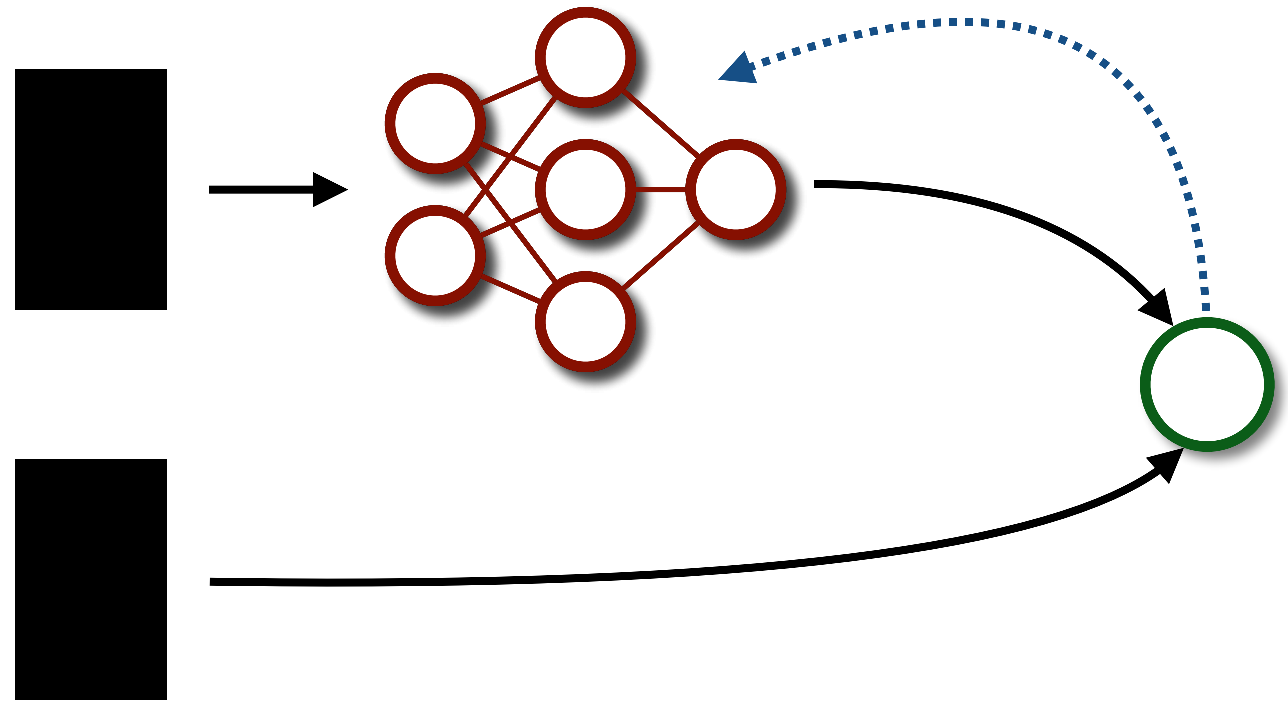


Neural Network Building Blocks (2/3)

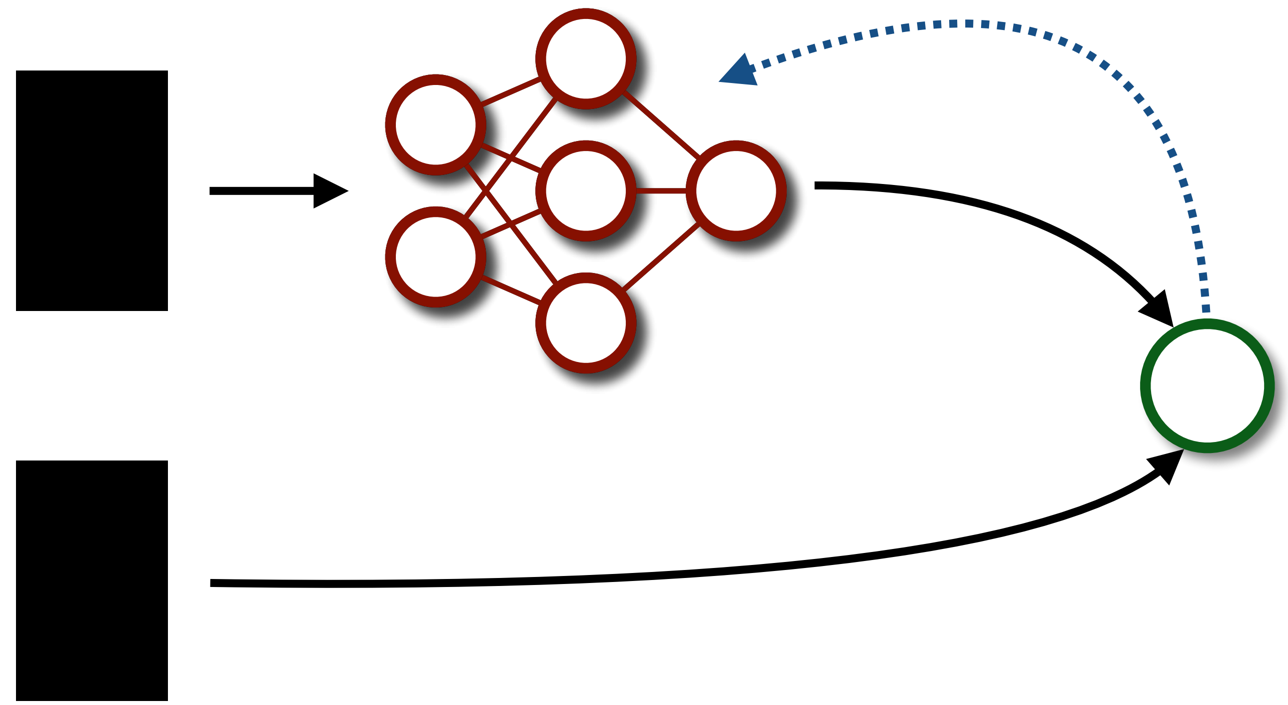
Docent: Dennis Noll (RWTH)
Tutor: Boyang Yu (LMU)

Deep Learning School "Basic Concepts"

09.08.22

RWTH AACHEN UNIVERSITY

Recap Yesterday

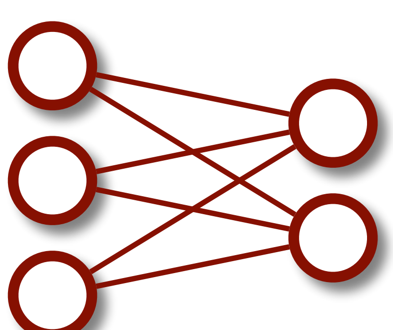


Model

- Linear Perceptron

  $y = w \cdot x + b$

  Parameter $\theta = (w, b)$



Objective

- Fit between training data and prediction
- Regression (mse):

  $\mathscr{L} = (y_{true} - y_{pred})^2$

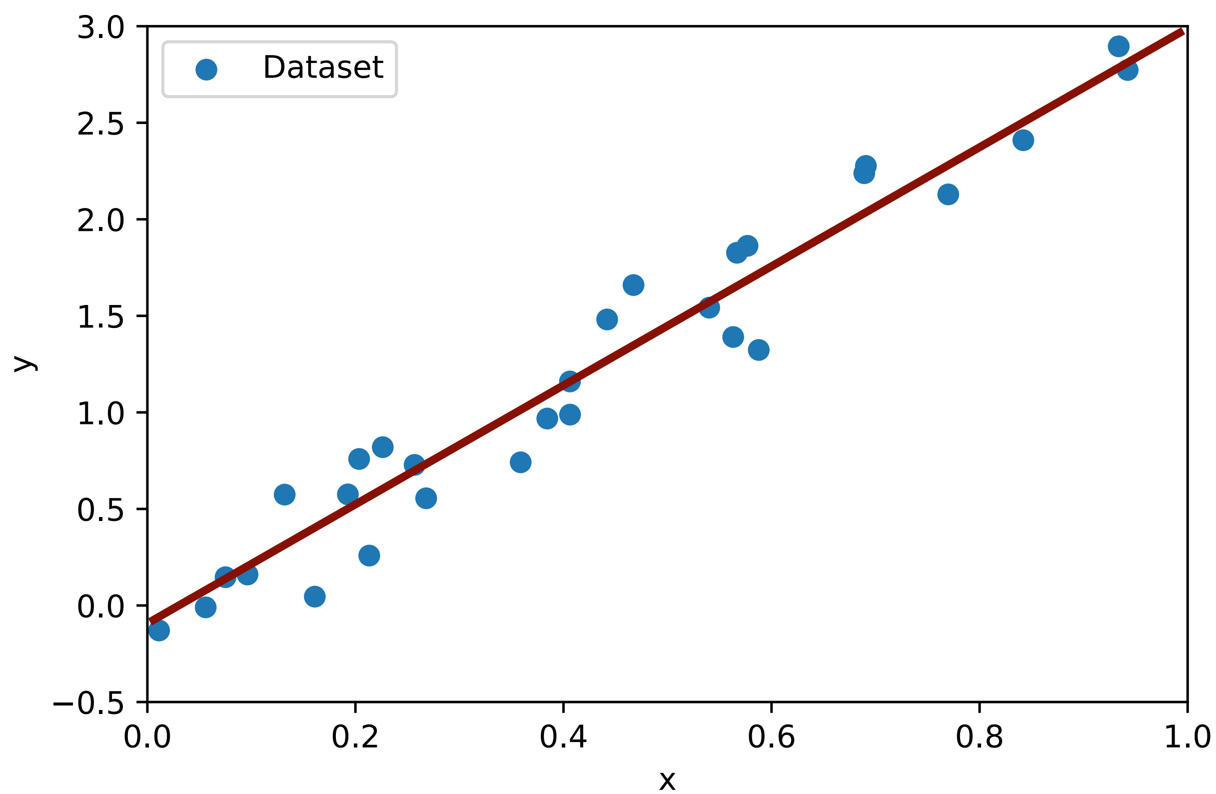
Training

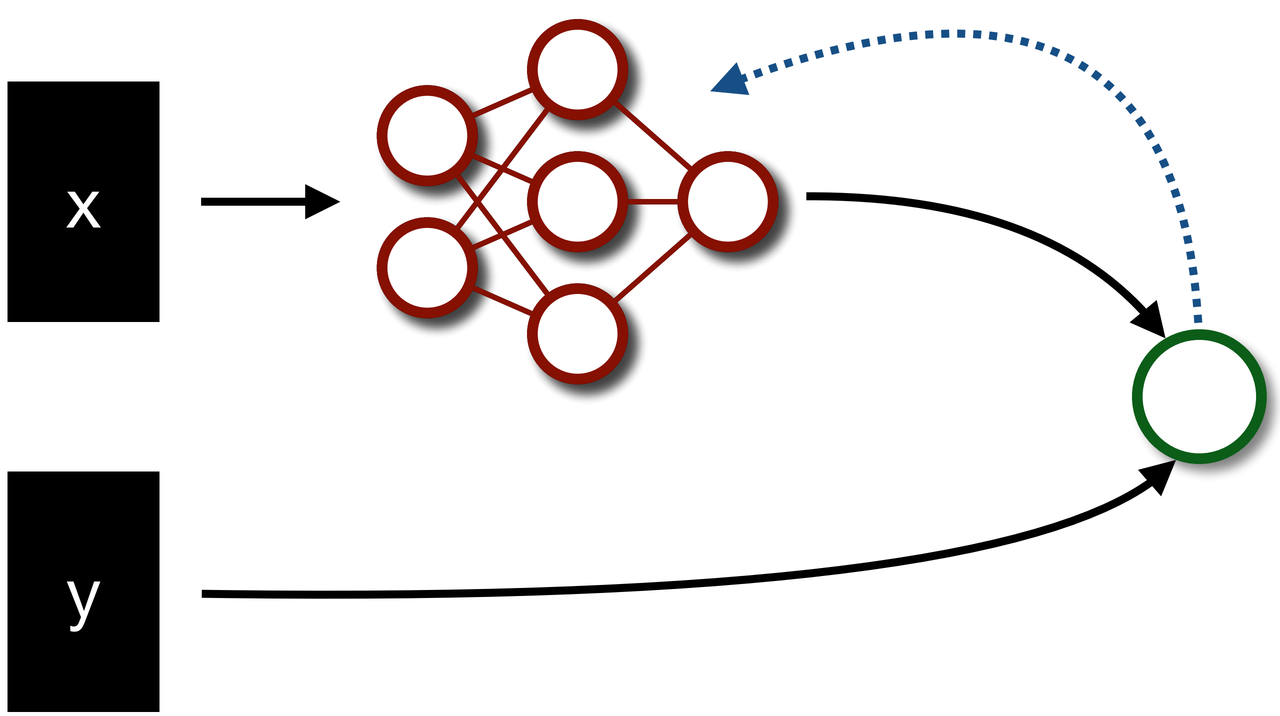
- Find parameters which minimize loss ($\mathscr{L}$)
- Gradient descent

  $$\theta \rightarrow \theta - \alpha \frac{d\mathscr{L}}{d\theta}$$

Approximate N-Dim Arbitrary Functions

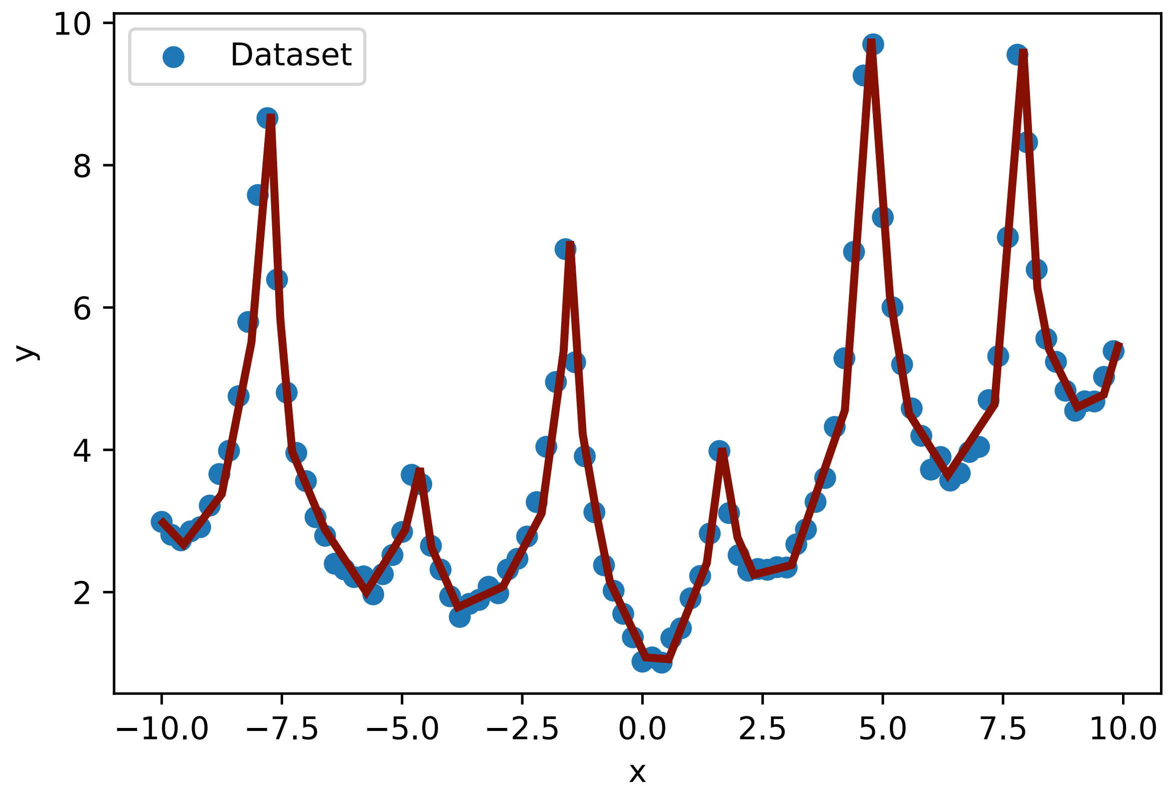
- Find **rules** which connect Data→Answers (x→y)



