Machine Learning

Rudolf Mayer

April 1st, 2025

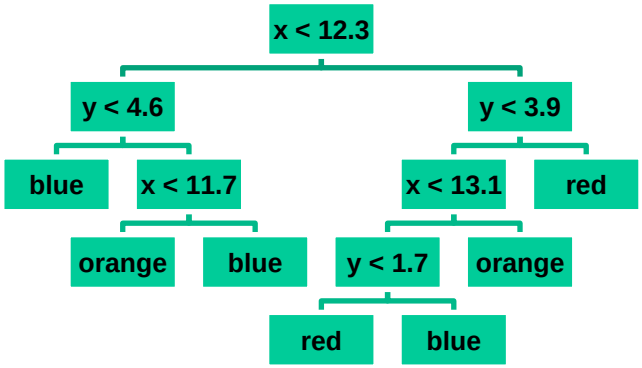
Outline

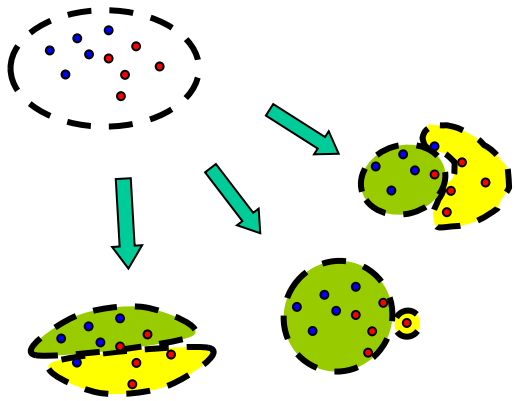
- Short Recap
- Neural Networks / MLP
- Feature Extraction (overview)
- Deep Learning
  - Intro / Definition
  - Convolutional Neural Networks
  - Convolutions in detail
  - …

Outline

- Short Recap
- Neural Networks / MLP
- Feature Extraction (overview)
- Deep Learning
  - Intro / Definition
  - Convolutional Neural Networks
  - Convolutions in detail
  - …

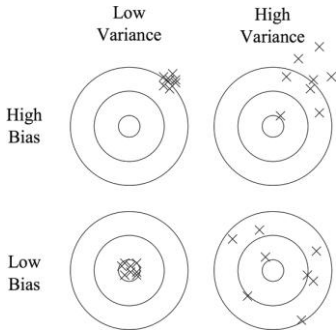
Short Recap



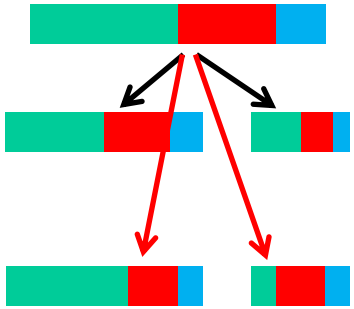


- Decision Tree Learning
  - Finding optimal split
    - Categorical attributes
  - Different criteria for optimality
    - Information Gain
    - Gini Index
  - Binary & multiple classes
  - Overfitting & (pre)pruning
  - Stability
  - Binary / n-ary trees
  - Categorical & numerical data

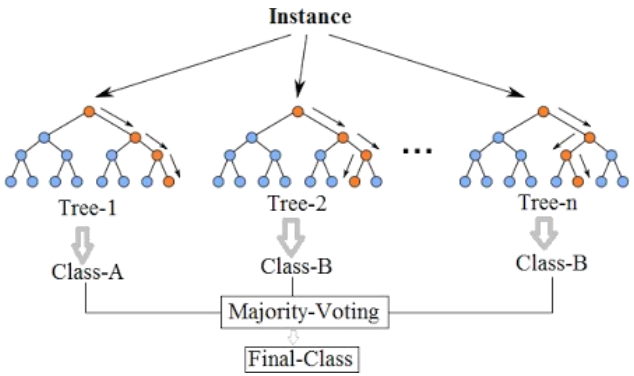
Short Recap



- Evaluation
  - Overfitting & Generalisation
  - Bias & Variance
  - Stratification

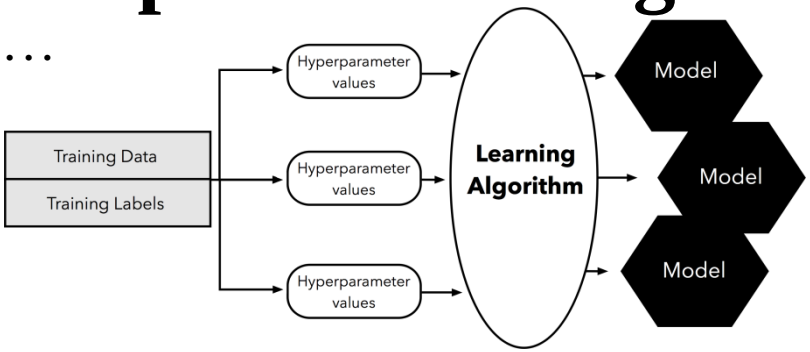


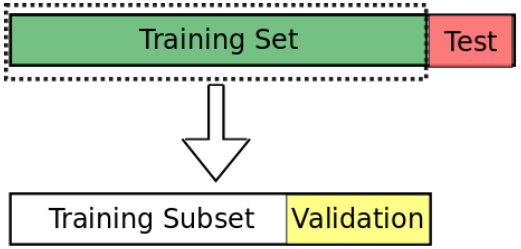
- Random forests
  - Bootstrapping
  - Majority voting
  - Out-of-bag estimate

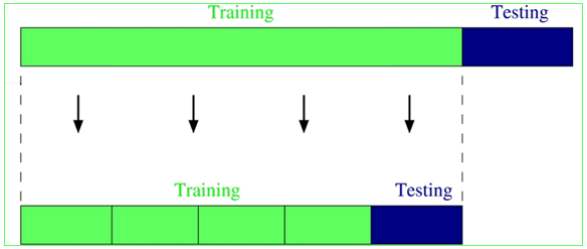


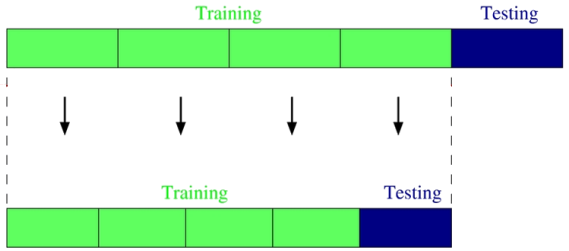
Recap: Evaluation / Experiment design

- Model selection:
  how to evaluate / estimate generalisation error









- More on that in later lectures
  - Grid Search / Randomised Search / Metalearning / Automated ML, ...

Remember: *Evaluation: data*

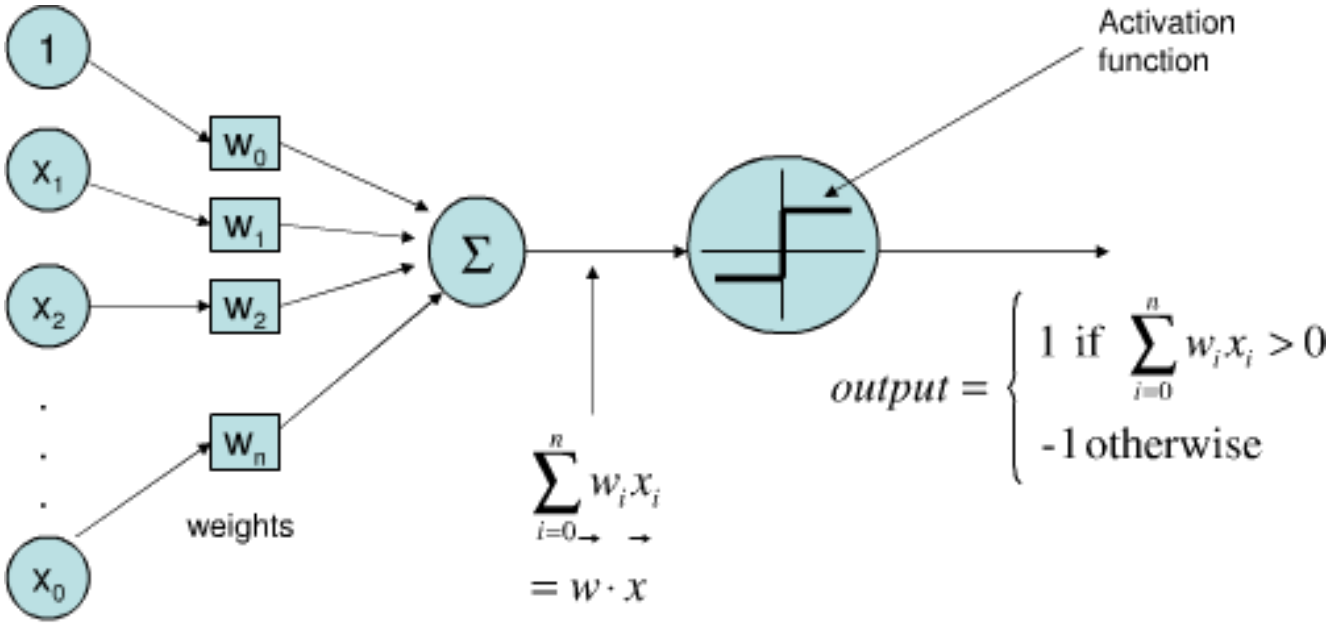
- Why not evaluate on *samples used for training?*
  - Would not tell us how good the model works for unknown data
    - Which is however why we train a model in first place ...
  - *If we test on training data – we are biased*
    - Fully grown decision trees – will be 100% on training data
    - Perceptron on linear separable data: training data will be 100% correctly "predicted"
    - K-NN, Naïve Bayes: not necessarily 100% correct on training data. Still biased!
    - Similar for other algorithms, e.g. SVMs, ...
  - *Want to actually find out:* how will model perform on **unseen** data!
- Need to avoid data (information) leakage
  - Data/information from test set must not be used in any step when training models
    - Includes several data preparation steps (especially if they use the class label)
    - Includes hyper-parameter optimisation / model selection
  - Further reading material in TUWEL

Outline

- Short Recap
- Neural Networks / MLP
- Feature Extraction (overview)
- Deep Learning
  - Intro / Definition
  - Convolutional Neural Networks
  - Convolutions in detail
  - …

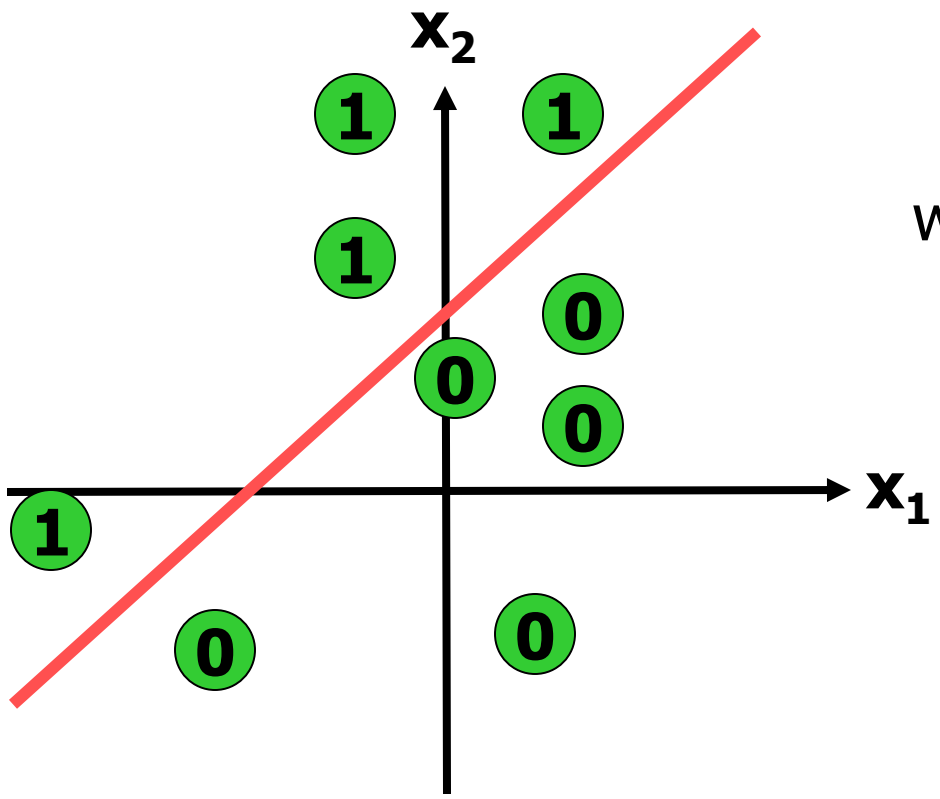
Recap: Perceptron

- Input: continuous values $\mathbf{X} = (x_1, x_2, x_3, \ldots x_n)$
- Output: activation a: Boolean value 0 / 1
  - Computed as weighted, linear combination of inputs (e.g. sum)
- Linear separation - can't e.g. model XOR function
- Artificial neural network – only one neuron



Perceptron: slope + bias

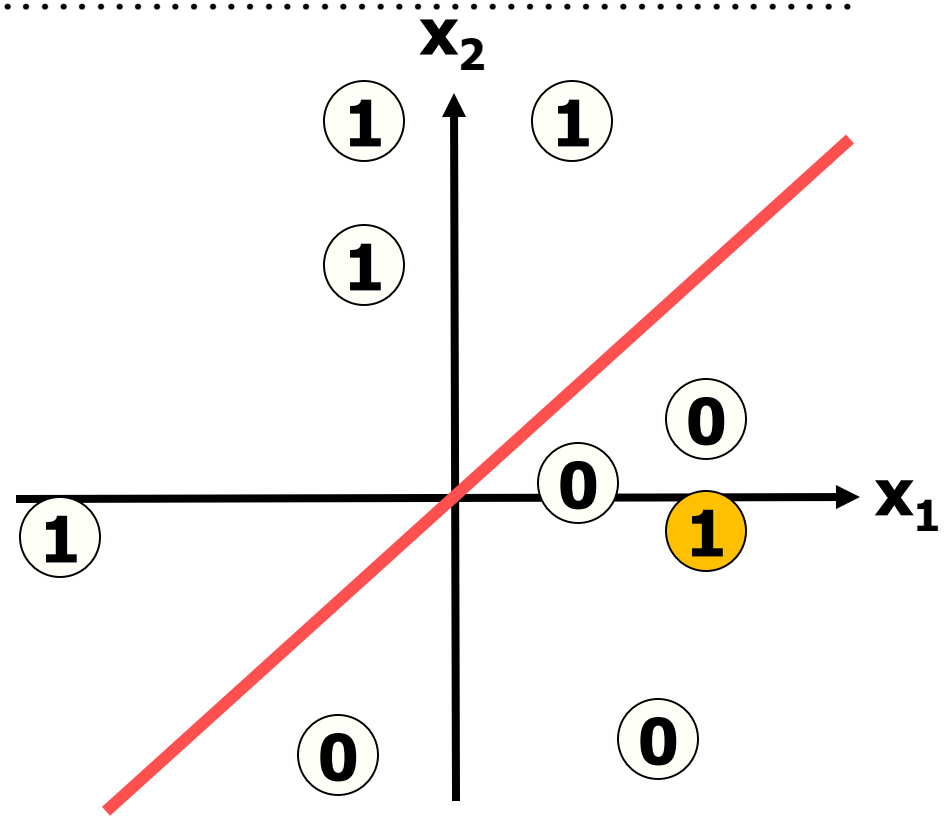
all f(x) = y ➔ final state



Decision line

$w_1 x_1 + w_2 x_2 + b = \theta$

Non-linear separable data



Perceptron will not converge to stable state

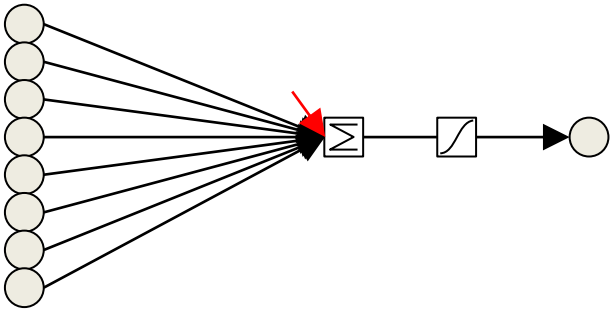
Need stopping criterion (e.g. number of iterations)

Extensions to Perceptrons

- (Kernel methods & perceptron ➔ see SVM)
- Perceptron is a (very) simple neural modell
  - Computes $y = \sigma(b + \mathbf{w}^T\mathbf{x})$ (σ … activation function)
- Many architectures proposed that combine multiple basic neurons
  - Input layer & output layer (as with Perceptron)
  - "Hidden" layers (not "visible" from the "outside")
- Early motivation: ***model of human brain***
  - *McCulloch-Pitt cell (1940s)*
  - Neurons connected via synaptic links
- ***Applications*** in AI / ML
- Different eras of popularity

Multi-Layer Perceptron

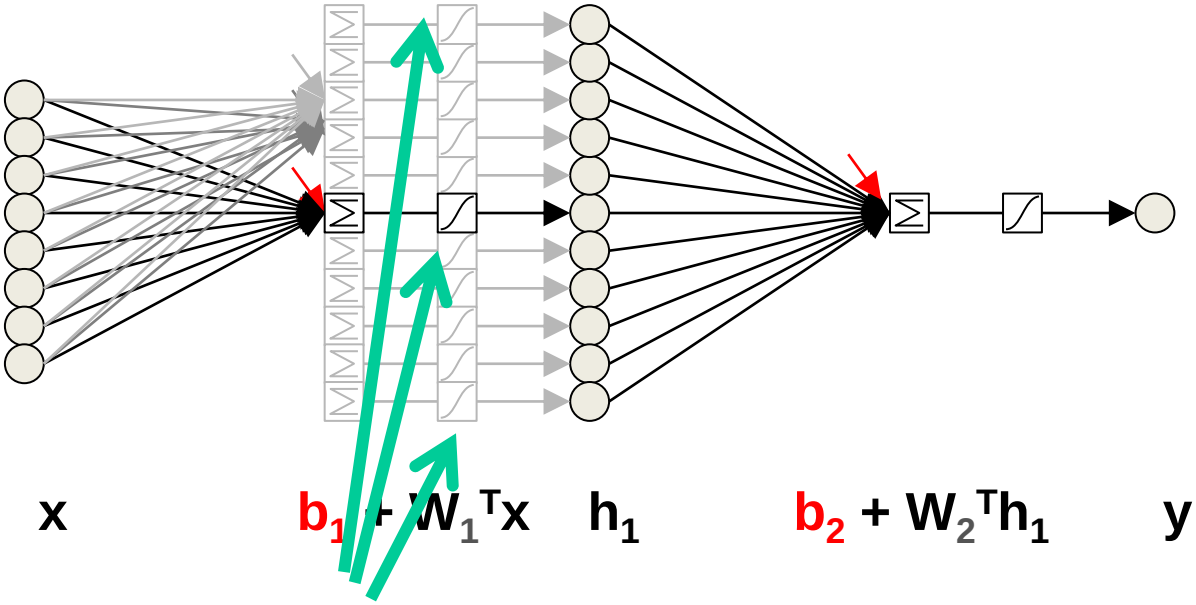
- Perceptron
- **Computes:** $y = \sigma(b_1 + W_1^T x))$



x **b** **+ W ^{T}x** **y**

Multi-Layer Perceptron

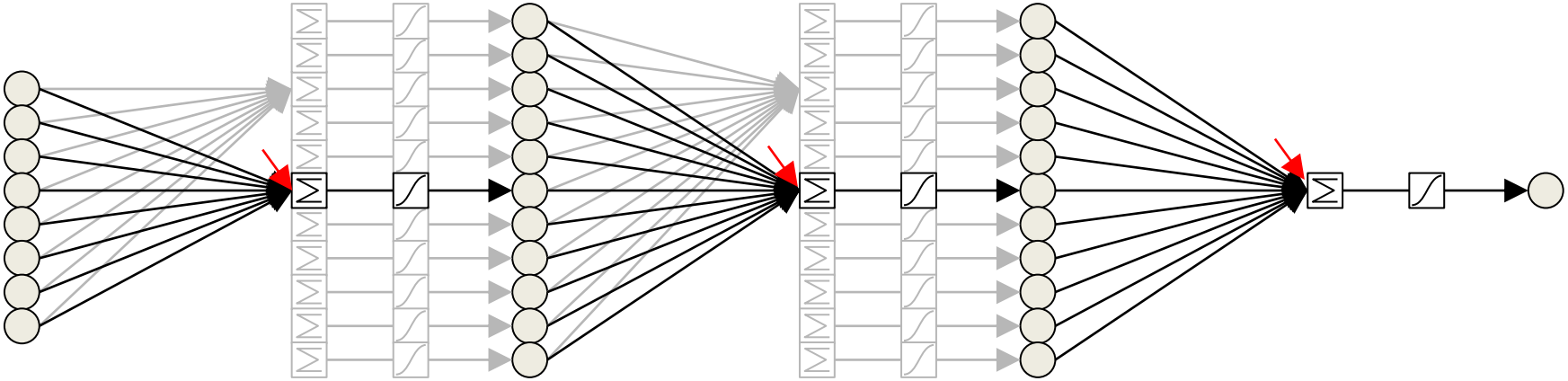
- MLP with *one* hidden layer
- **Computes:** $y = \sigma(b_2 + W_2^T \sigma(b_1 + W_1^T x))$



Layer composed of (11) perceptrons

Multi-Layer Perceptron

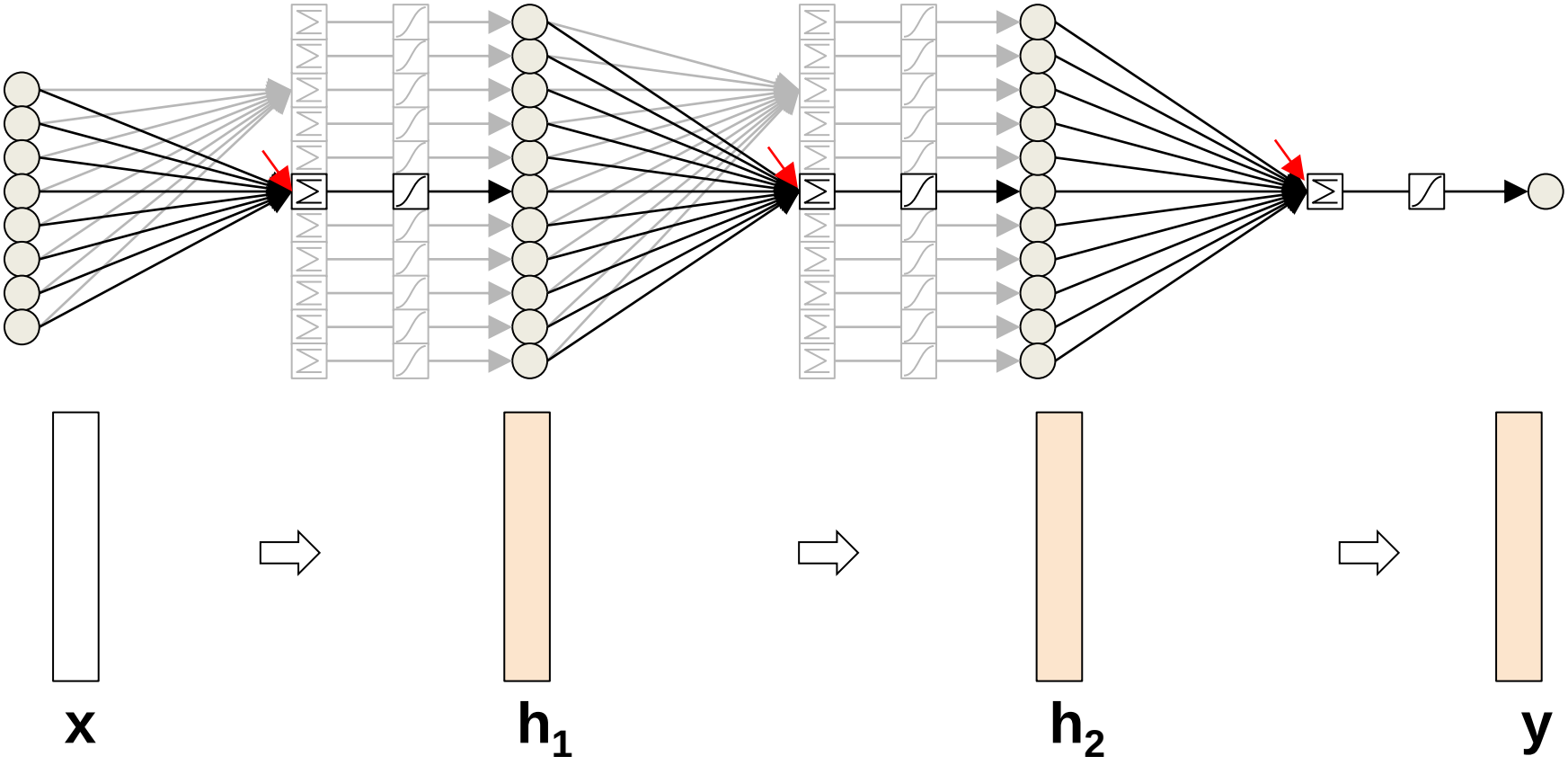
- MLP with *two* hidden layers
  - **Computes:** $y = \sigma(b_3 + W_3^T \sigma(b_2 + W_2^T \sigma(b_1 + W_1^T x)))$



x $\quad b_1 + W_1^T x$ $\quad h_1$ $\quad b_2 + W_2^T h_1$ $\quad h_2$ $\quad b_3 + W_3^T h_2$ $\quad y$

Multi-Layer Perceptron

- Abstract representation of layers

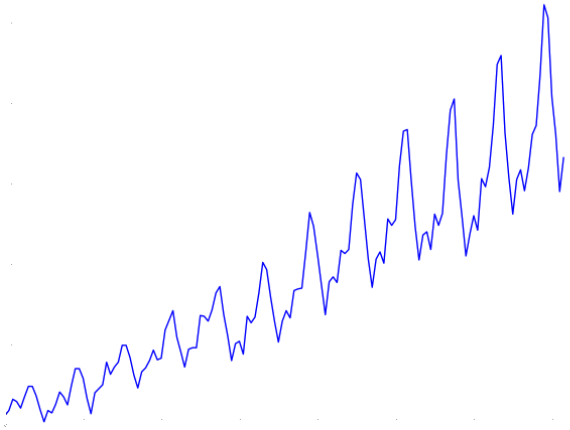


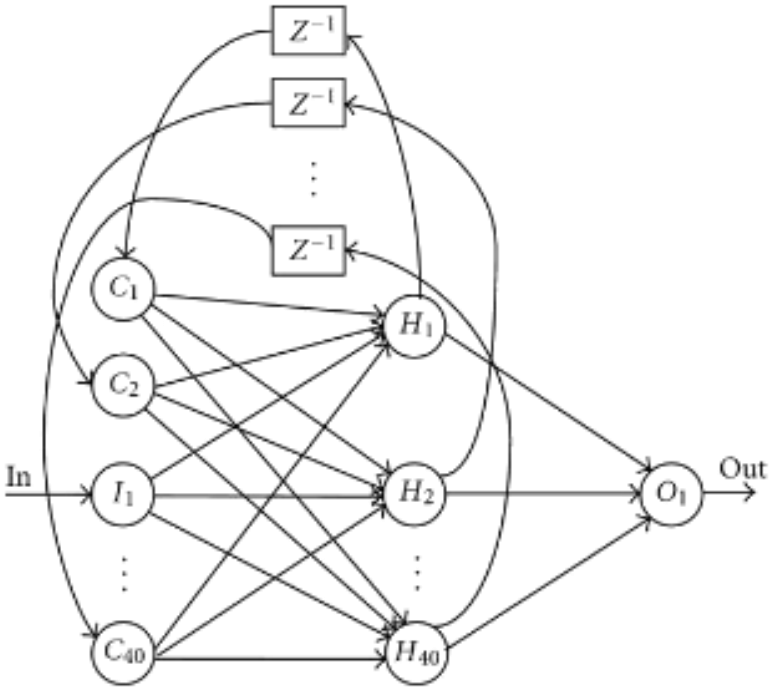
Multi-Layer Perceptron

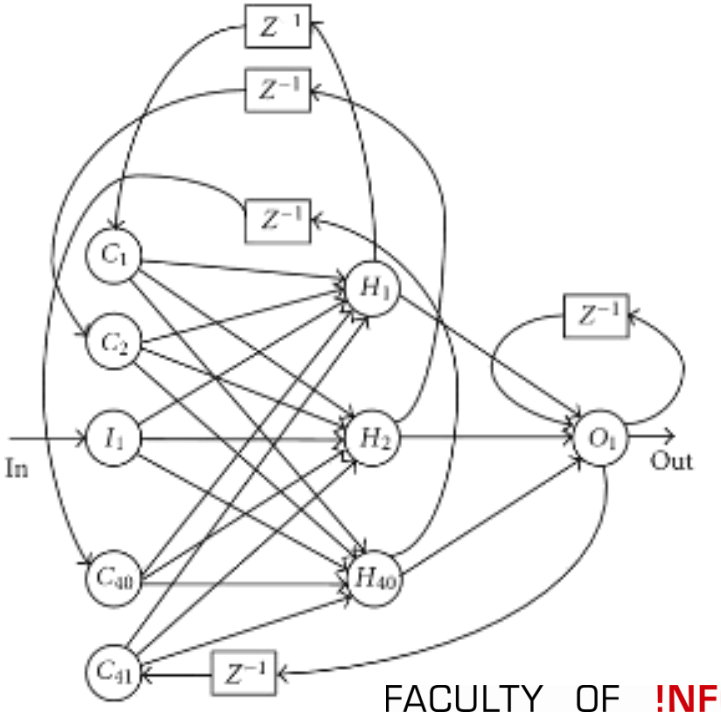
- Combination of simple perceptrons
  - Different activation function
  - Capable of universal function approximation
- Network: fully connected (directed) graph
- MLPs are *feed-forward* neural networks
  - Activation from a neuron feeds into next layer
  - Connections between neurons do not form directed cycle
  - Otherwise: *recurrent* neural networks

