ENSAE

Deep learning

Unpacking Transformers, LLMs and image generation

Session 1

About us: Olivier Koch

onfido

VP of AI at Onfido

ex-Criteo (2015-2020)

MIT DARPA Urban Challenge Finalist

ENSTA 2002, MIT PhD 2010 (computer vision)

Published @ CVPR 2007, IJFR 2008, ICCV 2009, ICRA 2010

oakfr

@olivkoch

olivierkoch.net

About us: Remi Verronneau

onfido

Agenda for today

1. What you will learn
2. Organization / setup
3. A brief history of deep learning
4. Practical #1: a first neural network

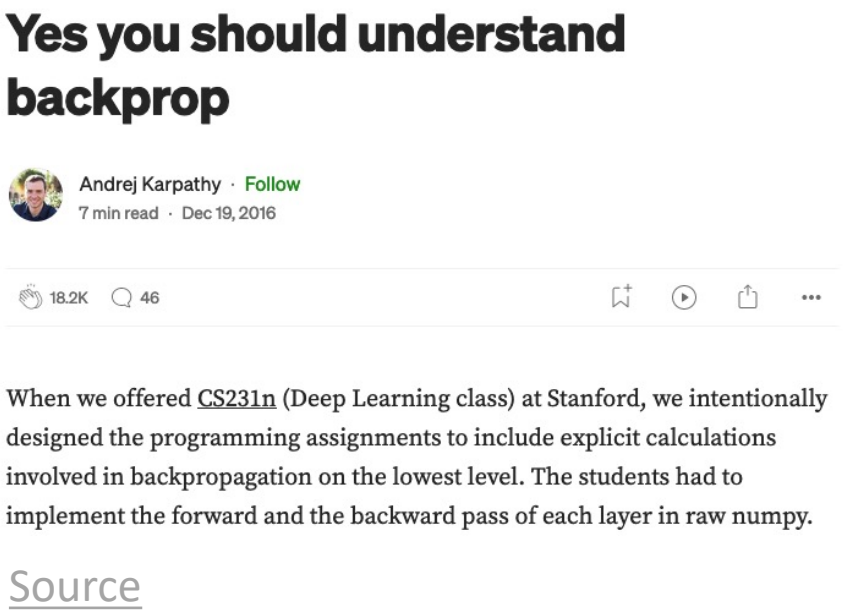
What you will learn

- Concepts and practical aspects of deep learning
- Build your own backprop, mini-gpt and diffusion models

This course is not a complete overview of all ML & deep learning techniques.

Why learning from scratch matters

1. You only understand it if you build it
2. Backprop is a « leaky abstraction »



Yes you should understand backprop

Andrej Karpathy · Follow
7 min read · Dec 19, 2016

18.2K 46

When we offered CS231n (Deep Learning class) at Stanford, we intentionally designed the programming assignments to include explicit calculations involved in backpropagation on the lowest level. The students had to implement the forward and the backward pass of each layer in raw numpy.

Source

Organization / Setup

- 6 sessions
- 1.5h theory + 1.5h practice

Grading: 6 notebooks (66%) + 1 quiz (33%)

I expect you to return your notebook at the end of each session.

You can then send an updated version over the following week to aim for a better grade (and learn more).

Rules of engagement

It’s OK to ask for help and share tips (use slack)

Each notebook should be yours

You must return your notebook as-is at the end of each TP (via slack)

You can then send an updated version within a week

Please do not use email except for emergencies

References

The theory:

Understanding Deep Learning, Simon Prince, December 2023

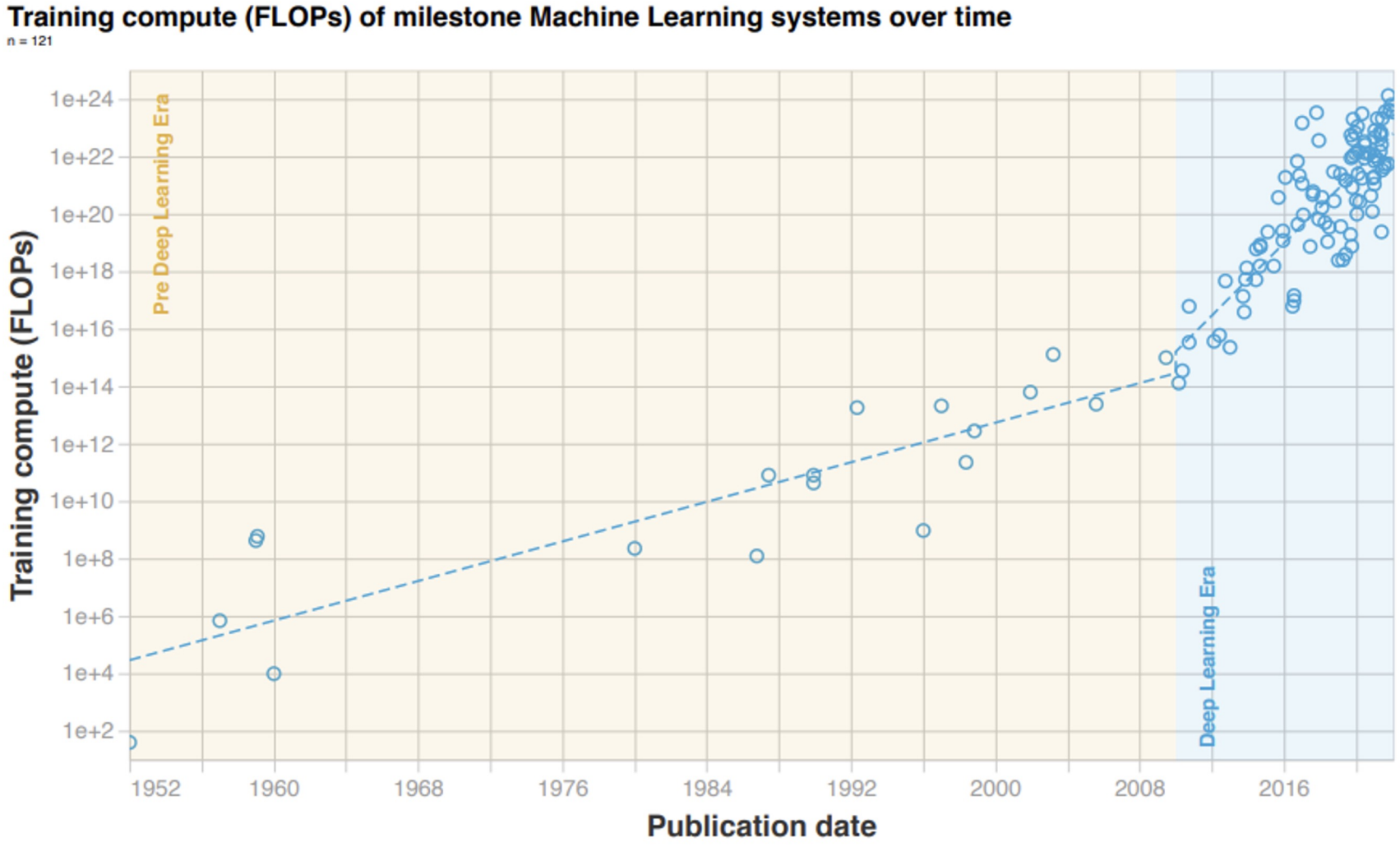
Deep Learning: Foundations and Concepts, Christopher and Hugh Bishop, 2023

The Practice:

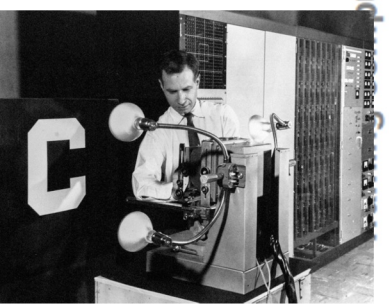
Neural networks: Zero to Hero, Andrej Karpathy, Youtube Lecture Series, 2022

Deep Learning Course, François Fleuret, 2018-2022

1. Historical perspective
2. Neural networks 101

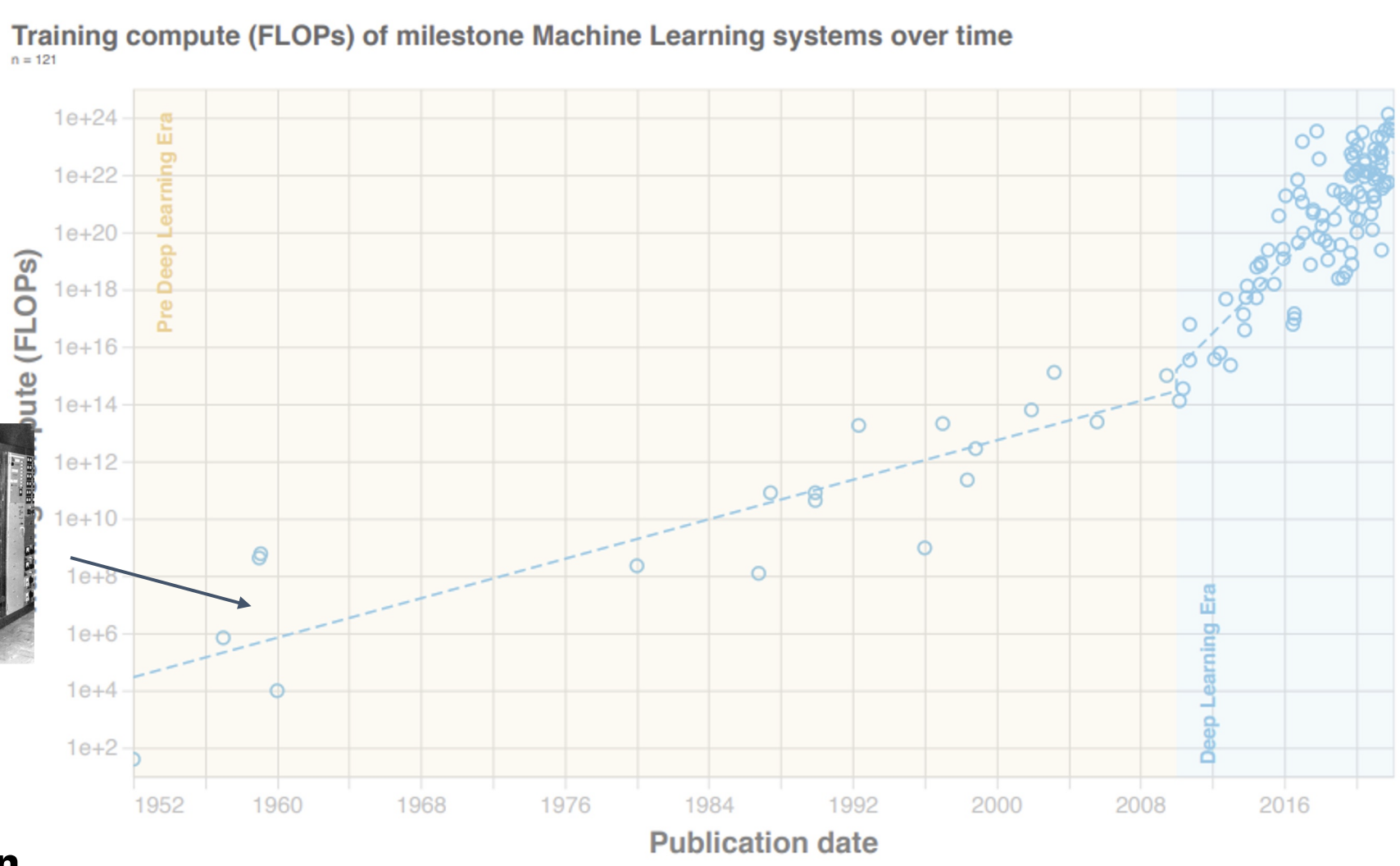

Training compute (FLOPs) of milestone Machine Learning systems over time
n = 121
Training compute (FLOPs)
1e+24
1e+22
1e+20
1e+18
1e+16
1e+14
1e+12
1e+10
1e+8
1e+6
1e+4
1e+2
Pre Deep Learning Era
Deep Learning Era
1952
1960
1968
1976
1984
1992
2000
2008
2016
Publication date

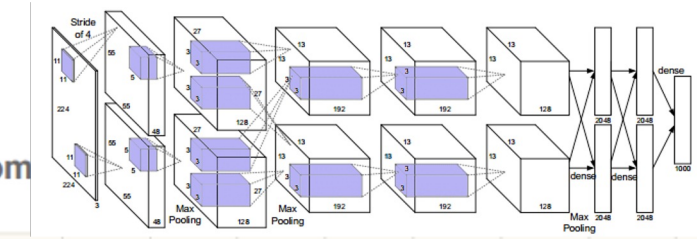
Source



1958

The Perceptron





Training com ng systems over time

n = 121

2012

AlexNet

Training compute (FLOPs)

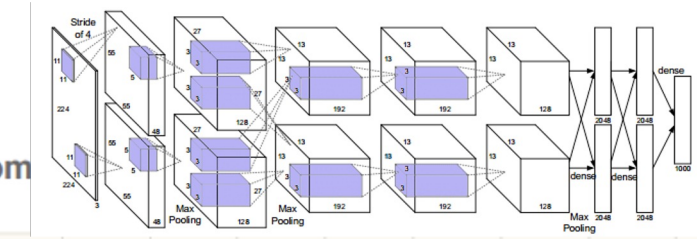
1e+24
1e+22
1e+20
1e+18
1e+16
1e+14
1e+12
1e+10
1e+8
1e+6
1e+4
1e+2

