UNIVERSIDADE CABO VERDE

Introduction to Deep Learning

Fabian Müller, PhD

Ziemer Ophthalmic Systems in Bienne, Switzerland

June 11th 2019, University of Cape Verde

What is Machine Learning?

- **Given:** A decision task humans can do without knowing the exact decision mechanics.
- Examples:
  1. From a time series of measurements, identify outliers.
  2. Classifying Emails for Spam/No Spam
  3. Identifying different parts of an eye from an image.
  4. Generating images from pictures in a certain style.
- **Problem:** The decision ”algorithm” our brain follows is far from understood, and therefore impossible to program.
- **Solution:** Use an approach **based on given data**. Find solutions to your task by minimizing an *error function* over the data.
- Main paradigms: **Supervised** vs **unsupervised** learning.

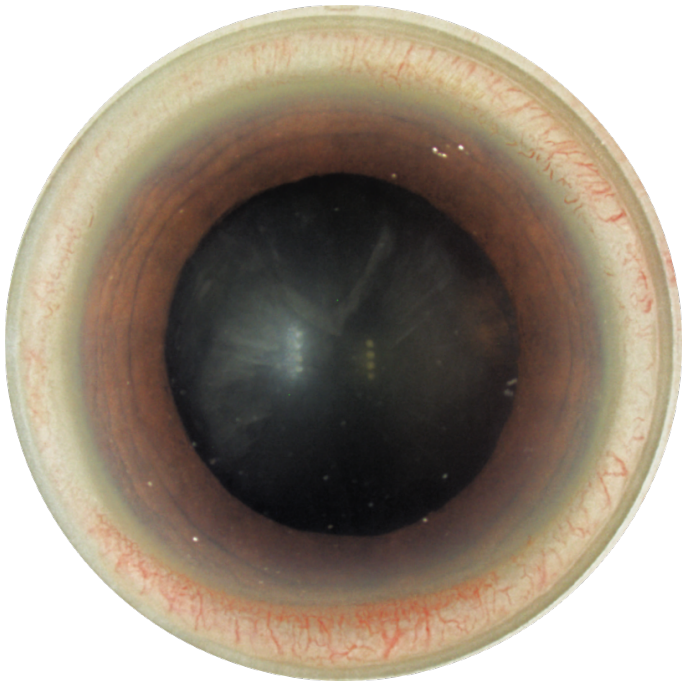
Example: Image Classification

- *Given:* An image taken with 4MP resolution: $3 \cdot 4 \cdot 10^6$ numbers.
  Goal: A number $p \in [0, 1]$ corresponding to the likelihood that there is a car in the picture.
- Deep neural networks in a nutshell:
  1. Compute linear combinations of the input numbers.
  2. Apply non-linear functions to the results.
  3. Repeat steps 1-2 with the output of step 2.
- Coefficients in the linear combinations of step 1:
  1. Each choice of coefficients yields an output $p(I)$ for each image I.
  2. Prepare a **dataset** of N_{car} images with cars ($p_{\text{exact}}(I) = 1$) and $N_{\text{not car}}$ images without cars ($p_{\text{exact}}(I) = 0$).
  3. Try to find a choice of coefficients which minimizes $p - p_{\text{exact}}$ on the given data set (in some sense).

$\Rightarrow$ Crucial to find good datasets and minimization procedures!

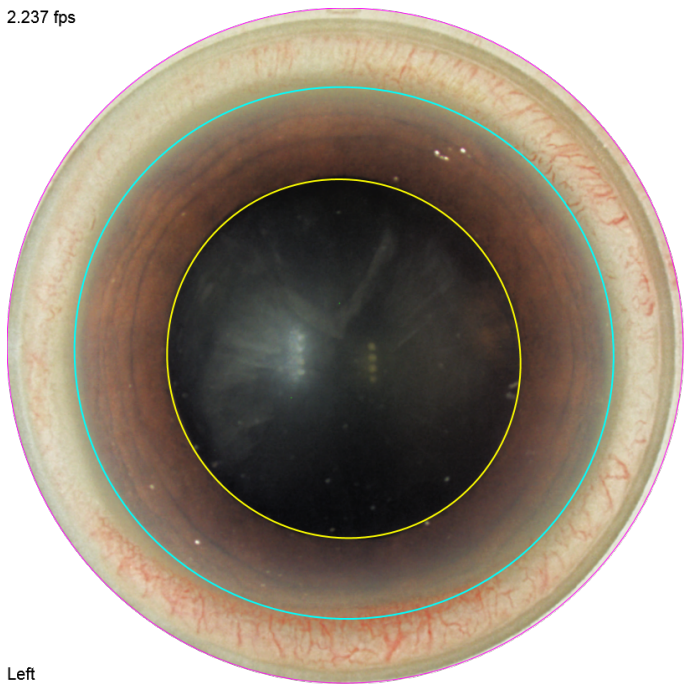
Another example

- **Example:** From the image of an eye, locate the pupil as ellipse.
- Ellipses: Determined by centre coordinates, two major axes, tilt angle: **Predict 5 numbers.**
- **Ground truth:** We are given a **labeled dataset** of 1000 images *with their "real" ellipses* entered by humans.
- Getting a dataset can be difficult: Noise, balance, representation.
- Deep learning vs. classical methods.



Another example

- **Example:** From the image of an eye, locate the pupil as ellipse.
- Ellipses: Determined by centre coordinates, two major axes, tilt angle: **Predict 5 numbers.**
- **Ground truth:** We are given a **labeled dataset** of 1000 images *with their "real" ellipses* entered by humans.
- Getting a dataset can be difficult: Noise, balance, representation.
- Deep learning vs. classical methods.



Goals today I

1. Understand the architecture of a Deep Neural Network.
2. Be able to diagnose most common issues in Neural Networks and to avoid them.
3. Know the building blocks of Convolutional Neural Networks.
4. How to use Neural Networks to measure properties hard to quantify.

Goals today II

5. Understand the following cartoon:

Table of Contents

General implementation notes

- **Python** has been established as the main programming language for machine learning tasks.
- Libraries are provided for the implementation of well-known algorithms: **SciKit Learn, Tensorflow, Keras, PyTorch** for Python, **Caffe2** for C++.
- Python is used for a simple implementation, configuration and training, the backend uses compiled C++ code.
- Training is computationally expensive. Speed-up through heavily parallelized implementations, using GPUs (CUDA), and using low-rank tensor approximation.
- The material shown in this course is provided on `https://github.com/fablukm/`

Linear Regression: Prediction I

- Consider a quantity $y \in \mathbb{R}^m$ which depends on $x \in \mathbb{R}^n$.
- We **assume** that y depends on x *linearly or affinely*:

$$y = Wx + b\,, \qquad \text{for some } W \in \mathbb{R}^{m\times n} \text{ and } b \in \mathbb{R}^m\,.$$

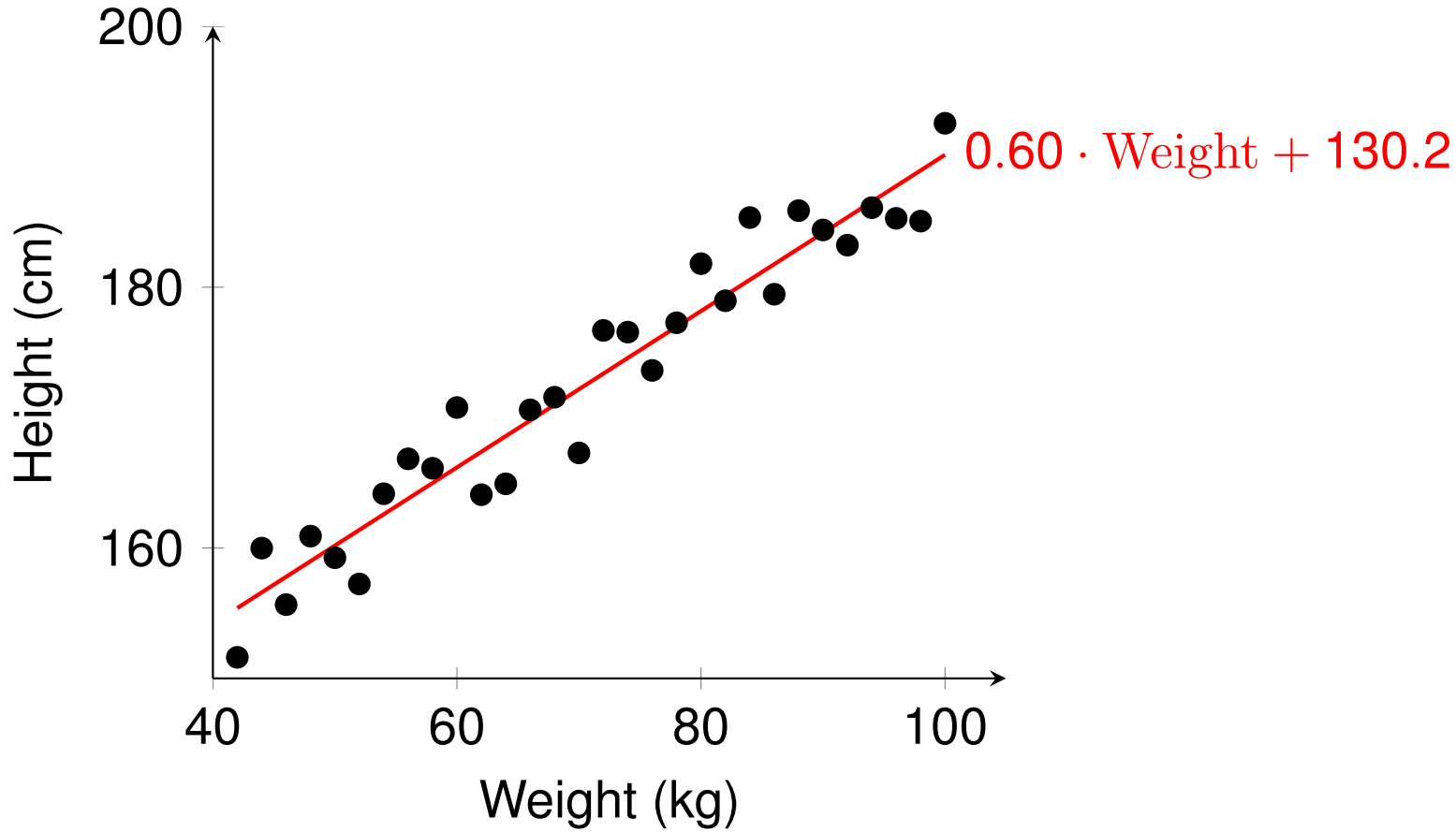
- However, we do not know the exact *parameters* W and b.
- Need to *learn* $(n+1)m$ parameters *from data*.
- Assume we are given data points $\{(x^1, y^1), (x^2, y^2), \ldots, (x^N, y^N)\}$. The data is inaccurate, subject to noise.
- **Classical Linear Regression:** Find W and b subject to an optimization task, for example

$$\min_{W\in\mathbb{R}^{m\times n}, b\in\mathbb{R}^m} \sum_{i=1}^{N} \left\| y^i - (Wx^i + b) \right\|_2^2$$

- Two norms involved here: l^1 over the data, l^2 over the components.