Neural Networks

- Recurrent network: architectures
  - Well-fitted for time-series analysis
  - Inputs of not-fixed length (text, …)
  - *More details later*

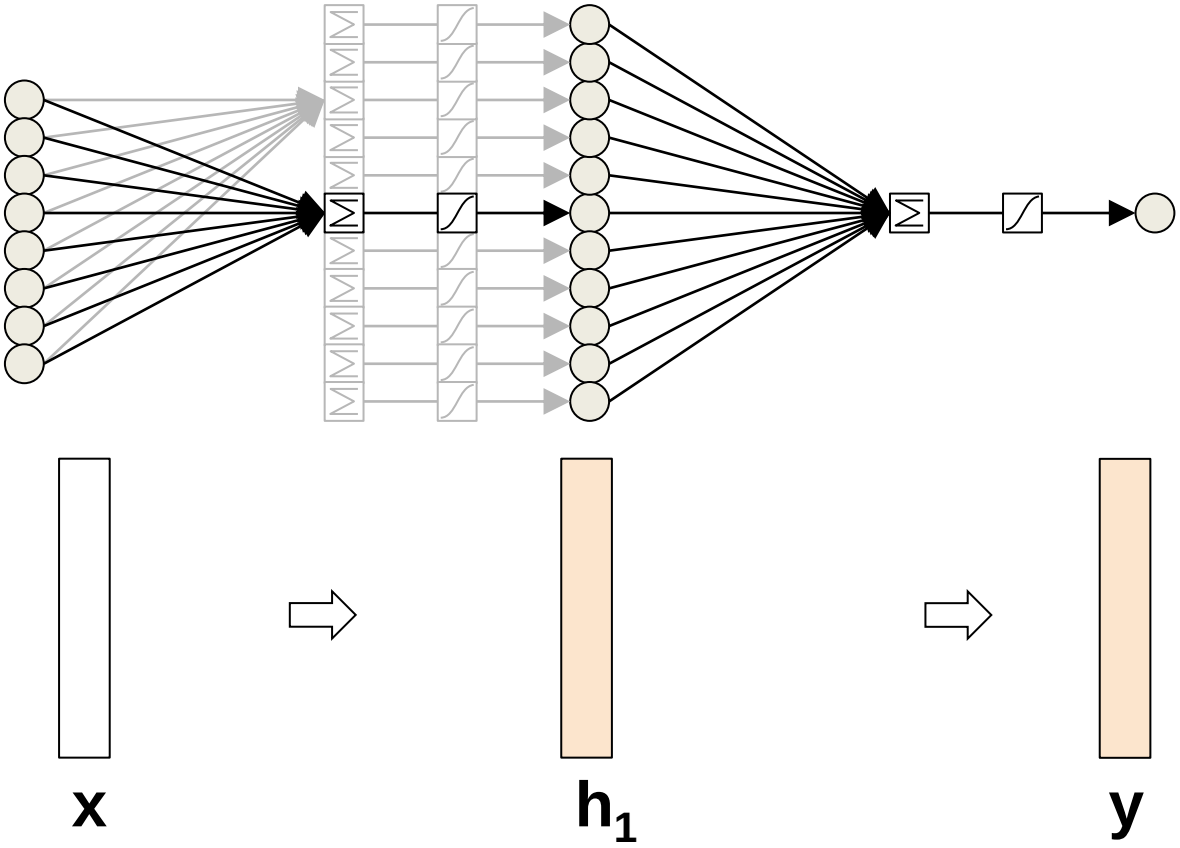






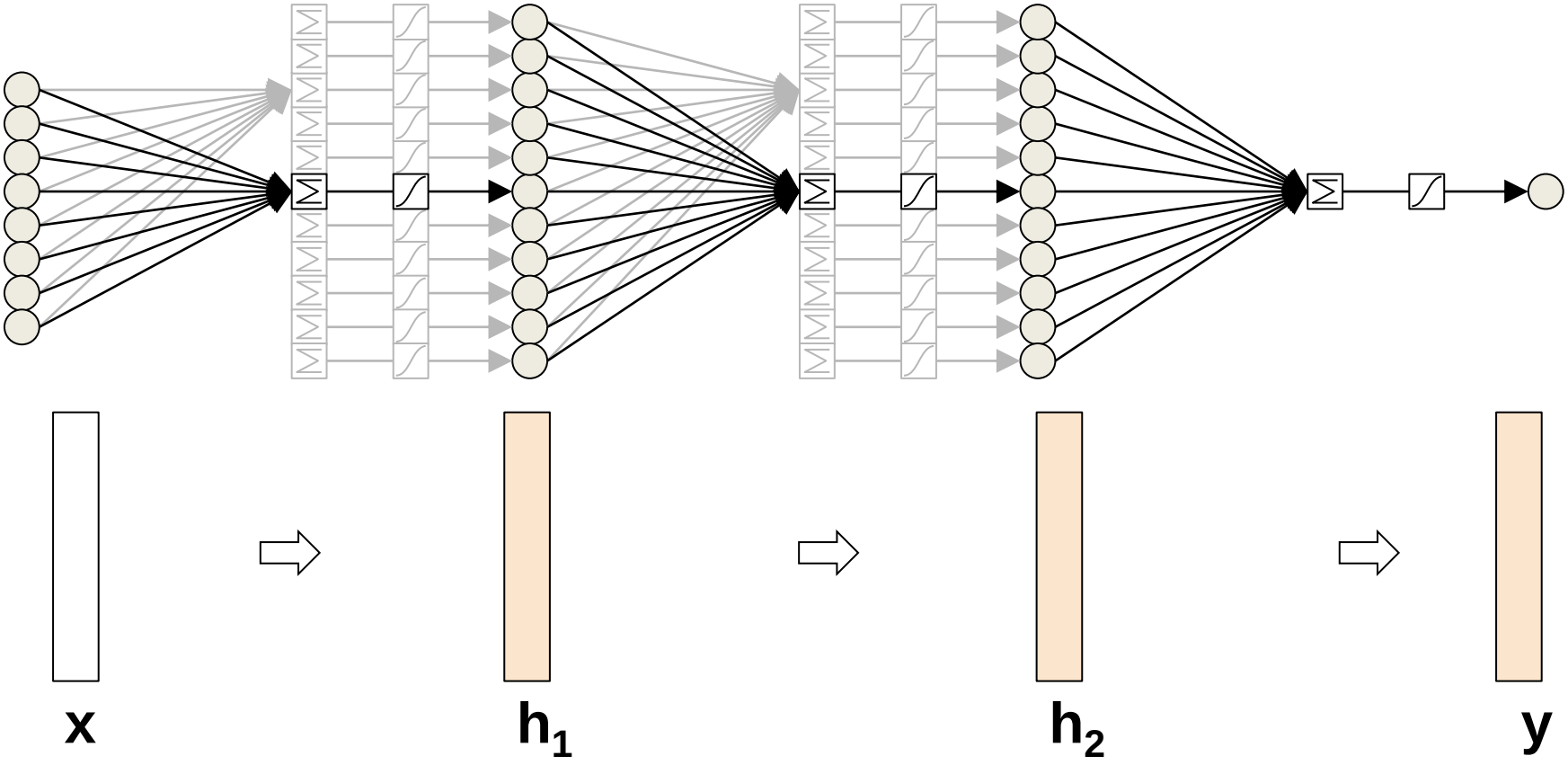
Multi-Layer Perceptron

Multiple hidden layers possible



Multi-Layer Perceptron

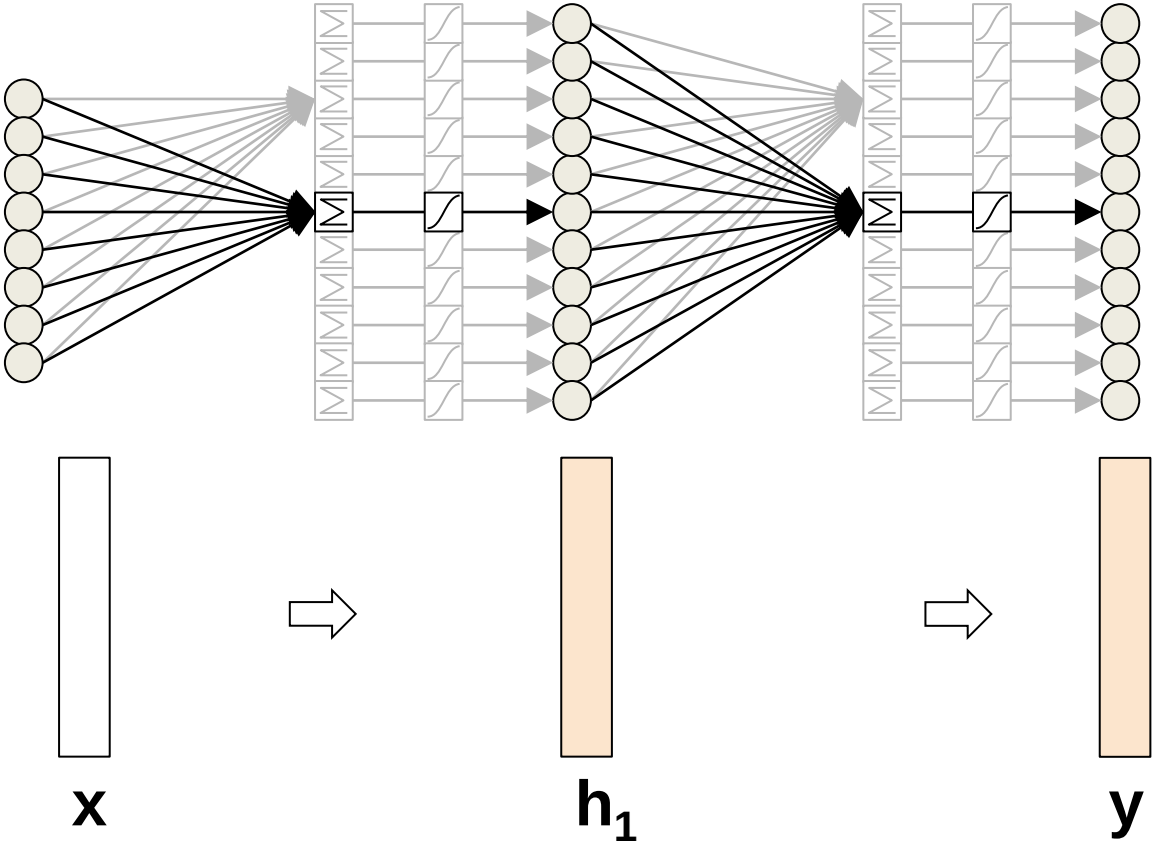
Multiple hidden layers possible



Multi-Layer Perceptron: Multi-output

Multiple output nodes possible – which scenario?

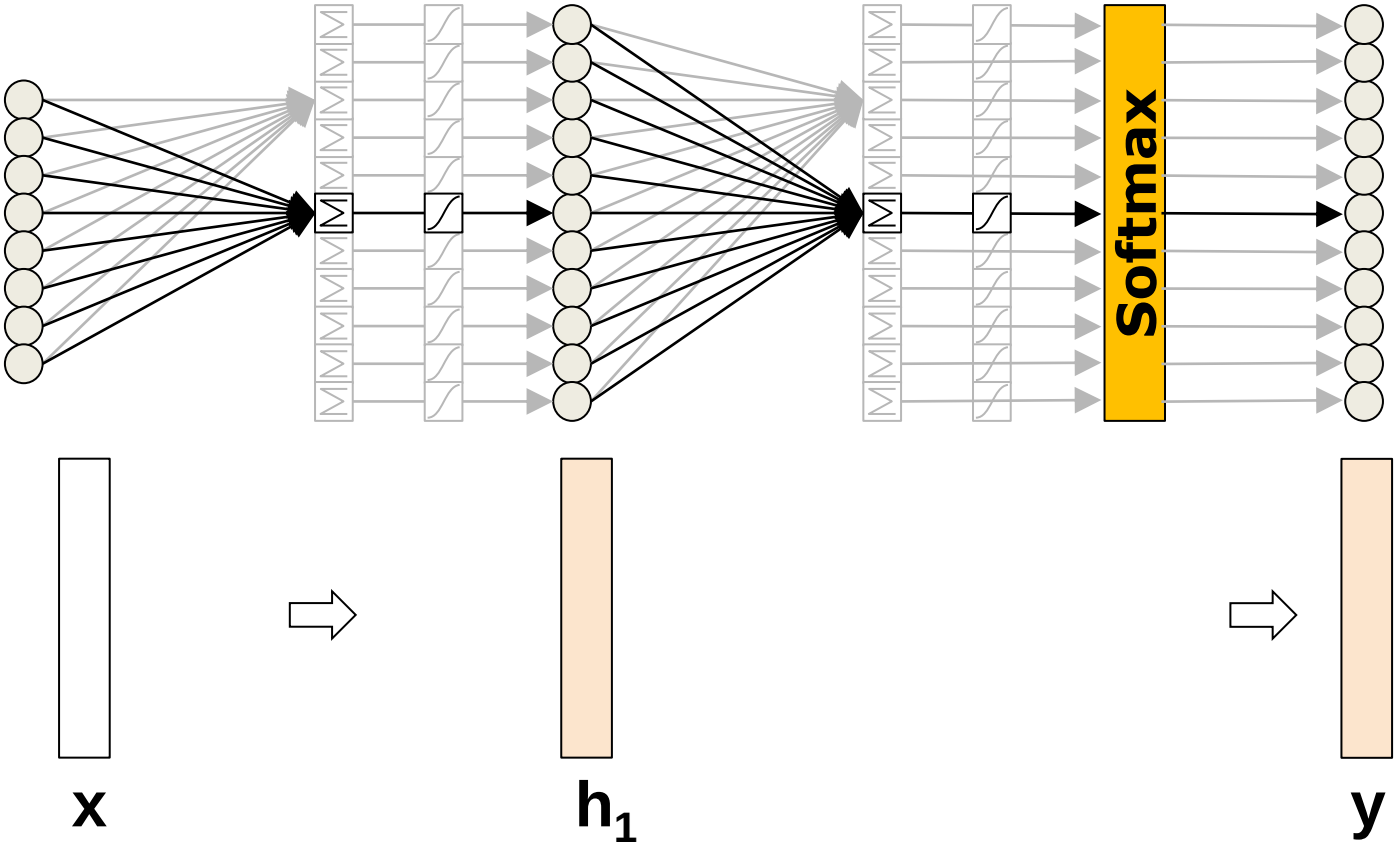
What would we want the outputs to look like?



Multi-Layer Perceptron: Multi-output

Multiple output nodes possible

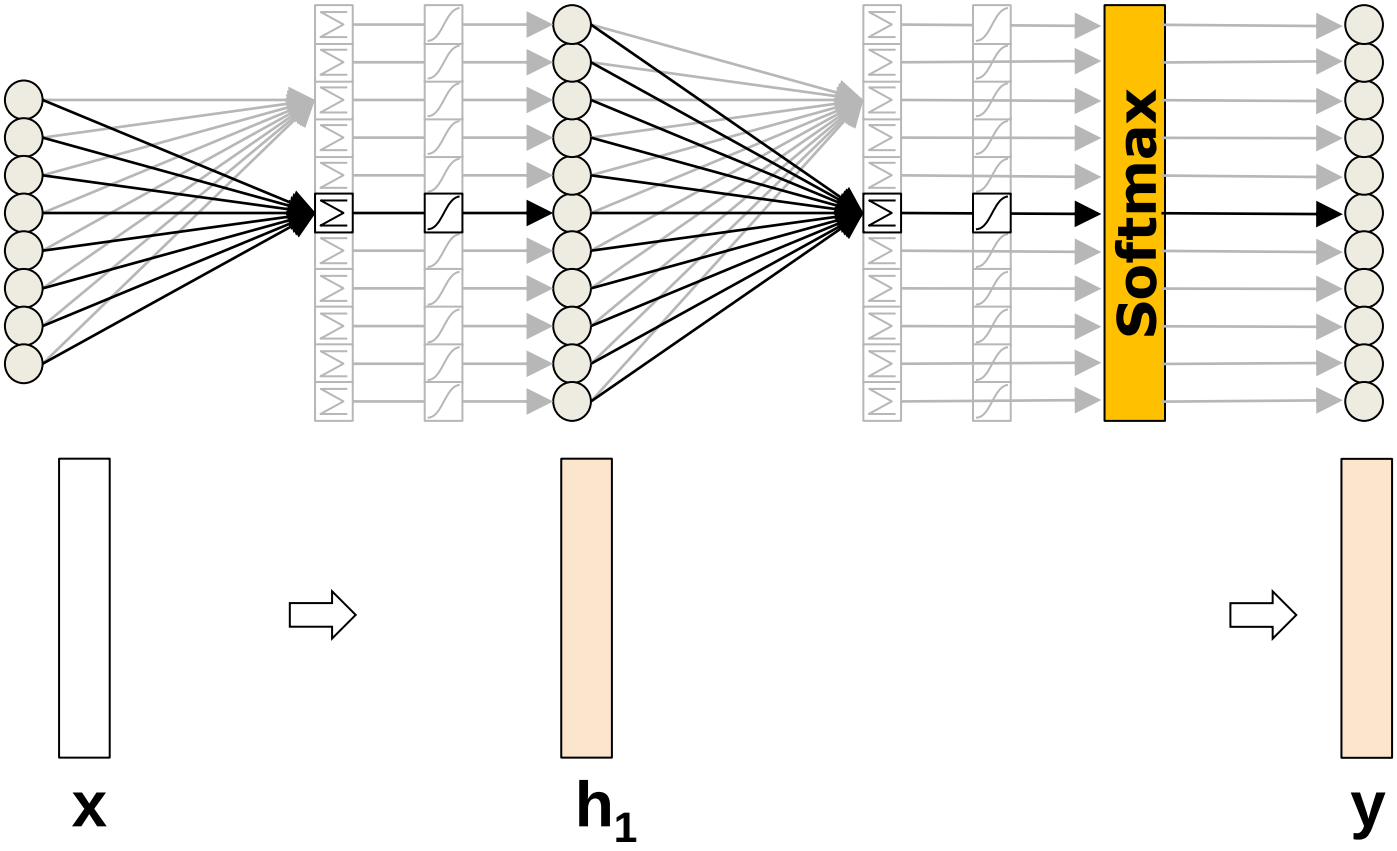
Softmax function



Multi-Layer Perceptron: Multi-output

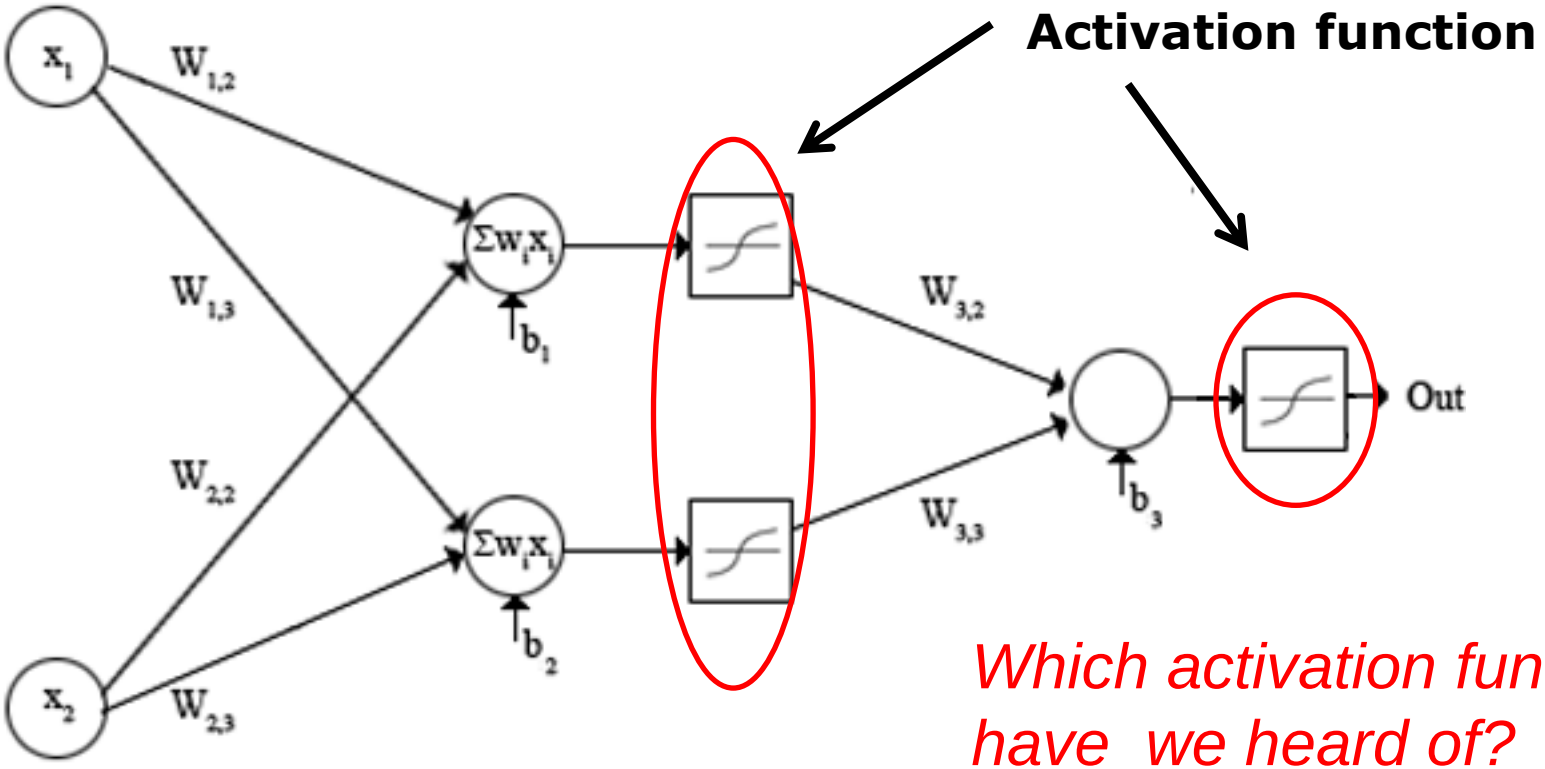
- Softmax: converts to probability distribution: (values sum up to 1; skewed to favour higherinputs)

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$



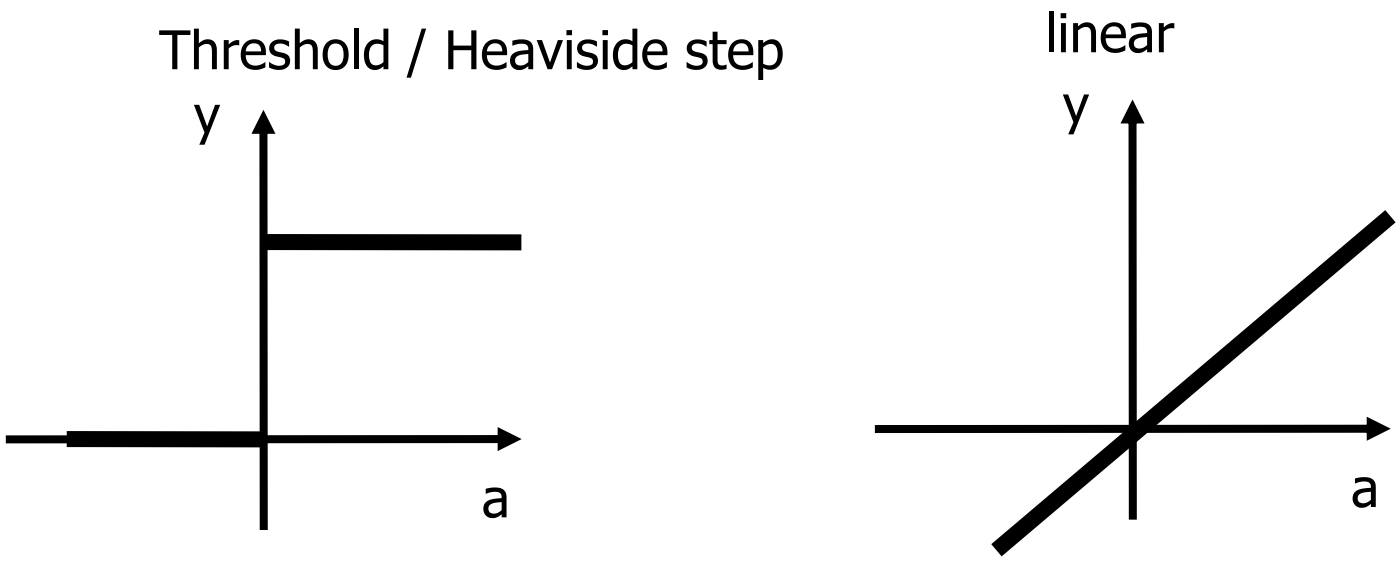
MLP: Activation Function

- Activation function (potentially) applied after each neuron computes output (“fires”)



Which activation functions have we heard of?

