Karlsruher Institut für Technologie

Machine Learning 1 – Fundamentals

Neural Networks - Basics

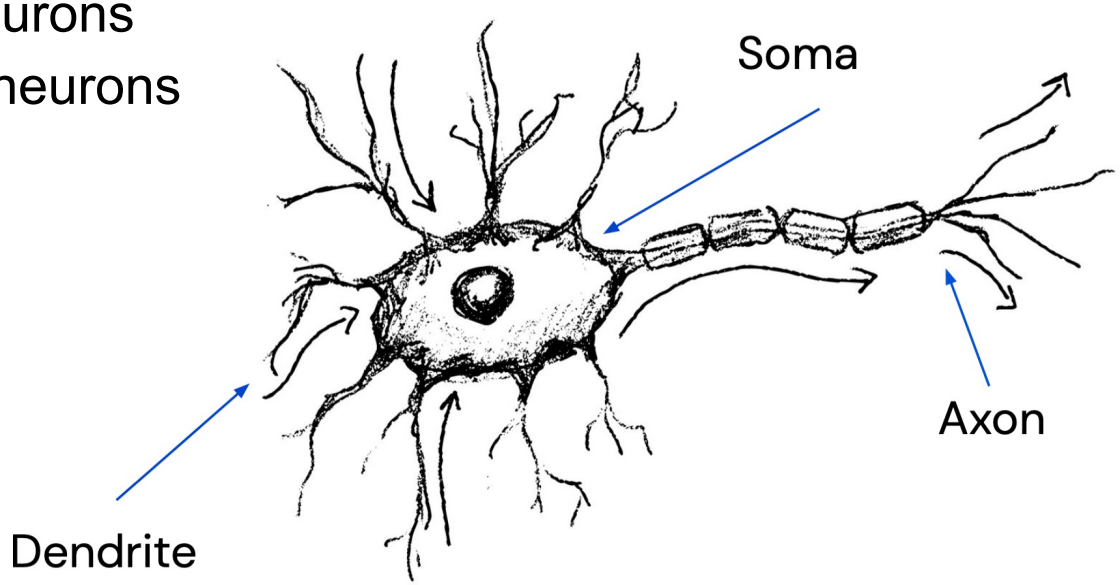
Prof. Dr. J. M. Zöllner, M.Sc. Marcus Fechner, M.Sc. Nikolai Polley

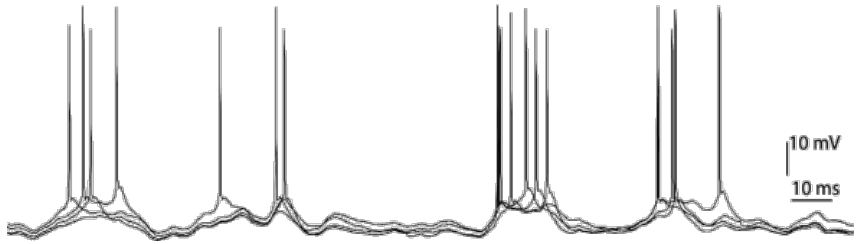
Outline

- Introduction
- Perceptron
- Deep Neural Networks
- Learning
- Applications
- Literature

Real Neuron

- Human brain contains around 86.000.000.000 neurons
- Each neuron is connected to thousands of other neurons
- Represents simple computation
- Distributed representation of knowledge
- Parallel computation
- Has inhibition and excitation connections
- Has a state
- Outputs spikes



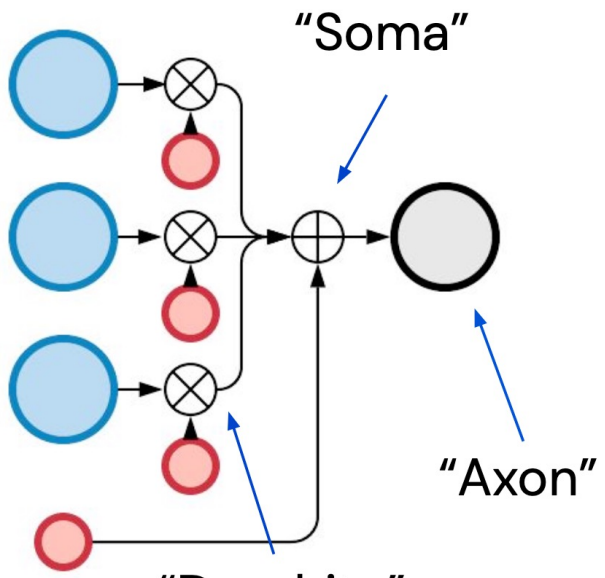


Artificial Neuron

- **Goal**: Model some of the neurophysiological properties of the real neuron, but not exactly reproduce them.

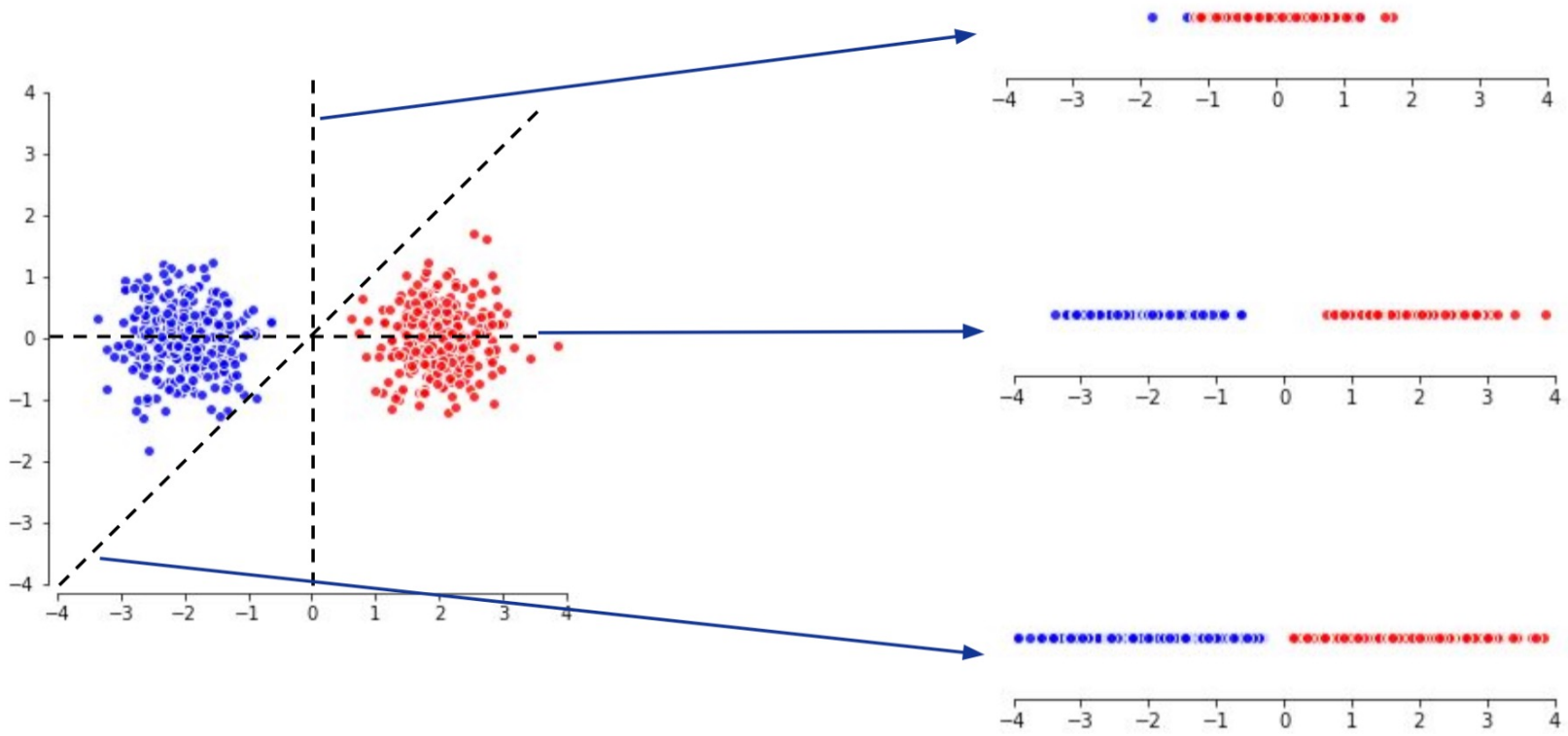
Properties

- Easy to compose ✅ 容易组合
- Represent simple computation ✅ 容易计算
- Has inhibition and excitation connections ✅ 有抑制和激励连接
- Is stateless with respect to time ❌
- Outputs real values ❌



$$\sum_{i=1}^{d} \mathbf{w}_i \mathbf{x}_i + b = \sum_{i=0}^{d} \mathbf{w}_i \mathbf{x}_i \qquad \mathbf{x}_0 := 1$$

Artificial Neuron – Geometric Interpretation



加权平均值定义了输入数据的仿射投影

- Weighted average defines an affine projection of the input data
- Depending on the weights, the projection changes

Linear Layer

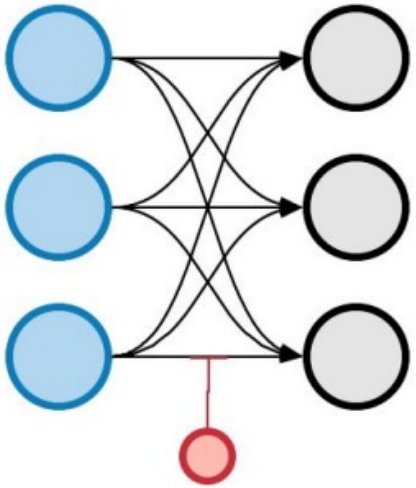
- Collection of artificial neurons
- Can be efficiently vectorized
- Efficiently computable by GPUs and TPUs

- **Despite the name, it's actually an affine transformation!**

尽管名字如此，它实际上是一种仿射变换！

Terminology

- Neurons in a layer are often called **units**
- Parameters are often called **weights**



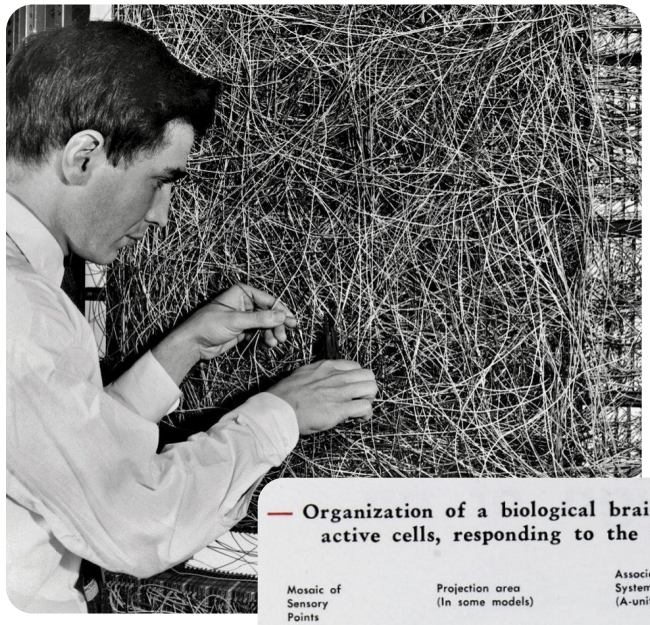
$$f(x) = Wx + b$$

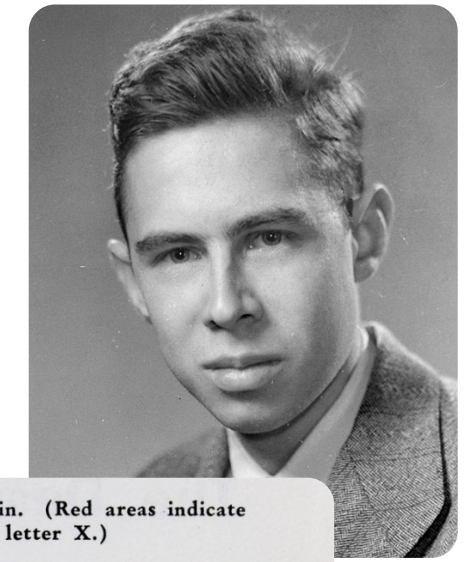
Outline

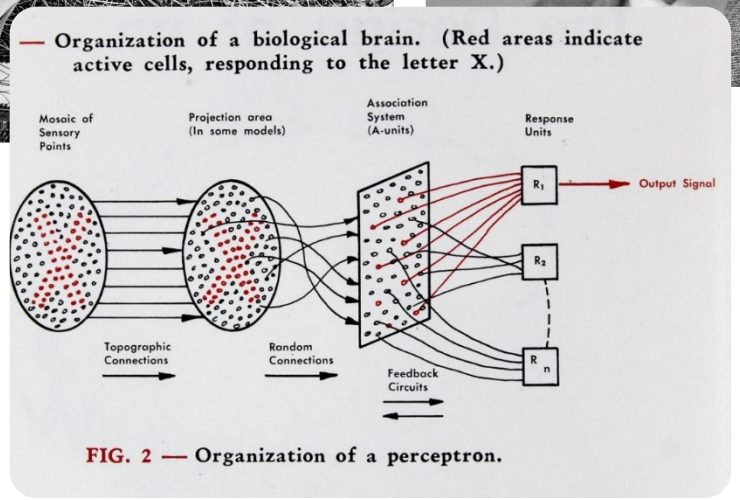
- Introduction
- Perceptron
- Deep Neural Networks
- Learning
- Applications
- Literature

Rosenblatt Perceptron

- Developed by Frank Rosenblatt in 1957
- Artificial neuron model with update rule for weights
- Laid the fundamentals for neural network development

