Deep Learning Bootcamp: Training of Neural Networks

Technische Hochschule Ingolstadt

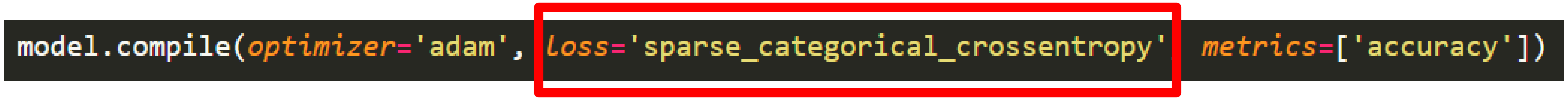
KI-basierte Optimierung in der Automobilproduktion

Introduction

Training of Neural Networks

- **Recap**: Our model should classify (new) data correctly
- We want *accuracy* as a metric to evaluate the performance of our model

But:



```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

→ During training, we don‘t maximize accuracy, we minimize a loss function!

Loss functions

What is a loss function?

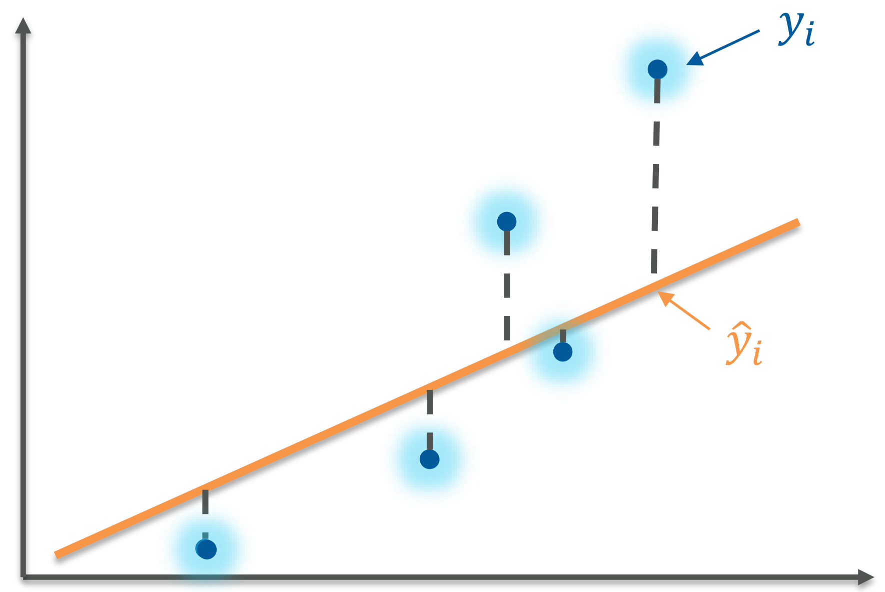
- It quantifies *how **bad*** our model's predictions are

→We want to minimize the loss of our model!

- **Examples**:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i|$$

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

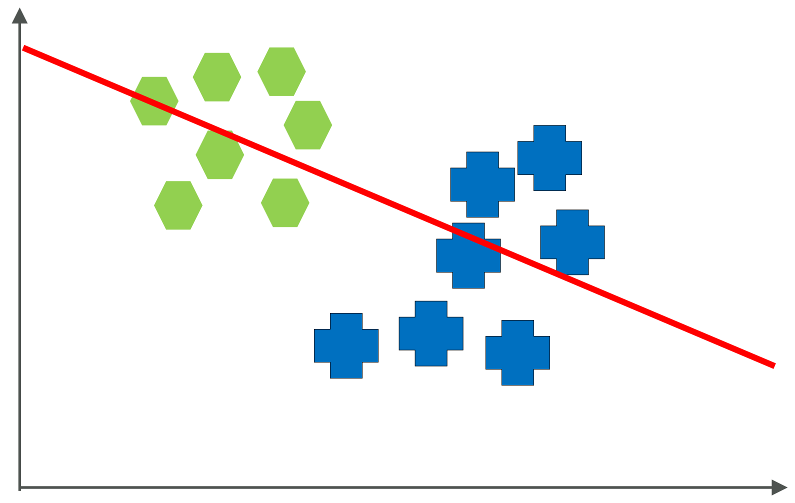


How can we optimize the parameters?

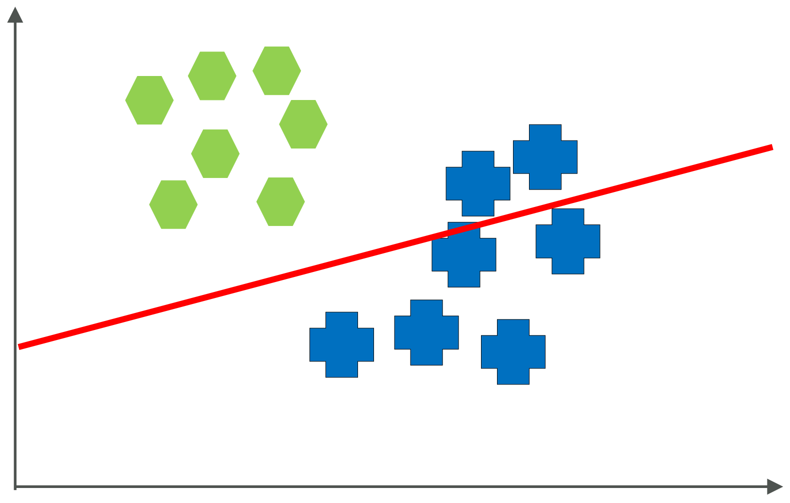
Loss functions

w and b

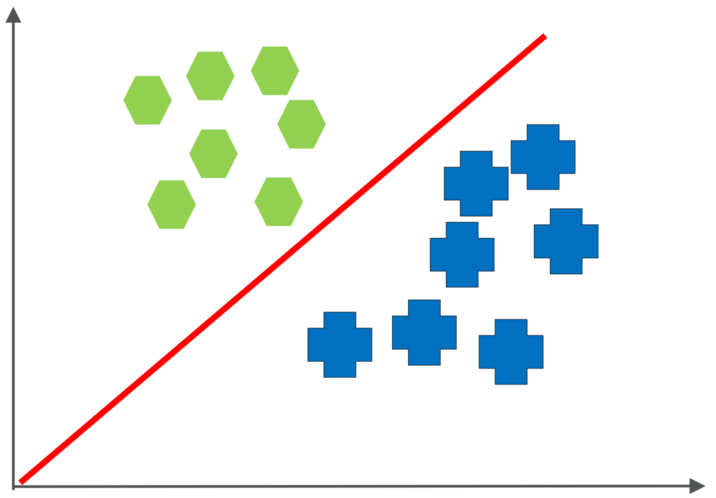
What is the relationship between model parameters and the loss value?
→Example: We can control slope w and intercept b of our linear decision function ($y = wx + b$)



Negative w, positive b → High loss



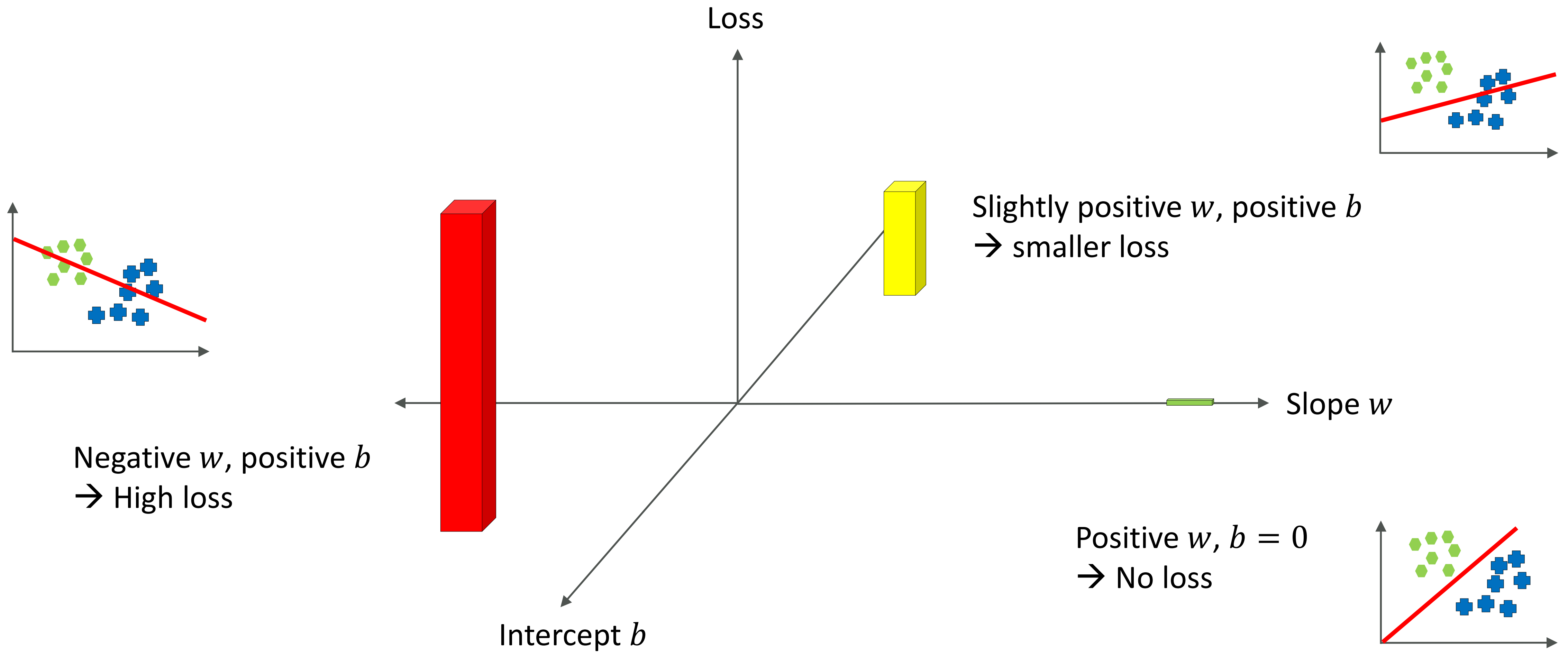
Slightly positive w, positive b
→ smaller loss



Positive w, $b = 0$
→ No loss

Loss functions

