Department of Mathematics
ETH Zurich

CAS ETH Machine Learning in Finance and Insurance

BLOCK I. Introduction to Machine Learning. Lecture 3.

Dr. A. Ferrario, ETH Zurich and UZH

Last week we…

1	Discussed linear regression
2	Introduced a well-known methodology to search for minima of functions: gradient descent and its stochastic version
3	Explored gradient descent in Google Colab

We will use slides to introduce our topics and the blackboard to deep-dive into selected items

Machine Learning Methods (Part 3)

- Logistic regression
- A brief history of artificial neural networks and deep learning
- Feedforward (artificial) neural networks

Logistic regression

Logistic regression: ideas

I

Logistic regression is specifically designed for binary classification problems, where the goal is to categorize data into two distinct groups (e.g., spam and fraud detection)

II

It provides probabilities for class membership, offering a measure of certainty about the classification, which is valuable in decision-making processes where understanding the likelihood of outcomes is crucial

III

It models the relationship between the features and the probability of a particular outcome, offering interpretable insights into how different predictors affect the odds of the target variable

Logistic regression
Notation

Model

A random variable Y is *Bernoulli distributed with parameter* $p \in [0,1]$ $(\text{Ber}(p))$ if

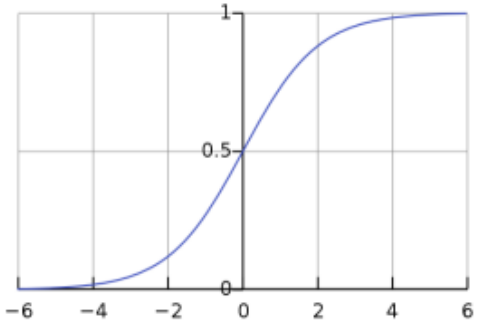
$$\begin{cases} \mathbb{P}[Y=1] = p \\ \mathbb{P}[Y=0] = 1-p \end{cases}$$

We are interested in an outcome rv with the Bernoulli distribution. The r.bv. Has two possible outcomes.

Model Let $Y \mid X \sim \text{Ber}(p(X))$, where

- $X = (X_1, \dots, X_d)$,
- $p(X) = \psi(\theta_0 + \theta_1 X_1 + \cdots + \theta_d X_d)$
- $\psi(x) = \dfrac{e^x}{e^x+1} = \dfrac{1}{1+e^{-x}}$

  (logistic function)



$$\begin{cases} \mathbb{P}[Y=1 \mid X] = p(X) \\ \mathbb{P}[Y=0 \mid X] = 1-p(X) \end{cases}$$

The logistic regression modeling states that the outcome r.v. conditioned on d+1 r.v. follows a Bernoulli distribution. The parameter of this distribution depends on the d+1 r.v.

Logistic regression
Empirical loss minimization

Empirical setting

- *Training Data:* let $(x_{i1}, \ldots, x_{id}, y_i)$, $i = 1, \ldots, m$, be iid realizations of $(X_1, \ldots, X_d, Y)$
- Set $x_{i0} = 1$, $x_i^T\boldsymbol{\theta} = \sum_{j=0}^{d} x_i j \theta_j$, $p_i(\boldsymbol{\theta}) = \psi(x_i^T\boldsymbol{\theta})$ for $\boldsymbol{\theta} \in \mathbb{R}^{d+1}$ and $i = 1, \ldots, m$

We are interested in an outcome rv with Bernoulli distribution.

Empirical risk minimization

$$\min_{\boldsymbol{\theta}\in\mathbb{R}^{d+1}} E(\boldsymbol{\theta}) = \min_{\boldsymbol{\theta}\in\mathbb{R}^{d+1}} \frac{1}{m} \sum_{i=1}^{m} L(p_i(\boldsymbol{\theta}), y_i)$$

Cross-entropy $L(p_i(\boldsymbol{\theta}), y_i) := -y_i \log(p_i(\boldsymbol{\theta})) - (1 - y_i) \log(1 - p_i(\boldsymbol{\theta}))$

The logistic regression modeling states that the outcome r.v. conditioned on d+1 r.v. follows a Bernoulli distribution. The parameter of this distribution depends on the d+1 r.v.

Empirical risk minimization - results

$$\min_{\boldsymbol{\theta}\in\mathbb{R}^{d+1}} E(\boldsymbol{\theta}) = \min_{\boldsymbol{\theta}\in\mathbb{R}^{d+1}} \sum_{i=1}^{m} \left(\log(1 + e^{x_i^T\boldsymbol{\theta}}) - y_i x_i^T\boldsymbol{\theta} \right)$$

Can be solved with SGD

Logistic regression

Empirical loss minimization in **`sklearn`** is regularized/penalized **by default**

`sklearn.linear_model`.LogisticRegression

$$\min_{\boldsymbol{\theta}\in\mathbb{R}^{d+1}} E(\boldsymbol{\theta}) = \min_{\boldsymbol{\theta}\in\mathbb{R}^{d+1}} C \sum_{i=1}^{m} \left(\log(1+e^{x_i^T\boldsymbol{\theta}}) - y_i x_i^T\boldsymbol{\theta}\right) + r(\boldsymbol{\theta})$$

Used to control the strength of the regularization (default C=1)

L1

$$r(\boldsymbol{\theta}) = \sum_{i=1}^{d} |\theta_i|$$

L2

$$r(\boldsymbol{\theta}) = \frac{1}{2}\sum_{i=1}^{d} \theta_i^2$$

default

Additional documentation

https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression

Logistic regression
Predicting probabilities and classes from new data points

Predictions

Solution of the empirical loss minimization problem: $\hat{\boldsymbol{\theta}} = \begin{pmatrix} \hat{\theta}_0 \\ \vdots \\ \hat{\theta}_d \end{pmatrix}$

New data point: $x = (x_1, \ldots, x_d)$

1. Predicted probability that $Y = 1$ conditioned on $X = (x_1, \ldots, x_d)$: $\hat{p}(x) = \psi\left(\hat{\theta}_0 + \sum_{i=j}^{d} \hat{\theta}_j x_j\right)$
2. Binary Classification
   - Choose a decision threshold $c \in (0, 1)$
   - Predict Y to be $\hat{y}(x) = \begin{cases} 1 & \text{if } \hat{p}(x) \geq c \\ 0 & \text{if } \hat{p}(x) < c \end{cases}$

If we want a binary classification of an input, we need to introduce a decision "threshold" *c*

The learned/trained logistic regression model allows us computing the empirical probabilities AND a class/label, at the cost of introducing an arbitrary decision threshold

Measuring the performance of logistic regression models

Download from Moodle a short guide to the performance measures in (binary) classification models

Useful for Project I – "Credit Analytics"

2024_CAS_ETH_ML_Fin_Insurance_Perf_Measures_Classification.pdf

Logistic regression: Key takeaways

I

Ideal for Binary Classification: Logistic regression excels in binary classification tasks, such as email spam detection or disease diagnosis, by categorizing data into two distinct groups

II

Probabilistic Outputs: Provides probability estimates for outcomes, offering insights into the likelihood of class membership, crucial for risk assessment and decision-making.

III

Feature Influence Analysis: Enables understanding of how different predictors affect the probability of the target class, useful for interpretative analysis in fields like healthcare and finance

IV

Foundation for Advanced Techniques: Serves as a stepping stone to more complex algorithms in machine learning, such as neural networks, while teaching important concepts like the sigmoid function

Self-Study: Closing with logistic regression

Self-Study

From Moodle download the book:

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An introduction to statistical learning* (Vol. 112, p. 18). New York: Springer.

- **Section 4.4.3** Logistic Regression

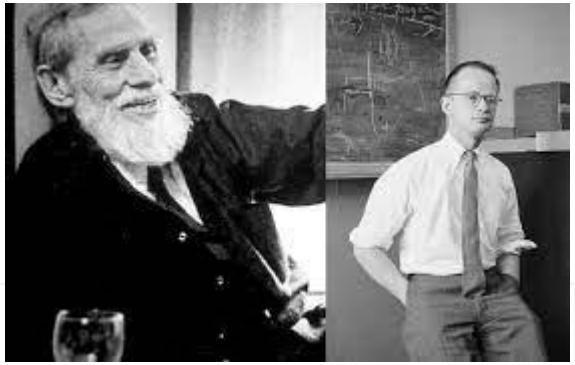
Artificial neural networks: A bit of history

- 1940s-1960s: Cybernetics
  - first mathematical models of biological neurons [McCulloch and Pitts, 1943, Hebb, 2005]
  - perceptron model [Rosenblatt, 1958]
- 1980s-1990s: Connectionism
  - back-propagation algorithm [Rumelhart et al., 1986]
  - long short-term memory (LSTM) architecture [Hochreiter and Schmidhuber,
- 2006-now: Deep learning revolution
  - efficient training of deep belief networks [Hinton et al., 2006]
  - efficient training of deep neural networks [Bengio et al., 2006, Ranzato et al.,

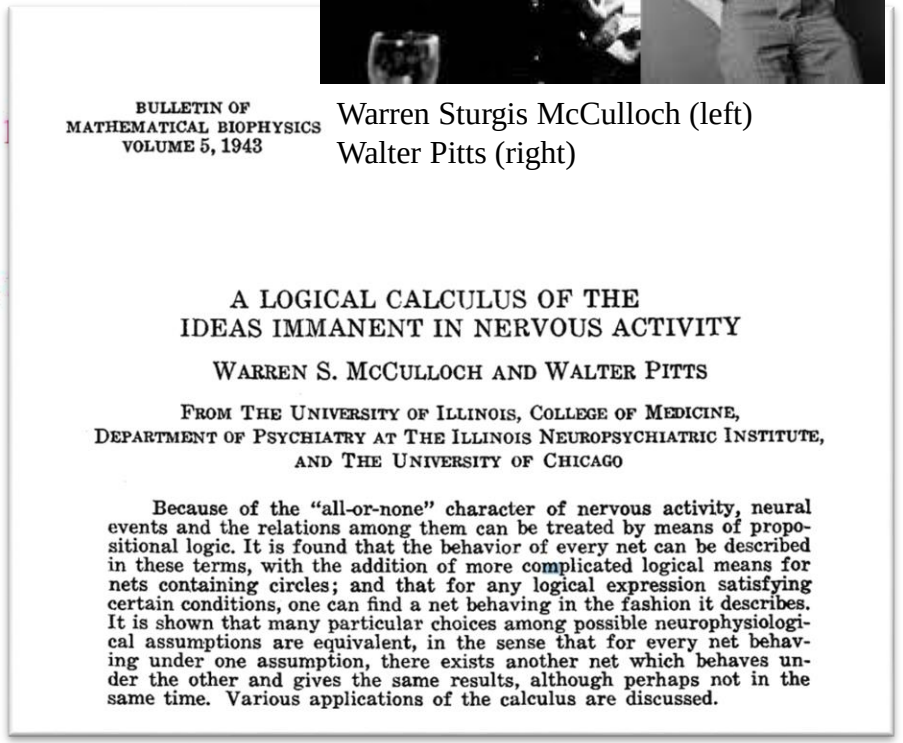
A few remarks:

- Neural networks have been known since the 1980s!
- Today's deep learning revolution is mainly due to:
  - increased ***computational power*** $\Longrightarrow$ we can train ***larger*** networks,
  - increased ***data availability*** $\Longrightarrow$ the networks can ***learn well***.

Source: https://donaldclarkplanb.blogspot.com/2021/11/mcculloch-pitts-neural-nets.html



Warren Sturgis McCulloch (left)
Walter Pitts (right)



BULLETIN OF
MATHEMATICAL BIOPHYSICS
VOLUME 5, 1943

A LOGICAL CALCULUS OF THE
IDEAS IMMANENT IN NERVOUS ACTIVITY

WARREN S. MCCULLOCH AND WALTER PITTS

FROM THE UNIVERSITY OF ILLINOIS, COLLEGE OF MEDICINE,
DEPARTMENT OF PSYCHIATRY AT THE ILLINOIS NEUROPSYCHIATRIC INSTITUTE,
AND THE UNIVERSITY OF CHICAGO

Because of the "all-or-none" character of nervous activity, neural events and the relations among them can be treated by means of propositional logic. It is found that the behavior of every net can be described in these terms, with the addition of more complicated logical means for nets containing circles; and that for any logical expression satisfying certain conditions, one can find a net behaving in the fashion it describes. It is shown that many particular choices among possible neurophysiological assumptions are equivalent, in the sense that for every net behaving under one assumption, there exists another net which behaves under the other and gives the same results, although perhaps not in the same time. Various applications of the calculus are discussed.

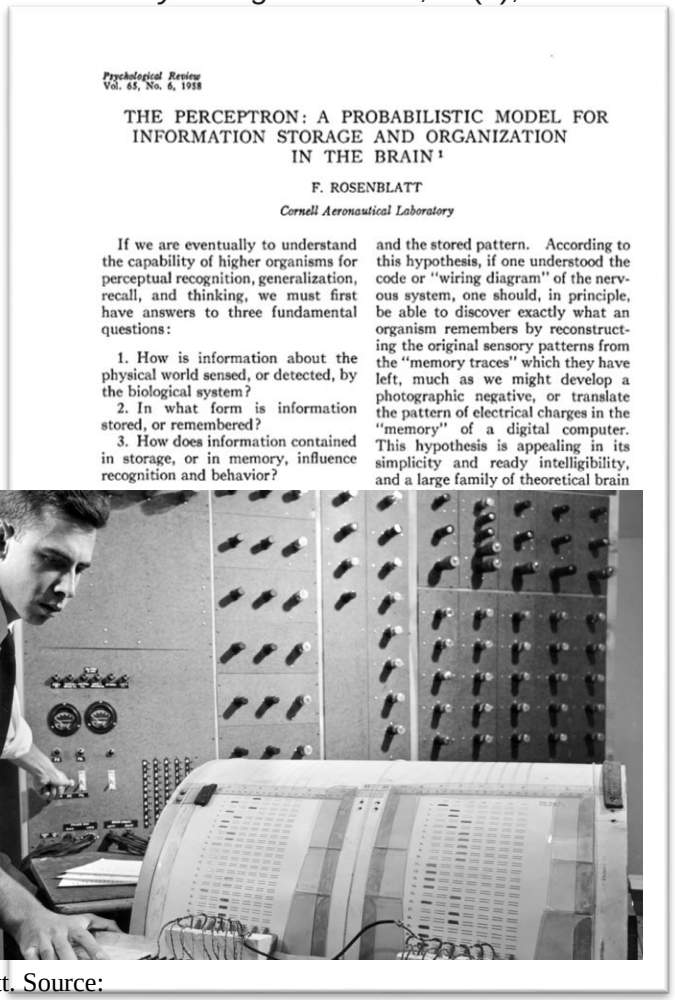
McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, *5*, 115-133.

- 1940s-1960s: Cybernetics
  - first mathematical models of biological neurons [McCulloch and Pitts, 1943, Hebb, 2005]
  - perceptron model [Rosenblatt, 1958]
- 1980s-1990s: Connectionism
  - back-propagation algorithm [Rumelhart et al., 1986]
  - long short-term memory (LSTM) architecture [Hochreiter and Schmidhuber, 1997]
- 2006-now: Deep learning revolution
  - efficient training of deep belief networks [Hinton et al., 2006]
  - efficient training of deep neural networks [Bengio et al., 2006, Ranzato et al., 2

A few remarks:

- Neural networks have been known since the 1980s!
- Today's deep learning revolution is mainly due to:
  - increased ***computational power*** $\Longrightarrow$ we can train ***larger*** networks,
  - increased ***data availability*** $\Longrightarrow$ the networks can ***learn well***.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, *65*(6), 386.



Psychological Review
Vol. 65, No. 6, 1958

THE PERCEPTRON: A PROBABILISTIC MODEL FOR INFORMATION STORAGE AND ORGANIZATION IN THE BRAIN[1]

F. ROSENBLATT

Cornell Aeronautical Laboratory

If we are eventually to understand the capability of higher organisms for perceptual recognition, generalization, recall, and thinking, we must first have answers to three fundamental questions:

1. How is information about the physical world sensed, or detected, by the biological system?
2. In what form is information stored, or remembered?
3. How does information contained in storage, or in memory, influence recognition and behavior?

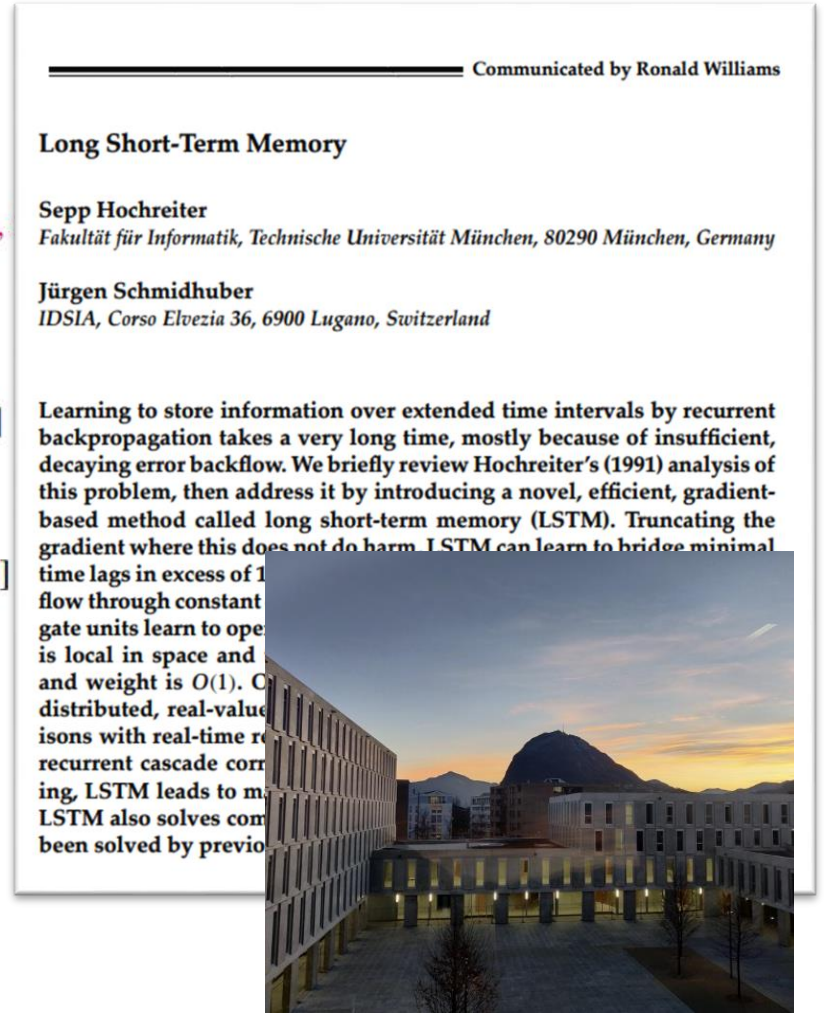
and the stored pattern. According to this hypothesis, if one understood the code or "wiring diagram" of the nervous system, one should, in principle, be able to discover exactly what an organism remembers by reconstructing the original sensory patterns from the "memory traces" which they have left, much as we might develop a photographic negative, or translate the pattern of electrical charges in the "memory" of a digital computer. This hypothesis is appealing in its simplicity and ready intelligibility, and a large family of theoretical brain

Frank Rosenblatt. Source: https://news.cornell.edu/stories/2019/09/professors-perceptron-paved-way-ai-60-years-too-soon

- 1940s-1960s: Cybernetics
  - first mathematical models of biological neurons [McCulloch and Pitts, 1943, Hebb,
  - perceptron model [Rosenblatt, 1958]
- 1980s-1990s: Connectionism
  - back-propagation algorithm [Rumelhart et al., 1986]
  - long short-term memory (LSTM) architecture [Hochreiter and Schmidhuber, 1997]
- 2006-now: Deep learning revolution
  - efficient training of deep belief networks [Hinton et al., 2006]
  - efficient training of deep neural networks [Bengio et al., 2006, Ranzato et al., 2006]

A few remarks:

- Neural networks have been known since the 1980s!
- Today's deep learning revolution is mainly due to:
  - increased ***computational power*** $\Longrightarrow$ we can train ***larger*** networks,
  - increased ***data availability*** $\Longrightarrow$ the networks can ***learn well***.