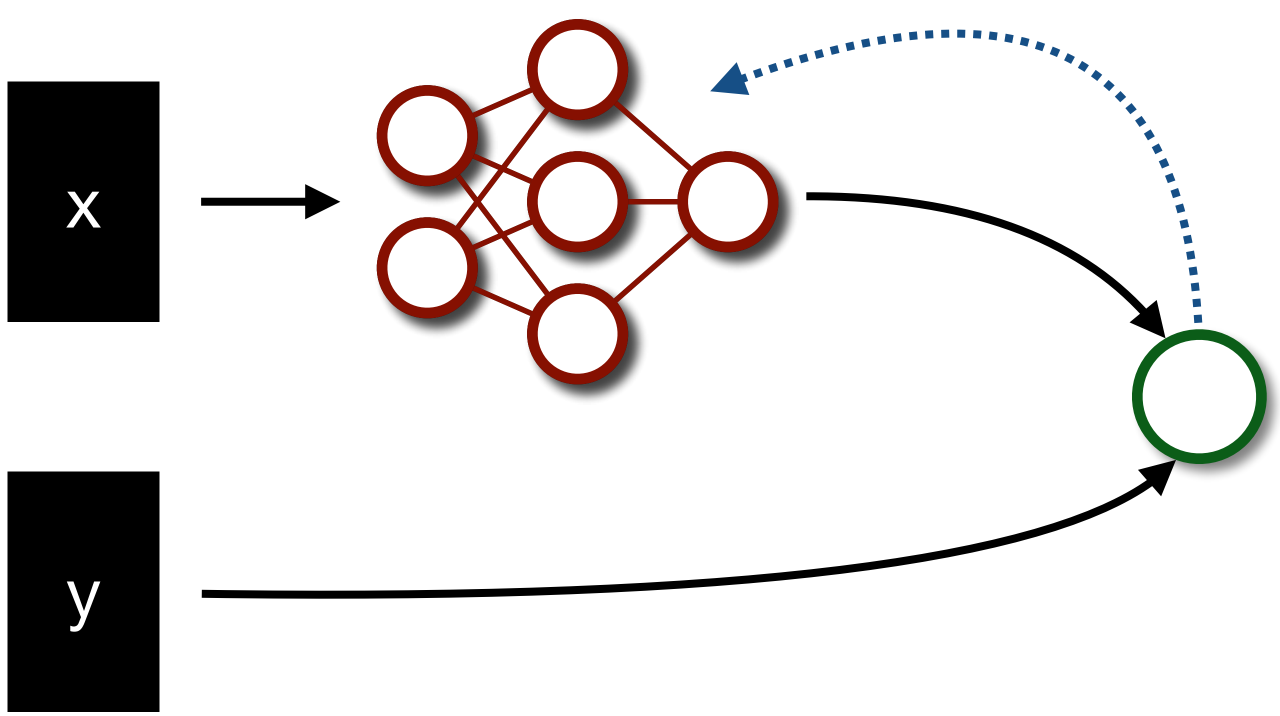


Approximate N-Dim Arbitrary Functions

- Find **rules** which connect Data→Answers (x→y)

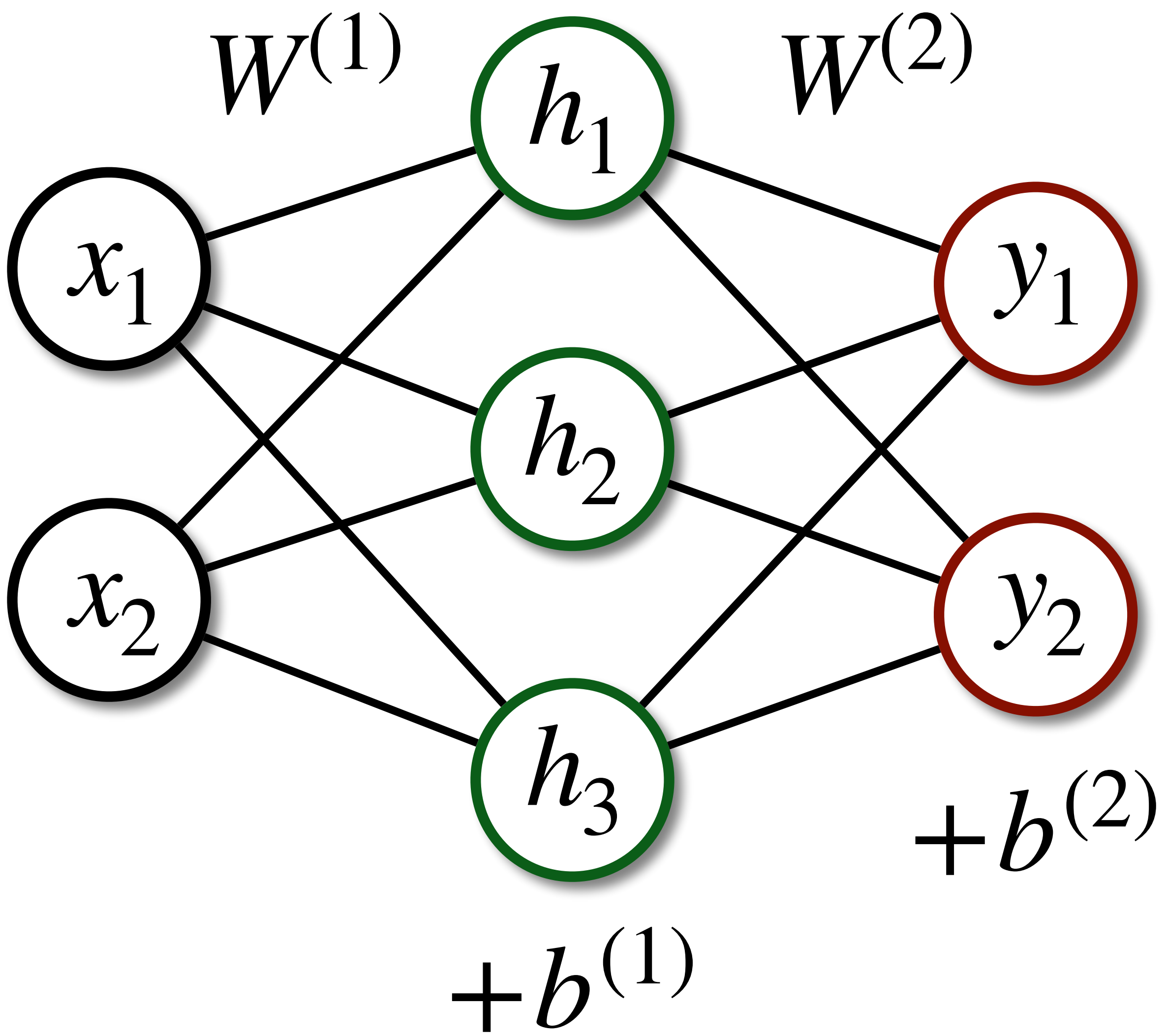


Need:
Non-linear model!



Non Linear Network Models

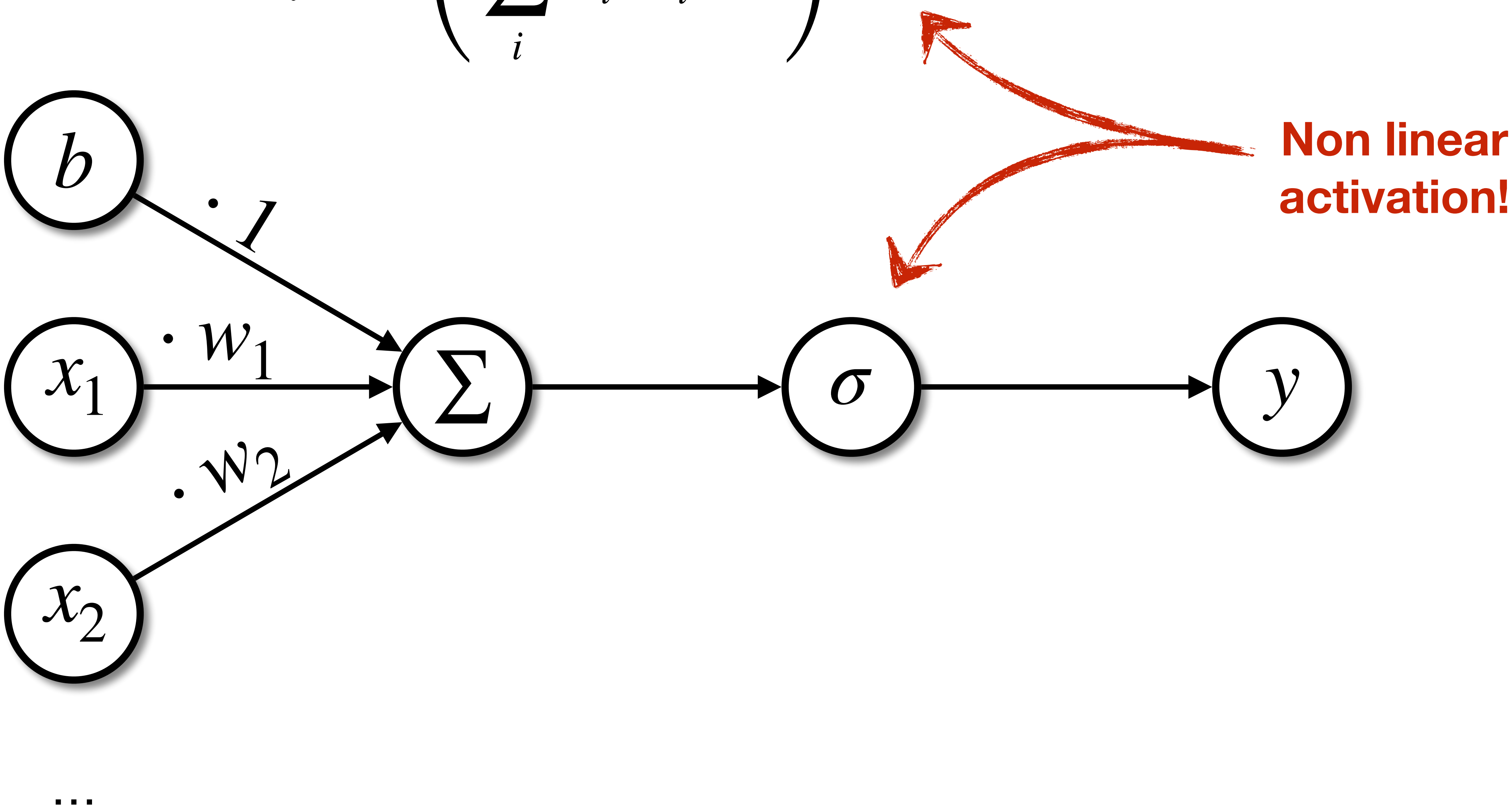
- Need non-linear model
- Chain layers
  - $h = W^{(1)}x + b^{(1)}$
  - $y = W^{(2)}h + b^{(2)}$
  - Model is still linear:
    - $y = W^{(2)}(W^{(1)}x + b^{(1)}) + b^{(2)}$
    - $y = \underbrace{W^{(2)}W^{(1)}}_{W}x + \underbrace{W^{(2)}b^{(1)} + b^{(2)}}_{b}$



- Solution: Apply non linear activation function σ to each element:
  - $h' = \sigma(W^1 x + b^1)$

The Perceptron (revisited)

- Approximate **Non-**Linear Function $f : \mathbb{R}^N \rightarrow \mathbb{R}^1$
- Parametrizable (N+1 parameters):
  - Weights: $w_1, w_2, \ldots, w_N$
  - Bias: b
- Functional form: $y = \sigma\left(\sum_i w_i \cdot x_i + b\right) = \sigma(Wx + b)$



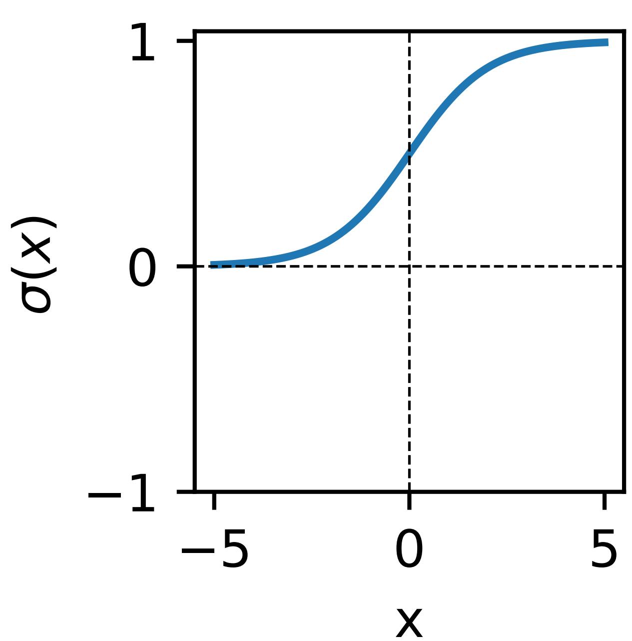
Activation Functions

- Non-linear perceptron: $y = \sigma(Wx + b)$ with non-linear activation function σ
- Many different possibilities, three common examples: Sigmoid, ReLU, Tanh

