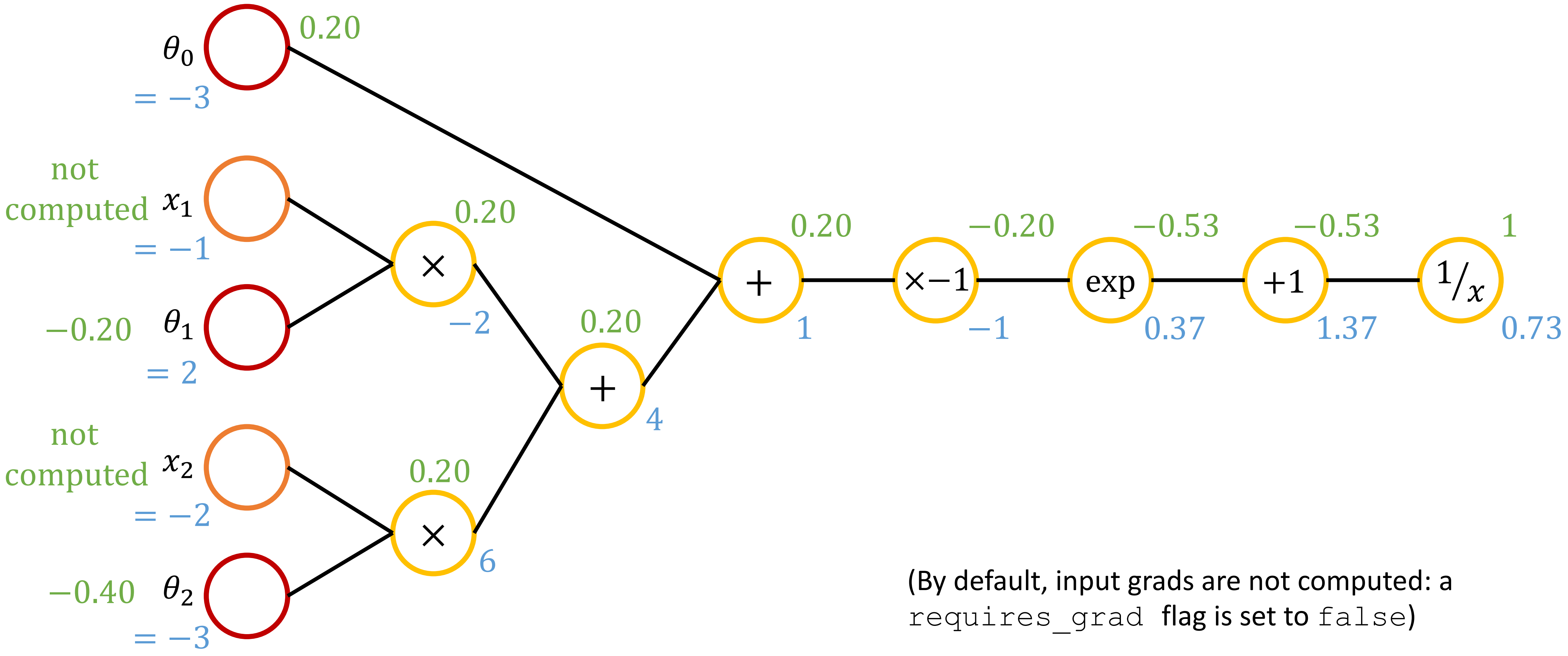


Another example



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

Another example



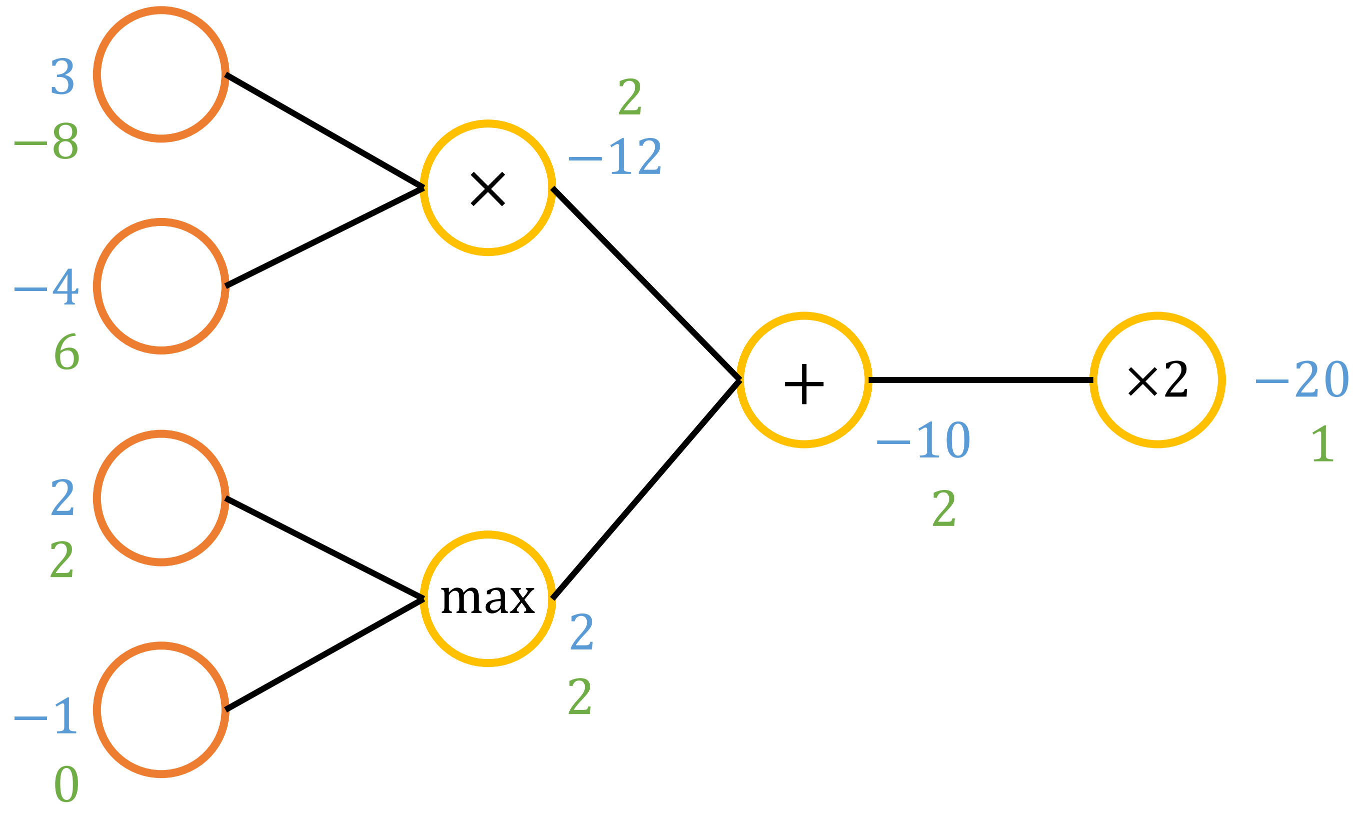
(By default, input grads are not computed: a `requires_grad` flag is set to `false`)

