

Machine Learning with Neural Networks

GridKa School 2019 – The Art of Data

2019-08-28 | Dr. Markus Götz | KIT-SCC

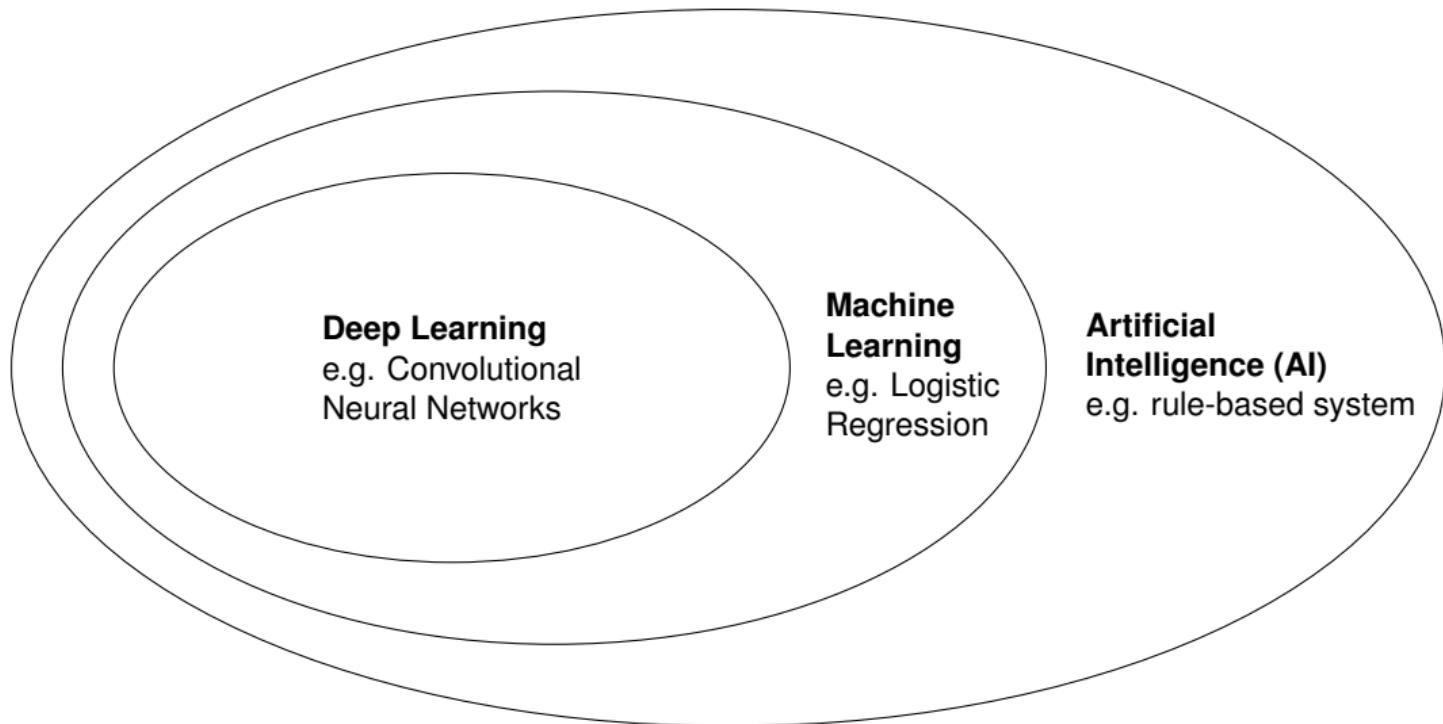


Outline

1. Machine Learning Fundamentals
2. Neural Networks
3. Convolutional Neural Networks
4. Summary
5. Discussion

Machine Learning Fundamentals

Terminology



Why now?

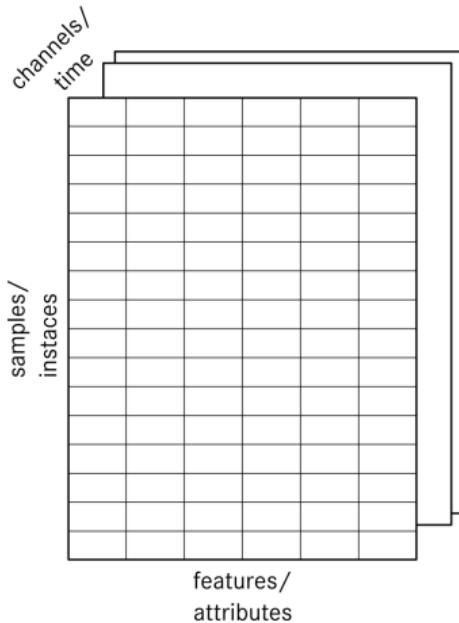
- ▶ **Technology revolution**—vector processors (e.g. GPGPUs), auto-gradient software
- ▶ **Data availability**—large, partially freely available, collections of labeled data
- ▶ **Mathematical advances**—latest addition, investigation of new model elements, e.g. activation functions, normalization

Learning Approaches

- ▶ **Supervised learning:** Learn by “mimicking supervisor”, i.e. pattern annotations
examples: image classification, stock forecasting
- ▶ **Unsupervised learning:** Determine patterns purely based on data
examples: customer cluster analysis, distribution estimation
- ▶ **Reinforcement learning:** Pavlov-style learning with punishment and reward in dynamic environments
examples: game AIs, e.g. AlphaGo or Dota OpenAI

Terminology

- ▶ **Samples** or instances,
individual observations in your data,
e.g. an image, a specimen
- ▶ **Features** or attributes,
single characteristic of a sample,
e.g. a pixel, measured weight
- ▶ **Channels** or time,
depth information,
color channels, change over time



MNIST Dataset

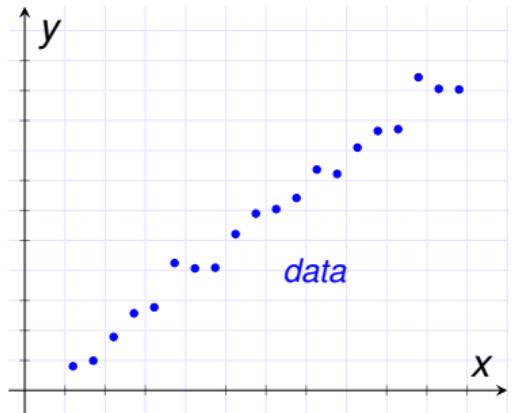
- ▶ **Goal for today:** classification of handwritten digits
- ▶ 70000 images, each 28×28 pixels, gray-scale



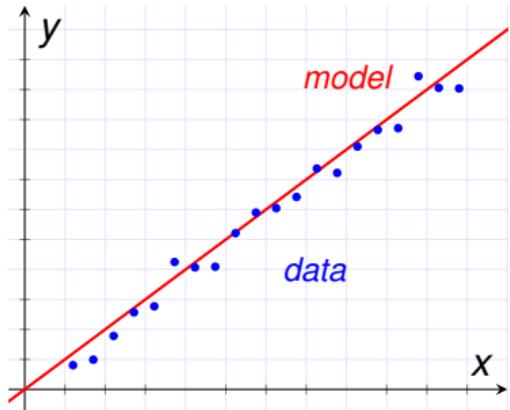
Notation Disclaimer

- ▶ **Small letters:** vectors or matrices, e.g. x or y
- ▶ **Hats:** predictions or estimates, e.g. \hat{y}
- ▶ **Indices:** elements of vectors and matrices, e.g. x_i

Linear Regression

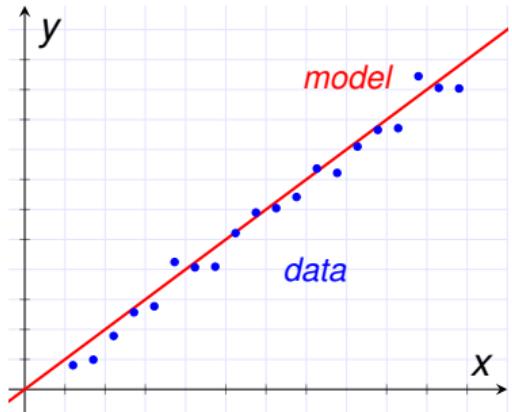


Linear Regression



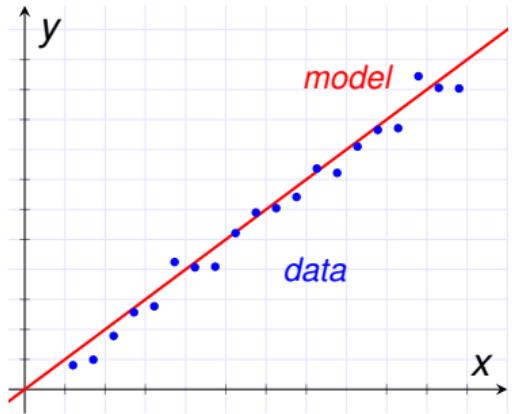
- ▶ **Data set:** $\{samples, labels\} = \{x, y\}$
- ▶ **Model:** definition $\hat{y} = wx + b$
with w and b trainable parameters

