

# Introduction to Machine Learning

## Session 3: Artificial Neural Networks

Benjamin Paaßen

The University of Sydney

IK 2020, Günne

Licensed according to CC-BY-SA 3.0



THE UNIVERSITY OF  
**SYDNEY**

## Motivation: Learning on difficult data

- ▶ How to do ML for image and language data?

## Motivation: Learning on difficult data

- ▶ How to do ML for image and language data?
- ▶ Unclear features, big data, long-range dependencies, much noise

## Motivation: Learning on difficult data

- ▶ How to do ML for image and language data?
- ▶ Unclear features, big data, long-range dependencies, much noise  $\Rightarrow$  classic ML methods fail and/or require a lot of manual feature engineering

## Motivation: Learning on difficult data

- ▶ How to do ML for image and language data?
- ▶ Unclear features, big data, long-range dependencies, much noise  $\Rightarrow$  classic ML methods fail and/or require a lot of manual feature engineering
- ▶ We would like to specify a rough **architecture** for the entire input-to-output pipeline and **learn** all parameters along that pipeline (**end-to-end learning**)

## Motivation: Learning on difficult data

- ▶ How to do ML for image and language data?
- ▶ Unclear features, big data, long-range dependencies, much noise  $\Rightarrow$  classic ML methods fail and/or require a lot of manual feature engineering
- ▶ We would like to specify a rough **architecture** for the entire input-to-output pipeline and **learn** all parameters along that pipeline (**end-to-end learning**)
- ▶ This pipeline should support **multiple layers of abstraction** (**deep learning**; LeCun, Bengio, and Hinton 2015)

## Motivation: Learning on difficult data

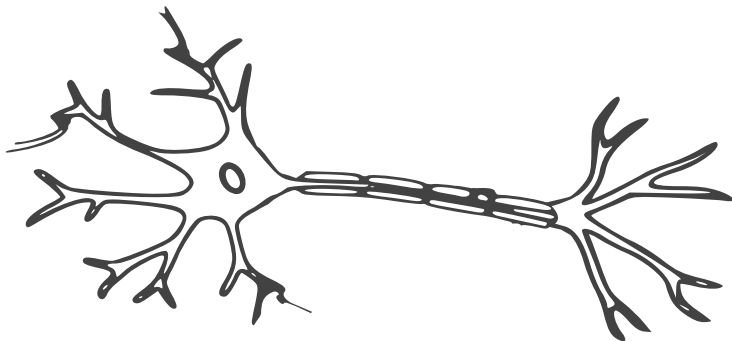
- ▶ How to do ML for image and language data?
- ▶ Unclear features, big data, long-range dependencies, much noise  $\Rightarrow$  classic ML methods fail and/or require a lot of manual feature engineering
- ▶ We would like to specify a rough **architecture** for the entire input-to-output pipeline and **learn** all parameters along that pipeline (**end-to-end learning**)
- ▶ This pipeline should support **multiple layers of abstraction** (**deep learning**; LeCun, Bengio, and Hinton 2015)
- ▶ Neural nets are **currently** the best way to do that

## Motivation: Learning on difficult data

- ▶ How to do ML for image and language data?
- ▶ Unclear features, big data, long-range dependencies, much noise  $\Rightarrow$  classic ML methods fail and/or require a lot of manual feature engineering
- ▶ We would like to specify a rough **architecture** for the entire input-to-output pipeline and **learn** all parameters along that pipeline (**end-to-end learning**)
- ▶ This pipeline should support **multiple layers of abstraction** (**deep learning**; LeCun, Bengio, and Hinton 2015)
- ▶ Neural nets are **currently** the best way to do that (and everything that does it has been dubbed a neural net)



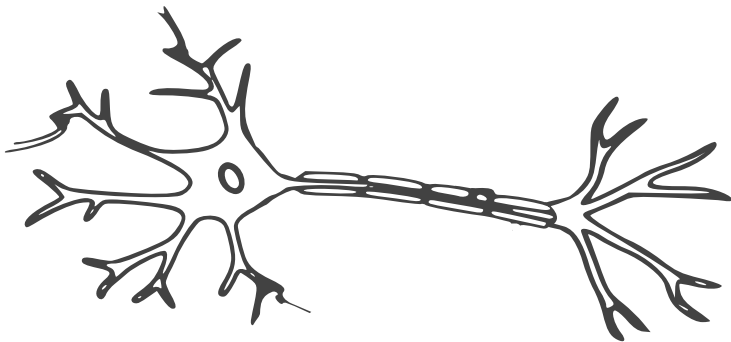
## What is an 'artificial neuron'? (1)



Brain neuron nerves cell by OpenClipart-Vectors-30363; usage according to pixabay license

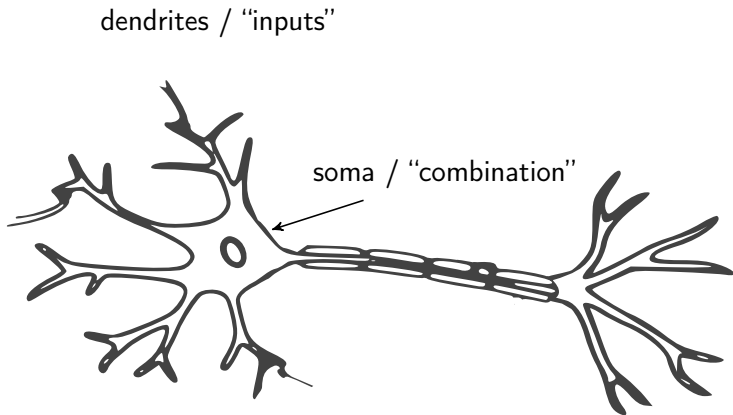
# What is an 'artificial neuron'? (1)

dendrites / "inputs"



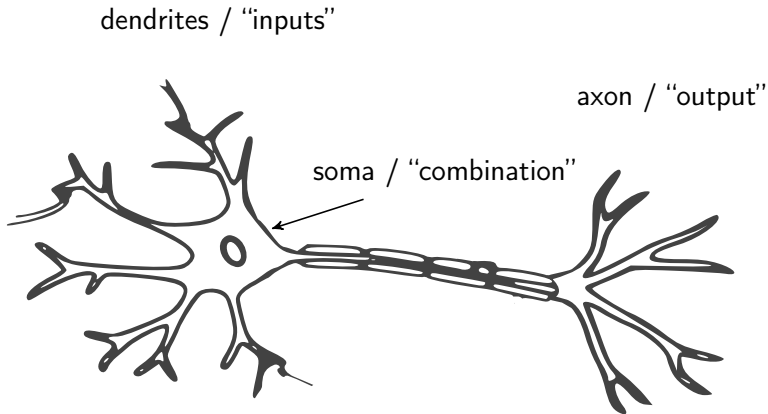
Brain neuron nerves cell by OpenClipart-Vectors-30363; usage according to pixabay license

# What is an 'artificial neuron'? (1)



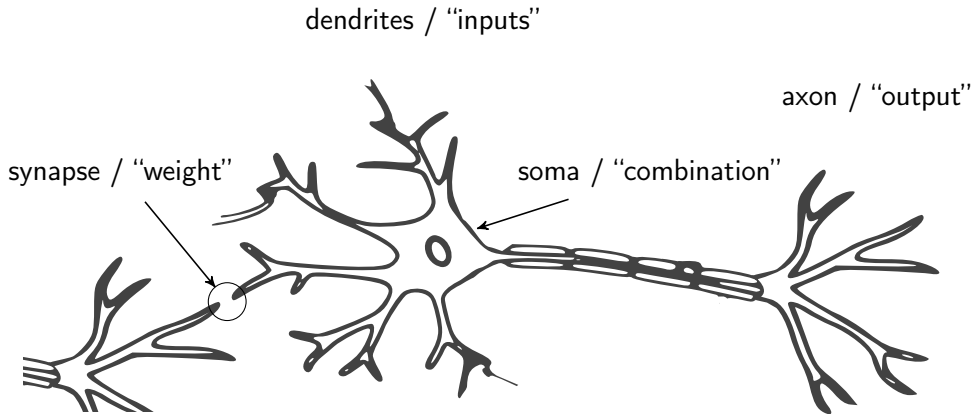
Brain neuron nerves cell by OpenClipart-Vectors-30363; usage according to pixabay license

## What is an 'artificial neuron'? (1)



Brain neuron nerves cell by OpenClipart-Vectors-30363; usage according to pixabay license

## What is an 'artificial neuron'? (1)



Brain neuron nerves cell by OpenClipart-Vectors-30363; usage according to pixabay license

## What is an 'artificial neuron'? (2)

outputs of other neurons  
“axons”

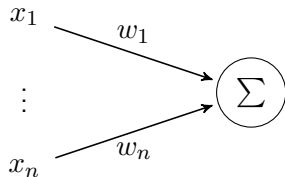
$x_1$

$\vdots$

$x_n$

## What is an 'artificial neuron'? (2)

outputs of other neurons  
“axons”

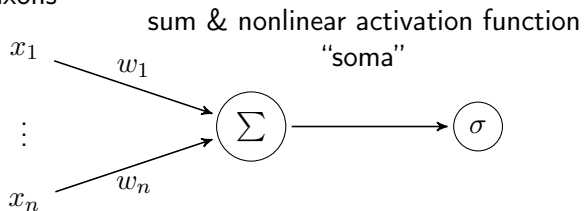


weight multiplication  
“synapses”

## What is an 'artificial neuron'? (2)

outputs of other neurons

"axons"



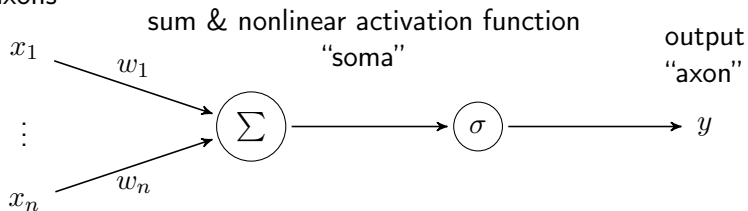
weight multiplication

"synapses"



## What is an 'artificial neuron'? (2)

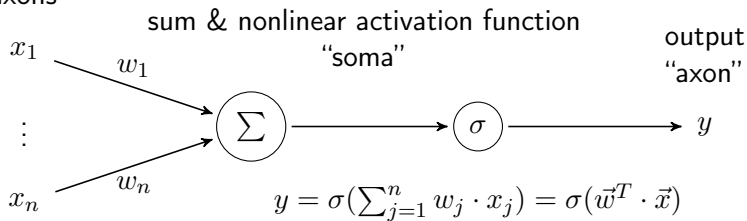
outputs of other neurons  
"axons"



weight multiplication  
"synapses"

## What is an 'artificial neuron'? (2)

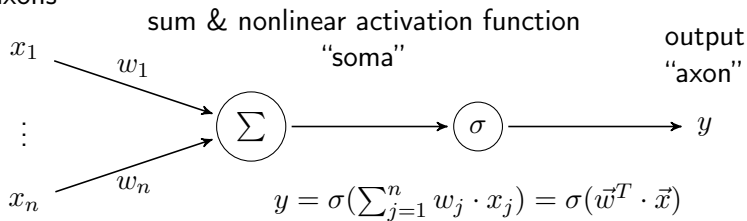
outputs of other neurons  
“axons”



weight multiplication  
“synapses”

## What is an 'artificial neuron'? (2)

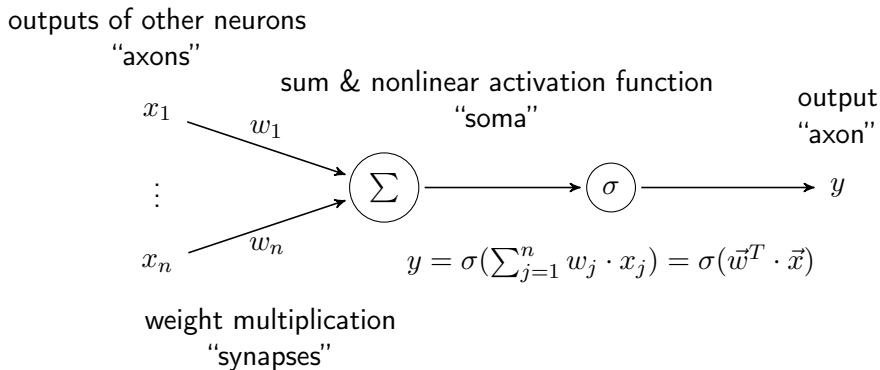
outputs of other neurons  
"axons"



weight multiplication  
"synapses"

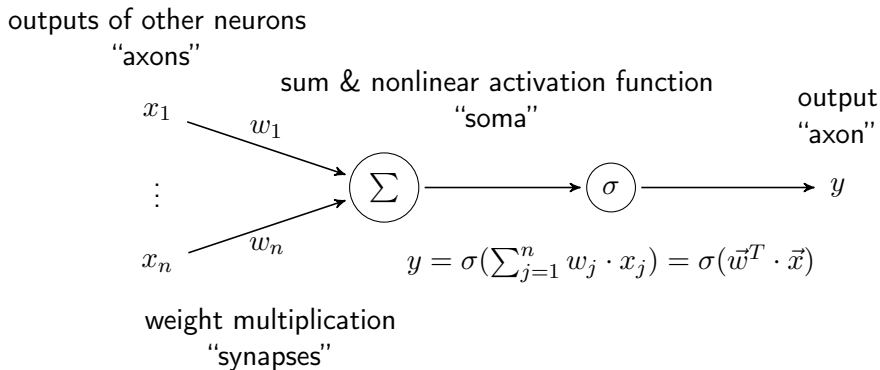
- Very loose relation to biology, if any

## What is an 'artificial neuron'? (2)



- ▶ Very loose relation to biology, if any
- ▶ For (most) MLers, artificial neurons are **engineering** tools

## What is an 'artificial neuron'? (2)



- ▶ Very loose relation to biology, if any
- ▶ For (most) MLers, artificial neurons are **engineering** tools, i.e. a set of **re-usable components** to build **model architectures** for ML **problem solving**

# A Gallery of Artificial Neural Network Modules



THE UNIVERSITY OF  
SYDNEY

# Binary Neurons

(McCulloch and Pitts 1943)

- ▶ Idea: Spikes are **binary** (they happen or not)

## Binary Neurons

(McCulloch and Pitts 1943)

- ▶ Idea: Spikes are **binary** (they happen or not)  $\Rightarrow$  outputs should be 0 or 1



## Binary Neurons

(McCulloch and Pitts 1943)

- ▶ Idea: Spikes are **binary** (they happen or not)  $\Rightarrow$  outputs should be 0 or 1
- ▶ Synaptic weights can be only excitatory or inhibitory, i.e.  $w_i \in \{-1, +1\}$

## Binary Neurons

(McCulloch and Pitts 1943)

- ▶ Idea: Spikes are **binary** (they happen or not)  $\Rightarrow$  outputs should be 0 or 1
- ▶ Synaptic weights can be only excitatory or inhibitory, i.e.  $w_i \in \{-1, +1\}$
- ▶ All neurons operate synchronously; weights are fixed, not learned

## Binary Neurons

(McCulloch and Pitts 1943)

- ▶ Idea: Spikes are **binary** (they happen or not)  $\Rightarrow$  outputs should be 0 or 1
- ▶ Synaptic weights can be only excitatory or inhibitory, i.e.  $w_i \in \{-1, +1\}$
- ▶ All neurons operate synchronously; weights are fixed, not learned

$x_1$

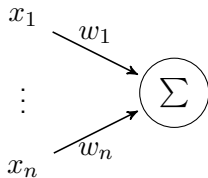
$\vdots$

$x_n$

# Binary Neurons

(McCulloch and Pitts 1943)

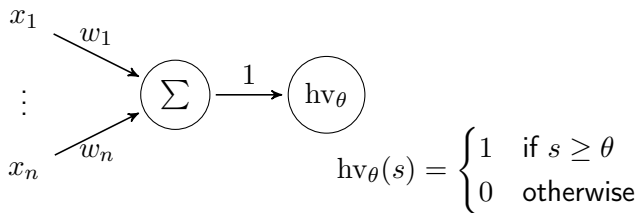
- ▶ Idea: Spikes are **binary** (they happen or not)  $\Rightarrow$  outputs should be 0 or 1
- ▶ Synaptic weights can be only excitatory or inhibitory, i.e.  $w_i \in \{-1, +1\}$
- ▶ All neurons operate synchronously; weights are fixed, not learned



## Binary Neurons

(McCulloch and Pitts 1943)

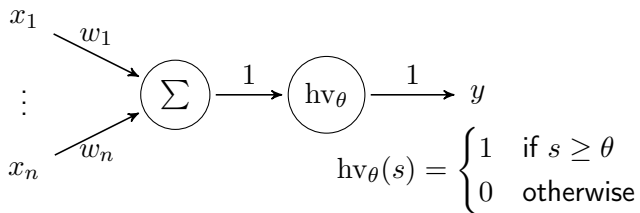
- ▶ Idea: Spikes are **binary** (they happen or not)  $\Rightarrow$  outputs should be 0 or 1
- ▶ Synaptic weights can be only excitatory or inhibitory, i.e.  $w_i \in \{-1, +1\}$
- ▶ All neurons operate synchronously; weights are fixed, not learned



## Binary Neurons

(McCulloch and Pitts 1943)

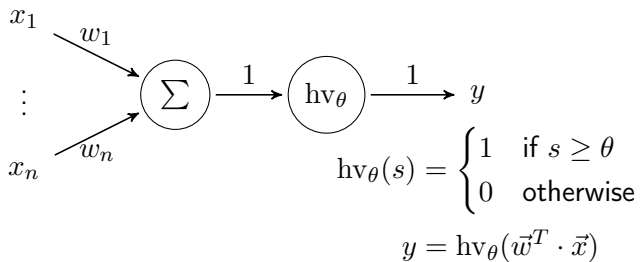
- ▶ Idea: Spikes are **binary** (they happen or not)  $\Rightarrow$  outputs should be 0 or 1
- ▶ Synaptic weights can be only excitatory or inhibitory, i.e.  $w_i \in \{-1, +1\}$
- ▶ All neurons operate synchronously; weights are fixed, not learned



## Binary Neurons

(McCulloch and Pitts 1943)

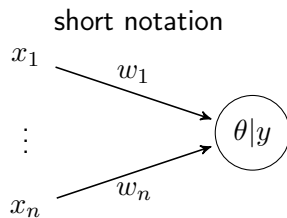
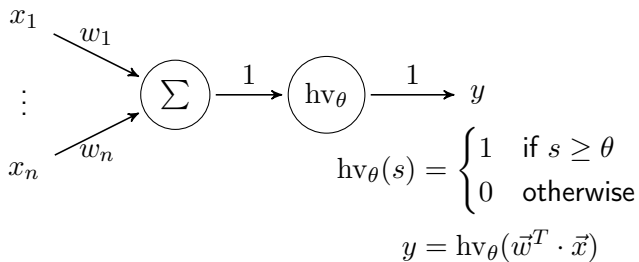
- ▶ Idea: Spikes are **binary** (they happen or not)  $\Rightarrow$  outputs should be 0 or 1
- ▶ Synaptic weights can be only excitatory or inhibitory, i.e.  $w_i \in \{-1, +1\}$
- ▶ All neurons operate synchronously; weights are fixed, not learned



# Binary Neurons

(McCulloch and Pitts 1943)

- ▶ Idea: Spikes are **binary** (they happen or not)  $\Rightarrow$  outputs should be 0 or 1
- ▶ Synaptic weights can be only excitatory or inhibitory, i.e.  $w_i \in \{-1, +1\}$
- ▶ All neurons operate synchronously; weights are fixed, not learned



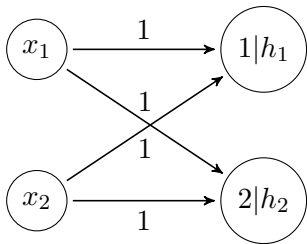


## Example: XOR-Network

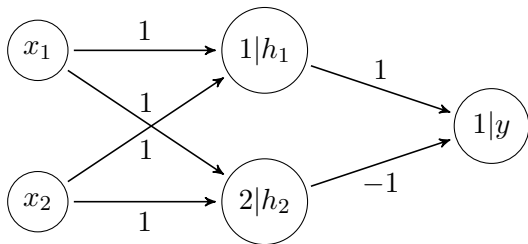
$x_1$

$x_2$

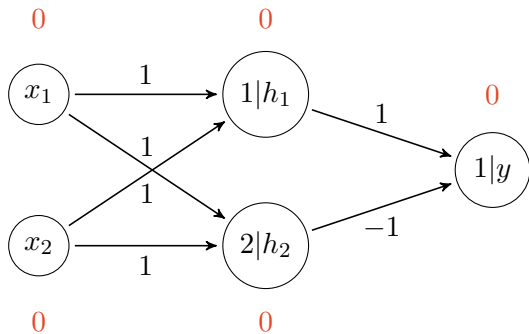
## Example: XOR-Network



## Example: XOR-Network

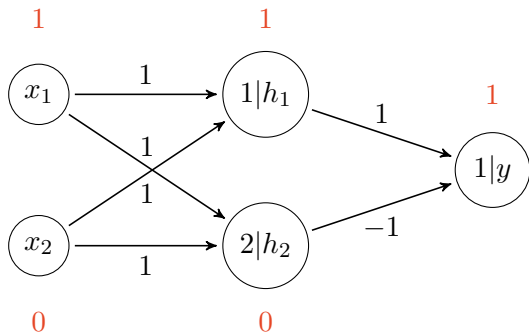


## Example: XOR-Network



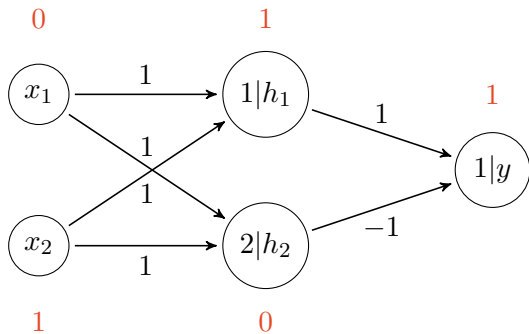
$x_1$	$x_2$	$h_1$	$h_2$	$y$
0	0	0	0	0

## Example: XOR-Network



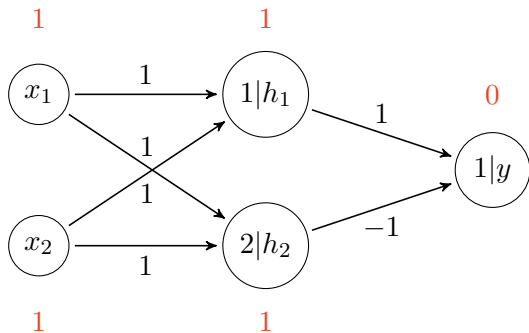
$x_1$	$x_2$	$h_1$	$h_2$	$y$
0	0	0	0	0
1	0	0	0	1

## Example: XOR-Network



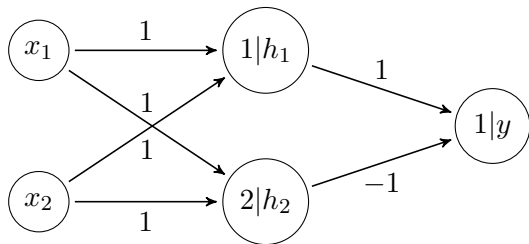
$x_1$	$x_2$	$h_1$	$h_2$	$y$
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1

## Example: XOR-Network



$x_1$	$x_2$	$h_1$	$h_2$	$y$
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
1	1	1	1	0

## Example: XOR-Network

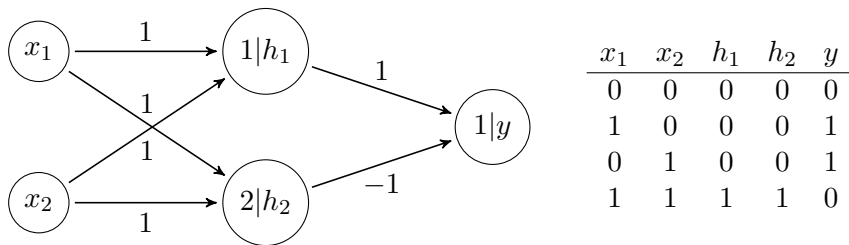


$x_1$	$x_2$	$h_1$	$h_2$	$y$
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
1	1	1	1	0

- It is easy to show that McCulloch and Pitts (1943) networks can compute **every** logical function by **decomposing** it into elementary blocks (or, and, not)



## Example: XOR-Network



- ▶ It is easy to show that McCulloch and Pitts (1943) networks can compute **every** logical function by **decomposing** it into elementary blocks (or, and, not)
- ▶ Highly influential in computer science via Von Neumann (1945) architecture

# Perceptron

(Rosenblatt 1958)

- ▶ Idea: A **learnable** classifier with **continuous** weights and inputs

# Perceptron

(Rosenblatt 1958)

- ▶ Idea: A **learnable** classifier with **continuous** weights and inputs
- ▶ Assume only **binary** class labels  $y \in \{-1, +1\}$

# Perceptron

(Rosenblatt 1958)

- ▶ Idea: A **learnable** classifier with **continuous** weights and inputs
- ▶ Assume only **binary** class labels  $y \in \{-1, +1\}$

$x_1$

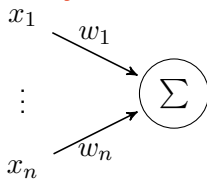
$\vdots$

$x_n$

# Perceptron

(Rosenblatt 1958)

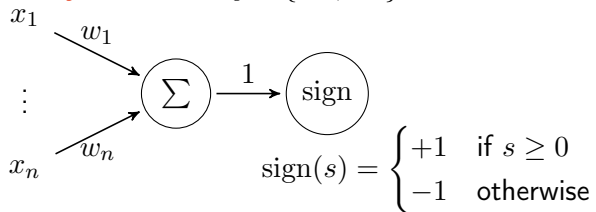
- ▶ Idea: A **learnable** classifier with **continuous** weights and inputs
- ▶ Assume only **binary** class labels  $y \in \{-1, +1\}$



# Perceptron

(Rosenblatt 1958)

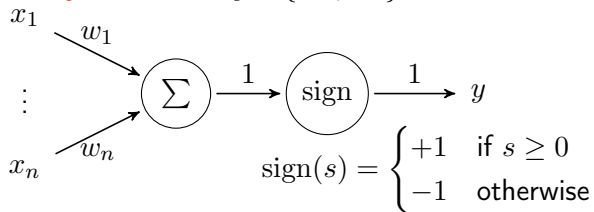
- ▶ Idea: A **learnable** classifier with **continuous** weights and inputs
- ▶ Assume only **binary** class labels  $y \in \{-1, +1\}$



# Perceptron

(Rosenblatt 1958)

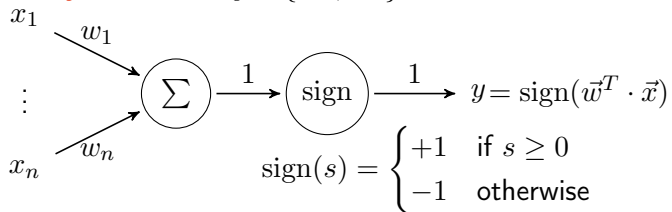
- ▶ Idea: A **learnable** classifier with **continuous** weights and inputs
- ▶ Assume only **binary** class labels  $y \in \{-1, +1\}$



# Perceptron

(Rosenblatt 1958)

- ▶ Idea: A **learnable** classifier with **continuous** weights and inputs
- ▶ Assume only **binary** class labels  $y \in \{-1, +1\}$

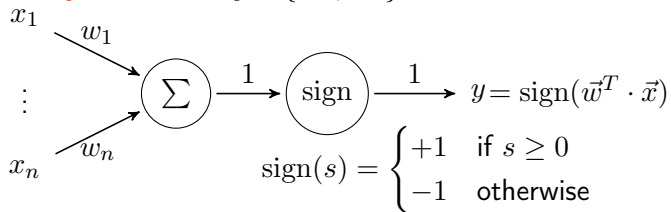




# Perceptron

(Rosenblatt 1958)

- ▶ Idea: A **learnable** classifier with **continuous** weights and inputs
- ▶ Assume only **binary** class labels  $y \in \{-1, +1\}$

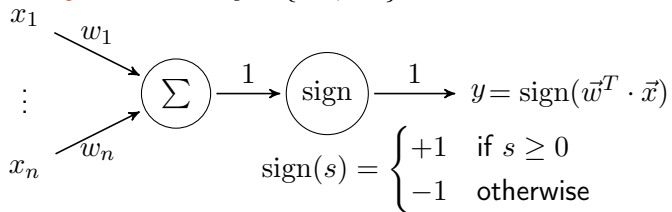


- ▶ Loss function:  $\ell_{\text{perc}}(\vec{w}) = \sum_{i=1}^N \text{ReLU}(-\vec{w}^T \cdot \vec{x}_i \cdot y_i)$

# Perceptron

(Rosenblatt 1958)

- ▶ Idea: A **learnable** classifier with **continuous** weights and inputs
- ▶ Assume only **binary** class labels  $y \in \{-1, +1\}$

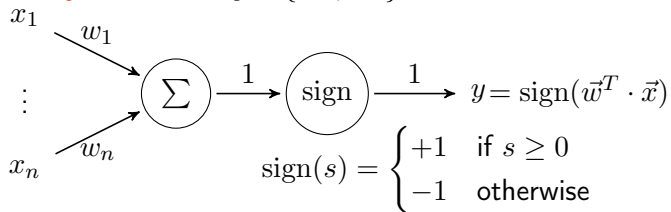


- ▶ Loss function:  $\ell_{\text{perc}}(\vec{w}) = \sum_{i=1}^N \text{ReLU}(-\vec{w}^T \cdot \vec{x}_i \cdot y_i)$  with  $\text{ReLU}(s) = \max\{s, 0\}$

# Perceptron

(Rosenblatt 1958)

- ▶ Idea: A **learnable** classifier with **continuous** weights and inputs
- ▶ Assume only **binary** class labels  $y \in \{-1, +1\}$

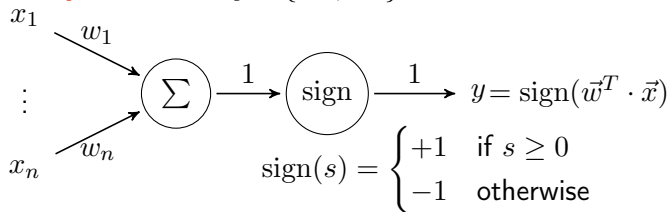


- ▶ Loss function:  $\ell_{\text{perc}}(\vec{w}) = \sum_{i=1}^N \text{ReLU}(-\vec{w}^T \cdot \vec{x}_i \cdot y_i)$  with  $\text{ReLU}(s) = \max\{s, 0\}$
- ▶ Learning via **stochastic** gradient descent from each sample  $(\vec{x}_i, y_i)$ :

# Perceptron

(Rosenblatt 1958)

- ▶ Idea: A **learnable** classifier with **continuous** weights and inputs
- ▶ Assume only **binary** class labels  $y \in \{-1, +1\}$



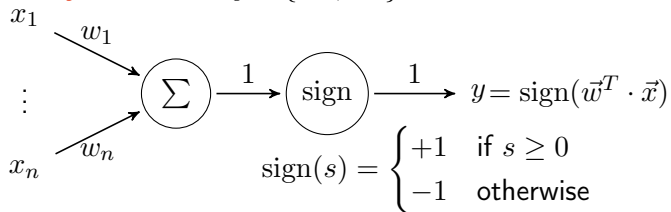
- ▶ Loss function:  $\ell_{\text{perc}}(\vec{w}) = \sum_{i=1}^N \text{ReLU}(-\vec{w}^T \cdot \vec{x}_i \cdot y_i)$  with  $\text{ReLU}(s) = \max\{s, 0\}$
- ▶ Learning via **stochastic** gradient descent from each sample  $(\vec{x}_i, y_i)$ :

$$\vec{w} \leftarrow \vec{w} - \eta \cdot \nabla_{\vec{w}} \text{ReLU}(-\vec{w}^T \cdot \vec{x}_i \cdot y_i)$$

# Perceptron

(Rosenblatt 1958)

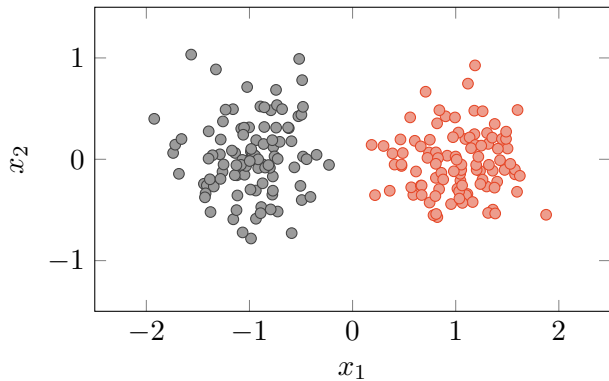
- ▶ Idea: A **learnable** classifier with **continuous** weights and inputs
- ▶ Assume only **binary** class labels  $y \in \{-1, +1\}$



