Neural Networks I

Machine Learning 1 — Lecture 7

26^{th} April 2022

Robert Peharz
Institute of Theoretical Computer Science
Graz University of Technology

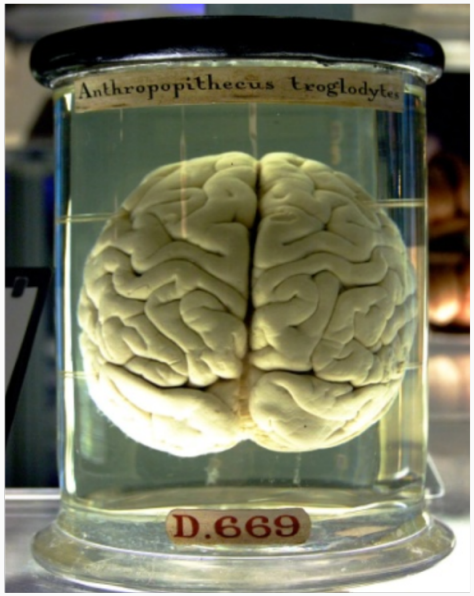
Brief Appetizer Video

https://www.youtube.com/watch?v=Suevq-kZdIw

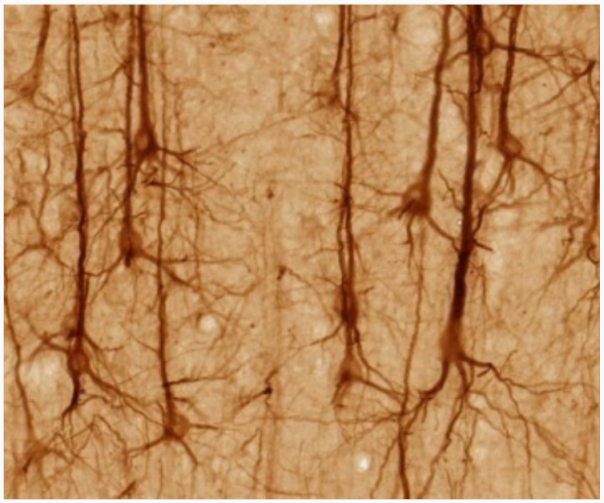
Excellent Tutorial by 3Blue1Brown [optional, recommended]

https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi

Biological Neural Networks



- The brain: still the only "truly" intelligent device on earth
- A biological neural network of 100 billion **neurons**, connected via 100 trillions of **synapses**
- Extremely complex and parallel



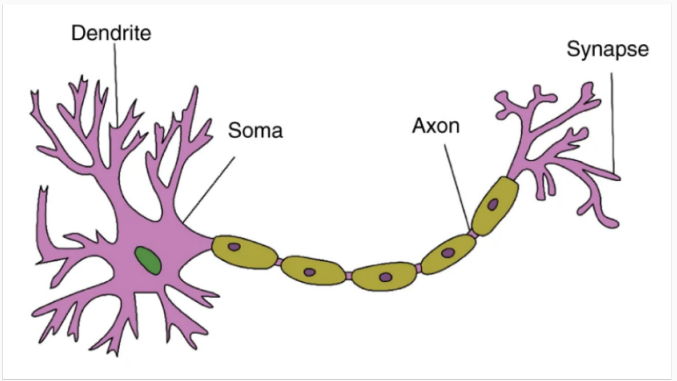


Image: Zhang et al., Artificial neural networks enabled by nanophotonics, 2019

Biological vs. Artificial Neuron

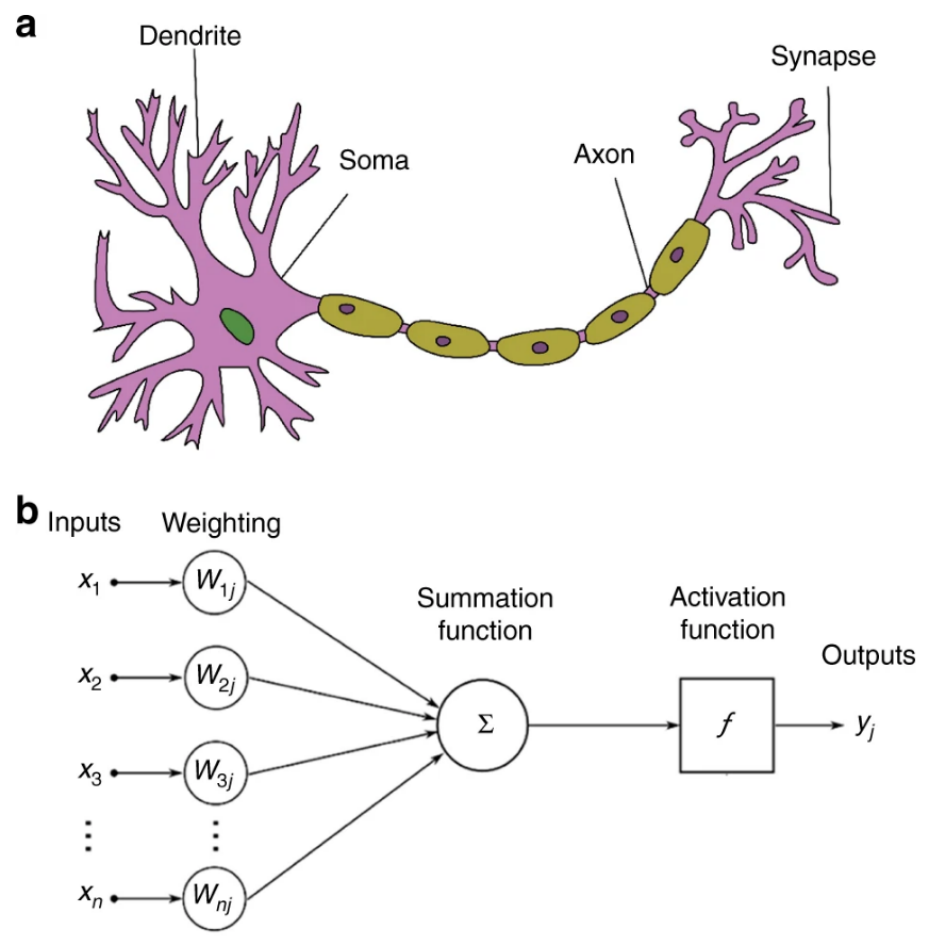
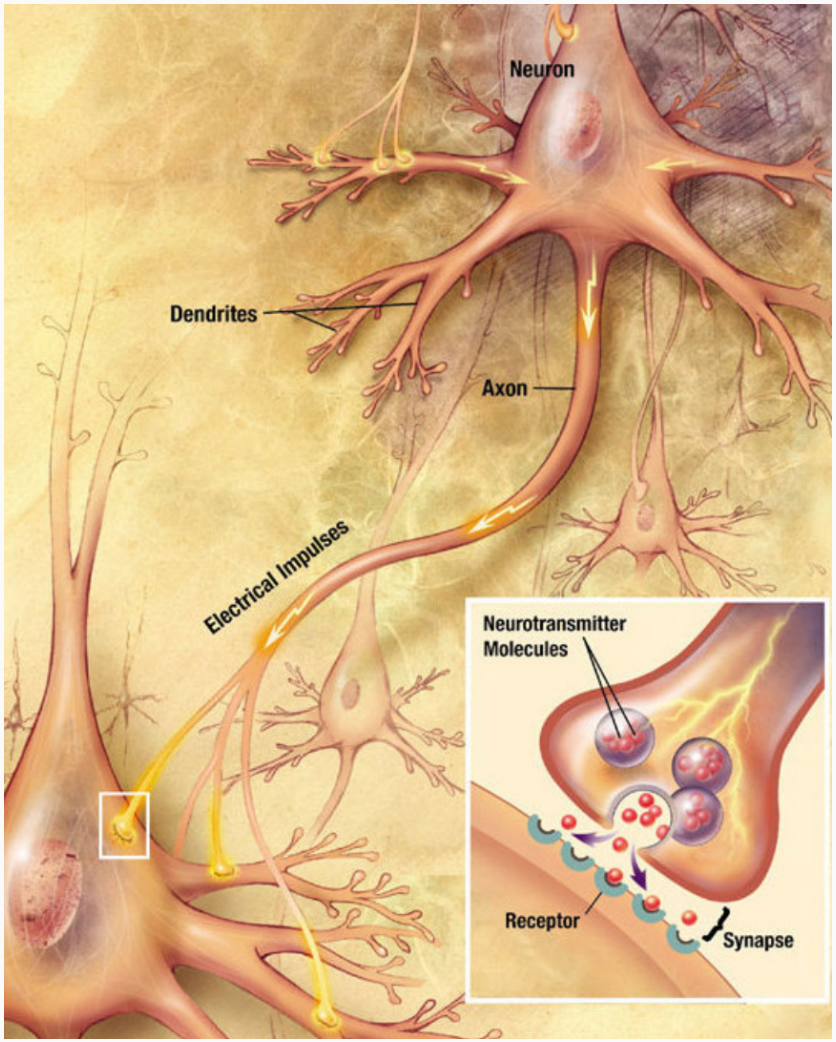


Image: Zhang et al., 2019

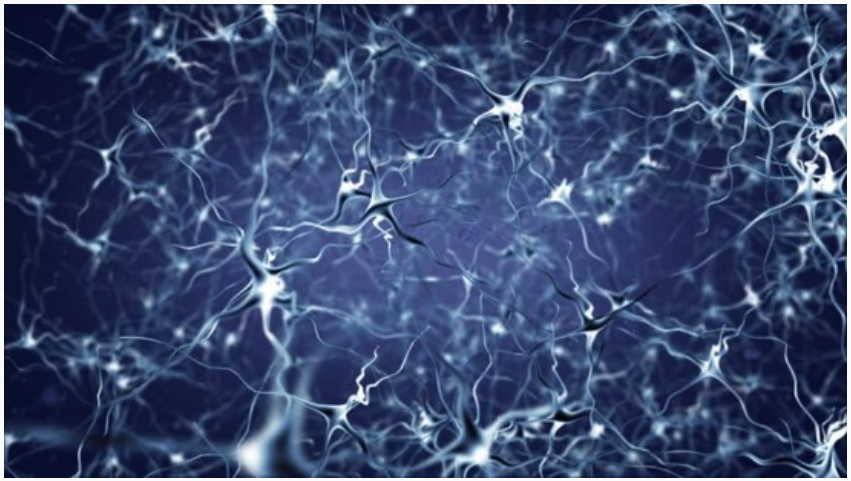
- **dendrite**: receive input spikes from previous neurons
- **soma**: accumulate input, fires/spikes if excited enough i)
- **axon**: long way transmission of spikes
- **synapses**: connection to next neurons (connection strength)
- **In ANNs:**
  - accumulation is $\approx \sum$
  - spiking is $\approx$ non-linear activation function
  - synapse strength is $\approx$ weights

Synapses

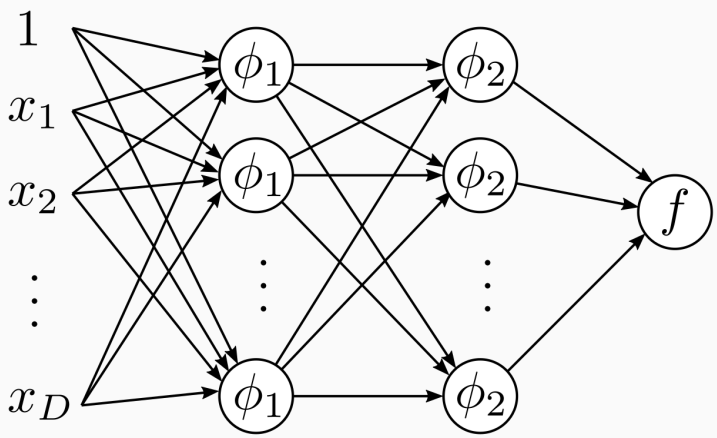


- Synapses are essential for the communication between neurons
- They are essentially a link transmitting spikes from the pre-synaptic to the post-synaptic neuron
- **Synaptic strength** can be adapted is an essential mechanism for learning