If we plot the previous example, we can approximate a landscape:

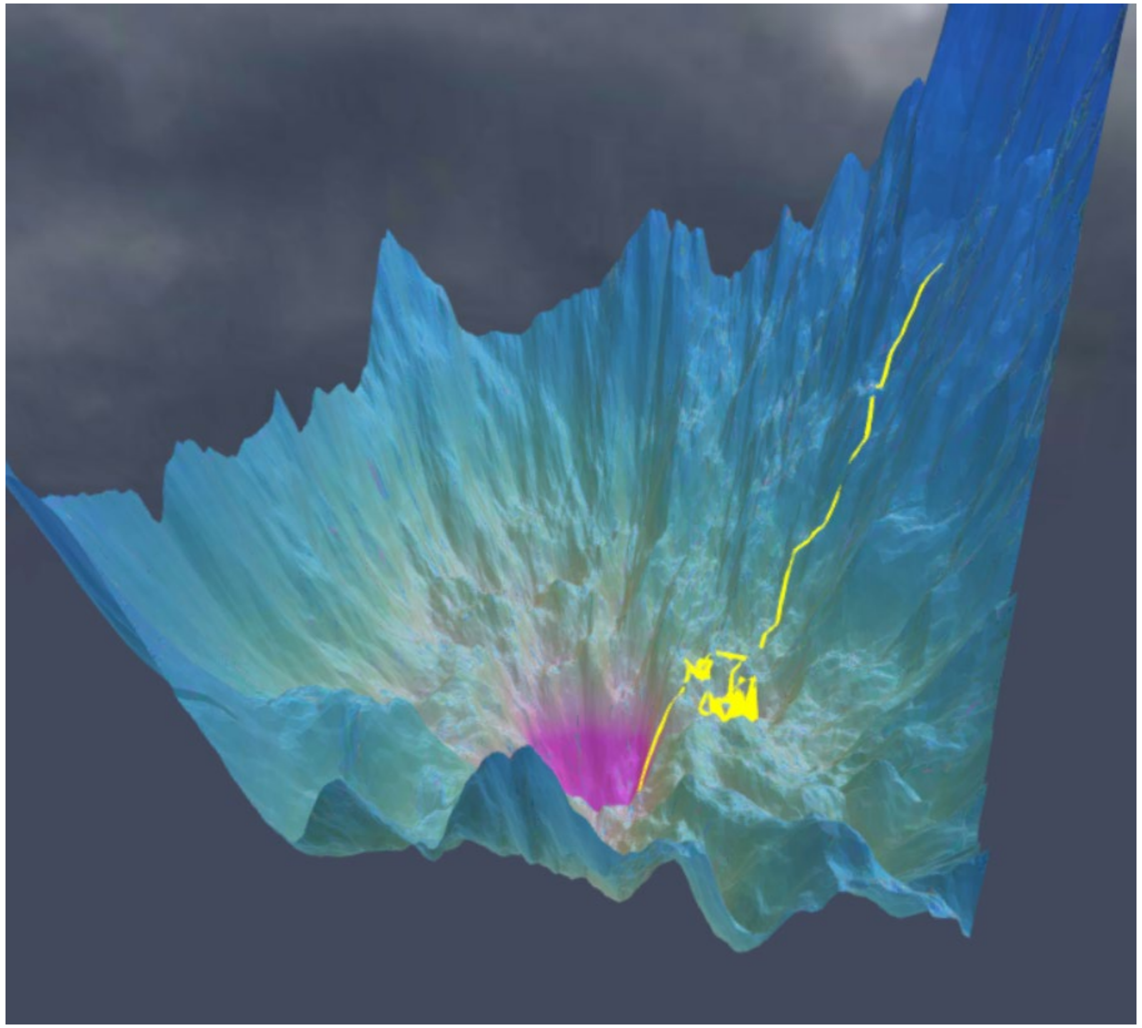


Intuition: Optimization in a Loss Landscape

- We want to choose the parameters of our model such that the loss value is minimal

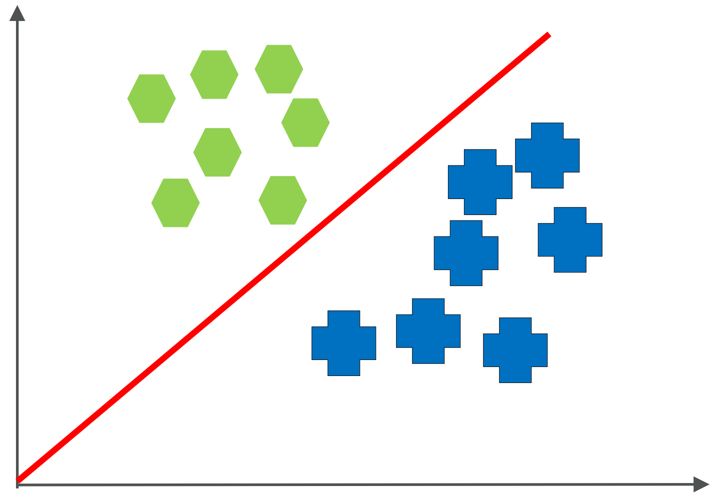
- **Intuition**: We want to roll down a mountain

- An optimization algorithm can‘t see the „landscape“, it must perceive the slope through the gradient

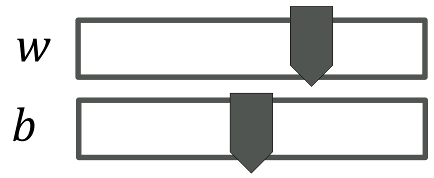


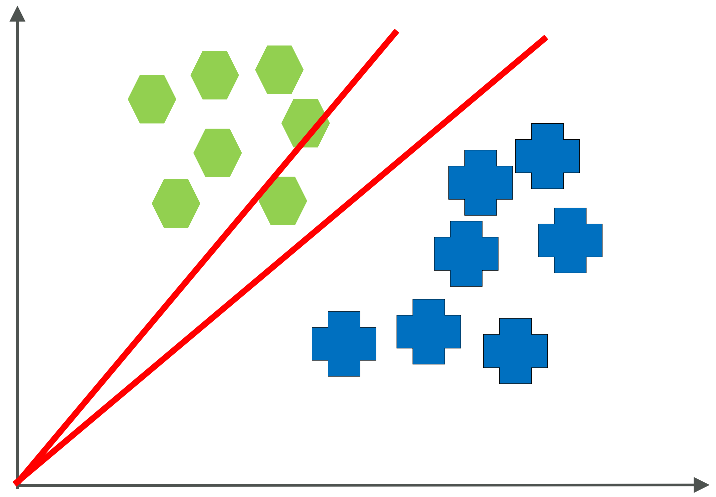
Partial Derivatives (1/2)

We want to know how much the change of a single parameter influences the function value

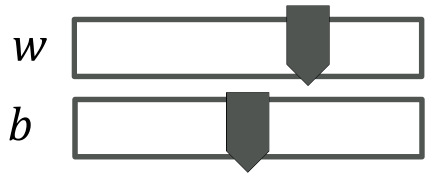


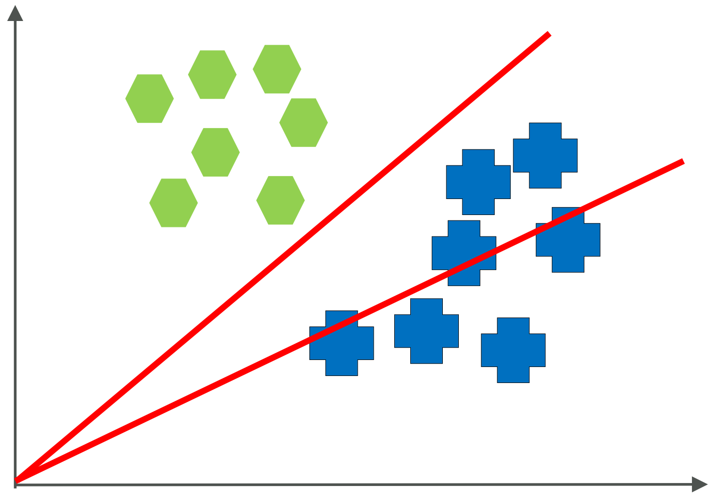
Positive w, $b = 0$



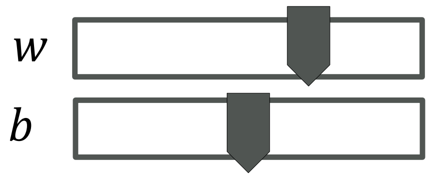


Larger w, $b = 0$



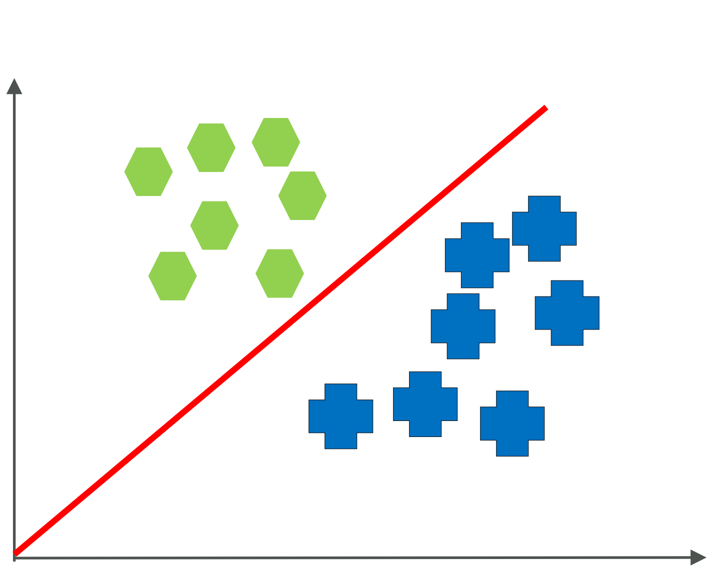


Smaller w, $b = 0$

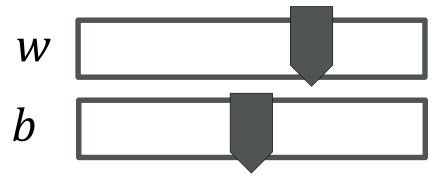


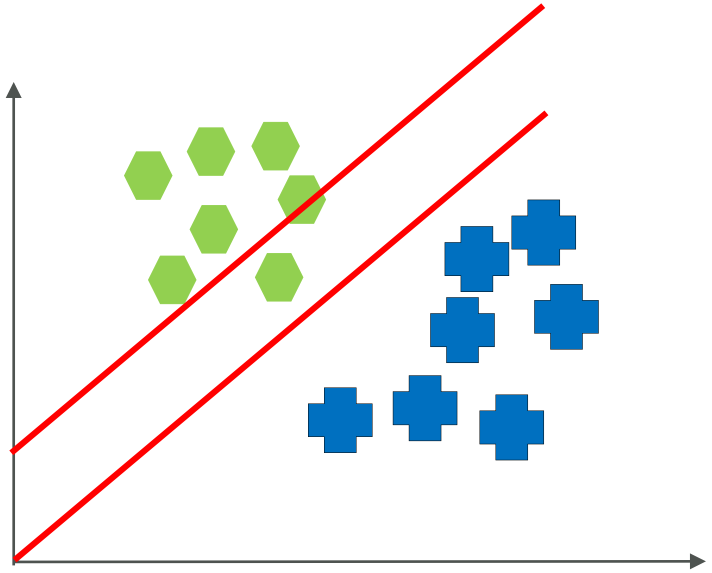
Partial Derivatives (2/2)

We want to know how much the change of a single parameter influences the function value

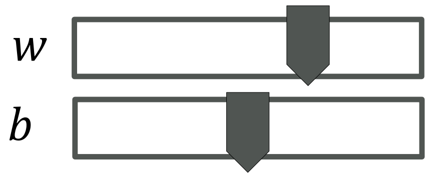


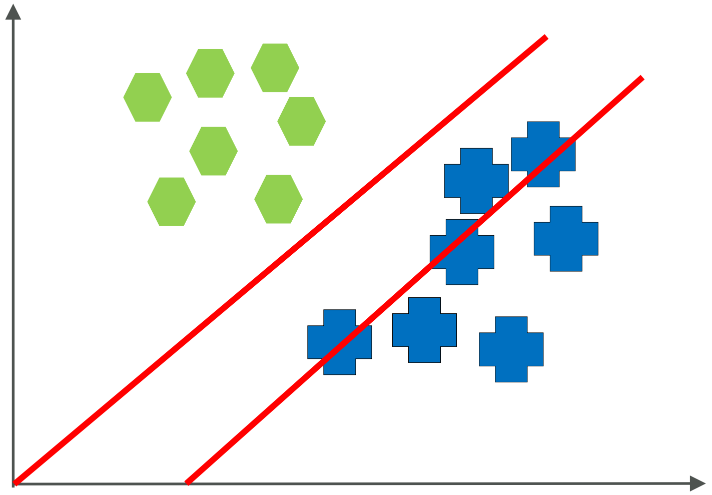
w fixed, $b = 0$



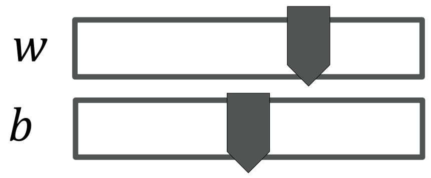


w fixed, Larger b





w fixed, Smaller b

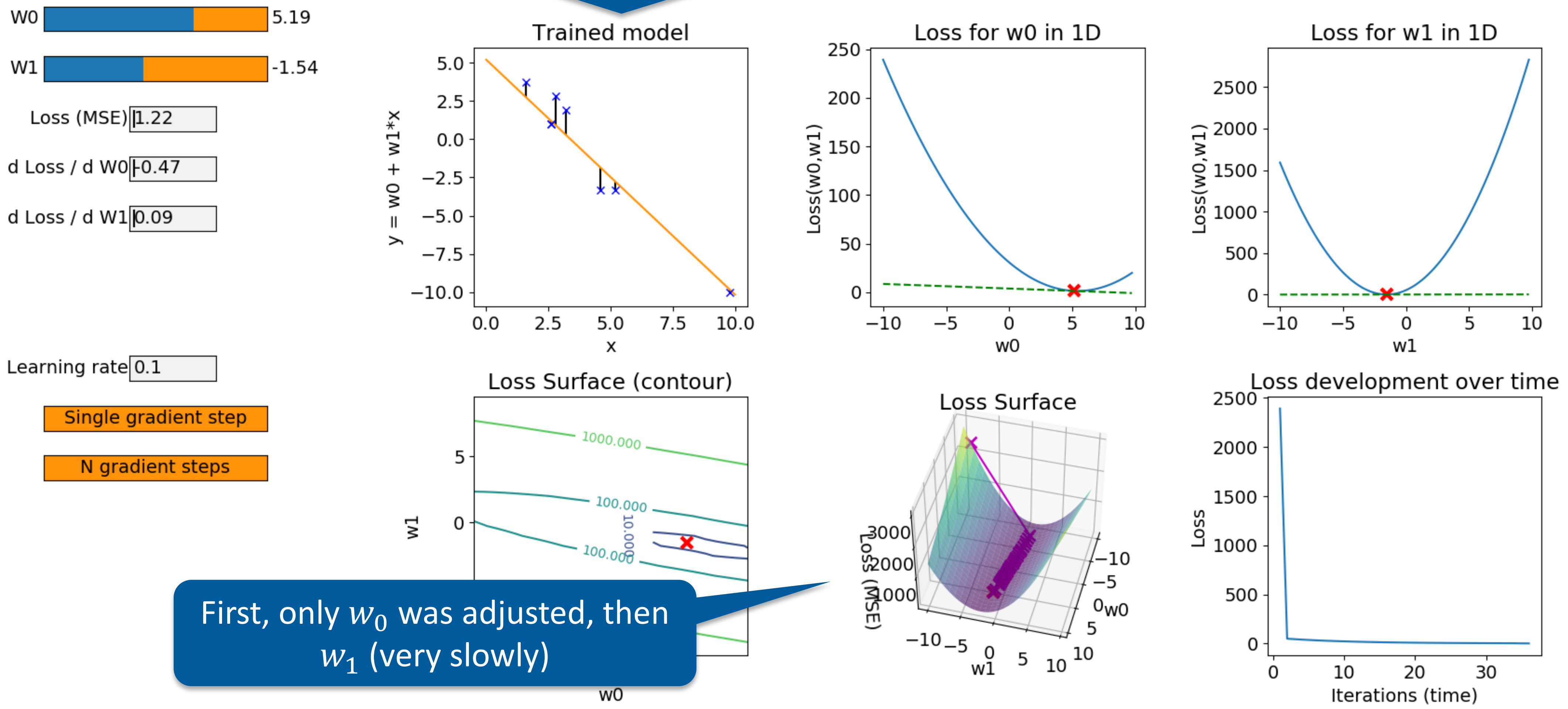


A visualization of the training process

After many more iterations …

w_1 is finally < 0 but that took a while!

