Neuronale Netze

Prof. Dr. Rudolf Kruse

Computational Intelligence
Institut für Intelligente Kooperierende Systeme
Fakultät für Informatik
rudolf.kruse@ovgu.de

Deep Learning - Mehrschichtige Neuronale Netze mit vielen Schichten

Deep Learning: Motivation

Zur Erinnerung: das **universelle Approximationstheorem**

- Jede kontinuierliche Funktion eines beliebigen kompakten Teilraums von $\mathbb{R}^n$ kann beliebig genau mit einem dreischichtigen Neuronalen Netz approximiert werden.

Dieses Theorem wird oft zitiert, um darauf hinzuweisen, dass:

- eine versteckte Schicht ist theoretisch ausreichend
- keine Notwendigkeit für tiefe Neuronale Netze

Allerdings macht das Theorem keine Aussage über die voraussichtliche Anzahl an Neuronen der verstecken Schicht

Abhängig von der zu approximierenden Funktion kann eine enorm große Anzahl an Neuronen notwendig sein.

Hinzufügen weiterer versteckter Schichten erlaubt vielleicht die gleiche Approximationsgüte mit weniger Neuronen zu erzielen.

Deep Learning: Motivation

Ein sehr simples und typischerweise verwendetes Beispiel ist die n-Bit Paritäts-Funktion. Ihre Ausgabe ist 1, wenn eine gerade Anzahl an Eingaben ebenfalls 1 ist, ansonsten ist die Ausgabe 0.

- Diese Funktion kann sehr leicht mit einem dreischichtigen Neuronalen Netz repräsentiert werden.
- die disjunktive Normalform besteht aus 2^{n-1} Konjunktionen, welche jeweils durch ein Neuron in der versteckten Schicht dargestellt werden können

Die Zahl der notwendigen Neuronen steigt exponentiell mit der Anzahl der Eingabeneuronen.

Verwenden wir jedoch mehrere Schichten können wir lineares Wachstum ermöglichen.

- Starte mit einer Biimplikation zweier Eingaben.
- Nutze eine Kette von Xor-Neuronen, deren Ausgabe jeweils allen Eingaben eine neue Eingabe hinzufügen.
- Solch ein Netzwerk benötigt $n + 3(n-1) = 4n - 3$ Neuronen. (n Eingabeneuronen, $3(n-1) - 1$ versteckte Neuronen, 1 Ausgabeneuron.)

Deep Learning: Motivation

Praxisnahe-Probleme sind im Vergleich oftmals anders, da die Trainingsdaten in ihrer Größe limitiert sind.

- Vollständige Trainingsdaten für eine n-stelligen Boolesche Funktion umfassen 2^n Trainingsbeispiele.
- Praxisnahe Datensätze umfassen oft deutlich weniger Trainingsbeispiele. Dadurch sind viele Eingabekombinationen nicht definiert und können frei belegt werden.

Nichtsdestotrotz kann das Verwenden weiterer versteckter Schichten oft zu einer Reduktion der notwendigen Neuronen führen.

Dies ist der Fokus von Deep Learning.

Für mehrschichtige Neuronen ist die Tiefe die Anzahl versteckter Schichten plus die Ausgabeschicht.

Deep Learning: Hauptprobleme

Tiefe Neuronale Netze leiden meisten unter zwei Hauptproblemen

overfitting und **vanishing gradient**

- Overfitting hängt zumeist mit der Vergrößerung der anpassbaren Parameter zusammen.
- Praxisnahe Datensätze umfassen oft deutlich weniger Trainingsbeispiele.

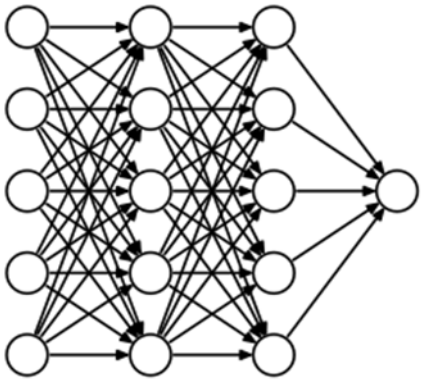
Regularisierung verhindert große Gewichte und damit überangepasste Anpassungen. Weiterhin können die Netze wie folgt eingeschränkt werden:

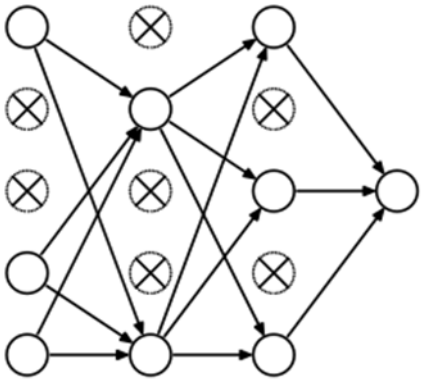
- jede Schicht verfügt nur über eine begrenzte Zahl an versteckten Neuronen
- nur wenige versteckte Neuronen pro Schicht sollten aktiv sein. (Dieser Effekt kann durch eine Regularisierung der Fehlerfunktion erzielt werden.)

Weiterhin kann die Dropout-Methode während des Lernprozesses angewandt werden, um Überanpassung vorzubeugen.

Dropout

ohne Dropout





mit Dropout

Gewünschte Eigenschaft:
Robustheit bei Ausfall von Neuronen

Ansatz beim Lernen:
- Nutze nur $p\%$ der Neuronen ($p < 50$)
- Wähle diese zufällig

Ansatz beim Anwenden:
- Nutze 100% der Neuronen
- Multipliziere alle Gewichte mit p

Ergebnis:
- Robustere Repräsentation
- Verbesserte Generalisierung
- Verringerte Überanpassung

Erinnerung: Fehlerrückpropagation - Vorgehensweise

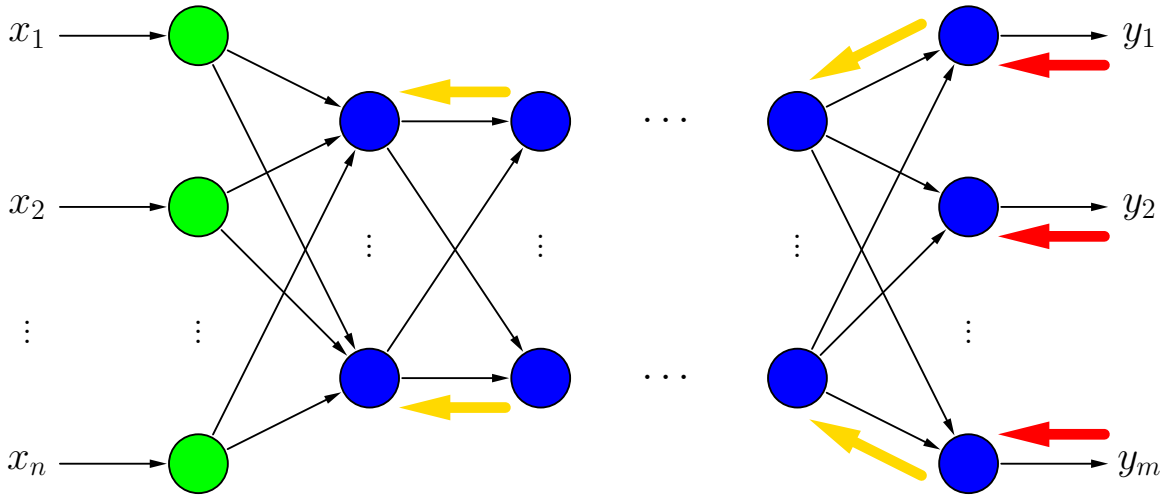
$\forall u \in U_{\text{in}}$:

$\text{out}_u^{(l)} = \text{ex}_u^{(l)}$

Vorwärts-propagation:

$\forall u \in U_{\text{hidden}} \cup U_{\text{out}}$:

$$\text{out}_u^{(l)} = \left(1 + \exp\left(-\sum_{p \in \text{pred}(u)} w_{up}\, \text{out}_p^{(l)}\right)\right)^{-1}$$



logistische Aktivierungs-funktion

impliziter Biaswert

Fehlerfaktor:

Rückwärts-propagation:

$\forall u \in U_{\text{hidden}}$:

$$\delta_u^{(l)} = \left(\sum_{s \in \text{succ}(u)} \delta_s^{(l)} w_{su}\right) \lambda_u^{(l)}$$