MLP: Activation Function

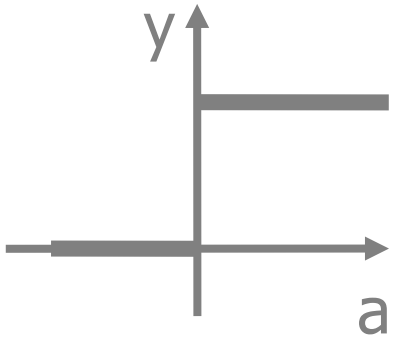


- If all neurons have linear activation function
  ➔ MLP could be reduced to simple Perceptron

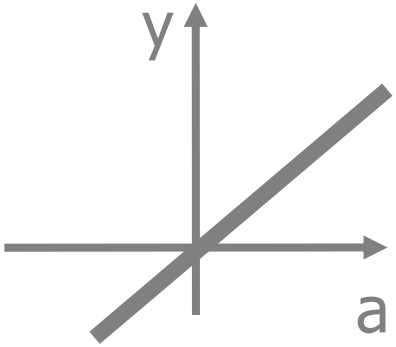
- Activation functions chosen to be not linear
  ➔ can solve more complex problems

Activation Functions

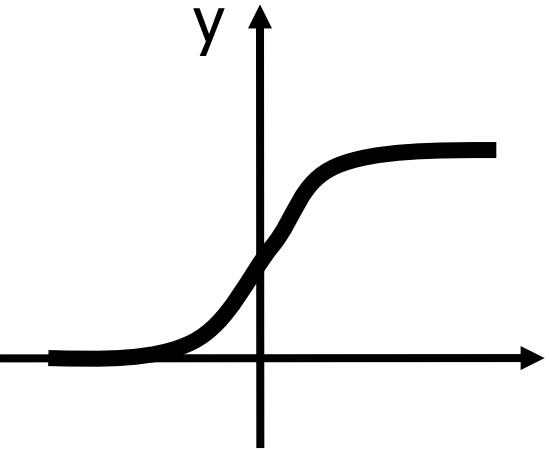
Threshold / Heaviside step



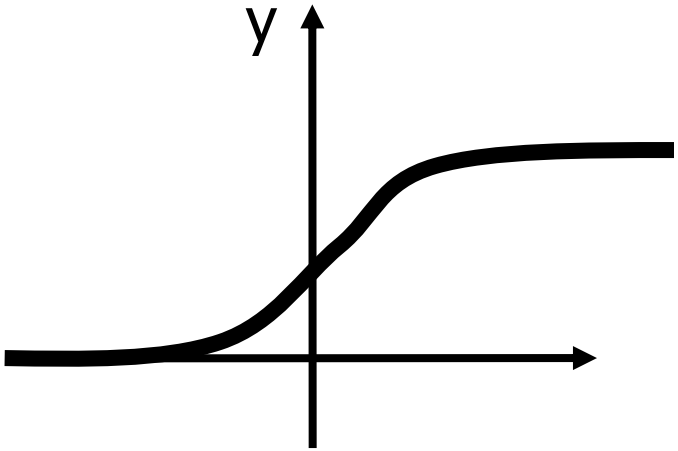
linear



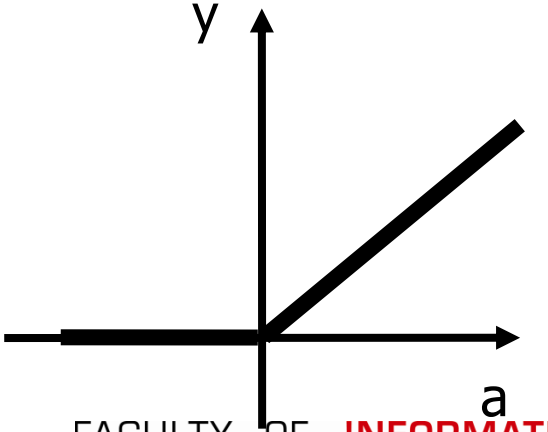
sigmoid



tanh

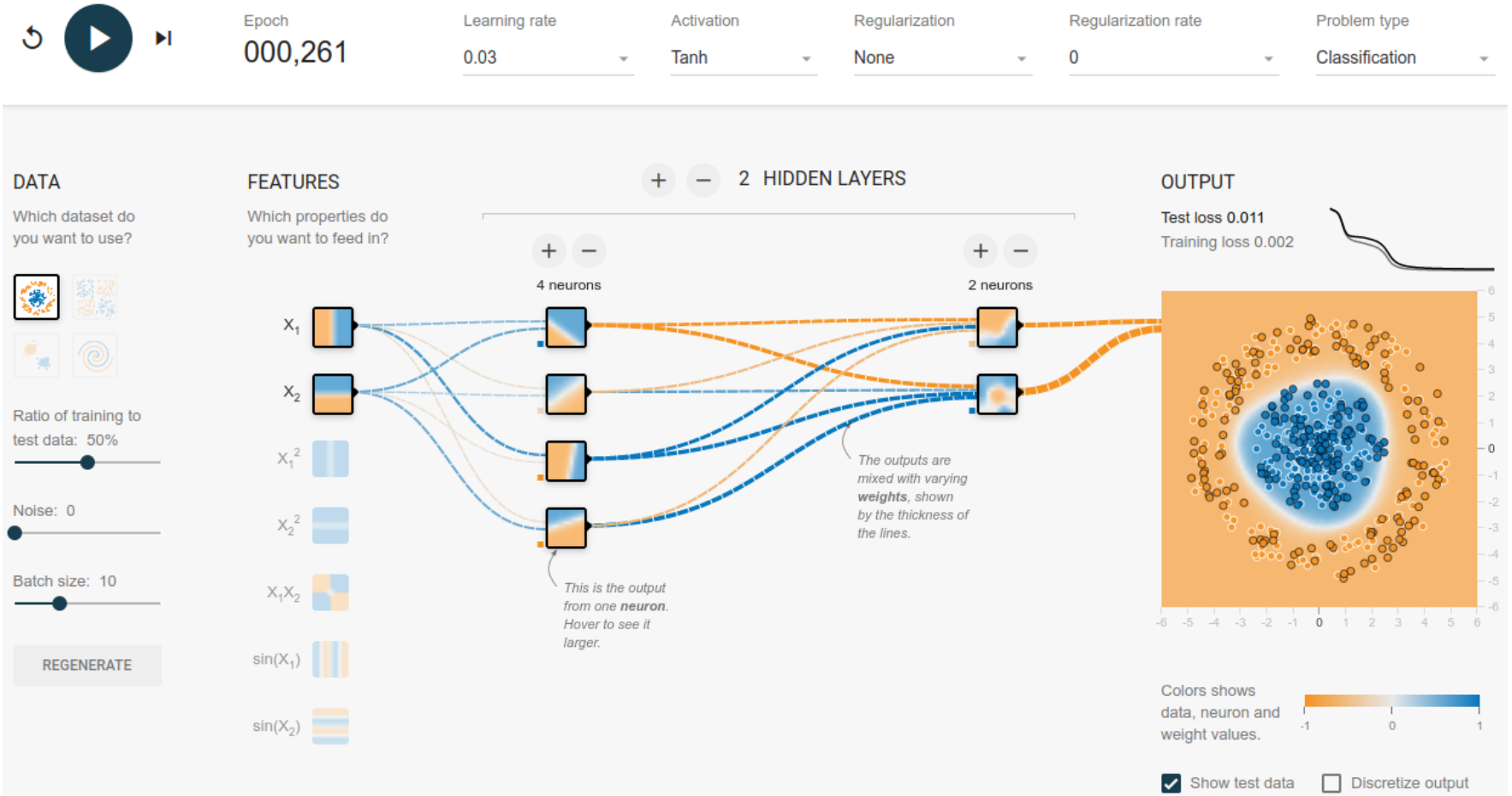


RELU



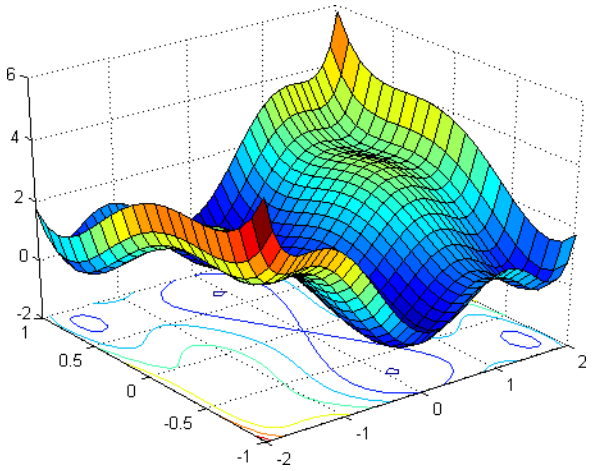
Demo

- Interactive demo:
  https://playground.tensorflow.org



Multi-Layer Perceptron: Training

- *Learning? Difference to Perceptron?*
- Goal: find values for parameters (weights & biases) that are optimal for the task to perform
  - Proxy: find values for parameters that perform well on the training (& validation) test
- Think of a Perceptron for a 1-dimensional dataset
  - How many parameters do we need to learn?



» What do the x, y, and z axis represent?

» x-axis: weight w_1

» y-axis: bias b

» z-axis: ?

» The error of the model with given w_1/b

Multi-Layer Perceptron: Training

- Goal: find values for parameters (weights & biases) that are optimal for the task to perform
  - Proxy: find values for parameters that perform well on the training (& validation) test
- Can not try all possible values for each weight & bias to find the optimum; no analytical solution feasible
- Alternative: start with random solution, and try to improve that using a heuristic, e.g. gradient descent
  - Gradient descent: step-wise adaptation of weights
    - New weights that reduce the error (loss function) of the network
      - Loss function quantifies the divergence between predicted and actual outputs
    - Repeated until loss is minimised / other stopping criterion

Multi-Layer Perceptron: Training

- Changing weights & biases after each iteration (i.e. one input sample presented) via gradient descent
  - Based on amount of errors in the output
    - Minimize the loss between the prediction and groundtruth
      - Still similar to Perceptron?
  - Carried out through *back propagation & gradient descent*
    - Starting at the output layer
      - Adapt weights by *going backwards* in the network
      - Change each weight to *reduce weight's contribution to the loss*
  - Error: evaluate $\hat{y} = f(x)$ ($\hat{y}$: predicted class, f: MLP)
    - Error e(n) in output node for sample *n*
      - Computation of error depends on the type of output
      - Different loss functions for single-value vs. multi-dimensional output
      - Mean squared error (MSE), cross-entropy, Hinge, ...

Multi-Layer Perceptron: Training

- *How do we know weight's contribution to error?*
- Change weights:

$$\Delta_{w_i} = -\alpha \frac{\partial e(n)}{\partial x(n)} y_i(n)$$

$$w_i^{t+1} = w_i^{t} - \alpha \frac{\partial e(n)}{\partial x(n)} y_i(n)$$

- y_i: output of the previous neuron
- α: learning rate

- Change depends on previous layer
  - ➔ start backwards from (output – 1) layer
  - ➔ Until we reach the first layer

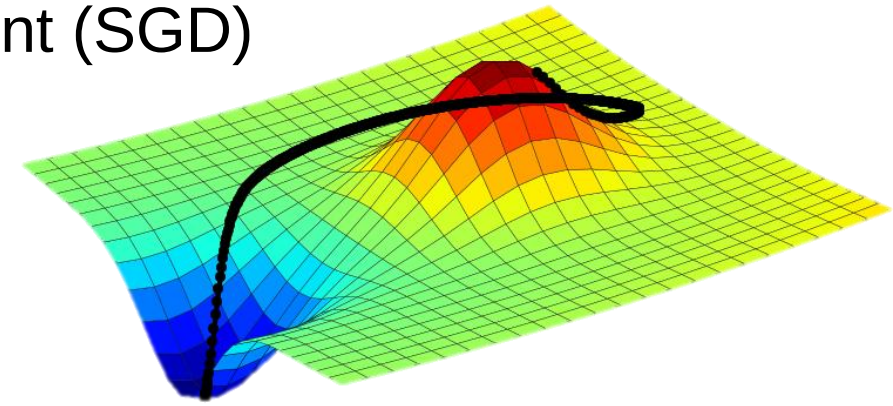
Multi-Layer Perceptron: Training

- Weight adaption determined by “loss“ **L**
- Loss computed e.g. as Cross-Entropy

$$Loss(\hat{y}, y, W) = -y \ln \hat{y} - (1 - y) \ln (1 - \hat{y})$$

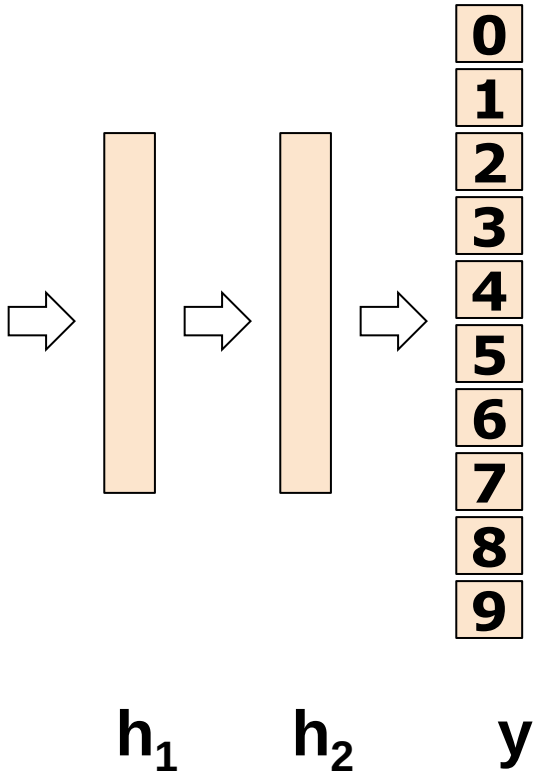
- Adaptation: $\frac{\partial \mathbf{L}}{\partial \boldsymbol{\theta}}$ (or $\frac{\partial \mathbf{L}}{\partial \mathbf{W}}$)

- Adaption of weights is a form of gradient decent
  - Stochastic Gradient Descent (SGD) (aka incremental GD):
  - ***Iterative*** method: take small step in direction of negative gradient

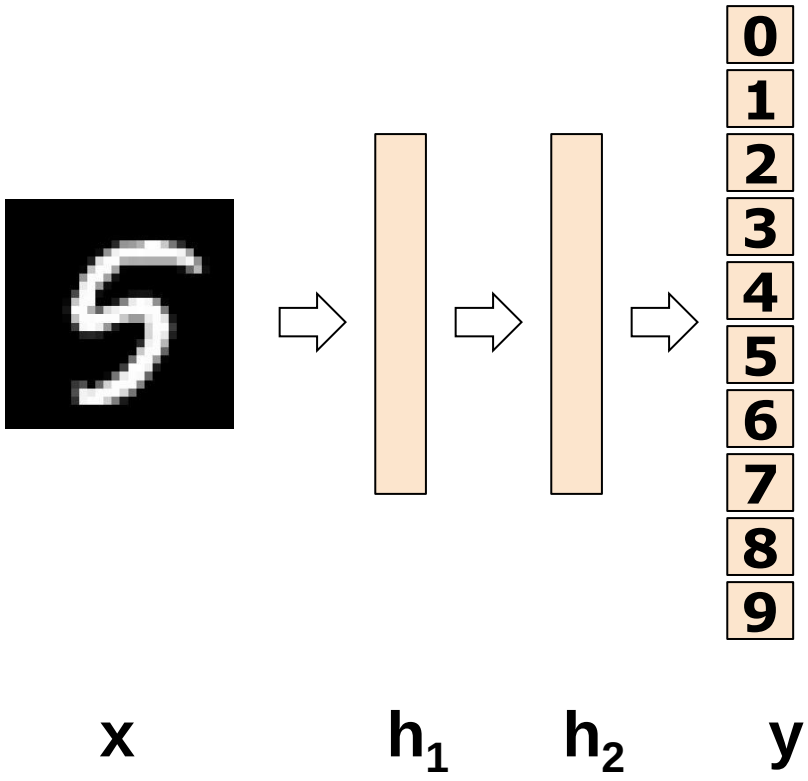


Multi-Layer Perceptron: Training

1. Initialize learnable weights randomly



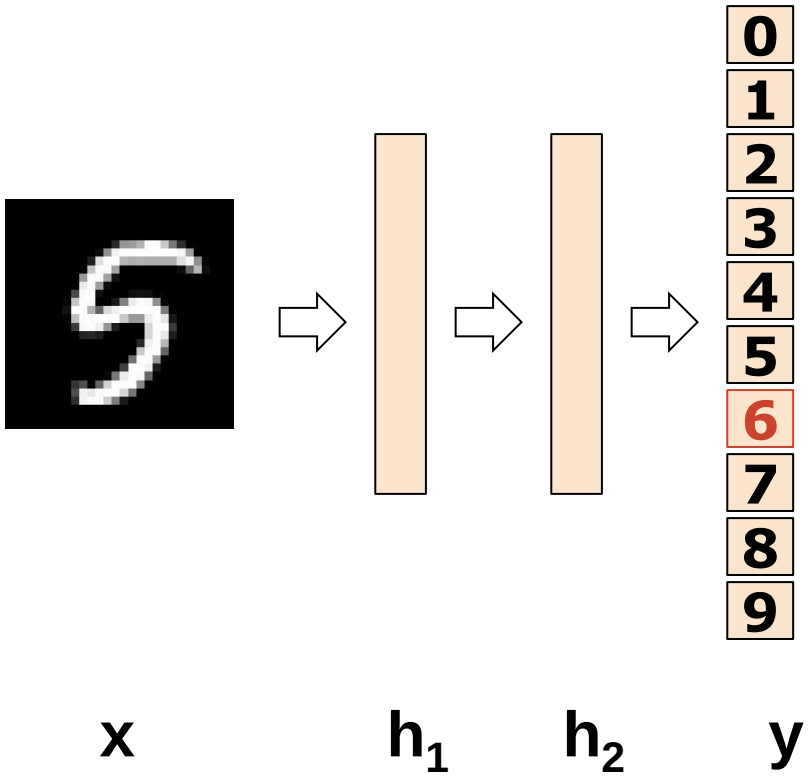
Multi-Layer Perceptron: Training



(maybe not actually the raw image (pixels) itself, but some extracted features)

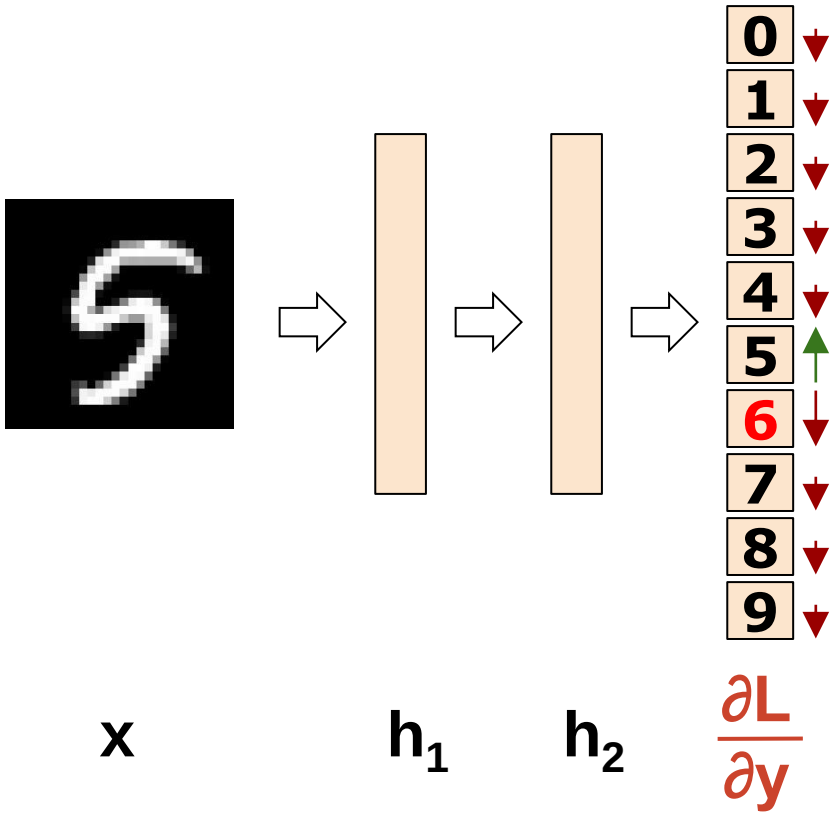
1. Initialize learnable weights randomly
2. Repeat until satisfied:
   a. pick training example

Multi-Layer Perceptron: Training



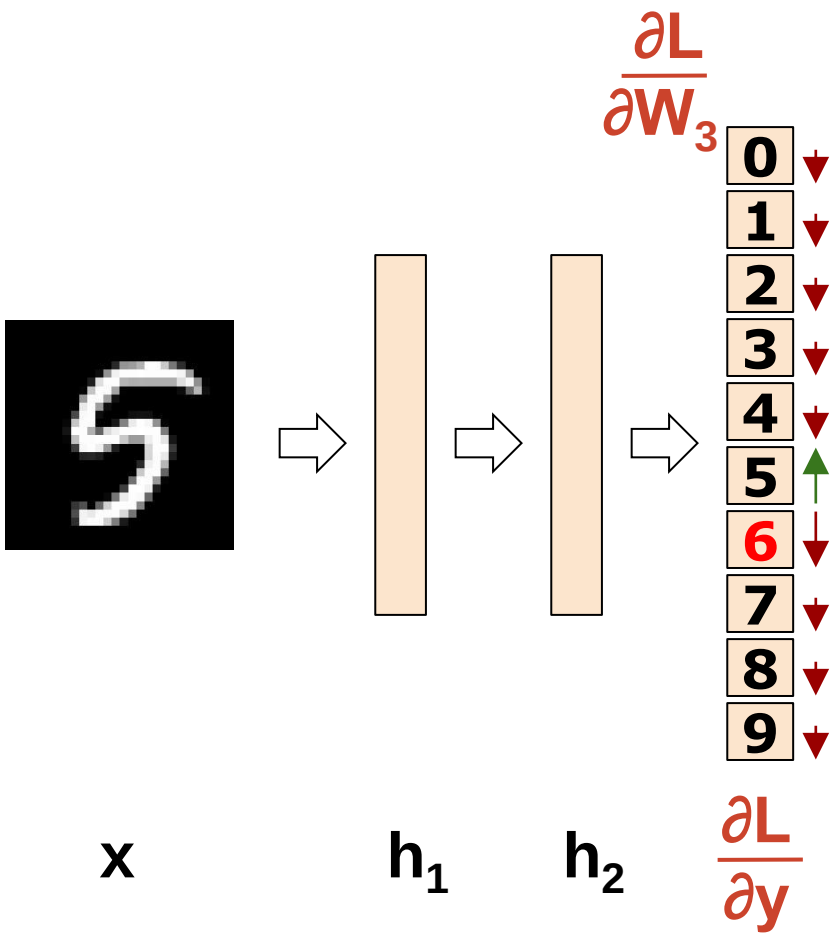
1. Initialize learnable weights randomly
2. Repeat until satisfied:
   a. pick training example
   b. compute output

Multi-Layer Perceptron: Training



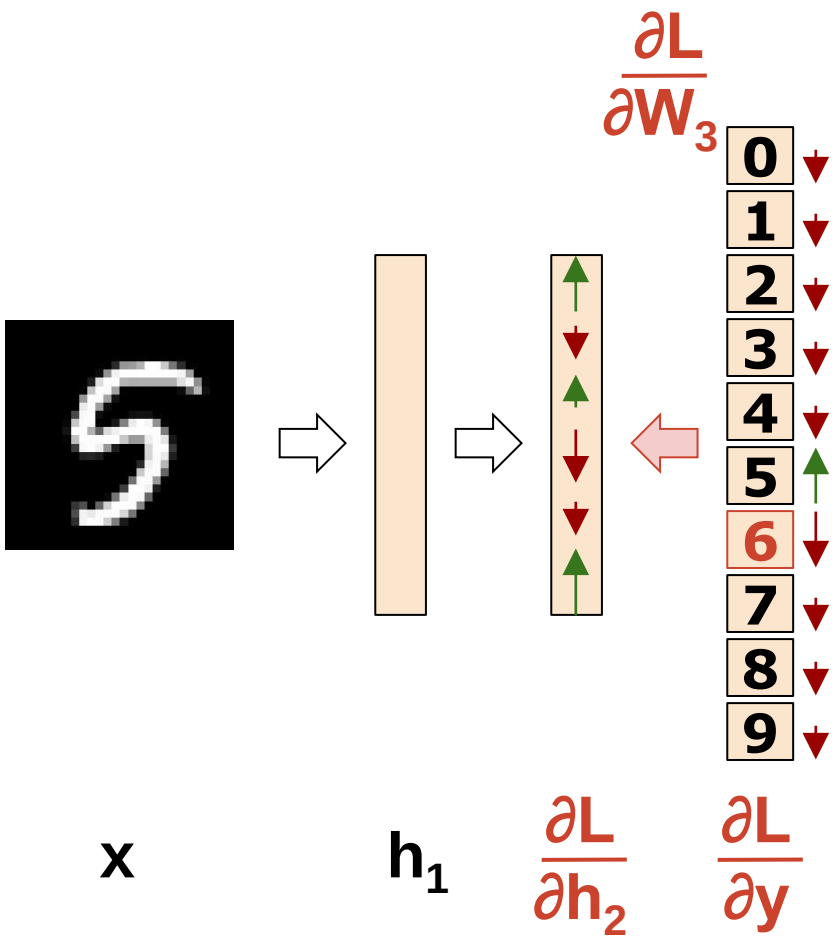
1. Initialize learnable weights randomly
2. Repeat until satisfied:
   a. pick training example
   b. compute output
   c. compute gradient of loss wrt. output

Multi-Layer Perceptron: Training



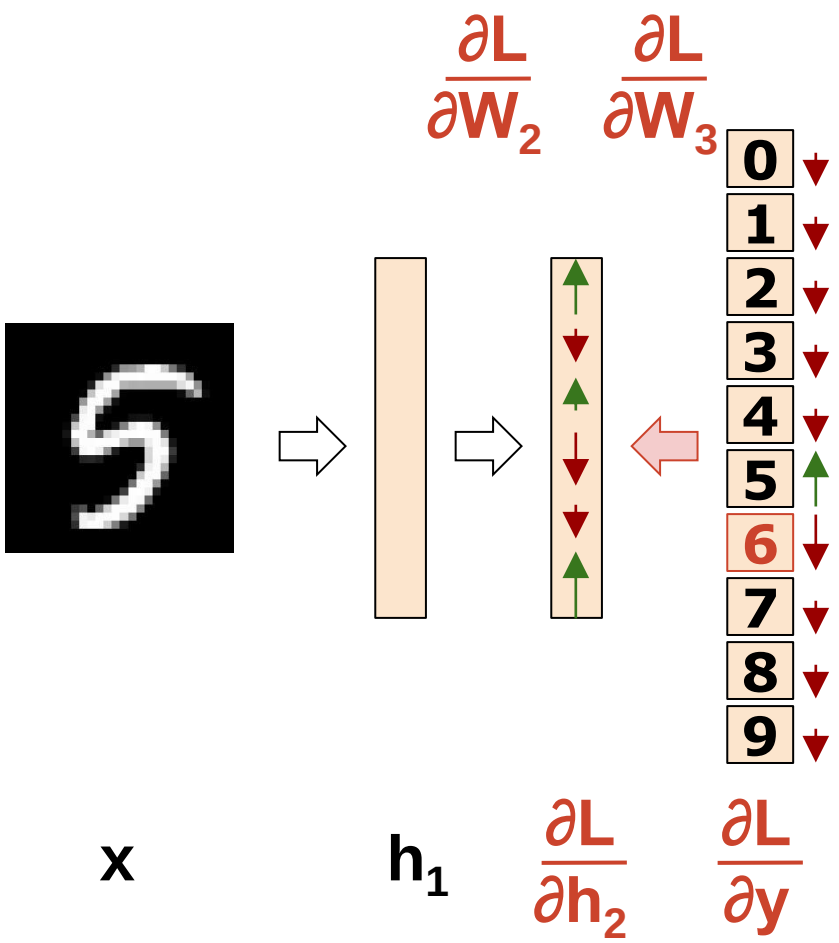
1. Initialize learnable weights randomly
2. Repeat until satisfied:
   a. pick training example
   b. compute output
   c. compute gradient of loss wrt. output
   d. backpropagate loss, computing gradients wrt. learnable weights

Multi-Layer Perceptron: Training



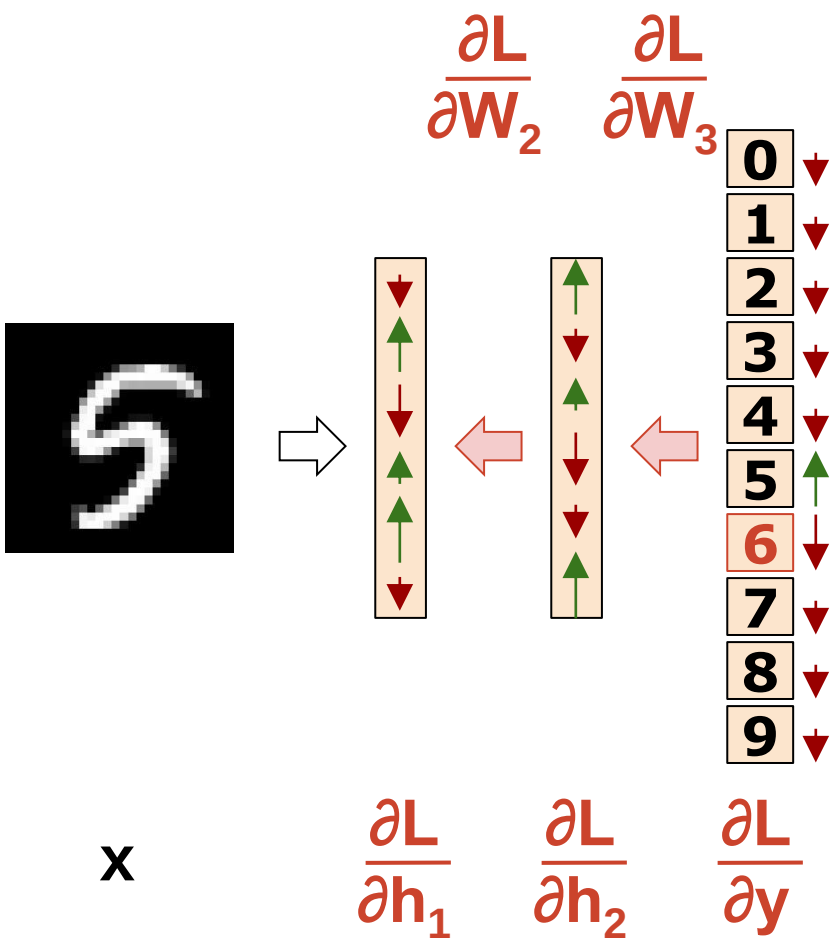
1. Initialize learnable weights randomly
2. Repeat until satisfied:
   a. pick training example
   b. compute output
   c. compute gradient of loss wrt. output
   d. backpropagate loss, computing gradients wrt. learnable weights

Multi-Layer Perceptron: Training



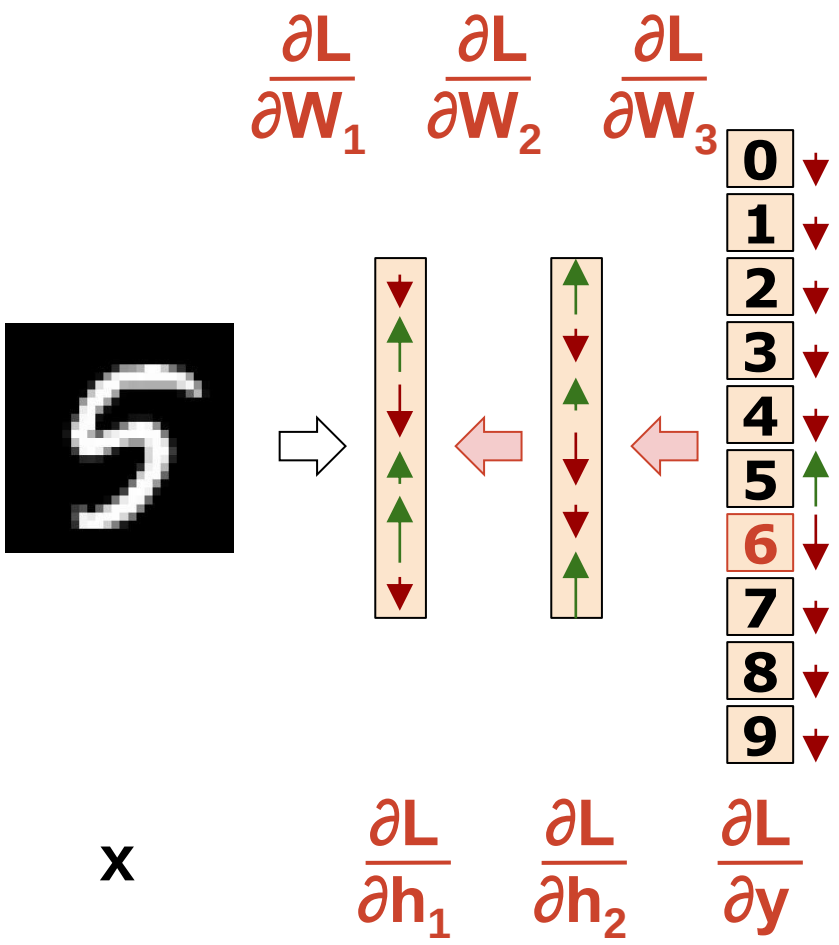
1. Initialize learnable weights randomly
2. Repeat until satisfied:
   a. pick training example
   b. compute output
   c. compute gradient of loss wrt. output
   d. backpropagate loss, computing gradients wrt. learnable weights

