Machine Learning and Artificial Neural Networks

Anh Khoa Doan – n.a.k.doan@tudelft.nl

TUDelft

Structure of the Lecture

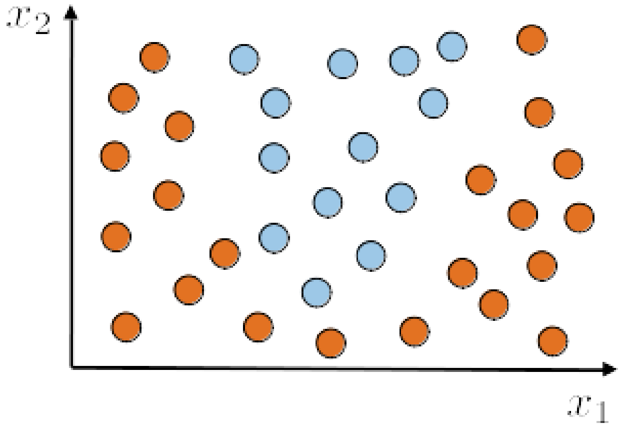
- Introduction to Deep Learning
    1. Motivation
    2. Activation function
    3. Feedforward neural network
    4. Backpropagation algorithm
    5. Training a neural network
- Exercises:

    Introduction to tensorflow/keras, implementation of a neuron

    Implementation of a neural network for regression/classificiation

Motivation

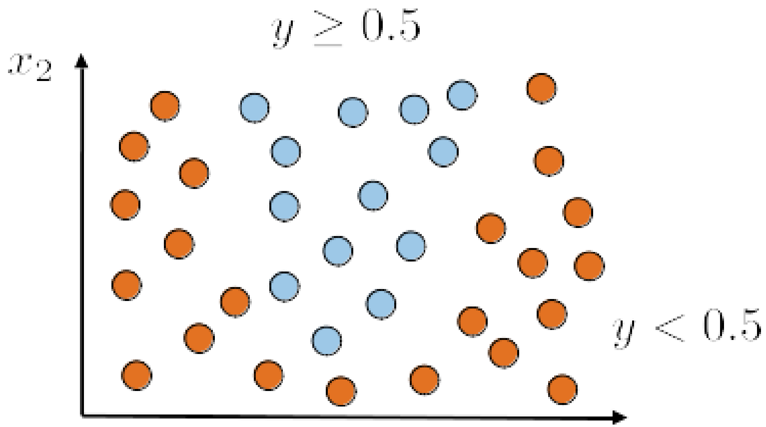
Assume we want to develop a classifier for this dataset

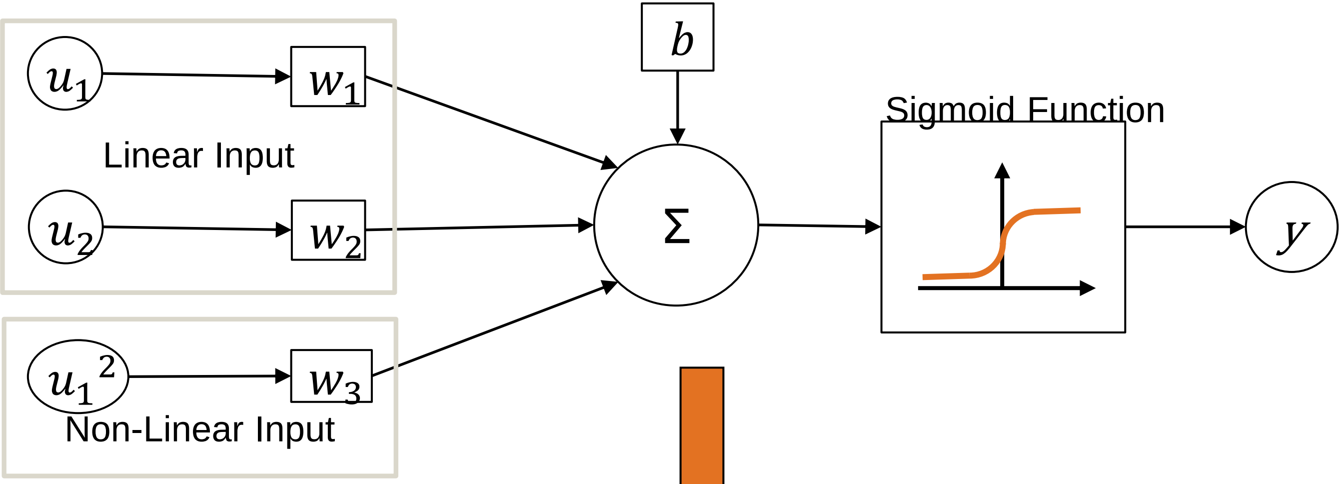


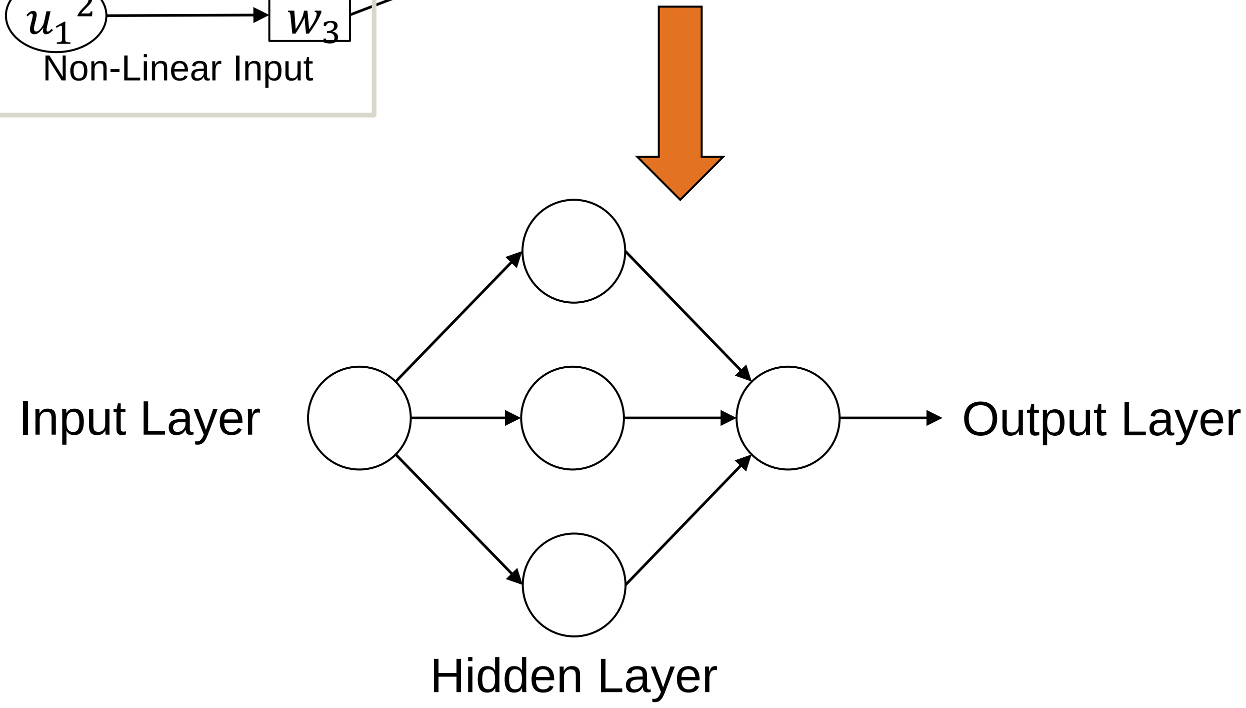
Simple logistic regression insufficient

→ need a transformation of the input

Nonlinear transformation of the input is often required







Aparté on the universal representation theorem

- If we add neurons/layers, more complex functions can be approximated
  - Universal approximator theorem
  - Several demonstrations with more/less limits
- Arbitrary width, bounded depth (Cybenko 1989, Hornik 1991, …)
  - Hornik: "Universal approximator for any bounded, non-constant, continuous activation function"
- Arbitrary depth, bounded width (Zhou et al. 2017, …)

Cybenko, G. (1989). "Approximation by superpositions of a sigmoidal function". *Mathematics of Control, Signals, and Systems*. **2** (4): 303–314.
Hornik, Kurt (1991). "Approximation capabilities of multilayer feedforward networks". *Neural Networks*. **4** (2): 251–257.
Lu, Zhou; Pu, Homgming; Wang, Feicheng; Hu, Zhiqiang; Wang, Liwei (2017). "The Expressive Power of Neural Networks: A View from the Width". *Advances in Neural Information Processing Systems*. Curran Associates. **30**: 6231–6239.

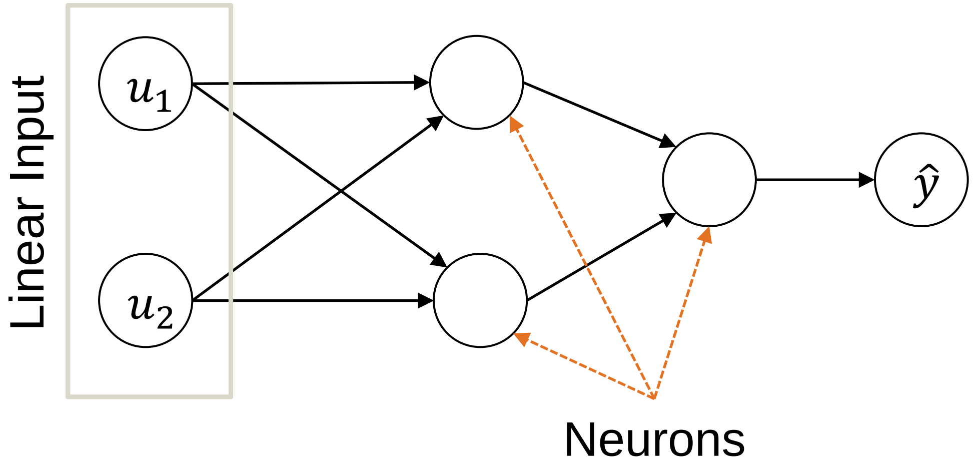
Structure of the Lecture

- Introduction to Deep Learning
    1. Motivation
    2. **Hyperparameters**
    3. Feedforward neural network
    4. Backpropagation algorithm
    5. Training a neural network
- Exercises:

    Introduction to tensorflow/keras, implementation of a neuron

    Implementation of a neural network for regression/classification

Neural network: network of neurons



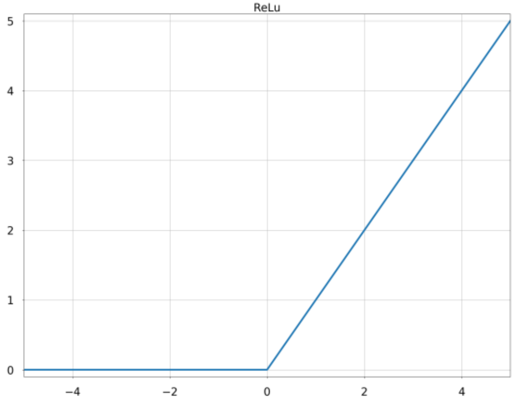
Neural network Hyperparameters

- Number of layers
- Number of neurons
- Activation function
- Loss function

Activation function can take many shape depending on sought properties

Linear rectifier (ReLU)

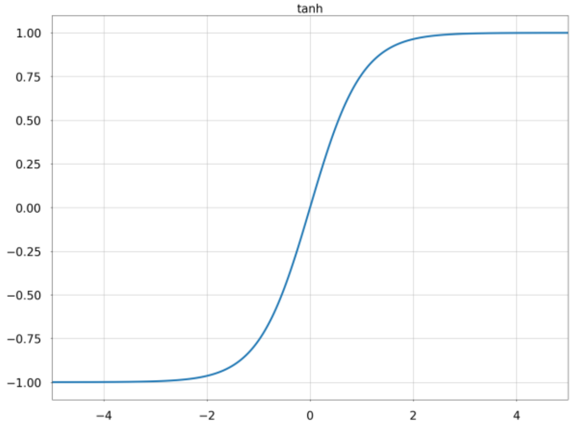
$$f(x) = \begin{cases} x, x > 0 \\ 0, x < 0 \end{cases}$$



$$f'(x) = \begin{cases} 1, x > 0 \\ 0, x < 0 \end{cases}$$

tanh

$$f(x) = \tanh x$$



$$f' = 1 - f^2$$

Softmax: “Generalization of sigmoid for n classes”

$$\boldsymbol{f}_i(x) = \frac{e^{x_i}}{\sum e^{x_i}}$$

Gives a “percentage” representation (smooth version of the argmax function)