Linear Regression



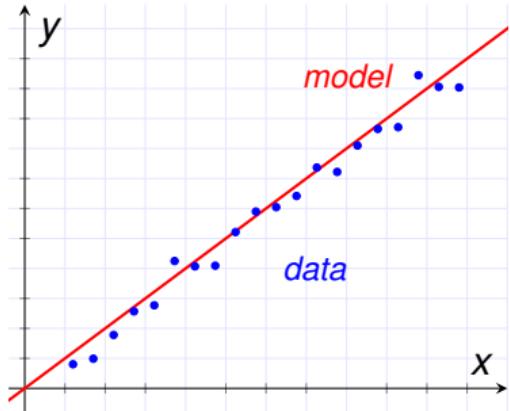
- ▶ **Data set:** $\{samples, labels\} = \{x, y\}$
- ▶ **Model:** definition $\hat{y} = wx + b$
with w and b trainable parameters
- ▶ **Loss function:** or cost/error/objective
$$J(w, b) = MSE(w, b) = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

Linear Regression



- ▶ **Data set:** $\{samples, labels\} = \{x, y\}$
- ▶ **Model:** definition $\hat{y} = wx + b$
with w and b trainable parameters
- ▶ **Loss function:** or cost/error/objective
$$J(w, b) = MSE(w, b) = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$
- ▶ **Train:** the model, e.g. optimization
$$\hat{w}, \hat{b} = \arg \min J(w, b)$$

Linear Regression

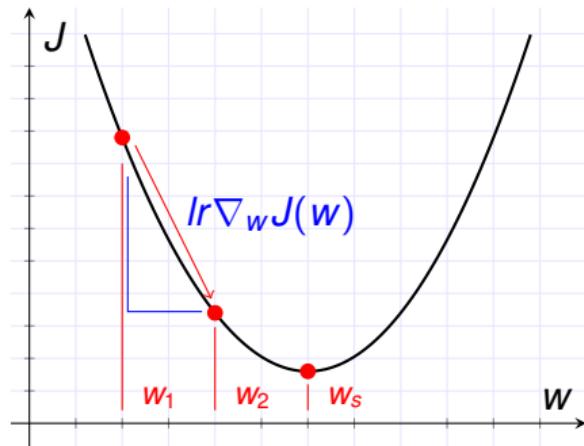
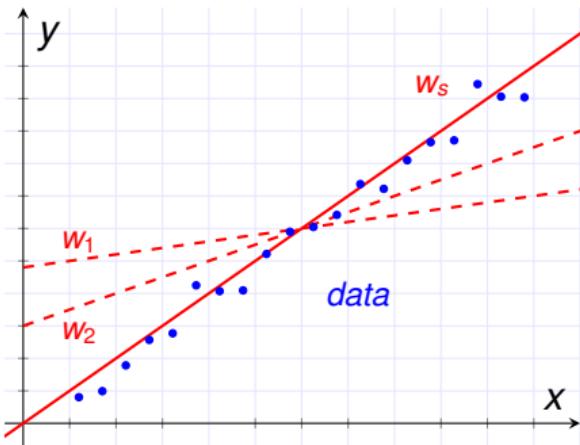


- ▶ **Data set:** $\{samples, labels\} = \{x, y\}$
- ▶ **Model:** definition $\hat{y} = wx + b$
with w and b trainable parameters
- ▶ **Loss function:** or cost/error/objective
$$J(w, b) = MSE(w, b) = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$
- ▶ **Train:** the model, e.g. optimization
$$\hat{w}, \hat{b} = \arg \min J(w, b)$$
- ▶ **Basic recipe for most machine learning algorithms**

Optimization: Gradient Descent

- ▶ Iterative optimization technique, weight update in direction of negative gradient

$$w_{i+1} = w_i - lr \nabla_{w_i} J(w_i)$$



- ▶ lr is learning rate, gradient update factor
- ▶ **Stochastic gradient descent (SGD)**, sample subset (**batch**) updates

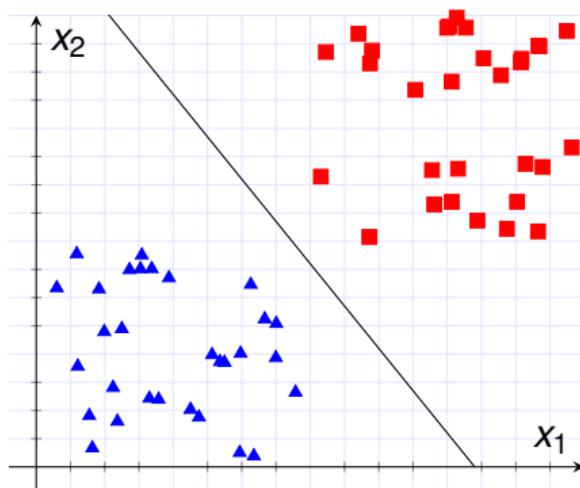
Bias Trick

- ▶ Cumbersome to keep track of weights w and bias b simultaneously
- ▶ **Idea:** fuse both into single weight matrix

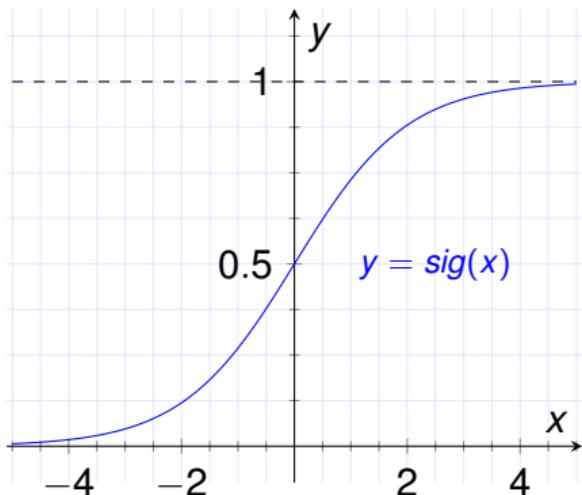
$$\hat{y} = \mathbf{w}x + b \leftrightarrow \hat{y} = \mathbf{w}^*x$$
$$\hat{y} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix}^T \begin{pmatrix} x_{i,1} \\ x_{i,2} \\ \vdots \\ x_{i,n} \end{pmatrix} + b \leftrightarrow \hat{y} = \begin{pmatrix} \mathbf{b} \\ w_1^* \\ w_2^* \\ \vdots \\ w_n^* \end{pmatrix}^T \begin{pmatrix} \mathbf{1} \\ x_{i,1} \\ x_{i,2} \\ \vdots \\ x_{i,n} \end{pmatrix}$$

Pattern Recognition Types

- ▶ **Regression:** predict continuous value, e.g. stock price, $y \in \mathbb{R}$
- ▶ **Classification:** assign sample to a category, e.g. “spam”/“no spam”
special form of regression, where y in fixed interval



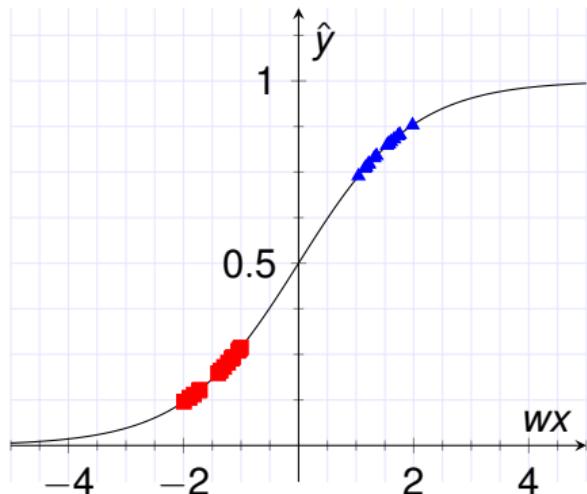
Logistic Regression



- ▶ Squash linear regression output into fixed interval, e.g. $y \in [0, 1]$
- ▶ Interpretation: **probability** of sample belonging to a binary class
- ▶ **sigmoid-logistic** function:
$$\text{sig}(z) = \frac{1}{1+e^{-z}}$$
- ▶ **Model:** $h = \text{sig}(wx) = \frac{1}{1+e^{-wx}}$
- ▶ **Prediction:** $\hat{y} = 1$ if $h \geq 0.5$
 $\hat{y} = 0$ if $h < 0.5$