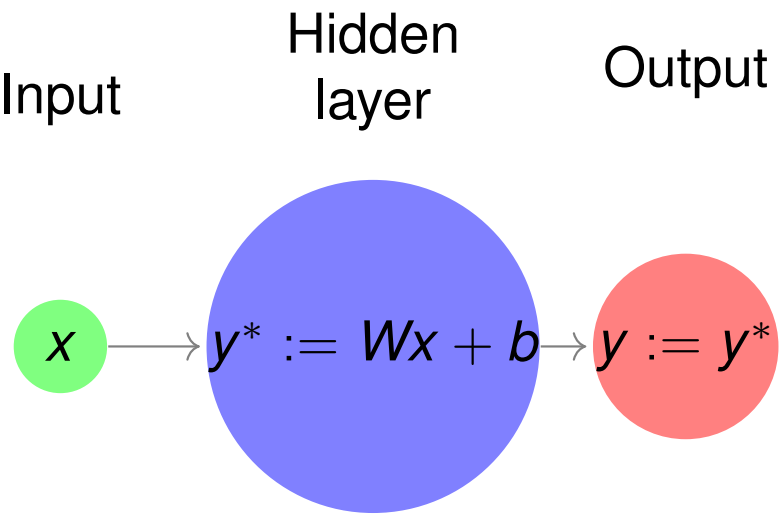
Linear Regression: Prediction II

- **Simple constructed example:**



Linear Regression: Prediction III

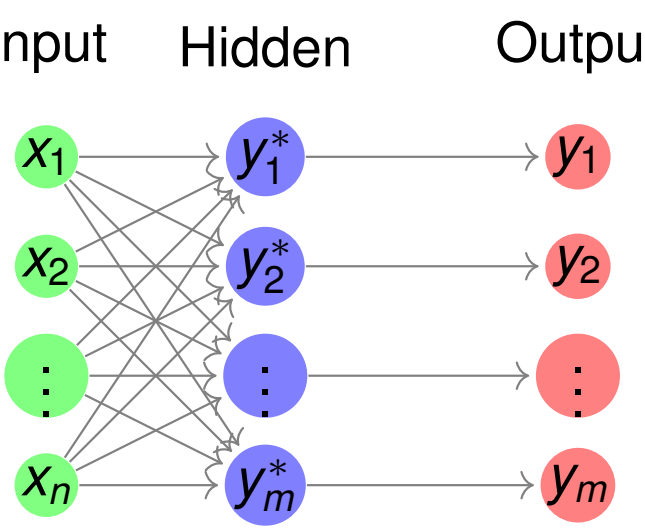
- We reformulate this problem in a neural network language.
- There are two steps: **Prediction** and **Learning**.
- **Prediciton:** Given a *new* input value $x \notin \{x^1, \dots, x^N\}$.
  If the parameters W and b are known, *predict* the value $y(x)$ to be $y^* := Wx + b$. A **prediction** follows the computation graph



- For further purposes, we put the linear combination into a *hidden layer* and define the output to be the output of the hidden layer.

Linear Regression: Prediction IV

- The same graph, but component-by-component:



where

$$y_k^* := \sum_{j=1}^{n} w_{kj} x_j + b_k \,, \qquad k = 1, \ldots, m \,,$$

$$y_k := y_k^* \,, \qquad k = 1, \ldots, m \,,$$

w_{kj} and b are *weights which are determined iteratively to fit the data.*

Linear Regression: Training

- **Loss function:** To *learn* the weights w_{kj}, we minimize the **mean squared error**. For one sample (x^i, y^i), it is defined as

$$\varepsilon_{mse}(x^i, y^i) := \frac{1}{m} \sum_{k=1}^{m} \left(y_k^i - \left(W x^i + b \right)_k \right)^2 .$$

- **Gradient descent:** Find weights W, b such that ε_{mse} is minimal. Standard iteration:

$$w_{kj;\text{new}} := w_{kj;\text{current}} - \alpha \frac{\partial \varepsilon_{mse}}{\partial w_{kj}} ,$$

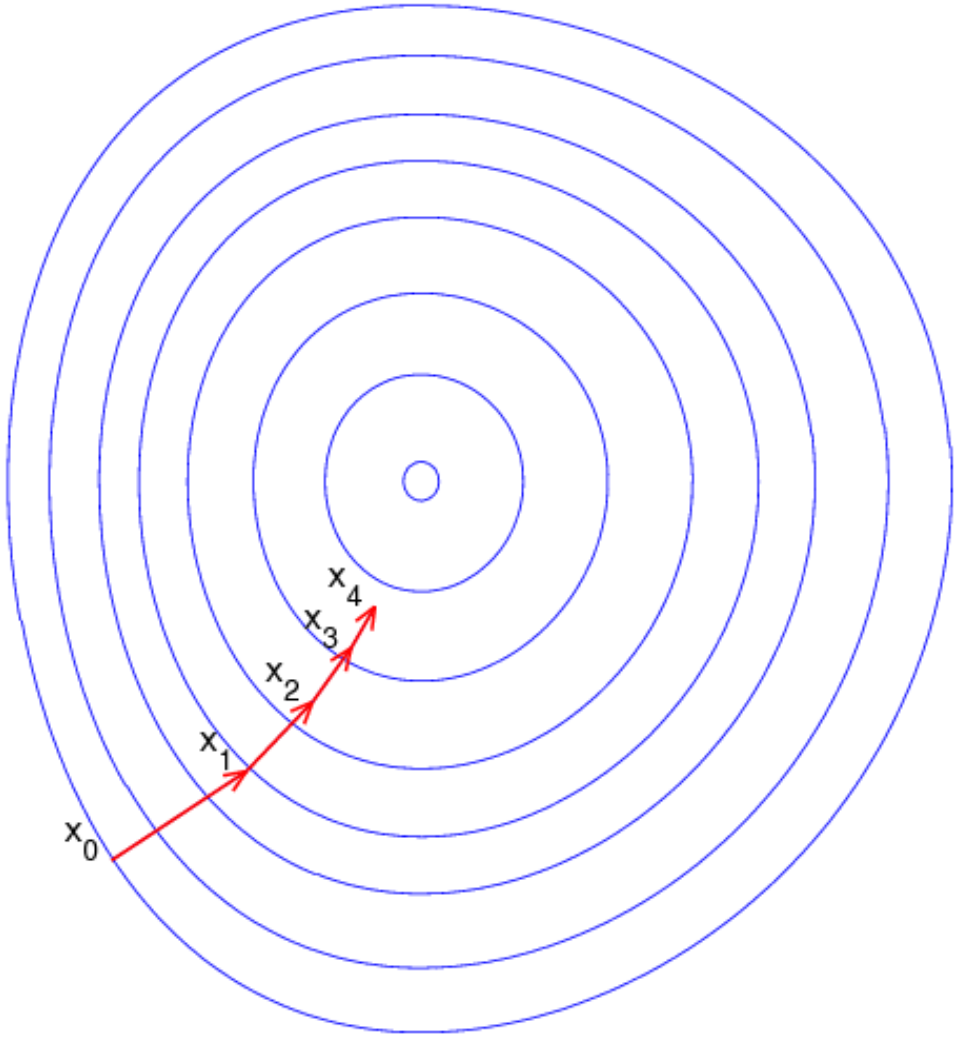
$$b_{k;\text{new}} := b_{k;\text{current}} - \alpha \frac{\partial \varepsilon_{mse}}{\partial b_k} ,$$

  with *randomly initialized weights* $w_{kj;\text{start}}$ and $b_{j;\text{start}}$, with *learning rate* $\alpha > 0$.

- Linear regression with loss ε_{mse}: Derivatives easy to compute.

