Vorlesung
MoSim - Modellierung und Simulation 2

Sommersemester 2018

Vorlesung: Jun.-Prof. Dr. Christian Mendl
Mitschrift: Jonas Hippold

1. August 2018

Inhaltsverzeichnis

1 Einleitung **3**

2 Grundlagen künstlicher neuronaler Netze **4**
2.1 Perzeptrons 5
2.2 Neuronen mit Sigmoid-Aktivierungsfunktion 6
2.3 Die Topologie künstlicher Feedforward-Netze 7
2.4 Ein einfaches Netzwerk zur Klassifizierung handschriftlicher Ziffern 7
2.5 Lernen mittels Gradientenverfahren 8
2.6 Der Backpropagation-Algorithmus 10
2.7 Verbesserungen beim Trainieren künstlicher neuronaler Netze 12
2.7.1 Cross-Entropy Kostenfunktion 12
2.7.2 Softmax-Ausgabeschicht und log-likelihood-Kostenfunktion 15
2.7.3 Overfitting und Regularisierung 15

3 Convolutional Neural Networks **21**
3.1 Grundlagen 21
3.2 Backpropagation für Convolutional Layers 23
3.3 Allgemeine Architektur von Convolutional Neural Networks 25
3.4 Anwendung Artistic Style Transfer 27

4 Recurrent Neural Networks **30**
4.1 Grundlegende Architektur 30
4.2 Backpropagation Through Time 32
4.3 Deep Recurrent Neural Networks 34
4.4 Long Short-Term Memory Networks 35
4.4.1 Anwendung: Computergenerierte Bildbeschriftung 37
4.4.2 Erweiterte Version: Bildbeschriftung zusammen mit Aufmerksamkeitsregionen 38

5 Deep Reinforcement Learning **39**
5.1 Markov Decision Processes 39
5.2 Reinforcement Learning, Q-value-Funktion und Q-Learning 44
5.2.1 Bellman-Gleichung für die Q-value-Funktion 45
5.2.2 Q-Learning-Algorithmus 45
5.3 Deep Q-Learning mit Experience Replay 45
5.4 Deep Reinforcement Learning mit Monte-Carlo Tree Search 49
5.5 Architektur von f_ϑ: Residual Network 52

1 Einleitung

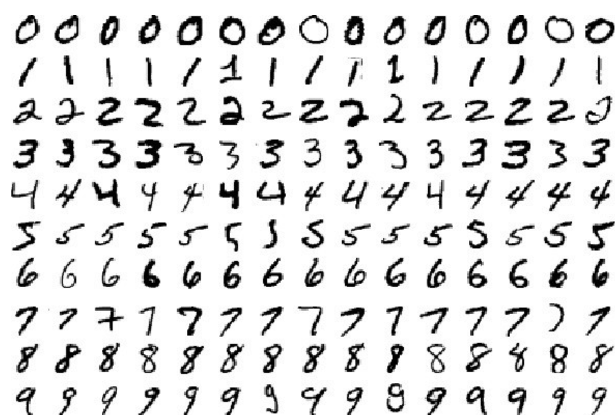
Folien

2 Grundlagen künstlicher neuronaler Netze

artificial neural networks (ANN)

Michael Nielsen: `neuralnetworksanddeeplearning.com`

Motivation: Handschrifterkennung



Ziel: Beliebige handschriftliche Zeichen automatisch identifizieren.

Menschliches Gehirn: Visueller Kortex, Hierarchie $V_1 \to V_2 \to \ldots \to V_5$, wobei die Abstraktion mit dem Index steigt. Vergleiche hierzu D. Hubel, T. Wiesel, Nobelpreis 1981.

Der "konventioneller" Programmieransatz (imperatives Programm mit if-then-else, for-Schleifen, usw.) stellt sich als mühselig bzw. schwierig umzusetzen heraus.

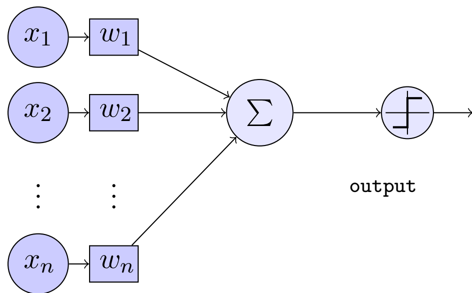
ANNs: "Trainieren" des Netzes mit vielen (zehntausende oder mehr) Trainingspaaren $(x^{(i)}, y^{(i)})$. $x^{(i)}$ ist ein Bitmap-Bild, $y^{(i)}$ die zugehörige Ziffer.

Referenz: (Benchmark-)Datensatz MNIST (modified NIST[1]). Hier ist jedes $x^{(i)}$ ein 28×28 Bitmap mit Graustufen.

[1]National Institute for Standards and Technology (USA)

2.1 Perzeptrons

Frank Rosenblatt 1950, 1960er: Vorläufer "moderner" ANN.



$$\texttt{output} = \begin{cases} 0, & \sum_j w_j x_j \leq \texttt{threshold}, \\ 1, & \sum_j w_j x_j > \texttt{threshold}. \end{cases}$$

$\texttt{output} = 1$ bedeutet, dass die Zelle "feuert".

Beispiel: Entscheidungsprozess "Soll ich das Festival besuchen?"

Kriterien:

1. Wetter
2. Kommen Freunde mit?
3. Gut erreichbar mit öffentlichen Verkehrsmitteln?

$x_1 = 1$ ... Wetter ist gut, $x_1 = 0$ ... Wetter ist schlecht.

Szenario 1: "Will unbedingt, aber nur falls Wetter in Ordnung."

$$w_1 = 6, \quad w_2 = 2, \quad w_3 = 2, \quad \texttt{threshold} = 5.$$

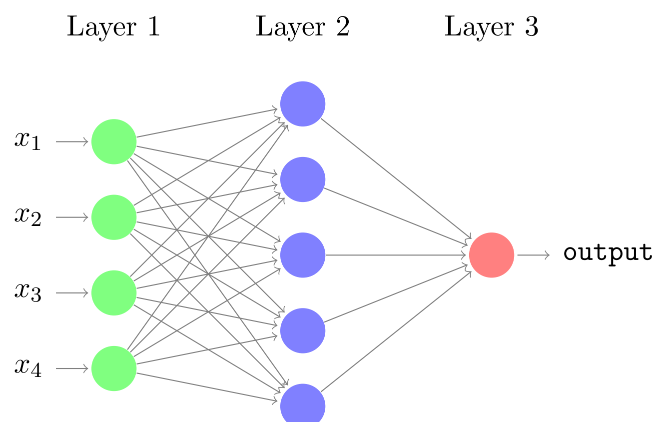
Die Entscheidung hängt nur vom Wetter ab.

Szenario 2:

$$w_1 = 6, \quad w_2 = 2, \quad w_3 = 2, \quad \texttt{threshold} = 3.$$

Besuche das Festival, falls das Wetter gut ist, oder wenn 2. und 3. gleichzeitig erfüllt werden.

Perzeptron-Netzwerk



Jede Zelle hat eigene Gewichte und Thresholds.

Der Output einer Zelle im ersten Layer wird an mehrere Zellen des zweiten Layers weitergeleitet. Jede Zelle produziert aber nur einen Output-Wert, der mehrfach kopiert wird.

Kompakte Notation: Bias $b := -\texttt{threshold}$

$$\texttt{output} = \begin{cases} 0, & \sum_j w \cdot x + b \leq 0, \\ 1, & \sum_j w \cdot x + b > 0 \end{cases}$$

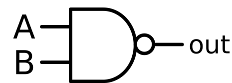
mit den Gewichts- bzw. Eingabevektoren $w = (w_1, w_2, \dots)^\top$ und $x = (x_1, x_2, \dots)^\top$.

Abbildung eines NAND-Gatters als Perzeptron

$$w_1 = -2, \quad w_2 = -2, \quad b = 3$$

x_1	x_2	output
0	0	1
0	1	1
1	0	1
1	1	0

Notation in der Elektrotechnik für NAND-Gatter:



Das NAND-Gate ist universell, man kann beliebige digitale Schaltungen aus Verbindungen dieser Gatter zusammen stellen. Beispiel:

$$\begin{aligned} \texttt{NOT}\, x &= x\, \texttt{NAND}\, x, \\ x_1\, \texttt{OR}\, x_2 &= (\texttt{NOT}\, x_1)\, \texttt{NAND}\, (\texttt{NOT}\, x_2) \end{aligned}$$

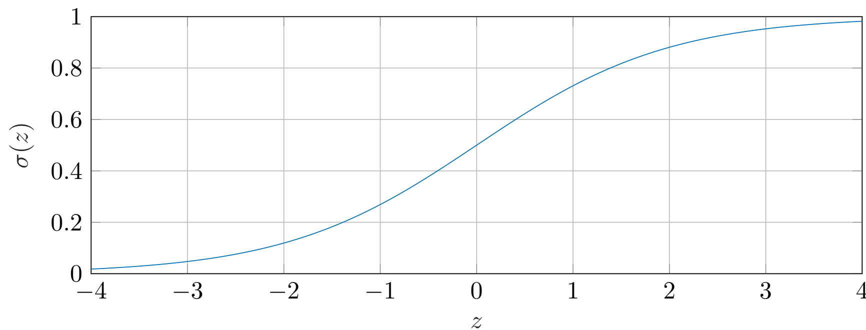
Idee bei ANN: Optimiere die Gewichte und Bias-Werte beim "Training", um die gewünschte Ausgabe zu erhalten. Damit "lernt" das Netzwerk, die geforderte Ausgabe zu erzeugen.

2.2 Neuronen mit Sigmoid-Aktivierungsfunktion

Eine Schwierigkeit des Perzeptron-Modells im Hinblick auf die Optimierung der Parameter ist, dass die Ausgabe keine stetige Funktion ist. Damit ist sie insbesondere nicht differenzierbar und man kann keinen Gradienten definieren.

Als Abhilfe ersetzt man die binäre Ausgabefunktion durch eine stetige Funktion $\sigma(w \cdot x + b)$, die *Sigmoid-Aktivierungsfunktion*, zum Beispiel

$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$



Es gilt $\lim_{z\to-\infty} \sigma(z) = 0$ und $\lim_{z\to\infty} \sigma(z) = 1$.

Die genaue Form von $\sigma(z)$ ist nicht entscheidend, wichtig ist die Differenzierbarkeit. Auch andere Aktivierungsfunktionen sind gebräuchlich.

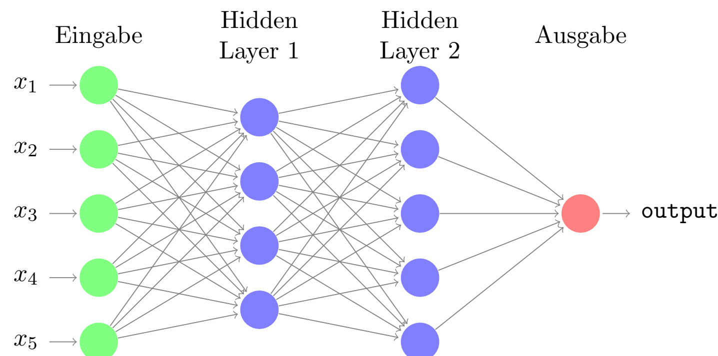
2.3 Die Topologie künstlicher Feedforward-Netze

Feedforward-Netzwerk

Information "fließt" stets von "links nach rechts", das heißt es handelt sich um einen azyklischen gerichteten Graph, es gibt keine "Feedback-Loops".

Input Layer: Zum Beispiel einzelne Pixelwerte eines Bildes bei Schrifterkennung. MNIST: 28×28 Pixel, also $28^2 = 784$ Input-Einheiten.

"Deep" network: Es gibt mehrere Zwischenschichten. Feedforward-Netze unterscheiden sich somit von "Recurrent Neural Networks" (Rekurrente Netze) mit Feedback-Loops.



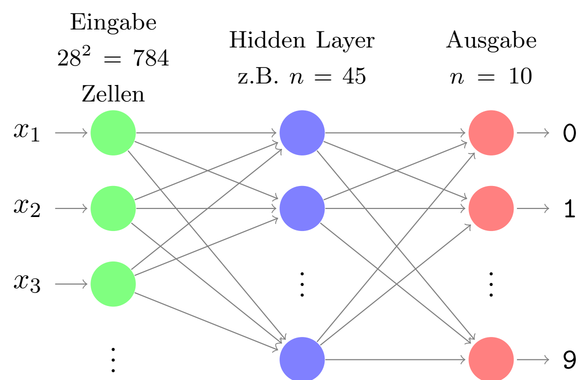
2.4 Ein einfaches Netzwerk zur Klassifizierung handschriftlicher Ziffern

Zwei Teilprobleme:

- Segmentierung: Einteilung eines Bildes von Handschrift in einzelne Zeichen
- Eigentliche Klassifizierung: Zuordnung der Ziffern zu den segmentierten Bildern.

Hier behandeln wir die eigentliche Klassifizierung.

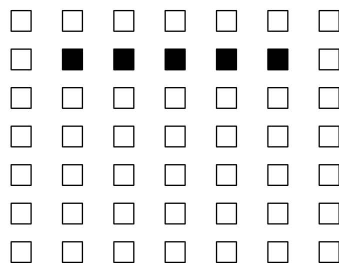
Netzwerk-Topologie



Ziffer mit höchster "Aktivität" (größter Ausgabewert des entsprechenden Neurons) wird als Klassifizierungswert des Netzwerks interpretiert.

Heuristische Interpretation

Nervenzellen im "Hidden Layer" erkenn charakteristische Segmente von Ziffern, zum Beispiel:



Diese Eingabe könnte zu einer 5 oder 7 gehören.

2.5 Lernen mittels Gradientenverfahren

Trainieren und Testen

Training: MNIST-Datensatz enthält 60000 handschriftliche Ziffern als Bilder (28×28 Grauwerte) und entsprechende Klassifizierung von 250 Personen.

Test: 10000 weitere handschriftliche Ziffern einer *anderen* Gruppe von 250 Personen.

Eingabe: $x \in [0,1]^{784}$ (Vektor aus Grauwerten)

Klassifizierung: $y(x) \in \{0,1\}^{10}$ (Indikatorfunktion), Beispiel:

$$y(x) = (0,0,0,0,0,0,1,0,0,0,0) \simeq 6.$$

(Quadratische) Kostenfunktion ("cost" bzw. "loss funciton")

$$C(w,b) = \frac{1}{2n} \sum_{j=1}^{n} \|y(x^{(i)}) - a(x^{(i)}, w, b)\|^2.$$

w, b ist die Menge aller Gewichte und Bias-Werte des Netzwerks, jede Zelle hat individuelle Parameter.

$a(x, w, b)$: Ausgabe des Netzwerks für Eingabe x.

Ziel: Das Netzwerk klassifiziert Ziffern korrekt, also

$$C(w, b) \approx 0.$$

Bemerkung. Die Kostenfunktion ist eine glatte Funktion der Netzwerk-Ausgabe (im Gegensatz zur *Anzahl* der korrekt klassifizierten Bilder).

Fasse Netzwerk-Parameter als $v = (w, b)$ zusammen.

Gradientenverfahren

Wiederholtes Anwenden von

$$v \to v' = v - \eta \nabla C$$

mit der "Lernrate" η, ∇C ist der Gradient bezüglich v.

Schwierigkeit hierbei:

$$C = \frac{1}{n} \sum_{j=1}^{n} C_{x^{(j)}}$$

mit

$$C_x = \frac{1}{2} \|y(x) - a(x, w, b)\|^2$$

und damit

$$\nabla C = \frac{1}{n} \sum_{j=1}^{n} \nabla C_{x^{(j)}}.$$

Das ist eine Summe über alle Trainingspaare $(x^{(i)}, y^{(i)})$, die Auswertung ist "teuer" (großer Rechenaufwand), zum Beispiel 60000 Summanden bei MNIST.

Ausweg

Approximiere den Gradienten mittels kleiner, zufällig ausgewählter Teilmenge aller Trainingspaare (sogenannter "mini batch")

$$\{(x(j_1), y(j_1)), \ldots, (x(j_m), y(j_m))\}.$$

Dann ist

$$\nabla C \approx \frac{1}{m} \sum_{i=1}^{m} \nabla C_{x(j_i)}$$

für einen Schritt des Gradientenverfahrens (mit neuem, disjunkten mini batch für den nächsten Schritt).

Explizit mit w_k und b_l als Gewichte und Bias-Werte:

$$w_k \to w_k' = w_k - \eta \frac{1}{m} \sum_{i=1}^{m} \frac{\partial C_{x(j_i)}}{\partial w_k}$$

$$b_l \to b_k' = b_k - \eta \frac{1}{m} \sum_{i=1}^{m} \frac{\partial C_{x(j_i)}}{\partial b_l}$$

Bemerkung. Ein Grenzfall ist $m = 1$, das sogenannte "online learning". Die Netzwerk-Parameter werden basierend auf einem einzelnen Trainingspaar aktualisiert.