Pre Deep Learning Era

Deep Learning Era

1952 1960 1968 1976 1984 1992 2000 2008 2016

Publication date

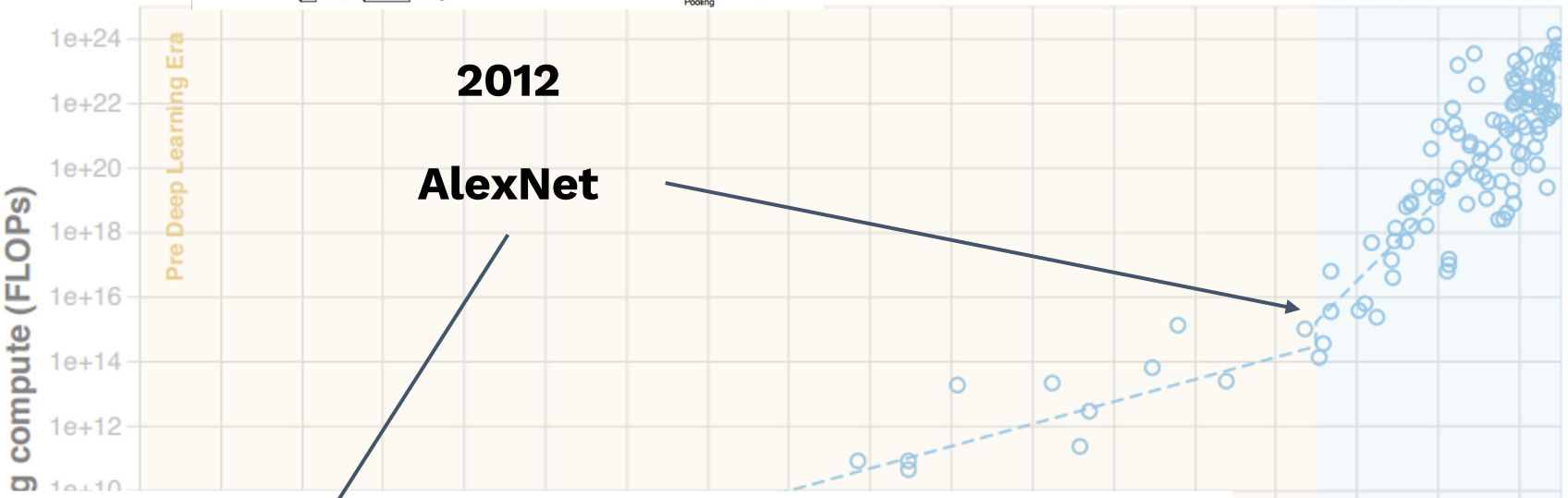


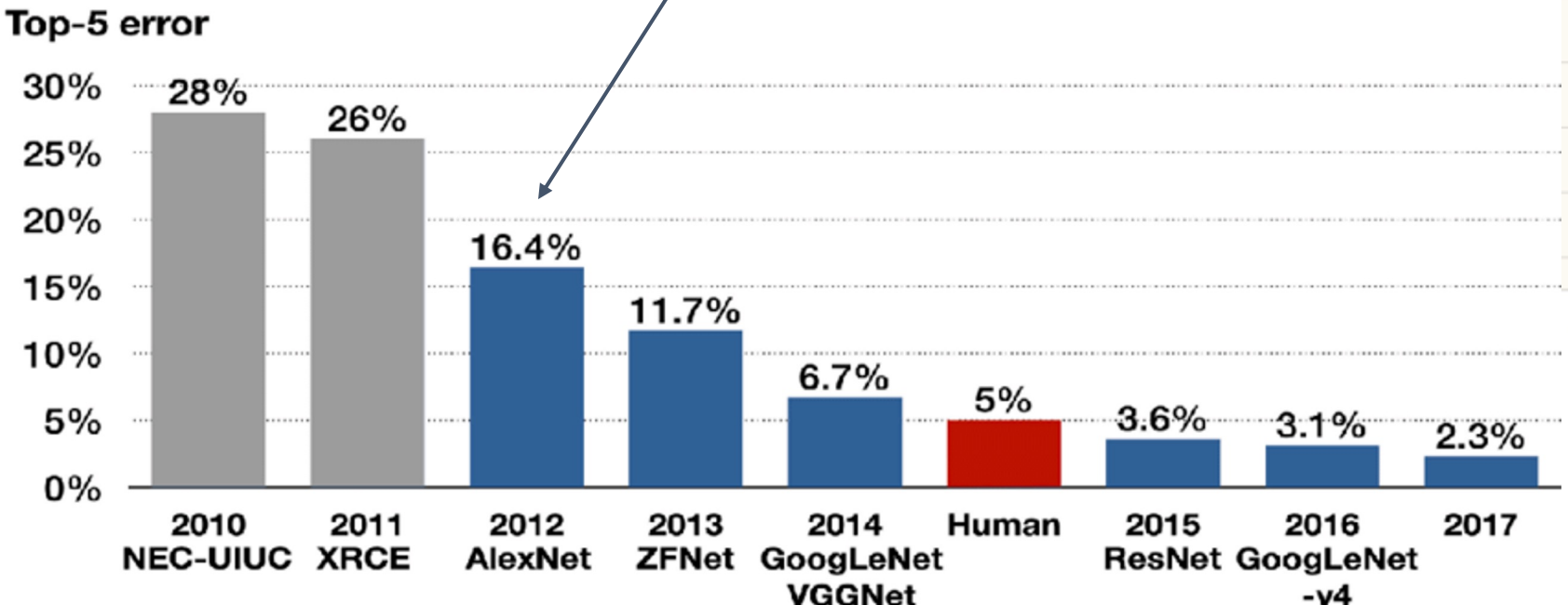
Training com ng systems over time

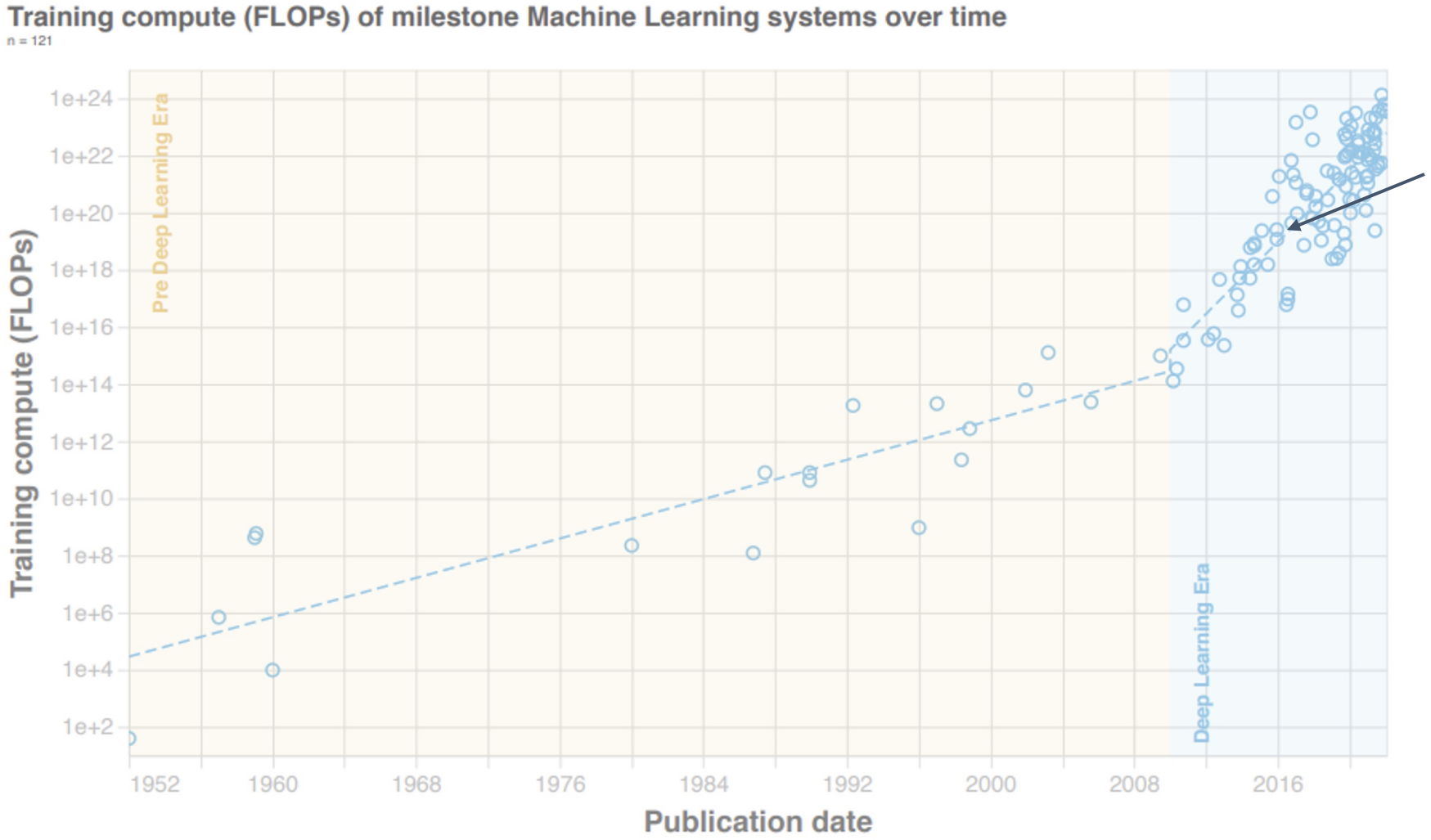
n = 121

2012

AlexNet



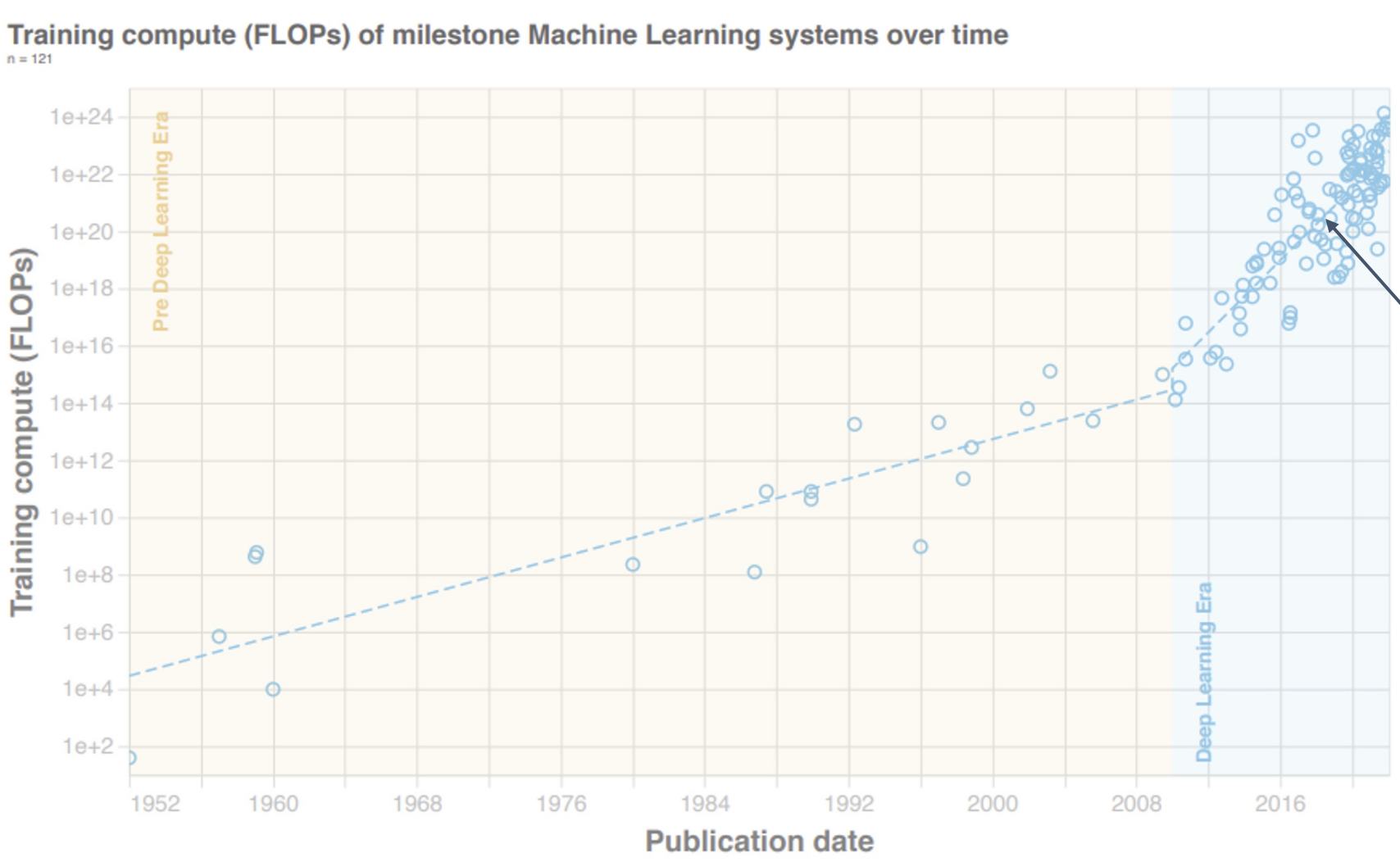


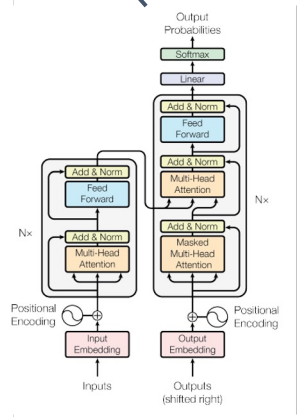


2016

AlphaGo

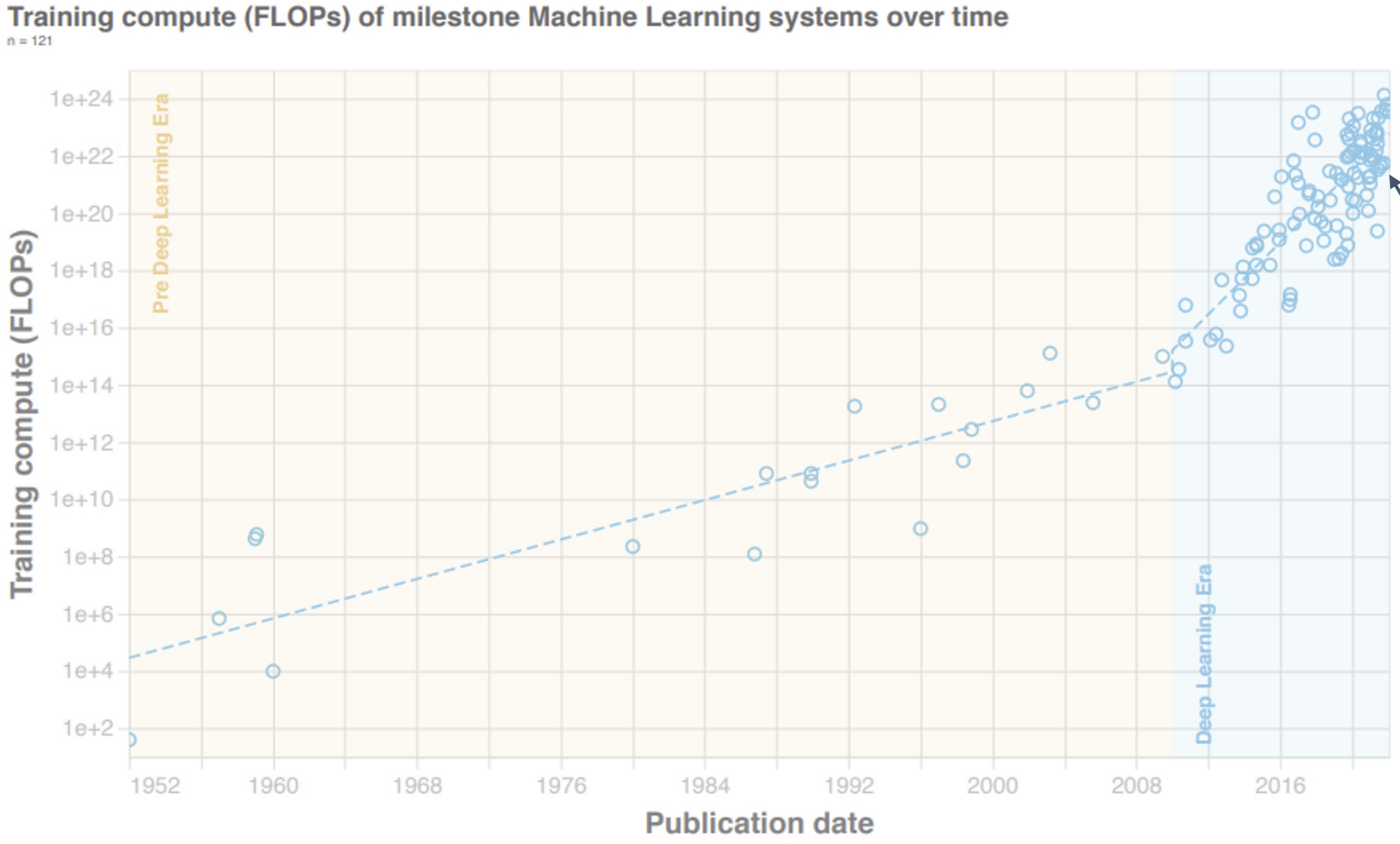
Reinforcement Learning (RL)





