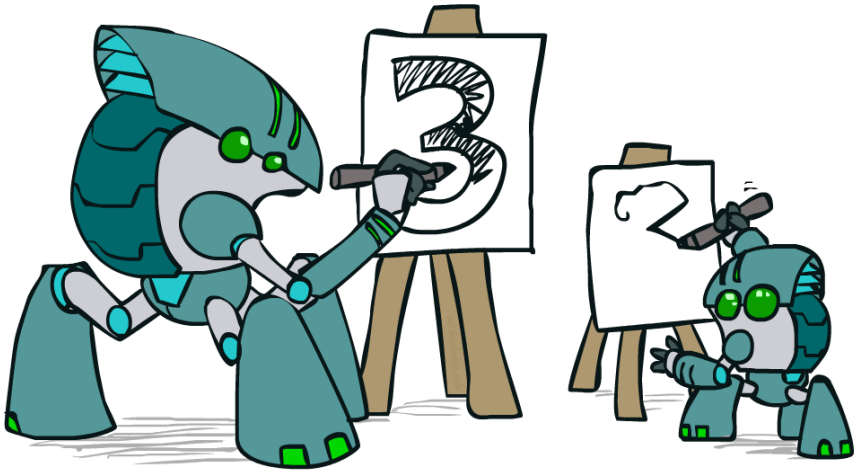
Introduction to Artificial Intelligence

Lecture 7: Machine learning and neural networks

Prof. Gilles Louppe
g.louppe@uliege.be

Today



Learning from data is a key component of artificial intelligence. In this lecture, we will introduce the principles of:

- Machine learning
- Neural networks

—

Credits: CS188, UC Berkeley.

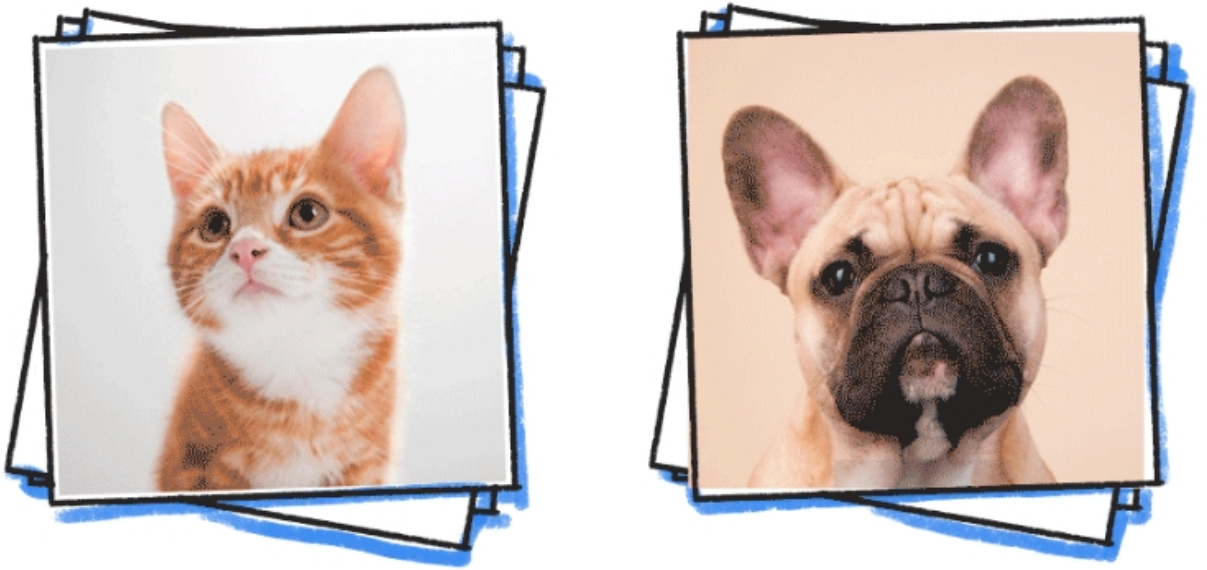
Learning agents

What if the environment is unknown?

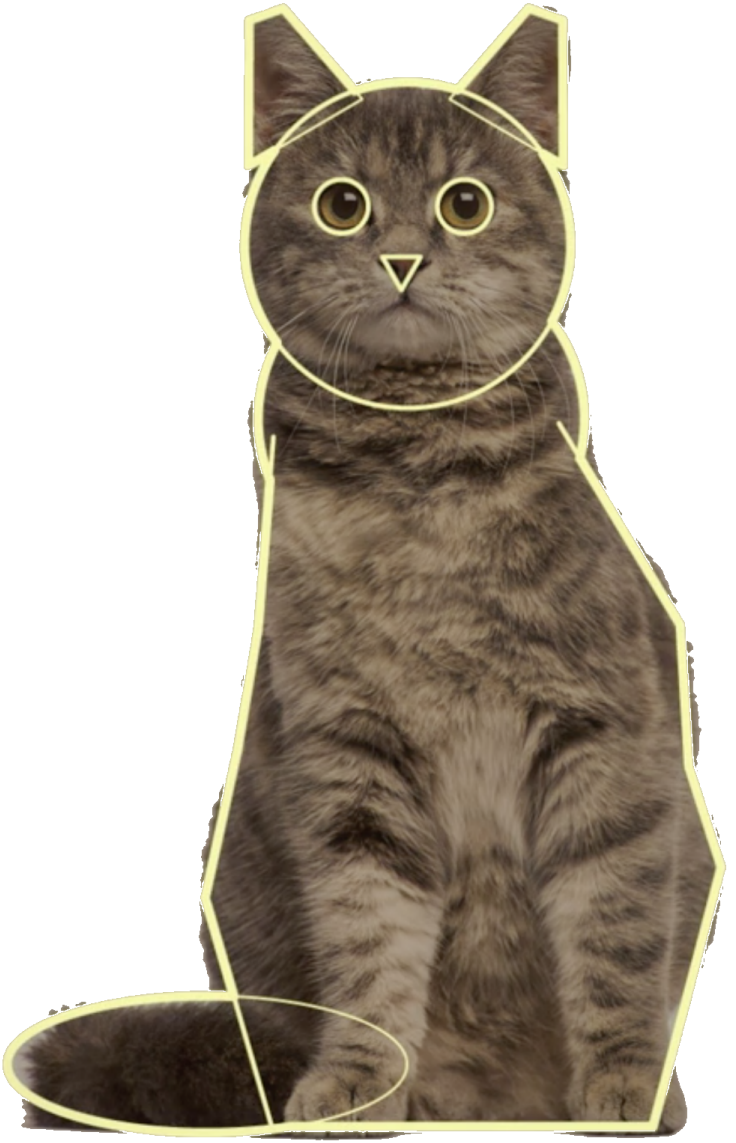
- Learning provides an automated way to modify the agent's internal decision mechanisms to improve its own performance.
- It exposes the agent to reality rather than trying to hardcode reality into the agent's program.

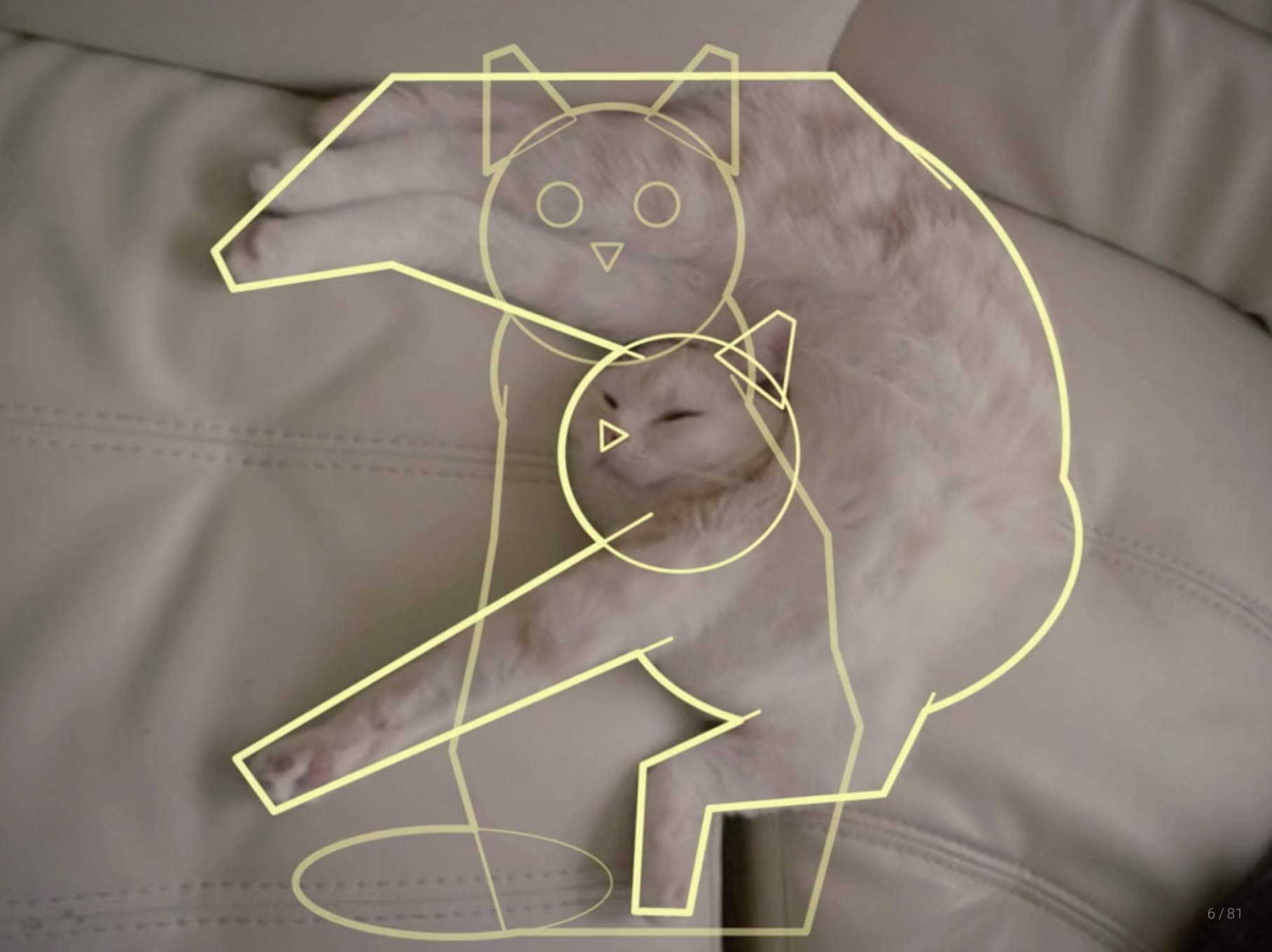
More generally, learning is useful for any task where it is difficult to write a program that performs the task but easy to obtain examples of desired behavior.

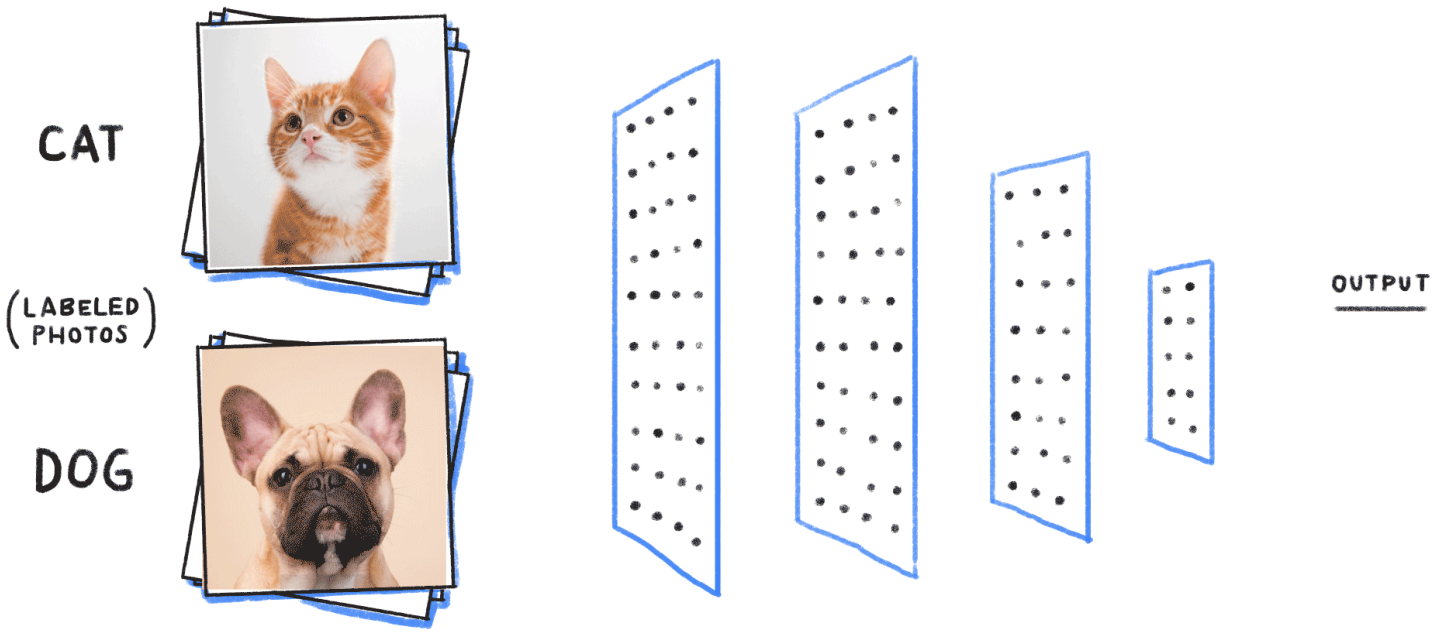
Machine learning



How would you write a computer program that recognizes cats from dogs?







The deep learning approach: train a highly parameterized model to produce correct outputs (e.g., class labels) from inputs (e.g., images).

Problem statement

Let $\mathbf{d} \sim p(\mathbf{x}, y)$ be a dataset of N example input-output pairs

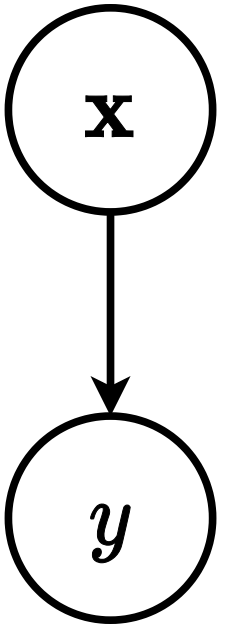
$$\mathbf{d} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), ..., (\mathbf{x}_N, y_N)\},$$

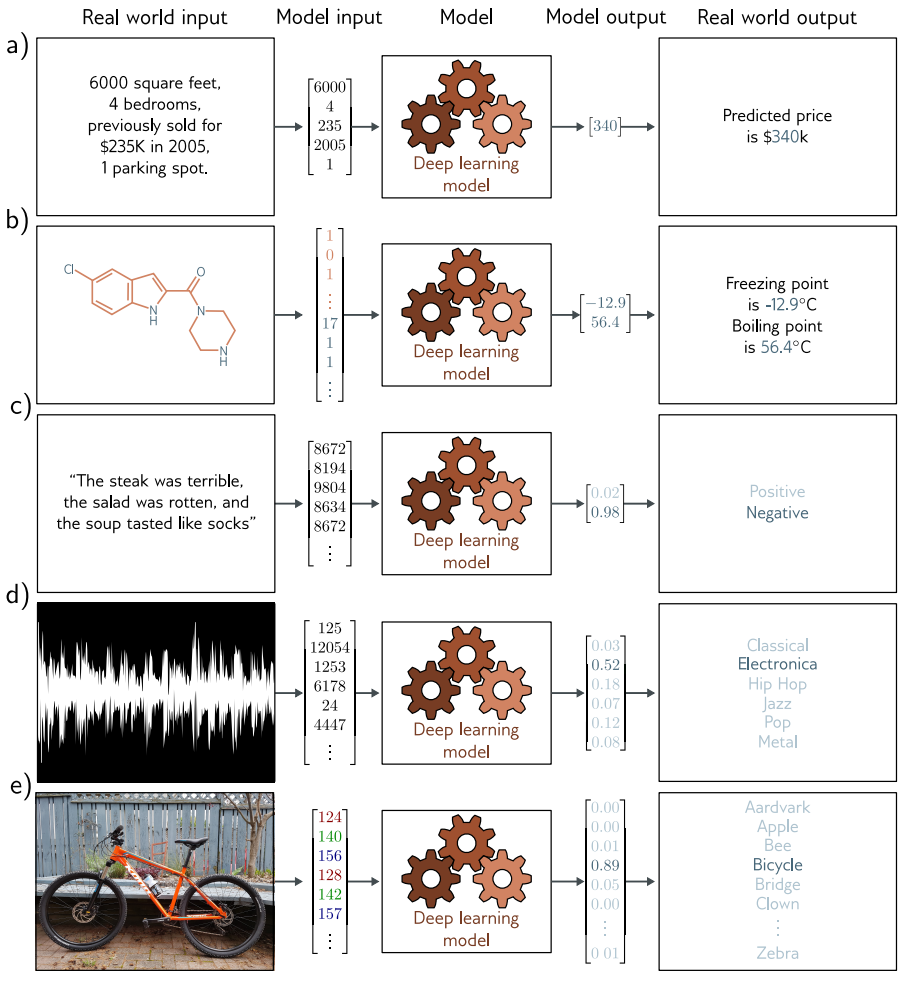
where $\mathbf{x}_i \in \mathbb{R}^d$ are d-dimensional vectors representing the input values and $y_i \in \mathcal{Y}$ are the corresponding output values.

From this data, we want to identify a probabilistic model

$$p_\theta(y|\mathbf{x})$$

that best explains the data.

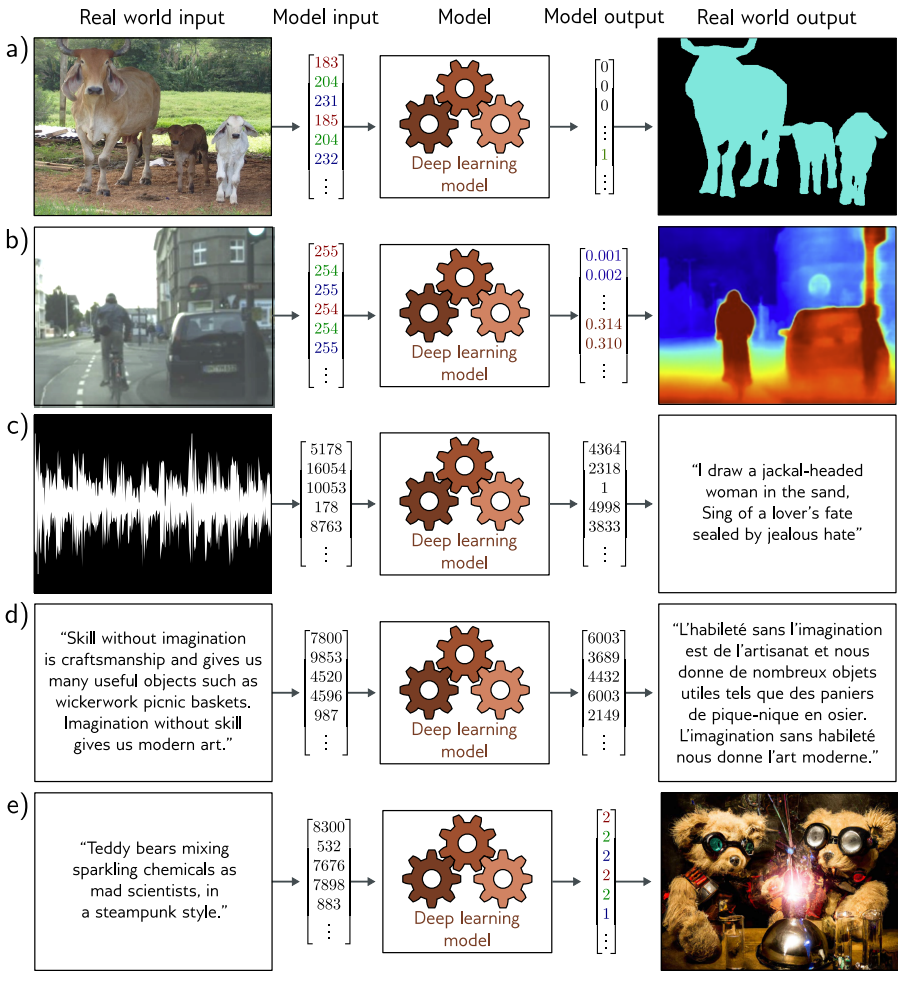




Regression ($y \in \mathbb{R}$) and classification ($y \in \{0, 1, ..., C-1\}$) problems.

