IMAGE DATA ANALYSIS (6CFU)

MODULE OF

REMOTE SENSING

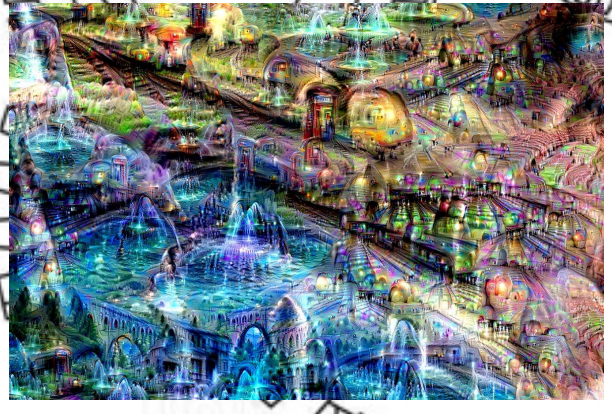
(9 CFU)

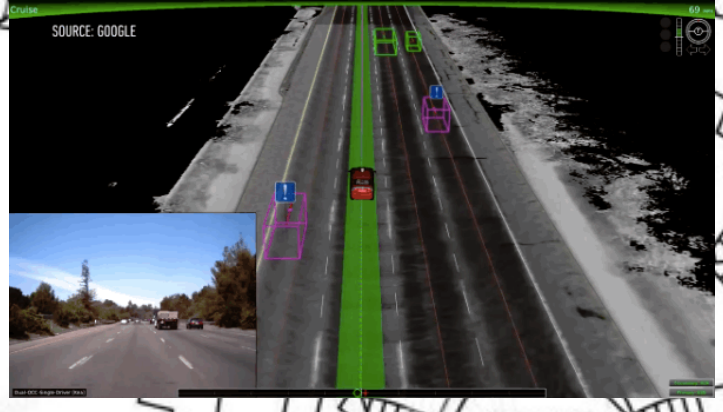
A.Y. 2022/23

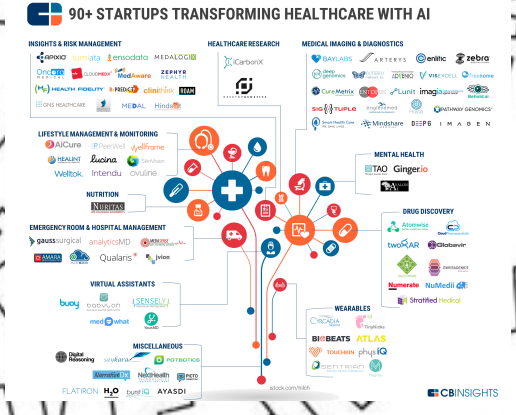
MASTER OF SCIENCE IN COMMUNICATION TECHNOLOGIES AND MULTIMEDIA
MASTER OF SCIENCE IN COMPUTER SCIENCE, LM INGEGNERIA INFORMATICA

PROF. ALBERTO SIGNORONI – DR. MATTIA SAVARDI

FROM SHALLOW TO DEEP NEURAL NETWORKS







Main resources

1. Deep Learning By — Ian Good fellow and Yoshua Bengio and Aaron Courville (http://www.deeplearningbook.org/)
2. Book-Lab Dive into Deep Learning https://d2l.ai/index.html
3. Deep Learning Tutorial – University of Montreal https://openlibra.com/en/book/deep-learning-tutorial
4. http://neuralnetworksanddeeplearning.com/
5. News feed https://www.quora.com/topic/Deep-Learning
6. Scientific literature (papers, conferences, preprints...)

Main related courses

Li (Stanford): Convolutional Neural Networks for Visual Recognition http://cs231n.stanford.edu/
https://www.youtube.com/playlist?list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv

Yann LeCun and Alfredo Canziani (NYU): Deep Learning
https://cds.nyu.edu/deep-learning/
https://atcold.github.io/pytorch-Deep-Learning/

Grosse (UoT): Intro to Neural Networks and Machine Learning
http://www.cs.toronto.edu/~rgrosse/courses/csc321_2018/

McAllester (TTI-C): Fundamentals of Deep Learning https://mcallester.github.io/ttic-31230/FALL2021/

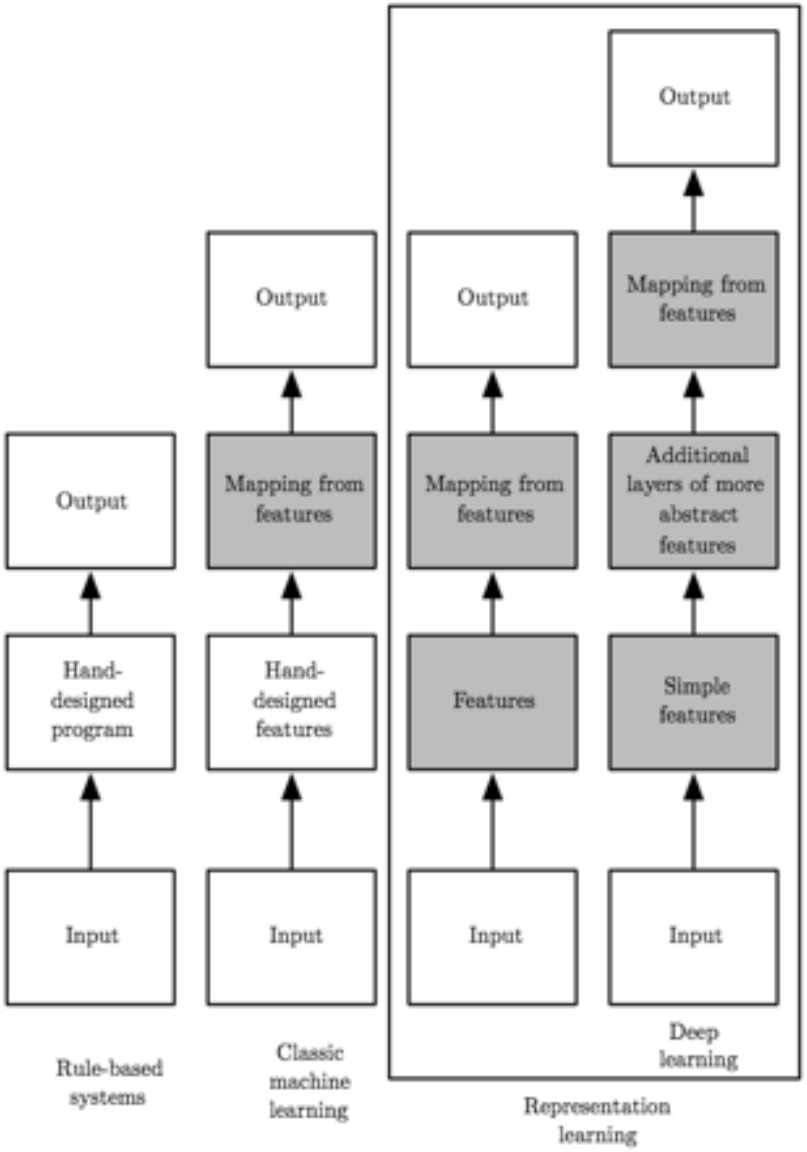
Leal-Taixe, Niessner (TUM): Introduction to Deep Learning http://niessner.github.io/I2DL/

Alexander Amini, Ava Soleimany (MIT)
http://introtodeeplearning.com

Sargur Srihari (Buffalo): Deep Learning
https://cedar.buffalo.edu/~srihari/CSE676/

Abbeel, Chen, Ho, Srinivas (Berkeley): Deep Unsupervised Learning
https://sites.google.com/view/berkeley-cs294-158-sp20/home

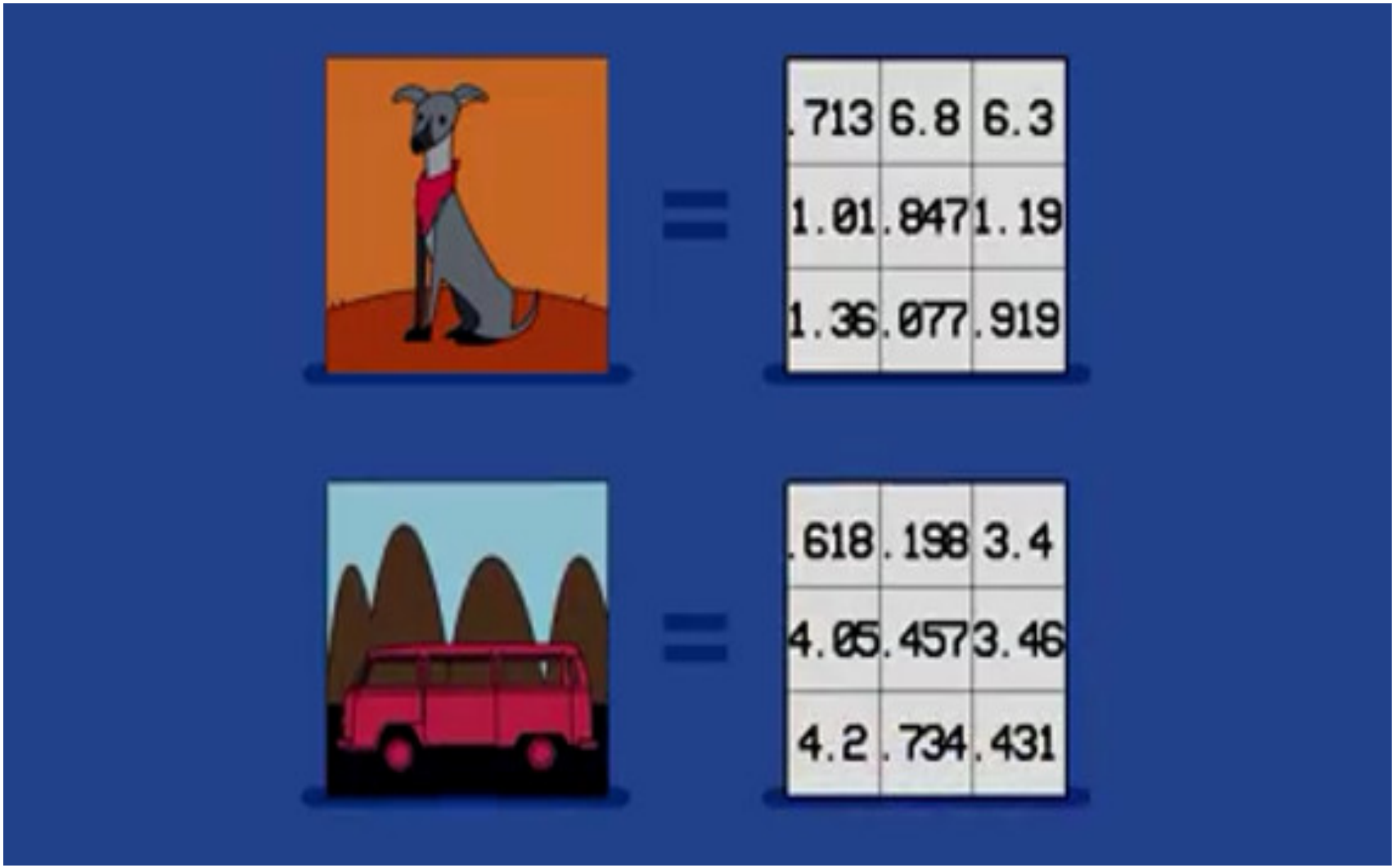
Knowledge-base vs Representation learning



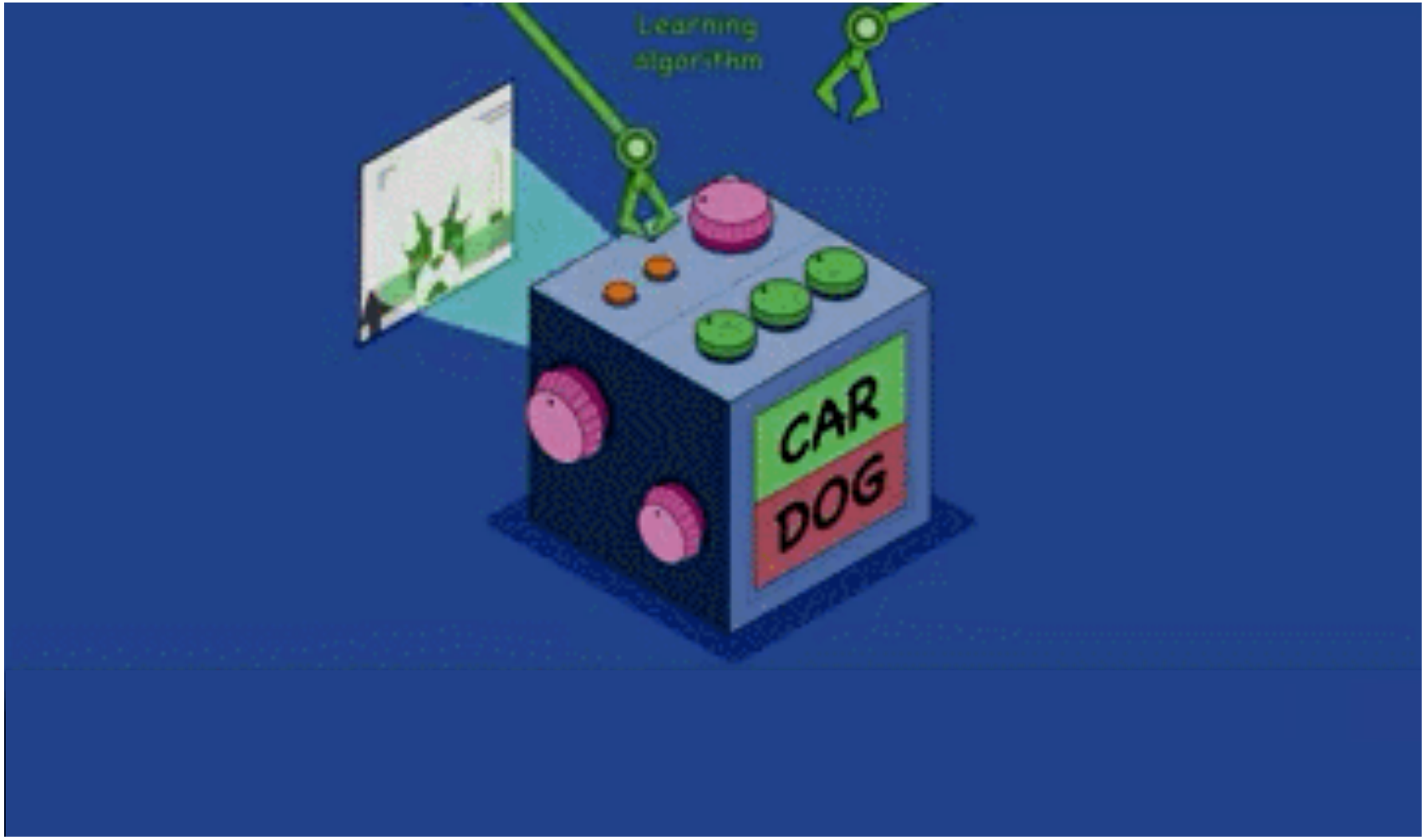
Knowledge-base vs Representation learning approaches
(learning blocks are shaded in gray)