very small gradients

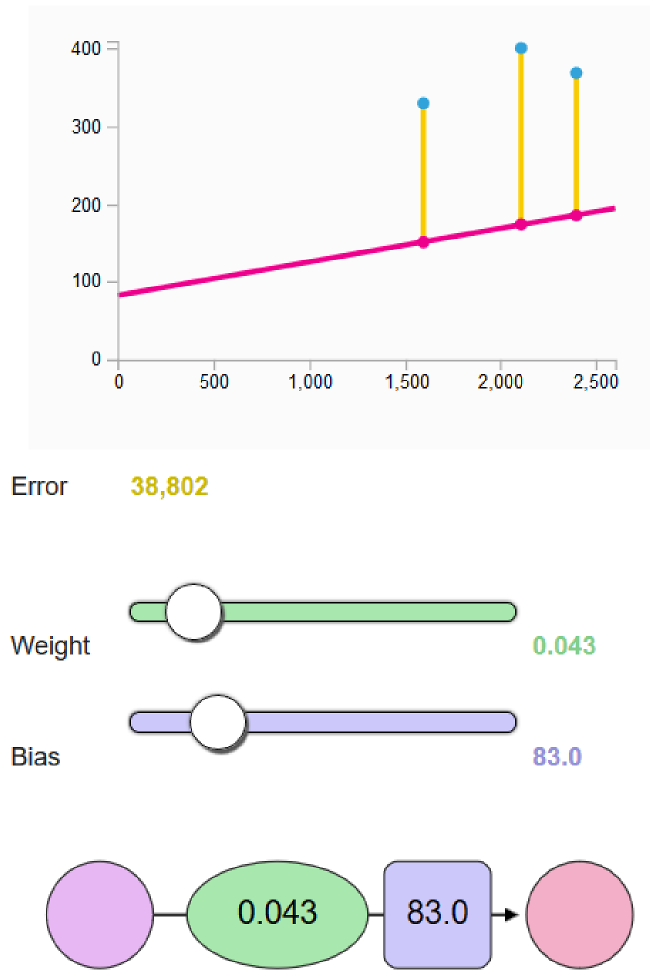


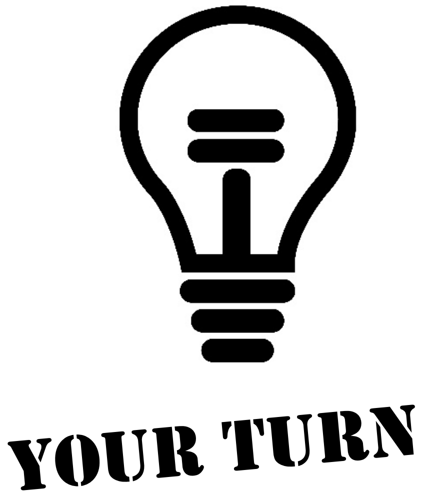
First, only w_0 was adjusted, then w_1 (very slowly)

Let‘s try it out!

Try to find an optimal solution by adjusting the parameters!

https://jalammar.github.io/visual-interactive-guide-basics-neural-networks/#train-your-dragon





Partial Derivatives and the Gradient

Differentiate a function with multiple variables for each in turn.
Collect results in a vector („the gradient“).

$$L(w_1, w_2) = 3w_1^2 + w_1 w_2 - 1$$

$$\left.\begin{aligned} \frac{\partial L}{\partial w_1} &= 6w_1 + w_2 \\[2em] \frac{\partial L}{\partial w_2} &= w_1 \end{aligned}\right\}$$

partial derivatives
(“Partielle Ableitungen”)

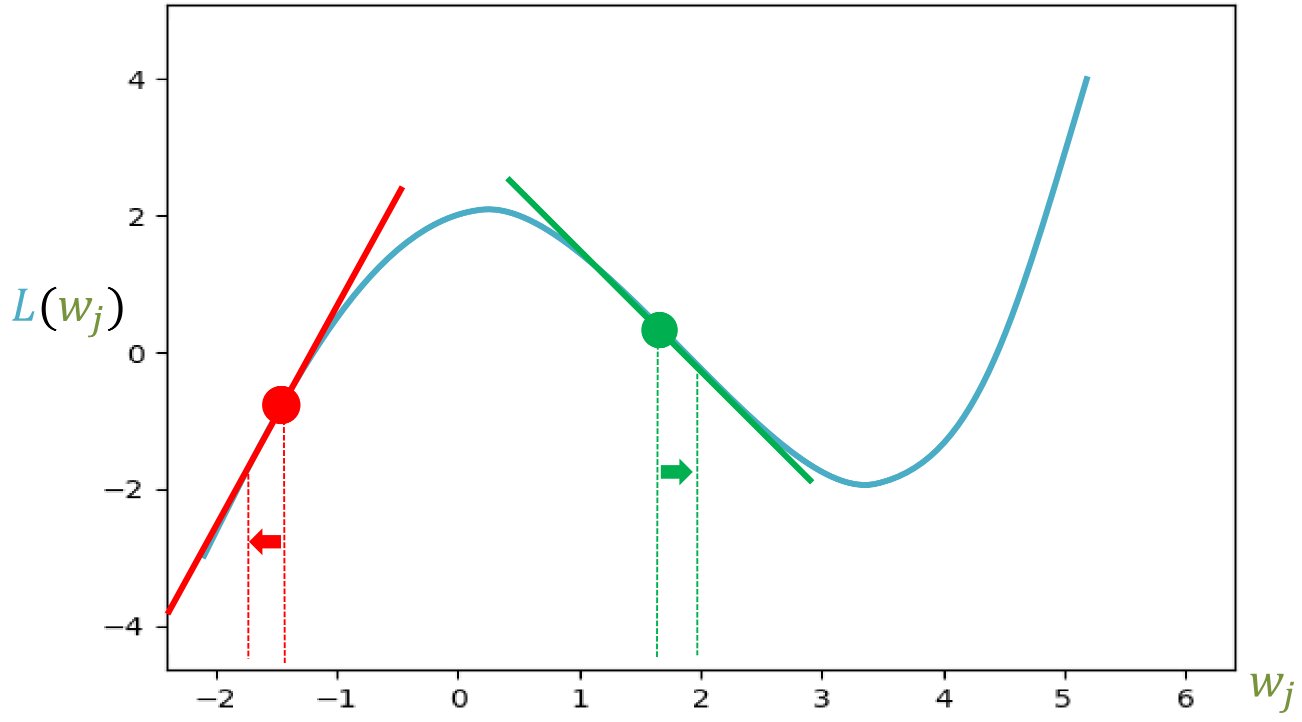
$$\Rightarrow \nabla_{\boldsymbol{x}} = \begin{bmatrix} 6w_1 + w_2 \\ w_1 \end{bmatrix}$$

gradient
(“Gradient”)

Idea: Gradient Descent

Determine the *rate of change* of our loss function L with respect to a parameter's change

1. „Increasing w_j a little bit will *increase* the loss a little bit" → so decrease w_j!
2. „Increasing w_j a little bit will *decrease* the loss a little bit" → so increase w_j!



Formally:

find $\frac{\partial L}{\partial w_j}$ and update $w_j \leftarrow w_j - \alpha \frac{\partial L}{\partial w_j}$

α is called the **„learning rate"**

Loss functions

Why don‘t we maximize accuracy directly?

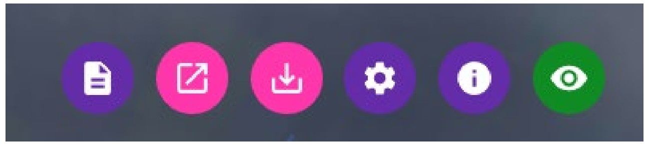
- We use *gradient-based* optimization techniques

→The metric that should me optimized must be differentiable

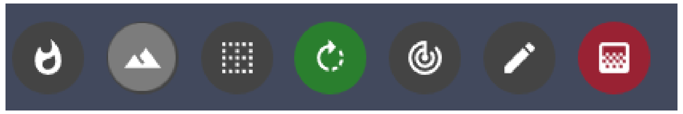
- Accuracy is *not differentiable* (sudden jumps)

Self-Study Time

Losslandscape Explorer:
https://losslandscape.com/explorer



Use these icons to get informations about what you see, change the landscape or take screenshots!



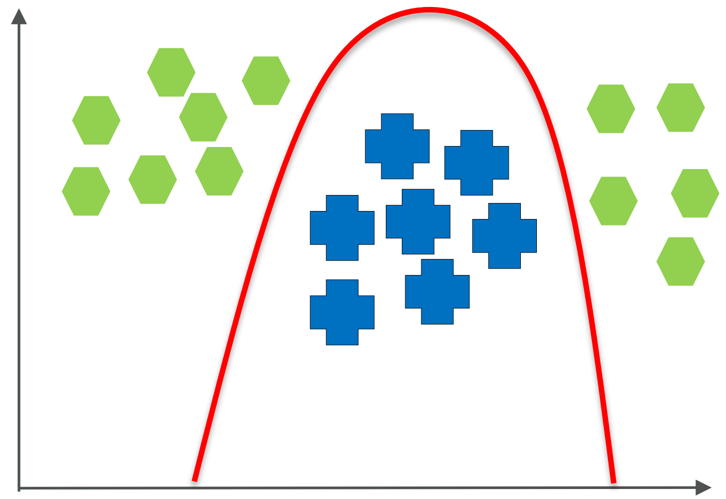
Use these icons to experiment with the plotted landscape

Tasks:

- ➢ Play around with the different visualizations on the site
- ➢ Take screenshots of landscapes you like

How can we calculate the gradient of a Neural network?

More complicated data sets



Larger w, $b = 0$

→ We need *non-linear* decision functions

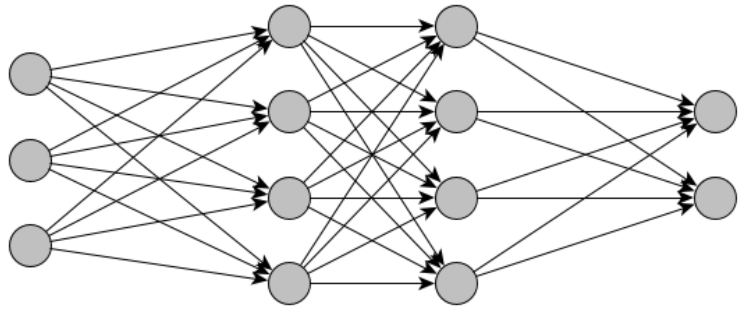
Deriving Neural Networks

- Essentially, Neural Networks are *huge functions* with thousands of parameters

- Neural Networks are a chain of simpler functions: $f^{(n)}(f^{(n-1)}\left(f^{(\dots)}(f^{0}(x_1, \dots, x_m))\right))$ → Neural Network with n layers and m variables

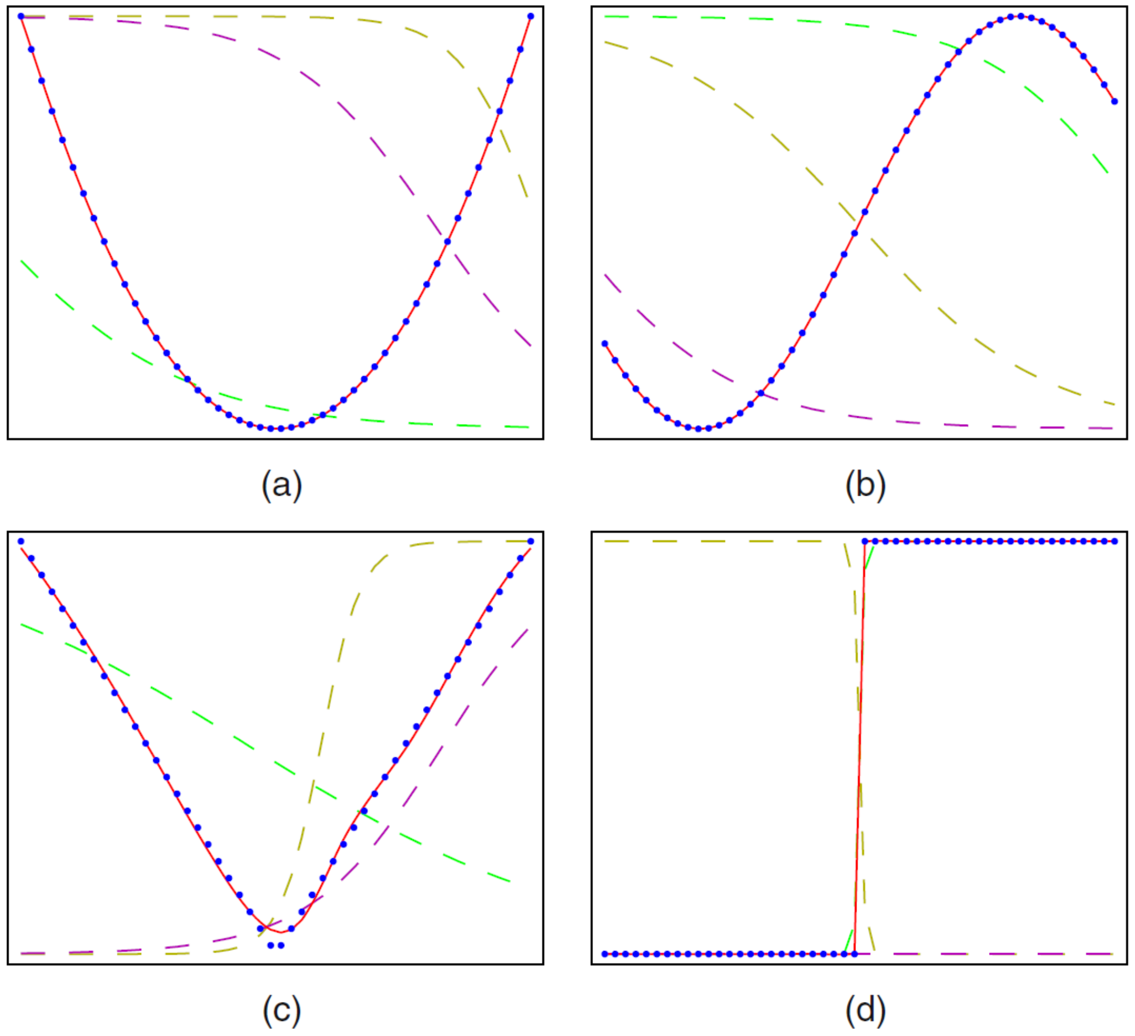
- Our graphical representation as network makes it easier to understand
- Can we use this simpler representation for deriving the network?