Communicated by Ronald Williams

Long Short-Term Memory

Sepp Hochreiter
Fakultät für Informatik, Technische Universität München, 80290 München, Germany

Jürgen Schmidhuber
IDSIA, Corso Elvezia 36, 6900 Lugano, Switzerland

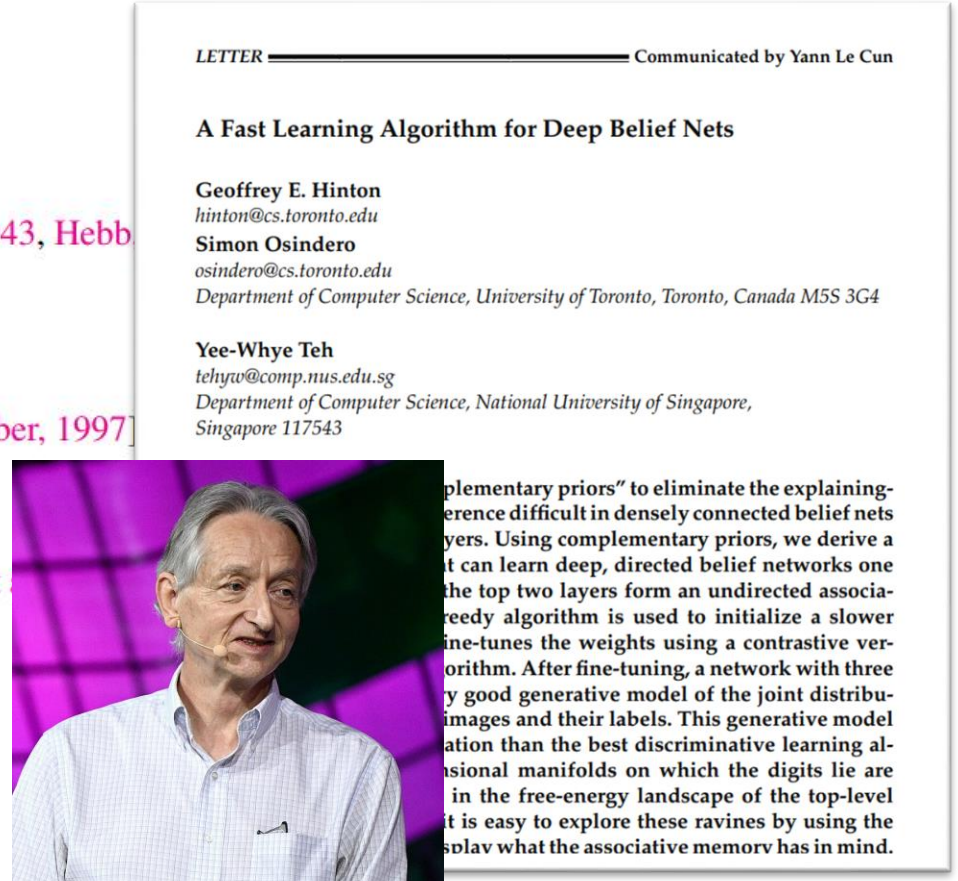
Learning to store information over extended time intervals by recurrent backpropagation takes a very long time, mostly because of insufficient, decaying error backflow. We briefly review Hochreiter's (1991) analysis of this problem, then address it by introducing a novel, efficient, gradient-based method called long short-term memory (LSTM). Truncating the gradient where this does not do harm, LSTM can learn to bridge minimal time lags in excess of 1 flow through constant gate units learn to ope is local in space and and weight is $O(1)$. C distributed, real-valu isons with real-time r recurrent cascade cor ing, LSTM leads to m LSTM also solves com been solved by previo

IDSIA, Lugano

- 1940s-1960s: Cybernetics
  - first mathematical models of biological neurons [McCulloch and Pitts, 1943, Hebb
  - perceptron model [Rosenblatt, 1958]
- 1980s-1990s: Connectionism
  - back-propagation algorithm [Rumelhart et al., 1986]
  - long short-term memory (LSTM) architecture [Hochreiter and Schmidhuber, 1997]
- 2006-now: Deep learning revolution
  - efficient training of deep belief networks [Hinton et al., 2006]
  - efficient training of deep neural networks [Bengio et al., 2006, Ranzato et

A few remarks:

- Neural networks have been known since the 1980s!
- Today's deep learning revolution is mainly due to:
  - increased ***computational power*** $\Longrightarrow$ we can train ***larger*** networks,
  - increased ***data availability*** $\Longrightarrow$ the networks can ***learn well***.



LETTER Communicated by Yann Le Cun

A Fast Learning Algorithm for Deep Belief Nets

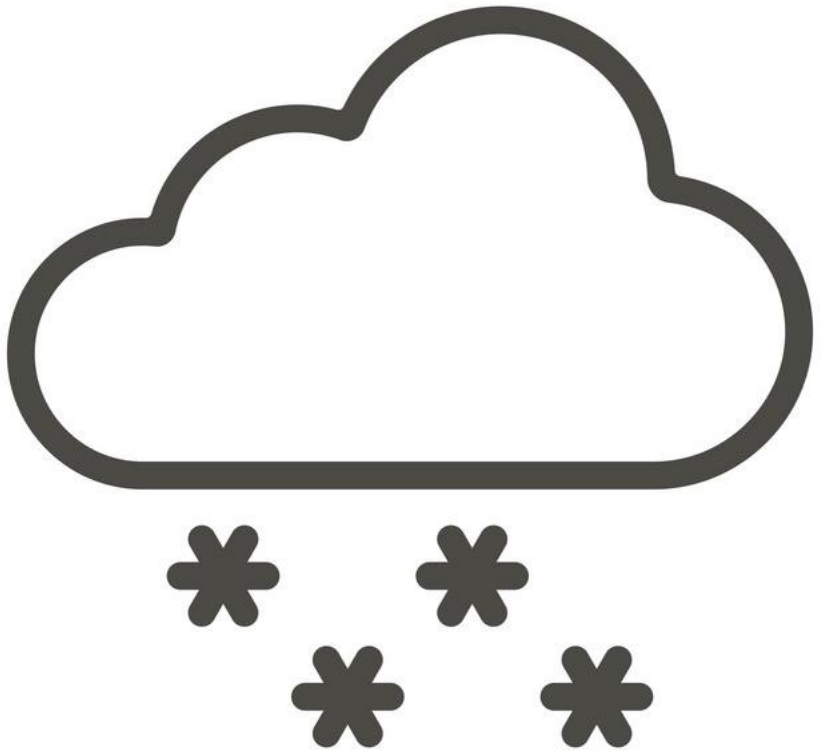
Geoffrey E. Hinton
hinton@cs.toronto.edu
Simon Osindero
osindero@cs.toronto.edu
Department of Computer Science, University of Toronto, Toronto, Canada M5S 3G4

Yee-Whye Teh
tehyw@comp.nus.edu.sg
Department of Computer Science, National University of Singapore, Singapore 117543

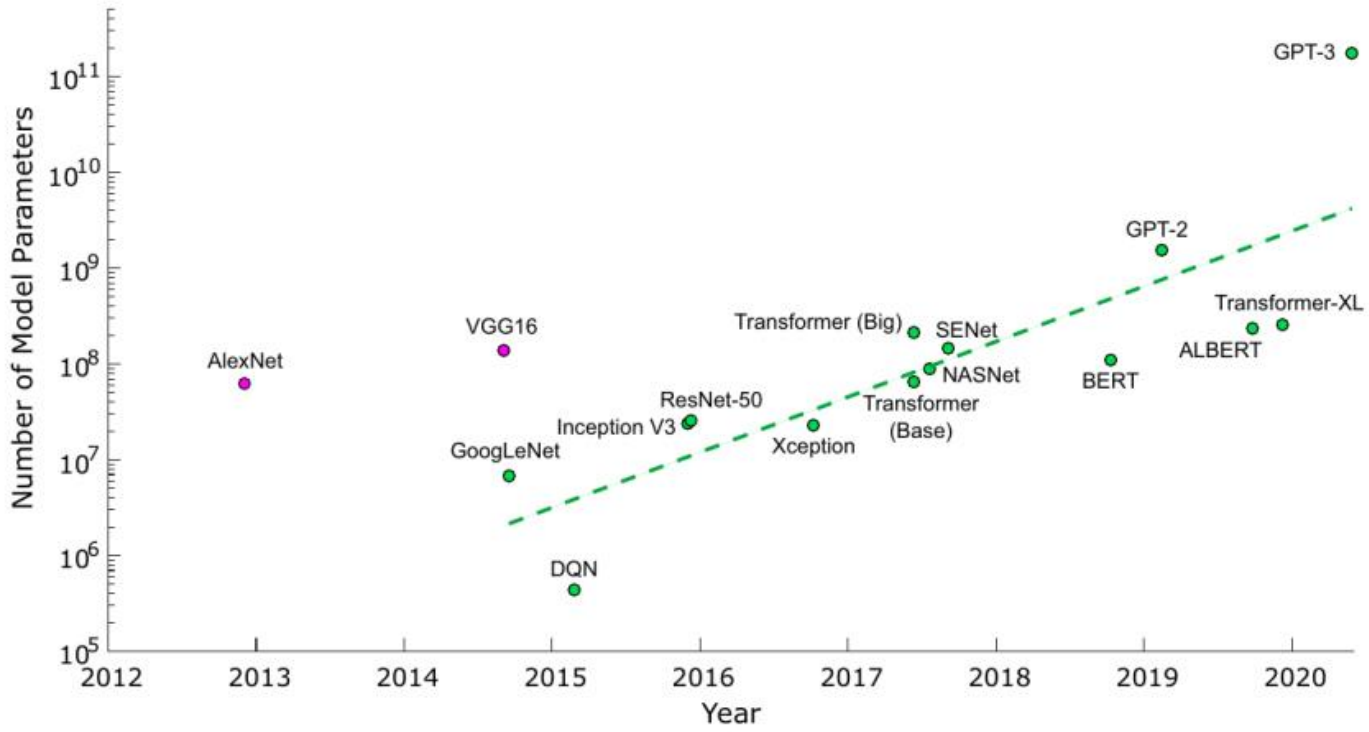
plementary priors" to eliminate the explaining-erence difficult in densely connected belief nets yers. Using complementary priors, we derive a t can learn deep, directed belief networks one he top two layers form an undirected associa-reedy algorithm is used to initialize a slower ine-tunes the weights using a contrastive ver-orithm. After fine-tuning, a network with three y good generative model of the joint distribu-images and their labels. This generative model ation than the best discriminative learning al-nsional manifolds on which the digits lie are in the free-energy landscape of the top-level t is easy to explore these ravines by using the splay what the associative memory has in mind.

Source: Wikipedia

Geoffrey Everest Hinton



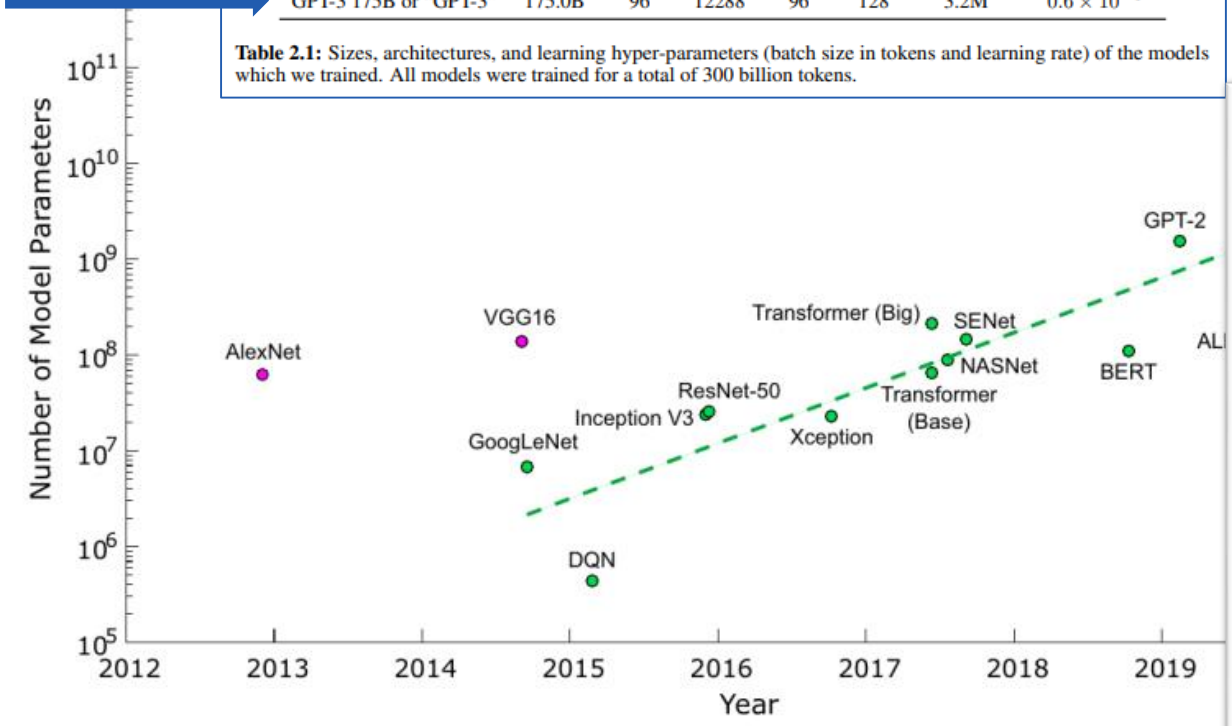
Main AI Winters:1974–1980 and 1987–1993



Neural networks are becoming larger (and especially ***deeper***).

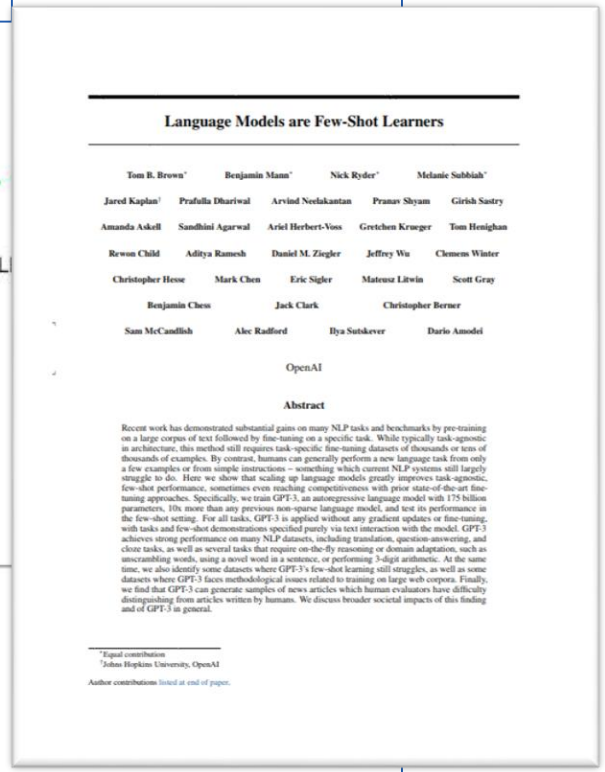
Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

Table 2.1: Sizes, architectures, and learning hyper-parameters (batch size in tokens and learning rate) of the models which we trained. All models were trained for a total of 300 billion tokens.



Neural networks are becoming larger (and especially ***deeper***).

GPT-3



Language Models are Few-Shot Learners

Tom B. Brown* Benjamin Mann* Nick Ryder* Melanie Subbiah*

Jared Kaplan† Prafulla Dhariwal Arvind Neelakantan Pranav Shyam Girish Sastry

Amanda Askell Sandhini Agarwal Ariel Herbert-Voss Gretchen Krueger Tom Henighan

Rewon Child Aditya Ramesh Daniel M. Ziegler Jeffrey Wu Clemens Winter

Christopher Hesse Mark Chen Eric Sigler Mateusz Litwin Scott Gray

Benjamin Chess Jack Clark Christopher Berner

Sam McCandlish Alec Radford Ilya Sutskever Dario Amodei

OpenAI

Abstract

Recent work has demonstrated substantial gains on many NLP tasks and benchmarks by pre-training on a large corpus of text followed by fine-tuning on a specific task. While typically task-agnostic in architecture, this method still requires task-specific fine-tuning datasets of thousands or tens of thousands of examples. By contrast, humans can generally perform a new language task from only a few examples or from simple instructions – something which current NLP systems still largely struggle to do. Here we show that scaling up language models greatly improves task-agnostic, few-shot performance, sometimes even reaching competitiveness with prior state-of-the-art fine-tuning approaches. Specifically, we train GPT-3, an autoregressive language model with 175 billion parameters, 10x more than any previous non-sparse language model, and test its performance in the few-shot setting. For all tasks, GPT-3 is applied without any gradient updates or fine-tuning, with tasks and few-shot demonstrations specified purely via text interaction with the model. GPT-3 achieves strong performance on many NLP datasets, including translation, question-answering, and cloze tasks, as well as several tasks that require on-the-fly reasoning or domain adaptation, such as unscrambling words, using a novel word in a sentence, or performing 3-digit arithmetic. At the same time, we also identify some datasets where GPT-3's few-shot learning still struggles, as well as some datasets where GPT-3 faces methodological issues related to training on large web corpora. Finally, we find that GPT-3 can generate samples of news articles which human evaluators have difficulty distinguishing from articles written by humans. We discuss broader societal impacts of this finding and of GPT-3 in general.

*Equal contribution
†Johns Hopkins University, OpenAI

Author contributions listed at end of paper.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., ... & Amodei, D. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, *33*, 1877-1901.

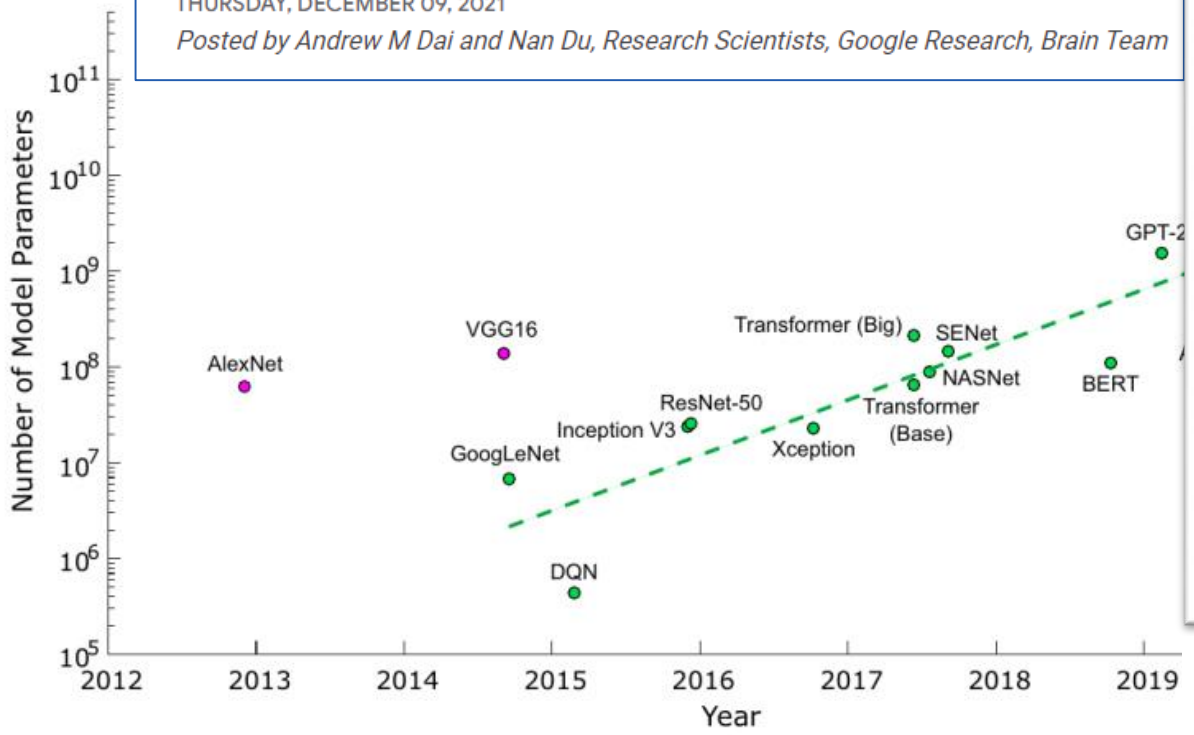
1.2T parameters

BLOG ›

More Efficient In-Context Learning with GLaM

THURSDAY, DECEMBER 09, 2021

Posted by Andrew M Dai and Nan Du, Research Scientists, Google Research, Brain Team



Neural networks are becoming larger (and especially ***deeper***).

GLaM: Efficient Scaling of Language Models with Mixture-of-Experts

Nan Du [*1] Yanping Huang [*1] Andrew M. Dai [*1] Simon Tong [1] Dmitry Lepikhin [1] Yuanzhong Xu [1] Maxim Krikun [1] Yanqi Zhou [1] Adams Wei Yu [1] Orhan Firat [1] Barret Zoph [1] Liam Fedus [1] Maarten Bosma [1] Zongwei Zhou [1] Tao Wang [1] Yu Emma Wang [1] Kellie Webster [1] Marie Pellat [1] Kevin Robinson [1] Kathleen Meier-Hellstern [1] Toju Duke [1] Lucas Dixon [1] Kun Zhang [1] Quoc V Le [1] Yonghui Wu [1] Zhifeng Chen [1] Claire Cui [1]

Abstract

Scaling language models with more data, compute and parameters has driven significant progress in natural language processing. For example, thanks to scaling, GPT-3 was able to achieve strong results on in-context learning tasks. However, training these large dense models requires significant amounts of computing resources. In this paper, we propose and develop a family of language models named GLaM (Generalist Language Model), which uses a sparsely activated mixture-of-experts architecture to scale the model capacity while also incurring substantially less training cost compared to dense variants. The largest GLaM has 1.2 trillion parameters, which is approximately 7x larger than GPT-3. It consumes only 1/3 of the energy used to train GPT-3 and requires half of the computation flops for inference, while still achieving better overall zero, one and few-shot performance across 29 NLP tasks.

Table 1. Comparison between GPT-3 and GLaM. In a nutshell, GLaM outperforms GPT-3 across 21 natural language understanding (NLU) benchmarks and 8 natural language generative (NLG) benchmarks in average while using about half the FLOPs per token during inference and consuming about one third the energy for training.

