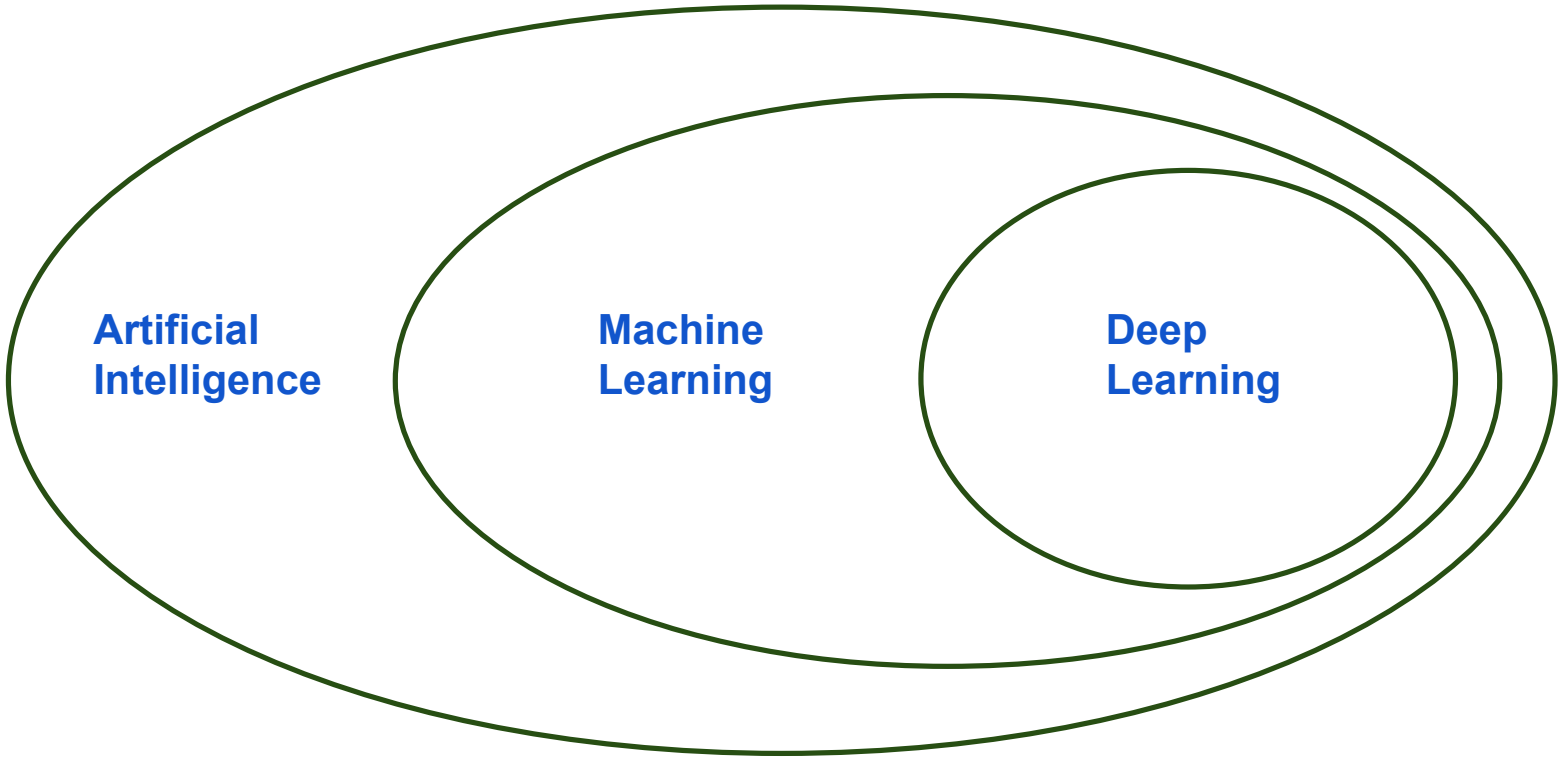
Introduction to Deep Learning

Essential Data Science Training GmbH

INTRODUCTION TO DEEP LEARNING

WHAT IS DEEP LEARNING



- Deep learning is a subfield of ML based on artificial neural networks.

DEEP LEARNING AND NEURAL NETWORKS

- Deep learning itself is not *new*:
  - Neural networks have been around since the 70s.
  - *Deep* neural networks, i.e., networks with multiple hidden layers, are not much younger.

- Why everybody is talking about deep learning now:
  1. Specialized, powerful hardware allows training of huge neural networks to push the state-of-the-art on difficult problems.
  2. Large amount of data is available.
  3. Special network architectures for image/text data.
  4. Better optimization and regularization strategies.

IMAGE CLASSIFICATION WITH NEURAL NETWORKS

"Machine learning algorithms, inspired by the brain, based on learning multiple levels of representation/abstraction."

Y. Bengio

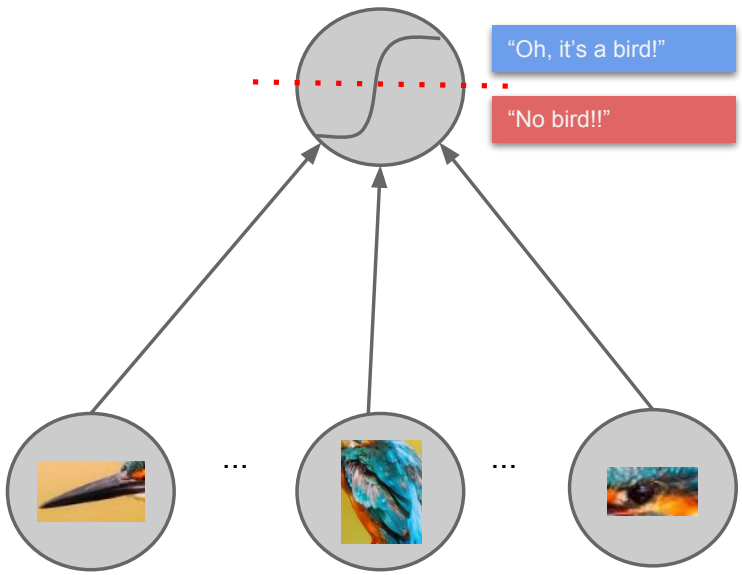


IMAGE CLASSIFICATION WITH NEURAL NETWORKS

"Machine learning algorithms, inspired by the brain, based on learning multiple levels of representation/abstraction."

Y. Bengio

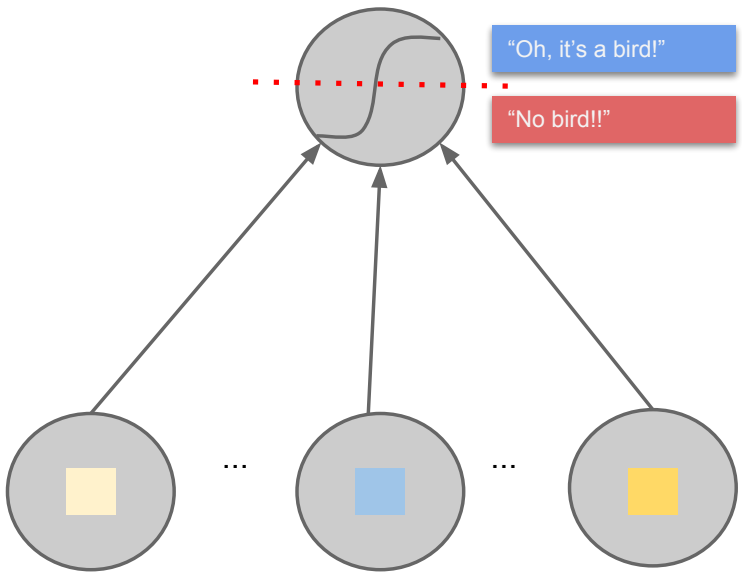


IMAGE CLASSIFICATION WITH NEURAL NETWORKS

"Machine learning algorithms, inspired by the brain, based on learning multiple levels of representation/abstraction."

Y. Bengio

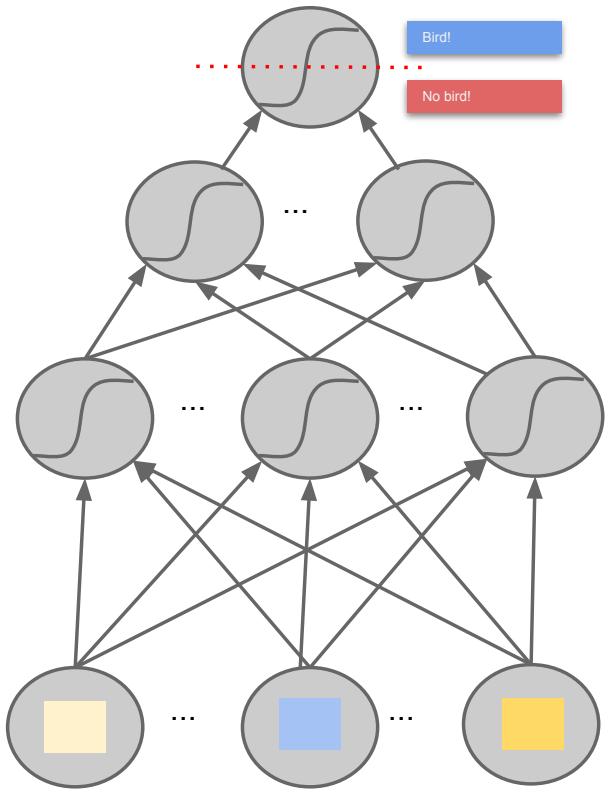
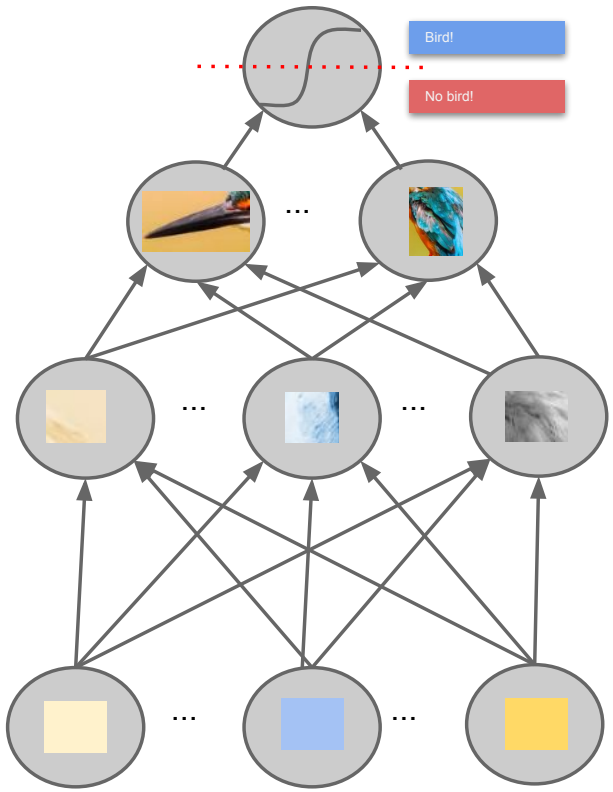


IMAGE CLASSIFICATION WITH NEURAL NETWORKS

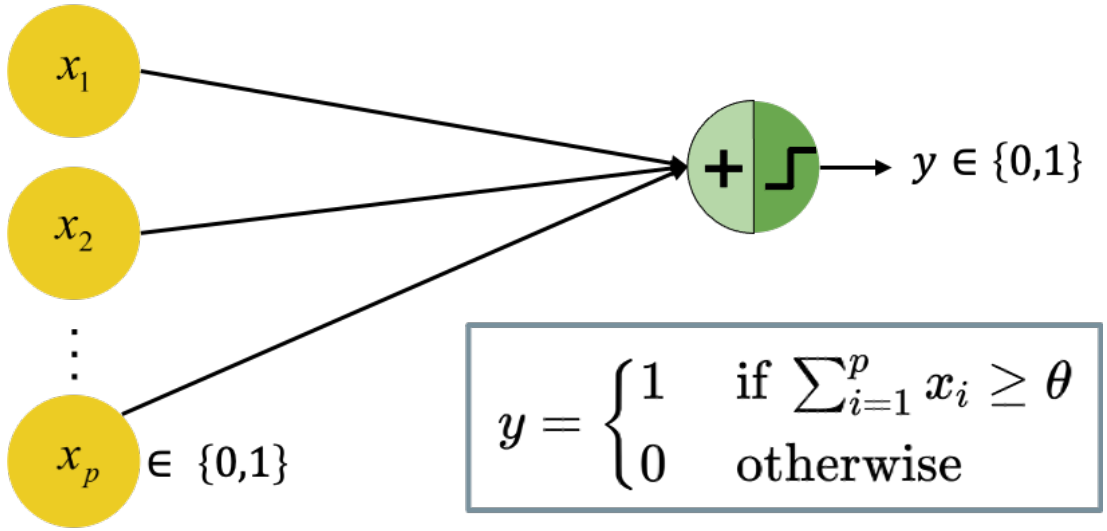
"Machine learning algorithms, inspired by the brain, based on learning multiple levels of representation/abstraction."

Y. Bengio



A BRIEF HISTORY OF NEURAL NETWORKS

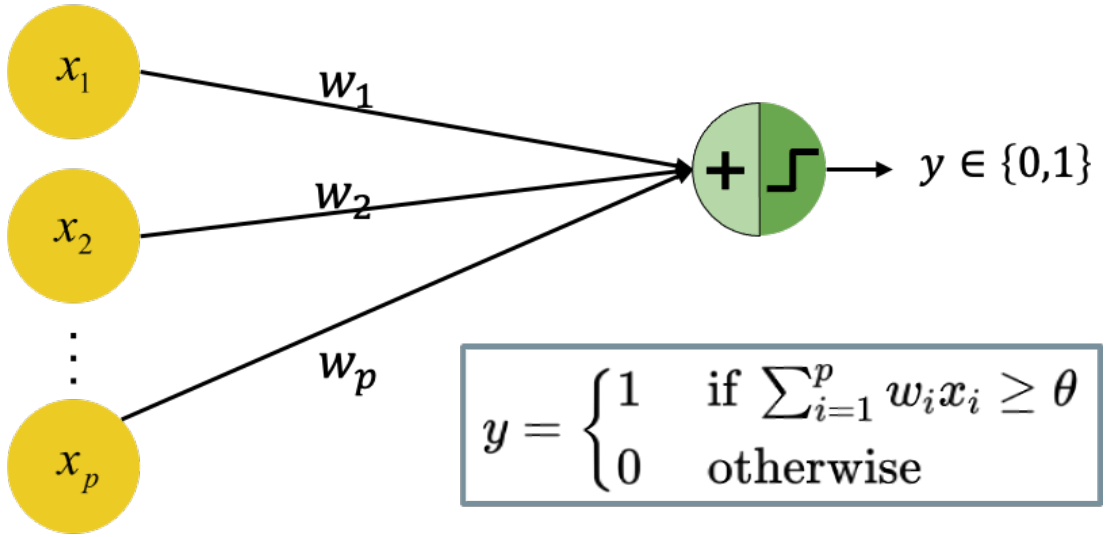
- **1943:** The first artificial neuron, the "Threshold Logic Unit (TLU)", was proposed by Warren McCulloch & Walter Pitts.



  - The model is limited to binary inputs.
  - It fires/outputs $+1$ if the input exceeds a certain threshold θ.
  - The weight are not adjustable, so learning could only be achieved by changing the threshold θ.

A BRIEF HISTORY OF NEURAL NETWORKS

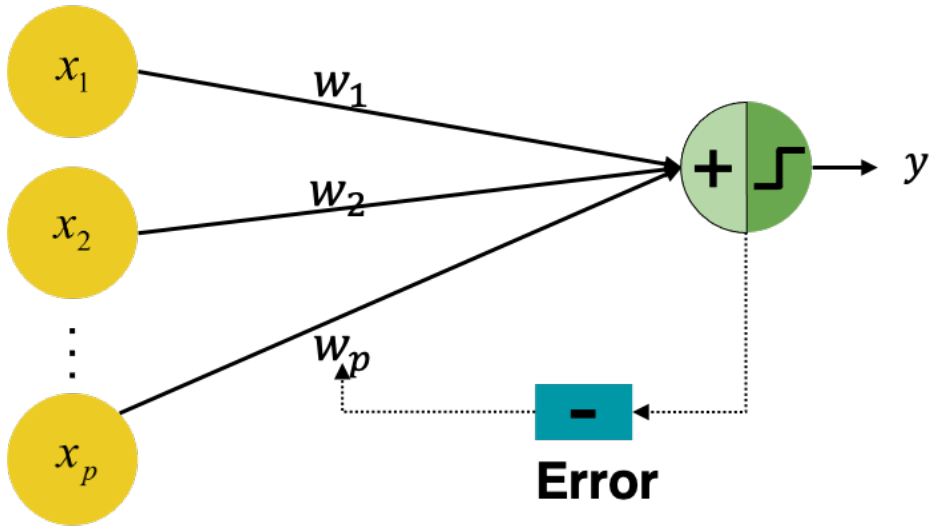
- **1957:** The perceptron was invented by Frank Rosenblatt.



  - The inputs are not restricted to be binary.
  - The weights are adjustable and can be learned by learning algorithms.
  - As for the TLU, the threshold is adjustable and decision boundaries are linear.

A BRIEF HISTORY OF NEURAL NETWORKS

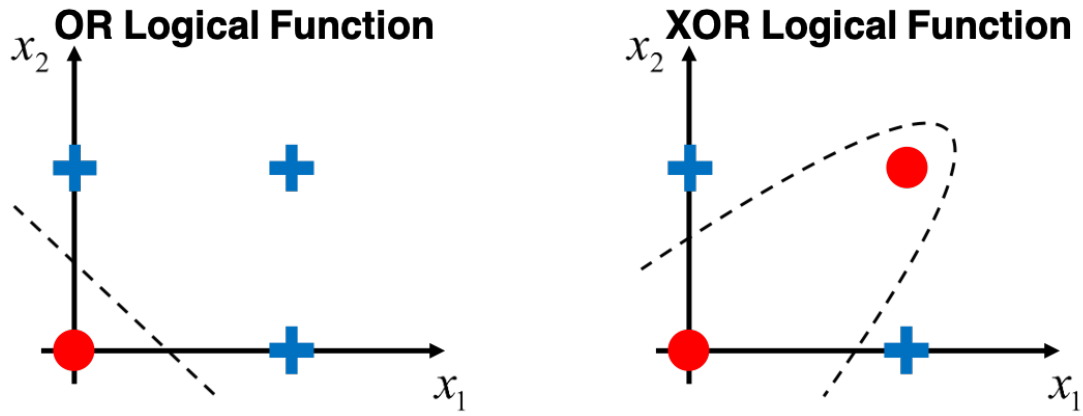
- **1960:** Adaptive Linear Neuron (ADALINE) was invented by Bernard Widrow & Ted Hoff; weights are now adjustable according to the weighted sum of the inputs.



- **1965:** Group method of data handling (also known as polynomial neural networks) by Alexey Ivakhnenko. The first learning algorithm for supervised deep feedforward multilayer perceptrons.

A BRIEF HISTORY OF NEURAL NETWORKS

- **1969:** The first “AI Winter” kicked in.
  - Marvin Minsky & Seymour Papert proved that a perceptron cannot solve the XOR-Problem (linear separability).
  - Less funding $\Rightarrow$ Standstill in AI/DL research.



- **1985:** Multilayer perceptron with backpropagation by David Rumelhart, Geoffrey Hinton, and Ronald Williams.
  - Efficiently compute derivatives of composite functions.
  - Backpropagation was developed already in 1970 by Linnainmaa.

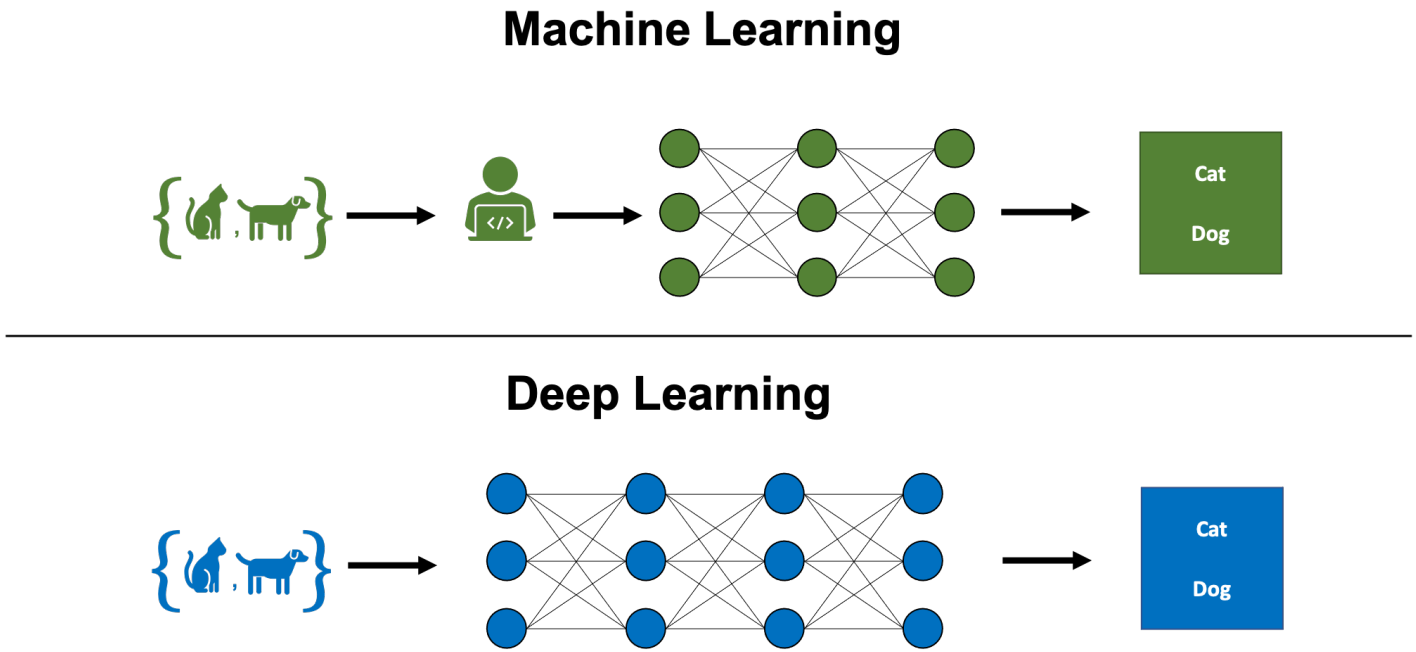
A BRIEF HISTORY OF NEURAL NETWORKS

- **1985:** The second “AI Winter” kicked in.
  - Overly optimistic expectations concerning potential of AI/DL.
  - The phrase “AI” even reached a pseudoscience status.
  - Kernel machines and graphical models both achieved good results on many important tasks.
  - Some fundamental mathematical difficulties in modeling long sequences were identified.

Figure: Symbolic image of an “AI Winter” (Fagella, 2019)

A BRIEF HISTORY OF NEURAL NETWORKS

- **2006:** Age of deep neural networks began.
  - Geoffrey Hinton showed that a deep belief network could be efficiently trained using *greedy layer-wise pretraining*.
  - This wave of research popularized the use of the term deep learning to emphasize that researchers were now able to train deeper neural networks than had been possible before.
  - At this time, deep neural networks outperformed competing AI systems based on other ML technologies as well as hand-designed functionality.



A BRIEF HISTORY OF NEURAL NETWORKS

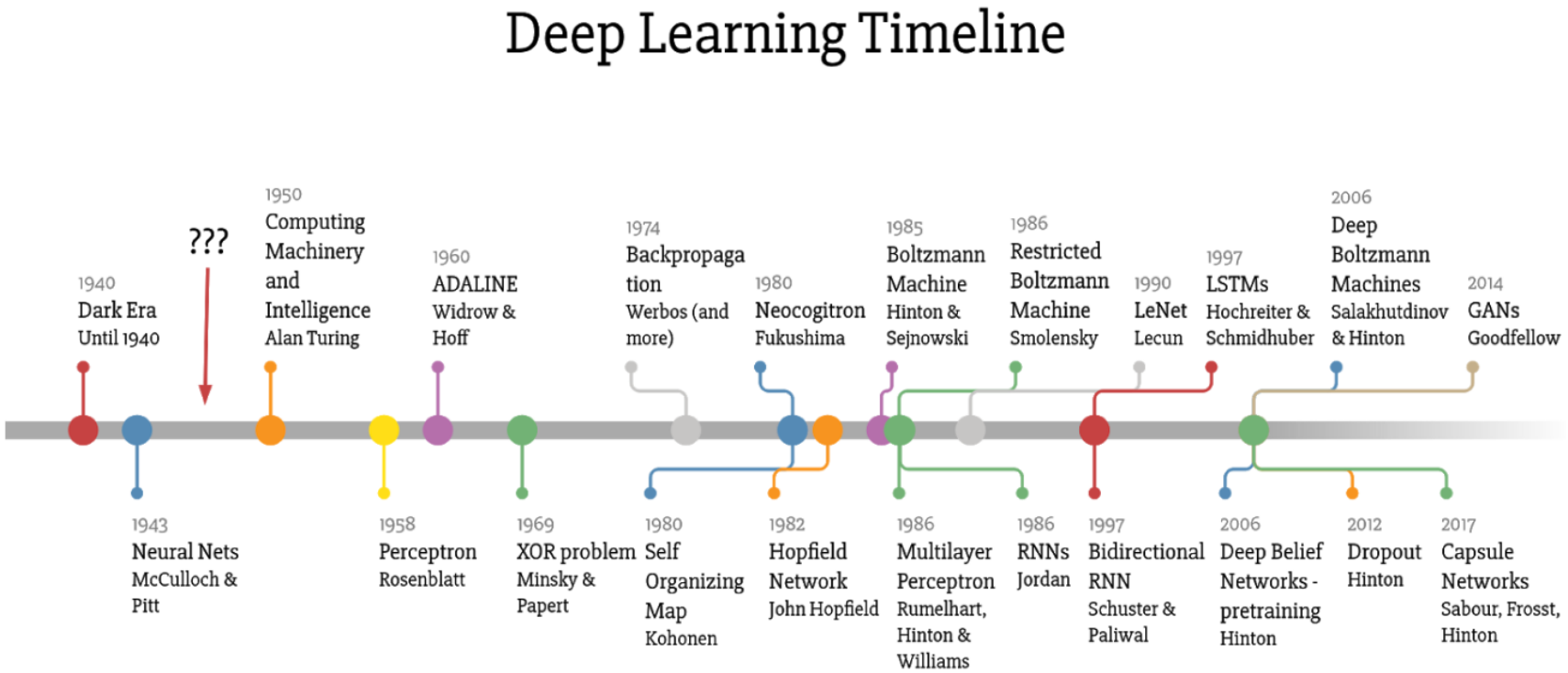
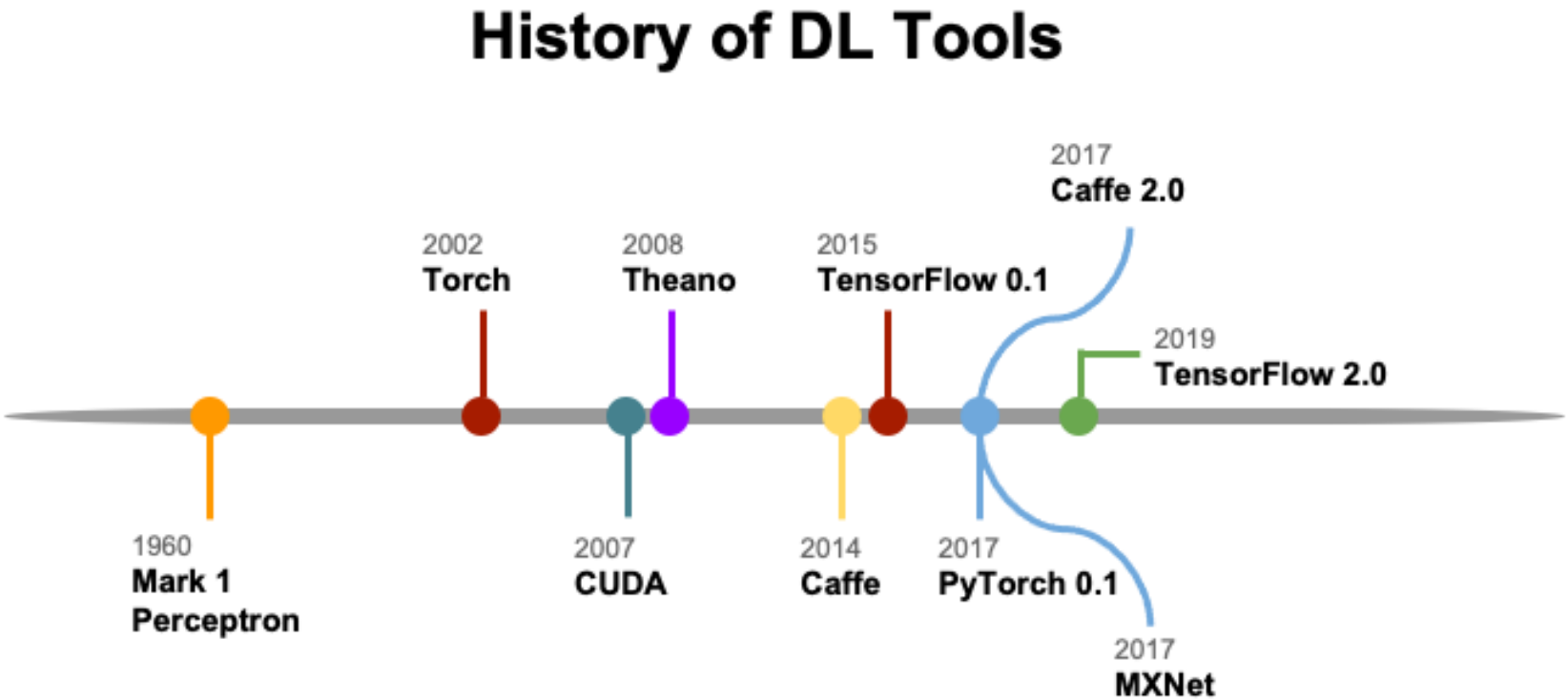


Figure: (Vazquez, 2018)

A BRIEF HISTORY OF NEURAL NETWORKS



A BRIEF HISTORY OF NEURAL NETWORKS

Figure: IBM Supercomputer Watson (IBM, 2011)

- Watson is a question-answering system capable of answering questions posed in natural language, developed in IBM's DeepQA project.
- In 2011, Watson competed on *Jeopardy!* against champions Brad Rutter and Ken Jennings, winning the first place prize of $1 million.

A BRIEF HISTORY OF NEURAL NETWORKS

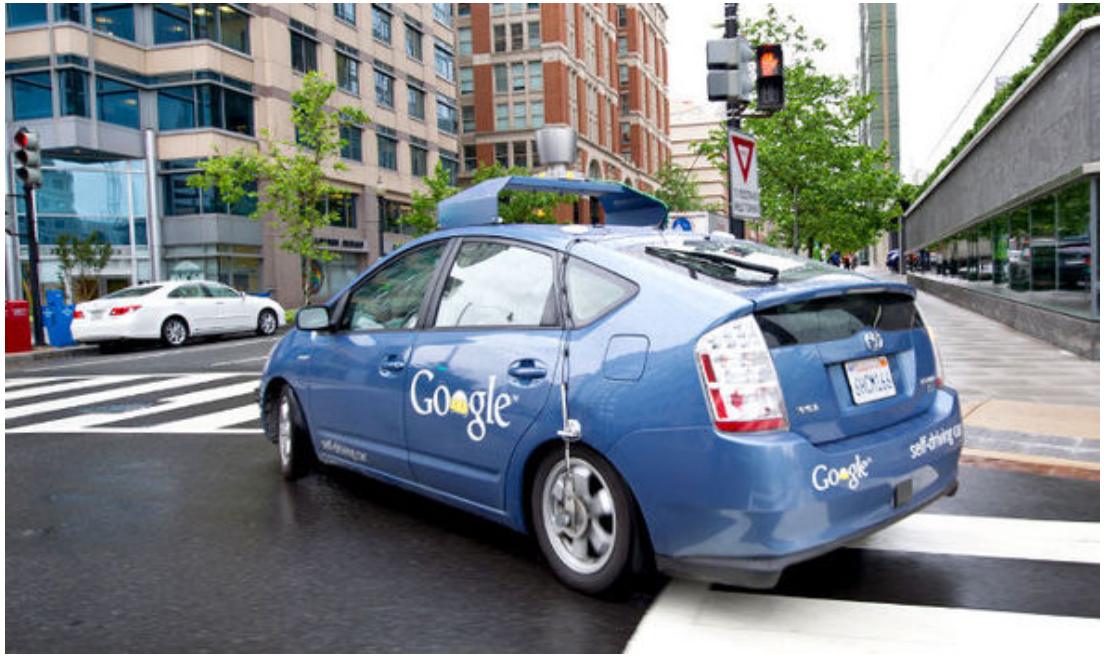


Figure: Google's self driving car (McCabe, 2016)

- Google's development of self-driving technology began on January 17, 2009, at the company's secretive X lab.
- By January 2020, 20 million miles of self-driving on public roads had been completed by Waymo.

A BRIEF HISTORY OF NEURAL NETWORKS

Figure: Protein structure by AlphaFold (DeepMind, 2022)

- **AlphaFold** is a deep learning system, developed by Google DeepMind, for determining a protein's 3D shape from its amino-acid sequence.
- In 2018 and 2020, AlphaFold placed first in the overall rankings of the Critical Assessment of Techniques for Protein Structure Prediction (CASP).

A BRIEF HISTORY OF NEURAL NETWORKS

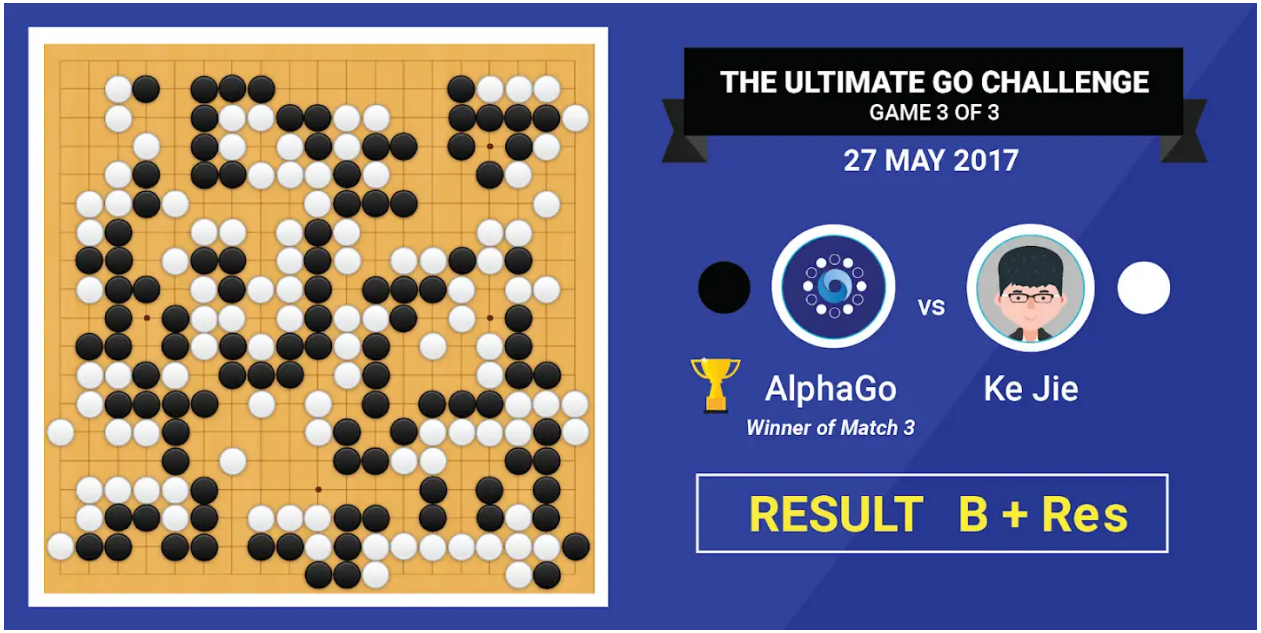


Figure: AlphaGo (DeepMind, 2022)

- **AlphaGo**, originally developed by DeepMind, is a deep learning system that plays the board game Go. In 2017, the Master version of AlphaGo beat Ke Jie, the number one ranked player in the world at the time.
- While there are several extensions to AlphaGo (e.g., Master AlphaGo, AlphaGo Zero, AlphaZero, and MuZero), the main idea is the same: search for optimal moves based on knowledge acquired by machine learning.

A BRIEF HISTORY OF NEURAL NETWORKS

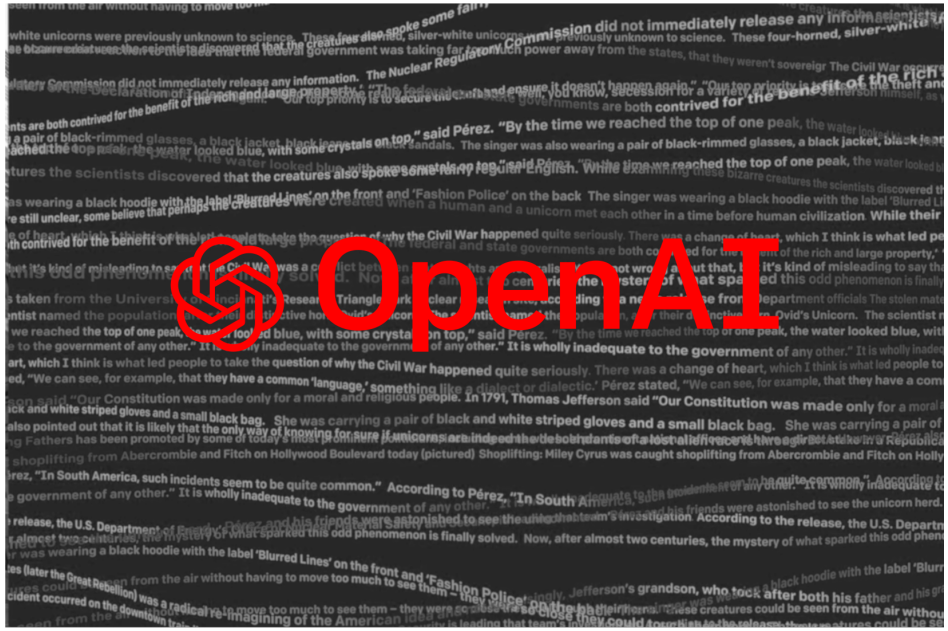
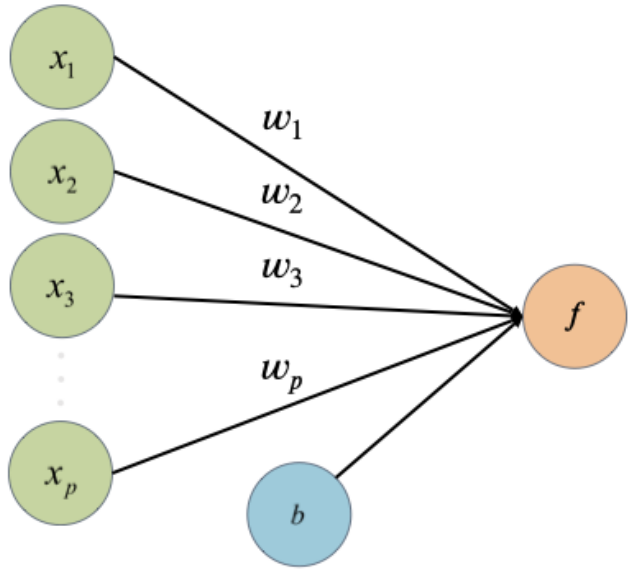


Figure: Logo OpenAI (OpenAI, 2023)

- **Generative Pre-trained Transformer 3 (GPT-3)** is the third generatation of the GPT model, introduced by OpenAI in May 2020, to produce human-like text.
- There are 175 billion parameters to be learned by the algorithm, but the quality of the generated text is so high that it is hardly possible to distinguish it from a human-written text.

INTRODUCTION TO DEEP LEARNING

A SINGLE NEURON



Perceptron with **input features** $x_1, x_2, \ldots, x_p$, **weights** $w_1, w_2, \ldots, w_p$, **bias term** b, and **activation function** τ.

- The perceptron is a single artificial neuron and the basic computational unit of neural networks.
- It is a weighted sum of input values, transformed by τ:

$$f(x) = \tau(w_1 x_1 + \ldots + w_p x_p + b) = \tau(\mathbf{w}^T \mathbf{x} + b)$$

A SINGLE NEURON

Activation function τ: a single neuron represents different functions depending on the choice of activation function.

- The identity function gives us the simple **linear regression**:

$$f(x) = \tau(\mathbf{w}^T\mathbf{x}) = \mathbf{w}^T\mathbf{x}$$

- The logistic function gives us the **logistic regression**:

$$f(x) = \tau(\mathbf{w}^T\mathbf{x}) = \frac{1}{1 + \exp(-\mathbf{w}^T\mathbf{x})}$$

A SINGLE NEURON

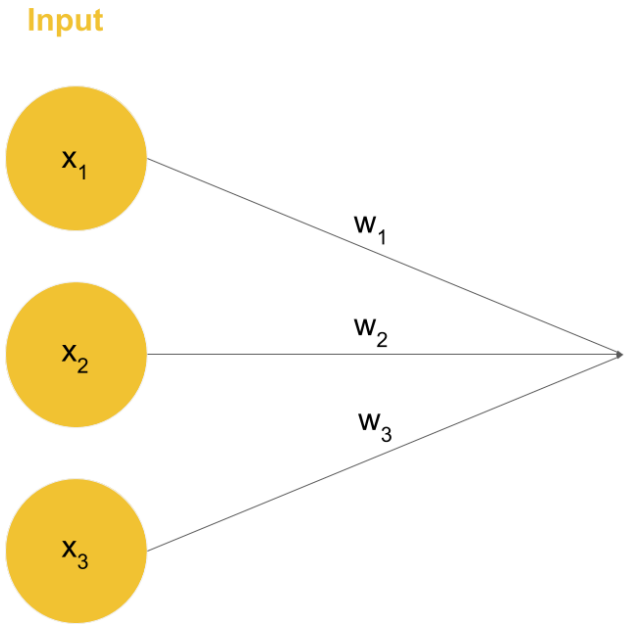
We consider a perceptron with 3-dimensional input, i.e.
$f(\mathbf{x}) = \tau(w_1 x_1 + w_2 x_2 + w_3 x_3 + b)$.

- Input features **x** are represented by nodes in the "input layer".

- In general, a p-dimensional input vector **x** will be represented by p nodes in the input layer.

A SINGLE NEURON

- Weights **w** are connected to edges from the input layer.



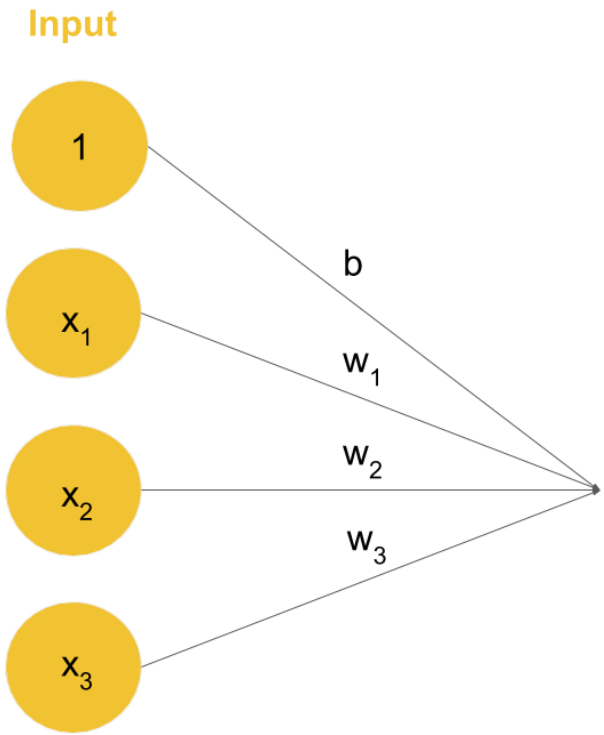
- The bias term *b* is implicit here. It is often not visualized as a separate node.

A SINGLE NEURON

For an explicit graphical representation, we do a simple trick:

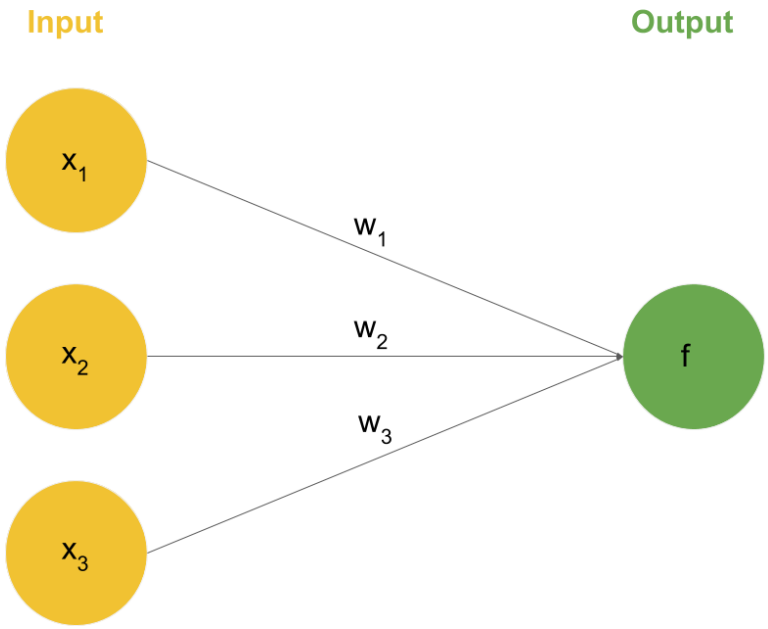
- Add a constant feature to the inputs $\tilde{\mathbf{x}} = (1, x_1, ..., x_p)^T$
- and absorb the bias into the weight vector $\tilde{\boldsymbol{w}} = (b, w_1, ..., w_p)$.

The graphical representation is then:



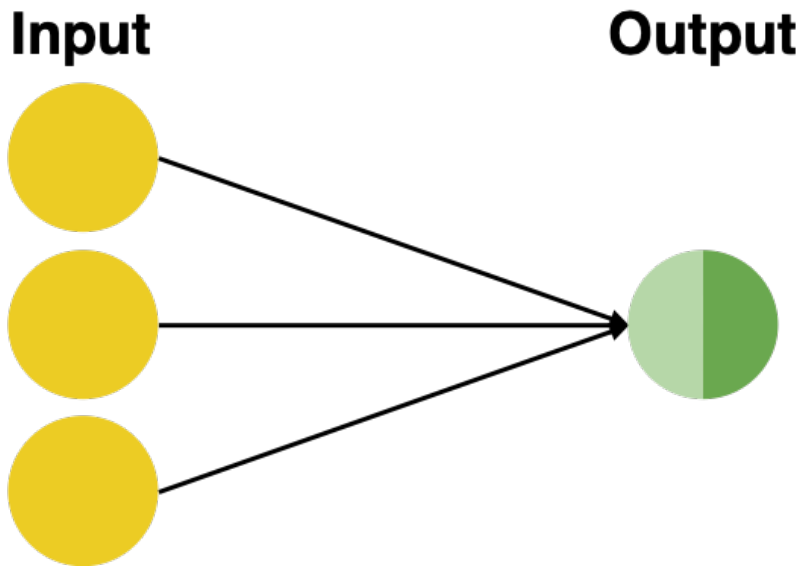
A SINGLE NEURON

- The computation $\tau(w_1 x_1 + w_2 x_2 + w_3 x_3 + b)$ is represented by the neuron in the "output layer".



A SINGLE NEURON

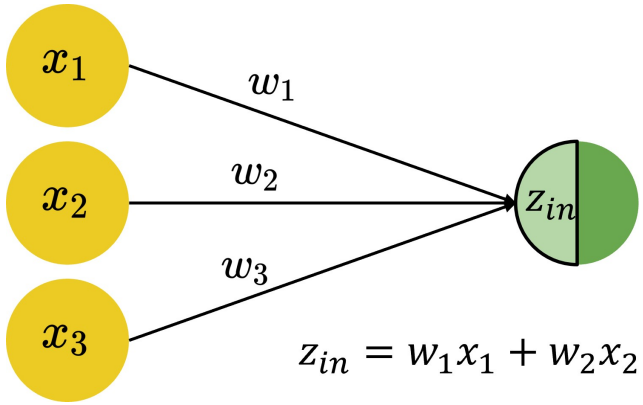
- You can picture the input vector being "fed" to neurons on the left followed by a sequence of computations performed from left to right. This is called a **forward pass**.



A SINGLE NEURON

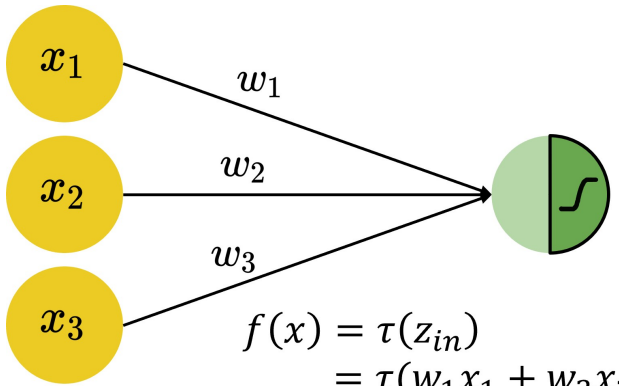
A neuron performs a 2-step computation:

1. **Affine Transformation:** weighted sum of inputs plus bias.



$$z_{in} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$$

2. **Non-linear Activation:** a non-linear transformation applied to the weighted sum.



$$f(x) = \tau(z_{in})$$
$$= \tau(w_1 x_1 + w_2 x_2 + w_3 x_3 + b)$$

A SINGLE NEURON: HYPOTHESIS SPACE

- The hypothesis space that is formed by single neuron is

$$\mathcal{H} = \left\{ f : R^p \to R \;\middle|\; f(\mathbf{x}) = \tau\left(\sum_{j=1}^{p} w_j x_j + b\right), \mathbf{w} \in R^p, b \in R \right\}.$$

- If τ is the logistic sigmoid or identity function, $\mathcal{H}$ corresponds to the hpothesis space of logistic or linear regression, respectively.

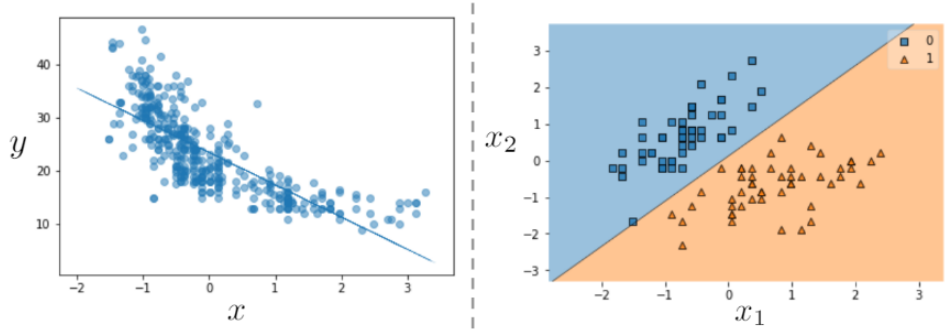


Figure: *Left*: A regression line learned by a single neuron. *Right*: A decision-boundary learned by a single neuron in a binary classification task.

A SINGLE NEURON: OPTIMIZATION

- To optimize this model, we minimize the empirical risk

$$\mathcal{R}_{\text{emp}} = \frac{1}{n} \sum_{i=1}^{n} L\left(y^{(i)}, f\left(\mathbf{x}^{(i)}\right)\right),$$

  where $L(y, f(\mathbf{x}))$ is a loss function. It compares the network's predictions $f(\mathbf{x})$ to the ground truth y.
- For regression, we typically use the L2 loss (rarely L1):

$$L(y, f(\mathbf{x})) = \frac{1}{2}(y - f(\mathbf{x}))^2$$

- For binary classification, we typically apply the cross entropy loss (also known as Bernoulli loss):

$$L(y, f(\mathbf{x})) = -(y \log f(\mathbf{x}) + (1 - y) \log(1 - f(\mathbf{x})))$$

A SINGLE NEURON: OPTIMIZATION

- For a single neuron and both choices of τ the loss function is convex.
- The global optimum can be found with an iterative algorithm like gradient descent.
- A single neuron with logistic sigmoid function trained with the Bernoulli loss yields the same result as logistic regression when trained until convergence.
- Note: In the case of regression and the L2-loss, the solution can also be found analytically using the “normal equations”. However, in other cases a closed-form solution is usually not available.

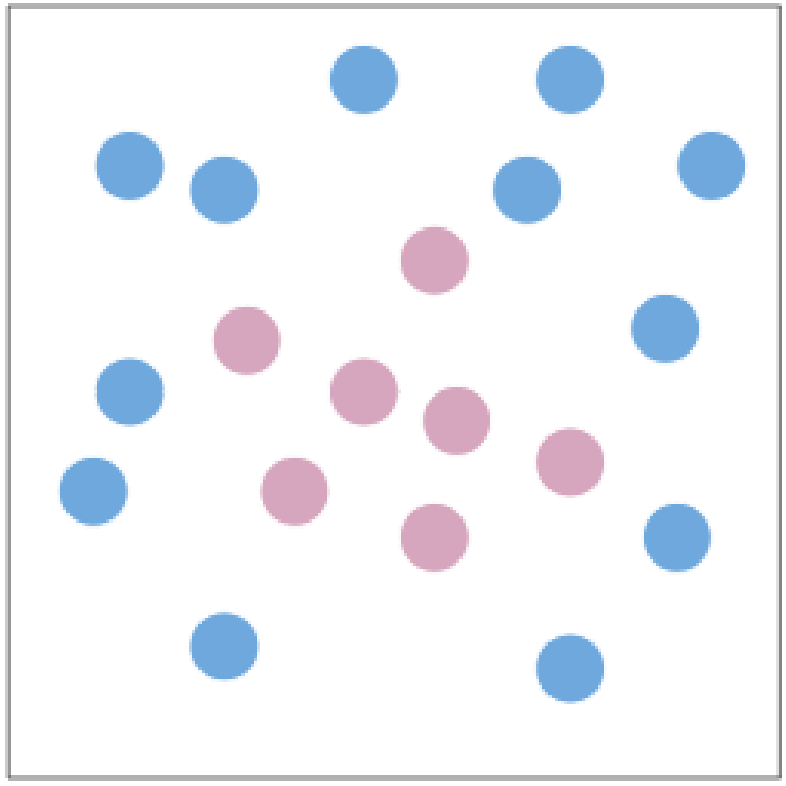
INTRODUCTION TO DEEP LEARNING

MOTIVATION

- The graphical way of representing simple functions/models, like logistic regression. Why is that useful?
- Because individual neurons can be used as building blocks of more complicated functions.
- Networks of neurons can represent extremely complex hypothesis spaces.
- Most importantly, it allows us to define the “right” kinds of hypothesis spaces to learn functions that are common in our universe in a data-efficient way (see Lin, Tegmark et al. 2016).

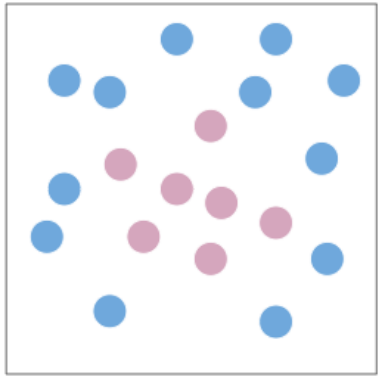
MOTIVATION

Can a single neuron perform binary classification of these points?



MOTIVATION

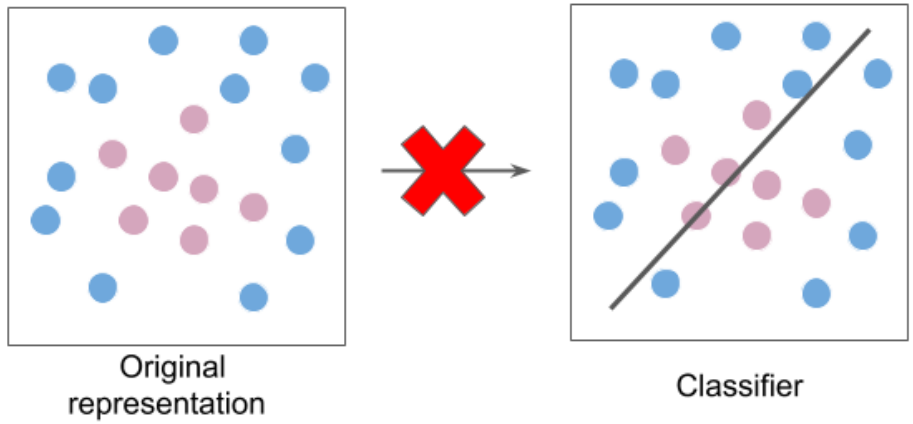
- As a single neuron is restricted to learning only linear decision boundaries, its performance on the following task is quite poor:



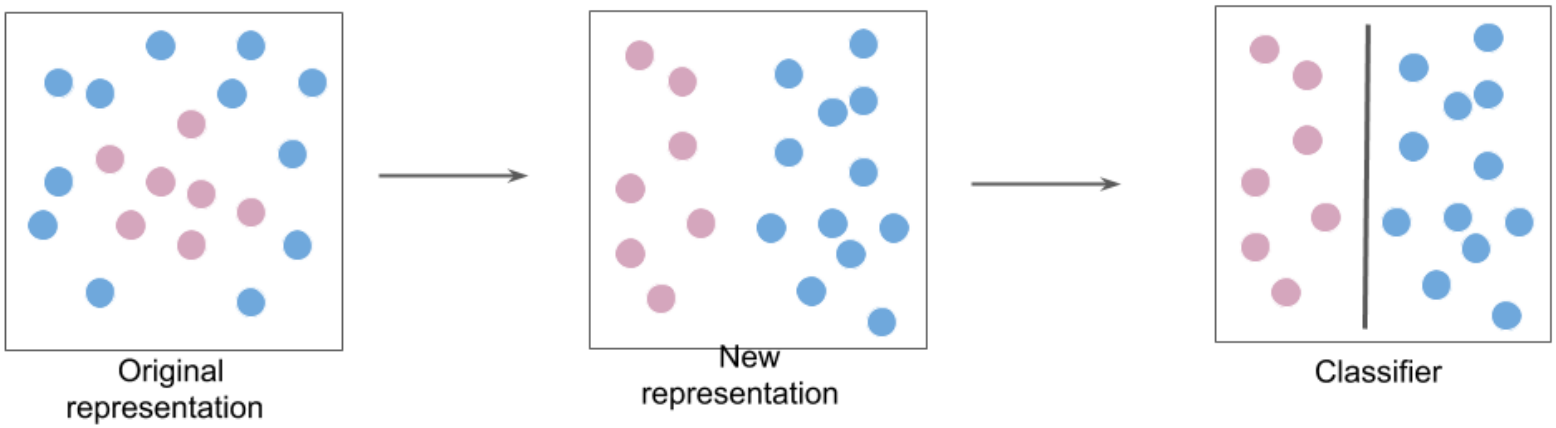
- However, the neuron can easily separate the classes if the original features are transformed (e.g., from Cartesian to polar coordinates):

MOTIVATION

- Instead of classifying the data in the original representation,

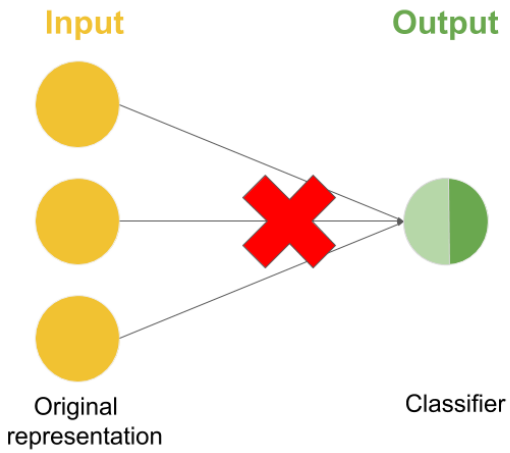


- we classify it in a new feature space.

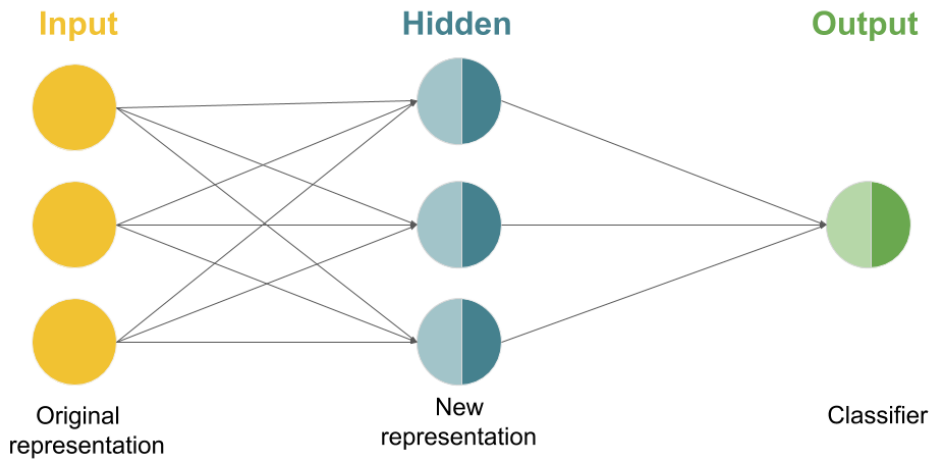


MOTIVATION

- Analogously, instead of a single neuron,



- we use more complex networks.



REPRESENTATION LEARNING

- It is *very* critical to feed a classifier the “right” features in order for it to perform well.

- Before deep learning took off, features for tasks like machine vision and speech recognition were “hand-designed” by domain experts. This step of the machine learning pipeline is called **feature engineering**.

- DL automates feature engineering. This is called **representation learning**.

SINGLE HIDDEN LAYER NETWORKS

Single neurons perform a 2-step computation:

1. **Affine Transformation:** a weighted sum of inputs plus bias.
2. **Activation:** a non-linear transformation on the weighted sum.