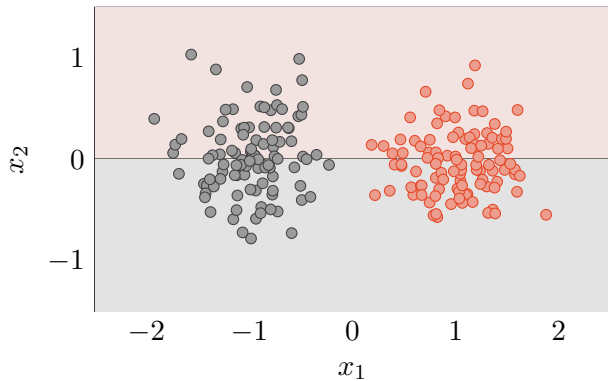
- ▶ Loss function:  $\ell_{\text{perc}}(\vec{w}) = \sum_{i=1}^N \text{ReLU}(-\vec{w}^T \cdot \vec{x}_i \cdot y_i)$  with  $\text{ReLU}(s) = \max\{s, 0\}$
- ▶ Learning via **stochastic** gradient descent from each sample  $(\vec{x}_i, y_i)$ :

$$\vec{w} \leftarrow \vec{w} - \eta \cdot \nabla_{\vec{w}} \text{ReLU}(-\vec{w}^T \cdot \vec{x}_i \cdot y_i) = \vec{w} + \eta \cdot \begin{cases} \vec{x}_i \cdot y_i & \text{if } \vec{w}^T \cdot \vec{x}_i \cdot y_i < 0 \\ 0 & \text{otherwise} \end{cases}$$

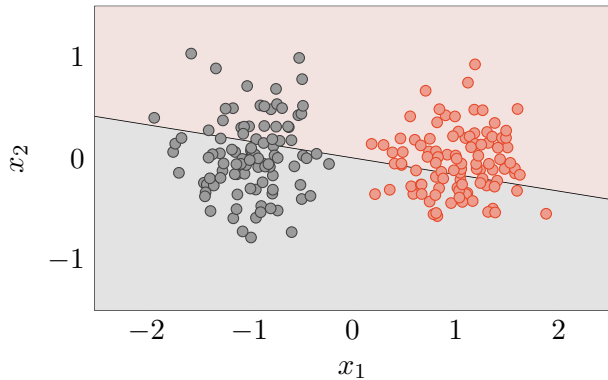
## Perceptron Learning Example



## Perceptron Learning Example

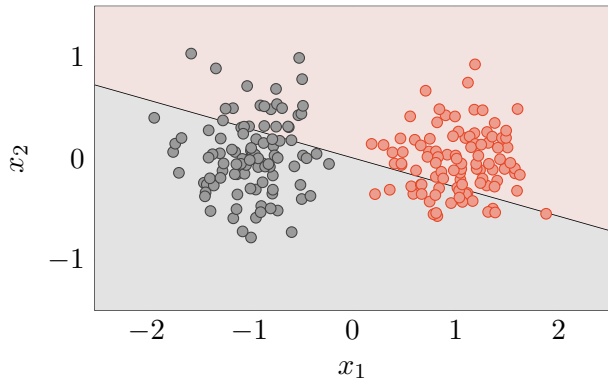


## Perceptron Learning Example

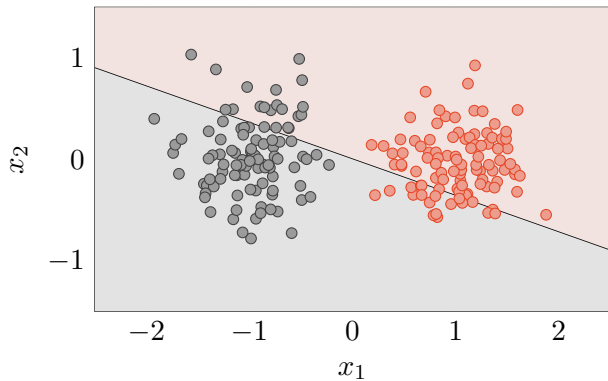




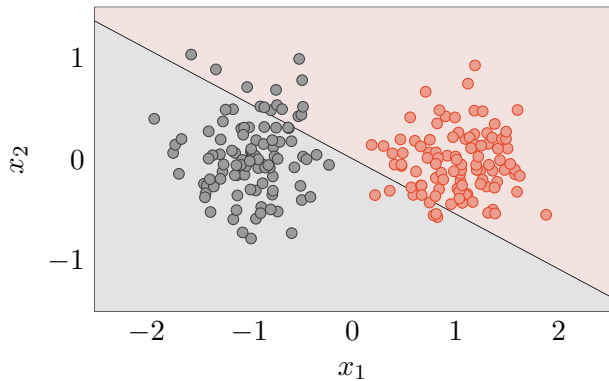
## Perceptron Learning Example



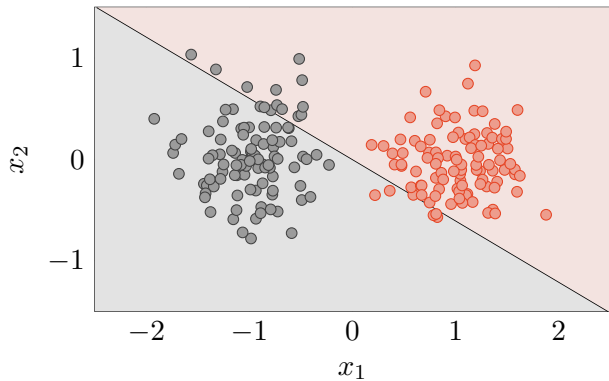
## Perceptron Learning Example



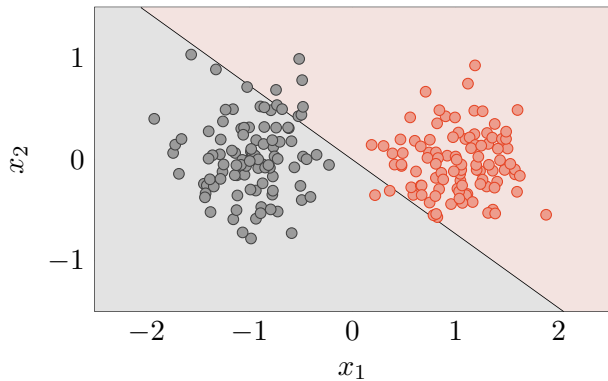
## Perceptron Learning Example



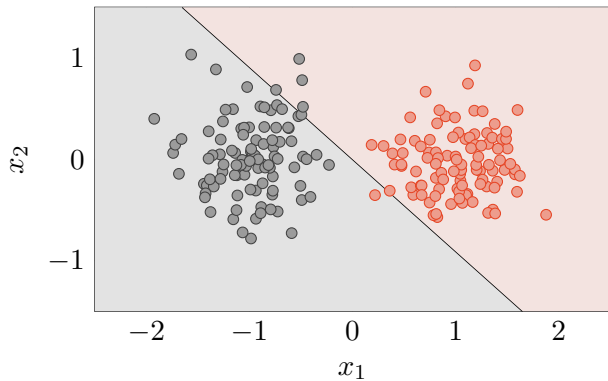
## Perceptron Learning Example



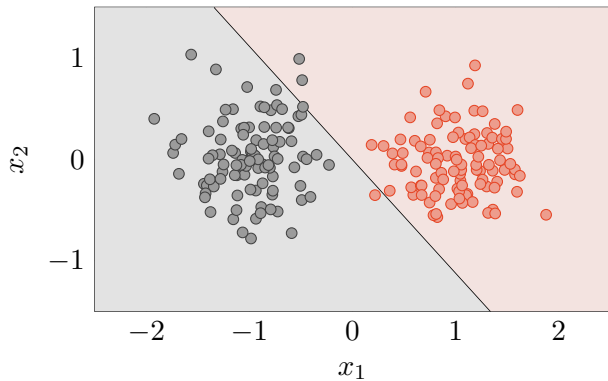
## Perceptron Learning Example



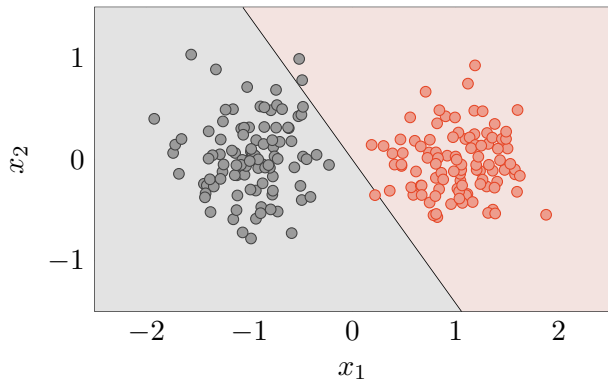
## Perceptron Learning Example



## Perceptron Learning Example

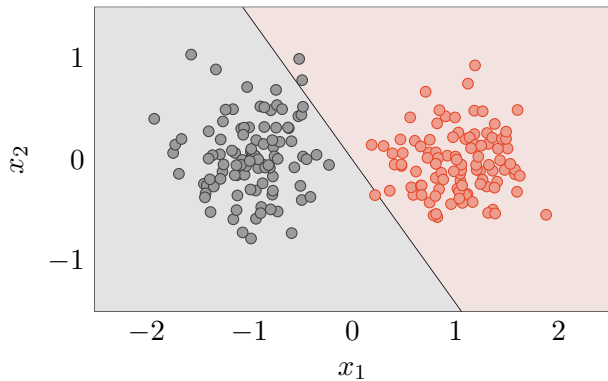


## Perceptron Learning Example

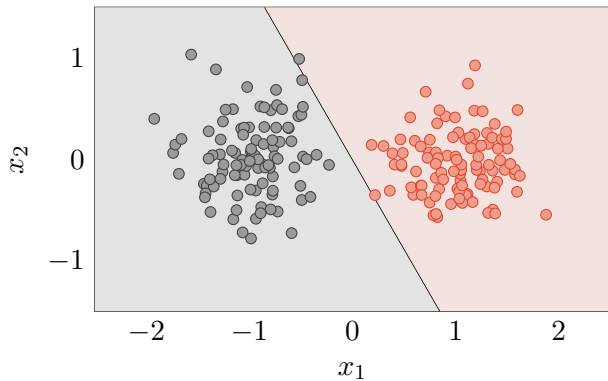




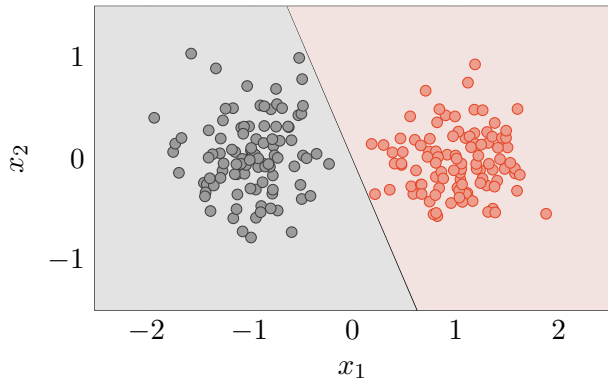
## Perceptron Learning Example



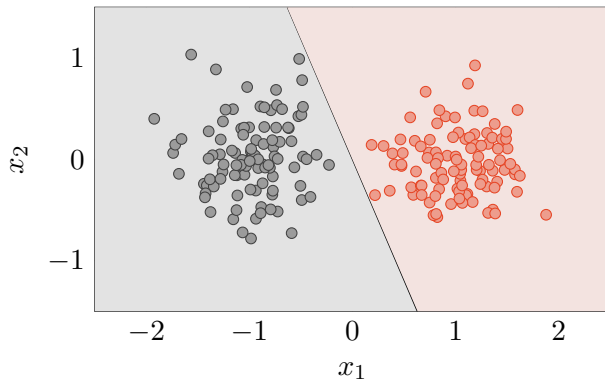
## Perceptron Learning Example



## Perceptron Learning Example

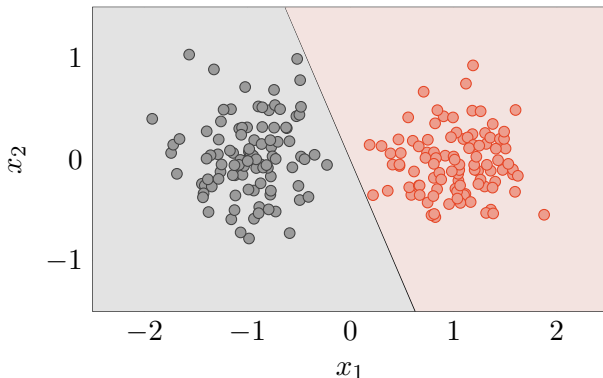


## Perceptron Learning Example



- Provably finds a solution with zero error if one exists!

## Perceptron Learning Example



- ▶ Provably finds a solution with zero error if one exists!
- ▶ **but** even some simple problems, like XOR, remain unsolvable (Minsky and Papert 1969; Olazaran 1996)

## Some historical notes

(Olazaran 1996)

- ▶ The first perceptron was implemented **in hardware**, contrasting Von Neumann (1945) architecture

## Some historical notes

(Olazaran 1996)

- ▶ The first perceptron was implemented **in hardware**, contrasting Von Neumann (1945) architecture
- ▶ Sparked considerable hype

## Some historical notes

(Olazaran 1996)

- ▶ The first perceptron was implemented **in hardware**, contrasting Von Neumann (1945) architecture
- ▶ Sparked considerable hype
- ▶ Limitations were known early on but solvable by multiple layers



## Some historical notes

(Olazaran 1996)

- ▶ The first perceptron was implemented **in hardware**, contrasting Von Neumann (1945) architecture
- ▶ Sparked considerable hype
- ▶ Limitations were known early on but solvable by multiple layers - just a multi-layer learning rule was missing

## Some historical notes

(Olazaran 1996)

- ▶ The first perceptron was implemented **in hardware**, contrasting Von Neumann (1945) architecture
- ▶ Sparked considerable hype
- ▶ Limitations were known early on but solvable by multiple layers - just a multi-layer learning rule was missing
- ▶ Success of digital computing and symbolic AI almost killed Perceptron research in the late 60s

## Some historical notes

(Olazaran 1996)

- ▶ The first perceptron was implemented **in hardware**, contrasting Von Neumann (1945) architecture
- ▶ Sparked considerable hype
- ▶ Limitations were known early on but solvable by multiple layers - just a multi-layer learning rule was missing
- ▶ Success of digital computing and symbolic AI almost killed Perceptron research in the late 60s, yet some in the field persisted

## Backpropagation(Linnainmaa 1970; Werbos 1974; Rumelhart, Hinton, and Williams 1986)

- ▶ Idea: Consider neural networks as cycle-free **computational graphs** where each operation is **differentiable**

## Backpropagation(Linnainmaa 1970; Werbos 1974; Rumelhart, Hinton, and Williams 1986)

- ▶ Idea: Consider neural networks as cycle-free **computational graphs** where each operation is **differentiable**
- ▶ Each node  $v$  has a **value**  $x_v = f(x_{u_1}, \dots, x_{u_n})$

## Backpropagation(Linnainmaa 1970; Werbos 1974; Rumelhart, Hinton, and Williams 1986)

- ▶ Idea: Consider neural networks as cycle-free **computational graphs** where each operation is **differentiable**
- ▶ Each node  $v$  has a **value**  $x_v = f(x_{u_1}, \dots, x_{u_n})$ , where  $f$  is some function and  $u_1, \dots, u_n$  are the **incoming nodes** of  $v$

## Backpropagation(Linnainmaa 1970; Werbos 1974; Rumelhart, Hinton, and Williams 1986)

- ▶ Idea: Consider neural networks as cycle-free **computational graphs** where each operation is **differentiable**
- ▶ Each node  $v$  has a **value**  $x_v = f(x_{u_1}, \dots, x_{u_n})$ , where  $f$  is some function and  $u_1, \dots, u_n$  are the **incoming nodes** of  $v$
- ▶ Example:

## Backpropagation (Linnainmaa 1970; Werbos 1974; Rumelhart, Hinton, and Williams 1986)

- ▶ Idea: Consider neural networks as cycle-free **computational graphs** where each operation is **differentiable**
- ▶ Each node  $v$  has a **value**  $x_v = f(x_{u_1}, \dots, x_{u_n})$ , where  $f$  is some function and  $u_1, \dots, u_n$  are the **incoming nodes** of  $v$
- ▶ Example:

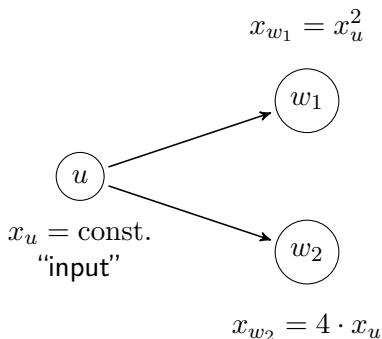


$x_u = \text{const.}$   
“input”



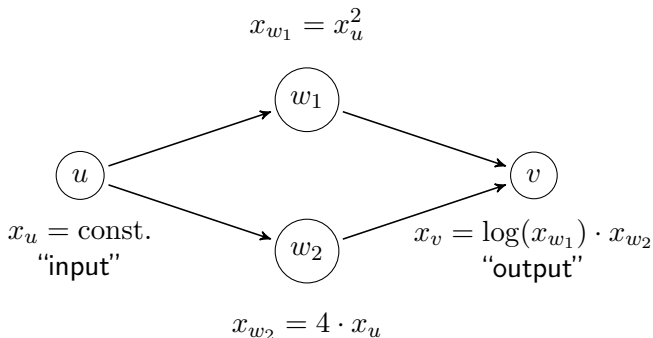
# Backpropagation (Linnainmaa 1970; Werbos 1974; Rumelhart, Hinton, and Williams 1986)

- ▶ Idea: Consider neural networks as cycle-free **computational graphs** where each operation is **differentiable**
- ▶ Each node  $v$  has a **value**  $x_v = f(x_{u_1}, \dots, x_{u_n})$ , where  $f$  is some function and  $u_1, \dots, u_n$  are the **incoming nodes** of  $v$
- ▶ Example:



# Backpropagation (Linnainmaa 1970; Werbos 1974; Rumelhart, Hinton, and Williams 1986)

- ▶ Idea: Consider neural networks as cycle-free **computational graphs** where each operation is **differentiable**
- ▶ Each node  $v$  has a **value**  $x_v = f(x_{u_1}, \dots, x_{u_n})$ , where  $f$  is some function and  $u_1, \dots, u_n$  are the **incoming nodes** of  $v$
- ▶ Example:



## Backpropagation (contd.)

- For any two nodes  $u, v$ , we can re-write:

$$\frac{\partial x_v}{\partial x_u} = \sum_{w \in N^+(u)} \frac{\partial x_w}{\partial x_u} \cdot \frac{\partial x_v}{\partial x_w}$$

## Backpropagation (contd.)

- For any two nodes  $u, v$ , we can re-write:

$$\frac{\partial x_v}{\partial x_u} = \sum_{w \in N^+(u)} \frac{\partial x_w}{\partial x_u} \cdot \frac{\partial x_v}{\partial x_w}$$

where  $N^+(u)$  are the **outgoing nodes** of  $u$

## Backpropagation (contd.)

- ▶ For any two nodes  $u, v$ , we can re-write:

$$\frac{\partial x_v}{\partial x_u} = \sum_{w \in N^+(u)} \frac{\partial x_w}{\partial x_u} \cdot \frac{\partial x_v}{\partial x_w}$$

where  $N^+(u)$  are the **outgoing nodes** of  $u$

- ▶ Example in our graph above:

$$\frac{\partial x_v}{\partial x_u} = \frac{\partial x_{w_1}}{\partial x_v} \cdot \frac{\partial x_u}{\partial x_{w_1}} + \frac{\partial x_{w_2}}{\partial x_v} \cdot \frac{\partial x_u}{\partial x_{w_2}}$$

## Backpropagation (contd.)

- ▶ For any two nodes  $u, v$ , we can re-write:

$$\frac{\partial x_v}{\partial x_u} = \sum_{w \in N^+(u)} \frac{\partial x_w}{\partial x_u} \cdot \frac{\partial x_v}{\partial x_w}$$

where  $N^+(u)$  are the **outgoing nodes** of  $u$

- ▶ Example in our graph above:

$$\frac{\partial x_v}{\partial x_u} = \frac{\partial x_{w_1}}{\partial x_v} \cdot \frac{\partial x_u}{\partial x_{w_1}} + \frac{\partial x_{w_2}}{\partial x_v} \cdot \frac{\partial x_u}{\partial x_{w_2}} = 2 \cdot x_v \cdot \frac{x_{w_2}}{x_{w_1}} + 4 \cdot \log(x_{w_1})$$

## Backpropagation (contd.)

- ▶ For any two nodes  $u, v$ , we can re-write:

$$\frac{\partial x_v}{\partial x_u} = \sum_{w \in N^+(u)} \frac{\partial x_w}{\partial x_u} \cdot \frac{\partial x_v}{\partial x_w}$$

where  $N^+(u)$  are the **outgoing nodes** of  $u$

- ▶ Example in our graph above:

$$\frac{\partial x_v}{\partial x_u} = \frac{\partial x_{w_1}}{\partial x_v} \cdot \frac{\partial x_u}{\partial x_{w_1}} + \frac{\partial x_{w_2}}{\partial x_v} \cdot \frac{\partial x_u}{\partial x_{w_2}} = 2 \cdot x_v \cdot \frac{x_{w_2}}{x_{w_1}} + 4 \cdot \log(x_{w_1})$$

- ▶ We can use this formula **recursively** to compute derivatives from any descendant nodes

## Backpropagation (contd.)

- ▶ For any two nodes  $u, v$ , we can re-write:

$$\frac{\partial x_v}{\partial x_u} = \sum_{w \in N^+(u)} \frac{\partial x_w}{\partial x_u} \cdot \frac{\partial x_v}{\partial x_w}$$

where  $N^+(u)$  are the **outgoing nodes** of  $u$

- ▶ Example in our graph above:

$$\frac{\partial x_v}{\partial x_u} = \frac{\partial x_{w_1}}{\partial x_v} \cdot \frac{\partial x_u}{\partial x_{w_1}} + \frac{\partial x_{w_2}}{\partial x_v} \cdot \frac{\partial x_u}{\partial x_{w_2}} = 2 \cdot x_v \cdot \frac{x_{w_2}}{x_{w_1}} + 4 \cdot \log(x_{w_1})$$

- ▶ We can use this formula **recursively** to compute derivatives from any descendant nodes, e.g. the derivative of the loss w.r.t. weights



## Backpropagation (contd.)

- ▶ For any two nodes  $u, v$ , we can re-write:

$$\frac{\partial x_v}{\partial x_u} = \sum_{w \in N^+(u)} \frac{\partial x_w}{\partial x_u} \cdot \frac{\partial x_v}{\partial x_w}$$

where  $N^+(u)$  are the **outgoing nodes** of  $u$

- ▶ Example in our graph above:

$$\frac{\partial x_v}{\partial x_u} = \frac{\partial x_{w_1}}{\partial x_v} \cdot \frac{\partial x_u}{\partial x_{w_1}} + \frac{\partial x_{w_2}}{\partial x_v} \cdot \frac{\partial x_u}{\partial x_{w_2}} = 2 \cdot x_v \cdot \frac{x_{w_2}}{x_{w_1}} + 4 \cdot \log(x_{w_1})$$

- ▶ We can use this formula **recursively** to compute derivatives from any descendant nodes, e.g. the derivative of the loss w.r.t. weights
- ▶ That is the main utility of frameworks like tensorflow and pytorch!

## Activation functions

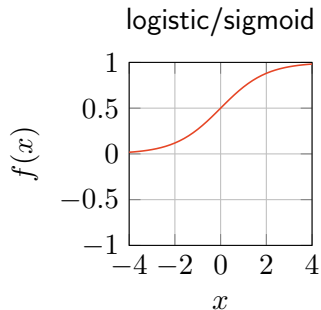
- ▶ But how to make artificial neurons differentiable?

## Activation functions

- ▶ But how to make artificial neurons differentiable?  $\Rightarrow$  replace threshold function with a differentiable surrogate

## Activation functions

- But how to make artificial neurons differentiable?  $\Rightarrow$  replace threshold function with a differentiable surrogate



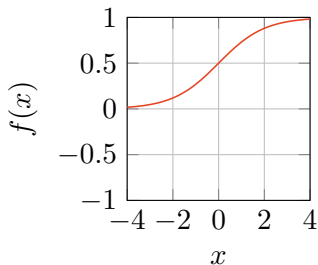
$$f(x) = \frac{1}{1 + \exp(-x)}$$

$$\frac{\partial}{\partial x} f(x) = f(x) \cdot (1 - f(x))$$

## Activation functions

- But how to make artificial neurons differentiable?  $\Rightarrow$  replace threshold function with a differentiable surrogate

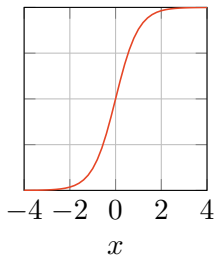
logistic/sigmoid



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

$$\frac{\partial}{\partial x} f(x) = f(x) \cdot (1 - f(x))$$

tanh



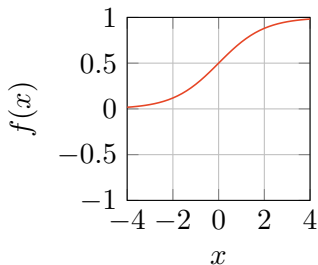
$$f(x) = \tanh(x)$$

$$\frac{\partial}{\partial x} f(x) = 1 - (\tanh(x))^2$$

## Activation functions

- But how to make artificial neurons differentiable?  $\Rightarrow$  replace threshold function with a differentiable surrogate

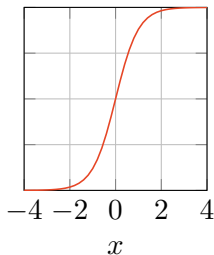
logistic/sigmoid



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

$$\frac{\partial}{\partial x} f(x) = f(x) \cdot (1 - f(x))$$

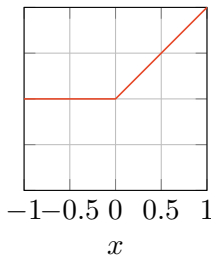
tanh



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

$$\frac{\partial}{\partial x} f(x) = 1 - (\tanh(x))^2$$

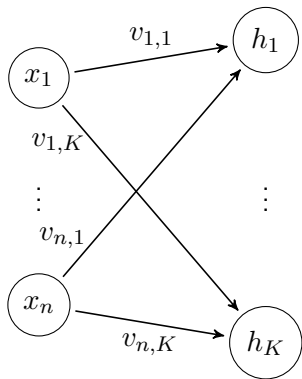
ReLU



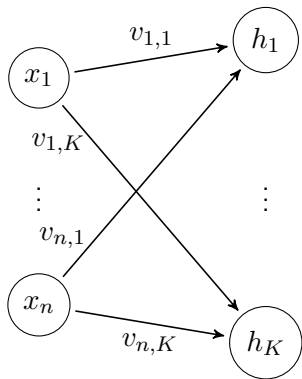
$$f(x) = \max\{0, x\}$$

$$\frac{\partial}{\partial x} f(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

## Example: Single-hidden-layer perceptron



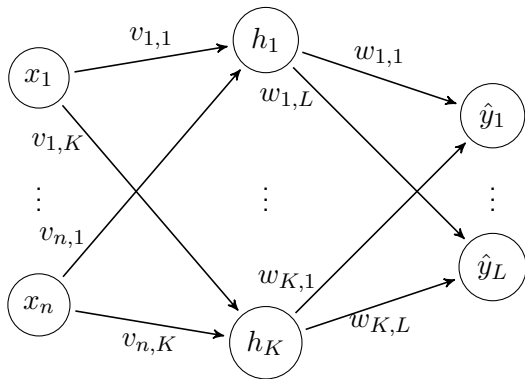
## Example: Single-hidden-layer perceptron



$$h_k = \frac{1}{1 + \exp\left(-\sum_{j=1}^n v_{j,k} \cdot x_j - b_k\right)}$$

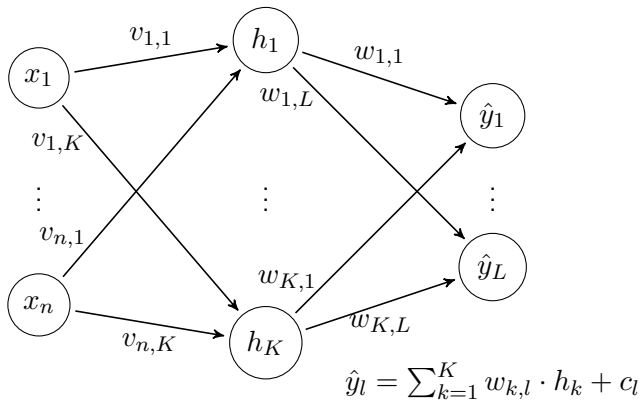


## Example: Single-hidden-layer perceptron



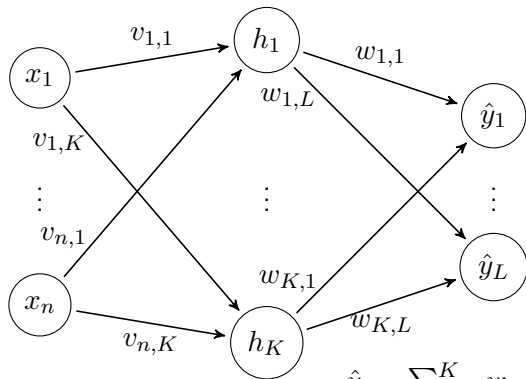
$$h_k = \frac{1}{1 + \exp\left(-\sum_{j=1}^n v_{j,k} \cdot x_j - b_k\right)}$$

## Example: Single-hidden-layer perceptron



$$h_k = \frac{1}{1 + \exp\left(-\sum_{j=1}^n v_{j,k} \cdot x_j - b_k\right)}$$

## Example: Single-hidden-layer perceptron

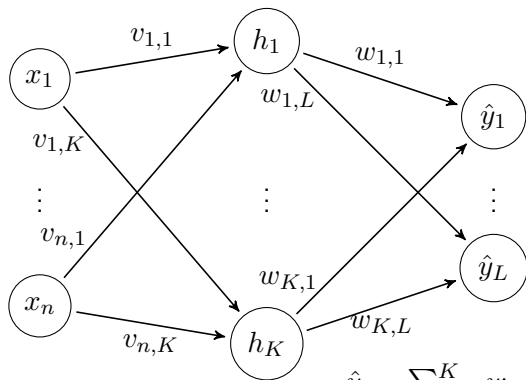


$$\hat{y}_l = \sum_{k=1}^K w_{k,l} \cdot h_k + c_l$$

$$h_k = \frac{1}{1 + \exp\left(-\sum_{j=1}^n v_{j,k} \cdot x_j - b_k\right)}$$

$$\ell = \sum_{l=1}^L (y_l - \hat{y}_l)^2$$

## Example: Single-hidden-layer perceptron

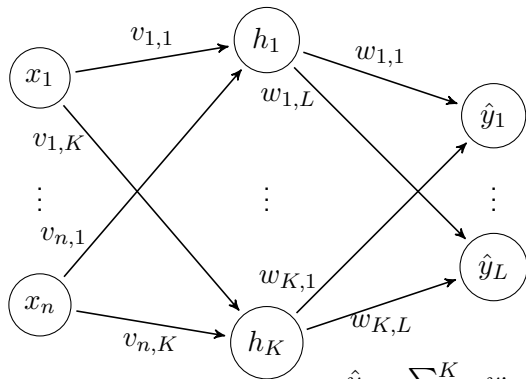


$$\hat{y}_l = \sum_{k=1}^K w_{k,l} \cdot h_k + c_l$$

$$h_k = \frac{1}{1 + \exp\left(-\sum_{j=1}^n v_{j,k} \cdot x_j - b_k\right)}$$

$$\ell = \sum_{l=1}^L (y_l - \hat{y}_l)^2$$
$$\frac{\partial \ell}{\partial \hat{y}_l} =$$

## Example: Single-hidden-layer perceptron

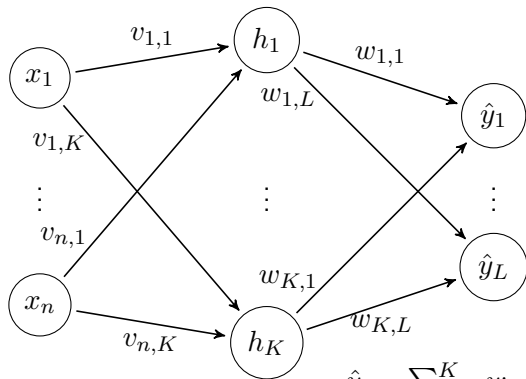


$$\hat{y}_l = \sum_{k=1}^K w_{k,l} \cdot h_k + c_l$$

$$h_k = \frac{1}{1 + \exp\left(-\sum_{j=1}^n v_{j,k} \cdot x_j - b_k\right)}$$

$$\ell = \sum_{l=1}^L (y_l - \hat{y}_l)^2$$
$$\frac{\partial \ell}{\partial \hat{y}_l} = -2 \cdot (y_l - \hat{y}_l)$$