—

Credits: Simon J.D. Prince, 2023.

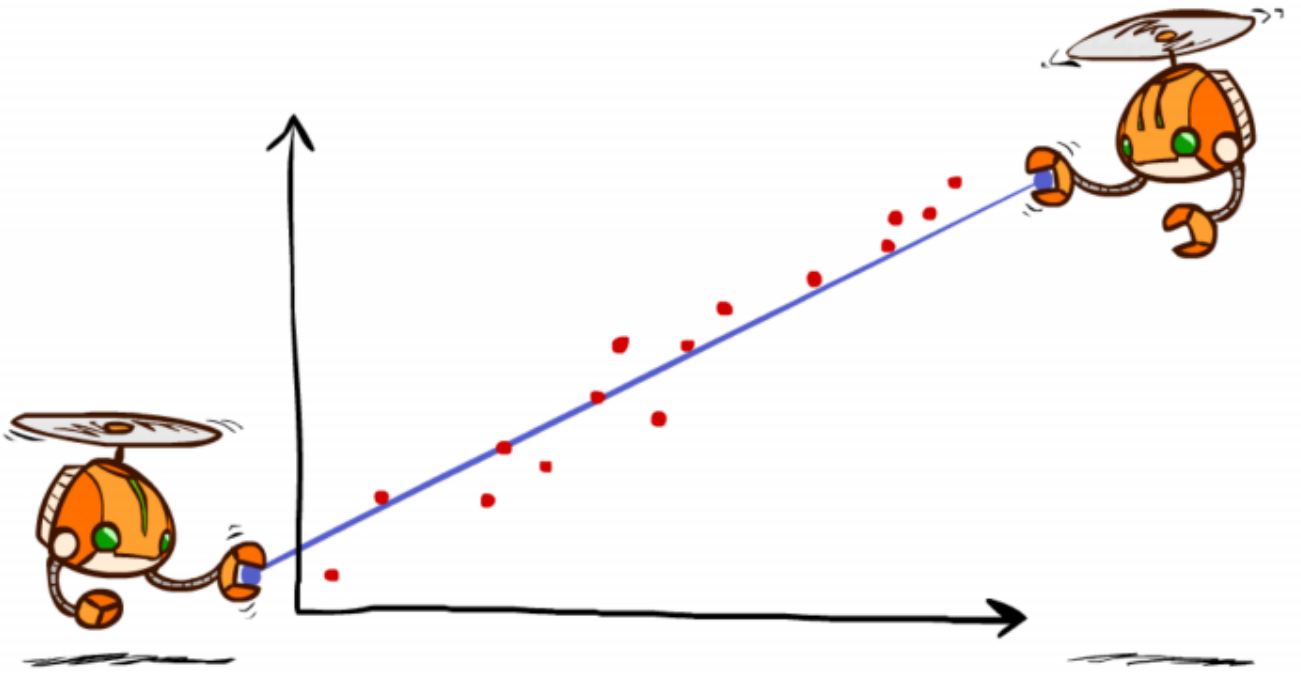


Supervised learning with structured outputs ($y \in \mathcal{Y}$).

—

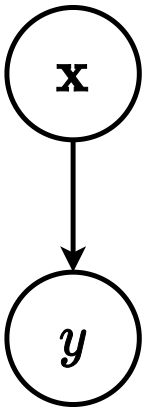
Linear regression

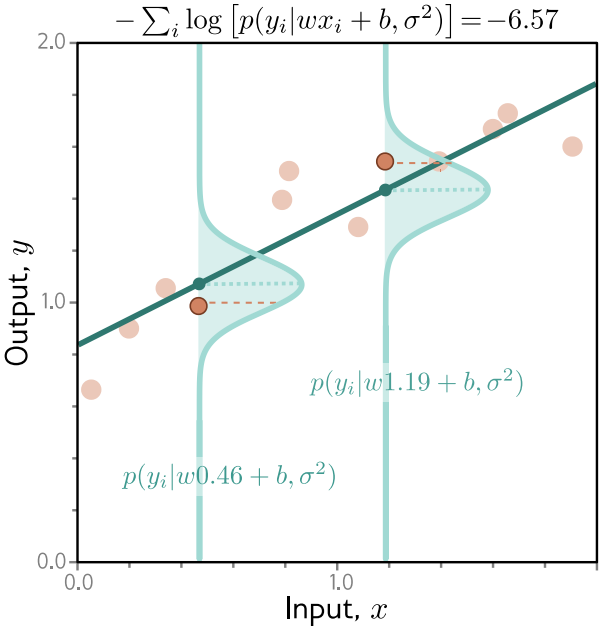
Let us first assume that $y \in \mathbb{R}$.



—

Credits: CS188, UC Berkeley.





Linear regression considers a parameterized linear Gaussian model for its parametric model of $p(y|\mathbf{x})$, that is

$$p(y|\mathbf{x}) = \mathcal{N}(y|\mathbf{w}^T\mathbf{x} + b, \sigma^2),$$

where $\mathbf{w}$ and b are parameters to determine.

Following the principle of maximum likelihood estimation, we maximize

$$p(y|\mathbf{x}) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2}\frac{(y - (\mathbf{w}^T\mathbf{x} + b))^2}{\sigma^2}\right)$$

w.r.t. $\mathbf{w}$ and b over the data $\mathbf{d} = \{(\mathbf{x}_j, y_j)\}$.

Following the principle of maximum likelihood estimation, we maximize

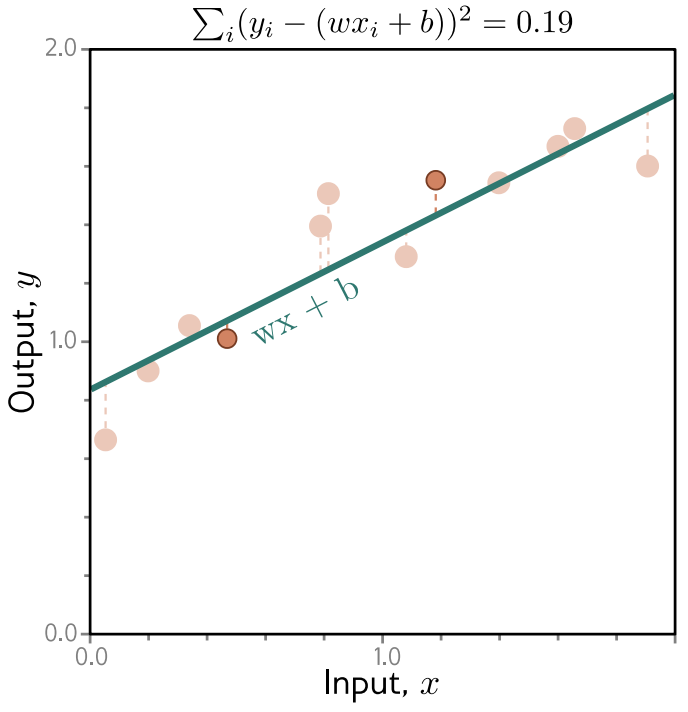
$$p(y|\mathbf{x}) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2}\frac{(y-(\mathbf{w}^T\mathbf{x}+b))^2}{\sigma^2}\right)$$

w.r.t. $\mathbf{w}$ and b over the data $\mathbf{d} = \{(\mathbf{x}_j, y_j)\}$.

After simplification, we arrive to the problem of minimizing

$$\mathcal{L}(\mathbf{w}, b) = \sum_{j=1}^{N}(y_j - (\mathbf{w}^T\mathbf{x}_j + b))^2.$$

Therefore, minimizing the sum of squared errors corresponds to the MLE solution for a linear fit, assuming Gaussian noise of fixed variance.



Minimizing the negative log-likelihood of a linear Gaussian model reduces to minimizing the sum of squared residuals.

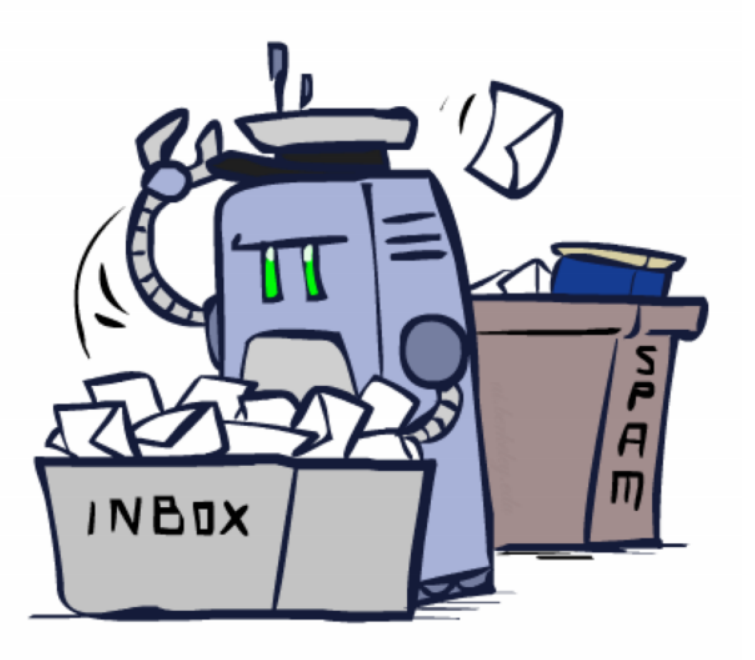
If we absorb the bias term b into the weight vector $\mathbf{w}$ by adding a constant feature $x_0 = 1$ to the input vector $\mathbf{x}$, the solution $\mathbf{w}^*$ is given analytically by

$$\mathbf{w}^* = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y},$$

where $\mathbf{X}$ is the input matrix made of the stacked input vectors $\mathbf{x}_j$ (including the constant feature) and $\mathbf{y}$ is the output vector made of the output values y_j.

Logistic regression

Let us now assume $y \in \{0, 1\}$.

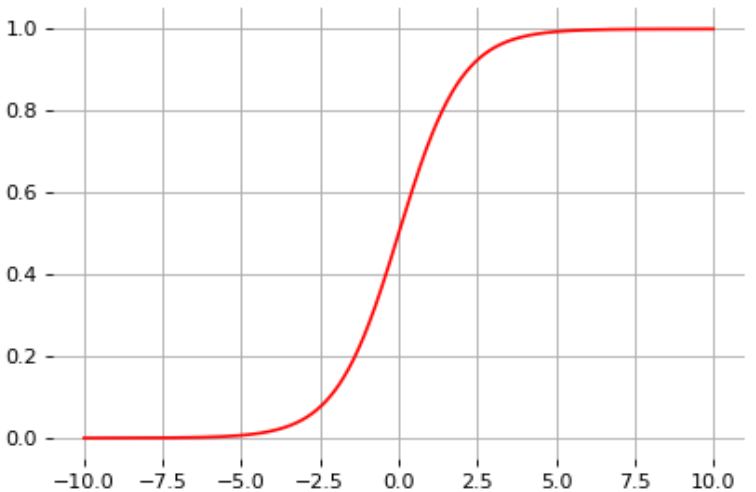


—

Logistic regression models the conditional as

$$P(Y = 1|\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x} + b),$$

where the sigmoid activation function $\sigma(x) = \frac{1}{1+\exp(-x)}$ looks like a soft heavyside:



Following the principle of maximum likelihood estimation, we have

$$
\begin{aligned}
&\arg\max_{\mathbf{w},b} P(\mathbf{d}|\mathbf{w},b) \\
&= \arg\max_{\mathbf{w},b} \prod_{\mathbf{x}_i,y_i\in\mathbf{d}} P(Y=y_i|\mathbf{x}_i,\mathbf{w},b) \\
&= \arg\max_{\mathbf{w},b} \prod_{\mathbf{x}_i,y_i\in\mathbf{d}} \sigma(\mathbf{w}^T\mathbf{x}_i+b)^{y_i}(1-\sigma(\mathbf{w}^T\mathbf{x}_i+b))^{1-y_i} \\
&= \arg\min_{\mathbf{w},b} \underbrace{\sum_{\mathbf{x}_i,y_i\in\mathbf{d}} -y_i\log\sigma(\mathbf{w}^T\mathbf{x}_i+b)-(1-y_i)\log(1-\sigma(\mathbf{w}^T\mathbf{x}_i+b))}_{\mathcal{L}(\mathbf{w},b)=\sum_i \ell(y_i,\hat{y}(\mathbf{x}_i;\mathbf{w},b))}
\end{aligned}
$$

This loss is an estimator of the cross-entropy

$$H(p,q)=\mathbb{E}_p[-\log q]$$

for $p=Y|\mathbf{x}_i$ and $q=\hat{Y}|\mathbf{x}_i$.

Unfortunately, there is no closed-form solution for the MLE of $\mathbf{w}$ and b.

Gradient descent

Let $\mathcal{L}(\theta)$ denote a loss function (e.g., the negative log-likelihood) defined over model parameters θ (e.g., $\mathbf{w}$ and b).

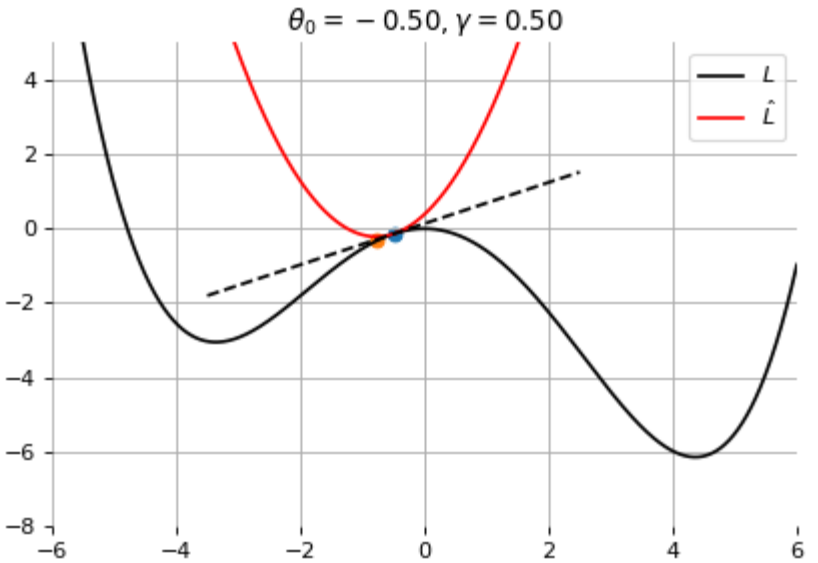
To minimize $\mathcal{L}(\theta)$, gradient descent relies on 1st-order Taylor approximations

$$\mathcal{L}(\theta + \epsilon) \approx \mathcal{L}(\theta) + \epsilon^T \nabla_\theta \mathcal{L}(\theta)$$

and adds a quadratic regularization term to ensure that the approximation is valid only in a neighborhood of θ, leading to the approximation

$$\hat{\mathcal{L}}(\epsilon; \theta) = \mathcal{L}(\theta) + \epsilon^T \nabla_\theta \mathcal{L}(\theta) + \frac{1}{2\gamma} |\epsilon|^2,$$

where $\gamma > 0$ controls the size of the neighborhood.


$\theta_0 = -0.50, \gamma = 0.50$
L
$\hat{L}$
4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6

A minimizer of the approximation $\hat{\mathcal{L}}(\epsilon; \theta_0)$ is given for

$$\begin{aligned} \nabla_\epsilon \hat{\mathcal{L}}(\epsilon; \theta_0) &= 0 \\ &= \nabla_\theta \mathcal{L}(\theta_0) + \frac{1}{\gamma}\epsilon, \end{aligned}$$

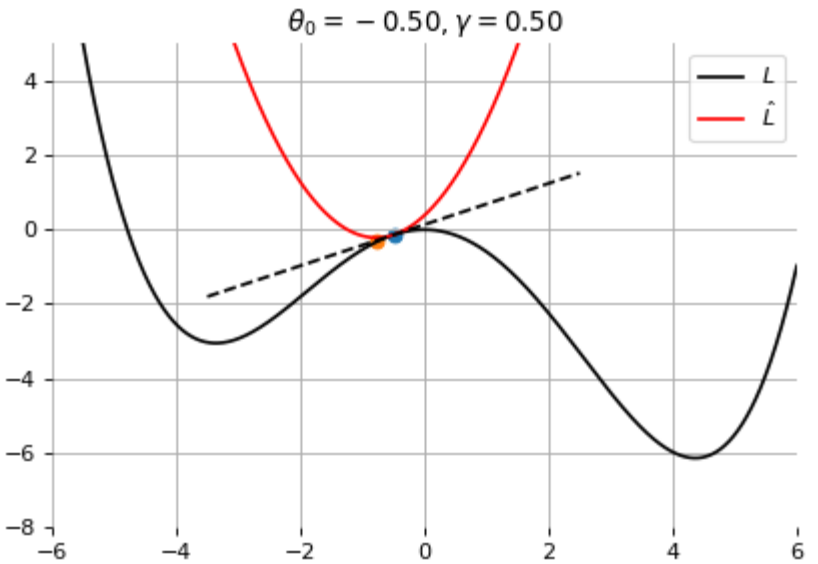
which results in the best improvement for the step $\epsilon = -\gamma \nabla_\theta \mathcal{L}(\theta_0)$.