Dieses Verfahren setzt man zum Beispiel bei der Echtzeit-Generierung der Daten ein, dann ist keine Speicherung der Trainingspaare notwendig.

Aber: Große Schwankungen in der Gradientenrichtung sind möglich.

Verbesserte Alternativen zum "stochastic gradient descent" sind ebenfalls gebräuchlich, zum Beispiel Adam optimizer.

2.6 Der Backpropagation-Algorithmus

Ziel: Effiziente Berechnung des Gradienten der Kostenfunktion λC bezüglich $v = (w, b)$.

$$C(v) = \sum_{j=1}^{n} C_{x^{(j)}}(v),$$

somit

$$\nabla C = \sum_{j=1}^{n} \nabla C_{x^{(j)}}(v).$$

Hier: Einzelner Inputvektor x.

Betrachte ein Netzwerk mit L Schichten. a_j^l bezeichne die Ausgabe des j-ten Neurons in Schicht l.

$$a_j^l = \sigma \left(\sum_k w_{jk}^j a_k^{l-1} + b_j^l \right)$$

Matrix-Vektor-Notation:

$$a^l = \sigma(\underbrace{w^l \cdot a^{l-1} + b^l}_{=: z^l}),$$

wobei hier σ komponentenweise angewendet wird.

Hilfsgröße: $\delta_j^l := \frac{\partial C}{\partial z_j^l}$, dann gilt mit der Kettenregel (K)

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} \overset{\text{(K)}}{=} \sum_k \frac{\partial C}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_j^l} = \frac{\partial C}{\partial a_j^L} \cdot \sigma'(z_j^L),$$

weil $\frac{\partial a_k^L}{\partial z_j^l} = 0$ für $k \neq j$ und $a_k^L = \sigma(z_k^L)$.

Zum Beispiel für $C = \frac{1}{2}\|a^L - y\|^2$:

$$\frac{\partial C}{\partial a_j^L} = a_j^L - y_j.$$

Matrix-Vektor-Notation:

$$\delta^L = \operatorname{diag}(\sigma'(z^L)) \cdot \nabla_{a^L} C. \tag{BP1}$$

Ähnliche Berechnung (siehe Tutor-Aufgabe):

$$\delta^l = \operatorname{diag}(\sigma'(z^l)) \cdot (w^{l+1})^\top \cdot \delta^{l+1} \tag{BP2}$$

für alle $l < L$, wobei das erste $\cdot$ eine Matrix-Matrix-Multiplikation darstellt.

Die Bezeichnung Backpropagation kommt von dieser Berechnungsvorschrift. Der Gradient "propagiert rückwarts" von Schicht $l+1$ zu Schicht l.

Mit analoger Berechnung erhält man für die Bias-Werte

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{BP3}$$

und für die einzelnen Gewichte

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l. \tag{BP4}$$

Insbesondere: Der Gradient ist klein für $\sigma'(z_j^l) \approx 0$, das heißt falls die Nervenzelle saturiert ist. Für $|z| \gg 0$ tritt also keine große Veränderung der Gewichte auf. Das kann auch hinderlich sein und das Lernen der Zelle verlangsamen.

Bemerkung. Wiederholtes Anwenden von (BP2) führt auf

$$\delta^l = \operatorname{diag}(\sigma'(z^l)) \cdot (w^{l+1})^\top \cdot \operatorname{diag}(\sigma'(z^{l+1})) \cdot (w^{l+2})^\top \cdot \ldots \cdot \operatorname{diag}(\sigma'(z^L)) \cdot \nabla_{a^L} C.$$

Algorithmus

Direkte Implementierung von (BP1) bis (BP4)

Gegeben: Input-Vektor x

1. Feedforward Pass (Durchlauf):
   for each $l = 1, 2, \ldots, L$:

$$\begin{aligned} z^l &= w^l a^{l-1} + b^l \qquad \text{(with } a^0 = x\text{)} \\ a^l &= \sigma(z^l) \end{aligned}$$

2. Berechne BP1:

$$\delta^L = \operatorname{diag}(\sigma'(z^L)) \cdot \nabla_{a^L} C$$

3. Backpropagate, BP2:
   for each $l = L-1, L-2, \ldots, 1$:

$$\delta^l = \operatorname{diag}(\sigma'(z^l)) \cdot (w^{l+1})^\top \cdot \delta^{l+1}$$

4. Berechne Gradienten, BP3, BP4:

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j, \qquad \frac{\partial C}{\partial b} = \delta^l_j.$$

Bemerkung.

- Der Algorithmus wäre auch für "individuelle" Aktivierungsfunktionen (für jedes Neuron eigene Funktion σ^l_j) anwendbar.
- In der Praxis steigert man die Effizienz durch eine Matrix-Formulierung für einen mini-batch. Das heißt, anstatt den Algorithmus nacheinander auf $x^{(j_1)}, x^{(j_2)}, \ldots, x^{(j_m)}$ anzuwenden, speichert an die auftretenden Vektoren als Spalten einer Matrix:

  $$(z^{l(j_1)} | z^{l(j_2)} | \cdots | z^{l(j_m)}), \quad (a^{l(j_1)} | a^{l(j_2)} | \cdots | a^{l(j_n)}), \quad (\delta^{l(j_1)} | \delta^{l(j_2)} | \cdots | \delta^{l(j_n)}).$$

  Hintergrund ist die optimale Verwendung des Caches bei der Berechnung mit dem Computer. Eine Matrix-Matrix-Multiplikation ist schneller als n-mal eine Matrix-Vektor-Multiplikation auszuführen.
- Naiver Ansatz: Die Numerische Approximation des Gradienten durch Differenzenquotienten, das heißt

  $$\frac{\partial C}{\partial v_j} \approx \frac{C(v + he_j) - C(v)}{h}$$

  mit $0 < h \ll 1$, ist in der Praxis zu rechenaufwendig. Man müsste die Kostenfunktion für jeden Eintrag der Gewichtsmatrizen und Bias-Vektoren neu auswerten (Feedforward-Pass durch das Netzwerk).

  Der Backpropagation-Algorithmus liefert alle Gradienten mit nur einem Feedforward- und Backward-Pass.

2.7 Verbesserungen beim Trainieren künstlicher neuronaler Netze

2.7.1 Cross-Entropy Kostenfunktion

Modellbeispiel: Einzelne Nervenzelle mit Ausgabe

$$a = \sigma(wx + b), \qquad w, b \in \mathbb{R}.$$

$x = 1$, gewünschte Ausgabe $y = 0$.

Training mit dem Gradientenverfahren angewendet auf w, b mit Kostenfunktion

$$C = \frac{1}{2}(y - a)^2.$$

Szenario 1: Startwerte $w = 0{,}6$, $b = 0{,}9$. Anfänglich ist also

$$a = 0{,}817.$$

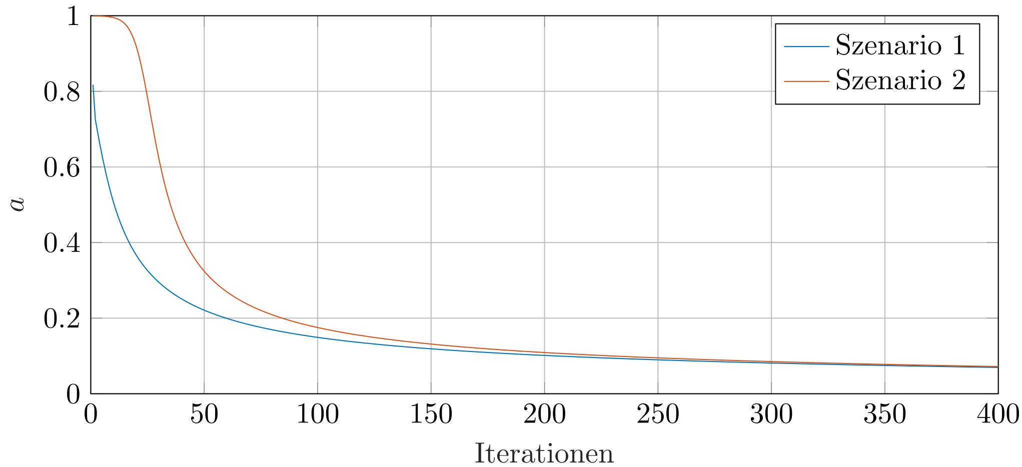
Lernrate $\eta = 0{,}15$. Nach 300 Iterationen ist

$$w = -1{,}28, \qquad b = -0{,}98, \qquad a = \sigma(wx + b) = 0{,}09.$$

Szenario 2: Startwerte $w = 4$, $b = 4$. Anfänglich ist also

$$a = 0{,}817$$

Lernrate $\eta = 0{,}15$.



Wunsch: Die Nervenzelle sollte in Szenario 2 schneller lernen, da die Abweichung von y größer ist als in Szenario 1. Aber das Gegenteil ist der Fall.

Analyse

Mit $C = \frac{1}{2}\|a - y\|^2$ und $x = 1$, $y = 0$ gilt

$$\frac{\partial C}{\partial w} = (a - y)\sigma'(z)x = a\sigma'(z),$$
$$\frac{\partial C}{\partial b} = (a - y)\sigma'(z) = a\sigma'(z).$$

Daher ist in Szenario 2 $\sigma'(z) \approx 0$. Der sehr kleine Gradient führt zum Effekt des "langsamen Lernens".

Dieses Problem tritt auch in allgemeinen Netzwerken auf.

Ein möglicher Ausweg ist die Cross-entropy-Kostenfunktion:

$$C(w, b) := -\frac{1}{n} \sum_{i=1}^{n} \sum_{j} \Big(y_j^{(i)} \log(a_j^L(x^{(i)}, w, b)) + (1 - y_j^{(i)}) \log(1 - a_j^L(x^{(i)}, w, b)) \Big),$$

wobei $a^L(x, w, b)$ die Ausgabe des Netzwerks (Schicht L) für die Eingabe x ist.

Grundeigenschaften der cross-entropy-Kostenfunktion

- $C(w,b) > 0$, denn $0 < a < 1 \Rightarrow -\log(a) > 0$, $-\log(1-a) > 0$, wobei angenommen wird, dass $0 \leq y_j^{(i)} \leq 1$ für alle i, j.
- Für $y_j^{(i)} \in [0,1]$:
$$C(w,b) \to 0 \quad \Leftrightarrow \quad a^L(x^{(i)}, w, b) \to y^{(i)}.$$
Beispiel $y_j^{(i)} = 0$:
$$y_j^{(i)} \log(a_j^L) + (1 - y_j^{(i)}) \log(1 - a_j^L) = 0 + 1 \cdot \log(1 - a_j^L) \to 0 \text{ für } a_j^L \to 0.$$

Der Vorteil dieser Kostenfunktion ist, dass das Problem des langsamen Lernens umgangen wird. Wir setzen in (BP1) ein, zur Vereinfachung zunächst nur für ein einzelnes Trainingspaar (x, y).

$$\begin{aligned}
\delta_j^L &= \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L) \\
&= \frac{\partial}{\partial a_j^L} \left(-\sum_k y_k \log(a_k^L) + (1 - y_k) \log(1 - a_j^L) \right) \cdot \sigma'(z_j^L) \\
&= -\frac{y_j(1 - a_j^L) - (1 - y_j) a_j^L}{a_j^L (1 - a_j^L)} \sigma'(z) \\
&= (a_j^L - y_j) \frac{\sigma'(z_j^L)}{a_j^L (1 - a_j^L)} \\
&= a_j^L - y_j,
\end{aligned}$$

weil $a_j^L = \sigma(z_j^L)$ und
$$\sigma'(z) = \sigma(z)(1 - \sigma(z)).$$

Also ist $\delta^L = a^L - y$. Im Allgemeinen ist
$$\delta^L = \frac{1}{n} \sum_{i=1}^{n} a^L(x^{(i)}, w, b) - y^{(i)}.$$

$\sigma'(z)$ taucht nicht mehr auf!

Die cross-entropy-Kostenfunktion ist bei Sigmoid-Aktivierungsfunktionen typischerweise der quadratischen Kostenfunktion vorzuziehen.

Bemerkung. Alternative Sichtweise: Die cross-entropy-Kostenfunktion ist im Hinblick auf den Backpropagation-Algorithmus äquivalent dazu, die Ausgabeschicht mit $a^L = z^L$ statt $\sigma(z^L)$ auszuführen und die quadratische Kostenfunktion $\frac{1}{2}\|a^L - y\|^2$ zu verwenden, denn in (BP1) ergibt sich dann
$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \cdot 1 = a_j^L - y_j.$$

2.7.2 Softmax-Ausgabeschicht und log-likelihood-Kostenfunktion

Die Ausgabe des Netzwerks sei nun

$$a_j^L = \frac{\exp(z_j^L)}{\sum_k \exp(z_k^L)}.$$

Man kann a_j^L als Wahrscheinlichkeitsverteilung interpretieren, da $a_j^L > 0$, $\sum_j a_j^L = 1$.

Anwenden von log auf beiden Seiten liefert

$$z_j^L = \log(a_j^L) + \text{const}\,.$$

Die Softmax-Ausgabeschicht weicht vom bisherigen Schema ab, da a_j^L nicht mehr nur von z_j^L abhängt, sondern von allen Einträgen in z^L.

Die kanonische Wahl der Kostenfunktion "passend" zu Softmax ist die *log-likelihood-Kostenfunktion*:

$$C = -\log(a^L \cdot y) = -\log(a^L \cdot e_j) = \log(a_j^L).$$

Die Ableitung der log-likelihood-Kostenfunktion bezüglich der Netzwerk-Parameter in der Ausgabeschicht ist

$$\frac{\partial C}{\partial b_j^L} = \frac{\partial C}{\partial z_j^L} = \frac{\partial}{\partial z_j^L}\left(-\log\left(\sum_k e^{z_k^L} y_k\right) + \log\left(\sum_k (e^{z_k^L})'\right)\right),$$

mit $z_j^L = \sum_k w_{jk}^L a_k^{L-1} + b_j^L$ und $y = e_j$ folgt

$$\frac{\partial C}{\partial b_j^L} = -1 y_j + \frac{e^{z_j^L}}{\sum_k e^{z_k^L}} = a_j^L - y_j.$$

Analog erhalten wir

$$\frac{\partial C}{\partial w_{jk}^L} = \frac{\partial C}{\partial z_j^L} \underbrace{\frac{\partial z_j^L}{\partial w_{jk}^L}}_{=a_j^{L-1}} = a_k^{L-1}(a_j^L - y_j).$$

Also stimmen $\frac{\partial C}{\partial b_j^L}$ und $\frac{\partial C}{\partial w_{jk}^L}$ mit der cross-entropy-Kostenfunktion überein.

In der Praxis funktionieren sigmoid mit cross-entropy und softmax mit log-likelihood typischerweise ähnlich gut. Ein Vorteil der softmax-Ausgabeschicht ist die Interpretierbarkeit als Wahrscheinlichkeitsverteilung.

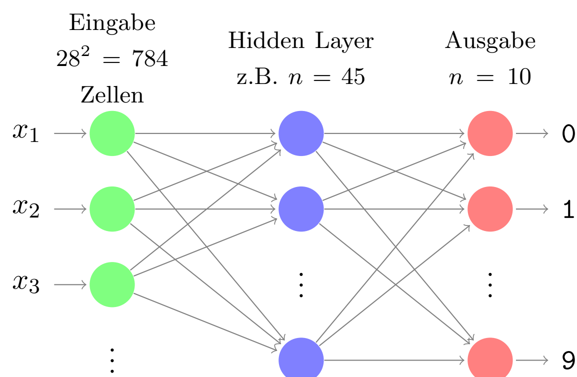
2.7.3 Overfitting und Regularisierung

(Naheliegende) Kritik an ANN: sehr viele Parameter. Das kann zu Overfitting-Problemen führen.

John von Neumann: "With four parameters I can fit an elephant, and with five I can make him wiggle his trunk."

ANN für MNIST: Größenordnung 10^4 Parameter. State-of-the-Art-ANNs können 10^6 und mehr Parameter aufweisen.

Experiment: Trainiere Netzwerk anhand der ersten 1000 Trainingspaare des MNIST-Datensatzes (so dass Overfitting-Probleme evident werden).



Wir nutzen die cross-entropy-Kostenfunktion.

Die Kostenfunktion konvergiert gegen 0 (wie erwünscht), es ist kein Indiz für Overfitting erkennbar.

Erinnerung "Epoche": Jedes Datenpaar wurde beim SGD genau einmal verwendet.

Aber: Die Klassifizierungsgenauigkeit beim Testdatensatz ist mangelhaft. Hier wird die Handschrift *anderer* Personen als im Trainingsdatensatz erkannt.