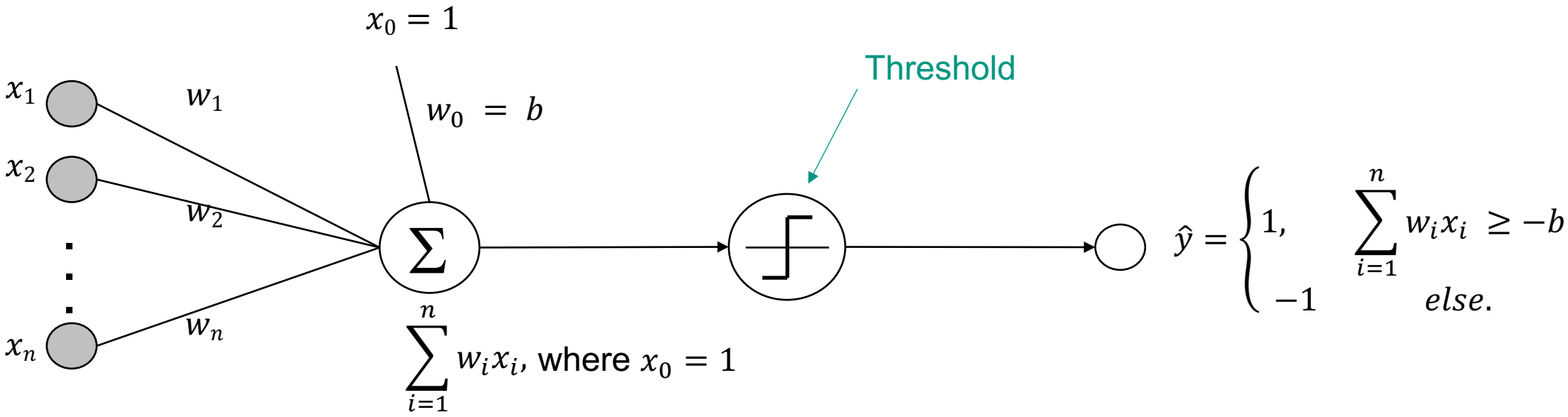






Rosenblatt Perceptron

$\boldsymbol{x}$ = Input vector

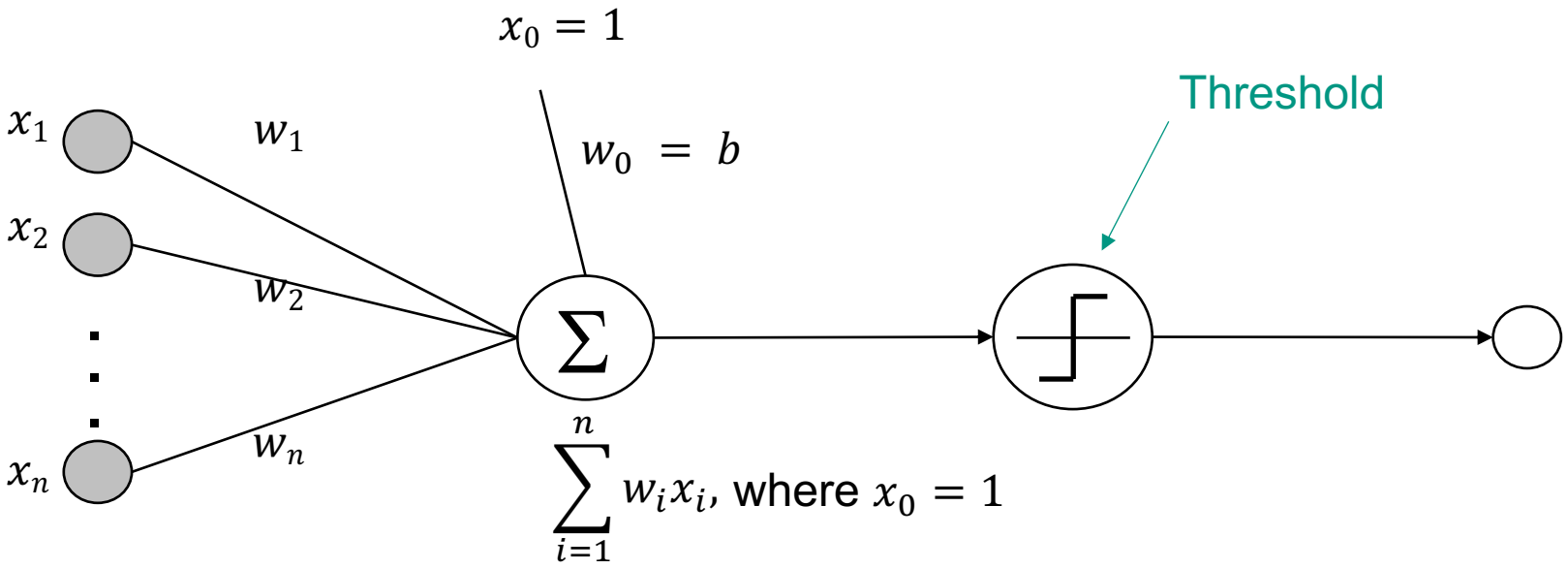


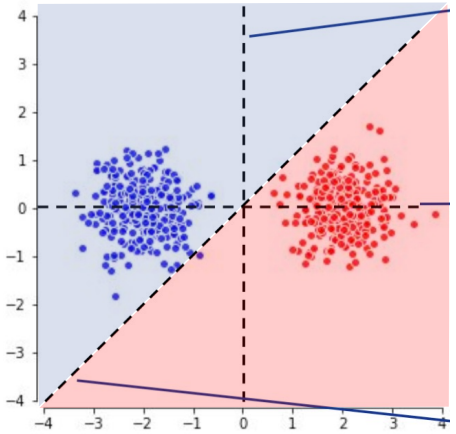
$\boldsymbol{w}$ = Weight vector

$\hat{y}$ = Predicted output

Rosenblatt Perceptron

$\boldsymbol{x}$ = Input vector



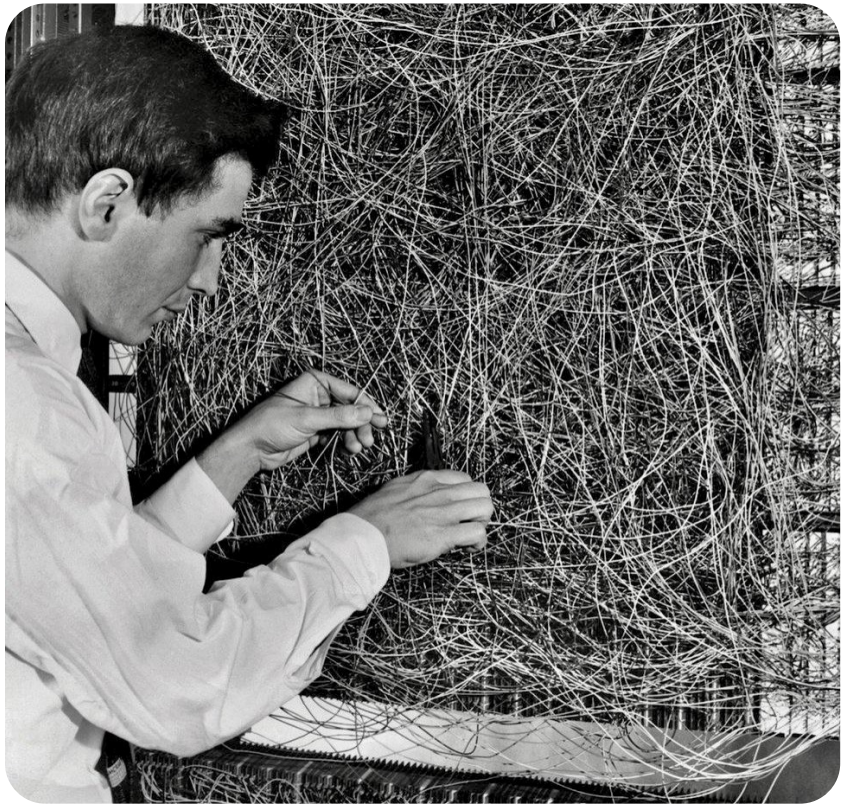


$\boldsymbol{w}$ = Weight vector

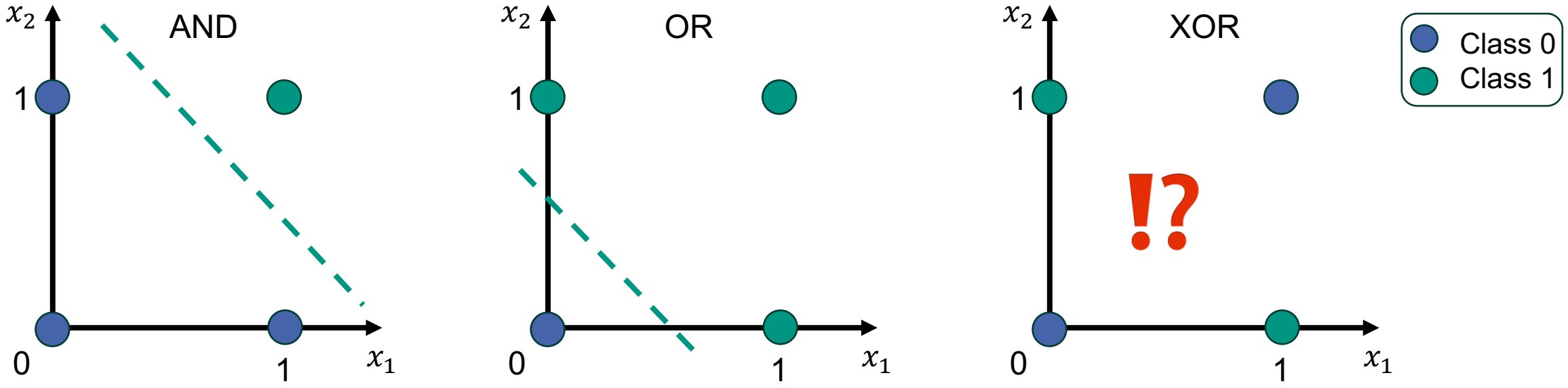
$\hat{y}$ = Predicted output

The XOR Problem

- Linear classification showed some promising results in the 50s and 60s on simple classification problems (Perceptron)
- But limitations became clear very soon (Minsky and Papert – Perceptrons)
- XOR problem is simple, but can’t be solved by linear models, as model capacity is limited
- Consequently neural network research declined in the 70s



The XOR Problem

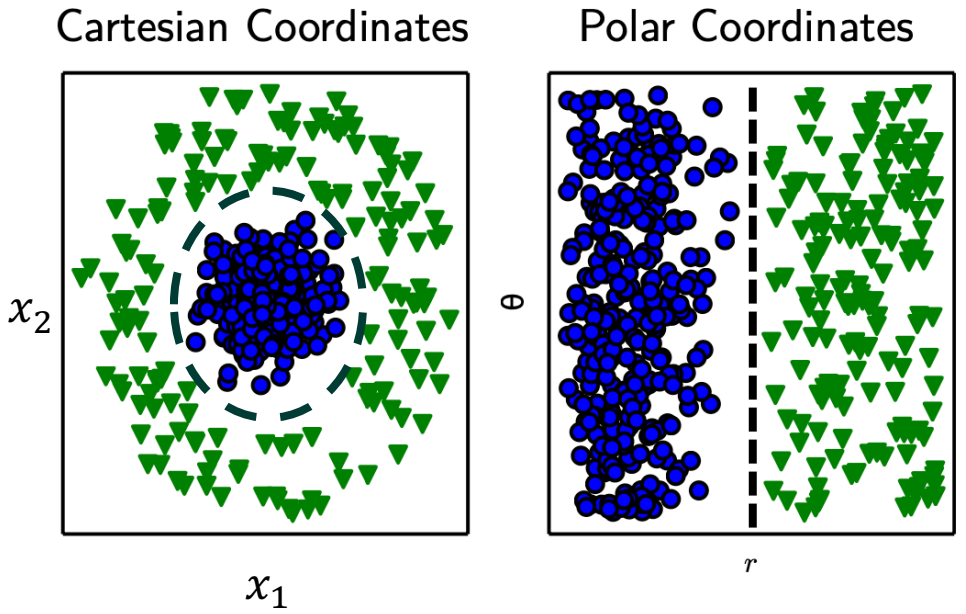


- Visually it is obvious, that the XOR problem is not linearly separable
- XOR problem is simple, but cannot be solved, as model capacity is limited to linear decisions
- How can we solve non-linear problems?

The XOR Problem

Representation Matters

- The right representation can transform a complex problem to a simple one
- Example
  - $f()$ is a linear classifier
  - $\hat{y} = f(x_1, x_2)$, not linear separable
  - $\hat{y} = f(r(x_1, x_2), \theta(x_1, x_2))$, linear separable
- Linear separation in transformed space leads to complex separation in origin space
- Known as **feature engineering**



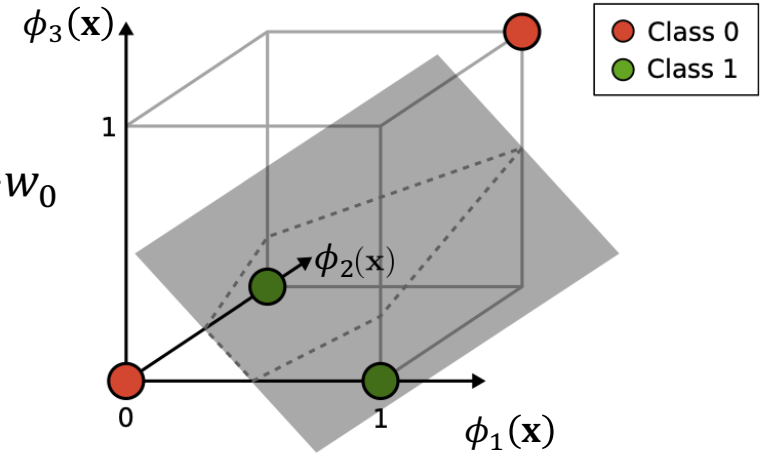
Representation Matters

- Linear classifier with non-linear features $\phi(x)$
- Non-linear features allow perceptron to solve XOR-problem

- But how to choose the transformation? **Very hard** in practice!
- Dominant approach before deep learning

- In this lecture we learn the representation
  - **Representation learning**
- Choose the right function family (network architecture) instead of the correct function

$$\mathbf{w}^\top \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_1 x_2 \end{pmatrix}}_{\phi(\mathbf{x})} > -w_0$$



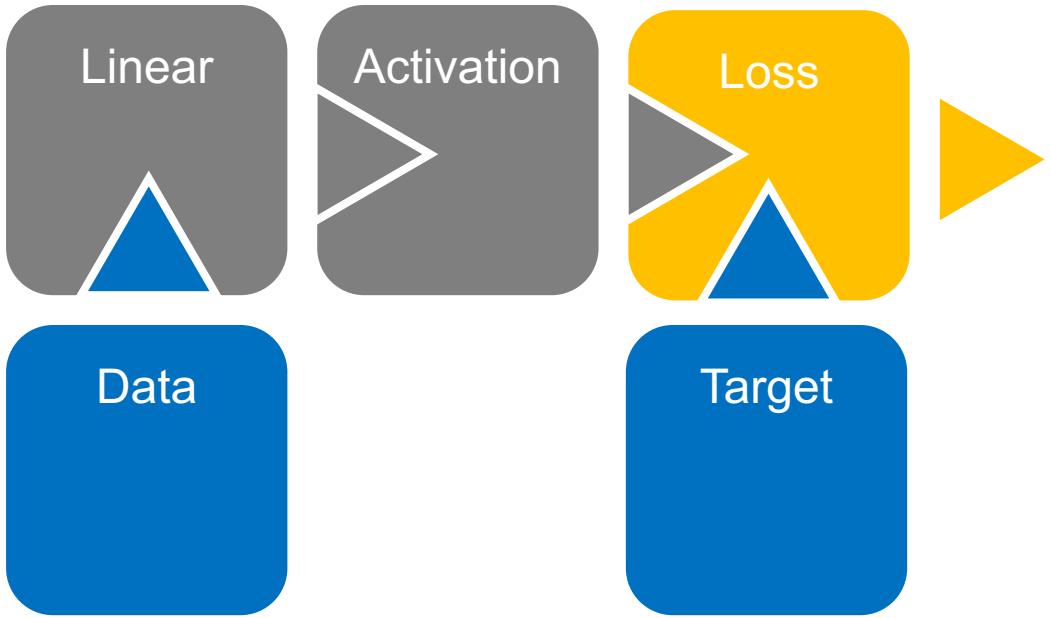
x_1	x_2	$\phi_1(\mathbf{x})$	$\phi_2(\mathbf{x})$	$\phi_3(\mathbf{x})$	XOR
0	0	0	0	0	0
0	1	0	1	0	1
1	0	1	0	0	1
1	1	1	1	1	0

Outline

- Introduction
- Perceptron
- Deep Neural Networks
- Learning
- Applications
- Literature

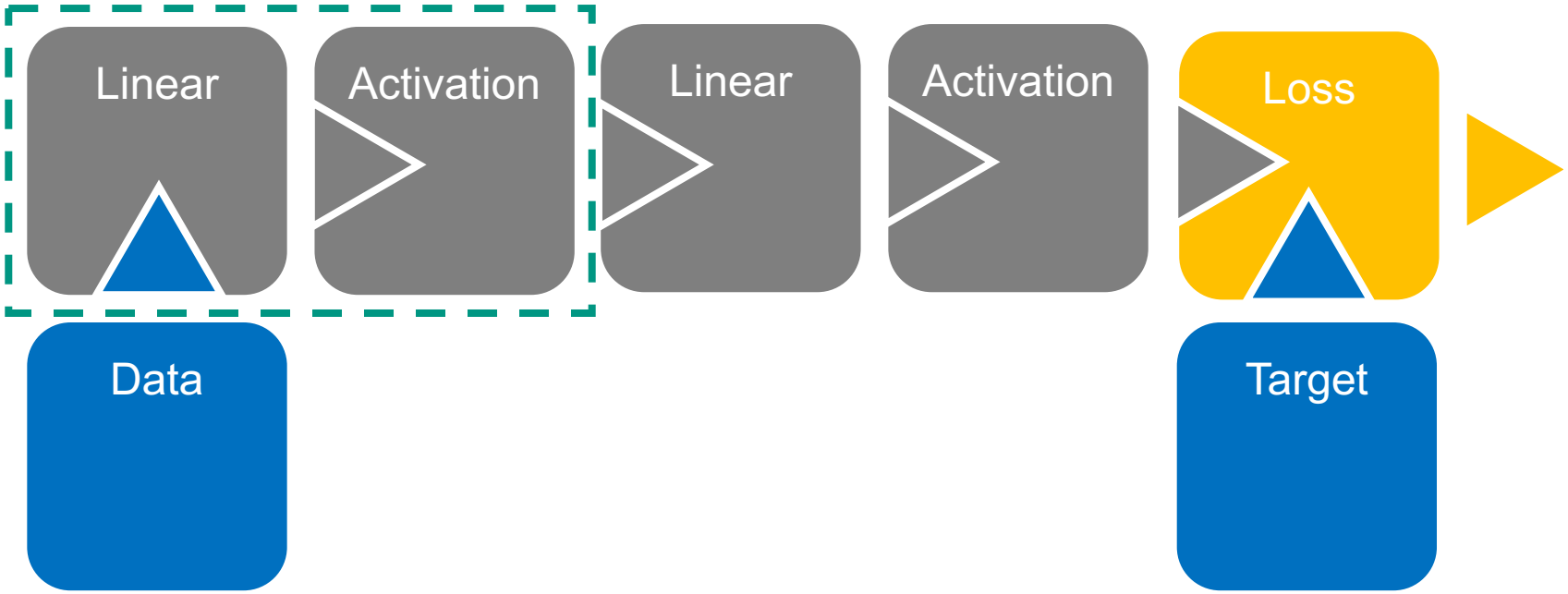
Single-Layer Neural Network

- **Previously**: Neural networks with a single layer (and multiple units)



Multi-Layer Neural Network

- **Now**: Compose multiple linear layers and nonlinearities
- **Example**: Neural network with two layers (one hidden layer)



