Machine Learning with Neural Networks

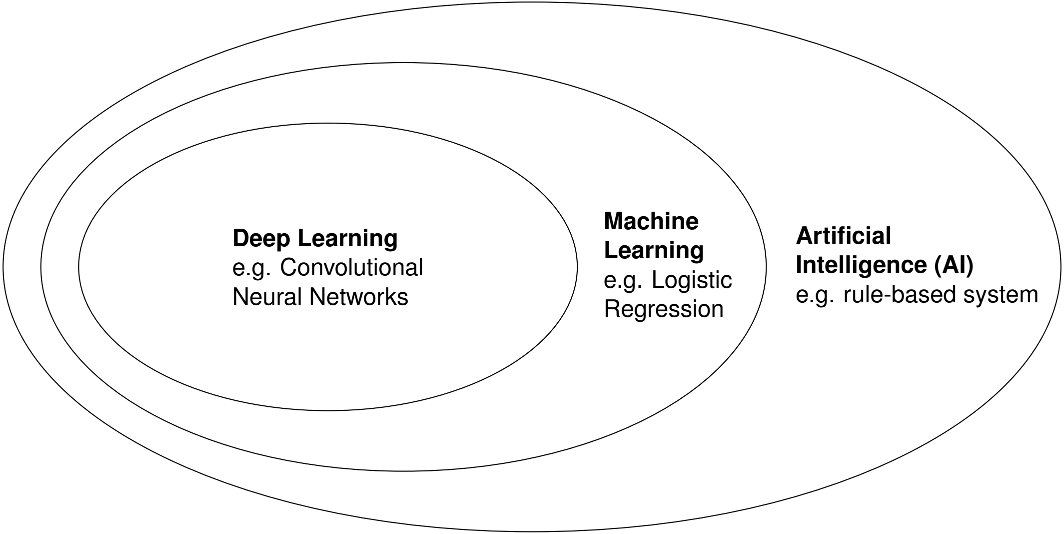
GridKa School 2018

2018-08-29 | Markus Götz | KIT

Outline

Machine Learning Fundamentals

Terminology



Why now?

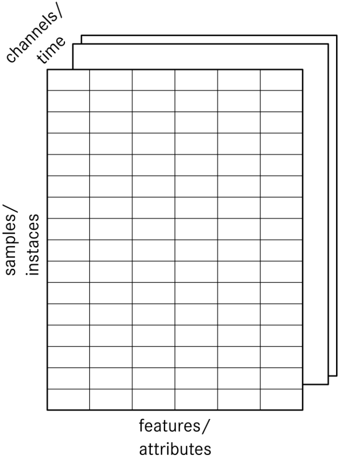
- **Technology revolution**—vector processors (e.g. GPGPUs), auto-gradient software
- **Data availability**—large, partially freely available, collections of labeled data
- **Mathematical advances**—latest addition, investigation of new model elements, e.g. activation functions, normalization

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



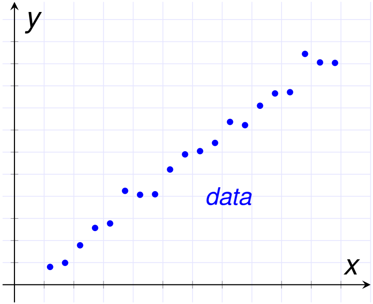
MNIST Dataset

- **Goal for today:** classification of handwritten digits
- 70000 images, each 28 $\times$ 28 pixels, gray-scale

Notation Disclaimer

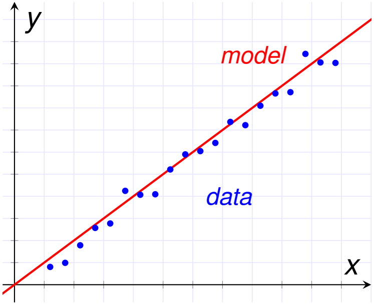
- **Small letters:** vectors or matrices, e.g. x or y
- **Hats:** predictions or estimates, e.g. $\hat{y}$
- **Indices:** elements of vectors and matrices, e.g. x_i

Linear Regression



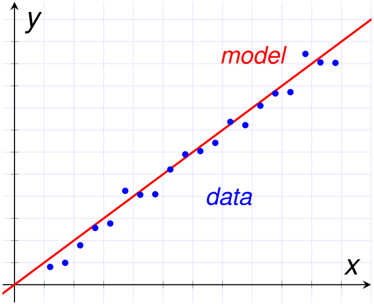
- **Data set:** $\{samples, labels\} = \{x, y\}$

Linear Regression



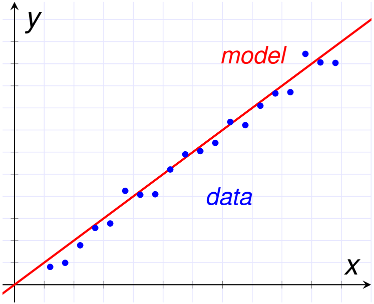
- **Data set:** $\{samples, labels\} = \{x, y\}$
- **Model:** definition $\hat{y} = wx + b$
  with w and b trainable parameters

