Deep Learning and Optimization

Unpacking Transformers, LLMs and Diffusion

Session 1

olivier.koch@ensae.fr

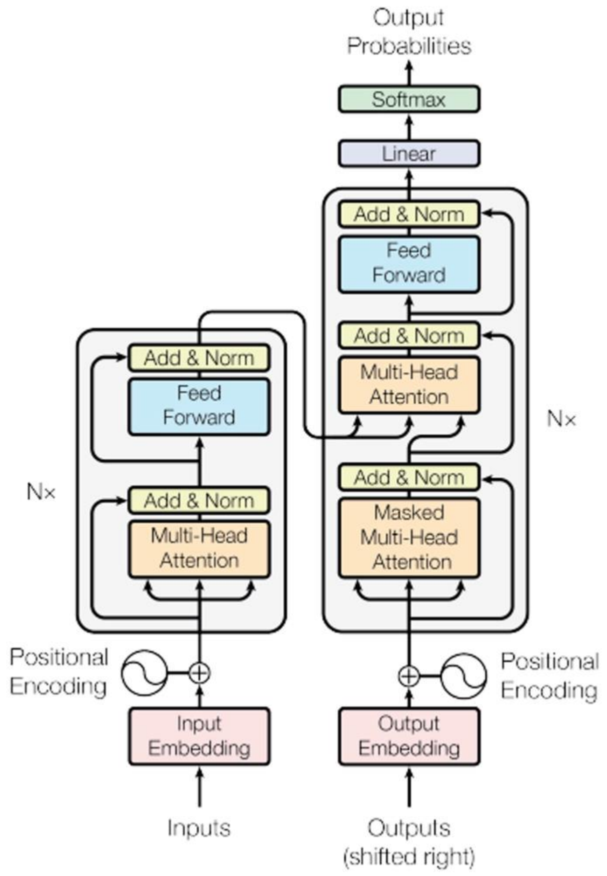
slack #ensae-dl-2026

What you will learn

- Concepts and practical aspects of deep learning
- Build your own backprop, mini-gpt and diffusion models

This course is <u>not</u> a complete overview of all ML & deep learning techniques.

Learning from scratch matters



mmm… so what?

Practical stuff

- 6 sessions
- 1.5h theory + 1.5h practice
- Grading: 6 notebooks (50%) + 1 quiz (50%)
- Each notebook should be yours
- I expect you to return your notebook at the end of each session.
- You can then send an updated version over the following week to aim for a better grade (and learn more).

On the usage of AI tools

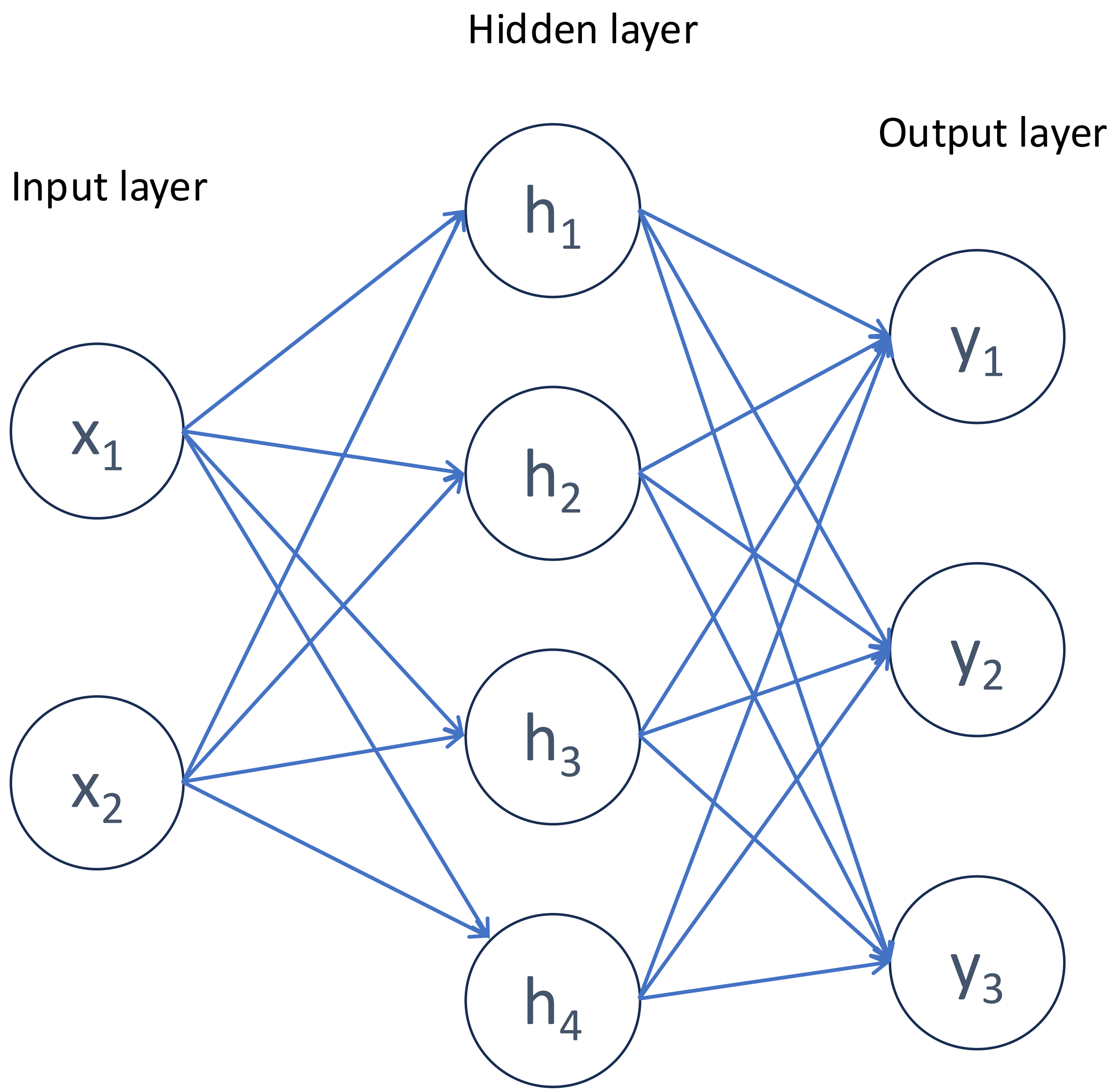
- This class will hardwire the fundamentals of AI in your brain
- You will only learn *by doing, from scratch*
- The use of AI tools in class is *discouraged*
- AI tools won’t help you during the quiz (on paper, open questions)
- AI tools are the future!

References

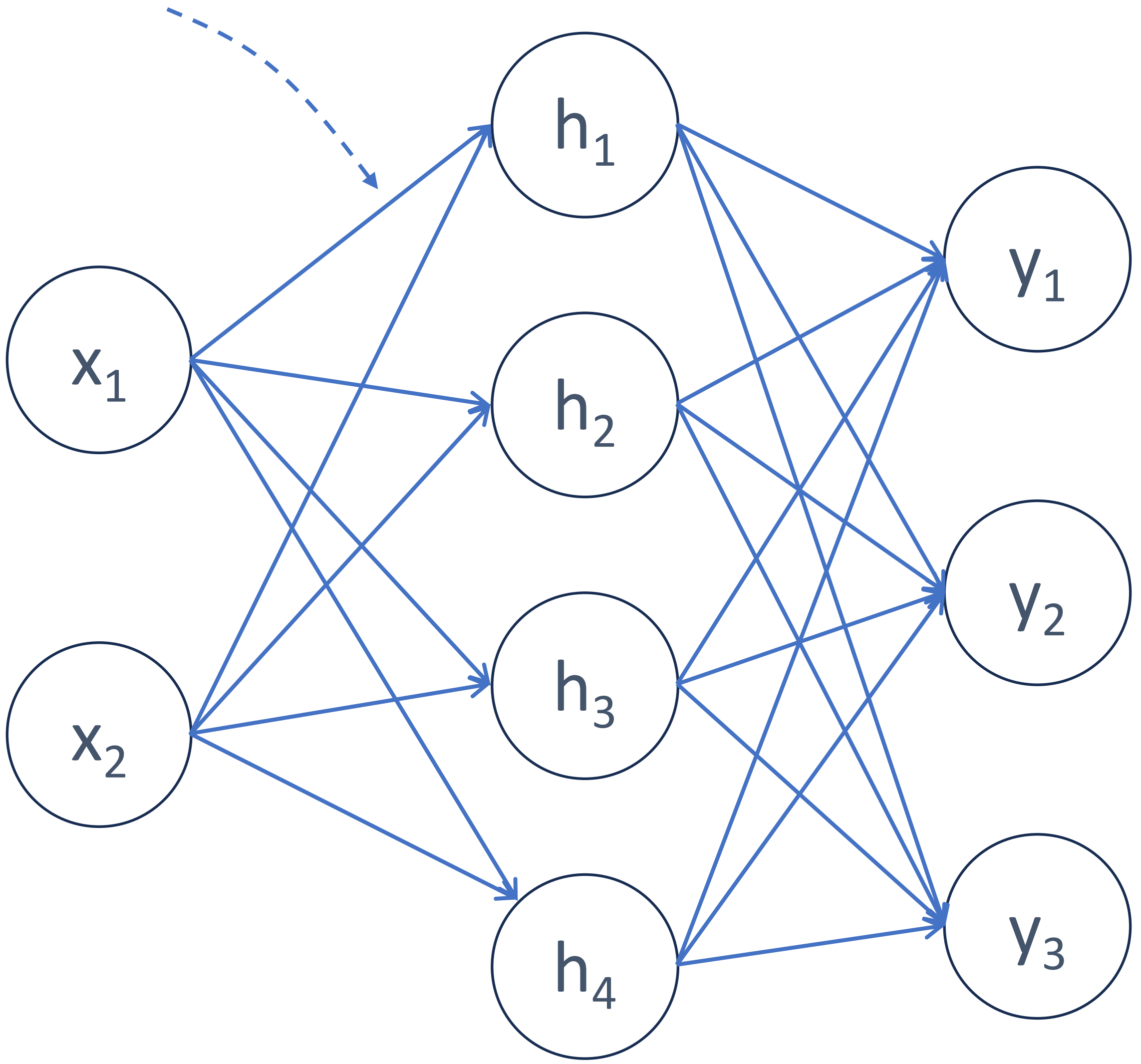
- Understanding Deep Learning, Simon Prince, December 2023
- Deep Learning: Foundations and Concepts, Christopher and Hugh Bishop, 2023
- Neural networks: Zero to Hero, Andrej Karpathy, Youtube Lecture Series, 2022

	Session	Date	Content
Foundations	1	Jan, 28	Intro to DL TP: micrograd
	2	Feb, 4	Fundamentals I: backprop, loss functions TP: bigram, MLP for next character prediction
	3	Feb, 11	Fundamentals II: DL architectures TP: tensor-based models
Applications	4	Feb, 18	Attention & Transformers TP: GPT from scratch
	5	Feb, 25	DL for Computer vision TP: convnets on CIFAR-10
	6	Mar, 3	VAE and Diffusion TP: diffusion from scratch Quiz / Exam

	Session	Date	Content
Foundations	1	Jan, 28	Intro to DL TP: micrograd
	2	Feb, 4	Fundamentals I: backprop, loss functions TP: bigram, MLP for next character prediction
	3	Feb, 11	Fundamentals II: DL architectures TP: tensor-based models
Applications	4	Feb, 18	Attention & Transformers TP: GPT from scratch
	5	Feb, 25	DL for Computer vision TP: convnets on CIFAR-10
	6	Mar, 3	VAE and Diffusion TP: diffusion from scratch Quiz / Exam


Hidden layer
Output layer
Input layer
x_1
x_2
h_1
h_2
h_3
h_4
y_1
y_2
y_3

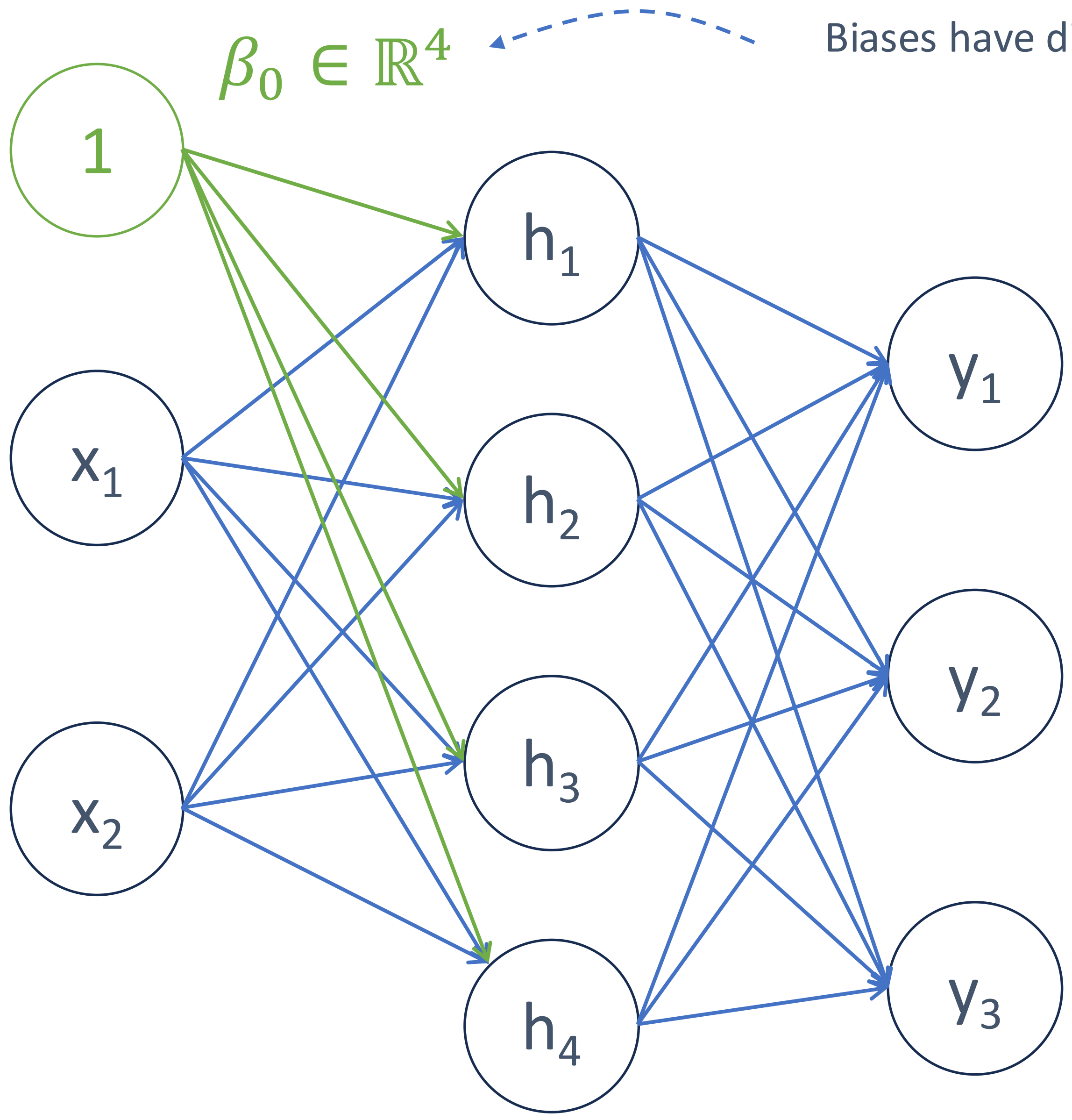
Linear transformation and non-linear activation



$$h_i = a(\beta_i + \sum \theta_{ij} x_j)$$

$$y_i = \gamma_i + \sum \varphi_{ij} h_j$$

Biases have dimension of the layer they feed *into*



Activation functions add non-linearity to the network. Will be discussed in Session 3!

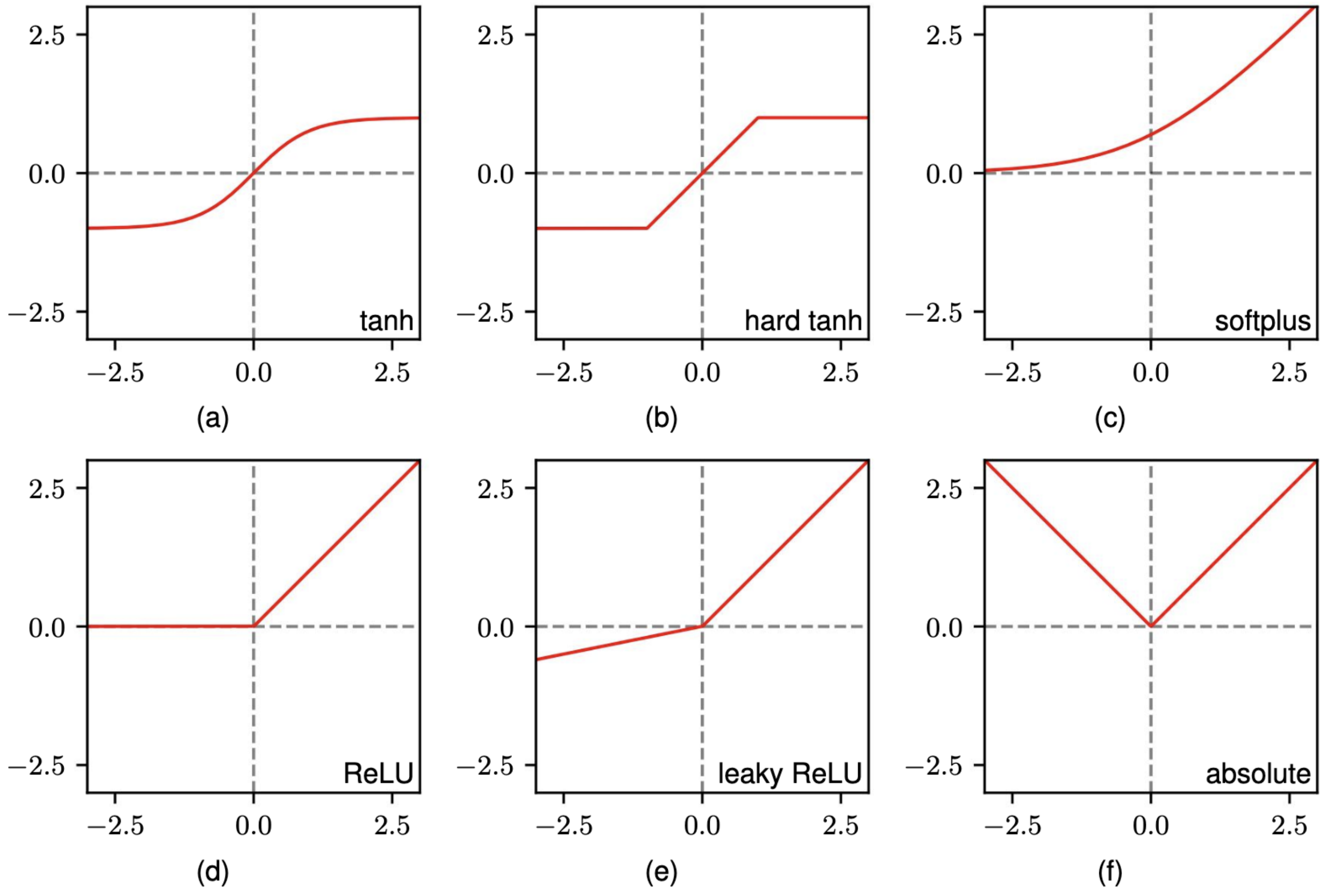


Figure 6.12 A variety of nonlinear activation functions.