Sigmoid

Logistic function

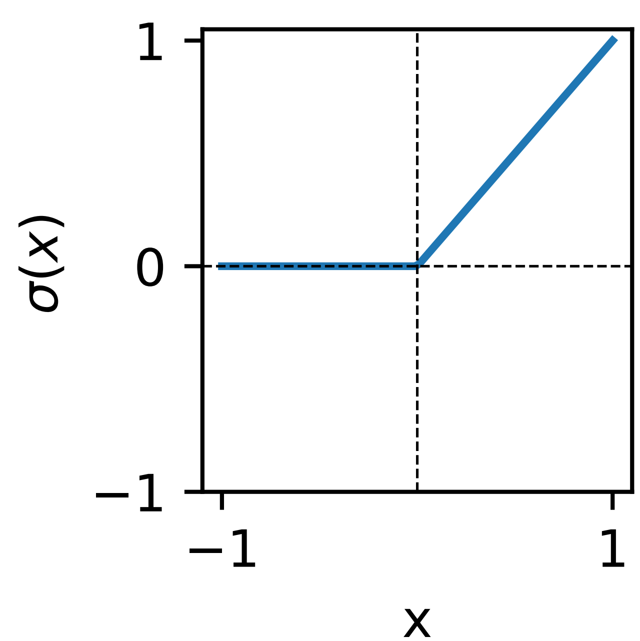
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



ReLU

Rectified Linear Unit

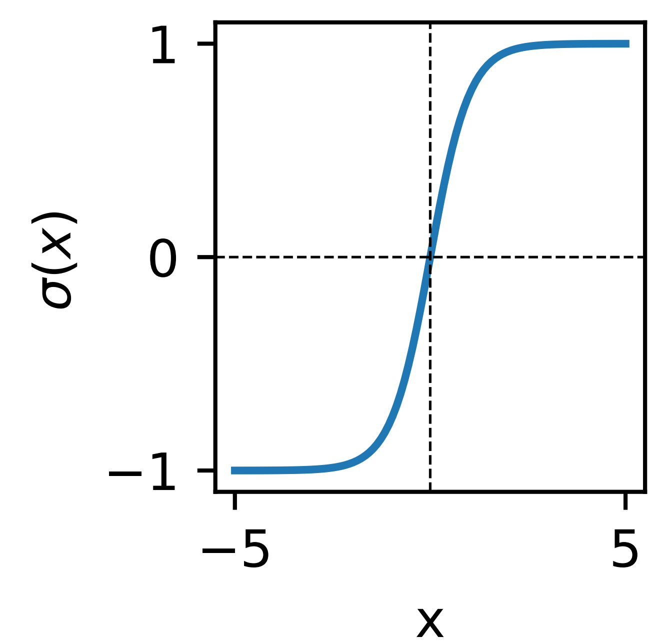
$$\sigma(x) = \max(0,x)$$



Tanh

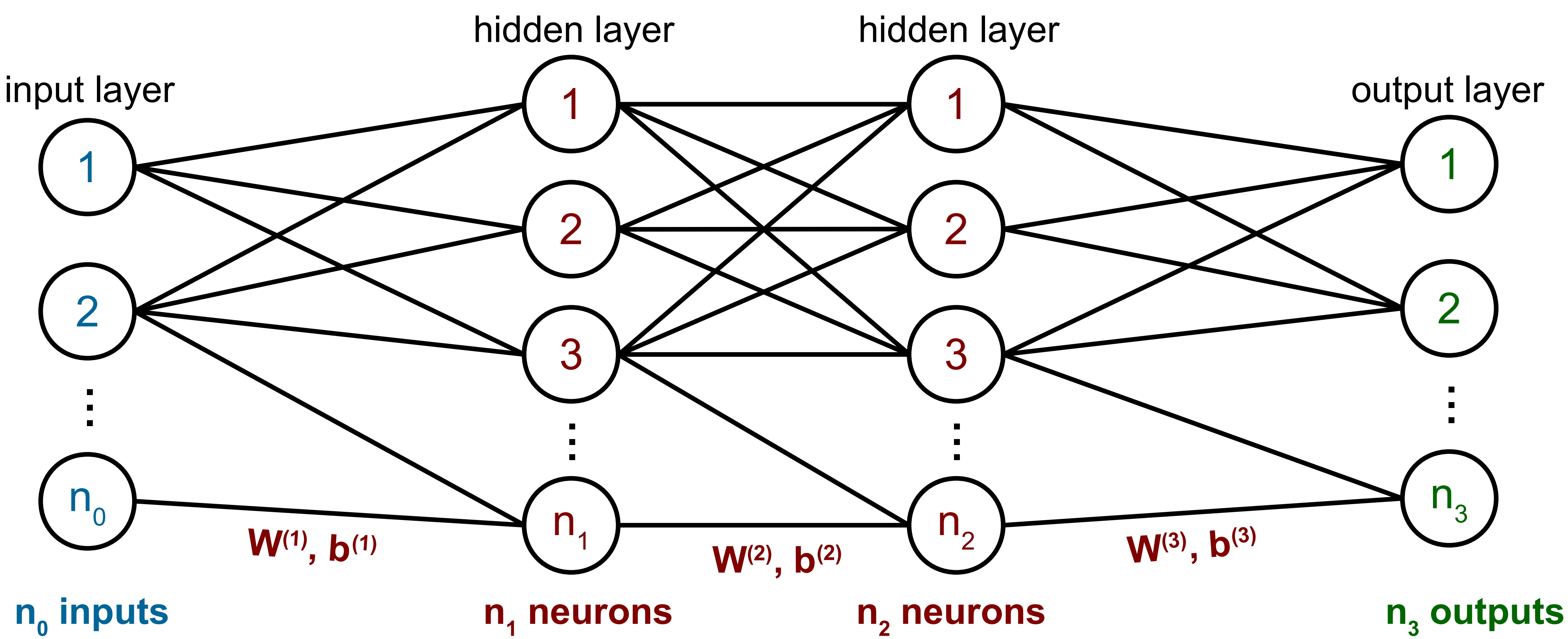
Hyperbolic tangent

$$\sigma(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$



Neural Network Architecture

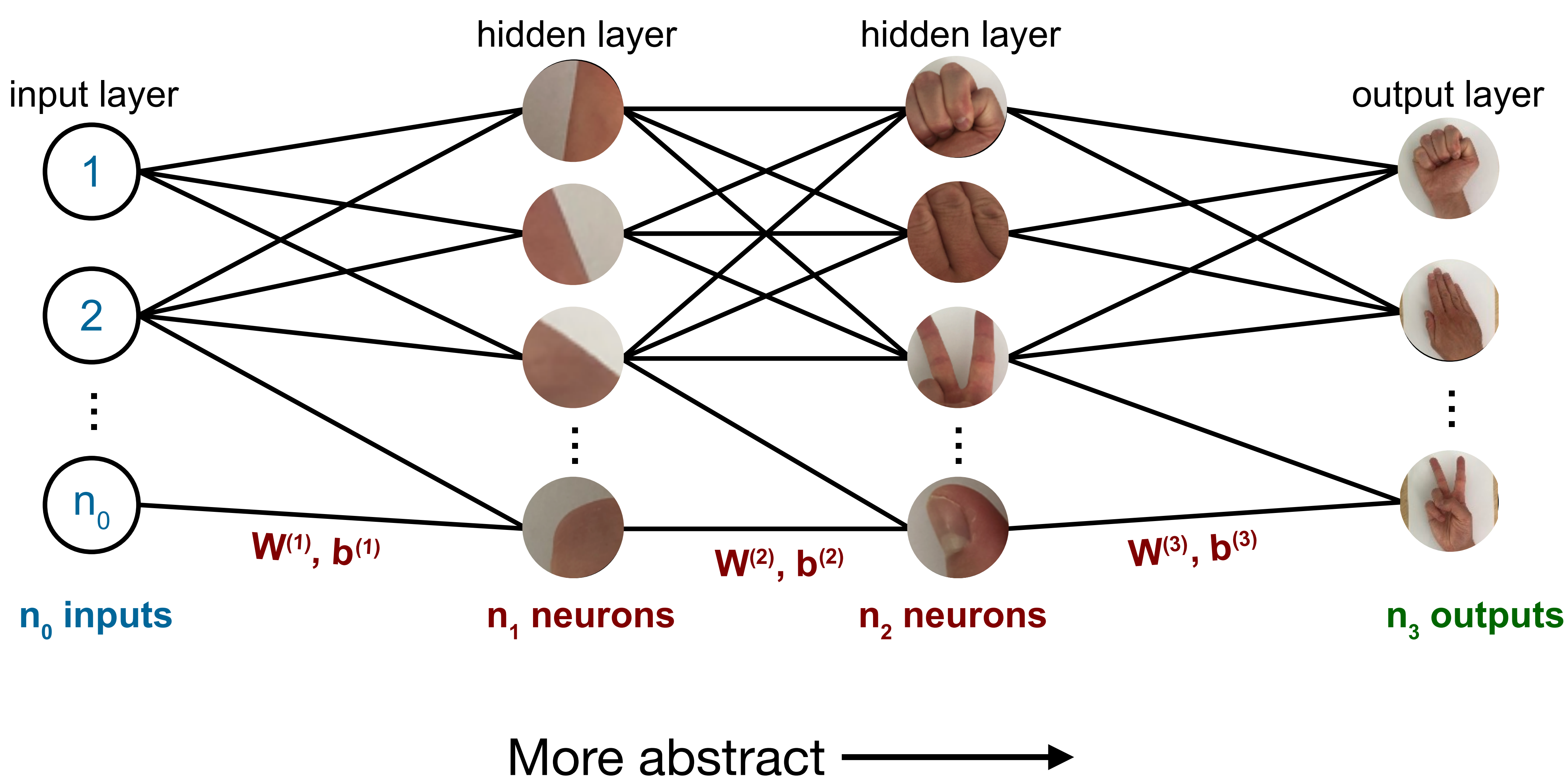
- Build network out of **nodes/neurons** $\sigma(Wx + b)$
  - Strength of connections between nodes in specified by **weight matrix W**
  - **Width**: number of nodes per layer
  - **Depth**: number of layers holding weights



[1]

Neural Network Architecture

- Build network out of **nodes/neurons** $\sigma(Wx + b)$
  - Strength of connections between nodes in specified by **weight matrix W**
  - **Width**: number of nodes per layer
  - **Depth**: number of layers holding weights
- Each new layer can extract more abstract features

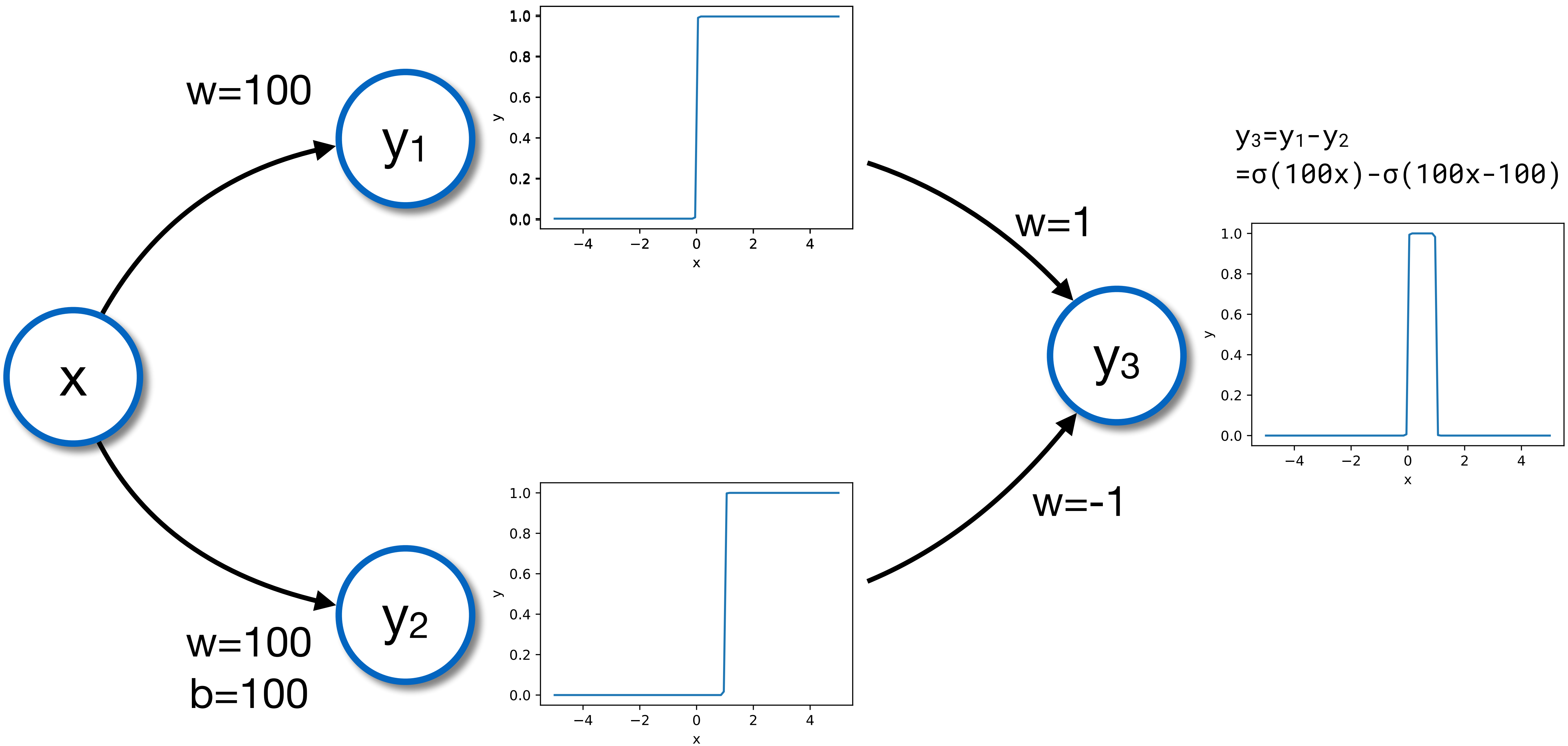


[1]