Aufgrund des Overfittings erreicht der Fehler nur einen Plateau-Wert.

Zum Vergleich: Klassifikationsgenauigkeit beim Traniningsdatensatz:

Das Netzwerk klassifiziert schon nach wenigen Epochen den gesamten Datensatz korrekt.

Interpretation: Das Netzwerk "lernt die Trainingsdaten auswendig" anstatt allgemeine Indikatoren für handschriftliche Ziffern zu finden. Dadurch lässt es sich nicht gut auf unbekannte Handschriften verallgemeinern.

Experiment 2: Benutze alle 50000 Datenpaare zum Training.

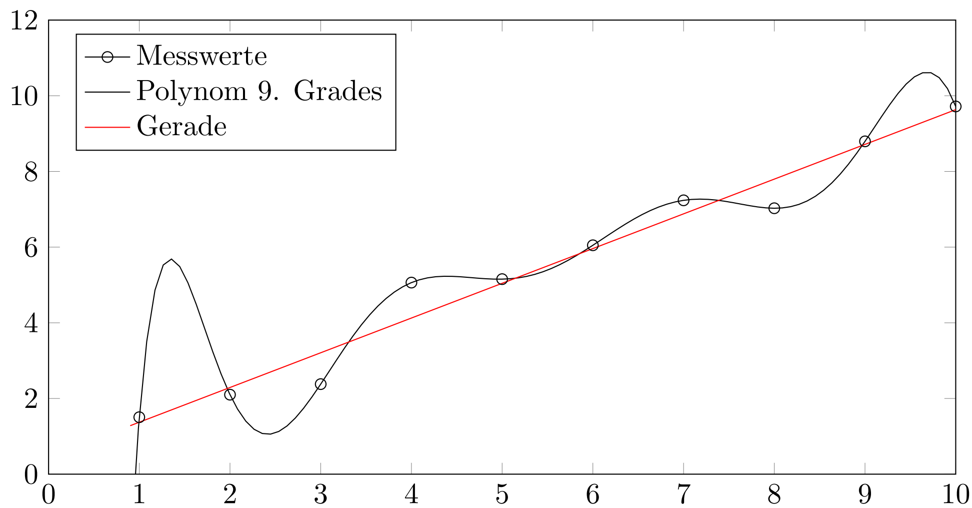
Jetzt beträgt der Overfitting-Fehler noch etwa 2,5 %.

Interpretation: Die Generalisierung auf eine unbekannte Handschrift funktioniert gut.

L2-Regularisierung

Idee: Einträge der Gewichtsmatrizen w_k^l sollten nicht "zu groß" sein, um Overfitting zu vermeiden.

Analogie:



Parameter:

$$y_1 = a_9 x^9 + a_8 x^8 + \dots a_1 x + a_0$$
$$y_2 = 0x^9 + 0x^8 + \dots + ax + b$$

Das gefittete Polynom vom Grad 9 beschreibt die Datenpunkte exakt, aber auf zusätzlichen Datenpunkten ist die Generalisierung wahrscheinlich sehr schlecht.

In künstlichen neuronalen Netzen wird das so umgesetzt: Addiere einen Term zur Kostenfunktion, typischerweise *L2-Regularisierung*:

$$C = C_0 + \lambda \frac{1}{2n} \sum_k w_k^2 = C_0 + \lambda \frac{1}{2n} \|w\|^2$$

mit der ursprünglichen Kostenfunktion C_0.

n ist die Anzahl der Trainingspaare, λ ein Regularisierungsparameter.

Interpretation der neuen Kostenfunktion: Kompromiss zwischen C_0, der Abweichung von der Referenzausgabe, und möglichst kleinen Gewichten.

Gradientenverfahren:

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C_0}{\partial w_{jk}^l} + \frac{\lambda}{n} w_{jk}^l,$$
$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C_0}{\partial b_j^l}$$

Einzelner Schritt:

$$b_j^l \to b_j^l - \eta \frac{\partial C_0}{\partial b_j^l}$$

$$w_{jk}^l \to w_{jk}^l - \eta \left(\frac{\partial C_0}{\partial w_{jk}^l} + \frac{\lambda}{n} w_{jk}^l \right)$$
$$= \left(1 - \frac{\lambda \eta}{n}\right) w_{jk}^l - \eta \frac{\partial C_0}{\partial w_{jk}^l}$$

Der Term $1 - \frac{\lambda\eta}{n} < 1$ wirkt als *Reskalierungsfaktor*.

Experiment: Wie zuvor (1000 Trainingspaare), nur mit zusätzlicher L2-Regularisierung, $\lambda = 0{,}1$.

Die Kostenfunktion verhält sich qualitativ ähnlich zum Fall ohne Regularisierung.

Genauigkeit beim Testdatensatz:

Die Performance ist besser, obwohl keine neuen Trainingsdaten verwendet wurden. Es tritt auch (noch) kein Plateau auf.

L1-Regularisierung

Angepasste Kostenfunktion:

$$C = C_0 + \frac{\lambda}{n} \sum_k |w_k|$$

mit dem Vektor aller Gewichte w, $\sum |w_k| = \|w\|_1$.

Das Gradientenverfahren wird dann zu

$$\frac{\partial C}{\partial w^l_{jk}} = \frac{\partial C_0}{\partial w^l_{jk}} + \frac{\lambda}{n} \operatorname{sgn}(w^l_{jk})$$

mit

$$\operatorname{sgn}(x) = \begin{cases} 1, & x > 0, \\ 0, & x = 0, \\ -1, & x < 0. \end{cases}$$

Ein Parameterupdate erfolgt über

$$w^l_{jk} \to w^l_{jk} - \eta \frac{\partial C}{\partial w^l_{jk}} = w^l_{jk} - \eta \frac{\partial C_0}{\partial w^l_{jk}} - \frac{\eta\lambda}{n} \operatorname{sgn}(w^l_{jk}).$$

Im Vergleich zur L2-Regularisierung nehmen bei der L1-Regularisierung also große Gewichte langsamer und kleine Gewichte schneller ab. Das heißt, die Gewichte im Netzwerk werden auf einige "wichtige" Verbindungen konzentriert. w^l besitzt also wenige Einträge mit $|w^l_{jk}|$ groß, die anderen Einträge sind klein.

Dropout

Temporäres Löschen (pro mini-batch) der Hälfte aller versteckten Zellen, das heißt Zellen in den hidden layers.

Pseudocode. Für jeden mini-batch:

- Wähle zufällig die Hälfte aller versteckten Zellen aus ("Dropout-Menge") und entferne diese Zellen temporär aus dem Netzwerk.
- Führe einen Schritt des Gradientenverfahrens mit den verbleibenden Zellen durch.
- Stelle die entfernten Zellen wieder her.

Konzeptionelle Idee von Dropout: Trainiere mehrere Netze parallel, die finale Ausgabe ist eine "Mehrheitsabstimmung" der einzelnen Netze.

Alternative Sichtweise: Die "Entscheidungsfindung" einer Zelle kann nicht auf einer einzelnen Zelle aus der vorherigen Schicht beruhen, da diese temporär entfernt worden sein könnte. Die Zelle muss verschieden Evidenzen abwägen. Das Netzwerk wird also robuster.

Künstliche Erweiterung des Trainingsdatensatzes

Wunsch: Möglichst großer Trainingsdatensatz, also zum Beispiel möglichst viele handschriftliche Ziffern. In der Praxis ist das aber oft schwer oder sehr aufwändig.

Ein alternativer Ansatz ist, den bestehenden Trainingsdatensatz $(x^{(i)}, y^{(i)})$, $i = 1, \ldots, N$ durch leichte Modifikationen M_j der $x^{(i)}$ zu verändern. Das heißt der Datensatz vergrößert sich zu

$$(x^{(i)}, y^{(i)}), (M_1(x^{(i)}), y^{(i)}), \ldots, \qquad i = 1, \ldots, N.$$

Den MNIST-Datensatz könnte man zum Beispiel leicht drehen, verschieben "schräg stellen", usw.

In der Spracherkennung könnte man Samples langsamer oder schneller abspielen und ein Hintergrundrauschen hinzufügen.

Initialisierung der Gewichtsmatrizen

Betrachte eine einzelne Zelle mit n Eingaben:

$$z = \sum_{j=1}^{n} w_j x_j + b, \qquad a = \sigma(z).$$

Initialisiere die $w_j \sim \mathcal{N}(0, s^2)$, also unabhängige Normalverteilungen mit Varianz s^2 bzw. Standardabweichung s, $\frac{w_j}{s} \sim \mathcal{N}(0,1)$. Die Bias-Werte werden mit $b \sim \mathcal{N}(0,1)$ initialisiert.

Wie sollte s gewählt werden? Für eine einfache Abschätzung nehme an, dass die Hälfte der Eingabewerte $x_i = 1$, die andere Hälfte $x_i = 0$ sind. Somit ist die Varianz von z:

$$\operatorname{var}(z) = \frac{n}{2} \operatorname{var}(w_j) + \operatorname{var}(b) = \frac{n}{2} s^2 + 1.$$

$\operatorname{var}(z)$ sollte von der Größenordnung 1 sein, sonst saturiert $a = \sigma(z)$, wenn $|z| \gg 1$.

Wähle also $s \approx \frac{1}{\sqrt{n}}$.

Experiment: MNIST-Klassifizierungserfolg des Testdatensatzes

Wahl der Hyperparameter

Die *Hyperparameter* sind die Lernrate η, der Regularisierungsparameter λ, die Anzahl versteckter Schichten, die Anzahl der Zellen in jeder Schicht usw.

η und λ beziehen sich nur auf das Training, sie sind keine Parameter des Netzes an sich.

Heuristische Strategien

Erstes Ziel ist ein erkennbarer Trainingserfolg, das heißt das Netzwerk soll besser als ein Zufallsgenerator funktionieren.

Vereinfachungen: Reduzierung der Anzahl der möglichen Ausgaben, zum Beispiel bei MNIST zunächst nur die Ziffern 0 und 1 betrachten.

"fail fast", das heißt schnelles Feedback. Zum Beispiel Validierung nach jeder Trainingsepoche, kleinerer Testdatensatz, usw.

Wahl der Lernrate η

Es muss ein Kompromiss gefunden werden. Sie sollte nicht zu klein sein (sonst langsames Lernen), aber auch nicht zu groß (Overshooting-Probleme).

Ein guter Kompromiss:

$$\eta = \frac{1}{2} \text{ “overshooting-threshold”}.$$

Oft ist es sinnvoll, die Lernrate im Verlauf des Trainings zu reduzieren.

Anzahl der Trainingsepochen

"Early stopping": Beende das Training, wenn der Klassifizierungserfolg beim Trainingsdatensatz nicht mehr zunimmt.

Automatisierung

Zum Beispiel Bayes'sche Optimierungsalgorithmen zur Suche guter Hyperparameter.

3 Convolutional Neural Networks

3.1 Grundlagen

Bisher: $28^2 = 784$ Pixel der MNIST-Bilder werden im Eingabevektor angereiht. Das Netz nutzt "fully connective layers", jede Zelle ist gleichwertig mit allen Zellen der vorherigen Schicht verbunden.

Dadurch verlieren wir die räumliche Struktur, die 2D-Nachbarschaftsbeziehung der Pixel. Ein Ansatz zur Berücksichtigung dieser Struktur sind *Convolutional Neural Networks* (CNNs).

Bausteine sind lokal rezeptive Felder, uniforme Gewichte und Bias-Werte, Pooling.

Lokal rezeptive Felder

Die Eingabe eines jeden Neurons ist ein lokaler Ausschnitt ("rezeptives Feld") des Bildes.

Das rezeptive Feld wird sukzessiv über das Eingabefeld verschoben. Diese Verschiebung erfolgt jeweils um die "stride length", eine feste Anzahl Pixel, nach rechts bzw. unten, typischerweise ist

$$\text{stride length} = 1 \text{ Pixel}.$$

Für 28×28-Bilder und ein 5×5-Feld benötigt man also 24×24 Zellen im versteckten Layer, da das rezeptive Feld nicht über die Bildränder hinaus laufen soll.

Uniforme Gewichte und Bias-Werte

Idee: Verwende dieselbe Gewichtsmatrix und Bias-Werte für jedes Neuron. Für ein 5×5 rezeptives Feld ist die Ausgabe des (j_x, j_y)-ten Neurons:

$$a^l_{j_x,j_y} = \sigma\left(\sum_{k_x,k_y=-2}^{2} w^l_{k_x,k_y} a^{l-1}_{j_x+k_x,j_y+k_y} + b^l\right). \qquad (*)$$

Dabei ist zu beachten, dass $w^l_{k_x,k_y}$ und b^l nicht von (j_x, j_y) abhängen.

Man kann diese Ausgabe als Kreuzkorrelation bzw. Faltung (convolution) der Eingabe mit $w_{-k_x,-k_y}$ in 2D interpretieren, wobei $w_{-k_x,-k_y}$ der *Kern* bzw. *Filter* der Faltung ist:

$$\sum_k w^l_k a^{l-1}_{j+k} = \sum_{\tilde{k}} w^l_{-\tilde{k}} a^{l-1}_{j-\tilde{k}}$$

mit $\tilde{k} := -k$.

Analog in 1D: Zum Beispiel mit $w = (0{,}75, 1, -0{,}5)$ ist

$$\begin{aligned} y_1 &= 0{,}75a_0 + a_1 - 0{,}5a_2, \\ y_2 &= 0{,}75a_1 + a_2 - 0{,}5a_3, \\ &\vdots \end{aligned}$$

Eine mögliche Interpretation der Operation $(*)$ ist die *feature map*, sie detektiert ein "feature" des Eingabefelds, zum Beispiel eine schräge Kante.

$$w_l = \begin{pmatrix} -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & +1 \\ -1 & -1 & -1 & +1 & -1 \\ -1 & -1 & +1 & -1 & -1 \\ -1 & +1 & -1 & -1 & -1 \end{pmatrix}$$

Man benötigt dann mehrere feature maps für die Bildklassifizierung, die jeweils eine eigene Gewichtsmatrix und einen eigenen Bias-Wert für jedes feature haben.

Die Anzahl der Parameter für ein Netzwerk mit $k \times k$-Feldern ist nun

$$\#\text{ features} \cdot (k \times k + 1).$$

Für 20 features ergeben sich also 520 Parameter. Im Vergleich dazu benötigt ein Netz mit der bisherigen "fully connected"-Topologie und 30 Zellen im ersten hidden layer

$$784 \cdot 30 + 30 = 23\,550 \gg 520$$

Parameter. Das CNN ist also wesentlich ökonomischer.

Pooling

Idee: "Kondensiere" die Information der Convolution-Schicht, "coarse graining" der räumlichen Auflösung. Zum Beispiel könnte es wichtig sein, ob ein schräger Strich in der Eingabe enthalten ist, aber nicht wichtig, wo genau er sich befindet.

Zum Beispiel wird beim 2×2 max-Pooling das Maximum der Ausgaben in 2×2-Regionen bestimmt, beim L2-Pooling berechnet man die L2-Norm der vier Ausgaben in der Region.

Die Pooling-Schicht verkleinert also die Anzahl der Zellen.

Somit ist die Topologie für Convolution und Pooling:

3.2 Backpropagation für Convolutional Layers

Die Ausgabe eines Felds ist

$$a_j^l = \sigma\left(\sum_k w_k^l a_{j+k}^l + b^l\right),$$

wobei $j = (j_x, j_y)$, $k = (k_x, k_y)$ zweidimensional sind. Analog für 3D, 12D usw.

Wie bisher sei $\delta_j^l = \frac{\partial C}{\partial z_j^l}$.

(BP1-conv) ist identisch mit (BP1):

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L) \tag{BP1-conv}$$

Für (BP2-conv) gilt:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l}.$$

Es ist

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = \frac{\partial}{\partial z_j^l}\left(\sum_m w_m^{l+1}\sigma(z_{k+m}^l) + b^{l+1}\right) = w_{j-k}^{l+1}\sigma'(z_j^l),$$

weil die Ableitung nur für $k + m = j$ ungleich 0 ist. Einsetzen:

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{j-k}^{l+1}\sigma'(z_j^l) = \sigma'(z_j^l)\sum_{\tilde{k}} w_{\tilde{k}}^{l+1}\delta_{j-\tilde{k}}^{l+1} \tag{BP2-conv}$$

mit $\tilde{k} := -k$. Das ist wieder eine Faltung mit Kern w^{l+1}.

Weiterhin ist

$$\frac{\partial C}{\partial b^l} = \sum_j \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial b^l} = \sum_j \delta_j^l \tag{BP3-conv}$$

und

$$\frac{\partial C}{\partial w_k^l} = \sum_j \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_k^l} = \sum_j \delta_j^l a_{j+k}^{l-1}. \tag{BP4-conv}$$

Alternative Darstellung der Backpropagation

Motivation: Kombination verschiedener Schichten erfordert separate Backpropagation-Gleichungen für jeden Schicht-Typ, ausgedrückt durch Ein- und Ausgabe (anstatt mittels δ_j^l).