→ YES! It will be more detailed, though



What neural networks are capable of

Figure 5.3 Illustration of the capability of a multilayer perceptron to approximate four different functions comprising (a) $f(x) = x^2$, (b) $f(x) = \sin(x)$, (c), $f(x) = |x|$, and (d) $f(x) = H(x)$ where $H(x)$ is the Heaviside step function. In each case, $N = 50$ data points, shown as blue dots, have been sampled uniformly in x over the interval $(-1, 1)$ and the corresponding values of $f(x)$ evaluated. These data points are then used to train a two-layer network having 3 hidden units with '$\tanh$' activation functions and linear output units. The resulting network functions are shown by the red curves, and the outputs of the three hidden units are shown by the three dashed curves.

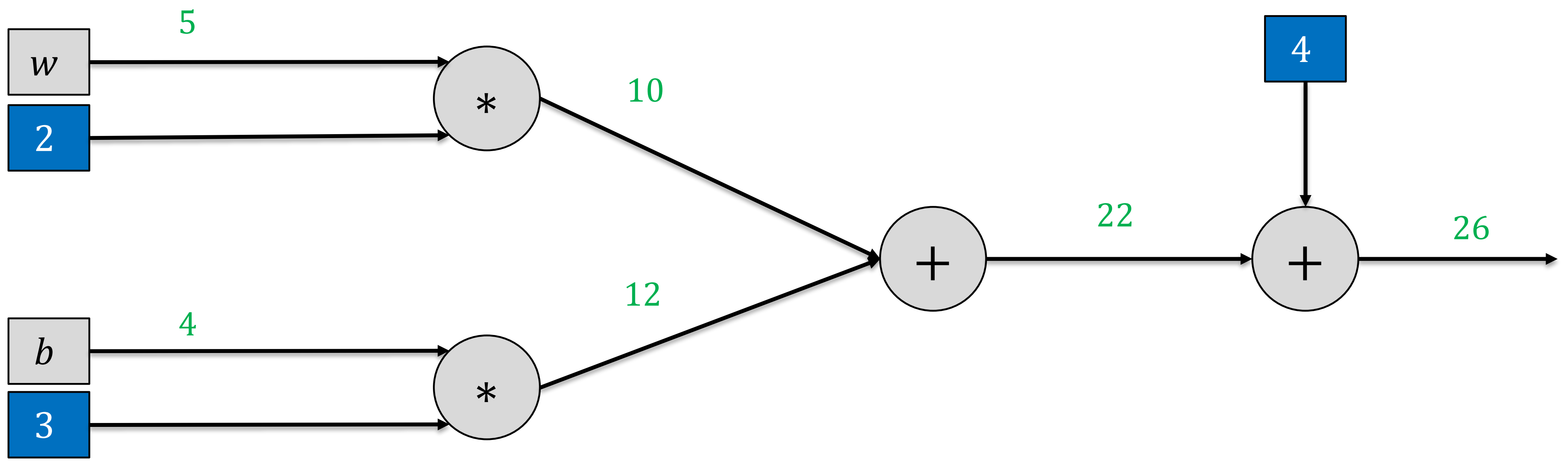


[Bishop, PRML 2006]

Computational Graphs

Example: $L(w,b) = 2w + 3b + 4$

Forward pass



Training Algorithm For A Neural Network – High-Level

Initialization: set all parameters (weights, biases) to small values

for every epoch, for every instance:

1. Forward pass:
   - Determine the current prediction $y = f(\boldsymbol{x})$ for a training instance $\langle \boldsymbol{x}, t(\boldsymbol{x}) \rangle$
   - Evaluate the loss $\mathrm{L}\big(y, t(\boldsymbol{x})\big)$

2. Backward pass:
   - Determine the partial derivatives $\frac{\partial L}{\partial W_{i,j}}$ for every weight and bias :

3. Gradient update:
   - Update weights and biases (take a gradient step)

How do we get gradients?

(1) manually working out derivatives and coding them;

- error-prone, does not scale well for large networks

(2) numerical differentiation using finite difference approximations;

- imprecise and computationally expensive! Requires evaluating $f(\boldsymbol{x}+\varepsilon)-f(\boldsymbol{x})$ for every parameter $\boldsymbol{w}_{\mathrm{j}}$!

(3) symbolic differentiation using expression manipulation in computer algebra

- complex and cryptic expressions; require closed-form equations

(4) automatic differentiation (autodiff), also called algorithmic differentiation

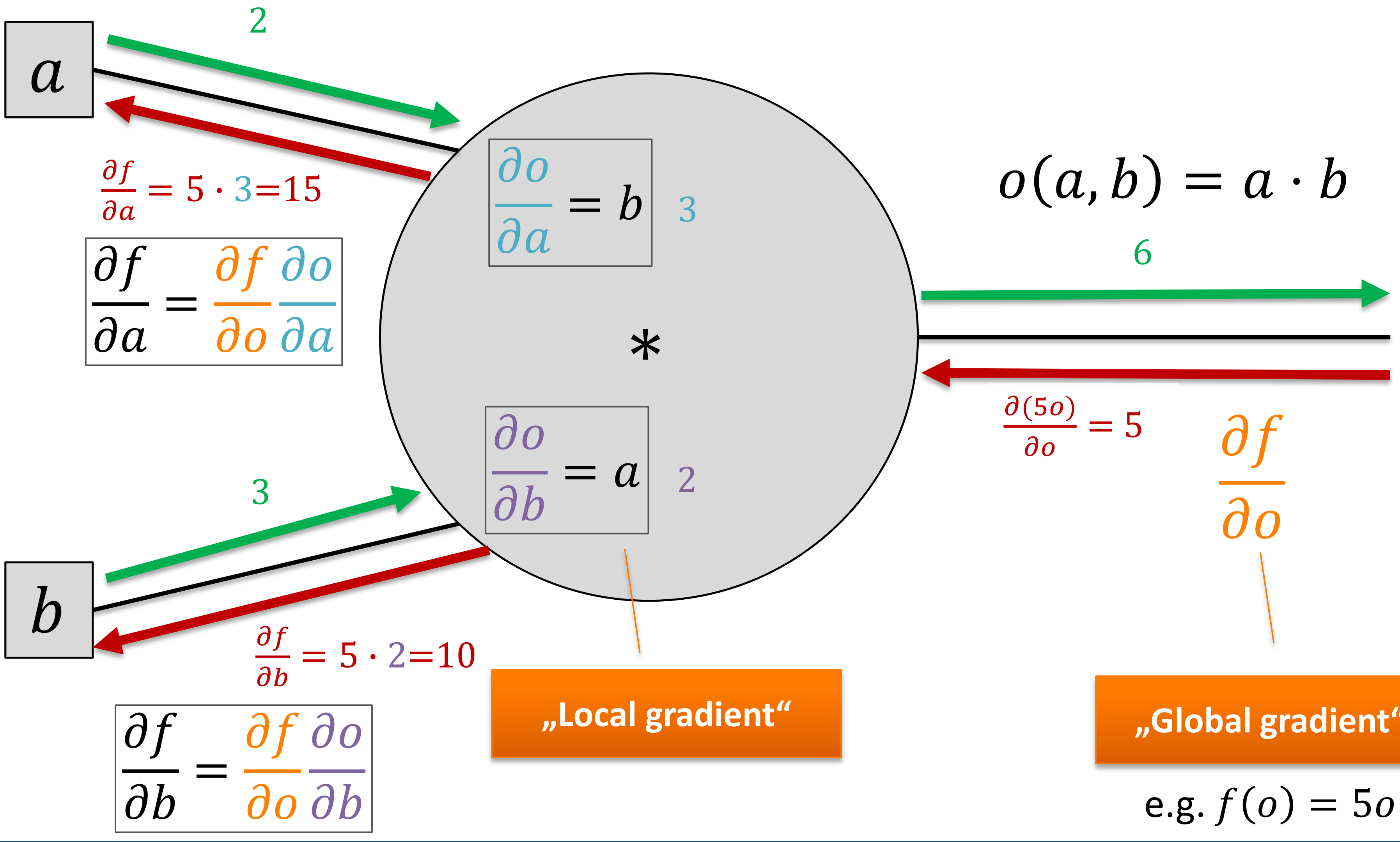
[Automatic Differentiation in Machine Learning: A Survey; Baydin, Pearlmutter, Radul, and Siskind; 2018]

Computational Graphs

Multiplication ~ Gradient switcher

Forward pass

Backward pass

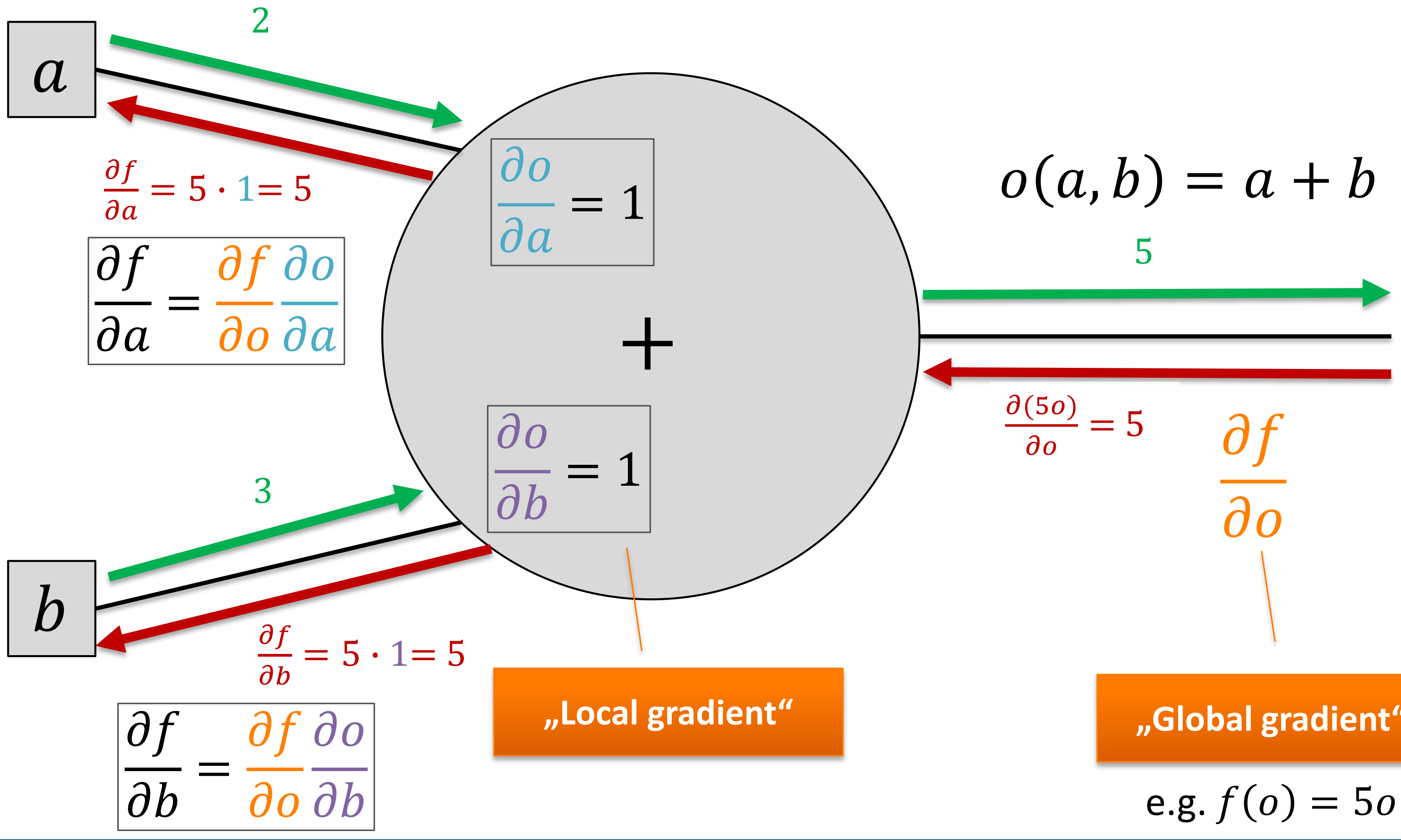

a
b
2
3
6
$*$
$\frac{\partial o}{\partial a} = b$
3
$\frac{\partial o}{\partial b} = a$