Linear Regression



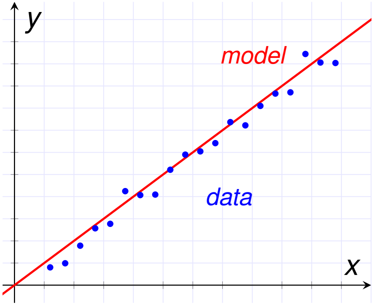
- **Data set:** $\{samples, labels\} = \{x, y\}$
- **Model:** definition $\hat{y} = wx + b$
  with w and b trainable parameters
- **Loss function:** or cost/objective
  $J(w, b) = MSE(w, b) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$

Linear Regression



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



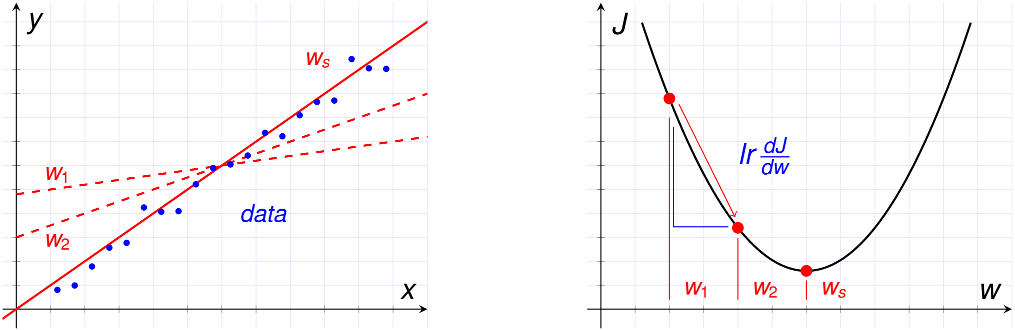
- **Data set:** $\{samples, labels\} = \{x, y\}$
- **Model:** definition $\hat{y} = wx + b$
  with w and b trainable parameters
- **Loss function:** or cost/objective
  $J(w, b) = MSE(w, b) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$
- **Train:** the model, e.g. optimization
  $\hat{w}, \hat{b} =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

- **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

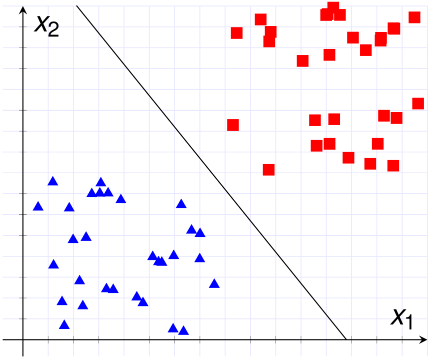
- Cumbersome to keep track of weights w and bias b
- **Idea:** fuse both into single weight matrix

$$\hat{y} = wx + b \leftrightarrow \hat{y} = wx$$

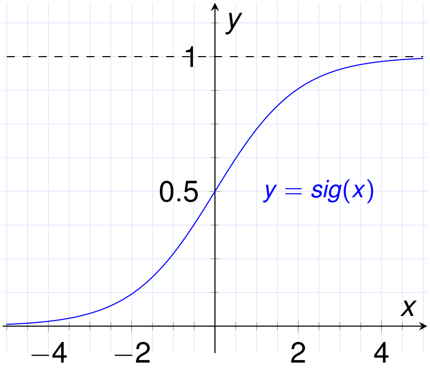
$$\hat{y} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix}' \begin{pmatrix} x_{i,1} \\ x_{i,2} \\ \vdots \\ x_{i,n} \end{pmatrix} + b \leftrightarrow \hat{y} = \begin{pmatrix} \mathbf{b} \\ w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix}' \begin{pmatrix} \mathbf{1} \\ x_{i,1} \\ x_{i,2} \\ \vdots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

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



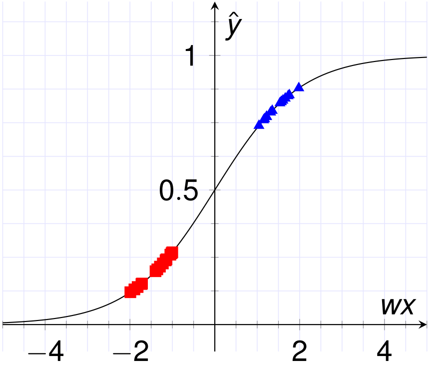
Logistic Regression



- 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:
  $sig(z) = \frac{1}{1+e^{-x}}$
- **Model:** $h = sig(wx) = \frac{1}{1+e^{-wx}}$
- **Prediction:** $\hat{y} = 1 \text{ if } h \geq 0.5$
  $\hat{y} = 0 \text{ if } h < 0.5$