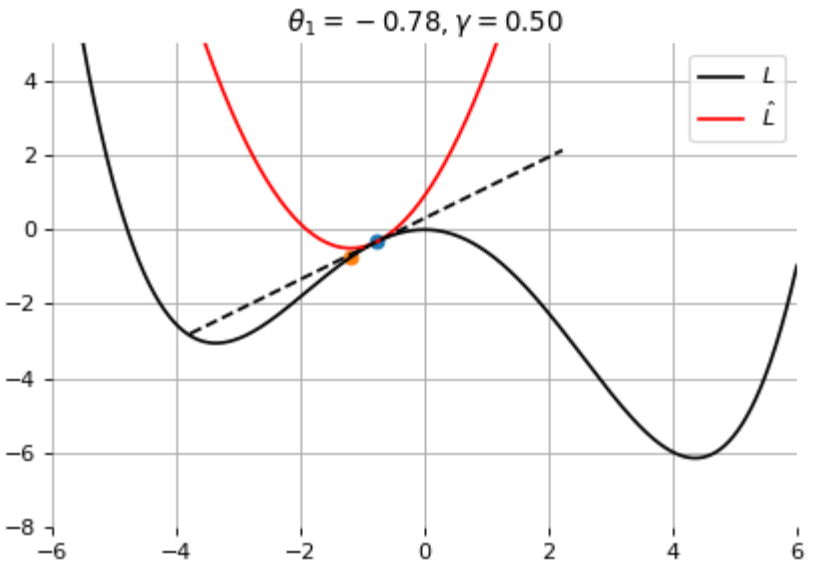
Therefore, model parameters can be updated iteratively using the update rule

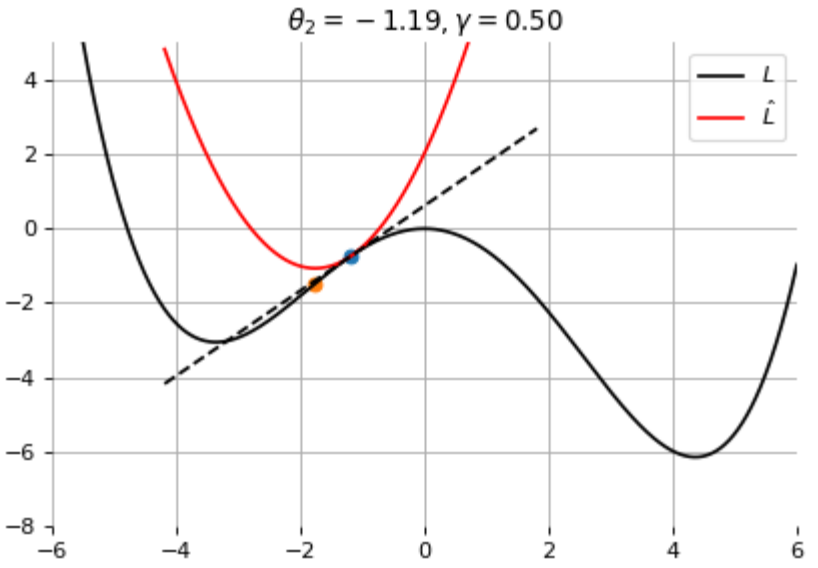
$$\theta_{t+1} = \theta_t - \gamma \nabla_\theta \mathcal{L}(\theta_t),$$

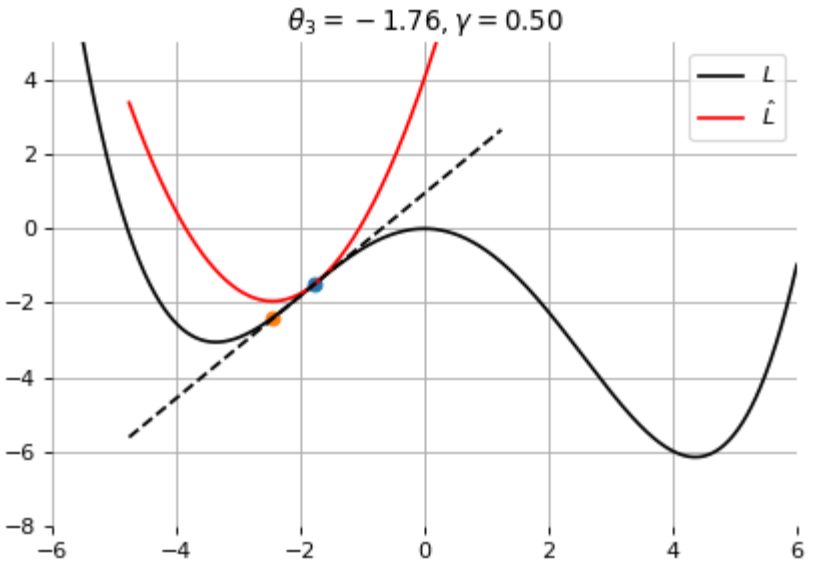
where

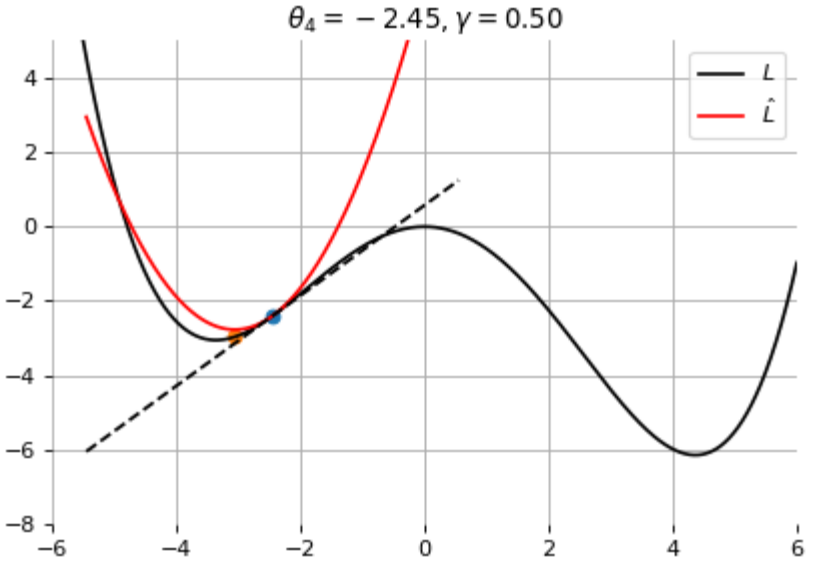
- θ_0 are the initial parameters of the model,
- γ is the learning rate.

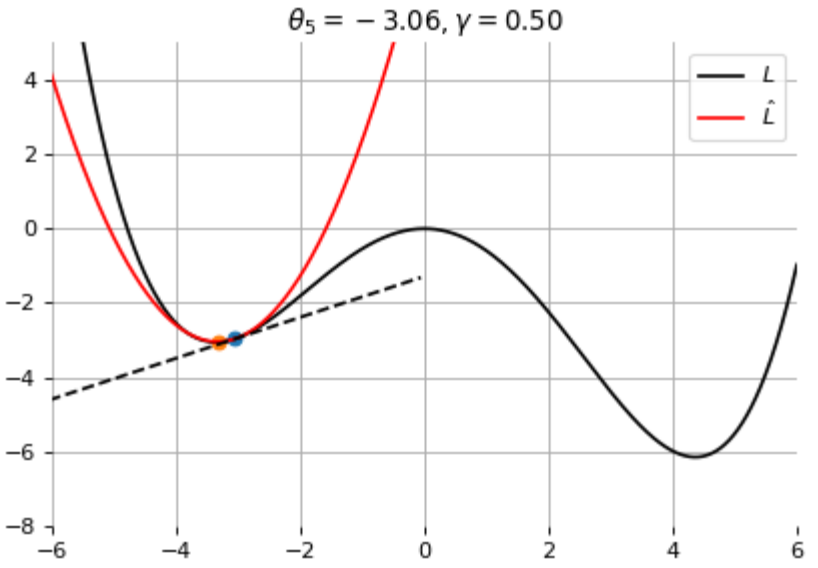

$\theta_0 = -0.50, \gamma = 0.50$
L
$\hat{L}$
4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6

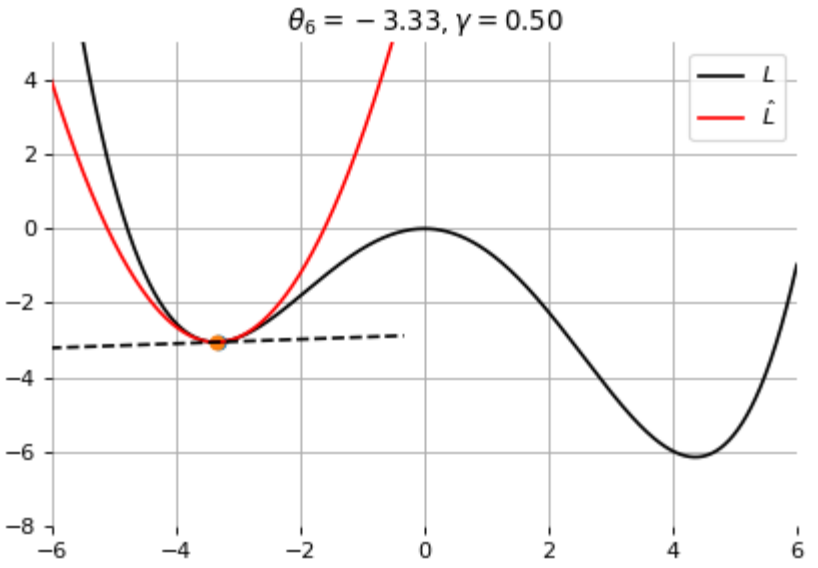

$\theta_1 = -0.78, \gamma = 0.50$
L
$\hat{L}$
4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6

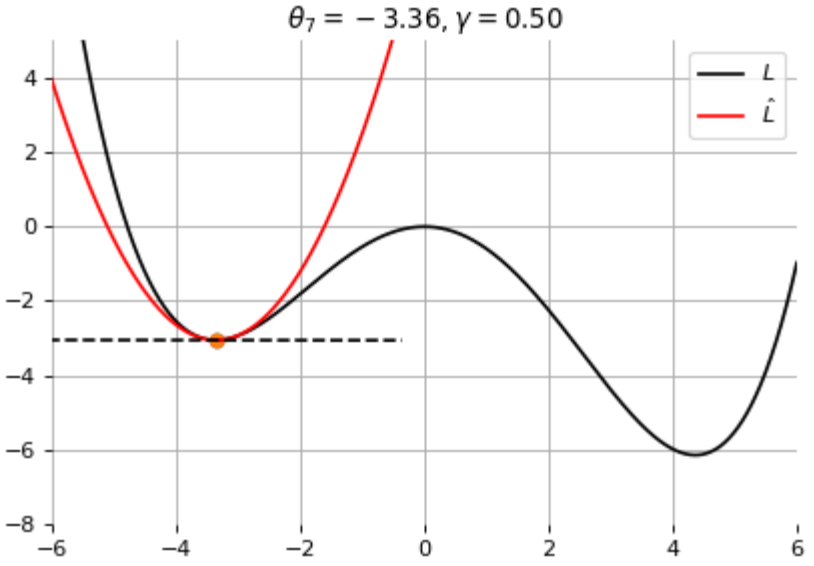

$\theta_2 = -1.19, \gamma = 0.50$
L
$\hat{L}$
4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6


$\theta_3 = -1.76, \gamma = 0.50$
L
$\hat{L}$
4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6


$\theta_4 = -2.45, \gamma = 0.50$
L
$\hat{L}$
4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6


$\theta_5 = -3.06, \gamma = 0.50$
L
$\hat{L}$
4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6


$\theta_6 = -3.33, \gamma = 0.50$
L
$\hat{L}$
4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6

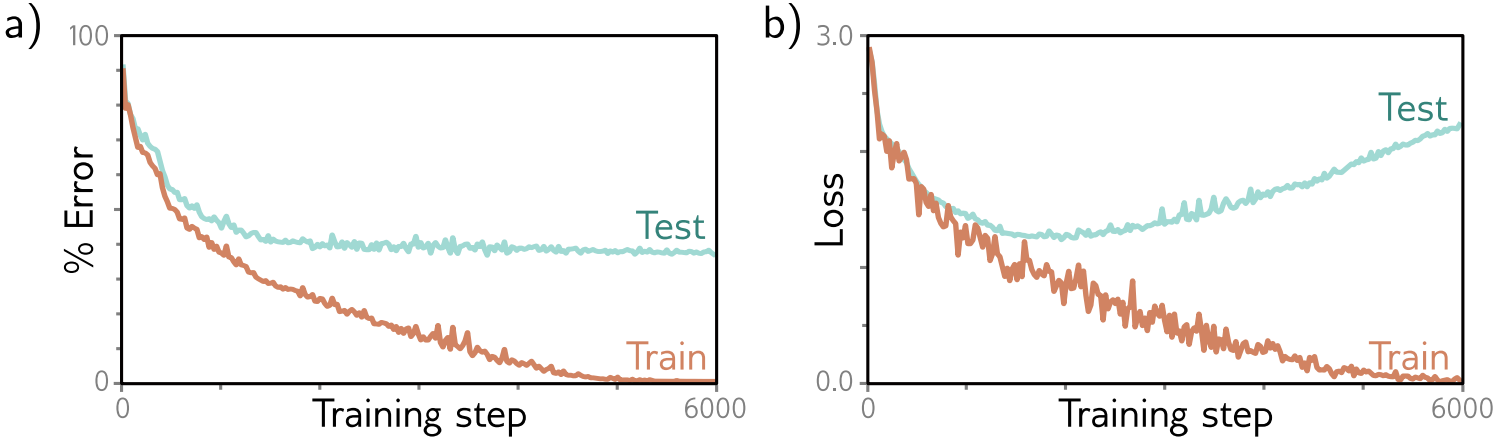

$\theta_7 = -3.36, \gamma = 0.50$
L
$\hat{L}$
4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6

(Step-by-step code example)

Measuring performance

ML models are trained to minimize the loss $\mathcal{L}(\theta)$ on a training dataset $\mathbf{d}_{\text{train}}$. However, a low training loss does not guarantee good performance on unseen data, as the model may have memorized the training data.

To assess its generalization performance, a model should be evaluated on a separate test dataset $\mathbf{d}_{\text{test}}$ that was not used during training. This evaluation provides an estimate of how well the model will perform on new, unseen data.

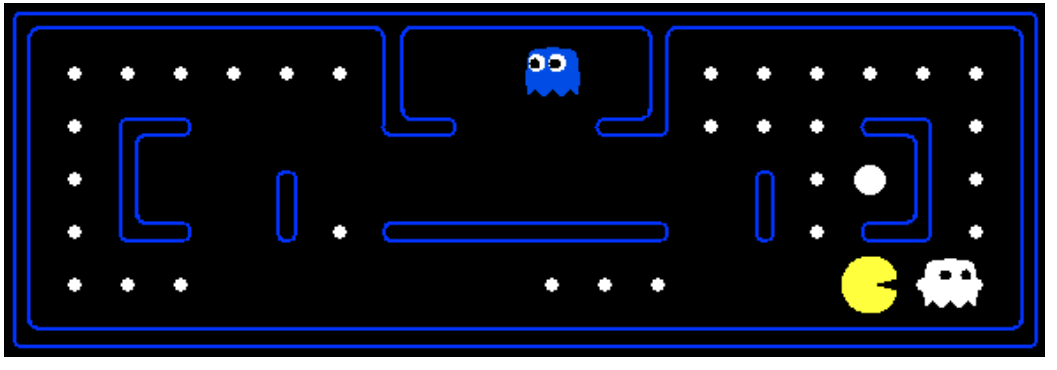


—

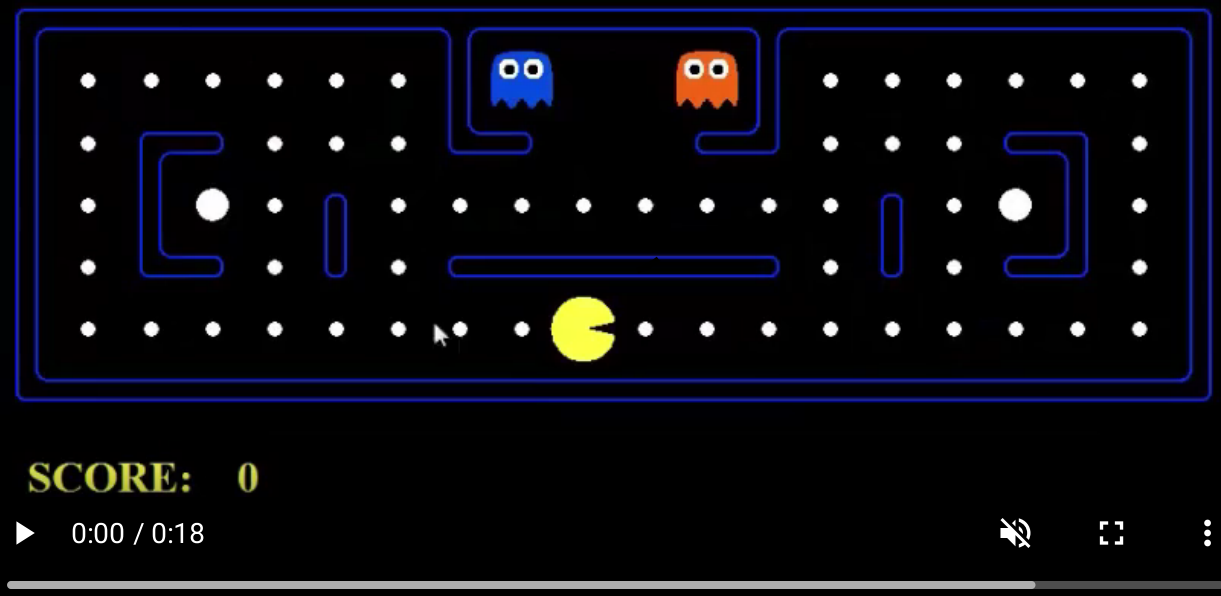
Example: imitation learning in Pacman

Can we learn to play Pacman only from observations?

- Feature vectors $\mathbf{x} = g(s)$ are extracted from the game states s. Output values y corresponds to actions a .
- State-action pairs $(\mathbf{x}, y)$ are collected by observing an expert playing.
- We want to learn the actions that the expert would take in a given situation. That is, learn the mapping $f : \mathbb{R}^d \to \mathcal{A}$.
- This is a multiclass classification problem.



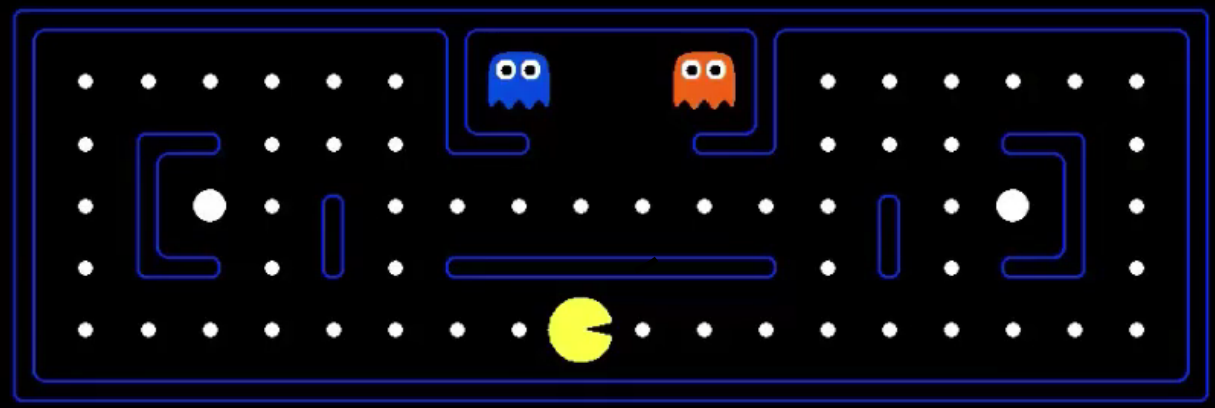
—



The agent observes a very good Minimax-based agent for two games and updates its weight vectors as data are collected.