Source: Deep Learning Foundations and Concept, Bishop, 2023

Backprop and gradient descent

$$\begin{tikzcd} x \arrow[r] & f_\omega \arrow[r] & y \end{tikzcd}$$

$$\ell = (y - \hat{y})^2$$

ground-truth

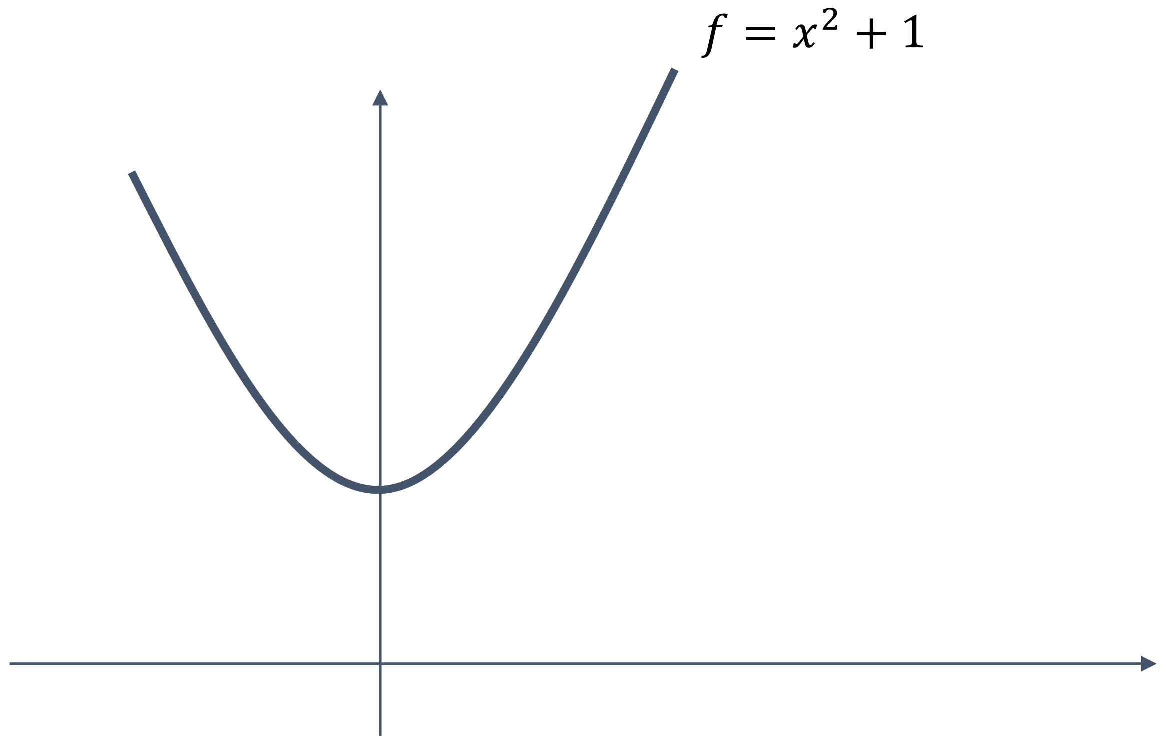
Backprop and gradient descent

$$\begin{tikzcd} x \arrow[r] & f_\omega \arrow[r] & \ell \end{tikzcd}$$

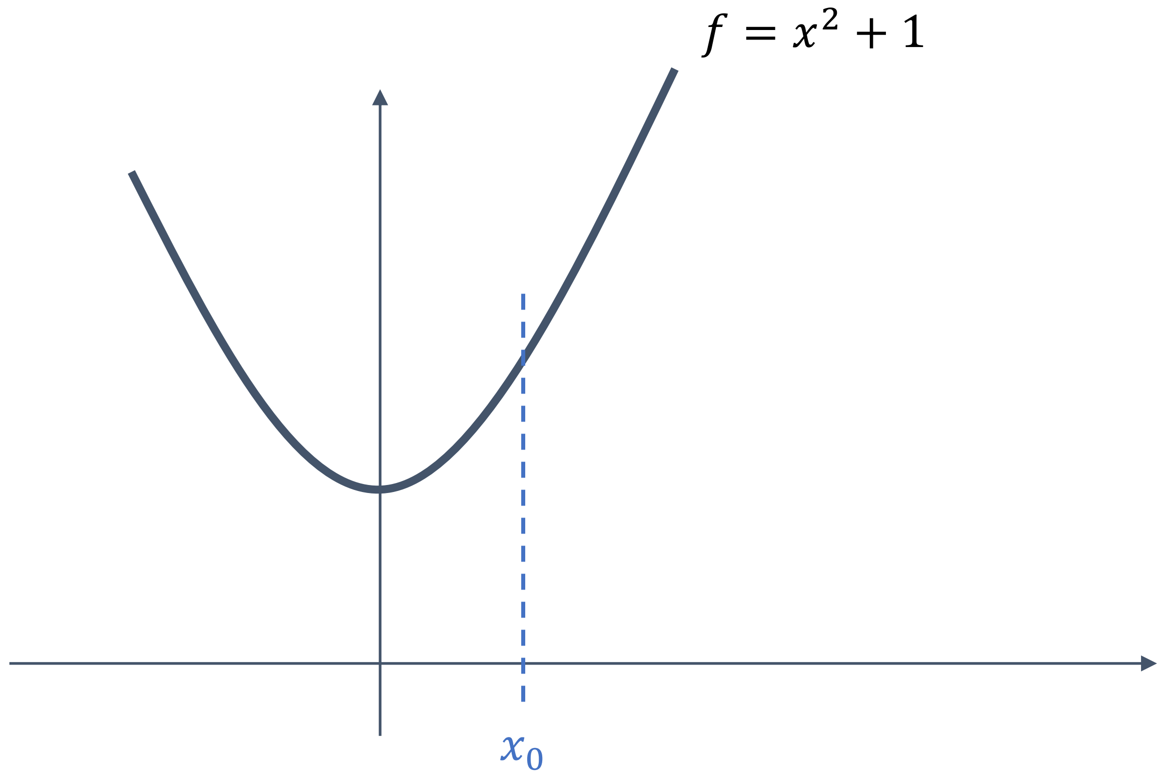
Goal: find ω that minimizes:

$$\ell = f_\omega(x)$$

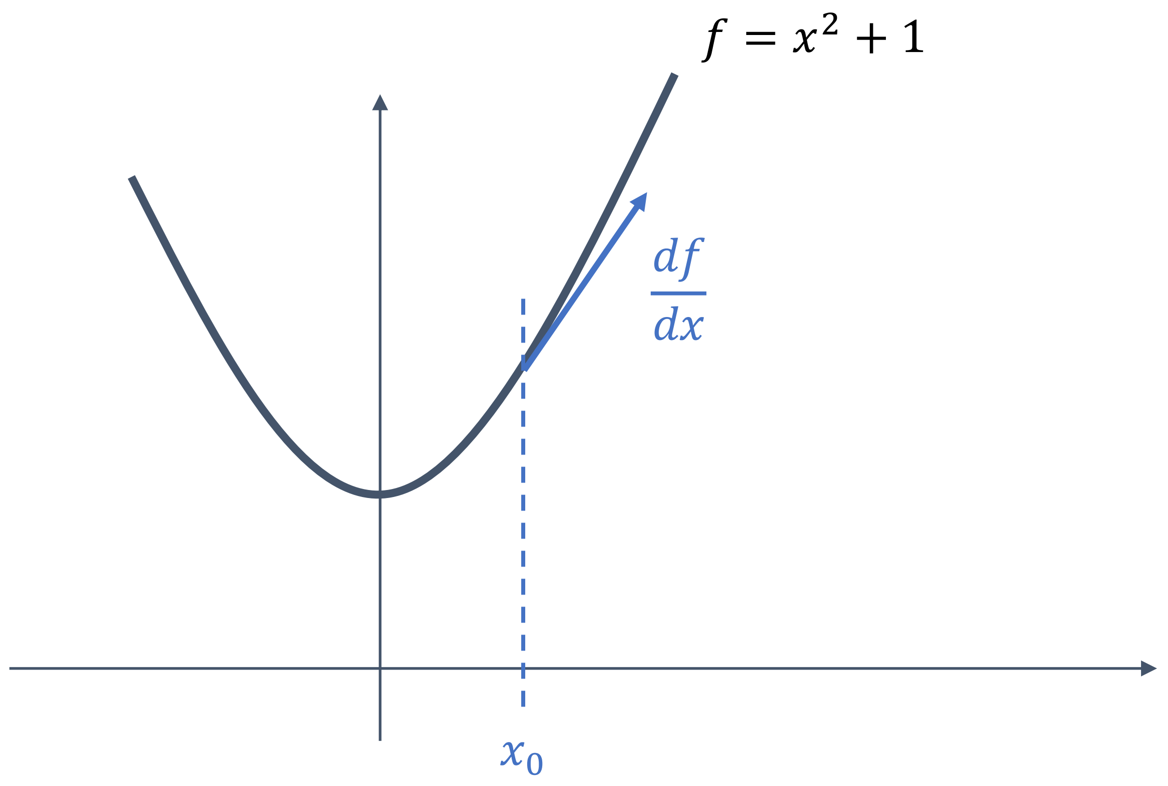
Gradient descent



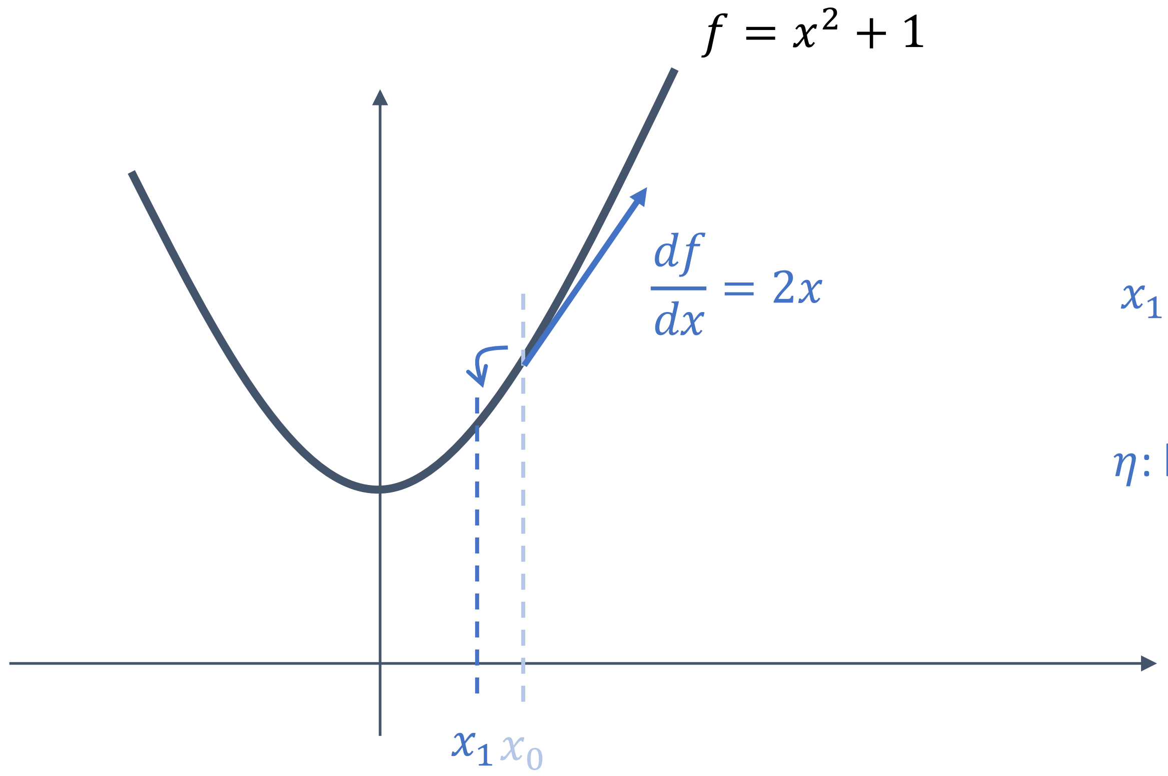
Gradient descent



Gradient descent



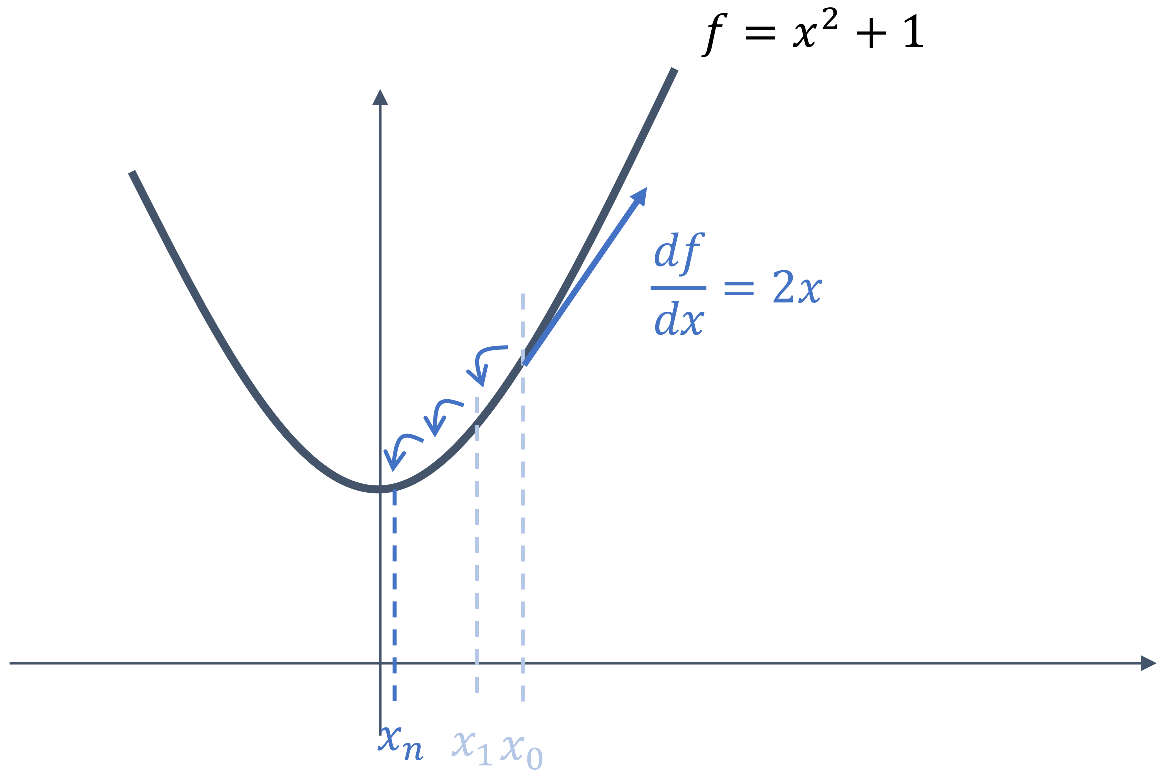
Gradient descent



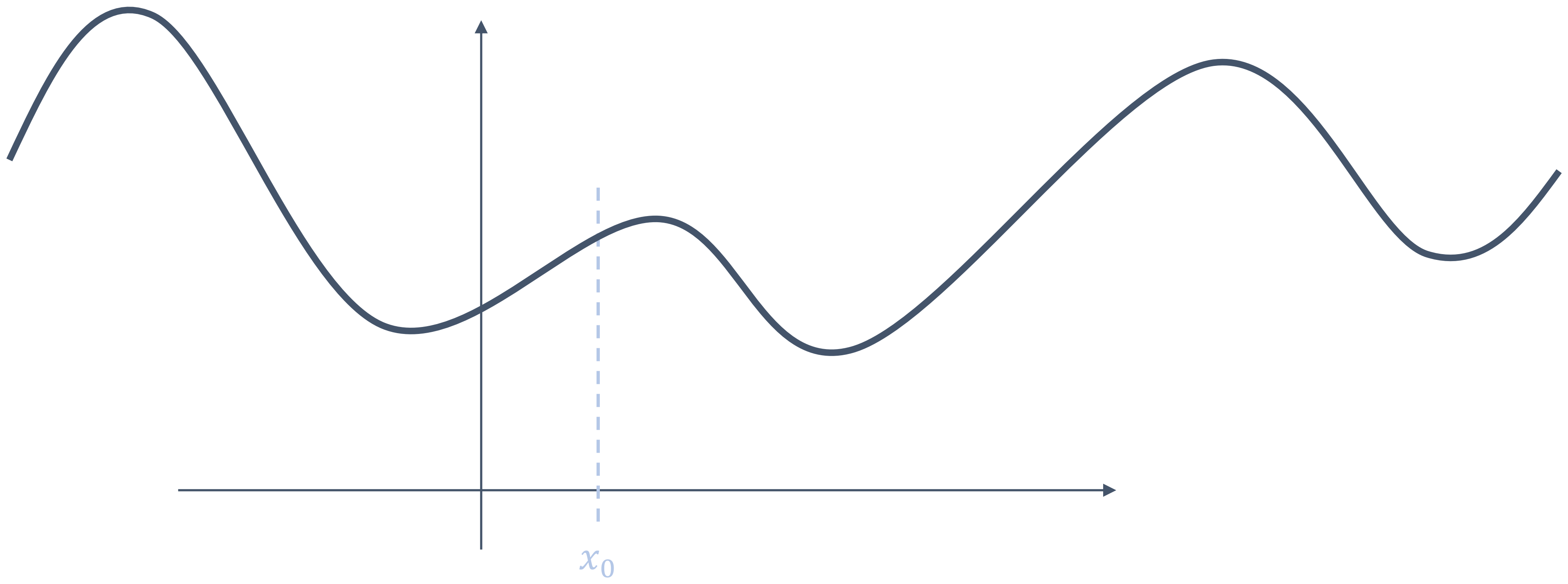
$$x_1 = x_0 - \eta \frac{df}{dx}$$

η: learning rate

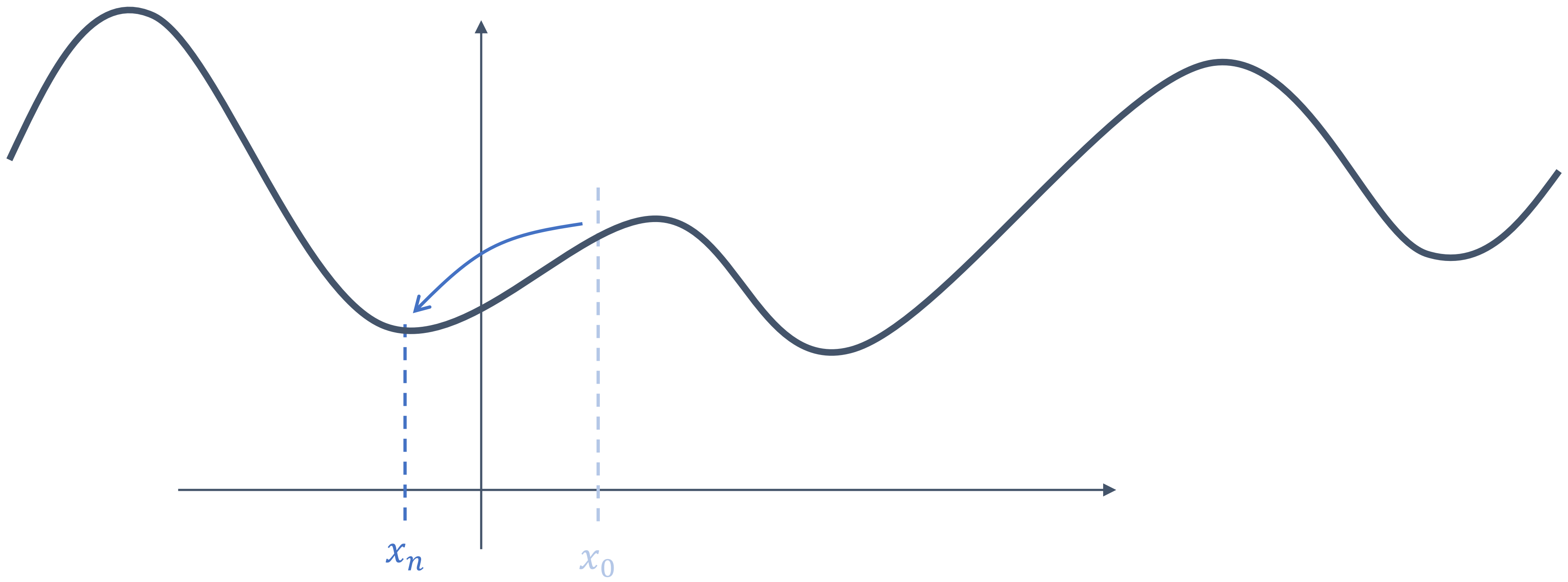
Gradient descent



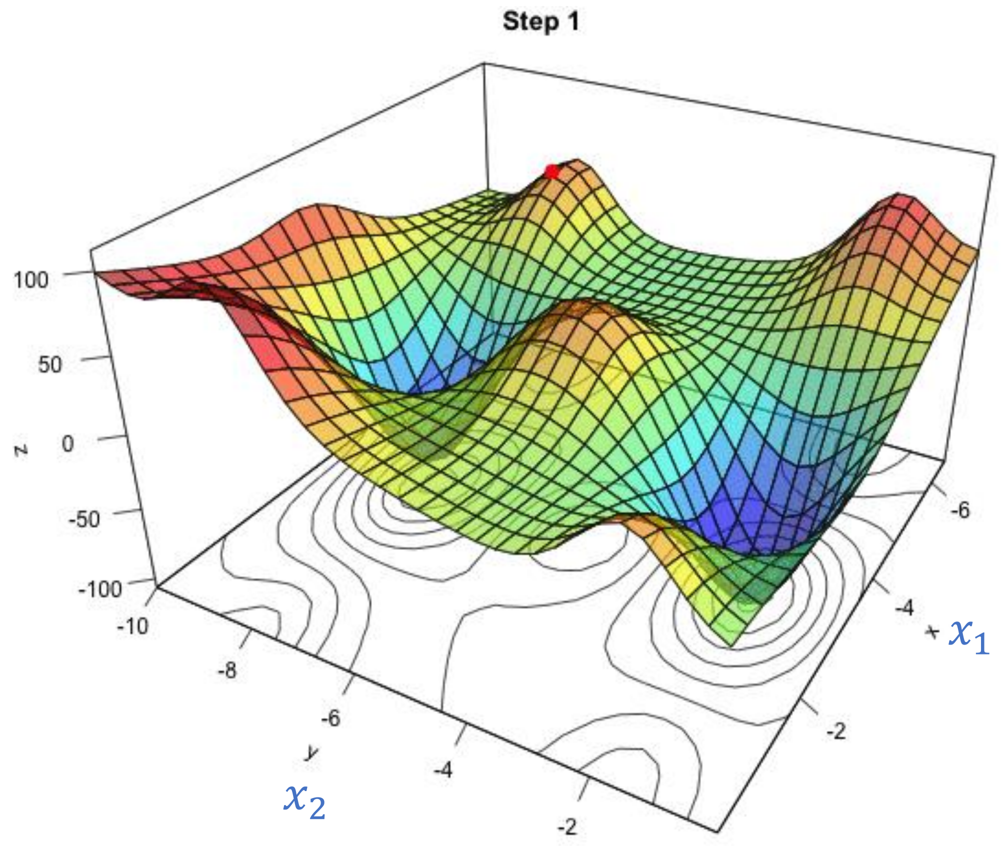
Gradient descent



Gradient descent

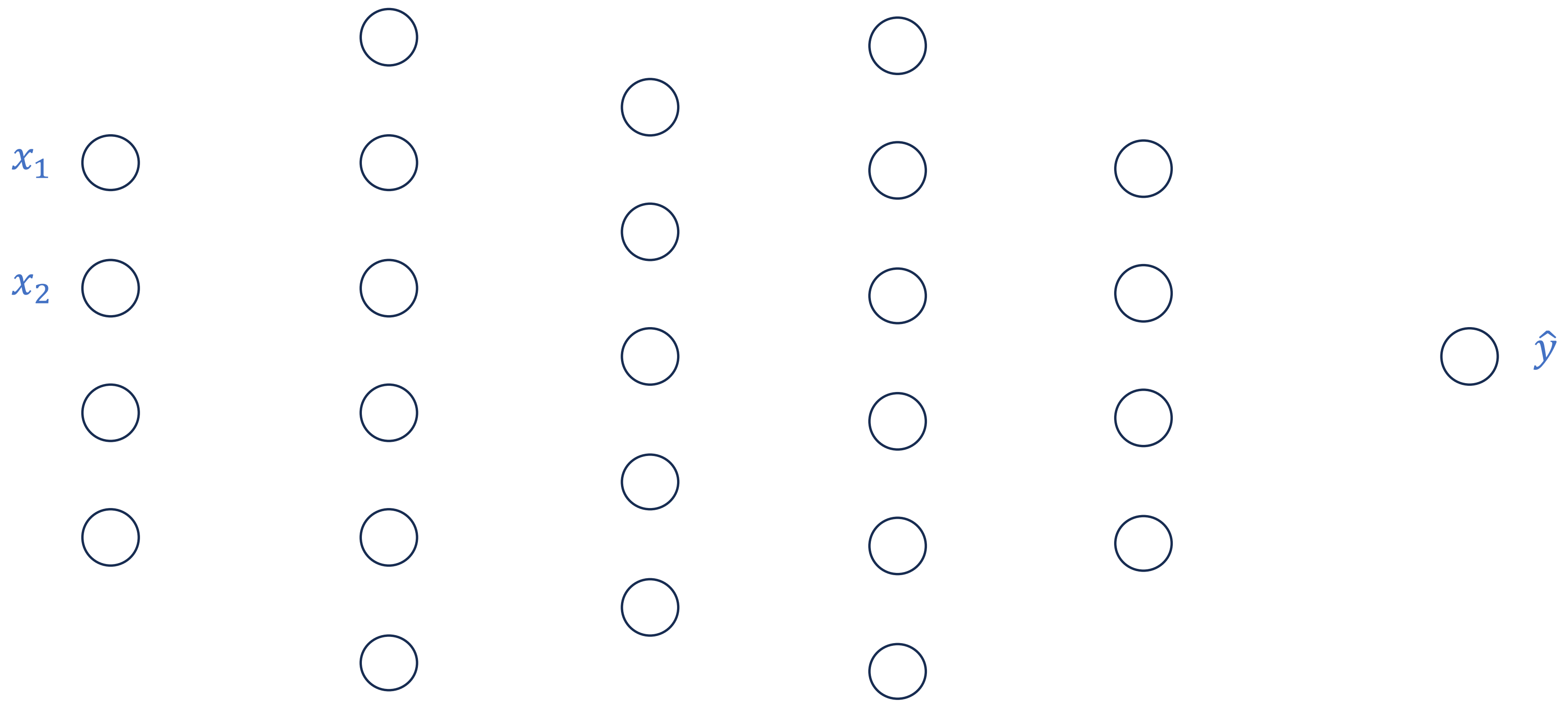


Gradient descent

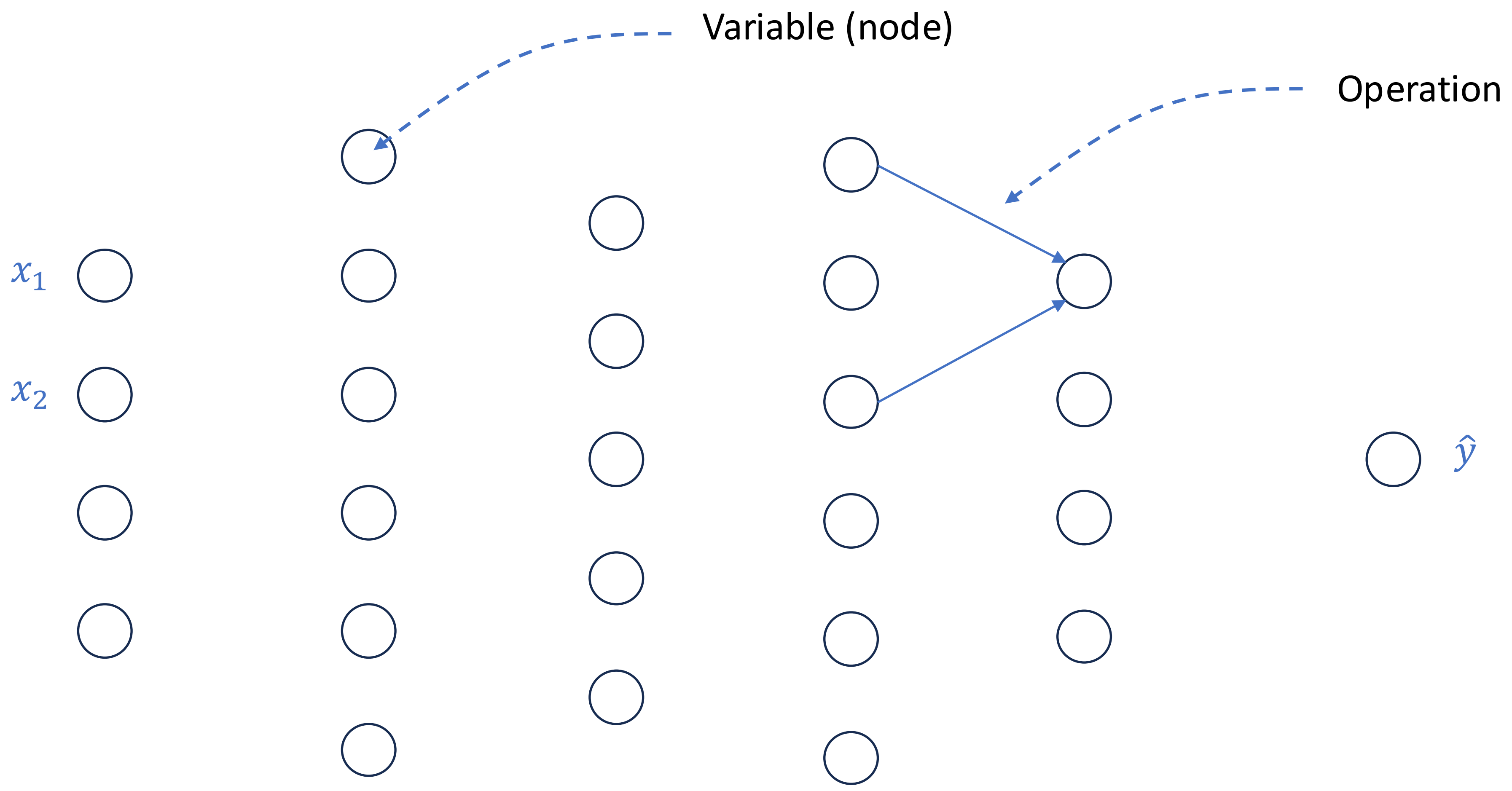


$$\begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix}$$

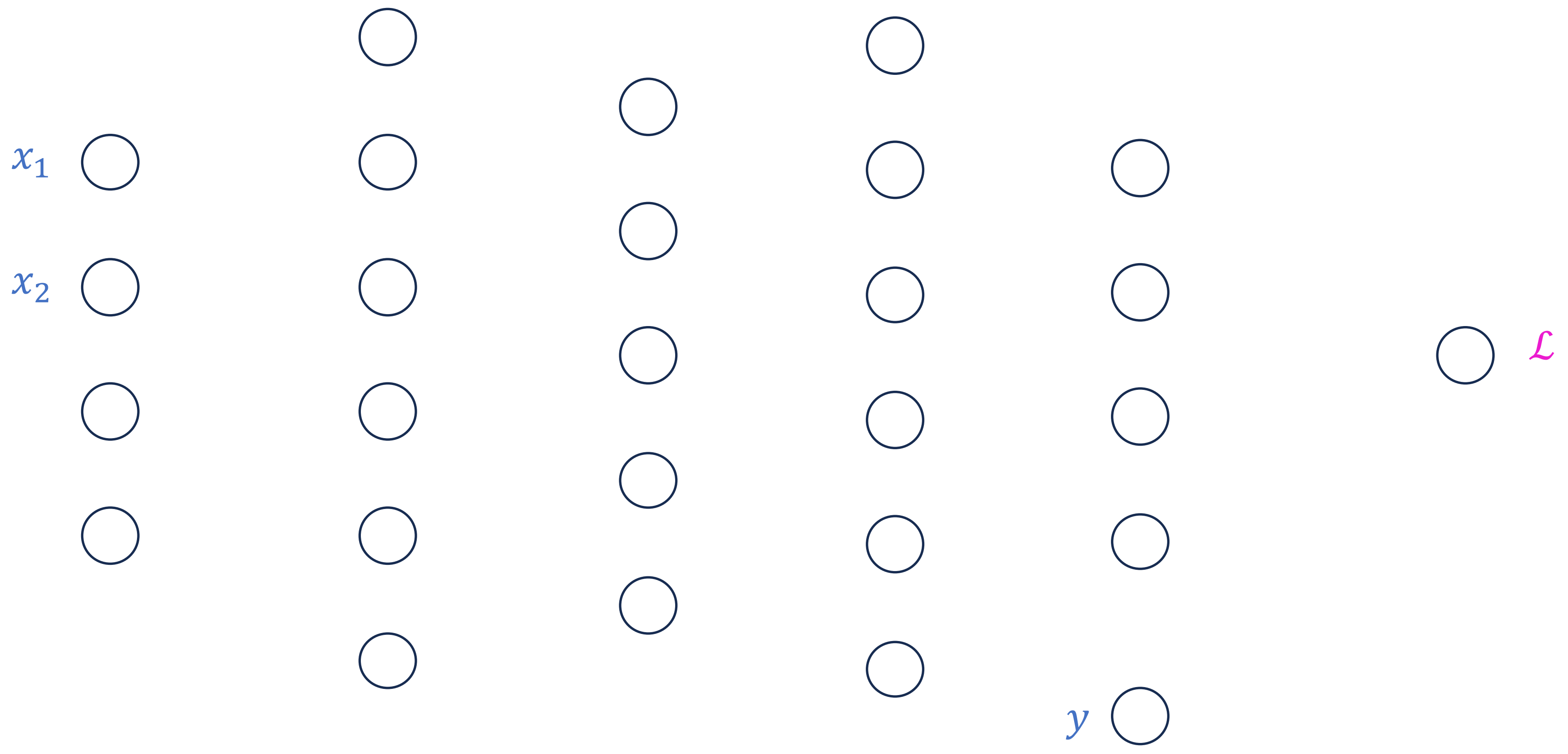
Gradient descent



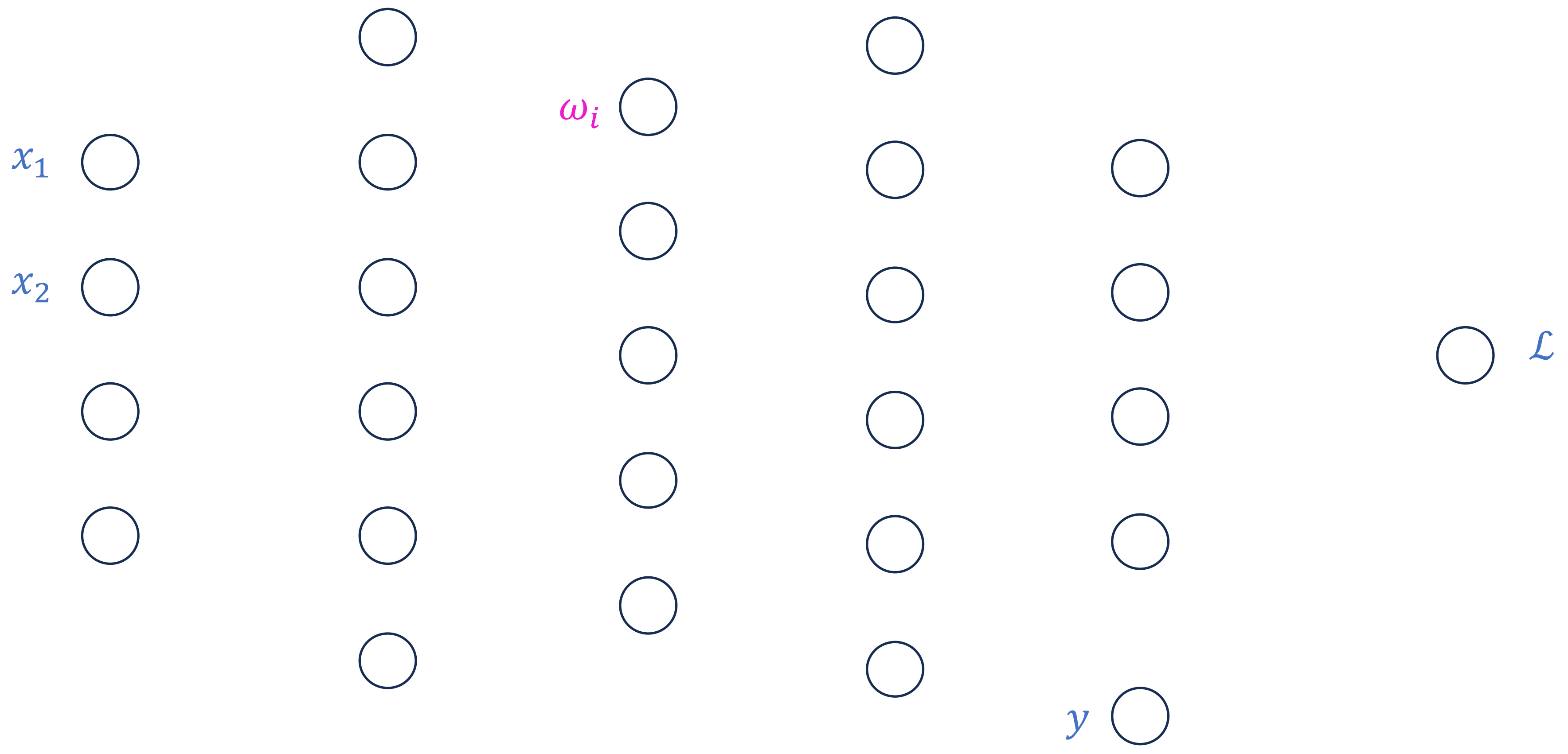