## Example: Single-hidden-layer perceptron

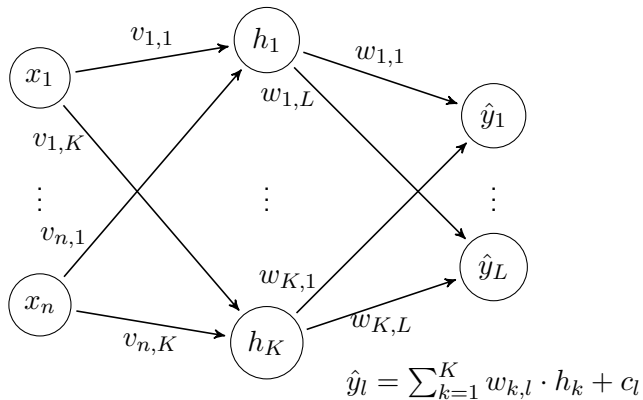


$$\hat{y}_l = \sum_{k=1}^K w_{k,l} \cdot h_k + c_l$$

$$h_k = \frac{1}{1 + \exp\left(-\sum_{j=1}^n v_{j,k} \cdot x_j - b_k\right)}$$

$$\ell = \sum_{l=1}^L (y_l - \hat{y}_l)^2$$
$$\frac{\partial \ell}{\partial \hat{y}_l} = -2 \cdot (y_l - \hat{y}_l)$$
$$\frac{\partial \ell}{\partial w_{k,l}} =$$

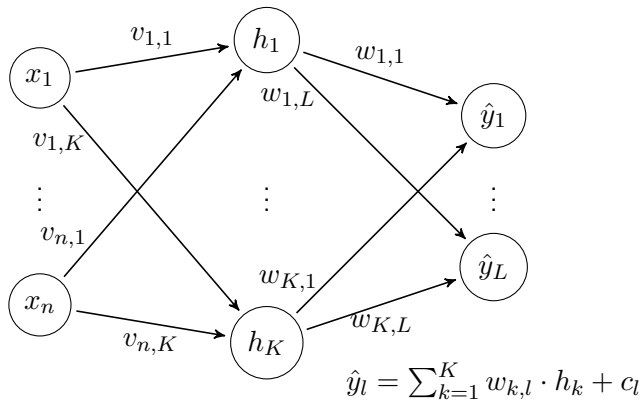
## Example: Single-hidden-layer perceptron



$$h_k = \frac{1}{1 + \exp\left(-\sum_{j=1}^n v_{j,k} \cdot x_j - b_k\right)}$$

$$\ell = \sum_{l=1}^L (y_l - \hat{y}_l)^2$$
$$\frac{\partial \ell}{\partial \hat{y}_l} = -2 \cdot (y_l - \hat{y}_l)$$
$$\frac{\partial \ell}{\partial w_{k,l}} = \frac{\partial \hat{y}_l}{\partial w_{k,l}} \cdot \frac{\partial \ell}{\partial \hat{y}_l}$$

## Example: Single-hidden-layer perceptron

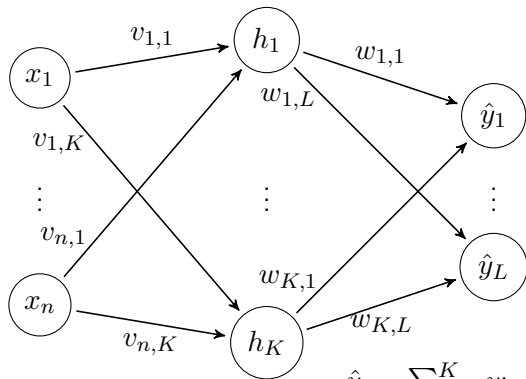


$$h_k = \frac{1}{1 + \exp\left(-\sum_{j=1}^n v_{j,k} \cdot x_j - b_k\right)}$$

$$\begin{aligned}\ell &= \sum_{l=1}^L (y_l - \hat{y}_l)^2 \\ \frac{\partial \ell}{\partial \hat{y}_l} &= -2 \cdot (y_l - \hat{y}_l) \\ \frac{\partial \ell}{\partial w_{k,l}} &= h_k \cdot \frac{\partial \ell}{\partial \hat{y}_l}\end{aligned}$$



## Example: Single-hidden-layer perceptron



$$\hat{y}_l = \sum_{k=1}^K w_{k,l} \cdot h_k + c_l$$

$$h_k = \frac{1}{1 + \exp\left(-\sum_{j=1}^n v_{j,k} \cdot x_j - b_k\right)}$$

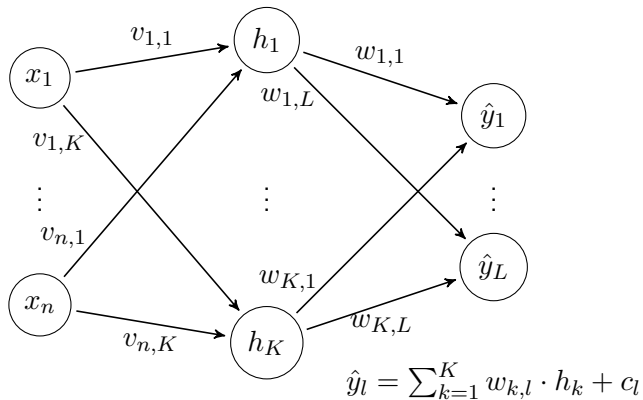
$$\ell = \sum_{l=1}^L (y_l - \hat{y}_l)^2$$

$$\frac{\partial \ell}{\partial \hat{y}_l} = -2 \cdot (y_l - \hat{y}_l)$$

$$\frac{\partial \ell}{\partial w_{k,l}} = h_k \cdot \frac{\partial \ell}{\partial \hat{y}_l}$$

$$\frac{\partial \ell}{\partial h_k} =$$

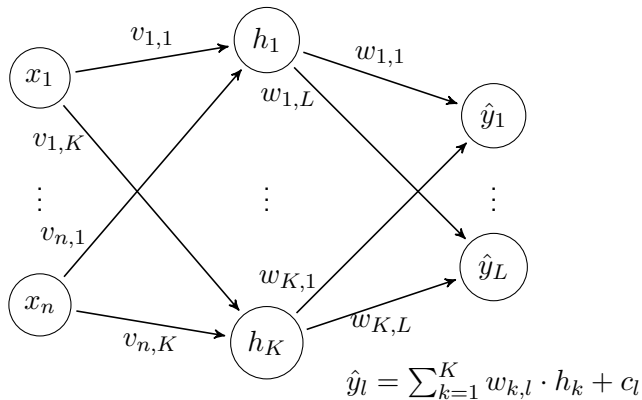
## Example: Single-hidden-layer perceptron



$$h_k = \frac{1}{1 + \exp\left(-\sum_{j=1}^n v_{j,k} \cdot x_j - b_k\right)}$$

$$\begin{aligned}\ell &= \sum_{l=1}^L (y_l - \hat{y}_l)^2 \\ \frac{\partial \ell}{\partial \hat{y}_l} &= -2 \cdot (y_l - \hat{y}_l) \\ \frac{\partial \ell}{\partial w_{k,l}} &= h_k \cdot \frac{\partial \ell}{\partial \hat{y}_l} \\ \frac{\partial \ell}{\partial h_k} &= \sum_{l=1}^L \frac{\partial \hat{y}_l}{\partial h_k} \cdot \frac{\partial \ell}{\partial \hat{y}_l}\end{aligned}$$

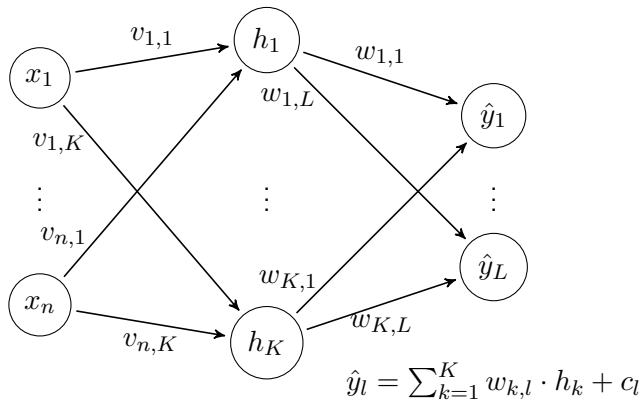
## Example: Single-hidden-layer perceptron



$$h_k = \frac{1}{1 + \exp\left(-\sum_{j=1}^n v_{j,k} \cdot x_j - b_k\right)}$$

$$\begin{aligned}\ell &= \sum_{l=1}^L (y_l - \hat{y}_l)^2 \\ \frac{\partial \ell}{\partial \hat{y}_l} &= -2 \cdot (y_l - \hat{y}_l) \\ \frac{\partial \ell}{\partial w_{k,l}} &= h_k \cdot \frac{\partial \ell}{\partial \hat{y}_l} \\ \frac{\partial \ell}{\partial h_k} &= \sum_{l=1}^L w_{k,l} \cdot \frac{\partial \ell}{\partial \hat{y}_l}\end{aligned}$$

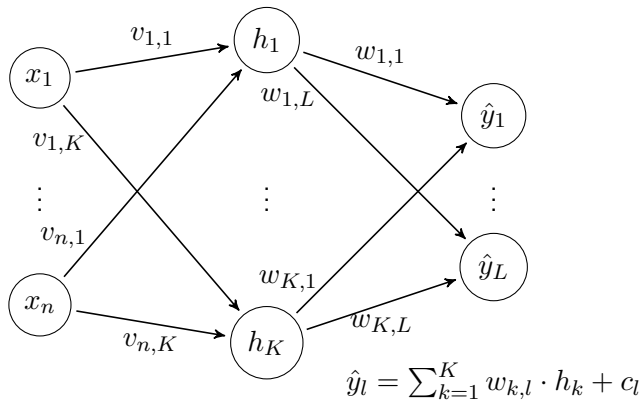
## Example: Single-hidden-layer perceptron



$$h_k = \frac{1}{1 + \exp\left(-\sum_{j=1}^n v_{j,k} \cdot x_j - b_k\right)}$$

$$\begin{aligned}\ell &= \sum_{l=1}^L (y_l - \hat{y}_l)^2 \\ \frac{\partial \ell}{\partial \hat{y}_l} &= -2 \cdot (y_l - \hat{y}_l) \\ \frac{\partial \ell}{\partial w_{k,l}} &= h_k \cdot \frac{\partial \ell}{\partial \hat{y}_l} \\ \frac{\partial \ell}{\partial h_k} &= \sum_{l=1}^L w_{k,l} \cdot \frac{\partial \ell}{\partial \hat{y}_l} \\ \frac{\partial \ell}{\partial v_{j,k}} &= \end{aligned}$$

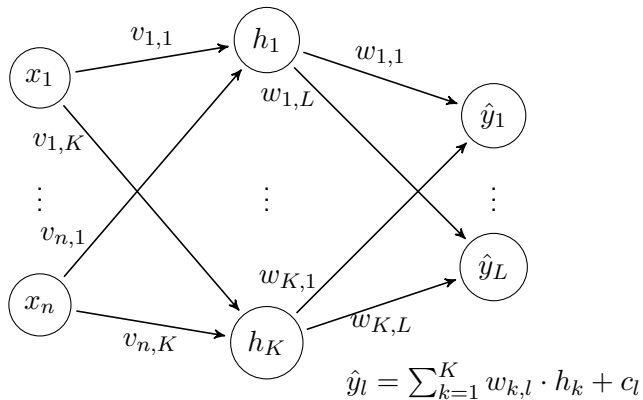
## Example: Single-hidden-layer perceptron



$$h_k = \frac{1}{1 + \exp\left(-\sum_{j=1}^n v_{j,k} \cdot x_j - b_k\right)}$$

$$\begin{aligned}\ell &= \sum_{l=1}^L (y_l - \hat{y}_l)^2 \\ \frac{\partial \ell}{\partial \hat{y}_l} &= -2 \cdot (y_l - \hat{y}_l) \\ \frac{\partial \ell}{\partial w_{k,l}} &= h_k \cdot \frac{\partial \ell}{\partial \hat{y}_l} \\ \frac{\partial \ell}{\partial h_k} &= \sum_{l=1}^L w_{k,l} \cdot \frac{\partial \ell}{\partial \hat{y}_l} \\ \frac{\partial \ell}{\partial v_{j,k}} &= \frac{\partial h_k}{\partial v_{j,k}} \cdot \frac{\partial \ell}{\partial h_k}\end{aligned}$$

## Example: Single-hidden-layer perceptron



$$h_k = \frac{1}{1 + \exp\left(-\sum_{j=1}^n v_{j,k} \cdot x_j - b_k\right)}$$

$$\ell = \sum_{l=1}^L (y_l - \hat{y}_l)^2$$

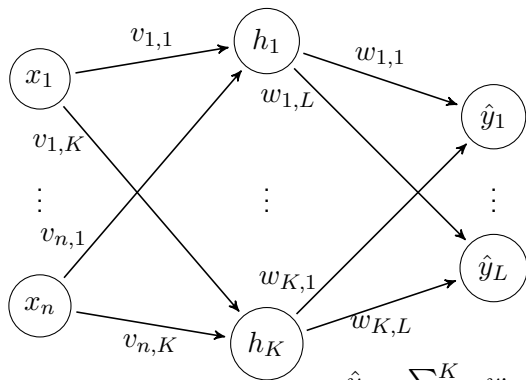
$$\frac{\partial \ell}{\partial \hat{y}_l} = -2 \cdot (y_l - \hat{y}_l)$$

$$\frac{\partial \ell}{\partial w_{k,l}} = h_k \cdot \frac{\partial \ell}{\partial \hat{y}_l}$$

$$\frac{\partial \ell}{\partial h_k} = \sum_{l=1}^L w_{k,l} \cdot \frac{\partial \ell}{\partial \hat{y}_l}$$

$$\frac{\partial \ell}{\partial v_{j,k}} = h_k \cdot (1 - h_k) \cdot x_j \cdot \frac{\partial \ell}{\partial h_k}$$

## Example: Single-hidden-layer perceptron



$$h_k = \frac{1}{1 + \exp\left(-\sum_{j=1}^n v_{j,k} \cdot x_j - b_k\right)}$$

$$\ell = \sum_{l=1}^L (y_l - \hat{y}_l)^2$$

$$\frac{\partial \ell}{\partial \hat{y}_l} = -2 \cdot (y_l - \hat{y}_l)$$

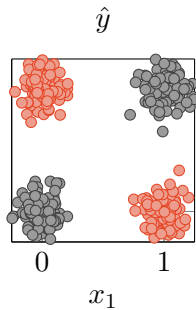
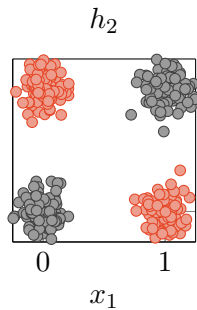
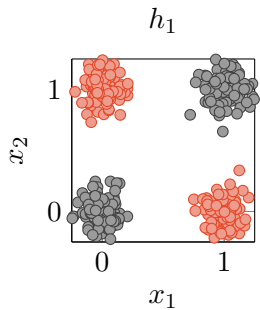
$$\frac{\partial \ell}{\partial w_{k,l}} = h_k \cdot \frac{\partial \ell}{\partial \hat{y}_l}$$

$$\frac{\partial \ell}{\partial h_k} = \sum_{l=1}^L w_{k,l} \cdot \frac{\partial \ell}{\partial \hat{y}_l}$$

$$\frac{\partial \ell}{\partial v_{j,k}} = h_k \cdot (1 - h_k) \cdot x_j \cdot \frac{\partial \ell}{\partial h_k}$$

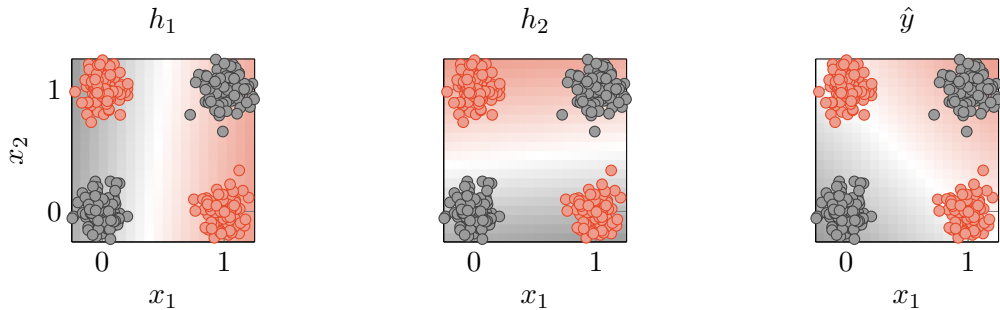
► Recall: One hidden layer is enough for universal approximation!

## Example: XOR with single-hidden-layer perceptron

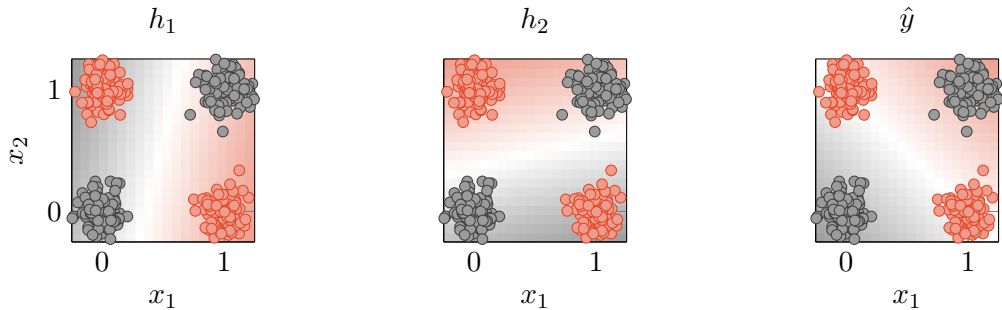




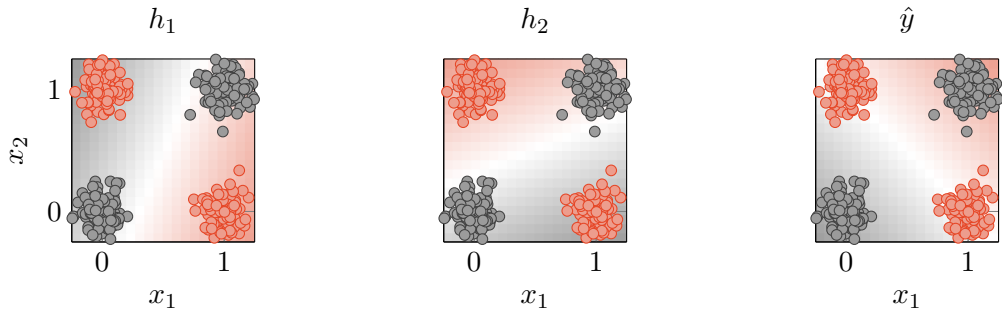
## Example: XOR with single-hidden-layer perceptron



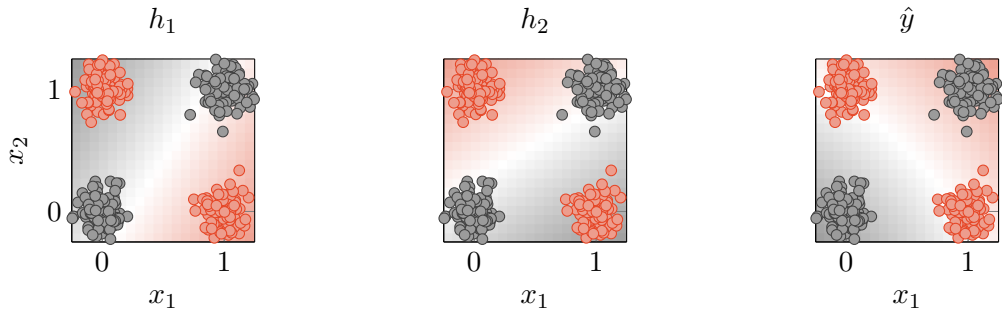
## Example: XOR with single-hidden-layer perceptron



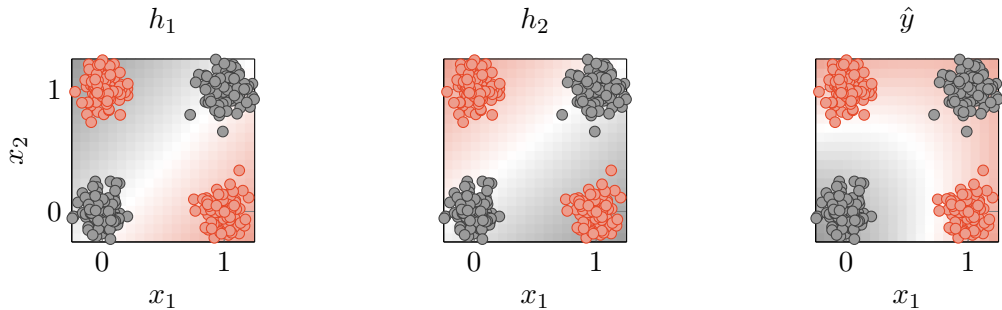
## Example: XOR with single-hidden-layer perceptron



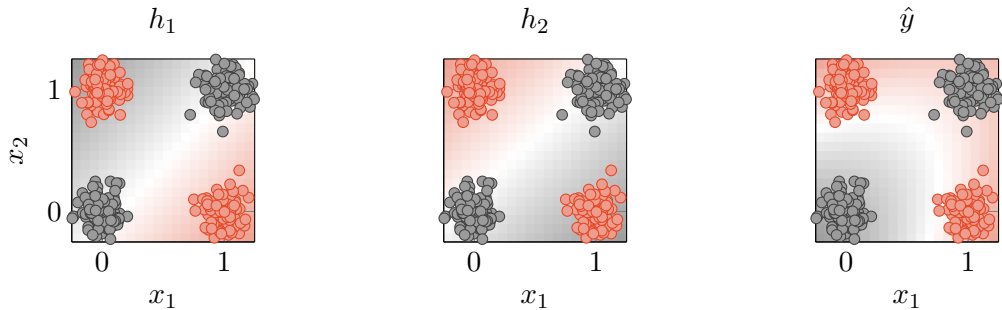
## Example: XOR with single-hidden-layer perceptron



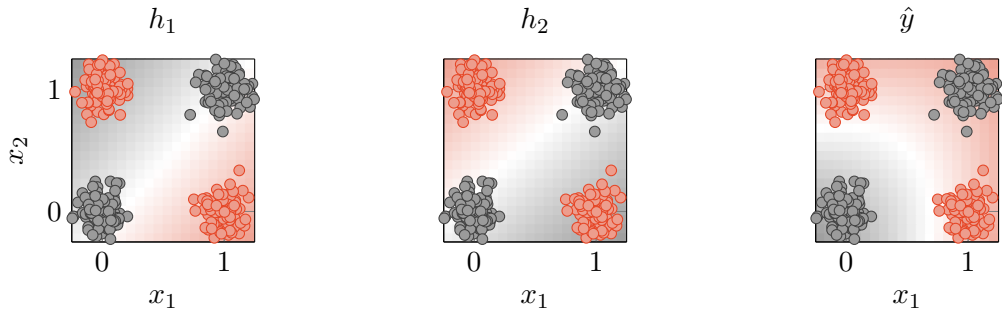
## Example: XOR with single-hidden-layer perceptron



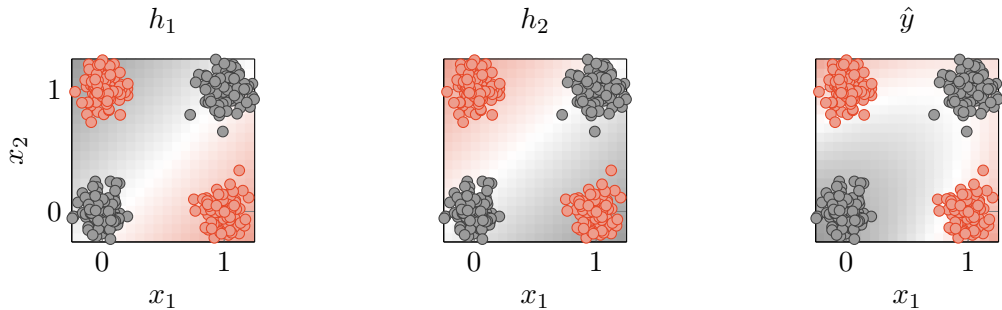
## Example: XOR with single-hidden-layer perceptron



## Example: XOR with single-hidden-layer perceptron

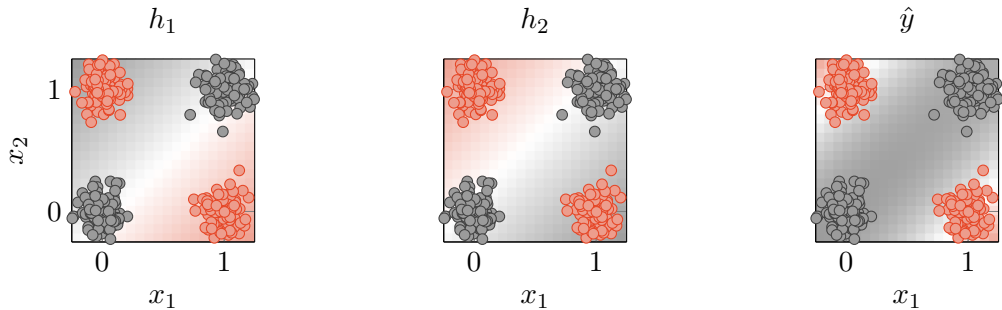


## Example: XOR with single-hidden-layer perceptron





## Example: XOR with single-hidden-layer perceptron



# Recurrent Nets

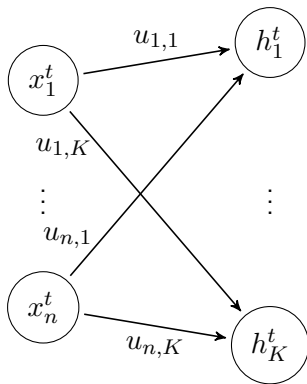
(Hopfield 1982)

- ▶ Idea: Process **dynamic** data (i.e. changes over time) via **feedback loops**

# Recurrent Nets

(Hopfield 1982)

- Idea: Process **dynamic** data (i.e. changes over time) via **feedback loops**

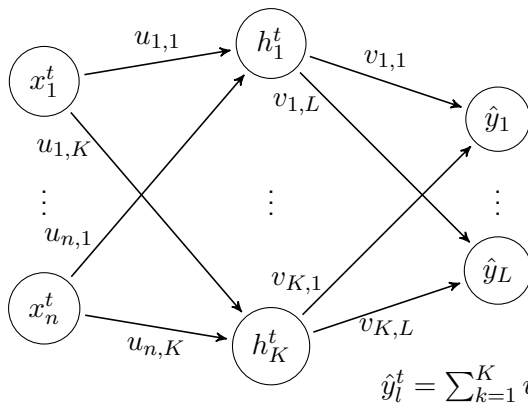


$$h_k^t = \sigma \left( \sum_{j=1}^n u_{j,k} \cdot x_j^t \right)$$

# Recurrent Nets

(Hopfield 1982)

- Idea: Process **dynamic** data (i.e. changes over time) via **feedback loops**

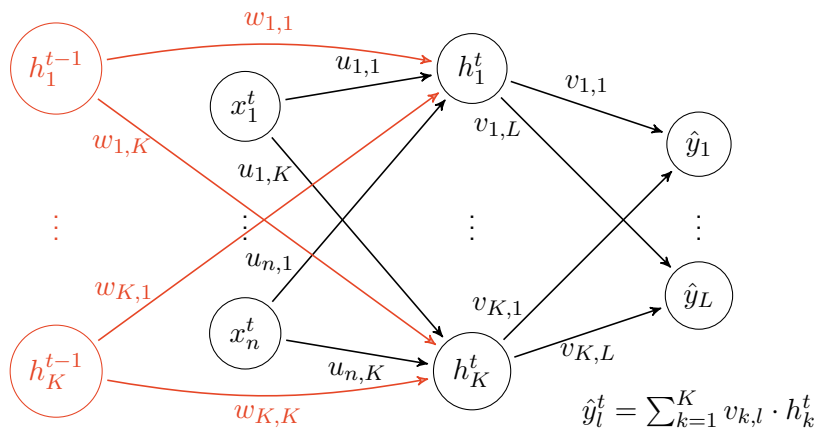


$$h_k^t = \sigma \left( \sum_{j=1}^n u_{j,k} \cdot x_j^t \right)$$

# Recurrent Nets

(Hopfield 1982)

- Idea: Process **dynamic** data (i.e. changes over time) via **feedback loops**

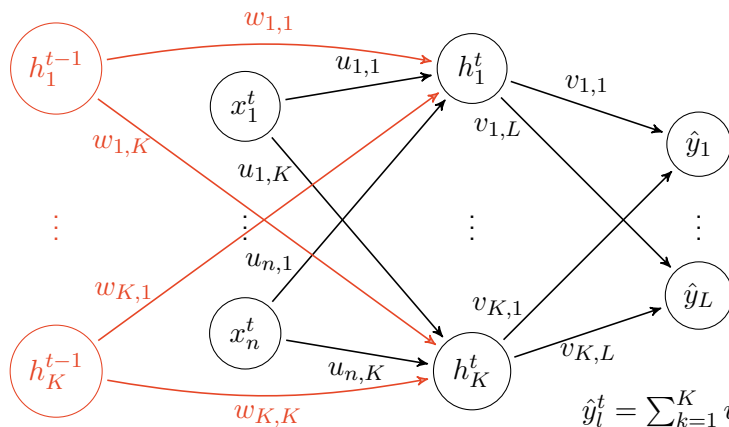


$$h_k^t = \sigma \left( \sum_{j=1}^n u_{j,k} \cdot x_j^t + \sum_{k'=1}^K w_{k',k} \cdot h_{k'}^{t-1} \right)$$

# Recurrent Nets

(Hopfield 1982)

- Idea: Process **dynamic** data (i.e. changes over time) via **feedback loops**



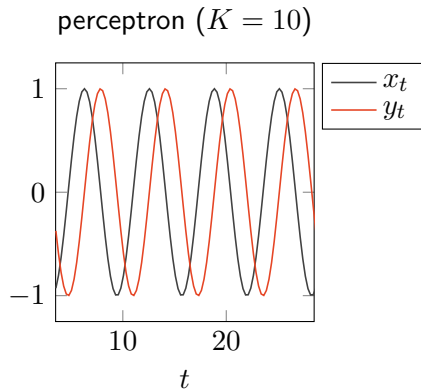
$$\vec{h}^t = \sigma(\mathbf{U}^T \cdot \vec{x}^t + \mathbf{W}^T \cdot \vec{h}^{t-1})$$

$$\vec{y}^t = \mathbf{V}^T \cdot \vec{h}^t$$

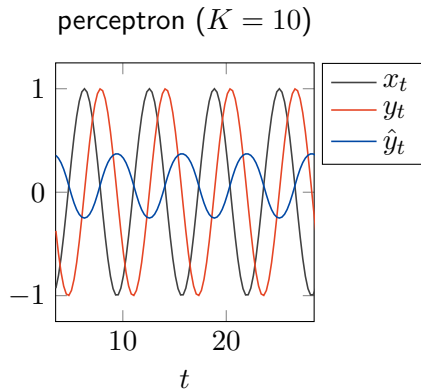
$$\hat{y}_l^t = \sum_{k=1}^K v_{k,l} \cdot h_k^t$$

$$h_k^t = \sigma\left(\sum_{j=1}^n u_{j,k} \cdot x_j^t + \sum_{k'=1}^K w_{k',k} \cdot h_{k'}^{t-1}\right)$$