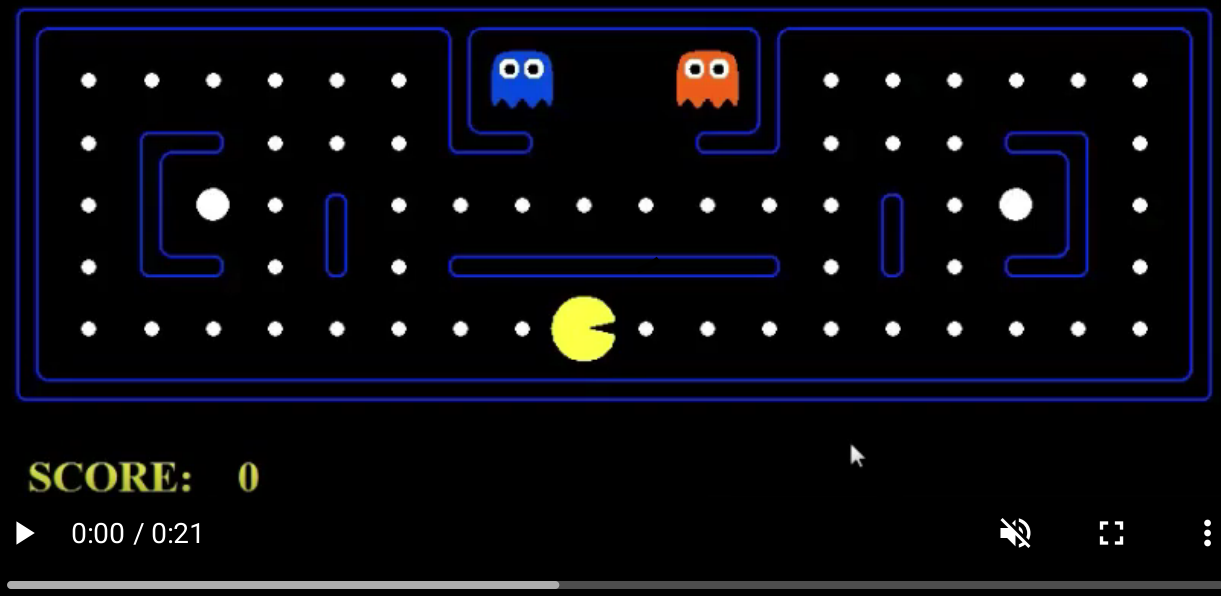
—

Credits: CS188, UC Berkeley.



—

Credits: CS188, UC Berkeley.



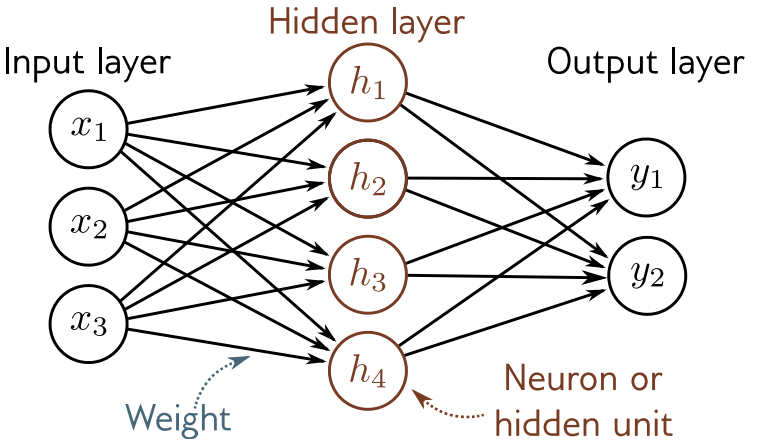
After two training episodes, the ML-based agents plays.
No more Minimax!

—
Credits: CS188, UC Berkeley.

Deep Learning

(a short introduction)

Shallow networks



A shallow network is a function

$$f : \mathbb{R}^{d_{\text{in}}} \rightarrow \mathbb{R}^{d_{\text{out}}}$$

that maps multi-dimensional inputs $\mathbf{x} \in \mathbb{R}^{d_{\text{in}}}$ to multi-dimensional outputs $\mathbf{y} \in \mathbb{R}^{d_{\text{out}}}$ through a hidden layer $\mathbf{h} = [h_0, h_1, ..., h_{q-1}] \in \mathbb{R}^q$.

—

The hidden layer is a non-linear transformation of the input, such that

$$h_j = \sigma\left(\sum_{i=0}^{d_{\text{in}}-1} w_{ji}x_i + b_j\right)$$

for $j = 0, ..., q-1$, where w_{ji} and b_j ($i = 0, ..., d_{\text{in}} - 1, j = 0, ..., q-1$) are the model parameters and σ is an activation function.

The output layer is a linear transformation of the hidden layer, such that

$$y_k = \sum_{j=0}^{q-1} v_{kj}h_j + c_k$$

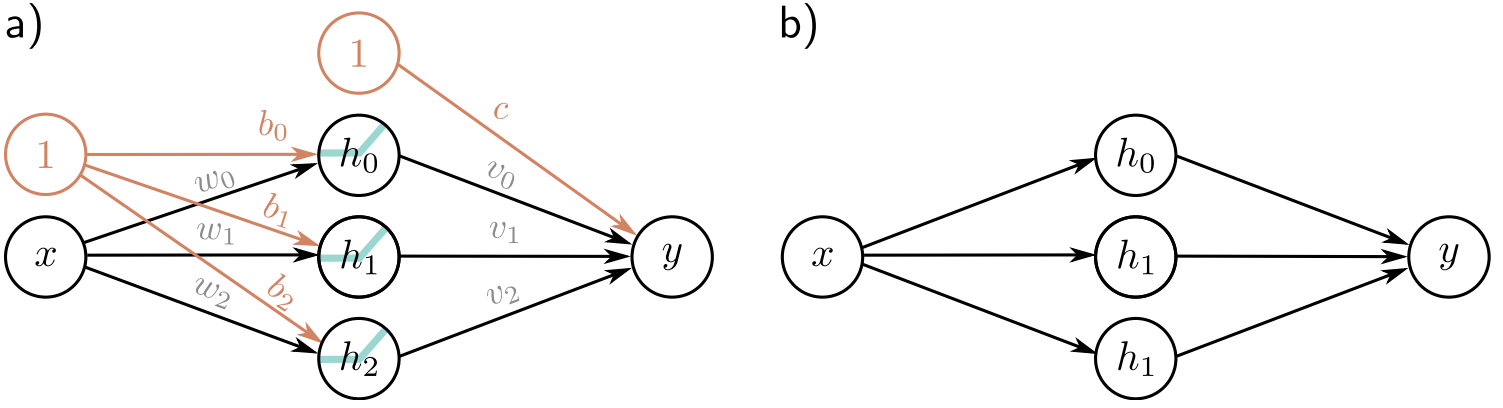
for $k = 0, ..., d_{\text{out}} - 1$, where v_{kj} and c_k ($j = 0, ..., q-1, k = 0, ..., d_{\text{out}} - 1$) are the model parameters.

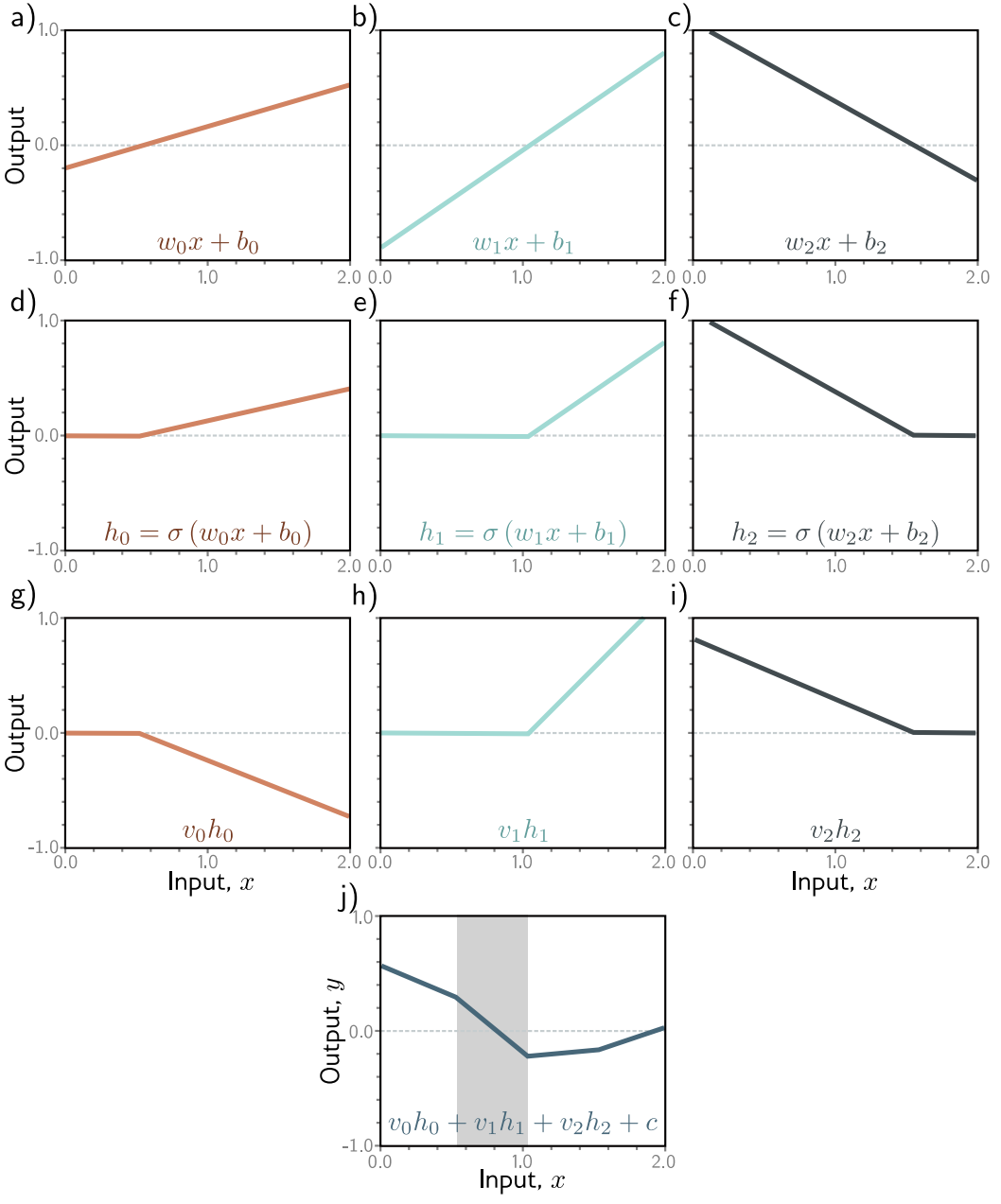
Single-input single-output networks

We first consider the case where $d_{\text{in}} = 1$ and $d_{\text{out}} = 1$ for the single-input single-output network

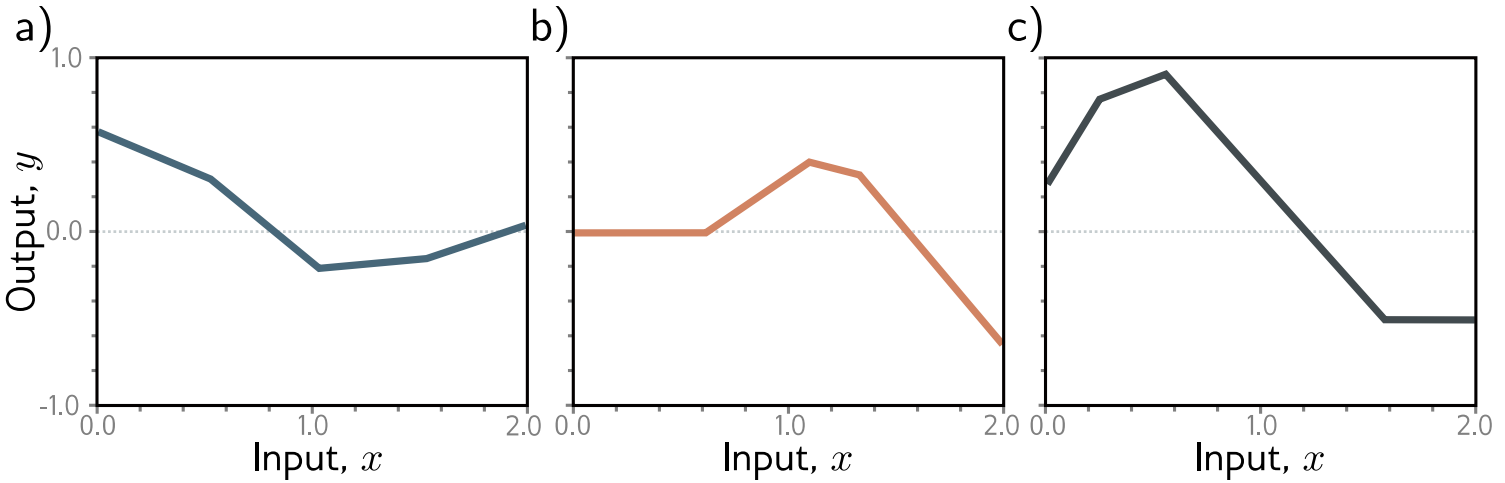
$$y = v_0\sigma(w_0x + b_0) + v_1\sigma(w_1x + b_1) + v_2\sigma(w_2x + b_2) + c$$

where $w_0, w_1, w_2, b_0, b_1, b_2, v_0, v_1, v_2$ and c are the model parameters and where the activation function σ is $\text{ReLU}(\cdot) = \max(0, \cdot)$.




a)
b)
c)
d)
e)
f)
g)
h)
i)
j)
Output
Output, y
Input, x
$w_0x + b_0$
$w_1x + b_1$
$w_2x + b_2$
$h_0 = \sigma(w_0x + b_0)$
$h_1 = \sigma(w_1x + b_1)$
$h_2 = \sigma(w_2x + b_2)$
v_0h_0
v_1h_1
v_2h_2
$v_0h_0 + v_1h_1 + v_2h_2 + c$
1.0
0.0
-1.0
0.0
1.0
2.0

—

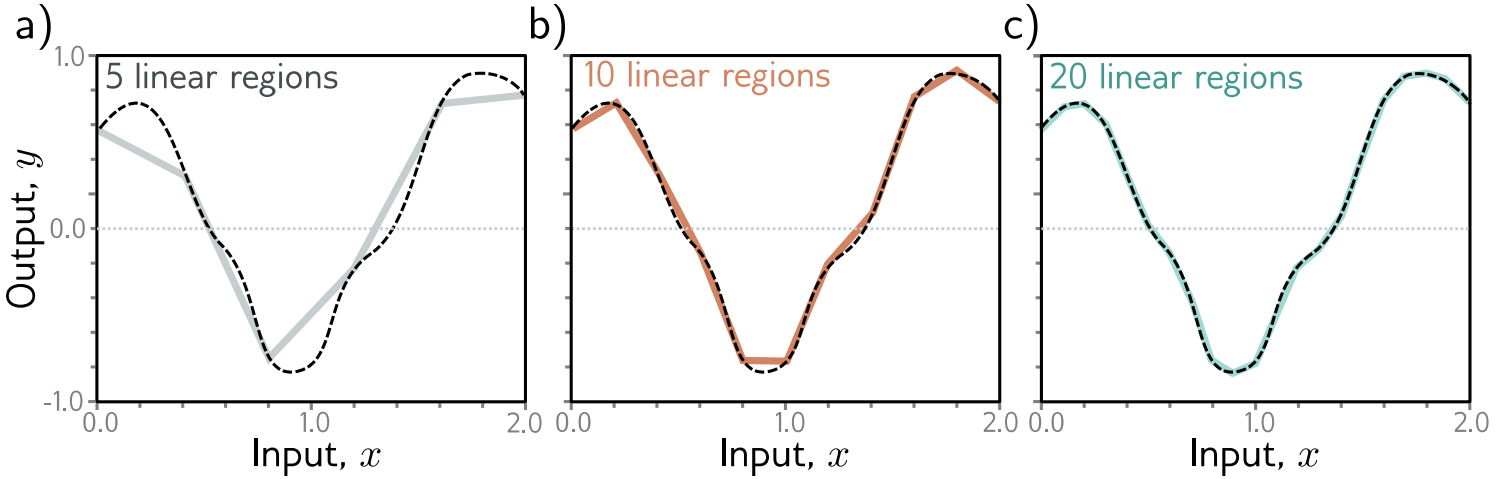


This shallow network defines a family of piecewise linear functions where the positions of the joints, the slopes and the heights of the functions are determined by the 10 parameters w_0, w_1, w_2, b_0, b_1, b_2, v_0, v_1, v_2 and c.

Universal approximation theorem

The universal approximation theorem states that any continuous function can be approximated arbitrarily well by a shallow network with sufficiently many hidden units.

For example, for 1d input and 1d output, any continuous function $f : \mathbb{R} \rightarrow \mathbb{R}$ can be approximated to arbitrary accuracy by a piecewise linear function with enough linear segments, which can be represented by a shallow network with enough hidden units.

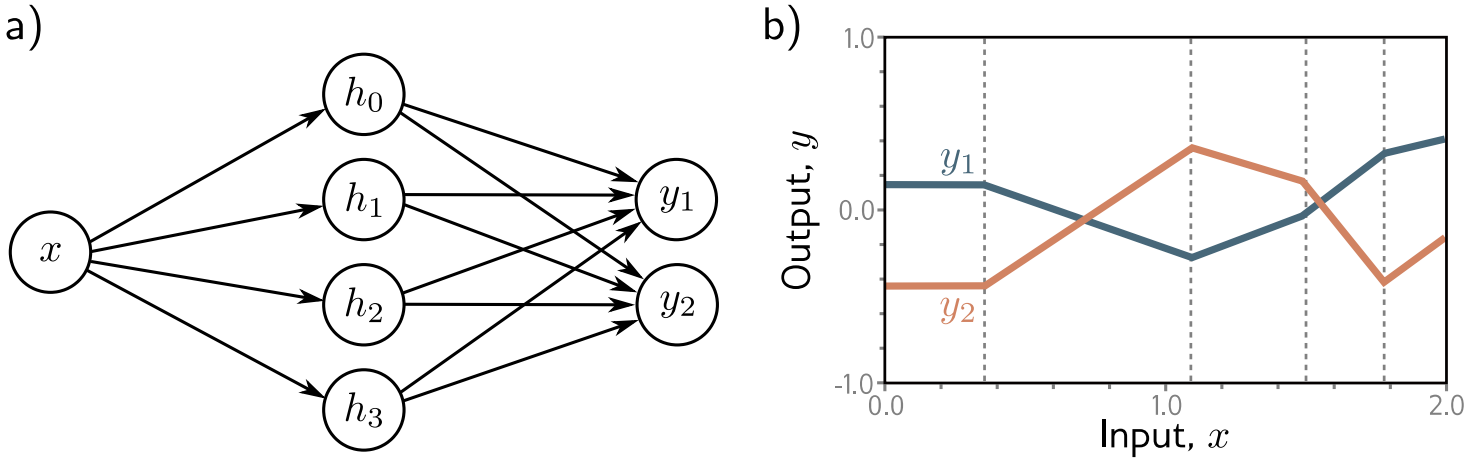

a)
5 linear regions
b)
10 linear regions
c)
20 linear regions
Output, y
1.0
0.0
-1.0
0.0
1.0
2.0
Input, x

Multivariate outputs

To extend the network to multivariate outputs $\mathbf{y} = [y_0, y_1, .., y_{d_{\text{out}}-1}]$, we simply add more output units as linear combinations of the hidden units.