Biological vs. Artificial Neural Networks



Biological neural network



Artificial neural network

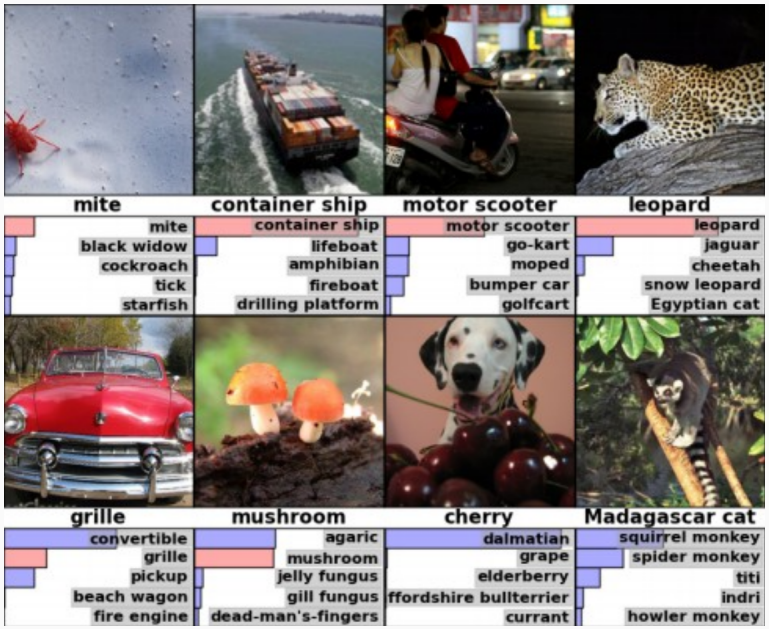
- Artificial neural networks (ANNs): learn from biology
- Two way street: Use ANNs to understand biological neural systems better
  - Develop biologically accurate neuron models
  - Simulations of biological neural networks
  - Spiking ANNs
- For ML purposes: **powerful function approximators**

Brief History

- 1943 – McCulloch and Pitts – *simple electrical circuit for connectionism*
- 1949 – Donald Hebb – *Hebbian learning*
- 1958 – Frank Rosenblatt – *Perceptron*
- $\approx$ 1970-1980 – "AI Winter"
- 1982 – Jon Hopfield – *Hopfield net*
- 1986 – Rumelhart, Hinton and Williams – *Backpropagation* (re-invented from 1960, actually just the chain rule)
- 1989 – Yann LeCun – *Convolutional Neural Networks* (image models)
- 1997 – Sepp Hochreiter and Jürgen Schmidhuber – *Long Short Term Memory Networks*
- $\approx$ 1995–2006 – Neural networks were not popular (lighter models were preferred, e.g. support vector machines)
- 2006 – Hinton, Osindero, Teh – *Deep Learning, Deep Belief Networks*
- 2012 – Krizhevsky, Sutskever, Hinton: New state of the art on ImageNet
- After 2012 – extremely fast growth in popularity due to their success in real-world computer vision problems, natural language processing, etc.

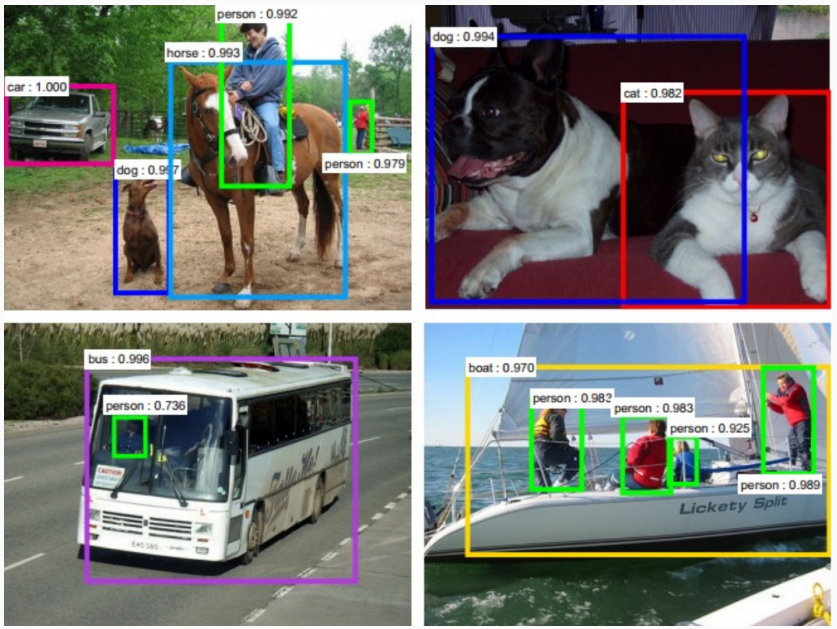
Deep Learning in Computer Vision

Image recognition



Krizhevsky et al., 2012

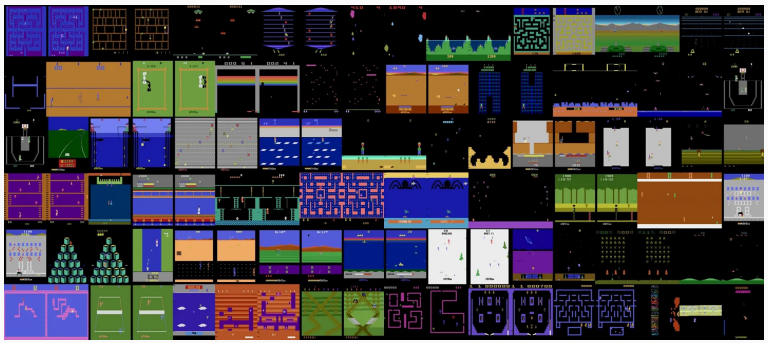
Object Detection



Ren et al., 2015

Deep Learning in Game Playing

Atari Games



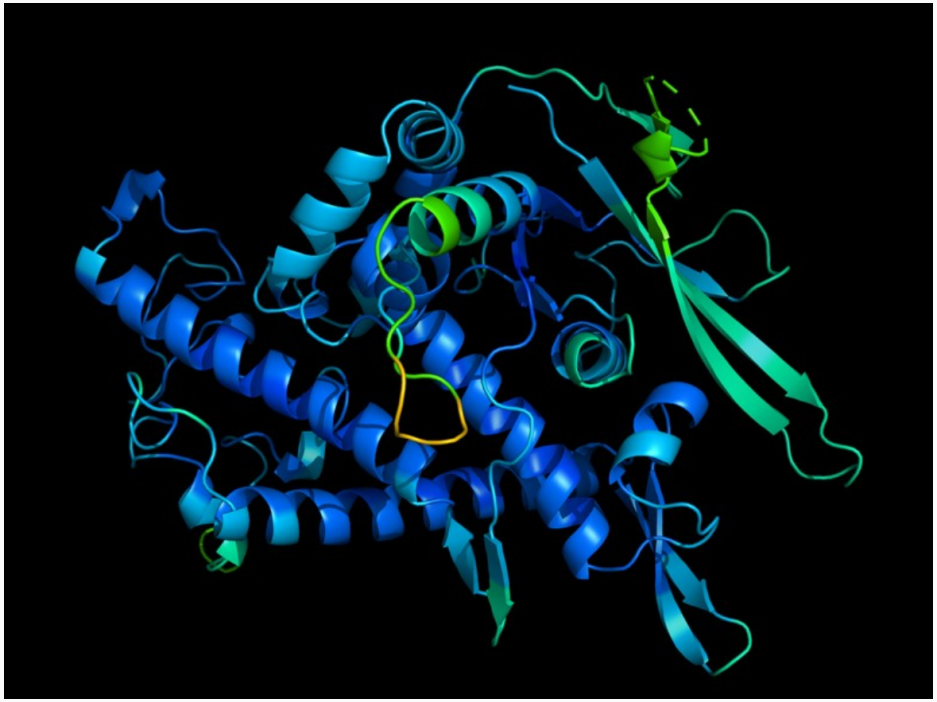
Mnih et al., 2015

Board Games

Silver et al., 2017

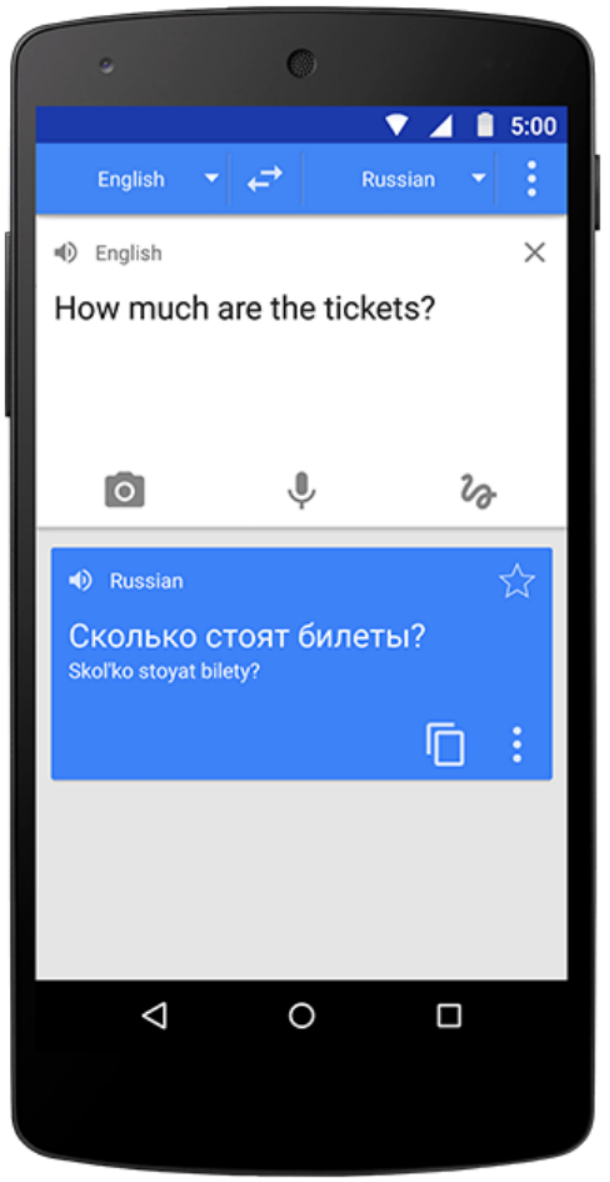
Deep Learning for Protein Folding

Alpha Fold



Senior et al., 2020

Deep Learning for Machine Translation



- **Long-short term memories**
  [Hochreiter and Schmidhuber, 1997]
- **Transformers** [Vaswani et al., 2017]