Gradient descent



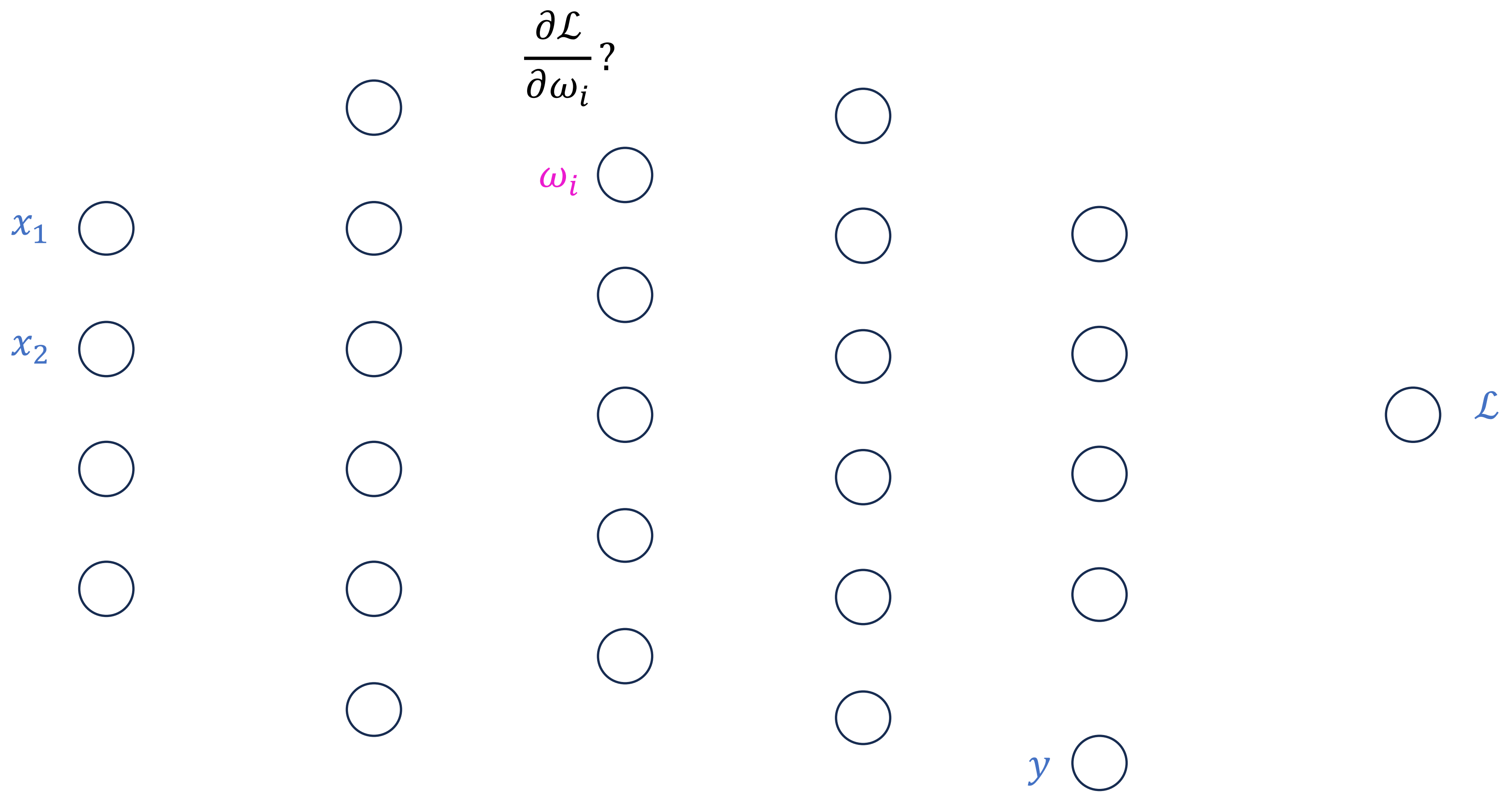
Gradient descent



Gradient descent

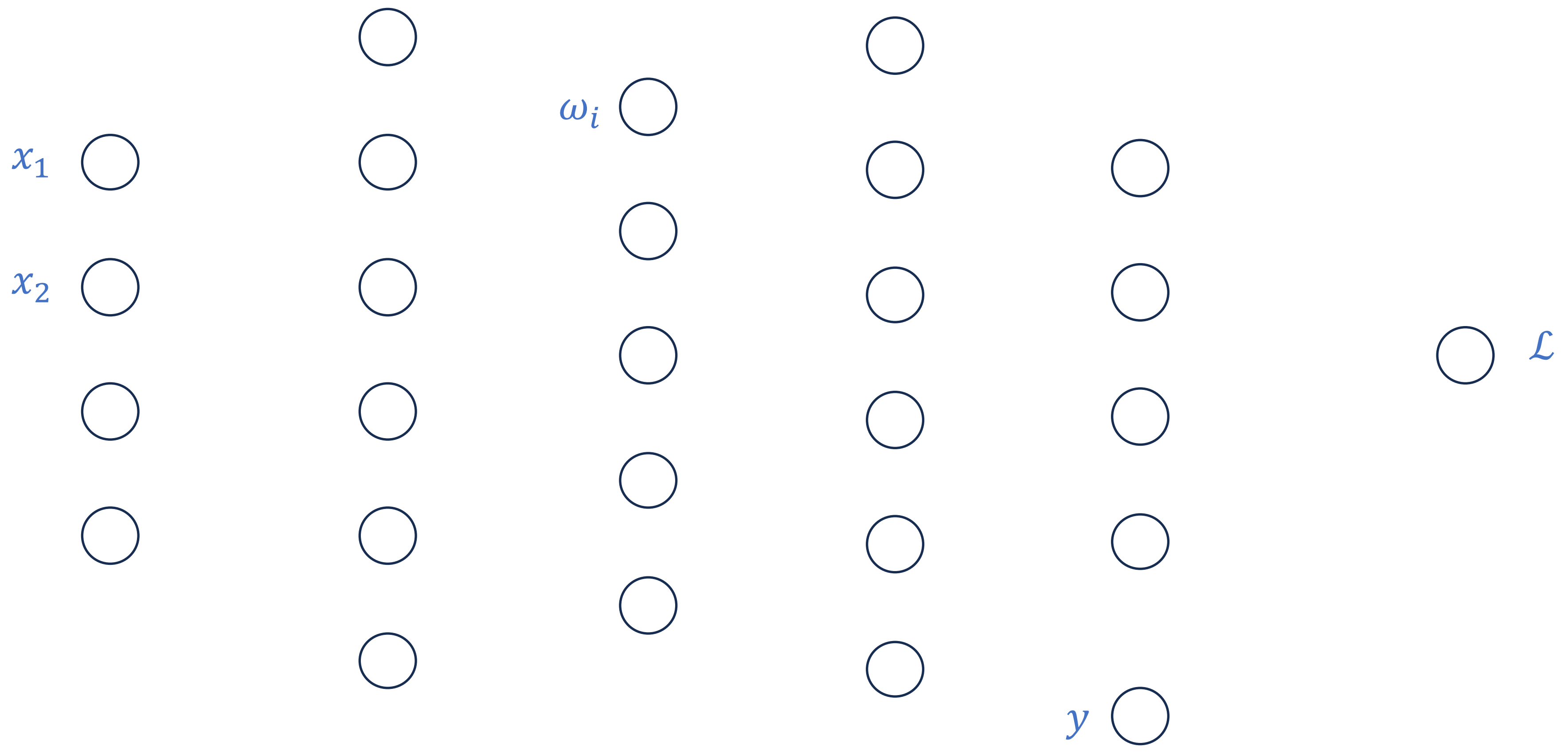


Gradient descent



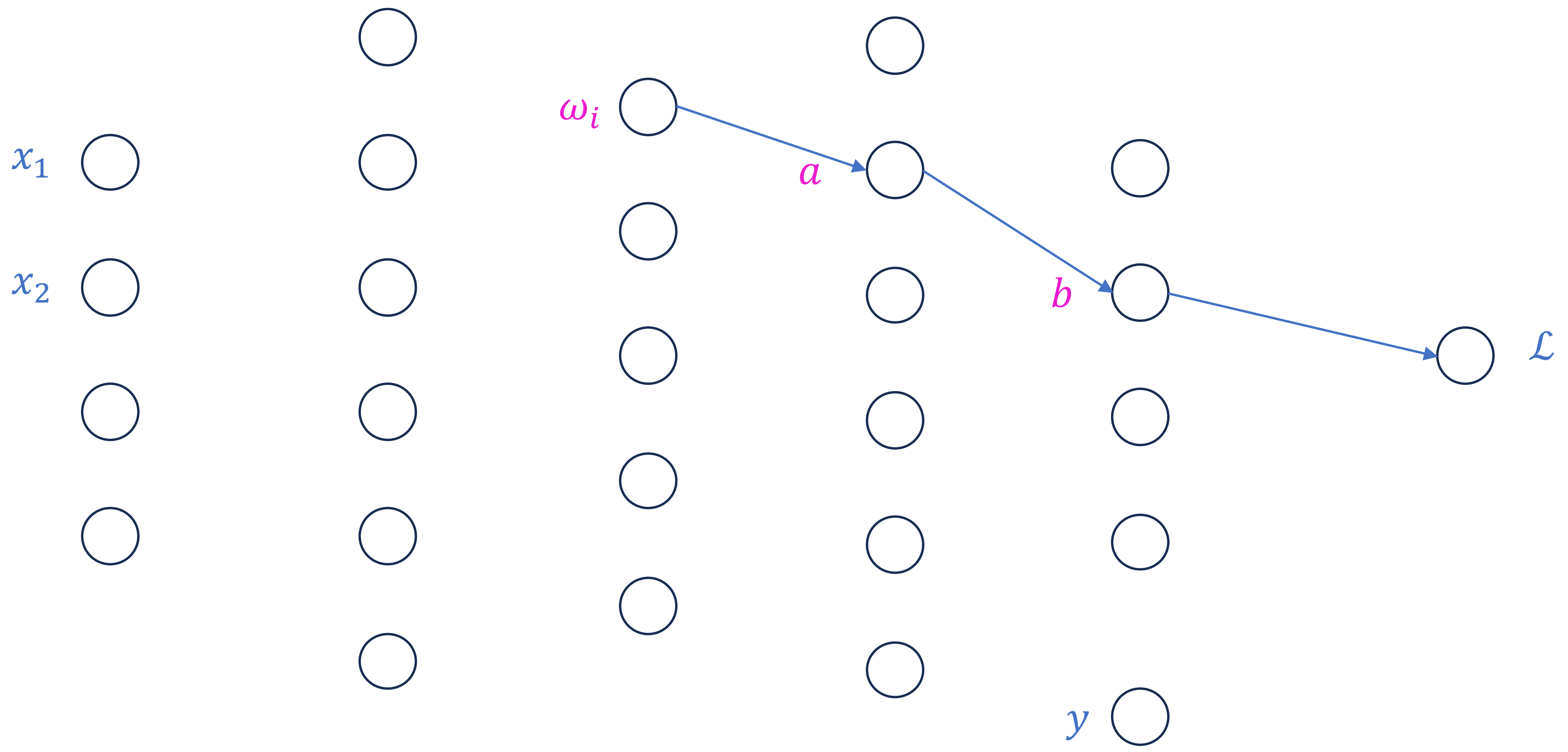
Gradient descent

$$\omega_i \mathrel{-}= \eta \frac{\partial \mathcal{L}}{\partial \omega_i}$$



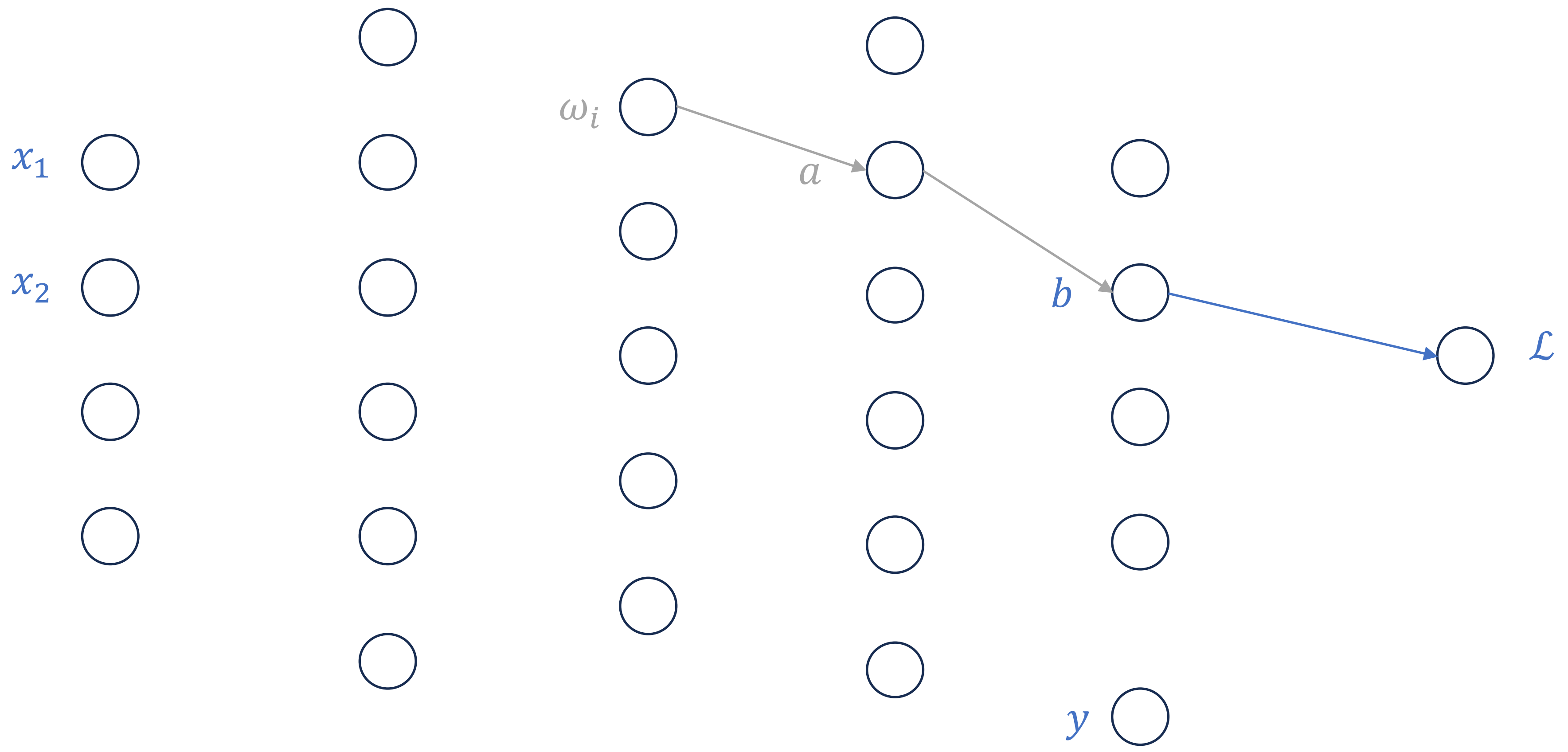
Chain rule

$$\frac{\partial \mathcal{L}}{\partial \omega_i} = \frac{\partial \mathcal{L}}{\partial b} \frac{\partial b}{\partial a} \frac{\partial a}{\partial \omega_i}$$



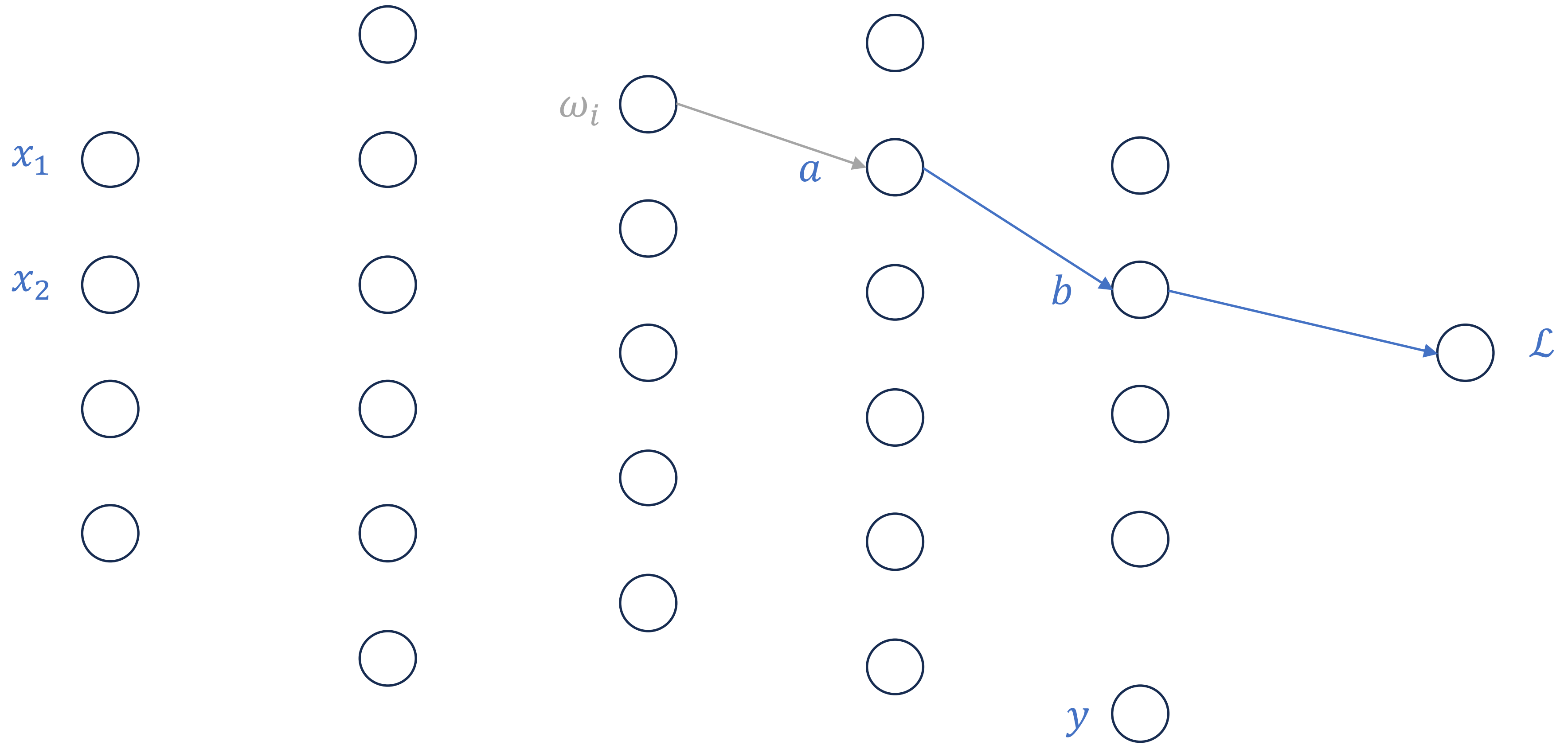
Chain rule

$$\frac{\partial \mathcal{L}}{\partial \omega_i} = \frac{\partial \mathcal{L}}{\partial b}\frac{\partial b}{\partial a}\frac{\partial a}{\partial \omega_i}$$