Multi-Layer Neural Network

- Composes multiple nonlinear functions $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{a}_L\left(\boldsymbol{h}_L\left(\ldots \boldsymbol{a}_1\left(\boldsymbol{h}_1(\boldsymbol{x})\right)\right)\right)$
  - $\boldsymbol{h}_l = \boldsymbol{W}_l \boldsymbol{a}_{l-1} + \boldsymbol{b}_1$
  - $\boldsymbol{a}_l = \boldsymbol{\sigma}(\boldsymbol{h}_l)$

- **Terminology**
  - **Input layer**: First layer of a NN
  - **Hidden layer**: Intermediate layer between input and output layer. Not directly observable.
  - **Output layer**: Final layer of a NN
- Each layer l comprises multiple neurons/units
- Each neuron in each layer is **fully connected to all neurons of the previous layer**
- The overall number of layers L is synonymous for the depth of the model, hence the name “Deep Learning"

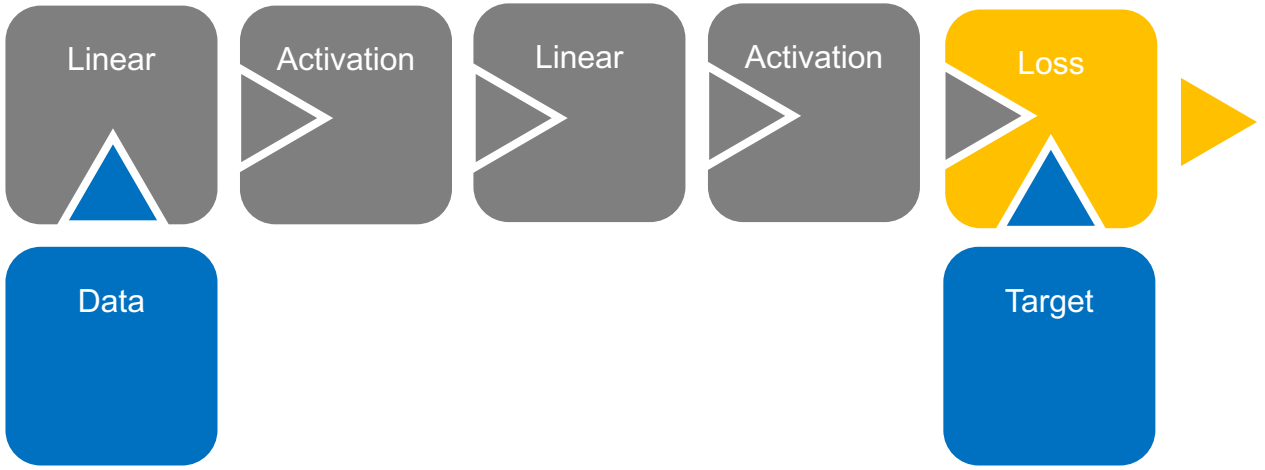
Multi-Layer Neural Network

Example: Neural network with two layers and two units per layer.

- Input layer: $\boldsymbol{x}$
- Hidden layer: $\boldsymbol{a}_1 = \boldsymbol{\sigma}(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1)$
- Output layer: $\hat{\boldsymbol{y}} = \boldsymbol{\sigma}(\boldsymbol{W}_2\boldsymbol{a}_1 + \boldsymbol{b}_2)$
- **Note:** Weights $\boldsymbol{W}, \boldsymbol{b}$ and any activation function $\boldsymbol{\sigma}()$

$$\boldsymbol{W_i} = \begin{bmatrix} w_{11}^l & w_{12}^l \\ w_{21}^l & w_{22}^l \end{bmatrix}$$

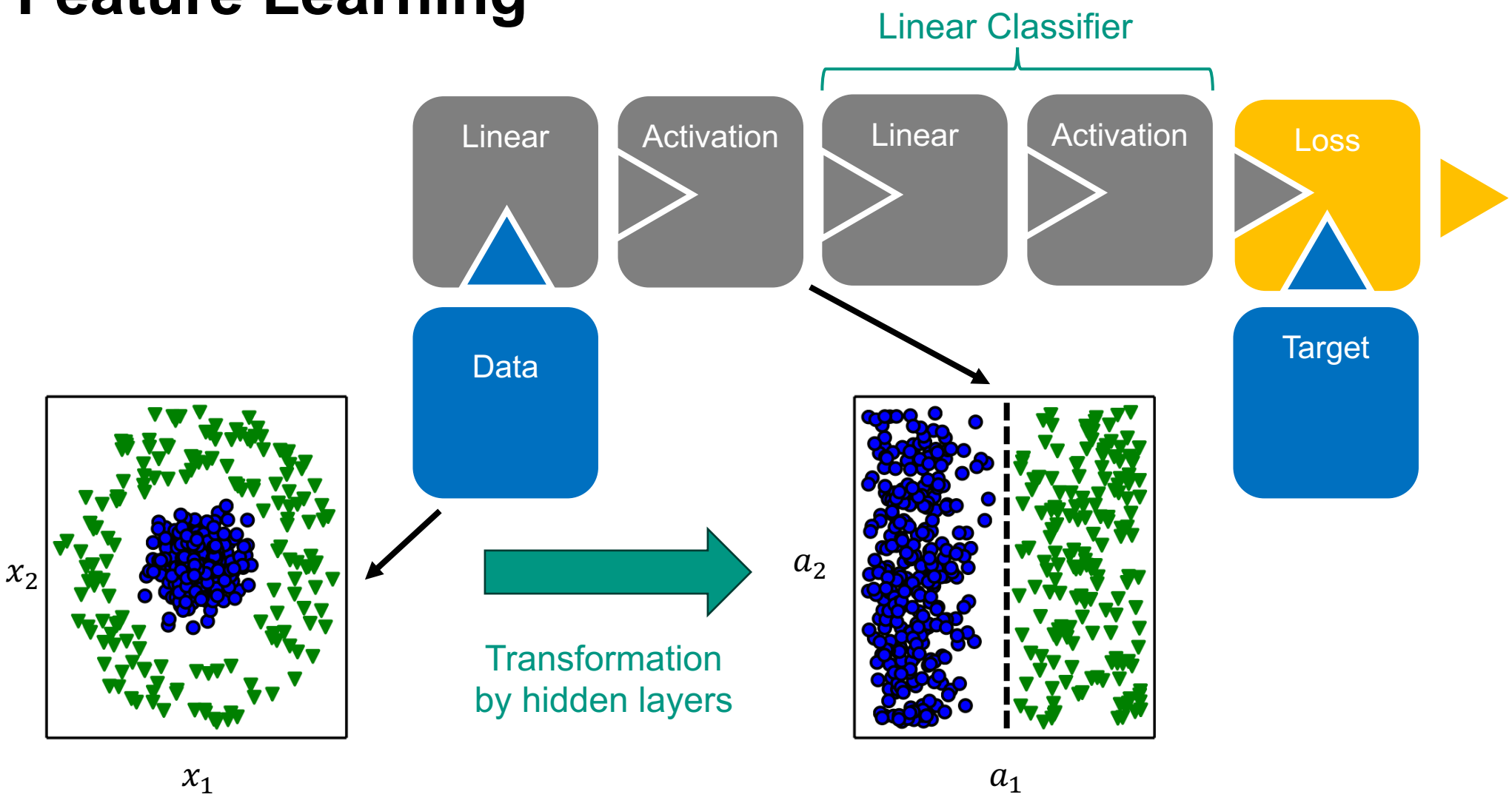
$$\boldsymbol{b}_\mathrm{i} = \begin{bmatrix} b_1^l \\ b_2^l \end{bmatrix}$$



Generally

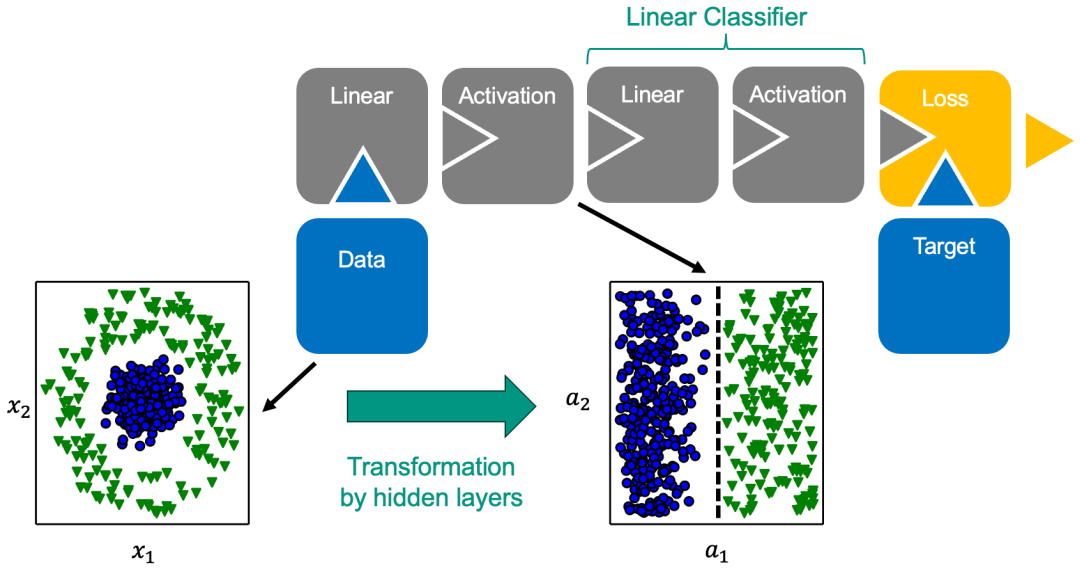
- Input layer: $\boldsymbol{x}$
- Hidden layer: $\boldsymbol{a}_l = \boldsymbol{\sigma}(\boldsymbol{W}_l a_{l-1} + \boldsymbol{b}_l)$
- Output layer: $\hat{\boldsymbol{y}} = \boldsymbol{\sigma}(\boldsymbol{W}_L \boldsymbol{a}_{L-1} + \boldsymbol{b}_\mathrm{L})$

Feature Learning



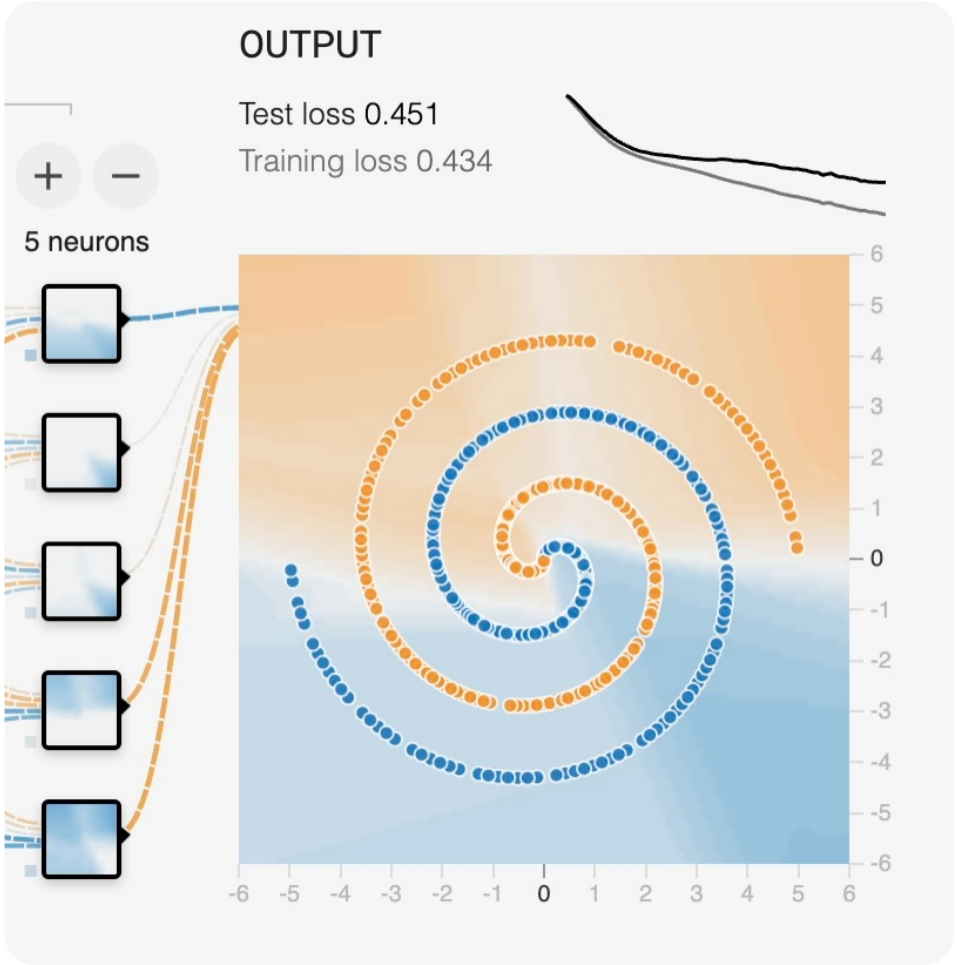
Feature Learning

- Hidden layer allows us to bent and twist input space
  - Nonlinear input space transformation
- We use a linear model on top, to do the classification
- **Allows us to solve the XOR problem!**



TensorFlow Playground

- Interactive web-based tool designed for experimenting with and visualizing the basics of neural networks using TensorFlow.js
- https://playground.tensorflow.org/



Epoch
000,000

Learning rate
0.03

Activation
Tanh

Regularization
None

Regularization rate
0

Problem type
Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

REGENERATE

FEATURES

Which properties do you want to feed in?

X_1

X_2

X_1^2

X_2^2

X_1X_2

$\sin(X_1)$

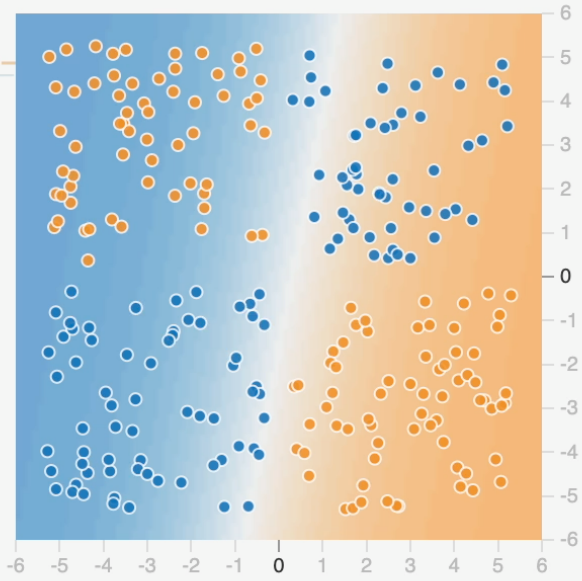
$\sin(X_2)$

+ − 0 HIDDEN LAYERS

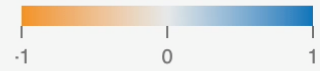
OUTPUT

Test loss 0.701

Training loss 0.690



Colors shows data, neuron and weight values.



Show test data

Discretize output

Epoch
000,000

Learning rate
0.03

Activation
Tanh

Regularization
None

Regularization rate
0

Problem type
Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

REGENERATE

FEATURES

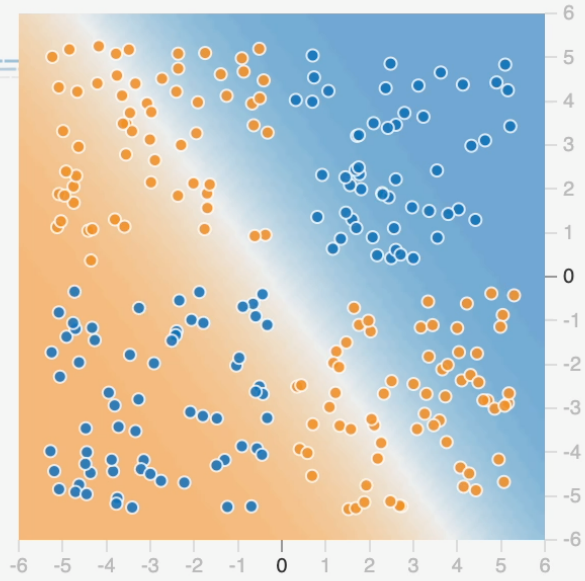
Which properties do you want to feed in?

X_1
X_2
X_1^2
X_2^2
X_1X_2
$\sin(X_1)$
$\sin(X_2)$

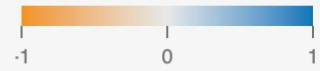
+ − 0 HIDDEN LAYERS

OUTPUT

Test loss 0.750
Training loss 0.845



Colors shows data, neuron and weight values.



Show test data Discretize output

Epoch
000,000

Learning rate
0.03

Activation
Tanh

Regularization
None

Regularization rate
0

Problem type
Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

REGENERATE

FEATURES

Which properties do you want to feed in?

X_1
X_2
X_1^2
X_2^2
X_1X_2
$\sin(X_1)$
$\sin(X_2)$

1 HIDDEN LAYER

2 neurons

*This is the output from one **neuron**. Hover to see it larger.*

OUTPUT

Test loss 0.514
Training loss 0.520

Colors shows data, neuron and weight values.