Logistic Regression

- **Data set** must be mapped
  - ■ $\rightarrow 0$
  - ▲ $\rightarrow 1$
- **Model:** $h = sig(wx) = \frac{1}{1+e^{-wx}}$
- **Loss function:**
  $J(w) = MSE(w) = \frac{1}{n}\sum_{i=1}^{n}(y - \hat{y})^2$
  $\frac{dJ}{dw} = (\hat{y} - y) * (\hat{y} - \hat{y}^2) * x$
- **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

© Google

© PyTorch

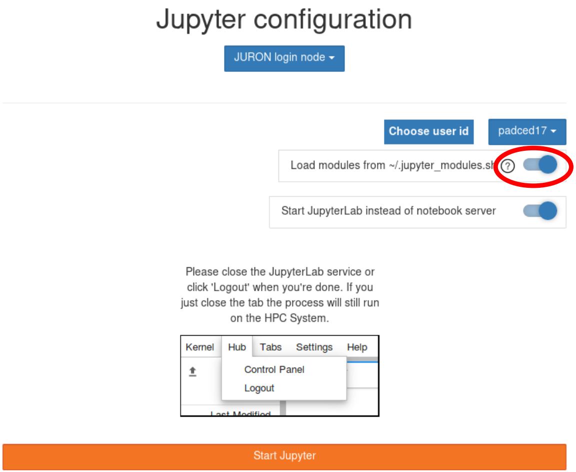
© Apache

© Keras

Keras an Example

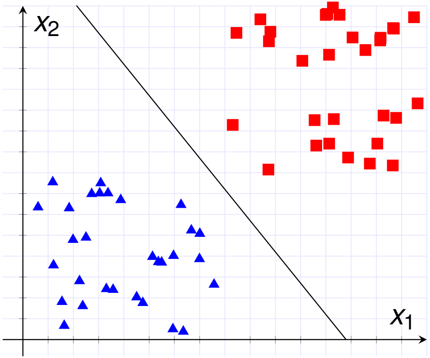
```python
from keras.models import Sequential
from keras.layers import Dense

# 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 1: Warm-up



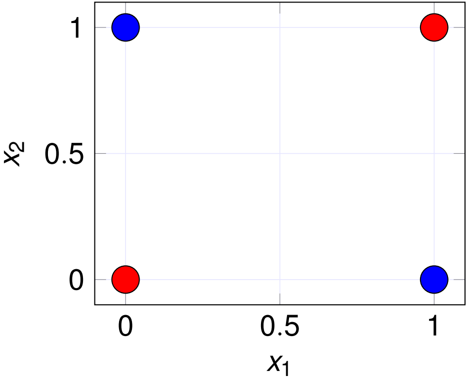
Exercise 2: Logistic Regression

- **Data set** must be mapped
  - $\blacksquare \rightarrow 0$
  - $\blacktriangle \rightarrow 1$
- **Model:** $h = sig(wx) = \frac{1}{1+e^{-wx}}$
- **Loss function:**
  $J(w) = MSE(w) = (y - \hat{y})^2$
  $\frac{dJ}{dw} = (\hat{y} - y) * (\hat{y} - \hat{y}^2) * x$
- **Train:** $w_{i+1} = w_i - lr \frac{dJ}{dw_i}$
- https://jupyter-jsc.fz-juelich.de/



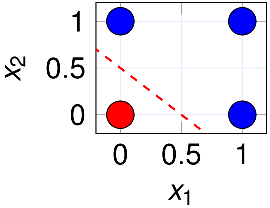
Neural Networks

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

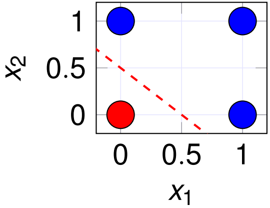
XOR-Problem

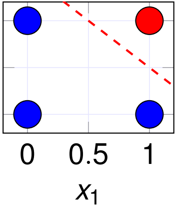


OR-gate

$$h_1 = sig(2x_1 + 2x_2 + 1)$$

XOR-Problem





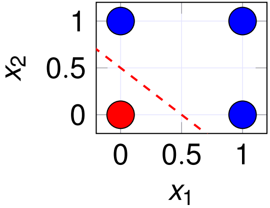
OR-gate

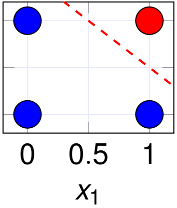
NAND-gate

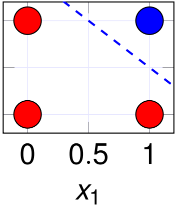