Multi-Layer Perceptron: Training



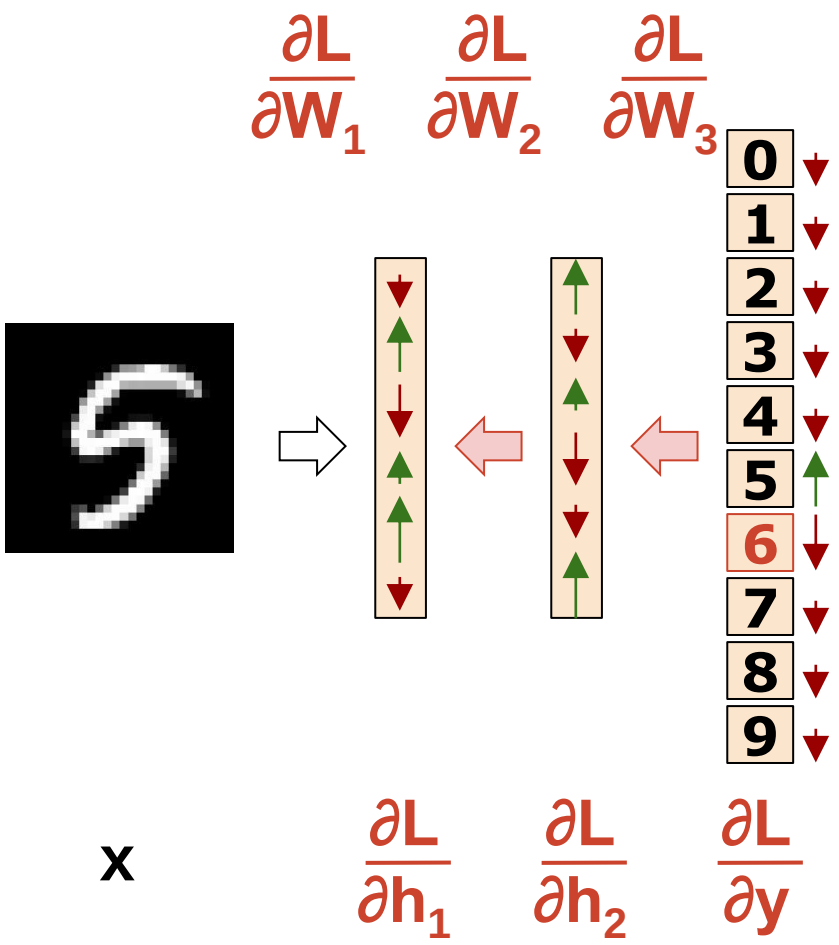
1. Initialize learnable weights randomly
2. Repeat until satisfied:
   a. pick training example
   b. compute output
   c. compute gradient of loss wrt. output
   d. backpropagate loss, computing gradients wrt. learnable weights

Multi-Layer Perceptron: Training



1. Initialize learnable weights randomly
2. Repeat until satisfied:
   a. pick training example
   b. compute output
   c. compute gradient of loss wrt. output
   d. backpropagate loss, computing gradients wrt. learnable weights

Multi-Layer Perceptron: Training

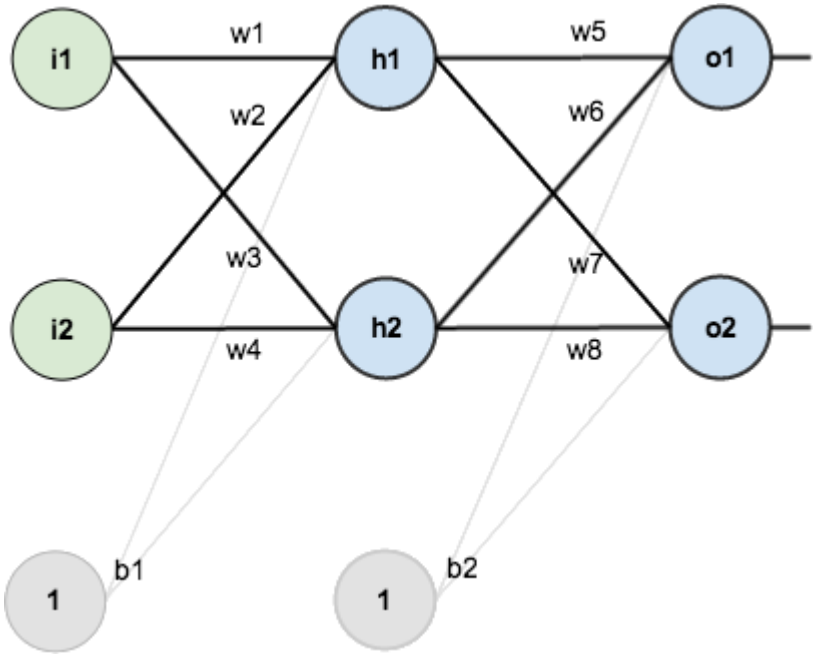


1. Initialize learnable weights randomly
2. Repeat until satisfied:
   a. pick training example
   b. compute output
   c. compute gradient of loss wrt. output
   d. backpropagate loss, computing gradients wrt. learnable weights
   e. update weights in direction of negative gradient (to reduce loss for this example)

MLP Training: example

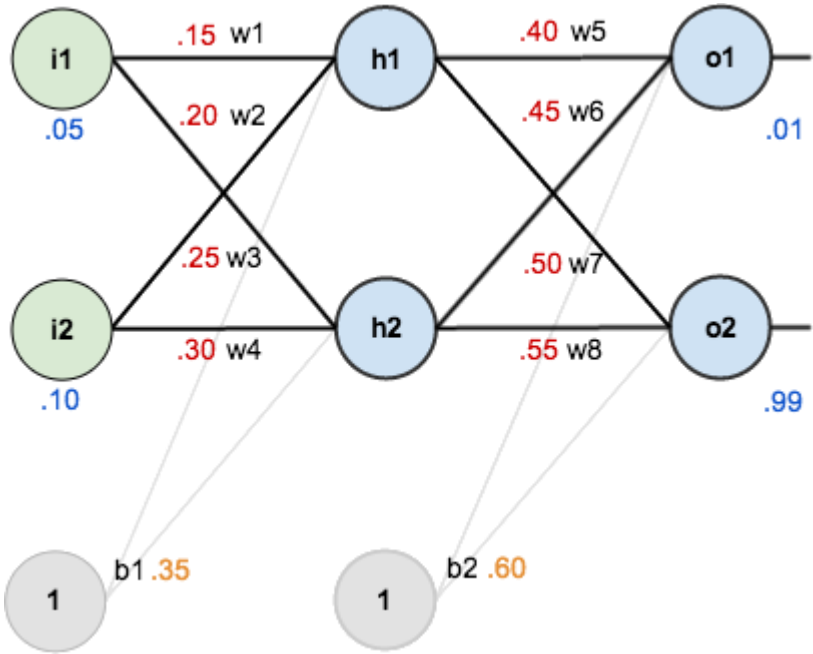
- Simple MLP
  - 2 inputs
  - 1 hidden layer with 2 nodes
  - 2 output nodes

- *How many weights to learn?*



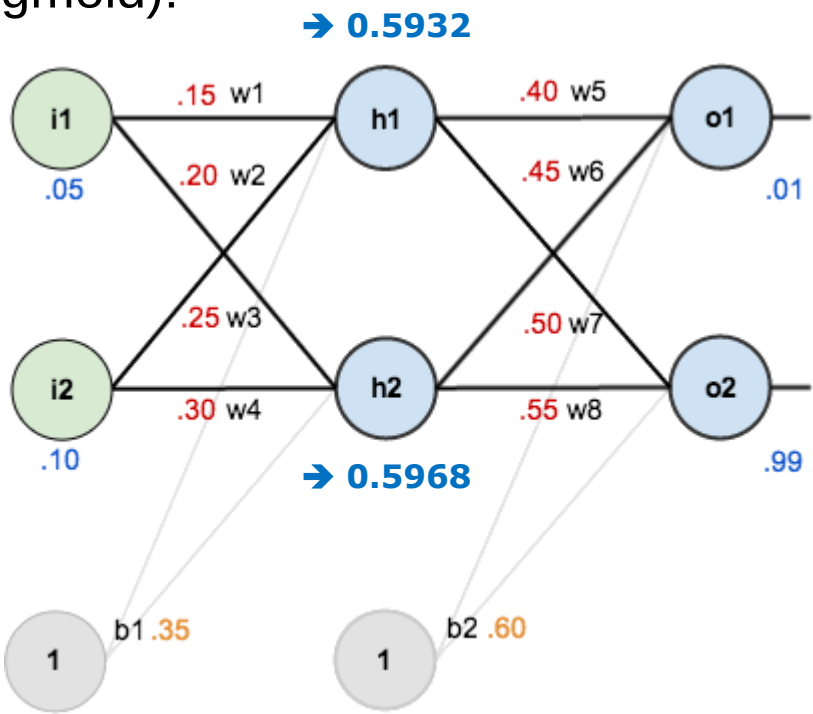
MLP Training: example

- Example (random initialisation) values for
  - Weights: Layer1 (hidden): [0.15, 0.20, 0.25, 0.30, 0.35]
    Layer2 (output): [0.40, 0.45, 0.50, 0.55, 0.60]
  - Data:
    - 1 Input vector:
      - X = [0.05, 0.10]
    - **Expected (actual)** outputs:
      - y = [0.01, 0.99]



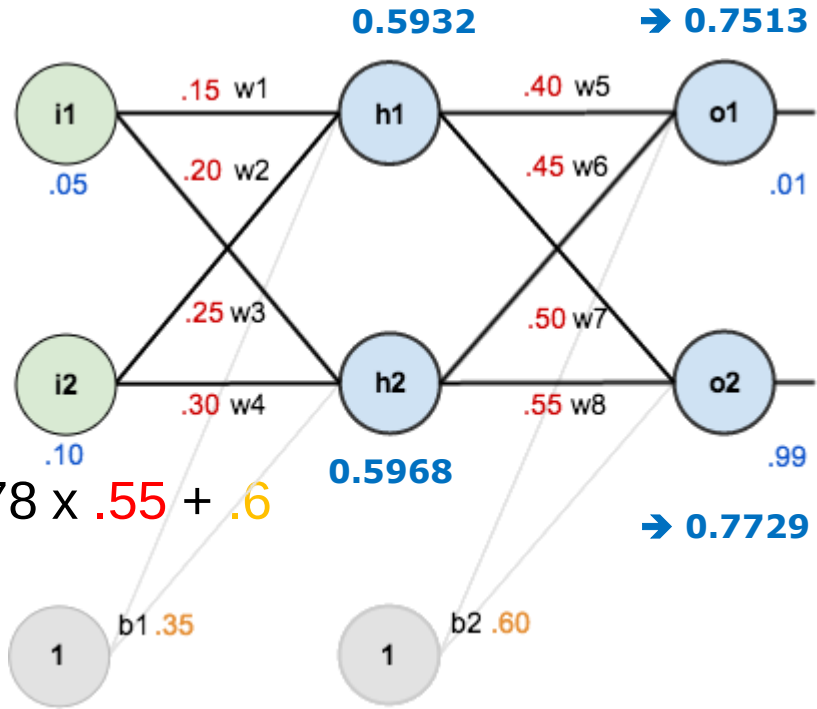
MLP Training: example

- **Feed Forward** to compute ***predicted*** output
  - h1:
    - Compute activation: activation(h1) =
      = i_1 x w_1 + i_2 x w_2 + b_1=
      = .05 x .15 + .10 x .20 + .35 = 0.3775
    - Pass through activation function (sigmoid):
      out(h1) = $\frac{1}{1 + e^{-activation(h1)}}$ =
      = 0.593269992
  - h2:
    - activation(h2) =
      = i_1 x w_3 + i_2 x w_4 + b1=
      = .05 x .25 + .10 x .30 + .35 =
      = 0.3925
    - out(h2) = 0.596884378



MLP Training: example

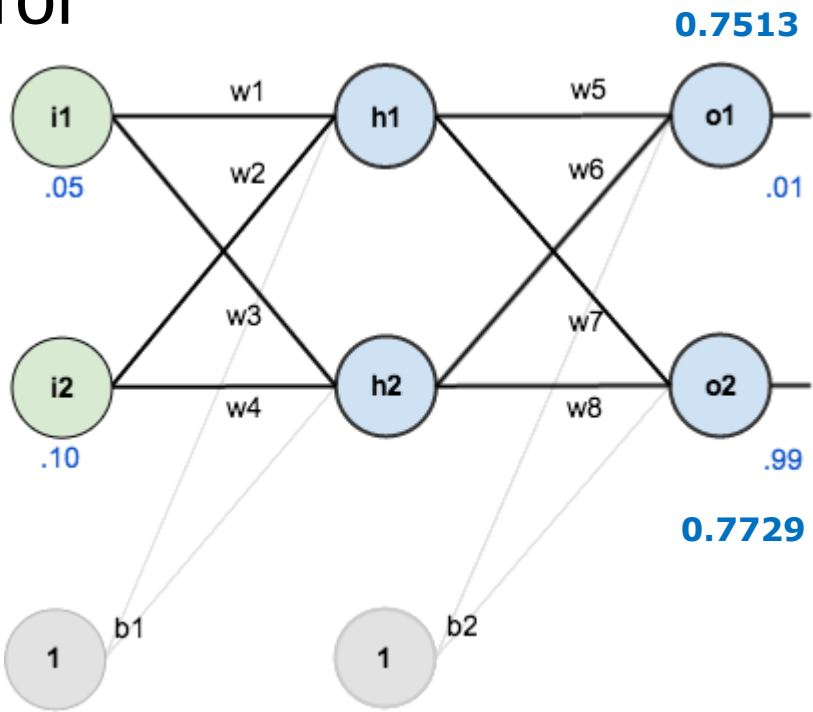
- **Feed Forward** to compute ***predicted*** output
  - o1:
    - Compute activation (net): activation(o1) =
      = h_1 x w_5 + h_2 x w6 + b_2=
      = 0.593269992 x .40 + 0.596884378 x .45 + .6 = 1.105905967
    - Pass through function (sigmoid):
      out(o1) = $\frac{1}{1 + e^{-activation\ (h1)}}$ =
      = 0.75136507
  - o2:
    - activation(o2) =
      = h_1 x w_7 + h_2 * w_8 + b_2=
      = 0.593269992 x .50 + 0.596884378 x .55 + .6
      = 1.224921404
    - out(o2) = 0.772928465



MLP Training: example

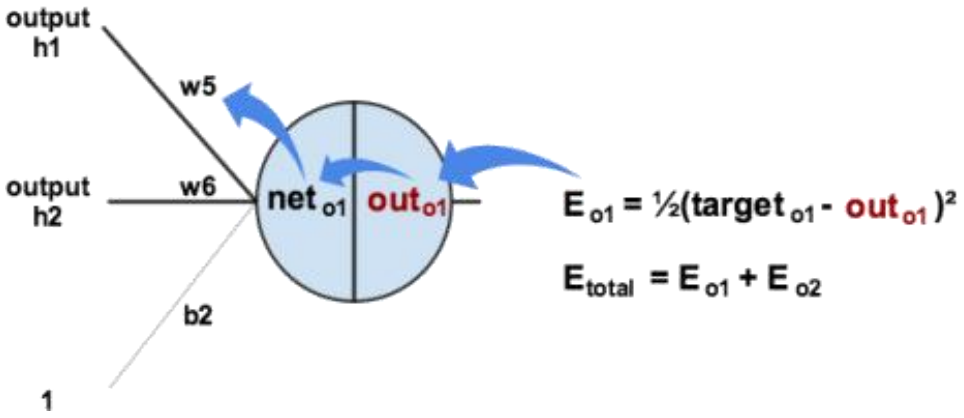
- **Compute loss**
  - o1: expected (target): 0.01, predicted (actual): 0.75136507
  - o2: expected: 0.99, predicted: 0.772928465

- Loss function, e.g. squared error
  - $\boldsymbol{E} = \Sigma \, ½ \, (\text{target} - \text{output})^2$
  - $\boldsymbol{E} = E_{o1} + E_{o2}$
  - $\boldsymbol{E}_{o1} = ½ \, (\text{target}_{o1} - \text{output}_{o2})^2 =$
    $= ½ \, (0.01 - 0.75136507)^2$
    $= 0.274811083$
  - $\boldsymbol{E}_{o2} = 0.023560026$
  - $\boldsymbol{E} = 0.298371109$



MLP Training: example

- **Adapt weights**
  - E.g. weight w_5
    - How much does a change in w_5 affect the error? $\frac{\partial E}{\partial w_5}$
    - Via chain rule: $\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$



MLP Training: example

- **Adapt weights**

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

$$E_{total} = \frac{1}{2}(target_{o1} - out_{o1})^2 + \frac{1}{2}(target_{o1} - out_{o1})^2$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = 2 * \frac{1}{2}(target_{o1} - out_{o1})^{2-1} * -1 + 0$$

$$= -(target_{o1} - out_{o1}) = -(0.01 - 0.75136507) = 0.74136507$$

$$out_{o1} = \frac{1}{1 + e^{-net_{01}}}$$

$$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1} * (1 - out_{o1}) = 0.75136507\ (1 - 0.75136507) = 0.186815602$$

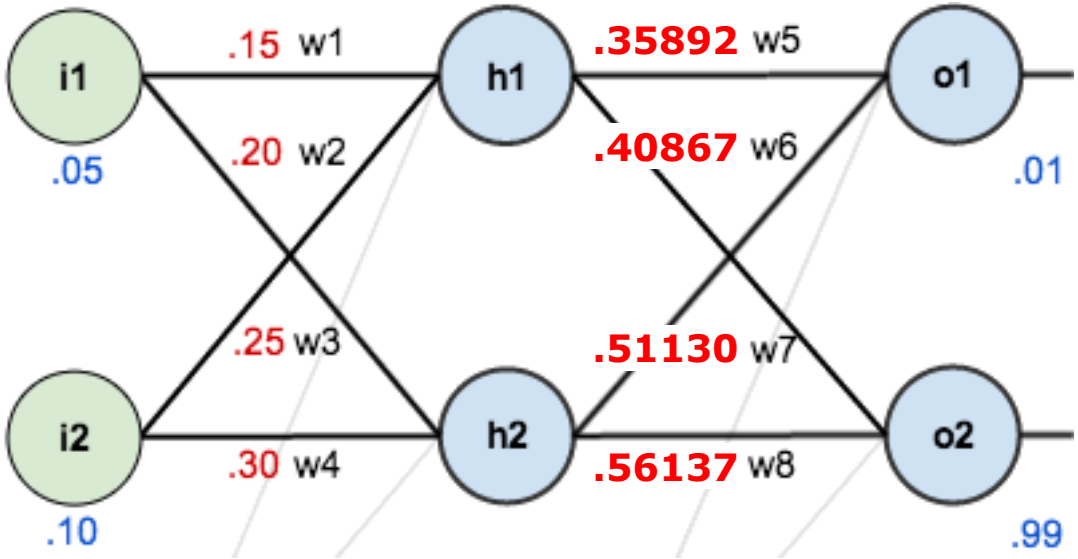
$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial w_5} = 1 * out_{h1} * w_5^{(1-1)} + 0 + 0 = out_{h1} = 0.593269992$$

$$\frac{\partial E_{total}}{\partial w_5} = 0.74136507 * 0.186815602 * 0.593269992 = 0.082167041$$

MLP Training: example

- **Adapt weights:** decrease error contribution by w_5
  - $w_5^{t+1} = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 * 0.082167041 = 0.35891648$
  - $w_6^{t+1} = 0.408666186$
  - $w_7^{t+1} = 0.511301270$
  - $w_8^{t+1} = 0.561370121$



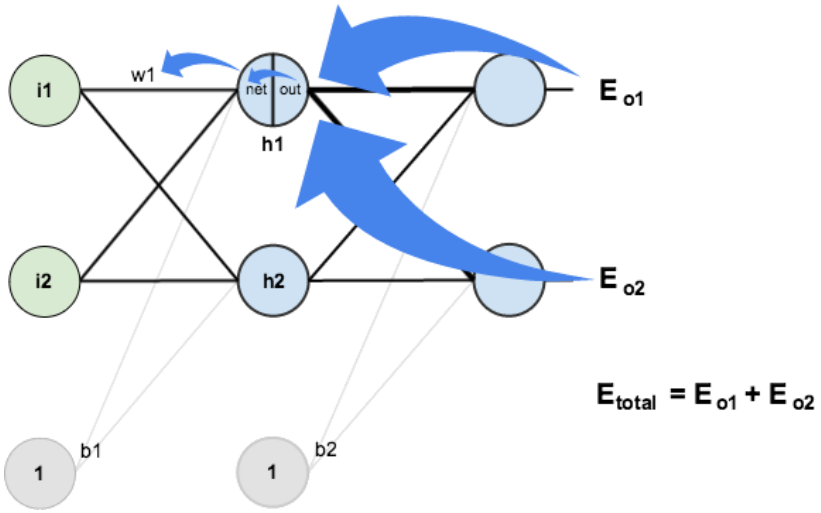
- Next step?
  - Adapt weights on hidden layer!
    - Nodes h_1 and h_2

MLP Training: example

- **Adapt weights on hidden layer**
  - E.g. w_1:

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$



  - NB: output of h_1 affects both o_1 and o_2!
    - Thus more chaining

MLP Training: example

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

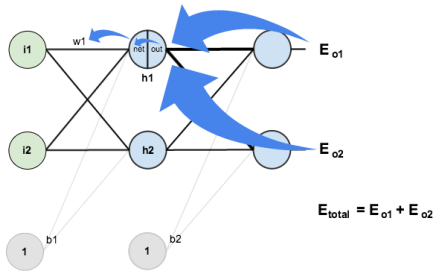
- **Adapt weights w_1:**

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} + \frac{\partial net_{o1}}{\partial out_{h1}}$$

$$\frac{\partial E_{o1}}{\partial net_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}} + \frac{\partial out_{o1}}{\partial net_{o1}} = 0.74136507 * 0.186815602 = 0.138498562$$

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial out_{h1}} = w_5 = 0.4$$



$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}} = 0.138498562 * 0.4 = 0.0036350306$$

$$\frac{\partial E_{o2}}{\partial out_{h1}} = -0.019049119$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}} = 0.055399425 + -0.019049119 = 0.036350306$$

- **Finally:**

$$\frac{\partial E_{total}}{\partial w_1} = 0.036350306 * 0.241300709 * 0.05 = 0.000438568$$

MLP Training: example

- Adapt weights $w_1 – w_4$
  - $w_1^{t+1} = w_1 - \eta * \frac{\partial E_{total}}{\partial w_1} = 0.15 - 0.5 * 0.000438568 = 0.149780716$
  - $w_2^{t+1} = 0.19956143$
  - $w_3^{t+1} = 0.24975114$
  - $w_4^{t+1} = 0.29950229$

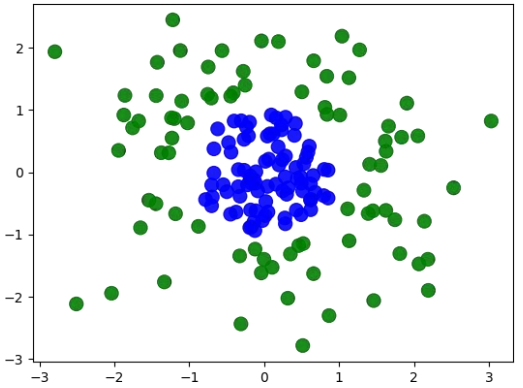
- Feeding sample forward again
  - Error: 0.291027924 (was: 0.298371109)
    - *How many steps (with constant change) would it take to reach 0?*
- Next step?
  - Repeat with next input sample ...

Multi-Layer Perceptron: Training

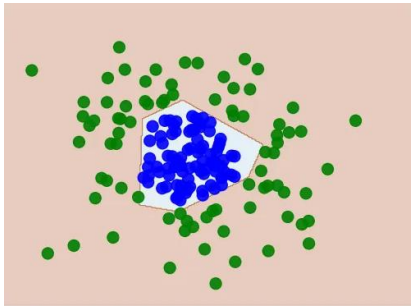
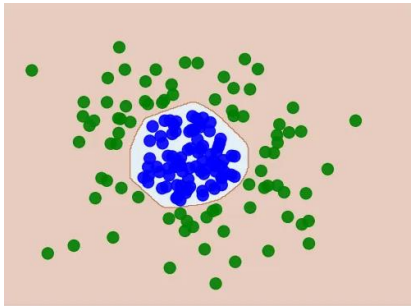
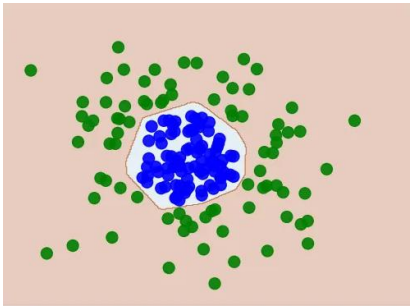
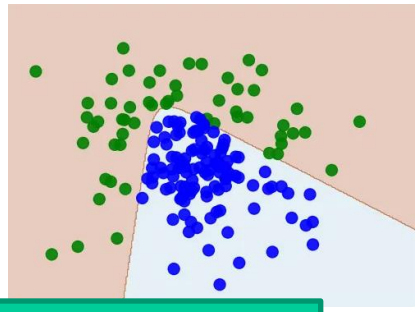
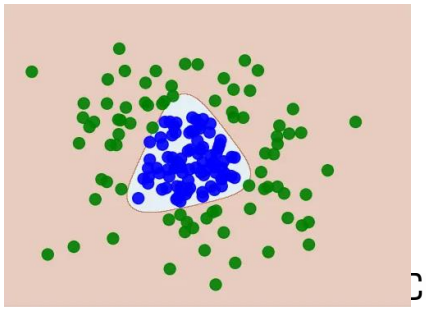
- Several choices to make in terms of training hyper-parameters
  - Architecture (# layers, nodes, ...)
  - Activation functions
  - Learning rate, ...
  - Regularisation parameters
  - Type of gradient descent (full, batch, stochastic..)
  - Issues with (vanishing/exploding) gradient
  - Initialisation (of weights, ...)

What actually happens in an MLP/NN?

- Decision: piece-wise combination of individual decision boundaries
  - Depends on
    - Activation function (shape)
    - Number of neurons (“complexity”)

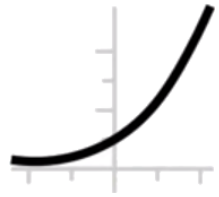


- Dataset: 2 classes, not linear separable
  - RELU
  - Sigmoid

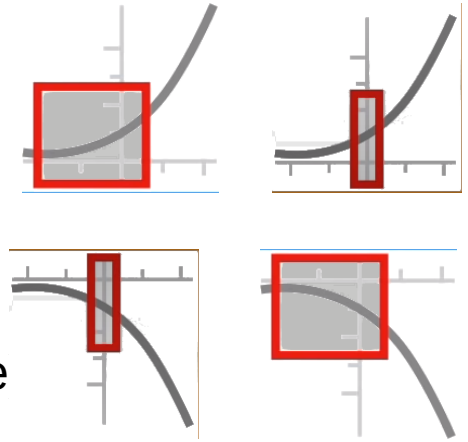






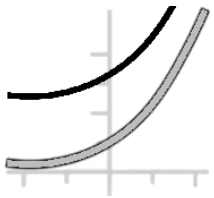
What actually happens in an NN?

- Universal function approximators (e.g. Hornik, 1991)
- Composing individual activation functions
- Softplus activation function: $f(x) = \log(1+e^x)$



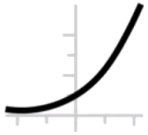
  - Input to the activation function?
  - Previous output ^{x} weight + bias
    - Weights?
      - Set range (slice & stretch initial value range
      - Determine orientation (flip)



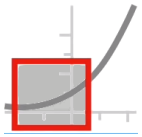


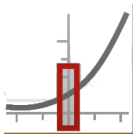
    - Bias?
      - Shifts up / down

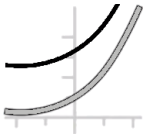
What actually happens in an NN?