$\forall u \in U_{\text{out}}$:

$$\delta_u^{(l)} = \left(o_u^{(l)} - \text{out}_u^{(l)}\right) \lambda_u^{(l)}$$

Aktivierungs-ableitung:

$$\lambda_u^{(l)} = \text{out}_u^{(l)} \left(1 - \text{out}_u^{(l)}\right)$$

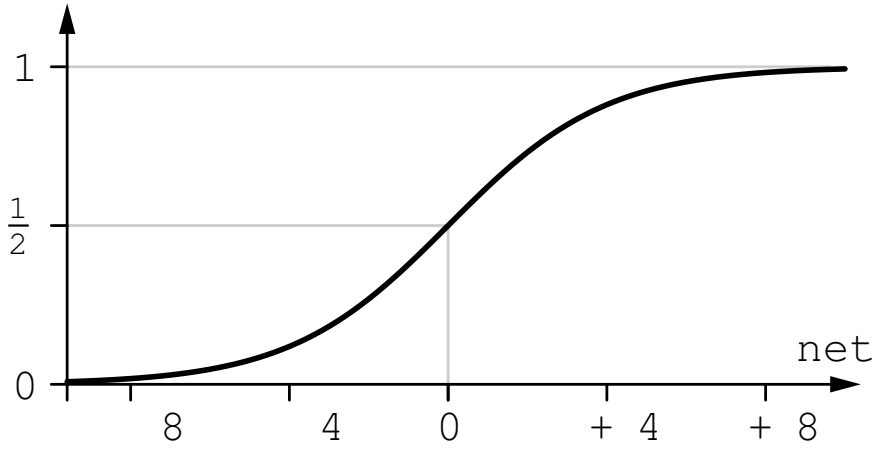
Gewichts-änderung:

$$\Delta w_{up}^{(l)} = \eta\, \delta_u^{(l)}\, \text{out}_p^{(l)}$$

Deep Learning: Vanishing Gradient Problem

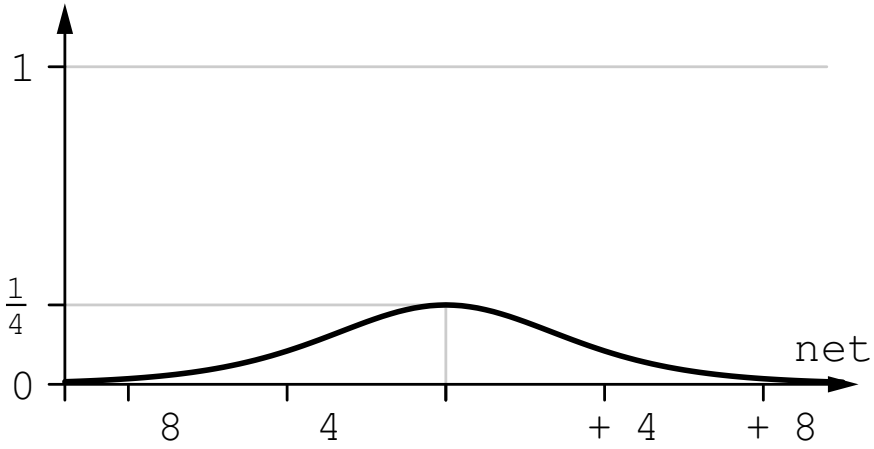
logistische Aktivierungsfunktion:

$$f_{\text{act}}(\text{net}, \theta) = \frac{1}{1 + e^{-(\text{net} - \theta)}}$$



Ableitung der logistischen Funktion:

$$f'_{\text{act}}(\text{net}, \theta) = f_{\text{act}}(\text{net}, \theta) \cdot (1 - f_{\text{act}}(\text{net}, \theta))$$



Wenn die logistische Aktivierungsfunktion verwendet wird, sind die Gewichtsänderungen proportional zu $\lambda_u^{(l)} = \text{out}_u^{(l)} \left(1 - \text{out}_u^{(l)}\right)$.

Dieser Faktor wird mit zurück propagiert ist aber nie größer als $\frac{1}{4}$ (siehe rechts).

Der Gradient tendiert dazu verschwindend klein zu werden wenn durch viele Schichten zurück propagiert wird.

Lernen in den vorderen versteckten Schichten kann sehr langsam werden [Hochreiter 1991].

Deep Learning: Vanishing Gradient Problem

Im Prinzip kann einem kleinen Gradienten mit einem großen Gewicht entgegengewirkt werden.

$$\delta_u^l = (\sum_{s \in succ(u)} \delta_s^{(l)} w_{su}) \lambda_u^{(l)}$$

Leider führt ein großes Gewicht innerhalb der Aktivierungsfunktion zu einer Verschiebung in Richtung der Grenzen.

Dadurch wird der absolute Wert der abgeleiteten Aktivierungsfunktion durch die Wahl eines großen Gewichts oft kleiner.

Außerdem werden in der Praxis die Gewichte oft mit Werten zwischen -1 und +1 initialisiert. Dies beeinflusst insbesondere die ersten Trainingsschritte in welchen die Gewichte und der Gradient jeweils meistens einen Wert kleiner als 1 annehmen. ist aber nie größer als $\frac{1}{4}$ (siehe rechts).

Rectified Linear Unit (ReLU)

Wähle statt klassisch definierter Neuronen Rectified Linear Unit

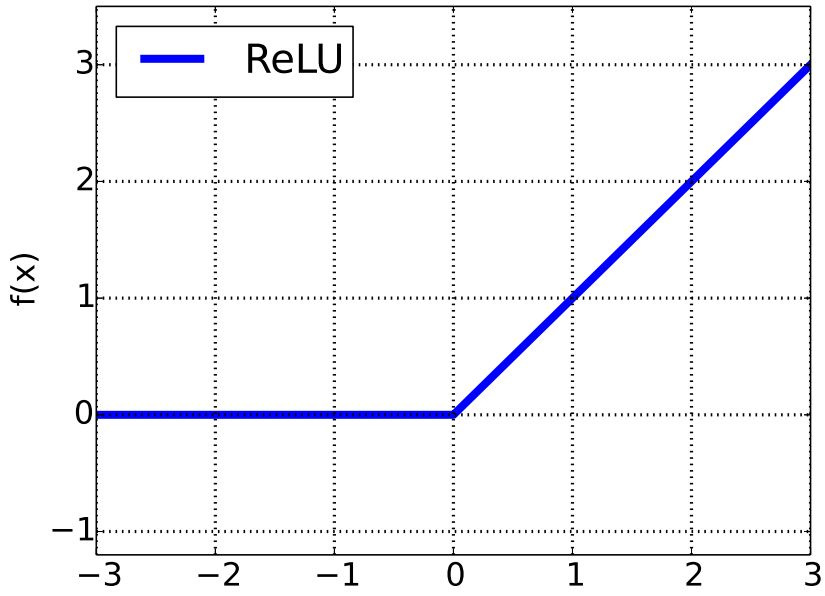
ReLU: $f(x) = max(0, x)$

Vorteile:

- sehr einfache Berechnung
- Ableitung ist leicht zu bilden
- 0-Werte vereinfachen Lernen

Nachteile:

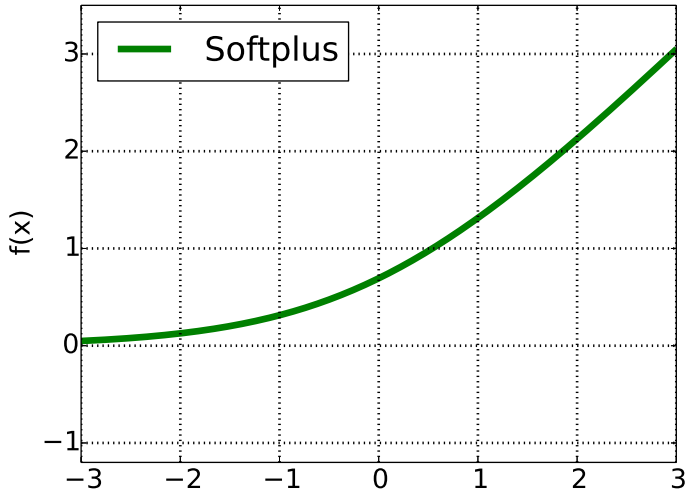
- kein Lernen links der 0
- mathematisch eher unschön
- Nicht-differenzierbarer „Knick“ bei 0



[ReLu nach Glorot et. al 2011]

ReLU: Berechnungsvarianten

Softplus:
$f(x) = ln(1 + e^x)$
- „Knick“ wurde beseitigt
- Einige Vorteile auch



[Softplus nach Glorot et. al 2011]

Noisy ReLU:
$f(x) = max(0, x + \mathcal{N}(0, \sigma(x)))$
- Addiert Gaussches Rauschen

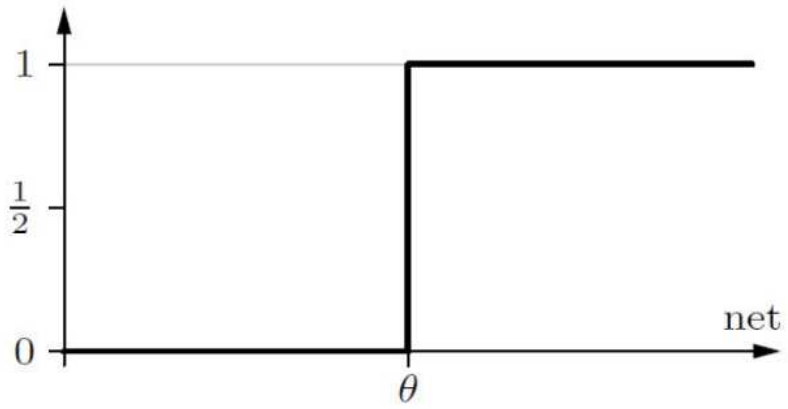
Leaky ReLU

$$f(x) = \begin{cases} x, & \text{falls } x > 0, \\ 0.01x, & \text{sonst.} \end{cases}$$