Logistic Regression

- ▶ Data set must be mapped
 - ▶ ■ → 0
 - ▶ ▲ → 1
- ▶ Model: $h = \text{sig}(wx) = \frac{1}{1+e^{-wx}}$
- ▶ Loss function:
 $J(w) = \text{MSE}(w) = \frac{1}{n} \sum_{i=1}^n (y - \hat{y})^2$
 $\nabla_w J(w) = (\hat{y} - y) * y^2 * e^{-wx} * x$
- ▶ Train: gradient descent optimization



Autograd Frameworks: TensorFlow & Co

- ▶ Numerical and **autograd** libraries
- ▶ Eager and **flow graph** computation
- ▶ Multiple supported devices
CPU, GPU, TPU, smartphone
- ▶ TensorFlow (Google), MXNet
(Amazon), PyTorch (Facebook)
- ▶ **Keras**—neural network wrapper for
TensorFlow and MXNet backends



© Apache

© Keras

Keras an Example

```
from keras.models import Sequential
from keras.layers import Dense

# Logistic regression model with two features
model = Sequential()
model.add(Dense(1, input_dim=2, activation="sigmoid"))

# Model compilation
model.compile(loss="mse", optimizer="sgd")

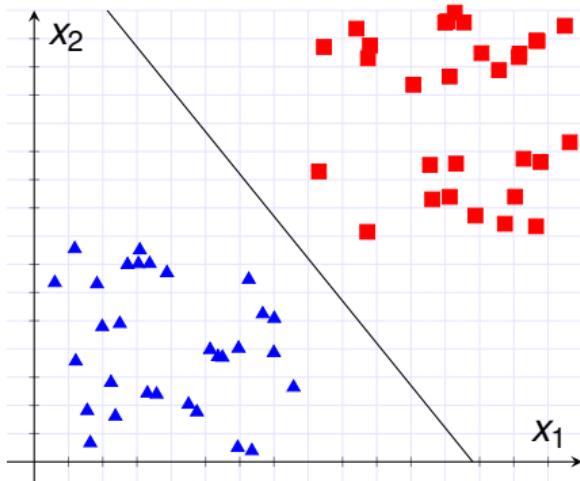
# Fit the model, i.e. optimize J for 10 iterations
model.fit(x, y, epochs=10)
```

Exercise 0: Warm-up

- ▶ Compute on **FH2 GPU** resources
- ▶ Credentials on paper slip
 - ▶ url: <https://gks-ml.scc.kit.edu/>
 - ▶ user: user<xx>
 - ▶ pass: <xxxxxxxx>

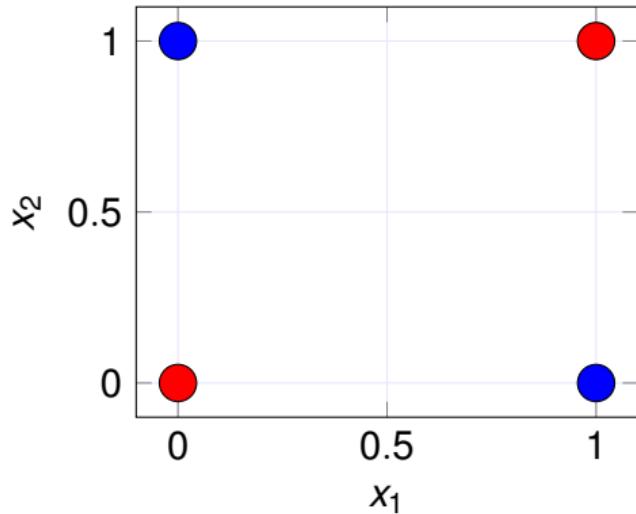
Exercise 1: Logistic Regression

- ▶ Data set must be mapped
 - ▶ ■ → 0
 - ▶ ▲ → 1
- ▶ Model: $h = \text{sig}(wx) = \frac{1}{1+e^{-wx}}$
- ▶ Loss function:
 $J(w) = \text{MSE}(w) = (y - \hat{y})^2$
 $\nabla_w J(w) = (\hat{y} - y) * y^2 * e^{-wx} * x$
- ▶ Train: $w_{i+1} = w_i - lr \nabla_w J(w)$
- ▶ <https://gks-ml.scc.kit.edu/>



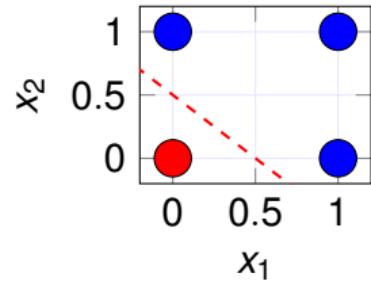
Neural Networks

XOR-Problem



- ▶ Binary exclusive operator, is 1 if one operand is 1, else 0
- ▶ Non-linearly separable, logistic regression cannot model problem
- ▶ **Idea:** decompose into linear problems

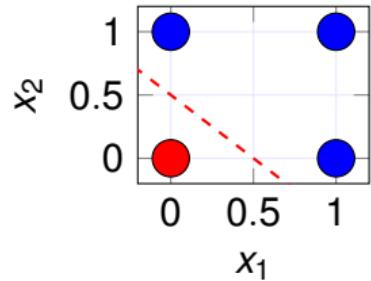
XOR-Problem



OR-gate

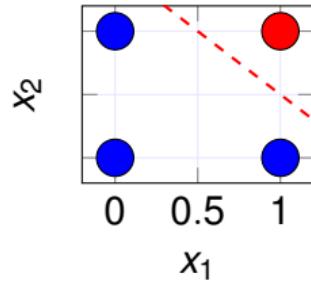
$$h_1 = x_1 + x_2 - 1$$

XOR-Problem



OR-gate

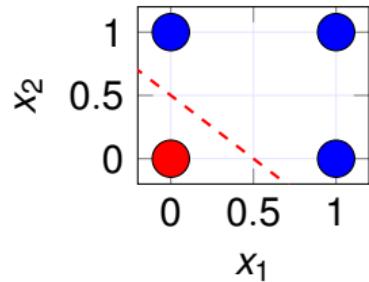
$$h_1 = x_1 + x_2 - 1$$



NAND-gate

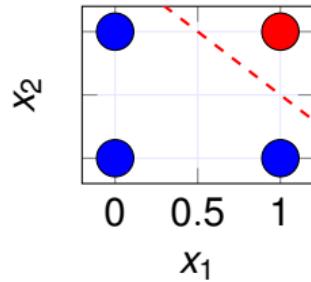
$$h_2 = -x_1 - x_2 - 1.5$$

XOR-Problem



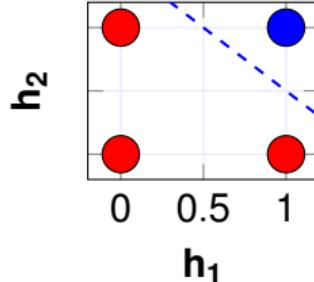
OR-gate

$$h_1 = x_1 + x_2 - 1$$



NAND-gate

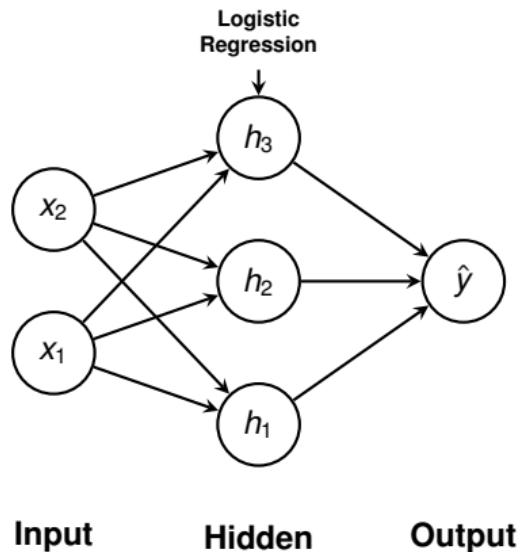
$$h_2 = -x_1 - x_2 - 1.5$$



AND-gate

$$\hat{y} = \mathbf{h}_1 + \mathbf{h}_2 - 1.5$$

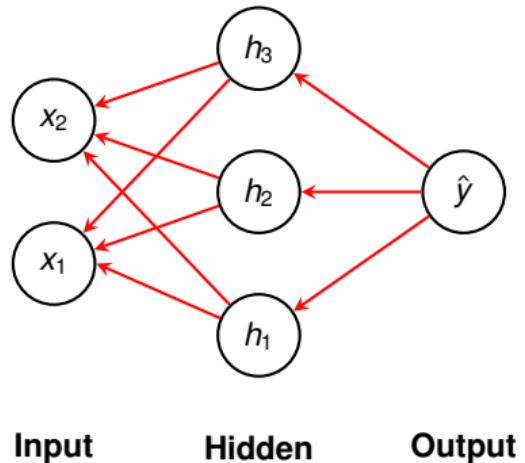
Fully-connected Neural Network



- ▶ Inspired by biological neural network
- ▶ A **neuron** is a logistic regression
- ▶ Neurons are arranged in **layers**
- ▶ Layers are **fully-connected** with subsequent layer, also called **Dense**
- ▶ **Width:** neuron count
- ▶ **Depth:** layer count