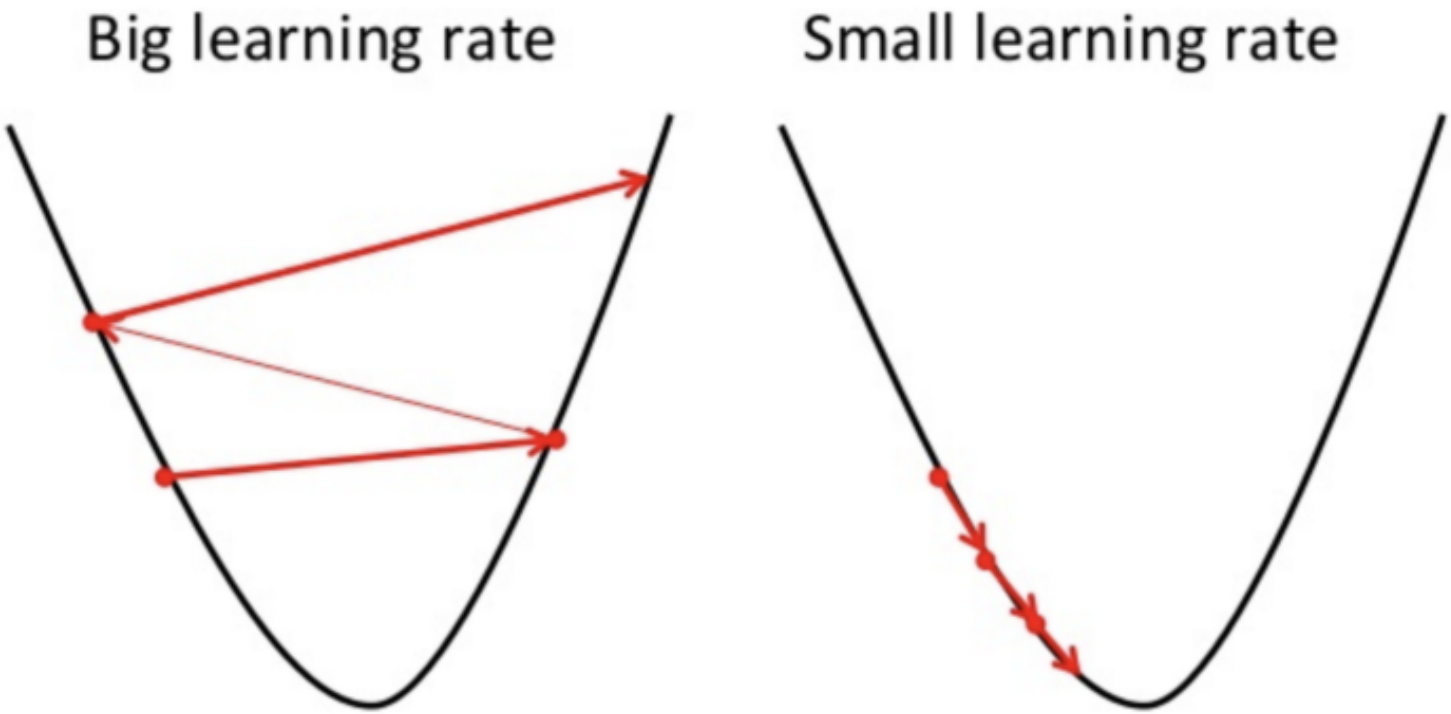
Intuition for Gradient Descent

Intuition for Gradient Descent

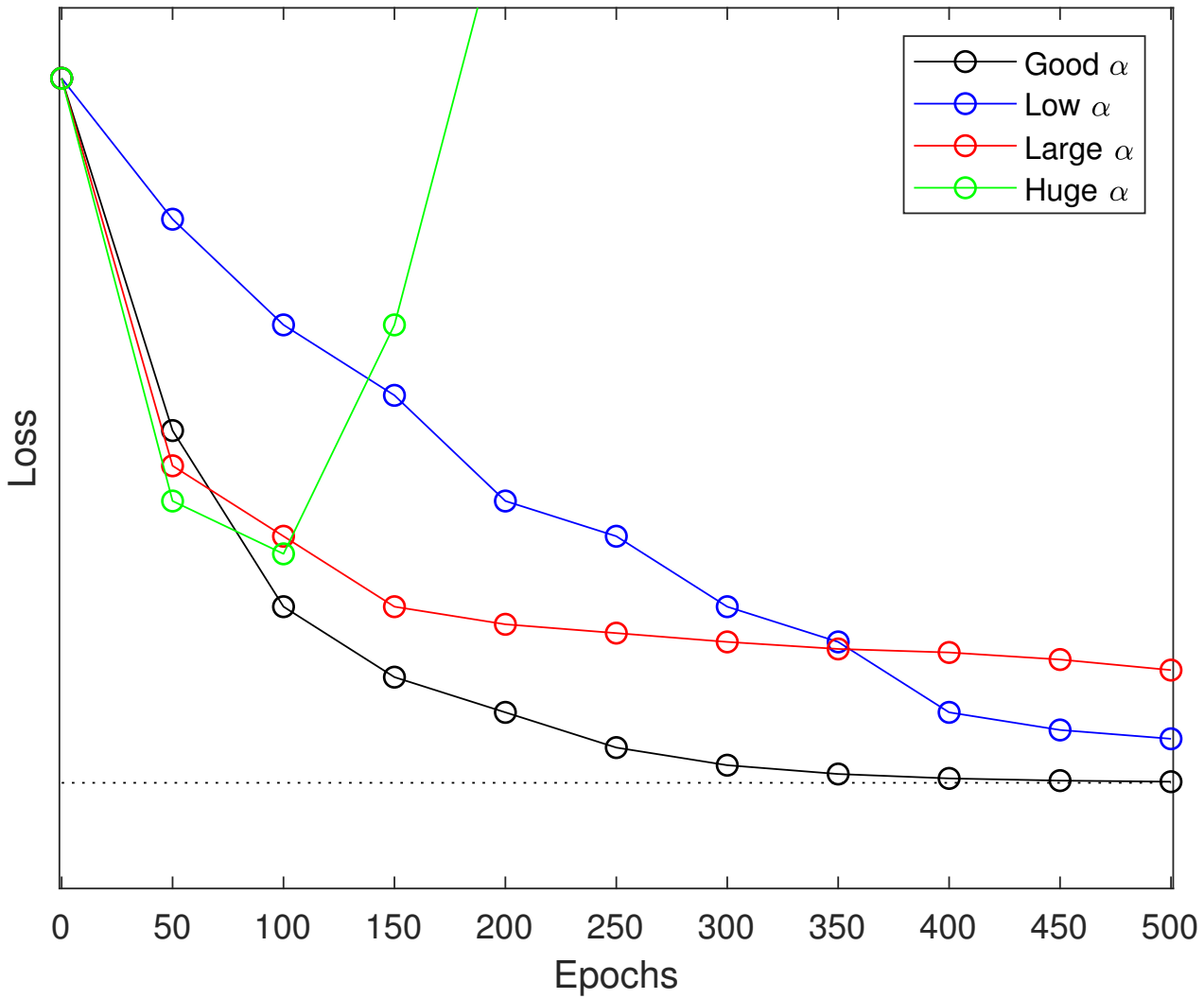


Intuition for Gradient Descent

Intuition for Gradient Descent



Intuition for Gradient Descent

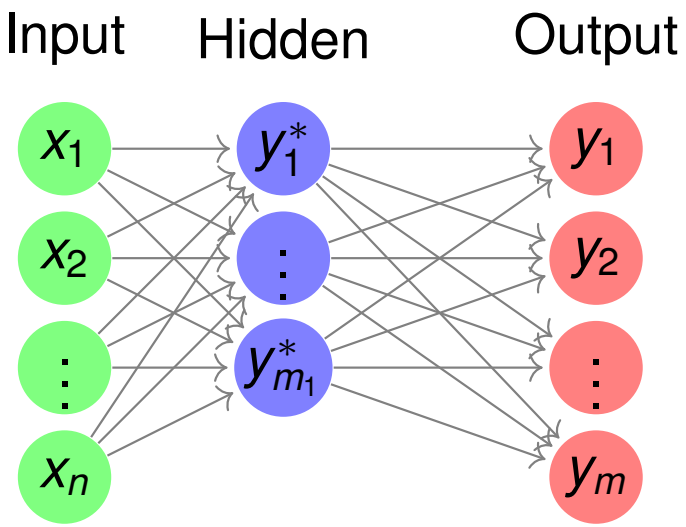


Remarks about Gradient Descent

1. The performance of the algorithm depends on the choice of a learning rate $\alpha > 0$.
   - Large learning rate: Fast descent at first, may not converge.
   - Small learning rate: Slow convergence overall.
2. No guarantee to find global minimum for a arbitrary loss functions.
3. **In practise:** Better minimization schemes, derived from gradient descent: ADAM, Stochastic GD.
4. **In practise:** A tolerance and a maximal number of iterations should be chosen as well.
5. **Definition:** (Only) here, we call one iteration one *epoch*.

$\Rightarrow$ **Parameters** are learned by minimization of the loss function, but **hyperparameters** are determining the algorithm, and are chosen at the beginning!

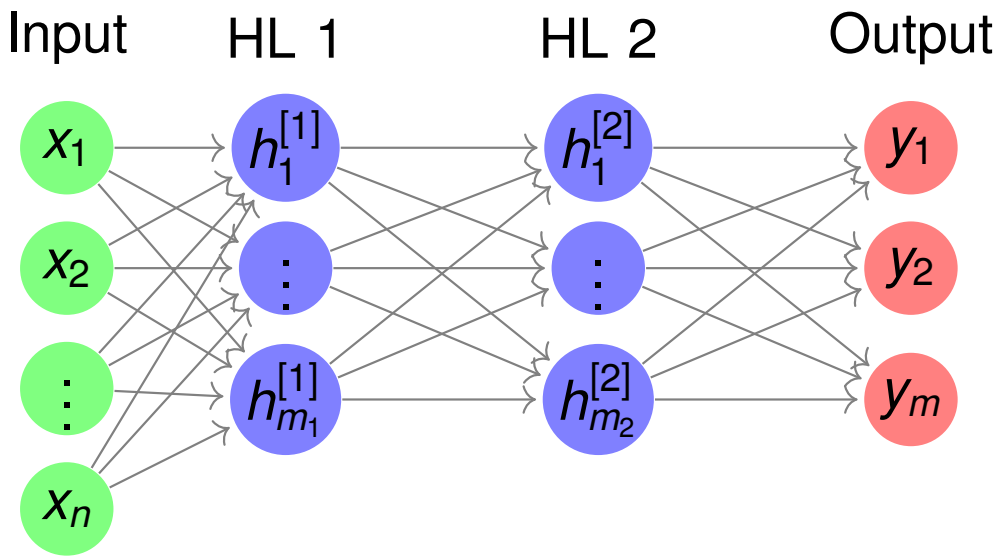
Towards Neural Networks 1: Hidden Layers



- Change the size of the hidden layer to $m_1 \neq m$.
- Compute y_k by another linear combination of the y_l^*.
- More weights to learn: $W^{[\text{in,hidden}]} \in \mathbb{R}^{m_1 \times n}$ and $W^{[\text{hidden,out}]} \in \mathbb{R}^{m \times m}$.
- Gradient descent needs derivatives w.r. to both weights.
- ? **What will be different now?**

Towards Neural Networks 2: Hidden Layers

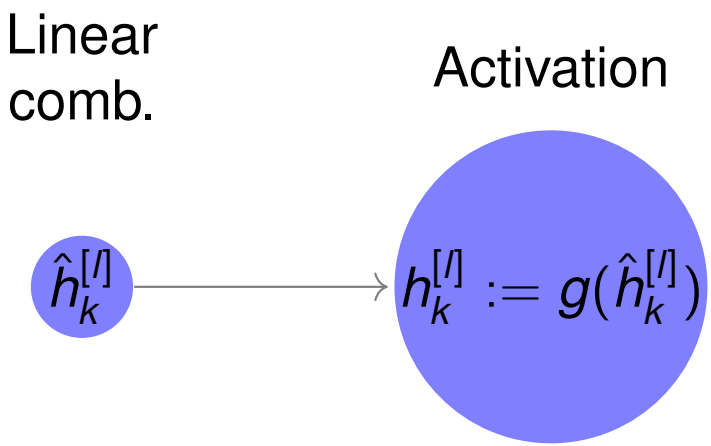
- Linear of linear is linear: **Nothing changed.**
- Add one more hidden layer with m_2 nodes.
- Hidden layer 1 *gets propagated* by linear combination to Hidden layer 2.
- Introduces weights $W^{[1,2]} \in \mathbb{R}^{m_2 \times m_1}$.
- ? **What will be different now?**



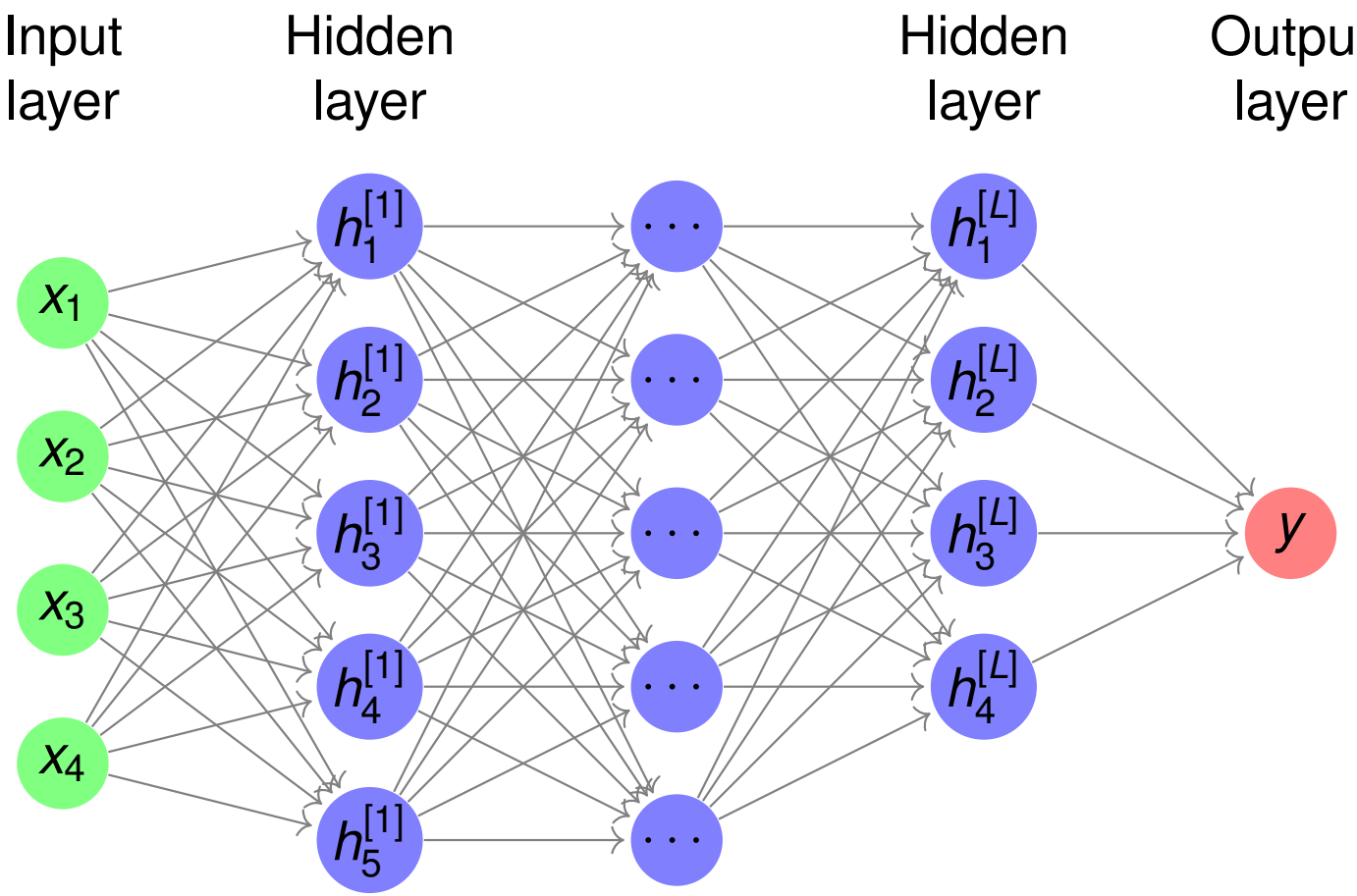
Towards Neural Networks 3: Activation functions