Knowledge-base vs Representation learning

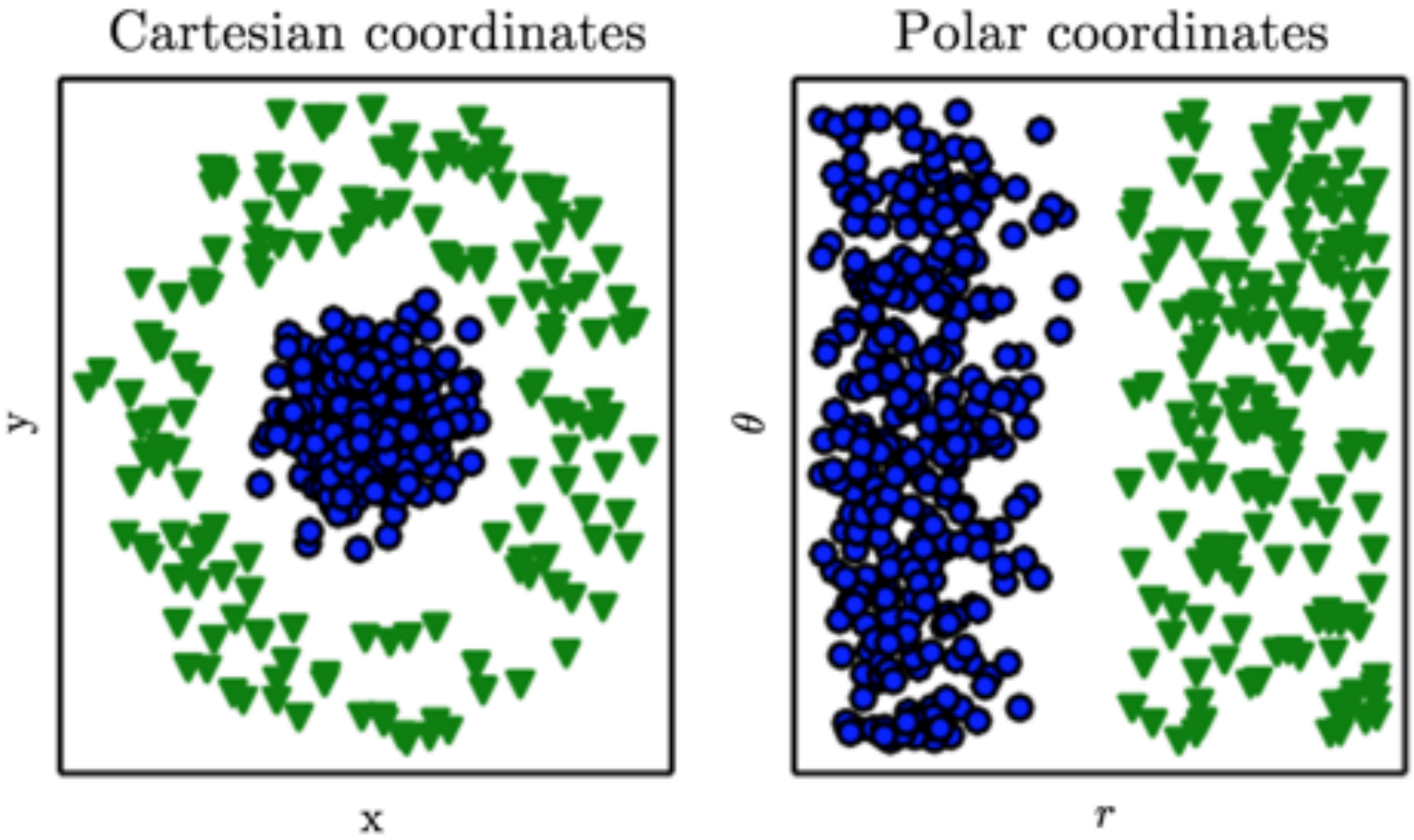
Knowledge-base vs Representation learning



Knowledge-base vs Representation learning

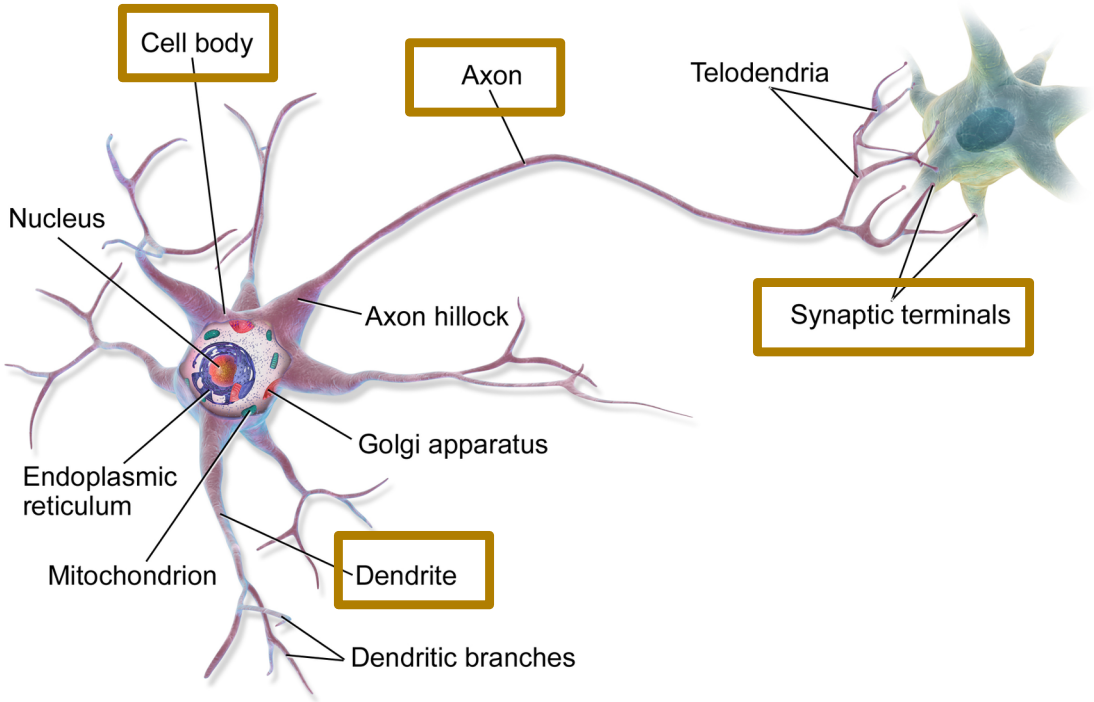


Knowledge-based vs Representation learning

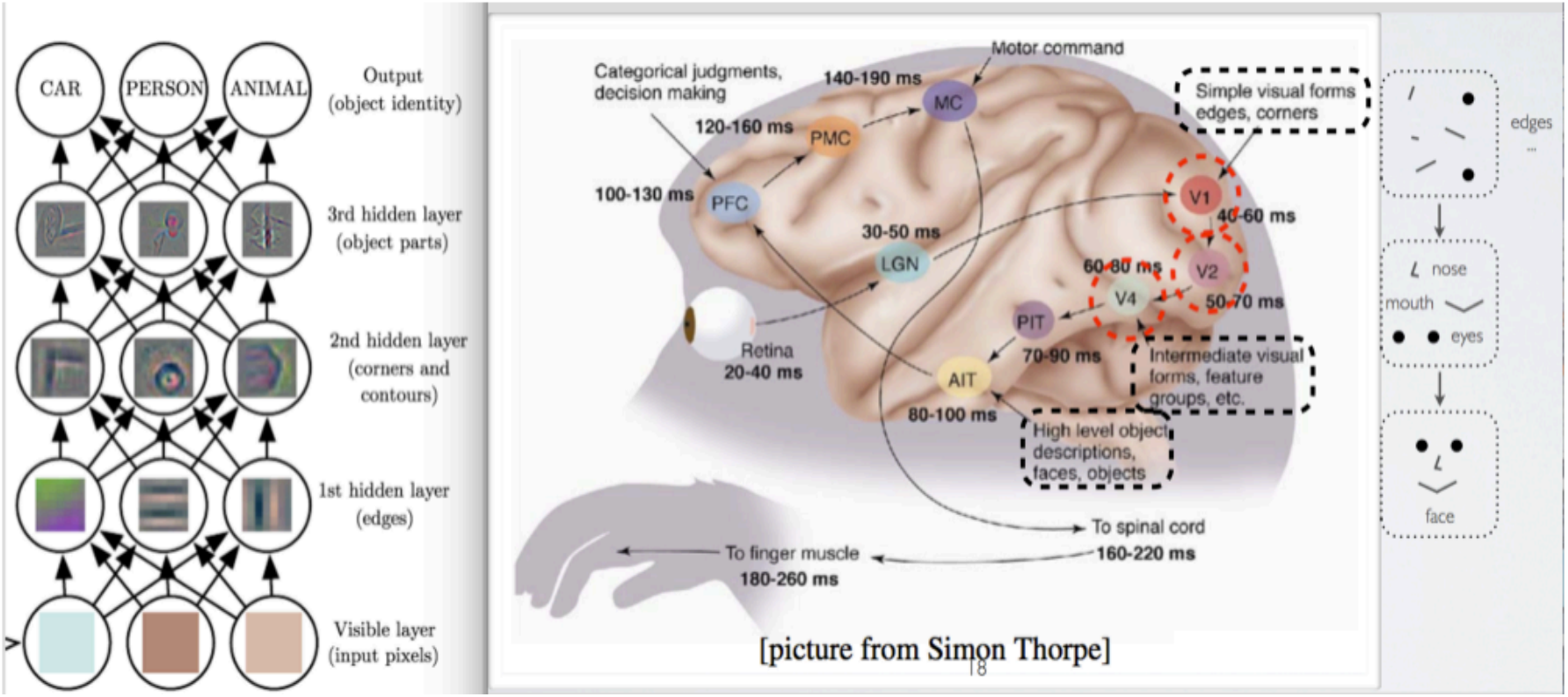


Feed-forward neural networks…

- … are barely inspired to human and animal brain neurons and their connections
- **Neurons in the brain are structured in layers**
  - They receive input from many other units and compute their own activation
  - The sigmoid activation function is guided by neuroscientific observations
  - However, the architecture and training of modern networks differs radically
  - Our main goal is not to model the brain, but to achieve statistical generalization



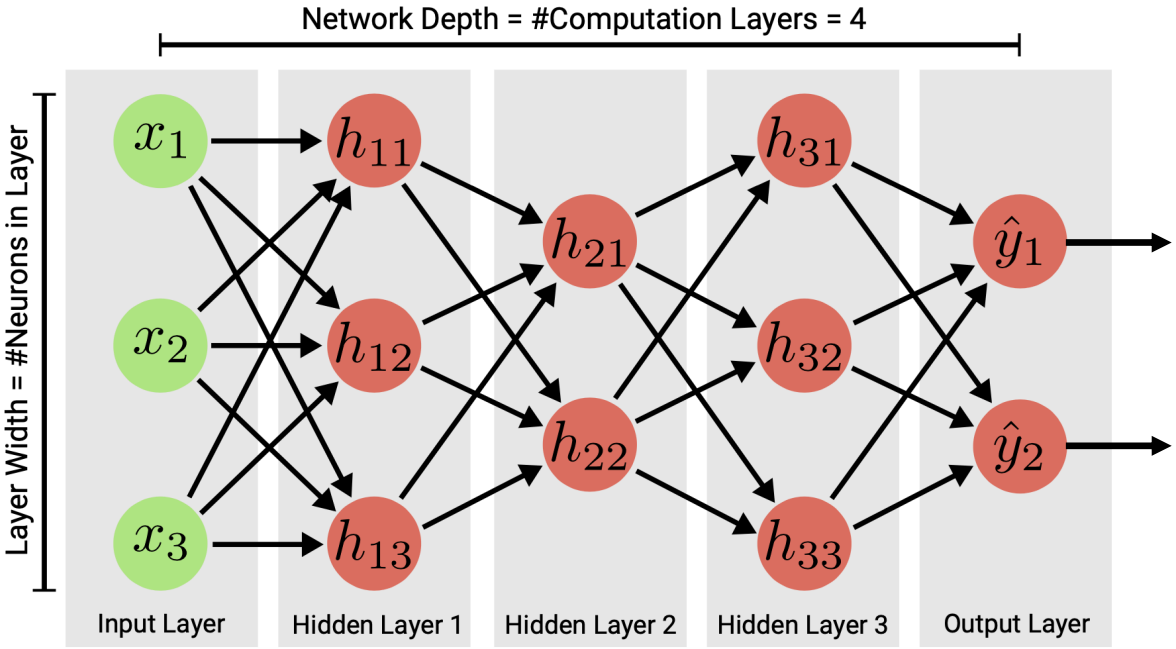
The mammalian visual cortex is hierarchical