2017

Transformers

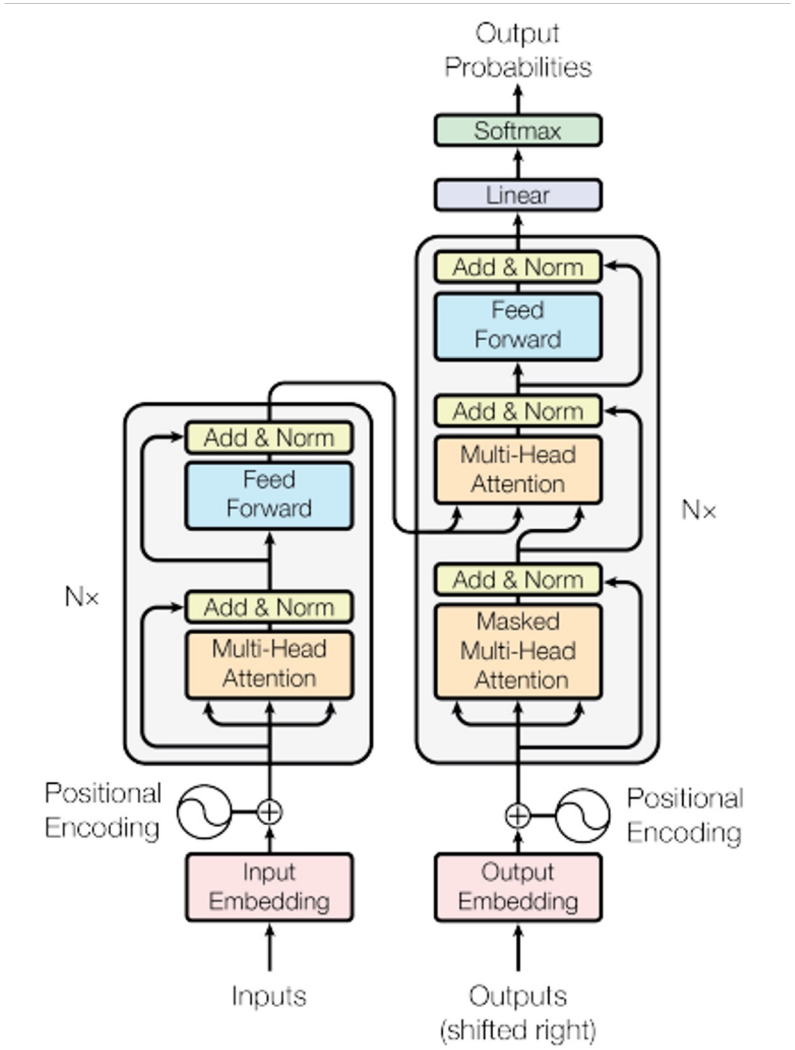

Training compute (FLOPs) of milestone Machine Learning systems over time
n = 121
Training compute (FLOPs)
1e+24
1e+22
1e+20
1e+18
1e+16
1e+14
1e+12
1e+10
1e+8
1e+6
1e+4
1e+2
Pre Deep Learning Era
Deep Learning Era
1952
1960
1968
1976
1984
1992
2000
2008
2016
Publication date

2023

GPT

The Transformer shake-up

What is a Transformer?



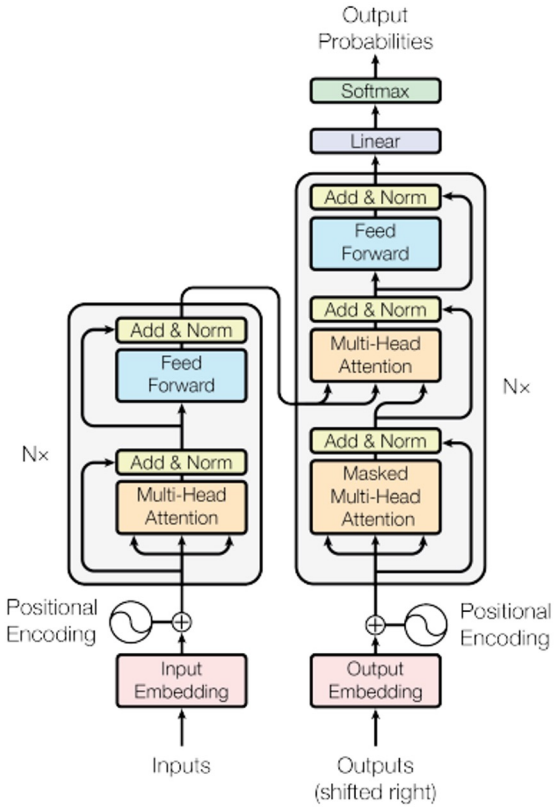
No, seriously… what is a Transformer?

We’ll spend a lot of time on this during the course.

Attention is a communication mechanism between tokens in a document, with a positional embedding allowing to incorporate ordering into sets of tokens.

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}})\mathbf{V}$$

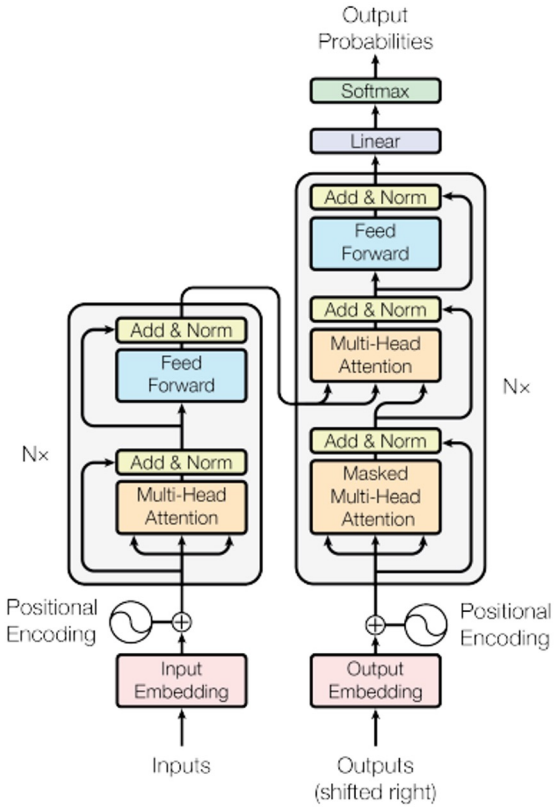
Transformers shake up translation



Abraham Lincoln (February 12, 1809 – April 15, 1865) was an American lawyer, politician and statesman who served as the 16th president of the United States from 1861 until his assassination in 1865. Lincoln led the Union through the American Civil War to defend the nation as a constitutional union and succeeded in abolishing slavery,...

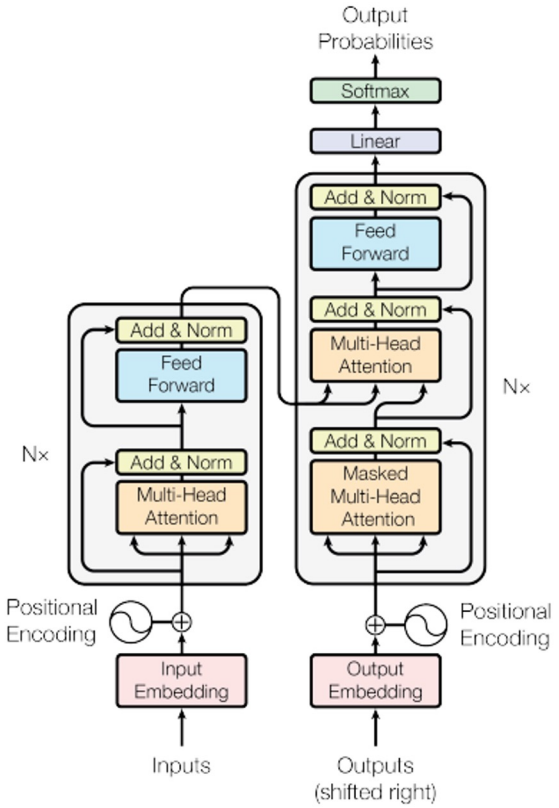
Abraham Lincoln (12. Februar 1809 – 15. April 1865) war ein amerikanischer Anwalt, Politiker und Staatsmann, der von 1861 bis zu seiner Ermordung im Jahr 1865 als 16. Präsident der Vereinigten Staaten diente. Lincoln führte die Union durch den Amerikanischen Bürgerkrieg, um die Nation als verfassungsmäßige Union zu verteidigen, und war erfolgreich bei der Abschaffung der Sklaverei.

Source



Abraham Lincoln (February 12, 1809 – April 15, 1865) was an American lawyer, politician and statesman who served as the 16th president of the United States from 1861 until his assassination in 1865. Lincoln led the Union through the American Civil War to defend the nation as a constitutional union and succeeded in abolishing ____ _____,...

Likelihood of “slavery”: 99.2%
Likelihood of “making”: 0.1%
Likelihood of “food”: 3%



Context length

Abraham Lincoln (February 12, 1809 – April 15, 1865) was an American lawyer, politician and statesman who served as the 16th president of the United States from 1861 until his assassination in 1865. Lincoln led the Union through the American Civil War to defend the nation as a constitutional union and succeeded in abolishing ____ _____,...

Likelihood of “slavery”: 99.2%
Likelihood of “making”: 0.1%
Likelihood of “food”: 3%

Transformers for text generation

Language models are few-shot learners

Try it yourself!

Write With Transformer

Get a modern neural network to
auto-complete your thoughts.

This web app, built by the Hugging Face team, is the official demo of the
`🤗/transformers` repository's text generation capabilities.

Star 93,671

https://transformer.huggingface.co/

Christopher Columbus discovered America in 1492. When he arrived, he saw a land full of unknown natives that would eventually become the United States . And while it's true that this discovery is one of the most famous events in our history, he didn't come across all the people he would meet , nor all the artifacts he would find .

OK, this is cool, but it hallucinates 100%.

The ChatGPT epiphany

But we know how to train a machine to learn from feedback!

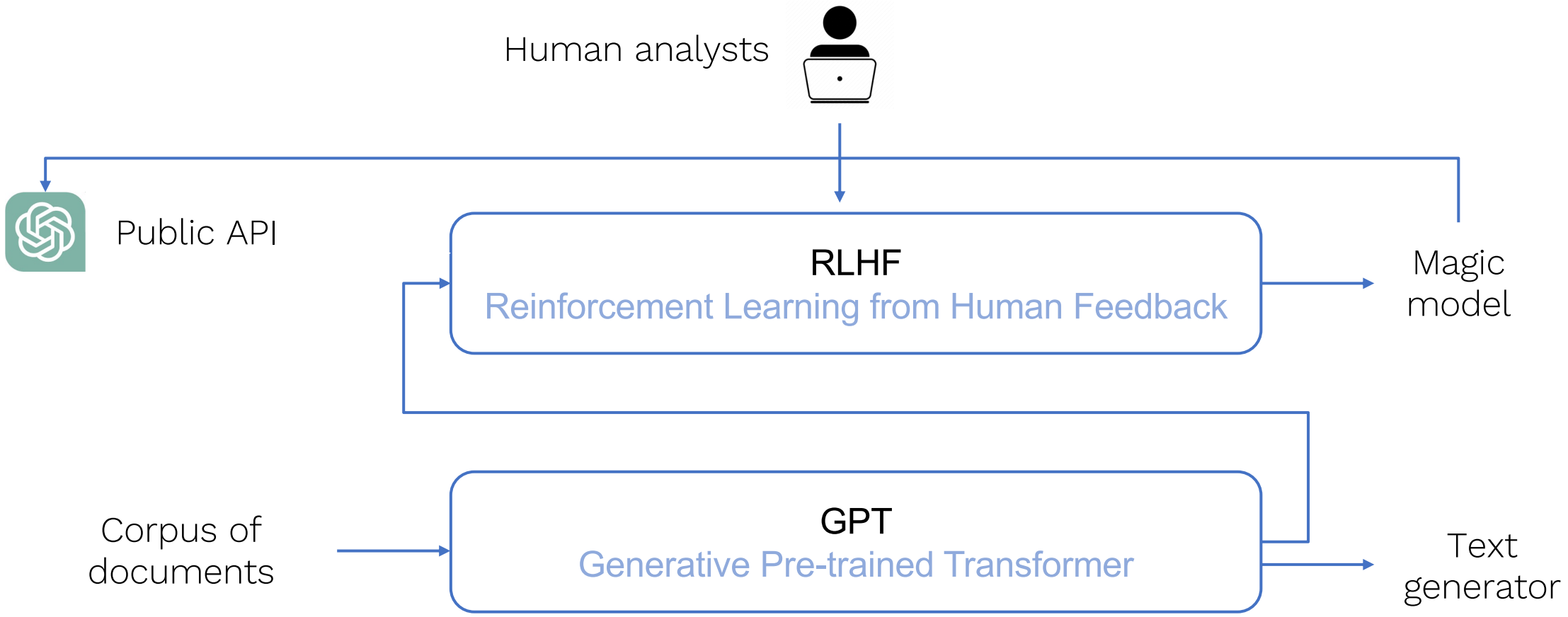
It is called Reinforcement Learning (RL).

Go World Champion Lee Sedol looking confused at move 37 by AlphaGo...

The ChatGPT epiphany

"This whole thing was just an experiment. We had no idea it would work so well", Ilya Sutskever
(full interview)

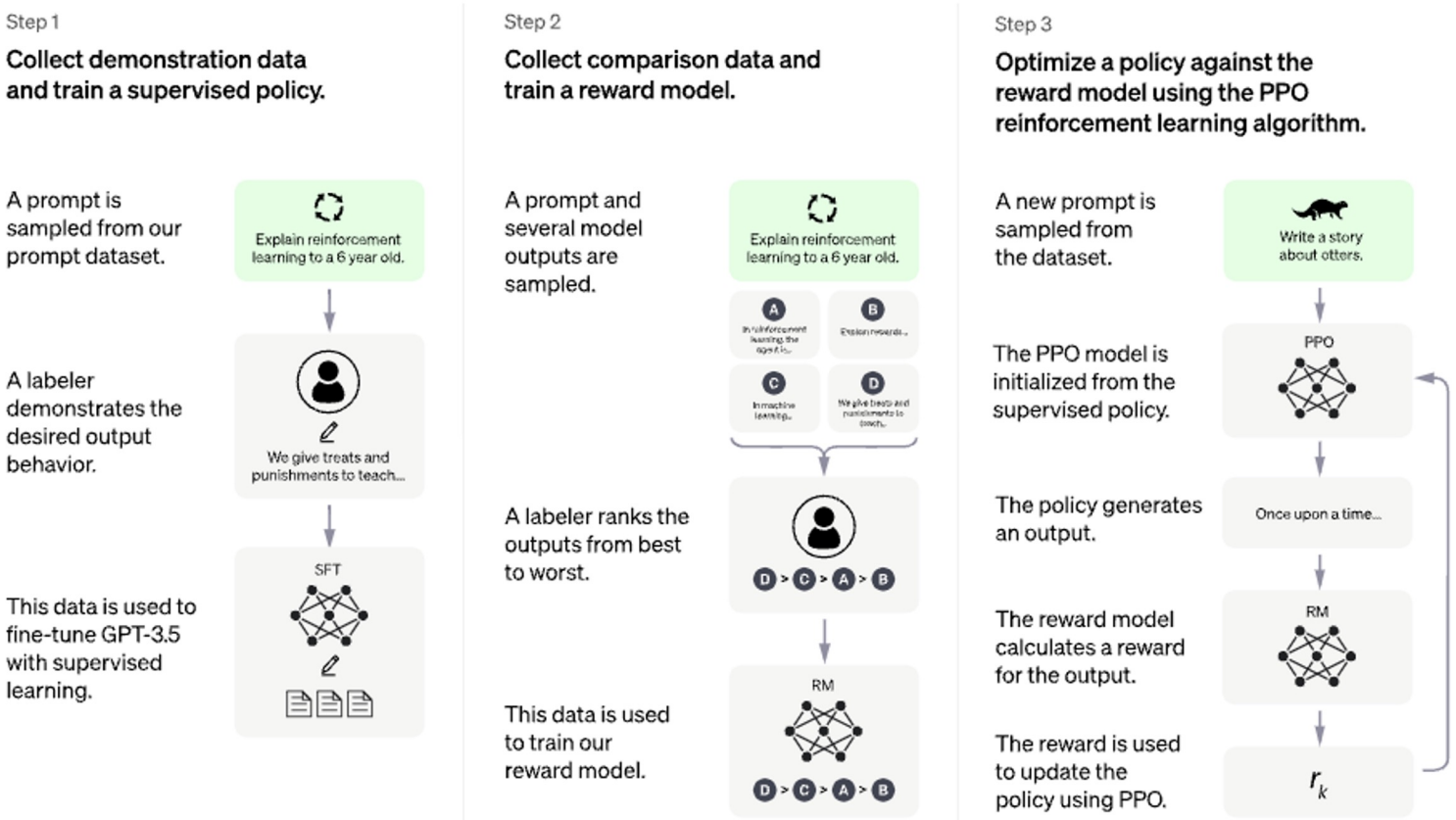
The main breakthrough of LLM is the discovery that training a very large model on the simplest task (predict the next token) could yield remarkable *apparent* intelligence.


