



Chapter 2

Neural Network Basics

Content of this Chapter

1. Classification
2. Neural Networks
3. Gradient Descent
 1. Backpropagation
 2. Optimisation Algorithms



2.1 Classification

- What is classification?
- What are some popular classifiers?
- What are their (dis-)advantages?

Classification

- General Task:
 - Given a set of objects and their labels (classes)...
 - ... train a classifier to predict the label for a previously unseen object
- Example: Text Classification (20 Newsgroups)
 - Given the text of a document (20 different forums)
 - Predict which forum it came from



20 Newsgroups Text Classification

- 20 Newsgroups is a dataset of online discussion
 - 18,828 documents total
 - 20 forums, some closely related (pc.hardware vs mac.hardware), some highly unrelated (misc.forsale vs religion.christian)
- Popular evaluation set in Natural Language Processing
- Not completely solved (Error rate ~ 11.4%)



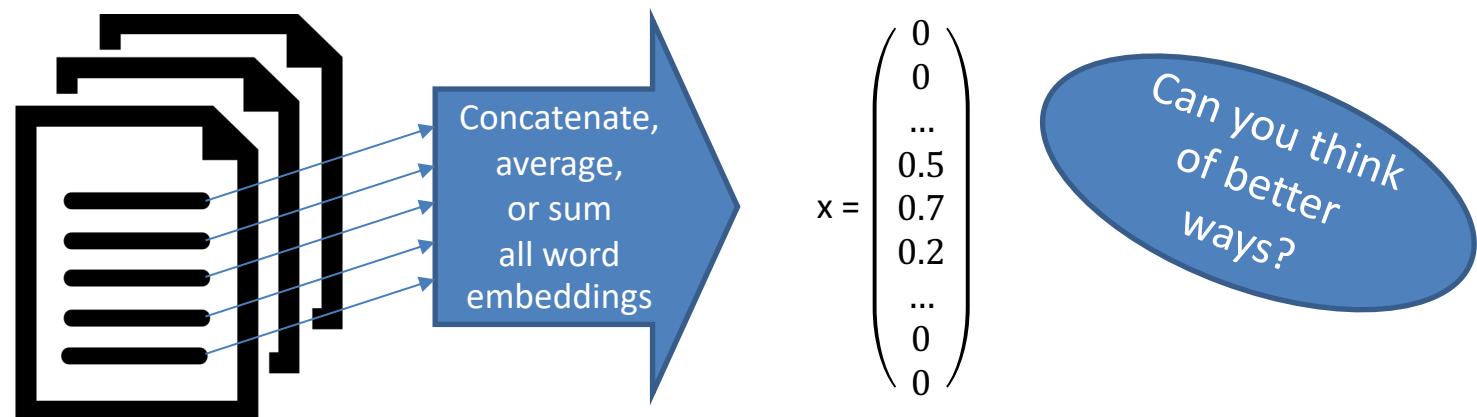
Classification

*Known from
Data Mining*

- Many classification methods exist
 - **Linear Classifiers**
 - Support Vector Machines
 - Decision Trees/Random Forests
 - **Perceptrons**
 - **(Deep) Neural Networks**

Linear Text Classification

- Represent the document as a vector x of (for now) fixed length



- Determine unnormalised probabilities using **weights** and a **bias**
 - Weights:** Which influence does each embedding dim. have on the classification?
 - Bias:** Roughly: Which class is more likely a priori?

Linear Text Classification

- Determine unnormalised probabilities by multiplying with a weight matrix W of dim. 20×300 (classes \times embedding length) and adding a bias b of dim. 1×20

$$y = Wx + b = [0.1, 0.3, 0.2, 0.8, 0.5, 0.7, 0.5, 0.01, 0.7, \dots]$$

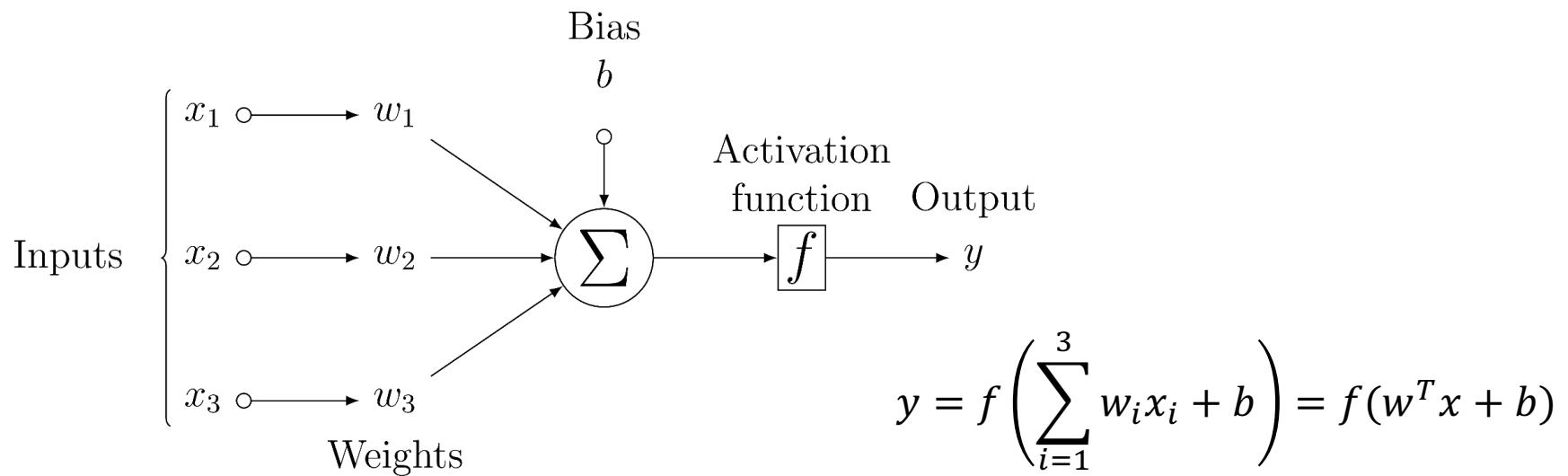


→ Our Goal:

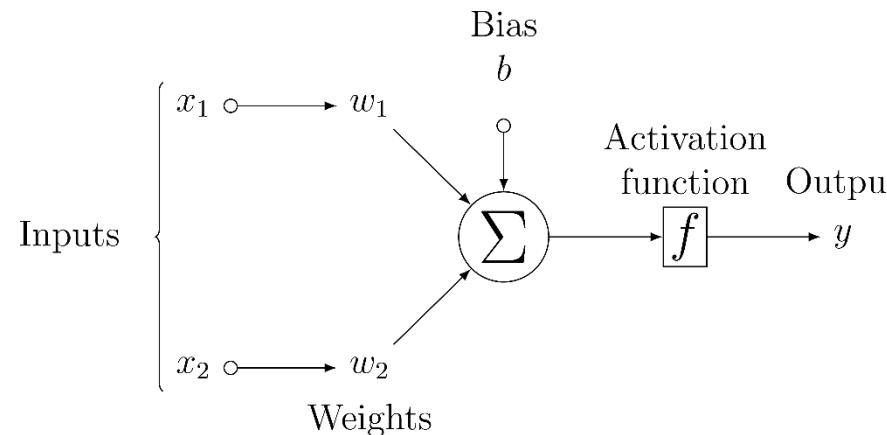
Find values for W and b that provide a good classification!

Perceptron – a simple Classifier

- Inspired by the human brain
 - Models one „brain cell“/**neuron**
 - Based on its inputs, the neuron can either „fire“ or not
- „Firing“ is interpreted as a positive classification



Perceptron for the „or“-function



x_1	x_2	$x_1 \text{ OR } x_2$
0	0	0
1	0	1
0	1	1
1	1	1

- Parameter Settings:

- $w_1 = 1, w_2 = 1$
- $b = -1$
- $f(x) = \sigma(x)$
- Activation:
Neuron „fires“ iff $y \geq 0.5$

Example:

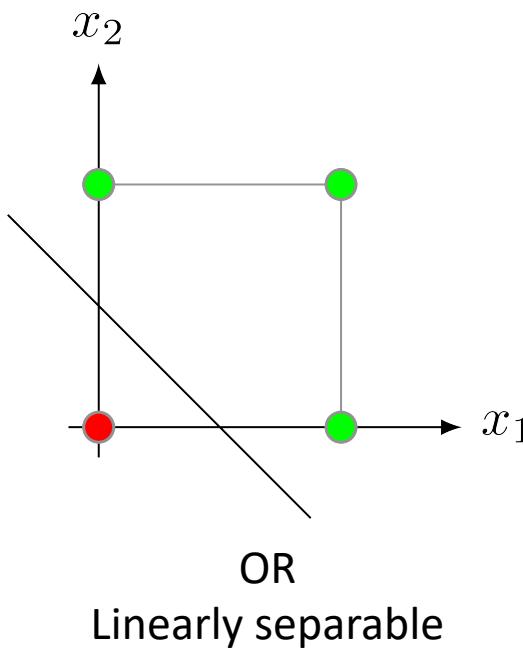
Let $x_1 = 1$ and $x_2 = 0 \rightarrow x_1 \text{ OR } x_2 = 1$

$$\begin{aligned}
 y &= \sigma(w_1 \cdot x_1 + w_2 \cdot x_2 + b) \\
 &= \sigma(1 \cdot 1 + 1 \cdot 0 + (-1)) \\
 &= \sigma(0) = 0.5
 \end{aligned}$$

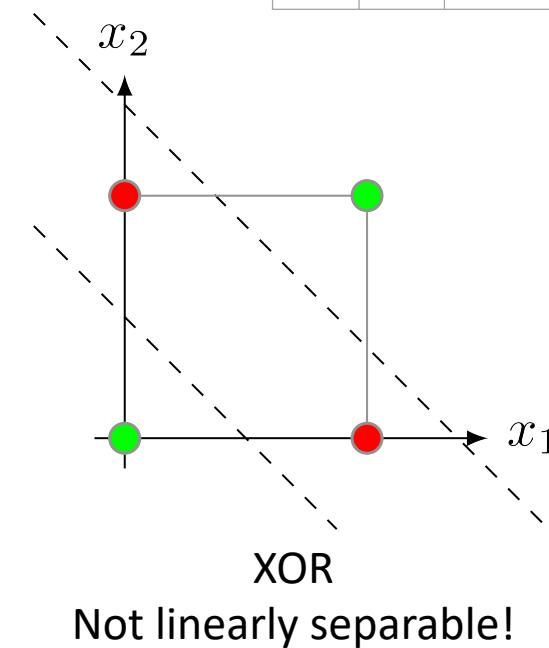
\rightarrow Neuron fires!

Perceptron for the „xor“-Function?

- No such perceptron exists!
- A perceptron can only model linear dependencies



x_1	x_2	$x_1 \text{ XOR } x_2$
0	0	0
1	0	1
0	1	1
1	1	0



Perceptron as a Linear Classifier

- A perceptron is an example of a linear classifier
- An SVM is another example
- These classifiers have some limitations:
They can only learn linear functions
- Many important functions are not linear!
→ For example XOR

Limitations of Linear Classification

- Many functions can not be learned by linear classifiers ☹

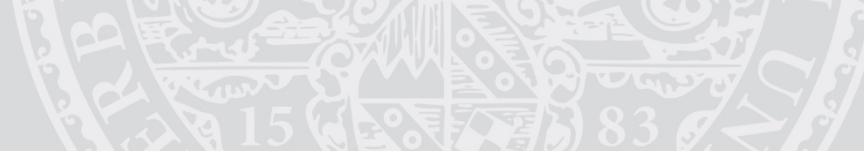
→ We need to modify the model to make it more expressive!