## Example: Cosine-to-Sine prediction

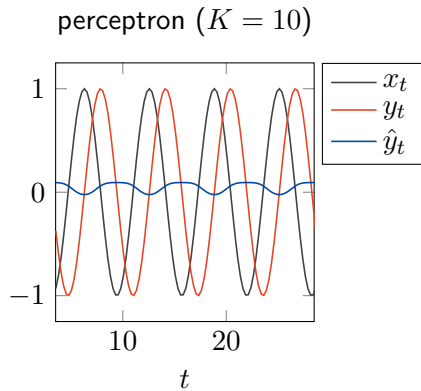


## Example: Cosine-to-Sine prediction

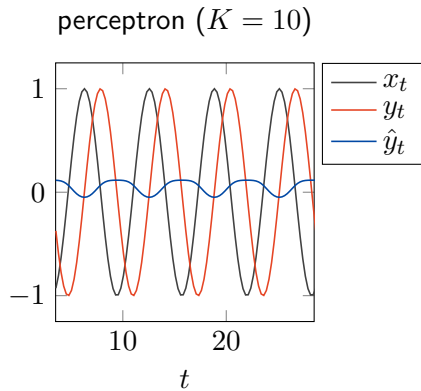




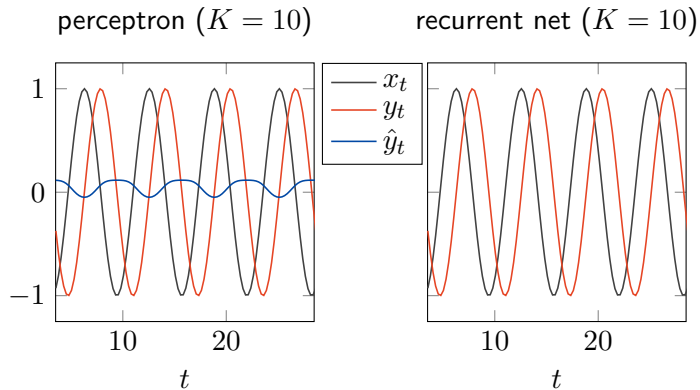
## Example: Cosine-to-Sine prediction



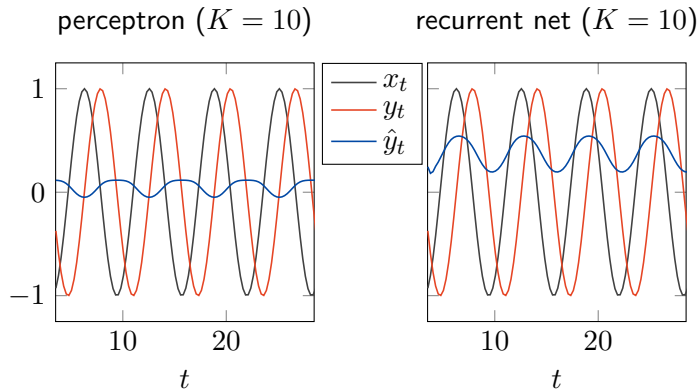
## Example: Cosine-to-Sine prediction



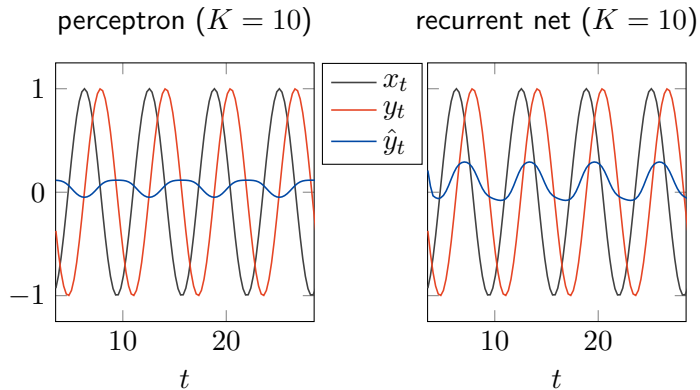
## Example: Cosine-to-Sine prediction



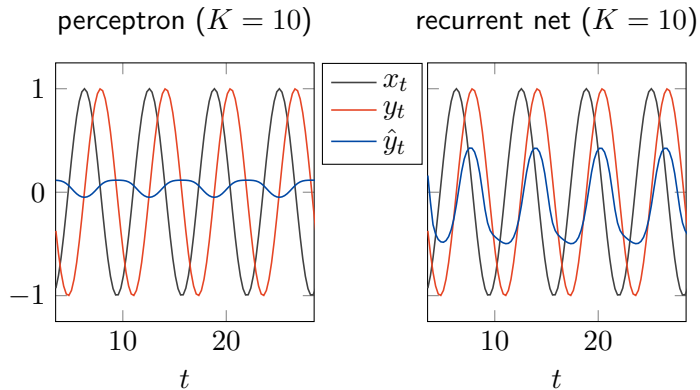
## Example: Cosine-to-Sine prediction



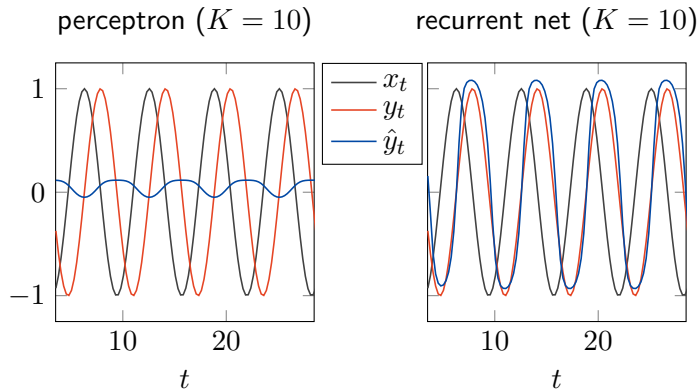
## Example: Cosine-to-Sine prediction



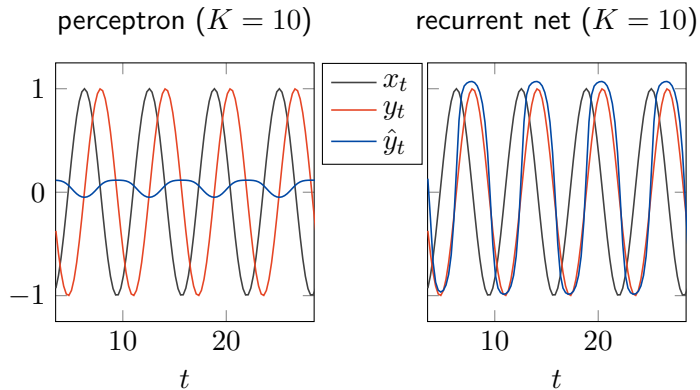
## Example: Cosine-to-Sine prediction



## Example: Cosine-to-Sine prediction

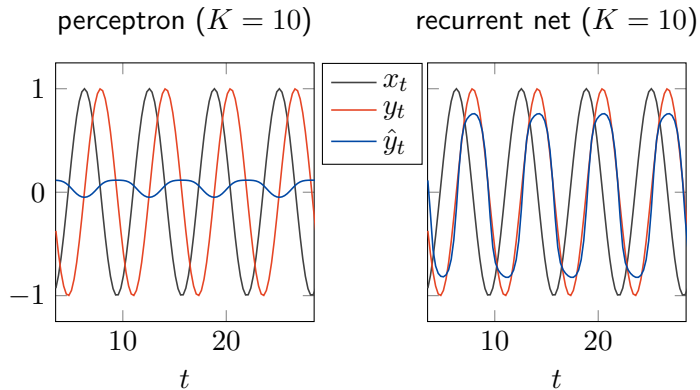


## Example: Cosine-to-Sine prediction



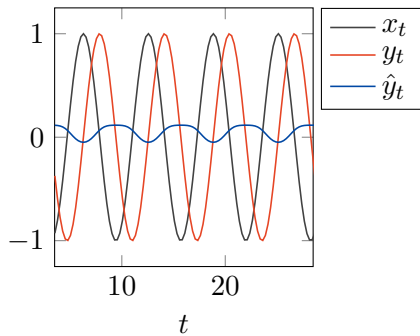


## Example: Cosine-to-Sine prediction

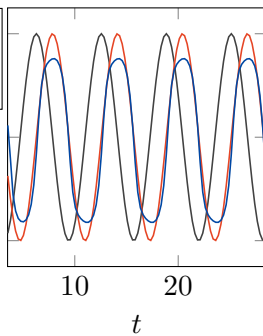


## Example: Cosine-to-Sine prediction

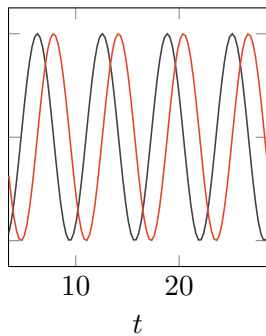
perceptron ( $K = 10$ )



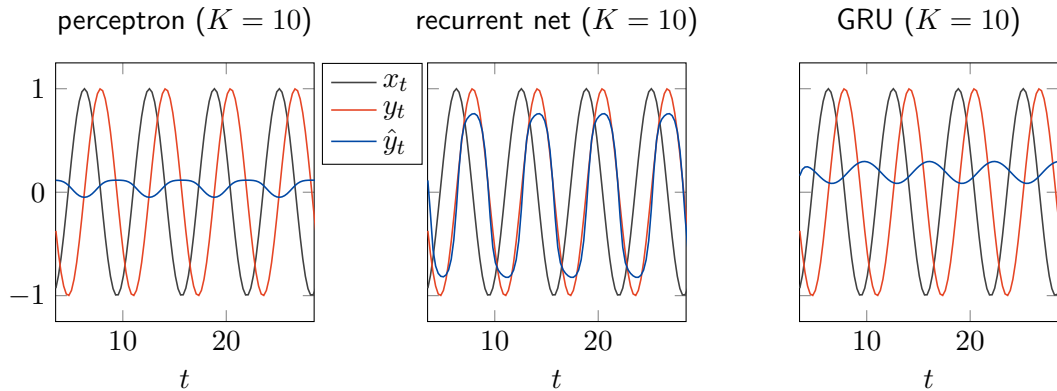
recurrent net ( $K = 10$ )



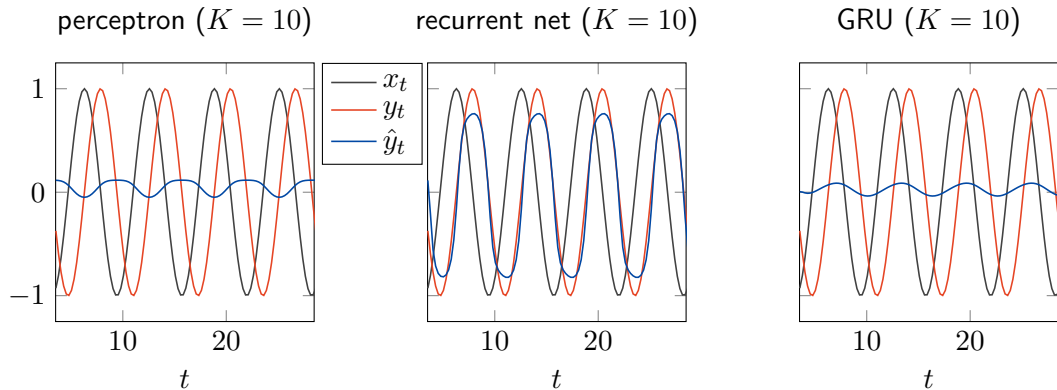
GRU ( $K = 10$ )



## Example: Cosine-to-Sine prediction

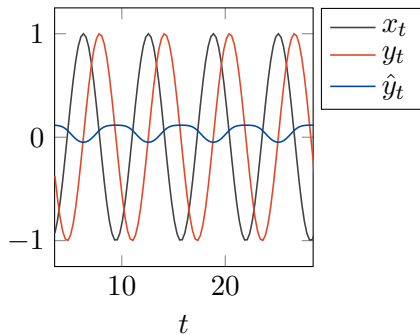


## Example: Cosine-to-Sine prediction

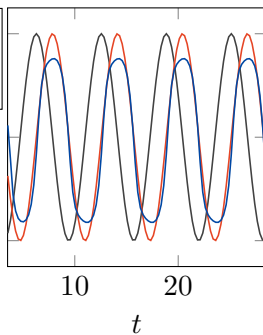


## Example: Cosine-to-Sine prediction

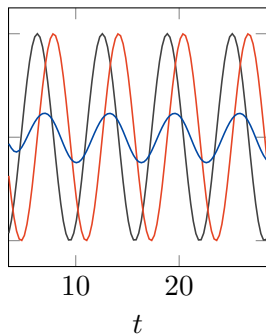
perceptron ( $K = 10$ )



recurrent net ( $K = 10$ )

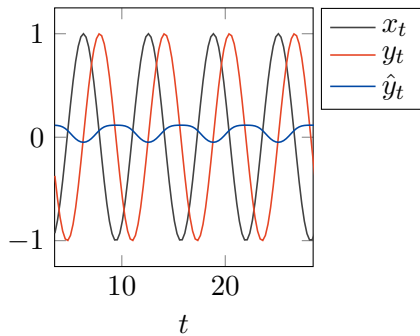


GRU ( $K = 10$ )

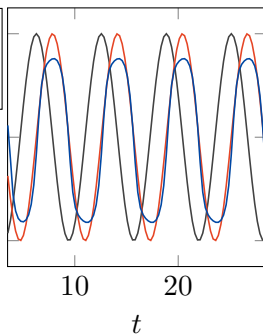


## Example: Cosine-to-Sine prediction

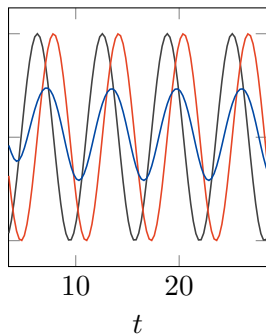
perceptron ( $K = 10$ )



recurrent net ( $K = 10$ )

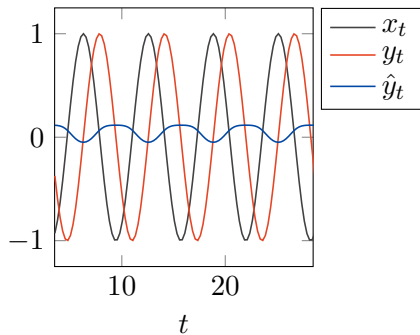


GRU ( $K = 10$ )

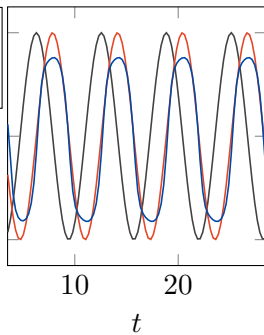


## Example: Cosine-to-Sine prediction

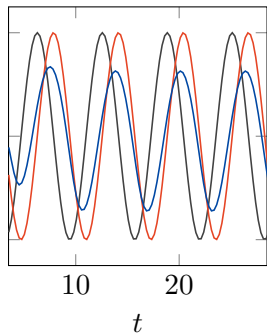
perceptron ( $K = 10$ )



recurrent net ( $K = 10$ )

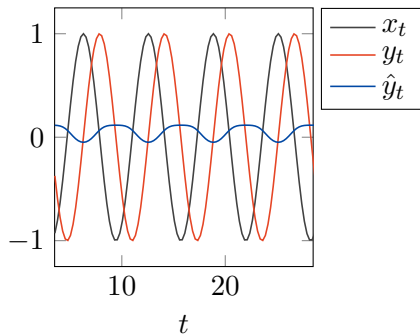


GRU ( $K = 10$ )

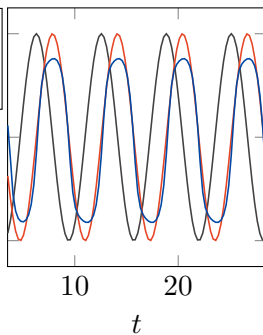


## Example: Cosine-to-Sine prediction

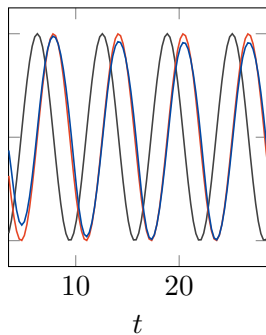
perceptron ( $K = 10$ )



recurrent net ( $K = 10$ )



GRU ( $K = 10$ )





## Backpropagation through time

(Werbos 1990)

- ▶ “Just” backpropagation with a graph ‘unrolled’ over time

## Backpropagation through time

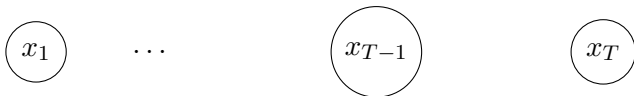
(Werbos 1990)

- ▶ “Just” backpropagation with a graph ‘unrolled’ over time
- ▶ Example here:  $n = K = L = 1$ ; recurrent weight  $w$  treated as explicit node

## Backpropagation through time

(Werbos 1990)

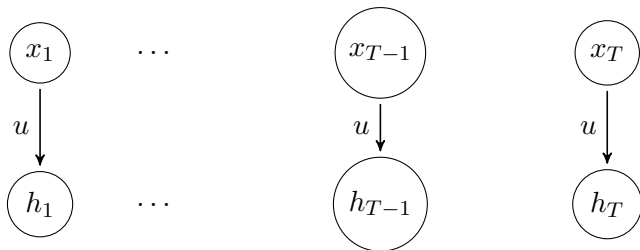
- ▶ “Just” backpropagation with a graph ‘unrolled’ over time
- ▶ Example here:  $n = K = L = 1$ ; recurrent weight  $w$  treated as explicit node



## Backpropagation through time

(Werbos 1990)

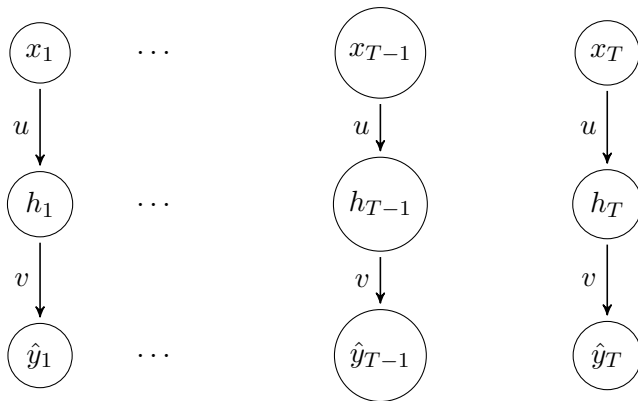
- ▶ “Just” backpropagation with a graph ‘unrolled’ over time
- ▶ Example here:  $n = K = L = 1$ ; recurrent weight  $w$  treated as explicit node



## Backpropagation through time

(Werbos 1990)

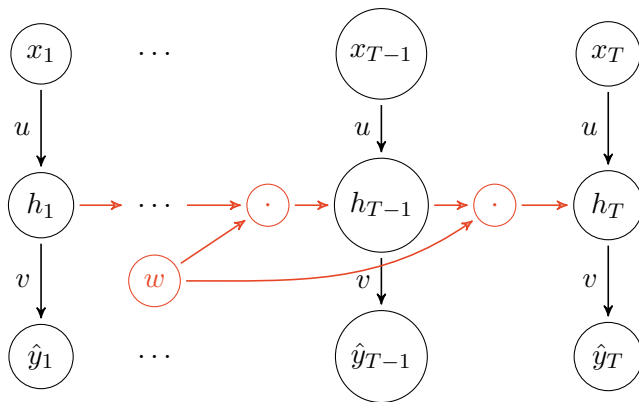
- ▶ “Just” backpropagation with a graph ‘unrolled’ over time
- ▶ Example here:  $n = K = L = 1$ ; recurrent weight  $w$  treated as explicit node



# Backpropagation through time

(Werbos 1990)

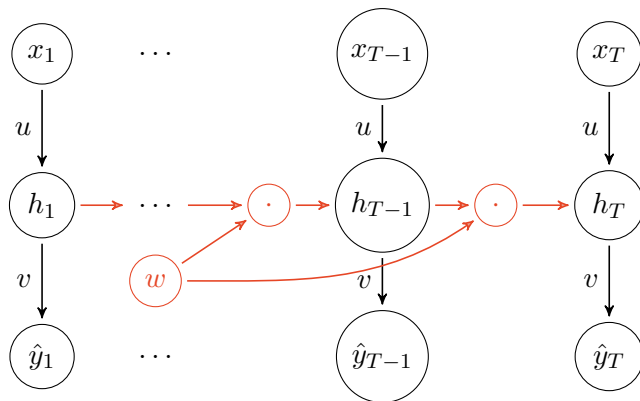
- ▶ “Just” backpropagation with a graph ‘unrolled’ over time
- ▶ Example here:  $n = K = L = 1$ ; recurrent weight  $w$  treated as explicit node



# Backpropagation through time

(Werbos 1990)

- ▶ “Just” backpropagation with a graph ‘unrolled’ over time
- ▶ Example here:  $n = K = L = 1$ ; recurrent weight  $w$  treated as explicit node

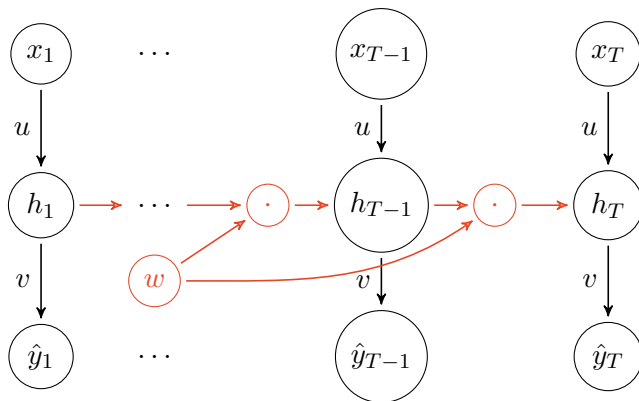


$$\ell_T = (y_T - \hat{y}_T)^2$$

# Backpropagation through time

(Werbos 1990)

- ▶ “Just” backpropagation with a graph ‘unrolled’ over time
- ▶ Example here:  $n = K = L = 1$ ; recurrent weight  $w$  treated as explicit node



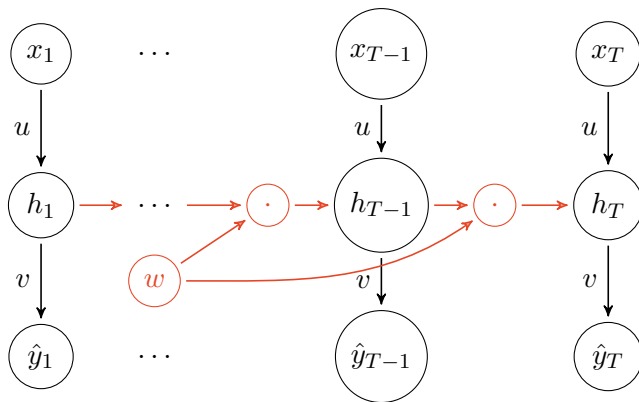
$$\ell_T = (y_T - \hat{y}_T)^2$$
$$\frac{\partial \ell_T}{\partial \hat{y}_T} =$$



# Backpropagation through time

(Werbos 1990)

- ▶ “Just” backpropagation with a graph ‘unrolled’ over time
- ▶ Example here:  $n = K = L = 1$ ; recurrent weight  $w$  treated as explicit node

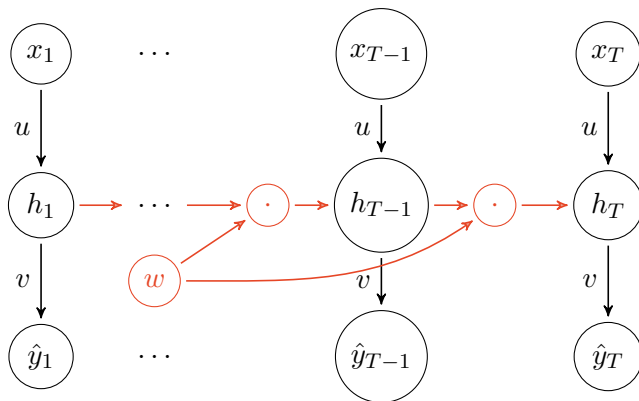


$$\ell_T = (y_T - \hat{y}_T)^2$$
$$\frac{\partial \ell_T}{\partial \hat{y}_T} = -2 \cdot (y_T - \hat{y}_T)$$

# Backpropagation through time

(Werbos 1990)

- ▶ “Just” backpropagation with a graph ‘unrolled’ over time
- ▶ Example here:  $n = K = L = 1$ ; recurrent weight  $w$  treated as explicit node

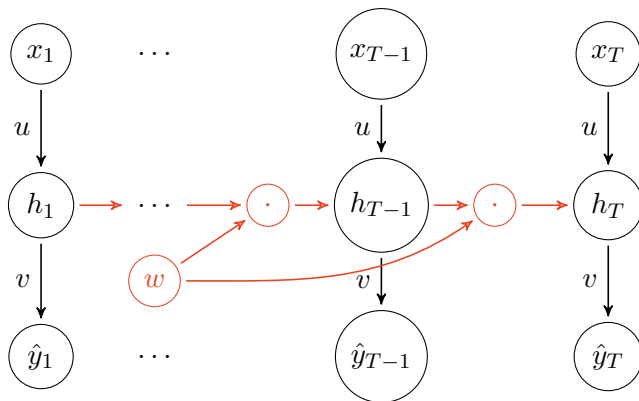


$$\begin{aligned}\ell_T &= (y_T - \hat{y}_T)^2 \\ \frac{\partial \ell_T}{\partial \hat{y}_T} &= -2 \cdot (y_T - \hat{y}_T) \\ \frac{\partial \ell_T}{\partial v} &= h_T \cdot \frac{\partial \ell_T}{\partial \hat{y}_T}\end{aligned}$$

# Backpropagation through time

(Werbos 1990)

- ▶ “Just” backpropagation with a graph ‘unrolled’ over time
- ▶ Example here:  $n = K = L = 1$ ; recurrent weight  $w$  treated as explicit node

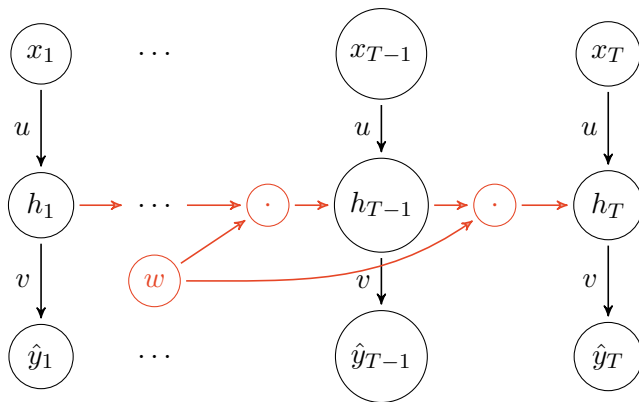


$$\begin{aligned}\ell_T &= (y_T - \hat{y}_T)^2 \\ \frac{\partial \ell_T}{\partial \hat{y}_T} &= -2 \cdot (y_T - \hat{y}_T) \\ \frac{\partial \ell_T}{\partial v} &= h_T \cdot \frac{\partial \ell_T}{\partial \hat{y}_T} \\ \frac{\partial \ell_T}{\partial h_T} &= v \cdot \frac{\partial \ell_T}{\partial \hat{y}_T}\end{aligned}$$

# Backpropagation through time

(Werbos 1990)

- ▶ “Just” backpropagation with a graph ‘unrolled’ over time
- ▶ Example here:  $n = K = L = 1$ ; recurrent weight  $w$  treated as explicit node

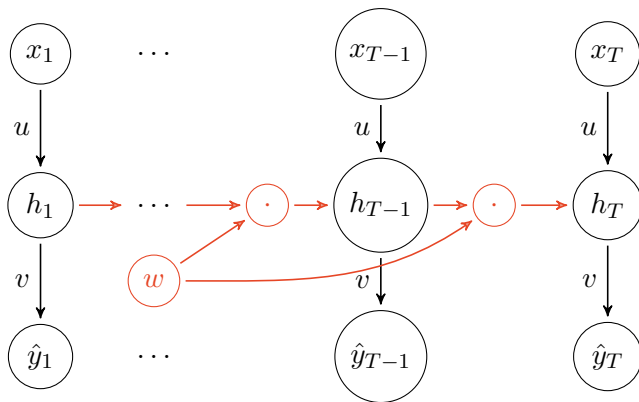


$$\begin{aligned}\ell_T &= (y_T - \hat{y}_T)^2 \\ \frac{\partial \ell_T}{\partial \hat{y}_T} &= -2 \cdot (y_T - \hat{y}_T) \\ \frac{\partial \ell_T}{\partial v} &= h_T \cdot \frac{\partial \ell_T}{\partial \hat{y}_T} \\ \frac{\partial \ell_T}{\partial h_T} &= v \cdot \frac{\partial \ell_T}{\partial \hat{y}_T} \\ \frac{\partial \ell_T}{\partial w} &= \sum_{t=1}^T \frac{\partial h_t}{\partial w} \cdot \frac{\partial \ell_T}{\partial h_t}\end{aligned}$$

# Backpropagation through time

(Werbos 1990)

- ▶ “Just” backpropagation with a graph ‘unrolled’ over time
- ▶ Example here:  $n = K = L = 1$ ; recurrent weight  $w$  treated as explicit node

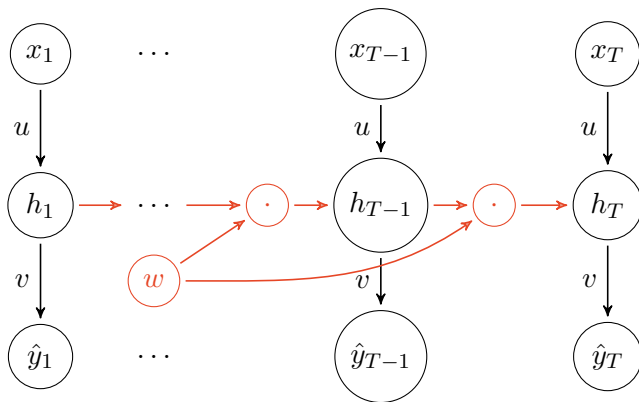


$$\begin{aligned}\ell_T &= (y_T - \hat{y}_T)^2 \\ \frac{\partial \ell_T}{\partial \hat{y}_T} &= -2 \cdot (y_T - \hat{y}_T) \\ \frac{\partial \ell_T}{\partial v} &= h_T \cdot \frac{\partial \ell_T}{\partial \hat{y}_T} \\ \frac{\partial \ell_T}{\partial h_T} &= v \cdot \frac{\partial \ell_T}{\partial \hat{y}_T} \\ \frac{\partial \ell_T}{\partial w} &= \sum_{t=1}^T h_{t-1} \cdot \sigma'_t \cdot \frac{\partial \ell_T}{\partial h_t}\end{aligned}$$

# Backpropagation through time

(Werbos 1990)

- ▶ “Just” backpropagation with a graph ‘unrolled’ over time
- ▶ Example here:  $n = K = L = 1$ ; recurrent weight  $w$  treated as explicit node

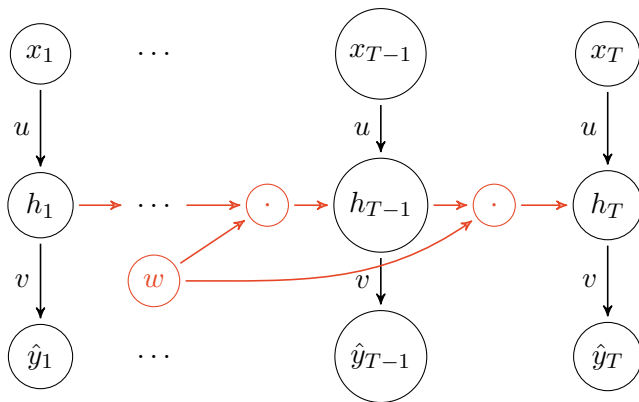


$$\begin{aligned}\ell_T &= (y_T - \hat{y}_T)^2 \\ \frac{\partial \ell_T}{\partial \hat{y}_T} &= -2 \cdot (y_T - \hat{y}_T) \\ \frac{\partial \ell_T}{\partial v} &= h_T \cdot \frac{\partial \ell_T}{\partial \hat{y}_T} \\ \frac{\partial \ell_T}{\partial h_T} &= v \cdot \frac{\partial \ell_T}{\partial \hat{y}_T} \\ \frac{\partial \ell_T}{\partial w} &= \sum_{t=1}^T h_{t-1} \cdot \sigma'_t \cdot \frac{\partial \ell_T}{\partial h_t} \\ \frac{\partial \ell_T}{\partial h_t} &= \frac{\partial h_{t+1}}{\partial h_t} \cdot \frac{\partial \ell_T}{\partial h_{t+1}}\end{aligned}$$

# Backpropagation through time

(Werbos 1990)

- ▶ “Just” backpropagation with a graph ‘unrolled’ over time
- ▶ Example here:  $n = K = L = 1$ ; recurrent weight  $w$  treated as explicit node



$$\ell_T = (y_T - \hat{y}_T)^2$$
$$\frac{\partial \ell_T}{\partial \hat{y}_T} = -2 \cdot (y_T - \hat{y}_T)$$

$$\frac{\partial \ell_T}{\partial v} = h_T \cdot \frac{\partial \ell_T}{\partial \hat{y}_T}$$

$$\frac{\partial \ell_T}{\partial h_T} = v \cdot \frac{\partial \ell_T}{\partial \hat{y}_T}$$