Human analysts
Public API
RLHF
Reinforcement Learning from Human Feedback
Magic model
Corpus of documents
GPT
Generative Pre-trained Transformer
Text generator

Midjourney

“Man hoarding tigers with a whip, black and white drawing”

The science of human feedback (alignment)



[Source]

Current state of AI

The Cambrian explosion of foundational models

Mistral AI

Experimentation is getting much easier

2010

Write your own backprop manually

2018

`loss.backward()`

2023

```
from transformers
import SwinModel
```

… but research is far from over

Durk Kingma @dpkingma · Jun 9

Why do LLMs hallucinate? TL;DR:
- LLM pretraining generally results in a well-calibrated distribution p(completion|prompt)
- Obviously, a single sample from this distribution doesn't express the uncertainty contained in it
- This is fixable, e.g. through sufficient RLHF/RLAIF

23 36 260 82.4K

Yann LeCun @ylecun · Jun 10

Replying to @dpkingma

I disagree. I don't think it's fixable within the auto-regressive prediction paradigm.

15 6 143 21.2K

1. Historical perspective
2. Shallow networks, deep networks

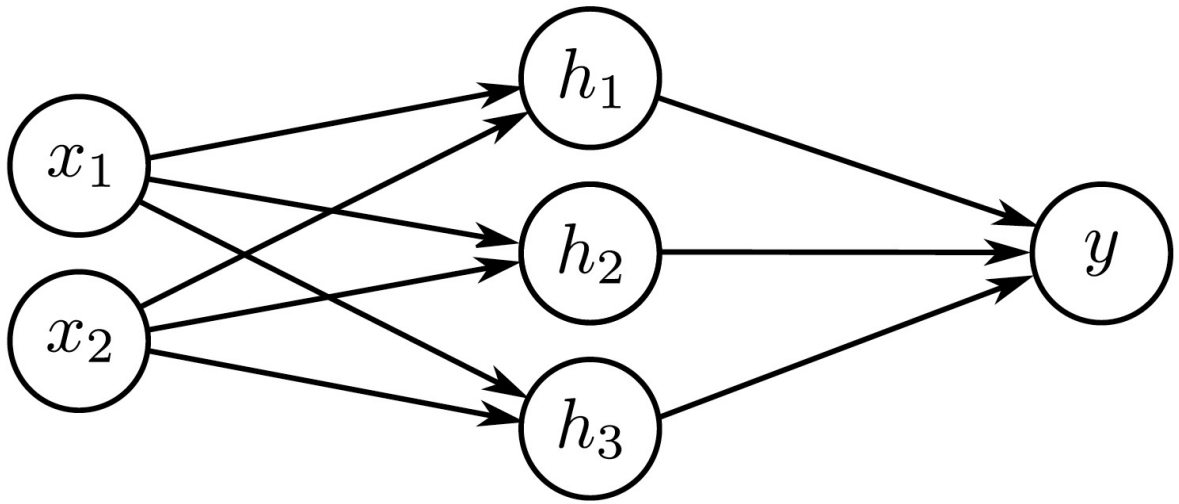


Figure 3.7 Visualization of neural network with 2D multivariate input $\mathbf{x} = [x_1, x_2]^T$ and scalar output y.

Source: Understanding Deep Learning, S. Prince, 2023

This work is subject to a Creative Commons CC-BY-NC-ND license. (C) MIT Press.

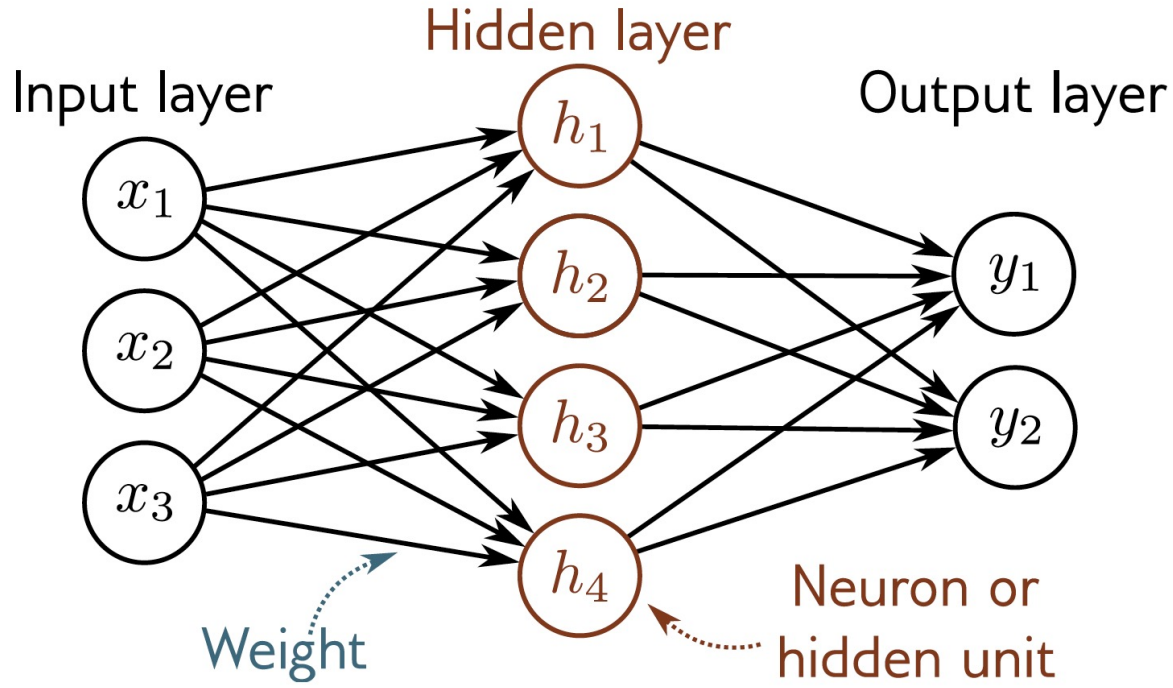


Figure 3.12 Terminology. A shallow network consists of an input layer, a hidden layer, and an output layer. Each layer is connected to the next by forward connections (arrows). For this reason, these models are referred to as feed-forward networks. When every variable in one layer connects to every variable in the next, we call this a fully connected network. Each connection represents a slope parameter in the underlying equation, and these parameters are termed weights. The variables in the hidden layer are termed neurons or hidden units. The values feeding into the hidden units are termed pre-activations, and the values at the hidden units (i.e., after the ReLU function is applied) are termed activations.

Source: Understanding Deep Learning, S. Prince, 2023

This work is subject to a Creative Commons CC-BY-NC-ND license. (C) MIT Press.

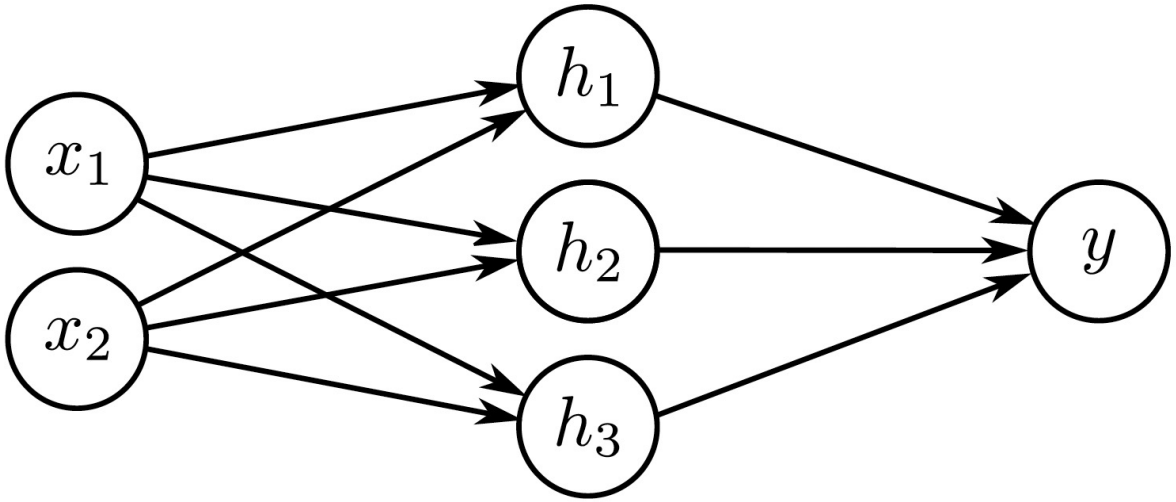


Figure 3.7 Visualization of neural network with 2D multivariate input $\mathbf{x} = [x_1, x_2]^T$ and scalar output y.

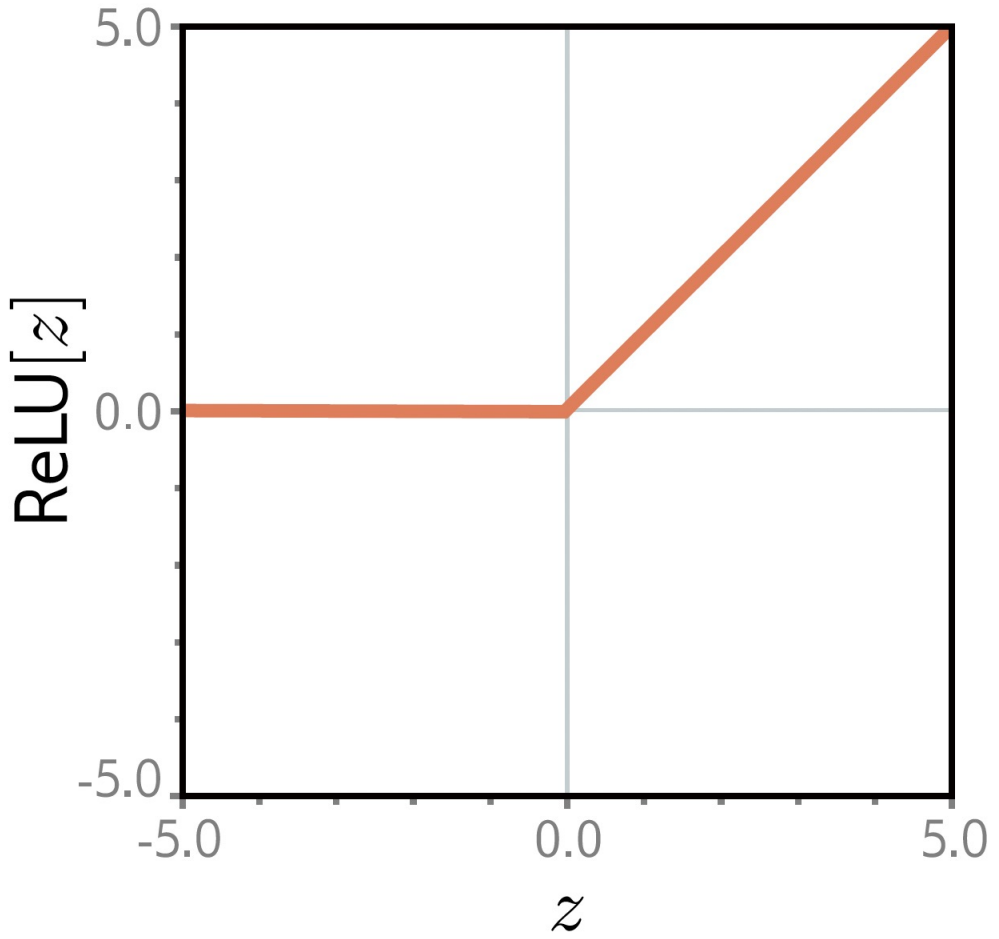
$$h_d = \mathrm{a}\left[\theta_{d0} + \sum_{i=1}^{D_i} \theta_{di} x_i\right], \tag{3.11}$$

$$y_j = \phi_{j0} + \sum_{d=1}^{D} \phi_{jd} h_d, \tag{3.12}$$

Source: Understanding Deep Learning, S. Prince, 2023

This work is subject to a Creative Commons CC-BY-NC-ND license. (C) MIT Press.

Figure 3.1 Rectified linear unit (ReLU). This activation function returns zero if the input is less than zero and returns the input unchanged otherwise. In other words, it clips negative values to zero. Note that there are many other possible choices for the activation function (see figure 3.13), but the ReLU is the most commonly used and the easiest to understand.



Source: Understanding Deep Learning, S. Prince, 2023

This work is subject to a Creative Commons CC-BY-NC-ND license. (C) MIT Press.