→ For SVMs, use **kernels**

See Data
Mining

→ For other linear classifiers, use multiple layers!

Deep Neural
Networks.
*This is what we
do here!*



2.2 Neural Networks

- What is a neural network?
- What can we do with it?
- How do we use it for classification?



What is a neural network?

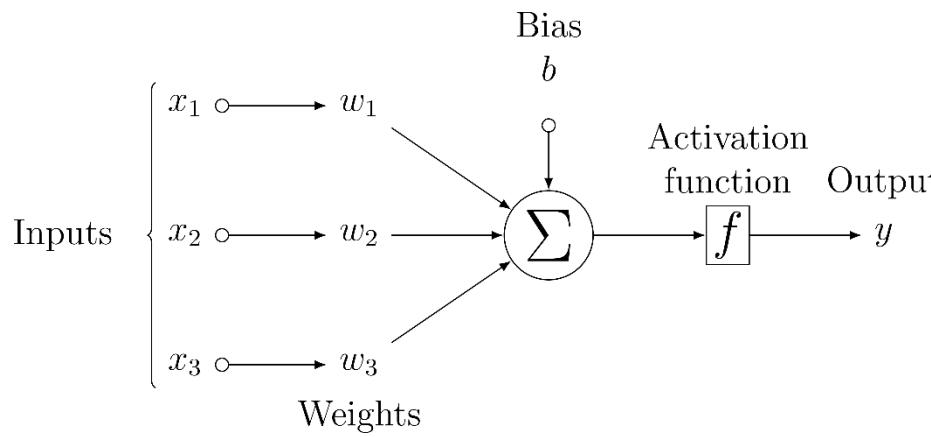
Rough and simple definition:

A simple (fully connected feed-forward) neural network is

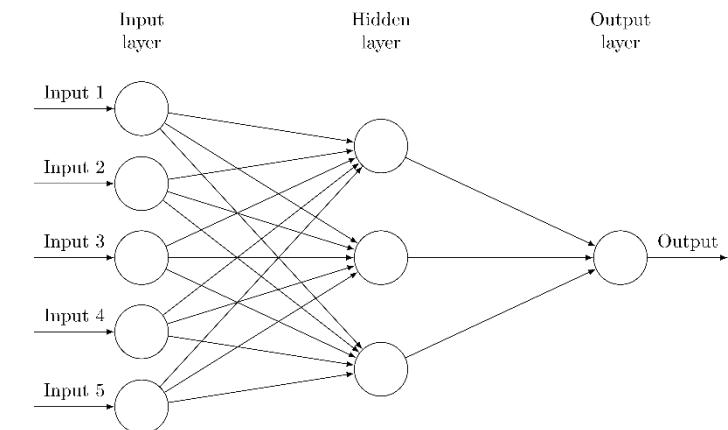
- a bunch of linear classifiers
- with non-linear activation functions
- chained together

Neural Network

- „Model for the human brain“
- Classifier based on neurons (perceptrons) structured in **layers**
- Remember: A single neuron/perceptron is a linear classifier

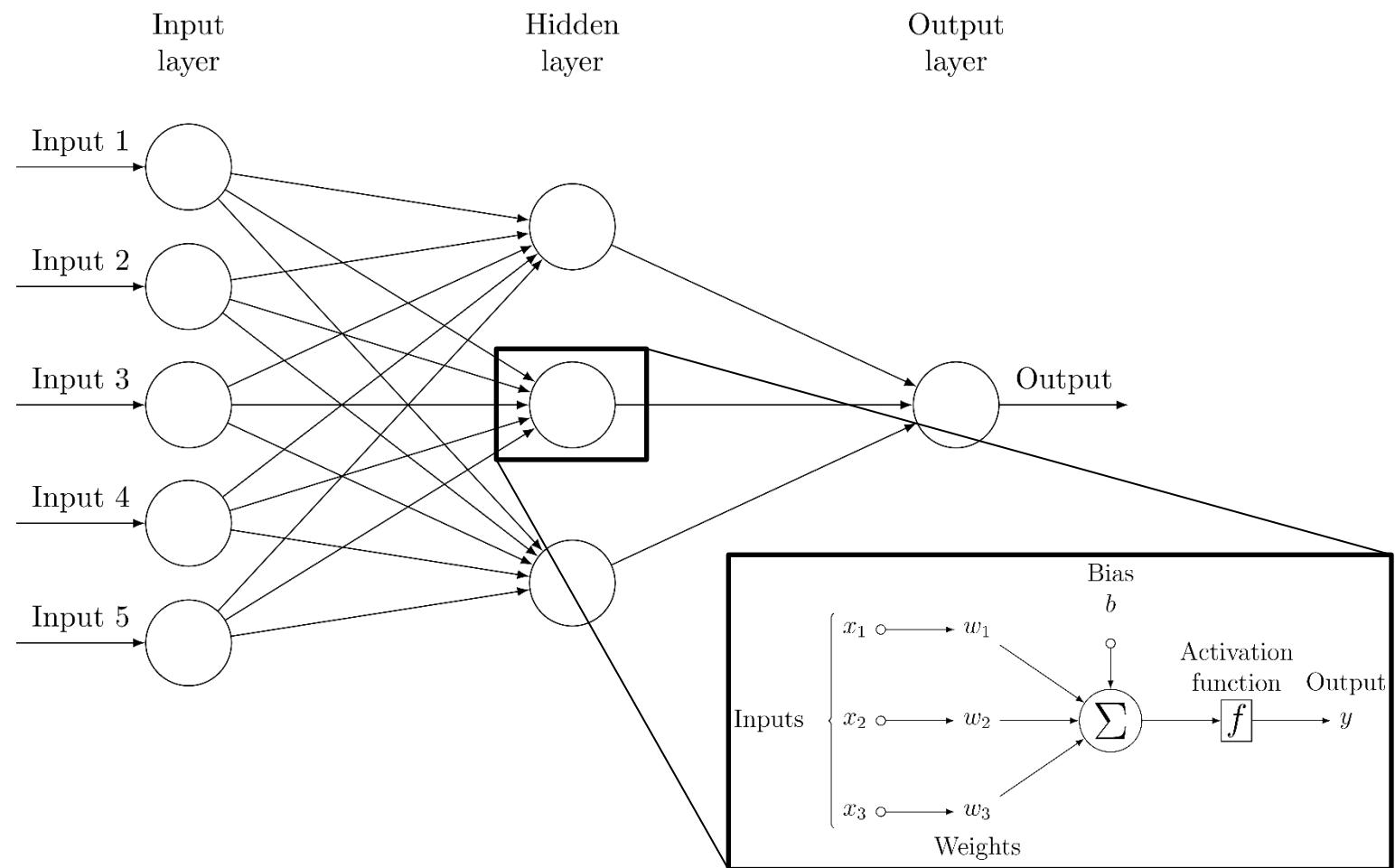


Single Neuron



Neural Network

Neural Networks



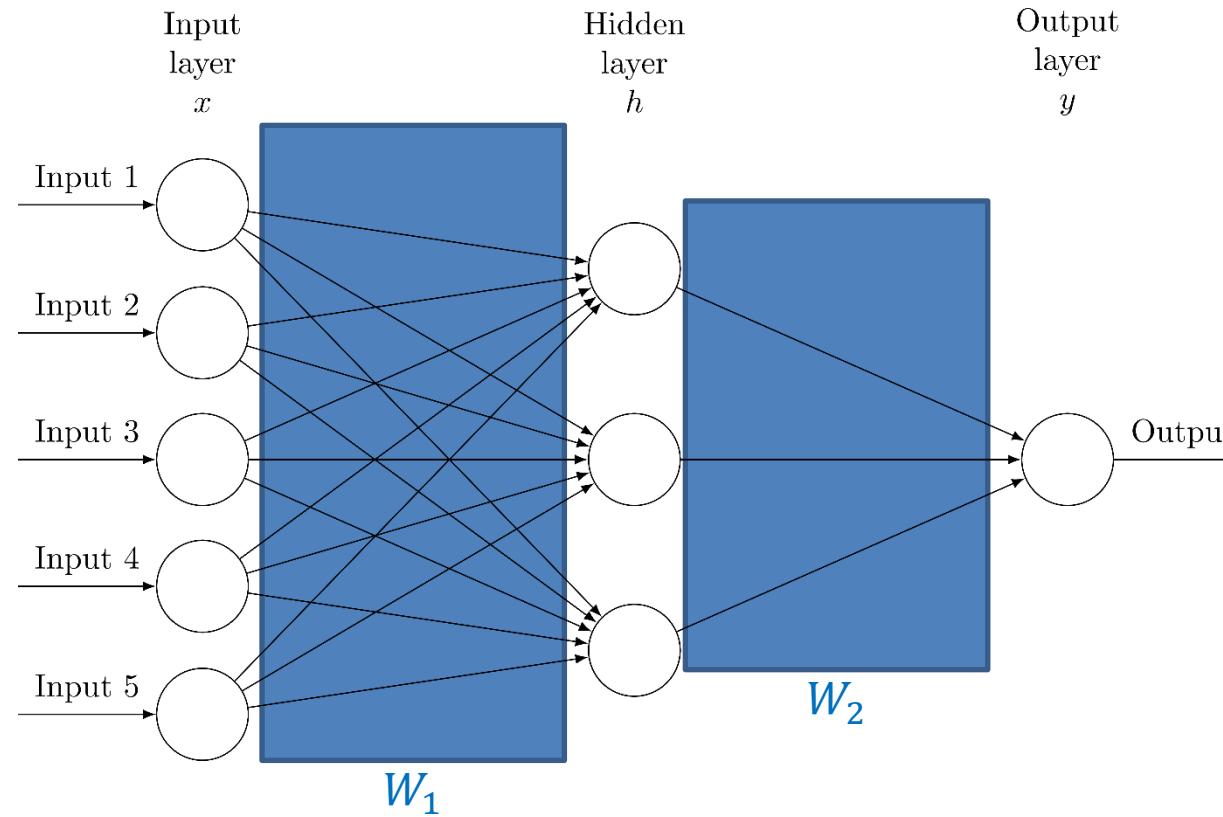
Neural Networks – Forward Pass

- Given
 - A neural network with a fixed set of weights
 - An input sample to be classified
- How to get the classification?

→ A layer is „applied“ by multiplying its weight matrix with its input vector

→ See next slide for an example

Neural Networks



$$\begin{aligned} h &= f(W_1 x + b_1) \\ y &= f(W_2 h + b_2) \end{aligned}$$

Linear classifiers!

Neural Networks as Chained Linear Classifiers

- Remember:
 - A linear classifier provides class scores by calculating $y = f(Wx + b)$
 - A neural network is a chain of linear classifiers with activation functions f (for very simple classifiers $f(q) = q$)

→ A neural network is some function like this:

$$y = f(W_3 f(W_2 f(W_1 x + b_1) + b_2) + b_3)$$

Linear classifier



→ We can chain this as deep as we want

Activation Functions

- After each layer, an **activation function** is used
- An activation function can be **any function**
- However, **non-linearity** is required for a more expressive model
- Why do we need this?
 - Remember: Linear classifiers can separate data linearly
 - A neural network with a linear activation function (e.g. $f(q) = q$) is still a linear classifier (we could reduce all the weight matrices back into one)
 - Non-linear activation functions enable us to learn non-linear relations!