Structure of the Lecture

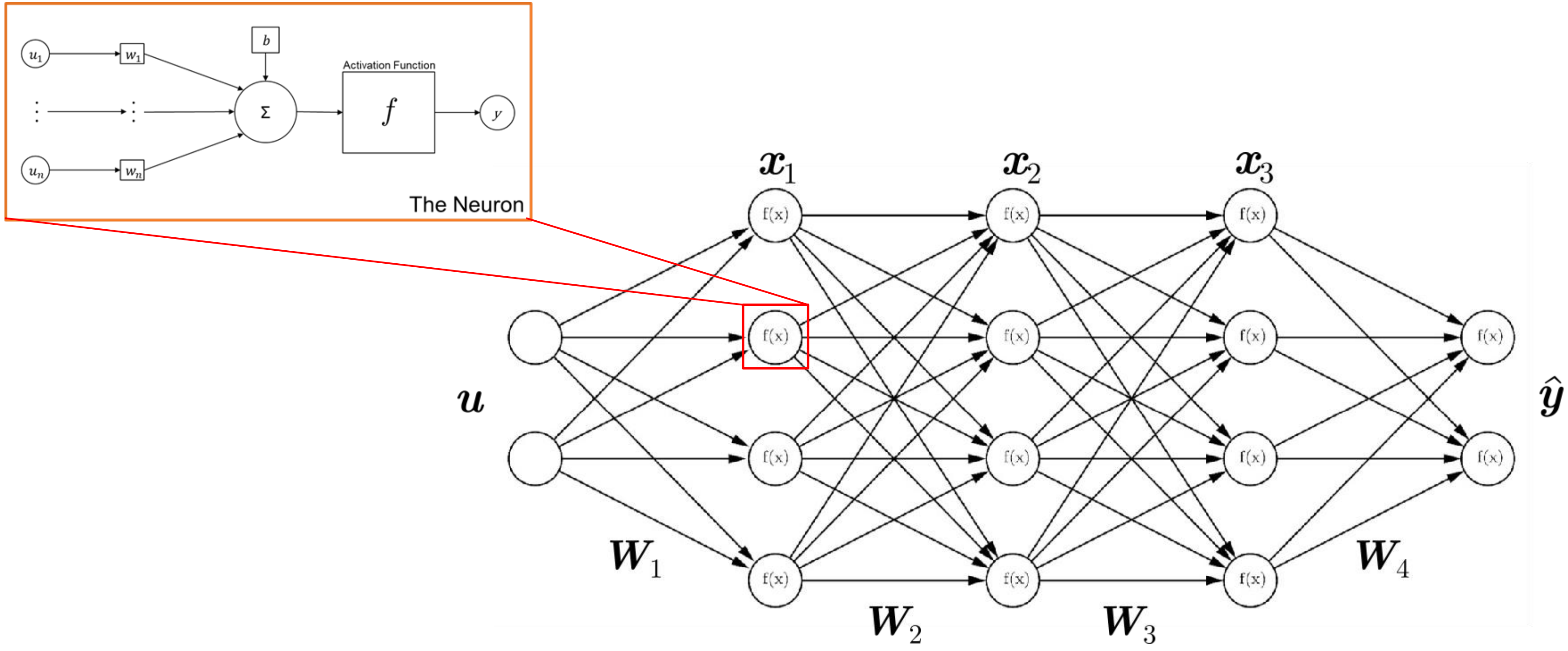
- Introduction to Deep Learning
    1. Motivation
    2. Activation function
    3. **Feedforward neural network**
    4. Backpropagation algorithm
    5. Training a neural network
- Exercises:

    Introduction to tensorflow/keras, implementation of a neuron

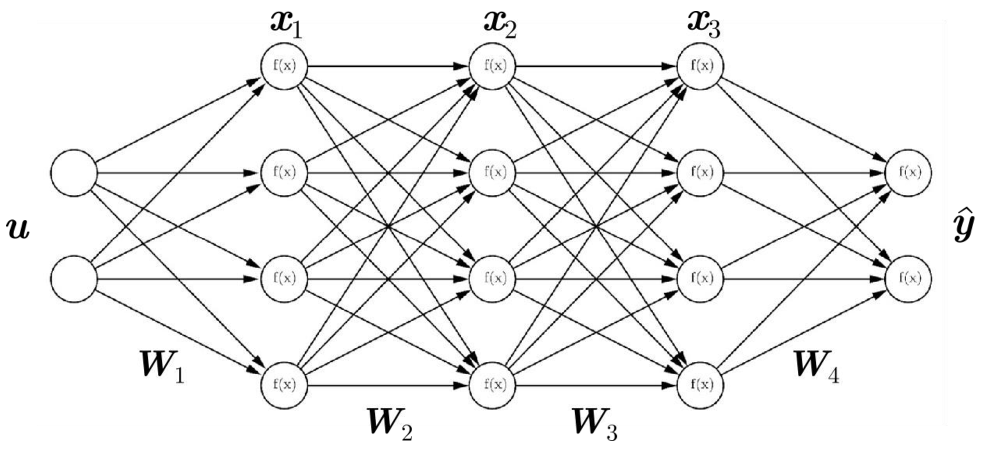
    Implementation of a neural network for regression/classificiation

Feedforward neural network/Multilayer perceptron are obtained by chaining layers of neurons

- Dense deep neural network/multilayer perceptron:
  - Fully connected neurons organised in layers



Feedforward neural network are obtained by chaining layers of neurons



- $x_i = f\left(x_{i-1}^T \cdot W^i + b^i\right)$
- $x_i \in \mathbb{R}^{N_i \times 1}$
- $x_{i-1} \in \mathbb{R}^{N_{i-1} \times 1}$
- $W^i \in \mathbb{R}^{N_i \times N_{i-1}}$
- $b^i \in \mathbb{R}^{N_i \times 1}$
- N_i: number of neurons in i-th layer

- Layers: find *useful* nonlinear transformation of the input (features)
- Depth: # of layers, Width: # of neurons in a layer
- See on-going discussions on respective roles (Nguyen et al. (2021), …)

Nguyen et al. (2021), Do Wide and Deep Networks Learn the Same Things? Uncovering How Neural Network Representations Vary with Width and Depth. https://arxiv.org/abs/2010.15327
Eldan & Shamir (2016). https://arxiv.org/abs/1512.03965

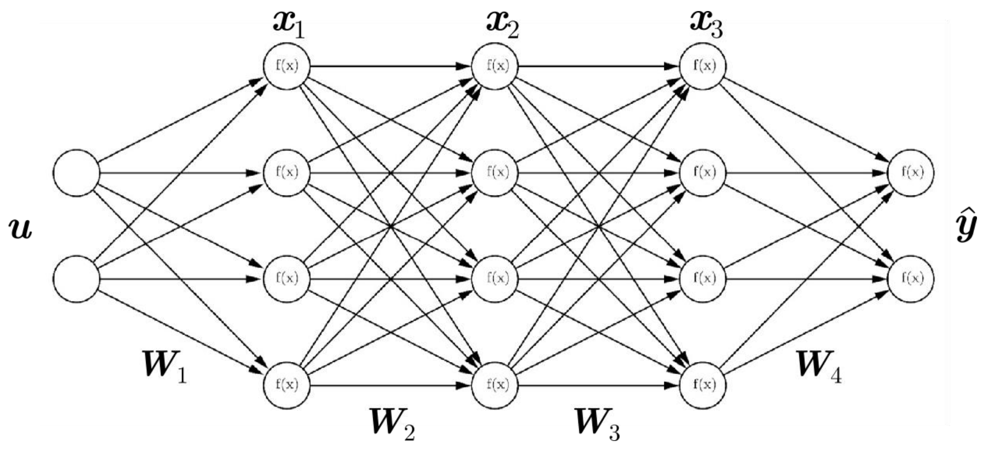
Structure of the Lecture

- Introduction to Deep Learning
  1. Motivation
  2. Activation function
  3. Feedforward neural network
  4. **Backpropagation algorithm**
  5. Training a neural network
- Exercises:

  Introduction to tensorflow/keras, implementation of a neuron

  Implementation of a neural network for regression/classificiation

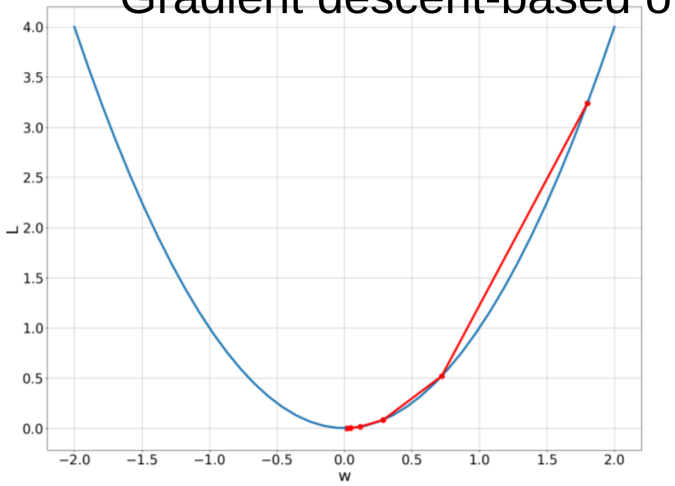
How can we find the “good” network to approximate our function of interest?



Loss function (MSE if supervised learning)

$$L = \sum_i \frac{1}{2} ||\hat{y}_i - y_i||^2$$

Gradient descent-based optimization



$$\boldsymbol{w}_{new} = \boldsymbol{w}_{old} - \alpha \nabla L$$

$$\nabla L = \begin{pmatrix} \frac{\partial L}{\partial w_1} \\ \cdots \\ \frac{\partial L}{\partial w_n} \end{pmatrix}_{\boldsymbol{w}}$$

→ How to get ∇L efficiently?
Backpropagation algorithm

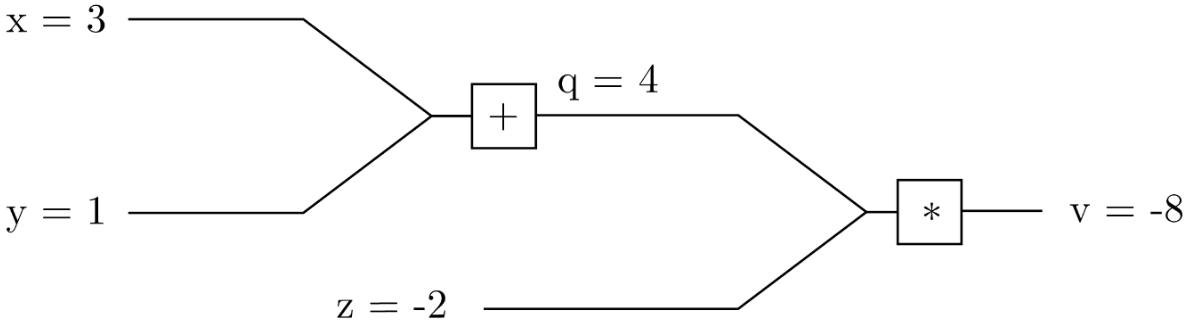
Feedforward neural network and computational graph

We know we need $\Delta w = -\alpha \frac{\partial L}{\partial w}$. How can we get it?

Let's start with the simple example below and compute the derivatives of v using our "graph"

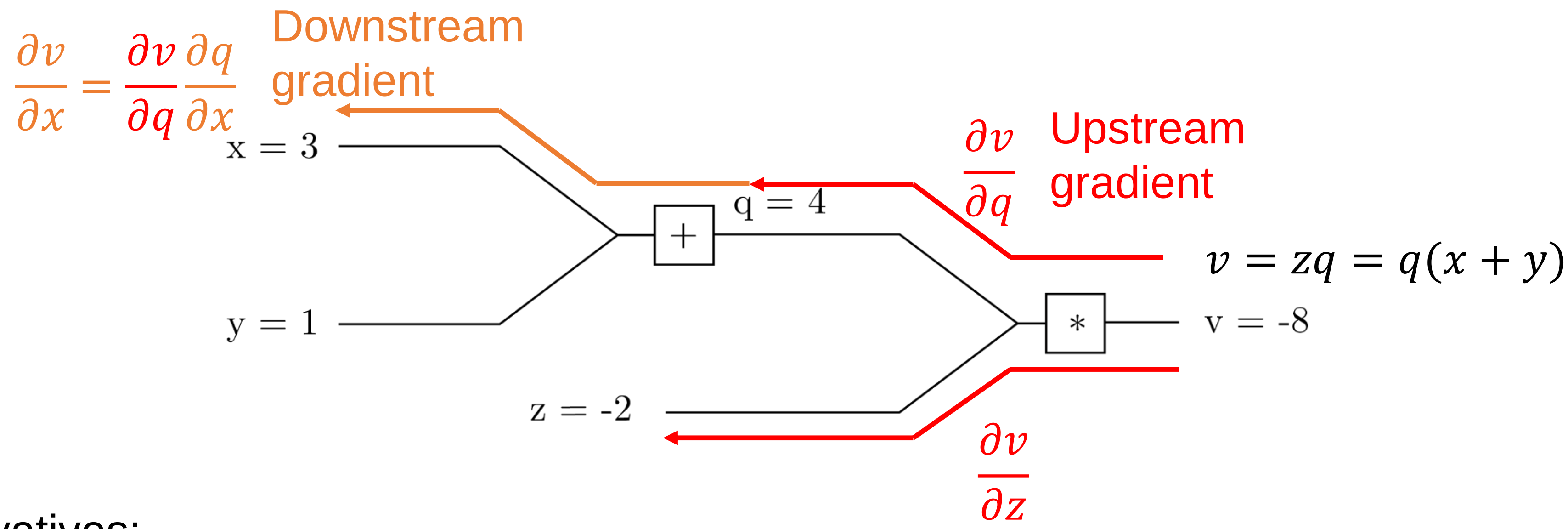
Simple graph:

- Nodes are operations
- Arrows are "data"



Derivatives can be obtained through chain rules

We have the chain of operations → Chain rules of derivatives possible



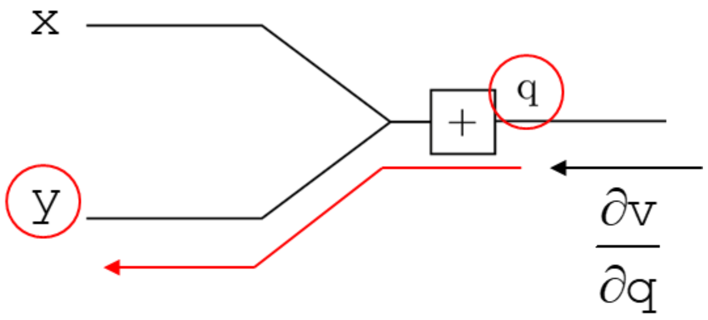
Derivatives:

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

Computation graph and chain derivatives

- Depending on the operation, the direction of the gradient “moving upstream” varies:

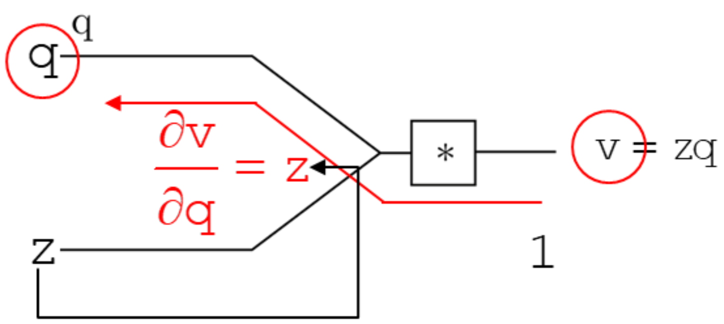
Addition:



$$\frac{\partial v}{\partial y} = \frac{\partial v}{\partial q}\frac{\partial q}{\partial y} = \frac{\partial v}{\partial q}$$

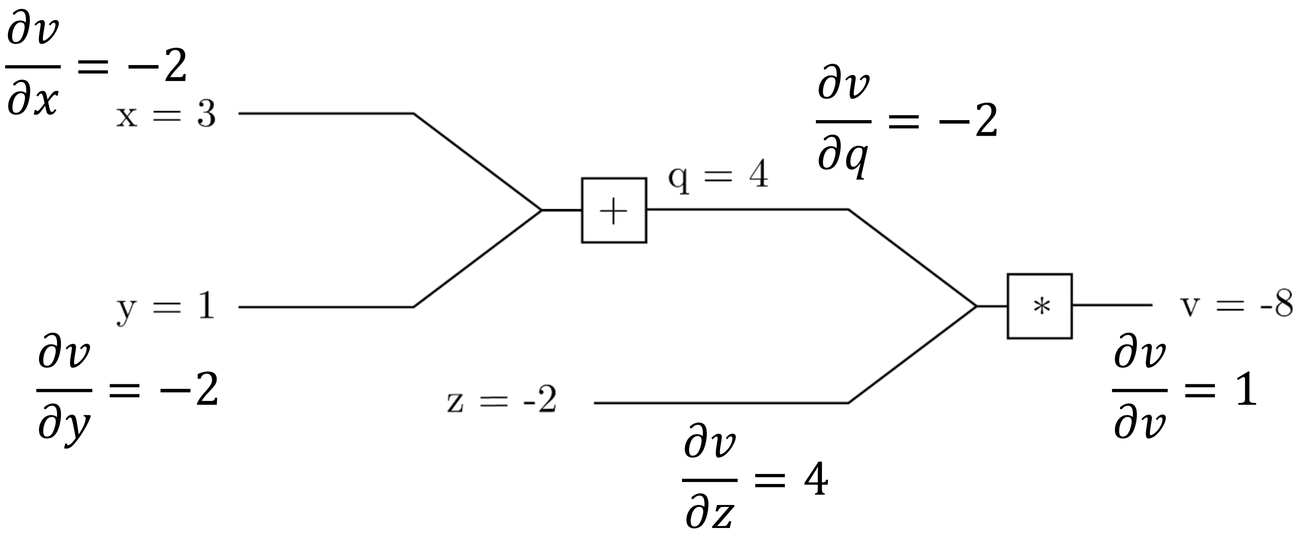
Addition keeps the direction of the gradient

Multiplication



Multiplication switches the direction of the gradient

Computation graph and chain derivatives

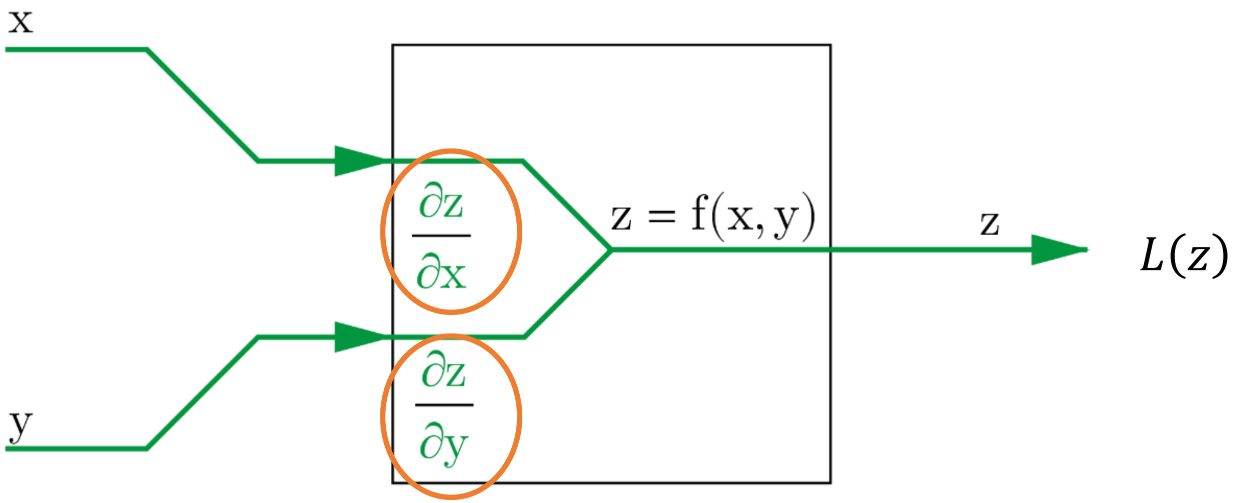


$$\frac{\partial v}{\partial z} = 1 \cdot q = 1 \cdot 4 \qquad \frac{\partial v}{\partial q} = 1 \cdot z = 1 \cdot -2$$

$$\frac{\partial v}{\partial x} = \frac{\partial v}{\partial q}\frac{\partial q}{\partial x} = z \cdot 1 = -2 \cdot 1 \qquad \frac{\partial v}{\partial y} = \frac{\partial v}{\partial q}\frac{\partial q}{\partial y} = z \cdot 1 = -2 \cdot 1$$

Computation graph and chain derivatives with abstract function

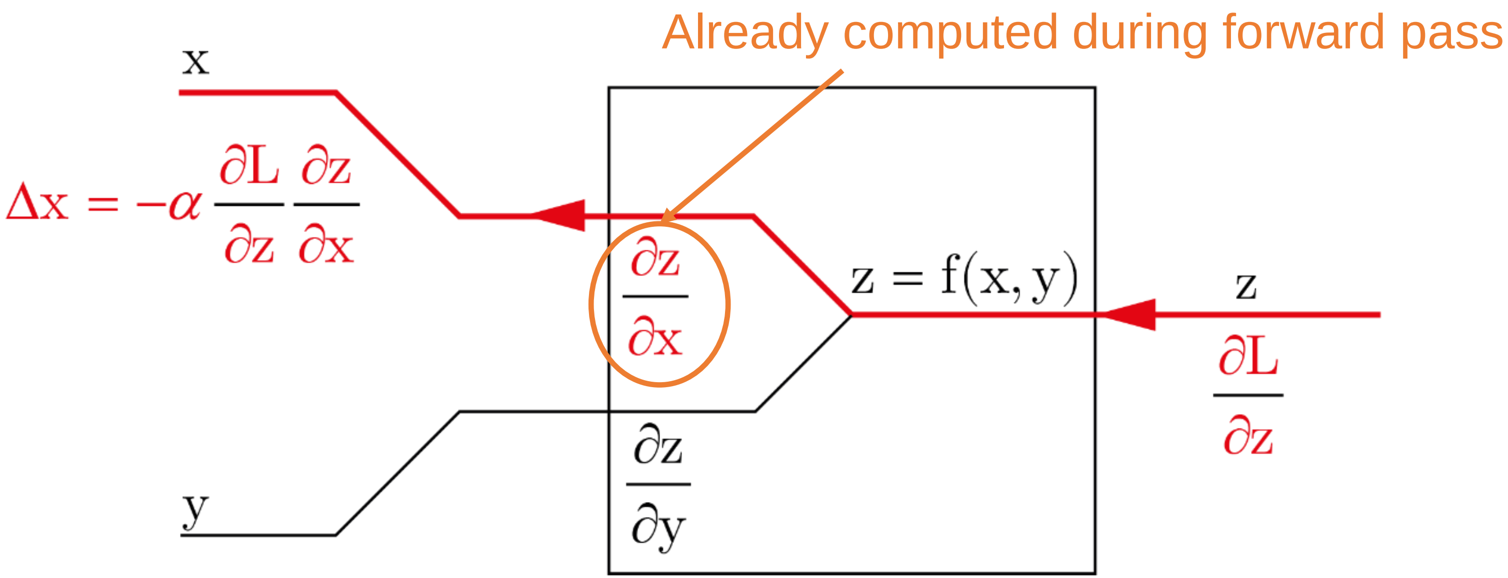
Forward pass



$\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$ can be saved during the forward pass if f' is known

Computation graph and chain derivatives with abstract function

Backward pass



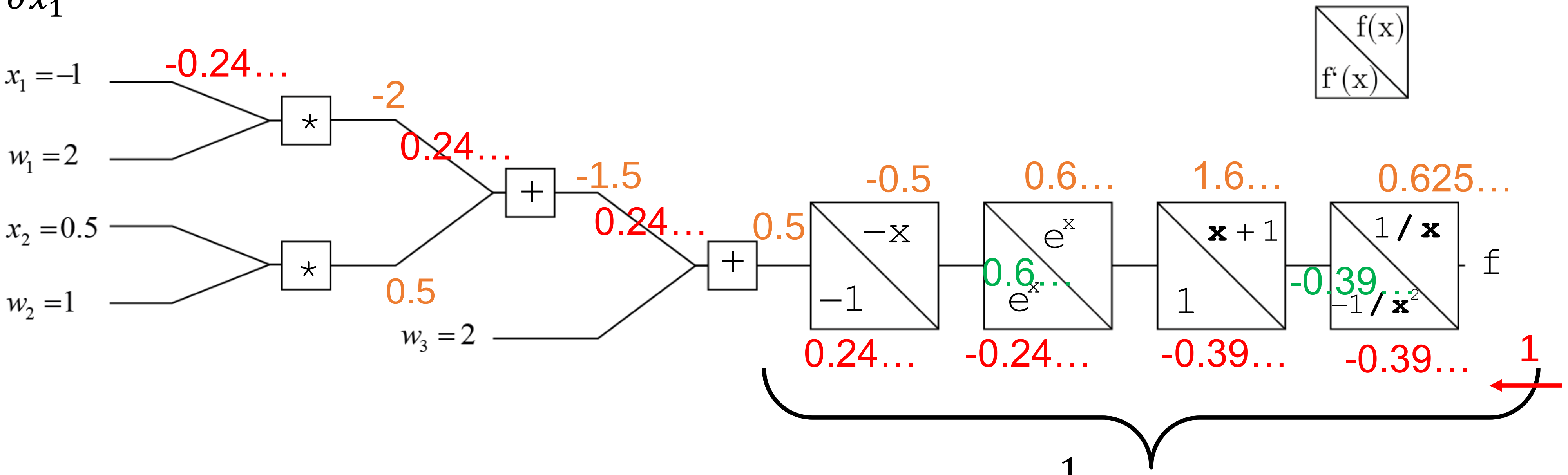
And the process can be chained “indefinitely”