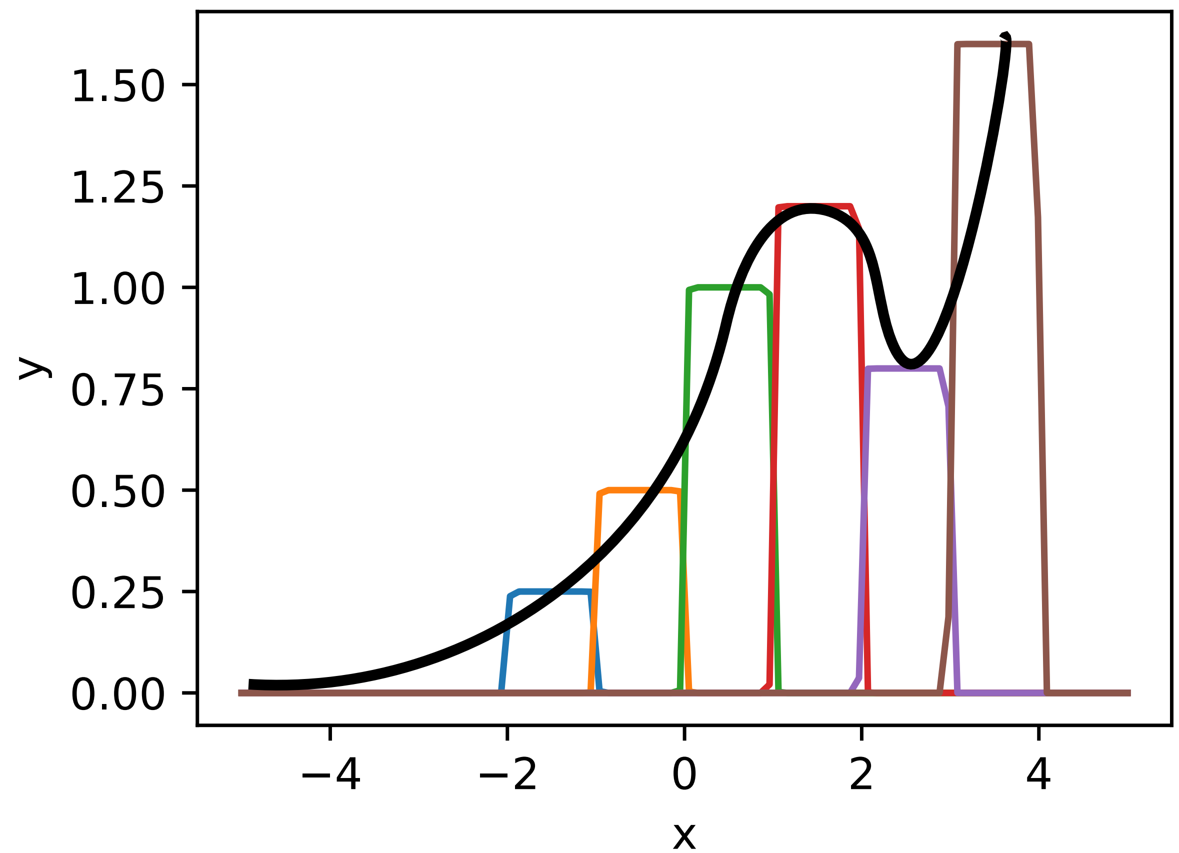
RWTH AACHEN UNIVERSITY
Dennis Noll - 09.08.22

Universal Approximation Theorem (1/2)

- A **neural network** with one hidden layer with a finite number of nodes **can (in theory) approximate any reasonable function** to arbitrary precision



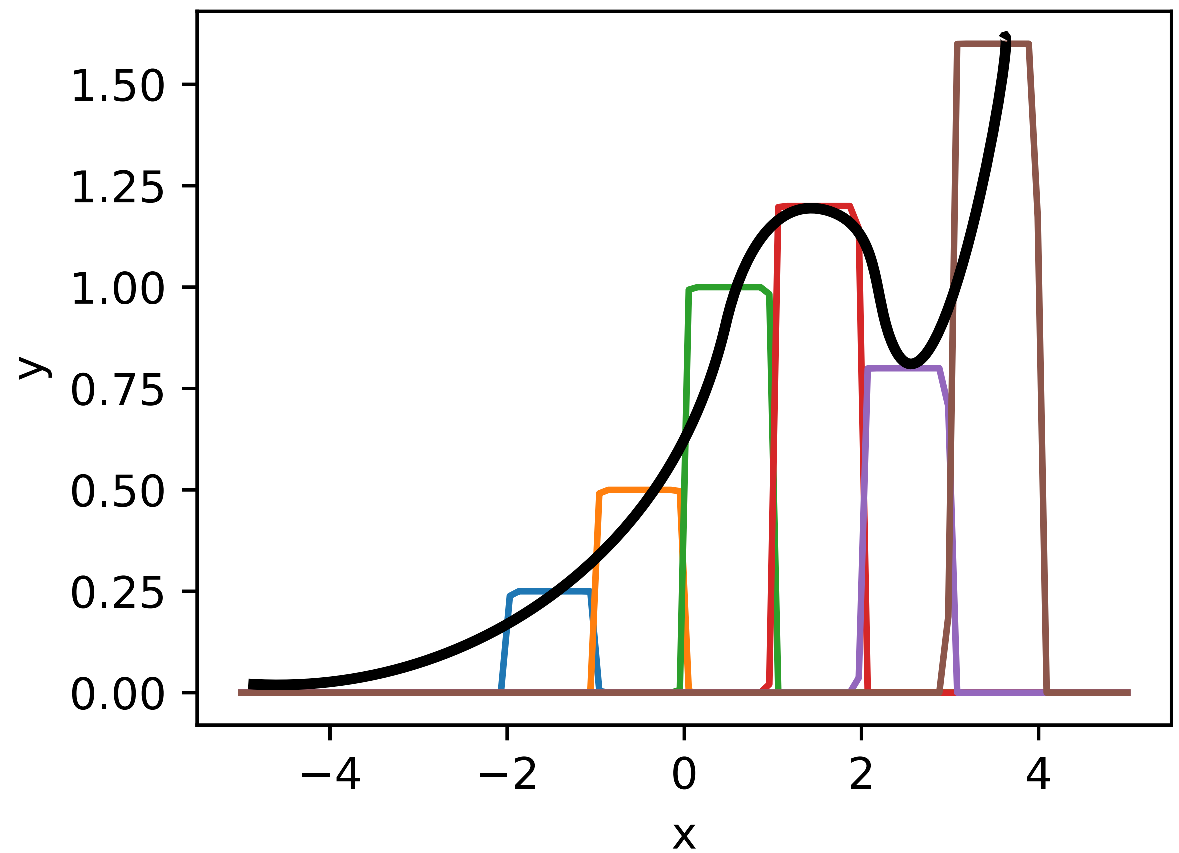
- A **neural network** with one hidden layer with a finite number of nodes **can (in theory) approximate any reasonable function** to arbitrary precision



$$
\begin{aligned}
f(x) = \; & 0.25\,(\sigma(100x + 200) - \sigma(100x + 100)) \\
& + 0.5\,(\sigma(100x + 100) - \sigma(100x + 0)) \\
& + 1.0\,(\sigma(100x + 0) - \sigma(100x - 100)) \\
& + 1.2\,(\sigma(100x - 100) - \sigma(100x - 200)) \\
& + 0.8\,(\sigma(100x - 200) - \sigma(100x - 300)) \\
& + 1.6\,(\sigma(100x - 300) - \sigma(100x - 400))
\end{aligned}
$$

Universal Approximation Theorem (2/2)

- A **neural network** with one hidden layer with a finite number of nodes **can (in theory) approximate any reasonable function** to arbitrary precision



But:

- No statement on number of nodes
- Shallow models hard to train

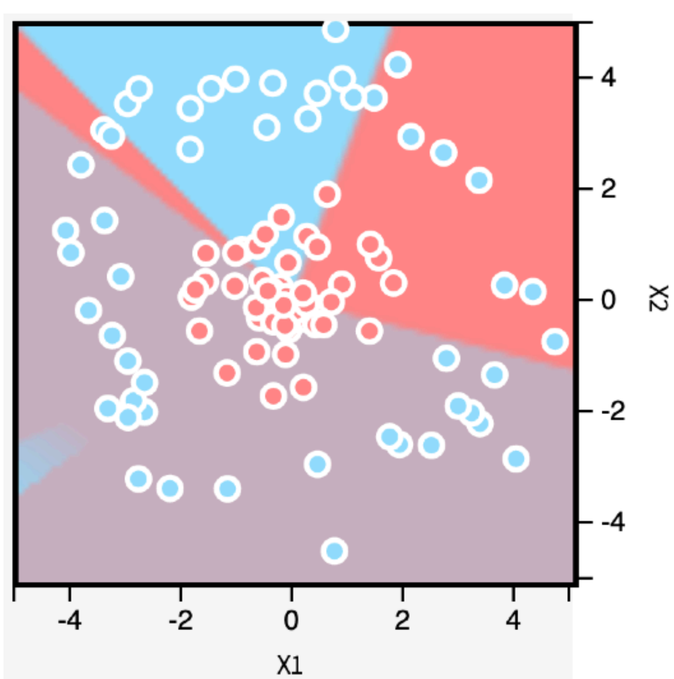
$\Rightarrow$ **Train Deep Models!**

Parameter Initialization

- Initialization of model parameters critical for performance
- Choose Gaussian distributed initial weights
- Two standard initializations:
  - Sigmoid, Tanh: $\sigma^2 = 2/(n_{in} + n_{out})$
  - ReLU: $\sigma^2 = 2/n_{in}$

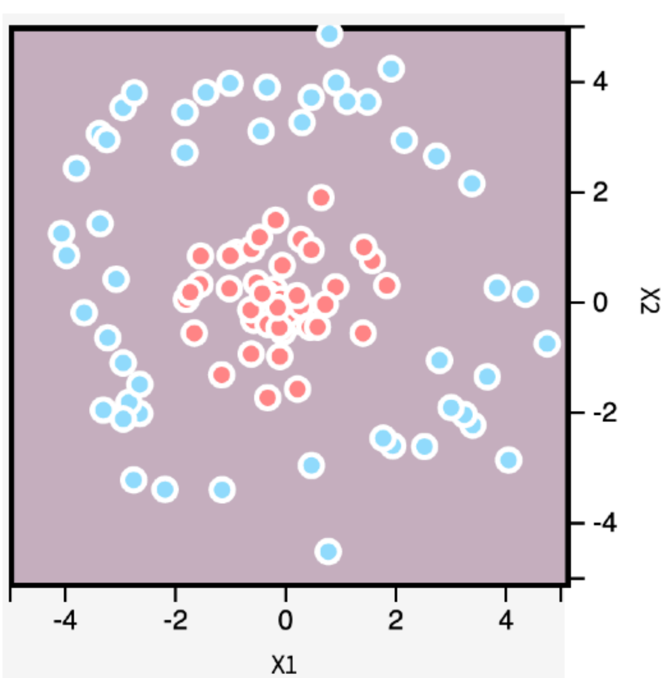
Weights too large

exploding signals



Weights too small

vanishing signals



[2]

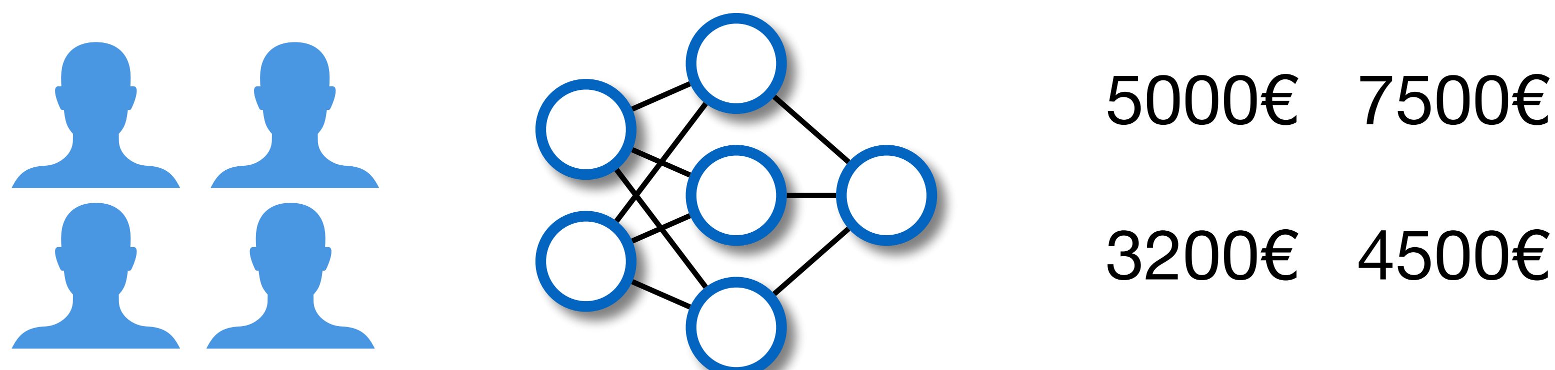
Regression and Classification

Dennis Noll - 09.08.22

- Two different tasks of supervised learning
- Different architecture, objective and training

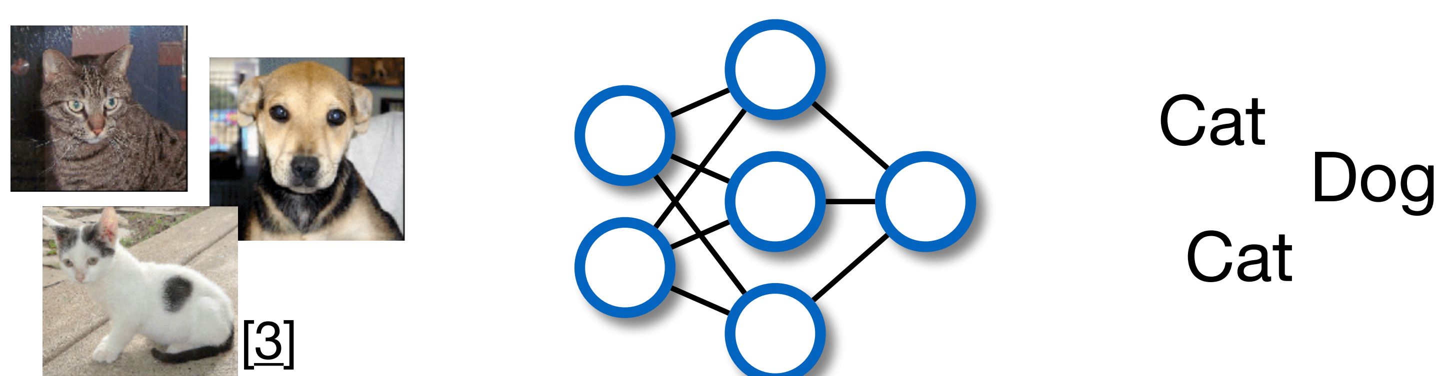
Regression

- Predict continuous variable (e.g. Student → Future net income)