Neural Networks: Basic Principle

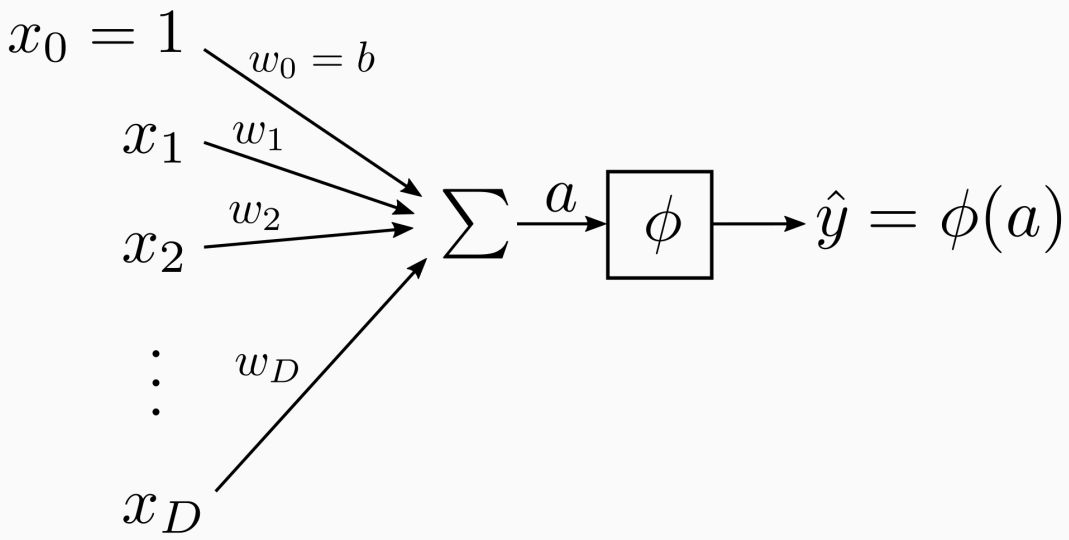
Neuron, Unit

The **neuron** (**unit**) is the basic building block of neural networks:

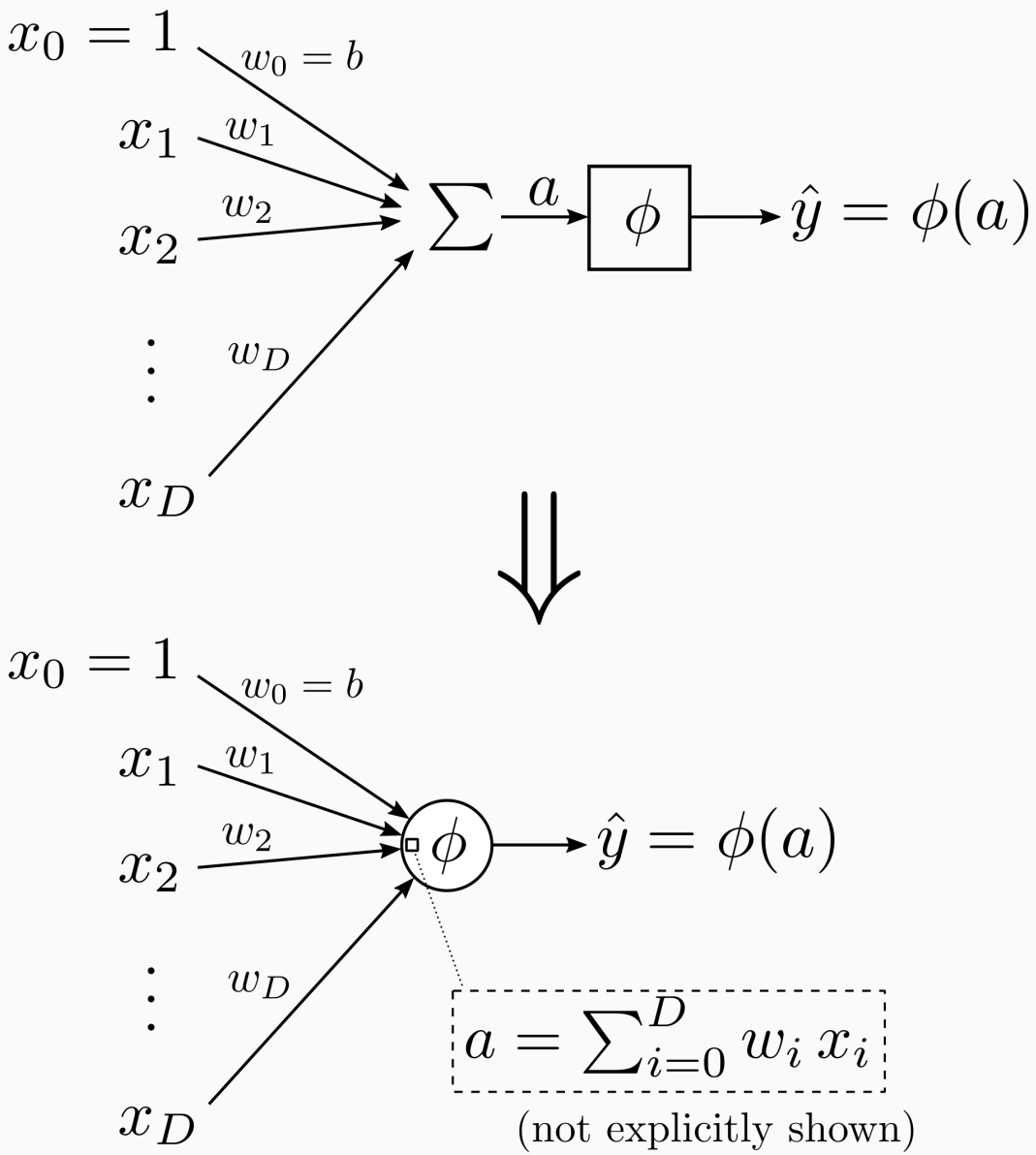
$$\hat{y} = \phi(a) = \phi\left(\sum_{i=0}^{D} w_i x_i\right)$$

- D: number of inputs
- x_i: inputs (features, outputs from previous neurons)
- w_i: weights
- w_0: bias term ($x_0 \equiv 1$)
- ϕ: non-linear **activation function**
- a: **activation** $\sum_{i=0}^{D} w_i x_i$
- $\hat{y}$: output $\phi(a)$
- **Affine function of x, followed by non-linearity ϕ**

Neuron as Computational Graph

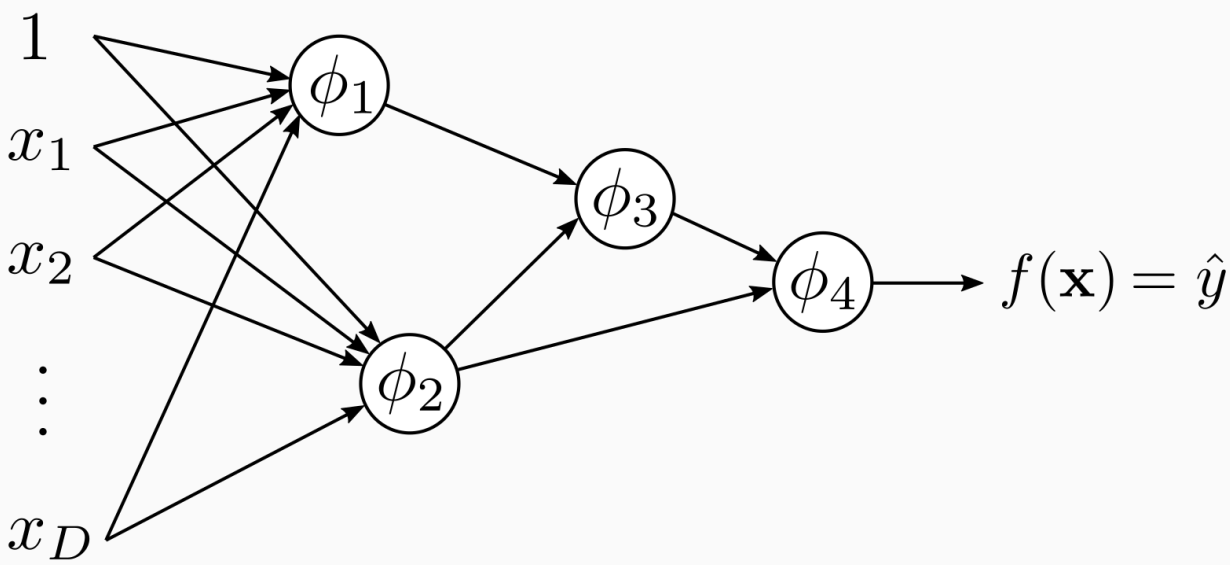


Neuron as Computational Graph

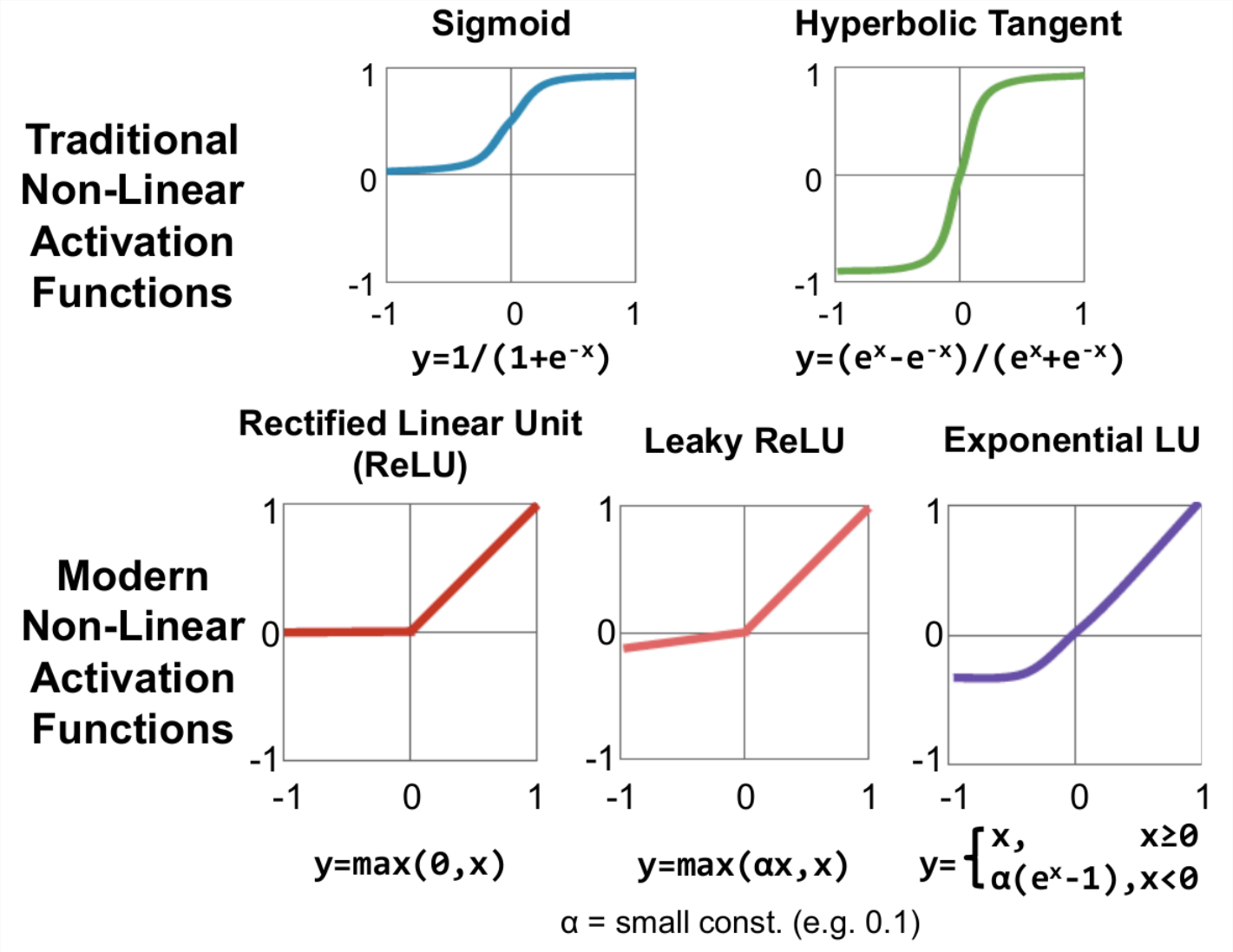


Neural Networks: Network of Neurons

- Like in biological neural nets, ANNs are networks of neurons
- Each neuron is simple, but when combined, they can represent complicated functions $f(\boldsymbol{x})$, to be used for
  - Classification
  - Regression
  - Advanced applications
- Training by optimizing the weights (discussed later)