For example, a network with two output units y_0 and y_1 might have the following structure:

$$
\begin{aligned}
h_0 &= \sigma\left(w_0 x + b_0\right) \\
h_1 &= \sigma\left(w_1 x + b_1\right) \\
h_2 &= \sigma\left(w_2 x + b_2\right) \\
h_3 &= \sigma\left(w_3 x + b_3\right) \\
y_0 &= v_{00} h_0 + v_{01} h_1 + v_{02} h_2 + v_{03} h_3 + c_0 \\
y_1 &= v_{10} h_0 + v_{11} h_1 + v_{12} h_2 + v_{13} h_3 + c_1
\end{aligned}
$$

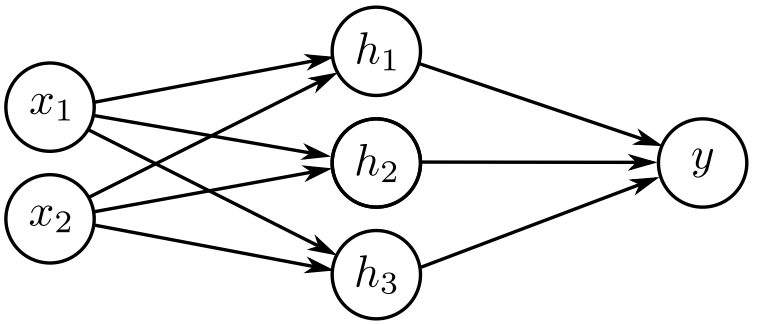


a) With two output units, the network can model two functions of the input $\boldsymbol{x}$.

b) The four joints of these functions are constrained to be at the same positions, but the slopes and heights of the functions can vary independently.

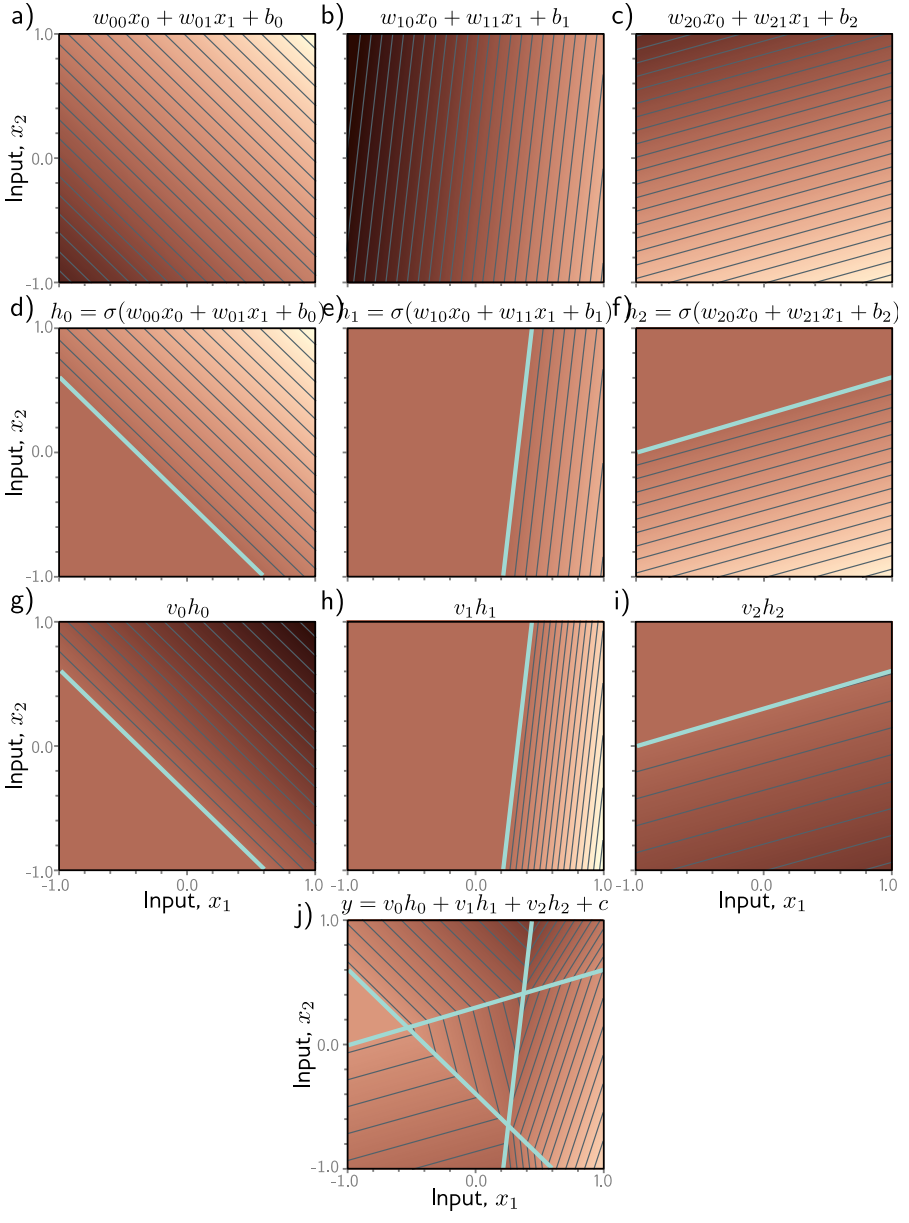
Multivariate inputs

To extend the network to multivariate inputs $\mathbf{x} = [x_0, x_1, \ldots, x_{d_{\text{in}}-1}]$, we extend the linear relations between the input and the hidden units.



For example, a network with two inputs $\mathbf{x} = [x_0, x_1]$ might have three hidden units h_0, h_1 and h_2 defined as

$$
\begin{aligned}
h_0 &= \sigma\left(w_{00}x_0 + w_{01}x_1 + b_0\right) \\
h_1 &= \sigma\left(w_{10}x_0 + w_{11}x_1 + b_1\right) \\
h_2 &= \sigma\left(w_{20}x_0 + w_{21}x_1 + b_2\right).
\end{aligned}
$$

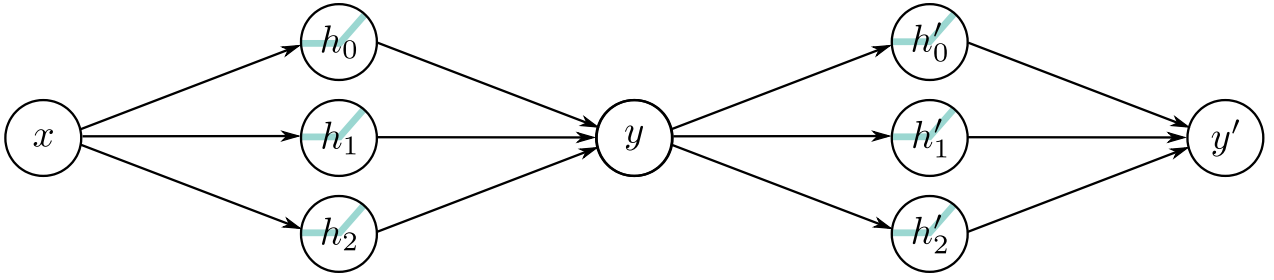

a) $w_{00}x_0 + w_{01}x_1 + b_0$
b) $w_{10}x_0 + w_{11}x_1 + b_1$
c) $w_{20}x_0 + w_{21}x_1 + b_2$
d) $h_0 = \sigma(w_{00}x_0 + w_{01}x_1 + b_0)$
e) $h_1 = \sigma(w_{10}x_0 + w_{11}x_1 + b_1)$
f) $h_2 = \sigma(w_{20}x_0 + w_{21}x_1 + b_2)$
g) v_0h_0
h) v_1h_1
i) v_2h_2
j) $y = v_0h_0 + v_1h_1 + v_2h_2 + c$
Input, x_1
Input, x_2
1.0
0.0
-1.0

—

Deep networks

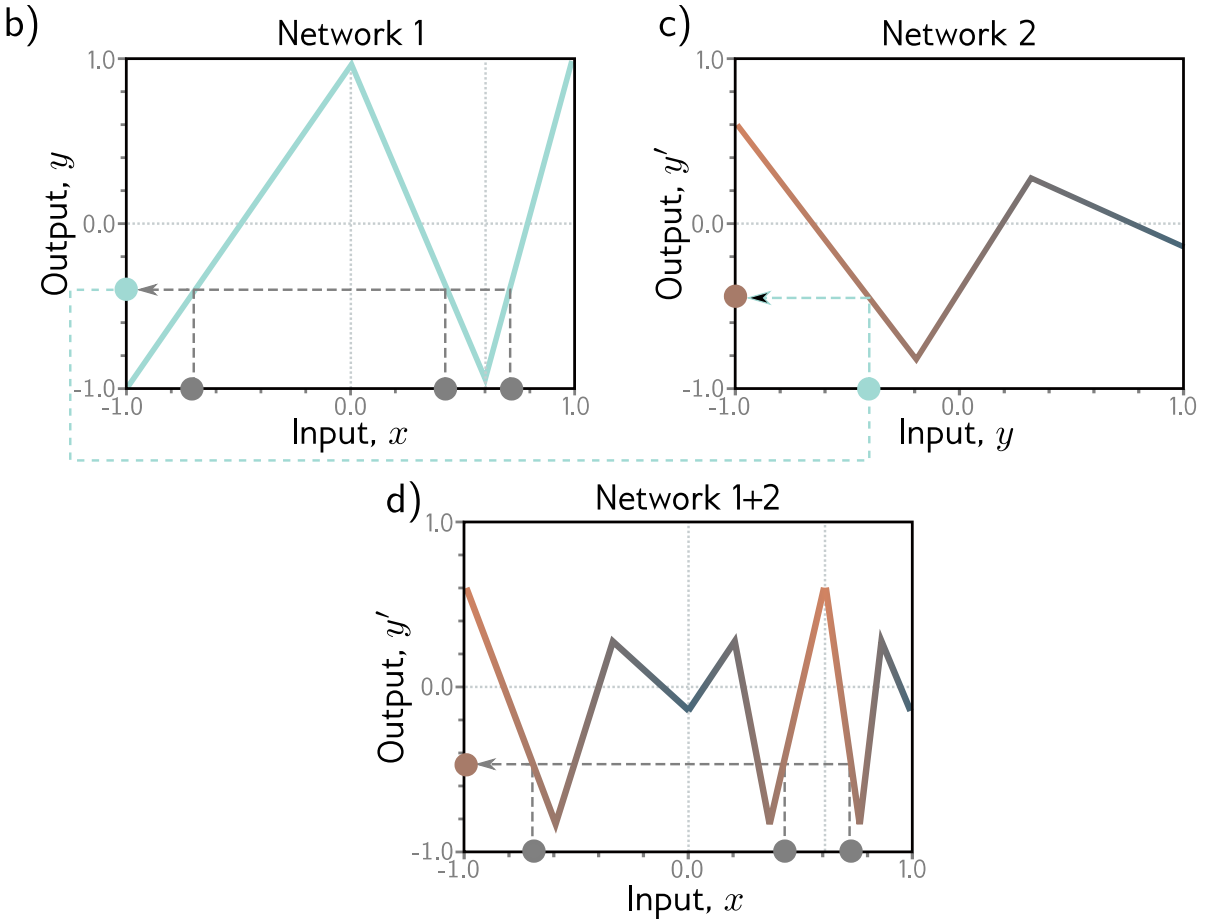
We first consider the composition of two shallow networks, where the output of the first network is fed as input to the second network as

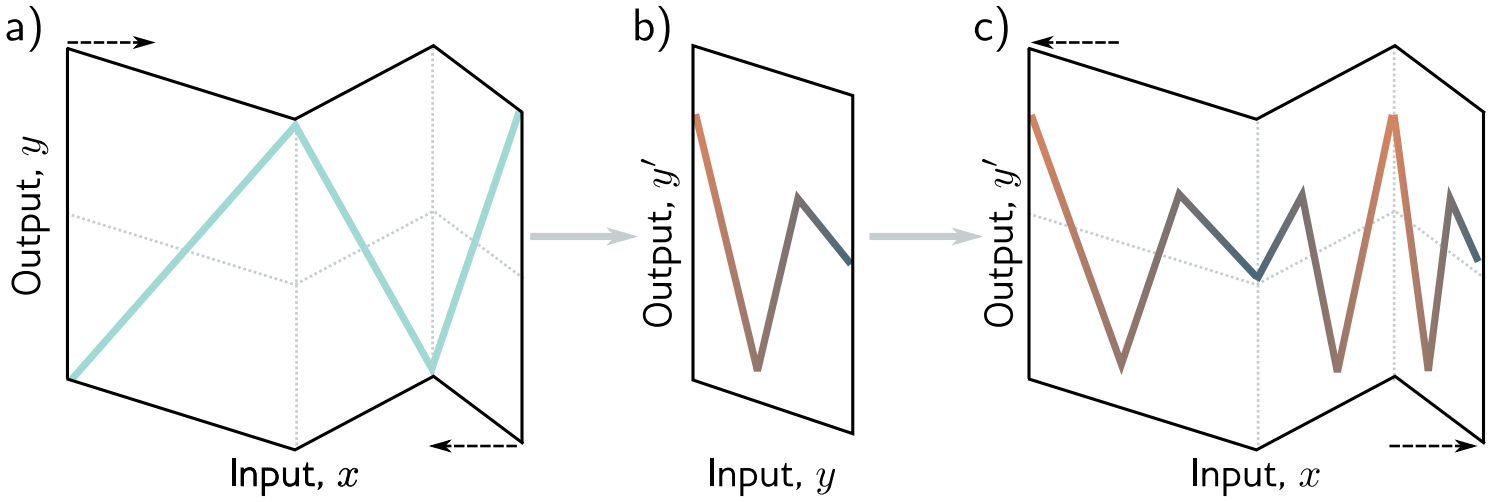
$$
\begin{aligned}
h_0 &= \sigma\left(w_0 x + b_0\right) \\
h_1 &= \sigma\left(w_1 x + b_1\right) \\
h_2 &= \sigma\left(w_2 x + b_2\right) \\
y &= v_0 h_0 + v_1 h_1 + v_2 h_2 + c \\
h'_0 &= \sigma\left(w'_0 y + b'_0\right) \\
h'_1 &= \sigma\left(w'_1 y + b'_1\right) \\
h'_2 &= \sigma\left(w'_2 y + b'_2\right) \\
y' &= v'_0 h'_0 + v'_1 h'_1 + v'_2 h'_2 + c'.
\end{aligned}
$$



—

With $\mathbf{ReLU}$ activation functions, this network also describes a family of piecewise linear functions. However, each linear region defined by the hidden units of the first network is further divided by the hidden units of the second network.

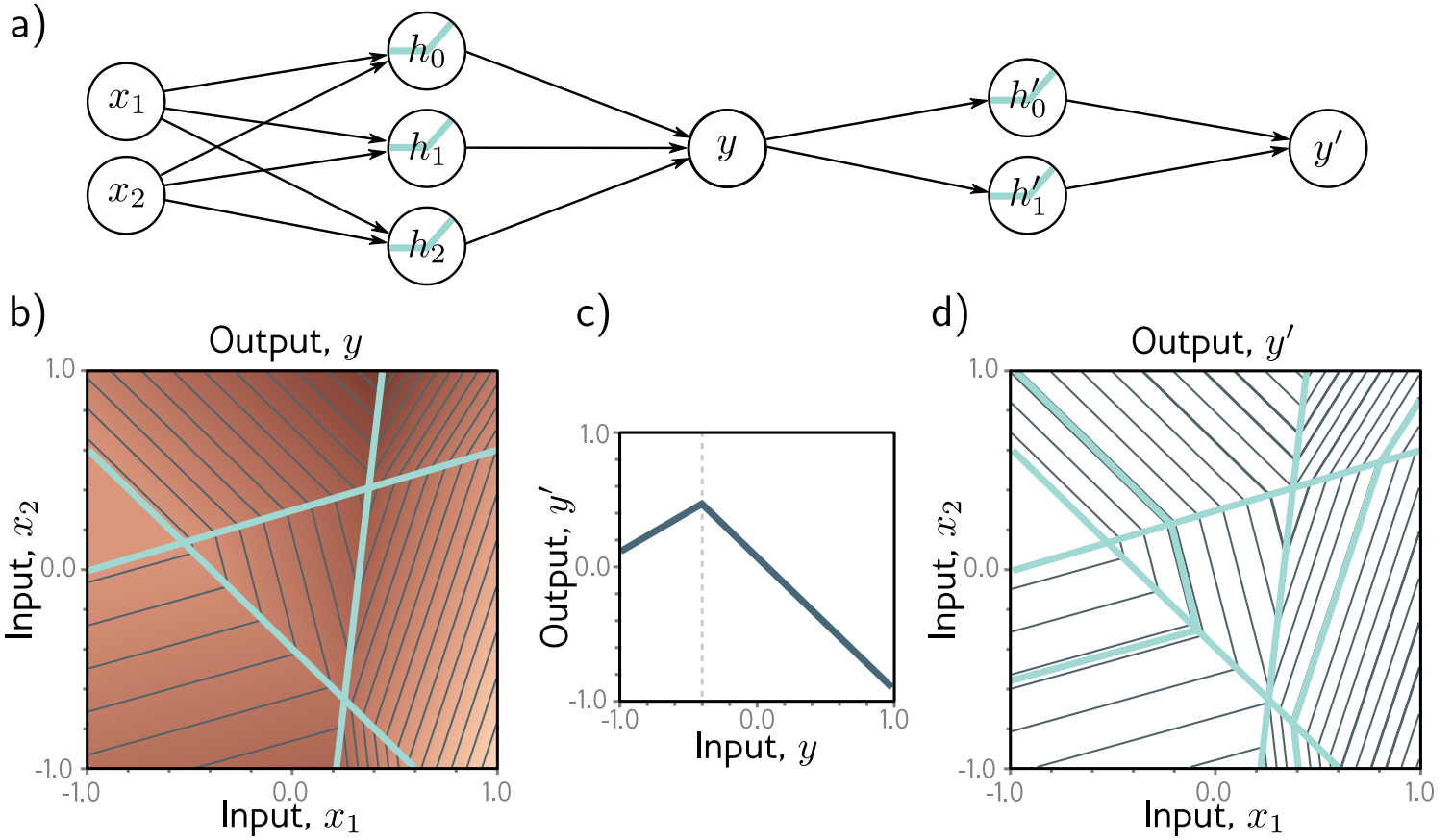




The composition of two shallow networks can be interpreted as a folding operation in three steps:

a) The first network folds the input space back on itself.
b) The second network applies its function to the folded space.
c) The final output is revealed by unfolding the folded space.

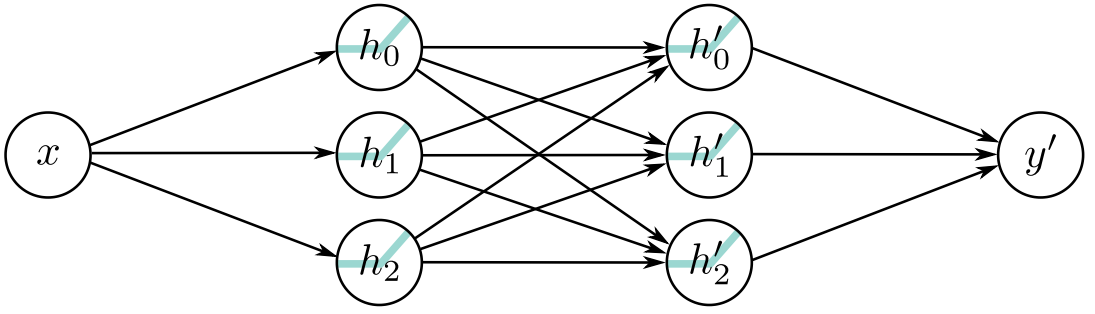
Similarly, composing a multivariate shallow network with a shallow network further divides the input space into more linear regions.



From composing shallow networks to deep networks

Since the operation from $[h_0, h_1, h_2]$ to y is linear and the operation from y to $[h'_0, h'_1, h'_2]$ is also linear, their composition in series is linear.

It follows that the composition of the two shallow networks is a special case of a deep network with two hidden layers.