-1 0 1

Show test data Discretize output

Epoch
000,056

Learning rate
0.03

Activation
Tanh

Regularization
None

Regularization rate
0

Problem type
Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

REGENERATE

FEATURES

Which properties do you want to feed in?

X_1

X_2

X_1^2

X_2^2

X_1X_2

$\sin(X_1)$

$\sin(X_2)$

1 HIDDEN LAYER

3 neurons

*This is the output from one **neuron**. Hover to see it larger.*

OUTPUT

Test loss 0.089

Training loss 0.083

Colors shows data, neuron and weight values.

-1 0 1

Show test data

Discretize output

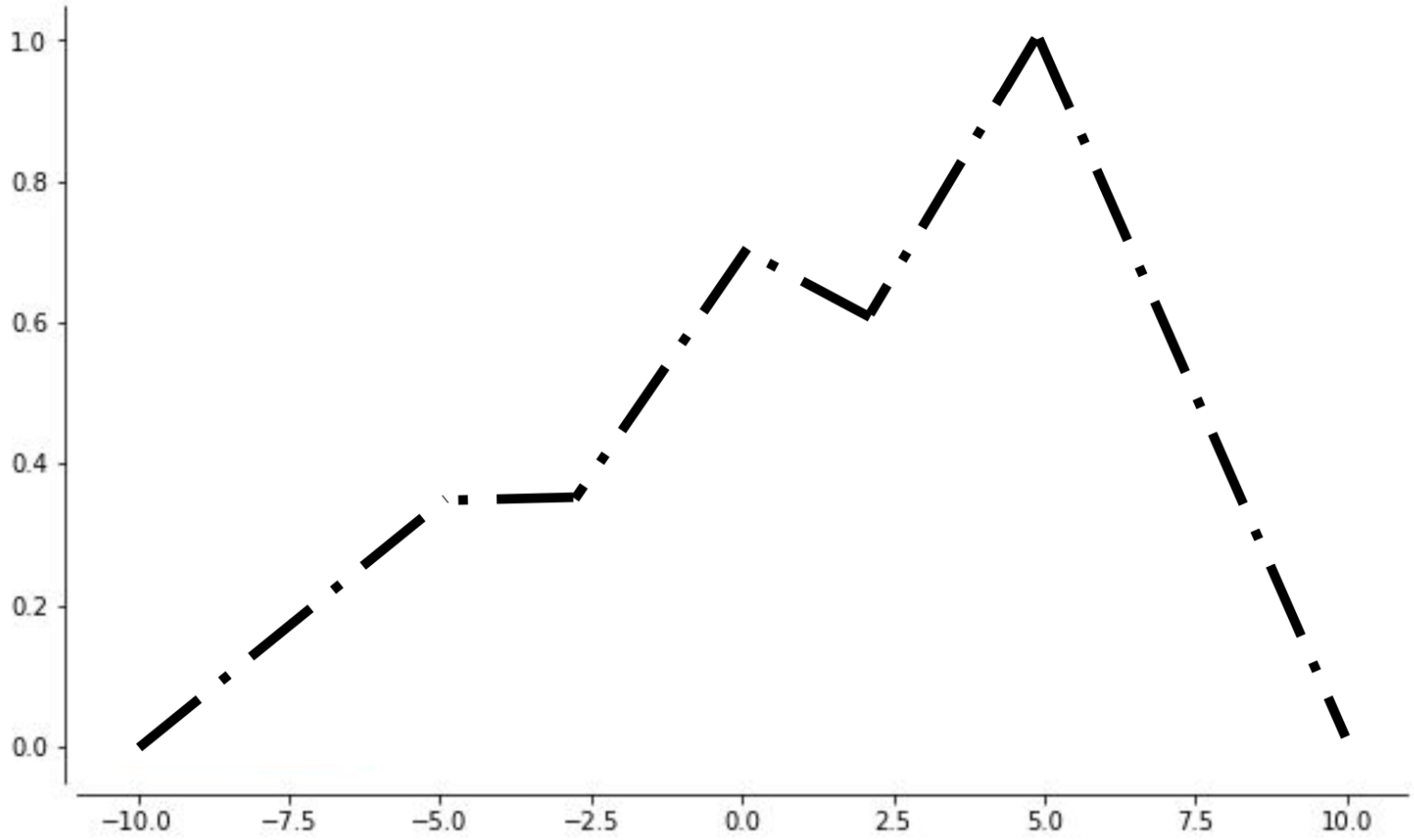
Universal Approximation Theorem

- **Question**: Which kind of problems can we solve with two-layer (1-hidden layer) neural networks?

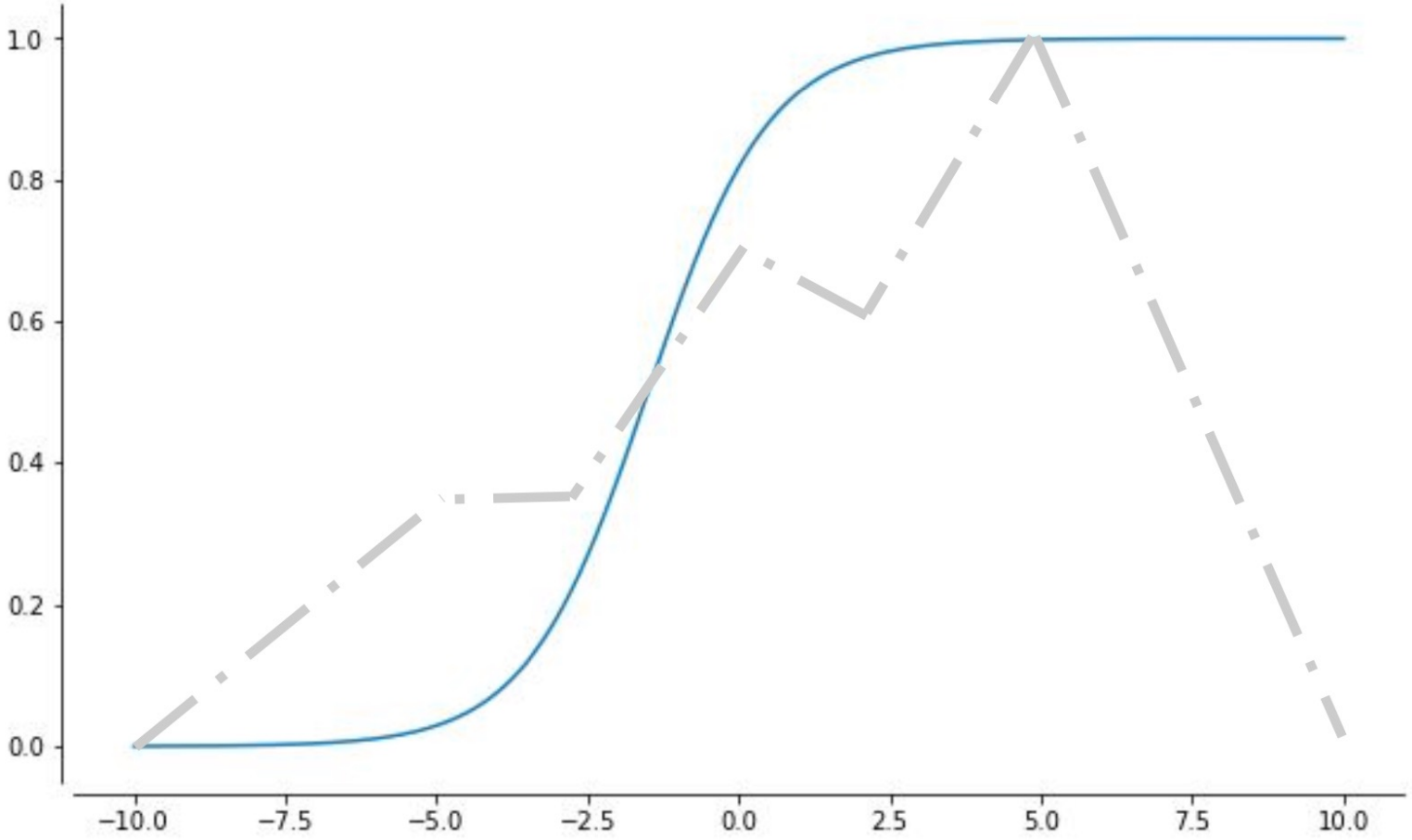
> *"For any continuous function from a hypercube $[0,1]^d$ to real numbers, and every positive epsilon, there exists a **sigmoid** based, 1-hidden layer neural network that obtains at most epsilon error in functional space."*

- **Interpretation**: A big enough two-layer NN can approximate (**not represent**) any smooth function.
- One of the most important theoretical results for NNs
- Shows that NNs are extremely flexible and expressive
- Paper: Cybenko., G. - Approximations by superpositions of sigmoidal functions

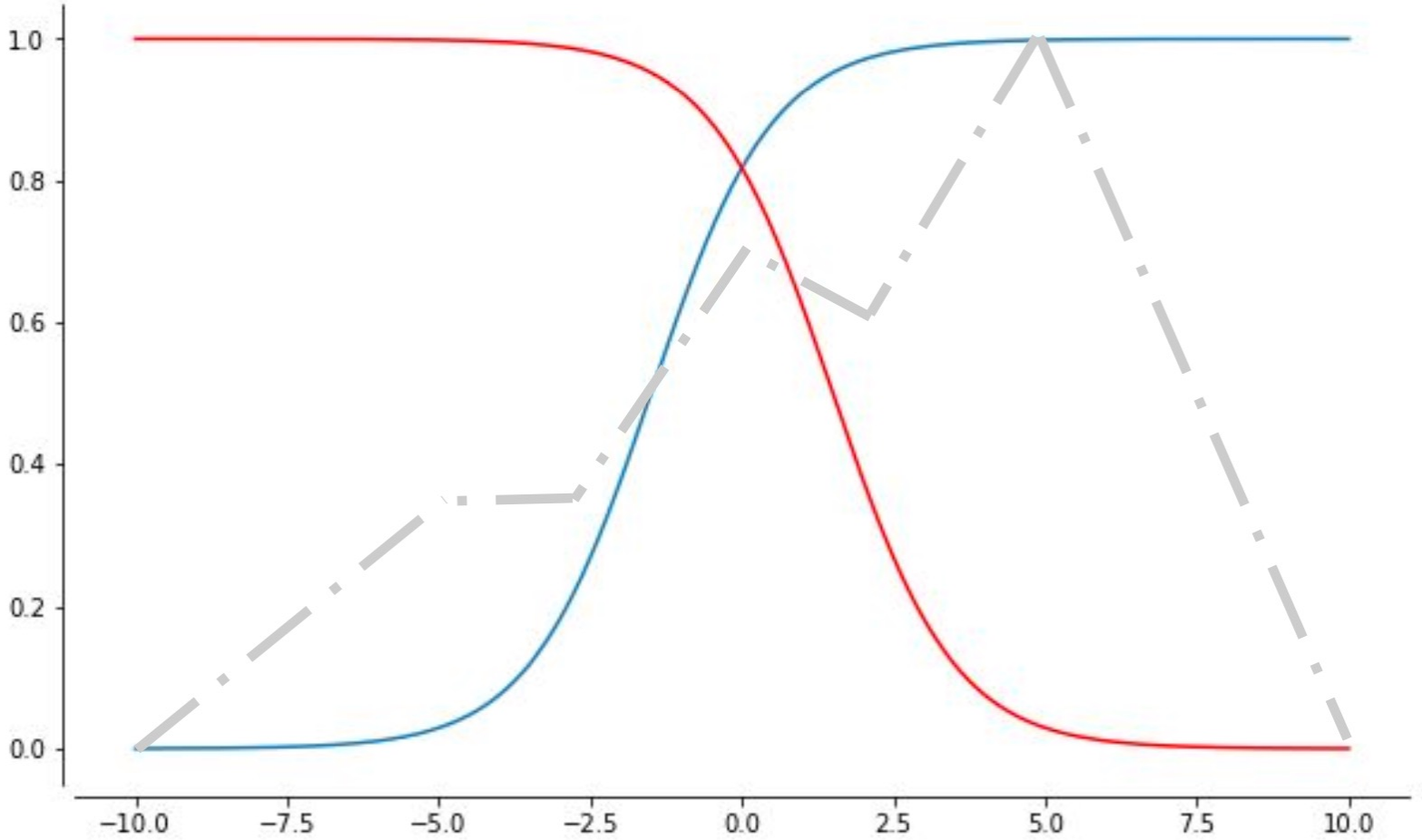
Universal Approximation ~~Theorem~~ Intuition



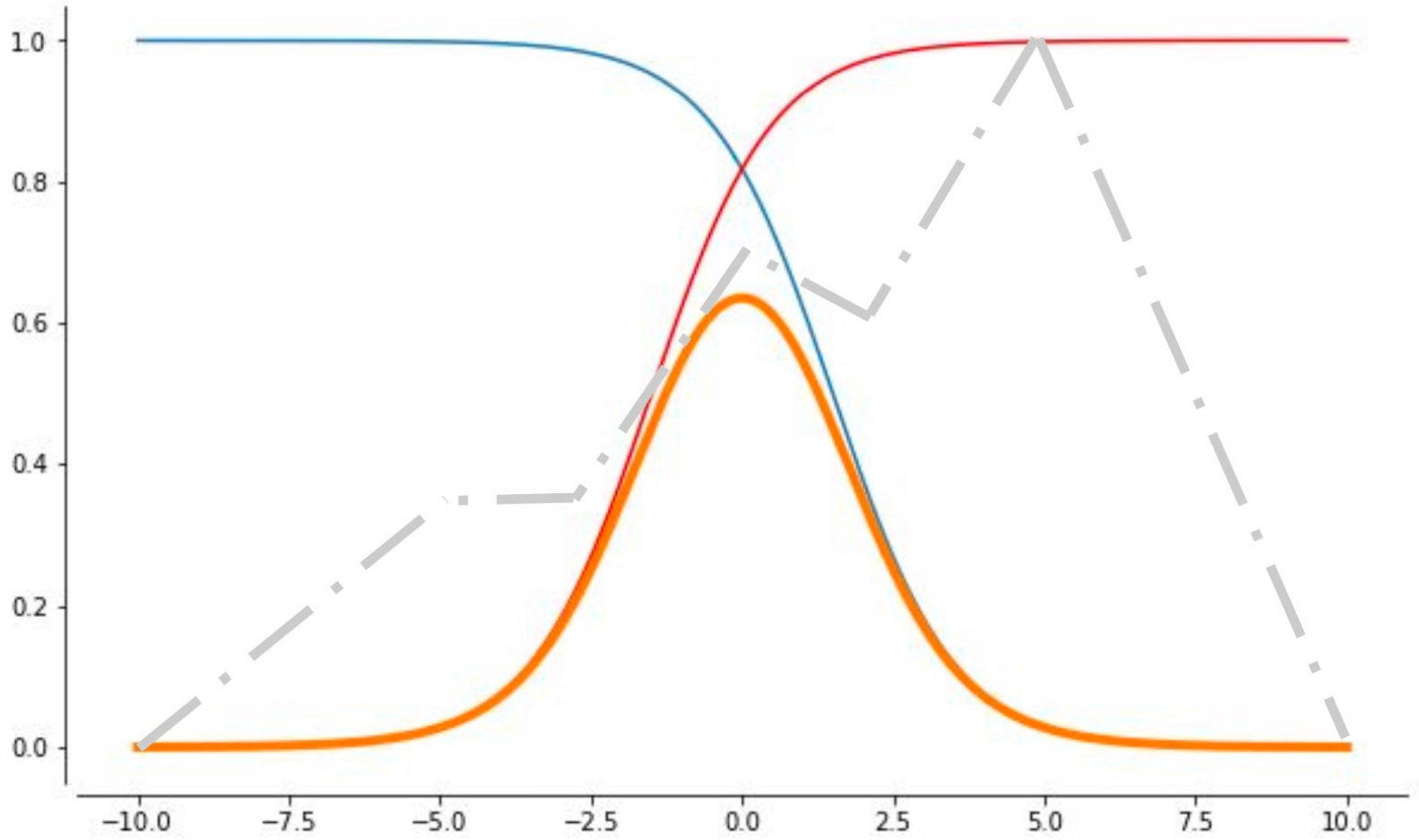
Universal Approximation ~~Theorem~~ Intuition