Common Activation Functions ϕ

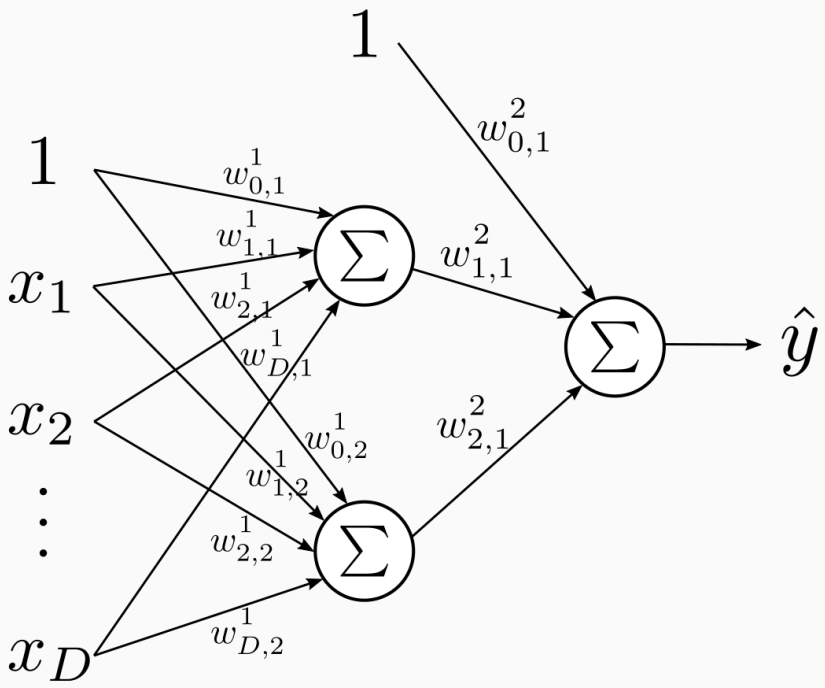


Source: Sze et al., Efficient Processing of Deep Neural Networks: A Tutorial and Survey, 2017.

Why do we need non-linear activation functions ϕ?

Why do we need non-linear activation functions ϕ?

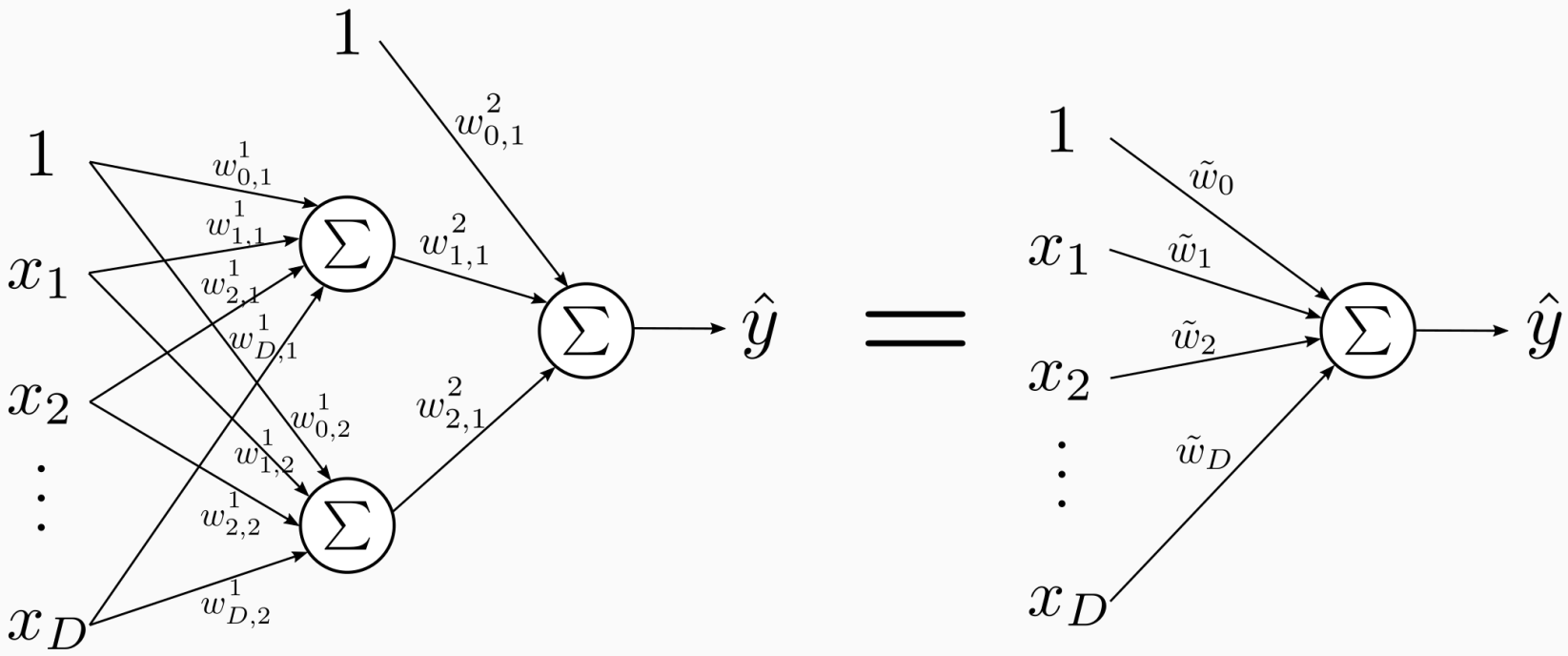
1. Linear functions often not adequate for real-world datasets.
2. Plug-and-play philosophy of ANNs: Construct complicated functions out of simple ones.
   This does not work with linear neurons!



$$\hat{y} = w^2_{0,1} + \sum_{i=1}^{2} w^2_{i,1} \left(\sum_{j=0}^{D} w^1_{j,i} x_j \right)$$

Neural Network of Linear Units

Example



$$\hat{y} = w^2_{0,1} + \sum_{i=1}^{2} w^2_{i,1} \left(\sum_{j=0}^{D} w^1_{j,i} x_j \right) = \sum_{j=0}^{D} \tilde{w}_j x_j$$

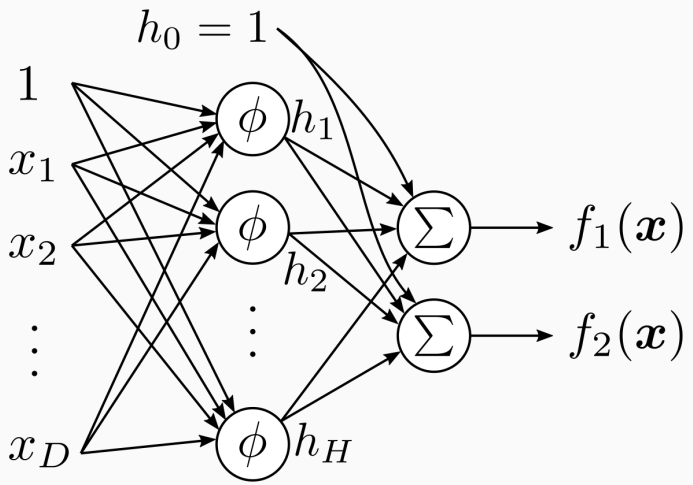
$$\tilde{w}_0 = w^2_{0,1} + w^1_{0,1}\, w^2_{1,1} + w^1_{0,2}\, w^2_{2,1}$$
$$\tilde{w}_j = w^1_{j,1}\, w^2_{1,1} + w^1_{j,2}\, w^2_{2,1}$$

Neural network of linear units = linear unit!

Neural network of linear units = linear unit!

Does non-linearity save us?

- **input layer** $\boldsymbol{x}$
- **hidden layer** with H **hidden units**, $h_j = \phi\left(\sum_{i=0}^{D} w_{i,j}^1 x_i\right)$
- hidden units compute "unobserved" features
- again, include bias with dummy feature $h_0 \equiv 1$
- $\phi\colon \mathbb{R} \mapsto \mathbb{R}$ is a fixed non-linear **activation** function
- **output layer**
  $\hat{y}_k = f_k(\boldsymbol{x}) = \sum_{j=0}^{H} w_{j,k}^2 h_j = \sum_{j=0}^{H} w_{j,k}^2 \, \phi\left(\sum_{i=0}^{D} w_{i,j}^1 x_i\right)$



(weights not shown in the graph)

Universal Approximation Theorem

Theorem **[A. Pinkus, Acta Numerica, 1999]**

Any continuous function $f : \mathbb{R}^D \mapsto \mathbb{R}^O$ can be arbitrarily well approximated (on any compact set) by a single-layer neural network, for a wide range of non-linearities ϕ (non-polynomials).

From Linear Models to Neural Networks

- In Lecture 3, we introduced **linear models**:

$$\hat{y} = \sum_{i=0}^{D} w_i x_i$$

- In Lecture 4, we generalized them by considering fixed **non-linear** features:

$$\hat{y} = \sum_{i=0}^{H} w_i \, \phi_i(\boldsymbol{x})$$

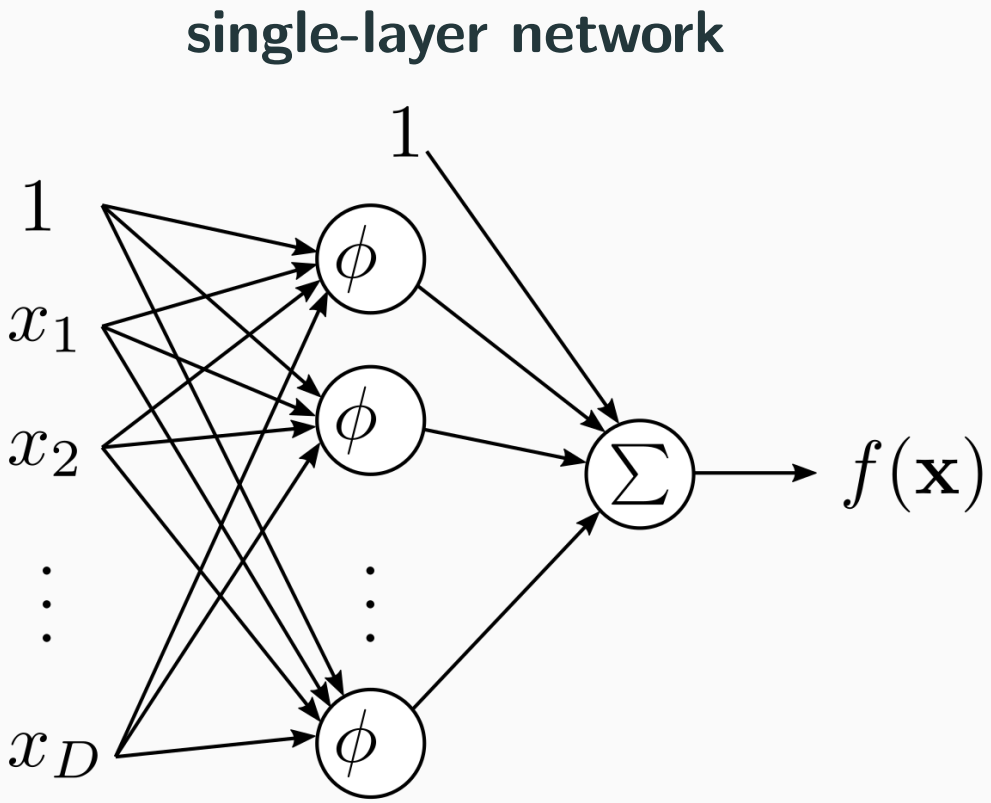
- Single-layer neural nets can be seen as linear models (output layer) with **learnable** non-linear features (hidden layer):

$$\hat{y} = \sum_{j=0}^{H} w_{j,1}^{2} \, \phi\Big(\sum_{i=0}^{D} w_{i,j}^{1} x_i \Big)$$