2
$o(a,b) = a \cdot b$
$\frac{\partial f}{\partial a} = 5 \cdot 3 = 15$
$\frac{\partial f}{\partial a} = \frac{\partial f}{\partial o}\frac{\partial o}{\partial a}$
$\frac{\partial f}{\partial b} = 5 \cdot 2 = 10$
$\frac{\partial f}{\partial b} = \frac{\partial f}{\partial o}\frac{\partial o}{\partial b}$
$\frac{\partial (5o)}{\partial o} = 5$
$\frac{\partial f}{\partial o}$
„Local gradient“
„Global gradient“
e.g. $f(o) = 5o$

Computational Graphs: Addition

Addition ~ Gradient distributor

Forward pass

Backward pass



a
b
2
3
5
$+$
$\frac{\partial o}{\partial a} = 1$
$\frac{\partial o}{\partial b} = 1$
$o(a,b) = a + b$
$\frac{\partial f}{\partial a} = 5 \cdot 1 = 5$
$\frac{\partial f}{\partial b} = 5 \cdot 1 = 5$
$\frac{\partial f}{\partial a} = \frac{\partial f}{\partial o}\frac{\partial o}{\partial a}$
$\frac{\partial f}{\partial b} = \frac{\partial f}{\partial o}\frac{\partial o}{\partial b}$
$\frac{\partial(5o)}{\partial o} = 5$
$\frac{\partial f}{\partial o}$

„Local gradient“

„Global gradient“

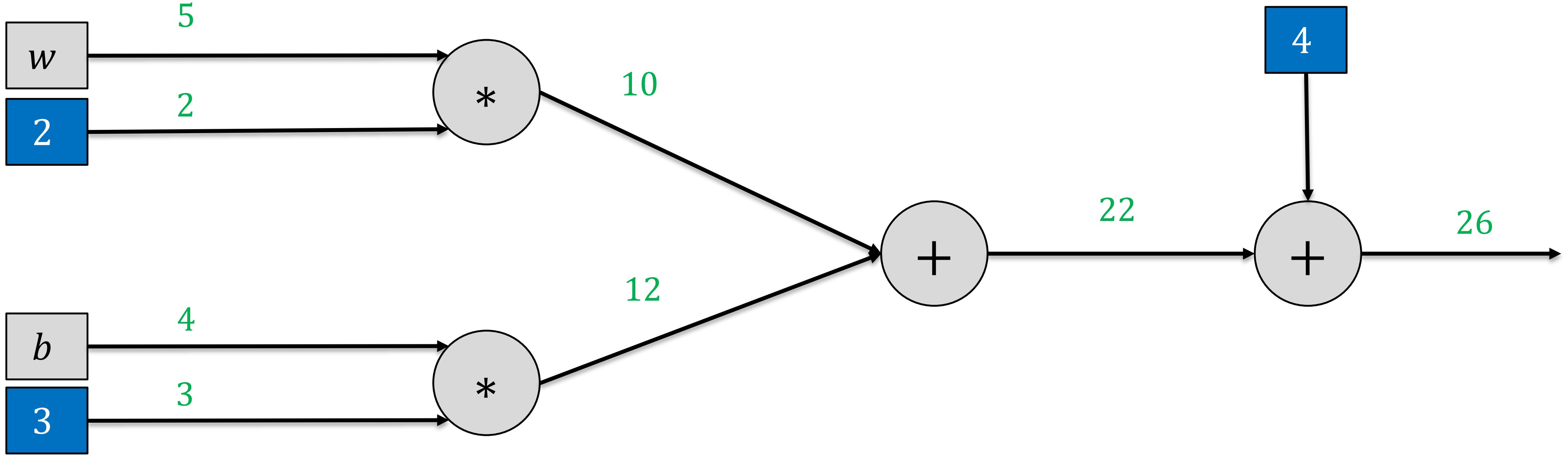
e.g. $f(o) = 5o$

Computational Graphs

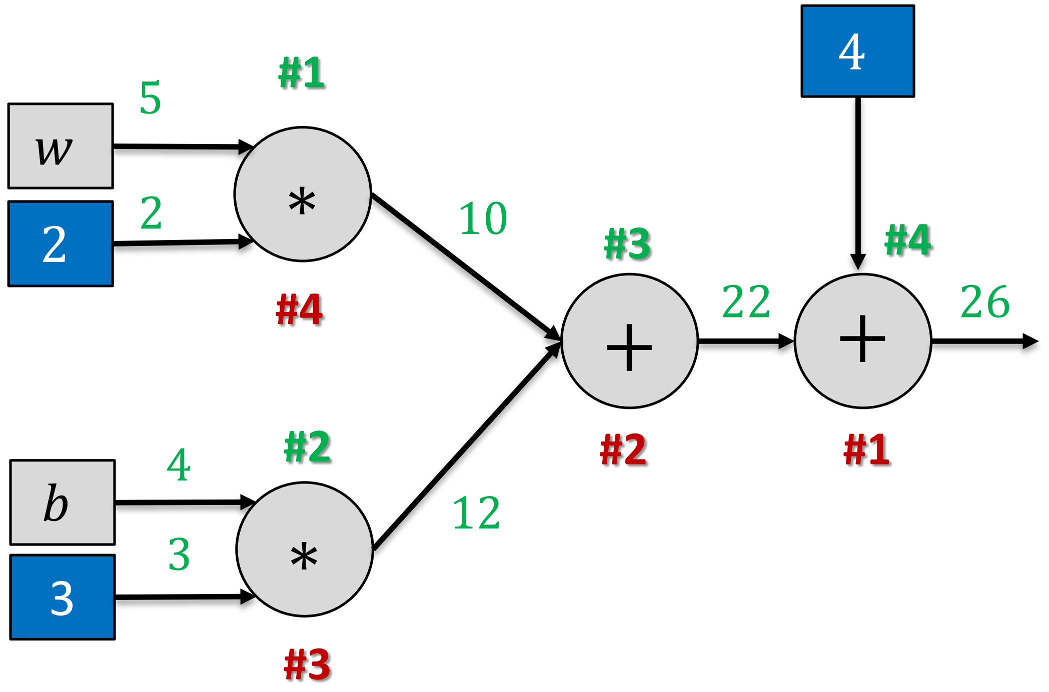
Example: $L(w, b) = 2w + 3b + 4$

Forward pass

Backward pass



Finding the gradients in the backward pass



Solution Recipe for a „Gradient Tape“:

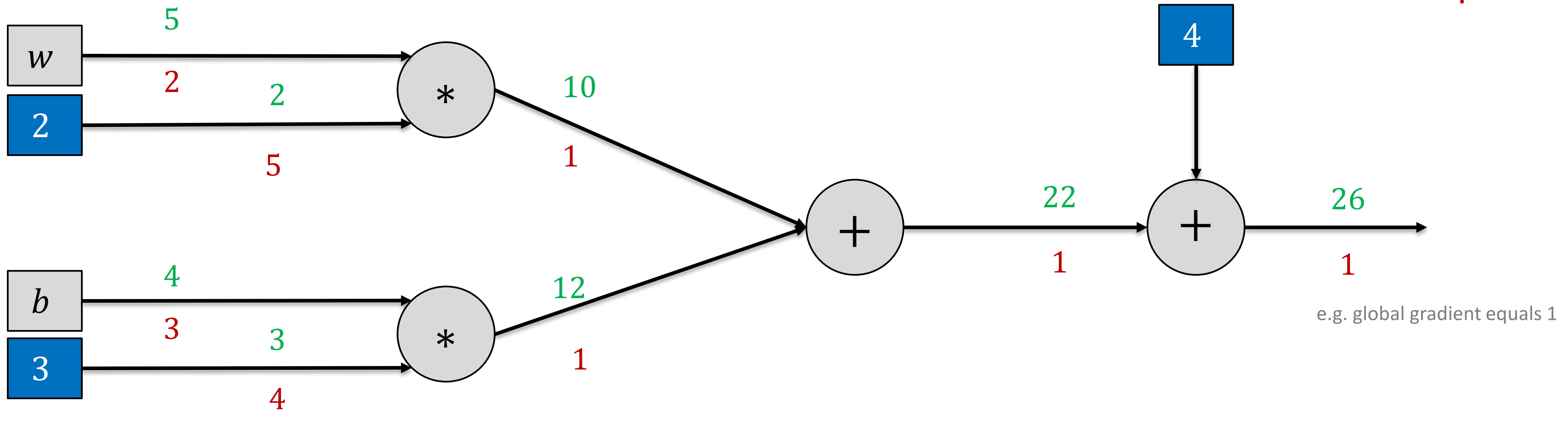
1. Start with the final node f and set its gradient to 1 (since $\frac{\partial f}{\partial f} = 1$)
2. Traverse all the operation nodes in the **opposite** order of the forward pass
3. Perform the local gradient multiplication at every node
   - You can be sure to have already calculated $\frac{\partial f}{\partial g}$ for all the outgoing nodes

Computational Graphs

Example: $L(w, b) = 2w + 3b + 4$

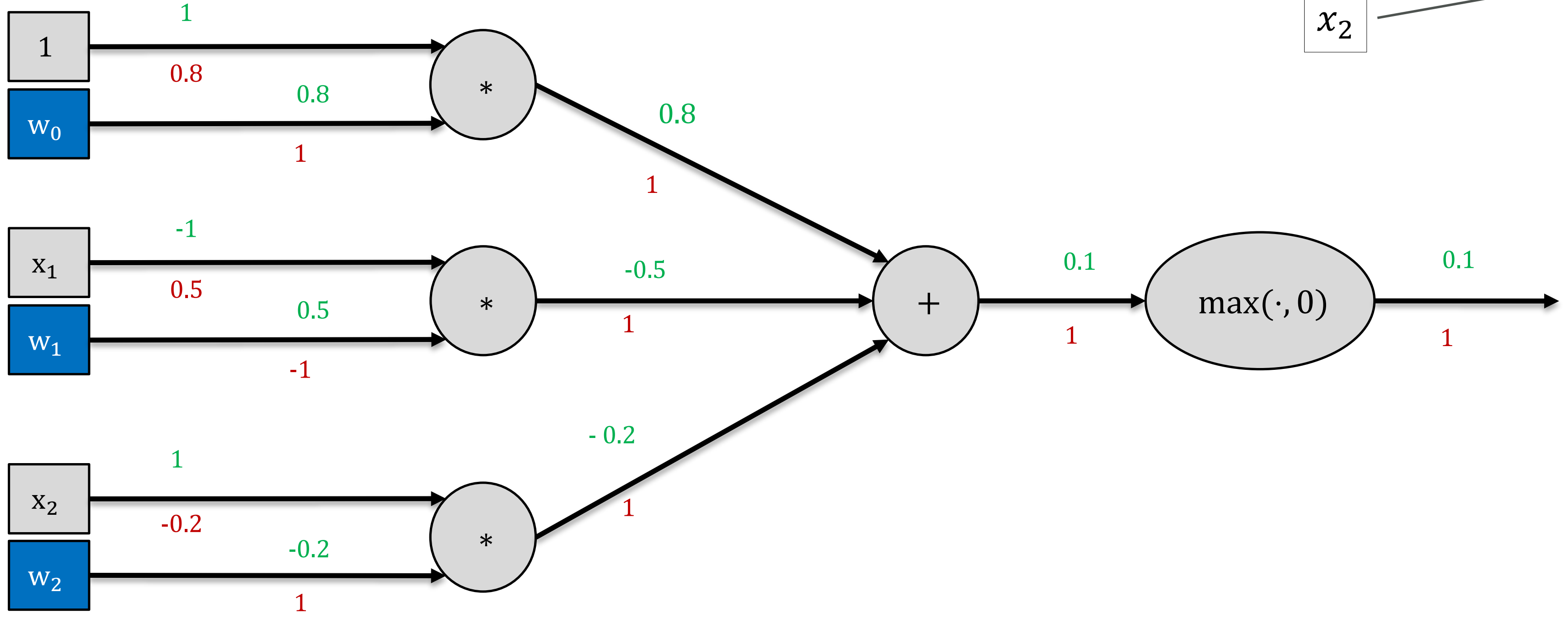
Forward pass

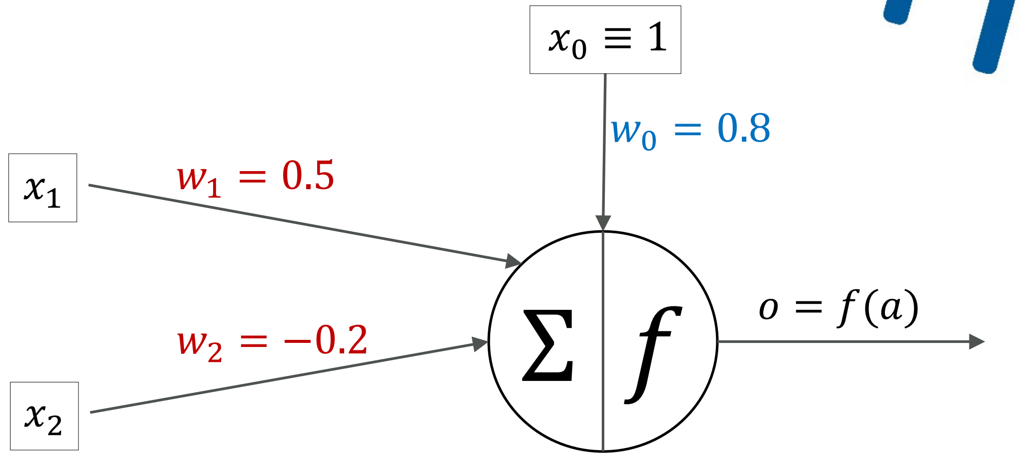
Backward pass



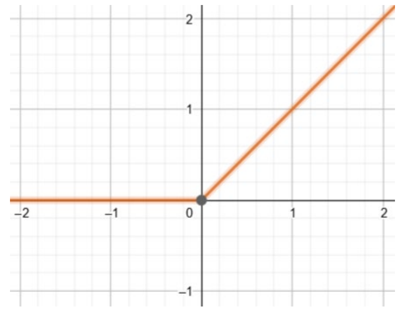
Computational Graph of a Single Neuron

$$f(w_0, w_1, w_2; x_1, x_2) = \max\left(\sum_{i=1}^{3} w_i x_i\,, 0\right)$$



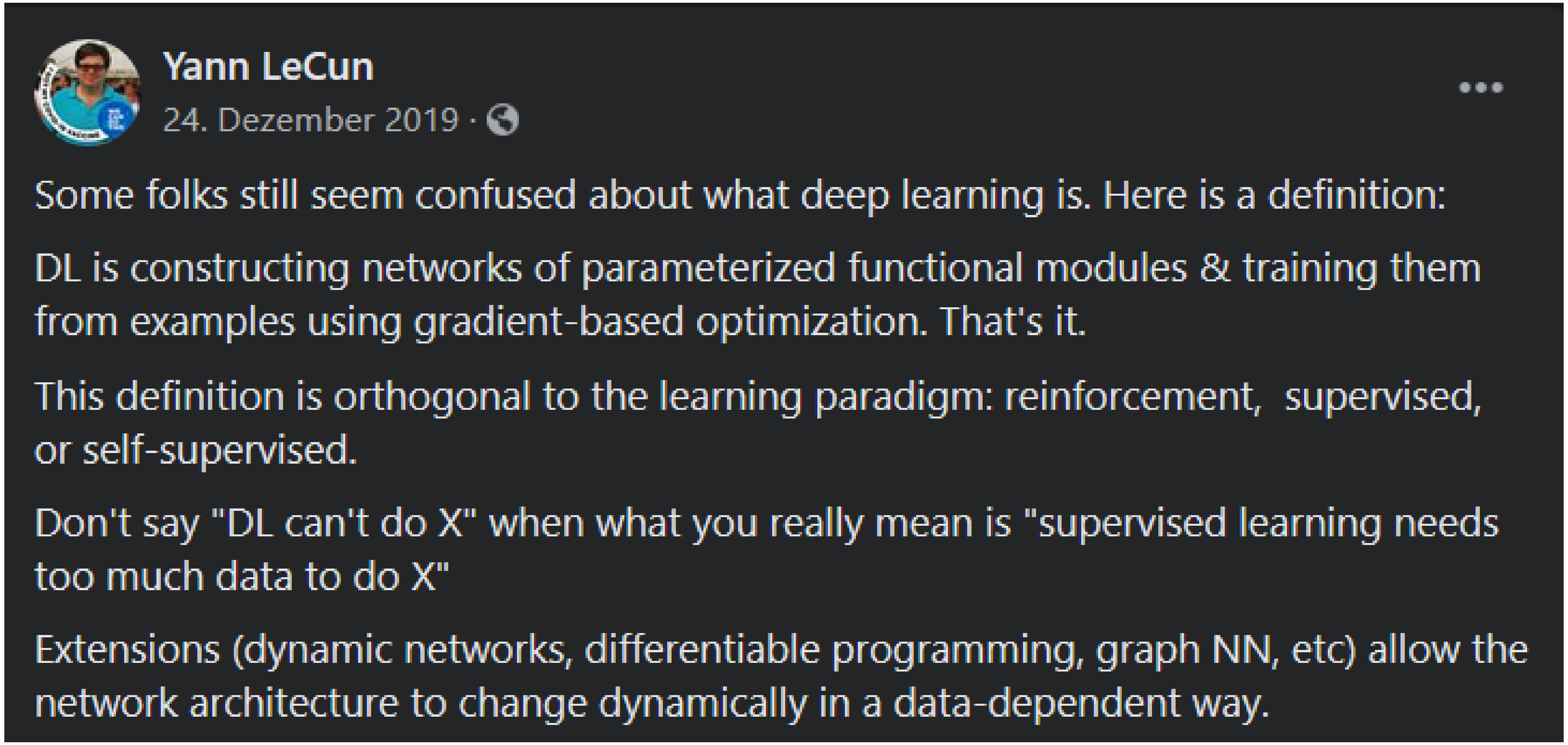


$$a = \sum_{i=0}^{n} w_i x_i$$



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

A modern definition of deep learning …



Yann LeCun

24. Dezember 2019 ·

Some folks still seem confused about what deep learning is. Here is a definition:

DL is constructing networks of parameterized functional modules & training them from examples using gradient-based optimization. That's it.