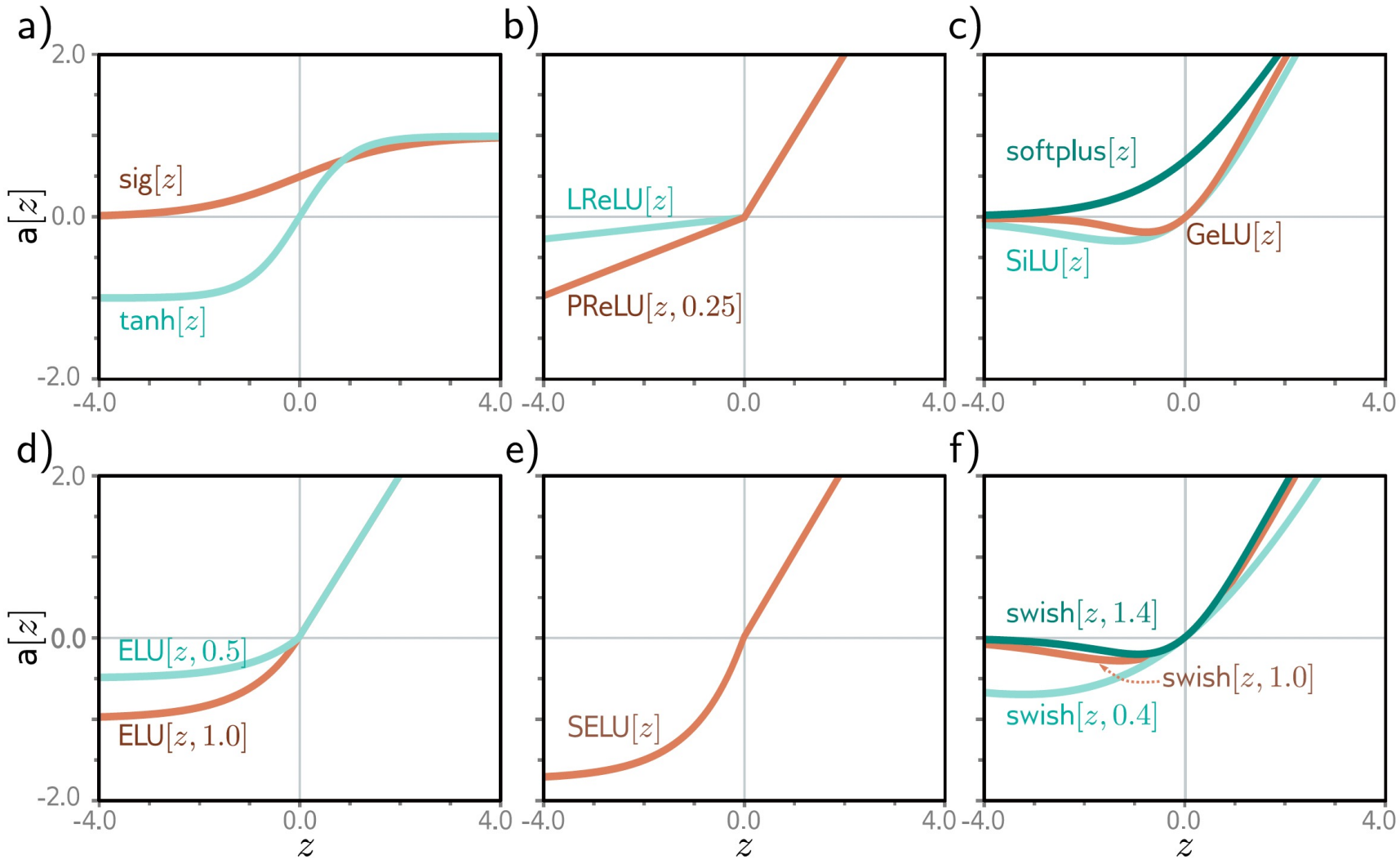


Figure 3.13 Activation functions. a) Logistic sigmoid and tanh functions. b) Leaky ReLU and parametric ReLU with parameter 0.25. c) SoftPlus, Gaussian error linear unit, and sigmoid linear unit. d) Exponential linear unit with parameters 0.5 and 1.0, e) Scaled exponential linear unit. f) Swish with parameters 0.4, 1.0, and 1.4.

Source: Understanding Deep Learning, S. Prince, 2023

This work is subject to a Creative Commons CC-BY-NC-ND license. (C) MIT Press.

Sigmoid: not blowing up activation

ReLU : not vanishing gradient

ReLU : More computationally efficient to compute than Sigmoid like functions since ReLU just needs to pick max(0,x) and not perform expensive exponential operations as in sigmoids

ReLU : In practice, networks with ReLU tend to show better convergence performance than sigmoid. (Krizhevsky et al.)

ImageNet Classification with Deep Convolutional
Neural Networks, Krizhevsky, Sutskever and Hinton, NIPS 2012

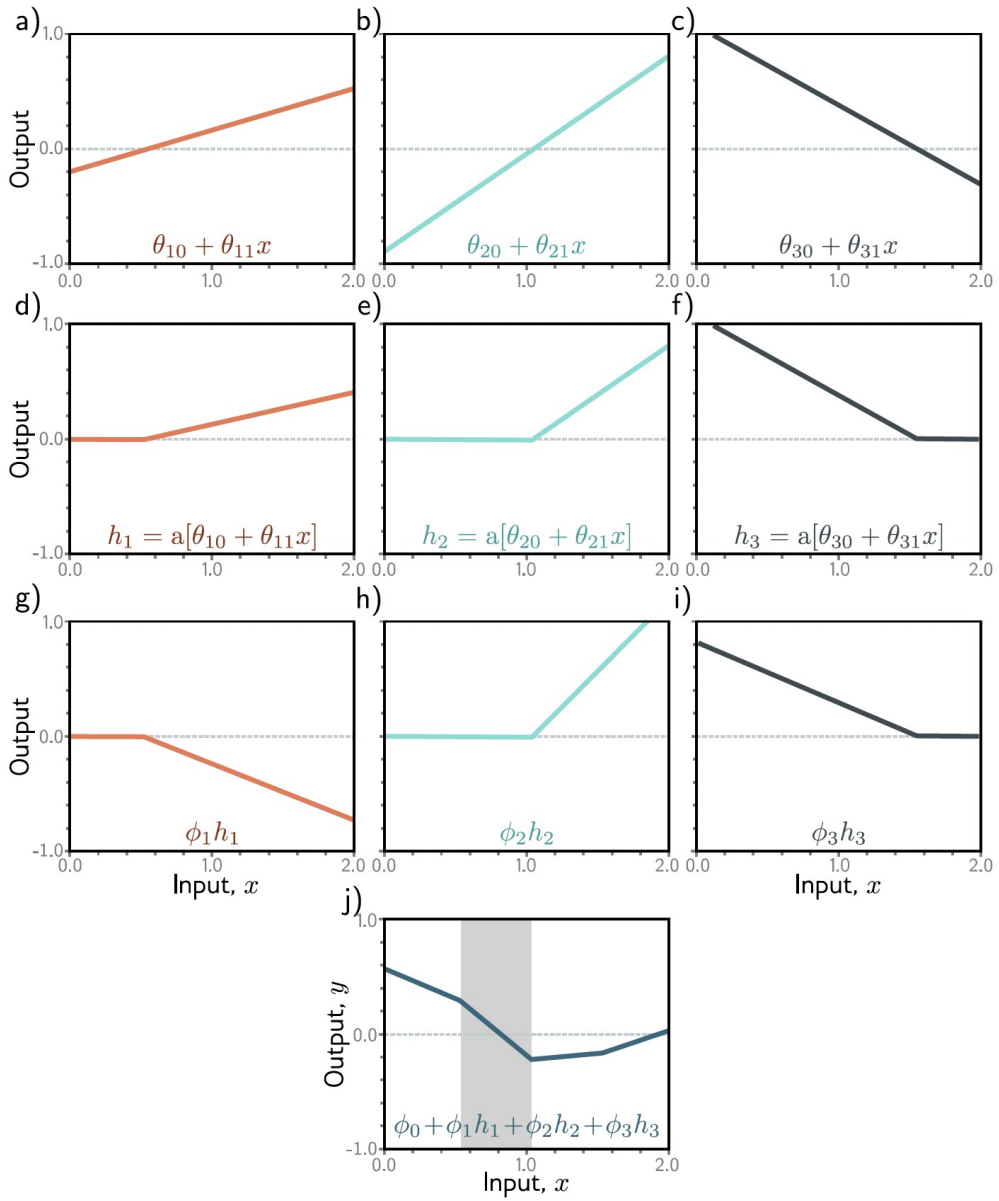


Figure 3.3 Computation for function in figure 3.2a. a–c) The input x is passed through three linear functions, each with a different y-intercept $\theta_{\bullet 0}$ and slope $\theta_{\bullet 1}$. d–f) Each line is passed through the ReLU activation function, which clips negative values to zero. g–i) The three clipped lines are then weighted (scaled) by ϕ_1, ϕ_2, and ϕ_3, respectively. j) Finally, the clipped and weighted functions are summed, and an offset ϕ_0 that controls the height is added. Each of the four linear regions corresponds to a different activation pattern in the hidden units. In the shaded region, h_2 is inactive (clipped), but h_1 and h_3 are both active.

Source: Understanding Deep Learning, S. Prince, 2023

ect to a Creative Commons CC-BY-NC-ND license. (C) MIT Press.

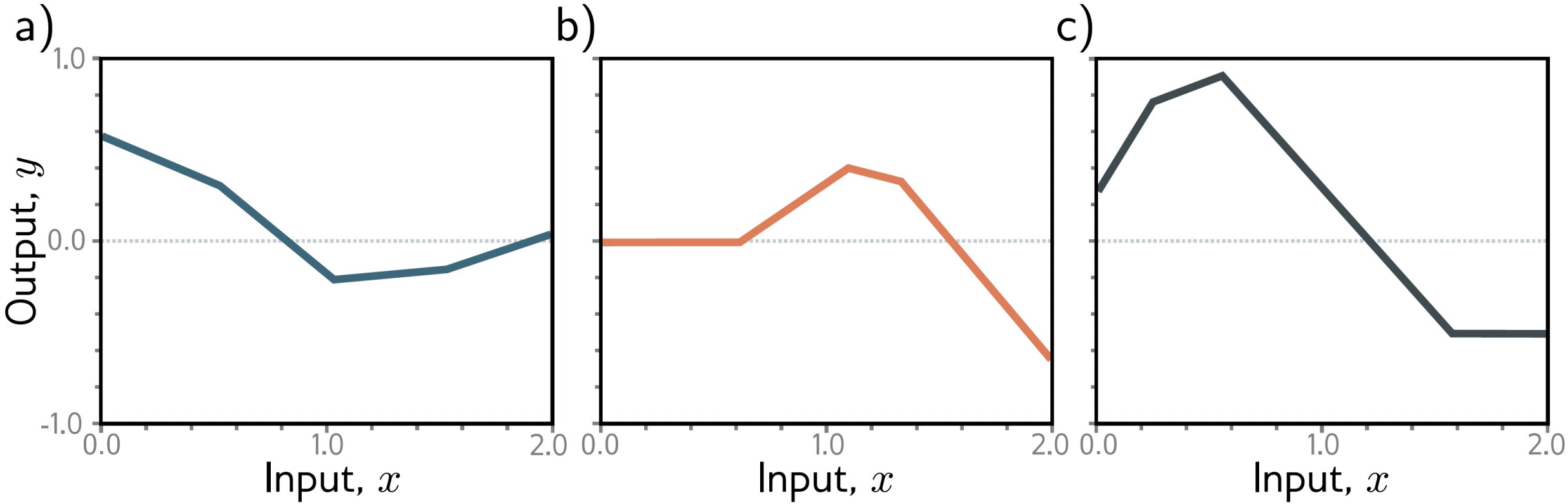


Figure 3.2 Family of functions defined by equation 3.1. a–c) Functions for three different choices of the ten parameters ϕ. In each case, the input/output relation is piecewise linear. However, the positions of the joints, the slopes of the linear regions between them, and the overall height vary.

Source: Understanding Deep Learning, S. Prince, 2023

This work is subject to a Creative Commons CC-BY-NC-ND license. (C) MIT Press.

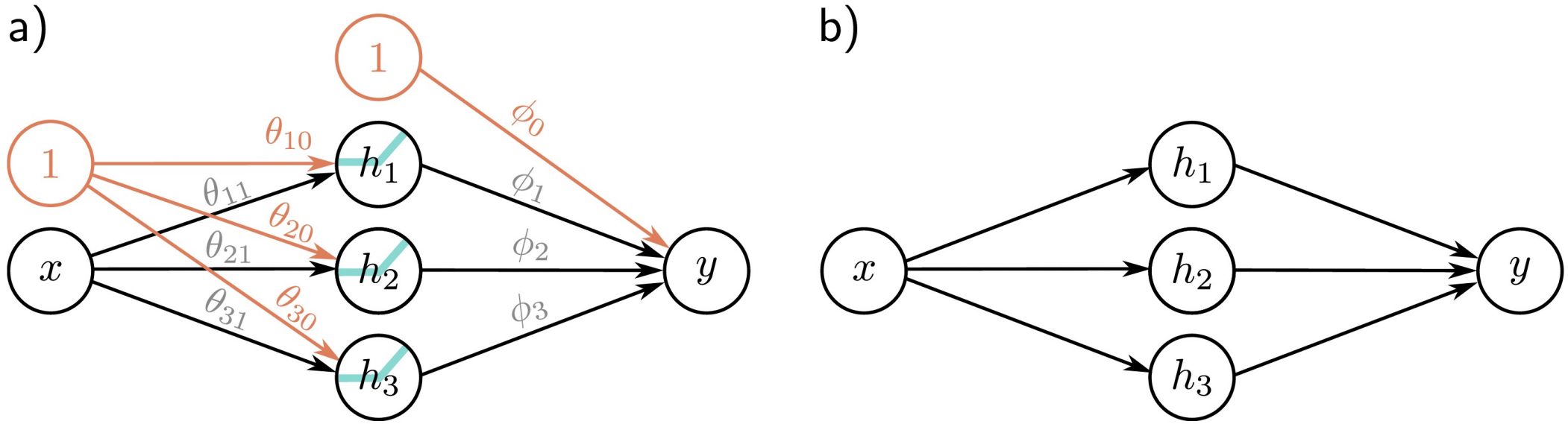


Figure 3.4 Depicting neural networks. a) The input x is on the left, the hidden units h_1, h_2, and h_3 in the center, and the output y on the right. Computation flows from left to right. The input is used to compute the hidden units, which are combined to create the output. Each of the ten arrows represents a parameter (intercepts in orange and slopes in black). Each parameter multiplies its source and adds the result to its target. For example, we multiply the parameter ϕ_1 by source h_1 and add it to y. We introduce additional nodes containing ones (orange circles) to incorporate the offsets into this scheme, so we multiply ϕ_0 by one (with no effect) and add it to y. ReLU functions are applied at the hidden units. b) More typically, the intercepts, ReLU functions, and parameter names are omitted; this simpler depiction represents the same network.

Source: Understanding Deep Learning, S. Prince, 2023

This work is subject to a Creative Commons CC-BY-NC-ND license. (C) MIT Press.

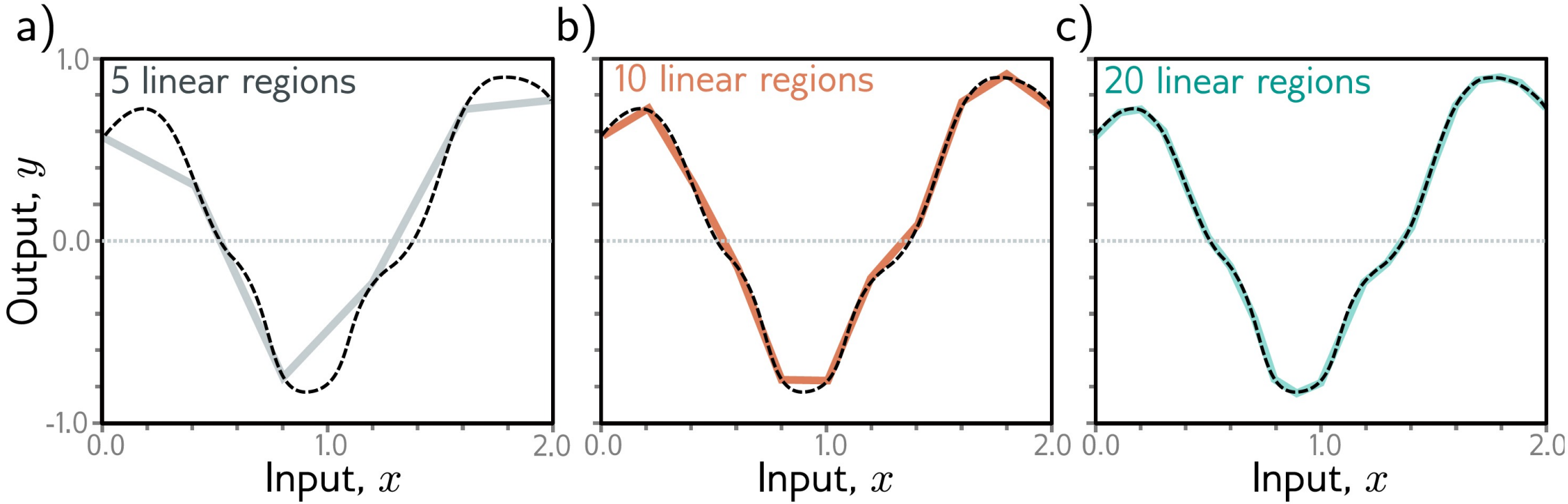


Figure 3.5 Approximation of a 1D function (dashed line) by a piecewise linear model. a–c) As the number of regions increases, the model becomes closer and closer to the continuous function. A neural network with a scalar input creates one extra linear region per hidden unit. The universal approximation theorem proves that, with enough hidden units, there exists a shallow neural network can describe any given continuous function defined on a compact subset of $\mathbb{R}^D$ to arbitrary precision.