Mathematically, the first layer is defined as

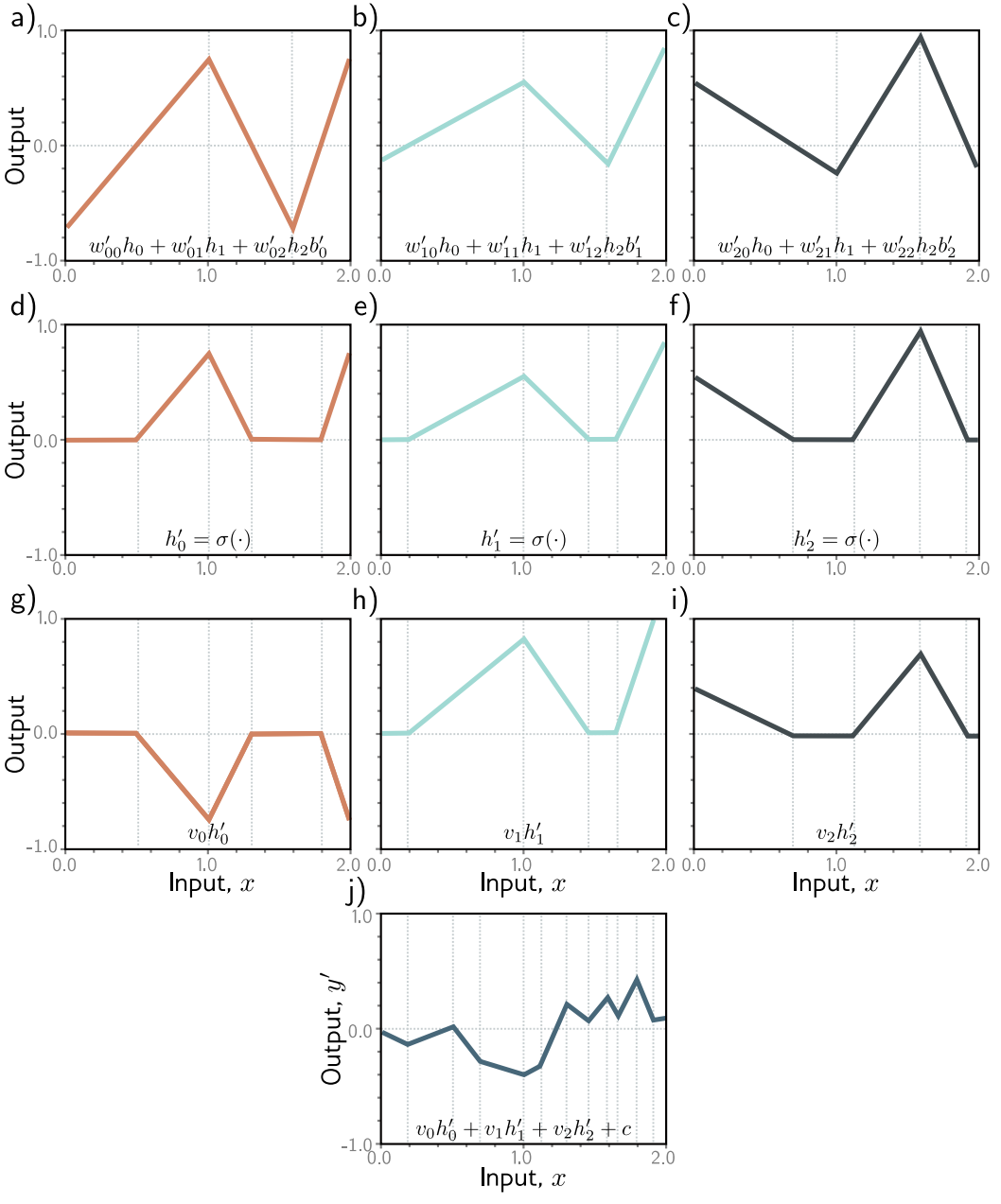
$$\begin{aligned} h_0 &= \sigma\left(w_0 x + b_0\right) \\ h_1 &= \sigma\left(w_1 x + b_1\right) \\ h_2 &= \sigma\left(w_2 x + b_2\right), \end{aligned}$$

while the second layer is defined from the outputs of the first layer as

$$\begin{aligned} h'_0 &= \sigma\left(w'_{00} h_0 + w'_{01} h_1 + w'_{02} h_2 + b'_0\right) \\ h'_1 &= \sigma\left(w'_{10} h_0 + w'_{11} h_1 + w'_{12} h_2 + b'_1\right) \\ h'_2 &= \sigma\left(w'_{20} h_0 + w'_{21} h_1 + w'_{22} j_2 + b'_2\right), \end{aligned}$$

and the output as

$$y = v_0 h'_0 + v_1 h'_1 + v_2 h'_2 + c.$$


a)
b)
c)
d)
e)
f)
g)
h)
i)
j)
Output
Output, y'
Input, x
$w'_{00}h_0 + w'_{01}h_1 + w'_{02}h_2b'_0$
$w'_{10}h_0 + w'_{11}h_1 + w'_{12}h_2b'_1$
$w'_{20}h_0 + w'_{21}h_1 + w'_{22}h_2b'_2$
$h'_0 = \sigma(\cdot)$
$h'_1 = \sigma(\cdot)$
$h'_2 = \sigma(\cdot)$
$v_0h'_0$
$v_1h'_1$
$v_2h'_2$
$v_0h'_0 + v_1h'_1 + v_2h'_2 + c$
1.0
0.0
-1.0
0.0
1.0
2.0

—

Multilayer perceptron

We first note that the computation of a hidden layer with q hidden units can be written in matrix form as

$$\begin{aligned}\mathbf{h} = \begin{bmatrix} h_0 \\ h_1 \\ \vdots \\ h_{q-1} \end{bmatrix} &= \sigma\left(\begin{bmatrix} w_{00} & w_{01} & \cdots & w_{0(d_{\text{in}}-1)} \\ w_{10} & w_{11} & \cdots & w_{1(d_{\text{in}}-1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{(q-1)0} & w_{(q-1)1} & \cdots & w_{(q-1)(d_{\text{in}}-1)} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{d_{\text{in}}-1} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{q-1} \end{bmatrix} \right) \\ &= \sigma(\mathbf{W}^T \mathbf{x} + \mathbf{b})\end{aligned}$$

where $\mathbf{x} \in \mathbb{R}^{d_{\text{in}}}$ is the input vector, $\mathbf{W} \in \mathbb{R}^{d_{\text{in}} \times q}$ is the weight matrix of the hidden layer and $\mathbf{b} \in \mathbb{R}^q$ is the bias vector.

Using this notation, layers can be composed in series to form a deep network with K hidden layers such that

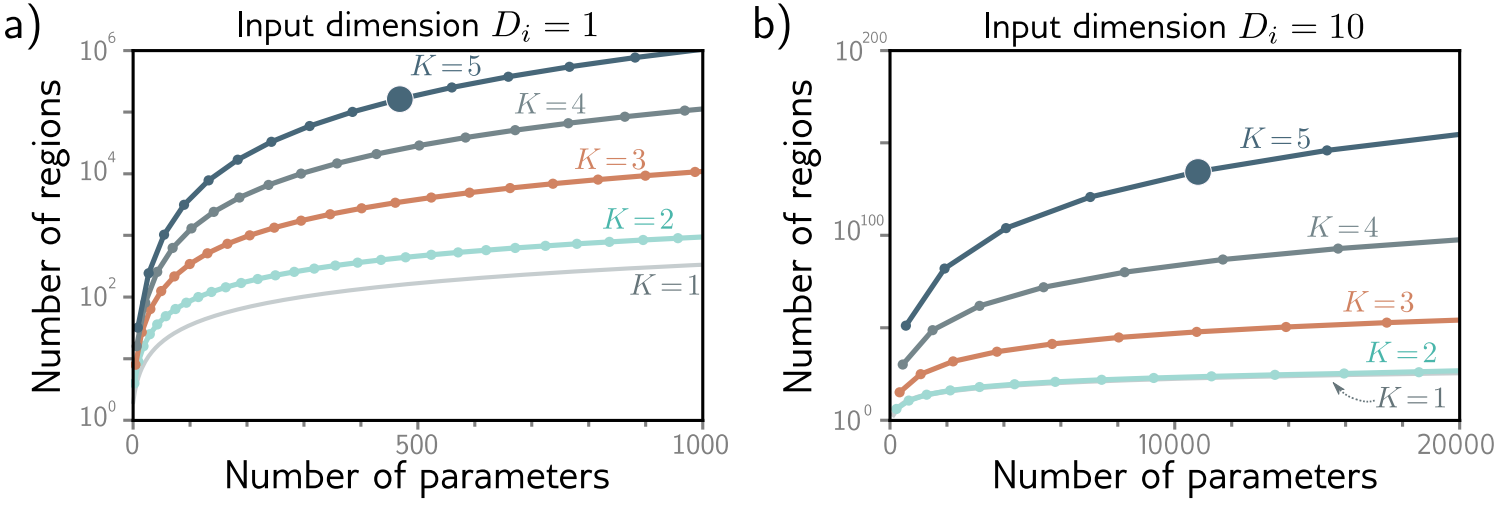
$$
\begin{aligned}
\mathbf{h}_0 &= \mathbf{x} \\
\mathbf{h}_1 &= \sigma(\mathbf{W}_1^T \mathbf{h}_0 + \mathbf{b}_1) \\
\mathbf{h}_2 &= \sigma(\mathbf{W}_2^T \mathbf{h}_1 + \mathbf{b}_2) \\
&\vdots \\
\mathbf{h}_K &= \sigma(\mathbf{W}_K^T \mathbf{h}_{K-1} + \mathbf{b}_K) \\
\mathbf{y} &= \sigma(\mathbf{W}_y^T \mathbf{h}_K + \mathbf{b}_y)
\end{aligned}
$$

where $\mathbf{W}_k \in \mathbb{R}^{q_{k-1} \times q_k}$ is the weight matrix of the k-th hidden layer and $\mathbf{b}_k \in \mathbb{R}^{q_k}$ is the bias vector of the k-th layer.

The resulting function $f(\mathbf{x}; \theta)$ is a highly non-linear mapping from inputs $\mathbf{x}$ to outputs $\mathbf{y}$, where θ denotes the set of all model parameters. This architecture is known as the **multilayer perceptron**.

A shallow network with one input, one output, and q hidden units can create up to $q+1$ linear regions and is defined by $O(q)$ parameters.

A deep network with K hidden layers, each with q hidden units, can create up to $(q+1)^K$ linear regions and is defined by $O(Kq^2)$ parameters.



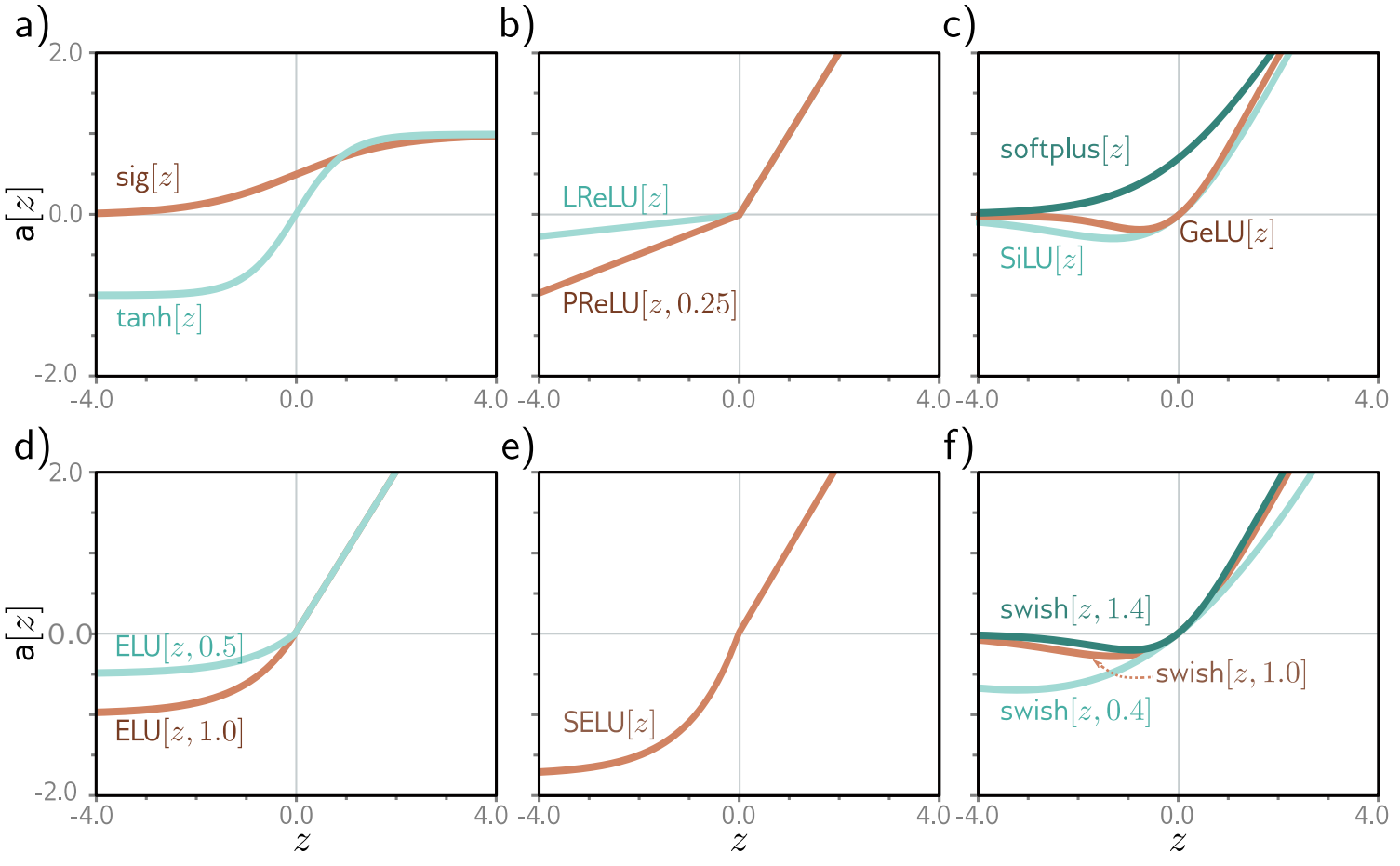
Activation functions

Activation functions $\sigma : \mathbb{R} \to \mathbb{R}$ introduce non-linearities in the network, enabling the model to learn complex functions.

Common activation functions include:

- Sigmoid: $\sigma(x) = \frac{1}{1+\exp(-x)}$, which maps inputs to the range $(0, 1)$.
- Hyperbolic tangent (tanh): $\sigma(x) = \tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$, which maps inputs to the range $(-1, 1)$.
- Rectified Linear Unit (ReLU): $\sigma(x) = \max(0, x)$, which introduces sparsity and mitigates the vanishing gradient problem.

The choice of the activation function σ is crucial for the expressiveness of the network and the optimization of the model parameters.


a)
b)
c)
d)
e)
f)
a[z]
z
sig[z]
tanh[z]
LReLU[z]
PReLU[z, 0.25]
softplus[z]
GeLU[z]
SiLU[z]
ELU[z, 0.5]
ELU[z, 1.0]
SELU[z]
swish[z, 1.4]
swish[z, 1.0]
swish[z, 0.4]
2.0
0.0
-2.0
-4.0
0.0
4.0

Output layers $\mathbf{y} = \sigma(\mathbf{W}_y^T \mathbf{h}_K + \mathbf{b}_y)$

- For regression, the width q of the output layer is set to the dimensionality of the output d_{out} and the activation function is the identity $\sigma(\cdot) = \cdot$.
- For binary classification, the width q of the output layer is set to 1 and the activation function is the sigmoid $\sigma(\cdot) = \frac{1}{1+\exp(-\cdot)}$, which results in a single output that models the probability $p(y = 1|\mathbf{x})$.
- For multi-class classification, the sigmoid activation σ in the output layer can be generalized to produce a vector $\mathbf{y} \in \triangle^C$ of probability estimates $p(Y = i|\mathbf{x})$. This activation is the $\mathrm{Softmax}$ function, where its i-th output is defined as

$$\mathrm{Softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_{j=1}^{C} \exp(z_j)},$$

for $i = 1, ..., C$.

Loss functions

The parameters (e.g., $\mathbf{W}_k$ and $\mathbf{b}_k$ for each layer k) of a deep network $f(\mathbf{x};\theta)$ are learned by minimizing a loss function $\mathcal{L}(\theta)$ over a dataset $\mathbf{d} = \{(\mathbf{x}_j, \mathbf{y}_j)\}$ of input-output pairs.

The loss function is derived from the likelihood:

- For regression, assuming a Gaussian likelihood, the loss is the mean squared error $\mathcal{L}(\theta) = \frac{1}{N} \sum_{(\mathbf{x}_j,\mathbf{y}_j)\in\mathbf{d}} (\mathbf{y}_j - f(\mathbf{x}_j;\theta))^2$.
- For classification, assuming a categorical likelihood, the loss is the cross-entropy $\mathcal{L}(\theta) = -\frac{1}{N} \sum_{(\mathbf{x}_j,\mathbf{y}_j)\in\mathbf{d}} \sum_{i=1}^{C} y_{ij} \log f_i(\mathbf{x}_j;\theta)$.

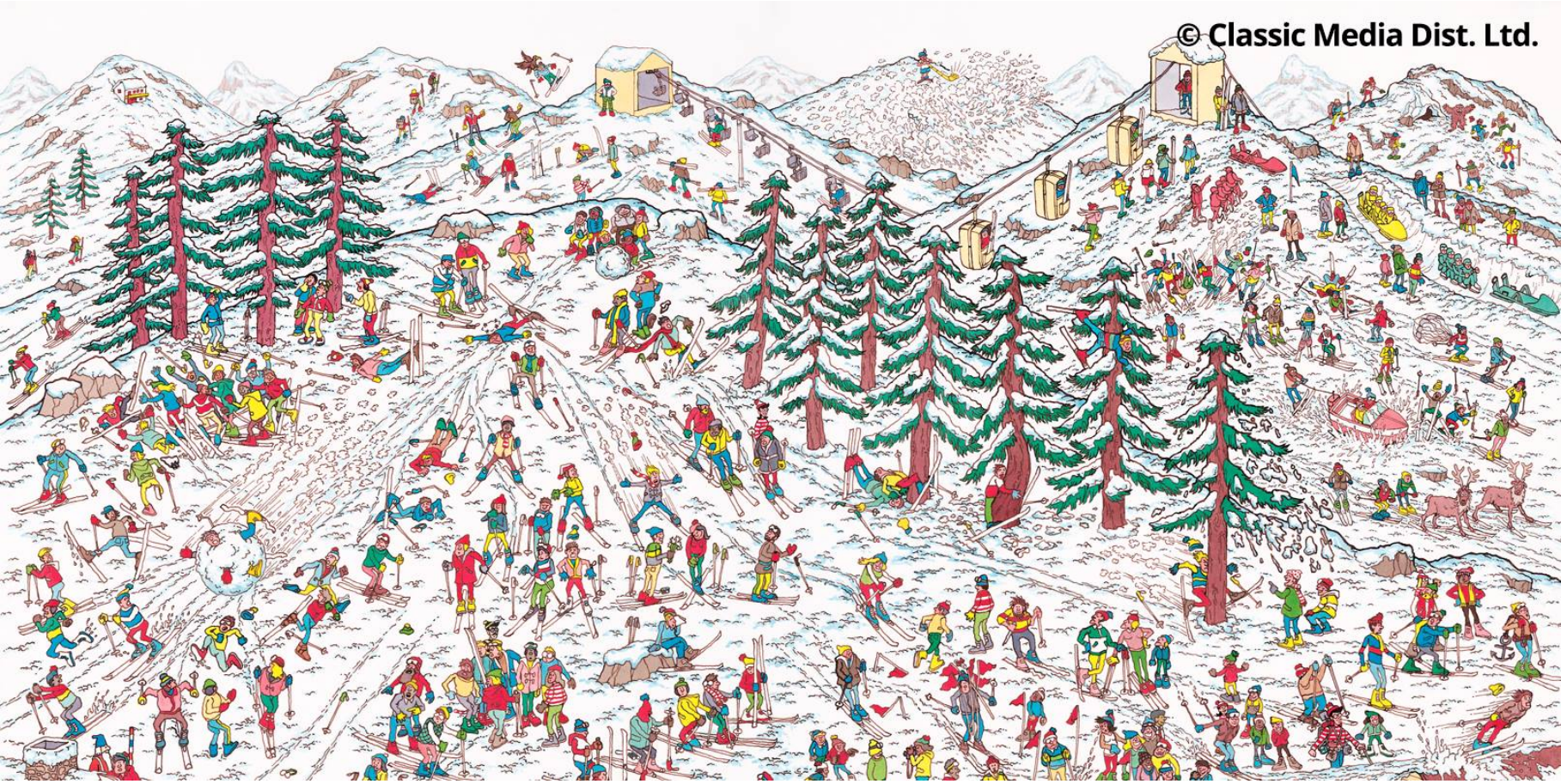
(Step-by-step code example)

Convolutional networks

Images have properties that suggest the need for specialized architectures:

- They are high-dimensional inputs (e.g., 512×512 pixels = 262144 inputs).
- Nearby pixels are strongly correlated.
- The same patterns can appear at different locations in the image.
- Patterns are hierarchical: simple patterns combine to form more complex patterns.
- The interpretation of an image is stable under geometric transformations.