Using Linear Activation Functions

- Using a linear activation function can be reduced to a new linear classifier
- For example, setting $f(q) = q$:

$$\begin{aligned}y &= f(W_3 f(W_2 f(W_1 x + b_1) + b_2) + b_3) \\&= W_3(W_2(W_1 x + b_1) + b_2) + b_3) \\&= W_3(W_2 W_1 x + W_2 b_1 + b_2) + b_3) \\&= W_3 W_2 W_1 x + \underbrace{W_3 W_2 b_1 + W_3 b_2 + b_3}_{\text{constant term}} \\&= \mathbf{Wx} + \mathbf{b}\end{aligned}$$

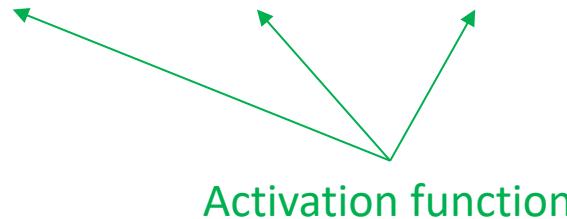
Neural Networks as Chained Linear Classifiers

Using some common non-linear activation functions for f , our network:

$$y = f(W_3 f(W_2 f(W_1 x + b_1) + b_2) + b_3)$$

becomes:

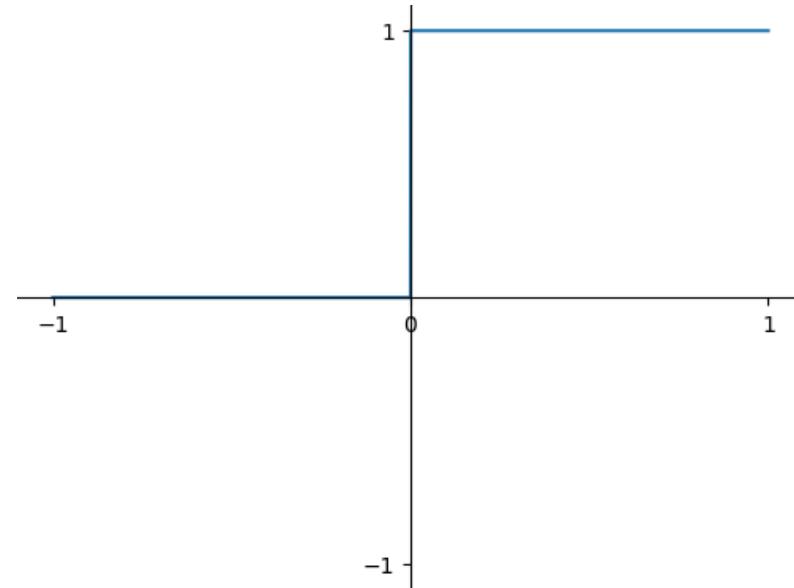
$$y = \text{softmax}(W_3 \tanh(W_2 \tanh(W_1 x + b_1) + b_2) + b_3)$$



Activation Functions

- Initial approaches were inspired by natural sciences

$$\text{heaviside}(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$$

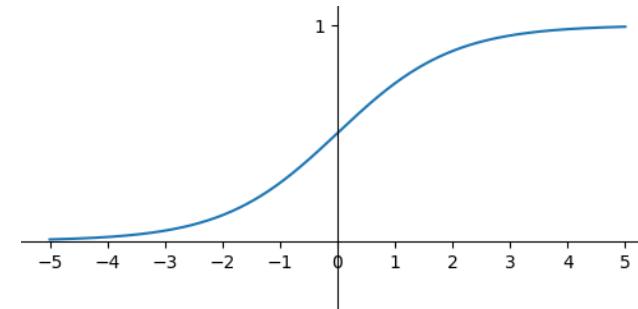


- Its derivative posed some problems however
 - Not defined at 0
 - 0 everywhere else

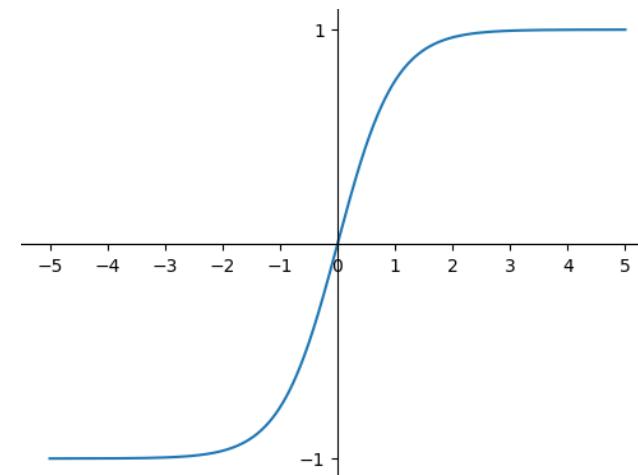
Activation Functions

- Better functions emerged to solve these problems

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



tanh

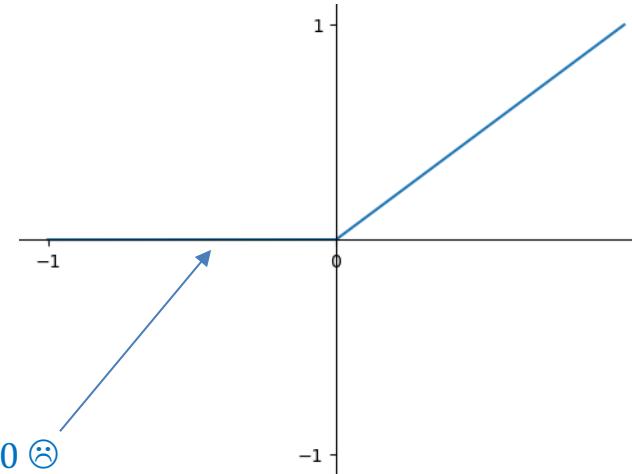


Activation Functions

- ReLU proved to be best, resulting in more variants thereof

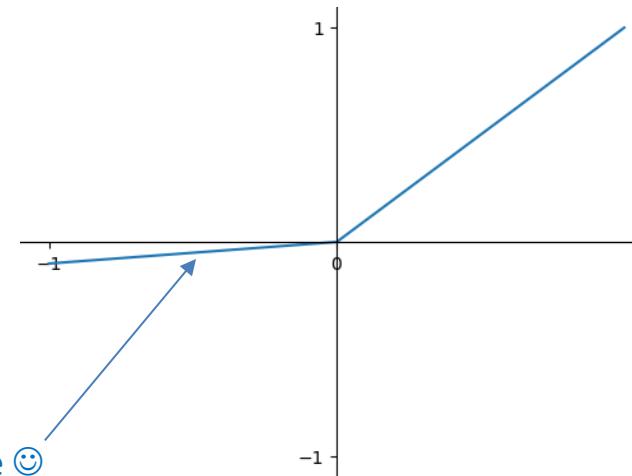
$$\text{ReLU}(x) = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases}$$

Derivative is zero for $x < 0$ ☹



$$\text{LeakyReLU}(x) = \begin{cases} ax, & x < 0 \\ x, & x \geq 0 \end{cases}$$

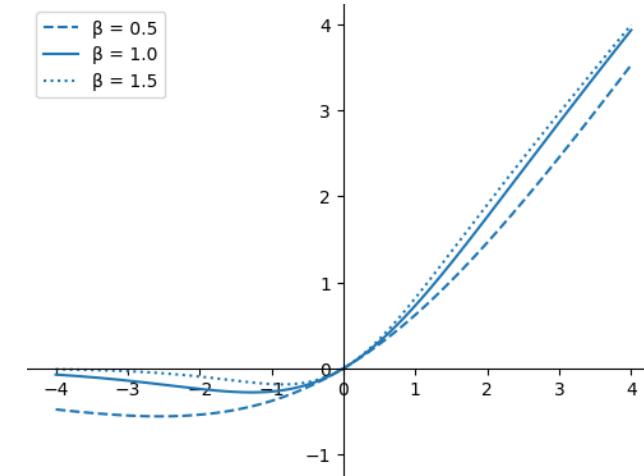
No „dead“ derivative ☺



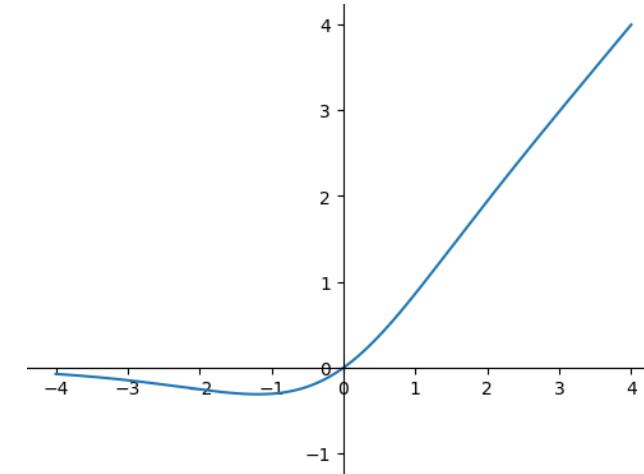
Even More Activation Functions

- A great deal of research has gone into further improvement

$$^1 \text{swish}(x) = x \cdot \frac{1}{1 + e^{-\beta x}}$$



$$^2 \text{mish}(x) = x \cdot \tanh(\ln(1 + e^x))$$



¹ Prajit Ramachandran, Barret Zoph, and Quoc V. Le. (2017).

[Searching for Activation Functions.](#)

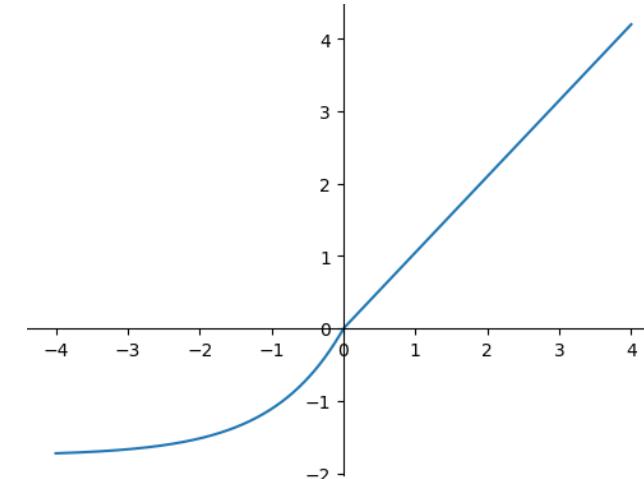
² Diganta Misra. (2019).

[Mish: A Self Regularized Non-Monotonic Neural Activation Function.](#)

Even More Activation Functions

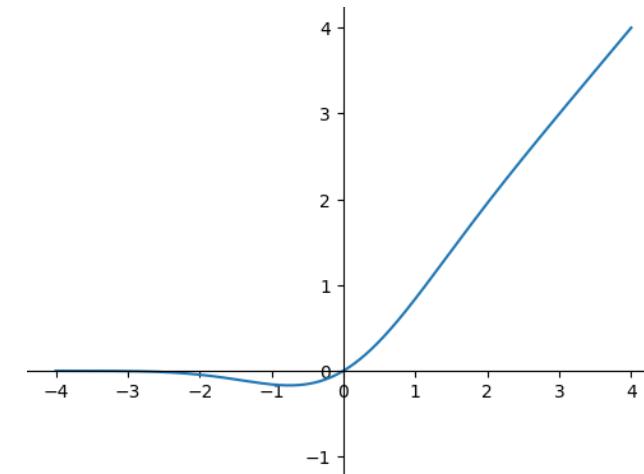
- A great deal of research has gone into further improvement

$$^1 \text{selu}(x) = \lambda \begin{cases} x, & x > 0 \\ \alpha e^{-x} - \alpha, & x \leq 0 \end{cases}$$



$$^2 \text{gelu}(x) = 0.5x \left(1 + \tanh \left(\sqrt{\frac{2}{\pi}} \cdot (x + \alpha x^3) \right) \right)$$

Mostly used in modern NLP



¹ Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. (2017). [Self-Normalizing Neural Networks](#).