Deep Learning: unterschiedliche Aktivierungsfunktionen

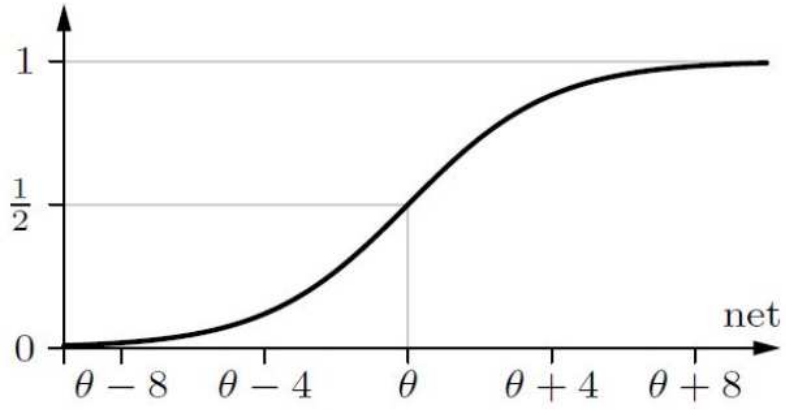
Ableitung der ReLu Funktion:

$$f'_{\text{act}}(\text{net}, \theta) = \begin{cases} 1, & \text{if } net \geq \theta \\ 0, & \text{otherwise} \end{cases}$$



Ableitung der Softplus Funktion:

$$f'_{\text{act}}(\text{net}, \theta) = \frac{1}{1 + e^{-(\text{net} - \theta)}}$$



Die Ableitung der Rampenfunktion ist die Stufen-Funktion

Die Ableitung der Softplus Funktion ist die Logistische-Funktion

Faktoren dieser abgeleiteten Funktionen sind zumeist größer. Darauf folgt:

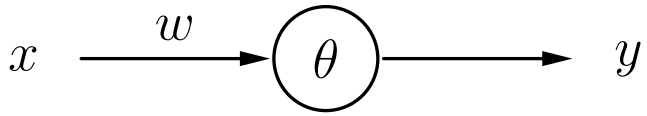
- größere Gradienten in den vorderen Schichten
- schnelleres Training

Weitere Beschleunigungen des Trainingsprozesses können erzielt werden durch:

- optimierte Hardware
- GPU bezogene Implementierungen

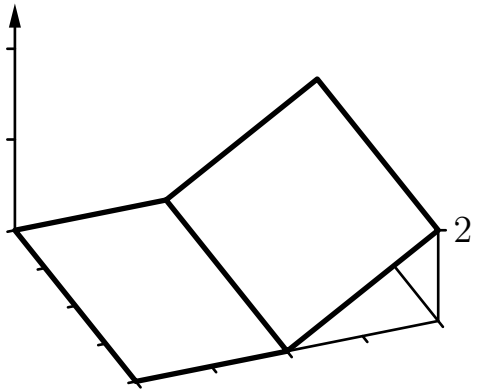
Erinnerung: Trainieren von Schwellenwertelementen

Schwellenwertelement mit einer Eingabe für die Negation $\neg x$.

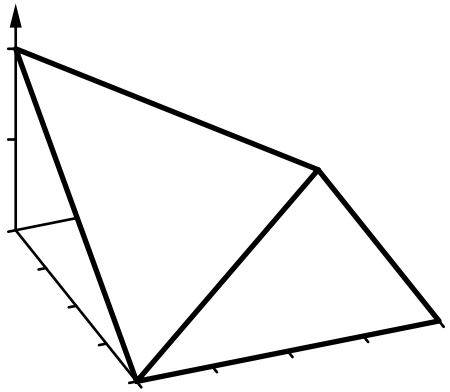


x	y
0	1
1	0

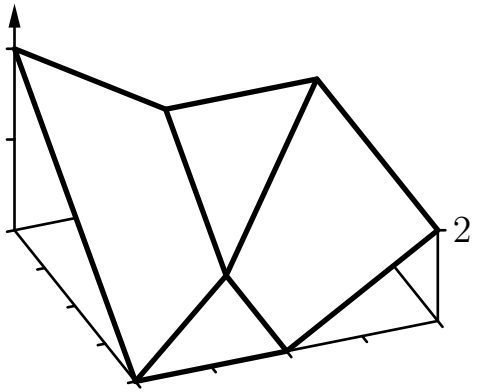
Modifizierter Ausgabefehler als Funktion von $\vec{w}$ und θ.



Fehler für $x = 0$



Fehler für $x = 1$



Summe der Fehler

Die Rectified Linear Unit ergibt nutzt diese Fehlerwerte direkt.

Deep Learning: Auto-Encoder

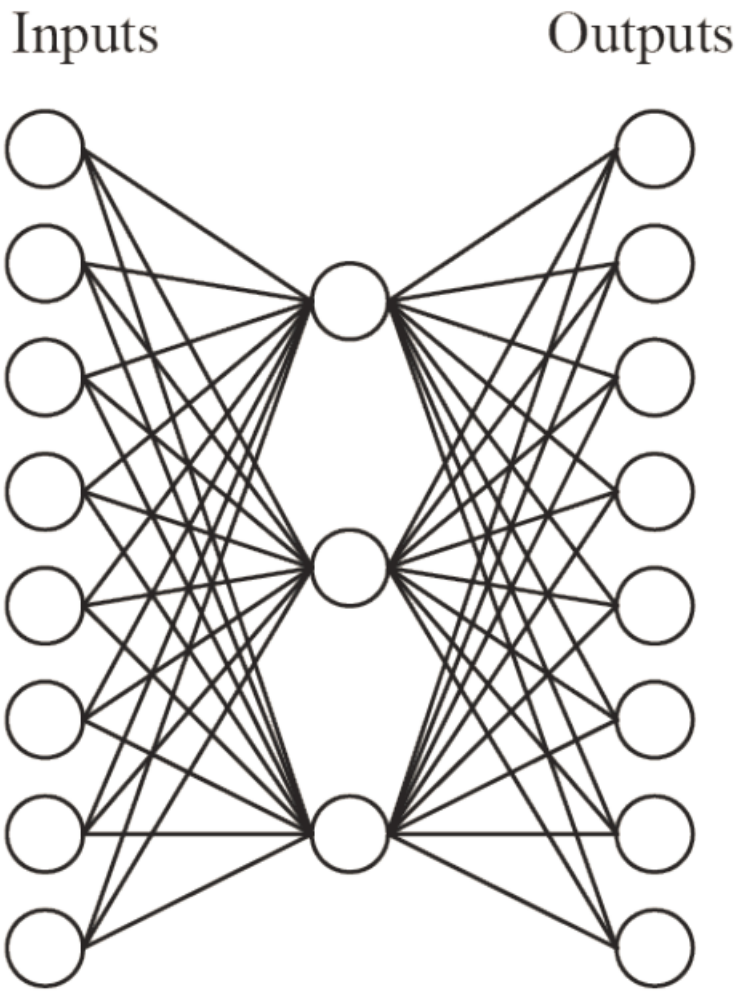
MLPs mit vielen Schichten können trainiert werden, jedoch ist Effektivität und Effizienz oft eingeschränkt
Ausweg: **Konstruiere das Netz Schicht für Schicht, Trainiere jeweils nur die neu hinzugefügte Schicht in jedem Schritt**

Populäre Lösung: **Konstruiere das Netz als gestapelter Auto-Encoder**

Ein Auto-Encoder ist ein 3-schichtiges Neuronales Netz, welches seine Eingaben auf Annäherungen desselben Inputs abbildet.

- Die versteckte Schicht bildet einen Kodierer in eine interne Repräsentation.
- Die Ausgabeschicht bildet einen Dekodierer in die ursprüngliche Repräsentation.