		GPT-3	GLaM	relative
cost	FLOPs / token (G)	350	**180**	−48.6%
	Train energy (MWh)	1287	**456**	−64.6%
accuracy on average	Zero-shot	56.9	**62.7**	+10.2%
	One-shot	61.6	**65.5**	+6.3%
	Few-shot	65.2	**68.1**	+4.4%

feasibility of in-context learning for few-shot or even zero-shot generalization, meaning very few labeled examples are needed to achieve good performance on NLP applications. While being effective and performant, scaling further is becoming prohibitively expensive and consumes significant amounts of energy (Patterson et al., 2021).

In this work, we show that a large sparsely activated network can achieve competitive results compared to state-of-the-art dense models on few-shot tasks while being more computationally efficient. We present a family of generalist language models called GLaM, that strike a balance between dense and conditional computation. The largest version of GLaM has 1.2T parameters in total with 64 experts per MoE layer (Shazeer et al., 2017; Lepikhin et al., 2021; Fedus et al., 2021) where each token in the input batch only activates a subnetwork of 96.6B (8% of 1.2T) parameters. On zero, one and few-shot learning, this model compares favorably to GPT-3 (175B), with significantly improved learning efficiency across 29 public NLP benchmarks, ranging from language completion tasks, open-domain QA tasks, to natural language inference tasks. Thanks to the sparsely activated architecture and the efficient implementation of the model parallelism algorithm, the total energy consumption during training is only one third of GPT-3's. We highlight the comparison between the largest version of GLaM and GPT-3 in Table 1 and Figure 1.

1. Introduction

Language models have played an important role in the progress of natural language processing (NLP) in the past decade. Variants of language models have been used to produce pretrained word vectors (Mikolov et al., 2013; Pennington et al., 2014), and contextualized word vectors (Peters et al., 2018; Devlin et al., 2019) for many NLP applications. The shift towards scaling with more data and larger models (Shazeer et al., 2017; Huang et al., 2019; Kaplan et al., 2020) has enabled complex natural language tasks to be performed with less labeled data. For example, GPT-3 (Brown et al., 2020) and FLAN (Wei et al., 2021) demonstrated the

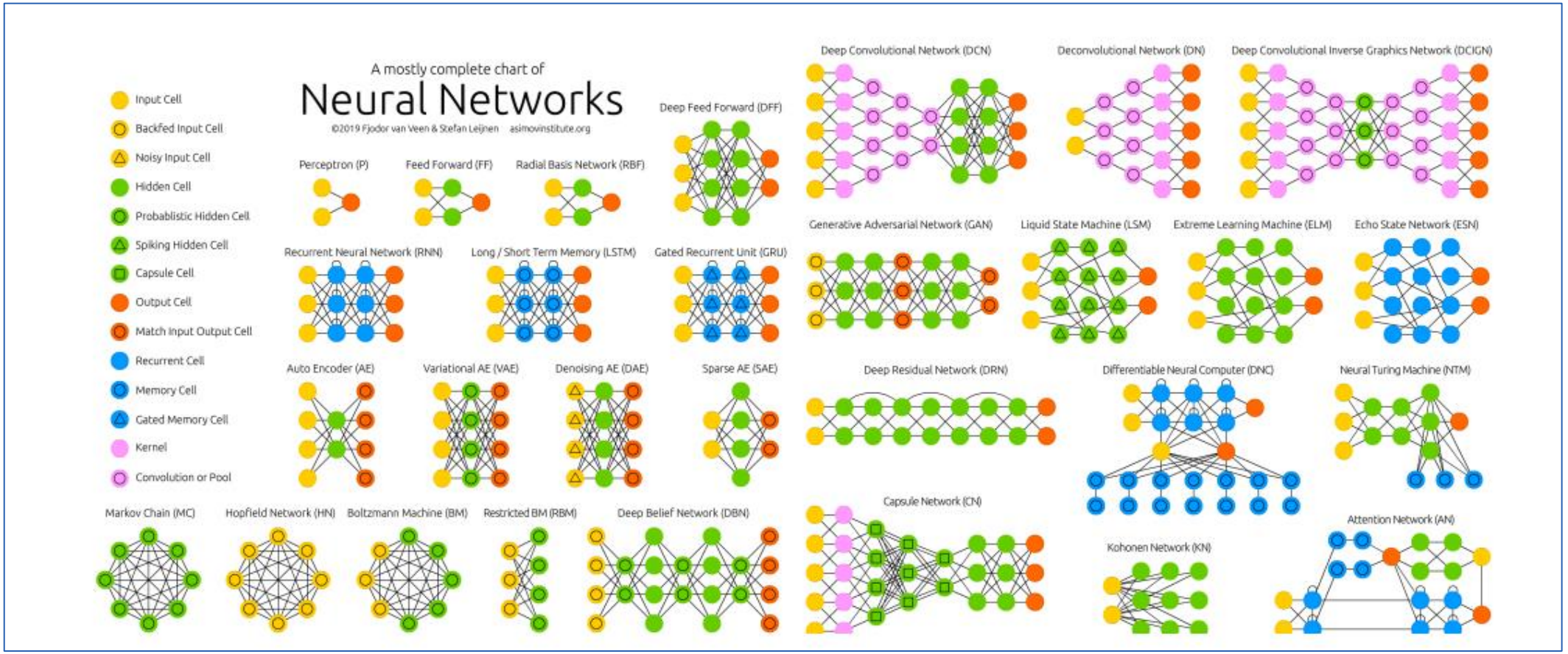
[*]Equal contribution [1]Google. Correspondence to: Nan Du, Yanping Huang, and Andrew M. Dai <{dunan,huangyp}@google.com, adai@google.com>.

Proceedings of the 39th International Conference on Machine Learning, Baltimore, Maryland, USA, PMLR 162, 2022.

Du, N., Huang, Y., Dai, A. M., Tong, S., Lepikhin, D., Xu, Y., ... & Cui, C. (2022, June). Glam: Efficient scaling of language models with mixture-of-experts. In *International Conference on Machine Learning* (pp. 5547-5569). PMLR.

Neural networks are networks of abstract neurons

Different network architectures solve different machine learning tasks



Neural networks are becoming ***more specialized***.

Self-Study: Closing with the history of artificial neural networks

Self-Study

1. Read the Wikipedia article on the “AI Winter”:

https://en.wikipedia.org/wiki/AI_winter#cite_note-3

2. Read about the AI Winters in the Wikipedia page:

https://en.wikipedia.org/wiki/History_of_artificial_intelligence

3. Go to: https://philippschmitt.com/blueprints-for-intelligence/introduction

- Read the blog
- Click on “Diagrams” and admire the evolution of graphical depictions of artificial neural networks (bottom left: you if hover with your cursor, the original reference will appear)

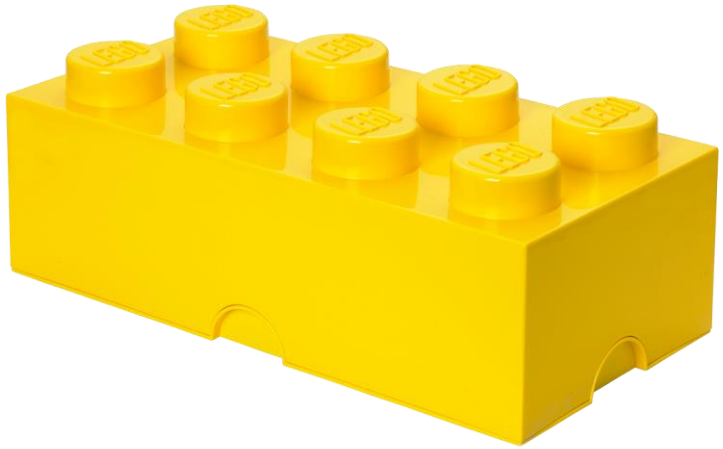
Feedforward (abstract) neural networks

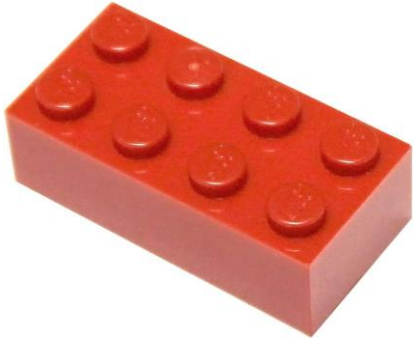
(Welcome to deep learning)

Our plan towards feedforward neural networks (FNNs)

I	We introduce the building blocks of FNNs: abstract neurons
II	We learn how to compose abstract neurons and define FNNs (we do deep learning!)
III	We show why FNNs are useful – two important theoretical results

Part I: Abstract neurons





Abstract neurons: ideas

I	**Abstract neurons are the basic computational units in neural networks**, simulating the function of biological neurons
II	**Abstract neurons are machine learning models**. They compute outputs by applying (1) weights, and (2) an activation function to their inputs
III	**They can be trained like any other machine learning models** (e.g., using gradient descent)

Abstract neurons: definition from Calin (2020)

Abstract neuron: what it does

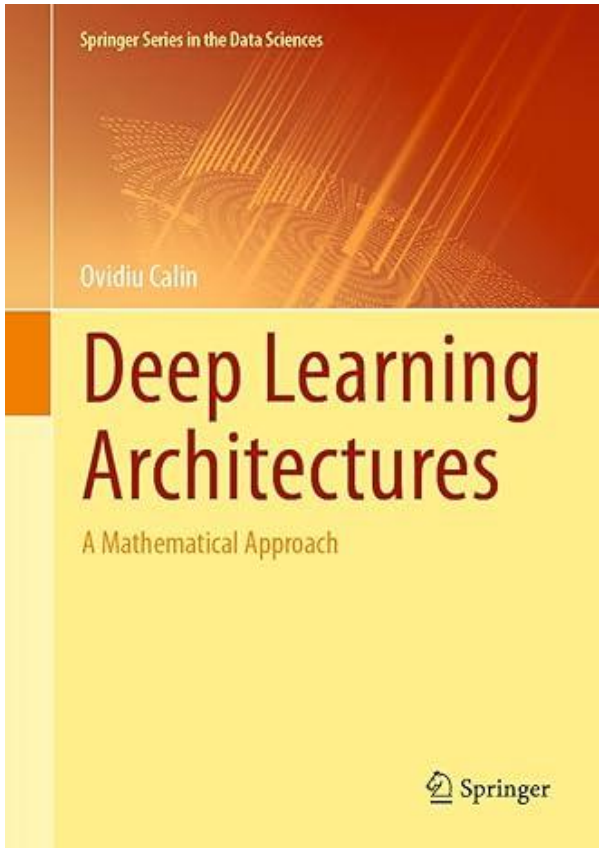
"[An abstract neuron] mimics a biological neuron, consisting of an input (incoming signal), weights (synaptic weights) and activation function (neuron firing model)" (pag. 133)

Abstract neuron: definition

An abstract neuron is a quadruple $(x, \boldsymbol{w}, \varphi, y)$ where $x^T = (x_0, x_1, \ldots, x_d)$ is the input vector with $x_0 := 1$, $\boldsymbol{w}^T = (w_0, w_1, \ldots, w_d)$ is the weights vector, and φ is the activation function that defines the output

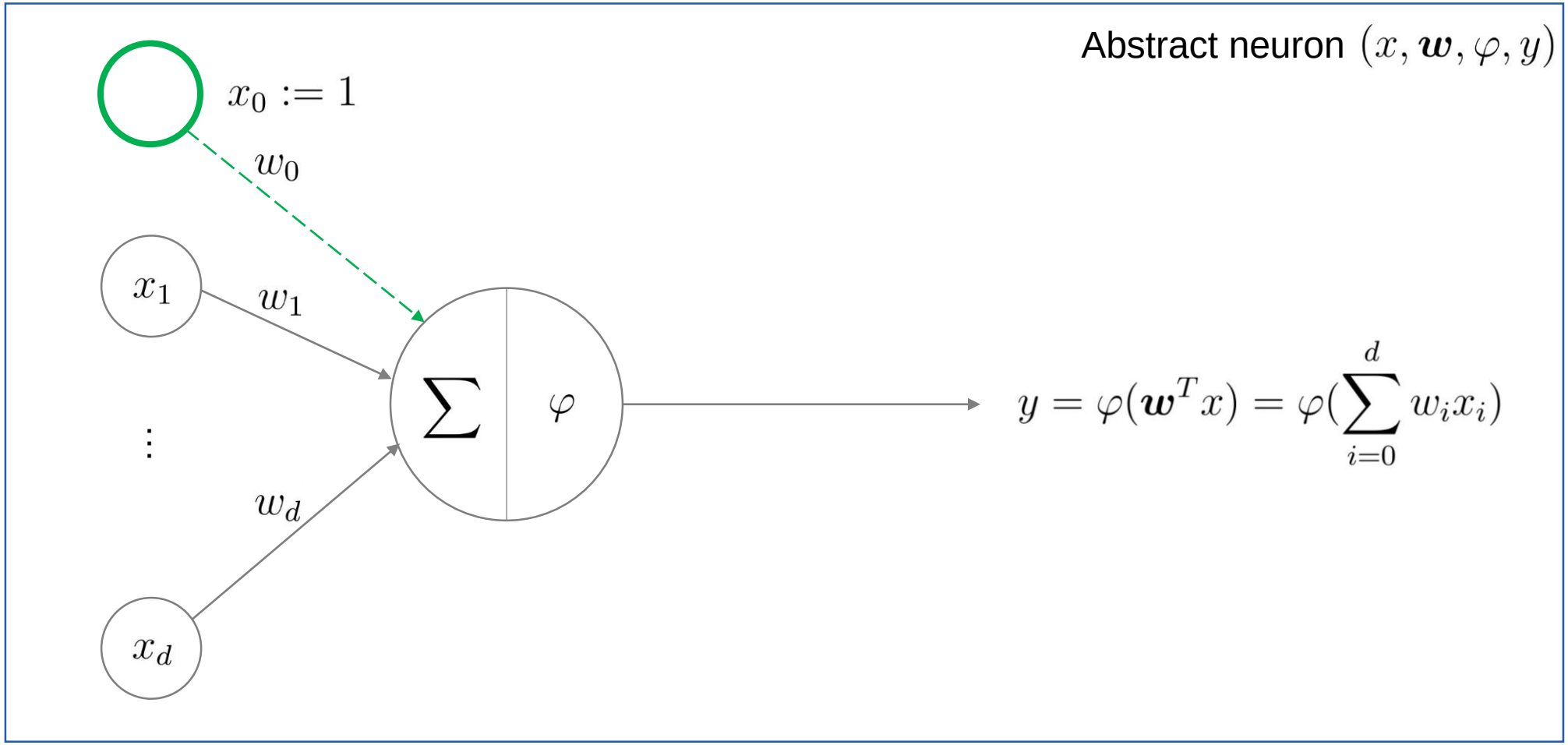
$$y = \varphi(\boldsymbol{w}^T x) = \varphi(\sum_{i=0}^{d} w_i x_i).$$

The weight w_0 is called the *bias*.



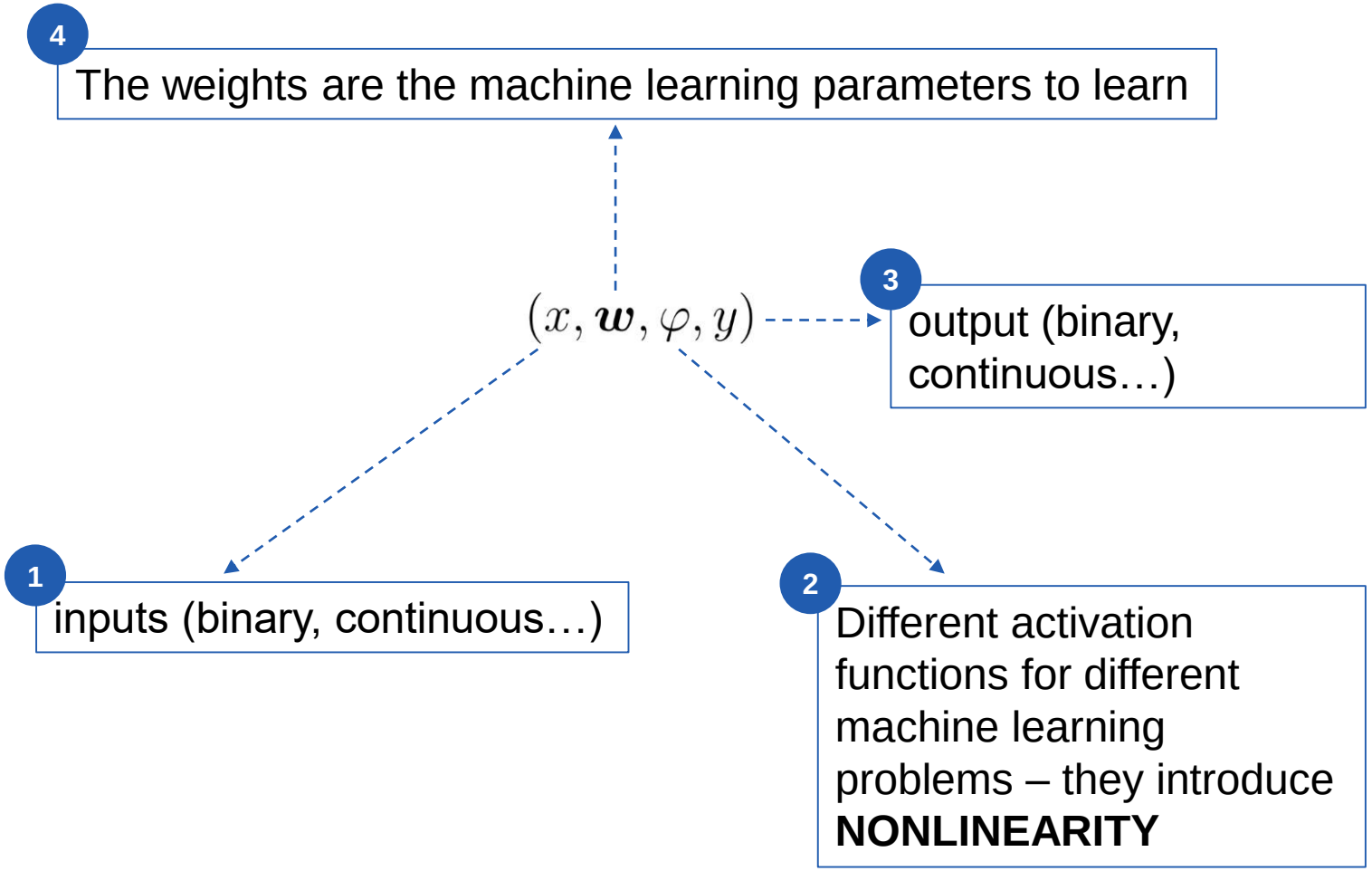
Calin, O. (2020). Deep learning architectures. A mathematical approach. New York City: Springer International Publishing.

Abstract neurons: graphical representation



…ok, but now? Why do we introduce them?

Abstract neurons are flexible, simple machine learning models



Machine learning model notation

$$f_{\boldsymbol{w}}(x) = \varphi(\boldsymbol{w}^T x)$$

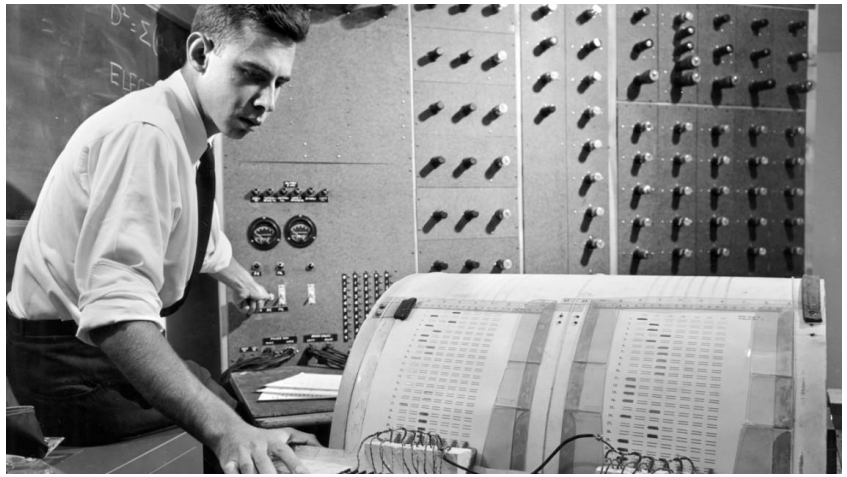
Perceptron

It is the first example of abstract neuron

Definition

A **perceptron** is an abstract neuron $(x, \boldsymbol{w}, \varphi, y)$ with binary inputs, i.e., $x_i \in \{0, 1\},\ \forall i$ and with the activation function given by the Heaviside function

$$\varphi(z) = \begin{cases} 0, & \text{if } z < 0 \\ 1, & \text{if } z \geq 0. \end{cases}$$



Frank Rosenblatt. Source: https://news.cornell.edu/stories/2019/09/professors-perceptron-paved-way-ai-60-years-too-soon

Perceptron

It is the first example of abstract neuron

Definition

A **perceptron** is an abstract neuron $(x, \boldsymbol{w}, \varphi, y)$ with binary inputs, i.e., $x_i \in \{0, 1\},\ \forall i$ and with the activation function given by the Heaviside function

$$\varphi(z) = \begin{cases} 0, & \text{if } z < 0 \\ 1, & \text{if } z \geq 0. \end{cases}$$

$$y = \varphi\Big(\sum_{i=0}^{d} w_i x_i\Big) = \varphi(w_0 + w_1 x_1 + \cdots + w_d x_d) = \begin{cases} 0, & \text{if } w_0 + w_1 x_1 + \cdots + w_d x_d < 0 \\ 1, & \text{if } w_0 + w_1 x_1 + \cdots + w_d x_d \geq 0. \end{cases}$$

Perceptron

It is the first example of abstract neuron

Definition

A **perceptron** is an abstract neuron $(x, \boldsymbol{w}, \varphi, y)$ with binary inputs, i.e., $x_i \in \{0, 1\},\ \forall i$ and with the activation function given by the Heaviside function

$$\varphi(z) = \begin{cases} 0, & \text{if } z < 0 \\ 1, & \text{if } z \geq 0. \end{cases}$$

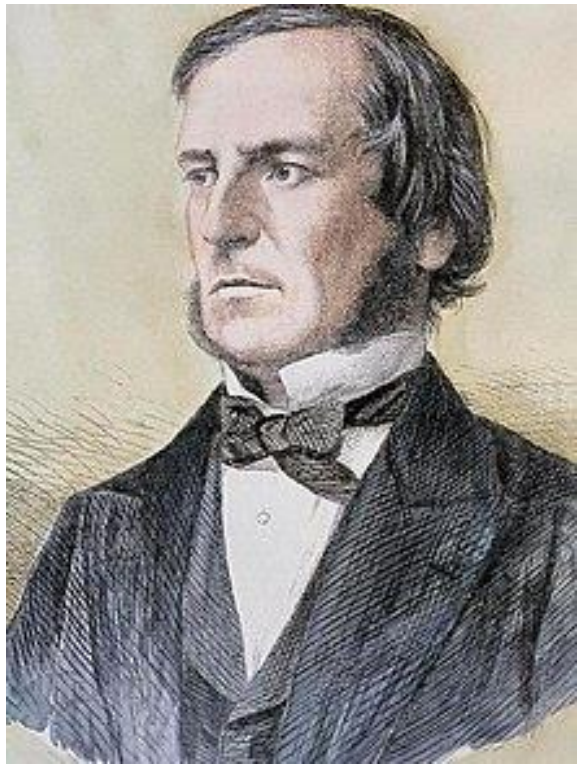
$$y = \varphi\left(\sum_{i=0}^{d} w_i x_i\right) = \varphi(w_0 + w_1 x_1 + \cdots + w_d x_d) = \begin{cases} 0, & \text{if } w_0 + w_1 x_1 + \cdots + w_d x_d < 0 \\ 1, & \text{if } w_0 + w_1 x_1 + \cdots + w_d x_d \geq 0. \end{cases}$$

The perceptron **fires** if the linear combination of its inputs and weights is equal or above a threshold, i.e., the bias. Otherwise, it does not fire.