Computation graph and chain derivatives with abstract function

$\frac{\partial f}{\partial w_1} = ?$ -0.24…

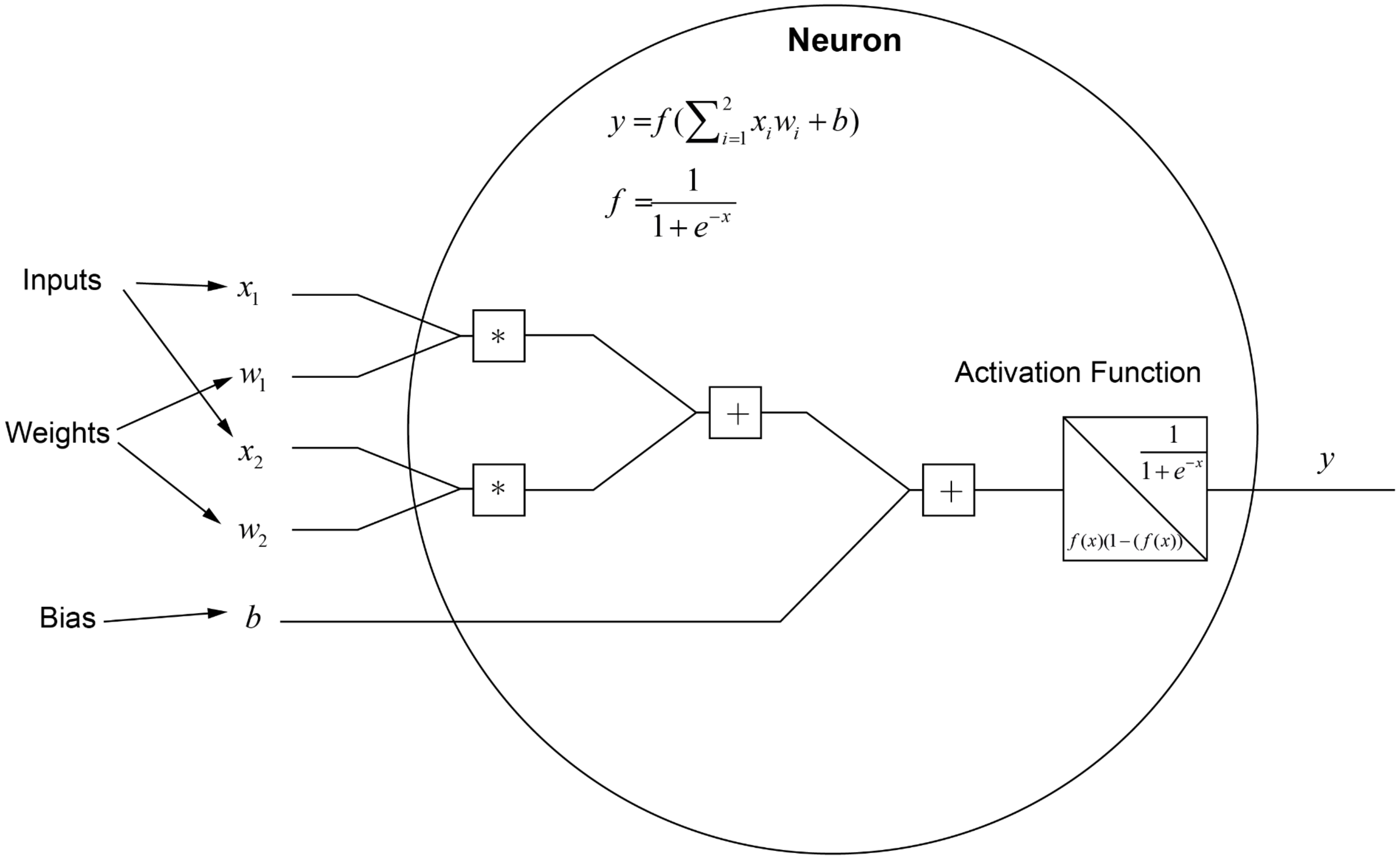
$\frac{\partial f}{\partial x_1} = ?$ 0.48…

$$f(x_1, x_2) = \frac{1}{1 + e^{-(x_1 w_1 + x_2 w_2 + w_3)}}$$

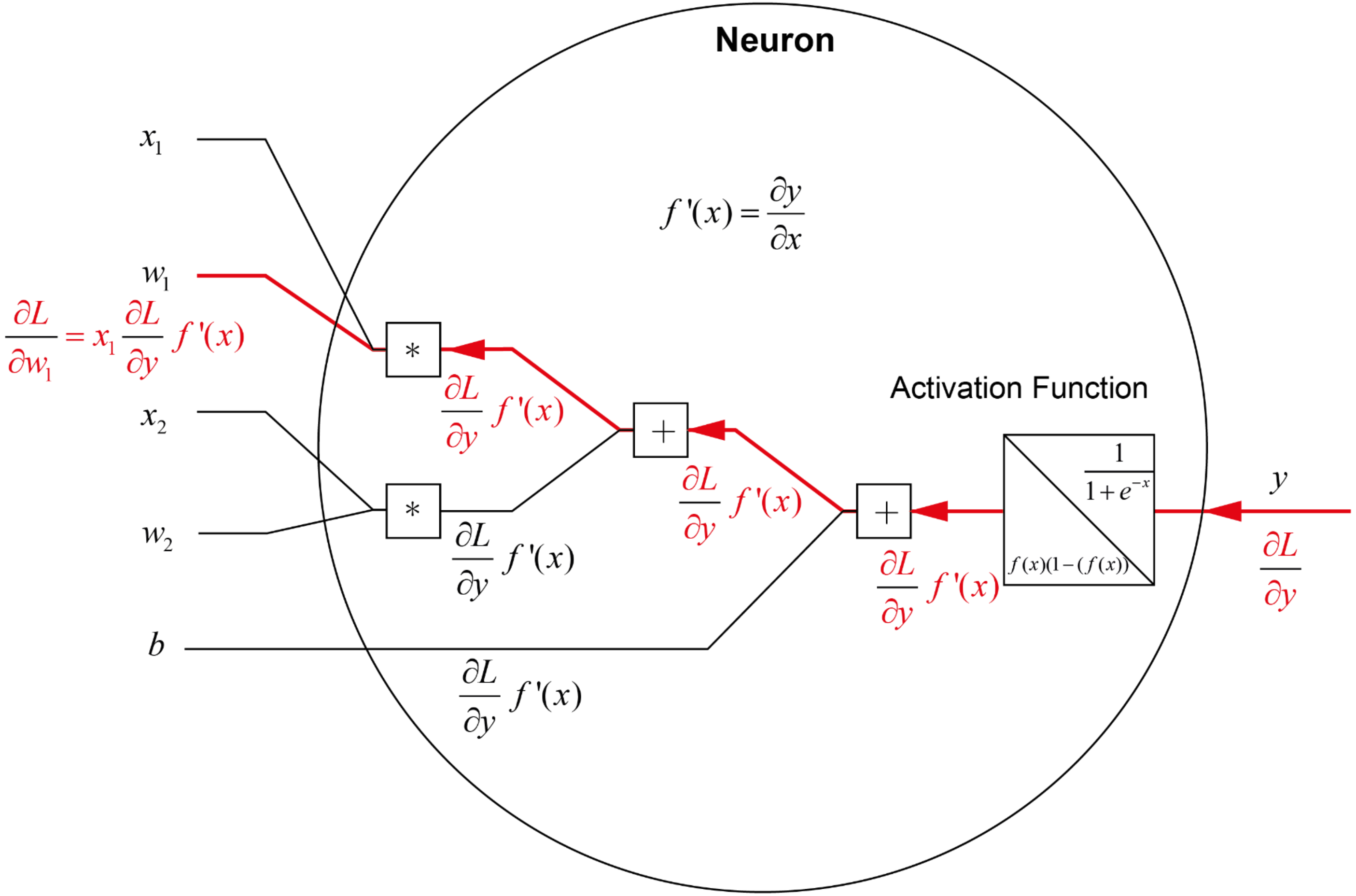


$$f(x) = \frac{1}{1 + e^{-x}}, f'(x) = f(1 - f)$$

The neuron and its derivative

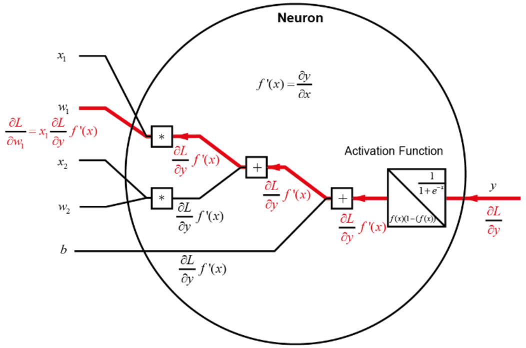


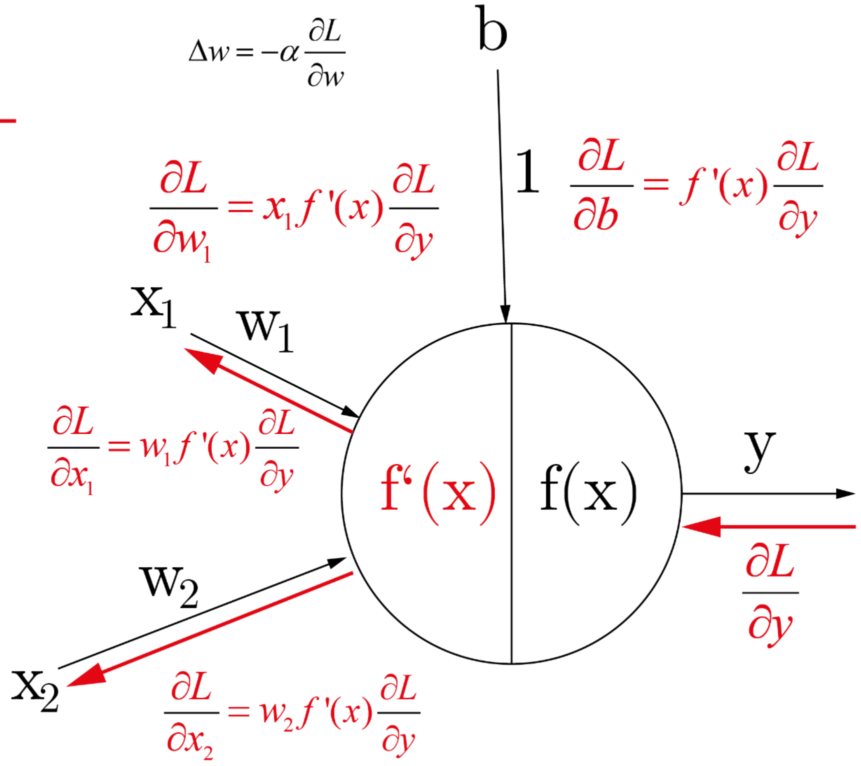
The neuron and its derivative



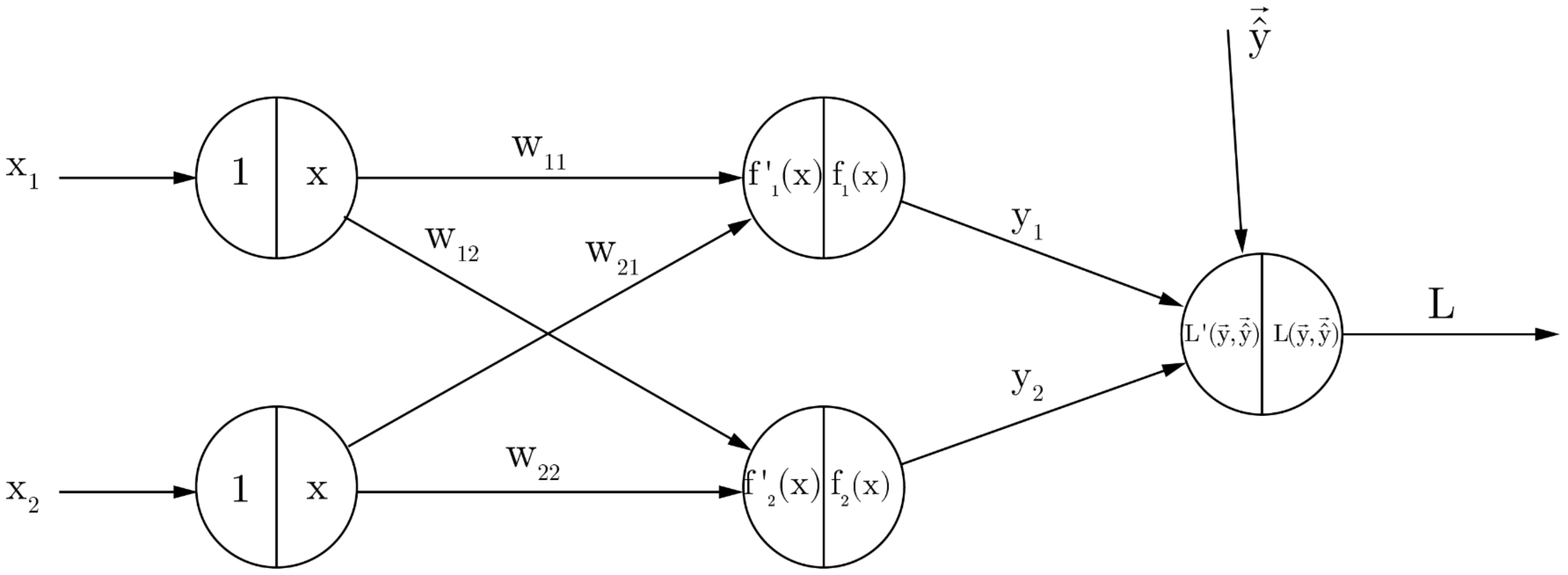
The neuron and its derivative

$$\hat{y} = f(\boldsymbol{w} \cdot \boldsymbol{x} + \boldsymbol{b})$$





Backpropagation

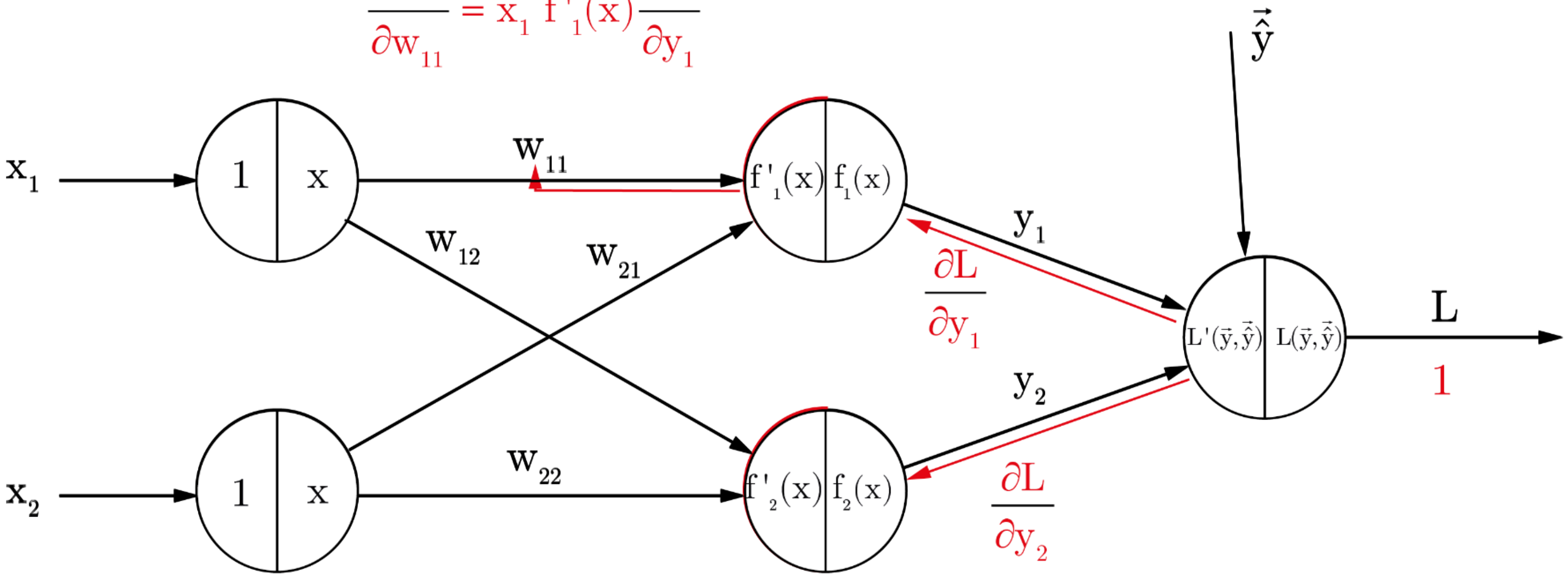


Remember we need ΔW in the gradient descent:

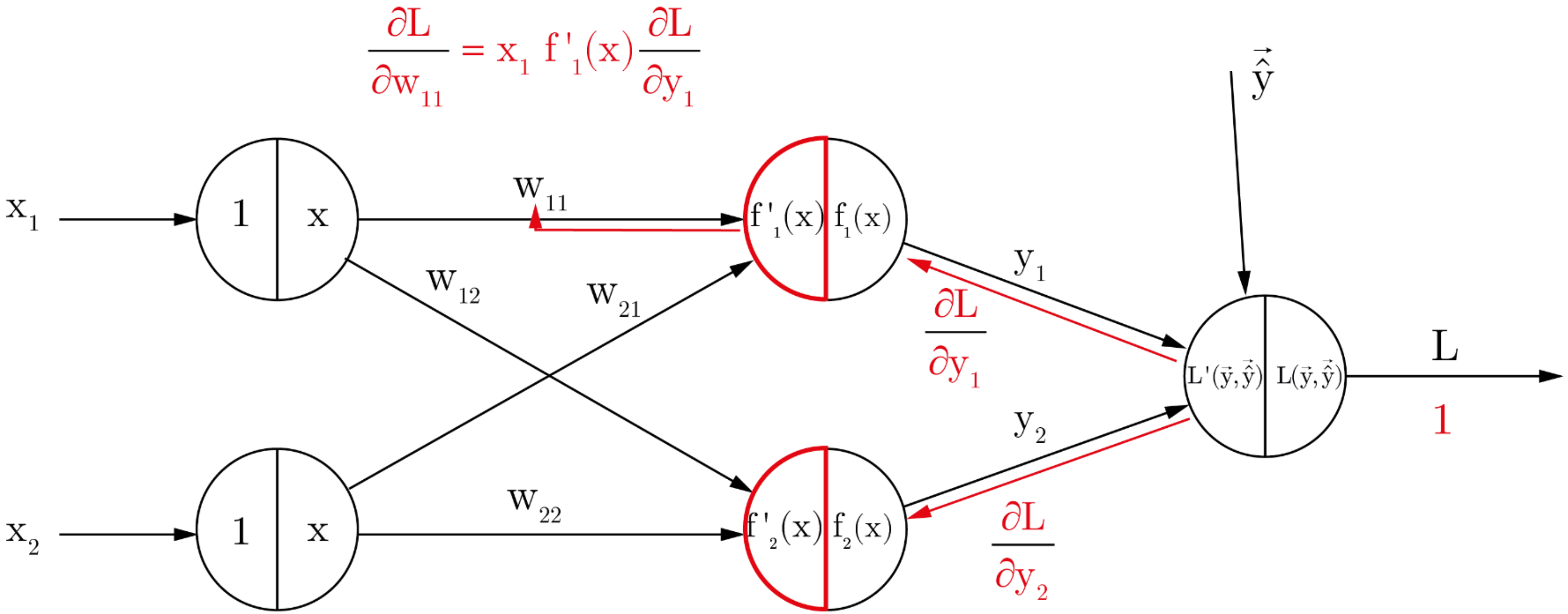
$$\Delta W = -\alpha \begin{pmatrix} \frac{\partial L}{\partial w_{11}} & \frac{\partial L}{\partial w_{12}} \\ \frac{\partial L}{\partial w_{21}} & \frac{\partial L}{\partial w_{22}} \end{pmatrix}$$

Backpropagation

$$\frac{\partial L}{\partial w_{11}} = x_1 \, f'_1(x) \frac{\partial L}{\partial y_1}$$

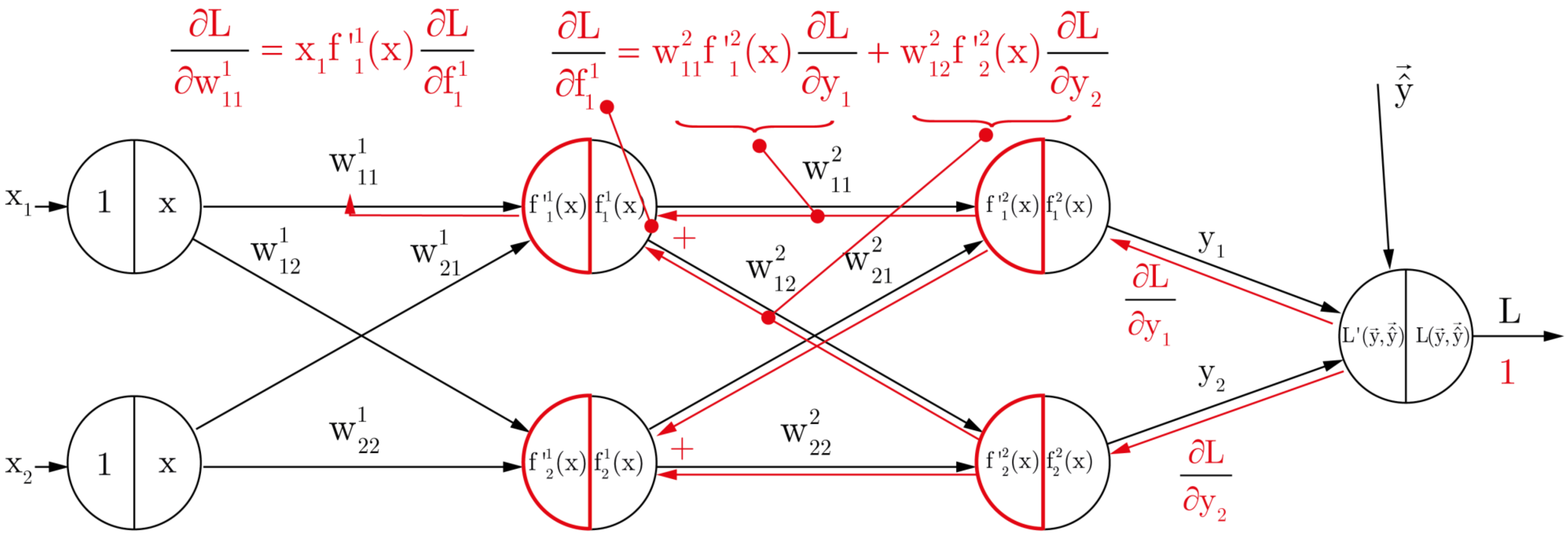


Backpropagation



$$\frac{\partial L}{\partial w_{11}} = x_1\, f'_1(x) \frac{\partial L}{\partial y_1} \qquad \frac{\partial L}{\partial w_{12}} = x_1\, f'_2(x) \frac{\partial L}{\partial y_2} \qquad \frac{\partial L}{\partial w_{22}} = x_2\, f'_2(x) \frac{\partial L}{\partial y_2} \qquad \frac{\partial L}{\partial w_{21}} = x_2\, f'_1(x) \frac{\partial L}{\partial y_1}$$

Backpropagation with a hidden layer



$$\frac{\partial L}{\partial w^1_{11}} = x_1 f'^1_1(x) \frac{\partial L}{\partial f^1_1}, \qquad \frac{\partial L}{\partial w^1_{12}} = x_1 f'^1_2(x) \frac{\partial L}{\partial f^1_2}, \qquad \frac{\partial L}{\partial w^1_{21}} = x_2 f'^1_1(x) \frac{\partial L}{\partial f^1_1}, \qquad \frac{\partial L}{\partial w^1_{22}} = x_2 f'^1_2(x) \frac{\partial L}{\partial f^1_2}$$

About the loss functions in regression problem

- Loss function determines how the network is trained.
- For regression
  - $L2$ or $L1$ error used. $L2$ preferred for smoother gradient

- For classification
  - Binary cross entropy loss
  - Categorical cross entropy loss

$$H_q = -\sum_{c=1}^{M} y_{o,c} \log(p_{o,c})$$

Note on the Categorical Entropy Loss

- In classification problems, the cross entropy combines the error on the prediction and the probability associated to that prediction within a loss function.

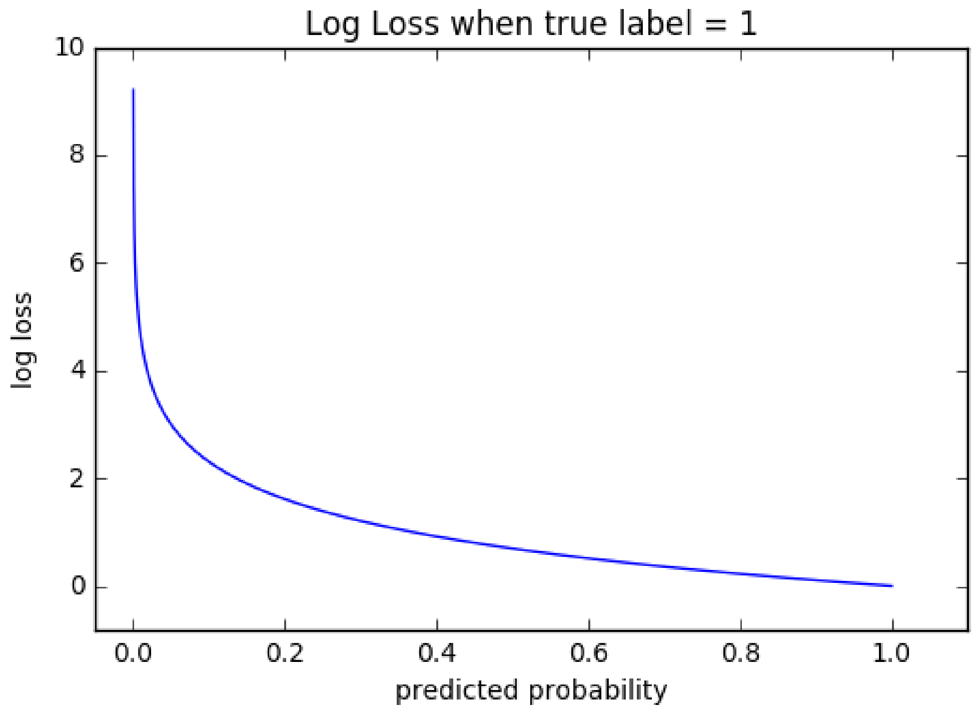
Mathematically, this is:

$$-\sum_{c=1}^{M} y_{o,c}\log\left(p_{o,c}\right)$$

M: number of classes

y: binary indicator (0 or 1) if the class c is the correct classification for the sample o

$p_{o,c}$: predicted probability that sample o is of class c



Structure of the Lecture

- Introduction to Deep Learning
    1. Motivation
    2. Activation function and loss function
    3. Feedforward neural network
    4. Backpropagation algorithm
    5. **Training a neural network**
- Exercises:

    Introduction to tensorflow/keras, implementation of a neuron

    Implementation of a neural network for regression/classificiation

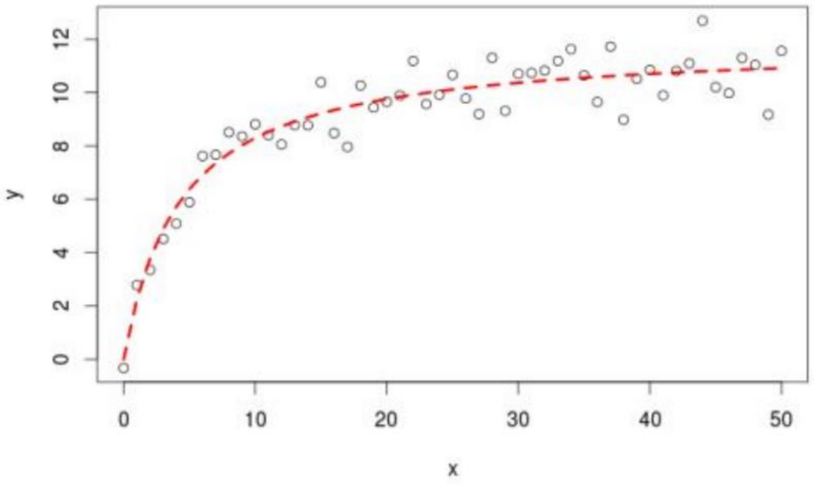
Strategizing the training of a neural network is important