- **Still nothing has changed.**
- **Idea:** Apply a non-linear **Activation Function** g to the output of a hidden layer.
- Several activation functions have been proposed for different purposes.
- Replace a hidden node $h_k^{[l]}$ (the k-th node in the l-th layer) by

$$h_k^{[l]} := g(h_k^{[l]})$$



Multilayer Perceptron: Forward I



Multilayer Perceptron: Forward II

Definition (Multilayer Perceptron (MLP))

A multilayer perceptron (*neural network*) architecture is defined by

1. the input size m_0
2. the number of hidden layers L
3. the size of each hidden layers $\{m_l\}_{l=1,\dots,L}$
4. the activation function applied to hidden layer l $\{g^{[l]}\}_{l=1,\dots,L}$
5. the output size m_{L+1}

To run an input through the MLP, one needs for each layer

$$\text{weights } W^{[l,l+1]} \in \mathbb{R}^{m_{l+1} \times m_l}, \qquad l \in \{0, \dots, L\},$$
$$\text{biases } b^{[l,l+1]} \in \mathbb{R}^{m_{l+1}}, \qquad l \in \{0, \dots, L\}.$$

Multilayer Perceptron: Forward III

- Forward step (if the weights are known):
  1. We start with an input $x = (x_1, \ldots, x_{m_0}) \in \mathbb{R}^{m_0}$.
  2. Compute a linear combination of the input to obtain the nodes in the first hidden layer, and apply an activation function $g^{[1]}$:

$$h^{[1]} := g^{[1]}\left(W^{[0,1]}x + b^{[0,1]}\right).$$

  3. For $l = 1, \ldots, L-1$, use layer l as input for layer $l+1$:

$$h^{[l+1]} := g^{[l]}\left(W^{[l,l+1]}h^{[l]} + b^{[l+1]}\right).$$

  4. Compute the output:

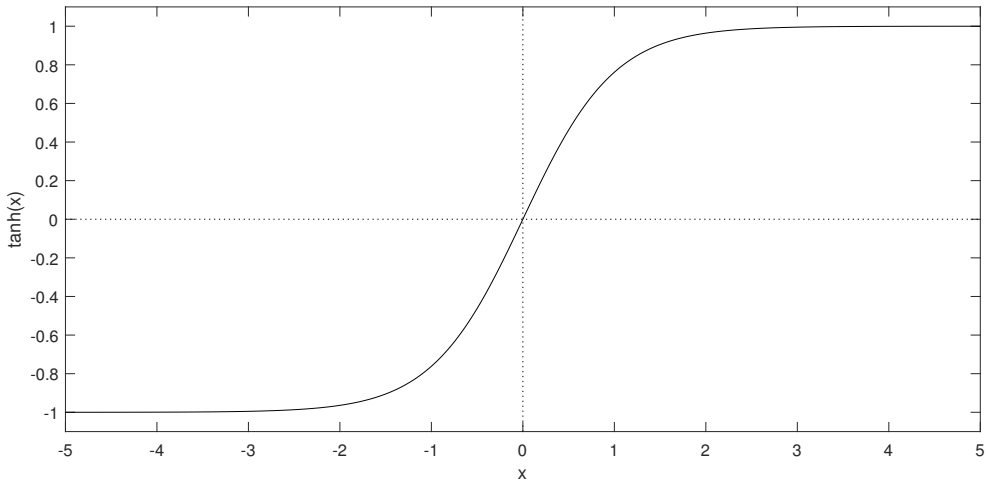
$$y := g^{[L]}\left(W^{[L,L+1]}h^{[L]} + b^{[L+1]}\right).$$

- A MLP with input x is a composition of maps.
  Notation: $y = \Phi(x)$.

Activation Functions I

The tanh activation function

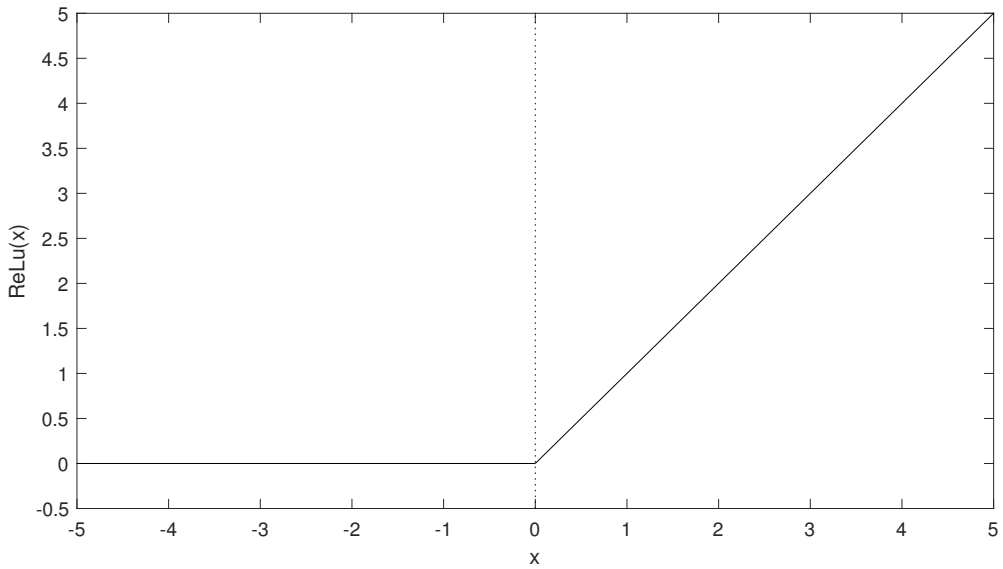
- Consider $g(\eta) := \tanh(\eta) := \frac{e^{\eta} - e^{-\eta}}{e^{\eta} + e^{-\eta}}$.
- Derivative: $g'(\eta) = 1 - (g(\eta))^2$
- $\lim_{\eta \to -\infty} g(\eta) = -1$, $\lim_{\eta \to \infty} g(\eta) = 1$
- Used for *Binary Classification*: “Is it a car or a house?”



Activation Functions II

The ReLU activation function

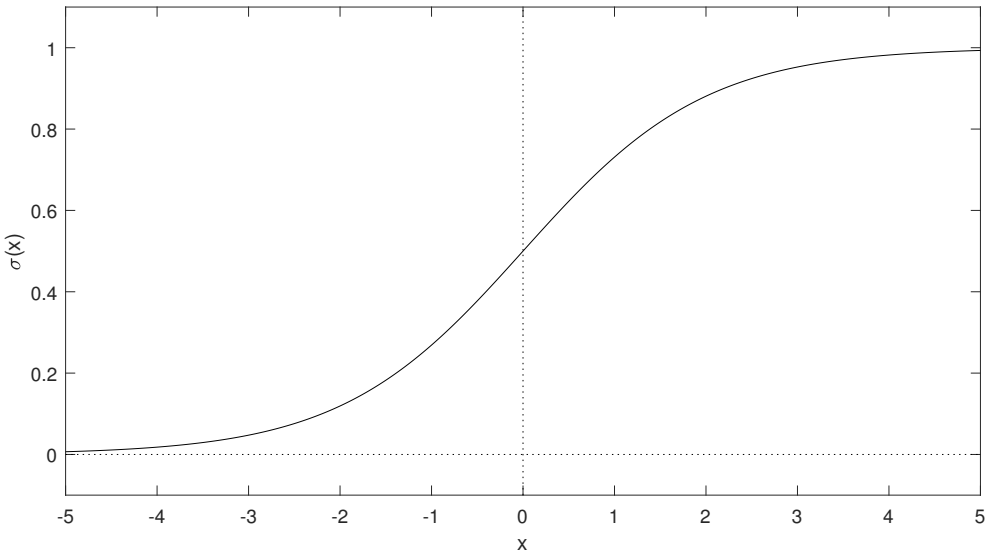
- Consider $g(\eta) := \text{ReLU}(\eta) := \max\{0, \eta\}$.
- Derivative piecewise: $g'(\eta) = \mathbb{I}_{(0,\infty]}(\eta)$ for $\eta \neq 0$.
- Drops negative values: “Activates only positive neurons”.



Activation Functions III

The logistic activation function σ

- Consider $g(\eta) := \sigma(\eta) := \frac{1}{1+e^{-\eta}}$.
- Derivative piecewise: $g'(\eta) = g(\eta)(1 - g(\eta))$.
- Binary $0 - 1$ classification.



Activation Functions IV

The Softmax activation function

- Consider $g : \mathbb{R}^m \to \mathbb{R}^m$ written as $(g_1, \ldots, g_m)$. Define

$$g_k(\eta_1, \ldots, \eta_m) := \frac{e^{\eta_k}}{\sum_j e^{\eta_j}} , \quad k = 1, \ldots, m ,$$

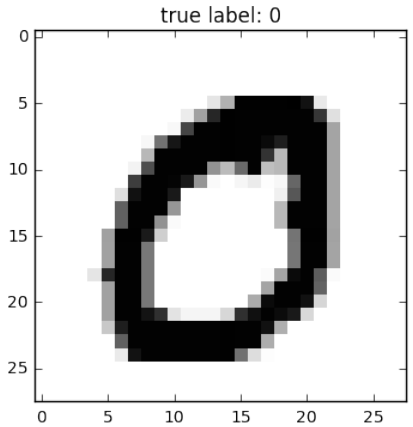
- Examples:

$$g(0, 1, 0) \simeq (0.2, 0.6, 0.2)$$
$$g(0, 10, 0) \simeq (0, 1, 0)$$
$$g(0, -10, 0) \simeq (0.5, 0, 0.5)$$

- "Scales to $[0, 1]$ and discriminates small values."
- $\sum_k g_k = 1$: Can be interpreted as a discrete probability over η.
- "Is it a dog, a cat, a bird, a house, or a car?"
- Derivative piecewise: $\frac{\partial g_k}{\partial \eta_j} = g_k(\delta_{kj} - g_j)$.

Lab: MLP for digit recognition 1

- **Classification Task:** Given an image of a handwritten digit, detect the digit on the image.
- **Input:** Greyscale image 28×28: $m_0 = 28^2 = 784$.
- **Output:** $\mathbf{p} = (p_0, \ldots, p_9) \in [0,1]^{10}$. p_k: probability that k on the image.
- Choice of last activation function $g^{[L]}$?
- **Architecture:** We choose a MLP with $L = 2, m_1 = m_2 = 512$. Clearly, $m_3 = 10$.
- **Activation function:** $g^{[1]} = \text{ReLU}$.