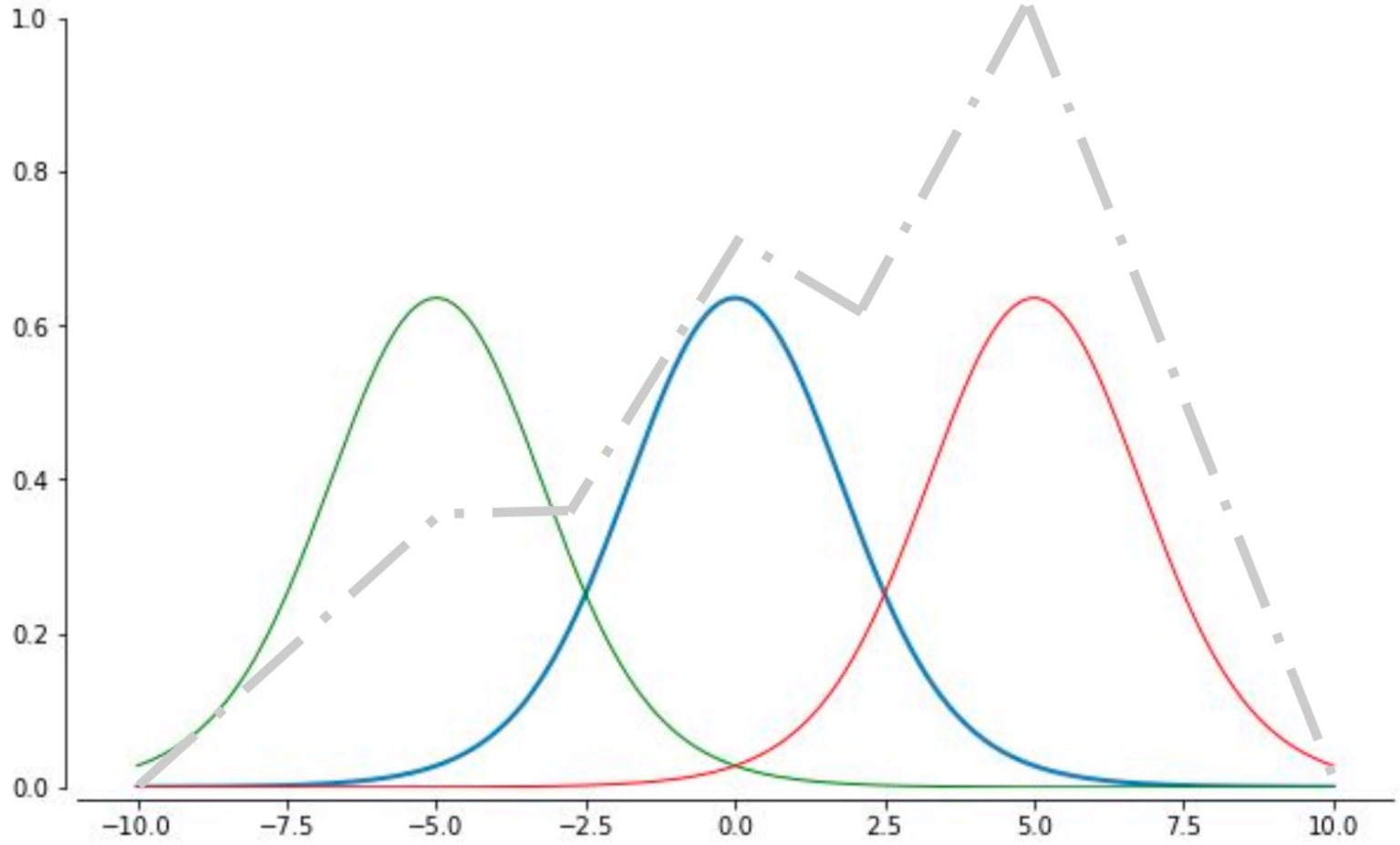
Universal Approximation ~~Theorem~~ Intuition



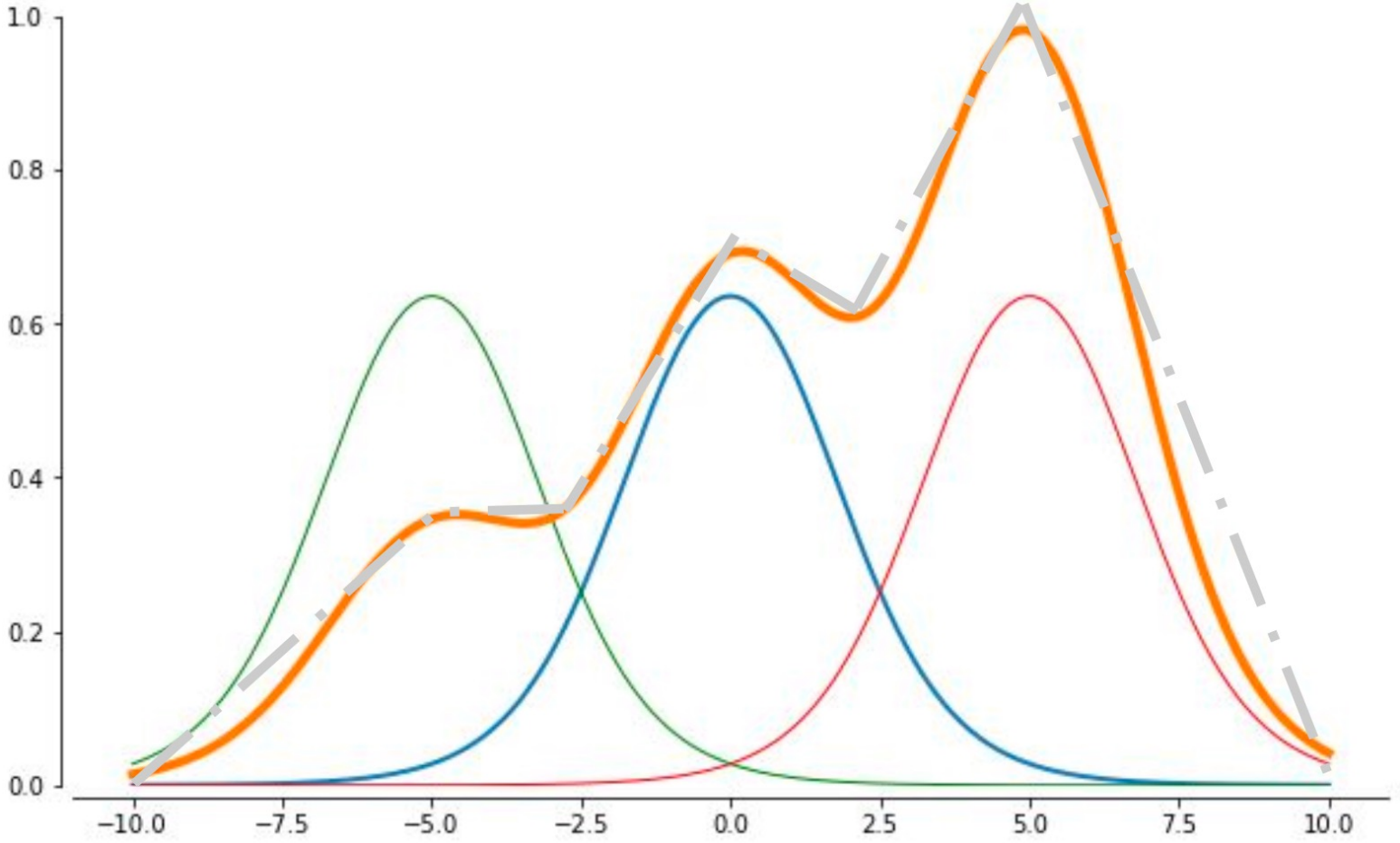
Universal Approximation ~~Theorem~~ Intuition



Universal Approximation ~~Theorem~~ Intuition



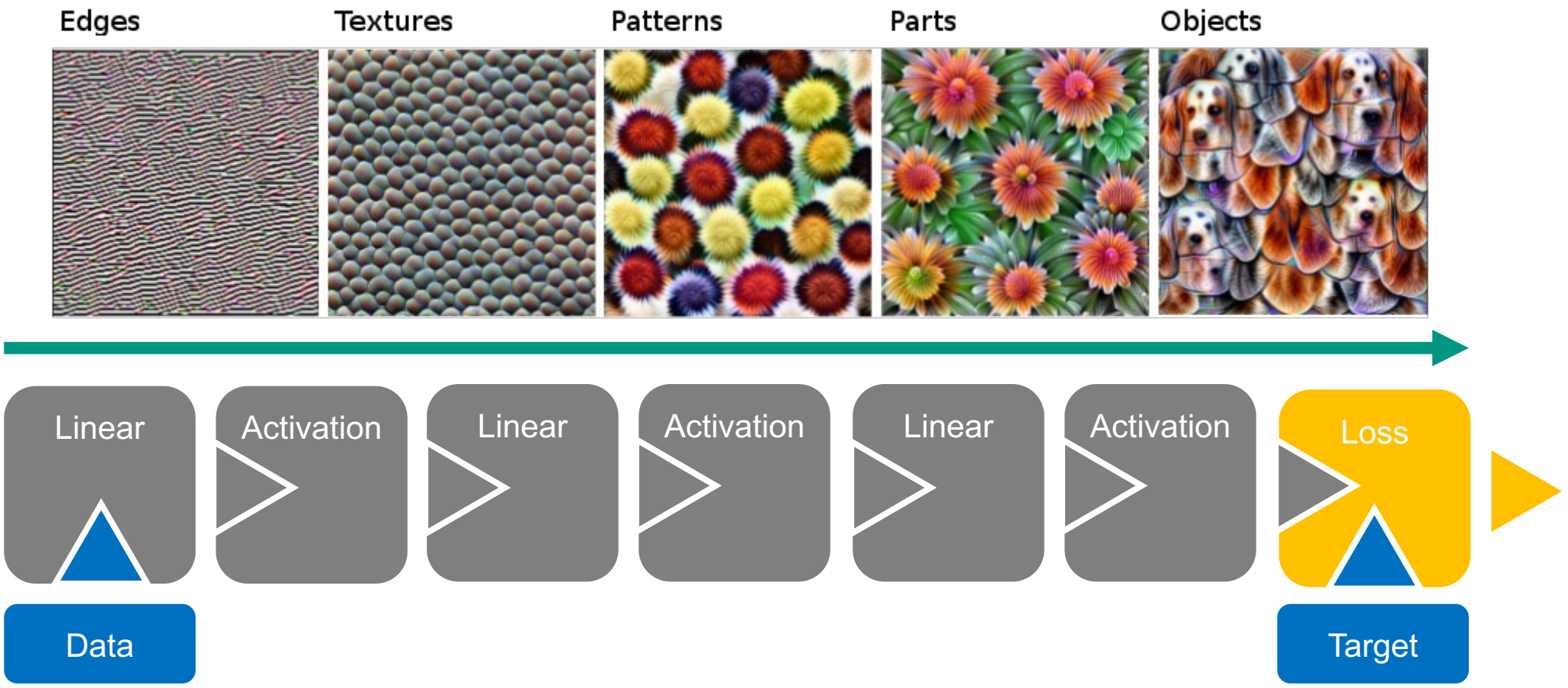
Universal Approximation ~~Theorem~~ Intuition



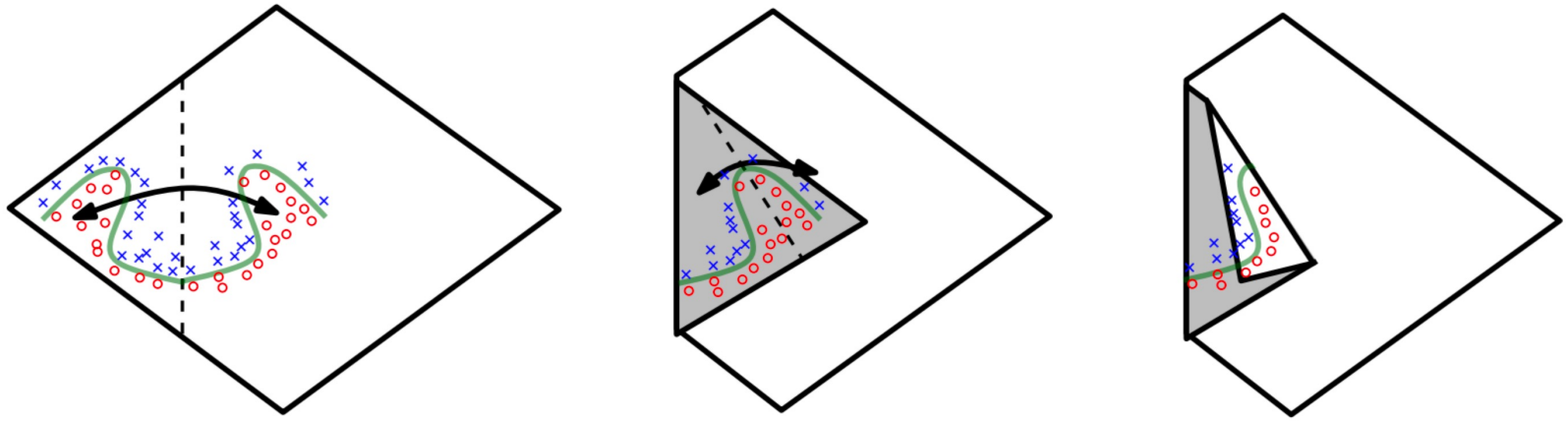
Deep Neural Networks - Width vs. Depth

- **Question**: So are two layers for a NN suitable for all problems?
- Universality of two-layer NNs is appealing, but requires exponential width
- Exponential increase in memory and computation time
- **Inductive bias**: Complex functions can be modeled as composition of simple functions
- More compact models and better generalization

Deep Neural Networks - Representation



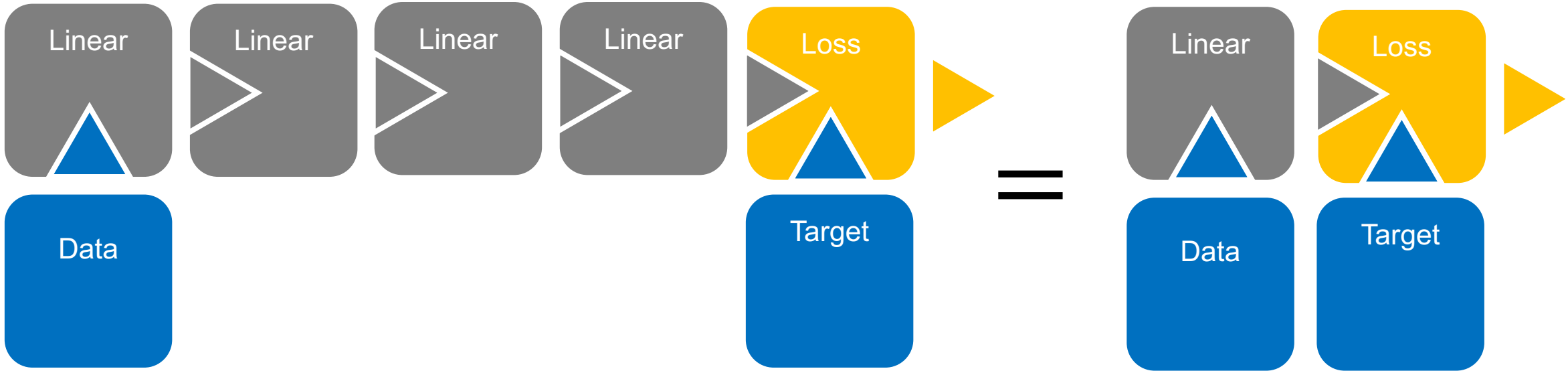
Network Depth vs. Width



- Geometric explanation why depth is preferred above width
- Mirror axis of symmetry given by hyperplane
- Expressing symmetries is easier with deep models than wide ones
- **Number of linear regions grows exponentially with depth and polynomially with width**
- Paper: Guido Montúfar - On the Number of Linear Regions of Deep Neural Networks

Deep Neural Networks Without Activation Functions

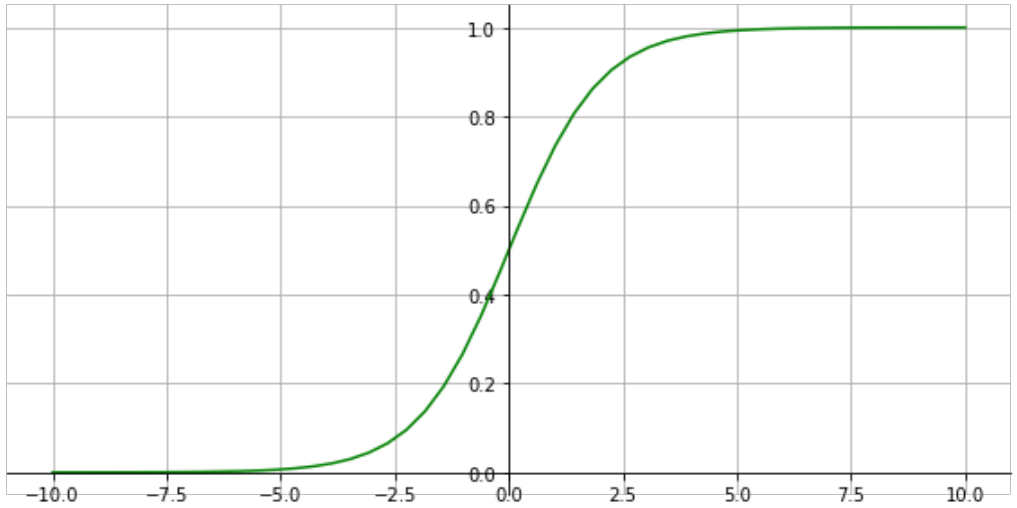
- **Question**: What happens if we remove activation functions from a NN?
- **Result**: Without nonlinear activation functions, neural networks of several layers collapse, always to a linear model.
- $\hat{\boldsymbol{y}} = \boldsymbol{f}_L \circ \cdots \circ \boldsymbol{f}_1 = \boldsymbol{W}_L(\boldsymbol{W}_{L-1}(\ldots) + \boldsymbol{b}_{\mathrm{L}-1}) + \boldsymbol{b}_{\mathrm{L}} = \mathbf{W}\mathbf{x} + \mathbf{b}$