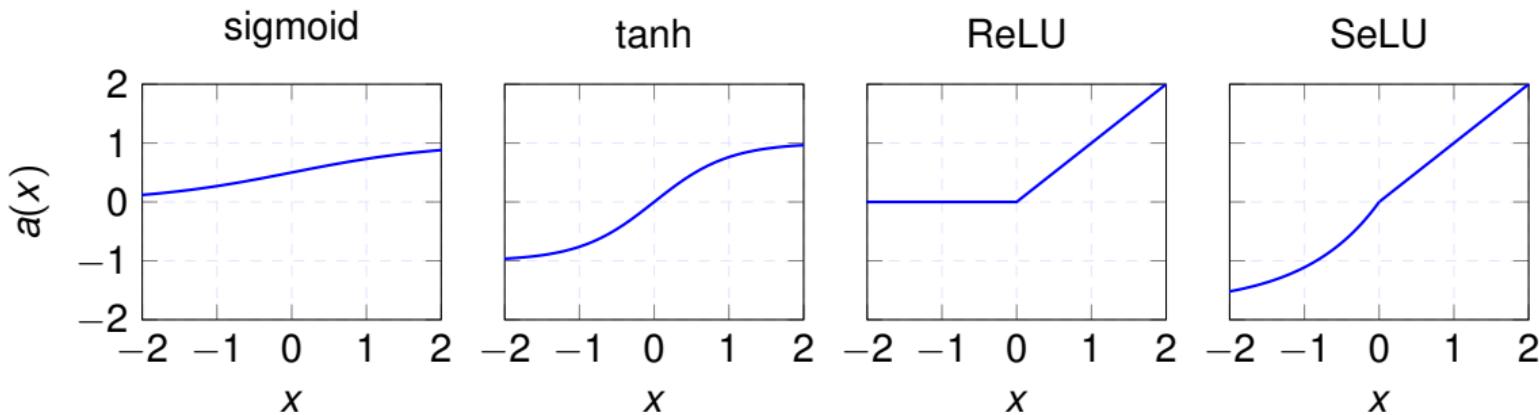
Backpropagation

- ▶ Alternate forward and backward pass
- ▶ **Hidden layer** are nested functions
 - ▶ Requires **chain rule** for gradient
 - ▶ $h'(x) = f'(g(x)) * g'(x)$
 - ▶ **Neurons store forward result**
- ▶ Weight initialization in network
small random numbers
- ▶ Iterations are now called **epochs**



Activation Functions

- ▶ Activation functions $a(x)$ introduce **non-linearity**, e.g. sigmoid function
- ▶ Other non-linear choices, e.g. $\tanh(x)$, $\text{relu}(x) = \max(0, x)$, etc.
- ▶ Better computational properties, e.g. avoid **vanishing gradient**



Universal Approximation Theorem

A feed-forward neural network with a linear output and at least one hidden layer can approximate any reasonable function to arbitrary precision with a finite number of nodes.

Universal Approximation Theorem

A feed-forward neural network with a linear output and at least one hidden layer can approximate any reasonable function to arbitrary precision with a finite number of nodes.

- ▶ **Good News**

- ▶ Networks can perform highly complex tasks
- ▶ All necessary ingredients available

Universal Approximation Theorem

A feed-forward neural network with a linear output and at least one hidden layer can approximate any reasonable function to arbitrary precision with a finite number of nodes.

- ▶ **Good News**

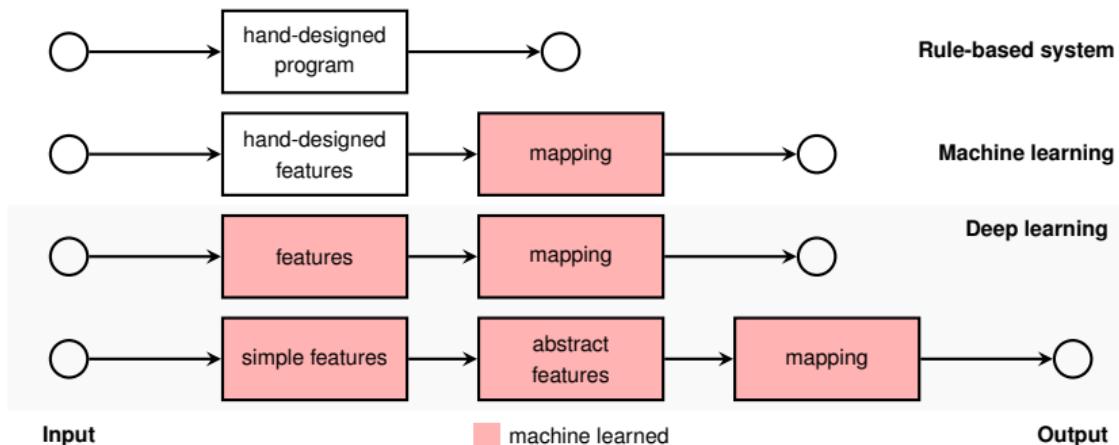
- ▶ Networks can perform highly complex tasks
- ▶ All necessary ingredients available

- ▶ **Bad News**

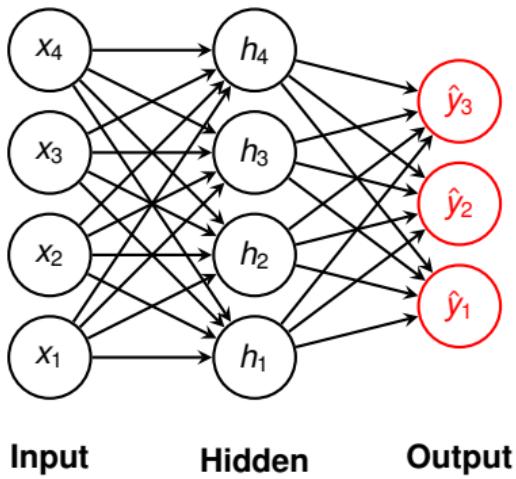
- ▶ Does not specify number of necessary nodes
- ▶ No remarks on neuron connectivity

Deep Learning

- ▶ In practice: **stacking layers** works better
- ▶ **Deep learning:** more than one stage of non-linearities, e.g. layers



Multi-class Classification



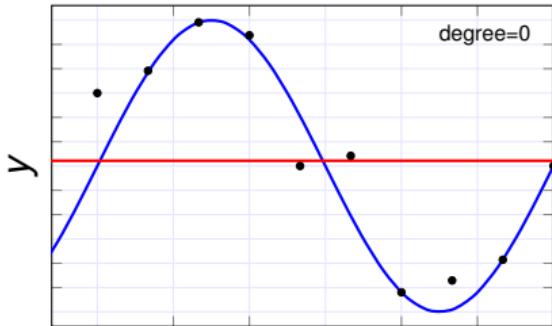
- ▶ Extension of binary classification concept
- ▶ **One-versus-all classification**
 - ▶ Build c binary classifiers
 - ▶ Pick class with highest confidence/probability
- ▶ In neural networks
 - ▶ Create multiple networks
 - ▶ **Add output neurons**

Multi-class Classification

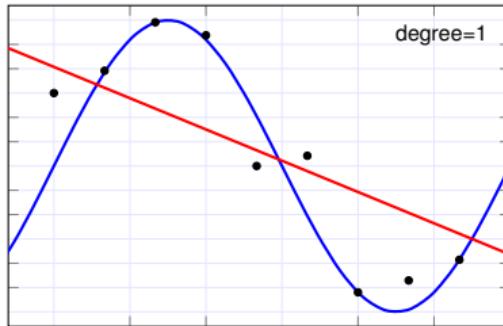
Multi-class classification recipe:

- ▶ **One-hot class encoding:** encode classes as sparse vectors
 $y = (y_1, y_2, \dots, y_c)$, only one is active, e.g. class 2 $\rightarrow (0, 1, \dots, 0)$
- ▶ **Softmax output activation:** $\hat{y} = softmax(z) = \frac{e^{z_j}}{\sum_j e^{z_j}}$ for $j = 1 \dots c$
achieve joint-probability of 1, normalize across model outputs z
- ▶ **Cross-entropy loss:** convex-function $J(w) = \frac{1}{n} \sum_{i=1}^n \sum_j^c y_{i,j} \log \hat{y}_{i,j}$
maximum likelihood principle

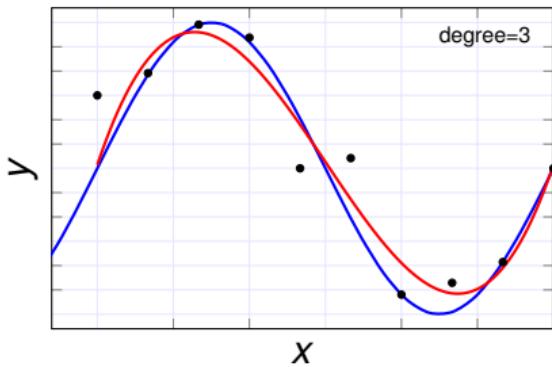
Over- and Underfitting



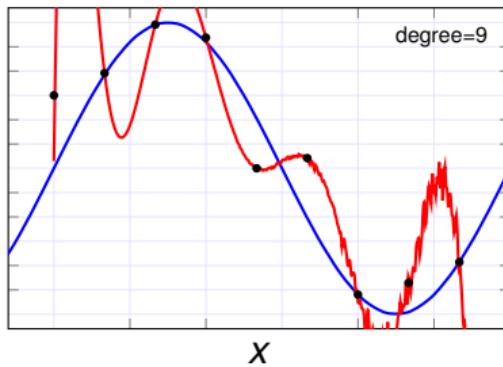
degree=0



degree=1



degree=3

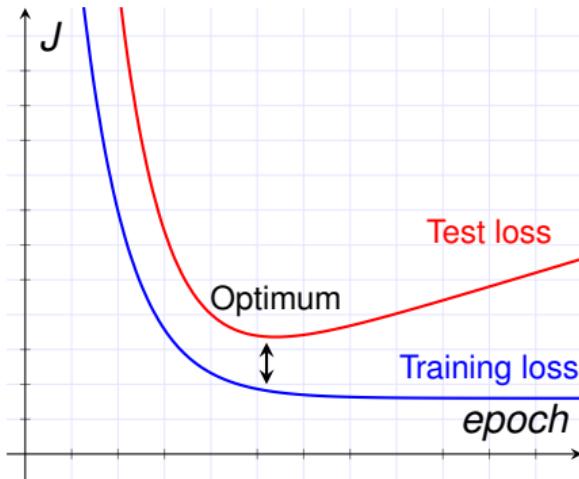


degree=9

Over- and Underfitting

- ▶ How do we know a network is not over- or underfitting?
- ▶ **Idea:** simulate “unseen” data
- ▶ Split data artificially into disjoint subsets
 - ▶ **Training set** for training the model (usually 60% – 80%)
 - ▶ **Validation set** for fine tuning the model (usually 20%)
 - ▶ **Test set** to test validation (usually 20% – 40%)

Over- and Underfitting



- ▶ Separate monitoring of training and test loss during training
- ▶ Training loss will decrease indefinitely $J \rightarrow 0$, **memorization effect**
- ▶ **Test loss minimum** is optimal

Regularization

Methods to avoid overfitting, reduced validation error, might increase train error

- ▶ More data
- ▶ Augment: generate artificially new samples (add noise, translate, ...)
- ▶ Early stopping: monitoring of train and test loss, stop at optimum
- ▶ Penalize large weights: add a penalty term
- ▶ Dropout neurons
- ▶ Batch Normalization neurons

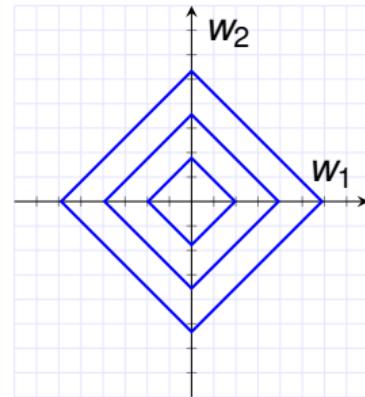
Penalty Terms

- ▶ Weights can become increasingly large, allowing overfitting
- ▶ **Idea:** penalize large weights
- ▶ Minimize function $J^*(w) = J(w) + \lambda\Omega(w)$
- ▶ $\Omega(w)$ is measure of weight magnitude
- ▶ λ is scale for $\Omega(w)$
 - ▶ $J(w) \gg \lambda\Omega(w)$ —no regularization
 - ▶ $J(w) \ll \lambda\Omega(w)$ —no training

L1- and L2-Norm

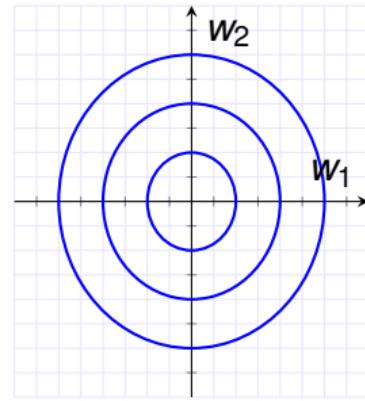
► L1-Norm

- ▶ $\|w\|_1 = (w_1 + w_2 + \dots + w_f)$
- ▶ All weights contribute to loss
- ▶ Encourages sparsity



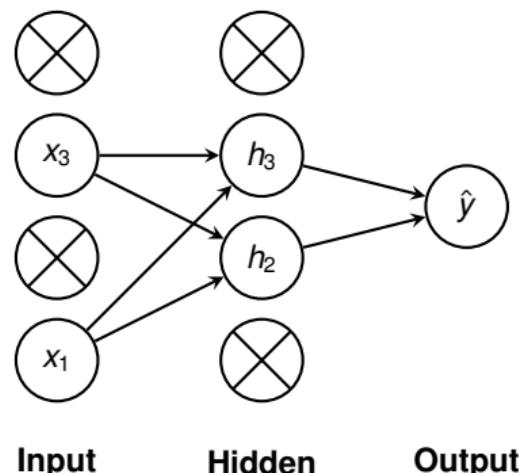
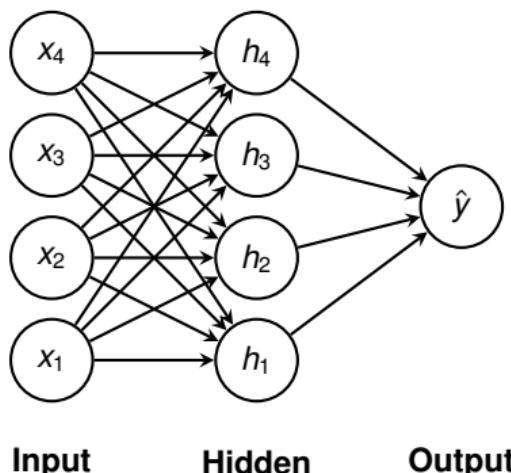
► L2-Norm, also **weight decay**

- ▶ $\|w\|_2^2 = (w_1^2 + w_2^2 + \dots + w_f^2)$
- ▶ Penalizes large weights
- ▶ Discourages sparsity



Dropout

- ▶ Randomly turn off neurons and connection, e.g. $p(\text{drop}) = 0.3$
- ▶ Equivalent to network regularization (proof omitted)
- ▶ Implementation: set passed input to zero



Neural Networks in Keras

```
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.regularizers import l2

classes, features = 10, 200

# Fully-connected neural network
model = Sequential()
model.add(Dense(16, input_dim=features, activation="tanh"))
model.add(Dense(16, kernel_regularizer=l2(0.02)))
model.add(Dropout(0.1))
model.add(Dense(classes, activation='softmax'))

# Model compilation
model.compile(loss="categorical_crossentropy", optimizer="sgd")
# Fitting
model.fit(x, y, batch_size=2, epochs=10)
```

Exercise 2: FNN MNIST Image Classification



Convolutional Neural Networks

Computer Vision



© Catster

- ▶ Easy for humans, hard for machines
- ▶ High input dimensionality $\mathbb{R}/\mathbb{N}^{10^1-10^8}$
- ▶ Have to deal with POV-shifts, light variation, occlusion, shape variation
- ▶ Classical computer vision answer:
 - ▶ Inclusion of context information
 - ▶ Usage of image filters
- ▶ **Idea:** combine with machine learning