Multilayer Perceptrons

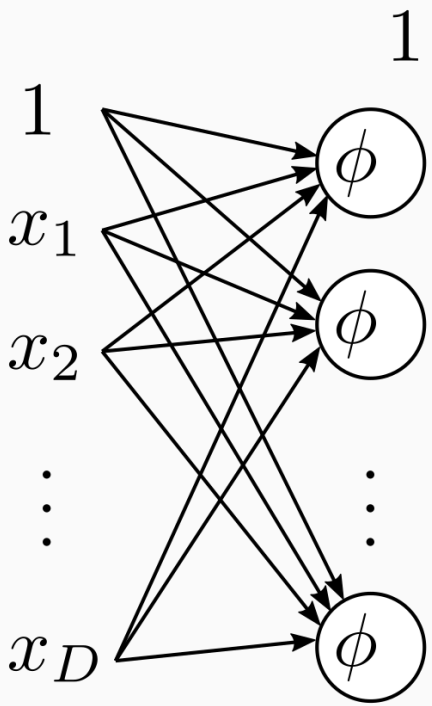
Multi Layer Perceptrons

- Stacking layers yields **multi-layer feed-forward neural networks**, often called **multi layer perceptrons** (MLPs)



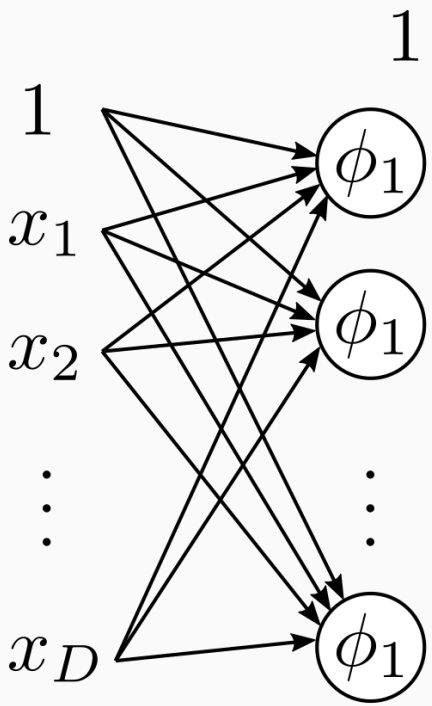
Multi Layer Perceptrons

- Stacking layers yields **multi-layer feed-forward neural networks**, often called **multi layer perceptrons** (MLPs)



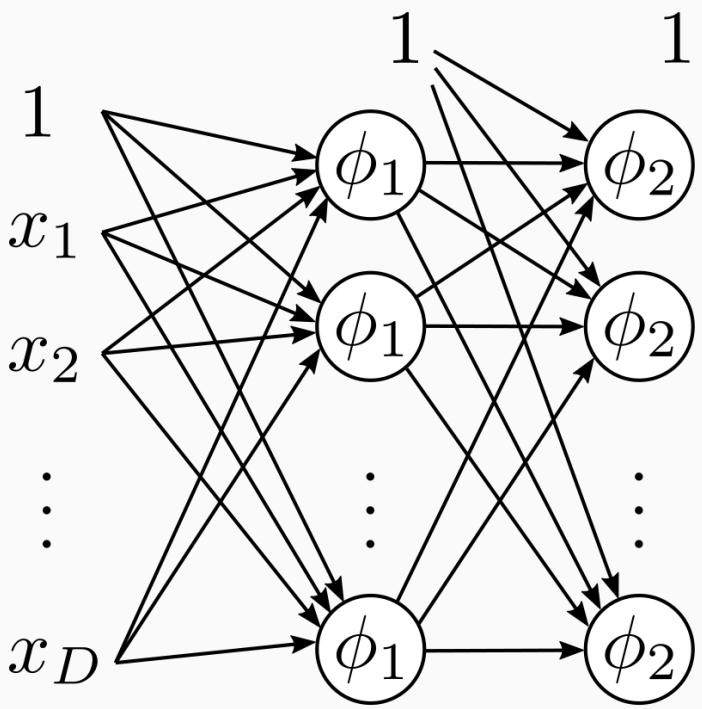
Multi Layer Perceptrons

- Stacking layers yields **multi-layer feed-forward neural networks**, often called **multi layer perceptrons** (MLPs)



Multi Layer Perceptrons

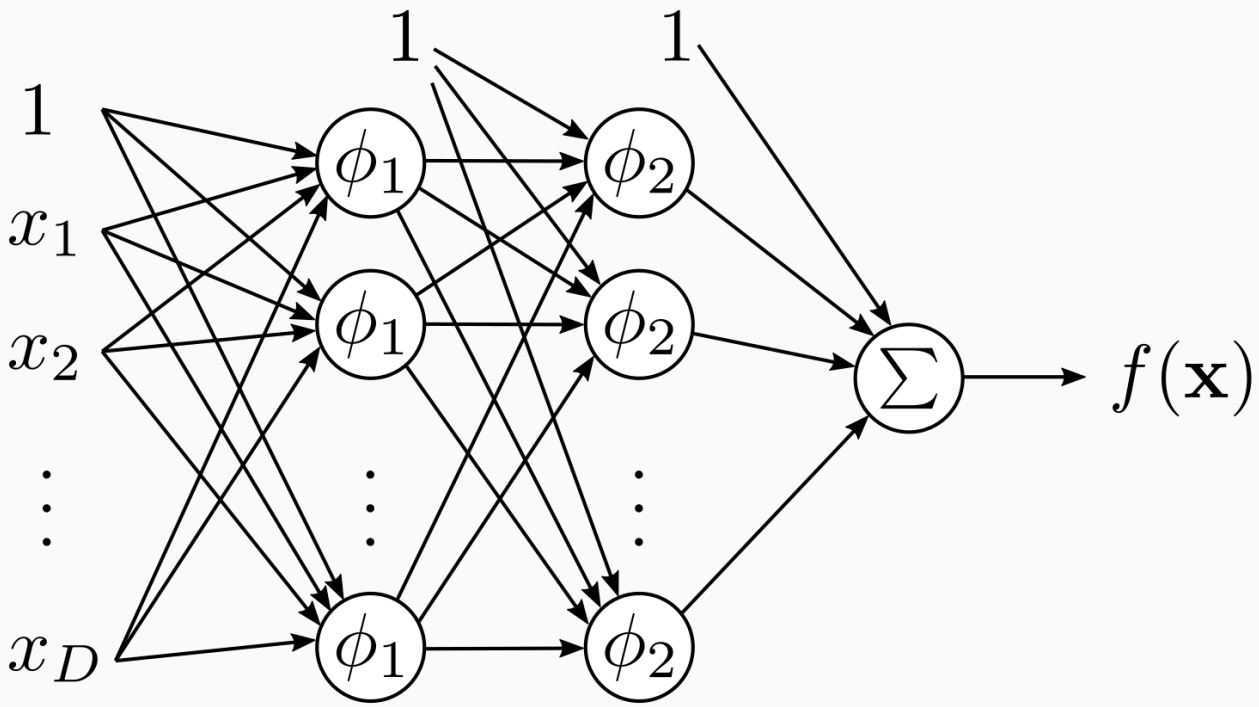
- Stacking layers yields **multi-layer feed-forward neural networks**, often called **multi layer perceptrons** (MLPs)
- Hidden layers can contain different numbers of hidden units H_1, H_2, ...



Multi Layer Perceptrons

- Stacking layers yields **multi-layer feed-forward neural networks**, often called **multi layer perceptrons** (MLPs)
- Hidden layers can contain different numbers of hidden units H_1, H_2, ...

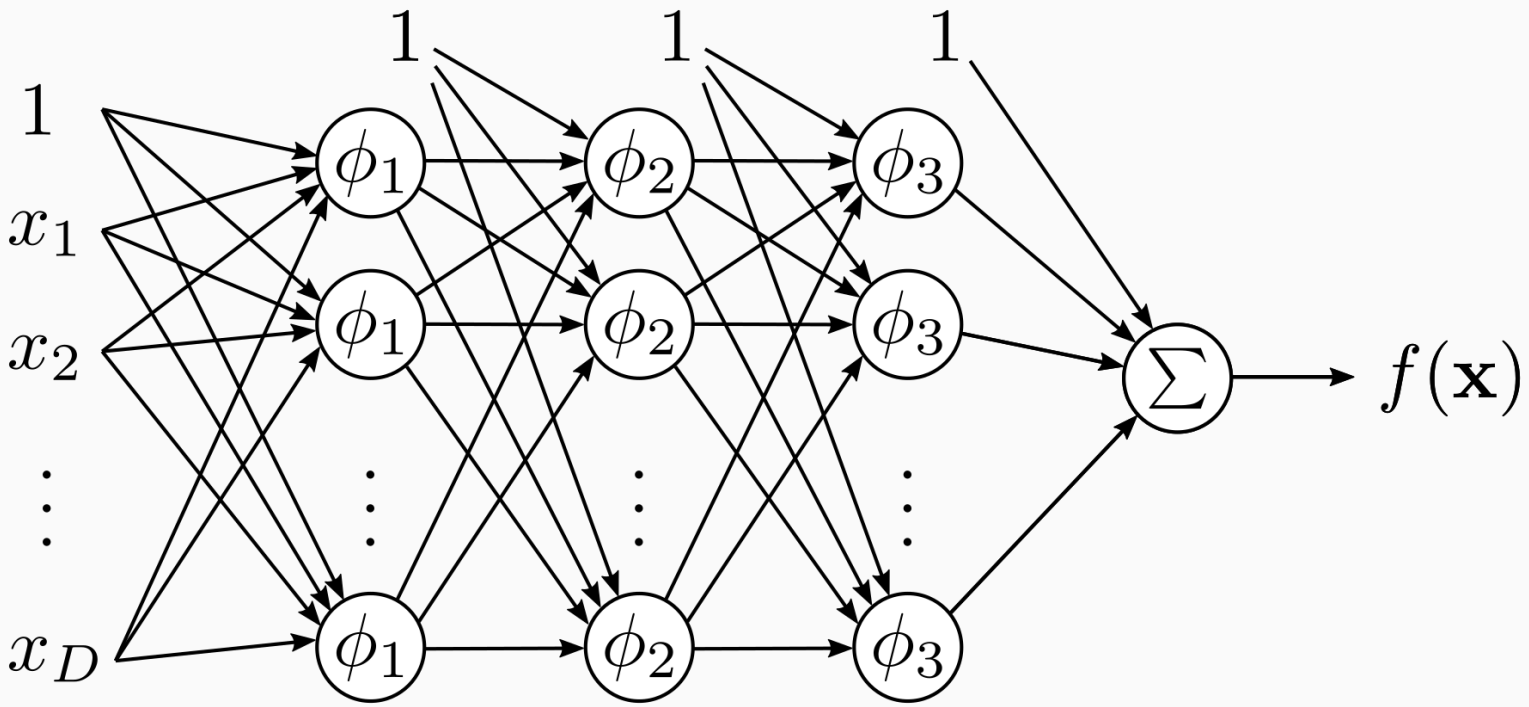
2-layer MLP



Multi Layer Perceptrons

- Stacking layers yields **multi-layer feed-forward neural networks**, often called **multi layer perceptrons** (MLPs)
- Hidden layers can contain different numbers of hidden units H_1, H_2, ...