This definition is orthogonal to the learning paradigm: reinforcement, supervised, or self-supervised.

Don't say "DL can't do X" when what you really mean is "supervised learning needs too much data to do X"

Extensions (dynamic networks, differentiable programming, graph NN, etc) allow the network architecture to change dynamically in a data-dependent way.

FYI: https://en.wikipedia.org/wiki/Yann_LeCun

Deep Learning – Turing Award (“Nobel prize of computer science”)

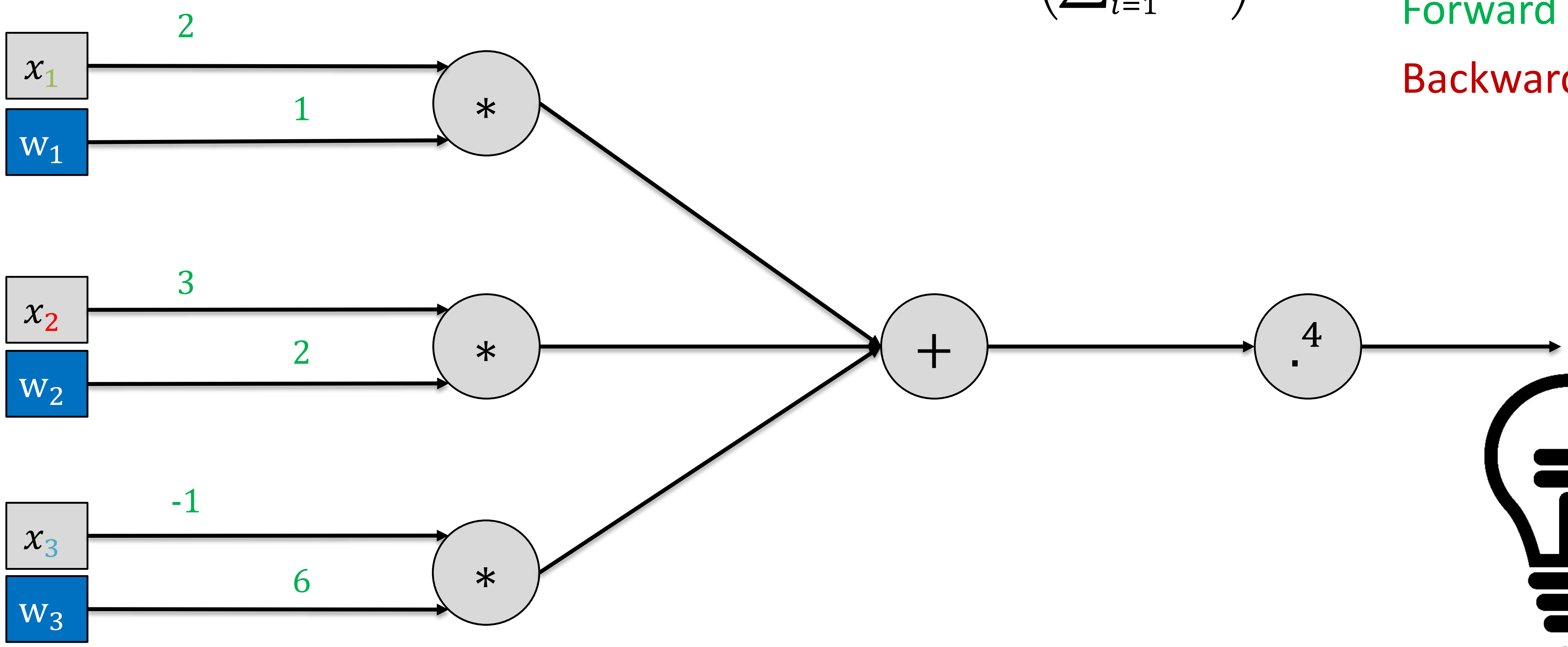
2018 ACM Turing Award for **Deep Learning**

Test your understanding: Computational Graphs

$$f(w_1, w_2, w_3; x_1, x_2, x_3) = \left(\sum_{i=1}^{3} w_i x_i\right)^4$$

Forward pass

Backward pass

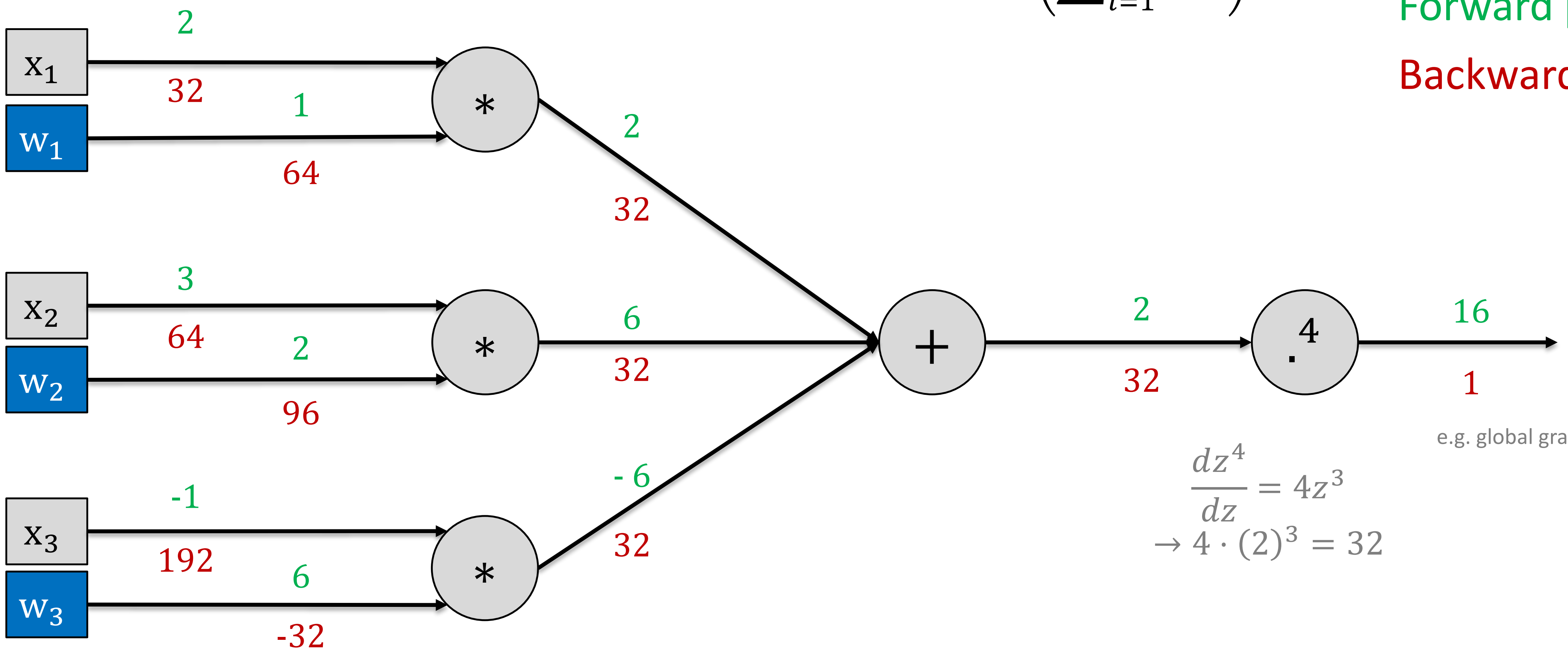


Computational Graphs

$$f(w_1, w_2, w_3; x_1, x_2, x_3) = \left(\sum_{i=1}^{3} w_i x_i\right)^4$$

Forward pass

Backward pass

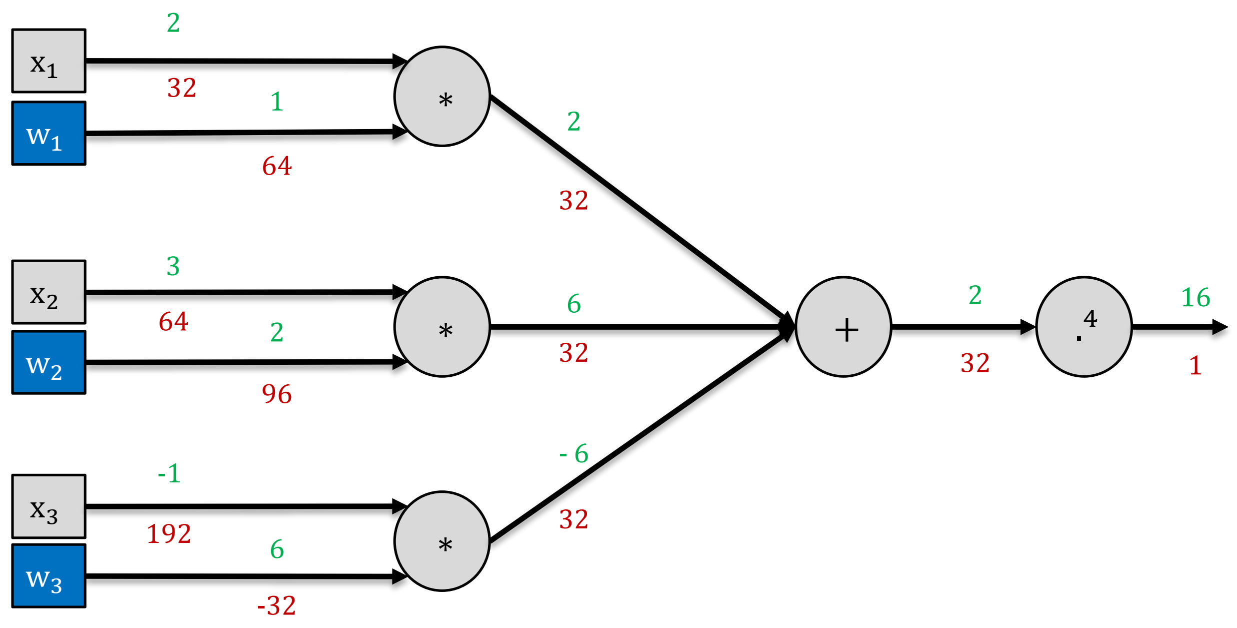


e.g. global gradient equals 1

$$\frac{dz^4}{dz} = 4z^3$$
$$\rightarrow 4 \cdot (2)^3 = 32$$

Computational Graphs in PyTorch

$$f(w_1, w_2, w_3; x_1, x_2, x_3) = \left(\sum_{i=1}^{3} w_i x_i\right)^4$$



```
import torch
x = torch.tensor([2., 3., -1.], requires_grad  = True)
w = torch.tensor([1., 2., 6.], requires_grad  = True)
z = torch.dot(x,w)
f = torch.pow(z,4)
print(f)
```

```
tensor(16., grad_fn=<PowBackward0>)
```

```
f.backward()
print(x.grad)
print(w.grad)
```

```
tensor([ 32.,  64., 192.])
tensor([ 64.,  96., -32.])