$$h_1 = sig(2x_1 + 2x_2 + 1)$$

$$h_2 = sig(-2x_1 - 2x_2 - 1.5)$$

XOR-Problem







OR-gate

$$h_1 = sig(2x_1 + 2x_2 + 1)$$

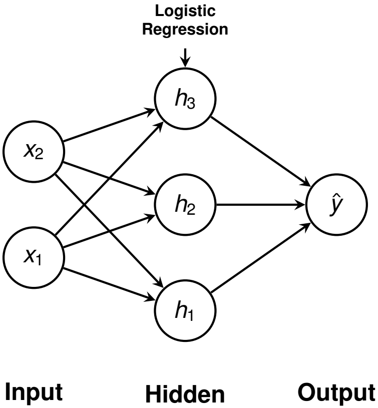
NAND-gate

$$h_2 = sig(-2x_1 - 2x_2 - 1.5)$$

AND-gate

$$\hat{y} = sig(2h_1 + 2h_2 + 3)$$

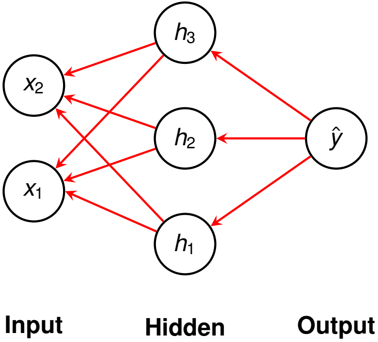
Fully-connected Neural Network



- 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

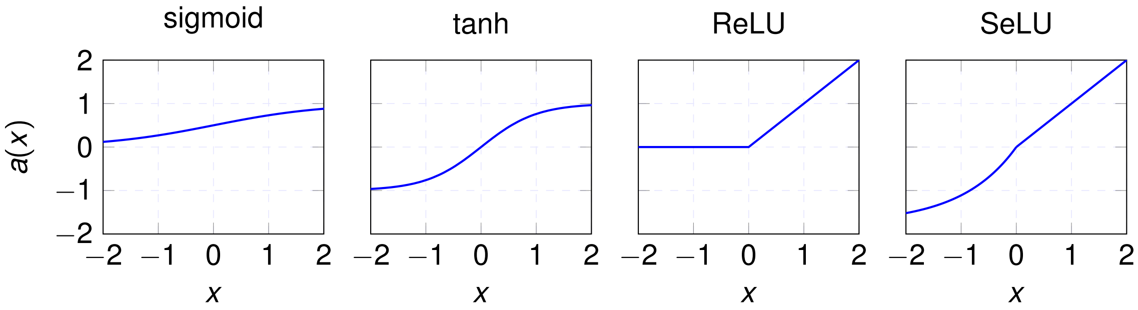
Backpropagation

- 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. $tanh(x)$, $relu(x) = max(0, x)$, 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

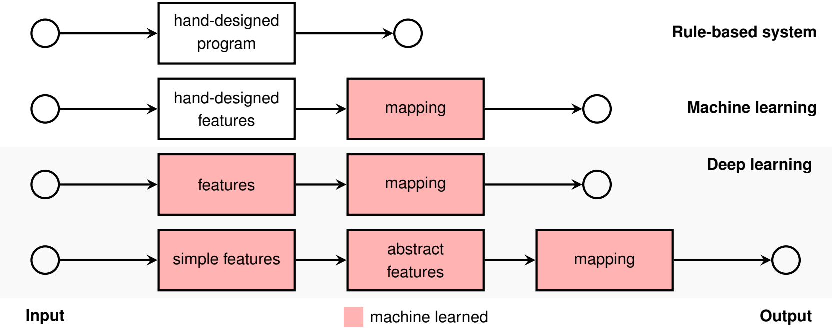
Universal Approximation Theorem

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.

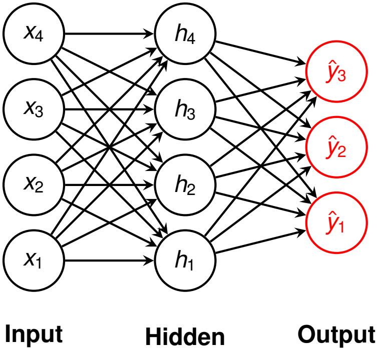
- **Good News**
  - Networks can perform highly complex tasks
  - All necessary ingredients available
- **Bad News**
  - Does not specify number of necessary nodes
  - No remarks on neuron connectivity

Deep Learning

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



Multi-class Classification



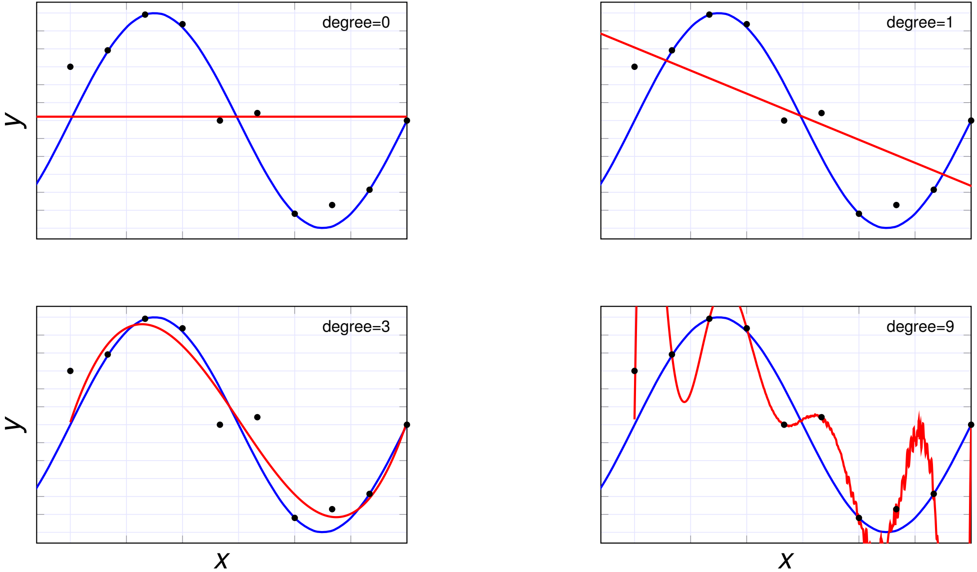
- Extension of binary classification concept
- **One-versus-all classification**
  - Build c binary classifiers
  - Pick class with highest confidence/probability
- In neural networks
  - Create multiple networks
  - **Add output neurons**

Multi-class Classification

Multi-class classification recipe:

- **One-hot class encoding:** encode classes as sparse vectors $y = (y_1, y_2, ..., y_c)$, only one is active, e.g. class 2 $\rightarrow (0, 1, ..., 0)$
- **Softmax output activation:** $\hat{y} = softmax(z) = \frac{e^{z_j}}{\sum_j e^{z_j}}$ for $j = 1...c$
  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

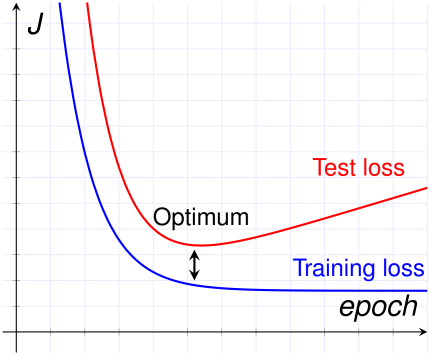
Over- and Underfitting



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 tunine 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 $J \to 0$, **memorization effect**
- **Test loss minimum** is optimal

Regularization

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

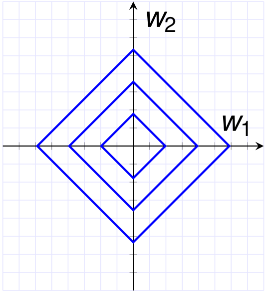
- More data
- Augment: generate artificially new samples (add noise, translate, ...)
- Early stopping: monitoring of train and test loss, stop at optimum
- Penalize large weights: add a penalty term
- Dropout neurons
- Batch Normalization neurons

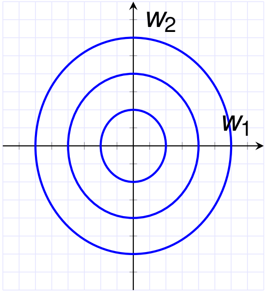
Penalty Terms

- 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

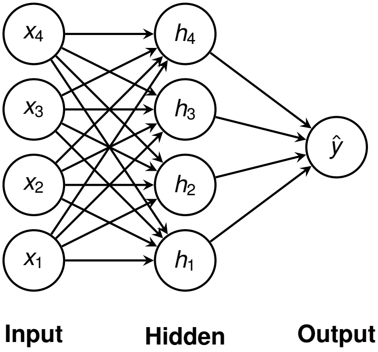
- **L1-Norm**, also **LASSO**
  - $\|w\|_1 = (w_1 + w_2 + \cdots + w_f)$
  - All weights contribute to loss
  - Encourages sparsity
- **L2-Norm**, also **weight decay**
  - $\|w\|_2^2 = (w_1^2 + w_2^2 + \cdots + w_f^2)$
  - Penalizes large weights
  - Discourages sparsity

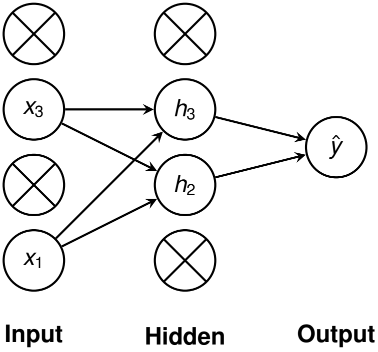




Dropout

- Randomly turn of neurons and connection, e.g. $p(drop) = 0.5$
- Equivalent to network regularization (proof omitted)





Neural Networks in Keras