The **bias** controls the propensity of the perceptron to **fire**

Logical (or Boolean) algebra

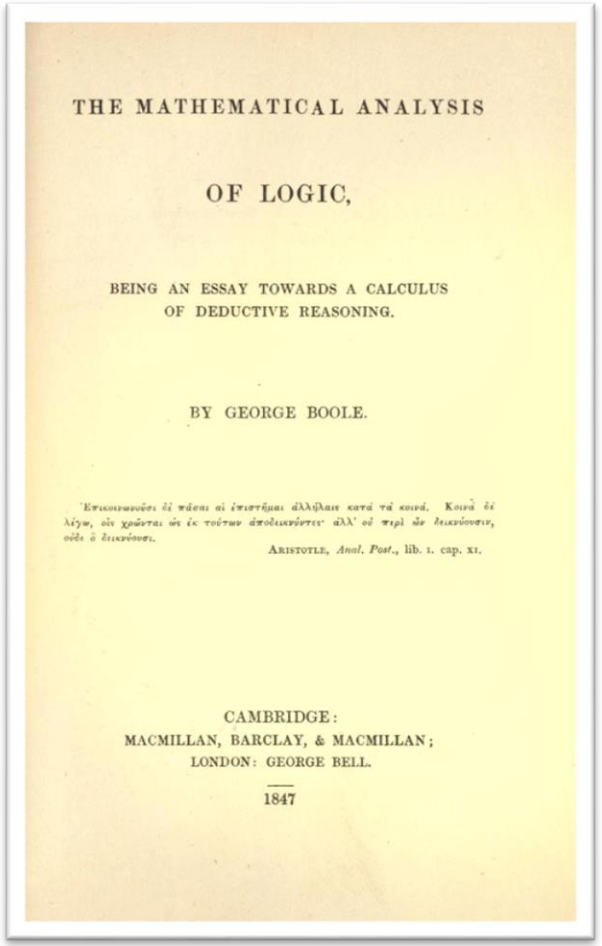
1	Logical (or Boolean) algebra is the branch of algebra that deals with variables that have two distinct values, typically represented as true (1) and false (0), and the logical operations that act on these values
2	It forms the foundation of digital logic in electrical engineering and computer science
3	Logical algebra allows for the representation and manipulation of logical statements and the design of digital circuits which are the core of our computational systems



George Boole (1815-1864)
Source: Wikipedia

Logical (or Boolean) algebra

1	Logical (or Boolean) algebra is the branch of algebra that deals with variables that have two distinct values, typically represented as true (1) and false (0), and the logical operations that act on these values
2	It forms the foundation of digital logic in electrical engineering and computer science
3	Logical algebra allows for the representation and manipulation of logical statements and the design of digital circuits which are the core of our computational systems



THE MATHEMATICAL ANALYSIS

OF LOGIC,

BEING AN ESSAY TOWARDS A CALCULUS
OF DEDUCTIVE REASONING.

BY GEORGE BOOLE.

Ἐπιστατοῦσι δὲ πᾶσαι αἱ ἐπιστῆμαι ἀλλήλαις κατὰ τὰ κοινά. Κοινὰ δὲ λέγω, οἷς χρῶνται ὡς ἐκ τούτων ἀποδεικνύοντες· ἀλλ' οὐ περὶ ὧν δεικνύουσιν, οὐδὲ ὃ δεικνύουσι.

ARISTOTLE, *Anal. Post.*, lib. i. cap. xi.

CAMBRIDGE:
MACMILLAN, BARCLAY, & MACMILLAN;
LONDON: GEORGE BELL.

1847

Boole, G. (1847). *The mathematical analysis of logic*. Philosophical Library.
Available at https://www.gutenberg.org/ebooks/36884

Logical (or Boolean) functions

Their arguments are binary, as well as the results they return

$$f : \{0,1\}^k \rightarrow \{0,1\} \quad k \geq 0$$

$$(x_1, \ldots, x_k) \mapsto f(x_1, \ldots, x_k)$$

1. **Warm-up**: unary (*k=1*) logical functions
2. Let us investigate whether perceptrons can implement *k=2* logical functions…

Which are the unary (*k=1*) logical functions?

Let us list them on the blackboard…

The perceptron implements two important logical functions (*k=2*)

In machine learning jargon: the perceptron learns **AND**

AND $(\wedge)$

x_1	x_2	$x_1 \wedge x_2$
0	0	0
0	1	0
1	0	0
1	1	1

$$\wedge : \{0,1\}^2 \to \{0,1\}$$
$$(x_1, x_2) \mapsto \wedge(x_1, x_2)$$

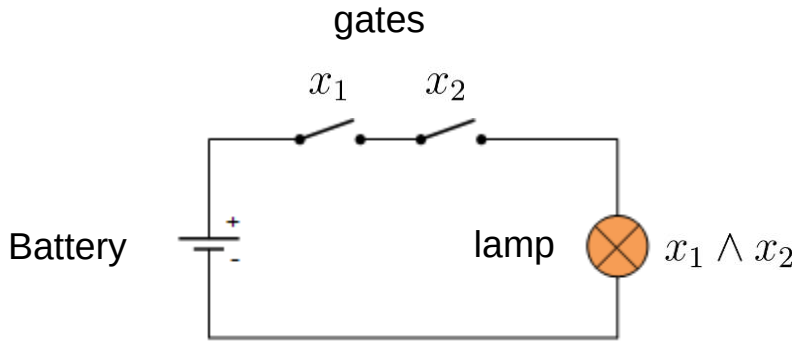
The perceptron implements two important logical functions (*k=2*)

In machine learning jargon: the perceptron learns **AND**

AND $(\wedge)$

x_1	x_2	$x_1 \wedge x_2$
0	0	0
0	1	0
1	0	0
1	1	1

Implementing **AND** with an electrical circuit



AND: “the lamp turns on if and only if both gates are closed”. The output of **AND** is true (=1) if and only if both inputs are true (=1)

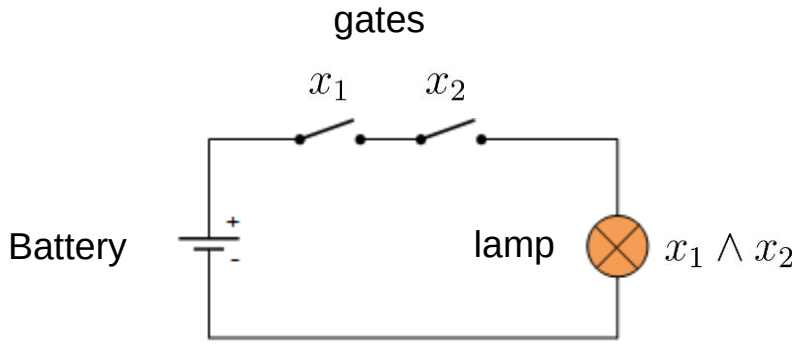
The perceptron implements two important logical functions (*k=2*)

In machine learning jargon: the perceptron learns **AND**

AND $(\wedge)$

x_1	x_2	$x_1 \wedge x_2$
0	0	0
0	1	0
1	0	0
1	1	1

Implementing **AND** with an electrical circuit



AND: "the lamp turns on if and only if both gates are closed". The output of **AND** is true (=1) if and only if both inputs are true (=1)

Can we introduce a perceptron $(x, \boldsymbol{w}, \varphi, y)$ such that $\wedge(x_1, x_2) = \varphi(\sum_{i=0}^{2} w_i x_i)$ for all inputs (x_1, x_2) ?

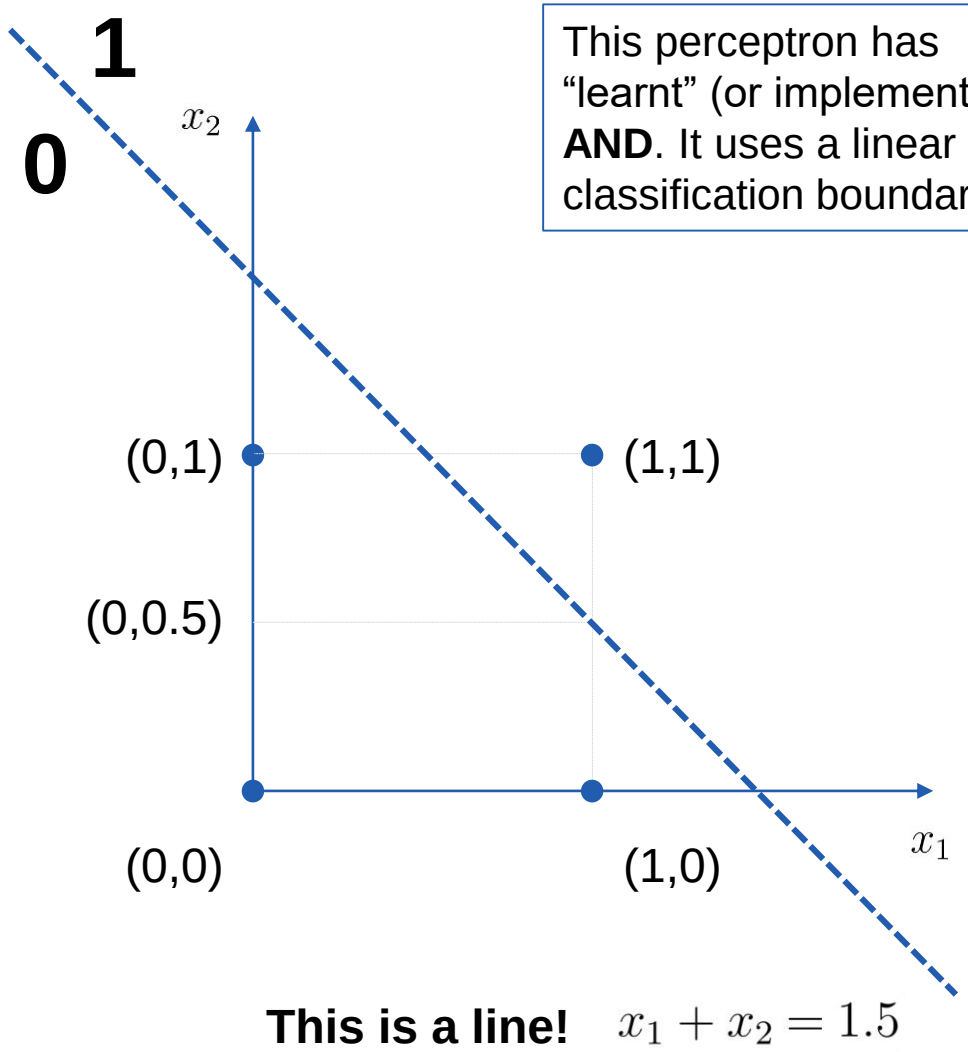
The perceptron implements two important logical functions (*k=2*)

In machine learning jargon: the perceptron learns **AND**

AND ($\wedge$)

x_1	x_2	$x_1 \wedge x_2$
0	0	0
0	1	0
1	0	0
1	1	1

$$y = \varphi(x_1 + x_2 - 1.5) = \begin{cases} 0, & \text{if } x_1 + x_2 < 1.5 \\ 1, & \text{if } x_1 + x_2 \geq 1.5 \end{cases}$$



This perceptron has "learnt" (or implemented) **AND**. It uses a linear classification boundary

This is a line! $x_1 + x_2 = 1.5$

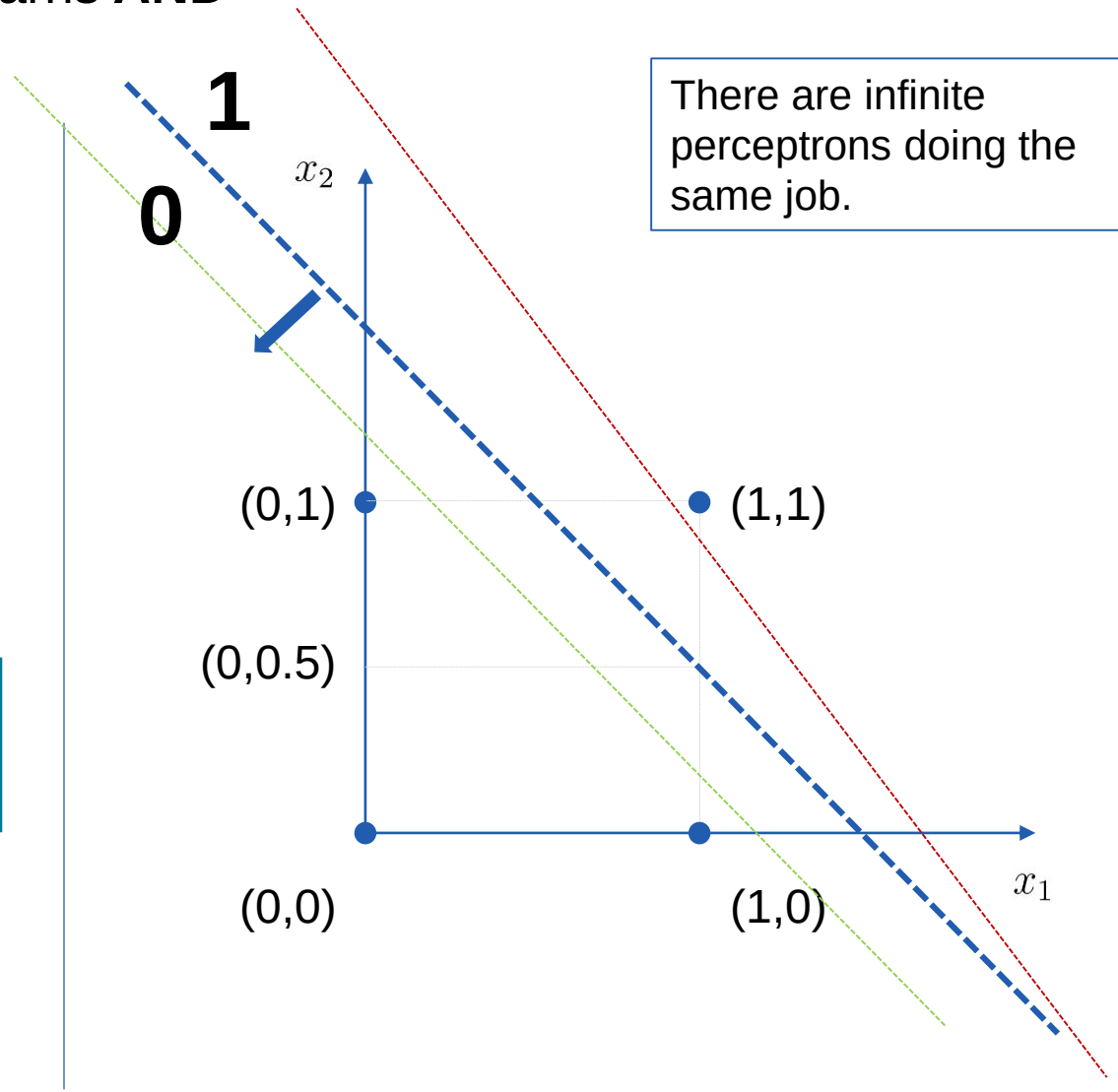
The perceptron implements two important logical functions (*k=2*)

In machine learning jargon: the perceptron learns **AND**

AND ($\wedge$)

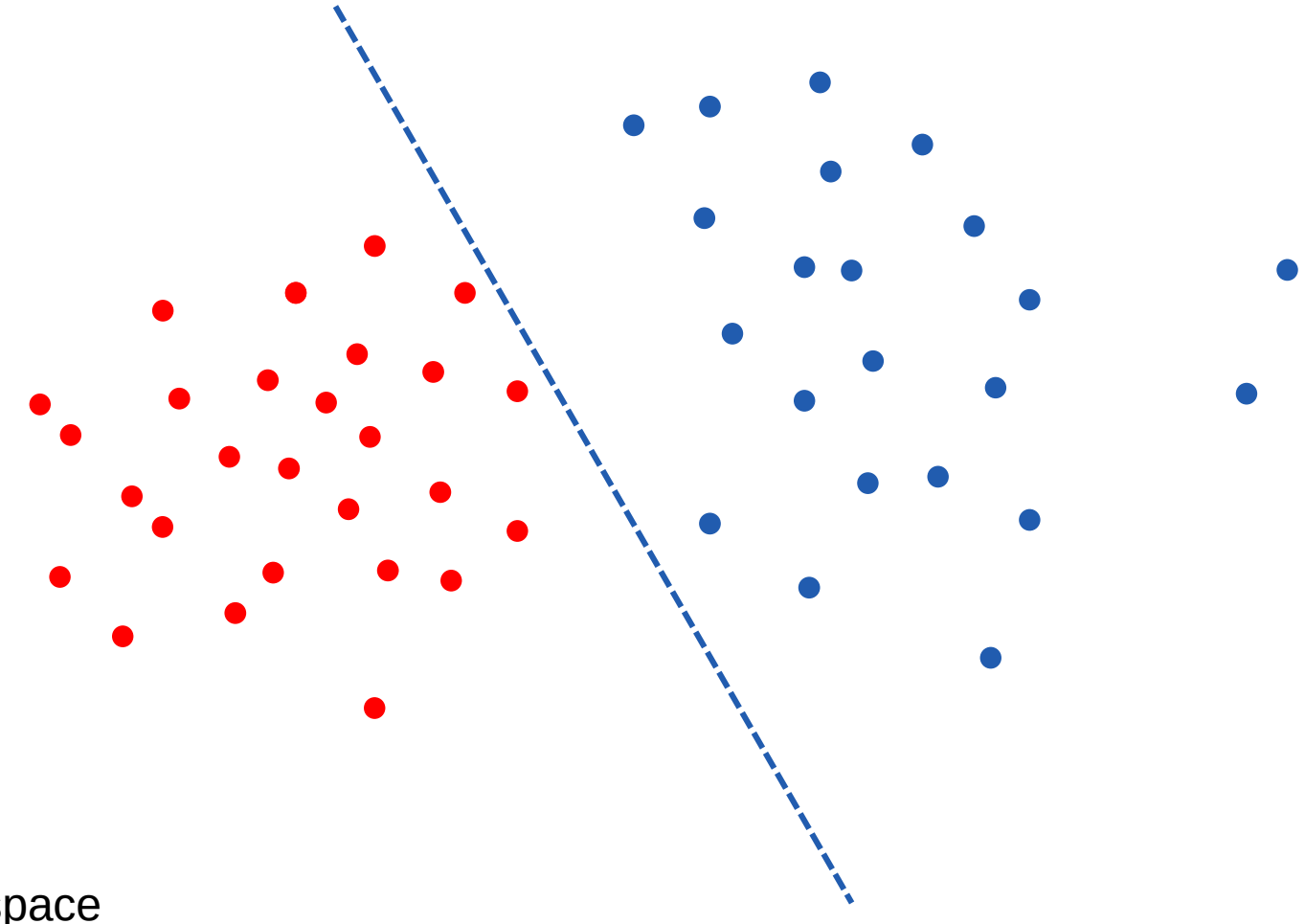
x_1	x_2	$x_1 \wedge x_2$
0	0	0
0	1	0
1	0	0
1	1	1

$$y = \varphi(x_1 + x_2 - 1.5) = \begin{cases} 0, & \text{if } x_1 + x_2 < 1.5 \\ 1, & \text{if } x_1 + x_2 \geq 1.5 \end{cases}$$