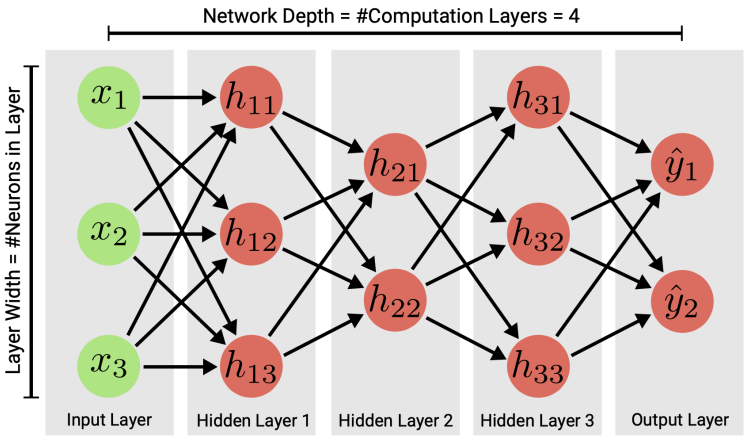
[picture from Simon Thorpe]

Multilayer Perceptron (MLP)

- MLPs are **feedforward** neural networks with no feedback connections
  - If all layers are fully connected, MLPs are also called Fully Connected networks
- They **compose** several non-linear functions $f(\mathbf{x}) = \hat{\mathbf{y}}(\mathbf{h}_3(\mathbf{h}_2(\mathbf{h}_1(\mathbf{x}))))$
  - where $\mathbf{h}_i(\cdot)$ are called **hidden layers** and $\hat{\mathbf{y}}(\cdot)$ is the **output layer**
- The data specifies only the behavior of the output layer (thus the name "hidden")
  - **Hidden layer** $\mathbf{h}_i = g(\mathbf{A}_i\mathbf{h}_{i-1} + \mathbf{b}_i)$ with **activation function** $g(\cdot)$ and weights $\mathbf{A}_i, \mathbf{b}_i$



Multilayer Perceptron (MLP)

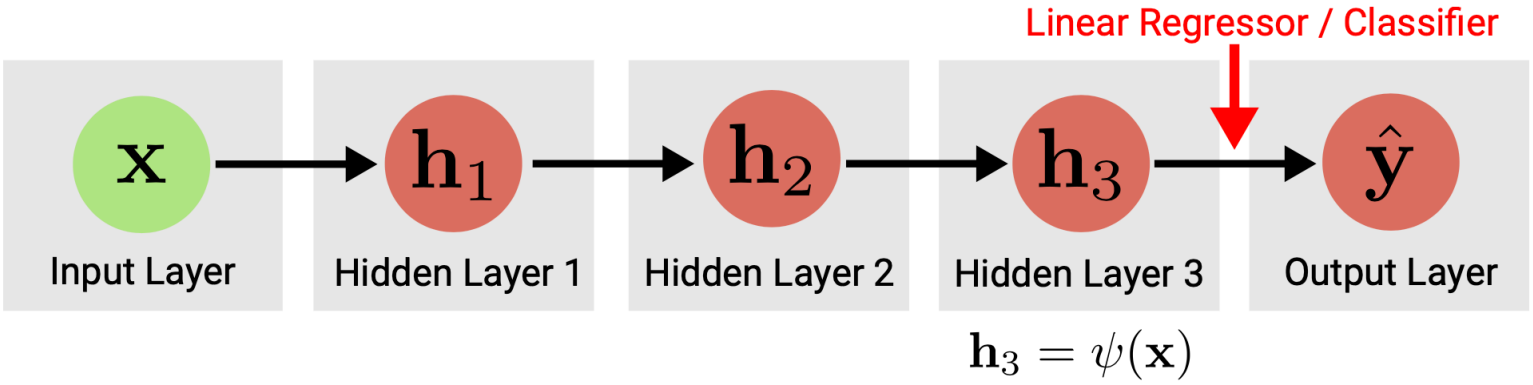


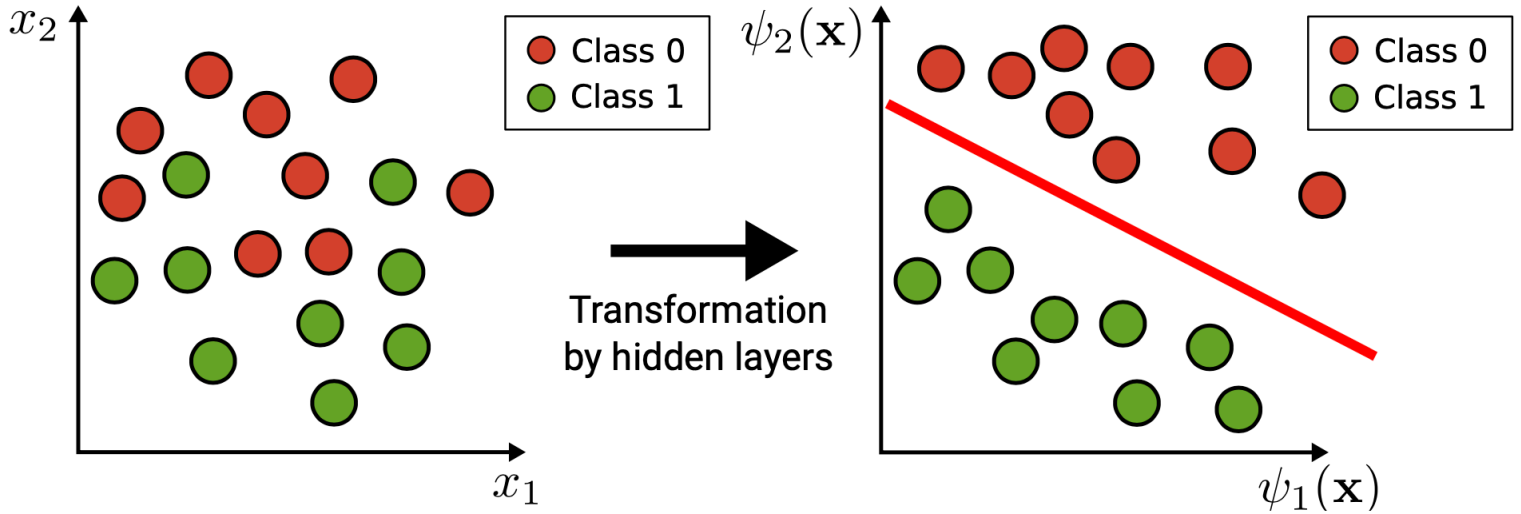
- Each layer i comprises multiple **neurons** j (NB the notation is different here with respect to our previous lesson on Neural Nets) which are implemented as **affine transformations** $(\mathbf{a}^{\mathrm{T}}\mathbf{x} + \mathbf{b})$ followed by non-linear **activation functions** (g):

$$\mathbf{h}_{ij} = g\left(\mathbf{a}_{ij}^{\mathrm{T}}\mathbf{h}_{i-1} + b_{ij}\right)$$

- Each neuron in each layer is **fully connected** to all neurons of the previous layer (this is the main architecture despite others have been proposed and used)
- The overall length of the chain is the **depth** of the model ⇒ "Deep Learning"
- The name MLP is somewhat misleading as we don't use Threshold Logic Units as in the Perceptron training algorithm (Rosenblatt, 1958). We use Backpropagation (first discovered in 1961 but «reinvented» by Hinton et al. in 1986) for learning.

Representation (feature) learning perspective of MLP





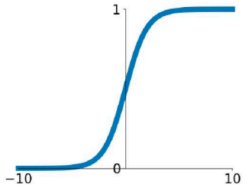
Original representation

Final (learned) representation

Activation (non-linear) functions $g(\cdot)$

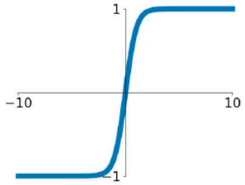
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



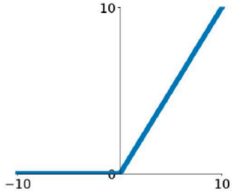
tanh

$$\tanh(x)$$



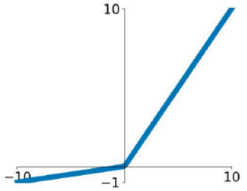
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

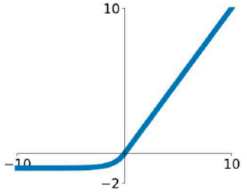


Maxout

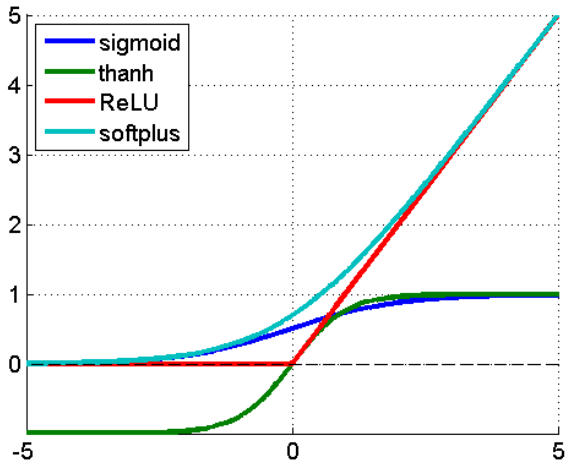
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

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



NB Sigmoid is neurobiologically inspired, however squashing functions cause the gradient to vanish (for large portions of input values) and this creates problems to the backpropagation of the gradient with the risk to curb the learning process.



Training a MLP (and a FCN)

- Training exploits parallel computing power of modern GPUs
- Initial observation:
  - Large datasets typically do not fit into GPU memory $\Rightarrow |\mathcal{X}_{\text{batch}}| < |\mathcal{X}|$

Instead of single pattern samples we use randomized selections of batches of samples

Algorithm for training an MLP using (stochastic) gradient descent:

1. Initialize weights $\mathbf{w}$, pick learning rate η and minibatch size $|\mathcal{X}_{\text{batch}}|$
2. Draw (random) minibatch $\mathcal{X}_{\text{batch}} \subseteq \mathcal{X}$
3. For all elements $(\mathbf{x}, \mathbf{y}) \in \mathcal{X}_{\text{batch}}$ of minibatch (in parallel) do:
   - 3.1 Forward propagate $\mathbf{x}$ through network to calculate $\mathbf{h}_1, \mathbf{h}_2, \ldots, \hat{\mathbf{y}}$
   - 3.2 Backpropagate gradients through network to obtain $\nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$
4. Update gradients: $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \frac{1}{|\mathcal{X}_{\text{batch}}|} \sum_{(\mathbf{x},\mathbf{y}) \in \mathcal{X}_{\text{batch}}} \nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$
5. If validation error decreases, go to step 2, otherwise stop

This is called “early stopping” although other stopping criteria can be used, e.g. reaching a defined number of iterations

MLP expressiveness

- This following two-layer MLP

$$\mathbf{h} = g(\mathbf{A}_1\mathbf{x} + \mathbf{b}_1)$$
$$\mathbf{y} = g(\mathbf{A}_2\mathbf{h} + \mathbf{b}_2)$$

  can be written as

$$\mathbf{y} = g(\mathbf{A}_2\, g(\mathbf{A}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2)$$

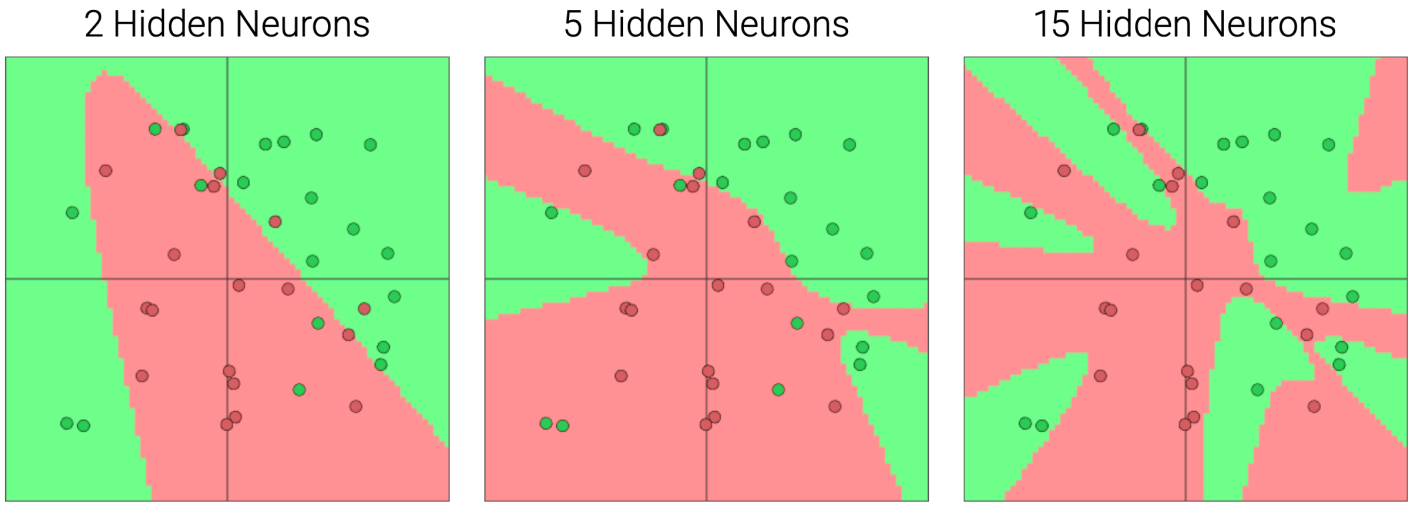
  What if we would be using a linear activation function $g(\mathbf{x}) = \mathbf{x}$?