² Dan Hendrycks, and Kevin Gimpel. (2016). [Gaussian Error Linear Units \(GELUs\)](#).



Neural Networks – Optimisation

- So far, we assumed given weights for our networks
- In practice, we have given **data** and need to **find the weights**

→ We need a method to do that!

- Roadmap:
 - Rate the current weights
 - Change the weights to get a better rating

Neural Networks – Loss Function

- Recall Word2Vec:
 - Given a **corpus** of contexts („training examples“)...
 - ... determine the **quality** of the network’s current weights
- Generally: Rate the current weights W (more generally parameters θ) by a **loss function**

A loss function $L_\theta(x, l)$ quantifies how unhappy you would be if you used θ to make a prediction on x when the correct output is l . It is the object we want to minimize.

<https://web.stanford.edu/class/cs221/lectures/learning1.pdf>

Neural Networks – Loss Function

- A loss function quantifies the error a classifier makes with its current weights
- Common loss function for classification:
Cross-Entropy Loss
 - Given
 - the **true label** l of a sample (that is, a 1-hot vector with a **1** at the position of the correct class)
 - The label $y_\theta(x)$ x as a vector of probabilities
 - The cross-entropy loss for parameters θ for one sample x is defined as

$$L_\theta(l, x) = - \sum_i l[i] \cdot \log y_\theta(x)[i]$$

All trainable weights

All possible classes

Cross-Entropy Loss

$$L_\theta(l, x) = - \sum_i l[i] \cdot \log y_\theta(x)[i]$$

- L gets smaller the closer y_θ is to l

→ Minimising L leads to better predictions!

→ This is done by changing θ in a way that minimises the loss



Neural Networks – Optimisation

- For shallow networks, finding good weights is easy
- But shallow networks are not very powerful*
- For deep networks, the influence of early layers on the classification is not obvious
- Early attempts at using deep networks failed, because
 - No good optimisation algorithm was known
 - Computational power did not suffice
- Very important step in 1975: Backpropagation!

* In theory, a network with one hidden layer and a finite number of neurons can learn any continuous function in a compact subset of \mathbb{R}^n (*Universal approximation theorem*). However, there is no guarantee that we can find such a network with our optimisation algorithms. In practice, deeper networks work better.



2.3 Backpropagation & Gradient Descent

- How do we optimise the weights in a neural network?

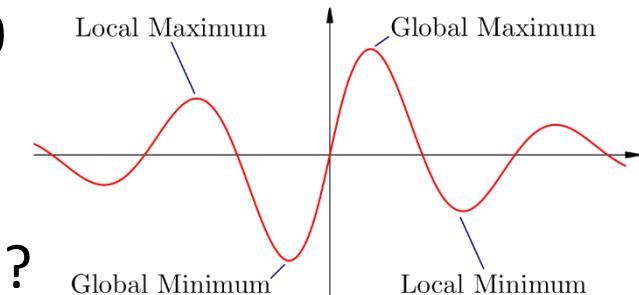
Optimisation

- The big goal in neural network optimisation:
 - Given a neural network, find the weights that provide the best classification by minimising the loss
- General Problem:
 - Given any function, find the input that minimises/maximises this function
 - Methods exist for this kind of problem!
 - Map the optimisation goal to this problem
 - Minimise the objective/loss function by finding the best set of parameters („inputs“)

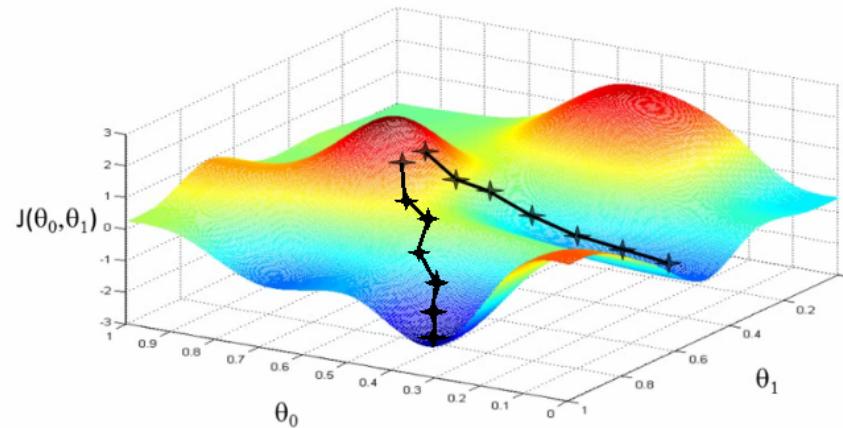
How can we do this?

Optimisation Idea (Known From School)

- Set the first derivative of the function to 0
- Solve for the input analytically
 - $x_{candidate}$
- Is the second derivative at $x_{candidate} \neq 0$?
 - local minimum or local maximum
 - check if minimum/maximum is global
- Check the function value at the found inputs
 - get input that obtains the lowest/highest value
- This is not feasible for neural networks
 - Too many parameters and inputs to solve analytically
- But calculating the derivative is possible
 - Gradient Descent



Gradient Descent – The Analogy



<http://blog.datumbox.com/wp-content/uploads/2013/10/gradient-descent.png>



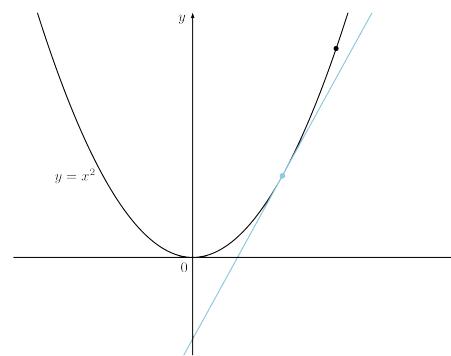
<https://raftrek.com/wp-content/uploads/2015/10/Hiking-down-mountain-ridge.jpg>

Gradient Descent – The Basics

Gradient Descent

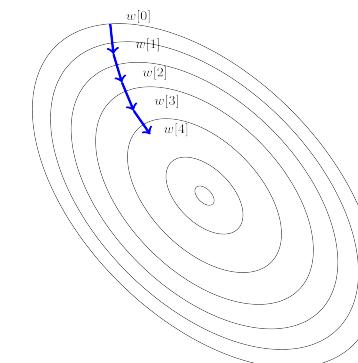
Gradient

= The vector of all partial derivatives of a function



Descent

= Finding a way towards the minimum of the function



Gradient Descent – The Gradient

- Any neural network is just an arbitrarily complex function
- More precisely:
Any neural network is just an arbitrarily complex **chain of simple functions!**
- Simple functions are easy to derive
- Chains of simple functions are also easy to derive by using the **chain rule**



Gradient Descent – Deriving very simple functions

- Some basics rules (you **really** know these from school):

- Multiplying with a constant c

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

- Adding a constant c

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

- Exponentiating with a constant c

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



Gradient Descent – Deriving simple functions

- All these functions had only one parameter
- Neural networks have thousands or even millions!

→ Partial derivatives (you may know these from school)

- Keep all variables except one (e.g. x) constant
 - Derive by x
 - Repeat for all other input variables
-
- The **Gradient** ∇ is the vector of all partial derivatives

Gradient Descent – Deriving simple functions

- An example:

$$f(x, y) = xy$$

$$\rightarrow \frac{\partial f}{\partial x} = y$$

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

y is treated as a constant

$$\rightarrow \frac{\partial f}{\partial y} = x$$

x is treated as a constant

- Remember: The Gradient is the vector of all partial derivatives:

$$\nabla f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right] = [y, x]$$

Gradient Descent – Deriving simple functions

- Another example:

$$g(x, y) = x + y$$

$$\rightarrow \frac{\partial g}{\partial x} = 1 \quad y \text{ is treated as a constant}$$

$$\rightarrow \frac{\partial g}{\partial y} = 1 \quad x \text{ is treated as a constant}$$

$$\rightarrow \nabla g = [1, 1]$$



Gradient Descent – Backpropagation

- We need to derive more complex functions...
- ... but these are just chained simple functions!

→ Use **Backpropagation**...

→ ... which is just recursive application of the **chain rule**

Gradient Descent – Backpropagation

- A slightly more complex function:

$$f(x, y, z) = (x + y)z$$

- This is a chain of two functions:

$$q(x, y) = x + y, \quad p(q, z) = qz$$

- We already know:

$$\frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1; \quad \frac{\partial p}{\partial q} = z, \frac{\partial p}{\partial z} = q$$

- The chain rule tells us:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x} = z \cdot 1$$
$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y} = z \cdot 1$$

$$\frac{\partial f}{\partial z} = (x + y)$$

$$f(x, y, z) = p(q(x, y), z)$$

Gradient Descent – Backpropagation

- The chain rule in general:

$$\text{If } f(x) = p(q(x)) \text{ then } \frac{df}{dx} = \frac{df}{dq} \frac{dq}{dx}$$

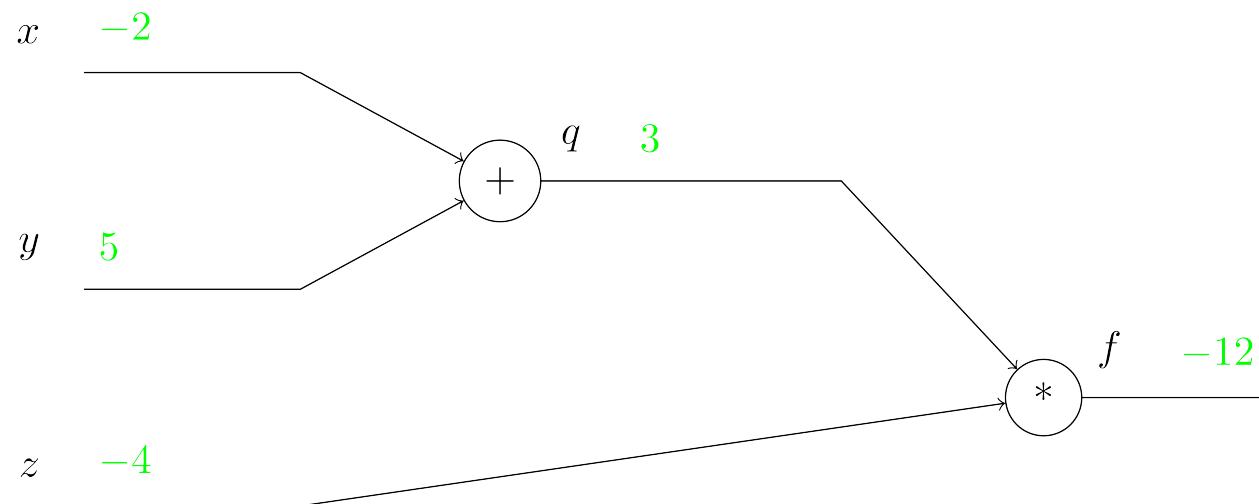
- This is the same thing as before, just the general notation