Patterns in the backward flow

Add gate
→ gradient distributor

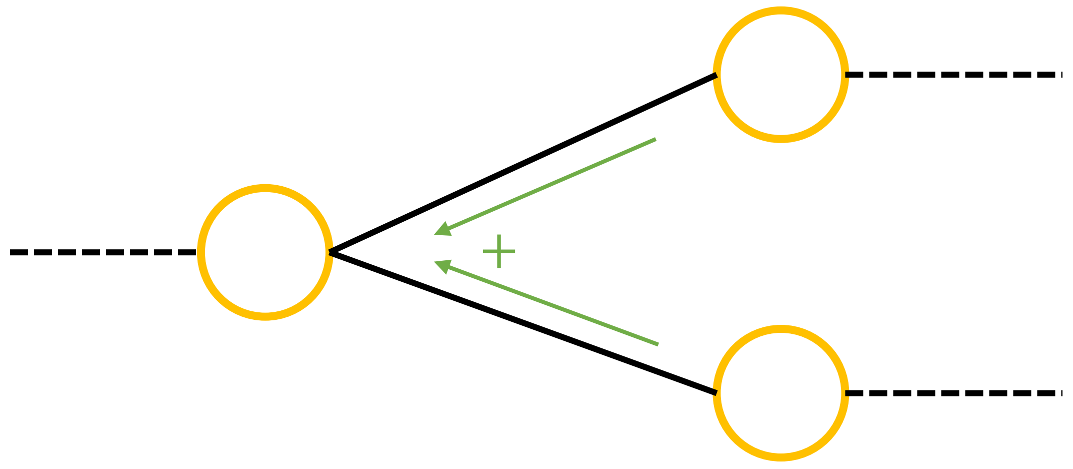
Max gate
→ gradient router

Mul gate
→ gradient switcher



Patterns in the backward flow

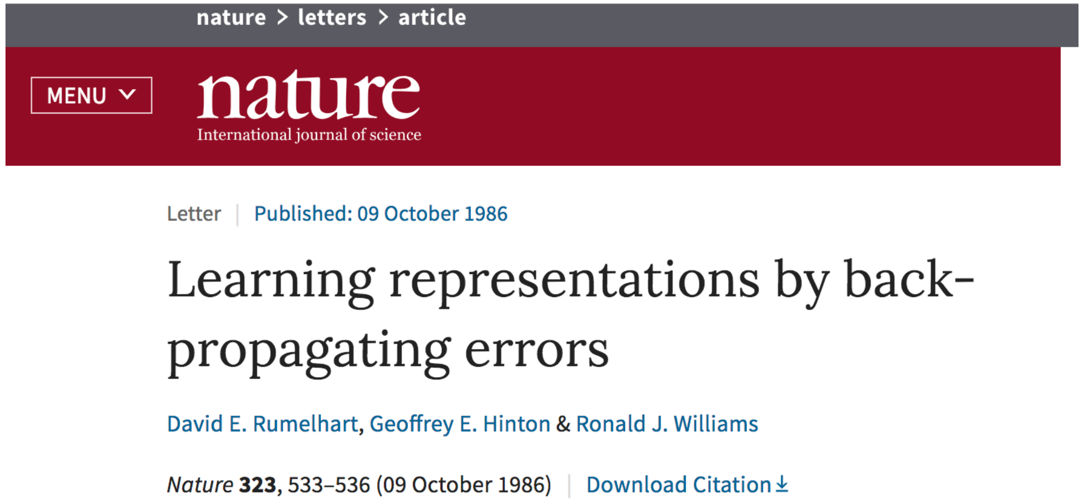
Gradients add at branches



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 compute
   the gradient of the error
   w.r.t. each variable

3. Update model parameters to
   reduce the error (gradient
   descent step)
```



Letter | Published: 09 October 1986

Learning representations by back-propagating errors

David E. Rumelhart, Geoffrey E. Hinton & Ronald J. Williams

Nature **323**, 533–536 (09 October 1986) | Download Citation

Back-propagation

Slide adapted from Daniel Rueckert

Optimizing Neural networks: stochastic gradient descent and variants

(Full-batch) gradient descent is impractical

Remember the loss function:

$$\mathcal{L}(\boldsymbol{\theta}) \triangleq \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(y^{(i)}, h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}))$$

- Computing the gradient of $\mathcal{L}(\boldsymbol{\theta})$ is linear in the number of data points N in the batch
- It is impractical to evaluate it frequently for large datasets (e.g., ImageNet has more than $14M$ annotated examples) or datasets for which evaluation of the loss for a single training sample is costly

Solution: **Stochastic Gradient Descent** (SGD), where the gradient $\nabla_{\boldsymbol{\theta}}\mathcal{L}$ is replaced by a minibatch estimate $\widehat{\nabla}_{\boldsymbol{\theta}}\mathcal{L}$ computed from $n \ll N$ training samples

Stochastic Gradient Descent

```
Initialize parameters $\boldsymbol{\theta}$ and learning rate $\eta$

Repeat until stopping criterion:

    Sample a minibatch of $n$ examples $\{\boldsymbol{x}^{(1)}, \cdots, \boldsymbol{x}^{(n)}\}$ from the training set with
    labels $\{y^{(1)}, \cdots, y^{(n)}\}$

    Compute the minibatch loss function $\tilde{\mathcal{L}}(\boldsymbol{\theta}) \triangleq \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(y^{(i)}, h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}))$ at $\boldsymbol{\theta}^t$

    Compute the gradient of the minibatch loss $\widehat{\nabla}_{\boldsymbol{\theta}} \triangleq \nabla_{\boldsymbol{\theta}}\tilde{\mathcal{L}}$ as an estimate of the
    full-batch gradient