Bezeichne die Eingabe mit $x = a^{l-1}$, die Ausgabe mit $a = a^l$.

Convolutional layer (zunächst für ein einzelnes Feature):

$$a_j = \sigma\left(\sum_k w_k x_{j+k} + b\right), \qquad j, k \in \mathbb{Z}.$$

Konzeptionelle Vereinfachung: Fasse die Aktivierungsfunktion σ als eigene Schicht auf. Dann berechnet der convolutional layer lediglich den linearen Anteil,

$$a_j = \sum_k w_k x_{j+k} + b.$$

Entsprechender Pseudocode für forward pass:

```
for i = 0, 1, ...
    for j = 0, 1, ...
        a[i,j] = sum( w * x[i : i + height, j : j + width]) + b
```

Dabei ist `*` eine komponentenweise Multiplikation und `height` und `width` sind die Dimensionen des Filters w, zum Beispiel 5×5.

Idee für Backpropagation: Analogie

$$\frac{\partial}{\partial x_j} f\left(\sum_k w_k x_k\right) = f'(a) w_j, \qquad \nabla_x f\left(\sum_k w_k x_k\right) = f'(a) \cdot \vec{w}.$$

f spielt die Rolle der Kostenfunktion.

Pseudocode für Backpropagation: Wir haben die vorgegebene Variable

$$\mathrm{d}a[i,j] = \frac{\partial C}{\partial a_{ij}},$$

berechnet werden sollen

$$\mathrm{d}x[i,j] = \frac{\partial C}{\partial x_{ij}}, \qquad \mathrm{d}w[k_1,k_2] = \frac{\partial C}{\partial w_k}, \qquad \mathrm{d}b = \frac{\partial C}{\partial b}.$$

```
dx = 0  ## Als Matrix mit selber Dim. wie x
dw = 0  ## Als Matrix mit selber Dim. wie w

for i = 0, 1, ...
    for j = 0, 1, ...
        dx[ i : i + height, j : j + width ] += da[ i,j ] * w
        dw += da[i,j] * x[ i : i + height, j : j + width ]

db = sum( da )
```

Modifikation für mehrere Features:

$$a_{f,ij} = \sum_k w_{f,k} x_{j+k} + b_f.$$

```
for f = 0, 1, ...
  for i = 0, 1, ...
    for j = 0, 1, ...
      a[ f, i, j ] = sum( w[f] * x[ i : i + height, j : j + width ] ) + b[f]
```

Für die Backpropagation gilt das analog.

Modifikationen für allgemeine "stride length": Ersetze

```
x[ i : i + height, j : j + width ]
```

durch

```
x[ i * stride : i * stride + height, j * stride : j * stride + width ]
```

3.3 Allgemeine Architektur von Convolutional Neural Networks

Für Farbbilder benötigen wir eine zusätzliche Dimension. Die Eingabe besteht nun aus drei Farbkanälen (RGB, rot, grün, blau), sie ist also ein 3D-Volumen $3 \times H \times W$.

Die hidden layers sind ebenfalls dreidimensional. "Features" bilden eine zusätzliche Dimension. Somit ist die allgemeine Formel für einen convolutional layer:

$$a_{f,j} = \sigma\left(\sum_{c,k} w_{f,c,k} x_{c,j+k} + b_f\right).$$

Für 2×2-Pooling wird die Höhe und Breite um den Faktor zwei reduziert.

Die Abfolge der verschiedenen Schicht-Typen, die Anzahl der Features und weitere Parameter des Netzwerks sind je nach Anwendung unterschiedlich. Meist können sie nur durch heuristisches Ausprobieren bestimmt werden.

Zero Padding

Oft üblich ist es, die Eingabe mit Nullen entlang der Bilddimension zu "padden". Das heißt, man ergänzt die Eingabe um eine Anzahl von Nullen, um die Programmierung zu vereinfachen. Zum Beispiel kann man für ein 3×3-Filter mit zwei Nullen padden, dann ist keine besondere Behandlung des Randbereichs mehr nötig.

Ausgabedimension eines convolutional layers

Hier betrachten wir nur die räumliche Bilddimension (Höhe und Breite). Die Anzahl der Channels bzw. Features sind separate, von der Bilddimension unabhängige Größen.

Entlang einer Richtung:

- W: Breite des Eingabe-Bilds,
- R: Breite des Filters,
- P: Padding, $P = 0, 1, \ldots,$
- S: Stride length.

Nach dem Padding ist die Breite $W + 2P$.

Wie oft muss man das Filter verschieben?

$$R + nS = W + 2P \quad \Rightarrow \quad n = \frac{1}{S}(W + 2P - R).$$

Die Breite der Ausgabe ist somit

$$W_{\text{out}} = n + 1 = \frac{1}{S}(W + 2P - R) + 1.$$

Insbesonder muss $W + 2P - R$ ein ganzzahliges Vielfaches von S sein.

Optimierte Implementierung der Faltungsoperation

Ziel ist die Laufzeitoptimierung der Rechenoperation eines convolutional layers.

$$a_{f,j} = \sum_{c,k} w_{f,c,k} x_{c,j+k} + b_f.$$

Es ist naheliegend, diese Operation mit der FFT durchzuführen (siehe Übungsaufgabe).

Aber man kann dann die räumliche Lokalität des Filters nicht ausnutzen.

Die FFT benötigt zero padding, um das Filter auf die selbe Dimension wie das Eingabebild zu erweitern. Die FFT ist nur für Vektoren gleicher Größe anwendbar. Also wird zusätzlicher Speicher benötigt.

Außerdem sind stride lengths ≥ 2 mittels FFT technisch kompliziert umzusetzen.

Anstatt die FFT zu verwenden, kann man die Faltungsoperation als Matrix-Multiplikation audrücken und die hochoptimierte BLAS-Funktion GEMM (general matrix multiplication) verwenden.

Illustration für ein Eingabe-Feld der Dimension $H = W = 3$ mit $C = 1$ "channels" und 2×2-Filter mit $F = 2$ "features"

Eingabe	Filter $f = 1$	Filter $f = 2$
$\begin{matrix} x_{00} & x_{01} & x_{02} \\ x_{10} & x_{11} & x_{12} \\ x_{20} & x_{21} & x_{22} \end{matrix}$	$\begin{matrix} w_{000} & w_{001} \\ w_{010} & w_{011} \end{matrix}$	$\begin{matrix} w_{100} & w_{101} \\ w_{110} & w_{111} \end{matrix}$

Die Verarbeitung der Eingabe im convolutional layer lässt sich als Als Matrix-Matrix-Multiplikation darstellen:

$$\begin{pmatrix} x_{00} & x_{01} & x_{10} & x_{11} \\ x_{01} & x_{02} & x_{11} & x_{12} \\ x_{10} & x_{11} & x_{20} & x_{21} \\ x_{11} & x_{12} & x_{21} & x_{22} \end{pmatrix} \cdot \begin{pmatrix} w_{000} & w_{001} & w_{010} & w_{011} \\ w_{100} & w_{101} & w_{110} & w_{111} \end{pmatrix}$$

Für mehrere Channels wird x zusätzlich zu "im 2 col" noch in die einzelnen Kanäle aufgeteilt. Die Gewichte erhalten einen zusätzlichen Index, $w_{i,0,i,i}$ und $w_{i,1,i,i}$ sind dann die Parameter für Kanal 0 bzw. 1.

Dimension allgemein

Die Eingabe x ist ein Tensor mit Dimension

$$N \times C \times H \times W.$$

Dabei steht N für den mini-batch, C für die Anzahl der Channels, H und W für die Höhe und Breite.

Der convolutional layer ("im 2 col") erzeugt eine Matrix der Dimension

$$(C \cdot Q \cdot R) \times (N \cdot W_{\text{out}} \cdot H_{\text{out}}),$$

Das Filter ist dann ein Tensor der Dimension

$$F \times C \times Q \times R$$

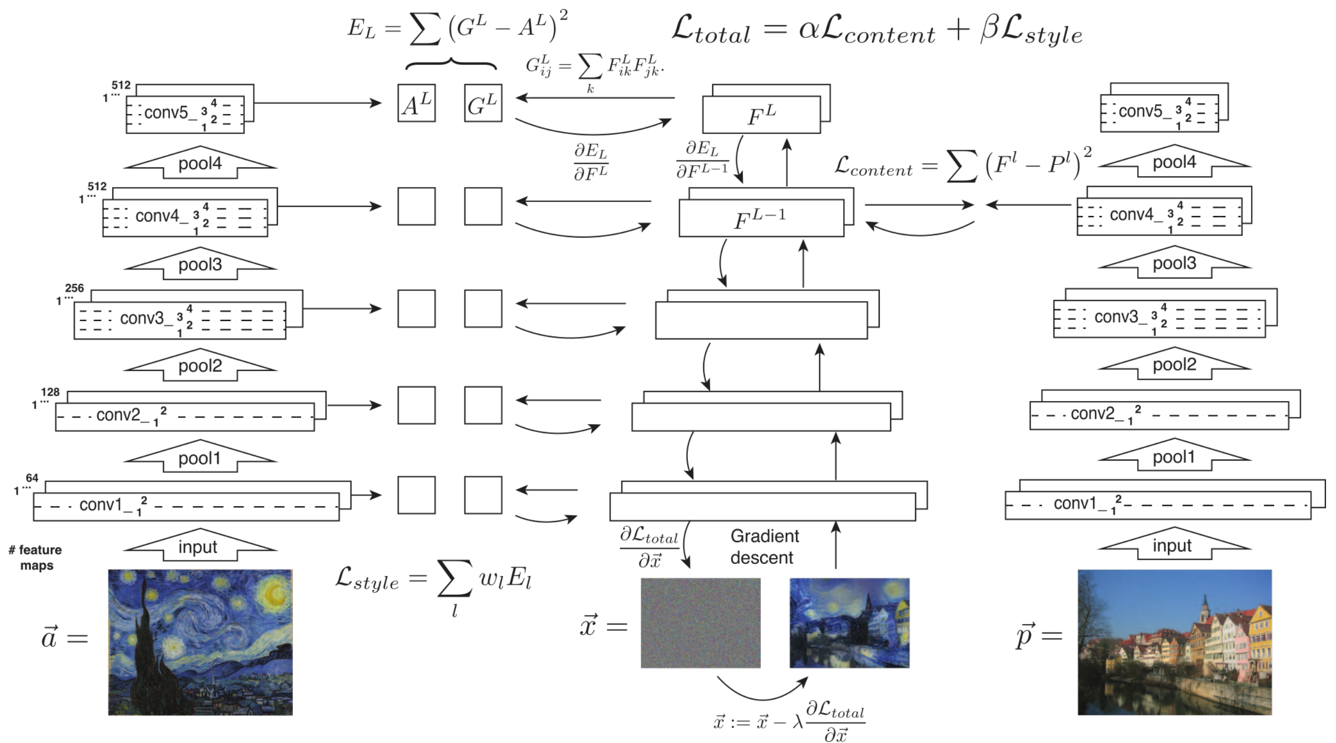
mit F der Anzahl der features, C der Anzahl der channels, Q und R die Höhe und Breite.

3.4 Anwendung Artistic Style Transfer

L. Gatys, A. Ecker, M. Bethge (2016)

Idee: Verwende ein deep convolutional network, um den semantischen Bildanteil (Objekte, allgemeine Szenerie, "content") vom Zeichenstil ("style") zu separieren.

Verwende ein bereits trainiertes Netzwerk aus der Literatur (im obigen Paper das VGG-Net, conv – max-pool) mit drei verschiedenen Eingaben.



Bemerkung. Grundlegender Unterschied zum bisherigen Trainieren: Die Netz-Parameter bleiben unverändert. Stattdessen wird das Gradientenverfahren auf $\vec{x}$ angewendet. Dazu kann man wieder Backpropagation verwenden.

Die Abweichung zwischen Content-Bild (Foto) und $\vec{x}$ auf Schicht ℓ ist

$$\mathcal{L}_{\text{content}}(x_{\text{content}}, \vec{x}, \ell) = \frac{1}{2} \| a^\ell_{\text{content}} - \hat{a}^\ell \| = \frac{1}{2} \sum_{f,j} (a^\ell_{\text{content},f,j} - \hat{a}^\ell_{f,j})^2$$

f ist der feature index, j ein 2D-Index, ℓ der Layer-Index.

Repräsentation des Zeichenstils

Überlegung: Die absolute räumliche (Pixel-)Position ist nicht relevant. Zum Beispiel die Sonne oben rechts im Van Gogh-Bild könnte auch an anderer Stelle auftauchen, ohne den Stil zu verändern.

Verwende die Gram-Matrix (Korrelationsmatrix) als Repräsentant für den Zeichenstil.

$$\hat{G}^{\ell}_{f,f'} = \sum_j \hat{a}^{\ell}_{f,j}\hat{a}^{\ell}_{f',j} = \left\langle \hat{a}^{\ell}_{f,:}, \hat{a}^{\ell}_{f',:} \right\rangle .$$

Die entsprechende Abweichung auf Schicht ℓ:

$$\mathcal{L}_{\text{style}}(x_{\text{style}}, \hat{x}, \ell) = \frac{1}{4(F^{\ell}M^{\ell})^2} \sum_{f,f'=1}^{F^{\ell}} (G^{\ell}_{\text{style},f,f'} - \hat{G}^{\ell}_{f,f'})^2$$

mit F^{ℓ}: Anzahl der Features in Schicht ℓ, $M^{\ell} = H^{\ell} \cdot W^{\ell}$: Höhe $\times$ Breite in Schicht ℓ.

Insgesamt versucht man, die Abweichungen bezüglich content und style gleichzeitig zu minimieren.

$$\mathcal{L}_{\text{total}}(x_{\text{content}}, x_{\text{style}}, \hat{x}) = \alpha\mathcal{L}_{\text{content}} + \beta\mathcal{L}_{\text{style}},$$

wobei

$$\mathcal{L}_{\text{style}} = \sum_{\ell=0}^{L} w_{\ell}\mathcal{L}_{\text{style}}(x_{\text{style}}, \hat{x}, \ell)$$

und $w_1 = w_2 = \ldots = \frac{1}{5}$. Für $\mathcal{L}_{\text{content}}$ verwendet man nur Layer 4.

$\alpha/\beta \approx 10^{-3}$

$$\hat{x}' = \hat{x} - \lambda\frac{\partial\mathcal{L}_{\text{total}}}{\partial\hat{x}}.$$

Zusammenfassung

$\mathcal{L}_{\text{style}}$ hängt direkt von den Gram-Matrizen ab, die aus der Ausgabe der Layer 1 bis 5 berechnet werden.

$\mathcal{L}_{\text{content}}$ hängt direkt nur von der Ausgabe von Layer 4 ab.

Abstrakte Darstellung:

$$\mathcal{L}_{\text{total}} = \alpha\mathcal{L}_{\text{content}} + \beta\mathcal{L}_{\text{style}}.$$

$\hat{x}$ wird optimiert, um $\mathcal{L}_{\text{total}}$ zu minimieren. Man benötigt den Gradient

$$\frac{\partial\mathcal{L}_{\text{total}}}{\partial\hat{x}},$$

dazu kann man wieder Backpropagation anwenden.

Fast Style Transfer

J. Johnson, A. Alahi, L. Fei-Fei: Perpetual losses for real-time style transfer and super-resolution (2016)

Idee: Trainiere ein CNN N_{FST} zur direkten Berechnung von $\hat{x}$ (mit vorgegebenem x_{style}).

x_{in} ist Eingabe von N_{FST}, $\hat{x}$ die gewünschte Ausgabe. Setze $x_{\text{in}} = x_{\text{content}}$ (Foto) und verwende $\mathcal{L}_{\text{total}}$ als Kostenfunktion zum Trainieren von N_{FST}.

Nach abgeschlossenem Training kann der Image Style Transfer

$$x_{\text{content}} \xrightarrow{N_{\text{FST}}} \hat{x}$$

in einem einzigen Feedforward-Pass durchgeführt werden, wobei x_{style} festgehalten wird.