```python
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")
```

Exercise 3: 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
- $(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m]$
- **Stride:** pixel distance for slide
- **Filter size:** window size of convolution kernel
- 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

Image Matrix

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

Kernel Matrix

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

Output Matrix

320					

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

Convolution with horizontal and vertical strides = 1

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

Kernel Matrix

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

320	206			

Output Matrix

$$0 * 0 + 0 * -1 + 0 * 0 \\ +105 * -1 + 102 * 5 + 100 * -1 \\ +103 * 0 + 99 * -1 + 103 * 0 = 206$$

Convolution with horizontal and vertical strides = 1

Discrete Convolution

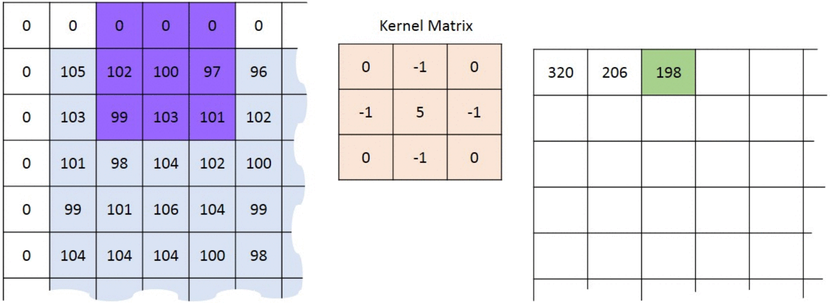


Image Matrix

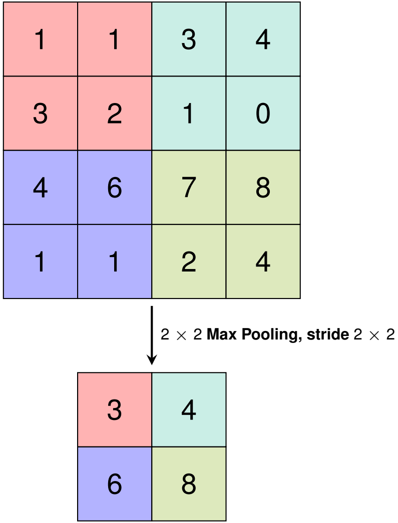
Output Matrix

$$0*0+0*-1+0*0$$
$$+102*-1+100*5+97*-1$$
$$+99*0+103*-1+101*0=198$$

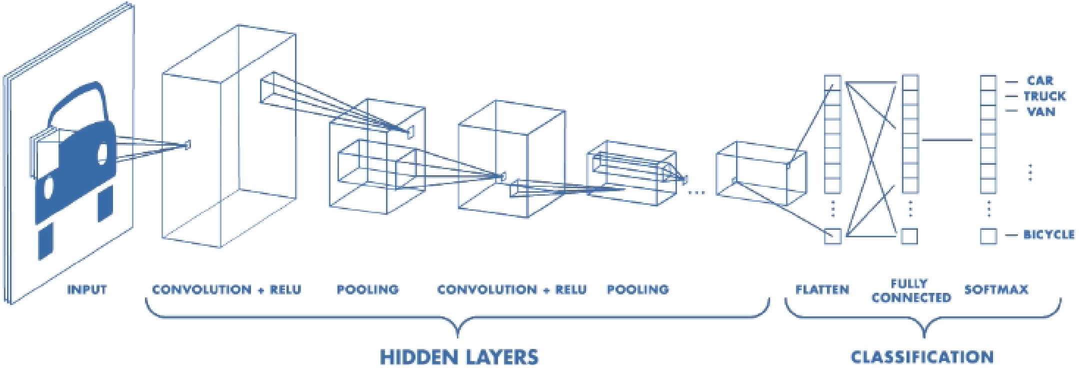
Convolution with horizontal and vertical strides = 1

Pooling

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

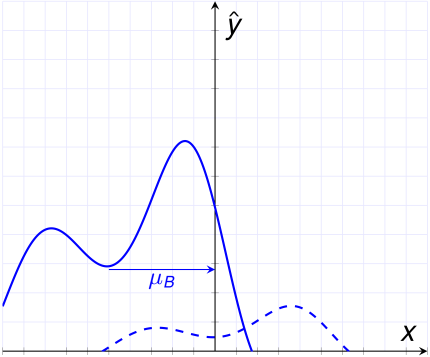