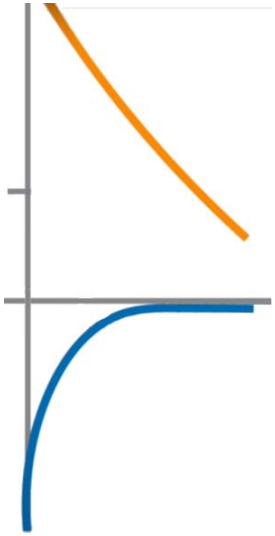
- Composing individual activation functions
- Softplus activation function;
  - Input: previous output x weight + bias
    - Weights:
      - Set range (slice & stretch)
      - Determine orientation (flip)
    - Bias:
      - Shifts up / down

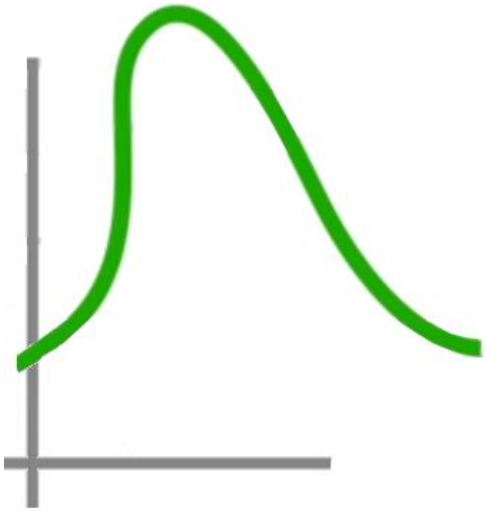


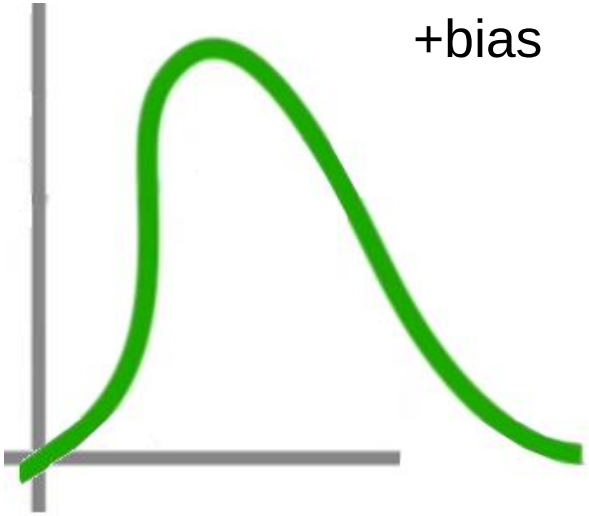




  - Sum function?
    - Composes individual functions



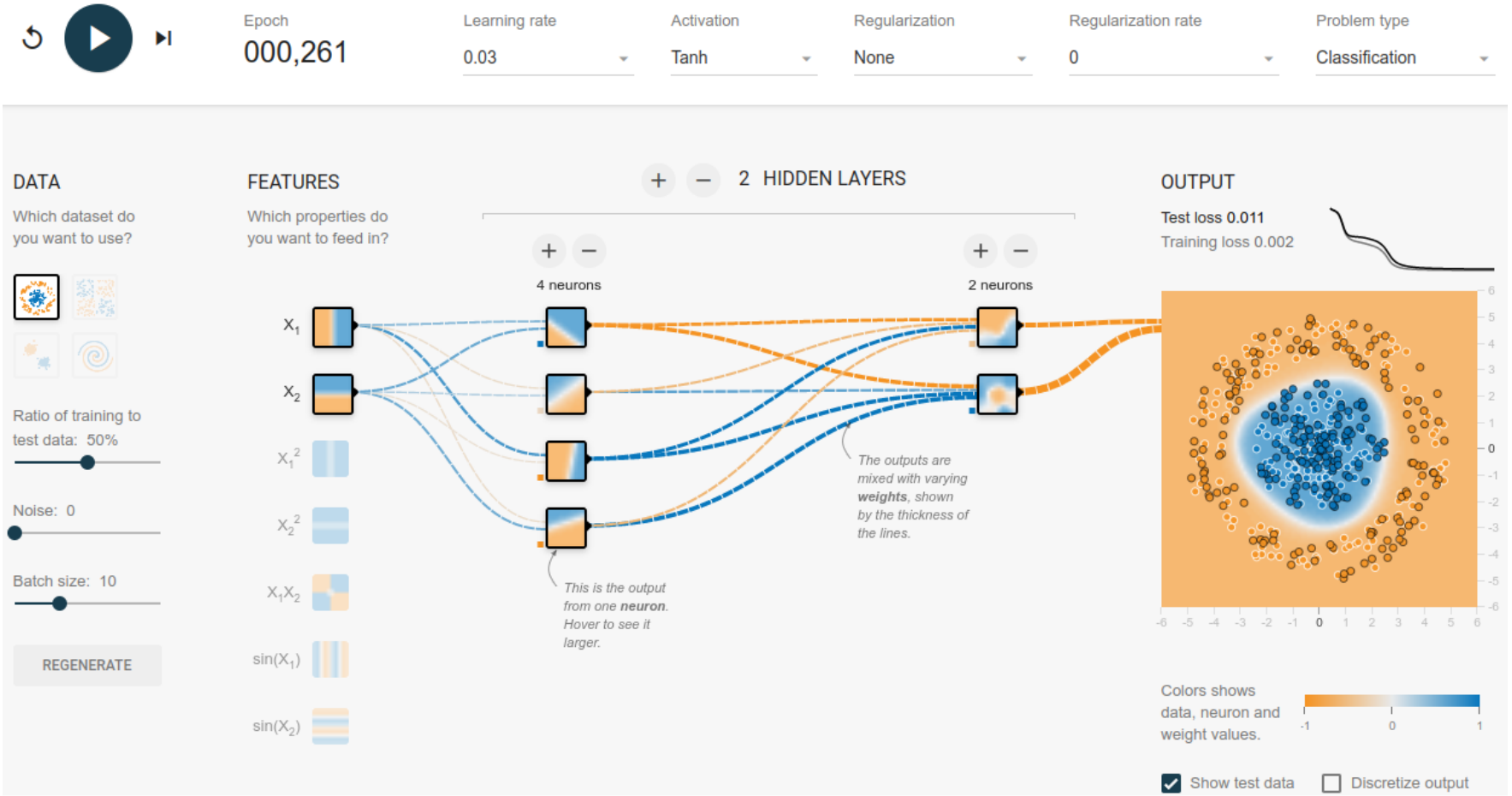




+bias

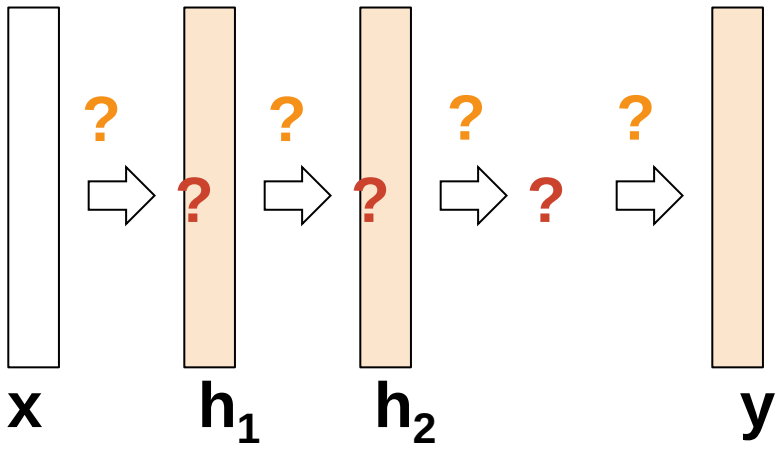
Demo

- Interactive demo:
  https://playground.tensorflow.org



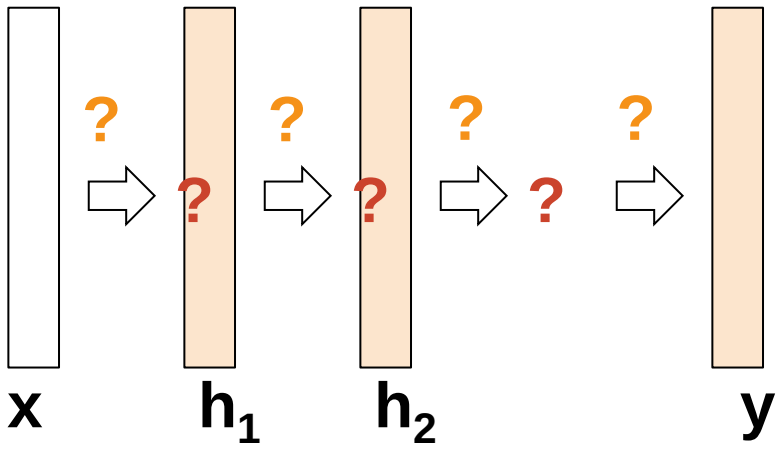
Multi-Layer Perceptron: Design Choices

- How big to make the hidden layers?
- How many hidden layers to choose?
- What kind of activation function to use?



Multi-Layer Perceptron: Design Choices

- Some rules of thumb exist, e.g.
  - Number of hidden layer neurons ~ 2/3 (70-90%) of inputs
  - Number of hidden layer neurons should be less than double (or quadruple?) the number of neurons in input layer
  - More than $\log_2$(#classes) neurons in each layer
  - …
- Number of learnable parameters depends on number of samples! *Why?*



Multi-Layer Perceptron: Training

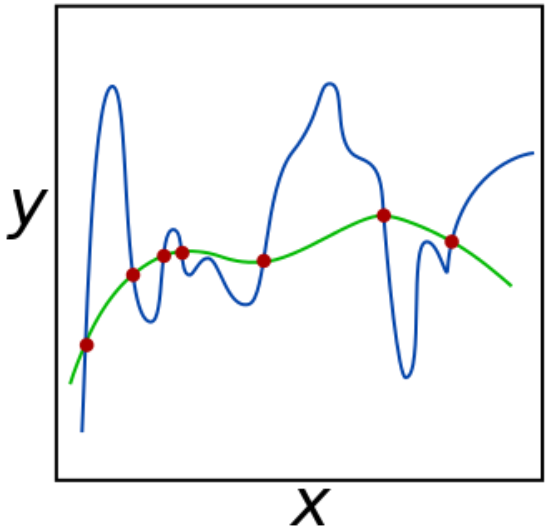
- Previous training: iterative / incremental (online)
  - Process one input sample at a time
    - ➔ adapt weights
    - ➔ process next input sample
  - Stochastic gradient descent

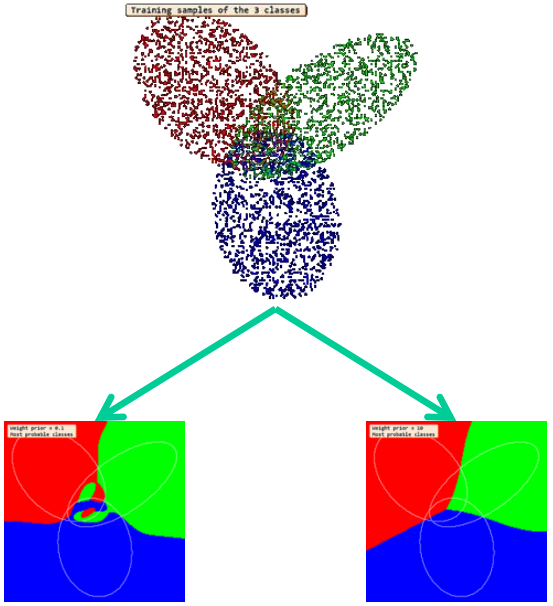
- Alternative: batch training
  - Compute errors for all samples at once
  - Adapt afterwards *(gradient descent)*

- "Middle" approach: "Mini-batch" gradient descent
  - E.g. Using 4,8,16,.. samples at once
    - Often size of a 2^x – *why?*

MLP: Regularisation

- Regularisation aims at preventing overfitted models
  - Overfitted model: too complex model
  - Idea: penalise too complex models





MLP: Regularisation

- Loss function (e.g. Cross-Entropy) often actually used in this form:

$$Loss(\hat{y}, y, W) = -y \ln \hat{y} - (1 - y) \ln (1 - \hat{y})$$

- $\alpha ||W||^2$: regularization term / penalty term
  - Penalises complex models
  - Reduces magnitude of weights

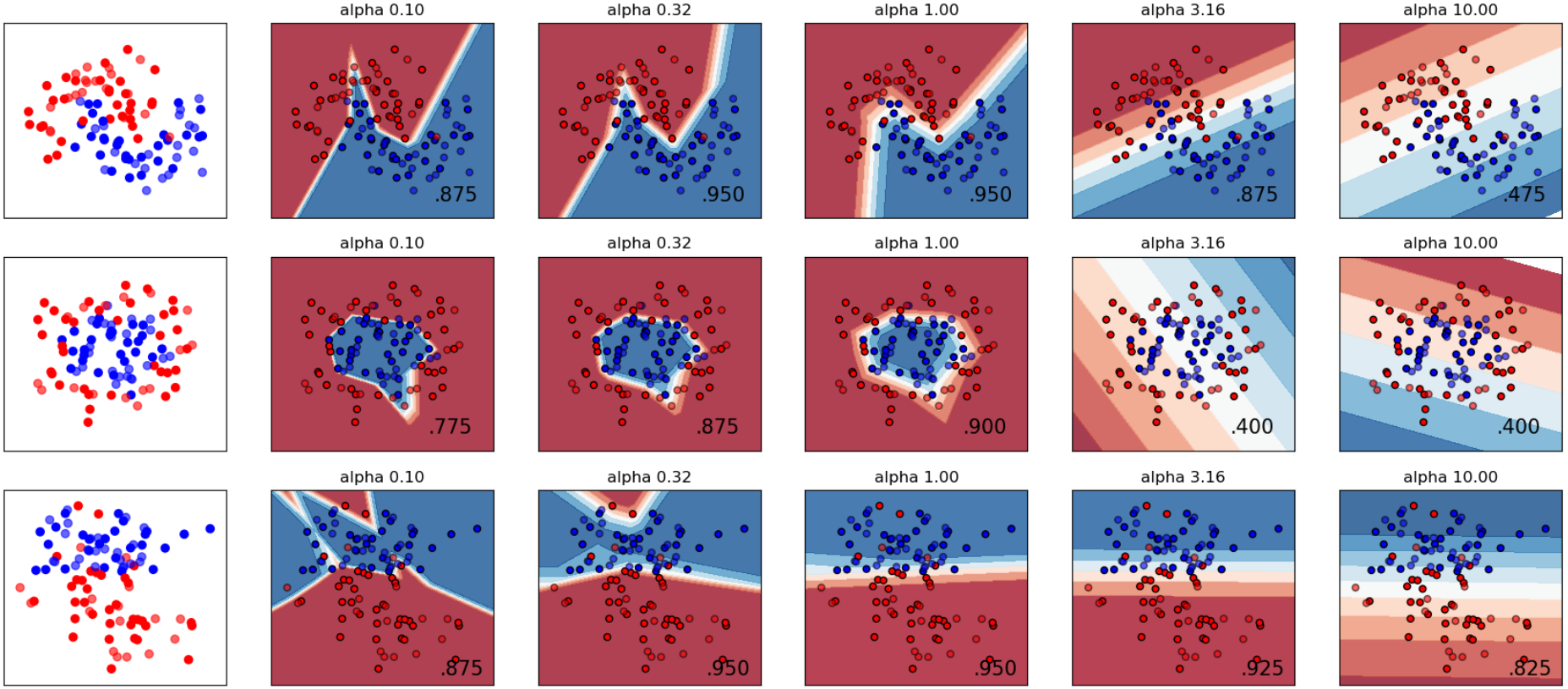
- α: controls magnitude of penalty
  - Different α yields different decision boundaries

MLP: Regularisation

- N.b.: Regularisation is not a technique specific to Neural Networks
  - Also commonly used e.g. in regression techniques
    - Ridge $\sum_{i=1}^{n} (y_i - \sum_{j=1}^{p} x_{ij} \beta_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$
    - Lasso $\sum_{i=1}^{n} (y_i - \sum_{j=1}^{p} x_{ij} \beta_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$
- Weight of regularisation is (yet) another parameter to set

MLP: Regularisation

- High α: reduces variance (~ overfitting)
- Low α: reduces bias (~ underfitting)



scikit-learn.org/stable/auto_examples/neural_networks/plot_mlp_alpha.html

MLP: Gradient problems

$\frac{\partial L}{\partial \theta}$ is sensitive to the initial values of θ

If we're not careful, gradients will either vanish or blow up

$$\theta \leftarrow \theta - \eta \frac{\partial L}{\partial \theta}$$

MLP: Gradient problems

$\frac{\partial L}{\partial \theta}$ is sensitive to the initial values of θ

If we're not careful, gradients will either vanish or blow up

$$\theta \leftarrow \theta - \eta \cdot 0$$

Small gradients: θ does not change, no learning happens

MLP: Gradient problems

$\frac{\partial L}{\partial \theta}$ is sensitive to the initial values of θ

If we're not careful, gradients will either vanish or blow up

$$\theta \leftarrow \theta - \eta \cdot \infty$$

Huge gradients: θ blows up, learning diverges

MLP: Gradient problems

$\frac{\partial L}{\partial \theta}$ is sensitive to the initial values of θ

If we're not careful, gradients will either vanish or blow up

In networks with many layers, this problem is amplified:

- At the output layer, the gradient is well-defined.
- Each layer we backpropagate through affects the scale
- More layers: More risk of vanishing/exploding gradient!

- Related problem: Vanishing/exploding output

MLP: Gradient problems

$\frac{\partial L}{\partial \theta}$ is sensitive to the initial values of θ

If we're not careful, gradients will either vanish or blow up

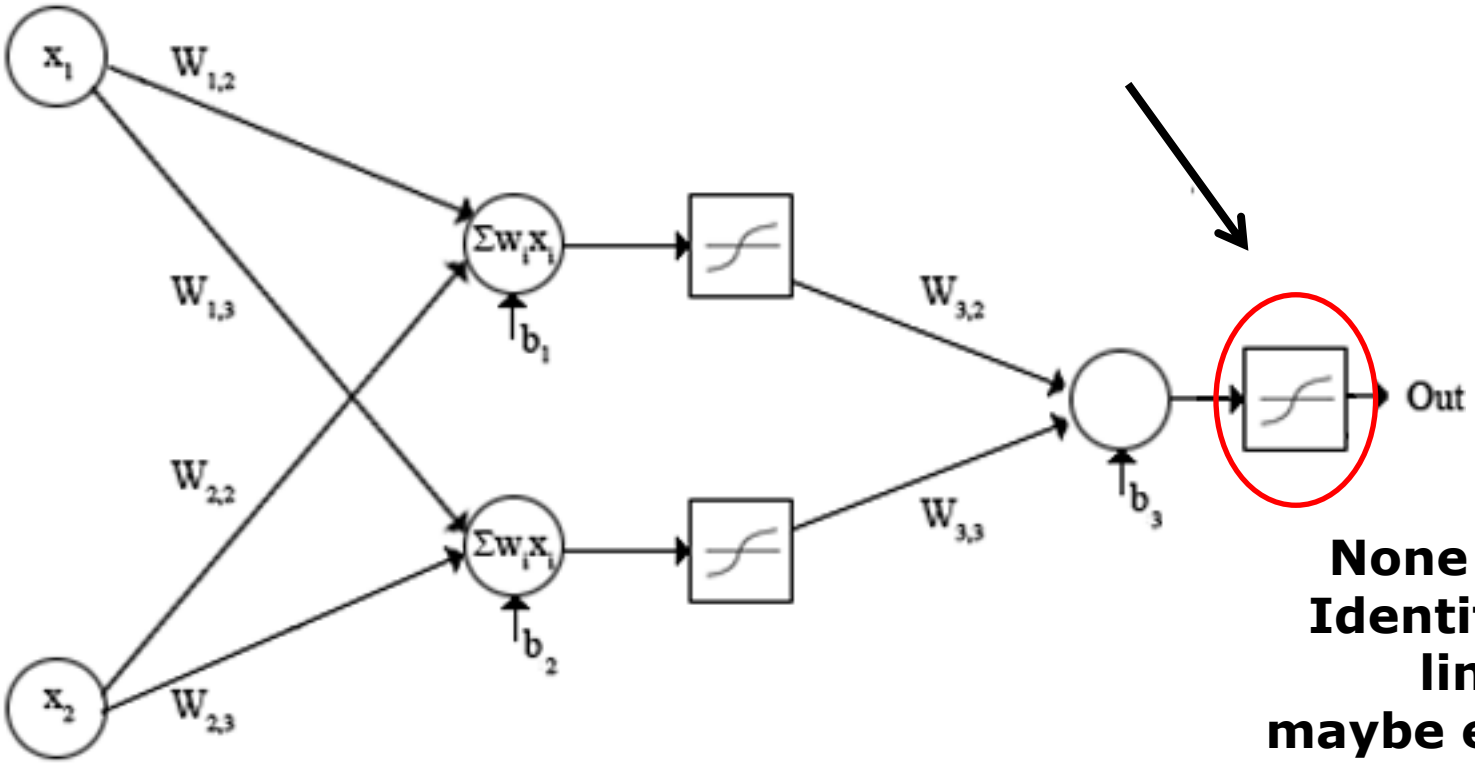
Solution: Initialize randomly, but scale such that variance of input and/or gradient is roughly preserved.

e.g.: $W_{ij} \sim N(\ =0,\ \ =2/(n_{in}+n_{out}))$

(ML Software packages provide support for this, often by default)

Multi-Layer Perceptron: Regression?

- *How to do regression?*



None (or e.g. Identity, other linear; maybe even RELU if you only require >0 values)

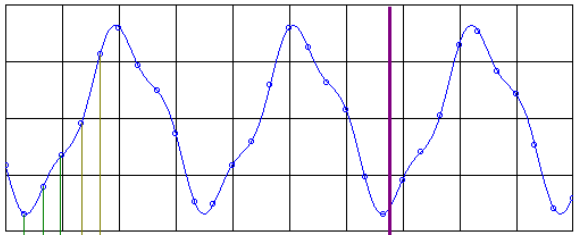
Neural Networks: properties

- Non-linear model – can solve complex dependencies
  - Non-linear activation function & hidden layers
- Also usable for regression

- Training: adapt weights between neurons
  - Back propagation
  - Training can be slow – # of hidden layers & neurons
  - Vanishing gradient problem: amount of adaption on first layers is relatively small
  - Many parameters requires many training examples!
- Black-box – difficult to understand

Neural Networks: popularity

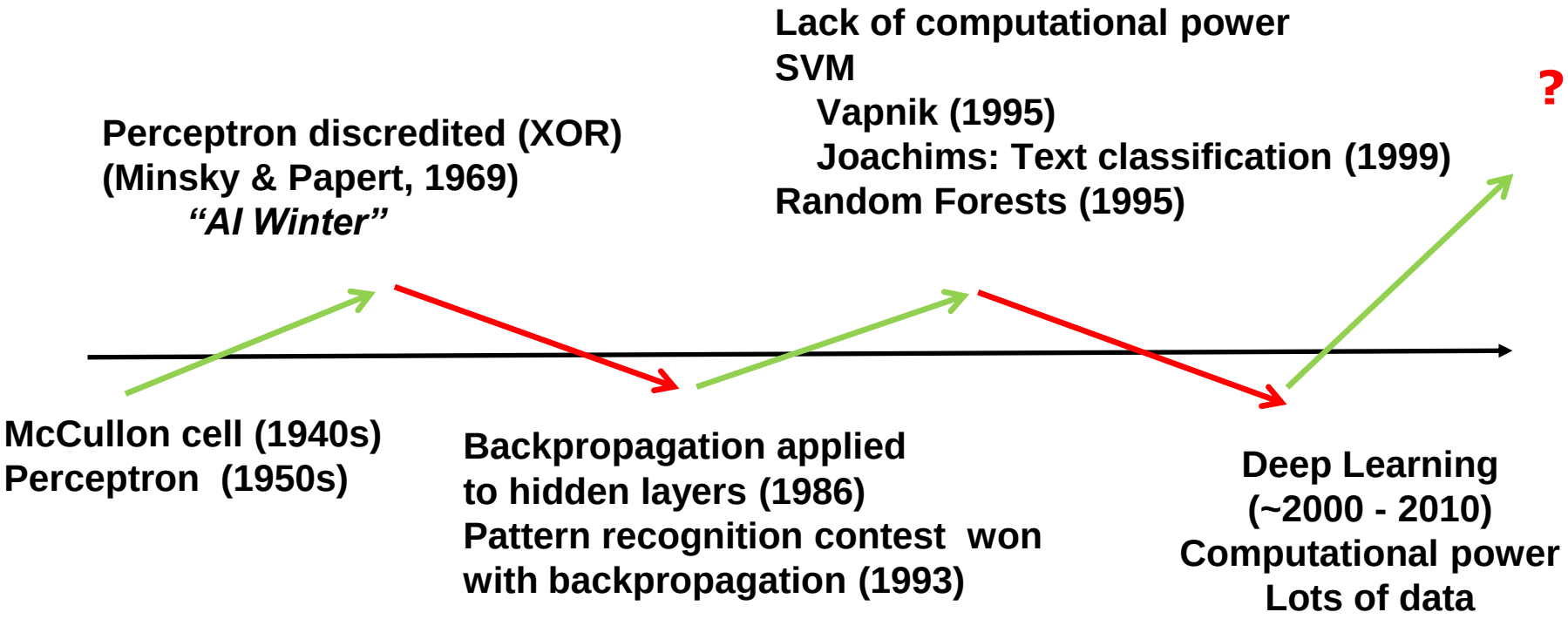
- Very popular in the late 50s/60s
- And again in 80s / early 90s
  - Speech recognition, image recognition, machine translation
  - Time series
    - Predicting stock exchange prices
  - Popular culture



- Declined popularity mid 90s
  - Also due to the emergence of SVMs & Random Forests

Neural Networks: popularity

- *Different eras of popularity!*



Several hype cycles, followed by disappointment, criticism, & funding cuts

Ups & Downs of Neural Networks

1950s – beginning of 70s: Golden years of AI (funded by DARPA):

Solve algebra, play chess & checkers, reasoning, semantic nets

"Within ten years a digital computer will be the world's chess champion"

1969: Shown that XOR problem cannot be solved by Perceptron

(led to the invention of multi-layer networks later on)

Mid 1970s: Chain reaction that begins with pessimism in the AI community, followed by pessimism in the press, followed by a severe cutback in funding, followed by the "end" of serious research ("AI winter")

Ups & Downs of Neural Networks

1980s: Governments (starting in Japan) and industry provide AI with billions of dollars. Boom of “expert systems”.

1986: **Backpropagation** had been invented in the 1970s, but only 1986 it became popular through a famous paper by David **Rumelhart**, Geoffrey **Hinton**, and Ronald Williams.
It showed that also complex functions became solvable through NNs by using multiple layers.

Late 1980s: Funders – despite actual progress in research – became disillusioned and withdrew funding again.

Ups & Downs of Neural Networks

1991: **Kurt Hornik** proved 1 hidden layer network can model any continuous function (universal approximation theorem)

1991/92 **Vanishing Gradient: problem** in multi-layer networks where training in front layers is slow due to backpropagation diminishing the gradient updates through the layers.
Identified by **Hochreiter & Schmidhuber** who also proposed solutions.

Mid 1990s - mid 2000s:

Due to lack of computational power, interest in NNs decreased again.
Other Machine Learning models, such as Bayesian models, **Random Forests** and **Support Vector Machines became popular**.

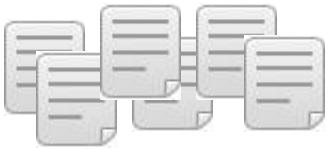
Outline

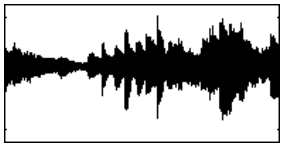
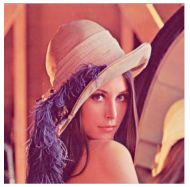
- Short Recap
- Neural Networks / MLP
- Feature Extraction (overview)
- Deep Learning
  - Intro / Definition
  - Convolutional Neural Networks
  - Convolutions in detail
  - …

Feature extraction

	F1	F2	F3	F4	Label
Sample_1	$v_{1,1}$	$v_{1,2}$	$v_{1,3}$	$v_{1,4}$	L_1
Sample_2	$v_{2,1}$	$v_{2,2}$	$v_{2,3}$	$v_{2,4}$	L_2
Sample_3	$v_{3,1}$	$v_{3,2}$	$v_{3,3}$	$v_{3,4}$	L_3
.	.				.
.		.			.
.			.		.
Sample_n	$v_{n,1}$	$v_{n,2}$	$v_{n,3}$	$v_{n,4}$	L_n

- Until now: data already “processable”
  - Numeric / categorical data
  - Needed only smaller transformations (1-n coding, ...)
- What about text, image, audio, ... resources?