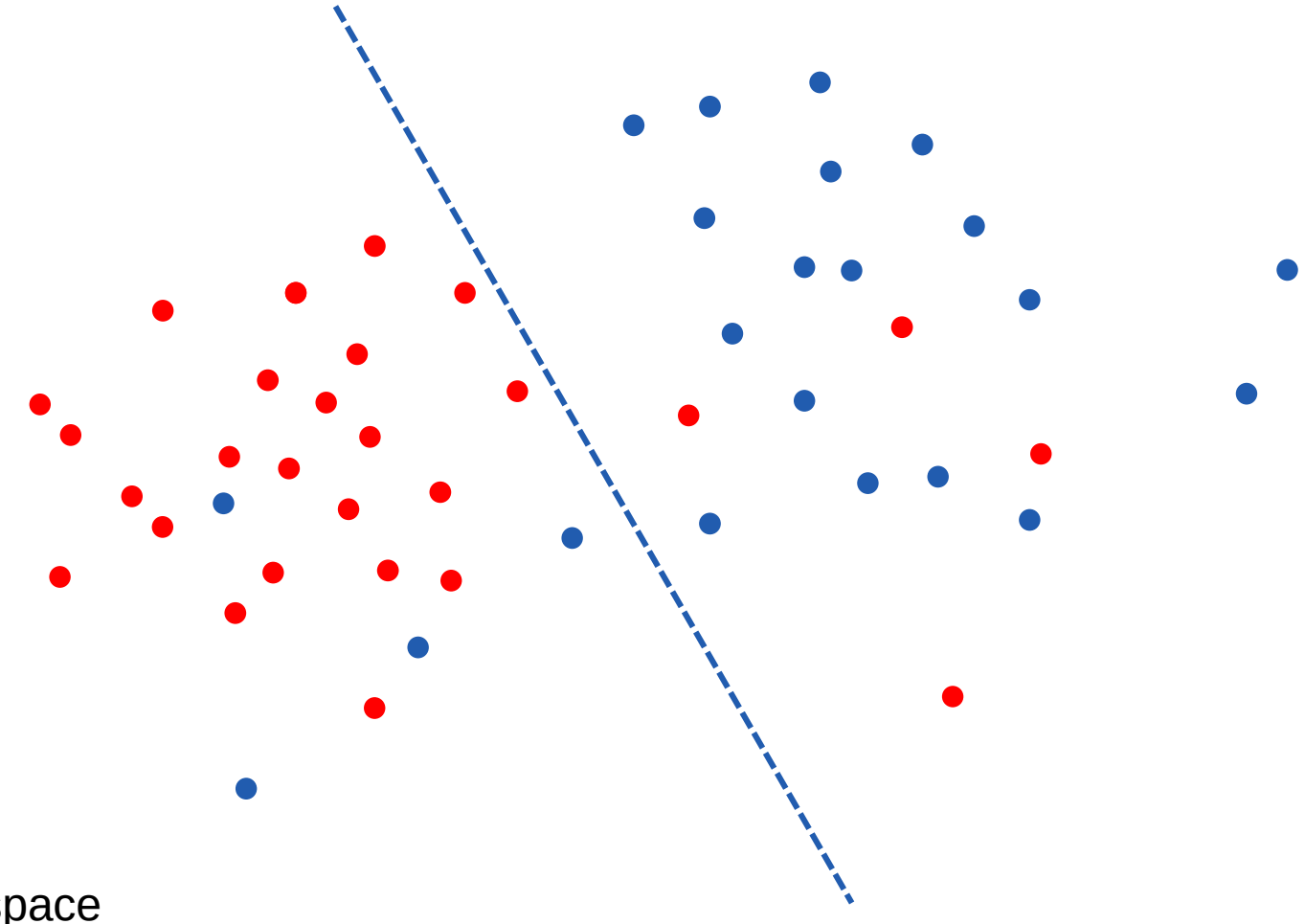
There are infinite perceptrons doing the same job.

Remark: the perceptron is a linear classifier



Some *d*-dimensional space

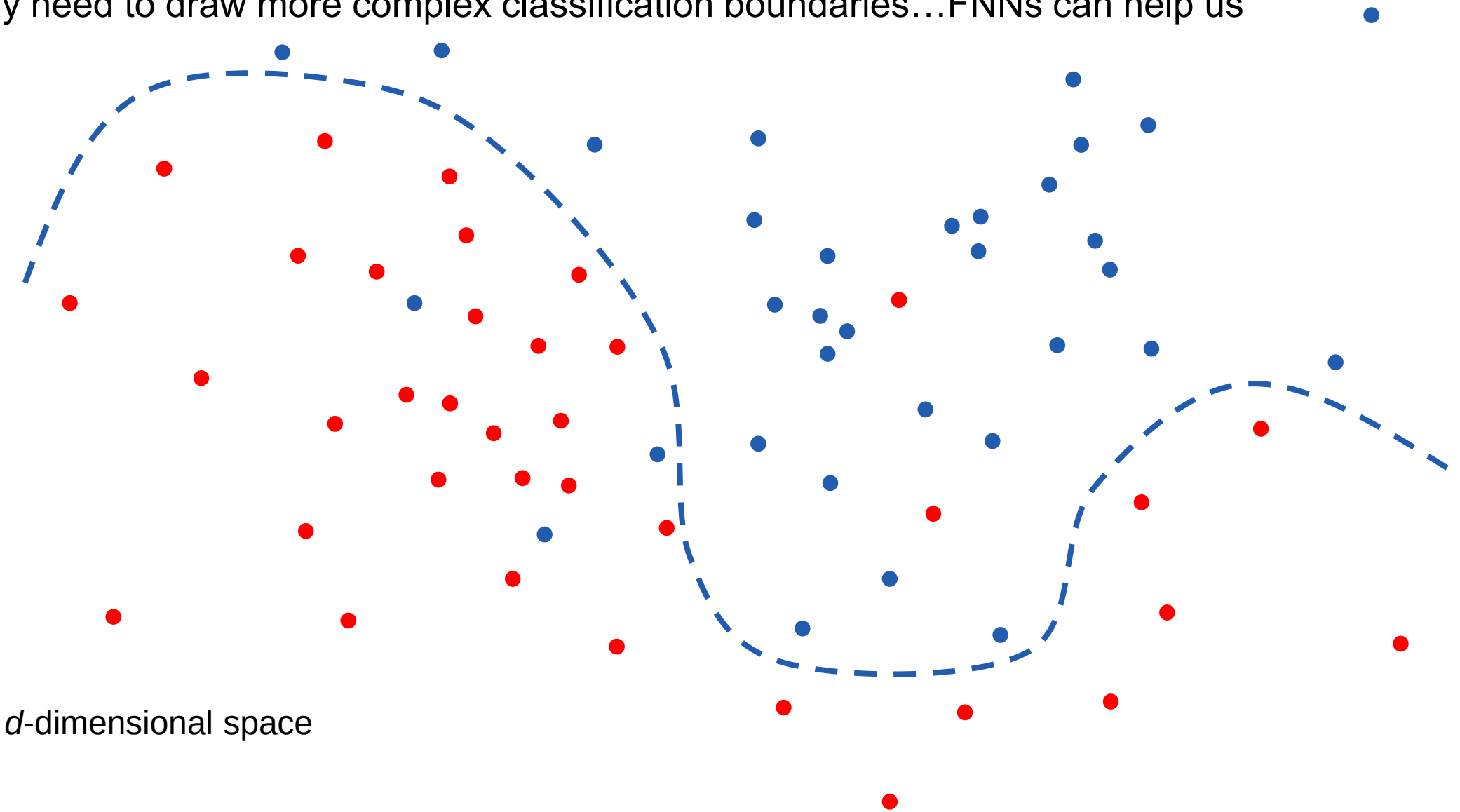
Remark: the perceptron is a linear classifier



Some *d*-dimensional space

Remark: the perceptron is a linear classifier

We may need to draw more complex classification boundaries…FNNs can help us



Some *d*-dimensional space

The perceptron implements two important logical functions

In machine learning jargon: the perceptron learns **AND** and **OR**

OR $(\vee)$

x_1	x_2	$x_1 \vee x_2$
0	0	0
0	1	1
1	0	1
1	1	1

The output of **OR** is true (=1) if and only if at least one input is true (=1)

(can you design an electric circuit implementing **OR***?)*

The perceptron implements two important logical functions

In machine learning jargon: the perceptron learns **AND** and **OR**

OR $(\vee)$

x_1	x_2	$x_1 \vee x_2$
0	0	0
0	1	1
1	0	1
1	1	1

$$y = \varphi(x_1 + x_2 - 0.5) = \begin{cases} 0, & \text{if } x_1 + x_2 < 0.5 \\ 1, & \text{if } x_1 + x_2 \geq 0.5 \end{cases}$$

(can you prove this? Can you draw this perceptron?)

Does the perceptron implement/learn also **XOR**?

A more complex example of logical function

XOR ($\oplus$)

x_1	x_2	$x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

XOR implements "exclusive or": its output is true (=1) if and only if *only one* of the inputs is true (=1)

(Designing an electric circuit implementing **XOR** *is not as trivial as in the previous examples…we skip this)*

Does the perceptron implement/learn also **XOR**?

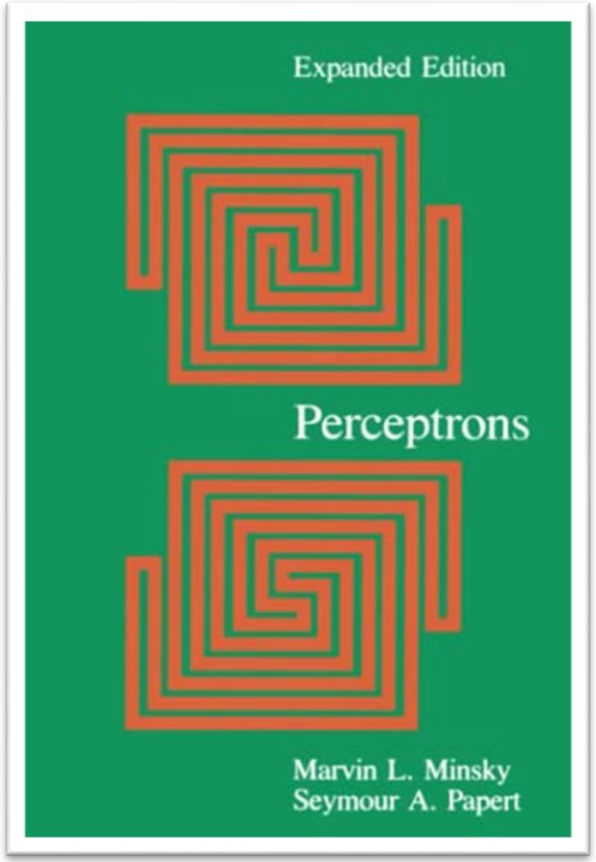
A more complex example of logical function

XOR $(\oplus)$

x_1	x_2	$x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

IMPORTANT RESULT

There exists no perceptron that can implement the logical function **XOR**.



Minsky, M. L., & Papert, S. A. (1988). Perceptrons: expanded edition.

Does the perceptron implement/learn also **XOR**?

A more complex example of logical function

XOR $(\oplus)$

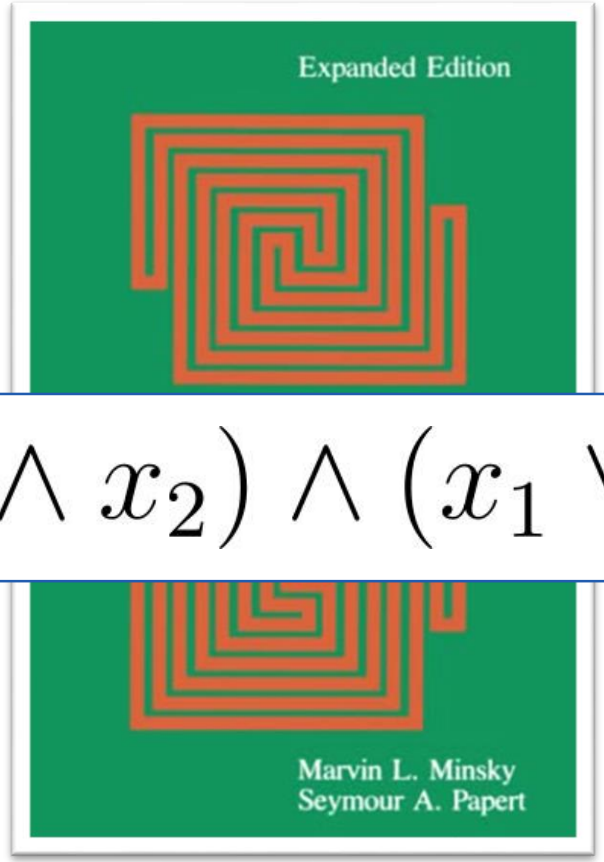
x_1	x_2	$x_1 \oplus x_2$
0	0	0
0	1	1

Hint

$$x_1 \oplus x_2 = \neg(x_1 \wedge x_2) \wedge (x_1 \vee x_2)$$

IMPORTANT RESULT

There exists no perceptron that can implement the logical function **XOR**.



Minsky, M. L., & Papert, S. A. (1988). Perceptrons: expanded edition.

Perceptrons: main takeways

I	**Perceptrons** can be used to implement/learn logical functions
II	**Perceptrons** can not implement all logical functions (e.g., XOR)
III	**The use of binary inputs and output** is a bit limiting…can we relax these conditions?

The sigmoid neuron

It is more flexible than the perceptron and it is an old friend of ours

Definition

A **sigmoid neuron** is an abstract neuron $(x, \boldsymbol{w}, \varphi, y)$ with the activation function given by the sigmoid function

$$\varphi(z) = \sigma(z) = \frac{1}{1+e^{-z}}.$$

Standard notation of the sigmoid function (on all books…)

The sigmoid neuron

It is more flexible than the perceptron and it is an old friend of ours

Definition

A **sigmoid neuron** is an abstract neuron $(x, \boldsymbol{w}, \varphi, y)$ with the activation function given by the sigmoid function

$$\varphi(z) = \sigma(z) = \frac{1}{1+e^{-z}}.$$

$$y = \varphi(\boldsymbol{w}^T x) = \sigma(\sum_{i=0}^{d} w_i x_i) = \frac{1}{1+e^{-w_0 - w_1 x_1 - \cdots - w_d x_d}}$$

The sigmoid neuron is equivalent to the logistic regression model

The linear neuron

It is more flexible than the perceptron and it is an old friend of ours

Definition

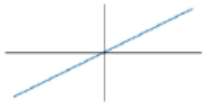
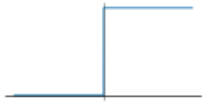
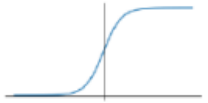
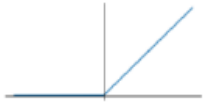
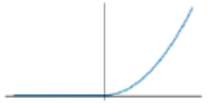
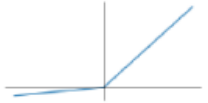
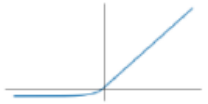
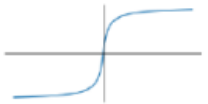
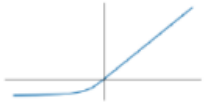
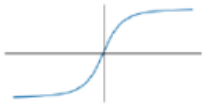
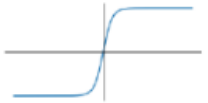
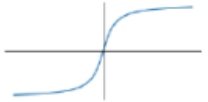
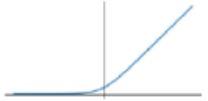
A **linear neuron** is an abstract neuron $(x, \boldsymbol{w}, \varphi, y)$ with the activation function given by the identity function

$$\varphi(z) = z$$

$$y = \varphi(\boldsymbol{w}^T x) = \sum_{i=0}^{d} w_i x_i = w_0 + w_1 x_1 + \cdots + w_d x_d$$

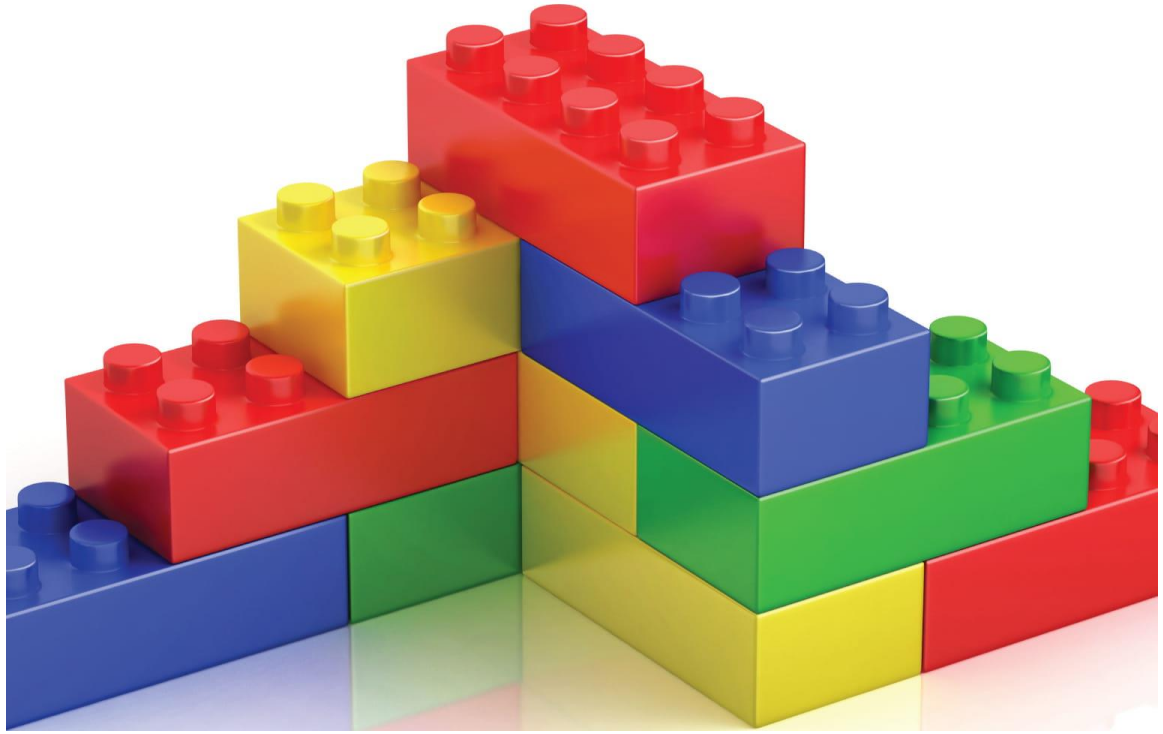
The linear neuron is equivalent to the linear regression model

A reminder: using different activation functions we can generate different outputs

linear	x	
Heaviside / step function	$1_{(0,\infty)}(x)$	
logistic / sigmoid	$\frac{1}{1+e^{-x}}$	
rectified linear unit (ReLU)	$\max\{0, x\}$	
power rectified linear unit	$\max\{0, x\}^k$ for $k \in \mathbb{N}$	
parametric ReLU (PReLU)	$\max\{ax, x\}$ for $a \geq 0, a \neq 1$	
exponential linear unit (ELU)	$x \cdot 1_{[0,\infty)}(x) + (e^x - 1) \cdot 1_{(-\infty,0)}(x)$	
softsign	$\frac{x}{1+\lvert x\rvert}$	
inverse square root linear unit	$x \cdot 1_{[0,\infty)}(x) + \frac{x}{\sqrt{1+ax^2}} \cdot 1_{(-\infty,0)}(x)$ for $a > 0$	
inverse square root unit	$\frac{x}{\sqrt{1+ax^2}}$ for $a > 0$	
tanh	$\frac{e^x - e^{-x}}{e^x + e^{-x}}$	
arctan	$\arctan(x)$	
softplus	$\ln(1 + e^x)$	

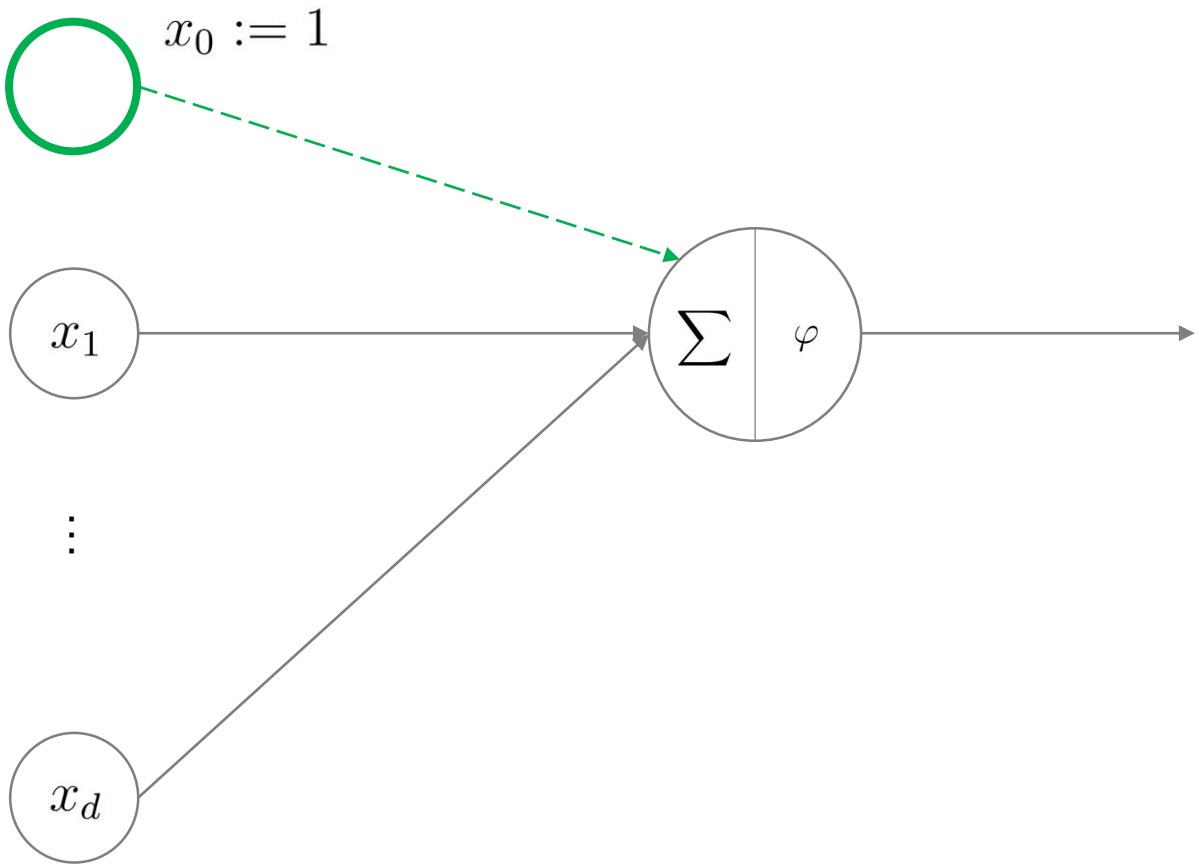
Main idea: Activation functions allow introducing NONLINEARITY

Part II: composing abstract neurons into FNNs



FNNs are obtained by connecting abstract neurons.