Loss function for classification

- **Softmax:** Output is a discrete probability distribution.
- **Dataset:** $\mathcal{S} = \{(x^i, y^i)\}_{i=1,\dots,N}$ with correct labels $y^i_k \in \{0, 1\}$
- True outputs are either 0 or 1 for each digit.
- **Cross-entropy loss for one sample:** (set $0 \log(0) := 0$)

$$\mathcal{J}(y_{\text{pred}}, y_{\text{true}}) = -\sum_{k=0}^{9} \log(y_{\text{pred}}) y_{\text{true}}$$

$$\overset{\text{binary}}{=} -\sum_{k=0}^{9} y_{\text{true}} \log(y_{\text{pred}}) + (1 - y_{\text{true}}) \log(1 - y_{\text{pred}}) \, .$$

- If true label is 1 and predicted tends to 0: $\mathcal{J}$ explodes.
- Avoid numerical instabilities in implementation.
- Loss for entire dataset:

$$\mathcal{J}_{\text{ce}}(\mathbf{\Phi}; \mathcal{S}) := -\frac{1}{N} \sum_{i=1}^{N} \mathcal{J}(\mathbf{\Phi}(x^i), y^i) \, .$$

Multilayer Perceptron: Learning I

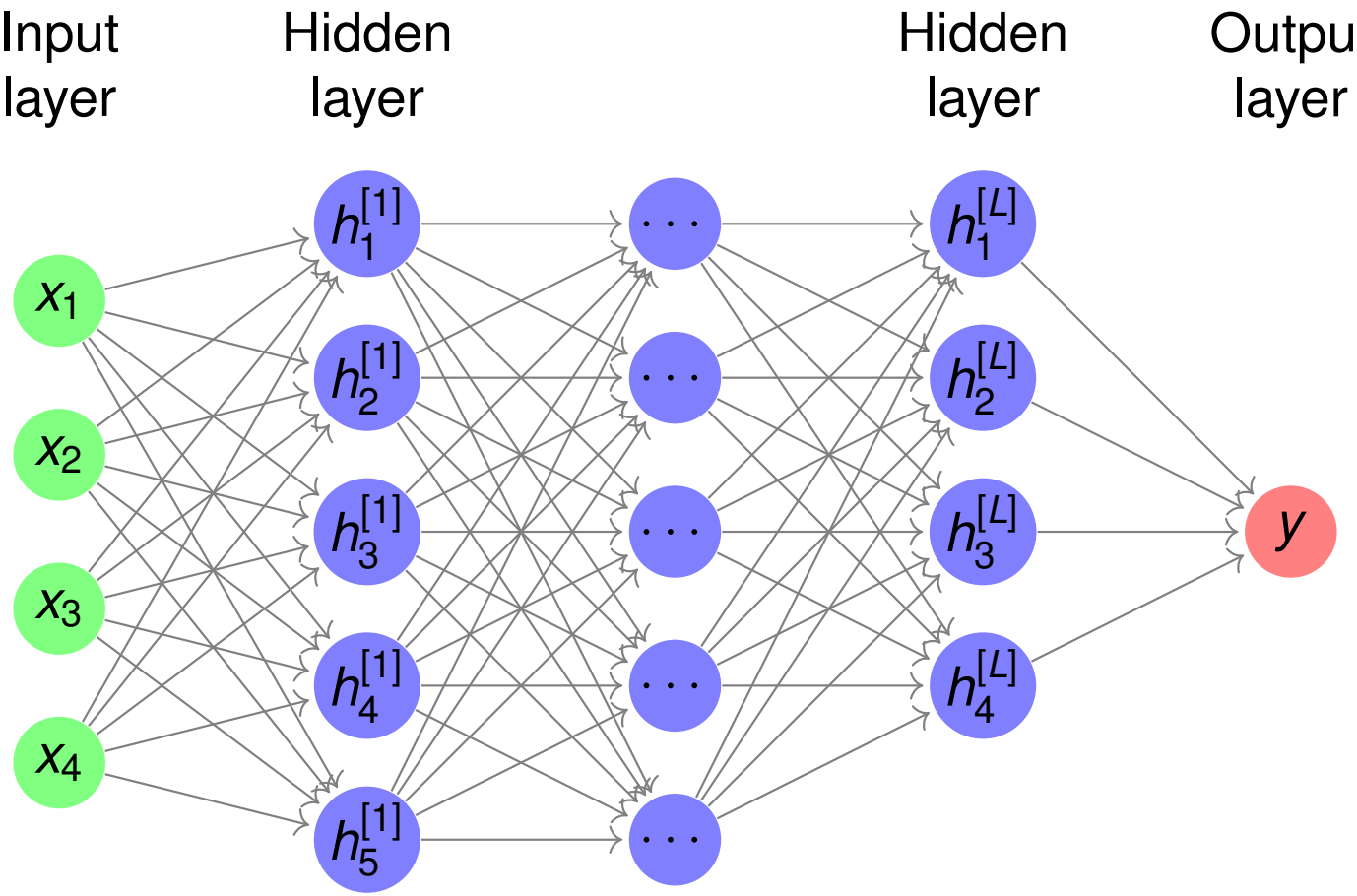
- **Ground truth:** Given a labeled dataset $\mathcal{S} = \{(x^i, y^i)\}_{i=1,\dots,N}$.
- We call x^i *features* and y^i *labels*.
- For theoretical purposes: All parameters in one vector $\boldsymbol{\xi}$, loss dependent on $\boldsymbol{\xi}$.
- Need to minimize a **loss function** ("error") $\mathcal{J}(\boldsymbol{\xi}; \mathcal{S})$:

$$\boldsymbol{\xi}^* = \arg\min_{\boldsymbol{\xi}} \mathcal{J}(\boldsymbol{\xi}; \mathcal{S}) \ .$$

- **Gradient descent** with learning rate $\alpha > 0$:

$$\boldsymbol{\xi}^{\text{new}} := \boldsymbol{\xi}^{\text{current}} - \alpha \nabla_{\boldsymbol{\xi}} J(\boldsymbol{\xi}^{\text{current}}; \mathcal{S})$$

Multilayer Perceptron: Learning II



Multilayer Perceptron: Learning III

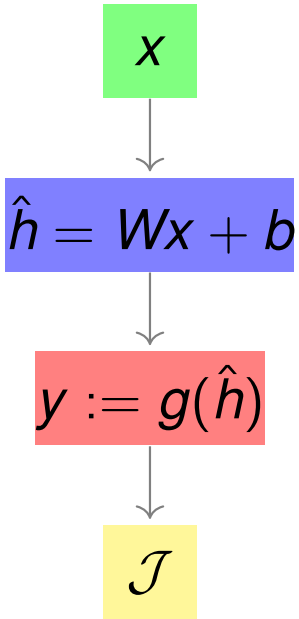
- $L = 1$, $m_1 = m_2 = 1$.
- No indices, and $W \in \mathbb{R}^{m_0}$ vector.
- Red: computable.
  Blue: already computed.
- Chain rule:
  1. First step (analogous for bias):

$$\frac{\partial \mathcal{J}}{\partial \hat{h}} = \frac{\partial \mathcal{J}}{\partial y}\frac{\partial y}{\partial \hat{h}} = \frac{\partial \mathcal{J}}{\partial y} g' \ .$$

  2. Second step:

$$\frac{\partial \mathcal{J}}{\partial W_k} = \frac{\partial \mathcal{J}}{\partial \hat{h}}\frac{\partial \hat{h}}{\partial W_k} = \frac{\partial \mathcal{J}}{\partial \hat{h}} x_k \ .$$

  (analogous for $\frac{\partial}{\partial b}$).



Multilayer Perceptron: Learning IV

- Gradient descent:

$$\begin{aligned} W_k^{\text{new}} &= W_k^{\text{current}} - \alpha \frac{\partial \mathcal{J}}{\partial W_k} \\ &= W_k^{\text{current}} - \alpha x_k \frac{\partial \mathcal{J}}{\partial \hat{h}} \\ &= W_k^{\text{current}} - \alpha x_k \, g' \left(W^{[\text{current}]} x + b^{[\text{current}]} \right) \frac{\partial \mathcal{J}}{\partial y} \end{aligned}$$

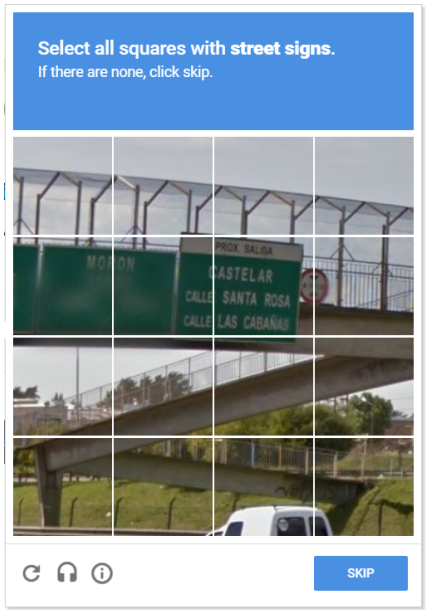
- Exercises: Generalise this formula for
  1. More than one layer, but still all dimensions $m_l = 1$, $l \geq 1$.
  2. Only one layer, but $m_1, m_2 \geq 1$.
  3. General case: L layers, dimensions m_l arbitrary.

  $\Rightarrow$ **No difficult exercise!** Chain rule, but careful with indices.

Backpropagation

Datasets I

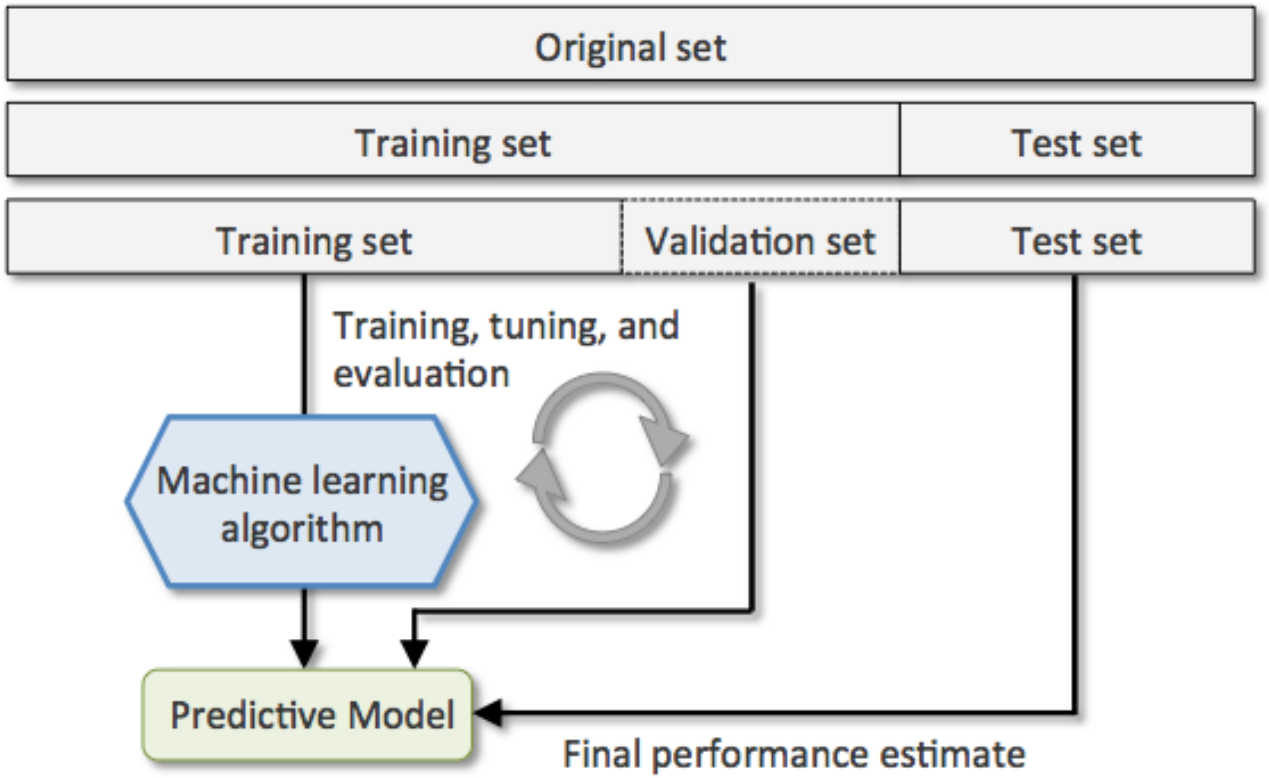
- In practise: **Obtaining good and large datasets is hard.**
- Large online corporations generate enormous datasets and have users label it (“captcha”).
- **Croudsourcing:** Humans label a dataset.
- Competitions with data sets (`https://www.kaggle.com/competitions`).



Datasets II

- **Training set:** We need data to train a neural network (e.g. a MLP).
- **Test set:** Once the neural network is trained: *Need to evaluate performance* on another labeled dataset!
- **Split the dataset:** Split all labeled data into a subset used for training and its complement used for testing.
- **Hyperparameters:** Parameters that determine Φ: $L, \{m_l\}_l, \alpha, \ldots$
- **Validation set:** Actually, keep a small part of the data set on the side to test different hyperparameters.