Gradient Descent – Backpropagation

- A nice visualisation: Circuit Diagrams

$$f(x, y, z) = (x + y)z$$

- Example: $x = -2, y = 5, z = -4$

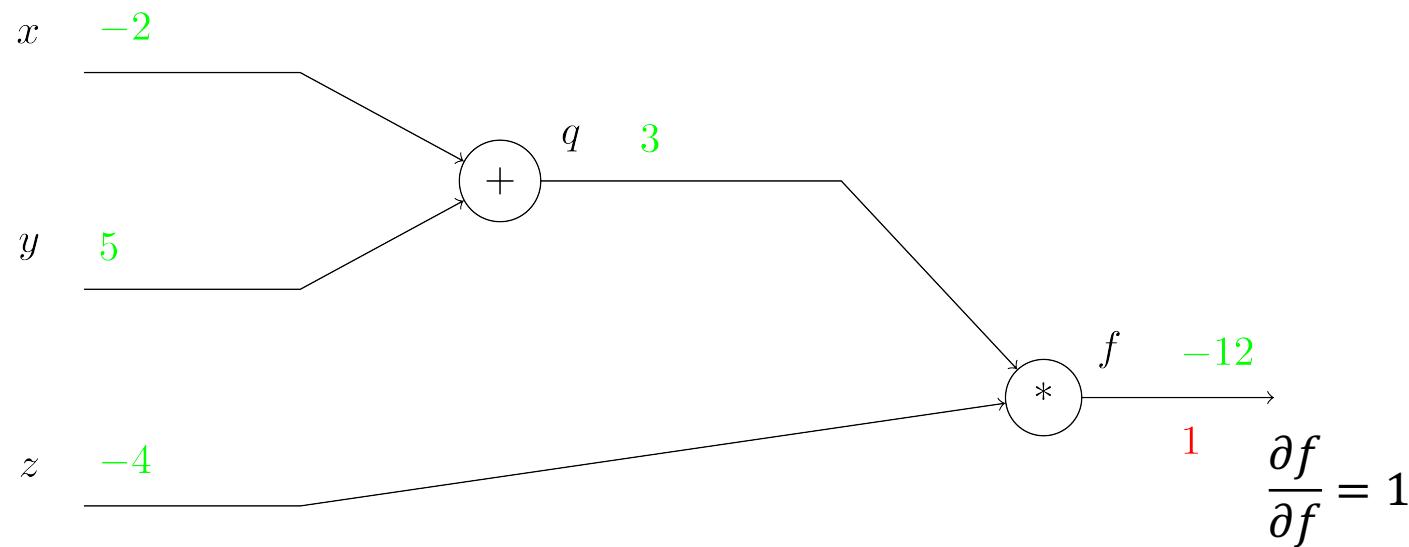


Gradient Descent – Backpropagation

- A nice visualisation: Circuit Diagrams

$$f(x, y, z) = (x + y)z$$

- Example: $x = -2, y = 5, z = -4$

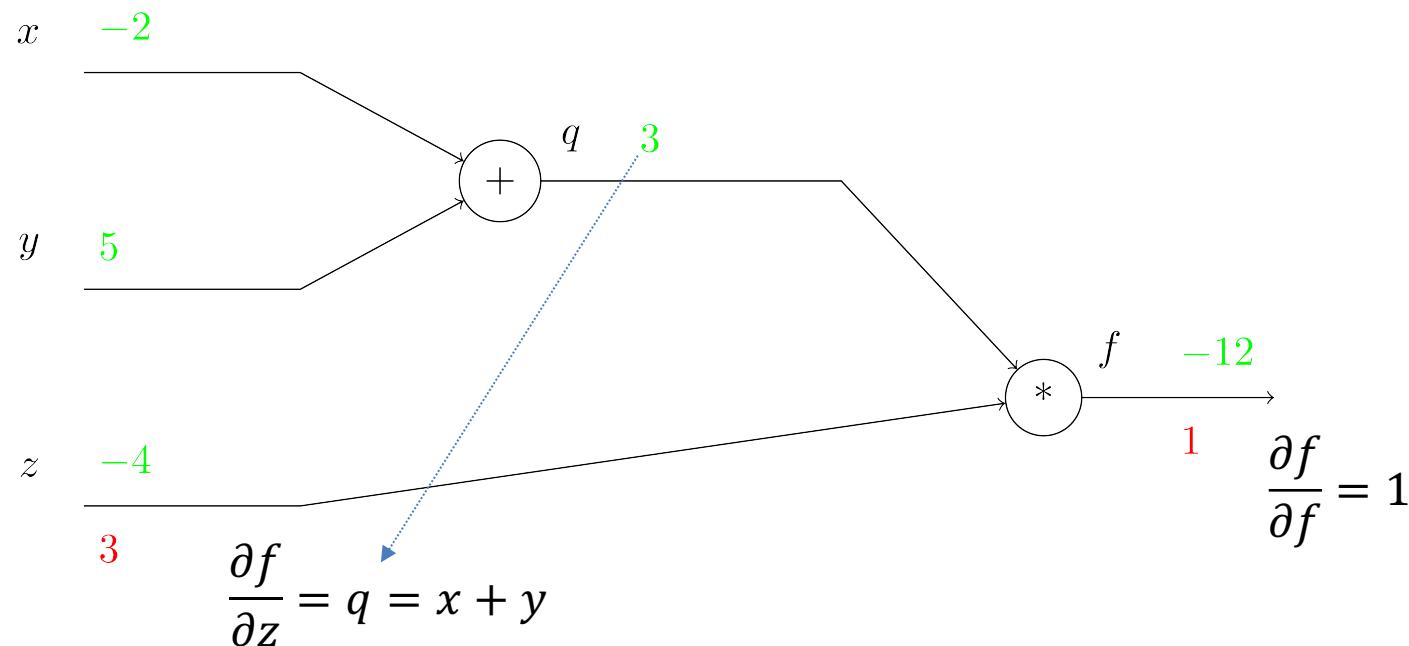


Gradient Descent – Backpropagation

- A nice visualisation: Circuit Diagrams

$$f(x, y, z) = (x + y)z$$

- Example: $x = -2, y = 5, z = -4$

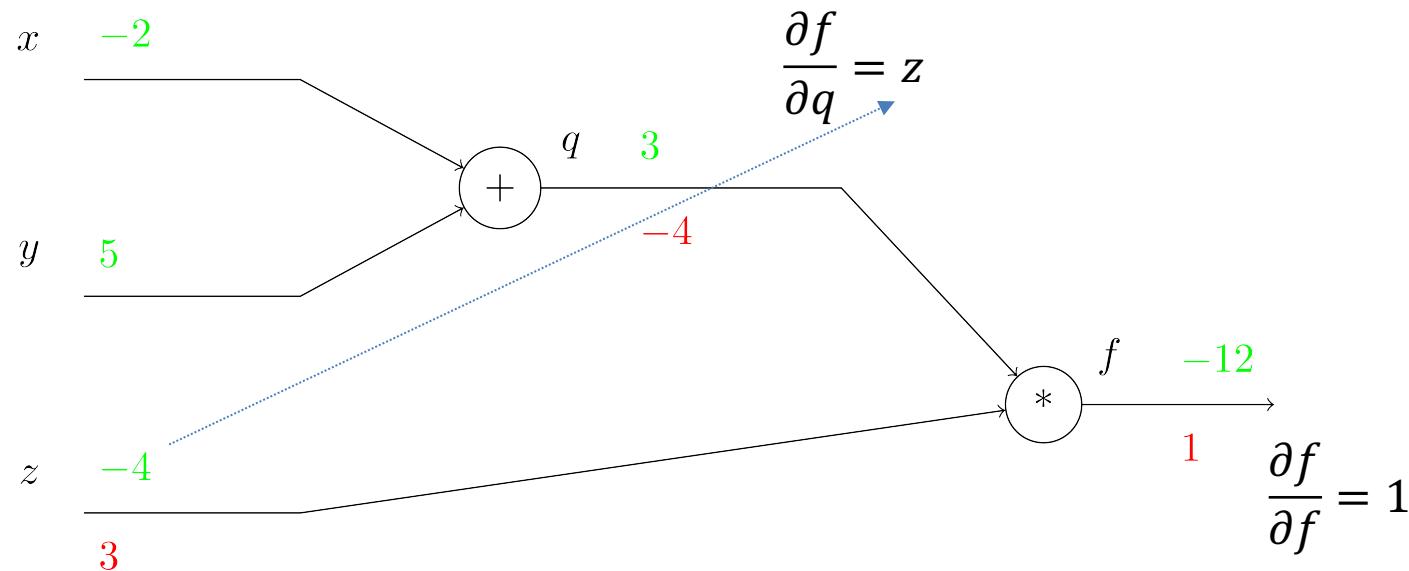


Gradient Descent – Backpropagation

- A nice visualisation: Circuit Diagrams

$$f(x, y, z) = (x + y)z$$

- Example: $x = -2, y = 5, z = -4$

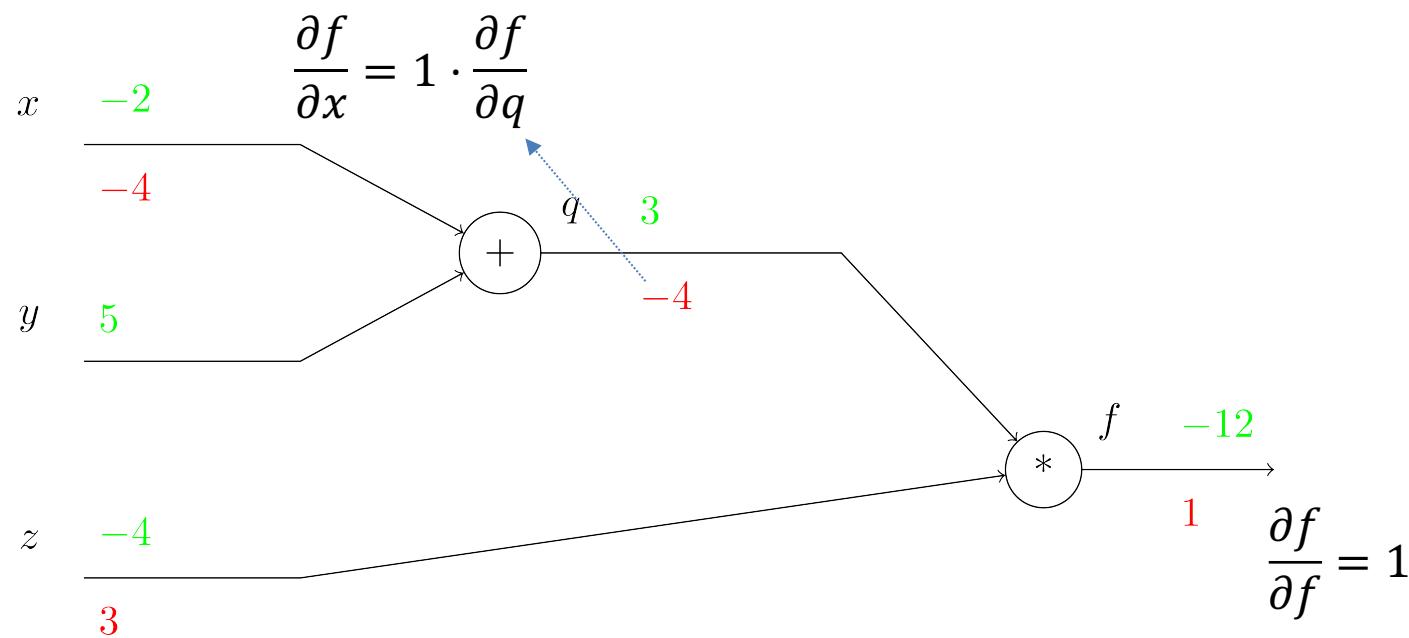