    Apply the parameter update: $\boldsymbol{\theta}^{t+1} \coloneqq \boldsymbol{\theta}^t - \eta\widehat{\nabla}_{\boldsymbol{\theta}}$
```

Epochs

- Each **iteration** ends with an SGD parameter **update step**
- An iteration processes a single minibatch

- The training process often sees 10s of thousands of iterations, so that the full data set is processed several times during optimization

- After a number of iterations, the **full training dataset has been seen** once and exactly once
    - This is called an **epoch**

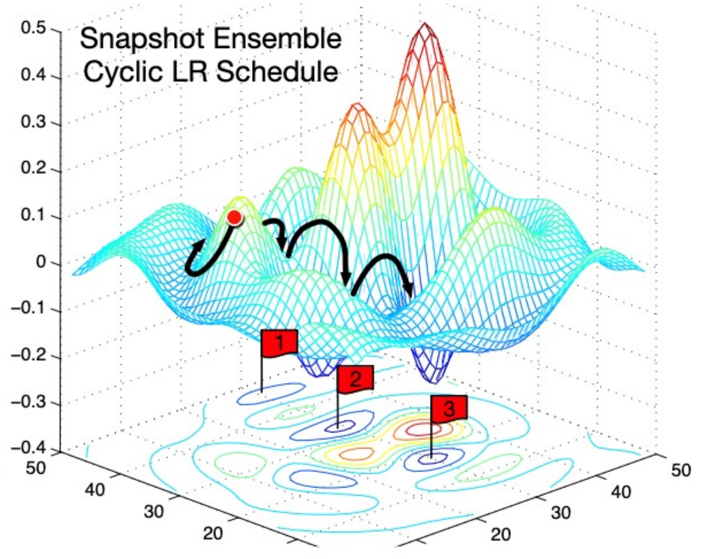
- At the end of an epoch is generally a good time to evaluate a validation loss on a validation dataset, to evaluate the training stage (are we still making progress in the training?)

Learning rate scheduler

- SGD is no longer guaranteed to converge to an exact local/global minimum
- It may keep overshooting and hover in the vicinity of a minimum

- When we slowly decrease the learning rate η, SGD shows the same convergence behaviour as batch gradient descent, almost certainly converging to a local minimum (non-convex optimization) or global minimum (convex optimization)
  - But large learning rates can have a regularization effect

- The learning rate (LR) can be **adjusted over iterations/epochs**
- Many **learning rate schedulers** have been devised:
  - Simple annealing: linear LR, exponential LR – to reduce the LR as we get closer to convergence
  - Adaptive LR: e.g., reduce the LR by an order of magnitude when a metric plateaus
  - Cyclic LR: triangular, cosine schedules – varies between high LR to escape from current local minimum and low LR to facilitate convergence
  - Cosine annealing with warm restarts: warm restarts facilitate moving to a different basin of attraction
  - 1cycle LR



Huang et al. ICLR 2017

Minibatch size

- Large **minibatch sizes** give a more accurate estimate $\hat{\nabla}_{\boldsymbol{\theta}}$ of $\nabla_{\boldsymbol{\theta}}\mathcal{L}$ but the **returns are not linear**
  - Standard error of $\bar{X} \triangleq \frac{1}{n}\sum_i X_i$ from μ is $\sigma/\sqrt{n}$, for n i.i.d. random variables X_i with mean μ and variance σ^2
  - Error decreased by a factor of 10 if number of samples increased by a factor of 100

- Computing architectures (GPUs…) are underutilized by extremely small batch sizes
  → motivates minimum minibatch size
- Memory usage grows linearly with the minibatch size
  → constrains maximum minibatch size

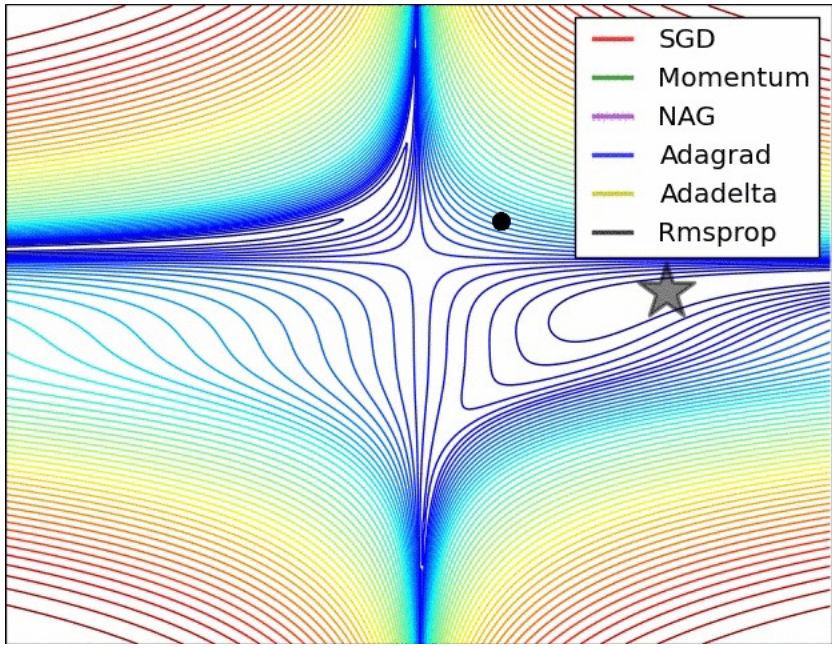
- Small minibatch sizes have a regularization effect

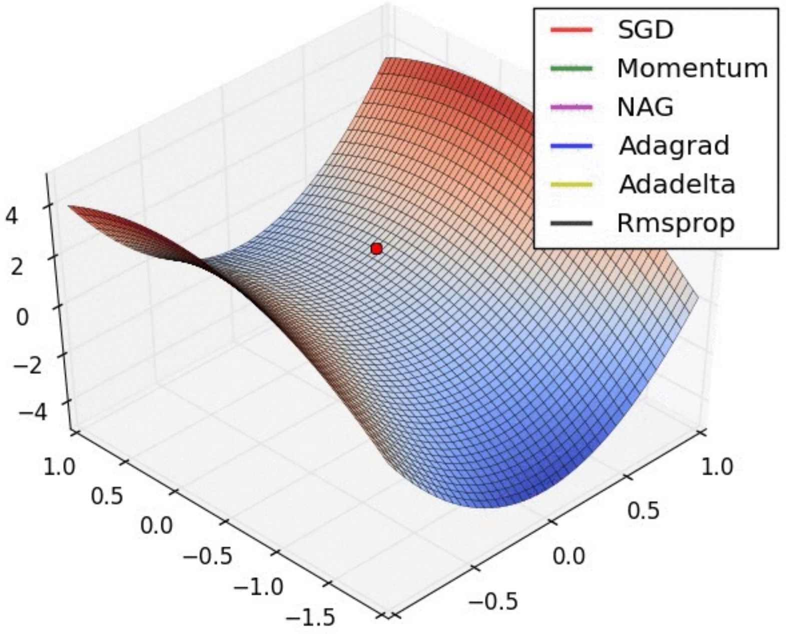
SGD variants

Variants have been designed to address **SGD defects**:

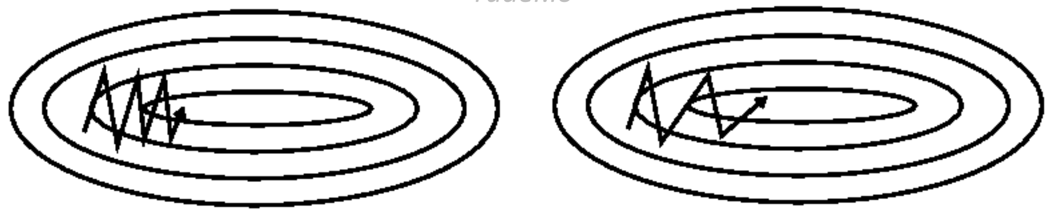
- "noisy" minibatch gradients that point away from the slope of the true loss function
- **Gradient descent direction suboptimal** in case of highly curved/anisotropic loss profiles

Good resource with the rationale for most variants: https://www.ruder.io/optimizing-gradient-descent/



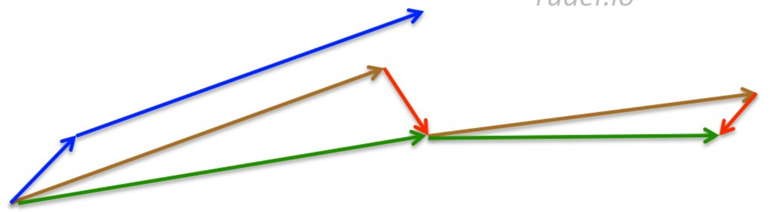


SGD with momentum