Datasets III



Lab: MLP for digit recognition 2: Training

1. Define the architecture of the Neural Network.
2. Load the dataset: Greyscale images are matrices with entries between 0 (black) and 1 (dark)[1].
3. Split it into Training and Test set.
4. Visualize some of the elements.
5. Train the network.
6. Run a prediction.
7. Test the accuracy on the test set.

1 Actually, images get saved as an array of integers between 0 an 255.

Optimisers

- Gradient descent: Easiest to understand, but outperformed by many derived algorithms.
- ADAM: *Adaptive moment estimation*:
  1. Each iteration on a *random batch* of the dataset.
  2. Renders gradient a random quantity.
  3. Estimates the first and second moments of gradient.
  4. New hyperparameters related to estimation.
  5. Finds an adaptive learning rate α in each step.
-

Remarks

- "Multilayer Perceptron" is "Feed-forward Neural Network"
- Do **one** optimization step on **various** samples in one iteration: **Batch size** as hyperparameter.
- **Epoch:** Number of times optimizer runs through entire dataset.
- The more complex a Neural Network gets, the more complex it can be to compute the gradient.
- **Rademacher's Theorem:** If $g^{[l]}$ is *locally Lipschitz continuous*, then it is (Lebesgue-)almost everywhere differentiable, but the the chain rule may not hold everywhere. Generalisations are being developed[2]. $\Rightarrow$ Many open questions.

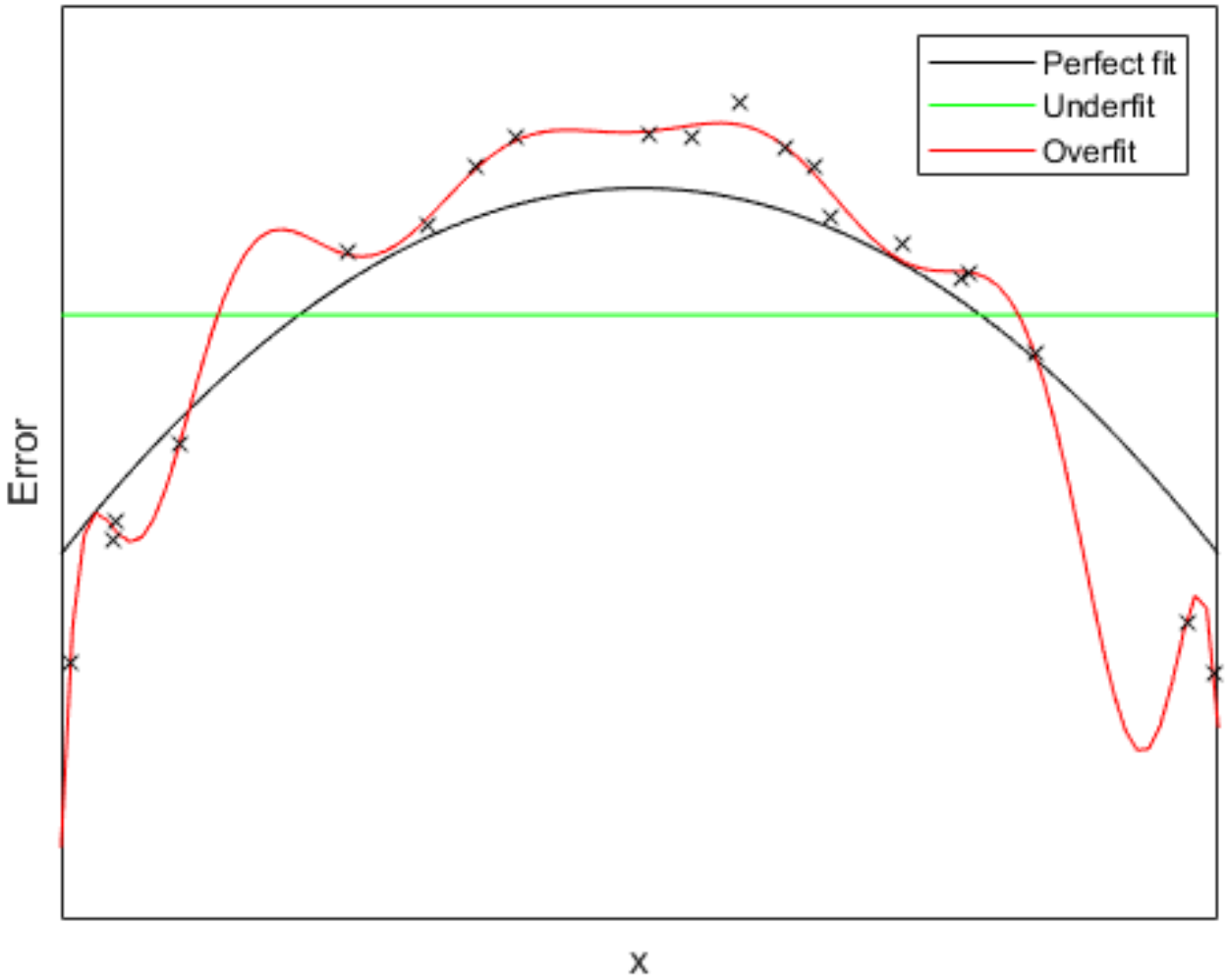
2 Berner et.al. *Towards a regularity theory for ReLU networks - Chain rule and global error estimates*. May 2019. `https://arxiv.org/pdf/1905.04992.pdf`

Overfitting and Underfitting I

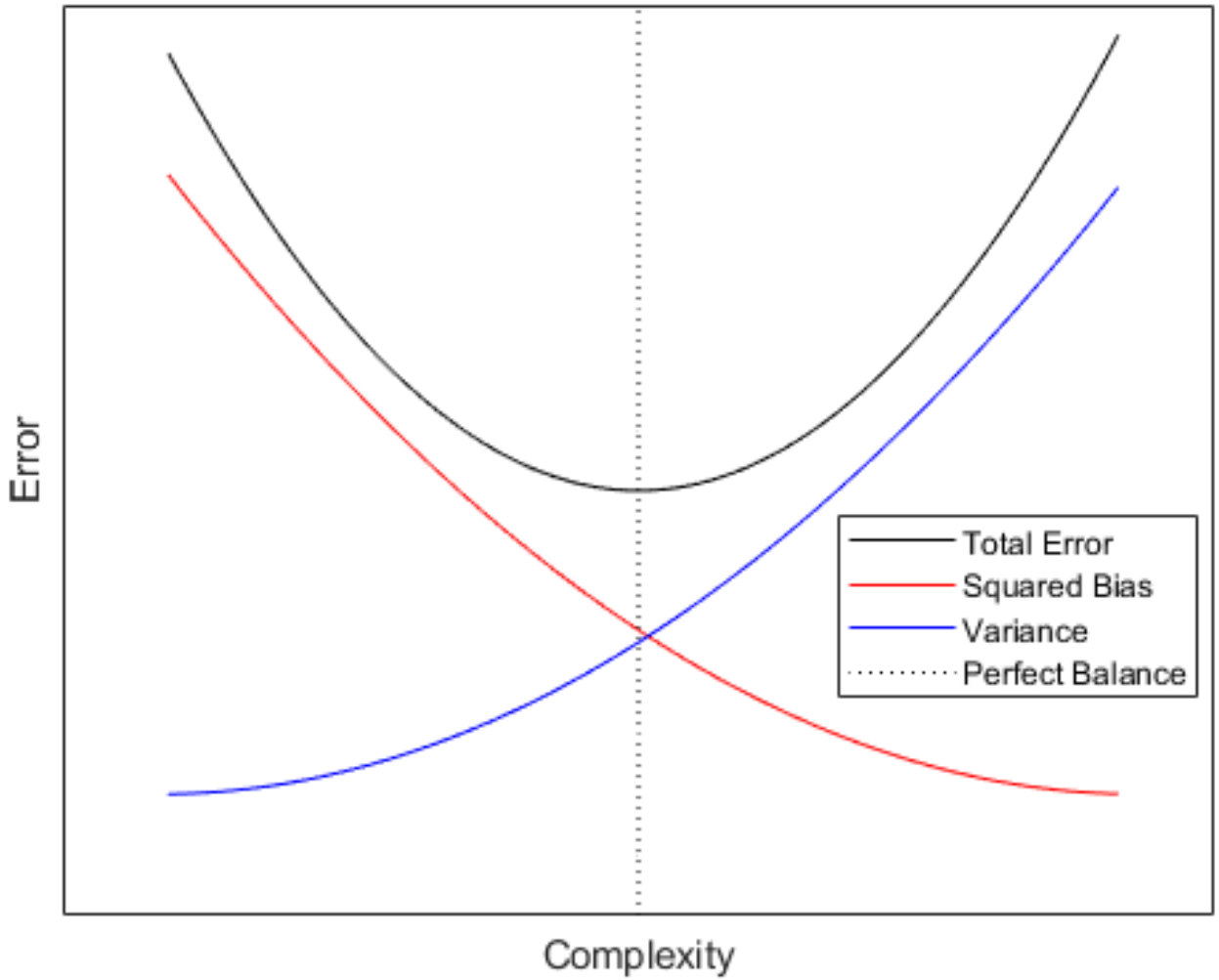
After training a Neural Network, observe accuracy on training set and on test set.

- If Φ is **inaccurate on the training and test set**
  - Model does not capture data (Underfitting).
  - Model has low variance, high bias.
  - **Change:** More epochs, bigger training set, more layers, larger layers.
- If Φ is **accurate on the training set, inaccurate on test set**
  - Model interpolates training data, extrapolates poorly: **Overfitting**
  - Model has high variance, low bias.
  - **Change:** Fewer epochs, smaller training set, smaller Φ.

Overfitting and Underfitting II



Overfitting and Underfitting III



Overfitting and Underfitting IV

- **Regularization:**
  - Replace $\mathcal{J}(\mathbf{\Phi}; \mathcal{S})$ by

$$\mathcal{J}(\mathbf{\Phi}; \mathcal{S}) + \lambda \, \text{Reg}(\mathbf{\Phi}) \, ,$$

    where

$$\text{Reg}(\mathbf{\Phi}) := \frac{1}{L} \sum_{l=0}^{L} \left\| W^{[l,l+1]} \right\|^2 \, .$$

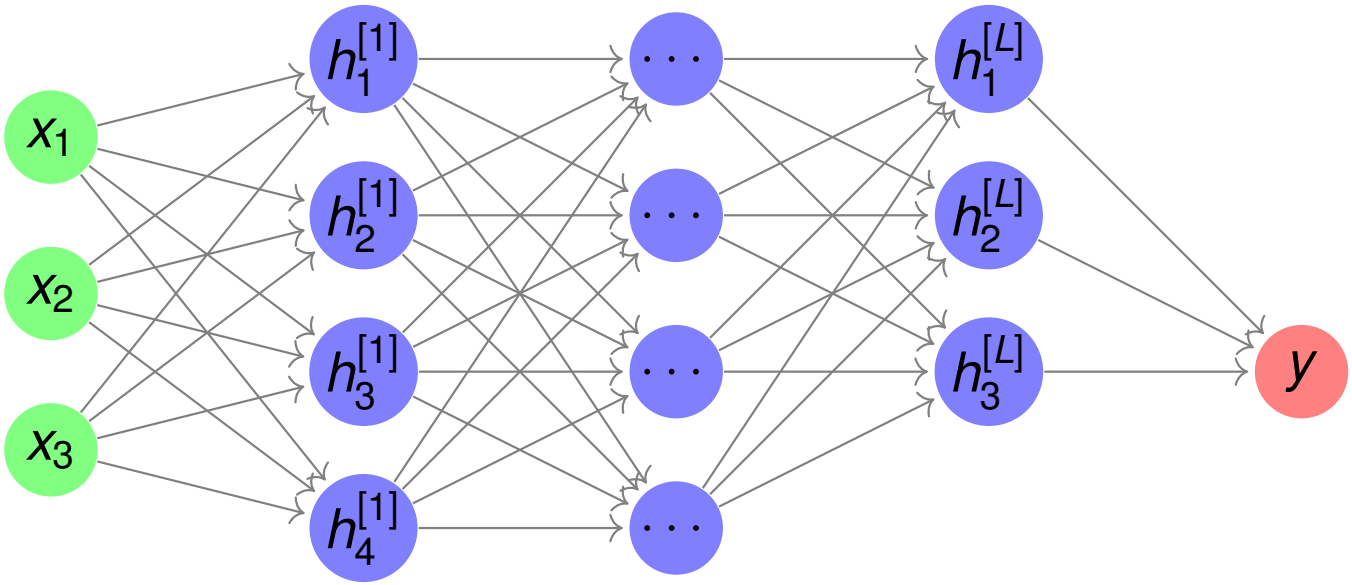
    Common to choose the Frobenius Norm.
  - Penalise large weights $\Rightarrow$ scaled input for activation.
  - **New hyperparameter** λ.
  - `PyTorch:` Parameter `weight_decay` in `optimizer`.