- *Why can’t we input the raw image/text to a classifier?*
  - Complex data doesn’t contain “positional features” (meaning of 5th word, pixel at position 10/5, sound at second 5:00 …. can’t be learned)
- Feature extraction: derive characteristic features, convert complex data to numeric representation: Vector/matrix
- *Related: Feature selection – more on that later..*

DL Advances: Image Classification

- MNIST Dataset
  - Inputs are very uniform
  - All pretty much same size
  - All pretty much centred
  - Inputs: pixel values at specific positions
    - MNIST: grey-scale (otherwise: RGB or other colour space)
  - They have an actual fixed pattern
  - Can actually learn a mapping from (pixel position + colour value) ➔ class
    - Wouldn't work if data was not so regular
- Does not work well for "normal" datasets (where objects may appear in parts of the image)

➔ Extract descriptive features

Text Feature Extraction

Can't process in ML Model

Four score and seven years ago our fathers brought forth on this continent, a new nation, conceived in Liberty, and dedicated to the proposition that all men are created equal. Now we are engaged in a great civil war, testing whether that nation, or …

Documents to be analysed

Tokenizer

Linguistic modules

Tokens

Nation | years | fathers

Modified tokens

Indexer

nation | year | father

Bag-of-words

nation – 5
civil - 1
war – 2
men – 2
died – 4
people – 5
Liberty – 1
God – 1
…

Text Feature Extraction

- Tokenization
  - Cut character sequence into word tokens
    - Deal with ***"John's"***, ***a state-of-the-art solution*** *(phrases)*
- Normalization
  - Map text and query term to same form
    - You want ***U.S.A.*** and ***USA*** to match
- Stemming
  - We may wish different forms of a root to match
    - ***authorize***, ***authorization***
- Stop words
  - We may omit very common words (or not)
    - ***the, a, to, of***

Text Feature Extraction

- Bag-of-words (tokens) indexing (model): obtain vector of features for each document
  - Each word (token) = one attributes/variable/feature
  - Values = count of tokens in a documents
  - Also known as: term/document matrix, count matrix
  - Can be processed by ML model

Document/Word	Nation	Civil	War	Year	...	Word n	∑
Gettysburg Address	1	0	0	2		0	16010
Hamlet	1	0	4	0		2	12250
CIvil Code	1	3	0	0		5	10000
...						0	
Doc m	1	2	1	3		0	7500

- Disregards context of words / “spatial” information
  - But often works well (enough)

Text Feature Extraction

Document/Word	Nation	Civil	War	Year	...	Word n	∑
Gettysburg Address	1	0	0	2		0	16010
Hamlet	1	0	4	0		2	12250
CIvil Code	1	3	0	0		5	10000
...						0	
Doc m	1	2	1	3		0	7500

- *Value range?*
  - *0 .. |Length of (largest) document|*
- *Any issues for classification with that?*
  - ***What about distances?***

Scaling / Normalisation / Standardisation

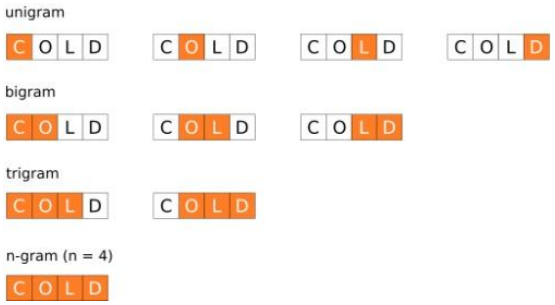
- Normalisation of text feature values
  - Length (size) of the object described numerically might influence the feature values in BOW
  - Longer documents ➔ in general higher values
  - **Relative** importance of words within a text not higher
  - ➔ Normalise data vectors to the same length
    - Dividing each vector by its vector length
      - That's the same as computing relative frequencies …

$$z_i = \frac{x_i}{\| x \|}$$

- Very different operation from min-max / std score
  - Unit length normalises the row
  - Min-max, std score, … normalise the column

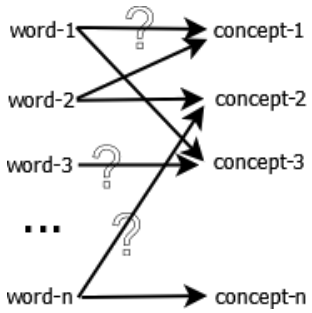
Text Feature Extraction

- Related/other approaches
  - tf-idf weighting of bag-of-words
    - Weight tokens by how often they appear in other documents
    - idf: inverse document frequency
    - *Why?*
  - n-grams: sequences of characters/words
    - Character n-grams: language identification (textcat tool)
    - Word n-grams: capture phrases
    - *Issues?*
      - *Number of tokens grows rapidly*
      - *E.g. English alphabet, 26 letters, trigram: 26^3 combinations (17576)*
    - *What is the unigram model?*
      - *If on tokens: same as bag-of-words*

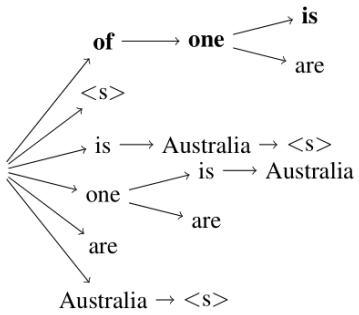


Text Feature Extraction

- Related approaches
  - Latent semantic analysis (LSA) / indexing (LSI)



  - More complex forms of "language models"



- More information: "Introduction to Information Retrieval", Manning et. al.
  - online at http://nlp.stanford.edu/IR-book/
- *Information Retrieval Lectures at TU Wien*

Image feature extraction

- *How are images different than text?*
  - Pixels are the smallest “unit” – but they are generally meaningless without their neighbours

- Simple features
  - Size, aspect ratio, ... – *useful ?*
  - Colour histograms – *useful?*

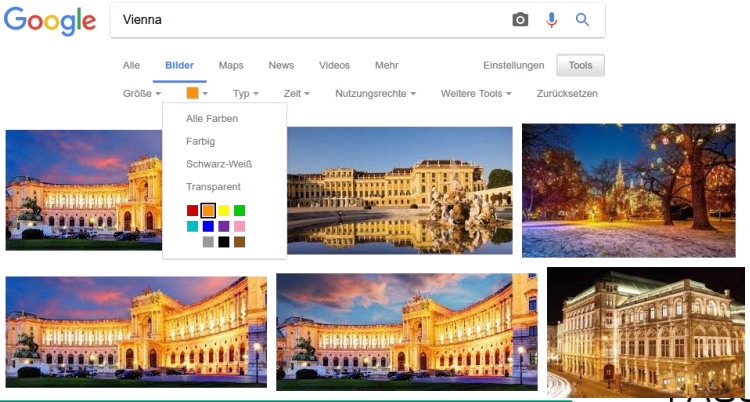


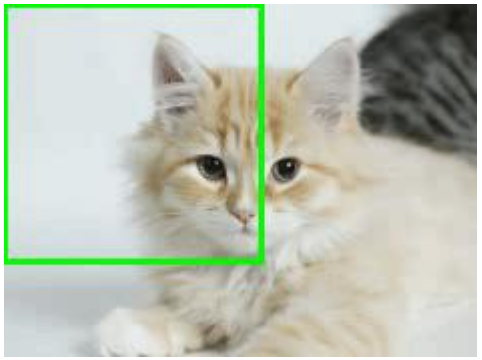
Image feature extraction

- Detect relevant parts of information
  - Edge detection
    - Points where brightness changes
    - Sobel, Canny edge detection, ...
    - Challenges: fragmentation, missing edges, false edges

  - Corner detection (Harris), ridge detection, ...

  - Scale-invariant feature transform (SIFT)

Image feature extraction



- General: process subsets of image (sliding window)

- Apply some *filter* on it
  - Multiply input within window with filter matrix

- Filters vary by values

0	0	0
0	1	0
0	0	0

custom

Image feature extraction

- Filters such as edge detection also used in image processing software

- Multiply image with a filter (kernel matrix) to detect edges, sharpen, blur, ..

Operation	Kernel	Image result
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	

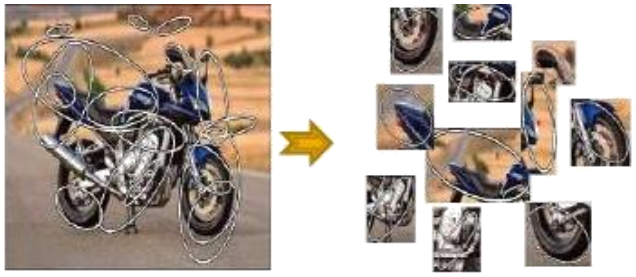
Image feature extraction

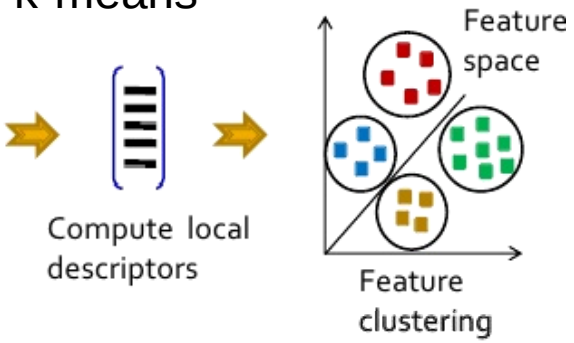
- Now, we detected some edges, other simple shapes… How to use this further?

1. Construct a “visual bag-of-words”
   – Visual words == prototypical edges etc.
2. Map each detected each to a prototype
   – Otherwise we have too many different entities
3. Count how often these appear in an image
   – We get a matrix “document” x “words” as for text
4. Train a classifier
   – Hope that features are characteristically enough

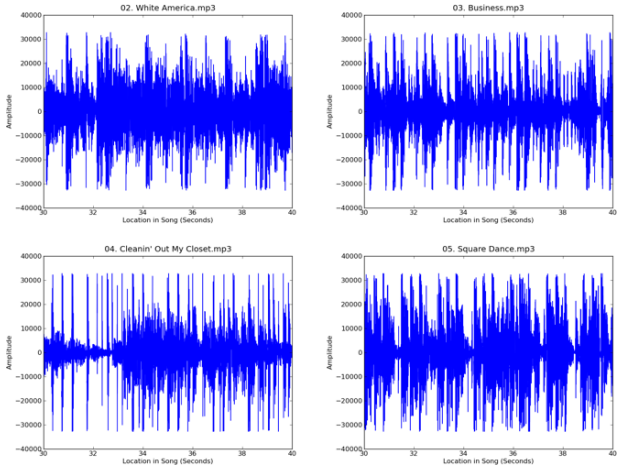
Image feature extraction

- Bag-of-visual-words
  - Extract e.g. SIFT descriptors
  - Obtain prototypes via clustering
    - E.g. say we want 500 features – compute k-means
  - Compute histogram for each image
    - Count “visual words”
  - Use as classifier input



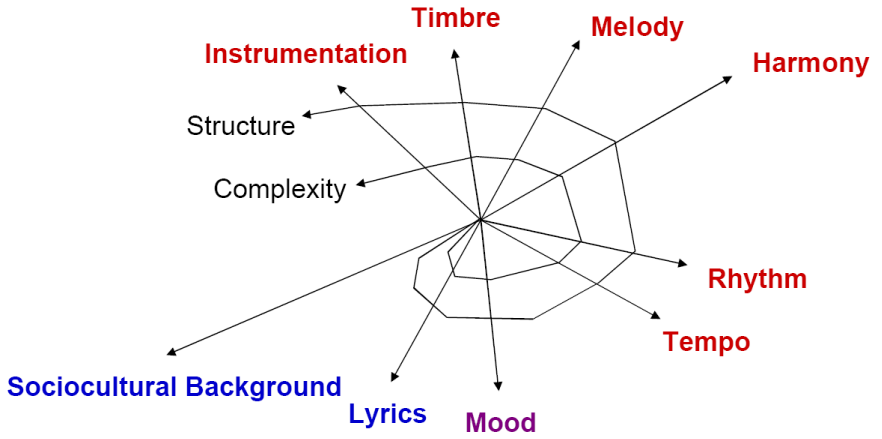


Music feature extraction

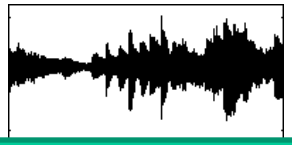


- *What's the challenge in music?*
  - Too much audio data
  - Time dimension

- Reduce audio to extract information about:
  - Pitch
  - Timbre
  - Rhythm (beat detection)
  - Etc.

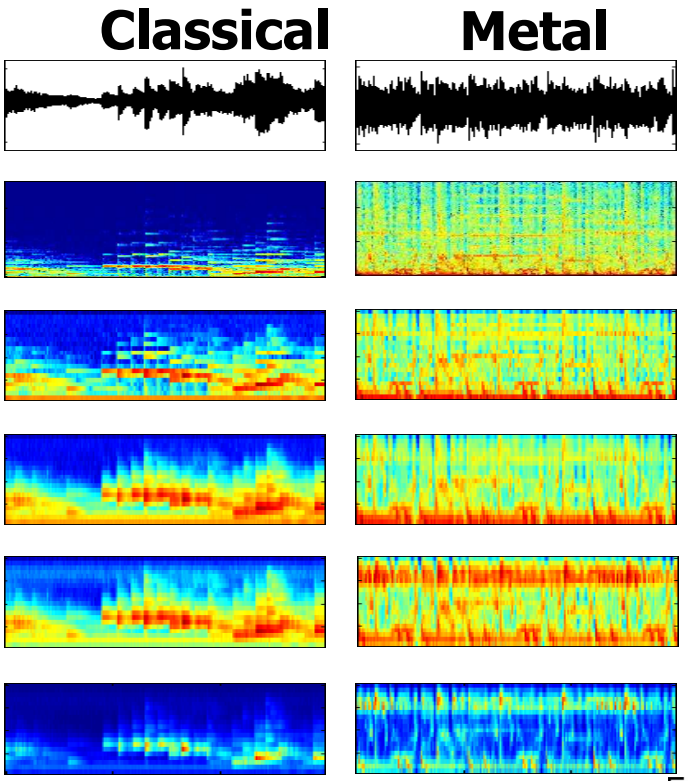


- Symbolic vs. audio features

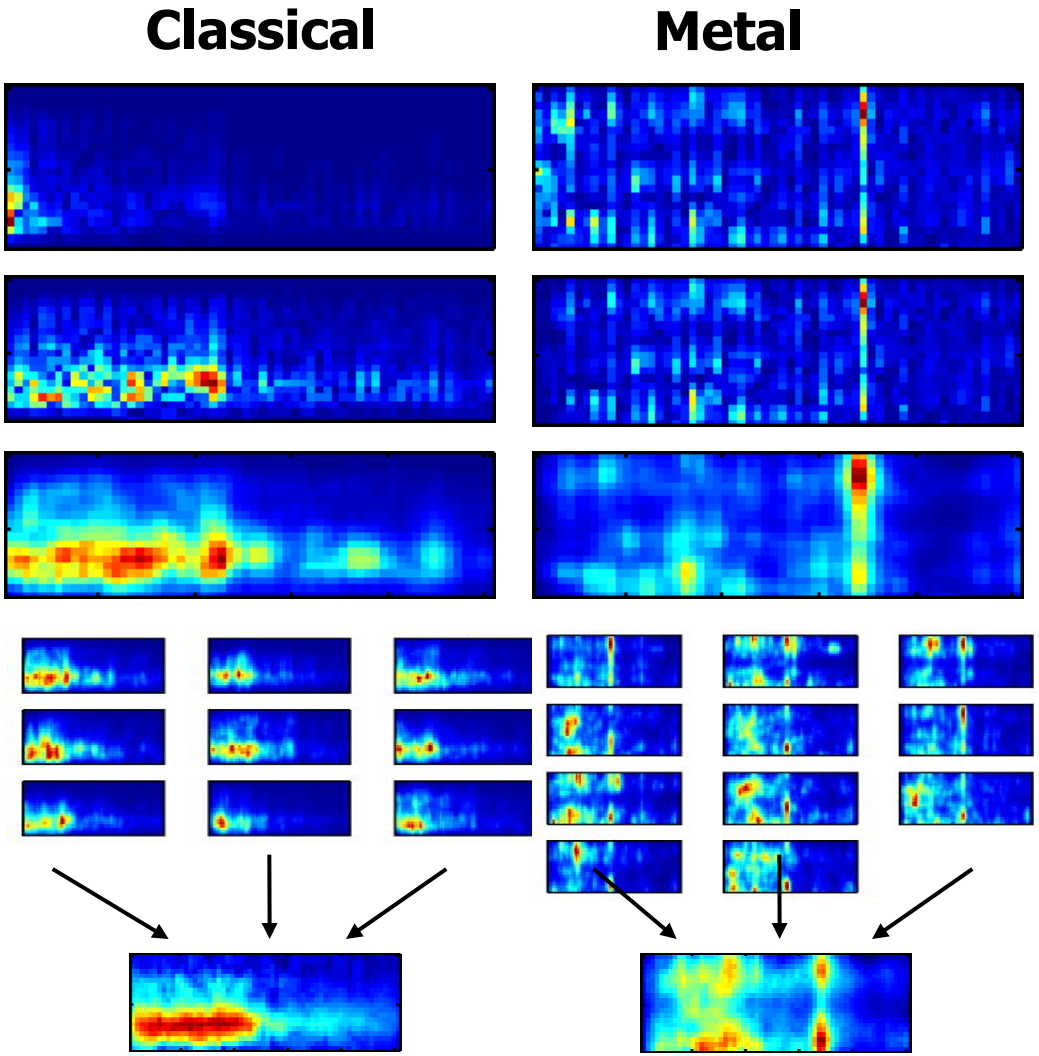


Music feature extraction: audio

- Often series of steps of transformations of audio signal
- Inspired by psycho- acoustics, mimics human listening



Music feature extraction: audio



Loudness Modulation Amplitude (60 values)

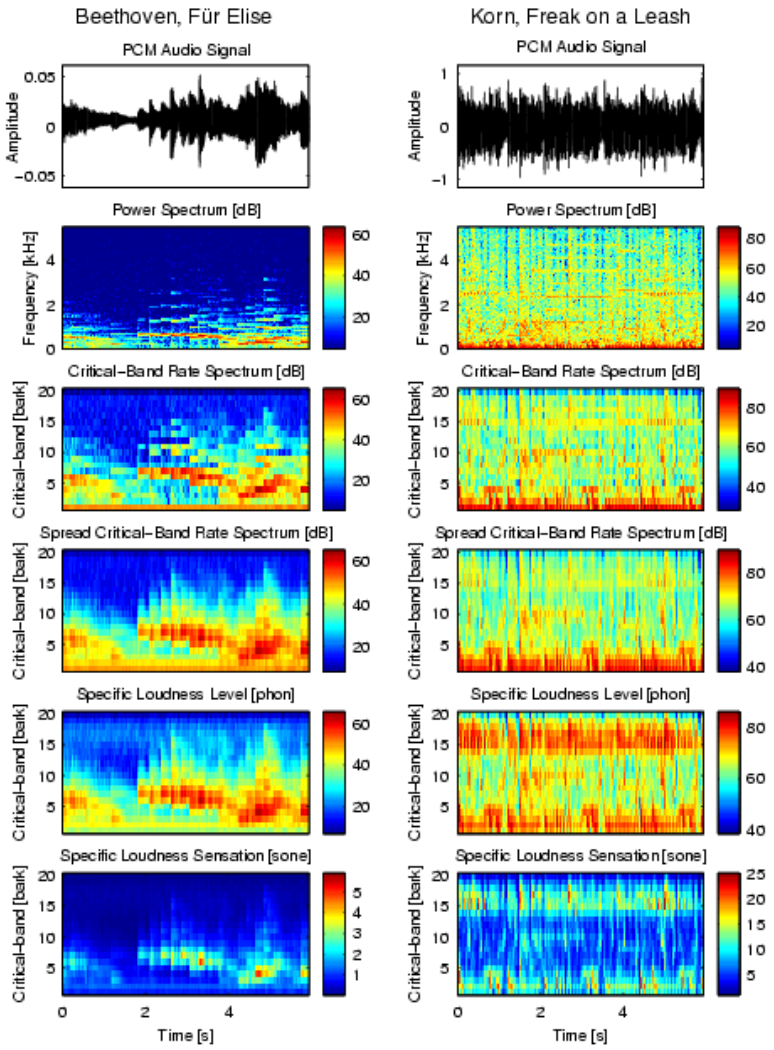
Fluctuation Strength

Filter (Gradient, Gauss)

How to handle time?
Median

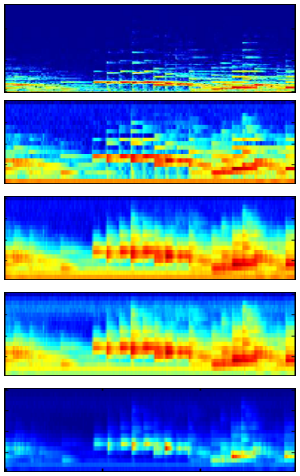
Music feature extraction: audio

- Often series of steps of trans-formations of audio signal (starting with Fourier transform)
- Inspired by psycho-acoustics, mimic human listening
- *How to handle time domain?*
  - Analyse short segments, then aggregate, e.g. average
- Feature sets
  - MFCCs, Rhythm Patterns (RP), SSD, ...
  - Count of activity on various frequencies



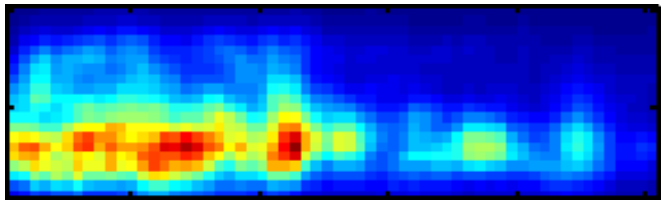
Music feature extraction: audio

- Statistical Spectrum Descriptors (SSD)



SSD: 24*7=168-dimensional vector

24
critical
bands



mean
median
variance
skewness
kurtosis
min
max

Feature extraction

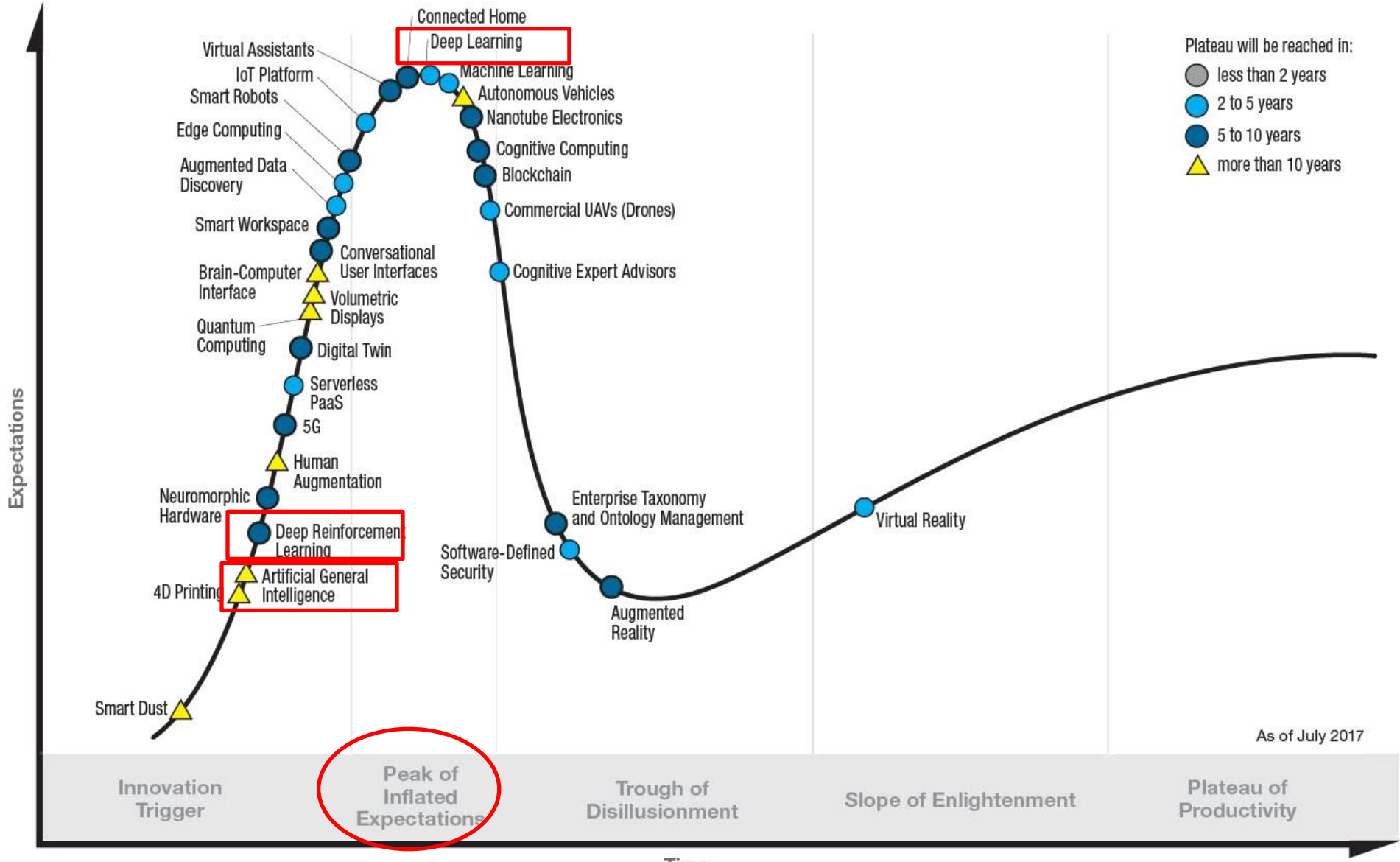
- Audio features: also for tasks such as *speaker recognition, speech-to-text*, ...
- Further modalities such as *video*
- Also from other sources
  - e.g. GPS traces to predict traffic – need to derive features from large amount of time/location data

- Often a lot of domain knowledge needed
- Especially for image features: Deep learning as a trend towards learning from “raw” data
  - *More on that soon ...*

Outline

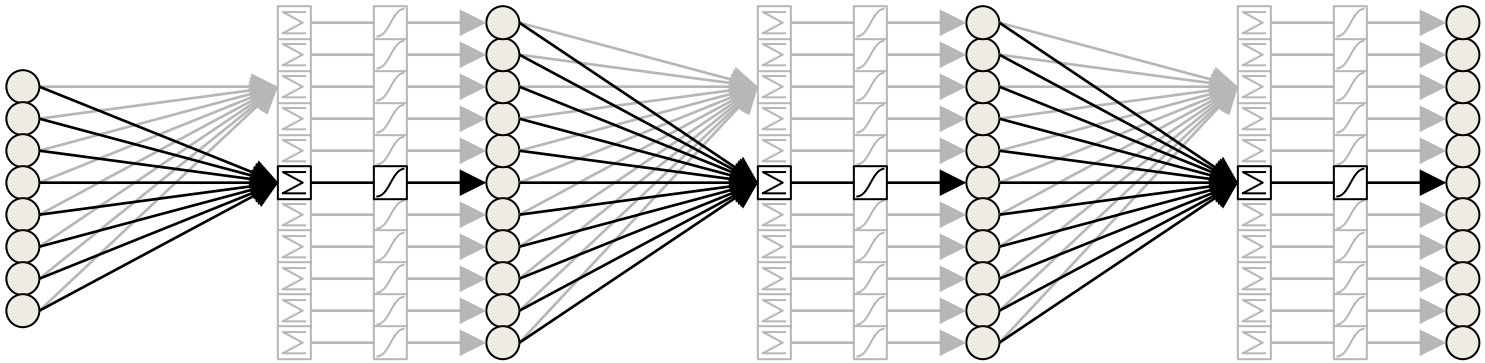
- Short Recap
- Neural Networks / MLP
- Feature Extraction (overview)
- Deep Learning
  - Intro / Definition
  - Convolutional Neural Networks
  - Convolutions in detail
  - …

Machine learning – Gartner Hype Cycle



Now what is Deep Learning?

- A term often misused, confused and hyped ☺
- Mostly associated with neural networks
- Generally: learning is considered deep when there are many (hidden) layers

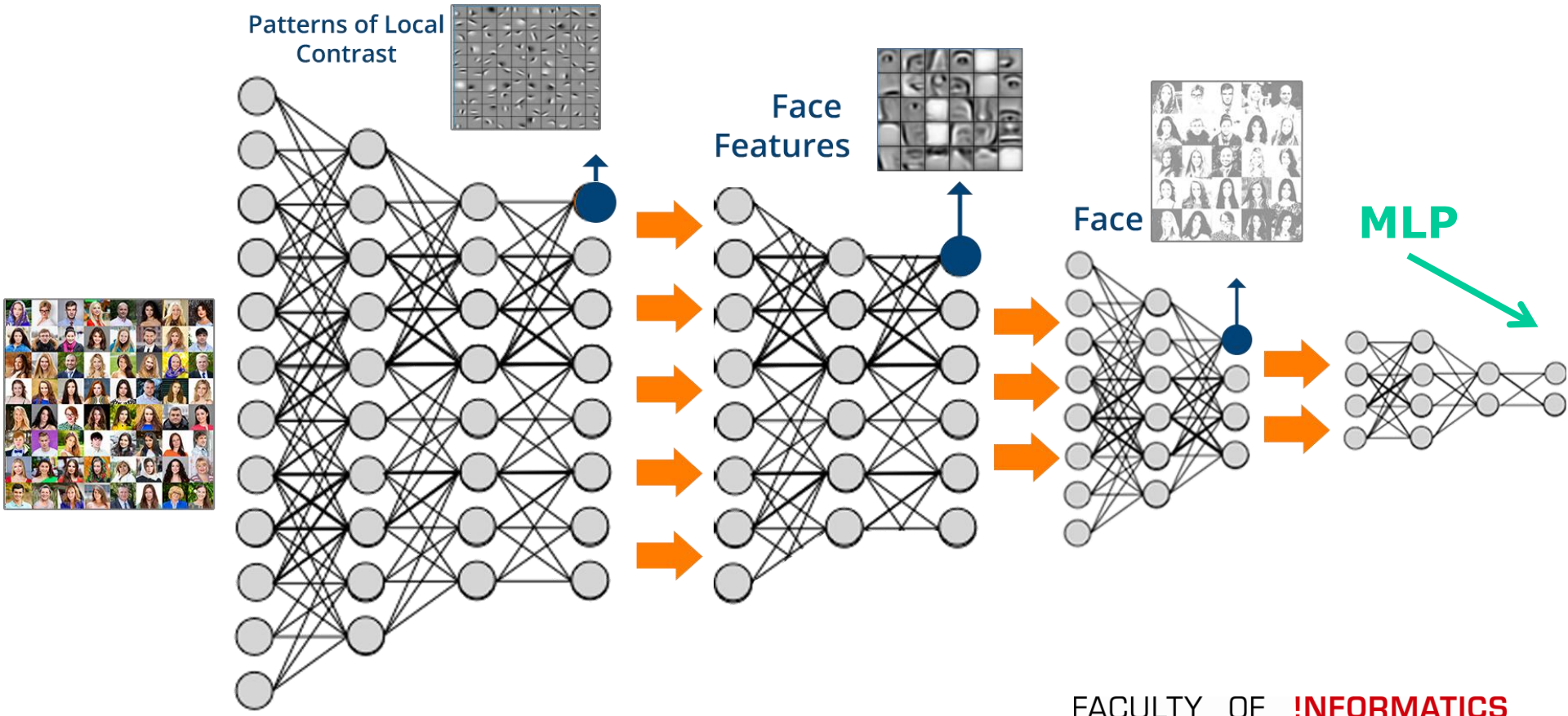


- *What is deep?*
  - Some definitions say >= 2 layers (e.g. of a neural network)

  ➔ i.e. even a relatively simple MLP is “deep learning”

Now what is Deep Learning?