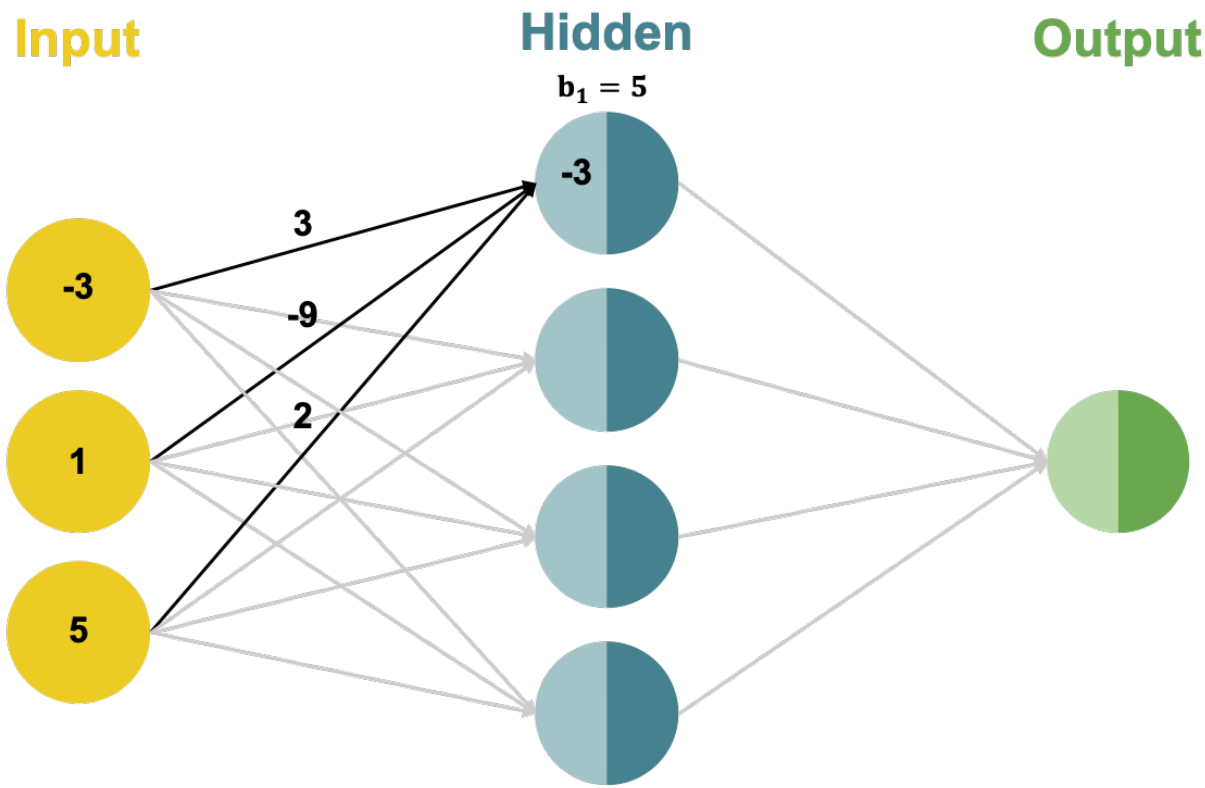
Single hidden layer networks consist of two layers (without input layer):

1. **Hidden Layer:** having a set of neurons.
2. **Output Layer:** having one or more output neurons.

- Multiple inputs are simultaneously fed to the network.
- Each neuron in the hidden layer performs a 2-step computation.
- The final output of the network is then calculated by another 2-step computation performed by the neuron in the output layer.

SINGLE HIDDEN LAYER NETWORKS: EXAMPLE

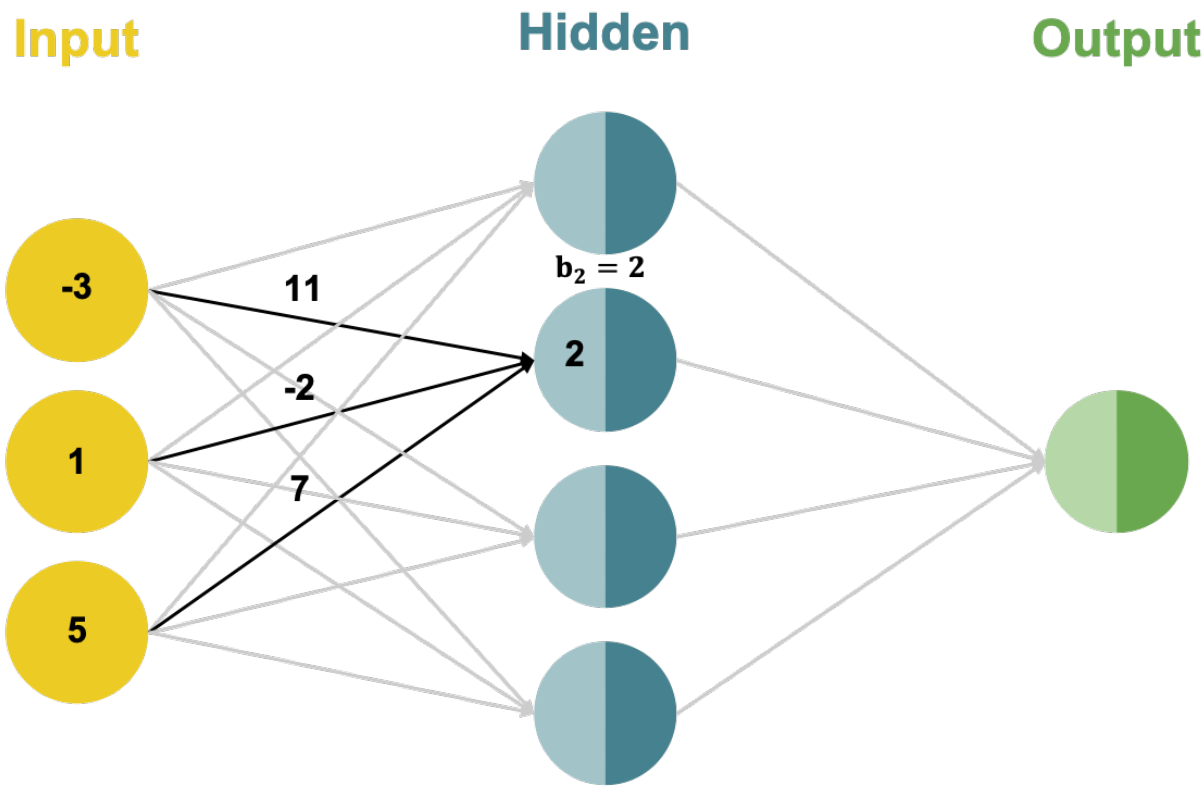
Each neuron in the hidden layer performs an **affine transformation** on the inputs:



$$z_{1,\text{in}} = w_{11}x_1 + w_{21}x_2 + w_{31}x_3 + b_1$$
$$z_{1,\text{in}} = 3 * (-3) + (-9) * 1 + 2 * 5 + 5 = -3$$

SINGLE HIDDEN LAYER NETWORKS: EXAMPLE

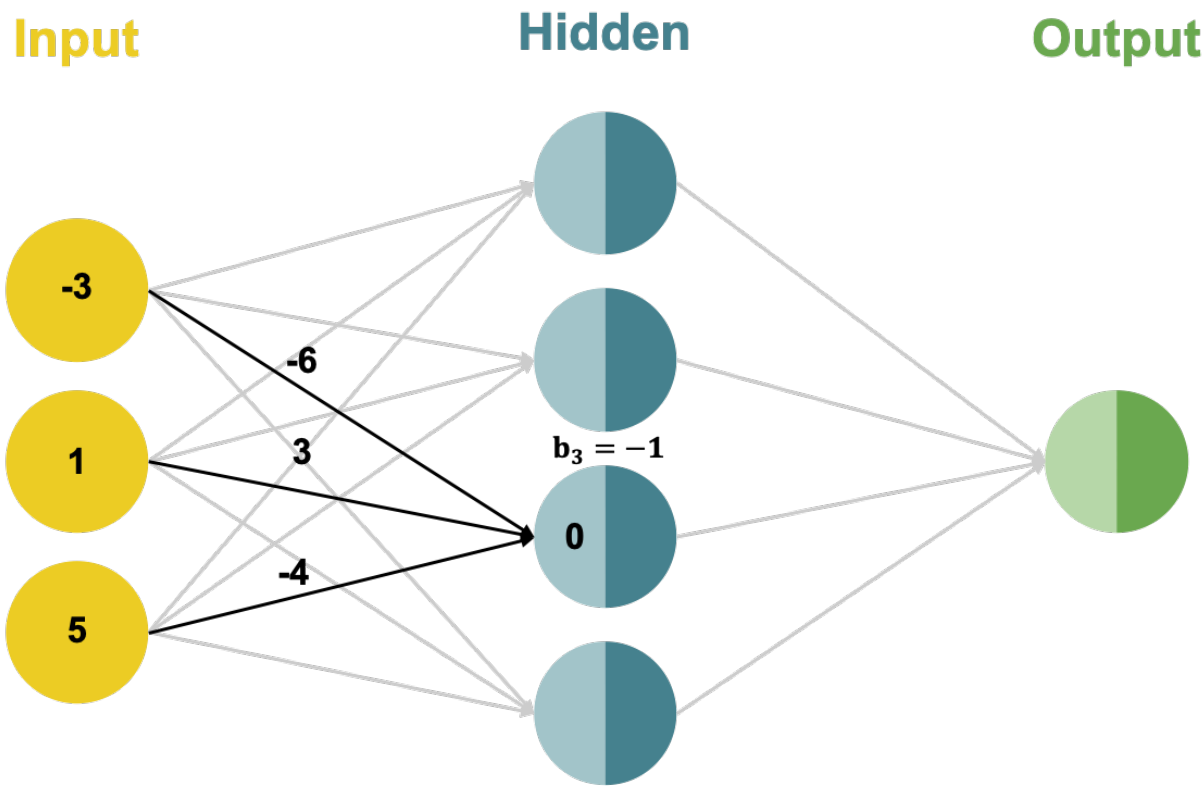
Each neuron in the hidden layer performs an **affine transformation** on the inputs:



$$z_{2,\text{in}} = w_{12}x_1 + w_{22}x_2 + w_{32}x_3 + b_2$$
$$z_{2,\text{in}} = 11 * (-3) + (-2) * 1 + 7 * 5 + 2 = 2$$

SINGLE HIDDEN LAYER NETWORKS: EXAMPLE

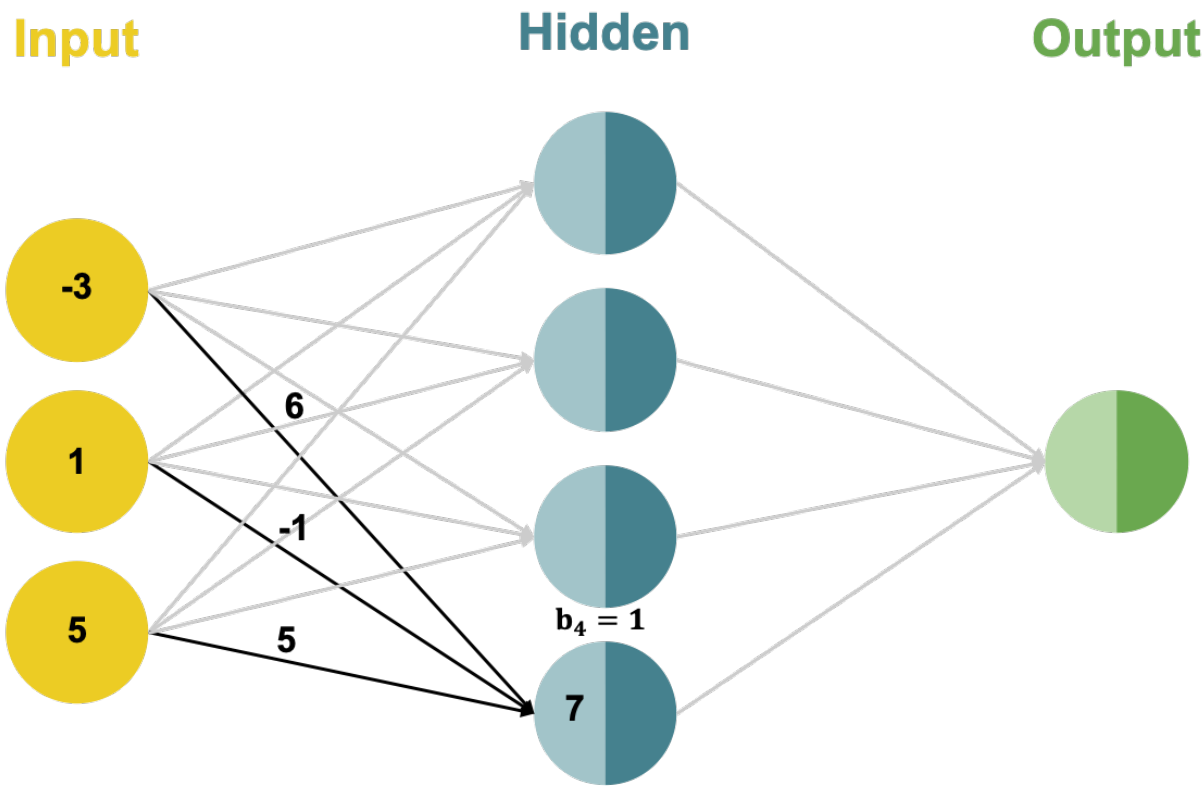
Each neuron in the hidden layer performs an **affine transformation** on the inputs:



$$z_{3,\text{in}} = w_{13}x_1 + w_{23}x_2 + w_{33}x_3 + b_3$$
$$z_{3,\text{in}} = (-6) * (-3) + 3 * 1 + (-4) * 5 - 1 = 0$$

SINGLE HIDDEN LAYER NETWORKS: EXAMPLE

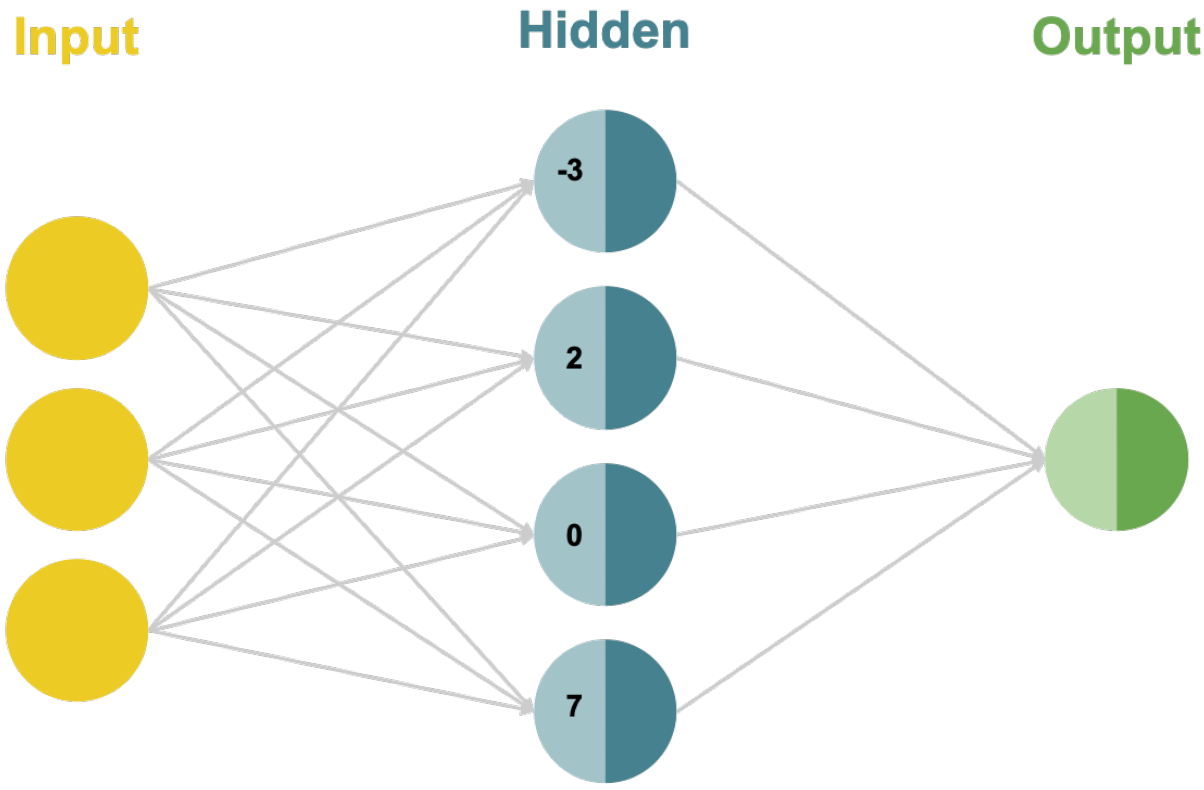
Each neuron in the hidden layer performs an **affine transformation** on the inputs:



$$z_{4,\text{in}} = w_{14}x_1 + w_{24}x_2 + w_{34}x_3 + b_4$$
$$z_{4,\text{in}} = 6 * (-3) + (-1) * 1 + 5 * 5 + 1 = 7$$

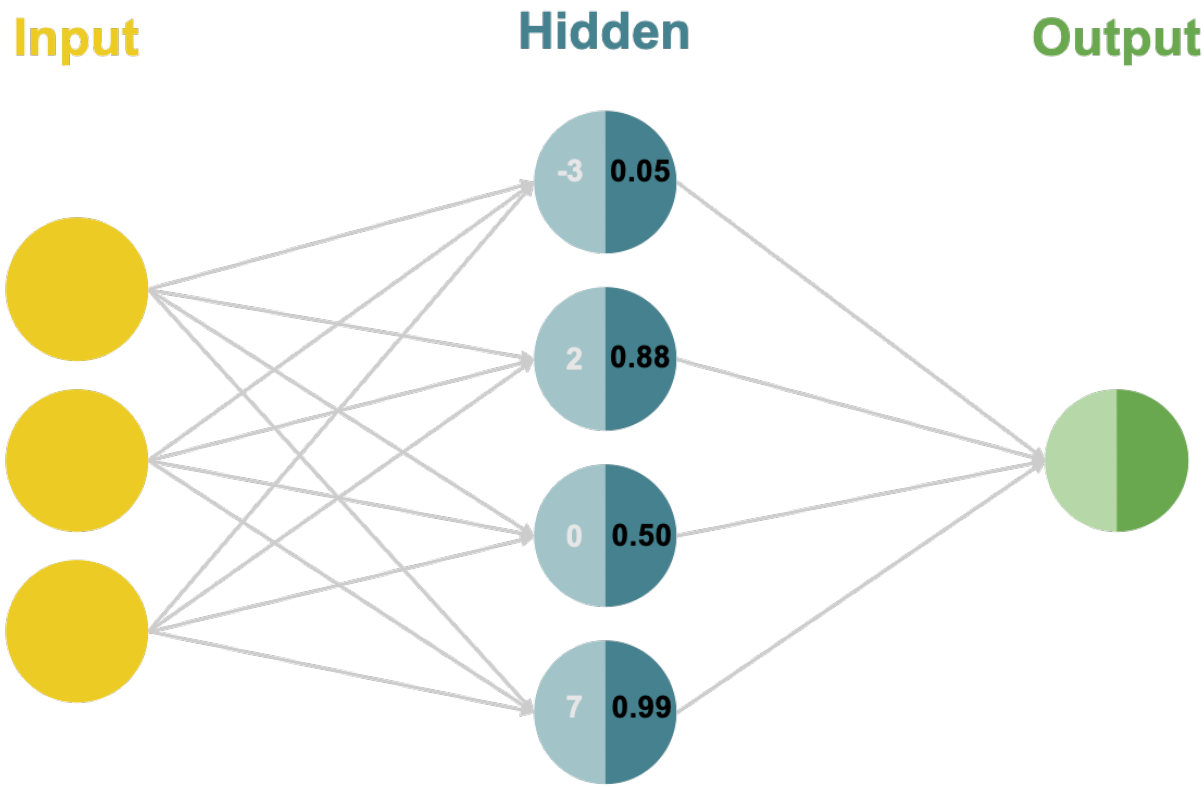
SINGLE HIDDEN LAYER NETWORKS: EXAMPLE

Each neuron in the hidden layer performs an **affine transformation** on the inputs:



SINGLE HIDDEN LAYER NETWORKS: EXAMPLE

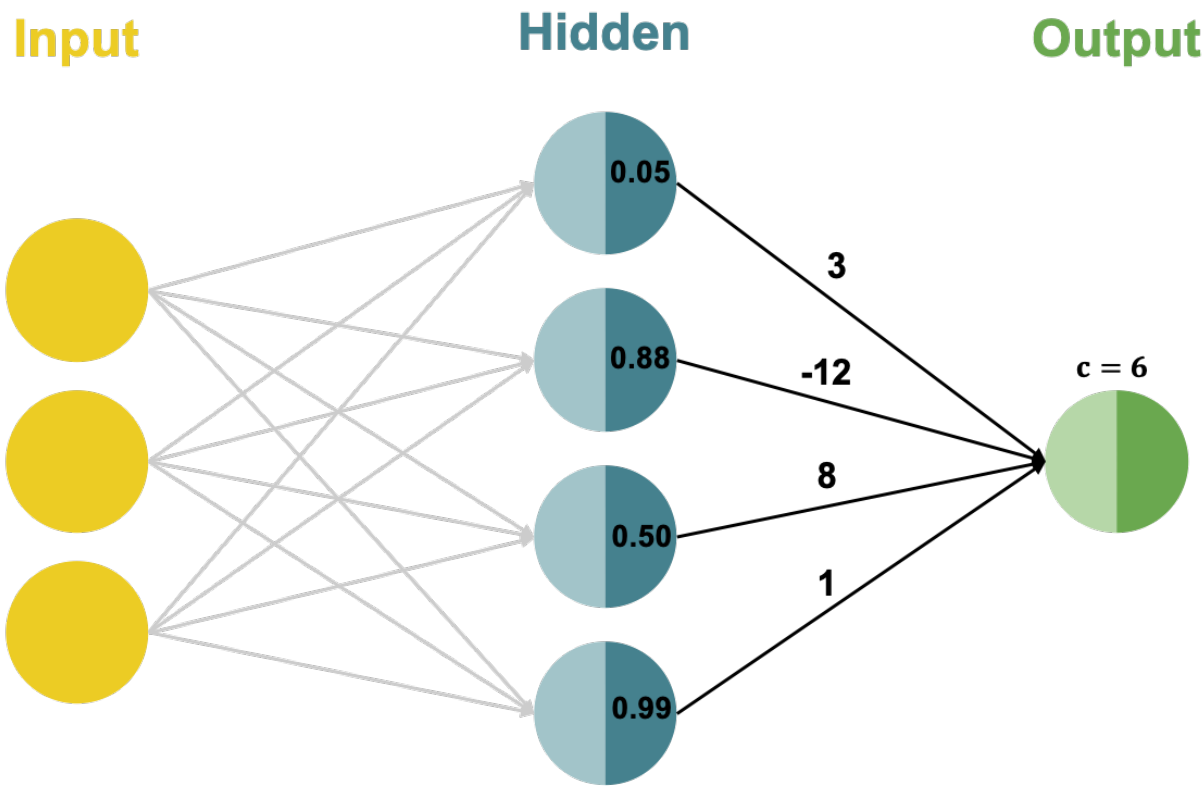
Each hidden neuron performs a non-linear **activation** transformation on the weight sum:



$$z_{\text{i,out}} = \sigma\left(z_{\text{i,in}}\right) = \frac{1}{1+e^{-z_{\text{in}}^{(i)}}}$$

SINGLE HIDDEN LAYER NETWORKS: EXAMPLE

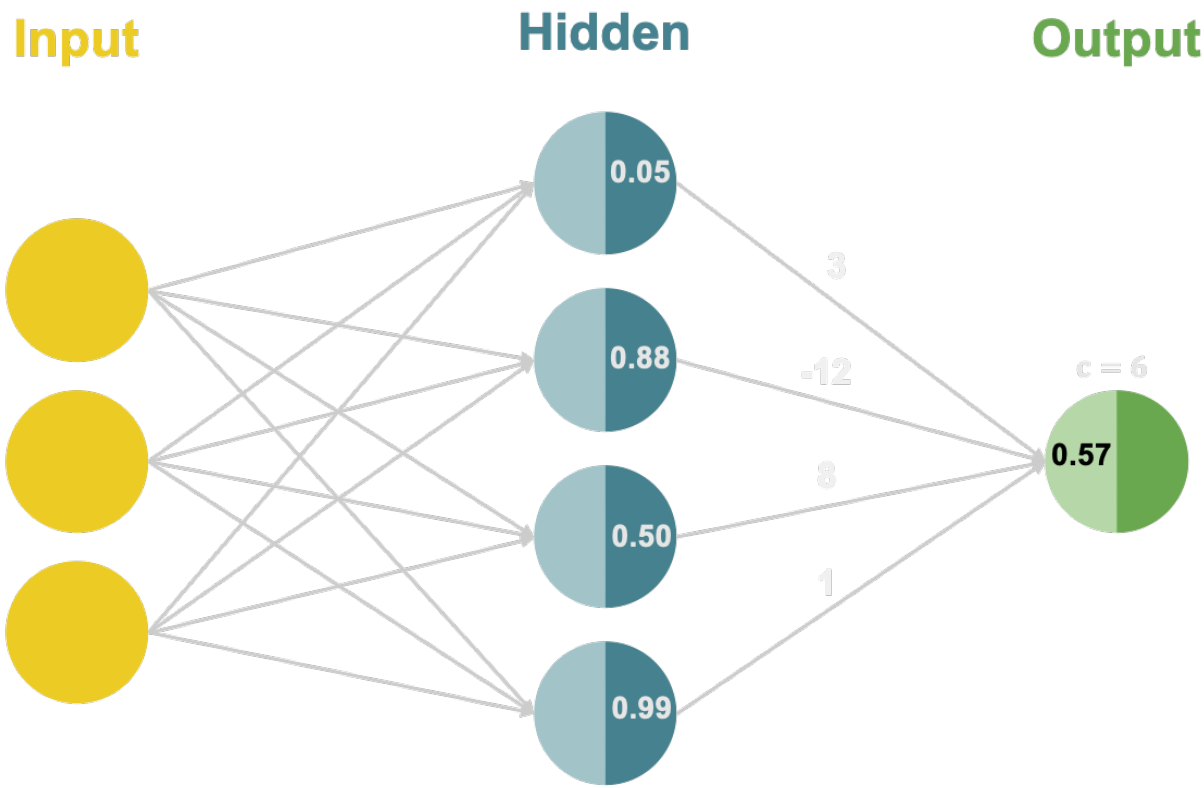
The output neuron performs an **affine transformation** on its inputs:



$$f_{\text{in}} = u_1 z_{1,\text{out}} + u_2 z_{2,\text{out}} + u_3 z_{3,\text{out}} + u_4 z_{4,\text{out}} + c$$

SINGLE HIDDEN LAYER NETWORKS: EXAMPLE

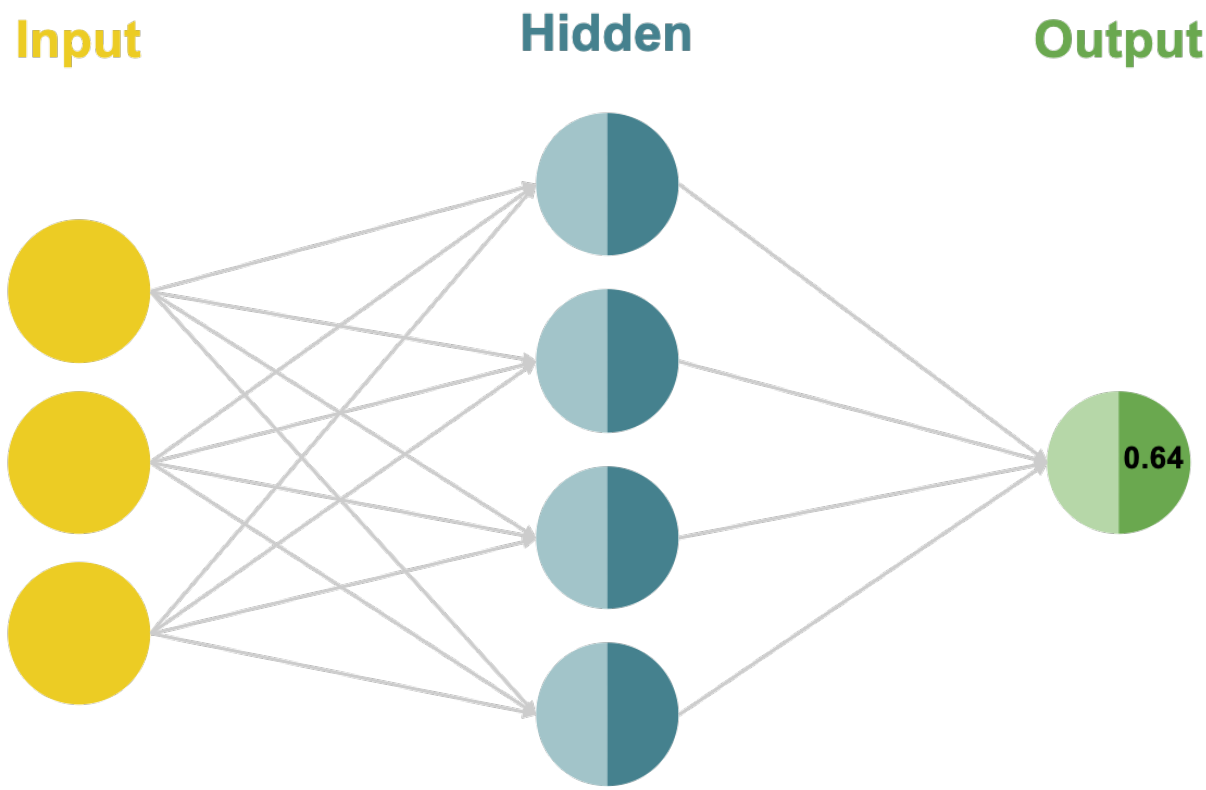
The output neuron performs an **affine transformation** on its inputs:



$$f_{\text{in}} = u_1 z_{1,\text{out}} + u_2 z_{2,\text{out}} + u_3 z_{3,\text{out}} + u_4 z_{4,\text{out}} + c$$
$$f_{\text{in}} = 3 * 0.05 + (-12) * 0.88 + 8 * 0.50 + 1 * 0.99 + 6 = 0.57$$

SINGLE HIDDEN LAYER NETWORKS: EXAMPLE

The output neuron performs a non-linear **activation** transformation on the weight sum:



$$f_{\text{out}} = \sigma(f_{\text{in}}) = \frac{1}{1+e^{-f_{\text{in}}}}$$
$$f_{\text{out}} = \frac{1}{1+e^{-0.57}} = 0.64$$

HIDDEN LAYER: ACTIVATION FUNCTION

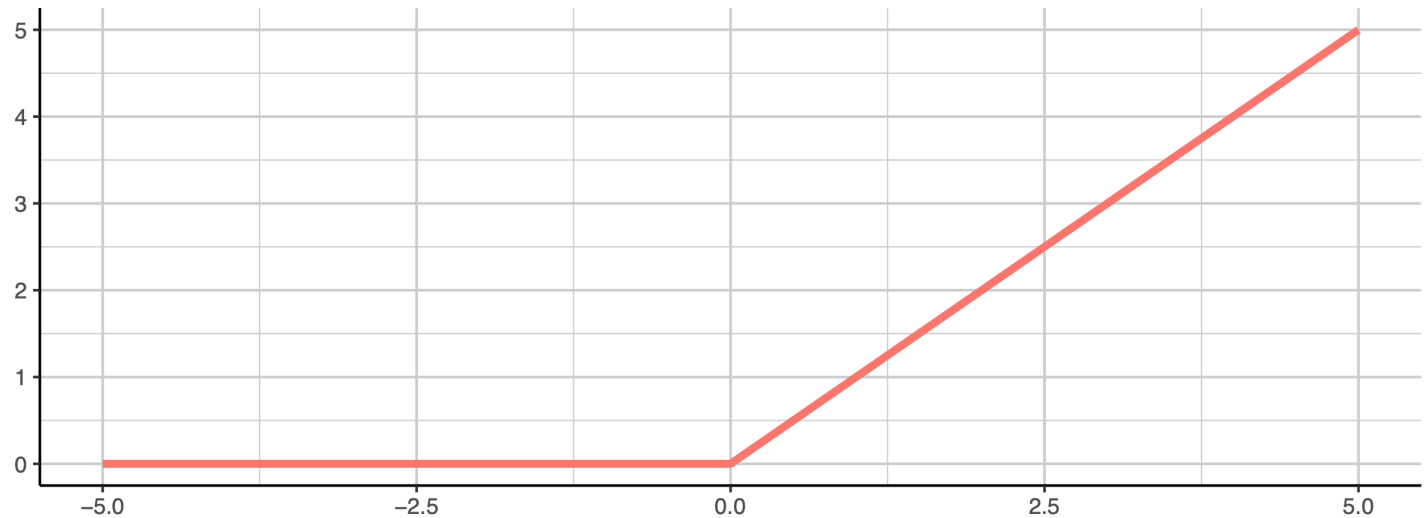
- If the hidden layer does not have a non-linear activation, the network can only learn linear decision boundaries.
- A lot of different activation functions exist.

HIDDEN LAYER: ACTIVATION FUNCTION

ReLU Activation:

- Currently the most popular choice is the ReLU (rectified linear unit):

$$\sigma(v) = \max(0, v)$$

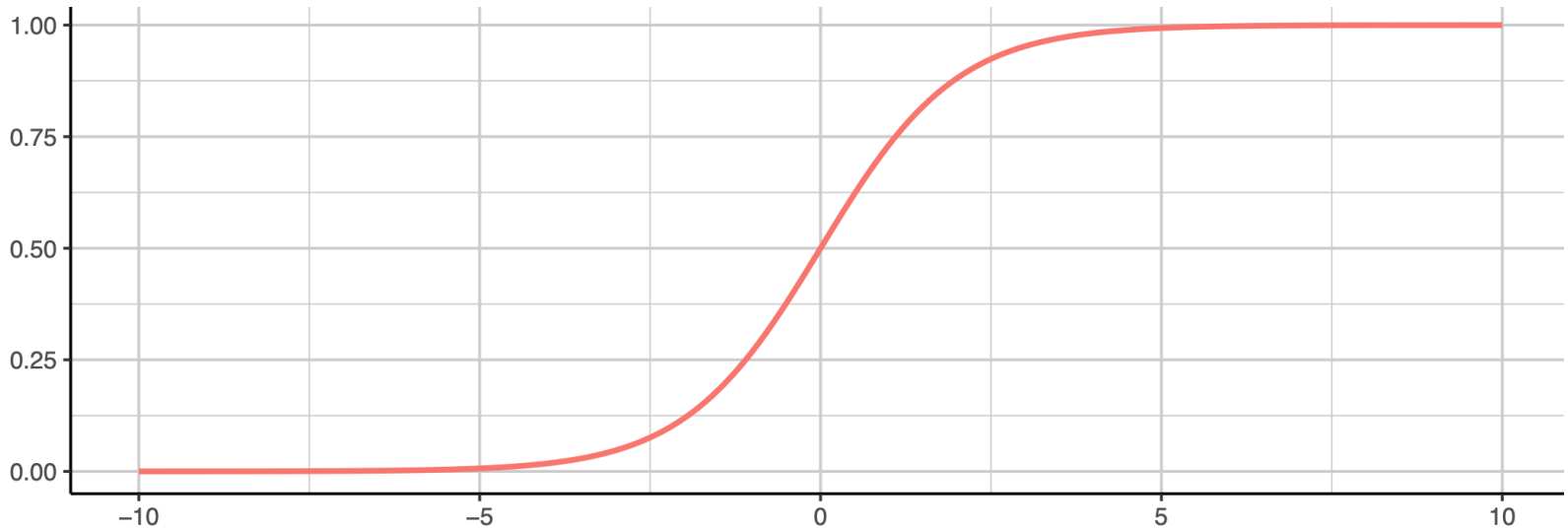


HIDDEN LAYER: ACTIVATION FUNCTION

Sigmoid Activation Function:

- The sigmoid function can be used even in the hidden layer:

$$\sigma(v) = \frac{1}{1 + \exp(-v)}$$



MULTI-CLASS CLASSIFICATION

- We have only considered regression and binary classification problems so far.
- How can we get a neural network to perform multiclass classification?

MULTI-CLASS CLASSIFICATION

- The first step is to add additional neurons to the output layer.
- Each neuron in the layer will represent a specific class (number of neurons in the output layer = number of classes).

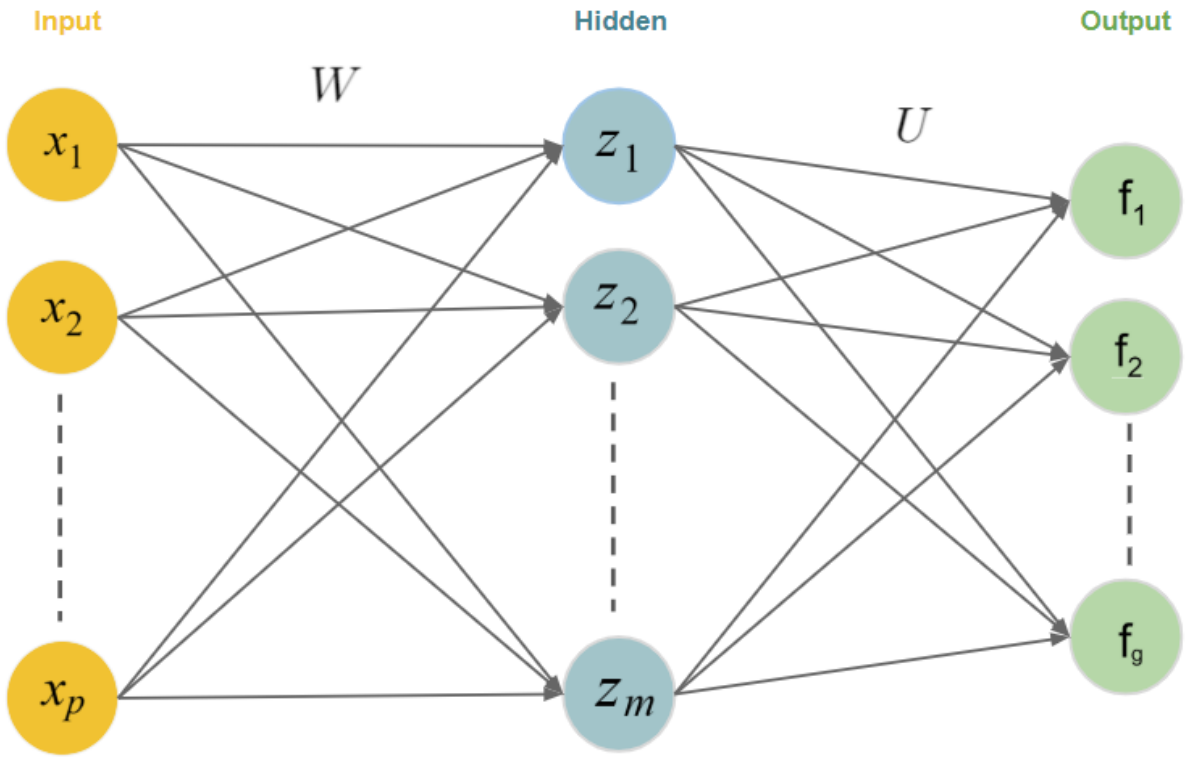


Figure: Structure of a single hidden layer, feed-forward neural network for g-class classification problems (bias term omitted).

MULTI-CLASS CLASSIFICATION

Notation:

- For g-class classification, g output units:

$$\mathbf{f} = (f_1, \ldots, f_g)$$

- m hidden neurons $z_1, \ldots, z_m$, with

$$z_j = \sigma(\mathbf{W}_j^T \mathbf{x}), \quad j = 1, \ldots, m.$$

- Compute linear combinations of derived features z:

$$f_{in,k} = \boldsymbol{U}_k^T \mathbf{z}, \quad \mathbf{z} = (z_1, \ldots, z_m)^T, \quad k = 1, \ldots, g$$

MULTI-CLASS CLASSIFICATION

- The second step is to apply a **softmax** activation function to the output layer.

- This gives us a probability distribution over g different possible classes:

$$f_{out,k} = \tau_k(f_{in,k}) = \frac{\exp(f_{in,k})}{\sum_{k'=1}^{g} \exp(f_{in,k'})}$$

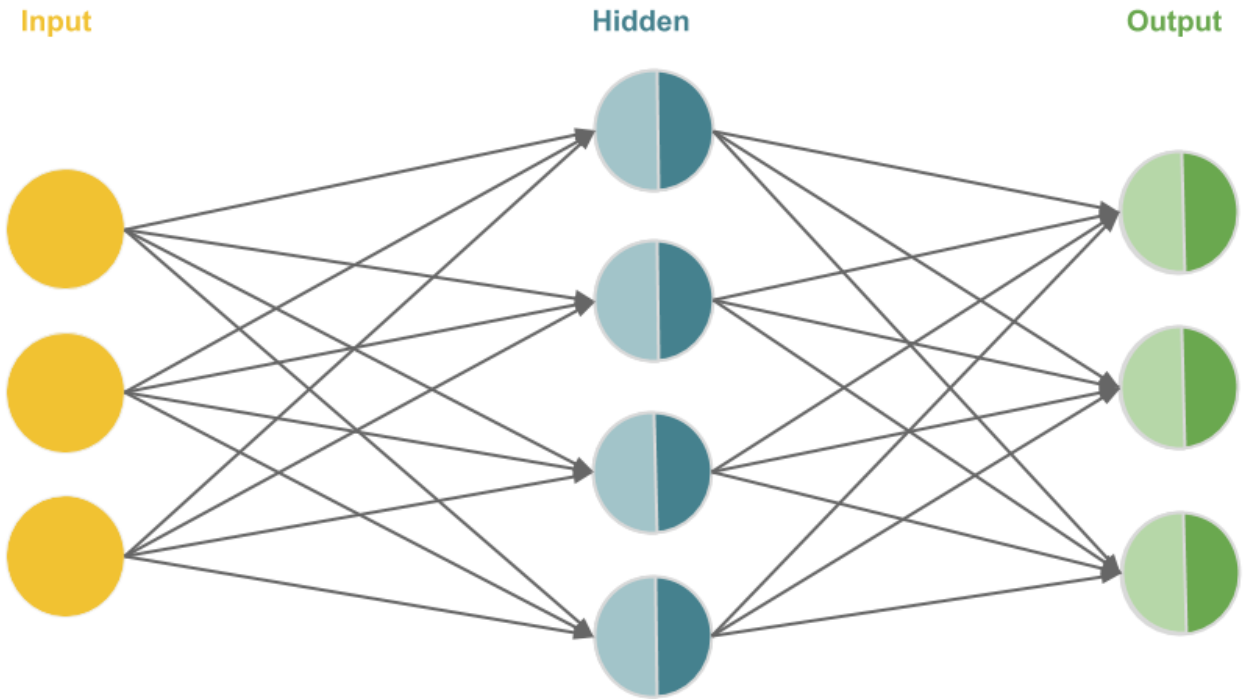
- This is the same transformation used in softmax regression!

- Derivative $\frac{\partial \tau(\mathbf{f}_{in})}{\partial \mathbf{f}_{in}} = \text{diag}(\tau(\mathbf{f}_{in})) - \tau(\mathbf{f}_{in})\tau(\mathbf{f}_{in})^T$

- It is a "smooth" approximation of the argmax operation, so $\tau((1, 1000, 2)^T) \approx (0, 1, 0)^T$ (picks out 2nd element!).

MULTI-CLASS CLASSIFICATION: EXAMPLE

Forward pass (Hidden: Sigmoid, Output: Softmax).



$$\begin{pmatrix} 3 & -9 & 2 \\ 11 & -2 & 7 \\ -6 & 3 & -4 \\ 6 & -1 & 5 \end{pmatrix} \begin{pmatrix} 5 \\ 2 \\ -1 \\ 1 \end{pmatrix}$$

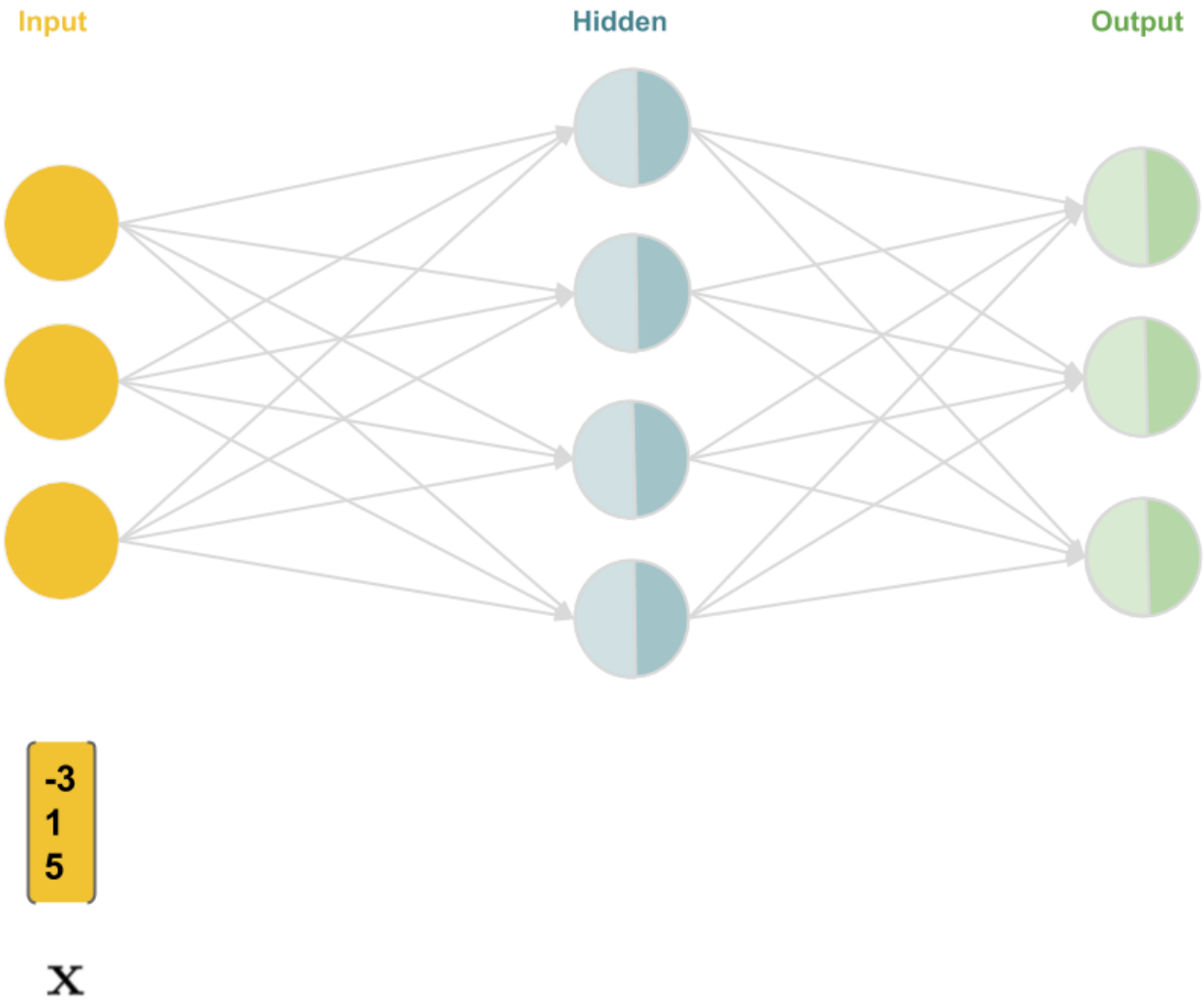
W^T $\mathbf{b}$

$$\begin{pmatrix} 3 & -12 & 8 & 1 \\ 2 & -3 & 9 & 1 \\ -5 & 1 & -1 & 7 \end{pmatrix} \begin{pmatrix} 6 \\ 0 \\ -8 \end{pmatrix}$$

U^T $\mathbf{c}$

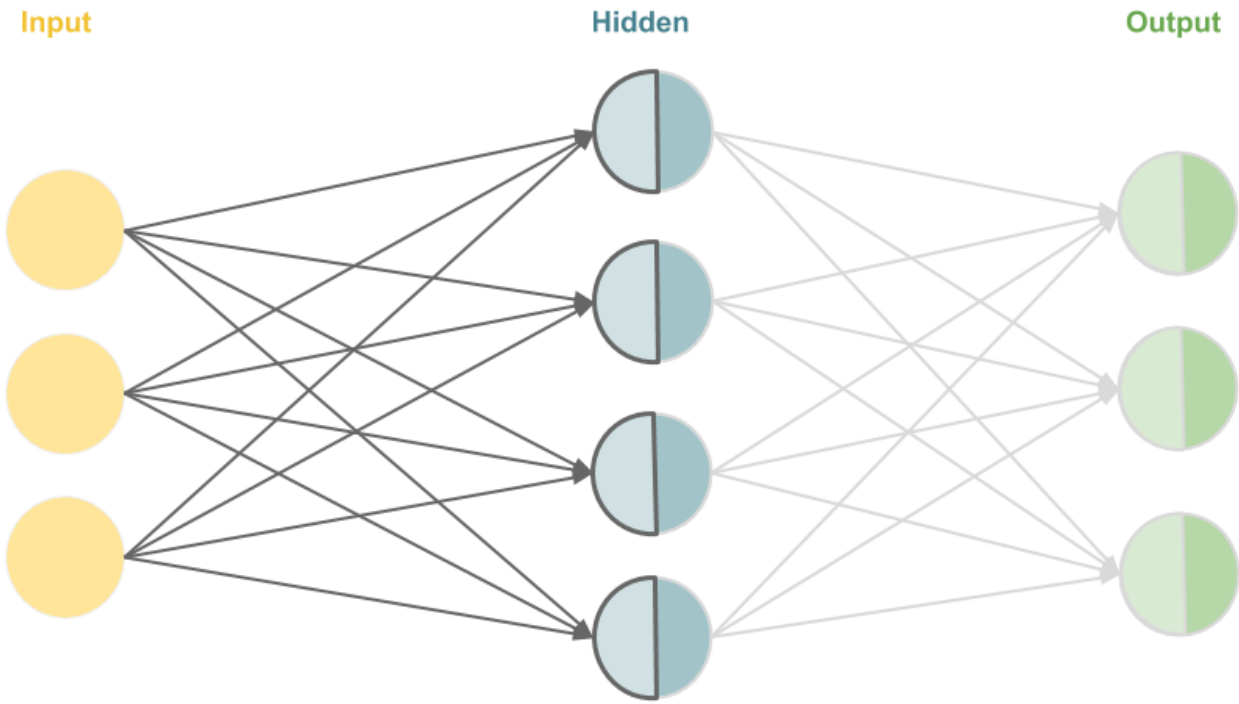
MULTI-CLASS CLASSIFICATION: EXAMPLE

Forward pass (Hidden: Sigmoid, Output: Softmax).



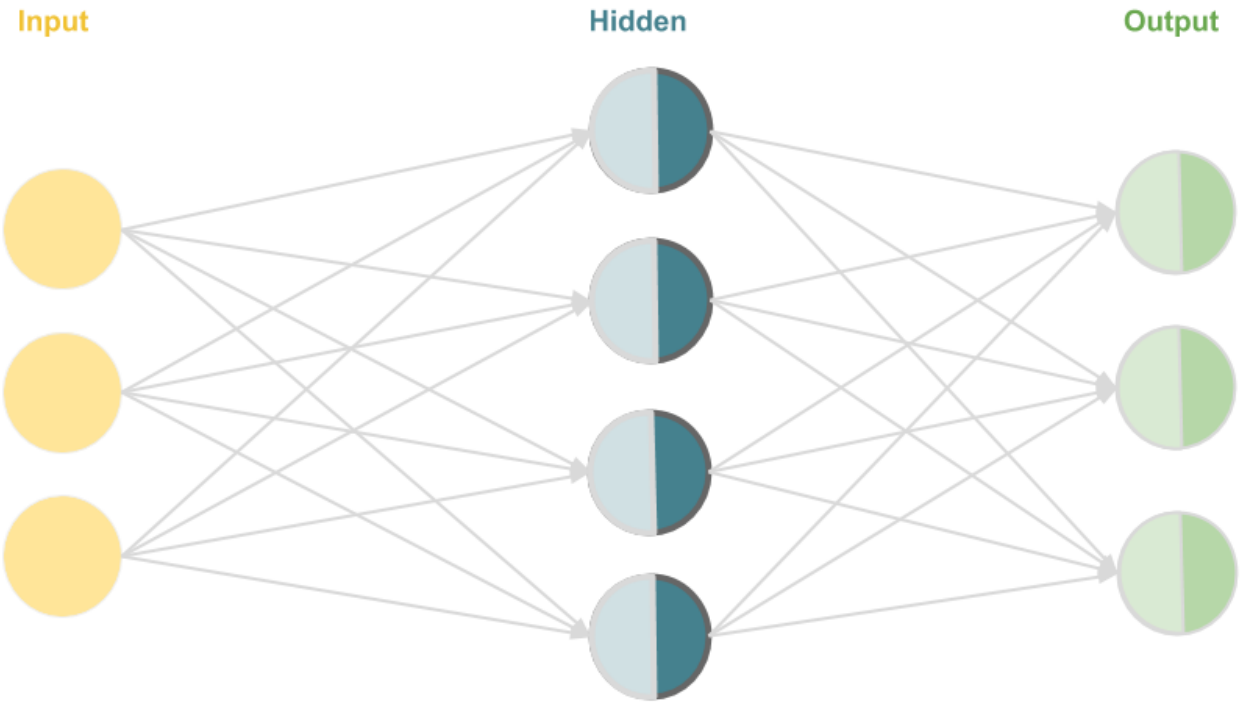
MULTI-CLASS CLASSIFICATION: EXAMPLE

Forward pass (Hidden: Sigmoid, Output: Softmax).



MULTI-CLASS CLASSIFICATION: EXAMPLE

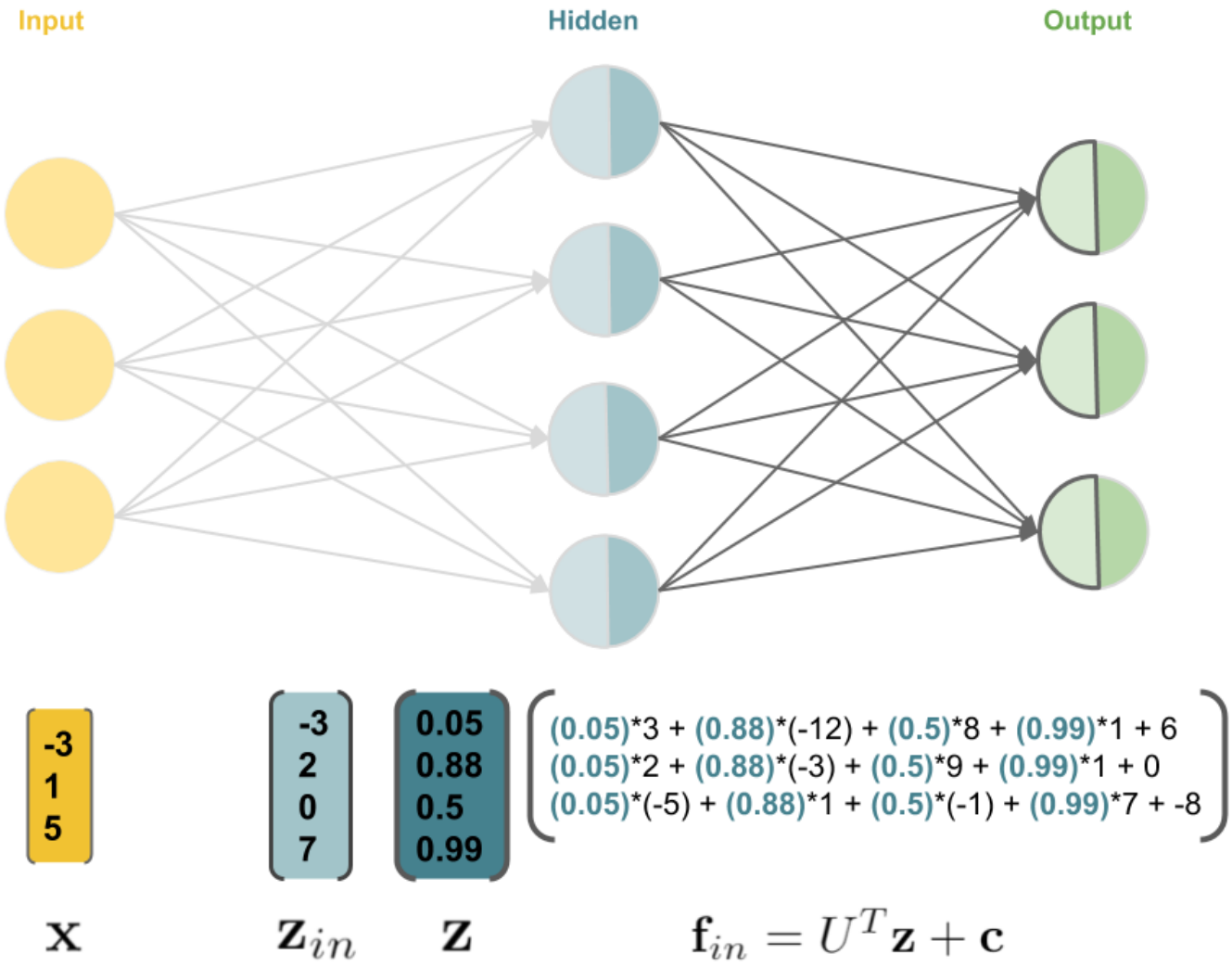
Forward pass (Hidden: Sigmoid, Output: Softmax).



$$\mathbf{x} = \begin{bmatrix} -3 \\ 1 \\ 5 \end{bmatrix} \qquad \mathbf{z}_{in} = \begin{bmatrix} -3 \\ 2 \\ 0 \\ 7 \end{bmatrix} \qquad \mathbf{z} = \mathbf{z}_{out} = \sigma(\mathbf{z}_{in}) = \begin{bmatrix} 1/(1+\exp(-(-3))) \\ 1/(1+\exp(-2)) \\ 1/(1+\exp(-0)) \\ 1/(1+\exp(-7)) \end{bmatrix}$$

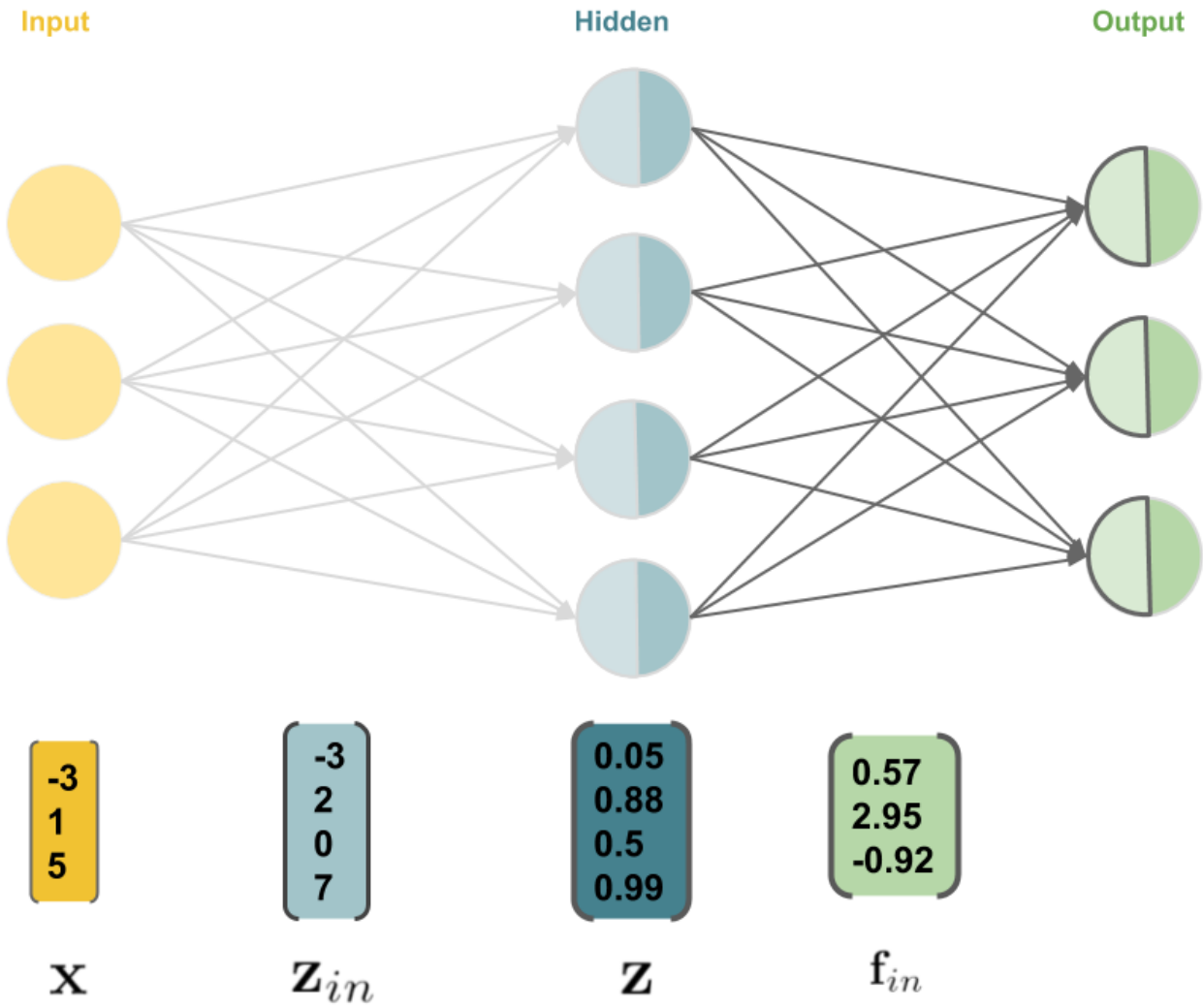
MULTI-CLASS CLASSIFICATION: EXAMPLE

Forward pass (Hidden: Sigmoid, Output: Softmax).



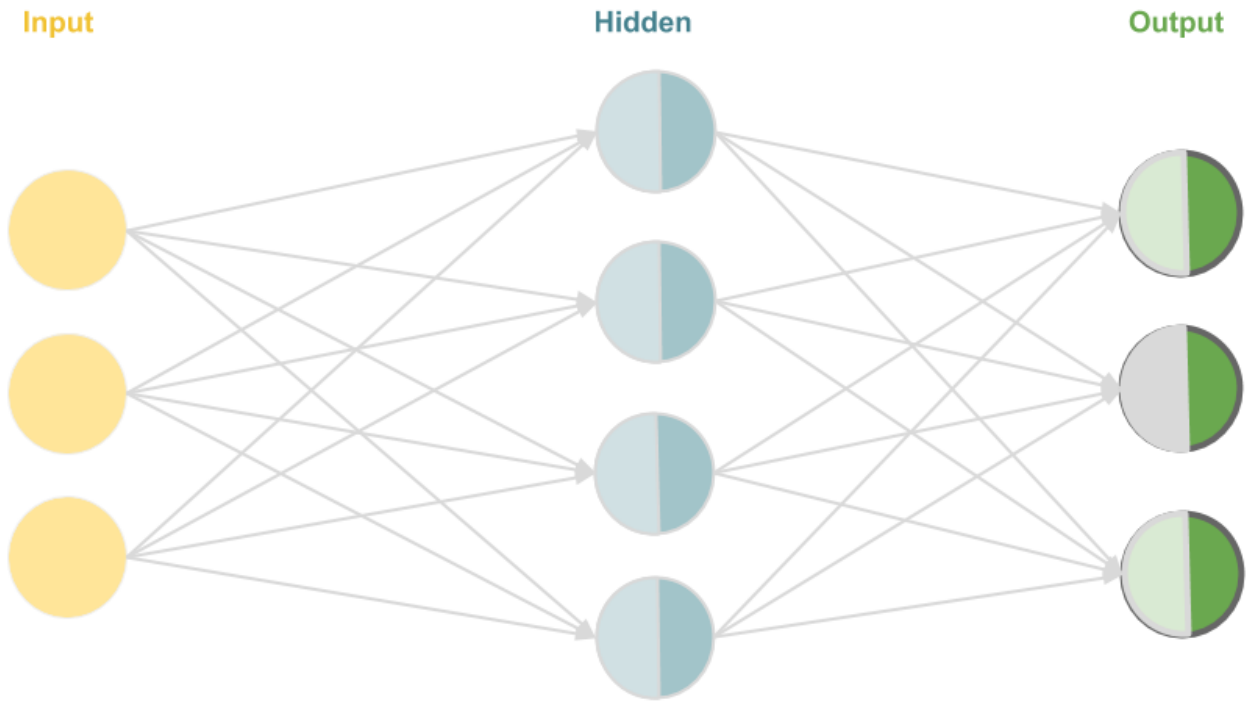
MULTI-CLASS CLASSIFICATION: EXAMPLE

Forward pass (Hidden: Sigmoid, Output: Softmax).