$$\mathbf{y} = \mathbf{A}_2\left(\mathbf{A}_1\mathbf{x} + \mathbf{b}_1\right) + \mathbf{b}_2 = \mathbf{A}_2\mathbf{A}_1\mathbf{x} + \mathbf{A}_2\mathbf{b}_1 + \mathbf{b}_2 = \mathbf{A}\mathbf{x} + \mathbf{b}$$

- With linear activations, a multi-layer network can only express linear functions
  - Non-linearities are crucial in terms of *expressiveness* of the network
- What is the model capacity of MLPs with non-linear activation functions?
  - **Is it better a *deeper* or a *wider* neural network?**

Number of neurons - Example 1: expressiveness

- An example taken from
  - https://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html



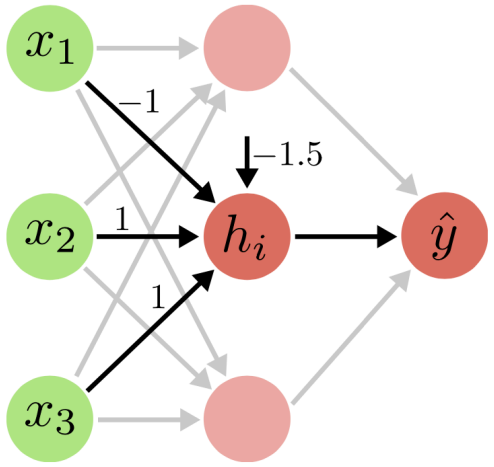
- More neurons in a layer increase the expressiveness
- Less neurons lead to more regular solutions
  - Decreasing or limiting the number of neurons within layers is a form of *regularization* (among others we will see)

Number of neurons - Example 2: binary case and complexity

- We are interested in a LTU network able to recognize only one case out of 2^D (here D=3)

x_1	x_2	x_3	y
⋮	⋮	⋮	⋮
0	1	0	0
0	**1**	**1**	**1**
1	0	0	0
⋮	⋮	⋮	⋮

$$\hat{y} = \sum_i \underbrace{[\mathbf{a}_i^\top \mathbf{x} + b_i > 0]}_{h_i}$$



- Each hidden **linear threshold unit** h_i recognizes one possible input vector
- We need 2^D hidden units to **recognize** all 2^D possible inputs in the binary case
- More general I/O mappings are possible generalizing LTU with sigmoidal activations
  - the intuition seems to lead to the fact that it should be **possible to approximate any input-output function** by means of a MLP with one hidden layer
  - however, this example shows that this could require **exponential increase of the number of neurons** with respect to the complexity of the mapping

Universal approximation theorems

- **Theorem:**

  Let σ be any continuous discriminatory function. Then finite sums of the form

$$G(\mathbf{x}) = \sum_{j=1}^{N} \alpha_j \sigma(\mathbf{a}_j^\top \mathbf{x} + b_j)$$

  are dense in the space of continuous functions $C(I_n)$ on the n-dimensional unit cube I_n. In other words, given any $f \in C(I_n)$ and $\epsilon > 0$, there is a sum, $G(\mathbf{x})$ for which

$$|G(\mathbf{x}) - f(\mathbf{x})| < \epsilon \quad \text{for all} \quad \mathbf{x} \in I_n$$

- There are many versions of this kind of theorem that can be targeted to different input convex sets and network properties (arbitrary layers'width or arbitrary network depth).
- These theorems has been proved for various activation functions (e.g. Sigmoid, ReLU,…) but they do not tell exactly how many neurons (width vs depth) are necessary to approximate a given function with a desired level of accuracy.

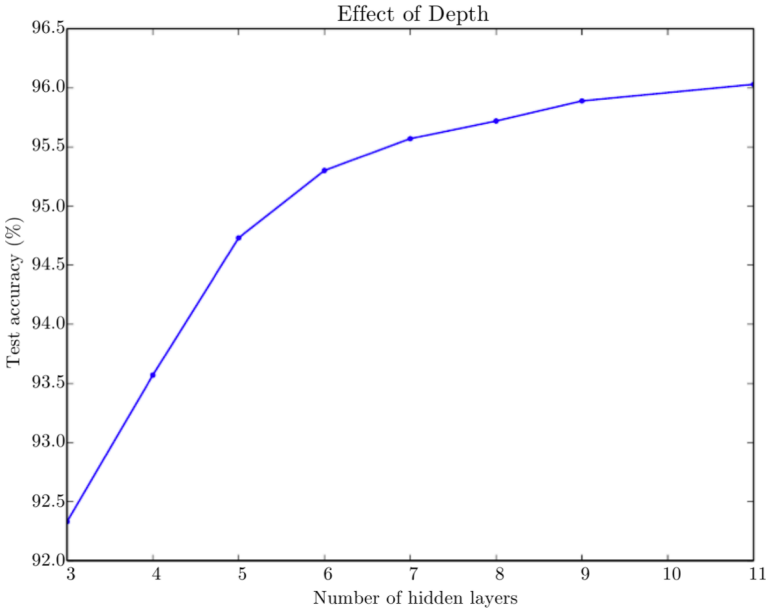
Approximation capability (expressiveness): the Width or Depth dilemma

- Universal approximation theorem on 2-layer networks (only one hidden layer) is appealing but requires **exponential width**
  - This leads to an exponential increase in memory and computation time
  - Moreover, it doesn't lead to generalization ⇒ network simply *memorizes inputs* (see the previous example about the binary case) ⇒ direct path to overfitting
- **Deep networks** can represent functions more compactly (with less parameters)
  - Complex functions modeled as **composition of simple functions** ⇒ *inductive bias* that leads to better generalization capabilities (better model vs noise understanding)
  - This leads to **more compact** models and (empirically) **better generalization** performance

- Example: multi-digit number classification
  - Not too complex task → one is tempted

  to use shallow neural networks to solve it (accordin

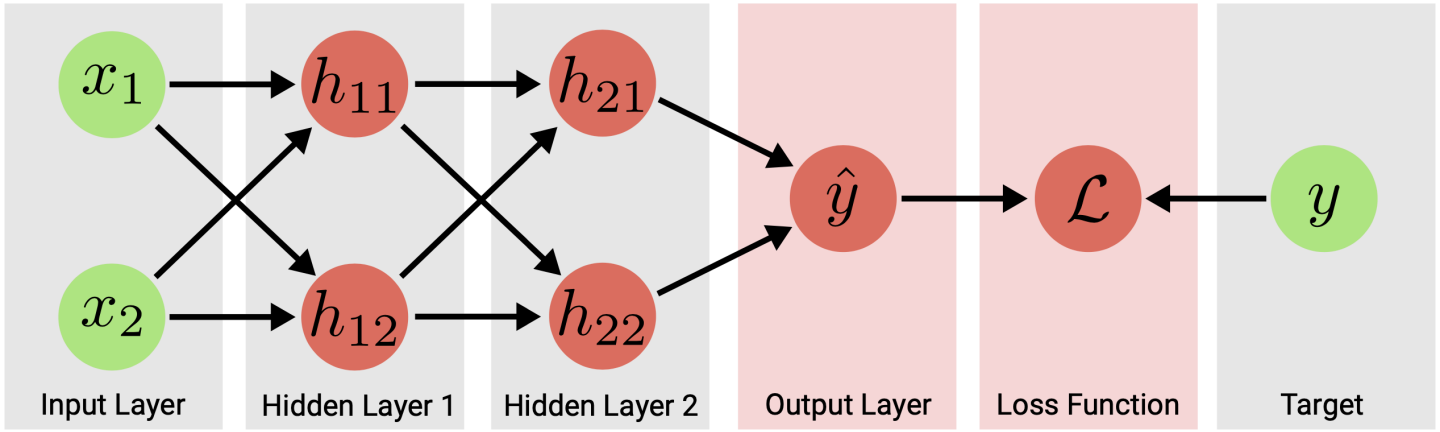
  more traditional approach)
  - From: *Goodfellow, Bulatov, Ibarz, Arnoud and Shet: Multi-digit number recognition from Street View imagery using deep convolutional neural networks. ICLR, 2014*
  - Deeper networks generalize better



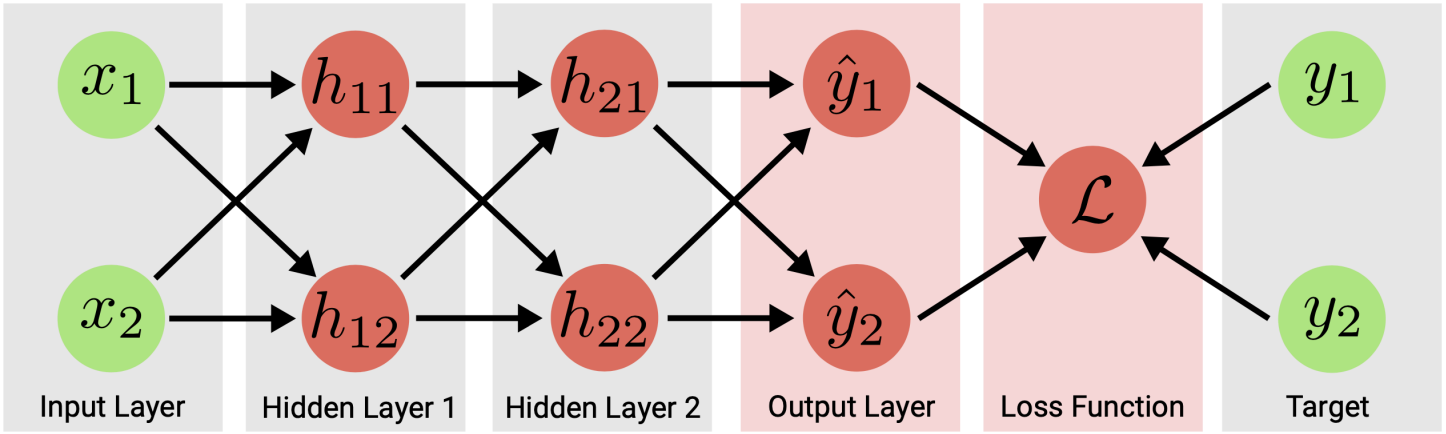
OUTPUTS AND LOSS FUNCTIONS

Output and Loss functions

- The **output layer** is the last layer in a neural network which computes the output
- The **loss function** compares the result of the output layer to the target value(s)
- Choice of output layer and loss function depends on task (discrete, continuous,...)



single output

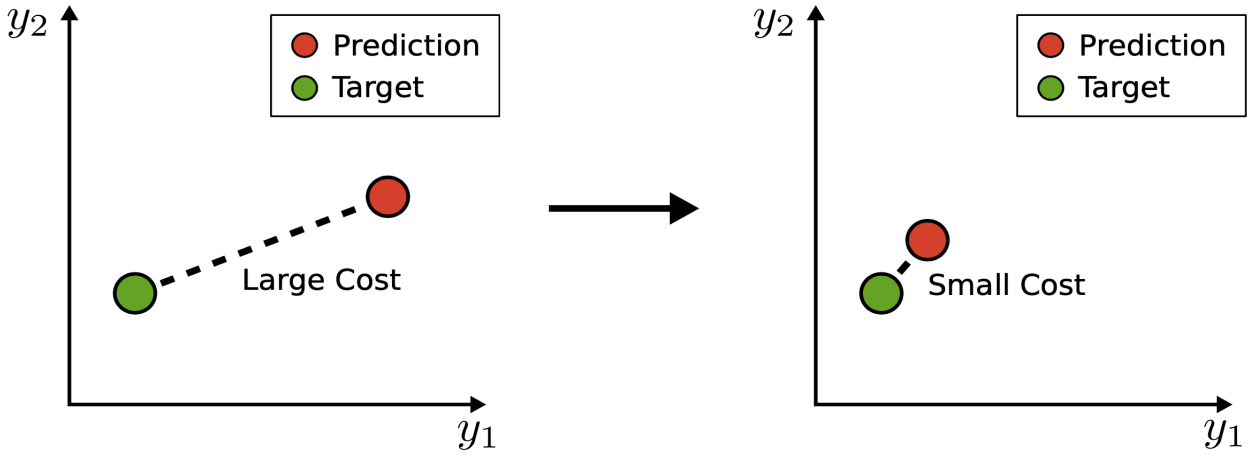


multiple (vector) output

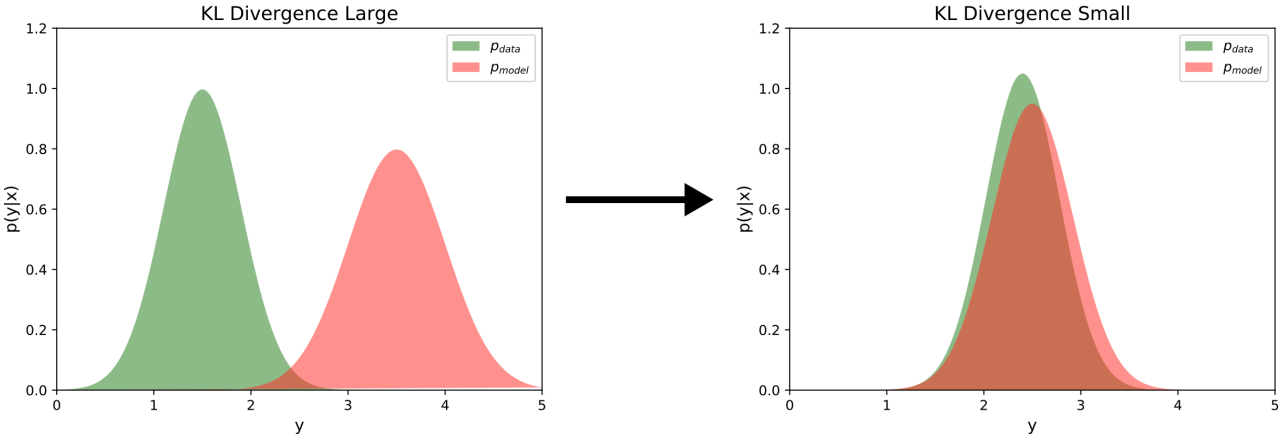
Loss Function

- What is the goal of optimizing the loss function?
  - To try to make the **model output** (=prediction) similar to the **target** (=data)
  - Think of the loss function as a **measure of cost** being paid for a prediction