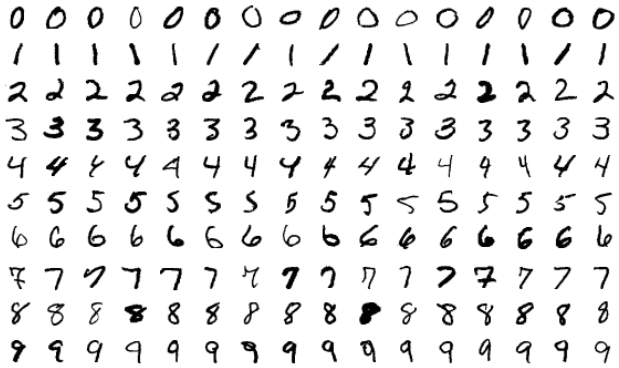
- When there are really many layer (MLPs and other processing elements, such as feature extraction), some containing layers themselves



DL Advances: Image Classification

- Pre-ImageNet: relatively small datasets
  - MNIST: 10 digits, CIFAR 10/100, …
  - Small # classes (10-100), low resolution (eg. 32 x 32 x 3)
  - Performance saturated: difficult to make significant advancements
  - MNIST benchmark results: http://yann.lecun.com/exdb/mnist/

K-NN, Tangent Distance	**1.10**	LeCun et al. 1998
K-NN, shape context matching	**0.63**	Belongie et al. IEEE PAMI 2002
Convolutional net LeNet-1	**1.70**	LeCun et al. 1998
Convolutional net LeNet-4	**1.10**	LeCun et al. 1998
Convolutional net LeNet-4 with K-NN instead of last layer	**1.10**	LeCun et al. 1998
Convolutional net LeNet-5, [no distortions]	**0.95**	LeCun et al. 1998
Convolutional net LeNet-5, [distortions]	**0.80**	LeCun et al. 1998
Convolutional net Boosted LeNet-4, [distortions]	**0.70**	LeCun et al. 1998
Trainable feature extractor + SVMs [affine distortions]	**0.54**	Lauer et al., Pattern Recognition 40-6, 2007
unsupervised sparse features + SVM, [no distortions]	**0.59**	Labusch et al., IEEE TNN 2008
Convolutional net, cross-entropy [elastic distortions]	**0.40**	Simard et al., ICDAR 2003
large conv. net, random features [no distortions]	**0.89**	Ranzato et al., CVPR 2007
large conv. net, unsup pretraining [elastic distortions]	**0.39**	Ranzato et al., NIPS 2006
large conv. net, unsup pretraining [no distortions]	**0.53**	Jarrett et al., ICCV 2009
large/deep conv. net, 1-20-40-60-80-100-120-120-10 [elastic distortions]	**0.35**	Ciresan et al. IJCAI 2011
committee of 7 conv. net, 1-20-P-40-P-150-10 [elastic distortions]	**0.27**	Ciresan et al. ICDAR 2011
committee of 35 conv. net, 1-20-P-40-P-150-10 [elastic distortions]	**0.23**	Ciresan et al. CVPR 2012



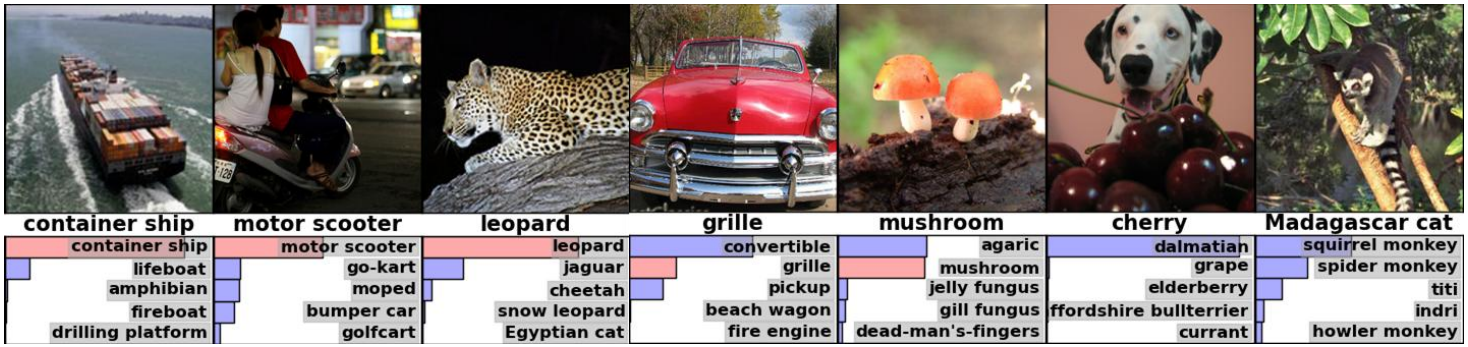
DL Advances: Image Classification

- MNIST Dataset
  - Inputs are very uniform
  - All pretty much same size
  - All pretty much centred
  - Inputs: pixel values at specific positions
    - MNIST: grey-scale (otherwise: RGB or other colour space)
- How to train a standard Neural Network (MLP)?

  ➔ Convert 2D input matrix to a 1D vector
- Feed in into the MLP
- Does not work well for "normal" datasets

DL Advances: Image Classification

IMAGENET

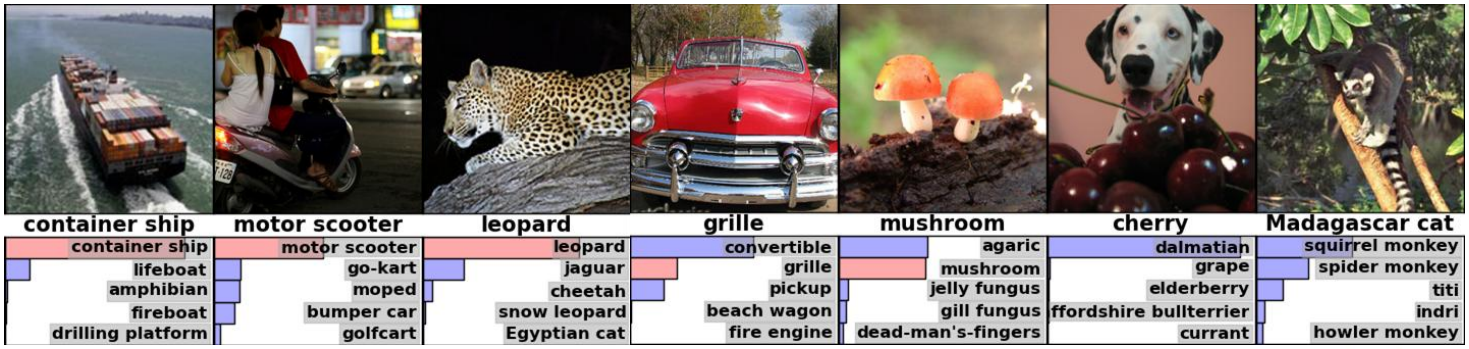


- Large Scale Visual Recognition Challenge (ILSVRC)
  - Started 2009, annotated by mechanical turk
  - Much larger scale than any previous
  - 1.2 million training images in 1000 classes
  - Task: assign labels (classes) to each image
  - Evaluation: set of 150.000 images
  - Metric: prediction among top-1 or top-5 labels

Deng et al. Imagenet: A large-scale hierarchical image database. *Computer Vision and Pattern Recognition, 2009*

DL Advances: Image Classification

IMAGENET



2012: AlexNet achieves 16.4% error (first deep net)

2nd-best entry: 26.2%.

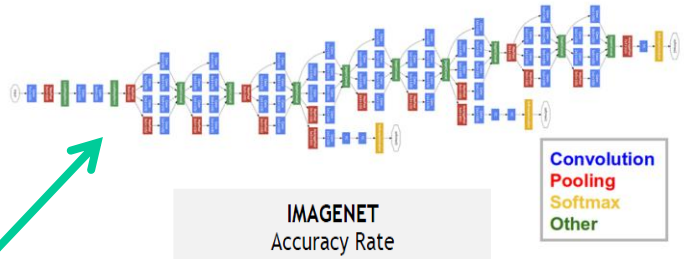
2013: Clarifai: 11.7% error (most entries are DL)

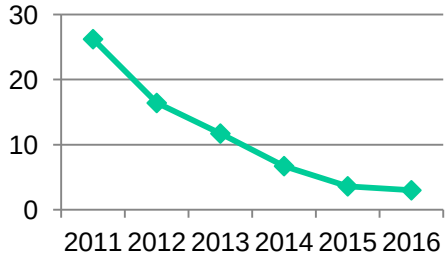
2014: GoogLeNet: 6.7% error (22 layers)

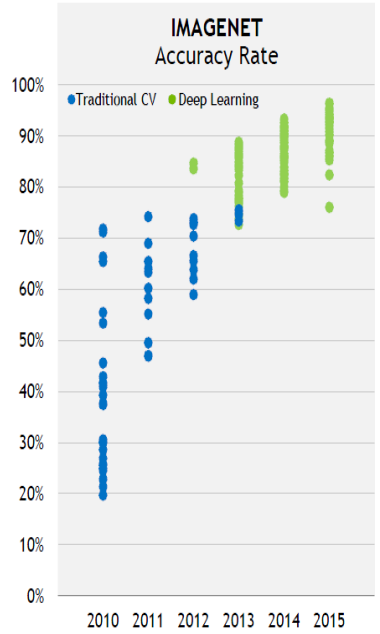
2015: ResNets: 3.6% error (138 layers !)

2016: 2.99% error (ensembles of GoogLeNet, Resnet, ..)

2017: 2.25% error

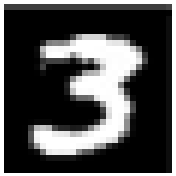






Now what is Deep Learning?

- **"Traditional" machine learning**
  - Express problem as a function, automatically adapt parameters from data



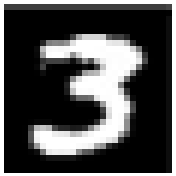
$$y = f(x; \theta)$$

adapt this

"3"

to predict this

- **Deep learning**
  - Learnable function is a **stack of many simpler functions** that often have the same form



$$y = f(f(f(x;\theta_1);\theta_2);\theta_3)$$

adapt this

to predict this

"3"

Now what is Deep Learning?

$$y = f(f(f(x;\theta_1);\theta_2);\theta_3)$$

adapt this

to predict this

"3"

- Deep learning
  - Learnable function is a **stack of many simpler functions** that often have the same form
  - Often, it is an artificial neural network (ANN, e.g. a DNN)
  - Often, one tries to minimize hard-coded steps
    - Such as feature extraction
  - Most commonly used functions:
    - Matrix product: $f(x; \theta) = W^T x$
    - 2D Convolution: $f(x; \theta) = x * W$
    - Subsampling / pooling
    - Element-wise nonlinearities (sigmoid, tanh, rectifier)

Now what is Deep Learning?

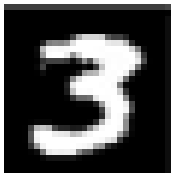
$$y = f(f(f(x;\theta_1);\theta_2);\theta_3)$$

adapt this

“3”

to predict this

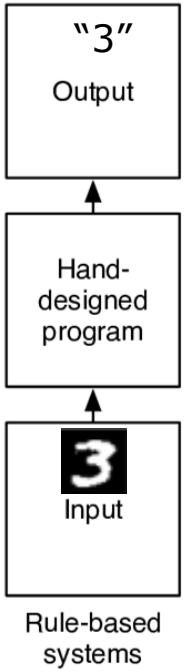
- It is convenient to express repeated function application as a sequence



- Computation in sequential steps of processing, often termed “layers”

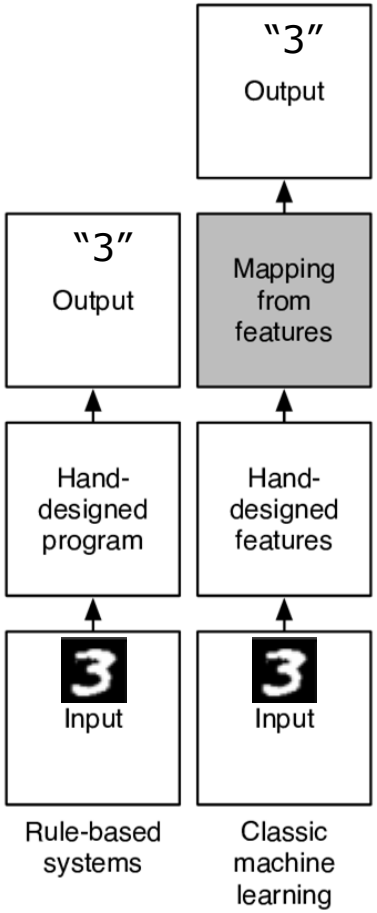
Learning Paradigms

Rule-based systems:
- Write algorithm by hand.



graphic: Y. Bengio, Deep Learning, MLSS 2015

Learning Paradigms

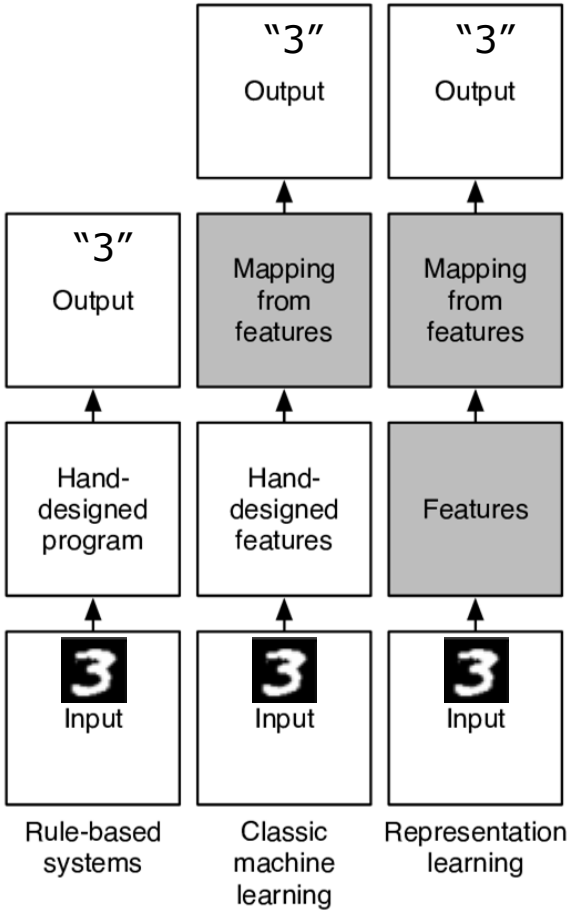


Rule-based systems:
- Write algorithm by hand.

Classic machine learning:
- ***Write feature extractor*** by hand
- Train classifier on top.

graphic: Y. Bengio, Deep Learning, MLSS 2015

Learning Paradigms



Rule-based systems:
- Write algorithm by hand.

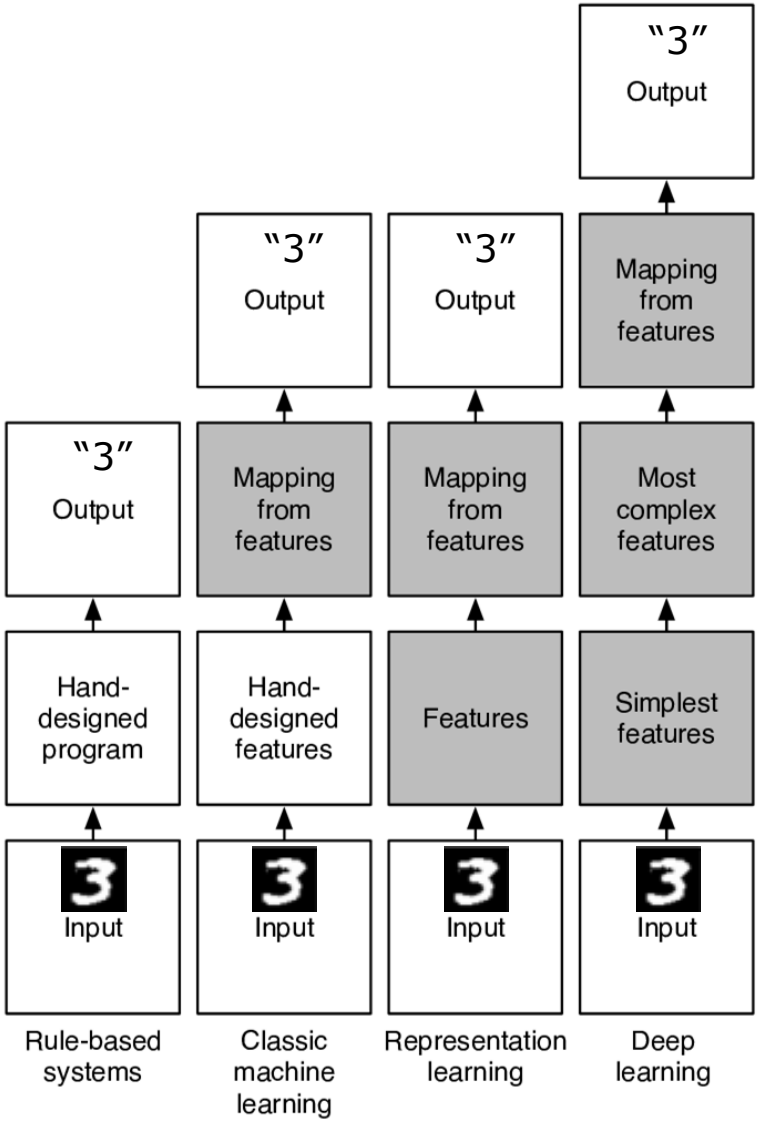
Classic machine learning:
- Write feature extractor by hand
- Train classifier on top.

Representation learning:
- ***Learn feature extractor***

(often unsupervised)
- Train classifier on top.

graphic: Y. Bengio, Deep Learning, MLSS 2015

Learning Paradigms



Deep learning:

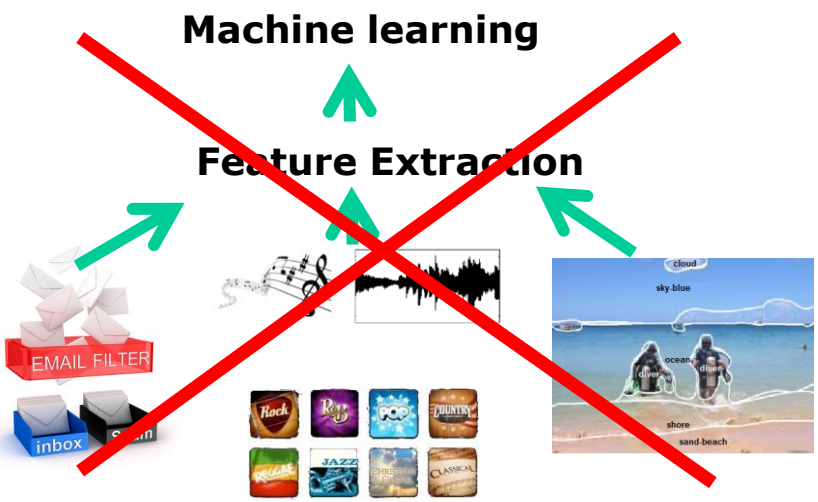
• Learn a stack of many simpler functions to map input to output.

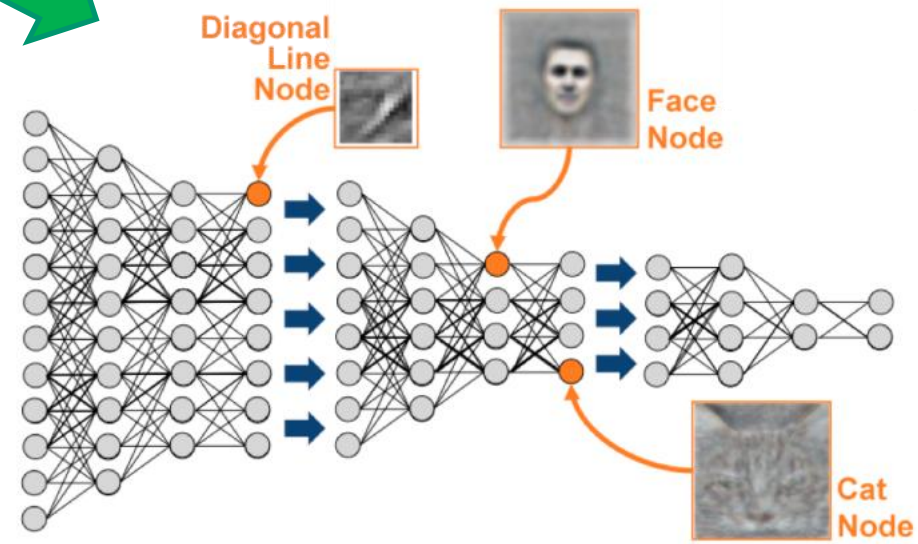
- Often, that stack is a *neural network*.
- Often, it is trained on raw input: optimize features & classification together, *minimize hand-crafting*

"**end-to-end learning**"

graphic: Y. Bengio, Deep Learning, MLSS 2015

Deep Learning





Mathematical Reasons for Going "Deep"

- A neural network with a **single hidden layer** of enough units can approximate **any continuous function** arbitrarily well.

- In other words, it can solve whatever problem you're interested in!

  (Cybenko 1998, Hornik 1991)

Mathematical Reasons for Going "Deep"

But:

- "**Enough units**" can be a very large number. There are functions representable with a deep (and small!) network, but would require exponentially many units with a single layer.

  (e.g., Hastad et al. 1986, Bengio & Delalleau 2011)

- The proof only says that a shallow network ***exists***, it does not say how to find it.

- Evidence indicates that it is easier to train a deep network to perform well than a shallow one.

Why no Deep Learning in the 1980s?

Methodically, not a lot has changed since the 1980s. But:

- **Computers were slow** ➔ neural networks tiny, could not achieve (expected) high performance on real problems

- **Datasets were small** ➔ no large datasets that had enough information to learn the numerous parameters of (hypothetical) large neural networks

- **Nobody knew how to train deep nets.** Today, object recognition networks have > 10 layers (**or many more**). In the past, everyone was very sure that such deep nets cannot be trained. Therefore, networks were shallow and did not achieve good results.

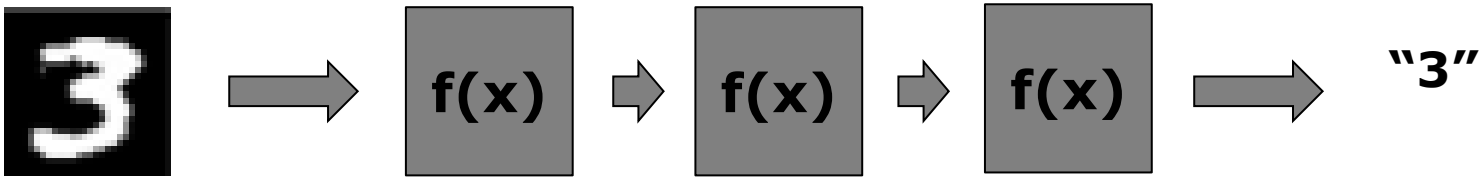
Ilya Sutskever, http://yyue.blogspot.co.at/2015/01/a-brief-overview-of-deep-learning.html

Availability of datasets

- *“Why Big Tech pays poor Kenyans to teach self-driving cars”*

- *https://www.bbc.com/news/technology-46055595*

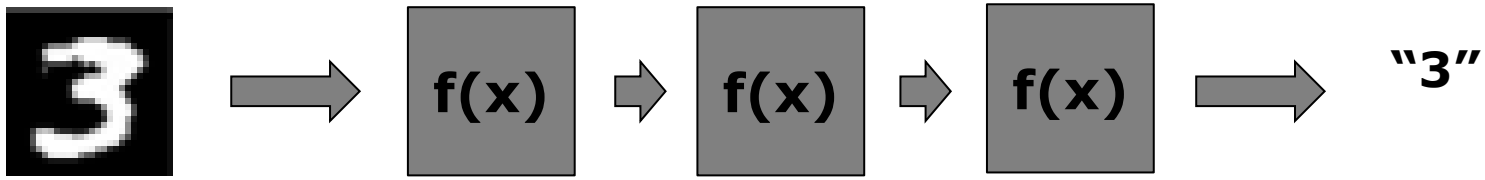
Training Deep Neural Networks



- **Training**: *heuristically find optimal weights*
  - **Randomly initialize** tuneable parameters (weights)
  - Define objective **function** telling how well the network does (on a set of training examples)
  - **Iteratively** adapt tuneable values to **minimize value of cost/loss function** (e.g. following simple gradient-based rule)

- For each step: solutions exist from the 1980s
- Recent advances since the 2000s added improvements along these steps

Popular Network Architectures

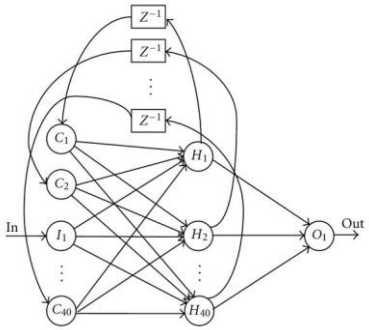


- Largest design space in employing deep learning
  - Which functions (or “layers”) to use, and in
  - Which combination / order / sequence?
    - Also: loss function, regularisation, learning rate, …

- Several popular architectures proposed
  - Particular type of layers suited for particular problems
    - *Re-use architecture* for your specific problem, or even
    - *Re-use learned parameters* via *Transfer Learning*

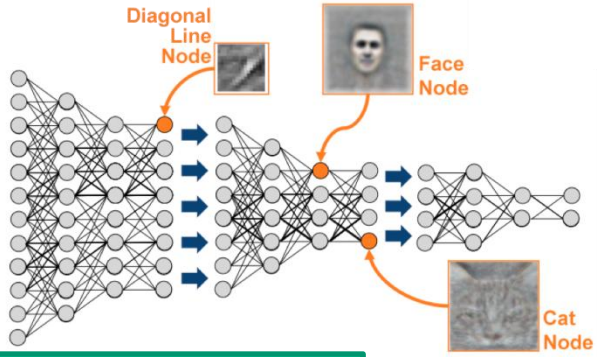
Deep Learning Architectures

- Deep (Artificial) Neural Networks (DNNs)
  - Anything with many (at least 2) hidden layers
    - E.g. a MLP
  - For any kind of data/problem



- Recurrent neural networks
  - E.g. Long short term memory network (LSTM) (1997!)
  - Mostly for time series / sequences

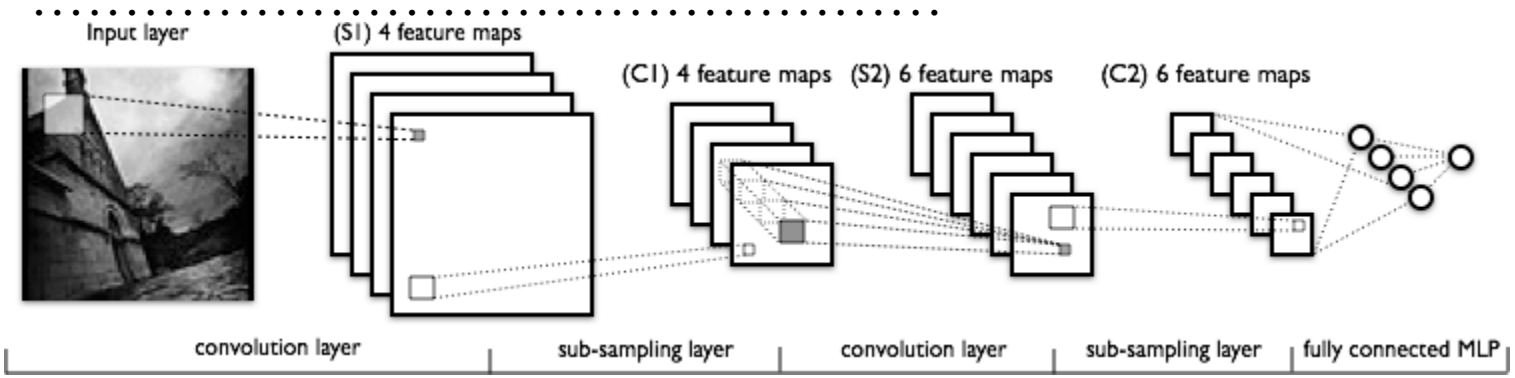
- Convolutional neural network
  - Mostly for image analysis (+audio, …)



Outline

- Short Recap
- Neural Networks / MLP
- Feature Extraction (overview)
- Deep Learning
  - Intro / Definition
  - Convolutional Neural Networks
  - Convolutions in detail
  - …

Convolutional Neural Network (CNN)



- Combines three types of layers:
  - Convolutional layer: performs 2D convolution of 2D input with multiple learned 2D kernels
  - Subsampling layer: replaces 2D patches by their maximum (“max-pooling”) or average
  - Fully-connected layer: computes weighted sums of its input with multiple sets of learned coefficients (~MLP)
- Applies nonlinear function after each linear operation

Face Recognition with CNNs

- Convolutional neural networks learn layers of “features”, from coarse to more detailed