3-layer MLP



But why do we need multiple layers?

A **single layer** already gives us universal approximation, right?

Expressive Efficiency

- **Shallow networks** (1-layer)
  - **universal approximators**
  - However, some function classes require **exponentially** many neurons, in the number of inputs
- **Deep networks**
  - can often represent represent the same functions with only **polynomial size**
  - This phenomenon is called **expressive efficiency** – Deep networks can represent non-linear function more efficiently than shallow ones
- Classical example: **parity function** – requires exponential size in shallow networks, but polynomial size in deep networks

Expressive Efficiency – Intuition

- Non-linearities ϕ are "simple", like sigmoid, ReLU, etc.
- Real-world problems like "predict apple/pear from an image" are "highly non-linear"
- Cascading non-linear layers improves their flexibility to express non-linear functions
- Lower layers tend to extract low-level features, while higher levels tend to learn more abstract features

Deep Vs. Shallow Networks

- **Traditional downside** of deep networks: harder to train (to be discussed later)
- **"Neural network winter" ($\sim$1990-2005):** "Neural networks with more than two layers cannot be trained."
- **Deep learning (2006–):** Tricks of the trade to train deep networks
- Neural networks can nowadays be scaled to hundreds or even thousands of layers

Backpropagation

Neural networks are powerful function approximators, with thousands, million, billions of parameters...

How do we learn these parameters?

Neural networks are powerful function approximators, with thousands, million, billions of parameters...

How do we learn these parameters?

Gradient descent!

Training Setup

- We collect all the parameters in one vector $\boldsymbol{\theta}$
- Let $\hat{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$ be the network's output for input $\boldsymbol{x}$ and parameters $\boldsymbol{\theta}$
- Training loss: $\mathcal{L}_{train}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \ell(y^{(i)}, f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}))$
- ℓ is some sample-wise loss, e.g. quadratic loss for regression, or cross-entropy for classification
- Minimize with gradient descent with $\nabla_{\boldsymbol{\theta}} \mathcal{L}_{train}$
- But how to compute the gradient? 

Training loss:

$$\mathcal{L}_{train}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} \ell(y_i, f(\boldsymbol{x}_i; \boldsymbol{\theta}))$$

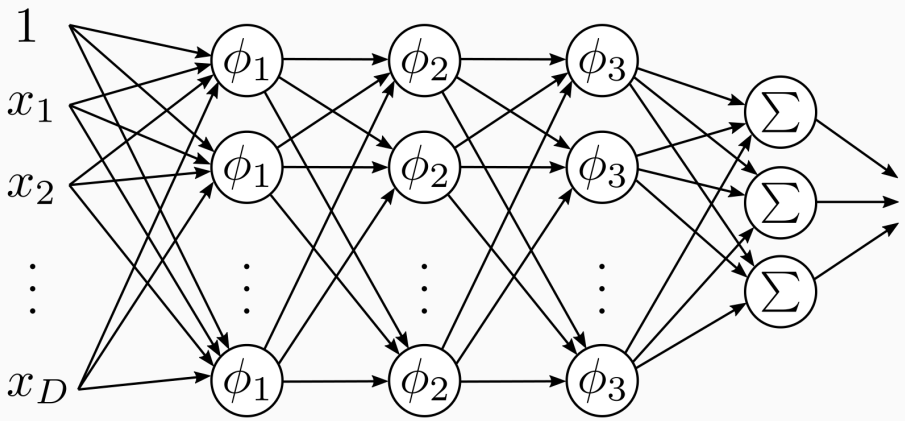
First simplification: gradient of sum is sum of gradients:

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}_{train}(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \frac{1}{N}\sum_{i=1}^{N} \ell(y_i, f(\boldsymbol{x}_i; \boldsymbol{\theta})) = \frac{1}{N}\sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} \ell(y_i, f(\boldsymbol{x}_i; \boldsymbol{\theta}))$$

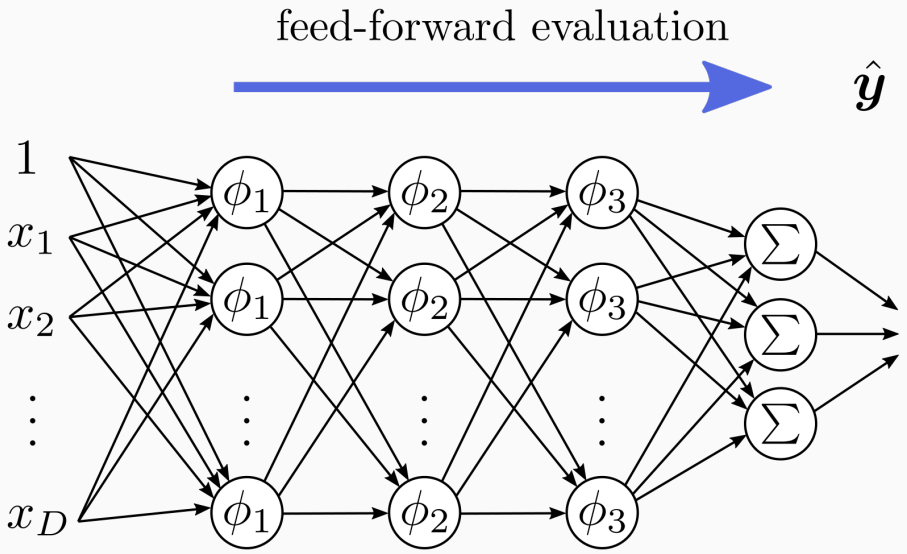
Thus, we can compute the gradients independently and parallel for each sample.

Task: Compute gradient for single sample

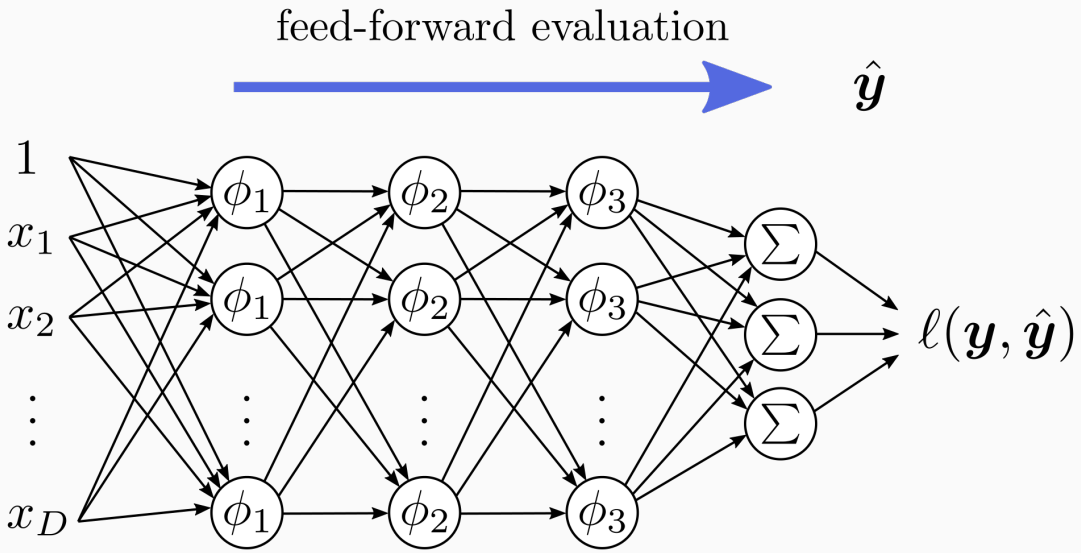
Backpropagation of Error (Backprop)



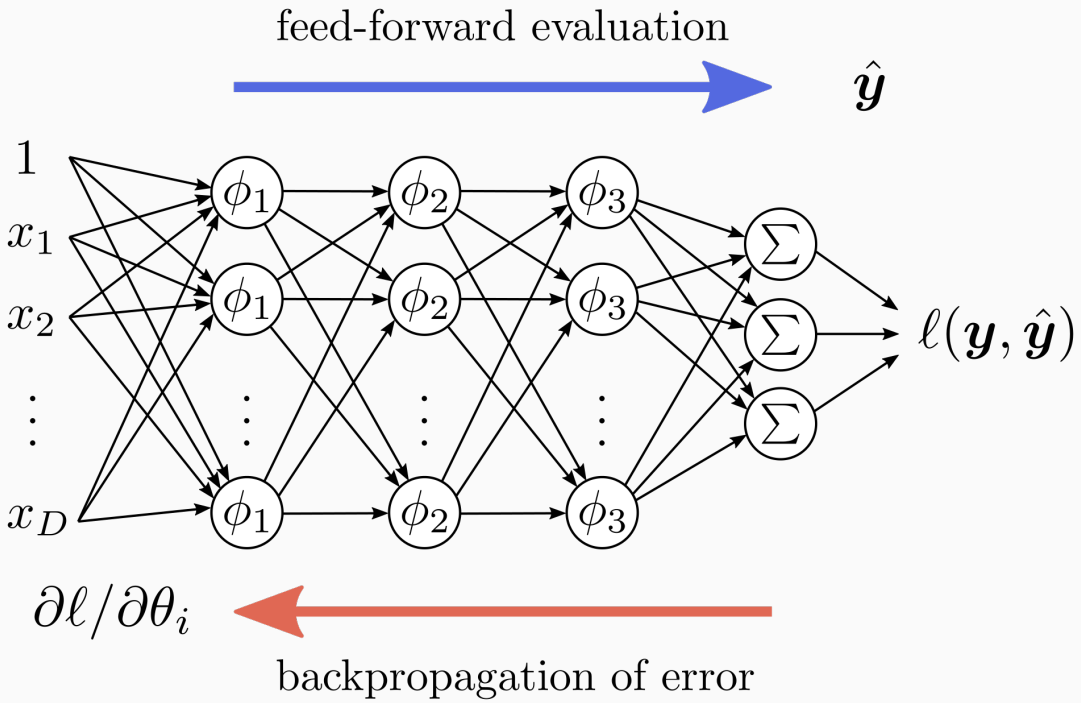
Backpropagation of Error (Backprop)