The MLP architecture would fail to exploit these properties, leading to poor performance and high computational cost.



—

Credits: Martin Handford, Where's Wally? series.

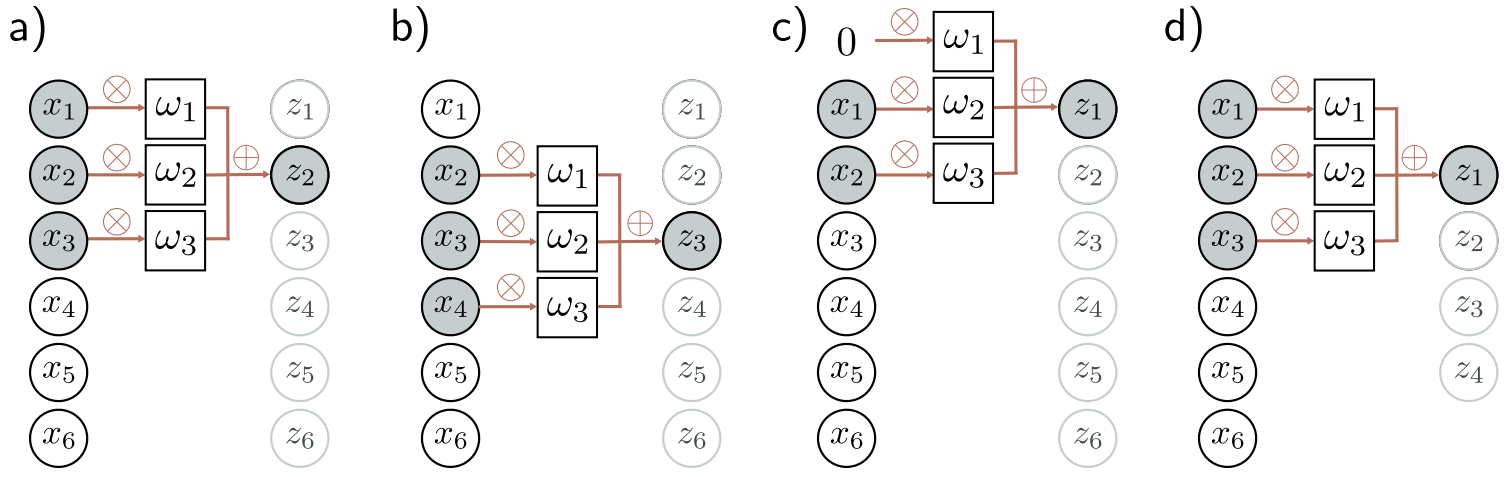
1d convolution

For the one-dimensional input $\mathbf{x} \in \mathbb{R}^W$ and the convolutional kernel $\boldsymbol{\omega} \in \mathbb{R}^w$, the discrete convolution $\mathbf{x} \circledast \boldsymbol{\omega}$ is a vector of size $W - w + 1$ such that

$$(\mathbf{x} \circledast \boldsymbol{\omega})[i] = \sum_{m=0}^{w-1} \mathbf{x}_{m+i} \boldsymbol{\omega}_m .$$

Optionally, the input $\mathbf{x}$ can be padded with p zeros on each side before applying the convolution, resulting in an output of size $W - w + 1 + 2p$. Setting $p = \frac{w-1}{2}$ (for odd w) preserves the input size.

Technically, $\circledast$ denotes the cross-correlation operator. However, most machine learning libraries call it convolution.



1d convolution with a kernel $\omega = (\omega_1, \omega_2, \omega_3)$ applied to the input signal $\mathbf{x}$. The output is obtained by sliding the kernel over the input and computing the inner product at each position.

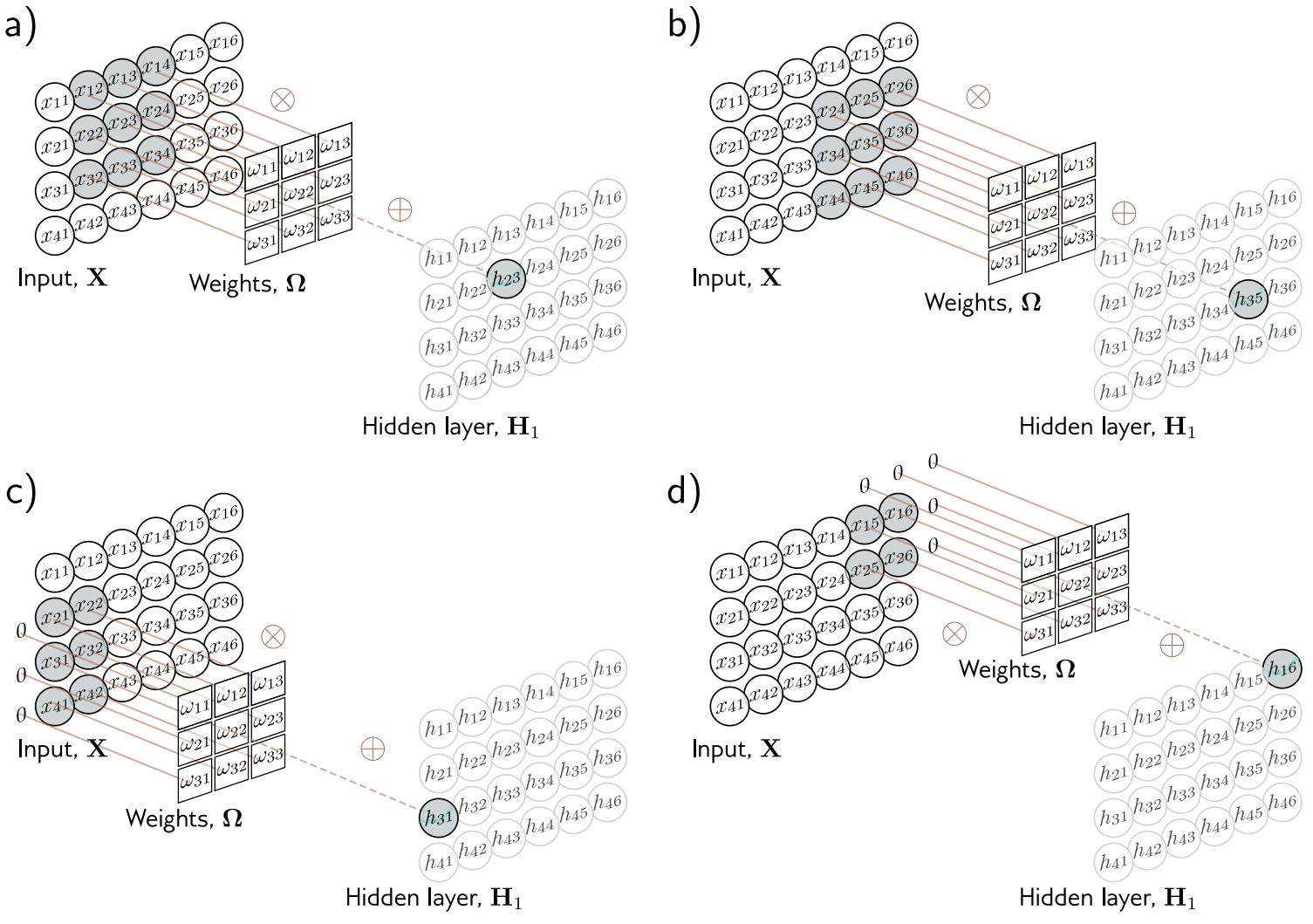
—

2d convolution

For the 2d input tensor $\mathbf{x} \in \mathbb{R}^{H \times W}$ and the 2d convolutional kernel $\boldsymbol{\omega} \in \mathbb{R}^{h \times w}$, the discrete convolution $\mathbf{x} \circledast \boldsymbol{\omega}$ is a matrix of size $(H - h + 1) \times (W - w + 1)$ such that

$$(\mathbf{x} \circledast \boldsymbol{\omega})[j, i] = \sum_{n=0}^{h-1} \sum_{m=0}^{w-1} \mathbf{x}_{n+j, m+i} \boldsymbol{\omega}_{n,m}$$

As for 1d convolution, padding can be applied to both spatial dimensions of the input to control the output size.



2d convolution with a kernel $\boldsymbol{\omega} \in \mathbb{R}^{3\times 3}$ applied to the input signal $\mathbf{x}$.

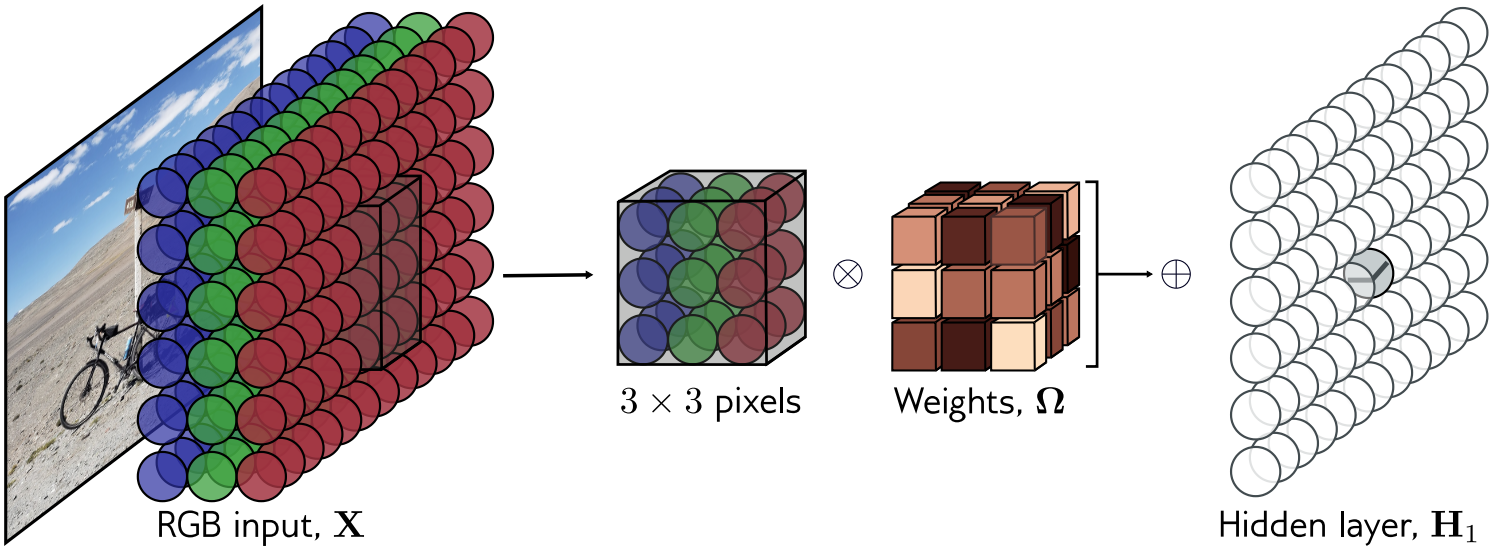
—

Channels

The 2d convolution can be extended to tensors with multiple channels.

For the 3d input tensor $\mathbf{x} \in \mathbb{R}^{C \times H \times W}$ and the 3d convolutional kernel $\omega \in \mathbb{R}^{C \times h \times w}$, the discrete 2d convolution $\mathbf{x} \circledast \omega$ is a 2d tensor of size $(H - h + 1) \times (W - w + 1)$ such that

$$(\mathbf{x} \circledast \omega)[j, i] = \sum_{c=0}^{C-1} \sum_{n=0}^{h-1} \sum_{m=0}^{w-1} \mathbf{x}_{c,n+j,m+i} \omega_{c,n,m}$$



2d convolution applied to a color image with three channels (R, G, B).

—

Convolutional layers

A convolutional layer is defined by a set of K kernels ω_k of size $C \times h \times w$. It applies the 2d convolution operation to the input tensor $\mathbf{x}$ of size $C \times H \times W$ to produce a set of K feature maps $\mathbf{o}_k$.

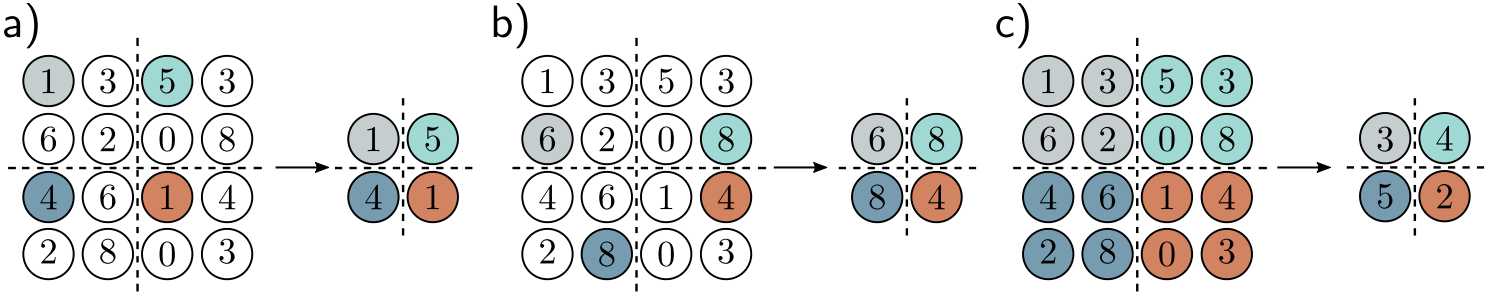
Each kernel ω_k acts as a feature detector that scans the input tensor and produces a feature map $\mathbf{o}_k$ specific to that feature.

Pooling layers

Pooling layers are used to progressively reduce the spatial size of the representation, hence capturing longer-range dependencies between features.

Considering a pooling area of size $h \times w$ and a 3D input tensor $\mathbf{x} \in \mathbb{R}^{C \times (rh) \times (sw)}$, max-pooling produces a tensor $\mathbf{o} \in \mathbb{R}^{C \times r \times s}$ such that

$$\mathbf{o}_{c,j,i} = \max_{n<h, m<w} \mathbf{x}_{c,rj+n,si+m}.$$



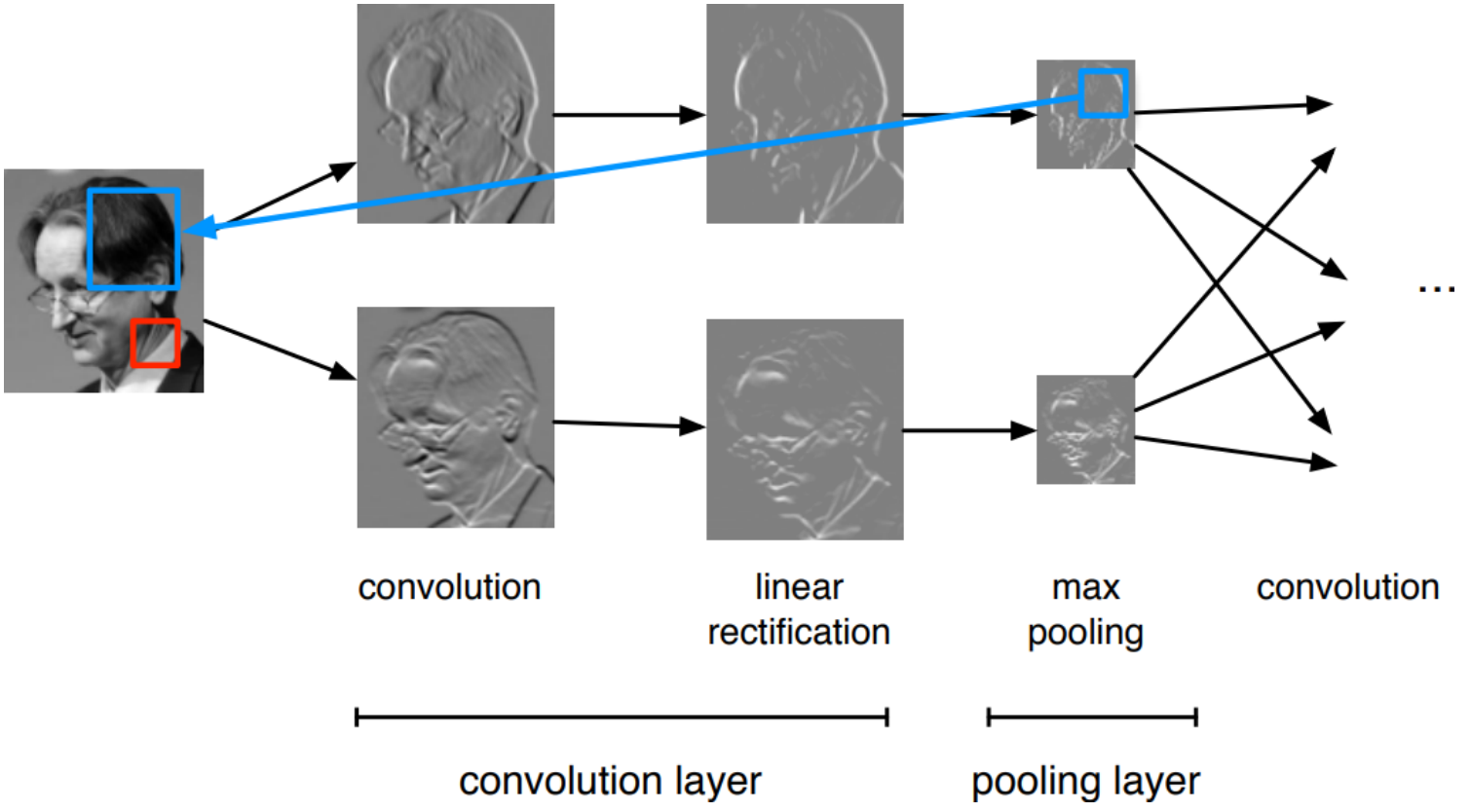
Subsampling, max-pooling and average-pooling with a pooling area of size $\mathbf{2 \times 2}$

.