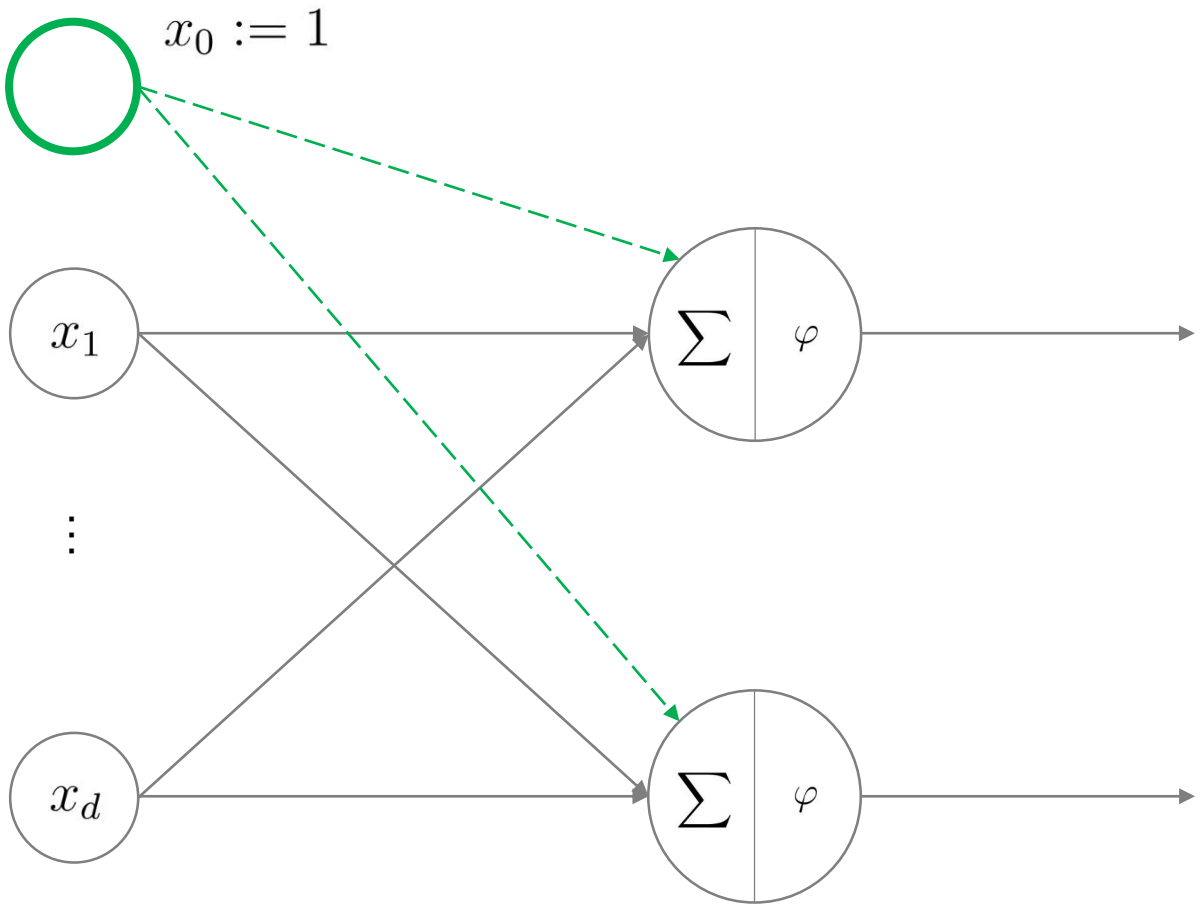
Let us understand the process with a simple example



We start with an abstract neuron

FNNs are obtained by connecting abstract neurons.

Let us understand the process with a simple example



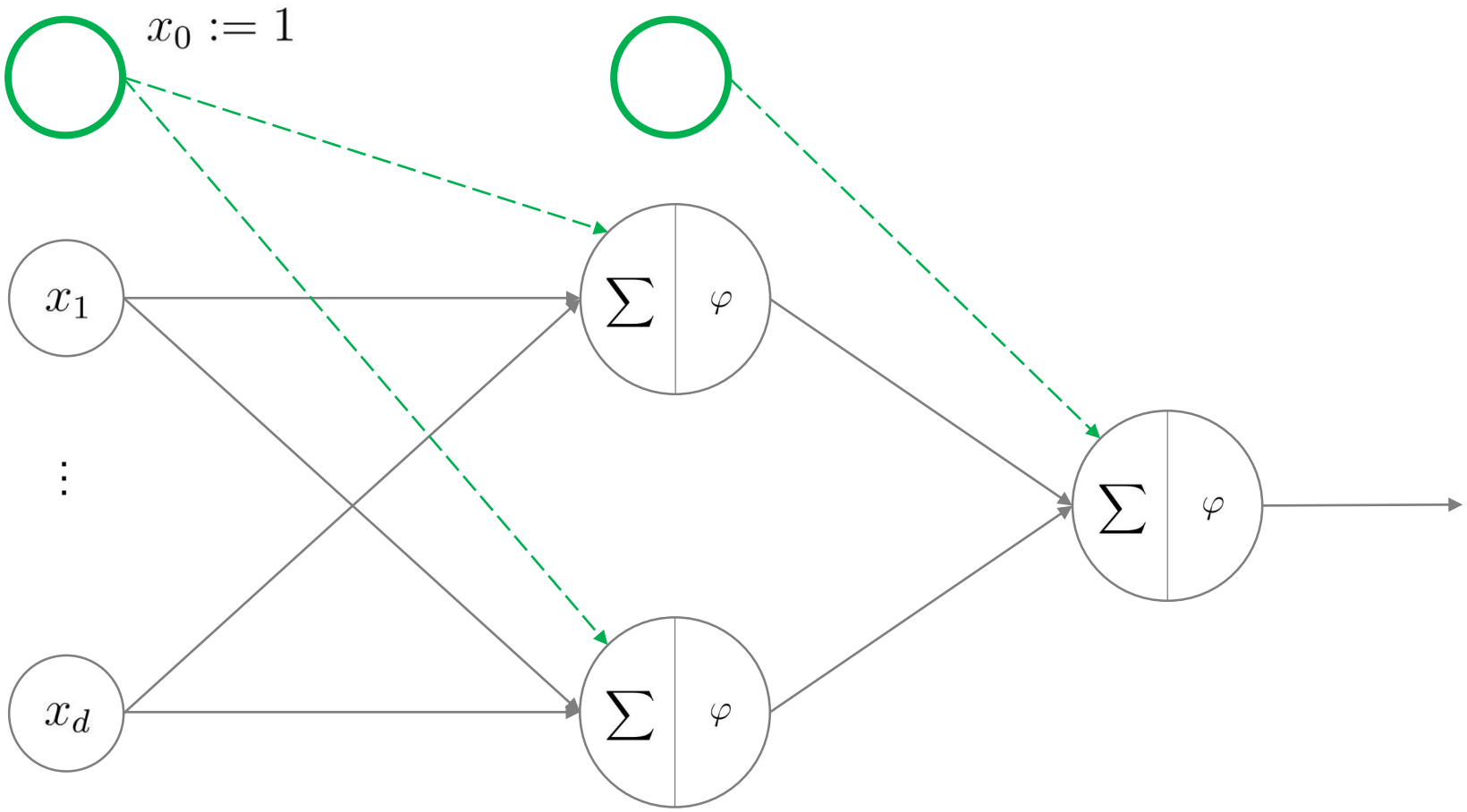
We connect the inputs to a second abstract neuron

Note that the bias is connected to both neurons (by definition of neuron)

We want a single output – how to do it?

FNNs are obtained by connecting abstract neurons.

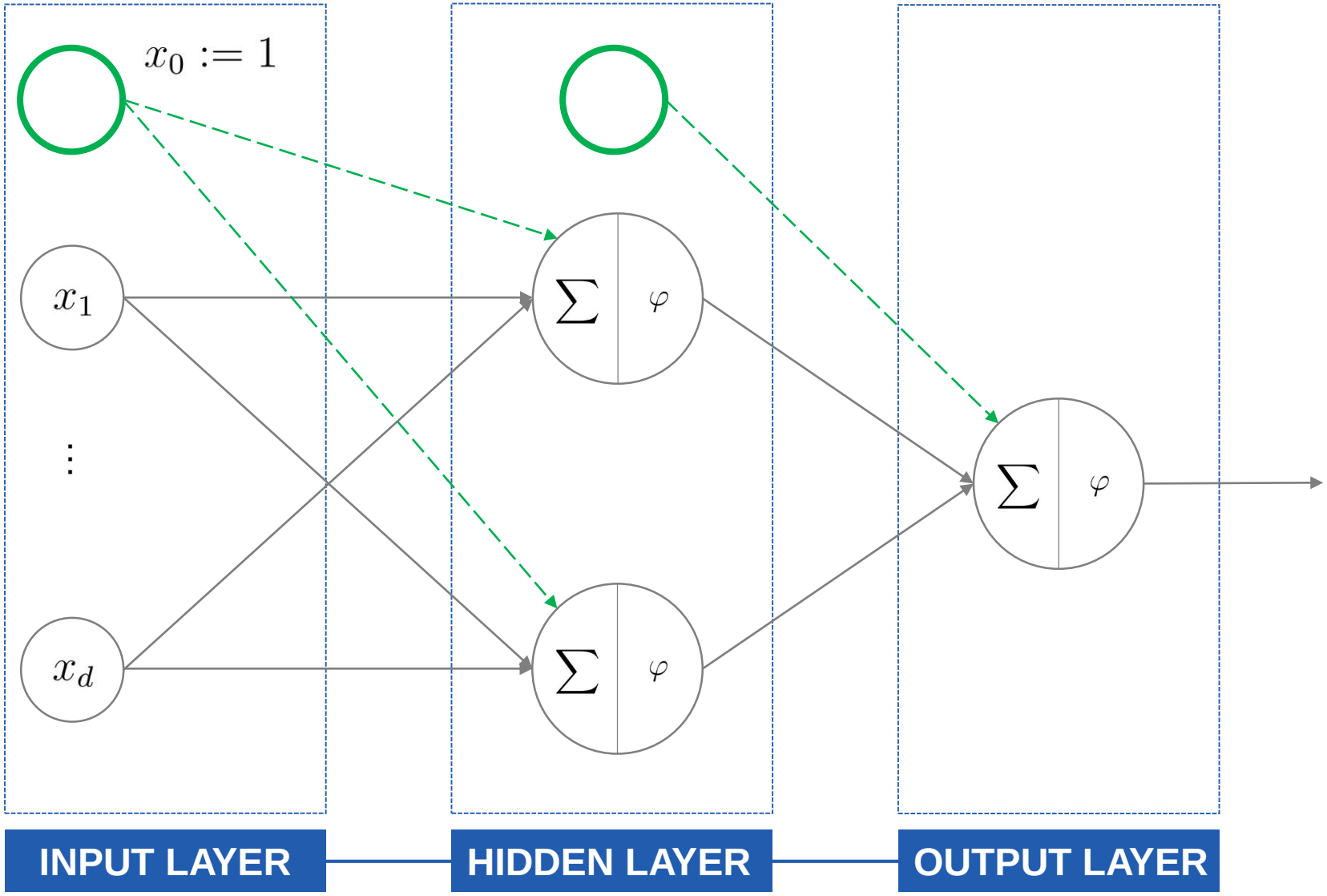
Let us understand the process with a simple example



We add a third neuron to generate a single output (note the additional bias)

FNNs are obtained by connecting abstract neurons.

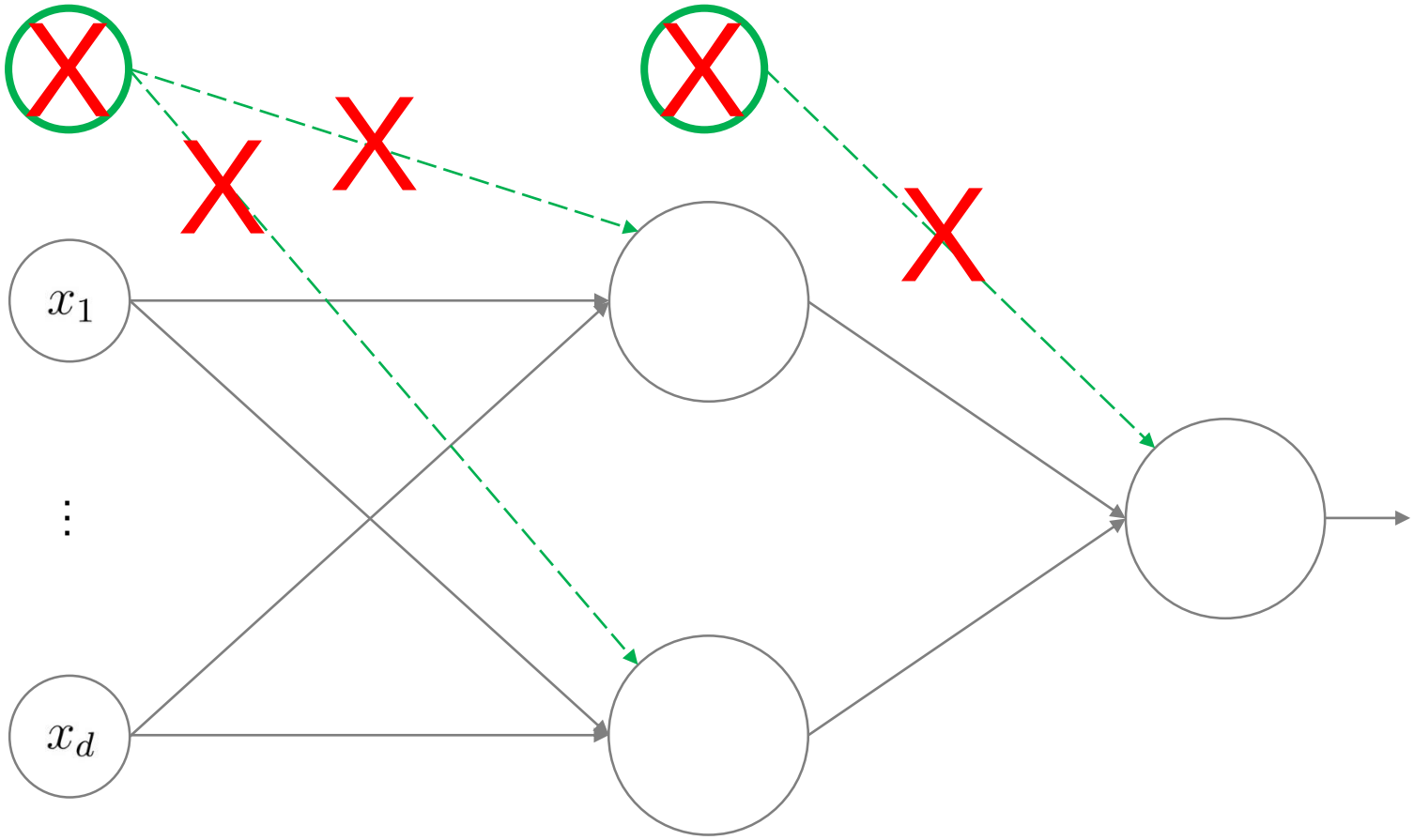
Let us understand the process with a simple example



This is a **FNN with one hidden layer**

FNNs are also called **“multilayer perceptrons”**

Before introducing the function implemented by this FNN, a reminder on the modern graphical notation of FNNs

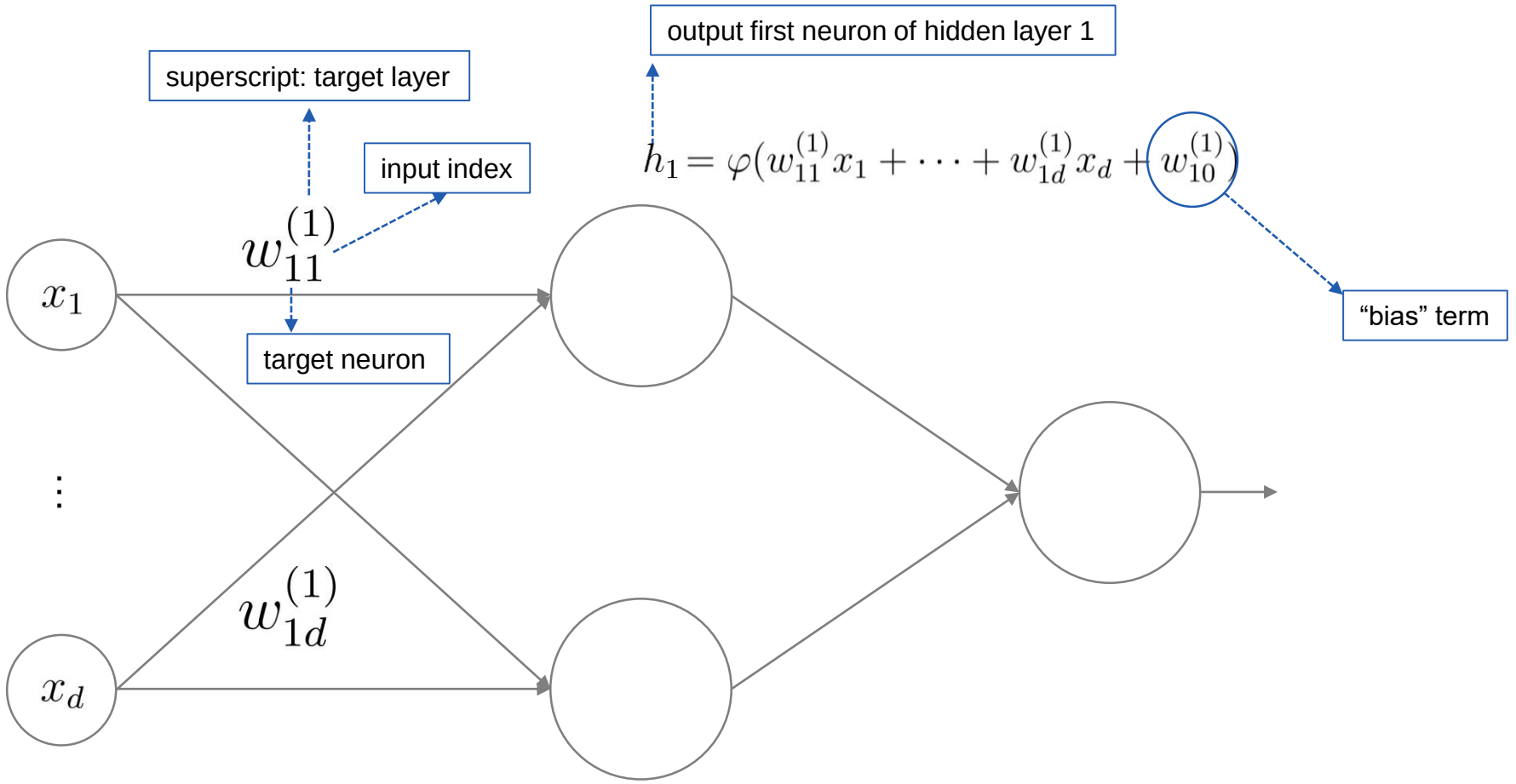


Drop the nodes and edges corresponding to the biases (we keep the biases in the FNN formulae)