(single) value level



ensemble (statistical) level



Loss Function

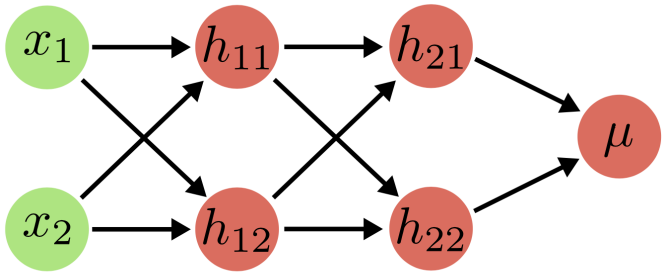
- How to design a good loss function?
  - A loss function can be any differentiable function that we wish to optimize
  - Deriving the cost function from the **maximum likelihood principle** removes the burden of manually designing the cost function for each model
  - Consider the output of the neural network as **parameters of a distribution** over y_i

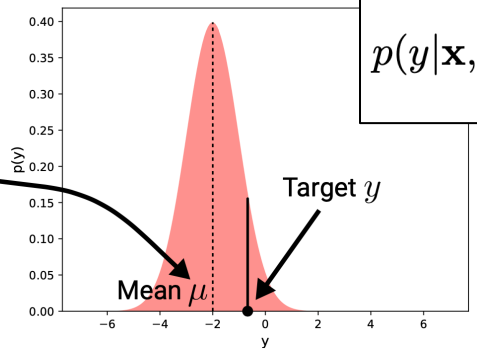
$$\hat{\mathbf{w}}_{ML} = \underset{\mathbf{w}}{\operatorname{argmax}}\ p_{model}(\mathbf{y}|\mathbf{X}, \mathbf{w}) \overset{\text{iid}}{=} \underset{\mathbf{w}}{\operatorname{argmax}} \prod_{i=1}^{N} p_{model}(y_i|\mathbf{x}_i, \mathbf{w})$$

$$= \underset{\mathbf{w}}{\operatorname{argmax}} \underbrace{\sum_{i=1}^{N} \log p_{model}(y_i|\mathbf{x}_i, \mathbf{w})}_{\textbf{\textit{Log-likelihood}}}$$

$\mathbf{X}$ is the entire dataset (columns features and raws i the data points, e.g. pixels)

- *Example*
  - Neural network $f_{\mathbf{w}}(\mathbf{x})$ predicts mean μ of Gaussian distribution over y:





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

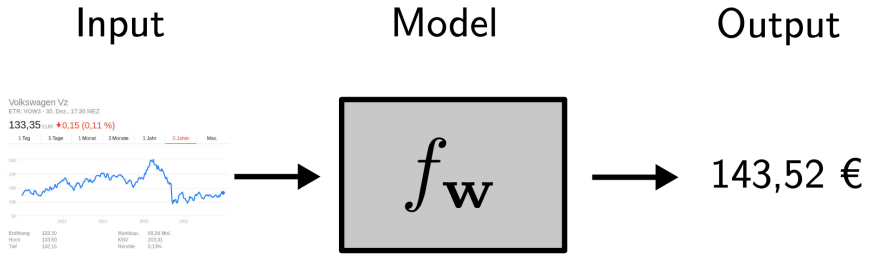
We want to maximize the probability of the target y under this distribution → chose the right distribution for the given problem

Task dependency of output and loss functions

- The selection of both the output layer function and of the loss function depend on the task at hand.
- Different tasks corresponds to different **I/O mapping functions** and to possible different ways to represent the network oputput and its probability distribution.
- For example, stock regression and image classification will lead to different choices

- **Stock Regression task**
  - Regression mapping

$$f_{\mathbf{w}} : \mathbb{R}^N \rightarrow \mathbb{R}$$

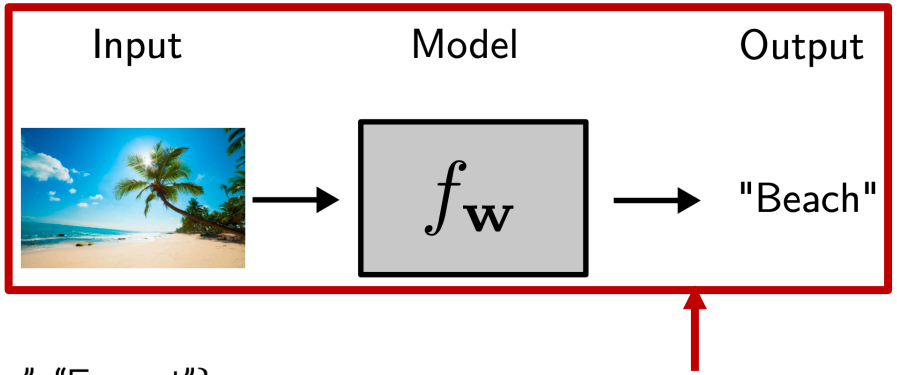


- **Image Classification task**
  - Binary classification mapping

$$f_{\mathbf{w}} : \mathbb{R}^{W \times H} \rightarrow \{\text{“Beach”}, \text{“No Beach”}\}$$

  - Multitask classification mapping

$$f_{\mathbf{w}} : \mathbb{R}^{W \times H} \rightarrow \{\text{“Beach”}, \text{“Mountain”}, \text{“City”}, \text{“Forest”}\}$$



In the following we only consider classification

Image classification

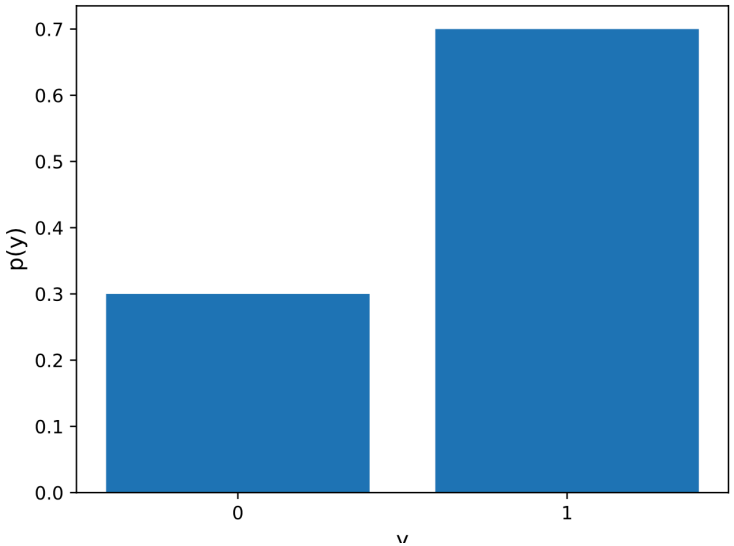
- Example (seen in the Lab): **MINST Handwritten digits**
  - One of the most popular datasets in ML (many variants, still in use today)
  - Based on a data from the National Institute of Standards and Technology
  - Hand written by Census Bureau employees and high-school children
  - **Resolution (i.e. problem dimensionality)**: 28 x 28 pixels, 60k training samples with labels, 10k test samples

- **«Curse of Dimensionality»:**
  - There exist $2^{784} = 10^{236}$ possible binary images of resolution 28 x 28 pixels
  - MNIST is gray-scale, thus 256^{784} combinations ⇒ impossible to enumerate
  - Why is image classification with just 60k labeled training images even possible?
  - Answer: Images concentrated on low-dimensional manifolds in → we do not need to search the entire space

Binary classification – Bernoulli distribution – BCE Loss

- **Bernoulli distribution**

$$p(y) = \mu^y \, (1-\mu)^{(1-y)}$$



- μ: probability for $y = 1$
- Handles only two classes
  e.g. (“cats” vs. “dogs”)

Let $p_{model}(y|\mathbf{x}, \mathbf{w}) = f_{\mathbf{w}}(\mathbf{x})^y \, (1 - f_{\mathbf{w}}(\mathbf{x}))^{(1-y)}$ be a **Bernoulli distribution**. We obtain:

$$\begin{aligned}
\hat{\mathbf{w}}_{ML} &= \underset{\mathbf{w}}{\operatorname{argmax}} \sum_{i=1}^{N} \log p_{model}(y_i|\mathbf{x}_i, \mathbf{w}) \\
&= \underset{\mathbf{w}}{\operatorname{argmax}} \sum_{i=1}^{N} \log \left[f_{\mathbf{w}}(\mathbf{x}_i)^{y_i} \, (1 - f_{\mathbf{w}}(\mathbf{x}_i))^{(1-y_i)} \right] \\
&= \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{N} \underbrace{-y_i \log f_{\mathbf{w}}(\mathbf{x}_i) - (1-y_i) \log(1 - f_{\mathbf{w}}(\mathbf{x}_i))}_{\text{BCE Loss}}
\end{aligned}$$

In other words, we minimize the **binary cross-entropy (BCE)** loss.

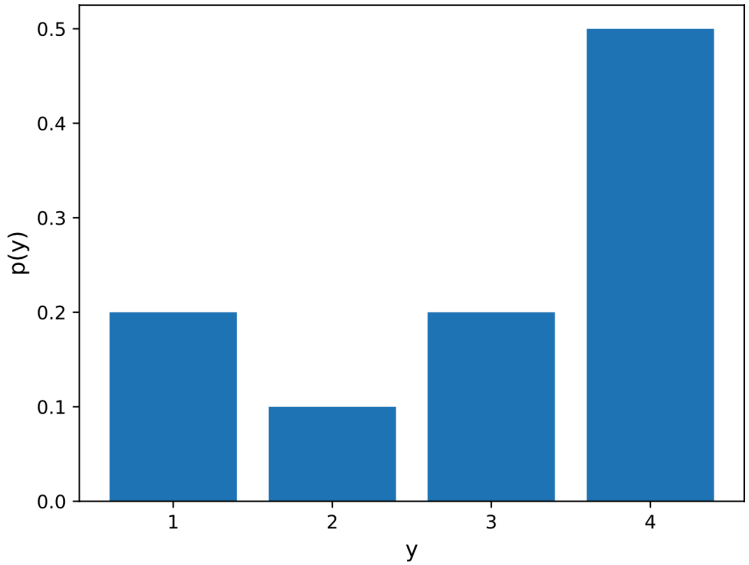
Remark: Last layer of $f_{\mathbf{w}}(\mathbf{x})$ can be a sigmoid function such that $f_{\mathbf{w}}(\mathbf{x})^y \in [0, 1]$.

Multiclass classification – Categorical distribution – CE Loss

Categorical distribution

$$p(y = c) = \mu_c$$

- μ_c: probability for class c
- Multiple classes, multiple modes



Alternative notation:

$$p(\mathbf{y}) = \prod_{c=1}^{C} \mu_c^{y_c}$$

- $\mathbf{y}$: "one-hot" vector with $y_c \in \{0, 1\}$
- $\mathbf{y} = (0, \ldots, 0, 1, 0, \ldots, 0)^\top$ with all zeros except for one (the true class)

- One-hot vector y with binary elements $y_c \in \{0,1\}$
- Index c with $y_c = 1$ determines the correct class, and $y_k = 0$ for $k \neq c$
- One-hot vectors can be seen as discrete distribution with all probability mass at the true class
- Often used in MaxLikelihood as it can make formalism more convenient (ground truth vectors similar to the discrete predicted output distribution)

class	y	Alternative notation $\mathbf{y}$
	1	$(1, 0, 0, 0)^\top$
	2	$(0, 1, 0, 0)^\top$
	3	$(0, 0, 1, 0)^\top$
	4	$(0, 0, 0, 1)^\top$

Multiclass classification – Categorical distribution – CE Loss

Categorical distribution

Let $p_{model}(\mathbf{y}|\mathbf{x}, \mathbf{w}) = \prod_{c=1}^{C} f_{\mathbf{w}}^{(c)}(\mathbf{x})^{y_c}$ be a **Categorical distribution**. We obtain:

$$\begin{aligned}
\hat{\mathbf{w}}_{ML} &= \operatorname*{argmax}_{\mathbf{w}} \sum_{i=1}^{N} \log p_{model}(\mathbf{y}_i|\mathbf{x}_i, \mathbf{w}) \\
&= \operatorname*{argmax}_{\mathbf{w}} \sum_{i=1}^{N} \log \prod_{c=1}^{C} f_{\mathbf{w}}^{(c)}(\mathbf{x}_i)^{y_{i,c}} \\
&= \operatorname*{argmin}_{\mathbf{w}} \sum_{i=1}^{N} \underbrace{\sum_{c=1}^{C} -y_{i,c} \log f_{\mathbf{w}}^{(c)}(\mathbf{x}_i)}_{\text{CE Loss}}
\end{aligned}$$

In other words, we minimize the **cross-entropy (CE)** loss.