Nonlinear Activation Functions

Sigmoid

- Function: $f(x) = \frac{1}{1+e^{-x}}$
- Derivative: $\frac{\partial f}{\partial x} = f(x)(1 - f(x))$



Note: Activation functions are applied element-wise

Properties

Pros

- Introduces nonlinear functional behavior
- Approximates a probability
  - $f(x) \in [0,1]$

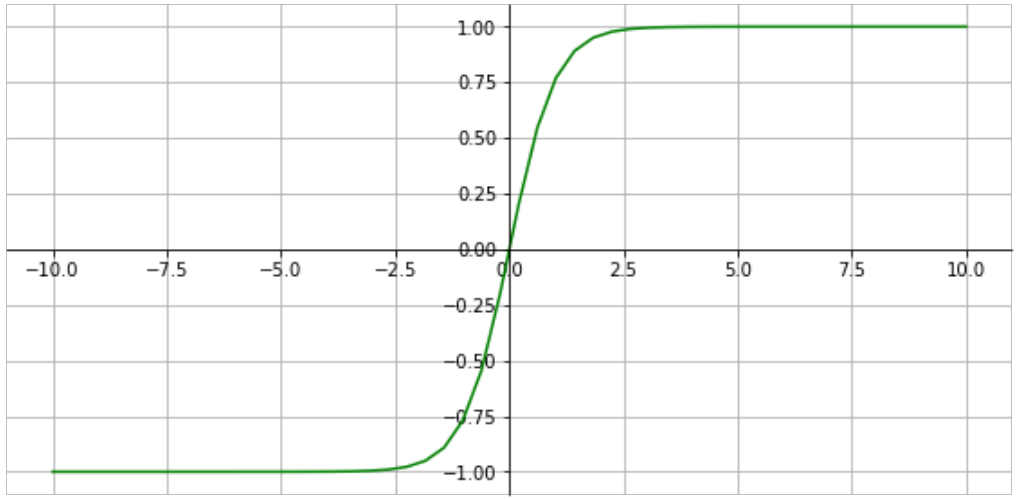
Cons

- Not centered around 0
- Saturation behavior
- Derivative vanishes in saturation areas
- Computationally intensive

Nonlinear Activation Functions

Hyperbolic tangent

- Function: $f(x) = \tanh(x)$
- Derivative: $\frac{\partial f}{\partial x} = (1 + f(x))(1 - f(x))$



Properties

Pros

- Centered around 0
- Normalizes the range of values
  - $f(x) \in [-1,1]$

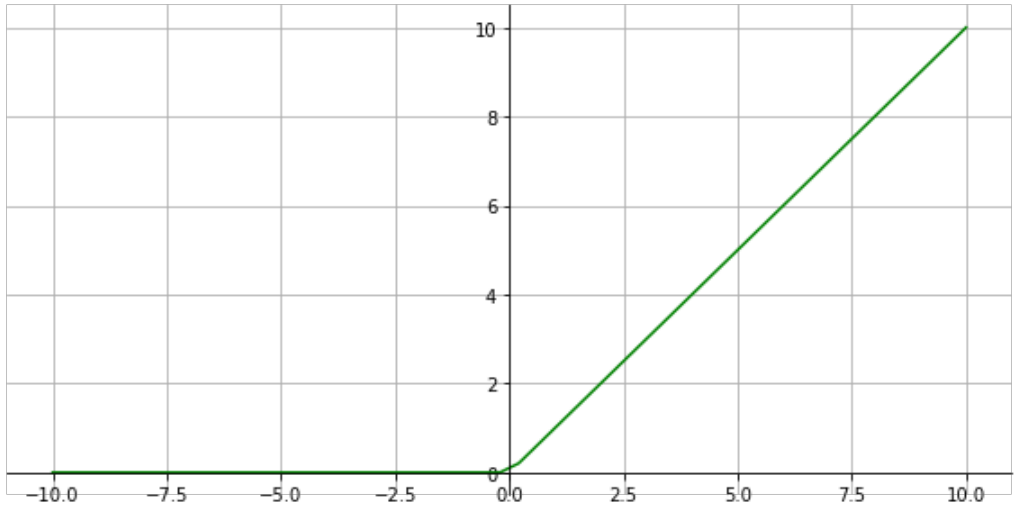
Cons

- Saturation behavior
- Derivative vanishes in saturation areas
- Computationally intensive

Nonlinear Activation Functions

Rectified Linear Unit (ReLU)

- Function: $f(x) = \max(0, x)$
- Derivative: $\frac{\partial f}{\partial x} = \begin{cases} 1, & x > 0 \\ 0, & else \end{cases}$



Properties

Pros

- Most used activation function
- Very easy to calculate
- No saturation behavior for $x > 0$

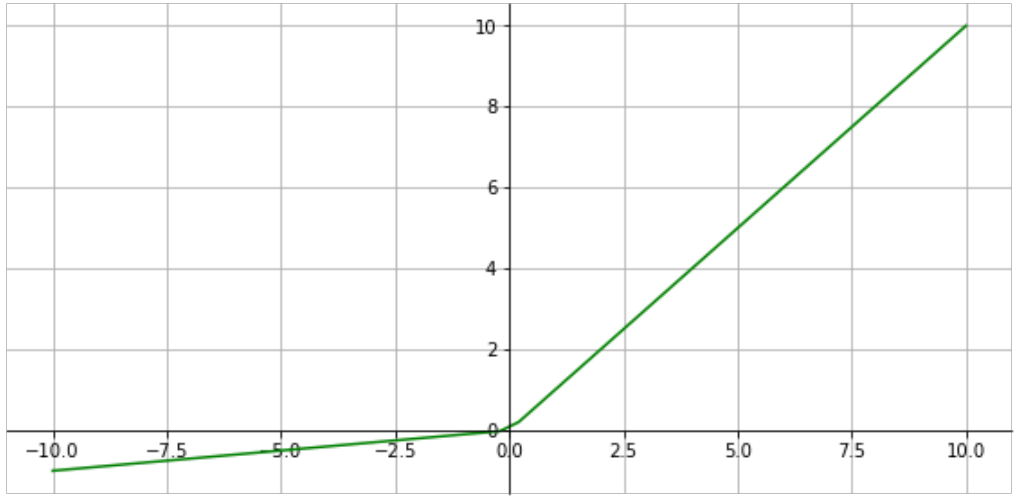
Cons

- Not centered around 0
- Derivative vanishes for $x < 0$
  - „Dying ReLU“

Nonlinear Activation Functions

LeakyReLU

- Function: $f(x) = \begin{cases} x, & x > 0 \\ \alpha x, & x \leq 0 \end{cases}$
- Derivative: $\frac{\partial f}{\partial x} = \begin{cases} 1, & x > 0 \\ \alpha, & else \end{cases}$



Properties

Pros

- No saturation behavior for $\alpha > 0$
- Easy to calculate
- Close to a function centered around 0

Cons

- Additional hyperparameter α

Nonlinear Activation Functions

可以用作归一化

Softmax

- Function: $f(\boldsymbol{x}) = \begin{bmatrix} \dots \\ \frac{e^{x_i}}{\sum_{j=0}^{k} e^{x_j}} \\ \dots \end{bmatrix}$
- Derivative: $\frac{\partial f_i(\boldsymbol{x})}{\partial x_j} = \begin{cases} f_i(\boldsymbol{x})(1 - f_i(\boldsymbol{x})), & i = j \\ -f_i(\boldsymbol{x}) f_j(\boldsymbol{x}). & i \neq j \end{cases}$

Properties

Pros

- Multi-dimensional generalization of the sigmoid function
- Discrete probability distribution
- Simple derivative

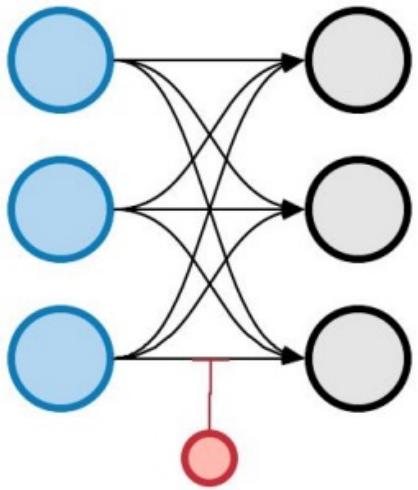
Cons

- Saturation behavior
- Derivative vanishes in saturation areas
- Computationally intensive

Linear Layer

Linear Layer

- Function: $\boldsymbol{f(x) = Wx + b}$
- Derivative: $\frac{\partial f}{\partial W} = \boldsymbol{x^T}$, $\frac{\partial f}{\partial b} = \boldsymbol{I}$



Properties

- Can be vectorized efficiently
- Efficiently computable by GPUs and TPUs
- **Despite the name, it's actually an affine transformation**

- **Info**: $\frac{\partial f}{\partial W}$ is actually a 3D tensor, but can be rewritten as a vector product

Outline

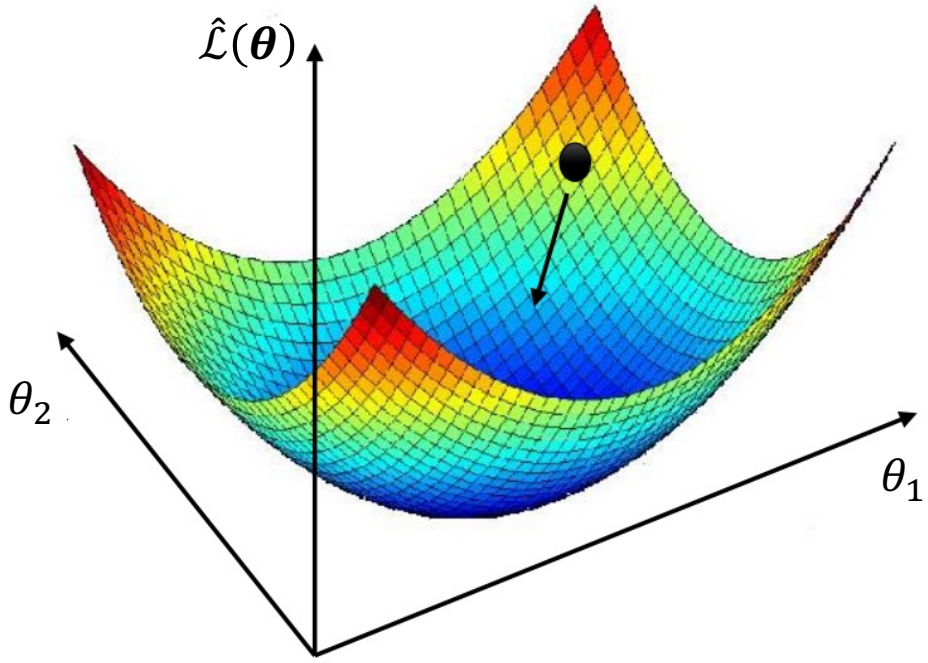
- Introduction
- Perceptron
- Deep Neural Networks
- Learning
- Applications
- Literature

Learning – Recap

- Learning = Optimization
- **Note**: Parameter vector $\boldsymbol{\theta} = \{\boldsymbol{W}^{(1)}, \dots, \boldsymbol{W}^{(L)}, \boldsymbol{b}^{(1)}, \dots, \boldsymbol{b}^{(L)}\}$
- **Goal**: Find the parameters $\boldsymbol{\theta}^*$ that minimize the empirical risk/error

$$\boldsymbol{\theta}^* \leftarrow \arg\min_{\boldsymbol{\theta}} \hat{\mathcal{L}}(\boldsymbol{\theta}) = \arg\min_{\boldsymbol{\theta}} \frac{1}{|D|} \sum_{(\boldsymbol{x},y)\in D} \ell(\hat{y}_{\boldsymbol{\theta}}(\boldsymbol{x}), y)$$

- The loss function $\ell(\hat{y}_{\boldsymbol{\theta}}(\boldsymbol{x}), y)$ specifies goodness of fit to the desired output y



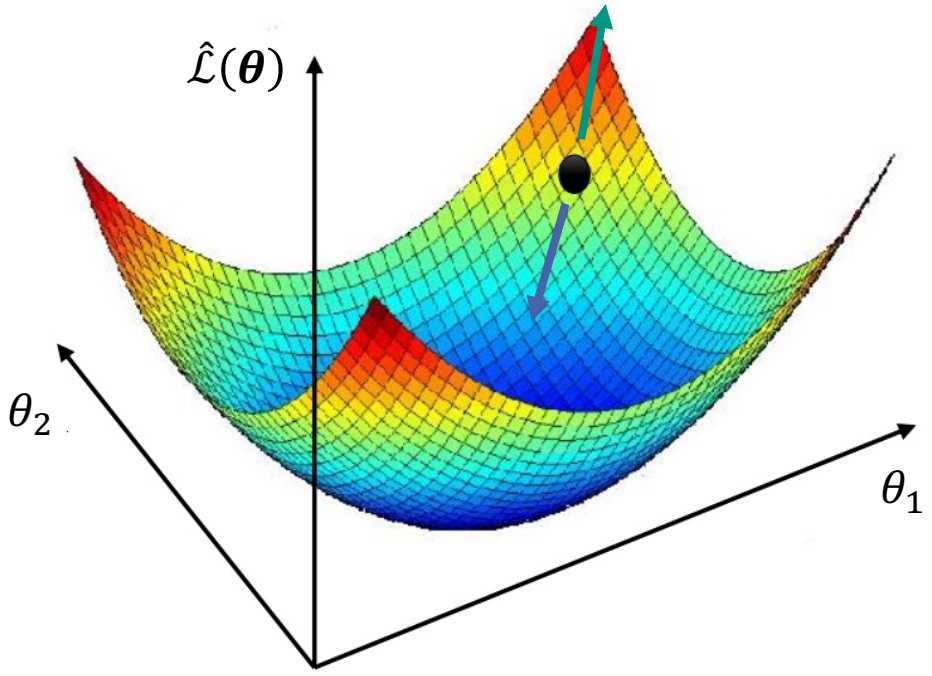
Gradient Descent

- First-order iterative optimization algorithm for finding a **local** minimum of a differentiable function

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \alpha \boldsymbol{g}_t = \boldsymbol{\theta}_t - \alpha \nabla_{\boldsymbol{\theta}} \hat{\mathcal{L}}(\boldsymbol{\theta}_t)$$

Algorithm

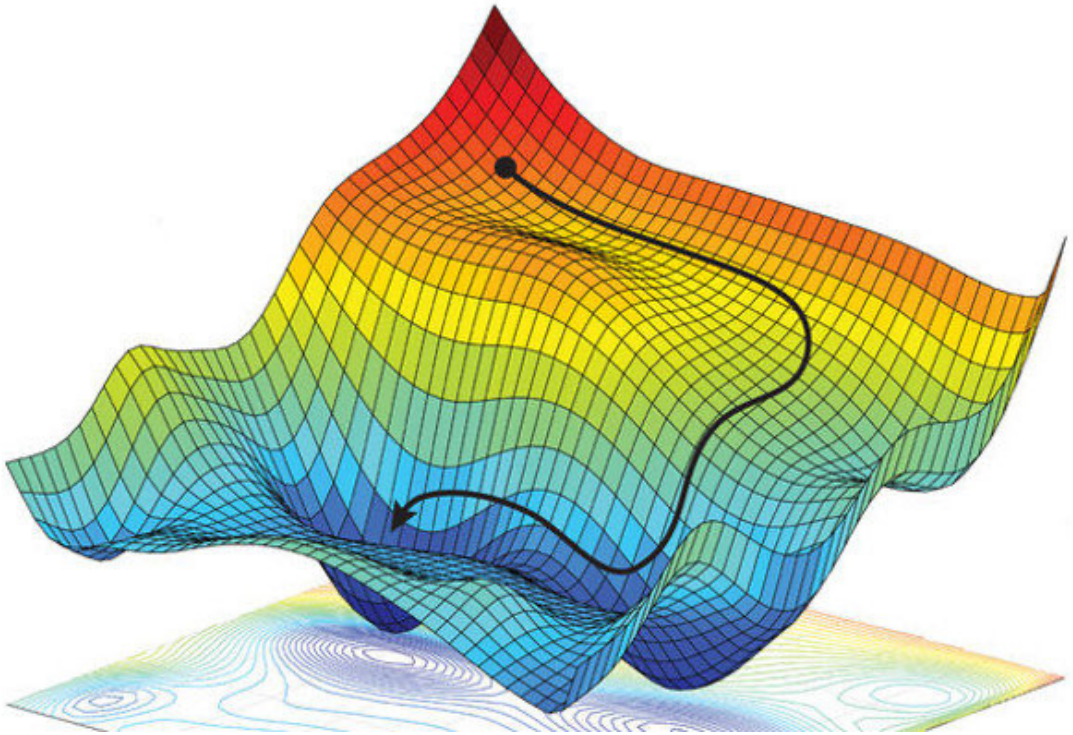
1. Calculate the gradient $\boldsymbol{g}_t$ (direction of steepest ascent) for the current parameters $\boldsymbol{\theta}_t$ for the objective function $\hat{\mathcal{L}}$
2. Take step with **step size** α into the **opposite** direction (steepest descent)
3. Update parameters with new values $\boldsymbol{\theta}_{t+1}$