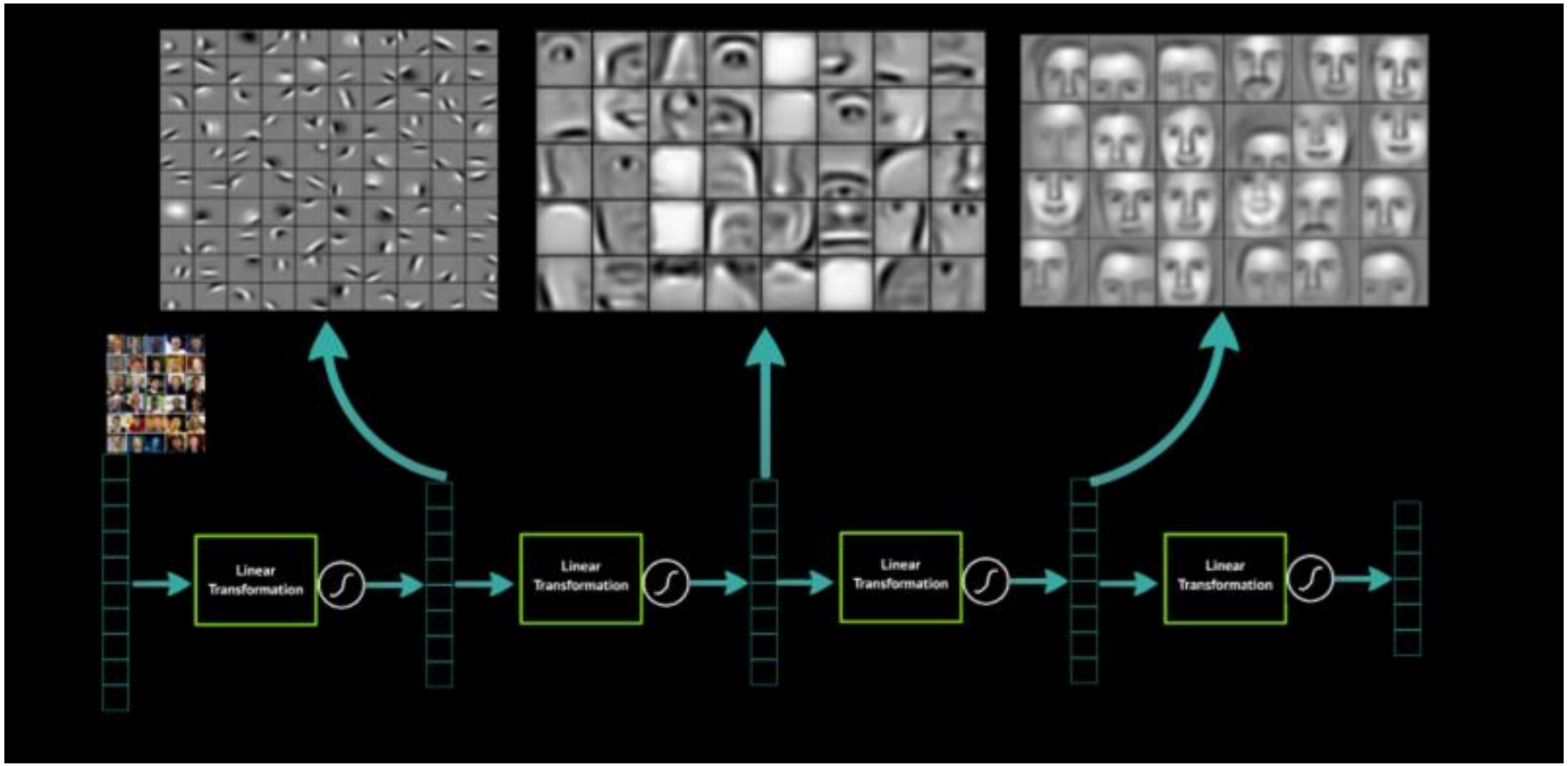
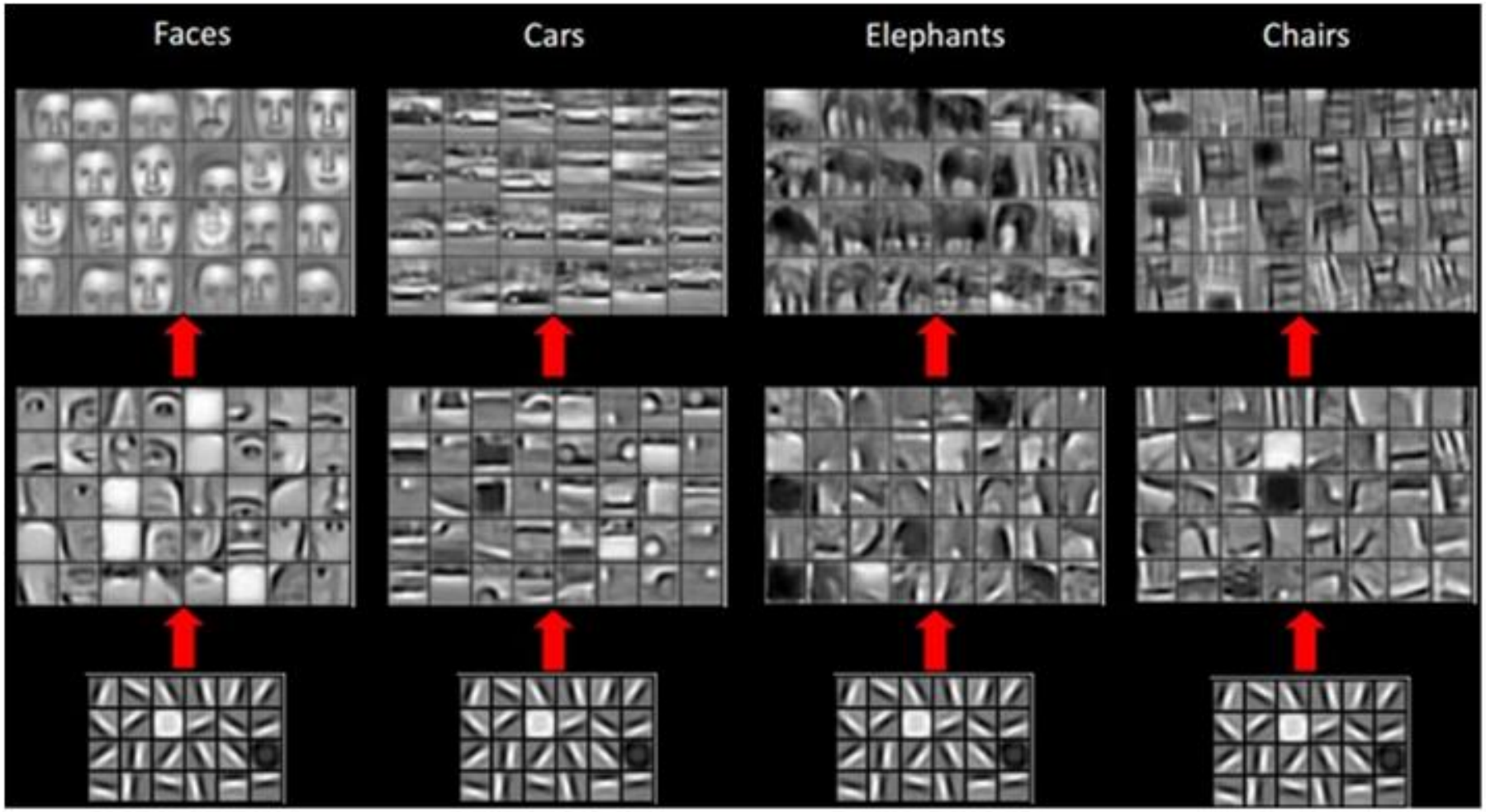


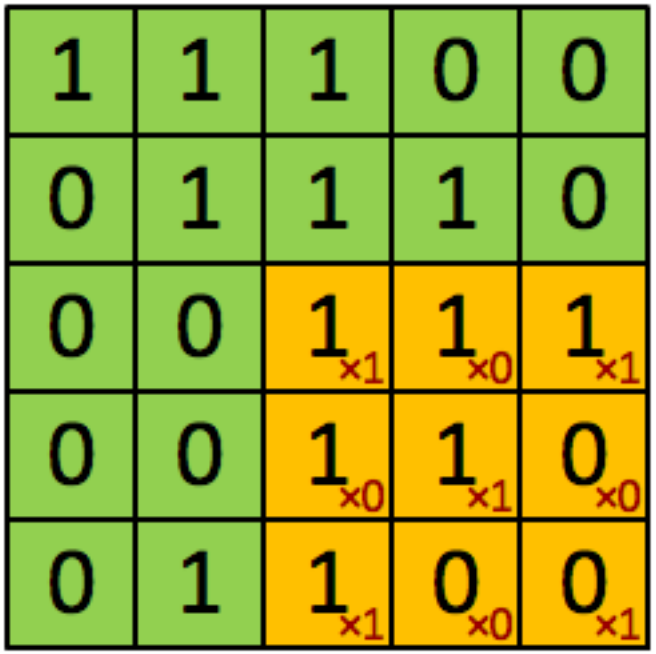
Image Object Recognition with CNNs

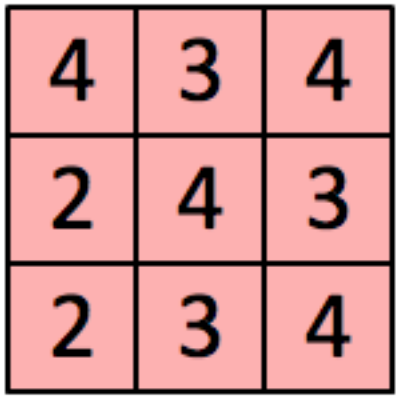


Src: http://stats.stackexchange.com/questions/146413/why-convolutional-neural-networks-belong-to-deep-learning

Convolutional Layer

- Convolution – step-wise multiply subset of input matrix (tensor) with a kernel matrix
  ➔ Produce a new matrix (tensor) output
  - Convolution only sees a part of the input (“receptive field”)

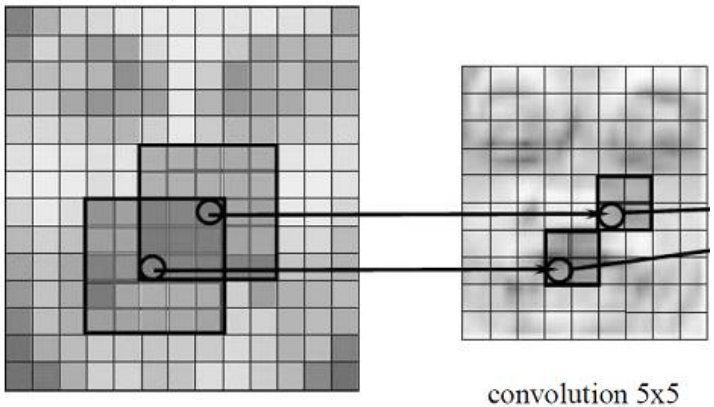




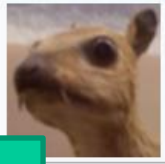
Convolutional Layer

- Apply local filter kernels (matrix / tensor)
- Performs feature extraction
  - E.g. Edge detection, etc...

- Similar concept in e.g. image processing



convolution 5x5

Operation	Kernel	Image result
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	

Outline

- Short Recap
- Neural Networks / MLP
- Feature Extraction (overview)
- Deep Learning
  - Intro / Definition
  - Convolutional Neural Networks
  - Convolutions in detail
  - …

What is a Convolution?

6 x 6 image

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

$\circledast$

3 x 3 filter

1	0	-1
1	0	-1
1	0	-1

$=$

4 x 4 result

-5			

3x1 + 0x0 + 1x-1 +
1x1 + 5x0 + 8x-1 +
2x1 + 7x0 + 2x-1 =
-5

What is a Convolution?

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

$\circledast$

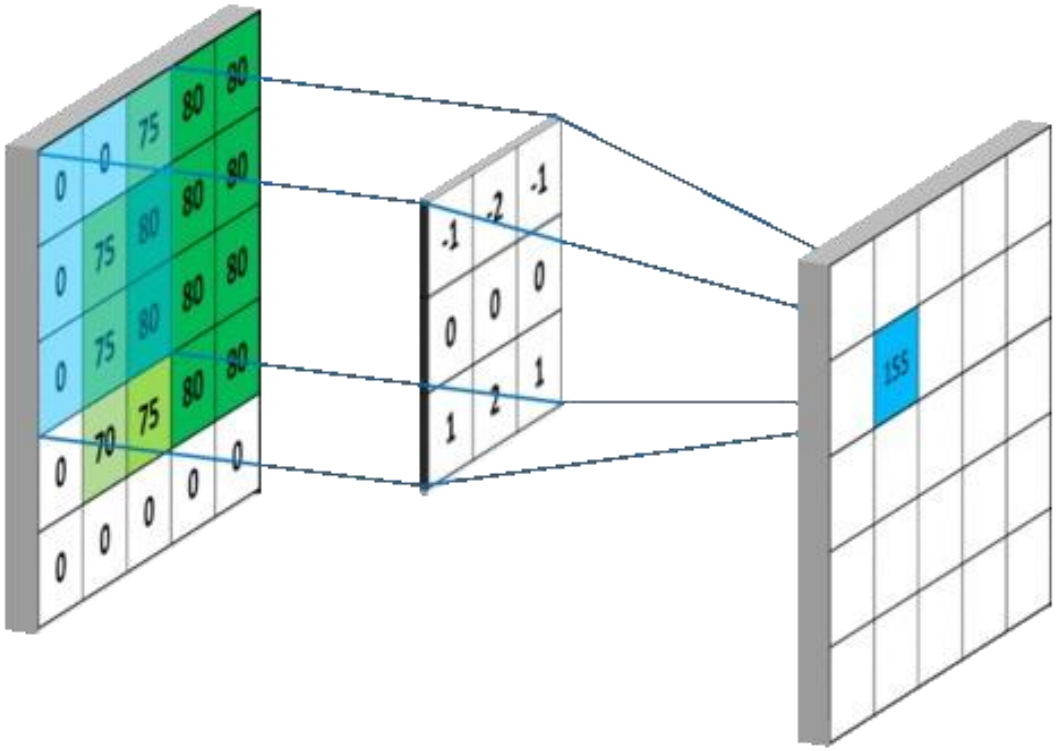
3 x 3 filter		
1	0	-1
1	0	-1
1	0	-1

$=$

4 x 4 result			
-5	-4		

0x1 + 1x0 + 2x-1 +
5x1 + 8x0 + 9x-1 +
7x1 + 2x0 + 5x-1 =
-4

What is a Convolution?



- https://mlnotebook.github.io/post/CNN1/

Convolutions: Padding

- Convolution result smaller than original image

$$n \times n \circledast f \times f \Rightarrow (n - f + 1) \times (n - f + 1)$$

$$6 \times 6 \circledast 3 \times 3 \Rightarrow (6 - 3 + 1) \times (6 - 3 + 1) = 4 \times 4$$

- Edge & corner pixels used much less than others

➔ Pad image with zeros at the edges

6 x 6 image with 1 pixel padding

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

$\circledast$

3 x 3 filter

1	0	-1
1	0	-1
1	0	-1

=

6 x 6 result

-5					

Convolutions: Padding

- With *p* pixels padding: output size defined as:

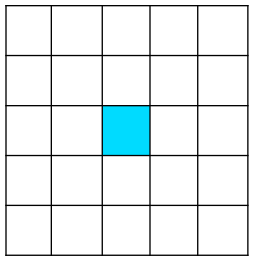
$$(n + 2p - f + 1) \times (n + 2p - f + 1)$$

- How to compute p to have output size *n x n*?

$$p = \frac{f - 1}{2}$$

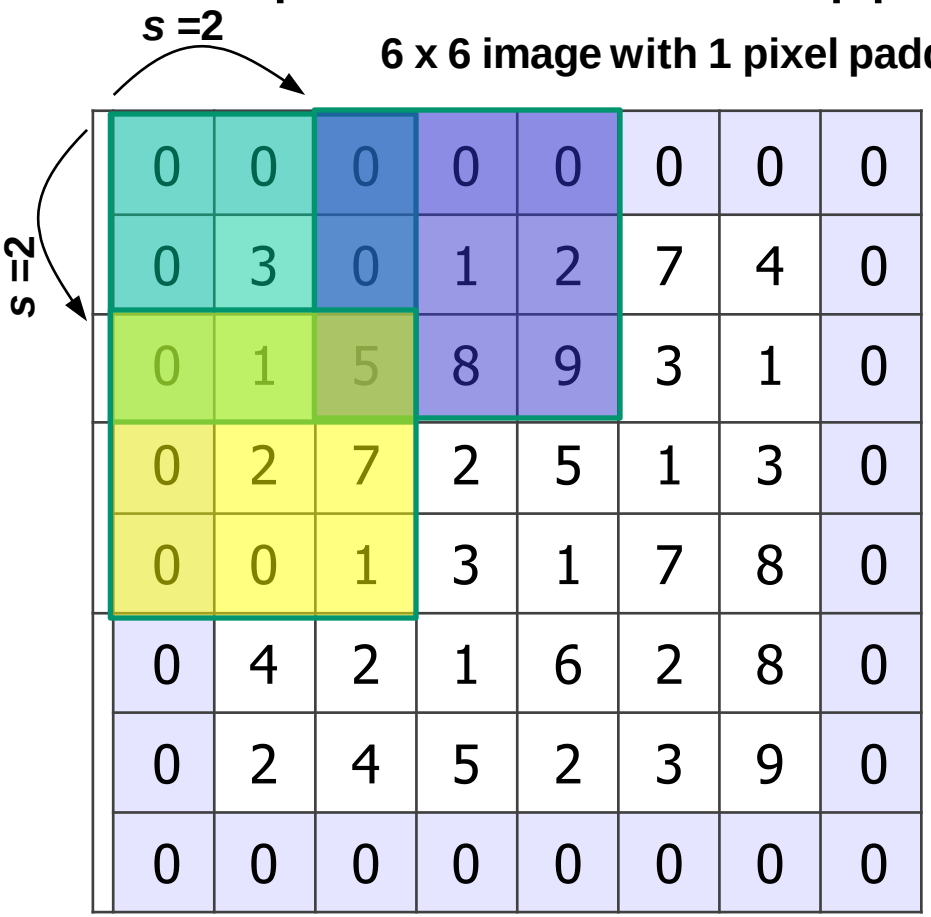
$$\begin{aligned} 3 \times 3 &\Rightarrow p = 1 \\ 5 \times 5 &\Rightarrow p = 2 \\ 7 \times 7 &\Rightarrow p = 3 \end{aligned}$$

- f is by convention odd
  - Such that p is well-defined
  - Such that the filter has a central pixel



Convolutions: Stride

- We can choose a stride parameter s to define the step size between applications of the filter



3 x 3 filter

1	0	-1
1	0	-1
1	0	-1

$\circledast$ $=$

3 x 3 result

1	-6	
13		

$$\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$$

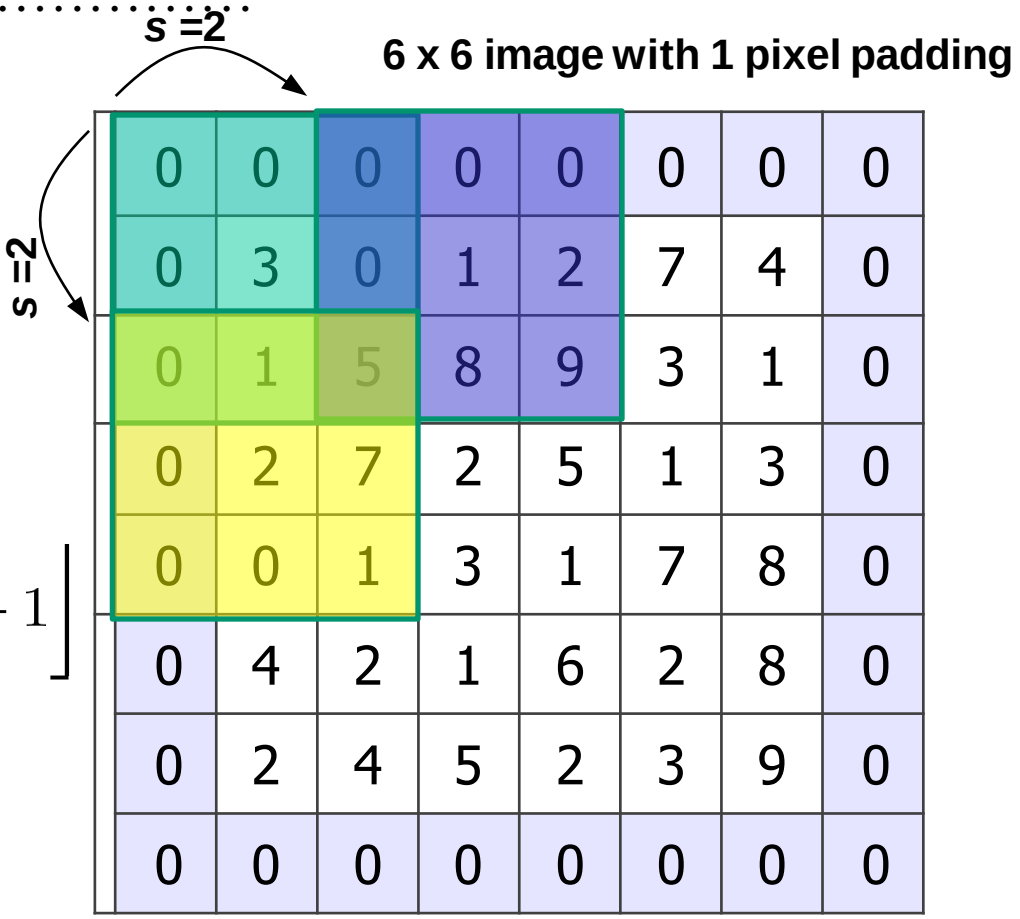
Convolutions: Stride

- *Output size?*
  - *Remember padding:*

$$(n + 2p - f + 1) \times (n + 2p - f + 1)$$

$$\left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor$$

$$\frac{6 + 2 \times 1 - 3}{2} + 1 = 3$$

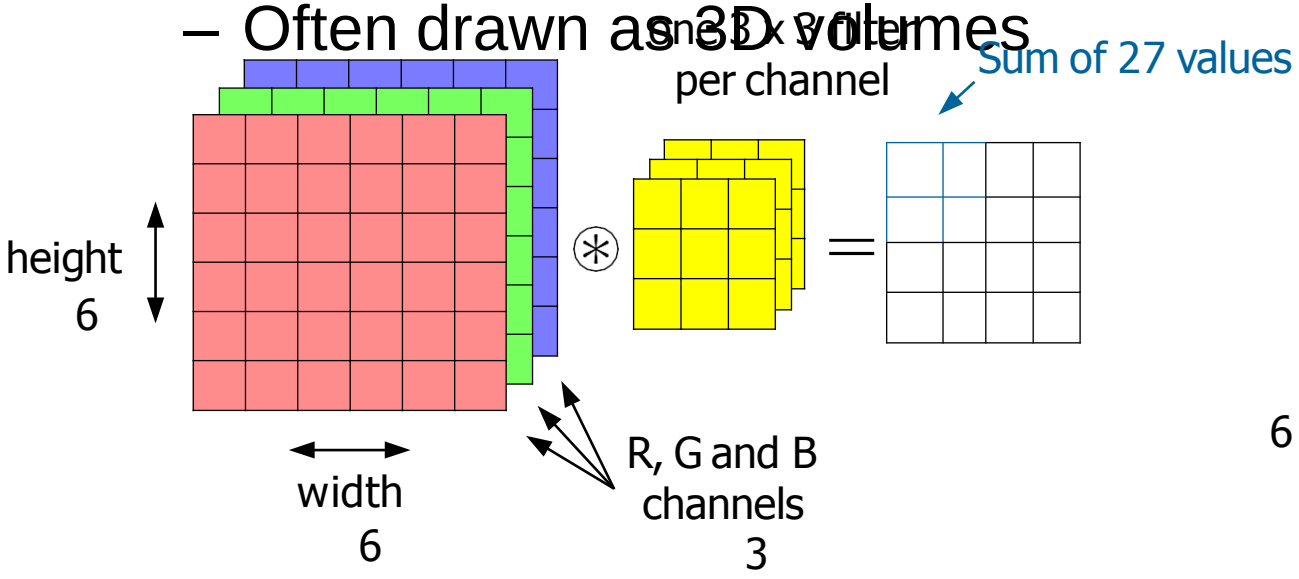


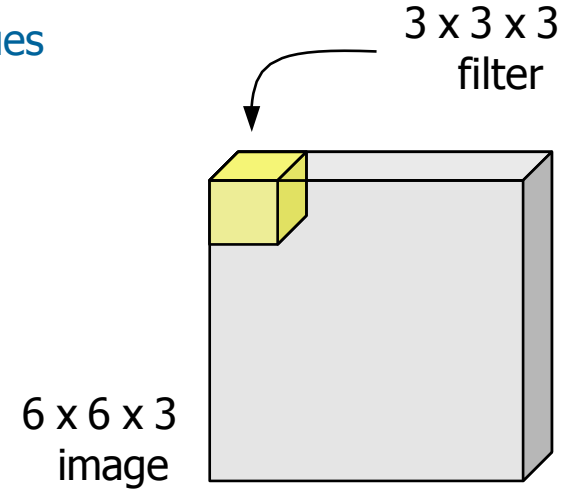
Convolutions over Volumes

- Input images are typically RGB (3 channels)

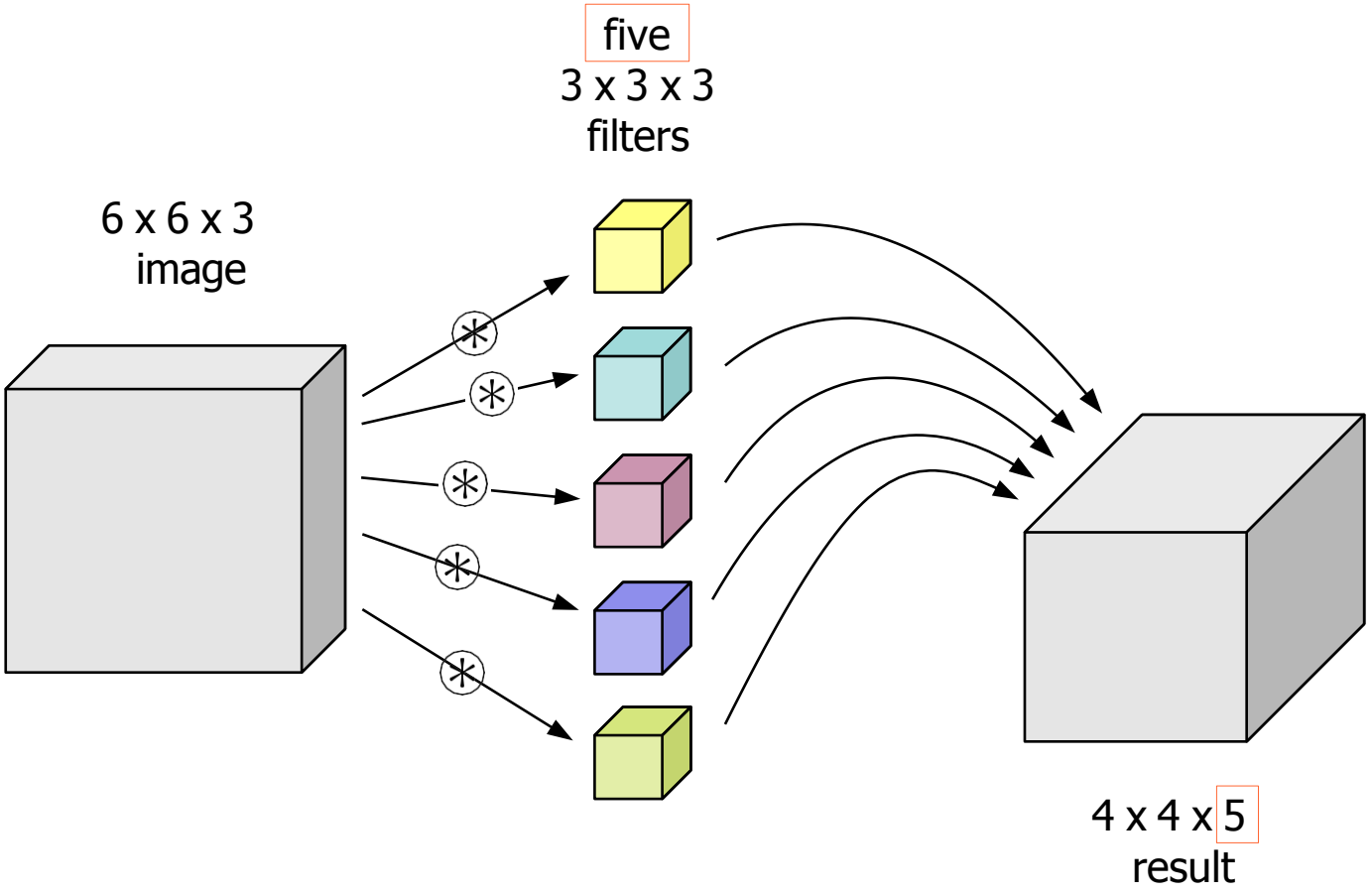
  ➔ volumes (tensors) of shape (width x height x 3)
- Filters should also have 3 channels

  ➔ volumes (tensors) of shape (f x f x 3)
  - Often drawn as 3D volumes





Convolutions over Volumes



No padding, stride 1

Questions ?