The target $\mathbf{y} = (0, \ldots, 0, 1, 0, \ldots, 0)^\top$ is a "one-hot" vector with y_c its c'th element.

For the $(\mathbf{x}_i, y_i)$ pair:

Target

Prediction (for class *c*)

$$y_{i,c} \log f_{\mathbf{w}}^{(c)}(\mathbf{x}_i)$$

Multiclass classification – Categorical distribution – Softmax output

Softmax output

How can we ensure that $f_{\mathbf{w}}^{(c)}(\mathbf{x})$ predicts a **valid Categorical (discrete) distribution?**

- We must guarantee (1) $f_{\mathbf{w}}^{(c)}(\mathbf{x}) \in [0, 1]$ and (2) $\sum_{c=1}^{C} f_{\mathbf{w}}^{(c)}(\mathbf{x}) = 1$
- An element-wise sigmoid as output function would ensure (1) but not (2)
- Solution: The **softmax function** guarantees both (1) and (2):

$$\text{softmax}(\mathbf{x}) = \left(\frac{\exp(x_1)}{\sum_{k=1}^{C} \exp(x_k)}, \dots, \frac{\exp(x_C)}{\sum_{k=1}^{C} \exp(x_k)} \right)$$

- Let $\mathbf{s}$ denote the network output after the last affine layer (=scores). Then:

$$f_{\mathbf{w}}^{(c)}(\mathbf{x}) = \frac{\exp(s_c)}{\sum_{k=1}^{C} \exp(s_k)} \quad \Rightarrow \quad \log f_{\mathbf{w}}^{(c)}(\mathbf{x}) = s_c - \log \sum_{k=1}^{C} \exp(s_k)$$

- Remark: s_c is a direct contribution to the loss function, i.e., it does not saturate

Multiclass classification – Categorical distribution – Softmax output

Log Softmax

Intuition: Assume c is the correct class. Our goal is to maximize the log softmax:

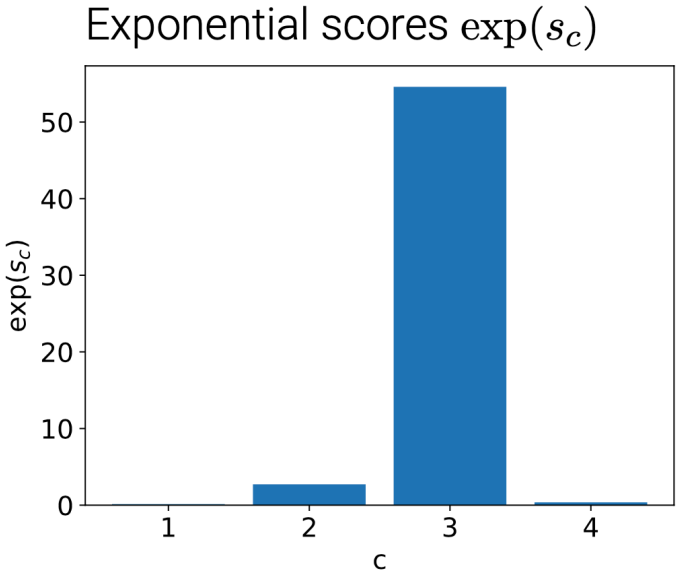
$$\log f_{\mathbf{w}}^{(c)}(\mathbf{x}) = s_c - \log \sum_{k=1}^{C} \exp(s_k)$$

- The first term encourages the score s_c for the correct class c to increase
- The second term encourages all scores in $\mathbf{s}$ to decrease
- The second term can be approximated by: $\log \sum_{k=1}^{C} \exp(s_k) \approx \max_k s_k$ as $\exp(s_k)$ is insignificant for all $s_k < \max_k s_k$
- Thus, the loss always strongly penalizes the most active incorrect prediction
- If the correct class already has the largest score (i.e., $s_c = \max_k s_k$), both terms roughly cancel and the example will contribute little to the overall training cost

Multiclass classification – Categorical distribution – Softmax output

- **Log Softmax Example**

Predicted classes



- The second term becomes: $\log \sum_{k=1}^{C} \exp(s_k) = 4.06 \approx s_3 = \max_k s_k$
- For $c = 2$ we obtain: $\log f_{\mathbf{w}}^{(c)}(\mathbf{x}) = s_c - \log \sum_{k=1}^{C} \exp(s_k) = 1 - 4.06 \approx -3$
- For $c = 3$ we obtain: $\log f_{\mathbf{w}}^{(c)}(\mathbf{x}) = s_c - \log \sum_{k=1}^{C} \exp(s_k) = 4 - 4.06 \approx 0$

Correct class hypothesis

Multiclass classification – Categorical distribution – Softmax output

- **Softmax:** minimal vs overparametrized versions
  - Predicting C values/scores overparameterizes the Categorical distribution
  - As the distribution sums to 1 only $C-1$ parameters are necessary
  - Example: Consider $C = 2$ and fix one degree of freedom ($x_2 = 0$):

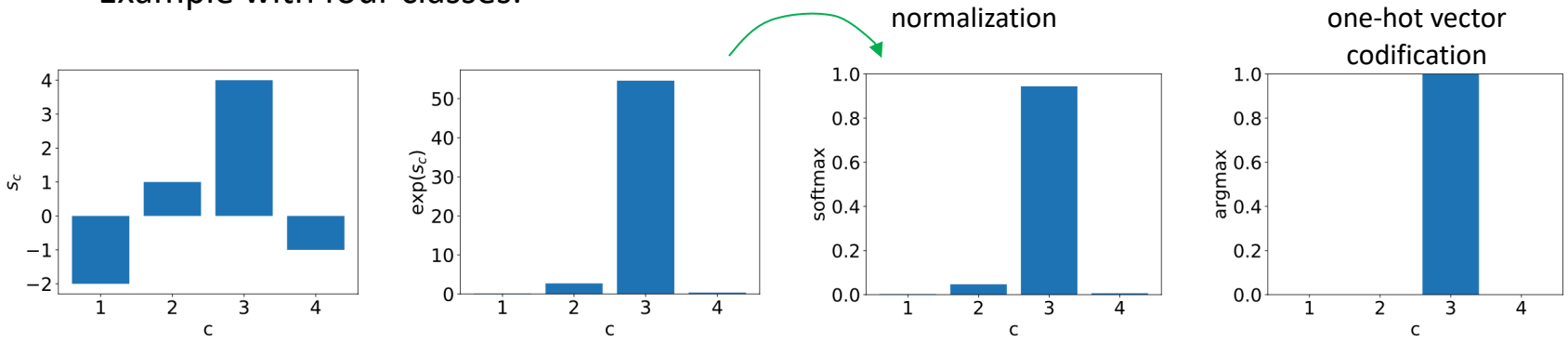
$$\begin{aligned}
\text{softmax}(\mathbf{x}) &= \left(\frac{\exp(x_1)}{\exp(x_1)+\exp(x_2)}, \frac{\exp(x_2)}{\exp(x_1)+\exp(x_2)} \right) \\
&= \left(\frac{\exp(x_1)}{\exp(x_1)+1}, \frac{1}{\exp(x_1)+1} \right) \\
&= \left(\frac{1}{1+\exp(-x_1)}, 1 - \frac{1}{1+\exp(-x_1)} \right) \\
&= (\sigma(x_1), 1-\sigma(x_1))
\end{aligned}$$

  - The softmax is a multi-class generalization of the sigmoid function
  - In practice, the overparameterized version is often used (simpler to implement)

$$\text{softmax}(\mathbf{s}) = \left(\frac{\exp(s_1)}{\sum_{k=1}^{C}\exp(s_k)}, \cdots, \frac{\exp(s_C)}{\sum_{k=1}^{C}\exp(s_k)} \right)$$

Multiclass classification – Categorical distribution – Softmax output

- **Softmax:** considerations
  - The name “softmax” is confusing, “soft argmax” would be more precise as it is a continuous and differentiable version of argmax (in one-hot representation)
  - Example with four classes:



  - Softmax responds to differences between inputs
  - It is invariant to adding the same scalar to all its inputs:

$$\mathrm{softmax}(\mathbf{x}) = \mathrm{softmax}(\mathbf{x} + c)$$

  - We can therefore derive a numerically more stable variant:

$$\mathrm{softmax}(\mathbf{x}) = \mathrm{softmax}(\mathbf{x} - \max_{k=1..L} x_k)$$

  - Allows accurate computation even when $\mathbf{x}$ is large
  - Illustrates again that softmax depends on differences between scores

Example with Cross Entropy Loss and Softmax output

Putting it together: Cross Entropy Loss for a single training sample $(\mathbf{x}, \mathbf{y}) \in \mathcal{X}$:

$$\text{CE Loss: } \sum_{c=1}^{C} -y_c \log f_{\mathbf{w}}^{(c)}(\mathbf{x})$$

Example: Suppose $C = 4$ and 4 training samples $\mathbf{x}$ with labels $\mathbf{y}$

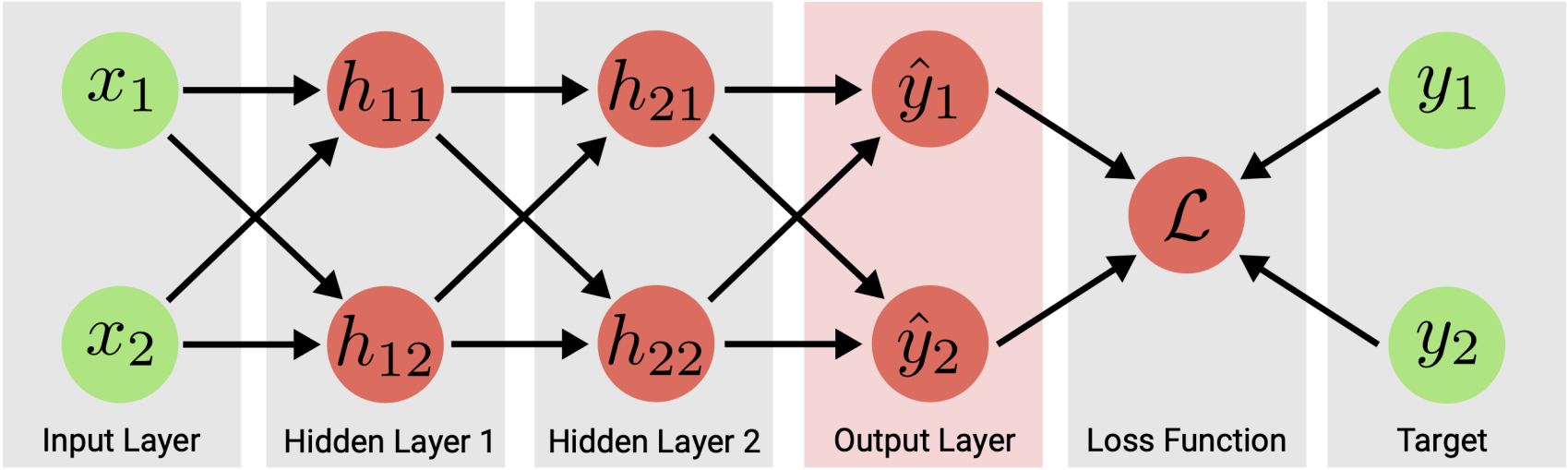
Input $\mathbf{x}$	Label $\mathbf{y}$	Predicted scores $\mathbf{s}$	softmax($\mathbf{s}$)	CE Loss
	$(1, 0, 0, 0)^\top$	$(+3, +1, -1, -1)^\top$	$(0.85, 0.12, 0.02, 0.02)^\top$	0.16
	$(0, 1, 0, 0)^\top$	$(+3, +3, +1, +0)^\top$	$(0.46, 0.46, 0.06, 0.02)^\top$	0.78
	$(0, 0, 1, 0)^\top$	$(+1, +1, +1, +1)^\top$	$(0.25, 0.25, 0.25, 0.25)^\top$	1.38
	$(0, 0, 0, 1)^\top$	$(+3, +2, +3, -1)^\top$	$(0.42, 0.16, 0.42, 0.01)^\top$	4.87

- Sample 4 contributes most strongly to the loss function! (elephant in the room)

Summary: classification deep neural networks

- **Output layer** for classification tasks
  - For 2 classes, we can predict 1 value and use a sigmoid, or 2 values with softmax
  - For $C > 2$ classes we typically predict C scores and use a softmax non-linearity