Auto-Encoder



Erstellt eine Kodierung der Daten

Lernt Gewichte mit Rückpropagation

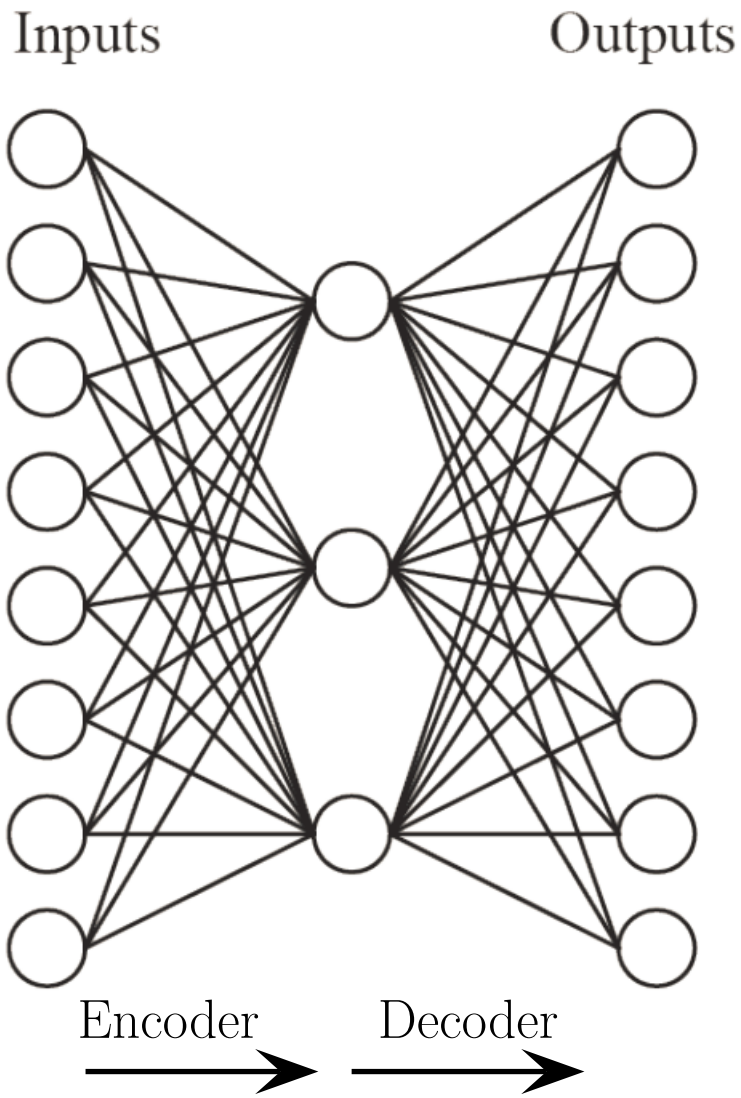
überwachtes Lernen mit Fehler

$$|out - in|^2$$

Inputs		Versteckte Gewichte		Outputs
10000000	→	?	→	10000000
01000000	→		→	01000000
00100000	→		→	00100000
00010000	→		→	00010000
00001000	→		→	00001000
00000100	→		→	00000100
00000010	→		→	00000010
00000001	→		→	00000001

[Grafiken nach T. Mitchell, Machine Learning, McGraw Hill, 1997]

Autoencoder



Nutze für Dekodierung die transponierte Gewichtsmatrix der Encodierung

Ergebnis nach 5000 Lerniterationen:

Binäre Kodierung annähernd erreicht

Inputs		Versteckte Gewichte				Outputs
10000000	→	.89	.04	.08	→	10000000
01000000	→	.15	.99	.99	→	01000000
00100000	→	.01	.97	.27.	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.01	.11	.88	→	00000100
00000010	→	.80	.01.	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

[Grafiken nach T. Mitchell, Machine Learning, McGraw Hill, 1997]

Deep Learning: Auto-Encoder

Durch die Kodierung und Dekodierung konstruiert ein Autoencoder im Laufe des Trainings neue Attribute.

Dabei wird sich erhofft, dass die gelernten Attribute die Eingabedaten auf sinnvolle Weise komprimiert, so dass die ursprüngliche Eingabe aus den reduzierten Attributen wieder rekonstruiert werden kann.

Hauptproblem:

- wieviele Neuronen werden für die versteckte Schicht benötigt?
- wie wird diese Schicht während des Trainings behandelt?

Falls die Zahl versteckter Neuronen gleich der Zahl der Eingabeneuronen ist, ist es wahrscheinlich, dass die Information unverändert durch das Netz gegeben wird.

Die offensichtliche Lösung sind sparse Autoencoder, welche aus deutlich weniger versteckten Neuronen als Eingabeneuronen bestehen.

Deep Learning: Sparse und rauschreduzierte Auto-Encoder

Wenige versteckte Neuronen zwingen den Auto-Encoder zum Lernen von relevanten Attributen.

Wieviele Neuronen sind eine gute Wahl? Kreuzvalidierung ist hier nicht unbedingt ein guter Lösungsansatz.

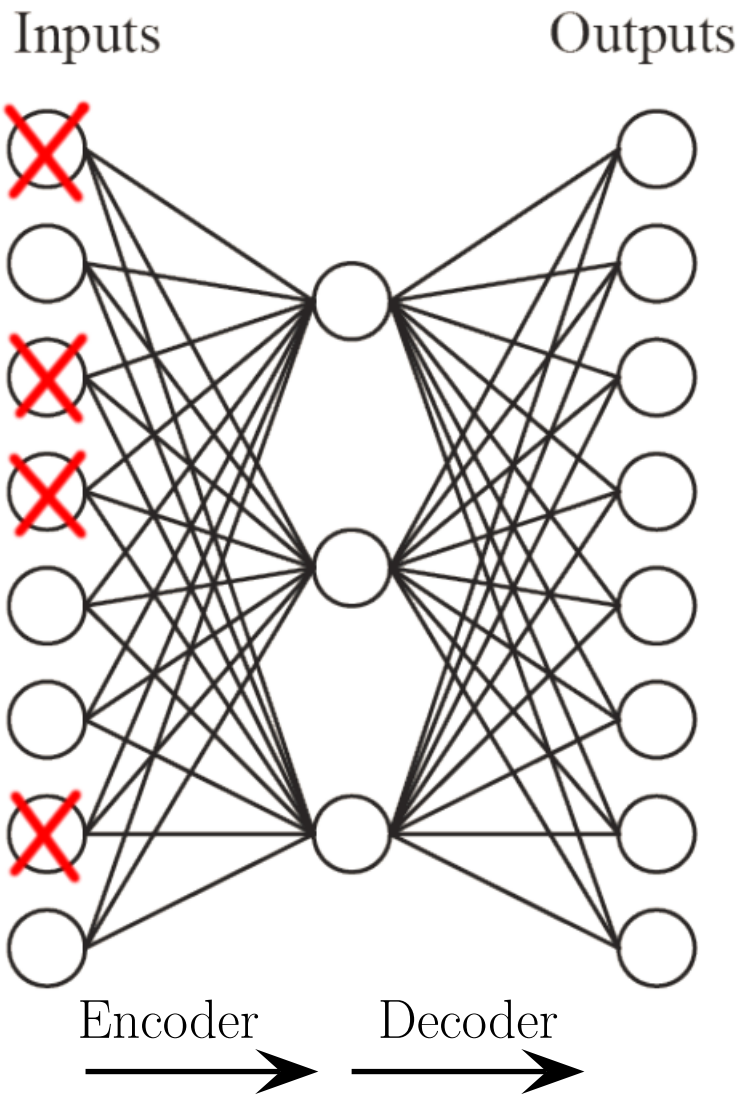
Alternativen zu wenigen versteckten Neuronen:

- Einschränken der aktivierten Neuronen in der versteckten Schicht
- Regularisierung der Fehlerfunktion, welche eine größere Anzahl an aktiven versteckten Neuronen bestraft

Ein weiterer Ansatz wäre das Hinzufügen von Rauschen zu den Eingabedaten, aber nicht bezüglich der Auswertung in den Ausgabeneuronen): denoising Auto-Encoder

Der auf diese Weise trainierte Auto-Encoder soll die rauschveränderten Daten auf ihre ursprünglichen unverrauschten Werte abbilden.

Rauschreduzierender (Denoising) Auto-Encoder



Gegeben:
eine dünne (sparse) Repräsentation

Gewünscht:
eine volle Repräsentation

Ansatz:
Kombiniere Autoencoder mit Dropout

Ergebnis:
komprimierte Darstellung
dynamisch auf Lernbeispiele zugeschnitten
Features für andere Algorithmen

Deep Learning: Stapeln von Auto-Encodern

Typischerweise werden gestapelte Auto-Encoder schichtweise konstruiert.

In der ersten Schicht wird ein Autoencoder auf den Eingabedaten trainiert. Die versteckte Schicht konstruiert die primären Attribute.