Overfitting and Underfitting V

- **Dropout:**
  - **During training**, randomly deactivate some neurons.
  - Choose a dropout probability $p_d \in [0, 1)$.
  - During training, each neuron will be multiplied by 0 with probability p_d.
  - "Decrease the influence of single neurons on the output of Φ."
  - `PyTorch`: Before training, set `model.eval()` to deactivate dropout.

Initialisation I

- Optimisation: $\xi^{\text{new}} := \xi^{\text{current}} - \alpha \nabla_{\xi} J(\xi^{\text{current}}; \mathcal{S})$
- Initial value needed.
- What happens if we initialise all weights to 0?



Initialisation II

- **Random initialisation:** Draw initial parameters from probability distribution (uniform, normal).
- What happens if the initial values are around zero?
- What happens if the initial values are large in absolute value?
- **Idea:** For $W^{[l,l+1]}$, decrease variance with m_l to ensure that weights get iterated differently.
- **Xavier initialisation**[3]**:** Initialisation of $W^{[l,l+1]}$ by unform distribution around 0 with standard deviation $\frac{1}{m_l}$.

  Works well with $\tanh$, σ activations.
- **He initialisation**[4]**:** Initialisation of $W^{[l,l+1]}, b^{[l,l+1]}$ by $\mathcal{N}\left(0, \frac{1}{m_l}\right)$.

  Works well with ReLU and

$$\text{PReLU}_\beta(x) := \max(-\beta x, x) , \qquad \text{(typically, } 0 < \beta << 1) .$$

3 Glorot and Bengio: *Understanding the difficulty of training deep feedforward neural networks.*, 2010.

4 He et al.: *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification.*, 2015.

Deep vs. Shallow

- Adding more layers vs. enlarging present layers: Adding more features vs. combining present features.
- *Deep* layers ($h^{[l]}$ for large l) learn more complex features.

Theorem (Approximation properties of Neural Networks (Sketch))

0. *Given a function $f \in C^{(0,1)(\Omega)}$ to be approximated by Φ, $\Omega \in \mathbb{R}^{m_0}$ compact.*
1. *Both shallow (small L) and deep (large L) NN can compute the best approximation of f in finitely many steps.*
2. *Let M be complexity needed to surely attain accuracy $\varepsilon > 0$.*
   - *If $L = 1$, need to increase m_1 exponentially to attain $\varepsilon > 0$.*
   - *In the deep case, it is sufficient to increase L at most linearly.*
   - *Depends on complexity of f: "How many compositions are needed to express f in elementary functions?"*

Recapitulation: Neural Networks

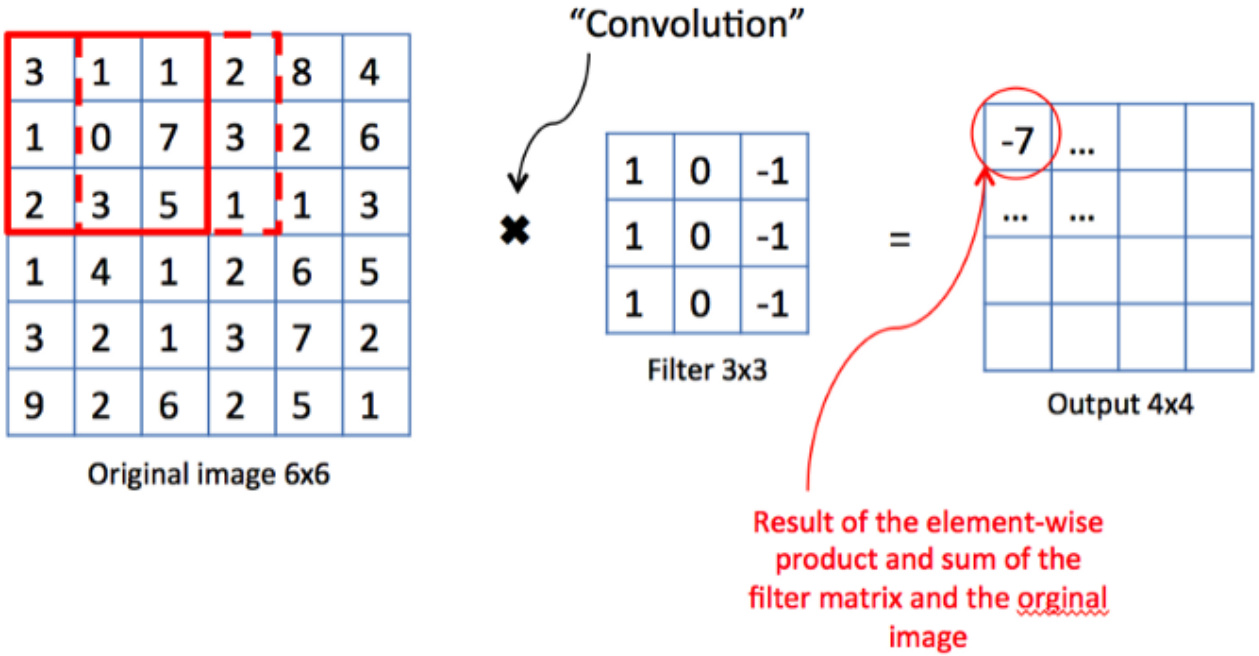
1. A Neural Network (MLP) Φ contains a forward step (prediction) and a backward step (learning).
2. **Forward step:** Composition of affine maps and activation functions, using learned parameters.
3. **Backward step:** Optimization ("learning") of parameters to minimise a *loss function* $\mathcal{J}(\Phi; \mathcal{S})$.
4. **Initialisation:** Find "smart" way to initialise minimisation.
5. **Train/Test set:** $\mathcal{S} = \mathcal{S}_{\text{train}} \dot{\cup} \mathcal{S}_{\text{test}}$, usually $\frac{|\mathcal{S}_{\text{train}}|}{|\mathcal{S}|} \in [0.6, 0.9]$.
6. **Bias-Variance tradeoff:** Φ needs to fit the data *reasonably* well, but still extrapolate on unknown data.
7. **"Magic":** Many heuristic tweaks, not fully understood why they work well.
8. "Deep": $L \simeq$"depth" of Φ. The deeper the network, the more complex the learned features.

Convolutional Layers I

- Before: Each layer is $g^{[l]}$ of a linear combination of $h^{[l-1]}$: **Fully connected layers**.
- Introduce new kinds of layers for image processing.
- **Motivation:** Visual computing.
- **Image input:** $n_x \times n_y$ pixels, n_c channels: $m_0 = n_x n_y n_c$!
- **Idea:** To obtain a sense of vicinity: Go through each pixel and combine only its surrounding pixels linearly.
- To maximize confusion, what we call convolution here is actually a cross-correlation. Difference: Transposition of the filter.

Convolutional Layers II

- **Convolution with a filter:**



- This is an example of a **vertical edge detector**: If the resulting number is large, it means that the pixel values to the left and to the right are different.

Convolutional Layers III

- Classical Edge detection: Design such filters for each use case (Sobel, Laplace, Scharr, ...)

- **Convolutional layer:** One layer consists of m_l filters. The entries of the filters are *learned*.
- Resulting size: Image size $n \times n$, Filter size $f \times f$: Results in output size $(n - f + 1) \times (n - f + 1)$.

Padding

- **Problem:** Each time a filter is applied, image size decreases.
- **Padding:** Embed the image inside an outer layer of zeroes.

0	0	0	0	0	0	0	0
0	3	1	1	2	8	4	0
0	1	0	7	3	2	6	0
0	2	3	5	1	1	3	0
0	1	4	1	2	6	5	0
0	3	2	1	3	7	2	0
0	9	2	6	2	5	1	0
0	0	0	0	0	0	0	0

- Resulting output size with padding p, input size n, filter size f: $(n + 2p - f + 1) \times (n + 2p - f + 1)$.
- Enforce input size=output size by $p = \frac{f-1}{2}$.
- Therefore filter sizes are usually odd numbers.

Strided convolution

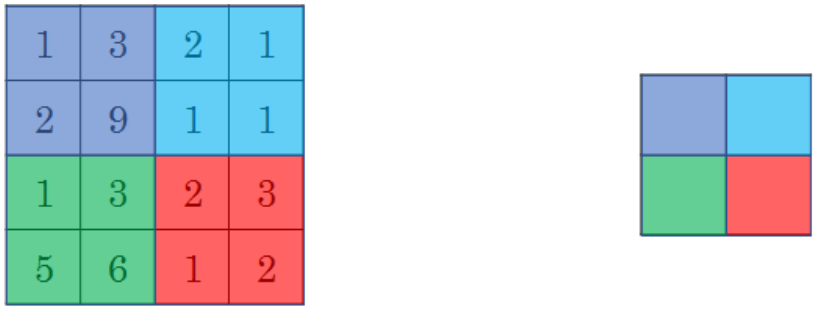
- So far:
  1. Apply a filter to pixel (i, j) in input to get pixel (k, l) in output.
  2. Apply a filter to pixel $(i+1, j)$ in input to get pixel $(k+1, l)$ in output.
- With **stride** s:
  1. Apply a filter to pixel (i, j) in input to get pixel (k, l) in output.
  2. Apply a filter to pixel $(i+s, j)$ in input to get pixel $(k+1, l)$ in output.
- Output size: $\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$

Recap: Convolutional Layers

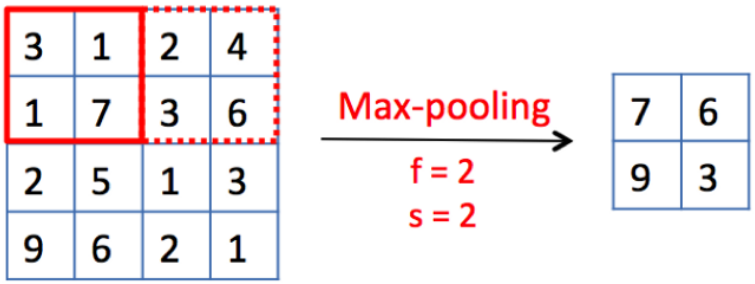
- A **convolutional layer** applies a filter to an output to detect a feature.
- What kind of filter yields the best classification results will be learned.
- To ensure more flexibility, a convolutional layer has hyperparameters **Filter size, Padding, Stride**.
- Usually followed by ReLU activation function.

Pooling layer I

- Convolution can result in large output image, features too detailed.
- **Idea:** Generate a “summary” of the different areas in the image.