Nodes have usually no reference to linear combinations and the activation function

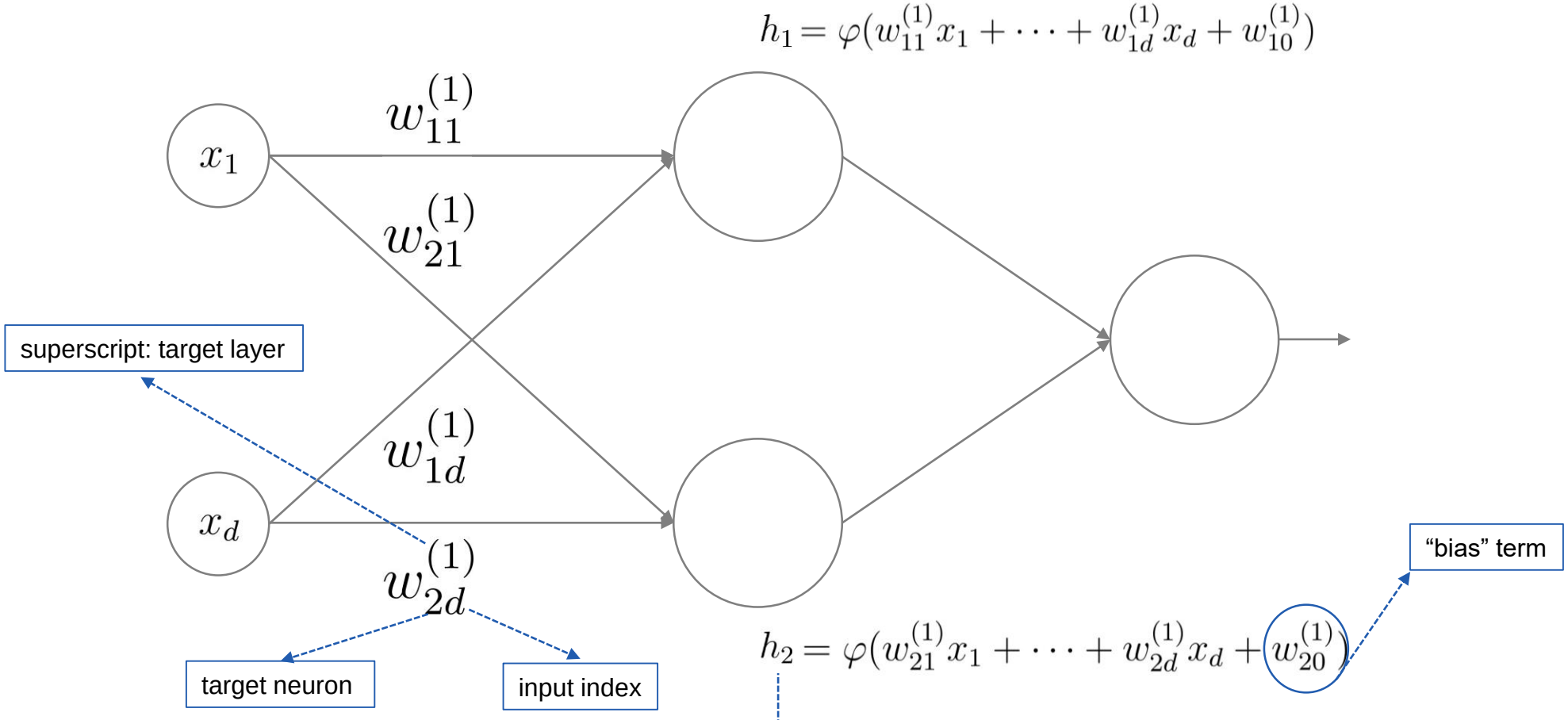
Let us introduce the function implemented by this FNN

We need to manage the different elements of the FNN by introducing some indices



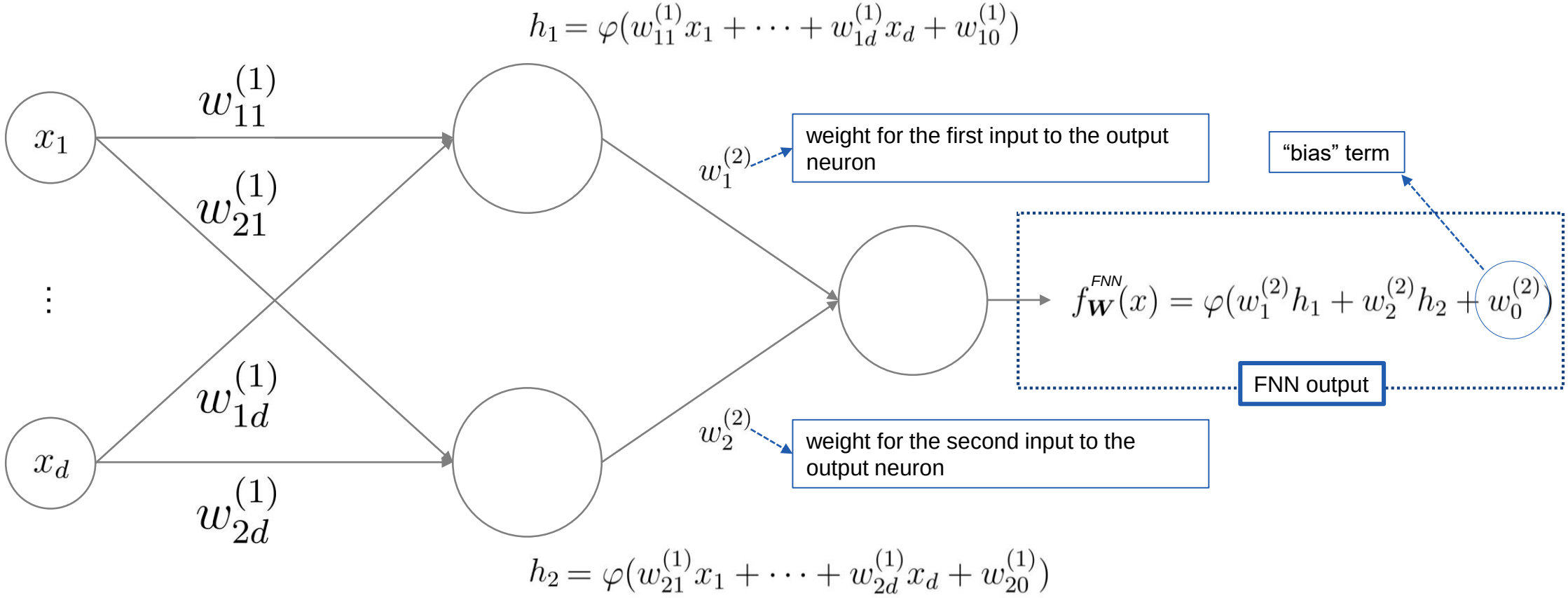
Let us introduce the function implemented by this FNN

We need to manage the different elements of the FNN by introducing some indices



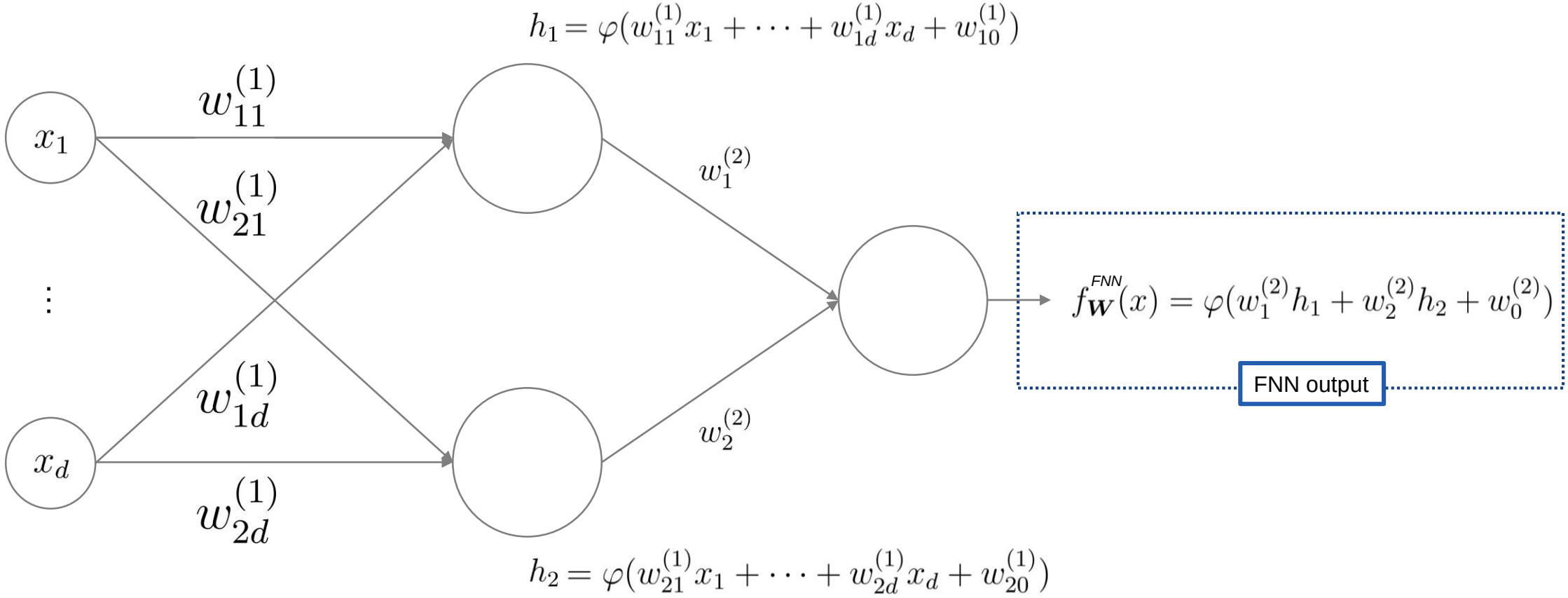
Let us introduce the function implemented by this FNN

We need to manage the different elements of the FNN by introducing some indices



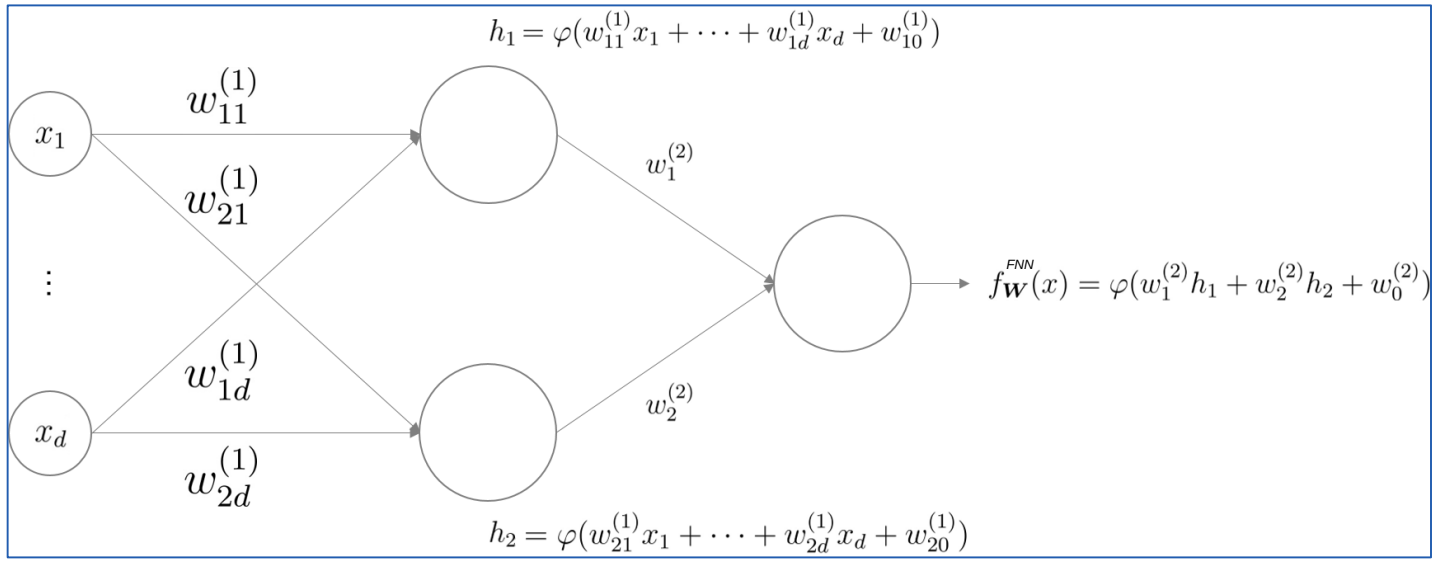
Let us introduce the function implemented by this FNN

Overview of our first deep learning model – one hidden layer FNN



Let us introduce the function implemented by this FNN

We summarize all steps using matrices



1

$$f_{\boldsymbol{w}}^{FNN}(x) = \varphi(W^{(2)} h + w_0^{(2)})$$

$$h = \varphi(W^{(1)} x + w_0^{(1)}) \in \mathbb{R}^2$$

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_d \end{pmatrix}$$

2

$$W^{(1)} = \begin{pmatrix} w_{11}^{(1)} & \cdots & w_{1d}^{(1)} \\ w_{21}^{(1)} & \cdots & w_{2d}^{(1)} \end{pmatrix}$$

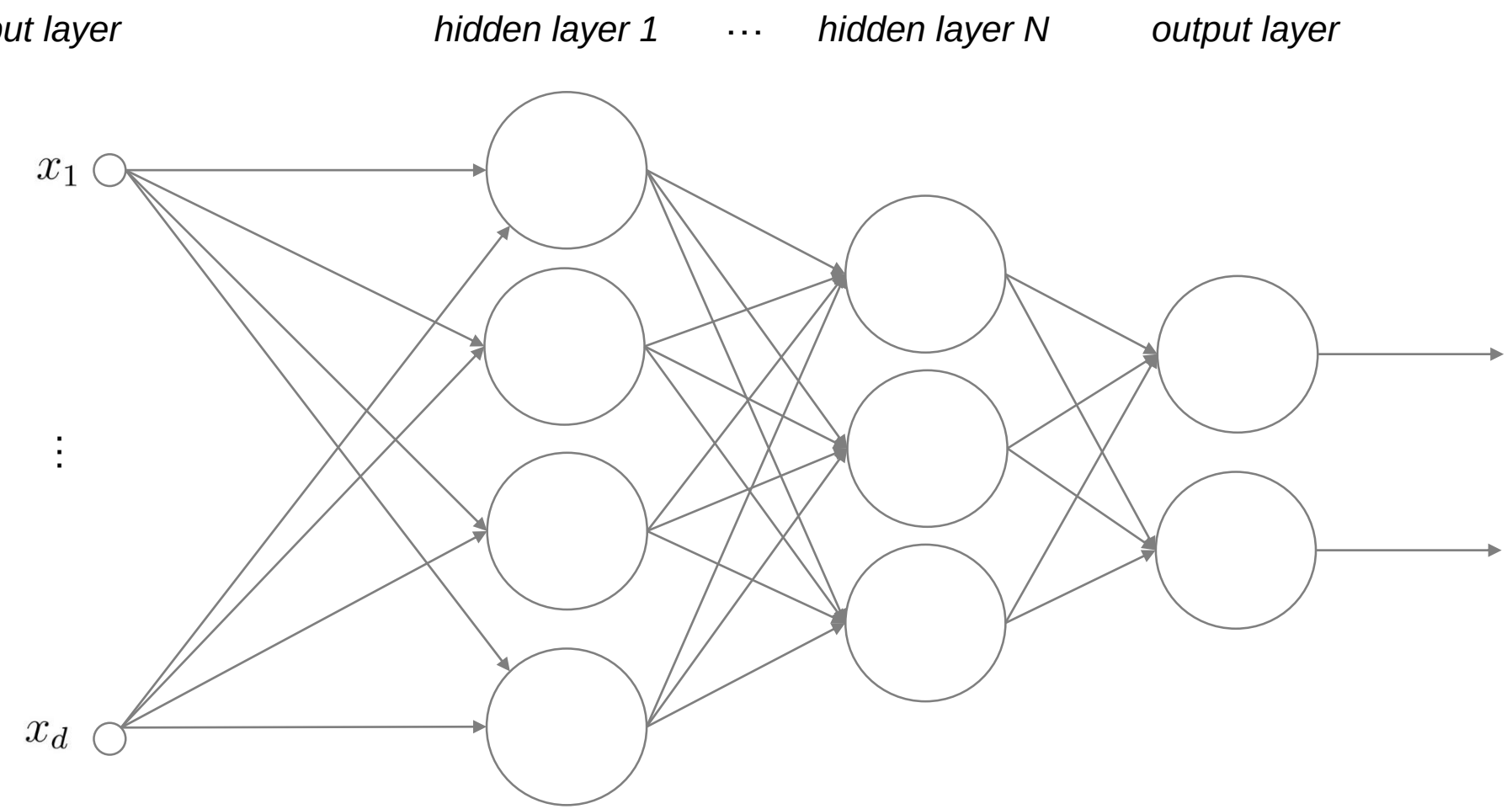
$$W^{(2)} = \begin{pmatrix} w_1^{(2)} & w_2^{(2)} \end{pmatrix}$$

$$w_0^{(1)} = \begin{pmatrix} w_{10}^{(1)} & w_{20}^{(1)} \end{pmatrix}^{\mathsf{T}}$$

$$\boldsymbol{W} = \{W^{(1)}, w_0^{(1)}, W^{(2)}, w_0^{(2)}\} \in \mathbb{R}^{2\times d+2+1\times 2+1}$$

We can add more hidden layers and have different outputs

FNNs are rather flexible



Let us create FNNs online

https://alexlenail.me/NN-SVG/index.html

FNNs: notation

1

$$f_{\boldsymbol{W}}^{FNN} = F^{(L)} \circ \varphi \odot F^{(L-1)} \circ \cdots \circ \varphi \odot F^{(1)}$$

$$F^{(k)} : \mathbb{R}^{N_{k-1}} \to \mathbb{R}^{N_k}, x \mapsto F^{(k)}(x) := W^{(k)} x + w_0^{(k)}$$

φ activation function

2

$$W^{(k)} \in \mathbb{R}^{N_k \times N_{k-1}}$$

$$w_0^k \in \mathbb{R}^{N_k}$$

$$\boldsymbol{W} = \{W^{(1)}, \ldots, W^{(L)}, w_0^{(1)}, \ldots, w_0^{(L)}\}$$

$$\sum_{k=1}^{L} (N_k N_{k-1} + N_k)$$

FNNs: notation

Hadamard product: the activation function is applied to each component of the vector resulting from the use of the layer-to-layer affine function

$f_{\boldsymbol{W}}^{FNN}$ is the composition of layer-to-layer affine functions and activation functions

1

$$f_{\boldsymbol{W}}^{FNN} = F^{(L)} \circ \varphi \odot F^{(L-1)} \circ \cdots \circ \varphi \odot F^{(1)}$$

layer-to-layer affine functions: they multiple the input to the *k*-th layer by weights and add biases

$$F^{(k)} : \mathbb{R}^{N_{k-1}} \to \mathbb{R}^{N_k}, x \mapsto F^{(k)}(x) := W^{(k)} x + w_0^{(k)}$$

φ activation function

2

matrices of weights

$$W^{(k)} \in \mathbb{R}^{N_k \times N_{k-1}}$$

$$w_0^k \in \mathbb{R}^{N_k}$$

biases (vectors)

Set of all parameters of the FNN

$$\boldsymbol{W} = \{W^{(1)}, \ldots, W^{(L)}, w_0^{(1)}, \ldots, w_0^{(L)}\}$$

$$\sum_{k=1}^{L} (N_k N_{k-1} + N_k)$$

Number of parameters of the FNN

Exercise together

1

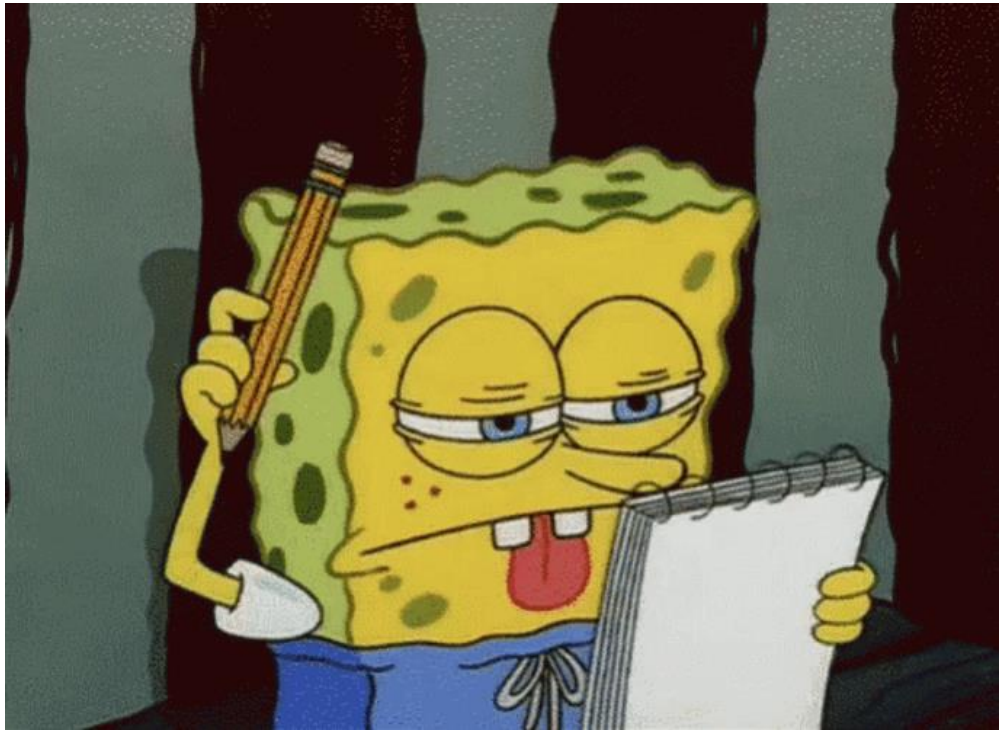
Draw an FNN on https://alexlenail.me/NN-SVG/index.html with

- 10 inputs
- Hidden layer 1: 8 neurons
- Hidden layer 2: 5 neurons
- Output layer: 2 neurons

2

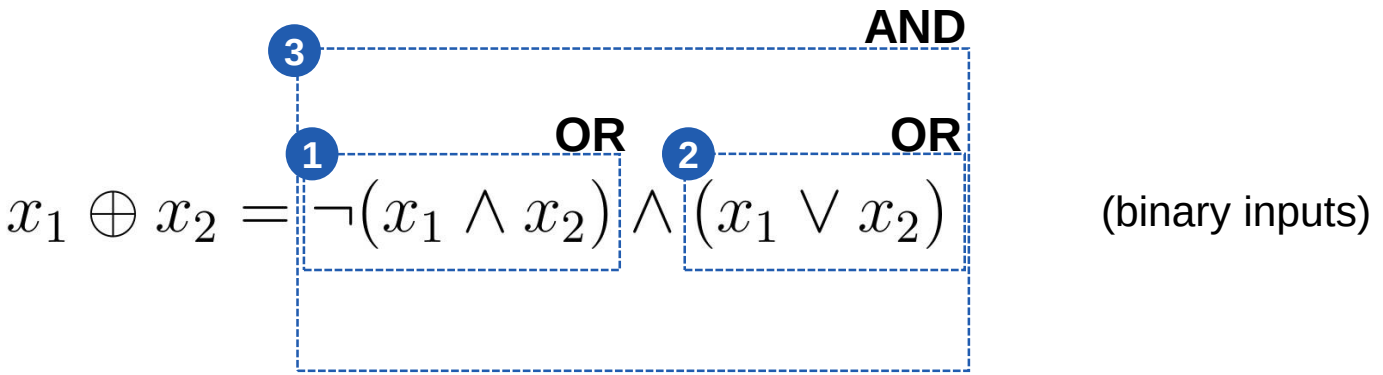
Write down $f_{\boldsymbol{W}}^{FNN}$, the set of parameters of the FNN, and compute their number

Part III: Why are FNNs useful?