Discrete Convolution

- ▶ Element-wise weighted sum of input and filter
- ▶ $(f * g)[n] = \sum_{m=-K}^K f[m]g[n-m]$
- ▶ **Filter size K:** window size of convolution kernel
- ▶ **Stride:** pixel distance for slide
- ▶ 2D input: volume of $width \times height (\times channels)$
- ▶ Models effects on images, e.g. edge detection
- ▶ In CNN: model “eye”, sparse weight sharing



Discrete Convolution

0	0	0	0	0	0	
0	105	102	100	97	96	
0	103	99	103	101	102	
0	101	98	104	102	100	
0	99	101	106	104	99	
0	104	104	104	100	98	

Image Matrix

0	-1	0
-1	5	-1
0	-1	0

Kernel Matrix

320				

Output Matrix

$$\begin{aligned} & 0 * 0 + 0 * -1 + 0 * 0 \\ & + 0 * -1 + 105 * 5 + 102 * -1 \\ & + 0 * 0 + 103 * -1 + 99 * 0 = 320 \end{aligned}$$

Convolution with horizontal and vertical strides = 1

© Machine Learning Guru

Discrete Convolution

0	0	0	0	0	0	
0	105	102	100	97	96	
0	103	99	103	101	102	
0	101	98	104	102	100	
0	99	101	106	104	99	
0	104	104	104	100	98	

Image Matrix

0	-1	0
-1	5	-1
0	-1	0

Kernel Matrix

320	206			

Output Matrix

$$\begin{aligned} & 0 * 0 + 0 * -1 + 0 * 0 \\ & + 105 * -1 + 102 * 5 + 100 * -1 \\ & + 103 * 0 + 99 * -1 + 103 * 0 = 206 \end{aligned}$$

Convolution with horizontal and vertical strides = 1

© Machine Learning Guru

Discrete Convolution

0	0	0	0	0	0	
0	105	102	100	97	96	
0	103	99	103	101	102	
0	101	98	104	102	100	
0	99	101	106	104	99	
0	104	104	104	100	98	

Image Matrix

0	-1	0
-1	5	-1
0	-1	0

Kernel Matrix

320	206	198		

Output Matrix

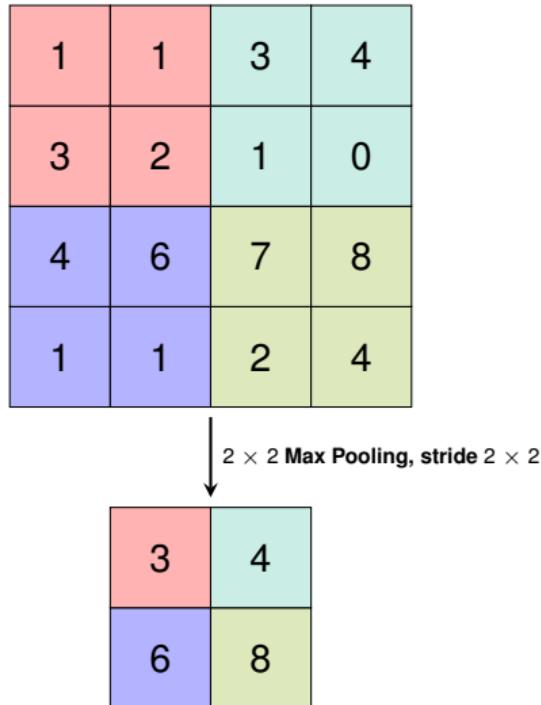
$$\begin{aligned} & 0 * 0 + 0 * -1 + 0 * 0 \\ & + 102 * -1 + 100 * 5 + 97 * -1 \\ & + 99 * 0 + 103 * -1 + 101 * 0 = 198 \end{aligned}$$

Convolution with horizontal and vertical strides = 1

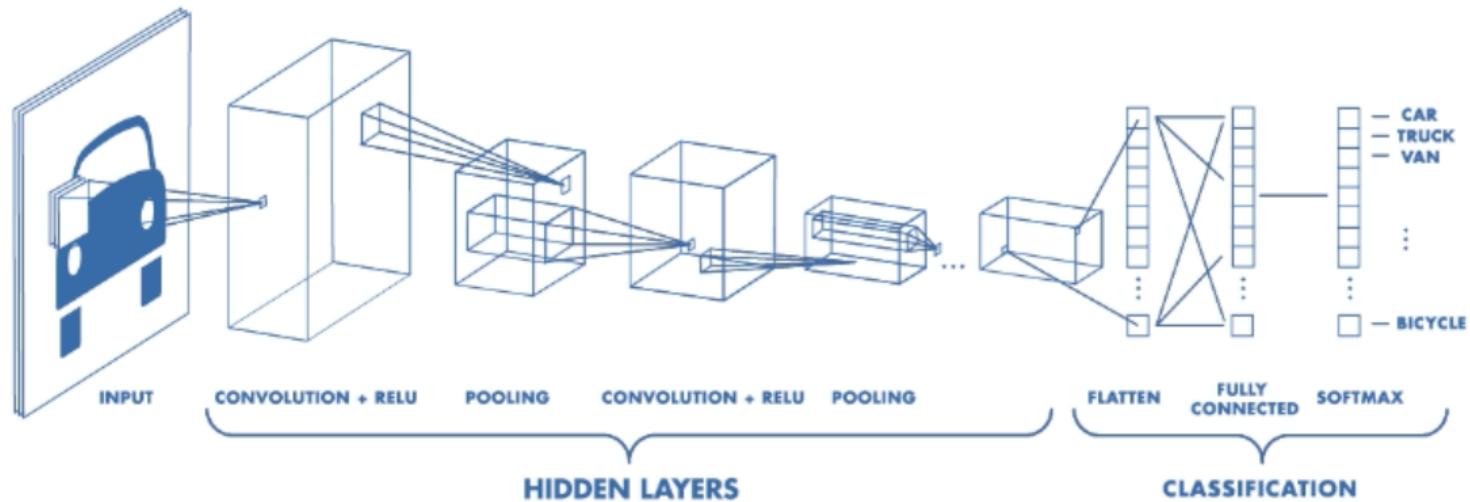
© Machine Learning Guru

Pooling

- ▶ **Pooling** reduces input sizes, abstract downsampled copy
- ▶ **Pool size:** kernel height/width
- ▶ **Strides:** step width
- ▶ Typical pooling layers
 - ▶ Max Pooling
 - ▶ Average Pooling

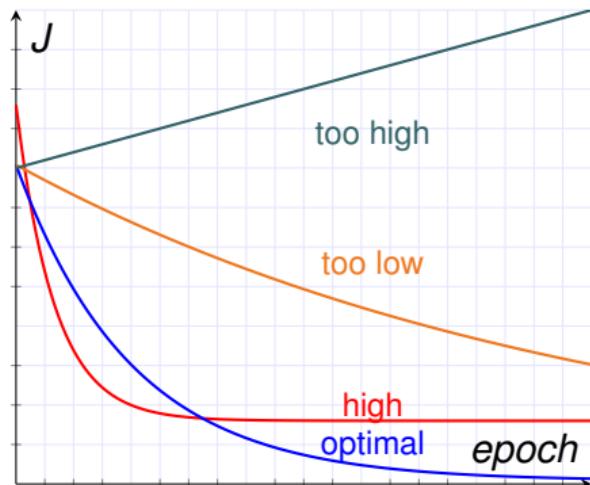


Convolutional Neural Network Pyramid



© Medium

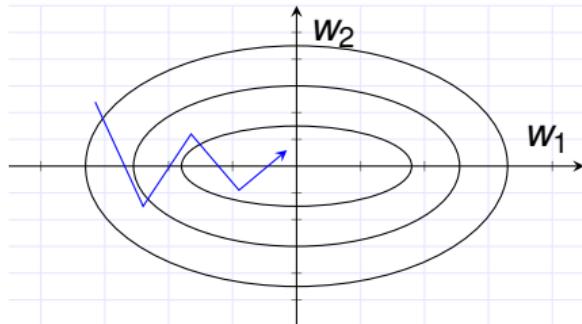
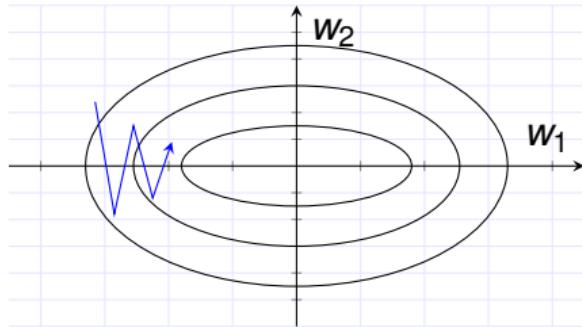
Optimizers