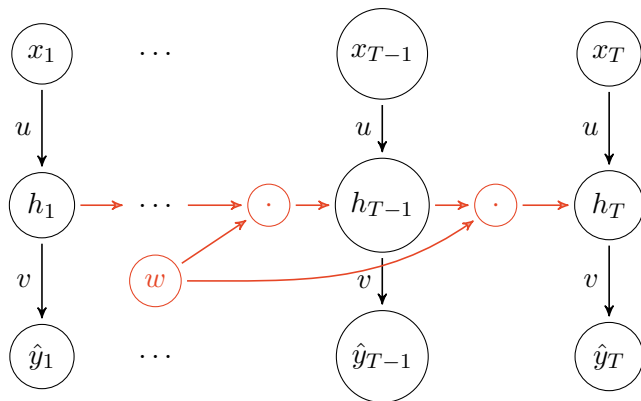
$$\frac{\partial \ell_T}{\partial w} = \sum_{t=1}^T h_{t-1} \cdot \sigma'_t \cdot \frac{\partial \ell_T}{\partial h_t}$$

$$\frac{\partial \ell_T}{\partial h_t} = w \cdot \sigma'_t \cdot \frac{\partial \ell_T}{\partial h_{t+1}}$$

# Backpropagation through time

(Werbos 1990)

- ▶ “Just” backpropagation with a graph ‘unrolled’ over time
- ▶ Example here:  $n = K = L = 1$ ; recurrent weight  $w$  treated as explicit node



$$\ell_T = (y_T - \hat{y}_T)^2$$
$$\frac{\partial \ell_T}{\partial \hat{y}_T} = -2 \cdot (y_T - \hat{y}_T)$$

$$\frac{\partial \ell_T}{\partial v} = h_T \cdot \frac{\partial \ell_T}{\partial \hat{y}_T}$$

$$\frac{\partial \ell_T}{\partial h_T} = v \cdot \frac{\partial \ell_T}{\partial \hat{y}_T}$$

$$\frac{\partial \ell_T}{\partial w} = \sum_{t=1}^T h_{t-1} \cdot \sigma'_t \cdot \frac{\partial \ell_T}{\partial h_t}$$

$$\frac{\partial \ell_T}{\partial h_t} = \left( \prod_{\tau=t}^{T-1} w \cdot \sigma'_\tau \right) \cdot \frac{\partial \ell_T}{\partial h_T}$$



## Long Short-Term Memory Networks (Hochreiter and Schmidhuber 1997)

- ▶ Motivation: The term  $\prod_{\tau=t}^{T-1} w \cdot \sigma'_t$  in backprop through time can easily vanish (or explode)  $\Rightarrow$  hard to learn for long-term memory effects

## Long Short-Term Memory Networks (Hochreiter and Schmidhuber 1997)

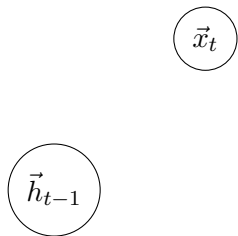
- ▶ Motivation: The term  $\prod_{\tau=t}^{T-1} w \cdot \sigma'_t$  in backprop through time can easily vanish (or explode)  $\Rightarrow$  hard to learn for long-term memory effects
- ▶ Idea: Let's build an explicit longer-term memory into the model

## Long Short-Term Memory Networks (Hochreiter and Schmidhuber 1997)

- ▶ Motivation: The term  $\prod_{\tau=t}^{T-1} w \cdot \sigma'_t$  in backprop through time can easily vanish (or explode)  $\Rightarrow$  hard to learn for long-term memory effects
- ▶ Idea: Let's build an explicit longer-term memory into the model
- ▶ Here: vector graph for the gated recurrent unit (Cho et al. 2014, GRU) architecture

## Long Short-Term Memory Networks (Hochreiter and Schmidhuber 1997)

- ▶ Motivation: The term  $\prod_{\tau=t}^{T-1} w \cdot \sigma'_t$  in backprop through time can easily vanish (or explode)  $\Rightarrow$  hard to learn for long-term memory effects
- ▶ Idea: Let's build an explicit longer-term memory into the model
- ▶ Here: vector graph for the gated recurrent unit (Cho et al. 2014, GRU) architecture

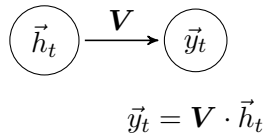
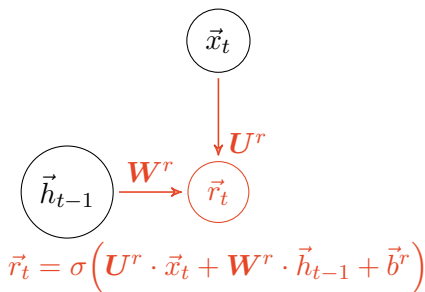


A diagram showing the transformation of the hidden state  $\vec{h}_t$  into the output  $\vec{y}_t$ . A circle containing  $\vec{h}_t$  has an arrow pointing to a circle containing  $\vec{y}_t$ , with the vector  $\mathbf{V}$  written above the arrow. Below the circles, the equation  $\vec{y}_t = \mathbf{V} \cdot \vec{h}_t$  is written.

$$\vec{y}_t = \mathbf{V} \cdot \vec{h}_t$$

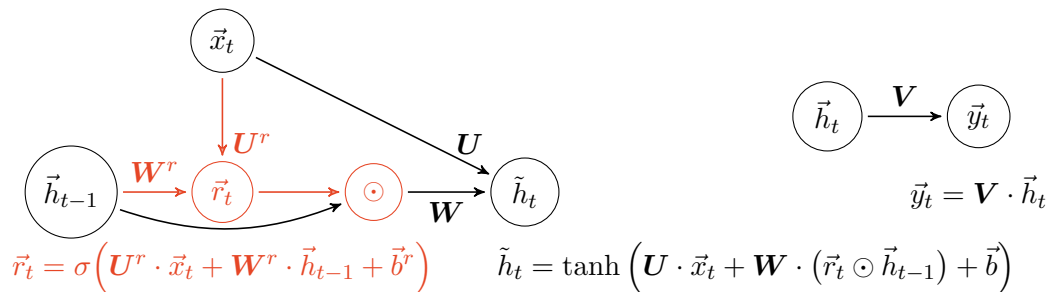
## Long Short-Term Memory Networks (Hochreiter and Schmidhuber 1997)

- ▶ Motivation: The term  $\prod_{\tau=t}^{T-1} w \cdot \sigma'_t$  in backprop through time can easily vanish (or explode)  $\Rightarrow$  hard to learn for long-term memory effects
- ▶ Idea: Let's build an explicit longer-term memory into the model
- ▶ Here: vector graph for the gated recurrent unit (Cho et al. 2014, GRU) architecture



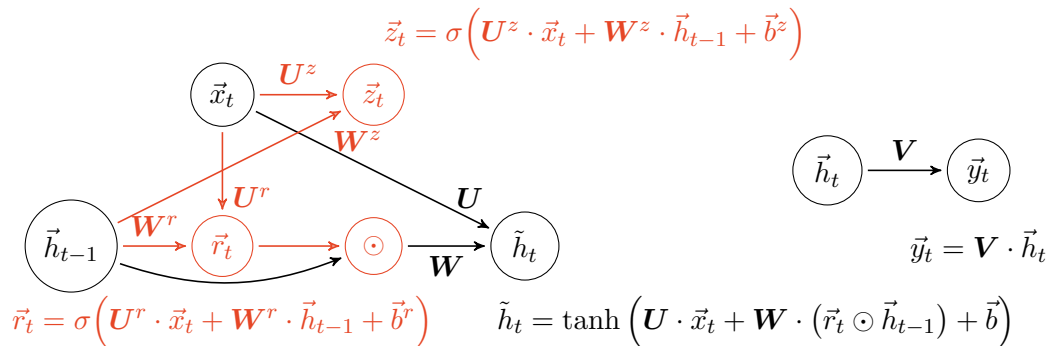
## Long Short-Term Memory Networks (Hochreiter and Schmidhuber 1997)

- ▶ Motivation: The term  $\prod_{\tau=t}^{T-1} w \cdot \sigma'_t$  in backprop through time can easily vanish (or explode)  $\Rightarrow$  hard to learn for long-term memory effects
- ▶ Idea: Let's build an explicit longer-term memory into the model
- ▶ Here: vector graph for the gated recurrent unit (Cho et al. 2014, GRU) architecture



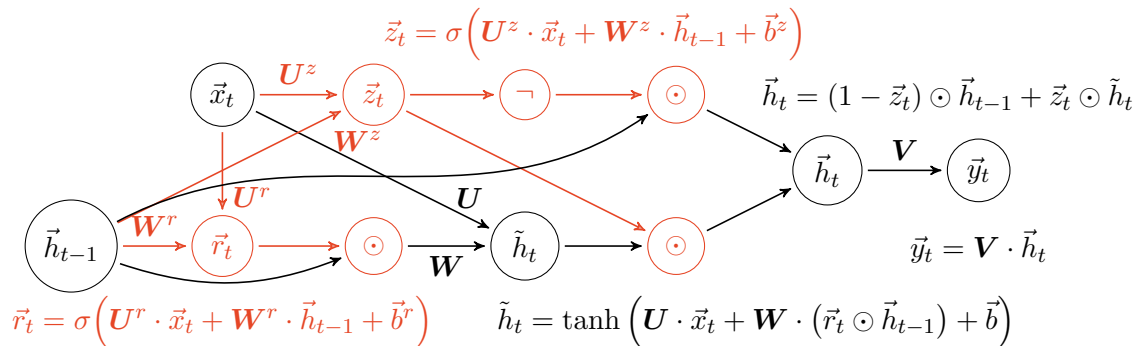
## Long Short-Term Memory Networks (Hochreiter and Schmidhuber 1997)

- ▶ Motivation: The term  $\prod_{\tau=t}^{T-1} w \cdot \sigma'_t$  in backprop through time can easily vanish (or explode)  $\Rightarrow$  hard to learn for long-term memory effects
- ▶ Idea: Let's build an explicit longer-term memory into the model
- ▶ Here: vector graph for the gated recurrent unit (Cho et al. 2014, GRU) architecture



## Long Short-Term Memory Networks (Hochreiter and Schmidhuber 1997)

- ▶ Motivation: The term  $\prod_{\tau=t}^{T-1} w \cdot \sigma'_t$  in backprop through time can easily vanish (or explode)  $\Rightarrow$  hard to learn for long-term memory effects
- ▶ Idea: Let's build an explicit longer-term memory into the model
- ▶ Here: vector graph for the gated recurrent unit (Cho et al. 2014, GRU) architecture





# Convolutions

(Fukushima, Miyake, and Ito 1983; LeCun and Bengio 1995)

- ▶ Idea: Handle variable-sized data via a **moving window**

# Convolutions

(Fukushima, Miyake, and Ito 1983; LeCun and Bengio 1995)

- ▶ Idea: Handle variable-sized data via a **moving window**

$x_1$

$x_2$

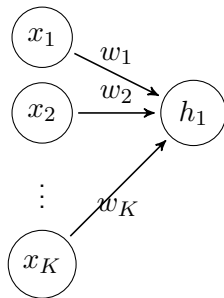
$\vdots$

$x_K$

# Convolutions

(Fukushima, Miyake, and Ito 1983; LeCun and Bengio 1995)

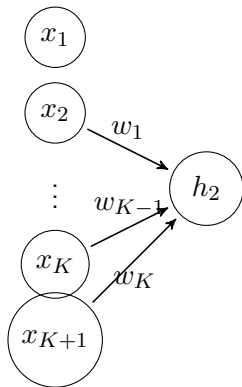
- Idea: Handle variable-sized data via a **moving window**



# Convolutions

(Fukushima, Miyake, and Ito 1983; LeCun and Bengio 1995)

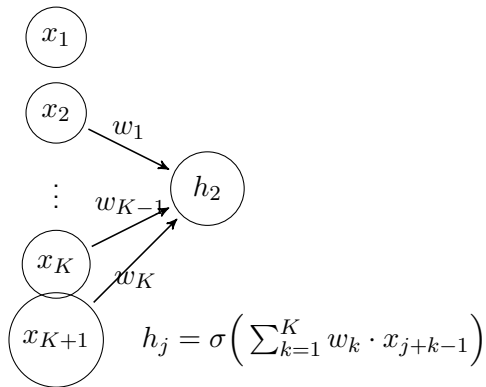
- Idea: Handle variable-sized data via a **moving window**



# Convolutions

(Fukushima, Miyake, and Ito 1983; LeCun and Bengio 1995)

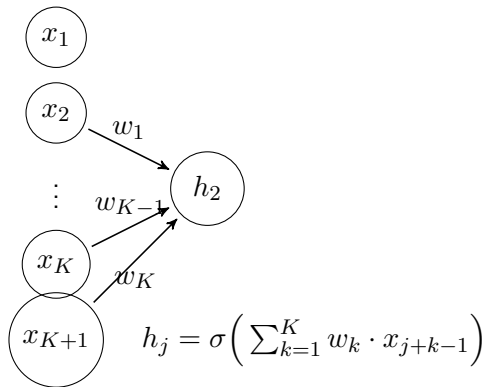
- Idea: Handle variable-sized data via a **moving window**



# Convolutions

(Fukushima, Miyake, and Ito 1983; LeCun and Bengio 1995)

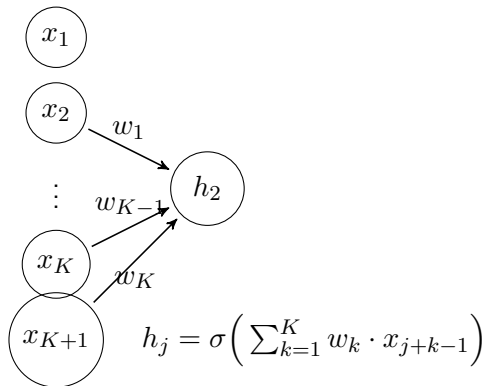
- ▶ Idea: Handle variable-sized data via a **moving window**
- ▶ Weights are **shared** across time steps



# Convolutions

(Fukushima, Miyake, and Ito 1983; LeCun and Bengio 1995)

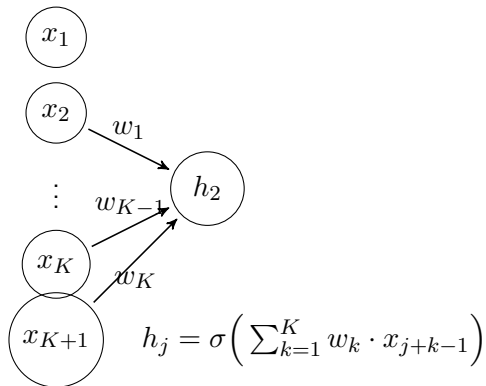
- ▶ Idea: Handle variable-sized data via a **moving window**
- ▶ Weights are **shared** across time steps
- ▶ Usually multiple convolutions in parallel and stacked



# Convolutions

(Fukushima, Miyake, and Ito 1983; LeCun and Bengio 1995)

- ▶ Idea: Handle variable-sized data via a **moving window**
- ▶ Weights are **shared** across time steps
- ▶ Usually multiple convolutions in parallel and stacked
- ▶ Especially useful for 2D (images) or 3D (video; voxels) data





## 2D Example

$X$

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

## 2D Example

$X$

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

$W$

-1	-1	-1
-1	+8	-1
-1	-1	-1

## 2D Example

$X$

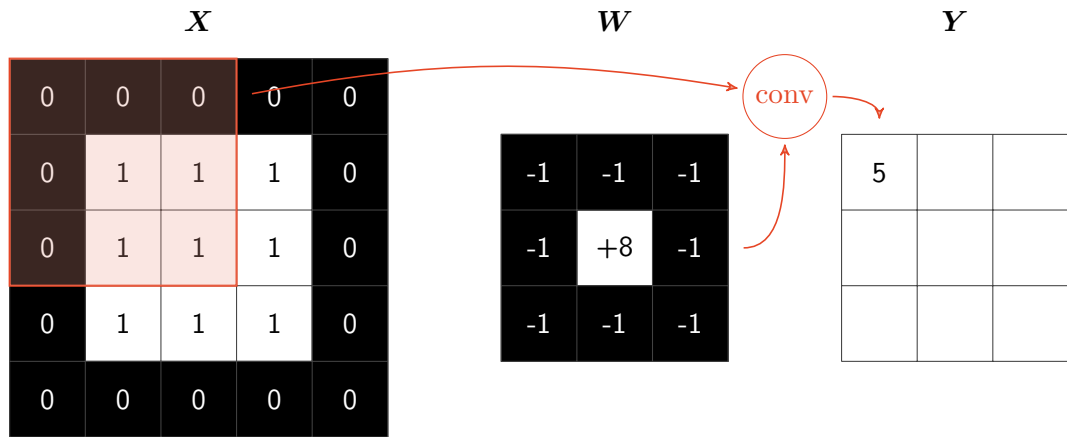
0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

$W$

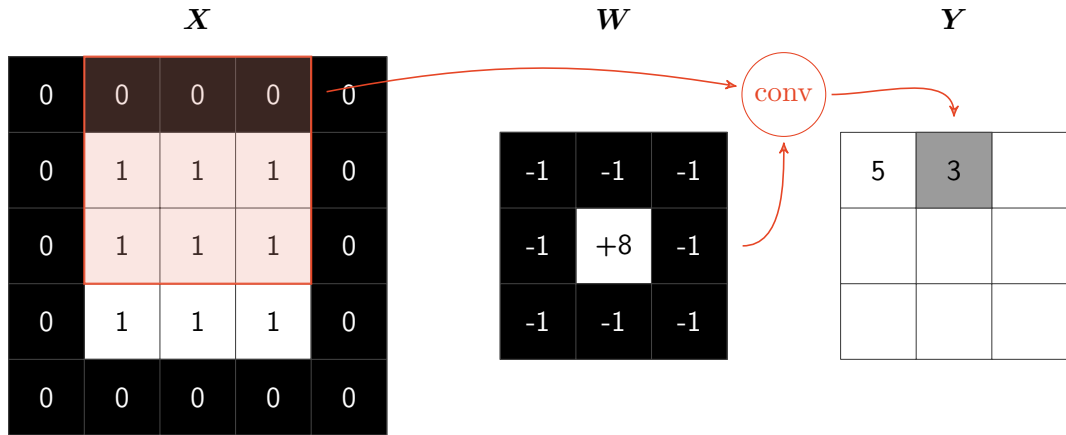
-1	-1	-1
-1	+8	-1
-1	-1	-1

$Y$

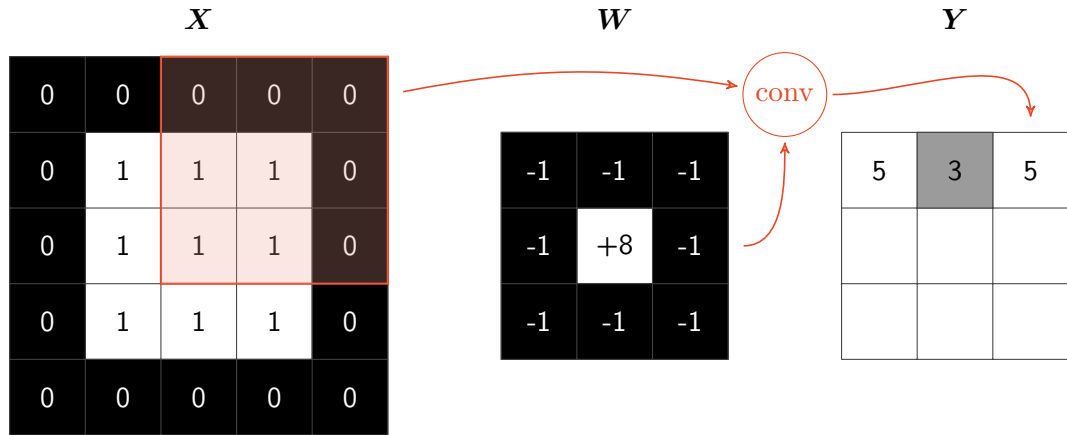

## 2D Example



## 2D Example



## 2D Example



## 2D Example

$X$

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

$W$

-1	-1	-1
-1	+8	-1
-1	-1	-1

conv

$Y$

5	3	5
3	0	3
5	3	5

## 2D Example

$X$

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

$W$

-1	-1	-1
-1	+8	-1
-1	-1	-1

conv

$Y$

5	3	5
3	0	3
5	3	5

- Strongly related to image filters (edge detection, sharpen, Gaussian blur, etc.)



## Softmax / Attention (Bridle 1990; Bahdanau, Cho, and Bengio 2014; Vaswani et al. 2017)

- Motivation: Assume an **unordered database**  $\{\vec{x}_1, \dots, \vec{x}_N\}$  of auxiliary vectors

## Softmax / Attention (Bridle 1990; Bahdanau, Cho, and Bengio 2014; Vaswani et al. 2017)

- Motivation: Assume an **unordered database**  $\{\vec{x}_1, \dots, \vec{x}_N\}$  of auxiliary vectors; what is the relevant information for vector  $\vec{q}$ ? (e.g.: machine translation)

## Softmax / Attention (Bridle 1990; Bahdanau, Cho, and Bengio 2014; Vaswani et al. 2017)

- ▶ Motivation: Assume an **unordered database**  $\{\vec{x}_1, \dots, \vec{x}_N\}$  of auxiliary vectors; what is the relevant information for vector  $\vec{q}$ ? (e.g.: machine translation)
- ▶ Idea: Compute **affinity**  $a(\vec{q}, \vec{x}_i)$  of  $\vec{q}$  to all  $\vec{x}_i$

## Softmax / Attention (Bridle 1990; Bahdanau, Cho, and Bengio 2014; Vaswani et al. 2014)

- ▶ Motivation: Assume an **unordered database**  $\{\vec{x}_1, \dots, \vec{x}_N\}$  of auxiliary vectors; what is the relevant information for vector  $\vec{q}$ ? (e.g.: machine translation)
- ▶ Idea: Compute **affinity**  $a(\vec{q}, \vec{x}_i)$  (e.g.  $\vec{q}^T \cdot \vec{x}_i$ ) of  $\vec{q}$  to all  $\vec{x}_i$

## Softmax / Attention (Bridle 1990; Bahdanau, Cho, and Bengio 2014; Vaswani et al. 2017)

- ▶ Motivation: Assume an **unordered database**  $\{\vec{x}_1, \dots, \vec{x}_N\}$  of auxiliary vectors; what is the relevant information for vector  $\vec{q}$ ? (e.g.: machine translation)
- ▶ Idea: Compute **affinity**  $a(\vec{q}, \vec{x}_i)$  (e.g.  $\vec{q}^T \cdot \vec{x}_i$ ) of  $\vec{q}$  to all  $\vec{x}_i$ , normalize to **attention scores**  $\alpha_i$

## Softmax / Attention (Bridle 1990; Bahdanau, Cho, and Bengio 2014; Vaswani et al. 2017)

- ▶ Motivation: Assume an **unordered database**  $\{\vec{x}_1, \dots, \vec{x}_N\}$  of auxiliary vectors; what is the relevant information for vector  $\vec{q}$ ? (e.g.: machine translation)
- ▶ Idea: Compute **affinity**  $a(\vec{q}, \vec{x}_i)$  (e.g.  $\vec{q}^T \cdot \vec{x}_i$ ) of  $\vec{q}$  to all  $\vec{x}_i$ , normalize to **attention scores**  $\alpha_i$  and then compute **weighted sum**  $\vec{y} = \sum_{i=1}^N \alpha_i \cdot \vec{x}_i$

## Softmax / Attention (Bridle 1990; Bahdanau, Cho, and Bengio 2014; Vaswani et al. 2017)

- ▶ Motivation: Assume an **unordered database**  $\{\vec{x}_1, \dots, \vec{x}_N\}$  of auxiliary vectors; what is the relevant information for vector  $\vec{q}$ ? (e.g.: machine translation)
- ▶ Idea: Compute **affinity**  $a(\vec{q}, \vec{x}_i)$  (e.g.  $\vec{q}^T \cdot \vec{x}_i$ ) of  $\vec{q}$  to all  $\vec{x}_i$ , normalize to **attention scores**  $\alpha_i$  and then compute **weighted sum**  $\vec{y} = \sum_{i=1}^N \alpha_i \cdot \vec{x}_i$

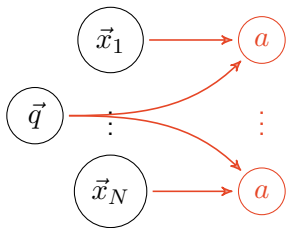
$$\vec{x}_1$$

$\vdots$

$$\vec{x}_N$$

## Softmax / Attention (Bridle 1990; Bahdanau, Cho, and Bengio 2014; Vaswani et al. 2017)

- Motivation: Assume an **unordered database**  $\{\vec{x}_1, \dots, \vec{x}_N\}$  of auxiliary vectors; what is the relevant information for vector  $\vec{q}$ ? (e.g.: machine translation)
- Idea: Compute **affinity**  $a(\vec{q}, \vec{x}_i)$  (e.g.  $\vec{q}^T \cdot \vec{x}_i$ ) of  $\vec{q}$  to all  $\vec{x}_i$ , normalize to **attention scores**  $\alpha_i$  and then compute **weighted sum**  $\vec{y} = \sum_{i=1}^N \alpha_i \cdot \vec{x}_i$

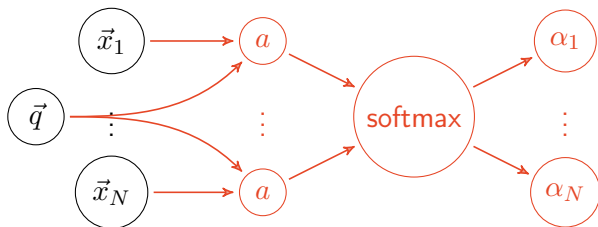




## Softmax / Attention (Bridle 1990; Bahdanau, Cho, and Bengio 2014; Vaswani et al. 2017)

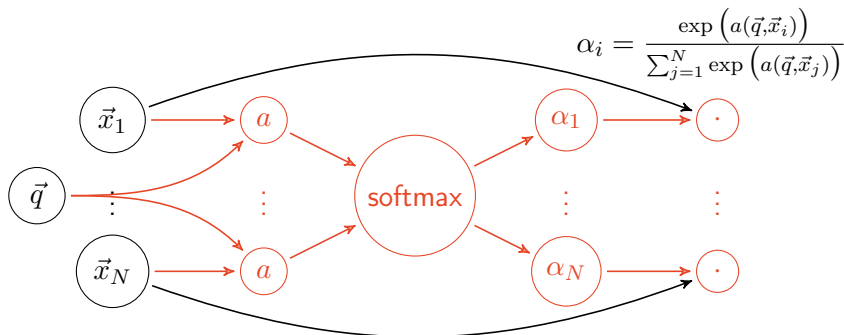
- Motivation: Assume an **unordered database**  $\{\vec{x}_1, \dots, \vec{x}_N\}$  of auxiliary vectors; what is the relevant information for vector  $\vec{q}$ ? (e.g.: machine translation)
- Idea: Compute **affinity**  $a(\vec{q}, \vec{x}_i)$  (e.g.  $\vec{q}^T \cdot \vec{x}_i$ ) of  $\vec{q}$  to all  $\vec{x}_i$ , normalize to **attention scores**  $\alpha_i$  and then compute **weighted sum**  $\vec{y} = \sum_{i=1}^N \alpha_i \cdot \vec{x}_i$

$$\alpha_i = \frac{\exp(a(\vec{q}, \vec{x}_i))}{\sum_{j=1}^N \exp(a(\vec{q}, \vec{x}_j))}$$



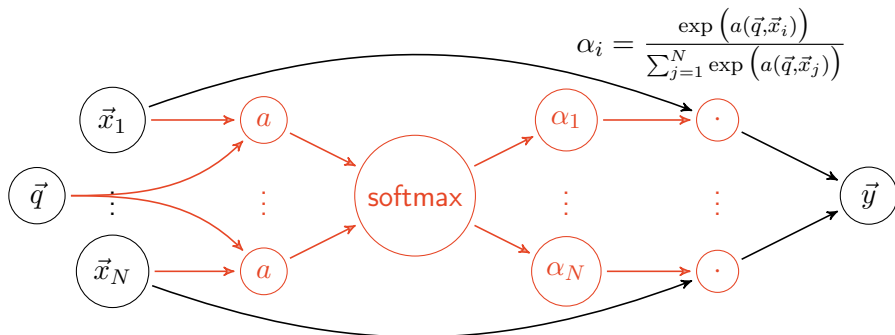
## Softmax / Attention (Bridle 1990; Bahdanau, Cho, and Bengio 2014; Vaswani et al. 2017)

- Motivation: Assume an **unordered database**  $\{\vec{x}_1, \dots, \vec{x}_N\}$  of auxiliary vectors; what is the relevant information for vector  $\vec{q}$ ? (e.g.: machine translation)
- Idea: Compute **affinity**  $a(\vec{q}, \vec{x}_i)$  (e.g.  $\vec{q}^T \cdot \vec{x}_i$ ) of  $\vec{q}$  to all  $\vec{x}_i$ , normalize to **attention scores**  $\alpha_i$  and then compute **weighted sum**  $\vec{y} = \sum_{i=1}^N \alpha_i \cdot \vec{x}_i$



## Softmax / Attention (Bridle 1990; Bahdanau, Cho, and Bengio 2014; Vaswani et al. 2017)

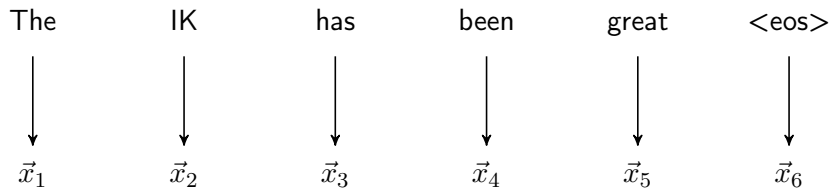
- Motivation: Assume an **unordered database**  $\{\vec{x}_1, \dots, \vec{x}_N\}$  of auxiliary vectors; what is the relevant information for vector  $\vec{q}$ ? (e.g.: machine translation)
- Idea: Compute **affinity**  $a(\vec{q}, \vec{x}_i)$  (e.g.  $\vec{q}^T \cdot \vec{x}_i$ ) of  $\vec{q}$  to all  $\vec{x}_i$ , normalize to **attention scores**  $\alpha_i$  and then compute **weighted sum**  $\vec{y} = \sum_{i=1}^N \alpha_i \cdot \vec{x}_i$



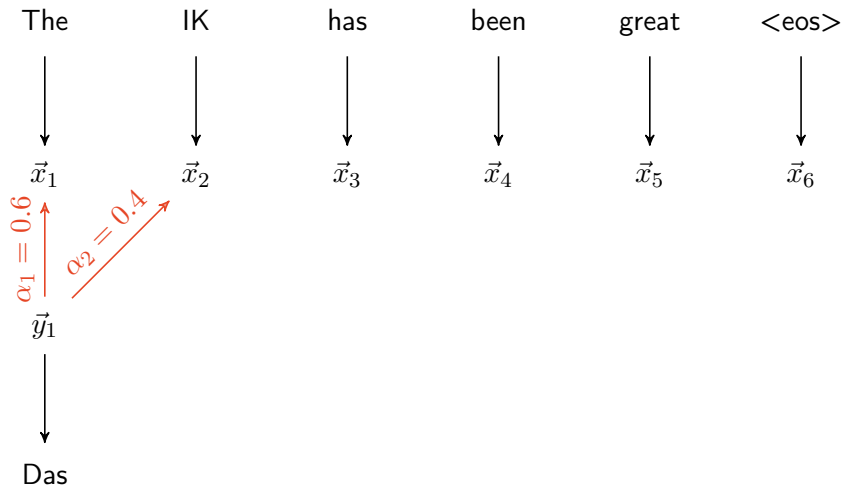
## Attention illustration

The            IK            has            been            great            <eos>