Source: Understanding Deep Learning, S. Prince, 2023

This work is subject to a Creative Commons CC-BY-NC-ND license. (C) MIT Press.

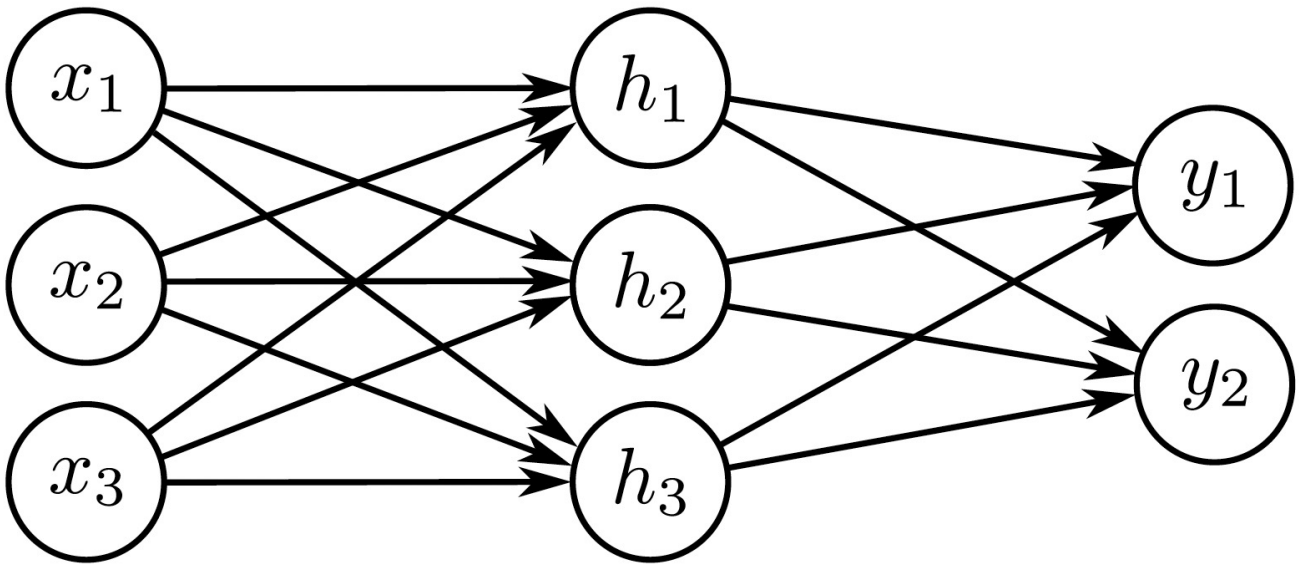


Figure 3.11 Visualization of neural network with three inputs and two outputs. This network has twenty parameters. There are fifteen slopes (indicated by arrows) and five offsets (not shown).

Source: Understanding Deep Learning, S. Prince, 2023

This work is subject to a Creative Commons CC-BY-NC-ND license. (C) MIT Press.

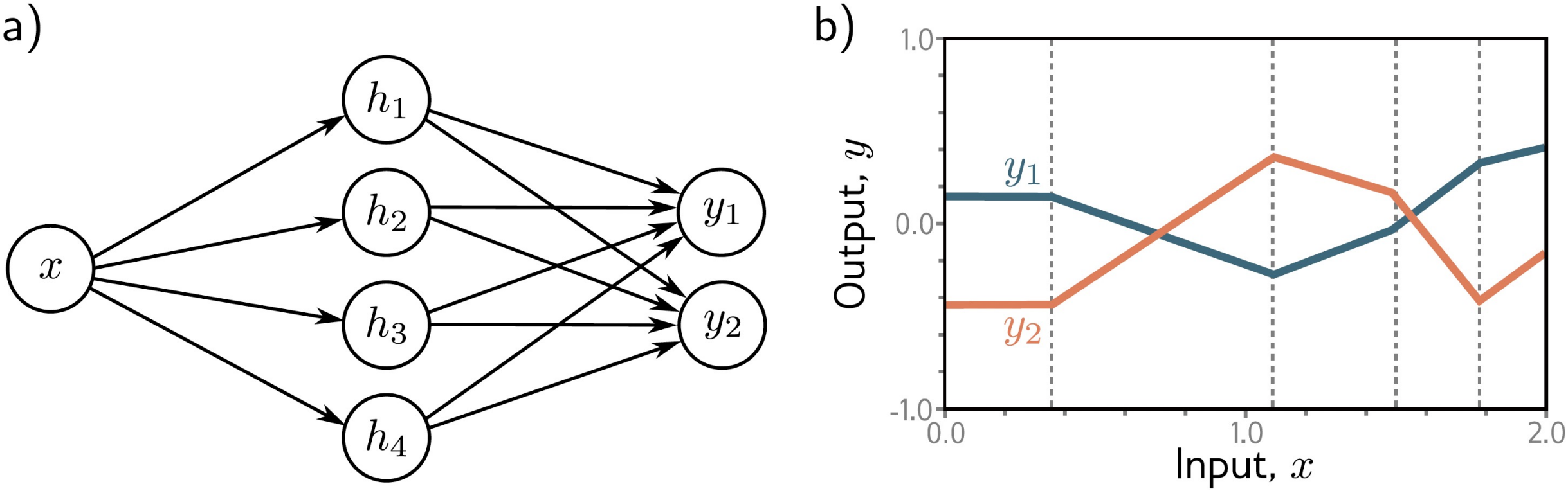


Figure 3.6 Network with one input, four hidden units, and two outputs. a) Visualization of network structure. b) This network produces two piecewise linear functions, $y_1[x]$ and $y_2[x]$. The four "joints" of these functions (at vertical dotted lines) are constrained to be in the same places since they share the same hidden units, but the slopes and overall height may differ.

Source: Understanding Deep Learning, S. Prince, 2023

This work is subject to a Creative Commons CC-BY-NC-ND license. (C) MIT Press.

Universal approximation theorem

For any continuous function, there exists a shallow network that can approximate this function to any specified precision.

Universal approximation theorem

Universal approximation theorem (Uniform non-affine activation, arbitrary depth, constrained width). Let $\mathcal{X}$ be a compact subset of $\mathbb{R}^d$. Let $\sigma : \mathbb{R} \to \mathbb{R}$ be any non-affine continuous function which is continuously differentiable at at least one point, with nonzero derivative at that point. Let $\mathcal{N}_{d,D:d+D+2}^{\sigma}$ denote the space of feed-forward neural networks with d input neurons, D output neurons, and an arbitrary number of hidden layers each with $d + D + 2$ neurons, such that every hidden neuron has activation function σ and every output neuron has the identity as its activation function, with input layer ϕ and output layer ρ. Then given any $\varepsilon > 0$ and any $f \in C(\mathcal{X}, \mathbb{R}^D)$, there exists $\hat{f} \in \mathcal{N}_{d,D:d+D+2}^{\sigma}$ such that

$$\sup_{x \in \mathcal{X}} \left\| \hat{f}(x) - f(x) \right\| < \varepsilon.$$

In other words, $\mathcal{N}$ is dense in $C(\mathcal{X}; \mathbb{R}^D)$ with respect to the topology of uniform convergence.

No free-lunch theorem

Every learning algorithm is as good as any other when averaged over all sets of problems.

You can’t just learn « purely from data » without bias.

Wolpert, D. H.; Macready, W. G. (1997). "No Free Lunch Theorems for Optimization". *IEEE Transactions on Evolutionary Computation*. **1**: 67–82.

Why build deep networks if shallow networks are universal appromixators already?

Reason #1: More linear regions per parameter

D: number of units / layer
K: number of layers

$3D + 1 + (K-1)D(D+1)$ regions

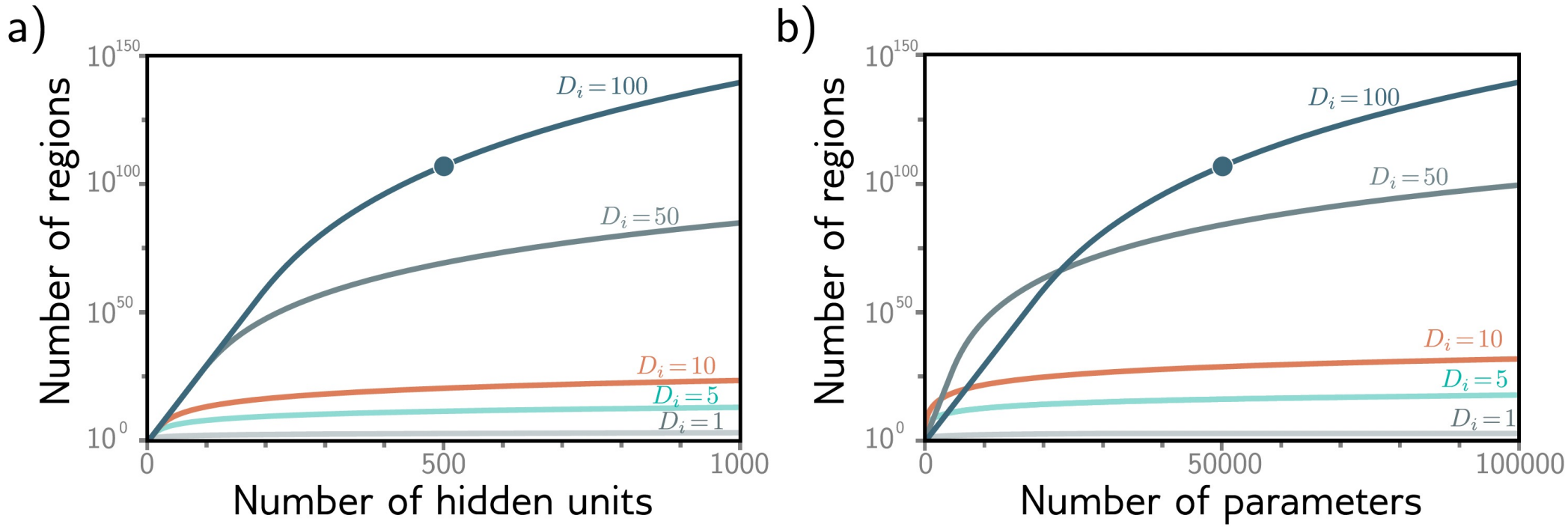


Figure 3.9 Linear regions vs. hidden units. a) Maximum possible regions as a function of the number of hidden units for five different input dimensions $D_i = \{1, 5, 10, 50, 100\}$. The number of regions increases rapidly in high dimensions; with $D = 500$ units and input size $D_i = 100$, there can be greater than 10^{107} regions (solid circle). b) The same data are plotted as a function of the number of parameters. The solid circle represents the same model as in panel (a) with $D = 500$ hidden units. This network has $51,001$ parameters and would be considered very small by modern standards.

Source: Understanding Deep Learning, S. Prince, 2023

This work is subject to a Creative Commons CC-BY-NC-ND license. (C) MIT Press.

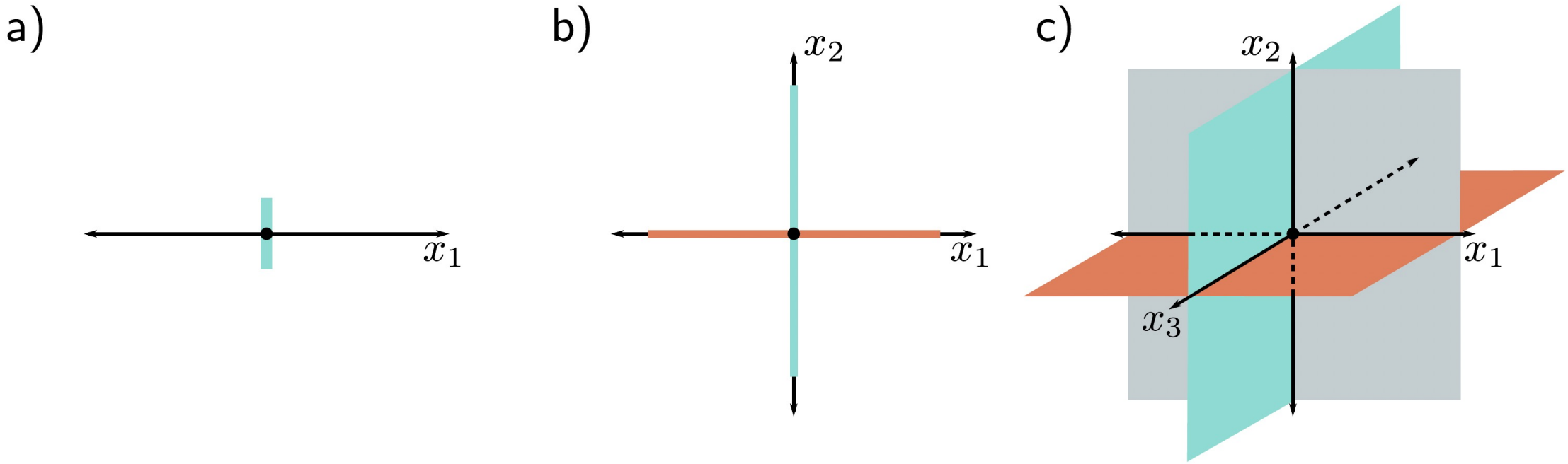


Figure 3.10 Number of linear regions vs. input dimensions. a) With a single input dimension, a model with one hidden unit creates one joint, which divides the axis into two linear regions. b) With two input dimensions, a model with two hidden units can divide the input space using two lines (here aligned with axes) to create four regions. c) With three input dimensions, a model with three hidden units can divide the input space using three planes (again aligned with axes) to create eight regions. Continuing this argument, it follows that a model with D_i input dimensions and D_i hidden units can divide the input space with D_i hyperplanes to create 2^{D_i} linear regions.

Source: Understanding Deep Learning, S. Prince, 2023

This work is subject to a Creative Commons CC-BY-NC-ND license. (C) MIT Press.

Reason #2: large, structured inputs

Deep networks allow us to bake the structure of the data into the structure of the network.

e.g. images and convnets

Reason #3: depth efficiency?

Try fitting the x^2 function with an MLP.