Gradient Descent – Backpropagation

- A nice visualisation: Circuit Diagrams

$$f(x, y, z) = (x + y)z$$

- Example: $x = -2, y = 5, z = -4$

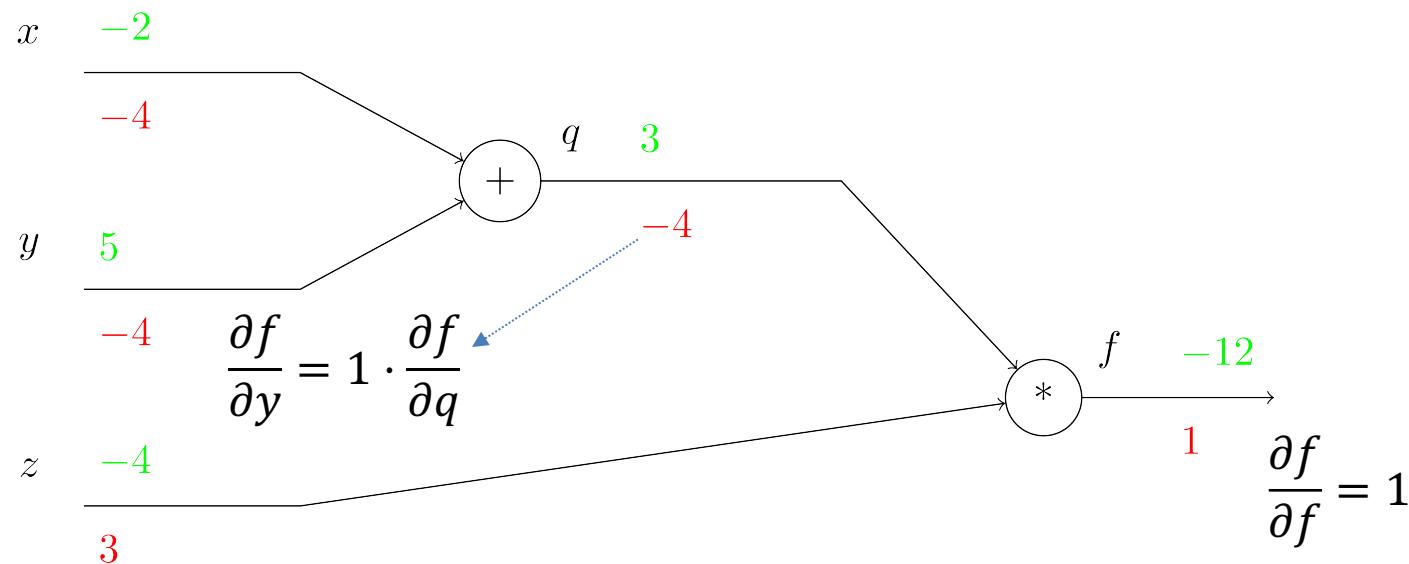


Gradient Descent – Backpropagation

- A nice visualisation: Circuit Diagrams

$$f(x, y, z) = (x + y)z$$

- Example: $x = -2, y = 5, z = -4$

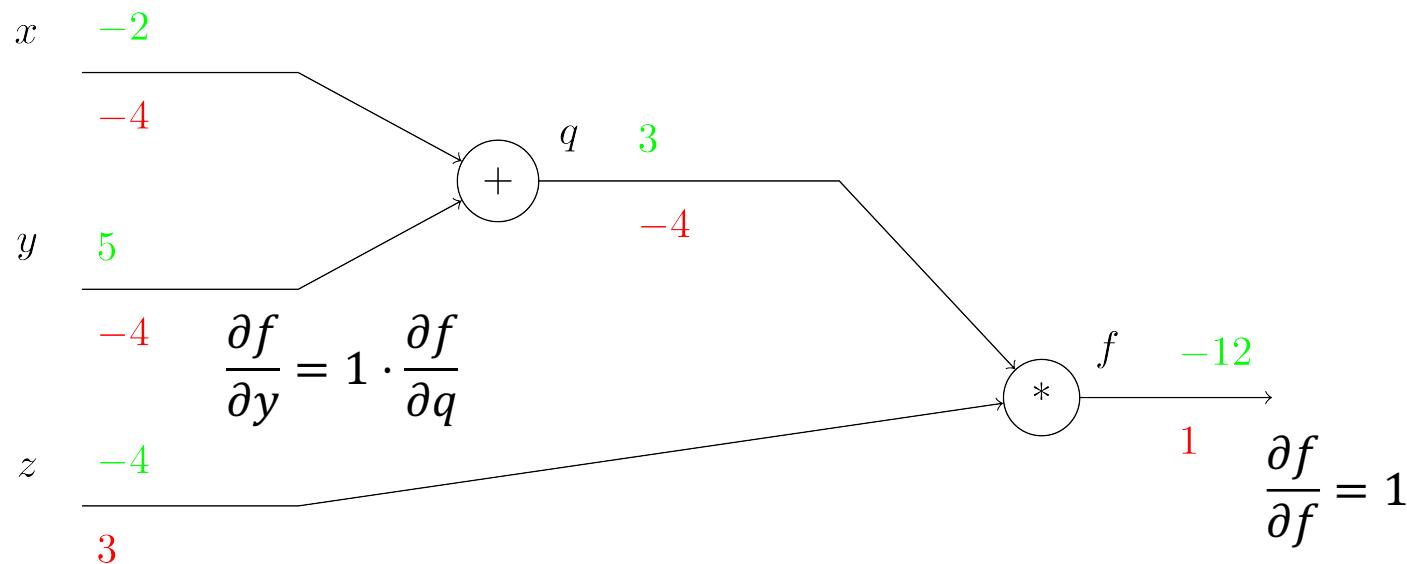


Gradient Descent – Backpropagation

- A nice visualisation: Circuit Diagrams

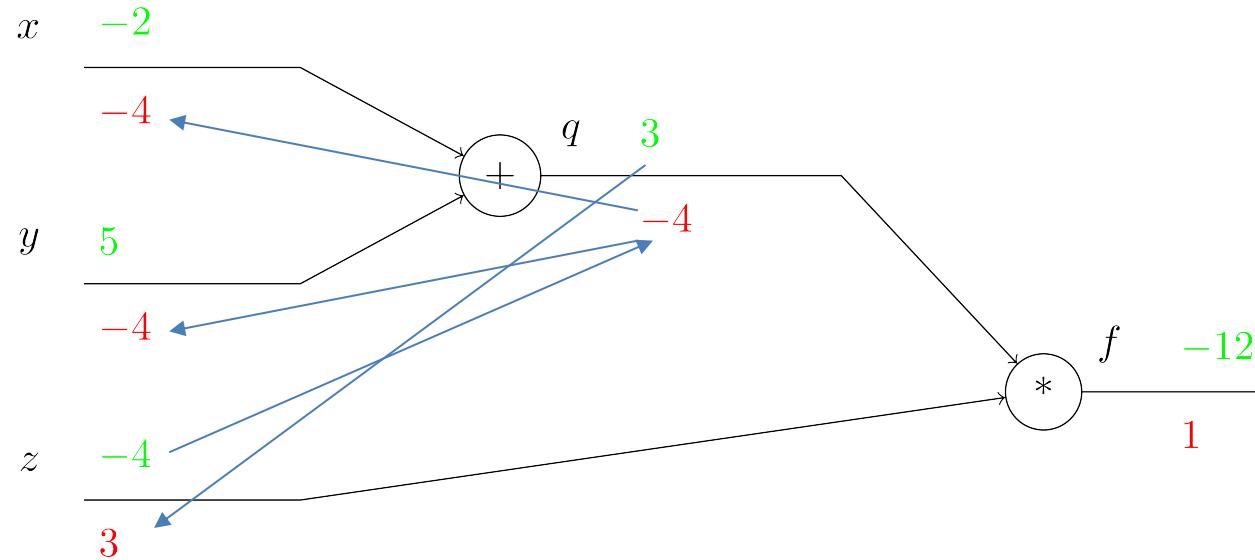
$$f(x, y, z) = (x + y)z$$

- Example: $x = -2, y = 5, z = -4$



Gradient Descent - Backpropagation

- Some gates have „intuitive“ gradient properties:
 - * „switches“ the inputs
 - + just „passes through“
 - More of those in homework ☺