"Real-time"-Anwendung auf Video-Streams (Webcam) möglich. N_{FST} besteht aus convolutional layers und fünf "residual blocks" (vgl. "ResNet" in Vorlesungsfolien)

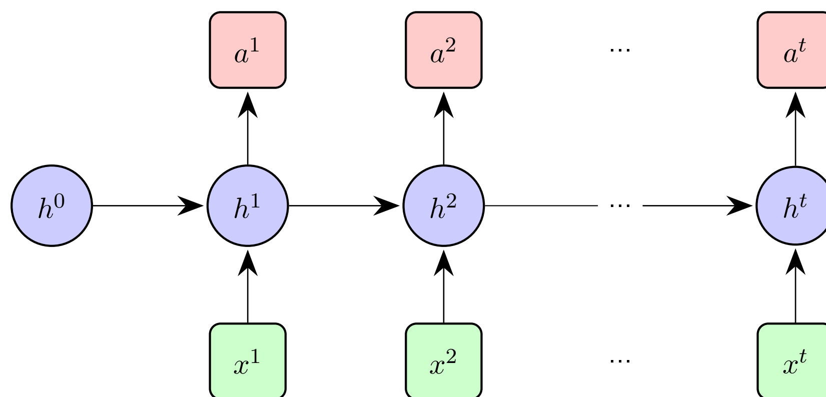
4 Recurrent Neural Networks

4.1 Grundlegende Architektur

Bisher: Feedforward-Netze, die Eingabe x wird vorwärts durch die Layer l propagiert, bis man in der Schicht L die Ausgabe erhält.

Problem: Diese Architektur ist nicht angepasst auf eine (zeitliche) Sequenz als Eingabe. Zum Beispiel haben die Wörter in einem zu übersetzenden Text einen komplexen Zusammenhang. Weitere Beispiele sind die Buchstaben bei der Wortvervollständigung, Video-Streams usw.

Für recurrent neural networks (RNN) modifiziert man die Architektur:



h ist der *hidden state* der Zelle.

Die interne Gewichtsmatrix und die Bias-Werte einer Zelle bleiben für verschiedene Zeitschritte konstant, nur die Ein- und Ausgabe ändern sich.

Anschauliche Interpretation des hidden state h^t: "Zusammengefasste Information" über die Input-Sequenz bis zum aktuellen Zeitpunkt t. Also hängt h^t von h^{t-1} und der aktuellen Eingabe x^t ab,

$$h^t = f(h^{t-1}, x^t).$$

Basisvariante eines RNN: "vanilla RNN" bzw. Elman-RNN:

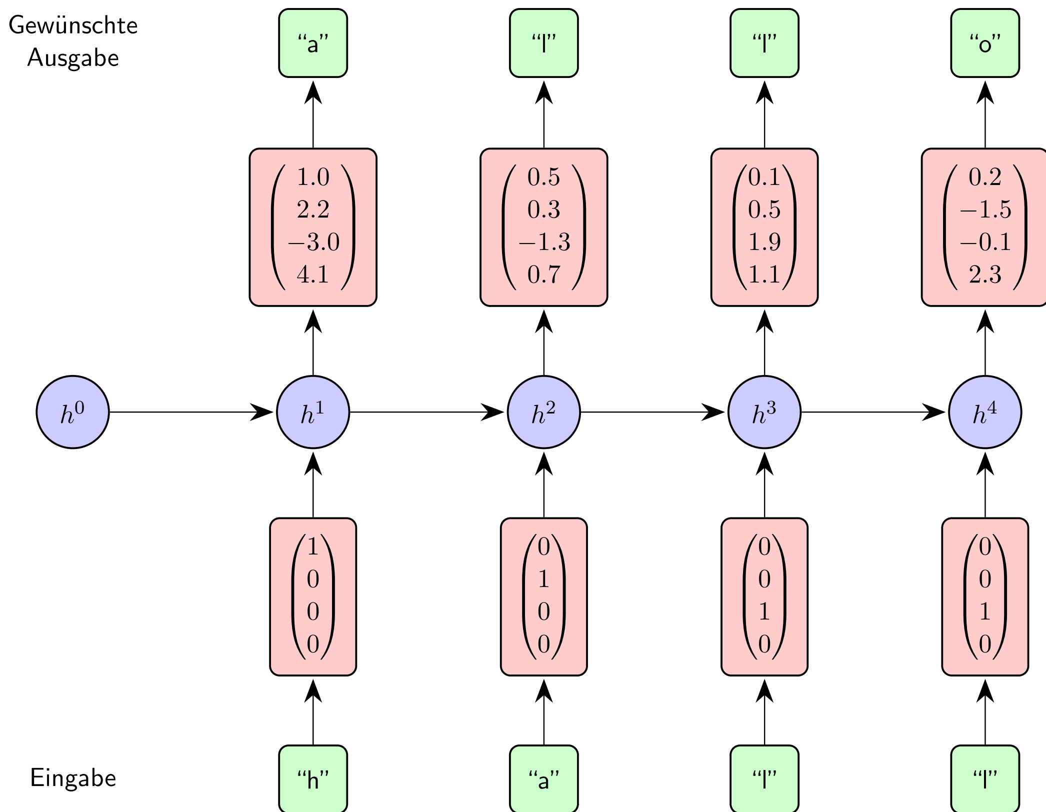
$$h^t = \tanh(W \cdot h^{t-1} + U \cdot x^t + b),$$

wobei tanh komponentenweise angewendet wird. $W \cdot h^{t-1}$ und $U \cdot x^t$ sind Matrix-Vektor-Multiplikationen. Die Ausgabe wird dann berechnet mittels

$$a^t = V \cdot h^t.$$

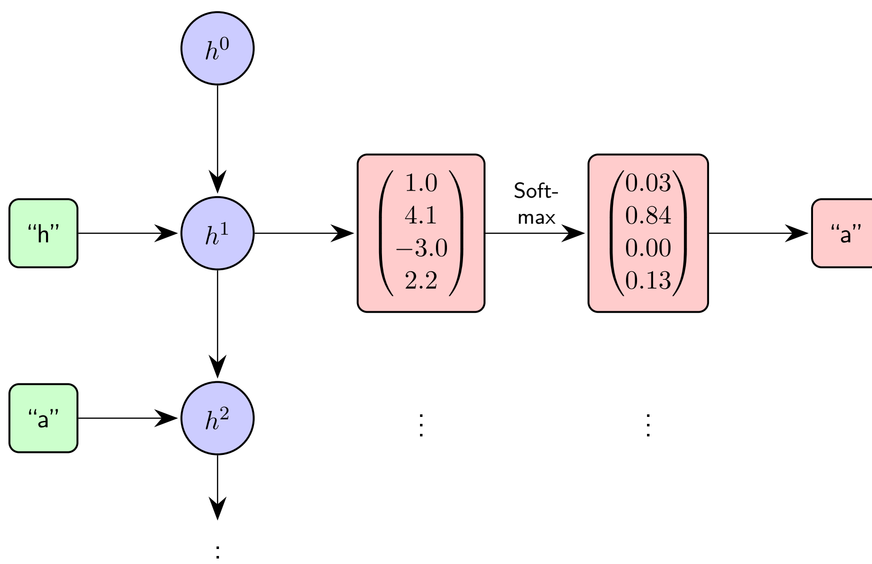
Beispielanwendung: Wortvervollständigung. Vokabular: $\{h, a, l, o\}$.

Trainieren:



Die Ausgabe des Netzwerks soll also immer für den nächsten Buchstaben des Wortes "hallo" am größten sein.

Testen mit zusätzlicher Softmax-Funktion $\frac{e^{a_i}}{\sum_j e^{a_j}}$ liefert eine Wahrscheinlichkeitsverteilung:



Wenn man die Ausgabe immer wieder als neue Eingabe verwendet, generiert das Netzwerk eine Buchstaben-Sequenz.

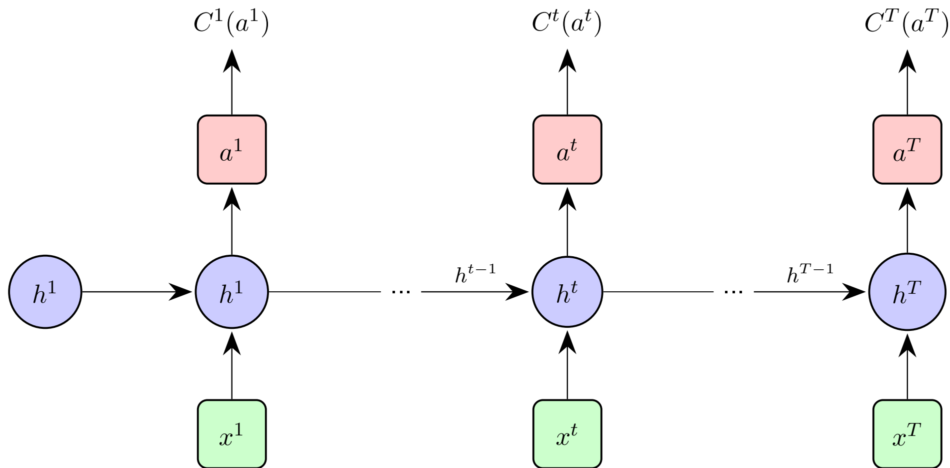
4.2 Backpropagation Through Time

(BPTT)

Mathematik: Gewöhnliche Berechnung des Gradienten (mittels Kettenregel).

Vanilla-RNN:

$$\begin{aligned} h^t &= \tanh(Wh^{-1} + Ux^t + b), \\ a^t &= Vh^t. \end{aligned}$$



Volle Kostenfunktion:

$$C = \sum_{t=1}^{T} C^t(a^t).$$

In einer online-Anwendung muss es kein größtes t geben, die Eingabe kann theoretisch unendlich lang sein. Dann teilt man die Daten in disjunkte Blöcke der Größe T ein.

$$\frac{\partial C}{\partial h_k^t} = \sum_{k'} \frac{\partial C}{\partial h_{k'}^{t+1}} \cdot \frac{\partial h_{k'}^{t+1}}{\partial h_k^t} + \sum_j \frac{\partial C}{\partial a_j^t} \cdot \underbrace{\frac{\partial a_j^t}{\partial h_k^t}}_{V_{jk}}.$$

Es gilt

$$h^t = \tanh(wh^{t-1} + Ux^t + b).$$

und

$$\tanh'(x) = \frac{1}{\cosh(x)^2} = 1 - \tanh(x)^2,$$

somit

$$\frac{\partial h_{k'}^{t+1}}{\partial h_k^t} = \tanh'((wh^t + Ux^{t-1} + b)_{k'}) = 1 - (h_{k'}^{t+1})^2.$$

Also folgt

$$\begin{aligned}\frac{\partial C}{\partial h_k^t} &= \sum_{k'} \frac{\partial C}{\partial h_{k'}^{t+1}} \cdot \frac{\partial h_{k'}^{t+1}}{\partial h_k^t} + \sum_j \frac{\partial C}{\partial a_j^t} \cdot V_{jk} \\ &= \sum_{k'} \frac{\partial C}{\partial h_{k'}^{t+1}} \tanh'((wh^t + Ux^{t-1} + b)_{k'}) w_{k'k} + \sum_j \frac{\partial C}{\partial a_j^t} \cdot V_{jk} \\ &= \sum_{k'} \frac{\partial C}{\partial h_{k'}^{t+1}(1-(h_{k'}^{t+1})^2) w_{k'k}} + \sum_j \frac{\partial C}{\partial a_j^t} \cdot V_{jk}.\end{aligned}$$

Der erste Term entfällt für $t = T$.

In Matrix-Vektor-Schreibweise:

$$\nabla_{h^t} C = w^\top \operatorname{diag}(1 - (h^{t+1})^2) \cdot (\nabla_{h^{t+1}} C) + V^\top (\nabla_{a^t} C).$$

$$\frac{\partial C}{\partial x_j^t} = \sum_k \frac{\partial C}{\partial h_k^t} \cdot \frac{\partial h_k^t}{\partial x_j^t} = \sum_k \frac{\partial C}{\partial h_k^t} (1 - (h_k^t)^2) U_{kj}.$$

Zur Berechnung des Gradienten bezüglich Gewichtsmatrizen und Bias-Werten verwende die Hilfsvariablen w^t, U^t, V^t, b^t mit demselben Wert wie w, U, V, b, aber nur zum Zeitpunkt t verwendet.

$$\frac{\partial C}{\partial b_k} = \sum_{t=1}^{T} \frac{\partial C}{\partial h_k^t} \cdot \frac{\partial h_k^t}{\partial b_k^t} = \sum_{t=1}^{T} \frac{\partial C}{\partial h_k^t} (1 - (h_k^t)^2).$$

Matrix-Vektor-Schreibweise:

$$(\nabla_b C) = \sum_{t=1}^{T} \operatorname{diag}(1 - (h^t)^2) \cdot (\nabla_{h^t} C).$$

$$\frac{\partial C}{\partial V_{jk}} = \sum_{t=1}^{T} \frac{\partial C}{\partial a_j^t} \cdot \underbrace{\frac{\partial a_j^t}{\partial V_{jk}}}_{h_k^t} = \sum_{t=1}^{T} \frac{\partial C}{\partial a_j^t} h_k^t.$$

$$\frac{\partial C}{\partial w_{kk'}} = \sum_{t=1}^{T} \frac{\partial C}{\partial h_k^t} \cdot \frac{\partial h_k^t}{\partial w_{kk'}} = \sum_{t=1}^{T} \frac{\partial C}{\partial h_k^t} (1 - (h_k^t)^2) h_{k'}^{t-1}$$

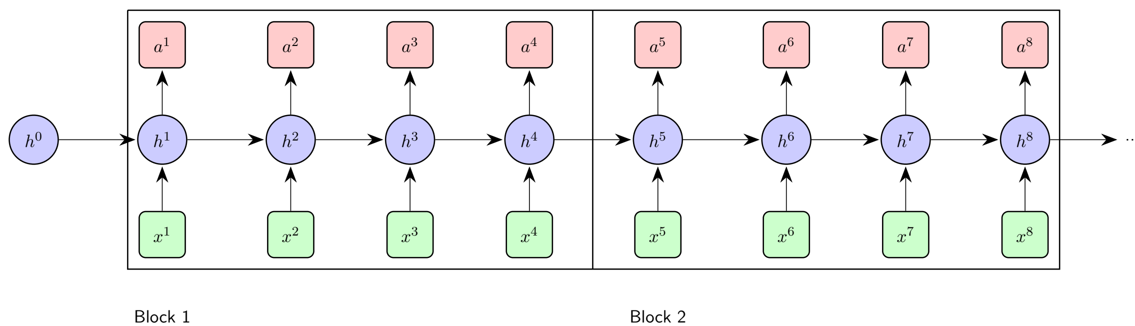
Analog:

$$\frac{\partial}{\partial x_i} \sum_{j=1}^{n} x_j y_j = y_i.$$

$$\frac{\partial C}{\partial U_{kj}} = \sum_{t=1}^{T} \frac{\partial C}{\partial h_k^t} \cdot \frac{\partial h_k^t}{\partial U_{kj}^t} = \sum_{t=1}^{T} \frac{\partial C}{\partial h_k^t} (1 - (h_k^t)^2) x_j^t.$$

Bemerkung. BPTT wird in der Praxis auf Sequenz-Blöcke angewendet ("truncated BPTT").

Beispiel für Blöcke der Größe 4:

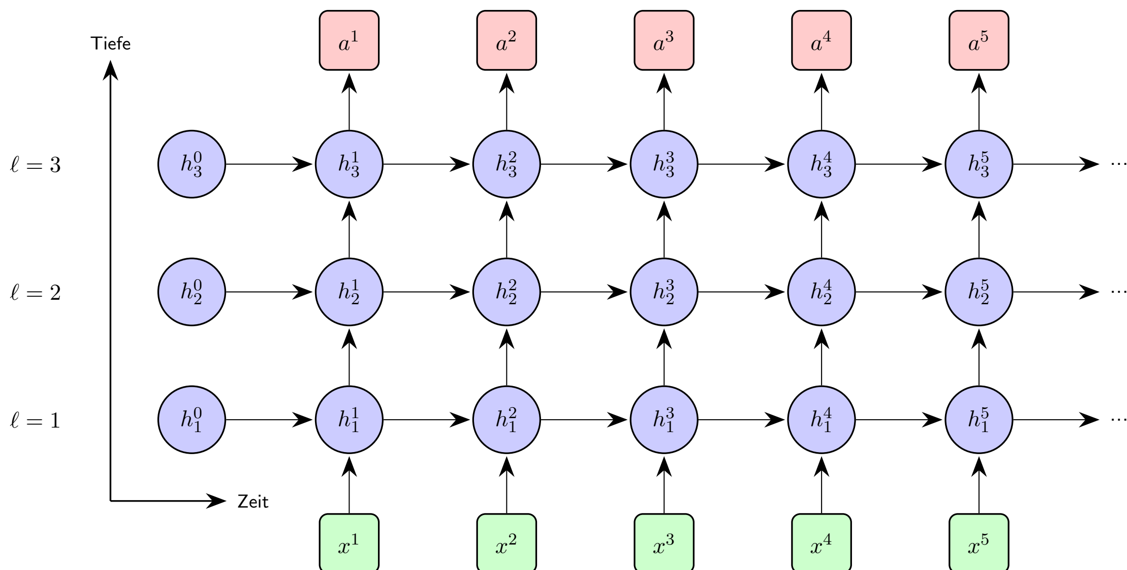


Der hidden state wird von einem Block zum nächsten weiter verwendet.