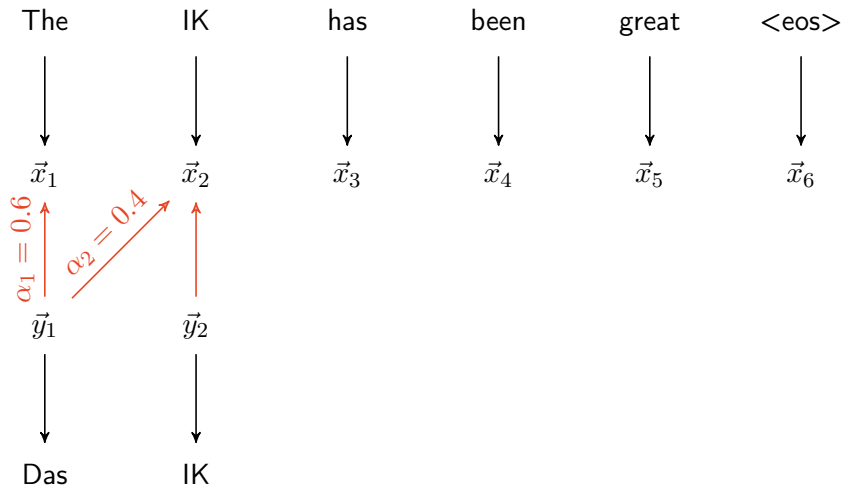
## Attention illustration



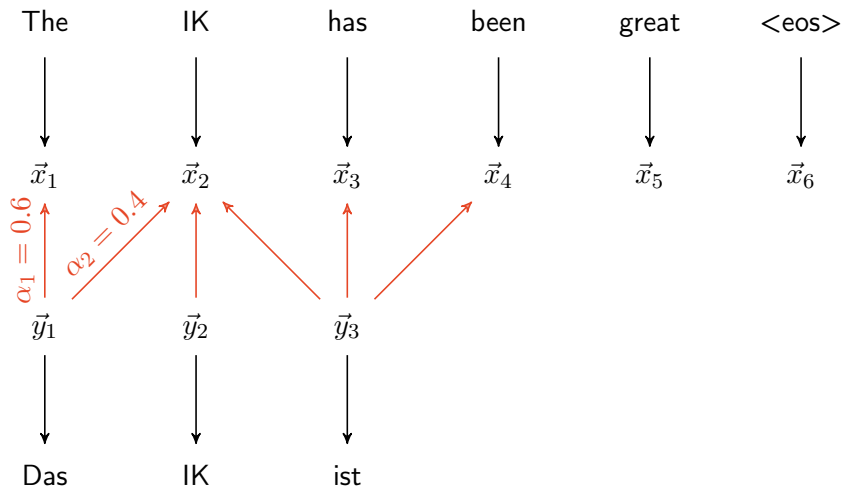
## Attention illustration



## Attention illustration

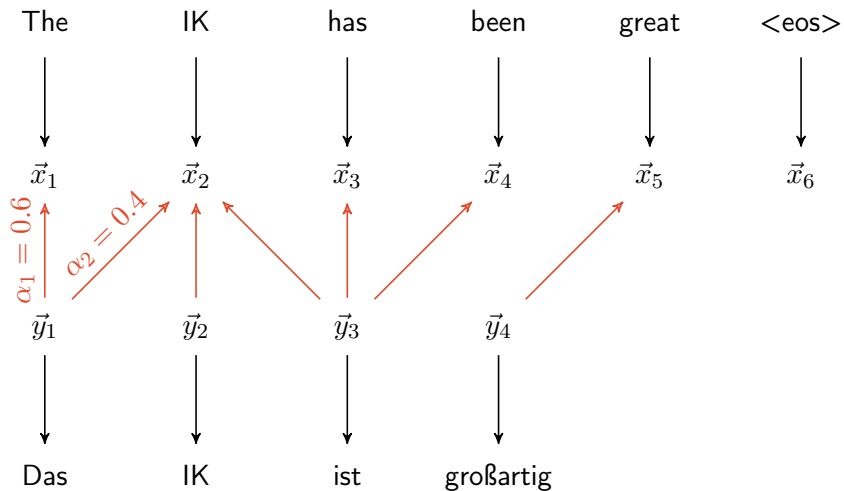


## Attention illustration

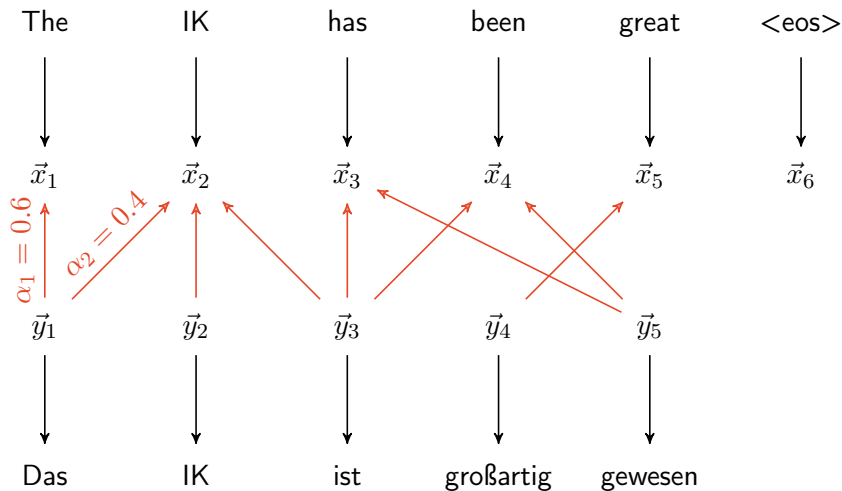




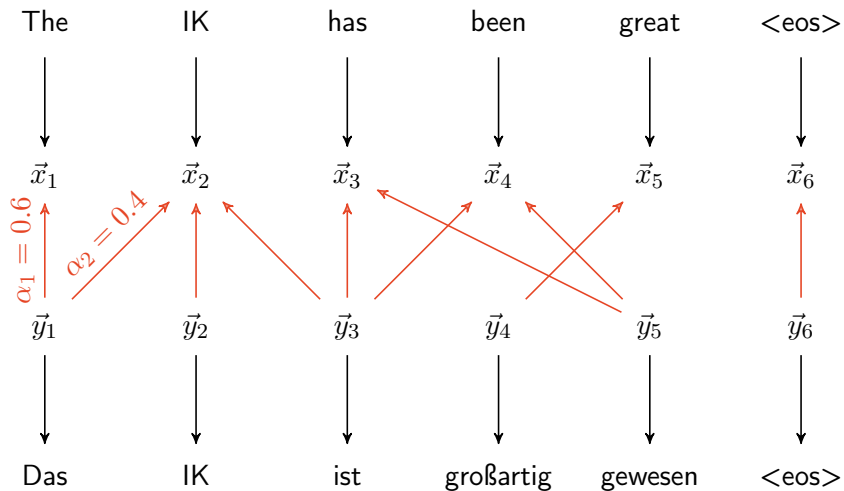
## Attention illustration



## Attention illustration



## Attention illustration



# Overview

perceptron/feedforward

$$\vec{x} \in \mathbb{R}^n$$

$$\vec{y} \in \mathbb{R}^L$$

# Overview

perceptron/feedforward

$$\vec{x} \in \mathbb{R}^n \xrightarrow{\mathbf{W} \in \mathbb{R}^{L \times n}} \vec{y} \in \mathbb{R}^L$$

# Overview

perceptron/feedforward

$$\vec{x} \in \mathbb{R}^n \xrightarrow{\mathbf{W} \in \mathbb{R}^{L \times n}} \vec{y} \in \mathbb{R}^L$$

convolution

$$\vec{x} \in \mathbb{R}^n$$

$$\vec{y} \in \mathbb{R}^{n-K}$$

# Overview

perceptron/feedforward

$$\vec{x} \in \mathbb{R}^n \xrightarrow{\mathbf{W} \in \mathbb{R}^{L \times n}} \vec{y} \in \mathbb{R}^L$$

convolution

$$\vec{x} \in \mathbb{R}^n \xrightarrow{\vec{w} \in \mathbb{R}^K} \vec{y} \in \mathbb{R}^{n-K}$$

## Overview

perceptron/feedforward

$$\vec{x} \in \mathbb{R}^n \xrightarrow{\mathbf{W} \in \mathbb{R}^{L \times n}} \vec{y} \in \mathbb{R}^L$$

convolution

$$\vec{x} \in \mathbb{R}^n \xrightarrow{\vec{w} \in \mathbb{R}^K} \vec{y} \in \mathbb{R}^{n-K}$$

$$\mathbf{X} \in \mathbb{R}^{n \times n} \xrightarrow{\mathbf{W} \in \mathbb{R}^{K \times K}} \mathbf{Y} \in \mathbb{R}^{n-K \times n-K}$$



## Overview

perceptron/feedforward

$$\vec{x} \in \mathbb{R}^n \xrightarrow{\mathbf{W} \in \mathbb{R}^{L \times n}} \vec{y} \in \mathbb{R}^L$$

convolution

$$\vec{x} \in \mathbb{R}^n \xrightarrow{\vec{w} \in \mathbb{R}^K} \vec{y} \in \mathbb{R}^{n-K}$$

$$\mathbf{X} \in \mathbb{R}^{n \times n} \xrightarrow{\mathbf{W} \in \mathbb{R}^{K \times K}} \mathbf{Y} \in \mathbb{R}^{n-K \times n-K}$$
$$\vdots$$

# Overview

perceptron/feedforward

$$\vec{x} \in \mathbb{R}^n \xrightarrow{\mathbf{W} \in \mathbb{R}^{L \times n}} \vec{y} \in \mathbb{R}^L$$

convolution

$$\vec{x} \in \mathbb{R}^n \xrightarrow{\vec{w} \in \mathbb{R}^K} \vec{y} \in \mathbb{R}^{n-K}$$

$$\mathbf{X} \in \mathbb{R}^{n \times n} \xrightarrow{\mathbf{W} \in \mathbb{R}^{K \times K}} \mathbf{Y} \in \mathbb{R}^{n-K \times n-K}$$
$$\vdots$$

recurrent

$$\vec{x}_1 \quad \cdots \quad \vec{x}_T \in \mathbb{R}^n$$

$$\vec{y}_1 \quad \cdots \quad \vec{y}_T \in \mathbb{R}^L$$

## Overview

perceptron/feedforward

$$\vec{x} \in \mathbb{R}^n \xrightarrow{\mathbf{W} \in \mathbb{R}^{L \times n}} \vec{y} \in \mathbb{R}^L$$

convolution

$$\vec{x} \in \mathbb{R}^n \xrightarrow{\vec{w} \in \mathbb{R}^K} \vec{y} \in \mathbb{R}^{n-K}$$

$$\mathbf{X} \in \mathbb{R}^{n \times n} \xrightarrow{\mathbf{W} \in \mathbb{R}^{K \times K}} \mathbf{Y} \in \mathbb{R}^{n-K \times n-K}$$
$$\vdots$$

recurrent

$$\vec{x}_1 \quad \cdots \quad \vec{x}_T \in \mathbb{R}^n$$

$$\vec{h}_1 \quad \cdots \quad \vec{h}_T \in \mathbb{R}^K$$

$$\vec{y}_1 \quad \cdots \quad \vec{y}_T \in \mathbb{R}^L$$

## Overview

perceptron/feedforward

$$\vec{x} \in \mathbb{R}^n \xrightarrow{\mathbf{W} \in \mathbb{R}^{L \times n}} \vec{y} \in \mathbb{R}^L$$

convolution

$$\vec{x} \in \mathbb{R}^n \xrightarrow{\vec{w} \in \mathbb{R}^K} \vec{y} \in \mathbb{R}^{n-K}$$

$$\mathbf{X} \in \mathbb{R}^{n \times n} \xrightarrow{\mathbf{W} \in \mathbb{R}^{K \times K}} \mathbf{Y} \in \mathbb{R}^{n-K \times n-K}$$
$$\vdots$$

recurrent

$$\begin{array}{ccccc} \vec{x}_1 & & \cdots & & \vec{x}_T \in \mathbb{R}^n \\ U \downarrow & & & & U \downarrow \\ \vec{h}_1 & \xrightarrow{\mathbf{W}} & \cdots & \xrightarrow{\mathbf{W}} & \vec{h}_T \in \mathbb{R}^K \\ V \downarrow & & & & V \downarrow \\ \vec{y}_1 & & \cdots & & \vec{y}_T \in \mathbb{R}^L \end{array}$$

# Overview

perceptron/feedforward

$$\vec{x} \in \mathbb{R}^n \xrightarrow{\mathbf{W} \in \mathbb{R}^{L \times n}} \vec{y} \in \mathbb{R}^L$$

recurrent

$$\begin{array}{ccccc} \vec{x}_1 & & \cdots & & \vec{x}_T \in \mathbb{R}^n \\ U \downarrow & & & & U \downarrow \\ \vec{h}_1 & \xrightarrow{\mathbf{W}} & \cdots & \xrightarrow{\mathbf{W}} & \vec{h}_T \in \mathbb{R}^K \\ V \downarrow & & & & V \downarrow \\ \vec{y}_1 & & \cdots & & \vec{y}_T \in \mathbb{R}^L \end{array}$$

convolution

$$\vec{x} \in \mathbb{R}^n \xrightarrow{\vec{w} \in \mathbb{R}^K} \vec{y} \in \mathbb{R}^{n-K}$$

$$\mathbf{X} \in \mathbb{R}^{n \times n} \xrightarrow{\mathbf{W} \in \mathbb{R}^{K \times K}} \mathbf{Y} \in \mathbb{R}^{n-K \times n-K}$$

$\vdots$

attention

$$\begin{array}{ccc} & \vec{q} & \\ \vec{x}_1 & \cdots & \vec{x}_N \in \mathbb{R}^n \end{array}$$

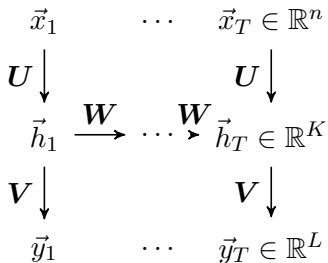
$$\vec{y} \in \mathbb{R}^n$$

# Overview

perceptron/feedforward

$$\vec{x} \in \mathbb{R}^n \xrightarrow{\mathbf{W} \in \mathbb{R}^{L \times n}} \vec{y} \in \mathbb{R}^L$$

recurrent



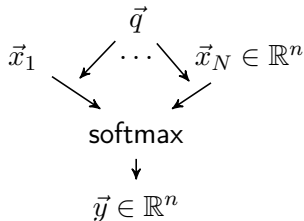
convolution

$$\vec{x} \in \mathbb{R}^n \xrightarrow{\vec{w} \in \mathbb{R}^K} \vec{y} \in \mathbb{R}^{n-K}$$

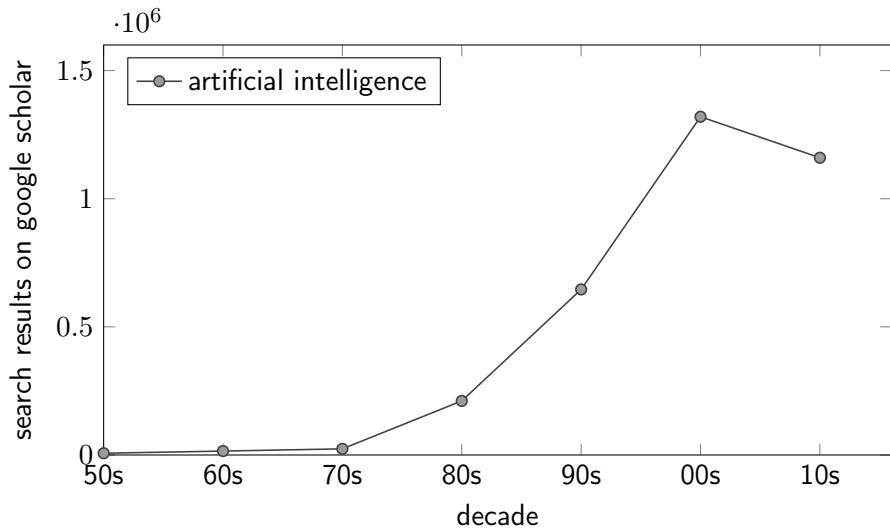
$$\mathbf{X} \in \mathbb{R}^{n \times n} \xrightarrow{\mathbf{W} \in \mathbb{R}^{K \times K}} \mathbf{Y} \in \mathbb{R}^{n-K \times n-K}$$

$\vdots$

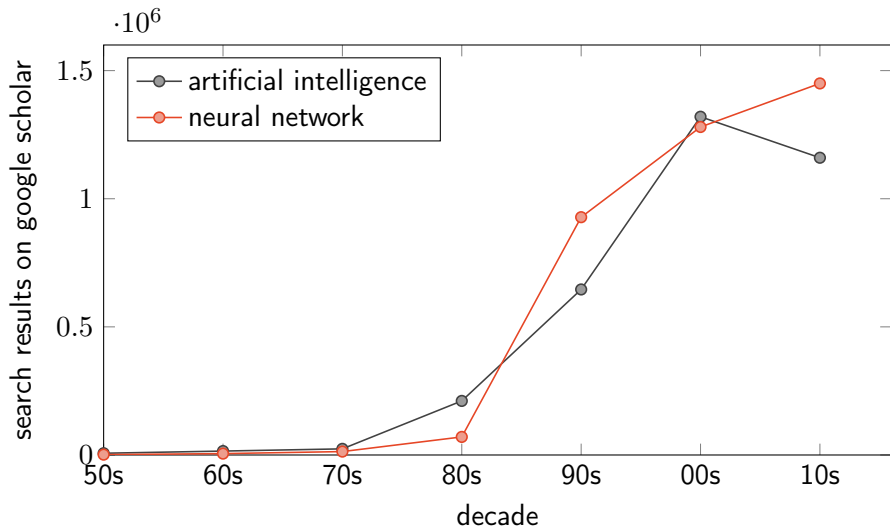
attention



## A short note on the history of neural nets

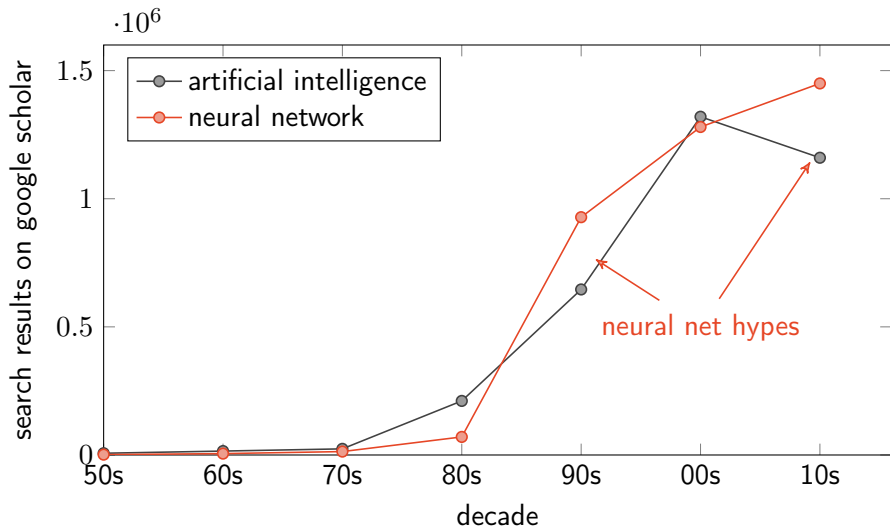


## A short note on the history of neural nets





## A short note on the history of neural nets



## Recent innovations and tricks

- ▶ **GPUs:** Run your neural net on graphics cards (Chellapilla, Puri, and Simard 2006)

## Recent innovations and tricks

- ▶ **GPUs:** Run your neural net on graphics cards (Chellapilla, Puri, and Simard 2006)
- ▶ **Max Pooling:** Take the maximum value over region after convolution layer for more **shift-invariance** and rapid size reduction (Jarrett et al. 2009)

## Recent innovations and tricks

- ▶ **GPUs:** Run your neural net on graphics cards (Chellapilla, Puri, and Simard 2006)
- ▶ **Max Pooling:** Take the maximum value over region after convolution layer for more **shift-invariance** and rapid size reduction (Jarrett et al. 2009)
- ▶ Use really **big datasets** (Deng et al. 2009; Krizhevsky, Sutskever, and Hinton 2012)

## Recent innovations and tricks

- ▶ **GPUs:** Run your neural net on graphics cards (Chellapilla, Puri, and Simard 2006)
- ▶ **Max Pooling:** Take the maximum value over region after convolution layer for more **shift-invariance** and rapid size reduction (Jarrett et al. 2009)
- ▶ Use really **big datasets** (Deng et al. 2009; Krizhevsky, Sutskever, and Hinton 2012)
- ▶ **Dropout:** During training, randomly disable neurons to force feature independence (Srivastava et al. 2014)

## Recent innovations and tricks

- ▶ **GPUs:** Run your neural net on graphics cards (Chellapilla, Puri, and Simard 2006)
- ▶ **Max Pooling:** Take the maximum value over region after convolution layer for more **shift-invariance** and rapid size reduction (Jarrett et al. 2009)
- ▶ Use really **big datasets** (Deng et al. 2009; Krizhevsky, Sutskever, and Hinton 2012)
- ▶ **Dropout:** During training, randomly disable neurons to force feature independence (Srivastava et al. 2014)
- ▶ **Batch normalization:** Ensure similar range for features (Ioffe and Szegedy 2015)

## Recent innovations and tricks

- ▶ **GPUs:** Run your neural net on graphics cards (Chellapilla, Puri, and Simard 2006)
- ▶ **Max Pooling:** Take the maximum value over region after convolution layer for more **shift-invariance** and rapid size reduction (Jarrett et al. 2009)
- ▶ Use really **big datasets** (Deng et al. 2009; Krizhevsky, Sutskever, and Hinton 2012)
- ▶ **Dropout:** During training, randomly disable neurons to force feature independence (Srivastava et al. 2014)
- ▶ **Batch normalization:** Ensure similar range for features (Ioffe and Szegedy 2015)
- ▶ **Residual Nets:** Add layer input to layer output for 'shortcuts' in the gradient and separation of concerns (He et al. 2016)

# Recipes for neural network construction

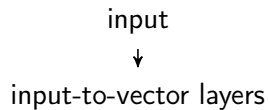


THE UNIVERSITY OF  
SYDNEY



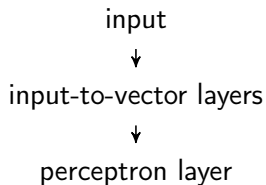
# Regressor

1. Transform input data into vector form



# Regressor

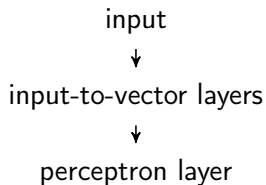
1. Transform input data into vector form
2. Push vectors through several perceptron layers, where last layer has  $L$  outputs



perceptron layer

# Regressor

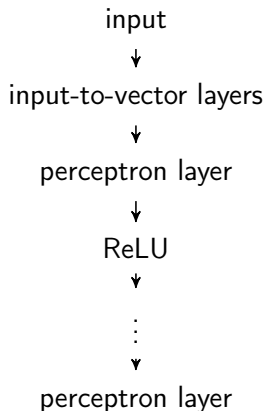
1. Transform input data into vector form
2. Push vectors through several perceptron layers, where last layer has  $L$  outputs
3. use residual connections and dropout or batch normalization if needed



perceptron layer

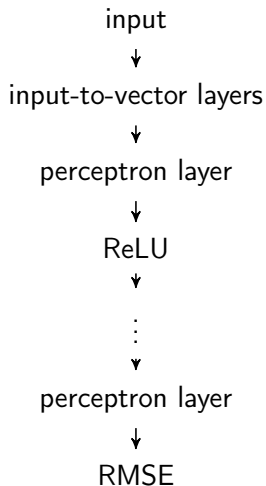
# Regressor

1. Transform input data into vector form
2. Push vectors through several perceptron layers, where last layer has  $L$  outputs
3. use residual connections and dropout or batch normalization if needed
4. Between each layer, apply nonlinearity, e.g. ReLU



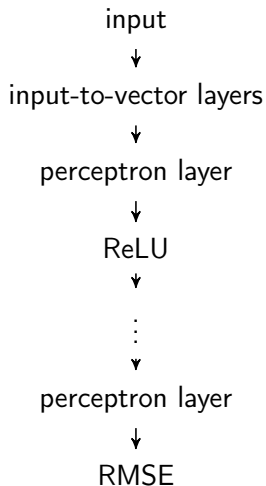
# Regressor

1. Transform input data into vector form
2. Push vectors through several perceptron layers, where last layer has  $L$  outputs
3. use residual connections and dropout or batch normalization if needed
4. Between each layer, apply nonlinearity, e.g. ReLU
5. Apply regression loss, e.g. RMSE



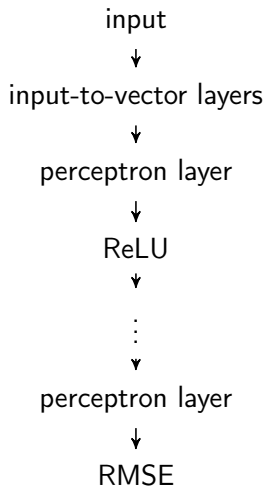
## Regressor

1. Transform input data into vector form
2. Push vectors through several perceptron layers, where last layer has  $L$  outputs
3. use residual connections and dropout or batch normalization if needed
4. Between each layer, apply nonlinearity, e.g. ReLU
5. Apply regression loss, e.g. RMSE
6. Apply an optimizer, e.g. ADAM, until loss is very low



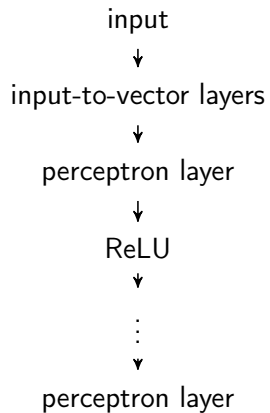
# Regressor

1. Transform input data into vector form
2. Push vectors through several perceptron layers, where last layer has  $L$  outputs
3. use residual connections and dropout or batch normalization if needed
4. Between each layer, apply nonlinearity, e.g. ReLU
5. Apply regression loss, e.g. RMSE
6. Apply an optimizer, e.g. ADAM, until loss is very low
7. Use weight decay if needed



# Classifier

- ▶ Same recipe as before, but:



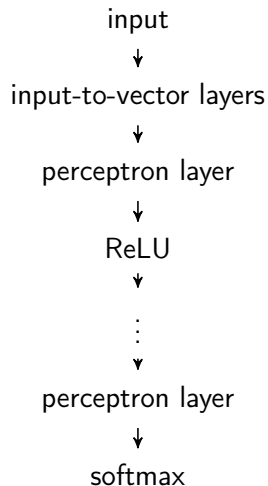


# Classifier

► Same recipe as before, but:

## 5.1. Apply softmax layer

$$\hat{y}_l = \frac{\exp(x_l)}{\sum_{l'=1}^L \exp(x_{l'})}$$



# Classifier

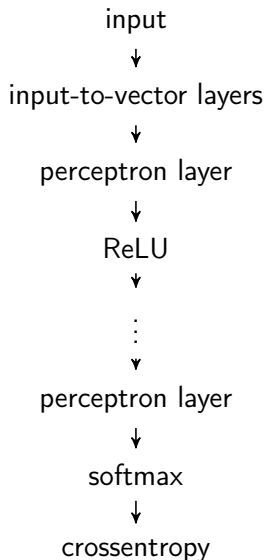
► Same recipe as before, but:

5.1. Apply softmax layer

$$\hat{y}_l = \frac{\exp(x_l)}{\sum_{l'=1}^L \exp(x_{l'})}$$

5.2. Apply classification loss, e.g. crossentropy

$$\ell = - \sum_{l=1}^L y_l \cdot \log [\hat{y}_l]$$



# Classifier

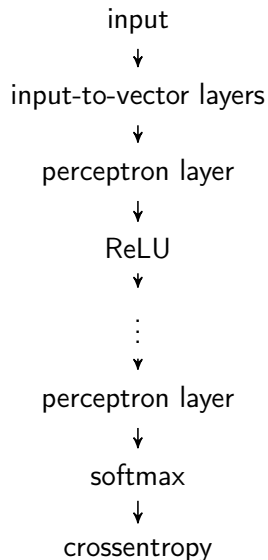
- ▶ Same recipe as before, but:

## 5.1. Apply softmax layer

$$\hat{y}_l = \frac{\exp(x_l)}{\sum_{l'=1}^L \exp(x_{l'})}$$

## 5.2. Apply classification loss, e.g. crossentropy

$$\ell = - \sum_{l=1}^L y_l \cdot \log [\hat{y}_l] = - \sum_{l=1}^L y_l \cdot (x_l - \log [\sum_{l'=1}^L \exp(x_{l'})])$$



# Variational Auto-Encoder

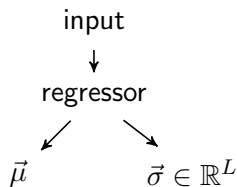
(Kingma and Welling 2013)

- ▶ Method for **dimensionality reduction**

# Variational Auto-Encoder

(Kingma and Welling 2013)

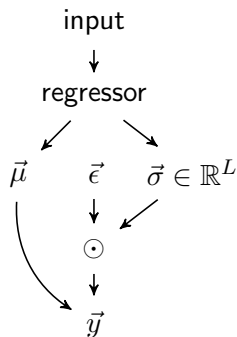
- ▶ Method for **dimensionality reduction**
- ▶ Same recipe as regression for first half, resulting in mean vector  $\vec{\mu}$  and standard deviation vector  $\vec{\sigma}$



# Variational Auto-Encoder

(Kingma and Welling 2013)

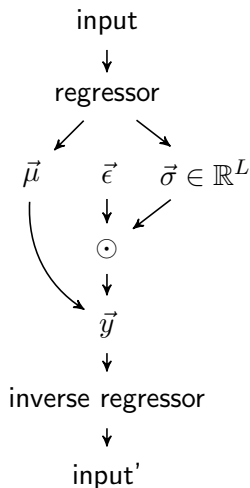
- ▶ Method for **dimensionality reduction**
- ▶ Same recipe as regression for first half, resulting in mean vector  $\vec{\mu}$  and standard deviation vector  $\vec{\sigma}$
- ▶ Generate low-dimensional representation as  $\vec{y} = \vec{\mu} + \vec{\epsilon} \odot \vec{\sigma}$  for random normal noise  $\vec{\epsilon}$



# Variational Auto-Encoder

(Kingma and Welling 2013)

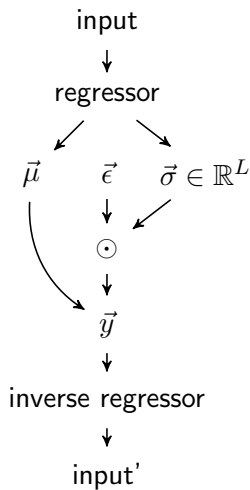
- ▶ Method for **dimensionality reduction**
- ▶ Same recipe as regression for first half, resulting in mean vector  $\vec{\mu}$  and standard deviation vector  $\vec{\sigma}$
- ▶ Generate low-dimensional representation as  $\vec{y} = \vec{\mu} + \vec{\epsilon} \odot \vec{\sigma}$  for random normal noise  $\vec{\epsilon}$
- ▶ Use an **inverse regressor** to decode back to input



# Variational Auto-Encoder

(Kingma and Welling 2013)

- ▶ Method for **dimensionality reduction**
- ▶ Same recipe as regression for first half, resulting in mean vector  $\vec{\mu}$  and standard deviation vector  $\vec{\sigma}$
- ▶ Generate low-dimensional representation as  $\vec{y} = \vec{\mu} + \vec{\epsilon} \odot \vec{\sigma}$  for random normal noise  $\vec{\epsilon}$
- ▶ Use an **inverse regressor** to decode back to input
- ▶ Loss:  
$$\ell_{\text{MSE}}(\text{input}, \text{input}') + \beta \cdot \left( \vec{\mu}^T \cdot \vec{\mu} + \sum_{l=1}^L \sigma_l^2 - \log[\sigma_l^2] \right)$$

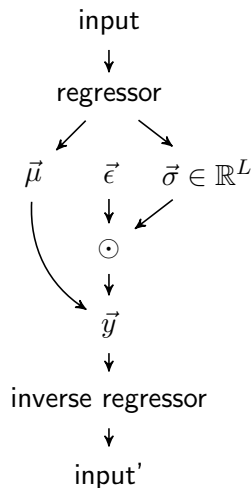




# Variational Auto-Encoder

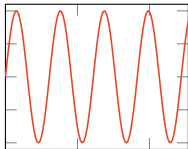
(Kingma and Welling 2013)

- ▶ Method for **dimensionality reduction**
- ▶ Same recipe as regression for first half, resulting in mean vector  $\vec{\mu}$  and standard deviation vector  $\vec{\sigma}$
- ▶ Generate low-dimensional representation as  $\vec{y} = \vec{\mu} + \vec{\epsilon} \odot \vec{\sigma}$  for random normal noise  $\vec{\epsilon}$
- ▶ Use an **inverse regressor** to decode back to input
- ▶ Loss:  
$$\ell_{\text{MSE}}(\text{input}, \text{input}') + \beta \cdot \left( \vec{\mu}^T \cdot \vec{\mu} + \sum_{l=1}^L \sigma_l^2 - \log[\sigma_l^2] \right)$$
- ▶ **Note:** Loss is very likely to fluctuate due to randomness