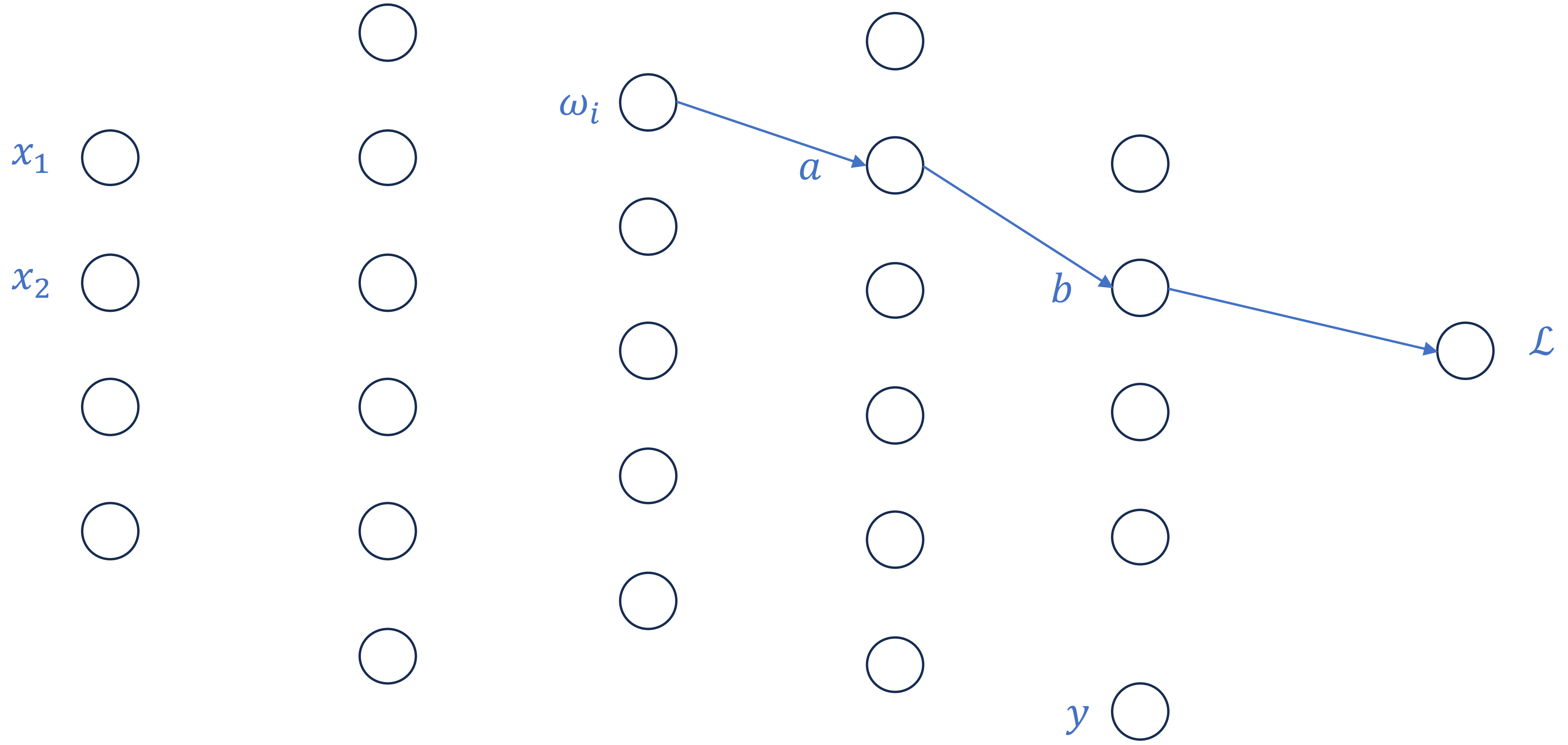
Chain rule

$$\frac{\partial \mathcal{L}}{\partial \omega_i} = \frac{\partial \mathcal{L}}{\partial b} \frac{\partial b}{\partial a} \frac{\partial a}{\partial \omega_i}$$



Chain rule

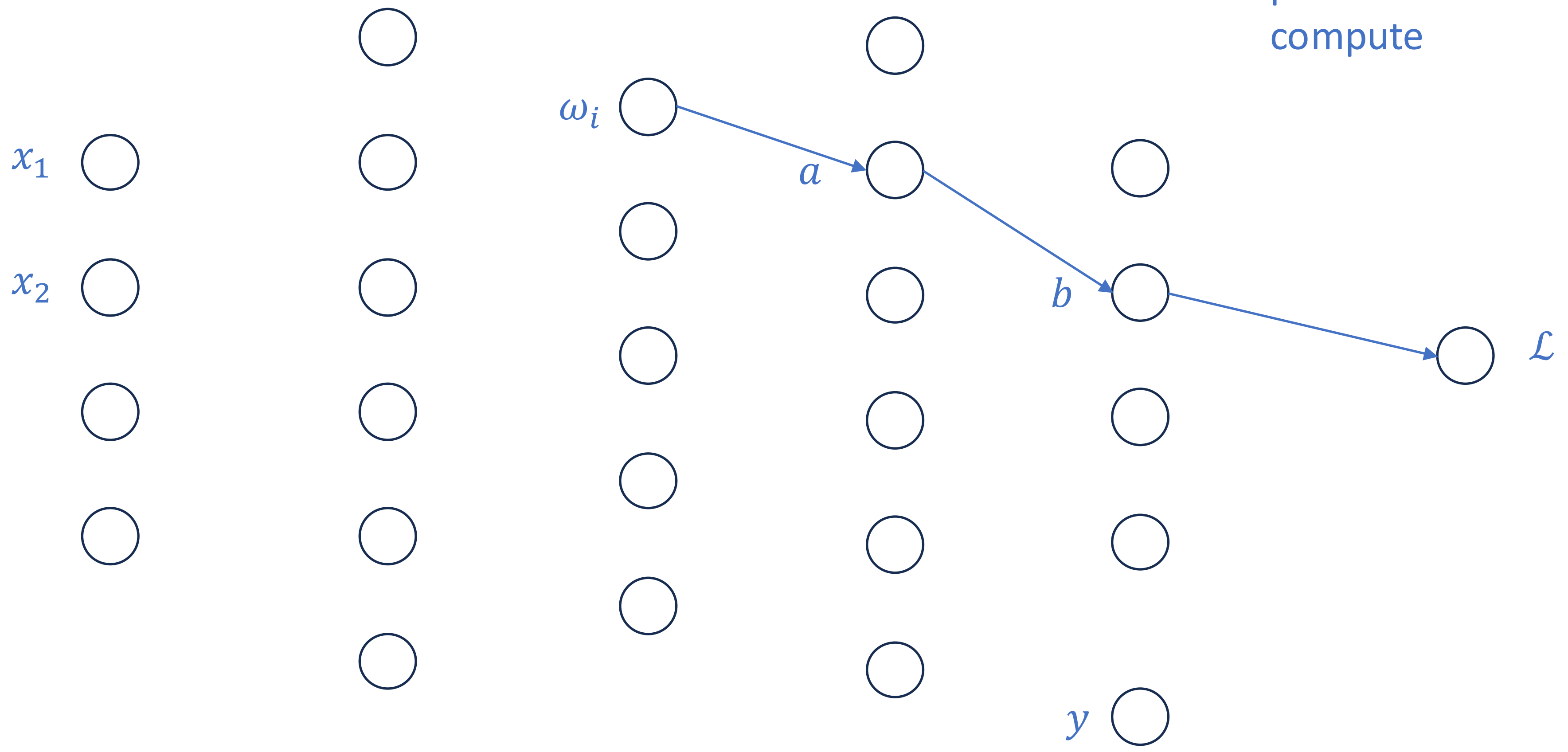
$$\frac{\partial \mathcal{L}}{\partial \omega_i} = \frac{\partial \mathcal{L}}{\partial b}\frac{\partial b}{\partial a}\frac{\partial a}{\partial \omega_i}$$



Chain rule

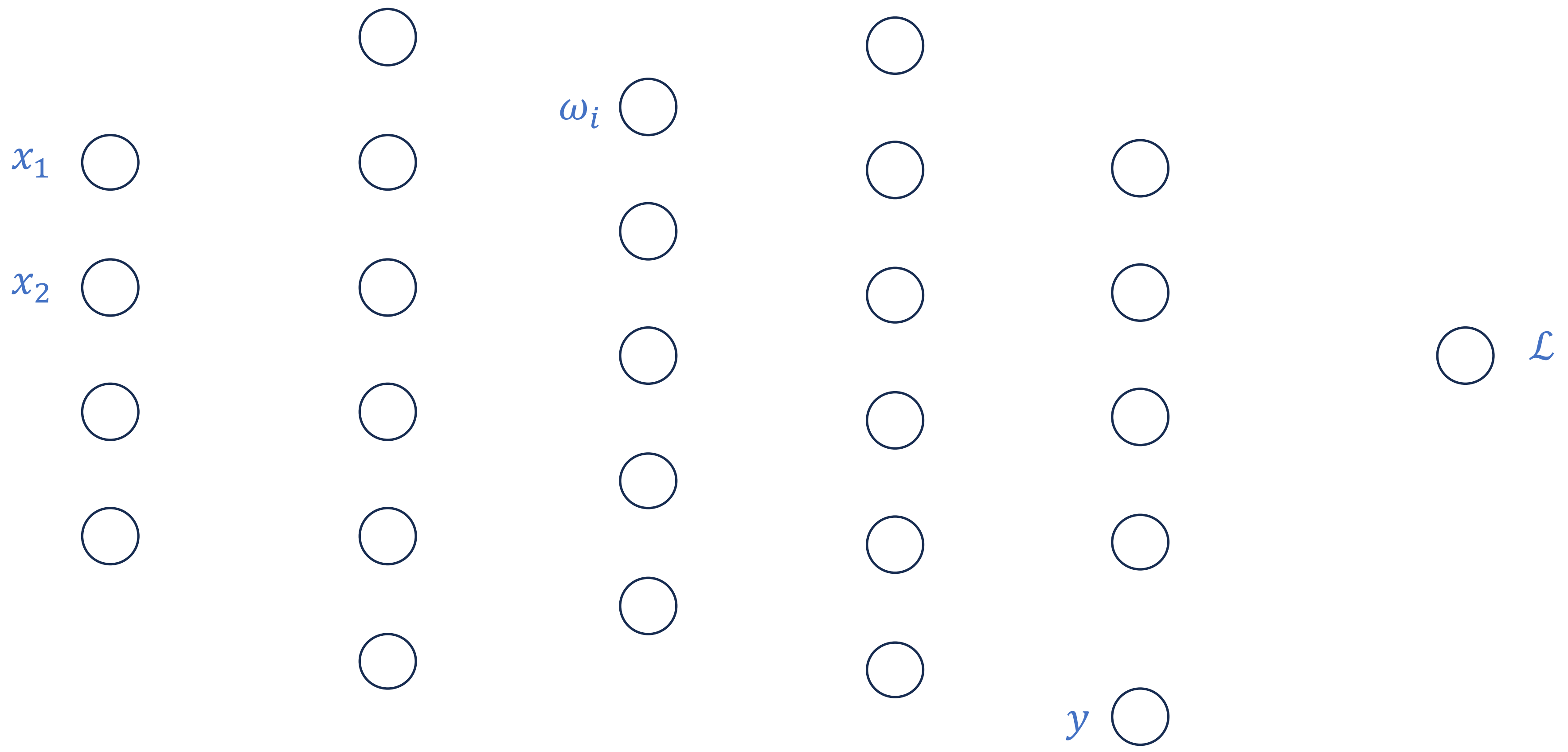
$$\frac{\partial \mathcal{L}}{\partial \omega_i} = \frac{\partial \mathcal{L}}{\partial b}\frac{\partial b}{\partial a}\frac{\partial a}{\partial \omega_i}$$

In a neural network, all these partial derivatives are easy to compute



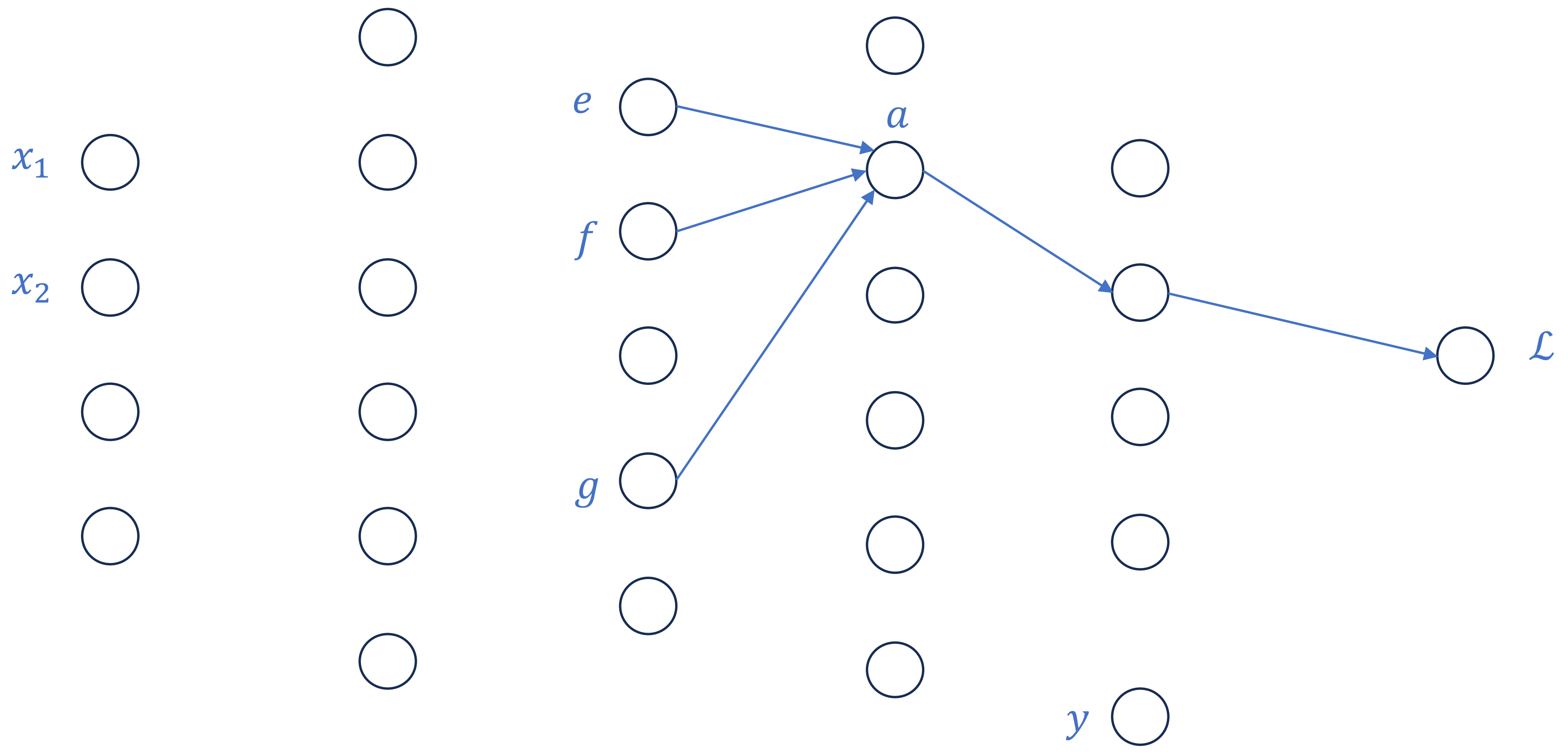
Chain rule

$\frac{\partial \mathcal{L}}{\partial x_i}$?!?