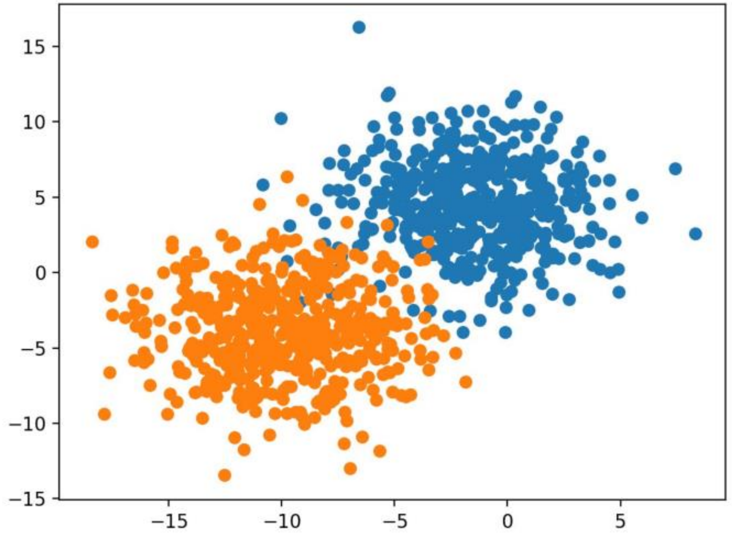
- Objective: get the best model as efficiently as possible
- Big data is not always the solution (or possible)
- How to spend the effort in the right direction
- Approaches
  - Collect more data
  - Diversify the available data
  - Hyperparameter tuning
  - Change the algorithm
  - Try regularization techniques
  - Try bigger/smaller architectures
  - Change the architecture

Training a neural network
1. Data visualization

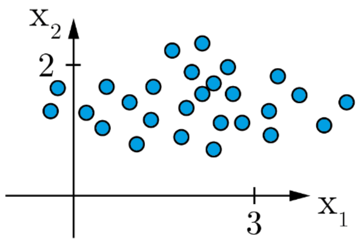
- (if possible) Always start by visualizing the data
  - Helps with spotting trends/outliers/peaks/…
  - Provides insights into pre-processing needed
  - Tools:

Histogram, scatter plot, box plot, violin plot, …

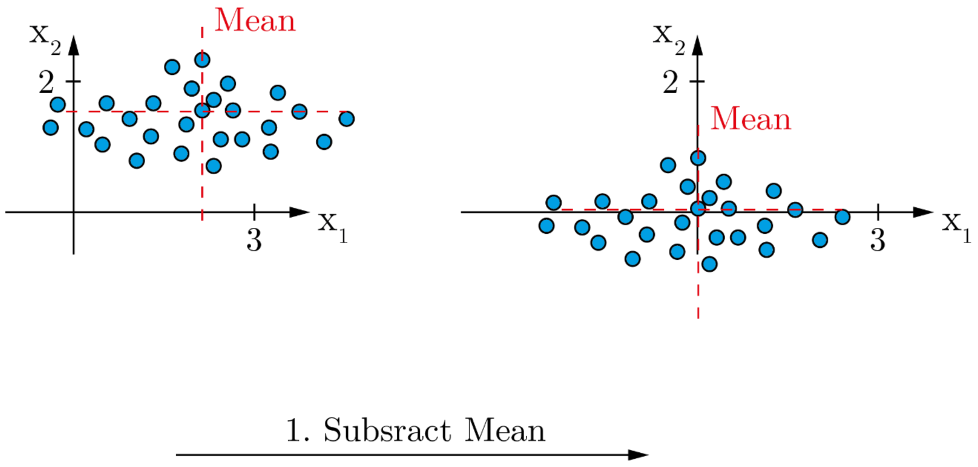




Training a neural network
2. Check Data Distribution of the Input Data: normalization



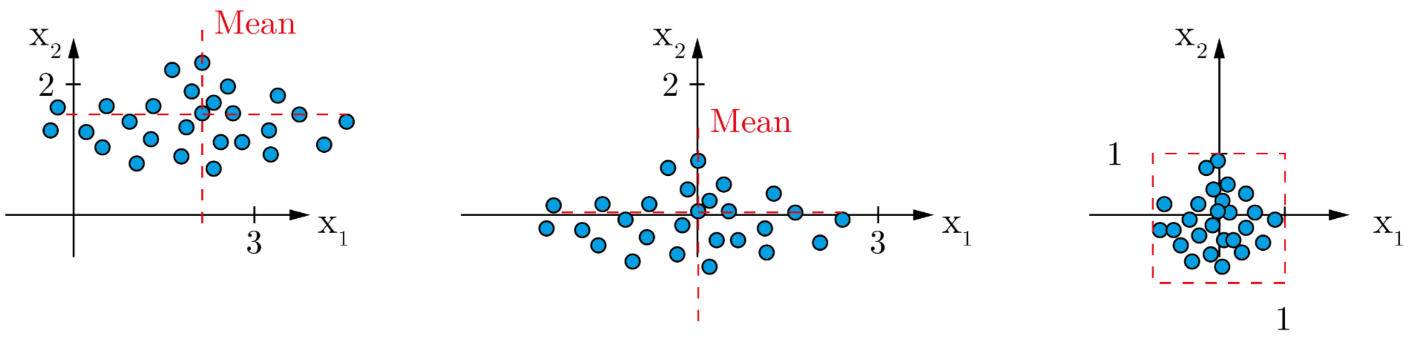
Training a neural network
2. Check Data Distribution of the Input Data: normalization



$$\mu = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$x_i = x_i - \mu$$

Training a neural network
2. Check Data Distribution of the Input Data: normalization



1. Subsract Mean

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x_i$$

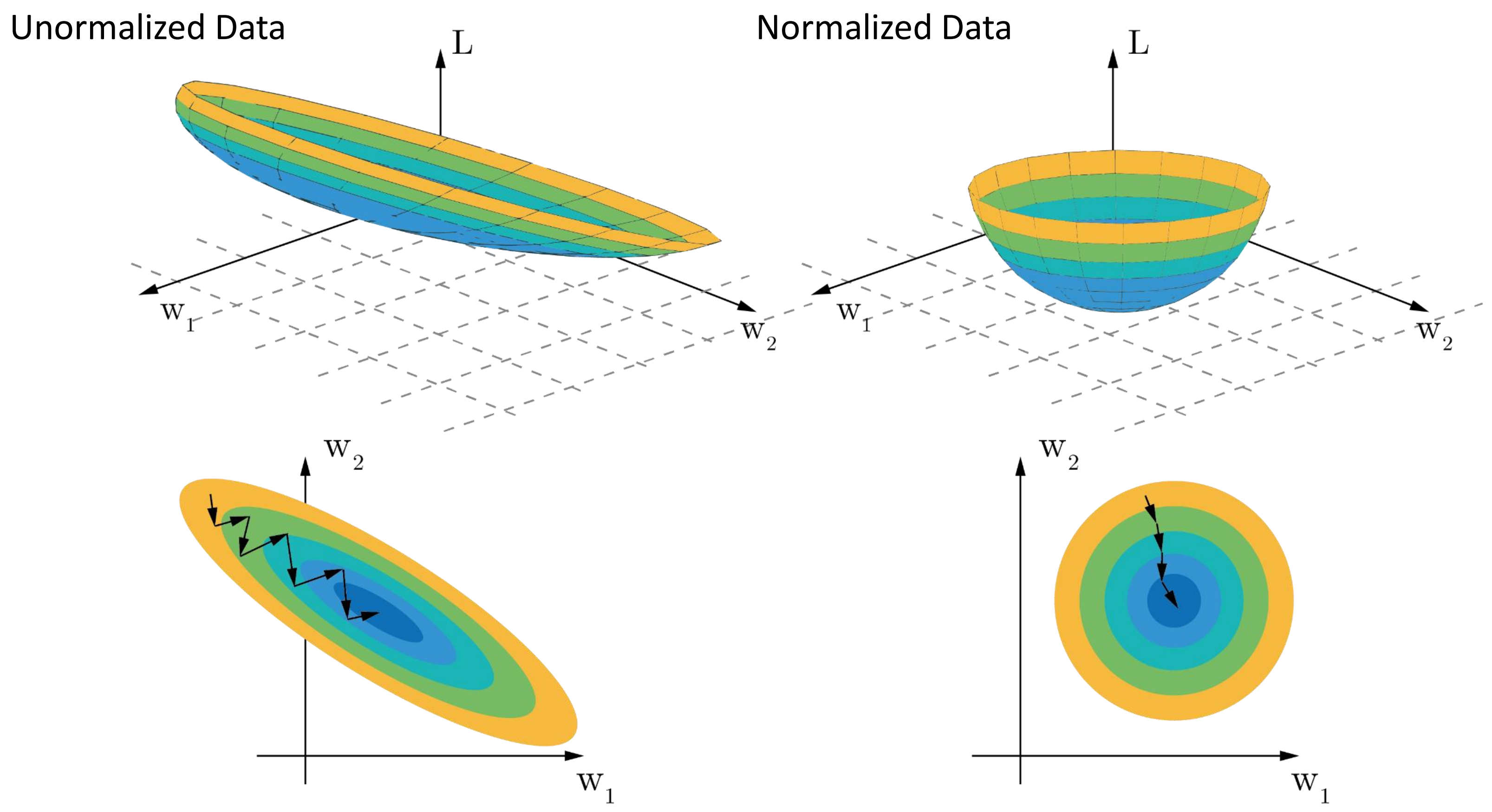
$$x_i = x_i - \mu$$

2. Normalize Variance

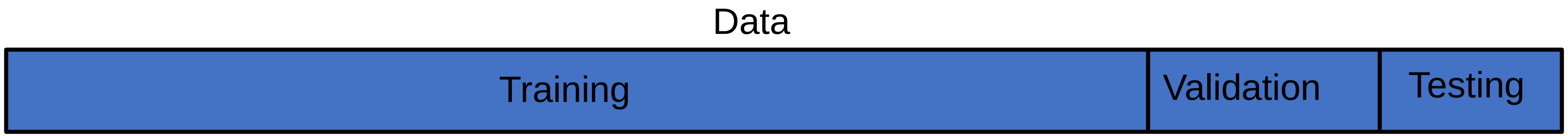
$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x_i^2$$

$$x_i = \frac{x_i}{\sigma^2}$$

Training a neural network
2. Check Data Distribution of the Input Data: normalization



Training a neural network
3. Dataset split



- Dataset generally split into three parts:
  - Training data: Data used during the training phase to compute gradient and loss
  - Validation data: Data used simultaneously during training to assess the risk of overfitting
  - Test data: Data never used during training and used to assess the performance of the trained neural network

Training a neural network
4. Network regularization – weight penalization

- Helps in avoiding overfitting of the model
- Discourage learning more complex model
- L2 regularization

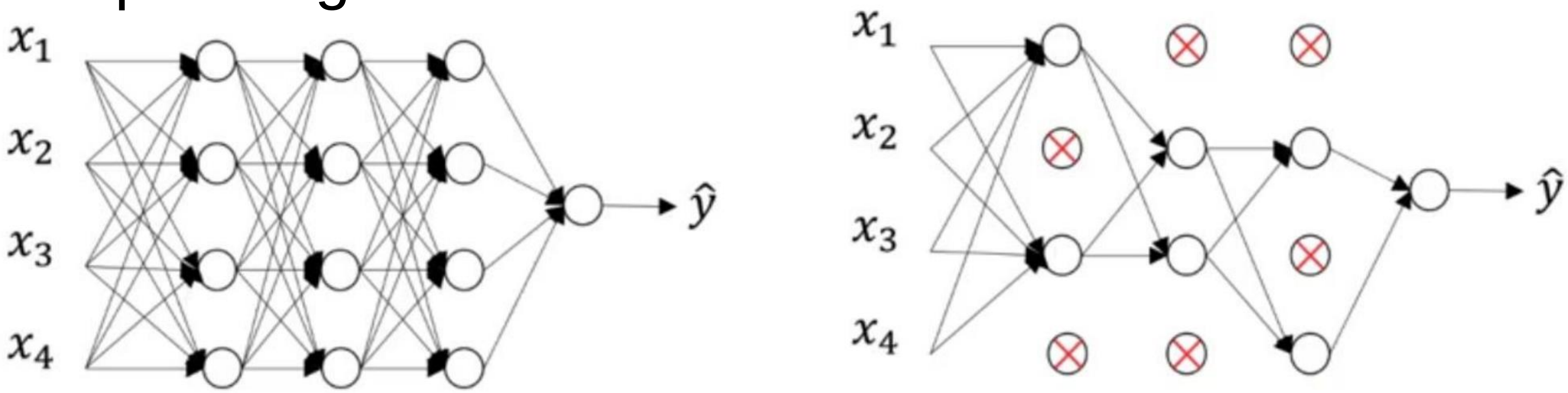
$$J(\boldsymbol{y}, \hat{\boldsymbol{y}}; \boldsymbol{W}, \boldsymbol{B}) = \frac{1}{m}\sum_{i=1}^{m} L(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L}\left\|\boldsymbol{w}^{(l)}\right\|_F^2$$

$$\left\|\boldsymbol{w}^{(l)}\right\|_F^2 = \sum_{i=1}^{n(l)}\sum_{j=1}^{n(l-1)} w_{ij}^2$$