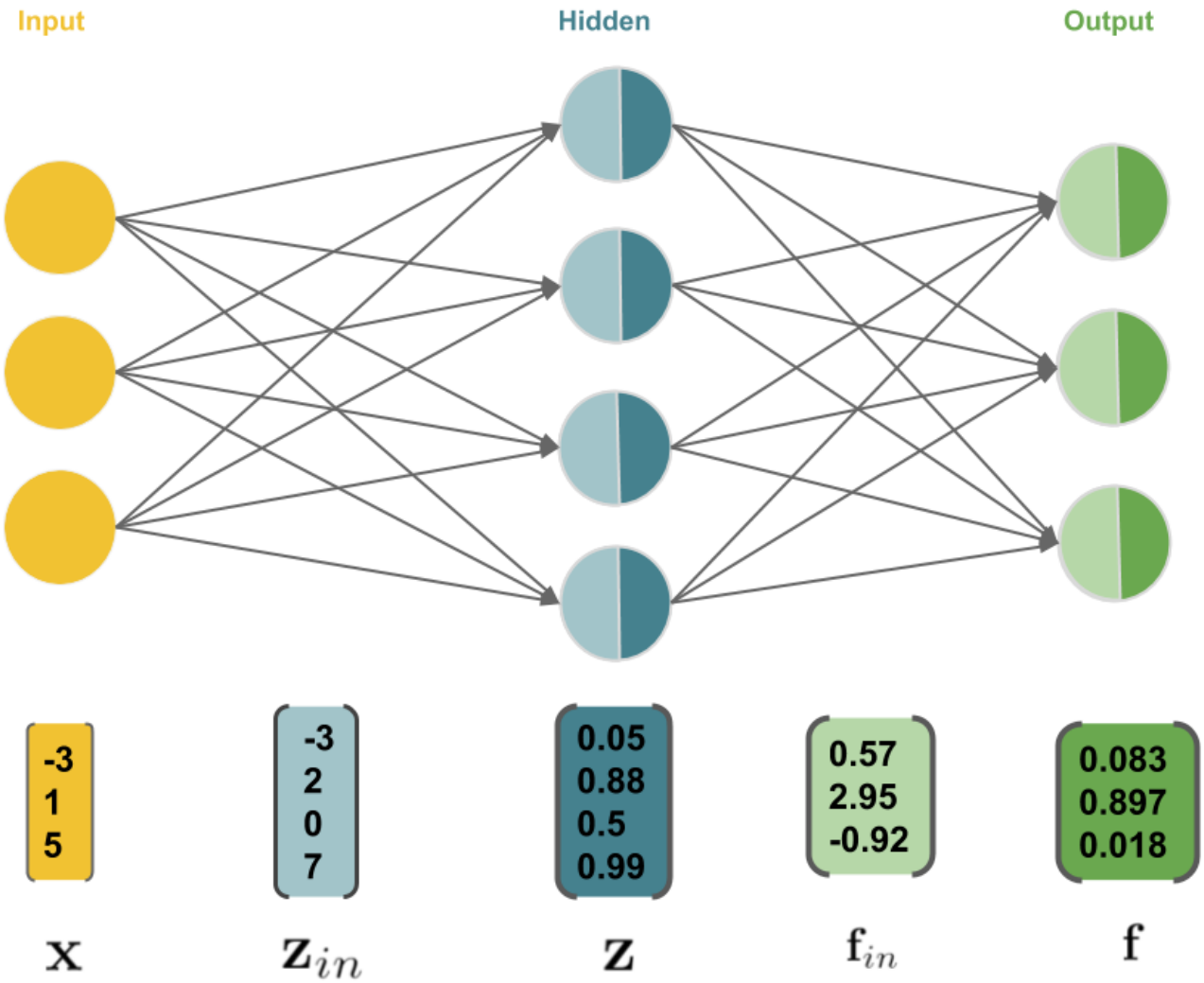
MULTI-CLASS CLASSIFICATION: EXAMPLE

Forward pass (Hidden: Sigmoid, Output: Softmax).



MULTI-CLASS CLASSIFICATION: EXAMPLE

Forward pass (Hidden: Sigmoid, Output: Softmax).



OPTIMIZATION: SOFTMAX LOSS

- The loss function for a softmax classifier is

$$L(y, f(\mathbf{x})) = -\sum_{k=1}^{g} [y = k] \log \left(\frac{\exp(f_{in,k})}{\sum_{k'=1}^{g} \exp(f_{in,k'})} \right)$$

where $[y = k] = \begin{cases} 1 & \text{if } y = k \\ 0 & \text{otherwise} \end{cases}$.

- This is equivalent to the cross-entropy loss when the label vector $\boldsymbol{y}$ is one-hot coded (e.g. $\mathbf{y} = (0, 0, 1, 0)^T$).
- Optimization: Again, there is no analytic solution.

INTRODUCTION TO DEEP LEARNING

FEEDFORWARD NEURAL NETWORKS

- We will now extend the model class once again, such that we allow an arbitrary amount l of hidden layers.

- The general term for this model class is (multi-layer) **feedforward networks** (inputs are passed through the network from left to right, no feedback-loops are allowed)

FEEDFORWARD NEURAL NETWORKS

- We can characterize those models by the following chain structure:

$$f(\mathbf{x}) = \tau \circ \phi \circ \sigma^{(l)} \circ \phi^{(l)} \circ \sigma^{(l-1)} \circ \phi^{(l-1)} \circ \ldots \circ \sigma^{(1)} \circ \phi^{(1)}$$

  where $\sigma^{(i)}$ and $\phi^{(i)}$ are the activation function and the weighted sum of hidden layer i, respectively. τ and ϕ are the corresponding components of the output layer.

- Each hidden layer has:
  - an associated weight matrix $\mathbf{W}^{(i)}$, bias $\mathbf{b}^{(i)}$, and activations $\mathbf{z}^{(i)}$ for $i \in \{1 \ldots l\}$.
  - $\mathbf{z}^{(i)} = \sigma^{(i)}(\phi^{(i)}) = \sigma^{(i)}(\mathbf{W}^{(i)T}\mathbf{z}^{(i-1)} + \mathbf{b}^{(i)})$, where $\mathbf{z}^{(0)} = \mathbf{x}$.

- Again, without non-linear activations in the hidden layers, the network can only learn linear decision boundaries.

FEEDFORWARD NEURAL NETWORKS

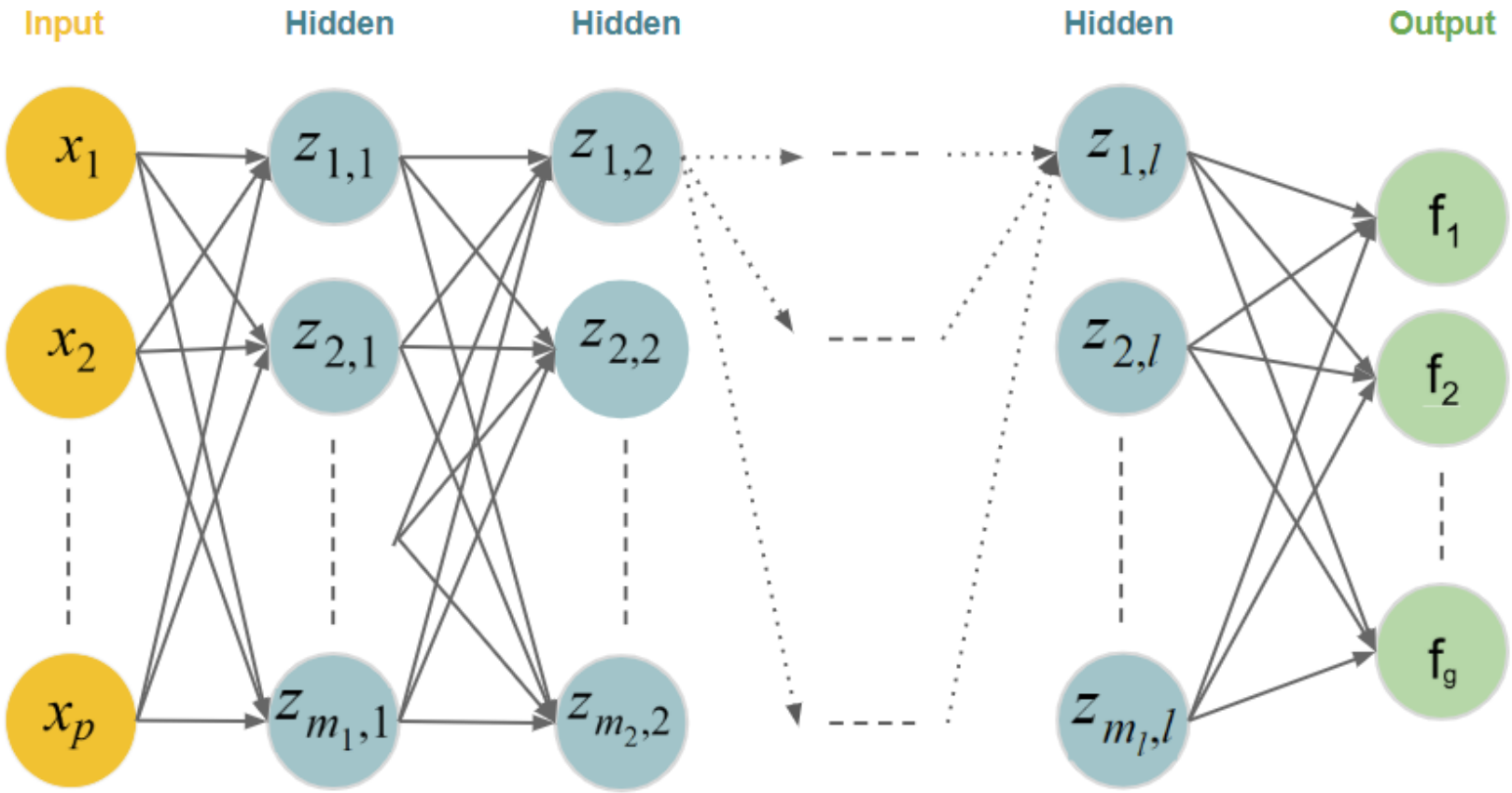
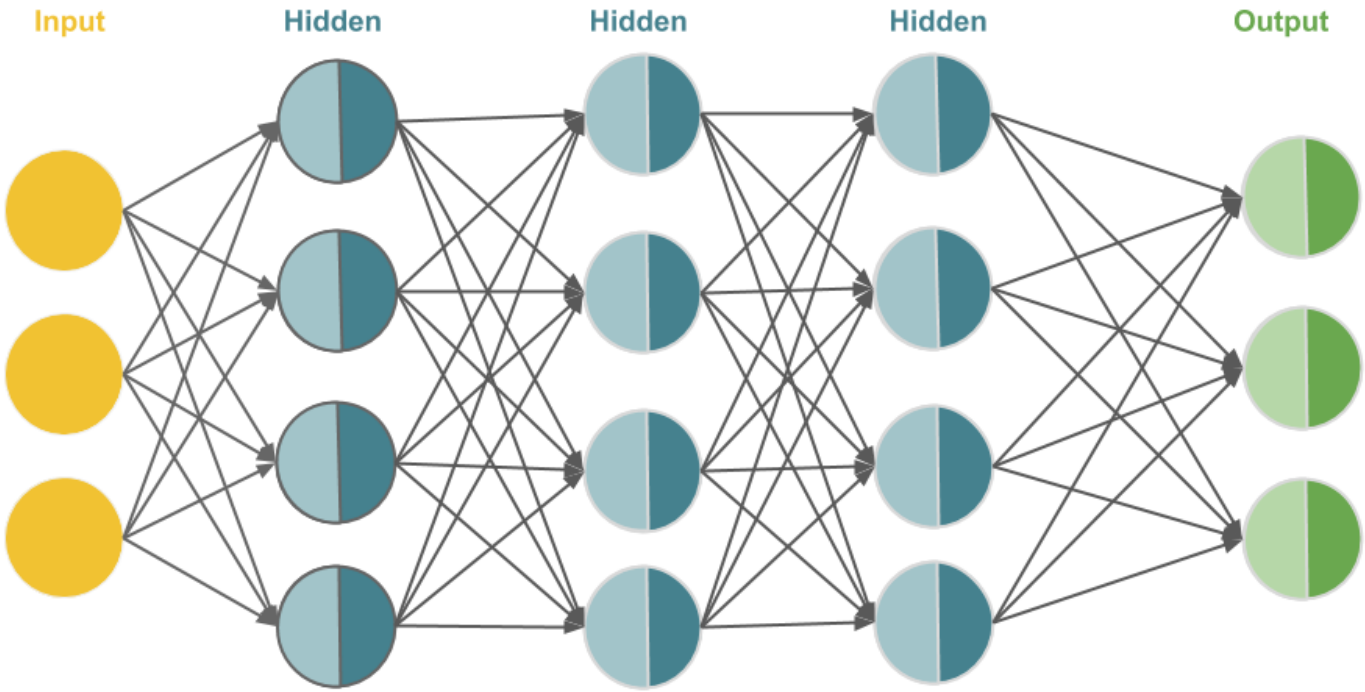


Figure: Structure of a deep neural network with *l* hidden layers (bias terms omitted).

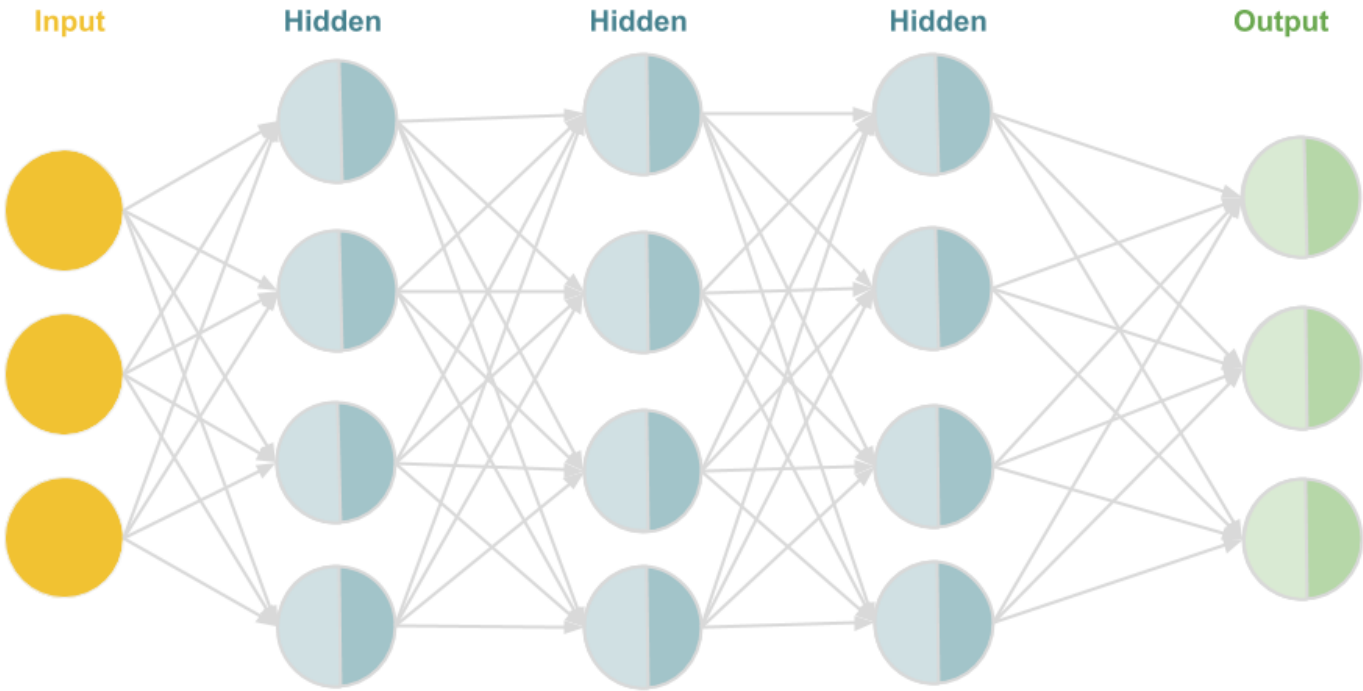
FEEDFORWARD NEURAL NETWORKS: EXAMPLE



$$\begin{pmatrix} 13 & -9 & 2 \\ -8 & 0 & 3 \\ 4 & -1 & 5 \\ -3 & 12 & 7 \end{pmatrix} \begin{pmatrix} 5 \\ -2 \\ 2 \\ 11 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & -4 & 1 \\ 0 & 11 & 2 & -14 \\ -1 & 5 & -2 & 16 \\ 0 & -9 & -3 & 4 \end{pmatrix} \begin{pmatrix} -5 \\ 3 \\ 1 \\ -8 \end{pmatrix} \quad \begin{pmatrix} 1 & -2 & -18 & -7 \\ 3 & -4 & 8 & 0 \\ -2 & 1 & 21 & 5 \\ 2 & -2 & 11 & -13 \end{pmatrix} \begin{pmatrix} 4 \\ -6 \\ 1 \\ -17 \end{pmatrix} \quad \begin{pmatrix} 9 & 3 & -1 & -4 \\ -8 & -2 & 14 & 3 \\ 13 & 2 & -9 & -1 \end{pmatrix} \begin{pmatrix} -1 \\ -4 \\ -30 \end{pmatrix}$$

$$(W^{(1)})^T \quad \mathbf{b}^{(1)} \quad (W^{(2)})^T \quad \mathbf{b}^{(2)} \quad (W^{(3)})^T \quad \mathbf{b}^{(3)} \quad U^T \quad \mathbf{c}$$

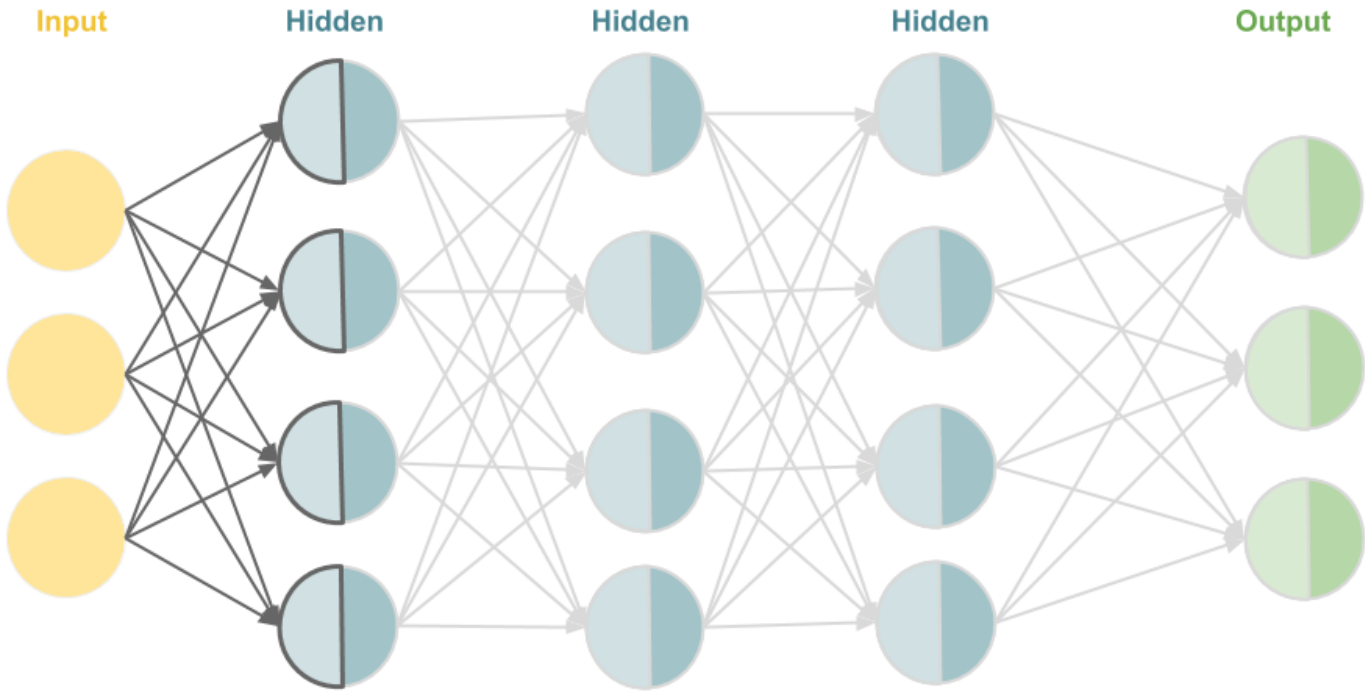
FEEDFORWARD NEURAL NETWORKS: EXAMPLE



$$\begin{bmatrix} 7 \\ 1 \\ -4 \end{bmatrix}$$

$$\mathbf{x}$$

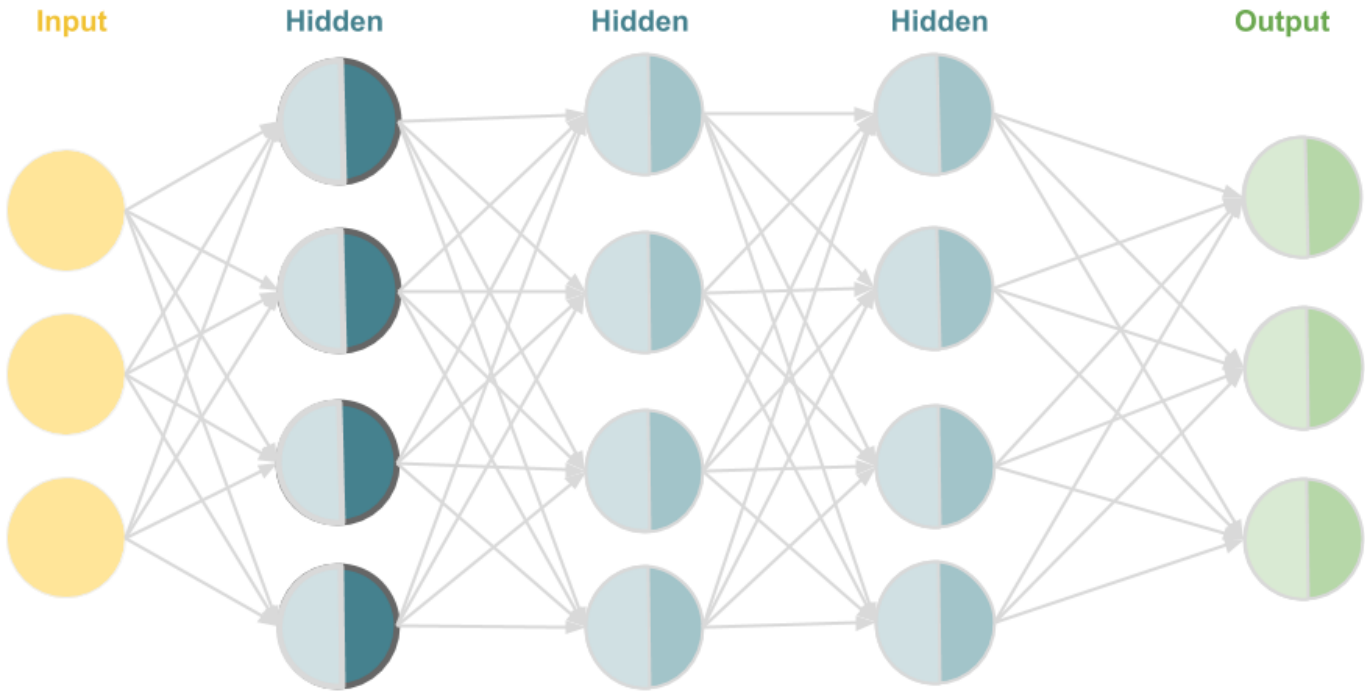
FEEDFORWARD NEURAL NETWORKS: EXAMPLE



$$\begin{matrix}7\\1\\-4\end{matrix}\begin{pmatrix}7*13 + 1*(-9) + (-4)*2 + 5\\7*(-8) + 1*0 + (-4)*3 + (-2)\\7*4 + 1*(-1) + (-4)*5 + 2\\7*(-3) + 1*12 + (-4)*7 + 11\end{pmatrix}$$

$\mathbf{x}$ $\quad \mathbf{z}_{in}^{(1)} = W^{(1)T}\mathbf{x} + \mathbf{b}^{(1)}$

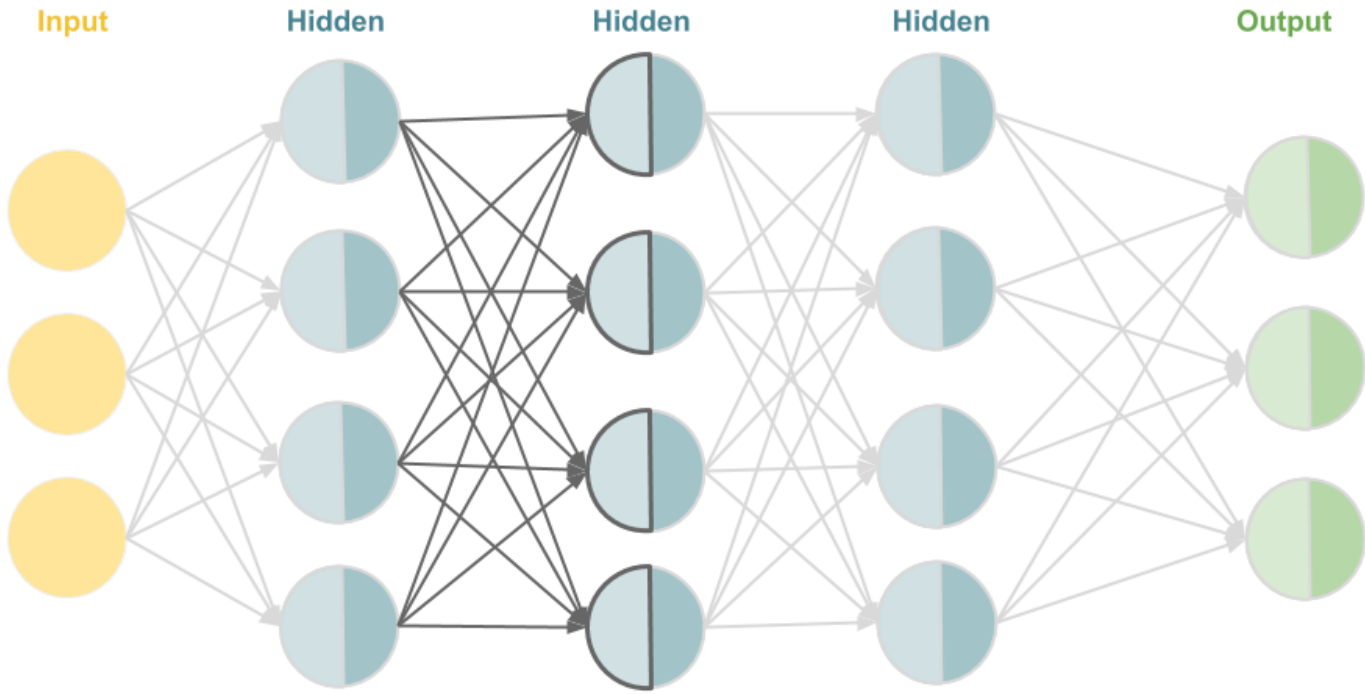
FEEDFORWARD NEURAL NETWORKS: EXAMPLE



$$\begin{bmatrix} 7 \\ 1 \\ -4 \end{bmatrix} \quad \begin{bmatrix} 79 \\ -70 \\ 9 \\ -26 \end{bmatrix} \quad \begin{bmatrix} \max(0, 79) \\ \max(0, -70) \\ \max(0, 9) \\ \max(0, -26) \end{bmatrix}$$

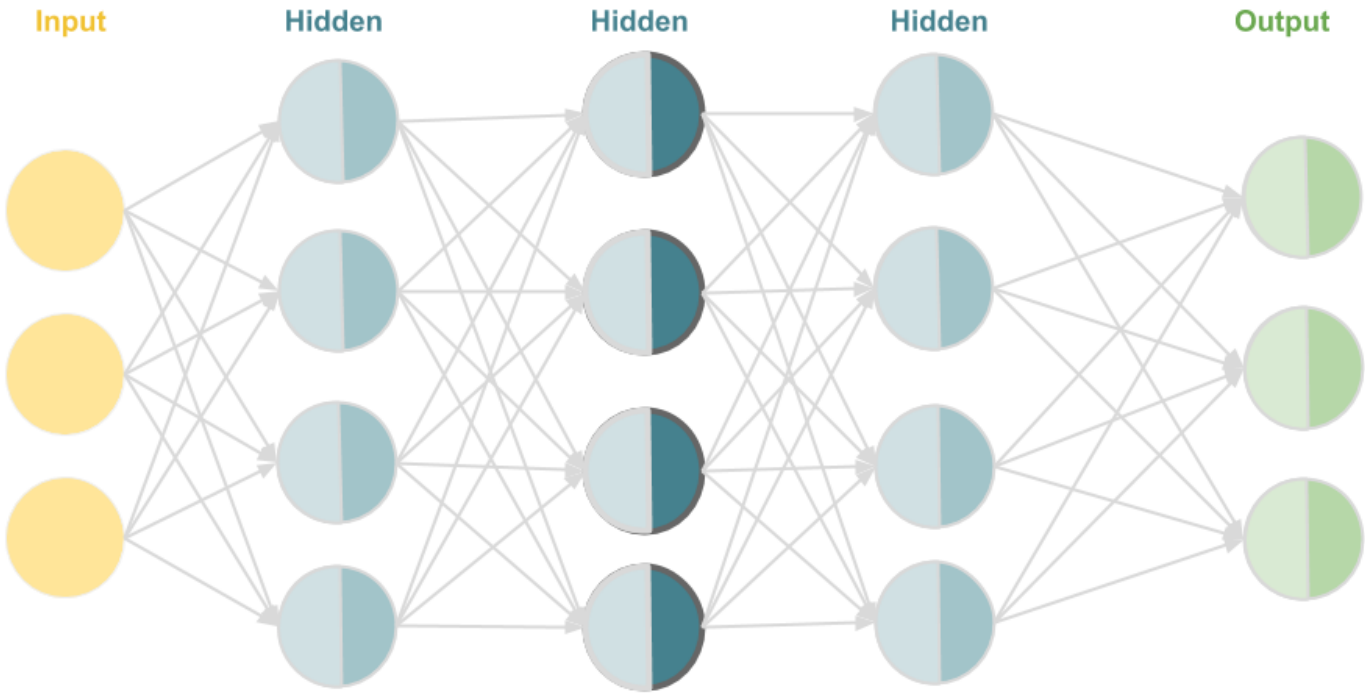
$\mathbf{x}$ $\quad \mathbf{z}_{in}^{(1)}$ $\quad \mathbf{z}^{(1)} = \mathbf{z}_{out}^{(1)} = \sigma(\mathbf{z}_{in}^{(1)})$

FEEDFORWARD NEURAL NETWORKS: EXAMPLE

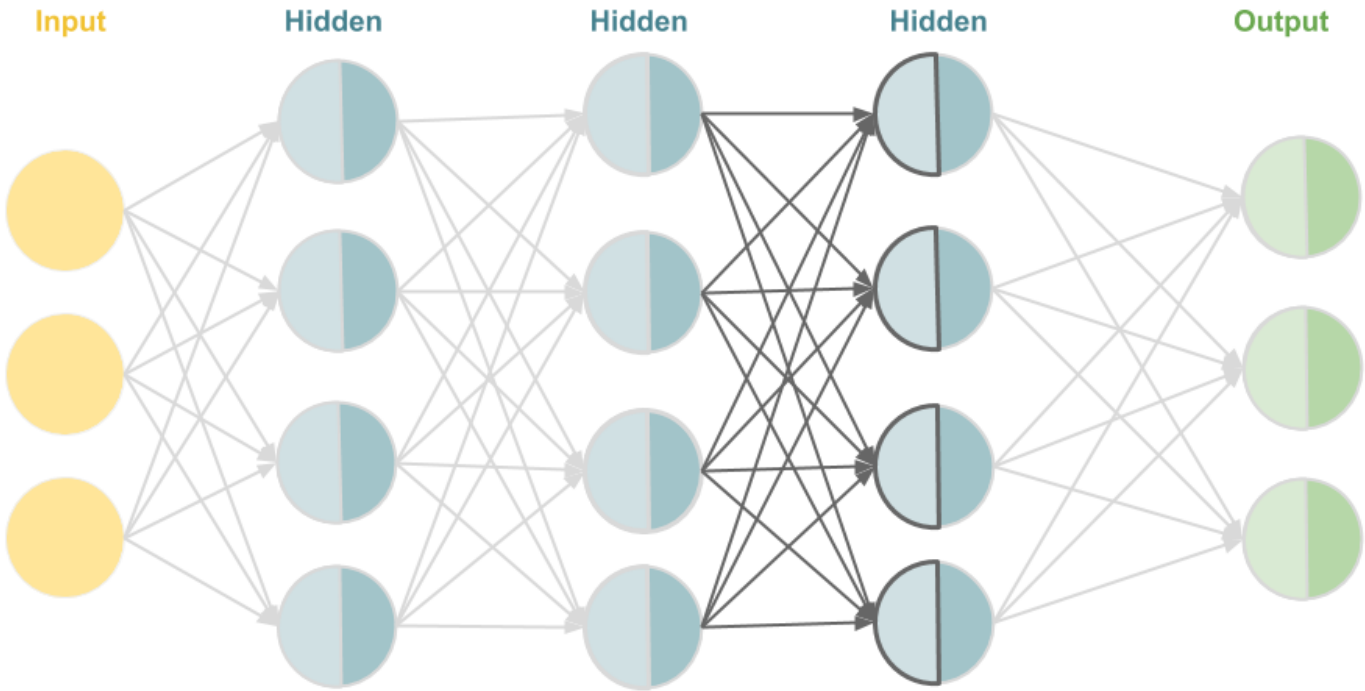


$\mathbf{x}$	$\mathbf{z}_{in}^{(1)}$	$\mathbf{z}^{(1)}$	$\mathbf{z}_{in}^{(2)} = W^{(2)T}\mathbf{z}^{(1)} + \mathbf{b}^{(2)}$
7 1 -4	79 -70 9 -26	79 0 9 0	79*1 + 0*0 + 9*(-4) + 0*1 + (-5) 79*0 + 0*11 + 9*2 + 0*(-14) + 3 79*(-1) + 0*5 + 9*(-2) + 0*16 + 1 79*0 + 0*(-9) + 9*(-3) + 0*4 + (-8)

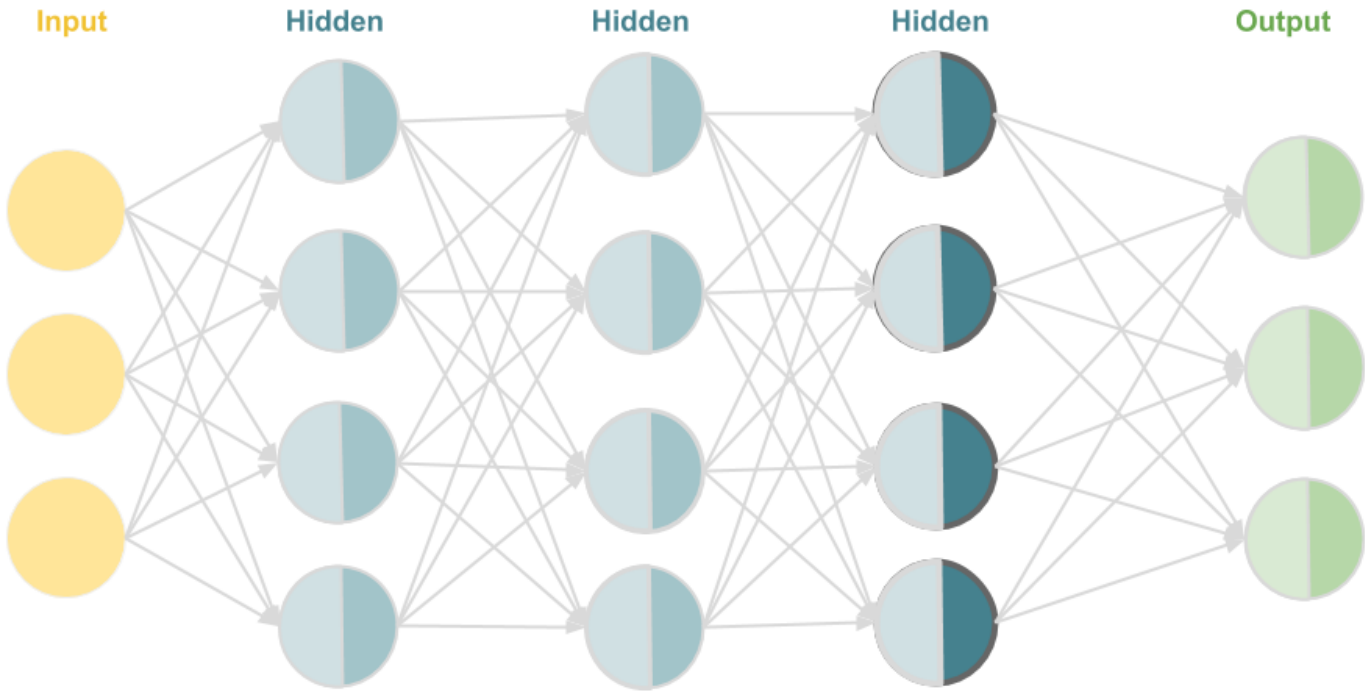
FEEDFORWARD NEURAL NETWORKS: EXAMPLE



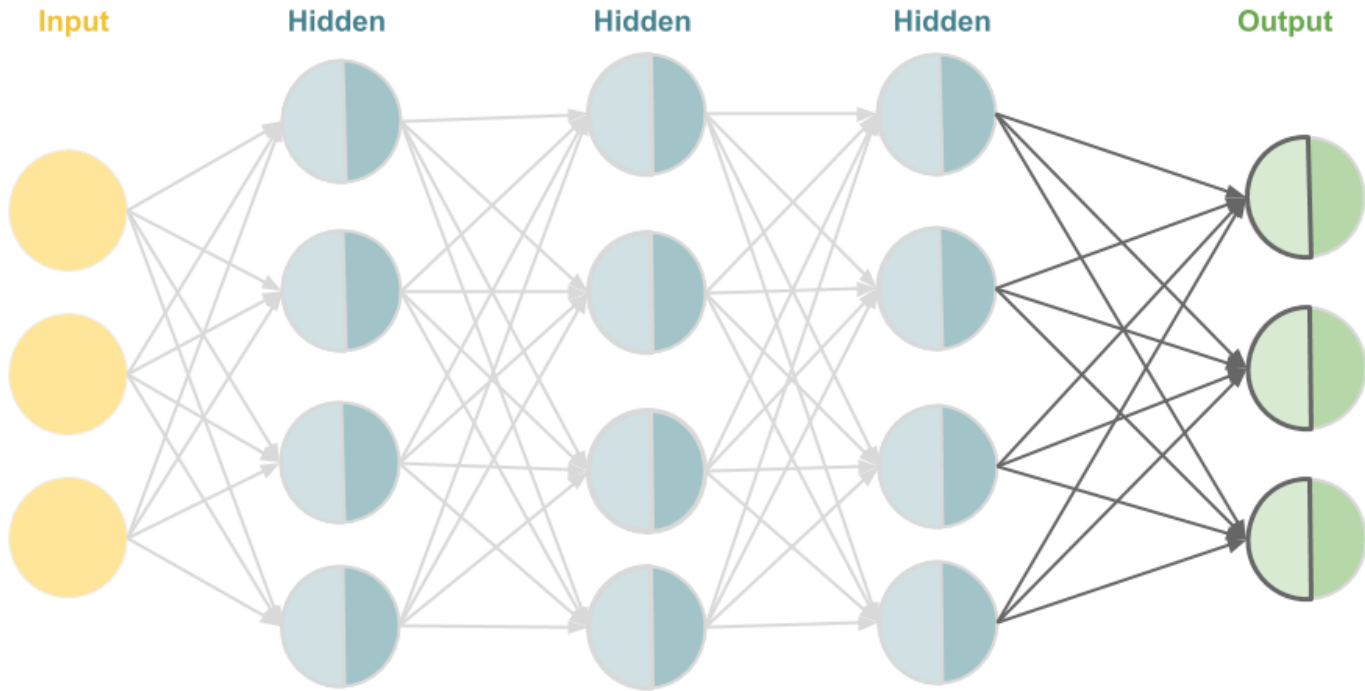
FEEDFORWARD NEURAL NETWORKS: EXAMPLE



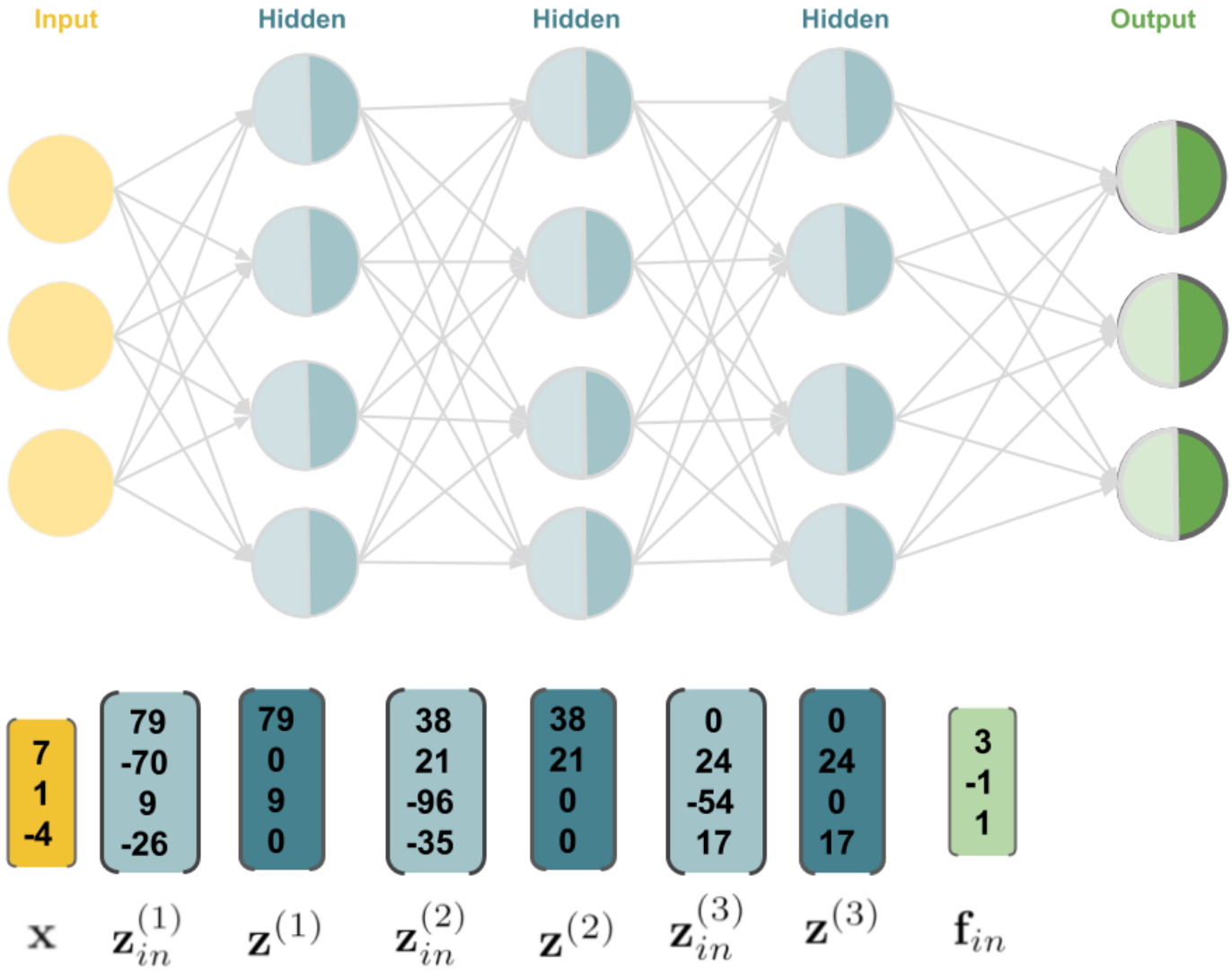
FEEDFORWARD NEURAL NETWORKS: EXAMPLE



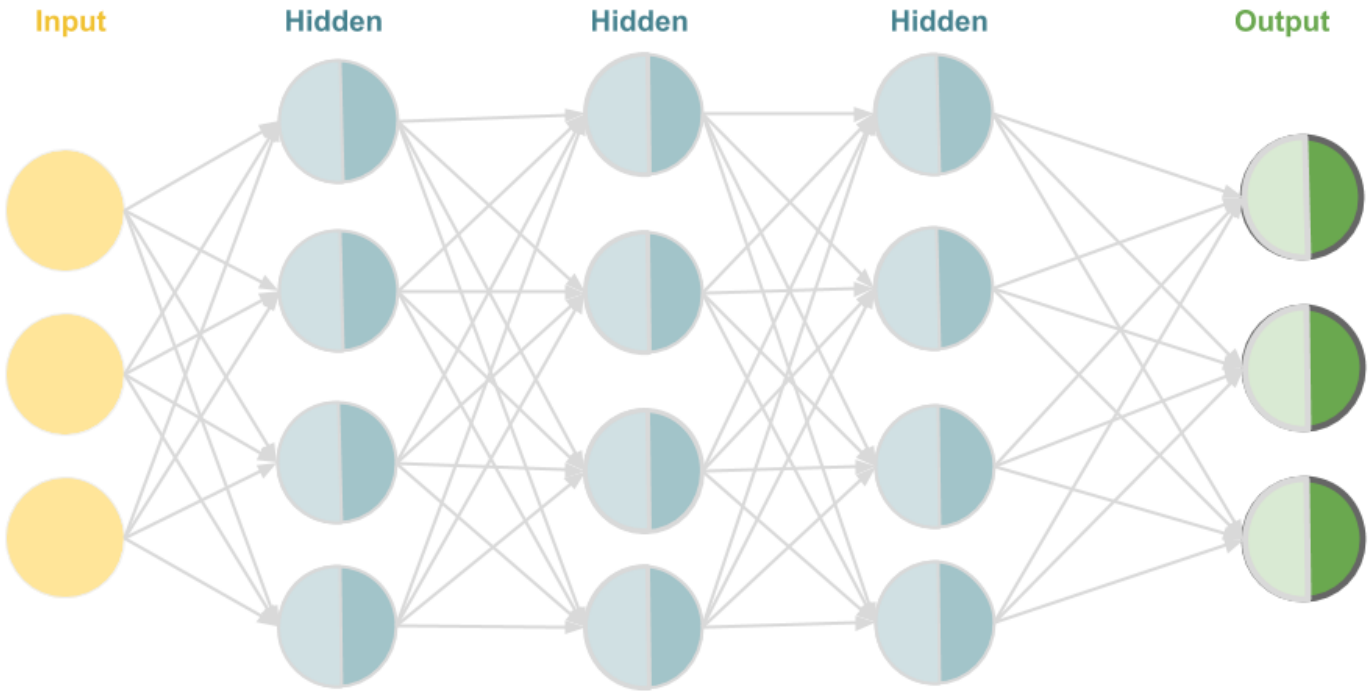
FEEDFORWARD NEURAL NETWORKS: EXAMPLE



FEEDFORWARD NEURAL NETWORKS: EXAMPLE



FEEDFORWARD NEURAL NETWORKS: EXAMPLE



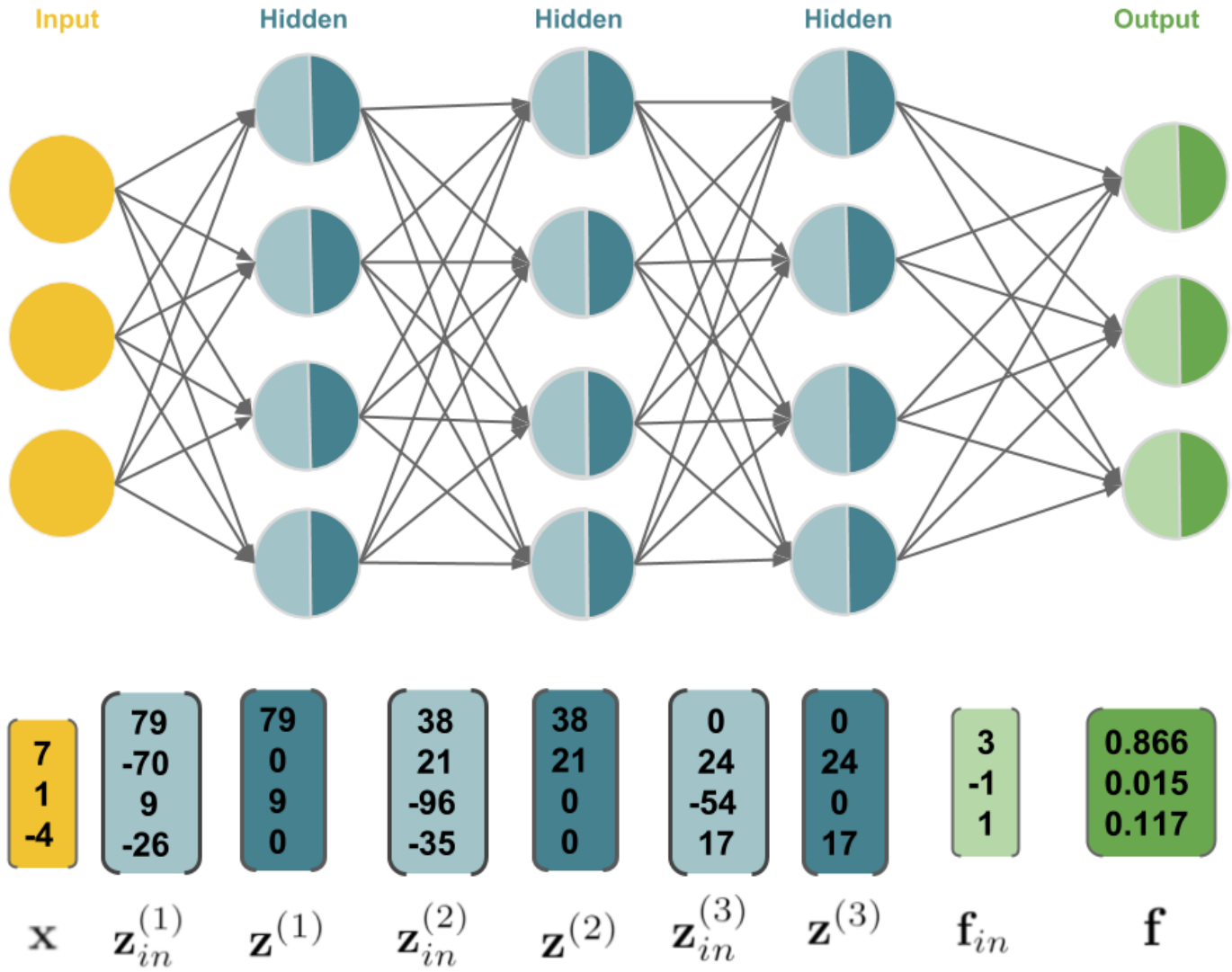
$$\mathbf{f}_{in}$$

3
-1
1

$$\frac{\exp(3)}{\exp 3+\exp(-1)+\exp(1)}$$
$$\frac{\exp(-1)}{\exp 3+\exp(-1)+\exp(1)}$$
$$\frac{\exp(1)}{\exp 3+\exp(-1)+\exp(1)}$$

$$\mathbf{f} = \mathbf{f}_{out} = \tau(\mathbf{f}_{in})$$

FEEDFORWARD NEURAL NETWORKS: EXAMPLE



WHY ADD MORE LAYERS?

- Multiple layers allow for the extraction of more and more abstract representations.
- Each layer in a feed-forward neural network adds its own degree of non-Inearity to the model.

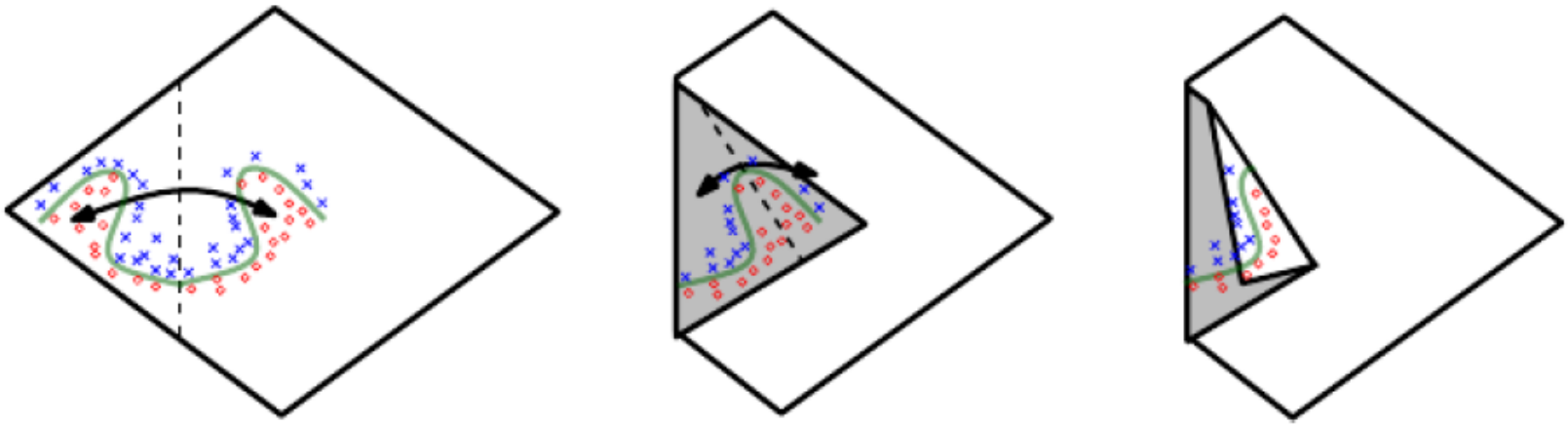


Figure: An intuitive, geometric explanation of the exponential advantage of deeper networks formally (Montúfar et al., 2014).

DEEP NEURAL NETWORKS

Neural networks today can have hundreds of hidden layers. The greater the number of layers, the "deeper" the network. Historically DNNs were very challenging to train and not popular until the late ’00s for several reasons:

- The use of sigmoid activations (e.g., logistic sigmoid and tanh) significantly slowed down training due to a phenomenon known as “vanishing gradients”. The introduction of the ReLU activation largely solved this problem.
- Training DNNs on CPUs was too slow to be practical. Switching over to GPUs cut down training time by more than an order of magnitude.
- When dataset sizes are small, other models (such as SVMs) and techniques (such as feature engineering) often outperform them.

DEEP NEURAL NETWORKS

- The availability of large datasets and novel architectures that are capable of handling even complex tensor-shaped data (e.g. CNNs for image data), faster hardware, and better optimization and regularization methods made it feasible to successfully implement deep neural networks.

- An increase in depth often translates to an increase in performance on a given task. State-of-the-art neural networks, however, are much more sophisticated than the simple architectures we have encountered so far.

The term "**deep learning**" encompasses all of these developments and refers to the field as a whole.

INTRODUCTION TO DEEP LEARNING

TRAINING NEURAL NETWORKS

- In ML we use **empirical risk minimization** to minimize prediction losses over the training data

$$\mathcal{R}_{\text{emp}} = \frac{1}{n} \sum_{i=1}^{n} L\left(y^{(i)}, f\left(\mathbf{x}^{(i)} \mid \boldsymbol{\theta}\right)\right)$$

- In DL, $\boldsymbol{\theta}$ represents the weights (and biases) of the NN.
- Often, L2 in regression:

$$L\left(y, f(\mathbf{x})\right) = \frac{1}{2}(y - f(\mathbf{x}))^2$$

- or cross-entropy for binary classification

$$L\left(y, f(\mathbf{x})\right) = -(y \log f(\mathbf{x}) + (1 - y) \log(1 - f(\mathbf{x})))$$

- ERM can be implemented by **gradient descent**.

GRADIENT DESCENT

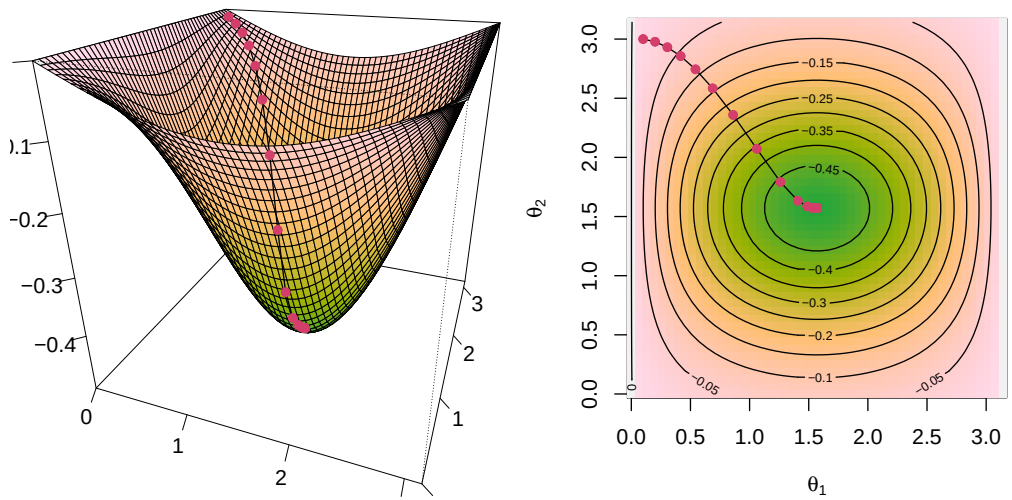
- Neg. risk gradient points in the direction of the **steepest descent**

$$-\mathbf{g} = -\nabla \mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) = -\left(\frac{\partial \mathcal{R}_{\text{emp}}}{\partial \theta_1}, \ldots, \frac{\partial \mathcal{R}_{\text{emp}}}{\partial \theta_d} \right)^\top$$

- "Standing" at a point $\boldsymbol{\theta}^{[t]}$, we locally improve by:

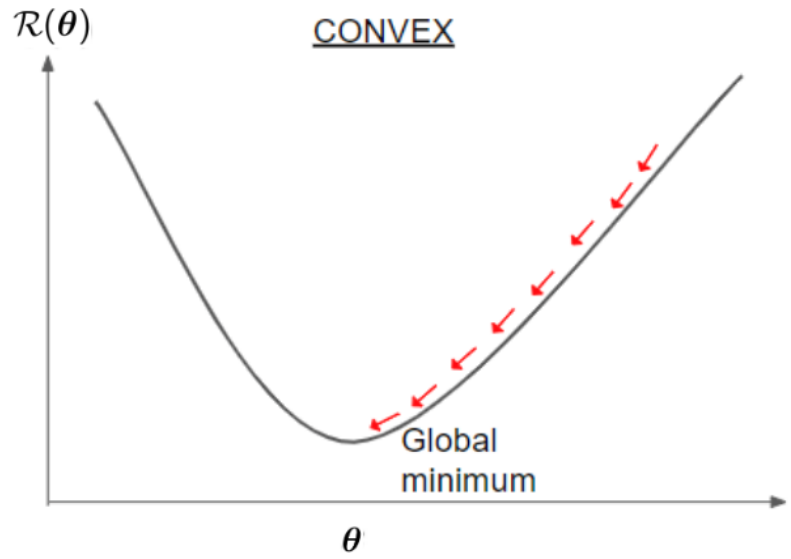
$$\boldsymbol{\theta}^{[t+1]} = \boldsymbol{\theta}^{[t]} - \alpha \mathbf{g},$$

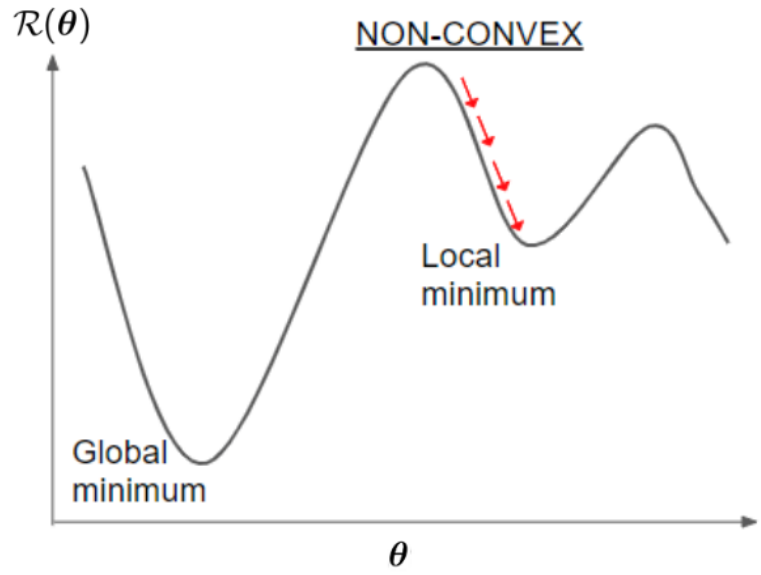
- α is called **step size** or **learning rate**.



GRADIENT DESCENT AND OPTIMALITY

- GD is a greedy algorithm: In every iteration, it makes locally optimal moves.
- If $\mathcal{R}_{\text{emp}}(\boldsymbol{\theta})$ is **convex** and **differentiable**, and its gradient is Lipschitz continuous, GD is guaranteed to converge to the global minimum (for small enough step-size).
- However, if $\mathcal{R}_{\text{emp}}(\boldsymbol{\theta})$ has multiple local optima and/or saddle points, GD might only converge to a stationary point (other than the global optimum), depending on the starting point.





GRADIENT DESCENT AND OPTIMALITY

Note: It might not be that bad if we do not find the global optimum:

- We don't optimize the (theoretical) risk, but only an approximate version, i.e. the empirical risk.
- For very flexible models, aggressive optimization might overfitting.
- Early-stopping might even increase generalization performance.

LEARNING RATE

The step-size α plays a key role in the convergence of the algorithm.

If the step size is too small, the training process may converge **very** slowly (see left image). If the step size is too large, the process may not converge, because it **jumps** around the optimal point (see right image).