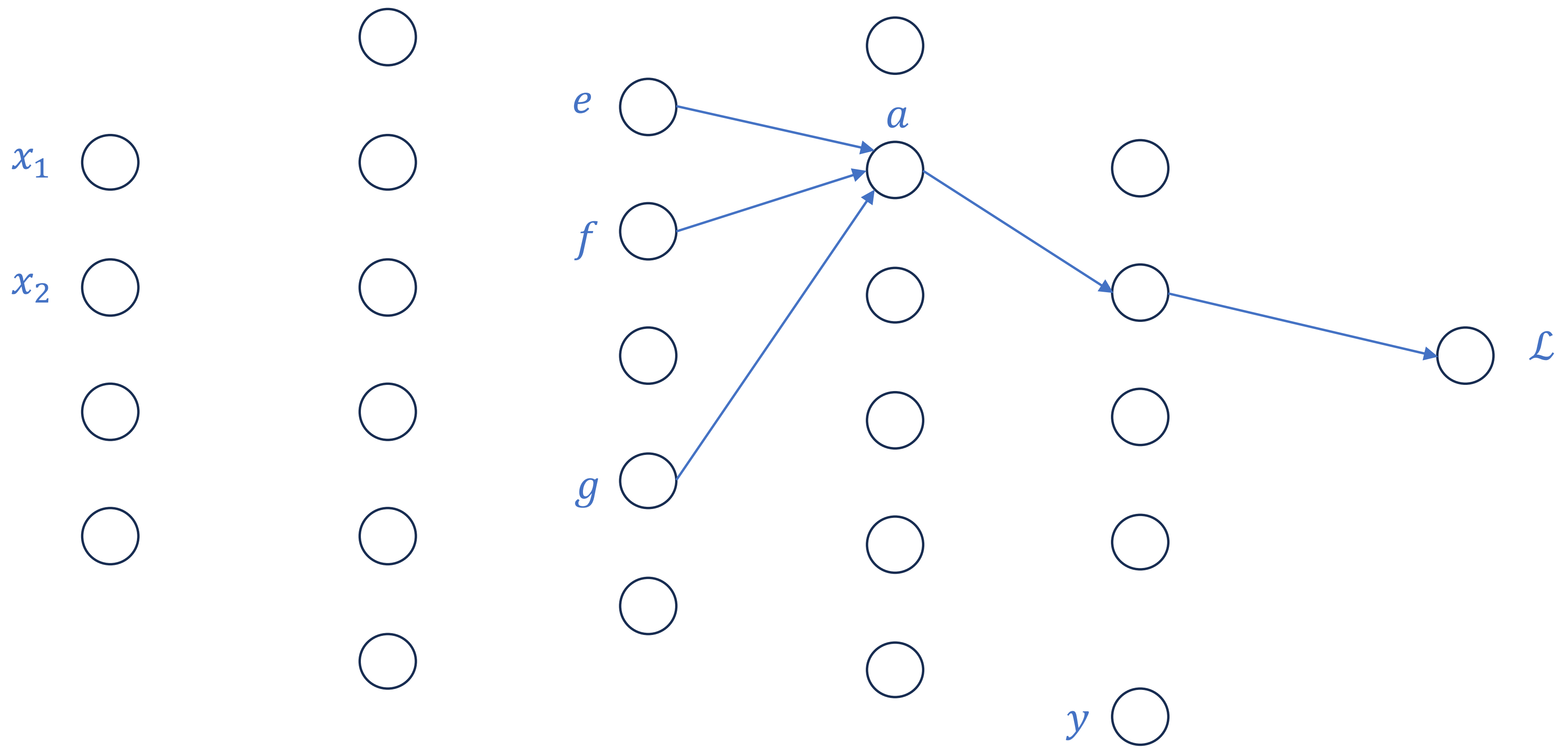
The gradient flow

$\frac{\partial \mathcal{L}}{\partial a}$ "flows" backward to e, f and g



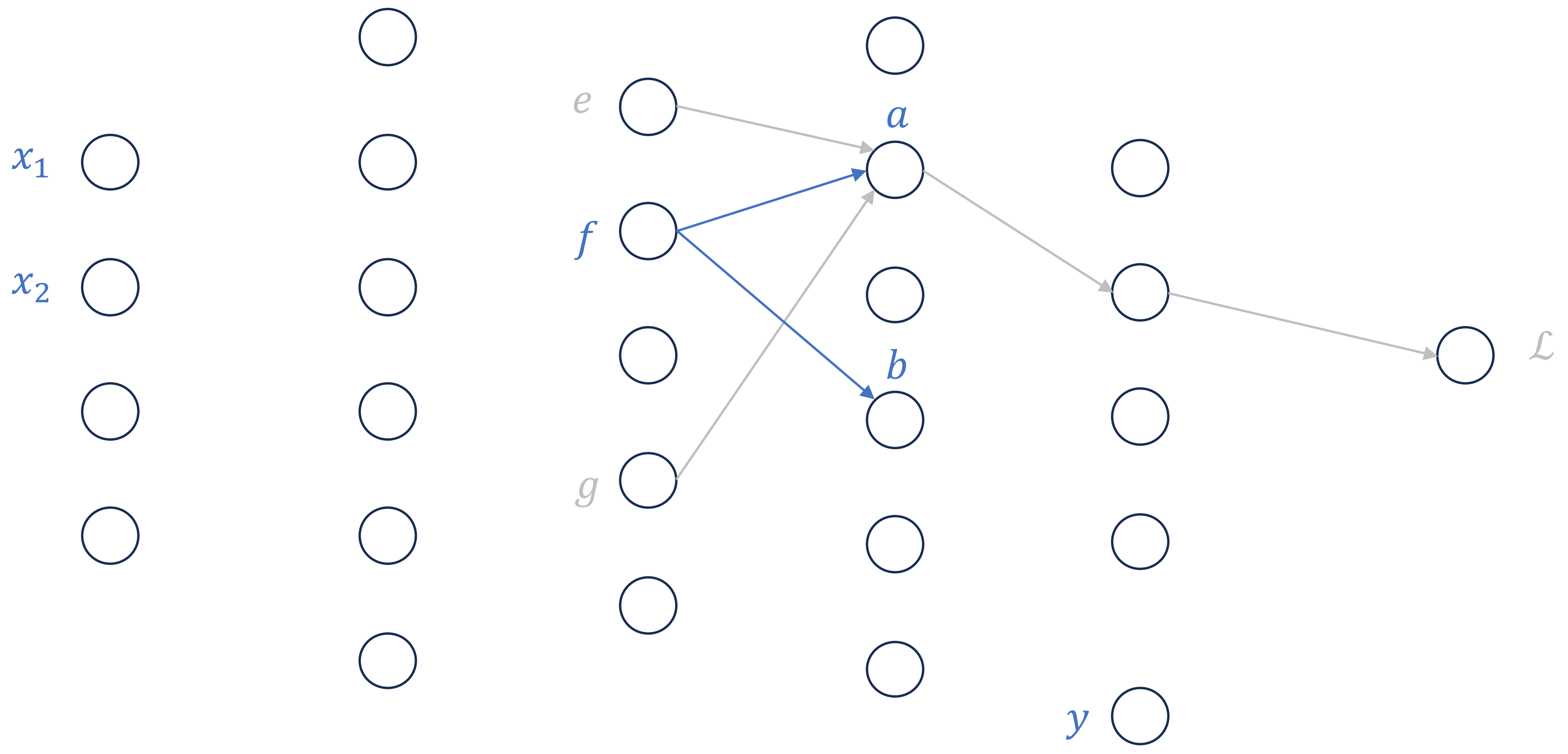
The gradient flow

Given $\frac{\partial \mathcal{L}}{\partial a}$, we can update $\frac{\partial \mathcal{L}}{\partial e}$, $\frac{\partial \mathcal{L}}{\partial f}$ and $\frac{\partial \mathcal{L}}{\partial g}$



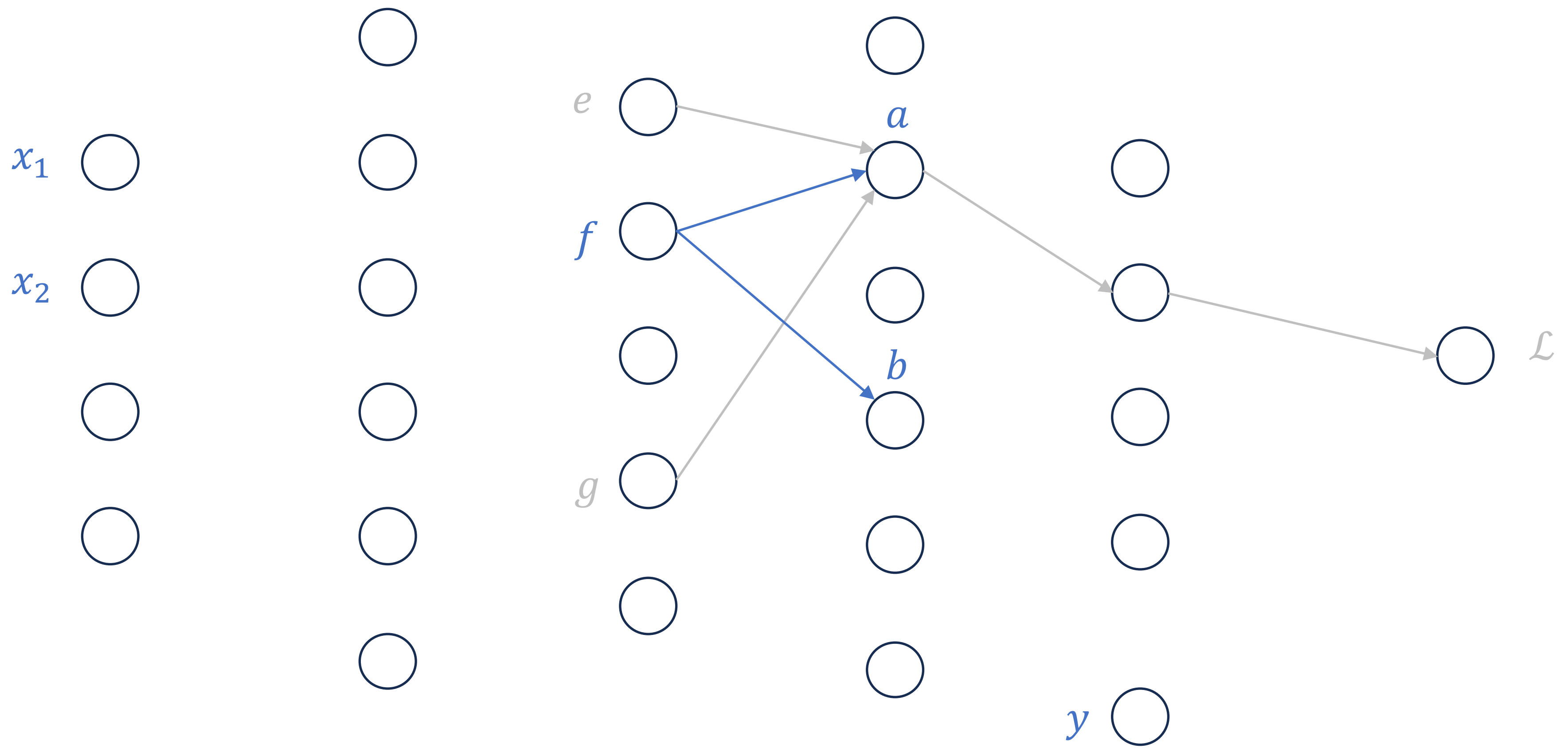
The gradient flow

$\frac{\partial \mathcal{L}}{\partial a}$ and $\frac{\partial \mathcal{L}}{\partial b}$ contribute to $\frac{\partial \mathcal{L}}{\partial f}$



The gradient flow

$$\frac{\partial \mathcal{L}}{\partial f} = \frac{\partial \mathcal{L}}{\partial a}\frac{\partial a}{\partial f} + \frac{\partial \mathcal{L}}{\partial b}\frac{\partial b}{\partial f}$$



Backprop and gradient descent

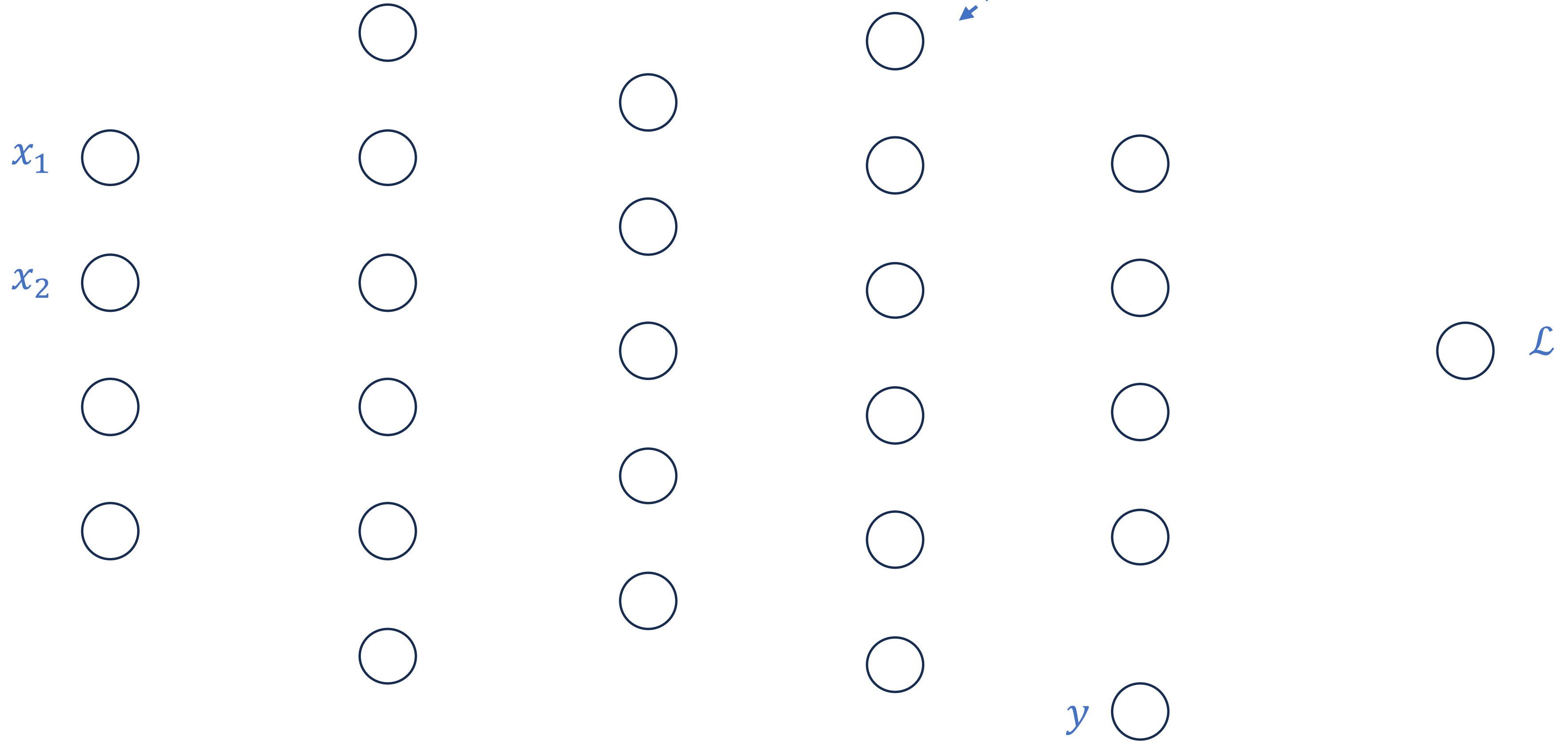
$$\mathcal{L} = f_\omega(x)$$

Assuming f_ω is smooth and differentiable w.r.t ω

```
ω = $\omega_0$
Repeat:
    Compute $\mathcal{L}$
    Reset gradients $\Delta\mathcal{L} = 0$
    Compute $\Delta\mathcal{L}$ (i.e. $\partial\mathcal{L}/\partial\omega_i$)
    $\omega_i = \omega_i - \eta . \partial\mathcal{L}/\partial\omega_i$
```

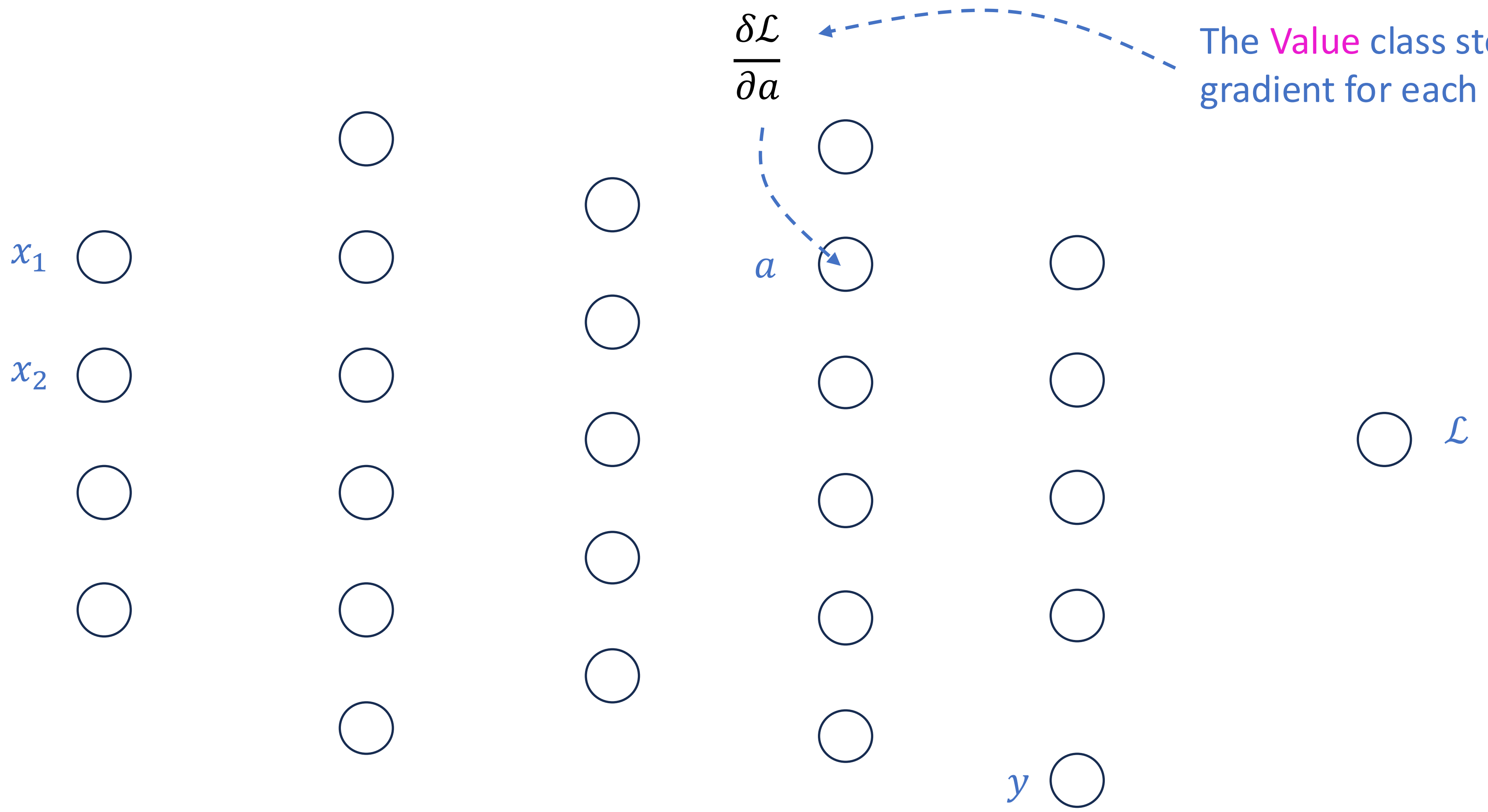
η: learning rate

Let’s build it by hand!



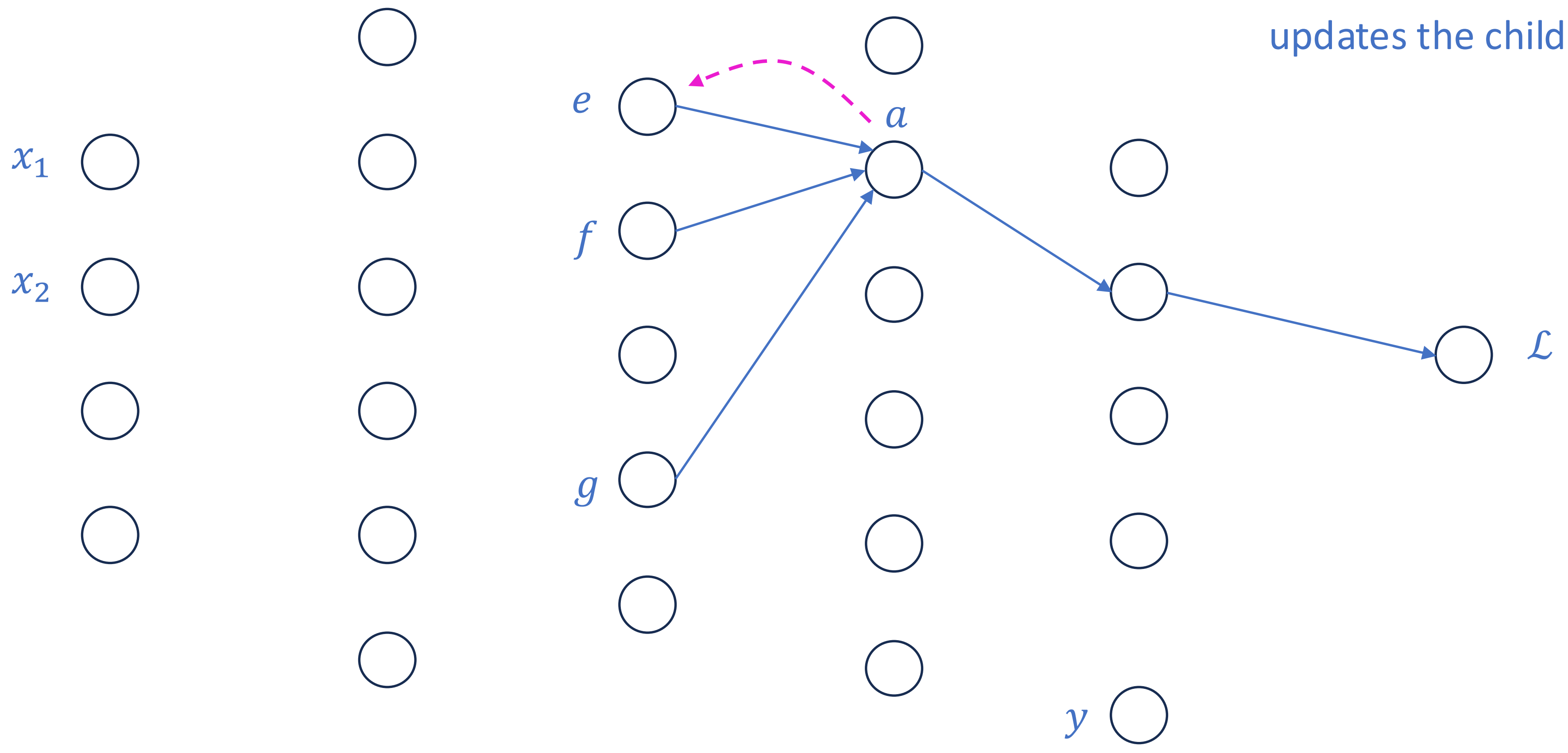
The Value class represents a node

Let’s build it by hand!