```
Initialize parameters $\boldsymbol{\theta}$, learning rate $\eta$,
momentum $\boldsymbol{v}$ and momentum parameter $\gamma$

Repeat until stopping criterion:

    Sample a minibatch of $n$ examples $\{\boldsymbol{x}^{(1)}, \cdots, \boldsymbol{x}^{(n)}\}$ from the training set with
    labels $\{y^{(1)}, \cdots, y^{(n)}\}$

    Compute the minibatch loss function $\tilde{\mathcal{L}}(\boldsymbol{\theta}) \triangleq \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(y^{(i)}, h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}))$ at $\boldsymbol{\theta}^t$

    Compute the gradient of the minibatch loss $\widehat{\nabla}_{\boldsymbol{\theta}} \triangleq \nabla_{\boldsymbol{\theta}} \tilde{\mathcal{L}}$

    Compute momentum update $\boldsymbol{v}^{t+1} \coloneqq \gamma \boldsymbol{v}^t + \eta \widehat{\nabla}_{\boldsymbol{\theta}}$

    Apply the parameter update: $\boldsymbol{\theta}^{t+1} \coloneqq \boldsymbol{\theta}^t - \boldsymbol{v}^{t+1}$
```

SGD with Nesterov momentum: reduce overshooting



Following the gradient then adding the momentum term could bring us too far (in blue)

```
Initialize parameters $\boldsymbol{\theta}$, learning rate $\eta$,
momentum $\boldsymbol{v}$ and momentum parameter $\gamma$

Repeat until stopping criterion:

        Sample a minibatch of $n$ examples $\{\boldsymbol{x}^{(1)}, \cdots, \boldsymbol{x}^{(n)}\}$ from the training set with
        labels $\{y^{(1)}, \cdots, y^{(n)}\}$

        Apply interim update $\tilde{\boldsymbol{\theta}} \coloneqq \boldsymbol{\theta}^t - \gamma \boldsymbol{v}^t$

        Compute the minibatch loss function $\tilde{\mathcal{L}}(\boldsymbol{\theta}) \triangleq \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(y^{(i)}, h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}))$ at $\tilde{\boldsymbol{\theta}}$

        Compute the gradient of the minibatch loss $\widehat{\nabla}_{\boldsymbol{\theta}} \triangleq \nabla_{\boldsymbol{\theta}} \tilde{\mathcal{L}}$ at $\tilde{\boldsymbol{\theta}}$

        Compute momentum update $\boldsymbol{v}^{t+1} \coloneqq \gamma \boldsymbol{v}^t + \eta \widehat{\nabla}_{\boldsymbol{\theta}}$

        Apply the parameter update: $\boldsymbol{\theta}^{t+1} \coloneqq \boldsymbol{\theta}^t - \boldsymbol{v}^{t+1}$
```

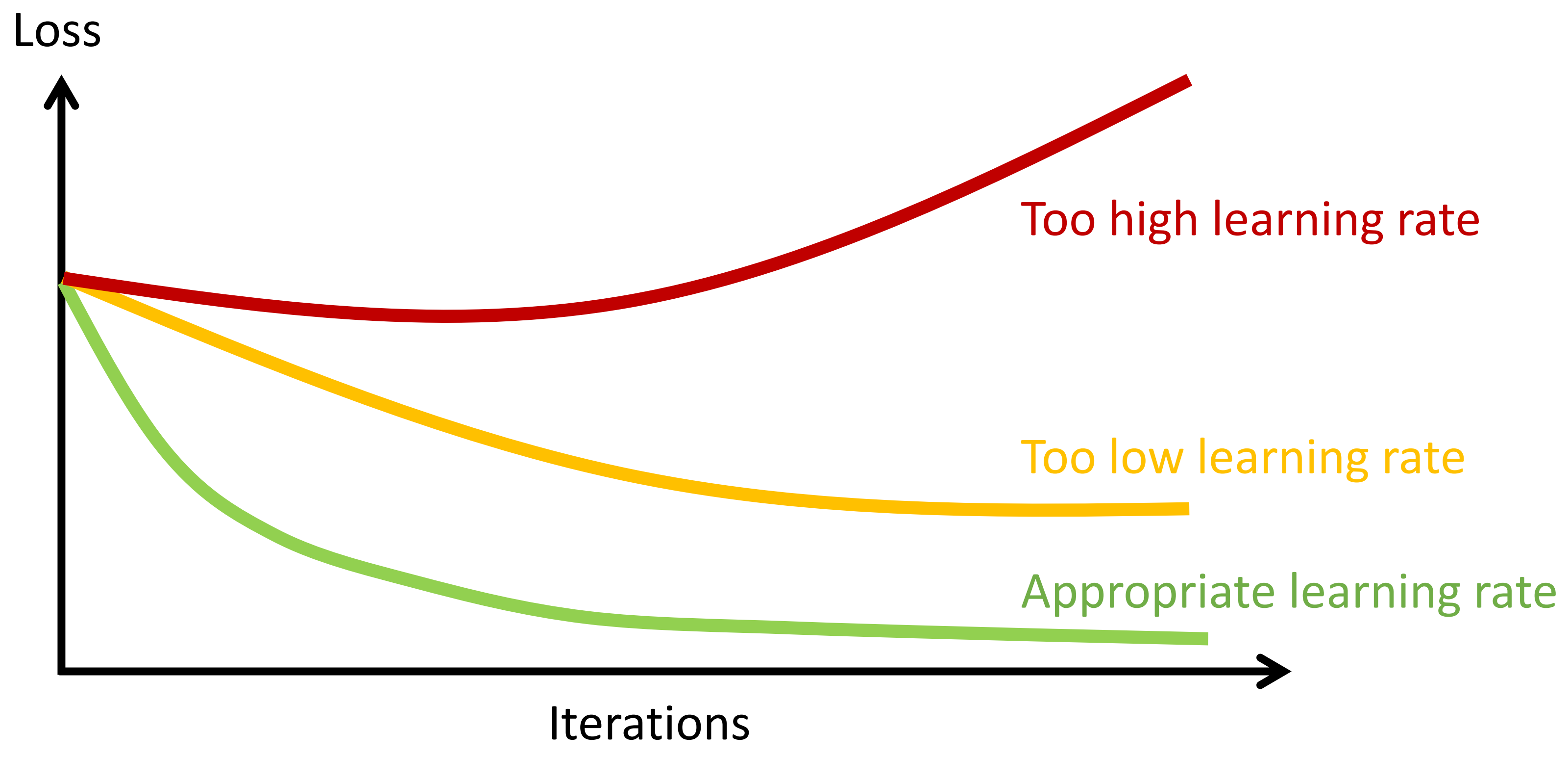
RMSprop: normalize gradients by a moving average