```

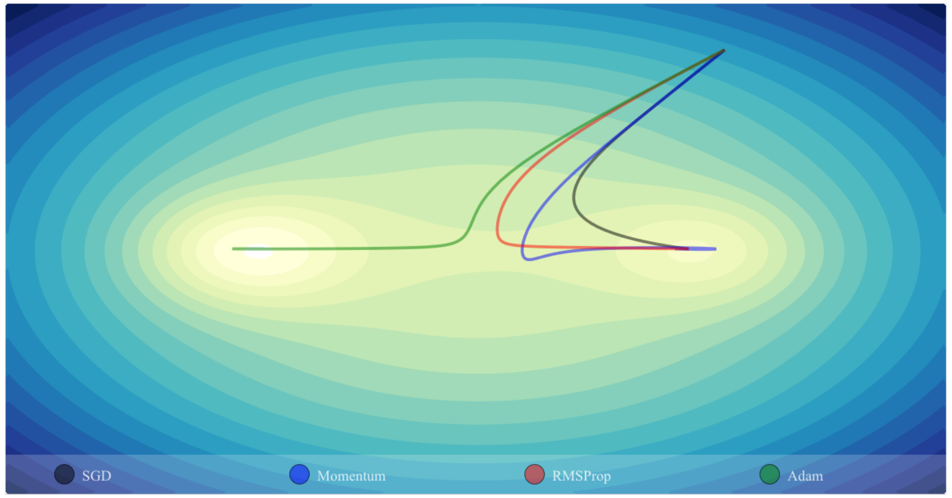
Efficient Gradient-based optimizers

Task

Have a look at:
https://bl.ocks.org/EmilienDupont/aaf429be5705b219aaaf8d691e27ca87

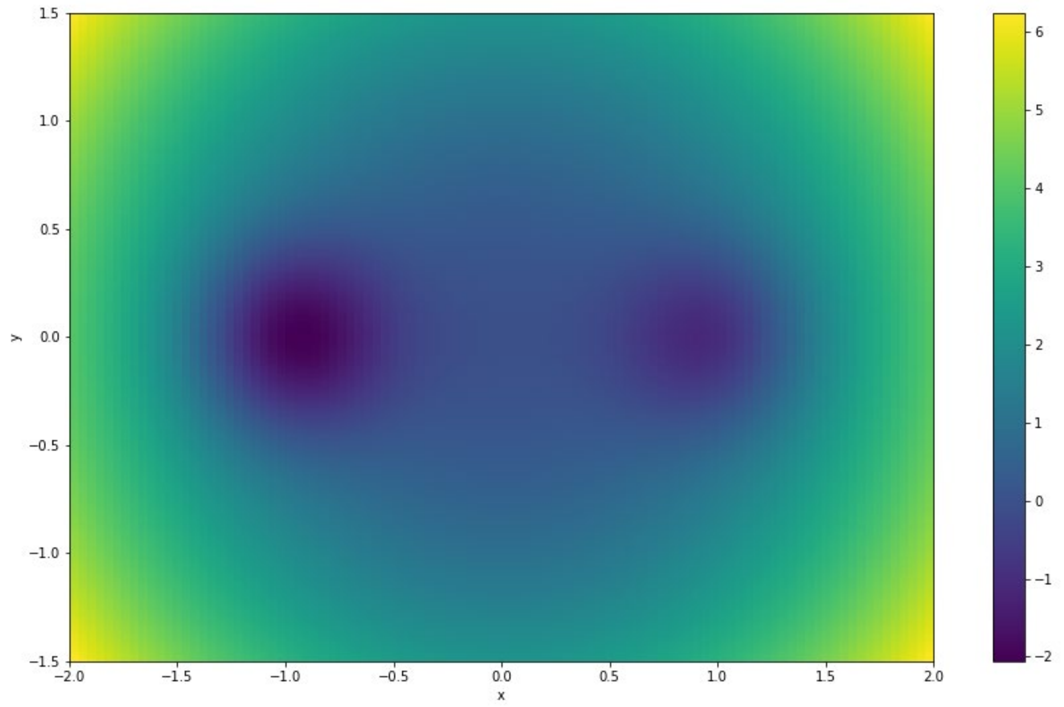
- What can you see? What are the differences between the trajectories?

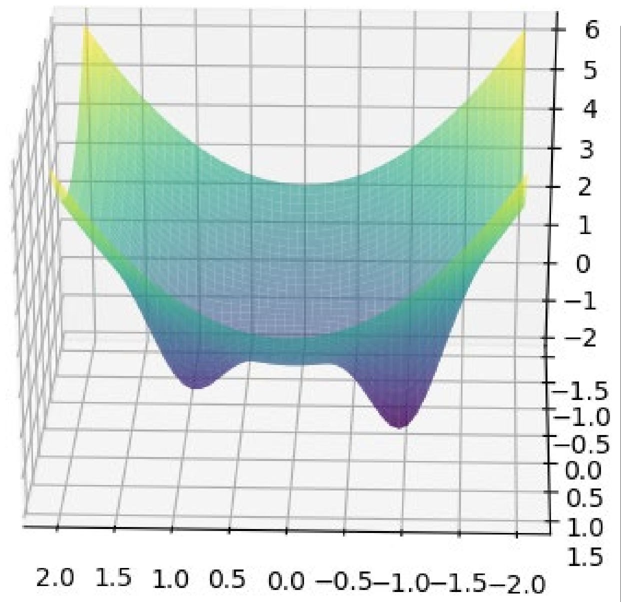
Optimization Algorithms Visualization



YOUR TURN

Our test optimization landscape

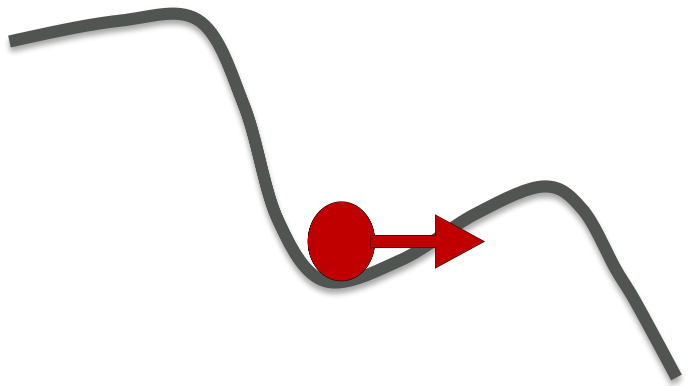




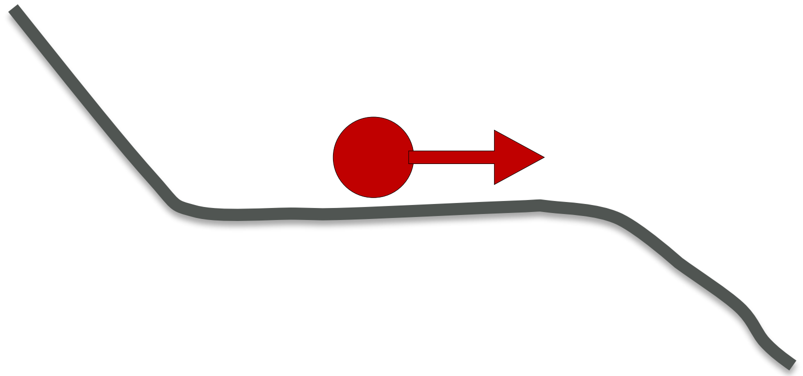
The same as https://bl.ocks.org/EmilienDupont/aaf429be5705b219aaaf8d691e27ca87

Problems for local gradient-based search

Local Minima



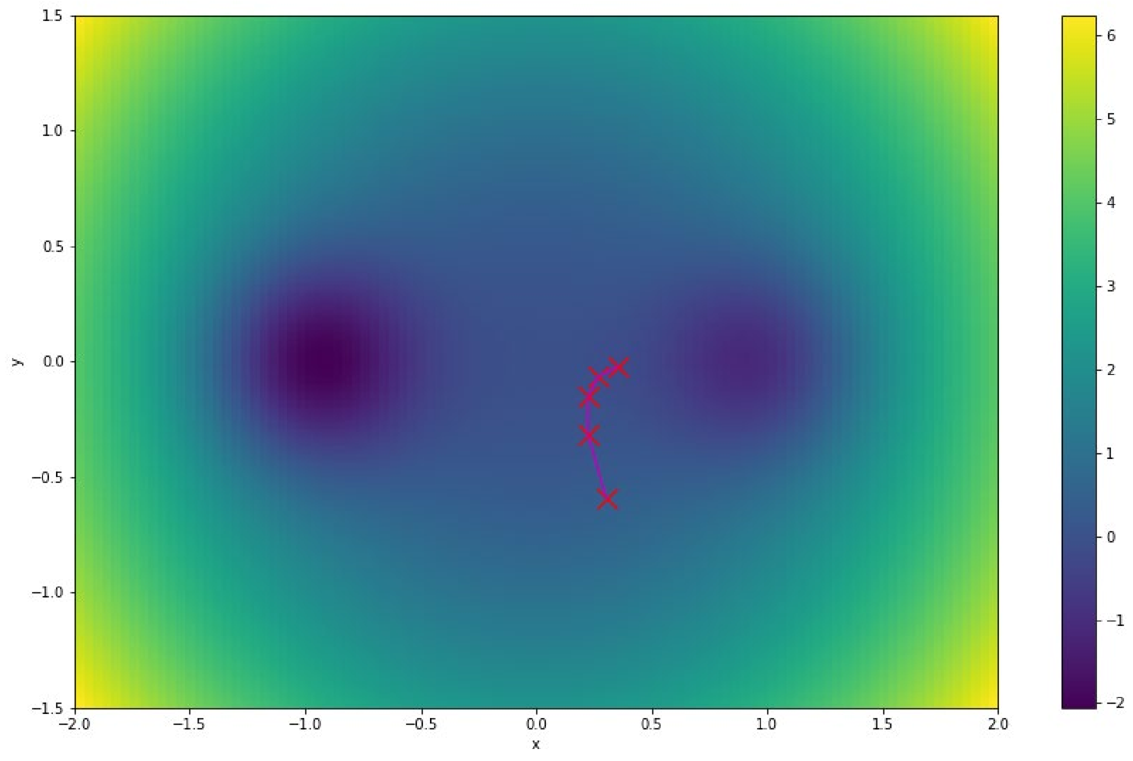
Saddle points



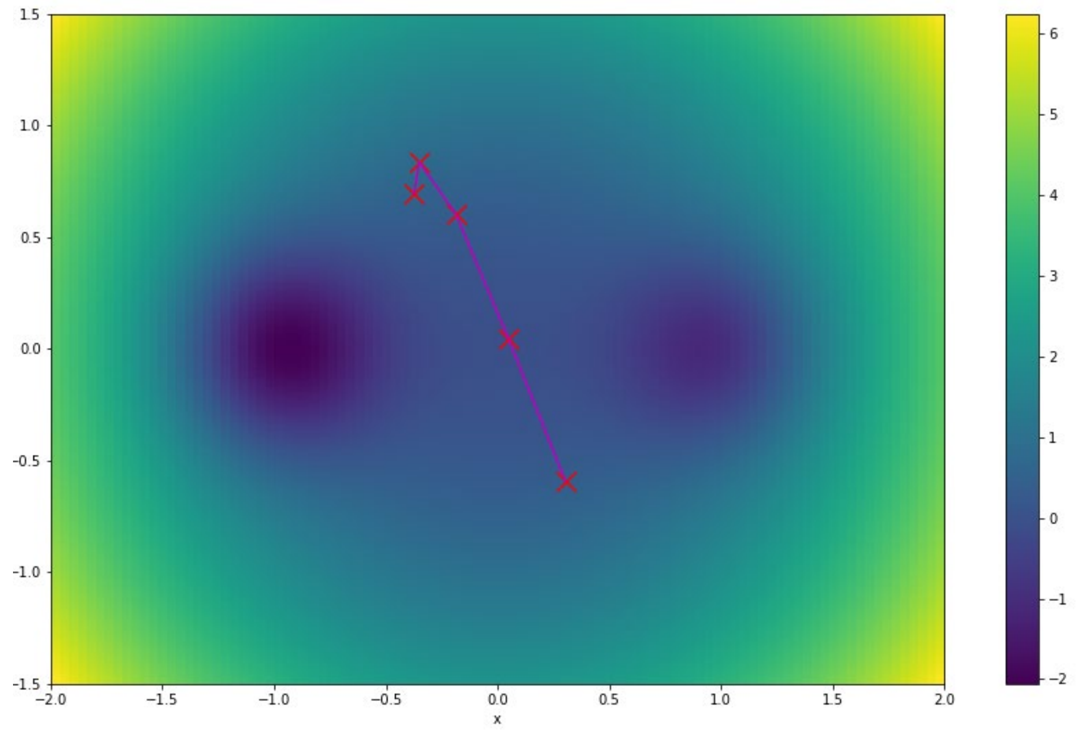
- Gradient gets close to 0
- Little progress in terms of learning
- Saddle points more common (and problematic) than local minima (esp. in high dimensions)

[Dauphin et al., "Identifying and attacking the saddle point problem in high-dimensional nonconvex optimization", 2014]

Gradient descent with momentum



Standard Gradient Descent



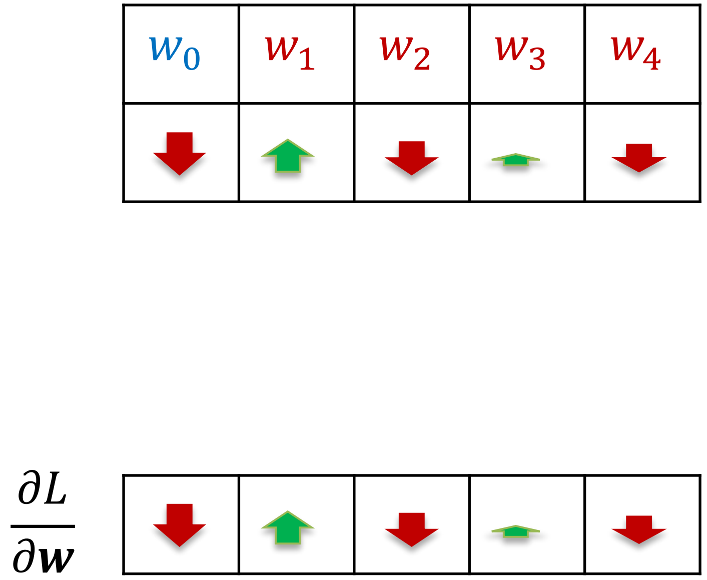
Momentum: Tries to accelarte, similar to a ball rolling down a hill

[Ruder, “An overview of gradient descent optimization algorithms”, 2017]

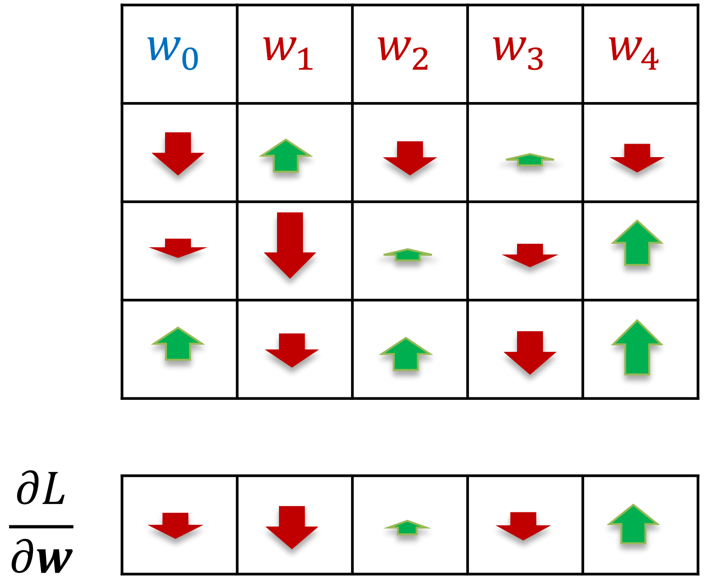
[Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013]

Sidenote for stochastic gradient descent

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```



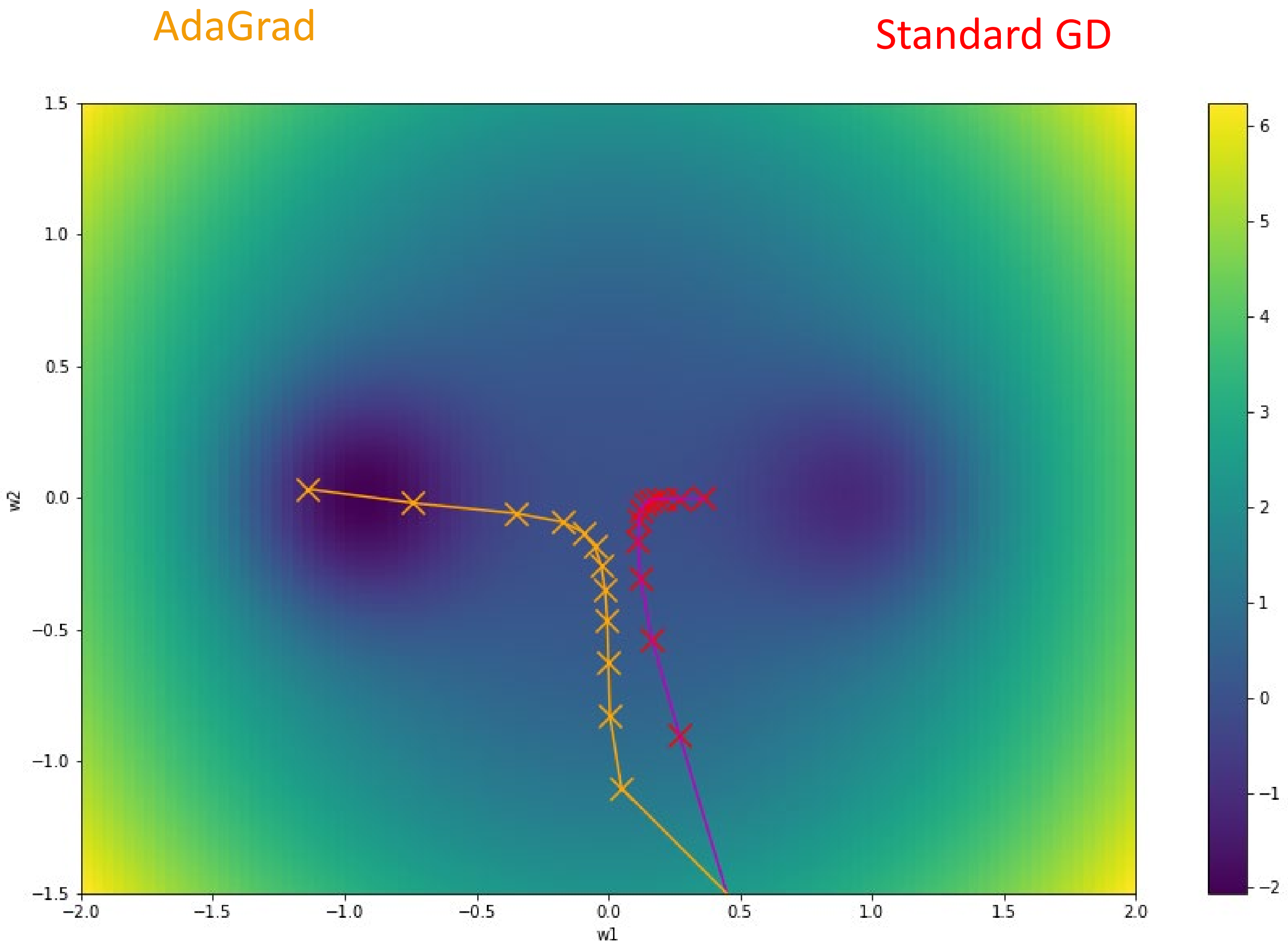
„Stochastic gradient descent (SGD)“



„Batch gradient descent “

[Ruder, “An overview of gradient descent optimization algorithms”, 2017]