4.3 Deep Recurrent Neural Networks

Idee: Staple mehrere RNNs übereinander.

Beispiel mit Tiefe = 3:



Die BPTT verläuft analog zum bisherigen Verfahren (Tiefe = 1). Zunächst wendet man sie auf Schicht $l = L$ an, dann auf $l = L - 1$ usw.

4.4 Long Short-Term Memory Networks

LSTM-Networks

Schwäche von Vanilla-RNNs: Langzeitabhängigkeiten sind schwer erfassbar.

Zum Beispiel Spracherkennung: "to" und "two" klingen ähnlich, Kontext ist wichtig. "two people" oder "to infinity and beyond".

Kontextabhängigkeit mit großem Abstand:

> "Ich bin in Großbritannien aufgewachsen und zur Schule gegangen, deshalb spreche ich fließend {Englisch, Französisch, Spanisch, ... }".

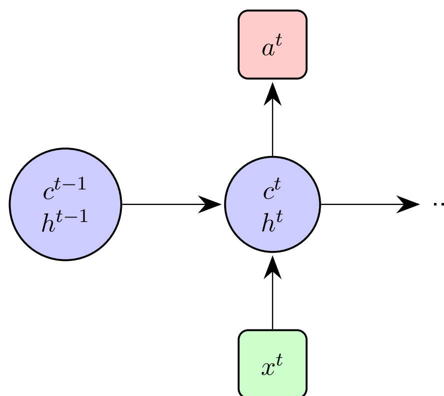
Für Menschen ist es einfach zu erkennen, dass "Englisch" das wahrscheinlichste Wort ist. Eine Maschine muss einen Kontextbezug über einen Abstand von mindestens zehn Wörtern erfassen können.

Dieser Zusammenhang ist von Vanilla-RNNS nur schwer erkennbar. Hintergrund sind die instabilen Gradienten analog zu den Deep Networks, vergleich Übungsaufgabe 26.

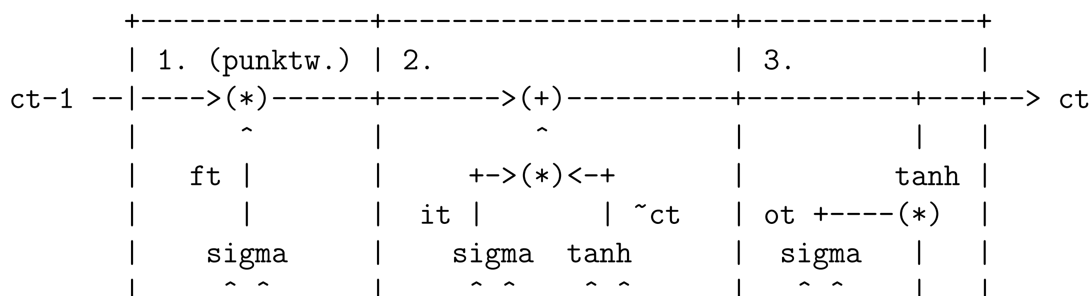
Ausweg: Sepp Hochreiter, Jürgen Schmidhuber, "Long Short-Term Memory", 1997.

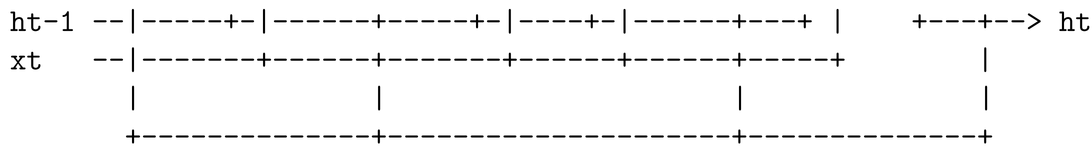
Schlüsselrolle für Handschrifterkennung, Sprachübersetzung und Spracherkennung (OK Google, Apple Siri, Amazon Alexa, usw.)

Idee: Zusätzlicher "cell state" c^t, der über viele Zeitschritte erhalten bleiben kann. Selektive Weitergabe von Information mittels drei "gates" ⇝ "Mini-Netzwerk" innerhalb einer LSTM-Zelle.



Das heißt, die Zelle hat die beiden internen Zustände c^t und h^t, die an die nächste Zelle weitergegeben werden.





1. "forget gate" f^t, Interpretation: f^t selektiert, welche Einträge des cell states vergessen werden sollen.

   Beispiel: Grammatikalisches Geschlecht ändert sich:

   > "Der Baum steht vor dem Haus, *das* ..."

   $$f^t = \sigma(w^{tF})h^{t-1} + U^{\text{forget}}x^t + b^{\text{forget}},$$

   wobei die Sigmoid-Funktion σ wieder komponentenweise angewendet wird.

   $$c^t = f^t \cdot c^{t-1} + \dots$$

   Extremfall: Alle Einträge von f^t sind (fast) null $\rightsquigarrow$ Vorheriger cell state c^{t-1} wird vollständig vergessen.

2. "input gate"

   $$i^t = \sigma(w^{\text{content}}h^{t-1} + u^{\text{input}}x^t + b^{\text{input}})$$

   und neue Daten für den cell state

   $$\tilde{c}^t = \tanh(w^{tCl}h^{t-1} + u^{\text{cell}}x^t + b^{\text{cell}}).$$

   Der neue cell state ist

   $$c^{\text{cell}} = f^t c^{t-1} + i^t \tilde{c}^t.$$

3. "output gate" und neuer hidden state

   $$o^t = \sigma(w^{\text{output}}h^{t-1} + u^{\text{output}}x^t + b^{\text{output}}),$$

   sowie

   $$h^t = o^t \cdot \tanh(c^t),$$

   wobei $\cdot$ punktweise ausgeführt wird.

Warum ist das Problem verschwindender Gradienten in LSTM-Netzwerken abgeschwächt?

$$\frac{\partial C}{\partial c_j^{t-1}} = \frac{\partial C}{\partial c_j^t} \underbrace{\frac{\partial c_j^t}{\partial c_j^{t-1}}}_{f_j} + \sum_k \frac{\partial C}{\partial h_k^t} \frac{\partial h_k^t}{\partial c_j^{t-1}}.$$

Beachte: f_k^t i_k^t und $\tilde{c}_k^t$ hängen nicht von c_k^{t-1} ab, also

$$\frac{\partial h_k^t}{\partial c_j^{t-1}} = \frac{\partial}{\partial c_j^{t-1}}(o_k^t \cdot \tanh(f_k^t c_k^{t-1} + i_k^t \tilde{c}_k^t)) = \delta_{jk} o_k^t (1 - \tanh(c_k^t)^2) \cdot f_k^t$$

und damit folgt

$$\frac{\partial C}{\partial c_j^{t-1}} = \frac{\partial C}{\partial c_j^t} f_j^t + \frac{\partial C}{\partial h_j^t} o_j^t f_j^t (1 - \tanh(c_j^t)^2).$$

Falls der erste Summand ≈ 1 und der der zweite ≈ 0 sind, wird der Gradient der Kostenfunktion (fast) unverändert zurück propagiert.

Bemerkung.

- Alle Gewichtsmatrizen und Bias-Werte werden "trainiert", das heißt nicht fest vorgegeben.
- Gewichtsmatrizen werden oft in eine größeren Matrix zusammengefasst:

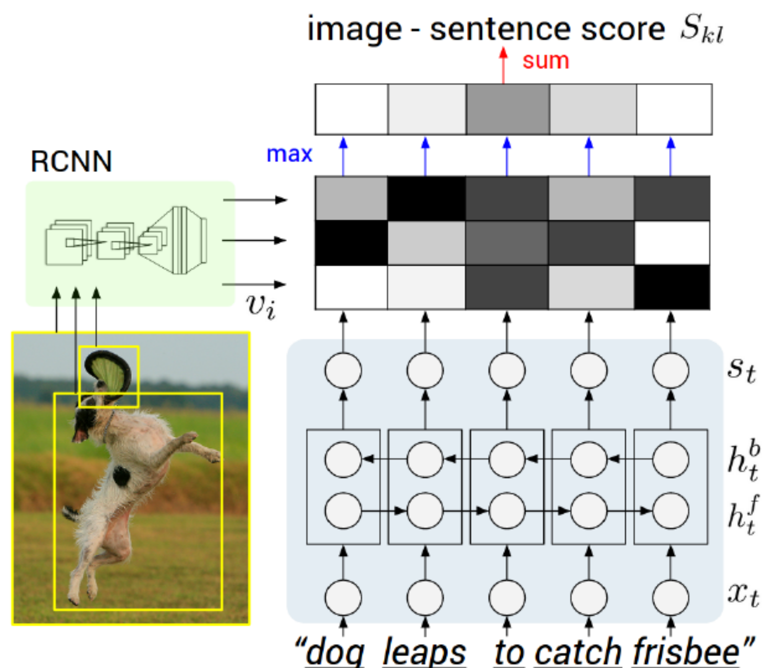
$$w = \begin{pmatrix} w^{\text{forget}} \\ w^{\text{input}} \\ w^{\text{output}} \\ w^{\text{content}} \end{pmatrix}, \quad U = \begin{pmatrix} U^{\text{forget}} \\ U^{\text{input}} \\ U^{\text{output}} \\ U^{\text{content}} \end{pmatrix}, \quad \Rightarrow \quad \begin{pmatrix} w^{\text{forget}} h^{t-1} \\ w^{\text{input}} h^{t-1} \\ w^{\text{output}} h^{t-1} \\ w^{\text{content}} h^{t-1} \end{pmatrix} = w h^{t-1},$$

  analog für Ux^t.
- Deep LSTM-Netzwerke sind völlig analog zu Deep RNNs.
- In der Literatur sind leicht verschiedene Varianten von LSTMs gebräuchlich.

4.4.1 Anwendung: Computergenerierte Bildbeschriftung

A Karpathy, Li Fei-Fei: Deep visual-semantic alignments for generating image descriptions, 2016.

O. Vinyals, A. Toshev, S. Bengio, D. Erhan: Show and tell: Lessons learned from the 2015 mscoco image captioning challenge, 2017.



Idee: Verwende (affin transformierte) Ausgabe eines CNN (bzw. die Aktivierung in einer tiefen Schicht) als ersten "hidden state" eines RNN.

Übersicht 4.1 noch einfügen

Testen bzw. Anwenden: Verwende Ausgabe eines Zeitschritts wieder als Eingabe zum nächsten Schritt:

$$a(\text{"Hund"}) = e_4 = \text{"fängt"}, \qquad a(\text{"fängt"}) = e_5 = \text{"Ball"}, \qquad a(\text{"Ball"}) = \texttt{END}.$$

4.4.2 Erweiterte Version: Bildbeschriftung zusammen mit Aufmerksamkeitsregionen

K. Xu et al.: Show, Attend and Tell: Neural image caption generation with visual attention. `https://arxiv.org/abs/1502.03044`

Idee: RNN wählt zusätzlich Aufmerksamkeitsregionen innerhalb a^{ℓ}_{CNN} während der Beschriftungsgenerierung. Die Wörter werden zusätzlich lokalisiert. Zum Beispiel: "Ball" ist rechts oben im Bild.

Umsetzung: CNN-Ausgabe a^{ℓ}_{CNN} wird in R räumliche Unterregionen aufgeteilt. Teile die Ausgabe in jedem Feature entlang W und H in Bereiche r_j ein.

5 Deep Reinforcement Learning

Motivation: Zielführendes Agieren in einer komplexen nicht-deterministischen Umgebung.

Beispiel: Autonomes Fahren. Steuere Fahrzeug sicher und zügig von A nach B, "Umgebung" aus anderen Verkehrsteilnehmern sehr komplex und nicht deterministisch.

Toy-Model: Atari 2600 Computerspiele (1977-1982): Pac-Man, Pong usw.

Verwende einzelne Bildschirm-Frames und Punktestand eines Computerspiels als Eingabe. Das Programm soll die optimale Steuerung selbst herausfinden.

V. Minh, K. Karukcouglu, ..., D. Hassabis (Google Deep Mind): Human-level control through deep reinforcement learning. Nature, 2015.

Selbe Programm-Architektur auf 49 verschiedene Atari-Spiele angewendet. Teilweise werden menschliche Leistungen übertroffen.

Etwa ein Jahr später "AlphaGo"-Programm. D. Silver, A. Huang, ..., D. Hassabis: Mastering the game Go with deep neural networks and tree search. Nature, 2016.

AlphaGo lernt auf Basis einer Datenbank von Spielzügen, Go zu spielen und besiegt Go-Weltmeister.

AlphaGo Zero: D. Silver, ..., D. Hassabis: Mastering the game of Go without human knowledge. Nature, 2017.

Im Gegensatz zu AlphaGo startet AlphaGo Zero den Lernprozess ohne Vorwissen. Das Programm "lernt", indem es gegen sich selbst spielt. Lediglich die Spielregeln sind ihm zu Beginn bekannt.

5.1 Markov Decision Processes

MDP

Formale Beschreibung: Agent und Umgebung befinden sich zum Zeitpunkt $t = 0, 1, \ldots$ im Zustand $s_t \in \mathcal{S}$, wobei $\mathcal{S}$ ein endlicher Zustandsraum ist. Der Agent wählt die Aktion $a_t \in \mathcal{A}$ aus. Dadurch geht das System in den nächsten Zustand $s_{t+1} \in \mathcal{S}$ über. Die Übergangswahrscheinlichkeit ist

$$\mathbb{P}(s_{t+1}|s_t, a_t).$$

Markov-Eigenschaft: Diese Wahrscheinlichkeit hängt nur von t, nicht von früheren Zeitpunkten ab.

Zu jedem Zeitpunkt erhält der Agent eine "Belohnung" (reward)

$$r_t = R(s_t).$$

Sie kann positiv, negativ oder 0 sein.

Ziel ist die Maximierung der kumulativen, (möglicherweise) zeitlich abgeschwächten Belohnung (cumulative discounted reward), quantifiziert durch die "Utility" (Nutzen)

$$U(s_0, s_1, \ldots) = \sum_{t \geq 0} \gamma^t R(s_r), \qquad 0 < \gamma \leq 1.$$

Für $\gamma < 1$ wird die Belohnung zu späteren Zeitpunkten weniger wichtig.

Zusammengefasst wird der MDP durch das Tupel $(\mathcal{S}, \mathcal{A}, R, \mathbb{P}, \gamma)$ charakterisiert.

Modellbeispiel: 3×4-Spielfeld mit einer Spielfigur

$$\begin{array}{cccc} \square & \square & \square & E_{+1} \\ \square & \blacksquare & \square & E_{-1} \\ S & \square & \square & \square \end{array}$$

S ist der Startpunkt, $\blacksquare$ darf nicht betreten werden, $E_{\pm 1}$ sind Endpunkte, wobei ± 1 für die zugehörige Belohnung stehen.

Der Zustand s_t ist die aktuelle Position der Spielfigur. Mögliche Aktionen: $\uparrow$, $\downarrow$, $\rightarrow$, $\leftarrow$. "Versuche", ein Feld nach oben/unten/rechts/links zu gehen.

Übergangswahrscheinlichkeit für "$\uparrow$":

$$\begin{array}{ccccc} & & p = 0{,}8 & & \\ & & \uparrow & & \\ p = 0{,}1 & \leftarrow & S & \rightarrow & p = 0{,}1 \end{array}$$

Ist das ausgewählte Feld nicht zulässig, so bewegt sich die Figur in diesem Zug nicht.

Formal:

$$\begin{aligned} \mathbb{P}(s_{t+1} = s_t + \begin{pmatrix} 0 \\ 1 \end{pmatrix} | s_t, a_t = \uparrow) &= 0{,}8, \\ \mathbb{P}(s_{t+1} = s_t + \begin{pmatrix} 1 \\ 0 \end{pmatrix} | s_t, a_t = \uparrow) &= 0{,}1, \\ \mathbb{P}(s_{t+1} = s_t + \begin{pmatrix} -1 \\ 0 \end{pmatrix} | s_t, a_t = \uparrow) &= 0{,}1. \end{aligned}$$

Die Wahrscheinlichkeit, dass der Agent sich tatsächlich in die gewünschte Richtung bewegt, ist 80 %. Mit jeweils 10 % landet er im Feld mit dazu senkrechter Richtung.

Analog für $\downarrow$, $\rightarrow$ und $\leftarrow$.

Der Agent erhält den reward $+1$ für das Feld $(4, 3)$ und -1 für das finale Feld $(4, 2)$. Für jedes andere Feld ist der reward $-0{,}04$. Der Agent wird dadurch motiviert, das Ziel möglichst schnell zu erreichen.

Wie sieht die Verhaltensstrategie aus? Das Verhalten des Agenten wird durch die "policy" $\pi : \mathcal{S} \rightarrow \mathcal{A}$ spezifiziert,

$$a_t = \pi(s_t).$$

Wegen der Markov-Eigenschaft genügt es, das Verhalten als Funktion von s_t auszudrücken anstatt von $s_0, s_1, \ldots$.