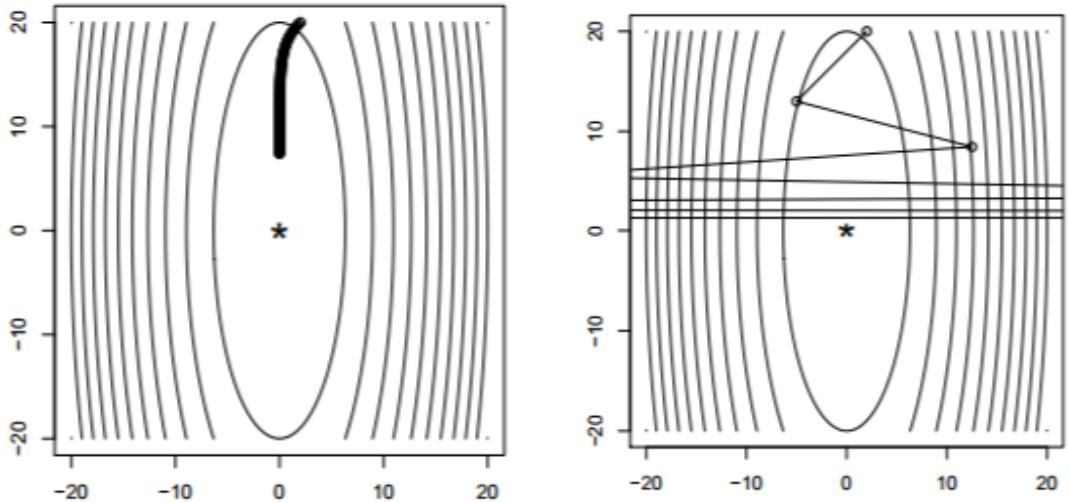


Figure: Small stepsize (left) vs. large stepsize (right) (Tibshirani, 2019)

LEARNING RATE

So far we have assumed a fixed value of α in every iteration:

$$\alpha^{[t]} = \alpha \quad \forall t = \{1, \ldots, T\}$$

However, it makes sense to adapt α in every iteration:

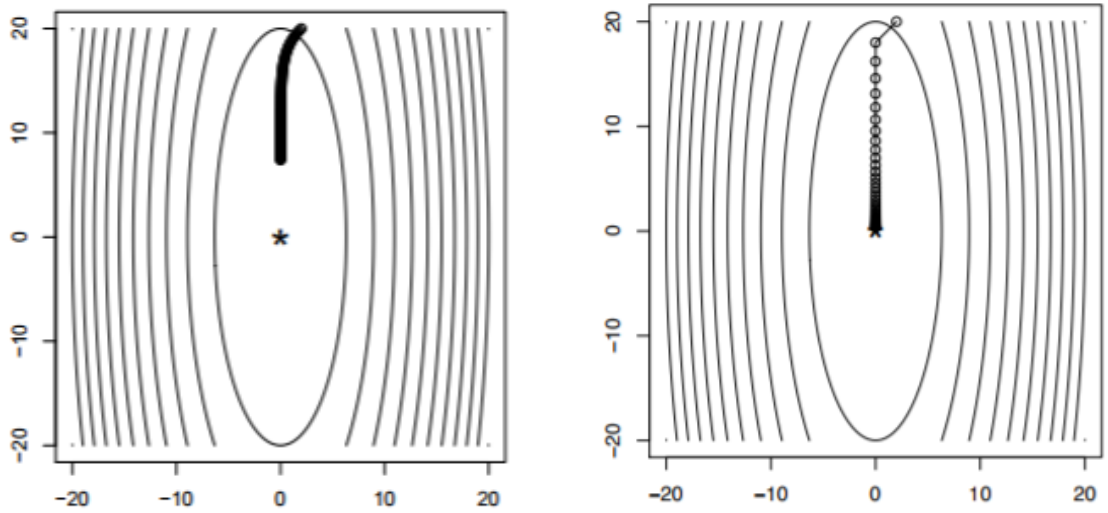


Figure: Steps of gradient descent for $\mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) = 10\,\theta_1^2 + 0.5\,\theta_2^2$. Left: 100 steps for with a fixed learning rate. Right: 40 steps with an adaptive learning rate (Tibshirani, 2019).

WEIGHT INITIALIZATION

- Weights (and biases) of an NN must be initialized in GD.
- We somehow must "break symmetry" – which would happen in full-0-initialization. If two neurons (with the same activation) are connected to the same inputs and have the same initial weights, then both neurons will have the same gradient update and learn the same features.
- Weights are typically drawn from a uniform a Gaussian distribution (both centered at 0 with a small variance).
- Two common initialization strategies are 'Glorot initialization' and 'He initialization' which tune the variance of these distributions based on the topology of the network.

STOCHASTIC GRADIENT DESCENT

GD for ERM was:

$$\boldsymbol{\theta}^{[t+1]} = \boldsymbol{\theta}^{[t]} - \alpha \cdot \frac{1}{n} \cdot \sum_{i=1}^{n} \nabla_\theta L \left(y^{(i)}, f(\mathbf{x}^{(i)} \mid \boldsymbol{\theta}^{[t]}) \right)$$

- Using the entire training set in GD to is called **batch** or **deterministic** or **offline training**. This can be computationally costly or impossible, if data does not fit into memory.
- **Idea:** Instead of letting the sum run over the whole dataset, use small stochastic subsets (**minibatches**), or only a single $\mathbf{x}^{(i)}$.
- If batches are uniformly sampled from $\mathcal{D}_{\text{train}}$, our stochastic gradient is in expectation the batch gradient $\nabla_\theta \mathcal{R}_{\text{emp}}(\boldsymbol{\theta})$.
- The gradient w.r.t. a single $\mathbf{x}$ is fast to compute but not reliable. But its often used in a theoretical analysis of SGD.

$\rightarrow$ We have a **stochastic**, noisy version of the batch GD.

STOCHASTIC GRADIENT DESCENT

SGD on function $1.25(x_1 + 6)^2 + (x_2 - 8)^2$.

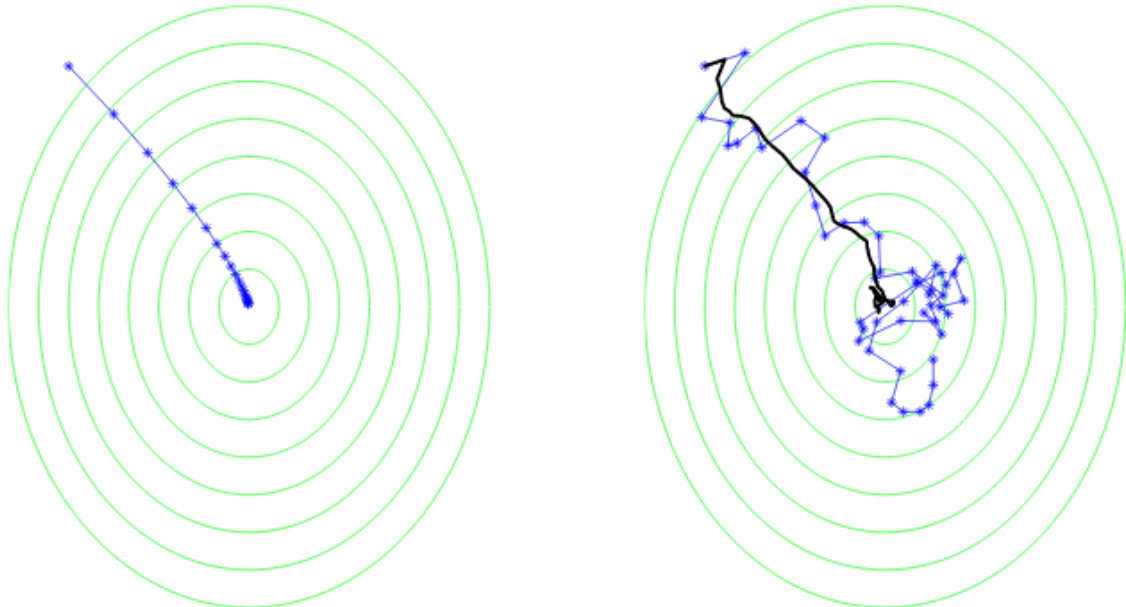


Figure: Left = GD, right = SGD. The black line is a smoothed θ (Shalev-Shwartz & Ben-David, 2022)

STOCHASTIC GRADIENT DESCENT

Algorithm 1 Basic SGD pseudo code

1: Initialize parameter vector $\boldsymbol{\theta}^{[0]}$
2: $t \leftarrow 0$
3: **while** stopping criterion not met **do**
4: Randomly shuffle data and partition into minibatches $J_1, ..., J_K$ of size m
5: **for** $k \in \{1, ..., K\}$ **do**
6: $t \leftarrow t + 1$
7: Compute gradient estimate with J_k: $\hat{g}^{[t]} \leftarrow \frac{1}{m} \sum_{i \in J_k} \nabla_\theta L(y^{(i)}, f(\mathbf{x}^{(i)} \mid \boldsymbol{\theta}^{[t-1]}))$
8: Apply update: $\boldsymbol{\theta}^{[t]} \leftarrow \boldsymbol{\theta}^{[t-1]} - \alpha \hat{g}^{[t]}$
9: **end for**
10: **end while**

- A full SGD pass over data is an **epoch**.
- Minibatch sizes are typically between 50 and 1000.

STOCHASTIC GRADIENT DESCENT

- SGD is the most used optimizer in ML and especially in DL.
- We usually have to add a considerable amount of tricks to SGD to make it really efficient (e.g. momentum). More on this later.
- SGD with (small) batches has a high variance, although is unbiased. Hence, the LR α is smaller than in the batch mode.
- When LR is slowly decreased, SGD converges to local minimum.
- Recent results indicate that SGD often leads to better generalization than GD, and may result in indirect regularization.

INTRODUCTION TO DEEP LEARNING

WHAT IS REGULARIZATION?

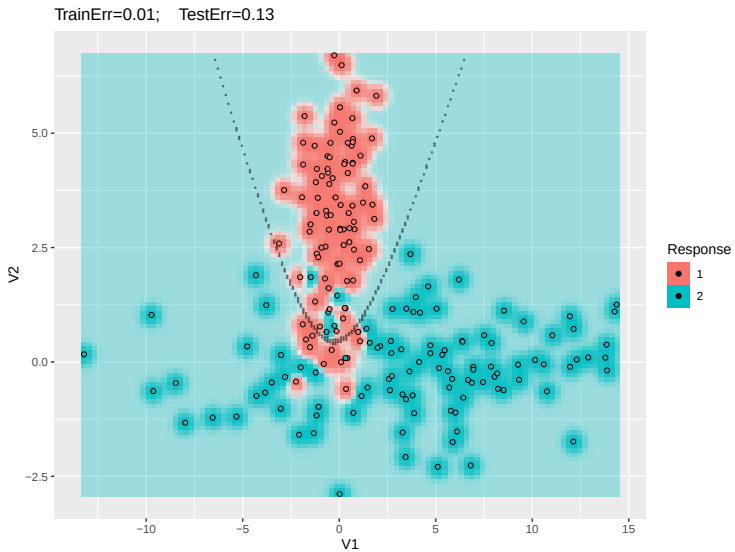
Methods that add **inductive bias** to model, usually some "low complexity" priors (shrinkage and sparsity) to reduce overfitting and get better bias-variance tradeoff

- **Explicit regularization**: penalize explicit measure of model complexity in ERM (e.g., $L1/L2$)
- **Implicit regularization**: early stopping, data augmentation, parameter sharing, dropout or ensembling
- **Structured regularization**: structural prior knowledge over groups of parameters or subnetworks (e.g., group lasso ▸ Yuan and Lin 2006)

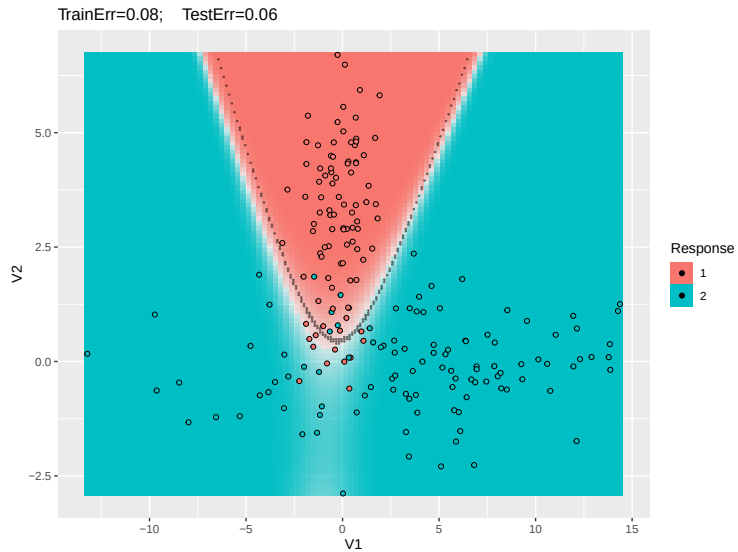
RECAP: OVERFITTING

- Occurs when model reflects noise or artifacts in training data
- Model often then does not generalize well (small train error, high test error) – or at least works better on train than on test data

Overfitted model

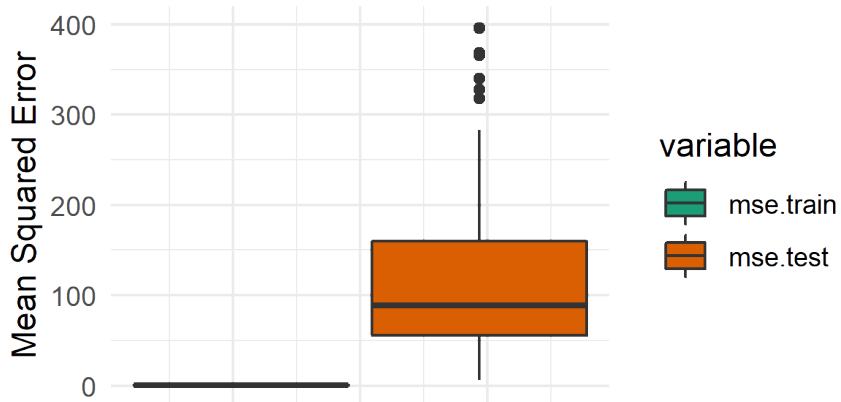


Appropriate model



EXAMPLE I: OVERFITTING

- Data set: daily maximum **ozone level** in LA; $n = 50$
- 12 features: time (weekday, month); weather (temperature at stations, humidity, wind speed); pressure gradient
- Orig. data was subsetted, so it feels "high-dim." now (low n in relation to p)
- LM with all features (L2 loss)
- MSE evaluation under 10×10 REP-CV



Model fits train data well, but generalizes poorly.

EXAMPLE II: OVERFITTING

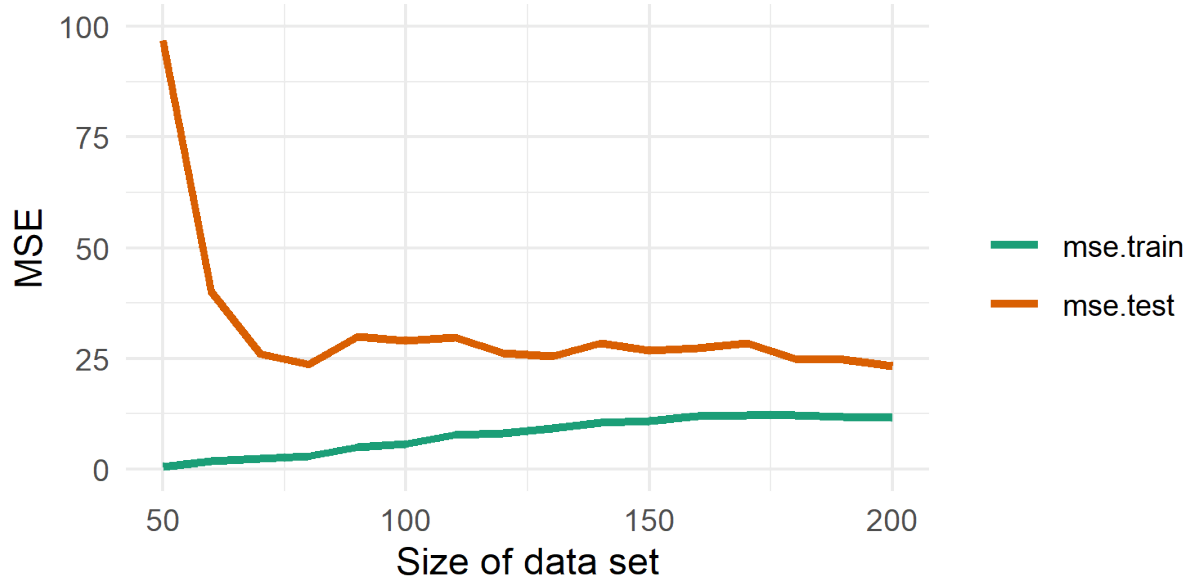
- We train an MLP and a CART on the mtcars data
- Both models are not regularized
- And configured to make overfitting more likely

	Train MSE	Test MSE
Neural Network	3.68	19.98
CART	0.00	10.21

(And we now switch back to the Ozone example...)

AVOIDING OVERFITTING – COLLECT MORE DATA

We explore our results for increased dataset size.

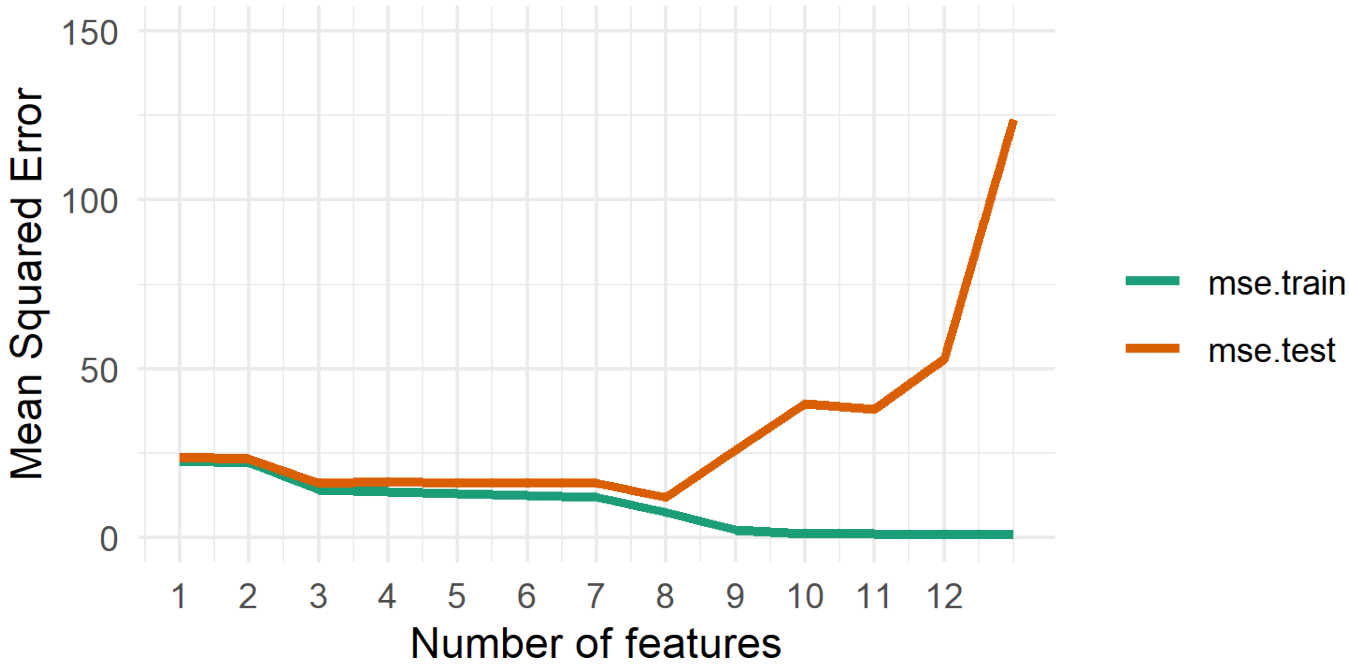


Fit slightly worsens, but test error decreases.
But: Often not feasible in practice.

AVOIDING OVERFITTING – REDUCE COMPLEXITY

We try the simplest model: a constant. So for *L2* loss the mean of $y^{(i)}$.

We then increase complexity by adding one feature at a time.



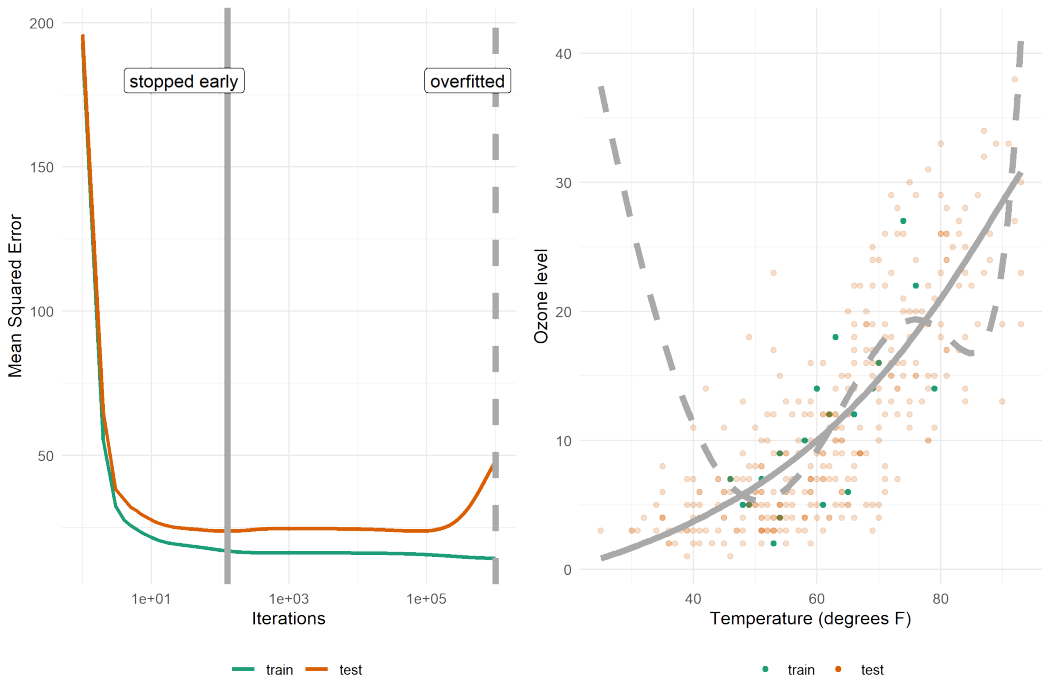
NB: We added features in a specific (clever) order, so we cheated a bit.

AVOIDING OVERFITTING – OPTIMIZE LESS

Now: polynomial regression with temperature as single feature

$$f(\mathbf{x} \mid \boldsymbol{\theta}) = \sum_{k=0}^{d} \theta_k \cdot (x_T)^k$$

We set $d = 15$ to overfit to small data. To investigate early stopping, we don't analytically solve the OLS problem, but run GD stepwise.



We see: Early stopping GD can improve results.

NB: GD for poly-regr usually needs many iters before it starts to overfit, so we used a very small training set.

REGULARIZED EMPIRICAL RISK MINIMIZATION

We have contradictory goals:

- **maximizing fit** (minimizing the train loss)
- **minimizing complexity** of the model

We saw how we can include features in a binary fashion.
But we would rather control complexity **on a continuum**.

REGULARIZED EMPIRICAL RISK MINIMIZATION

Common pattern:

$$\mathcal{R}_{\text{reg}}(f) = \mathcal{R}_{\text{emp}}(f) + \lambda \cdot J(f) = \sum_{i=1}^{n} L\left(y^{(i)}, f\left(\mathbf{x}^{(i)}\right)\right) + \lambda \cdot J(f)$$

- $J(f)$: **complexity penalty**, **roughness penalty** or **regularizer**
- $\lambda \geq 0$: **complexity control** parameter
- The higher λ, the more we penalize complexity
- $\lambda = 0$: We just do simple ERM; $\lambda \to \infty$: we don't care about loss, models become as "simple" as possible
- λ is hard to set manually and is usually selected via CV
- As for $\mathcal{R}_{\text{emp}}$, $\mathcal{R}_{\text{reg}}$ and J are often defined in terms of $\boldsymbol{\theta}$:

$$\mathcal{R}_{\text{reg}}(\boldsymbol{\theta}) = \mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) + \lambda \cdot J(\boldsymbol{\theta})$$

REGULARIZATION IN LM

- Can also overfit if p large and n small(er)
- OLS estimator requires full-rank design matrix
- For highly correlated features, OLS becomes sensitive to random errors in response, results in large variance in fit
- We now add a complexity penalty to the loss:

$$\mathcal{R}_{\text{reg}}(\boldsymbol{\theta}) = \sum_{i=1}^{n} \left(y^{(i)} - \boldsymbol{\theta}^\top \mathbf{x}^{(i)} \right)^2 + \lambda \cdot J(\boldsymbol{\theta}).$$

RIDGE REGRESSION / L2 PENALTY

Intuitive measure of model complexity is deviation from 0-origin; coeffs then have no or a weak effect. So we measure $J(\boldsymbol{\theta})$ through a vector norm, shrinking coeffs closer to 0.

$$\begin{aligned} \hat{\boldsymbol{\theta}}_{\text{ridge}} &= \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^{n} \left(y^{(i)} - \boldsymbol{\theta}^T \mathbf{x}^{(i)} \right)^2 + \lambda \sum_{j=1}^{p} \theta_j^2 \\ &= \arg\min_{\boldsymbol{\theta}} \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|_2^2 + \lambda \|\boldsymbol{\theta}\|_2^2 \end{aligned}$$

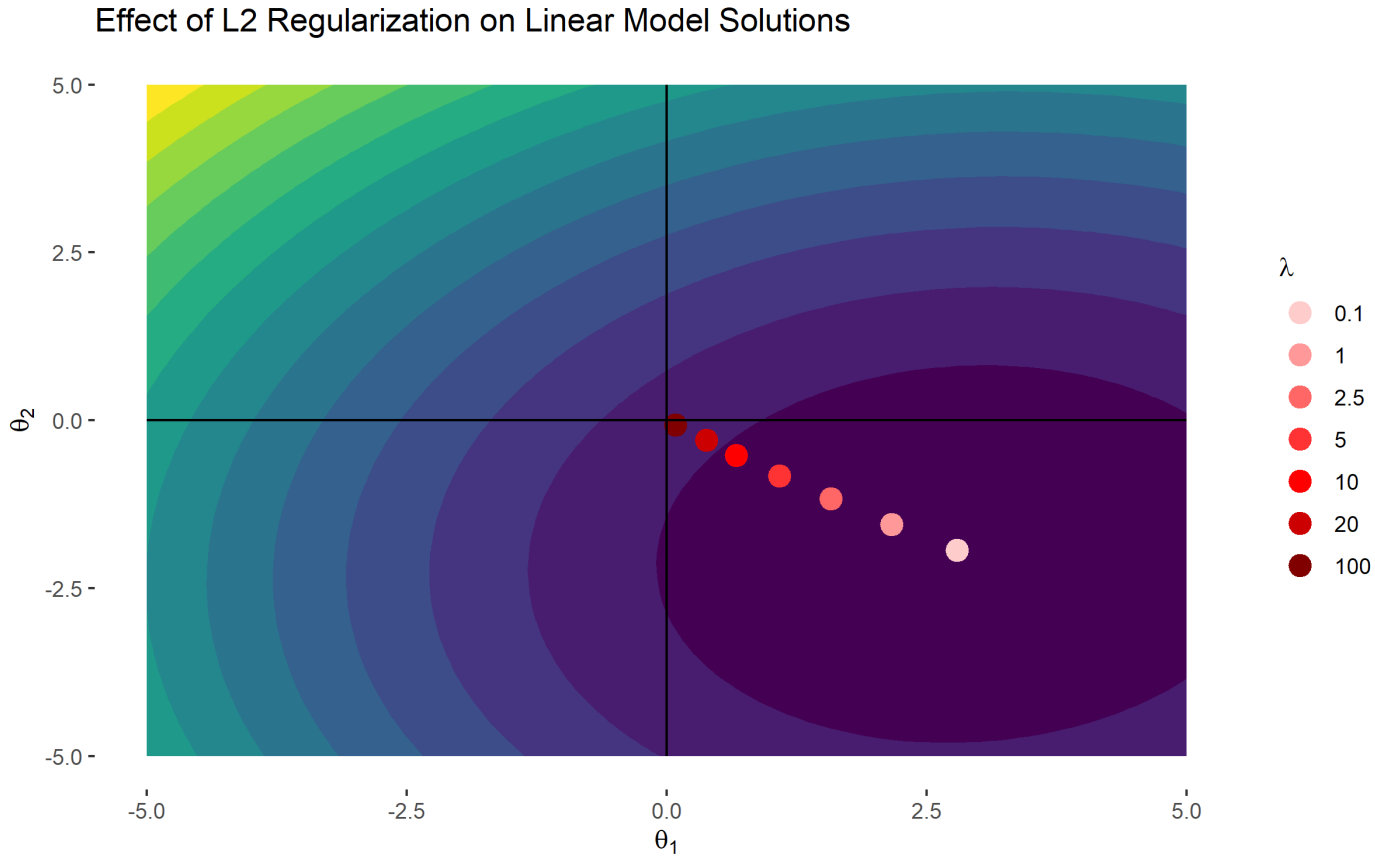
Can still analytically solve this:

$$\hat{\boldsymbol{\theta}}_{\text{ridge}} = (\mathbf{X}^T\mathbf{X} + \lambda \boldsymbol{I})^{-1}\mathbf{X}^T\mathbf{y}$$

Name: We add pos. entries along the diagonal "ridge" of $\mathbf{X}^T\mathbf{X}$

RIDGE REGRESSION / L2 PENALTY

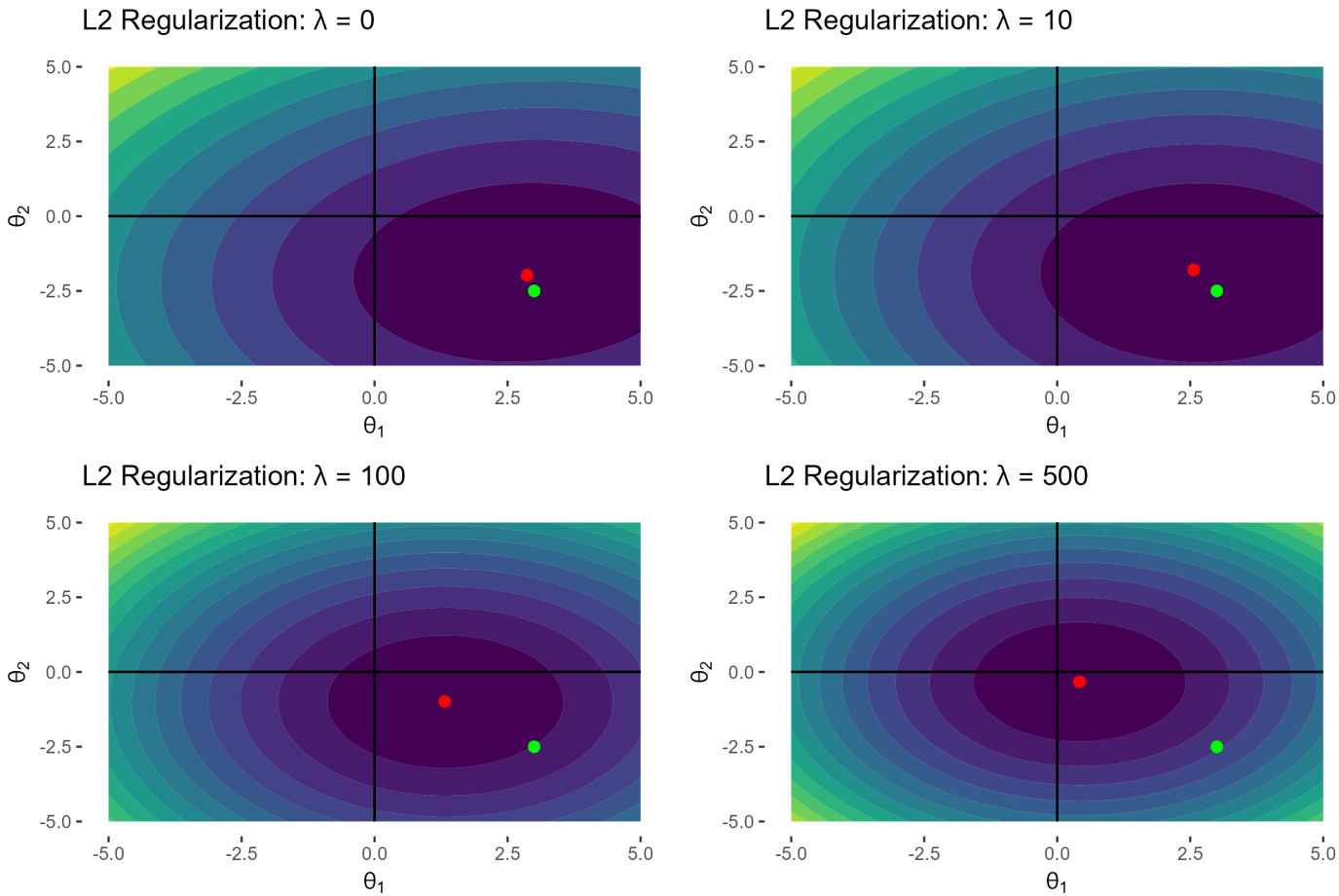
Let $y = 3x_1 - 2x_2 + \epsilon$, $\epsilon \sim N(0, 1)$. The true minimizer is $\theta^* = (3, -2)^T$, with $\hat{\boldsymbol{\theta}}_{\text{ridge}} = \arg\min_{\boldsymbol{\theta}} \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|^2 + \lambda\|\boldsymbol{\theta}\|^2$.



With increasing regularization, $\hat{\theta}_{ridge}$ is pulled back to the origin (contour lines show unregularized objective).

RIDGE REGRESSION / L2 PENALTY

Contours of regularized objective for different λ values.
$\hat{\boldsymbol{\theta}}_{\text{ridge}} = \arg\min_{\boldsymbol{\theta}} \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|^2 + \lambda\|\boldsymbol{\theta}\|^2$.

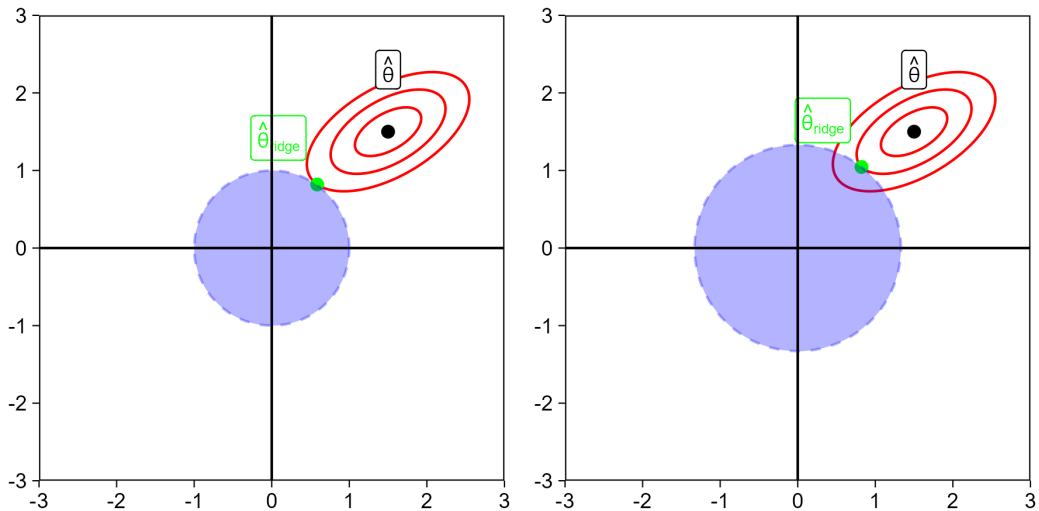


Green = true coefs of the DGP and red = ridge solution.

RIDGE REGRESSION / L2 PENALTY

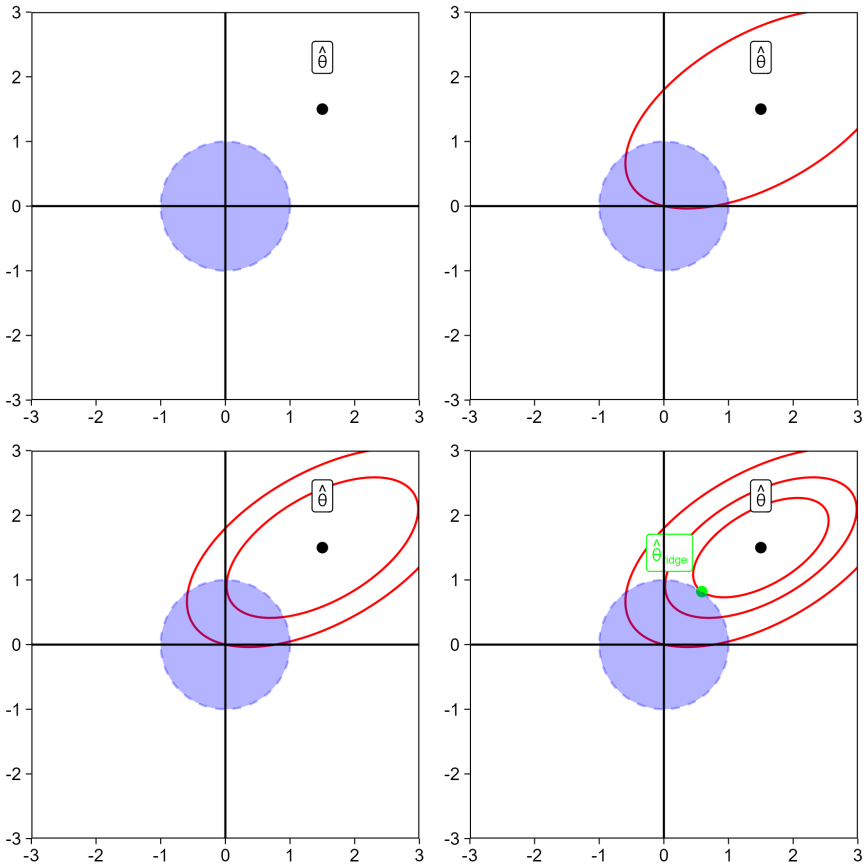
We understand the geometry of these 2 mixed components in our regularized risk objective much better, if we formulate the optimization as a constrained problem (see this as Lagrange multipliers in reverse).

$$\begin{aligned} \min_{\boldsymbol{\theta}} \quad & \sum_{i=1}^{n} \left(y^{(i)} - f\left(\mathbf{x}^{(i)} \mid \boldsymbol{\theta} \right) \right)^2 \\ \text{s.t.} \quad & \|\boldsymbol{\theta}\|_2^2 \leq t \end{aligned}$$



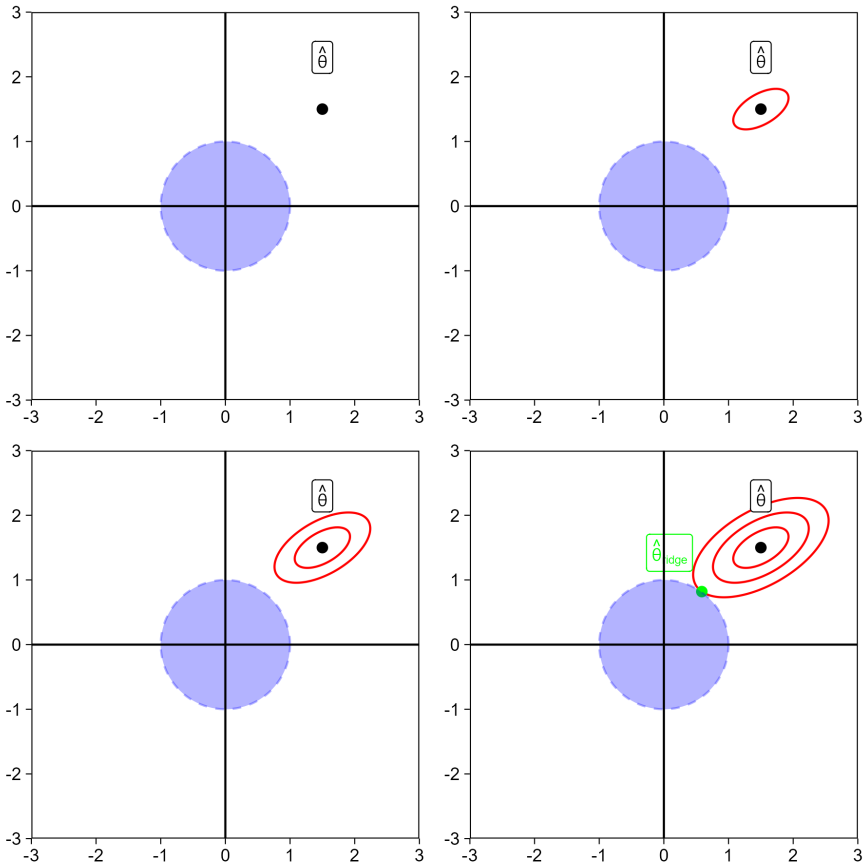
NB: There is a bijective relationship between λ and t: $\lambda \uparrow \Rightarrow \; t \downarrow$ and vice versa.

RIDGE REGRESSION / L2 PENALTY



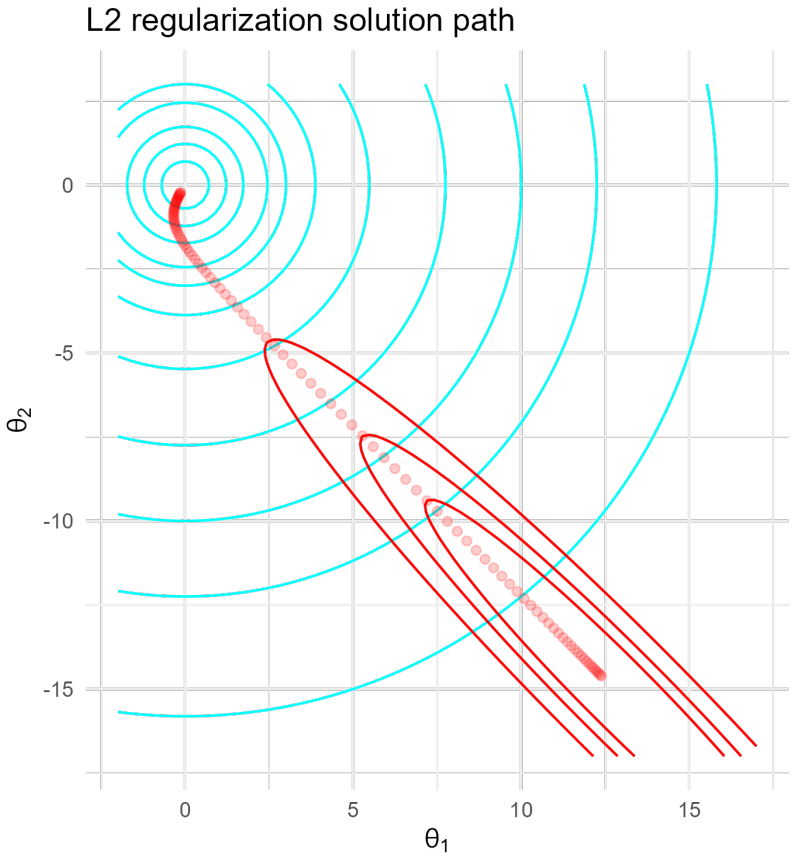
- Inside constraints perspective: From origin, jump from contour line to contour line (better) until you become infeasible, stop before.
- We still optimize the $\mathcal{R}_{\text{emp}}(\boldsymbol{\theta})$, but cannot leave a ball around the origin.
- $\mathcal{R}_{\text{emp}}(\boldsymbol{\theta})$ grows monotonically if we move away from $\hat{\boldsymbol{\theta}}$ (elliptic contours).
- Solution path moves from origin to border of feasible region with minimal L_2 distance.

RIDGE REGRESSION / L2 PENALTY



- Outside constraints perspective: From $\hat{\boldsymbol{\theta}}$, jump from contour line to contour line (worse) until you become feasible, stop then.
- So our new optimum will lie on the boundary of that ball.
- Solution path moves from unregularized estimate to feasible region of regularized objective with minimal L_2 distance.

RIDGE REGRESSION / L2 PENALTY



- Here we can see entire solution path for ridge regression
- Cyan contours indicate feasible regions induced by different λs
- Red contour lines indicate different levels of the unreg. objective
- Ridge solution (red points) gets pulled toward origin for increasing λ

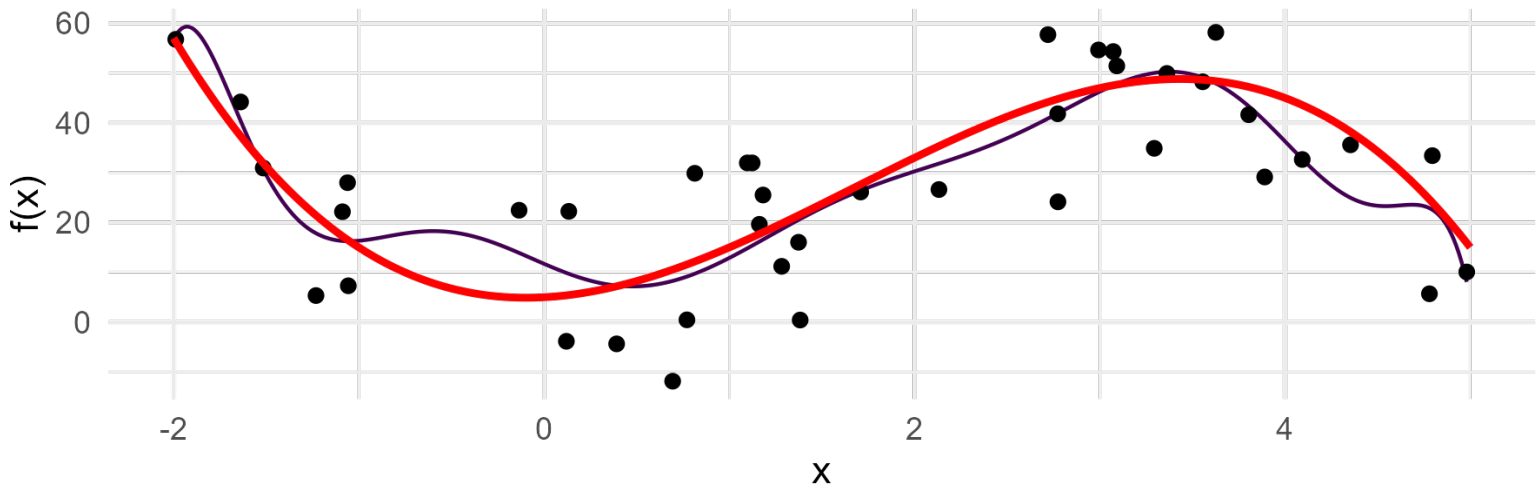
EXAMPLE: POLYNOMIAL RIDGE REGRESSION

Consider $y = f(x) + \epsilon$ where the true (unknown) function is $f(x) = 5 + 2x + 10x^2 - 2x^3$ (in red).

Let's use a dth-order polynomial

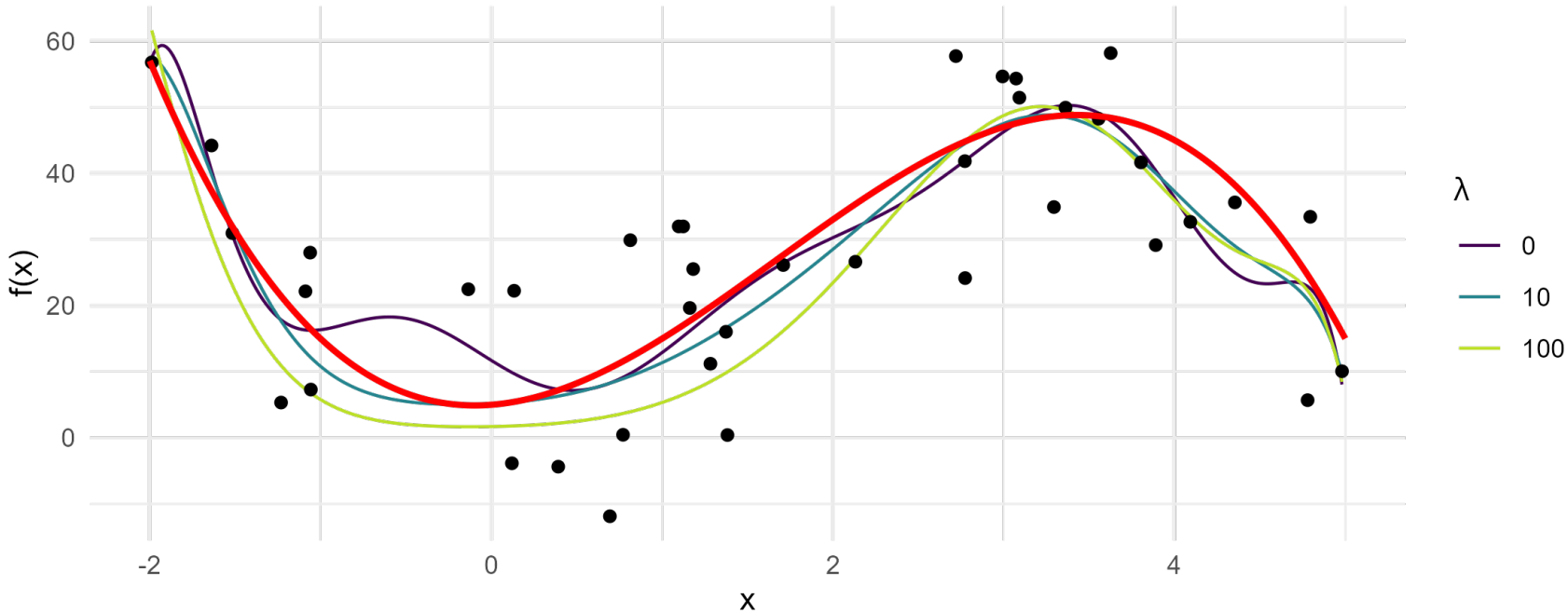
$$f(x) = \theta_0 + \theta_1 x + \cdots + \theta_d x^d = \sum_{j=0}^{d} \theta_j x^j.$$

Using model complexity $d = 10$ overfits:



EXAMPLE: POLYNOMIAL RIDGE REGRESSION

With an *L2* penalty we can now select *d* "too large" but regularize our model by shrinking its coefficients. Otherwise we have to optimize over the discrete *d*.



λ	θ_0	θ_1	θ_2	θ_3	θ_4	θ_5	θ_6	θ_7	θ_8	θ_9	θ_{10}
0.00	12.00	-16.00	4.80	23.00	-5.40	-9.30	4.20	0.53	-0.63	0.13	-0.01
10.00	5.20	1.30	3.70	0.69	1.90	-2.00	0.47	0.20	-0.14	0.03	-0.00
100.00	1.70	0.46	1.80	0.25	1.80	-0.94	0.34	-0.01	-0.06	0.02	-0.00

INTRODUCTION TO DEEP LEARNING

SUMMARY: REGULARIZED RISK MINIMIZATION

If we define (supervised) ML in one line, this might be it:

$$\min_{\boldsymbol{\theta}} \mathcal{R}_{\text{reg}}(\boldsymbol{\theta}) = \min_{\boldsymbol{\theta}} \left(\sum_{i=1}^{n} L\left(y^{(i)}, f\left(\mathbf{x}^{(i)} \mid \boldsymbol{\theta}\right)\right) + \lambda \cdot J(\boldsymbol{\theta}) \right)$$

Can choose for task at hand:

- **hypothesis space** of f, controls how features influence prediction
- **loss** function L, measures how errors are treated
- **regularizer** $J(\boldsymbol{\theta})$, encodes inductive bias

By varying these choices one can construct a huge number of different ML models. Many ML models follow this construction principle or can be interpreted through the lens of RRM.

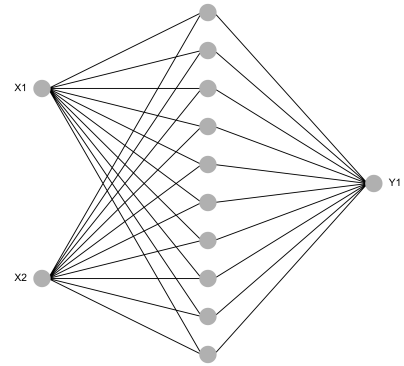
REGULARIZATION IN NONLINEAR MODELS

- So far we have mainly considered regularization in LMs
- Can in general also be applied to to non-linear models; vector-norm penalties require numeric params
- Here, we typically use *L2* regularization, which still results in parameter shrinkage and weight decay
- For non-linear models, regularization is even more important / basically required to prevent overfitting
- Commonplace in methods such as NNs, SVMs, or boosting
- Prediction surfaces / decision boundaries become smoother

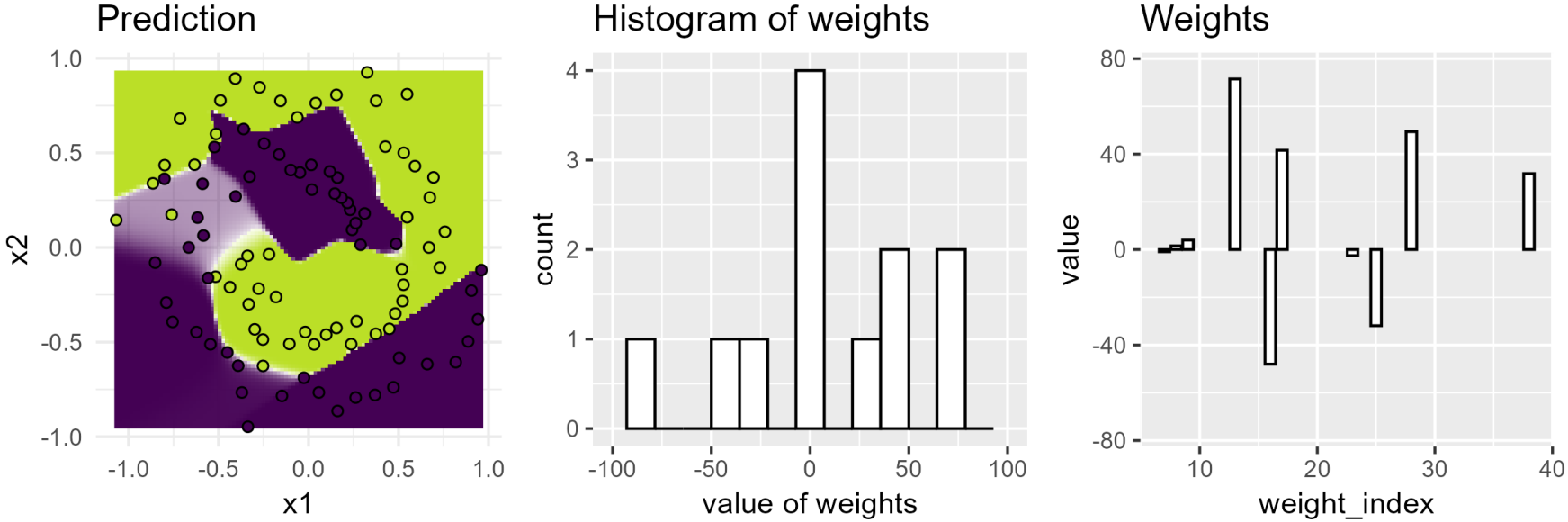
REGULARIZATION IN NONLINEAR MODELS

Classification for `spirals` data.

NN with single hidden layer, size 10, *L2* penalty:



$\lambda = 0$

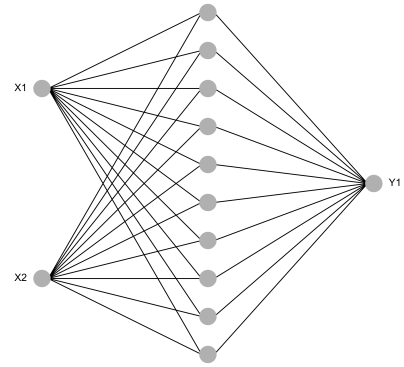


λ affects smoothness of decision boundary and magnitude of weights

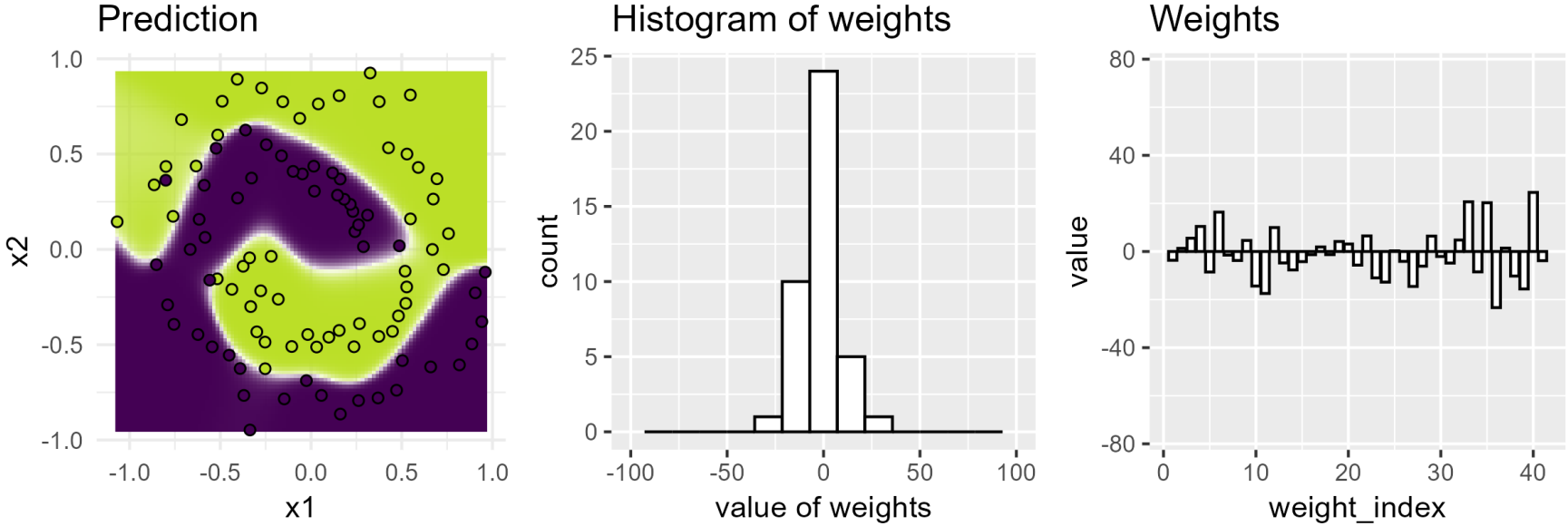
REGULARIZATION IN NONLINEAR MODELS

Classification for `spirals` data.

NN with single hidden layer, size 10, *L2* penalty:



λ = 0.001

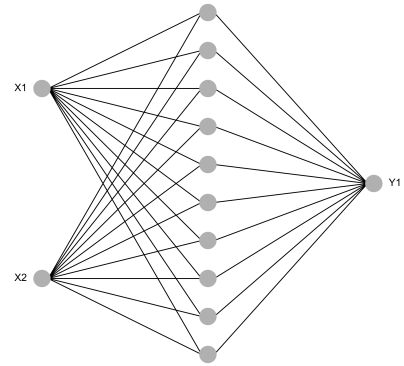


λ affects smoothness of decision boundary and magnitude of weights

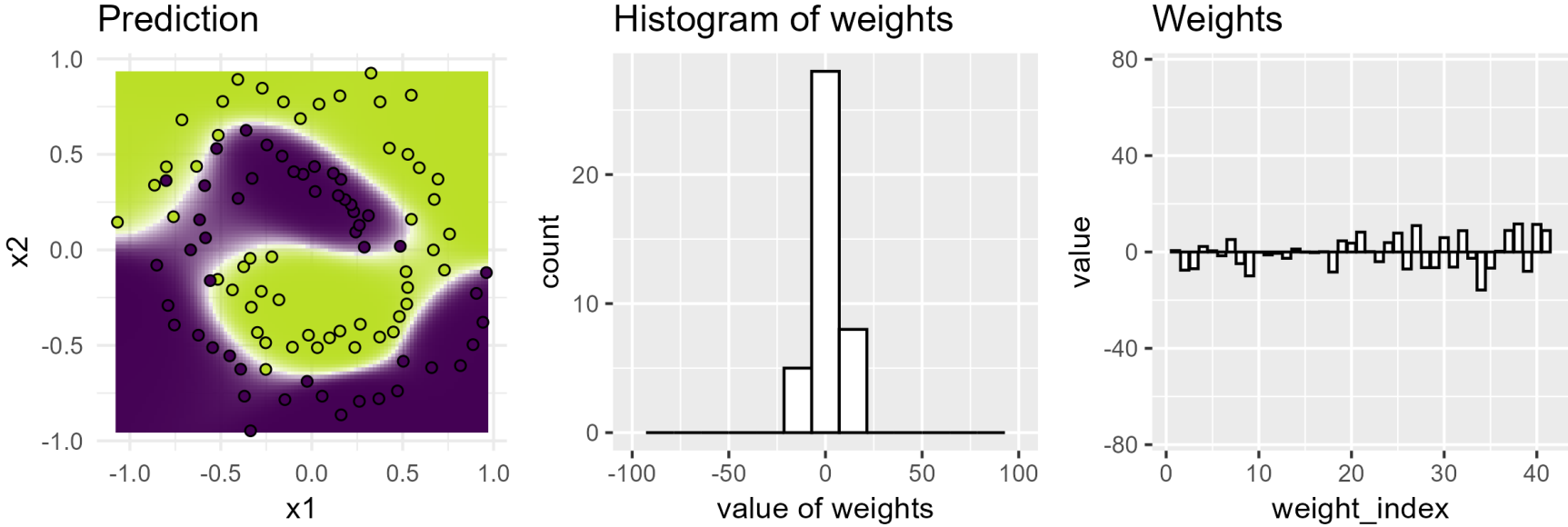
REGULARIZATION IN NONLINEAR MODELS

Classification for `spirals` data.

NN with single hidden layer, size 10, *L2* penalty:



$\lambda = 0.005$

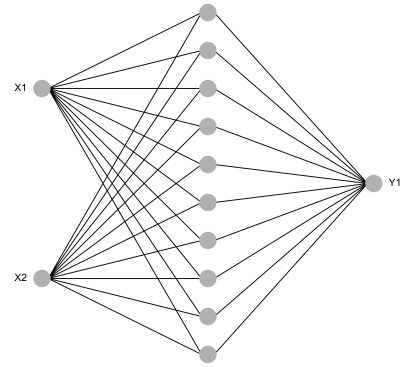


λ affects smoothness of decision boundary and magnitude of weights

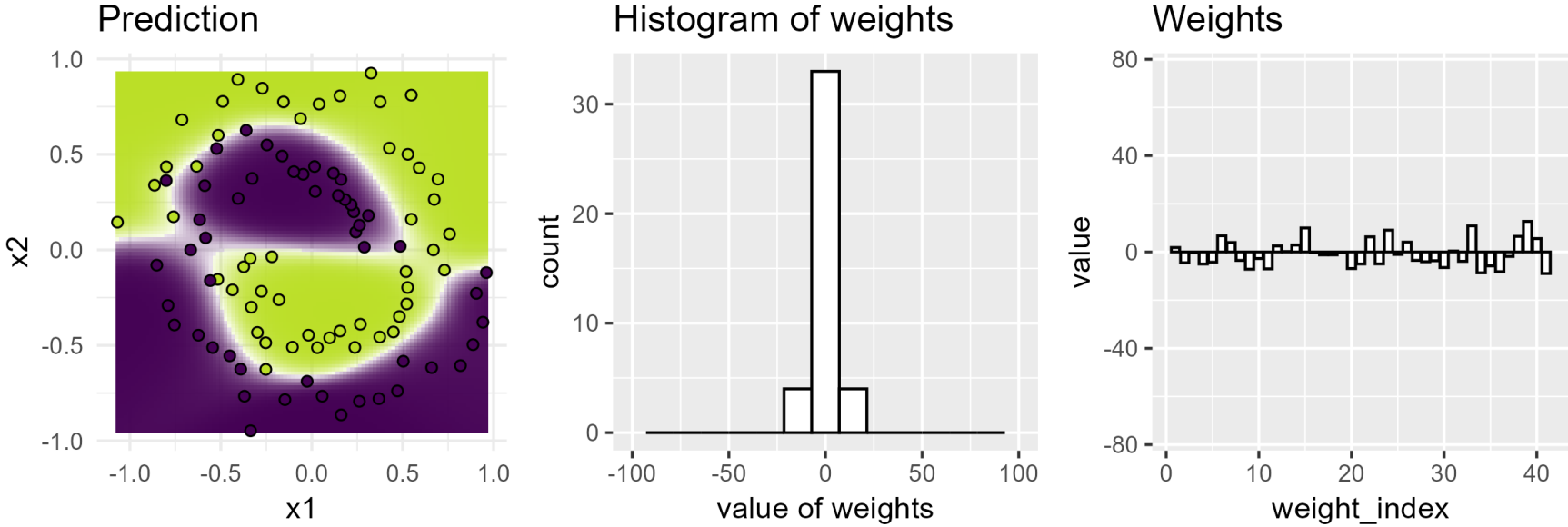
REGULARIZATION IN NONLINEAR MODELS

Classification for `spirals` data.