AdaGrad



Problem: Sometimes we have huge (>1000%) differences in gradients
→ We want to normalize gradients

Progress along "*steep*" directions is damped;

Progress along "*flat*" directions is accelerated;

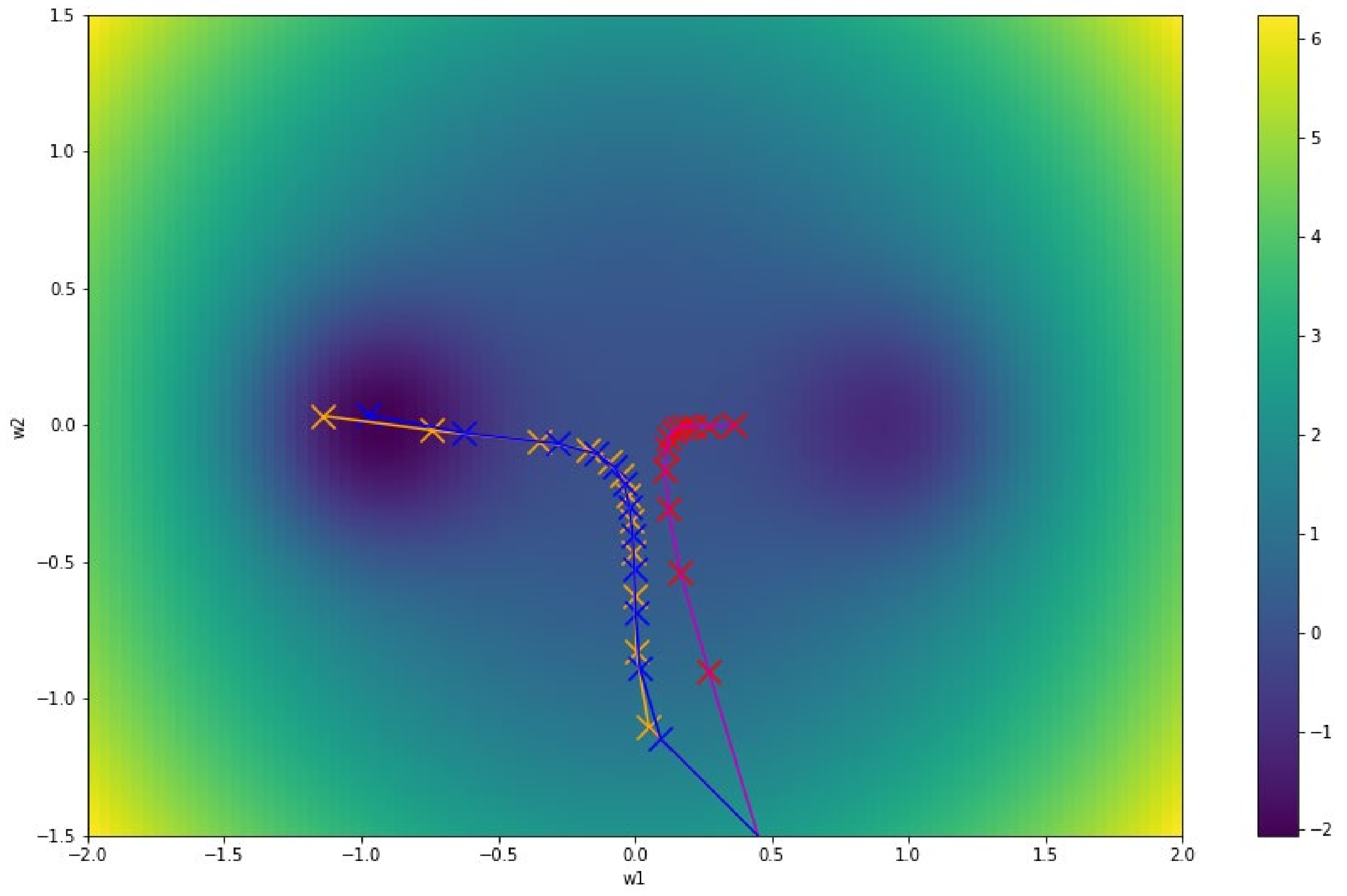
But what happens over the course of training?

Step sizes keep decreasing …

[Duchi et al., "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization", 2011]

RMSProp

AdaGrad RMSProp Standard GD



Fixes a detail of AdaGrad: AdaGrads leads to very small update steps with increasing training duration because it *accumulates* gradient steps

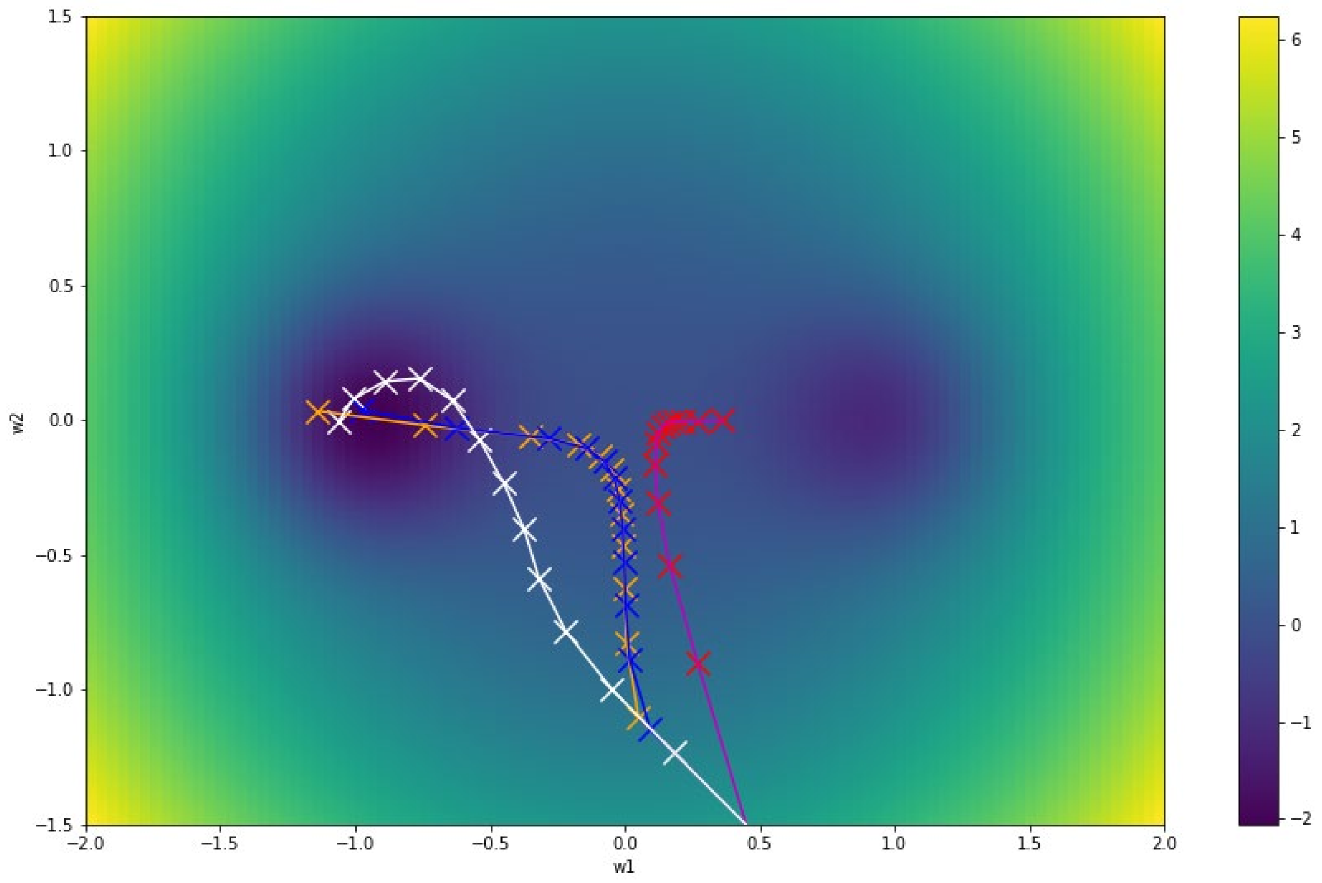
→ Normalize gradients by a moving average of squared gradients (the root of it, hence RMS)

Can‘t we combine this idea with momentum?

[Tieleman and Hinton, 2012]

Adam (= „Adaptive Momentum Optimizer“)

AdaGrad RMSProp Adam Standard GD



Combines Ideas from multiple Optimizers:

Building momentum (GD+Momentum)
First moment

Normalizing by squared gradient (AdaGrad, RMSProp) *Second moment*

[Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015]

Take away messages

- We use gradients to **optimize our models** w.r.t. to a **performance measure**
- Because our performance measure has to be differentiable, **we cannot use accuracy directly**. Instead, we have to use a **differentiable loss function**
- To calculate the gradient of a neural network, we use the **backpropagation algorithm**. It‘s based on representing our model as a **computational graph**
- There exist **many heuristics** for achieving good optimization trajectories. Some are based on natural counterparts, e.g. momentum