Die “Qualität” von π ist der erwartete Nutzen für den Startzustand s_0:

$$U^\pi(s_0) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t)|\pi\right].$$

Notation $\mathbb{E}[\dots|\pi]$: Policy π wird für Wahl der Aktion a_t verwendet.

Bemerkung. $U^\pi(s_0)$ wird oft auch als “value function” (Bewertungsfunktion) bezeichnet.

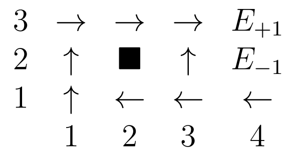
Optimale Policy

$$\pi^* = \arg\max_\pi U^\pi(s_0).$$

π^* hängt *nicht* von s_0 ab. Intuitiv: Falls zwei Trajektorien s_i und $\bar{s}_i$ zum Zeitpunkt t übereinstimmen, dann muss die optimale Verhaltensregel ab t die gleiche sein. Der restliche erwartete Nutzen ab t hat selbstähnliche Form:

$$\begin{aligned}\mathbb{E}\left[\sum_{t'=t}^{\infty} \gamma^{t'} R(s_{t'})|\pi\right] &= \gamma^t \mathbb{E}\left[\sum_{\delta t=0}^{\infty} \gamma^{\delta t} R(s_{t+\delta t})|\pi\right] \\ &= \gamma^t \mathbb{E}_{s_\tau} U^\pi(s_\tau).\end{aligned}$$

Beispiel.



$(1,3)$: “Nehme lieber einen Umweg, statt zu riskieren, auf $(2,4)$ zu landen.”

Entsprechende Bewertungsfunktion: $U = U^{\pi^*}$ (lasse Superskript π^* weg).

Man kann umgekehrt auch π^* aus U erhalten:

$$\pi^*(s) = \arg\max_{a\in\mathcal{A}} \sum_{s'} \mathbb{P}(s'|s,a)U(s').$$

“Wähle die Aktion, den erwarteten Nutzen maximiert.”

Bellman-Gleichung

Idee: Der erwartete Nutzen eines Zustands s ist gleich dem aktuellen Reward und erwarteten Nutzen des nächsten Zustand, bei optimalem Verhalten des Agenten.

Daraus erhält man die *Bellman-Gleichung*

$$U(s) = R(s) + \gamma \max_{a\in\mathcal{A}} \sum_{s'\in\mathcal{S}} \mathbb{P}(s'|s,a)U(s')$$

für alle $s \in \mathcal{S}$. Sie ist eindeutig lösbar für $\gamma < 1$, siehe Tutor-Aufgabe Blatt 13.

Value-Iteration-Algorithmus

Eine naheliegende Idee zur (numerischen) Lösung der Bellman-Gleichung. Iteration:

$$u_{i+1}(s) = R(s) + \gamma \max_{a \in \mathcal{A}} \sum_{s'} \mathbb{P}(s'|s,a) U_i(s')$$

für alle $s \in \mathcal{S}$.

Abstrakt:

$$U_{i+1} = B(U_i)$$

Der *Bellman-Schritt* B ist für $\Gamma < 1$ eine Kontraktion bezüglich der Maximumsnorm.

$$\|B(U_i) - B(U_i')\| \leq \gamma \|U_i - U_i'\|$$

mit $\|U_i\| = \max_{s \in \mathcal{S}} |U_i(s)|$.

Die Lösung $U = U^{\pi^*}$ ist der eindeutige Fixpunkt von B_i,

$$B(U) = U.$$

Somit folgt

$$\begin{aligned}
\|U_i - U\| &\leq \gamma \|U_{i-1} - U\| \\
&\leq \gamma^2 \|U_{i-2} - U\| \\
&\leq \dots \\
&\leq \gamma^i \|U_0 - U\|,
\end{aligned}$$

die Konvergenz ist exponentiell.

Man kann $\|U_i - U\|$ abschätzen, ohne U zu kennen:

$$\begin{aligned}
\|U_i - U\| &= \|U_i - U_{i+1} + U_{i+1} - U\| \\
&\leq \|U_{i+1} - U_i\| + \|U_{i+1} - U\| \\
&\leq \gamma \|U_i - U_{i-1}\| + \|U_{i+1} - U\| \\
&\leq \gamma \|U_i - U_{i-1}\| + \gamma \|U_{i+1} - U_i\| + \|U_{i+2} - U\| \\
&\leq \dots + \gamma^2 \|U_i - U_{i-1}\| + \|U_{i+2} - U\| \\
&\leq \dots \\
&\leq \|U_i - U_{i-1}\| \cdot \sum_{j=1}^{\infty} \gamma^j \\
&= \frac{\gamma}{1-\gamma} \cdot \|U_i - U_{i-1}\|.
\end{aligned}$$

Somit folgt: falls

$$\|U_i - U_{i-1}\| \leq \varepsilon \cdot \frac{1-\gamma}{\gamma},$$

dann ist

$$\|U_i - U\| < \varepsilon.$$

Damit kann man also ein Abbruchkriterium für die Iteration definieren.

Policy-Iteration-Algorithmus

Idee: Die "beste" Policy gemäß U_i, also

$$\pi_i(s) = \arg\max_{a\in\mathcal{A}} \sum_{s'} \mathbb{P}(s'|s,a)U_i(s'),$$

stimmt möglicherweise schon mit der optimalen Policy π^* überein, obwohl U_i noch von U^{π^*} abweicht.

Man kann also einen *Policy-Iteration-Algorithmus* durchführen.

Gegeben: Anfängliche Policy π_0, zum Beispiel zufällig bestimmt.

Für $i = 0, 1, 2, \ldots$

(a) Policy-Auswertung: Berechne (für alle $s \in \mathcal{S}$)

$$U_i = U^{\pi_i},$$

den erwarteten Nutzen, falls sich der Agent gemäß π_i verhält.

(b) Policy-Verbesserung: Verwende U_i zur Bestimmung einer neuen Policy π_{i+1},

$$\pi_{i+1}(s) = \arg\max_{a\in\mathcal{A}} \sum_{s'} \mathbb{P}(s'|s,a)U_i(s').$$

Die optimale Policy wurde gefunden, falls $U_{i+1} = U_i$.

Begründung: In diesem Fall ist U_i auch eindeutiger Fixpunkt der Value-Iteration.

Vorteil der Policy-Iteration im Vergleich zur Value-Iteration: In (a) ist für U_i ein *lineares* Gleichungssystem zu lösen,

$$U_i(s) = R(s) + \gamma \sum_{s'} \mathbb{P}(s'|s,\pi_i(s))U_i(s').$$

Die Kosten sind $\mathcal{O}(|\mathcal{S}|^3)$, wobei $|\mathcal{S}|$ die Anzahl aller Zustände ist. Die Lösung ist also nur durchführbar für $|\mathcal{S}|$ relativ klein.

Modifizierte Policy-Iteration: Vermeide $\mathcal{O}(|\mathcal{S}|^3)$ Kosten in (a) mit der Iteration

$$U_{i,j+1}(s) = R(s) + \gamma \sum_{s'} \mathbb{P}(s'|s,\pi_i(s))U_{i,j}(s'), \qquad j = 0, 1, \ldots, k-1.$$

Effizienzsteigerung durch Anwendung des Algorithmus auf eine Teilmenge aller Zustände. Idee: Man braucht keine genaue Bewertung für Zustände, die vermieden werden sollen.

5.2 Reinforcement Learning, Q-value-Funktion und Q-Learning

Problem des Value-Iteration- bzw. Policy-Iteration-Algorithmus: Die Auswertung aller möglichen Zustände $U_i(s) = \ldots$ für alle $s \in \mathcal{S}$ ist typischerweise nicht durchführbar. Der Zustandsraum ist zu groß, zum Beispiel bei Brettspielen:

$$|S| \approx b^d,$$

wobei S der Zustandsraum aller möglichen Brettpositionen ist, b die mittlere Anzahl der erlaubten Züge pro Position und d die Spieltiefe, die typische Anzahl an Zügen eines Spiels.

- Schach: $b \approx 35$, $d \approx 80$.
- Go: $b \approx 250$, $d \approx 150$.

Außerdem ist die Kenntnis der Übergangswahrscheinlichkeit $\mathbb{P}(s'|s,a)$ nötig. Diese ist aber in der Praxis oft unbekannt bzw. schwer abzuschätzen.

Zum Beispiel im autonomen Fahren: Die Wahrscheinlichkeit für eine bestimmte Verkehrssituation im nächsten Zeitschritt.

Stattdessen nutzt man folgende konzeptionelle Vorstellung beim Reinforcement Learning: Der Agent soll sich in (anfangs) völlig unbekannter Umgebung zurechtfinden, das heißt ohne Kenntnis der Übergangswahrscheinlichkeit $\mathbb{P}(s'|s,a)$ und der Reward-Funktion $R(s)$.

Q-value-Funktion

Idee: Bewerte die Aktion zusammen mit dem aktuellen Zustand.

$$Q^\pi(s,a) := \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma R(s_t) | s_0 = s, a_0 = a, \pi\right].$$

Q beschreibt also den erwarteten Nutzen, wenn der Agent im Zustand s die Aktion a wählt und ab dann die Policy π verfolgt.

Optimale Q-Funktion für die optimale Policy:

$$Q^*(s,a) = \max_\pi Q^\pi(s,a).$$

Vorteil: Die optimale Bewertungsfunktion U und die optimale Policy π^* folgen direkt aus Q^*:

$$U(s) = \max_a Q^*(s,a),$$
$$\pi^*(s) = \arg\max_a Q^*(s,a).$$

Dafür benötigt man keine Kenntnis der Übergangswahrscheinlichkeit $\mathbb{P}(s'|s,a)$, der Ansatz ist *modellfrei* (model free).

5.2.1 Bellman-Gleichung für die Q-value-Funktion

Analog zur Bellman-Gleichung für U:

$$\begin{aligned} Q^*(s,a) &= R(s) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s,a) \max_{a' \in \mathcal{A}} Q^*(s',a') \\ &= \mathbb{E}_{s'}\left[R(s) + \gamma \max_{a' \in \mathcal{A}} Q^*(s',a') | s,a \right], \end{aligned}$$

wobei $\mathbb{E}_{s'}[\cdot|s,a]$ der Erwartungswert bezüglich der Zufallsvariable $s' \sim \mathbb{P}(\cdot|s,a)$ ist.

5.2.2 Q-Learning-Algorithmus

C. J. Watkins, P. Dayan, 1992.

Idee zur numerischen Approximation von Q^*: Iteration $Q_i \to Q_{i+1}$, anstatt der (unbekannten) Übergangswahrscheinlichkeit $\mathbb{P}(s'|s,a)$ verwende den tatsächlich beobachteten Übergang $s_t \to s_{t+1}$.

Der Agent befindet sich im Zustand s_t und wählt die Aktion a_t. Er beobachtet den nächsten Zustand s_{t+1}.

Dann wird die Q-value-Funktion nur an der Stelle $(s,a) = (s_t,a_t)$ angepasst und bleibt sonst unverändert.

$$Q_{i+1}(s_t,a_t) = (1-\eta_i) Q_i(s_t,a_t) + \eta_i (R(s_t) + \gamma \max_{a'} Q_i(s_{t+1},a'))$$

mit der Lernrate $0 < \eta_i \leq 1$.

Das anfängliche $Q_0(s,a)$ wird als vorgegeben angenommen.

Watkins und Dayan beweisen die Konvergenz $Q_i \xrightarrow{i \to \infty} Q^*$ unter milden Voraussetzungen, zum Beispiel R beschränkt usw.

Aber: $Q_i(s,a)$ müsste für alle $s \in \mathcal{S}$ und $a \in \mathcal{A}$ abgespeichert werden.

5.3 Deep Q-Learning mit Experience Replay

Referenz: V. Mnih, ...: Human-level control through deep reinforcement learning. Nature, 2015.

Atari 2600 Computerspiele.

Das explizite Abspeichern von $Q^*(s,a)$ für alle $s \in \mathcal{S}$ und $a \in \mathcal{A}$ in einem Computerprogramm ist in vielen Fällen nicht möglich aufgrund des großen Zustandsraums $\mathcal{S}$.

Stattdessen approximiere

$$Q^*(s,a) \approx Q(s,a,\vartheta),$$

wobei ϑ ein beliebiger Parameter ist, zum Beispiel die Gewichte eines neuronalen Netzwerks (Q-Netzwerk).

Deep Q-Learning: Das Netzwerk ist ein *deep* neural network.

Ziel ist ein Algorithmus zur (approximativen) Lösung der Bellman-Gleichung

$$Q^*(s,a) = \mathbb{E}_{s'}\left[R(s) + \gamma \max_{a' \in \mathcal{A}} Q^*(s',a') | s,a\right]$$

mittels Iteration $\vartheta_{i-1} \to \vartheta_i$.

Idee: Approximiere

$$R(s) + \gamma \max_{a'} Q^*(s',a') \approx R(s) + \gamma \max_{a'} Q(s',a',\vartheta_{i-1}) =: Y(s')$$

und minimiere die Kostenfunktion

$$L_i(\vartheta_i) = \mathbb{E}_{s,a}\left[(E_{s'}[Y(s')|s,a] - Q(s,a,\vartheta_i))^2\right].$$

$E_{s'}[Y(s')|s,a]$ entspricht dem Term auf der rechten Seite der Bellman-Gleichung.

Erwartungswert $\mathbb{E}_{s,a}[\ldots]$: Bisher war die Aktion $a \in \mathcal{A}$ festgelegt durch die Policy π, das heißt $a = \pi(s)$. Jetzt ist die Wahrscheinlichkeitsverteilung von a noch offen gelassen.

Allgemein gilt für die Varianz einer Zufallsvariable Y:

$$\begin{aligned}
\operatorname{var}(Y) &:= \mathbb{E}[(Y - \mathbb{E}[Y])^2] \\
&= \mathbb{E}[Y^2 - 2Y\mathbb{E}[Y] + \mathbb{E}[Y]^2] \\
&= \mathbb{E}[Y^2] - 2\mathbb{E}[Y]\mathbb{E}[Y] + \mathbb{E}[Y]^2 \\
&= \mathbb{E}[Y]^2 - \mathbb{E}[Y^2],
\end{aligned}$$

insbesondere ist

$$\mathbb{E}[Y]^2 = \mathbb{E}[Y^2] - \operatorname{var}(Y).$$

Somit:

$$\begin{aligned}
L_i(\vartheta_i) &= \mathbb{E}_{s,a}\Big[\mathbb{E}_{s'}[(Y(s') - Q(s,a,\vartheta_i))^2 | s,a] - \operatorname{var}_{s'}(Y(s')|s,a)\Big] \\
&= \mathbb{E}_{s,a,s'}\left[(Y(s') - Q(s,a,\vartheta_i))^2\right] - \mathbb{E}_{s,a}\left[\operatorname{var}_{s'}(Y(s')|s,a)\right].
\end{aligned}$$

Der hintere Term hängt nicht von ϑ_i ab und kann damit für die Minimierung von L_i weggelassen werden. Also gilt

$$L_i(\vartheta_i)\mathbb{E}_{s,a,s'}\left[(Y(s') - Q(s,a,\vartheta_i))^2\right].$$

$\mathbb{E}_{s,a,s'}$ ist in der Praxis of nicht explizit auswertbar, da der Zustandsraum zu groß ist. Stattdessen approximieren wir den Erwartungswert zur Minimierung der Kostenfunktion durch einen stochastic gradient descent. Ein mini-batch besteht dabei aus Samples von früheren Spielen.

Wie beschreibt man bei Atari-Spielen den Zustand s_t? Eine naive Idee ist, den aktuellen Bildschirm-Frame zu benutzen, aber das verletzt die Markov-Eigenschaft. Zum Beispiel ist die Flugrichtung eines Balls aus einem Einzelbild nicht erkennbar, man benötigt weitere vorausgehende Frames.

Setze also

$$s_t = (x_1, a_1, x_2, a_2, \ldots, a_{t-1}, x_t),$$

wobei $x_{\tilde{t}}$ der Bildschirminhalt zum Zeitpunkt $\tilde{t}$ ist.

Insbesondere ist dadurch die Markov-Eigenschaft erfüllt. Die vorherigen Zustände $s_1, \ldots,$ s_{t-1} sind per Konstruktion bei gegebenem s_t festgelegt.

Für die praktische Umsetzung nehmen wir eine Vereinfachung durch die Preprocessing-Funktion ϕ vor:

- Original: Bildschirm-Dimension 210×160, 128 Farben. Konvertiere zu Graustufen-Bildern und skaliere auf 84×84 Pixel.
- Das Q-Netzwerk erhält nicht die gesamte Spiel-Historie, das heißt alle vorausgehenden Frames, sondern nur die letzten m Frames (konkret $m = 4$).