NN with single hidden layer, size 10, *L2* penalty:



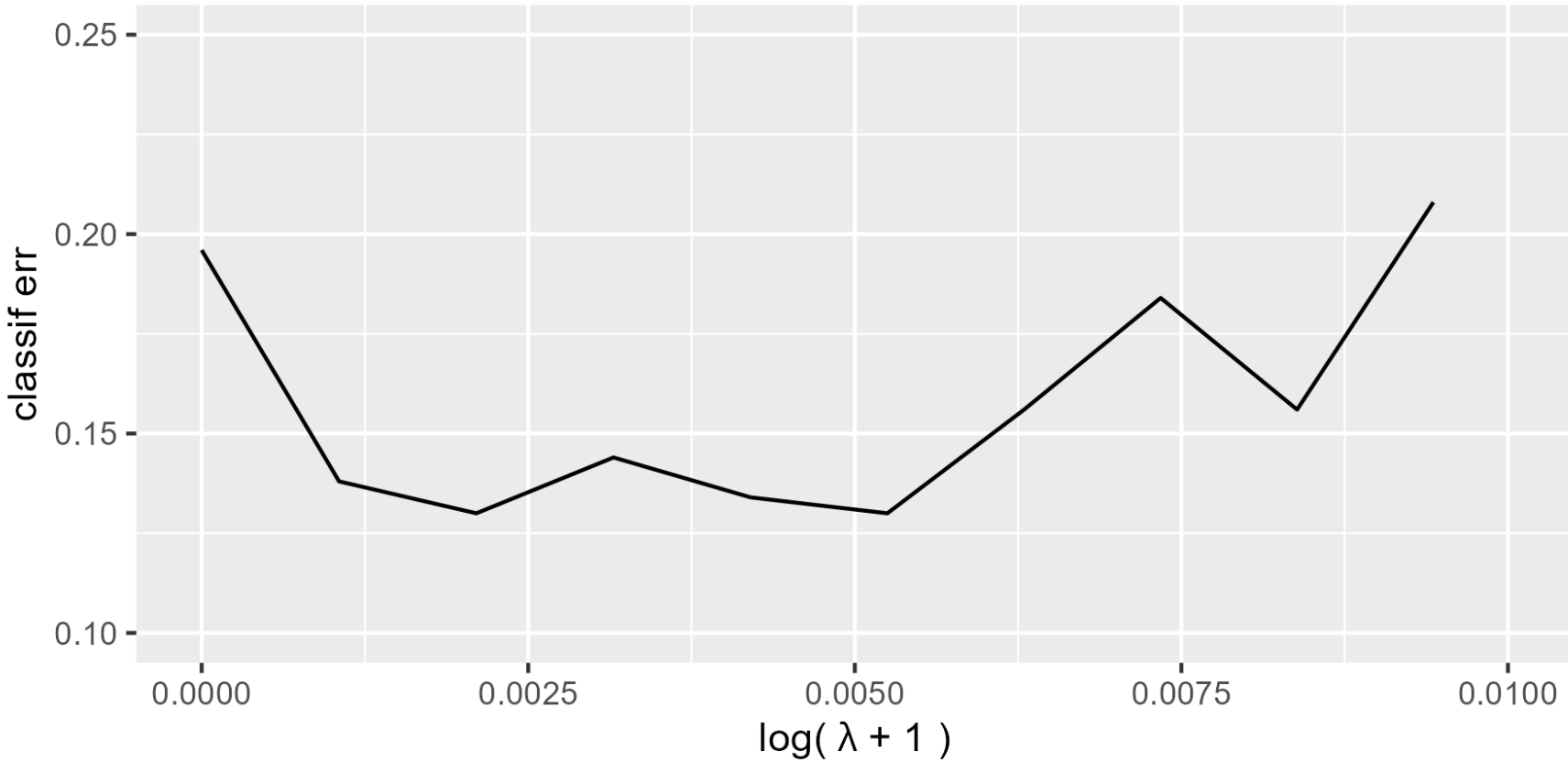
$\lambda = 0.01$



λ affects smoothness of decision boundary and magnitude of weights

REGULARIZATION IN NONLINEAR MODELS

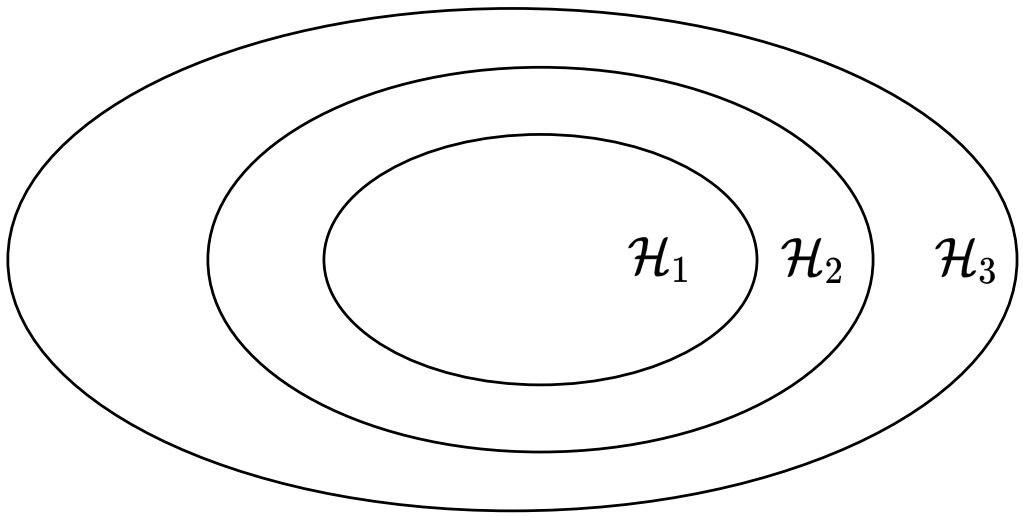
Prevention of overfitting can also be seen in CV.
Same settings as before, but each λ is evaluated with 5x10 REP-CV



Typical U-shape with sweet spot between overfitting and underfitting

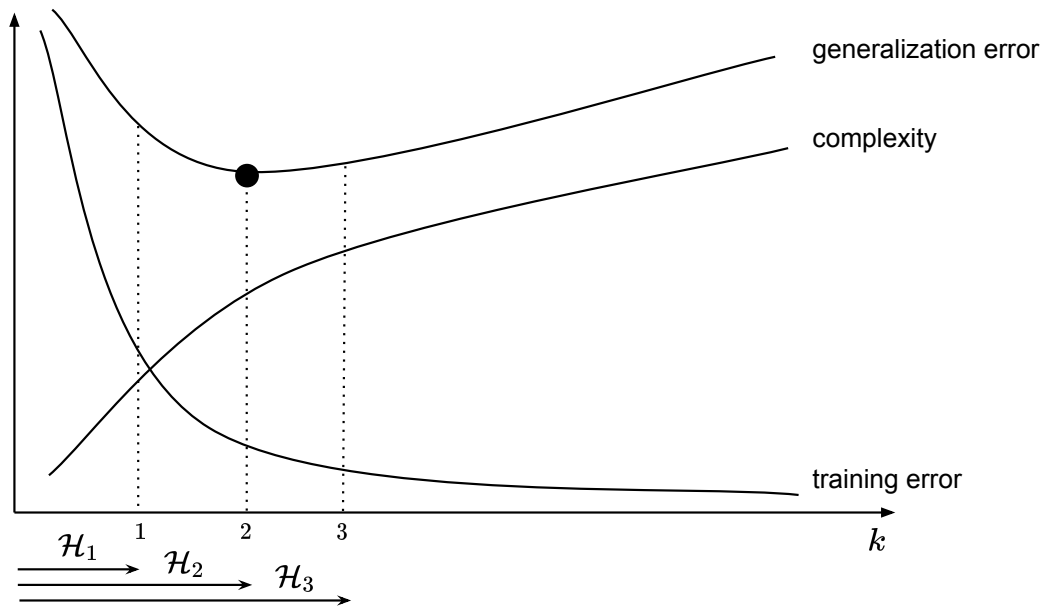
STRUCTURAL RISK MINIMIZATION

- Can also see this as an iterative process;
  more a “discrete” view on things
- SRM assumes that $\mathcal{H}$ can be decomposed into increasingly complex hypotheses: $\mathcal{H} = \cup_{k \geq 1} \mathcal{H}_k$
- Complexity parameters can be, e.g. the degree of polynomials in linear models or the size of hidden layers in neural networks



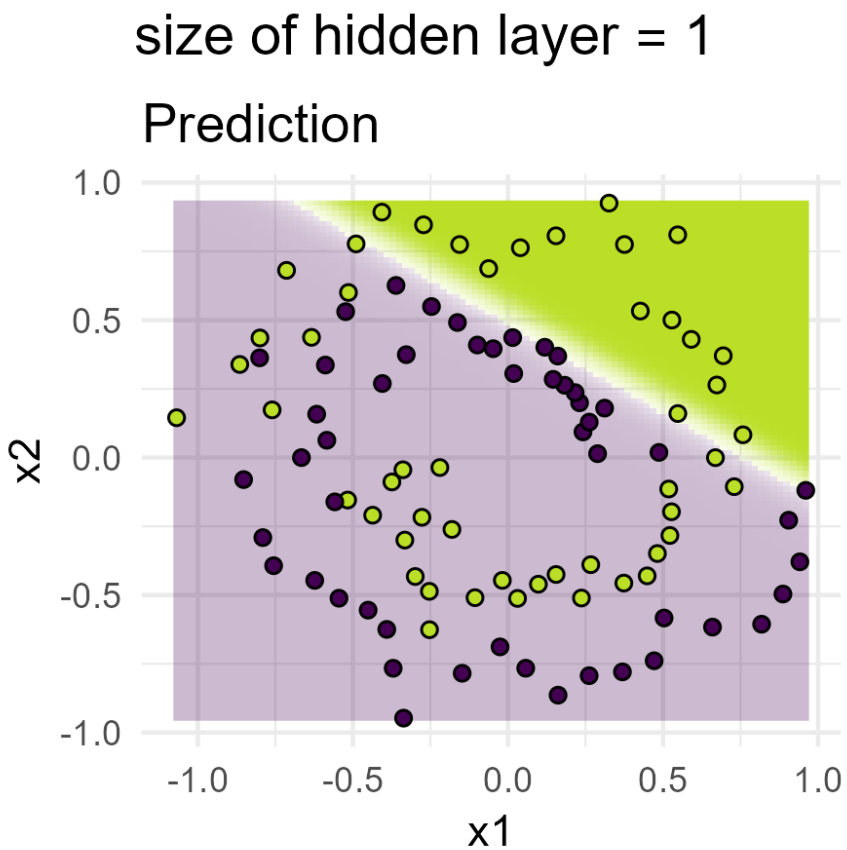
STRUCTURAL RISK MINIMIZATION

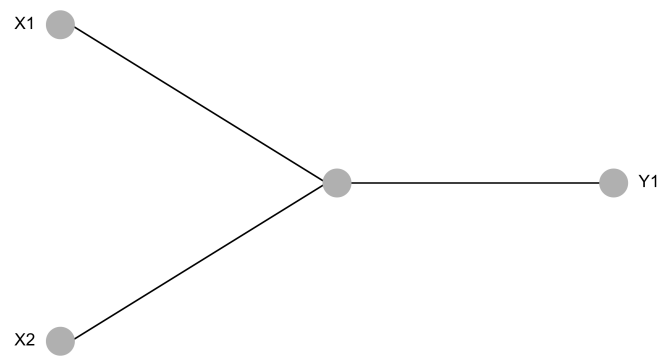
- SRM chooses the smallest k such that the optimal model from $\mathcal{H}_k$ found by ERM or RRM cannot significantly be outperformed by a model from a $\mathcal{H}_m$ with $m > k$
- Principle of Occam's razor
- One challenge might be choosing an adequate complexity measure, as for some models, multiple exist



STRUCTURAL RISK MINIMIZATION

Again `spirals`.
NN with 1 hidden layer, and fixed (small) L2 penalty.

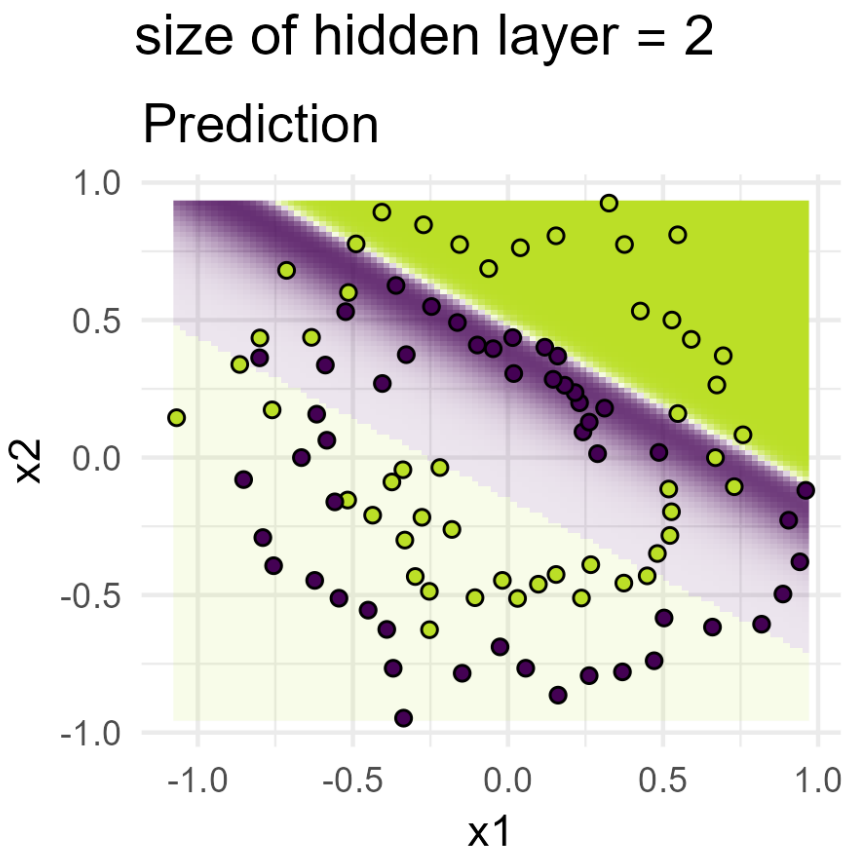


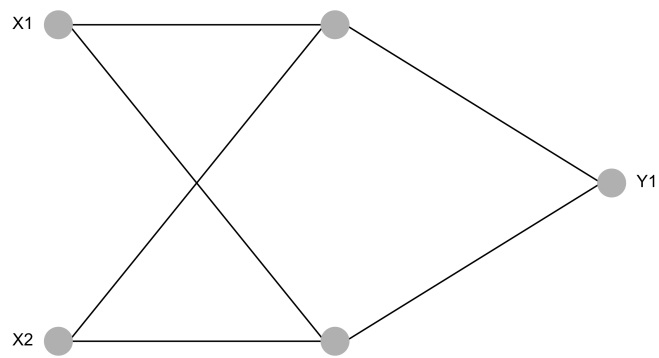


Size affects complexity and smoothness of decision boundary

STRUCTURAL RISK MINIMIZATION

Again `spirals`.
NN with 1 hidden layer, and fixed (small) L2 penalty.

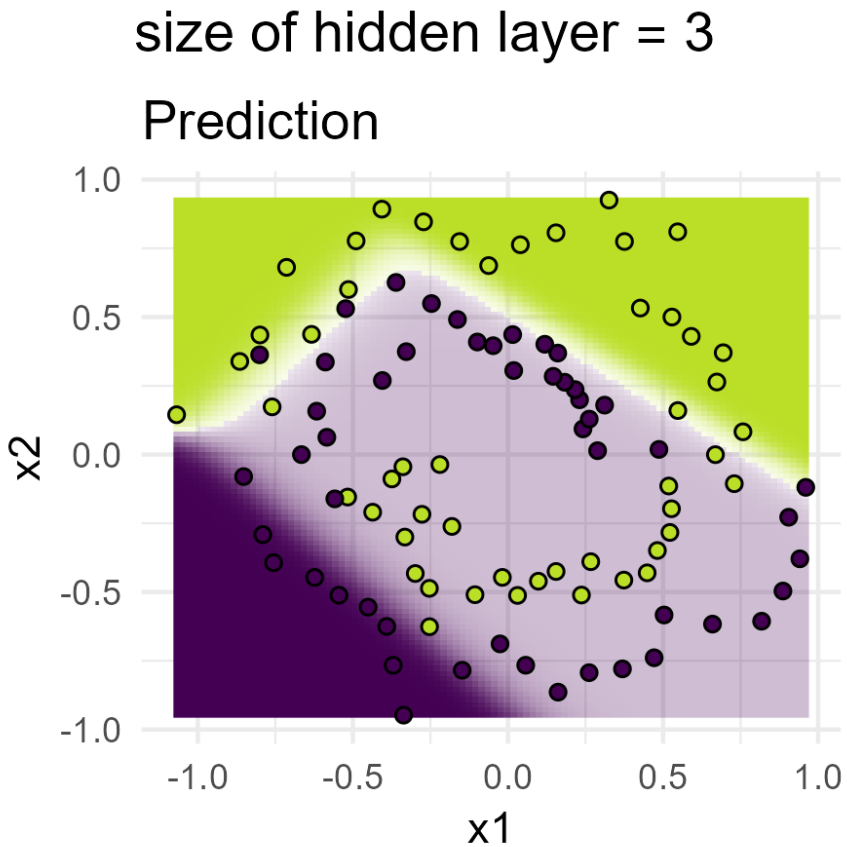


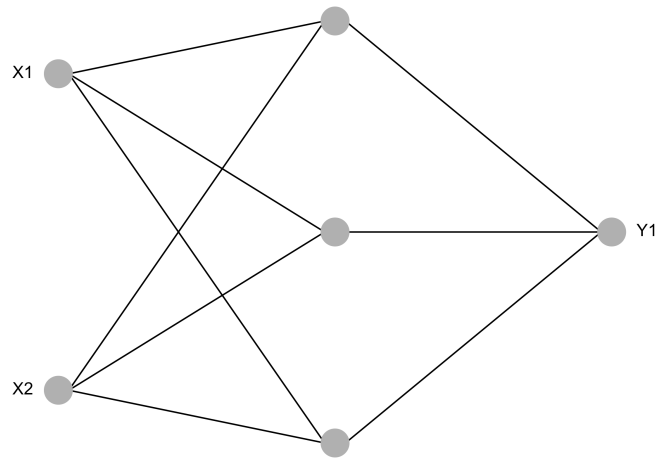


Size affects complexity and smoothness of decision boundary

STRUCTURAL RISK MINIMIZATION

Again `spirals`.
NN with 1 hidden layer, and fixed (small) L2 penalty.

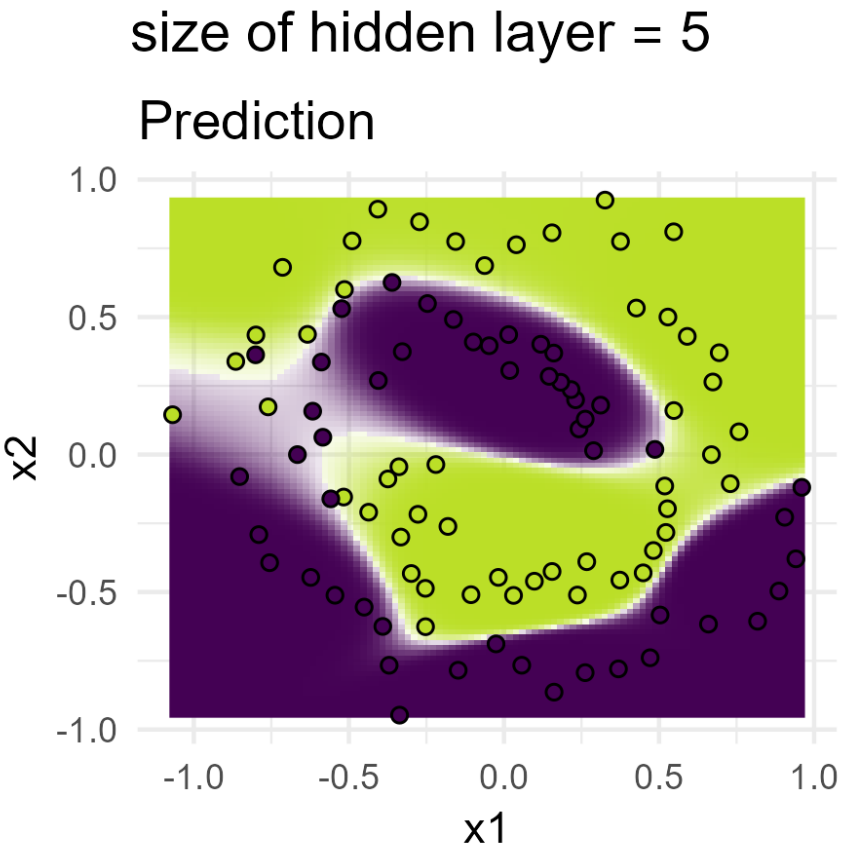


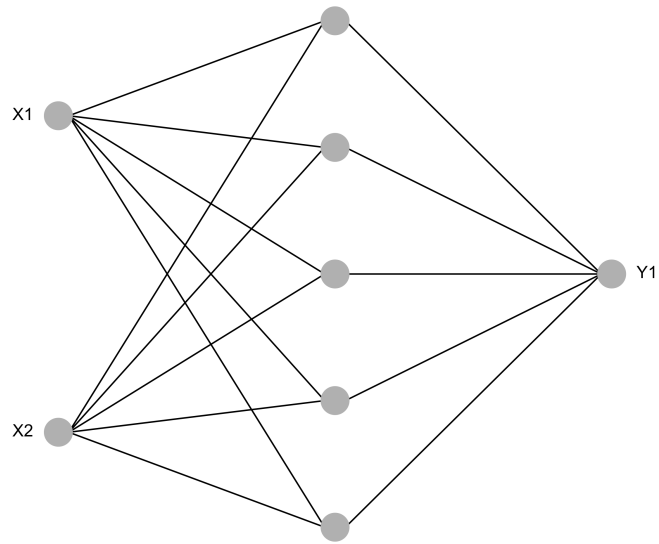


Size affects complexity and smoothness of decision boundary

STRUCTURAL RISK MINIMIZATION

Again `spirals`.
NN with 1 hidden layer, and fixed (small) L2 penalty.

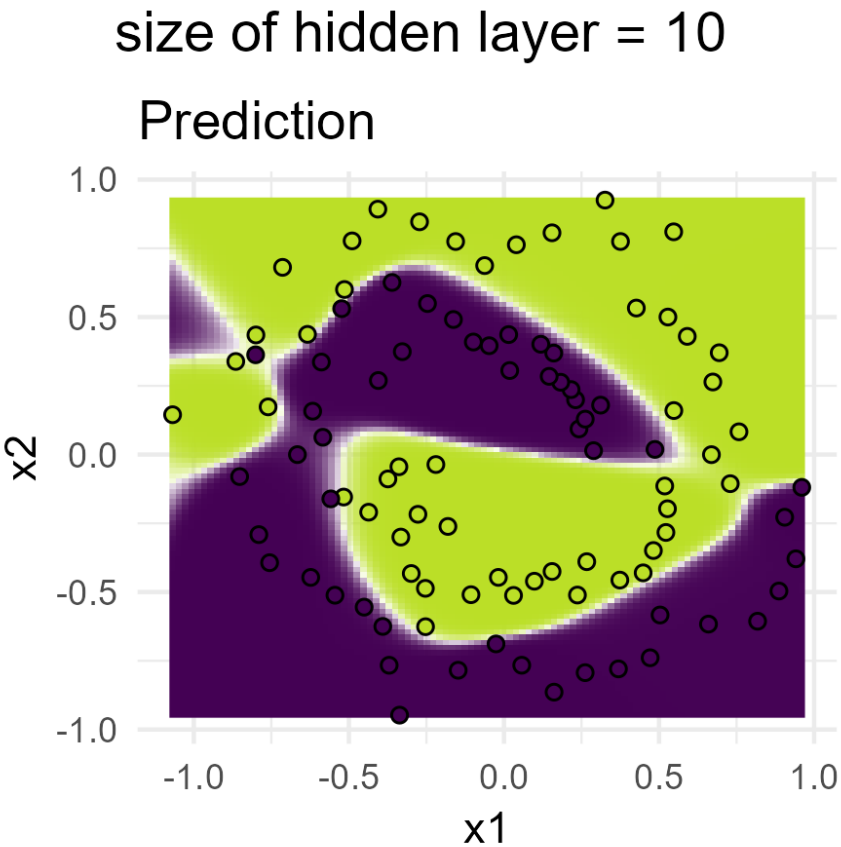


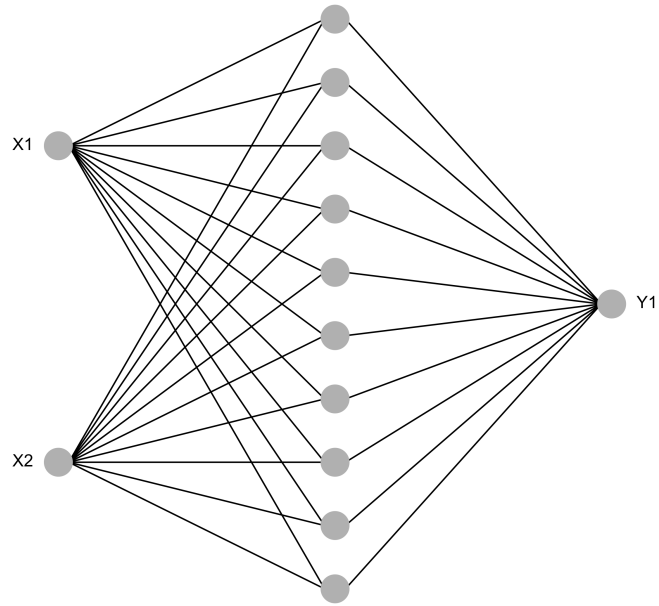


Size affects complexity and smoothness of decision boundary

STRUCTURAL RISK MINIMIZATION

Again `spirals`.
NN with 1 hidden layer, and fixed (small) L2 penalty.

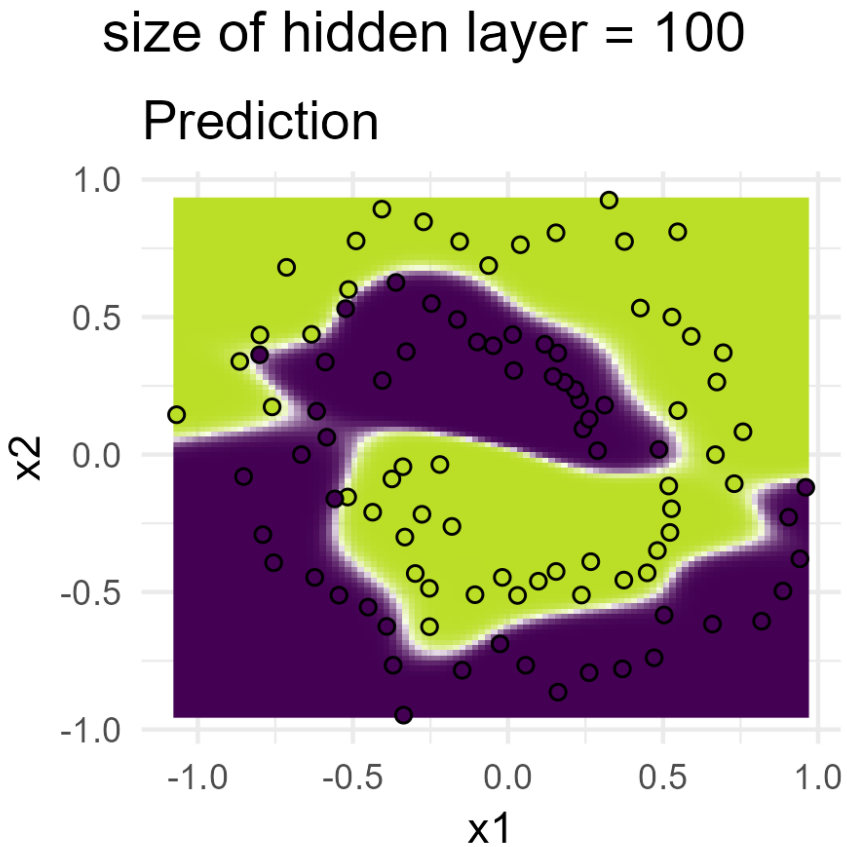


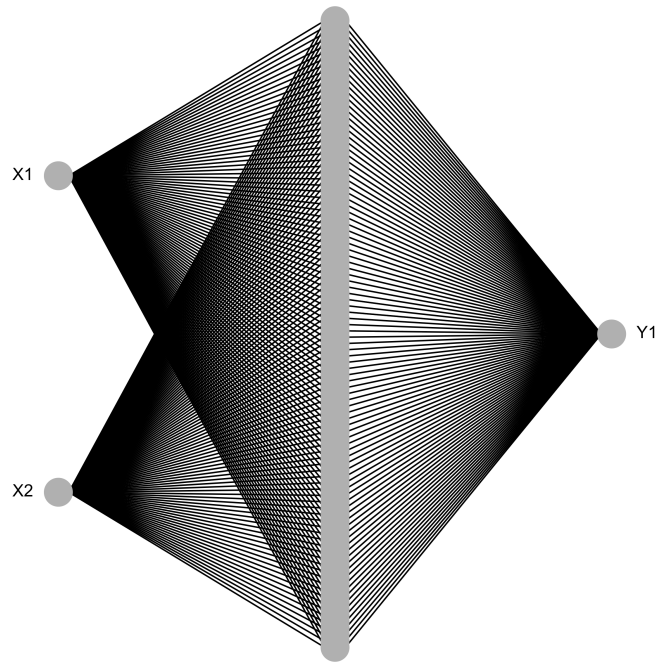


Size affects complexity and smoothness of decision boundary

STRUCTURAL RISK MINIMIZATION

Again `spirals`.
NN with 1 hidden layer, and fixed (small) L2 penalty.

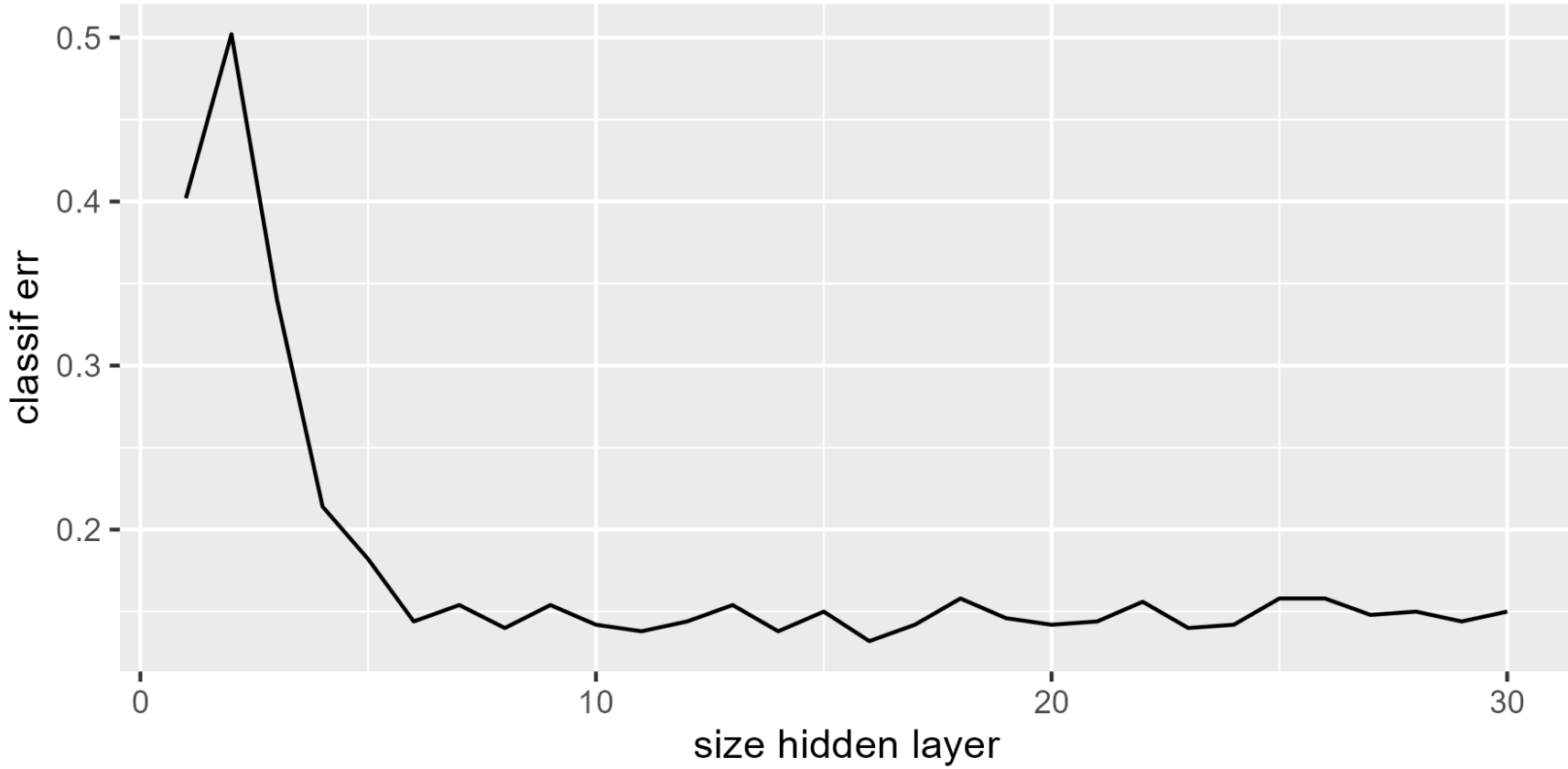




Size affects complexity and smoothness of decision boundary

STRUCTURAL RISK MINIMIZATION

Again, complexity vs CV score.

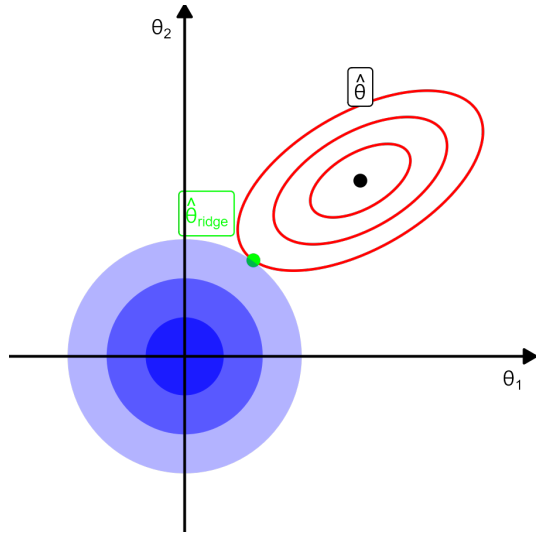


Minimal model with good generalization seems to size=10

STRUCTURAL RISK MINIMIZATION AND RRM

RRM can also be interpreted through SRM,
if we rewrite it in constrained form:

$$\begin{aligned} \min_{\boldsymbol{\theta}} \quad & \sum_{i=1}^{n} L\left(y^{(i)}, f\left(\mathbf{x}^{(i)} \mid \boldsymbol{\theta}\right)\right) \\ \text{s.t.} \quad & \|\boldsymbol{\theta}\|_2^2 \leq t \end{aligned}$$



Can interpret going through λ from large to small as through t from small to large. Constructs series of ERM problems with hypothesis spaces $\mathcal{H}_\lambda$, where we constrain norm of $\boldsymbol{\theta}$ to unit balls of growing sizes.

WEIGHT DECAY VS. L2 REGULARIZATION

Let's optimize *L2*-regularized risk of a model $f(\mathbf{x} \mid \boldsymbol{\theta})$

$$\min_{\boldsymbol{\theta}} \mathcal{R}_{\text{reg}}(\boldsymbol{\theta}) = \min_{\boldsymbol{\theta}} \mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) + \frac{\lambda}{2} \|\boldsymbol{\theta}\|_2^2$$

by GD. The gradient is

$$\nabla_{\boldsymbol{\theta}} \mathcal{R}_{\text{reg}}(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) + \lambda \boldsymbol{\theta}$$

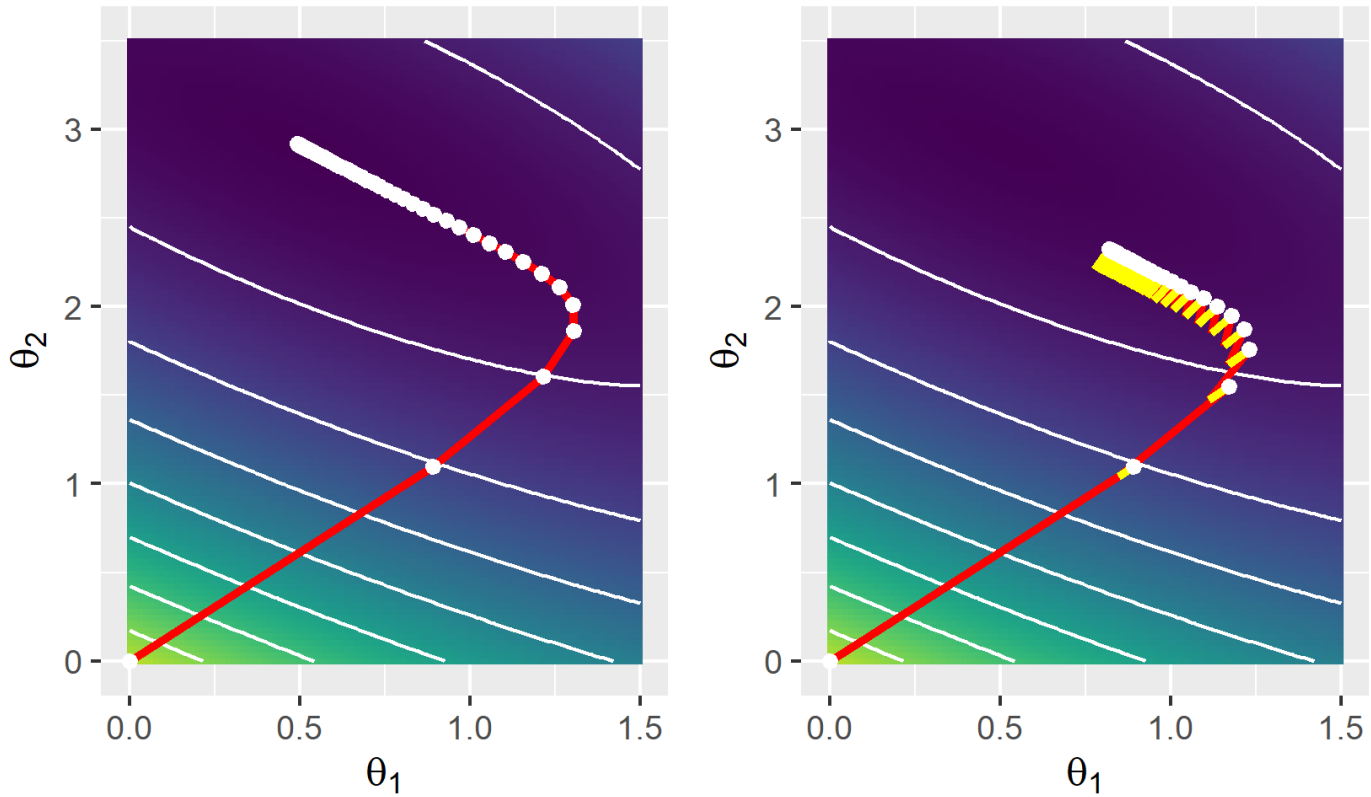
We iteratively update $\boldsymbol{\theta}$ by step size α times the negative gradient

$$\begin{aligned} \boldsymbol{\theta}^{[\text{new}]} &= \boldsymbol{\theta}^{[\text{old}]} - \alpha \left(\nabla_{\boldsymbol{\theta}} \mathcal{R}_{\text{emp}}(\boldsymbol{\theta}^{[\text{old}]}) + \lambda \boldsymbol{\theta}^{[\text{old}]} \right) \\ &= \boldsymbol{\theta}^{[\text{old}]} (1 - \alpha\lambda) - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{R}_{\text{emp}}(\boldsymbol{\theta}^{[\text{old}]}) \end{aligned}$$

We see how $\boldsymbol{\theta}^{[old]}$ decays in magnitude – for small α and λ – before we do the gradient step. Performing the decay directly, under this name, is a very well-known technique in DL - and simply *L2* regularization in disguise (for GD).

WEIGHT DECAY VS. L2 REGULARIZATION / 2

In GD With WD, we slide down neg. gradients of $\mathcal{R}_{\text{emp}}$,
but in every step, we are pulled back to origin.

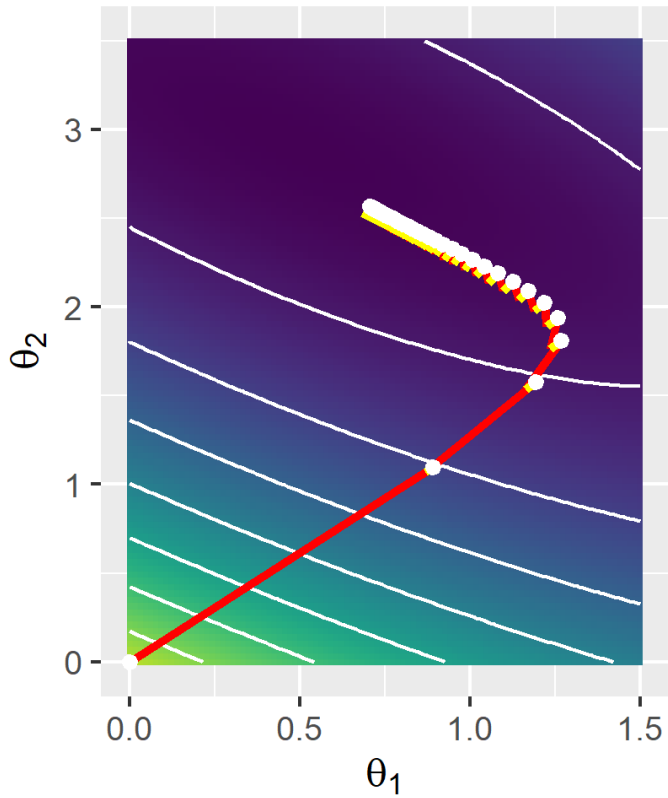


Without WD

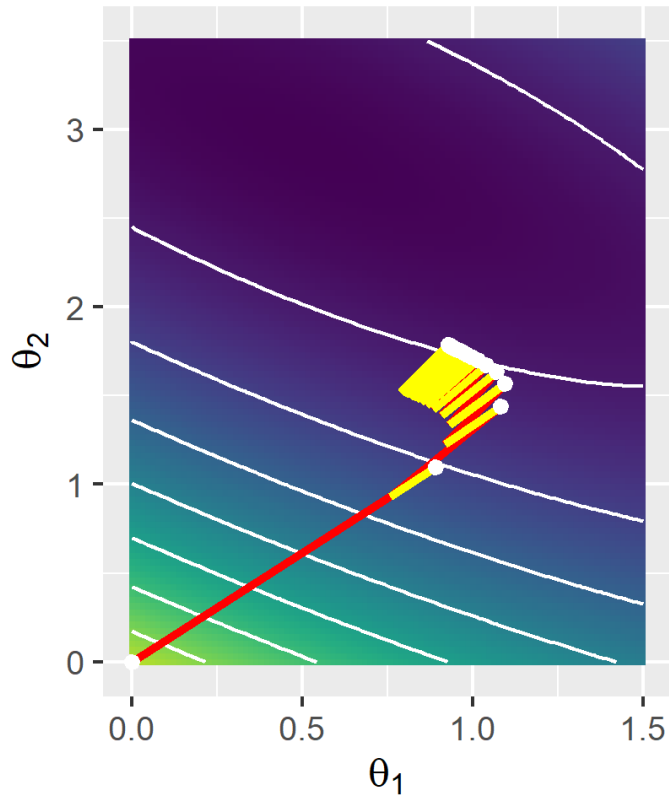
With GD

WEIGHT DECAY VS. L2 REGULARIZATION / 3

How strongly we are pulled back (for fixed α) depends on λ:



Small λ



Large λ

CAVEAT AND OTHER OPTIMIZERS

Caveat: Equivalence of weight decay and *L2* only holds for (S)GD!

- ▸ Hanson and Pratt 1988 originally define WD "decoupled" from gradient-updates $\alpha \nabla_{\boldsymbol{\theta}} \mathcal{R}_{\text{emp}}(\boldsymbol{\theta}^{[\text{old}]})$ as $\boldsymbol{\theta}^{[\text{new}]} = \boldsymbol{\theta}^{[\text{old}]}(1 - \lambda') - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{R}_{\text{emp}}(\boldsymbol{\theta}^{[\text{old}]})$
- This is equivalent to modern WD/*L2* (last slide) using reparameterization $\lambda' = \alpha\lambda$
- Consequence: if there is optimal λ', then optimal *L2* penalty is tightly coupled to α as $\lambda = \lambda'/\alpha$ (and vice versa)
- ▸ Loshchilov and Hutter 2019 show no equivalence of *L2* and WD possible for adaptive methods like Adam (Prop. 2)
- In many cases where SGD+*L2* works well, Adam+*L2* underperforms due to non-equivalence with WD
- They propose a variant of Adam decoupling WD from gradient updates (AdamW), increasing performance over Adam+*L2*

INTRODUCTION TO DEEP LEARNING

CONVOLUTIONAL NEURAL NETWORKS

- Convolutional Neural Networks (CNN, or ConvNet) are a powerful family of neural networks that are inspired by biological processes in which the connectivity pattern between neurons resembles the organization of the mamel visual cortex.

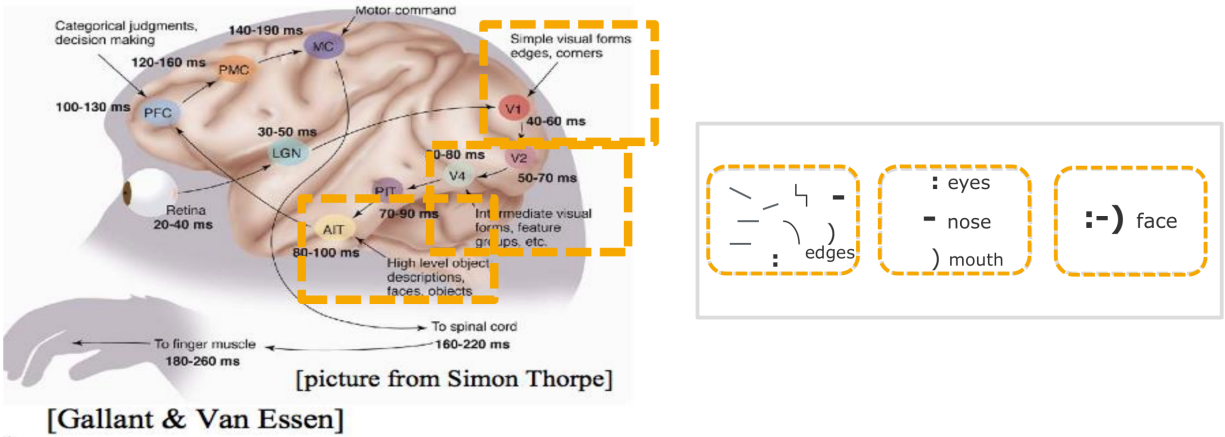


Figure: The ventral (recognition) pathway in the visual cortex has multiple stage: Retina - LGN - V1 - V2 - V4 - PIT - AIT etc., which consist of lots of intermediate representations.

CONVOLUTIONAL NEURAL NETWORKS

- Since 2012, given their success in the ILSVRC competition, CNNs are popular in many fields.
- Common applications of CNN-based architectures in computer vision are:
  - Image classification.
  - Object detection / localization.
  - Semantic segmentation.
- CNNs are widely applied in other domains such as natural language processing (NLP), audio, and time-series data.
- Basic idea: a CNN automatically extracts visual, or, more generally, spatial features from an input data such that it is able to make the optimal prediction based on the extracted features.
- It contains different building blocks and components.

CNNS - WHAT FOR?

Figure: All Tesla cars being produced now have full self-driving hardware. A convolutional neural network is used to map raw pixels from a single front-facing camera directly into steering commands. The system learns to drive in traffic, on local roads, with or without lane markings as well as on highways (The Tesla Team, 2016).

CNNS - WHAT FOR?

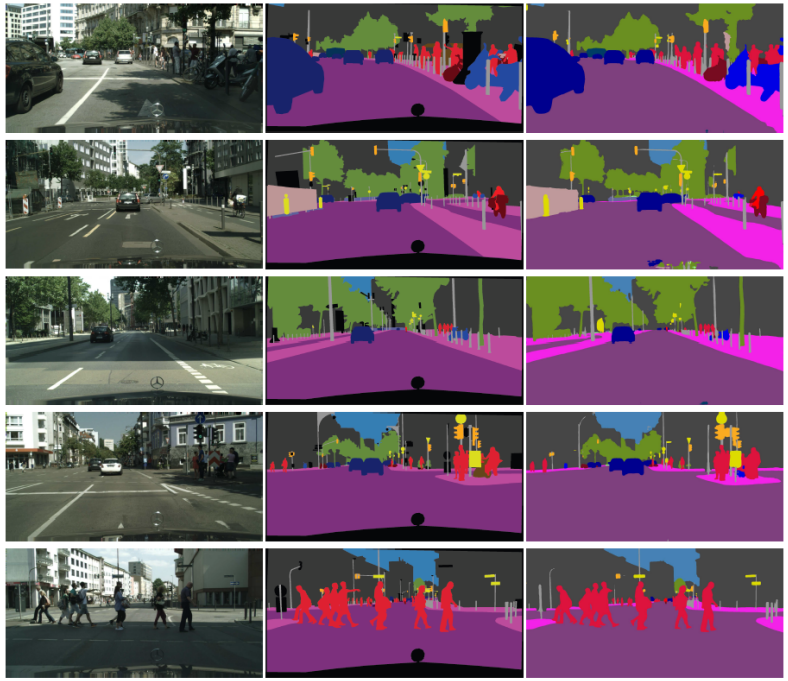


Figure: Given an input image, a CNN is first used to get the feature map of the last convolutional layer, then a pyramid parsing module is applied to harvest different sub-region representations, followed by upsampling and concatenation layers to form the final feature representation, which carries both local and global context information. Finally, the representation is fed into a convolution layer to get the final per-pixel prediction (Zhao et al., 2017).

CNNS - WHAT FOR?

Figure: Road segmentation: Aerial images and possibly outdated map pixels are segmented (Mnih & Hinton, 2010).

CNNS - WHAT FOR?

CNN for personalized medicine e.g tracking, diagnosis and localization of Covid-19 patients.

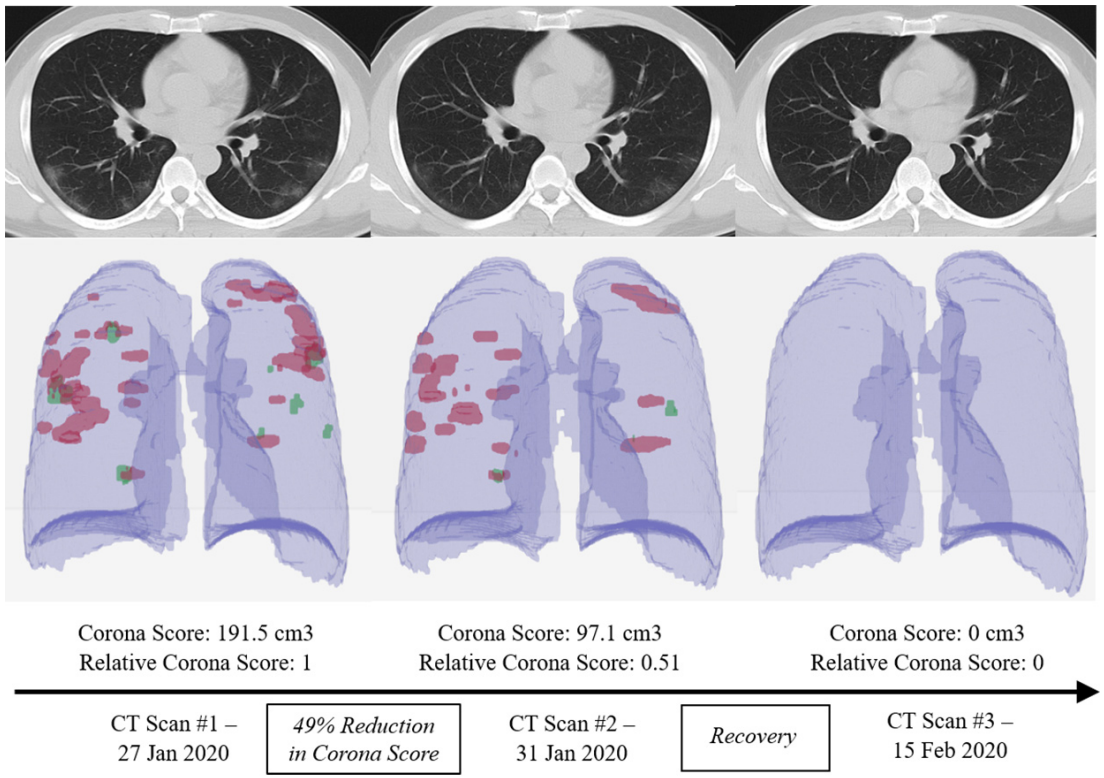


Figure: CNN based method (RadLogics algorithm) for personalized Covid-19 detection: three CT scans from a single Corona virus patient (Scudellari, 2020).

CNNS - WHAT FOR?

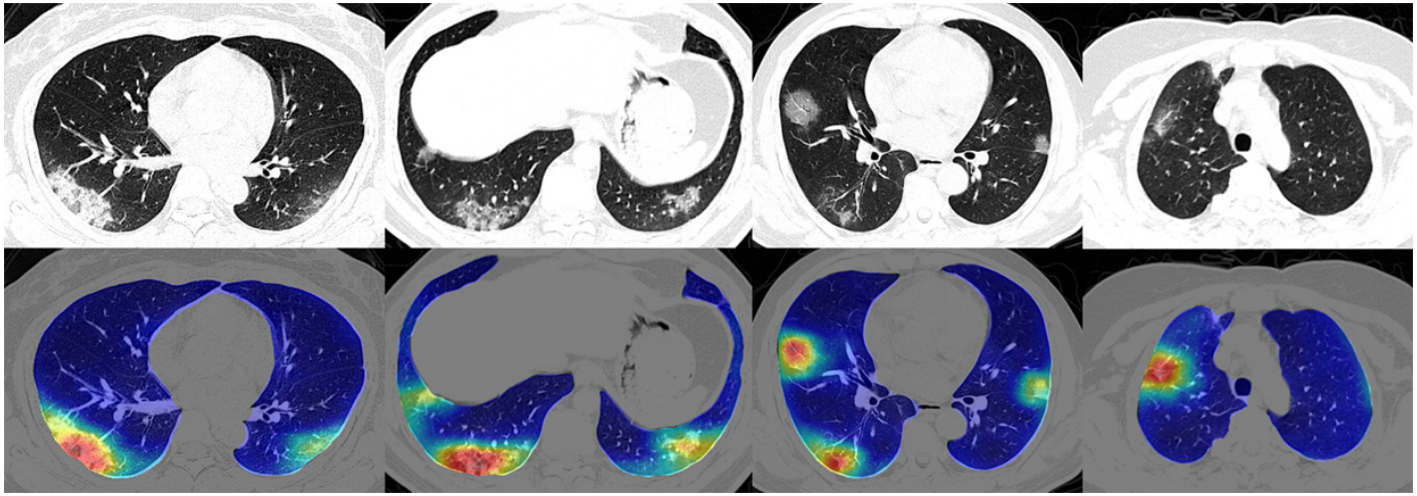


Figure: Four COVID-19 lung CT scans at the top with corresponding colored maps showing Corona virus abnormalities at the bottom using the RadLogics algorithm (Scudellari, 2020).

CNNS - WHAT FOR?

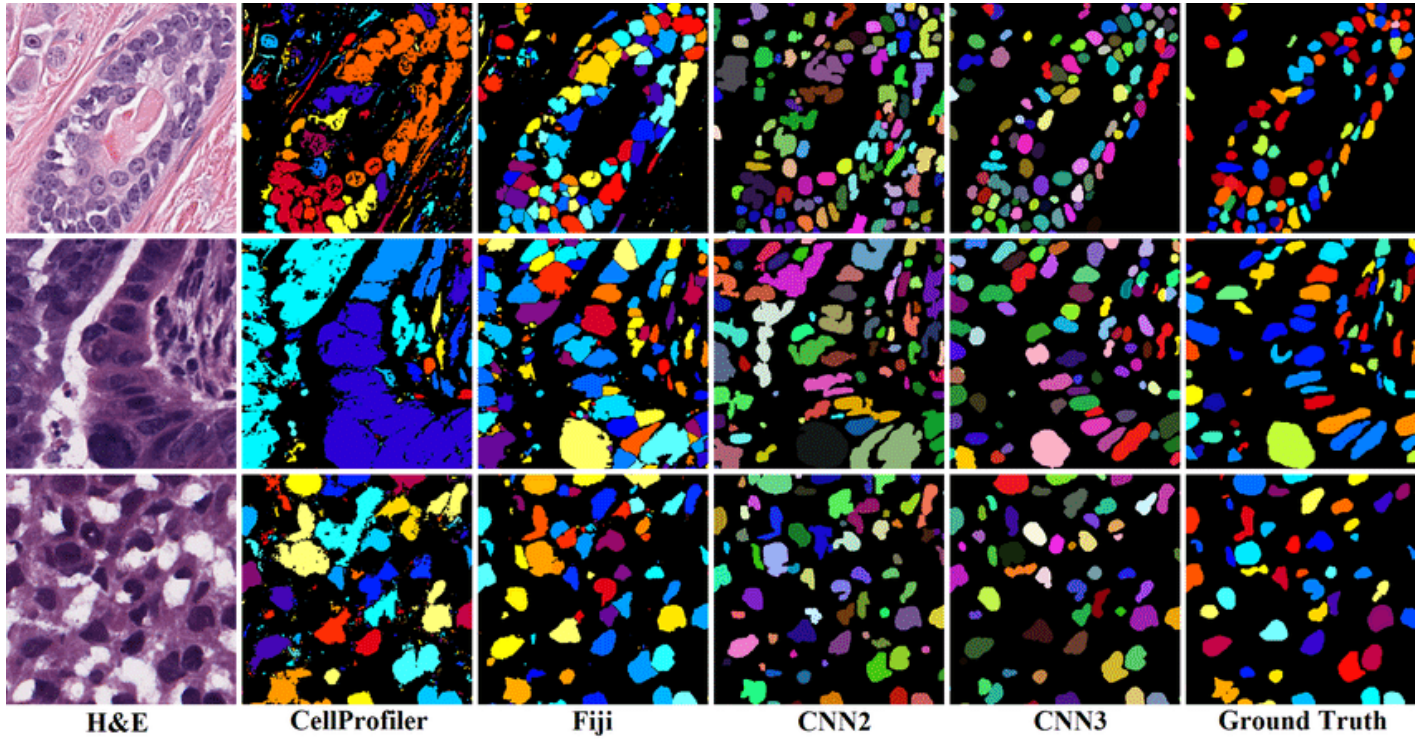


Figure: Various analyses in computational pathology are possible. For example, nuclear segmentation in digital microscopic tissue images enable extraction of high-quality features for nuclear morphometrics (Kumar et al., 2017).

CNNS - WHAT FOR?

Figure: Image Colorization: Given a grayscale photo as the input (top row), this network solves the problem of hallucinating a plausible color version of the photo (bottom row, i.e. the prediction of the network) (Zhang et al., 2016).

CNNS - WHAT FOR?

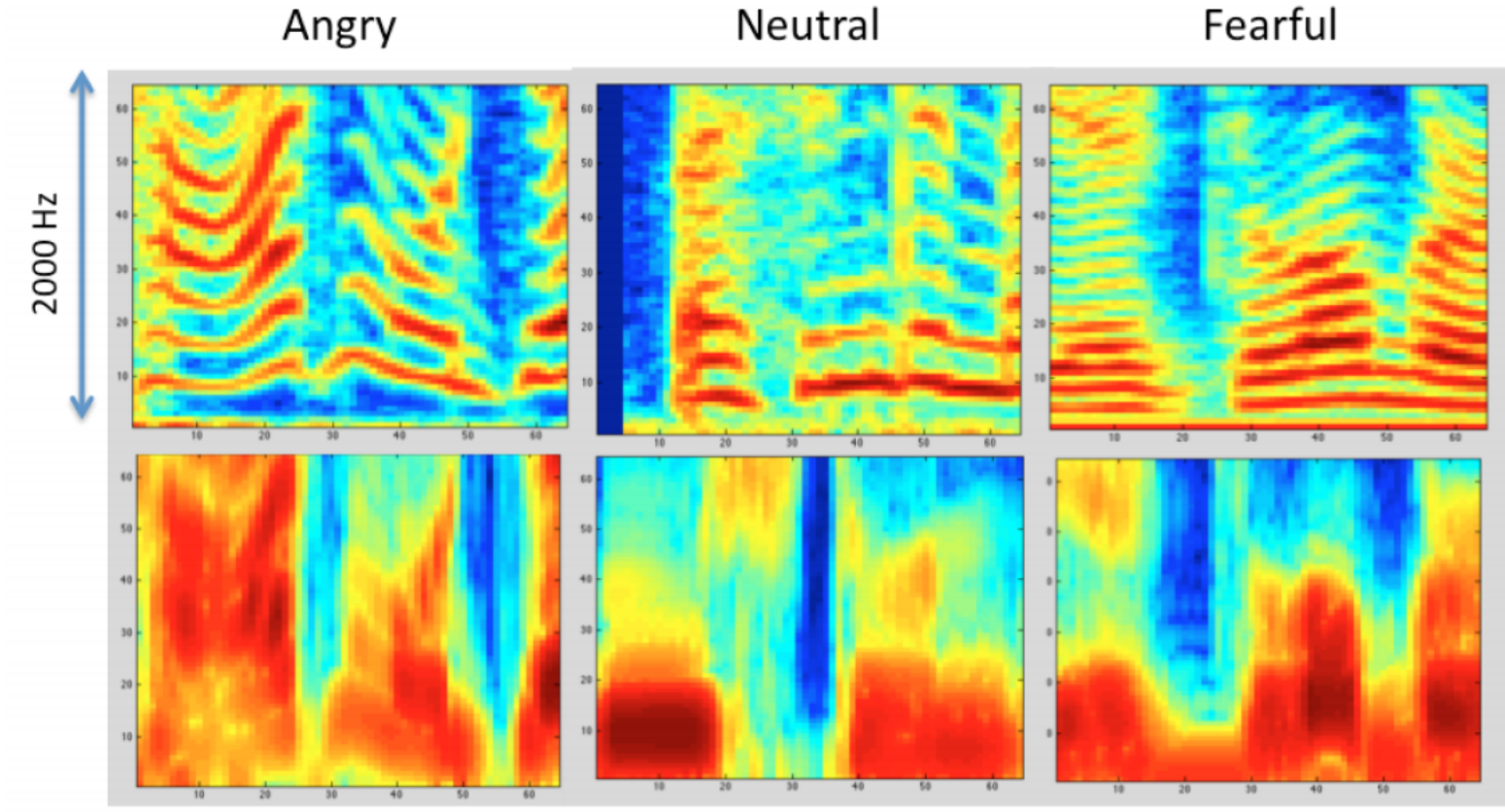


Figure: Speech recognition: Convolutional neural network is used to learn features from the audio data in order to classify emotions (Anand, 2015).