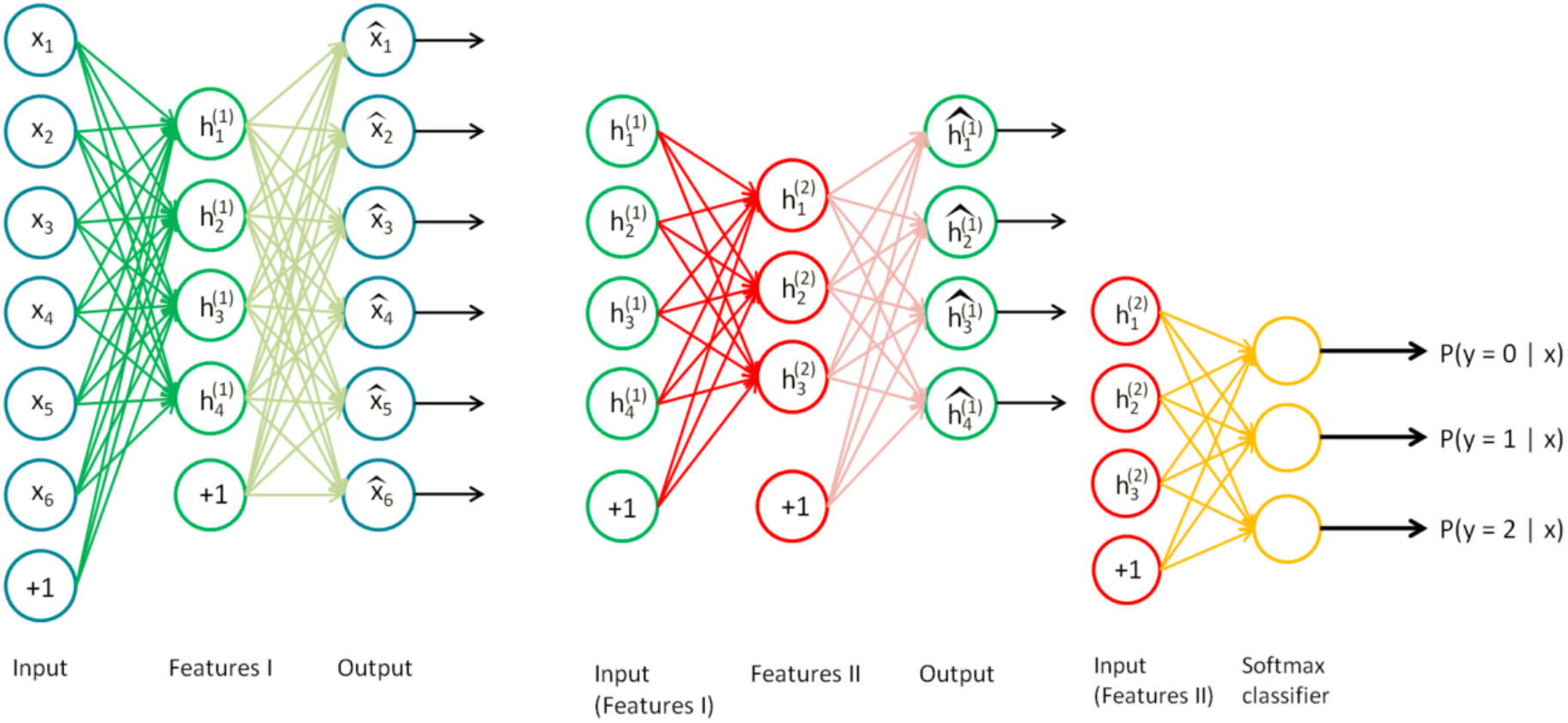
In einem zweiten Schritt werden die primären Attribute durch einen weiteren Auto-Encoder auf sekundäre Attribute reduziert.

Dieser Ablauf kann beliebig oft wiederholt werden.

Anschließend wird Backpropagation verwendet, um die gelernte Netzstruktur feinabzustimmen.

Stapeln von Autoencodern

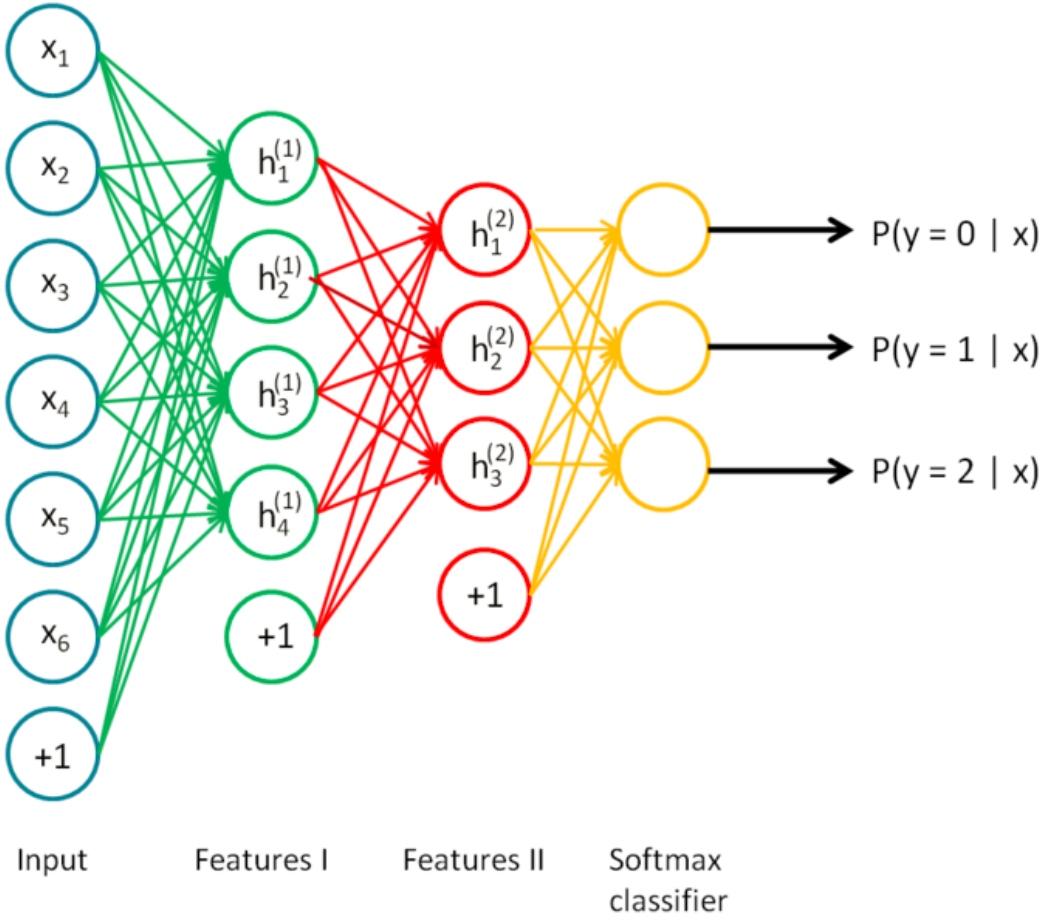
Staple Autoencoder, um die besten Features zu erhalten



[http://ufldl.stanford.edu/wiki/index.php/Stacked_Autoencoders]

Stapeln von Autoencodern

Nutze die (vor)gelernten Features zur Klassifikation



[http://ufldl.stanford.edu/wiki/index.php/Stacked_Autoencoders]

Hybrider Deep Learning Algorithmus

1. Definiere für die Lernaufgabe geeignete Netzstruktur
2. Erstelle entsprechend der Struktur Autoencoder und lasse sie mit Rückpropagation einzeln lernen
3. Verwende nur die Encoder, ihre Gewichte und eine weitere vollständig vernetzte, zufällig initialisierte Schicht zur Klassifikation
4. Lasse das so vortrainierte Netz mit Rückpropagation lernen

Deep Learning: Convoluational Neural Networks (CNNs)

MLPs mit mehreren versteckten Schichten wurden bereits erfolgreich zur Erkennung von handschriftlichen Zahlen verwendet. Die Daten wurden jedoch bereits vorher stark vorverarbeitet.

Es wurde daher angestrebt ähnliche Netze auch für allgemeinere Bilderkennungsaufgaben zu konstruieren.

Für solche Anwendungen ist es vorteilhaft, wenn die konstruierten Attribute der versteckten Schichten unabhängig von der Position des Bildausschnitts sind.

Spezielle Form des Deep Learning: Convolutional Neural Network

Inspiriert von der menschlichen Retina. Neuronen haben ein rezeptives Feld mit eingeschränkter Region.

Deep Learning: Convoluational Neural Networks (CNNs)

Convolution (Faltung) wird durch den Einsatz eines Kernel umgesetzt. Hierbei wird eine Matrix stückweise gefiltert.

Dieses Vorgehen ist bereits aus der Bildverarbeitung bekannt, in welcher beispielsweise Kanten Ausschnittweise in einem Bild erkannt werden.

Ein Bild wird hierbei als Matrix von Pixelinformationen dargestellt. Die Faltung wird mithilfe einer Kernelmatrix umgesetzt, welche Ausschnittweise mit den Pixelinformationen multipliziert wird.

Convolutional Neuronale Netze bilden eine Klasse vorwärtsgerichteter Neuronaler Netze, welche erfolgreich in zahlreichen Bildanalyse Aufgaben eingesetzt wurden.

Früher wurden Filter händisch konstruiert und in das Netz eingebunden. Mittlerweile werden Lernverfahren eingesetzt um automatisch passende Filter zu lernen.