Convolutional Neural Network Pyramid



Batch Normalization

- **Idea:** standardize layer outputs
- Training time
  - Calculate batch mean μ_B and standard deviation σ_B
  - Track rolling values μ and σ
  - Normalize by $\frac{x-\mu_B}{\sigma_B}$
- Prediction time
  - Track μ_B and σ_B for covariate shift
  - Add one degree of freedom
- Regularizes network

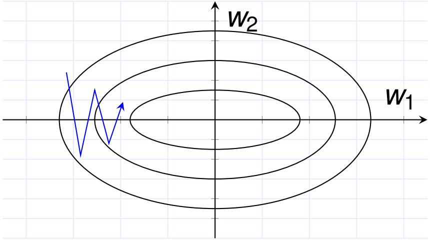


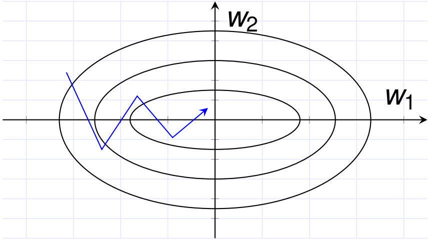
Optimizers

- 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

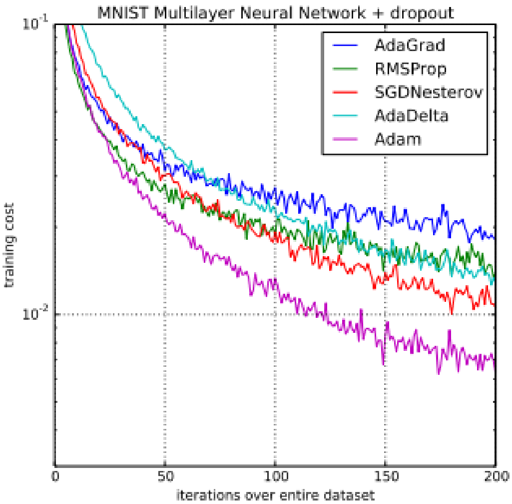
Optimizers

- 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



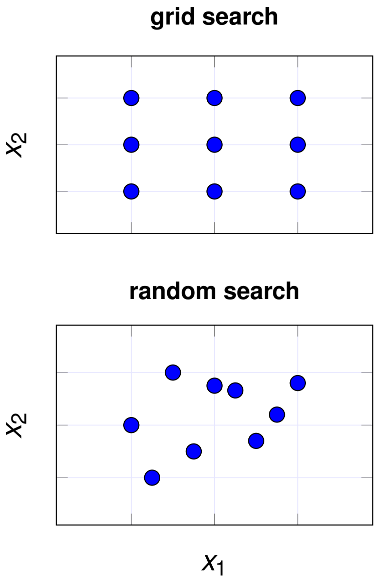
- **RMSprop:** enhanced Adagrad
  - Learning rate got infinitely small
  - Forgetting factor β for past gradients
- More intricate variants: Nesterov momentum, Adadelta, Nadam, . . .
- Some **non-SGD** alternatives
  - Particle-swarm optimization (PSO)
  - BFGS

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

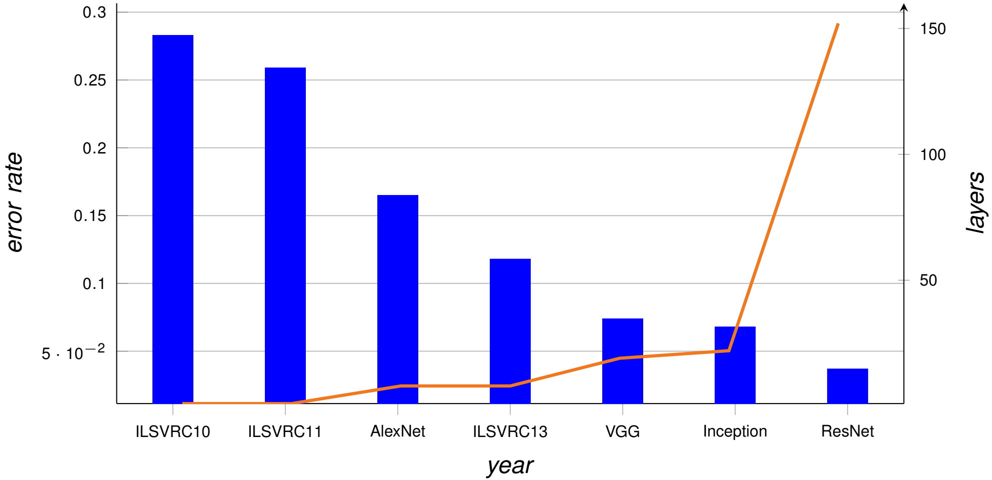


ImageNet Large Scale Visual Recognition Challenge (ILSVRC)

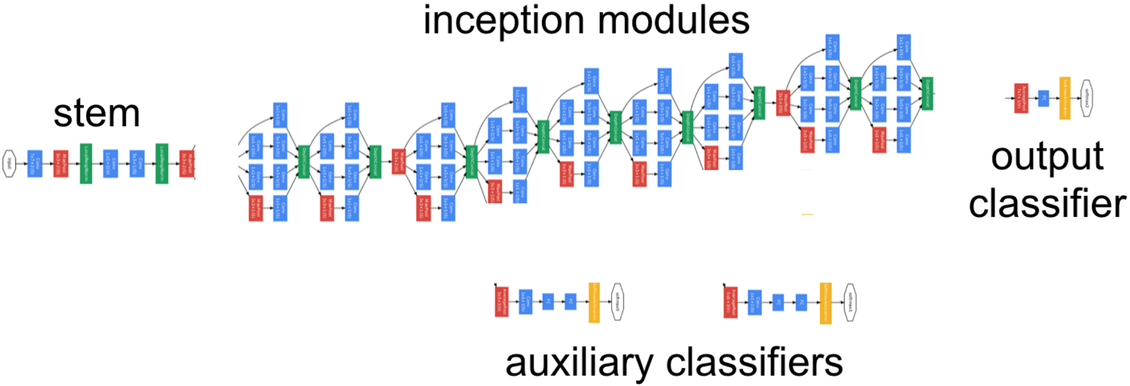
© Oren Kraus

- **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

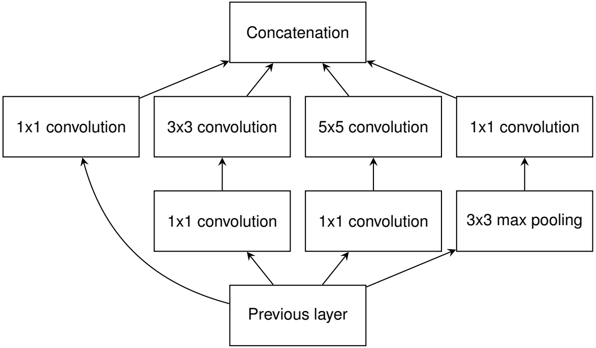


Inception


inception modules
stem
output
classifier
auxiliary classifiers