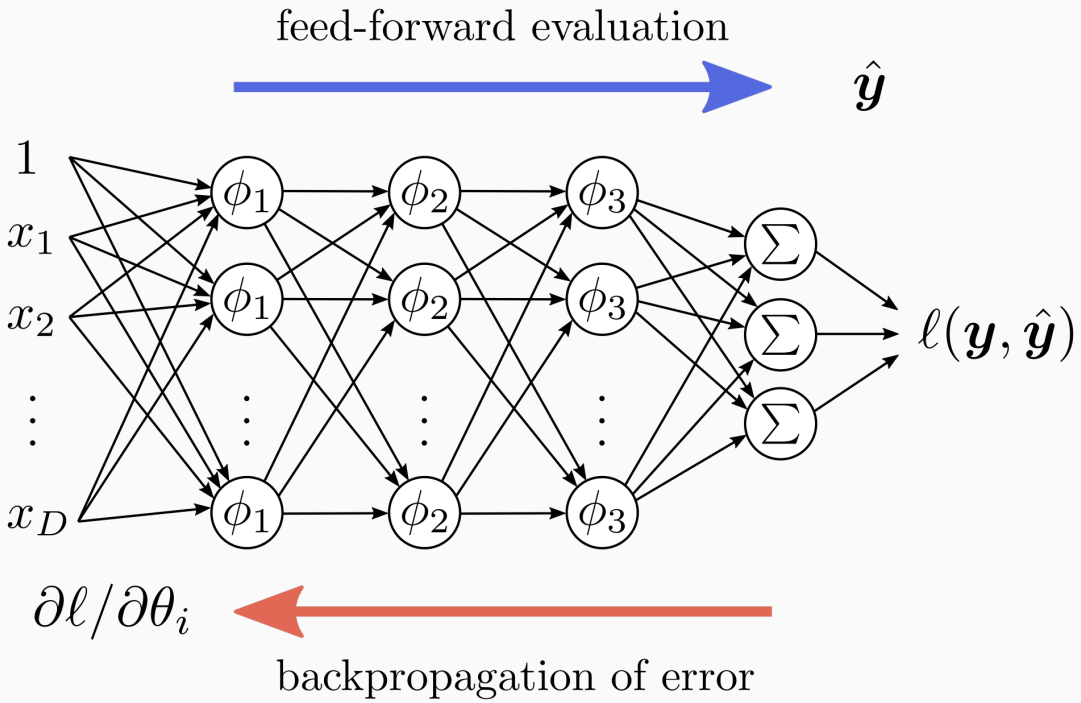
Backpropagation of Error (Backprop)



Backpropagation of Error (Backprop)



Backpropagation of Error (Backprop)



- Like in a company, where low-rank employees produce results for their respective managers
- Top managers present results to stakeholders, and get feedback about errors
- The managers' errors depend on their respective subordinates – so they communicate their contribution to the errors

Backprop in a Nutshell

Really, it's just the chain rule of calculus + some bookkeeping.

Recall Lecture 2: Chain rule in higher dimensions via **Jacobian**.

Backprop in a Nutshell

Really, it's just the chain rule of calculus + some bookkeeping.

Recall Lecture 2: Chain rule in higher dimensions via **Jacobian**.

$$f(x) = f(g(x), h(x))$$

Backprop in a Nutshell

Really, it's just the chain rule of calculus + some bookkeeping.

Recall Lecture 2: Chain rule in higher dimensions via **Jacobian**.

$$f(x) = f(g(x), h(x))$$

$$\frac{\partial f}{\partial x} = \begin{pmatrix} \frac{\partial f}{\partial g} & \frac{\partial f}{\partial h} \end{pmatrix} \begin{pmatrix} \frac{\partial g}{\partial x} \\ \frac{\partial h}{\partial x} \end{pmatrix} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x} + \frac{\partial f}{\partial h}\frac{\partial h}{\partial x}$$

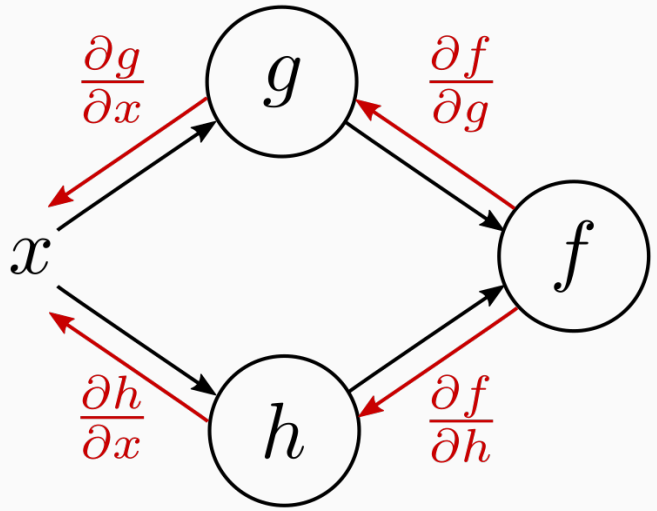
Backprop in a Nutshell

Really, it's just the chain rule of calculus + some bookkeeping.

Recall Lecture 2: Chain rule in higher dimensions via **Jacobian**.

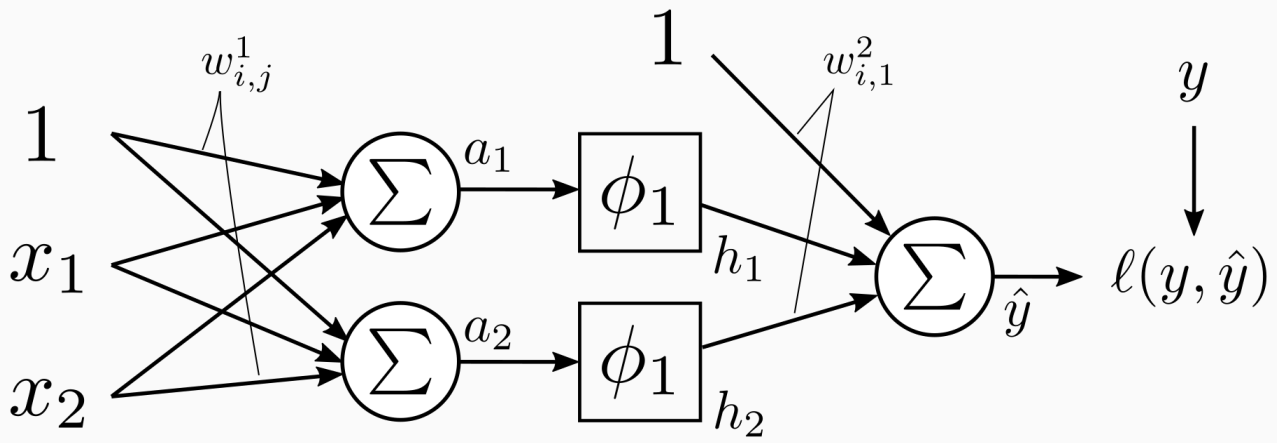
$$f(x) = f(g(x), h(x))$$

$$\frac{\partial f}{\partial x} = \begin{pmatrix} \frac{\partial f}{\partial g} & \frac{\partial f}{\partial h} \end{pmatrix} \begin{pmatrix} \frac{\partial g}{\partial x} \\ \frac{\partial h}{\partial x} \end{pmatrix} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x} + \frac{\partial f}{\partial h}\frac{\partial h}{\partial x}$$



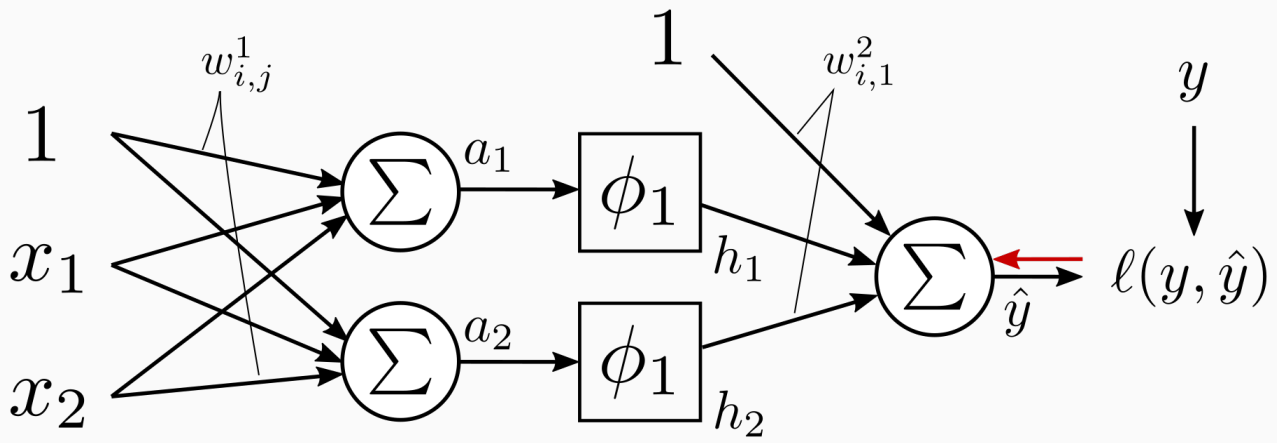
Backprop in Detail (small network)

Example



Backprop in Detail (small network)

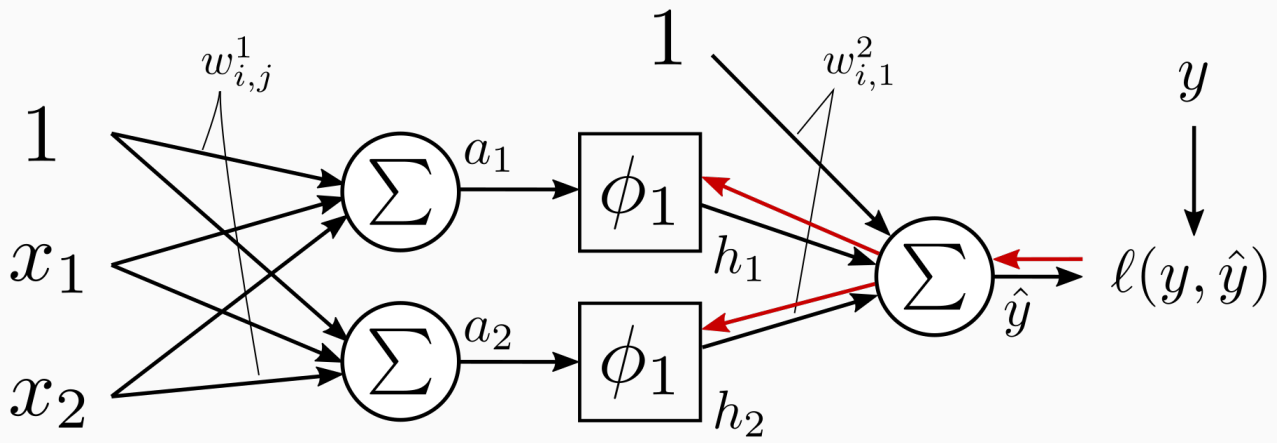
Example



- $\frac{\partial \ell}{\partial \hat{y}} =: B_1$ (closed form for most losses)

Backprop in Detail (small network)