CNNS - A FIRST GLIMPSE

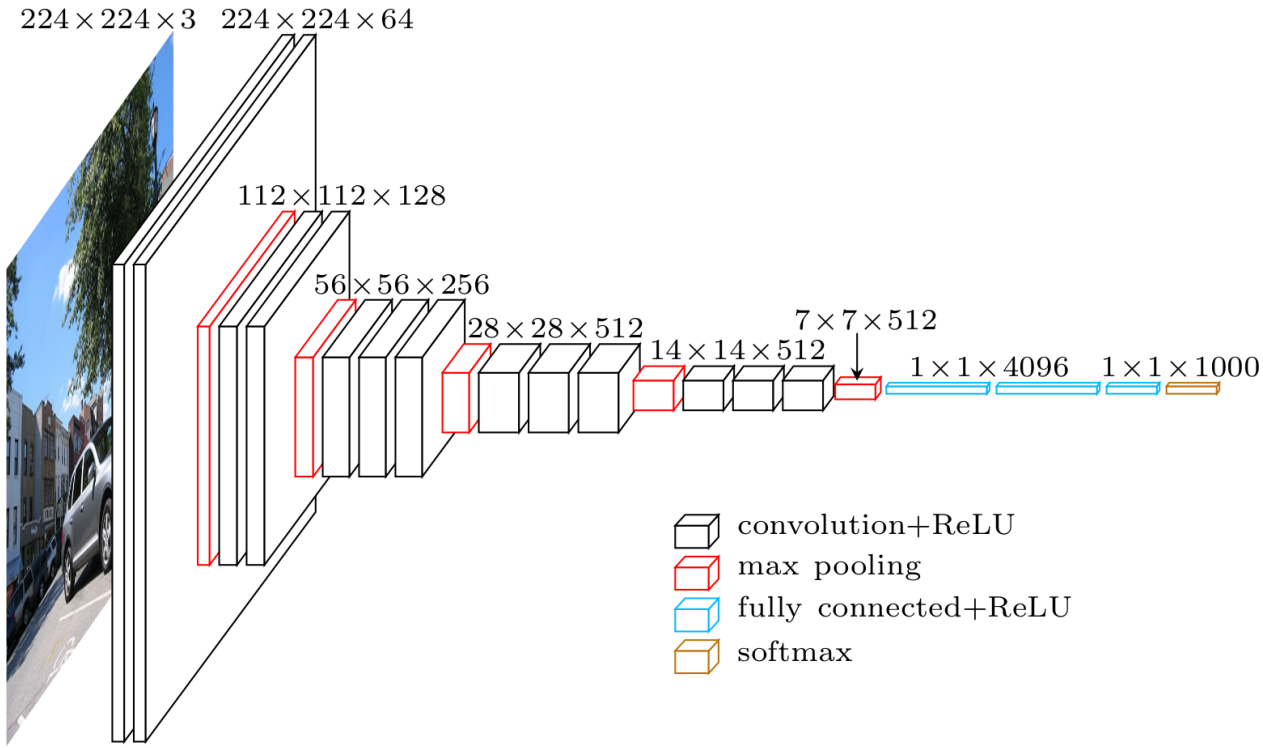


Figure: Example architecture (Simonyan & Zisserman, 2015)

- **Input layer** takes input data (e.g. image, audio).
- **Convolution layers** extract feature maps from the previous layers.
- **Pooling layers** reduce the dimensionality of feature maps and filter meaningful features.

FILTERS TO EXTRACT FEATURES

- Filters are widely applied in Computer Vision (CV) since the 70's.
- One prominent example: **Sobel-Filter**.
- It detects edges in images.

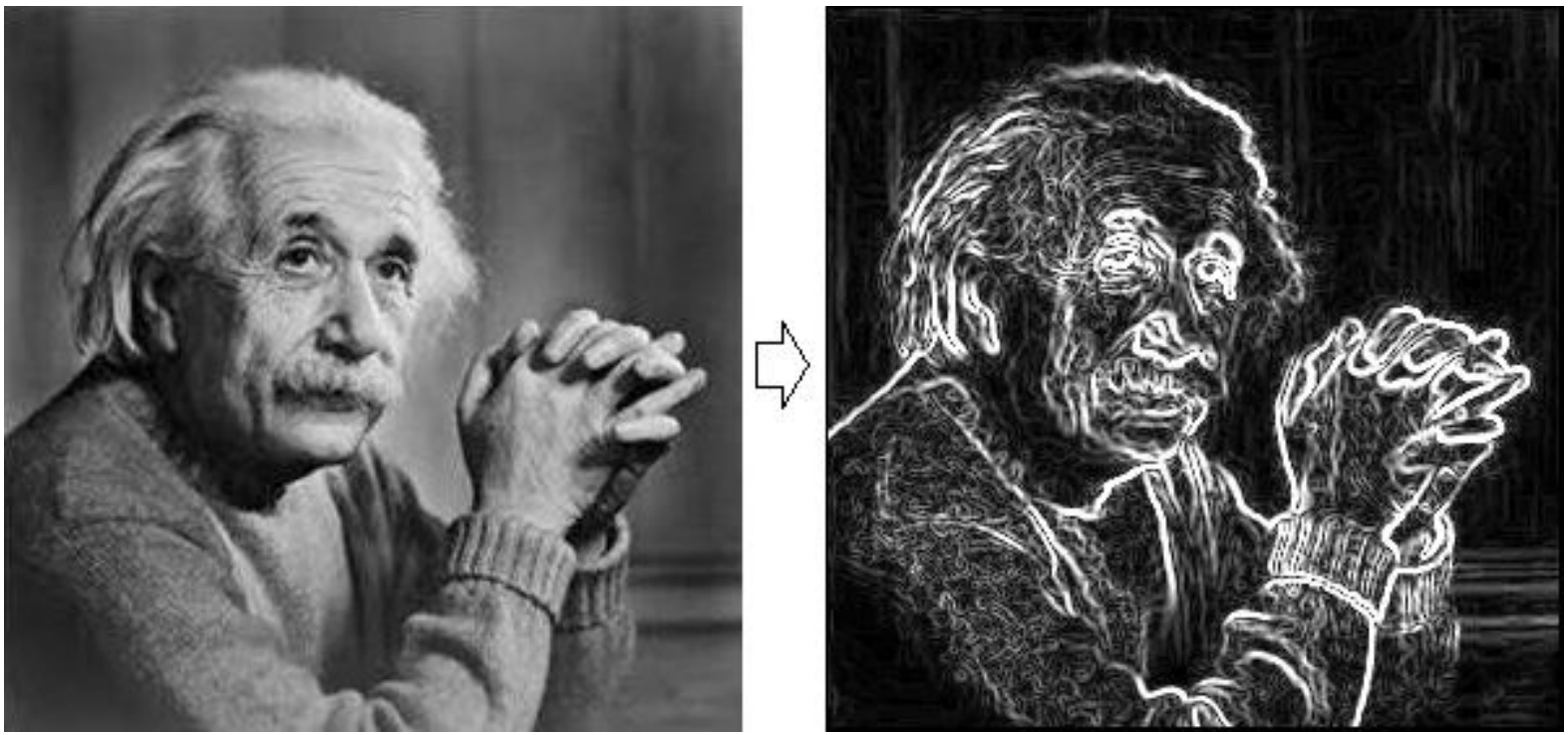


Figure: Sobel-filtered image (Qmegas, 2016).

FILTERS TO EXTRACT FEATURES

- Edges occur where the intensity over neighboring pixels changes fast.
- Thus, approximate the gradient of the intensity of each pixel.
- Sobel showed that the gradient image $\mathbf{G}_x$ of original image $\mathbf{A}$ in x-dimension can be approximated by:

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} = \mathbf{S}_x * \mathbf{A}$$

  where $*$ indicates a mathematical operation known as a **convolution**, not a traditional matrix multiplication.
- The filter matrix $\mathbf{S}_x$ consists of the product of an **averaging** and a **differentiation** kernel:

$$\underbrace{\begin{bmatrix} 1 & 2 & 1 \end{bmatrix}^T}_{\textit{averaging}} \underbrace{\begin{bmatrix} -1 & 0 & +1 \end{bmatrix}}_{\textit{differentiation}}$$

FILTERS TO EXTRACT FEATURES

- Similarly, the gradient image $\mathbf{G}_y$ in y-dimension can be approximated by:

$$\mathbf{G}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathbf{A} = \mathbf{S}_y * \mathbf{A}$$

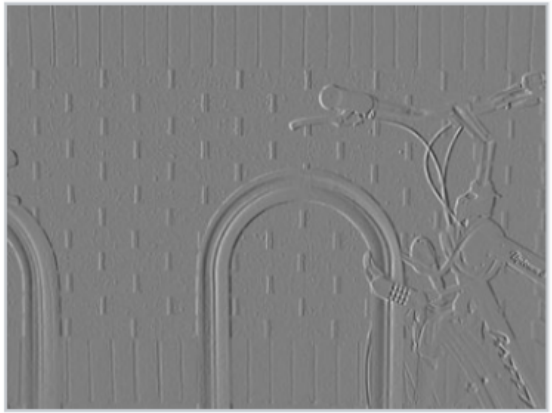
- The combination of both gradient images yields a dimension-independent gradient information $\mathbf{G}$:

$$\mathbf{G} = \sqrt{\mathbf{G}_x^2 + \mathbf{G}_y^2}$$

- These matrix operations were used to create the filtered picture of Albert Einstein.

HORIZONTAL VS VERTICAL EDGES

Input



Vertical edges detected by S_x

Horizontal edges detected by S_y

Combined

Figure: Sobel filtered images where outputs are normalized in each case (Wikipedia, 2022).

FILTERS TO EXTRACT FEATURES

- Let's do this on a dummy image.
- How to represent a digital image?

FILTERS TO EXTRACT FEATURES

0	0	0	0	255	255	0	0	0	0
0	0	0	255	255	255	255	0	0	0
0	0	0	0	255	255	0	0	0	0
255	0	0	255	255	255	255	0	0	0
0	0	255	255	255	255	255	255	0	0
0	255	0	255	255	255	255	0	255	0
0	0	0	255	255	255	255	0	0	0
0	0	0	255	255	255	255	0	0	0
0	0	0	255	0	0	255	0	0	255
0	0	255	255	0	0	255	255	0	0

- Basically as an array of integers.

FILTERS TO EXTRACT FEATURES

Sobel-Operator

$$S_X = \begin{bmatrix} -1 & 0 & +1 \\ -2 & \boxed{0} & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

0	0	0	0	255	255	0	0	0	0
0	0	0	255	255	255	255	0	0	0
0	0	0	0	255	255	0	0	0	0
255	0	0	255	255	255	255	0	0	0
0	0	255	255	255	255	255	255	0	0
0	255	0	255	255	255	255	0	255	0
0	0	0	255	255	255	255	0	0	0
0	0	0	255	255	255	255	0	0	0
0	0	0	255	0	0	255	0	0	255
0	0	255	255	0	0	255	255	0	0

- $\mathbf{S}_X$ enables us to to detect vertical edges!

FILTERS TO EXTRACT FEATURES

Sobel-Operator

$$S_X = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

0	0	0	0	255	255	0	0	0	0
0	0	0	255	255	255	255	0	0	0
0	0	0	0	255	255	0	0	0	0
255	0	0	255	255	255	255	0	0	0
0	0	255	255	255	255	255	255	0	0
0	255	0	255	255	255	255	0	255	0
0	0	0	255	255	255	255	0	0	0
0	0	0	255	255	255	255	0	0	0
0	0	0	255	0	0	255	0	0	255
0	0	255	255	0	0	255	255	0	0

FILTERS TO EXTRACT FEATURES

Sobel-Operator

$$S_X = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

0	0	0	0	255	255	0	0	0	0
0	0	0	255	255	255	255	0	0	0
0	0	0	0	255	255	0	0	0	0
255	0	0	255	255	255	255	0	0	0
0	0	255	255	255	255	255	255	0	0
0	255	0	255	255	255	255	0	255	0
0	0	0	255	255	255	255	0	0	0
0	0	0	255	255	255	255	0	0	0
0	0	0	255	0	0	255	0	0	255
0	0	255	255	0	0	255	255	0	0

$$\begin{aligned}(\mathbf{G}_x)_{(i,j)} = (\mathbf{I} \star \mathbf{S}_x)_{(i,j)} = &-1 \cdot 0 + 0 \cdot 255 + \mathbf{1 \cdot 255} \\ &-2 \cdot 0 + 0 \cdot 0 + \mathbf{2 \cdot 255} \\ &-1 \cdot 0 + 0 \cdot 255 + \mathbf{1 \cdot 255}\end{aligned}$$

FILTERS TO EXTRACT FEATURES

0	510	1020	510	-510	-1020	-510	0
-255	510	1020	510	-510	-1020	-510	0
-255	765	765	255	-255	-765	-765	-255
255	765	510	0	0	-510	-765	-510
255	510	765	0	0	-765	-510	-255
0	765	1020	0	0	-1020	-765	0
0	1020	765	-255	255	-765	-1020	255
255	1020	0	-765	765	0	-1020	255

- Applying the Sobel-Operator to every location in the input yields the **feature map**.

FILTERS TO EXTRACT FEATURES

128	191	255	191	64	0	64	128
96	191	255	191	64	0	64	128
96	223	223	159	96	32	32	96
159	223	191	128	128	64	32	64
159	191	223	128	128	32	64	96
128	223	255	128	128	0	32	128
128	255	223	96	159	32	0	159
159	255	128	32	223	128	0	159

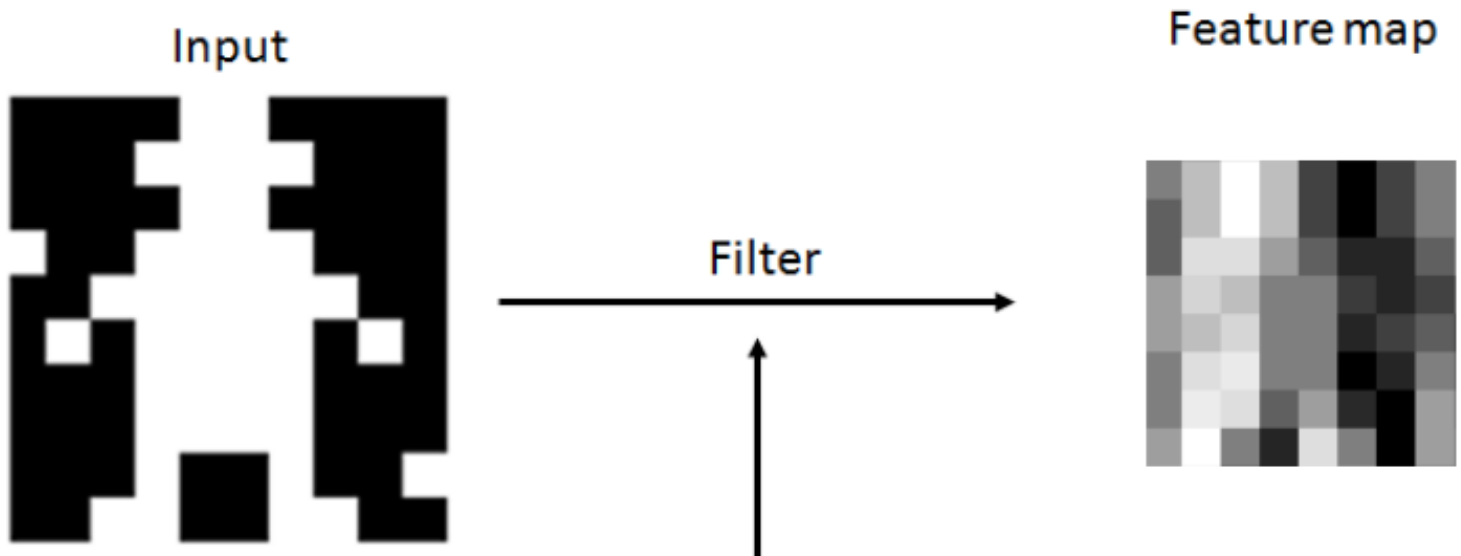
- Normalized feature map reveals vertical edges.
- Note the dimensional reduction compared to the dummy image.

WHY DO WE NEED TO KNOW ALL OF THAT?

- What we just did was extracting **pre-defined** features from our input (i.e. edges).
- A convolutional neural network does almost exactly the same: “extracting features from the input”.

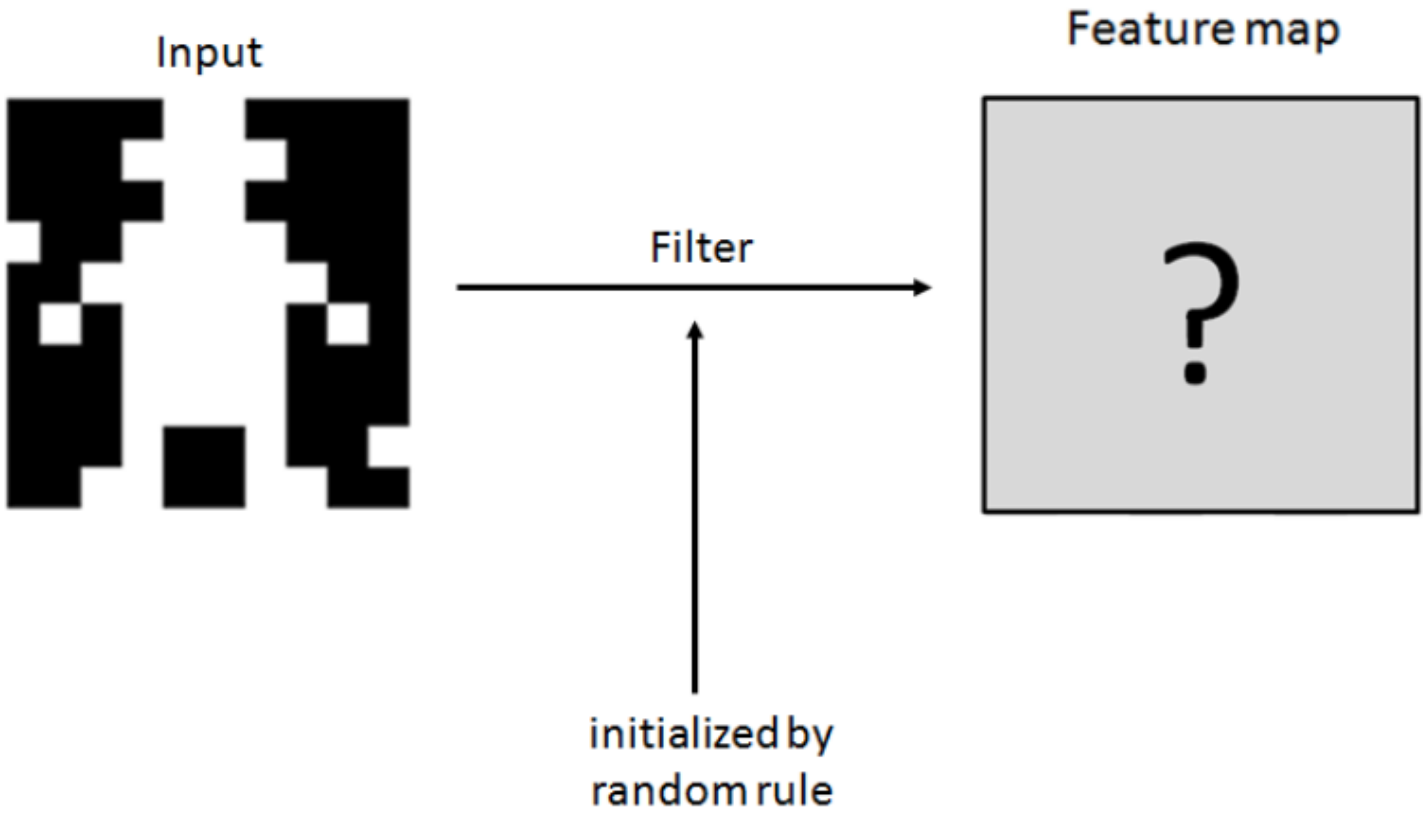
  $\Rightarrow$ The main difference is that we usually do not tell the CNN what to look for (pre-define them), **the CNN decides itself**.
- In a nutshell:
  - We initialize a lot of random filters (like the Sobel but just random entries) and apply them to our input.
  - Then, a classifier which (e.g. a feed forward neural net) uses them as input data.
  - Filter entries will be adjusted by common gradient descent methods.

WHY DO WE NEED TO KNOW ALL OF THAT?



$$S_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

WHY DO WE NEED TO KNOW ALL OF THAT?

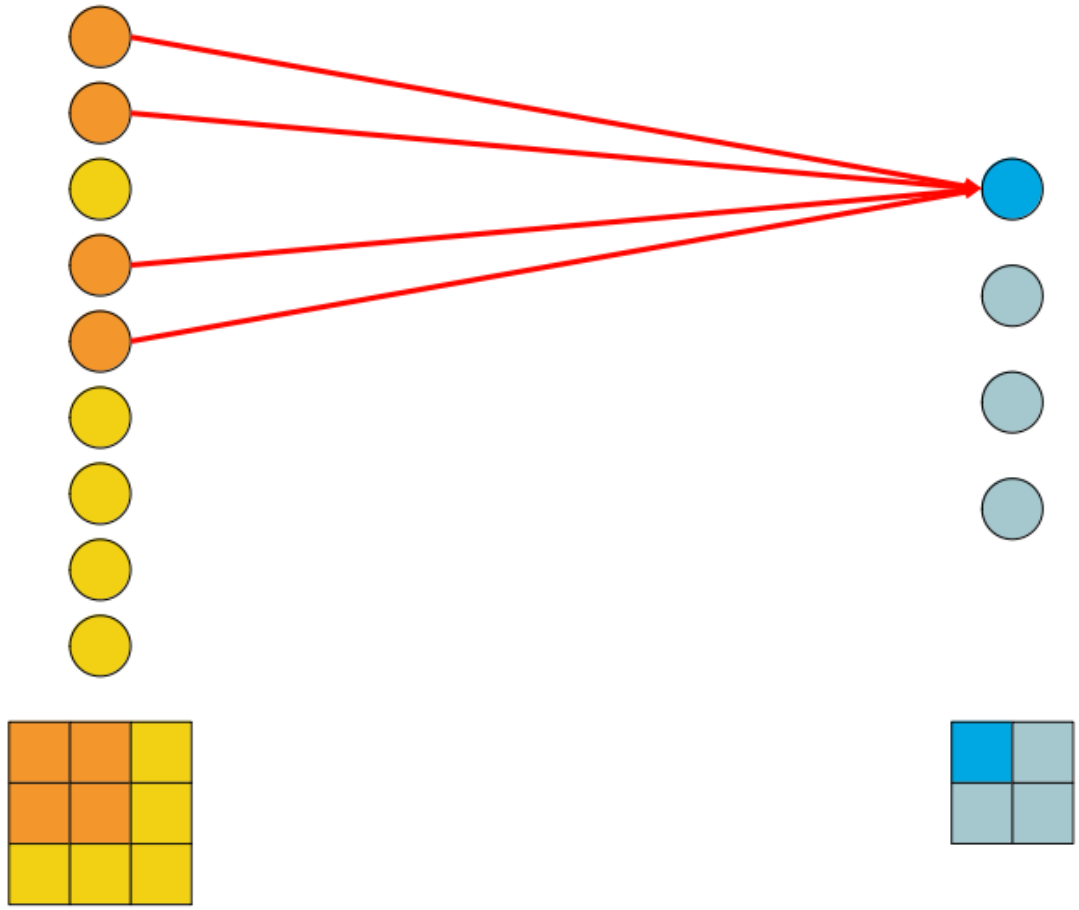


$$S_X = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix}$$

WORKING WITH IMAGES

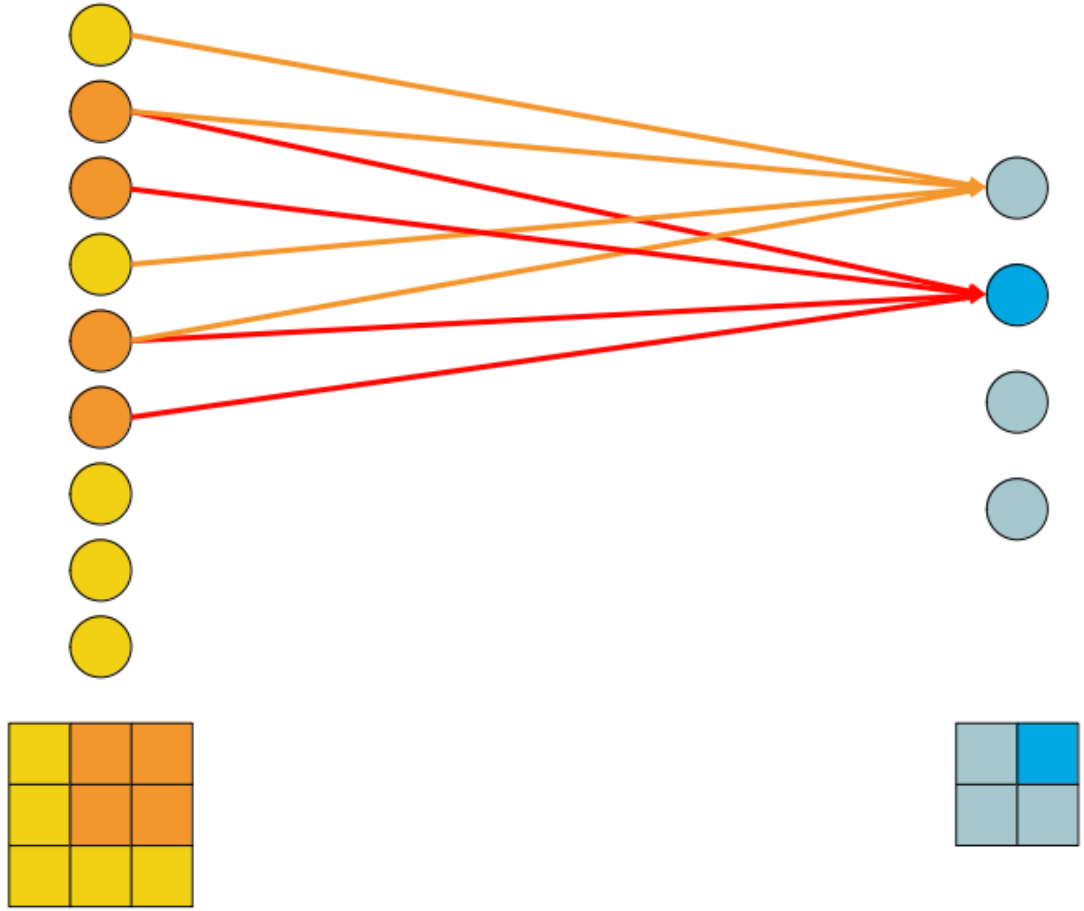
- In order to understand the functionality of CNNs, we have to familiarize ourselves with some properties of images.
- Grey scale images:
  - Matrix with dimensions **h**eight $\times$ **w**idth $\times$ 1.
  - Pixel entries differ from 0 (black) to 255 (white).
- Color images:
  - Tensor with dimensions **h**eight $\times$ **w**idth $\times$ 3.
  - The depth 3 denotes the RGB values (red - green - blue).
- Filters:
  - A filter's depth is **always** equal to the input's depth!
  - In practice, filters are usually square.
  - Thus we only need one integer to define its size.
  - For example, a filter of size 2 applied on a color image actually has the dimensions $2 \times 2 \times 3$.

SPARSE INTERACTIONS



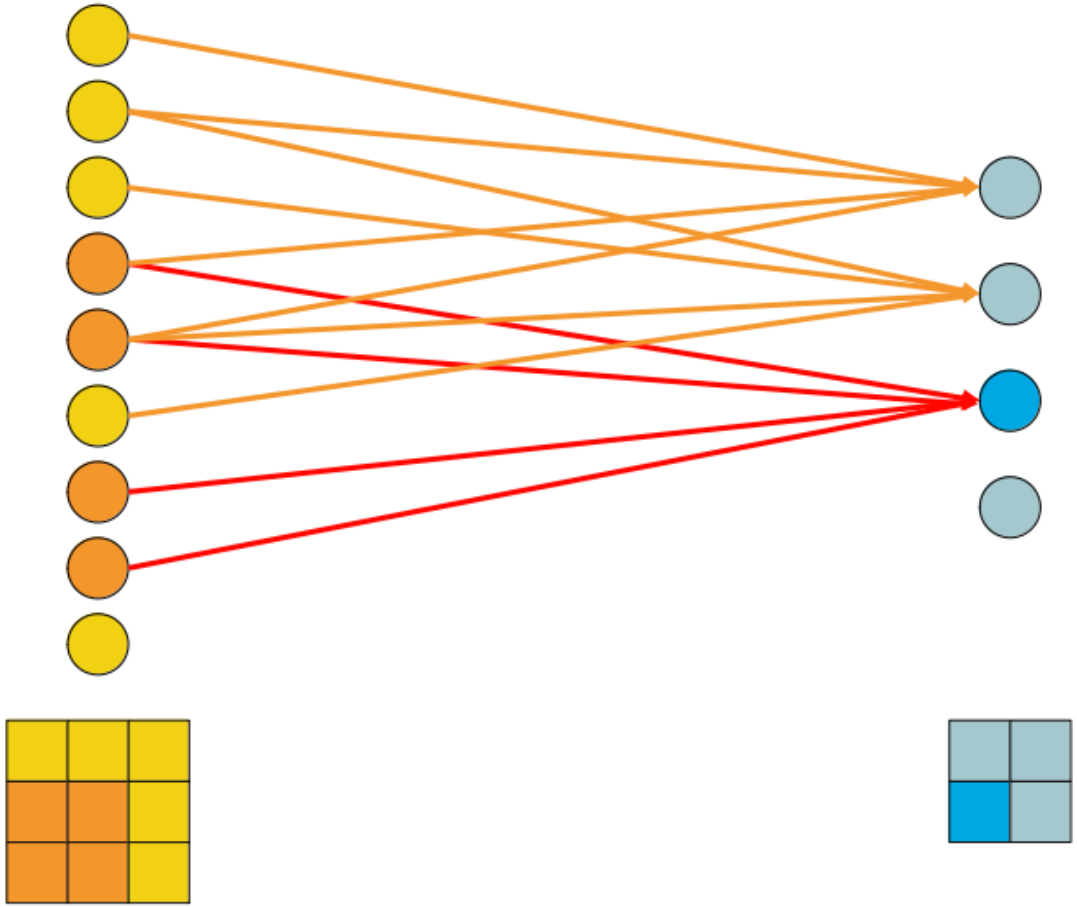
- We want to use the “neuron-wise” representation of our CNN.
- Moving the filter to the first spatial location yields the first entry of the feature map which is composed of these four connections.

SPARSE INTERACTIONS



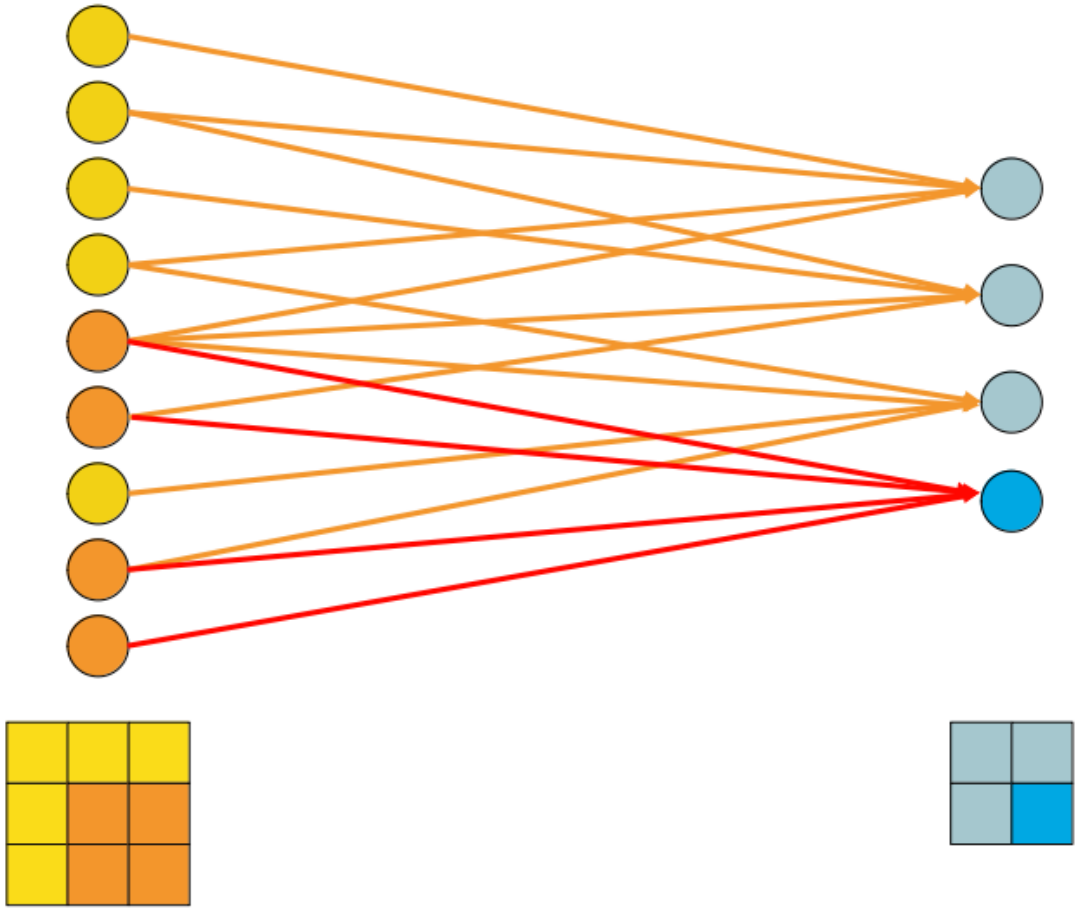
- Similarly...

SPARSE INTERACTIONS



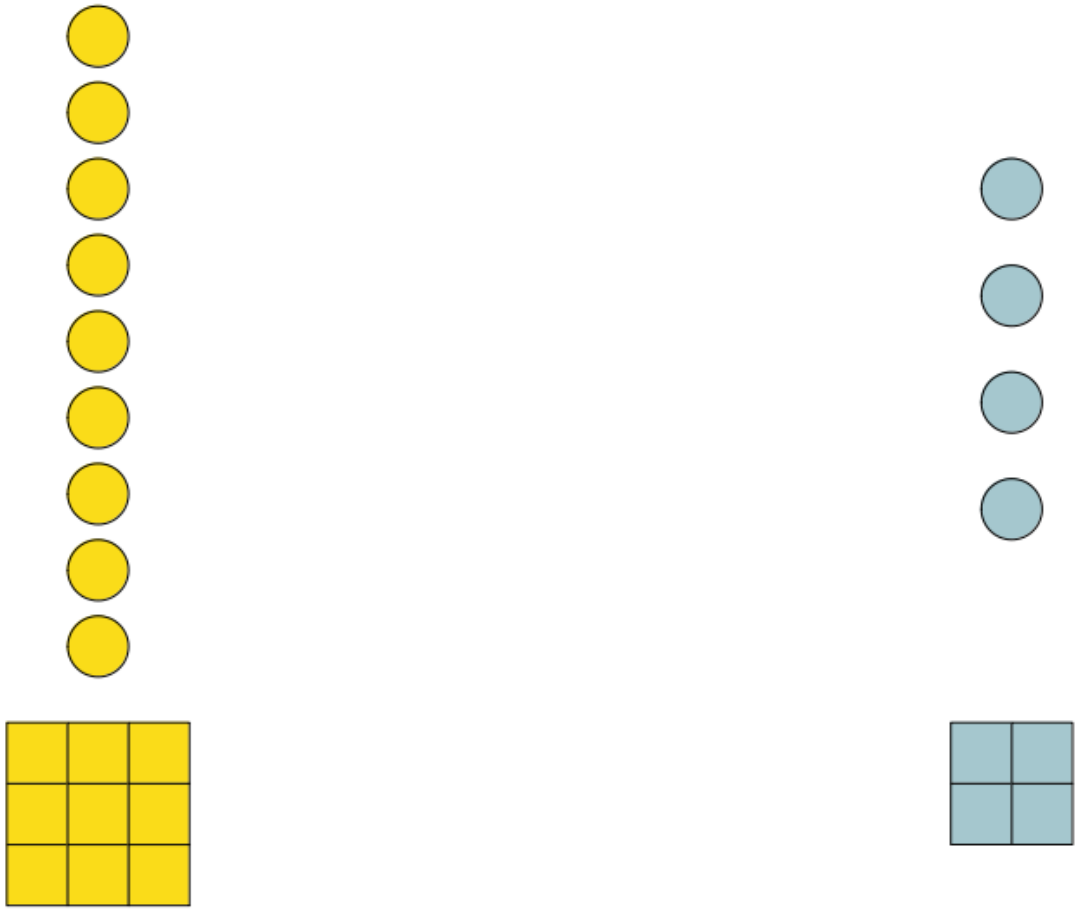
- Similarly...

SPARSE INTERACTIONS



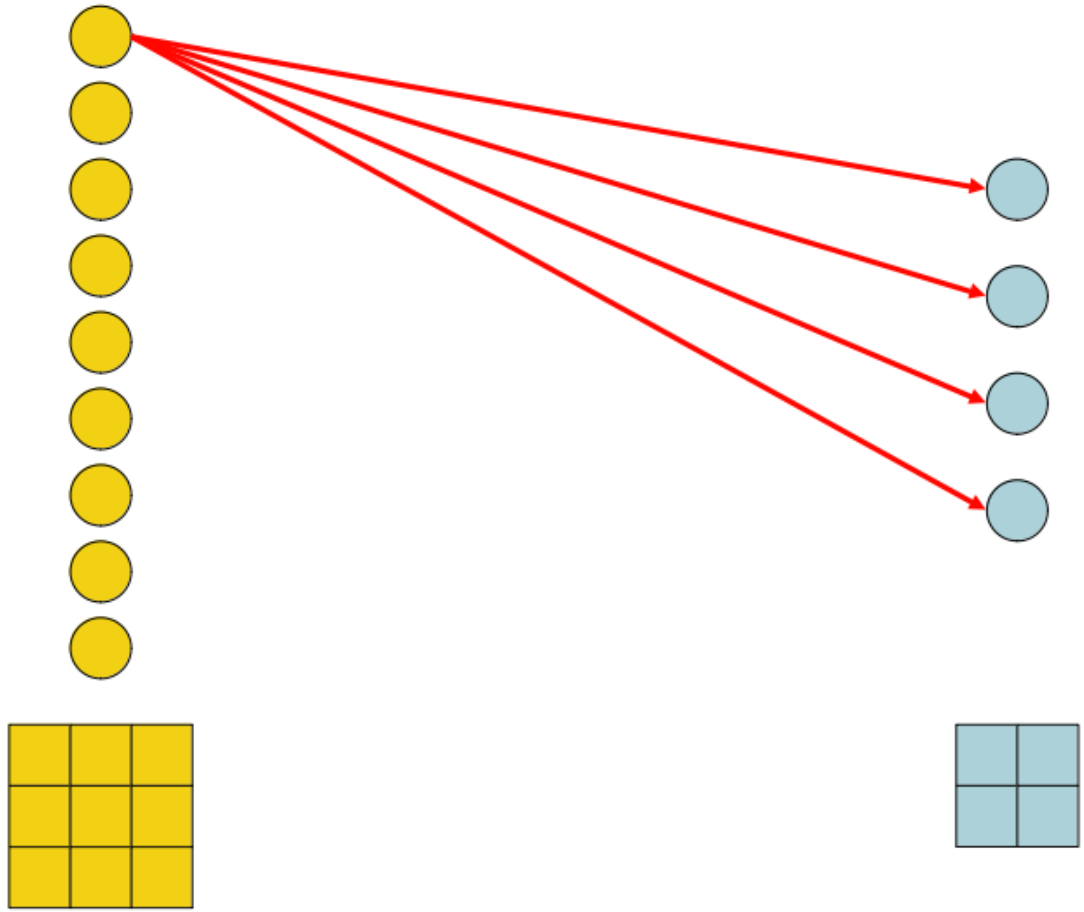
- and finally s_{22} by these and in total, we obtain 16 connections!

SPARSE INTERACTIONS



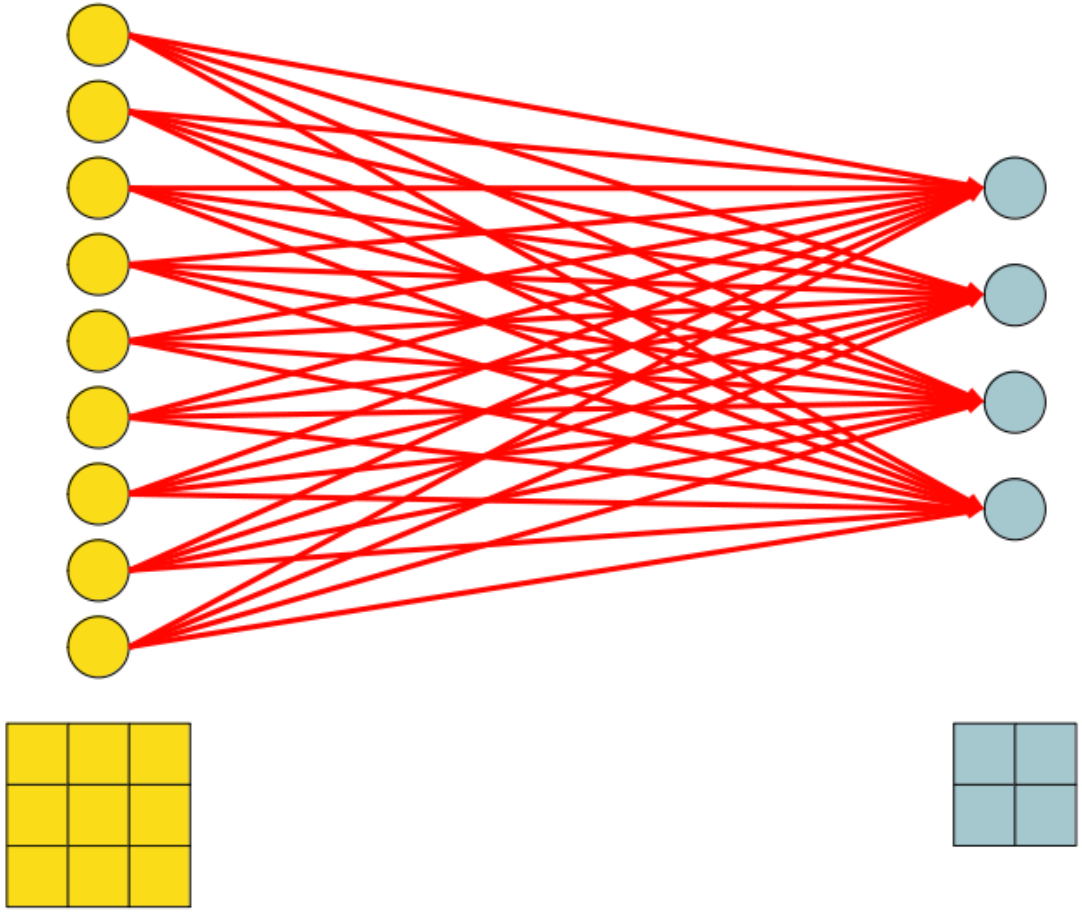
- Assume we would replicate the architecture with a dense net.

SPARSE INTERACTIONS



- Each input neuron is connected with each hidden layer neuron.

SPARSE INTERACTIONS

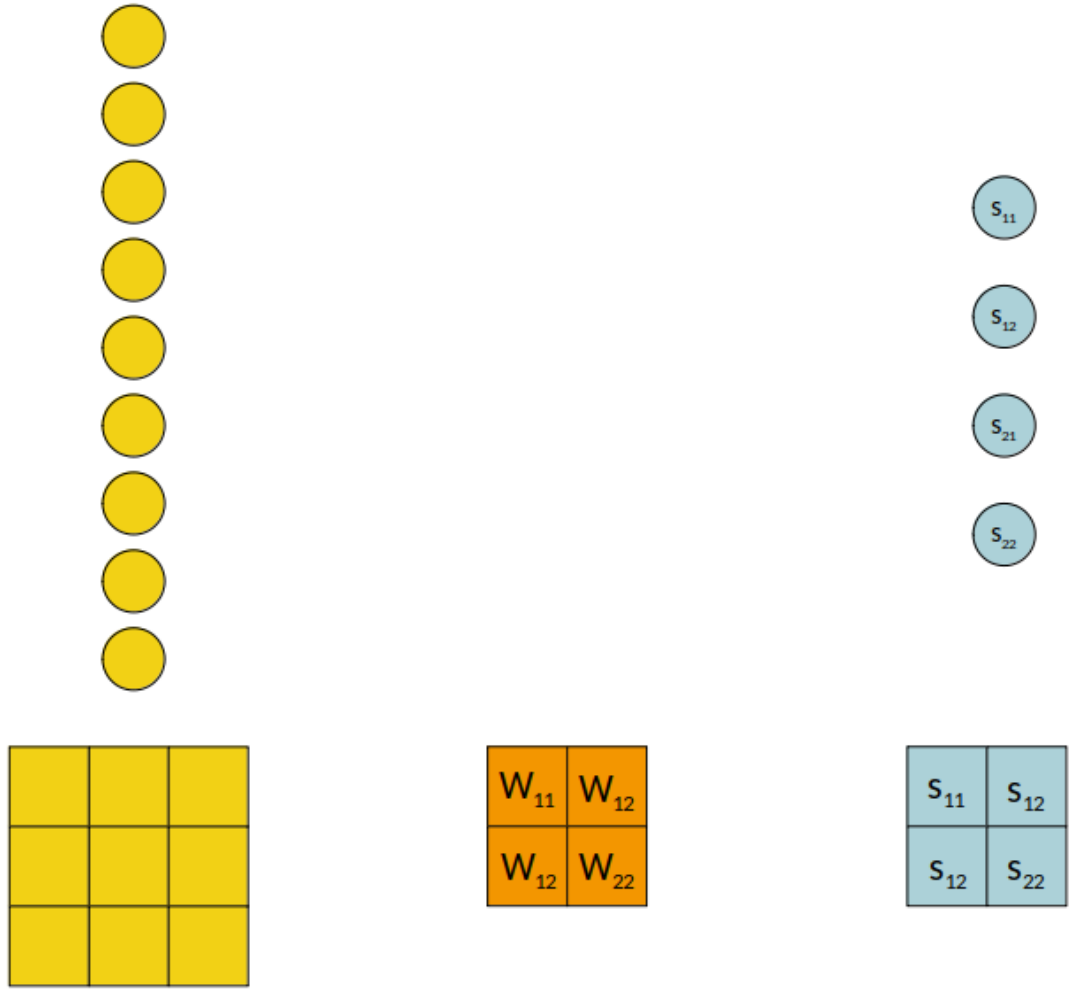


- In total, we obtain 36 connections!

SPARSE INTERACTIONS

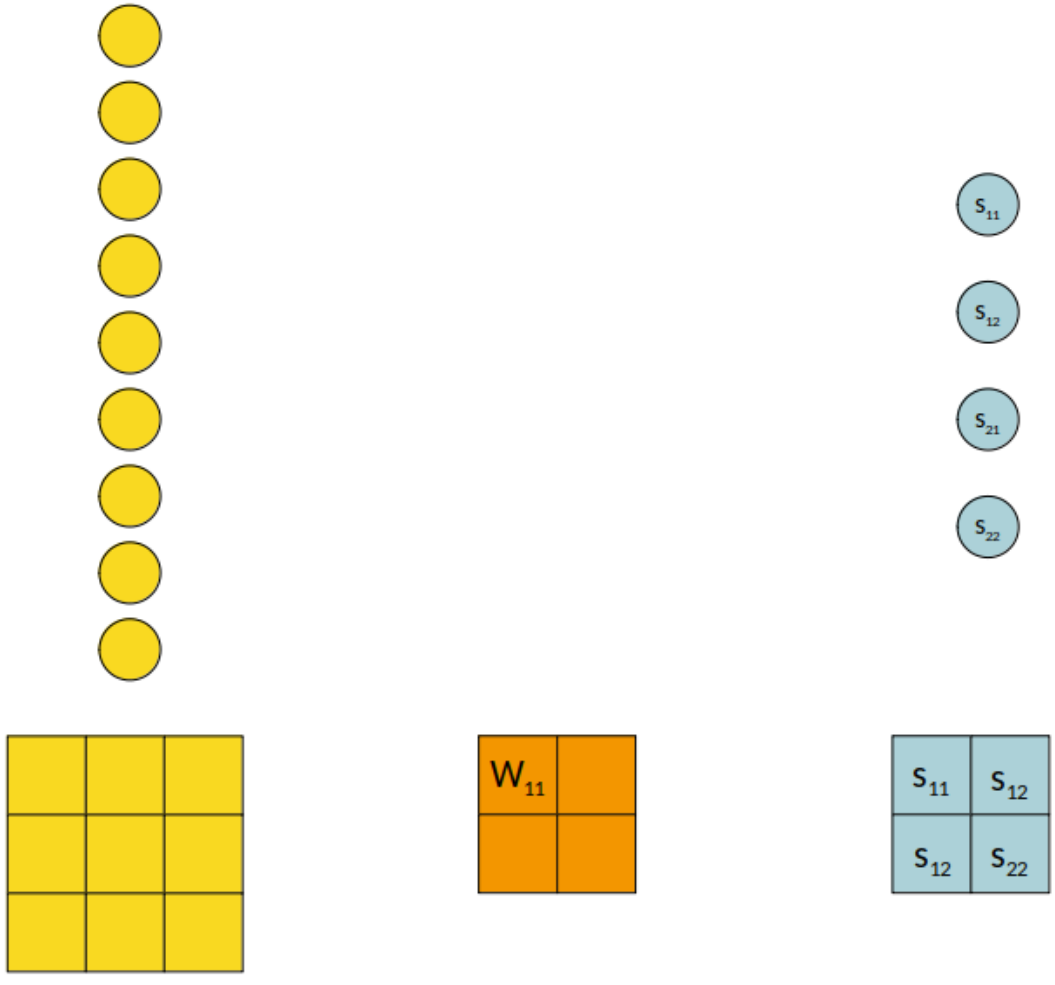
- What does that mean?
  - Our CNN has a **receptive field** of 4 neurons.
  - That means, we apply a “local search” for features.
  - A dense net on the other hand conducts a “global search”.
  - The receptive field of the dense net are 9 neurons.
- When processing images, it is more likely that features occur at specific locations in the input space.
- For example, it is more likely to find the eyes of a human in a certain area, like the face.
  - A CNN only incorporates the surrounding area of the filter into its feature extraction process.
  - The dense architecture on the other hand assumes that every single pixel entry has an influence on the eye, even pixels far away or in the background.

PARAMETER SHARING



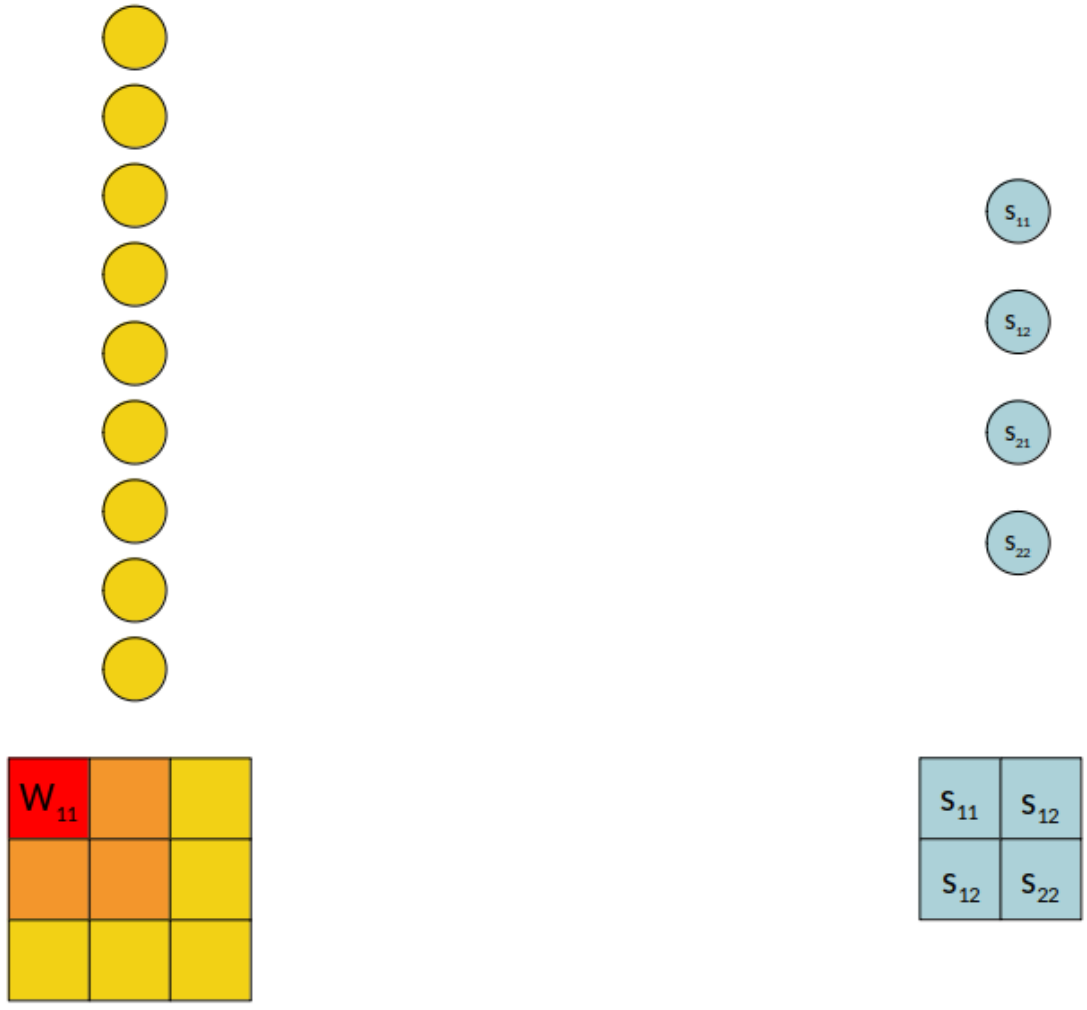
- For the next property we focus on the filter entries.

PARAMETER SHARING



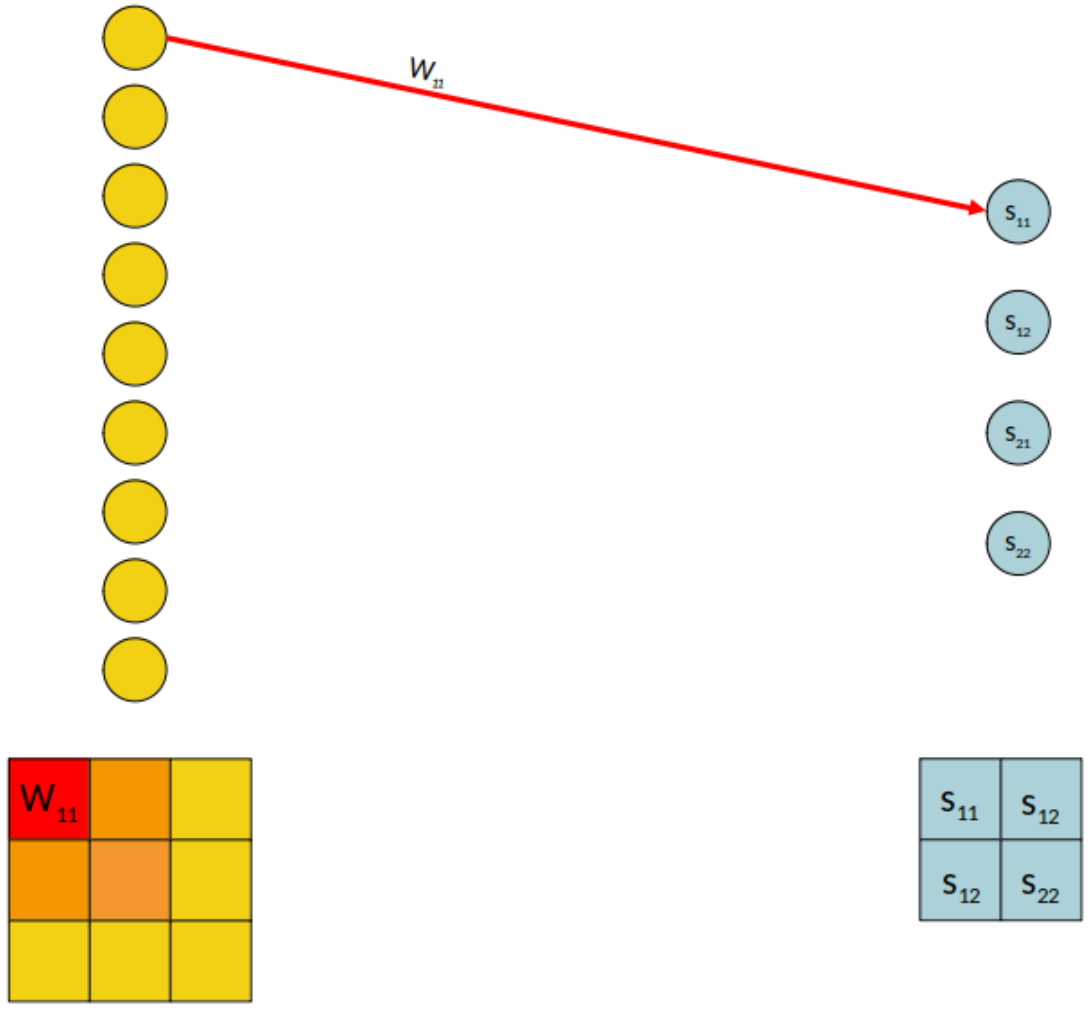
- In particular, we consider weight w_{11}

PARAMETER SHARING



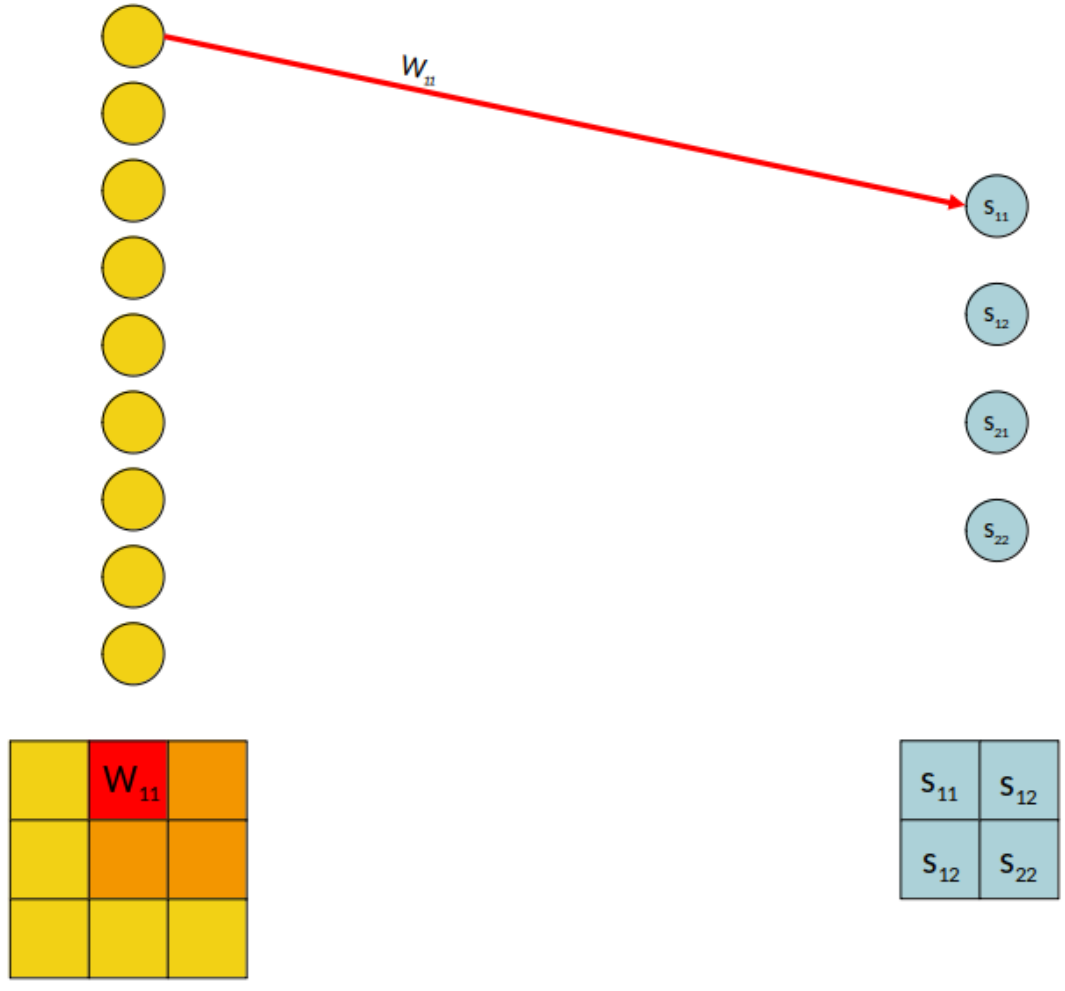
- As we move the filter to the first spatial location..