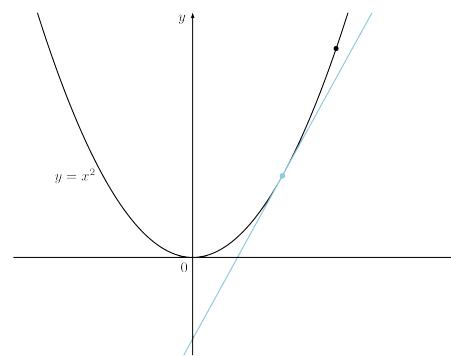


Gradient Descent – The Basics

Gradient Descent

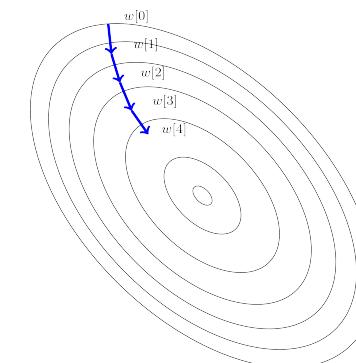
Gradient ✓

= The Vector of all partial derivatives of a function



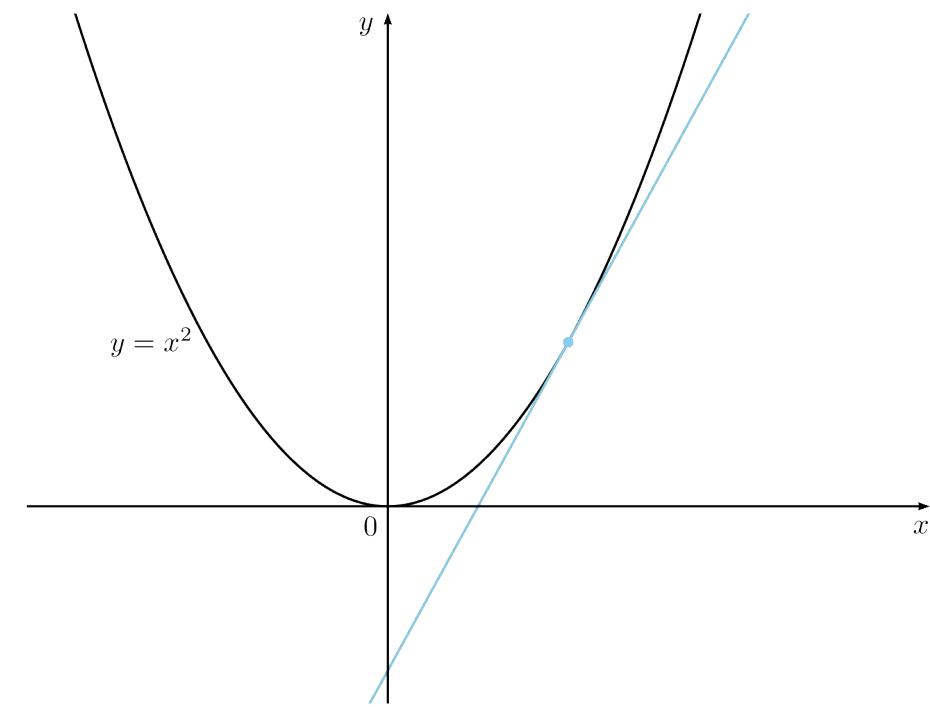
Descent

= Finding a way towards the minimum of the function



Gradient Descent

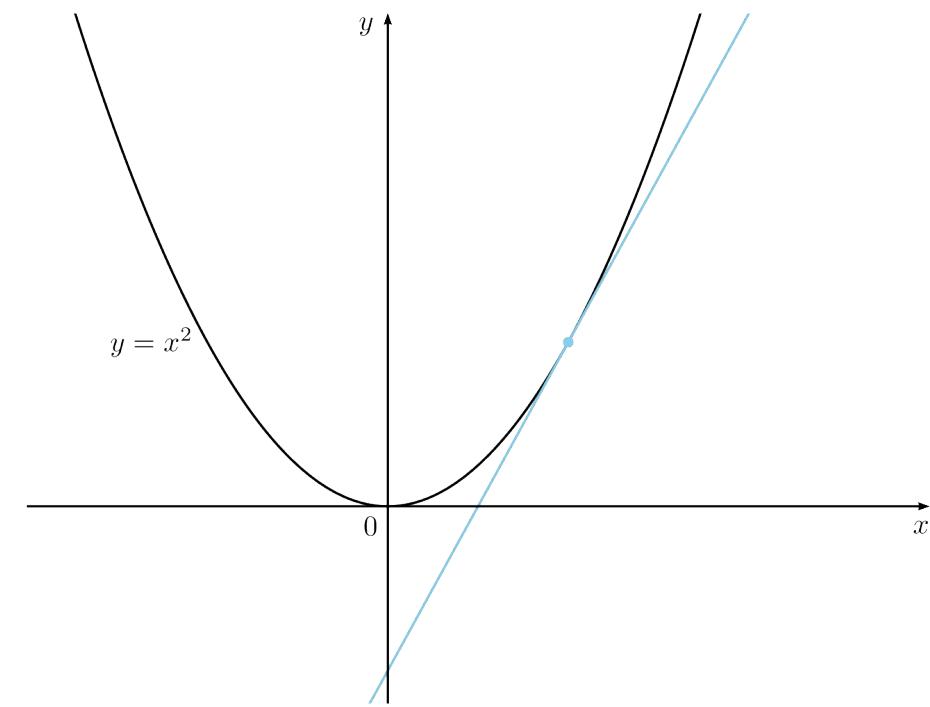
- We know how to get the gradient!
- But what to do with it?
- Take a step along the **inverse** direction of the gradient!
- How far to go?



Gradient Descent

- We know how to get the gradient!
- But what to do with it?
- Take a step along the **inverse** direction of the gradient!
- How far to go?

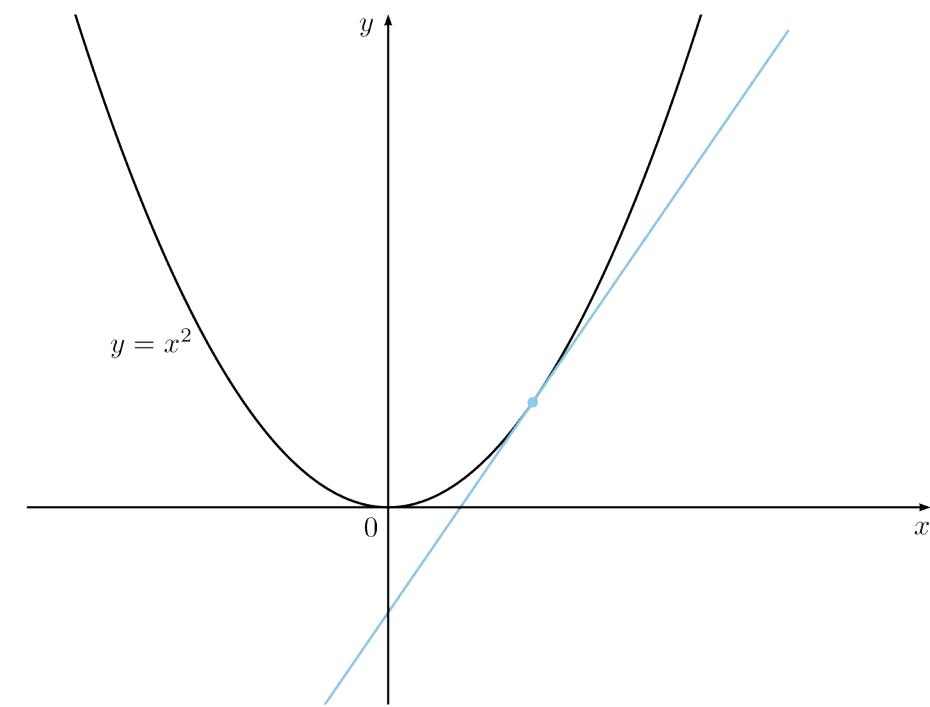
Gradient says to decrease x



Gradient Descent

- We know how to get the gradient!
- But what to do with it?
- Take a step along the **inverse** direction of the gradient!
- How far to go?

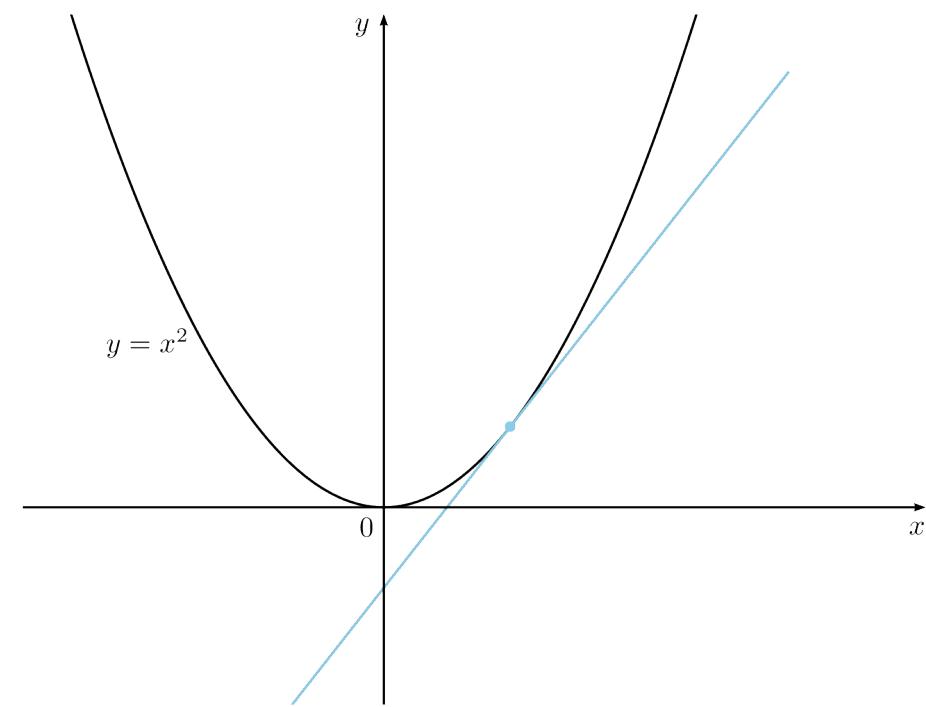
Gradient says to decrease x
Let's take small steps!



Gradient Descent

- We know how to get the gradient!
- But what to do with it?
- Take a step along the **inverse** direction of the gradient!
- How far to go?

Gradient says to decrease x
Let's take small steps!



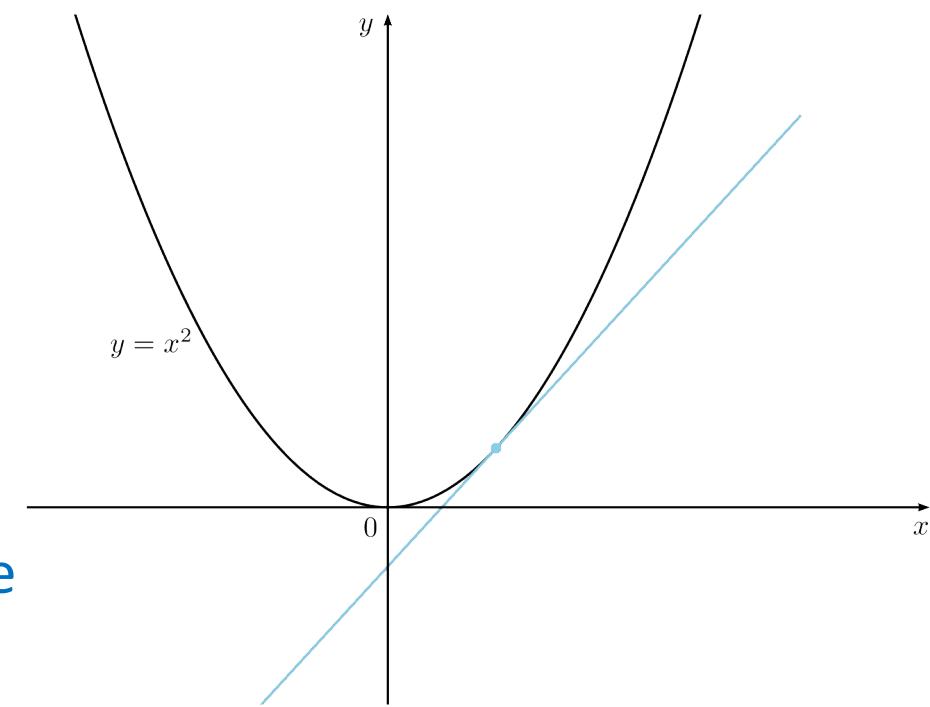
Gradient Descent

- We know how to get the gradient!
- But what to do with it?
- Take a step along the **inverse** direction of the gradient!
- How far to go?

Gradient says to decrease x

Let's take small steps!

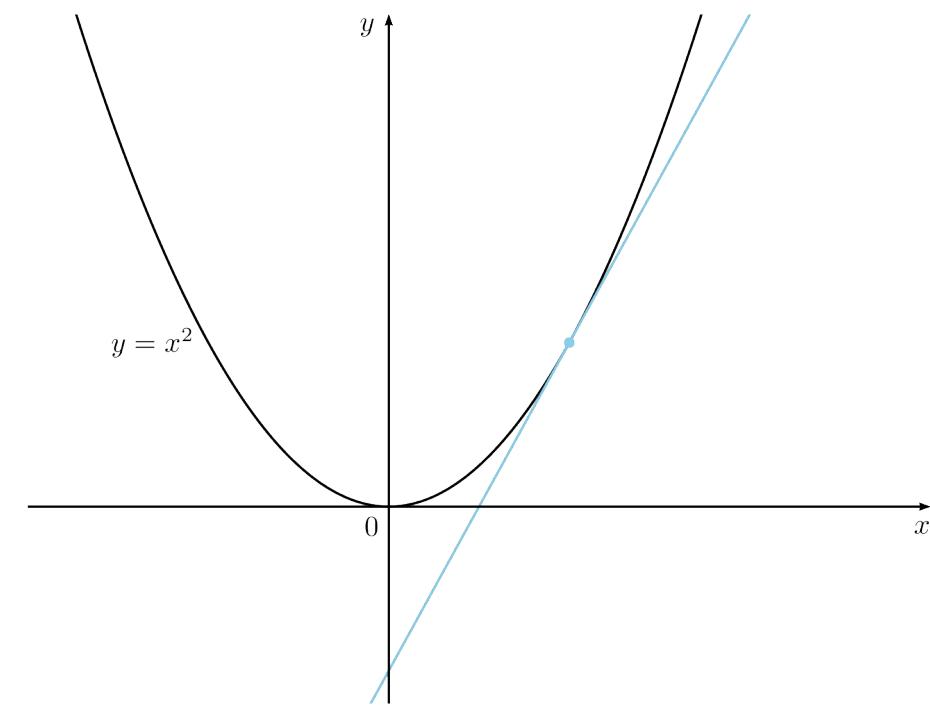
Works, but will take a long time



Gradient Descent

- We know how to get the gradient!
- But what to do with it?
- Take a step along the **inverse** direction of the gradient!
- How far to go?