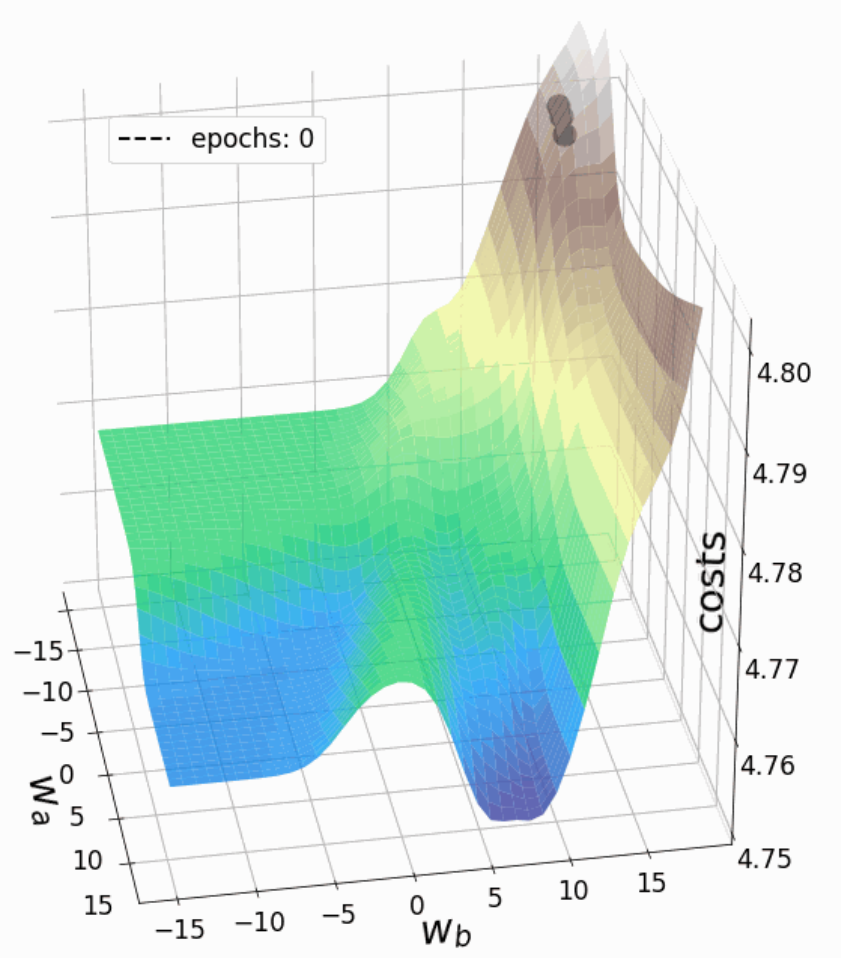
Gradient

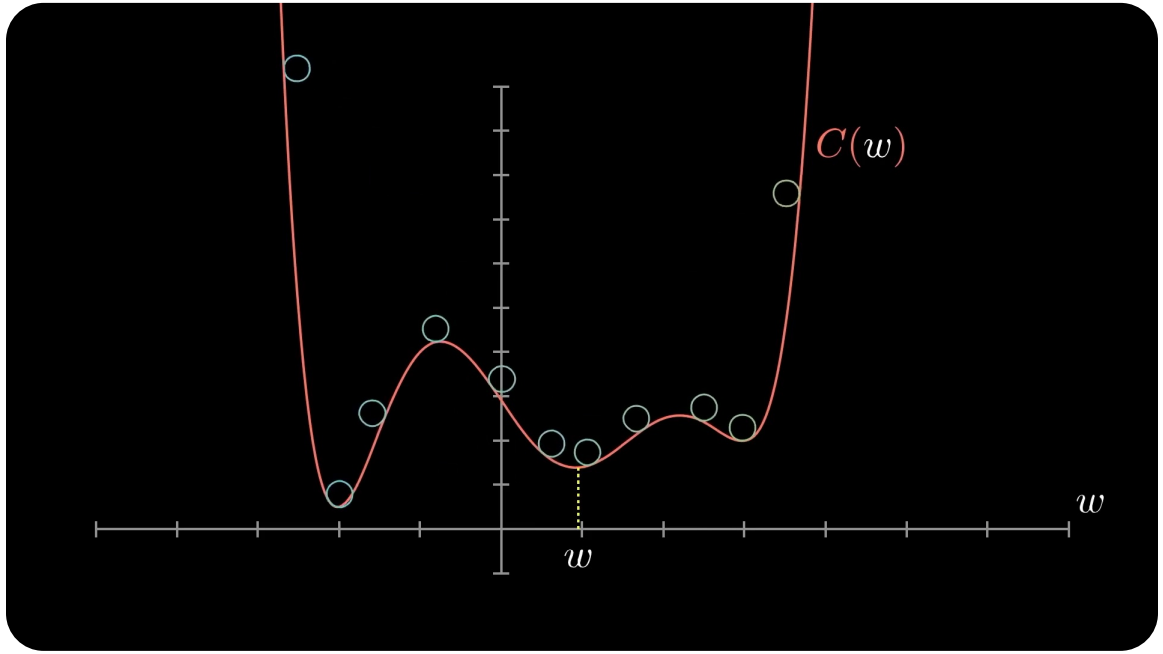
- Derivative of a scalar function w.r.t. to a vector
- $f\colon \mathbb{R}^n \to \mathbb{R}$
- Gradient vector $\nabla_{\boldsymbol{x}} f \in \mathbb{R}^n$

$$\nabla_{\boldsymbol{x}} f = \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial f(\boldsymbol{x})}{\partial x_1} \\ \cdots \\ \frac{\partial f(\boldsymbol{x})}{\partial x_n} \end{bmatrix}$$



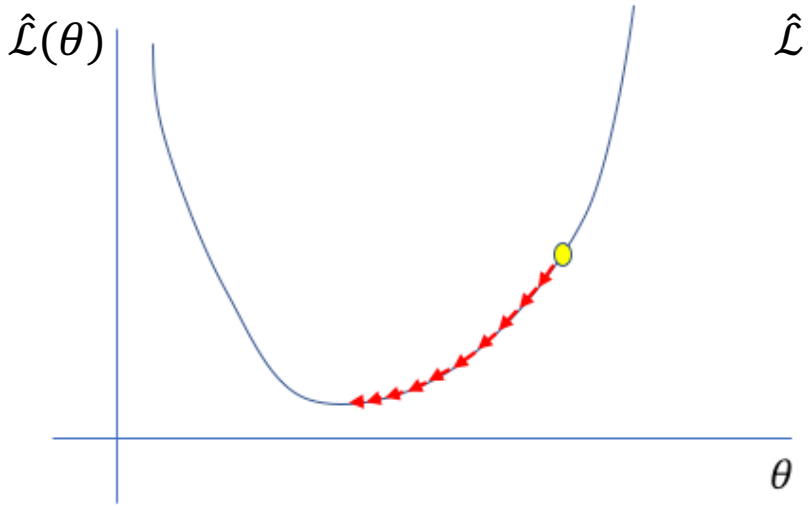
Gradient Descent - Visualization





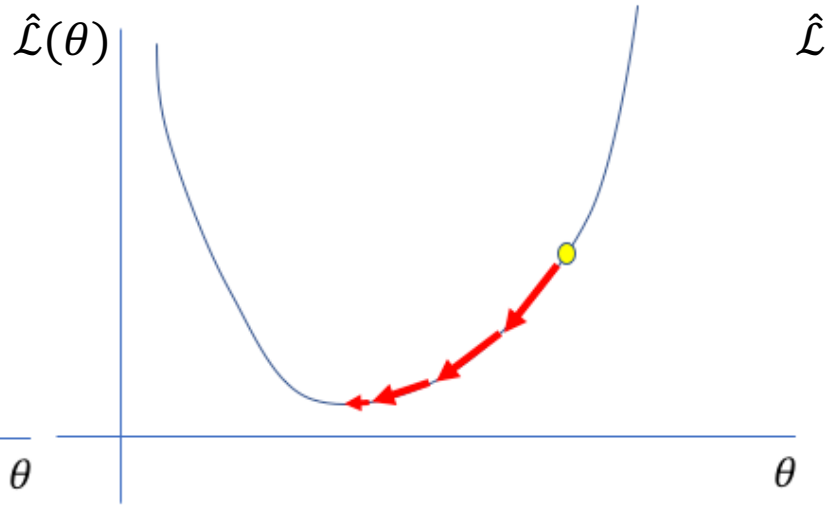
Gradient Descent - Learning Rate

Too low



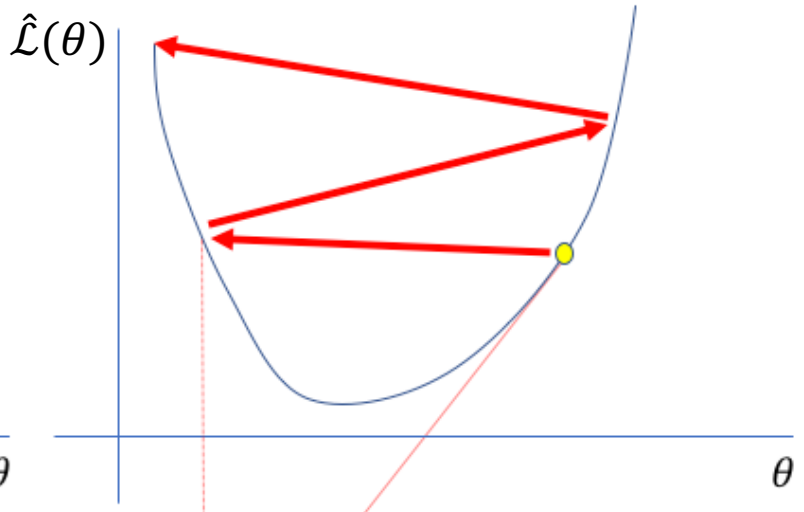
A small learning rate requires many updates before reaching the minimum point

Just right



The optimal learning rate swiftly reaches the minimum point

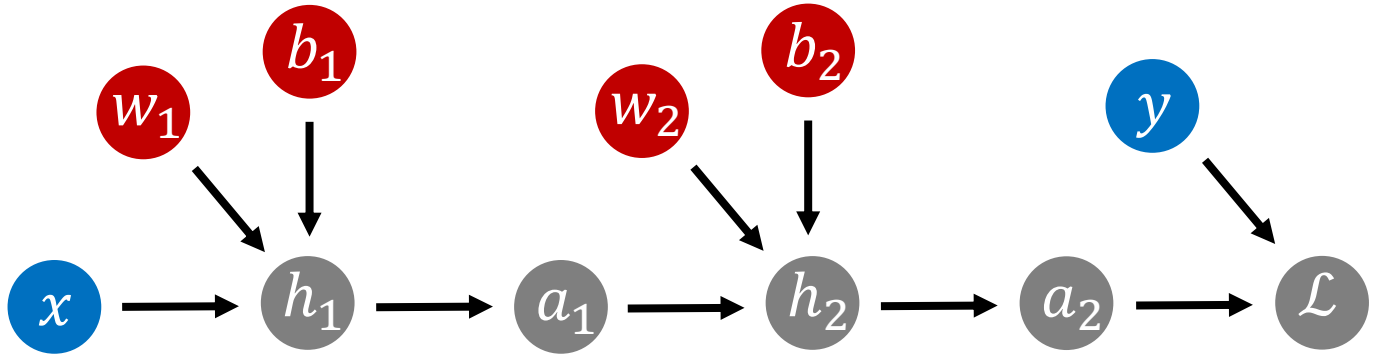
Too high



Too large of a learning rate causes drastic updates which lead to divergent behaviors

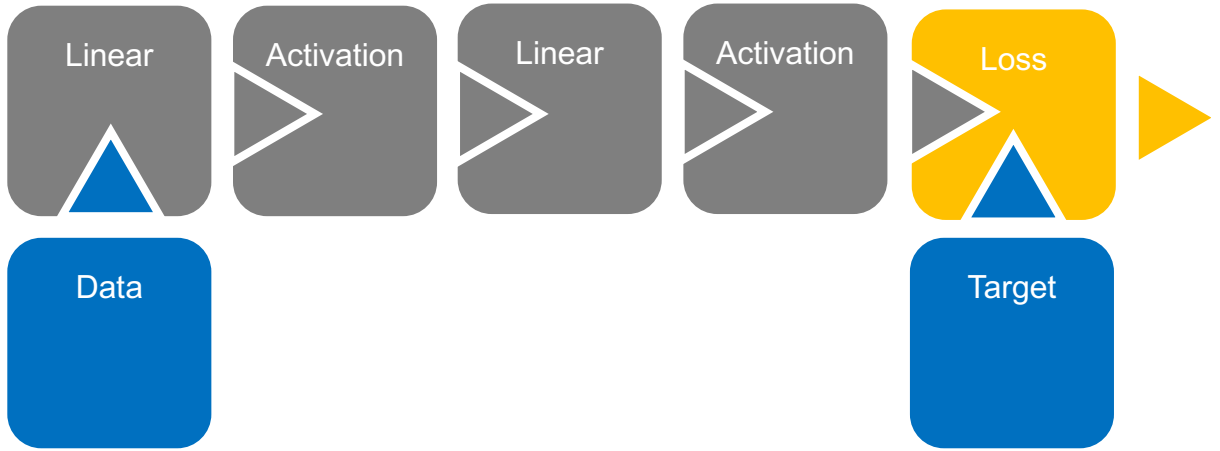
Computational Graph

- A computational graph is a directed acyclic graph (DAG) where the nodes correspond to operations or variables
- Decompose complex computations into sequence of atomic assignments
- Modularity: Each node must only ”know” how to compute gradients w.r.t. its own arguments
- This decomposition and reuse of computation is key to the success of the backpropagation algorithm (see later slides)

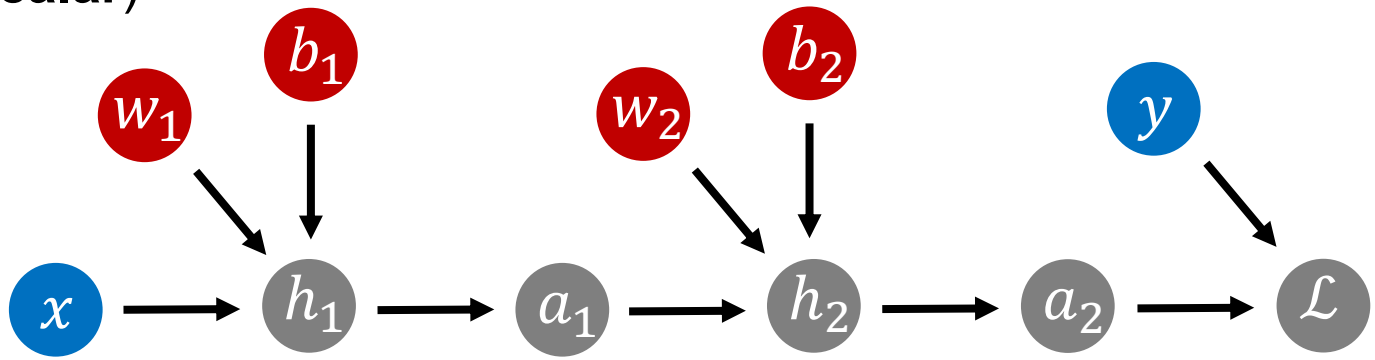


Computational Graph – Forward Pass

- A computational graph has 3 kinds of nodes
  - Input nodes
  - Parameter nodes
  - Compute nodes
- **Forward pass**: Input data is fed into the neural network, and computations are performed layer by layer to produce an output.