```
Initialize parameters $\boldsymbol{\theta}$, learning rate $\eta$, squared gradient moving average $E[\boldsymbol{g}^2]$,
parameter $0 < \gamma < 1$ and small constant $\epsilon$

Repeat until stopping criterion:

    Sample a minibatch of $n$ examples $\{\boldsymbol{x}^{(1)}, \cdots, \boldsymbol{x}^{(n)}\}$ from the training set with
    labels $\{y^{(1)}, \cdots, y^{(n)}\}$

    Compute the minibatch loss function $\tilde{\mathcal{L}}(\boldsymbol{\theta}) \triangleq \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(y^{(i)}, h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}))$ at $\boldsymbol{\theta}^t$
    Compute the gradient of the minibatch loss $\widehat{\nabla}_{\boldsymbol{\theta}} \triangleq \nabla_{\boldsymbol{\theta}}\tilde{\mathcal{L}}$ at $\boldsymbol{\theta}^t$

    Update running average $E[\boldsymbol{g}^2]_t \coloneqq \gamma E[\boldsymbol{g}^2]_{t-1} + (1-\gamma)\widehat{\nabla}_{\boldsymbol{\theta}} \odot \widehat{\nabla}_{\boldsymbol{\theta}}$

    Compute the parameter update $\Delta\boldsymbol{\theta}^t \coloneqq -\frac{\eta}{\sqrt{E[\boldsymbol{g}^2]_t + \epsilon}} \odot \widehat{\nabla}_{\boldsymbol{\theta}}$
    Apply the parameter update: $\boldsymbol{\theta}^{t+1} \coloneqq \boldsymbol{\theta}^t + \Delta\boldsymbol{\theta}^t$
```

Adam: use moving average of gradient

```
Initialize parameters $\boldsymbol{\theta}$, learning rate $\eta$, gradient moving average $\boldsymbol{m} \coloneqq \mathbf{0}$, squared gradient moving average $\boldsymbol{v} \coloneqq \mathbf{0}$, parameters $0 < \beta_1, \beta_2 < 1$ and small constant $\epsilon$

Repeat until stopping criterion:

    Sample a minibatch of $n$ examples $\{\boldsymbol{x}^{(1)}, \cdots, \boldsymbol{x}^{(n)}\}$ from the training set with labels $\{y^{(1)}, \cdots, y^{(n)}\}$

    Compute the minibatch loss function $\tilde{\mathcal{L}}(\boldsymbol{\theta}) \triangleq \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(y^{(i)}, h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}))$ at $\boldsymbol{\theta}^t$
    Compute the gradient of the minibatch loss $\widehat{\nabla}_{\boldsymbol{\theta}} \triangleq \nabla_{\boldsymbol{\theta}}\tilde{\mathcal{L}}$ at $\boldsymbol{\theta}^t$

    Update running averages: $\boldsymbol{m}_t \coloneqq \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\widehat{\nabla}_{\boldsymbol{\theta}}$ and $\boldsymbol{v}_t \coloneqq \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)\widehat{\nabla}_{\boldsymbol{\theta}} \odot \widehat{\nabla}_{\boldsymbol{\theta}}$
    De-bias the moment estimates: $\widehat{\boldsymbol{m}}_t \coloneqq {}^{\boldsymbol{m}_t}/_{1-\beta_1^t}$ and $\widehat{\boldsymbol{v}}_t \coloneqq {}^{\boldsymbol{v}_t}/_{1-\beta_2^t}$