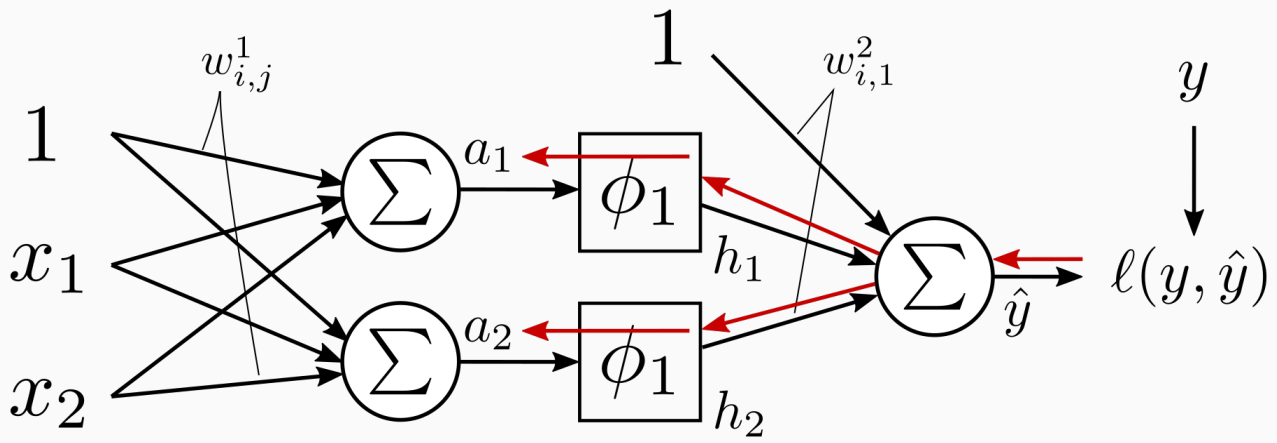
Example



- $\frac{\partial \ell}{\partial \hat{y}} =: B_1$ (closed form for most losses)
- $\frac{\partial \ell}{\partial h_i} = \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h_i} = B_1 w^2_{i,1} =: B_{2,i}$ $(\hat{y} = \sum_{i=0}^{2} w^2_{i,1} h_i)$

Backprop in Detail (small network)

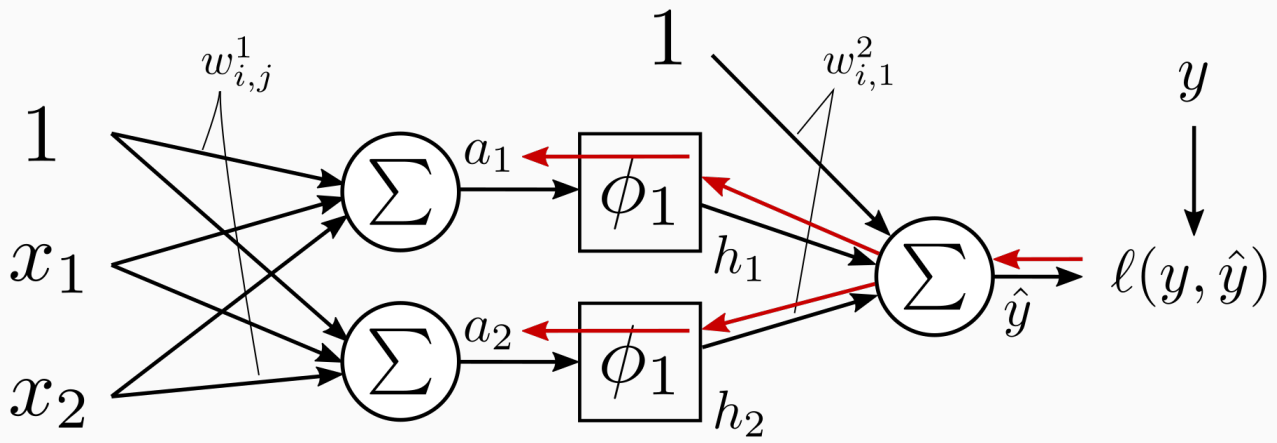
Example



- $\frac{\partial \ell}{\partial \hat{y}} =: B_1$ (closed form for most losses)
- $\frac{\partial \ell}{\partial h_i} = \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h_i} = B_1 w^2_{i,1} =: B_{2,i}$ $(\hat{y} = \sum_{i=0}^{2} w^2_{i,1} h_i)$
- $\frac{\partial \ell}{\partial a_i} = \frac{\partial \ell}{\partial h_i} \frac{\partial h_i}{\partial a_i} = B_{2,i}\, \phi_1'(a_i) =: B_{3,i}$ (closed form ϕ_1')

Backprop in Detail (small network)

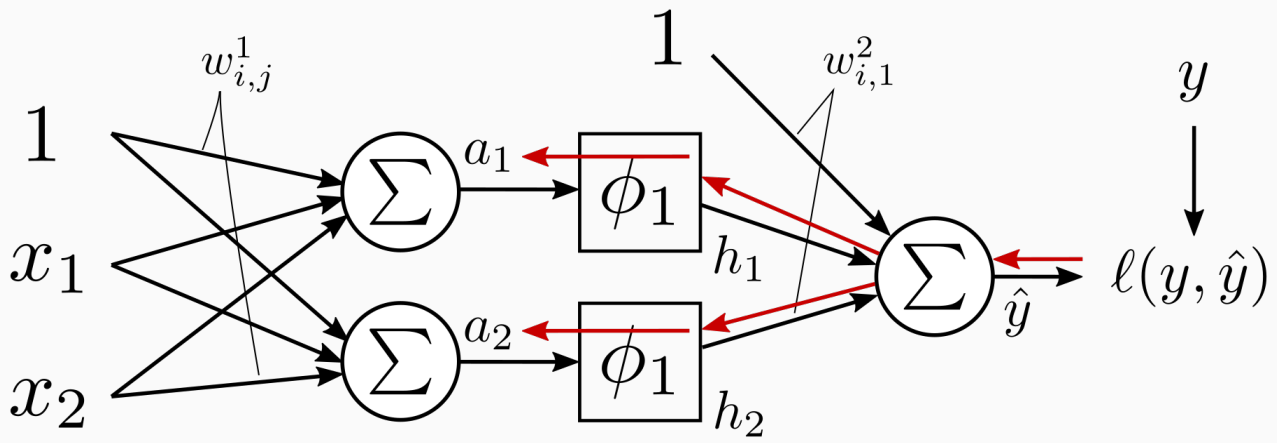
Example



- $\frac{\partial \ell}{\partial \hat{y}} =: B_1$ (closed form for most losses)
- $\frac{\partial \ell}{\partial h_i} = \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h_i} = B_1 w^2_{i,1} =: B_{2,i}$ $(\hat{y} = \sum_{i=0}^{2} w^2_{i,1} h_i)$
- $\frac{\partial \ell}{\partial a_i} = \frac{\partial \ell}{\partial h_i} \frac{\partial h_i}{\partial a_i} = B_{2,i}\, \phi_1'(a_i) =: B_{3,i}$ (closed form ϕ_1')
- $\frac{\partial \ell}{\partial w^2_{i,1}} = \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w^2_{i,1}} = B_1 h_i$ $(\hat{y} = \sum_{i=0}^{2} w^2_{i,1} h_i)$

Backprop in Detail (small network)

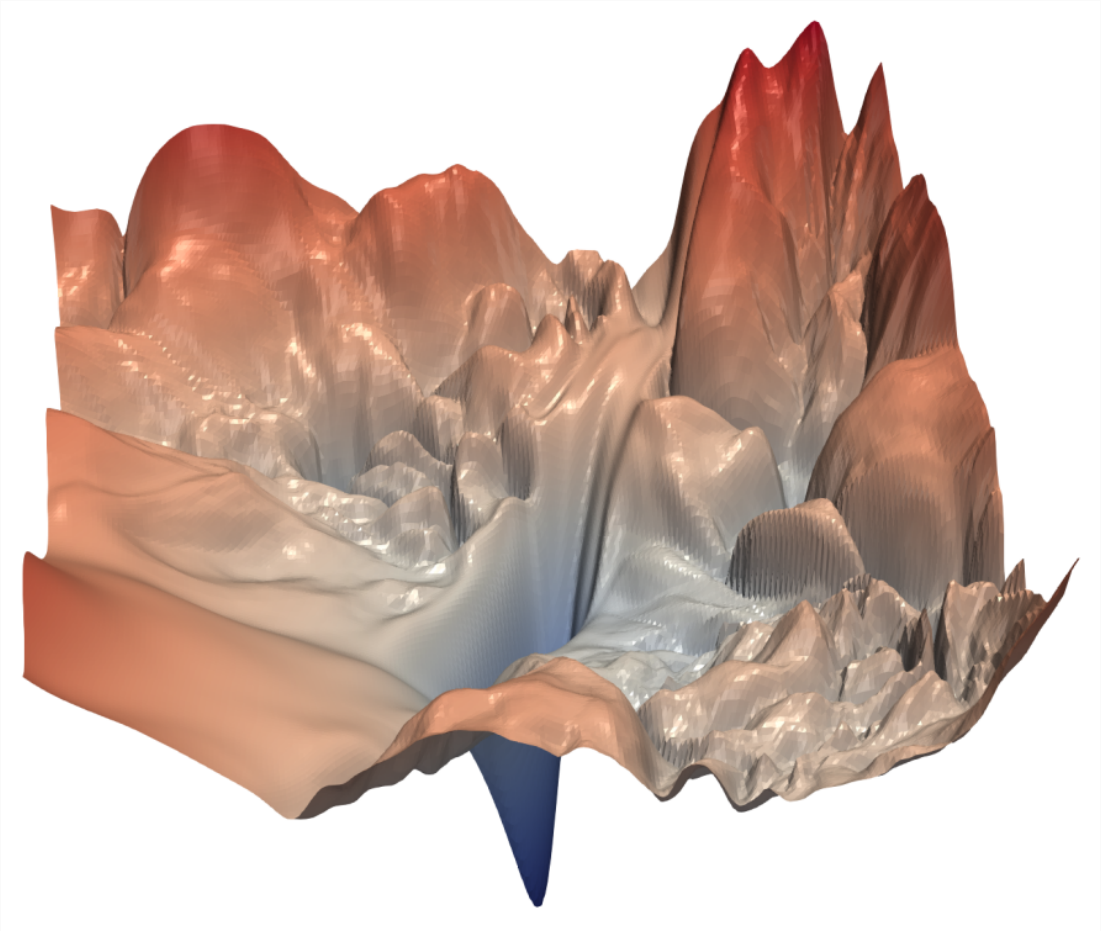
Example



- $\frac{\partial \ell}{\partial \hat{y}} =: B_1$ (closed form for most losses)
- $\frac{\partial \ell}{\partial h_i} = \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h_i} = B_1 w^2_{i,1} =: B_{2,i}$ $(\hat{y} = \sum_{i=0}^{2} w^2_{i,1} h_i)$
- $\frac{\partial \ell}{\partial a_i} = \frac{\partial \ell}{\partial h_i} \frac{\partial h_i}{\partial a_i} = B_{2,i}\, \phi_1'(a_i) =: B_{3,i}$ (closed form ϕ_1')
- $\frac{\partial \ell}{\partial w^2_{i,1}} = \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w^2_{i,1}} = B_1 h_i$ $(\hat{y} = \sum_{i=0}^{2} w^2_{i,1} h_i)$
- $\frac{\partial \ell}{\partial w^1_{i,j}} = \frac{\partial \ell}{\partial a_j} \frac{\partial a_j}{\partial w^1_{i,j}} = B_{3,j} x_i$ $(a_j = \sum_{i=0}^{2} w^1_{i,j} x_i)$

Loss Landscape of Neural Networks (Non-Convex)

Training loss in neural networks is highly non-convex, and we can find only local minima. However, in practice they work very well.



Source: Li et al., *Visualizing the Loss Landscape of Neural Nets*, NeurIPS 2018.

Summary

- ANN: network of artificial neurons
- Non-linearity is key, otherwise ANNs "collapse" to a linear function
- Universal function approximators, even for one hidden layer
- Deep networks are more expressive efficient
- Learning via gradient descent
- Gradient via backpropagation, aka the chain rule