Classification

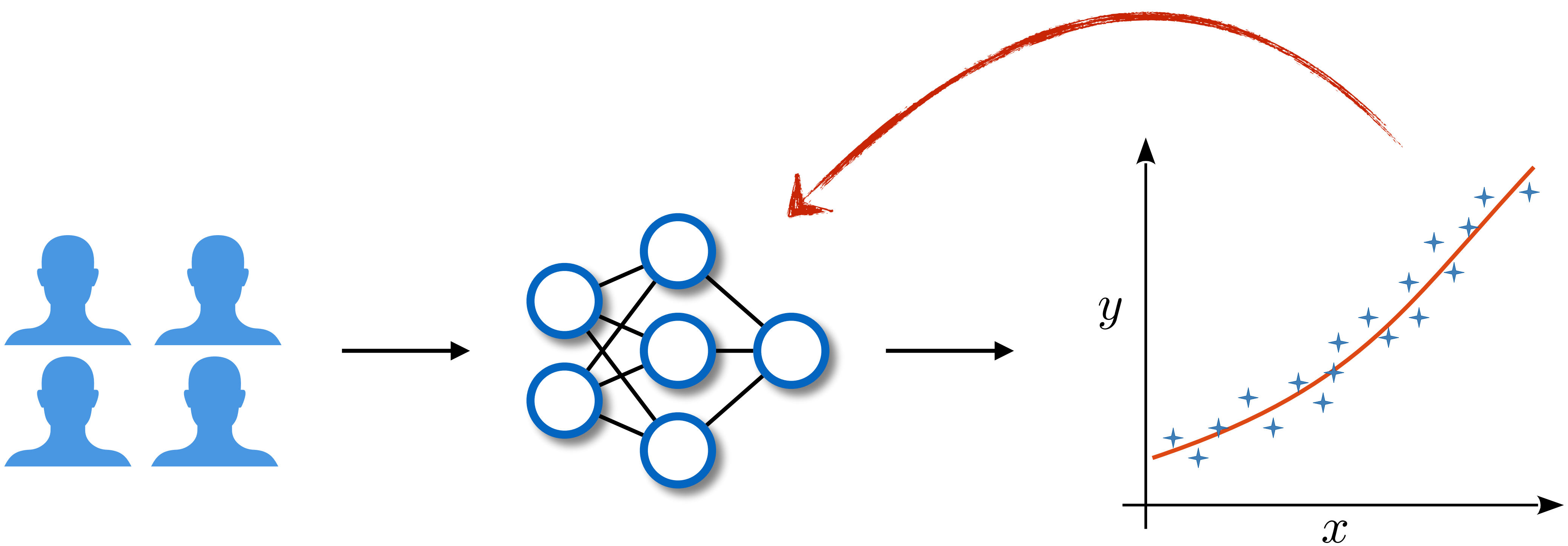
- Predict discrete class (e.g. Picture → Cat/Dog)



Regression: Predict continuous variables

- Predict a real number associated with a feature vector
- Example:
  - Prediction: What is the future net income of the students?
  - Input: Grade in course, Age, Participation
- Last activation: Linear (no activation)
- Loss: Mean squared error

$$\mathscr{L} = \frac{1}{n}\sum_{i}^{n}(y_i^{true} - y_i^{pred})^2$$



Regression (Example)

- Example: Dataset with n=3 samples

$$\mathscr{L} = \frac{1}{n} \sum_{i}^{n} (y_i^{true} - y_i^{pred})^2$$

Grade in course: A
Age: 32
Participation: Awesome

i = 3	Income [k€]
True y^{true}	9
Pred y^{pred}	8
$(y^{true}-y^{pred})^2$	1

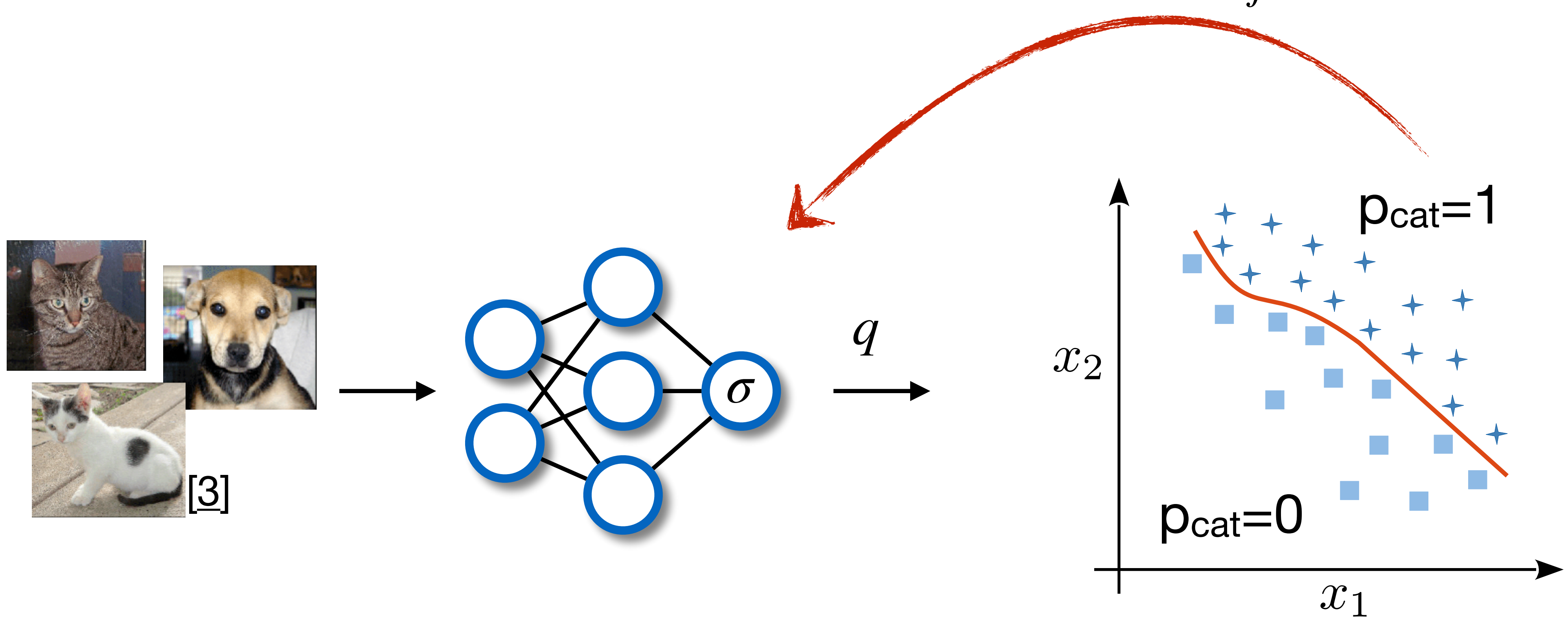
$$\mathscr{L}_3 = 1$$

$$\mathscr{L} \propto 0.25 + 4 + 1 = 5.25$$

Classification: Predict discrete classes

- Predict a discrete value (label) associated with a feature vector
- Example:
  - Prediction: Does this picture show a cat or a dog?
  - Input: Pixels of image
- Last activation: Sigmoid/softmax (probability $q \in [0,1]$)
- Loss: Cross-Entropy with c classes

$$\mathscr{L} = -\frac{1}{n}\sum_{i}^{n}\sum_{j}^{c} p_{ij}\cdot\log(q_{ij})$$



RWTH AACHEN UNIVERSITY
Dennis Noll - 09.08.22

- Example: Dataset with n=3 samples of c=2 classes (cats and dogs)

$$\mathscr{L}(\theta) = -\frac{1}{n}\sum_{i}^{n}\sum_{j}^{c} p_{ij} \cdot \log(q_{ij})$$


[3]

$i = 3$	Cat	Dog
True p_{3j}	1	0
Pred q_{3j}	90 %	10 %
$\log(q_{3j})$	-0.1	-2.3

$$\mathscr{L}_3 = 0.1$$

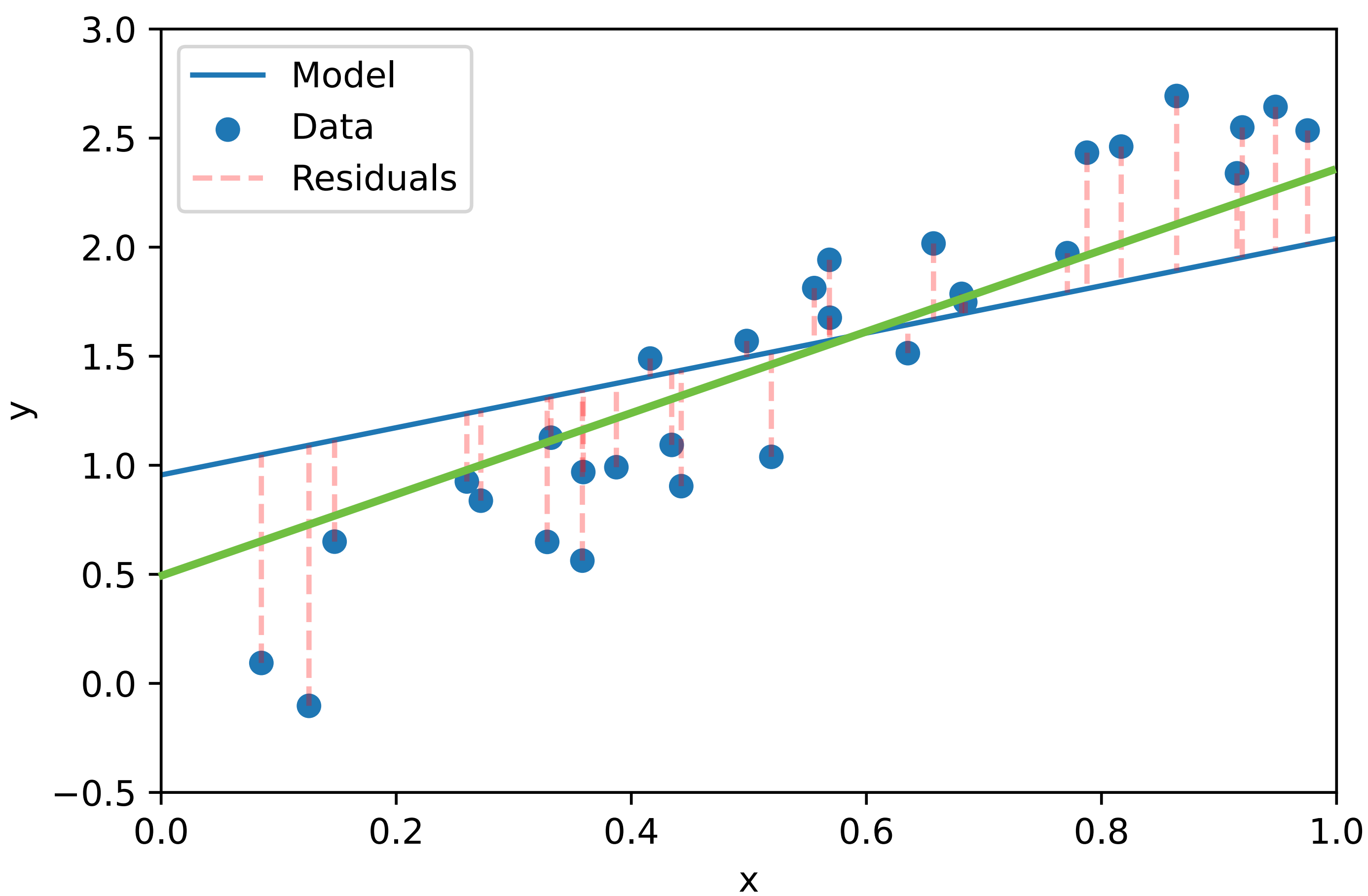
$$\mathscr{L} \propto 0.5 + 0.4 + 0.1 = 1.0$$

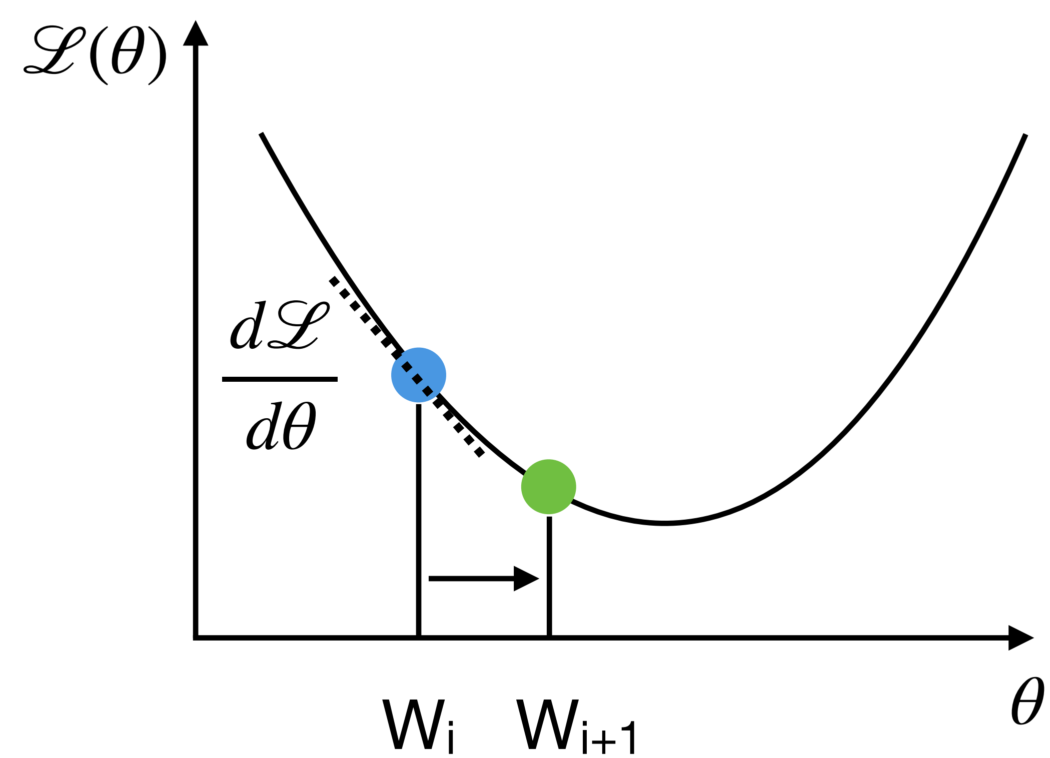
- Minimize objective function $\mathscr{L}(\theta)$
- Update model (θ) in opposite direction of gradient iteratively

$$\theta \to \theta - \alpha \frac{d\mathscr{L}}{d\theta}$$

Step size (learning rate)