    Compute the parameter update $\Delta\boldsymbol{\theta}^t \coloneqq -\frac{\eta}{\sqrt{\widehat{\boldsymbol{v}}_t}+\epsilon} \odot \widehat{\boldsymbol{m}}_t$
    Apply the parameter update: $\boldsymbol{\theta}^{t+1} \coloneqq \boldsymbol{\theta}^t + \Delta\boldsymbol{\theta}^t$
```

Choice of learning rate



Adapted from D. Rueckert's slides

Regularization

Regularization

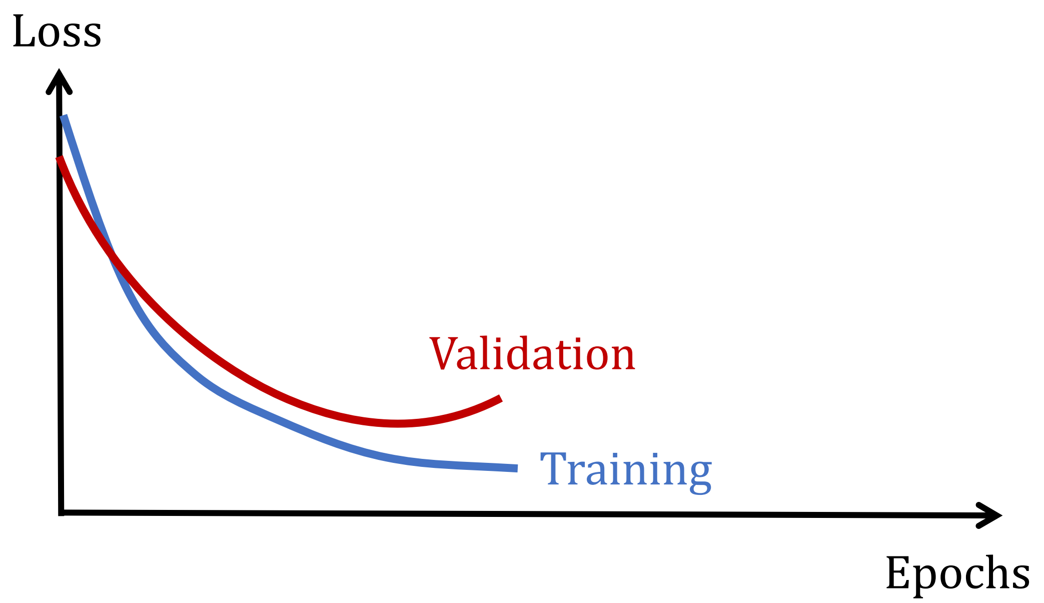
Regularization is useful when the optimization **problem is underdetermined**
- This is common in ML and leads to overfitting

Techniques **to avoid overfitting** applicable with NNs:
- Early stopping
- L^p regularization
- Dropout
- Data augmentation

Early stopping

- **Interrupt the training** when the **validation loss** stops decreasing for a few epochs
- Keep the best result (lowest validation loss)

- Necessitates to monitor the loss on a validation dataset at the end of each epoch (good practice anyway)



L^p regularization

Adds another term to the loss function in addition to the data term $\mathcal{L}_{\text{data}}(\boldsymbol{\theta}) \triangleq \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}\left(y^{(i)}, h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right)\right)$ seen in the previous slides

L^2 regularization
a.k.a. Tikhonov regularization, quadratic regularization

$$\mathcal{L} \triangleq \mathcal{L}_{\text{data}} + \lambda \|\boldsymbol{\theta}\|_2{}^2$$

Can be implemented directly via the optimizer under the name “weight decay”

L^1 regularization

$$\mathcal{L} \triangleq \mathcal{L}_{\text{data}} + \lambda \|\boldsymbol{\theta}\|_1$$

Weight decay

- Weight decay is often taken as synonymous to L2 regularization
- In the original work of Hanson et Pratt (1988), weight decay corresponds to the red term in the following parameter update:

$$\Delta\boldsymbol{\theta}_t \coloneqq -\lambda\boldsymbol{\theta}_t - \alpha\nabla\mathcal{L}_{\text{data}}(\boldsymbol{\theta}_t)$$

- When using SGD, this is indeed identical to L2 regularization with a weight λ/α

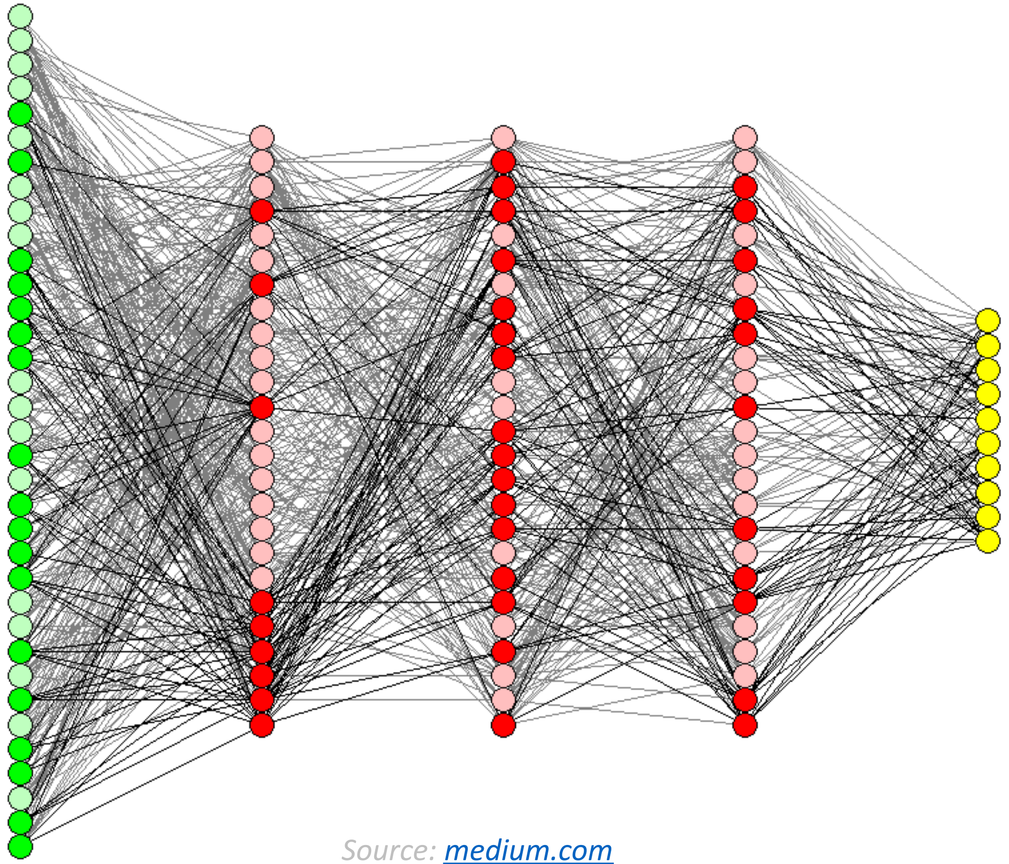
- But popular modern optimizers (RMSprop, Adam) use adaptive updates with modified gradients
- In this case, it is different to:
  - Use L2 regularization, or equivalently add $+\lambda\boldsymbol{\theta}_t$ to $\nabla\mathcal{L}_{\text{data}}(\boldsymbol{\theta}_t)$ before computing the normalized updates of RMSprop / Adam
  - Use weight decay i.e., add the term $-\lambda\boldsymbol{\theta}_t$ in the final computation of $\Delta\boldsymbol{\theta}_t$

- The latter strategy is shown in Loshchilov et Hutter, ICLR 2019, to be superior to the former strategy:
  a. better generalization error
  b. broader optimal parameter sets that better decouple optimal learning rate and weight decay values

- It is implemented in the **AdamW optimizer** ("Adam with decoupled weight decay")
  - Careful: in RMSprop and Adam, "weight decay" actually refers to the former strategy (L2 regularization)

Dropout