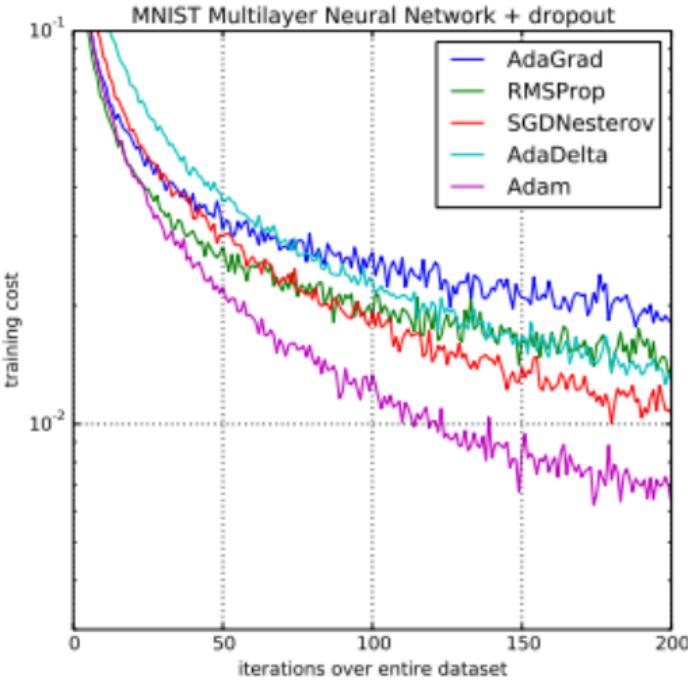
- ▶ Learning rate strongly impacts training
- ▶ Value for lr
 - ▶ high: jumpy, no convergence
 - ▶ low: slow training, local minima
- ▶ Earlier approach
 - ▶ stop training every e epochs
 - ▶ lower lr
 - ▶ continue training

Optimizers

- ▶ Various flavors of SGD
- ▶ Include **gradient momentum**
 - ▶ Accelerate previous gradient
 - ▶ $J_{t+1}^*(w) = \alpha J_t^*(w) - lr \nabla J_t(w)$
- ▶ **Adaptive learning rate**
 - ▶ Implemented in **Adagrad**
 - ▶ Learning rate lr_i for each weight w_i
 - ▶ $w_{i+1} = w_i \frac{lr_{t,i}}{\sqrt{\sum_{i=1}^{\infty} \nabla_{w_i} J_{t-i}(w_i)^2}} \nabla_{w_i} J_t(w_i)$



Optimizers



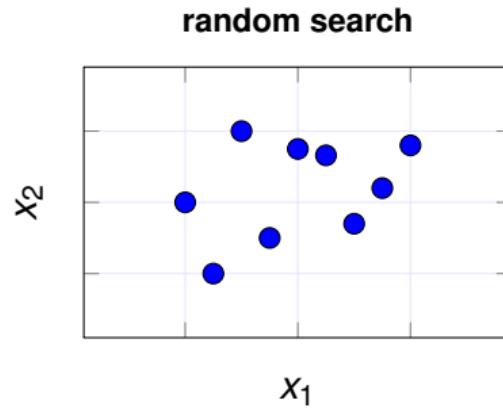
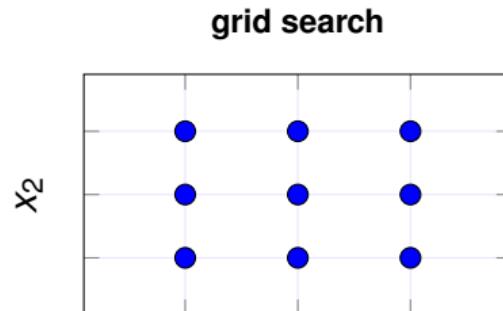
- ▶ **RMSprop:** enhanced Adagrad
 - ▶ Learning rate got infinitely small
 - ▶ Forgetting factor β for past gradients
- ▶ More intricate variants: Nesterov momentum, Adadelta, Nadam, ...
- ▶ Some **non-SGD** alternatives
 - ▶ Particle-swarm optimization (PSO)
 - ▶ BFGS

Hyperparameter Optimization

- ▶ **Hyperparameters are all non-weight parameters**
 - ▶ Optimization algorithm
 - ▶ Learning rate
 - ▶ Regularization
 - ▶ Network layer count
 - ▶ Network neurons
 - ▶ ...
- ▶ Some can be inferred through thought
- ▶ **Rest: trial and error**

Hyperparameter Optimization

- ▶ Naïve approaches
 - ▶ Grid search
 - ▶ **Random search**
- ▶ **Search** algorithms
 - ▶ Kriging-based (hyperopt, Vizier)
 - ▶ Particle-swarm optimization
 - ▶ Genetic algorithms
- ▶ **Meta-learning:** use NN to find NN



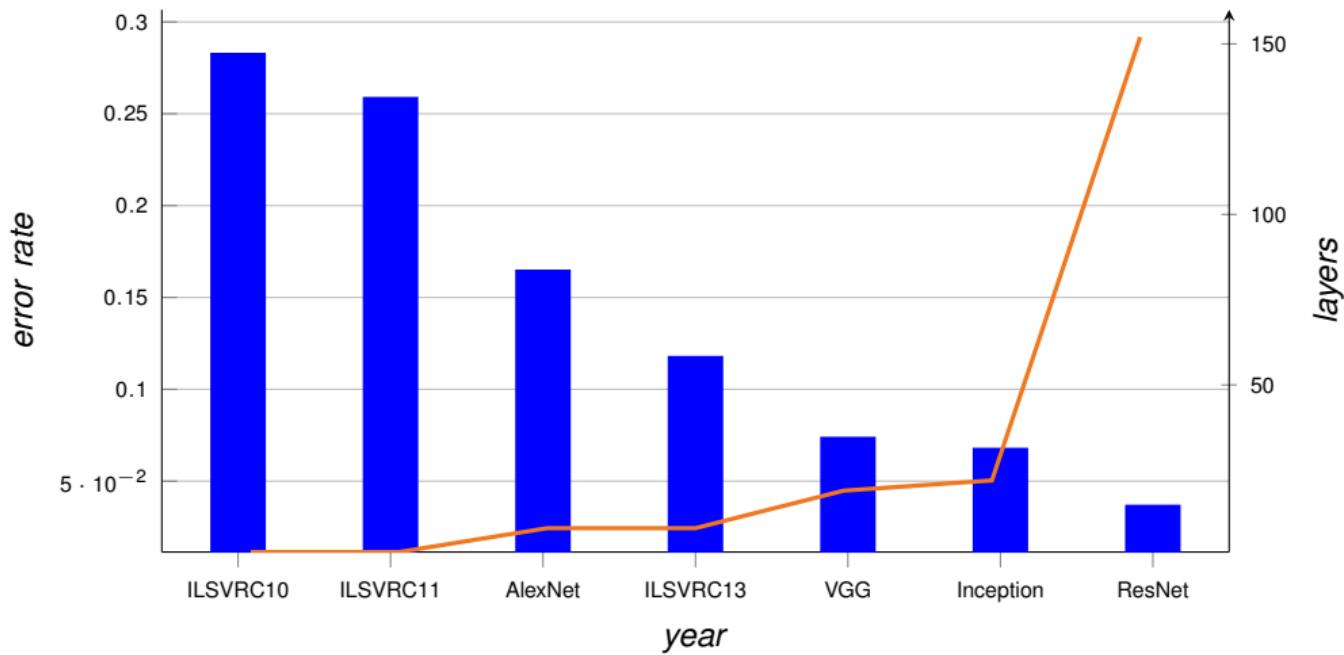
ImageNet Large Scale Visual Recognition Challenge (ILSVRC)



© Oren Kraus

- ▶ **Main image recognition benchmark**
- ▶ Natural image collection, 14m samples
- ▶ Presented in 2009
- ▶ Since 2010, annual prediction competition
 - ▶ Image classification
 - ▶ Object localization
 - ▶ Scene detection