PARAMETER SHARING



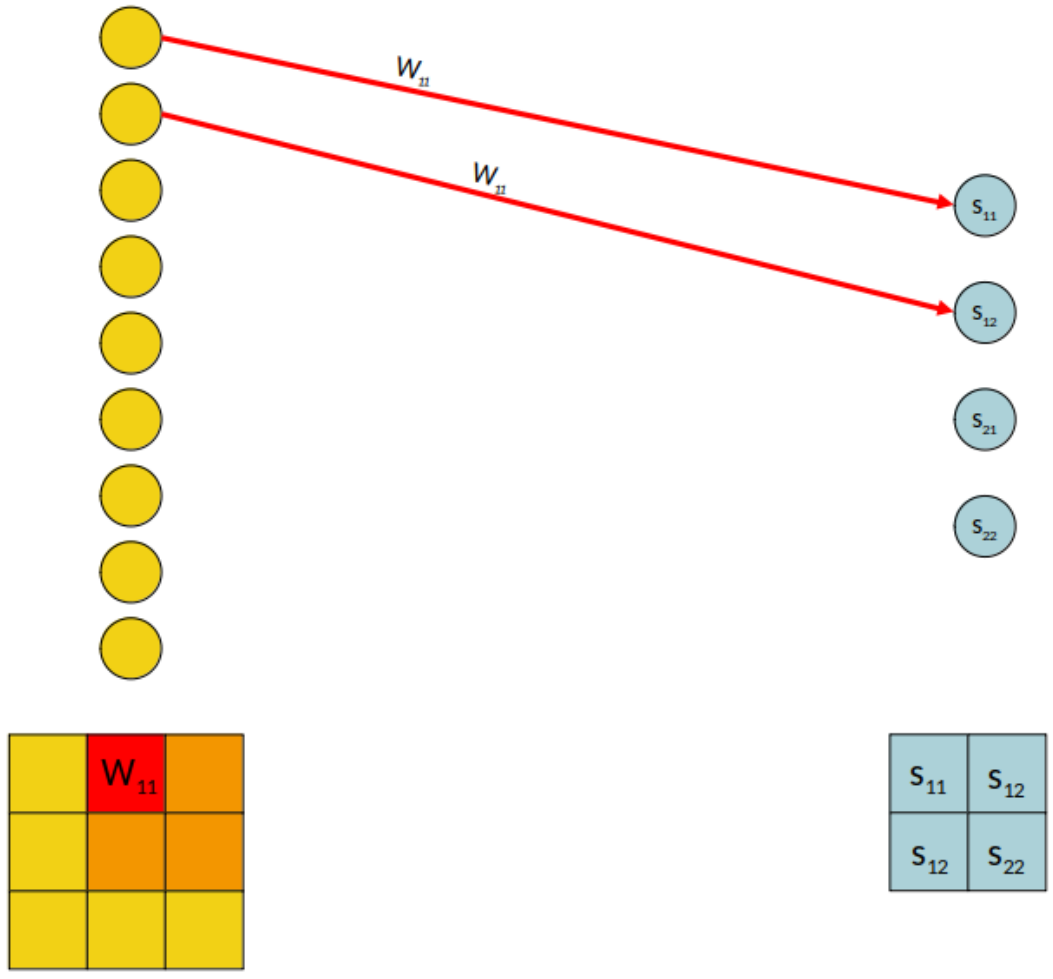
- ...we observe the following connection for weight w_{11}

PARAMETER SHARING



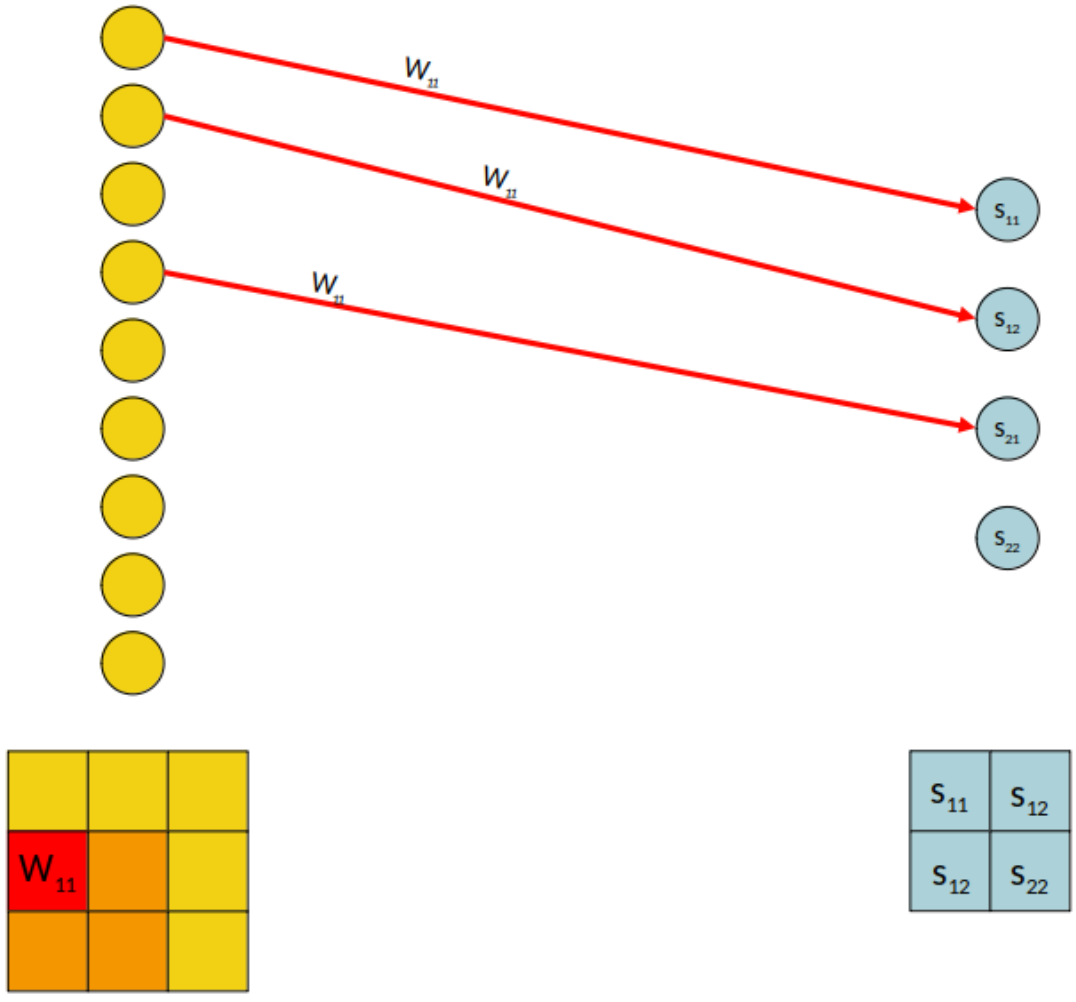
- Moving to the next location...

PARAMETER SHARING



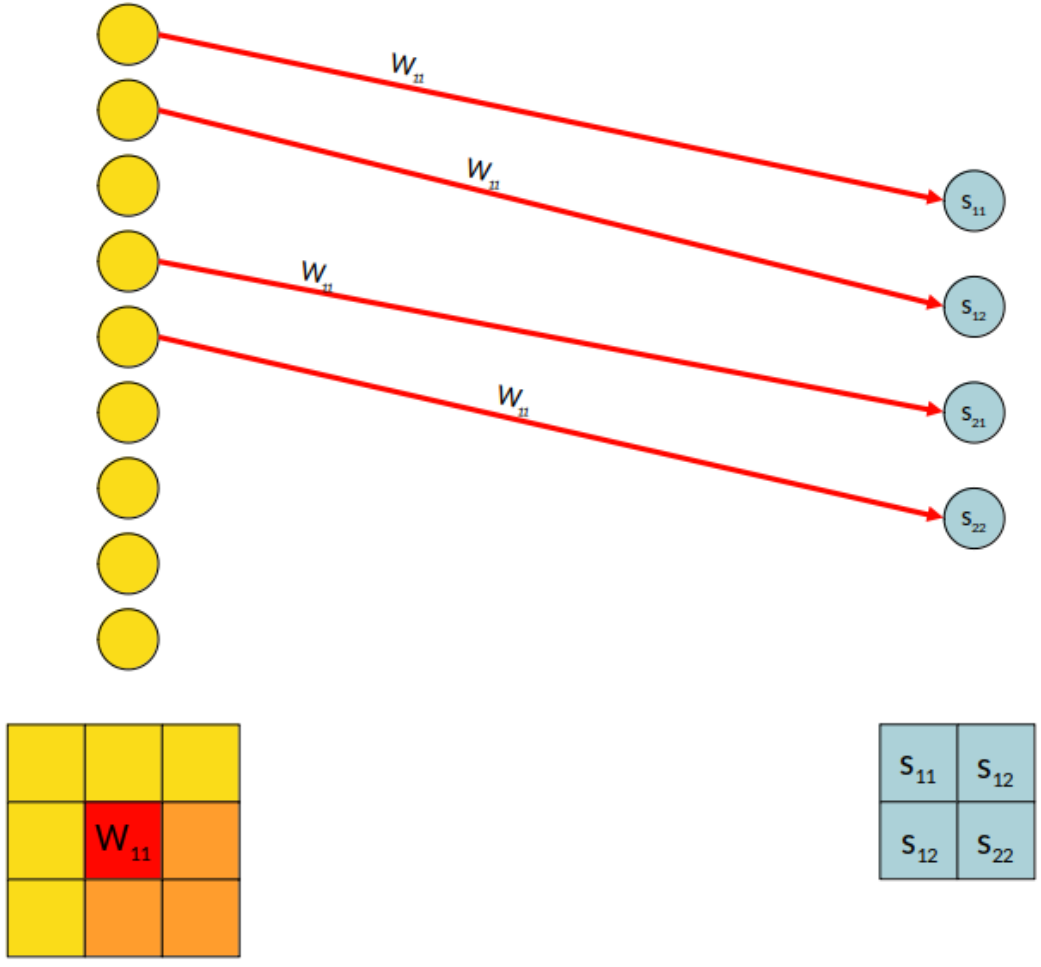
- ...highlights that we use the same weight more than once!

PARAMETER SHARING



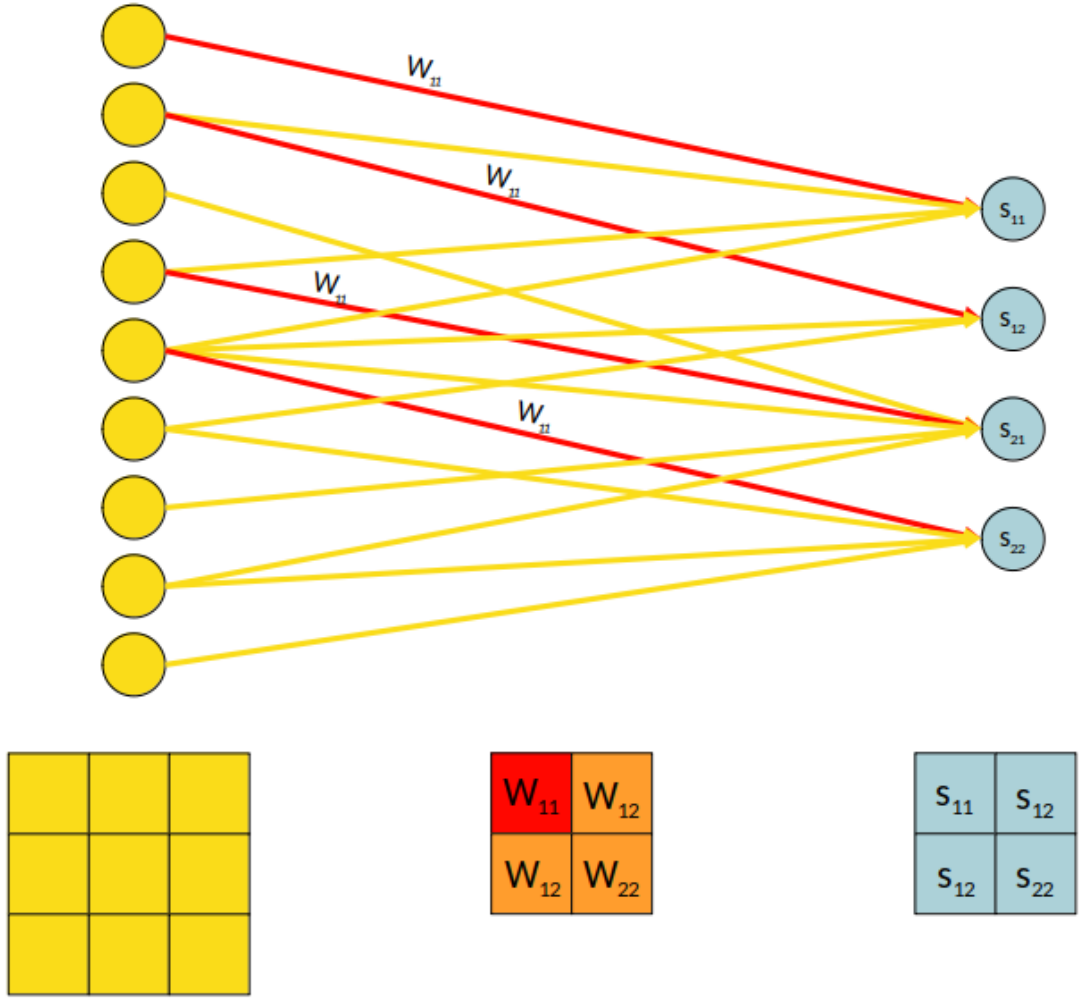
- Even three...

PARAMETER SHARING



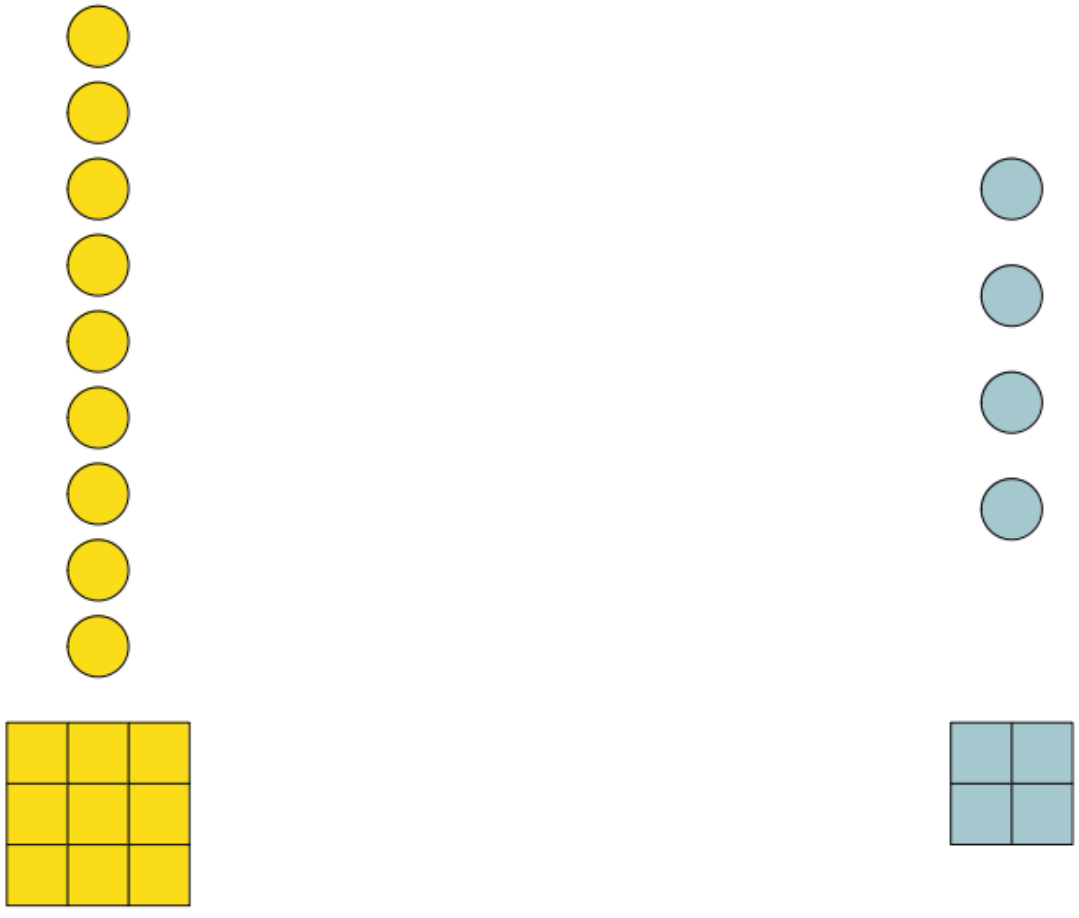
- And in total four times.

PARAMETER SHARING



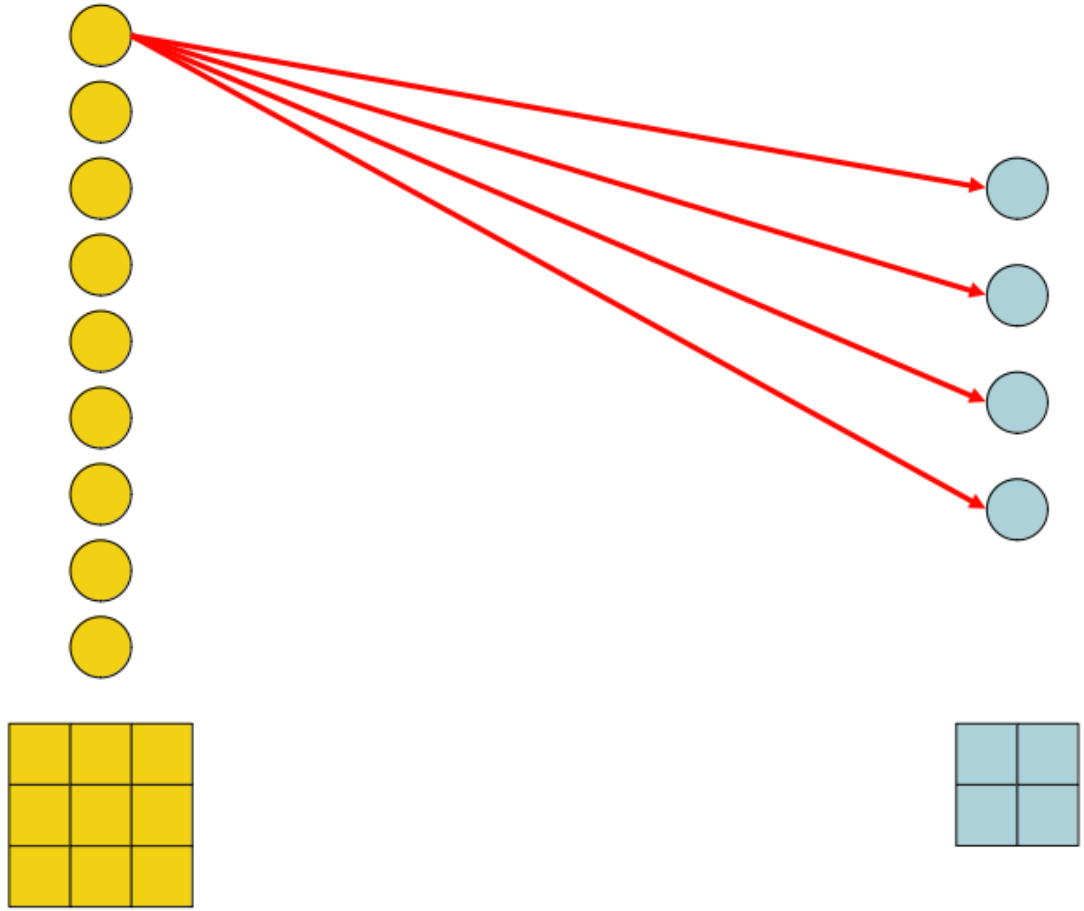
- All together, we have just used four weights.

PARAMETER SHARING



- How many weights does a corresponding dense net use?

PARAMETER SHARING



- $9 \cdot 4 = 36$! That is 9 times more weights!

SPARSE CONNECTIONS AND PARAMETER SHARING

- Why is that good?
- Less parameters drastically reduce memory requirements.
- Faster runtime:
  - For m inputs and n outputs, a fully connected layer requires $m \times n$ parameters and has $\mathcal{O}(m \times n)$ runtime.
  - A convolutional layer has limited connections $k << m$, thus only $k \times n$ parameters and $\mathcal{O}(k \times n)$ runtime.
- Less parameters mean less overfitting and better generalization!

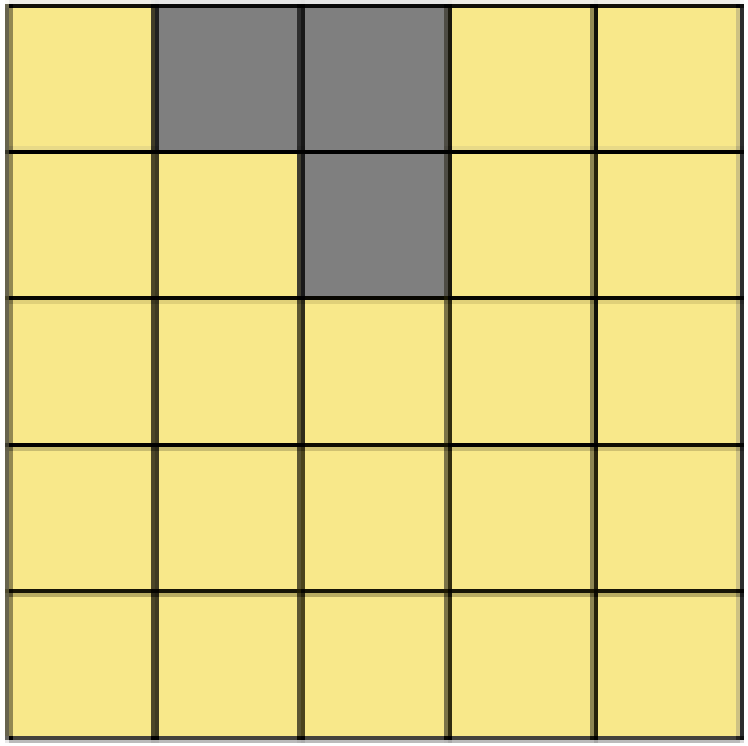
SPARSE CONNECTIONS AND PARAMETER SHARING

- Example: consider a color image with size 100×100.
- Suppose we would like to create one single feature map with a "same padding" (i.e. the hidden layer is of the same size).
  - Choosing a filter with size 5 means that we have a total of $5 \cdot 5 \cdot 3 = 75$ parameters (bias unconsidered).
  - A dense net with the same amount of "neurons" in the hidden layer results in

$$\underbrace{(100^2 \cdot 3)}_{\text{input}} \cdot \underbrace{(100^2)}_{\text{hidden layer}} = 300.000.000$$

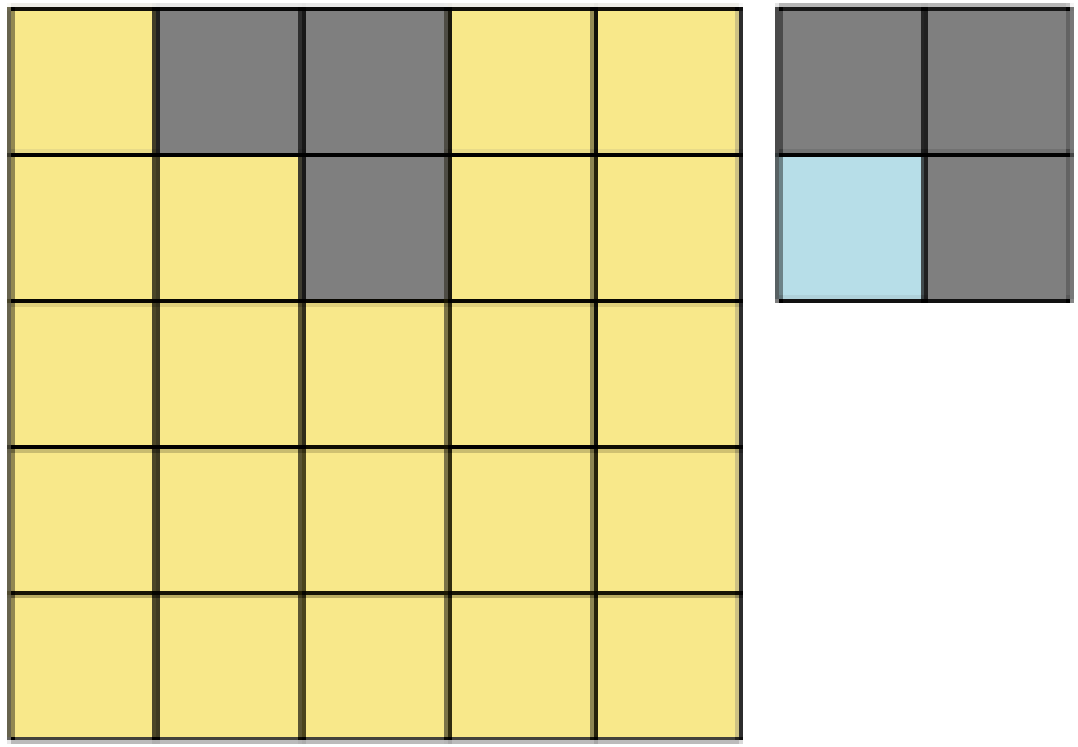
    parameters.
- Note that this was just a fictitious example. In practice we normally do not try to replicate CNN architectures with dense networks.

EQUIVARIANCE TO TRANSLATION



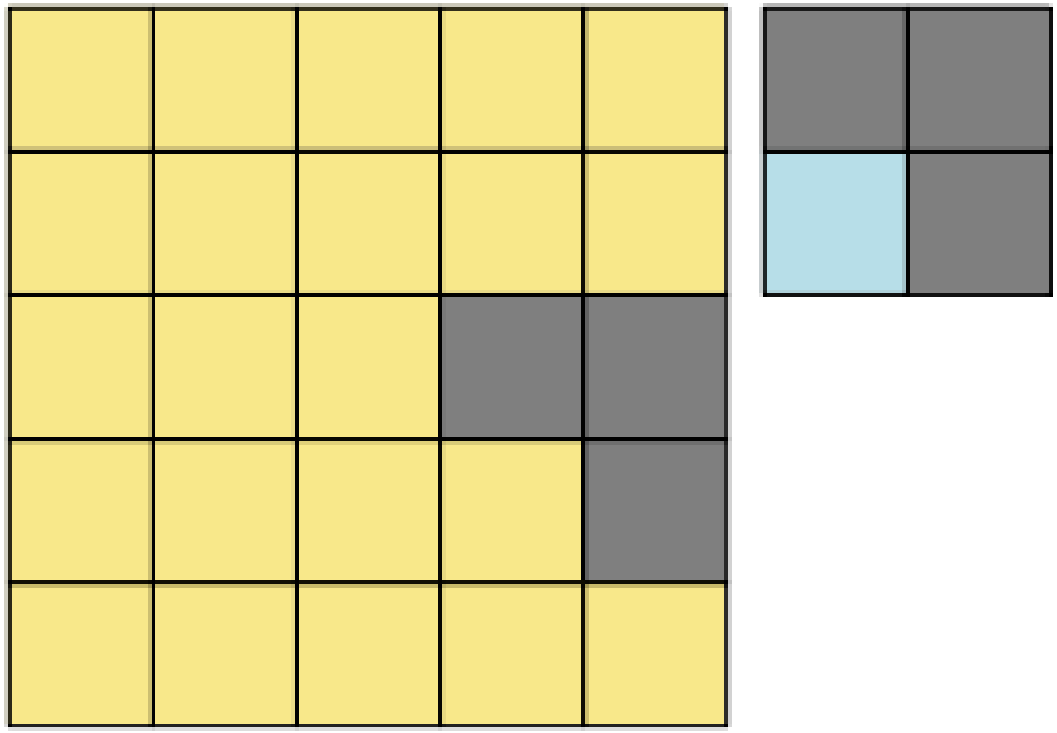
- Think of a specific feature of interest, here highlighted in grey.

EQUIVARIANCE TO TRANSLATION



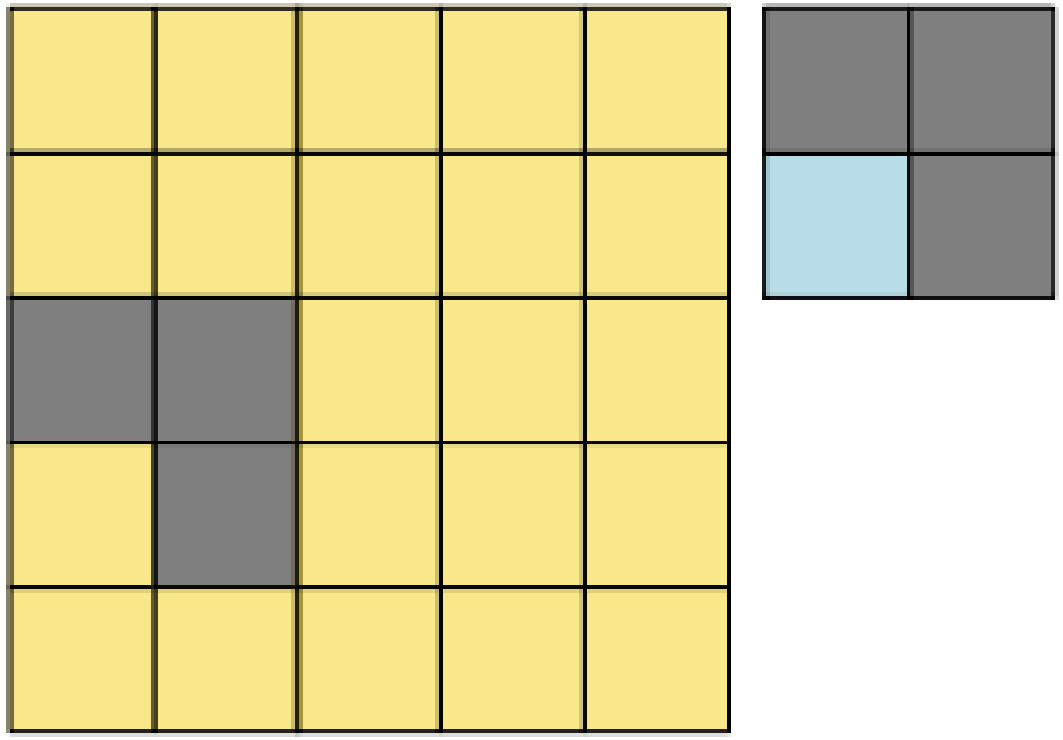
- Furthermore, assume we had a tuned filter looking for exactly that feature.

EQUIVARIANCE TO TRANSLATION



- The filter does not care at what location the feature of interest is located at.

EQUIVARIANCE TO TRANSLATION



- It is literally able to find it anywhere! That property is called **equivariance to translation**.

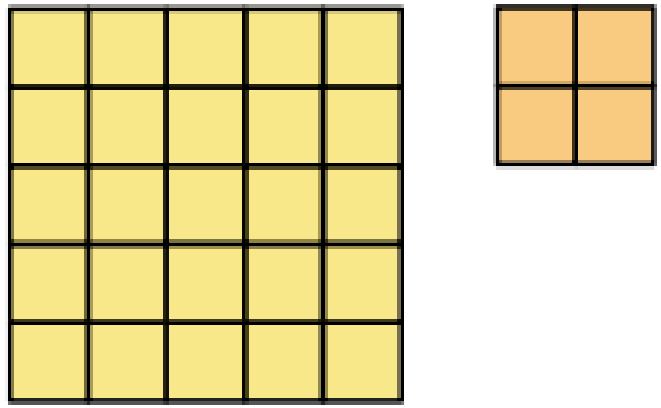
  Note: A function $f(x)$ is equivariant to a function g if $f(g(x)) = g(f(x))$.

NONLINEARITY IN FEATURE MAPS

- As in dense nets, we use activation functions on all feature map entries to introduce nonlinearity in the net.
- Typically rectified linear units (ReLU) are used in CNNs:
  - They reduce the danger of saturating gradients compared to sigmoid activations.
  - They can lead to *sparse activations*, as neurons ≤ 0 are squashed to 0 which increases computational speed.
- As seen in the last chapter, many variants of ReLU (Leaky ReLU, ELU, PReLU, etc.) exist.

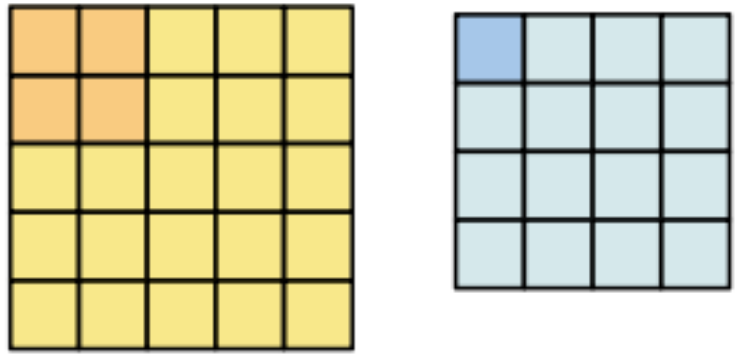
VALID PADDING

Suppose we have an input of size 5×5 and a filter of size 2×2.



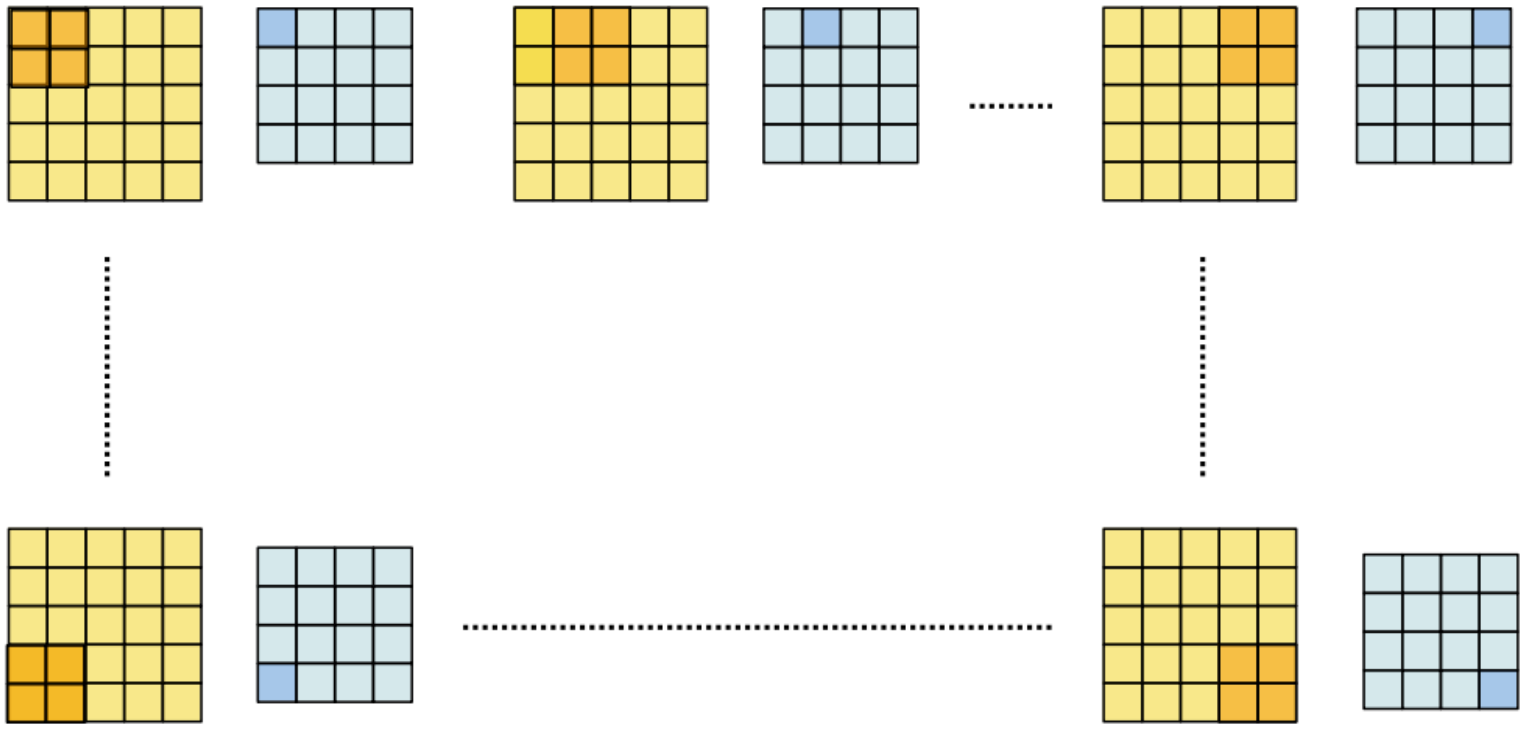
VALID PADDING

The filter is only allowed to move inside of the input space.



VALID PADDING

That will inevitably reduce the output dimensions.

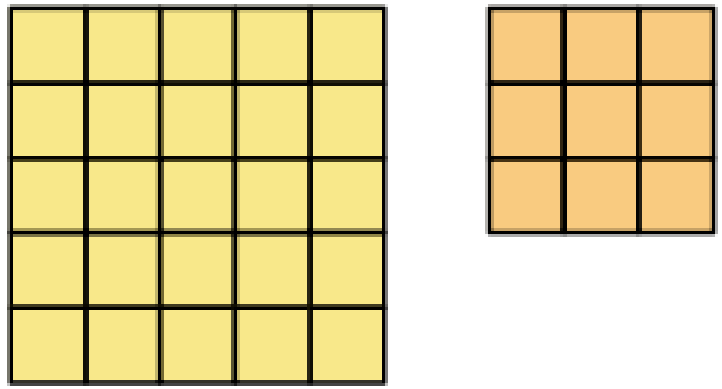


In general, for an input of size i ($\times i$) and filter size k ($\times k$), the size of the output feature map o ($\times o$) claculated by:

$$o = i - k + 1$$

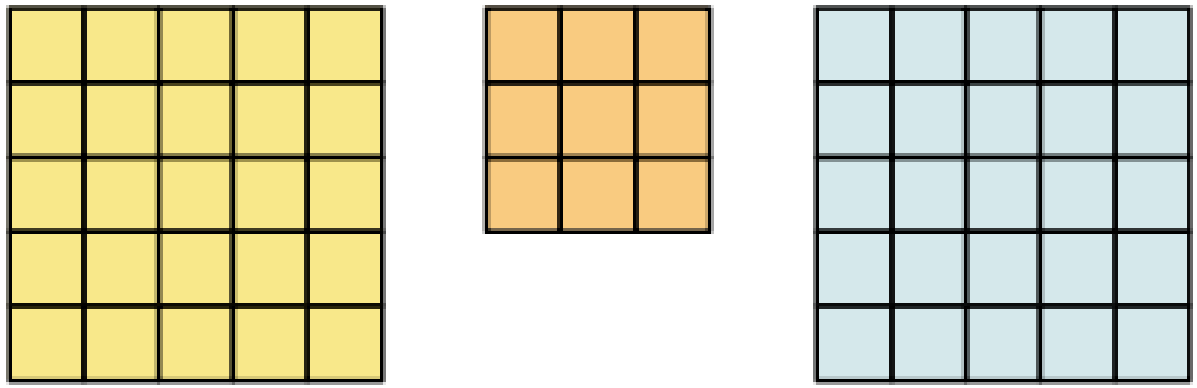
SAME PADDING

Suppose the following situation: an input with dimensions 5×5 and a filter with size 3×3.



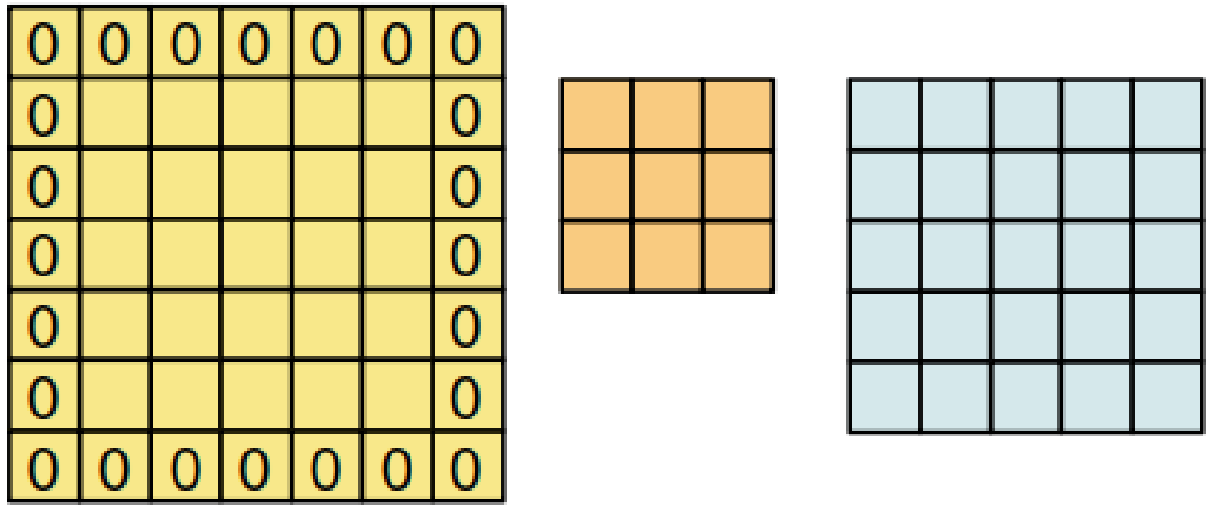
SAME PADDING

We would like to obtain an output with the same dimensions as the input.



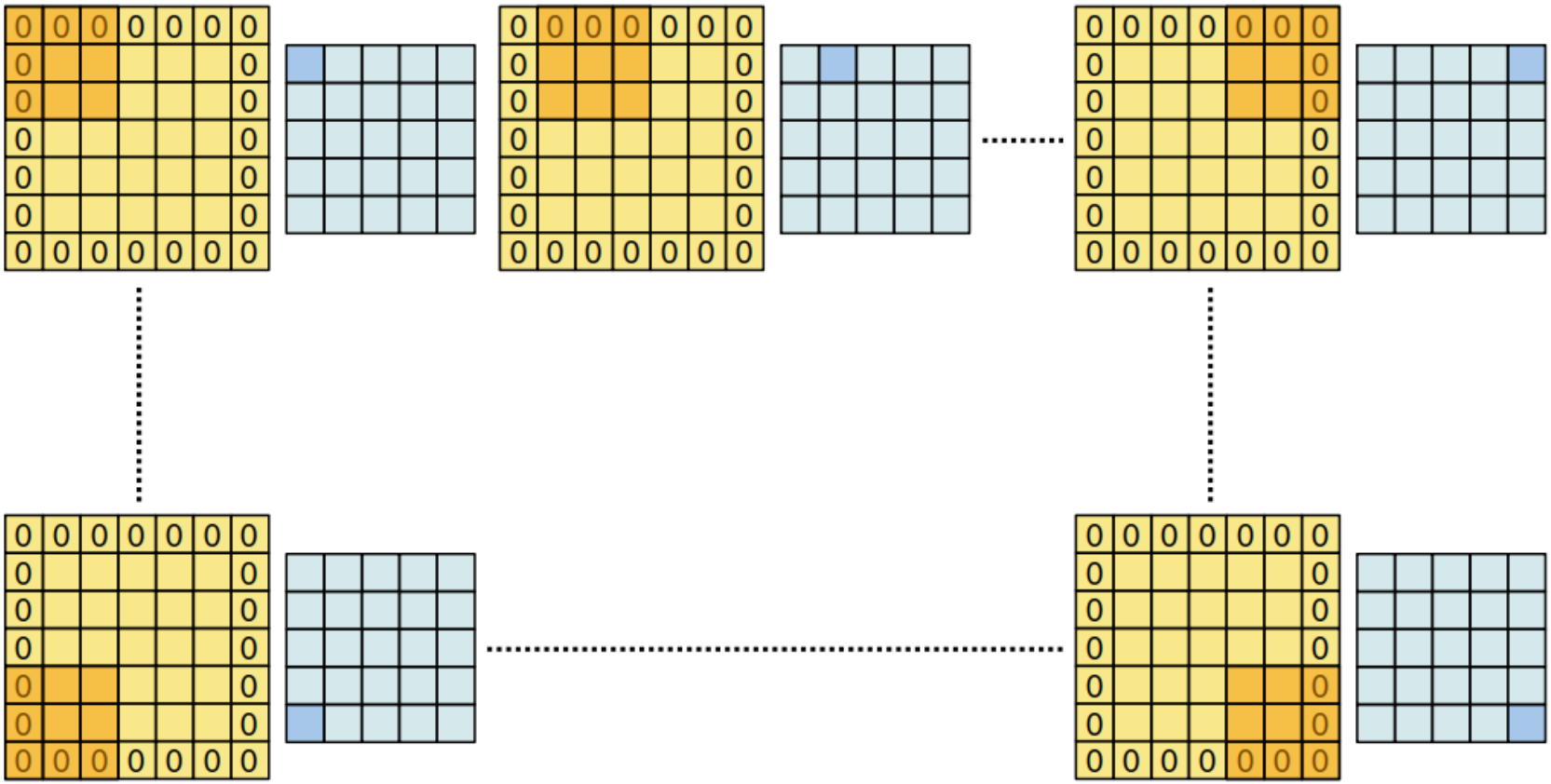
SAME PADDING

Hence, we apply a technique called zero padding. That is to say "pad" zeros around the input:



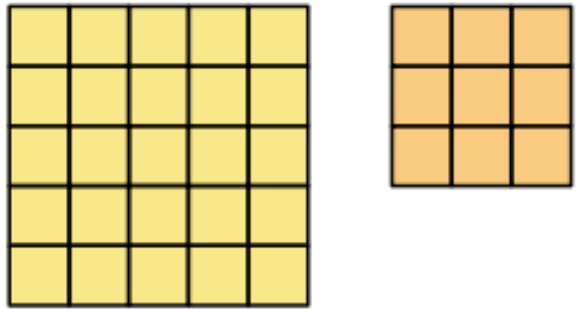
SAME PADDING

That always works! We just have to adjust the zeros according to the input dimensions and filter size (ie. one, two or more rows).



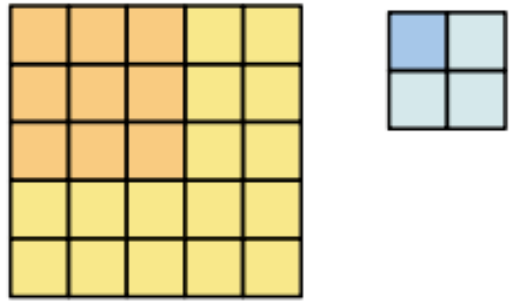
STRIDES

- Stepsize “strides” of our filter (stride = 2 shown below).



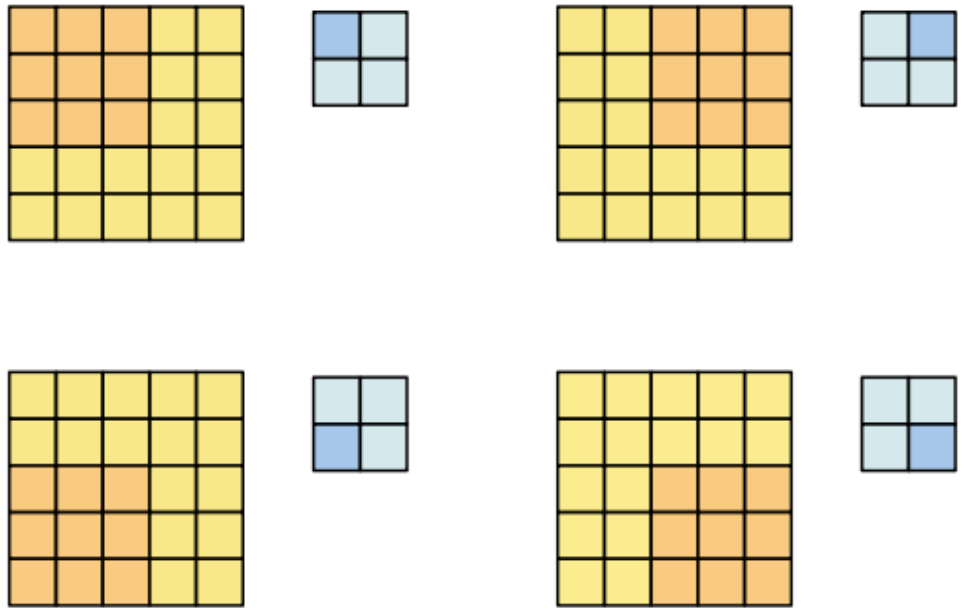
STRIDES

- Stepsize “strides” of our filter (stride = 2 shown below).



STRIDES

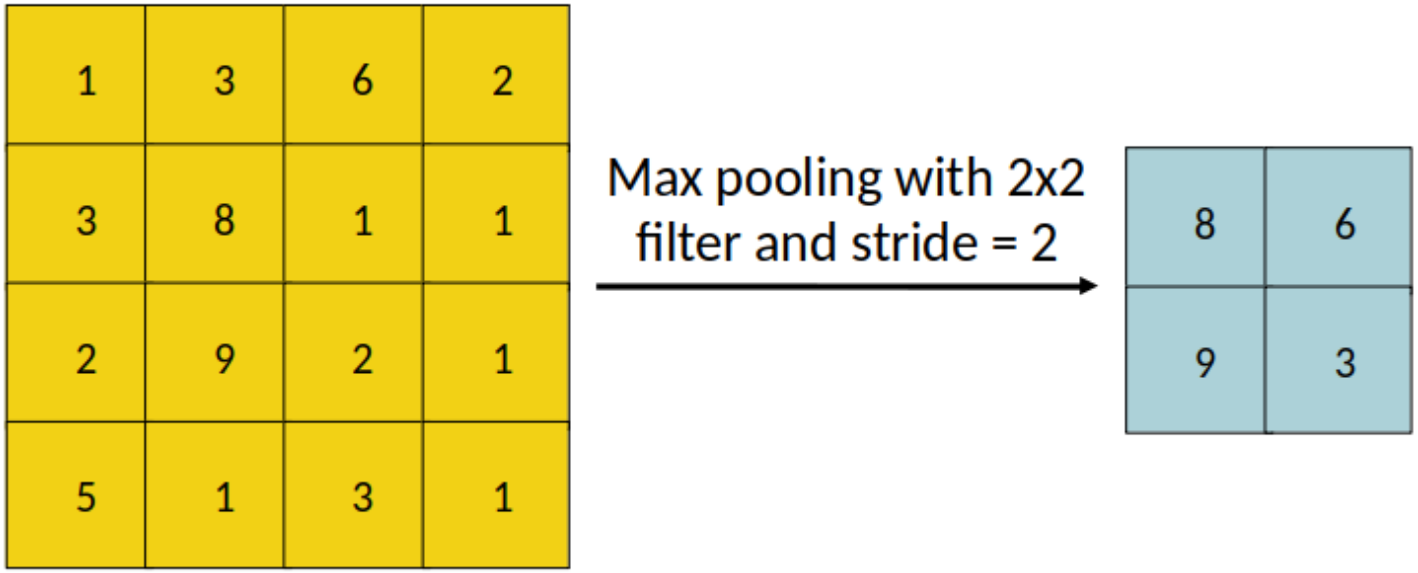
- Stepsize “strides” of our filter (stride = 2 shown below).



In general, when there is no padding, for an input of size i, filter size k and stride *stride*, the size o of the output feature map is:

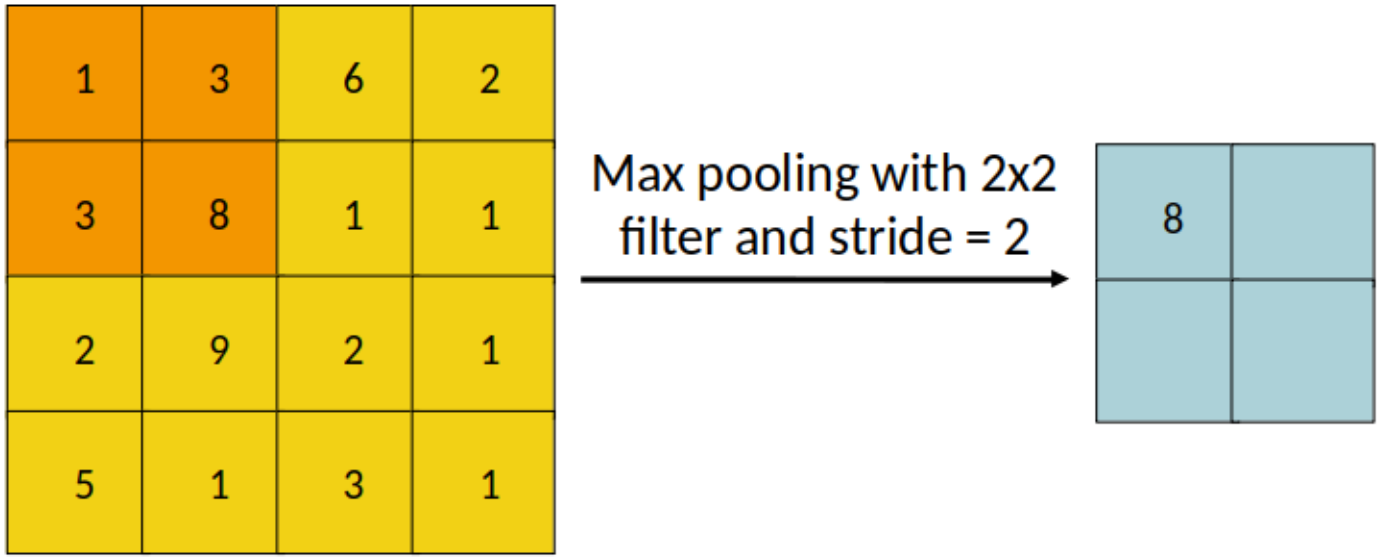
$$o = \left\lfloor \frac{i - k}{stride} \right\rfloor + 1$$

MAX POOLING



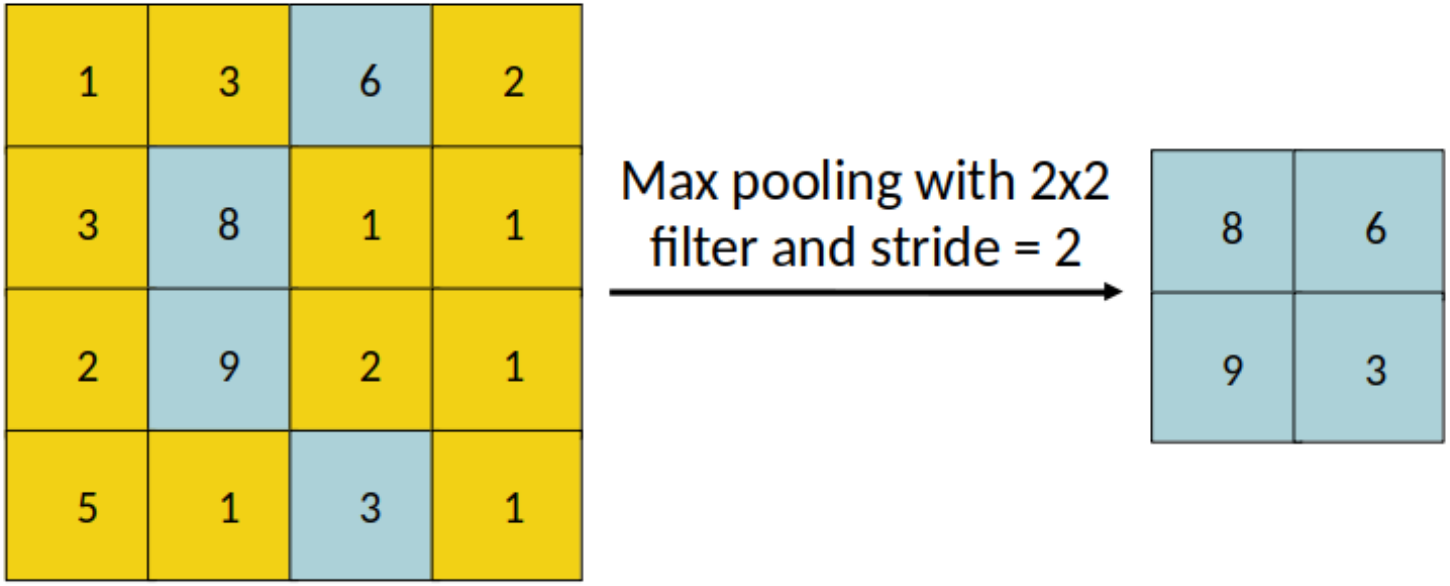
- We've seen how convolutions work, but there is one other operation we need to understand.
- We want to downsample the feature map but optimally lose no information.

MAX POOLING



- Applying the max pooling operation, we simply look for the maximum value at each spatial location.
- That is 8 for the first location.
- Due to the filter of size 2 we have the dimensions of the original feature map and obtain downsampling.

MAX POOLING



- The final pooled feature map has entries 8, 6, 9 and 3.
- Max pooling brings us 2 properties: 1) dimention reduction and 2) spatial invariance.
- Popular pooling functions: max and (weighted) average.

INTRODUCTION TO DEEP LEARNING

HARDWARE FOR DEEP LEARNING

- Deep neural networks require special hardware to be trained efficiently.
- The training is done using **G**raphics **P**rocessing **U**nits (GPUs) and a special programming language called CUDA.
- Training on standard CPUs takes a very long time.

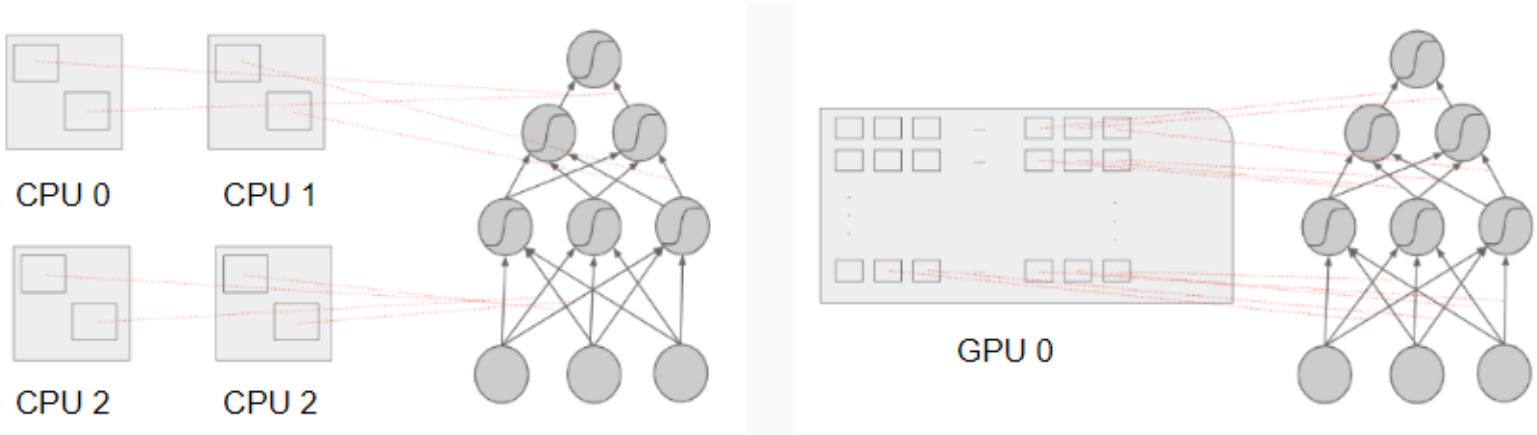


Figure: *Left:* Each CPU can do 2-8 parallel computations. *Right:* A single GPU can do thousands of simple parallel computations.

GRAPHICS PROCESSING UNITS (GPUS)

- Initially developed to accelerate the creation of graphics
- Massively parallel: identical and independent computations for every pixel
- Computer Graphics makes heavy use of linear algebra (just like neural networks)
- Less flexible than CPUs: all threads in a core concurrently execute the same instruction on different data.
- Very fast for CNNs, RNNs need more time
- Popular ones: GTX 1080 Ti, RTX 3080 / 2080 Ti, Titan RTX, Tesla V100 / A100
- Hundreds of threads per core, few thousands cores, around 10 teraFLOPS in single precision, some 10s GBs of memory
- Memory is important - some SOTA architectures do not fit GPUs with <10 GB

TENSOR PROCESSING UNITS (TPUS)

- Specialized and proprietary chip for deep learning developed by Google
- Hundreds of teraFLOPS per chip
- Can be connected together in *pods* of thousands TPUs each (result: hundreds of **peta**FLOPS per pod)
- Not a consumer product! Can be used in the Google Cloud Platform (from 1.35 USD / TPU / hour) or Google Colab (free!)
- Enables DeepMind to make impressive progress : AlphaZero for Chess became world champion after just 4 hours of training concurrently on 5064 TPUs

AND EVERYTHING ELSE...

- With such powerful devices, memory/disk access during training become the bottleneck
  - Nvidia DGX-1: Specialized solution with eight Tesla V100 GPUs, dual Intel Xeon, 512 GB of RAM, 4 SSD disks of 2TB each
- Specialized hardware for on-device inference
  - Example: Neural Engine on the Apple A11 (used for FaceID)
  - Keywords/buzzwords: *Edge computing* and *Federated learning*

DROPOUT

- Idea: reduce overfitting in neural networks by preventing complex co-adaptations of neurons.
- Method: during training, random subsets of the neurons are removed from the network (they are "dropped out").This is done by artificially setting the activations of those neurons to zero.
- Whether a given unit/neuron is dropped out or not is completely independent of the other units.
- If the network has N (input/hidden) units, applying dropout to these units can result in 2^N possible 'subnetworks'.
- Because these subnetworks are derived from the same 'parent' network, many of the weights are shared.
- Dropout can be seen as a form of "model averaging".

DROPOUT: ALGORITHM

- To train with dropout a minibatch-based learning algorithm such as stochastic gradient descent is used.
- For each training case in a minibatch, we randomly sample a binary vector/mask μ with one entry for each input or hidden unit in the network. The entries of μ are sampled independently from each other.
- The probability *p* of sampling a mask value of 0 (dropout) for one unit is a hyperparameter known as the 'dropout rate'.
- A typical value for the dropout rate is 0.2 for input units and 0.5 for hidden units.
- Each unit in the network is multiplied by the corresponding mask value resulting in a $subnet_{\mu}$.
- Forward propagation, backpropagation, and the learning update are run as usual.

DROPOUT: ALGORITHM

Algorithm 1 Training a (parent) neural network with dropout rate p

1: Define parent network and initialize weights
2: **for** each minibatch: **do**
3: **for** each training sample: **do**
4: Draw mask μ using p
5: Compute forward pass for $subnet_\mu$
6: **end for**
7: Update the weights of the (parent) network by performing a gradient descent step with weight decay
8: **end for**

- The derivatives wrt. each parameter are averaged over the training cases in each mini-batch. Any training case which does not use a parameter contributes a gradient of zero for that parameter.

DROPOUT: WEIGHT SCALING

- The weights of the network will be larger than normal because of dropout. Therefore, to obtain a prediction at test time the weights must be first scaled by the chosen dropout rate.
- This means that if a unit (neuron) is retrained with probability p during training, the weight at test time of that unit is multiplied by p.

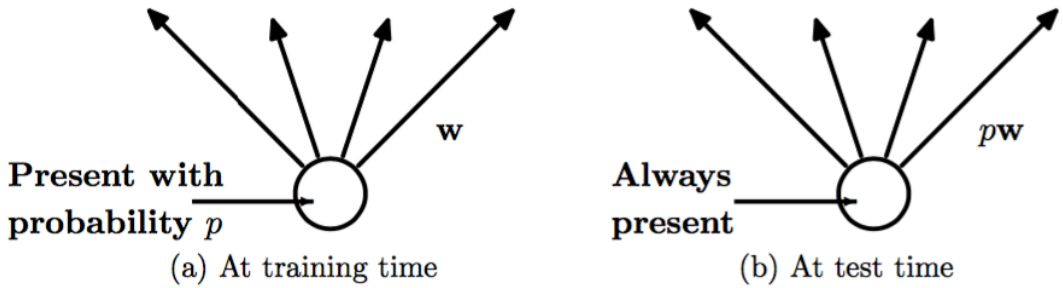


Figure: Training vs. Testing (Srivastava et. al., 2014)

- Weight scaling ensures that the expected total input to a neuron/unit at test time is roughly the same as the expected total input to that unit at train time, even though many of the units at train time were missing on average

DROPOUT: WEIGHT SCALING

- Rescaling of the weights can also be performed at training time instead, after each weight update at the end of the mini-batch. This is sometimes called 'inverse dropout'. Keras and PyTorch deep learning libraries implement dropout in this way.