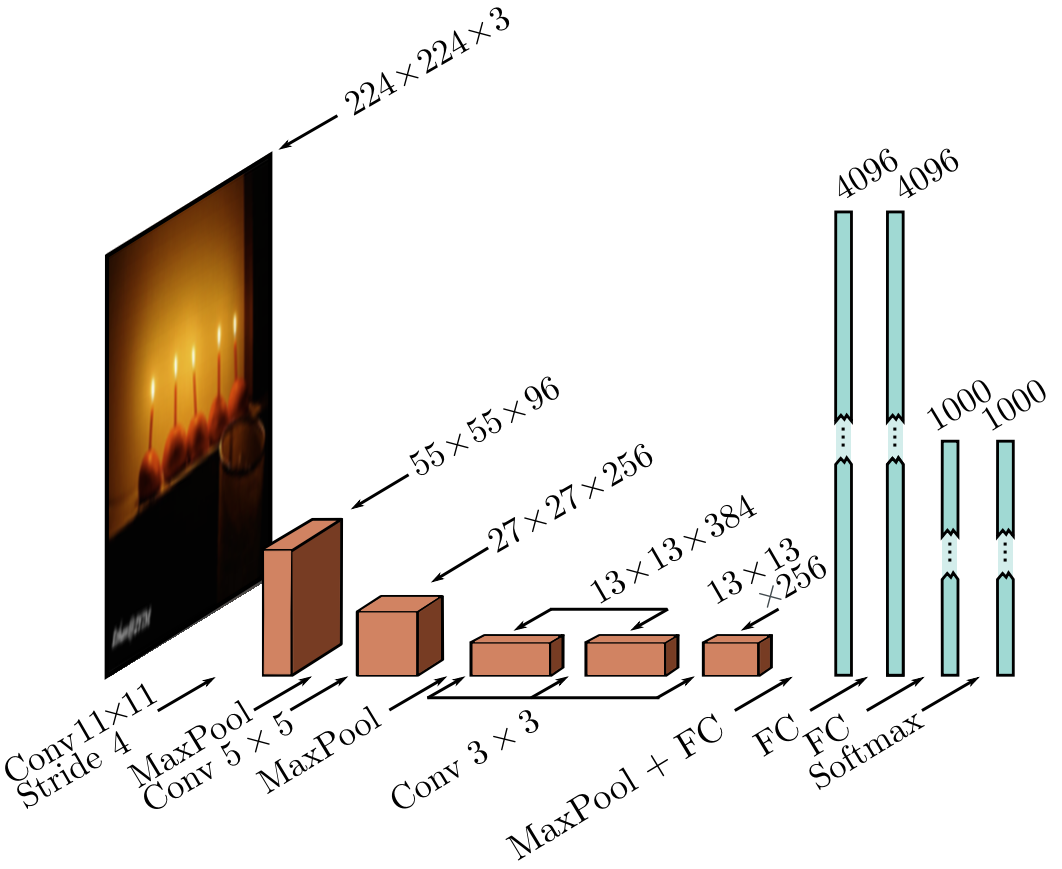
Convolutional neural networks (CNNs)

A convolutional neural network (CNN) is a deep neural network that uses convolutional layers, pooling layers, and fully connected layers to process multi-dimensional inputs such as images.

- Convolutional layers extract local features from the input while preserving spatial relationships.
- Pooling layers reduce the spatial dimensions of the feature maps, allowing the network to capture more global features.
- Fully connected layers at the end of the network combine the extracted features to make predictions.

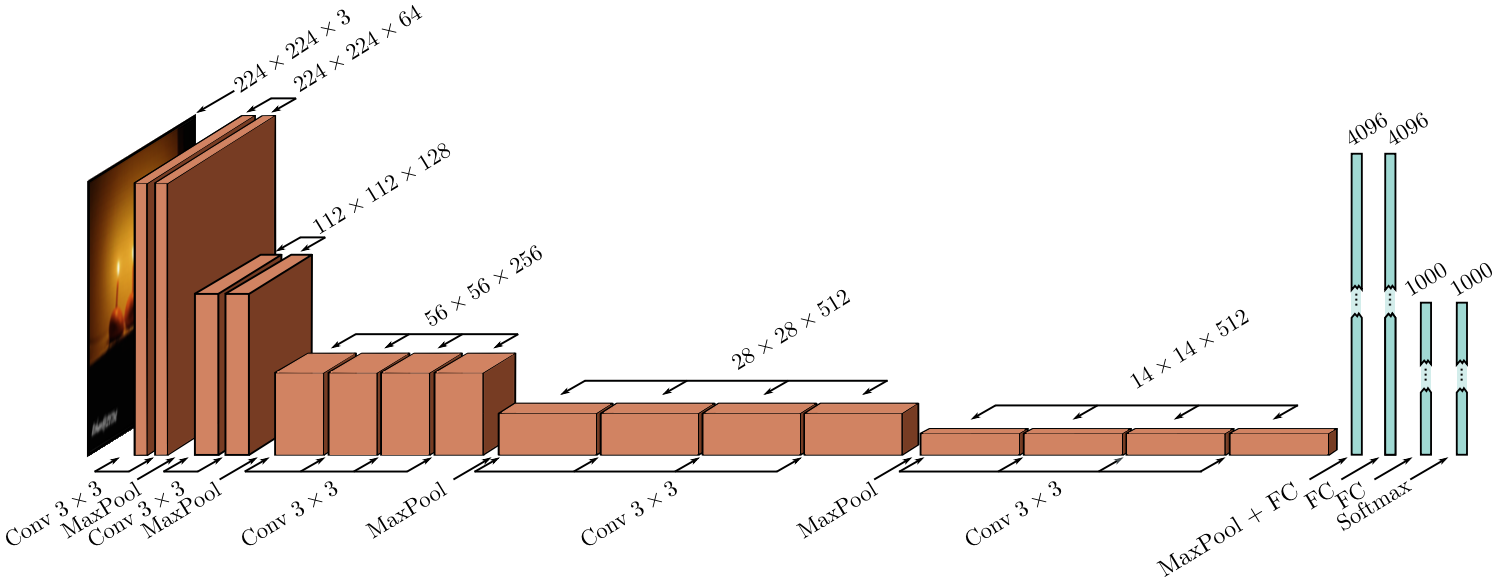


Interleaving convolutional and pooling layers enables the network to learn hierarchical features from local to global patterns.



The AlexNet architecture for image classification (Krizhevsky et al., 2012).

—



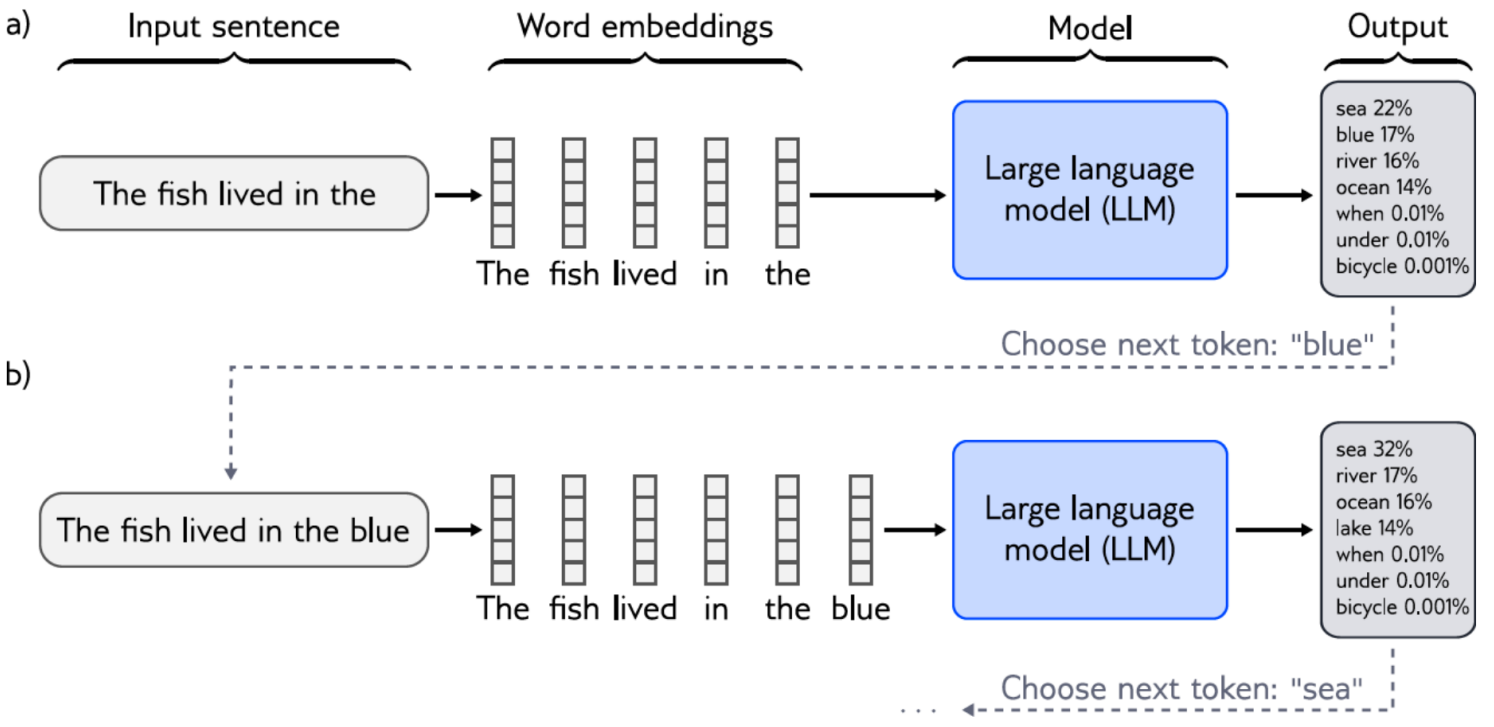
The VGG architecture for image classification (Simonyan and Zisserman, 2014).

—

(Step-by-step code example)

Transformers

Transformers are deep neural networks at the core of large language models.



—

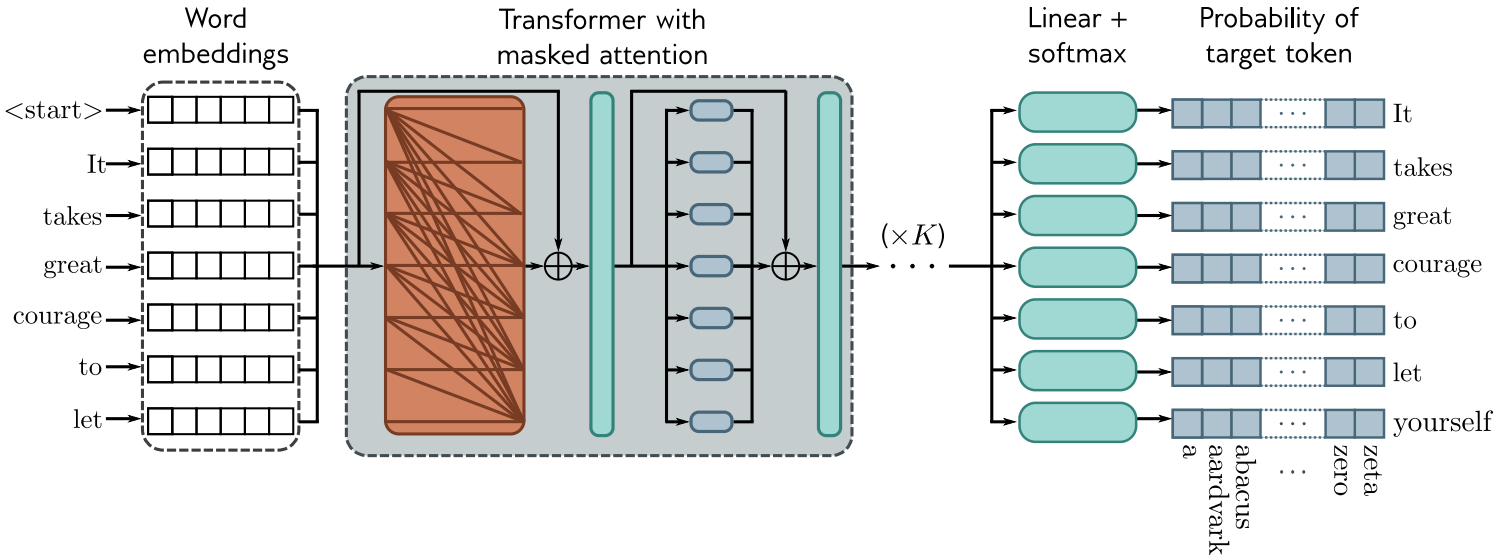
Credits: Simon J.D. Prince, 2023.

For language modeling, transformers define an **autoregressive model** that predicts the next word in a sequence given the previous words.

Formally,

$$p(w_{1:t}) = p(w_1) \prod_{t=2}^{T} p(w_t | w_{1:t-1}),$$

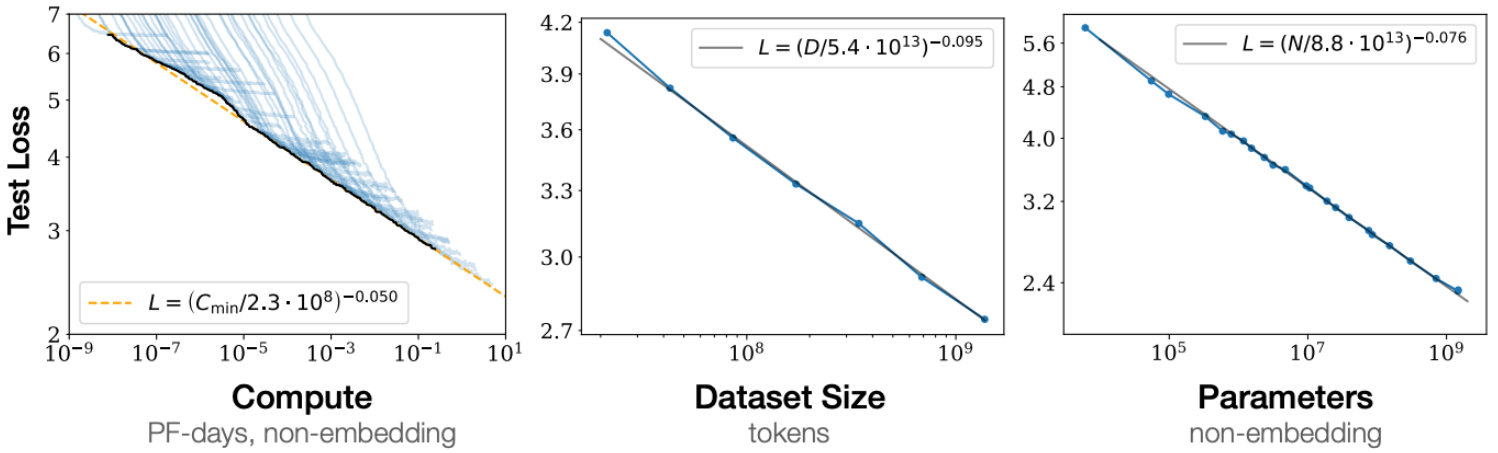
where w_t is the next word in the sequence and $w_{1:t-1}$ are the previous words.



The decoder-only transformer is a stack of K transformer blocks that process the input sequence in parallel using (masked) self-attention.

The output of the last block is used to predict the next word in the sequence, as in a regular classifier.

—



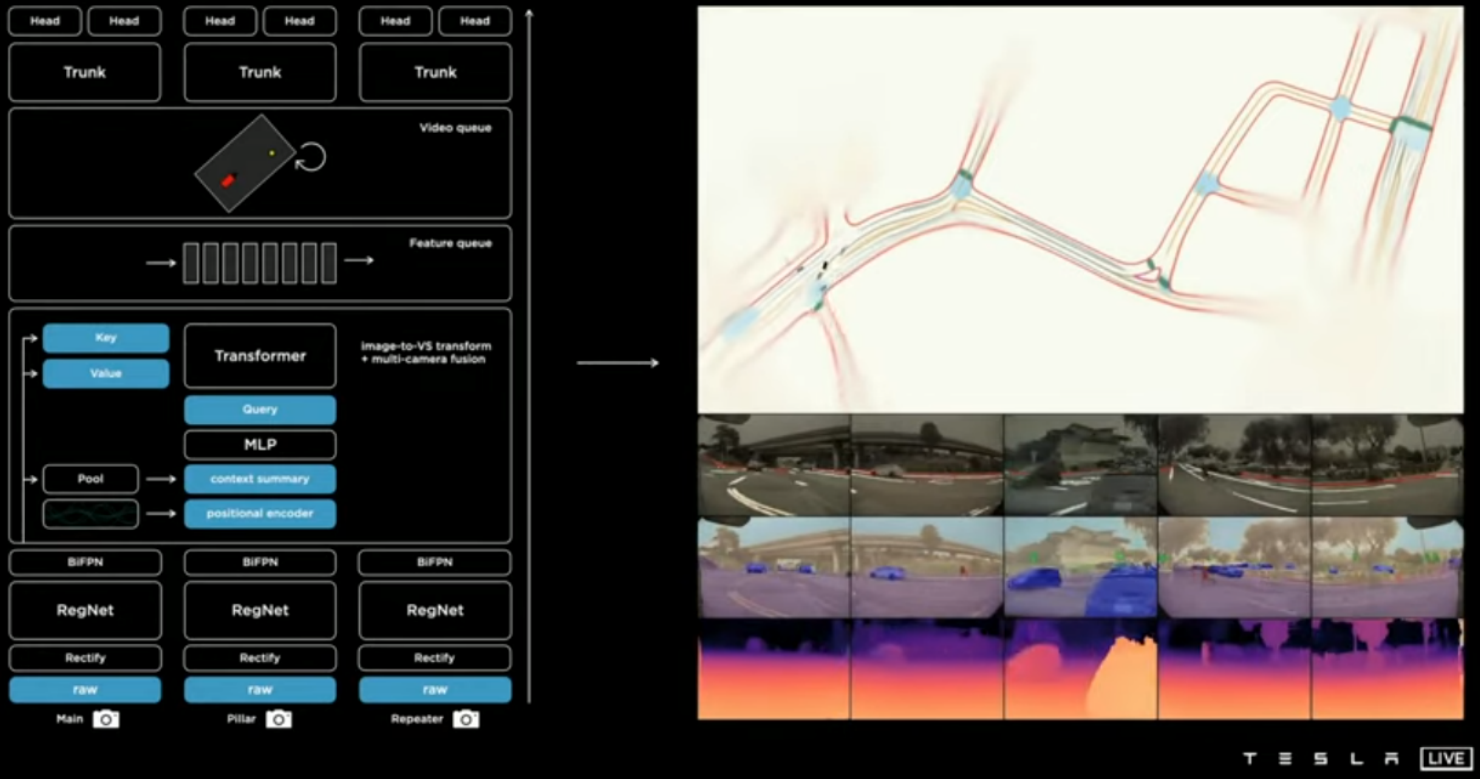
Scaling laws

- The more data, the better the model.
- The more parameters, the better the model.
- The more compute, the better the model.

(... at least until 2024-2025.)

AI beyond Pacman

How AI Helps Autonomous Vehicles See Outside the Box
(See also other episodes from NVIDIA DRIVE Labs)



Hydranet (Tesla, 2021)

How machine learning is advancing medicine (Google, 2018)

Summary

- Deep learning is a powerful tool for learning from data.
- Neural networks are composed of layers of neurons that are connected to each other.
- The weights of the connections are learned by minimizing a loss function.
- Convolutional networks are used for image processing.
- Transformers are used for language processing.

For the last forty years we have programmed computers; for the next forty years we will train them.

Chris Bishop, 2020.

The end.