Insgesamt:

$$s_t : \left\{\begin{matrix}\text{Spielhistorie aus allen bisherigen} \\ 210 \times 160 \text{ Frames} \\ \text{entsprechenden Aktionen}\end{matrix}\right\} \xrightarrow{\phi} \left\{\begin{matrix}\text{Letzte } m - 4 \text{ Frames} \\ \text{als } 84 \times 84\text{-Graustufenbilder}\end{matrix}\right\}$$

Q-Netzwerk-Architektur

$Q(\phi(s), a, \vartheta)$ mit ϑ: Netzwerk-Parameter (Gewichte und Bias-Werte).

CNN, bestehend aus drei conv+ReLU und zwei dense layers

Bild 5.1

Jede Zelle im letzten Layer steht für eine mögliche Aktion. Die Anzahl ist vom Spiel abhängig.

Das Netzwerk berechnet $Q(\phi(s), a, \vartheta)$ für alle möglichen Aktionen a in *einem* Feedforward-Pass.

Training mit Experience Replay

Ziel: Minimiere die Kostenfunktion mittels stochastic gradient descent (oder Varianten davon) angewendet auf die Netzwerk-Parameter θ_i.

Naive Vorgehensweise zu Generierung von Trainings-Samples: Verwende vorherige Abschnitte aus dem aktuellen Spieldurchlauf. Dieser Ansatz ist aber problematisch, da die Samples somit stark korreliert sind. Das führt zu ineffektivem Training, da de Varianz sehr klein wird.

Außerdem beeinflusst das aktuelle Verhaltensmuster (z.B. "Bewege den Balken nach rechts") die weiteren Samples ("Balken ist auf der rechten Seite"). Man nennt dieses Verfahren *on-policy-training*.

Es kann unerwünschtes Feedback auftreten, die berechnete Q-value-Funktion schaukelt sich künstlich auf.

Stattdessen verwende *off-policy-training*: Wähle (gleichverteilt) Samples für den mini-batch aus dem *replay memory* aus, das frühere Spieldurchläufe (Episoden) abspeichert.

Ein einzelner Eintrag im replay memory hat die Form

$$\big(\phi(s_t), a_t, r_t = R(s_t), \phi(s_{t+1})\big).$$

Das Modell “lernt aus früherer Erfahrung” (experience replay).

Die konkreten Hyperparameter: Die mini-batch-Größe ist 32 und die Anzahl der Einträge im replay memory ist 10^6.

Weiter Verbesserung: Stabilisierung der Zielfunktion $Y(s')$ durch “zeitliches Einfrieren”.

Erinnerung:

$$Y(s') = R(s) + \gamma \max_{a'} Q(s', a', \vartheta_{i-1}).$$

Referenzwert für die Kostenfunktion (entspricht tatsächlichem Label beim Trainieren eines Netzwerks zur Bildklassifizierung).

Y hängt nur von den vorherigen Parametern ϑ_{i-1} ab. Allerdings ändert sich ϑ_{i-1} in jedem Zeitschritt im SGD-Verfahren. Das kann zu künstlichem Aufschaukeln der berechneten Q-Value-Funktion führen.

Das kann man vermeiden, indem man die Parameter des Referenz-Netzwerks für $C = 10\,000$ Schritte unverändert lässt, statt immer wieder die Parameter des vorherigen Schritts zu verwenden. Man bezeichnet den festen Parameter mit ϑ_i^-.

Algorithmus

- Initialisiere das Q-Netzwerk mit zufälligen Gewichten ϑ, setze

$$\vartheta^- = \vartheta.$$

- `for` episode $= 1, ..., M$:
  - Initialisiere Sequenz $s_1 = (x_1)$ (Erster Frame)
  - Führe Pre-Processing $\varphi_1 = \Phi(s_1)$ aus
  - `for` t $= 1, 2, ..., T$:
    * Wähle zufällige Aktion mit W. eps aus, ansonsten

    $$a_t = \arg\max_a Q(\phi(s), a, \vartheta).$$

    (Beste Aktion gemäß aktuellem Q-Netzwerk)
    * Führe Aktion a_t (im Emulator) aus und erhalte den Reward $r_t = R(s_t)$.
    * Nächster Frame x_{t+1}, setze

    $$s_{t+1} = (s_t, a_t, x_{t+1}),$$

    führe Pre-Processing $\varphi_{t+1} = \Phi(s_{t+1})$ aus.
    * Speichere Übergang $(\varphi_t, a_t, r_t, \varphi_{t+1})$ im Replay-Memory D ab
    * Wähle (zufällig) mini-batch aus Übergängen $(\varphi_j, a_j, r_j, \varphi_{j+1})_{j=1,...,32}$ aus.[1]

[1] Damit realisiert man $\mathbb{E}_{s,a,s'}[\cdots]$ in der Formel für die Kostenfunktion

* Setze

$$y_j = \begin{cases} r_j, & \text{falls Spieldurchlauf (Episode)} \\ & \text{im Schritt } j+1 \text{ endet,} \\ r_j + \gamma \max_{a'} Q(\varphi_{j+1}, a', \vartheta^-), & \text{sonst.} \end{cases}$$

  $Q(\varphi_{j+1}, a', \vartheta^-)$ ist das Referenz-Q-Netzwerk mit dem früheren Parameter ϑ^-.

* Gradient-descent Schritt für die Kostenfunktion:

$$(y_j - Q(\varphi_j, a_j, \vartheta))^2$$

  angewendet auf den Netzwerkparameter ϑ.

* Nach C Schritten setze $\vartheta^- = \vartheta$.

Bemerkung.

- ε-Parameter erlaubt "Exploration": Probiere zufällige Aktion aus.

  Bei Atari-Spielen setzt man anfangs $\varepsilon = 1$, der Agent spielt also völlig zufällig. Dann geht man schrittweise zu $\varepsilon = 0{,}1$ über. Dieser Wert wird ab 10^6 Frames fixiert.

- Der Algorithmus funktioniert, obwohl die Aktionen oft erst nach vielen Zeitschritten zu einer Belohnung führen. Die "Planung" ist implizit in der Q-value-Funktion enthalten.

5.4 Deep Reinforcement Learning mit Monte-Carlo Tree Search

Referenz: D. Silver et al.: Mastering the game of Go without human knowledge. Nature (2017).

AlphaGo Zero.

Unterschied im Vergleich zu vorherigen Versionen von AlphaGo:

- Kein Trainieren anhand von menschlichen Spielen, sondern nur durch Spielen gegen sich selbst.
- Residual Network Architektur (vergleiche mit ResNet angewendet auf IMAGE-NET Klassifikationswettbewerb)
- Verbesserter Trainingsalgorithmus

Ansatz: Neuronales Netzwerk f_ϑ ist simultane Policy- und Bewertungsfunktion.

$$(p, \nu) = f_\vartheta(s).$$

- Zustand s: Aktuelle Brettposition und 7 vorherige Positionen
- ν: Geschätze Gewinnwahrscheinlichkeit für den aktuellen Spieler bei der Brettposition s.
- p: Wahrscheinlichkeitsverteilung über alle möglichen Aktionen (Züge) im Zustand s.

- Netzwerkparameter ϑ

Die Abbildung $s \xrightarrow{f_\vartheta} p$ entspricht einer Policy-Funktion. Der Unterschied zu vorher ist aber, dass eine Wahrscheinlichkeitsverteilung ausgegeben wird.

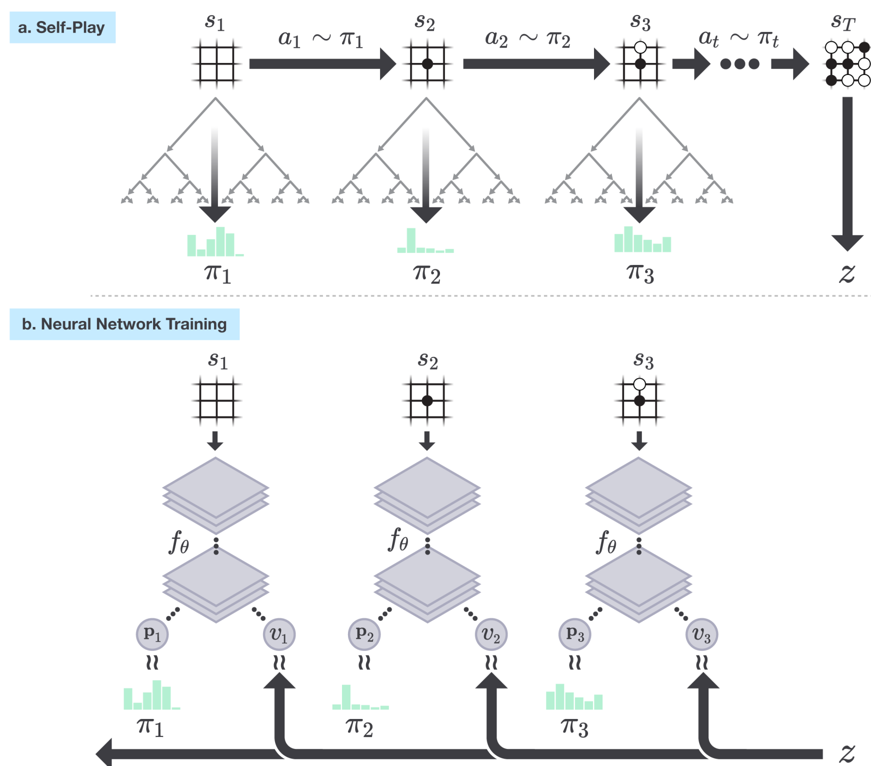
$s \xrightarrow{f_\vartheta} \nu$ entspricht der Bewertungsfunktion (vergleiche $U(s)$ in 5.1)

Idee: Ersetze Policy-Iteration (siehe 5.1.3) durch Monte-Carlo tree search ("Vorausberechnen" zukünftiger Züge) und Trainieren des Netzwerks f_ϑ.

(a) Das Programm spielt eine Partie gegen sich selbst mittels MCTS und f_ϑ.

Für die Brettpositionen $s_1, \ldots, s_T$ wird jeweils eine Zugvorschlagsverteilung π_t für die Position s_t und der Gewinner ermittelt.

(b) Trainiere f_ϑ anhand von π_t und ermitteltem Gewinner. Beginne mit dem verbesserten f_ϑ wieder bei (a).



Aus s_1 generiert man durch MCTS die Verteilung π_1 für den nächsten Zug und wählt $a_1 \sim \pi_1$ aus. Für den daraus entstehenden Zustand s_2 wiederholt man diesen Vorgang. Im Schritt T steht ein Gewinner fest, der mit $z \in \{\pm 1\}$ ausgedrückt wird. Dabei ist

$$z = \begin{cases} 1, & \text{Spiel gewonnen aus Sicht des aktuellen Spielers,} \\ -1, & \text{Spiel verloren aus Sicht des aktuellen Spielers.} \end{cases}$$

Das neurale Netzwerk wird dann so trainiert, dass

$$p_1 \approx \pi_1, \quad \nu_1 \approx z, \qquad \ldots \qquad p_{T-1} \approx \pi_{T-1}, \quad \nu_{T-1} \approx z.$$

Ein Trainings-Tupel hat die Form (s_t, π_t, z). Dabei entsprechen (π_t, z) den Referenzen (Labels) beim Trainieren eines Klassifikationsnetzwerks.

Kostenfunktion:

$$L = (z-\nu)^2 - \sum_\alpha \pi_\alpha \log(p_\alpha) + c\|\vartheta\|^2.$$

Es handelt sich also um eine quadratische Kostenfunktion für ν und eine Cross-Entropy-Kostenfunktion für p mit L2-Regularisierung.

Monte Carlo Tree Search

MCTS

Suchbaum:

Ideales Szenario: Stelle vollen Suchbaum auf. Jeder Endknoten (Blatt) ist Spielende-Zustand. Dann spielt der Agent perfekt.

Aber in der Praxis (bei Go, Schach etc.) ist das nicht durchführbar. Es gibt $\approx b^d$ Endknoten, wobei d die Spieltiefe ist. Go: $b \approx 250$, $b \approx 150$. Also ist b^d extrem groß.

Stattdessen (bei AlphaGo Zero) verwendet man f_ϑ als Leitfaden, um einen möglichst relevanten Ausschnitt des Suchbaums aufzustellen.

Jede Kante (s,a) speichert

- die a-priori-Wahrscheinlichkeitsverteilung $\mathbb{P}(s,a)$, berechnet durch Auswertung von f_ϑ:
$$(\mathbb{P}(s,\cdot), \nu(s)) = f_\vartheta(s).$$
- Zähler $N(s,a)$: Wie oft wurde die Kante im aktuellen Durchlauf der MCTS besucht?
- (approximative) Q-value-Funktion $Q(s,a)$.

Jeder Knoten speichert $\nu(s)$: Bewertungsfunktion des Netzwerks f_ϑ.

Einzelner Simulationsdurchlauf beim MCTS:

(i) Wähle “widersprechende” Zugfolge aus anhand von Maximierung von $Q+U$, mit

$$U(s,a) \propto \frac{\mathbb{P}(s,a)}{1+N(s,a)}.$$

U wird also umso kleiner, je größer $N(s,a)$ ist. Dadurch wird die Exploration gefördert.

Erhöhe den Zähler $N(s,a)$ für die Kanten, die in der Zugfolge enthalten sind, jeweils um 1.

(ii) Werte den neuen Knoten s' mittels f_ϑ aus und füge den Knoten zum Suchbaum hinzu.

Ersetze $\circ$ durch Auswertung $(\mathbb{P}(s',\cdot), \nu(s')) = f_\vartheta(s')$.

(iii) Update der Q-value-Funktion für besuchte Kanten der aktuellen Zugfolge.

$Q(s,a)$ speichert die durchschnittliche Bewertung im *Unterbaum* ab (s,a):

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{s'} \nu(s'),$$

wobei $\nu(s')$ die Bewertung des Netzwerks f_ϑ ist und s' der Knoten im Unterbaum.

Update im Beispiel oben:

Ausgabe des MCTS-Algorithmus

Gegeben: Startknoten s

Ausgabe: Wahrscheinlichkeitsverteilung π über alle möglichen Aktionen

$$\pi_a = \frac{N(s,a)^{1/\tau}}{\sum_{a'} N(s,a')^{1/\tau}}$$

mit der "Temperatur" τ.

Für $\tau \to 0$ gilt dann

$$\pi_a \to \delta_{a,a^*}$$

mit a^* dem Maximierer von $N(s,a)$.

Für $\tau \to \infty$ ist π eine Gleichverteilung.

Analogie zur Physik: Bei hoher Temperatur verteilt sich die Aktivität überall hin, bei niedriger Temperatur beschränkt sich die Aktivität auf wenige wichtige Orte.

5.5 Architektur von f_ϑ: Residual Network

Erinnerung: Problem der instabilen bzw. verschwindenden Gradienten bei tiefen Netzwerken, obwohl ein tiefes Netzwerk mindestens "genauso gut" funktionieren sollte.[2] Idee: Die Schichten berechnen die *Abweichung* von der Identitätsfunktion.

Ersetze $x \mapsto f(x)$ durch $x \mapsto g(x)$ mit

$$g(x) = f(x) - x.$$

Dann berechne $f(x) = g(x) + x$.

K. He et al.: Deep residual learning for image recognition, 2016.

In der Praxis nutzt man dann Residual-Blöcke aus mehreren Schichten, typischerweise mit verschiedenen Schichttypen und dann gilt

$$f(x) = g_L(g_{L-1}(\cdots g_1(x))) + x.$$

Damit wird das Trainieren von sehr tiefen (~ 100) Netzwerken möglich.

Bei AlphaGo Zero werden 39 Residual Blocks verwendet.

Aufbau eines Blocks:

[2] Man kann für zusätzliche Schichten einfach die Identitätsfunktion wählen.

Zusammenfassung

- Kapitel 1: Einführung
- Kapitel 2: Grundlagen
  - Aufbau, Aktivierungsfunktion
  - Architektur von “Dense” Feedforward-Netzen
  - Backpropagation
  - Cross-Entropy, Softmax-Kostenfunktion
  - (Vermeidung von) Overfitting, Regularisierung
- Kapitel 3: Convolutional Neural Networks
  - Aufbau: Rezeptives Feld, Filter-Kernel, channels, features
  - Backpropagation für CNN
  - Optimierte Implementierung der Faltung
  - Artistic Style Transfer
- Kapitel 4: RNN
  - Architektur
  - BPTT
  - Deep RNNs
  - LSTM
  - Bildbeschriftung
- Kapitel 5: Reinforcement Learning
  - Markov Decision Processes
  - Policy, Bewertungsfunktion (value function) U, Q-value-Funktion
  - Value-Iteration, Policy-Iteration
  - Bellman-Gleichung
  - Anwendungen: Atari, Go