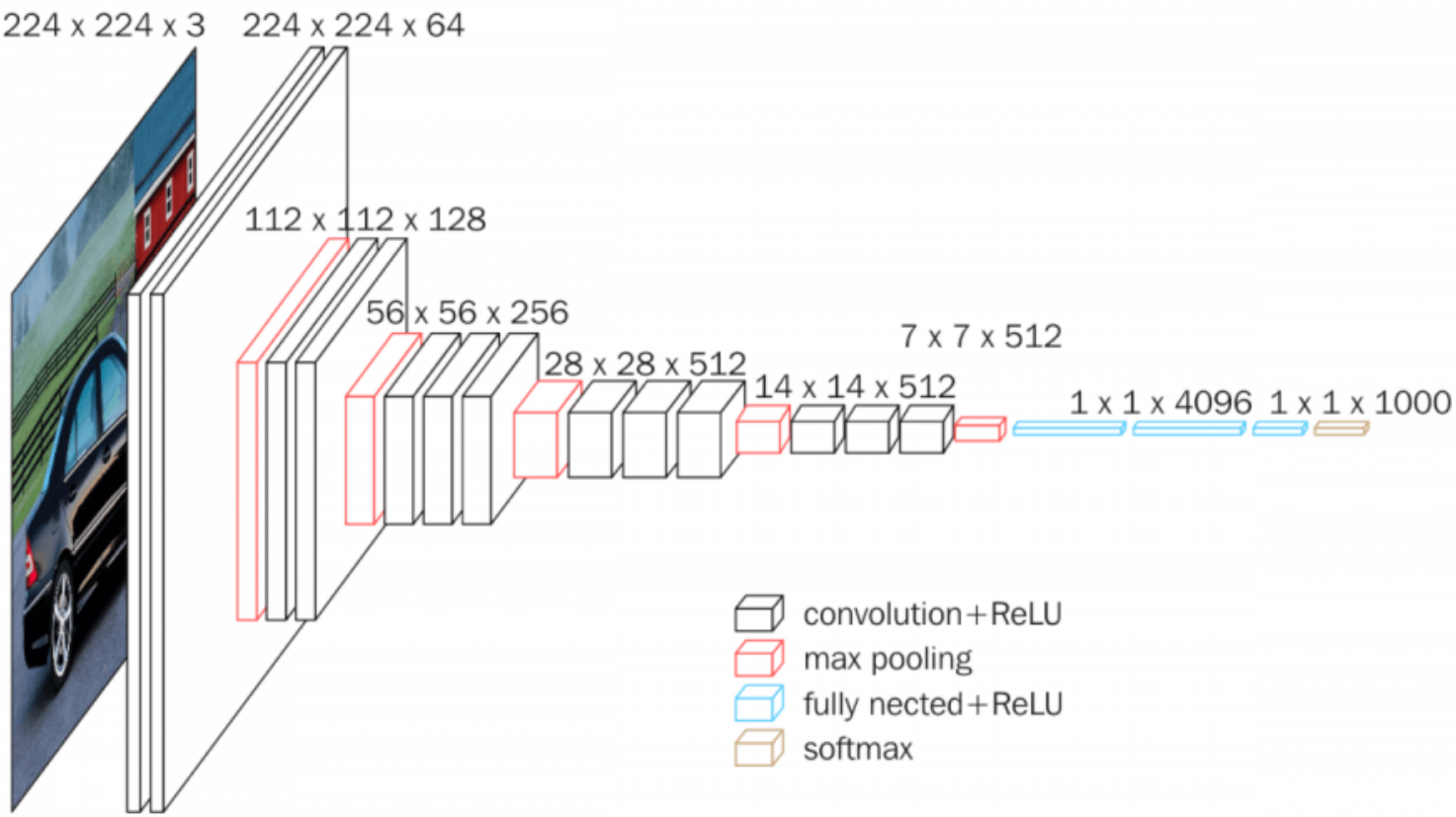
- Most common: **Max-pooling**.



Pooling layer II

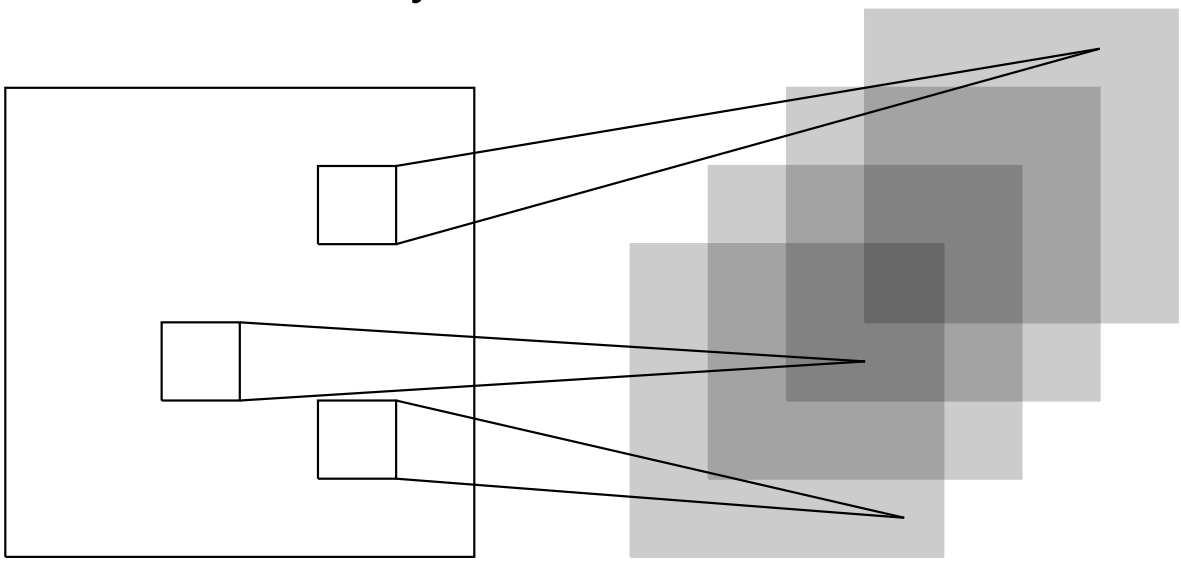
- Hyperparameters: filter size, stride, kind of pooling.
- Introduces more flexibility on output size.
- Reduces output to relevant features.
- Therefore usually combined with ReLU.
- Considered as part of the convolutional layer.

A typical CNN architecture: VGG-16 I



A typical CNN architecture: VGG-16 II

- Developed by the **V**isual **C**omputing **G**roup in Oxford[5]
- VGG-16 classifies for 1000 objects on an image.
- Image size decreases while number of filters increases.
- Backpropagation now involves derivatives w.r. to the new parameters.
- Convolutional Layers learn "m_l" filters: **Feature maps**



5 Simonyan and Zisserman: *Very Deep Convolutional Networks for Large-Scale Image Recognition.* 2015

Lab: Visualisation of Neurons

- Lower layers learn simple features: Edges, curves, repetitive patterns.
- Higher layers learn complex features, distinguishing very similar outputs (cat ears, dog ears, eyes,...)
- Visualization of the filters contained in one layer:
  1. Start with a random image.
  2. Choose a neuron $f_k^{[l]}$: Filter k in layer l.
  3. **Modify the image to maximize the activation of $f_k^{[l]}$.**
  4. Display the resulting image.

Lab: Style transfer learning I

- Given one image I_s as a style template and one image I_c as a content template.
- **Goal:** Create an image of the content of I_c, but in the style of I_s.

Lab: Style transfer learning II

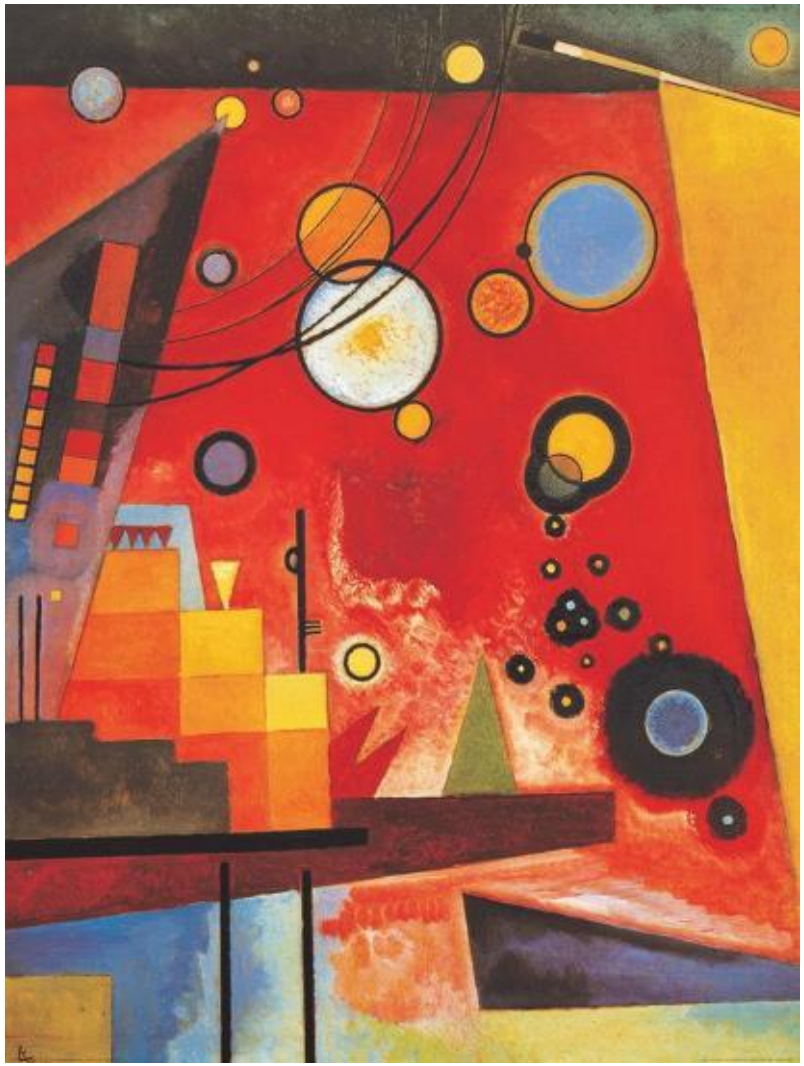
- **Style transfer learning:** Use a trained Neural Network to measure *difference in style* and *difference in content*.
  1. Load a pre-trained Convolutional Network.
  2. Identify layers and neurons (filters) corresponding to *style* and *content*.
     Their output define a content loss and a style loss:

$$\mathcal{J}_{\text{content}}(I, I_c) = \| \text{content}(I) - \text{content}(I_c) \|^2 ,$$

     analogously with style loss.
     Total variation loss: Combination of style and content loss, plus regularization.
  3. Start with randomised image and iterate the pixel values to minimize Total variation loss.

Style Transfer: Results

Style Transfer: Results



"Heavy Red", Kandinskiy 1924

Style Transfer: Results

"University of Cape Verde", Kandinskiy 2019

Style Transfer: Results

Style Transfer: Results

"Starry Night", Van Gogh 1889

Style Transfer: Results

"Starry Monte Cara", Van Gogh 2019

Style Transfer: Results

Style Transfer: Results

"Bavarian Village", Kandinskiy 1908

Style Transfer: Results

"Lake Biel", Kandinskiy 2019

Some things left out

- **Data augmentation:** How to get more out of a dataset.
- **Data normalization:** How to scale the input to increase learning performance.
- **Exploding gradients:** Really large L imply instabilities of gradients.
- **Image segmentation** vs. classification.
- Recurrent Neural Networks (RNNs), Residual Networks, Generative Adversarial Networks, U-Nets, ...
- **RNNs:** If your data is *sequential* (e.g. Time series, Music, Natural Language), there is a type of neuron that learns to predict the next value: *Long short-term memory cells* (LSTM).