Gradient

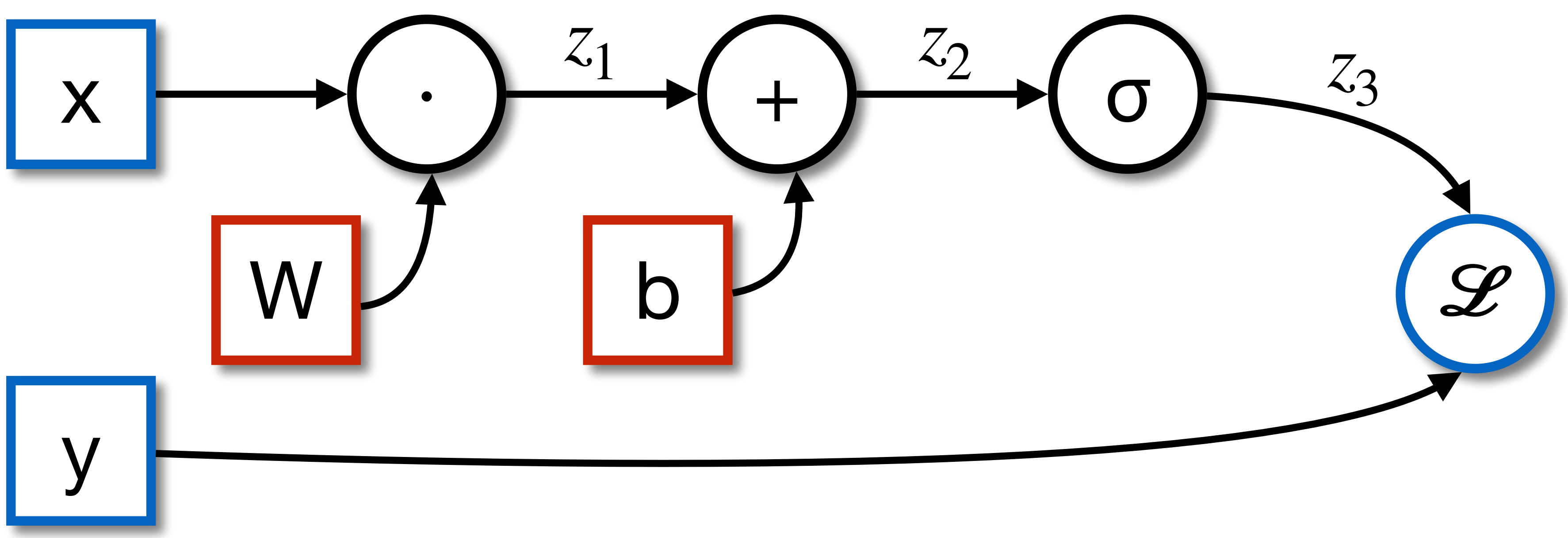




Backpropagation (Calculate $d\mathcal{L}/d\theta$)

- Each network is a series of (simple) mathematical operations
- Each operation has:
  - Local output (forward pass)
  - Local derivative (backward pass)
- Use chain rule to evaluate derivatives $d\mathcal{L}/d\theta_i$ for every parameter θ_i

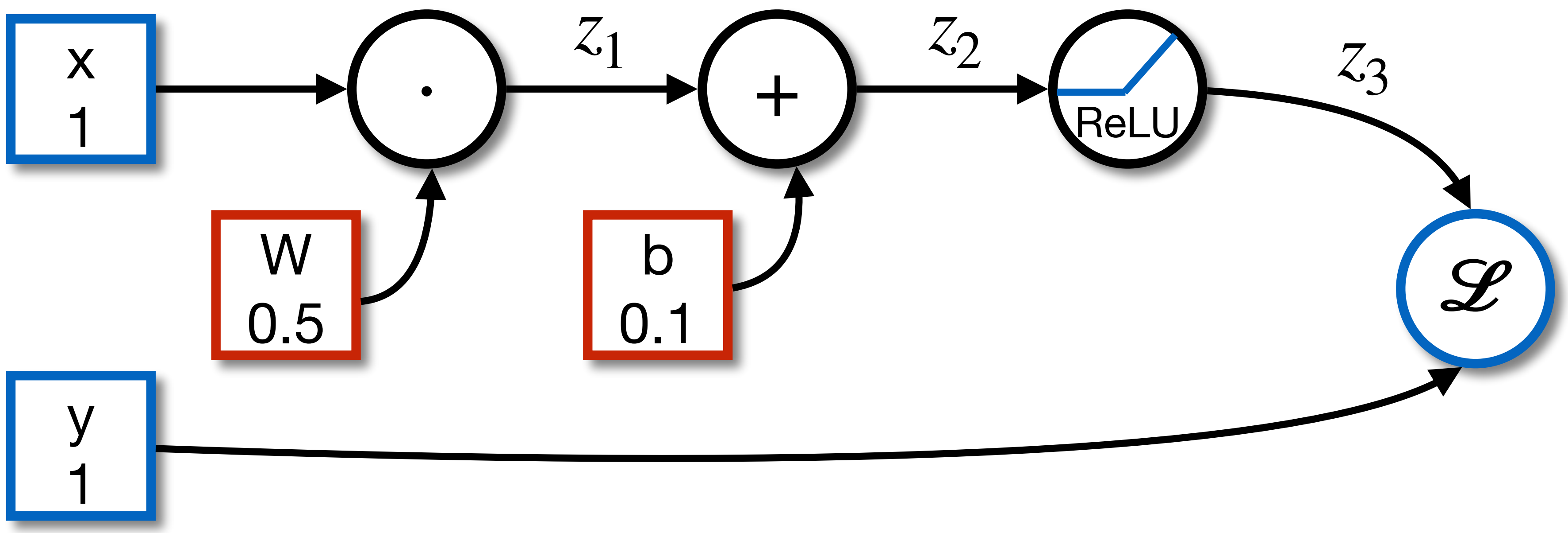
Example: $y^{pred} = z_3 = \sigma(Wx + b)$



$$\partial\mathcal{L}/\partial W = \partial\mathcal{L}/\partial z_3 \cdot \partial z_3/\partial z_2 \cdot \partial z_2/\partial z_1 \cdot \partial z_1/\partial W$$

Backpropagation (Example)

$$\partial \mathcal{L} / \partial W = \partial \mathcal{L} / \partial z_3 \cdot \partial z_3 / \partial z_2 \cdot \partial z_2 / \partial z_1 \cdot \partial z_1 / \partial W$$



Forward pass

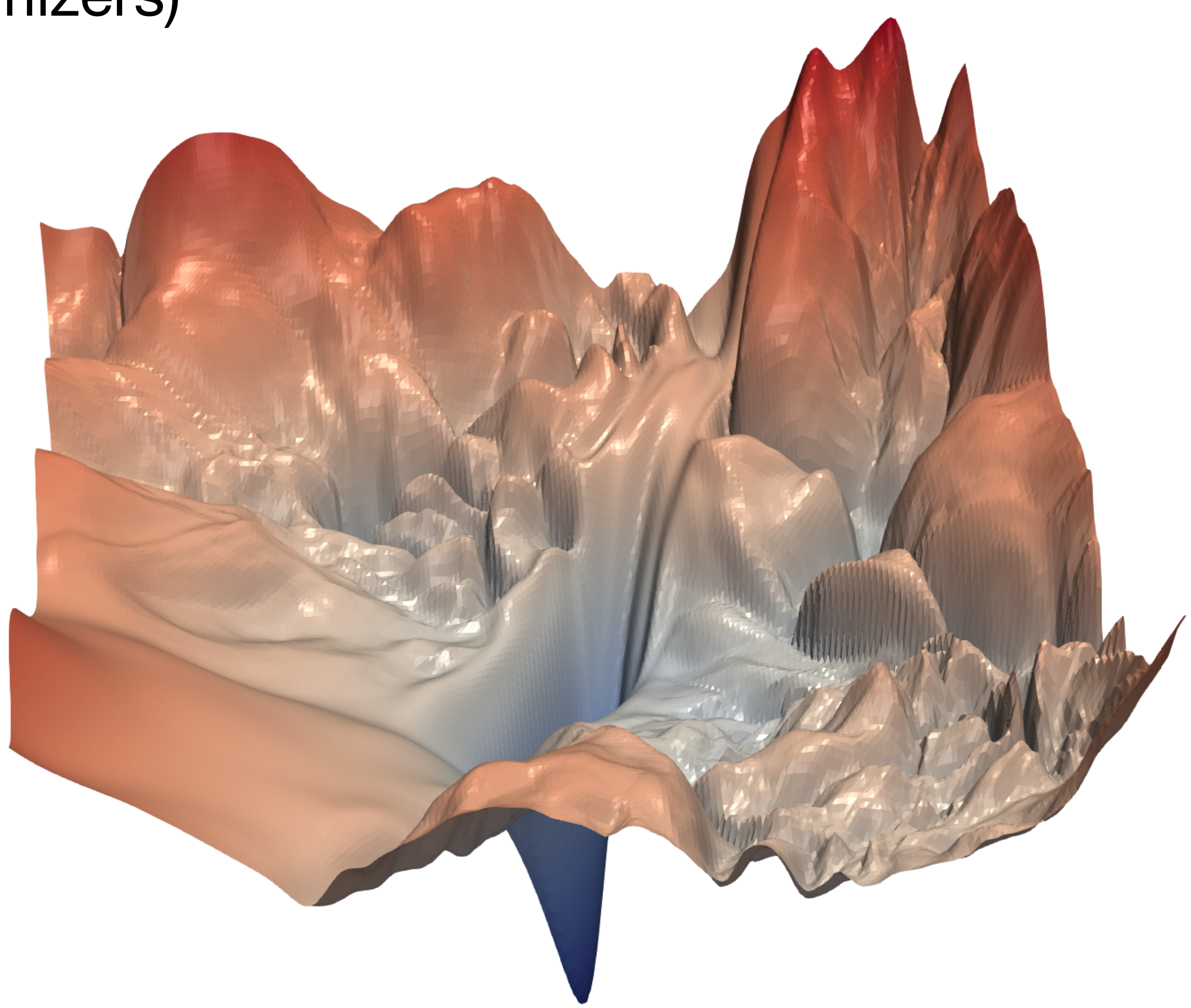
$$z_1 = Wx = 0.5$$
$$z_2 = z_1 + b = 0.6$$
$$z_3 = \sigma(z_2) = \text{ReLU}(z_2) = 0.6$$
$$\mathcal{L}(z_3) = (z_3 - y)^2 = 0.16$$

Backward pass

$$\partial \mathcal{L} / \partial z_3 = 2(z_3 - y) = -0.8$$
$$\partial z_3 / \partial z_2 = \partial \sigma(z_2) / \partial z_2 = 1$$
$$\partial z_2 / \partial z_1 = 1$$
$$\partial z_1 / \partial W = x = 1$$
$$\Rightarrow \partial \mathcal{L} / \partial W = -0.4 \cdot 1 \cdot 1 \cdot 1 = -0.4$$

Loss Landscape

- Loss (one number) describes how O(1k-1b) parameters must change
- Loss landscape can look very complicated (e.g. local minima)
- Different improvements possible:
    - Model (Architectures)
    - Loss (Regularisation)
    - Training (Optimizers)



[4]

Stochastic Gradient Descent

- Until now: Calculation of loss and gradient based on whole dataset
- New idea: Approximate loss and gradient on subset of dataset (mini-batch)