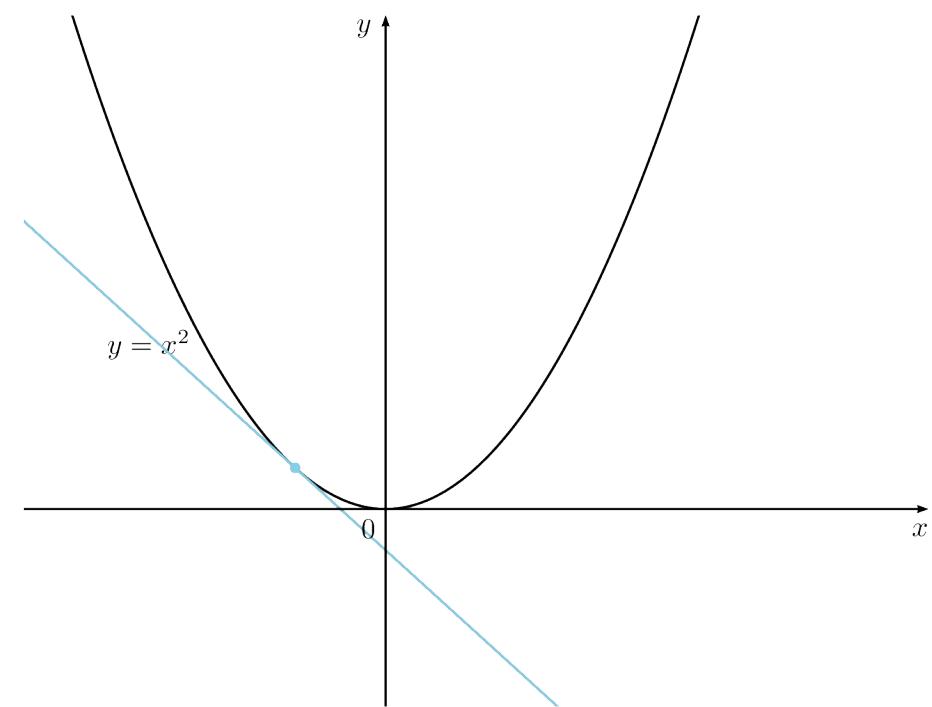
Gradient says to decrease x
Let's take big steps!



Gradient Descent

- We know how to get the gradient!
- But what to do with it?
- Take a step along the **inverse** direction of the gradient!
- How far to go?

Gradient says to decrease x
Let's take big steps!
Too far!

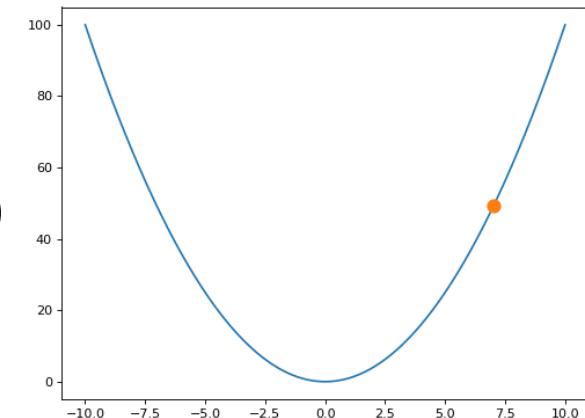


Gradient Descent – The Learning Rate

- Size of the steps = **Learning Rate**
- Second component of (Stochastic) Gradient Descent
- Simplest form of update for an input x :
Vanilla Gradient Descent

$$x += -\text{learningrate} \cdot \frac{df}{dx}(x)$$

- Drawbacks:
 - Somewhat unintuitive parameter learning rate (how to set this?)
 - Very dependent on the learning rate
 - Rather slow convergence in practice



Gradient Descent – Improvements

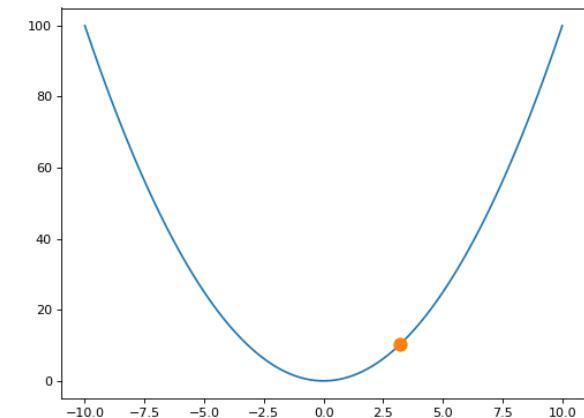
- Momentum
 - Keep part of the velocity v from the last step!
 - So far: A hiker takes steps towards the steepest descent
 - Now: A ball rolls down the hill

$$v = mu \cdot v - \text{learningrate} \cdot \frac{df}{dx}(x)$$

Momentum SGD

$$x += v$$

v: current speed of the ball
mu: how much of he speed to keep



Gradient Descent – Improvements

- Nesterov Momentum
 - Momentum tends to overshoot the minimum → Prevent that!
 - Use a lookahead: Take the gradient at the position that would be reached by the next step with the current velocity

Momentum:

$$v = mu \cdot v - learningrate \cdot \frac{df}{dx}(x)$$

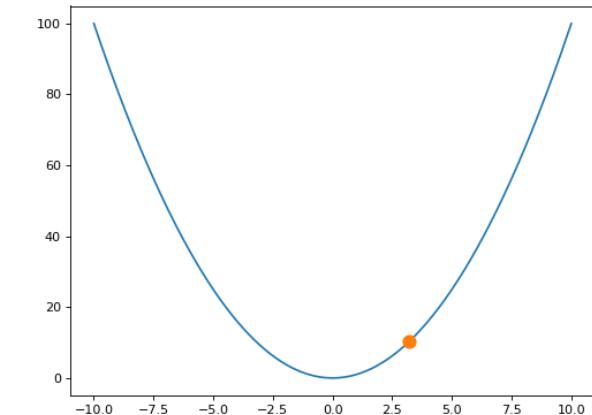
$$xahead = x + mu \cdot v$$

$$v = mu \cdot v - learningrate \cdot \frac{df}{dx}(xahead)$$

$$x += v$$

v: current speed of the ball

mu: how much of he speed to keep





Gradient Descent – Improvements

- All methods so far are very dependent on the learning rate
- Can we get away without tuning the learning rate?

→ Yes! Some methods adapt it themselves*

* Not completely, but mostly

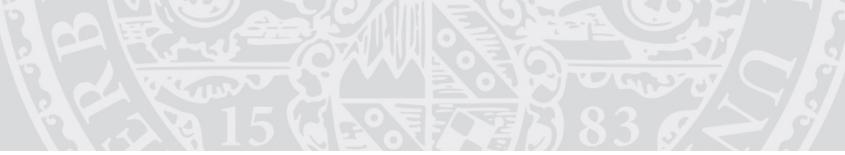
Gradient Descent – Improvements

- Adagrad
 - Introduces a **cache** variable that modifies the effective learning rate **per parameter**

$$\text{cache} += \frac{df}{dx}(x)^2$$

$$x += -\frac{\text{learningrate}}{\sqrt{\text{cache}} + \epsilon} \cdot \frac{df}{dx}(x)$$

- **Decreases** learning rate for parameters that have **large** updates
- **Increases** learning rate for parameters that have **small** updates



Gradient Descent – Improvements

- RMSProp
 - Adagrad usually decreases the learning rate too aggressively. Fix that!

$$\text{cache} = \text{decayrate} \cdot \text{cache} + (1 - \text{decayrate}) \cdot \frac{df}{dx}(x)^2$$

Same as Adagrad

$$x += -\frac{\text{learningrate}}{\sqrt{\text{cache}} + \epsilon} \cdot \frac{df}{dx}(x)$$

- Still tunes the learning rate per parameter...
- ... but does not decrease it monotonically

Gradient Descent – Improvements

- Adam
 - „RMSProp with Momentum“

$$m = \beta_1 \cdot m + (1 - \beta_1) \cdot \frac{df}{dx}(x)$$
$$v = \beta_2 \cdot v + (1 - \beta_2) \cdot \left(\frac{df}{dx}(x)\right)^2$$

$$x += -\frac{\text{learningrate}}{\sqrt{v} + \epsilon} \cdot m$$

- Usually works well
- Also try Nesterov Momentum!

Gradient Descent – Even More Improvements

- RAdam¹
 - „Rectified Adam“ → Introduces a rectifier term

$$m = \beta_1 \cdot m + (1 - \beta_1) \cdot \frac{df}{dx}(x)$$

$$v = \beta_2 \cdot v + (1 - \beta_2) \cdot \frac{df}{dx}(x)^2$$

$$\rho = \rho_\infty - \frac{2t\beta_2}{1 - \beta_2}, \quad \rho_\infty = \frac{2}{\beta_2} - 1$$

$$r = \frac{(\rho - 4)(\rho - 2)\rho_\infty}{(\rho_\infty - 4)(\rho_\infty - 2)\rho}$$

$$x += -\frac{\text{learningrate}}{\sqrt{v} + \epsilon} \cdot m \cdot \boxed{\sqrt{r}}$$

- Rectifies the variance of the adaptive learning rate
 → can be dramatically large in the early stage of training

¹ Liyuan Liu, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Jiawei Han. (2019). [On the Variance of the Adaptive Learning Rate and Beyond.](#)

Gradient Descent – Even More Improvements

- Ranger
 - Combines RAdam and Lookahead¹

Algorithm 1 Lookahead Optimizer:

Require: Initial parameters ϕ_0 , objective function L
Require: Synchronization period k , slow weights step size α , optimizer A

for $t = 1, 2, \dots$ **do**

- Synchronize parameters $\theta_{t,0} \leftarrow \phi_{t-1}$
- for** $i = 1, 2, \dots, k$ **do**

 - sample minibatch of data $d \sim \mathcal{D}$
 - $\theta_{t,i} \leftarrow \theta_{t,i-1} + A(L, \theta_{t,i-1}, d)$

- end for**
- Perform outer update $\phi_t \leftarrow \phi_{t-1} + \alpha(\theta_{t,k} - \phi_{t-1})$

end for

return parameters ϕ

A here is RAdam

- RAdam stabilizes training at the early stage...
- ... Lookahead stabilizes it during the rest

¹ Michael R. Zhang, James Lucas, Geoffrey Hinton, and Jimmy Ba. (2019). *Lookahead Optimizer: k steps forward, 1 step back.*

Gradient Descent

- Improving optimisers is becoming a new field of research
- **Research goal:** Find an optimiser that
 - **converges fast:** use as few optimisation steps as possible to find a minimum
 - **generalises well:** do not step into bad local minima
 - **is easy to use:** no large hyperparameter search to get good performance
- Currently, Adam is the state-of-the-art optimiser [1]
 - it usually converges fast, but
 - it shows bad generalisation performance on some tasks [2]
- New methods are getting more theoretically grounded and show empirical improvements w.r.t. Adam, e.g. MADGRAD [3]
- Time will show the applicability of optimisers in ML fields

[1] Choi, D., Shallue, C. J., Nado, Z., Lee, J., Maddison, C. J., & Dahl, G. E. (2019). On empirical comparisons of optimizers for deep learning.
arXiv preprint arXiv:1910.05446.

[2] Wilson, A. C., Roelofs, R., Stern, M., Srebro, N., & Recht, B. (2017). The marginal value of adaptive gradient methods in machine learning.
arXiv preprint arXiv:1705.08292.

[3] Defazio, A., & Jelassi, S. (2021). Adaptivity without Compromise: A Momentumized, Adaptive, Dual Averaged Gradient Method for Stochastic Optimization.
arXiv preprint arXiv:2101.11075.