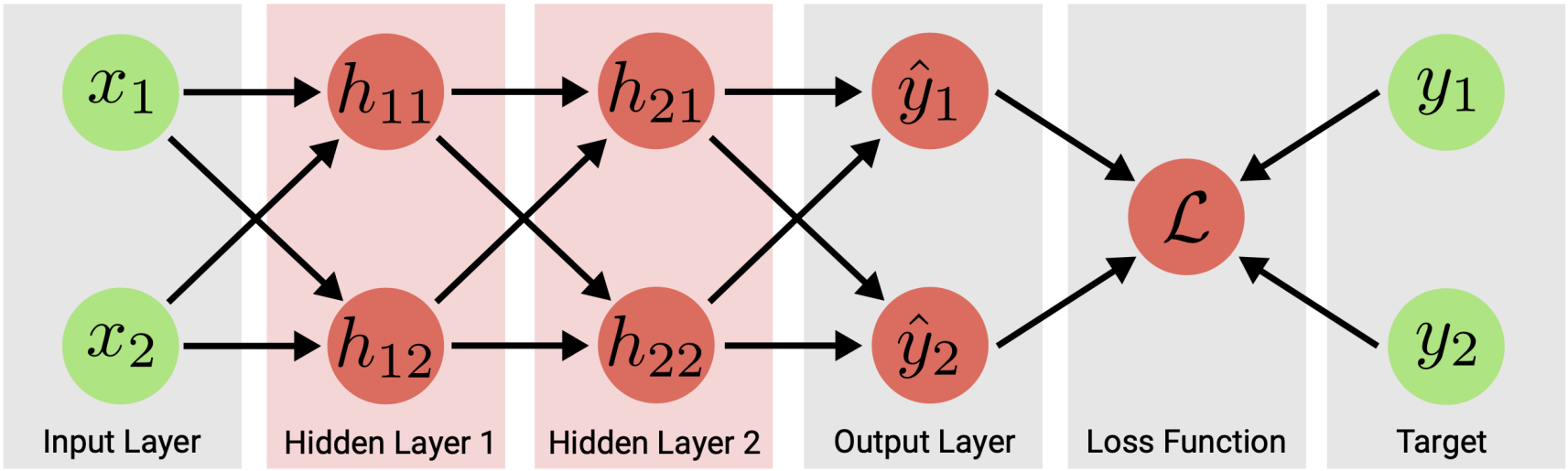
- **Loss Function** for Classification tasks
  - For 2 classes, we use the binary cross-entropy loss (BCE)
  - For $C > 2$ classes, we use the cross-entropy loss (CE)



ACTIVATION FUNCTIONS

Activation functions

- Hidden layer $\mathbf{h}_i = g(\mathbf{A}_i \mathbf{h}_{i-1} + \mathbf{b}_i)$ with **activation function** $g(\cdot)$ and weights $\mathbf{A}_i, \mathbf{b}_i$
- The activation function is frequently applied **element-wise** to its input
- Activation functions must be **non-linear** to learn non-linear mappings
- Some of them are not differentiable everywhere (but still ok for training)



Sigmoid

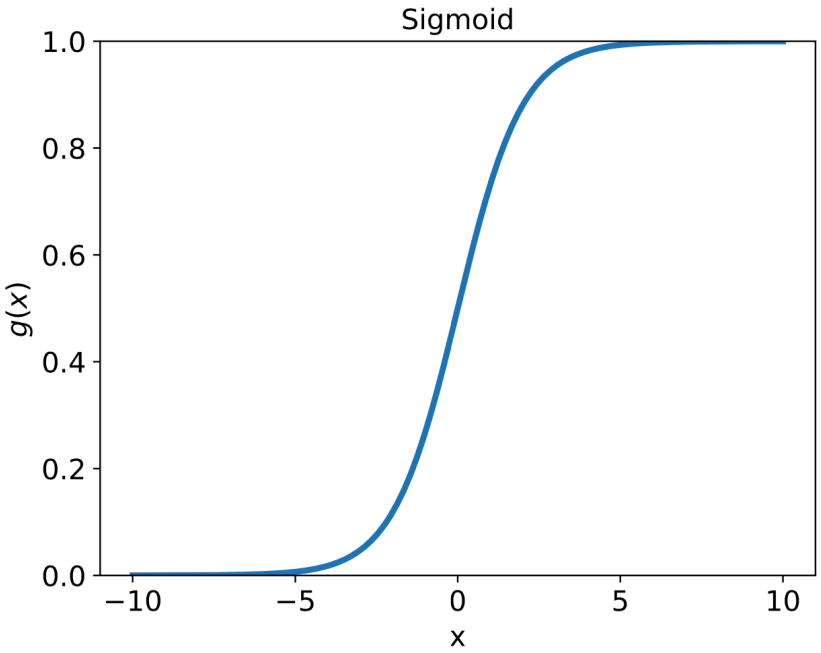
Sigmoid:

$$g(x) = \frac{1}{1 + \exp(-x)}$$

- Maps input to range $[0, 1]$
- Neuroscience interpretation as saturating “firing rate” of neurons

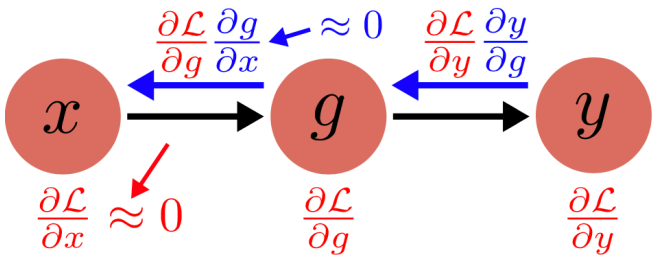
Problems:

- Saturation “kills” gradients
- Outputs are not zero-centered
- Introduces bias after first layer



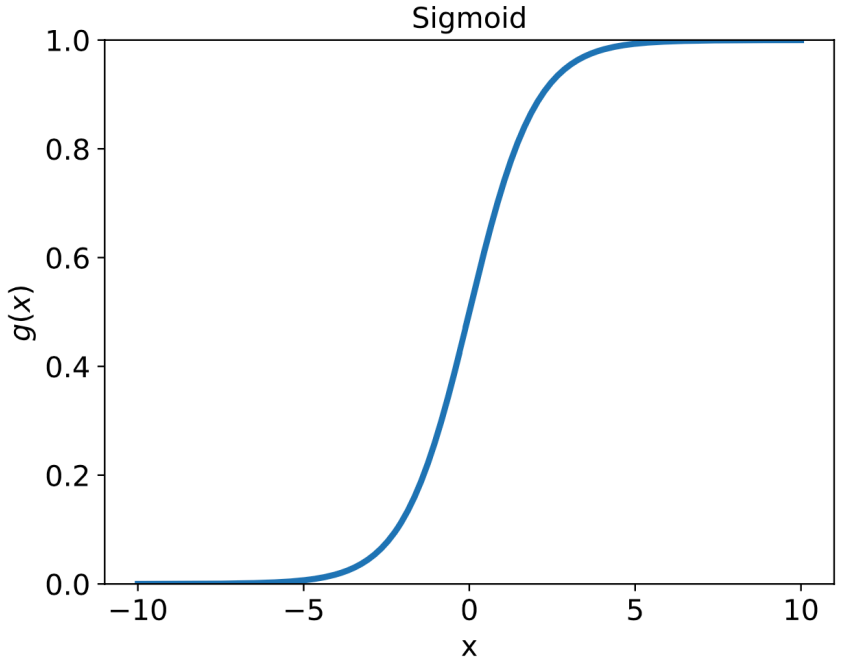
Sigmoid

Sigmoid Problem #1:



- Downstream gradient becomes zero when input x is saturated: $g'(x) \approx 0$
- No learning if x is very small (< -10)
- No learning if x is very large (> 10)

input to g



Sigmoid

Sigmoid Problem #2:

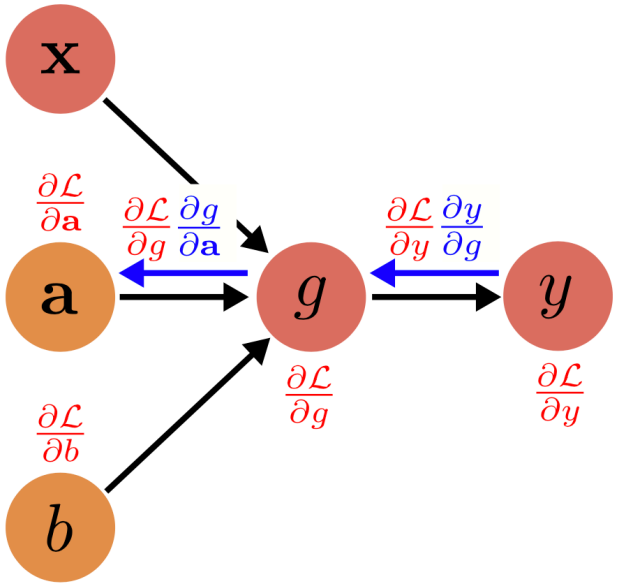
$$g(x) = \frac{1}{1+\exp(-x)} \qquad x = \sum_i a_i x_i + b$$

- Sigmoid is always positive $\Rightarrow x_i$ also
- Gradient of sigmoid is always positive

The gradient wrt. parameter a_i is given by:

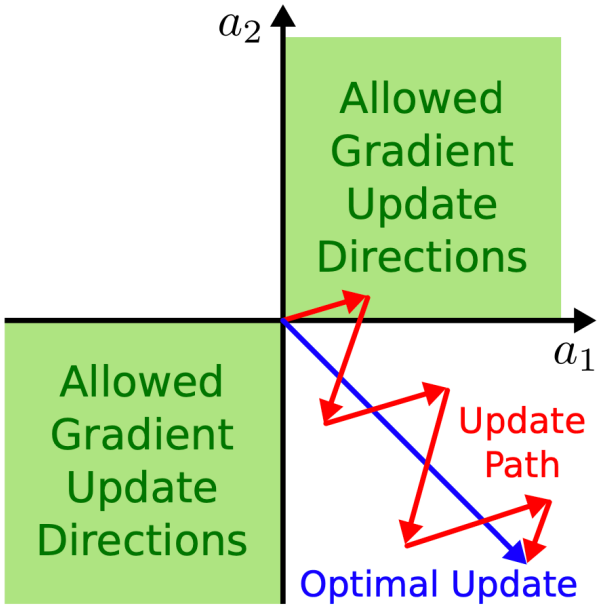
$$\frac{\partial \mathcal{L}}{\partial a_i} = \frac{\partial \mathcal{L}}{\partial g}\frac{\partial g}{\partial a_i} = \frac{\partial \mathcal{L}}{\partial g}\frac{\partial g}{\partial x}\frac{\partial x}{\partial a_i} = \frac{\partial \mathcal{L}}{\partial g}\frac{\partial g}{\partial x} x_i$$

- Therefore: $\text{sgn}(\partial \mathcal{L}/\partial a_i) = \text{sgn}(\partial \mathcal{L}/\partial g)$
- All gradients have the same sign (+ or -)

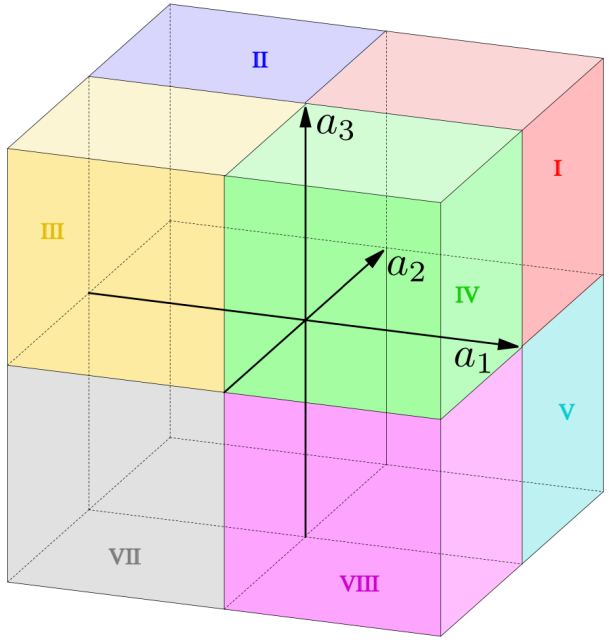


Sigmoid

Sigmoid Problem #2:



Only red and grey quadrants available out of 8



- Restricts gradient updates and leads to inefficient optimization (minibatches help)

Tanh

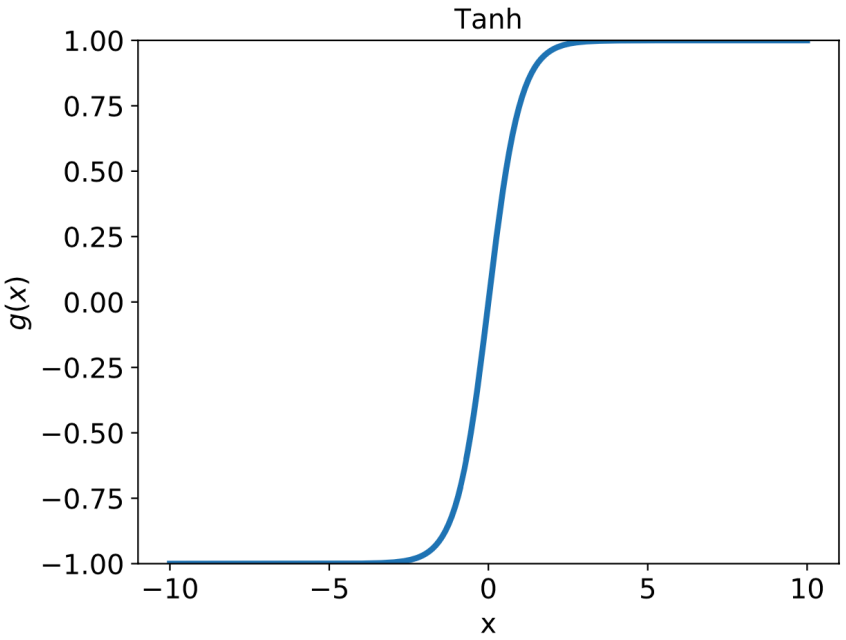
Tanh:

$$g(x) = \frac{2}{1 + \exp(-2x)} - 1$$

- Maps input to range $[-1, 1]$
- Anti-symmetric
- Zero-centered

Problems:

- Again, saturation “kills” gradients



Rectified Linear Unit (ReLU)