The Value class stores this gradient for each node

Let’s build it by hand!

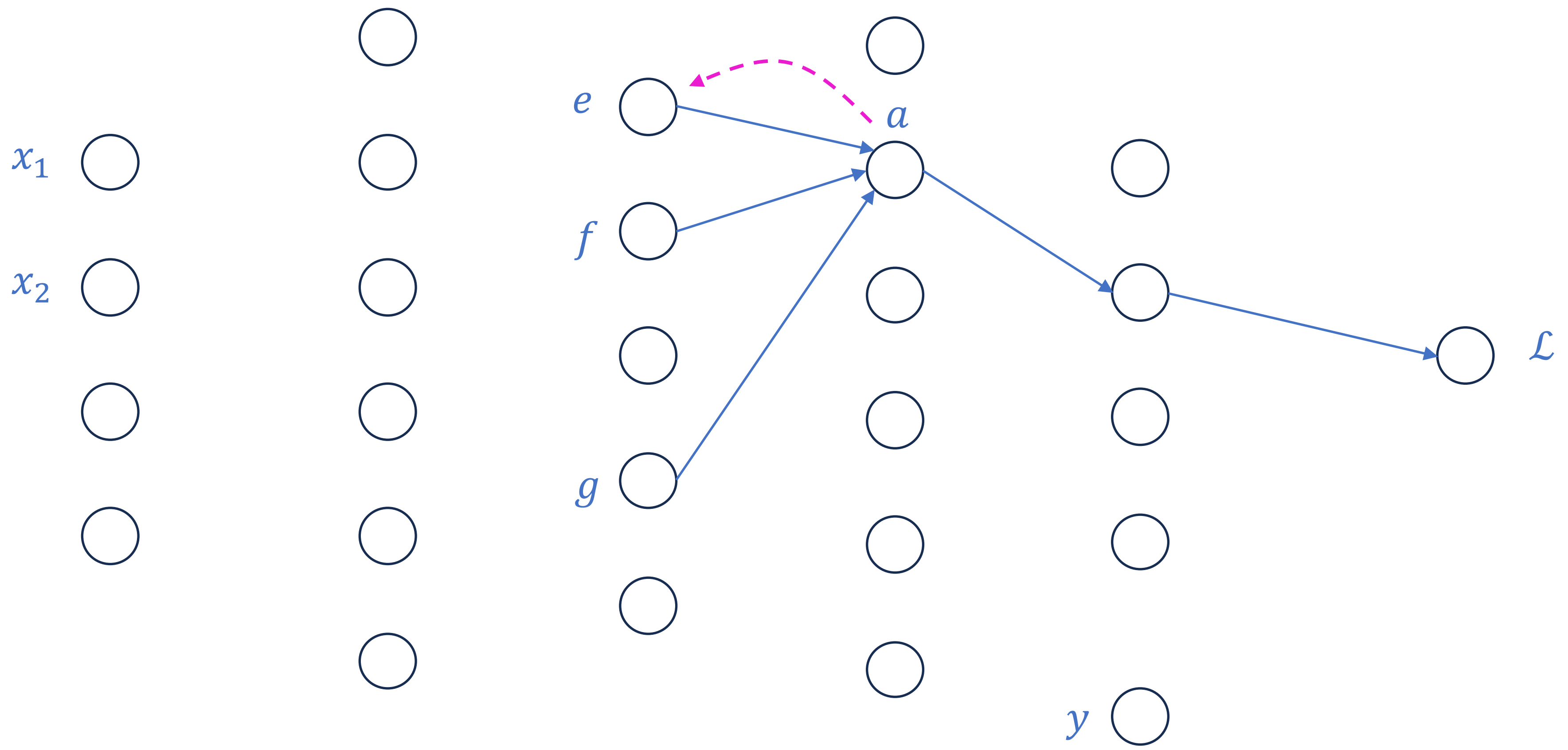


The Value class contains a “backward” function that updates the children’s gradient

Let’s build it by hand!

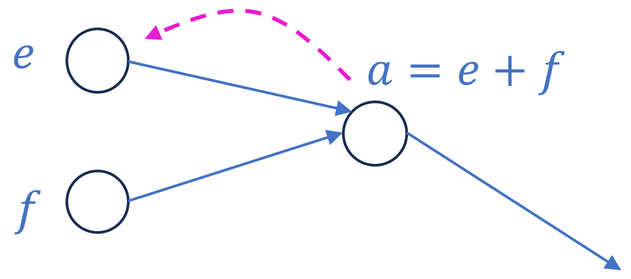
`e.grad +=` $\frac{\partial a}{\partial e}$ `* a.grad`

$$\frac{\partial \mathcal{L}}{\partial e} = \frac{\partial \mathcal{L}}{\partial a}\frac{\partial a}{\partial e} + \frac{\partial \mathcal{L}}{\partial b}\frac{\partial b}{\partial e} + \cdots$$



Let’s build it by hand!

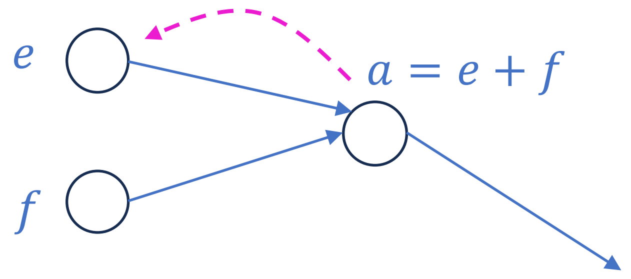
e.grad += $\frac{\partial a}{\partial e}$ * a.grad



```
class Value:
    def __init__(self, data):
        self.data = data
        self.grad = 0.0


    def __add__ (self, other):
        out = Value(self.data + other.data)
        return out
```

Let’s build it by hand!

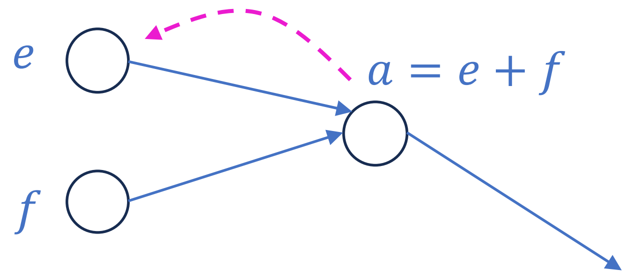
e.grad += $\frac{\partial a}{\partial e}$ * a.grad



```
class Value:
    def __init__(self, data):
        self.data = data
        self.grad = 0.0
        self.backward = lambda : None

    def __add__ (self, other):
        out = Value(self.data + other.data)
        def _backward():
            self.grad += ???
            other.grad += ???
        out.backward = _backward
        return out
```

Let’s build it by hand!

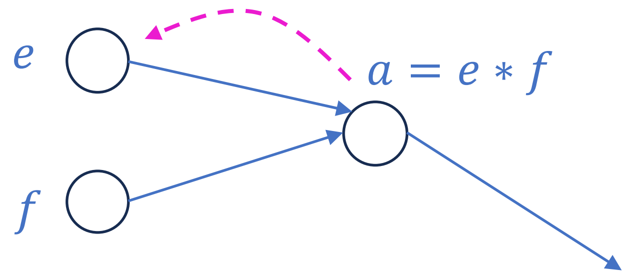
e.grad += $\frac{\partial a}{\partial e}$ * a.grad



```
class Value:
    def __init__(self, data):
        self.data = data
        self.grad = 0.0
        self.backward = lambda : None

    def __add__ (self, other):
        out = Value(self.data + other.data)
        def _backward():
            self.grad += 1.0 * out.grad
            other.grad += 1.0 * out.grad
        out.backward = _backward
        return out
```

Let’s build it by hand!

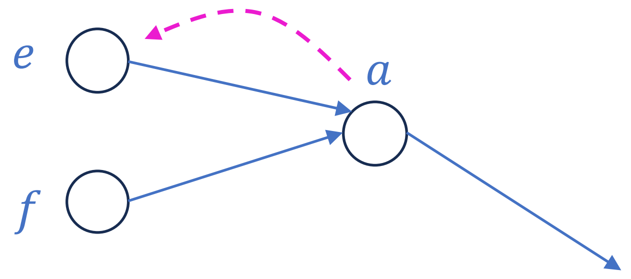
e.grad += $\frac{\partial a}{\partial e}$ * a.grad



```
class Value:
    def __init__(self, data):
        self.data = data
        self.grad = 0.0
        self.backward = lambda : None

    def __mul__ (self, other):
        out = Value(self.data * other.data)
        def _backward():
            self.grad += ??? * out.grad
            other.grad += ??? * out.grad
        out.backward = _backward
        return out
```

Let’s build it by hand!

e.grad += $\frac{\partial a}{\partial e}$ * a.grad



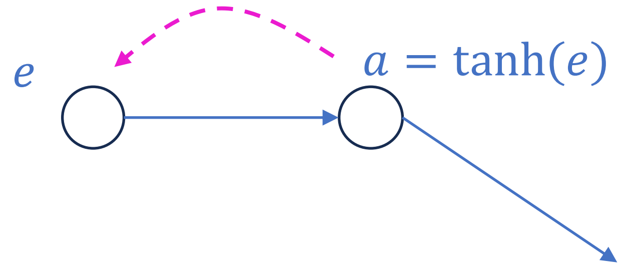
```
class Value:
    def __init__(self, data):
        self.data = data
        self.grad = 0.0
        self.backward = lambda : None

    def __mul__ (self, other):
        out = Value(self.data * other.data)
        def _backward():
            self.grad += other.data * out.grad
            other.grad += self.data * out.grad
        out.backward = _backward
        return out
```

$$a = e * f$$

$$\frac{\partial a}{\partial e} = f$$

Let's build it by hand!

e.grad += $\frac{\partial a}{\partial e}$ * a.grad



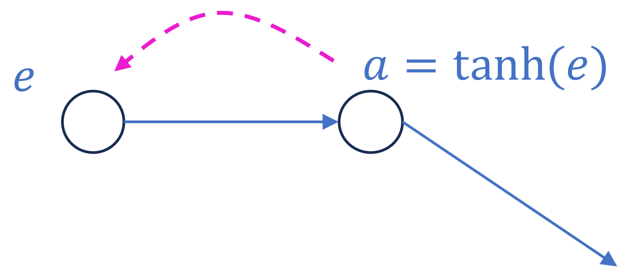
```
class Value:
    def __init__(self, data):
        self.data = data
        self.grad = 0.0
        self.backward = lambda : None

    def __tanh__ (self):
        out = Value(???)
        def _backward():
            self.grad += ??? * out.grad
        out.backward = _backward
        return out
```

$$a = \tanh(e)$$

$$\frac{\partial a}{\partial e} = ???$$

Let’s build it by hand!

e.grad += $\frac{\partial a}{\partial e}$ * a.grad



```
class Value:
    def __init__(self, data):
        self.data = data
        self.grad = 0.0
        self.backward = lambda : None

    def __tanh__ (self):
       t = (math.exp(2*x) - 1)/(math.exp(2*x) + 1)
        out = Value(t, (self, ), ‘tanh’)
        def _backward():
            self.grad += (1.0 - t**2) * out.grad
        out.backward = _backward
        return out
```

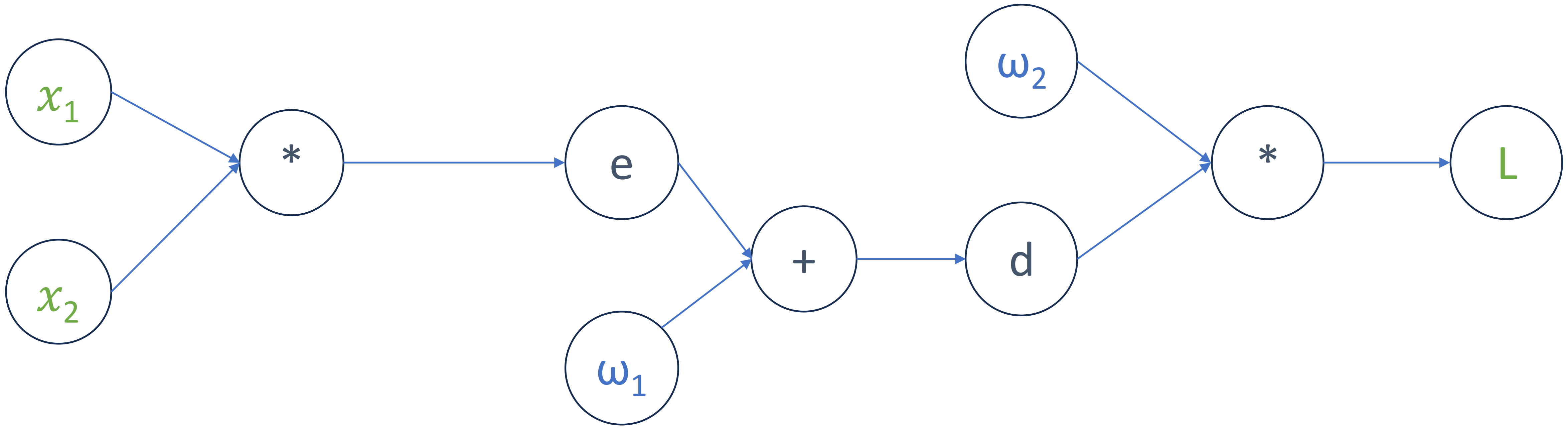
$$a = \tanh(e)$$

$$\frac{\partial a}{\partial e} = 1 - a^2$$

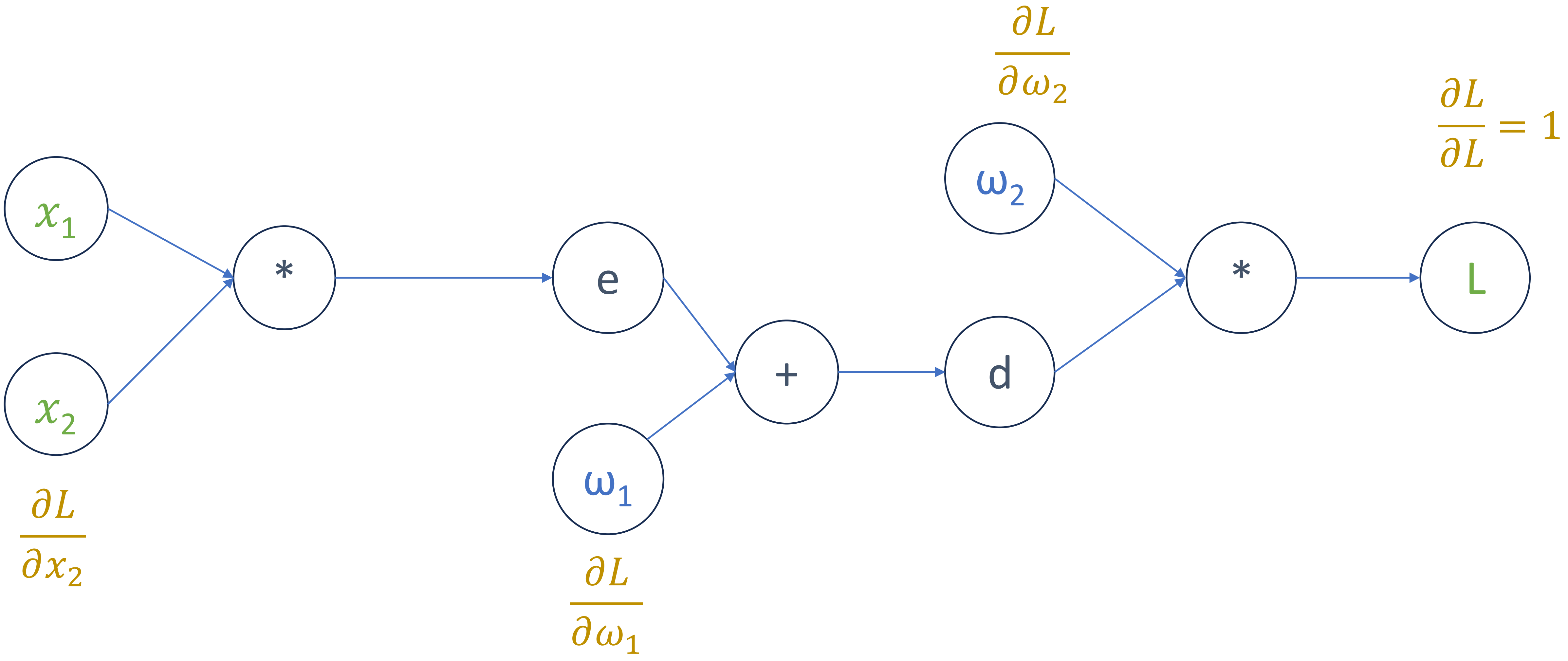
The lol() function

$$\begin{tikzcd}
 & & & & f \arrow[ddr] & & \\
a \arrow[dr] & & & & & & \\
 & * \arrow[r] & e \arrow[dr] & & & * \arrow[r] & L \\
b \arrow[ur] & & & + \arrow[r] & d \arrow[ur] & & \\
 & & c \arrow[ur] & & & &
\end{tikzcd}$$