- `https://docs.gimp.org/en/plug-in-convmatrix.html`
- `http://cs231n.github.io/convolutional-networks/`

Problem: Objekterkennung in Bildern

Imagenet Large Scale Visual Recognition Challenge (LSVRC) seit 2010

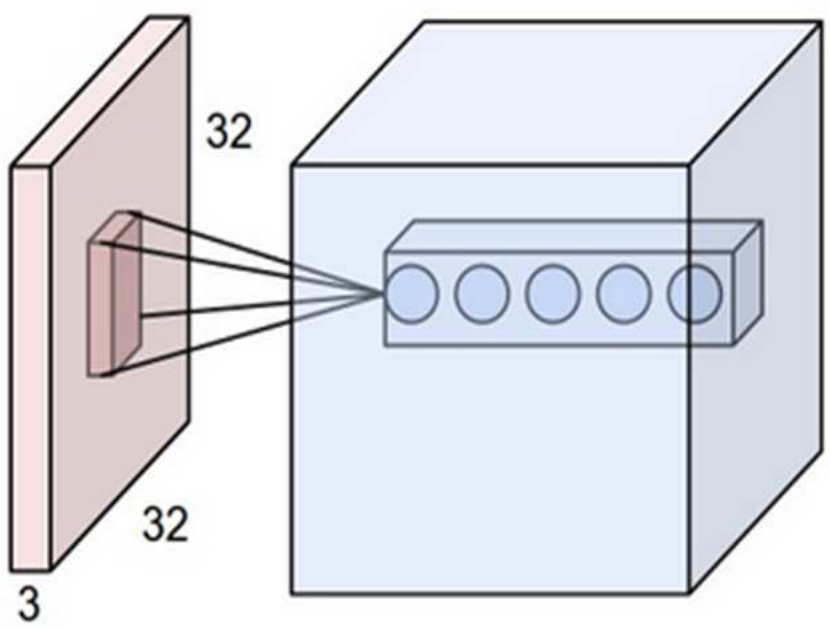
Finde 200 Objektklassen (Stuhl, Tisch, Person, Fahrrad,...)

in Bildern mit ca. 500 x 400 Pixeln, 3 Farbkanälen

Neuronales Netz mit ca. 600.000 Neuronen in der ersten Schicht

200 Neuronen in der Ausgabeschicht

Faltung (Convolution)



Motivation: Egal wo auf dem Bild ein Objekt ist, soll es erkannt werden

Idee: Verwende die selben Features auf dem gesamten Bild

Umsetzung: Filter / Kernel werden auf jedem Teil des Bildes angewandt und teilen sich die Gewichte

Parameter:
Anzahl der Filter
Stärke der Überlappung

Faltung (Convolution)

Image

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Filter

1	0	1
0	1	0
1	0	1

Convolved Feature

4	3	4
2	4	3
2	3	4

Featuretransformation

Schiebe einen „Filter“ über die Features und betrachte die „gefilterten“ Features

Multipliziere Originalfeature mit Filter und Summiere

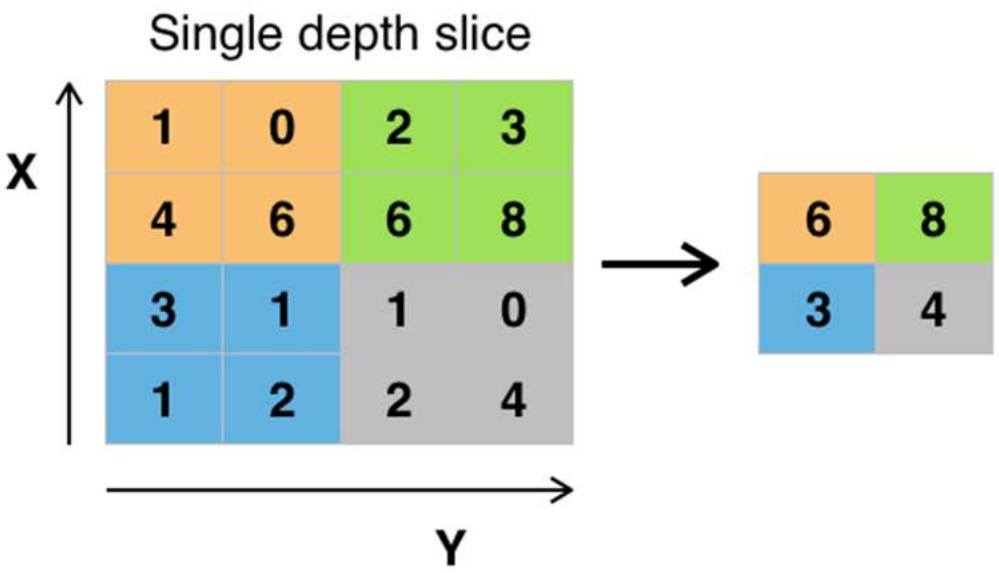
Originalraum: 5x5

Filtergröße: 3x3

Neue Featuregröße: 3x3

Featureraum wird kleiner

Pooling



Featuretransformation

Schiebe einen „Filter“ über die Features und betrachte die „gefilterten“ Features