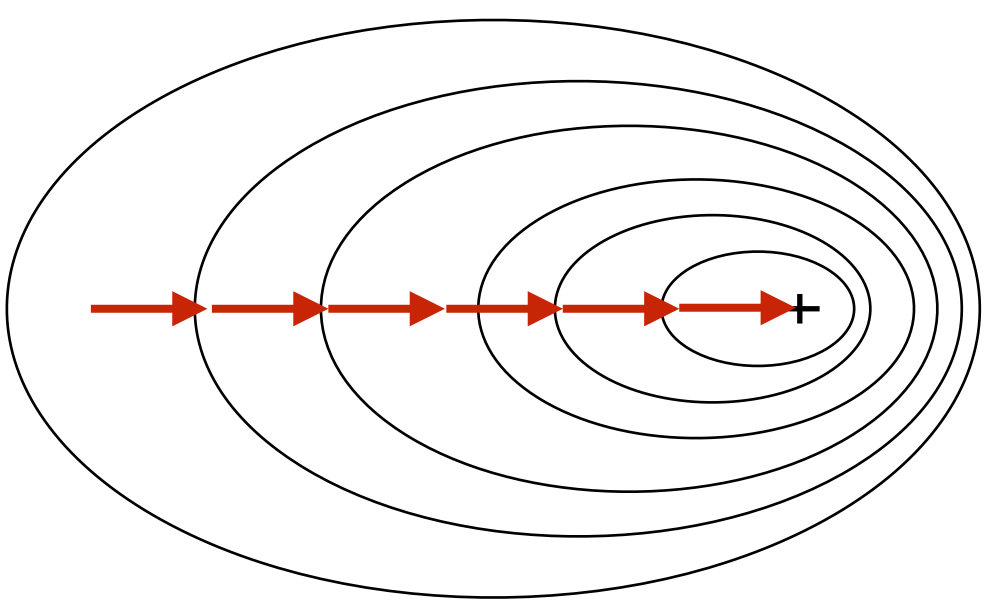
Pro

- More parameter updates
- Stochasticity helps escaping local minima

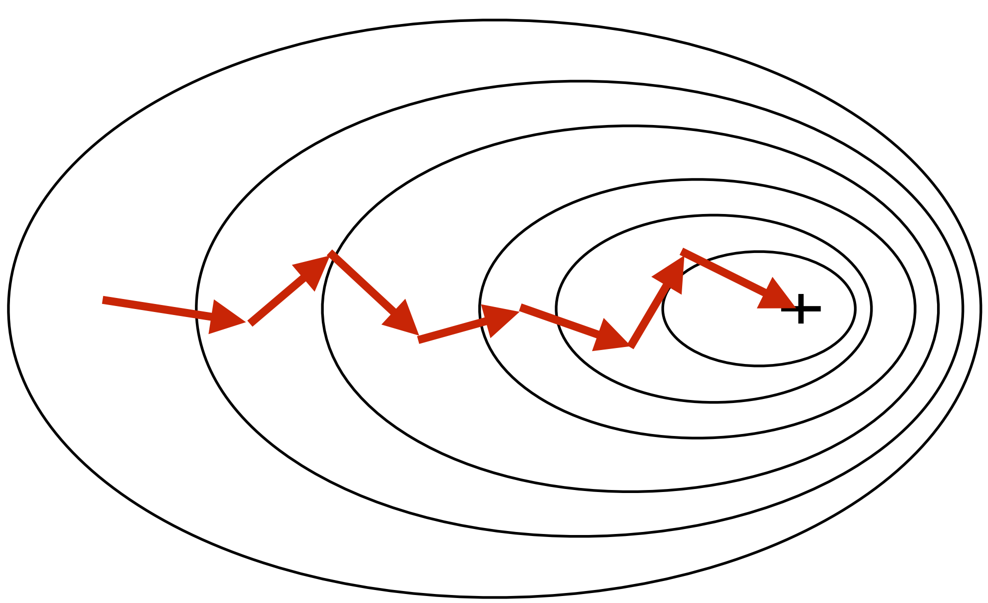
Contra

- Gradient not exact, however in practice good enough

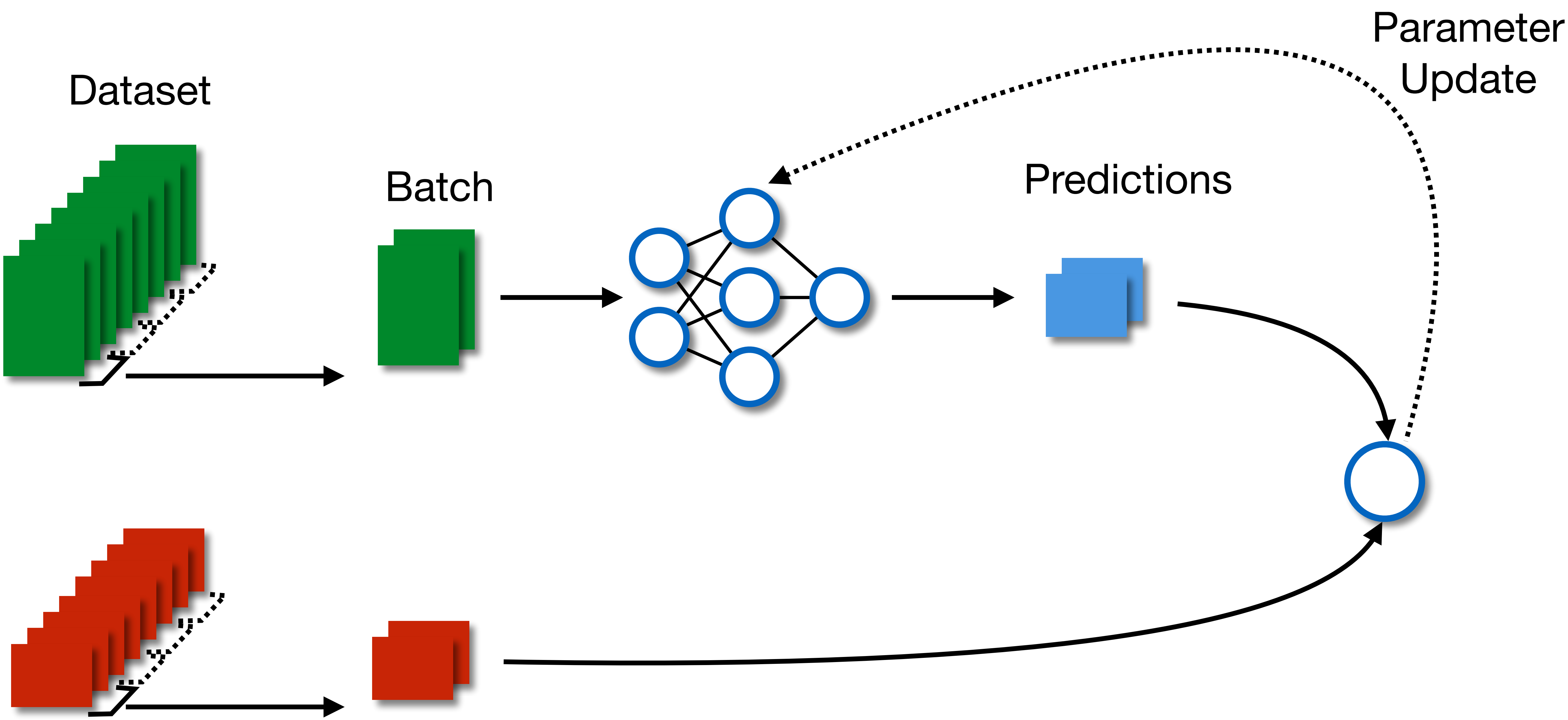
Gradient Descent



Stochastic Gradient Descent



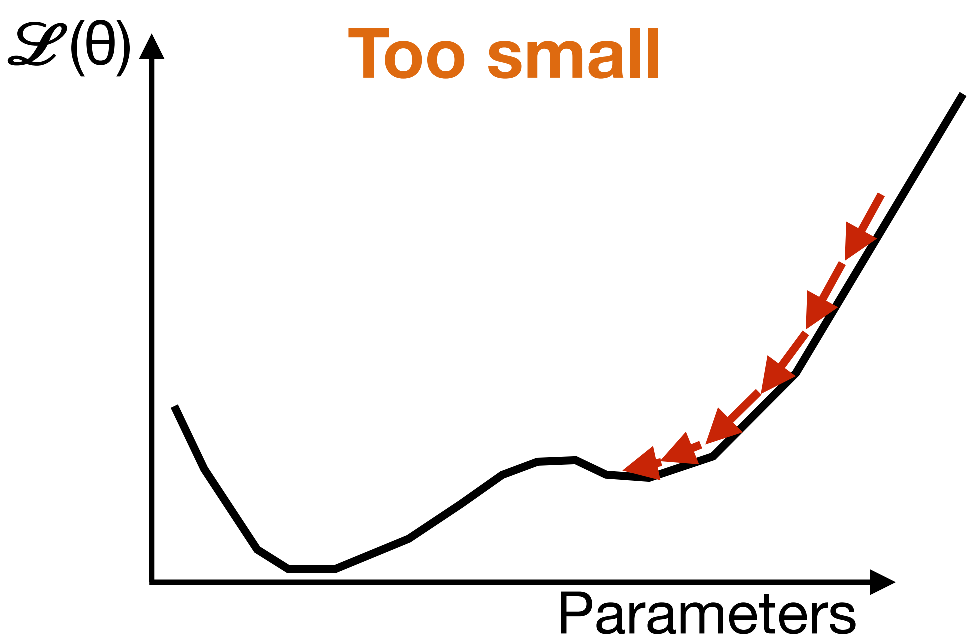
Epoch and Batches

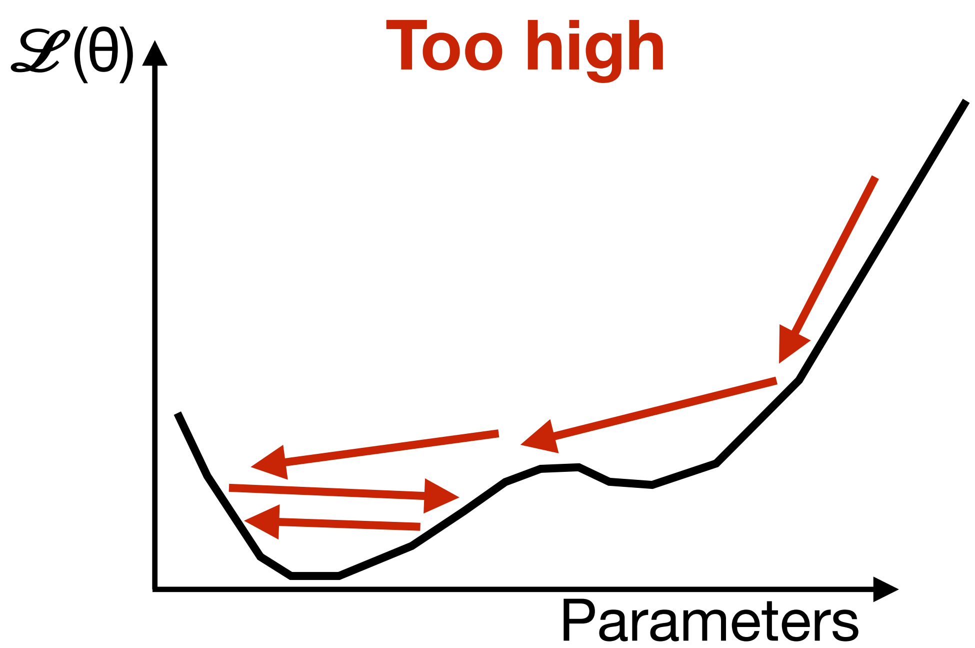


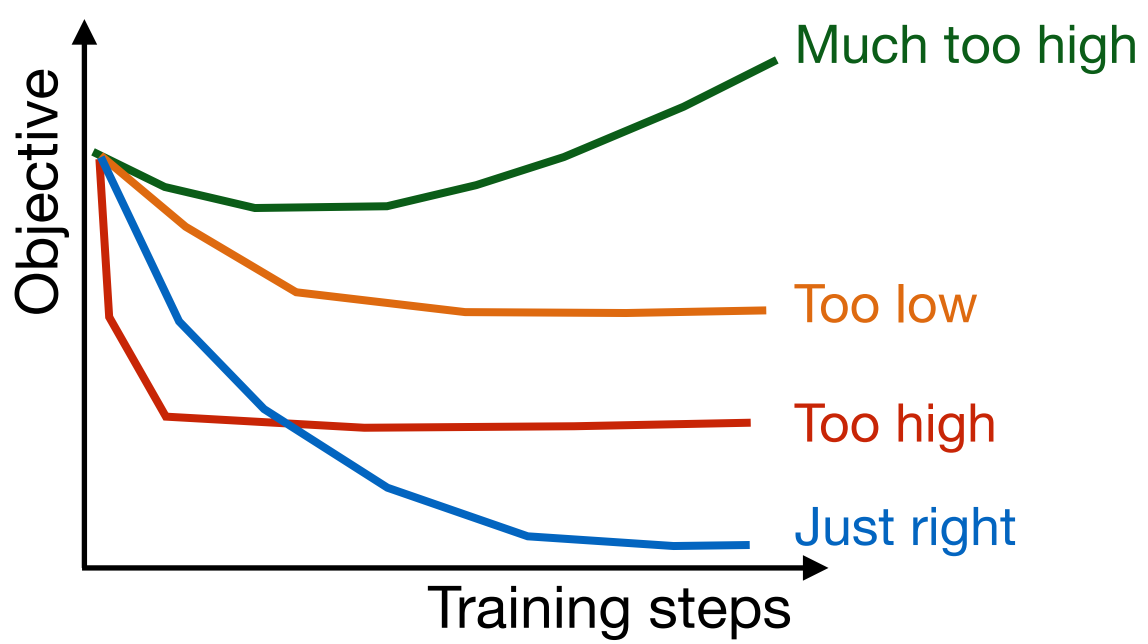
- Training over dataset in batches:
  - Batch = Certain number of samples for which gradients are calculated
  - Epoch = One run through the whole training dataset

Learning Rate (α)

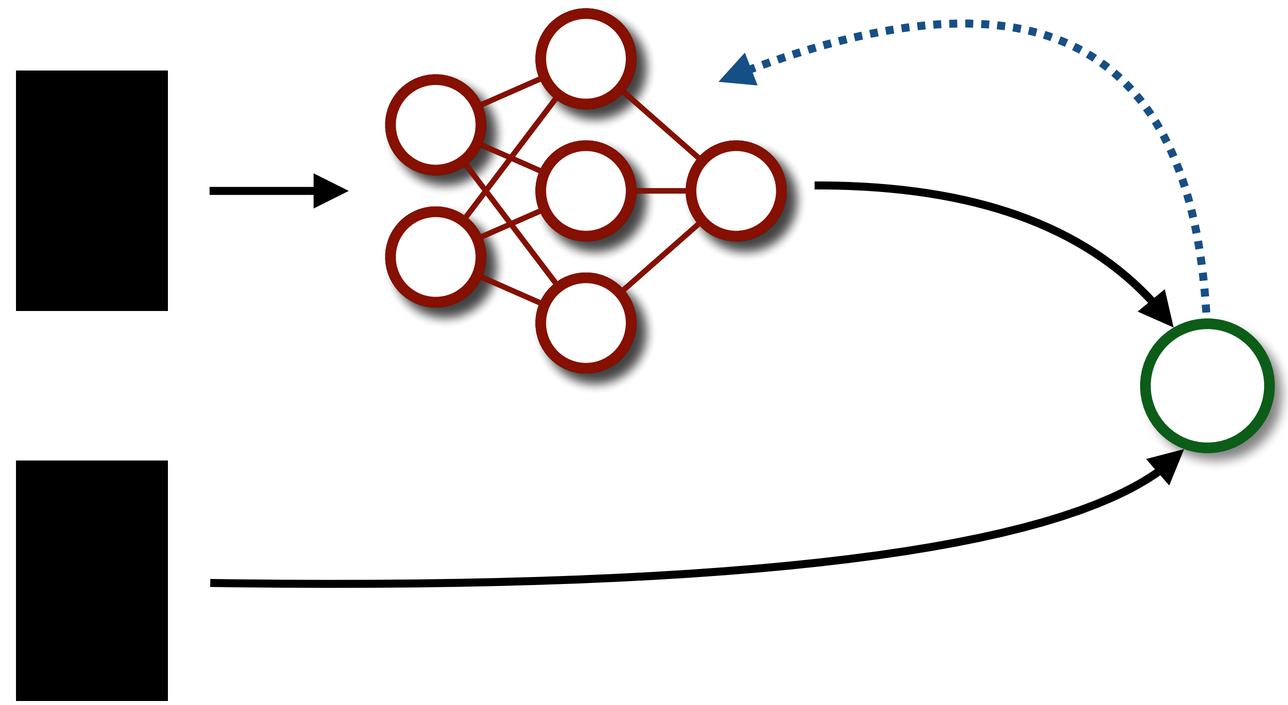
- Training with gradient descent (learning rate α = step size scale) $\theta \rightarrow \theta - \alpha \frac{d\mathscr{L}}{d\theta}$
- Stable learning rates converge smoothly and avoid local minima





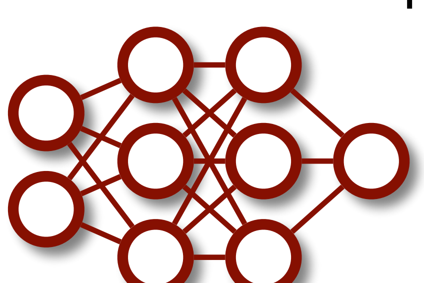


Summary



Model

- Non-linear perceptron
  $y = \sigma(w \cdot x + b)$
  Parameter $\theta = (w, b)$
- Deep networks
- General function approx.