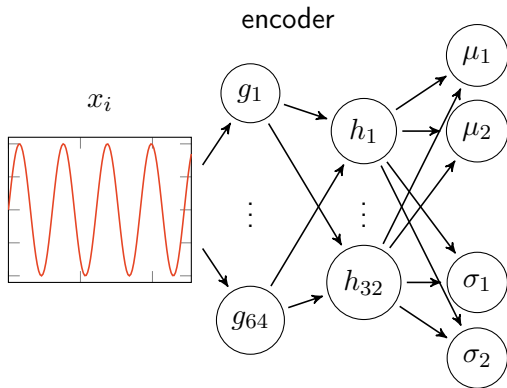


## Variational Auto-Encoder example

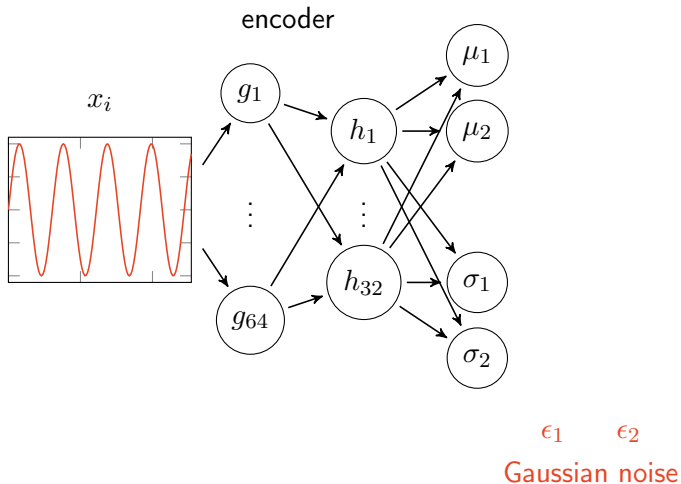
$x_i$



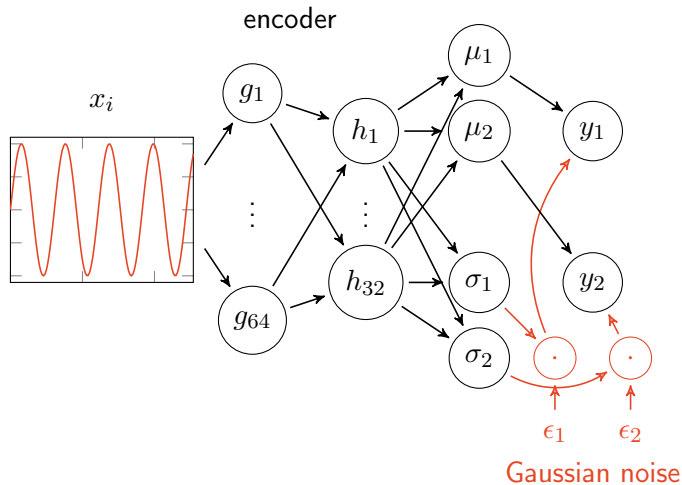
## Variational Auto-Encoder example



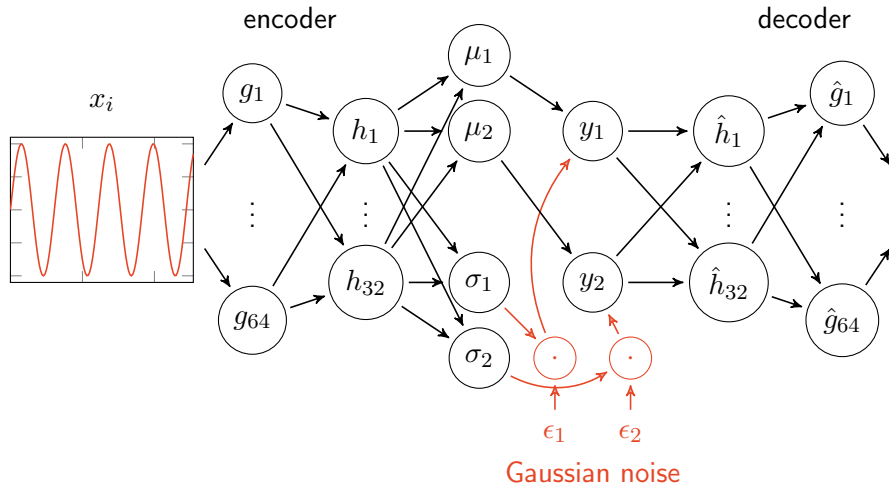
## Variational Auto-Encoder example



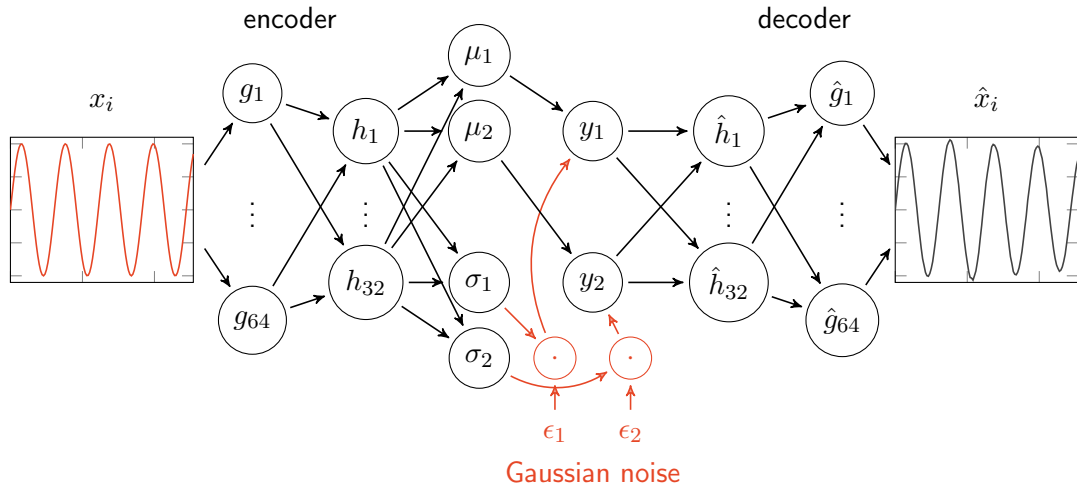
## Variational Auto-Encoder example



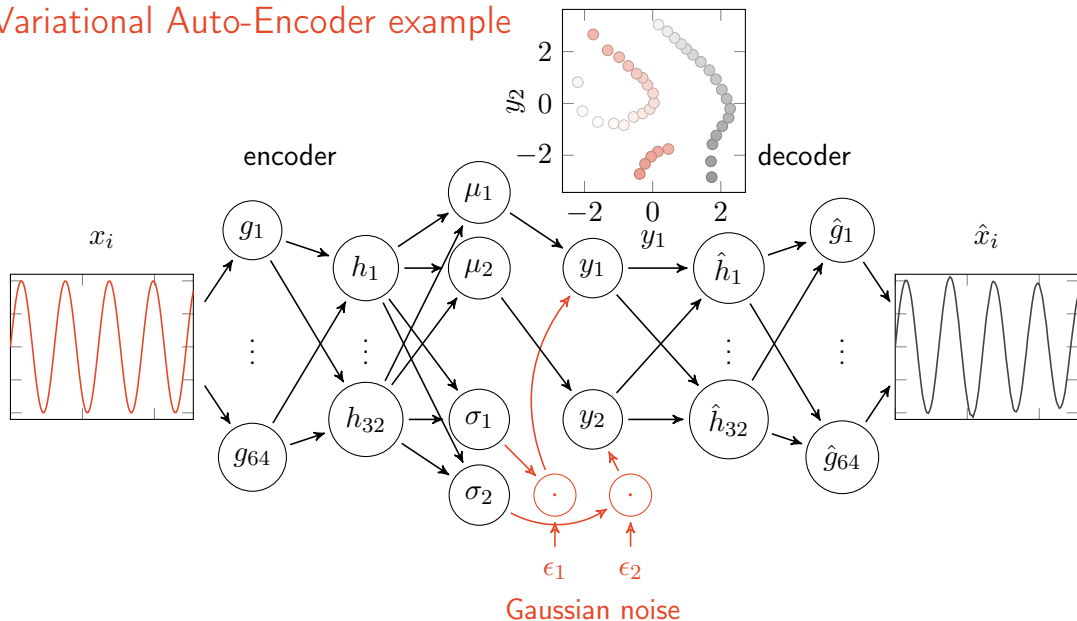
## Variational Auto-Encoder example



## Variational Auto-Encoder example

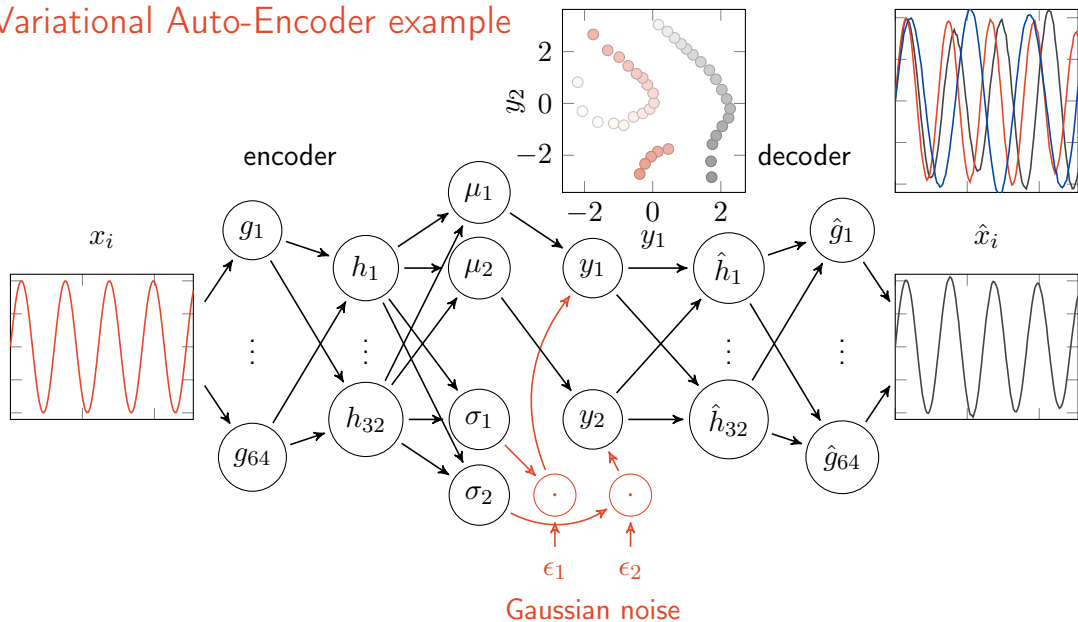


## Variational Auto-Encoder example





# Variational Auto-Encoder example



# Sequence-to-Sequence Learning

(Sutskever, Vinyals, and Le 2014)

- ▶ Motivation: How to map from a sequence to another sequence **of different length and order**? (e.g. machine translation)

# Sequence-to-Sequence Learning

(Sutskever, Vinyals, and Le 2014)

- ▶ Motivation: How to map from a sequence to another sequence **of different length and order**? (e.g. machine translation)
- ▶ Idea: RNN for encoding, auto-regressive RNN for decoding

# Sequence-to-Sequence Learning

(Sutskever, Vinyals, and Le 2014)

- ▶ Motivation: How to map from a sequence to another sequence **of different length and order?** (e.g. machine translation)
- ▶ Idea: RNN for encoding, auto-regressive RNN for decoding

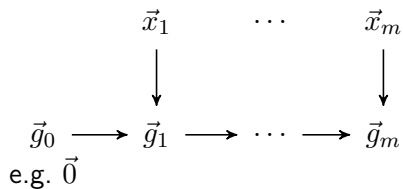
$\vec{x}_1 \quad \dots \quad \vec{x}_m$

$\vec{y}_1 \quad \dots \quad \vec{y}_n$

# Sequence-to-Sequence Learning

(Sutskever, Vinyals, and Le 2014)

- ▶ Motivation: How to map from a sequence to another sequence **of different length and order?** (e.g. machine translation)
- ▶ Idea: RNN for encoding, auto-regressive RNN for decoding



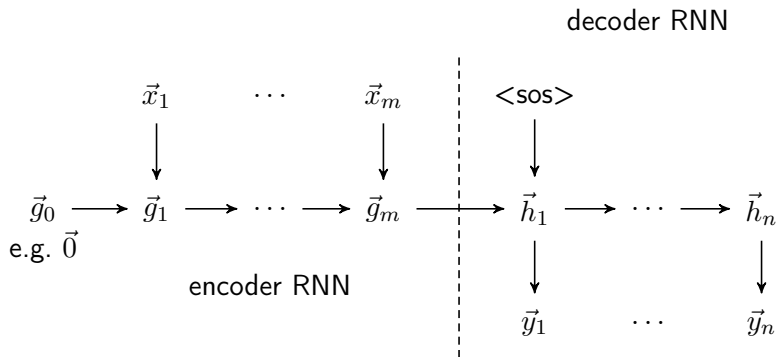
encoder RNN

$\vec{y}_1$        $\dots$        $\vec{y}_n$

# Sequence-to-Sequence Learning

(Sutskever, Vinyals, and Le 2014)

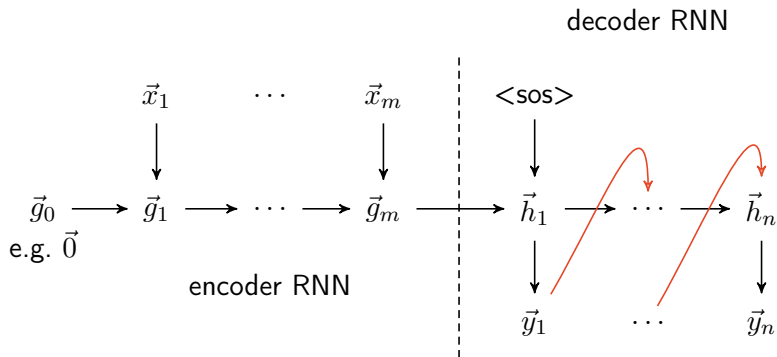
- Motivation: How to map from a sequence to another sequence **of different length and order?** (e.g. machine translation)
- Idea: RNN for encoding, auto-regressive RNN for decoding



# Sequence-to-Sequence Learning

(Sutskever, Vinyals, and Le 2014)

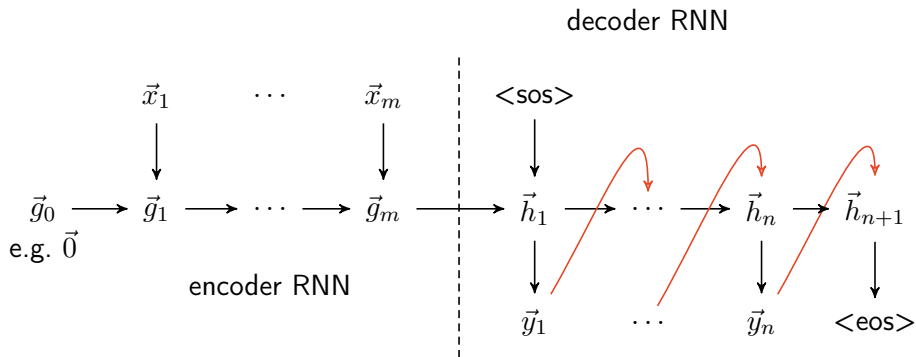
- ▶ Motivation: How to map from a sequence to another sequence **of different length and order?** (e.g. machine translation)
- ▶ Idea: RNN for encoding, auto-regressive RNN for decoding



# Sequence-to-Sequence Learning

(Sutskever, Vinyals, and Le 2014)

- ▶ Motivation: How to map from a sequence to another sequence **of different length and order?** (e.g. machine translation)
- ▶ Idea: RNN for encoding, auto-regressive RNN for decoding





## Seq2Seq example

A      B

A      B      A      B

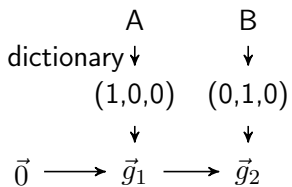
## Seq2Seq example

	A	B
dictionary ↓		↓
	(1,0,0)	(0,1,0)

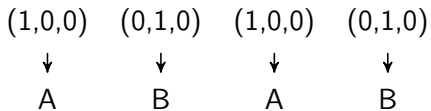
(1,0,0)	(0,1,0)	(1,0,0)	(0,1,0)
↓	↓	↓	↓
A	B	A	B

dictionary

## Seq2Seq example

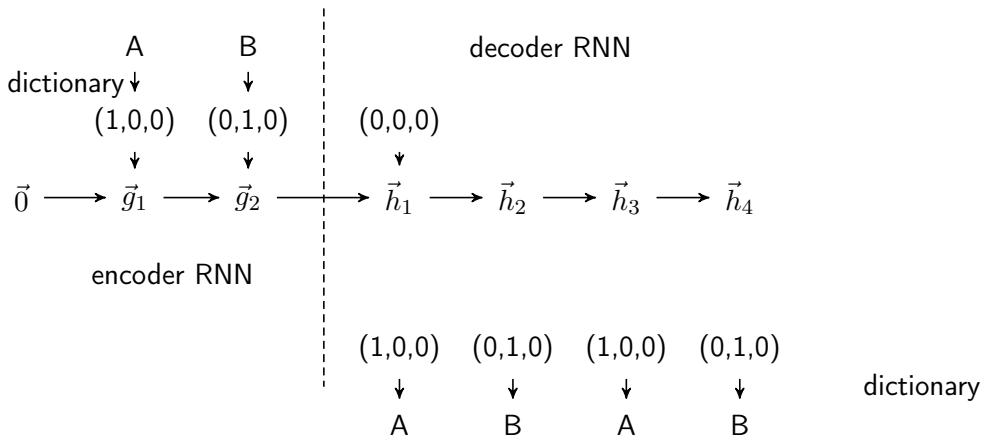


encoder RNN

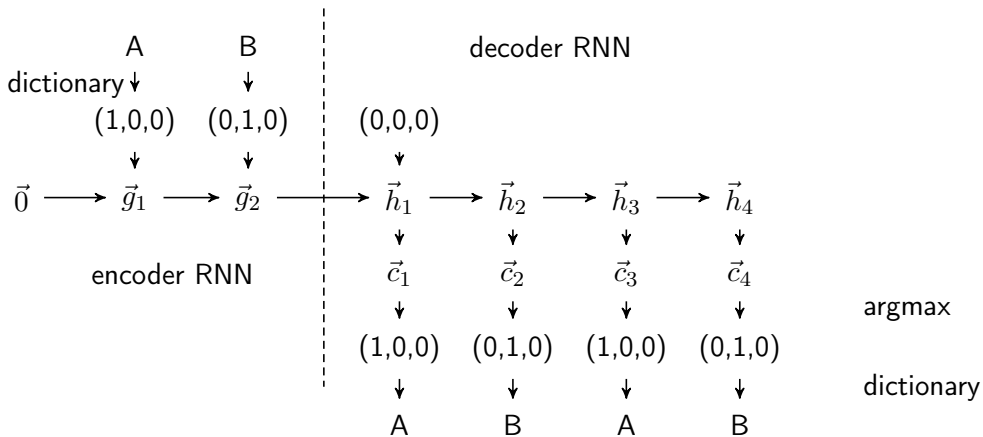


dictionary

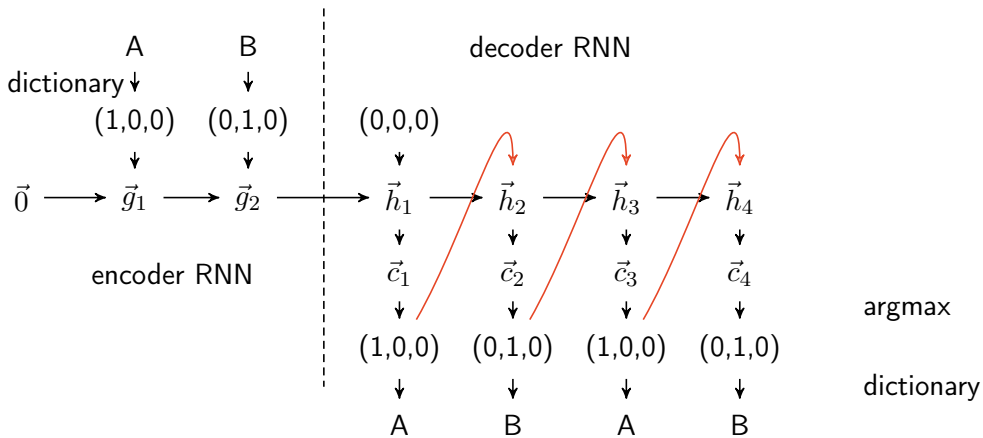
## Seq2Seq example



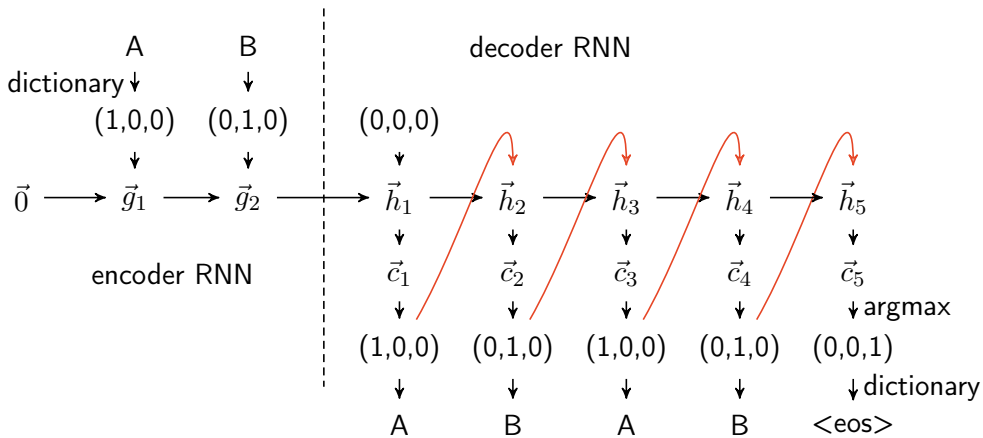
## Seq2Seq example



## Seq2Seq example



## Seq2Seq example



## Generative Adversarial Networks (Goodfellow, Pouget-Abadie, et al. 2014)

- ▶ Motivation: Generate realistic-looking 'fake' data, e.g. for data augmentation



## Generative Adversarial Networks (Goodfellow, Pouget-Abadie, et al. 2014)

- ▶ Motivation: Generate realistic-looking 'fake' data, e.g. for data augmentation
- ▶ Idea: Let a generator and a discriminator network compete

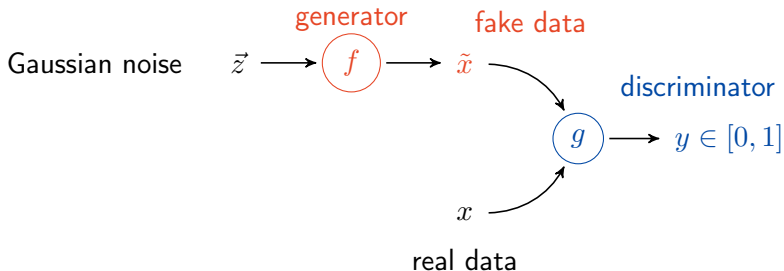
## Generative Adversarial Networks (Goodfellow, Pouget-Abadie, et al. 2014)

- ▶ Motivation: Generate realistic-looking 'fake' data, e.g. for data augmentation
- ▶ Idea: Let a generator and a discriminator network compete



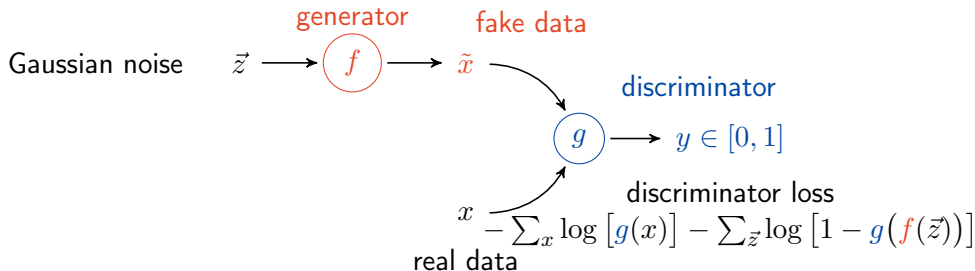
# Generative Adversarial Networks (Goodfellow, Pouget-Abadie, et al. 2014)

- ▶ Motivation: Generate realistic-looking 'fake' data, e.g. for data augmentation
- ▶ Idea: Let a generator and a discriminator network compete



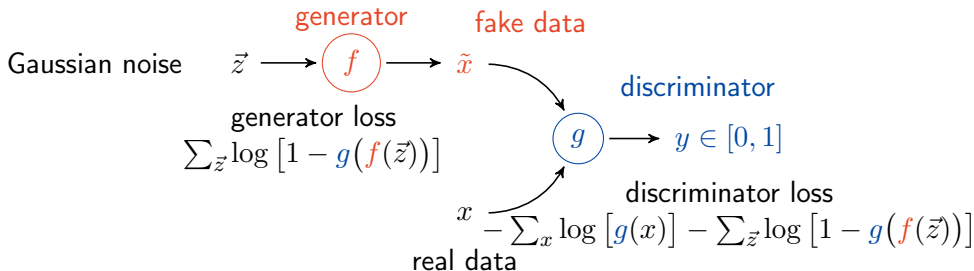
# Generative Adversarial Networks (Goodfellow, Pouget-Abadie, et al. 2014)

- ▶ Motivation: Generate realistic-looking 'fake' data, e.g. for data augmentation
- ▶ Idea: Let a generator and a discriminator network compete



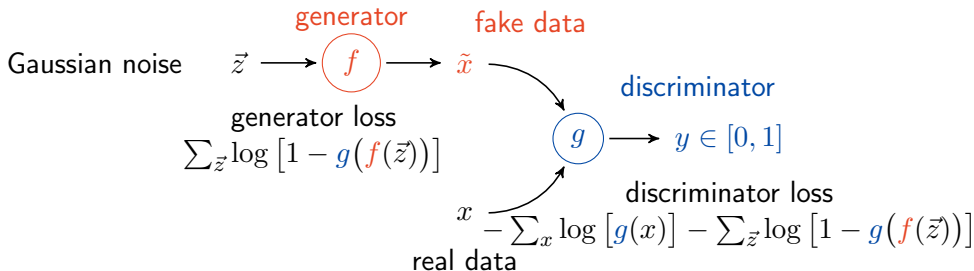
# Generative Adversarial Networks (Goodfellow, Pouget-Abadie, et al. 2014)

- Motivation: Generate realistic-looking 'fake' data, e.g. for data augmentation
- Idea: Let a generator and a discriminator network compete



## Generative Adversarial Networks (Goodfellow, Pouget-Abadie, et al. 2014)

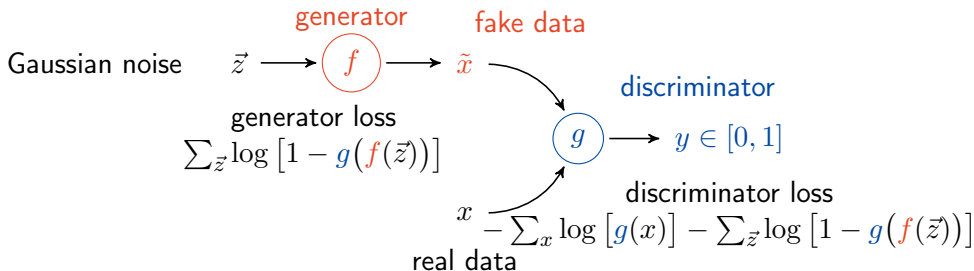
- ▶ Motivation: Generate realistic-looking 'fake' data, e.g. for data augmentation
- ▶ Idea: Let a generator and a discriminator network compete



- ▶ Very versatile architecture (not limited to data types or neural nets)

# Generative Adversarial Networks (Goodfellow, Pouget-Abadie, et al. 2014)

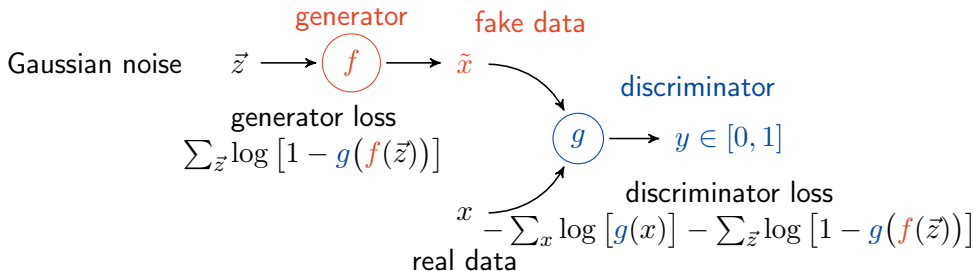
- ▶ Motivation: Generate realistic-looking 'fake' data, e.g. for data augmentation
- ▶ Idea: Let a generator and a discriminator network compete



- ▶ Very versatile architecture (not limited to data types or neural nets)
- ▶ **But:** Notoriously hard to train!

# Generative Adversarial Networks (Goodfellow, Pouget-Abadie, et al. 2014)

- ▶ Motivation: Generate realistic-looking 'fake' data, e.g. for data augmentation
- ▶ Idea: Let a generator and a discriminator network compete

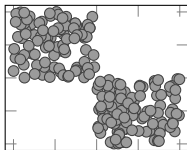


- ▶ Very versatile architecture (not limited to data types or neural nets)
- ▶ **But:** Notoriously hard to train! E.g.: Imbalanced learning speed of generator and discriminator, generator only does exact copies, generator loses variance



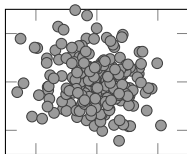
# GAN Example

real data  $x$

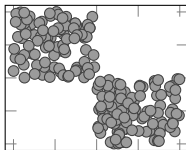


# GAN Example

Gaussian noise  $\vec{z}$

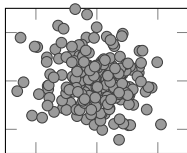


real data  $x$



# GAN Example

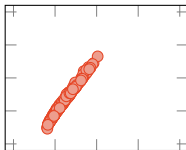
Gaussian noise  $\vec{z}$



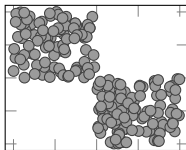
generator



fake data  $\vec{x}$

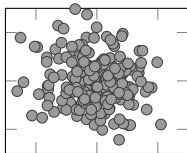


real data  $x$



# GAN Example

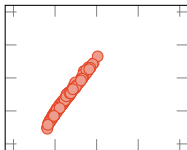
Gaussian noise  $\vec{z}$



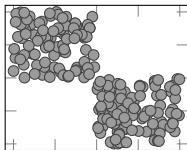
generator



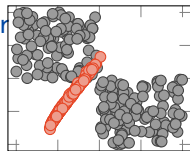
fake data  $\vec{x}$



real data  $x$

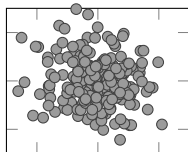


discriminator



# GAN Example

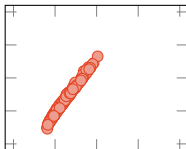
Gaussian noise  $\vec{z}$



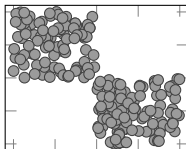
generator



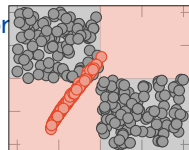
fake data  $\vec{x}$



real data  $x$

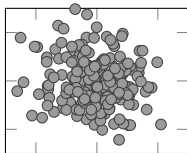


discriminator



# GAN Example

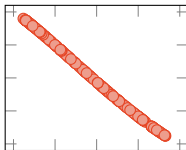
Gaussian noise  $\vec{z}$



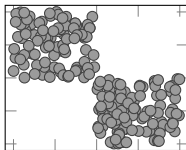
generator



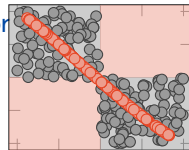
fake data  $\vec{x}$



real data  $x$



discriminator



# Adversarial Attacks



THE UNIVERSITY OF  
SYDNEY

## Examples of Adversarial Attacks

- ▶ An **adversarial attack** is an **imperceptible** change to a data point, such that the **predicted label changes**



## Examples of Adversarial Attacks

- ▶ An **adversarial attack** is an **imperceptible** change to a data point, such that the **predicted label changes**
- ▶ “Robust Physical-World Attacks on Deep Learning Visual Classification” (Eykholt et al. 2018): [Link](#)

## Examples of Adversarial Attacks

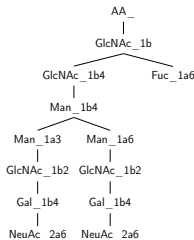
- ▶ An **adversarial attack** is an **imperceptible** change to a data point, such that the **predicted label changes**
- ▶ “Robust Physical-World Attacks on Deep Learning Visual Classification” (Eykholt et al. 2018): [Link](#)
- ▶ “Audio Adversarial Examples: Targeted Attacks on Speech-to-Text” (Carlini and Wagner 2018): [Link](#)

## Examples of Adversarial Attacks

- ▶ An **adversarial attack** is an **imperceptible** change to a data point, such that the **predicted label changes**
- ▶ “Robust Physical-World Attacks on Deep Learning Visual Classification” (Eykholt et al. 2018): [Link](#)
- ▶ “Audio Adversarial Examples: Targeted Attacks on Speech-to-Text” (Carlini and Wagner 2018): [Link](#)
- ▶ “Adversarial Edit Attacks for Tree Data” (Paaßen 2019)

# Examples of Adversarial Attacks

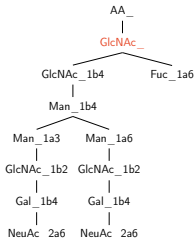
- ▶ An **adversarial attack** is an **imperceptible** change to a data point, such that the **predicted label changes**
- ▶ “Robust Physical-World Attacks on Deep Learning Visual Classification” (Eykholt et al. 2018): [Link](#)
- ▶ “Audio Adversarial Examples: Targeted Attacks on Speech-to-Text” (Carlini and Wagner 2018): [Link](#)
- ▶ “Adversarial Edit Attacks for Tree Data” (Paaßen 2019)



⇒ classified as leukemic

## Examples of Adversarial Attacks

- ▶ An **adversarial attack** is an **imperceptible** change to a data point, such that the **predicted label changes**
- ▶ “Robust Physical-World Attacks on Deep Learning Visual Classification” (Eykholt et al. 2018): [Link](#)
- ▶ “Audio Adversarial Examples: Targeted Attacks on Speech-to-Text” (Carlini and Wagner 2018): [Link](#)
- ▶ “Adversarial Edit Attacks for Tree Data” (Paaßen 2019)



⇒ classified as benign

## Formalization

(Goodfellow, Shlens, and Szegedy 2015)

- ▶ Assume some **perception distance**  $d$  (Göpfert et al. 2019).