- **Example**: Two-layer neural network (**scalar**)
  - $h_1 = w_1 x + b_1$
  - $a_1 = \sigma(h_1)$
  - $h_2 = w_2 a_1 + b_2$
  - $a_2 = \sigma(h_2)$
  - $\mathcal{L} = \ell(a_2, y)$



Chain Rule

- **Chain rule**: Express the derivative of the composition $h(x)$ of two differentiable functions f and g in terms of the derivatives of each individual function f and g.

$$h(x) = f\big(g(x)\big) = f \circ g \Rightarrow \frac{\partial f(g(x))}{\partial x} = \frac{\partial h}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$$

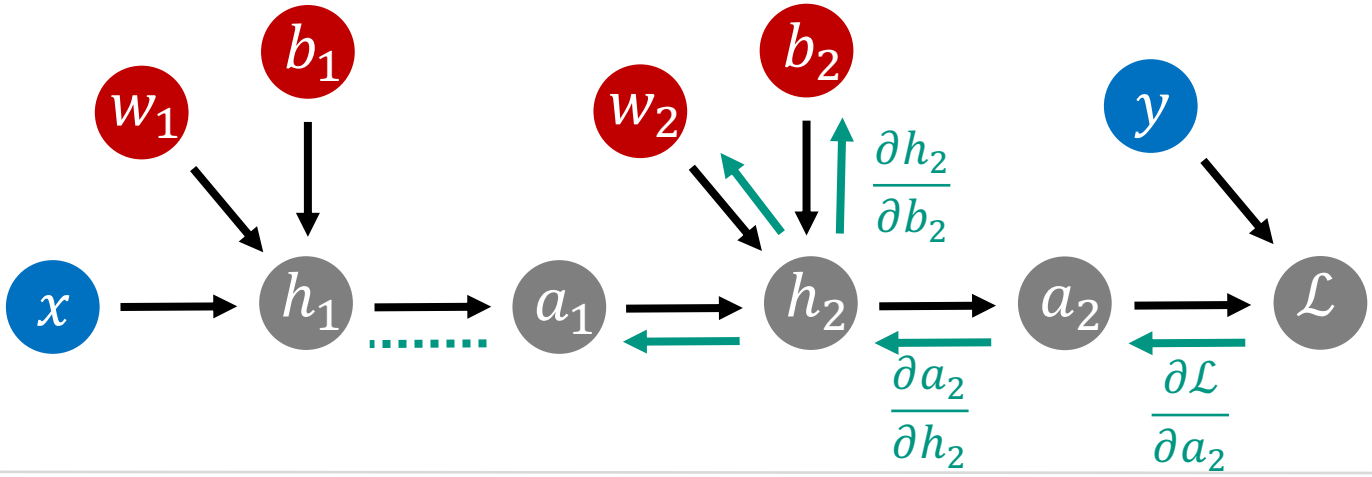
- **Multivariate chain rule**: An extension of the chain rule to functions with multiple input variables.
  - Here as special case, where g depends on the same input variable x

$$h(x) = f(g_1(x), \dots, g_M(x)) \Rightarrow \frac{\partial f(g_1(x), \dots, g_M(x))}{\partial x} = \sum_m \frac{\partial f}{\partial g_m}\frac{\partial g_m}{\partial x}$$

Computational Graph – Backward Pass

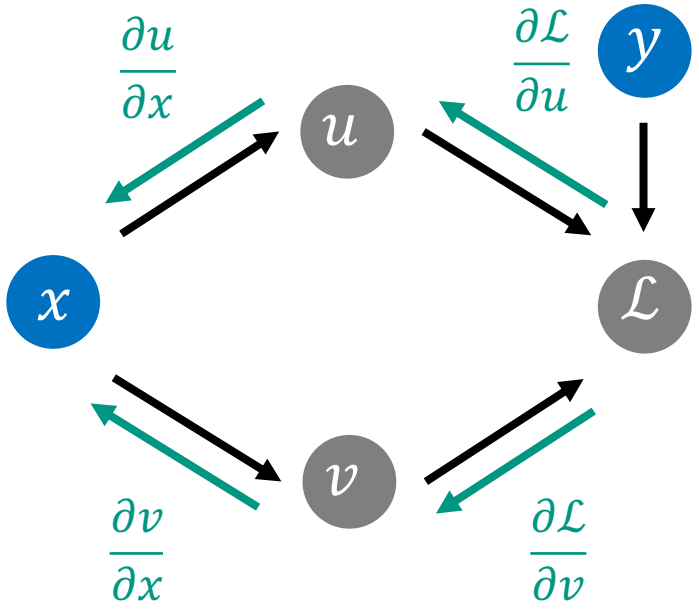
- **Backward pass**: Calculates the gradient for each parameter by propagating the error signal back through the network.
- **Example**: Two-layer neural network
  - $h_1 = w_1 x + b_1$
  - $a_1 = \sigma(h_1)$
  - $h_2 = w_2 a_1 + b_2$
  - $a_2 = \sigma(h_2)$
  - $\mathcal{L} = \ell(a_2, y)$

- **Gradients** Already computed parts
  - $\frac{\partial \mathcal{L}}{\partial w_2} = \frac{\partial \mathcal{L}}{\partial a_2}\frac{\partial a_2}{\partial h_2}\frac{\partial h_2}{\partial w_2}$
  - $\frac{\partial \mathcal{L}}{\partial b_2} = \frac{\partial \mathcal{L}}{\partial a_2}\frac{\partial a_2}{\partial h_2}\frac{\partial h_2}{\partial b_2}$
  - $\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial a_2}\frac{\partial a_2}{\partial h_2}\frac{\partial h_2}{\partial a_1}\frac{\partial a_1}{\partial h_1}\frac{\partial h_1}{\partial w_1}$
  - $\frac{\partial \mathcal{L}}{\partial b_1} = \frac{\partial \mathcal{L}}{\partial a_2}\frac{\partial a_2}{\partial h_2}\frac{\partial h_2}{\partial a_1}\frac{\partial a_1}{\partial h_1}\frac{\partial h_1}{\partial b_1}$



Computational Graph – Backward Pass

- **Example**: Multiple connections
  - $u = u(x)$
  - $v = v(x)$
  - $\mathcal{L} = \ell(u, v, y)$
- **Gradients**
  - $\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial u}\frac{\partial u}{\partial x} + \frac{\partial \mathcal{L}}{\partial v}\frac{\partial v}{\partial x}$
- Multiple incoming gradients are summed up (multivariate chain rule)

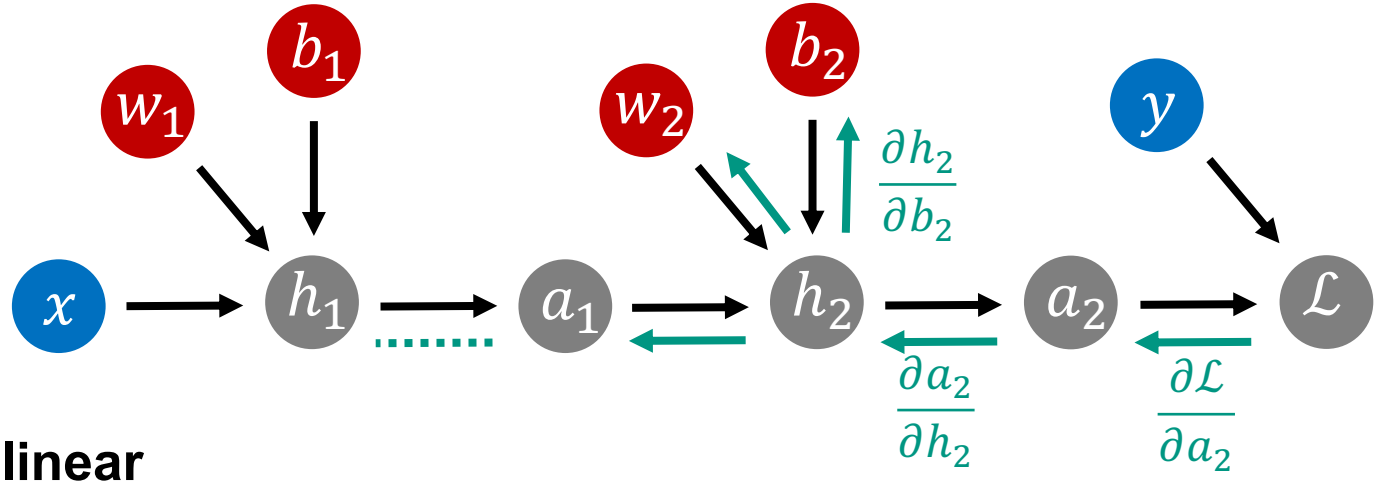


Backpropagation

- Efficient method of computing gradients in directed computational graphs

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \alpha \boldsymbol{g}_t = \boldsymbol{\theta}_t - \alpha \underbrace{\nabla_{\boldsymbol{\theta}} \hat{\mathcal{L}}(\boldsymbol{\theta}_t)}_{\text{Backpropagation}}$$

- Applies chain rule to calculate results
- Reuses intermediate results to reduce computational complexity



- **Reduces exponential complexity to linear complexity for gradient computation**

将梯度计算的指数级复杂度降低为线性复杂度

Gradient Descent

Algorithm

1. Initialize weights $\boldsymbol{\theta}_0$, pick learning rate α
2. For all elements in the training dataset $\mathrm{n} \in \{1, \ldots, N\}$ do:
   1. **Forward pass**: Propagate x_n through the network to calculate prediction $\hat{y}_n$
   2. **Backward pass**: Backpropagate error to obtain the gradient for all parameters $\nabla_{\boldsymbol{\theta}_t} \ell(\hat{y}_n, y_n)$
3. Gradient: $\boldsymbol{g}_t = \frac{1}{N} \sum_{n=1}^{N} \nabla_{\boldsymbol{\theta}_t} \ell(\hat{y}_n, y_n)$
4. Update weights: $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \alpha \boldsymbol{g}_t$
5. Stop if end criterion is met, else go to 2.
   (e.g. if validation error doesn‘t decrease anymore)

Gradient Descent

$$\boldsymbol{g}_t \leftarrow \nabla_{\boldsymbol{\theta}_t} \frac{1}{N} \sum_{n=1}^{N} \ell(\hat{y}_n, y_n) = \nabla_{\boldsymbol{\theta}} \hat{\mathcal{L}}(\boldsymbol{\theta}_t)$$

- Gradient ist calculated over the whole dataset
- Weights are updated **after** the whole dataset has been processed

- **Disadvantage**: Calculates the gradient for the entire dataset in a single update
  - Typically $N = 1$ Million or more
  - Calculation is very slow for big datasets
  - Often doesn‘t fit in memory for big datasets

ILSVRC (ImageNet), 2009:
1,5 Million images

https://cv.gluon.ai/build/examples_datasets/imagenet.html

Stochastic Gradient Descent (SGD)

- **Solution**: Use a stochastic approximation of gradient descent

$$\nabla_{\boldsymbol{\theta}} \hat{\mathcal{L}}(\boldsymbol{\theta}_t) = \nabla_{\boldsymbol{\theta}_t} \mathbb{E}_{(x,y)\sim D}[\ell(\hat{y}_i, y_i)] = \nabla_{\boldsymbol{\theta}_t} \frac{1}{N} \sum_{n=1}^{N} \ell(\hat{y}_n, y_n)$$

- Approximate the expectation by a smaller subset $B \ll N$ of the data

$$\nabla_{\boldsymbol{\theta}} \hat{\mathcal{L}}(\boldsymbol{\theta}_t) = \nabla_{\boldsymbol{\theta}_t} \mathbb{E}_{(x,y)\sim D}[\ell(\hat{y}_i, y_i)] \approx \nabla_{\boldsymbol{\theta}_t} \frac{1}{B} \sum_{b=1}^{B} \ell(\hat{y}_b, y_b)$$

- We call the subset $\{(x_1, y_1), \ldots, (x_B, y_B)\} \in D$ a „**minibatch**“

- **Properties**
  - Faster update cycles: Update of parameters after every minibatch
  - Computationally tractable with limited (GPU) memory
  - Introduces stochasticity in the training process
  - In practice, only variants of SGD are used, not GD

Stochastic Gradient Descent (SGD)

Algorithm

1. Initialize weights $\boldsymbol{\theta}_0$, pick learning rate α and minibatch size B
2. Sample a minibatch $\{(x_1, y_1), \ldots, (x_B, y_B)\} \in D$ randomly
3. For all minibatch elements $b \in \{1, \ldots, B\}$ do:
   1. Forward pass: Forward propagate x_b through the network to calculate prediction $\hat{y}_b$
   2. Backward pass: Backpropagate error to obtain the gradient for all parameters $\nabla_{\boldsymbol{\theta}_t} \ell(\hat{y}_b, y_b)$
4. Gradient: $\boldsymbol{g}_t = \frac{1}{B} \sum_{b=1}^{B} \nabla_{\boldsymbol{\theta}_t} \ell(\hat{y}_b, y_b)$
5. Update weights: $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \alpha \boldsymbol{g}_t$
6. Stop if end criterion is met, else go to 2.
   (e.g. if validation error doesn‘t decrease anymore)

Remarks 在开始前先对数据集进行随机打乱（shuffeling），以避免样本顺序的影响

- Shuffeling the dataset and splitting it into batches is typically done beforehand during preprocessing

Additional: Function Differentiation

Scalar Function w.r.t. Vector

- $f\colon \mathbb{R}^n \to \mathbb{R}$
- Gradient vector $\nabla_{\boldsymbol{x}} f \in \mathbb{R}^n$

$$\nabla_{\boldsymbol{x}} f = \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial f(\boldsymbol{x})}{\partial x_1} \\ \cdots \\ \frac{\partial f(\boldsymbol{x})}{\partial x_n} \end{bmatrix}$$

Scalar Function w.r.t. Matrix

- $f\colon \mathbb{R}^{n \times m} \to \mathbb{R}$
- Derivative matrix $\nabla_{\boldsymbol{X}} f \in \mathbb{R}^{n \times m}$

$$\nabla_{\boldsymbol{X}} f = \frac{\partial f(\boldsymbol{X})}{\partial \boldsymbol{X}} = \begin{bmatrix} \frac{\partial f(\boldsymbol{X})}{\partial X_{1,1}} & \cdots & \frac{\partial f(\boldsymbol{X})}{\partial X_{1,m}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f(\boldsymbol{X})}{\partial X_{n,1}} & \cdots & \frac{\partial f(\boldsymbol{X})}{\partial X_{n,m}} \end{bmatrix}$$

Additional: Function Differentiation

Vector Function w.r.t. Vector

- $\boldsymbol{f}\colon \mathbb{R}^n \to \mathbb{R}^m$
- Jacobian matrix $\nabla_{\boldsymbol{x}} f \in \mathbb{R}^{m \times n}$

$$\nabla_{\boldsymbol{x}} f = \frac{\partial \boldsymbol{f}(\boldsymbol{x})}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial f_1(\boldsymbol{x})}{\partial x_1} & \cdots & \frac{\partial f_1(\boldsymbol{x})}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m(\boldsymbol{x})}{\partial x_1} & \cdots & \frac{\partial f_m(\boldsymbol{x})}{\partial x_n} \end{bmatrix}$$

Vector Function w.r.t. Matrix

- $\boldsymbol{f}\colon \mathbb{R}^{n \times m} \to \mathbb{R}^m$
- $\nabla_{\boldsymbol{X}} \boldsymbol{f} = \frac{\partial \boldsymbol{f}(\boldsymbol{X})}{\partial \boldsymbol{X}} = \cdots =$ 3-dimensional tensor

- **Special case**: Can be expressed as a dyadic product for the case we are considering here, and thus corresponds to a matrix again.

$$\boldsymbol{f}(\boldsymbol{W}) = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

$$g\colon \mathbb{R}^m \to \mathbb{R}, \qquad g = g\big(\boldsymbol{f}(\boldsymbol{W})\big)$$

$$\nabla_{\boldsymbol{W}} g = \frac{\partial g}{\partial \boldsymbol{W}} = \frac{\partial g}{\partial \boldsymbol{f}} \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{W}} = \frac{\partial g}{\partial \boldsymbol{f}} \boldsymbol{x}^T$$

Outline

- Introduction
- Perceptron
- Deep Neural Networks
- Learning
- Applications
- Literature

What's new today?

Today

- Sensors, high-performance compute units, GPU ...
- Data from (many) hundreds of km, from fleets of vehicles
- Advanced algorithms, e.g. deep learning

Still not enough!

- Mastering all "normal" situations
- Between 100 million and 5 billion km are required for a release

[Prof. H. Winner]

Unknown objects

Unknown behavior

Unknown "flying" objects

Outline

- Introduction
- Perceptron
- Deep Neural Networks
- Learning
- Applications
- Literature

Literature

Tom Mitchell: Machine Learning, McGraw Hill, 1997

- Covers almost all the basics of machine learning
  http://www.cs.cmu.edu/~tom/mlbook.html

Ian Goodfellow and Yoshua Bengio and Aaron Courville: Deep Learning, MIT Press, 2016,

- More modern book that shows the progress of neural networks up to 2016 very well
  https://www.deeplearningbook.org/

Kevin P. Murphy: Probabilistic Machine Learning: An Introduction, MIT Press, 2022

- Modern book with great explanations of mathematical foundations, deep learning, machine learning methods. No reinforcement learning.
  https://github.com/probml/pml-book/releases/latest/download/book1.pdf

Literature

- C. Hubschneider, A Bauer, M Weber, JM Zöllner**: Adding navigation to the equation: Turning decisions for end-to-end vehicle control**, Intelligent Transportation Systems (ITSC), 2017