- At each iteration, **every neuron** (input or hidden), for each data point in the minibatch, **has a probability p of being "dropped out"**: its value is replaced by 0
- Prevents co-adaptation of neurons



Source: [medium.com]()

- p is called dropout rate
- can be set to 0.5 or smaller values

- At test time, neurons are not dropped out
  - Apply a small correction to correct for the fact that each neuron on average sees input from more neurons

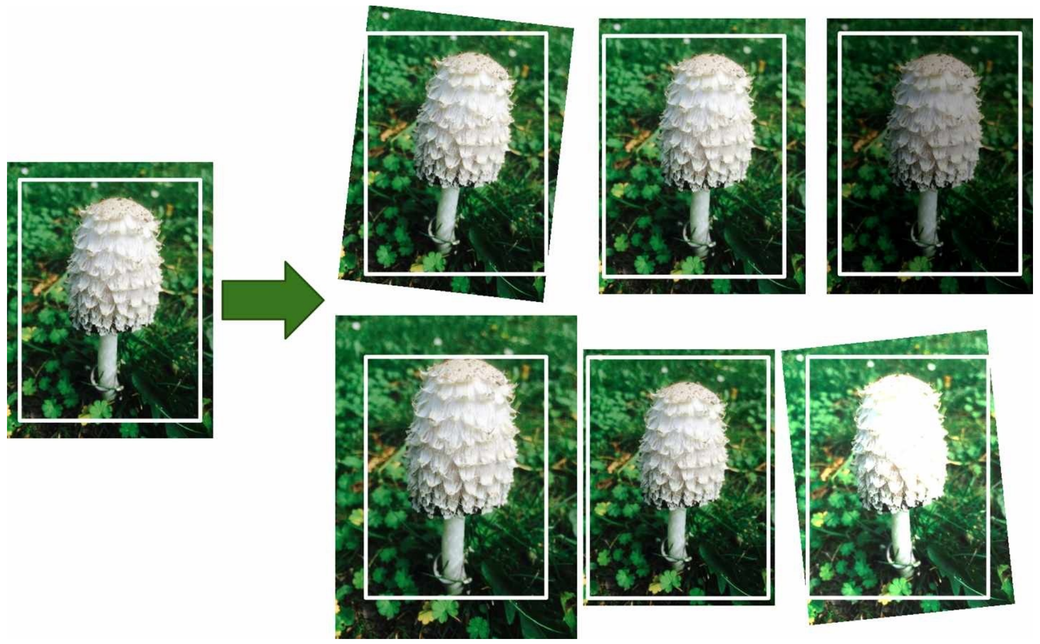
- Dropout **implemented as a layer** in the network, to be placed right after the layer to which it applies
- Very effective for fully connected layers

Data augmentation

Generate new training samples with artificial variability **from existing training samples**, artificially boosting the size of the dataset

Applied random transformations need to be realistic

- Intensity transformations (contrast, color, noise, etc.)
- Spatial transformations: crop, scale, rotate, shift (translate), flip



Implementation, etc.

Weight initialization

How to initialize weights $\theta^l_{i,j}$?

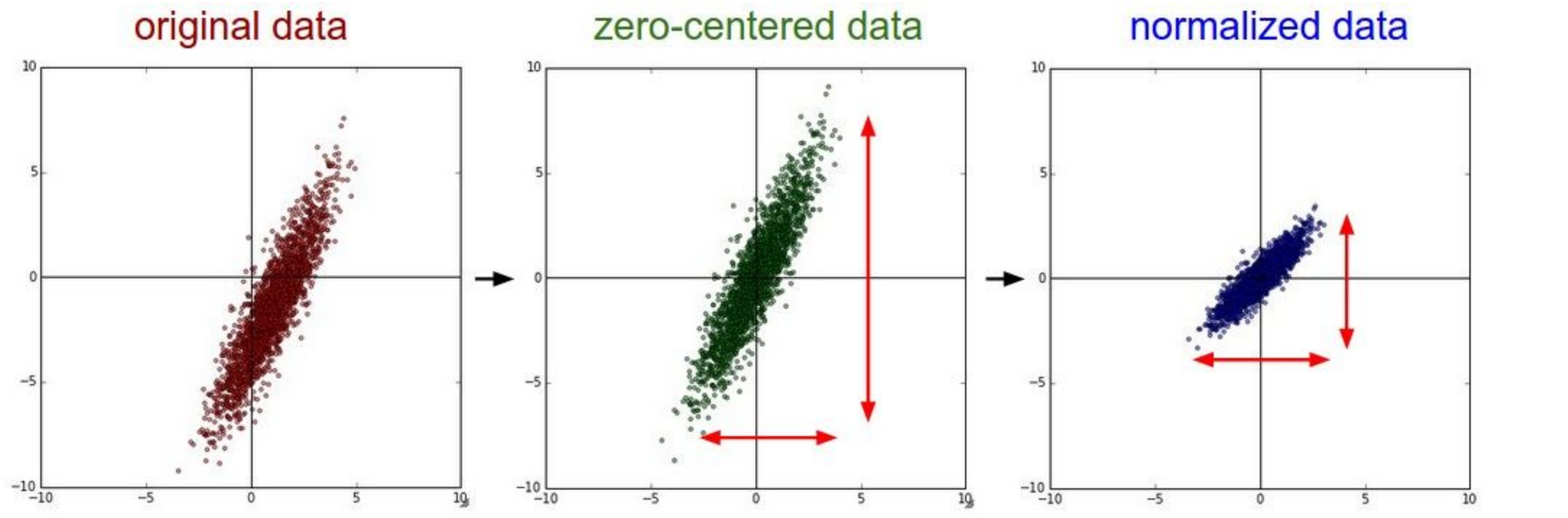
- ~~Constant value $\theta^l_{i,j} := 0$~~ does not break symmetry → bad idea

- Small random value: $\theta^l_{i,j} := 0.01 \cdot \mathcal{N}(0,1)$ or $\theta^l_{i,j} := \mathcal{N}(0,1)/\sqrt{M_l}$ or $\theta^l_{i,j} := \sqrt{2/M_l} \cdot \mathcal{N}(0,1)$ (He et al., 2015)
  - Scaling by number of neurons M_l input to the layer allows to keep variance of output constant

- "Xavier" initialization (Gloriot and Bengio, 2010): $\theta^k_{i,j} = \mathcal{U}(-1/\sqrt{n}, +1/\sqrt{n})$ where $\mathcal{U}(-a,+a)$ is the uniform distribution in the interval delimited by $-a$ and $+a$

- …

Normalizing input data



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

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

From D. Rueckert's slides

Normalizing intermediate activations

- Normalizing the input layer is great, but
- What about the intermediate activations?

- The inputs to intermediate layers are not normalized since different neurons have different weights and different activation statistics

- All previous layers $1 \cdots (l-1)$ have an effect on the lth layer's activations

- When backpropagating, these interactions are not accounted for:
  the weights of the layer l will be updated without accounting for the fact that the previous layers are updated too and their activation statistics change

- So-called **"internal covariate shift"** = change of distributions as network updates

Idea: Normalize the mean and variance in intermediate layers as well

Batch normalization

Let $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_M$ be intermediate inputs

- C-dimensional quantities if C is the number of neurons in the previous layer
- For us so far, m will index the sample index in the minibatch and $M = n$ is the minibatch size (because we deal with tabular data; for image data, this will be slightly different as per next slides)