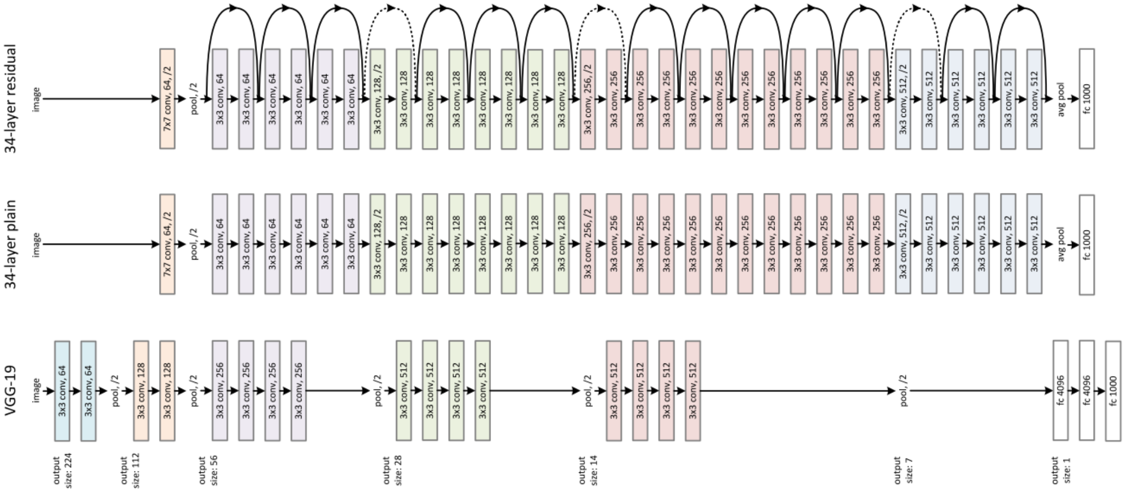
Inception

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



Inception Module

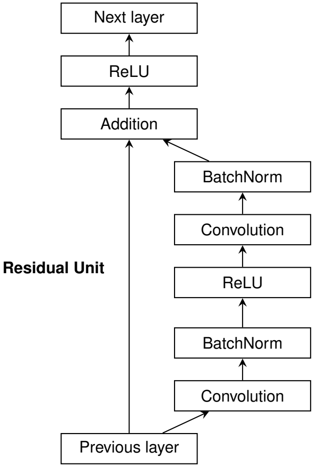
ResNet



© 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 multipe 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")
```

Keras Functional API

```python
from keras.models import Model
from keras.layers import Dense, Input

features = 20
data = Input(shape=(features,))

# layers can be connected by calling the previous layer
layer_1 = Dense(64)(data)
layer_2 = Dense(64)(layer_1)
predict = Dense(10, activation="softmax")(layer_2)

# the model must be created manually
model = Model(inputs=[data], outputs=[predict])
```

Exercise 4: CNN MNIST Image Classification

Regression

Abalone Dataset

- Collected by the University of Tasmania, Australia
- **Abalones are “sea snails”**
- 4177 instances, each 9 features
- Prediction task
  - Regress age ($rings + 1.5$) of abalone
  - Requires manual preprocessing
  - Reduce microscopy cost and labor

Exercise 5: Abalone Age Regression Analysis

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

- **Eileen Kühn**
  - GridKa School organization
  - Paperwork
- **Oskar Taubert**
  - Assignment preparation
  - Exercise supervision
- **Andreas Herten**
  - Access to JURON
  - Technical support

Discussion