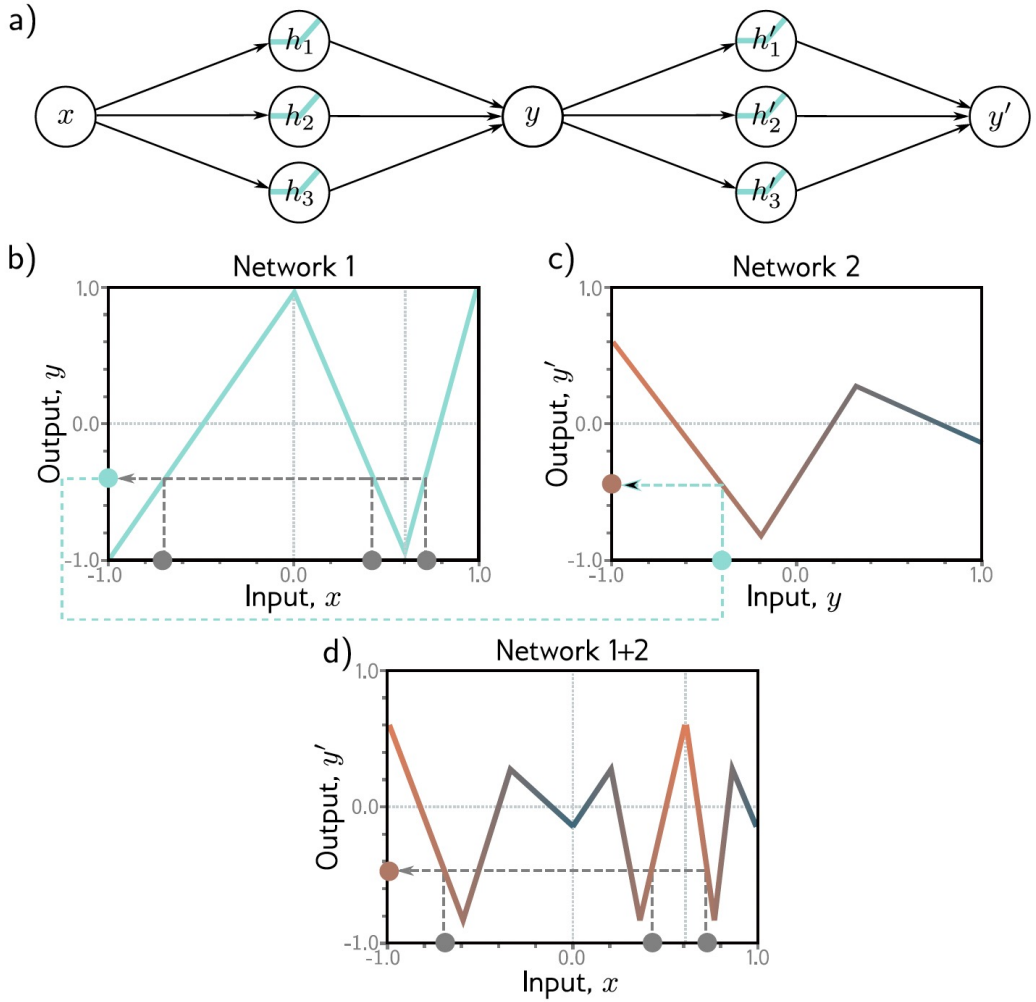


Figure 4.1 Composing two single-layer networks with three hidden units each. a) The output y of the first network constitutes the input to the second network. b) The first network maps inputs $x \in [-1, 1]$ to outputs $y \in [-1, 1]$ using a function comprised of three linear regions that are chosen so that they alternate the sign of their slope. Multiple inputs x (gray circles) now map to the same output y (cyan circle). c) The second network defines a function comprising three linear regions that takes y and returns y' (i.e., the cyan circle is mapped to the brown circle). d) The combined effect of these two functions when composed is that (i) three different inputs x are mapped to any given value of y by the first network and (ii) are processed in the same way by the second network; the result is that the function defined by the second network in panel (c) is duplicated three times, variously flipped and rescaled according to the slope of the regions of panel (b).

Source: Understanding Deep Learning, S. Prince, 2023

This work is subject to a Creative Commons CC-BY-NC-ND license. (C) MIT Press.

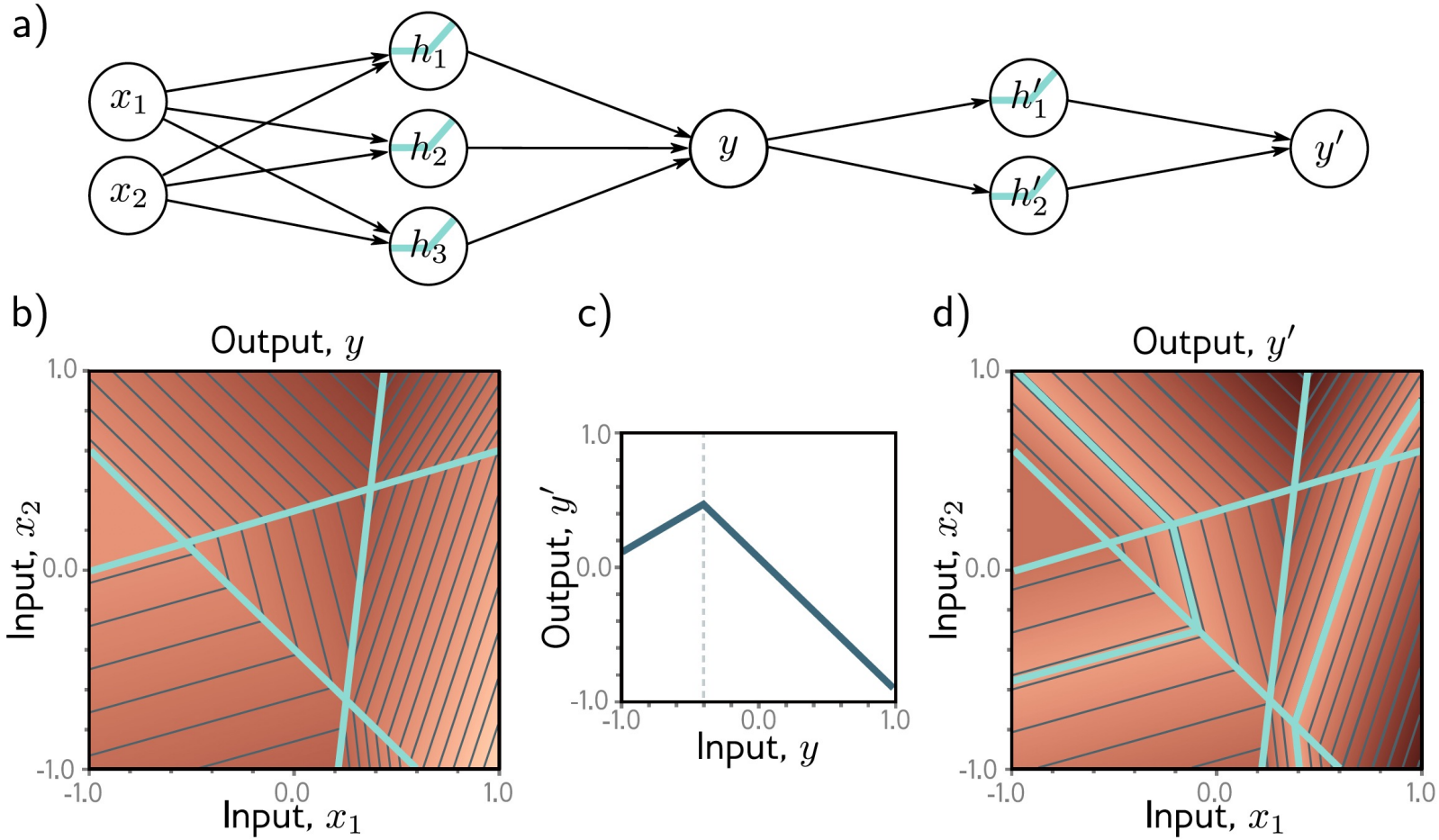


Figure 4.2 Composing neural networks with a 2D input. a) The first network (from figure 3.8) has three hidden units and takes two inputs x_1 and x_2 and returns a scalar output y. This is passed into a second network with two hidden units to produce y'. b) The first network produces a function consisting of seven linear regions, one of which is flat. c) The second network defines a function comprising two linear regions in $y \in [-1, 1]$. d) When these networks are composed, each of the six non-flat regions from the first network is divided into two new regions by the second network to create a total of 13 linear regions.

Source: Understanding Deep Learning, S. Prince, 2023

This work is subject to a Creative Commons CC-BY-NC-ND license. (C) MIT Press.

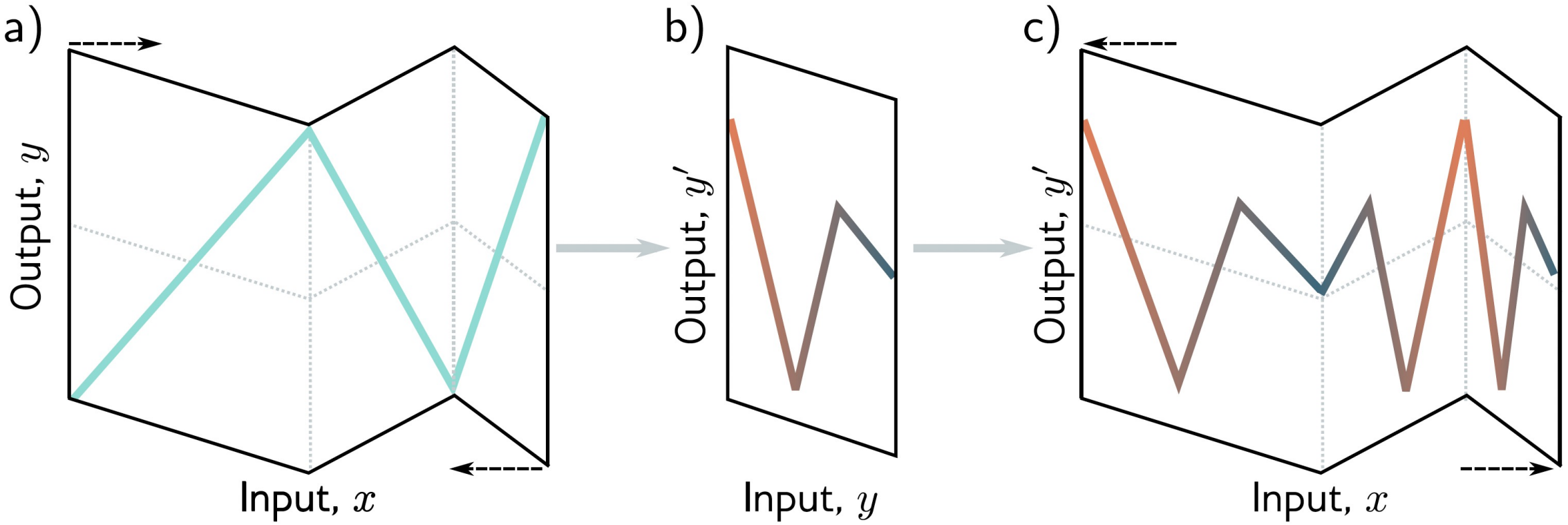


Figure 4.3 Deep networks as folding input space. a) One way to think about the first network from figure 4.1 is that it "folds" the input space back on top of itself. b) The second network applies its function to the folded space. c) The final output is revealed by "unfolding" again.

Source: Understanding Deep Learning, S. Prince, 2023

This work is subject to a Creative Commons CC-BY-NC-ND license. (C) MIT Press.

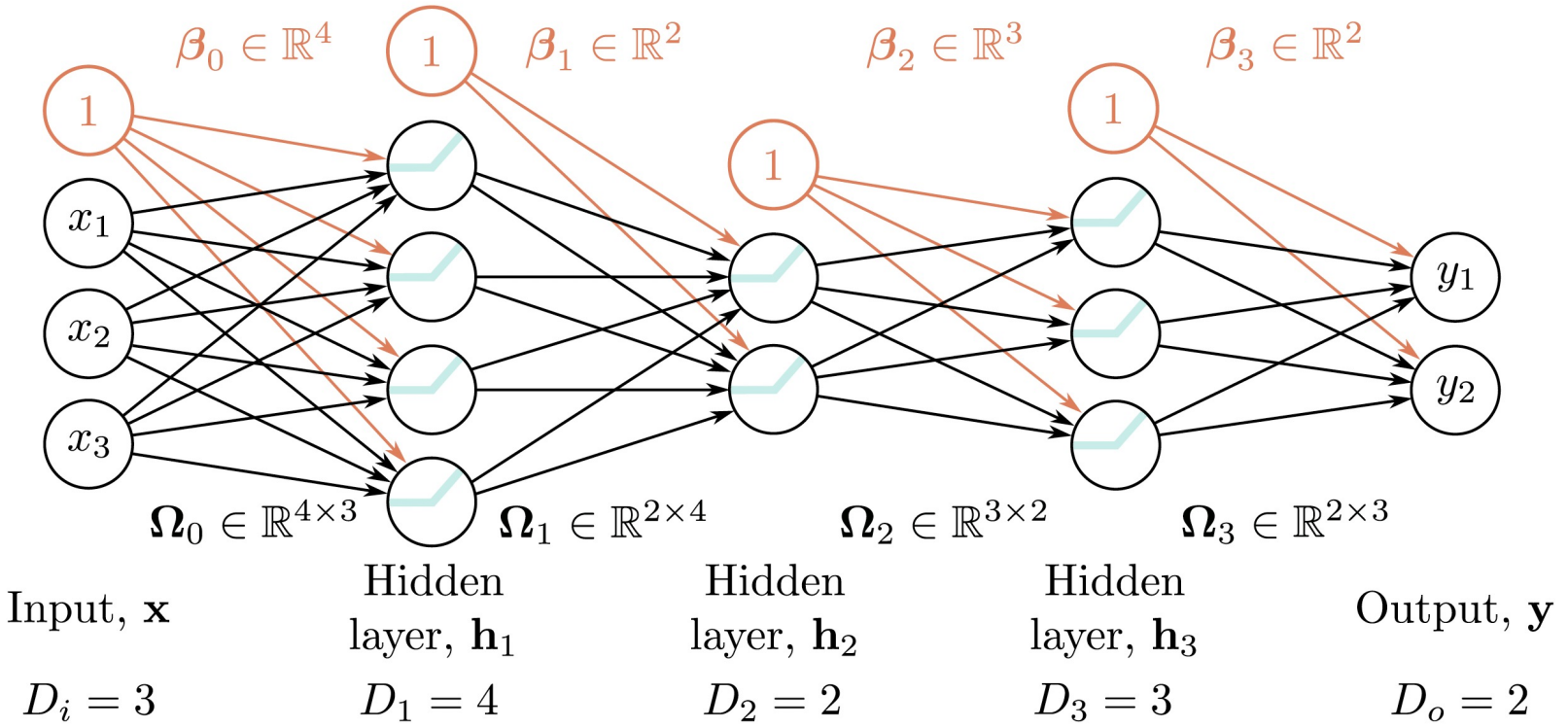


Figure 4.6 Matrix notation for network with $D_i = 3$-dimensional input $\mathbf{x}$, $D_o = 2$-dimensional output $\mathbf{y}$, and $K = 3$ hidden layers $\mathbf{h}_1, \mathbf{h}_2$, and $\mathbf{h}_3$ of dimensions $D_1 = 4$, $D_2 = 2$, and $D_3 = 3$ respectively. The weights are stored in matrices $\boldsymbol{\Omega}_k$ that pre-multiply the activations from the preceding layer to create the pre-activations at the subsequent layer. For example, the weight matrix $\boldsymbol{\Omega}_1$ that computes the pre-activations at $\mathbf{h}_2$ from the activations at $\mathbf{h}_1$ has dimension 2×4. It is applied to the four hidden units in layer one and creates the inputs to the two hidden units at layer two. The biases are stored in vectors $\boldsymbol{\beta}_k$ and have the dimension of the layer into which they feed. For example, the bias vector $\boldsymbol{\beta}_2$ is length three because layer $\mathbf{h}_3$ contains three hidden units.

Source: Understanding Deep Learning, S. Prince, 2023

This work is subject to a Creative Commons CC-BY-NC-ND license. (C) MIT Press.