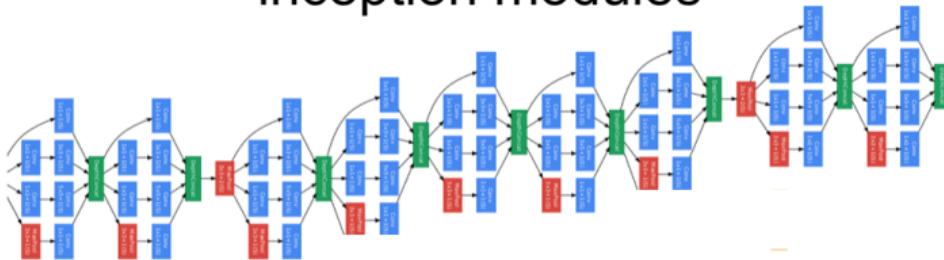
ILSVRC Results



Inception

inception modules

stem



output
classifier

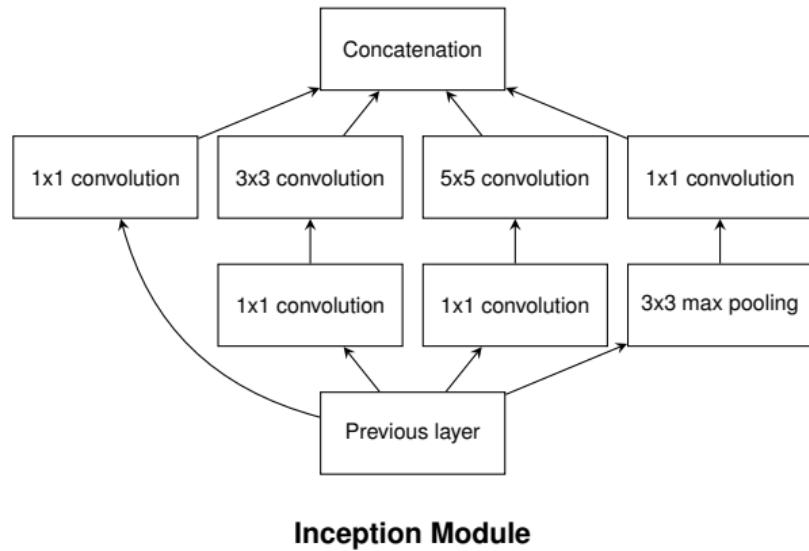


auxiliary classifiers

© Joe Marino

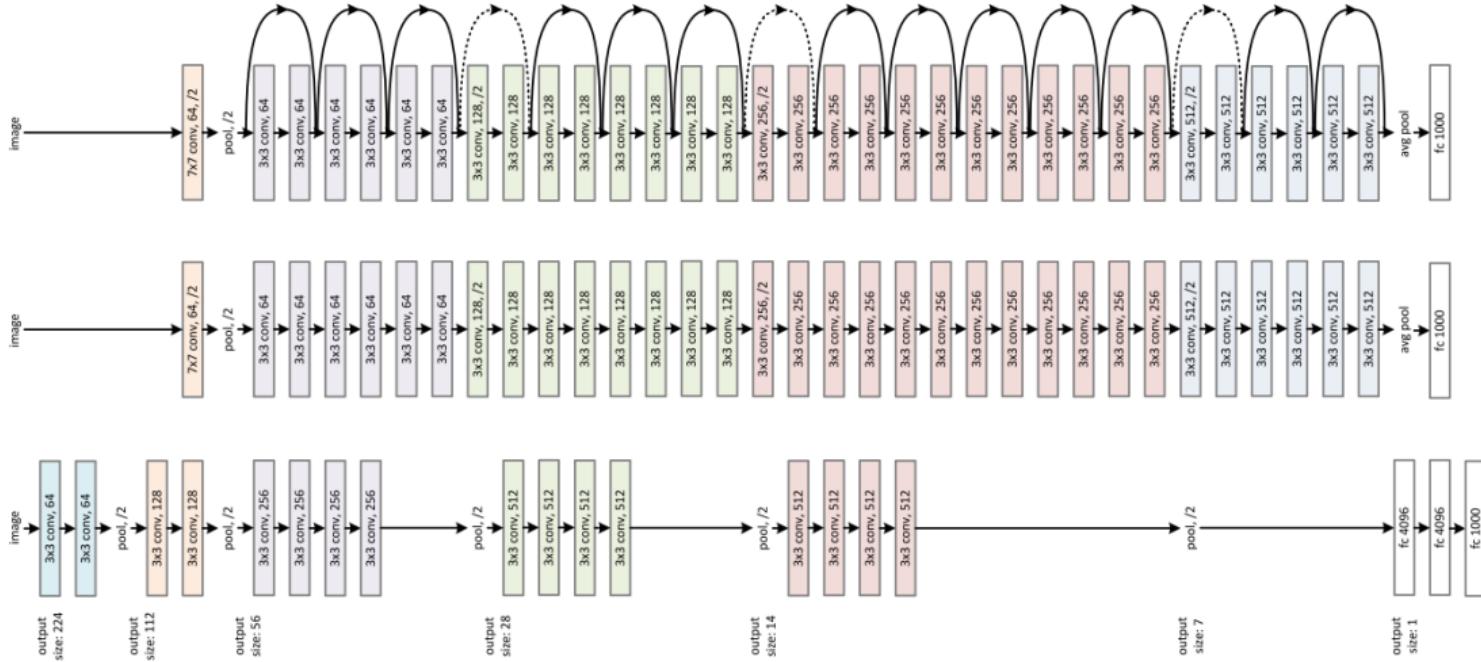
Inception

- ▶ Winner of ILSVRC 2014
- ▶ Repeating same kernels
- ▶ **Idea:** repeating Inception module
- ▶ Core concept
 - ▶ Cheap non-linearities (1×1 convolution)
 - ▶ Different resolution scales



ResNet

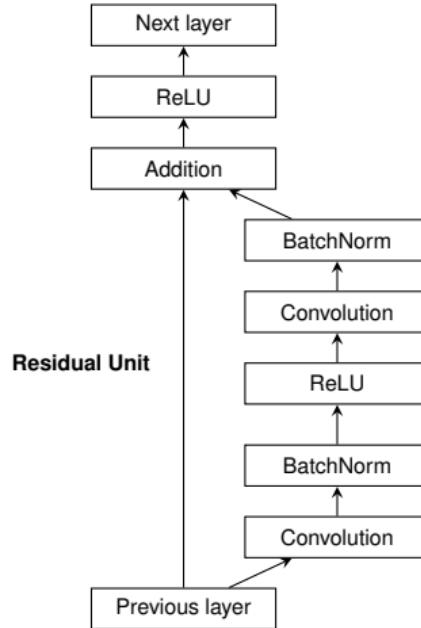
VGG-19 34-layer plain 34-layer residual



© Medium

Residual Unit

- ▶ Winner of ILSVRC 2015
- ▶ Ultra deep network (152 layers)
- ▶ **Idea:** nested mini networks
 - ▶ Learn residuals and add to input
 - ▶ Unused units do nothing
- ▶ First time, humans have been surpassed



Deep Learning Issues

- ▶ **GPU memory size:** forward and backward passes need to be stored
network parameters need to be stored
Counter by decreasing batch sizes, use multiple GPUs
- ▶ **Vanishing and shattered gradients:** gradients become too small or noise the deeper you stack the network
- ▶ **Debugging:** almost impossible to understand what each components does
Some tools available: Influence functions, layer output investigation

CNNs in Keras

```
from keras.models import Sequential
from keras.layers import BatchNormalization, Conv2D,
                      Dense, MaxPooling2D
from keras.regularizers import l2

# Convolutional neural network
w, h, c = 128, 128, 1 # width, height, channels
model = Sequential()
model.add(Conv2D(16, (5, 5), padding="same", input_shape=(w, h, c)))
model.add(MaxPooling2D((2, 2), strides=(1, 1)))
model.add(BatchNormalization())
model.add(Dense(128))
model.add(Dense(classes, activation='softmax'))

# Model compilation
model.compile(loss="categorical_crossentropy", optimizer="sgd")
# Fitting
model.fit(x, y, batch_size=2, epochs=10)
```

Exercise 3: CNN MNIST Image Classification



Summary

Summary

- ▶ **Supervised machine learning**
 - ▶ Logistic regression
 - ▶ Fully-connected neural networks (**FNN**)
 - ▶ Convolution neural networks (**CNN**)
- ▶ Network components
 - ▶ Activation functions
 - ▶ Regularization
 - ▶ Optimizers
- ▶ Application scenarios, **regression** and **classification**

What's more?

- ▶ **Data augmentation:** artificial data increase through rotation, scaling, translation, etc. to better abstract patterns and increase data set size
- ▶ **Embedding:** lower-dimensional representation of (sparse) input data
- ▶ **Capsule Networks:** hierarchy and translation-aware neural networks
- ▶ **Recurrent Neural Networks (RNN):** sequence-learning neural networks, e.g natural language processing and time series analysis
- ▶ **Attention:** learning “where to look”, e.g. for natural language translation

Acknowledgment

- ▶ **René Caspart**

- ▶ GridKa School organization
- ▶ Paperwork



- ▶ **Oskar Taubert**

- ▶ Assignment preparation
- ▶ Exercise supervision



- ▶ **Samuel Braun**

- ▶ Access to FH2
- ▶ Jupyter support

Discussion