- Large λ penalizes large w_{ij}
- (also L1 regularization)

Training a neural network
4. Network regularization – Dropout layer

- Dropout regularization

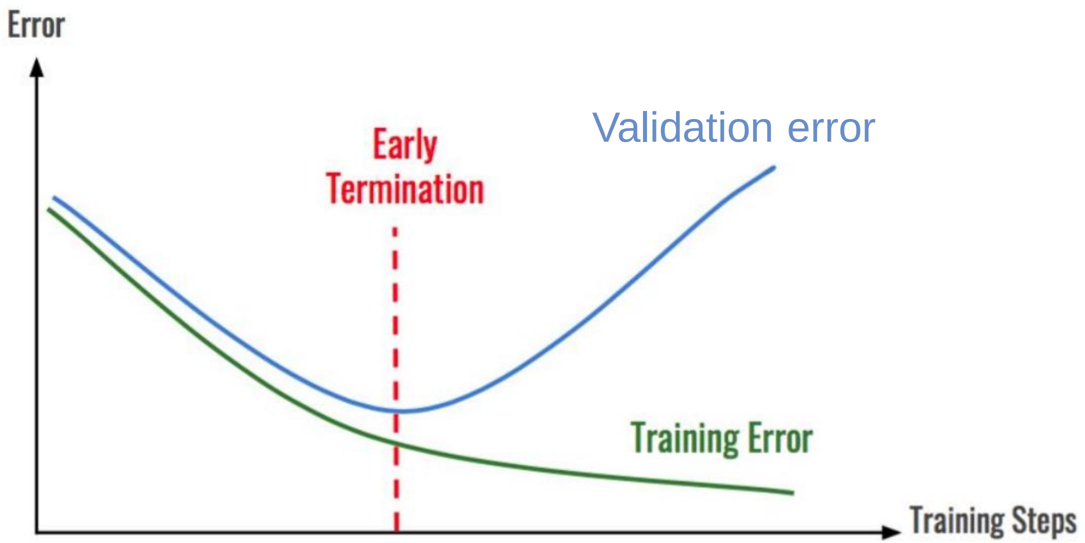


- Proportion of neurons are randomly "removed" during training for each batch
- Prevents excessive co-adaptation of the neurons
- Model cannot rely on a particular feature to make a prediction

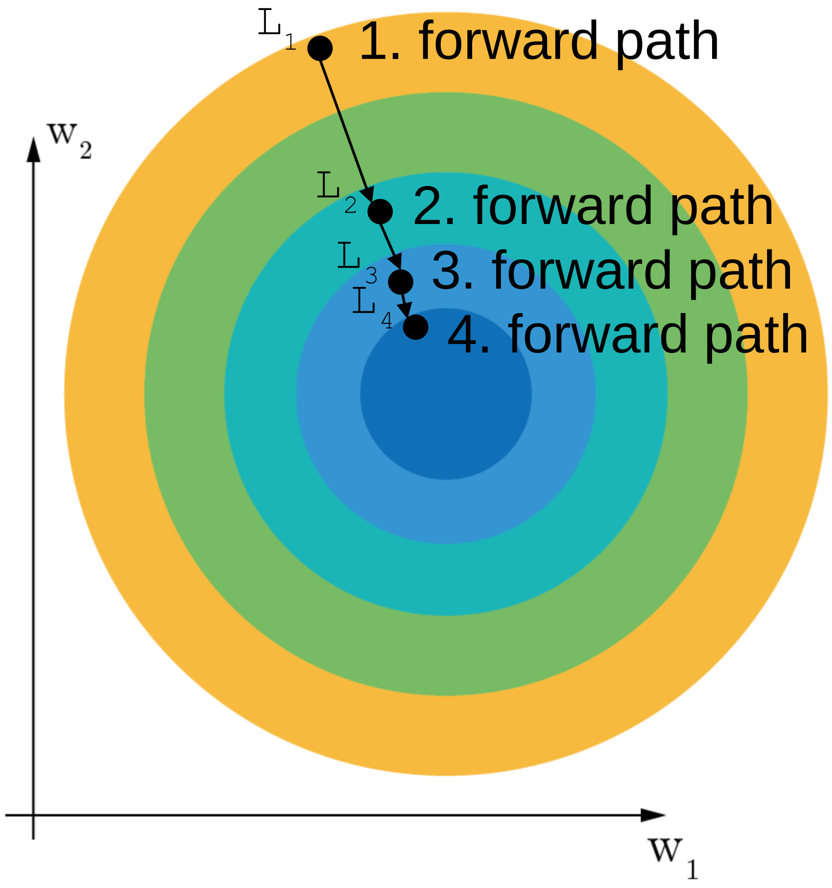
[Srivanista et al., JMLR, 2014]

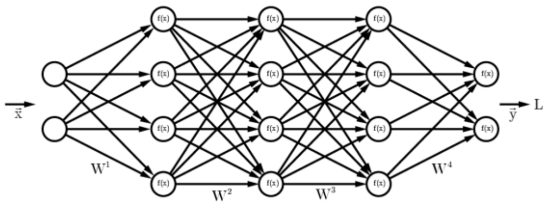
Training a neural network
4. Network regularization – Other

- Other regularization approaches:
  - Data augmentation (e.g. cropping, rotation, distortion in images)
  - Early stopping



Training a neural network
5. Optimization method – Gradient Descent





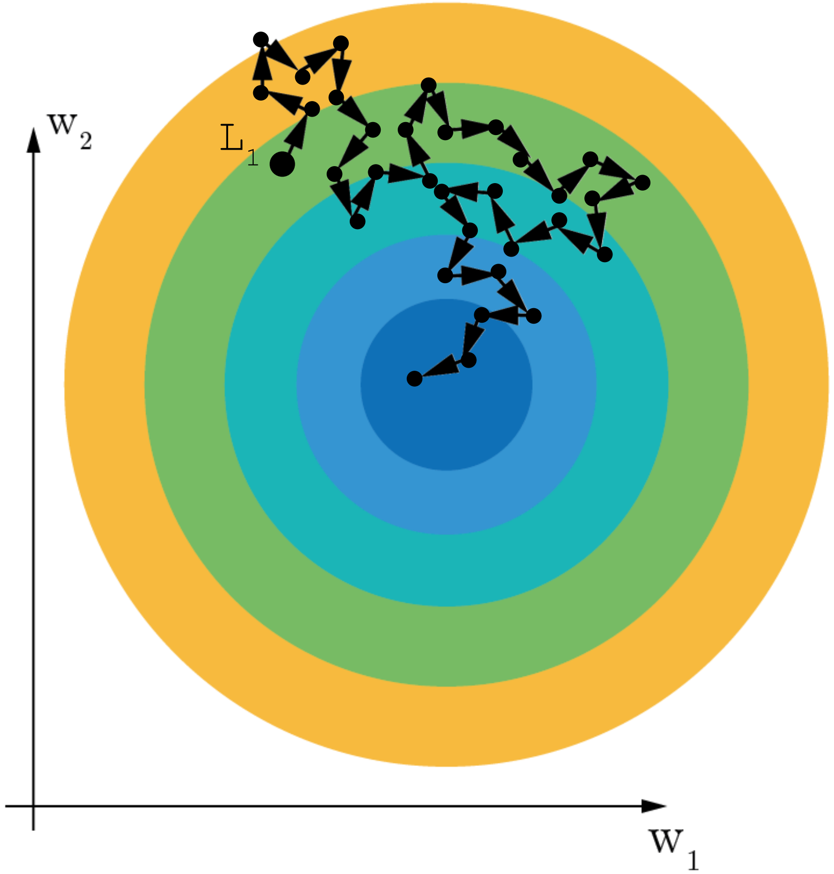
$$L(\boldsymbol{y}) = \frac{1}{N}\sum^{N}(y_i - \hat{y}_i)^2$$

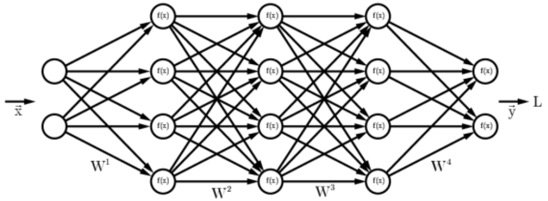
All the data

$\alpha \nabla L_w$

$$\boldsymbol{w}_{new} = \boldsymbol{w}_{old} - \alpha \nabla L_w$$

Training a neural network
5. Optimization method – Stochastic Gradient Descent





Only use one random datapoint at a time

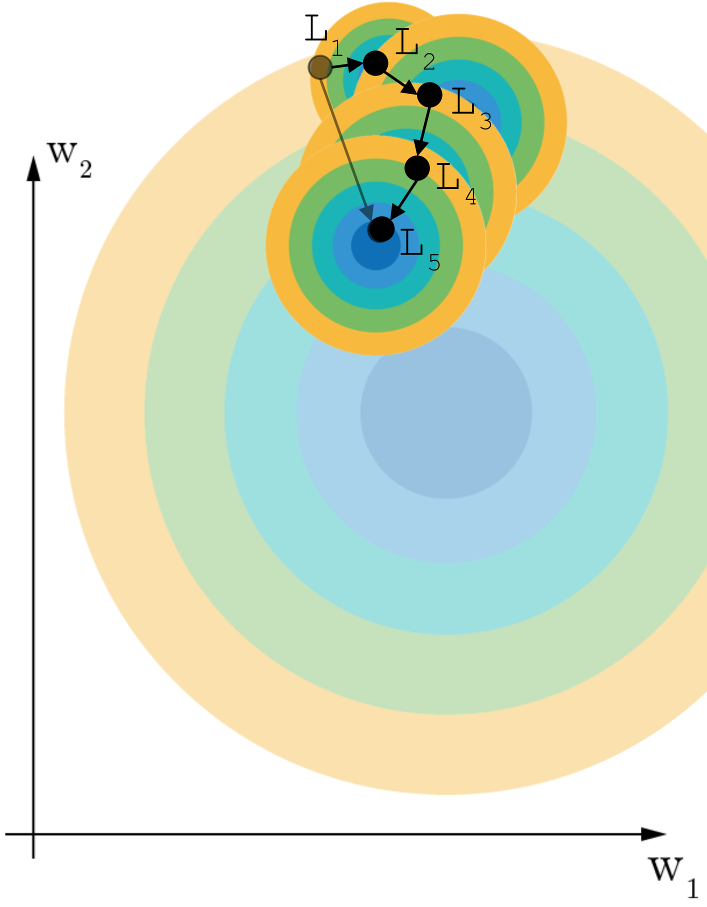
$$L(\boldsymbol{y}) = \sum^{1} \left(\boldsymbol{y}^{(i)} - \hat{\boldsymbol{y}}^{(i)}\right)^2 \rightarrow \alpha \nabla L_w$$

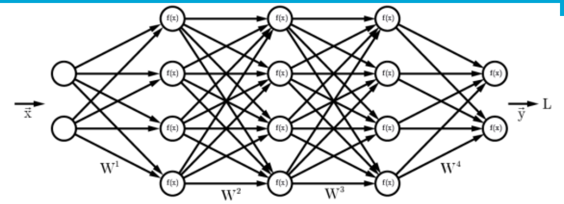
$$\boldsymbol{w}_{new} = \boldsymbol{w}_{old} - \alpha \nabla L_w$$

⇒ Reduces compute time per optimisation step

⇒ But finding local minimums takes longer

Batch Gradient Descent
A Compromise





$$L(\boldsymbol{y}) = \frac{1}{M}\sum^{M}(y_i - \hat{y}_i)^2$$

Split dataset in batches,
=> Trying to minimize global loss function with local cost functions

$$L(\boldsymbol{y}) = \frac{1}{M}\sum^{M}\left(\boldsymbol{y}^{(i)} - \hat{\boldsymbol{y}}^{(i)}\right)^2 \rightarrow \alpha\nabla L_w$$

$$\boldsymbol{w}_{new} = \boldsymbol{w}_{old} - \alpha\nabla L_w$$

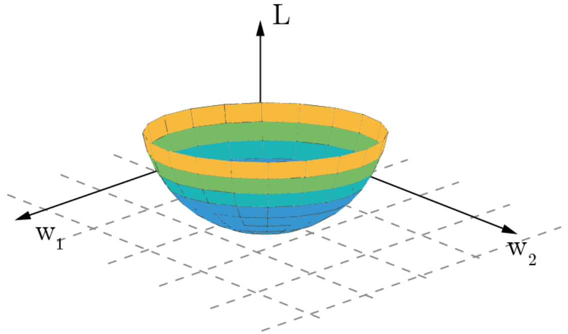
Learning rate can also be adapted: ADAM optimizer

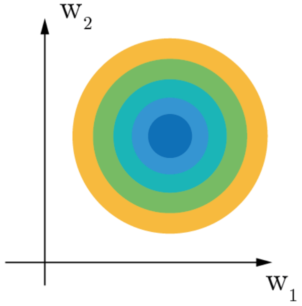
Training a neural network
5. Optimization method – Terminology

- **Training:**

for n < max_epochs:
    for sample in batch_size
        Forward path
        Backward path
        Accumulate loss
    Update weights

- Epochs: how often the entire training dataset is used
- Batch: how many samples are used to compute L and apply the gradient descent step





Summary

- Neural Network are tools that can approximate any nonlinear function
- Computational Graph enables a straightforward gradient calculation
- Backpropagation algorithm allows to compute weight updates
- Overview of training process for neural networks