Result 1: One hidden layer FNNs implement **XOR** ($\oplus$)

We have shown that perceptrons cannot implement **XOR** – can multilayer perceptrons do it?

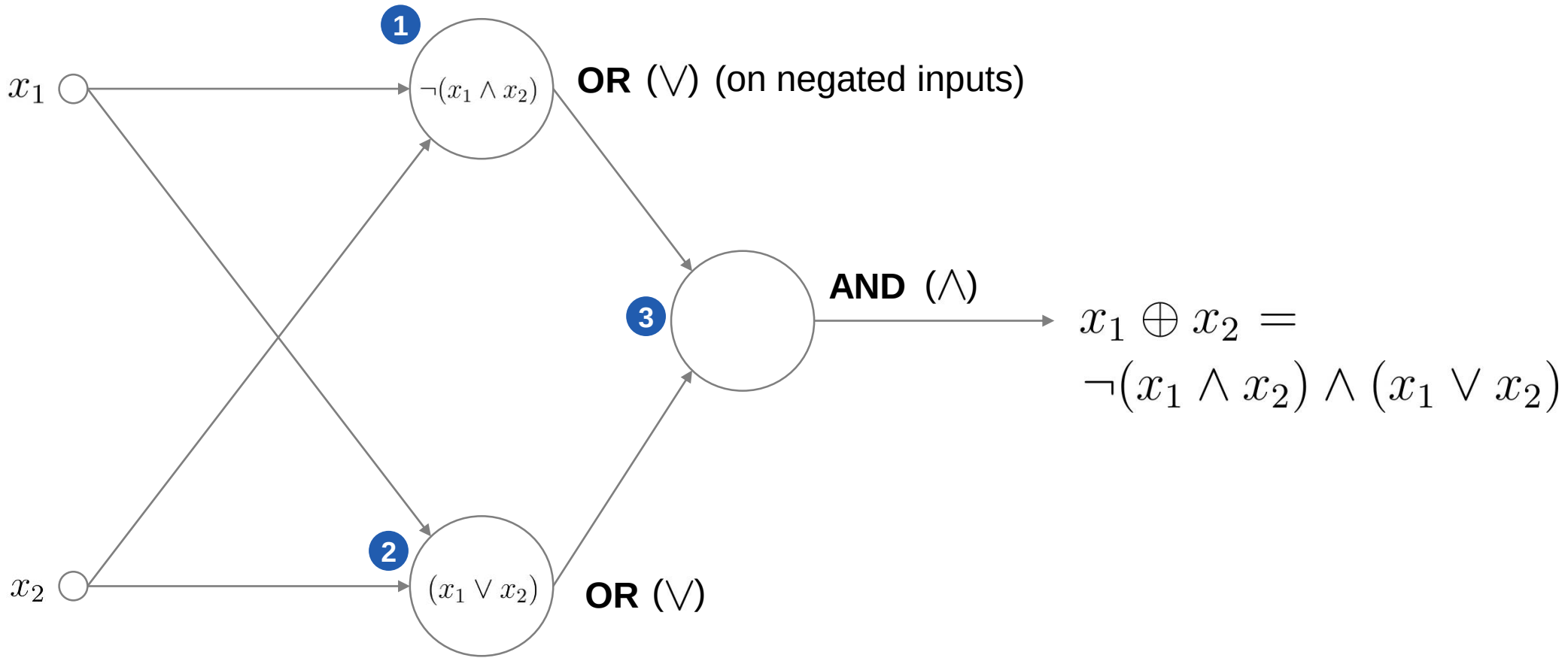


(binary inputs)

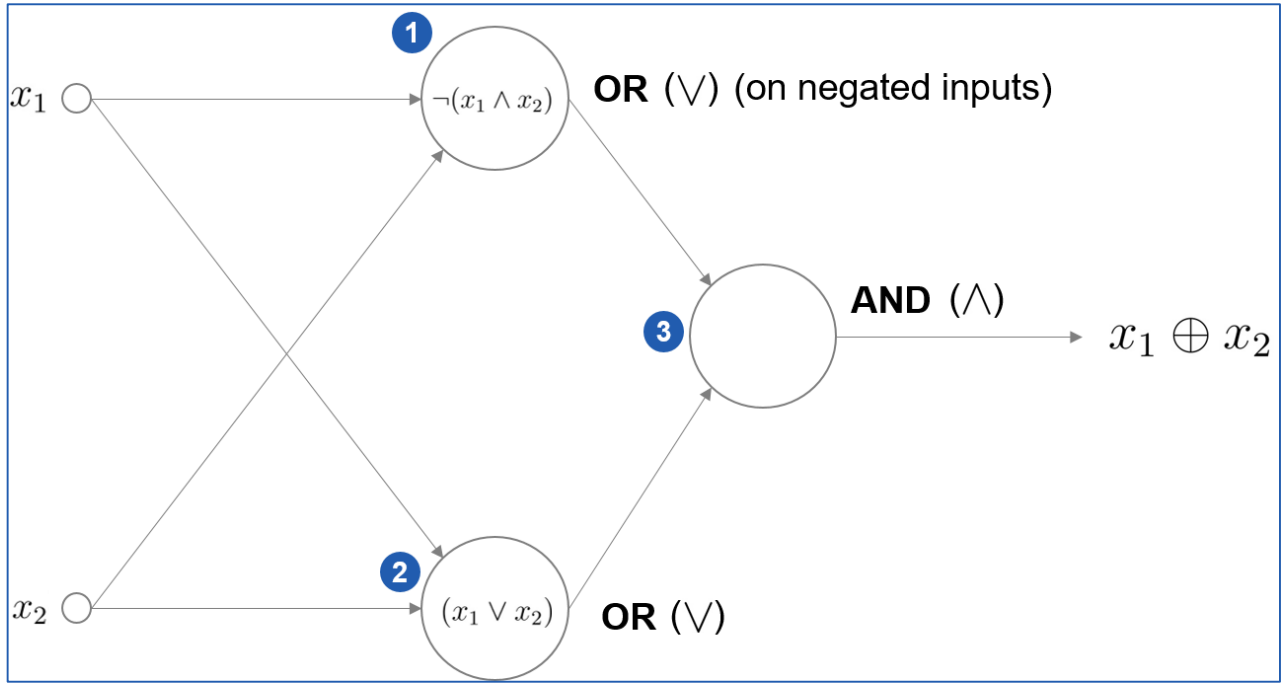
XOR comprises three binary operations…

Result 1: One hidden layer FNNs implement **XOR** ($\oplus$)

This multilayer perceptron implements **XOR**



Using the formalism so far, we can write down the function implemented by this multilayer perceptron



$$f_{\boldsymbol{W}}(x) = \varphi(W^{(2)}h + w_0^{(2)})$$

$$h = \varphi(W^{(1)}x + w_0^{(1)})$$

$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

φ is the Heaviside function

Can we find parameters $\boldsymbol{W}$ such that $f_{\boldsymbol{W}}(x) = x_1 \oplus x_2$ for all binary inputs x ?

A solution to the problem of implementing **XOR** with FNNs

Choose these **9** parameters/weights

$$W^{(1)} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \qquad W^{(2)} = \begin{pmatrix} 1 & -1 \end{pmatrix} \qquad w_0^{(1)} = \begin{pmatrix} -\frac{1}{2} & -\frac{3}{2} \end{pmatrix}^\mathsf{T} \qquad w_0^{(2)} = -\frac{1}{2}$$

$$f_{\boldsymbol{W}}(x) = \varphi(\varphi(x_1 + x_2 - \frac{1}{2}) - \varphi(x_1 + x_2 - \frac{3}{2}) - \frac{1}{2})$$

satisfies $f_{\boldsymbol{W}}(x) = x_1 \oplus x_2$ for all $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$

Result 2: One hidden layer FNNs can approximate functions

A renowned result by Cybenko

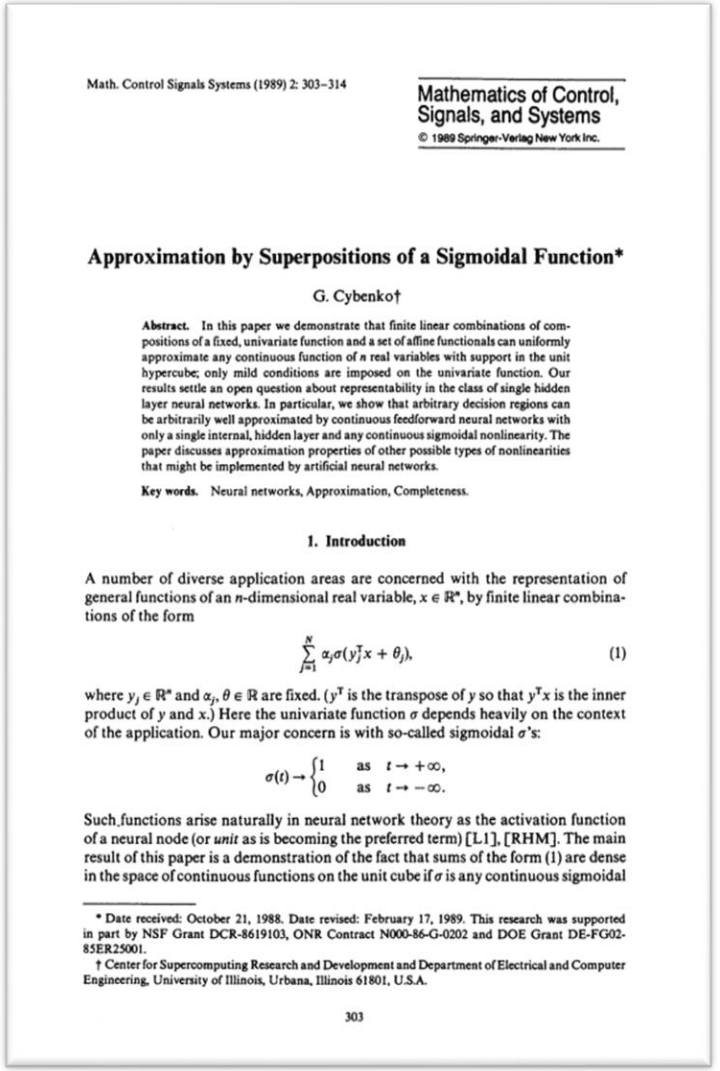
Cybenko (1989)

The functions

$$G(x) = W^{(2)}\sigma(W^{(1)}x + w_0^{(1)})$$
$$= \sum_{i=1}^{N} W_i^{(2)} \sum_{j=1}^{d} \sigma(W_{ij}^{(1)} x_j + w_{0,j}^{(1)})$$

approximate continuous functions on the compact subset

$$K = \underbrace{[0,1] \times \cdots \times [0,1]}_{d-\text{times}} \subset \mathbb{R}^d$$ arbitrarily well.



Math. Control Signals Systems (1989) 2: 303–314

Mathematics of Control, Signals, and Systems

© 1989 Springer-Verlag New York Inc.

Approximation by Superpositions of a Sigmoidal Function*

G. Cybenko†

Abstract. In this paper we demonstrate that finite linear combinations of compositions of a fixed, univariate function and a set of affine functionals can uniformly approximate any continuous function of n real variables with support in the unit hypercube; only mild conditions are imposed on the univariate function. Our results settle an open question about representability in the class of single hidden layer neural networks. In particular, we show that arbitrary decision regions can be arbitrarily well approximated by continuous feedforward neural networks with only a single internal, hidden layer and any continuous sigmoidal nonlinearity. The paper discusses approximation properties of other possible types of nonlinearities that might be implemented by artificial neural networks.

Key words. Neural networks, Approximation, Completeness.

1. Introduction

A number of diverse application areas are concerned with the representation of general functions of an n-dimensional real variable, $x \in \mathbb{R}^n$, by finite linear combinations of the form

$$\sum_{j=1}^{N} \alpha_j \sigma(y_j^T x + \theta_j), \quad (1)$$

where $y_j \in \mathbb{R}^n$ and $\alpha_j, \theta \in \mathbb{R}$ are fixed. (y^T is the transpose of y so that $y^T x$ is the inner product of y and x.) Here the univariate function σ depends heavily on the context of the application. Our major concern is with so-called sigmoidal σ's:

$$\sigma(t) \to \begin{cases} 1 & \text{as} \quad t \to +\infty, \\ 0 & \text{as} \quad t \to -\infty. \end{cases}$$

Such functions arise naturally in neural network theory as the activation function of a neural node (or *unit* as is becoming the preferred term) [L1], [RHM]. The main result of this paper is a demonstration of the fact that sums of the form (1) are dense in the space of continuous functions on the unit cube if σ is any continuous sigmoidal

* Date received: October 21, 1988. Date revised: February 17, 1989. This research was supported in part by NSF Grant DCR-8619103, ONR Contract N000-86-G-0202 and DOE Grant DE-FG02-85ER25001.

† Center for Supercomputing Research and Development and Department of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois 61801, U.S.A.

303

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, *2*(4), 303-314.

Result 2: One hidden layer FNNs can approximate functions

A renowned result by Cybenko

One hidden layer FNN with *d* inputs, *N* hidden neurons, and a linear neuron to compute the output

(appropriate) sigmoid function

Cybenko (1989)

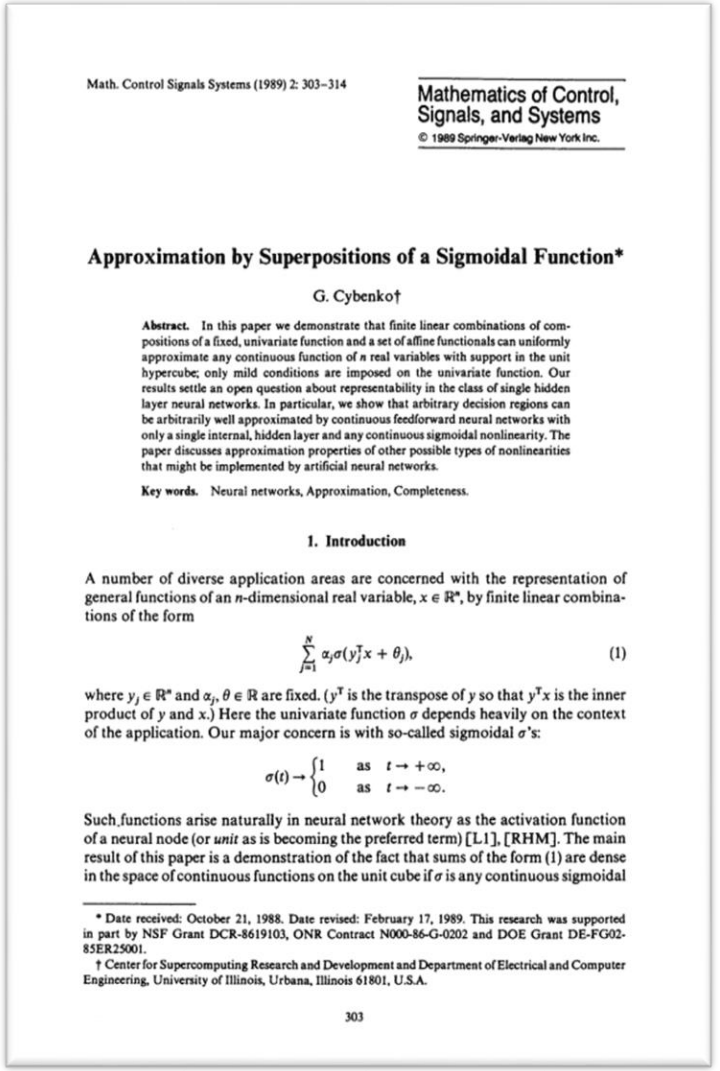
The functions

$$G(x) = W^{(2)}\sigma(W^{(1)}x + w_0^{(1)})$$
$$= \sum_{i=1}^{N} W_i^{(2)} \sum_{j=1}^{d} \sigma(W_{ij}^{(1)} x_j + w_{0,j}^{(1)})$$

approximate continuous functions on the compact subset

$$K = \underbrace{[0,1] \times \cdots \times [0,1]}_{d-\text{times}} \subset \mathbb{R}^d$$ arbitrarily well.

The functions are “dense” in the metric space of continuous functions on *K*



Math. Control Signals Systems (1989) 2: 303–314

Mathematics of Control, Signals, and Systems

© 1989 Springer-Verlag New York Inc.

Approximation by Superpositions of a Sigmoidal Function*

G. Cybenko†

Abstract. In this paper we demonstrate that finite linear combinations of compositions of a fixed, univariate function and a set of affine functionals can uniformly approximate any continuous function of n real variables with support in the unit hypercube; only mild conditions are imposed on the univariate function. Our results settle an open question about representability in the class of single hidden layer neural networks. In particular, we show that arbitrary decision regions can be arbitrarily well approximated by continuous feedforward neural networks with only a single internal, hidden layer and any continuous sigmoidal nonlinearity. The paper discusses approximation properties of other possible types of nonlinearities that might be implemented by artificial neural networks.

Key words. Neural networks, Approximation, Completeness.

1. Introduction

A number of diverse application areas are concerned with the representation of general functions of an n-dimensional real variable, $x \in \mathbb{R}^n$, by finite linear combinations of the form

$$\sum_{j=1}^{N} \alpha_j \sigma(y_j^T x + \theta_j), \quad (1)$$

where $y_j \in \mathbb{R}^n$ and $\alpha_j, \theta \in \mathbb{R}$ are fixed. (y^T is the transpose of y so that $y^T x$ is the inner product of y and x.) Here the univariate function σ depends heavily on the context of the application. Our major concern is with so-called sigmoidal σ's:

$$\sigma(t) \to \begin{cases} 1 & \text{as} \quad t \to +\infty, \\ 0 & \text{as} \quad t \to -\infty. \end{cases}$$

Such functions arise naturally in neural network theory as the activation function of a neural node (or *unit* as is becoming the preferred term) [L1], [RHM]. The main result of this paper is a demonstration of the fact that sums of the form (1) are dense in the space of continuous functions on the unit cube if σ is any continuous sigmoidal

* Date received: October 21, 1988. Date revised: February 17, 1989. This research was supported in part by NSF Grant DCR-8619103, ONR Contract N000-86-G-0202 and DOE Grant DE-FG02-85ER25001.

† Center for Supercomputing Research and Development and Department of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois 61801, U.S.A.

303

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, *2*(4), 303-314.

Google Colab: `abs_neurons.ipynb`

Self-Study: Closing with (designing) neurons and FNNs

Self-Study

From Moodle download the book:

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT press.

- **Chapter 6** Deep Feedforward Networks
- **Section 6.1** Example: Learning XOR

Feedback! See you on April 12th