The lol() function



The lol() function

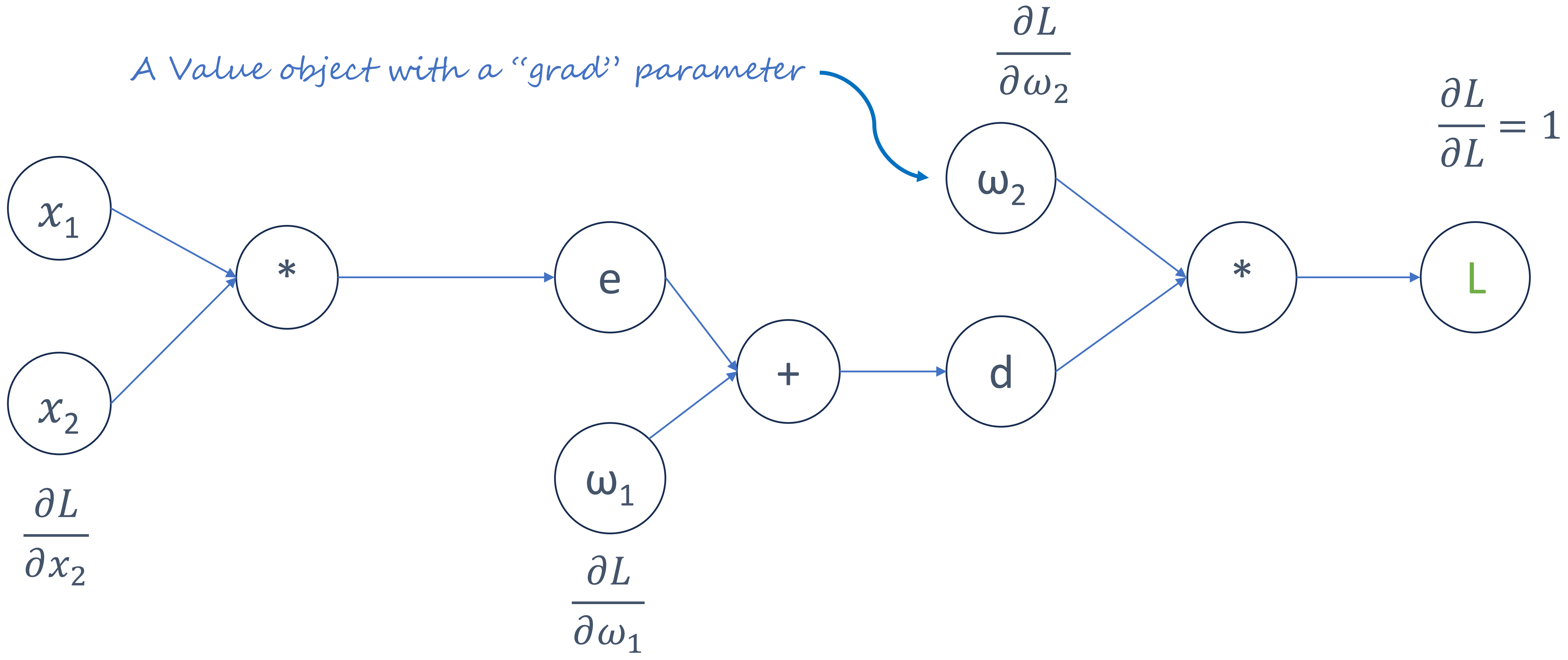

x_1
x_2
$*$
e
ω_1
$+$
d
ω_2
$*$
L
$\frac{\partial L}{\partial \omega_2}$
$\frac{\partial L}{\partial L} = 1$
$\frac{\partial L}{\partial x_2}$
$\frac{\partial L}{\partial \omega_1}$

Gradient descent for the lol() function

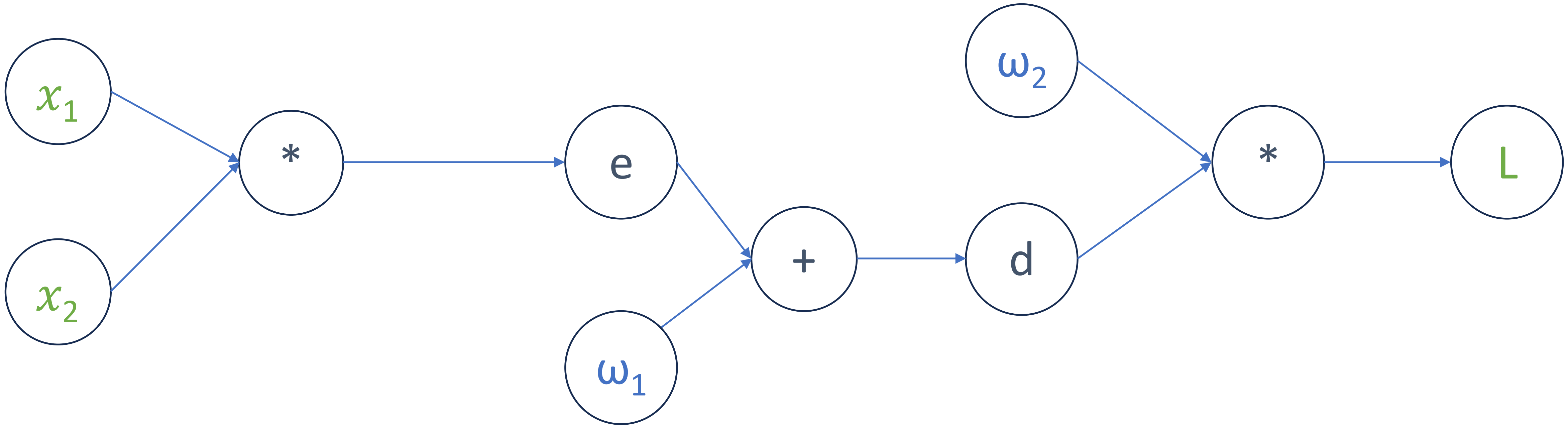
$$\omega_1 = \omega_1 - \eta \frac{\partial L}{\partial \omega_1}$$

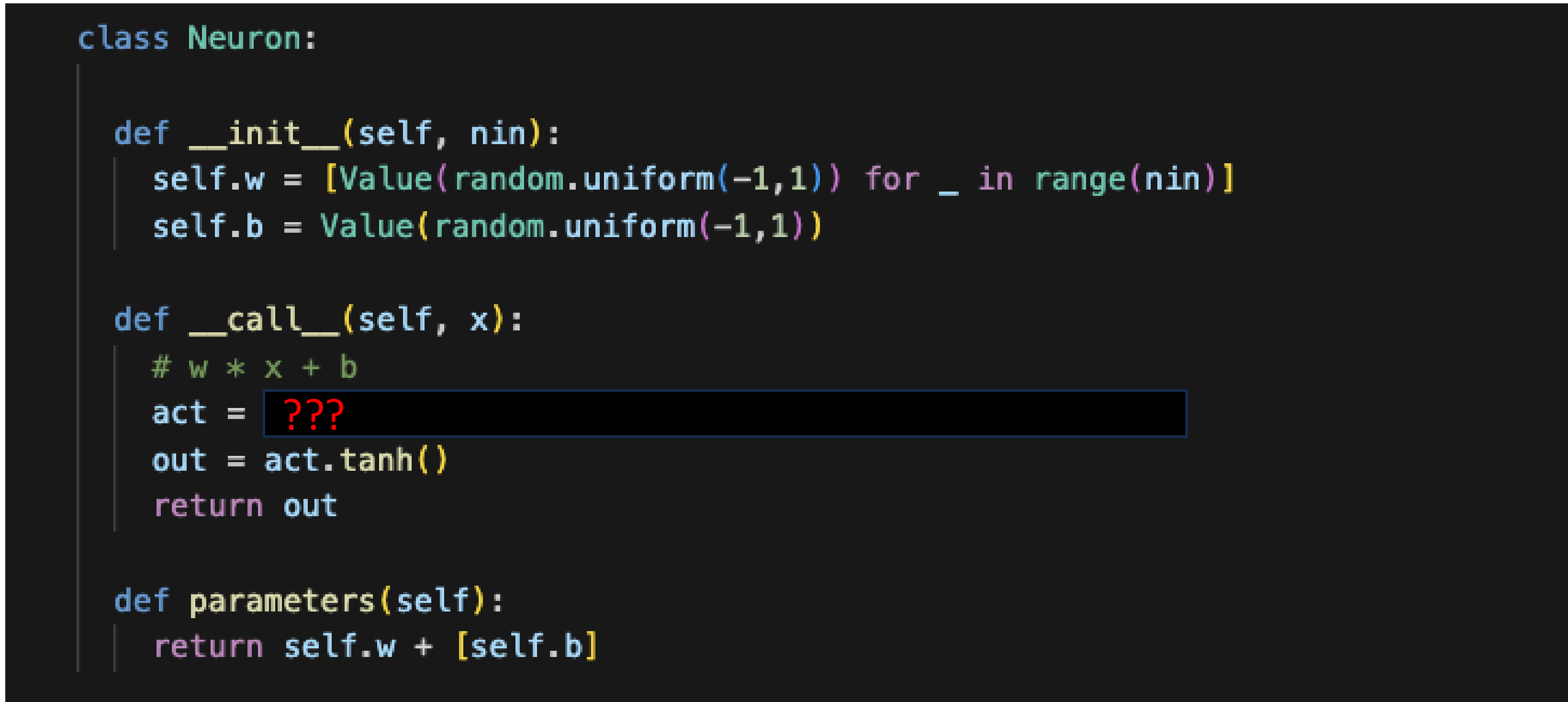
$$\omega_2 = \omega_2 - \eta \frac{\partial L}{\partial \omega_2}$$

The lol() function



A Value object with a "grad" parameter
$\frac{\partial L}{\partial \omega_2}$
$\frac{\partial L}{\partial L} = 1$
x_1
x_2
*
e
+
ω_1
ω_2
d
*
L
$\frac{\partial L}{\partial x_2}$
$\frac{\partial L}{\partial \omega_1}$




```
class Neuron:

  def __init__(self, nin):
    self.w = [Value(random.uniform(-1,1)) for _ in range(nin)]
    self.b = Value(random.uniform(-1,1))

  def __call__(self, x):
    # w * x + b
    act = ???
    out = act.tanh()
    return out

  def parameters(self):
    return self.w + [self.b]
```

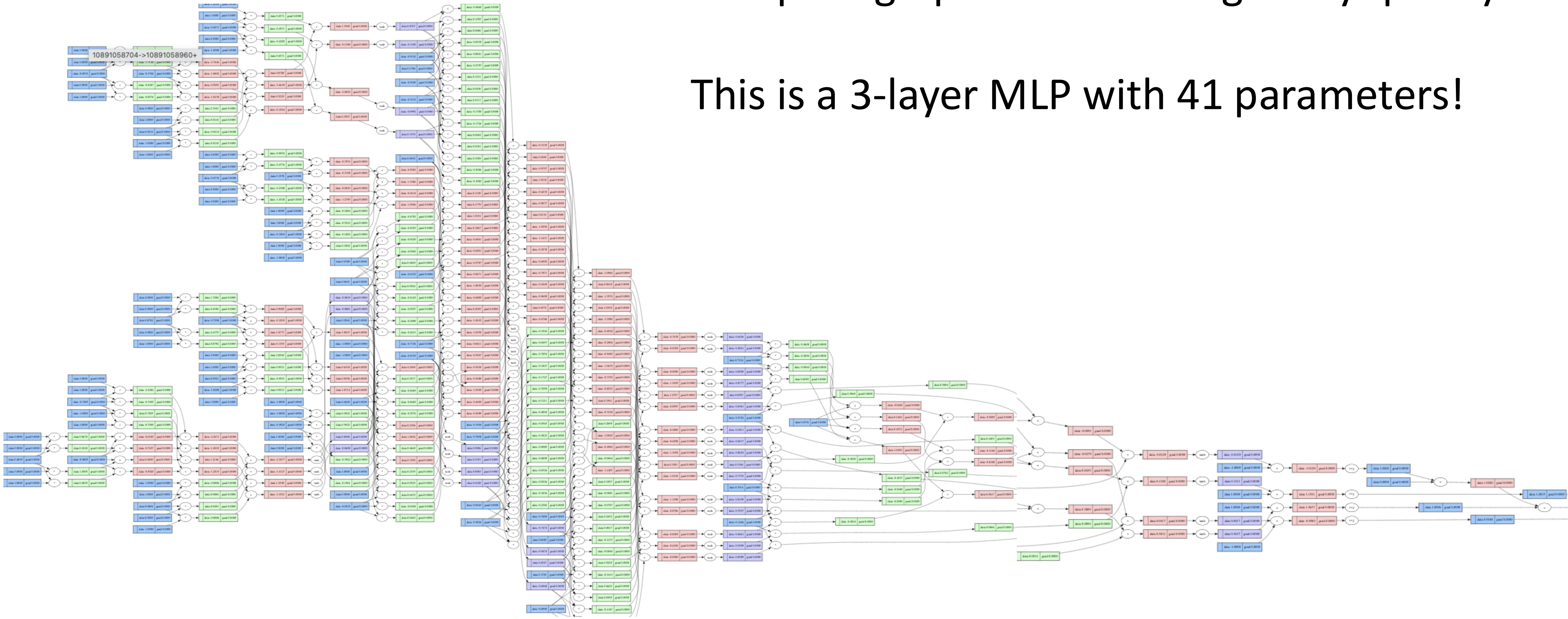
Mean square error loss

$$\ell = \sum (y - \hat{y})^2$$

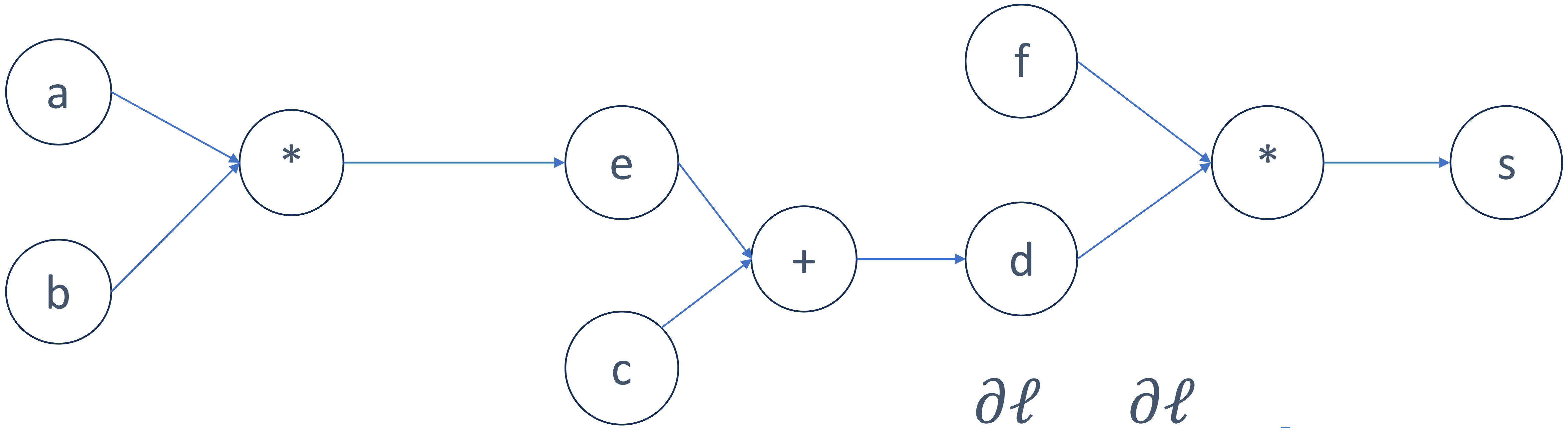
```
n = MLP(3, [4, 4, 1])

ypred = [n(x) for x in xs]

loss = sum([(a-b)**2 for (a,b) in zip(ypred, ys)])
```

Compute graphs become huge very quickly

This is a 3-layer MLP with 41 parameters!



The forward pass is critical to update the values in the network



$$\frac{\partial \ell}{\partial f} = \frac{\partial \ell}{\partial s} * d$$

Stochastic gradient descent

Iterate over batches of samples instead of the whole dataset at once

1. Scales to large datasets (that don’t fit in memory)
2. Adds randomness to the process
3. Adds implicit regularization

Conclusion

Neural networks are compute graphs.

Gradient descent minimizes a loss function over the network’s parameters.

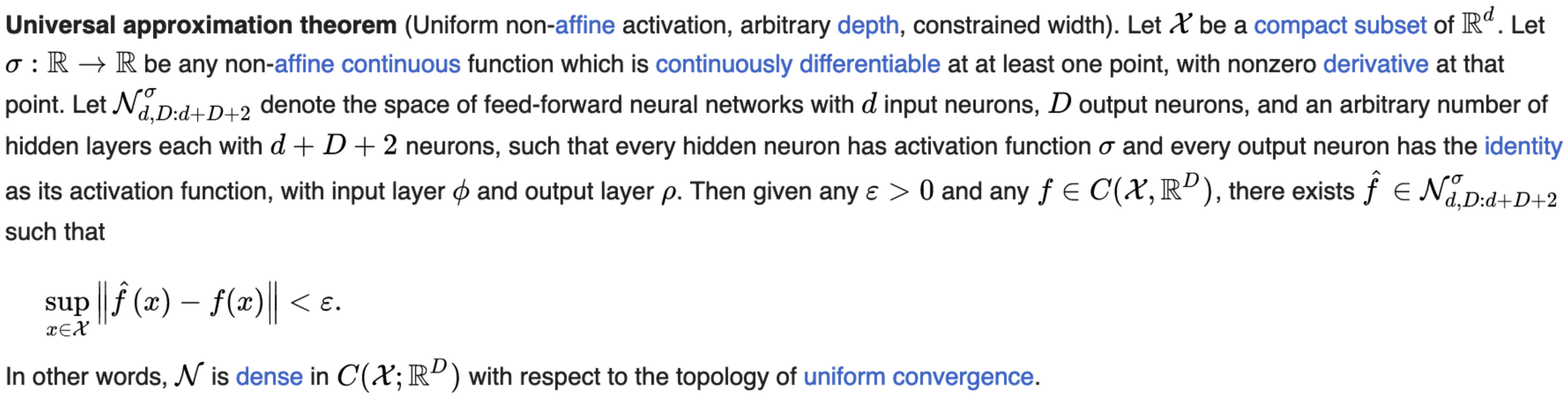
Back-propagation allows efficient learning (tuning of the network).

Let’s practice with a simple Multi-Layer Perception (MLP).

Universal approximation theorem

For any continuous function, there exists a shallow network that can approximate this function to any specified precision.

Universal approximation theorem



Universal approximation theorem (Uniform non-affine activation, arbitrary depth, constrained width). Let $\mathcal{X}$ be a compact subset of $\mathbb{R}^d$. Let $\sigma : \mathbb{R} \to \mathbb{R}$ be any non-affine continuous function which is continuously differentiable at at least one point, with nonzero derivative at that point. Let $\mathcal{N}^{\sigma}_{d,D:d+D+2}$ denote the space of feed-forward neural networks with d input neurons, D output neurons, and an arbitrary number of hidden layers each with $d + D + 2$ neurons, such that every hidden neuron has activation function σ and every output neuron has the identity as its activation function, with input layer ϕ and output layer ρ. Then given any $\varepsilon > 0$ and any $f \in C(\mathcal{X}, \mathbb{R}^D)$, there exists $\hat{f} \in \mathcal{N}^{\sigma}_{d,D:d+D+2}$ such that

$$\sup_{x \in \mathcal{X}} \left\| \hat{f}(x) - f(x) \right\| < \varepsilon.$$

In other words, $\mathcal{N}$ is dense in $C(\mathcal{X}; \mathbb{R}^D)$ with respect to the topology of uniform convergence.

No free-lunch theorem

Every learning algorithm is as good as any other when averaged over all sets of problems.

You can’t just learn « purely from data » without bias.

Wolpert, D. H.; Macready, W. G. (1997). "No Free Lunch Theorems for Optimization". *IEEE Transactions on Evolutionary Computation*. **1**: 67–82.