Training neural networks with backpropagation

Source: Deep Learning, Foundations and Concepts, Bishop, 2023

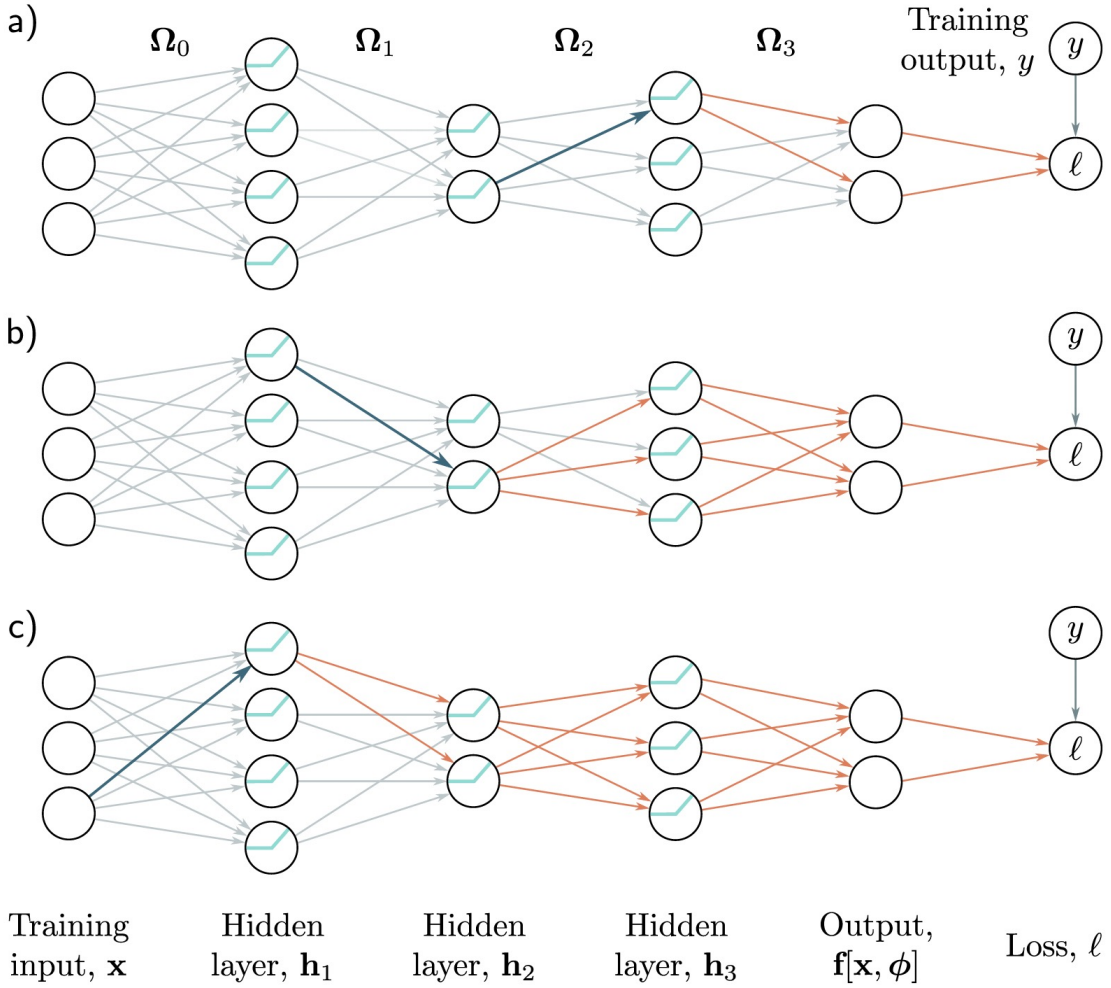


Figure 7.2 Backpropagation backward pass. a) To compute how a change to a weight feeding into layer $\mathbf{h}_3$ (blue arrow) changes the loss, we need to know how the hidden unit in $\mathbf{h}_3$ changes the model output $\mathbf{f}$ and how $\mathbf{f}$ changes the loss (orange arrows). b) To compute how a small change to a weight feeding into $\mathbf{h}_2$ (blue arrow) changes the loss, we need to know (i) how the hidden unit in $\mathbf{h}_2$ changes $\mathbf{h}_3$, (ii) how $\mathbf{h}_3$ changes $\mathbf{f}$, and (iii) how $\mathbf{f}$ changes the loss (orange arrows). c) Similarly, to compute how a small change to a weight feeding into $\mathbf{h}_1$ (blue arrow) changes the loss, we need to know how $\mathbf{h}_1$ changes $\mathbf{h}_2$ and how these changes propagate through to the loss (orange arrows). The backward pass first computes derivatives at the end of the network and then works backward to exploit the inherent redundancy of these computations.

This work is subject to a Creative Commons CC-BY-NC-ND license. (C) MIT Press.

$$
\begin{aligned}
\mathbf{f}_0 &= \boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}_i \\
\mathbf{h}_1 &= \mathbf{a}[\mathbf{f}_0] \\
\mathbf{f}_1 &= \boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1 \mathbf{h}_1 \\
\mathbf{h}_2 &= \mathbf{a}[\mathbf{f}_1] \\
\mathbf{f}_2 &= \boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2 \mathbf{h}_2 \\
\mathbf{h}_3 &= \mathbf{a}[\mathbf{f}_2] \\
\mathbf{f}_3 &= \boldsymbol{\beta}_3 + \boldsymbol{\Omega}_3 \mathbf{h}_3 \\
\ell_i &= \mathrm{l}[\mathbf{f}_3, y_i], \qquad (7.16)
\end{aligned}
$$

This work is subject to a Creative Commons CC-BY-NC-ND license. (C) MIT Press.

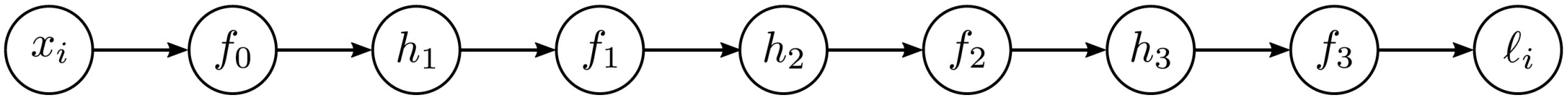


Figure 7.3 Backpropagation forward pass. We compute and store each of the intermediate variables in turn until we finally calculate the loss.

This work is subject to a Creative Commons CC-BY-NC-ND license. (C) MIT Press.

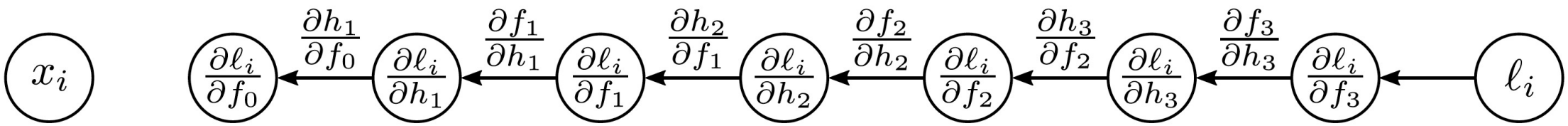


Figure 7.4 Backpropagation backward pass #1. We work backward from the end of the function computing the derivatives $\partial \ell_i / \partial f_\bullet$ and $\partial \ell_i / \partial h_\bullet$ of the loss with respect to the intermediate quantities. Each derivative is computed from the previous one by multiplying by terms of the form $\partial f_k / \partial h_k$ or $\partial h_k / \partial f_{k-1}$.

This work is subject to a Creative Commons CC-BY-NC-ND license. (C) MIT Press.

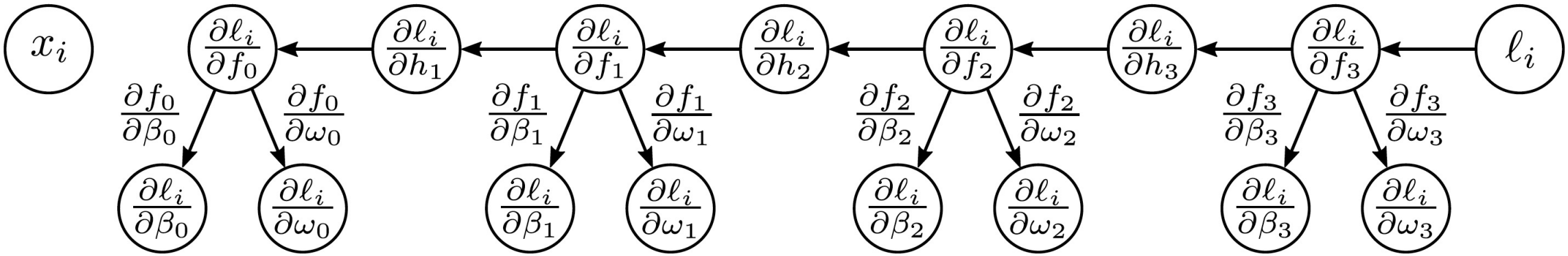


Figure 7.5 Backpropagation backward pass #2. Finally, we compute the derivatives $\partial \ell_i / \partial \beta_\bullet$ and $\partial \ell_i / \partial \omega_\bullet$. Each derivative is computed by multiplying the term $\partial \ell_i / \partial f_k$ by $\partial f_k / \partial \beta_k$ or $\partial f_k / \partial \omega_k$ as appropriate.

This work is subject to a Creative Commons CC-BY-NC-ND license. (C) MIT Press.

The chain rule for efficient backward computation

$$\frac{\partial \ell_i}{\partial \mathbf{f}_1} = \frac{\partial \mathbf{h}_2}{\partial \mathbf{f}_1}\frac{\partial \mathbf{f}_2}{\partial \mathbf{h}_2}\left(\frac{\partial \mathbf{h}_3}{\partial \mathbf{f}_2}\frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_3}\frac{\partial \ell_i}{\partial \mathbf{f}_3}\right) \tag{7.18}$$

$$\frac{\partial \ell_i}{\partial \mathbf{f}_0} = \frac{\partial \mathbf{h}_1}{\partial \mathbf{f}_0}\frac{\partial \mathbf{f}_1}{\partial \mathbf{h}_1}\left(\frac{\partial \mathbf{h}_2}{\partial \mathbf{f}_1}\frac{\partial \mathbf{f}_2}{\partial \mathbf{h}_2}\frac{\partial \mathbf{h}_3}{\partial \mathbf{f}_2}\frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_3}\frac{\partial \ell_i}{\partial \mathbf{f}_3}\right). \tag{7.19}$$

This work is subject to a Creative Commons CC-BY-NC-ND license. (C) MIT Press.

Forward pass: We compute and store the following quantities:

$$\begin{aligned}
\mathbf{f}_0 &= \boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}_i \\
\mathbf{h}_k &= \mathbf{a}[\mathbf{f}_{k-1}] & k \in \{1, 2, \ldots, K\} \\
\mathbf{f}_k &= \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{h}_k. & k \in \{1, 2, \ldots, K\}
\end{aligned} \tag{7.23}$$

Backward pass: We start with the derivative $\partial \ell_i / \partial \mathbf{f}_K$ of the loss function ℓ_i with respect to the network output $\mathbf{f}_K$ and work backward through the network:

$$\begin{aligned}
\frac{\partial \ell_i}{\partial \boldsymbol{\beta}_k} &= \frac{\partial \ell_i}{\partial \mathbf{f}_k} & k \in \{K, K-1, \ldots, 1\} \\
\frac{\partial \ell_i}{\partial \boldsymbol{\Omega}_k} &= \frac{\partial \ell_i}{\partial \mathbf{f}_k} \mathbf{h}_k^T & k \in \{K, K-1, \ldots, 1\} \\
\frac{\partial \ell_i}{\partial \mathbf{f}_{k-1}} &= \mathbb{I}[\mathbf{f}_{k-1} > 0] \odot \left(\boldsymbol{\Omega}_k^T \frac{\partial \ell_i}{\partial \mathbf{f}_k} \right), & k \in \{K, K-1, \ldots, 1\}
\end{aligned} \tag{7.24}$$

where $\odot$ denotes pointwise multiplication, and $\mathbb{I}[\mathbf{f}_{k-1} > 0]$ is a vector containing ones where $\mathbf{f}_{k-1}$ is greater than zero and zeros elsewhere. Finally, we compute the derivatives with respect to the first set of biases and weights:

$$\begin{aligned}
\frac{\partial \ell_i}{\partial \boldsymbol{\beta}_0} &= \frac{\partial \ell_i}{\partial \mathbf{f}_0} \\
\frac{\partial \ell_i}{\partial \boldsymbol{\Omega}_0} &= \frac{\partial \ell_i}{\partial \mathbf{f}_0} \mathbf{x}_i^T.
\end{aligned} \tag{7.25}$$

This work is subject to a Creative Commons CC-BY-NC-ND license. (C) MIT Press.

Practical 1

Source: Deep Learning, Foundations and Concepts, Bishop, 2023

Appendix

Source: Deep Learning, Foundations and Concepts, Bishop, 2023

Training neural networks with backpropagation

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}).$$

Error function

$$E_n = \frac{1}{2} \sum_k (y_{nk} - t_{nk})^2$$

SSE example

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j}$$

Chain rule

Source: Deep Learning, Foundations and Concepts, Bishop, 2023

Sanity checking with finite differences

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{E_n(w_{ji} + \epsilon) - E_n(w_{ji} - \epsilon)}{2\epsilon} + O(\epsilon^2).$$

Source: Deep Learning, Foundations and Concepts, Bishop, 2023

Input: Input vector $\mathbf{x}_n$

Network parameters $\mathbf{w}$

Error function $E_n(\mathbf{w})$ for input x_n

Activation function $h(a)$

Output: Error function derivatives $\{\partial E_n/\partial w_{ji}\}$

`// Forward propagation`

for $j \in$ all hidden and output units **do**

$a_j \leftarrow \sum_i w_{ji} z_i$ `// {z_i} includes inputs {x_i}`

$z_j \leftarrow h(a_j)$ `// activation function`

end for

`// Error evaluation`

for $k \in$ all output units **do**

$\delta_k \leftarrow \dfrac{\partial E_n}{\partial a_k}$ `// compute errors`

end for

`// Backward propagation, in reverse order`

for $j \in$ all hidden units **do**

$\delta_j \leftarrow h'(a_j) \sum_k w_{kj} \delta_k$ `// recursive backward evaluation`

$\dfrac{\partial E_n}{\partial w_{ji}} \leftarrow \delta_j z_i$ `// evaluate derivatives`

end for

return $\left\{ \dfrac{\partial E_n}{\partial w_{ji}} \right\}$

Source: Deep Learning, Foundations and Concepts, Bishop, 2023

But there are other universal approximators!

What about Gaussian processes?

GP *advantages* / **DNN** *disadvantages*:

1. Generally require less data than **DNNs**
2. Only need to optimize a small number of (hyper)parameters.
3. Are robust to phenomena such as exploding and vanishing gradients (since, unless you are using Deep **GPs**, there is no “layer structure” within this framework).

GP *disadvantages* / **DNN** *advantages*:

1. Runtime scales poorly with the number of samples. The runtime complexity is $\boldsymbol{O(n^3)}$, where $\boldsymbol{n}$ is the number of samples. This is a result of having to perform matrix inversion (or pseudo-inversion) of large covariance matrices.

2. There is less automatic learning relative to neural networks, and more design considerations need to be made for the choice of kernel/covariance function, mean function, and hyperparameter prior distributions. These parameters can have a substantial effect on what the **GP** is able to learn.

In short: deep learning scales to larger datasets