Objective

- Fit between training data and prediction
- Regression (mse):
  $\mathscr{L} = (y_{true} - y_{pred})^2$
- Classification (cross-ent.):
  $\mathscr{L} \propto -\Sigma_i \Sigma_j p_{ij} \log(q_{ij})$

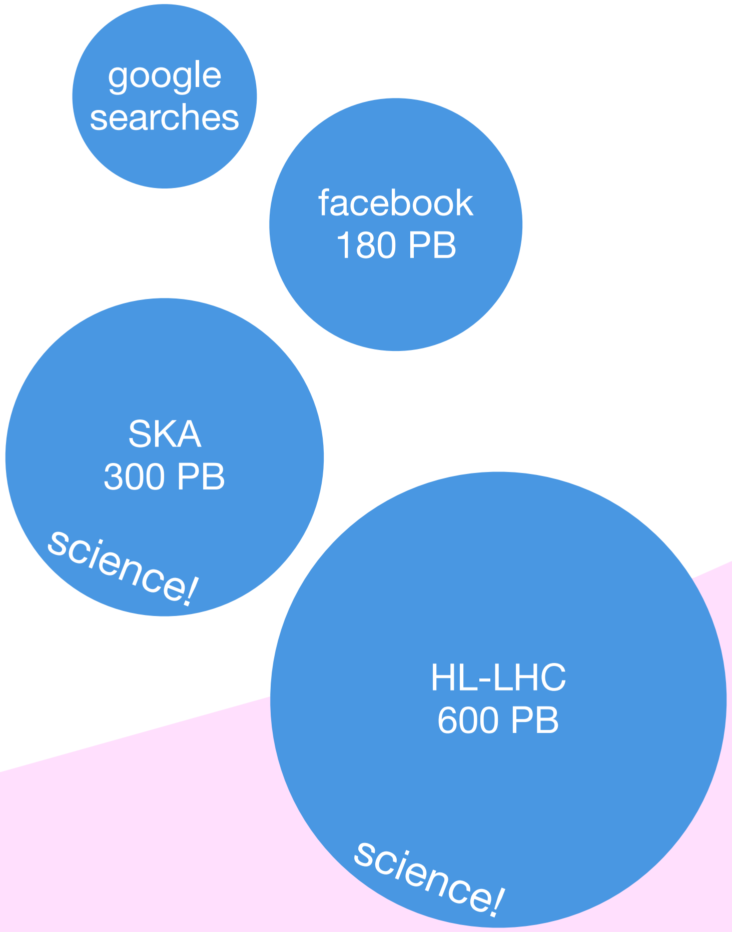
Training

- Find parameters which minimize loss ($\mathscr{L}$)
- Gradient descent
  $$\theta \rightarrow \theta - \alpha \frac{d\mathscr{L}}{d\theta}$$
- Backpropagation
- SGD & Batching

Now is a **Very Good Time!**

Data

- Enough data
- Often very well structured

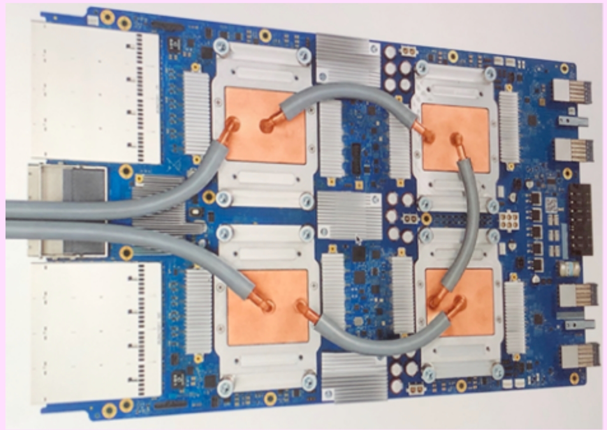


Software

- Very good open source libraries
- Industry-near

Hardware

- Highly parallel
- GPU, TPU, …

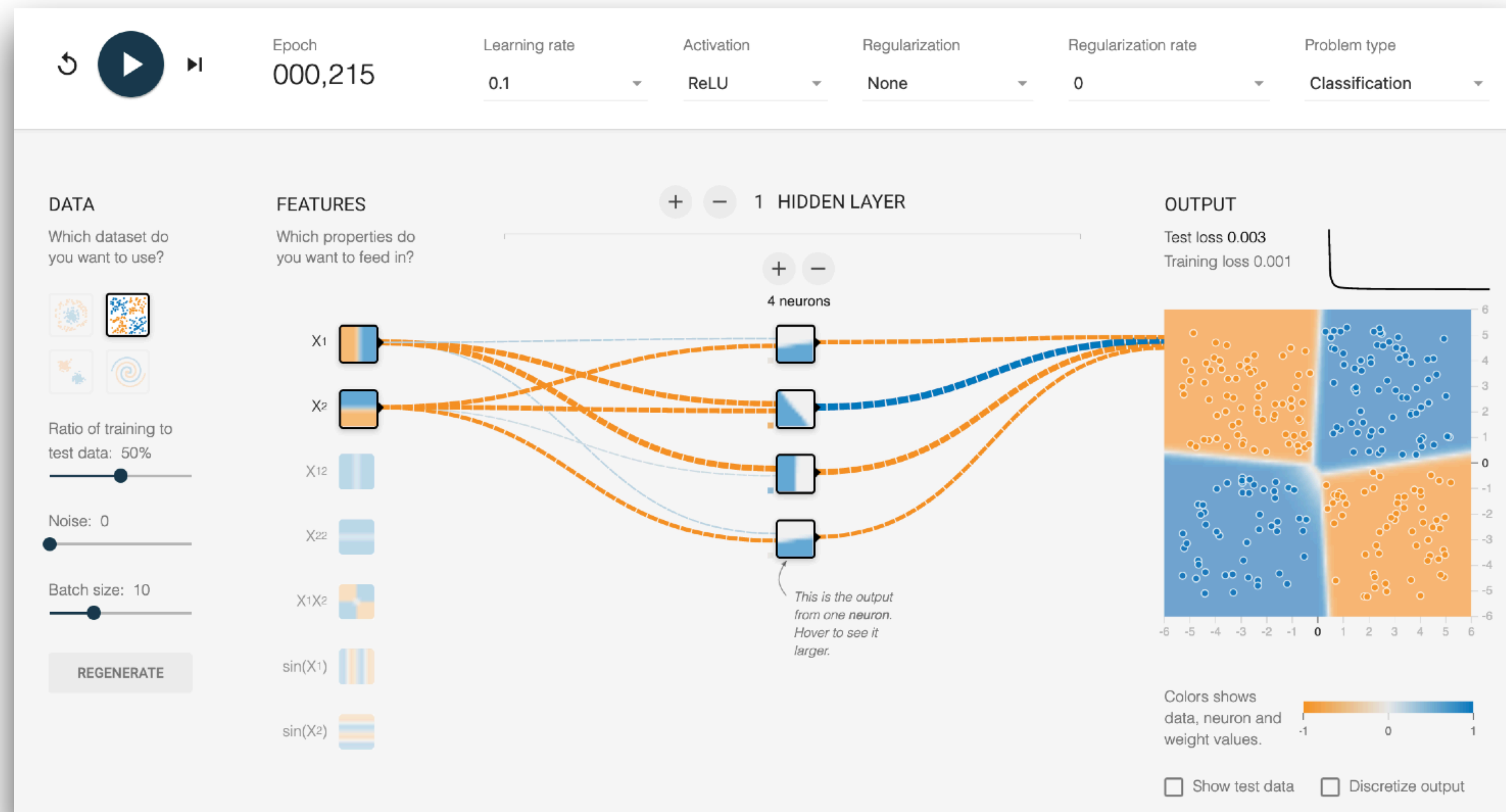


[5,6]

Exercise 2.1 - Non-linear function XOR

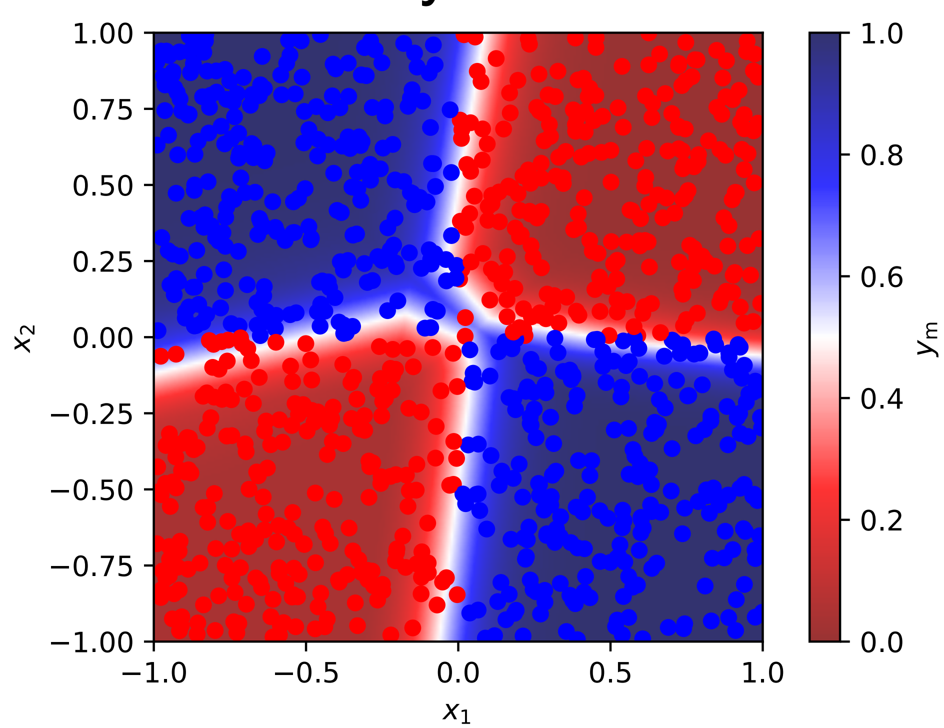
RWTH AACHEN UNIVERSITY
Dennis Noll - 09.08.22

- Try the Checkerboard example at playground.tensorflow.org
  1. Try various settings for the number of layers and neurons using ReLU as activation. What is the smallest network that gives a good fit result? Is the configuration stable?
  2. What do you observe for multiple trainings with the same settings?
  3. Try additional input features. Which one is most helpful?



- **Notes/Solutions**:

- Solve the checkerboard task using this notebook.
  1. Inspect the implemented model. What is the total number of parameters? First do the calculation on paper, then verify your result using model.summary().
  2. Train your model to an accuracy of at least 90%. How many epochs are needed to achieve this?
  3. Plot the raw data and the output of your model. Describe your observations.
  4. Change the network according to the following configurations and retrain the model. Describe your observations.
     - 8 neurons in the hidden layer
     - 2 neurons in the hidden layer
     - Add an additional hidden layer of 4 neurons with a ReLU activation



Exercise 2.2 - Non-linear function XOR

- **Notes/Solutions**:

Citations

1. **Deep Learning in Physics Research Lecture**, Martin Erdmann, Jonas Glombitza, Uwe Klemradt, Dennis Noll, RWTH Aachen University
2. **Initializing neural networks**: Deep Learning AI 2021, Link (accessed 04.08.22)
3. **How to Classify Photos of Dogs and Cats (with 97% accuracy)**: Jason Brownlee, 17.05.2019, Link (accessed 04.08.22)
4. **Visualizing the Loss Landscape of Neural Nets**: Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, Tom Goldstein, Advances in Neural Information Processing Systems 31 (NeurIPS), 2018, Link (accessed 04.08.22)
5. **NVIDIA GeForce GTX 1080 Ti**, NVIDIA, Link (accessed 04.08.22)
6. **Tensor Processing Unit 3.0**: Zinskauf, CC BY-SA 4.0, Link (accessed 04.08.22)

Backup

Excursus: Tensors!

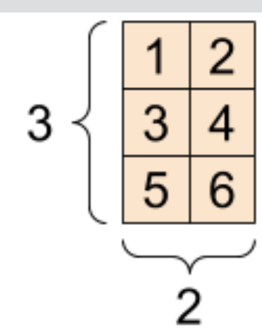
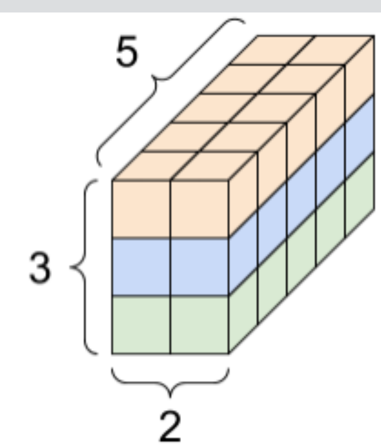
RWTH AACHEN UNIVERSITY
Dennis Noll - 09.08.22

- Immutable objects, usually placeholder for values (e.f. tf.Tensor)
- Defined by: Type (int, float, …) Rank & Shape
- First dimension usually “batch“

Examples

	Scalar	1D Array	2D Array	3D Array	…
Rank	0	1	2	3	

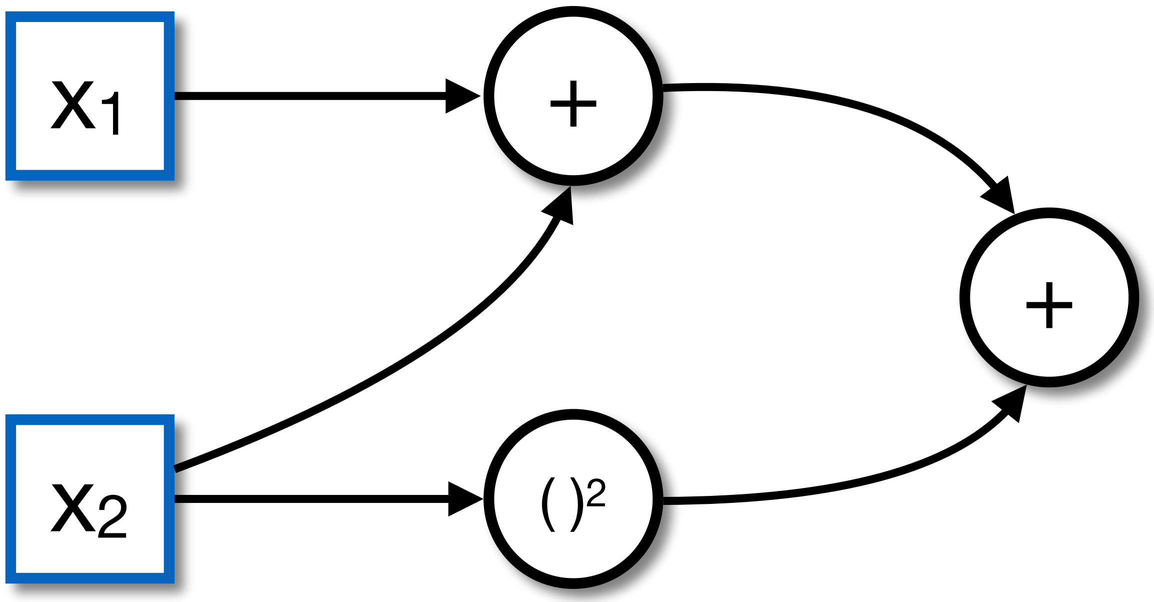
Code (Tensorflow)	`tf.constant(4)`	`tf.constant(`
		`[1, 2, 3],`
		`)`
			`tf.constant(`
			`[`
			`[1, 2],`
			`[3, 4],`
			`[5, 6],`
			`]`
			`)`
				`tf.constant(`
				`[`
				`[`
				`[…],`
				`…,`
				`]`
				`)`	

Shape	[ ]	[3]	[3, 2]	[3, 2, 5]	
Value	4	[1, 2, 3]			

Excursus: Graphs!

- Graph = static computing model consisting of
  - Tensors (value placeholders)
  - Structural elements which connect tensors (e.g. tf.Operation)
- Defined by: Inputs, Outputs, Operations and connections

$$f(x_1, x_2) = x_1 + x_2 + x_2^2$$



- Graphs can be **optimized** (parallel execution): Super fast!
- Graphs are **portable**: Run on CPU, GPU, TPU, Multiple devices in parallel
- Graphs are **static**: Everybody gets the same results, everywhere

Created by tf.function - see Marcels lecture