Betrachte den Bereich entsprechend der Filtergröße

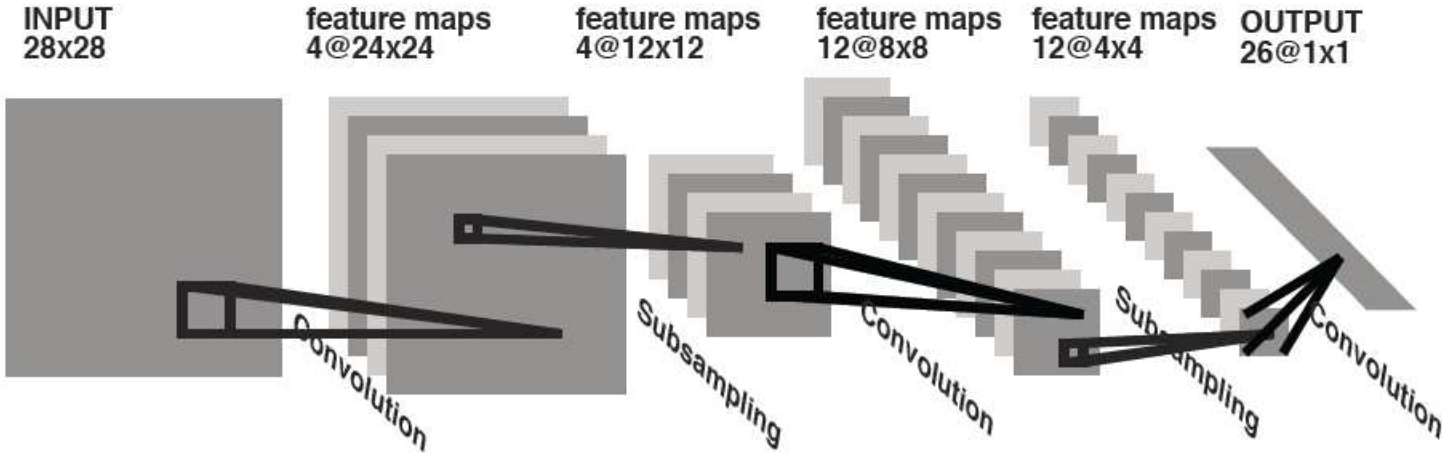
Max Pooling: Nimm maximalen Wert

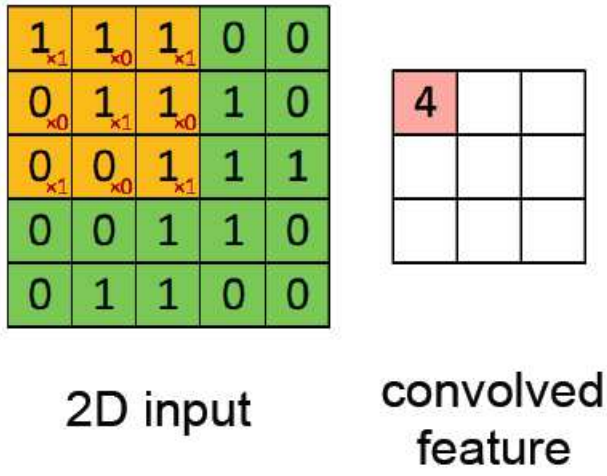
Mean Pooling: Nimm Mittelwert

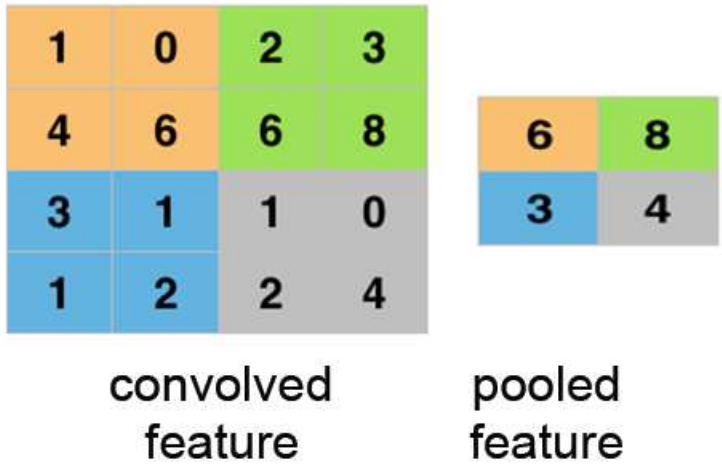
Featureraum wird kleiner

Faltendende (Convolutional) Neuronale Netze

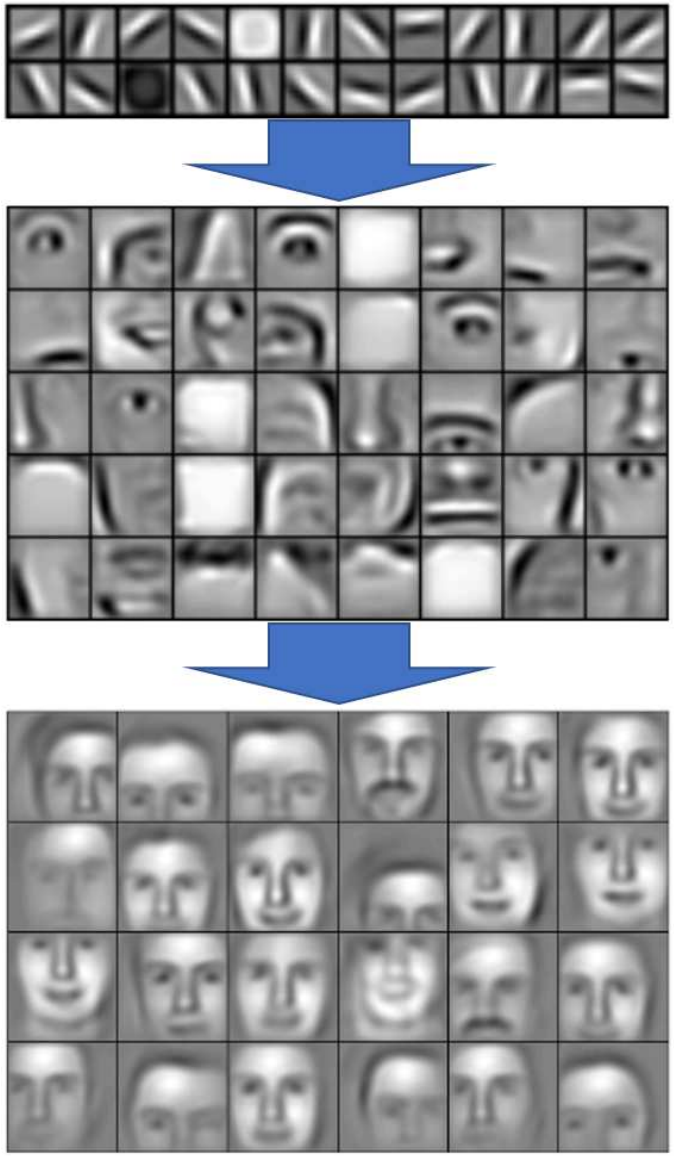
[Y. Bengio and Y. Lecun, 1995]







Features in Faltenden (Convolutional) Neuronalen Netzen



Gut trainierte Netze haben klar erkennbare Features

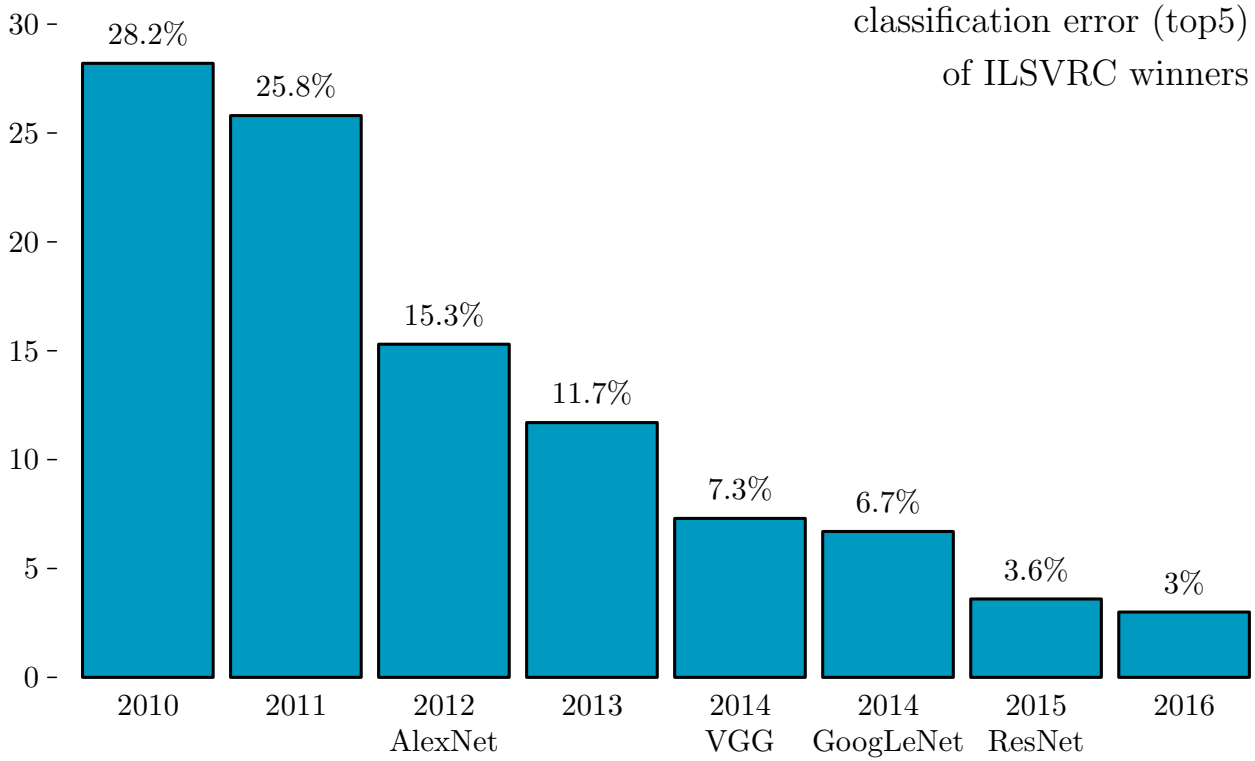
Features werden in tieferen Schichten komplexer

Layer 1:
Kantenzüge

Layer 2:
Augen, Nasen, Augenbrauen, Münder

Layer 3:
(abgeschnittene) ganze Gesichter

Resultate im Bereich Bildklassifizierung



Noch vor 10 Jahren: unmöglich

Rasante Entwicklung in den letzten Jahren

Oft verwendet: Ensembles von Netzen

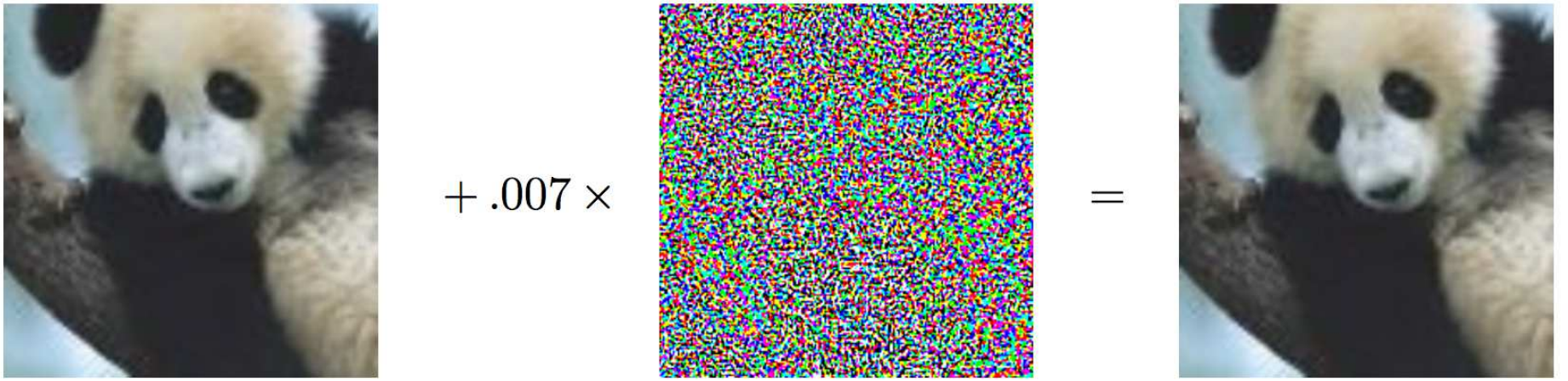
Netze werden tiefer: ResNet (2015) mehr als 150 Schichten

Adversarial Examples

kritische Anwendungen: Sicherheitskameras, selbstfahrende Autos

Wie sicher sind die Vorhersagen?

Neuronale Netze können ausgetrickst werden!



[Goodfellow et al. 2015]

German Traffic Sign Recognition Benchmark (GTSRB)

Wurde analysiert bei der International Joint Conference on Neural Networks (IJCNN) 2011

Problemstellung:

Ein Bild, mehrere Klassen Klassifikationsproblem

Mehr als 40 Klassen

Mehr als 50.000 Bilder

Ergebnis:

Erste übermenschliche visuelle Mustererkennung

Fehlerraten:
Mensch: 1.16%, NN:0.56%

Stallkamp et al. 2012

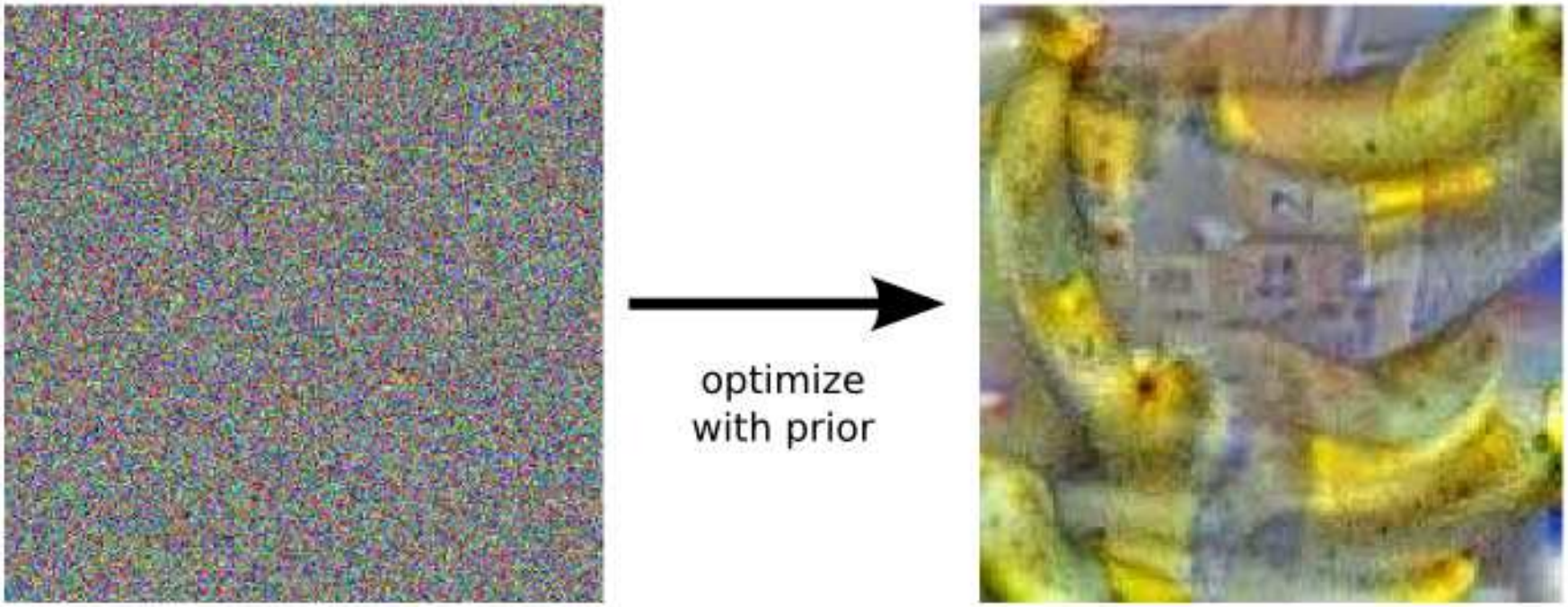
Verwendetes Netz:
Input, Conv., Max., Conv., Max., Conv., Max, Full, Full

Details zu den Gewinnern

Visualisierung von gelernten Neuronalen Netzen

Neuronale Netze zur Objekterkennung in Bildern

Was erkennt ein Neuronales Netz in Rauschen, wenn es Bananen gelernt hat?



Mehr Beispiele

Quelle: Heise: Wovon träumen neuronale Netze?

AlphaGo: Problemstellung Go

2 Spieler (Schwarz, Weiß)

Legen abwechselnd Steine

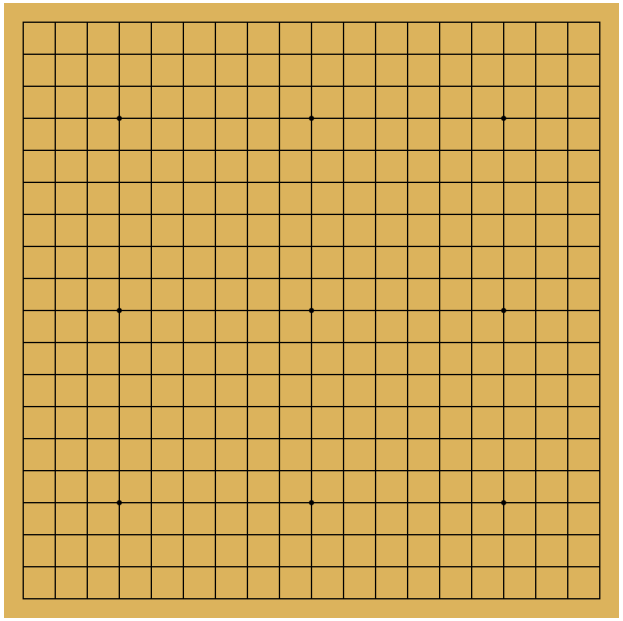
auf einem 19 x 19 Gitter

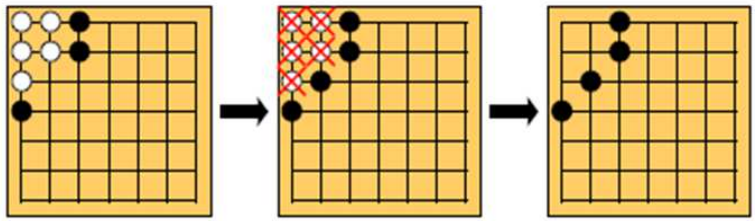
Ziel: Die Größte Fläche einkreisen

eingekreiste Steine werden weggenommen

Anzahl der Möglichkeiten: 250^{150}

Vergleich zu Schach: 35^{80}



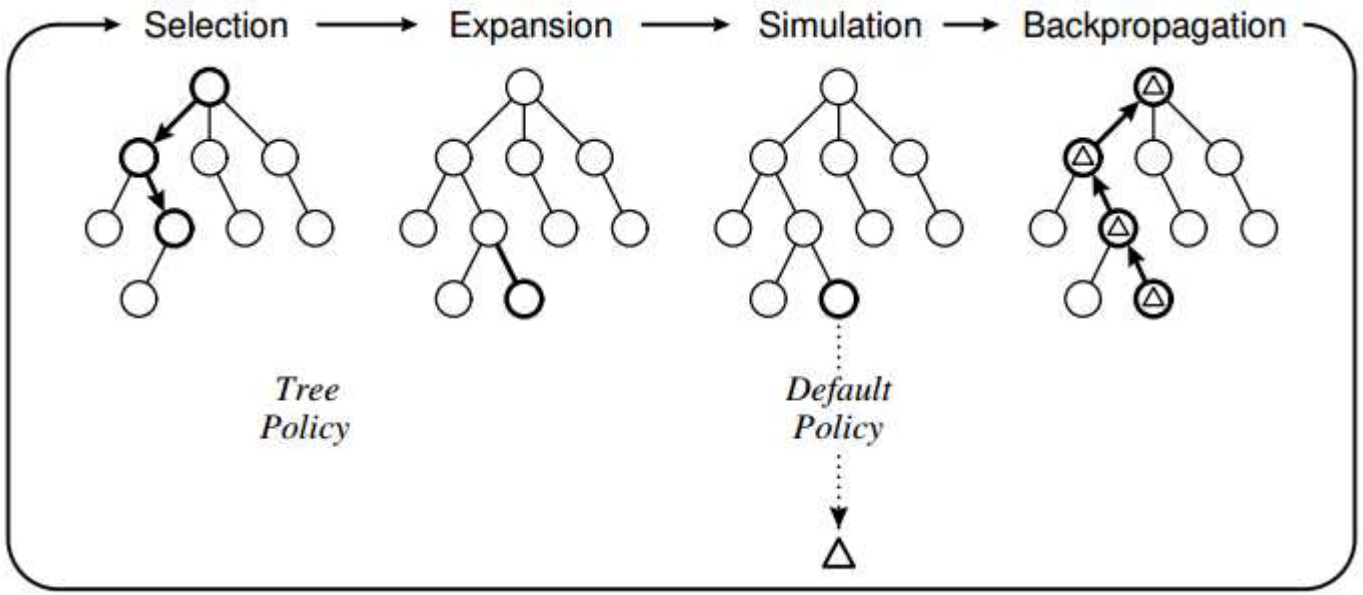


AlphaGo: Ansatz Monte Carlo Suche

Ansatz: Suche im Spielbaum

Lerne Netz 1 für menschenähnliche nächste Züge

Lerne Netz 2 zum Bewerten von Stellungen



AlphaGo: Ergebnisse

Sieg gegen Europameister, Fan Hui: 5 zu 0

Sieg gegen Top10 der Weltrangliste, Lee Sedol: 4 zu 1

AlphaGo	Search threads	CPUs	GPUs	Elo
Asynchronous	1	48	8	2203
Asynchronous	2	48	8	2393
Asynchronous	4	48	8	2564
Asynchronous	8	48	8	2665
Asynchronous	16	48	8	2778
Asynchronous	32	48	8	2867
Asynchronous	40	48	8	2890
Asynchronous	40	48	1	2181
Asynchronous	40	48	2	2738
Asynchronous	40	48	4	2850
Distributed	12	428	64	2937
Distributed	24	764	112	3079
Distributed	40	1202	176	3140
Distributed	64	1920	280	3168

Deep Learning Libraries

Theano
http://deeplearning.net/software/theano/
Python Implementierung für GPU-Verarbeitung von mathematischen Ausdrücken

Tensorflow https://www.tensorflow.org/
Verwendet von Googles DeepMind

Keras
http://keras.io
Python Implementierung, basierend auf Theano oder Tensorflow

Torch
http://torch.ch/
LuaJIT und C/CUDA Implementierung, verwendet bei Facebook, Google, Twitter

DL4J
http://deeplearning4j.org/
Plattformunabhängige Java Implementierung, kompatibel mit Spark, Hadoop

Caffe
http://caffe.berkeleyvision.org/
C++, CUDA Implementierung mit Python und MATLAB Schnittstelle
Sehr schnell, viel verwendet für Bildanalyse z.B. bei Facebook