## Formalization

(Goodfellow, Shlens, and Szegedy 2015)

- Assume some **perception distance**  $d$  (Göpfert et al. 2019). We define an **adversarial example**  $z$  w.r.t. model  $f$  and point  $x$  as

$$\begin{array}{ll} \min_z & d(x, z) \\ \text{s.t.} & f(z) \neq f(x) \end{array}$$

## Formalization

(Goodfellow, Shlens, and Szegedy 2015)

- Assume some **perception distance**  $d$  (Göpfert et al. 2019). We define an **adversarial example**  $z$  w.r.t. model  $f$ , loss  $\ell$ , threshold  $\epsilon$ , and point  $x$  as

$$\begin{array}{ll} \min_z & d(x, z) \\ \text{s.t.} & f(z) \neq f(x) \end{array} \quad \text{or} \quad \begin{array}{ll} \max_z & \ell(y, f(z)) \\ \text{s.t.} & d(x, z) \leq \epsilon \end{array}$$



## Formalization

(Goodfellow, Shlens, and Szegedy 2015)

- Assume some **perception distance**  $d$  (Göpfert et al. 2019). We define an **adversarial example**  $z$  w.r.t. model  $f$ , loss  $\ell$ , threshold  $\epsilon$ , and point  $x$  as

$$\begin{array}{ll} \min_z d(x, z) & \text{or} \quad \max_z \ell(y, f(z)) \\ \text{s.t. } f(z) \neq f(x) & \text{s.t. } d(x, z) \leq \epsilon \end{array}$$

- **fast gradient sign method**: sign gradient ascent on  $\ell$  and clipping:  
 $z \leftarrow x + \text{sign}[\nabla_x \ell(y, f(x))]$  (Goodfellow, Shlens, and Szegedy 2015)

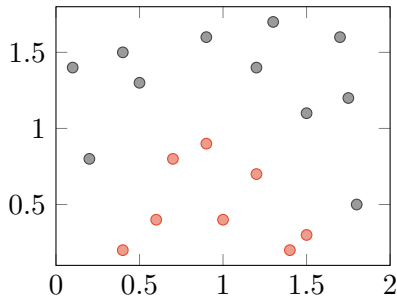
## Formalization

(Goodfellow, Shlens, and Szegedy 2015)

- Assume some **perception distance**  $d$  (Göpfert et al. 2019). We define an **adversarial example**  $z$  w.r.t. model  $f$ , loss  $\ell$ , threshold  $\epsilon$ , and point  $x$  as

$$\begin{array}{ll} \min_z d(x, z) & \text{or} \quad \max_z \ell(y, f(z)) \\ \text{s.t. } f(z) \neq f(x) & \text{s.t. } d(x, z) \leq \epsilon \end{array}$$

- fast gradient sign method**: sign gradient ascent on  $\ell$  and clipping:  
 $z \leftarrow x + \text{sign}[\nabla_x \ell(y, f(x))]$  (Goodfellow, Shlens, and Szegedy 2015)



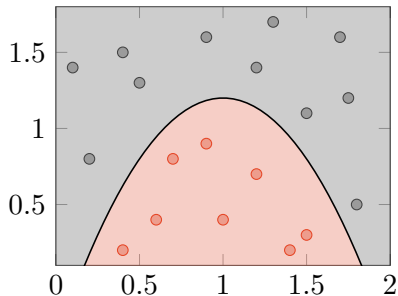
## Formalization

(Goodfellow, Shlens, and Szegedy 2015)

- Assume some **perception distance**  $d$  (Göpfert et al. 2019). We define an **adversarial example**  $z$  w.r.t. model  $f$ , loss  $\ell$ , threshold  $\epsilon$ , and point  $x$  as

$$\begin{array}{ll} \min_z d(x, z) & \text{or} \quad \max_z \ell(y, f(z)) \\ \text{s.t. } f(z) \neq f(x) & \text{s.t. } d(x, z) \leq \epsilon \end{array}$$

- fast gradient sign method**: sign gradient ascent on  $\ell$  and clipping:  
 $z \leftarrow x + \text{sign}[\nabla_x \ell(y, f(x))]$  (Goodfellow, Shlens, and Szegedy 2015)



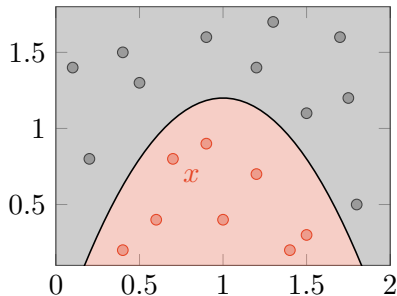
## Formalization

(Goodfellow, Shlens, and Szegedy 2015)

- Assume some **perception distance**  $d$  (Göpfert et al. 2019). We define an **adversarial example**  $z$  w.r.t. model  $f$ , loss  $\ell$ , threshold  $\epsilon$ , and point  $x$  as

$$\begin{array}{ll} \min_z & d(x, z) \\ \text{s.t.} & f(z) \neq f(x) \end{array} \quad \text{or} \quad \begin{array}{ll} \max_z & \ell(y, f(z)) \\ \text{s.t.} & d(x, z) \leq \epsilon \end{array}$$

- fast gradient sign method**: sign gradient ascent on  $\ell$  and clipping:  
 $z \leftarrow x + \text{sign}[\nabla_x \ell(y, f(x))]$  (Goodfellow, Shlens, and Szegedy 2015)



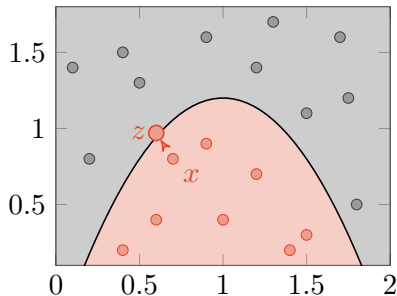
## Formalization

(Goodfellow, Shlens, and Szegedy 2015)

- Assume some **perception distance**  $d$  (Göpfert et al. 2019). We define an **adversarial example**  $z$  w.r.t. model  $f$ , loss  $\ell$ , threshold  $\epsilon$ , and point  $x$  as

$$\begin{array}{ll} \min_z & d(x, z) \\ \text{s.t.} & f(z) \neq f(x) \end{array} \quad \text{or} \quad \begin{array}{ll} \max_z & \ell(y, f(z)) \\ \text{s.t.} & d(x, z) \leq \epsilon \end{array}$$

- fast gradient sign method**: sign gradient ascent on  $\ell$  and clipping:  
 $z \leftarrow x + \text{sign}[\nabla_x \ell(y, f(x))]$  (Goodfellow, Shlens, and Szegedy 2015)

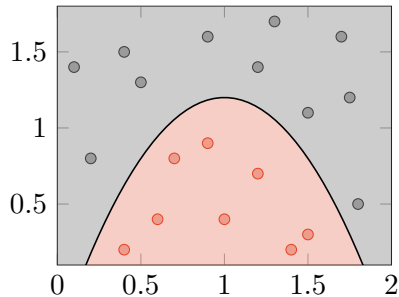


## Defenses against adversarial attacks

- ▶ Existence and quantity of adv. examples depends e.g. on choice of **model**  $f$ , **distance**  $d$ , and **threshold**  $\epsilon$

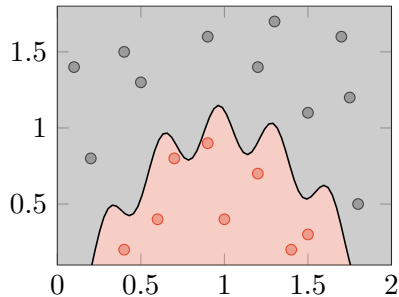
## Defenses against adversarial attacks

- Existence and quantity of adv. examples depends e.g. on choice of **model**  $f$ , **distance**  $d$ , and **threshold**  $\epsilon$



## Defenses against adversarial attacks

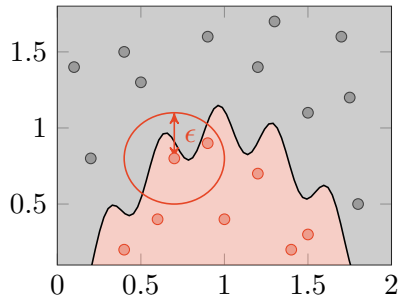
- Existence and quantity of adv. examples depends e.g. on choice of **model**  $f$ , **distance**  $d$ , and **threshold**  $\epsilon$





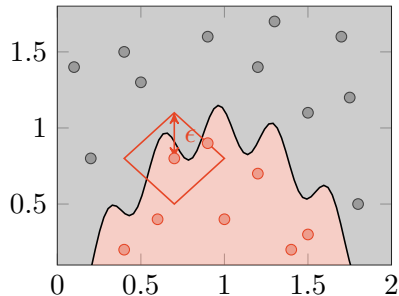
## Defenses against adversarial attacks

- Existence and quantity of adv. examples depends e.g. on choice of **model**  $f$ , **distance**  $d$ , and **threshold**  $\epsilon$



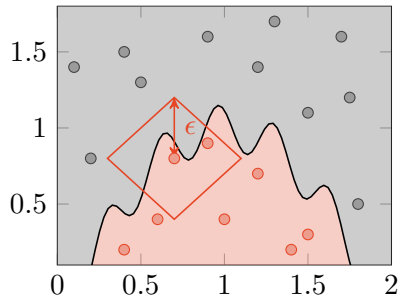
## Defenses against adversarial attacks

- Existence and quantity of adv. examples depends e.g. on choice of **model**  $f$ , **distance**  $d$ , and **threshold**  $\epsilon$



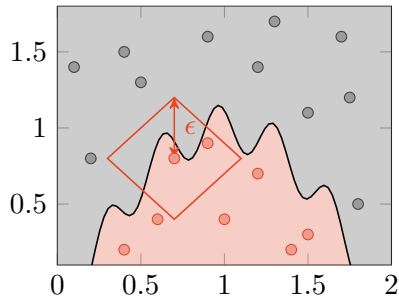
## Defenses against adversarial attacks

- Existence and quantity of adv. examples depends e.g. on choice of **model**  $f$ , **distance**  $d$ , and **threshold**  $\epsilon$



## Defenses against adversarial attacks

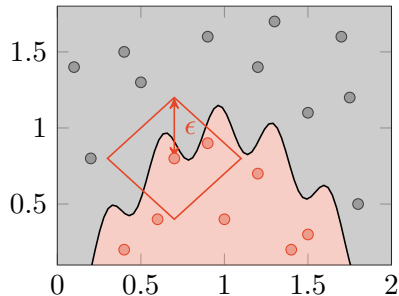
- Existence and quantity of adv. examples depends e.g. on choice of **model**  $f$ , **distance**  $d$ , and **threshold**  $\epsilon$



Rules of Thumb:

## Defenses against adversarial attacks

- Existence and quantity of adv. examples depends e.g. on choice of **model**  $f$ , **distance**  $d$ , and **threshold**  $\epsilon$

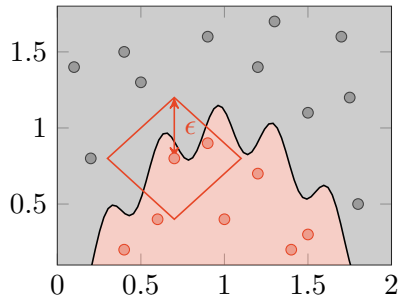


Rules of Thumb:

- Large-Margin methods are harder to attack (e.g. SVM)

## Defenses against adversarial attacks

- ▶ Existence and quantity of adv. examples depends e.g. on choice of **model**  $f$ , **distance**  $d$ , and **threshold**  $\epsilon$

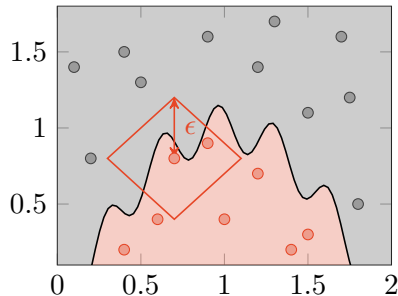


Rules of Thumb:

- ▶ Large-Margin methods are harder to attack (e.g. SVM)
- ▶ High-dimensional inputs are easier to attack (e.g. images)

## Defenses against adversarial attacks

- ▶ Existence and quantity of adv. examples depends e.g. on choice of **model**  $f$ , **distance**  $d$ , and **threshold**  $\epsilon$

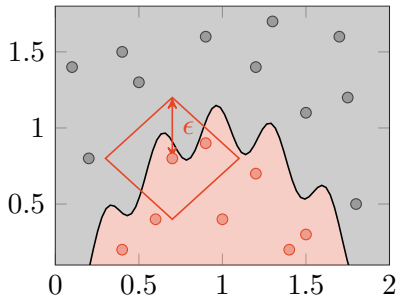


Rules of Thumb:

- ▶ Large-Margin methods are harder to attack (e.g. SVM)
  - ▶ High-dimensional inputs are easier to attack (e.g. images)
- ▶ many (intuitive) defenses don't work (Athalye, Carlini, and Wagner 2018)

## Defenses against adversarial attacks

- ▶ Existence and quantity of adv. examples depends e.g. on choice of **model**  $f$ , **distance**  $d$ , and **threshold**  $\epsilon$



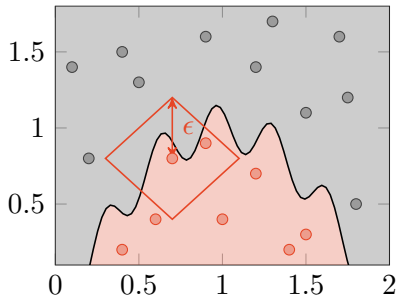
### Rules of Thumb:

- ▶ Large-Margin methods are harder to attack (e.g. SVM)
  - ▶ High-dimensional inputs are easier to attack (e.g. images)
- ▶ many (intuitive) defenses don't work (Athalye, Carlini, and Wagner 2018)
  - ▶ most promising to date: **robust optimization** (i.e. train with adversarials; Madry et al. 2018)



## Defenses against adversarial attacks

- ▶ Existence and quantity of adv. examples depends e.g. on choice of **model**  $f$ , **distance**  $d$ , and **threshold**  $\epsilon$



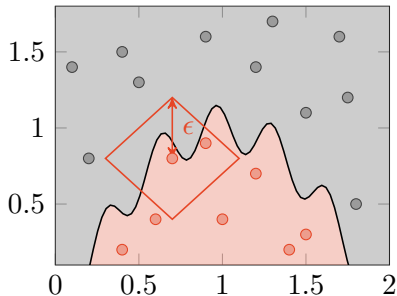
Rules of Thumb:

- ▶ Large-Margin methods are harder to attack (e.g. SVM)
- ▶ High-dimensional inputs are easier to attack (e.g. images)
- ▶ many (intuitive) defenses don't work (Athalye, Carlini, and Wagner 2018)
- ▶ most promising to date: **robust optimization** (i.e. train with adversarials; Madry et al. 2018)

$$\min_f \sum_{i=1}^N \ell(y_i, f(x_i))$$

## Defenses against adversarial attacks

- ▶ Existence and quantity of adv. examples depends e.g. on choice of **model**  $f$ , **distance**  $d$ , and **threshold**  $\epsilon$



Rules of Thumb:

- ▶ Large-Margin methods are harder to attack (e.g. SVM)
  - ▶ High-dimensional inputs are easier to attack (e.g. images)
- ▶ many (intuitive) defenses don't work (Athalye, Carlini, and Wagner 2018)
  - ▶ most promising to date: **robust optimization** (i.e. train with adversarials; Madry et al. 2018)

$$\min_f \sum_{i=1}^N \max_{z_i: d(x_i, z_i) \leq \epsilon} \ell(y_i, f(z_i))$$

# Summary



THE UNIVERSITY OF  
**SYDNEY**

## How to design & train a neural net

- ▶ Use only if 'classic' ML methods don't suffice

## How to design & train a neural net

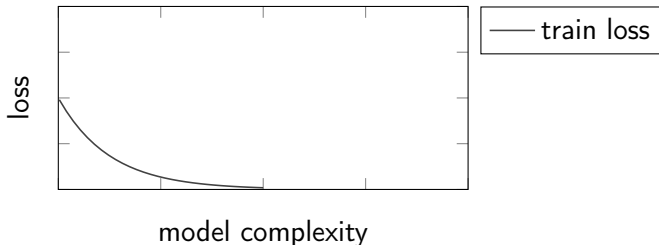
- ▶ Use only if 'classic' ML methods don't suffice
- ▶ Best use a pre-defined architecture with pre-defined loss as template

## How to design & train a neural net

- ▶ Use only if 'classic' ML methods don't suffice
- ▶ Best use a pre-defined architecture with pre-defined loss as template
- ▶ Overparametrize; design a network that can definitely bring the training error to zero; then use regularization to prevent too bad overfitting (Belkin et al. 2019, "double descent" idea)

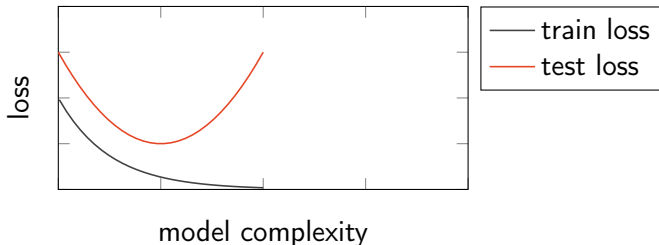
## How to design & train a neural net

- ▶ Use only if 'classic' ML methods don't suffice
- ▶ Best use a pre-defined architecture with pre-defined loss as template
- ▶ Overparametrize; design a network that can definitely bring the training error to zero; then use regularization to prevent too bad overfitting (Belkin et al. 2019, "double descent" idea)



## How to design & train a neural net

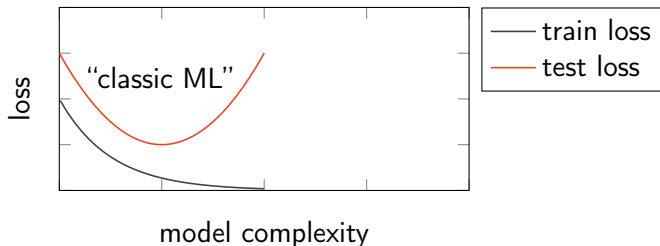
- ▶ Use only if 'classic' ML methods don't suffice
- ▶ Best use a pre-defined architecture with pre-defined loss as template
- ▶ Overparametrize; design a network that can definitely bring the training error to zero; then use regularization to prevent too bad overfitting (Belkin et al. 2019, "double descent" idea)





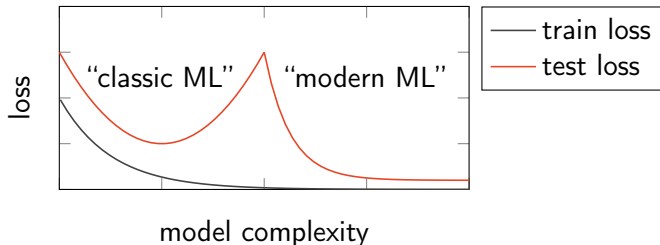
## How to design & train a neural net

- ▶ Use only if 'classic' ML methods don't suffice
- ▶ Best use a pre-defined architecture with pre-defined loss as template
- ▶ Overparametrize; design a network that can definitely bring the training error to zero; then use regularization to prevent too bad overfitting (Belkin et al. 2019, "double descent" idea)



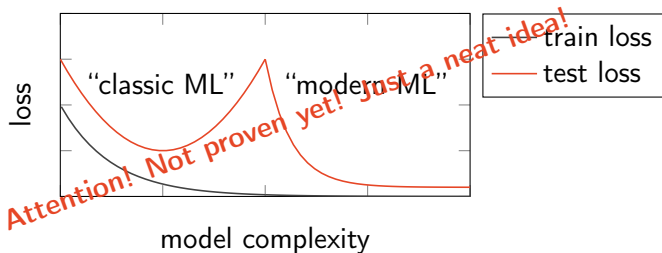
## How to design & train a neural net

- ▶ Use only if 'classic' ML methods don't suffice
- ▶ Best use a pre-defined architecture with pre-defined loss as template
- ▶ Overparametrize; design a network that can definitely bring the training error to zero; then use regularization to prevent too bad overfitting (Belkin et al. 2019, "double descent" idea)



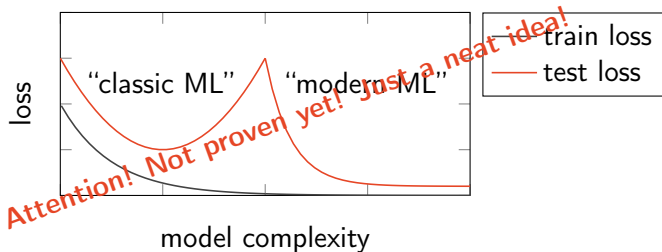
## How to design & train a neural net

- ▶ Use only if 'classic' ML methods don't suffice
- ▶ Best use a pre-defined architecture with pre-defined loss as template
- ▶ Overparametrize; design a network that can definitely bring the training error to zero; then use regularization to prevent too bad overfitting (Belkin et al. 2019, "double descent" idea)



## How to design & train a neural net

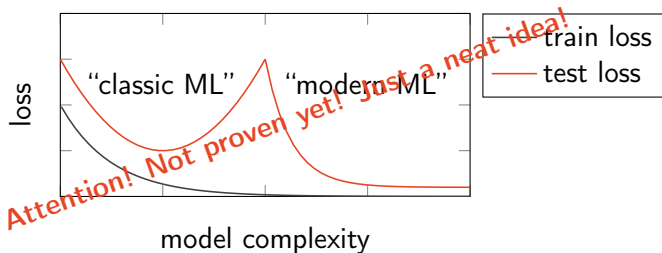
- ▶ Use only if 'classic' ML methods don't suffice
- ▶ Best use a pre-defined architecture with pre-defined loss as template
- ▶ Overparametrize; design a network that can definitely bring the training error to zero; then use regularization to prevent too bad overfitting (Belkin et al. 2019, "double descent" idea)



- ▶ Train in batches, not with single points

## How to design & train a neural net

- ▶ Use only if 'classic' ML methods don't suffice
- ▶ Best use a pre-defined architecture with pre-defined loss as template
- ▶ Overparametrize; design a network that can definitely bring the training error to zero; then use regularization to prevent too bad overfitting (Belkin et al. 2019, "double descent" idea)



- ▶ Train in batches, not with single points
- ▶ Perform lots of soundness check early on; monitor your progress well

# Literature



THE UNIVERSITY OF  
SYDNEY

## Literature I

- Athalye, Anish, Nicholas Carlini, and David Wagner (2018). “Obfuscated Gradients Give a False Sense of Security: Circumventing Defenses to Adversarial Examples”. In: **Proceedings of the 35th International Conference on Machine Learning (ICML 2018)**. Ed. by Jennifer Dy and Andreas Krause. PMLR, pp. 274–283. URL: <http://proceedings.mlr.press/v80/athalye18a.html>.
- Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio (2014). “Neural machine translation by jointly learning to align and translate”. In: **Proceedings of the 2nd International Conference on Learning Representations (ICLR 2014)**. URL: <https://arxiv.org/abs/1409.0473>.
- Belkin, Mikhail et al. (2019). “Reconciling modern machine-learning practice and the classical bias–variance trade-off”. In: **Proceedings of the National Academy of Sciences** 116.32, pp. 15849–15854. DOI: 10.1073/pnas.1903070116. URL: <https://arxiv.org/abs/1812.11118>.

## Literature II

Bridle, John S. (1990). “Probabilistic Interpretation of Feedforward Classification Network Outputs, with Relationships to Statistical Pattern Recognition”. In: **Neurocomputing**. Ed. by Françoise Fogelman Soulié and Jeanny Hérault. Berlin/Heidelberg, Germany: Springer, pp. 227–236. DOI: 10.1007/978-3-642-76153-9\_28.

Carlini, Nicholas and David Wagner (2018). “Audio Adversarial Examples: Targeted Attacks on Speech-to-Text”. In: **Proceedings of the 2018 IEEE Security and Privacy Workshops (SPW 2018)**, pp. 1–7. DOI: 10.1109/SPW.2018.00009. URL: <https://arxiv.org/abs/1801.01944>.

Chellapilla, Kumar, Sidd Puri, and Patrice Simard (2006). “High Performance Convolutional Neural Networks for Document Processing”. In: **Proceedings of the 10th International Workshop on Frontiers in Handwriting Recognition**. Ed. by Guy Lorette. URL: <https://hal.inria.fr/inria-00112631/>.



## Literature III

- Cho, Kyunghyun et al. (2014). “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation”. In: **Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)**. Ed. by Alessandro Moschitti, Bo Pang, and Walter Daelemans, pp. 1724–1734. URL: <https://www.aclweb.org/anthology/D14-1179>.
- Deng, J. et al. (2009). “ImageNet: A large-scale hierarchical image database”. In: **Proceedings of the 22nd IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2009)**, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.
- Eykholt, Kevin et al. (2018). “Robust Physical-World Attacks on Deep Learning Visual Classification”. In: **Proceedings of the 31st IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2018)**. Ed. by Michael Brown et al., pp. 1625–1634. URL: <https://arxiv.org/abs/1707.08945>.

## Literature IV

- Fukushima, Kunihiro, Sei Miyake, and Takayuki Ito (1983). “Neocognitron: A neural network model for a mechanism of visual pattern recognition”. In: **IEEE Transactions on Systems, Man, and Cybernetics** SMC-13.5, pp. 826–834. DOI: 10.1109/TSMC.1983.6313076.
- Goodfellow, Ian, Jean Pouget-Abadie, et al. (2014). “Generative Adversarial Nets”. In: **Proceedings of the 27th International Conference on Advances in Neural Information Processing Systems (NIPS 2014)**. Ed. by Z. Ghahramani et al., pp. 2672–2680. URL: <http://papers.nips.cc/paper/5423-generative-adversarial-nets>.
- Goodfellow, Ian, Jonathon Shlens, and Christian Szegedy (2015). “Explaining and Harnessing Adversarial Examples”. In: **Proceedings of the 3rd International Conference on Learning Representations (ICLR 2015)**. Ed. by Yoshua Bengio and Yann LeCun. URL: <http://arxiv.org/abs/1412.6572>.
- Göpfert, Jan Philip et al. (2019). **Adversarial attacks hidden in plain sight**. arXiv: 1902.09286 [stat.ML].

## Literature V

- He, Kaiming et al. (2016). “Deep Residual Learning for Image Recognition”. In: **Proceedings of the 29th IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2016)**, pp. 770–778. URL: [http://openaccess.thecvf.com/content\\_cvpr\\_2016/html/He\\_Deep\\_Residual\\_Learning\\_CVPR\\_2016\\_paper.html](http://openaccess.thecvf.com/content_cvpr_2016/html/He_Deep_Residual_Learning_CVPR_2016_paper.html).
- Hochreiter, Sepp and Jürgen Schmidhuber (1997). “Long Short-Term Memory”. In: **Neural Computation** 9.8, pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.
- Hopfield, John (1982). “Neural networks and physical systems with emergent collective computational abilities”. In: **Proceedings of the National Academy of Sciences** 79.8, pp. 2554–2558. DOI: 10.1073/pnas.79.8.2554.
- Ioffe, Sergey and Christian Szegedy (2015). **Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift**. arXiv: 1502.03167 [cs.LG].

## Literature VI

- Jarrett, K. et al. (2009). “What is the best multi-stage architecture for object recognition?” In: **Proceedings of the 12th IEEE International Conference on Computer Vision (ICCV 2009)**, pp. 2146–2153. DOI: 10.1109/ICCV.2009.5459469.
- Kingma, Diederik and Max Welling (2013). **Auto-Encoding Variational Bayes**. arXiv: 1312.6114 [stat.ML].
- Krizhevsky, Alex, Ilya Sutskever, and Geoffrey Hinton (2012). “ImageNet Classification with Deep Convolutional Neural Networks”. In: **Proceedings of the 25th International Conference on Advances in Neural Information Processing Systems (NIPS 2012)**. Ed. by F. Pereira et al., pp. 1097–1105. URL: <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks>.
- LeCun, Yann and Yoshua Bengio (1995). “Convolutional networks for images, speech, and time series”. In: **The handbook of brain theory and neural networks**. Cambridge, MA, USA: MIT Press, pp. 276–278.

## Literature VII

- LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). “Deep learning”. In: **Nature** 521, pp. 436–444. DOI: 10.1038/nature14539.
- Linnainmaa, Seppo (1970). “The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors”. MA thesis. University of Helsinki.
- Madry, Aleksander et al. (2018). “Towards Deep Learning Models Resistant to Adversarial Attacks”. In: **Proceedings of the 6th International Conference on Learning Representations (ICLR 2018)**. Ed. by Yoshua Bengio and Yann LeCun. URL: <https://openreview.net/forum?id=rJzIBfZAb>.
- McCulloch, Warren and Walter Pitts (1943). “A logical calculus of the ideas immanent in nervous activity”. In: **The Bulletin of Mathematical Biophysics** 5.4, pp. 115–133. DOI: 10.1007/BF02478259.
- Minsky, Marvin and Seymour Papert (1969). **Perceptrons: An introduction to computational geometry**. Cambridge, MA, USA: MIT Press.

## Literature VIII

- Olazaran, Mikel (1996). “A Sociological Study of the Official History of the Perceptrons Controversy”. In: **Social Studies of Science** 26.3, pp. 611–659. DOI: 10.1177/030631296026003005.
- Paaßen, Benjamin (2019). “Adversarial Edit Attacks for Tree Data”. In: **Proceedings of the 20th International Conference on Intelligent Data Engineering and Automated Learning (IDEAL 2019)**. Ed. by Hujun Yin et al., pp. 359–366. DOI: 10.1007/978-3-030-33607-3\_39. URL: <https://arxiv.org/abs/1908.09364>.
- Rosenblatt, Frank (1958). “The perceptron: A probabilistic model for information storage and organization in the brain”. In: **Psychological Review** 65.6, pp. 386–408. DOI: 10.1037/h0042519.
- Rumelhart, David, Geoffrey Hinton, and Ronald Williams (1986). “Learning representations by back-propagating errors”. In: **nature** 323.6088, pp. 533–536. DOI: 10.1038/323533a0.

## Literature IX

- Srivastava, Nitish et al. (2014). “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: **Journal of Machine Learning Research** 15, pp. 1929–1958. URL: <http://jmlr.org/papers/v15/srivastava14a.html>.
- Sutskever, Ilya, Oriol Vinyals, and Quoc V Le (2014). “Sequence to Sequence Learning with Neural Networks”. In: **Proceedings of the 27th International Conference on Advances in Neural Information Processing Systems (NIPS 2014)**. Ed. by Z. Ghahramani et al., pp. 3104–3112. URL: <http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural>.
- Vaswani, Ashish et al. (2017). “Attention is All you Need”. In: **Proceedings of the 30th International Conference on Advances in Neural Information Processing Systems (NIPS 2017)**. Ed. by I. Guyon et al., pp. 5998–6008. URL: <http://papers.nips.cc/paper/7181-attention-is-all-you-need>.
- Von Neumann, John (1945). **First Draft of a Report on the EDVAC**. Tech. rep. University of Pennsylvania. URL: <https://nsu.ru/xmlui/bitstream/handle/nsu/9018/2003-08-TheFirstDraft.pdf>.

## Literature X

- Werbos, Paul (1974). “Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences”. PhD thesis. Harvard University.
- (1990). “Backpropagation through time: what it does and how to do it”. In: **Proceedings of the IEEE** 78.10, pp. 1550–1560. DOI: 10.1109/5.58337.