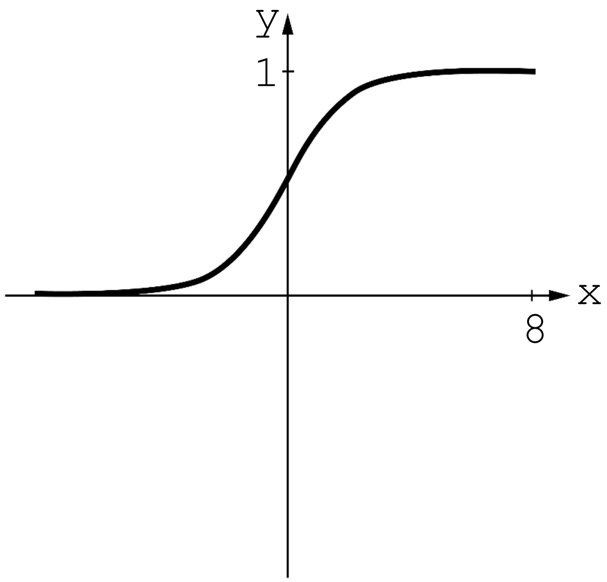
Extra slides on activation functions

Activation Functions

Sigmoid

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

$$f'(x) = f(x)\,(1 - f(x))$$



Two main problems:

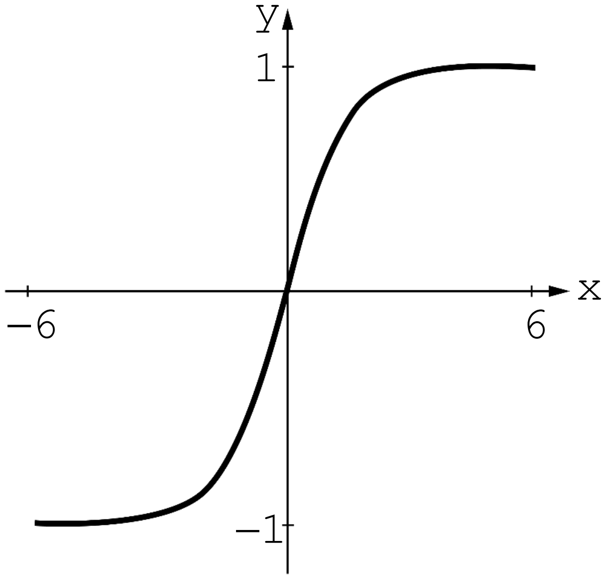
- Causes vanishing gradient: Gradient nearly zero for very large or small x, kills gradient and network stops learning
- Output isn‘t zero centered: Always all gradients positive or all negativ, inefficient weight updates

Activation Functions

Hyperbolic tangent

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

$$f'(x) = \frac{4}{(e^x + e^{-x})^2}$$



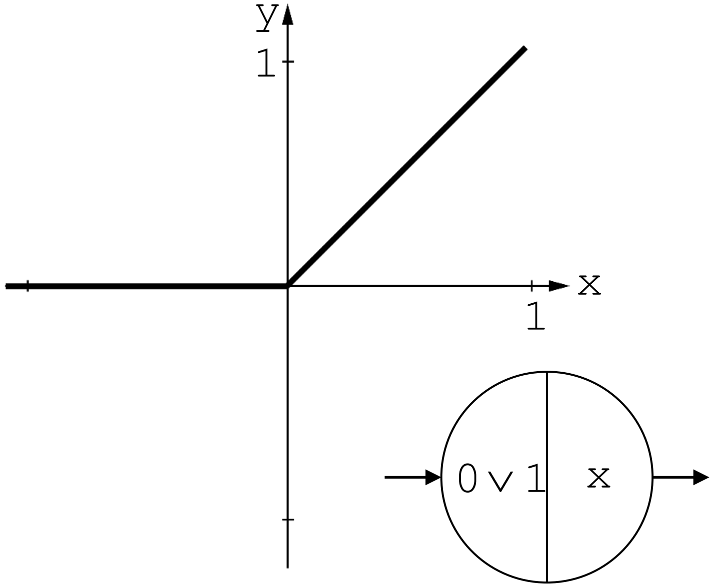
Better than sigmoid:

- Output is zero centered
- But still causes vanishing gradient

Activation Functions
Rectified Linear Unit (ReLU)

$$f(x) = \begin{cases} x, x \geq 0 \\ 0, x < 0 \end{cases}$$

$$f'(x) = \begin{cases} 1, x \geq 0 \\ 0, x < 0 \end{cases}$$



ReLU introducted in 1960s for visual feature extraction (Fukushima et al.)
Popularised in 2010s (Nair & Hinton)

Most common activation function:
- Computationally efficient
- Converges very fast
- Does not activate all neurons at the same time

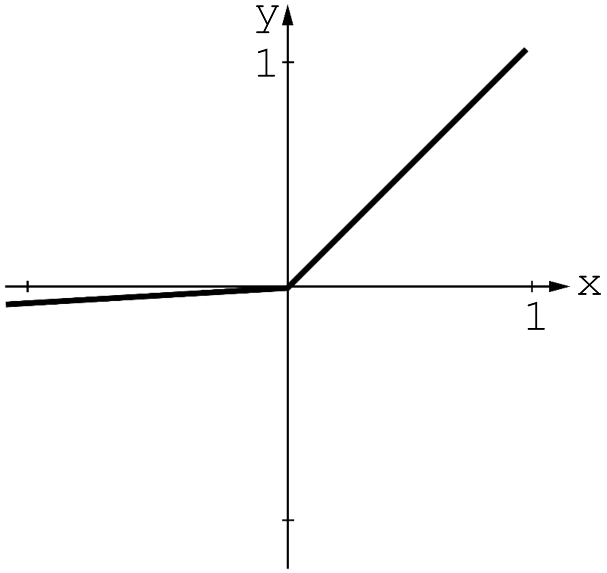
Problem:
- Gradient is zero for x<0 and can cause vanishing gradient -> dead relus may happen
- Not zero centered

Usage:
- Mostly used in hidden layers
- Positive bias at init to get active ReLU

Activation Functions
Leaky ReLU

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

$$f'(x) = \begin{cases} 1, x \geq 0 \\ a, x < 0 \end{cases}$$



Improves on ReLU:

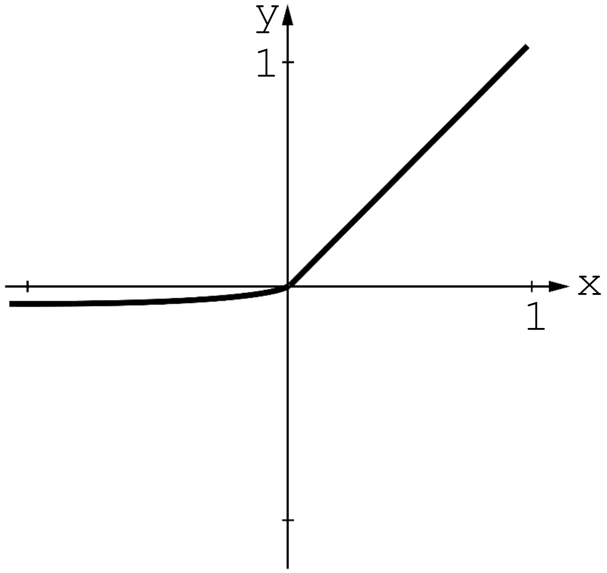
- Removes zero part of ReLU by adding a small slope. More stable then relu, but adds another paramter
- Computationally efficient
- Converges very fast
- Doesn‘t die
- Parameter a can also be learned by the network

Activation Functions

Exponential Linea Unit (ELU)

$$f(x) = \begin{cases} x, x \geq 0 \\ a(e^x - 1), x < 0 \end{cases}$$

$$f'(x) = \begin{cases} 1, x \geq 0 \\ ae^x, x < 0 \end{cases}$$



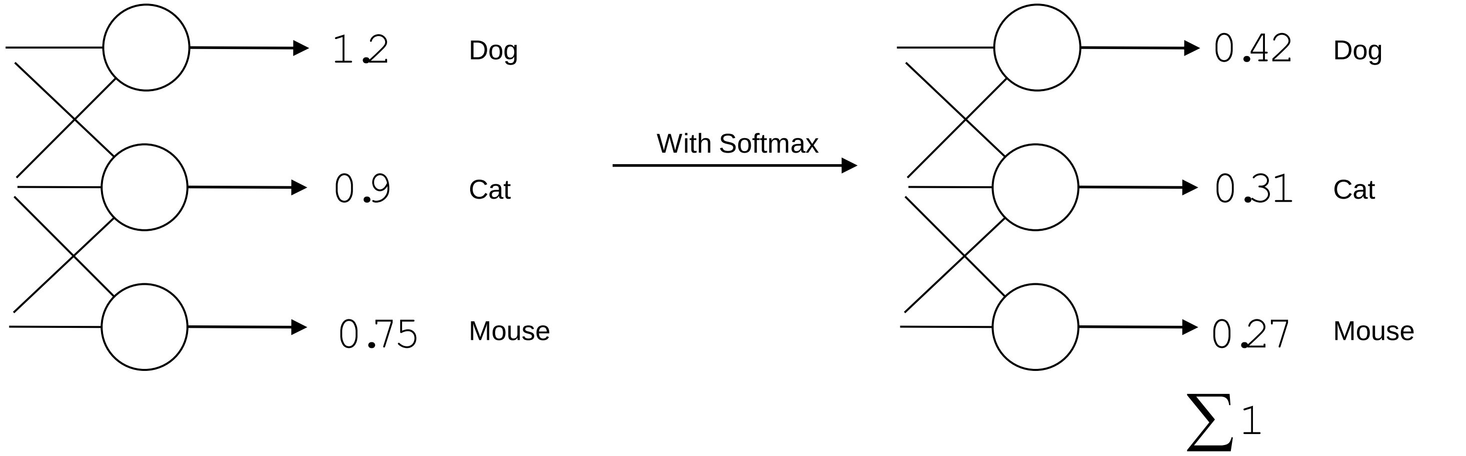
- Benefits of ReLU and Leaky ReLU
- Computation requires e^x

Activation Functions
Softmax

$$f(x) = \frac{e^{y_i}}{\sum_{k=1}^{K} e^{y_k}}$$

- Type of Sigmoid, handy for classification problems.
- Divides by the sum of all outputs, allows for percentage representation

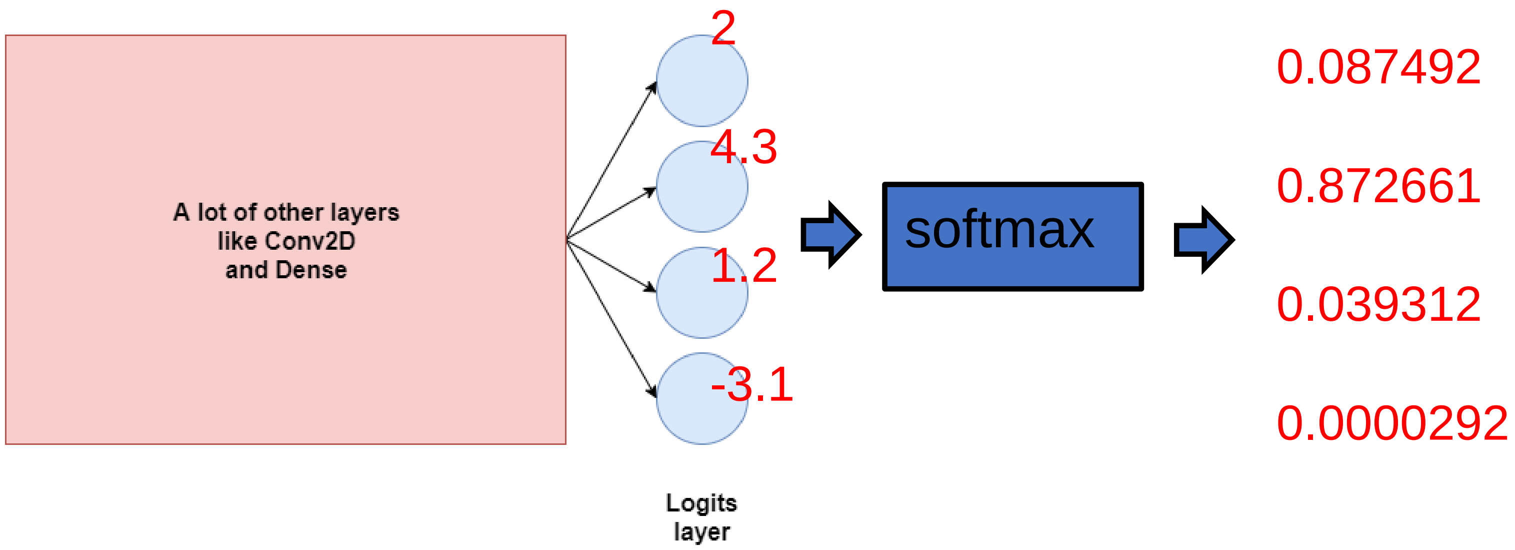
Classifier:



About the softmax function

Let's assume we need to classify between 4 classes

The softmax output the probability of each class



Then, during prediction, take the "argmax" to determine which class

Activation Functions

Rule of thumb

- Sigmoid / Softmax for classifiers
- Sigmoid, tanh sometimes avoided due to vanishing gradient
- ReLU mostly used today, but should only be used in hidden layer
- Start with ReLU if you don‘t get optimal results go for LeakyReLU or ELU
- Often, linear layer as the last layer of the network if regression problem