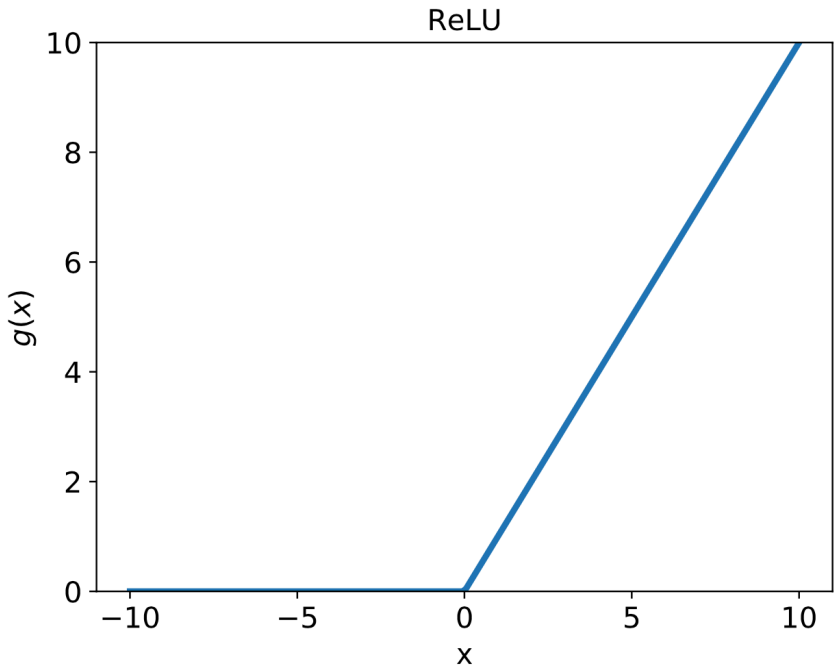
Rectified Linear Unit (ReLU):

$$g(x) = \max(0, x)$$

- Does not saturate (for $x > 0$)
- Leads to fast convergence
- Computationally efficient

Problems:

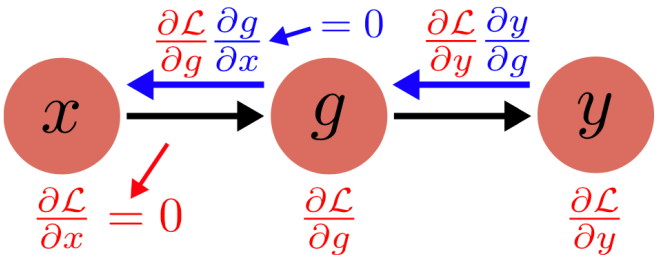
- Not zero-centered
- No learning for $x < 0 \Rightarrow$ dead ReLUs



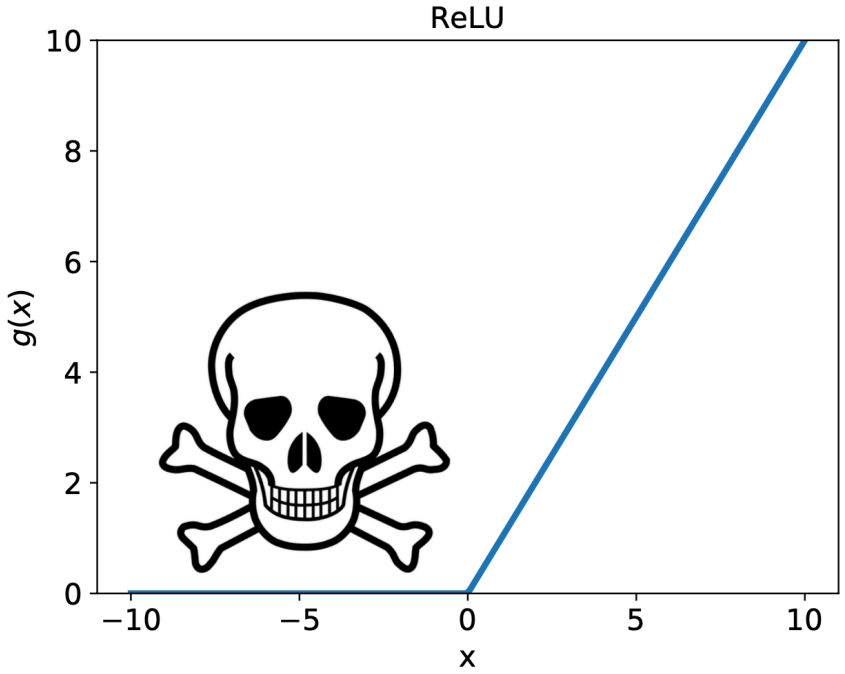
Less inspired by neuroscience but
works very well most of time
(most commonly used today)

Rectified Linear Unit (ReLU)

ReLU Problem:



- Downstream gradient becomes zero when input $x < 0$
- Results in so-called "dead ReLUs" that never participate in learning
- Often initialize with pos. bias ($b > 0$)

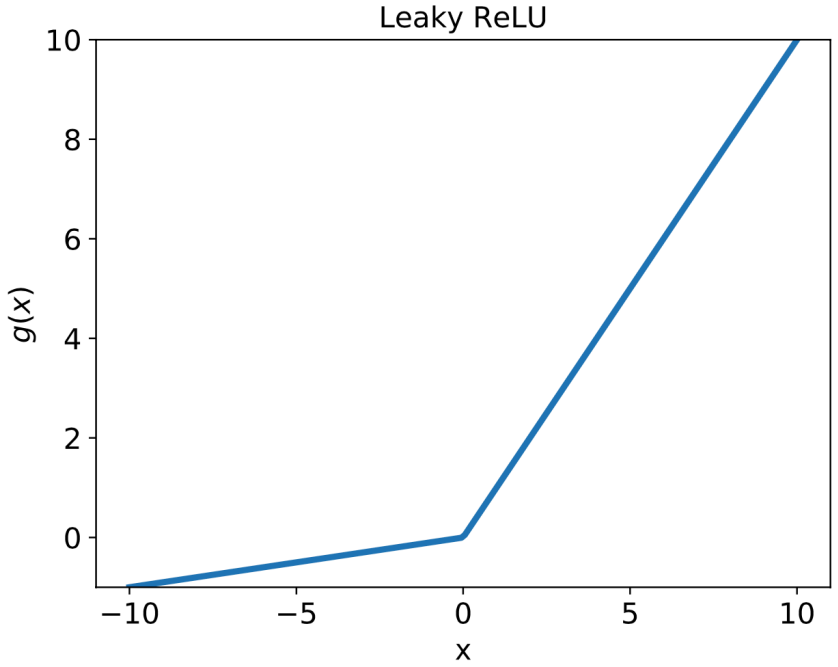


Leaky ReLU

Leaky ReLU:

$$g(x) = \max(0.01x, x)$$

- Does not saturate (i.e., will not die)
- Closer to zero-centered outputs
- Leads to fast convergence
- Computationally efficient

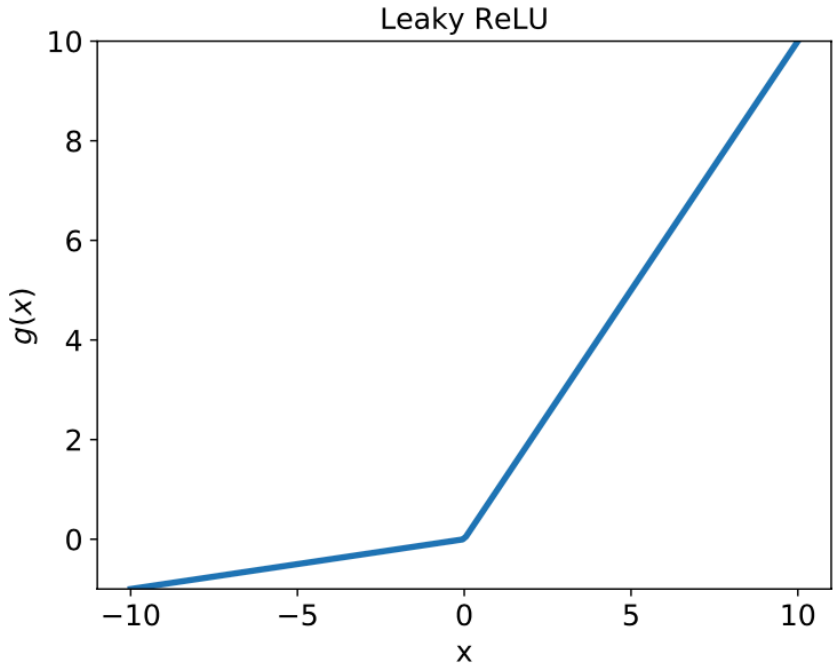


Parametric ReLU

Parametric ReLU:

$$g(x) = \max(\alpha x, x)$$

- Does not saturate (i.e., will not die)
- Leads to fast convergence
- Computationally efficient
- Parameter α learned from data

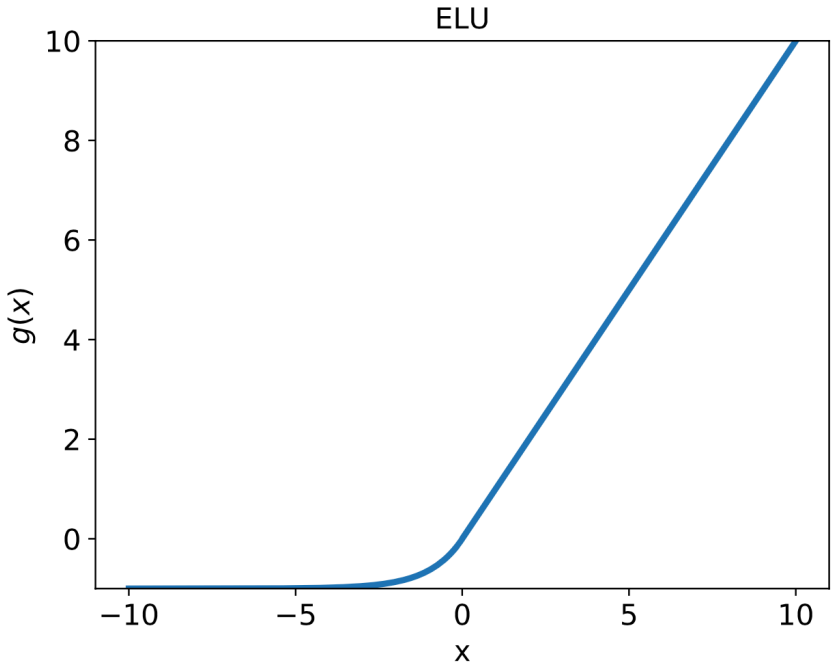


Exponential Linear Units

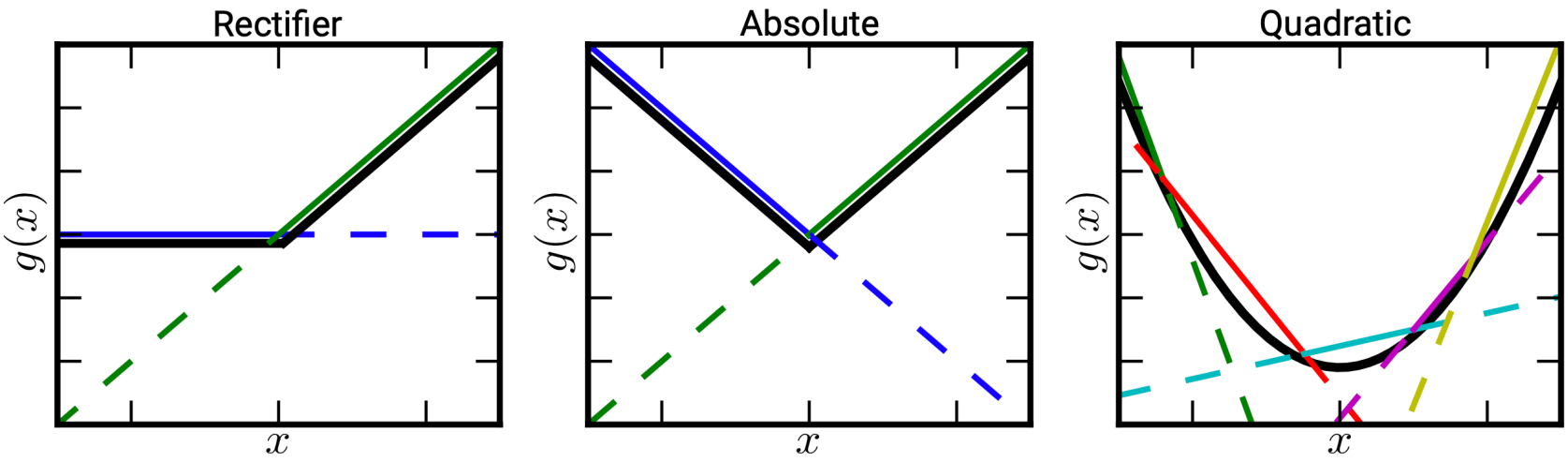
Exponential Linear Units (ELU):

$$g(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

- All benefits of Leaky ReLU
- Adds some robustness to noise
- Default $\alpha = 1$



Maxout



Maxout: $g(x) = \max(\mathbf{a}_1^\top \mathbf{x} + b_1, \mathbf{a}_2^\top \mathbf{x} + b_2)$

- Generalizes ReLU and Leaky ReLU
- Increases the number of parameters per neuron

Summary: activation functions

- No one-size-fits-all: Choice of activation function depends on problem
- We only showed the most common ones, there exist many more
- Best activation function/model is often found using trial-and-error in practice
- It is important to ensure a good “gradient flow” during optimization

Rule of Thumb:

- Use ReLU by default (with small enough learning rate)
- Try Leaky ReLU, Maxout, ELU for some small additional gain
- Prefer Tanh over Sigmoid (Tanh often used in recurrent models)