DATASET AUGMENTATION

- Problem: low generalization because high ratio of

$$\frac{\text{complexity of the model}}{\text{\#train data}}$$

- Idea: artificially increase the train data.
  - Limited data supply $\rightarrow$ create “fake data”!
- Increase variation in inputs **without** changing the labels.
- Application:
  - Image and Object recognition (rotation, scaling, pixel translation, flipping, noise injection, vignetting, color casting, lens distortion, injection of random negatives)
  - Speech recognition (speed augmentation, vocal tract perturbation)

DATASET AUGMENTATION

Figure: Example for data set augmentation (Wu et al., 2015)

DATASET AUGMENTATION

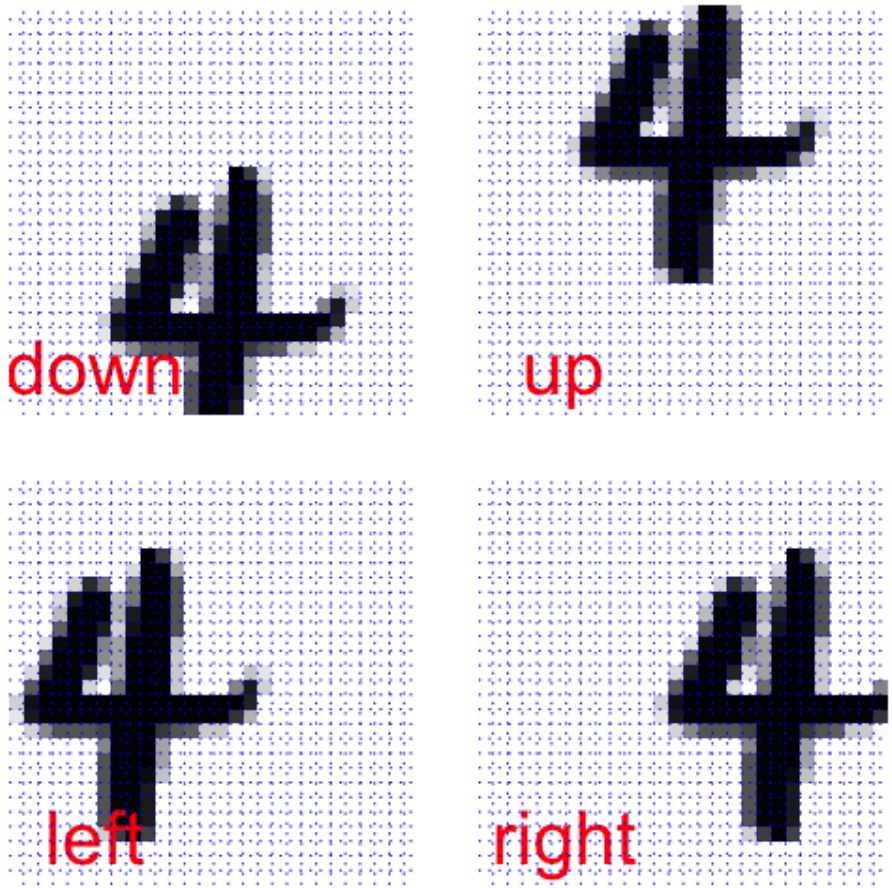


Figure: Example for data set augmentation (Wu et al., 2015)

$\Rightarrow$ careful when rotating digits (6 will become 9 and vice versa)!

MOMENTUM

- While SGD remains a popular optimization strategy, learning with it can sometimes be slow.
- Momentum is designed to accelerate learning, especially when facing high curvature, small but consistent or noisy gradients.
- Momentum accumulates an exponentially decaying moving average of past gradients:

$$\begin{aligned} \boldsymbol{\nu} &\leftarrow \varphi\boldsymbol{\nu} - \alpha \underbrace{\nabla_{\boldsymbol{\theta}} \Big[\frac{1}{m} \sum_i L(y^{(i)}, f(x^{(i)}, \boldsymbol{\theta})) \Big]}_{\mathbf{g}_{\boldsymbol{\theta}}} \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} + \boldsymbol{\nu} \end{aligned}$$

- We introduce a new hyperparameter $\varphi \in [0, 1)$, determining how quickly the contribution of previous gradients decay.
- $\boldsymbol{\nu}$ is called “velocity” and derives from a physical analogy describing how particles move through a parameter space (Newton’s law of motion).

MOMENTUM

- So far the step size was simply the gradient **g** multiplied by the learning rate α.
- Now, the step size depends on how **large** and how **aligned** a sequence of gradients is. The step size grows when many successive gradients point in the same direction.
- Common values for φ are $0.5, 0.9$ and even 0.99.
- Generally, the larger φ is relative to the learning rate α, the more previous gradients affect the current direction.
- A very good website with an in-depth analysis of momentum: `https://distill.pub/2017/momentum/`

SGD WITH MOMENTUM

Algorithm 1 Stochastic gradient descent with momentum

require learning rate α and momentum φ
require initial parameter $\boldsymbol{\theta}$ and initial velocity $\boldsymbol{\nu}$
while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\tilde{x}^{(1)}, \ldots, \tilde{x}^{(m)}\}$
    Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L\left(y^{(i)}, f\left(\tilde{\boldsymbol{x}}^{(i)} \mid \boldsymbol{\theta}\right)\right)$
    Compute velocity update: $\boldsymbol{\nu} \leftarrow \varphi \boldsymbol{\nu} - \alpha \hat{\mathbf{g}}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{\nu}$
end while

SGD WITH MOMENTUM

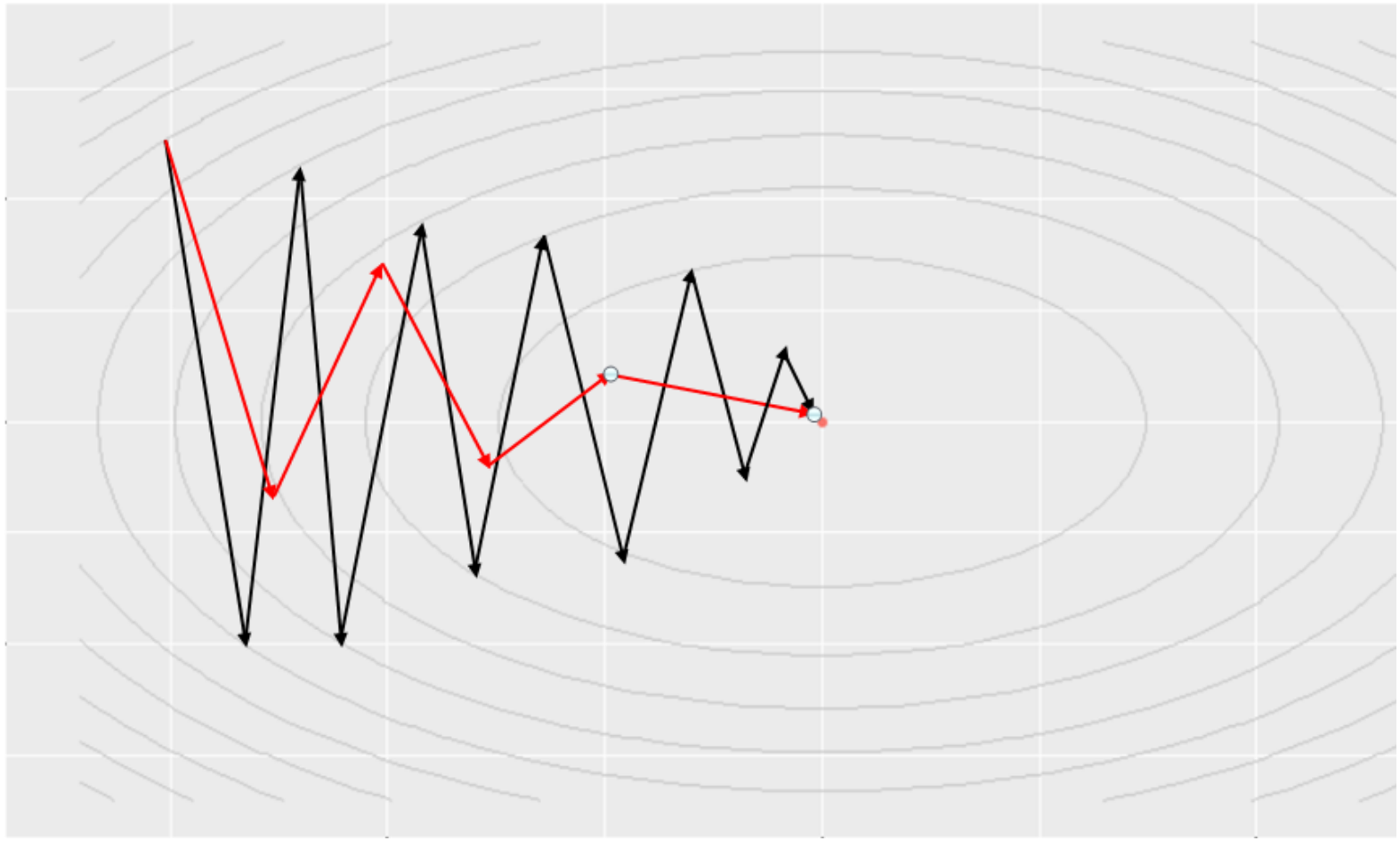
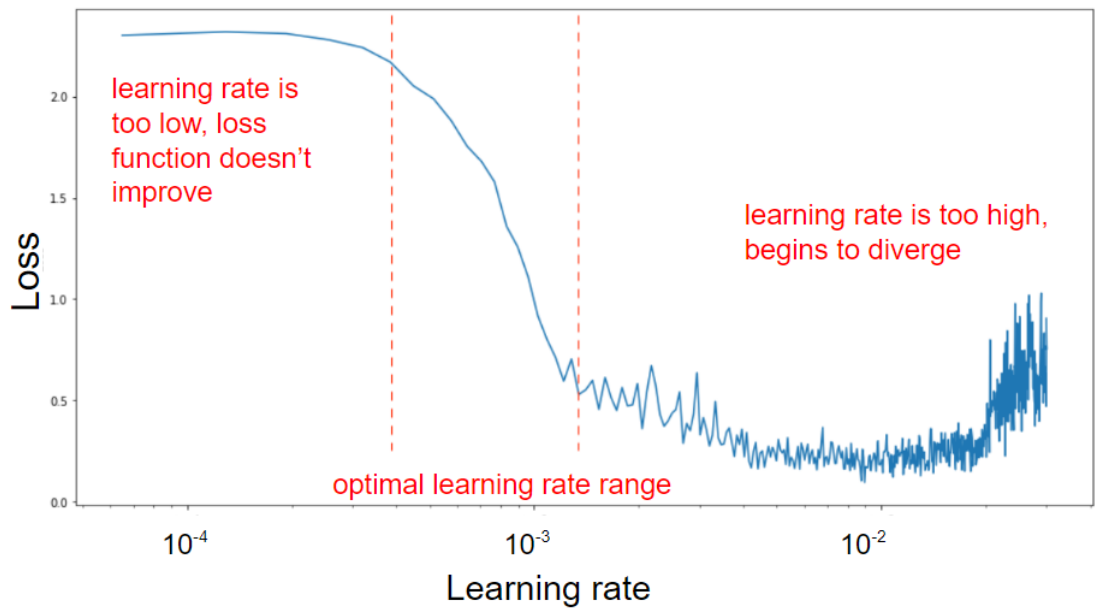


Figure: The contour lines show a quadratic loss function with a poorly conditioned Hessian matrix. The two curves show how standard gradient descent (black) and momentum (red) learn when dealing with ravines. Momentum reduces the oscillation and accelerates the convergence.

LEARNING RATE

- The learning rate is a very important hyperparameter.
- To systematically find a good learning rate, we can start at a very low learning rate and gradually increase it (linearly or exponentially) after each mini-batch.
- We can then plot the learning rate and the training loss for each batch.
- A good learning rate is one that results in a steep decline in the loss.



Credit: jeremyjordan

LEARNING RATE SCHEDULE

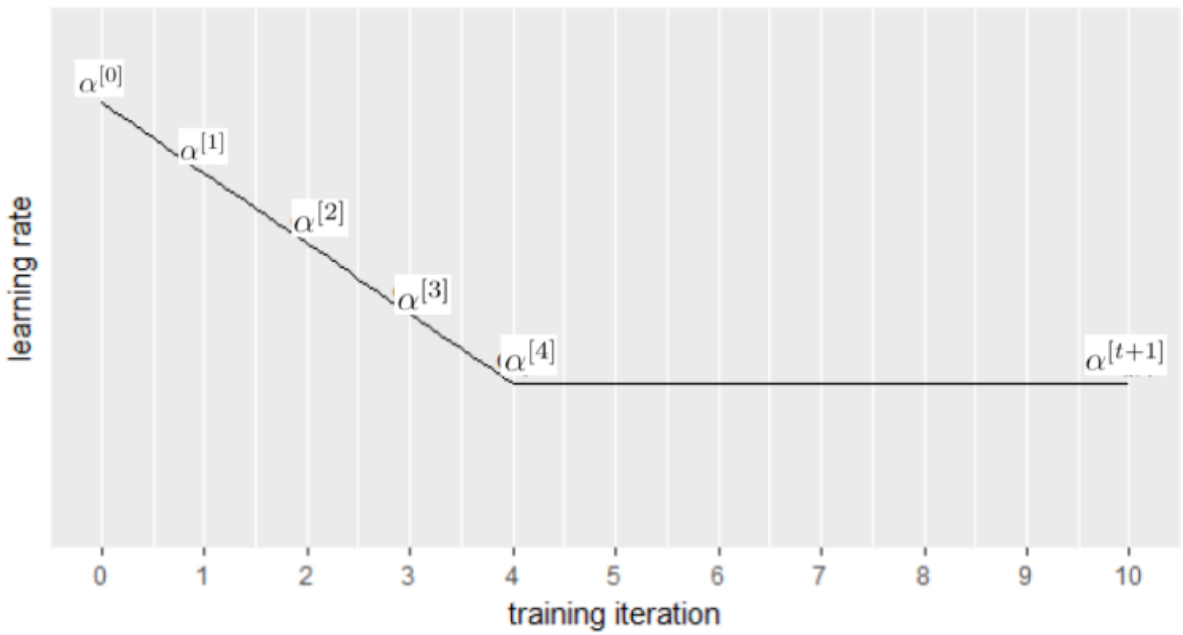
- We would like to force convergence until reaching a local minimum.
- Applying SGD, we have to decrease the learning rate over time, thus $\alpha^{[t]}$ (learning rate at training iteration t).
  - The estimator $\hat{\mathbf{g}}$ is computed based on small batches.
  - Random sampling m training samples introduces noise, that does not vanish even if we find a minimum.
- In practice, a common strategy is to decay the learning rate linearly over time until iteration τ:

$$\alpha^{[t]} = \begin{cases} \left(1 - \frac{t}{\tau}\right)\alpha^{[0]} + \frac{t}{\tau}\alpha^{[\tau]} = t\left(-\frac{\alpha^{[0]}+\alpha^{[\tau]}}{\tau}\right) + \alpha^{[0]} & \text{for } t \leq \tau \\ \alpha^{[\tau]} & \text{for } t > \tau \end{cases}$$

LEARNING RATE SCHEDULE

Example for $\tau = 4$:

iteration t	t/τ	$\alpha^{[t]}$
1	0.25	$\left(1 - \frac{1}{4}\right)\alpha^{[0]} + \frac{1}{4}\alpha^{[\tau]} = \frac{3}{4}\alpha^{[0]} + \frac{1}{4}\alpha^{[\tau]}$
2	0.5	$\frac{2}{4}\alpha^{[0]} + \frac{2}{4}\alpha^{[\tau]}$
3	0.75	$\frac{1}{4}\alpha^{[0]} + \frac{3}{4}\alpha^{[\tau]}$
4	1	$0 + \alpha^{[\tau]}$
...		$\alpha^{[\tau]}$
$t+1$		$\alpha^{[\tau]}$

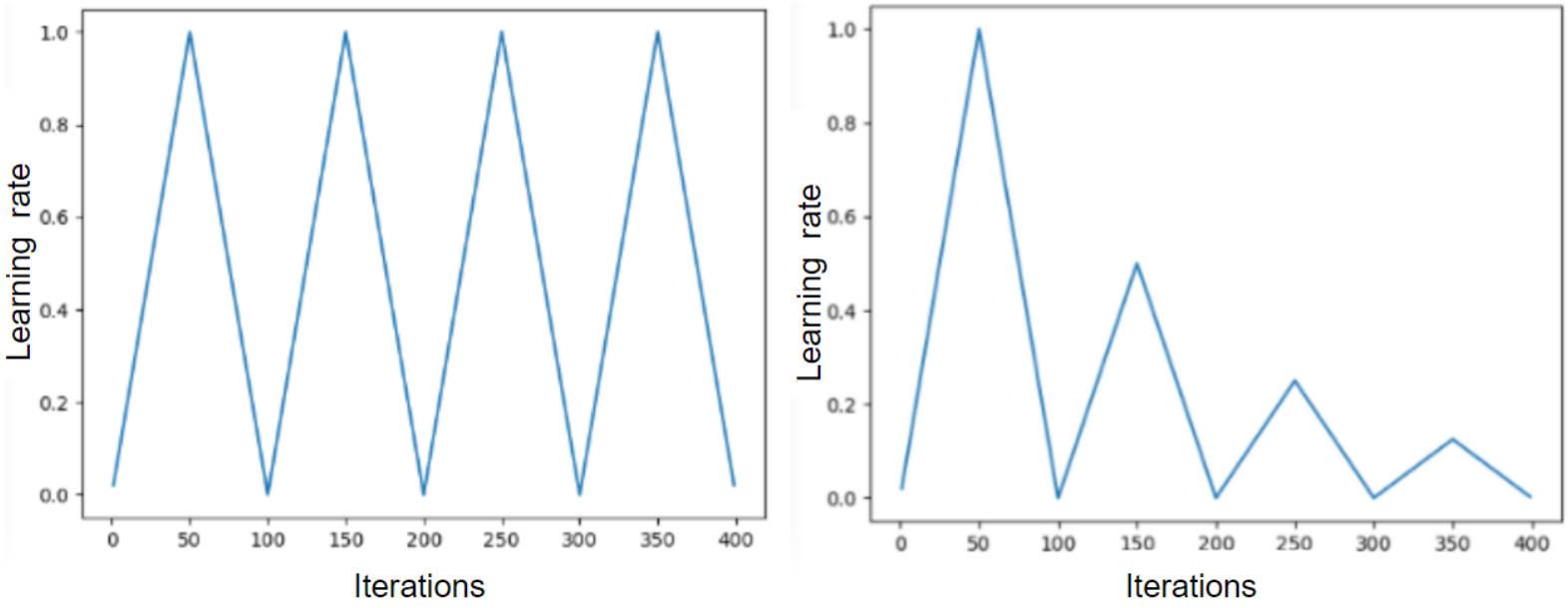


CYCLICAL LEARNING RATES

- Another option is to have a learning rate that periodically varies according to some cyclic function.
- Therefore, if training does not improve the loss anymore (possibly due to saddle points), increasing the learning rate makes it possible to rapidly traverse such regions.
- Recall, saddle points are far more likely than local minima in deep nets.
- Each cycle has a fixed length in terms of the number of iterations.

CYCLICAL LEARNING RATES

- One such cyclical function is the "triangular" function.

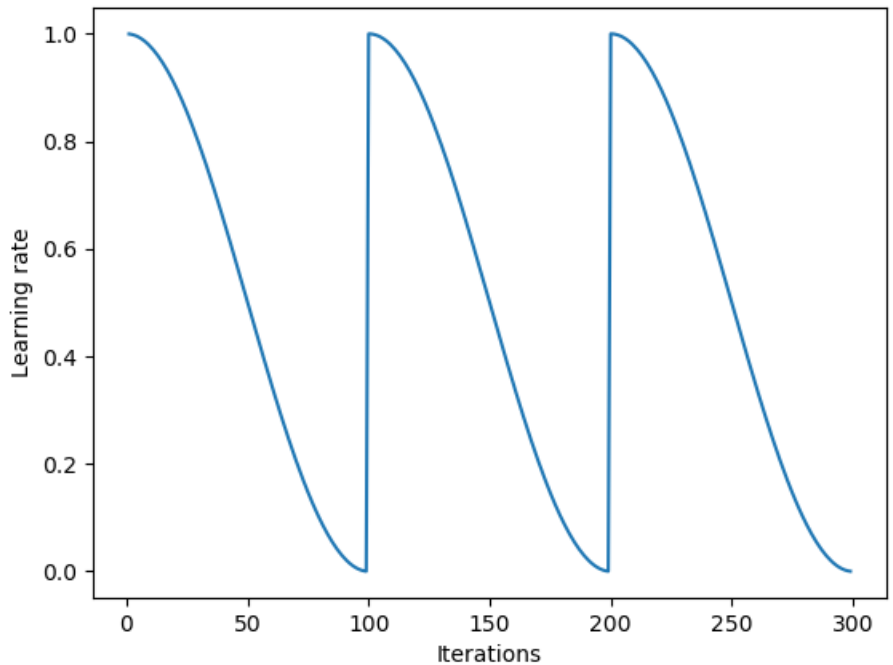


Credit: Hafidz Zulkifli

- In the right image, the range is cut in half after each cycle.

CYCLICAL LEARNING RATES

- Yet another option is to abruptly "restart" the learning rate after a fixed number of iterations.
- Loshchilov et al. (2016) proposed "cosine annealing" (between restarts).



Credit: Hafidz Zulkifli

ADAPTIVE LEARNING RATES

- The learning rate is reliably one of the hyperparameters that is the most difficult to set because it has a significant impact on the models performance.
- Naturally, it might make sense to use a different learning rate for each parameter, and automatically adapt them throughout the training process.

ADAGRAD

- Adagrad adapts the learning rate to the parameters.
- In fact, Adagrad scales learning rates inversely proportional to the square root of the sum of the past squared derivatives.
  - Parameters with large partial derivatives of the loss obtain a rapid decrease in their learning rate.
  - Parameters with small partial derivatives on the other hand obtain a relatively small decrease in their learning rate.
- For that reason, Adagrad might be well suited when dealing with sparse data.
- Goodfellow et al. (2016) say that the accumulation of squared gradients can result in a premature and overly decrease in the learning rate.

ADAGRAD

Algorithm 3 Adagrad

require Global learning rate α
require Initial parameter $\boldsymbol{\theta}$
require Small constant β, perhaps 10^{-7}, for numerical stability
Initialize gradient accumulation variable $\mathbf{r} = \mathbf{0}$
while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\tilde{x}^{(1)}, \ldots, \tilde{x}^{(m)}\}$
    Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L\left(y^{(i)}, f\left(\tilde{\boldsymbol{x}}^{(i)} \mid \boldsymbol{\theta}\right)\right)$
    Accumulate squared gradient $\mathbf{r} \leftarrow \mathbf{r} + \hat{\mathbf{g}} \odot \hat{\mathbf{g}}$
    Compute update: $\nabla\boldsymbol{\theta} = -\frac{\alpha}{\beta + \sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}}$ (division and square root applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \nabla\boldsymbol{\theta}$
end while

- "$\odot$" is called Hadamard or element-wise product.
- Example:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \; B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}, \; \text{then } A \odot B = \begin{bmatrix} 1 \cdot 5 & 2 \cdot 6 \\ 3 \cdot 7 & 4 \cdot 8 \end{bmatrix}$$

RMSPROP

- RMSprop is a modification of Adagrad.
- It's intention is to resolve Adagrad's radically diminishing learning rates.
- The gradient accumulation is replaced by an exponentially weighted moving average.
- Theoretically, that leads to performance gains in non-convex scenarios.
- Empirically, RMSProp is a very effective optimization algorithm. Particularly, it is employed routinely by deep learning practitioners.

RMSPROP

Algorithm 4 RMSProp

require Global learning rate α and decay rate $\rho \in [0, 1)$
require Initial parameter $\boldsymbol{\theta}$
require Small constant β, perhaps 10^{-6}, for numerical stability
Initialize gradient accumulation variable $\mathbf{r} = \mathbf{0}$
while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\tilde{x}^{(1)}, \ldots, \tilde{x}^{(m)}\}$
    Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow \frac{1}{m} \nabla_\theta \sum_i L\left(y^{(i)}, f\left(\tilde{\boldsymbol{x}}^{(i)} \mid \boldsymbol{\theta}\right)\right)$
    Accumulate squared gradient $\mathbf{r} \leftarrow \rho \mathbf{r} + (1 - \rho)\hat{\mathbf{g}} \odot \hat{\mathbf{g}}$
    Compute update: $\nabla\boldsymbol{\theta} = -\frac{\alpha}{\beta + \sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \nabla\boldsymbol{\theta}$
end while

ADAM

- Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter.
- Adam uses the first and the second moments of the gradients.
  - Adam keeps an exponentially decaying average of past gradients (first moment).
  - Like RMSProp it stores an exponentially decaying average of past squared gradients (second moment).
  - Thus, it can be seen as a combination of RMSProp and momentum.
- Basically Adam uses the combined averages of previous gradients at different moments to give it more “persuasive power” to adaptively update the parameters.

ADAM

Algorithm 5 Adam

require Step size α (suggested default: 0.001)
require Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$ (suggested defaults: 0.9 and 0.999 respectively)
require Small constant β (suggested default 10^{-8})
require Initial parameters $\boldsymbol{\theta}$
Initialize time step $t = 0$
Initialize 1st and 2nd moment variables $\mathbf{s}^{[0]} = 0, \mathbf{r}^{[0]} = 0$
while stopping criterion not met **do**
    $t \leftarrow t + 1$
    Sample a minibatch of m examples from the training set $\{\tilde{x}^{(1)}, \ldots, \tilde{x}^{(m)}\}$
    Compute gradient estimate: $\hat{\mathbf{g}}^{[t]} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L\left(y^{(i)}, f\left(\tilde{\boldsymbol{x}}^{(i)} \mid \boldsymbol{\theta}\right)\right)$
    Update biased first moment estimate: $\mathbf{s}^{[t]} \leftarrow \rho_1 \mathbf{s}^{[t-1]} + (1 - \rho_1)\hat{\mathbf{g}}^{[t]}$
    Update biased second moment estimate: $\mathbf{r}^{[t]} \leftarrow \rho_2 \mathbf{r}^{[t-1]} + (1 - \rho_2)\hat{\mathbf{g}}^{[t]} \odot \hat{\mathbf{g}}^{[t]}$
    Correct bias in first moment: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}^{[t]}}{1-\rho_1^t}$
    Correct bias in second moment: $\hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}^{[t]}}{1-\rho_2^t}$
    Compute update: $\nabla\boldsymbol{\theta} = -\alpha \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}}+\beta}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \nabla\boldsymbol{\theta}$
end while

BATCH NORMALIZATION

- Batch Normalization (BatchNorm) is an extremely popular technique that improves the training speed and stability of deep neural nets.
- It is an extra component that can be placed between each layer of the neural network.
- It works by changing the "distribution" of activations at each hidden layer of the network.
- We know that it is sometimes beneficial to normalize the inputs to a learning algorithm by shifting and scaling all the features so that they have 0 mean and unit variance.
- BatchNorm applies a similar transformation to the activations of the hidden layers (with a couple of additional tricks).

BATCH NORMALIZATION

- For a hidden layer with neurons z_j, $j = 1, \ldots, J$, BatchNorm is applied to each z_j by considering the activations of z_j **over a given minibatch** of inputs.
- Let $z_j^{(i)}$ denote the activation of z_j for input $x^{(i)}$ in the minibatch (of size m).
- The mean and variance of the activations are

$$\mu_j = \frac{1}{m} \sum_i^m z_j^{(i)}$$

$$\sigma_j^2 = \frac{1}{m} \sum_i^m (z_j^{(i)} - \mu_j)^2$$

- Each $z_j^{(i)}$ is then normalized

$$\tilde{z}_j^{(i)} = \frac{z_j^{(i)} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

where a small constant, ϵ, is added for numerical stability.

BATCH NORMALIZATION

- It may not be desirable to normalize the activations in such a rigid way because potentially useful information can be lost in the process.
- Therefore, we commonly let the training algorithm decide the "right amount" of normalization by allowing it to re-shift and re-scale $\tilde{z}_j^{(i)}$ to arrive at the batch normalized activation $\hat{z}_j^{(i)}$:

$$\hat{z}_j^{(i)} = \gamma_j \tilde{z}_j^{(i)} + \beta_j$$

- γ_j and β_j are learnable parameters that are also tweaked by backpropogation.
- $\hat{z}_j^{(i)}$ then becomes the input to the next layer.
- Note: The algorithm is free to scale and shift each $\tilde{z}_j^{(i)}$ back to its original (unnormalized) value.

BATCH NORMALIZATION: ILLUSTRATION

- Recall: $z_j = \sigma(W_j^T x + b_j)$
- So far, we have applied batch-norm to the activation z_j. It is possible (and more common) to apply batch norm to $W_j^T x + b_j$ *before* passing it to the nonlinear activation σ.

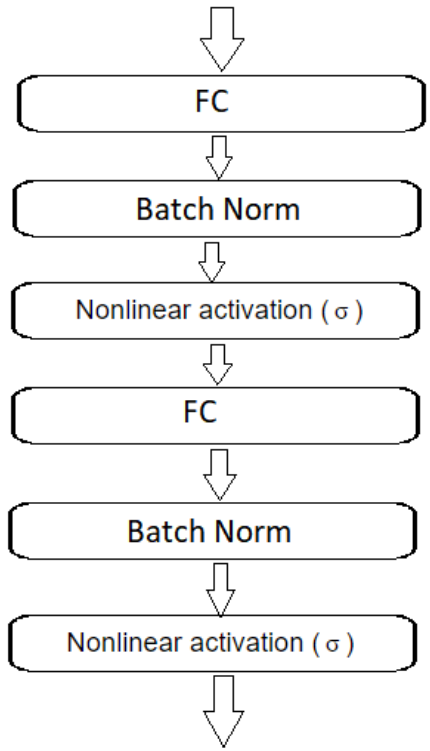
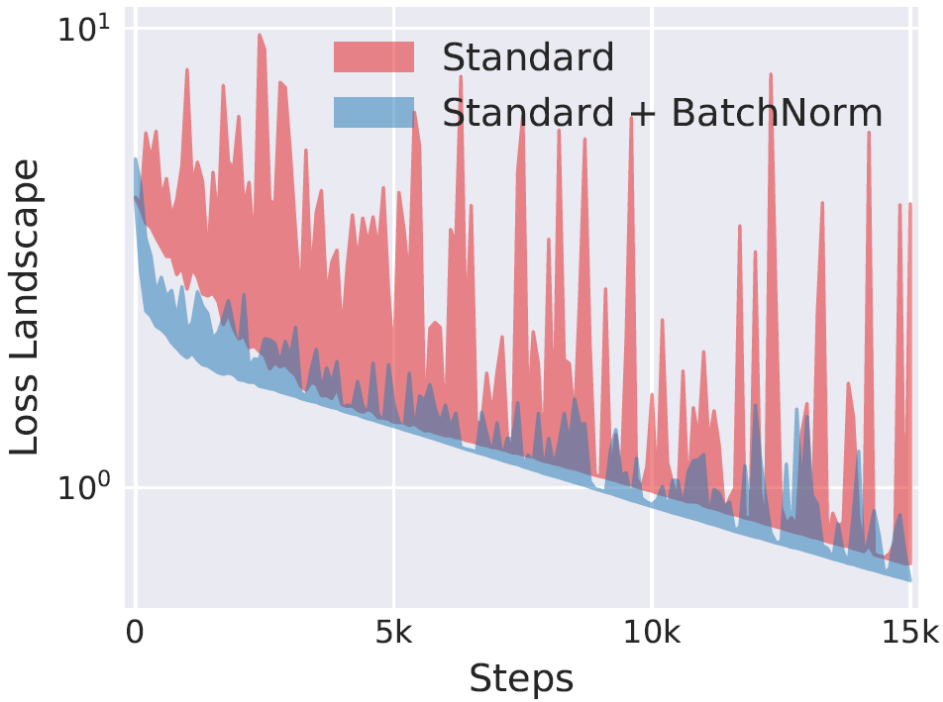


Figure: FC = Fully Connected layer. BatchNorm is applied *before* the nonlinear activation function.

BATCH NORMALIZATION

- The key impact of BatchNorm on the training process is this: It reparametrizes the underlying optimization problem to **make its landscape significantly more smooth**.
- One aspect of this is that the loss changes at a smaller rate and the magnitudes of the gradients are also smaller (see Santurkar et al. 2018).



BATCH NORMALIZATION: PREDICTION

- Once the network has been trained, how can we generate a prediction for a single input (either at test time or in production)?
- One option is to feed the entire training set to the (trained) network and compute the means and standard deviations.
- More commonly, during training, an exponentially weighted running average of each of these statistics over the minibatches is maintained.
- The learned γ and β parameters are then used (in conjunction with the running averages) to generate the output.

HIDDEN ACTIVATIONS

- Recall, hidden-layer activation functions make it possible for deep neural nets to learn complex non-linear functions.
- The design of hidden units is an extremely active area of research. It is usually not possible to predict in advance which activation will work best. Therefore, the design process often consists of trial and error.
- In the following, we will limit ourselves to the most popular activations - Sigmoidal activation and ReLU.
- It is possible for many other functions to perform as well as these standard ones. An overview of further activations can be found ▸ here.

SIGMOIDAL ACTIVATIONS

- Sigmoidal functions such as *tanh* and the *logistic sigmoid* bound the outputs to a certain range by "squashing" their inputs.

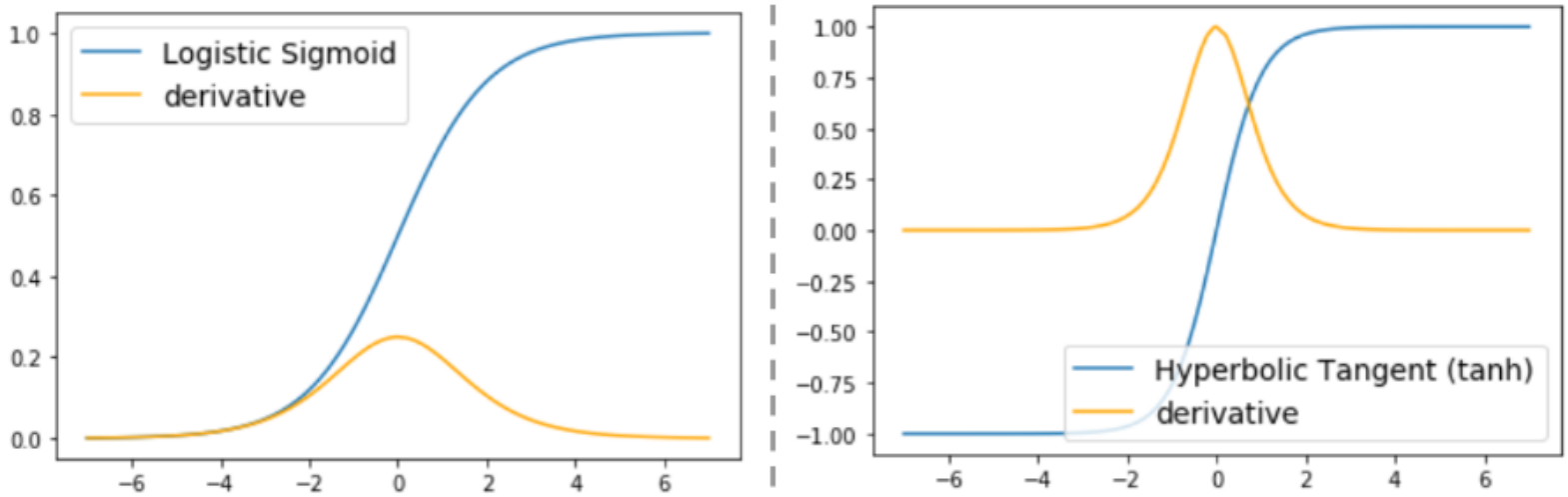


Figure: Sigmoidal activations (Savino & Tondolo, 2021)

- In each case, the function is only sensitive to its inputs in a small neighborhood around 0.
- Furthermore, the derivative is never greater than 1 and is close to zero across much of the domain.

SIGMOIDAL ACTIVATION FUNCTIONS

1. **Saturating Neurons:**
   - We know: $\sigma'(z_{in}) \to 0$ for $|z_{in}| \to \infty$.

   $\to$ Neurons with sigmoidal activations "saturate" easily, that is, they stop being responsive when $|z_{in}| \gg 0$.

SIGMOIDAL ACTIVATION FUNCTIONS

2. **Vanishing Gradients:** Consider the vector of error signals $\boldsymbol{\delta}^{(i)}$ in layer i

$$\boldsymbol{\delta}^{(i)} = \mathbf{W}^{(i+1)}\boldsymbol{\delta}^{(i+1)} \odot \sigma'\left(\boldsymbol{z}_{in}^{(i)}\right), i \in \{1, ..., O\}.$$

Each k-th component of the vector expresses how much the loss L changes when the input to the k-th neuron $z_{k,in}^{(i)}$ changes.

- We know: $\sigma'(z) < 1$ for all $z \in \mathbb{R}$.

$\rightarrow$ In each step of the recursive formula above, the value will be multiplied by a value smaller than one

$$\begin{aligned}
\boldsymbol{\delta}^{(1)} &= \mathbf{W}^{(2)}\boldsymbol{\delta}^{(2)} \odot \sigma'\left(z_{in}^{(i)}\right) \\
&= \mathbf{W}^{(2)}\left(\mathbf{W}^{(3)}\boldsymbol{\delta}^{(3)} \odot \sigma'\left(z_{in}^{(i)}\right)\right) \odot \sigma'\left(z_{in}^{(i)}\right) \\
&= \ldots
\end{aligned}$$

- When this occurs, earlier layers train *very* slowly (or not at all).

RECTIFIED LINEAR UNITS (RELU)

- The ReLU activation solves the vanishing gradient problem.

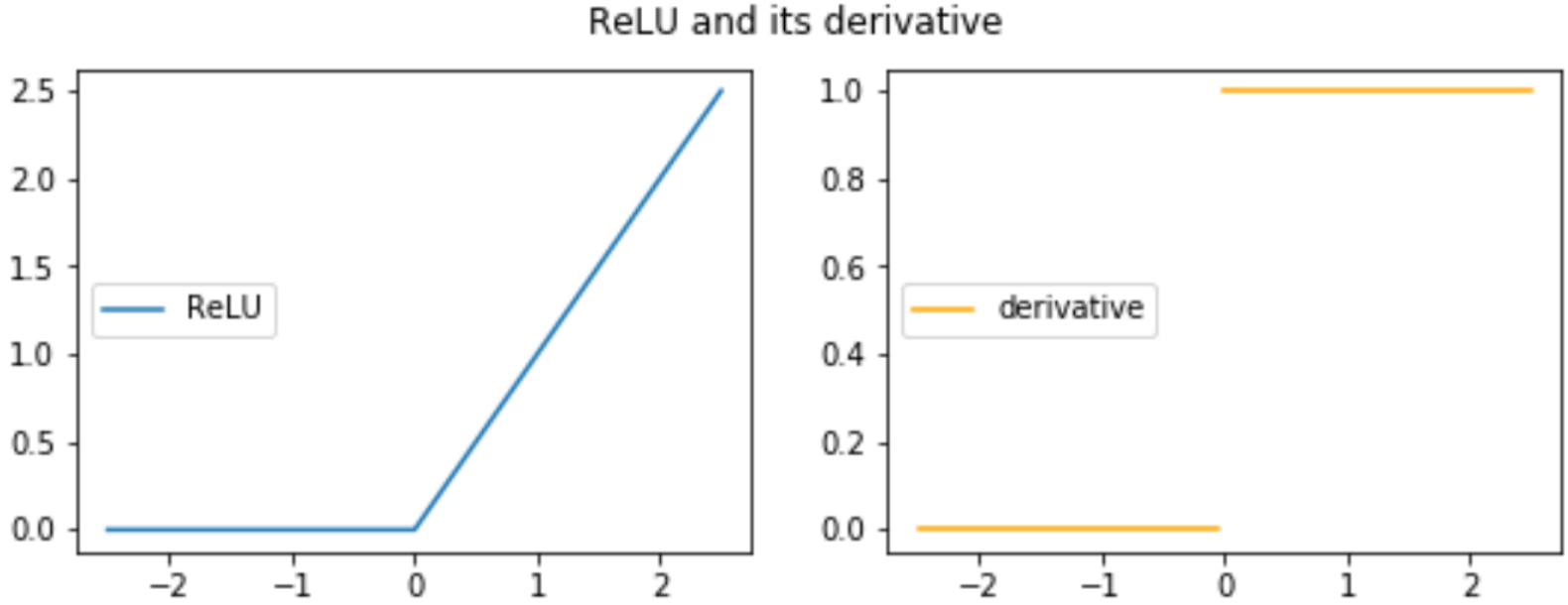


Figure: ReLU activation (Savino & Tondolo, 2021)

- In regions where the activation is positive, the derivative is 1.
- As a result, the derivatives do not vanish along paths that contain such "active" neurons even if the network is deep.
- Note that the ReLU is not differentiable at 0 (Software implementations return either 0 or 1 for the derivative at this point).

RECTIFIED LINEAR UNITS (RELU)

- ReLU units can significantly speed up training compared to units with saturating activations.

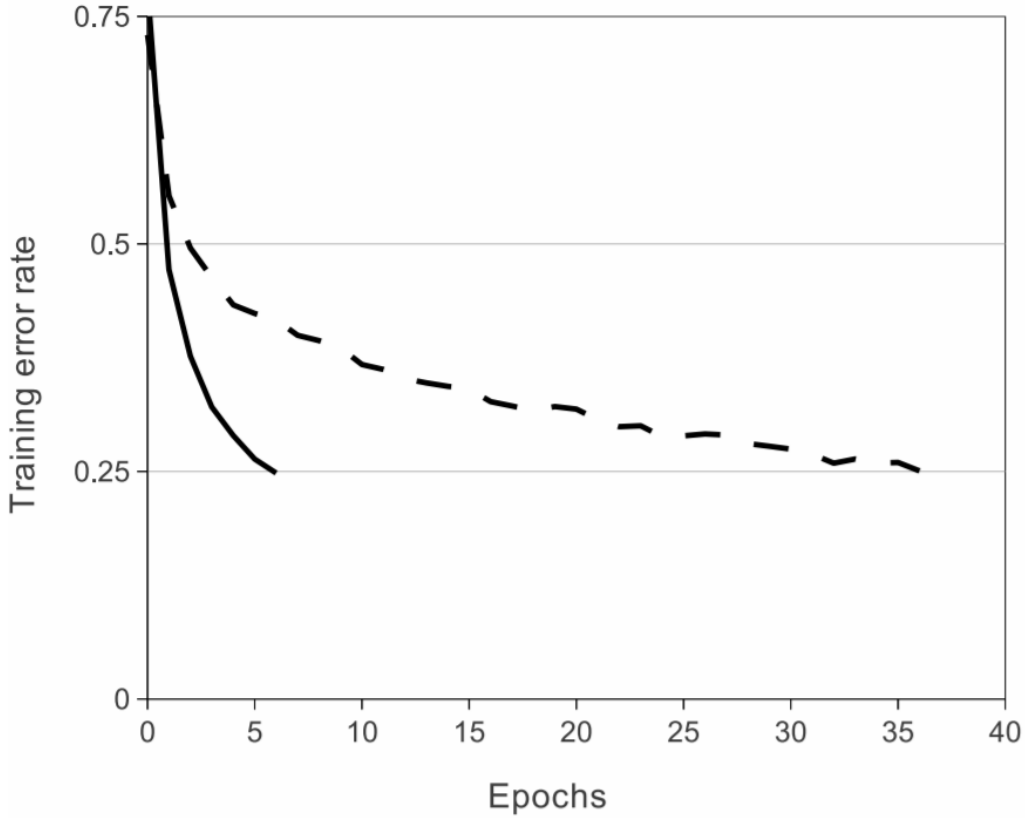


Figure: A four-layer convolutional neural network with ReLUs (solid line) reaches a 25% training error rate on the CIFAR-10 dataset six times faster than an equivalent network with tanh neurons (dashed line) (Krizhevsky et al., 2012).

RECTIFIED LINEAR UNITS (RELU)

- A downside of ReLU units is that when the input to the activation is negative, the derivative is zero. This is known as the "dying ReLU problem".

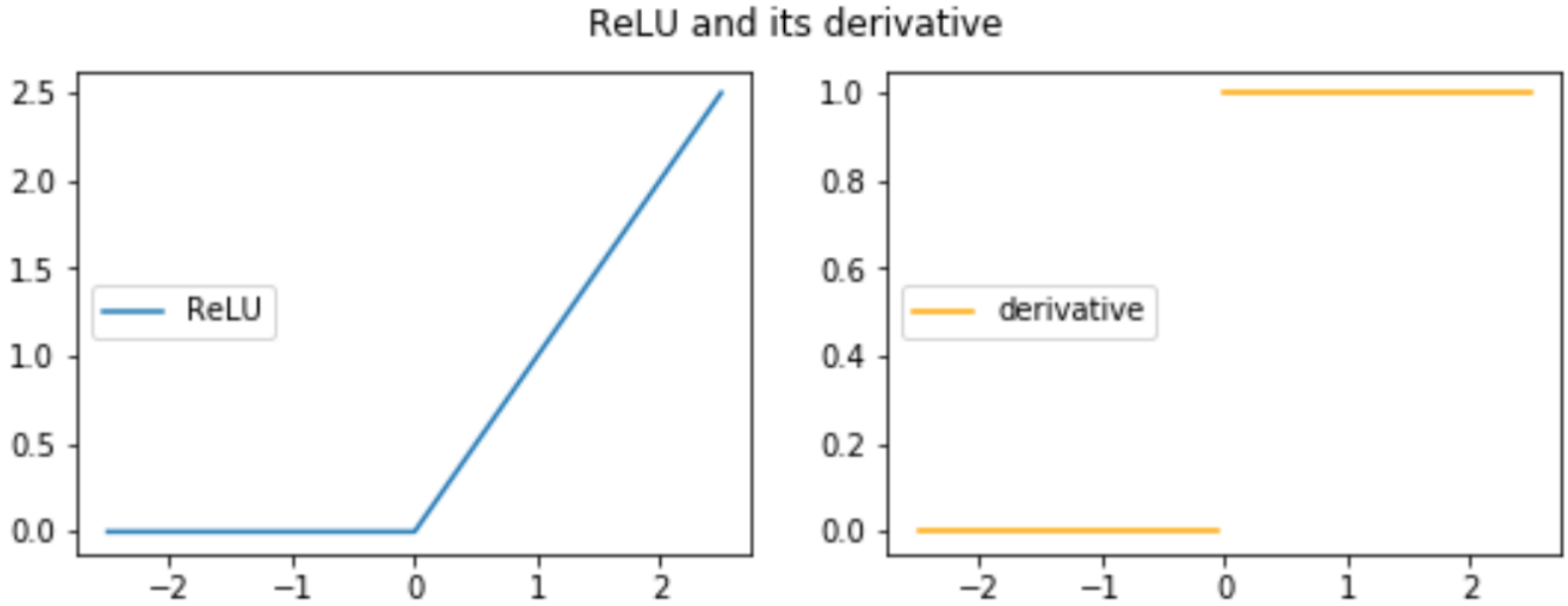


Figure: ReLU activation (Savino & Tondolo, 2021)

- When a ReLU unit "dies", that is, when its activation is 0 for all datapoints, it kills the gradient flowing through it during backpropogation.
- This means such units are never updated during training and the

GENERALIZATIONS OF RELU

- There exist several generalizations of the ReLU activation that have non-zero derivatives throughout their domains.
- *Leaky ReLU*:

$$LReLU(v) = \begin{cases} v & v \geq 0 \\ \alpha v & v < 0 \end{cases}$$

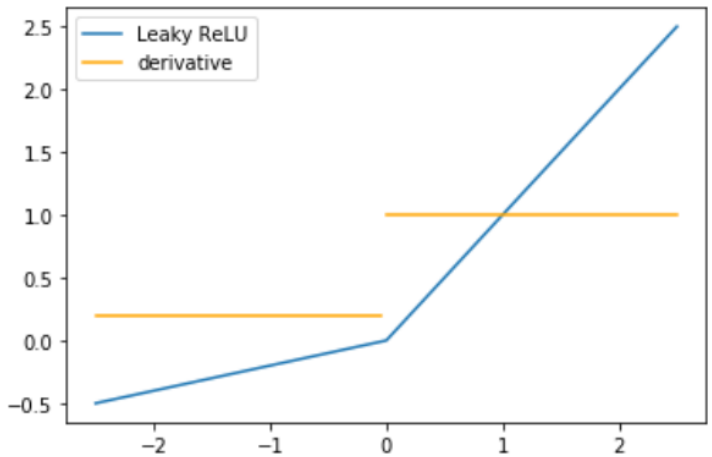


Figure: Leaky ReLU (Savino & Tondolo, 2021)

- Unlike the ReLU, when the input to the Leaky ReLU activation is negative, the derivative is α which is a small positive value (such as 0.01).

GENERALIZATIONS OF RELU

- A variant of the Leaky ReLU is the *Parametric ReLU (PReLU)* which learns the α from the data through backpropagation.
- *Exponential Linear Unit (ELU)*:

$$ELU(v) = \begin{cases} v & v \geq 0 \\ \alpha(e^v - 1) & v < 0 \end{cases}$$

- *Scaled Exponential Linear Unit (SELU)*:

$$SELU(v) = \lambda \begin{cases} v & v \geq 0 \\ \alpha(e^v - 1) & v < 0 \end{cases}$$

Note: In ELU and SELU, α and λ are hyperparameters that are set before training.

- These generalizations may perform as well as or better than the ReLU on some tasks.

GENERALIZATIONS OF RELU

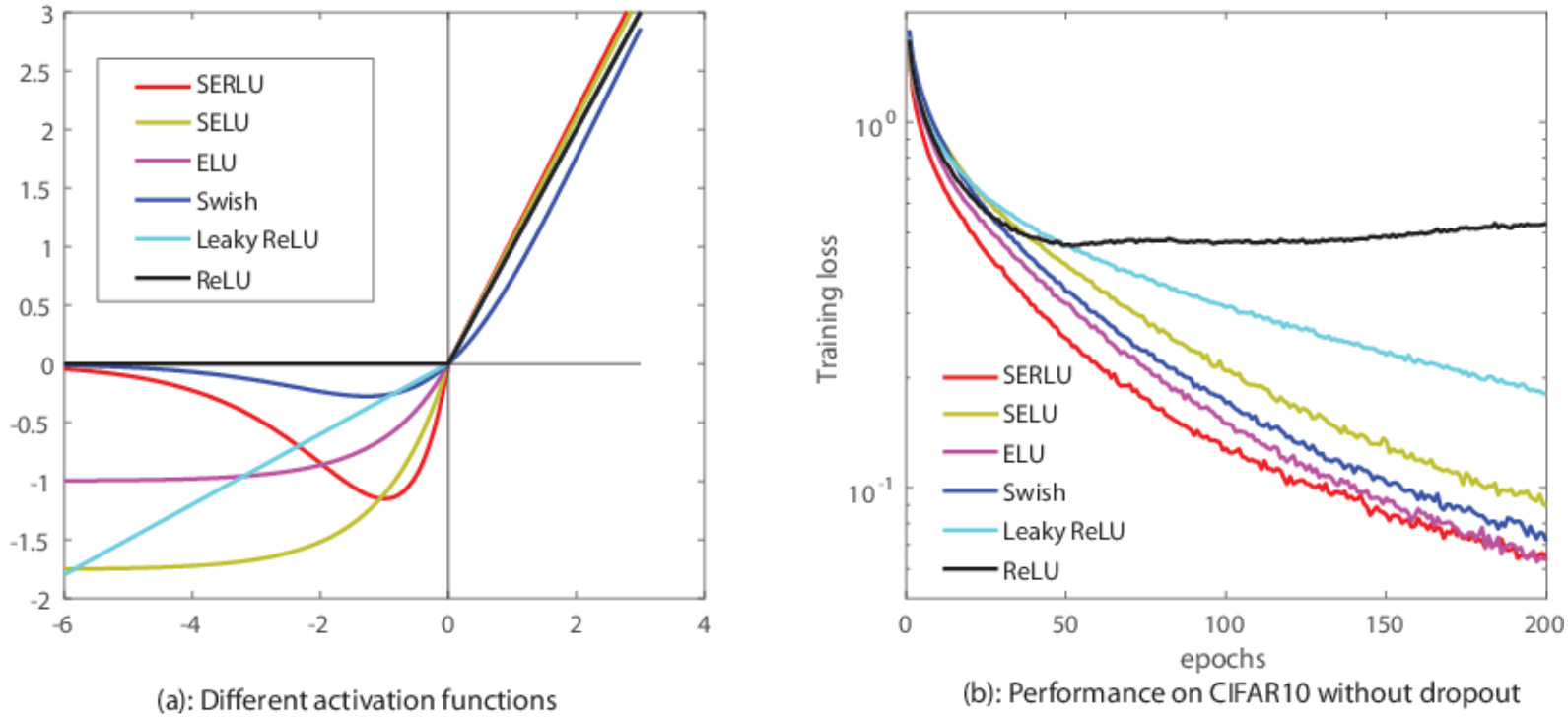


Figure: Visualization of different generalisations of ReLU (Zhang et al., 2018)

RESIDUAL BLOCK (SKIP CONNECTIONS)

Problem setting: theoretically, we could build infinitely deep architectures as the net should learn to pick the beneficial layers and skip those that do not improve the performance automatically.

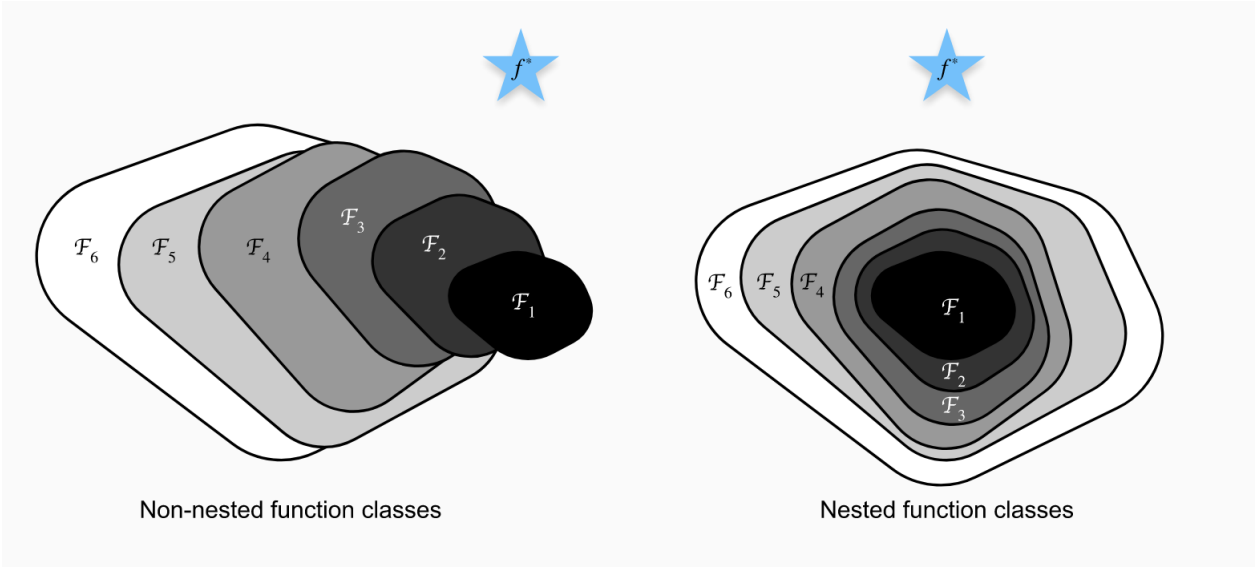


Figure: For non-nested function classes, a larger function class does not guarantee to get closer to the "truth" function ($\mathcal{F}*$). This does not happen in nested function classes (Dive into Deep Learning).

RESIDUAL BLOCK (SKIP CONNECTIONS)

- But: this skipping would imply learning an identity mapping $\mathbf{x} = \mathcal{F}(\mathbf{x})$. It is very hard for a neural net to learn such a 1:1 mapping through the many non-linear activations in the architecture.
- Solution: offer the model explicitly the opportunity to skip certain layers if they are not useful.
- Introduced in *He et. al , 2015* and motivated by the observation that stacking evermore layers increases the test- as well as the train-error ($\neq$ overfitting).

RESIDUAL BLOCK (SKIP CONNECTIONS)

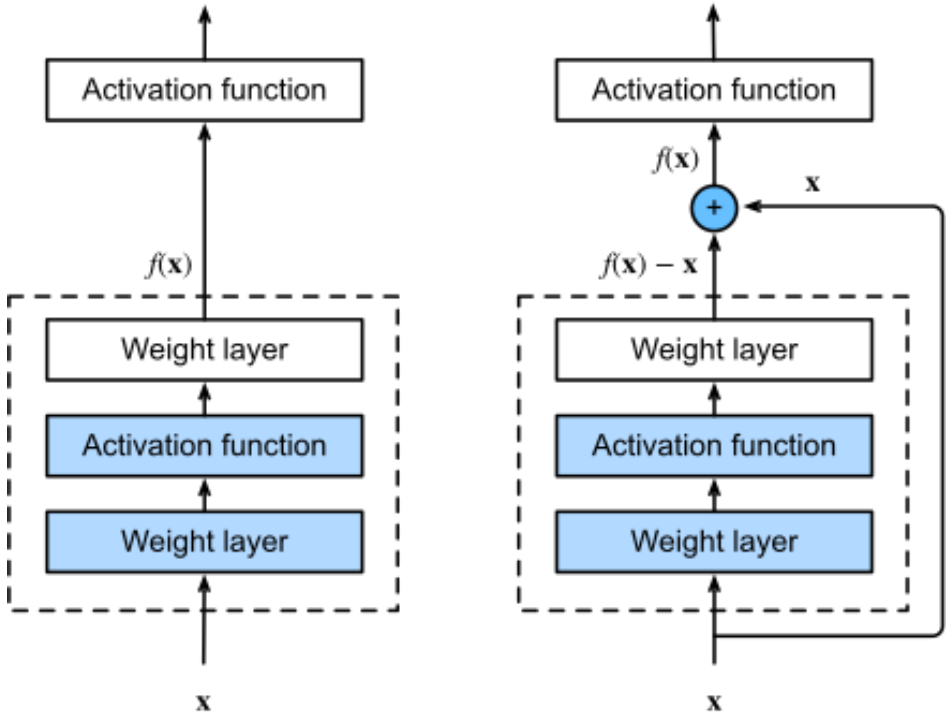


Figure: A regular block (left) and a residual block (right) (Dive into Deep Learning).

RESIDUAL BLOCK (SKIP CONNECTIONS)

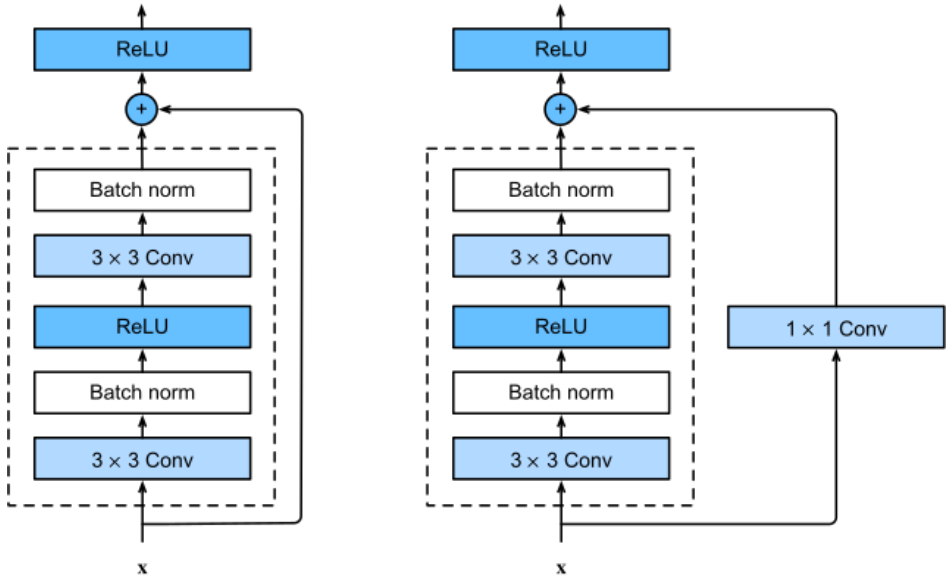


Figure: ResNet block with and without 1×1 convolution.The information flows through two layers and the identity function. Both streams of information are then element-wise summed and jointly activated (Dive into Deep Learning).

RESIDUAL BLOCK (SKIP CONNECTIONS)

- Let $\mathcal{H}(\mathbf{x})$ be the optimal underlying mapping that should be learned by (parts of) the net.
- **x** is the input in layer *l* (can be raw data input or the output of a previous layer).
- $\mathcal{H}(\mathbf{x})$ is the output from layer *l*.
- Instead of fitting $\mathcal{H}(\mathbf{x})$, the net is ought to learn the residual mapping $\mathcal{F}(\mathbf{x}) := \mathcal{H}(\mathbf{x}) - \mathbf{x}$ whilst **x** is added via the identity mapping.
- Thus, $\mathcal{H}(\mathbf{x}) = \mathcal{F}(\mathbf{x}) + \mathbf{x}$, as formulated on the previous slide.
- The model should only learn the **residual mapping** $\mathcal{F}(\mathbf{x})$
- Thus, the procedure is also referred to as **Residual Learning**.

RESIDUAL BLOCK (SKIP CONNECTIONS)

- The element-wise addition of the learned residuals $\mathcal{F}(\mathbf{x})$ and the identity-mapped data $\mathbf{x}$ requires both to have the same dimensions.
- To allow for downsampling within $\mathcal{F}(\mathbf{x})$ (via pooling or valid-padded convolutions), the authors introduce a linear projection layer W_s .
- W_s ensures that $\mathbf{x}$ is brought to the same dimensionality as $\mathcal{F}(\mathbf{x})$ such that:

$$y = \mathcal{F}(\mathbf{x}) + W_s\mathbf{x},$$

- y is the output of the skip module and W_s represents the weight matrix of the linear projection (# rows of W_s = dimensionality of $\mathcal{F}(\mathbf{x})$).
- This idea applies to fully connected layers as well as to convolutional layers.

RESNET ARCHITECTURE

- The residual mapping can learn the identity function more easily, such as pushing parameters in the weight layer to zero.
- We can train an effective deep neural network by having residual blocks.
- Inputs can forward propagate faster through the residual connections across layers.
- ResNet had a major influence on the design of subsequent deep neural networks, both for convolutional and sequential nature.

RESNET ARCHITECTURE

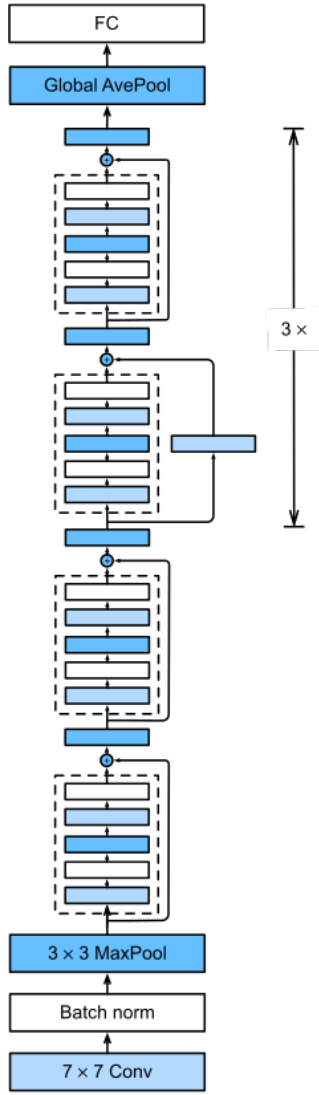


Figure: The ResNet-18 architecture (Dive into Deep Learning).