- Intermediate inputs can be before (more rarely) or after (more commonly) the nonlinearity

Compute the **normalized intermediate input**:

$$\widehat{\boldsymbol{x}}_m \coloneqq \frac{\boldsymbol{x}_m - \boldsymbol{\mu}}{\sqrt{\boldsymbol{\sigma}^2 + \epsilon}}$$

Where:

- $\boldsymbol{\mu} \coloneqq \frac{\mathbf{1}}{M} \sum_m \boldsymbol{x}_m$
- $\boldsymbol{\sigma}^2 \coloneqq \frac{\mathbf{1}}{M} \sum_m (\boldsymbol{x}_m - \boldsymbol{\mu})^2$

Output $\boldsymbol{h}_m \coloneqq \boldsymbol{\gamma} \odot \widehat{\boldsymbol{x}}_m + \boldsymbol{\beta} \triangleq BN_{\boldsymbol{\gamma},\boldsymbol{\beta}}(\boldsymbol{x}_{\boldsymbol{m}})$
where $\boldsymbol{\gamma}, \boldsymbol{\beta}$ are learnable parameters

Batch normalization

Let $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_M$ be intermediate inputs

- C-dimensional quantities if C is the number of neurons in the previous layer
- For us so far, m will index the sample index in the minibatch and $M = n$ is the minibatch size (because we deal with tabular data; for image data, this will be slightly different as per next slides)
- Intermediate inputs can be before (more rarely) or after (more commonly) the nonlinearity

Compute the **normalized intermediate input**:

$$\widehat{\boldsymbol{x}}_m \coloneqq \frac{\boldsymbol{x}_m - \boldsymbol{\mu}}{\sqrt{\boldsymbol{\sigma}^2 + \epsilon}}$$

Where:

- $\boldsymbol{\mu} \coloneqq \frac{\mathbf{1}}{M} \sum_m \boldsymbol{x}_m$
- $\boldsymbol{\sigma}^2 \coloneqq \frac{\mathbf{1}}{M} \sum_m (\boldsymbol{x}_m - \boldsymbol{\mu})^2$

At test time, minibatch averages are replaced by a fixed exponential moving average (EMA, computed during training)

Output $\boldsymbol{h}_m \coloneqq \boldsymbol{\gamma} \odot \widehat{\boldsymbol{x}}_m + \boldsymbol{\beta} \triangleq BN_{\boldsymbol{\gamma},\boldsymbol{\beta}}(\boldsymbol{x}_{\boldsymbol{m}})$
where $\boldsymbol{\gamma}, \boldsymbol{\beta}$ are learnable parameters

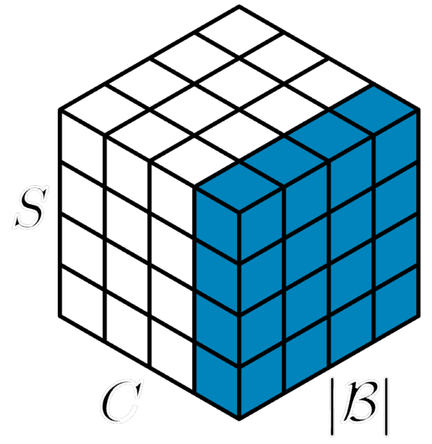
Batch normalization

Downsides of BN:

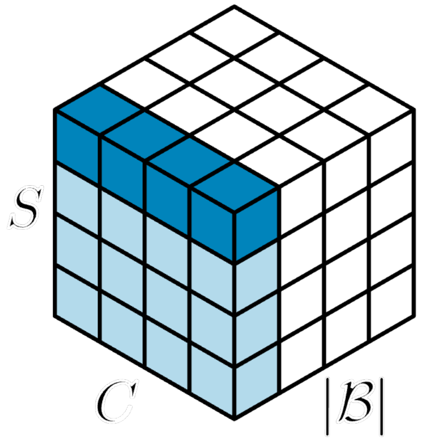
- Does not work well with small minibatches (estimated statistics are unreliable)
- Uses different statistics for inference / test-time than for training
- Does not always work well with L2 regularization (Hoffer et al. NeuRIPS 2018) or dropout (Li et al. CVPR 2019)
- Introduces a significant computational overhead during training
  The EMA also introduces an additional memory requirement

Alternatives to batch normalization

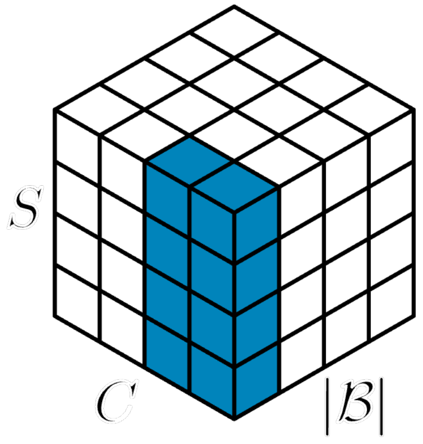
S: signal size ($H \times W$ for images), C: number of features/channels, $|B|$: minibatch size
Blue colors indicate the dimensions over which the averages are computed for the normalization



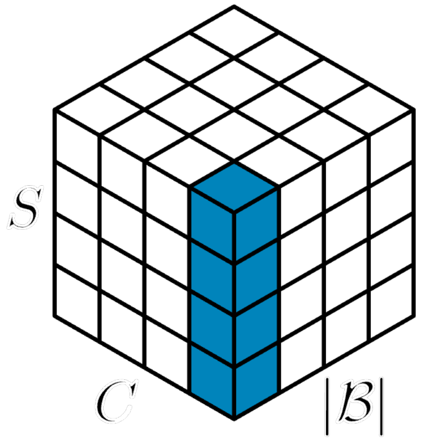
Batch Normalization



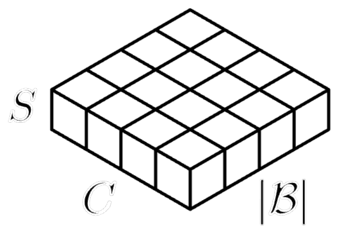
Layer Normalization



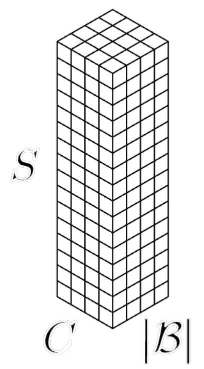
Group Normalization



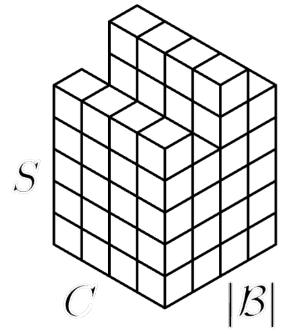
Instance Normalization



Tabular



Images



Sequential

Figures from: this source